Python

Real-World Data Science

CURATED COURSE

[PACKT]

Python

Real-World Data Science

A course in four modules

Unleash the power of Python and its robust data science capabilities
with your Course Guide Ankita Thakur

Learn to use powerful Python libraries for
effective data processing and analysis

To contact your Course Guide
Email: ankitat@packtpub.com

[PACKT]
BIRMINGHAM - MUMBAI

Meet Your Course Guide

Hello and welcome to this *Data Science with Python* course. You now have a clear pathway from learning Python core features right through to getting acquainted with the concepts and techniques of the data science field—all using Python!

This course has been planned and created for you by me Ankita Thakur – I am your Course Guide, and I am here to help you have a great journey along the pathways of learning that I have planned for you.

I've developed and created this course for you and you'll be seeing me through the whole journey, offering you my thoughts and ideas behind what you're going to learn next and why I recommend each step. I'll provide tests and quizzes to help you reflect on your learning, and code challenges that will be pitched just right for you through the course.

If you have any questions along the way, you can reach out to me over e-mail or telephone and I'll make sure you get everything from the course that we've planned – for you to start your career in the field of data science. Details of how to contact me are included on the first page of this course.

What's so cool about Data Science?

What is Data Science and why is there so much of buzz about this in the world? Is it of great importance? Well, the following sentence will answer all such questions:

> "*This hot new field promises to revolutionize industries from business to government, health care to academia.*"
>
> – *The New York Times*

The world is generating data at an increasing pace. Consumers, sensors, or scientific experiments emit data points every day. In finance, business, administration, and the natural or social sciences, working with data can make up a significant part of the job. Being able to efficiently work with small or large datasets has become a valuable skill. Also, we live in a world of connected things where tons of data is generated and it is humanly impossible to analyze all the incoming data and make decisions. Human decisions are increasingly replaced by decisions made by computers. Thanks to the field of **Data Science**!

Data science has penetrated deeply in our connected world and there is a growing demand in the market for people who not only understand data science algorithms thoroughly, but are also capable of programming these algorithms. A field that is at the intersection of many fields, including data mining, machine learning, and statistics, to name a few. This puts an immense burden on all levels of data scientists; from the one who is aspiring to become a data scientist and those who are currently practitioners in this field.

Treating these algorithms as a black box and using them in decision-making systems will lead to counterproductive results. With tons of algorithms and innumerable problems out there, it requires a good grasp of the underlying algorithms in order to choose the best one for any given problem.

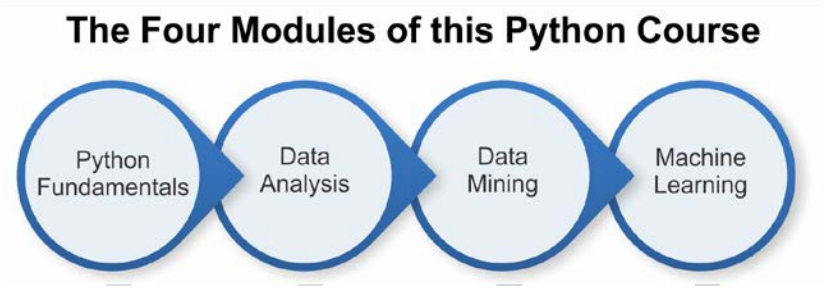
Python as a programming language has evolved over the years and today, it is the number one choice for a data scientist. Python has become the most popular programming language for data science because it allows us to forget about the tedious parts of programming and offers us an environment where we can quickly jot down our ideas and put concepts directly into action. It has been used in industry for a long time, but it has been popular among researchers as well.

In contrast to more specialized applications and environments, Python is not only about data analysis. The list of industrial-strength libraries for many general computing tasks is long, which makes working with data in Python even more compelling. Whether your data lives inside SQL or NoSQL databases or is out there on the Web and must be crawled or scraped first, the Python community has already developed packages for many of those tasks.

Course Structure

Frankly speaking, it's a wise decision to know the nitty-gritty of Python as it's a trending language. I'm sure you'll gain lot of knowledge through this course and be able to implement all those in practice. However, I want to highlight that the road ahead may be bumpy on occasions, and some topics may be more challenging than others, but I hope that you will embrace this opportunity and focus on the reward. Remember that we are on this journey together, and throughout this course, we will add many powerful techniques to your arsenal that will help us solve even the toughest problems the data-driven way.

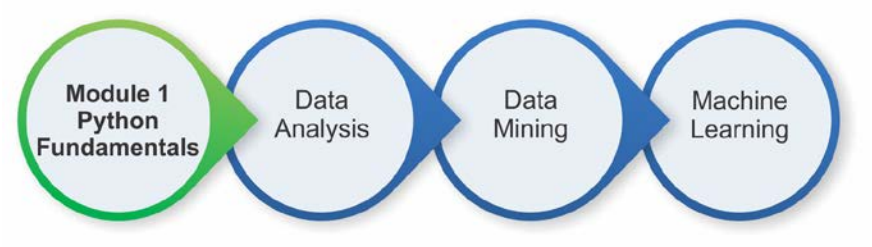
I've created this learning path for you that consist of four models. Each of these modules are a mini-course in their own way, and as you complete each one, you'll have gained key skills and be ready for the material in the next module.



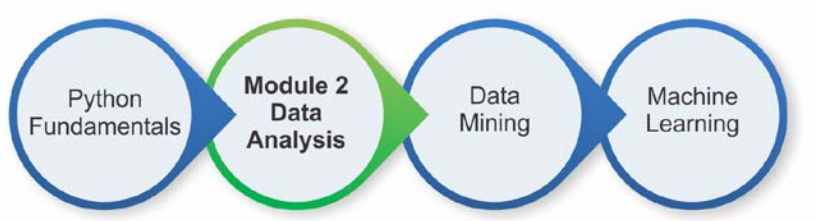
So let's now look at the pathway these modules create—basically all the topics that will be exploring in this learning journey.

Course Journey

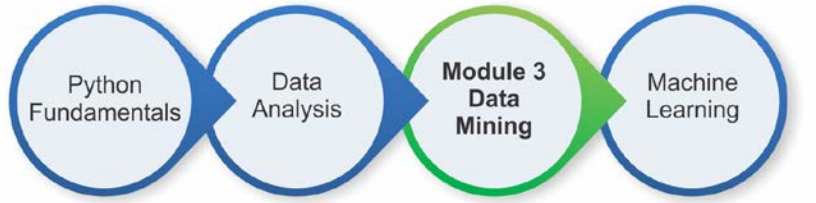
We start the course with our very first module, *Python Fundamentals*, to help you get familiar with Python. Installing Python correctly is equal to half job done. This module starts with the installation of Python, IPython, and all the necessary packages. Then, we'll see the fundamentals of object-oriented programming because Python itself is an object-oriented programming language. Finally, we'll make friends with some of the core concepts of Python—how to get Python programming basics nailed down.



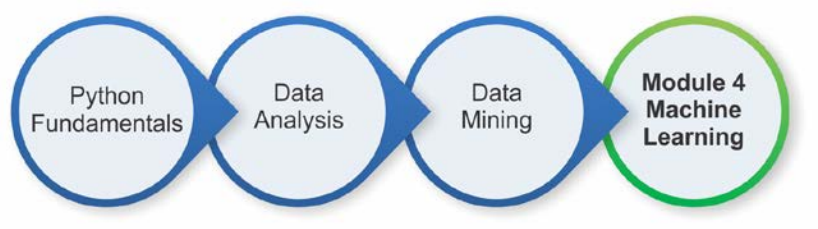
Then we'll move towards the analysis part. The second module, *Data Analysis*, will get you started with Python data analysis in a practical and example-driven way. You'll see how we can use Python libraries for effective data processing and analysis. So, if you want to to get started with basic data processing tasks or time series, then you can find lot of hands-on knowledge in the examples of this module.



The third module, *Data Mining*, is designed in a way that you have a good understanding of the basics, some best practices to jump into solving problems with data mining, and some pointers on the next steps you can take. Now, you can harness the power of Python to analyze data and create insightful predictive models.



Finally, we'll move towards exploring more advanced topics. Sometimes an analysis task is too complex to program by hand. Machine learning is a modern technique that enables computers to discover patterns and draw conclusions for themselves. The aim of our fourth module, *Machine Learning*, is to provide you with a module where we'll discuss the necessary details regarding machine learning concepts, offering intuitive yet informative explanations on how machine learning algorithms work, how to use them, and most importantly, how to avoid the common pitfalls. So, if you want to become a machine-learning practitioner, a better problem solver, or maybe even consider a career in machine learning research, I'm sure there is lot for you in this module!



Ankita Thakur

Your Course Guide

Did you know that data scientists in US earn about $118,000 per year?

Data Science is an emerging field and there is a high demand of people who can analyze the techniques of Data Science to get some useful information from the ever increasing data.

The Course Roadmap and Timeline

Here's a view of the entire course plan before we begin. This grid gives you a topic overview of the whole course and its modules, so you can see how we will move through particular phases of learning to use Python, what skills you'll be learning along the way, and what you can do with those skills at each point. I also offer you an estimate of the time you might want to take for each module, although a lot depends on your learning style how much you're able to give the course each week!

	Course Module 1	Course Module 2	Course Module 3	Course Module 4
Module	**Python Fundamentals**	**Data Analysis**	**Data Mining**	**Machine Learning**
Skills learned	OOPs concepts and the basic fundamentals of Python including proper installation steps	Use powerful Python libraries for effective data processing and analysis	Learn to design and develop data mining applications using a variety of datasets	Unlock deeper insights into machine learning with this vital module to cutting-edge predictive analysis
Key Topics	OOPs, Python basics, exception handling, Python data structures, and unit testing	Introduction to libraries used for data analysis, data visualization, working with date and time objects, interacting with databases, data analysis application examples	Affinity analysis, decision trees, Naive Bayes, and deep learning	Different machine learning models, perceptron learning algorithm, classification using scikit-learn, and dealing with missing data
Suggested Time per Module	**2 weeks** to familiarize yourself with core Python and OOPs concepts	**2 weeks** to get started with data analysis	**3 weeks** to broaden your data science knowledge	**2 weeks** to learn advanced machine learning models
Things you can do with Python by this point	Installing Python and making use of OOP features in your projects	Make use of the Python libraries for data analysis	Able to implement the specific techniques needed to work with large datasets	Able to find out answers to critical questions of your data

Table of Contents

Course Module 1: Python Fundamentals

Course Module 2: Data Analysis

Course Module 3: Data Mining

Course Module 4: Machine Learning

Course Module 1
Python Fundamentals

Course Module 1: Python Fundamentals
Chapter 1: Introduction and First Steps – Take a Deep Breath
Chapter 2: Object-oriented Design
Chapter 3: Objects in Python
Chapter 4: When Objects are alike
Chapter 5: Expecting the Unexpected
Chapter 6: When to use Object-oriented programming
Chapter 7: Python Data Structures
Chapter 8: Python Object-oriented Shortcuts
Chapter 9: Strings and Serialization
Chapter 10: The Iterator Pattern
Chapter 11: Python Design Patterns I
Chapter 12: Python Design Patterns II
Chapter 13: Testing Object-oriented Programs
Chapter 14: Concurrency

Let's begin...
Course Module 1
Python
Fundamentals
is ready for you

Course Module 2: Data Analysis
Chapter 1: Introducing Data Analysis and Libraries
Chapter 2: NumPy Arrays and Vectorized Computation
Chapter 3: Data Analysis with pandas
Chapter 4: Data Visualizaiton
Chapter 5: Time Series
Chapter 6: Interacting with Databases
Chapter 7: Data Analysis Application Examples

Course Module 3: Data Mining
Chapter 1: Getting Started with Data Mining
Chapter 2: Classifying with scikit-learn Estimators
Chapter 3: Predicting Sports Winners with Decision Trees
Chapter 4: Recommending Movies Using Affinity Analysis
Chapter 5: Extracting Features with Transformers
Chapter 6: Social Media Insight Using Naive Bayes
Chapter 7: Discovering Accounts to Follow Using Graph Mining
Chapter 8: Beating CAPTCHAs with Neural Networks
Chapter 9: Authorship Attribution
Chapter 10: Clustering News Articles
Chapter 11: Classifying Objects in Images Using Deep Learning
Chapter 12: Working with Big Data
Chapter 13: Next Steps…

Course Module 1

The very first module coming our way is the *Python Fundamentals*. Python is the most popular teaching language in top computer science universities. This module will lay the foundation of the entire course.

Ankita Thakur

Your Course Guide

It will also introduce the terminology of the object-oriented paradigm since Python has always been an object-oriented programming language. It will explain classes, data encapsulation, inheritance, polymorphism, abstraction, and exceptions with an emphasis on when we can use each principle to develop well-designed software. It will not only guide us to create maintainable applications by studying higher level design patterns but will also help us grasp the complexities of string and file manipulation, and how Python distinguishes between binary and textual data. As a bonus, we will also discover the joys of unit testing and the complexities of concurrent programming.

In short, we'll dive into the formal principles of object-oriented programming and how Python leverages them. Don't worry even if you're already familiar with object-oriented programming in other languages, this module will help you understand the idiomatic ways to apply your knowledge in the Python ecosystem. So, are you ready? Fasten your seat belts and let's start this exciting journey...

1

Introduction and First Steps – Take a Deep Breath

> "*Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime.*"
>
> *– Chinese proverb*

According to Wikipedia, **computer programming** is:

> "*...a process that leads from an original formulation of a computing problem to executable computer programs. Programming involves activities such as analysis, developing understanding, generating algorithms, verification of requirements of algorithms including their correctness and resources consumption, and implementation (commonly referred to as coding) of algorithms in a target programming language*".

In a nutshell, coding is telling a computer to do something using a language it understands.

Computers are very powerful tools, but unfortunately, they can't think for themselves. So they need to be told everything. They need to be told how to perform a task, how to evaluate a condition to decide which path to follow, how to handle data that comes from a device such as the network or a disk, and how to react when something unforeseen happens, say, something is broken or missing.

You can code in many different styles and languages. Is it hard? I would say "yes" and "no". It's a bit like writing. Everybody can learn how to write, and you can too. But what if you wanted to become a poet? Then writing alone is not enough. You have to acquire a whole other set of skills and this will take a longer and greater effort.

In the end, it all comes down to how far you want to go down the road. Coding is not just putting together some instructions that work. It is so much more!

Good code is short, fast, elegant, easy to read and understand, simple, easy to modify and extend, easy to scale and refactor, and easy to test. It takes time to be able to write code that has all these qualities at the same time, but the good news is that you're taking the first step towards it at this very moment by reading this module. And I have no doubt you can do it. Anyone can, in fact, we all program all the time, only we aren't aware of it.

Would you like an example?

Say you want to make instant coffee. You have to get a mug, the instant coffee jar, a teaspoon, water, and the kettle. Even if you're not aware of it, you're evaluating a lot of data. You're making sure that there is water in the kettle as well as the kettle is plugged-in, that the mug is clean, and that there is enough coffee in the jar. Then, you boil the water and maybe in the meantime you put some coffee in the mug. When the water is ready, you pour it into the cup, and stir.

So, how is this programming?

Well, we gathered resources (the kettle, coffee, water, teaspoon, and mug) and we verified some conditions on them (kettle is plugged-in, mug is clean, there is enough coffee). Then we started two actions (boiling the water and putting coffee in the mug), and when both of them were completed, we finally ended the procedure by pouring water in the mug and stirring.

Can you see it? I have just described the high-level functionality of a coffee program. It wasn't that hard because this is what the brain does all day long: evaluate conditions, decide to take actions, carry out tasks, repeat some of them, and stop at some point. Clean objects, put them back, and so on.

All you need now is to learn how to deconstruct all those actions you do automatically in real life so that a computer can actually make some sense of them. And you need to learn a language as well, to instruct it.

So this is what this module is for. I'll tell you how to do it and I'll try to do that by means of many simple but focused examples (my favorite kind).

A proper introduction

I love to make references to the real world when I teach coding; I believe they help people retain the concepts better. However, now is the time to be a bit more rigorous and see what coding is from a more technical perspective.

When we write code, we're instructing a computer on what are the things it has to do. Where does the action happen? In many places: the computer memory, hard drives, network cables, CPU, and so on. It's a whole "world", which most of the time is the representation of a subset of the real world.

If you write a piece of software that allows people to buy clothes online, you will have to represent real people, real clothes, real brands, sizes, and so on and so forth, within the boundaries of a program.

In order to do so, you will need to create and handle objects in the program you're writing. A person can be an object. A car is an object. A pair of socks is an object. Luckily, Python understands objects very well.

The two main features any object has are properties and methods. Let's take a person object as an example. Typically in a computer program, you'll represent people as customers or employees. The properties that you store against them are things like the name, the SSN, the age, if they have a driving license, their e-mail, gender, and so on. In a computer program, you store all the data you need in order to use an object for the purpose you're serving. If you are coding a website to sell clothes, you probably want to store the height and weight as well as other measures of your customers so that you can suggest the appropriate clothes for them. So, properties are characteristics of an object. We use them all the time: "Could you pass me that pen?" – "Which one?" – "The black one." Here, we used the "black" property of a pen to identify it (most likely amongst a blue and a red one).

Methods are things that an object can do. As a person, I have methods such as *speak*, *walk*, *sleep*, *wake-up*, *eat*, *dream*, *write*, *read*, and so on. All the things that I can do could be seen as methods of the objects that represents me.

So, now that you know what objects are and that they expose methods that you can run and properties that you can inspect, you're ready to start coding. Coding in fact is simply about managing those objects that live in the subset of the world that we're reproducing in our software. You can create, use, reuse, and delete objects as you please.

According to the *Data Model* chapter on the official Python documentation:

> "*Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects.*"

We'll take a closer look at Python objects in the upcoming chapter. For now, all we need to know is that every object in Python has an ID (or identity), a type, and a value.

Once created, the identity of an object is never changed. It's a unique identifier for it, and it's used behind the scenes by Python to retrieve the object when we want to use it.

The type as well, never changes. The type tells what operations are supported by the object and the possible values that can be assigned to it.

The value can either change or not. If it can, the object is said to be **mutable**, while when it cannot, the object is said to be **immutable**.

How do we use an object? We give it a name of course! When you give an object a name, then you can use the name to retrieve the object and use it.

In a more generic sense, objects such as numbers, strings (text), collections, and so on are associated with a name. Usually, we say that this name is the name of a variable. You can see the variable as being like a box, which you can use to hold data.

So, you have all the objects you need: what now? Well, we need to use them, right? We may want to send them over a network connection or store them in a database. Maybe display them on a web page or write them into a file. In order to do so, we need to react to a user filling in a form, or pressing a button, or opening a web page and performing a search. We react by running our code, evaluating conditions to choose which parts to execute, how many times, and under which circumstances.

And to do all this, basically we need a language. That's what Python is for. Python is the language we'll use together throughout this module to instruct the computer to do something for us.

Now, enough of this theoretical stuff, let's get started.

Enter the Python

Python is the marvelous creature of Guido Van Rossum, a Dutch computer scientist and mathematician who decided to gift the world with a project he was playing around with over Christmas 1989. The language appeared to the public somewhere around 1991, and since then has evolved to be one of the leading programming languages used worldwide today.

I started programming when I was 7 years old, on a Commodore VIC 20, which was later replaced by its bigger brother, the Commodore 64. The language was BASIC. Later on, I landed on Pascal, Assembly, C, C++, Java, JavaScript, Visual Basic, PHP, ASP, ASP .NET, C#, and other minor languages I cannot even remember, but only when I landed on Python, I finally had that feeling that you have when you find the right couch in the shop. When all of your body parts are yelling, "Buy this one! This one is perfect for us!"

It took me about a day to get used to it. Its syntax is a bit different from what I was used to, and in general, I very rarely worked with a language that defines scoping with indentation. But after getting past that initial feeling of discomfort (like having new shoes), I just fell in love with it. Deeply. Let's see why.

About Python

Before we get into the gory details, let's get a sense of why someone would want to use Python (I would recommend you to read the Python page on Wikipedia to get a more detailed introduction).

To my mind, Python exposes the following qualities.

Portability

Python runs everywhere, and porting a program from Linux to Windows or Mac is usually just a matter of fixing paths and settings. Python is designed for portability and it takes care of **operating system** (**OS**) specific quirks behind interfaces that shield you from the pain of having to write code tailored to a specific platform.

Coherence

Python is extremely logical and coherent. You can see it was designed by a brilliant computer scientist. Most of the time you can just guess how a method is called, if you don't know it.

You may not realize how important this is right now, especially if you are at the beginning, but this is a major feature. It means less cluttering in your head, less skimming through the documentation, and less need for mapping in your brain when you code.

Developer productivity

According to Mark Lutz (*Learning Python, 5th Edition*, *O'Reilly Media*), a Python program is typically one-fifth to one-third the size of equivalent Java or C++ code. This means the job gets done faster. And faster is good. Faster means a faster response on the market. Less code not only means less code to write, but also less code to read (and professional coders read much more than they write), less code to maintain, to debug, and to refactor.

Another important aspect is that Python runs without the need of lengthy and time consuming compilation and linkage steps, so you don't have to wait to see the results of your work.

An extensive library

Python has an incredibly wide standard library (it's said to come with "batteries included"). If that wasn't enough, the Python community all over the world maintains a body of third party libraries, tailored to specific needs, which you can access freely at the **Python Package Index** (**PyPI**). When you code Python and you realize that you need a certain feature, in most cases, there is at least one library where that feature has already been implemented for you.

Software quality

Python is heavily focused on readability, coherence, and quality. The language uniformity allows for high readability and this is crucial nowadays where code is more of a collective effort than a solo experience. Another important aspect of Python is its intrinsic multi-paradigm nature. You can use it as scripting language, but you also can exploit object-oriented, imperative, and functional programming styles. It is versatile.

Software integration

Another important aspect is that Python can be extended and integrated with many other languages, which means that even when a company is using a different language as their mainstream tool, Python can come in and act as a glue agent between complex applications that need to talk to each other in some way. This is kind of an advanced topic, but in the real world, this feature is very important.

Satisfaction and enjoyment

Last but not least, the fun of it! Working with Python is fun. I can code for 8 hours and leave the office happy and satisfied, alien to the struggle other coders have to endure because they use languages that don't provide them with the same amount of well-designed data structures and constructs. Python makes coding fun, no doubt about it. And fun promotes motivation and productivity.

These are the major aspects why I would recommend Python to everyone for.
Of course, there are many other technical and advanced features that I could have talked about, but they don't really pertain to an introductory section like this one. They will come up naturally, chapter after chapter, in this module.

What are the drawbacks?

Probably, the only drawback that one could find in Python, which is not due to personal preferences, is the *execution speed*. Typically, Python is slower than its compiled brothers. The standard implementation of Python produces, when you run an application, a compiled version of the source code called byte code (with the extension `.pyc`), which is then run by the Python interpreter. The advantage of this approach is portability, which we pay for with a slowdown due to the fact that Python is not compiled down to machine level as are other languages.

However, Python speed is rarely a problem today, hence its wide use regardless of this suboptimal feature. What happens is that in real life, hardware cost is no longer a problem, and usually it's easy enough to gain speed by parallelizing tasks. When it comes to number crunching though, one can switch to faster Python implementations, such as PyPy, which provides an average 7-fold speedup by implementing advanced compilation techniques (check `http://pypy.org/` for reference).

When doing data science, you'll most likely find that the libraries that you use with Python, such as Pandas and Numpy, achieve native speed due to the way they are implemented.

If that wasn't a good enough argument, you can always consider that Python is driving the backend of services such as Spotify and Instagram, where performance is a concern. Nonetheless, Python does its job perfectly adequately.

Who is using Python today?

Not yet convinced? Let's take a very brief look at the companies that are using Python today: Google, YouTube, Dropbox, Yahoo, Zope Corporation, Industrial Light & Magic, Walt Disney Feature Animation, Pixar, NASA, NSA, Red Hat, Nokia, IBM, Netflix, Yelp, Intel, Cisco, HP, Qualcomm, and JPMorgan Chase, just to name a few.

Even games such as *Battlefield 2*, *Civilization 4*, and *QuArK* are implemented using Python.

Python is used in many different contexts, such as system programming, web programming, GUI applications, gaming and robotics, rapid prototyping, system integration, data science, database applications, and much more.

Setting up the environment

Before we talk about installing Python on your system, let me tell you about which Python version I'll be using in this module.

Python 2 versus Python 3 – the great debate

Python comes in two main versions—Python 2, which is the past—and Python 3, which is the present. The two versions, though very similar, are incompatible on some aspects.

In the real world, Python 2 is actually quite far from being the past. In short, even though Python 3 has been out since 2008, the transition phase is still far from being over. This is mostly due to the fact that Python 2 is widely used in the industry, and of course, companies aren't so keen on updating their systems just for the sake of updating, following the *if it ain't broke, don't fix it* philosophy. You can read all about the transition between the two versions on the Web.

Another issue that was hindering the transition is the availability of third-party libraries. Usually, a Python project relies on tens of external libraries, and of course, when you start a new project, you need to be sure that there is already a version 3 compatible library for any business requirement that may come up. If that's not the case, starting a brand new project in Python 3 means introducing a potential risk, which many companies are not happy to take.

At the time of writing, the majority of the most widely used libraries have been ported to Python 3, and it's quite safe to start a project in Python 3 for most cases. Many of the libraries have been rewritten so that they are compatible with both versions, mostly harnessing the power of the six (2 x 3) library, which helps introspecting and adapting the behavior according to the version used.

All the examples in this module will be run using this Python 3.4.0. Most of them will run also in Python 2 (I have version 2.7.6 installed as well), and those that won't will just require some minor adjustments to cater for the small incompatibilities between the two versions.

Don't worry about this version thing though: it's not that big an issue in practice.

If any of the URLs or resources I'll point you to are no longer there by the time you read this course, just remember: Google is your friend.

What you need for this course

As you've seen there are too many requirements to get started, so I've prepared a table that will give you an overview of what you'll need for each module of the course:

Module 1	**Module 2**	**Module 3**	**Module 4**
All the examples in this module rely on the Python 3 interpreter. Some of the examples in this module rely on third-party libraries that do not ship with Python. These are introduced within the module at the time they are used, so you do not need to install them in advance. However, for completeness, here is a list: • pip • requests • pillow • bitarray	While all the examples can be run interactively in a Python shell however, we recommend using IPython for this module. The version of libraries used in this module are: • NumPy 1.9.2 • pandas 0.16.2 • matplotlib 1.4.3 • tables 3.2.2 • pymongo 3.0.3 • redis 2.10.3 • scikit-learn 0.16.1	Any modern processor (from about 2010 onwards) and 4 GB of RAM will suffice, and you can probably run almost all of the code on a slower system too. The exception here is with the final two chapters. In these chapters, I step through using Amazon Web Services (AWS) to run the code. This will probably cost you some money, but the advantage is less system setup than running the code locally. If you don't want to pay for those services, the tools used can all be set up on a local computer, but you will definitely need a modern system to run it. A processor built in at least 2012 and with more than 4 GB of RAM is necessary.	Although the code examples will also be compatible with Python 2.7, it's better if you have the latest version of Python 3 (may be 3.4.3 or newer).

Installing Python

Python is a fantastic, versatile, and an easy-to-use language. It's available for all three major operating systems—Microsoft Windows, Mac OS X, and Linux—and the installer, as well as the documentation, can be downloaded from the official Python website: `https://www.python.org`.

Windows users will need to set an environment variable in order to use Python from the command line. First, find where Python 3 is installed; the default location is `C:\Python34`. Next, enter this command into the command line (`cmd` program): set the environment to `PYTHONPATH=%PYTHONPATH%;C:\Python34`. Remember to change the `C:\Python34` if Python is installed into a different directory.

Once you have Python running on your system, you should be able to open a command prompt and run the following code:

```
$ python3
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on Linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, world!")
Hello, world!
>>> exit()
```

Note that we will be using the dollar sign (`$`) to denote that a command is to be typed into the terminal (also called a **shell** or **cmd** on Windows). You do not need to type this character (or the space that follows it). Just type in the rest of the line and press *Enter*.

After you have the above "Hello, world!" example running, exit the program and move on to installing a more advanced environment to run Python code, the IPython Notebook.

Installing IPython

IPython is a platform for Python development that contains a number of tools and environments for running Python and has more features than the standard interpreter. It contains the powerful IPython Notebook, which allows you to write programs in a web browser. It also formats your code, shows output, and allows you to annotate your scripts. It is a great tool for exploring datasets.

To install IPython on your computer, you can type the following into a command-line prompt (not into Python):

```
$ pip install ipython[all]
```

You will need administrator privileges to install this system-wide. If you do not want to (or can't) make system-wide changes, you can install it for just the current user by running this command:

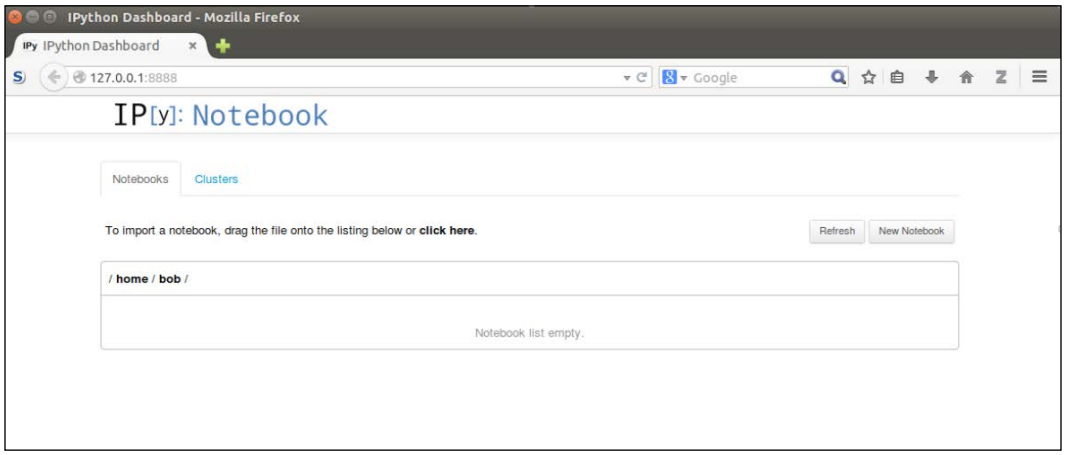
```
$ pip install --user ipython[all]
```

This will install the IPython package into a user-specific location—you will be able to use it, but nobody else on your computer can. If you are having difficulty with the installation, check the official documentation for more detailed installation instructions: `http://ipython.org/install.html`.

With the IPython Notebook installed, you can launch it with the following:

```
$ ipython3 notebook
```

This will do two things. First, it will create an IPython Notebook instance that will run in the command prompt you just used. Second, it will launch your web browser and connect to this instance, allowing you to create a new notebook. It will look something similar to the following screenshot (where `home/bob` will be replaced by your current working directory):



To stop the IPython Notebook from running, open the command prompt that has the instance running (the one you used earlier to run the IPython command). Then, press *Ctrl* + *C* and you will be prompted `Shutdown this notebook server (y/[n])?`. Type `y` and press *Enter* and the IPython Notebook will shut down.

Installing additional packages

Python 3.4 will include a program called `pip`, which is a package manager that helps to install new libraries on your system. You can verify that `pip` is working on your system by running the `$ pip3 freeze` command, which tells you which packages you have installed on your system.

The additional packages can be installed via the `pip` installer program, which has been part of the Python standard library since Python 3.3. More information about `pip` can be found at `https://docs.python.org/3/installing/index.html`.

After we have successfully installed Python, we can execute `pip` from the command-line terminal to install additional Python packages:

```
pip install SomePackage
```

Already installed packages can be updated via the `--upgrade` flag:

```
pip install SomePackage --upgrade
```

A highly recommended alternative Python distribution for scientific computing is **Anaconda** by Continuum Analytics. Anaconda is a free—including commercial use—enterprise-ready Python distribution that bundles all the essential Python packages for data science, math, and engineering in one user-friendly cross-platform distribution. The Anaconda installer can be downloaded at `http://continuum.io/downloads#py34`, and an Anaconda quick start-guide is available at `https://store.continuum.io/static/img/Anaconda-Quickstart.pdf`.

After successfully installing Anaconda, we can install new Python packages using the following command:

```
conda install SomePackage
```

Existing packages can be updated using the following command:

```
conda update SomePackage
```

The major Python packages that were used for writing this course are listed here:

- NumPy
- SciPy
- scikit-learn
- matplotlib
- pandas
- tables

- pymongo
- redis

As these packages are all hosted on PyPI, the Python package index, they can be easily installed with `pip`. To install NumPy, you would run:

```
$ pip install numpy
```

To install scikit-learn, you would run:

```
$ pip3 install -U scikit-learn
```

Important

Windows users may need to install the NumPy and SciPy libraries before installing scikit-learn. Installation instructions are available at `www.scipy.org/install.html` for those users.

Users of major Linux distributions such as Ubuntu or Red Hat may wish to install the official package from their package manager. Not all distributions have the latest versions of scikit-learn, so check the version before installing it.

Those wishing to install the latest version by compiling the source, or view more detailed installation instructions, can go to `http://scikit-learn.org/stable/install.html` to view the official documentation on installing scikit-learn.

Most libraries will have an attribute for the version, so if you already have a library installed, you can quickly check its version:

```
>>> import redis
>>> redis.__version__
'2.10.3'
```

This works well for most libraries. A few, such as pymongo, use a different attribute (pymongo uses just version, without the underscores).

How you can run a Python program

There are a few different ways in which you can run a Python program.

Running Python scripts

Python can be used as a scripting language. In fact, it always proves itself very useful. Scripts are files (usually of small dimensions) that you normally execute to do something like a task. Many developers end up having their own arsenal of tools that they fire when they need to perform a task. For example, you can have scripts to parse data in a format and render it into another different format. Or you can use a script to work with files and folders. You can create or modify configuration files, and much more. Technically, there is not much that cannot be done in a script.

It's quite common to have scripts running at a precise time on a server. For example, if your website database needs cleaning every 24 hours (for example, the table that stores the user sessions, which expire pretty quickly but aren't cleaned automatically), you could set up a cron job that fires your script at 3:00 A.M. every day.

According to Wikipedia, the software utility Cron is a time-based job scheduler in Unix-like computer operating systems. People who set up and maintain software environments use cron to schedule jobs (commands or shell scripts) to run periodically at fixed times, dates, or intervals.

I have Python scripts to do all the menial tasks that would take me minutes or more to do manually, and at some point, I decided to automate. For example, I have a laptop that doesn't have a *Fn* key to toggle the touchpad on and off. I find this very annoying, and I don't want to go clicking about through several menus when I need to do it, so I wrote a small script that is smart enough to tell my system to toggle the touchpad active state, and now I can do it with one simple click from my launcher. Priceless.

Running the Python interactive shell

Another way of running Python is by calling the interactive shell. This is something we already saw when we typed `python` on the command line of our console.

So open a console, activate your virtual environment (which by now should be second nature to you, right?), and type `python`. You will be presented with a couple of lines that should look like this (if you are on Linux):

```
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

Those `>>>` are the prompt of the shell. They tell you that Python is waiting for you to type something. If you type a simple instruction, something that fits in one line, that's all you'll see. However, if you type something that requires more than one line of code, the shell will change the prompt to `...`, giving you a visual clue that you're typing a multiline statement (or anything that would require more than one line of code).

Go on, try it out, let's do some basic maths:

```
>>> 2 + 4
6
>>> 10 / 4
2.5
>>> 2 ** 1024
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216
```

The last operation is showing you something incredible. We raise 2 to the power of 1024, and Python is handling this task with no trouble at all. Try to do it in Java, C++, or C#. It won't work, unless you use special libraries to handle such big numbers.

I use the interactive shell every day. It's extremely useful to debug very quickly, for example, to check if a data structure supports an operation. Or maybe to inspect or run a piece of code.

When you use Django (a web framework), the interactive shell is coupled with it and allows you to work your way through the framework tools, to inspect the data in the database, and many more things. You will find that the interactive shell will soon become one of your dearest friends on the journey you are embarking on.

Another solution, which comes in a much nicer graphic layout, is to use **IDLE** (**Integrated DeveLopment Environment**). It's quite a simple IDE, which is intended mostly for beginners. It has a slightly larger set of capabilities than the naked interactive shell you get in the console, so you may want to explore it. It comes for free in the Windows Python installer and you can easily install it in any other system. You can find information about it on the Python website.

Guido Van Rossum named Python after the British comedy group Monty Python, so it's rumored that the name IDLE has been chosen in honor of Erik Idle, one of Monty Python's founding members.

Running Python as a service

Apart from being run as a script, and within the boundaries of a shell, Python can be coded and run as proper software. We'll see many examples throughout the module about this mode. And we'll understand more about it in a moment, when we'll talk about how Python code is organized and run.

Running Python as a GUI application

Python can also be run as a **GUI** (**Graphical User Interface**). There are several frameworks available, some of which are cross-platform and some others are platform-specific.

Tk is a graphical user interface toolkit that takes desktop application development to a higher level than the conventional approach. It is the standard GUI for **Tool Command Language** (**TCL**), but also for many other dynamic languages and can produce rich native applications that run seamlessly under Windows, Linux, Mac OS X, and more.

Tkinter comes bundled with Python, therefore it gives the programmer easy access to the GUI world, and for these reasons, I have chosen it to be the framework for the GUI examples that I'll present in this module.

Among the other GUI frameworks, we find that the following are the most widely used:

- PyQt
- wxPython
- PyGtk

Describing them in detail is outside the scope of this module, but you can find all the information you need on the Python website in the *GUI Programming* section. If GUIs are what you're looking for, remember to choose the one you want according to some principles. Make sure they:

- Offer all the features you may need to develop your project
- Run on all the platforms you may need to support
- Rely on a community that is as wide and active as possible
- Wrap graphic drivers/tools that you can easily install/access

How is Python code organized

Let's talk a little bit about how Python code is organized. In this paragraph, we'll start going down the rabbit hole a little bit more and introduce a bit more technical names and concepts.

Starting with the basics, how is Python code organized? Of course, you write your code into files. When you save a file with the extension `.py`, that file is said to be a Python module.

If you're on Windows or Mac, which typically hide file extensions to the user, please make sure you change the configuration so that you can see the complete name of the files. This is not strictly a requirement, but a hearty suggestion.

It would be impractical to save all the code that it is required for software to work within one single file. That solution works for *scripts*, which are usually not longer than a few hundred lines (and often they are quite shorter than that).

A complete Python application can be made of hundreds of thousands of lines of code, so you will have to scatter it through different modules. Better, but not nearly good enough. It turns out that even like this it would still be impractical to work with the code. So Python gives you another structure, called **package**, which allows you to group modules together. A package is nothing more than a folder, which must contain a special file, `__init__.py` that doesn't need to hold any code but whose presence is required to tell Python that the folder is not just some folder, but it's actually a package (note that as of Python 3.3 `__init__.py` is not strictly required any more).

As always, an example will make all of this much clearer. I have created an example structure in my module project, and when I type in my Linux console:

```
$ tree -v example
```

I get a tree representation of the contents of the `ch1/example` folder, which holds the code for the examples of this chapter. Here's how a structure of a real simple application could look like:

```
example/
├── core.py
├── run.py
└── util
    ├── __init__.py
    ├── db.py
    ├── math.py
    └── network.py
```

You can see that within the root of this example, we have two modules, `core.py` and `run.py`, and one package: `util`. Within `core.py`, there may be the core logic of our application. On the other hand, within the `run.py` module, we can probably find the logic to start the application. Within the `util` package, I expect to find various utility tools, and in fact, we can guess that the modules there are called by the type of tools they hold: `db.py` would hold tools to work with databases, `math.py` would of course hold mathematical tools (maybe our application deals with financial data), and `network.py` would probably hold tools to send/receive data on networks.

As explained before, the `__init__.py` file is there just to tell Python that `util` is a package and not just a mere folder.

Had this software been organized within modules only, it would have been much harder to infer its structure. I put a *module only* example under the `ch1/files_only` folder, see it for yourself:

```
$ tree -v files_only
```

This shows us a completely different picture:

```
files_only/
├── core.py
├── db.py
├── math.py
├── network.py
└── run.py
```

It is a little harder to guess what each module does, right? Now, consider that this is just a simple example, so you can guess how much harder it would be to understand a real application if we couldn't organize the code in packages and modules.

How do we use modules and packages

When a developer is writing an application, it is very likely that they will need to apply the same piece of logic in different parts of it. For example, when writing a parser for the data that comes from a form that a user can fill in a web page, the application will have to validate whether a certain field is holding a number or not. Regardless of how the logic for this kind of validation is written, it's very likely that it will be needed in more than one place. For example in a poll application, where the user is asked many question, it's likely that several of them will require a numeric answer. For example:

- What is your age
- How many pets do you own
- How many children do you have
- How many times have you been married

It would be very bad practice to copy paste (or, more properly said: duplicate) the validation logic in every place where we expect a numeric answer. This would violate the **DRY** (**Don't Repeat Yourself**) principle, which states that you should never repeat the same piece of code more than once in your application. I feel the need to stress the importance of this principle: *you should never repeat the same piece of code more than once in your application* (got the irony?).

There are several reasons why repeating the same piece of logic can be very bad, the most important ones being:

- There could be a bug in the logic, and therefore, you would have to correct it in every place that logic is applied.
- You may want to amend the way you carry out the validation, and again you would have to change it in every place it is applied.
- You may forget to fix/amend a piece of logic because you missed it when searching for all its occurrences. This would leave wrong/inconsistent behavior in your application.
- Your code would be longer than needed, for no good reason.

Python is a wonderful language and provides you with all the tools you need to apply all the coding best practices. For this particular example, we need to be able to reuse a piece of code. To be able to reuse a piece of code, we need to have a construct that will hold the code for us so that we can call that construct every time we need to repeat the logic inside it. That construct exists, and it's called **function**.

I'm not going too deep into the specifics here, so please just remember that a function is a block of organized, reusable code which is used to perform a task. Functions can assume many forms and names, according to what kind of environment they belong to, but for now this is not important. We'll see the details when we are able to appreciate them, later on, in the module. Functions are the building blocks of modularity in your application, and they are almost indispensable (unless you're writing a super simple script, you'll use functions all the time).

Python comes with a very extensive library, as I already said a few pages ago. Now, maybe it's a good time to define what a library is: a **library** is a collection of functions and objects that provide functionalities that enrich the abilities of a language.

For example, within Python's `math` library we can find a plethora of functions, one of which is the `factorial` function, which of course calculates the factorial of a number.

In mathematics, the **factorial** of a non-negative integer number *N*, denoted as *N!*, is defined as the product of all positive integers less than or equal to N. For example, the factorial of 5 is calculated as:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

The factorial of `0` is `0! = 1`, to respect the convention for an empty product.

So, if you wanted to use this function in your code, all you would have to do is to import it and call it with the right input values. Don't worry too much if input values and the concept of calling is not very clear for now, please just concentrate on the import part.

We use a library by importing what we need from it, and then we use it.

In Python, to calculate the factorial of number 5, we just need the following code:

```
>>> from math import factorial
>>> factorial(5)
120
```

Whatever we type in the shell, if it has a printable representation, will be printed on the console for us (in this case, the result of the function call: 120).

So, let's go back to our example, the one with `core.py`, `run.py`, `util`, and so on.

In our example, the package `util` is our utility library. Our custom utility belt that holds all those reusable tools (that is, functions), which we need in our application. Some of them will deal with databases (`db.py`), some with the network (`network.py`), and some will perform mathematical calculations (`math.py`) that are outside the scope of Python's standard `math` library and therefore, we had to code them for ourselves.

Let's now talk about another very important concept: Python's execution model.

Python's execution model

In this paragraph, I would like to introduce you to a few very important concepts, such as scope, names, and namespaces. You can read all about Python's execution model in the official Language reference, of course, but I would argue that it is quite technical and abstract, so let me give you a less formal explanation first.

Names and namespaces

Say you are looking for a module, so you go to the library and ask someone for the module you want to fetch. They tell you something like "second floor, section X, row three". So you go up the stairs, look for section X, and so on.

It would be very different to enter a library where all the books are piled together in random order in one big room. No floors, no sections, no rows, no order. Fetching a module would be extremely hard.

When we write code we have the same issue: we have to try and organize it so that it will be easy for someone who has no prior knowledge about it to find what they're looking for. When software is structured correctly, it also promotes code reuse. On the other hand, disorganized software is more likely to expose scattered pieces of duplicated logic.

First of all, let's start with the module. We refer to a module by its title and in Python lingo, that would be a name. Python names are the closest abstraction to what other languages call variables. Names basically refer to objects and are introduced by name binding operations. Let's make a quick example (notice that anything that follows a `#` is a comment):

```
>>> n = 3  # integer number
>>> address = "221b Baker Street, NW1 6XE, London"  # S. Holmes
>>> employee = {
...     'age': 45,
...     'role': 'CTO',
...     'SSN': 'AB1234567',
... }
>>> # let's print them
>>> n
3
>>> address
'221b Baker Street, NW1 6XE, London'
>>> employee
```

```
{'role': 'CTO', 'SSN': 'AB1234567', 'age': 45}
>>> # what if I try to print a name I didn't define?
>>> other_name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'other_name' is not defined
```

We defined three objects in the preceding code (do you remember what are the three features every Python object has?):

- An integer number `n` (type: `int`, value: `3`)
- A string `address` (type: `str`, value: Sherlock Holmes' address)
- A dictionary `employee` (type: `dict`, value: a dictionary which holds three key/value pairs)

Don't worry, I know you're not supposed to know what a dictionary is. We'll see in the upcoming chapter that it's the king of Python data structures.

Have you noticed that the prompt changed from `>>>` to `...` when I typed in the definition of employee? That's because the definition spans over multiple lines.

So, what are `n`, `address` and `employee`? They are **names**. Names that we can use to retrieve data within our code. They need to be kept somewhere so that whenever we need to retrieve those objects, we can use their names to fetch them. We need some space to hold them, hence: namespaces!

A **namespace** is therefore a mapping from names to objects. Examples are the set of built-in names (containing functions that are always accessible for free in any Python program), the global names in a module, and the local names in a function. Even the set of attributes of an object can be considered a namespace.

The beauty of namespaces is that they allow you to define and organize your names with clarity, without overlapping or interference. For example, the namespace associated with that module we were looking for in the library can be used to import the module itself, like this:

```
from library.second_floor.section_x.row_three import module
```

We start from the `library` namespace, and by means of the dot (`.`) operator, we walk into that namespace. Within this namespace, we look for `second_floor`, and again we walk into it with the `.` operator. We then walk into `section_x`, and finally within the last namespace, `row_tree`, we find the name we were looking for: `module`.

Walking through a namespace will be clearer when we'll be dealing with real code examples. For now, just keep in mind that namespaces are places where names are associated to objects.

There is another concept, which is closely related to that of a namespace, which I'd like to briefly talk about: the **scope**.

Scopes

According to Python's documentation, *a scope is a textual region of a Python program, where a namespace is directly accessible*. Directly accessible means that when you're looking for an unqualified reference to a name, Python tries to find it in the namespace.

Scopes are determined statically, but actually during runtime they are used dynamically. This means that by inspecting the source code you can tell what the scope of an object is, but this doesn't prevent the software to alter that during runtime. There are four different scopes that Python makes accessible (not necessarily all of them present at the same time, of course):

- The **local** scope, which is the innermost one and contains the local names.
- The **enclosing** scope, that is, the scope of any enclosing function. It contains non-local names and also non-global names.
- The **global** scope contains the global names.
- The **built-in** scope contains the built-in names. Python comes with a set of functions that you can use in a off-the-shelf fashion, such as `print`, `all`, `abs`, and so on. They live in the built-in scope.

The rule is the following: when we refer to a name, Python starts looking for it in the current namespace. If the name is not found, Python continues the search to the enclosing scope and this continue until the built-in scope is searched. If a name hasn't been found after searching the built-in scope, then Python raises a `NameError` **exception**, which basically means that the name hasn't been defined (you saw this in the preceding example).

The order in which the namespaces are scanned when looking for a name is therefore: **local**, **enclosing**, **global**, **built-in** (**LEGB**).

This is all very theoretical, so let's see an example. In order to show you Local and Enclosing namespaces, I will have to define a few functions. Just remember that in the following code, when you see `def`, it means I'm defining a function.

scopes1.py

```
# Local versus Global

# we define a function, called local
def local():
    m = 7
    print(m)

m = 5
print(m)

# we call, or `execute` the function local
local()
```

In the preceding example, we define the same name `m`, both in the global scope and in the local one (the one defined by the function local). When we execute this program with the following command (have you activated your virtualenv?):

```
$ python scopes1.py
```

We see two numbers printed on the console: `5` and `7`.

What happens is that the Python interpreter parses the file, top to bottom. First, it finds a couple of comment lines, which are skipped, then it parses the definition of the function `local`. When called, this function does two things: it sets up a name to an object representing number 7 and prints it. The Python interpreter keeps going and it finds another name binding. This time the binding happens in the global scope and the value is 5. The next line is a call to the `print` function, which is executed (and so we get the first value printed on the console: `5`).

After this, there is a call to the function `local`. At this point, Python executes the function, so at this time, the binding `m = 7` happens and it's printed.

One very important thing to notice is that the part of the code that belongs to the definition of the function local is indented by four spaces on the right. Python in fact defines scopes by indenting the code. You walk into a scope by indenting and walk out of it by unindenting. Some coders use two spaces, others three, but the suggested number of spaces to use is four. It's a good measure to maximize readability. We'll talk more about all the conventions you should embrace when writing Python code later.

What would happen if we removed that `m = 7` line? Remember the LEGB rule. Python would start looking for `m` in the local scope (function `local`), and, not finding it, it would go to the next enclosing scope. The next one in this case is the global one because there is no enclosing function wrapped around `local`. Therefore, we would see two number `5` printed on the console. Let's actually see how the code would look like:

```
scopes2.py
    # Local versus Global

    def local():
        # m doesn't belong to the scope defined by the local function
        # so Python will keep looking into the next enclosing scope.
        # m is finally found in the global scope
        print(m, 'printing from the local scope')

    m = 5
    print(m, 'printing from the global scope')

    local()
```

Running `scopes2.py` will print this:

```
(.lpvenv)fab@xps:ch1$ python scopes2.py
5 printing from the global scope
5 printing from the local scope
```

As expected, Python prints `m` the first time, then when the function `local` is called, `m` isn't found in its scope, so Python looks for it following the LEGB chain until `m` is found in the global scope.

Let's see an example with an extra layer, the enclosing scope:

```
scopes3.py
    # Local, Enclosing and Global

    def enclosing_func():
        m = 13
        def local():
            # m doesn't belong to the scope defined by the local
            # function so Python will keep looking into the next
            # enclosing scope. This time m is found in the enclosing
            # scope
            print(m, 'printing from the local scope')

        # calling the function local
        local()
    m = 5
    print(m, 'printing from the global scope')

    enclosing_func()
```

Running `scopes3.py` will print on the console:

```
(.lpvenv)fab@xps:ch1$ python scopes3.py
5 printing from the global scope
13 printing from the local scope
```

As you can see, the `print` instruction from the function `local` is referring to `m` as before. `m` is still not defined within the function itself, so Python starts walking scopes following the LEGB order. This time `m` is found in the enclosing scope.

Don't worry if this is still not perfectly clear for now. It will come to you as we go through the examples in the module. The *Classes* section of the Python tutorial (official documentation) has an interesting paragraph about scopes and namespaces. Make sure you read it at some point if you wish for a deeper understanding of the subject.

Before we finish off this chapter, I would like to talk a bit more about objects. After all, basically everything in Python is an object, so I think they deserve a bit more attention.

Guidelines on how to write good code

Writing good code is not as easy as it seems. As I already said before, good code exposes a long list of qualities that is quite hard to put together. Writing good code is, to some extent, an art. Regardless of where on the path you will be happy to settle, there is something that you can embrace which will make your code instantly better: **PEP8**.

According to Wikipedia:

> "*Python's development is conducted largely through the Python Enhancement Proposal (PEP) process. The PEP process is the primary mechanism for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python.*"

Among all the PEPs, probably the most famous one is PEP8. It lays out a simple but effective set of guidelines to define Python aesthetic so that we write beautiful Python code. If you take one suggestion out of this chapter, please let it be this: use it. Embrace it. You will thank me later.

Coding today is no longer a check-in/check-out business. Rather, it's more of a social effort. Several developers collaborate to a piece of code through tools like git and mercurial, and the result is code that is fathered by many different hands.

Git and Mercurial are probably the most used distributed revision control systems today. They are essential tools designed to help teams of developers collaborate on the same software.

These days, more than ever, we need to have a consistent way of writing code, so that readability is maximized. When all developers of a company abide with PEP8, it's not uncommon for any of them landing on a piece of code to think they wrote it themselves. It actually happens to me all the time (I always forget the code I write).

This has a tremendous advantage: when you read code that you could have written yourself, you read it easily. Without a convention, every coder would structure the code the way they like most, or simply the way they were taught or are used to, and this would mean having to interpret every line according to someone else's style. It would mean having to lose much more time just trying to understand it. Thanks to PEP8, we can avoid this. I'm such a fan of it that I won't sign off a code review if the code doesn't respect it. So please take the time to study it, it's very important.

In the examples of this module, I will try to respect it as much as I can. Unfortunately, I don't have the luxury of 79 characters (which is the maximum line length suggested by PEP*), and I will have to cut down on blank lines and other things, but I promise you I'll try to layout my code so that it's as readable as possible.

The Python culture

Python has been adopted widely in all coding industries. It's used by many different companies for many different purposes, and it's also used in education (it's an excellent language for that purpose, because of its many qualities and the fact that it's easy to learn).

One of the reasons Python is so popular today is that the community around it is vast, vibrant, and full of brilliant people. Many events are organized all over the world, mostly either around Python or its main web framework, Django.

Python is open, and very often so are the minds of those who embrace it. Check out the community page on the Python website for more information and get involved!

There is another aspect to Python which revolves around the notion of being **Pythonic**. It has to do with the fact that Python allows you to use some idioms that aren't found elsewhere, at least not in the same form or easiness of use (I feel quite claustrophobic when I have to code in a language which is not Python now).

Anyway, over the years, this concept of being Pythonic has emerged and, the way I understand it, is something along the lines of *doing things the way they are supposed to be done in Python.*

To help you understand a little bit more about Python's culture and about being Pythonic, I will show you the *Zen of Python*. A lovely Easter egg that is very popular. Open up a Python console and type `import this`. What follows is the result of this line:

```
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Downloading the example code

The code files for all the four parts of the course are available at `https://github.com/PacktPublishing/Data-Science-With-Python`.

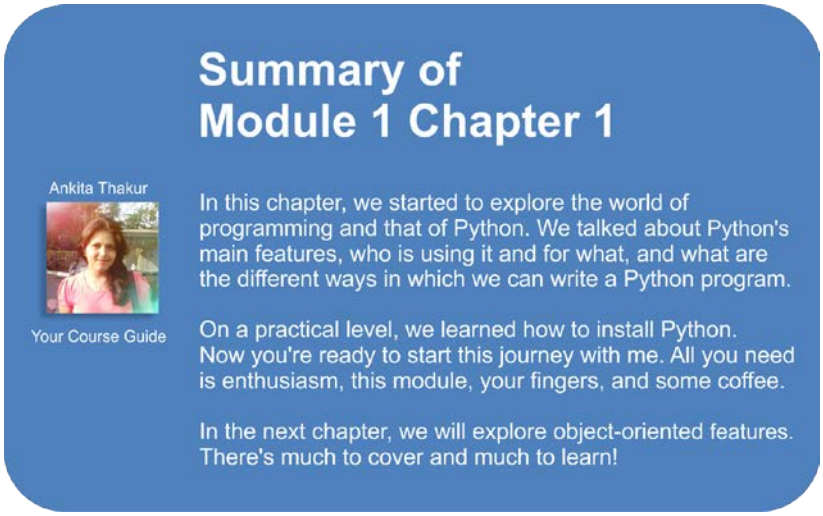
There are two levels of reading here. One is to consider it as a set of guidelines that have been put down in a fun way. The other one is to keep it in mind, and maybe read it once in a while, trying to understand how it refers to something deeper. Some Python characteristics that you will have to understand deeply in order to write Python the way it's supposed to be written. Start with the fun level, and then dig deeper. Always dig deeper.

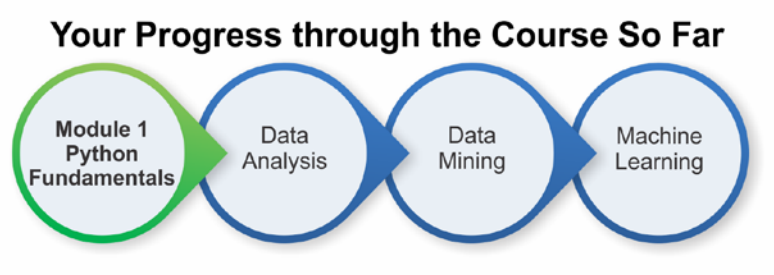
A note on the IDEs

Just a few words about **Integrated Development Environments** (**IDEs**). To follow the examples in this module you don't need one, any text editor will do fine. If you want to have more advanced features such as syntax coloring and auto completion, you will have to fetch yourself an IDE. You can find a comprehensive list of open source IDEs (just Google "python ides") on the Python website. I personally use Sublime Text editor. It's free to try out and it costs just a few dollars. I have tried many IDEs in my life, but this is the one that makes me most productive.

Two extremely important pieces of advice:

- Whatever IDE you will chose to use, try to learn it well so that you can exploit its strengths, but *don't depend on it*. Exercise yourself to work with VIM (or any other text editor) once in a while, learn to be able to do some work on any platform, with any set of tools.
- Whatever text editor/IDE you will use, when it comes to writing Python, *indentation is four spaces*. Don't use tabs, don't mix them with spaces. Use four spaces, not two, not three, not five. Just use four. The whole world works like that, and you don't want to become an outcast because you were fond of the three-space layout.


Summary of
Module 1 Chapter 1
Ankita Thakur
Your Course Guide
In this chapter, we started to explore the world of programming and that of Python. We talked about Python's main features, who is using it and for what, and what are the different ways in which we can write a Python program.
On a practical level, we learned how to install Python. Now you're ready to start this journey with me. All you need is enthusiasm, this module, your fingers, and some coffee.
In the next chapter, we will explore object-oriented features. There's much to cover and much to learn!


Your Progress through the Course So Far
Module 1 Python Fundamentals
Data Analysis
Data Mining
Machine Learning

2
Object-oriented Design

In software development, design is often considered as the step done *before* programming. This isn't true; in reality, analysis, programming, and design tend to overlap, combine, and interweave. In this chapter, we will cover the following topics:

- What object-oriented means
- The difference between object-oriented design and object-oriented programming
- The basic principles of an object-oriented design
- Basic **Unified Modeling Language** (**UML**) and when it isn't evil

Introducing object-oriented

Everyone knows what an object is—a tangible thing that we can sense, feel, and manipulate. The earliest objects we interact with are typically baby toys. Wooden blocks, plastic shapes, and over-sized puzzle pieces are common first objects. Babies learn quickly that certain objects do certain things: bells ring, buttons press, and levers pull.

The definition of an object in software development is not terribly different. Software objects are not typically tangible things that you can pick up, sense, or feel, but they are models of something that can do certain things and have certain things done to them. Formally, an object is a collection of **data** and associated **behaviors**.

So, knowing what an object is, what does it mean to be object-oriented? Oriented simply means *directed toward*. So object-oriented means functionally directed towards modeling objects. This is one of the many techniques used for modeling complex systems by describing a collection of interacting objects via their data and behavior.

If you've read any hype, you've probably come across the terms object-oriented analysis, object-oriented design, object-oriented analysis and design, and object-oriented programming. These are all highly related concepts under the general object-oriented umbrella.

In fact, analysis, design, and programming are all stages of software development. Calling them object-oriented simply specifies what style of software development is being pursued.

Object-oriented analysis (**OOA**) is the process of looking at a problem, system, or task (that somebody wants to turn into an application) and identifying the objects and interactions between those objects. The analysis stage is all about *what* needs to be done.

The output of the analysis stage is a set of requirements. If we were to complete the analysis stage in one step, we would have turned a task, such as, I need a website, into a set of requirements. For example:

Website visitors need to be able to (*italic* represents actions, **bold** represents objects):

- *review* our **history**
- *apply* for **jobs**
- *browse*, *compare*, and *order* **products**

In some ways, analysis is a misnomer. The baby we discussed earlier doesn't analyze the blocks and puzzle pieces. Rather, it will explore its environment, manipulate shapes, and see where they might fit. A better turn of phrase might be object-oriented exploration. In software development, the initial stages of analysis include interviewing customers, studying their processes, and eliminating possibilities.

Object-oriented design (**OOD**) is the process of converting such requirements into an implementation specification. The designer must name the objects, define the behaviors, and formally specify which objects can activate specific behaviors on other objects. The design stage is all about *how* things should be done.

The output of the design stage is an implementation specification. If we were to complete the design stage in a single step, we would have turned the requirements defined during object-oriented analysis into a set of classes and interfaces that could be implemented in (ideally) any object-oriented programming language.

Object-oriented programming (**OOP**) is the process of converting this perfectly defined design into a working program that does exactly what the CEO originally requested.

Yeah, right! It would be lovely if the world met this ideal and we could follow these stages one by one, in perfect order, like all the old textbooks told us to. As usual, the real world is much murkier. No matter how hard we try to separate these stages, we'll always find things that need further analysis while we're designing. When we're programming, we find features that need clarification in the design.

Most twenty-first century development happens in an iterative development model. In iterative development, a small part of the task is modeled, designed, and programmed, then the program is reviewed and expanded to improve each feature and include new features in a series of short development cycles.

The rest of this module is about object-oriented programming, but in this chapter, we will cover the basic object-oriented principles in the context of design. This allows us to understand these (rather simple) concepts without having to argue with software syntax or Python interpreters.

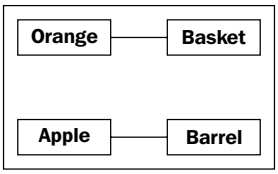
Objects and classes

So, an object is a collection of data with associated behaviors. How do we differentiate between types of objects? Apples and oranges are both objects, but it is a common adage that they cannot be compared. Apples and oranges aren't modeled very often in computer programming, but let's pretend we're doing an inventory application for a fruit farm. To facilitate the example, we can assume that apples go in barrels and oranges go in baskets.

Now, we have four kinds of objects: apples, oranges, baskets, and barrels. In object-oriented modeling, the term used for *kind of object* is **class**. So, in technical terms, we now have four classes of objects.

What's the difference between an object and a class? Classes describe objects. They are like blueprints for creating an object. You might have three oranges sitting on the table in front of you. Each orange is a distinct object, but all three have the attributes and behaviors associated with one class: the general class of oranges.

The relationship between the four classes of objects in our inventory system can be described using a **Unified Modeling Language** (invariably referred to as **UML**, because three letter acronyms never go out of style) class diagram. Here is our first class diagram:



This diagram shows that an **Orange** is somehow associated with a **Basket** and that an **Apple** is also somehow associated with a **Barrel**. Association is the most basic way for two classes to be related.

UML is very popular among managers, and occasionally disparaged by programmers. The syntax of a UML diagram is generally pretty obvious; you don't have to read a tutorial to (mostly) understand what is going on when you see one. UML is also fairly easy to draw, and quite intuitive. After all, many people, when describing classes and their relationships, will naturally draw boxes with lines between them. Having a standard based on these intuitive diagrams makes it easy for programmers to communicate with designers, managers, and each other.

However, some programmers think UML is a waste of time. Citing iterative development, they will argue that formal specifications done up in fancy UML diagrams are going to be redundant before they're implemented, and that maintaining these formal diagrams will only waste time and not benefit anyone.

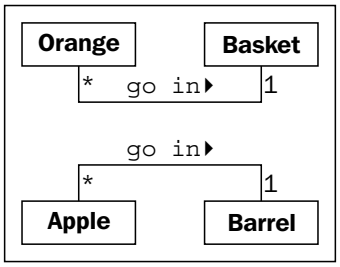
Depending on the corporate structure involved, this may or may not be true. However, every programming team consisting of more than one person will occasionally has to sit down and hash out the details of the subsystem it is currently working on. UML is extremely useful in these brainstorming sessions for quick and easy communication. Even those organizations that scoff at formal class diagrams tend to use some informal version of UML in their design meetings or team discussions.

Further, the most important person you will ever have to communicate with is yourself. We all think we can remember the design decisions we've made, but there will always be the *Why did I do that?* moments hiding in our future. If we keep the scraps of papers we did our initial diagramming on when we started a design, we'll eventually find them a useful reference.

This chapter, however, is not meant to be a tutorial in UML. There are many of these available on the Internet, as well as numerous books available on the topic. UML covers far more than class and object diagrams; it also has a syntax for use cases, deployment, state changes, and activities. We'll be dealing with some common class diagram syntax in this discussion of object-oriented design. You'll find that you can pick up the structure by example, and you'll subconsciously choose the UML-inspired syntax in your own team or personal design sessions.

Our initial diagram, while correct, does not remind us that apples go in barrels or how many barrels a single apple can go in. It only tells us that apples are somehow associated with barrels. The association between classes is often obvious and needs no further explanation, but we have the option to add further clarification as needed.

The beauty of UML is that most things are optional. We only need to specify as much information in a diagram as makes sense for the current situation. In a quick whiteboard session, we might just quickly draw lines between boxes. In a formal document, we might go into more detail. In the case of apples and barrels, we can be fairly confident that the association is, **many apples go in one barrel**, but just to make sure nobody confuses it with, **one apple spoils one barrel**, we can enhance the diagram as shown:



This diagram tells us that oranges **go in** baskets with a little arrow showing what goes in what. It also tells us the number of that object that can be used in the association on both sides of the relationship. One **Basket** can hold many (represented by a ***) **Orange** objects. Any one **Orange** can go in exactly one **Basket**. This number is referred to as the multiplicity of the object. You may also hear it described as the cardinality. These are actually slightly distinct terms. Cardinality refers to the actual number of items in the set, whereas multiplicity specifies how small or how large this number could be.

I frequently forget which side of a relationship the multiplicity goes on. The multiplicity nearest to a class is the number of objects of that class that can be associated with any one object at the other end of the association. For the apple goes in barrel association, reading from left to right, many instances of the **Apple** class (that is many **Apple** objects) can go in any one **Barrel**. Reading from right to left, exactly one **Barrel** can be associated with any one **Apple**.

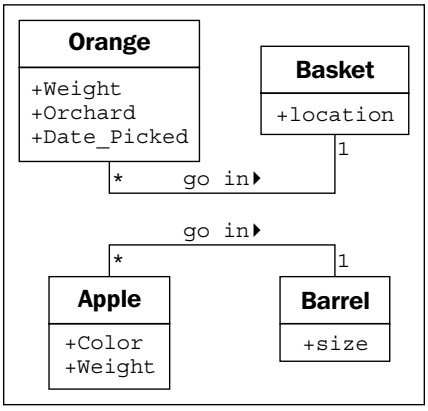
Specifying attributes and behaviors

We now have a grasp of some basic object-oriented terminology. Objects are instances of classes that can be associated with each other. An object instance is a specific object with its own set of data and behaviors; a specific orange on the table in front of us is said to be an instance of the general class of oranges. That's simple enough, but what are these data and behaviors that are associated with each object?

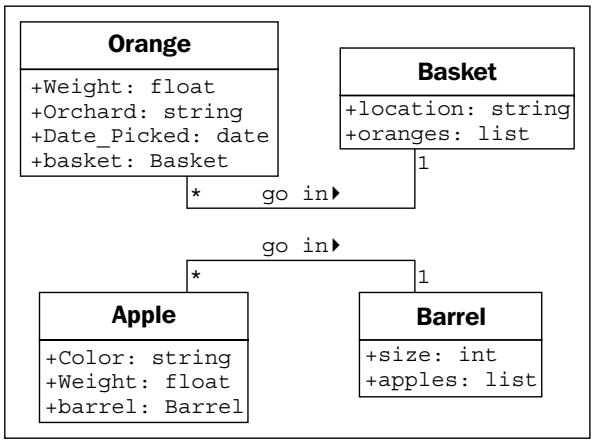
Data describes objects

Let's start with data. Data typically represents the individual characteristics of a certain object. A class can define specific sets of characteristics that are shared by all objects of that class. Any specific object can have different data values for the given characteristics. For example, our three oranges on the table (if we haven't eaten any) could each weigh a different amount. The orange class could then have a weight **attribute**. All instances of the orange class have a weight attribute, but each orange has a different value for this attribute. Attributes don't have to be unique, though; any two oranges may weigh the same amount. As a more realistic example, two objects representing different customers might have the same value for a first name attribute.

Attributes are frequently referred to as **members** or **properties**. Some authors suggest that the terms have different meanings, usually that attributes are settable, while properties are read-only. In Python, the concept of "read-only" is rather pointless, so throughout this module, we'll see the two terms used interchangeably. In addition, as we'll discuss in *Chapter 5*, *When to Use Object-oriented Programming*, the property keyword has a special meaning in Python for a particular kind of attribute.



In our fruit inventory application, the fruit farmer may want to know what orchard the orange came from, when it was picked, and how much it weighs. They might also want to keep track of where each basket is stored. Apples might have a color attribute, and barrels might come in different sizes. Some of these properties may also belong to multiple classes (we may want to know when apples are picked, too), but for this first example, let's just add a few different attributes to our class diagram:



Depending on how detailed our design needs to be, we can also specify the type for each attribute. Attribute types are often primitives that are standard to most programming languages, such as integer, floating-point number, string, byte, or Boolean. However, they can also represent data structures such as lists, trees, or graphs, or most notably, other classes. This is one area where the design stage can overlap with the programming stage. The various primitives or objects available in one programming language may be somewhat different from what is available in other languages.

Usually, we don't need to be overly concerned with data types at the design stage, as implementation-specific details are chosen during the programming stage. Generic names are normally sufficient for design. If our design calls for a list container type, the Java programmers can choose to use a `LinkedList` or an `ArrayList` when implementing it, while the Python programmers (that's us!) can choose between the `list` built-in and a `tuple`.

In our fruit-farming example so far, our attributes are all basic primitives. However, there are some implicit attributes that we can make explicit—the associations. For a given orange, we might have an attribute containing the basket that holds that orange.

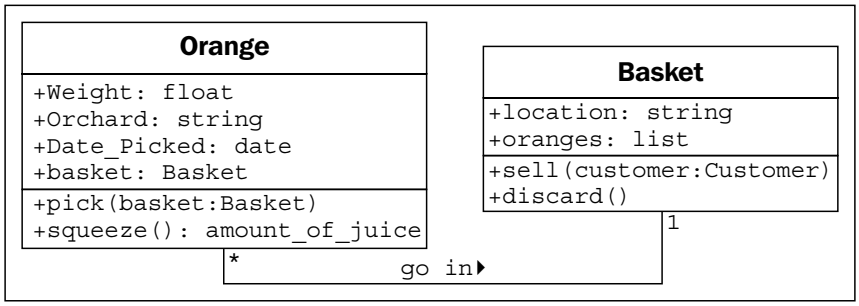
Behaviors are actions

Now, we know what data is, but what are behaviors? Behaviors are actions that can occur on an object. The behaviors that can be performed on a specific class of objects are called **methods**. At the programming level, methods are like functions in structured programming, but they magically have access to all the data associated with this object. Like functions, methods can also accept **parameters** and return **values**.

Parameters to a method are a list of objects that need to be **passed** into the method that is being called (the objects that are passed in from the calling object are usually referred to as **arguments**). These objects are used by the method to perform whatever behavior or task it is meant to do. Returned values are the results of that task.

We've stretched our "comparing apples and oranges" example into a basic (if far-fetched) inventory application. Let's stretch it a little further and see if it breaks. One action that can be associated with oranges is the **pick** action. If you think about implementation, **pick** would place the orange in a basket by updating the **basket** attribute of the orange, and by adding the orange to the **oranges** list on the **Basket**. So, **pick** needs to know what basket it is dealing with. We do this by giving the **pick** method a **basket** parameter. Since our fruit farmer also sells juice, we can add a **squeeze** method to **Orange**. When squeezed, **squeeze** might return the amount of juice retrieved, while also removing the **Orange** from the **basket** it was in.

Basket can have a **sell** action. When a basket is sold, our inventory system might update some data on as-yet unspecified objects for accounting and profit calculations. Alternatively, our basket of oranges might go bad before we can sell them, so we add a **discard** method. Let's add these methods to our diagram:



Adding models and methods to individual objects allows us to create a **system** of interacting objects. Each object in the system is a member of a certain class. These classes specify what types of data the object can hold and what methods can be invoked on it. The data in each object can be in a different state from other objects of the same class, and each object may react to method calls differently because of the differences in state.

Object-oriented analysis and design is all about figuring out what those objects are and how they should interact. The next section describes principles that can be used to make those interactions as simple and intuitive as possible.

Hiding details and creating the public interface

The key purpose of modeling an object in object-oriented design is to determine what the public **interface** of that object will be. The interface is the collection of attributes and methods that other objects can use to interact with that object. They do not need, and are often not allowed, to access the internal workings of the object. A common real-world example is the television. Our interface to the television is the remote control. Each button on the remote control represents a method that can be called on the television object. When we, as the calling object, access these methods, we do not know or care if the television is getting its signal from an antenna, a cable connection, or a satellite dish. We don't care what electronic signals are being sent to adjust the volume, or whether the sound is destined to speakers or headphones. If we open the television to access the internal workings, for example, to split the output signal to both external speakers and a set of headphones, we will void the warranty.

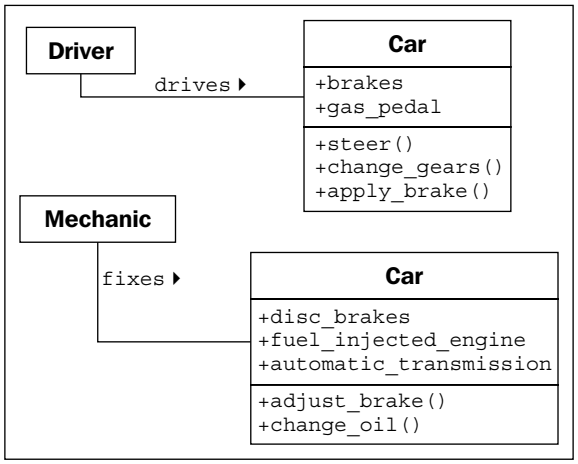
This process of hiding the implementation, or functional details, of an object is suitably called **information hiding**. It is also sometimes referred to as **encapsulation**, but encapsulation is actually a more all-encompassing term. Encapsulated data is not necessarily hidden. Encapsulation is, literally, creating a capsule and so think of creating a time capsule. If you put a bunch of information into a time capsule, lock and bury it, it is both encapsulated and the information is hidden. On the other hand, if the time capsule has not been buried and is unlocked or made of clear plastic, the items inside it are still encapsulated, but there is no information hiding.

The distinction between encapsulation and information hiding is largely irrelevant, especially at the design level. Many practical references use these terms interchangeably. As Python programmers, we don't actually have or need true information hiding, (we'll discuss the reasons for this in *Chapter 2*, *Objects in Python*) so the more encompassing definition for encapsulation is suitable.

The public interface, however, is very important. It needs to be carefully designed as it is difficult to change it in the future. Changing the interface will break any client objects that are calling it. We can change the internals all we like, for example, to make it more efficient, or to access data over the network as well as locally, and the client objects will still be able to talk to it, unmodified, using the public interface. On the other hand, if we change the interface by changing attribute names that are publicly accessed, or by altering the order or types of arguments that a method can accept, all client objects will also have to be modified. While on the topic of public interfaces, keep it simple. Always design the interface of an object based on how easy it is to use, not how hard it is to code (this advice applies to user interfaces as well).

Remember, program objects may represent real objects, but that does not make them real objects. They are models. One of the greatest gifts of modeling is the ability to ignore irrelevant details. The model car I built as a child may look like a real 1956 Thunderbird on the outside, but it doesn't run and the driveshaft doesn't turn. These details were overly complex and irrelevant before I started driving. The model is an **abstraction** of a real concept.

Abstraction is another object-oriented concept related to encapsulation and information hiding. Simply put, abstraction means dealing with the level of detail that is most appropriate to a given task. It is the process of extracting a public interface from the inner details. A driver of a car needs to interact with steering, gas pedal, and brakes. The workings of the motor, drive train, and brake subsystem don't matter to the driver. A mechanic, on the other hand, works at a different level of abstraction, tuning the engine and bleeding the brakes. Here's an example of two abstraction levels for a car:



Now, we have several new terms that refer to similar concepts. Condensing all this jargon into a couple of sentences: abstraction is the process of encapsulating information with separate public and private interfaces. The private interfaces can be subject to information hiding.

The important lesson to take from all these definitions is to make our models understandable to other objects that have to interact with them. This means paying careful attention to small details. Ensure methods and properties have sensible names. When analyzing a system, objects typically represent nouns in the original problem, while methods are normally verbs. Attributes can often be picked up as adjectives, although if the attribute refers to another object that is part of the current object, it will still likely be a noun. Name classes, attributes, and methods accordingly.

Don't try to model objects or actions that *might* be useful in the future. Model exactly those tasks that the system needs to perform, and the design will naturally gravitate towards the one that has an appropriate level of abstraction. This is not to say we should not think about possible future design modifications. Our designs should be open ended so that future requirements can be satisfied. However, when abstracting interfaces, try to model exactly what needs to be modeled and nothing more.

When designing the interface, try placing yourself in the object's shoes and imagine that the object has a strong preference for privacy. Don't let other objects have access to data about you unless you feel it is in your best interest for them to have it. Don't give them an interface to force you to perform a specific task unless you are certain you want them to be able to do that to you.

Composition

So far, we learned to design systems as a group of interacting objects, where each interaction involves viewing objects at an appropriate level of abstraction. But we don't know yet how to create these levels of abstraction. There are a variety of ways to do this. But even most design patterns rely on two basic object-oriented principles known as **composition** and **inheritance**. Composition is simpler, so let's start with it.

Composition is the act of collecting several objects together to create a new one. Composition is usually a good choice when one object is part of another object. We've already seen a first hint of composition in the mechanic example. A car is composed of an engine, transmission, starter, headlights, and windshield, among numerous other parts. The engine, in turn, is composed of pistons, a crank shaft, and valves. In this example, composition is a good way to provide levels of abstraction. The car object can provide the interface required by a driver, while also providing access to its component parts, which offers the deeper level of abstraction suitable for a mechanic. Those component parts can, of course, be further broken down if the mechanic needs more information to diagnose a problem or tune the engine.

This is a common introductory example of composition, but it's not overly useful when it comes to designing computer systems. Physical objects are easy to break into component objects. People have been doing this at least since the ancient Greeks originally postulated that atoms were the smallest units of matter (they, of course, didn't have access to particle accelerators). Computer systems are generally less complicated than physical objects, yet identifying the component objects in such systems does not happen as naturally.

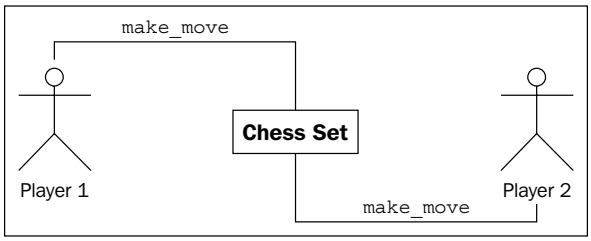
The objects in an object-oriented system occasionally represent physical objects such as people, books, or telephones. More often, however, they represent abstract ideas. People have names, books have titles, and telephones are used to make calls. Calls, titles, accounts, names, appointments, and payments are not usually considered objects in the physical world, but they are all frequently-modeled components in computer systems.

Let's try modeling a more computer-oriented example to see composition in action. We'll be looking at the design of a computerized chess game. This was a very popular pastime among academics in the 80s and 90s. People were predicting that computers would one day be able to defeat a human chess master. When this happened in 1997 (IBM's Deep Blue defeated world chess champion, Gary Kasparov), interest in the problem waned, although there are still contests between computer and human chess players. (The computers usually win.)

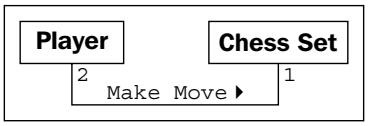
As a basic, high-level analysis, a game of chess is played between two players, using a chess set featuring a board containing sixty-four positions in an 8 X 8 grid. The board can have two sets of sixteen pieces that can be moved, in alternating turns by the two players in different ways. Each piece can take other pieces. The board will be required to draw itself on the computer screen after each turn.

I've identified some of the possible objects in the description using *italics*, and a few key methods using **bold**. This is a common first step in turning an object-oriented analysis into a design. At this point, to emphasize composition, we'll focus on the board, without worrying too much about the players or the different types of pieces.

Let's start at the highest level of abstraction possible. We have two players interacting with a chess set by taking turns making moves:



What is this? It doesn't quite look like our earlier class diagrams. That's because it isn't a class diagram! This is an **object diagram**, also called an instance diagram. It describes the system at a specific state in time, and is describing specific instances of objects, not the interaction between classes. Remember, both players are members of the same class, so the class diagram looks a little different:



The diagram shows that exactly two players can interact with one chess set. It also indicates that any one player can be playing with only one chess set at a time.

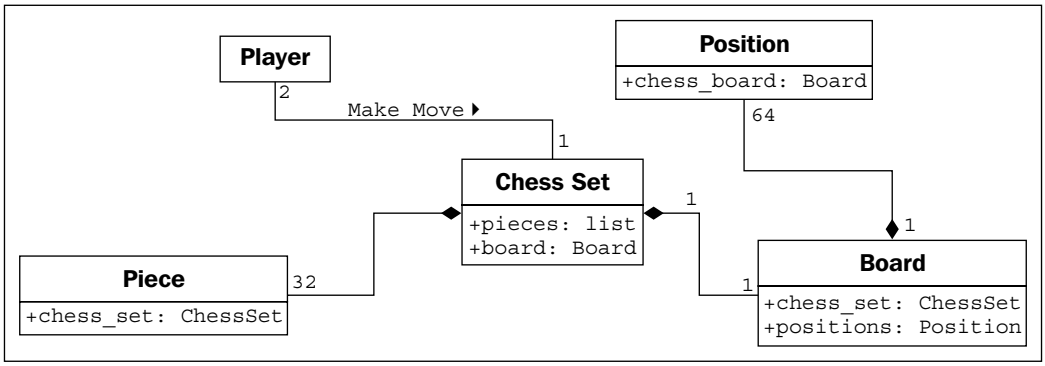
However, we're discussing composition, not UML, so let's think about what the **Chess Set** is composed of. We don't care what the player is composed of at this time. We can assume that the player has a heart and brain, among other organs, but these are irrelevant to our model. Indeed, there is nothing stopping said player from being Deep Blue itself, which has neither a heart nor a brain.

The chess set, then, is composed of a board and 32 pieces. The board further comprises 64 positions. You could argue that pieces are not part of the chess set because you could replace the pieces in a chess set with a different set of pieces. While this is unlikely or impossible in a computerized version of chess, it introduces us to **aggregation**.

Aggregation is almost exactly like composition. The difference is that aggregate objects can exist independently. It would be impossible for a position to be associated with a different chess board, so we say the board is composed of positions. But the pieces, which might exist independently of the chess set, are said to be in an aggregate relationship with that set.

Another way to differentiate between aggregation and composition is to think about the lifespan of the object. If the composite (outside) object controls when the related (inside) objects are created and destroyed, composition is most suitable. If the related object is created independently of the composite object, or can outlast that object, an aggregate relationship makes more sense. Also, keep in mind that composition is aggregation; aggregation is simply a more general form of composition. Any composite relationship is also an aggregate relationship, but not vice versa.

Let's describe our current chess set composition and add some attributes to the objects to hold the composite relationships:



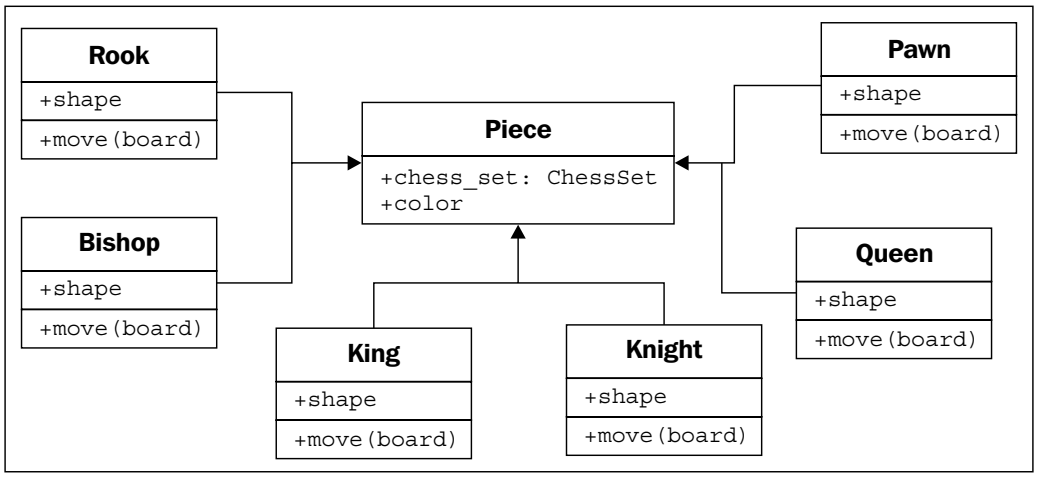
The composition relationship is represented in UML as a solid diamond. The hollow diamond represents the aggregate relationship. You'll notice that the board and pieces are stored as part of the chess set in exactly the same way a reference to them is stored as an attribute on the chess set. This shows that, once again, in practice, the distinction between aggregation and composition is often irrelevant once you get past the design stage. When implemented, they behave in much the same way. However, it can help to differentiate between the two when your team is discussing how the different objects interact. Often, you can treat them as the same thing, but when you need to distinguish between them, it's great to know the difference (this is abstraction at work).

Inheritance

We discussed three types of relationships between objects: association, composition, and aggregation. However, we have not fully specified our chess set, and these tools don't seem to give us all the power we need. We discussed the possibility that a player might be a human or it might be a piece of software featuring artificial intelligence. It doesn't seem right to say that a player is *associated* with a human, or that the artificial intelligence implementation is *part of* the player object. What we really need is the ability to say that "Deep Blue *is a* player" or that "Gary Kasparov *is a* player".

The *is a* relationship is formed by **inheritance**. Inheritance is the most famous, well-known, and over-used relationship in object-oriented programming. Inheritance is sort of like a family tree. My grandfather's last name was Phillips and my father inherited that name. I inherited it from him (along with blue eyes and a penchant for writing). In object-oriented programming, instead of inheriting features and behaviors from a person, one class can inherit attributes and methods from another class.

For example, there are 32 chess pieces in our chess set, but there are only six different types of pieces (pawns, rooks, bishops, knights, king, and queen), each of which behaves differently when it is moved. All of these classes of piece have properties, such as color and the chess set they are part of, but they also have unique shapes when drawn on the chess board, and make different moves. Let's see how the six types of pieces can inherit from a **Piece** class:



The hollow arrows indicate that the individual classes of pieces inherit from the **Piece** class. All the subtypes automatically have a **chess_set** and **color** attribute inherited from the base class. Each piece provides a different shape property (to be drawn on the screen when rendering the board), and a different **move** method to move the piece to a new position on the board at each turn.

We actually know that all subclasses of the **Piece** class need to have a **move** method; otherwise, when the board tries to move the piece, it will get confused. It is possible that we would want to create a new version of the game of chess that has one additional piece (the wizard). Our current design allows us to design this piece without giving it a **move** method. The board would then choke when it asked the piece to move itself.

We can implement this by creating a dummy move method on the **Piece** class. The subclasses can then **override** this method with a more specific implementation. The default implementation might, for example, pop up an error message that says: **That piece cannot be moved**.

Overriding methods in subtypes allows very powerful object-oriented systems to be developed. For example, if we wanted to implement a player class with artificial intelligence, we might provide a `calculate_move` method that takes a **Board** object and decides which piece to move where. A very basic class might randomly choose a piece and direction and move it accordingly. We could then override this method in a subclass with the Deep Blue implementation. The first class would be suitable for play against a raw beginner, the latter would challenge a grand master. The important thing is that other methods in the class, such as the ones that inform the board as to which move was chose need not be changed; this implementation can be shared between the two classes.

In the case of chess pieces, it doesn't really make sense to provide a default implementation of the move method. All we need to do is specify that the move method is required in any subclasses. This can be done by making **Piece** an **abstract class** with the move method declared **abstract**. Abstract methods basically say, "We demand this method exist in any non-abstract subclass, but we are declining to specify an implementation in this class."

Indeed, it is possible to make a class that does not implement any methods at all. Such a class would simply tell us what the class should do, but provides absolutely no advice on how to do it. In object-oriented parlance, such classes are called **interfaces**.

Inheritance provides abstraction

Let's explore the longest word in object-oriented argot. **Polymorphism** is the ability to treat a class differently depending on which subclass is implemented. We've already seen it in action with the pieces system we've described. If we took the design a bit further, we'd probably see that the **Board** object can accept a move from the player and call the **move** function on the piece. The board need not ever know what type of piece it is dealing with. All it has to do is call the **move** method, and the proper subclass will take care of moving it as a **Knight** or a **Pawn**.

Polymorphism is pretty cool, but it is a word that is rarely used in Python programming. Python goes an extra step past allowing a subclass of an object to be treated like a parent class. A board implemented in Python could take any object that has a **move** method, whether it is a bishop piece, a car, or a duck. When **move** is called, the **Bishop** will move diagonally on the board, the car will drive someplace, and the duck will swim or fly, depending on its mood.

This sort of polymorphism in Python is typically referred to as **duck typing**: "If it walks like a duck or swims like a duck, it's a duck". We don't care if it really *is a* duck (inheritance), only that it swims or walks. Geese and swans might easily be able to provide the duck-like behavior we are looking for. This allows future designers to create new types of birds without actually specifying an inheritance hierarchy for aquatic birds. It also allows them to create completely different drop-in behaviors that the original designers never planned for. For example, future designers might be able to make a walking, swimming penguin that works with the same interface without ever suggesting that penguins are ducks.

Multiple inheritance

When we think of inheritance in our own family tree, we can see that we inherit features from more than just one parent. When strangers tell a proud mother that her son has, "his fathers eyes", she will typically respond along the lines of, "yes, but he got my nose."

Object-oriented design can also feature such **multiple inheritance**, which allows a subclass to inherit functionality from multiple parent classes. In practice, multiple inheritance can be a tricky business, and some programming languages (most notably, Java) strictly prohibit it. However, multiple inheritance can have its uses. Most often, it can be used to create objects that have two distinct sets of behaviors. For example, an object designed to connect to a scanner and send a fax of the scanned document might be created by inheriting from two separate `scanner` and `faxer` objects.

As long as two classes have distinct interfaces, it is not normally harmful for a subclass to inherit from both of them. However, it gets messy if we inherit from two classes that provide overlapping interfaces. For example, if we have a motorcycle class that has a `move` method, and a boat class also featuring a `move` method, and we want to merge them into the ultimate amphibious vehicle, how does the resulting class know what to do when we call `move`? At the design level, this needs to be explained, and at the implementation level, each programming language has different ways of deciding which parent class's method is called, or in what order.

Often, the best way to deal with it is to avoid it. If you have a design showing up like this, you're *probably* doing it wrong. Take a step back, analyze the system again, and see if you can remove the multiple inheritance relationship in favor of some other association or composite design.

Inheritance is a very powerful tool for extending behavior. It is also one of the most marketable advancements of object-oriented design over earlier paradigms. Therefore, it is often the first tool that object-oriented programmers reach for. However, it is important to recognize that owning a hammer does not turn screws into nails. Inheritance is the perfect solution for obvious *is a* relationships, but it can be abused. Programmers often use inheritance to share code between two kinds of objects that are only distantly related, with no *is a* relationship in sight. While this is not necessarily a bad design, it is a terrific opportunity to ask just why they decided to design it that way, and whether a different relationship or design pattern would have been more suitable.

Case study

Let's tie all our new object-oriented knowledge together by going through a few iterations of object-oriented design on a somewhat real-world example. The system we'll be modeling is a library catalog. Libraries have been tracking their inventory for centuries, originally using card catalogs, and more recently, electronic inventories. Modern libraries have web-based catalogs that we can query from our homes.

Let's start with an analysis. The local librarian has asked us to write a new card catalog program because their ancient DOS-based program is ugly and out of date. That doesn't give us much detail, but before we start asking for more information, let's consider what we already know about library catalogs.

Catalogs contain lists of books. People search them to find books on certain subjects, with specific titles, or by a particular author. Books can be uniquely identified by an **International Standard Book Number** (**ISBN**). Each module has a **Dewey Decimal System** (**DDS**) number assigned to help find it on a particular shelf.

This simple analysis tells us some of the obvious objects in the system. We quickly identify **Book** as the most important object, with several attributes already mentioned, such as author, title, subject, ISBN, and DDS number, and catalog as a sort of manager for books.

We also notice a few other objects that may or may not need to be modeled in the system. For cataloging purposes, all we need to search a module by author is an `author_name` attribute on the module. However, authors are also objects, and we might want to store some other data about the author. As we ponder this, we might remember that some books have multiple authors. Suddenly, the idea of having a single `author_name` attribute on objects seems a bit silly. A list of authors associated with each module is clearly a better idea.

The relationship between author and module is clearly association, since you would never say, "a module is an author" (it's not inheritance), and saying "a module has an author", though grammatically correct, does not imply that authors are part of books (it's not aggregation). Indeed, any one author may be associated with multiple books.

We should also pay attention to the noun (nouns are always good candidates for objects) *shelf*. Is a shelf an object that needs to be modeled in a cataloging system? How do we identify an individual shelf? What happens if a module is stored at the end of one shelf, and later moved to the beginning of the next shelf because another module was inserted in the previous shelf?

DDS was designed to help locate physical books in a library. As such, storing a DDS attribute with the module should be enough to locate it, regardless of which shelf it is stored on. So we can, at least for the moment, remove shelf from our list of contending objects.

Another questionable object in the system is the user. Do we need to know anything about a specific user, such as their name, address, or list of overdue books? So far, the librarian has told us only that they want a catalog; they said nothing about tracking subscriptions or overdue notices. In the back of our minds, we also note that authors and users are both specific kinds of people; there might be a useful inheritance relationship here in the future.

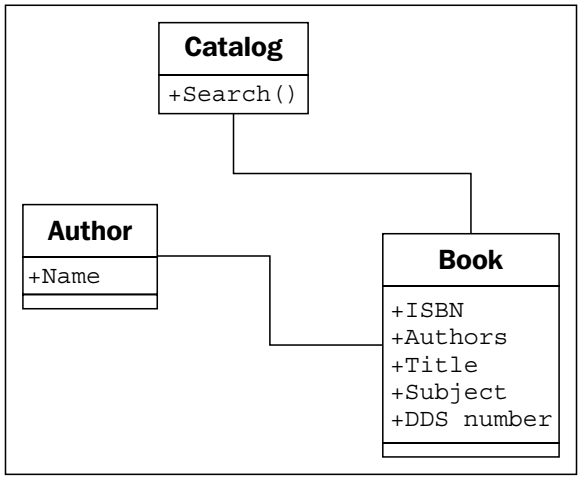
For cataloging purposes, we decide we don't need to identify the user for now. We can assume that a user will be searching the catalog, but we don't have to actively model them in the system, beyond providing an interface that allows them to search.

We have identified a few attributes on the module, but what properties does a catalog have? Does any one library have more than one catalog? Do we need to uniquely identify them? Obviously, the catalog has to have a collection of the books it contains, somehow, but this list is probably not part of the public interface.

What about behaviors? The catalog clearly needs a search method, possibly separate ones for authors, titles, and subjects. Are there any behaviors on books? Would it need a preview method? Or could preview be identified by a first pages attribute instead of a method?

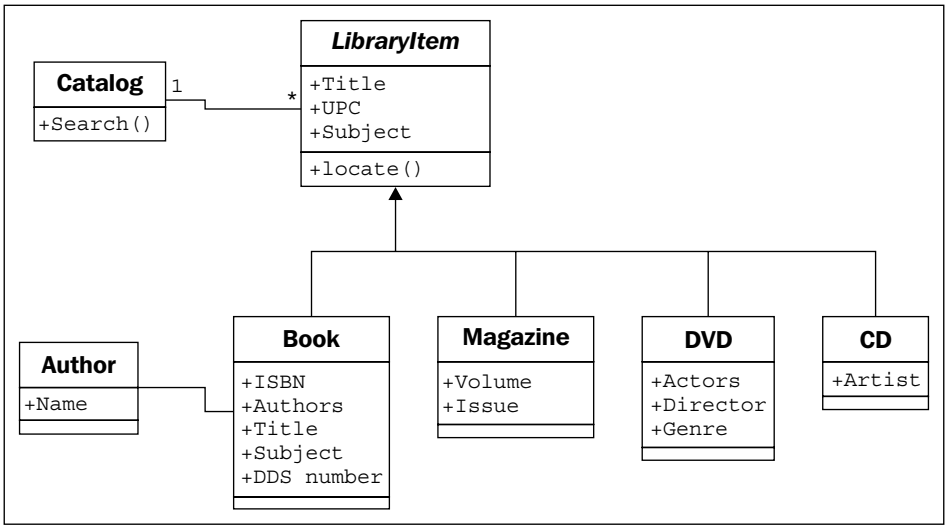
The questions in the preceding discussion are all part of the object-oriented analysis phase. But intermixed with the questions, we have already identified a few key objects that are part of the design. Indeed, what you have just seen are several microiterations between analysis and design.

Likely, these iterations would all occur in an initial meeting with the librarian. Before this meeting, however, we can already sketch out a most basic design for the objects we have concretely identified:



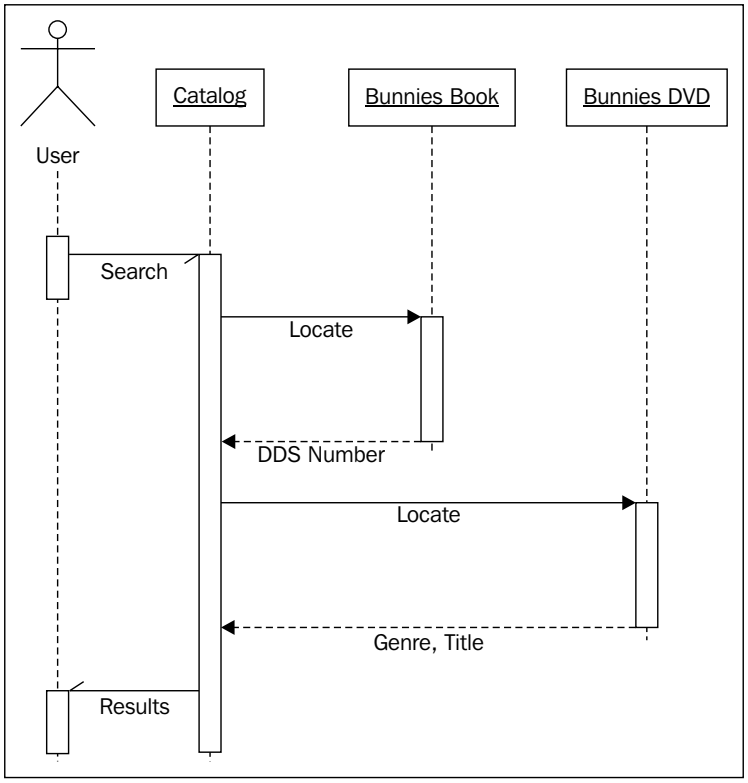
Armed with this basic diagram and a pencil to interactively improve it, we meet up with the librarian. They tell us that this is a good start, but libraries don't serve only books, they also have DVDs, magazines, and CDs, none of which have an ISBN or DDS number. All of these types of items can be uniquely identified by a UPC number though. We remind the librarian that they have to find the items on the shelf, and these items probably aren't organized by UPC. The librarian explains that each type is organized in a different way. The CDs are mostly audio books, and they only have a couple of dozen in stock, so they are organized by the author's last name. DVDs are divided into genre and further organized by title. Magazines are organized by title and then refined by the volume and issue number. Books are, as we had guessed, organized by the DDS number.

With no previous object-oriented design experience, we might consider adding separate lists of DVDs, CDs, magazines, and books to our catalog, and search each one in turn. The trouble is, except for certain extended attributes, and identifying the physical location of the item, these items all behave as much the same. This is a job for inheritance! We quickly update our UML diagram:



The librarian understands the gist of our sketched diagram, but is a bit confused by the **locate** functionality. We explain using a specific use case where the user is searching for the word "bunnies". The user first sends a search request to the catalog. The catalog queries its internal list of items and finds a module and a DVD with "bunnies" in the title. At this point, the catalog doesn't care if it is holding a DVD, module, CD, or magazine; all items are the same, as far as the catalog is concerned. However, the user wants to know how to find the physical items, so the catalog would be remiss if it simply returned a list of titles. So, it calls the **locate** method on the two items it has uncovered. The module's **locate** method returns a DDS number that can be used to find the shelf holding the module. The DVD is located by returning the genre and title of the DVD. The user can then visit the DVD section, find the section containing that genre, and find the specific DVD as sorted by the titles.

As we explain, we sketch a UML **sequence diagram** explaining how the various objects are communicating:



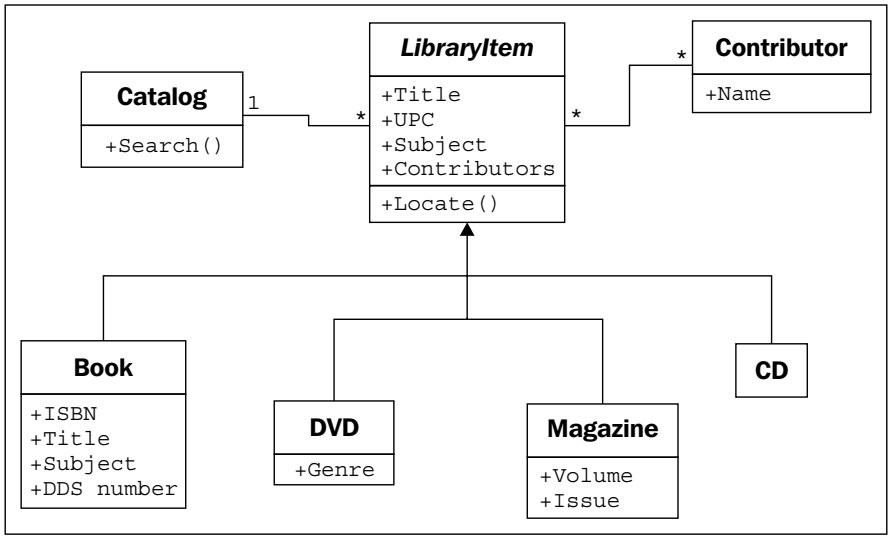
Where, class diagrams describe the relationships between classes, and sequence diagrams describe specific sequences of messages passed between objects. The dashed line hanging from each object is a **lifeline** describing the lifetime of the object. The wider boxes on each lifeline represent active processing in that object (where there's no box, the object is basically sitting idle, waiting for something to happen). The horizontal arrows between the lifelines indicate specific messages. The solid arrows represent methods being called, while the dashed arrows with solid heads represent the method return values.

The half arrowheads indicate asynchronous messages sent to or from an object. An asynchronous message typically means the first object calls a method on the second object, which returns immediately. After some processing, the second object calls a method on the first object to give it a value. This is in contrast to normal method calls, which do the processing in the method, and return a value immediately.

Sequence diagrams, like all UML diagrams, are best used only when they are needed. There is no point in drawing a UML diagram for the sake of drawing a diagram. However, when you need to communicate a series of interactions between two objects, the sequence diagram is a very useful tool.

Unfortunately, our class diagram so far is still a messy design. We notice that actors on DVDs and artists on CDs are all types of people, but are being treated differently from the module authors. The librarian also reminds us that most of their CDs are audio books, which have authors instead of artists.

How can we deal with different kinds of people that contribute to a title? An obvious implementation is to create a `Person` class with the person's name and other relevant details, and then create subclasses of this for the artists, authors, and actors. However, is inheritance really necessary here? For searching and cataloging purposes, we don't really care that acting and writing are two very different activities. If we were doing an economic simulation, it would make sense to give separate actor and author classes, and different `calculate_income` and `perform_job` methods, but for cataloging purposes, it is probably enough to know how the person contributed to the item. We recognize that all items have one or more `Contributor` objects, so we move the author relationship from the module to its parent class:

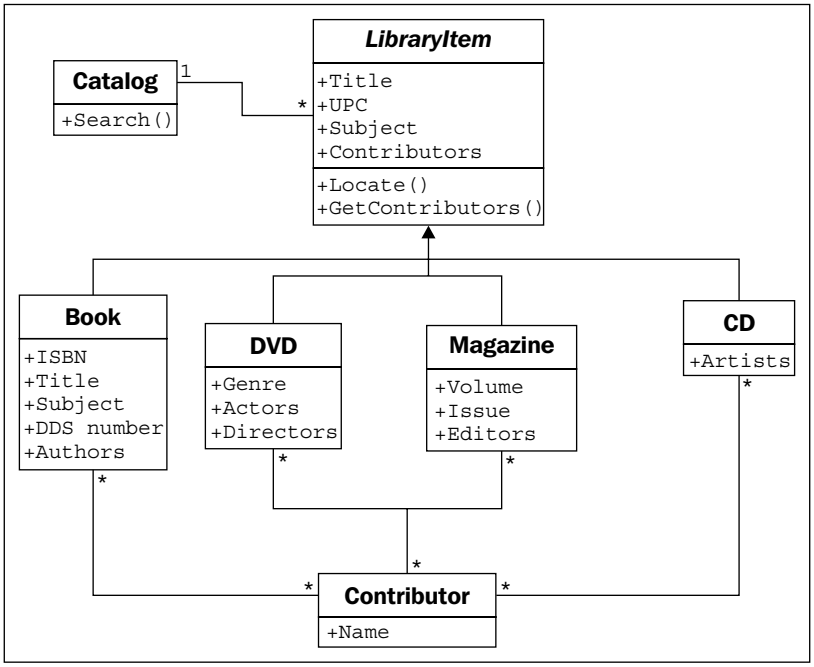


The multiplicity of the **Contributor/LibraryItem** relationship is **many-to-many**, as indicated by the * character at both ends of one relationship. Any one library item might have more than one contributor (for example, several actors and a director on a DVD). And many authors write many books, so they would be attached to multiple library items.

This little change, while it looks a bit cleaner and simpler, has lost some vital information. We can still tell who contributed to a specific library item, but we don't know how they contributed. Were they the director or an actor? Did they write the audio module, or were they the voice that narrated the module?

It would be nice if we could just add a `contributor_type` attribute on the **Contributor** class, but this will fall apart when dealing with multitalented people who have both authored books and directed movies.

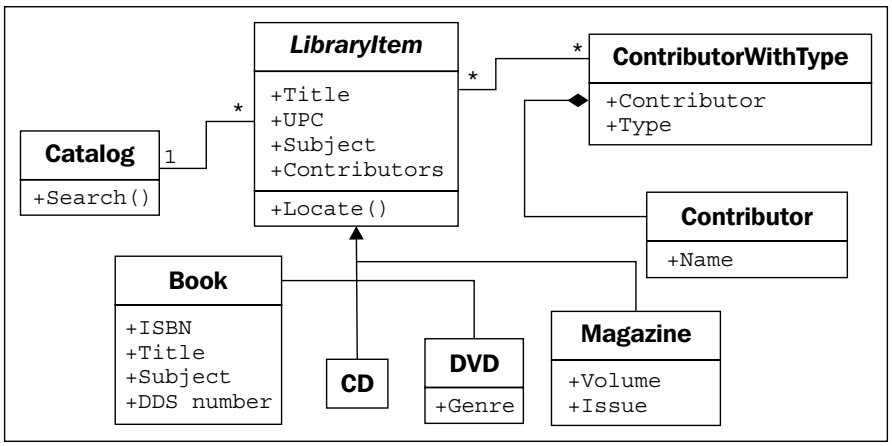
One option is to add attributes to each of our **LibraryItem** subclasses that hold the information we need, such as **Author** on **Book**, or **Artist** on **CD**, and then make the relationship to those properties all point to the **Contributor** class. The problem with this is that we lose a lot of polymorphic elegance. If we want to list the contributors to an item, we have to look for specific attributes on that item, such as **Authors** or **Actors**. We can alleviate this by adding a **GetContributors** method on the **LibraryItem** class that subclasses can override. Then the catalog never has to know what attributes the objects are querying; we've abstracted the public interface:



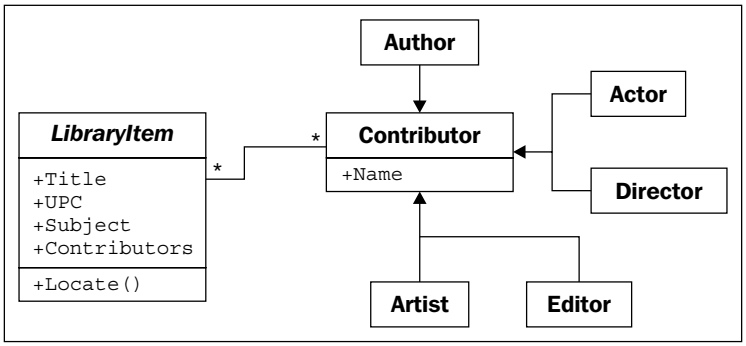
Just looking at this class diagram, it feels like we are doing something wrong. It is bulky and fragile. It may do everything we need, but it feels like it will be hard to maintain or extend. There are too many relationships, and too many classes would be affected by modifications to any one class. It looks like spaghetti and meatballs.

Now that we've explored inheritance as an option, and found it wanting, we might look back at our previous composition-based diagram, where **Contributor** was attached directly to **LibraryItem**. With some thought, we can see that we actually only need to add one more relationship to a brand-new class to identify the type of contributor. This is an important step in object-oriented design. We are now adding a class to the design that is intended to *support* the other objects, rather than modeling any part of the initial requirements. We are **refactoring** the design to facilitate the objects in the system, rather than objects in real life. Refactoring is an essential process in the maintenance of a program or design. The goal of refactoring is to improve the design by moving code around, removing duplicate code or complex relationships in favor of simpler, more elegant designs.

This new class is composed of a **Contributor** and an extra attribute identifying the type of contribution the person has made to the given **LibraryItem**. There can be many such contributions to a particular **LibraryItem**, and one contributor can contribute in the same way to different items. The diagram communicates this design very well:



At first, this composition relationship looks less natural than the inheritance-based relationships. However, it has the advantage of allowing us to add new types of contributions without adding a new class to the design. Inheritance is most useful when the subclasses have some kind of specialization. Specialization is creating or changing attributes or behaviors on the subclass to make it somehow different from the parent class. It seems silly to create a bunch of empty classes solely for identifying different types of objects (this attitude is less prevalent among Java and other "everything is an object" programmers, but it is common among more practical Python designers). If we look at the inheritance version of the diagram, we can see a bunch of subclasses that don't actually do anything:



Sometimes it is important to recognize when not to use object-oriented principles. This example of when not to use inheritance is a good reminder that objects are just tools, and not rules.

Your Coding Challenge

Well as such, I'm not about to assign you a bunch of fake object-oriented analysis problems to create designs for bunch of fake object-oriented problems to analyze and design. Instead, I want to give you some thoughts that you can apply to your own projects. If you have previous object-oriented experience, you won't need to put much effort into these. However, they are useful mental exercises if you've been using Python for a while, but never really cared about all that class stuff.

First, think about a recent programming project you've completed. Identify the most prominent object in the design. Try to think of as many attributes for this object as possible. Did it have: Color? Weight? Size? Profit? Cost? Name? ID number? Price? Style? Think about the attribute types. Were they primitives or classes? Were some of those attributes actually behaviors in disguise? Sometimes what looks like data is actually calculated from other data on the object, and you can use a method to do those calculations. What other methods or behaviors did the object have? Which objects called those methods? What kinds of relationships did they have with this object?

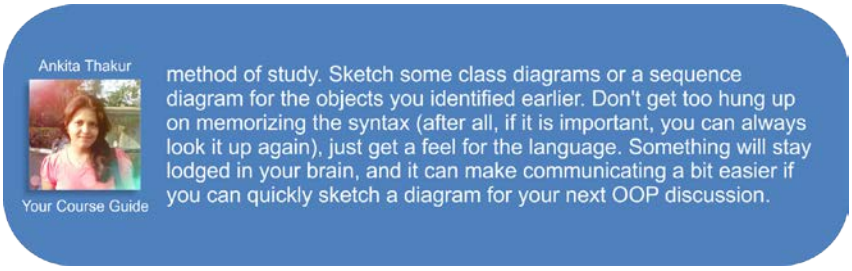
Ankita Thakur

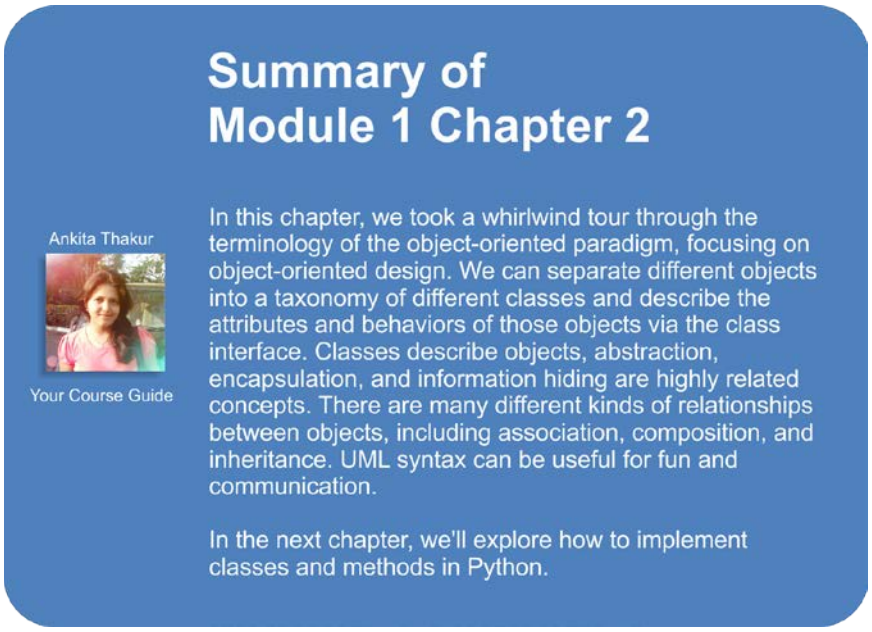
Your Course Guide

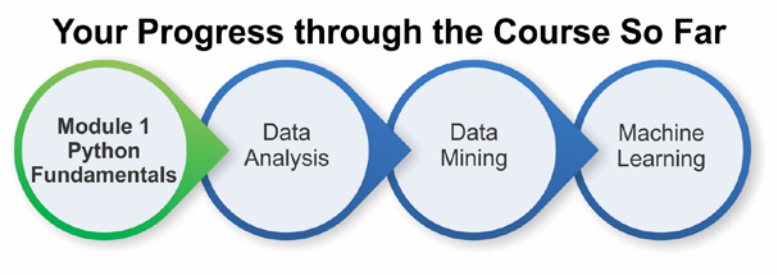
Now, think about an upcoming project. It doesn't matter what the project is; it might be a fun free-time project or a multimillion dollar contract. It doesn't have to be a complete application; it could just be one subsystem. Perform a basic object-oriented analysis. Identify the requirements and the interacting objects. Sketch out a class diagram featuring the highest level of abstraction on that system. Identify the major interacting objects. Identify minor supporting objects. Go into detail for the attributes and methods of some of the most interesting ones. Take different objects to different levels of abstraction. Look for places you can use inheritance or composition. Look for places you should avoid inheritance.

The goal is not to design a system (although you're certainly welcome to do so if inclination meets both ambition and available time). The goal is to think about object-oriented designs. Focusing on projects that you have worked on, or are expecting to work on in the future, simply makes it real.

Now, visit your favorite search engine and look up some tutorials on UML. There are dozens, so find the one that suits your preferred


Ankita Thakur
Your Course Guide
method of study. Sketch some class diagrams or a sequence diagram for the objects you identified earlier. Don't get too hung up on memorizing the syntax (after all, if it is important, you can always look it up again), just get a feel for the language. Something will stay lodged in your brain, and it can make communicating a bit easier if you can quickly sketch a diagram for your next OOP discussion.


Summary of
Module 1 Chapter 2
Ankita Thakur
Your Course Guide
In this chapter, we took a whirlwind tour through the terminology of the object-oriented paradigm, focusing on object-oriented design. We can separate different objects into a taxonomy of different classes and describe the attributes and behaviors of those objects via the class interface. Classes describe objects, abstraction, encapsulation, and information hiding are highly related concepts. There are many different kinds of relationships between objects, including association, composition, and inheritance. UML syntax can be useful for fun and communication.
In the next chapter, we'll explore how to implement classes and methods in Python.


Your Progress through the Course So Far
Module 1 Python Fundamentals
Data Analysis
Data Mining
Machine Learning

3
Objects in Python

So, we now have a design in hand and are ready to turn that design into a working program! Of course, it doesn't usually happen this way. We'll be seeing examples and hints for good software design throughout the module, but our focus is object-oriented programming. So, let's have a look at the Python syntax that allows us to create object-oriented software.

After completing this chapter, we will understand:

- How to create classes and instantiate objects in Python
- How to add attributes and behaviors to Python objects
- How to organize classes into packages and modules
- How to suggest people don't clobber our data

Creating Python classes

We don't have to write much Python code to realize that Python is a very "clean" language. When we want to do something, we just do it, without having to go through a lot of setup. The ubiquitous "hello world" in Python, as you've likely seen, is only one line.

Similarly, the simplest class in Python 3 looks like this:

```
class MyFirstClass:
    pass
```

There's our first object-oriented program! The class definition starts with the `class` keyword. This is followed by a name (of our choice) identifying the class, and is terminated with a colon.

The class name must follow standard Python variable naming rules (it must start with a letter or underscore, and can only be comprised of letters, underscores, or numbers). In addition, the Python style guide (search the web for "PEP 8") recommends that classes should be named using **CamelCase** notation (start with a capital letter; any subsequent words should also start with a capital).

The class definition line is followed by the class contents indented. As with other Python constructs, indentation is used to delimit the classes, rather than braces or brackets as many other languages use. Use four spaces for indentation unless you have a compelling reason not to (such as fitting in with somebody else's code that uses tabs for indents). Any decent programming editor can be configured to insert four spaces whenever the *Tab* key is pressed.

Since our first class doesn't actually do anything, we simply use the `pass` keyword on the second line to indicate that no further action needs to be taken.

We might think there isn't much we can do with this most basic class, but it does allow us to instantiate objects of that class. We can load the class into the Python 3 interpreter, so we can interactively play with it. To do this, save the class definition mentioned earlier into a file named `first_class.py` and then run the command `python -i first_class.py`. The `-i` argument tells Python to "run the code and then drop to the interactive interpreter". The following interpreter session demonstrates basic interaction with this class:

```
>>> a = MyFirstClass()
>>> b = MyFirstClass()
>>> print(a)
<__main__.MyFirstClass object at 0xb7b7faec>
>>> print(b)
<__main__.MyFirstClass object at 0xb7b7fbac>
>>>
```

This code instantiates two objects from the new class, named `a` and `b`. Creating an instance of a class is a simple matter of typing the class name followed by a pair of parentheses. It looks much like a normal function call, but Python knows we're "calling" a class and not a function, so it understands that its job is to create a new object. When printed, the two objects tell us which class they are and what memory address they live at. Memory addresses aren't used much in Python code, but here, they demonstrate that there are two distinct objects involved.

Adding attributes

Now, we have a basic class, but it's fairly useless. It doesn't contain any data, and it doesn't do anything. What do we have to do to assign an attribute to a given object?

It turns out that we don't have to do anything special in the class definition. We can set arbitrary attributes on an instantiated object using the dot notation:

```
class Point:
    pass

p1 = Point()
p2 = Point()

p1.x = 5
p1.y = 4

p2.x = 3
p2.y = 6

print(p1.x, p1.y)
print(p2.x, p2.y)
```

If we run this code, the two `print` statements at the end tell us the new attribute values on the two objects:

```
5 4
3 6
```

This code creates an empty `Point` class with no data or behaviors. Then it creates two instances of that class and assigns each of those instances `x` and `y` coordinates to identify a point in two dimensions. All we need to do to assign a value to an attribute on an object is use the <*object*>.<*attribute*> = <*value*> syntax. This is sometimes referred to as **dot notation**. The value can be anything: a Python primitive, a built-in data type, or another object. It can even be a function or another class!

Making it do something

Now, having objects with attributes is great, but object-oriented programming is really about the interaction between objects. We're interested in invoking actions that cause things to happen to those attributes. It is time to add behaviors to our classes.

Let's model a couple of actions on our `Point` class. We can start with a method called `reset` that moves the point to the origin (the origin is the point where `x` and `y` are both zero). This is a good introductory action because it doesn't require any parameters:

```
class Point:
    def reset(self):
        self.x = 0
        self.y = 0

p = Point()
p.reset()
print(p.x, p.y)
```

This `print` statement shows us the two zeros on the attributes:

```
0 0
```

A method in Python is formatted identically to a function. It starts with the keyword `def` followed by a space and the name of the method. This is followed by a set of parentheses containing the parameter list (we'll discuss that `self` parameter in just a moment), and terminated with a colon. The next line is indented to contain the statements inside the method. These statements can be arbitrary Python code operating on the object itself and any parameters passed in as the method sees fit.

Talking to yourself

The one difference between methods and normal functions is that all methods have one required argument. This argument is conventionally named `self`; I've never seen a programmer use any other name for this variable (convention is a very powerful thing). There's nothing stopping you, however, from calling it `this` or even `Martha`.

The `self` argument to a method is simply a reference to the object that the method is being invoked on. We can access attributes and methods of that object as if it were any another object. This is exactly what we do inside the `reset` method when we set the `x` and `y` attributes of the `self` object.

Notice that when we call the `p.reset()` method, we do not have to pass the `self` argument into it. Python automatically takes care of this for us. It knows we're calling a method on the `p` object, so it automatically passes that object to the method.

However, the method really is just a function that happens to be on a class. Instead of calling the method on the object, we can invoke the function on the class, explicitly passing our object as the `self` argument:

```
p = Point()
Point.reset(p)
print(p.x, p.y)
```

The output is the same as the previous example because internally, the exact same process has occurred.

What happens if we forget to include the `self` argument in our class definition? Python will bail with an error message:

```
>>> class Point:
...     def reset():
...         pass
...
>>> p = Point()
>>> p.reset()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reset() takes no arguments (1 given)
```

The error message is not as clear as it could be ("You silly fool, you forgot the `self` argument" would be more informative). Just remember that when you see an error message that indicates missing arguments, the first thing to check is whether you forgot `self` in the method definition.

More arguments

So, how do we pass multiple arguments to a method? Let's add a new method that allows us to move a point to an arbitrary position, not just to the origin. We can also include one that accepts another `Point` object as input and returns the distance between them:

```
import math

class Point:
```

```
        def move(self, x, y):
            self.x = x
            self.y = y

        def reset(self):
            self.move(0, 0)

        def calculate_distance(self, other_point):
            return math.sqrt(
                    (self.x - other_point.x)**2 +
                    (self.y - other_point.y)**2)

    # how to use it:
    point1 = Point()
    point2 = Point()

    point1.reset()
    point2.move(5,0)
    print(point2.calculate_distance(point1))
    assert (point2.calculate_distance(point1) ==
            point1.calculate_distance(point2))
    point1.move(3,4)
    print(point1.calculate_distance(point2))
    print(point1.calculate_distance(point1))
```

The `print` statements at the end give us the following output:

```
5.0
4.472135955
0.0
```

A lot has happened here. The class now has three methods. The `move` method accepts two arguments, `x` and `y`, and sets the values on the `self` object, much like the old `reset` method from the previous example. The old `reset` method now calls `move`, since a reset is just a move to a specific known location.

The `calculate_distance` method uses the not-too-complex Pythagorean theorem to calculate the distance between two points. I hope you understand the math (`**2` means squared, and `math.sqrt` calculates a square root), but it's not a requirement for our current focus, learning how to write methods.

The sample code at the end of the preceding example shows how to call a method with arguments: simply include the arguments inside the parentheses, and use the same dot notation to access the method. I just picked some random positions to test the methods. The test code calls each method and prints the results on the console. The `assert` function is a simple test tool; the program will bail if the statement after `assert` is `False` (or zero, empty, or `None`). In this case, we use it to ensure that the distance is the same regardless of which point called the other point's `calculate_distance` method.

Initializing the object

If we don't explicitly set the `x` and `y` positions on our `Point` object, either using `move` or by accessing them directly, we have a broken point with no real position. What will happen when we try to access it?

Well, let's just try it and see. "Try it and see" is an extremely useful tool for Python study. Open up your interactive interpreter and type away. The following interactive session shows what happens if we try to access a missing attribute. If you saved the previous example as a file or are using the examples distributed with the module, you can load it into the Python interpreter with the command `python -i filename.py`:

```
>>> point = Point()
>>> point.x = 5
>>> print(point.x)
5
>>> print(point.y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Point' object has no attribute 'y'
```

Well, at least it threw a useful exception. We'll cover exceptions in detail in *Chapter 4*, *Expecting the Unexpected*. You've probably seen them before (especially the ubiquitous **SyntaxError**, which means you typed something incorrectly!). At this point, simply be aware that it means something went wrong.

The output is useful for debugging. In the interactive interpreter, it tells us the error occurred at **line 1**, which is only partially true (in an interactive session, only one line is executed at a time). If we were running a script in a file, it would tell us the exact line number, making it easy to find the offending code. In addition, it tells us the error is an `AttributeError`, and gives a helpful message telling us what that error means.

We can catch and recover from this error, but in this case, it feels like we should have specified some sort of default value. Perhaps every new object should be `reset()` by default, or maybe it would be nice if we could force the user to tell us what those positions should be when they create the object.

Most object-oriented programming languages have the concept of a **constructor**, a special method that creates and initializes the object when it is created. Python is a little different; it has a constructor *and* an initializer. The constructor function is rarely used unless you're doing something exotic. So, we'll start our discussion with the initialization method.

The Python initialization method is the same as any other method, except it has a special name, `__init__`. The leading and trailing double underscores mean this is a special method that the Python interpreter will treat as a special case.

Never name a function of your own with leading and trailing double underscores. It may mean nothing to Python, but there's always the possibility that the designers of Python will add a function that has a special purpose with that name in the future, and when they do, your code will break.

Let's start with an initialization function on our `Point` class that requires the user to supply `x` and `y` coordinates when the `Point` object is instantiated:

```
class Point:
    def __init__(self, x, y):
        self.move(x, y)

    def move(self, x, y):
        self.x = x
        self.y = y

    def reset(self):
        self.move(0, 0)

# Constructing a Point
point = Point(3, 5)
print(point.x, point.y)
```

Now, our point can never go without a `y` coordinate! If we try to construct a point without including the proper initialization parameters, it will fail with a **not enough arguments** error similar to the one we received earlier when we forgot the `self` argument.

What if we don't want to make those two arguments required? Well, then we can use the same syntax Python functions use to provide default arguments. The keyword argument syntax appends an equals sign after each variable name. If the calling object does not provide this argument, then the default argument is used instead. The variables will still be available to the function, but they will have the values specified in the argument list. Here's an example:

```
class Point:
    def __init__(self, x=0, y=0):
        self.move(x, y)
```

Most of the time, we put our initialization statements in an `__init__` function. But as mentioned earlier, Python has a constructor in addition to its initialization function. You may never need to use the other Python constructor, but it helps to know it exists, so we'll cover it briefly.

The constructor function is called `__new__` as opposed to `__init__`, and accepts exactly one argument; the class that is being constructed (it is called *before* the object is constructed, so there is no `self` argument). It also has to return the newly created object. This has interesting possibilities when it comes to the complicated art of metaprogramming, but is not very useful in day-to-day programming. In practice, you will rarely, if ever, need to use `__new__` and `__init__` will be sufficient.

Explaining yourself

Python is an extremely easy-to-read programming language; some might say it is self-documenting. However, when doing object-oriented programming, it is important to write API documentation that clearly summarizes what each object and method does. Keeping documentation up-to-date is difficult; the best way to do it is to write it right into our code.

Python supports this through the use of **docstrings**. Each class, function, or method header can have a standard Python string as the first line following the definition (the line that ends in a colon). This line should be indented the same as the following code.

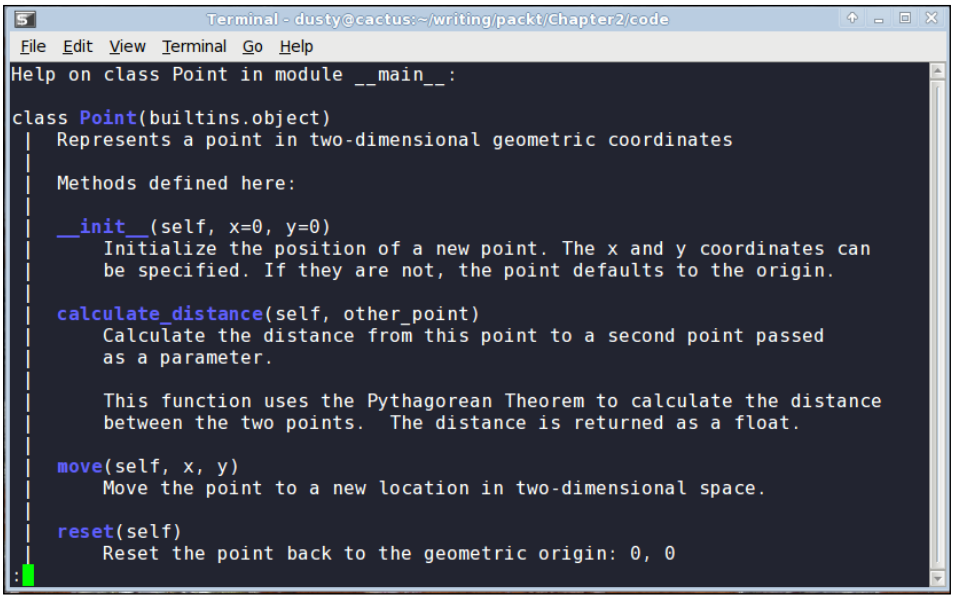
Docstrings are simply Python strings enclosed with apostrophe (`'`) or quote (`"`) characters. Often, docstrings are quite long and span multiple lines (the style guide suggests that the line length should not exceed 80 characters), which can be formatted as multi-line strings, enclosed in matching triple apostrophe (`'''`) or triple quote (`"""`) characters.

A docstring should clearly and concisely summarize the purpose of the class or method it is describing. It should explain any parameters whose usage is not immediately obvious, and is also a good place to include short examples of how to use the API. Any caveats or problems an unsuspecting user of the API should be aware of should also be noted.

To illustrate the use of docstrings, we will end this section with our completely documented `Point` class:

```
import math

class Point:
    'Represents a point in two-dimensional geometric coordinates'

    def __init__(self, x=0, y=0):
        '''Initialize the position of a new point. The x and y
coordinates can be specified. If they are not, the
           point defaults to the origin.'''
        self.move(x, y)

    def move(self, x, y):
        "Move the point to a new location in 2D space."
        self.x = x
        self.y = y

    def reset(self):
        'Reset the point back to the geometric origin: 0, 0'
        self.move(0, 0)

    def calculate_distance(self, other_point):
        """Calculate the distance from this point to a second
        point passed as a parameter.

        This function uses the Pythagorean Theorem to calculate
the distance between the two points. The distance is
        returned as a float."""

        return math.sqrt(
                (self.x - other_point.x)**2 +
                (self.y - other_point.y)**2)
```

Try typing or loading (remember, it's `python -i filename.py`) this file into the interactive interpreter. Then, enter `help(Point)<enter>` at the Python prompt. You should see nicely formatted documentation for the class, as shown in the following screenshot:



Modules and packages

Now, we know how to create classes and instantiate objects, but how do we organize them? For small programs, we can just put all our classes into one file and add a little script at the end of the file to start them interacting. However, as our projects grow, it can become difficult to find the one class that needs to be edited among the many classes we've defined. This is where **modules** come in. Modules are simply Python files, nothing more. The single file in our small program is a module. Two Python files are two modules. If we have two files in the same folder, we can load a class from one module for use in the other module.

For example, if we are building an e-commerce system, we will likely be storing a lot of data in a database. We can put all the classes and functions related to database access into a separate file (we'll call it something sensible: `database.py`). Then, our other modules (for example, customer models, product information, and inventory) can import classes from that module in order to access the database.

The `import` statement is used for importing modules or specific classes or functions from modules. We've already seen an example of this in our `Point` class in the previous section. We used the `import` statement to get Python's built-in `math` module and use its `sqrt` function in our `distance` calculation.

Here's a concrete example. Assume we have a module called `database.py` that contains a class called `Database`, and a second module called `products.py` that is responsible for product-related queries. At this point, we don't need to think too much about the contents of these files. What we know is that `products.py` needs to instantiate the `Database` class from `database.py` so that it can execute queries on the product table in the database.

There are several variations on the `import` statement syntax that can be used to access the class:

```
import database
db = database.Database()
# Do queries on db
```

This version imports the `database` module into the `products` namespace (the list of names currently accessible in a module or function), so any class or function in the `database` module can be accessed using the `database.<something>` notation. Alternatively, we can import just the one class we need using the `from...import` syntax:

```
from database import Database
db = Database()
# Do queries on db
```

If, for some reason, `products` already has a class called `Database`, and we don't want the two names to be confused, we can rename the class when used inside the `products` module:

```
from database import Database as DB
db = DB()
# Do queries on db
```

We can also import multiple items in one statement. If our `database` module also contains a `Query` class, we can import both classes using:

```
from database import Database, Query
```

Some sources say that we can import all classes and functions from the `database` module using this syntax:

```
from database import *
```

Don't do this. Every experienced Python programmer will tell you that you should never use this syntax. They'll use obscure justifications such as "it clutters up the namespace", which doesn't make much sense to beginners. One way to learn why to avoid this syntax is to use it and try to understand your code two years later. But we can save some time and two years of poorly written code with a quick explanation now!

When we explicitly import the `database` class at the top of our file using `from database import Database`, we can easily see where the `Database` class comes from. We might use `db = Database()` 400 lines later in the file, and we can quickly look at the imports to see where that `Database` class came from. Then, if we need clarification as to how to use the `Database` class, we can visit the original file (or import the module in the interactive interpreter and use the `help(database.Database)` command). However, if we use the `from database import *` syntax, it takes a lot longer to find where that class is located. Code maintenance becomes a nightmare.

In addition, most editors are able to provide extra functionality, such as reliable code completion, the ability to jump to the definition of a class, or inline documentation, if normal imports are used. The `import *` syntax usually completely destroys their ability to do this reliably.

Finally, using the `import *` syntax can bring unexpected objects into our local namespace. Sure, it will import all the classes and functions defined in the module being imported from, but it will also import any classes or modules that were themselves imported into that file!

Every name used in a module should come from a well-specified place, whether it is defined in that module, or explicitly imported from another module. There should be no magic variables that seem to come out of thin air. We should *always* be able to immediately identify where the names in our current namespace originated. I promise that if you use this evil syntax, you will one day have extremely frustrating moments of "where on earth can this class be coming from?".

Organizing the modules

As a project grows into a collection of more and more modules, we may find that we want to add another level of abstraction, some kind of nested hierarchy on our modules' levels. However, we can't put modules inside modules; one file can hold only one file after all, and modules are nothing more than Python files.

Files, however, can go in folders and so can modules. A **package** is a collection of modules in a folder. The name of the package is the name of the folder. All we need to do to tell Python that a folder is a package is place a (normally empty) file in the folder named `__init__.py`. If we forget this file, we won't be able to import modules from that folder.

Let's put our modules inside an `ecommerce` package in our working folder, which will also contain a `main.py` file to start the program. Let's additionally add another package in the `ecommerce` package for various payment options. The folder hierarchy will look like this:

```
parent_directory/
    main.py
    ecommerce/
        __init__.py
        database.py
        products.py
        payments/
            __init__.py
            square.py
            stripe.py
```

When importing modules or classes between packages, we have to be cautious about the syntax. In Python 3, there are two ways of importing modules: absolute imports and relative imports.

Absolute imports

Absolute imports specify the complete path to the module, function, or path we want to import. If we need access to the `Product` class inside the `products` module, we could use any of these syntaxes to do an absolute import:

```
import ecommerce.products
product = ecommerce.products.Product()
```

or

```
from ecommerce.products import Product
product = Product()
```

or

```
from ecommerce import products
product = products.Product()
```

The `import` statements use the period operator to separate packages or modules.

These statements will work from any module. We could instantiate a `Product` class using this syntax in `main.py`, in the `database` module, or in either of the two payment modules. Indeed, assuming the packages are available to Python, it will be able to import them. For example, the packages can also be installed to the Python site packages folder, or the `PYTHONPATH` environment variable could be customized to dynamically tell Python what folders to search for packages and modules it is going to import.

So, with these choices, which syntax do we choose? It depends on your personal taste and the application at hand. If there are dozens of classes and functions inside the `products` module that I want to use, I generally import the module name using the `from ecommerce import products` syntax, and then access the individual classes using `products.Product`. If I only need one or two classes from the `products` module, I can import them directly using the `from ecommerce.proucts import Product` syntax. I don't personally use the first syntax very often unless I have some kind of name conflict (for example, I need to access two completely different modules called `products` and I need to separate them). Do whatever you think makes your code look more elegant.

Relative imports

When working with related modules in a package, it seems kind of silly to specify the full path; we know what our parent module is named. This is where **relative imports** come in. Relative imports are basically a way of saying find a class, function, or module as it is positioned relative to the current module. For example, if we are working in the `products` module and we want to import the `Database` class from the `database` module next to it, we could use a relative import:

```
from .database import Database
```

The period in front of `database` says "*use the database module inside the current package*". In this case, the current package is the package containing the `products.py` file we are currently editing, that is, the `ecommerce` package.

If we were editing the `paypal` module inside the `ecommerce.payments` package, we would want to say "*use the database package inside the parent package*" instead. This is easily done with two periods, as shown here:

```
from ..database import Database
```

We can use more periods to go further up the hierarchy. Of course, we can also go down one side and back up the other. We don't have a deep enough example hierarchy to illustrate this properly, but the following would be a valid import if we had an `ecommerce.contact` package containing an `email` module and wanted to import the `send_mail` function into our `paypal` module:

```
from ..contact.email import send_mail
```

This import uses two periods to say, *the parent of the payments package*, and then uses the normal `package.module` syntax to go back *up* into the contact package.

Finally, we can import code directly from packages, as opposed to just modules inside packages. In this example, we have an `ecommerce` package containing two modules named `database.py` and `products.py`. The database module contains a `db` variable that is accessed from a lot of places. Wouldn't it be convenient if this could be imported as `import ecommerce.db` instead of `import ecommerce.database.db`?

Remember the `__init__.py` file that defines a directory as a package? This file can contain any variable or class declarations we like, and they will be available as part of the package. In our example, if the `ecommerce/__init__.py` file contained this line:

```
from .database import db
```

We can then access the `db` attribute from `main.py` or any other file using this import:

```
from ecommerce import db
```

It might help to think of the `__init__.py` file as if it was an `ecommerce.py` file if that file were a module instead of a package. This can also be useful if you put all your code in a single module and later decide to break it up into a package of modules. The `__init__.py` file for the new package can still be the main point of contact for other modules talking to it, but the code can be internally organized into several different modules or subpackages.

I recommend not putting all your code in an `__init__.py` file, though. Programmers do not expect actual logic to happen in this file, and much like with `from x import *`, it can trip them up if they are looking for the declaration of a particular piece of code and can't find it until they check `__init__.py`.

Organizing module contents

Inside any one module, we can specify variables, classes, or functions. They can be a handy way to store the global state without namespace conflicts. For example, we have been importing the `Database` class into various modules and then instantiating it, but it might make more sense to have only one `database` object globally available from the `database` module. The `database` module might look like this:

```
class Database:
    # the database implementation
    pass

database = Database()
```

Then we can use any of the import methods we've discussed to access the `database` object, for example:

```
from ecommerce.database import database
```

A problem with the preceding class is that the `database` object is created immediately when the module is first imported, which is usually when the program starts up. This isn't always ideal since connecting to a database can take a while, slowing down startup, or the database connection information may not yet be available. We could delay creating the database until it is actually needed by calling an `initialize_database` function to create the module-level variable:

```
class Database:
    # the database implementation
    pass

database = None

def initialize_database():
    global database
    database = Database()
```

The `global` keyword tells Python that the database variable inside `initialize_database` is the module level one we just defined. If we had not specified the variable as global, Python would have created a new local variable that would be discarded when the method exits, leaving the module-level value unchanged.

As these two examples illustrate, all module-level code is executed immediately at the time it is imported. However, if it is inside a method or function, the function will be created, but its internal code will not be executed until the function is called. This can be a tricky thing for scripts (such as the main script in our e-commerce example) that perform execution. Often, we will write a program that does something useful, and then later find that we want to import a function or class from that module in a different program. However, as soon as we import it, any code at the module level is immediately executed. If we are not careful, we can end up running the first program when we really only meant to access a couple functions inside that module.

To solve this, we should always put our startup code in a function (conventionally, called `main`) and only execute that function when we know we are running the module as a script, but not when our code is being imported from a different script. But how do we know this?

```
class UsefulClass:
    '''This class might be useful to other modules.'''
    pass

def main():
    '''creates a useful class and does something with it for our
module.'''
    useful = UsefulClass()
    print(useful)

if __name__ == "__main__":
    main()
```

Every module has a `__name__` special variable (remember, Python uses double underscores for special variables, such as a class's `__init__` method) that specifies the name of the module when it was imported. When the module is executed directly with `python module.py`, it is never imported, so the `__name__` is arbitrarily set to the string `"__main__"`. Make it a policy to wrap all your scripts in an `if __name__ == "__main__":` test, just in case you write a function you will find useful to be imported by other code someday.

So, methods go in classes, which go in modules, which go in packages. Is that all there is to it?

Actually, no. This is the typical order of things in a Python program, but it's not the only possible layout. Classes can be defined anywhere. They are typically defined at the module level, but they can also be defined inside a function or method, like this:

```
def format_string(string, formatter=None):
    '''Format a string using the formatter object, which
    is expected to have a format() method that accepts
    a string.'''
    class DefaultFormatter:
        '''Format a string in title case.'''
        def format(self, string):
            return str(string).title()

    if not formatter:
        formatter = DefaultFormatter()

    return formatter.format(string)

hello_string = "hello world, how are you today?"
print(" input: " + hello_string)
print("output: " + format_string(hello_string))
```

The output will be as follows:

```
input: hello world, how are you today?
output: Hello World, How Are You Today?
```

The `format_string` function accepts a string and optional formatter object, and then applies the formatter to that string. If no formatter is supplied, it creates a formatter of its own as a local class and instantiates it. Since it is created inside the scope of the function, this class cannot be accessed from anywhere outside of that function. Similarly, functions can be defined inside other functions as well; in general, any Python statement can be executed at any time.

These inner classes and functions are occasionally useful for one-off items that don't require or deserve their own scope at the module level, or only make sense inside a single method. However, it is not common to see Python code that frequently uses this technique.

Who can access my data?

Most object-oriented programming languages have a concept of access control. This is related to abstraction. Some attributes and methods on an object are marked private, meaning only that object can access them. Others are marked protected, meaning only that class and any subclasses have access. The rest are public, meaning any other object is allowed to access them.

Python doesn't do this. Python doesn't really believe in enforcing laws that might someday get in your way. Instead, it provides unenforced guidelines and best practices. Technically, all methods and attributes on a class are publicly available. If we want to suggest that a method should not be used publicly, we can put a note in docstrings indicating that the method is meant for internal use only (preferably, with an explanation of how the public-facing API works!).

By convention, we should also prefix an attribute or method with an underscore character, _. Python programmers will interpret this as "*this is an internal variable, think three times before accessing it directly*". But there is nothing inside the interpreter to stop them from accessing it if they think it is in their best interest to do so. Because if they think so, why should we stop them? We may not have any idea what future uses our classes may be put to.

There's another thing you can do to strongly suggest that outside objects don't access a property or method: prefix it with a double underscore, __. This will perform **name mangling** on the attribute in question. This basically means that the method can still be called by outside objects if they really want to do it, but it requires extra work and is a strong indicator that you demand that your attribute remains private. For example:

```
class SecretString:
    '''A not-at-all secure way to store a secret string.'''

    def __init__(self, plain_string, pass_phrase):
        self.__plain_string = plain_string
        self.__pass_phrase = pass_phrase

    def decrypt(self, pass_phrase):
        '''Only show the string if the pass_phrase is correct.'''
        if pass_phrase == self.__pass_phrase:
            return self.__plain_string
        else:
            return ''
```

If we load this class and test it in the interactive interpreter, we can see that it hides the plain text string from the outside world:

```
>>> secret_string = SecretString("ACME: Top Secret", "antwerp")
>>> print(secret_string.decrypt("antwerp"))
ACME: Top Secret
>>> print(secret_string.__plain_string)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'SecretString' object has no attribute
'__plain_text'
```

It looks like it works; nobody can access our `plain_text` attribute without the passphrase, so it must be safe. Before we get too excited, though, let's see how easy it can be to hack our security:

```
>>> print(secret_string._SecretString__plain_string)
ACME: Top Secret
```

Oh no! Somebody has hacked our secret string. Good thing we checked! This is Python name mangling at work. When we use a double underscore, the property is prefixed with `_<classname>`. When methods in the class internally access the variable, they are automatically unmangled. When external classes wish to access it, they have to do the name mangling themselves. So, name mangling does not guarantee privacy, it only strongly recommends it. Most Python programmers will not touch a double underscore variable on another object unless they have an extremely compelling reason to do so.

However, most Python programmers will not touch a single underscore variable without a compelling reason either. Therefore, there are very few good reasons to use a name-mangled variable in Python, and doing so can cause grief. For example, a name-mangled variable may be useful to a subclass, and it would have to do the mangling itself. Let other objects access your hidden information if they want to, just let them know, using a single-underscore prefix or some clear docstrings, that you think this is not a good idea.

Third-party libraries

Python ships with a lovely standard library, which is a collection of packages and modules that are available on every machine that runs Python. However, you'll soon find that it doesn't contain everything you need. When this happens, you have two options:

- Write a supporting package yourself
- Use somebody else's code

We won't be covering the details about turning your packages into libraries, but if you have a problem you need to solve and you don't feel like coding it (the best programmers are extremely lazy and prefer to reuse existing, proven code, rather than write their own), you can probably find the library you want on the **Python Package Index** (**PyPI**) at `http://pypi.python.org/`. Once you've identified a package that you want to install, you can use a tool called `pip` to install it. However, `pip` does not come with Python, but Python 3.4 contains a useful tool called `ensurepip`, which will install it:

```
python -m ensurepip
```

This may fail for you on Linux, Mac OS, or other Unix systems, in which case, you'll need to become root to make it work. On most modern Unix systems, this can be done with `sudo python -m ensurepip`.

If you are using an older version of Python than Python 3.4, you'll need to download and install `pip` yourself, since `ensurepip` doesn't exist. You can do this by following the instructions at `http://pip.readthedocs.org/`.

Once `pip` is installed and you know the name of the package you want to install, you can install it using syntax such as:

```
pip install requests
```

However, if you do this, you'll either be installing the third-party library directly into your system Python directory, or more likely, get an error that you don't have permission to do so. You could force the installation as an administrator, but common consensus in the Python community is that you should only use system installers to install the third-party library to your system Python directory.

Instead, Python 3.4 supplies the `venv` tool. This utility basically gives you a mini Python installation called a *virtual environment* in your working directory. When you activate the mini Python, commands related to Python will work on that directory instead of the system directory. So when you run `pip` or `python`, it won't touch the system Python at all. Here's how to use it:

```
cd project_directory
python -m venv env
source env/bin/activate  # on Linux or MacOS
env/bin/activate.bat     # on Windows
```

Typically, you'll create a different virtual environment for each Python project you work on. You can store your virtual environments anywhere, but I keep mine in the same directory as the rest of my project files (but ignored in version control), so first we `cd` into that directory. Then we run the `venv` utility to create a virtual environment named `env`. Finally, we use one of the last two lines (depending on the operating system, as indicated in the comments) to activate the environment. We'll need to execute this line each time we want to use that particular virtualenv, and then use the command `deactivate` when we are done working on this project.

Virtual environments are a terrific way to keep your third-party dependencies separate. It is common to have different projects that depend on different versions of a particular library (for example, an older website might run on Django 1.5, while newer versions run on Django 1.8). Keeping each project in separate virtualenvs makes it easy to work in either version of Django. Further, it prevents conflicts between system-installed packages and `pip` installed packages if you try to install the same package using different tools.

Case study

To tie it all together, let's build a simple command-line notebook application. This is a fairly simple task, so we won't be experimenting with multiple packages. We will, however, see common usage of classes, functions, methods, and docstrings.

Let's start with a quick analysis: notes are short memos stored in a notebook. Each note should record the day it was written and can have tags added for easy querying. It should be possible to modify notes. We also need to be able to search for notes. All of these things should be done from the command line.

The obvious object is the `Note` object; less obvious one is a `Notebook` container object. Tags and dates also seem to be objects, but we can use dates from Python's standard library and a comma-separated string for tags. To avoid complexity, in the prototype, let's not define separate classes for these objects.

`Note` objects have attributes for `memo` itself, `tags`, and `creation_date`. Each note will also need a unique integer `id` so that users can select them in a menu interface. Notes could have a method to modify note content and another for tags, or we could just let the notebook access those attributes directly. To make searching easier, we should put a `match` method on the `Note` object. This method will accept a string and can tell us if a note matches the string without accessing the attributes directly. This way, if we want to modify the search parameters (to search tags instead of note contents, for example, or to make the search case-insensitive), we only have to do it in one place.

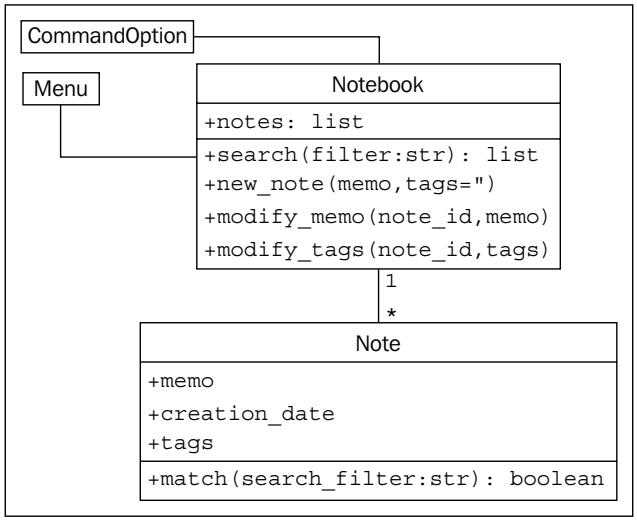
The `Notebook` object obviously has the list of notes as an attribute. It will also need a search method that returns a list of filtered notes.

But how do we interact with these objects? We've specified a command-line app, which can mean either that we run the program with different options to add or edit commands, or we have some kind of a menu that allows us to pick different things to do to the notebook. We should try to design it such that either interface is supported and future interfaces, such as a GUI toolkit or web-based interface, could be added in the future.

As a design decision, we'll implement the menu interface now, but will keep the command-line options version in mind to ensure we design our `Notebook` class with extensibility in mind.

If we have two command-line interfaces, each interacting with the `Notebook` object, then `Notebook` will need some methods for those interfaces to interact with. We need to be able to `add` a new note, and `modify` an existing note by `id`, in addition to the `search` method we've already discussed. The interfaces will also need to be able to list all notes, but they can do that by accessing the `notes` list attribute directly.

We may be missing a few details, but that gives us a really good overview of the code we need to write. We can summarize all this in a simple class diagram:



Before writing any code, let's define the folder structure for this project. The menu interface should clearly be in its own module, since it will be an executable script, and we may have other executable scripts accessing the notebook in the future. The `Notebook` and `Note` objects can live together in one module. These modules can both exist in the same top-level directory without having to put them in a package. An empty `command_option.py` module can help remind us in the future that we were planning to add new user interfaces.

```
parent_directory/
    notebook.py
    menu.py
    command_option.py
```

Now let's see some code. We start by defining the `Note` class as it seems simplest. The following example presents `Note` in its entirety. Docstrings within the example explain how it all fits together.

```
import datetime

# Store the next available id for all new notes
last_id = 0

class Note:
    '''Represent a note in the notebook. Match against a
```

```
        string in searches and store tags for each note.'''

        def __init__(self, memo, tags=''):
            '''initialize a note with memo and optional
            space-separated tags. Automatically set the note's
            creation date and a unique id.'''
            self.memo = memo
            self.tags = tags
            self.creation_date = datetime.date.today()
            global last_id
            last_id += 1
            self.id = last_id

        def match(self, filter):
            '''Determine if this note matches the filter
            text. Return True if it matches, False otherwise.

            Search is case sensitive and matches both text and
            tags.'''
            return filter in self.memo or filter in self.tags
```

Before continuing, we should quickly fire up the interactive interpreter and test our code so far. Test frequently and often because things never work the way you expect them to. Indeed, when I tested my first version of this example, I found out I had forgotten the `self` argument in the `match` function! For now, check a few things using the interpreter:

```
>>> from notebook import Note
>>> n1 = Note("hello first")
>>> n2 = Note("hello again")
>>> n1.id
1
>>> n2.id
2
>>> n1.match('hello')
True
>>> n2.match('second')
False
```

It looks like everything is behaving as expected. Let's create our notebook next:

```
class Notebook:
    '''Represent a collection of notes that can be tagged,
    modified, and searched.'''

    def __init__(self):
        '''Initialize a notebook with an empty list.'''
        self.notes = []

    def new_note(self, memo, tags=''):
        '''Create a new note and add it to the list.'''
        self.notes.append(Note(memo, tags))

    def modify_memo(self, note_id, memo):
        '''Find the note with the given id and change its
        memo to the given value.'''
        for note in self.notes:
            if note.id == note_id:
                note.memo = memo
                break

    def modify_tags(self, note_id, tags):
        '''Find the note with the given id and change its
        tags to the given value.'''
        for note in self.notes:
            if note.id == note_id:
                note.tags = tags
                break

    def search(self, filter):
        '''Find all notes that match the given filter
        string.'''
        return [note for note in self.notes if
                note.match(filter)]
```

We'll clean this up in a minute. First, let's test it to make sure it works:

```
>>> from notebook import Note, Notebook
>>> n = Notebook()
>>> n.new_note("hello world")
>>> n.new_note("hello again")
>>> n.notes
[<notebook.Note object at 0xb730a78c>, <notebook.Note object at
  0xb73103ac>]
```

```
>>> n.notes[0].id
1
>>> n.notes[1].id
2
>>> n.notes[0].memo
'hello world'
>>> n.search("hello")
[<notebook.Note object at 0xb730a78c>, <notebook.Note object at
  0xb73103ac>]
>>> n.search("world")
[<notebook.Note object at 0xb730a78c>]
>>> n.modify_memo(1, "hi world")
>>> n.notes[0].memo
'hi world'
```

It does work. The code is a little messy though; our `modify_tags` and `modify_memo` methods are almost identical. That's not good coding practice. Let's see how we can improve it.

Both methods are trying to identify the note with a given ID before doing something to that note. So, let's add a method to locate the note with a specific ID. We'll prefix the method name with an underscore to suggest that the method is for internal use only, but of course, our menu interface can access the method if it wants to:

```
    def _find_note(self, note_id):
        '''Locate the note with the given id.'''
        for note in self.notes:
            if note.id == note_id:
                return note
        return None

    def modify_memo(self, note_id, memo):
        '''Find the note with the given id and change its
        memo to the given value.'''
        self._find_note(note_id).memo = memo
```

This should work for now. Let's have a look at the menu interface. The interface simply needs to present a menu and allow the user to input choices. Here's our first try:

```
import sys
```

```
from notebook import Notebook, Note

class Menu:
    '''Display a menu and respond to choices when run.'''
    def __init__(self):
        self.notebook = Notebook()
        self.choices = {
                "1": self.show_notes,
                "2": self.search_notes,
                "3": self.add_note,
                "4": self.modify_note,
                "5": self.quit
                }

    def display_menu(self):
        print("""
Notebook Menu

1. Show all Notes
2. Search Notes
3. Add Note
4. Modify Note
5. Quit
""")

    def run(self):
        '''Display the menu and respond to choices.'''
        while True:
            self.display_menu()
            choice = input("Enter an option: ")
            action = self.choices.get(choice)
            if action:
                action()
            else:
                print("{0} is not a valid choice".format(choice))

    def show_notes(self, notes=None):
        if not notes:
            notes = self.notebook.notes
        for note in notes:
            print("{0}: {1}\n{2}".format(
                note.id, note.tags, note.memo))

    def search_notes(self):
```

```
            filter = input("Search for: ")
            notes = self.notebook.search(filter)
            self.show_notes(notes)

        def add_note(self):
            memo = input("Enter a memo: ")
            self.notebook.new_note(memo)
            print("Your note has been added.")

        def modify_note(self):
            id = input("Enter a note id: ")
            memo = input("Enter a memo: ")
            tags = input("Enter tags: ")
            if memo:
                self.notebook.modify_memo(id, memo)
            if tags:
                self.notebook.modify_tags(id, tags)

        def quit(self):
            print("Thank you for using your notebook today.")
            sys.exit(0)

    if __name__ == "__main__":
        Menu().run()
```

This code first imports the notebook objects using an absolute import. Relative imports wouldn't work because we haven't placed our code inside a package. The `Menu` class's `run` method repeatedly displays a menu and responds to choices by calling functions on the notebook. This is done using an idiom that is rather peculiar to Python. The choices entered by the user are strings. In the menu's `__init__` method, we create a dictionary that maps strings to functions on the menu object itself. Then, when the user makes a choice, we retrieve the object from the dictionary. The `action` variable actually refers to a specific method, and is called by appending empty brackets (since none of the methods require parameters) to the variable. Of course, the user might have entered an inappropriate choice, so we check if the action really exists before calling it.

Each of the various methods request user input and call appropriate methods on the `Notebook` object associated with it. For the `search` implementation, we notice that after we've filtered the notes, we need to show them to the user, so we make the `show_notes` function serve double duty; it accepts an optional `notes` parameter. If it's supplied, it displays only the filtered notes, but if it's not, it displays all notes. Since the `notes` parameter is optional, `show_notes` can still be called with no parameters as an empty menu item.

If we test this code, we'll find that modifying notes doesn't work. There are two bugs, namely:

- The notebook crashes when we enter a note ID that does not exist. We should never trust our users to enter correct data!
- Even if we enter a correct ID, it will crash because the note IDs are integers, but our menu is passing a string.

The latter bug can be solved by modifying the `Notebook` class's `_find_note` method to compare the values using strings instead of the integers stored in the note, as follows:

```
def _find_note(self, note_id):
    '''Locate the note with the given id.'''
    for note in self.notes:
        if str(note.id) == str(note_id):
            return note
    return None
```

We simply convert both the input (`note_id`) and the note's ID to strings before comparing them. We could also convert the input to an integer, but then we'd have trouble if the user had entered the letter "a" instead of the number "1".

The problem with users entering note IDs that don't exist can be fixed by changing the two `modify` methods on the notebook to check whether `_find_note` returned a note or not, like this:

```
def modify_memo(self, note_id, memo):
    '''Find the note with the given id and change its
    memo to the given value.'''
    note = self._find_note(note_id)
    if note:
        note.memo = memo
        return True
    return False
```

This method has been updated to return `True` or `False`, depending on whether a note has been found. The menu could use this return value to display an error if the user entered an invalid note. This code is a bit unwieldy though; it would look a bit better if it raised an exception instead. We'll cover those in *Chapter 4, Expecting the Unexpected*.

Your Coding Challenge

Try writing some object-oriented code. The goal is to use the principles and syntax you learned in this chapter to ensure you can use it, instead of just reading about it. If you've been working on a Python project, go back over it and see if there are some objects you can create and add properties or methods to. If it's large, try dividing it into a few modules or even packages and play with the syntax.

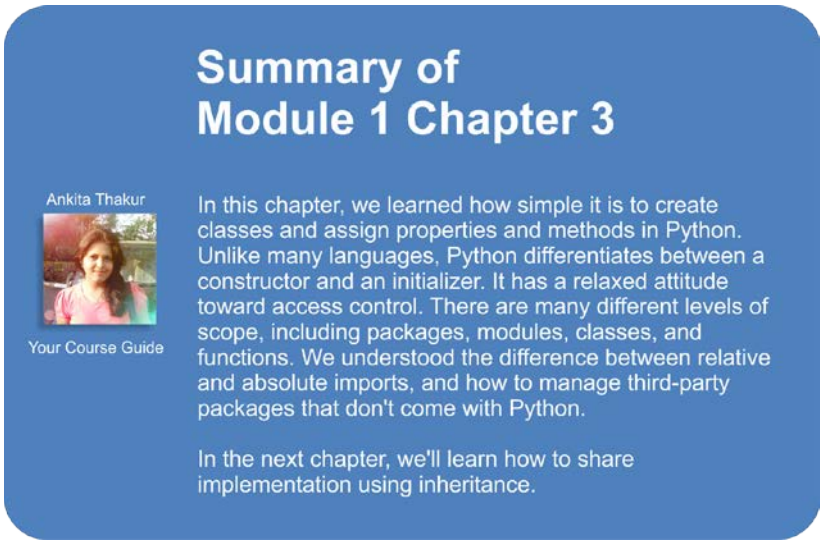
If you don't have such a project, try starting a new one. It doesn't have to be something you intend to finish, just stub out some basic design parts. You don't need to fully implement everything, often just a print("this method will do something") is all you need to get the overall design in place. This is called top-down design, in which you work out the different interactions and describe how they should work before actually implementing what they do. The converse, bottom-up design, implements details first and then ties them all together. Both patterns are useful at different times, but for understanding object-oriented principles, a top-down workflow is more suitable.

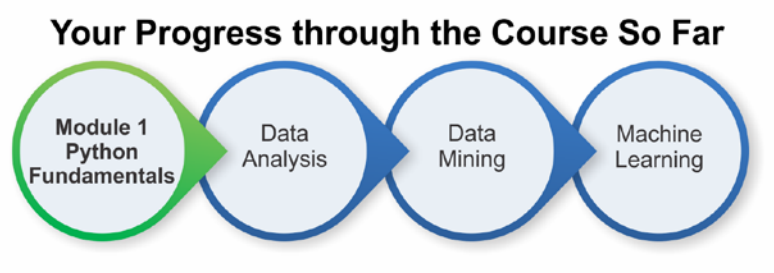
Ankita Thakur

Your Course Guide

If you're having trouble coming up with ideas, try writing a to-do application. (**Hint:** It would be similar to the design of the notebook application, but with extra date management methods.) It can keep track of things you want to do each day, and allow you to mark them as completed.

Now, try designing a bigger project. It doesn't have to actually do anything, but make sure you experiment with the package and module importing syntax. Add some functions in various modules and try importing them from other modules and packages. Use relative and absolute imports. See the difference, and try to imagine scenarios where you would want to use each one.


Summary of
Module 1 Chapter 3
Ankita Thakur
Your Course Guide
In this chapter, we learned how simple it is to create classes and assign properties and methods in Python. Unlike many languages, Python differentiates between a constructor and an initializer. It has a relaxed attitude toward access control. There are many different levels of scope, including packages, modules, classes, and functions. We understood the difference between relative and absolute imports, and how to manage third-party packages that don't come with Python.
In the next chapter, we'll learn how to share implementation using inheritance.


Your Progress through the Course So Far
Module 1 Python Fundamentals
Data Analysis
Data Mining
Machine Learning

4
When Objects Are Alike

In the programming world, duplicate code is considered evil. We should not have multiple copies of the same, or similar, code in different places.

There are many ways to merge pieces of code or objects that have a similar functionality. In this chapter, we'll be covering the most famous object-oriented principle: inheritance. As discussed in *Chapter 1*, *Object-oriented Design*, inheritance allows us to create *is a* relationships between two or more classes, abstracting common logic into superclasses and managing specific details in the subclass. In particular, we'll be covering the Python syntax and principles for:

- Basic inheritance
- Inheriting from built-ins
- Multiple inheritance
- Polymorphism and duck typing

Basic inheritance

Technically, every class we create uses inheritance. All Python classes are subclasses of the special class named `object`. This class provides very little in terms of data and behaviors (the behaviors it does provide are all double-underscore methods intended for internal use only), but it does allow Python to treat all objects in the same way.

If we don't explicitly inherit from a different class, our classes will automatically inherit from `object`. However, we can openly state that our class derives from `object` using the following syntax:

```
class MySubClass(object):
    pass
```

This is inheritance! This example is, technically, no different from our very first example in *Chapter 2*, *Objects in Python*, since Python 3 automatically inherits from `object` if we don't explicitly provide a different superclass. A superclass, or parent class, is a class that is being inherited from. A subclass is a class that is inheriting from a superclass. In this case, the superclass is `object`, and `MySubClass` is the subclass. A subclass is also said to be derived from its parent class or that the subclass extends the parent.

As you've probably figured out from the example, inheritance requires a minimal amount of extra syntax over a basic class definition. Simply include the name of the parent class inside parentheses after the class name but before the colon terminating the class definition. This is all we have to do to tell Python that the new class should be derived from the given superclass.

How do we apply inheritance in practice? The simplest and most obvious use of inheritance is to add functionality to an existing class. Let's start with a simple contact manager that tracks the name and e-mail address of several people. The contact class is responsible for maintaining a list of all contacts in a class variable, and for initializing the name and address for an individual contact:

```
class Contact:
    all_contacts = []

    def __init__(self, name, email):
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)
```

This example introduces us to class variables. The `all_contacts` list, because it is part of the class definition, is shared by all instances of this class. This means that there is only one `Contact.all_contacts` list, which we can access as `Contact.all_contacts`. Less obviously, we can also access it as `self.all_contacts` on any object instantiated from `Contact`. If the field can't be found on the object, then it will be found on the class and thus refer to the same single list.

Be careful with this syntax, for if you ever *set* the variable using `self.all_contacts`, you will actually be creating a **new** instance variable associated only with that object. The class variable will still be unchanged and accessible as `Contact.all_contacts`.

This is a simple class that allows us to track a couple pieces of data about each contact. But what if some of our contacts are also suppliers that we need to order supplies from? We could add an `order` method to the `Contact` class, but that would allow people to accidentally order things from contacts who are customers or family friends. Instead, let's create a new `Supplier` class that acts like our `Contact` class, but has an additional `order` method:

```
class Supplier(Contact):
    def order(self, order):
        print("If this were a real system we would send "
                "'{}' order to '{}'".format(order, self.name))
```

Now, if we test this class in our trusty interpreter, we see that all contacts, including suppliers, accept a name and e-mail address in their `__init__`, but only suppliers have a functional order method:

```
>>> c = Contact("Some Body", "somebody@example.net")
>>> s = Supplier("Sup Plier", "supplier@example.net")
>>> print(c.name, c.email, s.name, s.email)
Some Body somebody@example.net Sup Plier supplier@example.net
>>> c.all_contacts
[<__main__.Contact object at 0xb7375ecc>,
 <__main__.Supplier object at 0xb7375f8c>]
>>> c.order("I need pliers")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Contact' object has no attribute 'order'
>>> s.order("I need pliers")
If this were a real system we would send 'I need pliers' order to
'Sup Plier '
```

So, now our `Supplier` class can do everything a contact can do (including adding itself to the list of `all_contacts`) and all the special things it needs to handle as a supplier. This is the beauty of inheritance.

Extending built-ins

One interesting use of this kind of inheritance is adding functionality to built-in classes. In the `Contact` class seen earlier, we are adding contacts to a list of all contacts. What if we also wanted to search that list by name? Well, we could add a method on the `Contact` class to search it, but it feels like this method actually belongs to the list itself. We can do this using inheritance:

```
class ContactList(list):
    def search(self, name):
        '''Return all contacts that contain the search value
        in their name.'''
        matching_contacts = []
        for contact in self:
            if name in contact.name:
                matching_contacts.append(contact)
        return matching_contacts

class Contact:
    all_contacts = ContactList()

    def __init__(self, name, email):
        self.name = name
        self.email = email
        self.all_contacts.append(self)
```

Instead of instantiating a normal list as our class variable, we create a new `ContactList` class that extends the built-in `list`. Then, we instantiate this subclass as our `all_contacts` list. We can test the new search functionality as follows:

```
>>> c1 = Contact("John A", "johna@example.net")
>>> c2 = Contact("John B", "johnb@example.net")
>>> c3 = Contact("Jenna C", "jennac@example.net")
>>> [c.name for c in Contact.all_contacts.search('John')]
['John A', 'John B']
```

Are you wondering how we changed the built-in syntax `[]` into something we can inherit from? Creating an empty list with `[]` is actually a shorthand for creating an empty list using `list()`; the two syntaxes behave identically:

```
>>> [] == list()
True
```

In reality, the `[]` syntax is actually so-called **syntax sugar** that calls the `list()` constructor under the hood. The `list` data type is a class that we can extend. In fact, the list itself extends the `object` class:

```
>>> isinstance([], object)
True
```

As a second example, we can extend the `dict` class, which is, similar to the list, the class that is constructed when using the `{}` syntax shorthand:

```
class LongNameDict(dict):
    def longest_key(self):
        longest = None
        for key in self:
            if not longest or len(key) > len(longest):
                longest = key
        return longest
```

This is easy to test in the interactive interpreter:

```
>>> longkeys = LongNameDict()
>>> longkeys['hello'] = 1
>>> longkeys['longest yet'] = 5
>>> longkeys['hello2'] = 'world'
>>> longkeys.longest_key()
'longest yet'
```

Most built-in types can be similarly extended. Commonly extended built-ins are `object`, `list`, `set`, `dict`, `file`, and `str`. Numerical types such as `int` and `float` are also occasionally inherited from.

Overriding and super

So, inheritance is great for *adding* new behavior to existing classes, but what about *changing* behavior? Our `contact` class allows only a name and an e-mail address. This may be sufficient for most contacts, but what if we want to add a phone number for our close friends?

As we saw in *Chapter 2*, *Objects in Python*, we can do this easily by just setting a `phone` attribute on the contact after it is constructed. But if we want to make this third variable available on initialization, we have to override `__init__`. Overriding means altering or replacing a method of the superclass with a new method (with the same name) in the subclass. No special syntax is needed to do this; the subclass's newly created method is automatically called instead of the superclass's method. For example:

```
class Friend(Contact):
    def __init__(self, name, email, phone):
        self.name = name
        self.email = email
        self.phone = phone
```

Any method can be overridden, not just `__init__`. Before we go on, however, we need to address some problems in this example. Our `Contact` and `Friend` classes have duplicate code to set up the `name` and `email` properties; this can make code maintenance complicated as we have to update the code in two or more places. More alarmingly, our `Friend` class is neglecting to add itself to the `all_contacts` list we have created on the `Contact` class.

What we really need is a way to execute the original `__init__` method on the `Contact` class. This is what the `super` function does; it returns the object as an instance of the parent class, allowing us to call the parent method directly:

```
class Friend(Contact):
    def __init__(self, name, email, phone):
        super().__init__(name, email)
        self.phone = phone
```

This example first gets the instance of the parent object using `super`, and calls `__init__` on that object, passing in the expected arguments. It then does its own initialization, namely, setting the `phone` attribute.

Note that the `super()` syntax does not work in older versions of Python. Like the [] and {} syntaxes for lists and dictionaries, it is a shorthand for a more complicated construct. We'll learn more about this shortly when we discuss multiple inheritance, but know for now that in Python 2, you would have to call `super(EmailContact, self).__init__()`. Specifically notice that the first argument is the name of the child class, not the name as the parent class you want to call, as some might expect. Also, remember the class comes before the object. I always forget the order, so the new syntax in Python 3 has saved me hours of having to look it up.

A `super()` call can be made inside any method, not just `__init__`. This means all methods can be modified via overriding and calls to `super`. The call to `super` can also be made at any point in the method; we don't have to make the call as the first line in the method. For example, we may need to manipulate or validate incoming parameters before forwarding them to the superclass.

Multiple inheritance

Multiple inheritance is a touchy subject. In principle, it's very simple: a subclass that inherits from more than one parent class is able to access functionality from both of them. In practice, this is less useful than it sounds and many expert programmers recommend against using it.

As a rule of thumb, if you think you need multiple inheritance, you're probably wrong, but if you know you need it, you're probably right.

The simplest and most useful form of multiple inheritance is called a **mixin**. A mixin is generally a superclass that is not meant to exist on its own, but is meant to be inherited by some other class to provide extra functionality. For example, let's say we wanted to add functionality to our `Contact` class that allows sending an e-mail to `self.email`. Sending e-mail is a common task that we might want to use on many other classes. So, we can write a simple mixin class to do the e-mailing for us:

```
class MailSender:
    def send_mail(self, message):
        print("Sending mail to " + self.email)
        # Add e-mail logic here
```

For brevity, we won't include the actual e-mail logic here; if you're interested in studying how it's done, see the `smtplib` module in the Python standard library.

This class doesn't do anything special (in fact, it can barely function as a standalone class), but it does allow us to define a new class that describes both a `Contact` and a `MailSender`, using multiple inheritance:

```
class EmailableContact(Contact, MailSender):
    pass
```

The syntax for multiple inheritance looks like a parameter list in the class definition. Instead of including one base class inside the parentheses, we include two (or more), separated by a comma. We can test this new hybrid to see the mixin at work:

```
>>> e = EmailableContact("John Smith", "jsmith@example.net")
```

```
>>> Contact.all_contacts
[<__main__.EmailableContact object at 0xb7205fac>]
>>> e.send_mail("Hello, test e-mail here")
Sending mail to jsmith@example.net
```

The `Contact` initializer is still adding the new contact to the `all_contacts` list, and the mixin is able to send mail to `self.email` so we know everything is working.

This wasn't so hard, and you're probably wondering what the dire warnings about multiple inheritance are. We'll get into the complexities in a minute, but let's consider some other options we had, rather than using a mixin here:

- We could have used single inheritance and added the `send_mail` function to the subclass. The disadvantage here is that the e-mail functionality then has to be duplicated for any other classes that need e-mail.
- We can create a standalone Python function for sending an e-mail, and just call that function with the correct e-mail address supplied as a parameter when the e-mail needs to be sent.
- We could have explored a few ways of using composition instead of inheritance. For example, `EmailableContact` could have a `MailSender` object instead of inheriting from it.
- We could monkey-patch (we'll briefly cover monkey-patching in *Chapter 7*, *Python Object-oriented Shortcuts*) the `Contact` class to have a `send_mail` method after the class has been created. This is done by defining a function that accepts the `self` argument, and setting it as an attribute on an existing class.

Multiple inheritance works all right when mixing methods from different classes, but it gets very messy when we have to call methods on the superclass. There are multiple superclasses. How do we know which one to call? How do we know what order to call them in?

Let's explore these questions by adding a home address to our `Friend` class. There are a few approaches we might take. An address is a collection of strings representing the street, city, country, and other related details of the contact. We could pass each of these strings as a parameter into the `Friend` class's `__init__` method. We could also store these strings in a tuple or dictionary and pass them into `__init__` as a single argument. This is probably the best course of action if there are no methods that need to be added to the address.

Another option would be to create a new `Address` class to hold those strings together, and then pass an instance of this class into the `__init__` method of our `Friend` class. The advantage of this solution is that we can add behavior (say, a method to give directions or to print a map) to the data instead of just storing it statically. This is an example of composition, as we discussed in *Chapter 1, Object-oriented Design*. The "has a" relationship of composition is a perfectly viable solution to this problem and allows us to reuse `Address` classes in other entities such as buildings, businesses, or organizations.

However, inheritance is also a viable solution, and that's what we want to explore. Let's add a new class that holds an address. We'll call this new class "AddressHolder" instead of "Address" because inheritance defines an *is a* relationship. It is not correct to say a "Friend" is an "Address" , but since a friend can have an "Address" , we can argue that a "Friend" is an "AddressHolder". Later, we could create other entities (companies, buildings) that also hold addresses. Here's our `AddressHolder` class:

```
class AddressHolder:
    def __init__(self, street, city, state, code):
        self.street = street
        self.city = city
        self.state = state
        self.code = code
```

Very simple; we just take all the data and toss it into instance variables upon initialization.

The diamond problem

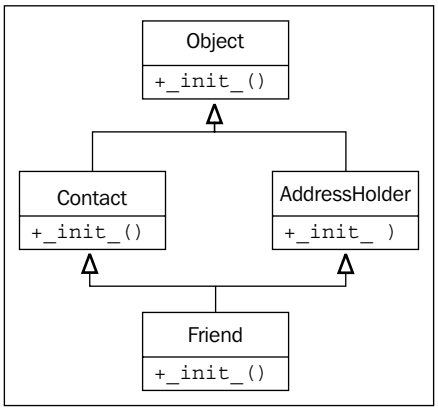
We can use multiple inheritance to add this new class as a parent of our existing `Friend` class. The tricky part is that we now have two parent `__init__` methods both of which need to be initialized. And they need to be initialized with different arguments. How do we do this? Well, we could start with a naive approach:

```
class Friend(Contact, AddressHolder):
    def __init__(
        self, name, email, phone,street, city, state, code):
        Contact.__init__(self, name, email)
        AddressHolder.__init__(self, street, city, state, code)
        self.phone = phone
```

In this example, we directly call the `__init__` function on each of the superclasses and explicitly pass the `self` argument. This example technically works; we can access the different variables directly on the class. But there are a few problems.

First, it is possible for a superclass to go uninitialized if we neglect to explicitly call the initializer. That wouldn't break this example, but it could cause hard-to-debug program crashes in common scenarios. Imagine trying to insert data into a database that has not been connected to, for example.

Second, and more sinister, is the possibility of a superclass being called multiple times because of the organization of the class hierarchy. Look at this inheritance diagram:

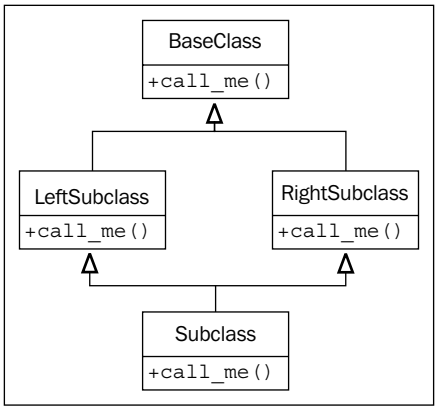


The `__init__` method from the `Friend` class first calls `__init__` on `Contact`, which implicitly initializes the `object` superclass (remember, all classes derive from `object`). `Friend` then calls `__init__` on `AddressHolder`, which implicitly initializes the `object` superclass *again*. This means the parent class has been set up twice. With the `object` class, that's relatively harmless, but in some situations, it could spell disaster. Imagine trying to connect to a database twice for every request!

The base class should only be called once. Once, yes, but when? Do we call `Friend`, then `Contact`, then `Object`, then `AddressHolder`? Or `Friend`, then `Contact`, then `AddressHolder`, then `Object`?

The order in which methods can be called can be adapted on the fly by modifying the `__mro__` (**Method Resolution Order**) attribute on the class. This is beyond the scope of this module. If you think you need to understand it, I recommend *Expert Python Programming*, *Tarek Ziadé*, *Packt Publishing*, or read the original documentation on the topic at `http://www.python.org/download/releases/2.3/mro/`.

Let's look at a second contrived example that illustrates this problem more clearly. Here we have a base class that has a method named `call_me`. Two subclasses override that method, and then another subclass extends both of these using multiple inheritance. This is called diamond inheritance because of the diamond shape of the class diagram:



Let's convert this diagram to code; this example shows when the methods are called:

```
class BaseClass:
    num_base_calls = 0
    def call_me(self):
        print("Calling method on Base Class")
        self.num_base_calls += 1

class LeftSubclass(BaseClass):
    num_left_calls = 0
    def call_me(self):
        BaseClass.call_me(self)
        print("Calling method on Left Subclass")
        self.num_left_calls += 1

class RightSubclass(BaseClass):
    num_right_calls = 0
    def call_me(self):
        BaseClass.call_me(self)
        print("Calling method on Right Subclass")
        self.num_right_calls += 1

class Subclass(LeftSubclass, RightSubclass):
```

```
        num_sub_calls = 0
        def call_me(self):
            LeftSubclass.call_me(self)
            RightSubclass.call_me(self)
            print("Calling method on Subclass")
            self.num_sub_calls += 1
```

This example simply ensures that each overridden `call_me` method directly calls the parent method with the same name. It lets us know each time a method is called by printing the information to the screen. It also updates a static variable on the class to show how many times it has been called. If we instantiate one `Subclass` object and call the method on it once, we get this output:

```
>>> s = Subclass()
>>> s.call_me()
Calling method on Base Class
Calling method on Left Subclass
Calling method on Base Class
Calling method on Right Subclass
Calling method on Subclass
>>> print(
... s.num_sub_calls,
... s.num_left_calls,
... s.num_right_calls,
... s.num_base_calls)
1 1 1 2
```

Thus we can clearly see the base class's `call_me` method being called twice. This could lead to some insidious bugs if that method is doing actual work—like depositing into a bank account—twice.

The thing to keep in mind with multiple inheritance is that we only want to call the "next" method in the class hierarchy, not the "parent" method. In fact, that next method may not be on a parent or ancestor of the current class. The `super` keyword comes to our rescue once again. Indeed, `super` was originally developed to make complicated forms of multiple inheritance possible. Here is the same code written using `super`:

```
    class BaseClass:
        num_base_calls = 0
        def call_me(self):
```

```
            print("Calling method on Base Class")
            self.num_base_calls += 1

    class LeftSubclass(BaseClass):
        num_left_calls = 0
        def call_me(self):
            super().call_me()
            print("Calling method on Left Subclass")
            self.num_left_calls += 1

    class RightSubclass(BaseClass):
        num_right_calls = 0
        def call_me(self):
            super().call_me()
            print("Calling method on Right Subclass")
            self.num_right_calls += 1

    class Subclass(LeftSubclass, RightSubclass):
        num_sub_calls = 0
        def call_me(self):
            super().call_me()
            print("Calling method on Subclass")
            self.num_sub_calls += 1
```

The change is pretty minor; we simply replaced the naive direct calls with calls to `super()`, although the bottom subclass only calls `super` once rather than having to make the calls for both the left and right. The change is simple enough, but look at the difference when we execute it:

```
>>> s = Subclass()
>>> s.call_me()
Calling method on Base Class
Calling method on Right Subclass
Calling method on Left Subclass
Calling method on Subclass
>>> print(s.num_sub_calls, s.num_left_calls, s.num_right_calls,
s.num_base_calls)
1 1 1 1
```

Looks good, our base method is only being called once. But what is `super()` actually doing here? Since the `print` statements are executed after the `super` calls, the printed output is in the order each method is actually executed. Let's look at the output from back to front to see who is calling what.

First, `call_me` of `Subclass` calls `super().call_me()`, which happens to refer to `LeftSubclass.call_me()`. The `LeftSubclass.call_me()` method then calls `super().call_me()`, but in this case, `super()` is referring to `RightSubclass.call_me()`.

Pay particular attention to this: the `super` call is *not* calling the method on the superclass of `LeftSubclass` (which is `BaseClass`). Rather, it is calling `RightSubclass`, even though it is not a direct parent of `LeftSubclass`! This is the *next* method, not the parent method. `RightSubclass` then calls `BaseClass` and the `super` calls have ensured each method in the class hierarchy is executed once.

Different sets of arguments

This is going to make things complicated as we return to our `Friend` multiple inheritance example. In the `__init__` method for `Friend`, we were originally calling `__init__` for both parent classes, *with different sets of arguments*:

```
Contact.__init__(self, name, email)
AddressHolder.__init__(self, street, city, state, code)
```

How can we manage different sets of arguments when using `super`? We don't necessarily know which class `super` is going to try to initialize first. Even if we did, we need a way to pass the "extra" arguments so that subsequent calls to `super`, on other subclasses, receive the right arguments.

Specifically, if the first call to `super` passes the `name` and `email` arguments to `Contact.__init__`, and `Contact.__init__` then calls `super`, it needs to be able to pass the address-related arguments to the "next" method, which is `AddressHolder.__init__`.

This is a problem whenever we want to call superclass methods with the same name, but with different sets of arguments. Most often, the only time you would want to call a superclass with a completely different set of arguments is in `__init__`, as we're doing here. Even with regular methods, though, we may want to add optional parameters that only make sense to one subclass or set of subclasses.

Sadly, the only way to solve this problem is to plan for it from the beginning. We have to design our base class parameter lists to accept keyword arguments for any parameters that are not required by every subclass implementation. Finally, we must ensure the method freely accepts unexpected arguments and passes them on to its `super` call, in case they are necessary to later methods in the inheritance order.

Python's function parameter syntax provides all the tools we need to do this, but it makes the overall code look cumbersome. Have a look at the proper version of the `Friend` multiple inheritance code:

```
class Contact:
    all_contacts = []

    def __init__(self, name='', email='', **kwargs):
        super().__init__(**kwargs)
        self.name = name
        self.email = email
        self.all_contacts.append(self)

class AddressHolder:
    def __init__(self, street='', city='', state='', code='',
            **kwargs):
        super().__init__(**kwargs)
        self.street = street
        self.city = city
        self.state = state
        self.code = code

class Friend(Contact, AddressHolder):
    def __init__(self, phone='', **kwargs):
        super().__init__(**kwargs)
        self.phone = phone
```

We've changed all arguments to keyword arguments by giving them an empty string as a default value. We've also ensured that a `**kwargs` parameter is included to capture any additional parameters that our particular method doesn't know what to do with. It passes these parameters up to the next class with the `super` call.

If you aren't familiar with the `**kwargs` syntax, it basically collects any keyword arguments passed into the method that were not explicitly listed in the parameter list. These arguments are stored in a dictionary named `kwargs` (we can call the variable whatever we like, but convention suggests `kw`, or `kwargs`). When we call a different method (for example, `super().__init__`) with a `**kwargs` syntax, it unpacks the dictionary and passes the results to the method as normal keyword arguments. We'll cover this in detail in *Chapter 7, Python Object-oriented Shortcuts*.

The previous example does what it is supposed to do. But it's starting to look messy, and it has become difficult to answer the question, *What arguments do we need to pass into* `Friend.__init__`? This is the foremost question for anyone planning to use the class, so a docstring should be added to the method to explain what is happening.

Further, even this implementation is insufficient if we want to *reuse* variables in parent classes. When we pass the `**kwargs` variable to `super`, the dictionary does not include any of the variables that were included as explicit keyword arguments. For example, in `Friend.__init__`, the call to `super` does not have `phone` in the `kwargs` dictionary. If any of the other classes need the `phone` parameter, we need to ensure it is in the dictionary that is passed. Worse, if we forget to do this, it will be tough to debug because the superclass will not complain, but will simply assign the default value (in this case, an empty string) to the variable.

There are a few ways to ensure that the variable is passed upwards. Assume the `Contact` class does, for some reason, need to be initialized with a `phone` parameter, and the `Friend` class will also need access to it. We can do any of the following:

- Don't include `phone` as an explicit keyword argument. Instead, leave it in the `kwargs` dictionary. `Friend` can look it up using the syntax `kwargs['phone']`. When it passes `**kwargs` to the `super` call, `phone` will still be in the dictionary.
- Make `phone` an explicit keyword argument but update the `kwargs` dictionary before passing it to `super`, using the standard dictionary syntax `kwargs['phone'] = phone`.
- Make `phone` an explicit keyword argument, but update the `kwargs` dictionary using the `kwargs.update` method. This is useful if you have several arguments to update. You can create the dictionary passed into `update` using either the `dict(phone=phone)` constructor, or the dictionary syntax `{'phone': phone}`.
- Make `phone` an explicit keyword argument, but pass it to the super call explicitly with the syntax `super().__init__(phone=phone, **kwargs)`.

We have covered many of the caveats involved with multiple inheritance in Python. When we need to account for all the possible situations, we have to plan for them and our code will get messy. Basic multiple inheritance can be handy but, in many cases, we may want to choose a more transparent way of combining two disparate classes, usually using composition or one of the design patterns we'll be covering in *Chapter 10*, *Python Design Patterns I* and *Chapter 11*, *Python Design Patterns II*.

Polymorphism

We were introduced to polymorphism in *Chapter 1*, *Object-oriented Design*. It is a fancy name describing a simple concept: different behaviors happen depending on which subclass is being used, without having to explicitly know what the subclass actually is. As an example, imagine a program that plays audio files. A media player might need to load an `AudioFile` object and then `play` it. We'd put a `play()` method on the object, which is responsible for decompressing or extracting the audio and routing it to the sound card and speakers. The act of playing an `AudioFile` could feasibly be as simple as:

```
audio_file.play()
```

However, the process of decompressing and extracting an audio file is very different for different types of files. The `.wav` files are stored uncompressed, while `.mp3`, `.wma`, and `.ogg` files all have totally different compression algorithms.

We can use inheritance with polymorphism to simplify the design. Each type of file can be represented by a different subclass of `AudioFile`, for example, `WavFile`, `MP3File`. Each of these would have a `play()` method, but that method would be implemented differently for each file to ensure the correct extraction procedure is followed. The media player object would never need to know which subclass of `AudioFile` it is referring to; it just calls `play()` and polymorphically lets the object take care of the actual details of playing. Let's look at a quick skeleton showing how this might look:

```
class AudioFile:
    def __init__(self, filename):
        if not filename.endswith(self.ext):
            raise Exception("Invalid file format")

        self.filename = filename

class MP3File(AudioFile):
    ext = "mp3"
    def play(self):
        print("playing {} as mp3".format(self.filename))

class WavFile(AudioFile):
    ext = "wav"
    def play(self):
```

```
            print("playing {} as wav".format(self.filename))

    class OggFile(AudioFile):
        ext = "ogg"
        def play(self):
            print("playing {} as ogg".format(self.filename))
```

All audio files check to ensure that a valid extension was given upon initialization. But did you notice how the `__init__` method in the parent class is able to access the `ext` class variable from different subclasses? That's polymorphism at work. If the filename doesn't end with the correct name, it raises an exception (exceptions will be covered in detail in the next chapter). The fact that `AudioFile` doesn't actually store a reference to the `ext` variable doesn't stop it from being able to access it on the subclass.

In addition, each subclass of `AudioFile` implements `play()` in a different way (this example doesn't actually play the music; audio compression algorithms really deserve a separate module!). This is also polymorphism in action. The media player can use the exact same code to play a file, no matter what type it is; it doesn't care what subclass of `AudioFile` it is looking at. The details of decompressing the audio file are *encapsulated*. If we test this example, it works as we would hope:

```
>>> ogg = OggFile("myfile.ogg")
>>> ogg.play()
playing myfile.ogg as ogg
>>> mp3 = MP3File("myfile.mp3")
>>> mp3.play()
playing myfile.mp3 as mp3
>>> not_an_mp3 = MP3File("myfile.ogg")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "polymorphic_audio.py", line 4, in __init__
    raise Exception("Invalid file format")
Exception: Invalid file format
```

See how `AudioFile.__init__` is able to check the file type without actually knowing what subclass it is referring to?

Polymorphism is actually one of the coolest things about object-oriented programming, and it makes some programming designs obvious that weren't possible in earlier paradigms. However, Python makes polymorphism less cool because of duck typing. Duck typing in Python allows us to use *any* object that provides the required behavior without forcing it to be a subclass. The dynamic nature of Python makes this trivial. The following example does not extend `AudioFile`, but it can be interacted with in Python using the exact same interface:

```
class FlacFile:
    def __init__(self, filename):
        if not filename.endswith(".flac"):
            raise Exception("Invalid file format")

        self.filename = filename

    def play(self):
        print("playing {} as flac".format(self.filename))
```

Our media player can play this object just as easily as one that extends `AudioFile`.

Polymorphism is one of the most important reasons to use inheritance in many object-oriented contexts. Because any objects that supply the correct interface can be used interchangeably in Python, it reduces the need for polymorphic common superclasses. Inheritance can still be useful for sharing code but, if all that is being shared is the public interface, duck typing is all that is required. This reduced need for inheritance also reduces the need for multiple inheritance; often, when multiple inheritance appears to be a valid solution, we can just use duck typing to mimic one of the multiple superclasses.

Of course, just because an object satisfies a particular interface (by providing required methods or attributes) does not mean it will simply work in all situations. It has to fulfill that interface in a way that makes sense in the overall system. Just because an object provides a `play()` method does not mean it will automatically work with a media player. For example, our chess AI object from *Chapter 1*, *Object-oriented Design*, may have a `play()` method that moves a chess piece. Even though it satisfies the interface, this class would likely break in spectacular ways if we tried to plug it into a media player!

Another useful feature of duck typing is that the duck-typed object only needs to provide those methods and attributes that are actually being accessed. For example, if we needed to create a fake file object to read data from, we can create a new object that has a `read()` method; we don't have to override the `write` method if the code that is going to interact with the object will only be reading from the file. More succinctly, duck typing doesn't need to provide the entire interface of an object that is available, it only needs to fulfill the interface that is actually accessed.

Abstract base classes

While duck typing is useful, it is not always easy to tell in advance if a class is going to fulfill the protocol you require. Therefore, Python introduced the idea of abstract base classes. **Abstract base classes**, or **ABCs**, define a set of methods and properties that a class must implement in order to be considered a duck-type instance of that class. The class can extend the abstract base class itself in order to be used as an instance of that class, but it must supply all the appropriate methods.

In practice, it's rarely necessary to create new abstract base classes, but we may find occasions to implement instances of existing ABCs. We'll cover implementing ABCs first, and then briefly see how to create your own if you should ever need to.

Using an abstract base class

Most of the abstract base classes that exist in the Python Standard Library live in the `collections` module. One of the simplest ones is the `Container` class. Let's inspect it in the Python interpreter to see what methods this class requires:

```
>>> from collections import Container
>>> Container.__abstractmethods__
frozenset(['__contains__'])
```

So, the `Container` class has exactly one abstract method that needs to be implemented, `__contains__`. You can issue `help(Container.__contains__)` to see what the function signature should look like:

```
Help on method __contains__ in module _abcoll:
__contains__(self, x) unbound _abcoll.Container method
```

So, we see that `__contains__` needs to take a single argument. Unfortunately, the help file doesn't tell us much about what that argument should be, but it's pretty obvious from the name of the ABC and the single method it implements that this argument is the value the user is checking to see if the container holds.

This method is implemented by `list`, `str`, and `dict` to indicate whether or not a given value is in that data structure. However, we can also define a silly container that tells us whether a given value is in the set of odd integers:

```
class OddContainer:
    def __contains__(self, x):
        if not isinstance(x, int) or not x % 2:
            return False
        return True
```

Now, we can instantiate an `OddContainer` object and determine that, even though we did not extend `Container`, the class *is a* `Container` object:

```
>>> from collections import Container
>>> odd_container = OddContainer()
>>> isinstance(odd_container, Container)
True
>>> issubclass(OddContainer, Container)
True
```

And that is why duck typing is way more awesome than classical polymorphism. We can create *is a* relationships without the overhead of using inheritance (or worse, multiple inheritance).

The interesting thing about the `Container` ABC is that any class that implements it gets to use the `in` keyword for free. In fact, `in` is just syntax sugar that delegates to the `__contains__` method. Any class that has a `__contains__` method is a `Container` and can therefore be queried by the `in` keyword, for example:

```
>>> 1 in odd_container
True
>>> 2 in odd_container
False
>>> 3 in odd_container
True
>>> "a string" in odd_container
False
```

Creating an abstract base class

As we saw earlier, it's not necessary to have an abstract base class to enable duck typing. However, imagine we were creating a media player with third-party plugins. It is advisable to create an abstract base class in this case to document what API the third-party plugins should provide. The `abc` module provides the tools you need to do this, but I'll warn you in advance, this requires some of Python's most arcane concepts:

```
import abc

class MediaLoader(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def play(self):
        pass

    @abc.abstractproperty
    def ext(self):
```

```
            pass

        @classmethod
        def __subclasshook__(cls, C):
            if cls is MediaLoader:
                attrs = set(dir(C))
                if set(cls.__abstractmethods__) <= attrs:
                    return True

            return NotImplemented
```

This is a complicated example that includes several Python features that won't be explained until later in this module. It is included here for completeness, but you don't need to understand all of it to get the gist of how to create your own ABC.

The first weird thing is the `metaclass` keyword argument that is passed into the class where you would normally see the list of parent classes. This is a rarely used construct from the mystic art of metaclass programming. We won't be covering metaclasses in this module, so all you need to know is that by assigning the `ABCMeta` metaclass, you are giving your class superpower (or at least superclass) abilities.

Next, we see the `@abc.abstractmethod` and `@abc.abstractproperty` constructs. These are Python decorators. We'll discuss those in *Chapter 5*, *When to Use Object-oriented Programming*. For now, just know that by marking a method or property as being abstract, you are stating that any subclass of this class must implement that method or supply that property in order to be considered a proper member of the class.

See what happens if you implement subclasses that do or don't supply those properties:

```
>>> class Wav(MediaLoader):
...     pass
...
>>> x = Wav()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Wav with abstract methods
ext, play
>>> class Ogg(MediaLoader):
...     ext = '.ogg'
...     def play(self):
...         pass
...
>>> o = Ogg()
```

Since the `Wav` class fails to implement the abstract attributes, it is not possible to instantiate that class. The class is still a legal abstract class, but you'd have to subclass it to actually do anything. The `Ogg` class supplies both attributes, so it instantiates cleanly.

Going back to the `MediaLoader` ABC, let's dissect that `__subclasshook__` method. It is basically saying that any class that supplies concrete implementations of all the abstract attributes of this ABC should be considered a subclass of `MediaLoader`, even if it doesn't actually inherit from the `MediaLoader` class.

More common object-oriented languages have a clear separation between the interface and the implementation of a class. For example, some languages provide an explicit `interface` keyword that allows us to define the methods that a class must have without any implementation. In such an environment, an abstract class is one that provides both an interface and a concrete implementation of some but not all methods. Any class can explicitly state that it implements a given interface.

Python's ABCs help to supply the functionality of interfaces without compromising on the benefits of duck typing.

Demystifying the magic

You can copy and paste the subclass code without understanding it if you want to make abstract classes that fulfill this particular contract. We'll cover most of the unusual syntaxes throughout the module, but let's go over it line by line to get an overview.

```
@classmethod
```

This decorator marks the method as a class method. It essentially says that the method can be called on a class instead of an instantiated object:

```
def __subclasshook__(cls, C):
```

This defines the `__subclasshook__` class method. This special method is called by the Python interpreter to answer the question, *Is the class `C` a subclass of this class?*

```
    if cls is MediaLoader:
```

We check to see if the method was called specifically on this class, rather than, say a subclass of this class. This prevents, for example, the `Wav` class from being thought of as a parent class of the `Ogg` class:

```
        attrs = set(dir(C))
```

All this line does is get the set of methods and properties that the class has, including any parent classes in its class hierarchy:

```
            if set(cls.__abstractmethods__) <= attrs:
```

This line uses set notation to see whether the set of abstract methods in this class have been supplied in the candidate class. Note that it doesn't check to see whether the methods have been implemented, just if they are there. Thus, it's possible for a class to be a subclass and yet still be an abstract class itself.

```
                return True
```

If all the abstract methods have been supplied, then the candidate class is a subclass of this class and we return `True`. The method can legally return one of the three values: `True`, `False`, or `NotImplemented`. `True` and `False` indicate that the class is or is not definitively a subclass of this class:

```
        return NotImplemented
```

If any of the conditionals have not been met (that is, the class is not `MediaLoader` or not all abstract methods have been supplied), then return `NotImplemented`. This tells the Python machinery to use the default mechanism (does the candidate class explicitly extend this class?) for subclass detection.

In short, we can now define the `Ogg` class as a subclass of the `MediaLoader` class without actually extending the `MediaLoader` class:

```
>>> class Ogg():
...     ext = '.ogg'
...     def play(self):
...         print("this will play an ogg file")
...
>>> issubclass(Ogg, MediaLoader)
True
>>> isinstance(Ogg(), MediaLoader)
True
```

Case study

Let's try to tie everything we've learned together with a larger example. We'll be designing a simple real estate application that allows an agent to manage properties available for purchase or rent. There will be two types of properties: apartments and houses. The agent needs to be able to enter a few relevant details about new properties, list all currently available properties, and mark a property as sold or rented. For brevity, we won't worry about editing property details or reactivating a property after it is sold.

The project will allow the agent to interact with the objects using the Python interpreter prompt. In this world of graphical user interfaces and web applications, you might be wondering why we're creating such old-fashioned looking programs. Simply put, both windowed programs and web applications require a lot of overhead knowledge and boilerplate code to make them do what is required. If we were developing software using either of these paradigms, we'd get so lost in GUI programming or web programming that we'd lose sight of the object-oriented principles we're trying to master.

Luckily, most GUI and web frameworks utilize an object-oriented approach, and the principles we're studying now will help in understanding those systems in the future. We'll discuss them both briefly in *Chapter 13*, *Concurrency*, but complete details are far beyond the scope of a single module.

Looking at our requirements, it seems like there are quite a few nouns that might represent classes of objects in our system. Clearly, we'll need to represent a property. Houses and apartments may need separate classes. Rentals and purchases also seem to require separate representation. Since we're focusing on inheritance right now, we'll be looking at ways to share behavior using inheritance or multiple inheritance.

`House` and `Apartment` are both types of properties, so `Property` can be a superclass of those two classes. `Rental` and `Purchase` will need some extra thought; if we use inheritance, we'll need to have separate classes, for example, for `HouseRental` and `HousePurchase`, and use multiple inheritance to combine them. This feels a little clunky compared to a composition or association-based design, but let's run with it and see what we come up with.

Now then, what attributes might be associated with a `Property` class? Regardless of whether it is an apartment or a house, most people will want to know the square footage, number of bedrooms, and number of bathrooms. (There are numerous other attributes that might be modeled, but we'll keep it simple for our prototype.)

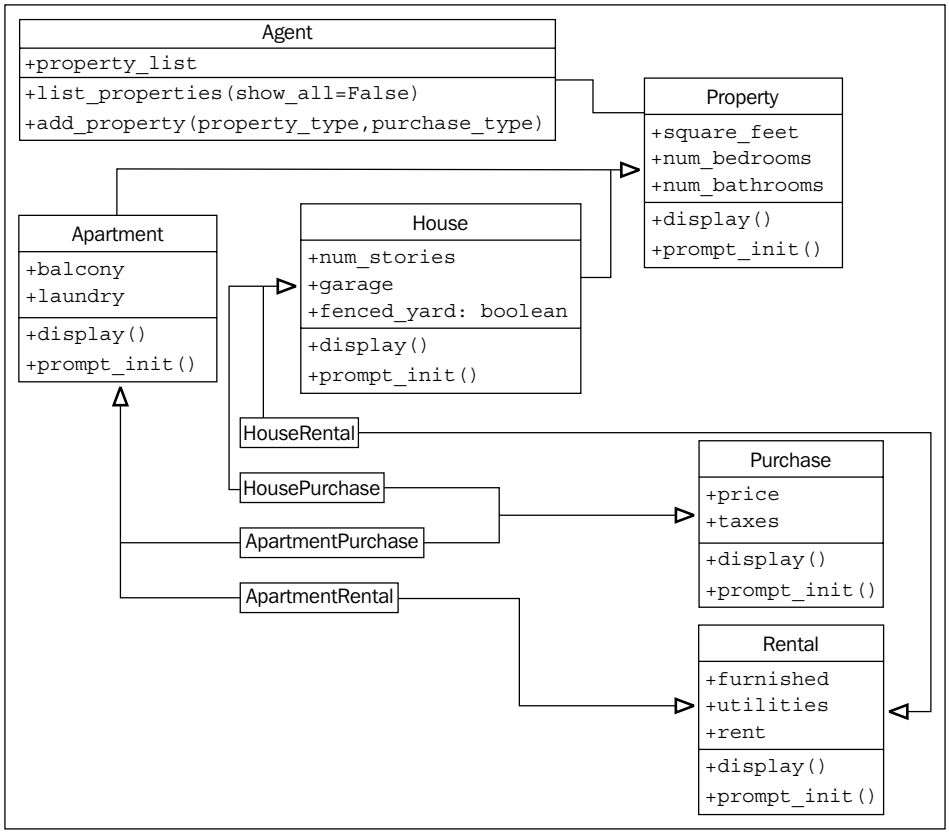
If the property is a house, it will want to advertise the number of stories, whether it has a garage (attached, detached, or none), and whether the yard is fenced. An apartment will want to indicate if it has a balcony, and if the laundry is ensuite, coin, or off-site.

Both property types will require a method to display the characteristics of that property. At the moment, no other behaviors are apparent.

Rental properties will need to store the rent per month, whether the property is furnished, and whether utilities are included, and if not, what they are estimated to be. Properties for purchase will need to store the purchase price and estimated annual property taxes. For our application, we'll only need to display this data, so we can get away with just adding a `display()` method similar to that used in the other classes.

Finally, we'll need an `Agent` object that holds a list of all properties, displays those properties, and allows us to create new ones. Creating properties will entail prompting the user for the relevant details for each property type. This could be done in the `Agent` object, but then `Agent` would need to know a lot of information about the types of properties. This is not taking advantage of polymorphism. Another alternative would be to put the prompts in the initializer or even a constructor for each class, but this would not allow the classes to be applied in a GUI or web application in the future. A better idea is to create a static method that does the prompting and returns a dictionary of the prompted parameters. Then, all the `Agent` has to do is prompt the user for the type of property and payment method, and ask the correct class to instantiate itself.

That's a lot of designing! The following class diagram may communicate our design decisions a little more clearly:



Wow, that's a lot of inheritance arrows! I don't think it would be possible to add another level of inheritance without crossing arrows. Multiple inheritance is a messy business, even at the design stage.

The trickiest aspects of these classes is going to be ensuring superclass methods get called in the inheritance hierarchy. Let's start with the `Property` implementation:

```
class Property:
    def __init__(self, square_feet='', beds='',
            baths='', **kwargs):
        super().__init__(**kwargs)
        self.square_feet = square_feet
        self.num_bedrooms = beds
        self.num_baths = baths

    def display(self):
        print("PROPERTY DETAILS")
        print("================")
        print("square footage: {}".format(self.square_feet))
        print("bedrooms: {}".format(self.num_bedrooms))
        print("bathrooms: {}".format(self.num_baths))
        print()

    def prompt_init():
        return dict(square_feet=input("Enter the square feet: "),
                beds=input("Enter number of bedrooms: "),
                baths=input("Enter number of baths: "))
    prompt_init = staticmethod(prompt_init)
```

This class is pretty straightforward. We've already added the extra `**kwargs` parameter to `__init__` because we know it's going to be used in a multiple inheritance situation. We've also included a call to `super().__init__` in case we are not the last call in the multiple inheritance chain. In this case, we're *consuming* the keyword arguments because we know they won't be needed at other levels of the inheritance hierarchy.

We see something new in the `prompt_init` method. This method is made into a static method immediately after it is initially created. Static methods are associated only with a class (something like class variables), rather than a specific object instance. Hence, they have no `self` argument. Because of this, the `super` keyword won't work (there is no parent object, only a parent class), so we simply call the static method on the parent class directly. This method uses the Python `dict` constructor to create a dictionary of values that can be passed into `__init__`. The value for each key is prompted with a call to `input`.

The `Apartment` class extends `Property`, and is similar in structure:

```
class Apartment(Property):
    valid_laundries = ("coin", "ensuite", "none")
    valid_balconies = ("yes", "no", "solarium")

    def __init__(self, balcony='', laundry='', **kwargs):
        super().__init__(**kwargs)
        self.balcony = balcony
        self.laundry = laundry

    def display(self):
        super().display()
        print("APARTMENT DETAILS")
        print("laundry: %s" % self.laundry)
        print("has balcony: %s" % self.balcony)

    def prompt_init():
        parent_init = Property.prompt_init()
        laundry = ''
        while laundry.lower() not in \
                Apartment.valid_laundries:
            laundry = input("What laundry facilities does "
                    "the property have? ({})".format(
                    ", ".join(Apartment.valid_laundries)))
        balcony = ''
        while balcony.lower() not in \
                Apartment.valid_balconies:
            balcony = input(
                "Does the property have a balcony? "
                "({})".format(
                ", ".join(Apartment.valid_balconies)))
        parent_init.update({
            "laundry": laundry,
            "balcony": balcony
        })
        return parent_init
    prompt_init = staticmethod(prompt_init)
```

The `display()` and `__init__()` methods call their respective parent class methods using `super()` to ensure the `Property` class is properly initialized.

The `prompt_init` static method is now getting dictionary values from the parent class, and then adding some additional values of its own. It calls the `dict.update` method to merge the new dictionary values into the first one. However, that `prompt_init` method is looking pretty ugly; it loops twice until the user enters a valid input using structurally similar code but different variables. It would be nice to extract this validation logic so we can maintain it in only one location; it will likely also be useful to later classes.

With all the talk on inheritance, we might think this is a good place to use a mixin. Instead, we have a chance to study a situation where inheritance is not the best solution. The method we want to create will be used in a static method. If we were to inherit from a class that provided validation functionality, the functionality would also have to be provided as a static method that did not access any instance variables on the class. If it doesn't access any instance variables, what's the point of making it a class at all? Why don't we just make this validation functionality a module-level function that accepts an input string and a list of valid answers, and leave it at that?

Let's explore what this validation function would look like:

```
def get_valid_input(input_string, valid_options):
    input_string += " ({}) ".format(", ".join(valid_options))
    response = input(input_string)
    while response.lower() not in valid_options:
        response = input(input_string)
    return response
```

We can test this function in the interpreter, independent of all the other classes we've been working on. This is a good sign, it means different pieces of our design are not tightly coupled to each other and can later be improved independently, without affecting other pieces of code.

```
>>> get_valid_input("what laundry?", ("coin", "ensuite", "none"))
what laundry? (coin, ensuite, none) hi
what laundry? (coin, ensuite, none) COIN
'COIN'
```

Now, let's quickly update our `Apartment.prompt_init` method to use this new function for validation:

```
        def prompt_init():
            parent_init = Property.prompt_init()
            laundry = get_valid_input(
```

```
                "What laundry facilities does "
                "the property have? ",
                Apartment.valid_laundries)
        balcony = get_valid_input(
            "Does the property have a balcony? ",
            Apartment.valid_balconies)
        parent_init.update({
            "laundry": laundry,
            "balcony": balcony
        })
        return parent_init
    prompt_init = staticmethod(prompt_init)
```

That's much easier to read (and maintain!) than our original version. Now we're ready to build the `House` class. This class has a parallel structure to `Apartment`, but refers to different prompts and variables:

```
class House(Property):
    valid_garage = ("attached", "detached", "none")
    valid_fenced = ("yes", "no")

    def __init__(self, num_stories='',
            garage='', fenced='', **kwargs):
        super().__init__(**kwargs)
        self.garage = garage
        self.fenced = fenced
        self.num_stories = num_stories

    def display(self):
        super().display()
        print("HOUSE DETAILS")
        print("# of stories: {}".format(self.num_stories))
        print("garage: {}".format(self.garage))
        print("fenced yard: {}".format(self.fenced))


    def prompt_init():
        parent_init = Property.prompt_init()
        fenced = get_valid_input("Is the yard fenced? ",
                    House.valid_fenced)
        garage = get_valid_input("Is there a garage? ",
                House.valid_garage)
```

```
        num_stories = input("How many stories? ")

        parent_init.update({
            "fenced": fenced,
            "garage": garage,
            "num_stories": num_stories
        })
        return parent_init
    prompt_init = staticmethod(prompt_init)
```

There's nothing new to explore here, so let's move on to the `Purchase` and `Rental` classes. In spite of having apparently different purposes, they are also similar in design to the ones we just discussed:

```
class Purchase:
    def __init__(self, price='', taxes='', **kwargs):
        super().__init__(**kwargs)
        self.price = price
        self.taxes = taxes

    def display(self):
        super().display()
        print("PURCHASE DETAILS")
        print("selling price: {}".format(self.price))
        print("estimated taxes: {}".format(self.taxes))

    def prompt_init():
        return dict(
            price=input("What is the selling price? "),
            taxes=input("What are the estimated taxes? "))
    prompt_init = staticmethod(prompt_init)

class Rental:
    def __init__(self, furnished='', utilities='',
            rent='', **kwargs):
        super().__init__(**kwargs)
        self.furnished = furnished
        self.rent = rent
        self.utilities = utilities

    def display(self):
        super().display()
        print("RENTAL DETAILS")
```

```
            print("rent: {}".format(self.rent))
            print("estimated utilities: {}".format(
                self.utilities))
            print("furnished: {}".format(self.furnished))

        def prompt_init():
            return dict(
                rent=input("What is the monthly rent? "),
                utilities=input(
                    "What are the estimated utilities? "),
                furnished = get_valid_input(
                    "Is the property furnished? ",
                        ("yes", "no")))
        prompt_init = staticmethod(prompt_init)
```

These two classes don't have a superclass (other than `object`), but we still call `super().__init__` because they are going to be combined with the other classes, and we don't know what order the `super` calls will be made in. The interface is similar to that used for `House` and `Apartment`, which is very useful when we combine the functionality of these four classes in separate subclasses. For example:

```
class HouseRental(Rental, House):
    def prompt_init():
        init = House.prompt_init()
        init.update(Rental.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)
```

This is slightly surprising, as the class on its own has neither an `__init__` nor `display` method! Because both parent classes appropriately call `super` in these methods, we only have to extend those classes and the classes will behave in the correct order. This is not the case with `prompt_init`, of course, since it is a static method that does not call `super`, so we implement this one explicitly. We should test this class to make sure it is behaving properly before we write the other three combinations:

```
>>> init = HouseRental.prompt_init()
Enter the square feet: 1
Enter number of bedrooms: 2
Enter number of baths: 3
Is the yard fenced?  (yes, no) no
Is there a garage?  (attached, detached, none) none
How many stories? 4
What is the monthly rent? 5
What are the estimated utilities? 6
Is the property furnished?  (yes, no) no
```

```
>>> house = HouseRental(**init)
>>> house.display()
PROPERTY DETAILS
================
square footage: 1
bedrooms: 2
bathrooms: 3

HOUSE DETAILS
# of stories: 4
garage: none
fenced yard: no

RENTAL DETAILS
rent: 5
estimated utilities: 6
furnished: no
```

It looks like it is working fine. The `prompt_init` method is prompting for initializers to all the super classes, and `display()` is also cooperatively calling all three superclasses.

The order of the inherited classes in the preceding example is important. If we had written `class HouseRental(House, Rental)` instead of `class HouseRental(Rental, House)`, `display()` would not have called `Rental.display()`! When `display` is called on our version of `HouseRental`, it refers to the `Rental` version of the method, which calls `super.display()` to get the `House` version, which again calls `super.display()` to get the property version. If we reversed it, `display` would refer to the `House` class's `display()`. When super is called, it calls the method on the `Property` parent class. But `Property` does not have a call to `super` in its `display` method. This means `Rental` class's `display` method would not be called! By placing the inheritance list in the order we did, we ensure that `Rental` calls `super`, which then takes care of the `House` side of the hierarchy. You might think we could have added a `super` call to `Property.display()`, but that will fail because the next superclass of `Property` is `object`, and `object` does not have a `display` method. Another way to fix this is to allow `Rental` and `Purchase` to extend the `Property` class instead of deriving directly from `object`. (Or we could modify the method resolution order dynamically, but that is beyond the scope of this module.)

Now that we have tested it, we are prepared to create the rest of our combined subclasses:

```
class ApartmentRental(Rental, Apartment):
    def prompt_init():
        init = Apartment.prompt_init()
        init.update(Rental.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)

class ApartmentPurchase(Purchase, Apartment):
    def prompt_init():
        init = Apartment.prompt_init()
        init.update(Purchase.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)

class HousePurchase(Purchase, House):
    def prompt_init():
        init = House.prompt_init()
        init.update(Purchase.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)
```

That should be the most intense designing out of our way! Now all we have to do is create the `Agent` class, which is responsible for creating new listings and displaying existing ones. Let's start with the simpler storing and listing of properties:

```
class Agent:
    def __init__(self):
        self.property_list = []

    def display_properties(self):
        for property in self.property_list:
            property.display()
```

Adding a property will require first querying the type of property and whether property is for purchase or rental. We can do this by displaying a simple menu. Once this has been determined, we can extract the correct subclass and prompt for all the details using the `prompt_init` hierarchy we've already developed. Sounds simple? It is. Let's start by adding a dictionary class variable to the `Agent` class:

```
type_map = {
    ("house", "rental"): HouseRental,
    ("house", "purchase"): HousePurchase,
    ("apartment", "rental"): ApartmentRental,
    ("apartment", "purchase"): ApartmentPurchase
    }
```

That's some pretty funny looking code. This is a dictionary, where the keys are tuples of two distinct strings, and the values are class objects. Class objects? Yes, classes can be passed around, renamed, and stored in containers just like *normal* objects or primitive data types. With this simple dictionary, we can simply hijack our earlier `get_valid_input` method to ensure we get the correct dictionary keys and look up the appropriate class, like this:

```
def add_property(self):
    property_type = get_valid_input(
            "What type of property? ",
            ("house", "apartment")).lower()
    payment_type = get_valid_input(
            "What payment type? ",
            ("purchase", "rental")).lower()

    PropertyClass = self.type_map[
        (property_type, payment_type)]
    init_args = PropertyClass.prompt_init()
    self.property_list.append(PropertyClass(**init_args))
```

This may look a bit funny too! We look up the class in the dictionary and store it in a variable named `PropertyClass`. We don't know exactly which class is available, but the class knows itself, so we can polymorphically call `prompt_init` to get a dictionary of values appropriate to pass into the constructor. Then we use the keyword argument syntax to convert the dictionary into arguments and construct the new object to load the correct data.

Now our user can use this `Agent` class to add and view lists of properties. It wouldn't take much work to add features to mark a property as available or unavailable or to edit and remove properties. Our prototype is now in a good enough state to take to a real estate `agent` and demonstrate its functionality. Here's how a demo session might work:

```
>>> agent = Agent()
>>> agent.add_property()
What type of property?  (house, apartment) house
What payment type?  (purchase, rental) rental
Enter the square feet: 900
Enter number of bedrooms: 2
Enter number of baths: one and a half
Is the yard fenced?  (yes, no) yes
Is there a garage?  (attached, detached, none) detached
How many stories? 1
What is the monthly rent? 1200
What are the estimated utilities? included
Is the property furnished?  (yes, no) no
>>> agent.add_property()
What type of property?  (house, apartment) apartment
What payment type?  (purchase, rental) purchase
Enter the square feet: 800
Enter number of bedrooms: 3
Enter number of baths: 2
What laundry facilities does the property have?  (coin, ensuite,
one) ensuite
Does the property have a balcony? (yes, no, solarium) yes
What is the selling price? $200,000
What are the estimated taxes? 1500
>>> agent.display_properties()
```

```
PROPERTY DETAILS
================
square footage: 900
bedrooms: 2
bathrooms: one and a half

HOUSE DETAILS
# of stories: 1
garage: detached
fenced yard: yes
RENTAL DETAILS
rent: 1200
estimated utilities: included
furnished: no
PROPERTY DETAILS
================
square footage: 800
bedrooms: 3
bathrooms: 2

APARTMENT DETAILS
laundry: ensuite
has balcony: yes
PURCHASE DETAILS
selling price: $200,000
estimated taxes: 1500
```

Your Coding Challenge

Look around you at some of the physical objects in your workspace and see if you can describe them in an inheritance hierarchy. Humans have been dividing the world into taxonomies like this for centuries, so it shouldn't be difficult. Are there any non-obvious inheritance relationships between classes of objects? If you were to model these objects in a computer application, what properties and methods would they share? Which ones would have to be polymorphically overridden? What properties would be completely different between them?

Now, write some code. No, not for the physical hierarchy; that's boring. Physical items have more properties than methods. Just think about a pet programming project you've wanted to tackle in the past year, but never got around to. For whatever problem you want to solve, try to think of some basic inheritance relationships. Then implement them. Make sure that you also pay attention to the sorts of relationships that you actually don't need to use inheritance for. Are there any places where you might want to use multiple inheritance? Are you sure? Can you see any place you would want to use a mixin? Try to knock together a quick prototype. It doesn't have to be useful or even partially working. You've seen how you can test code using python -i already; just write some code and test it in the interactive interpreter. If it works, write some more. If it doesn't, fix it!

Ankita Thakur

Your Course Guide

Now, take a look at the real estate example. This turned out to be quite an effective use of multiple inheritance. I have to admit though, I had my doubts when I started the design. Have a look at the original problem and see if you can come up with another design to solve it that uses only single inheritance. How would you do it with abstract base classes? What about a design that doesn't use inheritance at all? Which do you think is the most elegant solution? Elegance is a primary goal in Python development, but each programmer has a different opinion as to what is the most elegant solution. Some people tend to think and understand problems most clearly using composition, while others find multiple inheritance to be the most useful model.

Finally, try adding some new features to the three designs. Whatever features strike your fancy are fine. I'd like to see a way to differentiate between available and unavailable properties, for starters. It's not of much use to me if it's already rented!

Ankita Thakur

Your Course Guide

Which design is easiest to extend? Which is hardest? If somebody asked you why you thought this, would you be able to explain yourself?

Summary of Module 1 Chapter 4

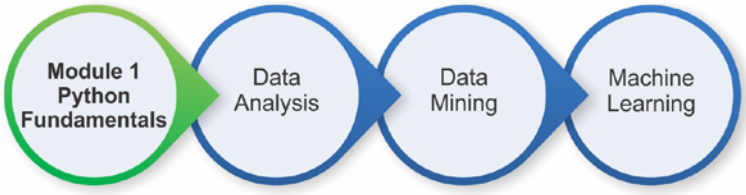
Ankita Thakur

Your Course Guide

We've gone from simple inheritance, one of the most useful tools in the object-oriented programmer's toolbox, all the way through to multiple inheritance, one of the most complicated. Inheritance can be used to add functionality to existing classes and built-ins using inheritance. Abstracting similar code into a parent class can help increase maintainability. Methods on parent classes can be called using super and argument lists must be formatted safely for these calls to work when using multiple inheritance.

In the next chapter, we'll cover the subtle art of handling exceptional circumstances.

Your Progress through the Course So Far



5
Expecting the Unexpected

Programs are very fragile. It would be ideal if code always returned a valid result, but sometimes a valid result can't be calculated. For example, it's not possible to divide by zero, or to access the eighth item in a five-item list.

In the old days, the only way around this was to rigorously check the inputs for every function to make sure they made sense. Typically, functions had special return values to indicate an error condition; for example, they could return a negative number to indicate that a positive value couldn't be calculated. Different numbers might mean different errors occurred. Any code that called this function would have to explicitly check for an error condition and act accordingly. A lot of code didn't bother to do this, and programs simply crashed. However, in the object-oriented world, this is not the case.

In this chapter, we will study **exceptions**, special error objects that only need to be handled when it makes sense to handle them. In particular, we will cover:

- How to cause an exception to occur
- How to recover when an exception has occurred
- How to handle different exception types in different ways
- Cleaning up when an exception has occurred
- Creating new types of exception
- Using the exception syntax for flow control

Raising exceptions

In principle, an exception is just an object. There are many different exception classes available, and we can easily define more of our own. The one thing they all have in common is that they inherit from a built-in class called `BaseException`. These exception objects become special when they are handled inside the program's flow of control. When an exception occurs, everything that was supposed to happen doesn't happen, unless it was supposed to happen when an exception occurred. Make sense? Don't worry, it will!

The easiest way to cause an exception to occur is to do something silly! Chances are you've done this already and seen the exception output. For example, any time Python encounters a line in your program that it can't understand, it bails with `SyntaxError`, which is a type of exception. Here's a common one:

```
>>> print "hello world"
  File "<stdin>", line 1
    print "hello world"
                      ^
SyntaxError: invalid syntax
```

This `print` statement was a valid command in Python 2 and previous versions, but in Python 3, because `print` is now a function, we have to enclose the arguments in parenthesis. So, if we type the preceding command into a Python 3 interpreter, we get the `SyntaxError`.

In addition to `SyntaxError`, some other common exceptions, which we can handle, are shown in the following example:

```
>>> x = 5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero

>>> lst = [1,2,3]
>>> print(lst[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> lst + 2
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list

>>> lst.add
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'add'

>>> d = {'a': 'hello'}
>>> d['b']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'b'

>>> print(this_is_not_a_var)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'this_is_not_a_var' is not defined
```

Sometimes these exceptions are indicators of something wrong in our program (in which case we would go to the indicated line number and fix it), but they also occur in legitimate situations. A `ZeroDivisionError` doesn't always mean we received an invalid input. It could also mean we have received a different input. The user may have entered a zero by mistake, or on purpose, or it may represent a legitimate value, such as an empty bank account or the age of a newborn child.

You may have noticed all the preceding built-in exceptions end with the name `Error`. In Python, the words `error` and `exception` are used almost interchangeably. Errors are sometimes considered more dire than exceptions, but they are dealt with in exactly the same way. Indeed, all the error classes in the preceding example have `Exception` (which extends `BaseException`) as their superclass.

Raising an exception

We'll get to handling exceptions in a minute, but first, let's discover what we should do if we're writing a program that needs to inform the user or a calling function that the inputs are somehow invalid. Wouldn't it be great if we could use the same mechanism that Python uses? Well, we can! Here's a simple class that adds items to a list only if they are even numbered integers:

```
    class EvenOnly(list):
        def append(self, integer):
            if not isinstance(integer, int):
                raise TypeError("Only integers can be added")
            if integer % 2:
                raise ValueError("Only even numbers can be added")
            super().append(integer)
```

This class extends the `list` built-in, as we discussed in *Chapter 2, Objects in Python*, and overrides the `append` method to check two conditions that ensure the item is an even integer. We first check if the input is an instance of the `int` type, and then use the modulus operator to ensure it is divisible by two. If either of the two conditions is not met, the `raise` keyword causes an exception to occur. The `raise` keyword is simply followed by the object being raised as an exception. In the preceding example, two objects are newly constructed from the built-in classes `TypeError` and `ValueError`. The raised object could just as easily be an instance of a new exception class we create ourselves (we'll see how shortly), an exception that was defined elsewhere, or even an exception object that has been previously raised and handled. If we test this class in the Python interpreter, we can see that it is outputting useful error information when exceptions occur, just as before:

```
>>> e = EvenOnly()
>>> e.append("a string")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "even_integers.py", line 7, in add
    raise TypeError("Only integers can be added")
TypeError: Only integers can be added

>>> e.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "even_integers.py", line 9, in add
    raise ValueError("Only even numbers can be added")
ValueError: Only even numbers can be added
>>> e.append(2)
```

While this class is effective for demonstrating exceptions in action, it isn't very good at its job. It is still possible to get other values into the list using index notation or slice notation. This can all be avoided by overriding other appropriate methods, some of which are double-underscore methods.

The effects of an exception

When an exception is raised, it appears to stop program execution immediately. Any lines that were supposed to run after the exception is raised are not executed, and unless the exception is dealt with, the program will exit with an error message. Take a look at this simple function:

```
def no_return():
    print("I am about to raise an exception")
    raise Exception("This is always raised")
    print("This line will never execute")
    return "I won't be returned"
```

If we execute this function, we see that the first `print` call is executed and then the exception is raised. The second `print` statement is never executed, and the `return` statement never executes either:

```
>>> no_return()
I am about to raise an exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "exception_quits.py", line 3, in no_return
    raise Exception("This is always raised")
Exception: This is always raised
```

Furthermore, if we have a function that calls another function that raises an exception, nothing will be executed in the first function after the point where the second function was called. Raising an exception stops all execution right up through the function call stack until it is either handled or forces the interpreter to exit. To demonstrate, let's add a second function that calls the earlier one:

```
def call_exceptor():
    print("call_exceptor starts here...")
    no_return()
    print("an exception was raised...")
    print("...so these lines don't run")
```

When we call this function, we see that the first `print` statement executes, as well as the first line in the `no_return` function. But once the exception is raised, nothing else executes:

```
>>> call_exceptor()
call_exceptor starts here...
I am about to raise an exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "method_calls_excepting.py", line 9, in call_exceptor
    no_return()
  File "method_calls_excepting.py", line 3, in no_return
    raise Exception("This is always raised")
Exception: This is always raised
```

We'll soon see that when the interpreter is not actually taking a shortcut and exiting immediately, we can react to and deal with the exception inside either method. Indeed, exceptions can be handled at any level after they are initially raised.

Look at the exception's output (called a traceback) from bottom to top, and notice how both methods are listed. Inside `no_return`, the exception is initially raised. Then, just above that, we see that inside `call_exceptor`, that pesky `no_return` function was called and the exception bubbled up to the calling method. From there, it went up one more level to the main interpreter, which, not knowing what else to do with it, gave up and printed a traceback.

Handling exceptions

Now let's look at the tail side of the exception coin. If we encounter an exception situation, how should our code react to or recover from it? We handle exceptions by wrapping any code that might throw one (whether it is exception code itself, or a call to any function or method that may have an exception raised inside it) inside a `try...except` clause. The most basic syntax looks like this:

```
try:
    no_return()
except:
    print("I caught an exception")
print("executed after the exception")
```

If we run this simple script using our existing `no_return` function, which as we know very well, always throws an exception, we get this output:

```
I am about to raise an exception
I caught an exception
executed after the exception
```

The `no_return` function happily informs us that it is about to raise an exception, but we fooled it and caught the exception. Once caught, we were able to clean up after ourselves (in this case, by outputting that we were handling the situation), and continue on our way, with no interference from that offensive function. The remainder of the code in the `no_return` function still went unexecuted, but the code that called the function was able to recover and continue.

Note the indentation around `try` and `except`. The `try` clause wraps any code that might throw an exception. The `except` clause is then back on the same indentation level as the `try` line. Any code to handle the exception is indented after the `except` clause. Then normal code resumes at the original indentation level.

The problem with the preceding code is that it will catch any type of exception. What if we were writing some code that could raise both a `TypeError` and a `ZeroDivisionError`? We might want to catch the `ZeroDivisionError`, but let the `TypeError` propagate to the console. Can you guess the syntax?

Here's a rather silly function that does just that:

```
def funny_division(divider):
    try:
        return 100 / divider
    except ZeroDivisionError:
        return "Zero is not a good idea!"

print(funny_division(0))
print(funny_division(50.0))
print(funny_division("hello"))
```

The function is tested with `print` statements that show it behaving as expected:

```
Zero is not a good idea!
2.0
Traceback (most recent call last):
  File "catch_specific_exception.py", line 9, in <module>
    print(funny_division("hello"))
  File "catch_specific_exception.py", line 3, in funny_division
    return 100 / anumber
TypeError: unsupported operand type(s) for /: 'int' and 'str'.
```

The first line of output shows that if we enter `0`, we get properly mocked. If we call with a valid number (note that it's not an integer, but it's still a valid divisor), it operates correctly. Yet if we enter a string (you were wondering how to get a `TypeError`, weren't you?), it fails with an exception. If we had used an empty `except` clause that didn't specify a `ZeroDivisionError`, it would have accused us of dividing by zero when we sent it a string, which is not a proper behavior at all.

We can even catch two or more different exceptions and handle them with the same code. Here's an example that raises three different types of exception. It handles `TypeError` and `ZeroDivisionError` with the same exception handler, but it may also raise a `ValueError` if you supply the number `13`:

```
    def funny_division2(anumber):
        try:
            if anumber == 13:
                raise ValueError("13 is an unlucky number")
            return 100 / anumber
        except (ZeroDivisionError, TypeError):
            return "Enter a number other than zero"

    for val in (0, "hello", 50.0, 13):

        print("Testing {}:".format(val), end=" ")
        print(funny_division2(val))
```

The `for` loop at the bottom loops over several test inputs and prints the results. If you're wondering about that `end` argument in the `print` statement, it just turns the default trailing newline into a space so that it's joined with the output from the next line. Here's a run of the program:

```
Testing 0: Enter a number other than zero
Testing hello: Enter a number other than zero
Testing 50.0: 2.0
Testing 13: Traceback (most recent call last):
  File "catch_multiple_exceptions.py", line 11, in <module>
    print(funny_division2(val))
  File "catch_multiple_exceptions.py", line 4, in funny_division2
    raise ValueError("13 is an unlucky number")
ValueError: 13 is an unlucky number
```

The number `0` and the string are both caught by the `except` clause, and a suitable error message is printed. The exception from the number `13` is not caught because it is a `ValueError`, which was not included in the types of exceptions being handled. This is all well and good, but what if we want to catch different exceptions and do different things with them? Or maybe we want to do something with an exception and then allow it to continue to bubble up to the parent function, as if it had never been caught? We don't need any new syntax to deal with these cases. It's possible to stack `except` clauses, and only the first match will be executed. For the second question, the `raise` keyword, with no arguments, will reraise the last exception if we're already inside an exception handler. Observe in the following code:

```
def funny_division3(anumber):
    try:
        if anumber == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / anumber
    except ZeroDivisionError:
        return "Enter a number other than zero"
    except TypeError:
        return "Enter a numerical value"
    except ValueError:
        print("No, No, not 13!")
        raise
```

The last line reraises the `ValueError`, so after outputting `No, No, not 13!`, it will raise the exception again; we'll still get the original stack trace on the console.

If we stack exception clauses like we did in the preceding example, only the first matching clause will be run, even if more than one of them fits. How can more than one clause match? Remember that exceptions are objects, and can therefore be subclassed. As we'll see in the next section, most exceptions extend the `Exception` class (which is itself derived from `BaseException`). If we catch `Exception` before we catch `TypeError`, then only the `Exception` handler will be executed, because `TypeError` is an `Exception` by inheritance.

This can come in handy in cases where we want to handle some exceptions specifically, and then handle all remaining exceptions as a more general case. We can simply catch `Exception` after catching all the specific exceptions and handle the general case there.

Sometimes, when we catch an exception, we need a reference to the `Exception` object itself. This most often happens when we define our own exceptions with custom arguments, but can also be relevant with standard exceptions. Most exception classes accept a set of arguments in their constructor, and we might want to access those attributes in the exception handler. If we define our own exception class, we can even call custom methods on it when we catch it. The syntax for capturing an exception as a variable uses the `as` keyword:

```
try:
    raise ValueError("This is an argument")
except ValueError as e:
    print("The exception arguments were", e.args)
```

If we run this simple snippet, it prints out the string argument that we passed into `ValueError` upon initialization.

We've seen several variations on the syntax for handling exceptions, but we still don't know how to execute code regardless of whether or not an exception has occurred. We also can't specify code that should be executed only if an exception does not occur. Two more keywords, `finally` and `else`, can provide the missing pieces. Neither one takes any extra arguments. The following example randomly picks an exception to throw and raises it. Then some not-so-complicated exception handling code is run that illustrates the newly introduced syntax:

```
import random
some_exceptions = [ValueError, TypeError, IndexError, None]

try:
    choice = random.choice(some_exceptions)
    print("raising {}".format(choice))
    if choice:
        raise choice("An error")
except ValueError:
    print("Caught a ValueError")
except TypeError:
    print("Caught a TypeError")
except Exception as e:
    print("Caught some other error: %s" %
        ( e.__class__.__name__))
else:
    print("This code called if there is no exception")
finally:
    print("This cleanup code is always called")
```

If we run this example—which illustrates almost every conceivable exception handling scenario—a few times, we'll get different output each time, depending on which exception `random` chooses. Here are some example runs:

```
$ python finally_and_else.py
raising None
This code called if there is no exception
This cleanup code is always called

$ python finally_and_else.py
raising <class 'TypeError'>
Caught a TypeError
This cleanup code is always called

$ python finally_and_else.py
raising <class 'IndexError'>
Caught some other error: IndexError
This cleanup code is always called

$ python finally_and_else.py
raising <class 'ValueError'>
Caught a ValueError
This cleanup code is always called
```

Note how the `print` statement in the `finally` clause is executed no matter what happens. This is extremely useful when we need to perform certain tasks after our code has finished running (even if an exception has occurred). Some common examples include:

- Cleaning up an open database connection
- Closing an open file
- Sending a closing handshake over the network

The `finally` clause is also very important when we execute a `return` statement from inside a `try` clause. The `finally` handle will still be executed before the value is returned.

Also, pay attention to the output when no exception is raised: both the `else` and the `finally` clauses are executed. The `else` clause may seem redundant, as the code that should be executed only when no exception is raised could just be placed after the entire `try...except` block. The difference is that the `else` block will still be executed if an exception is caught and handled. We'll see more on this when we discuss using exceptions as flow control later.

Any of the `except`, `else`, and `finally` clauses can be omitted after a `try` block (although `else` by itself is invalid). If you include more than one, the `except` clauses must come first, then the `else` clause, with the `finally` clause at the end. The order of the `except` clauses normally goes from most specific to most generic.

The exception hierarchy

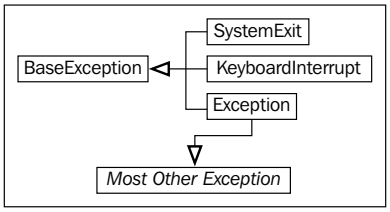
We've already seen several of the most common built-in exceptions, and you'll probably encounter the rest over the course of your regular Python development. As we noticed earlier, most exceptions are subclasses of the `Exception` class. But this is not true of all exceptions. `Exception` itself actually inherits from a class called `BaseException`. In fact, all exceptions must extend the `BaseException` class or one of its subclasses.

There are two key exceptions, `SystemExit` and `KeyboardInterrupt`, that derive directly from `BaseException` instead of `Exception`. The `SystemExit` exception is raised whenever the program exits naturally, typically because we called the `sys.exit` function somewhere in our code (for example, when the user selected an exit menu item, clicked the "close" button on a window, or entered a command to shut down a server). The exception is designed to allow us to clean up code before the program ultimately exits, so we generally don't need to handle it explicitly (because cleanup code happens inside a `finally` clause).

If we do handle it, we would normally reraise the exception, since catching it would stop the program from exiting. There are, of course, situations where we might want to stop the program exiting, for example, if there are unsaved changes and we want to prompt the user if they really want to exit. Usually, if we handle `SystemExit` at all, it's because we want to do something special with it, or are anticipating it directly. We especially don't want it to be accidentally caught in generic clauses that catch all normal exceptions. This is why it derives directly from `BaseException`.

The `KeyboardInterrupt` exception is common in command-line programs. It is thrown when the user explicitly interrupts program execution with an OS-dependent key combination (normally, *Ctrl* + *C*). This is a standard way for the user to deliberately interrupt a running program, and like `SystemExit`, it should almost always respond by terminating the program. Also, like `SystemExit`, it should handle any cleanup tasks inside `finally` blocks.

Here is a class diagram that fully illustrates the exception hierarchy:



When we use the `except:` clause without specifying any type of exception, it will catch all subclasses of `BaseException`; which is to say, it will catch all exceptions, including the two special ones. Since we almost always want these to get special treatment, it is unwise to use the `except:` statement without arguments. If you want to catch all exceptions other than `SystemExit` and `KeyboardInterrupt`, explicitly catch `Exception`.

Furthermore, if you do want to catch all exceptions, I suggest using the syntax `except BaseException:` instead of a raw `except:`. This helps explicitly tell future readers of your code that you are intentionally handling the special case exceptions.

Defining our own exceptions

Often, when we want to raise an exception, we find that none of the built-in exceptions are suitable. Luckily, it's trivial to define new exceptions of our own. The name of the class is usually designed to communicate what went wrong, and we can provide arbitrary arguments in the initializer to include additional information.

All we have to do is inherit from the `Exception` class. We don't even have to add any content to the class! We can, of course, extend `BaseException` directly, but then it will not be caught by generic `except Exception` clauses.

Here's a simple exception we might use in a banking application:

```
class InvalidWithdrawal(Exception):
    pass

raise InvalidWithdrawal("You don't have $50 in your account")
```

The last line illustrates how to raise the newly defined exception. We are able to pass an arbitrary number of arguments into the exception. Often a string message is used, but any object that might be useful in a later exception handler can be stored. The `Exception.__init__` method is designed to accept any arguments and store them as a tuple in an attribute named `args`. This makes exceptions easier to define without needing to override `__init__`.

Of course, if we do want to customize the initializer, we are free to do so. Here's an exception whose initializer accepts the current balance and the amount the user wanted to withdraw. In addition, it adds a method to calculate how overdrawn the request was:

```
class InvalidWithdrawal(Exception):
    def __init__(self, balance, amount):
        super().__init__("account doesn't have ${}".format(
            amount))
        self.amount = amount
        self.balance = balance

    def overage(self):
        return self.amount - self.balance

raise InvalidWithdrawal(25, 50)
```

The `raise` statement at the end illustrates how to construct this exception. As you can see, we can do anything with an exception that we would do with other objects. We could catch an exception and pass it around as a working object, although it is more common to include a reference to the working object as an attribute on an exception and pass that around instead.

Here's how we would handle an `InvalidWithdrawal` exception if one was raised:

```
try:
    raise InvalidWithdrawal(25, 50)
except InvalidWithdrawal as e:
    print("I'm sorry, but your withdrawal is "
            "more than your balance by "
            "${}".format(e.overage()))
```

Here we see a valid use of the `as` keyword. By convention, most Python coders name the exception variable `e`, although, as usual, you are free to call it `ex`, `exception`, or `aunt_sally` if you prefer.

There are many reasons for defining our own exceptions. It is often useful to add information to the exception or log it in some way. But the utility of custom exceptions truly comes to light when creating a framework, library, or API that is intended for access by other programmers. In that case, be careful to ensure your code is raising exceptions that make sense to the client programmer. They should be easy to handle and clearly describe what went on. The client programmer should easily see how to fix the error (if it reflects a bug in their code) or handle the exception (if it's a situation they need to be made aware of).

Exceptions aren't exceptional. Novice programmers tend to think of exceptions as only useful for exceptional circumstances. However, the definition of exceptional circumstances can be vague and subject to interpretation. Consider the following two functions:

```
def divide_with_exception(number, divisor):
    try:
        print("{} / {} = {}".format(
            number, divisor, number / divisor * 1.0))
    except ZeroDivisionError:
        print("You can't divide by zero")

def divide_with_if(number, divisor):
    if divisor == 0:
        print("You can't divide by zero")
    else:
        print("{} / {} = {}".format(
            number, divisor, number / divisor * 1.0))
```

These two functions behave identically. If `divisor` is zero, an error message is printed; otherwise, a message printing the result of division is displayed. We could avoid a `ZeroDivisionError` ever being thrown by testing for it with an `if` statement. Similarly, we can avoid an `IndexError` by explicitly checking whether or not the parameter is within the confines of the list, and a `KeyError` by checking if the key is in a dictionary.

But we shouldn't do this. For one thing, we might write an `if` statement that checks whether or not the index is lower than the parameters of the list, but forget to check negative values.

Remember, Python lists support negative indexing; `-1` refers to the last element in the list.

Eventually, we would discover this and have to find all the places where we were checking code. But if we had simply caught the `IndexError` and handled it, our code would just work.

Python programmers tend to follow a model of *Ask forgiveness rather than permission*, which is to say, they execute code and then deal with anything that goes wrong. The alternative, to *look before you leap*, is generally frowned upon. There are a few reasons for this, but the main one is that it shouldn't be necessary to burn CPU cycles looking for an unusual situation that is not going to arise in the normal path through the code. Therefore, it is wise to use exceptions for exceptional circumstances, even if those circumstances are only a little bit exceptional. Taking this argument further, we can actually see that the exception syntax is also effective for flow control. Like an `if` statement, exceptions can be used for decision making, branching, and message passing.

Imagine an inventory application for a company that sells widgets and gadgets. When a customer makes a purchase, the item can either be available, in which case the item is removed from inventory and the number of items left is returned, or it might be out of stock. Now, being out of stock is a perfectly normal thing to happen in an inventory application. It is certainly not an exceptional circumstance. But what do we return if it's out of stock? A string saying out of stock? A negative number? In both cases, the calling method would have to check whether the return value is a positive integer or something else, to determine if it is out of stock. That seems a bit messy. Instead, we can raise `OutOfStockException` and use the `try` statement to direct program flow control. Make sense? In addition, we want to make sure we don't sell the same item to two different customers, or sell an item that isn't in stock yet. One way to facilitate this is to lock each type of item to ensure only one person can update it at a time. The user must lock the item, manipulate the item (purchase, add stock, count items left…), and then unlock the item. Here's an incomplete `Inventory` example with docstrings that describes what some of the methods should do:

```
class Inventory:
    def lock(self, item_type):
        '''Select the type of item that is going to
        be manipulated. This method will lock the
        item so nobody else can manipulate the
        inventory until it's returned. This prevents
        selling the same item to two different
        customers.'''
        pass

    def unlock(self, item_type):
        '''Release the given type so that other
        customers can access it.'''
        pass

    def purchase(self, item_type):
```

```
        '''If the item is not locked, raise an
        exception. If the item_type  does not exist,
        raise an exception. If the item is currently
        out of stock, raise an exception. If the item
        is available, subtract one item and return
        the number of items left.'''
        pass
```

We could hand this object prototype to a developer and have them implement the methods to do exactly as they say while we work on the code that needs to make a purchase. We'll use Python's robust exception handling to consider different branches, depending on how the purchase was made:

```
item_type = 'widget'
inv = Inventory()
inv.lock(item_type)
try:
    num_left = inv.purchase(item_type)
except InvalidItemType:
    print("Sorry, we don't sell {}".format(item_type))
except OutOfStock:
    print("Sorry, that item is out of stock.")
else:
    print("Purchase complete. There are "
            "{} {}s left".format(num_left, item_type))
finally:
    inv.unlock(item_type)
```

Pay attention to how all the possible exception handling clauses are used to ensure the correct actions happen at the correct time. Even though `OutOfStock` is not a terribly exceptional circumstance, we are able to use an exception to handle it suitably. This same code could be written with an `if...elif...else` structure, but it wouldn't be as easy to read or maintain.

We can also use exceptions to pass messages between different methods. For example, if we wanted to inform the customer as to what date the item is expected to be in stock again, we could ensure our `OutOfStock` object requires a `back_in_stock` parameter when it is constructed. Then, when we handle the exception, we can check that value and provide additional information to the customer. The information attached to the object can be easily passed between two different parts of the program. The exception could even provide a method that instructs the inventory object to reorder or backorder an item.

Using exceptions for flow control can make for some handy program designs. The important thing to take from this discussion is that exceptions are not a bad thing that we should try to avoid. Having an exception occur does not mean that you should have prevented this exceptional circumstance from happening. Rather, it is just a powerful way to communicate information between two sections of code that may not be directly calling each other.

Case study

We've been looking at the use and handling of exceptions at a fairly low level of detail—syntax and definitions. This case study will help tie it all in with our previous chapters so we can see how exceptions are used in the larger context of objects, inheritance, and modules.

Today, we'll be designing a simple central authentication and authorization system. The entire system will be placed in one module, and other code will be able to query that module object for authentication and authorization purposes. We should admit, from the start, that we aren't security experts, and that the system we are designing may be full of security holes. Our purpose is to study exceptions, not to secure a system. It will be sufficient, however, for a basic login and permission system that other code can interact with. Later, if that other code needs to be made more secure, we can have a security or cryptography expert review or rewrite our module, preferably without changing the API.

Authentication is the process of ensuring a user is really the person they say they are. We'll follow the lead of common web systems today, which use a username and private password combination. Other methods of authentication include voice recognition, fingerprint or retinal scanners, and identification cards.

Authorization, on the other hand, is all about determining whether a given (authenticated) user is permitted to perform a specific action. We'll create a basic permission list system that stores a list of the specific people allowed to perform each action.

In addition, we'll add some administrative features to allow new users to be added to the system. For brevity, we'll leave out editing of passwords or changing of permissions once they've been added, but these (highly necessary) features can certainly be added in the future.

There's a simple analysis; now let's proceed with design. We're obviously going to need a `User` class that stores the username and an encrypted password. This class will also allow a user to log in by checking whether a supplied password is valid. We probably won't need a `Permission` class, as those can just be strings mapped to a list of users using a dictionary. We should have a central `Authenticator` class that handles user management and logging in or out. The last piece of the puzzle is an `Authorizor` class that deals with permissions and checking whether a user can perform an activity. We'll provide a single instance of each of these classes in the `auth` module so that other modules can use this central mechanism for all their authentication and authorization needs. Of course, if they want to instantiate private instances of these classes, for non-central authorization activities, they are free to do so.

We'll also be defining several exceptions as we go along. We'll start with a special `AuthException` base class that accepts a `username` and optional `user` object as parameters; most of our self-defined exceptions will inherit from this one.

Let's build the `User` class first; it seems simple enough. A new user can be initialized with a username and password. The password will be stored encrypted to reduce the chances of its being stolen. We'll also need a `check_password` method to test whether a supplied password is the correct one. Here is the class in full:

```
import hashlib

class User:
    def __init__(self, username, password):
        '''Create a new user object. The password
        will be encrypted before storing.'''
        self.username = username
        self.password = self._encrypt_pw(password)
        self.is_logged_in = False

    def _encrypt_pw(self, password):
        '''Encrypt the password with the username and return
        the sha digest.'''
        hash_string = (self.username + password)
        hash_string = hash_string.encode("utf8")
        return hashlib.sha256(hash_string).hexdigest()

    def check_password(self, password):
        '''Return True if the password is valid for this
        user, false otherwise.'''
        encrypted = self._encrypt_pw(password)
        return encrypted == self.password
```

Since the code for encrypting a password is required in both `__init__` and `check_password`, we pull it out to its own method. This way, it only needs to be changed in one place if someone realizes it is insecure and needs improvement. This class could easily be extended to include mandatory or optional personal details, such as names, contact information, and birth dates.

Before we write code to add users (which will happen in the as-yet undefined `Authenticator` class), we should examine some use cases. If all goes well, we can add a user with a username and password; the `User` object is created and inserted into a dictionary. But in what ways can all not go well? Well, clearly we don't want to add a user with a username that already exists in the dictionary. If we did so, we'd overwrite an existing user's data and the new user might have access to that user's privileges. So, we'll need a `UsernameAlreadyExists` exception. Also, for security's sake, we should probably raise an exception if the password is too short. Both of these exceptions will extend `AuthException`, which we mentioned earlier. So, before writing the `Authenticator` class, let's define these three exception classes:

```
class AuthException(Exception):
    def __init__(self, username, user=None):
        super().__init__(username, user)
        self.username = username
        self.user = user

class UsernameAlreadyExists(AuthException):
    pass

class PasswordTooShort(AuthException):
    pass
```

The `AuthException` requires a username and has an optional user parameter. This second parameter should be an instance of the `User` class associated with that username. The two specific exceptions we're defining simply need to inform the calling class of an exceptional circumstance, so we don't need to add any extra methods to them.

Now let's start on the `Authenticator` class. It can simply be a mapping of usernames to user objects, so we'll start with a dictionary in the initialization function. The method for adding a user needs to check the two conditions (password length and previously existing users) before creating a new `User` instance and adding it to the dictionary:

```
class Authenticator:
    def __init__(self):
        '''Construct an authenticator to manage
```

```
            users logging in and out.'''
            self.users = {}

        def add_user(self, username, password):
            if username in self.users:
                raise UsernameAlreadyExists(username)
            if len(password) < 6:
                raise PasswordTooShort(username)
            self.users[username] = User(username, password)
```

We could, of course, extend the password validation to raise exceptions for passwords that are too easy to crack in other ways, if we desired. Now let's prepare the `login` method. If we weren't thinking about exceptions just now, we might just want the method to return `True` or `False`, depending on whether the login was successful or not. But we are thinking about exceptions, and this could be a good place to use them for a not-so-exceptional circumstance. We could raise different exceptions, for example, if the username does not exist or the password does not match. This will allow anyone trying to log a user in to elegantly handle the situation using a `try/except/else` clause. So, first we add these new exceptions:

```
class InvalidUsername(AuthException):
    pass

class InvalidPassword(AuthException):
    pass
```

Then we can define a simple `login` method to our `Authenticator` class that raises these exceptions if necessary. If not, it flags the `user` as logged in and returns:

```
        def login(self, username, password):
            try:
                user = self.users[username]
            except KeyError:
                raise InvalidUsername(username)

            if not user.check_password(password):
                raise InvalidPassword(username, user)

            user.is_logged_in = True
            return True
```

Notice how the `KeyError` is handled. This could have been handled using `if username not in self.users:` instead, but we chose to handle the exception directly. We end up eating up this first exception and raising a brand new one of our own that better suits the user-facing API.

We can also add a method to check whether a particular username is logged in. Deciding whether to use an exception here is trickier. Should we raise an exception if the username does not exist? Should we raise an exception if the user is not logged in?

To answer these questions, we need to think about how the method would be accessed. Most often, this method will be used to answer the yes/no question, "Should I allow them access to <*something*>?" The answer will either be, "Yes, the username is valid and they are logged in", or "No, the username is not valid or they are not logged in". Therefore, a Boolean return value is sufficient. There is no need to use exceptions here, just for the sake of using an exception.

```
        def is_logged_in(self, username):
            if username in self.users:
                return self.users[username].is_logged_in
            return False
```

Finally, we can add a default authenticator instance to our module so that the client code can access it easily using `auth.authenticator`:

```
    authenticator = Authenticator()
```

This line goes at the module level, outside any class definition, so the authenticator variable can be accessed as `auth.authenticator`. Now we can start on the `Authorizor` class, which maps permissions to users. The `Authorizor` class should not permit user access to a permission if they are not logged in, so they'll need a reference to a specific authenticator. We'll also need to set up the permission dictionary upon initialization:

```
    class Authorizor:
        def __init__(self, authenticator):
            self.authenticator = authenticator
            self.permissions = {}
```

Now we can write methods to add new permissions and to set up which users are associated with each permission:

```
        def add_permission(self, perm_name):
            '''Create a new permission that users
            can be added to'''
            try:
                perm_set = self.permissions[perm_name]
            except KeyError:
                self.permissions[perm_name] = set()
            else:
```

```
            raise PermissionError("Permission Exists")

    def permit_user(self, perm_name, username):
        '''Grant the given permission to the user'''
        try:
            perm_set = self.permissions[perm_name]
        except KeyError:
            raise PermissionError("Permission does not exist")
        else:
            if username not in self.authenticator.users:
                raise InvalidUsername(username)
            perm_set.add(username)
```

The first method allows us to create a new permission, unless it already exists, in which case an exception is raised. The second allows us to add a username to a permission, unless either the permission or the username doesn't yet exist.

We use `set` instead of `list` for usernames, so that even if you grant a user permission more than once, the nature of sets means the user is only in the set once. We'll discuss sets further in a later chapter.

A `PermissionError` is raised in both methods. This new error doesn't require a username, so we'll make it extend `Exception` directly, instead of our custom `AuthException`:

```
class PermissionError(Exception):
    pass
```

Finally, we can add a method to check whether a user has a specific `permission` or not. In order for them to be granted access, they have to be both logged into the authenticator and in the set of people who have been granted access to that privilege. If either of these conditions is unsatisfied, an exception is raised:

```
    def check_permission(self, perm_name, username):
        if not self.authenticator.is_logged_in(username):
            raise NotLoggedInError(username)
        try:
            perm_set = self.permissions[perm_name]
        except KeyError:
            raise PermissionError("Permission does not exist")
        else:
            if username not in perm_set:
                raise NotPermittedError(username)
            else:
                return True
```

There are two new exceptions in here; they both take usernames, so we'll define them as subclasses of `AuthException`:

```
class NotLoggedInError(AuthException):
    pass

class NotPermittedError(AuthException):
    pass
```

Finally, we can add a default `authorizor` to go with our default authenticator:

```
authorizor = Authorizor(authenticator)
```

That completes a basic authentication/authorization system. We can test the system at the Python prompt, checking to see whether a user, `joe`, is permitted to do tasks in the paint department:

```
>>> import auth
>>> auth.authenticator.add_user("joe", "joepassword")
>>> auth.authorizor.add_permission("paint")
>>> auth.authorizor.check_permission("paint", "joe")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 109, in check_permission
    raise NotLoggedInError(username)
auth.NotLoggedInError: joe
>>> auth.authenticator.is_logged_in("joe")
False
>>> auth.authenticator.login("joe", "joepassword")
True
>>> auth.authorizor.check_permission("paint", "joe")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 116, in check_permission
    raise NotPermittedError(username)
auth.NotPermittedError: joe
>>> auth.authorizor.check_permission("mix", "joe")
Traceback (most recent call last):
  File "auth.py", line 111, in check_permission
    perm_set = self.permissions[perm_name]
```

```
KeyError: 'mix'

During handling of the above exception, another exception occurred:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 113, in check_permission
    raise PermissionError("Permission does not exist")
auth.PermissionError: Permission does not exist
>>> auth.authorizor.permit_user("mix", "joe")
Traceback (most recent call last):
  File "auth.py", line 99, in permit_user
    perm_set = self.permissions[perm_name]
KeyError: 'mix'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 101, in permit_user
    raise PermissionError("Permission does not exist")
auth.PermissionError: Permission does not exist
>>> auth.authorizor.permit_user("paint", "joe")
>>> auth.authorizor.check_permission("paint", "joe")
True
```

While verbose, the preceding output shows all of our code and most of our exceptions in action, but to really understand the API we've defined, we should write some exception handling code that actually uses it. Here's a basic menu interface that allows certain users to change or test a program:

```
import auth

# Set up a test user and permission
auth.authenticator.add_user("joe", "joepassword")
auth.authorizor.add_permission("test program")
auth.authorizor.add_permission("change program")
auth.authorizor.permit_user("test program", "joe")

class Editor:
```

```
    def __init__(self):
        self.username = None
        self.menu_map = {
                "login": self.login,
                "test": self.test,
                "change": self.change,
                "quit": self.quit
           }

    def login(self):
        logged_in = False
        while not logged_in:
            username = input("username: ")
            password = input("password: ")
            try:
                logged_in = auth.authenticator.login(
                        username, password)
            except auth.InvalidUsername:
                print("Sorry, that username does not exist")
            except auth.InvalidPassword:
                print("Sorry, incorrect password")
            else:
                self.username = username
    def is_permitted(self, permission):
        try:
            auth.authorizor.check_permission(
                permission, self.username)
        except auth.NotLoggedInError as e:
            print("{} is not logged in".format(e.username))
            return False
        except auth.NotPermittedError as e:
            print("{} cannot {}".format(
                e.username, permission))
            return False
        else:
            return True

    def test(self):
        if self.is_permitted("test program"):
            print("Testing program now...")

    def change(self):
```

```
            if self.is_permitted("change program"):
                print("Changing program now...")

        def quit(self):
            raise SystemExit()

        def menu(self):
            try:
                answer = ""
                while True:
                    print("""
    Please enter a command:
    \tlogin\tLogin
    \ttest\tTest the program
    \tchange\tChange the program
    \tquit\tQuit
    """)
                    answer = input("enter a command: ").lower()
                    try:
                        func = self.menu_map[answer]
                    except KeyError:
                        print("{} is not a valid option".format(
                            answer))
                    else:
                        func()
            finally:
                print("Thank you for testing the auth module")

    Editor().menu()
```

This rather long example is conceptually very simple. The `is_permitted` method is probably the most interesting; this is a mostly internal method that is called by both `test` and `change` to ensure the user is permitted access before continuing. Of course, those two methods are stubs, but we aren't writing an editor here; we're illustrating the use of exceptions and exception handlers by testing an authentication and authorization framework

Your Coding Challenge

If you've never dealt with exceptions before, the first thing you need to do is look at any old Python code you've written and notice if there are places you should have been handling exceptions. How would you handle them? Do you need to handle them at all? Sometimes, letting the exception propagate to the console is the best way to communicate to the user, especially if the user is also the script's coder. Sometimes, you can recover from the error and allow the program to continue. Sometimes, you can only reformat the error into something the user can understand and display it to them.

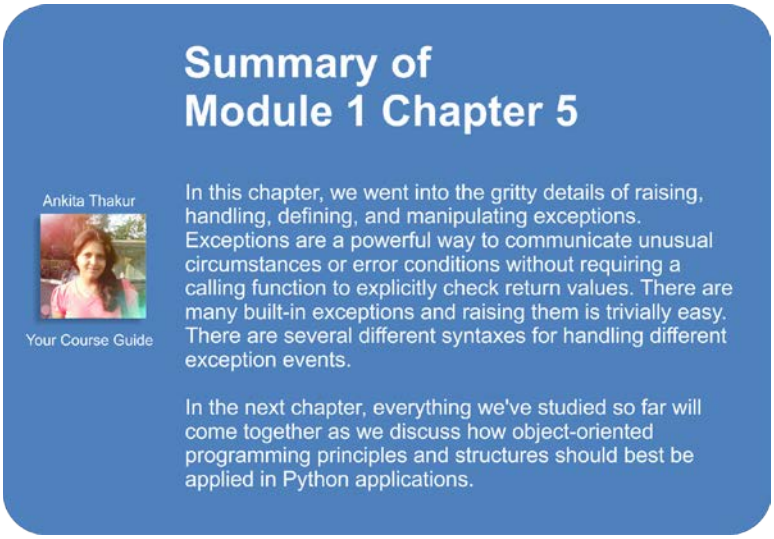
Some common places to look are file I/O (is it possible your code will try to read a file that doesn't exist?), mathematical expressions (is it possible that a value you are dividing by is zero?), list indices (is the list empty?), and dictionaries (does the key exist?). Ask yourself if you should ignore the problem, handle it by checking values first, or handle it with an exception. Pay special attention to areas where you might have used finally and else to ensure the correct code is executed under all conditions.

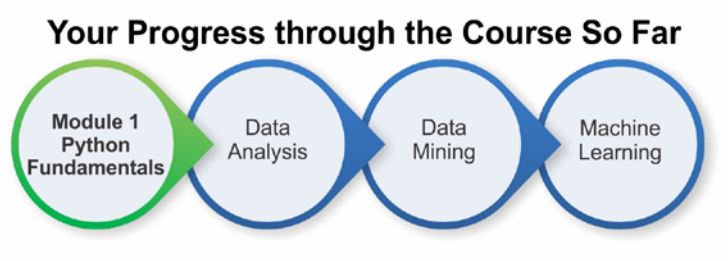
Ankita Thakur

Your Course Guide

Now write some new code. Think of a program that requires authentication and authorization, and try writing some code that uses the auth module we built in the case study. Feel free to modify the module if it's not flexible enough. Try to handle all the exceptions in a sensible way. If you're having trouble coming up with something that requires authentication, try adding authorization to the notepad example from Chapter 3, Objects in Python, or add authorization to the auth module itself—it's not a terribly useful module if just anybody can start adding permissions! Maybe require an administrator username and password before allowing privileges to be added or changed.

Finally, try to think of places in your code where you can raise exceptions. It can be in code you've written or are working on; or you can write a new project as an exercise. You'll probably have the best luck for designing a small framework or API that is meant to be used by other people; exceptions are a terrific communication tool between your code and someone else's. Remember to design and document any self-raised exceptions as part of the API, or they won't know whether or how to handle them!


Summary of
Module 1 Chapter 5
Ankita Thakur
Your Course Guide
In this chapter, we went into the gritty details of raising, handling, defining, and manipulating exceptions. Exceptions are a powerful way to communicate unusual circumstances or error conditions without requiring a calling function to explicitly check return values. There are many built-in exceptions and raising them is trivially easy. There are several different syntaxes for handling different exception events.
In the next chapter, everything we've studied so far will come together as we discuss how object-oriented programming principles and structures should best be applied in Python applications.


Your Progress through the Course So Far
Module 1 Python Fundamentals
Data Analysis
Data Mining
Machine Learning

6
When to Use Object-oriented Programming

In previous chapters, we've covered many of the defining features of object-oriented programming. We now know the principles and paradigms of object-oriented design, and we've covered the syntax of object-oriented programming in Python.

Yet, we don't know exactly how and when to utilize these principles and syntax in practice. In this chapter, we'll discuss some useful applications of the knowledge we've gained, picking up some new topics along the way:

- How to recognize objects
- Data and behaviors, once again
- Wrapping data in behavior using properties
- Restricting data using behavior
- The Don't Repeat Yourself principle
- Recognizing repeated code

Treat objects as objects

This may seem obvious; you should generally give separate objects in your problem domain a special class in your code. We've seen examples of this in the case studies in previous chapters; first, we identify objects in the problem and then model their data and behaviors.

Identifying objects is a very important task in object-oriented analysis and programming. But it isn't always as easy as counting the nouns in a short paragraph, as we've been doing. Remember, objects are things that have both data and behavior. If we are working only with data, we are often better off storing it in a list, set, dictionary, or some other Python data structure (which we'll be covering thoroughly in *Chapter 6*, *Python Data Structures*). On the other hand, if we are working only with behavior, but no stored data, a simple function is more suitable.

An object, however, has both data and behavior. Proficient Python programmers use built-in data structures unless (or until) there is an obvious need to define a class. There is no reason to add an extra level of abstraction if it doesn't help organize our code. On the other hand, the "obvious" need is not always self-evident.

We can often start our Python programs by storing data in a few variables. As the program expands, we will later find that we are passing the same set of related variables to a set of functions. This is the time to think about grouping both variables and functions into a class. If we are designing a program to model polygons in two-dimensional space, we might start with each polygon being represented as a list of points. The points would be modeled as two-tuples (*x*, *y*) describing where that point is located. This is all data, stored in a set of nested data structures (specifically, a list of tuples):

```
square = [(1,1), (1,2), (2,2), (2,1)]
```

Now, if we want to calculate the distance around the perimeter of the polygon, we simply need to sum the distances between the two points. To do this, we also need a function to calculate the distance between two points. Here are two such functions:

```
import math

def distance(p1, p2):
    return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)

def perimeter(polygon):
    perimeter = 0
    points = polygon + [polygon[0]]
    for i in range(len(polygon)):
        perimeter += distance(points[i], points[i+1])
    return perimeter
```

Now, as object-oriented programmers, we clearly recognize that a `polygon` class could encapsulate the list of points (data) and the `perimeter` function (behavior). Further, a `point` class, such as we defined in *Chapter 2*, *Objects in Python*, might encapsulate the `x` and `y` coordinates and the `distance` method. The question is: is it valuable to do this?

For the previous code, maybe yes, maybe no. With our recent experience in object-oriented principles, we can write an object-oriented version in record time. Let's compare them

```
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, p2):
        return math.sqrt((self.x-p2.x)**2 + (self.y-p2.y)**2)

class Polygon:
    def __init__(self):
        self.vertices = []

    def add_point(self, point):
        self.vertices.append((point))

    def perimeter(self):
        perimeter = 0
        points = self.vertices + [self.vertices[0]]
        for i in range(len(self.vertices)):
            perimeter += points[i].distance(points[i+1])
        return perimeter
```

As we can see from the highlighted sections, there is twice as much code here as there was in our earlier version, although we could argue that the `add_point` method is not strictly necessary.

Now, to understand the differences a little better, let's compare the two APIs in use. Here's how to calculate the perimeter of a square using the object-oriented code:

```
>>> square = Polygon()
>>> square.add_point(Point(1,1))
>>> square.add_point(Point(1,2))
>>> square.add_point(Point(2,2))
>>> square.add_point(Point(2,1))
>>> square.perimeter()
4.0
```

That's fairly succinct and easy to read, you might think, but let's compare it to the function-based code:

```
>>> square = [(1,1), (1,2), (2,2), (2,1)]
>>> perimeter(square)
4.0
```

Hmm, maybe the object-oriented API isn't so compact! That said, I'd argue that it was easier to *read* than the functional example: How do we know what the list of tuples is supposed to represent in the second version? How do we remember what kind of object (a list of two-tuples? That's not intuitive!) we're supposed to pass into the `perimeter` function? We would need a lot of documentation to explain how these functions should be used.

In contrast, the object-oriented code is relatively self-documenting, we just have to look at the list of methods and their parameters to know what the object does and how to use it. By the time we wrote all the documentation for the functional version, it would probably be longer than the object-oriented code.

Finally, code length is not a good indicator of code complexity. Some programmers get hung up on complicated "one liners" that do an incredible amount of work in one line of code. This can be a fun exercise, but the result is often unreadable, even to the original author the following day. Minimizing the amount of code can often make a program easier to read, but do not blindly assume this is the case.

Luckily, this trade-off isn't necessary. We can make the object-oriented `Polygon` API as easy to use as the functional implementation. All we have to do is alter our `Polygon` class so that it can be constructed with multiple points. Let's give it an initializer that accepts a list of `Point` objects. In fact, let's allow it to accept tuples too, and we can construct the `Point` objects ourselves, if needed:

```
def __init__(self, points=None):
    points = points if points else []
    self.vertices = []
    for point in points:
        if isinstance(point, tuple):
            point = Point(*point)
        self.vertices.append(point)
```

This initializer goes through the list and ensures that any tuples are converted to points. If the object is not a tuple, we leave it as is, assuming that it is either a `Point` object already, or an unknown duck-typed object that can act like a `Point` object.

Still, there's no clear winner between the object-oriented and more data-oriented versions of this code. They both do the same thing. If we have new functions that accept a polygon argument, such as `area(polygon)` or `point_in_polygon(polygon, x, y)`, the benefits of the object-oriented code become increasingly obvious. Likewise, if we add other attributes to the polygon, such as `color` or `texture`, it makes more and more sense to encapsulate that data into a single class.

The distinction is a design decision, but in general, the more complicated a set of data is, the more likely it is to have multiple functions specific to that data, and the more useful it is to use a class with attributes and methods instead.

When making this decision, it also pays to consider how the class will be used. If we're only trying to calculate the perimeter of one polygon in the context of a much greater problem, using a function will probably be quickest to code and easier to use "one time only". On the other hand, if our program needs to manipulate numerous polygons in a wide variety of ways (calculate perimeter, area, intersection with other polygons, move or scale them, and so on), we have most certainly identified an object; one that needs to be extremely versatile.

Additionally, pay attention to the interaction between objects. Look for inheritance relationships; inheritance is impossible to model elegantly without classes, so make sure to use them. Look for the other types of relationships we discussed in *Chapter 1*, *Object-oriented Design*, association and composition. Composition can, technically, be modeled using only data structures; for example, we can have a list of dictionaries holding tuple values, but it is often less complicated to create a few classes of objects, especially if there is behavior associated with the data.

Don't rush to use an object just because you can use an object, but *never* neglect to create a class when you need to use a class.

Adding behavior to class data with properties

Throughout this module, we've been focusing on the separation of behavior and data. This is very important in object-oriented programming, but we're about to see that, in Python, the distinction can be uncannily blurry. Python is very good at blurring distinctions; it doesn't exactly help us to "think outside the box". Rather, it teaches us to stop thinking about the box.

Before we get into the details, let's discuss some bad object-oriented theory. Many object-oriented languages (Java is the most notorious) teach us to never access attributes directly. They insist that we write attribute access like this:

```
class Color:
    def __init__(self, rgb_value, name):
        self._rgb_value = rgb_value
        self._name = name

    def set_name(self, name):
        self._name = name

    def get_name(self):
        return self._name
```

The variables are prefixed with an underscore to suggest that they are private (other languages would actually force them to be private). Then the get and set methods provide access to each variable. This class would be used in practice as follows:

```
>>> c = Color("#ff0000", "bright red")
>>> c.get_name()
'bright red'
>>> c.set_name("red")
>>> c.get_name()
'red'
```

This is not nearly as readable as the direct access version that Python favors:

```
class Color:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self.name = name

c = Color("#ff0000", "bright red")
print(c.name)
c.name = "red"
```

So why would anyone insist upon the method-based syntax? Their reasoning is that someday we may want to add extra code when a value is set or retrieved. For example, we could decide to cache a value and return the cached value, or we might want to validate that the value is a suitable input.

In code, we could decide to change the `set_name()` method as follows:

```
def set_name(self, name):
    if not name:
        raise Exception("Invalid Name")
    self._name = name
```

Now, in Java and similar languages, if we had written our original code to do direct attribute access, and then later changed it to a method like the preceding one, we'd have a problem: anyone who had written code that accessed the attribute directly would now have to access the method. If they don't change the access style from attribute access to a function call, their code will be broken. The mantra in these languages is that we should never make public members private. This doesn't make much sense in Python since there isn't any real concept of private members!

Python gives us the `property` keyword to make methods look like attributes. We can therefore write our code to use direct member access, and if we unexpectedly need to alter the implementation to do some calculation when getting or setting that attribute's value, we can do so without changing the interface. Let's see how it looks:

```
class Color:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self._name = name

    def _set_name(self, name):
        if not name:
            raise Exception("Invalid Name")
        self._name = name

    def _get_name(self):
        return self._name

    name = property(_get_name, _set_name)
```

If we had started with the earlier non-method-based class, which set the `name` attribute directly, we could later change the code to look like the preceding one. We first change the `name` attribute into a (semi-) private `_name` attribute. Then we add two more (semi-) private methods to get and set that variable, doing our validation when we set it.

Finally, we have the `property` declaration at the bottom. This is the magic. It creates a new attribute on the `Color` class called `name`, which now replaces the previous `name` attribute. It sets this attribute to be a property, which calls the two methods we just created whenever the property is accessed or changed. This new version of the `Color` class can be used exactly the same way as the previous version, yet it now does validation when we set the `name` attribute:

```
>>> c = Color("#0000ff", "bright red")
>>> print(c.name)
bright red
>>> c.name = "red"
>>> print(c.name)
red
>>> c.name = ""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "setting_name_property.py", line 8, in _set_name
    raise Exception("Invalid Name")
Exception: Invalid Name
```

So, if we'd previously written code to access the `name` attribute, and then changed it to use our `property` object, the previous code would still work, unless it was sending an empty `property` value, which is the behavior we wanted to forbid in the first place. Success!

Bear in mind that even with the `name` property, the previous code is not 100 percent safe. People can still access the `_name` attribute directly and set it to an empty string if they want to. But if they access a variable we've explicitly marked with an underscore to suggest it is private, they're the ones that have to deal with the consequences, not us.

Properties in detail

Think of the `property` function as returning an object that proxies any requests to set or access the attribute value through the methods we have specified. The `property` keyword is like a constructor for such an object, and that object is set as the public facing member for the given attribute.

This `property` constructor can actually accept two additional arguments, a deletion function and a docstring for the property. The `delete` function is rarely supplied in practice, but it can be useful for logging that a value has been deleted, or possibly to veto deleting if we have reason to do so. The docstring is just a string describing what the property does, no different from the docstrings we discussed in *Chapter 2*, *Objects in Python*. If we do not supply this parameter, the docstring will instead be copied from the docstring for the first argument: the getter method. Here is a silly example that simply states whenever any of the methods are called:

```
class Silly:
    def _get_silly(self):
        print("You are getting silly")
        return self._silly
    def _set_silly(self, value):
        print("You are making silly {}".format(value))
        self._silly = value
    def _del_silly(self):
        print("Whoah, you killed silly!")
        del self._silly

    silly = property(_get_silly, _set_silly,
            _del_silly, "This is a silly property")
```

If we actually use this class, it does indeed print out the correct strings when we ask it to:

```
>>> s = Silly()
>>> s.silly = "funny"
You are making silly funny
>>> s.silly
You are getting silly
'funny'
>>> del s.silly
Whoah, you killed silly!
```

Further, if we look at the help file for the `Silly` class (by issuing `help(silly)` at the interpreter prompt), it shows us the custom docstring for our `silly` attribute:

```
Help on class Silly in module __main__:

class Silly(builtins.object)
```

```
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  silly
|      This is a silly property
```

Once again, everything is working as we planned. In practice, properties are normally only defined with the first two parameters: the getter and setter functions. If we want to supply a docstring for a property, we can define it on the getter function; the property proxy will copy it into its own docstring. The deletion function is often left empty because object attributes are rarely deleted. If a coder does try to delete a property that doesn't have a deletion function specified, it will raise an exception. Therefore, if there is a legitimate reason to delete our property, we should supply that function.

Decorators – another way to create properties

If you've never used Python decorators before, you might want to skip this section and come back to it after we've discussed the decorator pattern in *Chapter 10*, *Python Design Patterns I*. However, you don't need to understand what's going on to use the decorator syntax to make property methods more readable.

The property function can be used with the decorator syntax to turn a get function into a property:

```
class Foo:
    @property
    def foo(self):
        return "bar"
```

This applies the `property` function as a decorator, and is equivalent to the previous `foo = property(foo)` syntax. The main difference, from a readability perspective, is that we get to mark the `foo` function as a property at the top of the method, instead of after it is defined, where it can be easily overlooked. It also means we don't have to create private methods with underscore prefixes just to define a property.

Going one step further, we can specify a setter function for the new property as follows:

```
class Foo:
    @property
    def foo(self):
        return self._foo

    @foo.setter
    def foo(self, value):
        self._foo = value
```

This syntax looks pretty odd, although the intent is obvious. First, we decorate the `foo` method as a getter. Then, we decorate a second method with exactly the same name by applying the `setter` attribute of the originally decorated `foo` method! The `property` function returns an object; this object always comes with its own `setter` attribute, which can then be applied as a decorator to other functions. Using the same name for the get and set methods is not required, but it does help group the multiple methods that access one property together.

We can also specify a deletion function with `@foo.deleter`. We cannot specify a docstring using `property` decorators, so we need to rely on the property copying the docstring from the initial getter method.

Here's our previous `Silly` class rewritten to use `property` as a decorator:

```
class Silly:
    @property
    def silly(self):
        "This is a silly property"
        print("You are getting silly")
        return self._silly

    @silly.setter
    def silly(self, value):
        print("You are making silly {}".format(value))
        self._silly = value

    @silly.deleter
    def silly(self):
        print("Whoah, you killed silly!")
        del self._silly
```

This class operates *exactly* the same as our earlier version, including the help text. You can use whichever syntax you feel is more readable and elegant.

Deciding when to use properties

With the property built-in clouding the division between behavior and data, it can be confusing to know which one to choose. The example use case we saw earlier is one of the most common uses of properties; we have some data on a class that we later want to add behavior to. There are also other factors to take into account when deciding to use a property.

Technically, in Python, data, properties, and methods are all attributes on a class. The fact that a method is callable does not distinguish it from other types of attributes; indeed, we'll see in *Chapter 7*, *Python Object-oriented Shortcuts*, that it is possible to create normal objects that can be called like functions. We'll also discover that functions and methods are themselves normal objects.

The fact that methods are just callable attributes, and properties are just customizable attributes can help us make this decision. Methods should typically represent actions; things that can be done to, or performed by, the object. When you call a method, even with only one argument, it should *do* something. Method names are generally verbs.

Once confirming that an attribute is not an action, we need to decide between standard data attributes and properties. In general, always use a standard attribute until you need to control access to that property in some way. In either case, your attribute is usually a noun. The only difference between an attribute and a property is that we can invoke custom actions automatically when a property is retrieved, set, or deleted.

Let's look at a more realistic example. A common need for custom behavior is caching a value that is difficult to calculate or expensive to look up (requiring, for example, a network request or database query). The goal is to store the value locally to avoid repeated calls to the expensive calculation.

We can do this with a custom getter on the property. The first time the value is retrieved, we perform the lookup or calculation. Then we could locally cache the value as a private attribute on our object (or in dedicated caching software), and the next time the value is requested, we return the stored data. Here's how we might cache a web page:

```
from urllib.request import urlopen

class WebPage:
    def __init__(self, url):
        self.url = url
        self._content = None

    @property
```

```
        def content(self):
            if not self._content:
                print("Retrieving New Page...")
                self._content = urlopen(self.url).read()
            return self._content
```

We can test this code to see that the page is only retrieved once:

```
>>> import time
>>> webpage = WebPage("http://ccphillips.net/")
>>> now = time.time()
>>> content1 = webpage.content
Retrieving New Page...
>>> time.time() - now
22.43316888809204
>>> now = time.time()
>>> content2 = webpage.content
>>> time.time() - now
1.9266459941864014
>>> content2 == content1
True
```

I was on an awful satellite connection when I originally tested this code and it took 20 seconds the first time I loaded the content. The second time, I got the result in 2 seconds (which is really just the amount of time it took to type the lines into the interpreter).

Custom getters are also useful for attributes that need to be calculated on the fly, based on other object attributes. For example, we might want to calculate the average for a list of integers:

```
    class AverageList(list):
        @property
        def average(self):
            return sum(self) / len(self)
```

This very simple class inherits from `list`, so we get list-like behavior for free. We just add a property to the class, and presto, our list can have an average:

```
>>> a = AverageList([1,2,3,4])
>>> a.average
2.5
```

Of course, we could have made this a method instead, but then we should call it `calculate_average()`, since methods represent actions. But a property called `average` is more suitable, both easier to type, and easier to read.

Custom setters are useful for validation, as we've already seen, but they can also be used to proxy a value to another location. For example, we could add a content setter to the `WebPage` class that automatically logs into our web server and uploads a new page whenever the value is set.

Manager objects

We've been focused on objects and their attributes and methods. Now, we'll take a look at designing higher-level objects: the kinds of objects that manage other objects. The objects that tie everything together.

The difference between these objects and most of the examples we've seen so far is that our examples tend to represent concrete ideas. Management objects are more like office managers; they don't do the actual "visible" work out on the floor, but without them, there would be no communication between departments and nobody would know what they are supposed to do (although, this can be true anyway if the organization is badly managed!). Analogously, the attributes on a management class tend to refer to other objects that do the "visible" work; the behaviors on such a class delegate to those other classes at the right time, and pass messages between them.

As an example, we'll write a program that does a find and replace action for text files stored in a compressed ZIP file. We'll need objects to represent the ZIP file and each individual text file (luckily, we don't have to write these classes, they're available in the Python standard library). The manager object will be responsible for ensuring three steps occur in order:

1. Unzipping the compressed file.
2. Performing the find and replace action.
3. Zipping up the new files.

The class is initialized with the `.zip` filename and search and replace strings. We create a temporary directory to store the unzipped files in, so that the folder stays clean. The Python 3.4 `pathlib` library helps out with file and directory manipulation. We'll learn more about that in *Chapter 8*, *Strings and Serialization*, but the interface should be pretty clear in the following example:

```
import sys
import shutil
import zipfile
```

```
from pathlib import Path

class ZipReplace:
    def __init__(self, filename, search_string, replace_string):
        self.filename = filename
        self.search_string = search_string
        self.replace_string = replace_string
        self.temp_directory = Path("unzipped-{}".format(
                filename))
```

Then, we create an overall "manager" method for each of the three steps. This method delegates responsibility to other methods. Obviously, we could do all three steps in one method, or indeed, in one script without ever creating an object. There are several advantages to separating the three steps:

- **Readability**: The code for each step is in a self-contained unit that is easy to read and understand. The method names describe what the method does, and less additional documentation is required to understand what is going on.
- **Extensibility**: If a subclass wanted to use compressed TAR files instead of ZIP files, it could override the `zip` and `unzip` methods without having to duplicate the `find_replace` method.
- **Partitioning**: An external class could create an instance of this class and call the `find_replace` method directly on some folder without having to `zip` the content.

The delegation method is the first in the following code; the rest of the methods are included for completeness:

```
    def zip_find_replace(self):
        self.unzip_files()
        self.find_replace()
        self.zip_files()

    def unzip_files(self):
        self.temp_directory.mkdir()
        with zipfile.ZipFile(self.filename) as zip:
            zip.extractall(str(self.temp_directory))

    def find_replace(self):
        for filename in self.temp_directory.iterdir():
            with filename.open() as file:
                contents = file.read()
            contents = contents.replace(
                    self.search_string, self.replace_string)
```

```
                with filename.open("w") as file:
                    file.write(contents)

        def zip_files(self):
            with zipfile.ZipFile(self.filename, 'w') as file:
                for filename in self.temp_directory.iterdir():
                    file.write(str(filename), filename.name)
            shutil.rmtree(str(self.temp_directory))

    if __name__ == "__main__":
        ZipReplace(*sys.argv[1:4]).zip_find_replace()
```

For brevity, the code for zipping and unzipping files is sparsely documented. Our current focus is on object-oriented design; if you are interested in the inner details of the `zipfile` module, refer to the documentation in the standard library, either online or by typing `import zipfile ; help(zipfile)` into your interactive interpreter. Note that this example only searches the top-level files in a ZIP file; if there are any folders in the unzipped content, they will not be scanned, nor will any files inside those folders.

The last two lines in the example allow us to run the program from the command line by passing the `zip` filename, search string, and replace string as arguments:

```
python zipsearch.py hello.zip hello hi
```

Of course, this object does not have to be created from the command line; it could be imported from another module (to perform batch ZIP file processing) or accessed as part of a GUI interface or even a higher-level management object that knows where to get ZIP files (for example, to retrieve them from an FTP server or back them up to an external disk).

As programs become more and more complex, the objects being modeled become less and less like physical objects. Properties are other abstract objects and methods are actions that change the state of those abstract objects. But at the heart of every object, no matter how complex, is a set of concrete properties and well-defined behaviors.

Removing duplicate code

Often the code in management style classes such as `ZipReplace` is quite generic and can be applied in a variety of ways. It is possible to use either composition or inheritance to help keep this code in one place, thus eliminating duplicate code. Before we look at any examples of this, let's discuss a tiny bit of theory. Specifically, why is duplicate code a bad thing?

There are several reasons, but they all boil down to readability and maintainability. When we're writing a new piece of code that is similar to an earlier piece, the easiest thing to do is copy the old code and change whatever needs to be changed (variable names, logic, comments) to make it work in the new location. Alternatively, if we're writing new code that seems similar, but not identical to code elsewhere in the project, it is often easier to write fresh code with similar behavior, rather than figure out how to extract the overlapping functionality.

But as soon as someone has to read and understand the code and they come across duplicate blocks, they are faced with a dilemma. Code that might have made sense suddenly has to be understood. How is one section different from the other? How are they the same? Under what conditions is one section called? When do we call the other? You might argue that you're the only one reading your code, but if you don't touch that code for eight months it will be as incomprehensible to you as it is to a fresh coder. When we're trying to read two similar pieces of code, we have to understand why they're different, as well as how they're different. This wastes the reader's time; code should always be written to be readable first.

I once had to try to understand someone's code that had three identical copies of the same 300 lines of very poorly written code. I had been working with the code for a month before I finally comprehended that the three "identical" versions were actually performing slightly different tax calculations. Some of the subtle differences were intentional, but there were also obvious areas where someone had updated a calculation in one function without updating the other two. The number of subtle, incomprehensible bugs in the code could not be counted. I eventually replaced all 900 lines with an easy-to-read function of 20 lines or so.

Reading such duplicate code can be tiresome, but code maintenance is even more tormenting. As the preceding story suggests, keeping two similar pieces of code up to date can be a nightmare. We have to remember to update both sections whenever we update one of them, and we have to remember how the multiple sections differ so we can modify our changes when we are editing each of them. If we forget to update both sections, we will end up with extremely annoying bugs that usually manifest themselves as, "but I fixed that already, why is it still happening?"

The result is that people who are reading or maintaining our code have to spend astronomical amounts of time understanding and testing it compared to if we had written the code in a nonrepetitive manner in the first place. It's even more frustrating when we are the ones doing the maintenance; we find ourselves saying, "why didn't I do this right the first time?" The time we save by copy-pasting existing code is lost the very first time we have to maintain it. Code is both read and modified many more times and much more often than it is written. Comprehensible code should always be paramount.

This is why programmers, especially Python programmers (who tend to value elegant code more than average), follow what is known as the **Don't Repeat Yourself** (**DRY**) principle. DRY code is maintainable code. My advice to beginning programmers is to never use the copy and paste feature of their editor. To intermediate programmers, I suggest they think thrice before they hit *Ctrl* + *C*.

But what should we do instead of code duplication? The simplest solution is often to move the code into a function that accepts parameters to account for whatever parts are different. This isn't a terribly object-oriented solution, but it is frequently optimal.

For example, if we have two pieces of code that unzip a ZIP file into two different directories, we can easily write a function that accepts a parameter for the directory to which it should be unzipped instead. This may make the function itself slightly more difficult to read, but a good function name and docstring can easily make up for that, and any code that invokes the function will be easier to read.

That's certainly enough theory! The moral of the story is: always make the effort to refactor your code to be easier to read instead of writing bad code that is only easier to write.

In practice

Let's explore two ways we can reuse existing code. After writing our code to replace strings in a ZIP file full of text files, we are later contracted to scale all the images in a ZIP file to 640 x 480. Looks like we could use a very similar paradigm to what we used in `ZipReplace`. The first impulse might be to save a copy of that file and change the `find_replace` method to `scale_image` or something similar.

But, that's uncool. What if someday we want to change the `unzip` and `zip` methods to also open TAR files? Or maybe we want to use a guaranteed unique directory name for temporary files. In either case, we'd have to change it in two different places!

We'll start by demonstrating an inheritance-based solution to this problem. First we'll modify our original `ZipReplace` class into a superclass for processing generic ZIP files:

```
import os
import shutil
import zipfile
from pathlib import Path


class ZipProcessor:
    def __init__(self, zipname):
        self.zipname = zipname
```

```
            self.temp_directory = Path("unzipped-{}".format(
                    zipname[:-4]))

        def process_zip(self):
            self.unzip_files()
            self.process_files()
            self.zip_files()

        def unzip_files(self):
            self.temp_directory.mkdir()
            with zipfile.ZipFile(self.zipname) as zip:
                zip.extractall(str(self.temp_directory))

        def zip_files(self):
            with zipfile.ZipFile(self.zipname, 'w') as file:
                for filename in self.temp_directory.iterdir():
                    file.write(str(filename), filename.name)
            shutil.rmtree(str(self.temp_directory))
```

We changed the `filename` property to `zipname` to avoid confusion with the `filename` local variables inside the various methods. This helps make the code more readable even though it isn't actually a change in design.

We also dropped the two parameters to `__init__` (`search_string` and `replace_string`) that were specific to `ZipReplace`. Then we renamed the `zip_find_replace` method to `process_zip` and made it call an (as yet undefined) `process_files` method instead of `find_replace`; these name changes help demonstrate the more generalized nature of our new class. Notice that we have removed the `find_replace` method altogether; that code is specific to `ZipReplace` and has no business here.

This new `ZipProcessor` class doesn't actually define a `process_files` method; so if we ran it directly, it would raise an exception. Because it isn't meant to run directly, we removed the main call at the bottom of the original script.

Now, before we move on to our image processing app, let's fix up our original `zipsearch` class to make use of this parent class:

```
from zip_processor import ZipProcessor
import sys
import os

class ZipReplace(ZipProcessor):
    def __init__(self, filename, search_string,
            replace_string):
        super().__init__(filename)
```

```
            self.search_string = search_string
            self.replace_string = replace_string

        def process_files(self):
            '''perform a search and replace on all files in the
            temporary directory'''
            for filename in self.temp_directory.iterdir():
                with filename.open() as file:
                    contents = file.read()
                contents = contents.replace(
                        self.search_string, self.replace_string)
                with filename.open("w") as file:
                    file.write(contents)

    if __name__ == "__main__":
        ZipReplace(*sys.argv[1:4]).process_zip()
```

This code is a bit shorter than the original version, since it inherits its ZIP processing abilities from the parent class. We first import the base class we just wrote and make `ZipReplace` extend that class. Then we use `super()` to initialize the parent class. The `find_replace` method is still here, but we renamed it to `process_files` so the parent class can call it from its management interface. Because this name isn't as descriptive as the old one, we added a docstring to describe what it is doing.

Now, that was quite a bit of work, considering that all we have now is a program that is functionally not different from the one we started with! But having done that work, it is now much easier for us to write other classes that operate on files in a ZIP archive, such as the (hypothetically requested) photo scaler. Further, if we ever want to improve or bug fix the zip functionality, we can do it for all classes by changing only the one `ZipProcessor` base class. Maintenance will be much more effective.

See how simple it is now to create a photo scaling class that takes advantage of the `ZipProcessor` functionality. (Note: this class requires the third-party `pillow` library to get the `PIL` module. You can install it with `pip install pillow`.)

```
from zip_processor import ZipProcessor
import sys
from PIL import Image

class ScaleZip(ZipProcessor):

    def process_files(self):
        '''Scale each image in the directory to 640x480'''
        for filename in self.temp_directory.iterdir():
            im = Image.open(str(filename))
```

```
            scaled = im.resize((640, 480))
            scaled.save(str(filename))

if __name__ == "__main__":
    ScaleZip(*sys.argv[1:4]).process_zip()
```

Look how simple this class is! All that work we did earlier paid off. All we do is open each file (assuming that it is an image; it will unceremoniously crash if a file cannot be opened), scale it, and save it back. The `ZipProcessor` class takes care of the zipping and unzipping without any extra work on our part.

Case study

For this case study, we'll try to delve further into the question, "when should I choose an object versus a built-in type?" We'll be modeling a `Document` class that might be used in a text editor or word processor. What objects, functions, or properties should it have?

We might start with a `str` for the `Document` contents, but in Python, strings aren't mutable (able to be changed). Once a `str` is defined, it is forever. We can't insert a character into it or remove one without creating a brand new string object. That would be leaving a lot of `str` objects taking up memory until Python's garbage collector sees fit to clean up behind us.

So, instead of a string, we'll use a list of characters, which we can modify at will. In addition, a `Document` class would need to know the current cursor position within the list, and should probably also store a filename for the document.

Real text editors use a binary-tree based data structure called a `rope` to model their document contents. This module's title isn't "advanced data structures", so if you're interested in learning more about this fascinating topic, you may want to search the web for the rope data structure.

Now, what methods should it have? There are a lot of things we might want to do to a text document, including inserting, deleting, and selecting characters, cut, copy, paste, the selection, and saving or closing the document. It looks like there are copious amounts of both data and behavior, so it makes sense to put all this stuff into its own `Document` class.

A pertinent question is: should this class be composed of a bunch of basic Python objects such as `str` filenames, `int` cursor positions, and a `list` of characters? Or should some or all of those things be specially defined objects in their own right? What about individual lines and characters, do they need to have classes of their own?

We'll answer these questions as we go, but let's start with the simplest possible `Document` class first and see what it can do:

```
class Document:
    def __init__(self):
        self.characters = []
        self.cursor = 0
        self.filename = ''

    def insert(self, character):
        self.characters.insert(self.cursor, character)
        self.cursor += 1

    def delete(self):
        del self.characters[self.cursor]

    def save(self):
        with open(self.filename, 'w') as f:
            f.write(''.join(self.characters))

    def forward(self):
        self.cursor += 1

    def back(self):
        self.cursor -= 1
```

This simple class allows us full control over editing a basic document. Have a look at it in action:

```
>>> doc = Document()
>>> doc.filename = "test_document"
>>> doc.insert('h')
>>> doc.insert('e')
>>> doc.insert('l')
>>> doc.insert('l')
>>> doc.insert('o')
>>> "".join(doc.characters)
'hello'
>>> doc.back()
>>> doc.delete()
>>> doc.insert('p')
>>> "".join(doc.characters)
'hellp'
```

Looks like it's working. We could connect a keyboard's letter and arrow keys to these methods and the document would track everything just fine.

But what if we want to connect more than just arrow keys. What if we want to connect the *Home* and *End* keys as well? We could add more methods to the `Document` class that search forward or backwards for newline characters (in Python, a newline character, or `\n` represents the end of one line and the beginning of a new one) in the string and jump to them, but if we did that for every possible movement action (move by words, move by sentences, *Page Up*, *Page Down*, end of line, beginning of whitespace, and more), the class would be huge. Maybe it would be better to put those methods on a separate object. So, let us turn the cursor attribute into an object that is aware of its position and can manipulate that position. We can move the forward and back methods to that class, and add a couple more for the *Home* and *End* keys:

```
class Cursor:
    def __init__(self, document):
        self.document = document
        self.position = 0

    def forward(self):
        self.position += 1

    def back(self):
        self.position -= 1

    def home(self):
        while self.document.characters[
                self.position-1] != '\n':
            self.position -= 1
            if self.position == 0:
                # Got to beginning of file before newline
                break

    def end(self):
        while self.position < len(self.document.characters
                ) and self.document.characters[
                    self.position] != '\n':
            self.position += 1
```

This class takes the document as an initialization parameter so the methods have access to the content of the document's character list. It then provides simple methods for moving backwards and forwards, as before, and for moving to the `home` and `end` positions.

This code is not very safe. You can very easily move past the ending position, and if you try to go home on an empty file, it will crash. These examples are kept short to make them readable, but that doesn't mean they are defensive! You can improve the error checking of this code as an exercise; it might be a great opportunity to expand your exception handling skills.

The `Document` class itself is hardly changed, except for removing the two methods that were moved to the `Cursor` class:

```
class Document:
    def __init__(self):
        self.characters = []
        self.cursor = Cursor(self)
        self.filename = ''

       def insert(self, character):
        self.characters.insert(self.cursor.position,
                character)
        self.cursor.forward()

    def delete(self):
        del self.characters[self.cursor.position]

    def save(self):
        f = open(self.filename, 'w')
        f.write(''.join(self.characters))
        f.close()
```

We simply updated anything that accessed the old cursor integer to use the new object instead. We can test that the `home` method is really moving to the newline character:

```
>>> d = Document()
>>> d.insert('h')
>>> d.insert('e')
>>> d.insert('l')
>>> d.insert('l')
>>> d.insert('o')
>>> d.insert('\n')
>>> d.insert('w')
```

```
>>> d.insert('o')
>>> d.insert('r')
>>> d.insert('l')
>>> d.insert('d')
>>> d.cursor.home()
>>> d.insert("*")
>>> print("".join(d.characters))
hello
*world
```

Now, since we've been using that string `join` function a lot (to concatenate the characters so we can see the actual document contents), we can add a property to the `Document` class to give us the complete string:

```
    @property
    def string(self):
        return "".join(self.characters)
```

This makes our testing a little simpler:

```
>>> print(d.string)
hello
world
```

This framework is simple (though it might be a bit time consuming!) to extend to create and edit a complete plaintext document. Now, let's extend it to work for rich text; text that can have **bold**, underlined, or *italic* characters.

There are two ways we could process this; the first is to insert "fake" characters into our character list that act like instructions, such as "bold characters until you find a stop bold character". The second is to add information to each character indicating what formatting it should have. While the former method is probably more common, we'll implement the latter solution. To do that, we're obviously going to need a class for characters. This class will have an attribute representing the character, as well as three Boolean attributes representing whether it is bold, italic, or underlined.

Hmm, wait! Is this `Character` class going to have any methods? If not, maybe we should use one of the many Python data structures instead; a tuple or named tuple would probably be sufficient. Are there any actions that we would want to do to, or invoke on a character?

Well, clearly, we might want to do things with characters, such as delete or copy them, but those are things that need to be handled at the `Document` level, since they are really modifying the list of characters. Are there things that need to be done to individual characters?

Actually, now that we're thinking about what a `Character` class actually is... what is it? Would it be safe to say that a `Character` class is a string? Maybe we should use an inheritance relationship here? Then we can take advantage of the numerous methods that `str` instances come with.

What sorts of methods are we talking about? There's `startswith`, `strip`, `find`, `lower`, and many more. Most of these methods expect to be working on strings that contain more than one character. In contrast, if `Character` were to subclass `str`, we'd probably be wise to override `__init__` to raise an exception if a multi-character string were supplied. Since all those methods we'd get for free wouldn't really apply to our `Character` class, it seems we needn't use inheritance, after all.

This brings us back to our original question; should `Character` even be a class? There is a very important special method on the `object` class that we can take advantage of to represent our characters. This method, called `__str__` (two underscores, like `__init__`), is used in string manipulation functions like `print` and the `str` constructor to convert any class to a string. The default implementation does some boring stuff like printing the name of the module and class and its address in memory. But if we override it, we can make it print whatever we like. For our implementation, we could make it prefix characters with special characters to represent whether they are bold, italic, or underlined. So, we will create a class to represent a character, and here it is:

```
class Character:
    def __init__(self, character,
                 bold=False, italic=False, underline=False):
        assert len(character) == 1
        self.character = character
        self.bold = bold
        self.italic = italic
        self.underline = underline

    def __str__(self):
        bold = "*" if self.bold else ''
        italic = "/" if self.italic else ''
        underline = "_" if self.underline else ''
        return bold + italic + underline + self.character
```

This class allows us to create characters and prefix them with a special character when the `str()` function is applied to them. Nothing too exciting there. We only have to make a few minor modifications to the `Document` and `Cursor` classes to work with this class. In the `Document` class, we add these two lines at the beginning of the `insert` method:

```
def insert(self, character):
    if not hasattr(character, 'character'):
        character = Character(character)
```

This is a rather strange bit of code. Its basic purpose is to check whether the character being passed in is a `Character` or a `str`. If it is a string, it is wrapped in a `Character` class so all objects in the list are `Character` objects. However, it is entirely possible that someone using our code would want to use a class that is neither `Character` nor string, using duck typing. If the object has a character attribute, we assume it is a "`Character`-like" object. But if it does not, we assume it is a "`str`-like" object and wrap it in `Character`. This helps the program take advantage of duck typing as well as polymorphism; as long as an object has a character attribute, it can be used in the `Document` class.

This generic check could be very useful, for example, if we wanted to make a programmer's editor with syntax highlighting: we'd need extra data on the character, such as what type of syntax token the character belongs to. Note that if we are doing a lot of this kind of comparison, it's probably better to implement `Character` as an abstract base class with an appropriate `__subclasshook__`, as discussed in *Chapter 3*, *When Objects Are Alike*.

In addition, we need to modify the string property on `Document` to accept the new `Character` values. All we need to do is call `str()` on each character before we join it:

```
@property
def string(self):
    return "".join((str(c) for c in self.characters))
```

This code uses a generator expression, which we'll discuss in *Chapter 9*, *The Iterator Pattern*. It's a shortcut to perform a specific action on all the objects in a sequence.

Finally, we also need to check `Character.character`, instead of just the string character we were storing before, in the `home` and `end` functions when we're looking to see whether it matches a newline character:

```
    def home(self):
        while self.document.characters[
                self.position-1].character != '\n':
            self.position -= 1
            if self.position == 0:
                # Got to beginning of file before newline
                break

    def end(self):
        while self.position < len(
                self.document.characters) and \
                self.document.characters[
                        self.position
                        ].character != '\n':
            self.position += 1
```

This completes the formatting of characters. We can test it to see that it works:

```
>>> d = Document()
>>> d.insert('h')
>>> d.insert('e')
>>> d.insert(Character('l', bold=True))
>>> d.insert(Character('l', bold=True))
>>> d.insert('o')
>>> d.insert('\n')
>>> d.insert(Character('w', italic=True))
>>> d.insert(Character('o', italic=True))
>>> d.insert(Character('r', underline=True))
>>> d.insert('l')
>>> d.insert('d')
>>> print(d.string)
he*l*lo
/w/o_rld
>>> d.cursor.home()
>>> d.delete()
```

```
>>> d.insert('W')
>>> print(d.string)
he*l*lo
W/o_rld
>>> d.characters[0].underline = True
>>> print(d.string)
_he*l*lo
W/o_rld
```

As expected, whenever we print the string, each bold character is preceded by a `*` character, each italic character by a `/` character, and each underlined character by a _ character. All our functions seem to work, and we can modify characters in the list after the fact. We have a working rich text document object that could be plugged into a proper user interface and hooked up with a keyboard for input and a screen for output. Naturally, we'd want to display real bold, italic, and underlined characters on the screen, instead of using our `__str__` method, but it was sufficient for the basic testing we demanded of it.

Your Coding Challenge

We've looked at various ways that objects, data, and methods can interact with each other in an object-oriented Python program. As usual, your first thoughts should be how you can apply these principles to your own work. Do you have any messy scripts lying around that could be rewritten using an object-oriented manager? Look through some of your old code and look for methods that are not actions. If the name isn't a verb, try rewriting it as a property.

Think about code you've written in any language. Does it break the DRY principle? Is there any duplicate code? Did you copy and paste code? Did you write two versions of similar pieces of code because you didn't feel like understanding the original code? Go back over some of your recent code now and see whether you can refactor the duplicate code using inheritance or composition. Try to pick a project you're still interested in maintaining; not code so old that you never want to touch it again. It helps keep your interest up when you do the improvements!

Ankita Thakur

Your Course Guide

Now, look back over some of the examples we saw in this chapter. Start with the cached web page example that uses a property to cache the retrieved data. An obvious problem with this example is that the cache is never refreshed. Add a timeout to the property's getter, and only return the cached page if the page has been requested before the timeout has expired. You can use the time module (time.time() - an_old_time returns the number of seconds that have elapsed since an_old_time) to determine whether the cache has expired.

Now look at the inheritance-based ZipProcessor. It might be reasonable to use composition instead of inheritance here. Instead of extending the class in the ZipReplace and ScaleZip classes, you could pass instances of those classes into the ZipProcessor constructor and call them to do the processing part. Implement this.

Which version do you find easier to use? Which is more elegant? What is easier to read? These are subjective questions; the answer varies for each of us. Knowing the answer, however, is important; if you find you prefer inheritance over composition, you have to pay attention that you don't overuse inheritance in your daily coding. If you prefer composition, make sure you don't miss opportunities to create an elegant inheritance-based solution.

Ankita Thakur

Your Course Guide

Finally, add some error handlers to the various classes we created in the case study. They should ensure single characters are entered, that you don't try to move the cursor past the end or beginning of the file, that you don't delete a character that doesn't exist, and that you don't save a file without a filename. Try to think of as many edge cases as you can, and account for them (thinking about edge cases is about 90 percent of a professional programmer's job!) Consider different ways to handle them; should you raise an exception when the user tries to move past the end of the file, or just stay on the last character?

Pay attention, in your daily coding, to the copy and paste commands. Every time you use them in your editor, consider whether it would be a good idea to improve your program's organization so that you only have one version of the code you are about to copy.

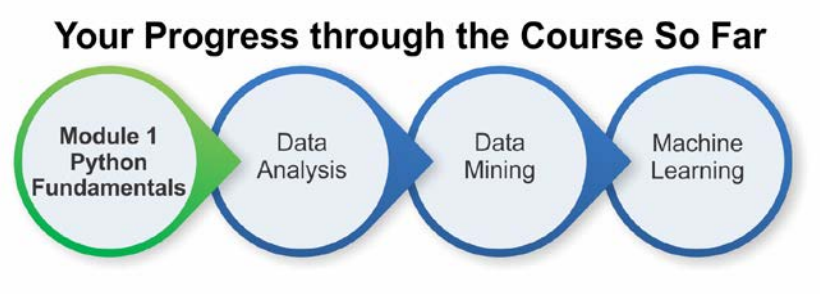
Summary of Module 1 Chapter 6

Ankita Thakur

Your Course Guide

In this chapter, we focused on identifying objects, especially objects that are not immediately apparent; objects that manage and control. Objects should have both data and behavior, but properties can be used to blur the distinction between the two. The DRY principle is an important indicator of code quality and inheritance and composition can be applied to reduce code duplication.

In the next chapter, we'll cover several of the built-in Python data structures and objects, focusing on their object-oriented properties and how they can be extended or adapted.


Your Progress through the Course So Far
Module 1
Python
Fundamentals
Data
Analysis
Data
Mining
Machine
Learning

7
Python Data Structures

In our examples so far, we've already seen many of the built-in Python data structures in action. You've probably also covered many of them in introductory books or tutorials. In this chapter, we'll be discussing the object-oriented features of these data structures, when they should be used instead of a regular class, and when they should not be used. In particular, we'll be covering:

- Tuples and named tuples
- Dictionaries
- Lists and sets
- How and why to extend built-in objects
- Three types of queues

Empty objects

Let's start with the most basic Python built-in, one that we've seen many times already, the one that we've extended in every class we have created: the `object`. Technically, we can instantiate an `object` without writing a subclass:

```
>>> o = object()
>>> o.x = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'object' object has no attribute 'x'
```

Unfortunately, as you can see, it's not possible to set any attributes on an `object` that was instantiated directly. This isn't because the Python developers wanted to force us to write our own classes, or anything so sinister. They did this to save memory; a lot of memory. When Python allows an object to have arbitrary attributes, it takes a certain amount of system memory to keep track of what attributes each object has, for storing both the attribute name and its value. Even if no attributes are stored, memory is allocated for *potential* new attributes. Given the dozens, hundreds, or thousands of objects (every class extends object) in a typical Python program; this small amount of memory would quickly become a large amount of memory. So, Python disables arbitrary properties on `object`, and several other built-ins, by default.

It is possible to restrict arbitrary properties on our own classes using **slots**. Slots are beyond the scope of this module, but you now have a search term if you are looking for more information. In normal use, there isn't much benefit to using slots, but if you're writing an object that will be duplicated thousands of times throughout the system, they can help save memory, just as they do for `object`.

It is, however, trivial to create an empty object class of our own; we saw it in our earliest example:

```
class MyObject:
    pass
```

And, as we've already seen, it's possible to set attributes on such classes:

```
>>> m = MyObject()
>>> m.x = "hello"
>>> m.x
'hello'
```

If we wanted to group properties together, we could store them in an empty object like this. But we are usually better off using other built-ins designed for storing data. It has been stressed throughout this module that classes and objects should only be used when you want to specify *both* data and behaviors. The main reason to write an empty class is to quickly block something out, knowing we'll come back later to add behavior. It is much easier to adapt behaviors to a class than it is to replace a data structure with an object and change all references to it. Therefore, it is important to decide from the outset if the data is just data, or if it is an object in disguise. Once that design decision is made, the rest of the design naturally falls into place.

Tuples and named tuples

Tuples are objects that can store a specific number of other objects in order. They are immutable, so we can't add, remove, or replace objects on the fly. This may seem like a massive restriction, but the truth is, if you need to modify a tuple, you're using the wrong data type (usually a list would be more suitable). The primary benefit of tuples' immutability is that we can use them as keys in dictionaries, and in other locations where an object requires a hash value.

Tuples are used to store data; behavior cannot be stored in a tuple. If we require behavior to manipulate a tuple, we have to pass the tuple into a function (or method on another object) that performs the action.

Tuples should generally store values that are somehow different from each other. For example, we would not put three stock symbols in a tuple, but we might create a tuple of stock symbol, current price, high, and low for the day. The primary purpose of a tuple is to aggregate different pieces of data together into one container. Thus, a tuple can be the easiest tool to replace the "object with no data" idiom.

We can create a tuple by separating the values with a comma. Usually, tuples are wrapped in parentheses to make them easy to read and to separate them from other parts of an expression, but this is not always mandatory. The following two assignments are identical (they record a stock, the current price, the high, and the low for a rather profitable company):

```
>>> stock = "FB", 75.00, 75.03, 74.90
>>> stock2 = ("FB", 75.00, 75.03, 74.90)
```

If we're grouping a tuple inside of some other object, such as a function call, list comprehension, or generator, the parentheses are required. Otherwise, it would be impossible for the interpreter to know whether it is a tuple or the next function parameter. For example, the following function accepts a tuple and a date, and returns a tuple of the date and the middle value between the stock's high and low value:

```
import datetime
def middle(stock, date):
    symbol, current, high, low = stock
    return (((high + low) / 2), date)

mid_value, date = middle(("FB", 75.00, 75.03, 74.90),
        datetime.date(2014, 10, 31))
```

The tuple is created directly inside the function call by separating the values with commas and enclosing the entire tuple in parenthesis. This tuple is then followed by a comma to separate it from the second argument.

This example also illustrates tuple unpacking. The first line inside the function unpacks the `stock` parameter into four different variables. The tuple has to be exactly the same length as the number of variables, or it will raise an exception. We can also see an example of tuple unpacking on the last line, where the tuple returned inside the function is unpacked into two values, `mid_value` and `date`. Granted, this is a strange thing to do, since we supplied the date to the function in the first place, but it gave us a chance to see unpacking at work.

Unpacking is a very useful feature in Python. We can group variables together to make storing and passing them around simpler, but the moment we need to access all of them, we can unpack them into separate variables. Of course, sometimes we only need access to one of the variables in the tuple. We can use the same syntax that we use for other sequence types (lists and strings, for example) to access an individual value:

```
>>> stock = "FB", 75.00, 75.03, 74.90
>>> high = stock[2]
>>> high
75.03
```

We can even use slice notation to extract larger pieces of tuples:

```
>>> stock[1:3]
(75.00, 75.03)
```

These examples, while illustrating how flexible tuples can be, also demonstrate one of their major disadvantages: readability. How does someone reading this code know what is in the second position of a specific tuple? They can guess, from the name of the variable we assigned it to, that it is `high` of some sort, but if we had just accessed the tuple value in a calculation without assigning it, there would be no such indication. They would have to paw through the code to find where the tuple was declared before they could discover what it does.

Accessing tuple members directly is fine in some circumstances, but don't make a habit of it. Such so-called "magic numbers" (numbers that seem to come out of thin air with no apparent meaning within the code) are the source of many coding errors and lead to hours of frustrated debugging. Try to use tuples only when you know that all the values are going to be useful at once and it's normally going to be unpacked when it is accessed. If you have to access a member directly or using a slice and the purpose of that value is not immediately obvious, at least include a comment explaining where it came from.

Named tuples

So, what do we do when we want to group values together, but know we're frequently going to need to access them individually? Well, we could use an empty object, as discussed in the previous section (but that is rarely useful unless we anticipate adding behavior later), or we could use a dictionary (most useful if we don't know exactly how many or which specific data will be stored), as we'll cover in the next section.

If, however, we do not need to add behavior to the object, and we know in advance what attributes we need to store, we can use a named tuple. Named tuples are tuples with attitude. They are a great way to group read-only data together.

Constructing a named tuple takes a bit more work than a normal tuple. First, we have to import `namedtuple`, as it is not in the namespace by default. Then, we describe the named tuple by giving it a name and outlining its attributes. This returns a class-like object that we can instantiate with the required values as many times as we want:

```
from collections import namedtuple
Stock = namedtuple("Stock", "symbol current high low")
stock = Stock("FB", 75.00, high=75.03, low=74.90)
```

The `namedtuple` constructor accepts two arguments. The first is an identifier for the named tuple. The second is a string of space-separated attributes that the named tuple can have. The first attribute should be listed, followed by a space (or comma if you prefer), then the second attribute, then another space, and so on. The result is an object that can be called just like a normal class to instantiate other objects. The constructor must have exactly the right number of arguments that can be passed in as arguments or keyword arguments. As with normal objects, we can create as many instances of this "class" as we like, with different values for each.

The resulting `namedtuple` can then be packed, unpacked, and otherwise treated like a normal tuple, but we can also access individual attributes on it as if it were an object:

```
>>> stock.high
75.03
>>> symbol, current, high, low = stock
>>> current
75.00
```

Remember that creating named tuples is a two-step process. First, use `collections.namedtuple` to create a class, and then construct instances of that class.

Named tuples are perfect for many "data only" representations, but they are not ideal for all situations. Like tuples and strings, named tuples are immutable, so we cannot modify an attribute once it has been set. For example, the current value of my company's stock has gone down since we started this discussion, but we can't set the new value:

```
>>> stock.current = 74.98
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

If we need to be able to change stored data, a dictionary may be what we need instead.

Dictionaries

Dictionaries are incredibly useful containers that allow us to map objects directly to other objects. An empty object with attributes to it is a sort of dictionary; the names of the properties map to the property values. This is actually closer to the truth than it sounds; internally, objects normally represent attributes as a dictionary, where the values are properties or methods on the objects (see the `__dict__` attribute if you don't believe me). Even the attributes on a module are stored, internally, in a dictionary.

Dictionaries are extremely efficient at looking up a value, given a specific key object that maps to that value. They should always be used when you want to find one object based on some other object. The object that is being stored is called the **value**; the object that is being used as an index is called the **key**. We've already seen dictionary syntax in some of our previous examples.

Dictionaries can be created either using the `dict()` constructor or using the `{}` syntax shortcut. In practice, the latter format is almost always used. We can prepopulate a dictionary by separating the keys from the values using a colon, and separating the key value pairs using a comma.

For example, in a stock application, we would most often want to look up prices by the stock symbol. We can create a dictionary that uses stock symbols as keys, and tuples of current, high, and low as values like this:

```
stocks = {"GOOG": (613.30, 625.86, 610.50),
          "MSFT": (30.25, 30.70, 30.19)}
```

As we've seen in previous examples, we can then look up values in the dictionary by requesting a key inside square brackets. If the key is not in the dictionary, it will raise an exception:

```
>>> stocks["GOOG"]
(613.3, 625.86, 610.5)
>>> stocks["RIM"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'RIM'
```

We can, of course, catch the `KeyError` and handle it. But we have other options. Remember, dictionaries are objects, even if their primary purpose is to hold other objects. As such, they have several behaviors associated with them. One of the most useful of these methods is the `get` method; it accepts a key as the first parameter and an optional default value if the key doesn't exist:

```
>>> print(stocks.get("RIM"))
None
>>> stocks.get("RIM", "NOT FOUND")
'NOT FOUND'
```

For even more control, we can use the `setdefault` method. If the key is in the dictionary, this method behaves just like `get`; it returns the value for that key. Otherwise, if the key is not in the dictionary, it will not only return the default value we supply in the method call (just like `get` does), it will also set the key to that same value. Another way to think of it is that `setdefault` sets a value in the dictionary only if that value has not previously been set. Then it returns the value in the dictionary, either the one that was already there, or the newly provided default value.

```
>>> stocks.setdefault("GOOG", "INVALID")
(613.3, 625.86, 610.5)
>>> stocks.setdefault("BBRY", (10.50, 10.62, 10.39))
(10.50, 10.62, 10.39)
>>> stocks["BBRY"]
(10.50, 10.62, 10.39)
```

The `GOOG` stock was already in the dictionary, so when we tried to `setdefault` it to an invalid value, it just returned the value already in the dictionary. `BBRY` was not in the dictionary, so `setdefault` returned the default value and set the new value in the dictionary for us. We then check that the new stock is, indeed, in the dictionary.

Three other very useful dictionary methods are `keys()`, `values()`, and `items()`. The first two return a list over all the keys and all the values in the dictionary. We can use these like lists or in `for` loops if we want to process all the keys or values. The `items()` method is probably the most useful; it returns an iterator over tuples of `(key, value)` pairs for every item in the dictionary. This works great with tuple unpacking in a `for` loop to loop over associated keys and values. This example does just that to print each stock in the dictionary with its current value:

```
>>> for stock, values in stocks.items():
...     print("{} last value is {}".format(stock, values[0]))
...
GOOG last value is 613.3
BBRY last value is 10.50
MSFT last value is 30.25
```

Each key/value tuple is unpacked into two variables named `stock` and `values` (we could use any variable names we wanted, but these both seem appropriate) and then printed in a formatted string.

Notice that the stocks do not show up in the same order in which they were inserted. Dictionaries, due to the efficient algorithm (known as hashing) that is used to make key lookup so fast, are inherently unsorted.

So, there are numerous ways to retrieve data from a dictionary once it has been instantiated; we can use square brackets as index syntax, the `get` method, the `setdefault` method, or iterate over the `items` method, among others.

Finally, as you likely already know, we can set a value in a dictionary using the same indexing syntax we use to retrieve a value:

```
>>> stocks["GOOG"] = (597.63, 610.00, 596.28)
>>> stocks['GOOG']
(597.63, 610.0, 596.28)
```

Google's price is lower today, so I've updated the tuple value in the dictionary. We can use this index syntax to set a value for any key, regardless of whether the key is in the dictionary. If it is in the dictionary, the old value will be replaced with the new one; otherwise, a new key/value pair will be created.

We've been using strings as dictionary keys, so far, but we aren't limited to string keys. It is common to use strings as keys, especially when we're storing data in a dictionary to gather it together (instead of using an object with named properties). But we can also use tuples, numbers, or even objects we've defined ourselves as dictionary keys. We can even use different types of keys in a single dictionary:

```
random_keys = {}
random_keys["astring"] = "somestring"
random_keys[5] = "aninteger"
random_keys[25.2] = "floats work too"
random_keys[("abc", 123)] = "so do tuples"

class AnObject:
    def __init__(self, avalue):
        self.avalue = avalue

my_object = AnObject(14)
random_keys[my_object] = "We can even store objects"
my_object.avalue = 12
try:
    random_keys[[1,2,3]] = "we can't store lists though"
except:
    print("unable to store list\n")

for key, value in random_keys.items():
    print("{} has value {}".format(key, value))
```

This code shows several different types of keys we can supply to a dictionary. It also shows one type of object that cannot be used. We've already used lists extensively, and we'll be seeing many more details of them in the next section. Because lists can change at any time (by adding or removing items, for example), they cannot hash to a specific value.

Objects that are **hashable** basically have a defined algorithm that converts the object into a unique integer value for rapid lookup. This hash is what is actually used to look up values in a dictionary. For example, strings map to integers based on the characters in the string, while tuples combine hashes of the items inside the tuple. Any two objects that are somehow considered equal (like strings with the same characters or tuples with the same values) should have the same hash value, and the hash value for an object should never ever change. Lists, however, can have their contents changed, which would change their hash value (two lists should only be equal if their contents are the same). Because of this, they can't be used as dictionary keys. For the same reason, dictionaries cannot be used as keys into other dictionaries.

In contrast, there are no limits on the types of objects that can be used as dictionary values. We can use a string key that maps to a list value, for example, or we can have a nested dictionary as a value in another dictionary.

Dictionary use cases

Dictionaries are extremely versatile and have numerous uses. There are two major ways that dictionaries can be used. The first is dictionaries where all the keys represent different instances of similar objects; for example, our stock dictionary. This is an indexing system. We use the stock symbol as an index to the values. The values could even have been complicated self-defined objects that made buy and sell decisions or set a stop-loss, rather than our simple tuples.

The second design is dictionaries where each key represents some aspect of a single structure; in this case, we'd probably use a separate dictionary for each object, and they'd all have similar (though often not identical) sets of keys. This latter situation can often also be solved with named tuples. These should typically be used when we know exactly what attributes the data must store, and we know that all pieces of the data must be supplied at once (when the item is constructed). But if we need to create or change dictionary keys over time or we don't know exactly what the keys might be, a dictionary is more suitable.

Using defaultdict

We've seen how to use `setdefault` to set a default value if a key doesn't exist, but this can get a bit monotonous if we need to set a default value every time we look up a value. For example, if we're writing code that counts the number of times a letter occurs in a given sentence, we could do this:

```
def letter_frequency(sentence):
    frequencies = {}
    for letter in sentence:
        frequency = frequencies.setdefault(letter, 0)
        frequencies[letter] = frequency + 1
    return frequencies
```

Every time we access the dictionary, we need to check that it has a value already, and if not, set it to zero. When something like this needs to be done every time an empty key is requested, we can use a different version of the dictionary, called `defaultdict`:

```
from collections import defaultdict
def letter_frequency(sentence):
    frequencies = defaultdict(int)
```

```
    for letter in sentence:
        frequencies[letter] += 1
    return frequencies
```

This code looks like it couldn't possibly work. The `defaultdict` accepts a function in its constructor. Whenever a key is accessed that is not already in the dictionary, it calls that function, with no parameters, to create a default value.

In this case, the function it calls is `int`, which is the constructor for an integer object. Normally, integers are created simply by typing an integer number into our code, and if we do create one using the `int` constructor, we pass it the item we want to create (for example, to convert a string of digits into an integer). But if we call `int` without any arguments, it returns, conveniently, the number zero. In this code, if the letter doesn't exist in the `defaultdict`, the number zero is returned when we access it. Then we add one to this number to indicate we've found an instance of that letter, and the next time we find one, that number will be returned and we can increment the value again.

The `defaultdict` is useful for creating dictionaries of containers. If we want to create a dictionary of stock prices for the past 30 days, we could use a stock symbol as the key and store the prices in `list`; the first time we access the stock price, we would want it to create an empty list. Simply pass `list` into the `defaultdict`, and it will be called every time an empty key is accessed. We can do similar things with sets or even empty dictionaries if we want to associate one with a key.

Of course, we can also write our own functions and pass them into the `defaultdict`. Suppose we want to create a `defaultdict` where each new element contains a tuple of the number of items inserted into the dictionary at that time and an empty list to hold other things. Nobody knows why we would want to create such an object, but let's have a look:

```
from collections import defaultdict
num_items = 0
def tuple_counter():
    global num_items
    num_items += 1
    return (num_items, [])

d = defaultdict(tuple_counter)
```

When we run this code, we can access empty keys and insert into the list all in one statement:

```
>>> d = defaultdict(tuple_counter)
>>> d['a'][1].append("hello")
>>> d['b'][1].append('world')
```

```
>>> d
defaultdict(<function tuple_counter at 0x82f2c6c>,
{'a': (1, ['hello']), 'b': (2, ['world'])})
```

When we print `dict` at the end, we see that the counter really was working.

This example, while succinctly demonstrating how to create our own function for `defaultdict`, is not actually very good code; using a global variable means that if we created four different `defaultdict` segments that each used `tuple_counter`, it would count the number of entries in all dictionaries, rather than having a different count for each one. It would be better to create a class and pass a method on that class to `defaultdict`.

Counter

You'd think that you couldn't get much simpler than `defaultdict(int)`, but the "I want to count specific instances in an iterable" use case is common enough that the Python developers created a specific class for it. The previous code that counts characters in a string can easily be calculated in a single line:

```
from collections import Counter
def letter_frequency(sentence):
    return Counter(sentence)
```

The `Counter` object behaves like a beefed up dictionary where the keys are the items being counted and the values are the number of such items. One of the most useful functions is the `most_common()` method. It returns a list of (key, count) tuples ordered by the count. You can optionally pass an integer argument into `most_common()` to request only the top most common elements. For example, you could write a simple polling application as follows:

```
from collections import Counter

responses = [
    "vanilla",
    "chocolate",
    "vanilla",
    "vanilla",
    "caramel",
    "strawberry",
    "vanilla"
]

print(
```

```
        "The children voted for {} ice cream".format(
            Counter(responses).most_common(1)[0][0]
        )
    )
```

Presumably, you'd get the responses from a database or by using a complicated vision algorithm to count the kids who raised their hands. Here, we hardcode it so that we can test the `most_common` method. It returns a list that has only one element (because we requested one element in the parameter). This element stores the name of the top choice at position zero, hence the double `[0][0]` at the end of the call. I think they look like a surprised face, don't you? Your computer is probably amazed it can count data so easily. It's ancestor, Hollerith's Tabulating Machine for the 1890 US census, must be so jealous!

Lists

Lists are the least object-oriented of Python's data structures. While lists are, themselves, objects, there is a lot of syntax in Python to make using them as painless as possible. Unlike many other object-oriented languages, lists in Python are simply available. We don't need to import them and rarely need to call methods on them. We can loop over a list without explicitly requesting an iterator object, and we can construct a list (as with a dictionary) with custom syntax. Further, list comprehensions and generator expressions turn them into a veritable Swiss-army knife of computing functionality.

We won't go into too much detail of the syntax; you've seen it in introductory tutorials across the Web and in previous examples in this module. You can't code Python very long without learning how to use lists! Instead, we'll be covering when lists should be used, and their nature as objects. If you don't know how to create or append to a list, how to retrieve items from a list, or what "slice notation" is, I direct you to the official Python tutorial, post-haste. It can be found online at `http://docs.python.org/3/tutorial/`.

In Python, lists should normally be used when we want to store several instances of the "same" type of object; lists of strings or lists of numbers; most often, lists of objects we've defined ourselves. Lists should always be used when we want to store items in some kind of order. Often, this is the order in which they were inserted, but they can also be sorted by some criteria.

Lists are also very useful when we need to modify the contents: insert to or delete from an arbitrary location of the list, or update a value within the list.

Like dictionaries, Python lists use an extremely efficient and well-tuned internal data structure so we can worry about what we're storing, rather than how we're storing it. Many object-oriented languages provide different data structures for queues, stacks, linked lists, and array-based lists. Python does provide special instances of some of these classes, if optimizing access to huge sets of data is required. Normally, however, the list data structure can serve all these purposes at once, and the coder has complete control over how they access it.

Don't use lists for collecting different attributes of individual items. We do not want, for example, a list of the properties a particular shape has. Tuples, named tuples, dictionaries, and objects would all be more suitable for this purpose. In some languages, they might create a list in which each alternate item is a different type; for example, they might write `['a', 1, 'b', 3]` for our letter frequency list. They'd have to use a strange loop that accesses two elements in the list at once or a modulus operator to determine which position was being accessed.

Don't do this in Python. We can group related items together using a dictionary, as we did in the previous section (if sort order doesn't matter), or using a list of tuples. Here's a rather convoluted example that demonstrates how we could do the frequency example using a list. It is much more complicated than the dictionary examples, and illustrates the effect choosing the right (or wrong) data structure can have on the readability of our code:

```
import string
CHARACTERS  = list(string.ascii_letters) + [" "]

def letter_frequency(sentence):
    frequencies = [(c, 0) for c in CHARACTERS]
    for letter in sentence:
        index = CHARACTERS.index(letter)
        frequencies[index] = (letter,frequencies[index][1]+1)
    return frequencies
```

This code starts with a list of possible characters. The `string.ascii_letters` attribute provides a string of all the letters, lowercase and uppercase, in order. We convert this to a list, and then use list concatenation (the plus operator causes two lists to be merged into one) to add one more character, the space. These are the available characters in our frequency list (the code would break if we tried to add a letter that wasn't in the list, but an exception handler could solve this).

The first line inside the function uses a list comprehension to turn the `CHARACTERS` list into a list of tuples. List comprehensions are an important, non-object-oriented tool in Python; we'll be covering them in detail in the next chapter.

Then we loop over each of the characters in the sentence. We first look up the index of the character in the `CHARACTERS` list, which we know has the same index in our frequencies list, since we just created the second list from the first. We then update that index in the frequencies list by creating a new tuple, discarding the original one. Aside from the garbage collection and memory waste concerns, this is rather difficult to read!

Like dictionaries, lists are objects too, and they have several methods that can be invoked upon them. Here are some common ones:

- The `append(element)` method adds an element to the end of the list
- The `insert(index, element)` method inserts an item at a specific position
- The `count(element)` method tells us how many times an element appears in the list
- The `index()` method tells us the index of an item in the list, raising an exception if it can't find it
- The `find()` method does the same thing, but returns `-1` instead of raising an exception for missing items
- The `reverse()` method does exactly what it says—turns the list around
- The `sort()` method has some rather intricate object-oriented behaviors, which we'll cover now

Sorting lists

Without any parameters, `sort` will generally do the expected thing. If it's a list of strings, it will place them in alphabetical order. This operation is case sensitive, so all capital letters will be sorted before lowercase letters, that is `Z` comes before `a`. If it is a list of numbers, they will be sorted in numerical order. If a list of tuples is provided, the list is sorted by the first element in each tuple. If a mixture containing unsortable items is supplied, the sort will raise a `TypeError` exception.

If we want to place objects we define ourselves into a list and make those objects sortable, we have to do a bit more work. The special method `__lt__`, which stands for "less than", should be defined on the class to make instances of that class comparable. The `sort` method on list will access this method on each object to determine where it goes in the list. This method should return `True` if our class is somehow less than the passed parameter, and `False` otherwise. Here's a rather silly class that can be sorted based on either a string or a number:

```
class WeirdSortee:
    def __init__(self, string, number, sort_num):
        self.string = string
        self.number = number
        self.sort_num = sort_num

    def __lt__(self, object):
        if self.sort_num:
            return self.number < object.number
        return self.string < object.string

    def __repr__(self):
        return"{}:{}".format(self.string, self.number)
```

The `__repr__` method makes it easy to see the two values when we print a list. The `__lt__` method's implementation compares the object to another instance of the same class (or any duck typed object that has `string`, `number`, and `sort_num` attributes; it will fail if those attributes are missing). The following output illustrates this class in action, when it comes to sorting:

```
>>> a = WeirdSortee('a', 4, True)
>>> b = WeirdSortee('b', 3, True)
>>> c = WeirdSortee('c', 2, True)
>>> d = WeirdSortee('d', 1, True)
>>> l = [a,b,c,d]
>>> l
[a:4, b:3, c:2, d:1]
>>> l.sort()
>>> l
[d:1, c:2, b:3, a:4]
```

```
>>> for i in l:
...     i.sort_num = False
...
>>> l.sort()
>>> l
[a:4, b:3, c:2, d:1]
```

The first time we call `sort`, it sorts by numbers because `sort_num` is `True` on all the objects being compared. The second time, it sorts by letters. The `__lt__` method is the only one we need to implement to enable sorting. Technically, however, if it is implemented, the class should normally also implement the similar `__gt__`, `__eq__`, `__ne__`, `__ge__`, and `__le__` methods so that all of the `<`, `>`, `==`, `!=`, `>=`, and `<=` operators also work properly. You can get this for free by implementing `__lt__` and `__eq__`, and then applying the `@total_ordering` class decorator to supply the rest:

```
from functools import total_ordering

@total_ordering
class WeirdSortee:
    def __init__(self, string, number, sort_num):
        self.string = string
        self.number = number
        self.sort_num = sort_num

    def __lt__(self, object):
        if self.sort_num:
            return self.number < object.number
        return self.string < object.string

    def __repr__(self):
        return"{}:{}".format(self.string, self.number)

    def __eq__(self, object):
        return all((
            self.string == object.string,
            self.number == object.number,
            self.sort_num == object.number
        ))
```

This is useful if we want to be able to use operators on our objects. However, if all we want to do is customize our sort orders, even this is overkill. For such a use case, the `sort` method can take an optional `key` argument. This argument is a function that can translate each object in a list into an object that can somehow be compared. For example, we can use `str.lower` as the key argument to perform a case-insensitive sort on a list of strings:

```
>>> l = ["hello", "HELP", "Helo"]
>>> l.sort()
>>> l
['HELP', 'Helo', 'hello']
>>> l.sort(key=str.lower)
>>> l
['hello', 'Helo', 'HELP']
```

Remember, even though `lower` is a method on string objects, it is also a function that can accept a single argument, `self`. In other words, `str.lower(item)` is equivalent to `item.lower()`. When we pass this function as a key, it performs the comparison on lowercase values instead of doing the default case-sensitive comparison.

There are a few sort key operations that are so common that the Python team has supplied them so you don't have to write them yourself. For example, it is often common to sort a list of tuples by something other than the first item in the list. The `operator.itemgetter` method can be used as a key to do this:

```
>>> from operator import itemgetter
>>> l = [('h', 4), ('n', 6), ('o', 5), ('p', 1), ('t', 3), ('y', 2)]
>>> l.sort(key=itemgetter(1))
>>> l
[('p', 1), ('y', 2), ('t', 3), ('h', 4), ('o', 5), ('n', 6)]
```

The `itemgetter` function is the most commonly used one (it works if the objects are dictionaries, too), but you will sometimes find use for `attrgetter` and `methodcaller`, which return attributes on an object and the results of method calls on objects for the same purpose. See the `operator` module documentation for more information.

Sets

Lists are extremely versatile tools that suit most container object applications. But they are not useful when we want to ensure objects in the list are unique. For example, a song library may contain many songs by the same artist. If we want to sort through the library and create a list of all the artists, we would have to check the list to see if we've added the artist already, before we add them again.

This is where sets come in. Sets come from mathematics, where they represent an unordered group of (usually) unique numbers. We can add a number to a set five times, but it will show up in the set only once.

In Python, sets can hold any hashable object, not just numbers. Hashable objects are the same objects that can be used as keys in dictionaries; so again, lists and dictionaries are out. Like mathematical sets, they can store only one copy of each object. So if we're trying to create a list of song artists, we can create a set of string names and simply add them to the set. This example starts with a list of (song, artist) tuples and creates a set of the artists:

```
song_library = [("Phantom Of The Opera", "Sarah Brightman"),
        ("Knocking On Heaven's Door", "Guns N' Roses"),
        ("Captain Nemo", "Sarah Brightman"),
        ("Patterns In The Ivy", "Opeth"),
        ("November Rain", "Guns N' Roses"),
        ("Beautiful", "Sarah Brightman"),
        ("Mal's Song", "Vixy and Tony")]

artists = set()
for song, artist in song_library:
    artists.add(artist)

print(artists)
```

There is no built-in syntax for an empty set as there is for lists and dictionaries; we create a set using the `set()` constructor. However, we can use the curly braces (borrowed from dictionary syntax) to create a set, so long as the set contains values. If we use colons to separate pairs of values, it's a dictionary, as in `{'key': 'value', 'key2': 'value2'}`. If we just separate values with commas, it's a set, as in `{'value', 'value2'}`. Items can be added individually to the set using its `add` method. If we run this script, we see that the set works as advertised:

```
{'Sarah Brightman', "Guns N' Roses", 'Vixy and Tony', 'Opeth'}
```

If you're paying attention to the output, you'll notice that the items are not printed in the order they were added to the sets. Sets, like dictionaries, are unordered. They both use an underlying hash-based data structure for efficiency. Because they are unordered, sets cannot have items looked up by index. The primary purpose of a set is to divide the world into two groups: "things that are in the set", and, "things that are not in the set". It is easy to check whether an item is in the set or to loop over the items in a set, but if we want to sort or order them, we'll have to convert the set to a list. This output shows all three of these activities:

```
>>> "Opeth" in artists
True
>>> for artist in artists:
...     print("{} plays good music".format(artist))
...
Sarah Brightman plays good music
Guns N' Roses plays good music
Vixy and Tony play good music
Opeth plays good music
>>> alphabetical = list(artists)
>>> alphabetical.sort()
>>> alphabetical
["Guns N' Roses", 'Opeth', 'Sarah Brightman', 'Vixy and Tony']
```

While the primary *feature* of a set is uniqueness, that is not its primary *purpose*. Sets are most useful when two or more of them are used in combination. Most of the methods on the set type operate on other sets, allowing us to efficiently combine or compare the items in two or more sets. These methods have strange names, since they use the same terminology used in mathematics. We'll start with three methods that return the same result, regardless of which is the calling set and which is the called set.

The `union` method is the most common and easiest to understand. It takes a second set as a parameter and returns a new set that contains all elements that are in *either* of the two sets; if an element is in both original sets, it will, of course, only show up once in the new set. Union is like a logical `or` operation, indeed, the `|` operator can be used on two sets to perform the union operation, if you don't like calling methods.

Conversely, the intersection method accepts a second set and returns a new set that contains only those elements that are in *both* sets. It is like a logical `and` operation, and can also be referenced using the `&` operator.

Finally, the `symmetric_difference` method tells us what's left; it is the set of objects that are in one set or the other, but not both. The following example illustrates these methods by comparing some artists from my song library to those in my sister's:

```
my_artists = {"Sarah Brightman", "Guns N' Roses",
        "Opeth", "Vixy and Tony"}

auburns_artists = {"Nickelback", "Guns N' Roses",
        "Savage Garden"}

print("All: {}".format(my_artists.union(auburns_artists)))
print("Both: {}".format(auburns_artists.intersection(my_artists)))
print("Either but not both: {}".format(
    my_artists.symmetric_difference(auburns_artists)))
```

If we run this code, we see that these three methods do what the print statements suggest they will do:

```
All: {'Sarah Brightman', "Guns N' Roses", 'Vixy and Tony',
'Savage Garden', 'Opeth', 'Nickelback'}
Both: {"Guns N' Roses"}
Either but not both: {'Savage Garden', 'Opeth', 'Nickelback',
'Sarah Brightman', 'Vixy and Tony'}
```

These methods all return the same result, regardless of which set calls the other. We can say `my_artists.union(auburns_artists)` or `auburns_artists.union(my_artists)` and get the same result. There are also methods that return different results depending on who is the caller and who is the argument.

These methods include `issubset` and `issuperset`, which are the inverse of each other. Both return a `bool`. The `issubset` method returns `True`, if all of the items in the calling set are also in the set passed as an argument. The `issuperset` method returns `True` if all of the items in the argument are also in the calling set. Thus `s.issubset(t)` and `t.issuperset(s)` are identical. They will both return `True` if `t` contains all the elements in `s`.

Finally, the `difference` method returns all the elements that are in the calling set, but not in the set passed as an argument; this is like half a `symmetric_difference`. The `difference` method can also be represented by the `-` operator. The following code illustrates these methods in action:

```
my_artists = {"Sarah Brightman", "Guns N' Roses",
        "Opeth", "Vixy and Tony"}

bands = {"Guns N' Roses", "Opeth"}

print("my_artists is to bands:")
print("issuperset: {}".format(my_artists.issuperset(bands)))
print("issubset: {}".format(my_artists.issubset(bands)))
print("difference: {}".format(my_artists.difference(bands)))
print("*"*20)
print("bands is to my_artists:")
print("issuperset: {}".format(bands.issuperset(my_artists)))
print("issubset: {}".format(bands.issubset(my_artists)))
print("difference: {}".format(bands.difference(my_artists)))
```

This code simply prints out the response of each method when called from one set on the other. Running it gives us the following output:

```
my_artists is to bands:
issuperset: True
issubset: False
difference: {'Sarah Brightman', 'Vixy and Tony'}
********************
bands is to my_artists:
issuperset: False
issubset: True
difference: set()
```

The `difference` method, in the second case, returns an empty set, since there are no items in `bands` that are not in `my_artists`.

The `union`, `intersection`, and `difference` methods can all take multiple sets as arguments; they will return, as we might expect, the set that is created when the operation is called on all the parameters.

So the methods on sets clearly suggest that sets are meant to operate on other sets, and that they are not just containers. If we have data coming in from two different sources and need to quickly combine them in some way, to determine where the data overlaps or is different, we can use set operations to efficiently compare them. Or if we have data incoming that may contain duplicates of data that has already been processed, we can use sets to compare the two and process only the new data.

Finally, it is valuable to know that sets are much more efficient than lists when checking for membership using the `in` keyword. If you use the syntax `value in container` on a set or a list, it will return `True` if one of the elements in `container` is equal to `value` and `False` otherwise. However, in a list, it will look at every object in the container until it finds the value, whereas in a set, it simply hashes the value and checks for membership. This means that a set will find the value in the same amount of time no matter how big the container is, but a list will take longer and longer to search for a value as the list contains more and more values.

Extending built-ins

Now, we'll go into more detail as to when we would want to do that.

When we have a built-in container object that we want to add functionality to, we have two options. We can either create a new object, which holds that container as an attribute (composition), or we can subclass the built-in object and add or adapt methods on it to do what we want (inheritance).

Composition is usually the best alternative if all we want to do is use the container to store some objects using that container's features. That way, it's easy to pass that data structure into other methods and they will know how to interact with it. But we need to use inheritance if we want to change the way the container actually works. For example, if we want to ensure every item in a `list` is a string with exactly five characters, we need to extend `list` and override the `append()` method to raise an exception for invalid input. We'd also minimally have to override `__setitem__(self, index, value)`, a special method on lists that is called whenever we use the `x[index] = "value"` syntax, and the `extend()` method.

Yes, lists are objects. All that special non-object-oriented looking syntax we've been looking at for accessing lists or dictionary keys, looping over containers, and similar tasks is actually "syntactic sugar" that maps to an object-oriented paradigm underneath. We might ask the Python designers why they did this. Isn't object-oriented programming *always* better? That question is easy to answer. In the following hypothetical examples, which is easier to read, as a programmer? Which requires less typing?

```
c = a + b
c = a.add(b)

l[0] = 5
l.setitem(0, 5)
d[key] = value
d.setitem(key, value)

for x in alist:
    #do something with x
it = alist.iterator()
while it.has_next():
    x = it.next()
    #do something with x
```

The highlighted sections show what object-oriented code might look like (in practice, these methods actually exist as special double-underscore methods on associated objects). Python programmers agree that the non-object-oriented syntax is easier both to read and to write. Yet all of the preceding Python syntaxes map to object-oriented methods underneath the hood. These methods have special names (with double-underscores before and after) to remind us that there is a better syntax out there. However, it gives us the means to override these behaviors. For example, we can make a special integer that always returns `0` when we add two of them together:

```
class SillyInt(int):
    def __add__(self, num):
        return 0
```

This is an extremely bizarre thing to do, granted, but it perfectly illustrates these object-oriented principles in action:

```
>>> a = SillyInt(1)
>>> b = SillyInt(2)
>>> a + b
0
```

The awesome thing about the `__add__` method is that we can add it to any class we write, and if we use the `+` operator on instances of that class, it will be called. This is how string, tuple, and list concatenation works, for example.

This is true of all the special methods. If we want to use `x in myobj` syntax for a custom-defined object, we can implement `__contains__`. If we want to use `myobj[i] = value` syntax, we supply a `__setitem__` method and if we want to use `something = myobj[i]`, we implement `__getitem__`.

There are 33 of these special methods on the `list` class. We can use the `dir` function to see all of them:

```
>>> dir(list)


['__add__', '__class__', '__contains__', '__delattr__','__delitem__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__
getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__
new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__
subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort'
```

Further, if we desire additional information on how any of these methods works, we can use the `help` function:

```
>>> help(list.__add__)
Help on wrapper_descriptor:


__add__(self, value, /)
    Return self+value.
```

The plus operator on lists concatenates two lists. We don't have room to discuss all of the available special functions in this module, but you are now able to explore all this functionality with `dir` and `help`. The official online Python reference (`https://docs.python.org/3/`) has plenty of useful information as well. Focus, especially, on the abstract base classes discussed in the `collections` module.

So, to get back to the earlier point about when we would want to use composition versus inheritance: if we need to somehow change any of the methods on the class—including the special methods—we definitely need to use inheritance. If we used composition, we could write methods that do the validation or alterations and ask the caller to use those methods, but there is nothing stopping them from accessing the property directly. They could insert an item into our list that does not have five characters, and that might confuse other methods in the list.

Often, the need to extend a built-in data type is an indication that we're using the wrong sort of data type. It is not always the case, but if we are looking to extend a built-in, we should carefully consider whether or not a different data structure would be more suitable.

For example, consider what it takes to create a dictionary that remembers the order in which keys were inserted. One way to do this is to keep an ordered list of keys that is stored in a specially derived subclass of `dict`. Then we can override the methods `keys`, `values`, `__iter__`, and `items` to return everything in order. Of course, we'll also have to override `__setitem__` and `setdefault` to keep our list up to date. There are likely to be a few other methods in the output of `dir(dict)` that need overriding to keep the list and dictionary consistent (`clear` and `__delitem__` come to mind, to track when items are removed), but we won't worry about them for this example.

So we'll be extending `dict` and adding a list of ordered keys. Trivial enough, but where do we create the actual list? We could include it in the `__init__` method, which would work just fine, but we have no guarantees that any subclass will call that initializer. Remember the `__new__` method we discussed in *Chapter 2*, *Objects in Python*? I said it was generally only useful in very special cases. This is one of those special cases. We know `__new__` will be called exactly once, and we can create a list on the new instance that will always be available to our class. With that in mind, here is our entire sorted dictionary:

```
from collections import KeysView, ItemsView, ValuesView
class DictSorted(dict):
    def __new__(*args, **kwargs):
        new_dict = dict.__new__(*args, **kwargs)
        new_dict.ordered_keys = []
        return new_dict

    def __setitem__(self, key, value):
        '''self[key] = value syntax'''
        if key not in self.ordered_keys:
            self.ordered_keys.append(key)
        super().__setitem__(key, value)

    def setdefault(self, key, value):
        if key not in self.ordered_keys:
            self.ordered_keys.append(key)
        return super().setdefault(key, value)

    def keys(self):
        return KeysView(self)

    def values(self):
```

```
            return ValuesView(self)

        def items(self):
            return ItemsView(self)

        def __iter__(self):
            '''for x in self syntax'''
            return self.ordered_keys.__iter__()
```

The `__new__` method creates a new dictionary and then puts an empty list on that object. We don't override `__init__`, as the default implementation works (actually, this is only true if we initialize an empty `DictSorted` object, which is standard behavior. If we want to support other variations of the `dict` constructor, which accept dictionaries or lists of tuples, we'd need to fix `__init__` to also update our `ordered_keys` list). The two methods for setting items are very similar; they both update the list of keys, but only if the item hasn't been added before. We don't want duplicates in the list, but we can't use a set here; it's unordered!

The `keys`, `items`, and `values` methods all return views onto the dictionary. The collections library provides three read-only `View` objects onto the dictionary; they use the `__iter__` method to loop over the keys, and then use `__getitem__` (which we didn't need to override) to retrieve the values. So, we only need to define our custom `__iter__` method to make these three views work. You would think the superclass would create these views properly using polymorphism, but if we don't override these three methods, they don't return properly ordered views.

Finally, the `__iter__` method is the really special one; it ensures that if we loop over the dictionary's keys (using `for...in` syntax), it will return the values in the correct order. It does this by returning the `__iter__` of the `ordered_keys` list, which returns the same iterator object that would be used if we used `for...in` on the list instead. Since `ordered_keys` is a list of all available keys (due to the way we overrode other methods), this is the correct iterator object for the dictionary as well.

Let's look at a few of these methods in action, compared to a normal dictionary:

```
>>> ds = DictSorted()
>>> d = {}
>>> ds['a'] = 1
>>> ds['b'] = 2
>>> ds.setdefault('c', 3)
3
>>> d['a'] = 1
>>> d['b'] = 2
>>> d.setdefault('c', 3)
3
```

```
>>> for k,v in ds.items():
...     print(k,v)
...
a 1
b 2
c 3
>>> for k,v in d.items():
...     print(k,v)
...
a 1
c 3
b 2
```

Ah, our dictionary is sorted and the normal dictionary is not. Hurray!

If you wanted to use this class in production, you'd have to override several other special methods to ensure the keys are up to date in all cases. However, you don't need to do this; the functionality this class provides is already available in Python, using the `OrderedDict` object in the `collections` module. Try importing the class from `collections`, and use `help(OrderedDict)` to find out more about it.

Queues

Queues are peculiar data structures because, like sets, their functionality can be handled entirely using lists. However, while lists are extremely versatile general-purpose tools, they are occasionally not the most efficient data structure for container operations. If your program is using a small dataset (up to hundreds or even thousands of elements on today's processors), then lists will probably cover all your use cases. However, if you need to scale your data into the millions, you may need a more efficient container for your particular use case. Python therefore provides three types of queue data structures, depending on what kind of access you are looking for. All three utilize the same API, but differ in both behavior and data structure.

Before we start our queues, however, consider the trusty list data structure. Python lists are the most advantageous data structure for many use cases:

- They support efficient random access to any element in the list

- They have strict ordering of elements
- They support the append operation efficiently

They tend to be slow, however, if you are inserting elements anywhere but the end of the list (especially so if it's the beginning of the list). As we discussed in the section on sets, they are also slow for checking if an element exists in the list, and by extension, searching. Storing data in a sorted order or reordering the data can also be inefficient.

Let's look at the three types of containers provided by the Python `queue` module.

FIFO queues

FIFO stands for **First In First Out** and represents the most commonly understood definition of the word "queue". Imagine a line of people standing in line at a bank or cash register. The first person to enter the line gets served first, the second person in line gets served second, and if a new person desires service, they join the end of the line and wait their turn.

The Python `Queue` class is just like that. It is typically used as a sort of communication medium when one or more objects is producing data and one or more other objects is consuming the data in some way, probably at a different rate. Think of a messaging application that is receiving messages from the network, but can only display one message at a time to the user. The other messages can be buffered in a queue in the order they are received. FIFO queues are utilized a lot in such concurrent applications. (We'll talk more about concurrency in *Chapter 12*, *Testing Object-oriented Programs*.)

The `Queue` class is a good choice when you don't need to access any data inside the data structure except the next object to be consumed. Using a list for this would be less efficient because under the hood, inserting data at (or removing from) the beginning of a list can require shifting every other element in the list.

Queues have a very simple API. A `Queue` can have "infinite" (until the computer runs out of memory) capacity, but it is more commonly bounded to some maximum size. The primary methods are `put()` and `get()`, which add an element to the back of the line, as it were, and retrieve them from the front, in order. Both of these methods accept optional arguments to govern what happens if the operation cannot successfully complete because the queue is either empty (can't get) or full (can't put). The default behavior is to block or idly wait until the `Queue` object has data or room available to complete the operation. You can have it raise exceptions instead by passing the `block=False` parameter. Or you can have it wait a defined amount of time before raising an exception by passing a `timeout` parameter.

The class also has methods to check whether the `Queue` is `full()` or `empty()` and there are a few additional methods to deal with concurrent access that we won't discuss here. Here is a interactive session demonstrating these principles:

```
>>> from queue import Queue
>>> lineup = Queue(maxsize=3)
>>> lineup.get(block=False)
Traceback (most recent call last):
  File "<ipython-input-5-a1c8d8492c59>", line 1, in <module>
    lineup.get(block=False)
  File "/usr/lib64/python3.3/queue.py", line 164, in get
    raise Empty
queue.Empty
>>> lineup.put("one")
>>> lineup.put("two")
>>> lineup.put("three")
>>> lineup.put("four", timeout=1)
Traceback (most recent call last):
  File "<ipython-input-9-4b9db399883d>", line 1, in <module>
    lineup.put("four", timeout=1)
  File "/usr/lib64/python3.3/queue.py", line 144, in put
raise Full
queue.Full
>>> lineup.full()
True
>>> lineup.get()
'one'
>>> lineup.get()
'two'
>>> lineup.get()
'three'
>>> lineup.empty()
True
```

Underneath the hood, Python implements queues on top of the `collections.deque` data structure. Deques are advanced data structures that permits efficient access to both ends of the collection. It provides a more flexible interface than is exposed by `Queue`. I refer you to the Python documentation if you'd like to experiment more with it.

LIFO queues

LIFO (**Last In First Out**) queues are more frequently called **stacks**. Think of a stack of papers where you can only access the top-most paper. You can put another paper on top of the stack, making it the new top-most paper, or you can take the top-most paper away to reveal the one beneath it.

Traditionally, the operations on stacks are named push and pop, but the Python `queue` module uses the exact same API as for FIFO queues: `put()` and `get()`. However, in a LIFO queue, these methods operate on the "top" of the stack instead of at the front and back of a line. This is an excellent example of polymorphism. If you look at the `Queue` source code in the Python standard library, you'll actually see that there is a superclass with subclasses for FIFO and LIFO queues that implement the few operations (operating on the top of a stack instead of front and back of a `deque` instance) that are critically different between the two.

Here's an example of the LIFO queue in action:

```
>>> from queue import LifoQueue
>>> stack = LifoQueue(maxsize=3)
>>> stack.put("one")
>>> stack.put("two")
>>> stack.put("three")
>>> stack.put("four", block=False)
Traceback (most recent call last):
  File "<ipython-input-21-5473b359e5a8>", line 1, in <module>
    stack.put("four", block=False)
  File "/usr/lib64/python3.3/queue.py", line 133, in put
    raise Full
queue.Full

>>> stack.get()
'three'
>>> stack.get()
```

```
'two'
>>> stack.get()
'one'
>>> stack.empty()
True
>>> stack.get(timeout=1)
Traceback (most recent call last):
  File "<ipython-input-26-28e084a84a10>", line 1, in <module>
    stack.get(timeout=1)
  File "/usr/lib64/python3.3/queue.py", line 175, in get
    raise Empty
queue.Empty
```

You might wonder why you couldn't just use the `append()` and `pop()` methods on a standard list. Quite frankly, that's probably what I would do. I rarely have occasion to use the `LifoQueue` class in production code. Working with the end of a list is an efficient operation; so efficient, in fact, that the `LifoQueue` uses a standard list under the hood!

There are a couple of reasons that you might want to use `LifoQueue` instead of a list. The most important one is that `LifoQueue` supports clean concurrent access from multiple threads. If you need stack-like behavior in a concurrent setting, you should leave the list at home. Second, `LifoQueue` enforces the stack interface. You can't unwittingly insert a value to the wrong position in a `LifoQueue`, for example (although, as an exercise, you can work out how to do this completely wittingly).

Priority queues

The priority queue enforces a very different style of ordering from the previous queue implementations. Once again, they follow the exact same `get()` and `put()` API, but instead of relying on the order that items arrive to determine when they should be returned, the most "important" item is returned. By convention, the most important, or highest priority item is the one that sorts lowest using the less than operator.

A common convention is to store tuples in the priority queue, where the first element in the tuple is the priority for that element, and the second element is the data. Another common paradigm is to implement the `__lt__` method, as we discussed earlier in this chapter. It is perfectly acceptable to have multiple elements with the same priority in the queue, although there are no guarantees on which one will be returned first.

A priority queue might be used, for example, by a search engine to ensure it refreshes the content of the most popular web pages before crawling sites that are less likely to be searched for. A product recommendation tool might use one to display information about the most highly ranked products while still loading data for the lower ranks.

Note that a priority queue will always return the most important element currently in the queue. The `get()` method will block (by default) if the queue is empty, but it will not block and wait for a higher priority element to be added if there is already something in the queue. The queue knows nothing about elements that have not been added yet (or even about elements that have been previously extracted), and only makes decisions based on the current contents of the queue.

This interactive session shows a priority queue in action, using tuples as weights to determine what order items are processed in:

```
>>> heap.put((3, "three"))
>>> heap.put((4, "four"))
>>> heap.put((1, "one") )
>>> heap.put((2, "two"))
>>> heap.put((5, "five"), block=False)
Traceback (most recent call last):
  File "<ipython-input-23-d4209db364ed>", line 1, in <module>
    heap.put((5, "five"), block=False)
  File "/usr/lib64/python3.3/queue.py", line 133, in put
    raise Full
Full
>>> while not heap.empty():
    print(heap.get())
(1, 'one')
(2, 'two')
(3, 'three')
(4, 'four')
```

Priority queues are almost universally implemented using the `heap` data structure. Python's implementation utilizes the `heapq` module to effectively store a heap inside a normal list. I direct you to an algorithm and data-structure's textbook for more information on heaps, not to mention many other fascinating structures we haven't covered here. No matter what the data structure, you can use object-oriented principles to wrap relevant algorithms (behaviors), such as those supplied in the `heapq` module, around the data they are structuring in the computer's memory, just as the `queue` module has done on our behalf in the standard library.

Case study

To tie everything together, we'll be writing a simple link collector, which will visit a website and collect every link on every page it finds in that site. Before we start, though, we'll need some test data to work with. Simply write some HTML files to work with that contain links to each other and to other sites on the Internet, something like this:

```
<html>
    <body>
        <a href="contact.html">Contact us</a>
        <a href="blog.html">Blog</a>
        <a href="esme.html">My Dog</a>
        <a href="/hobbies.html">Some hobbies</a>
        <a href="/contact.html">Contact AGAIN</a>
        <a href="http://www.archlinux.org/">Favorite OS</a>
    </body>
</html>
```

Name one of the files `index.html` so it shows up first when pages are served. Make sure the other files exist, and keep things complicated so there is lots of linking between them. The examples for this chapter include a directory called `case_study_serve` (one of the lamest personal websites in existence!) if you would rather not set them up yourself.

Now, start a simple web server by entering the directory containing all these files and run the following command:

```
python3 -m http.server
```

This will start a server running on port 8000; you can see the pages you made by visiting `http://localhost:8000/` in your web browser.

I doubt anyone can get a website up and running with less work! Never let it be said, "you can't do that easily with Python."

The goal will be to pass our collector the base URL for the site (in this case: `http://localhost:8000/`), and have it create a list containing every unique link on the site. We'll need to take into account three types of URLs (links to external sites, which start with `http://`, absolute internal links, which start with a `/` character, and relative links, for everything else). We also need to be aware that pages may link to each other in a loop; we need to be sure we don't process the same page multiple times, or it may never end. With all this uniqueness going on, it sounds like we're going to need some sets.

Before we get into that, let's start with the basics. What code do we need to connect to a page and parse all the links from that page?

```
from urllib.request import urlopen
from urllib.parse import urlparse
import re
import sys
LINK_REGEX = re.compile(
        "<a [^>]*href=['\"]([^'\"]+)['\"][^>]*>")

class LinkCollector:
    def __init__(self, url):
        self.url = "" + urlparse(url).netloc

    def collect_links(self, path="/"):
        full_url = self.url + path
        page = str(urlopen(full_url).read())
        links = LINK_REGEX.findall(page)
        print(links)

if __name__ == "__main__":
    LinkCollector(sys.argv[1]).collect_links()
```

This is a short piece of code, considering what it's doing. It connects to the server in the argument passed on the command line, downloads the page, and extracts all the links on that page. The `__init__` method uses the `urlparse` function to extract just the hostname from the URL; so even if we pass in `http://localhost:8000/some/page.html`, it will still operate on the top level of the host `http://localhost:8000/`. This makes sense, because we want to collect all the links on the site, although it assumes every page is connected to the index by some sequence of links.

The `collect_links` method connects to and downloads the specified page from the server, and uses a regular expression to find all the links in the page. Regular expressions are an extremely powerful string processing tool. Unfortunately, they have a steep learning curve; if you haven't used them before, I strongly recommend studying any of the entire books or websites on the topic. If you don't think they're worth knowing, try writing the preceding code without them and you'll change your mind.

The example also stops in the middle of the `collect_links` method to print the value of links. This is a common way to test a program as we're writing it: stop and output the value to ensure it is the value we expect. Here's what it outputs for our example:

```
['contact.html', 'blog.html', 'esme.html', '/hobbies.html',
'/contact.html', 'http://www.archlinux.org/']
```

So now we have a collection of all the links in the first page. What can we do with it? We can't just pop the links into a set to remove duplicates because links may be relative or absolute. For example, `contact.html` and `/contact.html` point to the same page. So the first thing we should do is normalize all the links to their full URL, including hostname and relative path. We can do this by adding a `normalize_url` method to our object:

```
    def normalize_url(self, path, link):
        if link.startswith("http://"):
            return link
        elif link.startswith("/"):
            return self.url + link
        else:
            return self.url + path.rpartition(
                '/')[0] + '/' + link
```

This method converts each URL to a complete address that includes protocol and hostname. Now the two contact pages have the same value and we can store them in a set. We'll have to modify `__init__` to create the set, and `collect_links` to put all the links into it.

Then, we'll have to visit all the non-external links and collect them too. But wait a minute; if we do this, how do we keep from revisiting a link when we encounter the same page twice? It looks like we're actually going to need two sets: a set of collected links, and a set of visited links. This suggests that we were wise to choose a set to represent our data; we know that sets are most useful when we're manipulating more than one of them. Let's set these up:

```
class LinkCollector:
    def __init__(self, url):
        self.url = "http://+" + urlparse(url).netloc
        self.collected_links = set()
        self.visited_links = set()

    def collect_links(self, path="/"):
        full_url = self.url + path
        self.visited_links.add(full_url)
        page = str(urlopen(full_url).read())
        links = LINK_REGEX.findall(page)
        links = {self.normalize_url(path, link
            ) for link in links}
        self.collected_links = links.union(
                self.collected_links)
        unvisited_links = links.difference(
                self.visited_links)
```

```
            print(links, self.visited_links,
                    self.collected_links, unvisited_links)
```

The line that creates the normalized list of links uses a `set` comprehension, no different from a list comprehension, except that the result is a set of values. We'll be covering these in detail in the next chapter. Once again, the method stops to print out the current values, so we can verify that we don't have our sets confused, and that `difference` really was the method we wanted to call to collect `unvisited_links`. We can then add a few lines of code that loop over all the unvisited links and add them to the collection as well:

```
            for link in unvisited_links:
                if link.startswith(self.url):
                    self.collect_links(urlparse(link).path)
```

The `if` statement ensures that we are only collecting links from the one website; we don't want to go off and collect all the links from all the pages on the Internet (unless we're Google or the Internet Archive!). If we modify the main code at the bottom of the program to output the collected links, we can see it seems to have collected them all:

```
if __name__ == "__main__":
    collector = LinkCollector(sys.argv[1])
    collector.collect_links()
    for link in collector.collected_links:
        print(link)
```

It displays all the links we've collected, and only once, even though many of the pages in my example linked to each other multiple times:

```
$ python3 link_collector.py http://localhost:8000
http://localhost:8000/
http://en.wikipedia.org/wiki/Cavalier_King_Charles_Spaniel
http://beluminousyoga.com
http://archlinux.me/dusty/
http://localhost:8000/blog.html
http://ccphillips.net/
http://localhost:8000/contact.html
http://localhost:8000/taichi.html
http://www.archlinux.org/
http://localhost:8000/esme.html
http://localhost:8000/hobbies.html
```

Even though it collected links *to* external pages, it didn't go off collecting links *from* any of the external pages we linked to. This is a great little program if we want to collect all the links in a site. But it doesn't give me all the information I might need to build a site map; it tells me which pages I have, but it doesn't tell me which pages link to other pages. If we want to do that instead, we're going to have to make some modifications.

The first thing we should do is look at our data structures. The set of collected links doesn't work anymore; we want to know which links were linked to from which pages. The first thing we could do, then, is turn that set into a dictionary of sets for each page we visit. The dictionary keys will represent the exact same data that is currently in the set. The values will be sets of all the links on that page. Here are the changes:

```
from urllib.request import urlopen
from urllib.parse import urlparse
import re
import sys
LINK_REGEX = re.compile(
        "<a [^>]*href=['\"]([^'\"]+)['\"][^>]*>")

class LinkCollector:
    def __init__(self, url):
        self.url = "http://%s" % urlparse(url).netloc
        self.collected_links = {}
        self.visited_links = set()

    def collect_links(self, path="/"):
        full_url = self.url + path
        self.visited_links.add(full_url)
        page = str(urlopen(full_url).read())
        links = LINK_REGEX.findall(page)
        links = {self.normalize_url(path, link
            ) for link in links}
        self.collected_links[full_url] = links
        for link in links:
            self.collected_links.setdefault(link, set())
        unvisited_links = links.difference(
                self.visited_links)
        for link in unvisited_links:
            if link.startswith(self.url):
                self.collect_links(urlparse(link).path)

    def normalize_url(self, path, link):
        if link.startswith("http://"):
```

```
            return link
        elif link.startswith("/"):
            return self.url + link
        else:
            return self.url + path.rpartition('/'
                    )[0] + '/' + link
if __name__ == "__main__":
    collector = LinkCollector(sys.argv[1])
    collector.collect_links()
    for link, item in collector.collected_links.items():
        print("{}: {}".format(link, item))
```

It is a surprisingly small change; the line that originally created a union of two sets has been replaced with three lines that update the dictionary. The first of these simply tells the dictionary what the collected links for that page are. The second creates an empty set for any items in the dictionary that have not already been added to the dictionary, using `setdefault`. The result is a dictionary that contains all the links as its keys, mapped to sets of links for all the internal links, and empty sets for the external links.

Finally, instead of recursively calling `collect_links`, we can use a queue to store the links that haven't been processed yet. This implementation won't support it, but this would be a good first step to creating a multithreaded version that makes multiple requests in parallel to save time.

```
from urllib.request import urlopen
from urllib.parse import urlparse
import re
import sys
from queue import Queue
LINK_REGEX = re.compile("<a [^>]*href=['\"]([^'\"]+)['\"][^>]*>")


class LinkCollector:
    def __init__(self, url):
        self.url = "http://%s" % urlparse(url).netloc
        self.collected_links = {}
        self.visited_links = set()

    def collect_links(self):
        queue = Queue()
        queue.put(self.url)
        while not queue.empty():
            url = queue.get().rstrip('/')
            self.visited_links.add(url)
            page = str(urlopen(url).read())
```

```
                links = LINK_REGEX.findall(page)
                links = {
                    self.normalize_url(urlparse(url).path, link)
                    for link in links
                }
                self.collected_links[url] = links
                for link in links:
                    self.collected_links.setdefault(link, set())
                unvisited_links = links.difference(self.visited_links)
                for link in unvisited_links:
                    if link.startswith(self.url):
                        queue.put(link)

        def normalize_url(self, path, link):
            if link.startswith("http://"):
                return link.rstrip('/')
            elif link.startswith("/"):
                return self.url + link.rstrip('/')
            else:
                return self.url + path.rpartition('/')[0] + '/' + link.
    rstrip('/')

    if __name__ == "__main__":
        collector = LinkCollector(sys.argv[1])
        collector.collect_links()
        for link, item in collector.collected_links.items():
            print("%s: %s" % (link, item))
```

I had to manually strip any trailing forward slashes in the `normalize_url` method to remove duplicates in this version of the code.

Because the end result is an unsorted dictionary, there is no restriction on what order the links should be processed in. Therefore, we could just as easily have used a `LifoQueue` instead of a `Queue` here. A priority queue probably wouldn't make a lot of sense since there is no obvious priority to attach to a link in this case.

Your Coding Challenge

The best way to learn how to choose the correct data structure is to do it wrong a few times. Take some code you've recently written, or write some new code that uses a list. Try rewriting it using some different data structures. Which ones make more sense? Which ones don't? Which have the most elegant code?

Try this with a few different pairs of data structures. You can look at examples you've done for previous chapter exercises. Are there objects with methods where you could have used namedtuple or dict instead? Attempt both and see. Are there dictionaries that could have been sets because you don't really access the values? Do you have lists that check for duplicates? Would a set suffice? Or maybe several sets? Would one of the queue implementations be more efficient? Is it useful to restrict the API to the top of a stack rather than allowing random access to the list?

Ankita Thakur

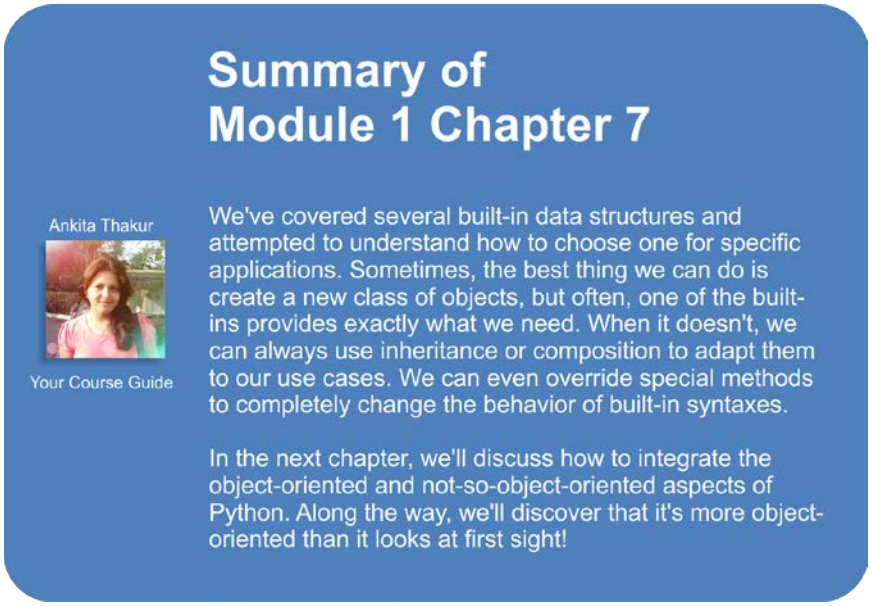
Your Course Guide

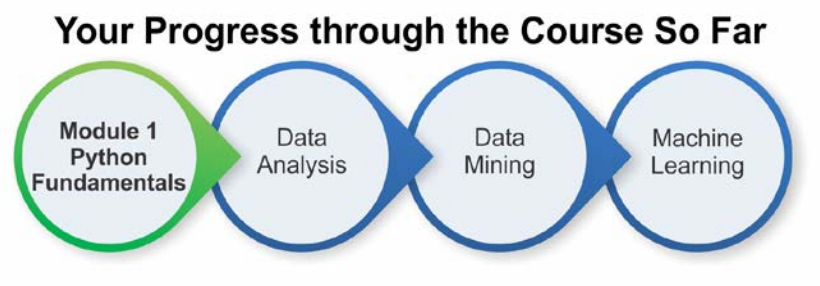
If you want some specific examples to work with, try adapting the link collector to also save the title used for each link. Perhaps you can generate a site map in HTML that lists all the pages on the site, and contains a list of links to other pages, named with the same link titles.

Have you written any container objects recently that you could improve by inheriting a built-in and overriding some of the "special" double-underscore methods? You may have to do some research (using dir and help, or the Python library reference) to find out which methods need overriding. Are you sure inheritance is the correct tool to apply; could a composition-based solution be more effective? Try both (if it's possible) before you decide. Try to find different situations where each method is better than the other.

If you were familiar with the various Python data structures and their uses before you started this chapter, you may have been bored. But if that is the case, there's a good chance you use data structures too much! Look at some of your old code and rewrite it to use more self-made objects. Carefully consider the alternatives and try them all out; which one makes for the most readable and maintainable system?

Always critically evaluate your code and design decisions. Make a habit of reviewing old code and take note if your understanding of "good design" has changed since you've written it. Software design has a large aesthetic component, and like artists with oil on canvas, we all have to find the style that suits us best.


Summary of
Module 1 Chapter 7
Ankita Thakur
Your Course Guide
We've covered several built-in data structures and attempted to understand how to choose one for specific applications. Sometimes, the best thing we can do is create a new class of objects, but often, one of the built-ins provides exactly what we need. When it doesn't, we can always use inheritance or composition to adapt them to our use cases. We can even override special methods to completely change the behavior of built-in syntaxes.
In the next chapter, we'll discuss how to integrate the object-oriented and not-so-object-oriented aspects of Python. Along the way, we'll discover that it's more object-oriented than it looks at first sight!


Your Progress through the Course So Far
Module 1
Python
Fundamentals
Data
Analysis
Data
Mining
Machine
Learning

8
Python Object-oriented Shortcuts

There are many aspects of Python that appear more reminiscent of structural or functional programming than object-oriented programming. Although object-oriented programming has been the most visible paradigm of the past two decades, the old models have seen a recent resurgence. As with Python's data structures, most of these tools are syntactic sugar over an underlying object-oriented implementation; we can think of them as a further abstraction layer built on top of the (already abstracted) object-oriented paradigm. In this chapter, we'll be covering a grab bag of Python features that are not strictly object-oriented:

- Built-in functions that take care of common tasks in one call
- File I/O and context managers
- An alternative to method overloading
- Functions as objects

Python built-in functions

There are numerous functions in Python that perform a task or calculate a result on certain types of objects without being methods on the underlying class. They usually abstract common calculations that apply to multiple types of classes. This is duck typing at its best; these functions accept objects that have certain attributes or methods, and are able to perform generic operations using those methods. Many, but not all, of these are special double underscore methods. We've used many of the built-in functions already, but let's quickly go through the important ones and pick up a few neat tricks along the way.

The len() function

The simplest example is the `len()` function, which counts the number of items in some kind of container object, such as a dictionary or list. You've seen it before:

```
>>> len([1,2,3,4])
4
```

Why don't these objects have a length property instead of having to call a function on them? Technically, they do. Most objects that `len()` will apply to have a method called `__len__()` that returns the same value. So `len(myobj)` seems to call `myobj.__len__()`.

Why should we use the `len()` function instead of the `__len__` method? Obviously `__len__` is a special double-underscore method, suggesting that we shouldn't call it directly. There must be an explanation for this. The Python developers don't make such design decisions lightly.

The main reason is efficiency. When we call `__len__` on an object, the object has to look the method up in its namespace, and, if the special `__getattribute__` method (which is called every time an attribute or method on an object is accessed) is defined on that object, it has to be called as well. Further, `__getattribute__` for that particular method may have been written to do something nasty, like refusing to give us access to special methods such as `__len__`! The `len()` function doesn't encounter any of this. It actually calls the `__len__` function on the underlying class, so `len(myobj)` maps to `MyObj.__len__(myobj)`.

Another reason is maintainability. In the future, the Python developers may want to change `len()` so that it can calculate the length of objects that don't have `__len__`, for example, by counting the number of items returned in an iterator. They'll only have to change one function instead of countless `__len__` methods across the board.

There is one other extremely important and often overlooked reason for `len()` being an external function: backwards compatibility. This is often cited in articles as "for historical reasons", which is a mildly dismissive phrase that an author will use to say something is the way it is because a mistake was made long ago and we're stuck with it. Strictly speaking, `len()` isn't a mistake, it's a design decision, but that decision was made in a less object-oriented time. It has stood the test of time and has some benefits, so do get used to it.

Reversed

The `reversed()` function takes any sequence as input, and returns a copy of that sequence in reverse order. It is normally used in `for` loops when we want to loop over items from back to front.

Similar to `len`, `reversed` calls the `__reversed__()` function on the class for the parameter. If that method does not exist, `reversed` builds the reversed sequence itself using calls to `__len__` and `__getitem__`, which are used to define a sequence. We only need to override `__reversed__` if we want to somehow customize or optimize the process:

```
normal_list=[1,2,3,4,5]

class CustomSequence():
    def __len__(self):
        return 5


    def __getitem__(self, index):
        return "x{0}".format(index)

class FunkyBackwards():

    def __reversed__(self):
        return "BACKWARDS!"

for seq in normal_list, CustomSequence(), FunkyBackwards():
    print("\n{}: ".format(seq.__class__.__name__), end="")
    for item in reversed(seq):
        print(item, end=", ")
```

The `for` loops at the end print the reversed versions of a normal list, and instances of the two custom sequences. The output shows that `reversed` works on all three of them, but has very different results when we define `__reversed__` ourselves:

```
list: 5, 4, 3, 2, 1,
CustomSequence: x4, x3, x2, x1, x0,
FunkyBackwards: B, A, C, K, W, A, R, D, S, !,
```

When we reverse `CustomSequence`, the `__getitem__` method is called for each item, which just inserts an `x` before the index. For `FunkyBackwards`, the `__reversed__` method returns a string, each character of which is output individually in the `for` loop.

The preceding two classes aren't very good sequences as they don't define a proper version of `__iter__`, so a forward `for` loop over them will never end.

Enumerate

Sometimes, when we're looping over a container in a `for` loop, we want access to the index (the current position in the list) of the current item being processed. The `for` loop doesn't provide us with indexes, but the `enumerate` function gives us something better: it creates a sequence of tuples, where the first object in each tuple is the index and the second is the original item.

This is useful if we need to use index numbers directly. Consider some simple code that outputs each of the lines in a file with line numbers:

```
import sys
filename = sys.argv[1]

with open(filename) as file:
    for index, line in enumerate(file):
        print("{0}: {1}".format(index+1, line), end='')
```

Running this code using it's own filename as the input file shows how it works:

```
1: import sys
2: filename = sys.argv[1]
3:
4: with open(filename) as file:
5:     for index, line in enumerate(file):
6:         print("{0}: {1}".format(index+1, line), end='')
```

The `enumerate` function returns a sequence of tuples, our `for` loop splits each tuple into two values, and the `print` statement formats them together. It adds one to the index for each line number, since `enumerate`, like all sequences, is zero-based.

We've only touched on a few of the more important Python built-in functions. As you can see, many of them call into object-oriented concepts, while others subscribe to purely functional or procedural paradigms. There are numerous others in the standard library; some of the more interesting ones include:

- `all` and `any`, which accept an iterable object and return `True` if all, or any, of the items evaluate to true (such as a nonempty string or list, a nonzero number, an object that is not `None`, or the literal `True`).
- `eval`, `exec`, and `compile`, which execute string as code inside the interpreter. Be careful with these ones; they are not safe, so don't execute code an unknown user has supplied to you (in general, assume all unknown users are malicious, foolish, or both).

- `hasattr`, `getattr`, `setattr`, and `delattr`, which allow attributes on an object to be manipulated by their string names.
- `zip`, which takes two or more sequences and returns a new sequence of tuples, where each tuple contains a single value from each sequence.
- And many more! See the interpreter help documentation for each of the functions listed in `dir(__builtins__)`.

File I/O

Our examples so far that touch the filesystem have operated entirely on text files without much thought to what is going on under the hood. Operating systems, however, actually represent files as a sequence of bytes, not text. Be aware that reading textual data from a file is a fairly involved process. Python, especially Python 3, takes care of most of this work for us behind the scenes. Aren't we lucky?

The concept of files has been around since long before anyone coined the term object-oriented programming. However, Python has wrapped the interface that operating systems provide in a sweet abstraction that allows us to work with file (or file-like, vis-á-vis duck typing) objects.

The `open()` built-in function is used to open a file and return a file object. For reading text from a file, we only need to pass the name of the file into the function. The file will be opened for reading, and the bytes will be converted to text using the platform default encoding.

Of course, we don't always want to read files; often we want to write data to them! To open a file for writing, we need to pass a `mode` argument as the second positional argument, with a value of `"w"`:

```
contents = "Some file contents"
file = open("filename", "w")
file.write(contents)
file.close()
```

We could also supply the value `"a"` as a mode argument, to append to the end of the file, rather than completely overwriting existing file contents.

These files with built-in wrappers for converting bytes to text are great, but it'd be awfully inconvenient if the file we wanted to open was an image, executable, or other binary file, wouldn't it?

To open a binary file, we modify the mode string to append `'b'`. So, `'wb'` would open a file for writing bytes, while `'rb'` allows us to read them. They will behave like text files, but without the automatic encoding of text to bytes. When we read such a file, it will return `bytes` objects instead of `str`, and when we write to it, it will fail if we try to pass a text object.

These mode strings for controlling how files are opened are rather cryptic and are neither pythonic nor object-oriented. However, they are consistent with virtually every other programming language out there. File I/O is one of the fundamental jobs an operating system has to handle, and all programming languages have to talk to the OS using the same system calls. Just be glad that Python returns a file object with useful methods instead of the integer that most major operating systems use to identify a file handle!

Once a file is opened for reading, we can call the `read`, `readline`, or `readlines` methods to get the contents of the file. The `read` method returns the entire contents of the file as a `str` or `bytes` object, depending on whether there is `'b'` in the mode. Be careful not to use this method without arguments on huge files. You don't want to find out what happens if you try to load that much data into memory!

It is also possible to read a fixed number of bytes from a file; we pass an integer argument to the `read` method describing how many bytes we want to read. The next call to `read` will load the next sequence of bytes, and so on. We can do this inside a `while` loop to read the entire file in manageable chunks.

The `readline` method returns a single line from the file (where each line ends in a newline, a carriage return, or both, depending on the operating system on which the file was created). We can call it repeatedly to get additional lines. The plural `readlines` method returns a list of all the lines in the file. Like the `read` method, it's not safe to use on very large files. These two methods even work when the file is open in `bytes` mode, but it only makes sense if we are parsing text-like data that has newlines at reasonable positions. An image or audio file, for example, will not have newline characters in it (unless the newline byte happened to represent a certain pixel or sound), so applying `readline` wouldn't make sense.

For readability, and to avoid reading a large file into memory at once, it is often better to use a `for` loop directly on a file object. For text files, it will read each line, one at a time, and we can process it inside the loop body. For binary files, it's better to read fixed-sized chunks of data using the `read()` method, passing a parameter for the maximum number of bytes to read.

Writing to a file is just as easy; the `write` method on file objects writes a string (or bytes, for binary data) object to the file. It can be called repeatedly to write multiple strings, one after the other. The `writelines` method accepts a sequence of strings and writes each of the iterated values to the file. The `writelines` method does *not* append a new line after each item in the sequence. It is basically a poorly named convenience function to write the contents of a sequence of strings without having to explicitly iterate over it using a `for` loop.

Lastly, and I do mean lastly, we come to the `close` method. This method should be called when we are finished reading or writing the file, to ensure any buffered writes are written to the disk, that the file has been properly cleaned up, and that all resources associated with the file are released back to the operating system. Technically, this will happen automatically when the script exits, but it's better to be explicit and clean up after ourselves, especially in long-running processes.

Placing it in context

The need to close files when we are finished with them can make our code quite ugly. Because an exception may occur at any time during file I/O, we ought to wrap all calls to a file in a `try...finally` clause. The file should be closed in the `finally` clause, regardless of whether I/O was successful. This isn't very Pythonic. Of course, there is a more elegant way to do it.

If we run `dir` on a file-like object, we see that it has two special methods named `__enter__` and `__exit__`. These methods turn the file object into what is known as a **context manager**. Basically, if we use a special syntax called the `with` statement, these methods will be called before and after nested code is executed. On file objects, the `__exit__` method ensures the file is closed, even if an exception is raised. We no longer have to explicitly manage the closing of the file. Here is what the `with` statement looks like in practice:

```
with open('filename') as file:
    for line in file:
        print(line, end='')
```

The `open` call returns a file object, which has `__enter__` and `__exit__` methods. The returned object is assigned to the variable named `file` by the `as` clause. We know the file will be closed when the code returns to the outer indentation level, and that this will happen even if an exception is raised.

The `with` statement is used in several places in the standard library where startup or cleanup code needs to be executed. For example, the `urlopen` call returns an object that can be used in a `with` statement to clean up the socket when we're done. Locks in the threading module can automatically release the lock when the statement has been executed.

Most interestingly, because the `with` statement can apply to any object that has the appropriate special methods, we can use it in our own frameworks. For example, remember that strings are immutable, but sometimes you need to build a string from multiple parts. For efficiency, this is usually done by storing the component strings in a list and joining them at the end. Let's create a simple context manager that allows us to construct a sequence of characters and automatically convert it to a string upon exit:

```
class StringJoiner(list):
    def __enter__(self):
        return self

    def __exit__(self, type, value, tb):
        self.result = "".join(self)
```

This code adds the two special methods required of a context manager to the `list` class it inherits from. The `__enter__` method performs any required setup code (in this case, there isn't any) and then returns the object that will be assigned to the variable after `as` in the `with` statement. Often, as we've done here, this is just the context manager object itself. The `__exit__` method accepts three arguments. In a normal situation, these are all given a value of `None`. However, if an exception occurs inside the `with` block, they will be set to values related to the type, value, and traceback for the exception. This allows the `__exit__` method to do any cleanup code that may be required, even if an exception occurred. In our example, we take the irresponsible path and create a result string by joining the characters in the string, regardless of whether an exception was thrown.

While this is one of the simplest context managers we could write, and its usefulness is dubious, it does work with a `with` statement. Have a look at it in action:

```
import random, string
with StringJoiner() as joiner:
    for i in range(15):
        joiner.append(random.choice(string.ascii_letters))

print(joiner.result)
```

This code constructs a string of 15 random characters. It appends these to a `StringJoiner` using the `append` method it inherited from `list`. When the `with` statement goes out of scope (back to the outer indentation level), the `__exit__` method is called, and the `result` attribute becomes available on the joiner object. We print this value to see a random string.

An alternative to method overloading

One prominent feature of many object-oriented programming languages is a tool called **method overloading**. Method overloading simply refers to having multiple methods with the same name that accept different sets of arguments. In statically typed languages, this is useful if we want to have a method that accepts either an integer or a string, for example. In non-object-oriented languages, we might need two functions, called `add_s` and `add_i`, to accommodate such situations. In statically typed object-oriented languages, we'd need two methods, both called `add`, one that accepts strings, and one that accepts integers.

In Python, we only need one method, which accepts any type of object. It may have to do some testing on the object type (for example, if it is a string, convert it to an integer), but only one method is required.

However, method overloading is also useful when we want a method with the same name to accept different numbers or sets of arguments. For example, an e-mail message method might come in two versions, one of which accepts an argument for the "from" e-mail address. The other method might look up a default "from" e-mail address instead. Python doesn't permit multiple methods with the same name, but it does provide a different, equally flexible, interface.

We've seen some of the possible ways to send arguments to methods and functions in previous examples, but now we'll cover all the details. The simplest function accepts no arguments. We probably don't need an example, but here's one for completeness:

```
def no_args():
    pass
```

Here's how it's called:

```
no_args()
```

A function that does accept arguments will provide the names of those arguments in a comma-separated list. Only the name of each argument needs to be supplied.

When calling the function, these positional arguments must be specified in order, and none can be missed or skipped. This is the most common way we've specified arguments in our previous examples:

```
def mandatory_args(x, y, z):
    pass
```

To call it:

```
mandatory_args("a string", a_variable, 5)
```

Any type of object can be passed as an argument: an object, a container, a primitive, even functions and classes. The preceding call shows a hardcoded string, an unknown variable, and an integer passed into the function.

Default arguments

If we want to make an argument optional, rather than creating a second method with a different set of arguments, we can specify a default value in a single method, using an equals sign. If the calling code does not supply this argument, it will be assigned a default value. However, the calling code can still choose to override the default by passing in a different value. Often, a default value of `None`, or an empty string or list is suitable.

Here's a function definition with default arguments:

```
def default_arguments(x, y, z, a="Some String", b=False):
    pass
```

The first three arguments are still mandatory and must be passed by the calling code. The last two parameters have default arguments supplied.

There are several ways we can call this function. We can supply all arguments in order as though all the arguments were positional arguments:

```
default_arguments("a string", variable, 8, "", True)
```

Alternatively, we can supply just the mandatory arguments in order, leaving the keyword arguments to be assigned their default values:

```
default_arguments("a longer string", some_variable, 14)
```

We can also use the equals sign syntax when calling a function to provide values in a different order, or to skip default values that we aren't interested in. For example, we can skip the first keyword arguments and supply the second one:

```
default_arguments("a string", variable, 14, b=True)
```

Surprisingly, we can even use the equals sign syntax to mix up the order of positional arguments, so long as all of them are supplied:

```
>>> default_arguments(y=1,z=2,x=3,a="hi")
3 1 2 hi False
```

With so many options, it may seem hard to pick one, but if you think of the positional arguments as an ordered list, and keyword arguments as sort of like a dictionary, you'll find that the correct layout tends to fall into place. If you need to require the caller to specify an argument, make it mandatory; if you have a sensible default, then make it a keyword argument. Choosing how to call the method normally takes care of itself, depending on which values need to be supplied, and which can be left at their defaults.

One thing to take note of with keyword arguments is that anything we provide as a default argument is evaluated when the function is first interpreted, not when it is called. This means we can't have dynamically generated default values. For example, the following code won't behave quite as expected:

```
number = 5
def funky_function(number=number):
    print(number)

number=6
funky_function(8)
funky_function()
print(number)
```

If we run this code, it outputs the number 8 first, but then it outputs the number 5 for the call with no arguments. We had set the variable to the number 6, as evidenced by the last line of output, but when the function is called, the number 5 is printed; the default value was calculated when the function was defined, not when it was called.

This is tricky with empty containers such as lists, sets, and dictionaries. For example, it is common to ask calling code to supply a list that our function is going to manipulate, but the list is optional. We'd like to make an empty list as a default argument. We can't do this; it will create only one list, when the code is first constructed:

```
>>> def hello(b=[]):
...     b.append('a')
...     print(b)
...
```

```
>>> hello()
['a']
>>> hello()
['a', 'a']
```

Whoops, that's not quite what we expected! The usual way to get around this is to make the default value `None`, and then use the idiom `iargument = argument if argument else []` inside the method. Pay close attention!

Variable argument lists

Default values alone do not allow us all the flexible benefits of method overloading. The thing that makes Python really slick is the ability to write methods that accept an arbitrary number of positional or keyword arguments without explicitly naming them. We can also pass arbitrary lists and dictionaries into such functions.

For example, a function to accept a link or list of links and download the web pages could use such variadic arguments, or **varargs**. Instead of accepting a single value that is expected to be a list of links, we can accept an arbitrary number of arguments, where each argument is a different link. We do this by specifying the `*` operator in the function definition:

```
def get_pages(*links):
    for link in links:
        #download the link with urllib
        print(link)
```

The `*links` parameter says "I'll accept any number of arguments and put them all in a list named `links`". If we supply only one argument, it'll be a list with one element; if we supply no arguments, it'll be an empty list. Thus, all these function calls are valid:

```
get_pages()
get_pages('http://www.archlinux.org')
get_pages('http://www.archlinux.org',
        'http://ccphillips.net/')
```

We can also accept arbitrary keyword arguments. These arrive into the function as a dictionary. They are specified with two asterisks (as in `**kwargs`) in the function declaration. This tool is commonly used in configuration setups. The following class allows us to specify a set of options with default values:

```
class Options:
    default_options = {
```

```
            'port': 21,
            'host': 'localhost',
            'username': None,
            'password': None,
            'debug': False,
            }
    def __init__(self, **kwargs):
        self.options = dict(Options.default_options)
        self.options.update(kwargs)

    def __getitem__(self, key):
        return self.options[key]
```

All the interesting stuff in this class happens in the `__init__` method. We have a dictionary of default options and values at the class level. The first thing the `__init__` method does is make a copy of this dictionary. We do that instead of modifying the dictionary directly in case we instantiate two separate sets of options. (Remember, class-level variables are shared between instances of the class.) Then, `__init__` uses the `update` method on the new dictionary to change any non-default values to those supplied as keyword arguments. The `__getitem__` method simply allows us to use the new class using indexing syntax. Here's a session demonstrating the class in action:

```
>>> options = Options(username="dusty", password="drowssap",
        debug=True)
>>> options['debug']
True
>>> options['port']
21
>>> options['username']
'dusty'
```

We're able to access our options instance using dictionary indexing syntax, and the dictionary includes both default values and the ones we set using keyword arguments.

The keyword argument syntax can be dangerous, as it may break the "explicit is better than implicit" rule. In the preceding example, it's possible to pass arbitrary keyword arguments to the `Options` initializer to represent options that don't exist in the default dictionary. This may not be a bad thing, depending on the purpose of the class, but it makes it hard for someone using the class to discover what valid options are available. It also makes it easy to enter a confusing typo ("Debug" instead of "debug", for example) that adds two options where only one should have existed.

Keyword arguments are also very useful when we need to accept arbitrary arguments to pass to a second function, but we don't know what those arguments will be. We can, of course, combine the variable argument and variable keyword argument syntax in one function call, and we can use normal positional and default arguments as well. The following example is somewhat contrived, but demonstrates the four types in action:

```
import shutil
import os.path
def augmented_move(target_folder, *filenames,
        verbose=False, **specific):
    '''Move all filenames into the target_folder, allowing
    specific treatment of certain files.'''

    def print_verbose(message, filename):
        '''print the message only if verbose is enabled'''
        if verbose:
            print(message.format(filename))

    for filename in filenames:
        target_path = os.path.join(target_folder, filename)
        if filename in specific:
            if specific[filename] == 'ignore':
                print_verbose("Ignoring {0}", filename)
            elif specific[filename] == 'copy':
                print_verbose("Copying {0}", filename)
                shutil.copyfile(filename, target_path)
        else:
            print_verbose("Moving {0}", filename)
            shutil.move(filename, target_path)
```

This example will process an arbitrary list of files. The first argument is a target folder, and the default behavior is to move all remaining non-keyword argument files into that folder. Then there is a keyword-only argument, `verbose`, which tells us whether to print information on each file processed. Finally, we can supply a dictionary containing actions to perform on specific filenames; the default behavior is to move the file, but if a valid string action has been specified in the keyword arguments, it can be ignored or copied instead. Notice the ordering of the parameters in the function; first the positional argument is specified, then the `*filenames` list, then any specific keyword-only arguments, and finally, a `**specific` dictionary to hold remaining keyword arguments.

We create an inner helper function, `print_verbose`, which will print messages only if the `verbose` key has been set. This function keeps code readable by encapsulating this functionality into a single location.

In common cases, assuming the files in question exist, this function could be called as:

```
>>> augmented_move("move_here", "one", "two")
```

This command would move the files `one` and `two` into the `move_here` directory, assuming they exist (there's no error checking or exception handling in the function, so it would fail spectacularly if the files or target directory didn't exist). The move would occur without any output, since `verbose` is `False` by default.

If we want to see the output, we can call it with:

```
>>> augmented_move("move_here", "three", verbose=True)
Moving three
```

This moves one file named `three`, and tells us what it's doing. Notice that it is impossible to specify `verbose` as a positional argument in this example; we must pass a keyword argument. Otherwise, Python would think it was another filename in the `*filenames` list.

If we want to copy or ignore some of the files in the list, instead of moving them, we can pass additional keyword arguments:

```
>>> augmented_move("move_here", "four", "five", "six",
        four="copy", five="ignore")
```

This will move the sixth file and copy the fourth, but won't display any output, since we didn't specify `verbose`. Of course, we can do that too, and keyword arguments can be supplied in any order:

```
>>> augmented_move("move_here", "seven", "eight", "nine",
        seven="copy", verbose=True, eight="ignore")
Copying seven
Ignoring eight
Moving nine
```

Unpacking arguments

There's one more nifty trick involving variable arguments and keyword arguments. We've used it in some of our previous examples, but it's never too late for an explanation. Given a list or dictionary of values, we can pass those values into a function as if they were normal positional or keyword arguments. Have a look at this code:

```
def show_args(arg1, arg2, arg3="THREE"):
    print(arg1, arg2, arg3)

some_args = range(3)
more_args = {
        "arg1": "ONE",
        "arg2": "TWO"}

print("Unpacking a sequence:", end=" ")

show_args(*some_args)
print("Unpacking a dict:", end=" ")

show_args(**more_args)
```

Here's what it looks like when we run it:

```
Unpacking a sequence: 0 1 2
Unpacking a dict: ONE TWO THREE
```

The function accepts three arguments, one of which has a default value. But when we have a list of three arguments, we can use the `*` operator inside a function call to unpack it into the three arguments. If we have a dictionary of arguments, we can use the `**` syntax to unpack it as a collection of keyword arguments.

This is most often useful when mapping information that has been collected from user input or from an outside source (for example, an Internet page or a text file) to a function or method call.

Remember our earlier example that used headers and lines in a text file to create a list of dictionaries with contact information? Instead of just adding the dictionaries to a list, we could use keyword unpacking to pass the arguments to the `__init__` method on a specially built `Contact` object that accepts the same set of arguments. See if you can adapt the example to make this work.

Functions are objects too

Programming languages that overemphasize object-oriented principles tend to frown on functions that are not methods. In such languages, you're expected to create an object to sort of wrap the single method involved. There are numerous situations where we'd like to pass around a small object that is simply called to perform an action. This is most frequently done in event-driven programming, such as graphical toolkits or asynchronous servers.

In Python, we don't need to wrap such methods in an object, because functions already are objects! We can set attributes on functions (though this isn't a common activity), and we can pass them around to be called at a later date. They even have a few special properties that can be accessed directly. Here's yet another contrived example:

```
def my_function():
    print("The Function Was Called")
my_function.description = "A silly function"

def second_function():
    print("The second was called")
second_function.description = "A sillier function."

def another_function(function):
    print("The description:", end=" ")
    print(function.description)
    print("The name:", end=" ")
    print(function.__name__)
    print("The class:", end=" ")
    print(function.__class__)
    print("Now I'll call the function passed in")
    function()

another_function(my_function)
another_function(second_function)
```

If we run this code, we can see that we were able to pass two different functions into our third function, and get different output for each one:

```
The description: A silly function
The name: my_function
The class: <class 'function'>
Now I'll call the function passed in
```

```
The Function Was Called
The description: A sillier function.
The name: second_function
The class: <class 'function'>
Now I'll call the function passed in
The second was called
```

We set an attribute on the function, named `description` (not very good descriptions, admittedly). We were also able to see the function's `__name__` attribute, and to access its class, demonstrating that the function really is an object with attributes. Then we called the function by using the callable syntax (the parentheses).

The fact that functions are top-level objects is most often used to pass them around to be executed at a later date, for example, when a certain condition has been satisfied. Let's build an event-driven timer that does just this:

```
import datetime
import time

class TimedEvent:
    def __init__(self, endtime, callback):
        self.endtime = endtime
        self.callback = callback

    def ready(self):
        return self.endtime <= datetime.datetime.now()

class Timer:
    def __init__(self):
        self.events = []

    def call_after(self, delay, callback):
        end_time = datetime.datetime.now() + \
                datetime.timedelta(seconds=delay)

        self.events.append(TimedEvent(end_time, callback))

    def run(self):
        while True:
            ready_events = (e for e in self.events if e.ready())
            for event in ready_events:
                event.callback(self)
                self.events.remove(event)
            time.sleep(0.5)
```

In production, this code should definitely have extra documentation using docstrings! The `call_after` method should at least mention that the `delay` parameter is in seconds, and that the `callback` function should accept one argument: the timer doing the calling.

We have two classes here. The `TimedEvent` class is not really meant to be accessed by other classes; all it does is store `endtime` and `callback`. We could even use a `tuple` or `namedtuple` here, but as it is convenient to give the object a behavior that tells us whether or not the event is ready to run, we use a class instead.

The `Timer` class simply stores a list of upcoming events. It has a `call_after` method to add a new event. This method accepts a `delay` parameter representing the number of seconds to wait before executing the callback, and the `callback` function itself: a function to be executed at the correct time. This `callback` function should accept one argument.

The `run` method is very simple; it uses a generator expression to filter out any events whose time has come, and executes them in order. The timer loop then continues indefinitely, so it has to be interrupted with a keyboard interrupt (*Ctrl + C* or *Ctrl + Break*). We sleep for half a second after each iteration so as to not grind the system to a halt.

The important things to note here are the lines that touch callback functions. The function is passed around like any other object and the timer never knows or cares what the original name of the function is or where it was defined. When it's time to call the function, the timer simply applies the parenthesis syntax to the stored variable.

Here's a set of callbacks that test the timer:

```
from timer import Timer
import datetime

def format_time(message, *args):
    now = datetime.datetime.now().strftime("%I:%M:%S")
    print(message.format(*args, now=now))

def one(timer):
    format_time("{now}: Called One")

def two(timer):
    format_time("{now}: Called Two")

def three(timer):
```

```
        format_time("{now}: Called Three")

    class Repeater:
        def __init__(self):
            self.count = 0
        def repeater(self, timer):
            format_time("{now}: repeat {0}", self.count)
            self.count += 1
            timer.call_after(5, self.repeater)

    timer = Timer()
    timer.call_after(1, one)
    timer.call_after(2, one)
    timer.call_after(2, two)
    timer.call_after(4, two)
    timer.call_after(3, three)
    timer.call_after(6, three)
    repeater = Repeater()
    timer.call_after(5, repeater.repeater)
    format_time("{now}: Starting")
    timer.run()
```

This example allows us to see how multiple callbacks interact with the timer. The first function is the `format_time` function. It uses the string `format` method to add the current time to the message, and illustrates variable arguments in action. The `format_time` method will accept any number of positional arguments, using variable argument syntax, which are then forwarded as positional arguments to the string's `format` method. After this, we create three simple callback methods that simply output the current time and a short message telling us which callback has been fired.

The `Repeater` class demonstrates that methods can be used as callbacks too, since they are really just functions. It also shows why the `timer` argument to the callback functions is useful: we can add a new timed event to the timer from inside a presently running callback. We then create a timer and add several events to it that are called after different amounts of time. Finally, we start the timer running; the output shows that events are run in the expected order:

```
02:53:35: Starting
02:53:36: Called One
02:53:37: Called One
02:53:37: Called Two
02:53:38: Called Three
02:53:39: Called Two
02:53:40: repeat 0
```

```
02:53:41: Called Three
02:53:45: repeat 1
02:53:50: repeat 2
02:53:55: repeat 3
02:54:00: repeat 4
```

Python 3.4 introduces a generic event-loop architecture similar to this.

Using functions as attributes

One of the interesting effects of functions being objects is that they can be set as callable attributes on other objects. It is possible to add or change a function to an instantiated object:

```
class A:
    def print(self):
        print("my class is A")

def fake_print():
    print("my class is not A")

a = A()
a.print()
a.print = fake_print
a.print()
```

This code creates a very simple class with a `print` method that doesn't tell us anything we didn't know. Then we create a new function that tells us something we don't believe.

When we call `print` on an instance of the `A` class, it behaves as expected. If we then set the `print` method to point at a new function, it tells us something different:

```
my class is A
my class is not A
```

It is also possible to replace methods on classes instead of objects, although in that case we have to add the `self` argument to the parameter list. This will change the method for all instances of that object, even ones that have already been instantiated. Obviously, replacing methods like this can be both dangerous and confusing to maintain. Somebody reading the code will see that a method has been called and look up that method on the original class. But the method on the original class is not the one that was called. Figuring out what really happened can become a tricky, frustrating debugging session.

It does have its uses though. Often, replacing or adding methods at run time (called **monkey-patching**) is used in automated testing. If testing a client-server application, we may not want to actually connect to the server while testing the client; this may result in accidental transfers of funds or embarrassing test e-mails being sent to real people. Instead, we can set up our test code to replace some of the key methods on the object that sends requests to the server, so it only records that the methods have been called.

Monkey-patching can also be used to fix bugs or add features in third-party code that we are interacting with, and does not behave quite the way we need it to. It should, however, be applied sparingly; it's almost always a "messy hack". Sometimes, though, it is the only way to adapt an existing library to suit our needs.

Callable objects

Just as functions are objects that can have attributes set on them, it is possible to create an object that can be called as though it were a function.

Any object can be made callable by simply giving it a `__call__` method that accepts the required arguments. Let's make our `Repeater` class, from the timer example, a little easier to use by making it a callable:

```
class Repeater:
    def __init__(self):
        self.count = 0


    def __call__(self, timer):
        format_time("{now}: repeat {0}", self.count)
        self.count += 1

        timer.call_after(5, self)

timer = Timer()

timer.call_after(5, Repeater())
format_time("{now}: Starting")
timer.run()
```

This example isn't much different from the earlier class; all we did was change the name of the `repeater` function to `__call__` and pass the object itself as a callable. Note that when we make the `call_after` call, we pass the argument `Repeater()`. Those two parentheses are creating a new instance of the class; they are not explicitly calling the class. This happens later, inside the timer. If we want to execute the `__call__` method on a newly instantiated object, we'd use a rather odd syntax: `Repeater()()`. The first set of parentheses constructs the object; the second set executes the `__call__` method. If we find ourselves doing this, we may not be using the correct abstraction. Only implement the `__call__` function on an object if the object is meant to be treated like a function.

Case study

To tie together some of the principles presented in this chapter, let's build a mailing list manager. The manager will keep track of e-mail addresses categorized into named groups. When it's time to send a message, we can pick a group and send the message to all e-mail addresses assigned to that group.

Now, before we start working on this project, we ought to have a safe way to test it, without sending e-mails to a bunch of real people. Luckily, Python has our back here; like the test HTTP server, it has a built-in **Simple Mail Transfer Protocol** (**SMTP**) server that we can instruct to capture any messages we send without actually sending them. We can run the server with the following command:

```
python -m smtpd -n -c DebuggingServer localhost:1025
```

Running this command at a command prompt will start an SMTP server running on port 1025 on the local machine. But we've instructed it to use the `DebuggingServer` class (it comes with the built-in SMTP module), which, instead of sending mails to the intended recipients, simply prints them on the terminal screen as it receives them. Neat, eh?

Now, before writing our mailing list, let's write some code that actually sends mail. Of course, Python supports this in the standard library, too, but it's a bit of an odd interface, so we'll write a new function to wrap it all cleanly:

```
import smtplib
from email.mime.text import MIMEText

def send_email(subject, message, from_addr, *to_addrs,
```

```
            host="localhost", port=1025, **headers):

        email = MIMEText(message)
        email['Subject'] = subject
        email['From'] = from_addr
        for header, value in headers.items():
            email[header] = value

        sender = smtplib.SMTP(host, port)
        for addr in to_addrs:
            del email['To']
            email['To'] = addr
            sender.sendmail(from_addr, addr, email.as_string())
        sender.quit()
```

We won't cover the code inside this method too thoroughly; the documentation in the standard library can give you all the information you need to use the `smtplib` and `email` modules effectively.

We've used both variable argument and keyword argument syntax in the function call. The variable argument list allows us to supply a single string in the default case of having a single `to` address, as well as permitting multiple addresses to be supplied if required. Any extra keyword arguments are mapped to e-mail headers. This is an exciting use of variable arguments and keyword arguments, but it's not really a great interface for the person calling the function. In fact, it makes many things the programmer will want to do impossible.

The headers passed into the function represent auxiliary headers that can be attached to a method. Such headers might include `Reply-To`, `Return-Path`, or *X-pretty-much-anything*. But in order to be a valid identifier in Python, a name cannot include the `-` character. In general, that character represents subtraction. So, it's not possible to call a function with `Reply-To = my@email.com`. It appears we were too eager to use keyword arguments because they are a new tool we just learned about in this chapter.

We'll have to change the argument to a normal dictionary; this will work because any string can be used as a key in a dictionary. By default, we'd want this dictionary to be empty, but we can't make the default parameter an empty dictionary. So, we'll have to make the default argument `None`, and then set up the dictionary at the beginning of the method:

```
def send_email(subject, message, from_addr, *to_addrs,
```

```
            host="localhost", port=1025, headers=None):

        headers = {} if headers is None else headers
```

If we have our debugging SMTP server running in one terminal, we can test this code in a Python interpreter:

```
>>> send_email("A model subject", "The message contents",
 "from@example.com", "to1@example.com", "to2@example.com")
```

Then, if we check the output from the debugging SMTP server, we get the following:

```
---------- MESSAGE FOLLOWS ----------
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: A model subject
From: from@example.com
To: to1@example.com
X-Peer: 127.0.0.1

The message contents
------------ END MESSAGE ------------
---------- MESSAGE FOLLOWS ----------
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: A model subject
From: from@example.com
To: to2@example.com
X-Peer: 127.0.0.1

The message contents
------------ END MESSAGE ------------
```

Excellent, it has "sent" our e-mail to the two expected addresses with subject and message contents included. Now that we can send messages, let's work on the e-mail group management system. We'll need an object that somehow matches e-mail addresses with the groups they are in. Since this is a many-to-many relationship (any one e-mail address can be in multiple groups; any one group can be associated with multiple e-mail addresses), none of the data structures we've studied seems quite ideal. We could try a dictionary of group-names matched to a list of associated e-mail addresses, but that would duplicate e-mail addresses. We could also try a dictionary of e-mail addresses matched to groups, resulting in a duplication of groups. Neither seems optimal. Let's try this latter version, even though intuition tells me the groups to e-mail address solution would be more straightforward.

Since the values in our dictionary will always be collections of unique e-mail addresses, we should probably store them in a `set` container. We can use `defaultdict` to ensure that there is always a `set` container available for each key:

```
from collections import defaultdict
class MailingList:
    '''Manage groups of e-mail addresses for sending e-mails.'''
    def __init__(self):
        self.email_map = defaultdict(set)

    def add_to_group(self, email, group):
        self.email_map[email].add(group)
```

Now, let's add a method that allows us to collect all the e-mail addresses in one or more groups. This can be done by converting the list of groups to a set:

```
    def emails_in_groups(self, *groups):
        groups = set(groups)
        emails = set()
        for e, g in self.email_map.items():
            if g & groups:
                emails.add(e)
        return emails
```

First, look at what we're iterating over: `self.email_map.items()`. This method, of course, returns a tuple of key-value pairs for each item in the dictionary. The values are sets of strings representing the groups. We split these into two variables named `e` and `g`, short for e-mail and groups. We add the e-mail address to the set of return values only if the passed in groups intersect with the e-mail address groups. The `g & groups` syntax is a shortcut for `g.intersection(groups)`; the `set` class does this by implementing the special `__and__` method to call `intersection`.

Now, with these building blocks, we can trivially add a method to our `MailingList` class that sends messages to specific groups:

```
    def send_mailing(self, subject, message, from_addr,
            *groups, headers=None):
        emails = self.emails_in_groups(*groups)
        send_email(subject, message, from_addr,
                *emails, headers=headers)
```

This function relies on variable argument lists. As input, it takes a list of groups as variable arguments. It gets the list of e-mails for the specified groups and passes those as variable arguments into `send_email`, along with other arguments that were passed into this method.

The program can be tested by ensuring the SMTP debugging server is running in one command prompt, and, in a second prompt, loading the code using:

```
python -i mailing_list.py
```

Create a `MailingList` object with:

```
>>> m = MailingList()
```

Then create a few fake e-mail addresses and groups, along the lines of:

```
>>> m.add_to_group("friend1@example.com", "friends")
>>> m.add_to_group("friend2@example.com", "friends")
>>> m.add_to_group("family1@example.com", "family")
>>> m.add_to_group("pro1@example.com", "professional")
```

Finally, use a command like this to send e-mails to specific groups:

```
>>> m.send_mailing("A Party",
"Friends and family only: a party", "me@example.com", "friends",
"family", headers={"Reply-To": "me2@example.com"})
```

E-mails to each of the addresses in the specified groups should show up in the console on the SMTP server.

The mailing list works fine as it is, but it's kind of useless; as soon as we exit the program, our database of information is lost. Let's modify it to add a couple of methods to load and save the list of e-mail groups from and to a file.

In general, when storing structured data on disk, it is a good idea to put a lot of thought into how it is stored. One of the reasons myriad database systems exist is that if someone else has put this thought into how data is stored, you don't have to. For this example, let's keep it simple and go with the first solution that could possibly work.

The data format I have in mind is to store each e-mail address followed by a space, followed by a comma-separated list of groups. This format seems reasonable, and we're going to go with it because data formatting isn't the topic of this chapter. However, to illustrate just why you need to think hard about how you format data on disk, let's highlight a few problems with the format.

First, the space character is technically legal in e-mail addresses. Most e-mail providers prohibit it (with good reason), but the specification defining e-mail addresses says an e-mail can contain a space if it is in quotation marks. If we are to use a space as a sentinel in our data format, we should technically be able to differentiate between that space and a space that is part of an e-mail. We're going to pretend this isn't true, for simplicity's sake, but real-life data encoding is full of stupid issues like this. Second, consider the comma-separated list of groups. What happens if someone decides to put a comma in a group name? If we decide to make commas illegal in group names, we should add validation to ensure this to our `add_to_group` method. For pedagogical clarity, we'll ignore this problem too. Finally, there are many security implications we need to consider: can someone get themselves into the wrong group by putting a fake comma in their e-mail address? What does the parser do if it encounters an invalid file?

The takeaway from this discussion is to try to use a data-storage method that has been field tested, rather than designing your own data serialization protocol. There are a ton of bizarre edge cases you might overlook, and it's better to use code that has already encountered and fixed those edge cases.

But forget that, let's just write some basic code that uses an unhealthy dose of wishful thinking to pretend this simple data format is safe:

```
email1@mydomain.com group1,group2
email2@mydomain.com group2,group3
```

The code to do this is as follows:

```
        def save(self):
            with open(self.data_file, 'w') as file:
                for email, groups in self.email_map.items():
                    file.write(
```

```
                    '{} {}\n'.format(email, ','.join(groups))
                )

    def load(self):
        self.email_map = defaultdict(set)
        try:
            with open(self.data_file) as file:
                for line in file:
                    email, groups = line.strip().split(' ')
                    groups = set(groups.split(','))
                    self.email_map[email] = groups
        except IOError:
            pass
```

In the `save` method, we open the file in a context manager and write the file as a formatted string. Remember the newline character; Python doesn't add that for us. The `load` method first resets the dictionary (in case it contains data from a previous call to `load`) uses the `for...in` syntax, which loops over each line in the file. Again, the newline character is included in the line variable, so we have to call `.strip()` to take it off.

Before using these methods, we need to make sure the object has a `self.data_file` attribute, which can be done by modifying `__init__`:

```
    def __init__(self, data_file):
        self.data_file = data_file
        self.email_map = defaultdict(set)
```

We can test these two methods in the interpreter as follows:

```
>>> m = MailingList('addresses.db')
>>> m.add_to_group('friend1@example.com', 'friends')
>>> m.add_to_group('family1@example.com', 'friends')
>>> m.add_to_group('family1@example.com', 'family')
>>> m.save()
```

The resulting `addresses.db` file contains the following lines, as expected:

```
friend1@example.com friends
family1@example.com friends,family
```

We can also load this data back into a `MailingList` object successfully:

```
>>> m = MailingList('addresses.db')
>>> m.email_map
defaultdict(<class 'set'>, {})
>>> m.load()
>>> m.email_map
defaultdict(<class 'set'>, {'friend2@example.com': {'friends\n'},
'family1@example.com': {'family\n'}, 'friend1@example.com':
{'friends\n'}})
```

As you can see, I forgot to do the `load` command, and it might be easy to forget the `save` command as well. To make this a little easier for anyone who wants to use our `MailingList` API in their own code, let's provide the methods to support a context manager:

```
    def __enter__(self):
        self.load()
        return self

    def __exit__(self, type, value, tb):
        self.save()
```

These simple methods just delegate their work to load and save, but we can now write code like this in the interactive interpreter and know that all the previously stored addresses were loaded on our behalf, and that the whole list will be saved to the file when we are done:

```
>>> with MailingList('addresses.db') as ml:
...     ml.add_to_group('friend2@example.com', 'friends')
...     ml.send_mailing("What's up", "hey friends, how's it going", 'me@
example.com', 'friends')
```

Your Coding Challenge

If you haven't encountered the with statements and context managers before, I encourage you, as usual, to go through your old code and find all the places you were opening files, and make sure they are safely closed using the with statement. Look for places that you could write your own context managers as well. Ugly or repetitive try...finally clauses are a good place to start, but you may find them useful any time you need to do before and/or after tasks in context.

You've probably used many of the basic built-in functions before now. We covered several of them, but didn't go into a great deal of detail. Play with enumerate, zip, reversed, any and all, until you know you'll remember to use them when they are the right tool for the job. The enumerate function is especially important; because not using it results in some pretty ugly code.

Ankita Thakur

Your Course Guide

Also explore some applications that pass functions around as callable objects, as well as using the __call__ method to make your own objects callable. You can get the same effect by attaching attributes to functions or by creating a __call__ method on an object. In which case would you use one syntax, and when would it be more suitable to use the other?

Our mailing list object could overwhelm an e-mail server if there is a massive number of e-mails to be sent out. Try refactoring it so that you can use different send_email functions for different purposes. One such function could be the version we used here. A different version might put the e-mails in a queue to be sent by a server in a different thread or process. A third version could just output the data to the terminal, obviating the need for a dummy SMTP server. Can you construct the mailing list with a callback such that the send_mailing function uses whatever is passed in? It would default to the current version if no callback is supplied.

The relationship between arguments, keyword arguments, variable arguments, and variable keyword arguments can be a bit confusing. We saw how painfully they can interact when we covered multiple inheritance. Devise some other examples to see how they can work well together, as well as to understand when they don't.

Summary of Module 1 Chapter 8

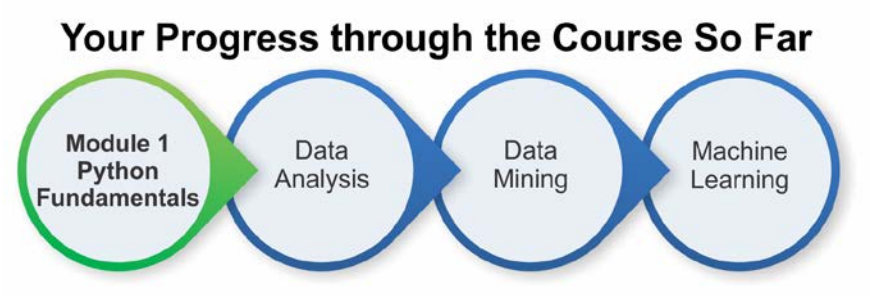
We covered a grab bag of topics in this chapter. Each represented an important non-object-oriented feature that is popular in Python. Just because we can use object-oriented principles does not always mean we should!

Ankita Thakur

Your Course Guide

However, we also saw that Python typically implements such features by providing a syntax shortcut to traditional object-oriented syntax. Knowing the object-oriented principles underlying these tools allows us to use them more effectively in our own classes.

We discussed a series of built-in functions and file I/O operations. There are a whole bunch of different syntaxes available to us when calling functions with arguments, keyword arguments, and variable argument lists. Context managers are useful for the common pattern of sandwiching a piece of code between two method calls. Even functions are objects, and, conversely, any normal object can be made callable.
In the next chapter, we'll learn more about string and file manipulation, and even spend some time with one of the least object-oriented topics in the standard library: regular expressions.



9
Strings and Serialization

Before we get involved with higher level design patterns, let's take a deep dive into one of Python's most common objects: the string. We'll see that there is a lot more to the string than meets the eye, and also cover searching strings for patterns and serializing data for storage or transmission.

In particular, we'll visit:

- The complexities of strings, bytes, and byte arrays
- The ins and outs of string formatting
- A few ways to serialize data
- The mysterious regular expression

Strings

Strings are a basic primitive in Python; we've used them in nearly every example we've discussed so far. All they do is represent an immutable sequence of characters. However, though you may not have considered it before, "character" is a bit of an ambiguous word; can Python strings represent sequences of accented characters? Chinese characters? What about Greek, Cyrillic, or Farsi?

In Python 3, the answer is yes. Python strings are all represented in Unicode, a character definition standard that can represent virtually any character in any language on the planet (and some made-up languages and random characters as well). This is done seamlessly, for the most part. So, let's think of Python 3 strings as an immutable sequence of Unicode characters. So what can we do with this immutable sequence? We've touched on many of the ways strings can be manipulated in previous examples, but let's quickly cover it all in one place: a crash course in string theory!

String manipulation

As you know, strings can be created in Python by wrapping a sequence of characters in single or double quotes. Multiline strings can easily be created using three quote characters, and multiple hardcoded strings can be concatenated together by placing them side by side. Here are some examples:

```
a = "hello"
b = 'world'
c = '''a multiple
line string'''
d = """More
multiple"""
e = ("Three " "Strings "
        "Together")
```

That last string is automatically composed into a single string by the interpreter. It is also possible to concatenate strings using the `+` operator (as in `"hello " + "world"`). Of course, strings don't have to be hardcoded. They can also come from various outside sources such as text files, user input, or encoded on the network.

The automatic concatenation of adjacent strings can make for some hilarious bugs when a comma is missed. It is, however, extremely useful when a long string needs to be placed inside a function call without exceeding the 79 character line-length limit suggested by the Python style guide.

Like other sequences, strings can be iterated over (character by character), indexed, sliced, or concatenated. The syntax is the same as for lists.

The `str` class has numerous methods on it to make manipulating strings easier. The `dir` and `help` commands in the Python interpreter can tell us how to use all of them; we'll consider some of the more common ones directly.

Several Boolean convenience methods help us identify whether or not the characters in a string match a certain pattern. Here is a summary of these methods. Most of these, such as `isalpha`, `isupper`/`islower`, and `startswith`/`endswith` have obvious interpretations. The `isspace` method is also fairly obvious, but remember that all whitespace characters (including tab, newline) are considered, not just the space character.

The `istitle` method returns `True` if the first character of each word is capitalized and all other characters are lowercase. Note that it does not strictly enforce the English grammatical definition of title formatting. For example, Leigh Hunt's poem "The Glove and the Lions" should be a valid title, even though not all words are capitalized. Robert Service's "The Cremation of Sam McGee" should also be a valid title, even though there is an uppercase letter in the middle of the last word.

Be careful with the `isdigit`, `isdecimal`, and `isnumeric` methods, as they are more nuanced than you would expect. Many Unicode characters are considered numbers besides the ten digits we are used to. Worse, the period character that we use to construct floats from strings is not considered a decimal character, so `'45.2'.isdecimal()` returns `False`. The real decimal character is represented by Unicode value 0660, as in 45.2, (or `45\u06602`). Further, these methods do not verify whether the strings are valid numbers; "127.0.0.1" returns `True` for all three methods. We might think we should use that decimal character instead of a period for all numeric quantities, but passing that character into the `float()` or `int()` constructor converts that decimal character to a zero:

```
>>> float('45\u06602')
4502.0
```

Other methods useful for pattern matching do not return Booleans. The `count` method tells us how many times a given substring shows up in the string, while `find`, `index`, `rfind`, and `rindex` tell us the position of a given substring within the original string. The two 'r' (for 'right' or 'reverse') methods start searching from the end of the string. The `find` methods return `-1` if the substring can't be found, while `index` raises a `ValueError` in this situation. Have a look at some of these methods in action:

```
>>> s = "hello world"
>>> s.count('l')
3
>>> s.find('l')
2
>>> s.rindex('m')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Most of the remaining string methods return transformations of the string. The `upper`, `lower`, `capitalize`, and `title` methods create new strings with all alphabetic characters in the given format. The `translate` method can use a dictionary to map arbitrary input characters to specified output characters.

For all of these methods, note that the input string remains unmodified; a brand new `str` instance is returned instead. If we need to manipulate the resultant string, we should assign it to a new variable, as in `new_value = value.capitalize()`. Often, once we've performed the transformation, we don't need the old value anymore, so a common idiom is to assign it to the same variable, as in `value = value.title()`.

Finally, a couple of string methods return or operate on lists. The `split` method accepts a substring and splits the string into a list of strings wherever that substring occurs. You can pass a number as a second parameter to limit the number of resultant strings. The `rsplit` behaves identically to `split` if you don't limit the number of strings, but if you do supply a limit, it starts splitting from the end of the string. The `partition` and `rpartition` methods split the string at only the first or last occurrence of the substring, and return a tuple of three values: characters before the substring, the substring itself, and the characters after the substring.

As the inverse of `split`, the `join` method accepts a list of strings, and returns all of those strings combined together by placing the original string between them. The `replace` method accepts two arguments, and returns a string where each instance of the first argument has been replaced with the second. Here are some of these methods in action:

```
>>> s = "hello world, how are you"
>>> s2 = s.split(' ')
>>> s2
['hello', 'world,', 'how', 'are', 'you']
>>> '#'.join(s2)
'hello#world,#how#are#you'
>>> s.replace(' ', '**')
'hello**world,**how**are**you'
>>> s.partition(' ')
('hello', ' ', 'world, how are you')
```

There you have it, a whirlwind tour of the most common methods on the `str` class! Now, let's look at Python 3's method for composing strings and variables to create new strings.

String formatting

Python 3 has a powerful string formatting and templating mechanism that allows us to construct strings comprised of hardcoded text and interspersed variables. We've used it in many previous examples, but it is much more versatile than the simple formatting specifiers we've used.

Any string can be turned into a format string by calling the `format()` method on it. This method returns a new string where specific characters in the input string have been replaced with values provided as arguments and keyword arguments passed into the function. The `format` method does not require a fixed set of arguments; internally, it uses the `*args` and `**kwargs` syntax that we discussed in *Chapter 7*, *Python Object-oriented Shortcuts*.

The special characters that are replaced in formatted strings are the opening and closing brace characters: { and }. We can insert pairs of these in a string and they will be replaced, in order, by any positional arguments passed to the `str.format` method:

```
template = "Hello {}, you are currently {}."
print(template.format('Dusty', 'writing'))
```

If we run these statements, it replaces the braces with variables, in order:

```
Hello Dusty, you are currently writing.
```

This basic syntax is not terribly useful if we want to reuse variables within one string or decide to use them in a different position. We can place zero-indexed integers inside the curly braces to tell the formatter which positional variable gets inserted at a given position in the string. Let's repeat the name:

```
template = "Hello {0}, you are {1}. Your name is {0}."
print(template.format('Dusty', 'writing'))
```

If we use these integer indexes, we have to use them in all the variables. We can't mix empty braces with positional indexes. For example, this code fails with an appropriate `ValueError` exception:

```
template = "Hello {}, you are {}. Your name is {0}."
print(template.format('Dusty', 'writing'))
```

Escaping braces

Brace characters are often useful in strings, aside from formatting. We need a way to escape them in situations where we want them to be displayed as themselves, rather than being replaced. This can be done by doubling the braces. For example, we can use Python to format a basic Java program:

```
template = """
public class {0} {{
    public static void main(String[] args) {{
        System.out.println("{1}");
    }}
}}"""

print(template.format("MyClass", "print('hello world')"));
```

Wherever we see the {{ or }} sequence in the template, that is, the braces enclosing the Java class and method definition, we know the `format` method will replace them with single braces, rather than some argument passed into the `format` method. Here's the output:

```
public class MyClass {
    public static void main(String[] args) {
        System.out.println("print('hello world')");
    }
}
```

The class name and contents of the output have been replaced with two parameters, while the double braces have been replaced with single braces, giving us a valid Java file. Turns out, this is about the simplest possible Python program to print the simplest possible Java program that can print the simplest possible Python program!

Keyword arguments

If we're formatting complex strings, it can become tedious to remember the order of the arguments or to update the template if we choose to insert a new argument. The `format` method therefore allows us to specify names inside the braces instead of numbers. The named variables are then passed to the `format` method as keyword arguments:

```
template = """
From: <{from_email}>
To: <{to_email}>
Subject: {subject}

{message}"""
print(template.format(
    from_email = "a@example.com",
    to_email = "b@example.com",
    message = "Here's some mail for you. "
    " Hope you enjoy the message!",
    subject = "You have mail!"
    ))
```

We can also mix index and keyword arguments (as with all Python function calls, the keyword arguments must follow the positional ones). We can even mix unlabeled positional braces with keyword arguments:

```
print("{} {label} {}".format("x", "y", label="z"))
```

As expected, this code outputs:

```
x z y
```

Container lookups

We aren't restricted to passing simple string variables into the `format` method. Any primitive, such as integers or floats can be printed. More interestingly, complex objects, including lists, tuples, dictionaries, and arbitrary objects can be used, and we can access indexes and variables (but not methods) on those objects from within the `format` string.

For example, if our e-mail message had grouped the from and to e-mail addresses into a tuple, and placed the subject and message in a dictionary, for some reason (perhaps because that's the input required for an existing `send_mail` function we want to use), we can format it like this:

```
emails = ("a@example.com", "b@example.com")
message = {
        'subject': "You Have Mail!",
        'message': "Here's some mail for you!"
        }
template = """
From: <{0[0]}>
To: <{0[1]}>
Subject: {message[subject]}
{message[message]}"""
print(template.format(emails, message=message))
```

The variables inside the braces in the template string look a little weird, so let's look at what they're doing. We have passed one argument as a position-based parameter and one as a keyword argument. The two e-mail addresses are looked up by `0[x]`, where `x` is either `0` or `1`. The initial zero represents, as with other position-based arguments, the first positional argument passed to `format` (the `emails` tuple, in this case).

The square brackets with a number inside are the same kind of index lookup we see in regular Python code, so `0[0]` maps to `emails[0]`, in the `emails` tuple. The indexing syntax works with any indexable object, so we see similar behavior when we access `message[subject]`, except this time we are looking up a string key in a dictionary. Notice that unlike in Python code, we do not need to put quotes around the string in the dictionary lookup.

We can even do multiple levels of lookup if we have nested data structures. I would recommend against doing this often, as template strings rapidly become difficult to understand. If we have a dictionary that contains a tuple, we can do this:

```
emails = ("a@example.com", "b@example.com")
message = {
        'emails': emails,
        'subject': "You Have Mail!",
        'message': "Here's some mail for you!"
        }
template = """
From: <{0[emails][0]}>
To: <{0[emails][1]}>
Subject: {0[subject]}
{0[message]}"""
print(template.format(message))
```

Object lookups

Indexing makes `format` lookup powerful, but we're not done yet! We can also pass arbitrary objects as parameters, and use the dot notation to look up attributes on those objects. Let's change our e-mail message data once again, this time to a class:

```
class EMail:
    def __init__(self, from_addr, to_addr, subject, message):
        self.from_addr = from_addr
        self.to_addr = to_addr
        self.subject = subject
        self.message = message

email = EMail("a@example.com", "b@example.com",
        "You Have Mail!",
         "Here's some mail for you!")

template = """
From: <{0.from_addr}>
To: <{0.to_addr}>
Subject: {0.subject}

{0.message}"""
print(template.format(email))
```

The template in this example may be more readable than the previous examples, but the overhead of creating an e-mail class adds complexity to the Python code. It would be foolish to create a class for the express purpose of including the object in a template. Typically, we'd use this sort of lookup if the object we are trying to format already exists. This is true of all the examples; if we have a tuple, list, or dictionary, we'll pass it into the template directly. Otherwise, we'd just create a simple set of positional and keyword arguments.

Making it look right

It's nice to be able to include variables in template strings, but sometimes the variables need a bit of coercion to make them look right in the output. For example, if we are doing calculations with currency, we may end up with a long decimal that we don't want to show up in our template:

```
subtotal = 12.32
tax = subtotal * 0.07
total = subtotal + tax

print("Sub: ${0} Tax: ${1} Total: ${total}".format(
    subtotal, tax, total=total))
```

If we run this formatting code, the output doesn't quite look like proper currency:

```
Sub: $12.32 Tax: $0.8624 Total: $13.182400000000001
```

Technically, we should never use floating-point numbers in currency calculations like this; we should construct `decimal.Decimal()` objects instead. Floats are dangerous because their calculations are inherently inaccurate beyond a specific level of precision. But we're looking at strings, not floats, and currency is a great example for formatting!

To fix the preceding `format` string, we can include some additional information inside the curly braces to adjust the formatting of the parameters. There are tons of things we can customize, but the basic syntax inside the braces is the same; first, we use whichever of the earlier layouts (positional, keyword, index, attribute access) is suitable to specify the variable that we want to place in the template string. We follow this with a colon, and then the specific syntax for the formatting. Here's an improved version:

```
print("Sub: ${0:0.2f} Tax: ${1:0.2f} "
        "Total: ${total:0.2f}".format(
            subtotal, tax, total=total))
```

The `0.2f` format specifier after the colons basically says, from left to right: for values lower than one, make sure a zero is displayed on the left side of the decimal point; show two places after the decimal; format the input value as a float.

We can also specify that each number should take up a particular number of characters on the screen by placing a value before the period in the precision. This can be useful for outputting tabular data, for example:

```
orders = [('burger', 2, 5),
        ('fries', 3.5, 1),
        ('cola', 1.75, 3)]

print("PRODUCT    QUANTITY    PRICE    SUBTOTAL")
for product, price, quantity in orders:
    subtotal = price * quantity
    print("{0:10s}{1: ^9d}    ${2: <8.2f}${3: >7.2f}".format(
        product, quantity, price, subtotal))
```

Ok, that's a pretty scary looking format string, so let's see how it works before we break it down into understandable parts:

```
PRODUCT    QUANTITY    PRICE    SUBTOTAL
burger        5        $2.00    $  10.00
fries         1        $3.50    $   3.50
cola          3        $1.75    $   5.25
```

Nifty! So, how is this actually happening? We have four variables we are formatting, in each line in the `for` loop. The first variable is a string and is formatted with `{0:10s}`. The `s` means it is a string variable, and the `10` means it should take up ten characters. By default, with strings, if the string is shorter than the specified number of characters, it appends spaces to the right side of the string to make it long enough (beware, however: if the original string is too long, it won't be truncated!). We can change this behavior (to fill with other characters or change the alignment in the format string), as we do for the next value, `quantity`.

The formatter for the `quantity` value is `{1: ^9d}`. The `d` represents an integer value. The `9` tells us the value should take up nine characters. But with integers, instead of spaces, the extra characters are zeros, by default. That looks kind of weird. So we explicitly specify a space (immediately after the colon) as a padding character. The caret character ^ tells us that the number should be aligned in the center of this available padding; this makes the column look a bit more professional. The specifiers have to be in the right order, although all are optional: fill first, then align, then the size, and finally, the type.

We do similar things with the specifiers for price and subtotal. For `price`, we use `{2:<8.2f}` and for `subtotal`, `{3:>7.2f}`. In both cases, we're specifying a space as the fill character, but we use the `<` and `>` symbols, respectively, to represent that the numbers should be aligned to the left or right within the minimum space of eight or seven characters. Further, each float should be formatted to two decimal places.

The "type" character for different types can affect formatting output as well. We've seen the `s`, `d`, and `f` types, for strings, integers, and floats. Most of the other format specifiers are alternative versions of these; for example, `o` represents octal format and `X` represents hexadecimal for integers. The `n` type specifier can be useful for formatting integer separators in the current locale's format. For floating-point numbers, the `%` type will multiply by 100 and format a float as a percentage.

While these standard formatters apply to most built-in objects, it is also possible for other objects to define nonstandard specifiers. For example, if we pass a `datetime` object into `format`, we can use the specifiers used in the `datetime.strftime` function, as follows:

```
import datetime
print("{0:%Y-%m-%d %I:%M%p }".format(
    datetime.datetime.now()))
```

It is even possible to write custom formatters for objects we create ourselves, but that is beyond the scope of this module. Look into overriding the `__format__` special method if you need to do this in your code. The most comprehensive instructions can be found in PEP 3101 at `http://www.python.org/dev/peps/pep-3101/`, although the details are a bit dry. You can find more digestible tutorials using a web search.

The Python formatting syntax is quite flexible but it is a difficult mini-language to remember. I use it every day and still occasionally have to look up forgotten concepts in the documentation. It also isn't powerful enough for serious templating needs, such as generating web pages. There are several third-party templating libraries you can look into if you need to do more than basic formatting of a few strings.

Strings are Unicode

At the beginning of this section, we defined strings as collections of immutable Unicode characters. This actually makes things very complicated at times, because Unicode isn't really a storage format. If you get a string of bytes from a file or a socket, for example, they won't be in Unicode. They will, in fact, be the built-in type `bytes`. Bytes are immutable sequences of... well, bytes. Bytes are the lowest-level storage format in computing. They represent 8 bits, usually described as an integer between 0 and 255, or a hexadecimal equivalent between 0 and FF. Bytes don't represent anything specific; a sequence of bytes may store characters of an encoded string, or pixels in an image.

If we print a byte object, any bytes that map to ASCII representations will be printed as their original character, while non-ASCII bytes (whether they are binary data or other characters) are printed as hex codes escaped by the `\x` escape sequence. You may find it odd that a byte, represented as an integer, can map to an ASCII character. But ASCII is really just a code where each letter is represented by a different byte pattern, and therefore, a different integer. The character "a" is represented by the same byte as the integer 97, which is the hexadecimal number 0x61. Specifically, all of these are an interpretation of the binary pattern 01100001.

Many I/O operations only know how to deal with `bytes`, even if the bytes object refers to textual data. It is therefore vital to know how to convert between `bytes` and Unicode.

The problem is that there are many ways to map `bytes` to Unicode text. Bytes are machine-readable values, while text is a human-readable format. Sitting in between is an encoding that maps a given sequence of bytes to a given sequence of text characters.

However, there are multiple such encodings (ASCII is only one of them). The same sequence of bytes represents completely different text characters when mapped using different encodings! So, `bytes` must be decoded using the same character set with which they were encoded. It's not possible to get text from bytes without knowing how the bytes should be decoded. If we receive unknown bytes without a specified encoding, the best we can do is guess what format they are encoded in, and we may be wrong.

Converting bytes to text

If we have an array of `bytes` from somewhere, we can convert it to Unicode using the `.decode` method on the `bytes` class. This method accepts a string for the name of the character encoding. There are many such names; common ones for Western languages include ASCII, UTF-8, and latin-1.

The sequence of bytes (in hex), 63 6c 69 63 68 e9, actually represents the characters of the word cliché in the latin-1 encoding. The following example will encode this sequence of bytes and convert it to a Unicode string using the latin-1 encoding:

```
characters = b'\x63\x6c\x69\x63\x68\xe9'
print(characters)
print(characters.decode("latin-1"))
```

The first line creates a `bytes` object; the `b` character immediately before the string tells us that we are defining a `bytes` object instead of a normal Unicode string. Within the string, each byte is specified using—in this case—a hexadecimal number. The `\x` character escapes within the byte string, and each say, "the next two characters represent a byte using hexadecimal digits."

Provided we are using a shell that understands the latin-1 encoding, the two `print` calls will output the following strings:

```
b'clich\xe9'
cliché
```

The first `print` statement renders the bytes for ASCII characters as themselves. The unknown (unknown to ASCII, that is) character stays in its escaped hex format. The output includes a `b` character at the beginning of the line to remind us that it is a `bytes` representation, not a string.

The next call decodes the string using latin-1 encoding. The `decode` method returns a normal (Unicode) string with the correct characters. However, if we had decoded this same string using the Cyrillic "iso8859-5" encoding, we'd have ended up with the string 'clichщ'! This is because the `\xe9` byte maps to different characters in the two encodings.

Converting text to bytes

If we need to convert incoming bytes into Unicode, clearly we're also going to have situations where we convert outgoing Unicode into byte sequences. This is done with the `encode` method on the `str` class, which, like the `decode` method, requires a character set. The following code creates a Unicode string and encodes it in different character sets:

```
characters = "cliché"
print(characters.encode("UTF-8"))
print(characters.encode("latin-1"))
print(characters.encode("CP437"))
print(characters.encode("ascii"))
```

The first three encodings create a different set of bytes for the accented character. The fourth one can't even handle that byte:

```
b'clich\xc3\xa9'
b'clich\xe9'
b'clich\x82'
Traceback (most recent call last):
```

```
  File "1261_10_16_decode_unicode.py", line 5, in <module>
    print(characters.encode("ascii"))
UnicodeEncodeError: 'ascii' codec can't encode character '\xe9' in
position 5: ordinal not in range(128)
```

Do you understand the importance of encoding now? The accented character is represented as a different byte for each encoding; if we use the wrong one when we are decoding bytes to text, we get the wrong character.

The exception in the last case is not always the desired behavior; there may be cases where we want the unknown characters to be handled in a different way. The `encode` method takes an optional string argument named `errors` that can define how such characters should be handled. This string can be one of the following:

- `strict`
- `replace`
- `ignore`
- `xmlcharrefreplace`

The `strict` replacement strategy is the default we just saw. When a byte sequence is encountered that does not have a valid representation in the requested encoding, an exception is raised. When the `replace` strategy is used, the character is replaced with a different character; in ASCII, it is a question mark; other encodings may use different symbols, such as an empty box. The `ignore` strategy simply discards any bytes it doesn't understand, while the `xmlcharrefreplace` strategy creates an `xml` entity representing the Unicode character. This can be useful when converting unknown strings for use in an XML document. Here's how each of the strategies affects our sample word:

Strategy	"cliché".encode("ascii", strategy)
`replace`	`b'clich?'`
`ignore`	`b'clich'`
`xmlcharrefreplace`	`b'clich&#233;'`

It is possible to call the `str.encode` and `bytes.decode` methods without passing an encoding string. The encoding will be set to the default encoding for the current platform. This will depend on the current operating system and locale or regional settings; you can look it up using the `sys.getdefaultencoding()` function. It is usually a good idea to specify the encoding explicitly, though, since the default encoding for a platform may change, or the program may one day be extended to work on text from a wider variety of sources.

If you are encoding text and don't know which encoding to use, it is best to use the UTF-8 encoding. UTF-8 is able to represent any Unicode character. In modern software, it is a de facto standard encoding to ensure documents in any language—or even multiple languages—can be exchanged. The various other possible encodings are useful for legacy documents or in regions that still use different character sets by default.

The UTF-8 encoding uses one byte to represent ASCII and other common characters, and up to four bytes for more complex characters. UTF-8 is special because it is backwards-compatible with ASCII; any ASCII document encoded using UTF-8 will be identical to the original ASCII document.

I can never remember whether to use `encode` or `decode` to convert from binary bytes to Unicode. I always wished these methods were named "to_binary" and "from_binary" instead. If you have the same problem, try mentally replacing the word "code" with "binary"; "enbinary" and "debinary" are pretty close to "to_binary" and "from_binary". I have saved a lot of time by not looking up the method help files since devising this mnemonic.

Mutable byte strings

The `bytes` type, like `str`, is immutable. We can use index and slice notation on a `bytes` object and search for a particular sequence of bytes, but we can't extend or modify them. This can be very inconvenient when dealing with I/O, as it is often necessary to buffer incoming or outgoing bytes until they are ready to be sent. For example, if we are receiving data from a socket, it may take several `recv` calls before we have received an entire message.

This is where the `bytearray` built-in comes in. This type behaves something like a list, except it only holds bytes. The constructor for the class can accept a `bytes` object to initialize it. The `extend` method can be used to append another `bytes` object to the existing array (for example, when more data comes from a socket or other I/O channel).

Slice notation can be used on `bytearray` to modify the item inline. For example, this code constructs a `bytearray` from a `bytes` object and then replaces two bytes:

```
b = bytearray(b"abcdefgh")
b[4:6] = b"\x15\xa3"
print(b)
```

The output looks like this:

```
bytearray(b'abcd\x15\xa3gh')
```

Be careful; if we want to manipulate a single element in the `bytearray`, it will expect us to pass an integer between 0 and 255 inclusive as the value. This integer represents a specific `bytes` pattern. If we try to pass a character or `bytes` object, it will raise an exception.

A single byte character can be converted to an integer using the `ord` (short for ordinal) function. This function returns the integer representation of a single character:

```
b = bytearray(b'abcdef')
b[3] = ord(b'g')
b[4] = 68
print(b)
```

The output looks like this:

```
bytearray(b'abcgDf')
```

After constructing the array, we replace the character at index `3` (the fourth character, as indexing starts at `0`, as with lists) with byte 103. This integer was returned by the `ord` function and is the ASCII character for the lowercase `g`. For illustration, we also replaced the next character up with the byte number `68`, which maps to the ASCII character for the uppercase `D`.

The `bytearray` type has methods that allow it to behave like a list (we can append integer bytes to it, for example), but also like a `bytes` object; we can use methods like `count` and `find` the same way they would behave on a `bytes` or `str` object. The difference is that `bytearray` is a mutable type, which can be useful for building up complex sequences of bytes from a specific input source.

Regular expressions

You know what's really hard to do using object-oriented principles? Parsing strings to match arbitrary patterns, that's what. There have been a fair number of academic papers written in which object-oriented design is used to set up string parsing, but the result is always very verbose and hard to read, and they are not widely used in practice.

In the real world, string parsing in most programming languages is handled by regular expressions. These are not verbose, but, boy, are they ever hard to read, at least until you learn the syntax. Even though regular expressions are not object oriented, the Python regular expression library provides a few classes and objects that you can use to construct and run regular expressions.

Regular expressions are used to solve a common problem: Given a string, determine whether that string matches a given pattern and, optionally, collect substrings that contain relevant information. They can be used to answer questions like:

- Is this string a valid URL?
- What is the date and time of all warning messages in a log file?
- Which users in `/etc/passwd` are in a given group?
- What username and document were requested by the URL a visitor typed?

There are many similar scenarios where regular expressions are the correct answer. Many programmers have made the mistake of implementing complicated and fragile string parsing libraries because they didn't know or wouldn't learn regular expressions. In this section, we'll gain enough knowledge of regular expressions to not make such mistakes!

Matching patterns

Regular expressions are a complicated mini-language. They rely on special characters to match unknown strings, but let's start with literal characters, such as letters, numbers, and the space character, which always match themselves. Let's see a basic example:

```
import re

search_string = "hello world"
pattern = "hello world"

match = re.match(pattern, search_string)

if match:
    print("regex matches")
```

The Python Standard Library module for regular expressions is called `re`. We import it and set up a search string and pattern to search for; in this case, they are the same string. Since the search string matches the given pattern, the conditional passes and the `print` statement executes.

Bear in mind that the `match` function matches the pattern to the beginning of the string. Thus, if the pattern were `"ello world"`, no match would be found. With confusing asymmetry, the parser stops searching as soon as it finds a match, so the pattern `"hello wo"` matches successfully. Let's build a small example program to demonstrate these differences and help us learn other regular expression syntax:

```
import sys
```

```
import re

pattern = sys.argv[1]
search_string = sys.argv[2]
match = re.match(pattern, search_string)

if match:
    template = "'{}' matches pattern '{}'"
else:
    template = "'{}' does not match pattern '{}'"

print(template.format(search_string, pattern))
```

This is just a generic version of the earlier example that accepts the pattern and search string from the command line. We can see how the start of the pattern must match, but a value is returned as soon as a match is found in the following command-line interaction:

```
$ python regex_generic.py "hello worl" "hello world"
'hello world' matches pattern 'hello worl'
$ python regex_generic.py "ello world" "hello world"
'hello world' does not match pattern 'ello world'
```

We'll be using this script throughout the next few sections. While the script is always invoked with the command line `python regex_generic.py "<pattern>" "<string>"`, we'll only see the output in the following examples, to conserve space.

If you need control over whether items happen at the beginning or end of a line (or if there are no newlines in the string, at the beginning and end of the string), you can use the `^` and `$` characters to represent the start and end of the string respectively. If you want a pattern to match an entire string, it's a good idea to include both of these:

```
'hello world' matches pattern '^hello world$'
'hello worl' does not match pattern '^hello world$'
```

Matching a selection of characters

Let's start with matching an arbitrary character. The period character, when used in a regular expression pattern, can match any single character. Using a period in the string means you don't care what the character is, just that there is a character there. For example:

```
'hello world' matches pattern 'hel.o world'
'helpo world' matches pattern 'hel.o world'
```

```
'hel o world' matches pattern 'hel.o world'
'helo world' does not match pattern 'hel.o world'
```

Notice how the last example does not match because there is no character at the period's position in the pattern.

That's all well and good, but what if we only want a few specific characters to match? We can put a set of characters inside square brackets to match any one of those characters. So if we encounter the string `[abc]` in a regular expression pattern, we know that those five (including the two square brackets) characters will only match one character in the string being searched, and further, that this one character will be either an `a`, a `b`, or a `c`. See a few examples:

```
'hello world' matches pattern 'hel[lp]o world'
'helpo world' matches pattern 'hel[lp]o world'
'helPo world' does not match pattern 'hel[lp]o world'
```

These square bracket sets should be named character sets, but they are more often referred to as **character classes**. Often, we want to include a large range of characters inside these sets, and typing them all out can be monotonous and error-prone. Fortunately, the regular expression designers thought of this and gave us a shortcut. The dash character, in a character set, will create a range. This is especially useful if you want to match "all lower case letters", "all letters", or "all numbers" as follows:

```
'hello   world' does not match pattern 'hello [a-z] world'
'hello b world' matches pattern 'hello [a-z] world'
'hello B world' matches pattern 'hello [a-zA-Z] world'
'hello 2 world' matches pattern 'hello [a-zA-Z0-9] world'
```

There are other ways to match or exclude individual characters, but you'll need to find a more comprehensive tutorial via a web search if you want to find out what they are!

Escaping characters

If putting a period character in a pattern matches any arbitrary character, how do we match just a period in a string? One way might be to put the period inside square brackets to make a character class, but a more generic method is to use backslashes to escape it. Here's a regular expression to match two digit decimal numbers between 0.00 and 0.99:

```
'0.05' matches pattern '0\.[0-9][0-9]'
'005' does not match pattern '0\.[0-9][0-9]'
'0,05' does not match pattern '0\.[0-9][0-9]'
```

For this pattern, the two characters `\.` match the single `.` character. If the period character is missing or is a different character, it does not match.

This backslash escape sequence is used for a variety of special characters in regular expressions. You can use `\[` to insert a square bracket without starting a character class, and `\(` to insert a parenthesis, which we'll later see is also a special character.

More interestingly, we can also use the escape symbol followed by a character to represent special characters such as newlines (`\n`), and tabs (`\t`). Further, some character classes can be represented more succinctly using escape strings; `\s` represents whitespace characters, `\w` represents letters, numbers, and underscore, and `\d` represents a digit:

```
'(abc]' matches pattern '\(abc\]'
' 1a' matches pattern '\s\d\w'
'\t5n' does not match pattern '\s\d\w'
'5n' matches pattern '\s\d\w'
```

Matching multiple characters

With this information, we can match most strings of a known length, but most of the time we don't know how many characters to match inside a pattern. Regular expressions can take care of this, too. We can modify a pattern by appending one of several hard-to-remember punctuation symbols to match multiple characters.

The asterisk (`*`) character says that the previous pattern can be matched zero or more times. This probably sounds silly, but it's one of the most useful repetition characters. Before we explore why, consider some silly examples to make sure we understand what it does:

```
'hello' matches pattern 'hel*o'
'heo' matches pattern 'hel*o'
'hellllllo' matches pattern 'hel*o'
```

So, the `*` character in the pattern says that the previous pattern (the `l` character) is optional, and if present, can be repeated as many times as possible to match the pattern. The rest of the characters (`h`, `e`, and `o`) have to appear exactly once.

It's pretty rare to want to match a single letter multiple times, but it gets more interesting if we combine the asterisk with patterns that match multiple characters. `.*`, for example, will match any string, whereas `[a-z]*` matches any collection of lowercase words, including the empty string.

For example:

```
'A string.' matches pattern '[A-Z][a-z]* [a-z]*\.'
'No .' matches pattern '[A-Z][a-z]* [a-z]*\.'
'' matches pattern '[a-z]*.*'
```

The plus (+) sign in a pattern behaves similarly to an asterisk; it states that the previous pattern can be repeated one or more times, but, unlike the asterisk is not optional. The question mark (?) ensures a pattern shows up exactly zero or one times, but not more. Let's explore some of these by playing with numbers (remember that `\d` matches the same character class as `[0-9]`:

```
'0.4' matches pattern '\d+\.\d+'
'1.002' matches pattern '\d+\.\d+'
'1.' does not match pattern '\d+\.\d+'
'1%' matches pattern '\d?\d%'
'99%' matches pattern '\d?\d%'
'999%' does not match pattern '\d?\d%'
```

Grouping patterns together

So far we've seen how we can repeat a pattern multiple times, but we are restricted in what patterns we can repeat. If we want to repeat individual characters, we're covered, but what if we want a repeating sequence of characters? Enclosing any set of patterns in parenthesis allows them to be treated as a single pattern when applying repetition operations. Compare these patterns:

```
'abccc' matches pattern 'abc{3}'
'abccc' does not match pattern '(abc){3}'
'abcabcabc' matches pattern '(abc){3}'
```

Combined with complex patterns, this grouping feature greatly expands our pattern-matching repertoire. Here's a regular expression that matches simple English sentences:

```
'Eat.' matches pattern '[A-Z][a-z]*( [a-z]+)*\.$'
'Eat more good food.' matches pattern '[A-Z][a-z]*( [a-z]+)*\.$'
'A good meal.' matches pattern '[A-Z][a-z]*( [a-z]+)*\.$'
```

The first word starts with a capital, followed by zero or more lowercase letters. Then, we enter a parenthetical that matches a single space followed by a word of one or more lowercase letters. This entire parenthetical is repeated zero or more times, and the pattern is terminated with a period. There cannot be any other characters after the period, as indicated by the `$` matching the end of string.

We've seen many of the most basic patterns, but the regular expression language supports many more. I spent my first few years using regular expressions looking up the syntax every time I needed to do something. It is worth bookmarking Python's documentation for the `re` module and reviewing it frequently. There are very few things that regular expressions cannot match, and they should be the first tool you reach for when parsing strings.

Getting information from regular expressions

Let's now focus on the Python side of things. The regular expression syntax is the furthest thing from object-oriented programming. However, Python's `re` module provides an object-oriented interface to enter the regular expression engine.

We've been checking whether the `re.match` function returns a valid object or not. If a pattern does not match, that function returns `None`. If it does match, however, it returns a useful object that we can introspect for information about the pattern.

So far, our regular expressions have answered questions such as "Does this string match this pattern?" Matching patterns is useful, but in many cases, a more interesting question is, "If this string matches this pattern, what is the value of a relevant substring?" If you use groups to identify parts of the pattern that you want to reference later, you can get them out of the match return value as illustrated in the next example:

```
pattern = "^[a-zA-Z.]+@([a-z.]*\.[a-z]+)$"
search_string = "some.user@example.com"
match = re.match(pattern, search_string)

if match:
    domain = match.groups()[0]
    print(domain)
```

The specification describing valid e-mail addresses is extremely complicated, and the regular expression that accurately matches all possibilities is obscenely long. So we cheated and made a simple regular expression that matches some common e-mail addresses; the point is that we want to access the domain name (after the `@` sign) so we can connect to that address. This is done easily by wrapping that part of the pattern in parenthesis and calling the `groups()` method on the object returned by match.

The `groups` method returns a tuple of all the groups matched inside the pattern, which you can index to access a specific value. The groups are ordered from left to right. However, bear in mind that groups can be nested, meaning you can have one or more groups inside another group. In this case, the groups are returned in the order of their left-most brackets, so the outermost group will be returned before its inner matching groups.

In addition to the match function, the `re` module provides a couple other useful functions, `search`, and `findall`. The `search` function finds the first instance of a matching pattern, relaxing the restriction that the pattern start at the first letter of the string. Note that you can get a similar effect by using match and putting a `^.*` character at the front of the pattern to match any characters between the start of the string and the pattern you are looking for.

The `findall` function behaves similarly to search, except that it finds all non-overlapping instances of the matching pattern, not just the first one. Basically, it finds the first match, then it resets the search to the end of that matching string and finds the next one.

Instead of returning a list of match objects, as you would expect, it returns a list of matching strings. Or tuples. Sometimes it's strings, sometimes it's tuples. It's not a very good API at all! As with all bad APIs, you'll have to memorize the differences and not rely on intuition. The type of the return value depends on the number of bracketed groups inside the regular expression:

- If there are no groups in the pattern, `re.findall` will return a list of strings, where each value is a complete substring from the source string that matches the pattern
- If there is exactly one group in the pattern, `re.findall` will return a list of strings where each value is the contents of that group
- If there are multiple groups in the pattern, then `re.findall` will return a list of tuples where each tuple contains a value from a matching group, in order

When you are designing function calls in your own Python libraries, try to make the function always return a consistent data structure. It is often good to design functions that can take arbitrary inputs and process them, but the return value should not switch from single value to a list, or a list of values to a list of tuples depending on the input. Let `re.findall` be a lesson!

The examples in the following interactive session will hopefully clarify the differences:

```
>>> import re
>>> re.findall('a.', 'abacadefagah')
['ab', 'ac', 'ad', 'ag', 'ah']
>>> re.findall('a(.)', 'abacadefagah')
['b', 'c', 'd', 'g', 'h']
>>> re.findall('(a)(.)', 'abacadefagah')
```

```
[('a', 'b'), ('a', 'c'), ('a', 'd'), ('a', 'g'), ('a', 'h')]
>>> re.findall('((a)(.))', 'abacadefagah')
[('ab', 'a', 'b'), ('ac', 'a', 'c'), ('ad', 'a', 'd'), ('ag', 'a', 'g'),
('ah', 'a', 'h')]
```

Making repeated regular expressions efficient

Whenever you call one of the regular expression methods, the engine has to convert the pattern string into an internal structure that makes searching strings fast. This conversion takes a non-trivial amount of time. If a regular expression pattern is going to be reused multiple times (for example, inside a `for` or `while` loop), it would be better if this conversion step could be done only once.

This is possible with the `re.compile` method. It returns an object-oriented version of the regular expression that has been compiled down and has the methods we've explored (`match`, `search`, `findall`) already, among others. We'll see examples of this in the case study.

This has definitely been a condensed introduction to regular expressions. At this point, we have a good feel for the basics and will recognize when we need to do further research. If we have a string pattern matching problem, regular expressions will almost certainly be able to solve them for us. However, we may need to look up new syntaxes in a more comprehensive coverage of the topic. But now we know what to look for! Let's move on to a completely different topic: serializing data for storage.

Serializing objects

Nowadays, we take the ability to write data to a file and retrieve it at an arbitrary later date for granted. As convenient as this is (imagine the state of computing if we couldn't store anything!), we often find ourselves converting data we have stored in a nice object or design pattern in memory into some kind of clunky text or binary format for storage, transfer over the network, or remote invocation on a distant server.

The Python `pickle` module is an object-oriented way to store objects directly in a special storage format. It essentially converts an object (and all the objects it holds as attributes) into a sequence of bytes that can be stored or transported however we see fit.

For basic work, the `pickle` module has an extremely simple interface. It is comprised of four basic functions for storing and loading data; two for manipulating file-like objects, and two for manipulating `bytes` objects (the latter are just shortcuts to the file-like interface, so we don't have to create a `BytesIO` file-like object ourselves).

The `dump` method accepts an object to be written and a file-like object to write the serialized bytes to. This object must have a `write` method (or it wouldn't be file-like), and that method must know how to handle a `bytes` argument (so a file opened for text output wouldn't work).

The `load` method does exactly the opposite; it reads a serialized object from a file-like object. This object must have the proper file-like `read` and `readline` arguments, each of which must, of course, return `bytes`. The `pickle` module will load the object from these bytes and the `load` method will return the fully reconstructed object. Here's an example that stores and then loads some data in a list object:

```
import pickle

some_data = ["a list", "containing", 5,
        "values including another list",
        ["inner", "list"]]

with open("pickled_list", 'wb') as file:
    pickle.dump(some_data, file)

with open("pickled_list", 'rb') as file:
    loaded_data = pickle.load(file)

print(loaded_data)
assert loaded_data == some_data
```

This code works as advertised: the objects are stored in the file and then loaded from the same file. In each case, we open the file using a `with` statement so that it is automatically closed. The file is first opened for writing and then a second time for reading, depending on whether we are storing or loading data.

The `assert` statement at the end would raise an error if the newly loaded object is not equal to the original object. Equality does not imply that they are the same object. Indeed, if we print the `id()` of both objects, we would discover they are different. However, because they are both lists whose contents are equal, the two lists are also considered equal.

The `dumps` and `loads` functions behave much like their file-like counterparts, except they return or accept `bytes` instead of file-like objects. The `dumps` function requires only one argument, the object to be stored, and it returns a serialized `bytes` object. The `loads` function requires a `bytes` object and returns the restored object. The `'s'` character in the method names is short for string; it's a legacy name from ancient versions of Python, where `str` objects were used instead of `bytes`.

Both `dump` methods accept an optional `protocol` argument. If we are saving and loading pickled objects that are only going to be used in Python 3 programs, we don't need to supply this argument. Unfortunately, if we are storing objects that may be loaded by older versions of Python, we have to use an older and less efficient protocol. This should not normally be an issue. Usually, the only program that would load a pickled object would be the same one that stored it. Pickle is an unsafe format, so we don't want to be sending it unsecured over the Internet to unknown interpreters.

The argument supplied is an integer version number. The default version is number 3, representing the current highly efficient storage system used by Python 3 pickling. The number 2 is the older version, which will store an object that can be loaded on all interpreters back to Python 2.3. As 2.6 is the oldest of Python that is still widely used in the wild, version 2 pickling is normally sufficient. Versions 0 and 1 are supported on older interpreters; 0 is an ASCII format, while 1 is a binary format. There is also an optimized version 4 that may one day become the default.

As a rule of thumb, then, if you know that the objects you are pickling will only be loaded by a Python 3 program (for example, only your program will be loading them), use the default pickling protocol. If they may be loaded by unknown interpreters, pass a protocol value of 2, unless you really believe they may need to be loaded by an archaic version of Python.

If we do pass a protocol to `dump` or `dumps`, we should use a keyword argument to specify it: `pickle.dumps(my_object, protocol=2)`. This is not strictly necessary, as the method only accepts two arguments, but typing out the full keyword argument reminds readers of our code what the purpose of the number is. Having a random integer in the method call would be hard to read. Two what? Store two copies of the object, maybe? Remember, code should always be readable. In Python, less code is often more readable than longer code, but not always. Be explicit.

It is possible to call `dump` or `load` on a single open file more than once. Each call to `dump` will store a single object (plus any objects it is composed of or contains), while a call to `load` will load and return just one object. So for a single file, each separate call to `dump` when storing the object should have an associated call to `load` when restoring at a later date.

Customizing pickles

With most common Python objects, pickling "just works". Basic primitives such as integers, floats, and strings can be pickled, as can any container object, such as lists or dictionaries, provided the contents of those containers are also picklable. Further, and importantly, any object can be pickled, so long as all of its attributes are also picklable.

So what makes an attribute unpicklable? Usually, it has something to do with time-sensitive attributes that it would not make sense to load in the future. For example, if we have an open network socket, open file, running thread, or database connection stored as an attribute on an object, it would not make sense to pickle these objects; a lot of operating system state would simply be gone when we attempted to reload them later. We can't just pretend a thread or socket connection exists and make it appear! No, we need to somehow customize how such transient data is stored and restored.

Here's a class that loads the contents of a web page every hour to ensure that they stay up to date. It uses the `threading.Timer` class to schedule the next update:

```
from threading import Timer
import datetime
from urllib.request import urlopen

class UpdatedURL:
    def __init__(self, url):
        self.url = url
        self.contents = ''
        self.last_updated = None
        self.update()

    def update(self):
        self.contents = urlopen(self.url).read()
        self.last_updated = datetime.datetime.now()
        self.schedule()

    def schedule(self):
        self.timer = Timer(3600, self.update)
        self.timer.setDaemon(True)
        self.timer.start()
```

The `url`, `contents`, and `last_updated` are all pickleable, but if we try to pickle an instance of this class, things go a little nutty on the `self.timer` instance:

```
>>> u = UpdatedURL("http://news.yahoo.com/")
>>> import pickle
>>> serialized = pickle.dumps(u)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    serialized = pickle.dumps(u)
_pickle.PicklingError: Can't pickle <class '_thread.lock'>: attribute
lookup lock on _thread failed
```

That's not a very useful error, but it looks like we're trying to pickle something we shouldn't be. That would be the `Timer` instance; we're storing a reference to `self.timer` in the schedule method, and that attribute cannot be serialized.

When `pickle` tries to serialize an object, it simply tries to store the object's `__dict__` attribute; `__dict__` is a dictionary mapping all the attribute names on the object to their values. Luckily, before checking `__dict__`, `pickle` checks to see whether a `__getstate__` method exists. If it does, it will store the return value of that method instead of the `__dict__`.

Let's add a `__getstate__` method to our `UpdatedURL` class that simply returns a copy of the `__dict__` without a timer:

```
def __getstate__(self):
    new_state = self.__dict__.copy()
    if 'timer' in new_state:
        del new_state['timer']
    return new_state
```

If we pickle the object now, it will no longer fail. And we can even successfully restore that object using `loads`. However, the restored object doesn't have a timer attribute, so it will not be refreshing the content like it is designed to do. We need to somehow create a new timer (to replace the missing one) when the object is unpickled.

As we might expect, there is a complementary `__setstate__` method that can be implemented to customize unpickling. This method accepts a single argument, which is the object returned by `__getstate__`. If we implement both methods, `__getstate__` is not required to return a dictionary, since `__setstate__` will know what to do with whatever object `__getstate__` chooses to return. In our case, we simply want to restore the `__dict__`, and then create a new timer:

```
def __setstate__(self, data):
    self.__dict__ = data
    self.schedule()
```

The `pickle` module is very flexible and provides other tools to further customize the pickling process if you need them. However, these are beyond the scope of this module. The tools we've covered are sufficient for many basic pickling tasks. Objects to be pickled are normally relatively simple data objects; we would not likely pickle an entire running program or complicated design pattern, for example.

Serializing web objects

It is not a good idea to load a pickled object from an unknown or untrusted source. It is possible to inject arbitrary code into a pickled file to maliciously attack a computer via the pickle. Another disadvantage of pickles is that they can only be loaded by other Python programs, and cannot be easily shared with services written in other languages.

There are many formats that have been used for this purpose over the years. XML (Extensible Markup Language) used to be very popular, especially with Java developers. YAML (Yet Another Markup Language) is another format that you may see referenced occasionally. Tabular data is frequently exchanged in the CSV (Comma Separated Value) format. Many of these are fading into obscurity and there are many more that you will encounter over time. Python has solid standard or third-party libraries for all of them.

Before using such libraries on untrusted data, make sure to investigate security concerns with each of them. XML and YAML, for example, both have obscure features that, used maliciously, can allow arbitrary commands to be executed on the host machine. These features may not be turned off by default. Do your research.

JavaScript Object Notation (**JSON**) is a human readable format for exchanging primitive data. JSON is a standard format that can be interpreted by a wide array of heterogeneous client systems. Hence, JSON is extremely useful for transmitting data between completely decoupled systems. Further, JSON does not have any support for executable code, only data can be serialized; thus, it is more difficult to inject malicious statements into it.

Because JSON can be easily interpreted by JavaScript engines, it is often used for transmitting data from a web server to a JavaScript-capable web browser. If the web application serving the data is written in Python, it needs a way to convert internal data into the JSON format.

There is a module to do this, predictably named `json`. This module provides a similar interface to the `pickle` module, with `dump`, `load`, `dumps`, and `loads` functions. The default calls to these functions are nearly identical to those in `pickle`, so let us not repeat the details. There are a couple differences; obviously, the output of these calls is valid JSON notation, rather than a pickled object. In addition, the `json` functions operate on `str` objects, rather than `bytes`. Therefore, when dumping to or loading from a file, we need to create text files rather than binary ones.

The JSON serializer is not as robust as the `pickle` module; it can only serialize basic types such as integers, floats, and strings, and simple containers such as dictionaries and lists. Each of these has a direct mapping to a JSON representation, but JSON is unable to represent classes, methods, or functions. It is not possible to transmit complete objects in this format. Because the receiver of an object we have dumped to JSON format is normally not a Python object, it would not be able to understand classes or methods in the same way that Python does, anyway. In spite of the O for Object in its name, JSON is a **data** notation; objects, as you recall, are composed of both data and behavior.

If we do have objects for which we want to serialize only the data, we can always serialize the object's `__dict__` attribute. Or we can semiautomate this task by supplying custom code to create or parse a JSON serializable dictionary from certain types of objects.

In the `json` module, both the object storing and loading functions accept optional arguments to customize the behavior. The `dump` and `dumps` methods accept a poorly named `cls` (short for class, which is a reserved keyword) keyword argument. If passed, this should be a subclass of the `JSONEncoder` class, with the `default` method overridden. This method accepts an arbitrary object and converts it to a dictionary that `json` can digest. If it doesn't know how to process the object, we should call the `super()` method, so that it can take care of serializing basic types in the normal way.

The `load` and `loads` methods also accept such a `cls` argument that can be a subclass of the inverse class, `JSONDecoder`. However, it is normally sufficient to pass a function into these methods using the `object_hook` keyword argument. This function accepts a dictionary and returns an object; if it doesn't know what to do with the input dictionary, it can return it unmodified.

Let's look at an example. Imagine we have the following simple contact class that we want to serialize:

```
class Contact:
    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def full_name(self):
        return("{} {}".format(self.first, self.last))
```

We could just serialize the `__dict__` attribute:

```
>>> c = Contact("John", "Smith")
>>> json.dumps(c.__dict__)
'{"last": "Smith", "first": "John"}'
```

But accessing special (double-underscore) attributes in this fashion is kind of crude. Also, what if the receiving code (perhaps some JavaScript on a web page) wanted that `full_name` property to be supplied? Of course, we could construct the dictionary by hand, but let's create a custom encoder instead:

```
import json
class ContactEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Contact):
            return {'is_contact': True,
                    'first': obj.first,
                    'last': obj.last,
                    'full': obj.full_name}
        return super().default(obj)
```

The `default` method basically checks to see what kind of object we're trying to serialize; if it's a contact, we convert it to a dictionary manually; otherwise, we let the parent class handle serialization (by assuming that it is a basic type, which `json` knows how to handle). Notice that we pass an extra attribute to identify this object as a contact, since there would be no way to tell upon loading it. This is just a convention; for a more generic serialization mechanism, it might make more sense to store a string type in the dictionary, or possibly even the full class name, including package and module. Remember that the format of the dictionary depends on the code at the receiving end; there has to be an agreement as to how the data is going to be specified.

We can use this class to encode a contact by passing the class (not an instantiated object) to the `dump` or `dumps` function:

```
>>> c = Contact("John", "Smith")
>>> json.dumps(c, cls=ContactEncoder)
'{"is_contact": true, "last": "Smith", "full": "John Smith",
"first": "John"}'
```

For decoding, we can write a function that accepts a dictionary and checks the existence of the `is_contact` variable to decide whether to convert it to a contact:

```
def decode_contact(dic):
        if dic.get('is_contact'):
            return Contact(dic['first'], dic['last'])
        else:
            return dic
```

We can pass this function to the `load` or `loads` function using the `object_hook` keyword argument:

```
>>> data = ('{"is_contact": true, "last": "smith",'
     '"full": "john smith", "first": "john"}')

>>> c = json.loads(data, object_hook=decode_contact)
>>> c
<__main__.Contact object at 0xa02918c>
>>> c.full_name
'john smith'
```

Case study

Let's build a basic regular expression-powered templating engine in Python. This engine will parse a text file (such as an HTML page) and replace certain directives with text calculated from the input to those directives. This is about the most complicated task we would want to do with regular expressions; indeed, a full-fledged version of this would likely utilize a proper language parsing mechanism.

Consider the following input file:

```
/** include header.html **/
<h1>This is the title of the front page</h1>
/** include menu.html **/
<p>My name is /** variable name **/.
This is the content of my front page. It goes below the menu.</p>
<table>
<tr><th>Favourite Books</th></tr>
/** loopover module_list **/
<tr><td>/** loopvar **/</td></tr>

/** endloop **/
</table>
/** include footer.html **/
Copyright &copy; Today
```

This file contains "tags" of the form `/** <directive> <data> **/` where the data is an optional single word and the directives are:

- `include`: Copy the contents of another file here
- `variable`: Insert the contents of a variable here
- `loopover`: Repeat the contents of the loop for a variable that is a list
- `endloop`: Signal the end of looped text
- `loopvar`: Insert a single value from the list being looped over

This template will render a different page depending which variables are passed into it. These variables will be passed in from a so-called context file. This will be encoded as a `json` object with keys representing the variables in question. My context file might look like this, but you would derive your own:

```
{
    "name": "Dusty",
    "module_list": [
        "Thief Of Time",
        "The Thief",
        "Snow Crash",
        "Lathe Of Heaven"
    ]
}
```

Before we get into the actual string processing, let's throw together some object-oriented boilerplate code for processing files and grabbing data from the command line:

```
import re
import sys
import json
from pathlib import Path

DIRECTIVE_RE = re.compile(
    r'/\*\*\s*(include|variable|loopover|endloop|loopvar)'
    r'\s*([^ *]*)\s*\*\*/')


class TemplateEngine:
    def __init__(self, infilename, outfilename, contextfilename):
        self.template = open(infilename).read()
        self.working_dir = Path(infilename).absolute().parent
        self.pos = 0
        self.outfile = open(outfilename, 'w')
```

```
            with open(contextfilename) as contextfile:
                self.context = json.load(contextfile)

        def process(self):
            print("PROCESSING...")


if __name__ == '__main__':
    infilename, outfilename, contextfilename = sys.argv[1:]
    engine = TemplateEngine(infilename, outfilename, contextfilename)
    engine.process()
```

This is all pretty basic, we create a class and initialize it with some variables passed in on the command line.

Notice how we try to make the regular expression a little bit more readable by breaking it across two lines? We use raw strings (the r prefix), so we don't have to double escape all our backslashes. This is common in regular expressions, but it's still a mess. (Regular expressions always are, but they're often worth it.)

The `pos` indicates the current character in the content that we are processing; we'll see a lot more of it in a moment.

Now "all that's left" is to implement that process method. There are a few ways to do this. Let's do it in a fairly explicit way.

The process method has to find each directive that matches the regular expression and do the appropriate work with it. However, it also has to take care of outputting the normal text before, after, and between each directive to the output file, unmodified.

One good feature of the compiled version of regular expressions is that we can tell the `search` method to start searching at a specific position by passing the `pos` keyword argument. If we temporarily define doing the appropriate work with a directive as "ignore the directive and delete it from the output file", our process loop looks quite simple:

```
def process(self):
    match = DIRECTIVE_RE.search(self.template, pos=self.pos)
    while match:
        self.outfile.write(self.template[self.pos:match.start()])
        self.pos = match.end()
        match = DIRECTIVE_RE.search(self.template, pos=self.pos)
    self.outfile.write(self.template[self.pos:])
```

In English, this function finds the first string in the text that matches the regular expression, outputs everything from the current position to the start of that match, and then advances the position to the end of aforesaid match. Once it's out of matches, it outputs everything since the last position.

Of course, ignoring the directive is pretty useless in a templating engine, so let's set up replace that position advancing line with code that delegates to a different method on the class depending on the directive:

```
def process(self):
    match = DIRECTIVE_RE.search(self.template, pos=self.pos)
    while match:
        self.outfile.write(self.template[self.pos:match.start()])
        directive, argument = match.groups()
        method_name = 'process_{}'.format(directive)
        getattr(self, method_name)(match, argument)
        match = DIRECTIVE_RE.search(self.template, pos=self.pos)
    self.outfile.write(self.template[self.pos:])
```

So we grab the directive and the single argument from the regular expression. The directive becomes a method name and we dynamically look up that method name on the `self` object (a little error processing here in case the template writer provides an invalid directive would be better). We pass the match object and argument into that method and assume that method will deal with everything appropriately, including moving the `pos` pointer.

Now that we've got our object-oriented architecture this far, it's actually pretty simple to implement the methods that are delegated to. The `include` and `variable` directives are totally straightforward:

```
def process_include(self, match, argument):
    with (self.working_dir / argument).open() as includefile:
        self.outfile.write(includefile.read())
        self.pos = match.end()

def process_variable(self, match, argument):
    self.outfile.write(self.context.get(argument, ''))
    self.pos = match.end()
```

The first simply looks up the included file and inserts the file contents, while the second looks up the variable name in the context dictionary (which was loaded from `json` in the `__init__` method), defaulting to an empty string if it doesn't exist.

The three methods that deal with looping are a bit more intense, as they have to share state between the three of them. For simplicity (I'm sure you're eager to see the end of this long chapter, we're almost there!), we'll handle this as instance variables on the class itself. As an exercise, you might want to consider better ways to architect this, especially after reading the next three chapters.

```
    def process_loopover(self, match, argument):
        self.loop_index = 0
        self.loop_list = self.context.get(argument, [])
        self.pos = self.loop_pos = match.end()

    def process_loopvar(self, match, argument):
        self.outfile.write(self.loop_list[self.loop_index])
        self.pos = match.end()

    def process_endloop(self, match, argument):
        self.loop_index += 1
        if self.loop_index >= len(self.loop_list):
            self.pos = match.end()
            del self.loop_index
            del self.loop_list
            del self.loop_pos
        else:
            self.pos = self.loop_pos
```

When we encounter the `loopover` directive, we don't have to output anything, but we do have to set the initial state on three variables. The `loop_list` variable is assumed to be a list pulled from the context dictionary. The `loop_index` variable indicates what position in that list should be output in this iteration of the loop, while `loop_pos` is stored so we know where to jump back to when we get to the end of the loop.

The `loopvar` directive outputs the value at the current position in the `loop_list` variable and skips to the end of the directive. Note that it doesn't increment the loop index because the `loopvar` directive could be called multiple times inside a loop.

The `endloop` directive is more complicated. It determines whether there are more elements in the `loop_list`; if there are, it just jumps back to the start of the loop, incrementing the index. Otherwise, it resets all the variables that were being used to process the loop and jumps to the end of the directive so the engine can carry on with the next match.

Note that this particular looping mechanism is very fragile; if a template designer were to try nesting loops or forget an `endloop` call, it would go poorly for them. We would need a lot more error checking and probably want to store more loop state to make this a production platform. But I promised that the end of the chapter was nigh, so let's just head to the exercises, after seeing how our sample template is rendered with its context:

```
<html>
    <body>

<h1>This is the title of the front page</h1>
<a href="link1.html">First Link</a>
<a href="link2.html">Second Link</a>

<p>My name is Dusty.
This is the content of my front page. It goes below the menu.</p>
<table>
<tr><th>Favourite Books</th></tr>

<tr><td>Thief Of Time</td></tr>

<tr><td>The Thief</td></tr>

<tr><td>Snow Crash</td></tr>

<tr><td>Lathe Of Heaven</td></tr>

</table>
    </body>
</html>

Copyright &copy; Today
```

There are some weird newline effects due to the way we planned our template, but it works as expected.

Your Coding Challenge

We've covered a wide variety of topics in this chapter, from strings to regular expressions, to object serialization, and back again. Now it's time to consider how these ideas can be applied to your own code.

Python strings are very flexible, and Python is an extremely powerful tool for string-based manipulations. If you don't do a lot of string processing in your daily work, try designing a tool that is exclusively intended for manipulating strings. Try to come up with something innovative, but if you're stuck, consider writing a web log analyzer (how many requests per hour? How many people visit more than five pages?) or a template tool that replaces certain variable names with the contents of other files.

Ankita Thakur

Your Course Guide

Spend a lot of time toying with the string formatting operators until you've got the syntax memorized. Write a bunch of template strings and objects to pass into the format function, and see what kind of output you get. Try the exotic formatting operators, such as percentage or hexadecimal notation. Try out the fill and alignment operators, and see how they behave differently for integers, strings, and floats. Consider writing a class of your own that has a __format__ method; we didn't discuss this in detail, but explore just how much you can customize formatting.

Make sure you understand the difference between bytes and str objects. The distinction is very complicated in older versions of Python (there was no bytes, and str acted like both bytes and str unless we needed non-ASCII characters in which case there was a separate unicode object, which was similar to Python 3's str class. It's even more confusing than it sounds!). It's clearer nowadays; bytes is for binary data, and str is for character data. The only tricky part is knowing how and when to convert between the two. For practice, try writing text data to a file opened for writing bytes (you'll have to encode the text yourself), and then reading from the same file.

Do some experimenting with bytearray; see how it can act both like a bytes object and a list or container object at the same time. Try writing to a buffer that holds data in the bytes array until it is a certain length before returning it. You can simulate the code that puts data into the buffer by using time.sleep calls to ensure data doesn't arrive too quickly.

Study regular expressions online. Study them some more. Especially learn about named groups greedy versus lazy matching, and regex flags, three features that we didn't cover in this chapter. Make conscious decisions about when not to use them. Many people have very strong opinions about regular expressions and either overuse them or refuse to use them at all. Try to convince yourself to use them only when appropriate, and figure out when that is.

If you've ever written an adapter to load small amounts of data from a file or database and convert it to an object, consider using a pickle instead. Pickles are not efficient for storing massive amounts of data, but they can be useful for loading configuration or other simple objects. Try coding it multiple ways: using a pickle, a text file, or a small database. Which do you find easiest to work with?

Ankita Thakur

Your Course Guide

Try experimenting with pickling data, then modifying the class that holds the data, and loading the pickle into the new class. What works? What doesn't? Is there a way to make drastic changes to a class, such as renaming an attribute or splitting it into two new attributes and still get the data out of an older pickle? (Hint: try placing a private pickle version number on each object and update it each time you change the class; you can then put a migration path in __setstate__.)

If you do any web development at all, do some experimenting with the JSON serializer. Personally, I prefer to serialize only standard JSON serializable objects, rather than writing custom encoders or object_hooks, but the desired effect really depends on the interaction between the frontend (JavaScript, typically) and backend code.

Create some new directives in the templating engine that take more than one or an arbitrary number of arguments. You might need to modify the regular expression or add new ones. Have a look at the Django project's online documentation, and see if there are any other template tags you'd like to work with. Try mimicking their filter syntax instead of using the variable tag. Revisit this xhapter when you've studied iteration and coroutines and see if you can come up with a more compact way of representing the state between related directives, such as the loop.

Summary of Module 1 Chapter 9

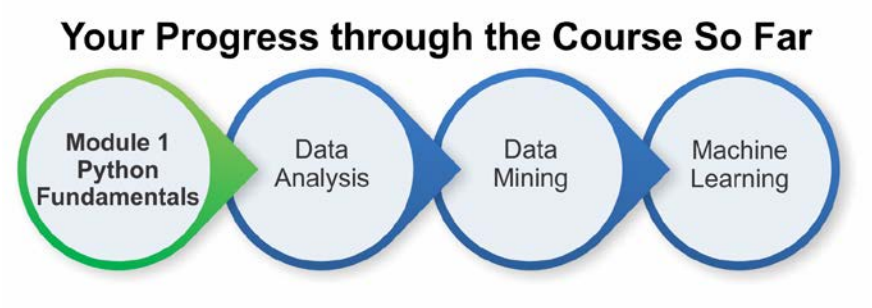
Ankita Thakur

Your Course Guide

We've covered string manipulation, regular expressions, and object serialization in this chapter. Hardcoded strings and program variables can be combined into outputtable strings using the powerful string formatting system. It is important to distinguish between binary and textual data and bytes and str have specific purposes that must be understood. Both are immutable, but the bytearray type can be used when manipulating bytes.

Regular expressions are a complex topic, but we scratched the surface. There are many ways to serialize Python data; pickles and JSON are two of the most popular.

In the next chapter, we'll look at a design pattern that is so fundamental to Python programming that it has been given special syntax support: the iterator pattern.



10
The Iterator Pattern

We've discussed how many of Python's built-ins and idioms that seem, at first blush, to be non-object-oriented are actually providing access to major objects under the hood. In this chapter, we'll discuss how the `for` loop that seems so structured is actually a lightweight wrapper around a set of object-oriented principles. We'll also see a variety of extensions to this syntax that automatically create even more types of object. We will cover:

- What design patterns are
- The iterator protocol—one of the most powerful design patterns
- List, set, and dictionary comprehensions
- Generators and coroutines

Design patterns in brief

When engineers and architects decide to build a bridge, or a tower, or a building, they follow certain principles to ensure structural integrity. There are various possible designs for bridges (suspension or cantilever, for example), but if the engineer doesn't use one of the standard designs, and doesn't have a brilliant new design, it is likely the bridge he/she designs will collapse.

Design patterns are an attempt to bring this same formal definition for correctly designed structures to software engineering. There are many different design patterns to solve different general problems. People who create design patterns first identify a common problem faced by developers in a wide variety of situations. They then suggest what might be considered the ideal solution for that problem, in terms of object-oriented design.

Knowing a design pattern and choosing to use it in our software does not, however, guarantee that we are creating a "correct" solution. In 1907, the Québec Bridge (to this day, the longest cantilever bridge in the world) collapsed before construction was completed, because the engineers who designed it grossly underestimated the weight of the steel used to construct it. Similarly, in software development, we may incorrectly choose or apply a design pattern, and create software that "collapses" under normal operating situations or when stressed beyond its original design limits.

Any one design pattern proposes a set of objects interacting in a specific way to solve a general problem. The job of the programmer is to recognize when they are facing a specific version of that problem, and to adapt the general design in their solution.

In this chapter, we'll be covering the iterator design pattern. This pattern is so powerful and pervasive that the Python developers have provided multiple syntaxes to access the object-oriented principles underlying the pattern. We will be covering other design patterns in the next two chapters. Some of them have language support and some don't, but none of them is as intrinsically a part of the Python coder's daily life as the iterator pattern.

Iterators

In typical design pattern parlance, an iterator is an object with a `next()` method and a `done()` method; the latter returns `True` if there are no items left in the sequence. In a programming language without built-in support for iterators, the iterator would be looped over like this:

```
while not iterator.done():
    item = iterator.next()
    # do something with the item
```

In Python, iteration is a special feature, so the method gets a special name, `__next__`. This method can be accessed using the `next(iterator)` built-in. Rather than a `done` method, the iterator protocol raises `StopIteration` to notify the loop that it has completed. Finally, we have the much more readable `for item in iterator` syntax to actually access items in an iterator instead of messing around with a `while` loop. Let's look at these in more detail.

The iterator protocol

The abstract base class `Iterator`, in the `collections.abc` module, defines the iterator protocol in Python. As mentioned, it must have a `__next__` method that the `for` loop (and other features that support iteration) can call to get a new element from the sequence. In addition, every iterator must also fulfill the `Iterable` interface. Any class that provides an `__iter__` method is iterable; that method must return an `Iterator` instance that will cover all the elements in that class. Since an iterator is already looping over elements, its `__iter__` function traditionally returns itself.

This might sound a bit confusing, so have a look at the following example, but note that this is a very verbose way to solve this problem. It clearly explains iteration and the two protocols in question, but we'll be looking at several more readable ways to get this effect later in this chapter:

```
class CapitalIterable:
    def __init__(self, string):
        self.string = string

    def __iter__(self):
        return CapitalIterator(self.string)


class CapitalIterator:
    def __init__(self, string):
        self.words = [w.capitalize() for w in string.split()]
        self.index = 0

    def __next__(self):
        if self.index == len(self.words):
            raise StopIteration()

        word = self.words[self.index]
        self.index += 1
        return word

    def __iter__(self):
        return self
```

This example defines an `CapitalIterable` class whose job is to loop over each of the words in a string and output them with the first letter capitalized. Most of the work of that iterable is passed to the `CapitalIterator` implementation. The canonical way to interact with this iterator is as follows:

```
>>> iterable = CapitalIterable('the quick brown fox jumps over the lazy
dog')
>>> iterator = iter(iterable)
>>> while True:
...     try:
...         print(next(iterator))
...     except StopIteration:
...         break
...
The
Quick
Brown
Fox
Jumps
Over
The
Lazy
Dog
```

This example first constructs an iterable and retrieves an iterator from it. The distinction may need explanation; the iterable is an object with elements that can be looped over. Normally, these elements can be looped over multiple times, maybe even at the same time or in overlapping code. The iterator, on the other hand, represents a specific location in that iterable; some of the items have been consumed and some have not. Two different iterators might be at different places in the list of words, but any one iterator can mark only one place.

Each time `next()` is called on the iterator, it returns another token from the iterable, in order. Eventually, the iterator will be exhausted (won't have any more elements to return), in which case `Stopiteration` is raised, and we break out of the loop.

Of course, we already know a much simpler syntax for constructing an iterator from an iterable:

```
>>> for i in iterable:
...     print(i)
...
The
Quick
Brown
Fox
Jumps
Over
The
Lazy
Dog
```

As you can see, the `for` statement, in spite of not looking terribly object-oriented, is actually a shortcut to some obviously object-oriented design principles. Keep this in mind as we discuss comprehensions, as they, too, appear to be the polar opposite of an object-oriented tool. Yet, they use the exact same iteration protocol as `for` loops and are just another kind of shortcut.

Comprehensions

Comprehensions are simple, but powerful, syntaxes that allow us to transform or filter an iterable object in as little as one line of code. The resultant object can be a perfectly normal list, set, or dictionary, or it can be a generator expression that can be efficiently consumed in one go.

List comprehensions

List comprehensions are one of the most powerful tools in Python, so people tend to think of them as advanced. They're not. Indeed, I've taken the liberty of littering previous examples with comprehensions and assuming you'd understand them. While it's true that advanced programmers use comprehensions a lot, it's not because they're advanced, it's because they're trivial, and handle some of the most common operations in software development.

Let's have a look at one of those common operations; namely, converting a list of items into a list of related items. Specifically, let's assume we just read a list of strings from a file, and now we want to convert it to a list of integers. We know every item in the list is an integer, and we want to do some activity (say, calculate an average) on those numbers. Here's one simple way to approach it:

```
input_strings = ['1', '5', '28', '131', '3']

output_integers = []
for num in input_strings:
    output_integers.append(int(num))
```

This works fine and it's only three lines of code. If you aren't used to comprehensions, you may not even think it looks ugly! Now, look at the same code using a list comprehension:

```
input_strings = ['1', '5', '28', '131', '3']output_integers =
[int(num) for num in input_strings]
```

We're down to one line and, importantly for performance, we've dropped an `append` method call for each item in the list. Overall, it's pretty easy to tell what's going on, even if you're not used to comprehension syntax.

The square brackets indicate, as always, that we're creating a list. Inside this list is a `for` loop that iterates over each item in the input sequence. The only thing that may be confusing is what's happening between the list's opening brace and the start of the `for` loop. Whatever happens here is applied to *each* of the items in the input list. The item in question is referenced by the `num` variable from the loop. So, it's converting each individual element to an `int` data type.

That's all there is to a basic list comprehension. They are not so advanced after all. Comprehensions are highly optimized C code; list comprehensions are far faster than `for` loops when looping over a huge number of items. If readability alone isn't a convincing reason to use them as much as possible, speed should be.

Converting one list of items into a related list isn't the only thing we can do with a list comprehension. We can also choose to exclude certain values by adding an `if` statement inside the comprehension. Have a look:

```
output_ints = [int(n) for n in input_strings if len(n) < 3]
```

I shortened the name of the variable from `num` to `n` and the result variable to `output_ints` so it would still fit on one line. Other than this, all that's different between this example and the previous one is the `if len(n) < 3` part. This extra code excludes any strings with more than two characters. The `if` statement is applied before the `int` function, so it's testing the length of a string. Since our input strings are all integers at heart, it excludes any number over 99. Now that is all there is to list comprehensions! We use them to map input values to output values, applying a filter along the way to include or exclude any values that meet a specific condition.

Any iterable can be the input to a list comprehension; anything we can wrap in a `for` loop can also be placed inside a comprehension. For example, text files are iterable; each call to `__next__` on the file's iterator will return one line of the file. We could load a tab delimited file where the first line is a header row into a dictionary using the `zip` function:

```
import sys
filename = sys.argv[1]

with open(filename) as file:
    header = file.readline().strip().split('\t')
    contacts = [
            dict(
                zip(header, line.strip().split('\t'))
                ) for line in file
            ]

for contact in contacts:
    print("email: {email} -- {last}, {first}".format(
        **contact))
```

This time, I've added some whitespace to make it somewhat more readable (list comprehensions don't *have* to fit on one line). This example creates a list of dictionaries from the zipped header and split lines for each line in the file.

Er, what? Don't worry if that code or explanation doesn't make sense; it's a bit confusing. One list comprehension is doing a pile of work here, and the code is hard to understand, read, and ultimately, maintain. This example shows that list comprehensions aren't always the best solution; most programmers would agree that a `for` loop would be more readable than this version.

Remember: the tools we are provided with should not be abused! Always pick the right tool for the job, which is always to write maintainable code.

Set and dictionary comprehensions

Comprehensions aren't restricted to lists. We can use a similar syntax with braces to create sets and dictionaries as well. Let's start with sets. One way to create a set is to wrap a list comprehension in the `set()` constructor, which converts it to a set. But why waste memory on an intermediate list that gets discarded, when we can create a set directly?

Here's an example that uses a named tuple to model author/title/genre triads, and then retrieves a set of all the authors that write in a specific genre:

```
from collections import namedtuple

Book = namedtuple("Book", "author title genre")
books = [
        Book("Pratchett", "Nightwatch", "fantasy"),
        Book("Pratchett", "Thief Of Time", "fantasy"),
        Book("Le Guin", "The Dispossessed", "scifi"),
        Book("Le Guin", "A Wizard Of Earthsea", "fantasy"),
        Book("Turner", "The Thief", "fantasy"),
        Book("Phillips", "Preston Diamond", "western"),
        Book("Phillips", "Twice Upon A Time", "scifi"),
        ]

fantasy_authors = {
        b.author for b in books if b.genre == 'fantasy'}
```

The highlighted set comprehension sure is short in comparison to the demo-data setup! If we were to use a list comprehension, of course, Terry Pratchett would have been listed twice.. As it is, the nature of sets removes the duplicates, and we end up with:

```
>>> fantasy_authors
{'Turner', 'Pratchett', 'Le Guin'}
```

We can introduce a colon to create a dictionary comprehension. This converts a sequence into a dictionary using *key: value* pairs. For example, it may be useful to quickly look up the author or genre in a dictionary if we know the title. We can use a dictionary comprehension to map titles to module objects:

```
fantasy_titles = {
        b.title: b for b in books if b.genre == 'fantasy'}
```

Now, we have a dictionary, and can look up books by title using the normal syntax.

In summary, comprehensions are not advanced Python, nor are they "non-object-oriented" tools that should be avoided. They are simply a more concise and optimized syntax for creating a list, set, or dictionary from an existing sequence.

Generator expressions

Sometimes we want to process a new sequence without placing a new list, set, or dictionary into system memory. If we're just looping over items one at a time, and don't actually care about having a final container object created, creating that container is a waste of memory. When processing one item at a time, we only need the current object stored in memory at any one moment. But when we create a container, all the objects have to be stored in that container before we start processing them.

For example, consider a program that processes log files. A very simple log might contain information in this format:

```
Jan 26, 2015 11:25:25    DEBUG        This is a debugging message.
Jan 26, 2015 11:25:36    INFO         This is an information method.
Jan 26, 2015 11:25:46    WARNING      This is a warning. It could be
serious.
Jan 26, 2015 11:25:52    WARNING      Another warning sent.
Jan 26, 2015 11:25:59    INFO         Here's some information.
Jan 26, 2015 11:26:13    DEBUG        Debug messages are only useful
if you want to figure something out.
Jan 26, 2015 11:26:32    INFO         Information is usually harmless,
but helpful.
Jan 26, 2015 11:26:40    WARNING      Warnings should be heeded.
Jan 26, 2015 11:26:54    WARNING      Watch for warnings.
```

Log files for popular web servers, databases, or e-mail servers can contain many gigabytes of data (I recently had to clean nearly 2 terabytes of logs off a misbehaving system). If we want to process each line in the log, we can't use a list comprehension; it would create a list containing every line in the file. This probably wouldn't fit in RAM and could bring the computer to its knees, depending on the operating system.

If we used a `for` loop on the log file, we could process one line at a time before reading the next one into memory. Wouldn't be nice if we could use comprehension syntax to get the same effect?

This is where generator expressions come in. They use the same syntax as comprehensions, but they don't create a final container object. To create a generator expression, wrap the comprehension in `()` instead of `[]` or `{}`.

The following code parses a log file in the previously presented format, and outputs a new log file that contains only the `WARNING` lines:

```
import sys

inname = sys.argv[1]
outname = sys.argv[2]

with open(inname) as infile:
    with open(outname, "w") as outfile:
        warnings = (l for l in infile if 'WARNING' in l)
        for l in warnings:
            outfile.write(l)
```

This program takes the two filenames on the command line, uses a generator expression to filter out the warnings (in this case, it uses the `if` syntax, and leaves the line unmodified), and then outputs the warnings to another file. If we run it on our sample file, the output looks like this:

```
Jan 26, 2015 11:25:46    WARNING     This is a warning. It could be
serious.
Jan 26, 2015 11:25:52    WARNING     Another warning sent.
Jan 26, 2015 11:26:40    WARNING     Warnings should be heeded.
Jan 26, 2015 11:26:54    WARNING     Watch for warnings.
```

Of course, with such a short input file, we could have safely used a list comprehension, but if the file is millions of lines long, the generator expression will have a huge impact on both memory and speed.

Generator expressions are frequently most useful inside function calls. For example, we can call `sum`, `min`, or `max`, on a generator expression instead of a list, since these functions process one object at a time. We're only interested in the result, not any intermediate container.

In general, a generator expression should be used whenever possible. If we don't actually need a list, set, or dictionary, but simply need to filter or convert items in a sequence, a generator expression will be most efficient. If we need to know the length of a list, or sort the result, remove duplicates, or create a dictionary, we'll have to use the comprehension syntax.

Generators

Generator expressions are actually a sort of comprehension too; they compress the more advanced (this time it really is more advanced!) generator syntax into one line. The greater generator syntax looks even less object-oriented than anything we've seen, but we'll discover that once again, it is a simple syntax shortcut to create a kind of object.

Let's take the log file example a little further. If we want to delete the `WARNING` column from our output file (since it's redundant: this file contains only warnings), we have several options, at various levels of readability. We can do it with a generator expression:

```
import sys
inname, outname = sys.argv[1:3]

with open(inname) as infile:
    with open(outname, "w") as outfile:
        warnings = (l.replace('\tWARNING', '')
                for l in infile if 'WARNING' in l)
        for l in warnings:
            outfile.write(l)
```

That's perfectly readable, though I wouldn't want to make the expression much more complicated than that. We could also do it with a normal `for` loop:

```
import sys
inname, outname = sys.argv[1:3]

with open(inname) as infile:
    with open(outname, "w") as outfile:
        for l in infile:
            if 'WARNING' in l:
                outfile.write(l.replace('\tWARNING', ''))
```

That's maintainable, but so many levels of indent in so few lines is kind of ugly. More alarmingly, if we wanted to do something different with the lines, rather than just printing them out, we'd have to duplicate the looping and conditional code, too. Now let's consider a truly object-oriented solution, without any shortcuts:

```
import sys
inname, outname = sys.argv[1:3]

class WarningFilter:
    def __init__(self, insequence):
```

```
            self.insequence = insequence
        def __iter__(self):
            return self
        def __next__(self):
            l = self.insequence.readline()
            while l and 'WARNING' not in l:
                l = self.insequence.readline()
            if not l:
                raise StopIteration
            return l.replace('\tWARNING', '')

    with open(inname) as infile:
        with open(outname, "w") as outfile:
            filter = WarningFilter(infile)
            for l in filter:
                outfile.write(l)
```

No doubt about it: that is so ugly and difficult to read that you may not even be able to tell what's going on. We created an object that takes a file object as input, and provides a `__next__` method like any iterator.

This `__next__` method reads lines from the file, discarding them if they are not `WARNING` lines. When it encounters a `WARNING` line, it returns it. Then the `for` loop will call `__next__` again to process the next `WARNING` line. When we run out of lines, we raise `StopIteration` to tell the loop we're finished iterating. It's pretty ugly compared to the other examples, but it's also powerful; now that we have a class in our hands, we can do whatever we want with it.

With that background behind us, we finally get to see generators in action. This next example does *exactly* the same thing as the previous one: it creates an object with a `__next__` method that raises `StopIteration` when it's out of inputs:

```
    import sys
    inname, outname = sys.argv[1:3]

    def warnings_filter(insequence):
        for l in insequence:
            if 'WARNING' in l:
                yield l.replace('\tWARNING', '')

    with open(inname) as infile:
        with open(outname, "w") as outfile:
```

```
filter = warnings_filter(infile)
for l in filter:
    outfile.write(l)
```

OK, that's pretty readable, maybe... at least it's short. But what on earth is going on here, it makes no sense whatsoever. And what is `yield`, anyway?

In fact, `yield` is the key to generators. When Python sees `yield` in a function, it takes that function and wraps it up in an object not unlike the one in our previous example. Think of the `yield` statement as similar to the `return` statement; it exits the function and returns a line. Unlike `return`, however, when the function is called again (via `next()`), it will start where it left off—on the line after the `yield` statement—instead of at the beginning of the function. In this example, there is no line "after" the `yield` statement, so it jumps to the next iteration of the `for` loop. Since the `yield` statement is inside an `if` statement, it only yields lines that contain `WARNING`.

While it looks like this is just a function looping over the lines, it is actually creating a special type of object, a generator object:

```
>>> print(warnings_filter([]))
<generator object warnings_filter at 0xb728c6bc>
```

I passed an empty list into the function to act as an iterator. All the function does is create and return a generator object. That object has `__iter__` and `__next__` methods on it, just like the one we created in the previous example. Whenever `__next__` is called, the generator runs the function until it finds a `yield` statement. It then returns the value from `yield`, and the next time `__next__` is called, it picks up where it left off.

This use of generators isn't that advanced, but if you don't realize the function is creating an object, it can seem like magic. This example was quite simple, but you can get really powerful effects by making multiple calls to `yield` in a single function; the generator will simply pick up at the most recent `yield` and continue to the next one.

Yield items from another iterable

Often, when we build a generator function, we end up in a situation where we want to yield data from another iterable object, possibly a list comprehension or generator expression we constructed inside the generator, or perhaps some external items that were passed into the function. This has always been possible by looping over the iterable and individually yielding each item. However, in Python version 3.3, the Python developers introduced a new syntax to make this a little more elegant.

Let's adapt the generator example a bit so that instead of accepting a sequence of lines, it accepts a filename. This would normally be frowned upon as it ties the object to a particular paradigm. When possible we should operate on iterators as input; this way the same function could be used regardless of whether the log lines came from a file, memory, or a web-based log aggregator. So the following example is contrived for pedagogical reasons.

This version of the code illustrates that your generator can do some basic setup before yielding information from another iterable (in this case, a generator expression):

```
import sys
inname, outname = sys.argv[1:3]

def warnings_filter(infilename):
    with open(infilename) as infile:
        yield from (
            l.replace('\tWARNING', '')
            for l in infile
            if 'WARNING' in l
        )

filter = warnings_filter(inname)
with open(outname, "w") as outfile:
    for l in filter:
        outfile.write(l)
```

This code combines the `for` loop from the previous example into a generator expression. Notice how I put the three clauses of the generator expression (the transformation, the loop, and the filter) on separate lines to make them more readable. Notice also that this transformation didn't help enough; the previous example with a `for` loop was more readable.

So let's consider an example that is more readable than its alternative. It can be useful to construct a generator that yields data from multiple other generators. The `itertools.chain` function, for example, yields data from iterables in sequence until they have all been exhausted. This can be implemented far too easily using the `yield from` syntax, so let's consider a classic computer science problem: walking a general tree.

A common implementation of the general tree data structure is a computer's filesystem. Let's model a few folders and files in a Unix filesystem so we can use `yield from` to walk them effectively:

```
class File:
    def __init__(self, name):
```

```
        self.name = name

class Folder(File):
    def __init__(self, name):
        super().__init__(name)
        self.children = []

root = Folder('')
etc = Folder('etc')
root.children.append(etc)
etc.children.append(File('passwd'))
etc.children.append(File('groups'))
httpd = Folder('httpd')
etc.children.append(httpd)
httpd.children.append(File('http.conf'))
var = Folder('var')
root.children.append(var)
log = Folder('log')
var.children.append(log)
log.children.append(File('messages'))
log.children.append(File('kernel'))
```

This setup code looks like a lot of work, but in a real filesystem, it would be even more involved. We'd have to read data from the hard drive and structure it into the tree. Once in memory, however, the code that outputs every file in the filesystem is quite elegant:

```
def walk(file):
    if isinstance(file, Folder):
        yield file.name + '/'
        for f in file.children:
            yield from walk(f)
    else:
        yield file.name
```

If this code encounters a directory, it recursively asks `walk()` to generate a list of all files subordinate to each of its children, and then yields all that data plus its own filename. In the simple case that it has encountered a normal file, it just yields that name.

As an aside, solving the preceding problem without using a generator is tricky enough that this problem is a common interview question. If you answer it as shown like this, be prepared for your interviewer to be both impressed and somewhat irritated that you answered it so easily. They will likely demand that you explain exactly what is going on. Of course, armed with the principles you've leaned in this chapter, you won't have any problem.

The `yield from` syntax is a useful shortcut when writing chained generators, but it is more commonly used for a different purpose: piping data through coroutines. We'll see many examples of this in *Chapter 13*, *Concurrency*, but for now, let's discover what a coroutine is.

Coroutines

Coroutines are extremely powerful constructs that are often confused with generators. Many authors inappropriately describe coroutines as "generators with a bit of extra syntax." This is an easy mistake to make, as, way back in Python 2.5, when coroutines were introduced, they were presented as "we added a `send` method to the generator syntax." This is further complicated by the fact that when you create a coroutine in Python, the object returned is a generator. The difference is actually a lot more nuanced and will make more sense after you've seen a few examples.

While coroutines in Python are currently tightly coupled to the generator syntax, they are only superficially related to the iterator protocol we have been discussing. The upcoming (as this is published) Python 3.5 release makes coroutines a truly standalone object and will provide a new syntax to work with them.

The other thing to bear in mind is that coroutines are pretty hard to understand. They are not used all that often in the wild, and you could likely skip this section and happily develop in Python for years without missing or even encountering them. There are a couple libraries that use coroutines extensively (mostly for concurrent or asynchronous programming), but they are normally written such that you can use coroutines without actually understanding how they work! So if you get lost in this section, don't despair.

But you won't get lost, having studied the following examples. Here's one of the simplest possible coroutines; it allows us to keep a running tally that can be increased by arbitrary values:

```
def tally():
    score = 0
```

```
        while True:
            increment = yield score
            score += increment
```

This code looks like black magic that couldn't possibly work, so we'll see it working before going into a line-by-line description. This simple object could be used by a scoring application for a baseball team. Separate tallies could be kept for each team, and their score could be incremented by the number of runs accumulated at the end of every half-inning. Look at this interactive session:

```
>>> white_sox = tally()
>>> blue_jays = tally()
>>> next(white_sox)
0
>>> next(blue_jays)
0
>>> white_sox.send(3)
3
>>> blue_jays.send(2)
2
>>> white_sox.send(2)
5
>>> blue_jays.send(4)
6
```

First we construct two `tally` objects, one for each team. Yes, they look like functions, but as with the generator objects in the previous section, the fact that there is a `yield` statement inside the function tells Python to put a great deal of effort into turning the simple function into an object.

We then call `next()` on each of the coroutine objects. This does the same thing as calling next on any generator, which is to say, it executes each line of code until it encounters a `yield` statement, returns the value at that point, and then *pauses* until the next `next()` call.

So far, then, there's nothing new. But look back at the `yield` statement in our coroutine:

```
    increment = yield score
```

Unlike with generators, this yield function looks like it's supposed to return a value and assign it to a variable. This is, in fact, exactly what's happening. The coroutine is still paused at the `yield` statement and waiting to be activated again by another call to `next()`.

Or rather, as you see in the interactive session, a call to a method called `send()`. The `send()` method does *exactly* the same thing as `next()` except that in addition to advancing the generator to the next `yield` statement. It also allows you to pass in a value from outside the generator. This value is assigned to the left side of the `yield` statement.

The thing that is really confusing for many people is the order in which this happens:

- `yield` occurs and the generator pauses
- `send()` occurs from outside the function and the generator wakes up
- The value sent in is assigned to the left side of the `yield` statement
- The generator continues processing until it encounters another `yield` statement

So, in this particular example, after we construct the coroutine and advance it to the `yield` statement with a call to `next()`, each successive call to `send()` passes a value into the coroutine, which adds this value to its score, goes back to the top of the `while` loop, and keeps processing until it hits the `yield` statement. The `yield` statement returns a value, and this value becomes the return value of the most recent call to `send`. Don't miss that: the `send()` method does not just submit a value to the generator, it also returns the value from the upcoming `yield` statement, just like `next()`. This is how we define the difference between a generator and a coroutine: a generator only produces values, while a coroutine can also consume them.

The behavior and syntax of `next(i)`, `i.__next__()`, and `i.send(value)` are rather unintuitive and frustrating. The first is a normal function, the second is a special method, and the last is a normal method. But all three do the same thing: advance the generator until it yields a value and pause. Further, the `next()` function and associated method can be replicated by calling `i.send(None)`. There is value to having two different method names here, since it helps the reader of our code easily see whether they are interacting with a coroutine or a generator. I just find the fact that in one case it's a function call and in the other it's a normal method somewhat irritating.

Back to log parsing

Of course, the previous example could easily have been coded using a couple integer variables and calling `x += increment` on them. Let's look at a second example where coroutines actually save us some code. This example is a somewhat simplified (for pedagogical reasons) version of a problem I had to solve in my real job. The fact that it logically follows from the earlier discussions about processing a log file is completely serendipitous; those examples were written for the first edition of this module, whereas this problem came up four years later!

The Linux kernel log contains lines that look somewhat, but not quite completely, unlike this:

```
unrelated log messages
sd 0:0:0:0 Attached Disk Drive
unrelated log messages
sd 0:0:0:0 (SERIAL=ZZ12345)
unrelated log messages
sd 0:0:0:0 [sda] Options
unrelated log messages
XFS ERROR [sda]
unrelated log messages
sd 2:0:0:1 Attached Disk Drive
unrelated log messages
sd 2:0:0:1 (SERIAL=ZZ67890)
unrelated log messages
sd 2:0:0:1 [sdb] Options
unrelated log messages
sd 3:0:1:8 Attached Disk Drive
unrelated log messages
sd 3:0:1:8 (SERIAL=WW11111)
unrelated log messages
sd 3:0:1:8 [sdc] Options
unrelated log messages
XFS ERROR [sdc]
unrelated log messages
```

There are a whole bunch of interspersed kernel log messages, some of which pertain to hard disks. The hard disk messages might be interspersed with other messages, but they occur in a predictable format and order, in which a specific drive with a known serial number is associated with a bus identifier (such as `0:0:0:0`), and a block device identifier (such as `sda`) is associated with that bus. Finally, if the drive has a corrupt filesystem, it might fail with an XFS error.

Now, given the preceding log file, the problem we need to solve is how to obtain the serial number of any drives that have XFS errors on them. This serial number might later be used by a data center technician to identify and replace the drive.

We know we can identify the individual lines using regular expressions, but we'll have to change the regular expressions as we loop through the lines, since we'll be looking for different things depending on what we found previously. The other difficult bit is that if we find an error string, the information about which bus contains that string, and what serial number is attached to the drive on that bus has already been processed. This can easily be solved by iterating through the lines of the file in reverse order.

Before you look at this example, be warned—the amount of code required for a coroutine-based solution is scarily small:

```
import re

def match_regex(filename, regex):
    with open(filename) as file:
        lines = file.readlines()
    for line in reversed(lines):
        match = re.match(regex, line)
        if match:
            regex = yield match.groups()[0]

def get_serials(filename):
    ERROR_RE = 'XFS ERROR (\[sd[a-z]\])'
    matcher = match_regex(filename, ERROR_RE)
    device = next(matcher)
    while True:
        bus = matcher.send(
            '(sd \S+) {}.*'.format(re.escape(device)))
        serial = matcher.send('{} \(SERIAL=([^)]*)\)'.format(bus))
        yield serial
        device = matcher.send(ERROR_RE)

for serial_number in get_serials('EXAMPLE_LOG.log'):
    print(serial_number)
```

This code neatly divides the job into two separate tasks. The first task is to loop over all the lines and spit out any lines that match a given regular expression. The second task is to interact with the first task and give it guidance as to what regular expression it is supposed to be searching for at any given time.

Look at the `match_regex` coroutine first. Remember, it doesn't execute any code when it is constructed; rather, it just creates a coroutine object. Once constructed, someone outside the coroutine will eventually call `next()` to start the code running, at which point it stores the state of two variables, `filename` and `regex`. It then reads all the lines in the file and iterates over them in reverse. Each line is compared to the regular expression that was passed in until it finds a match. When the match is found, the coroutine yields the first group from the regular expression and waits.

At some point in the future, other code will send in a new regular expression to search for. Note that the coroutine never cares what regular expression it is trying to match; it's just looping over lines and comparing them to a regular expression. It's somebody else's responsibility to decide what regular expression to supply.

In this case, that somebody else is the `get_serials` generator. It doesn't care about the lines in the file, in fact it isn't even aware of them. The first thing it does is create a `matcher` object from the `match_regex` coroutine constructor, giving it a default regular expression to search for. It advances the coroutine to its first `yield` and stores the value it returns. It then goes into a loop that instructs the matcher object to search for a bus ID based on the stored device ID, and then a serial number based on that bus ID.

It idly yields that serial number to the outside `for` loop before instructing the matcher to find another device ID and repeat the cycle.

Basically, the coroutine's (`match_regex`, as it uses the `regex = yield` syntax) job is to search for the next important line in the file, while the generator's (`get_serial`, which uses the `yield` syntax without assignment) job is to decide which line is important. The generator has information about this particular problem, such as what order lines will appear in the file. The coroutine, on the other hand, could be plugged into any problem that required searching a file for given regular expressions.

Closing coroutines and throwing exceptions

Normal generators signal their exit from inside by raising `StopIteration`. If we chain multiple generators together (for example, by iterating over one generator from inside another), the `StopIteration` exception will be propagated outward. Eventually, it will hit a `for` loop that will see the exception and know that it's time to exit the loop.

Coroutines don't normally follow the iteration mechanism; rather than pulling data through one until an exception is encountered, data is usually pushed into it (using `send`). The entity doing the pushing is normally the one in charge of telling the coroutine when it's finished; it does this by calling the `close()` method on the coroutine in question.

When called, the `close()` method will raise a `GeneratorExit` exception at the point the coroutine was waiting for a value to be sent in. It is normally good policy for coroutines to wrap their `yield` statements in a `try...finally` block so that any cleanup tasks (such as closing associated files or sockets) can be performed.

If we need to raise an exception inside a coroutine, we can use the `throw()` method in a similar way. It accepts an exception type with optional `value` and `traceback` arguments. The latter is useful when we encounter an exception in one coroutine and want to cause an exception to occur in an adjacent coroutine while maintaining the traceback.

Both of these features are vital if you're building robust coroutine-based libraries, but we are unlikely to encounter them in our day-to-day coding lives.

The relationship between coroutines, generators, and functions

We've seen coroutines in action, so now let's go back to that discussion of how they are related to generators. In Python, as is so often the case, the distinction is quite blurry. In fact, all coroutines are generator objects, and authors often use the two terms interchangeably. Sometimes, they describe coroutines as a subset of generators (only generators that return values from yield are considered coroutines). This is technically true in Python, as we've seen in the previous sections.

However, in the greater sphere of theoretical computer science, coroutines are considered the more general principles, and generators are a specific type of coroutine. Further, normal functions are yet another distinct subset of coroutines.

A coroutine is a routine that can have data passed in at one or more points and get it out at one or more points. In Python, the point where data is passed in and out is the `yield` statement.

A function, or subroutine, is the simplest type of coroutine. You can pass data in at one point, and get data out at one other point when the function returns. While a function can have multiple `return` statements, only one of them can be called for any given invocation of the function.

Finally, a generator is a type of coroutine that can have data passed in at one point, but can pass data out at multiple points. In Python, the data would be passed out at a `yield` statement, but you can't pass data back in. If you called `send`, the data would be silently discarded.

So in theory, generators are types of coroutines, functions are types of coroutines, and there are coroutines that are neither functions nor generators. That's simple enough, eh? So why does it feel more complicated in Python?

In Python, generators and coroutines are both constructed using a syntax that looks like we are constructing a function. But the resulting object is not a function at all; it's a totally different kind of object. Functions are, of course, also objects. But they have a different interface; functions are callable and return values, generators have data pulled out using `next()`, and coroutines have data pushed in using `send`.

Case study

One of the fields in which Python is the most popular these days is data science. Let's implement a basic machine learning algorithm! Machine learning is a huge topic, but the general idea is to make predictions or classifications about future data by using knowledge gained from past data. Uses of such algorithms abound, and data scientists are finding new ways to apply machine learning every day. Some important machine learning applications include computer vision (such as image classification or facial recognition), product recommendation, identifying spam, and speech recognition. We'll look at a simpler problem: given an RGB color definition, what name would humans identify that color as?

There are more than 16 million colors in the standard RGB color space, and humans have come up with names for only a fraction of them. While there are thousands of names (some quite ridiculous; just go to any car dealership or makeup store), let's build a classifier that attempts to divide the RGB space into the basic colors:

- Red
- Purple
- Blue
- Green
- Yellow
- Orange
- Grey
- White
- Pink

The first thing we need is a dataset to train our algorithm on. In a production system, you might scrape a *list of colors* website or survey thousands of people. Instead, I created a simple application that renders a random color and asks the user to select one of the preceding nine options to classify it. This application is included with the example code for this chapter in the `kivy_color_classifier` directory, but we won't be going into the details of this code as its only purpose here is to generate sample data.

Kivy has an incredibly well-engineered object-oriented API that you may want to explore on your own time. If you would like to develop graphical programs that run on many systems, from your laptop to your cell phone, you might want to check out my module, *Creating Apps In Kivy*, *O'Reilly*.

For the purposes of this case study, the important thing about that application is the output, which is a **comma-separated value** (**CSV**) file that contains four values per row: the red, green, and blue values (represented as a floating-point number between zero and one), and one of the preceding nine names that the user assigned to that color. The dataset looks something like this:

```
0.30928279150905513,0.7536768153744394,0.3244011790604804,Green
0.4991001855115986,0.6394567277907686,0.6340502030888825,Grey
0.21132621004927998,0.3307376167520666,0.704037576789711,Blue
0.7260420945787928,0.4025279573860123,0.49781705131696363,Pink
0.706469868610228,0.28530423638868196,0.7880240251003464,Purple
0.692243900051664,0.7053550777777416,0.1845069151913028,Yellow
0.3628979381122397,0.11079495501215897,0.26924540840045075,Purple
0.611273677646518,0.48798521783547677,0.5346130557761224,Purple
.
.
.
0.4014121109376566,0.42176706818252674,0.9601866228083298,Blue
0.17750449496124632,0.8008214961070862,0.5073944321437429,Green
```

I made 200 datapoints (a very few of them untrue) before I got bored and decided it was time to start machine learning on this dataset. These datapoints are shipped with the examples for this chapter if you would like to use my data (nobody's ever told me I'm color-blind, so it should be somewhat reasonable).

We'll be implementing one of the simpler machine-learning algorithms, referred to as k-nearest neighbor. This algorithm relies on some kind of "distance" calculation between points in the dataset (in our case, we can use a three-dimensional version of the Pythagorean theorem). Given a new datapoint, it finds a certain number (referred to as k, as in k-nearest neighbors) of datapoints that are closest to it when measured by that distance calculation. Then it combines those datapoints in some way (an average might work for linear calculations; for our classification problem, we'll use the mode), and returns the result.

We won't go into too much detail about what the algorithm does; rather, we'll focus on some of the ways we can apply the iterator pattern or iterator protocol to this problem.

Let's now write a program that performs the following steps in order:

1. Load the sample data from the file and construct a model from it.
2. Generate 100 random colors.
3. Classify each color and output it to a file in the same format as the input.

Once we have this second CSV file, another Kivy program can load the file and render each color, asking a human user to confirm or deny the accuracy of the prediction, thus informing us of how accurate our algorithm and initial data set are.

The first step is a fairly simple generator that loads CSV data and converts it into a format that is amenable to our needs:

```
import csv

dataset_filename = 'colors.csv'

def load_colors(filename):
    with open(filename) as dataset_file:
        lines = csv.reader(dataset_file)
        for line in lines:
            yield tuple(float(y) for y in line[0:3]), line[3]
```

We haven't seen the `csv.reader` function before. It returns an iterator over the lines in the file. Each value returned by the iterator is a list of strings. In our case, we could have just split on commas and been fine, but `csv.reader` also takes care of managing quotation marks and various other nuances of the comma-separated value format.

We then loop over these lines and convert them to a tuple of color and name, where the color is a tuple of three floating value integers. This tuple is constructed using a generator expression. There might be more readable ways to construct this tuple; do you think the code brevity and the speed of a generator expression is worth the obfuscation? Instead of returning a list of color tuples, it yields them one at a time, thus constructing a generator object.

Now, we need a hundred random colors. There are so many ways this can be done:

- A list comprehension with a nested generator expression: `[tuple(random() for r in range(3)) for r in range(100)]`
- A basic generator function
- A class that implements the `__iter__` and `__next__` protocols

- Push the data through a pipeline of coroutines
- Even just a basic `for` loop

The generator version seems to be most readable, so let's add that function to our program:

```
from random import random


def generate_colors(count=100):
    for i in range(count):
        yield (random(), random(), random())
```

Notice how we parameterize the number of colors to generate. We can now reuse this function for other color-generating tasks in the future.

Now, before we do the classification step, we need a function to calculate the "distance" between two colors. Since it's possible to think of colors as being three dimensional (red, green, and blue could map to *x*, *y*, and *z* axes, for example), let's use a little basic math:

```
import math

def color_distance(color1, color2):
    channels = zip(color1, color2)
    sum_distance_squared = 0
    for c1, c2 in channels:
        sum_distance_squared += (c1 - c2) ** 2
    return math.sqrt(sum_distance_squared)
```

This is a pretty basic-looking function; it doesn't look like it's even using the iterator protocol. There's no `yield` function, no comprehensions. However, there is a `for` loop, and that call to the `zip` function is doing some real iteration as well (remember that `zip` yields tuples containing one element from each input iterator).

Note, however, that this function is going to be called a lot of times inside our k-nearest neighbors algorithm. If our code ran too slow and we were able to identify this function as a bottleneck, we might want to replace it with a less readable, but more optimized, generator expression:

```
def color_distance(color1, color2):
    return math.sqrt(sum((x[0] - x[1]) ** 2 for x in zip(
    color1, color2)))
```

However, I strongly recommend not making such optimizations until you have proven that the readable version is too slow.

Now that we have some plumbing in place, let's do the actual k-nearest neighbor implementation. This seems like a good place to use a coroutine. Here it is with some test code to ensure it's yielding sensible values:

```
def nearest_neighbors(model_colors, num_neighbors):
    model = list(model_colors)
    target = yield
    while True:
        distances = sorted(
            ((color_distance(c[0], target), c) for c in model),
        )
        target = yield [
            d[1] for d in distances[0:num_neighbors]
        ]


model_colors = load_colors(dataset_filename)
target_colors = generate_colors(3)
get_neighbors = nearest_neighbors(model_colors, 5)
next(get_neighbors)

for color in target_colors:
    distances = get_neighbors.send(color)
    print(color)
    for d in distances:
        print(color_distance(color, d[0]), d[1])
```

The coroutine accepts two arguments, the list of colors to be used as a model, and the number of neighbors to query. It converts the model to a list because it's going to be iterated over multiple times. In the body of the coroutine, it accepts a tuple of RGB color values using the `yield` syntax. Then it combines a call to `sorted` with an odd generator expression. See if you can figure out what that generator expression is doing.

It returns a tuple of `(distance, color_data)` for each color in the model. Remember, the model itself contains tuples of `(color, name)`, where `color` is a tuple of three RGB values. Therefore, the generator is returning an iterator over a weird data structure that looks like this:

```
(distance, (r, g, b), color_name)
```

The `sorted` call then sorts the results by their first element, which is distance. This is a complicated piece of code and isn't object-oriented at all. You may want to break it down into a normal `for` loop to ensure you understand what the generator expression is doing. It might also be a good exercise to imagine how this code would look if you were to pass a key argument into the sorted function instead of constructing a tuple.

The `yield` statement is a bit less complicated; it pulls the second value from each of the first k `(distance, color_data)` tuples. In more concrete terms, it yields the `((r, g, b), color_name)` tuple for the k values with the lowest distance. Or, if you prefer more abstract terms, it yields the target's k-nearest neighbors in the given model.

The remaining code is just boilerplate to test this method; it constructs the model and a color generator, primes the coroutine, and prints the results in a `for` loop.

The two remaining tasks are to choose a color based on the nearest neighbors, and to output the results to a CSV file. Let's make two more coroutines to take care of these tasks. Let's do the output first because it can be tested independently:

```
def write_results(filename="output.csv"):
    with open(filename, "w") as file:
        writer = csv.writer(file)
        while True:
            color, name = yield
            writer.writerow(list(color) + [name])

results = write_results()
next(results)
for i in range(3):
    print(i)
    results.send(((i, i, i), i * 10))
```

This coroutine maintains an open file as state and writes lines of code to it as they are sent in using `send()`. The test code ensures the coroutine is working correctly, so now we can connect the two coroutines with a third one.

The second coroutine uses a bit of an odd trick:

```
from collections import Counter
def name_colors(get_neighbors):
    color = yield
    while True:
        near = get_neighbors.send(color)
        name_guess = Counter(
            n[1] for n in near).most_common(1)[0][0]
        color = yield name_guess
```

This coroutine accepts, *as its argument*, an existing coroutine. In this case, it's an instance of `nearest_neighbors`. This code basically proxies all the values sent into it through that `nearest_neighbors` instance. Then it does some processing on the result to get the most common color out of the values that were returned. In this case, it would probably make just as much sense to adapt the original coroutine to return a name, since it isn't being used for anything else. However, there are many cases where it is useful to pass coroutines around; this is how we do it.

Now all we have to do is connect these various coroutines and pipelines together, and kick off the process with a single function call:

```
def process_colors(dataset_filename="colors.csv"):
    model_colors = load_colors(dataset_filename)
    get_neighbors = nearest_neighbors(model_colors, 5)
    get_color_name = name_colors(get_neighbors)
    output = write_results()
    next(output)
    next(get_neighbors)
    next(get_color_name)

    for color in generate_colors():
        name = get_color_name.send(color)
        output.send((color, name))

process_colors()
```

So, this function, unlike almost every other function we've defined, is a perfectly normal function without any `yield` statements. It doesn't get turned into a coroutine or generator object. It does, however, construct a generator and three coroutines. Notice how the `get_neighbors` coroutine is passed into the constructor for `name_colors`? Pay attention to how all three coroutines are advanced to their first `yield` statements by calls to `next`.

Once all the pipes are created, we use a `for` loop to send each of the generated colors into the `get_color_name` coroutine, and then we pipe each of the values yielded by that coroutine to the output coroutine, which writes it to a file.

And that's it! I created a second Kivy app that loads the resulting CSV file and presents the colors to the user. The user can select either *Yes* or *No* depending on whether they think the choice made by the machine-learning algorithm matches the choice they would have made. This is not scientifically accurate (it's ripe for observation bias), but it's good enough for playing around. Using my eyes, it succeeded about 84 percent of the time, which is better than my grade 12 average. Not bad for our first ever machine learning experience, eh?

You might be wondering, "what does this have to do with object-oriented programming? There isn't even one class in this code!". In some ways, you'd be right; neither coroutines nor generators are commonly considered object-oriented. However, the functions that create them return objects; in fact, you could think of those functions as constructors. The constructed object has appropriate `send()` and `__next__()` methods. Basically, the coroutine/generator syntax is a syntax shortcut for a particular kind of object that would be quite verbose to create without it.

This case study has been an exercise in bottom-up design. We created various low-level objects that did specific tasks and hooked them all together at the end. I find this to be a common practice when developing with coroutines. The alternative, top-down design sometimes results in more monolithic pieces of code instead of unique individual pieces. In general, we want to find a happy medium between methods that are too large and methods that are too small and it's hard to see how they fit together. This is true, of course, regardless of whether the iterator protocol is being used as we did here.

Your Coding Challenge

If you don't use comprehensions in your daily coding very often, the first thing you should do is search through some existing code and find some for loops. See if any of them can be trivially converted to a generator expression or a list, set, or dictionary comprehension.

Test the claim that list comprehensions are faster than for loops. This can be done with the built-in timeit module. Use the help documentation for the timeit. timeit function to find out how to use it. Basically, write two functions that do the same thing, one using a list comprehension, and one using a for loop. Pass each function into timeit.timeit, and compare the results. If you're feeling adventurous, compare generators and generator expressions as well. Testing code using timeit can become addictive, so bear in mind that code does not need to be hyperfast unless it's being executed an immense number of times, such as on a huge input list or file.

Ankita Thakur

Your Course Guide

Play around with generator functions. Start with basic iterators that require multiple values (mathematical sequences are canonical examples; the Fibonacci sequence is overused if you can't think of anything better). Try some more advanced generators that do things like take multiple input lists and somehow yield values that merge them. Generators can also be used on files; can you write a simple generator that shows those lines that are identical in two files?

Coroutines abuse the iterator protocol but don't actually fulfill the iterator pattern. Can you build a non-coroutine version of the code that gets a serial number from a log file? Take an object-oriented approach so that you can store an additional state on a class. You'll learn a lot about coroutines if you can create an object that is a drop-in replacement for the existing coroutine.

See if you can abstract the coroutines used in the case study so that the k-nearest-neighbor algorithm can be used on a variety of datasets. You'll likely want to construct a coroutine that accepts other coroutines or functions that do the distance and recombination calculations as parameters, and then calls into those functions to find the actual nearest neighbors.

Summary of Module 1 Chapter 10

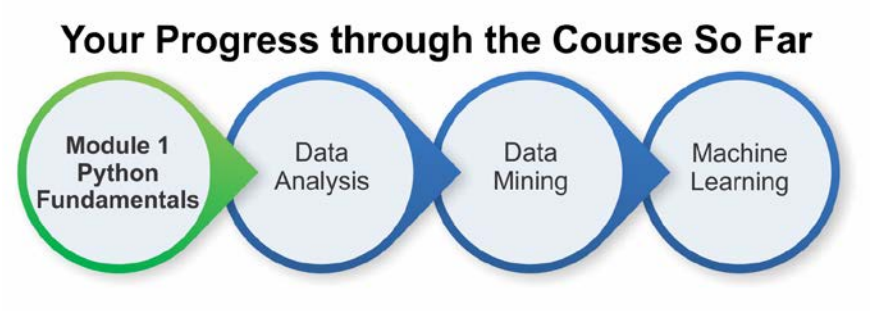
Ankita Thakur

Your Course Guide

In this chapter, we learned that design patterns are useful abstractions that provide "best practice" solutions for common programming problems. We covered our first design pattern, the iterator, as well as numerous ways that Python uses and abuses this pattern for its own nefarious purposes. The original iterator pattern is extremely object-oriented, but it is also rather ugly and verbose to code around. However, Python's built-in syntax abstracts the ugliness away, leaving us with a clean interface to these object-oriented constructs.

Comprehensions and generator expressions can combine container construction with iteration in a single line. Generator objects can be constructed using the yield syntax. Coroutines look like generators on the outside but serve a much different purpose.

We'll cover several more design patterns in the next two chapters. So stay connected!



11
Python Design Patterns I

In the last chapter, we were briefly introduced to design patterns, and covered the iterator pattern, a pattern so useful and common that it has been abstracted into the core of the programming language itself. In this chapter, we'll be reviewing other common patterns, and how they are implemented in Python. As with iteration, Python often provides an alternative syntax to make working with such problems simpler. We will cover both the "traditional" design, and the Python version for these patterns. In summary, we'll see:

- Numerous specific patterns
- A canonical implementation of each pattern in Python
- Python syntax to replace certain patterns

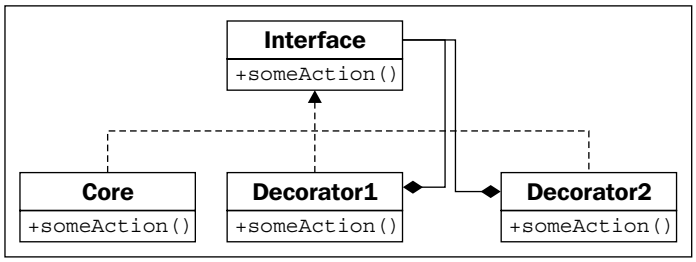
The decorator pattern

The decorator pattern allows us to "wrap" an object that provides core functionality with other objects that alter this functionality. Any object that uses the decorated object will interact with it in exactly the same way as if it were undecorated (that is, the interface of the decorated object is identical to that of the core object).

There are two primary uses of the decorator pattern:

- Enhancing the response of a component as it sends data to a second component
- Supporting multiple optional behaviors

The second option is often a suitable alternative to multiple inheritance. We can construct a core object, and then create a decorator around that core. Since the decorator object has the same interface as the core object, we can even wrap the new object in other decorators. Here's how it looks in UML:



Here, **Core** and all the decorators implement a specific **Interface**. The decorators maintain a reference to another instance of that **Interface** via composition. When called, the decorator does some added processing before or after calling its wrapped interface. The wrapped object may be another decorator, or the core functionality. While multiple decorators may wrap each other, the object in the "center" of all those decorators provides the core functionality.

A decorator example

Let's look at an example from network programming. We'll be using a TCP socket. The `socket.send()` method takes a string of input bytes and outputs them to the receiving socket at the other end. There are plenty of libraries that accept sockets and access this function to send data on the stream. Let's create such an object; it will be an interactive shell that waits for a connection from a client and then prompts the user for a string response:

```
import socket

def respond(client):
    response = input("Enter a value: ")
    client.send(bytes(response, 'utf8'))
    client.close()

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('localhost',2401))
server.listen(1)
try:
    while True:
```

```
            client, addr = server.accept()
            respond(client)
    finally:
        server.close()
```

The `respond` function accepts a socket parameter and prompts for data to be sent as a reply, then sends it. To use it, we construct a server socket and tell it to listen on port `2401` (I picked the port randomly) on the local computer. When a client connects, it calls the `respond` function, which requests data interactively and responds appropriately. The important thing to notice is that the `respond` function only cares about two methods of the socket interface: `send` and `close`. To test this, we can write a very simple client that connects to the same port and outputs the response before exiting:

```
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('localhost', 2401))
print("Received: {0}".format(client.recv(1024)))
client.close()
```

To use these programs:

1. Start the server in one terminal.
2. Open a second terminal window and run the client.
3. At the **Enter a value:** prompt in the server window, type a value and press enter.
4. The client will receive what you typed, print it to the console, and exit. Run the client a second time; the server will prompt for a second value.

Now, looking again at our server code, we see two sections. The `respond` function sends data into a socket object. The remaining script is responsible for creating that socket object. We'll create a pair of decorators that customize the socket behavior without having to extend or modify the socket itself.

Let's start with a "logging" decorator. This object outputs any data being sent to the server's console before it sends it to the client:

```
class LogSocket:
    def __init__(self, socket):
        self.socket = socket

    def send(self, data):
```

```
            print("Sending {0} to {1}".format(
                data, self.socket.getpeername()[0]))
            self.socket.send(data)

        def close(self):
            self.socket.close()
```

This class decorates a socket object and presents the `send` and `close` interface to client sockets. A better decorator would also implement (and possibly customize) all of the remaining socket methods. It should properly implement all of the arguments to `send`, (which actually accepts an optional flags argument) as well, but let's keep our example simple! Whenever `send` is called on this object, it logs the output to the screen before sending data to the client using the original socket.

We only have to change one line in our original code to use this decorator. Instead of calling `respond` with the socket, we call it with a decorated socket:

```
respond(LogSocket(client))
```

While that's quite simple, we have to ask ourselves why we didn't just extend the socket class and override the `send` method. We could call `super().send` to do the actual sending, after we logged it. There is nothing wrong with this design either.

When faced with a choice between decorators and inheritance, we should only use decorators if we need to modify the object dynamically, according to some condition. For example, we may only want to enable the logging decorator if the server is currently in debugging mode. Decorators also beat multiple inheritance when we have more than one optional behavior. As an example, we can write a second decorator that compresses data using `gzip` compression whenever `send` is called:

```
import gzip
from io import BytesIO

class GzipSocket:
    def __init__(self, socket):
        self.socket = socket

    def send(self, data):
        buf = BytesIO()
        zipfile = gzip.GzipFile(fileobj=buf, mode="w")
        zipfile.write(data)
        zipfile.close()
        self.socket.send(buf.getvalue())

    def close(self):
        self.socket.close()
```

The `send` method in this version compresses the incoming data before sending it on to the client.

Now that we have these two decorators, we can write code that dynamically switches between them when responding. This example is not complete, but it illustrates the logic we might follow to mix and match decorators:

```
client, addr = server.accept()
if log_send:
    client = LoggingSocket(client)
if client.getpeername()[0] in compress_hosts:
    client = GzipSocket(client)
respond(client)
```

This code checks a hypothetical configuration variable named `log_send`. If it's enabled, it wraps the socket in a `LoggingSocket` decorator. Similarly, it checks whether the client that has connected is in a list of addresses known to accept compressed content. If so, it wraps the client in a `GzipSocket` decorator. Notice that none, either, or both of the decorators may be enabled, depending on the configuration and connecting client. Try writing this using multiple inheritance and see how confused you get!

Decorators in Python

The decorator pattern is useful in Python, but there are other options. For example, we may be able to use monkey-patching, which we discussed in *Chapter 7*, *Python Object-oriented Shortcuts*, to get a similar effect. Single inheritance, where the "optional" calculations are done in one large method can be an option, and multiple inheritance should not be written off just because it's not suitable for the specific example seen previously!

In Python, it is very common to use this pattern on functions. As we saw in a previous chapter, functions are objects too. In fact, function decoration is so common that Python provides a special syntax to make it easy to apply such decorators to functions.

For example, we can look at the logging example in a more general way. Instead of logging, only send calls on sockets, we may find it helpful to log all calls to certain functions or methods. The following example implements a decorator that does just this:

```
import time

def log_calls(func):
    def wrapper(*args, **kwargs):
        now = time.time()
```

```
            print("Calling {0} with {1} and {2}".format(
                func.__name__, args, kwargs))
            return_value = func(*args, **kwargs)
            print("Executed {0} in {1}ms".format(
                func.__name__, time.time() - now))
            return return_value
        return wrapper

    def test1(a,b,c):
        print("\ttest1 called")

    def test2(a,b):
        print("\ttest2 called")

    def test3(a,b):
        print("\ttest3 called")
        time.sleep(1)

    test1 = log_calls(test1)
    test2 = log_calls(test2)
    test3 = log_calls(test3)

    test1(1,2,3)
    test2(4,b=5)
    test3(6,7)
```

This decorator function is very similar to the example we explored earlier; in those cases, the decorator took a socket-like object and created a socket-like object. This time, our decorator takes a function object and returns a new function object. This code is comprised of three separate tasks:

- A function, `log_calls`, that accepts another function
- This function defines (internally) a new function, named `wrapper`, that does some extra work before calling the original function
- This new function is returned

Three sample functions demonstrate the decorator in use. The third one includes a sleep call to demonstrate the timing test. We pass each function into the decorator, which returns a new function. We assign this new function to the original variable name, effectively replacing the original function with a decorated one.

This syntax allows us to build up decorated function objects dynamically, just as we did with the socket example; if we don't replace the name, we can even keep decorated and non-decorated versions for different situations.

Often these decorators are general modifications that are applied permanently to different functions. In this situation, Python supports a special syntax to apply the decorator at the time the function is defined. We've already seen this syntax when we discussed the `property` decorator; now, let's understand how it works.

Instead of applying the decorator function after the method definition, we can use the `@decorator` syntax to do it all at once:

```
@log_calls
def test1(a,b,c):
    print("\ttest1 called")
```

The primary benefit of this syntax is that we can easily see that the function has been decorated at the time it is defined. If the decorator is applied later, someone reading the code may miss that the function has been altered at all. Answering a question like, "Why is my program logging function calls to the console?" can become much more difficult! However, the syntax can only be applied to functions we define, since we don't have access to the source code of other modules. If we need to decorate functions that are part of somebody else's third-party library, we have to use the earlier syntax.

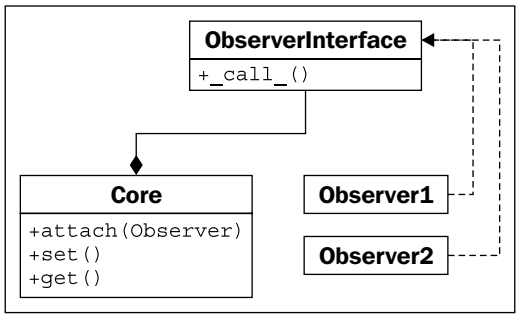
There is more to the decorator syntax than we've seen here. We don't have room to cover the advanced topics here, so check the Python reference manual or other tutorials for more information. Decorators can be created as callable objects, not just functions that return functions. Classes can also be decorated; in that case, the decorator returns a new class instead of a new function. Finally, decorators can take arguments to customize them on a per-function basis.

The observer pattern

The observer pattern is useful for state monitoring and event handling situations. This pattern allows a given object to be monitored by an unknown and dynamic group of "observer" objects.

Whenever a value on the core object changes, it lets all the observer objects know that a change has occurred, by calling an `update()` method. Each observer may be responsible for different tasks whenever the core object changes; the core object doesn't know or care what those tasks are, and the observers don't typically know or care what other observers are doing.

Here, it is in UML:



An observer example

The observer pattern might be useful in a redundant backup system. We can write a core object that maintains certain values, and then have one or more observers create serialized copies of that object. These copies might be stored in a database, on a remote host, or in a local file, for example. Let's implement the core object using properties:

```
class Inventory:
    def __init__(self):
        self.observers = []
        self._product = None
        self._quantity = 0

    def attach(self, observer):
        self.observers.append(observer)

    @property
    def product(self):
        return self._product
    @product.setter
    def product(self, value):
        self._product = value
        self._update_observers()

    @property
    def quantity(self):
        return self._quantity
    @quantity.setter
    def quantity(self, value):
        self._quantity = value
```

```
            self._update_observers()

        def _update_observers(self):
            for observer in self.observers:
                observer()
```

This object has two properties that, when set, call the `_update_observers` method on itself. All this method does is loop over the available observers and let each one know that something has changed. In this case, we call the observer object directly; the object will have to implement `__call__` to process the update. This would not be possible in many object-oriented programming languages, but it's a useful shortcut in Python that can help make our code more readable.

Now let's implement a simple observer object; this one will just print out some state to the console:

```
class ConsoleObserver:
    def __init__(self, inventory):
        self.inventory = inventory

    def __call__(self):
        print(self.inventory.product)
        print(self.inventory.quantity)
```

There's nothing terribly exciting here; the observed object is set up in the initializer, and when the observer is called, we do "something." We can test the observer in an interactive console:

```
>>> i = Inventory()
>>> c = ConsoleObserver(i)
>>> i.attach(c)
>>> i.product = "Widget"
Widget
0
>>> i.quantity = 5
Widget
5
```

After attaching the observer to the inventory object, whenever we change one of the two observed properties, the observer is called and its action is invoked. We can even add two different observer instances:

```
>>> i = Inventory()
>>> c1 = ConsoleObserver(i)
```

```
>>> c2 = ConsoleObserver(i)
>>> i.attach(c1)
>>> i.attach(c2)
>>> i.product = "Gadget"
Gadget
0
Gadget
0
```

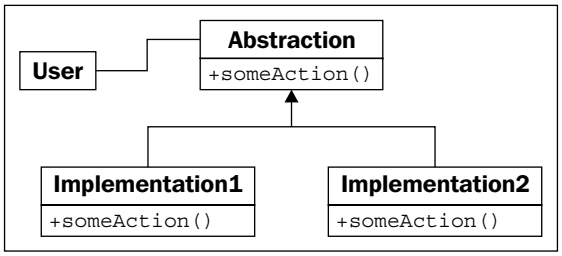
This time when we change the product, there are two sets of output, one for each observer. The key idea here is that we can easily add totally different types of observers that back up the data in a file, database, or Internet application at the same time.

The observer pattern detaches the code being observed from the code doing the observing. If we were not using this pattern, we would have had to put code in each of the properties to handle the different cases that might come up; logging to the console, updating a database or file, and so on. The code for each of these tasks would all be mixed in with the observed object. Maintaining it would be a nightmare, and adding new monitoring functionality at a later date would be painful.

The strategy pattern

The strategy pattern is a common demonstration of abstraction in object-oriented programming. The pattern implements different solutions to a single problem, each in a different object. The client code can then choose the most appropriate implementation dynamically at runtime.

Typically, different algorithms have different trade-offs; one might be faster than another, but uses a lot more memory, while a third algorithm may be most suitable when multiple CPUs are present or a distributed system is provided. Here is the strategy pattern in UML:



The **User** code connecting to the strategy pattern simply needs to know that it is dealing with the **Abstraction** interface. The actual implementation chosen performs the same task, but in different ways; either way, the interface is identical.

A strategy example

The canonical example of the strategy pattern is sort routines; over the years, numerous algorithms have been invented for sorting a collection of objects; quick sort, merge sort, and heap sort are all fast sort algorithms with different features, each useful in its own right, depending on the size and type of inputs, how out of order they are, and the requirements of the system.

If we have client code that needs to sort a collection, we could pass it to an object with a `sort()` method. This object may be a `QuickSorter` or `MergeSorter` object, but the result will be the same in either case: a sorted list. The strategy used to do the sorting is abstracted from the calling code, making it modular and replaceable.

Of course, in Python, we typically just call the `sorted` function or `list.sort` method and trust that it will do the sorting in a near-optimal fashion. So, we really need to look at a better example.

Let's consider a desktop wallpaper manager. When an image is displayed on a desktop background, it can be adjusted to the screen size in different ways. For example, assuming the image is smaller than the screen, it can be tiled across the screen, centered on it, or scaled to fit. There are other, more complicated, strategies that can be used as well, such as scaling to the maximum height or width, combining it with a solid, semi-transparent, or gradient background color, or other manipulations. While we may want to add these strategies later, let's start with the basic ones.

Our strategy objects takes two inputs; the image to be displayed, and a tuple of the width and height of the screen. They each return a new image the size of the screen, with the image manipulated to fit according to the given strategy. You'll need to install the `pillow` module with `pip3 install pillow` for this example to work:

```
from PIL import Image


class TiledStrategy:
    def make_background(self, img_file, desktop_size):
        in_img = Image.open(img_file)
        out_img = Image.new('RGB', desktop_size)
        num_tiles = [
```

```
            o // i + 1 for o, i in
            zip(out_img.size, in_img.size)
        ]
        for x in range(num_tiles[0]):
            for y in range(num_tiles[1]):
                out_img.paste(
                    in_img,
                    (
                        in_img.size[0] * x,
                        in_img.size[1] * y,
                        in_img.size[0] * (x+1),
                        in_img.size[1] * (y+1)
                    )
                )
        return out_img


class CenteredStrategy:
    def make_background(self, img_file, desktop_size):
        in_img = Image.open(img_file)
        out_img = Image.new('RGB', desktop_size)
        left = (out_img.size[0] - in_img.size[0]) // 2
        top = (out_img.size[1] - in_img.size[1]) // 2
        out_img.paste(
            in_img,
            (
                left,
                top,
                left+in_img.size[0],
                top + in_img.size[1]
            )
        )
        return out_img


class ScaledStrategy:
    def make_background(self, img_file, desktop_size):
        in_img = Image.open(img_file)
        out_img = in_img.resize(desktop_size)
        return out_img
```

Here we have three strategies, each using `PIL` to perform their task. Individual strategies have a `make_background` method that accepts the same set of parameters. Once selected, the appropriate strategy can be called to create a correctly sized version of the desktop image. `TiledStrategy` loops over the number of input images that would fit in the width and height of the image and copies it into each location, repeatedly. `CenteredStrategy` figures out how much space needs to be left on the four edges of the image to center it. `ScaledStrategy` forces the image to the output size (ignoring aspect ratio).

Consider how switching between these options would be implemented without the strategy pattern. We'd need to put all the code inside one great big method and use an awkward `if` statement to select the expected one. Every time we wanted to add a new strategy, we'd have to make the method even more ungainly.

Strategy in Python

The preceding canonical implementation of the strategy pattern, while very common in most object-oriented libraries, is rarely seen in Python programming.

These classes each represent objects that do nothing but provide a single function. We could just as easily call that function `__call__` and make the object callable directly. Since there is no other data associated with the object, we need do no more than create a set of top-level functions and pass them around as our strategies instead.

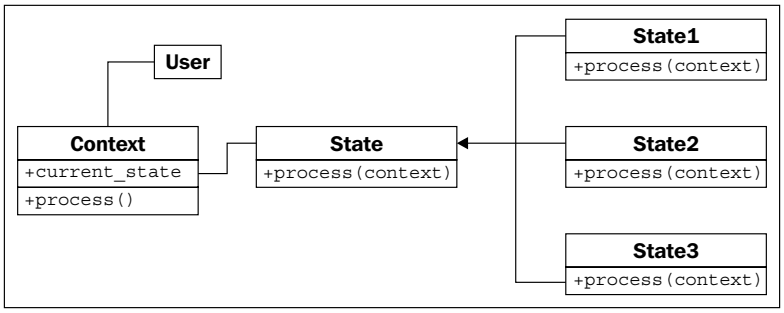
Opponents of design pattern philosophy will therefore say, "because Python has first-class functions, the strategy pattern is unnecessary". In truth, Python's first-class functions allow us to implement the strategy pattern in a more straightforward way. Knowing the pattern exists can still help us choose a correct design for our program, but implement it using a more readable syntax. The strategy pattern, or a top-level function implementation of it, should be used when we need to allow client code or the end user to select from multiple implementations of the same interface.

The state pattern

The state pattern is structurally similar to the strategy pattern, but its intent and purpose are very different. The goal of the state pattern is to represent state-transition systems: systems where it is obvious that an object can be in a specific state, and that certain activities may drive it to a different state.

To make this work, we need a manager, or context class that provides an interface for switching states. Internally, this class contains a pointer to the current state; each state knows what other states it is allowed to be in and will transition to those states depending on actions invoked upon it.

So we have two types of classes, the context class and multiple state classes. The context class maintains the current state, and forwards actions to the state classes. The state classes are typically hidden from any other objects that are calling the context; it acts like a black box that happens to perform state management internally. Here's how it looks in UML:



A state example

To illustrate the state pattern, let's build an XML parsing tool. The context class will be the parser itself. It will take a string as input and place the tool in an initial parsing state. The various parsing states will eat characters, looking for a specific value, and when that value is found, change to a different state. The goal is to create a tree of node objects for each tag and its contents. To keep things manageable, we'll parse only a subset of XML - tags and tag names. We won't be able to handle attributes on tags. It will parse text content of tags, but won't attempt to parse "mixed" content, which has tags inside of text. Here is an example "simplified XML" file that we'll be able to parse:

```
<module>
    <author>Dusty Phillips</author>
    <publisher>Packt Publishing</publisher>
    <title>Python 3 Object Oriented Programming</title>
    <content>
        <chapter>
            <number>1</number>
            <title>Object Oriented Design</title>
        </chapter>
        <chapter>
            <number>2</number>
            <title>Objects In Python</title>
        </chapter>
    </content>
</module>
```

Before we look at the states and the parser, let's consider the output of this program. We know we want a tree of `Node` objects, but what does a `Node` look like? Well, clearly it'll need to know the name of the tag it is parsing, and since it's a tree, it should probably maintain a pointer to the parent node and a list of the node's children in order. Some nodes have a text value, but not all of them. Let's look at this `Node` class first:

```
class Node:
    def __init__(self, tag_name, parent=None):
        self.parent = parent
        self.tag_name = tag_name
        self.children = []
        self.text=""

    def __str__(self):
        if self.text:
            return self.tag_name + ": " + self.text
        else:
            return self.tag_name
```

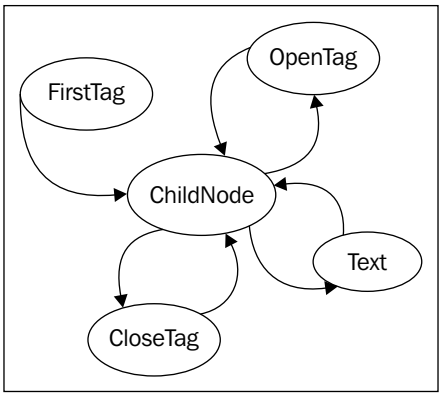
This class sets default attribute values upon initialization. The `__str__` method is supplied to help visualize the tree structure when we're finished.

Now, looking at the example document, we need to consider what states our parser can be in. Clearly it's going to start in a state where no nodes have yet been processed. We'll need a state for processing opening tags and closing tags. And when we're inside a tag with text contents, we'll have to process that as a separate state, too.

Switching states can be tricky; how do we know if the next node is an opening tag, a closing tag, or a text node? We could put a little logic in each state to work this out, but it actually makes more sense to create a new state whose sole purpose is figuring out which state we'll be switching to next. If we call this transition state **ChildNode**, we end up with the following states:

- **FirstTag**
- **ChildNode**
- **OpenTag**
- **CloseTag**
- **Text**

The **FirstTag** state will switch to **ChildNode**, which is responsible for deciding which of the other three states to switch to; when those states are finished, they'll switch back to **ChildNode**. The following state-transition diagram shows the available state changes:



The states are responsible for taking "what's left of the string", processing as much of it as they know what to do with, and then telling the parser to take care of the rest of it. Let's construct the `Parser` class first:

```
class Parser:
    def __init__(self, parse_string):
        self.parse_string = parse_string
        self.root = None
        self.current_node = None

        self.state = FirstTag()

    def process(self, remaining_string):
        remaining = self.state.process(remaining_string, self)
        if remaining:
            self.process(remaining)

    def start(self):
        self.process(self.parse_string)
```

The initializer sets up a few variables on the class that the individual states will access. The `parse_string` instance variable is the text that we are trying to parse. The `root` node is the "top" node in the XML structure. The `current_node` instance variable is the one that we are currently adding children to.

The important feature of this parser is the `process` method, which accepts the remaining string, and passes it off to the current state. The parser (the `self` argument) is also passed into the state's process method so that the state can manipulate it. The state is expected to return the remainder of the unparsed string when it is finished processing. The parser then recursively calls the `process` method on this remaining string to construct the rest of the tree.

Now, let's have a look at the `FirstTag` state:

```
class FirstTag:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        i_end_tag = remaining_string.find('>')
        tag_name = remaining_string[i_start_tag+1:i_end_tag]
        root = Node(tag_name)
        parser.root = parser.current_node = root
        parser.state = ChildNode()
        return remaining_string[i_end_tag+1:]
```

This state finds the index (the `i_` stands for index) of the opening and closing angle brackets on the first tag. You may think this state is unnecessary, since XML requires that there be no text before an opening tag. However, there may be whitespace that needs to be consumed; this is why we search for the opening angle bracket instead of assuming it is the first character in the document. Note that this code is assuming a valid input file. A proper implementation would be rigorously testing for invalid input, and would attempt to recover or display an extremely descriptive error message.

The method extracts the name of the tag and assigns it to the root node of the parser. It also assigns it to `current_node`, since that's the one we'll be adding children to next.

Then comes the important part: the method changes the current state on the parser object to a `ChildNode` state. It then returns the remainder of the string (after the opening tag) to allow it to be processed.

The `ChildNode` state, which seems quite complicated, turns out to require nothing but a simple conditional:

```
class ChildNode:
    def process(self, remaining_string, parser):
        stripped = remaining_string.strip()
        if stripped.startswith("</"):
            parser.state = CloseTag()
```

```
        elif stripped.startswith("<"):
            parser.state = OpenTag()
        else:
            parser.state = TextNode()
        return stripped
```

The `strip()` call removes whitespace from the string. Then the parser determines if the next item is an opening or closing tag, or a string of text. Depending on which possibility occurs, it sets the parser to a particular state, and then tells it to parse the remainder of the string.

The `OpenTag` state is similar to the `FirstTag` state, except that it adds the newly created node to the previous `current_node` object's `children` and sets it as the new `current_node`. It places the processor back in the `ChildNode` state before continuing:

```
class OpenTag:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        i_end_tag = remaining_string.find('>')
        tag_name = remaining_string[i_start_tag+1:i_end_tag]
        node = Node(tag_name, parser.current_node)
        parser.current_node.children.append(node)
        parser.current_node = node
        parser.state = ChildNode()
        return remaining_string[i_end_tag+1:]
```

The `CloseTag` state basically does the opposite; it sets the parser's `current_node` back to the parent node so any further children in the outside tag can be added to it:

```
class CloseTag:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        i_end_tag = remaining_string.find('>')
        assert remaining_string[i_start_tag+1] == "/"
        tag_name = remaining_string[i_start_tag+2:i_end_tag]
        assert tag_name == parser.current_node.tag_name
        parser.current_node = parser.current_node.parent
        parser.state = ChildNode()
        return remaining_string[i_end_tag+1:].strip()
```

The two `assert` statements help ensure that the parse strings are consistent. The `if` statement at the end of the method ensures that the processor terminates when it is finished. If the parent of a node is `None`, it means that we are working on the root node.

Finally, the `TextNode` state very simply extracts the text before the next close tag and sets it as a value on the current node:

```
class TextNode:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        text = remaining_string[:i_start_tag]
        parser.current_node.text = text
        parser.state = ChildNode()
        return remaining_string[i_start_tag:]
```

Now we just have to set up the initial state on the parser object we created. The initial state is a `FirstTag` object, so just add the following to the `__init__` method:

```
        self.state = FirstTag()
```

To test the class, let's add a main script that opens an file from the command line, parses it, and prints the nodes:

```
if __name__ == "__main__":
    import sys
    with open(sys.argv[1]) as file:
        contents = file.read()
        p = Parser(contents)
        p.start()

        nodes = [p.root]
        while nodes:
            node = nodes.pop(0)
            print(node)
            nodes = node.children + nodes
```

This code opens the file, loads the contents, and parses the result. Then it prints each node and its children in order. The `__str__` method we originally added on the node class takes care of formatting the nodes for printing. If we run the script on the earlier example, it outputs the tree as follows:

```
module
author: Dusty Phillips
publisher: Packt Publishing
title: Python 3 Object Oriented Programming
content
chapter
number: 1
```

```
title: Object Oriented Design
chapter
number: 2
title: Objects In Python
```

Comparing this to the original simplified XML document tells us the parser is working.

State versus strategy

The state pattern looks very similar to the strategy pattern; indeed, the UML diagrams for the two are identical. The implementation, too, is identical; we could even have written our states as first-class functions instead of wrapping them in objects, as was suggested for strategy.

While the two patterns have identical structures, they solve completely different problems. The strategy pattern is used to choose an algorithm at runtime; generally, only one of those algorithms is going to be chosen for a particular use case. The state pattern, on the other hand is designed to allow switching between different states dynamically, as some process evolves. In code, the primary difference is that the strategy pattern is not typically aware of other strategy objects. In the state pattern, either the state or the context needs to know which other states that it can switch to.

State transition as coroutines

The state pattern is the canonical object-oriented solution to state-transition problems. However, the syntax for this pattern is rather verbose. You can get a similar effect by constructing your objects as coroutines. Remember the regular expression log file parser we built in *Chapter 9*, *The Iterator Pattern*? That was a state-transition problem in disguise. The main difference between that implementation and one that defines all the objects (or functions) used in the state pattern is that the coroutine solution allows us to encode more of the boilerplate in language constructs. There are two implementations, but neither one is inherently better than the other, but you may find that coroutines are more readable, for a given definition of "readable" (you have to understand the syntax of coroutines, first!).

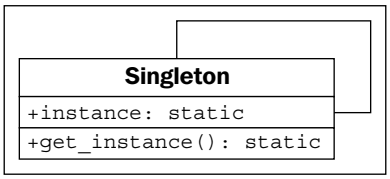
The singleton pattern

The singleton pattern is one of the most controversial patterns; many have accused it of being an "anti-pattern", a pattern that should be avoided, not promoted. In Python, if someone is using the singleton pattern, they're almost certainly doing something wrong, probably because they're coming from a more restrictive programming language.

So why discuss it at all? Singleton is one of the most famous of all design patterns. It is useful in overly object-oriented languages, and is a vital part of traditional object-oriented programming. More relevantly, the idea behind singleton is useful, even if we implement that idea in a totally different way in Python.

The basic idea behind the singleton pattern is to allow exactly one instance of a certain object to exist. Typically, this object is a sort of manager class like those we discussed in *Chapter 5*, *When to Use Object-oriented Programming*. Such objects often need to be referenced by a wide variety of other objects, and passing references to the manager object around to the methods and constructors that need them can make code hard to read.

Instead, when a singleton is used, the separate objects request the single instance of the manager object from the class, so a reference to it need not to be passed around. The UML diagram doesn't fully describe it, but here it is for completeness:



In most programming environments, singletons are enforced by making the constructor private (so no one can create additional instances of it), and then providing a static method to retrieve the single instance. This method creates a new instance the first time it is called, and then returns that same instance each time it is called again.

Singleton implementation

Python doesn't have private constructors, but for this purpose, it has something even better. We can use the `__new__` class method to ensure that only one instance is ever created:

```
class OneOnly:
    _singleton = None
    def __new__(cls, *args, **kwargs):
        if not cls._singleton:
            cls._singleton = super(OneOnly, cls
                ).__new__(cls, *args, **kwargs)
        return cls._singleton
```

When `__new__` is called, it normally constructs a new instance of that class. When we override it, we first check if our singleton instance has been created; if not, we create it using a `super` call. Thus, whenever we call the constructor on `OneOnly`, we always get the exact same instance:

```
>>> o1 = OneOnly()
>>> o2 = OneOnly()
>>> o1 == o2
True
>>> o1
<__main__.OneOnly object at 0xb71c008c>
>>> o2
<__main__.OneOnly object at 0xb71c008c>
```

The two objects are equal and located at the same address; thus, they are the same object. This particular implementation isn't very transparent, since it's not obvious that a singleton object has been created. Whenever we call a constructor, we expect a new instance of that object; in this case, that contract is violated. Perhaps, good docstrings on the class could alleviate this problem if we really think we need a singleton.

But we don't need it. Python coders frown on forcing the users of their code into a specific mindset. We may think only one instance of a class will ever be required, but other programmers may have different ideas. Singletons can interfere with distributed computing, parallel programming, and automated testing, for example. In all those cases, it can be very useful to have multiple or alternative instances of a specific object, even though a "normal' operation may never require one.

Module variables can mimic singletons

Normally, in Python, the singleton pattern can be sufficiently mimicked using module-level variables. It's not as "safe" as a singleton in that people could reassign those variables at any time, but as with the private variables we discussed in *Chapter 2*, *Objects in Python*, this is acceptable in Python. If someone has a valid reason to change those variables, why should we stop them? It also doesn't stop people from instantiating multiple instances of the object, but again, if they have a valid reason to do so, why interfere?

Ideally, we should give them a mechanism to get access to the "default singleton" value, while also allowing them to create other instances if they need them. While technically not a singleton at all, it provides the most Pythonic mechanism for singleton-like behavior.

To use module-level variables instead of a singleton, we instantiate an instance of the class after we've defined it. We can improve our state pattern to use singletons. Instead of creating a new object every time we change states, we can create a module-level variable that is always accessible:

```
class FirstTag:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        i_end_tag = remaining_string.find('>')
        tag_name = remaining_string[i_start_tag+1:i_end_tag]
        root = Node(tag_name)
        parser.root = parser.current_node = root
        parser.state = child_node
        return remaining_string[i_end_tag+1:]

class ChildNode:
    def process(self, remaining_string, parser):
        stripped = remaining_string.strip()
        if stripped.startswith("</"):
            parser.state = close_tag
        elif stripped.startswith("<"):
            parser.state = open_tag
        else:
            parser.state = text_node
        return stripped

class OpenTag:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        i_end_tag = remaining_string.find('>')
        tag_name = remaining_string[i_start_tag+1:i_end_tag]
        node = Node(tag_name, parser.current_node)
        parser.current_node.children.append(node)
        parser.current_node = node
        parser.state = child_node
        return remaining_string[i_end_tag+1:]
class TextNode:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        text = remaining_string[:i_start_tag]
        parser.current_node.text = text
        parser.state = child_node
```

```
            return remaining_string[i_start_tag:]

    class CloseTag:
        def process(self, remaining_string, parser):
            i_start_tag = remaining_string.find('<')
            i_end_tag = remaining_string.find('>')
            assert remaining_string[i_start_tag+1] == "/"
            tag_name = remaining_string[i_start_tag+2:i_end_tag]
            assert tag_name == parser.current_node.tag_name
            parser.current_node = parser.current_node.parent
            parser.state = child_node
            return remaining_string[i_end_tag+1:].strip()

    first_tag = FirstTag()
    child_node = ChildNode()
    text_node = TextNode()
    open_tag = OpenTag()
close_tag = CloseTag()
```

All we've done is create instances of the various state classes that can be reused. Notice how we can access these module variables inside the classes, even before the variables have been defined? This is because the code inside the classes is not executed until the method is called, and by this point, the entire module will have been defined.

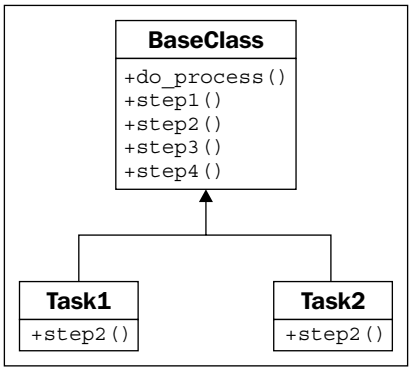
The difference in this example is that instead of wasting memory creating a bunch of new instances that must be garbage collected, we are reusing a single state object for each state. Even if multiple parsers are running at once, only these state classes need to be used.

When we originally created the state-based parser, you may have wondered why we didn't pass the parser object to `__init__` on each individual state, instead of passing it into the process method as we did. The state could then have been referenced as `self.parser`. This is a perfectly valid implementation of the state pattern, but it would not have allowed leveraging the singleton pattern. If the state objects maintain a reference to the parser, then they cannot be used simultaneously to reference other parsers.

Remember, these are two different patterns with different purposes; the fact that singleton's purpose may be useful for implementing the state pattern does not mean the two patterns are related.

The template pattern

The template pattern is useful for removing duplicate code; it's an implementation to support the **Don't Repeat Yourself** principle we discussed in *Chapter 5*, *When to Use Object-oriented Programming*. It is designed for situations where we have several different tasks to accomplish that have some, but not all, steps in common. The common steps are implemented in a base class, and the distinct steps are overridden in subclasses to provide custom behavior. In some ways, it's like a generalized strategy pattern, except similar sections of the algorithms are shared using a base class. Here it is in the UML format:



A template example

Let's create a car sales reporter as an example. We can store records of sales in an SQLite database table. SQLite is a simple file-based database engine that allows us to store records using SQL syntax. Python 3 includes SQLite in its standard library, so there are no extra modules required.

We have two common tasks we need to perform:

- Select all sales of new vehicles and output them to the screen in a comma-delimited format
- Output a comma-delimited list of all salespeople with their gross sales and save it to a file that can be imported to a spreadsheet

These seem like quite different tasks, but they have some common features. In both cases, we need to perform the following steps:

1. Connect to the database.
2. Construct a query for new vehicles or gross sales.

3. Issue the query.
4. Format the results into a comma-delimited string.
5. Output the data to a file or e-mail.

The query construction and output steps are different for the two tasks, but the remaining steps are identical. We can use the template pattern to put the common steps in a base class, and the varying steps in two subclasses.

Before we start, let's create a database and put some sample data in it, using a few lines of SQL:

```
import sqlite3

conn = sqlite3.connect("sales.db")

conn.execute("CREATE TABLE Sales (salesperson text, "
        "amt currency, year integer, model text, new boolean)")
conn.execute("INSERT INTO Sales values"
        " ('Tim', 16000, 2010, 'Honda Fit', 'true')")
conn.execute("INSERT INTO Sales values"
        " ('Tim', 9000, 2006, 'Ford Focus', 'false')")
conn.execute("INSERT INTO Sales values"
        " ('Gayle', 8000, 2004, 'Dodge Neon', 'false')")
conn.execute("INSERT INTO Sales values"
        " ('Gayle', 28000, 2009, 'Ford Mustang', 'true')")
conn.execute("INSERT INTO Sales values"
        " ('Gayle', 50000, 2010, 'Lincoln Navigator', 'true')")
conn.execute("INSERT INTO Sales values"
        " ('Don', 20000, 2008, 'Toyota Prius', 'false')")
conn.commit()
conn.close()
```

Hopefully you can see what's going on here even if you don't know SQL; we've created a table to hold the data, and used six insert statements to add sales records. The data is stored in a file named `sales.db`. Now we have a sample we can work with in developing our template pattern.

Since we've already outlined the steps that the template has to perform, we can start by defining the base class that contains the steps. Each step gets its own method (to make it easy to selectively override any one step), and we have one more managerial method that calls the steps in turn. Without any method content, here's how it might look:

```
class QueryTemplate:
    def connect(self):
        pass
```

```
    def construct_query(self):
        pass
    def do_query(self):
        pass
    def format_results(self):
        pass
    def output_results(self):
        pass

    def process_format(self):
        self.connect()
        self.construct_query()
        self.do_query()
        self.format_results()
        self.output_results()
```

The `process_format` method is the primary method to be called by an outside client. It ensures each step is executed in order, but it does not care if that step is implemented in this class or in a subclass. For our examples, we know that three methods are going to be identical between our two classes:

```
import sqlite3

class QueryTemplate:
    def connect(self):
        self.conn = sqlite3.connect("sales.db")

    def construct_query(self):
        raise NotImplementedError()

    def do_query(self):
        results = self.conn.execute(self.query)
        self.results = results.fetchall()

    def format_results(self):
        output = []
        for row in self.results:
            row =[str(i) for i in row]
            output.append(", ".join(row))
        self.formatted_results = "\n".join(output)

    def output_results(self):
        raise NotImplementedError()
```

To help with implementing subclasses, the two methods that are not specified raise `NotImplementedError`. This is a common way to specify abstract interfaces in Python when abstract base classes seem too heavyweight. The methods could have empty implementations (with `pass`), or could be fully unspecified. Raising `NotImplementedError`, however, helps the programmer understand that the class is meant to be subclassed and these methods overridden; empty methods or methods that do not exist are harder to identify as needing to be implemented and to debug if we forget to implement them.

Now we have a template class that takes care of the boring details, but is flexible enough to allow the execution and formatting of a wide variety of queries. The best part is, if we ever want to change our database engine from SQLite to another database engine (such as py-postgresql), we only have to do it here, in this template class, and we don't have to touch the two (or two hundred) subclasses we might have written.

Let's have a look at the concrete classes now:

```
import datetime
class NewVehiclesQuery(QueryTemplate):
    def construct_query(self):
        self.query = "select * from Sales where new='true'"

    def output_results(self):
        print(self.formatted_results)

class UserGrossQuery(QueryTemplate):
    def construct_query(self):
        self.query = ("select salesperson, sum(amt) " +
        " from Sales group by salesperson")

    def output_results(self):
        filename = "gross_sales_{0}".format(
                datetime.date.today().strftime("%Y%m%d")
                )
        with open(filename, 'w') as outfile:
            outfile.write(self.formatted_results)
```

These two classes are actually pretty short, considering what they're doing: connecting to a database, executing a query, formatting the results, and outputting them. The superclass takes care of the repetitive work, but lets us easily specify those steps that vary between tasks. Further, we can also easily change steps that are provided in the base class. For example, if we wanted to output something other than a comma-delimited string (for example: an HTML report to be uploaded to a website), we can still override `format_results`.

Your Coding Challenge

While writing this chapter, I discovered that it can be very difficult, and extremely educational, to come up with good examples where specific design patterns should be used. Instead of going over current or old projects to see where you can apply these patterns, as I've suggested in previous chapters, think about the patterns and different situations where they might come up. Try to think outside your own experiences. If your current projects are in the banking business, consider how you'd apply these design patterns in a retail or point-of-sale application. If you normally write web applications, think about using design patterns while writing a compiler.

Ankita Thakur

Your Course Guide

Look at the decorator pattern and come up with some good examples of when to apply it. Focus on the pattern itself, not the Python syntax we discussed; it's a bit more general than the actual pattern. The special syntax for decorators is, however, something you may want to look for places to apply in existing projects too.

What are some good areas to use the observer pattern? Why? Think about not only how you'd apply the pattern, but how you would implement the same task without using observer? What do you gain, or lose, by choosing to use it?

Consider the difference between the strategy and state patterns. Implementation-wise, they look very similar, yet they have different purposes. Can you think of cases where the patterns could be interchanged? Would it be reasonable to redesign a state-based system to use strategy instead, or vice versa? How different would the design actually be?

The template pattern is such an obvious application of inheritance to reduce duplicate code that you may have used it before, without knowing its name. Try to think of at least half a dozen different scenarios where it would be useful. If you can do this, you'll be finding places for it in your daily coding all the time.

Summary of Module 1 Chapter 11

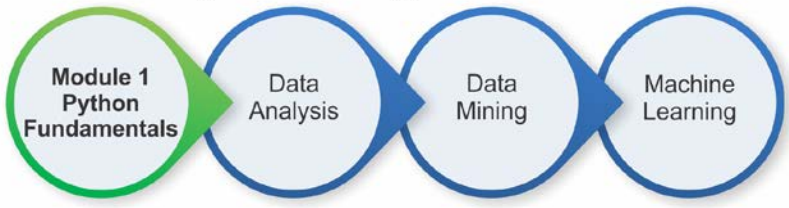
Ankita Thakur

Your Course Guide

This chapter discussed several common design patterns in detail, with examples, UML diagrams, and a discussion of the differences between Python and statically typed object-oriented languages. The decorator pattern is often implemented using Python's more generic decorator syntax. The observer pattern is a useful way to decouple events from actions taken on those events. The strategy pattern allows different algorithms to be chosen to accomplish the same task. The state pattern looks similar, but is used instead to represent systems can move between different states using well-defined actions. The singleton pattern, popular in some statically typed languages, is almost always an anti-pattern in Python.

In the next chapter, we'll wrap up our discussion of design patterns.

Your Progress through the Course So Far



12
Python Design Patterns II

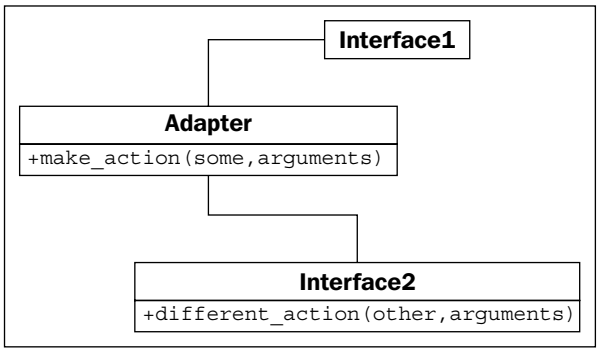
In this chapter we will be introduced to several more design patterns. Once again, we'll cover the canonical examples as well as any common alternative implementations in Python. We'll be discussing:

- The adapter pattern
- The facade pattern
- Lazy initialization and the flyweight pattern
- The command pattern
- The abstract factory pattern
- The composition pattern

The adapter pattern

Unlike most of the patterns we reviewed in *Chapter 8*, *Strings and Serialization*, the adapter pattern is designed to interact with existing code. We would not design a brand new set of objects that implement the adapter pattern. Adapters are used to allow two pre-existing objects to work together, even if their interfaces are not compatible. Like the display adapters that allow VGA projectors to be plugged into HDMI ports, an adapter object sits between two different interfaces, translating between them on the fly. The adapter object's sole purpose is to perform this translation job. Adapting may entail a variety of tasks, such as converting arguments to a different format, rearranging the order of arguments, calling a differently named method, or supplying default arguments.

In structure, the adapter pattern is similar to a simplified decorator pattern. Decorators typically provide the same interface that they replace, whereas adapters map between two different interfaces. Here it is in UML form:



Here, **Interface1** is expecting to call a method called **make_action(some, arguments)**. We already have this perfect **Interface2** class that does everything we want (and to avoid duplication, we don't want to rewrite it!), but it provides a method called **different_action(other, arguments)** instead. The **Adapter** class implements the **make_action** interface and maps the arguments to the existing interface.

The advantage here is that the code that maps from one interface to another is all in one place. The alternative would be really ugly; we'd have to perform the translation in multiple places whenever we need to access this code.

For example, imagine we have the following preexisting class, which takes a string date in the format "YYYY-MM-DD" and calculates a person's age on that day:

```
class AgeCalculator:
    def __init__(self, birthday):
        self.year, self.month, self.day = (
                int(x) for x in birthday.split('-'))

    def calculate_age(self, date):
        year, month, day = (
                int(x) for x in date.split('-'))
        age = year - self.year
        if (month,day) < (self.month,self.day):
            age -= 1
        return age
```

This is a pretty simple class that does what it's supposed to do. But we have to wonder what the programmer was thinking, using a specifically formatted string instead of using Python's incredibly useful built-in `datetime` library. As conscientious programmers who reuse code whenever possible, most of the programs we write will interact with `datetime` objects, not strings.

We have several options to address this scenario; we could rewrite the class to accept `datetime` objects, which would probably be more accurate anyway. But if this class had been provided by a third party and we don't know or can't change its internal structure, we need to try something else. We could use the class as it is, and whenever we want to calculate the age on a `datetime.date` object, we could call `datetime.date.strftime('%Y-%m-%d')` to convert it to the proper format. But that conversion would be happening in a lot of places, and worse, if we mistyped the `%m` as `%M`, it would give us the current minute instead of the entered month! Imagine if you wrote that in a dozen different places only to have to go back and change it when you realized your mistake. It's not maintainable code, and it breaks the DRY principle.

Instead, we can write an adapter that allows a normal date to be plugged into a normal `AgeCalculator` class:

```
import datetime
class DateAgeAdapter:
    def _str_date(self, date):
        return date.strftime("%Y-%m-%d")

    def __init__(self, birthday):
        birthday = self._str_date(birthday)
        self.calculator = AgeCalculator(birthday)

    def get_age(self, date):
        date = self._str_date(date)
        return self.calculator.calculate_age(date)
```

This adapter converts `datetime.date` and `datetime.time` (they have the same interface to `strftime`) into a string that our original `AgeCalculator` can use. Now we can use the original code with our new interface. I changed the method signature to `get_age` to demonstrate that the calling interface may also be looking for a different method name, not just a different type of argument.

Creating a class as an adapter is the usual way to implement this pattern, but, as usual, there are other ways to do it in Python. Inheritance and multiple inheritance can be used to add functionality to a class. For example, we could add an adapter on the `date` class so that it works with the original `AgeCalculator` class:

```
import datetime
class AgeableDate(datetime.date):
    def split(self, char):
        return self.year, self.month, self.day
```

It's code like this that makes one wonder if Python should even be legal. We have added a `split` method to our subclass that takes a single argument (which we ignore) and returns a tuple of year, month, and day. This works flawlessly with the original `AgeCalculator` class because the code calls `strip` on a specially formatted string, and `strip`, in that case, returns a tuple of year, month, and day. The `AgeCalculator` code only cares if `strip` exists and returns acceptable values; it doesn't care if we really passed in a string. It really works:

```
>>> bd = AgeableDate(1975, 6, 14)
>>> today = AgeableDate.today()
>>> today
AgeableDate(2015, 8, 4)
>>> a = AgeCalculator(bd)
>>> a.calculate_age(today)
40
```

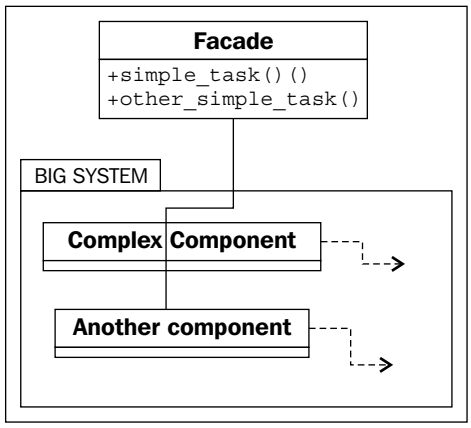
It works but it's a stupid idea. In this particular instance, such an adapter would be hard to maintain. We'd soon forget why we needed to add a `strip` method to a `date` class. The method name is ambiguous. That can be the nature of adapters, but creating an adapter explicitly instead of using inheritance usually clarifies its purpose.

Instead of inheritance, we can sometimes also use monkey-patching to add a method to an existing class. It won't work with the `datetime` object, as it doesn't allow attributes to be added at runtime, but in normal classes, we can just add a new method that provides the adapted interface that is required by calling code. Alternatively, we could extend or monkey-patch the `AgeCalculator` itself to replace the `calculate_age` method with something more amenable to our needs.

Finally, it is often possible to use a function as an adapter; this doesn't obviously fit the actual design of the adapter pattern, but if we recall that functions are essentially objects with a `__call__` method, it becomes an obvious adapter adaptation.

The facade pattern

The facade pattern is designed to provide a simple interface to a complex system of components. For complex tasks, we may need to interact with these objects directly, but there is often a "typical" usage for the system for which these complicated interactions aren't necessary. The facade pattern allows us to define a new object that encapsulates this typical usage of the system. Any time we want access to common functionality, we can use the single object's simplified interface. If another part of the project needs access to more complicated functionality, it is still able to interact with the system directly. The UML diagram for the facade pattern is really dependent on the subsystem, but in a cloudy way, it looks like this:



A facade is, in many ways, like an adapter. The primary difference is that the facade is trying to abstract a simpler interface out of a complex one, while the adapter is only trying to map one existing interface to another.

Let's write a simple facade for an e-mail application. The low-level library for sending e-mail in Python, as we saw in *Chapter 7*, *Python Object-oriented Shortcuts*, is quite complicated. The two libraries for receiving messages are even worse.

It would be nice to have a simple class that allows us to send a single e-mail, and list the e-mails currently in the inbox on an IMAP or POP3 connection. To keep our example short, we'll stick with IMAP and SMTP: two totally different subsystems that happen to deal with e-mail. Our facade performs only two tasks: sending an e-mail to a specific address, and checking the inbox on an IMAP connection. It makes some common assumptions about the connection, such as the host for both SMTP and IMAP is at the same address, that the username and password for both is the same, and that they use standard ports. This covers the case for many e-mail servers, but if a programmer needs more flexibility, they can always bypass the facade and access the two subsystems directly.

The class is initialized with the hostname of the e-mail server, a username, and a password to log in:

```
import smtplib
import imaplib

class EmailFacade:
    def __init__(self, host, username, password):
        self.host = host
        self.username = username
        self.password = password
```

The `send_email` method formats the e-mail address and message, and sends it using `smtplib`. This isn't a complicated task, but it requires quite a bit of fiddling to massage the "natural" input parameters that are passed into the facade to the correct format to enable `smtplib` to send the message:

```
    def send_email(self, to_email, subject, message):
        if not "@" in self.username:
            from_email = "{0}@{1}".format(
                    self.username, self.host)
        else:
            from_email = self.username
        message = ("From: {0}\r\n"
                "To: {1}\r\n"
                "Subject: {2}\r\n\r\n{3}").format(
                    from_email,
                    to_email,
                    subject,
                    message)

        smtp = smtplib.SMTP(self.host)
        smtp.login(self.username, self.password)
        smtp.sendmail(from_email, [to_email], message)
```

The `if` statement at the beginning of the method is catching whether or not the `username` is the entire "from" e-mail address or just the part on the left side of the `@` symbol; different hosts treat the login details differently.

Finally, the code to get the messages currently in the inbox is a ruddy mess; the IMAP protocol is painfully over-engineered, and the `imaplib` standard library is only a thin layer over the protocol:

```
    def get_inbox(self):
        mailbox = imaplib.IMAP4(self.host)
```

```
        mailbox.login(bytes(self.username, 'utf8'),
            bytes(self.password, 'utf8'))
        mailbox.select()
        x, data = mailbox.search(None, 'ALL')
        messages = []
        for num in data[0].split():
            x, message = mailbox.fetch(num, '(RFC822)')
            messages.append(message[0][1])
        return messages
```

Now, if we add all this together, we have a simple facade class that can send and receive messages in a fairly straightforward manner, much simpler than if we had to interact with these complex libraries directly.

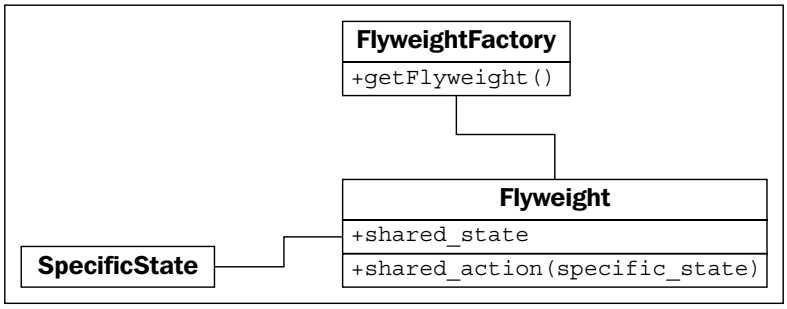
Although it is rarely named in the Python community, the facade pattern is an integral part of the Python ecosystem. Because Python emphasizes language readability, both the language and its libraries tend to provide easy-to-comprehend interfaces to complicated tasks. For example, `for` loops, `list` comprehensions, and generators are all facades into a more complicated iterator protocol. The `defaultdict` implementation is a facade that abstracts away annoying corner cases when a key doesn't exist in a dictionary. The third-party requests library is a powerful facade over less readable libraries for HTTP requests.

The flyweight pattern

The flyweight pattern is a memory optimization pattern. Novice Python programmers tend to ignore memory optimization, assuming the built-in garbage collector will take care of them. This is often perfectly acceptable, but when developing larger applications with many related objects, paying attention to memory concerns can have a huge payoff.

The flyweight pattern basically ensures that objects that share a state can use the same memory for that shared state. It is often implemented only after a program has demonstrated memory problems. It may make sense to design an optimal configuration from the beginning in some situations, but bear in mind that premature optimization is the most effective way to create a program that is too complicated to maintain.

Let's have a look at the UML diagram for the flyweight pattern:



Each **Flyweight** has no specific state; any time it needs to perform an operation on **SpecificState**, that state needs to be passed into the **Flyweight** by the calling code. Traditionally, the factory that returns a flyweight is a separate object; its purpose is to return a flyweight for a given key identifying that flyweight. It works like the singleton pattern we discussed in *Chapter 10*, *Python Design Patterns I*; if the flyweight exists, we return it; otherwise, we create a new one. In many languages, the factory is implemented, not as a separate object, but as a static method on the `Flyweight` class itself.

Think of an inventory system for car sales. Each individual car has a specific serial number and is a specific color. But most of the details about that car are the same for all cars of a particular model. For example, the Honda Fit DX model is a bare-bones car with few features. The LX model has A/C, tilt, cruise, and power windows and locks. The Sport model has fancy wheels, a USB charger, and a spoiler. Without the flyweight pattern, each individual car object would have to store a long list of which features it did and did not have. Considering the number of cars Honda sells in a year, this would add up to a huge amount of wasted memory. Using the flyweight pattern, we can instead have shared objects for the list of features associated with a model, and then simply reference that model, along with a serial number and color, for individual vehicles. In Python, the flyweight factory is often implemented using that funky `__new__` constructor, similar to what we did with the singleton pattern. Unlike singleton, which only needs to return one instance of the class, we need to be able to return different instances depending on the keys. We could store the items in a dictionary and look them up based on the key. This solution is problematic, however, because the item will remain in memory as long as it is in the dictionary. If we sold out of LX model Fits, the Fit flyweight is no longer necessary, yet it will still be in the dictionary. We could, of course, clean this up whenever we sell a car, but isn't that what a garbage collector is for?

We can solve this by taking advantage of Python's `weakref` module. This module provides a `WeakValueDictionary` object, which basically allows us to store items in a dictionary without the garbage collector caring about them. If a value is in a weak referenced dictionary and there are no other references to that object stored anywhere in the application (that is, we sold out of LX models), the garbage collector will eventually clean up for us.

Let's build the factory for our car flyweights first:

```
import weakref

class CarModel:
    _models = weakref.WeakValueDictionary()

    def __new__(cls, model_name, *args, **kwargs):
        model = cls._models.get(model_name)
        if not model:
            model = super().__new__(cls)
            cls._models[model_name] = model

        return model
```

Basically, whenever we construct a new flyweight with a given name, we first look up that name in the weak referenced dictionary; if it exists, we return that model; if not, we create a new one. Either way, we know the `__init__` method on the flyweight will be called every time, regardless of whether it is a new or existing object. Our `__init__` method can therefore look like this:

```
    def __init__(self, model_name, air=False, tilt=False,
            cruise_control=False, power_locks=False,
            alloy_wheels=False, usb_charger=False):
        if not hasattr(self, "initted"):
            self.model_name = model_name
            self.air = air
            self.tilt = tilt
            self.cruise_control = cruise_control
            self.power_locks = power_locks
            self.alloy_wheels = alloy_wheels
            self.usb_charger = usb_charger
            self.initted=True
```

The `if` statement ensures that we only initialize the object the first time `__init__` is called. This means we can call the factory later with just the model name and get the same flyweight object back. However, because the flyweight will be garbage-collected if no external references to it exist, we have to be careful not to accidentally create a new flyweight with null values.

Let's add a method to our flyweight that hypothetically looks up a serial number on a specific model of vehicle, and determines if it has been involved in any accidents. This method needs access to the car's serial number, which varies from car to car; it cannot be stored with the flyweight. Therefore, this data must be passed into the method by the calling code:

```
        def check_serial(self, serial_number):
            print("Sorry, we are unable to check "
                    "the serial number {0} on the {1} "
                    "at this time".format(
                        serial_number, self.model_name))
```

We can define a class that stores the additional information, as well as a reference to the flyweight:

```
    class Car:
        def __init__(self, model, color, serial):
            self.model = model
            self.color = color
            self.serial = serial

        def check_serial(self):
            return self.model.check_serial(self.serial)
```

We can also keep track of the available models as well as the individual cars on the lot:

```
>>> dx = CarModel("FIT DX")
>>> lx = CarModel("FIT LX", air=True, cruise_control=True,
... power_locks=True, tilt=True)
>>> car1 = Car(dx, "blue", "12345")
>>> car2 = Car(dx, "black", "12346")
>>> car3 = Car(lx, "red", "12347")
```

Now, let's demonstrate the weak referencing at work:

```
>>> id(lx)
3071620300
>>> del lx
>>> del car3
>>> import gc
>>> gc.collect()
0
```

```
>>> lx = CarModel("FIT LX", air=True, cruise_control=True,
... power_locks=True, tilt=True)
>>> id(lx)
3071576140
>>> lx = CarModel("FIT LX")
>>> id(lx)
3071576140
>>> lx.air
True
```

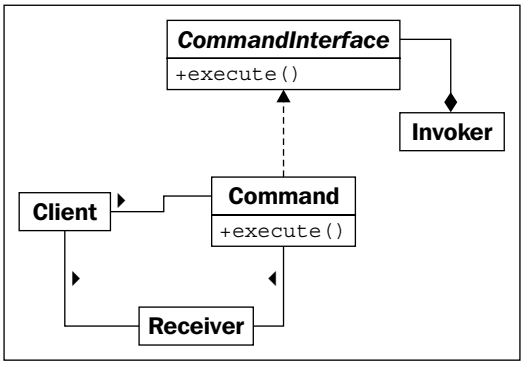
The `id` function tells us the unique identifier for an object. When we call it a second time, after deleting all references to the LX model and forcing garbage collection, we see that the ID has changed. The value in the `CarModel __new__` factory dictionary was deleted and a fresh one created. If we then try to construct a second `CarModel` instance, however, it returns the same object (the IDs are the same), and, even though we did not supply any arguments in the second call, the `air` variable is still set to `True`. This means the object was not initialized the second time, just as we designed.

Obviously, using the flyweight pattern can be more complicated than just storing features on a single car class. When should we choose to use it? The flyweight pattern is designed for conserving memory; if we have hundreds of thousands of similar objects, combining similar properties into a flyweight can have an enormous impact on memory consumption. It is common for programming solutions that optimize CPU, memory, or disk space result in more complicated code than their unoptimized brethren. It is therefore important to weigh up the tradeoffs when deciding between code maintainability and optimization. When choosing optimization, try to use patterns such as flyweight to ensure that the complexity introduced by optimization is confined to a single (well documented) section of the code.

The command pattern

The command pattern adds a level of abstraction between actions that must be done, and the object that invokes those actions, normally at a later time. In the command pattern, client code creates a `Command` object that can be executed at a later date. This object knows about a receiver object that manages its own internal state when the command is executed on it. The `Command` object implements a specific interface (typically it has an `execute` or `do_action` method, and also keeps track of any arguments required to perform the action. Finally, one or more `Invoker` objects execute the command at the correct time.

Here's the UML diagram:



A common example of the command pattern is actions on a graphical window. Often, an action can be invoked by a menu item on the menu bar, a keyboard shortcut, a toolbar icon, or a context menu. These are all examples of `Invoker` objects. The actions that actually occur, such as `Exit`, `Save`, or `Copy`, are implementations of `CommandInterface`. A GUI window to receive exit, a document to receive save, and `ClipboardManager` to receive copy commands, are all examples of possible `Receivers`.

Let's implement a simple command pattern that provides commands for `Save` and `Exit` actions. We'll start with some modest receiver classes:

```
import sys

class Window:
    def exit(self):
        sys.exit(0)

class Document:
    def __init__(self, filename):
        self.filename = filename
        self.contents = "This file cannot be modified"

    def save(self):
        with open(self.filename, 'w') as file:
            file.write(self.contents)
```

These mock classes model objects that would likely be doing a lot more in a working environment. The window would need to handle mouse movement and keyboard events, and the document would need to handle character insertion, deletion, and selection. But for our example these two classes will do what we need.

Now let's define some invoker classes. These will model toolbar, menu, and keyboard events that can happen; again, they aren't actually hooked up to anything, but we can see how they are decoupled from the command, receiver, and client code:

```
class ToolbarButton:
    def __init__(self, name, iconname):
        self.name = name
        self.iconname = iconname

    def click(self):
        self.command.execute()

class MenuItem:
    def __init__(self, menu_name, menuitem_name):
        self.menu = menu_name
        self.item = menuitem_name

    def click(self):
        self.command.execute()

class KeyboardShortcut:
    def __init__(self, key, modifier):
        self.key = key
        self.modifier = modifier

    def keypress(self):
        self.command.execute()
```

Notice how the various action methods each call the `execute` method on their respective commands? This code doesn't show the `command` attribute being set on each object. They could be passed into the `__init__` function, but because they may be changed (for example, with a customizable keybinding editor), it makes more sense to set the attributes on the objects afterwards.

Now, let's hook up the commands themselves:

```
class SaveCommand:
    def __init__(self, document):
        self.document = document

    def execute(self):
        self.document.save()

class ExitCommand:
```

```
    def __init__(self, window):
        self.window = window

    def execute(self):
        self.window.exit()
```

These commands are straightforward; they demonstrate the basic pattern, but it is important to note that we can store state and other information with the command if necessary. For example, if we had a command to insert a character, we could maintain state for the character currently being inserted.

Now all we have to do is hook up some client and test code to make the commands work. For basic testing, we can just include this at the end of the script:

```
window = Window()
document = Document("a_document.txt")
save = SaveCommand(document)
exit = ExitCommand(window)

save_button = ToolbarButton('save', 'save.png')
save_button.command = save
save_keystroke = KeyboardShortcut("s", "ctrl")
save_keystroke.command = save
exit_menu = MenuItem("File", "Exit")
exit_menu.command = exit
```

First we create two receivers and two commands. Then we create several of the available invokers and set the correct command on each of them. To test, we can use `python3 -i filename.py` and run code like `exit_menu.click()`, which will end the program, or `save_keystroke.keystroke()`, which will save the fake file.

Unfortunately, the preceding examples do not feel terribly Pythonic. They have a lot of "boilerplate code" (code that does not accomplish anything, but only provides structure to the pattern), and the `Command` classes are all eerily similar to each other. Perhaps we could create a generic command object that takes a function as a callback?

In fact, why bother? Can we just use a function or method object for each command? Instead of an object with an `execute()` method, we can write a function and use that as the command directly. This is a common paradigm for the command pattern in Python:

```
import sys

class Window:
```

```
        def exit(self):
            sys.exit(0)

    class MenuItem:
        def click(self):
            self.command()

    window = Window()
    menu_item = MenuItem()
    menu_item.command = window.exit
```

Now that looks a lot more like Python. At first glance, it looks like we've removed the command pattern altogether, and we've tightly connected the `menu_item` and `Window` classes. But if we look closer, we find there is no tight coupling at all. Any callable can be set up as the command on the `MenuItem`, just as before. And the `Window.exit` method can be attached to any invoker. Most of the flexibility of the command pattern has been maintained. We have sacrificed complete decoupling for readability, but this code is, in my opinion, and that of many Python programmers, more maintainable than the fully abstracted version.

Of course, since we can add a `__call__` method to any object, we aren't restricted to functions. The previous example is a useful shortcut when the method being called doesn't have to maintain state, but in more advanced usage, we can use this code as well:

```
    class Document:
        def __init__(self, filename):
            self.filename = filename
            self.contents = "This file cannot be modified"

        def save(self):
            with open(self.filename, 'w') as file:
                file.write(self.contents)

    class KeyboardShortcut:
        def keypress(self):
            self.command()
    class SaveCommand:
        def __init__(self, document):
            self.document = document

        def __call__(self):
```

```
            self.document.save()

document = Document("a_file.txt")
shortcut = KeyboardShortcut()
save_command = SaveCommand(document)
shortcut.command = save_command
```

Here we have something that looks like the first command pattern, but a bit more idiomatic. As you can see, making the invoker call a callable instead of a command object with an execute method has not restricted us in any way. In fact, it's given us more flexibility. We can link to functions directly when that works, yet we can build a complete callable command object when the situation calls for it.

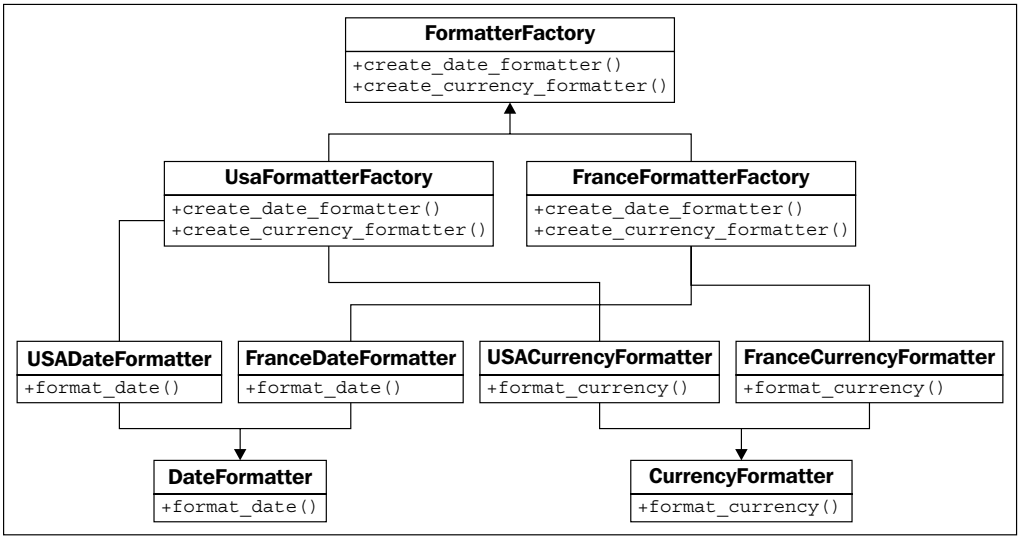
The command pattern is often extended to support undoable commands. For example, a text program may wrap each insertion in a separate command with not only an `execute` method, but also an `undo` method that will delete that insertion. A graphics program may wrap each drawing action (rectangle, line, freehand pixels, and so on) in a command that has an `undo` method that resets the pixels to their original state. In such cases, the decoupling of the command pattern is much more obviously useful, because each action has to maintain enough of its state to undo that action at a later date.

The abstract factory pattern

The abstract factory pattern is normally used when we have multiple possible implementations of a system that depend on some configuration or platform issue. The calling code requests an object from the abstract factory, not knowing exactly what class of object will be returned. The underlying implementation returned may depend on a variety of factors, such as current locale, operating system, or local configuration.

Common examples of the abstract factory pattern include code for operating-system independent toolkits, database backends, and country-specific formatters or calculators. An operating-system-independent GUI toolkit might use an abstract factory pattern that returns a set of WinForm widgets under Windows, Cocoa widgets under Mac, GTK widgets under Gnome, and QT widgets under KDE. Django provides an abstract factory that returns a set of object relational classes for interacting with a specific database backend (MySQL, PostgreSQL, SQLite, and others) depending on a configuration setting for the current site. If the application needs to be deployed in multiple places, each one can use a different database backend by changing only one configuration variable. Different countries have different systems for calculating taxes, subtotals, and totals on retail merchandise; an abstract factory can return a particular tax calculation object.

The UML class diagram for an abstract factory pattern is hard to understand without a specific example, so let's turn things around and create a concrete example first. We'll create a set of formatters that depend on a specific locale and help us format dates and currencies. There will be an abstract factory class that picks the specific factory, as well as a couple example concrete factories, one for France and one for the USA. Each of these will create formatter objects for dates and times, which can be queried to format a specific value. Here's the diagram:



Comparing that image to the earlier simpler text shows that a picture is not always worth a thousand words, especially considering we haven't even allowed for factory selection code here.

Of course, in Python, we don't have to implement any interface classes, so we can discard `DateFormatter`, `CurrencyFormatter`, and `FormatterFactory`. The formatting classes themselves are pretty straightforward, if verbose:

```
class FranceDateFormatter:
    def format_date(self, y, m, d):
        y, m, d = (str(x) for x in (y,m,d))
        y = '20' + y if len(y) == 2 else y
        m = '0' + m if len(m) == 1 else m
        d = '0' + d if len(d) == 1 else d
        return("{0}/{1}/{2}".format(d,m,y))

class USADateFormatter:
```

```
        def format_date(self, y, m, d):
            y, m, d = (str(x) for x in (y,m,d))
            y = '20' + y if len(y) == 2 else y
            m = '0' + m if len(m) == 1 else m
            d = '0' + d if len(d) == 1 else d
            return("{0}-{1}-{2}".format(m,d,y))

    class FranceCurrencyFormatter:
        def format_currency(self, base, cents):
            base, cents = (str(x) for x in (base, cents))
            if len(cents) == 0:
                cents = '00'
            elif len(cents) == 1:
                cents = '0' + cents

            digits = []
            for i,c in enumerate(reversed(base)):
                if i and not i % 3:
                    digits.append(' ')
                digits.append(c)
            base = ''.join(reversed(digits))
            return "{0}€{1}".format(base, cents)

    class USACurrencyFormatter:
        def format_currency(self, base, cents):
            base, cents = (str(x) for x in (base, cents))
            if len(cents) == 0:
                cents = '00'
            elif len(cents) == 1:
                cents = '0' + cents
            digits = []
            for i,c in enumerate(reversed(base)):
                if i and not i % 3:
                    digits.append(',')
                digits.append(c)
            base = ''.join(reversed(digits))
            return "${0}.{1}".format(base, cents)
```

These classes use some basic string manipulation to try to turn a variety of possible inputs (integers, strings of different lengths, and others) into the following formats:

	USA	**France**
Date	mm-dd-yyyy	dd/mm/yyyy
Currency	$14,500.50	14 500€50

There could obviously be more validation on the input in this code, but let's keep it simple and dumb for this example.

Now that we have the formatters set up, we just need to create the formatter factories:

```
class USAFormatterFactory:
    def create_date_formatter(self):
        return USADateFormatter()
    def create_currency_formatter(self):
        return USACurrencyFormatter()

class FranceFormatterFactory:
    def create_date_formatter(self):
        return FranceDateFormatter()
    def create_currency_formatter(self):
        return FranceCurrencyFormatter()
```

Now we set up the code that picks the appropriate formatter. Since this is the kind of thing that only needs to be set up once, we could make it a singleton—except singletons aren't very useful in Python. Let's just make the current formatter a module-level variable instead:

```
country_code = "US"
factory_map = {
        "US": USAFormatterFactory,
        "FR": FranceFormatterFactory}
formatter_factory = factory_map.get(country_code)()
```

In this example, we hardcode the current country code; in practice, it would likely introspect the locale, the operating system, or a configuration file to choose the code. This example uses a dictionary to associate the country codes with factory classes. Then we grab the correct class from the dictionary and instantiate it.

It is easy to see what needs to be done when we want to add support for more countries: create the new formatter classes and the abstract factory itself. Bear in mind that `Formatter` classes might be reused; for example, Canada formats its currency the same way as the USA, but its date format is more sensible than its Southern neighbor.

Abstract factories often return a singleton object, but this is not required; in our code, it's returning a new instance of each formatter every time it's called. There's no reason the formatters couldn't be stored as instance variables and the same instance returned for each factory.

Looking back at these examples, we see that, once again, there appears to be a lot of boilerplate code for factories that just doesn't feel necessary in Python. Often, the requirements that might call for an abstract factory can be more easily fulfilled by using a separate module for each factory type (for example: the USA and France), and then ensuring that the correct module is being accessed in a factory module. The package structure for such modules might look like this:

```
localize/
    __init__.py
    backends/
        __init__.py
        USA.py
        France.py
        …
```

The trick is that `__init__.py` in the `localize` package can contain logic that redirects all requests to the correct backend. There is a variety of ways this could be done.

If we know that the backend is never going to change dynamically (that is, without a restart), we can just put some `if` statements in `__init__.py` that check the current country code, and use the usually unacceptable `from .backends.USA import *` syntax to import all variables from the appropriate backend. Or, we could import each of the backends and set a `current_backend` variable to point at a specific module:

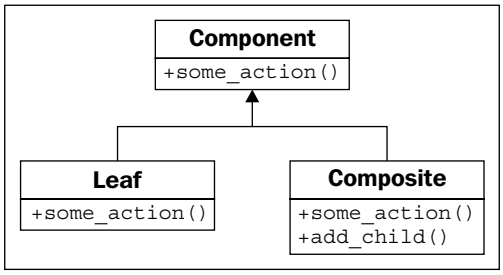
```
from .backends import USA, France

if country_code == "US":
    current_backend = USA
```

Depending on which solution we choose, our client code would have to call either `localize.format_date` or `localize.current_backend.format_date` to get a date formatted in the current country's locale. The end result is much more Pythonic than the original abstract factory pattern, and, in typical usage, just as flexible.

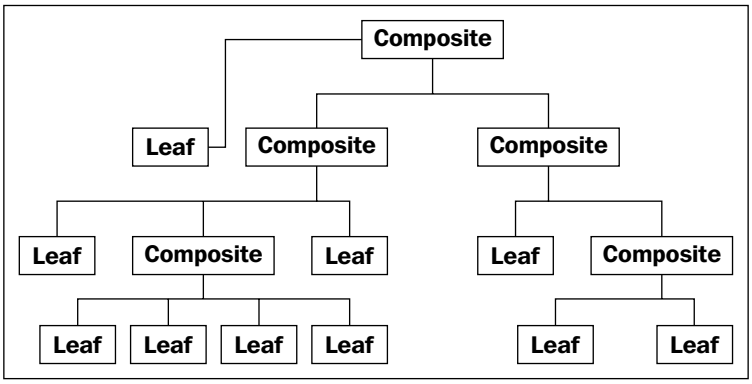
The composite pattern

The composite pattern allows complex tree-like structures to be built from simple components. These components, called composite objects, are able to behave sort of like a container and sort of like a variable depending on whether they have child components. Composite objects are container objects, where the content may actually be another composite object.

Traditionally, each component in a composite object must be either a leaf node (that cannot contain other objects) or a composite node. The key is that both composite and leaf nodes can have the same interface. The UML diagram is very simple:



This simple pattern, however, allows us to create complex arrangements of elements, all of which satisfy the interface of the component object. Here is a concrete instance of such a complicated arrangement:



The composite pattern is commonly useful in file/folder-like trees. Regardless of whether a node in the tree is a normal file or a folder, it is still subject to operations such as moving, copying, or deleting the node. We can create a component interface that supports these operations, and then use a composite object to represent folders, and leaf nodes to represent normal files.

Of course, in Python, once again, we can take advantage of duck typing to implicitly provide the interface, so we only need to write two classes. Let's define these interfaces first:

```
class Folder:
    def __init__(self, name):
        self.name = name
        self.children = {}

    def add_child(self, child):
        pass

    def move(self, new_path):
        pass

    def copy(self, new_path):
        pass

    def delete(self):
        pass

class File:
    def __init__(self, name, contents):
        self.name = name
        self.contents = contents

    def move(self, new_path):
        pass

    def copy(self, new_path):
        pass

    def delete(self):
        pass
```

For each folder (composite) object, we maintain a dictionary of children. Often, a list is sufficient, but in this case, a dictionary will be useful for looking up children by name. Our paths will be specified as node names separated by the `/` character, similar to paths in a Unix shell.

Thinking about the methods involved, we can see that moving or deleting a node behaves in a similar way, regardless of whether or not it is a file or folder node. Copying, however, has to do a recursive copy for folder nodes, while copying a file node is a trivial operation.

To take advantage of the similar operations, we can extract some of the common methods into a parent class. Let's take that discarded `Component` interface and change it to a base class:

```
class Component:
    def __init__(self, name):
        self.name = name

    def move(self, new_path):
        new_folder =get_path(new_path)
        del self.parent.children[self.name]
        new_folder.children[self.name] = self
        self.parent = new_folder

    def delete(self):
        del self.parent.children[self.name]

class Folder(Component):
    def __init__(self, name):
        super().__init__(name)
        self.children = {}

    def add_child(self, child):
        pass

    def copy(self, new_path):
        pass

class File(Component):
    def __init__(self, name, contents):
        super().__init__(name)
        self.contents = contents

    def copy(self, new_path):
        pass

root = Folder('')
def get_path(path):
```

```
        names = path.split('/')[1:]
        node = root
        for name in names:
            node = node.children[name]
        return node
```

We've created the `move` and `delete` methods on the `Component` class. Both of them access a mysterious `parent` variable that we haven't set yet. The `move` method uses a module-level `get_path` function that finds a node from a predefined root node, given a path. All files will be added to this root node or a child of that node. For the `move` method, the target should be a currently existing folder, or we'll get an error. As with many of the examples in technical books, error handling is woefully absent, to help focus on the principles under consideration.

Let's set up that mysterious `parent` variable first; this happens, in the folder's `add_child` method:

```
        def add_child(self, child):
            child.parent = self
            self.children[child.name] = child
```

Well, that was easy enough. Let's see if our composite file hierarchy is working properly:

```
$ python3 -i 1261_09_18_add_child.py


>>> folder1 = Folder('folder1')
>>> folder2 = Folder('folder2')
>>> root.add_child(folder1)
>>> root.add_child(folder2)
>>> folder11 = Folder('folder11')
>>> folder1.add_child(folder11)
>>> file111 = File('file111', 'contents')
>>> folder11.add_child(file111)
>>> file21 = File('file21', 'other contents')
>>> folder2.add_child(file21)
>>> folder2.children
{'file21': <__main__.File object at 0xb7220a4c>}
>>> folder2.move('/folder1/folder11')
>>> folder11.children
{'folder2': <__main__.Folder object at 0xb722080c>, 'file111': <__main__.
File object at 0xb72209ec>}
```

```
>>> file21.move('/folder1')
>>> folder1.children
{'file21': <__main__.File object at 0xb7220a4c>, 'folder11': <__main__.
Folder object at 0xb722084c>}
```

Yes, we can create folders, add folders to other folders, add files to folders, and move them around! What more could we ask for in a file hierarchy?

Well, we could ask for copying to be implemented, but to conserve trees, let's leave that as an exercise.

The composite pattern is extremely useful for a variety of tree-like structures, including GUI widget hierarchies, file hierarchies, tree sets, graphs, and HTML DOM. It can be a useful pattern in Python when implemented according to the traditional implementation, as the example earlier demonstrated. Sometimes, if only a shallow tree is being created, we can get away with a list of lists or a dictionary of dictionaries, and do not need to implement custom component, leaf, and composite classes. Other times, we can get away with implementing only one composite class, and treating leaf and composite objects as a single class. Alternatively, Python's duck typing can make it easy to add other objects to a composite hierarchy, as long as they have the correct interface.

Your Coding Challenge

Before diving into exercises for each design pattern, take a moment to implement the copy method for the File and Folder objects in the previous section. The File method should be quite trivial; just create a new node with the same name and contents, and add it to the new parent folder. The copy method on Folder is quite a bit more complicated, as you first have to duplicate the folder, and then recursively copy each of its children to the new location. You can call the copy() method on the children indiscriminately, regardless of whether each is a file or a folder object. This will drive home just how powerful the composite pattern can be.

Now, as with the previous chapter, look at the patterns we've discussed, and consider ideal places where you might implement them. You may want to apply the adapter pattern to existing code, as it is usually applicable when interfacing with existing libraries, rather than new code. How can you use an adapter to force two interfaces to interact with each other correctly?

Ankita Thakur

Your Course Guide

Can you think of a system complex enough to justify using the facade pattern? Consider how facades are used in real-life situations, such as the driver-facing interface of a car, or the control panel in a factory. It is similar in software, except the users of the facade interface are other programmers, rather than people trained to use them. Are there complex systems in your latest project that could benefit from the facade pattern?

It's possible you don't have any huge, memory-consuming code that would benefit from the flyweight pattern, but can you think of situations where it might be useful? Anywhere that large amounts of overlapping data need to be processed, a flyweight is waiting to be used. Would it be useful in the banking industry? In web applications? At what point does the flyweight pattern make sense? When is it overkill?

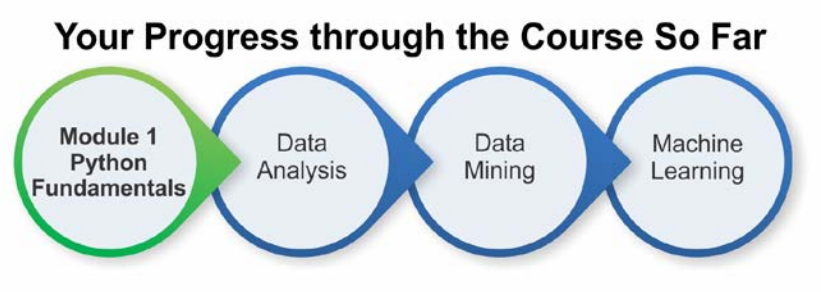
What about the command pattern? Can you think of any common (or better yet, uncommon) examples of places where the decoupling of action from invocation would be useful? Look at the programs you use on a daily basis, and imagine how they are implemented internally. It's likely that many of them use the command pattern for one purpose or another.

Ankita Thakur

Your Course Guide

The abstract factory pattern, or the somewhat more Pythonic derivatives we discussed, can be very useful for creating one-touch-configurable systems. Can you think of places where such systems are useful?

Finally, consider the composite pattern. There are tree-like structures all around us in programming; some of them, like our file hierarchy example, are blatant; others are fairly subtle. What situations might arise where the composite pattern would be useful? Can you think of places where you can use it in your own code? What if you adapted the pattern slightly; for example, to contain different types of leaf or composite nodes for different types of objects?



13
Testing Object-oriented Programs

Skilled Python programmers agree that testing is one of the most important aspects of software development. Even though this chapter is placed near the end of the module, it is not an afterthought; everything we have studied so far will help us when writing tests. We'll be studying:

- The importance of unit testing and test-driven development
- The standard `unittest` module
- The `py.test` automated testing suite
- The `mock` module
- Code coverage
- Cross-platform testing with `tox`

Why test?

A large collection of programmers already know how important it is to test their code. If you're among them, feel free to skim this section. You'll find the next section—where we actually see how to do the tests in Python—much more scintillating. If you're not convinced of the importance of testing, I promise that your code is broken, you just don't know it. Read on!

Some people argue that testing is more important in Python code because of its dynamic nature; compiled languages such as Java and C++ are occasionally thought to be somehow "safer" because they enforce type checking at compile time. However, Python tests rarely check types. They're checking values. They're making sure that the right attributes have been set at the right time or that the sequence has the right length, order, and values. These higher-level things need to be tested in any language. The real reason Python programmers test more than programmers of other languages is that it is so easy to test in Python!

But why test? Do we really need to test? What if we didn't test? To answer those questions, write a tic-tac-toe game from scratch without any testing at all. Don't run it until it is completely written, start to finish. Tic-tac-toe is fairly simple to implement if you make both players human players (no artificial intelligence). You don't even have to try to calculate who the winner is. Now run your program. And fix all the errors. How many were there? I recorded eight on my tic-tac-toe implementation, and I'm not sure I caught them all. Did you?

We need to test our code to make sure it works. Running the program, as we just did, and fixing the errors is one crude form of testing. Python programmers are able to write a few lines of code and run the program to make sure those lines are doing what they expect. But changing a few lines of code can affect parts of the program that the developer hadn't realized will be influenced by the changes, and therefore won't test it. Furthermore, as a program grows, the various paths that the interpreter can take through that code also grow, and it quickly becomes impossible to manually test all of them.

To handle this, we write automated tests. These are programs that automatically run certain inputs through other programs or parts of programs. We can run these test programs in seconds and cover more possible input situations than one programmer would think to test every time they change something.

There are four main reasons to write tests:

- To ensure that code is working the way the developer thinks it should
- To ensure that code continues working when we make changes
- To ensure that the developer understood the requirements
- To ensure that the code we are writing has a maintainable interface

The first point really doesn't justify the time it takes to write a test; we can simply test the code directly in the interactive interpreter. But when we have to perform the same sequence of test actions multiple times, it takes less time to automate those steps once and then run them whenever necessary. It is a good idea to run tests whenever we change code, whether it is during initial development or maintenance releases. When we have a comprehensive set of automated tests, we can run them after code changes and know that we didn't inadvertently break anything that was tested.

The last two points are more interesting. When we write tests for code, it helps us design the API, interface, or pattern that code takes. Thus, if we misunderstood the requirements, writing a test can help highlight that misunderstanding. On the other side, if we're not certain how we want to design a class, we can write a test that interacts with that class so we have an idea what the most natural way to test it would be. In fact, it is often beneficial to write the tests before we write the code we are testing.

Test-driven development

"Write tests first" is the mantra of test-driven development. Test-driven development takes the "untested code is broken code" concept one step further and suggests that only unwritten code should be untested. Do not write any code until you have written the tests for this code. So the first step is to write a test that proves the code would work. Obviously, the test is going to fail, since the code hasn't been written. Then write the code that ensures the test passes. Then write another test for the next segment of code.

Test-driven development is fun. It allows us to build little puzzles to solve. Then we implement the code to solve the puzzles. Then we make a more complicated puzzle, and we write code that solves the new puzzle without unsolving the previous one.

There are two goals to the test-driven methodology. The first is to ensure that tests really get written. It's so very easy, after we have written code, to say: "Hmm, it seems to work. I don't have to write any tests for this. It was just a small change, nothing could have broken." If the test is already written before we write the code, we will know exactly when it works (because the test will pass), and we'll know in the future if it is ever broken by a change we, or someone else has made.

Secondly, writing tests first forces us to consider exactly how the code will be interacted with. It tells us what methods objects need to have and how attributes will be accessed. It helps us break up the initial problem into smaller, testable problems, and then to recombine the tested solutions into larger, also tested, solutions. Writing tests can thus become a part of the design process. Often, if we're writing a test for a new object, we discover anomalies in the design that force us to consider new aspects of the software.

As a concrete example, imagine writing code that uses an object-relational mapper to store object properties in a database. It is common to use an automatically assigned database ID in such objects. Our code might use this ID for various purposes. If we are writing a test for such code, before we write it, we may realize that our design is faulty because objects do not have these IDs until they have been saved to the database. If we want to manipulate an object without saving it in our test, it will highlight this problem before we have written code based on the faulty premise.

Testing makes software better. Writing tests before we release the software makes it better before the end user sees or purchases the buggy version (I have worked for companies that thrive on the "the users can test it" philosophy. It's not a healthy business model!). Writing tests before we write software makes it better the first time it is written.

Unit testing

Let's start our exploration with Python's built-in test library. This library provides a common interface for **unit tests**. Unit tests focus on testing the least amount of code possible in any one test. Each one tests a single unit of the total amount of available code.

The Python library for this is called, unsurprisingly, `unittest`. It provides several tools for creating and running unit tests, the most important being the `TestCase` class. This class provides a set of methods that allow us to compare values, set up tests, and clean up when they have finished.

When we want to write a set of unit tests for a specific task, we create a subclass of `TestCase`, and write individual methods to do the actual testing. These methods must all start with the name `test`. When this convention is followed, the tests automatically run as part of the test process. Normally, the tests set some values on an object and then run a method, and use the built-in comparison methods to ensure that the right results were calculated. Here's a very simple example:

```
import unittest

class CheckNumbers(unittest.TestCase):
    def test_int_float(self):
        self.assertEqual(1, 1.0)

if __name__ == "__main__":
    unittest.main()
```

This code simply subclasses the `TestCase` class and adds a method that calls the `TestCase.assertEqual` method. This method will either succeed or raise an exception, depending on whether the two parameters are equal. If we run this code, the `main` function from `unittest` will give us the following output:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Did you know that floats and integers can compare as equal? Let's add a failing test:

```
        def test_str_float(self):
            self.assertEqual(1, "1")
```

The output of this code is more sinister, as integers and strings are not considered equal:

```
.F
======================================================================
FAIL: test_str_float (__main__.CheckNumbers)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "simplest_unittest.py", line 8, in test_str_float
    self.assertEqual(1, "1")
AssertionError: 1 != '1'

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

The dot on the first line indicates that the first test (the one we wrote before) passed successfully; the letter `F` after it shows that the second test failed. Then, at the end, it gives us some informative output telling us how and where the test failed, along with a summary of the number of failures.

We can have as many test methods on one `TestCase` class as we like; as long as the method name begins with `test`, the test runner will execute each one as a separate test. Each test should be completely independent of other tests. Results or calculations from a previous test should have no impact on the current test. The key to writing good unit tests is to keep each test method as short as possible, testing a small unit of code with each test case. If your code does not seem to naturally break up into such testable units, it's probably a sign that your design needs rethinking.

Assertion methods

The general layout of a test case is to set certain variables to known values, run one or more functions, methods, or processes, and then "prove" that correct expected results were returned or calculated by using `TestCase` assertion methods.

There are a few different assertion methods available to confirm that specific results have been achieved. We just saw `assertEqual`, which will cause a test failure if the two parameters do not pass an equality check. The inverse, `assertNotEqual`, will fail if the two parameters do compare as equal. The `assertTrue` and `assertFalse` methods each accept a single expression, and fail if the expression does not pass an `if` test. These tests are not checking for the Boolean values `True` or `False`. Rather, they test the same condition as though an `if` statement were used: `False`, `None`, `0`, or an empty list, dictionary, string, set, or tuple would pass a call to the `assertFalse` method, while nonzero numbers, containers with values in them, or the value `True` would succeed when calling the `assertTrue` method.

There is an `assertRaises` method that can be used to ensure a specific function call raises a specific exception or, optionally, it can be used as a context manager to wrap inline code. The test passes if the code inside the `with` statement raises the proper exception; otherwise, it fails. Here's an example of both versions:

```
import unittest

def average(seq):
    return sum(seq) / len(seq)

class TestAverage(unittest.TestCase):
    def test_zero(self):
        self.assertRaises(ZeroDivisionError,
                average,
```

```
                []):

    def test_with_zero(self):
        with self.assertRaises(ZeroDivisionError):
            average([])

if __name__ == "__main__":
    unittest.main()
```

The context manager allows us to write the code the way we would normally write it (by calling functions or executing code directly), rather than having to wrap the function call in another function call.

There are also several other assertion methods, summarized in the following table:

Methods	Description

`assertGreater`
`assertGreaterEqual`
`assertLess`
`assertLessEqual`
	Accept two comparable objects and ensure the named inequality holds.

`assertIn` `assertNotIn`	Ensure an element is (or is not) an element in a container object.
`assertIsNone` `assertIsNotNone`	Ensure an element is (or is not) the exact value `None` (but not another falsey value).
`assertSameElements`	Ensure two container objects have the same elements, ignoring the order.

`assertSequenceEqualassertDictEqual`
`assertSetEqual`
`assertListEqual`
`assertTupleEqual`
	Ensure two containers have the same elements in the same order. If there's a failure, show a code diff comparing the two lists to see where they differ. The last four methods also test the type of the list.

Each of the assertion methods accepts an optional argument named `msg`. If supplied, it is included in the error message if the assertion fails. This is useful for clarifying what was expected or explaining where a bug may have occurred to cause the assertion to fail.

Reducing boilerplate and cleaning up

After writing a few small tests, we often find that we have to do the same setup code for several related tests. For example, the following `list` subclass has three methods for statistical calculations:

```
from collections import defaultdict

class StatsList(list):
    def mean(self):
        return sum(self) / len(self)

    def median(self):
        if len(self) % 2:
            return self[int(len(self) / 2)]
        else:
            idx = int(len(self) / 2)
            return (self[idx] + self[idx-1]) / 2

    def mode(self):
        freqs = defaultdict(int)
        for item in self:
            freqs[item] += 1
        mode_freq = max(freqs.values())
        modes = []
        for item, value in freqs.items():
            if value == mode_freq:
                modes.append(item)
        return modes
```

Clearly, we're going to want to test situations with each of these three methods that have very similar inputs; we'll want to see what happens with empty lists or with lists containing non-numeric values or with lists containing a normal dataset. We can use the `setUp` method on the `TestCase` class to do initialization for each test. This method accepts no arguments, and allows us to do arbitrary setup before each test is run. For example, we can test all three methods on identical lists of integers as follows:

```
from stats import StatsList
import unittest

class TestValidInputs(unittest.TestCase):
    def setUp(self):
        self.stats = StatsList([1,2,2,3,3,4])

    def test_mean(self):
```

```
            self.assertEqual(self.stats.mean(), 2.5)

        def test_median(self):
            self.assertEqual(self.stats.median(), 2.5)
            self.stats.append(4)
            self.assertEqual(self.stats.median(), 3)

        def test_mode(self):
            self.assertEqual(self.stats.mode(), [2,3])
            self.stats.remove(2)
            self.assertEqual(self.stats.mode(), [3])

    if __name__ == "__main__":
        unittest.main()
```

If we run this example, it indicates that all tests pass. Notice first that the `setUp` method is never explicitly called inside the three `test_*` methods. The test suite does this on our behalf. More importantly notice how `test_median` alters the list, by adding an additional `4` to it, yet when `test_mode` is called, the list has returned to the values specified in `setUp` (if it had not, there would be two fours in the list, and the `mode` method would have returned three values). This shows that `setUp` is called individually before each test, to ensure the test class starts with a clean slate. Tests can be executed in any order, and the results of one test should not depend on any other tests.

In addition to the `setUp` method, `TestCase` offers a no-argument `tearDown` method, which can be used for cleaning up after each and every test on the class has run. This is useful if cleanup requires anything other than letting an object be garbage collected. For example, if we are testing code that does file I/O, our tests may create new files as a side effect of testing; the `tearDown` method can remove these files and ensure the system is in the same state it was before the tests ran. Test cases should never have side effects. In general, we group test methods into separate `TestCase` subclasses depending on what setup code they have in common. Several tests that require the same or similar setup will be placed in one class, while tests that require unrelated setup go in another class.

Organizing and running tests

It doesn't take long for a collection of unit tests to grow very large and unwieldy. It quickly becomes complicated to load and run all the tests at once. This is a primary goal of unit testing; it should be trivial to run all tests on our program and get a quick "yes or no" answer to the question, "Did my recent changes break any existing tests?".

Python's `discover` module basically looks for any modules in the current folder or subfolders with names that start with the characters `test`. If it finds any `TestCase` objects in these modules, the tests are executed. It's a painless way to ensure we don't miss running any tests. To use it, ensure your test modules are named `test_<something>.py` and then run the command `python3 -m unittest discover`.

Ignoring broken tests

Sometimes, a test is known to fail, but we don't want the test suite to report the failure. This may be because a broken or unfinished feature has had tests written, but we aren't currently focusing on improving it. More often, it happens because a feature is only available on a certain platform, Python version, or for advanced versions of a specific library. Python provides us with a few decorators to mark tests as expected to fail or to be skipped under known conditions.

The decorators are:

- `expectedFailure()`
- `skip(reason)`
- `skipIf(condition, reason)`
- `skipUnless(condition, reason)`

These are applied using the Python decorator syntax. The first one accepts no arguments, and simply tells the test runner not to record the test as a failure when it fails. The `skip` method goes one step further and doesn't even bother to run the test. It expects a single string argument describing why the test was skipped. The other two decorators accept two arguments, one a Boolean expression that indicates whether or not the test should be run, and a similar description. In use, these three decorators might be applied like this:

```
import unittest
import sys

class SkipTests(unittest.TestCase):
    @unittest.expectedFailure
    def test_fails(self):
        self.assertEqual(False, True)

    @unittest.skip("Test is useless")
```

```
        def test_skip(self):
            self.assertEqual(False, True)

        @unittest.skipIf(sys.version_info.minor == 4,
                "broken on 3.4")
        def test_skipif(self):
            self.assertEqual(False, True)

        @unittest.skipUnless(sys.platform.startswith('linux'),
                "broken unless on linux")
        def test_skipunless(self):
            self.assertEqual(False, True)

    if __name__ == "__main__":
        unittest.main()
```

The first test fails, but it is reported as an expected failure; the second test is never run. The other two tests may or may not be run depending on the current Python version and operating system. On my Linux system running Python 3.4, the output looks like this:

```
xssF
======================================================================
FAIL: test_skipunless (__main__.SkipTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "skipping_tests.py", line 21, in test_skipunless
    self.assertEqual(False, True)
AssertionError: False != True


----------------------------------------------------------------------
Ran 4 tests in 0.001s


FAILED (failures=1, skipped=2, expected failures=1)
```

The `x` on the first line indicates an expected failure; the two `s` characters represent skipped tests, and the `F` indicates a real failure, since the conditional to `skipUnless` was `True` on my system.

Testing with py.test

The Python `unittest` module requires a lot of boilerplate code to set up and initialize tests. It is based on the very popular JUnit testing framework for Java. It even uses the same method names (you may have noticed they don't conform to the PEP-8 naming standard, which suggests underscores rather than CamelCase to separate words in a method name) and test layout. While this is effective for testing in Java, it's not necessarily the best design for Python testing.

Because Python programmers like their code to be elegant and simple, other test frameworks have been developed, outside the standard library. Two of the more popular ones are `py.test` and `nose`. The former is more robust and has had Python 3 support for much longer, so we'll discuss it here.

Since `py.test` is not part of the standard library, you'll need to download and install it yourself; you can get it from the `py.test` home page at `http://pytest.org/`. The website has comprehensive installation instructions for a variety of interpreters and platforms, but you can usually get away with the more common python package installer, pip. Just type `pip install pytest` on your command line and you'll be good to go.

`py.test` has a substantially different layout from the `unittest` module. It doesn't require test cases to be classes. Instead, it takes advantage of the fact that Python functions are objects, and allows any properly named function to behave like a test. Rather than providing a bunch of custom methods for asserting equality, it uses the `assert` statement to verify results. This makes tests more readable and maintainable. When we run `py.test`, it will start in the current folder and search for any modules in that folder or subpackages whose names start with the characters `test_`. If any functions in this module also start with `test`, they will be executed as individual tests. Furthermore, if there are any classes in the module whose name starts with `Test`, any methods on that class that start with `test_` will also be executed in the test environment.

Let's port the simplest possible `unittest` example we wrote earlier to `py.test`:

```
def test_int_float():
    assert 1 == 1.0
```

For the exact same test, we've written two lines of more readable code, in comparison to the six lines required in our first `unittest` example.

However, we are not forbidden from writing class-based tests. Classes can be useful for grouping related tests together or for tests that need to access related attributes or methods on the class. This example shows an extended class with a passing and a failing test; we'll see that the error output is more comprehensive than that provided by the `unittest` module:

```
class TestNumbers:
    def test_int_float(self):
        assert 1 == 1.0

    def test_int_str(self):
        assert 1 == "1"
```

Notice that the class doesn't have to extend any special objects to be picked up as a test (although `py.test` will run standard `unittest TestCases` just fine). If we run `py.test <filename>`, the output looks like this:

```
============== test session starts ==============
python: platform linux2 -- Python 3.4.1 -- pytest-2.6.4
test object 1: class_pytest.py

class_pytest.py .F

=================== FAILURES====================
___________ TestNumbers.test_int_str _____________

self = <class_pytest.TestNumbers object at 0x85b4fac>

    def test_int_str(self):
>       assert 1 == "1"
E       assert 1 == '1'

class_pytest.py:7: AssertionError
====== 1 failed, 1 passed in 0.10 seconds =======
```

The output starts with some useful information about the platform and interpreter. This can be useful for sharing bugs across disparate systems. The third line tells us the name of the file being tested (if there are multiple test modules picked up, they will all be displayed), followed by the familiar `.F` we saw in the `unittest` module; the `.` character indicates a passing test, while the letter `F` demonstrates a failure.

After all tests have run, the error output for each of them is displayed. It presents a summary of local variables (there is only one in this example: the `self` parameter passed into the function), the source code where the error occurred, and a summary of the error message. In addition, if an exception other than an `AssertionError` is raised, `py.test` will present us with a complete traceback, including source code references.

By default, `py.test` suppresses output from `print` statements if the test is successful. This is useful for test debugging; when a test is failing, we can add `print` statements to the test to check the values of specific variables and attributes as the test runs. If the test fails, these values are output to help with diagnosis. However, once the test is successful, the `print` statement output is not displayed, and they can be easily ignored. We don't have to "clean up" the output by removing `print` statements. If the tests ever fail again, due to future changes, the debugging output will be immediately available.

One way to do setup and cleanup

`py.test` supports setup and teardown methods similar to those used in `unittest`, but it provides even more flexibility. We'll discuss these briefly, since they are familiar, but they are not used as extensively as in the `unittest` module, as `py.test` provides us with a powerful funcargs facility, which we'll discuss in the next section.

If we are writing class-based tests, we can use two methods called `setup_method` and `teardown_method` in basically the same way that `setUp` and `tearDown` are called in `unittest`. They are called before and after each test method in the class to perform setup and cleanup duties. There is one difference from the `unittest` methods though. Both methods accept an argument: the function object representing the method being called.

In addition, `py.test` provides other setup and teardown functions to give us more control over when setup and cleanup code is executed. The `setup_class` and `teardown_class` methods are expected to be class methods; they accept a single argument (there is no `self` argument) representing the class in question.

Finally, we have the `setup_module` and `teardown_module` functions, which are run immediately before and after all tests (in functions or classes) in that module. These can be useful for "one time" setup, such as creating a socket or database connection that will be used by all tests in the module. Be careful with this one, as it can accidentally introduce dependencies between tests if the object being set up stores the state.

That short description doesn't do a great job of explaining exactly when these methods are called, so let's look at an example that illustrates exactly when it happens:

```
def setup_module(module):
    print("setting up MODULE {0}".format(
        module.__name__))

def teardown_module(module):
    print("tearing down MODULE {0}".format(
        module.__name__))

def test_a_function():
    print("RUNNING TEST FUNCTION")

class BaseTest:
    def setup_class(cls):
        print("setting up CLASS {0}".format(
            cls.__name__))

    def teardown_class(cls):
        print("tearing down CLASS {0}\n".format(
            cls.__name__))

    def setup_method(self, method):
        print("setting up METHOD {0}".format(
            method.__name__))

    def teardown_method(self, method):
        print("tearing down  METHOD {0}".format(
            method.__name__))

class TestClass1(BaseTest):
    def test_method_1(self):
        print("RUNNING METHOD 1-1")

    def test_method_2(self):
```

```
            print("RUNNING METHOD 1-2")

    class TestClass2(BaseTest):
        def test_method_1(self):
            print("RUNNING METHOD 2-1")

        def test_method_2(self):
            print("RUNNING METHOD 2-2")
```

The sole purpose of the `BaseTest` class is to extract four methods that would be otherwise identical to the test classes, and use inheritance to reduce the amount of duplicate code. So, from the point of view of `py.test`, the two subclasses have not only two test methods each, but also two setup and two teardown methods (one at the class level, one at the method level).

If we run these tests using `py.test` with the `print` function output suppression disabled (by passing the `-s` or `--capture=no` flag), they show us when the various functions are called in relation to the tests themselves:

```
py.test setup_teardown.py -s
setup_teardown.py
setting up MODULE setup_teardown
RUNNING TEST FUNCTION
.setting up CLASS TestClass1
setting up METHOD test_method_1
RUNNING METHOD 1-1
.tearing down  METHOD test_method_1
setting up METHOD test_method_2
RUNNING METHOD 1-2
.tearing down  METHOD test_method_2
tearing down CLASS TestClass1
setting up CLASS TestClass2
setting up METHOD test_method_1
RUNNING METHOD 2-1
.tearing down  METHOD test_method_1
setting up METHOD test_method_2
RUNNING METHOD 2-2
.tearing down  METHOD test_method_2
tearing down CLASS TestClass2

tearing down MODULE setup_teardown
```

The setup and teardown methods for the module are executed at the beginning and end of the session. Then the lone module-level test function is run. Next, the setup method for the first class is executed, followed by the two tests for that class. These tests are each individually wrapped in separate `setup_method` and `teardown_method` calls. After the tests have executed, the class teardown method is called. The same sequence happens for the second class, before the `teardown_module` method is finally called, exactly once.

A completely different way to set up variables

One of the most common uses for the various setup and teardown functions is to ensure certain class or module variables are available with a known value before each test method is run.

`py.test` offers a completely different way to do this using what are known as **funcargs**, short for function arguments. Funcargs are basically named variables that are predefined in a test configuration file. This allows us to separate configuration from execution of tests, and allows the funcargs to be used across multiple classes and modules.

To use them, we add parameters to our test function. The names of the parameters are used to look up specific arguments in specially named functions. For example, if we wanted to test the `StatsList` class we used while demonstrating `unittest`, we would again want to repeatedly test a list of valid integers. But we can write our tests like so instead of using a setup method:

```
from stats import StatsList

def pytest_funcarg__valid_stats(request):
    return StatsList([1,2,2,3,3,4])


def test_mean(valid_stats):
    assert valid_stats.mean() == 2.5

def test_median(valid_stats):
    assert valid_stats.median() == 2.5
    valid_stats.append(4)
    assert valid_stats.median() == 3

def test_mode(valid_stats):
    assert valid_stats.mode() == [2,3]
    valid_stats.remove(2)
    assert valid_stats.mode() == [3]
```

Each of the three test methods accepts a parameter named `valid_stats`; this parameter is created by calling the `pytest_funcarg__valid_stats` function defined at the top of the file. It can also be defined in a file called `conftest.py` if the funcarg is needed by multiple modules. The `conftest.py` file is parsed by `py.test` to load any "global" test configuration; it is a sort of catch-all for customizing the `py.test` experience.

As with other `py.test` features, the name of the factory for returning a funcarg is important; funcargs are functions that are named `pytest_funcarg__<identifier>`, where `<identifier>` is a valid variable name that can be used as a parameter in a test function. This function accepts a mysterious `request` parameter, and returns the object to be passed as an argument into the individual test functions. The funcarg is created afresh for each call to an individual test function; this allows us, for example, to change the list in one test and know that it will be reset to its original values in the next test.

Funcargs can do a lot more than return basic variables. That `request` object passed into the funcarg factory provides some extremely useful methods and attributes to modify the funcarg's behavior. The `module`, `cls`, and `function` attributes allow us to see exactly which test is requesting the funcarg. The `config` attribute allows us to check command-line arguments and other configuration data.

More interestingly, the request object provides methods that allow us to do additional cleanup on the funcarg, or to reuse it across tests, activities that would otherwise be relegated to setup and teardown methods of a specific scope.

The `request.addfinalizer` method accepts a callback function that performs cleanup after each test function that uses the funcarg has been called. This provides the equivalent of a teardown method, allowing us to clean up files, close connections, empty lists, or reset queues. For example, the following code tests the `os.mkdir` functionality by creating a temporary directory `funcarg`:

```
import tempfile
import shutil
import os.path

def pytest_funcarg__temp_dir(request):
    dir = tempfile.mkdtemp()
    print(dir)

    def cleanup():
        shutil.rmtree(dir)
    request.addfinalizer(cleanup)
```

```
    return dir

def test_osfiles(temp_dir):
    os.mkdir(os.path.join(temp_dir, 'a'))
    os.mkdir(os.path.join(temp_dir, 'b'))
    dir_contents = os.listdir(temp_dir)
    assert len(dir_contents) == 2
    assert 'a' in dir_contents
    assert 'b' in dir_contents
```

The funcarg creates a new empty temporary directory for files to be created in. Then it adds a finalizer call to remove that directory (using `shutil.rmtree`, which recursively removes a directory and anything inside it) after the test has completed. The filesystem is then left in the same state in which it started.

We can use the `request.cached_setup` method to create function argument variables that last longer than one test. This is useful when setting up an expensive operation that can be reused by multiple tests as long as the resource reuse doesn't break the atomic or unit nature of the tests (so that one test does not rely on and is not impacted by a previous one). For example, if we were to test the following echo server, we may want to run only one instance of the server in a separate process, and then have multiple tests connect to that instance:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(('localhost',1028))
s.listen(1)

    while True:
        client, address = s.accept()
        data = client.recv(1024)
        client.send(data)
        client.close()
```

All this code does is listen on a specific port and wait for input from a client socket. When it receives input, it sends the same value back. To test this, we can start the server in a separate process and cache the result for use in multiple tests. Here's how the test code might look:

```
import subprocess
import socket
```

```
import time

def pytest_funcarg__echoserver(request):
    def setup():
        p = subprocess.Popen(
                ['python3', 'echo_server.py'])
        time.sleep(1)
        return p

    def cleanup(p):
        p.terminate()

    return request.cached_setup(
            setup=setup,
            teardown=cleanup,
            scope="session")

def pytest_funcarg__clientsocket(request):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('localhost', 1028))
    request.addfinalizer(lambda: s.close())
    return s

def test_echo(echoserver, clientsocket):
    clientsocket.send(b"abc")
    assert clientsocket.recv(3) == b'abc'

def test_echo2(echoserver, clientsocket):
    clientsocket.send(b"def")
    assert clientsocket.recv(3) == b'def'
```

We've created two funcargs here. The first runs the echo server in a separate process, and returns the process object. The second instantiates a new socket object for each test, and closes it when the test has completed, using `addfinalizer`. The first funcarg is the one we're currently interested in. It looks much like a traditional unit test setup and teardown. We create a `setup` function that accepts no parameters and returns the correct argument; in this case, a process object that is actually ignored by the tests, since they only care that the server is running. Then, we create a `cleanup` function (the name of the function is arbitrary since it's just an object we pass into another function), which accepts a single argument: the argument returned by `setup`. This cleanup code terminates the process.

Instead of returning a funcarg directly, the parent function returns the results of a call to `request.cached_setup`. It accepts two arguments for the `setup` and `teardown` functions (which we just created), and a `scope` argument. This last argument should be one of the three strings "function", "module", or "session"; it determines just how long the argument will be cached. We set it to "session" in this example, so it is cached for the duration of the entire `py.test` run. The process will not be terminated or restarted until all tests have run. The "module" scope, of course, caches it only for tests in that module, and the "function" scope treats the object more like a normal funcarg, in that it is reset after each test function is run.

Skipping tests with py.test

As with the `unittest` module, it is frequently necessary to skip tests in `py.test`, for a variety of reasons: the code being tested hasn't been written yet, the test only runs on certain interpreters or operating systems, or the test is time consuming and should only be run under certain circumstances.

We can skip tests at any point in our code using the `py.test.skip` function. It accepts a single argument: a string describing why it has been skipped. This function can be called anywhere; if we call it inside a test function, the test will be skipped. If we call it at the module level, all the tests in that module will be skipped. If we call it inside a funcarg function, all tests that call that funcarg will be skipped.

Of course, in all these locations, it is often desirable to skip tests only if certain conditions are or are not met. Since we can execute the `skip` function at any place in Python code, we can execute it inside an `if` statement. So we may write a test that looks like this:

```
import sys
import py.test

def test_simple_skip():
    if sys.platform != "fakeos":
        py.test.skip("Test works only on fakeOS")

    fakeos.do_something_fake()
    assert fakeos.did_not_happen
```

That's some pretty silly code, really. There is no Python platform named `fakeos`, so this test will skip on all operating systems. It shows how we can skip conditionally, and since the `if` statement can check any valid conditional, we have a lot of power over when tests are skipped. Often, we check `sys.version_info` to check the Python interpreter version, `sys.platform` to check the operating system, or `some_library.__version__` to check whether we have a recent enough version of a given API.

Since skipping an individual test method or function based on a certain conditional is one of the most common uses of test skipping, `py.test` provides a convenience decorator that allows us to do this in one line. The decorator accepts a single string, which can contain any executable Python code that evaluates to a Boolean value. For example, the following test will only run on Python 3 or higher:

```
import py.test

@py.test.mark.skipif("sys.version_info <= (3,0)")
def test_python3():
    assert b"hello".decode() == "hello"
```

The `py.test.mark.xfail` decorator behaves similarly, except that it marks a test as expected to fail, similar to `unittest.expectedFailure()`. If the test is successful, it will be recorded as a failure; if it fails, it will be reported as expected behavior. In the case of `xfail`, the conditional argument is optional; if it is not supplied, the test will be marked as expected to fail under all conditions.

Imitating expensive objects

Sometimes, we want to test code that requires an object be supplied that is either expensive or difficult to construct. While this may mean your API needs rethinking to have a more testable interface (which typically means a more usable interface), we sometimes find ourselves writing test code that has a ton of boilerplate to set up objects that are only incidentally related to the code under test.

For example, imagine we have some code that keeps track of flight statuses in a key-value store (such as `redis` or `memcache`) such that we can store the timestamp and the most recent status. A basic version of such code might look like this:

```
import datetime
import redis

class FlightStatusTracker:
    ALLOWED_STATUSES = {'CANCELLED', 'DELAYED', 'ON TIME'}

    def __init__(self):
        self.redis = redis.StrictRedis()

    def change_status(self, flight, status):
```

```
        status = status.upper()
        if status not in self.ALLOWED_STATUSES:
            raise ValueError(
                   "{} is not a valid status".format(status))

        key = "flightno:{}".format(flight)
        value = "{}|{}".format(
            datetime.datetime.now().isoformat(), status)
        self.redis.set(key, value)
```

There are a lot of things we ought to test in that `change_status` method. We should check that it raises the appropriate error if a bad status is passed in. We need to ensure that it converts statuses to uppercase. We can see that the key and value have the correct formatting when the `set()` method is called on the `redis` object.

One thing we don't have to check in our unit tests, however, is that the `redis` object is properly storing the data. This is something that absolutely should be tested in integration or application testing, but at the unit test level, we can assume that the py-redis developers have tested their code and that this method does what we want it to. As a rule, unit tests should be self-contained and not rely on the existence of outside resources, such as a running Redis instance.

Instead, we only need to test that the `set()` method was called the appropriate number of times and with the appropriate arguments. We can use `Mock()` objects in our tests to replace the troublesome method with an object we can introspect. The following example illustrates the use of mock:

```
from unittest.mock import Mock
import py.test
def pytest_funcarg__tracker():
    return FlightStatusTracker()

def test_mock_method(tracker):
    tracker.redis.set = Mock()
    with py.test.raises(ValueError) as ex:
        tracker.change_status("AC101", "lost")
    assert ex.value.args[0] == "LOST is not a valid status"
    assert tracker.redis.set.call_count == 0
```

This test, written using `py.test` syntax, asserts that the correct exception is raised when an inappropriate argument is passed in. In addition, it creates a mock object for the `set` method and makes sure that it is never called. If it was, it would mean there was a bug in our exception handling code.

Simply replacing the method worked fine in this case, since the object being replaced was destroyed in the end. However, we often want to replace a function or method only for the duration of a test. For example, if we want to test the timestamp formatting in the mock method, we need to know exactly what `datetime.datetime.now()` is going to return. However, this value changes from run to run. We need some way to pin it to a specific value so we can test it deterministically.

Remember monkey-patching? Temporarily setting a library function to a specific value is an excellent use of it. The mock library provides a patch context manager that allows us to replace attributes on existing libraries with mock objects. When the context manager exits, the original attribute is automatically restored so as not to impact other test cases. Here's an example:

```
from unittest.mock import patch
def test_patch(tracker):
    tracker.redis.set = Mock()
    fake_now = datetime.datetime(2015, 4, 1)
    with patch('datetime.datetime') as dt:
        dt.now.return_value = fake_now
        tracker.change_status("AC102", "on time")
    dt.now.assert_called_once_with()
    tracker.redis.set.assert_called_once_with(
        "flightno:AC102",
        "2015-04-01T00:00:00|ON TIME")
```

In this example, we first construct a value called `fake_now`, which we will set as the return value of the `datetime.datetime.now` function. We have to construct this object before we patch `datetime.datetime` because otherwise we'd be calling the patched `now` function before we constructed it!

The `with` statement invites the patch to replace the `datetime.datetime` module with a mock object, which is returned as the value `dt`. The neat thing about mock objects is that any time you access an attribute or method on that object, it returns another mock object. Thus when we access `dt.now`, it gives us a new mock object. We set the `return_value` of that object to our `fake_now` object; that way, whenever the `datetime.datetime.now` function is called, it will return our object instead of a new mock object.

Then, after calling our `change_status` method with known values, we use the mock class's `assert_called_once_with` function to ensure that the `now` function was indeed called exactly once with no arguments. We then call it a second time to prove that the `redis.set` method was called with arguments that were formatted as we expected them to be.

The previous example is a good indication of how writing tests can guide our API design. The `FlightStatusTracker` object looks sensible at first glance; we construct a `redis` connection when the object is constructed, and we call into it when we need it. When we write tests for this code, however, we discover that even if we mock out that `self.redis` variable on a `FlightStatusTracker`, the `redis` connection still has to be constructed. This call actually fails if there is no Redis server running, and our tests also fail.

We could solve this problem by mocking out the `redis.StrictRedis` class to return a mock in a `setUp` method. A better idea, however, might be to rethink our example. Instead of constructing the `redis` instance inside`__init__`, perhaps we should allow the user to pass one in, as in the following example:

```
    def __init__(self, redis_instance=None):
        self.redis = redis_instance if redis_instance else redis.
StrictRedis()
```

This allows us to pass a mock in when we are testing, so the `StrictRedis` method never gets constructed. However, it also allows any client code that talks to `FlightStatusTracker` to pass in their own `redis` instance. There are a variety of reasons they might want to do this. They may have already constructed one for other parts of their code. They may have created an optimized implementation of the `redis` API. Perhaps they have one that logs metrics to their internal monitoring systems. By writing a unit test, we've uncovered a use case that makes our API more flexible from the start, rather than waiting for clients to demand we support their exotic needs.

This has been a brief introduction to the wonders of mocking code. Mocks are part of the standard `unittest` library since Python 3.3, but as you see from these examples, they can also be used with `py.test` and other libraries. Mocks have other more advanced features that you may need to take advantage of as your code gets more complicated. For example, you can use the `spec` argument to invite a mock to imitate an existing class so that it raises an error if code tries to access an attribute that does not exist on the imitated class. You can also construct mock methods that return different arguments each time they are called by passing a list as the `side_effect` argument. The `side_effect` parameter is quite versatile; you can also use it to execute arbitrary functions when the mock is called or to raise an exception.

In general, we should be quite stingy with mocks. If we find ourselves mocking out multiple elements in a given unit test, we may end up testing the mock framework rather than our real code. This serves no useful purpose whatsoever; after all, mocks are well-tested already! If our code is doing a lot of this, it's probably another sign that the API we are testing is poorly designed. Mocks should exist at the boundaries between the code under test and the libraries they interface with. If this isn't happening, we may need to change the API so that the boundaries are redrawn in a different place.

How much testing is enough?

We've already established that untested code is broken code. But how can we tell how well our code is tested? How do we know how much of our code is actually being tested and how much is broken? The first question is the more important one, but it's hard to answer. Even if we know we have tested every line of code in our application, we do not know that we have tested it properly. For example, if we write a stats test that only checks what happens when we provide a list of integers, it may still fail spectacularly if used on a list of floats or strings or self-made objects. The onus of designing complete test suites still lies with the programmer.

The second question—how much of our code is actually being tested—is easy to verify. Code coverage is essentially an estimate of the number of lines of code that are executed by a program. If we know that number and the number of lines that are in the program, we can get an estimate of what percentage of the code was really tested, or covered. If we additionally have an indicator as to which lines were not tested, we can more easily write new tests to ensure those lines are less broken.

The most popular tool for testing code coverage is called, memorably enough, `coverage.py`. It can be installed like most other third-party libraries using the command `pip install coverage`.

We don't have space to cover all the details of the coverage API, so we'll just look at a few typical examples. If we have a Python script that runs all our unit tests for us (for example, using `unittest.main`, a custom test runner or `discover`), we can use the following command to perform a coverage analysis:

```
coverage run coverage_unittest.py
```

This command will exit normally, but it creates a file named `.coverage` that holds the data from the run. We can now use the `coverage report` command to get an analysis of code coverage:

```
>>> coverage report
```

The output is as follows:

```
Name                              Stmts   Exec  Cover
-------------------------------------------------
coverage_unittest                     7      7   100%
stats                                19      6    31%
-------------------------------------------------
TOTAL                                26     13    50%
```

This basic report lists the files that were executed (our unit test and a module it imported). The number of lines of code in each file, and the number that were executed by the test are also listed. The two numbers are then combined to estimate the amount of code coverage. If we pass the `-m` option to the report command, it will additionally add a column that looks like this:

```
Missing
-----------
8-12, 15-23
```

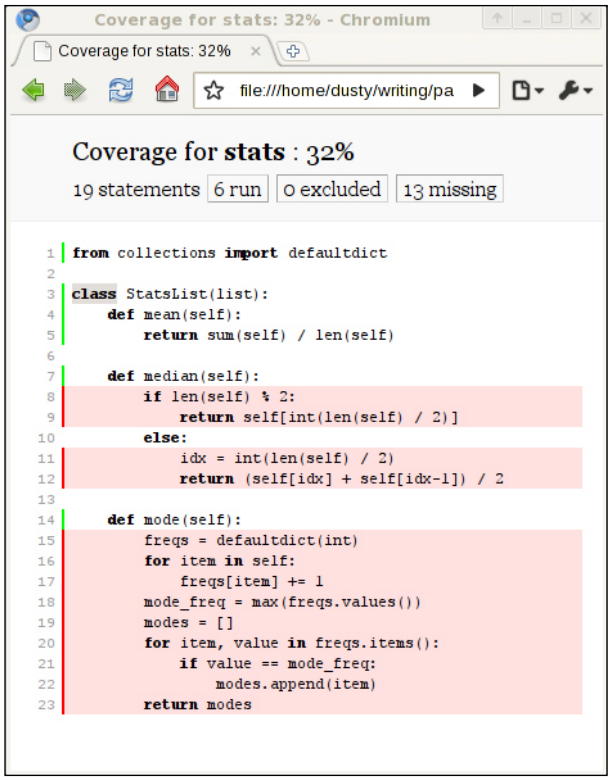
The ranges of lines listed here identify lines in the `stats` module that were not executed during the test run.

The example we just ran the code coverage tool on uses the same stats module we created earlier in the chapter. However, it deliberately uses a single test that fails to test a lot of code in the file. Here's the test:

```
from stats import StatsList
import unittest

class TestMean(unittest.TestCase):
    def test_mean(self):
        self.assertEqual(StatsList([1,2,2,3,3,4]).mean(), 2.5)

if __name__ == "__main__":

    unittest.main()
```

This code doesn't test the median or mode functions, which correspond to the line numbers that the coverage output told us were missing.

The textual report is sufficient, but if we use the command `coverage html`, we can get an even fancier interactive HTML report that we can view in a web browser. The web page even highlights which lines in the source code were and were not tested. Here's how it looks:



We can use the `coverage.py` module with `py.test` as well. We'll need to install the `py.test` plugin for code coverage, using `pip install pytest-coverage`. The plugin adds several command-line options to `py.test`, the most useful being `--cover-report`, which can be set to `html`, `report`, or `annotate` (the latter actually modifies the source code to highlight any lines that were not covered).

Unfortunately, if we could somehow run a coverage report on this section of the chapter, we'd find that we have not covered most of what there is to know about code coverage! It is possible to use the coverage API to manage code coverage from within our own programs (or test suites), and `coverage.py` accepts numerous configuration options that we haven't touched on. We also haven't discussed the difference between statement coverage and branch coverage (the latter is much more useful, and the default in recent versions of `coverage.py`) or other styles of code coverage.

Bear in mind that while 100 percent code coverage is a lofty goal that we should all strive for, 100 percent coverage is not enough! Just because a statement was tested does not mean that it was tested properly for all possible inputs.

Case study

Let's walk through test-driven development by writing a small, tested, cryptography application. Don't worry, you won't need to understand the mathematics behind complicated modern encryption algorithms such as Threefish or RSA. Instead, we'll be implementing a sixteenth-century algorithm known as the Vigenère cipher. The application simply needs to be able to encode and decode a message, given an encoding keyword, using this cipher.

First, we need to understand how the cipher works if we apply it manually (without a computer). We start with a table like this:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
E F G H I J K L M N O P Q R S T U V W X Y Z A B C D
F G H I J K L M N O P Q R S T U V W X Y Z A B C D E
G H I J K L M N O P Q R S T U V W X Y Z A B C D E F
H I J K L M N O P Q R S T U V W X Y Z A B C D E F G
I J K L M N O P Q R S T U V W X Y Z A B C D E F G H
J K L M N O P Q R S T U V W X Y Z A B C D E F G H I
K L M N O P Q R S T U V W X Y Z A B C D E F G H I J
L M N O P Q R S T U V W X Y Z A B C D E F G H I J K
M N O P Q R S T U V W X Y Z A B C D E F G H I J K L
N O P Q R S T U V W X Y Z A B C D E F G H I J K L M
O P Q R S T U V W X Y Z A B C D E F G H I J K L M N
P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
Q R S T U V W X Y Z A B C D E F G H I J K L M N O P
R S T U V W X Y Z A B C D E F G H I J K L M N O P Q
S T U V W X Y Z A B C D E F G H I J K L M N O P Q R
T U V W X Y Z A B C D E F G H I J K L M N O P Q R S
U V W X Y Z A B C D E F G H I J K L M N O P Q R S T
V W X Y Z A B C D E F G H I J K L M N O P Q R S T U
W X Y Z A B C D E F G H I J K L M N O P Q R S T U V
X Y Z A B C D E F G H I J K L M N O P Q R S T U V W
Y Z A B C D E F G H I J K L M N O P Q R S T U V W X
Z A B C D E F G H I J K L M N O P Q R S T U V W X Y
```

Given a keyword, TRAIN, we can encode the message ENCODED IN PYTHON as follows:

1. Repeat the keyword and message together such that it is easy to map letters from one to the other:

   ```
   E N C O D E D I N P Y T H O N   T R A I N T R A I N T R A I N
   ```

2. For each letter in the plain text, find the row that begins with that letter in the table.
3. Find the column with the letter associated with the keyword letter for the chosen plaintext letter.
4. The encoded character is at the intersection of this row and column.

For example, the row starting with E intersects the column starting with T at the character X. So, the first letter in the ciphertext is X. The row starting with N intersects the column starting with R at the character E, leading to the ciphertext XE. C intersects A at C, and O intersects I at W. D and N map to Q while E and T map to X. The full encoded message is XECWQXUIVCRKHWA.

Decoding basically follows the opposite procedure. First, find the row with the character for the shared keyword (the T row), then find the location in that row where the encoded character (the X) is located. The plaintext character is at the top of the column for that row (the E).

Implementing it

Our program will need an `encode` method that takes a keyword and plaintext and returns the ciphertext, and a `decode` method that accepts a keyword and ciphertext and returns the original message.

But rather than just writing those methods, let's follow a test-driven development strategy. We'll be using `py.test` for our unit testing. We need an `encode` method, and we know what it has to do; let's write a test for that method first:

```
def test_encode():
    cipher = VigenereCipher("TRAIN")
    encoded = cipher.encode("ENCODEDINPYTHON")
    assert encoded == "XECWQXUIVCRKHWA"
```

This test fails, naturally, because we aren't importing a `VigenereCipher` class anywhere. Let's create a new module to hold that class.

Let's start with the following `VigenereCipher` class:

```
class VigenereCipher:
    def __init__(self, keyword):
        self.keyword = keyword

    def encode(self, plaintext):
        return "XECWQXUIVCRKHWA"
```

If we add a `from vigenere_cipher import VigenereCipher` line to the top of our test class and run `py.test`, the preceding test will pass! We've finished our first test-driven development cycle.

Obviously, returning a hardcoded string is not the most sensible implementation of a cipher class, so let's add a second test:

```
def test_encode_character():
    cipher = VigenereCipher("TRAIN")
    encoded = cipher.encode("E")
    assert encoded == "X"
```

Ah, now that test will fail. It looks like we're going to have to work harder. But I just thought of something: what if someone tries to encode a string with spaces or lowercase characters? Before we start implementing the encoding, let's add some tests for these cases, so we don't we forget them. The expected behavior will be to remove spaces, and to convert lowercase letters to capitals:

```
def test_encode_spaces():
    cipher = VigenereCipher("TRAIN")
    encoded = cipher.encode("ENCODED IN PYTHON")
    assert encoded == "XECWQXUIVCRKHWA"

def test_encode_lowercase():
    cipher = VigenereCipher("TRain")
    encoded = cipher.encode("encoded in Python")
    assert encoded == "XECWQXUIVCRKHWA"
```

If we run the new test suite, we find that the new tests pass (they expect the same hardcoded string). But they ought to fail later if we forget to account for these cases.

Now that we have some test cases, let's think about how to implement our encoding algorithm. Writing code to use a table like we used in the earlier manual algorithm is possible, but seems complicated, considering that each row is just an alphabet rotated by an offset number of characters. It turns out (I asked Wikipedia) that we can use modulo arithmetic to combine the characters instead of doing a table lookup. Given plaintext and keyword characters, if we convert the two letters to their numerical values (with A being 0 and Z being 25), add them together, and take the remainder mod 26, we get the ciphertext character! This is a straightforward calculation, but since it happens on a character-by-character basis, we should probably put it in its own function. And before we do that, we should write a test for the new function:

```
from vigenere_cipher import combine_character
def test_combine_character():
    assert combine_character("E", "T") == "X"
    assert combine_character("N", "R") == "E"
```

Now we can write the code to make this function work. In all honesty, I had to run the test several times before I got this function completely correct; first I returned an integer, and then I forgot to shift the character back up to the normal ASCII scale from the zero-based scale. Having the test available made it easy to test and debug these errors. This is another bonus of test-driven development.

```
def combine_character(plain, keyword):
    plain = plain.upper()
    keyword = keyword.upper()
    plain_num = ord(plain) - ord('A')
    keyword_num = ord(keyword) - ord('A')
    return chr(ord('A') + (plain_num + keyword_num) % 26)
```

Now that `combine_characters` is tested, I thought we'd be ready to implement our `encode` function. However, the first thing we want inside that function is a repeating version of the keyword string that is as long as the plaintext. Let's implement a function for that first. Oops, I mean let's implement the test first!

```
def test_extend_keyword():
    cipher = VigenereCipher("TRAIN")
    extended = cipher.extend_keyword(16)
    assert extended == "TRAINTRAINTRAINT"
```

Before writing this test, I expected to write `extend_keyword` as a standalone function that accepted a keyword and an integer. But as I started drafting the test, I realized it made more sense to use it as a helper method on the `VigenereCipher` class. This shows how test-driven development can help design more sensible APIs. Here's the method implementation:

```
def extend_keyword(self, number):
    repeats = number // len(self.keyword) + 1
    return (self.keyword * repeats)[:number]
```

Once again, this took a few runs of the test to get right. I ended up adding a second versions of the test, one with fifteen and one with sixteen letters, to make sure it works if the integer division has an even number.

Now we're finally ready to write our `encode` method:

```
def encode(self, plaintext):
    cipher = []
    keyword = self.extend_keyword(len(plaintext))
    for p,k in zip(plaintext, keyword):
        cipher.append(combine_character(p,k))
    return "".join(cipher)
```

That looks correct. Our test suite should pass now, right?

Actually, if we run it, we'll find that two tests are still failing. We totally forgot about the spaces and lowercase characters! It is a good thing we wrote those tests to remind us. We'll have to add this line at the beginning of the method:

```
plaintext = plaintext.replace(" ", "").upper()
```

If we have an idea about a corner case in the middle of implementing something, we can create a test describing that idea. We don't even have to implement the test; we can just run `assert False` to remind us to implement it later. The failing test will never let us forget the corner case and it can't be ignored like filing a task can. If it takes a while to get around to fixing the implementation, we can mark the test as an expected failure.

Now all the tests pass successfully. This chapter is pretty long, so we'll condense the examples for decoding. Here are a couple tests:

```
def test_separate_character():
    assert separate_character("X", "T") == "E"
    assert separate_character("E", "R") == "N"

def test_decode():
    cipher = VigenereCipher("TRAIN")
    decoded = cipher.decode("XECWQXUIVCRKHWA")
    assert decoded == "ENCODEDINPYTHON"
```

Here's the `separate_character` function:

```
def separate_character(cypher, keyword):
    cypher = cypher.upper()
    keyword = keyword.upper()
    cypher_num = ord(cypher) - ord('A')
    keyword_num = ord(keyword) - ord('A')
    return chr(ord('A') + (cypher_num - keyword_num) % 26)
```

And the `decode` method:

```
    def decode(self, ciphertext):
        plain = []
        keyword = self.extend_keyword(len(ciphertext))
        for p,k in zip(ciphertext, keyword):
            plain.append(separate_character(p,k))
        return "".join(plain)
```

These methods have a lot of similarity to those used for encoding. The great thing about having all these tests written and passing is that we can now go back and modify our code, knowing it is still safely passing the tests. For example, if we replace our existing `encode` and `decode` methods with these refactored methods, our tests still pass:

```
    def _code(self, text, combine_func):
        text = text.replace(" ", "").upper()
        combined = []
        keyword = self.extend_keyword(len(text))
        for p,k in zip(text, keyword):
            combined.append(combine_func(p,k))
        return "".join(combined)

    def encode(self, plaintext):
```

```
            return self._code(plaintext, combine_character)

        def decode(self, ciphertext):
            return self._code(ciphertext, separate_character)
```

This is the final benefit of test-driven development, and the most important. Once the tests are written, we can improve our code as much as we like and be confident that our changes didn't break anything we have been testing for. Furthermore, we know exactly when our refactor is finished: when the tests all pass.

Of course, our tests may not comprehensively test everything we need them to; maintenance or code refactoring can still cause undiagnosed bugs that don't show up in testing. Automated tests are not foolproof. If bugs do occur, however, it is still possible to follow a test-driven plan; step one is to write a test (or multiple tests) that duplicates or "proves" that the bug in question is occurring. This will, of course, fail. Then write the code to make the tests stop failing. If the tests were comprehensive, the bug will be fixed, and we will know if it ever happens again, as soon as we run the test suite.

Finally, we can try to determine how well our tests operate on this code. With the `py.test` coverage plugin installed, `py.test --coverage-report=report` tells us that our test suite has 100 percent code coverage. This is a great statistic, but we shouldn't get too cocky about it. Our code hasn't been tested when encoding messages that have numbers, and its behavior with such inputs is thus undefined.

Your Coding Challenge

Practice test-driven development. That is your first exercise. It's easier to do this if you're starting a new project, but if you have existing code you need to work on, you can start by writing tests for each new feature you implement. This can become frustrating as you become more enamored with automated tests. The old, untested code will start to feel rigid and tightly coupled, and will become uncomfortable to maintain; you'll start feeling like changes you make are breaking the code and you have no way of knowing, for lack of tests. But if you start small, adding tests will improve, the codebase improves over time.

So to get your feet wet with test-driven development, start a fresh project. Once you've started to appreciate the benefits (you will) and realize that the time spent writing tests is quickly regained in terms of more maintainable code, you'll want to start writing tests for existing code. This is when you should start doing it, not before. Writing tests for code that we "know" works is boring. It is hard to get interested in the project until you realize just how broken the code we thought was working really is.

Ankita Thakur

Your Course Guide

Try writing the same set of tests using both the built-in unittest module and py.test. Which do you prefer? unittest is more similar to test frameworks in other languages, while py.test is arguably more Pythonic. Both allow us to write object-oriented tests and to test object-oriented programs with ease.

We used py.test in our case study, but we didn't touch on any features that wouldn't have been easily testable using unittest. Try adapting the tests to use test skipping or funcargs. Try the various setup and teardown methods, and compare their use to funcargs. Which feels more natural to you?

In our case study, we have a lot of tests that use a similar VigenereCipher object; try reworking this code to use a funcarg. How many lines of code does it save?

Try running a coverage report on the tests you've written. Did you miss testing any lines of code? Even if you have 100 percent coverage, have you tested all the possible inputs? If you're doing test-driven development, 100 percent coverage should follow quite

Ankita Thakur

Your Course Guide

naturally, as you will write a test before the code that satisfies that test. However, if writing tests for existing code, it is more likely that there will be edge conditions that go untested.

Think carefully about the values that are somehow different: empty lists when you expect full ones, zero or one or infinity compared to intermediate integers, floats that don't round to an exact decimal place, strings when you expected numerals, or the ubiquitous None value when you expected something meaningful. If your tests cover such edge cases, your code will be in good shape.

Summary of Module 1 Chapter 13

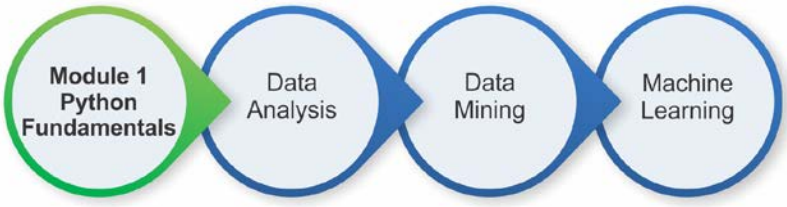
Ankita Thakur

Your Course Guide

We have finally covered the most important topic in Python programming: automated testing. Test-driven development is considered a best practice. The standard library unittest module provides a great out-of-the-box solution for testing, while the py.test framework has some more Pythonic syntaxes. Mocks can be used to emulate complex classes in our tests. Code coverage gives us an estimate of how much of our code is being run by our tests, but it does not tell us that we have tested the right things.

In the next chapter, we'll jump into a completely different topic: concurrency.

Your Progress through the Course So Far



14
Concurrency

Concurrency is the art of making a computer do (or appear to do) multiple things at once. Historically, this meant inviting the processor to switch between different tasks many times per second. In modern systems, it can also literally mean doing two or more things simultaneously on separate processor cores.

Concurrency is not inherently an object-oriented topic, but Python's concurrent systems are built on top of the object-oriented constructs we've covered throughout the module. This chapter will introduce you to the following topics:

- Threads
- Multiprocessing
- Futures
- AsyncIO

Concurrency is complicated. The basic concepts are fairly simple, but the bugs that can occur are notoriously difficult to track down. However, for many projects, concurrency is the only way to get the performance we need. Imagine if a web server couldn't respond to a user's request until the previous one was completed! We won't be going into all the details of just how hard it is (another full module would be required) but we'll see how to do basic concurrency in Python, and some of the most common pitfalls to avoid.

Threads

Most often, concurrency is created so that work can continue happening while the program is waiting for I/O to happen. For example, a server can start processing a new network request while it waits for data from a previous request to arrive. An interactive program might render an animation or perform a calculation while waiting for the user to press a key. Bear in mind that while a person can type more than 500 characters per minute, a computer can perform billions of instructions per second. Thus, a ton of processing can happen between individual key presses, even when typing quickly.

It's theoretically possible to manage all this switching between activities within your program, but it would be virtually impossible to get right. Instead, we can rely on Python and the operating system to take care of the tricky switching part, while we create objects that appear to be running independently, but simultaneously. These objects are called **threads**; in Python they have a very simple API. Let's take a look at a basic example:

```
from threading import Thread

class InputReader(Thread):
    def run(self):
        self.line_of_text = input()

print("Enter some text and press enter: ")
thread = InputReader()
thread.start()

count = result = 1
while thread.is_alive():
    result = count * count
    count += 1

print("calculated squares up to {0} * {0} = {1}".format(
    count, result))
print("while you typed '{}'".format(thread.line_of_text))
```

This example runs two threads. Can you see them? Every program has one thread, called the main thread. The code that executes from the beginning is happening in this thread. The second thread, more obviously, exists as the `InputReader` class.

To construct a thread, we must extend the `Thread` class and implement the `run` method. Any code inside the `run` method (or that is called from within that method) is executed in a separate thread.

The new thread doesn't start running until we call the `start()` method on the object. In this case, the thread immediately pauses to wait for input from the keyboard. In the meantime, the original thread continues executing at the point `start` was called. It starts calculating squares inside a `while` loop. The condition in the `while` loop checks if the `InputReader` thread has exited its `run` method yet; once it does, it outputs some summary information to the screen.

If we run the example and type the string "hello world", the output looks as follows:

```
Enter some text and press enter:
hello world
calculated squares up to 1044477 * 1044477 = 1090930114576
while you typed 'hello world'
```

You will, of course, calculate more or less squares while typing the string as the numbers are related to both our relative typing speeds, and to the processor speeds of the computers we are running.

A thread only starts running in concurrent mode when we call the `start` method. If we want to take out the concurrent call to see how it compares, we can call `thread.run()` in the place that we originally called `thread.start()`. The output is telling:

```
Enter some text and press enter:
hello world
calculated squares up to 1 * 1 = 1
while you typed 'hello world'
```

In this case, the thread never becomes alive and the `while` loop never executes. We wasted a lot of CPU power sitting idle while we were typing.

There are a lot of different patterns for using threads effectively. We won't be covering all of them, but we will look at a common one so we can learn about the `join` method. Let's check the current temperature in the capital city of every province in Canada:

```
from threading import Thread
import json
from urllib.request import urlopen
import time

CITIES = [
    'Edmonton', 'Victoria', 'Winnipeg', 'Fredericton',
    "St. John's", 'Halifax', 'Toronto', 'Charlottetown',
```

```
        'Quebec City', 'Regina'
    ]

    class TempGetter(Thread):
        def __init__(self, city):
            super().__init__()
            self.city = city

        def run(self):
            url_template = (
                'http://api.openweathermap.org/data/2.5/'
                'weather?q={},CA&units=metric')
            response = urlopen(url_template.format(self.city))
            data = json.loads(response.read().decode())
            self.temperature = data['main']['temp']

    threads = [TempGetter(c) for c in CITIES]
    start = time.time()
    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()

    for thread in threads:
        print(
            "it is {0.temperature:.0f}°C in {0.city}".format(thread))
    print(
        "Got {} temps in {} seconds".format(
        len(threads), time.time() - start))
```

This code constructs 10 threads before starting them. Notice how we can override the constructor to pass them into the `Thread` object, remembering to call `super` to ensure the `Thread` is properly initialized. Pay attention to this: the new thread isn't running yet, so the `__init__` method is still executing from inside the main thread. Data we construct in one thread is accessible from other running threads.

After the 10 threads have been started, we loop over them again, calling the `join()` method on each. This method essentially says "wait for the thread to complete before doing anything". We call this ten times in sequence; the for loop won't exit until all ten threads have completed.

At this point, we can print the temperature that was stored on each thread object. Notice once again that we can access data that was constructed within the thread from the main thread. In threads, all state is shared by default.

Executing this code on my 100 mbit connection takes about two tenths of a second:

```
it is 5°C in Edmonton
it is 11°C in Victoria
it is 0°C in Winnipeg
it is -10°C in Fredericton
it is -12°C in St. John's
it is -8°C in Halifax
it is -6°C in Toronto
it is -13°C in Charlottetown
it is -12°C in Quebec City
it is 2°C in Regina
    Got 10 temps in 0.18970298767089844 seconds
```

If we run this code in a single thread (by changing the `start()` call to `run()` and commenting out the `join()` call), it takes closer to 2 seconds because each 0.2 second request has to complete before the next one begins. This speedup of 10 times shows just how useful concurrent programming can be.

The many problems with threads

Threads can be useful, especially in other programming languages, but modern Python programmers tend to avoid them for several reasons. As we'll see, there are other ways to do concurrent programming that are receiving more attention from the Python developers. Let's discuss some of these pitfalls before moving on to more salient topics.

Shared memory

The main problem with threads is also their primary advantage. Threads have access to all the memory and thus all the variables in the program. This can too easily cause inconsistencies in the program state. Have you ever encountered a room where a single light has two switches and two different people turn them on at the same time? Each person (thread) expects their action to turn the lamp (a variable) on, but the resulting value (the lamp is off) is inconsistent with those expectations. Now imagine if those two threads were transferring funds between bank accounts or managing the cruise control in a vehicle.

The solution to this problem in threaded programming is to "synchronize" access to any code that reads or writes a shared variable. There are a few different ways to do this, but we won't go into them here so we can focus on more Pythonic constructs. The synchronization solution works, but it is way too easy to forget to apply it. Worse, bugs due to inappropriate use of synchronization are really hard to track down because the order in which threads perform operations is inconsistent. We can't easily reproduce the error. Usually, it is safest to force communication between threads to happen using a lightweight data structure that already uses locks appropriately. Python offers the `queue.Queue` class to do this; it's functionality is basically the same as the `multiprocessing.Queue` that we will discuss in the next section.

In some cases, these disadvantages might be outweighed by the one advantage of allowing shared memory: it's fast. If multiple threads need access to a huge data structure, shared memory can provide that access quickly. However, this advantage is usually nullified by the fact that, in Python, it is impossible for two threads running on different CPU cores to be performing calculations at exactly the same time. This brings us to our second problem with threads.

The global interpreter lock

In order to efficiently manage memory, garbage collection, and calls to machine code in libraries, Python has a utility called the **global interpreter lock**, or **GIL**. It's impossible to turn off, and it means that threads are useless in Python for one thing that they excel at in other languages: parallel processing. The GIL's primary effect, for our purposes is to prevent any two threads from doing work at the exact same time, even if they have work to do. In this case, "doing work" means using the CPU, so it's perfectly ok for multiple threads to access the disk or network; the GIL is released as soon as the thread starts to wait for something.

The GIL is quite highly disparaged, mostly by people who don't understand what it is or all the benefits it brings to Python. It would definitely be nice if our language didn't have this restriction, but the Python reference developers have determined that, for now at least, it brings more value than it costs. It makes the reference implementation easier to maintain and develop, and during the single-core processor days when Python was originally developed, it actually made the interpreter faster. The net result of the GIL, however, is that it limits the benefits that threads bring us, without alleviating the costs.

While the GIL is a problem in the reference implementation of Python that most people use, it has been solved in some of the nonstandard implementations such as IronPython and Jython. Unfortunately, at the time of publication, none of these support Python 3.

Thread overhead

One final limitation of threads as compared to the asynchronous system we will be discussing later is the cost of maintaining the thread. Each thread takes up a certain amount of memory (both in the Python process and the operating system kernel) to record the state of that thread. Switching between the threads also uses a (small) amount of CPU time. This work happens seamlessly without any extra coding (we just have to call `start()` and the rest is taken care of), but the work still has to happen somewhere.

This can be alleviated somewhat by structuring our workload so that threads can be reused to perform multiple jobs. Python provides a `ThreadPool` feature to handle this. It is shipped as part of the multiprocessing library and behaves identically to the `ProcessPool`, that we will discuss shortly, so let's defer discussion until the next section.

Multiprocessing

The multiprocessing API was originally designed to mimic the thread API. However, it has evolved and in recent versions of Python 3, it supports more features more robustly. The multiprocessing library is designed when CPU-intensive jobs need to happen in parallel and multiple cores are available (given that a four core Raspberry Pi can currently be purchased for $35, there are usually multiple cores available). Multiprocessing is not useful when the processes spend a majority of their time waiting on I/O (for example, network, disk, database, or keyboard), but they are the way to go for parallel computation.

The multiprocessing module spins up new operating system processes to do the work. On Windows machines, this is a relatively expensive operation; on Linux, processes are implemented in the kernel the same way threads are, so the overhead is limited to the cost of running separate Python interpreters in each process.

Let's try to parallelize a compute-heavy operation using similar constructs to those provided by the `threading` API:

```
from multiprocessing import Process, cpu_count
import time
import os

class MuchCPU(Process):
    def run(self):
        print(os.getpid())
        for i in range(200000000):
            pass

if __name__ == '__main__':
    procs =  [MuchCPU() for f in range(cpu_count())]
    t = time.time()
    for p in procs:
        p.start()
    for p in procs:
        p.join()
    print('work took {} seconds'.format(time.time() - t))
```

This example just ties up the CPU for 200 million iterations. You may not consider this to be useful work, but it's a cold day and I appreciate the heat my laptop generates under such load.

The API should be familiar; we implement a subclass of `Process` (instead of `Thread`) and implement a `run` method. This method prints out the process ID (a unique number the operating system assigns to each process on the machine) before doing some intense (if misguided) work.

Pay special attention to the `if __name__ == '__main__':` guard around the module level code that prevents it to run if the module is being imported, rather than run as a program. This is good practice in general, but when using multiprocessing on some operating systems, it is essential. Behind the scenes, multiprocessing may have to import the module inside the new process in order to execute the `run()` method. If we allowed the entire module to execute at that point, it would start creating new processes recursively until the operating system ran out of resources.

We construct one process for each processor core on our machine, then start and join each of those processes. On my 2014 era quad-core laptop, the output looks like this:

```
6987
6988
```

```
6989
6990
work took 12.96659541130066 seconds
```

The first four lines are the process ID that was printed inside each `MuchCPU` instance. The last line shows that the 200 million iterations can run in about 13 seconds on my machine. During that 13 seconds, my process monitor indicated that all four of my cores were running at 100 percent.

If we subclass `threading.Thread` instead of `multiprocessing.Process` in `MuchCPU`, the output looks like this:

```
7235
7235
7235
7235
work took 28.577413082122803 seconds
```

This time, the four threads are running inside the same process and take close to three times as long to run. This is the cost of the global interpreter lock; in other languages or implementations of Python, the threaded version would run at least as fast as the multiprocessing version, We might expect it to be four times as long, but remember that many other programs are running on my laptop. In the multiprocessing version, these programs also need a share of the four CPUs. In the threading version, those programs can use the other three CPUs instead.

Multiprocessing pools

In general, there is no reason to have more processes than there are processors on the computer. There are a few reasons for this:

- Only `cpu_count()` processes can run simultaneously
- Each process consumes resources with a full copy of the Python interpreter
- Communication between processes is expensive
- Creating processes takes a nonzero amount of time

Given these constraints, it makes sense to create at most `cpu_count()` processes when the program starts and then have them execute tasks as needed. It is not difficult to implement a basic series of communicating processes that does this, but it can be tricky to debug, test, and get right. Of course, Python being Python, we don't have to do all this work because the Python developers have already done it for us in the form of multiprocessing pools.

The primary advantage of pools is that they abstract away the overhead of figuring out what code is executing in the main process and which code is running in the subprocess. As with the threading API that multiprocessing mimics, it can often be hard to remember who is executing what. The pool abstraction restricts the number of places that code in different processes interact with each other, making it much easier to keep track of.

- Pools also seamlessly hide the process of passing data between processes. Using a pool looks much like a function call; you pass data into a function, it is executed in another process or processes, and when the work is done, a value is returned. It is important to understand that under the hood, a lot of work is being done to support this: objects in one process are being pickled and passed into a pipe.
- Another process retrieves data from the pipe and unpickles it. Work is done in the subprocess and a result is produced. The result is pickled and passed into a pipe. Eventually, the original process unpickles it and returns it.

All this pickling and passing data into pipes takes time and memory. Therefore, it is ideal to keep the amount and size of data passed into and returned from the pool to a minimum, and it is only advantageous to use the pool if a lot of processing has to be done on the data in question.

Armed with this knowledge, the code to make all this machinery work is surprisingly simple. Let's look at the problem of calculating all the prime factors of a list of random numbers. This is a common and expensive part of a variety of cryptography algorithms (not to mention attacks on those algorithms!). It requires years of processing power to crack the extremely large numbers used to secure your bank accounts. The following implementation, while readable, is not at all efficient, but that's ok because we want to see it using lots of CPU time:

```
import random
from multiprocessing.pool import Pool

def prime_factor(value):
    factors = []
    for divisor in range(2, value-1):
        quotient, remainder = divmod(value, divisor)
        if not remainder:
            factors.extend(prime_factor(divisor))
            factors.extend(prime_factor(quotient))
            break
```

```
        else:
            factors = [value]
        return factors

    if __name__ == '__main__':
        pool = Pool()

        to_factor = [
            random.randint(100000, 50000000) for i in range(20)
        ]
        results = pool.map(prime_factor, to_factor)
        for value, factors in zip(to_factor, results):
            print("The factors of {} are {}".format(value, factors))
```

Let's focus on the parallel processing aspects as the brute force recursive algorithm for calculating factors is pretty clear. We first construct a multiprocessing pool instance. By default, this pool creates a separate process for each of the CPU cores in the machine it is running on.

The `map` method accepts a function and an iterable. The pool pickles each of the values in the iterable and passes it into an available process, which executes the function on it. When that process is finished doing it's work, it pickles the resulting list of factors and passes it back to the pool. Once all the pools are finished processing work (which could take some time), the results list is passed back to the original process, which has been waiting patiently for all this work to complete.

It is often more useful to use the similar `map_async` method, which returns immediately even though the processes are still working. In that case, the results variable would not be a list of values, but a promise to return a list of values later by calling `results.get()`. This promise object also has methods like `ready()`, and `wait()`, which allow us to check whether all the results are in yet.

Alternatively, if we don't know all the values we want to get results for in advance, we can use the `apply_async` method to queue up a single job. If the pool has a process that isn't already working, it will start immediately; otherwise, it will hold onto the task until there is a free process available.

Pools can also be `close`d, which refuses to take any further tasks, but processes everything currently in the queue, or `terminate`d, which goes one step further and refuses to start any jobs still on the queue, although any jobs currently running are still permitted to complete.

Queues

If we need more control over communication between processes, we can use a `Queue`. `Queue` data structures are useful for sending messages from one process into one or more other processes. Any picklable object can be sent into a `Queue`, but remember that pickling can be a costly operation, so keep such objects small. To illustrate queues, let's build a little search engine for text content that stores all relevant entries in memory.

This is not the most sensible way to build a text-based search engine, but I have used this pattern to query numerical data that needed to use CPU-intensive processes to construct a chart that was then rendered to the user.

This particular search engine scans all files in the current directory in parallel. A process is constructed for each core on the CPU. Each of these is instructed to load some of the files into memory. Let's look at the function that does the loading and searching:

```
def search(paths, query_q, results_q):
    lines = []
    for path in paths:
        lines.extend(l.strip() for l in path.open())

    query = query_q.get()
    while query:
        results_q.put([l for l in lines if query in l])
        query = query_q.get()
```

Remember, this function is run in a different process (in fact, it is run in `cpucount()` different processes) from the main thread. It is passes a list of `path.path` objects and two `multiprocessing.Queue` objects; one for incoming queries and one to send outgoing results. These queues have a similar interface to the `Queue` class we discussed in *Chapter 6, Python Data Structures*. However, they are doing extra work to pickle the data in the queue and pass it into the subprocess over a pipe. These two queues are set up in the main process and passed through the pipes into the search function inside the child processes.

The search code is pretty dumb, both in terms of efficiency and of capabilities; it loops over every line stored in memory and puts the matching ones in a list. The list is placed on a queue and passed back to the main process.

Let's look at the main process, which sets up these queues:

```
if __name__ == '__main__':
    from multiprocessing import Process, Queue, cpu_count
    from path import path
    cpus = cpu_count()
```

```
pathnames = [f for f in path('.').listdir() if f.isfile()]
paths = [pathnames[i::cpus] for i in range(cpus)]
query_queues = [Queue() for p in range(cpus)]
results_queue = Queue()

search_procs = [
    Process(target=search, args=(p, q, results_queue))
    for p, q in zip(paths, query_queues)
]
for proc in search_procs: proc.start()
```

For easier description, let's assume `cpu_count` is four. Notice how the import statements are placed inside the `if` guard? This is a small optimization that prevents them from being imported in each subprocess (where they aren't needed) on certain operating systems. We list all the paths in the current directory and then split the list into four approximately equal parts. We also construct a list of four `Queue` objects to send data into each subprocess. Finally, we construct a `single` results queue; this is passed into all four of the subprocesses. Each of them can put data into the queue and it will be aggregated in the main process.

Now let's look at the code that makes a search actually happen:

```
for q in query_queues:
    q.put("def")
    q.put(None)  # Signal process termination

for i in range(cpus):
    for match in results_queue.get():
        print(match)
for proc in search_procs: proc.join()
```

This code performs a single search for `"def"` (because it's a common phrase in a directory full of Python files!). In a more production ready system, we would probably hook a socket up to this search code. In that case, we'd have to change the inter-process protocol so that the message coming back on the return queue contained enough information to identify which of many queries the results were attached to.

This use of queues is actually a local version of what could become a distributed system. Imagine if the searches were being sent out to multiple computers and then recombined. We won't discuss it here, but the multiprocessing module includes a manager class that can take a lot of the boilerplate out of the preceding code. There is even a version of the `multiprocessing.Manager` that can manage subprocesses on remote systems to construct a rudimentary distributed application. Check the Python multiprocessing documentation if you are interested in pursuing this further.

The problems with multiprocessing

As threads do, multiprocessing also has problems, some of which we have already discussed. There is no best way to do concurrency; this is especially true in Python. We always need to examine the parallel problem to figure out which of the many available solutions is the best one for that problem. Sometimes, there is no best solution.

In the case of multiprocessing, the primary drawback is that sharing data between processes is very costly. As we have discussed, all communication between processes, whether by queues, pipes, or a more implicit mechanism requires pickling the objects. Excessive pickling quickly dominates processing time. Multiprocessing works best when relatively small objects are passed between processes and a tremendous amount of work needs to be done on each one. On the other hand, if no communication between processes is required, there may not be any point in using the module at all; we can spin up four separate Python processes and use them independently.

The other major problem with multiprocessing is that, like threads, it can be hard to tell which process a variable or method is being accessed in. In multiprocessing, if you access a variable from another process it will usually overwrite the variable in the currently running process while the other process keeps the old value. This is really confusing to maintain, so don't do it.

Futures

Let's start looking at a more asynchronous way of doing concurrency. Futures wrap either multiprocessing or threading depending on what kind of concurrency we need (tending towards I/O versus tending towards CPU). They don't completely solve the problem of accidentally altering shared state, but they allow us to structure our code such that it is easier to track down when we do so. Futures provide distinct boundaries between the different threads or processes. Similar to the multiprocessing pool, they are useful for "call and answer" type interactions in which processing can happen in another thread and then at some point in the future (they are aptly named, after all), you can ask it for the result. It's really just a wrapper around multiprocessing pools and thread pools, but it provides a cleaner API and encourages nicer code.

A future is an object that basically wraps a function call. That function call is run in the background in a thread or process. The future object has methods to check if the future has completed and to get the results after it has completed.

Let's do another file search example. In the last section, we implemented a version of the `unix grep` command. This time, let's do a simple version of the `find` command. The example will search the entire filesystem for paths that contain a given string of characters:

```
from concurrent.futures import ThreadPoolExecutor
from pathlib import Path
from os.path import sep as pathsep
from collections import deque

def find_files(path, query_string):
    subdirs = []
    for p in path.iterdir():
        full_path = str(p.absolute())
        if p.is_dir() and not p.is_symlink():
            subdirs.append(p)
        if query_string in full_path:
                print(full_path)

    return subdirs

query = '.py'
futures = deque()
basedir = Path(pathsep).absolute()

with ThreadPoolExecutor(max_workers=10) as executor:
    futures.append(
        executor.submit(find_files, basedir, query))
    while futures:
        future = futures.popleft()
        if future.exception():
            continue
        elif future.done():
            subdirs = future.result()
            for subdir in subdirs:
                futures.append(executor.submit(
                    find_files, subdir, query))
        else:
            futures.append(future)
```

This code consists of a function named `find_files` that is run in a separate thread (or process, if we used `ProcessPoolExecutor`). There isn't anything particularly special about this function, but note how it does not access any global variables. All interaction with the external environment is passed into the function or returned from it. This is not a technical requirement, but it is the best way to keep your brain inside your skull when programming with futures.

Accessing outside variables without proper synchronization results in something called a **race** condition. For example, imagine two concurrent writes trying to increment an integer counter. They start at the same time and both read the value as 5. Then they both increment the value and write back the result as 6. But if two processes are trying to increment a variable, the expected result would be that it gets incremented by two, so the result should be 7. Modern wisdom is that the easiest way to avoid doing this is to keep as much state as possible private and share them through known-safe constructs, such as queues.

We set up a couple variables before we get started; we'll be searching for all files that contain the characters `'.py'` for this example. We have a queue of futures that we'll discuss shortly. The `basedir` variable points to the root of the filesystem; `'/'` on Unix machines and probably `C:\` on Windows.

First, let's have a short course on search theory. This algorithm implements breadth first search in parallel. Rather than recursively searching every directory using a depth first search, it adds all the subdirectories in the current folder to the queue, then all the subdirectories of each of those folders and so on.

The meat of the program is known as an event loop. We can construct a `ThreadPoolExecutor` as a context manager so that it is automatically cleaned up and its threads closed when it is done. It requires a `max_workers` argument to indicate the number of threads running at a time; if more than this many jobs are submitted, it queues up the rest until a worker thread becomes available. When using `ProcessPoolExecutor`, this is normally constrained to the number of CPUs on the machine, but with threads, it can be much higher, depending how many are waiting on I/O at a time. Each thread takes up a certain amount of memory, so it shouldn't be too high; it doesn't take all that many threads before the speed of the disk, rather than number of parallel requests, is the bottleneck.

Once the executor has been constructed, we submit a job to it using the root directory. The `submit()` method immediately returns a `Future` object, which promises to give us a result eventually. The future is placed on the queue. The loop then repeatedly removes the first future from the queue and inspects it. If it is still running, it gets added back to the end of the queue. Otherwise, we check if the function raised an exception with a call to `future.exception()`. If it did, we just ignore it (it's usually a permission error, although a real app would need to be more careful about what the exception was). If we didn't check this exception here, it would be raised when we called `result()` and could be handled through the normal `try...except` mechanism.

Assuming no exception occurred, we can call `result()` to get the return value of the function call. Since the function returns a list of subdirectories that are not symbolic links (my lazy way of preventing an infinite loop), `result()` returns the same thing. These new subdirectories are submitted to the executor and the resulting futures are tossed onto the queue to have their contents searched in a later iteration.

So that's all that is required to develop a future-based I/O-bound application. Under the hood, it's using the same thread or process APIs we've already discussed, but it provides a more understandable interface and makes it easier to see the boundaries between concurrently running functions (just don't try to access global variables from inside the future!).

AsyncIO

AsyncIO is the current state of the art in Python concurrent programming. It combines the concept of futures and an event loop with the coroutines we discussed in *Chapter 9, The Iterator Pattern*. The result is about as elegant and easy to understand as it is possible to get when writing concurrent code, though that isn't saying a lot!

AsyncIO can be used for a few different concurrent tasks, but it was specifically designed for network I/O. Most networking applications, especially on the server side, spend a lot of time waiting for data to come in from the network. This can be solved by handling each client in a separate thread, but threads use up memory and other resources. AsyncIO uses coroutines instead of threads.

The library also provides its own event loop, obviating the need for the several lines long while loop in the previous example. However, event loops come with a cost. When we run code in an async task on the event loop, that code must return immediately, blocking neither on I/O nor on long-running calculations. This is a minor thing when writing our own code, but it means that any standard library or third-party functions that block on I/O have to have non-blocking versions created.

AsyncIO solves this by creating a set of coroutines that use the `yield from` syntax to return control to the event loop immediately. The event loop takes care of checking whether the blocking call has completed and performing any subsequent tasks, just like we did manually in the previous section.

AsyncIO in action

A canonical example of a blocking function is the `time.sleep` call. Let's use the asynchronous version of this call to illustrate the basics of an AsyncIO event loop:

```
import asyncio
import random

@asyncio.coroutine
def random_sleep(counter):
    delay = random.random() * 5
    print("{} sleeps for {:.2f} seconds".format(counter, delay))
    yield from asyncio.sleep(delay)
    print("{} awakens".format(counter))

@asyncio.coroutine
def five_sleepers():
    print("Creating five tasks")
    tasks = [
        asyncio.async(random_sleep(i)) for i in range(5)]
    print("Sleeping after starting five tasks")
    yield from asyncio.sleep(2)
    print("Waking and waiting for five tasks")
    yield from asyncio.wait(tasks)

asyncio.get_event_loop().run_until_complete(five_sleepers())
print("Done five tasks")
```

This is a fairly basic example, but it covers several features of AsyncIO programming. It is easiest to understand in the order that it executes, which is more or less bottom to top.

The second last line gets the event loop and instructs it to run a future until it is finished. The future in question is named `five_sleepers`. Once that future has done its work, the loop will exit and our code will terminate. As asynchronous programmers, we don't need to know too much about what happens inside that `run_until_complete` call, but be aware that a lot is going on. It's a souped up coroutine version of the futures loop we wrote in the previous chapter that knows how to deal with iteration, exceptions, function returns, parallel calls, and more.

Now look a little more closely at that `five_sleepers` future. Ignore the decorator for a few paragraphs; we'll get back to it. The coroutine first constructs five instances of the `random_sleep` future. The resulting futures are wrapped in an `asyncio.async` task, which adds them to the loop's task queue so they can execute concurrently when control is returned to the event loop.

That control is returned whenever we call `yield from`. In this case, we call `yield from asyncio.sleep` to pause execution of this coroutine for two seconds. During this break, the event loop executes the tasks that it has queued up; namely the five `random_sleep` futures. These coroutines each print a starting message, then send control back to the event loop for a specific amount of time. If any of the sleep calls inside `random_sleep` are shorter than two seconds, the event loop passes control back into the relevant future, which prints its awakening message before returning. When the sleep call inside `five_sleepers` wakes up, it executes up to the next yield from call, which waits for the remaining `random_sleep` tasks to complete. When all the sleep calls have finished executing, the `random_sleep` tasks return, which removes them from the event queue. Once all five of those are completed, the `asyncio.wait` call and then the `five_sleepers` method also return. Finally, since the event queue is now empty, the `run_until_complete` call is able to terminate and the program ends.

The `asyncio.coroutine` decorator mostly just documents that this coroutine is meant to be used as a future in an event loop. In this case, the program would run just fine without the decorator. However, the `asyncio.coroutine` decorator can also be used to wrap a normal function (one that doesn't yield) so that it can be treated as a future. In this case, the entire function executes before returning control to the event loop; the decorator just forces the function to fulfill the coroutine API so the event loop knows how to handle it.

Reading an AsyncIO future

An AsyncIO coroutine executes each line in order until it encounters a `yield from` statement, at which point it returns control to the event loop. The event loop then executes any other tasks that are ready to run, including the one that the original coroutine was waiting on. Whenever that child task completes, the event loop sends the result back into the coroutine so that it can pick up executing until it encounters another `yield from` statement or returns.

This allows us to write code that executes synchronously until we explicitly need to wait for something. This removes the nondeterministic behavior of threads, so we don't need to worry nearly so much about shared state.

It's still a good idea to avoid accessing shared state from inside a coroutine. It makes your code much easier to reason about. More importantly, even though an ideal world might have all asynchronous execution happen inside coroutines, the reality is that some futures are executed behind the scenes inside threads or processes. Stick to a "share nothing" philosophy to avoid a ton of difficult bugs.

In addition, AsyncIO allows us to collect logical sections of code together inside a single coroutine, even if we are waiting for other work elsewhere. As a specific instance, even though the `yield from asyncio.sleep` call in the `random_sleep` coroutine is allowing a ton of stuff to happen inside the event loop, the coroutine itself looks like it's doing everything in order. This ability to read related pieces of asynchronous code without worrying about the machinery that waits for tasks to complete is the primary benefit of the AsyncIO module.

AsyncIO for networking

AsyncIO was specifically designed for use with network sockets, so let's implement a DNS server. More accurately, let's implement one extremely basic feature of a DNS server.

The domain name system's basic purpose is to translate domain names, such as www.amazon.com into IP addresses such as 72.21.206.6. It has to be able to perform many types of queries and know how to contact other DNS servers if it doesn't have the answer required. We won't be implementing any of this, but the following example is able to respond directly to a standard DNS query to look up IPs for my three most recent employers:

```
import asyncio
from contextlib import suppress

ip_map = {
    b'facebook.com.': '173.252.120.6',
    b'yougov.com.': '213.52.133.246',
    b'wipo.int.': '193.5.93.80'
}

def lookup_dns(data):
    domain = b''
    pointer, part_length = 13, data[12]
    while part_length:
```

```
            domain += data[pointer:pointer+part_length] + b'.'
            pointer += part_length + 1
            part_length = data[pointer - 1]

        ip = ip_map.get(domain, '127.0.0.1')

        return domain, ip

    def create_response(data, ip):
        ba = bytearray
        packet = ba(data[:2]) + ba([129, 128]) + data[4:6] * 2
        packet += ba(4) + data[12:]
        packet += ba([192, 12, 0, 1, 0, 1, 0, 0, 0, 60, 0, 4])
        for x in ip.split('.'): packet.append(int(x))
        return packet

    class DNSProtocol(asyncio.DatagramProtocol):
        def connection_made(self, transport):
            self.transport = transport

        def datagram_received(self, data, addr):
            print("Received request from {}".format(addr[0]))
            domain, ip = lookup_dns(data)
            print("Sending IP {} for {} to {}".format(
                domain.decode(), ip, addr[0]))
            self.transport.sendto(
                create_response(data, ip), addr)

    loop = asyncio.get_event_loop()
    transport, protocol = loop.run_until_complete(
        loop.create_datagram_endpoint(
            DNSProtocol, local_addr=('127.0.0.1', 4343)))
    print("DNS Server running")

    with suppress(KeyboardInterrupt):
        loop.run_forever()
    transport.close()
    loop.close()
```

This example sets up a dictionary that dumbly maps a few domains to IPv4 addresses. It is followed by two functions that extract information from a binary DNS query packet and construct the response. We won't be discussing these; if you want to know more about DNS read RFC ("request for comment", the format for defining most Internet protocols) 1034 and 1035.

You can test this service by running the following command in another terminal:

```
nslookup -port=4343 facebook.com localhost
```

Let's get on with the entrée. AsyncIO networking revolves around the intimately linked concepts of transports and protocols. A protocol is a class that has specific methods that are called when relevant events happen. Since DNS runs on top of **UDP** (**User Datagram Protocol**); we build our protocol class as a subclass of `DatagramProtocol`. This class has a variety of events that it can respond to; we are specifically interested in the initial connection occurring (solely so we can store the transport for future use) and the `datagram_received` event. For DNS, each received datagram must be parsed and responded to, at which point the interaction is over.

So, when a datagram is received, we process the packet, look up the IP, and construct a response using the functions we aren't talking about (they're black sheep in the family). Then we instruct the underlying transport to send the resulting packet back to the requesting client using its `sendto` method.

The transport essentially represents a communication stream. In this case, it abstracts away all the fuss of sending and receiving data on a UDP socket on an event loop. There are similar transports for interacting with TCP sockets and subprocesses, for example.

The UDP transport is constructed by calling the loop's `create_datagram_endpoint` coroutine. This constructs the appropriate UDP socket and starts listening on it. We pass it the address that the socket needs to listen on, and importantly, the protocol class we created so that the transport knows what to call when it receives data.

Since the process of initializing a socket takes a non-trivial amount of time and would block the event loop, the `create_datagram_endpoint` function is a coroutine. In our example, we don't really need to do anything while we wait for this initialization, so we wrap the call in `loop.run_until_complete`. The event loop takes care of managing the future, and when it's complete, it returns a tuple of two values: the newly initialized transport and the protocol object that was constructed from the class we passed in.

Behind the scenes, the transport has set up a task on the event loop that is listening for incoming UDP connections. All we have to do, then, is start the event loop running with the call to `loop.run_forever()` so that task can process these packets. When the packets arrive, they are processed on the protocol and everything just works.

The only other major thing to pay attention to is that transports (and, indeed, event loops) are supposed to be closed when we are finished with them. In this case, the code runs just fine without the two calls to `close()`, but if we were constructing transports on the fly (or just doing proper error handling!), we'd need to be quite a bit more conscious of it.

You may have been dismayed to see how much boilerplate is required in setting up a protocol class and underlying transport. AsyncIO provides an abstraction on top of these two key concepts called streams. We'll see an example of streams in the TCP server in the next example.

Using executors to wrap blocking code

AsyncIO provides its own version of the futures library to allow us to run code in a separate thread or process when there isn't an appropriate non-blocking call to be made. This essentially allows us to combine threads and processes with the asynchronous model. One of the more useful applications of this feature is to get the best of both worlds when an application has bursts of I/O-bound and CPU-bound activity. The I/O-bound portions can happen in the event-loop while the CPU-intensive work can be spun off to a different process. To illustrate this, let's implement "sorting as a service" using AsyncIO:

```
import asyncio
import json
from concurrent.futures import ProcessPoolExecutor

def sort_in_process(data):
    nums = json.loads(data.decode())
    curr = 1
    while curr < len(nums):
        if nums[curr] >= nums[curr-1]:
            curr += 1
        else:
            nums[curr], nums[curr-1] = \
                nums[curr-1], nums[curr]
            if curr > 1:
```

```
                curr -= 1

    return json.dumps(nums).encode()

@asyncio.coroutine
def sort_request(reader, writer):
    print("Received connection")
    length = yield from reader.read(8)
    data = yield from reader.readexactly(
        int.from_bytes(length, 'big'))
    result = yield from asyncio.get_event_loop().run_in_executor(
        None, sort_in_process, data)
    print("Sorted list")
    writer.write(result)
    writer.close()
    print("Connection closed")

loop = asyncio.get_event_loop()
loop.set_default_executor(ProcessPoolExecutor())
server = loop.run_until_complete(
    asyncio.start_server(sort_request, '127.0.0.1', 2015))
print("Sort Service running")

loop.run_forever()
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

This is an example of good code implementing some really stupid ideas. The whole idea of sort as a service is pretty ridiculous. Using our own sorting algorithm instead of calling Python's `sorted` is even worse. The algorithm we used is called gnome sort, or in some cases, "stupid sort". It is a slow sort algorithm implemented in pure Python. We defined our own protocol instead of using one of the many perfectly suitable application protocols that exist in the wild. Even the idea of using multiprocessing for parallelism might be suspect here; we still end up passing all the data into and out of the subprocesses. Sometimes, it's important to take a step back from the program you are writing and ask yourself if you are trying to meet the right goals.

But let's look at some of the smart features of this design. First, we are passing bytes into and out of the subprocess. This is a lot smarter than decoding the JSON in the main process. It means the (relatively expensive) decoding can happen on a different CPU. Also, pickled JSON strings are generally smaller than pickled lists, so less data is passing between processes.

Second, the two methods are very linear; it looks like code is being executed one line after another. Of course, in AsyncIO, this is an illusion, but we don't have to worry about shared memory or concurrency primitives.

Streams

The previous example should look familiar by now as it has a similar boilerplate to other AsyncIO programs. However, there are a few differences. You'll notice we called `start_server` instead of `create_server`. This method hooks into AsyncIO's streams instead of using the underlying transport/protocol code. Instead of passing in a protocol class, we can pass in a normal coroutine, which receives reader and writer parameters. These both represent streams of bytes that can be read from and written like files or sockets. Second, because this is a TCP server instead of UDP, there is some socket cleanup required when the program finishes. This cleanup is a blocking call, so we have to run the `wait_closed` coroutine on the event loop.

Streams are fairly simple to understand. Reading is a potentially blocking call so we have to call it with `yield from`. Writing doesn't block; it just puts the data on a queue, which AsyncIO sends out in the background.

Our code inside the `sort_request` method makes two read requests. First, it reads 8 bytes from the wire and converts them to an integer using big endian notation. This integer represents the number of bytes of data the client intends to send. So in the next call, to `readexactly`, it reads that many bytes. The difference between `read` and `readexactly` is that the former will read up to the requested number of bytes, while the latter will buffer reads until it receives all of them, or until the connection closes.

Executors

Now let's look at the executor code. We import the exact same `ProcessPoolExecutor` that we used in the previous section. Notice that we don't need a special AsyncIO version of it. The event loop has a handy `run_in_executor` coroutine that we can use to run futures on. By default, the loop runs code in `ThreadPoolExecutor`, but we can pass in a different executor if we wish. Or, as we did in this example, we can set a different default when we set up the event loop by calling `loop.set_default_executor()`.

As you probably recall from the previous section, there is not a lot of boilerplate for using futures with an executor. However, when we use them with AsyncIO, there is none at all! The coroutine automatically wraps the function call in a future and submits it to the executor. Our code blocks until the future completes, while the event loop continues processing other connections, tasks, or futures. When the future is done, the coroutine wakes up and continues on to write the data back to the client.

You may be wondering if, instead of running multiple processes inside an event loop, it might be better to run multiple event loops in different processes. The answer is: "maybe". However, depending on the exact problem space, we are probably better off running independent copies of a program with a single event loop than to try to coordinate everything with a master multiprocessing process.

We've hit most of the high points of AsyncIO in this section, and the chapter has covered many other concurrency primitives. Concurrency is a hard problem to solve, and no one solution fits all use cases. The most important part of designing a concurrent system is deciding which of the available tools is the correct one to use for the problem. We have seen advantages and disadvantages of several concurrent systems, and now have some insights into which are the better choices for different types of requirements.

Case study

To wrap up this chapter, and the module, let's build a basic image compression tool. It will take black and white images (with 1 bit per pixel, either on or off) and attempt to compress it using a very basic form of compression known as run-length encoding. You may find black and white images a bit far-fetched. If so, you haven't enjoyed enough hours at `http://xkcd.com`!

I've included some sample black and white BMP images (which are easy to read data into and leave a lot of opportunity to improve on file size) with the example code for this chapter.

We'll be compressing the images using a simple technique called run-length encoding. This technique basically takes a sequence of bits and replaces any strings of repeated bits with the number of bits that are repeated. For example, the string 000011000 might be replaced with 04 12 03 to indicate that 4 zeros are followed by 2 ones and then 3 more zeroes. To make things a little more interesting, we will break each row into 127 bit chunks.

I didn't pick 127 bits arbitrarily. 127 different values can be encoded into 7 bits, which means that if a row contains all ones or all zeros, we can store it in a single byte; the first bit indicating whether it is a row of 0s or a row of 1s, and the remaining 7 bits indicating how many of that bit exists.

Breaking up the image into blocks has another advantage; we can process individual blocks in parallel without them depending on each other. However, there's a major disadvantage as well; if a run has just a few ones or zeros in it, then it will take up `more` space in the compressed file. When we break up long runs into blocks, we may end up creating more of these small runs and bloat the size of the file.

When dealing with files, we have to think about the exact layout of the bytes in the compressed file. Our file will store two byte little-endian integers at the beginning of the file representing the width and height of the completed file. Then it will write bytes representing the 127 bit chunks of each row.

Now before we start designing a concurrent system to build such compressed images, we should ask a fundamental question: Is this application I/O-bound or CPU-bound?

My answer, honestly, is "I don't know". I'm not sure whether the app will spend more time loading data from disk and writing it back or doing the compression in memory. I suspect that it is a CPU bound app in principle, but once we start passing image strings into subprocesses, we may lose any benefit of parallelism. The optimal solution to this problem is probably to write a C or Cython extension, but let's see how far we can get in pure Python.

We'll build this application using bottom-up design. That way we'll have some building blocks that we can combine into different concurrency patterns to see how they compare. Let's start with the code that compresses a 127-bit chunk using run-length encoding:

```
from bitarray import bitarray
def compress_chunk(chunk):
    compressed = bytearray()
    count = 1
    last = chunk[0]
    for bit in chunk[1:]:
        if bit != last:
            compressed.append(count | (128 * last))
            count = 0
            last = bit
        count += 1
    compressed.append(count | (128 * last))
    return compressed
```

This code uses the `bitarray` class for manipulating individual zeros and ones. It is distributed as a third-party module, which you can install with the command `pip install bitarray`. The chunk that is passed into `compress_chunks` is an instance of this class (although the example would work just as well with a list of Booleans). The primary benefit of the bitarray in this case is that when pickling them between processes, they take up an 8th of the space of a list of Booleans or a bytestring of 1s and 0s. Therefore, they pickle faster. They are also a bit (pun intended) easier to work with than doing a ton of bitwise operations.

The method compresses the data using run-length encoding and returns a bytearray containing the packed data. Where a bitarray is like a list of ones and zeros, a bytearray is like a list of byte objects (each byte, of course, containing 8 ones or zeros).

The algorithm that performs the compression is pretty simple (although I'd like to point out that it took me two days to implement and debug it. Simple to understand does not necessarily imply easy to write!). It first sets the `last` variable to the type of bit in the current run (either `True` or `False`). It then loops over the bits, counting each one, until it finds one that is different. When it does, it constructs a new byte by making the leftmost bit of the byte (the 128 position) either a zero or a one, depending on what the `last` variable contained. Then it resets the counter and repeats the operation. Once the loop is done, it creates one last byte for the last run, and returns the result.

While we're creating building blocks, let's make a function that compresses a row of image data:

```
def compress_row(row):
    compressed = bytearray()
    chunks = split_bits(row, 127)
    for chunk in chunks:
        compressed.extend(compress_chunk(chunk))
    return compressed
```

This function accepts a bitarray named row. It splits it into chunks that are each 127 bits wide using a function that we'll define very shortly. Then it compresses each of those chunks using the previously defined `compress_chunk`, concatenating the results into a `bytearray`, which it returns.

We define `split_bits` as a simple generator:

```
def split_bits(bits, width):
    for i in range(0, len(bits), width):
        yield bits[i:i+width]
```

Now, since we aren't certain yet whether this will run more effectively in threads or processes, let's wrap these functions in a method that runs everything in a provided executor:

```
def compress_in_executor(executor, bits, width):
    row_compressors = []
    for row in split_bits(bits, width):
        compressor = executor.submit(compress_row, row)
```

```
            row_compressors.append(compressor)

        compressed = bytearray()
        for compressor in row_compressors:
            compressed.extend(compressor.result())
        return compressed
```

This example barely needs explaining; it splits the incoming bits into rows based on the width of the image using the same `split_bits` function we have already defined (hooray for bottom-up design!).

Note that this code will compress any sequence of bits, although it would bloat, rather than compress binary data that has frequent changes in bit values. Black and white images are definitely good candidates for the compression algorithm in question. Let's now create a function that loads an image file using the third-party pillow module, converts it to bits, and compresses it. We can easily switch between executors using the venerable comment statement:

```
from PIL import Image
def compress_image(in_filename, out_filename, executor=None):
    executor = executor if executor else ProcessPoolExecutor()
    with Image.open(in_filename) as image:
        bits = bitarray(image.convert('1').getdata())
        width, height = image.size

    compressed = compress_in_executor(executor, bits, width)

    with open(out_filename, 'wb') as file:
        file.write(width.to_bytes(2, 'little'))
        file.write(height.to_bytes(2, 'little'))
        file.write(compressed)

def single_image_main():
    in_filename, out_filename = sys.argv[1:3]
    #executor = ThreadPoolExecutor(4)
    executor = ProcessPoolExecutor()
    compress_image(in_filename, out_filename, executor)
```

The `image.convert()` call changes the image to black and white (one bit) mode, while `getdata()` returns an iterator over those values. We pack the results into a bitarray so they transfer across the wire more quickly. When we output the compressed file, we first write the width and height of the image followed by the compressed data, which arrives as a bytearray, which can be written directly to the binary file.

Having written all this code, we are finally able to test whether thread pools or process pools give us better performance. I created a large (7200 x 5600 pixels) black and white image and ran it through both pools. The `ProcessPool` takes about 7.5 seconds to process the image on my system, while the `ThreadPool` consistently takes about 9. Thus, as we suspected, the cost of pickling bits and bytes back and forth between processes is eating almost all the efficiency gains from running on multiple processors (though looking at my CPU monitor, it does fully utilize all four cores on my machine).

So it looks like compressing a single image is most effectively done in a separate process, but only barely because we are passing so much data back and forth between the parent and subprocesses. Multiprocessing is more effective when the amount of data passed between processes is quite low.

So let's extend the app to compress all the bitmaps in a directory in parallel. The only thing we'll have to pass into the subprocesses are filenames, so we should get a speed gain compared to using threads. Also, to be kind of crazy, we'll use the existing code to compress individual images. This means we'll be running a `ProcessPoolExecutor` inside each subprocess to create even more subprocesses. I don't recommend doing this in real life!

```
from pathlib import Path
def compress_dir(in_dir, out_dir):
    if not out_dir.exists():
        out_dir.mkdir()

    executor = ProcessPoolExecutor()
    for file in (
            f for f in in_dir.iterdir() if f.suffix == '.bmp'):
        out_file = (out_dir / file.name).with_suffix('.rle')
        executor.submit(
            compress_image, str(file), str(out_file))

def dir_images_main():
    in_dir, out_dir = (Path(p) for p in sys.argv[1:3])
    compress_dir(in_dir, out_dir)
```

This code uses the `compress_image` function we defined previously, but runs it in a separate process for each image. It doesn't pass an executor into the function, so `compress_image` creates a `ProcessPoolExecutor` once the new process has started running.

Now that we are running executors inside executors, there are four combinations of threads and process pools that we can be using to compress images. They each have quite different timing profiles:

	Process pool per image	**Thread pool per image**
Process pool per row	42 seconds	53 seconds
Thread pool per row	34 seconds	64 seconds

As we might expect, using threads for each image and again using threads for each row is the slowest, since the GIL prevents us from doing any work in parallel. Given that we were slightly faster when using separate processes for each row when we were using a single image, you may be surprised to see that it is faster to use a `ThreadPool` feature for rows if we are processing each image in a separate process. Take some time to understand why this might be.

My machine contains only four processor cores. Each row in each image is being processed in a separate pool, which means that all those rows are competing for processing power. When there is only one image, we get a (very modest) speedup by running each row in parallel. However, when we increase the number of images being processed at once, the cost of passing all that row data into and out of a subprocess is actively stealing processing time from each of the other images. So, if we can process each image on a separate processor, where the only thing that has to get pickled into the subprocess pipe is a couple filenames, we get a solid speedup.

Thus, we see that different workloads require different concurrency paradigms. Even if we are just using futures we have to make informed decisions about what kind of executor to use.

Also note that for typically-sized images, the program runs quickly enough that it really doesn't matter which concurrency structures we use. In fact, even if we didn't use any concurrency at all, we'd probably end up with about the same user experience.

This problem could also have been solved using the threading and/or multiprocessing modules directly, though there would have been quite a bit more boilerplate code to write. You may be wondering whether or not AsyncIO would be useful here. The answer is: "probably not". Most operating systems don't have a good way to do non-blocking reads from the filesystem, so the library ends up wrapping all the calls in futures anyway.

For completeness, here's the code that I used to decompress the RLE images to confirm that the algorithm was working correctly (indeed, it wasn't until I fixed bugs in both compression and decompression, and I'm still not sure if it is perfect. I should have used test-driven development!):

```
from PIL import Image
import sys

def decompress(width, height, bytes):
    image = Image.new('1', (width, height))

    col = 0
    row = 0
    for byte in bytes:
        color = (byte & 128) >> 7
        count = byte & ~128
        for i in range(count):
            image.putpixel((row, col), color)
            row += 1
        if not row % width:
            col += 1
            row = 0
    return image


with open(sys.argv[1], 'rb') as file:
    width = int.from_bytes(file.read(2), 'little')
    height = int.from_bytes(file.read(2), 'little')

    image = decompress(width, height, file.read())
    image.save(sys.argv[2], 'bmp')
```

This code is fairly straightforward. Each run is encoded in a single byte. It uses some bitwise math to extract the color of the pixel and the length of the run. Then it sets each pixel from that run in the image, incrementing the row and column of the next pixel to check at appropriate intervals.

Your Coding Challenge

We've covered several different concurrency paradigms in this chapter and still don't have a clear idea of when each one is useful. As we saw in the case study, it is often a good idea to prototype a few different strategies before committing to one.

Concurrency in Python 3 is a huge topic and an entire module of this size could not cover everything there is to know about it. As your first exercise, I encourage you to check out several third-party libraries that may provide additional context:

- execnet, a library that permits local and remote share-nothing concurrency
- Parallel python, an alternative interpreter that can execute threads in parallel
- Cython, a python-compatible language that compiles to C and has primitives to release the gil and take advantage of fully parallel multi-threading
- PyPy-STM, an experimental implementation of software transactional memory on top of the ultra-fast PyPy implementation of the Python interpreter
- Gevent

Ankita Thakur

Your Course Guide

If you have used threads in a recent application, take a look at the code and see if you can make it more readable and less bug-prone by using futures. Compare thread and multiprocessing futures to see if you can gain anything by using multiple CPUs.

Try implementing an AsyncIO service for some basic HTTP requests. You may need to look up the structure of an HTTP request on the web; they are fairly simple ASCII packets to decipher. If you can get it to the point that a web browser can render a simple GET request, you'll have a good understanding of AsyncIO network transports and protocols.

Make sure you understand the race conditions that happen in threads when you access shared data. Try to come up with a program that uses multiple threads to set shared values in such a way that the data deliberately becomes corrupt or invalid.

Remember the link collector we covered for the case study in

Ankita Thakur

Your Course Guide

Chapter 7, Python Data Structures? Can you make it run faster by making requests in parallel? Is it better to use raw threads, futures, or AsyncIO for this?

Try writing the run-length encoding example using threads or multiprocessing directly. Do you get any speed gains? Is the code easier or harder to reason about? Is there any way to speed up the decompression script by using concurrency or parallelism?

Summary of Module 1 Chapter 14

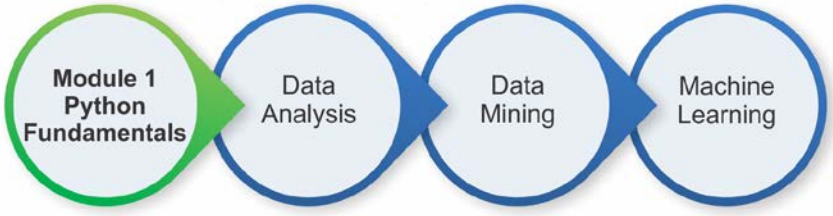
Ankita Thakur

Your Course Guide

This chapter ends our exploration of object-oriented programming with a topic that isn't very object-oriented. Concurrency is a difficult problem and we've only scratched the surface. While the underlying OS abstractions of processes and threads do not provide an API that is remotely object-oriented, Python offers some really good object-oriented abstractions around them. The threading and multiprocessing packages both provide an object-oriented interface to the underlying mechanics. Futures are able to encapsulate a lot of the messy details into a single object. AsyncIO uses coroutine objects to make our code read as though it runs synchronously, while hiding ugly and complicated implementation details behind a very simple loop abstraction.

I hope you've enjoyed the ride and are eager to start implementing object-oriented software in all your future projects!

Your Progress through the Course So Far



Course Module 2
Data Analysis

Course Module 1: Python Fundamentals
Chapter 1: Introduction and First Steps – Take a Deep Breath
Chapter 2: Object-oriented Design
Chapter 3: Objects in Python
Chapter 4: When Objects are alike
Chapter 5: Expecting the Unexpected
Chapter 6: When to use Object-oriented programming
Chapter 7: Python Data Structures
Chapter 8: Python Object-oriented Shortcuts
Chapter 9: Strings and Serialization
Chapter 10: The Iterator Pattern
Chapter 11: Python Design Patterns I
Chapter 12: Python Design Patterns II
Chapter 13: Testing Object-oriented Programs
Chapter 14: Concurrency

Course Module 2: Data Analysis
Chapter 1: Introducing Data Analysis and Libraries
Chapter 2: NumPy Arrays and Vectorized Computation
Chapter 3: Data Analysis with pandas
Chapter 4: Data Visualizaiton
Chapter 5: Time Series
Chapter 6: Interacting with Databases
Chapter 7: Data Analysis Application Examples

Jump into the field of Data Science with Course Module 2, Data Analysis

Course Module 3: Data Mining
Chapter 1: Getting Started with Data Mining
Chapter 2: Classifying with scikit-learn Estimators
Chapter 3: Predicting Sports Winners with Decision Trees
Chapter 4: Recommending Movies Using Affinity Analysis
Chapter 5: Extracting Features with Transformers
Chapter 6: Social Media Insight Using Naive Bayes
Chapter 7: Discovering Accounts to Follow Using Graph Mining
Chapter 8: Beating CAPTCHAs with Neural Networks
Chapter 9: Authorship Attribution
Chapter 10: Clustering News Articles
Chapter 11: Classifying Objects in Images Using Deep Learning
Chapter 12: Working with Big Data
Chapter 13: Next Steps…

Course Module 4: Machine Learning

Course Module 2

Since you've been exposed to programming and Python in module 1, now it's time to broaden your horizons and get to know some key libraries in the data analysis field. Let's jump into the world of data science through our second module, *Data Analysis*! I personally think that people with different backgrounds can benefit from this module. So, if you work in business, finance, in research and development at a lab or university, or if your work contains any data processing or analysis steps and you want know what Python has to offer, then this module can be of great help.

Ankita Thakur

Your Course Guide

The module starts by introducing the principles of data analysis and supported Python libraries, along with the basics of NumPy for numerical data processing. Next, it will provide an overview of pandas, a powerful library to solve data processing problems.

Moving on, the module will give a brief overview of the Matplotlib API and some common plotting functions for visualization. Next, it will explain how to manipulate time series data and how to persist data structures to files or databases. It will also show how we can reshape data to be able to ask interesting questions about it.

Sound interesting? What are you waiting for, let's dive in!

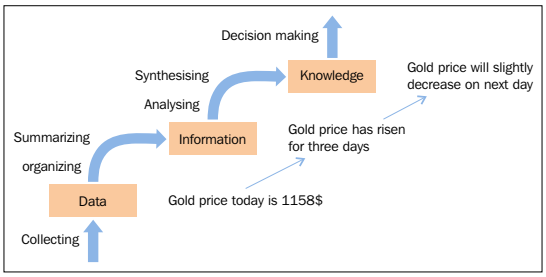
1
Introducing Data Analysis and Libraries

Data is raw information that can exist in any form, usable or not. We can easily get data everywhere in our lives; for example, the price of gold on the day of writing was $ 1.158 per ounce. This does not have any meaning, except describing the price of gold. This also shows that data is useful based on context.

With the relational data connection, information appears and allows us to expand our knowledge beyond the range of our senses. When we possess gold price data gathered over time, one piece of information we might have is that the price has continuously risen from $1.152 to $1.158 over three days. This could be used by someone who tracks gold prices.

Knowledge helps people to create value in their lives and work. This value is based on information that is organized, synthesized, or summarized to enhance comprehension, awareness, or understanding. It represents a state or potential for action and decisions. When the price of gold continuously increases for three days, it will likely decrease on the next day; this is useful knowledge.

The following figure illustrates the steps from data to knowledge; we call this process, the data analysis process and we will introduce it in the next section:

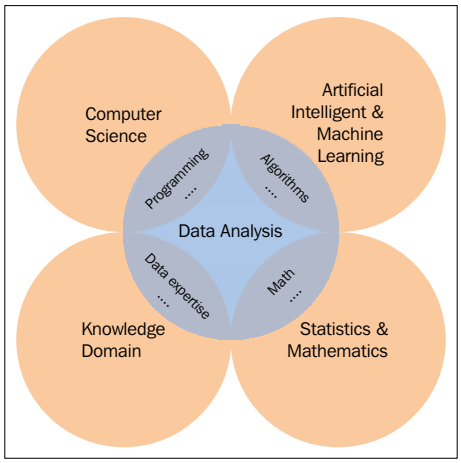


In this chapter, we will cover the following topics:

- Data analysis and process
- An overview of libraries in data analysis using different programming languages
- Common Python data analysis libraries

Data analysis and processing

Data is getting bigger and more diverse every day. Therefore, analyzing and processing data to advance human knowledge or to create value is a big challenge. To tackle these challenges, you will need domain knowledge and a variety of skills, drawing from areas such as computer science, **artificial intelligence** (**AI**) and **machine learning** (**ML**), statistics and mathematics, and knowledge domain, as shown in the following figure:



Let's go through data analysis and its domain knowledge:

- **Computer science**: We need this knowledge to provide abstractions for efficient data processing. Basic Python programming experience is required to follow the next chapters. We will introduce Python libraries used in data analysis.
- **Artificial intelligence and machine learning**: If computer science knowledge helps us to program data analysis tools, artificial intelligence and machine learning help us to model the data and learn from it in order to build smart products.
- **Statistics and mathematics**: We cannot extract useful information from raw data if we do not use statistical techniques or mathematical functions.
- **Knowledge domain**: Besides technology and general techniques, it is important to have an insight into the specific domain. What do the data fields mean? What data do we need to collect? Based on the expertise, we explore and analyze raw data by applying the preceding techniques, step by step.

Data analysis is a process composed of the following steps:

- **Data requirements**: We have to define what kind of data will be collected based on the requirements or problem analysis. For example, if we want to detect a user's behavior while reading news on the Internet, we should be aware of visited article links, dates and times, article categories, and the time the user spends on different pages.
- **Data collection**: Data can be collected from a variety of sources: mobile, personal computer, camera, or recording devices. It may also be obtained in different ways: communication, events, and interactions between person and person, person and device, or device and device. Data appears whenever and wherever in the world. The problem is how we can find and gather it to solve our problem? This is the mission of this step.
- **Data processing**: Data that is initially obtained must be processed or organized for analysis. This process is performance-sensitive. How fast can we create, insert, update, or query data? When building a real product that has to process big data, we should consider this step carefully. What kind of database should we use to store data? What kind of data structure, such as analysis, statistics, or visualization, is suitable for our purposes?

- **Data cleaning**: After being processed and organized, the data may still contain duplicates or errors. Therefore, we need a cleaning step to reduce those situations and increase the quality of the results in the following steps. Common tasks include record matching, deduplication, and column segmentation. Depending on the type of data, we can apply several types of data cleaning. For example, a user's history of visits to a news website might contain a lot of duplicate rows, because the user might have refreshed certain pages many times. For our specific issue, these rows might not carry any meaning when we explore the user's behavior so we should remove them before saving it to our database. Another situation we may encounter is click fraud on news—someone just wants to improve their website ranking or sabotage awebsite. In this case, the data will not help us to explore a user's behavior. We can use thresholds to check whether a visit page event comes from a real person or from malicious software.
- **Exploratory data analysis**: Now, we can start to analyze data through a variety of techniques referred to as exploratory data analysis. We may detect additional problems in data cleaning or discover requests for further data. Therefore, these steps may be iterative and repeated throughout the whole data analysis process. Data visualization techniques are also used to examine the data in graphs or charts. Visualization often facilitates understanding of data sets, especially if they are large or high-dimensional.
- **Modelling and algorithms**: A lot of mathematical formulas and algorithms may be applied to detect or predict useful knowledge from the raw data. For example, we can use similarity measures to cluster users who have exhibited similar news-reading behavior and recommend articles of interest to them next time. Alternatively, we can detect users' genders based on their news reading behavior by applying classification models such as the **Support Vector Machine** (**SVM**) or linear regression. Depending on the problem, we may use different algorithms to get an acceptable result. It can take a lot of time to evaluate the accuracy of the algorithms and choose the best one to implement for a certain product.

- **Data product**: The goal of this step is to build data products that receive data input and generate output according to the problem requirements. We will apply computer science knowledge to implement our selected algorithms as well as manage the data storage.

An overview of the libraries in data analysis

There are numerous data analysis libraries that help us to process and analyze data. They use different programming languages, and have different advantages and disadvantages of solving various data analysis problems. Now, we will introduce some common libraries that may be useful for you. They should give you an overview of the libraries in the field. However, the rest of this module focuses on Python-based libraries.

Some of the libraries that use the Java language for data analysis are as follows:

- **Weka**: This is the library that I became familiar with the first time I learned about data analysis. It has a graphical user interface that allows you to run experiments on a small dataset. This is great if you want to get a feel for what is possible in the data processing space. However, if you build a complex product, I think it is not the best choice, because of its performance, sketchy API design, non-optimal algorithms, and little documentation (`http://www.cs.waikato.ac.nz/ml/weka/`).

- **Mallet**: This is another Java library that is used for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other machine-learning applications on text. There is an add-on package for Mallet, called GRMM, that contains support for inference in general, graphical models, and training of **Conditional random fields** (**CRF**) with arbitrary graphical structures. In my experience, the library performance and the algorithms are better than Weka. However, its only focus is on text-processing problems. The reference page is at `http://mallet.cs.umass.edu/`.
- **Mahout**: This is Apache's machine-learning framework built on top of Hadoop; its goal is to build a scalable machine-learning library. It looks promising, but comes with all the baggage and overheads of Hadoop. The homepage is at `http://mahout.apache.org/`.
- **Spark**: This is a relatively new Apache project, supposedly up to a hundred times faster than Hadoop. It is also a scalable library that consists of common machine-learning algorithms and utilities. Development can be done in Python as well as in any JVM language. The reference page is at `https://spark.apache.org/docs/1.5.0/mllib-guide.html`.

Here are a few libraries that are implemented in C++:

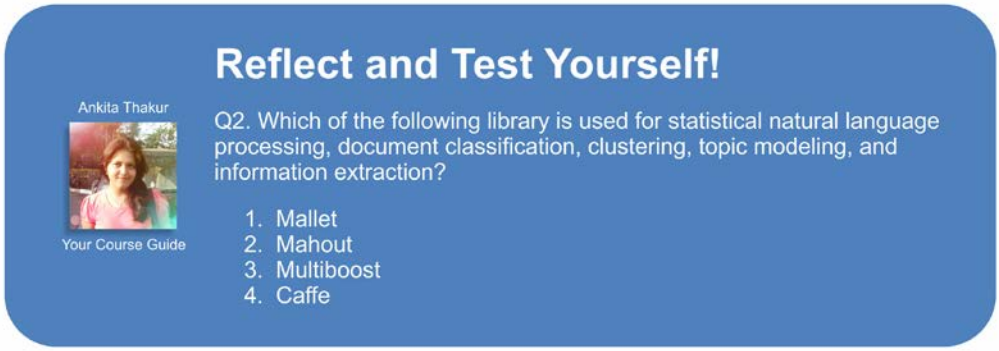
- **Vowpal Wabbit**: This library is a fast, out-of-core learning system sponsored by Microsoft Research and, previously, Yahoo! Research. It has been used to learn a tera-feature (1012) dataset on 1,000 nodes in one hour. More information can be found in the publication at `http://arxiv.org/abs/1110.4198`.
- **MultiBoost**: This package is a multiclass, multi label, and multitask classification boosting software implemented in C++. If you use this software, you should refer to the paper published in 2012 in the Journal*Machine Learning Research*, *MultiBoost: A Multi-purpose Boosting Package*, *D.Benbouzid*, *R. Busa-Fekete*, *N. Casagrande*, *F.-D. Collin*, and *B. Kégl*.
- **MLpack**: This is also a C++ machine-learning library, developed by the **Fundamental Algorithmic and Statistical Tools Laboratory** (**FASTLab**) at Georgia Tech. It focusses on scalability, speed, and ease-of-use, and was presented at the BigLearning workshop of NIPS 2011. Its homepage is at `http://www.mlpack.org/about.html`.

- **Caffe**: The last C++ library we want to mention is Caffe. This is a deep learning framework made with expression, speed, and modularity in mind. It is developed by the **Berkeley Vision and Learning Center** (**BVLC**) and community contributors. You can find more information about it at `http://caffe.berkeleyvision.org/`.

Other libraries for data processing and analysis are as follows:

- **Statsmodels**: This is a great Python library for statistical modeling and is mainly used for predictive and exploratory analysis.
- **Modular toolkit for data processing** (**MDP**): This is a collection of supervised and unsupervised learning algorithms and other data processing units that can be combined into data processing sequences and more complex feed-forward network architectures (`http://mdp-toolkit.sourceforge.net/index.html`).
- **Orange**: This is an open source data visualization and analysis for novices and experts. It is packed with features for data analysis and has add-ons for bioinformatics and text mining. It contains an implementation of self-organizing maps, which sets it apart from the other projects as well (`http://orange.biolab.si/`).
- **Mirador**: This is a tool for the visual exploration of complex datasets, supporting Mac and Windows. It enables users to discover correlation patterns and derive new hypotheses from data (`http://orange.biolab.si/`).
- **RapidMiner**: This is another GUI-based tool for data mining, machine learning, and predictive analysis (`https://rapidminer.com/`).
- **Theano**: This bridges the gap between Python and lower-level languages. Theano gives very significant performance gains, particularly for large matrix operations, and is, therefore, a good choice for deep learning models. However, it is not easy to debug because of the additional compilation layer.
- **Natural language processing toolkit** (**NLTK**): This is written in Python with very unique and salient features.

Here, I could not list all libraries for data analysis. However, I think the preceding libraries are enough to take a lot of your time to learn and build data analysis applications. I hope you will enjoy them after reading this module.



Python libraries in data analysis

Python is a multi-platform, general-purpose programming language that can run on Windows, Linux/Unix, and Mac OS X, and has been ported to Java and .NET virtual machines as well. It has a powerful standard library. In addition, it has many libraries for data analysis: Pylearn2, Hebel, Pybrain, Pattern, MontePython, and MILK. In this module, we will cover some common Python data analysis libraries such as Numpy, pandas, Matplotlib, PyMongo, and scikit-learn. Now, to help you get started, I will briefly present an overview of each library for those who are less familiar with the scientific Python stack.

NumPy

One of the fundamental packages used for scientific computing in Python is Numpy. Among other things, it contains the following:

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions for performing array computations
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra operations, Fourier transformations, and random number capabilities

Besides this, it can also be used as an efficient multidimensional container of generic data. Arbitrary data types can be defined and integrated with a wide variety of databases.

pandas

pandas is a Python package that supports rich data structures and functions for analyzing data, and is developed by the PyData Development Team. It is focused on the improvement of Python's data libraries. pandas consists of the following things:

- A set of labeled array data structures; the primary of which are Series, DataFrame, and Panel
- Index objects enabling both simple axis indexing and multilevel/hierarchical axis indexing
- An intergraded group by engine for aggregating and transforming datasets
- Date range generation and custom date offsets
- Input/output tools that load and save data from flat files or PyTables/HDF5 format
- Optimal memory versions of the standard data structures
- Moving window statistics and static and moving window linear/panel regression

Due to these features, pandas is an ideal tool for systems that need complex data structures or high-performance time series functions such as financial data analysis applications.

Matplotlib

Matplotlib is the single most used Python package for 2D-graphics. It provides both a very quick way to visualize data from Python and publication-quality figures in many formats: line plots, contour plots, scatter plots, and Basemap plots. It comes with a set of default settings, but allows customization of all kinds of properties. However, we can easily create our chart with the defaults of almost every property in Matplotlib.

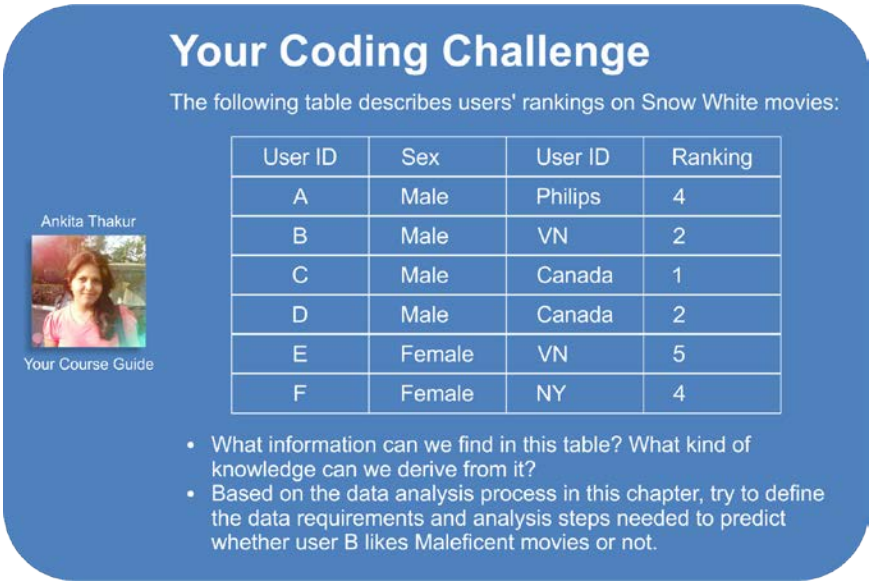
PyMongo

MongoDB is a type of NoSQL database. It is highly scalable, robust, and perfect to work with JavaScript-based web applications, because we can store data as JSON documents and use flexible schemas.

PyMongo is a Python distribution containing tools for working with MongoDB. Many tools have also been written for working with PyMongo to add more features such as MongoKit, Humongolus, MongoAlchemy, and Ming.

The scikit-learn library

The scikit-learn is an open source machine-learning library using the Python programming language. It supports various machine learning models, such as classification, regression, and clustering algorithms, interoperated with the Python numerical and scientific libraries NumPy and SciPy.

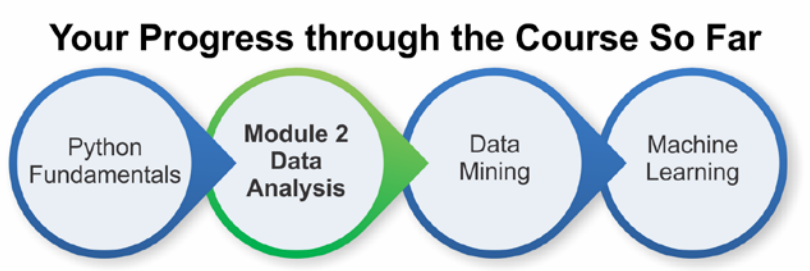


Summary of Module 2 Chapter 1

Ankita Thakur

Your Course Guide

In this chapter, we presented three main points. Firstly, we figured out the relationship between raw data, information and knowledge. Due to its contribution to our lives, we continued to discuss an overview of data analysis and processing steps in the second section. Finally, we introduced a few common supported libraries that are useful for practical data analysis applications. Among those, in the next chapters, we will focus on Python libraries in data analysis.



2
NumPy Arrays and Vectorized Computation

NumPy is the fundamental package supported for presenting and computing data with high performance in Python. It provides some interesting features as follows:

- Extension package to Python for multidimensional arrays (`ndarrays`), various derived objects (such as masked arrays), matrices providing vectorization operations, and broadcasting capabilities. Vectorization can significantly increase the performance of array computations by taking advantage of **Single Instruction Multiple Data** (**SIMD**) instruction sets in modern CPUs.
- Fast and convenient operations on arrays of data, including mathematical manipulation, basic statistical operations, sorting, selecting, linear algebra, random number generation, discrete Fourier transforms, and so on.
- Efficiency tools that are closer to hardware because of integrating C/C++/Fortran code.

NumPy is a good starting package for you to get familiar with arrays and array-oriented computing in data analysis. Also, it is the basic step to learn other, more effective tools such as pandas, which we will see in the next chapter. We will be using NumPy version 1.9.1.

NumPy arrays

An array can be used to contain values of a data object in an experiment or simulation step, pixels of an image, or a signal recorded by a measurement device. For example, the latitude of the Eiffel Tower, Paris is 48.858598 and the longitude is 2.294495. It can be presented in a NumPy array object as `p`:

```
>>> import numpy as np
>>> p = np.array([48.858598, 2.294495])
>>> p
Output: array([48.858598, 2.294495])
```

This is a manual construction of an array using the `np.array` function. The standard convention to import NumPy is as follows:

```
>>> import numpy as np
```

You can, of course, put `from numpy import *` in your code to avoid having to write `np`. However, you should be careful with this habit because of the potential code conflicts (further information on code conventions can be found in the *Python Style Guide*, also known as **PEP8**, at `https://www.python.org/dev/peps/pep-0008/`).

There are two requirements of a NumPy array: a fixed size at creation and a uniform, fixed data type, with a fixed size in memory. The following functions help you to get information on the `p` matrix:

```
>>> p.ndim    # getting dimension of array p
1
>>> p.shape   # getting size of each array dimension
(2,)
>>> len(p)    # getting dimension length of array p
2
>>> p.dtype    # getting data type of array p
dtype('float64')
```

Data types

There are five basic numerical types including Booleans (`bool`), integers (`int`), unsigned integers (`uint`), floating point (`float`), and complex. They indicate how many bits are needed to represent elements of an array in memory. Besides that, NumPy also has some types, such as `intc` and `intp`, that have different bit sizes depending on the platform.

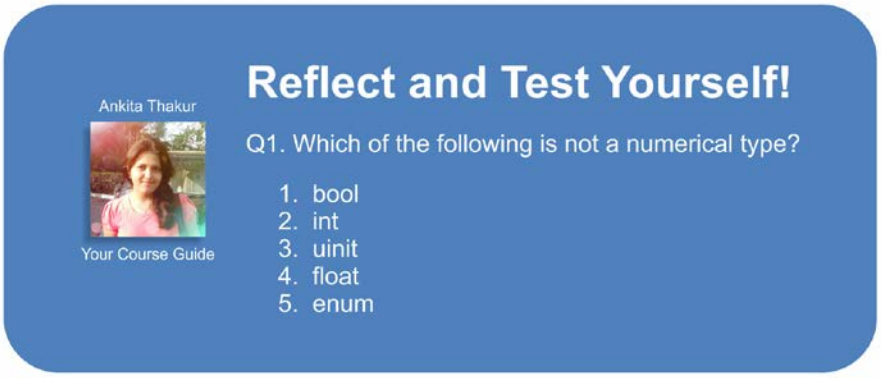
See the following table for a listing of NumPy's supported data types:

Type	Type code	Description	Range of value
`bool`		Boolean stored as a byte	True/False
`intc`		Similar to C int (int32 or int 64)	
`intp`		Integer used for indexing (same as C size_t)	
`int8`, `uint8`	i1, u1	Signed and unsigned 8-bit integer types	int8: (-128 to 127) uint8: (0 to 255)
`int16`, `uint16`	i2, u2	Signed and unsigned 16-bit integer types	int16: (-32768 to 32767) uint16: (0 to 65535)
`int32`, `uint32`	I4, u4	Signed and unsigned 32-bit integer types	int32: (-2147483648 to 2147483647 uint32: (0 to 4294967295)
`int64`, `uinit64`	i8, u8	Signed and unsigned 64-bit integer types	Int64: (-9223372036854775808 to 9223372036854775807) uint64: (0 to 18446744073709551615)
`float16`	f2	Half precision float: sign bit, 5 bits exponent, and 10b bits mantissa	
`float32`	f4 / f	Single precision float: sign bit, 8 bits exponent, and 23 bits mantissa	
`float64`	f8 / d	Double precision float: sign bit, 11 bits exponent, and 52 bits mantissa	
`complex64`, `complex128`, `complex256`	c8, c16, c32	Complex numbers represented by two 32-bit, 64-bit, and 128-bit floats	
`object`	0	Python object type	
`string_`	S	Fixed-length string type	Declare a string `dtype` with length 10, using `S10`
`unicode_`	U	Fixed-length Unicode type	Similar to string_ example, we have 'U10'

We can easily convert or cast an array from one `dtype` to another using the `astype` method:

```
>>> a = np.array([1, 2, 3, 4])
>>> a.dtype
dtype('int64')
>>> float_b = a.astype(np.float64)
>>> float_b.dtype
dtype('float64')
```

The `astype` function will create a new array with a copy of data from an old array, even though the new `dtype` is similar to the old one.



Array creation

There are various functions provided to create an array object. They are very useful for us to create and store data in a multidimensional array in different situations.

Now, in the following table we will summarize some of NumPy's common functions and their use by examples for array creation:

Function	Description	Example

`empty`, `empty_like`	Create a new array of the given shape and type, without initializing elements	`>>> np.empty([3,2], dtype=np.float64)`
		`array([[0., 0.], [0., 0.], [0., 0.]])`
		`>>> a = np.array([[1, 2], [4, 3]])`
		`>>> np.empty_like(a)`
		`array([[0, 0], [0, 0]])`

`eye`, `identity`	Create a NxN identity matrix with ones on the diagonal and zero elsewhere	`>>> np.eye(2, dtype=np.int)` `array([[1, 0], [0, 1]])`

`ones`, `ones_like`	Create a new array with the given shape and type, filled with 1s for all elements	`>>> np.ones(5)`
		`array([1., 1., 1., 1., 1.])`
		`>>> np.ones(4, dtype=np.int)`
		`array([1, 1, 1, 1])`
		`>>> x = np.array([[0,1,2], [3,4,5]])`
		`>>> np.ones_like(x)`
		`array([[1, 1, 1],[1, 1, 1]])`

`zeros`, `zeros_like`	This is similar to `ones`, `ones_like`, but initializing elements with 0s instead	`>>> np.zeros(5)`
		`array([0., 0., 0., 0-, 0.])`
		`>>> np.zeros(4, dtype=np.int)`
		`array([0, 0, 0, 0])`
		`>>> x = np.array([[0, 1, 2], [3, 4, 5]])`
		`>>> np.zeros_like(x)`
		`array([[0, 0, 0],[0, 0, 0]])`

`arange`	Create an array with even spaced values in a given interval	`>>> np.arange(2, 5)`
		`array([2, 3, 4])`
		`>>> np.arange(4, 12, 5)`
		`array([4, 9])`

`full`, `full_like`	Create a new array with the given shape and type, filled with a selected value	`>>> np.full((2,2), 3, dtype=np.int)`
		`array([[3, 3], [3, 3]])`
		`>>> x = np.ones(3)`
		`>>> np.full_like(x, 2)`
		`array([2., 2., 2.])`

Function	Description	Example
`array`	Create an array from the existing data	`>>> np.array([[1.1, 2.2, 3.3], [4.4, 5.5, 6.6]])` `array([1.1, 2.2, 3.3], [4.4, 5.5, 6.6]])`

`asarray`	Convert the input to an array	`>>> a = [3.14, 2.46]`
		`>>> np.asarray(a)`
		`array([3.14, 2.46])`

`copy`	Return an array copy of the given object	`>>> a = np.array([[1, 2], [3, 4]])`
		`>>> np.copy(a)`
		`array([[1, 2], [3, 4]])`

`fromstring`	Create 1-D array from a string or text	`>>> np.fromstring('3.14 2.17', dtype=np.float, sep=' ')` `array([3.14, 2.17])`

Indexing and slicing

As with other Python sequence types, such as lists, it is very easy to access and assign a value of each array's element:

```
>>> a = np.arange(7)
>>> a
array([0, 1, 2, 3, 4, 5, 6])
>>> a[1], a [4], a[-1]
(1, 4, 6)
```

In Python, array indices start at 0. This is in contrast to Fortran or Matlab, where indices begin at 1.

As another example, if our array is multidimensional, we need tuples of integers to index an item:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a[0, 2]      # first row, third column
3
>>> a[0, 2] = 10
>>> a
array([[1, 2, 10], [4, 5, 6], [7, 8, 9]])
>>> b = a[2]
>>> b
```

```
array([7, 8, 9])
>>> c = a[:2]
>>> c
array([[1, 2, 10], [4, 5, 6]])
```

We call `b` and `c` as array slices, which are views on the original one. It means that the data is not copied to `b` or `c`, and whenever we modify their values, it will be reflected in the array `a` as well:

```
>>> b[-1] = 11
>>> a
array([[1, 2, 10], [4, 5, 6], [7, 8, 11]])
```

We use a colon (`:`) character to take the entire axis when we omit the index number.

Fancy indexing

Besides indexing with slices, NumPy also supports indexing with Boolean or integer arrays (masks). This method is called **fancy indexing**. It creates copies, not views.

First, we take a look at an example of indexing with a Boolean mask array:

```
>>> a = np.array([3, 5, 1, 10])
>>> b = (a % 5 == 0)
>>> b
array([False, True, False, True], dtype=bool)
>>> c = np.array([[0, 1], [2, 3], [4, 5], [6, 7]])
>>> c[b]
array([[2, 3], [6, 7]])
```

The second example is an illustration of using integer masks on arrays:

```
>>> a = np.array([[1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12],
 [13, 14, 15, 16]])
>>> a[[2, 1]]
array([[9, 10, 11, 12], [5, 6, 7, 8]])
>>> a[[-2, -1]]          # select rows from the end
array([[ 9, 10, 11, 12], [13, 14, 15, 16]])
>>> a[[2, 3], [0, 1]]    # take elements at (2, 0) and (3, 1)
array([9, 14])
```

The mask array must have the same length as the axis that it's indexing.

Numerical operations on arrays

We are getting familiar with creating and accessing `ndarrays`. Now, we continue to the next step, applying some mathematical operations to array data without writing any for loops, of course, with higher performance.

Scalar operations will propagate the value to each element of the array:

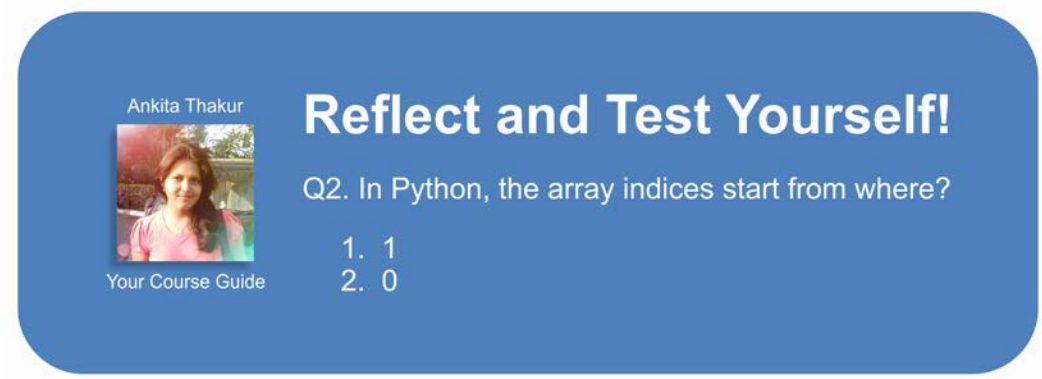
```
>>> a = np.ones(4)
>>> a * 2
array([2., 2., 2., 2.])
>>> a + 3
array([4., 4., 4., 4.])
```

All arithmetic operations between arrays apply the operation element wise:

```
>>> a = np.ones([2, 4])
>>> a * a
array([[1., 1., 1., 1.], [1., 1., 1., 1.]])
>>> a + a
array([[2., 2., 2., 2.], [2., 2., 2., 2.]])
```

Also, here are some examples of comparisons and logical operations:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([1, 1, 5, 3])
>>> a == b
array([True, False, False, False], dtype=bool)

>>> np.array_equal(a, b)      # array-wise comparison
False

>>> c = np.array([1, 0])
>>> d = np.array([1, 1])
>>> np.logical_and(c, d)      # logical operations
array([True, False])
```



Array functions

Many helpful array functions are supported in NumPy for analyzing data. We will list some part of them that are common in use. Firstly, the transposing function is another kind of reshaping form that returns a view on the original data array without copying anything:

```
>>> a = np.array([[0, 5, 10], [20, 25, 30]])
>>> a.reshape(3, 2)
array([[0, 5], [10, 20], [25, 30]])
>>> a.T
array([[0, 20], [5, 25], [10, 30]])
```

In general, we have the `swapaxes` method that takes a pair of axis numbers and returns a view on the data, without making a copy:

```
>>> a = np.array([[[0, 1, 2], [3, 4, 5]],
 [[6, 7, 8], [9, 10, 11]]])
>>> a.swapaxes(1, 2)
array([[[0, 3],
    [1, 4],
    [2, 5]],
   [[6, 9],
    [7, 10],
    [8, 11]]])
```

The transposing function is used to do matrix computations; for example, computing the inner matrix product `XT.X` using `np.dot`:

```
>>> a = np.array([[1, 2, 3],[4,5,6]])
>>> np.dot(a.T, a)
array([[17, 22, 27],
   [22, 29, 36],
   [27, 36, 45]])
```

Sorting data in an array is also an important demand in processing data. Let's take a look at some sorting functions and their use:

```
>>> a = np.array ([[6, 34, 1, 6], [0, 5, 2, -1]])

>>> np.sort(a)     # sort along the last axis
array([[1, 6, 6, 34], [-1, 0, 2, 5]])

>>> np.sort(a, axis=0)    # sort along the first axis
array([[0, 5, 1, -1], [6, 34, 2, 6]])

>>> b = np.argsort(a)    # fancy indexing of sorted array
>>> b
array([[2, 0, 3, 1], [3, 0, 2, 1]])
>>> a[0][b[0]]
array([1, 6, 6, 34])

>>> np.argmax(a)    # get index of maximum element
1
```

See the following table for a listing of array functions:

Function	Description	Example

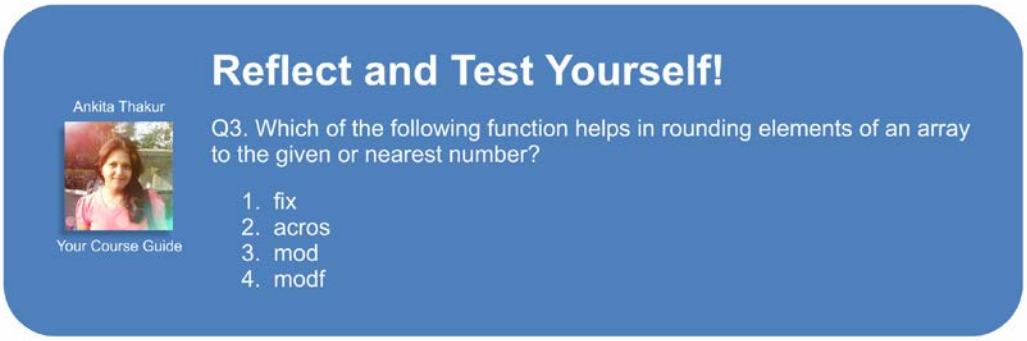
`sin`, `cos`, `tan`, `cosh`, `sinh`, `tanh`, `arcos`, `arctan`, `deg2rad`	Trigonometric and hyperbolic functions	`>>> a = np.array([0.,30., 45.])`
		`>>> np.sin(a * np.pi / 180)`
		`array([0., 0.5, 0.7071678])`

`around`, `round`, `rint`, `fix`, `floor`, `ceil`, `trunc`	Rounding elements of an array to the given or nearest number	`>>> a = np.array([0.34, 1.65])`
		`>>> np.round(a)`
		`array([0., 2.])`

Function	Description	Example
`sqrt`, `square`, `exp`, `expm1`, `exp2`, `log`, `log10`, `log1p`, `logaddexp`	Computing the exponents and logarithms of an array	`>>> np.exp(np.array([2.25, 3.16]))` `array([9.4877, 23.5705])`

`add`, `negative`, `multiply`, `devide`, `power`, `substract`, `mod`, `modf`, `remainder`	Set of arithmetic functions on arrays	`>>> a = np.arange(6)`
		`>>> x1 = a.reshape(2,3)`
		`>>> x2 = np.arange(3)`
		`>>> np.multiply(x1, x2)`
		`array([[0,1,4],[0,4,10]])`

`greater`, `greater_equal`, `less`, `less_equal`, `equal`, `not_equal`	Perform elementwise comparison: >, >=, <, <=, ==, !=	`>>> np.greater(x1, x2)` `array([[False, False, False], [True, True, True]], dtype = bool)`



Data processing using arrays

With the NumPy package, we can easily solve many kinds of data processing tasks without writing complex loops. It is very helpful for us to control our code as well as the performance of the program. In this part, we want to introduce some mathematical and statistical functions.

See the following table for a listing of mathematical and statistical functions:

Function	Description	Example

`sum`	Calculate the sum of all the elements in an array or along the axis	`>>> a = np.array([[2,4], [3,5]])`
		`>>> np.sum(a, axis=0)`
		`array([5, 9])`

`prod`	Compute the product of array elements over the given axis	`>>> np.prod(a, axis=1)` `array([8, 15])`
`diff`	Calculate the discrete difference along the given axis	`>>> np.diff(a, axis=0)` `array([[1,1]])`
`gradient`	Return the gradient of an array	`>>> np.gradient(a)` `[array([[1., 1.], [1., 1.]]), array([[2., 2.], [2., 2.]])]`

`cross`	Return the cross product of two arrays	`>>> b = np.array([[1,2], [3,4]])`
		`>>> np.cross(a,b)`
		`array([0, -3])`

`std`, `var`	Return standard deviation and variance of arrays	`>>> np.std(a)`
		`1.1180339`
		`>>> np.var(a)`
		`1.25`

`mean`	Calculate arithmetic mean of an array	`>>> np.mean(a)` `3.5`
`where`	Return elements, either from x or y, that satisfy a condition	`>>> np.where([[True, True], [False, True]], [[1,2],[3,4]], [[5,6],[7,8]])` `array([[1,2], [7, 4]])`

`unique`	Return the sorted unique values in an array	`>>> id = np.array(['a', 'b', 'c', 'c', 'd'])`
		`>>> np.unique(id)`
		`array(['a', 'b', 'c', 'd'], dtype='|S1')`

Function	Description	Example

`intersect1d`	Compute the sorted and common elements in two arrays	`>>> a = np.array(['a', 'b', 'a', 'c', 'd', 'c'])`
		`>>> b = np.array(['a', 'xyz', 'klm', 'd'])`
		`>>> np.intersect1d(a,b)`
		`array(['a', 'd'], dtype='|S3')`

Loading and saving data

We can also save and load data to and from a disk, either in text or binary format, by using different supported functions in NumPy package.

Saving an array

Arrays are saved by default in an uncompressed raw binary format, with the file extension `.npy` by the `np.save` function:

```
>>> a = np.array([[0, 1, 2], [3, 4, 5]])
>>> np.save('test1.npy', a)
```

The library automatically assigns the `.npy` extension, if we omit it.

If we want to store several arrays into a single file in an uncompressed `.npz` format, we can use the `np.savez` function, as shown in the following example:

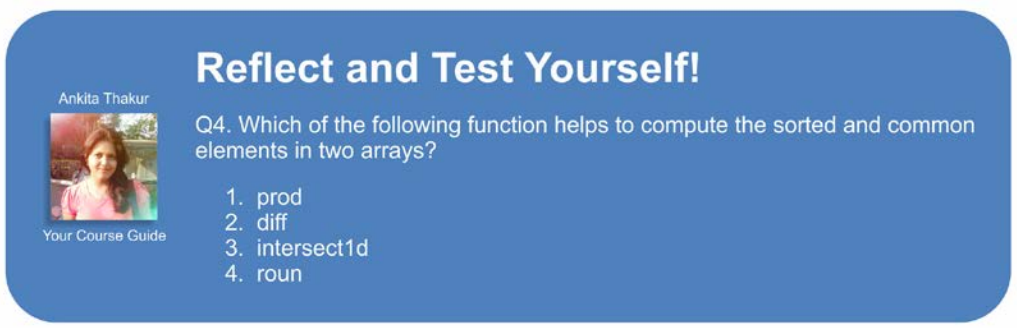
```
>>> a = np.arange(4)
>>> b = np.arange(7)
>>> np.savez('test2.npz', arr0=a, arr1=b)
```

The `.npz` file is a zipped archive of files named after the variables they contain. When we load an `.npz` file, we get back a dictionary-like object that can be queried for its lists of arrays:

```
>>> dic = np.load('test2.npz')
>>> dic['arr0']
array([0, 1, 2, 3])
```

Another way to save array data into a file is using the `np.savetxt` function that allows us to set format properties in the output file:

```
>>> x = np.arange(4)
>>> # e.g., set comma as separator between elements
>>> np.savetxt('test3.out', x, delimiter=',')
```



Loading an array

We have two common functions such as `np.load` and `np.loadtxt`, which correspond to the saving functions, for loading an array:

```
>>> np.load('test1.npy')
array([[0, 1, 2], [3, 4, 5]])
>>> np.loadtxt('test3.out', delimiter=',')
array([0., 1., 2., 3.])
```

Similar to the `np.savetxt` function, the `np.loadtxt` function also has a lot of options for loading an array from a text file.

Linear algebra with NumPy

Linear algebra is a branch of mathematics concerned with vector spaces and the mappings between those spaces. NumPy has a package called **linalg** that supports powerful linear algebra functions. We can use these functions to find eigenvalues and eigenvectors or to perform singular value decomposition:

```
>>> A = np.array([[1, 4, 6],
    [5, 2, 2],
    [-1, 6, 8]])
```

```
>>> w, v = np.linalg.eig(A)
>>> w                              # eigenvalues
array([-0.111 + 1.5756j, -0.111 - 1.5756j, 11.222+0.j])
>>> v                              # eigenvector
array([[-0.0981 + 0.2726j, -0.0981 - 0.2726j, 0.5764+0.j],
    [0.7683+0.j, 0.7683-0.j, 0.4591+0.j],
    [-0.5656 - 0.0762j, -0.5656 + 0.00763j, 0.6759+0.j]])
```

The function is implemented using the geev Lapack routines that compute the eigenvalues and eigenvectors of general square matrices.

Another common problem is solving linear systems such as `Ax = b` with `A` as a matrix and `x` and `b` as vectors. The problem can be solved easily using the `numpy.linalg.solve` function:

```
>>> A = np.array([[1, 4, 6], [5, 2, 2], [-1, 6, 8]])
>>> b = np.array([[1], [2], [3]])
>>> x = np.linalg.solve(A, b)
>>> x
array([[-1.77635e-16], [2.5], [-1.5]])
```

The following table will summarise some commonly used functions in the `numpy.linalg` package:

Function	Description	Example

`dot`	Calculate the dot product of two arrays	`>>> a = np.array([[1, 0],[0, 1]])`
		`>>> b = np.array( [[4, 1],[2, 2]])`
		`>>> np.dot(a,b)`
		`array([[4, 1],[2, 2]])`

`inner, outer`	Calculate the inner and outer product of two arrays	`>>> a = np.array([1, 1, 1])`
		`>>> b = np.array([3, 5, 1])`
		`>>> np.inner(a,b)`
		`9`

`linalg.norm`	Find a matrix or vector norm	`>>> a = np.arange(3)`
		`>>> np.linalg.norm(a)`
		`2.23606`

`linalg.det`	Compute the determinant of an array	`>>> a = np.array([[1,2],[3,4]])`
		`>>> np.linalg.det(a)`
		`-2.0`

Function	Description	Example

`linalg.inv`	Compute the inverse of a matrix	`>>> a = np.array([[1,2],[3,4]])`
		`>>> np.linalg.inv(a)`
		`array([[-2., 1.],[1.5, -0.5]])`

`linalg.qr`	Calculate the QR decomposition	`>>> a = np.array([[1,2],[3,4]])`
		`>>> np.linalg.qr(a)`
		`(array([[0.316, 0.948], [0.948, 0.316]]), array([[ 3.162, 4.427], [ 0., 0.632]]))`

`linalg.cond`	Compute the condition number of a matrix	`>>> a = np.array([[1,3],[2,4]])`
		`>>> np.linalg.cond(a)`
		`14.933034`

`trace`	Compute the sum of the diagonal element	`>>> np.trace(np.arange(6)).`
		`reshape(2,3))`
		`4`

NumPy random numbers

An important part of any simulation is the ability to generate random numbers. For this purpose, NumPy provides various routines in the submodule `random`. It uses a particular algorithm, called the Mersenne Twister, to generate pseudorandom numbers.

First, we need to define a seed that makes the random numbers predictable. When the value is reset, the same numbers will appear every time. If we do not assign the seed, NumPy automatically selects a random seed value based on the system's random number generator device or on the clock:

```
>>> np.random.seed(20)
```

An array of random numbers in the `[0.0, 1.0]` interval can be generated as follows:

```
>>> np.random.rand(5)
array([0.5881308, 0.89771373, 0.89153073, 0.81583748,
         0.03588959])
>>> np.random.rand(5)
array([0.69175758, 0.37868094, 0.51851095, 0.65795147,
       0.19385022])
```

```
>>> np.random.seed(20)    # reset seed number
>>> np.random.rand(5)
array([0.5881308, 0.89771373, 0.89153073, 0.81583748,
       0.03588959])
```

If we want to generate random integers in the half-open interval `[min, max]`, we can user the `randint`(`min`, `max`, `length`) function:

```
>>> np.random.randint(10, 20, 5)
array([17, 12, 10, 16, 18])
```

NumPy also provides for many other distributions, including the `Beta`, `bionomial`, `chi-square`, `Dirichlet`, `exponential`, `F`, `Gamma`, `geometric`, or `Gumbel`.

The following table will list some distribution functions and give examples for generating random numbers:

Function	Description	Example

`binomial`	Draw samples from a binomial distribution (n: number of trials, p: probability)	`>>> n, p = 100, 0.2`
		`>>> np.random.binomial(n, p, 3)`
		`array([17, 14, 23])`

`dirichlet`	Draw samples using a Dirichlet distribution	`>>> np.random.`
		`dirichlet(alpha=(2,3), size=3)`
		`array([[0.519, 0.480], [0.639, 0.36],`
		`[0.838, 0.161]])`

`poisson`	Draw samples from a Poisson distribution	`>>> np.random.poisson(lam=2, size=2)` `array([4,1])`

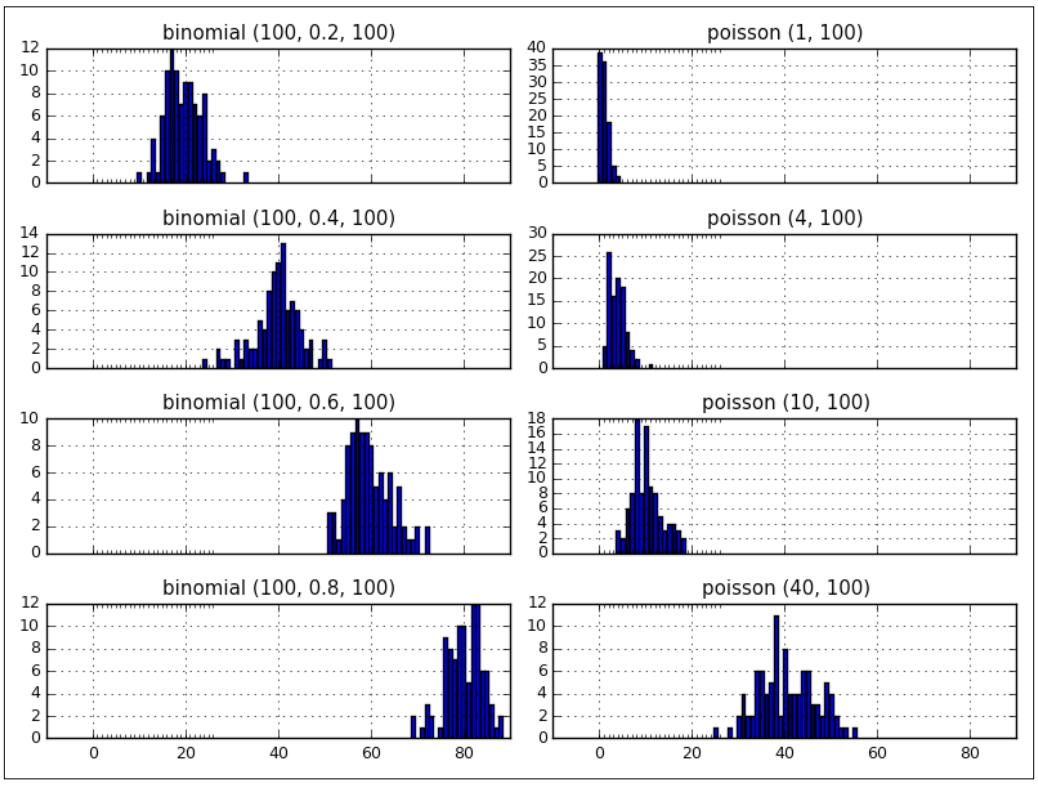
`normal`	Draw samples using a normal Gaussian distribution	`>>> np.random.normal`
		`(loc=2.5, scale=0.3, size=3)`
		`array([2.4436, 2.849, 2.741)`

`uniform`	Draw samples using a uniform distribution	`>>> np.random.uniform(`
		`low=0.5, high=2.5, size=3)`
		`array([1.38, 1.04, 2.19[)`

We can also use the random number generation to shuffle items in a list. Sometimes this is useful when we want to sort a list in a random order:

```
>>> a = np.arange(10)
>>> np.random.shuffle(a)
>>> a
array([7, 6, 3, 1, 4, 2, 5, 0, 9, 8])
```

The following figure shows two distributions, `binomial` and `poisson`, side by side with various parameters (the visualization was created with `matplotlib`, which will be covered in *Chapter 4, Data Visualization*):



Your Coding Challenge

1. 1. Using an array creation function, let's try to create arrays variable in the following situations:
   - Create ndarray from the existing data
   - Initialize ndarray which elements are filled with ones, zeros, or a given interval
   - Loading and saving data from a file to an ndarray

2. What is the difference between np.dot(a, b) and (a*b)?

3. Consider the vector [1, 2, 3, 4, 5] building a new vector with four consecutive zeros interleaved between each value.

4. Taking the data example file chapter2-data.txt, which includes information on a system log, and solve the following tasks:
   - Try to build an ndarray from the data file
   - Statistic frequency of each device type in the built matrix
   - List unique OS that appears in the data log
   - Sort user by provinceID and count the number of users in each province

Ankita Thakur

Your Course Guide

Summary of Module 2 Chapter 2

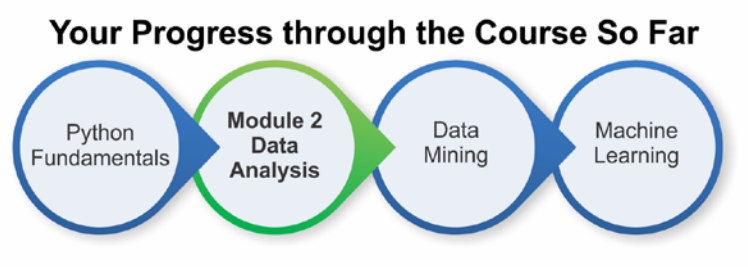
In this chapter, we covered a lot of information related to the NumPy package, especially commonly used functions that are very helpful to process and analyze data in ndarray. Firstly, we learned the properties and data type of ndarray in the NumPy package. Secondly, we focused on how to create and manipulate an ndarray in different ways, such as conversion from other structures, reading an array from disk, or just generating a new array with given values. Thirdly, we studied how to access and control the value of each element in ndarray by using indexing and slicing.

Then, we are getting familiar with some common functions and operations on ndarray. And finally, we continue with some advance functions that are related to statistic, linear algebra and sampling data. Those functions play important role in data analysis.

However, while NumPy by itself does not provide very much high-level data analytical functionality, having an understanding of it will help you use tools such as pandas much more effectively. This tool will be discussed in the next chapter.

Ankita Thakur

Your Course Guide


Your Progress through the Course So Far
Python Fundamentals
Module 2 Data Analysis
Data Mining
Machine Learning

3
Data Analysis with pandas

In this chapter, we will explore another data analysis library called pandas. The goal of this chapter is to give you some basic knowledge and concrete examples for getting started with pandas.

An overview of the pandas package

pandas is a Python package that supports fast, flexible, and expressive data structures, as well as computing functions for data analysis. The following are some prominent features that pandas supports:

- Data structure with labeled axes. This makes the program clean and clear and avoids common errors from misaligned data.
- Flexible handling of missing data.
- Intelligent label-based slicing, fancy indexing, and subset creation of large datasets.
- Powerful arithmetic operations and statistical computations on a custom axis via axis label.
- Robust input and output support for loading or saving data from and to files, databases, or HDF5 format.

Related to pandas installation, we recommend an easy way, that is to install it as a part of Anaconda, a cross-platform distribution for data analysis and scientific computing. You can refer to the reference at `http://docs.continuum.io/anaconda/` to download and install the library.

After installation, we can use it like other Python packages. Firstly, we have to import the following packages at the beginning of the program:

```
>>> import pandas as pd
>>> import numpy as np
```

The pandas data structure

Let's first get acquainted with two of pandas' primary data structures: the Series and the DataFrame. They can handle the majority of use cases in finance, statistic, social science, and many areas of engineering.

Series

A Series is a one-dimensional object similar to an array, list, or column in table. Each item in a Series is assigned to an entry in an index:

```
>>> s1 = pd.Series(np.random.rand(4),
                   index=['a', 'b', 'c', 'd'])
>>> s1
a    0.6122
b    0.98096
c    0.3350
d    0.7221
dtype: float64
```

By default, if no index is passed, it will be created to have values ranging from `0` to `N-1`, where `N` is the length of the Series:

```
>>> s2 = pd.Series(np.random.rand(4))
>>> s2
0    0.6913
1    0.8487
2    0.8627
3    0.7286
dtype: float64
```

We can access the value of a Series by using the index:

```
>>> s1['c']
0.3350
>>>s1['c'] = 3.14
>>> s1['c', 'a', 'b']
c    3.14
a    0.6122
b    0.98096
```

This accessing method is similar to a Python dictionary. Therefore, pandas also allows us to initialize a Series object directly from a Python dictionary:

```
>>> s3 = pd.Series({'001': 'Nam', '002': 'Mary',
                    '003': 'Peter'})
>>> s3
001    Nam
002    Mary
003    Peter
dtype: object
```

Sometimes, we want to filter or rename the index of a Series created from a Python dictionary. At such times, we can pass the selected index list directly to the initial function, similarly to the process in the preceding example. Only elements that exist in the index list will be in the Series object. Conversely, indexes that are missing in the dictionary are initialized to default `NaN` values by pandas:

```
>>> s4 = pd.Series({'001': 'Nam', '002': 'Mary',
                    '003': 'Peter'}, index=[
                    '002', '001', '024', '065'])
>>> s4
002    Mary
001    Nam
024    NaN
065    NaN
dtype:   object
ect
```

The library also supports functions that detect missing data:

```
>>> pd.isnull(s4)
002    False
001    False
024    True
065    True
dtype: bool
```

Similarly, we can also initialize a Series from a scalar value:

```
>>> s5 = pd.Series(2.71, index=['x', 'y'])
>>> s5
x    2.71
y    2.71
dtype: float64
```

A Series object can be initialized with NumPy objects as well, such as `ndarray`. Moreover, pandas can automatically align data indexed in different ways in arithmetic operations:

```
>>> s6 = pd.Series(np.array([2.71, 3.14]), index=['z', 'y'])
>>> s6
z    2.71
y    3.14
dtype: float64
>>> s5 + s6
x    NaN
y    5.85
z    NaN
dtype: float64
```

The DataFrame

The DataFrame is a tabular data structure comprising a set of ordered columns and rows. It can be thought of as a group of Series objects that share an index (the column names). There are a number of ways to initialize a DataFrame object. Firstly, let's take a look at the common example of creating DataFrame from a dictionary of lists:

```
>>> data = {'Year': [2000, 2005, 2010, 2014],
         'Median_Age': [24.2, 26.4, 28.5, 30.3],
```

```
         'Density': [244, 256, 268, 279]}
>>> df1 = pd.DataFrame(data)
>>> df1
    Density    Median_Age    Year
0  244         24.2         2000
1  256         26.4         2005
2  268         28.5         2010
3  279         30.3         2014
```

By default, the DataFrame constructor will order the column alphabetically. We can edit the default order by passing the column's attribute to the initializing function:

```
>>> df2 = pd.DataFrame(data, columns=['Year', 'Density',
                                      'Median_Age'])
>>> df2
    Year    Density    Median_Age
0    2000    244         24.2
1    2005    256         26.4
2    2010    268         28.5
3    2014    279         30.3
>>> df2.index
Int64Index([0, 1, 2, 3], dtype='int64')
```

We can provide the index labels of a DataFrame similar to a Series:

```
>>> df3 = pd.DataFrame(data, columns=['Year', 'Density',
                   'Median_Age'], index=['a', 'b', 'c', 'd'])
>>> df3.index
Index([u'a', u'b', u'c', u'd'], dtype='object')
```

We can construct a DataFrame out of nested lists as well:

```
>>> df4 = pd.DataFrame([
    ['Peter', 16, 'pupil', 'TN', 'M', None],
    ['Mary', 21, 'student', 'SG', 'F', None],
    ['Nam', 22, 'student', 'HN', 'M', None],
    ['Mai', 31, 'nurse', 'SG', 'F', None],
    ['John', 28, 'laywer', 'SG', 'M', None]],
columns=['name', 'age', 'career', 'province', 'sex', 'award'])
```

Columns can be accessed by column name as a Series can, either by dictionary-like notation or as an attribute, if the column name is a syntactically valid attribute name:

```
>>> df4.name    # or df4['name']
0    Peter
1    Mary
2    Nam
3    Mai
4    John
Name: name, dtype: object
```

To modify or append a new column to the created DataFrame, we specify the column name and the value we want to assign:

```
>>> df4['award'] = None
>>> df4
    name age   career province  sex award
0  Peter  16    pupil       TN    M  None
1    Mary  21  student       SG    F  None
2    Nam   22  student       HN  M  None
3    Mai    31    nurse        SG    F    None
4    John    28    lawer        SG    M    None
```

Using a couple of methods, rows can be retrieved by position or name:

```
>>> df4.ix[1]
name           Mary
age              21
career      student
province         SG
sex               F
award          None
Name: 1, dtype: object
```

A DataFrame object can also be created from different data structures such as a list of dictionaries, a dictionary of Series, or a record array. The method to initialize a DataFrame object is similar to the preceding examples.

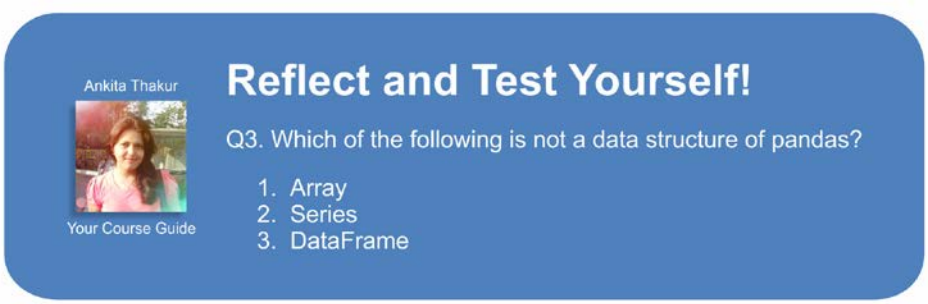
Another common case is to provide a DataFrame with data from a location such as a text file. In this situation, we use the `read_csv` function that expects the column separator to be a comma, by default. However, we can change that by using the `sep` parameter:

```
# person.csv file
name,age,career,province,sex
Peter,16,pupil,TN,M
Mary,21,student,SG,F
Nam,22,student,HN,M
Mai,31,nurse,SG,F
John,28,lawyer,SG,M
# loading person.cvs into a DataFrame
>>> df4 = pd.read_csv('person.csv')
>>> df4
     name   age   career   province  sex
0    Peter    16    pupil       TN        M
1    Mary     21    student     SG       F
2    Nam      22    student     HN       M
3    Mai      31    nurse       SG       F
4    John     28    lawyer      SG       M
```

While reading a data file, we sometimes want to skip a line or an invalid value. As for pandas `0.16.2`, `read_csv` supports over 50 parameters for controlling the loading process. Some common useful parameters are as follows:

- `sep`: This is a delimiter between columns. The default is comma symbol.
- `dtype`: This is a data type for data or columns.
- `header`: This sets row numbers to use as the column names.
- `skiprows`: This skips line numbers to skip at the start of the file.
- `error_bad_lines`: This shows invalid lines (too many fields) that will, by default, cause an exception, such that no DataFrame will be returned. If we set the value of this parameter as `false`, the bad lines will be skipped.

Moreover, pandas also has support for reading and writing a DataFrame directly from or to a database such as the `read_frame` or `write_frame` function within the pandas module. We will come back to these methods later in this chapter.



The essential basic functionality

pandas supports many essential functionalities that are useful to manipulate pandas data structures. In this module, we will focus on the most important features regarding exploration and analysis.

Reindexing and altering labels

Reindex is a critical method in the pandas data structures. It confirms whether the new or modified data satisfies a given set of labels along a particular axis of pandas object.

First, let's view a `reindex` example on a Series object:

```
>>> s2.reindex([0, 2, 'b', 3])
0    0.6913
2    0.8627
b       NaN
3    0.7286
dtype: float64
```

When `reindexed` labels do not exist in the data object, a default value of `NaN` will be automatically assigned to the position; this holds true for the DataFrame case as well:

```
>>> df1.reindex(index=[0, 2, 'b', 3],
        columns=['Density', 'Year', 'Median_Age','C'])
   Density  Year  Median_Age        C
```

```
0      244  2000        24.2      NaN
2      268  2010        28.5      NaN
b      NaN   NaN         NaN      NaN
3      279  2014        30.3      NaN
```

We can change the `NaN` value in the missing index case to a custom value by setting the `fill_value` parameter. Let us take a look at the arguments that the `reindex` function supports, as shown in the following table:

Argument	Description
`index`	This is the new labels/index to conform to.
`method`	This is the method to use for filling holes in a `reindexed` object. The default setting is unfill gaps. `pad/ffill`: fill values forward `backfill/bfill`: fill values backward `nearest`: use the nearest value to fill the gap
`copy`	This return a new object. The default setting is `true`.
`level`	The matches index values on the passed multiple index level.
`fill_value`	This is the value to use for missing values. The default setting is `NaN`.
`limit`	This is the maximum size gap to fill in `forward` or `backward` method.

Head and tail

In common data analysis situations, our data structure objects contain many columns and a large number of rows. Therefore, we cannot view or load all information of the objects. pandas supports functions that allow us to inspect a small sample. By default, the functions return five elements, but we can set a custom number as well. The following example shows how to display the first five and the last three rows of a longer Series:

```
>>> s7 = pd.Series(np.random.rand(10000))
>>> s7.head()
0    0.631059
1    0.766085
2    0.066891
3    0.867591
4    0.339678
```

```
dtype: float64
>>> s7.tail(3)
9997    0.412178
9998    0.800711
9999    0.438344
dtype: float64
```

We can also use these functions for DataFrame objects in the same way.

Binary operations

Firstly, we will consider arithmetic operations between objects. In different indexes objects case, the expected result will be the union of the index pairs. We will not explain this again because we had an example about it in the previous section (`s5 + s6`). This time, we will show another example with a DataFrame:

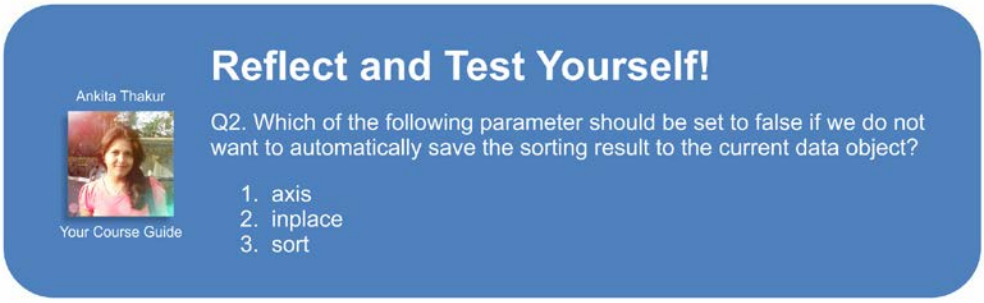
```
>>> df5 = pd.DataFrame(np.arange(9).reshape(3,3)
                       columns=['a','b','c'])
>>> df5
   a  b  c
0  0  1  2
1  3  4  5
2  6  7  8
>>> df6 = pd.DataFrame(np.arange(8).reshape(2,4),
                      columns=['a','b','c','d'])
>>> df6
   a  b  c  d
0  0  1  2  3
1  4  5  6  7
>>> df5 + df6
    a   b   c   d
0   0   2   4 NaN
1   7   9  11 NaN
2   NaN NaN NaN NaN
```

The mechanisms for returning the result between two kinds of data structure are similar. A problem that we need to consider is the missing data between objects. In this case, if we want to fill with a fixed value, such as `0`, we can use the arithmetic functions such as `add`, `sub`, `div`, and `mul`, and the function's supported parameters such as `fill_value`:

```
>>> df7 = df5.add(df6, fill_value=0)
>>> df7
   a  b   c   d
0  0  2   4   3
1  7  9  11   7
2  6  7   8  NaN
```

Next, we will discuss comparison operations between data objects. We have some supported functions such as **equal** (**eq**), **not equal** (**ne**), **greater than** (**gt**), **less than** (**lt**), **less equal** (**le**), and **greater equal** (**ge**). Here is an example:

```
>>> df5.eq(df6)
       a      b      c      d
0   True   True   True  False
1  False  False  False  False
2  False  False  False  False
```



Functional statistics

The supported statistics method of a library is really important in data analysis. To get inside a big data object, we need to know some summarized information such as mean, sum, or quantile. pandas supports a large number of methods to compute them. Let's consider a simple example of calculating the `sum` information of `df5`, which is a DataFrame object:

```
>>> df5.sum()
a     9
```

```
b    12
c    15
dtype: int64
```

When we do not specify which axis we want to calculate `sum` information, by default, the function will calculate on index axis, which is axis `0`:

- **Series**: We do not need to specify the axis.
- **DataFrame**: Columns (`axis = 1`) or index (`axis = 0`). The default setting is `axis 0`.

We also have the `skipna` parameter that allows us to decide whether to exclude missing data or not. By default, it is set as `true`:

```
>>> df7.sum(skipna=False)
a    13
b    18
c    23
d   NaN
dtype: float64
```

Another function that we want to consider is `describe()`. It is very convenient for us to summarize most of the statistical information of a data structure such as the Series and DataFrame, as well:

```
>>> df5.describe()
         a    b    c
count  3.0  3.0  3.0
mean   3.0  4.0  5.0
std    3.0  3.0  3.0
min    0.0  1.0  2.0
25%    1.5  2.5  3.5
50%    3.0  4.0  5.0
75%    4.5  5.5  6.5
max    6.0  7.0  8.0
```

We can specify percentiles to include or exclude in the output by using the `percentiles` parameter; for example, consider the following:

```
>>> df5.describe(percentiles=[0.5, 0.8])
         a    b    c
count  3.0  3.0  3.0
```

```
mean    3.0  4.0  5.0
std     3.0  3.0  3.0
min     0.0  1.0  2.0
50%     3.0  4.0  5.0
80%     4.8  5.8  6.8
max     6.0  7.0  8.0
```

Here, we have a summary table for common supported statistics functions in pandas:

Function	Description
`idxmin(axis), idxmax(axis)`	This compute the index labels with the minimum or maximum corresponding values.
`value_counts()`	This compute the frequency of unique values.
`count()`	This return the number of non-null values in a data object.
`mean(),median(), min(),max()`	This return mean, median, minimum, and maximum values of an axis in a data object.
`std(),var(),sem()`	These return the standard deviation, variance, and standard error of mean.
`abs()`	This gets the absolute value of a data object.

Function application

pandas supports function application that allows us to apply some functions supported in other packages such as NumPy or our own functions on data structure objects. Here, we illustrate two examples of these cases, firstly, using `apply` to execute the `std()` function, which is the standard deviation calculating function of the NumPy package:

```
>>> df5.apply(np.std, axis=1)    # default: axis=0
0    0.816497
1    0.816497
2    0.816497
dtype: float64
```

Secondly, if we want to apply a formula to a data object, we can also use apply function by following these steps:

1. Define the function or formula that you want to apply on a data object.
2. Call the defined function or formula via `apply`. In this step, we also need to figure out the axis that we want to apply the calculation to:

```
>>> f = lambda x: x.max() - x.min()    # step 1
>>> df5.apply(f, axis=1)               # step 2
0    2
1    2
2    2
dtype: int64
>>> def sigmoid(x):
    return 1/(1 + np.exp(x))
>>> df5.apply(sigmoid)
          a           b         c
0  0.500000  0.268941  0.119203
1  0.047426  0.017986  0.006693
2  0.002473  0.000911  0.000335
```

Sorting

There are two kinds of sorting method that we are interested in: sorting by row or column index and sorting by data value.

Firstly, we will consider methods for sorting by row and column index. In this case, we have the `sort_index()` function. We also have `axis` parameter to set whether the function should sort by row or column. The `ascending` option with the `true` or `false` value will allow us to sort data in ascending or descending order. The default setting for this option is `true`:

```
>>> df7 = pd.DataFrame(np.arange(12).reshape(3,4),
                       columns=['b', 'd', 'a', 'c'],
                       index=['x', 'y', 'z'])
>>> df7
   b  d   a   c
x  0  1   2   3
y  4  5   6   7
z  8  9  10  11
```

```
>>> df7.sort_index(axis=1)
    a  b   c  d
x   2  0   3  1
y   6  4   7  5
z  10  8  11  9
```

Series has a method order that sorts by value. For `NaN` values in the object, we can also have a special treatment via the `na_position` option:

```
>>> s4.order(na_position='first')
024     NaN
065     NaN
002    Mary
001     Nam
dtype: object
>>> s4
002    Mary
001     Nam
024     NaN
065     NaN
dtype: object
```

Besides that, Series also has the `sort()` function that sorts data by value. However, the function will not return a copy of the sorted data:

```
>>> s4.sort(na_position='first')
>>> s4
024     NaN
065     NaN
002    Mary
001     Nam
dtype: object
```

If we want to apply sort function to a DataFrame object, we need to figure out which columns or rows will be sorted:

```
>>> df7.sort(['b', 'd'], ascending=False)
   b  d   a   c
z  8  9  10  11
y  4  5   6   7
x  0  1   2   3
```

If we do not want to automatically save the sorting result to the current data object, we can change the setting of the `inplace` parameter to `False`.

Indexing and selecting data

In this section, we will focus on how to get, set, or slice subsets of pandas data structure objects. As we learned in previous sections, Series or DataFrame objects have axis labeling information. This information can be used to identify items that we want to select or assign a new value to in the object:

```
>>> s4[['024', '002']]    # selecting data of Series object
024     NaN
002    Mary
dtype: object
>>> s4[['024', '002']] = 'unknown' # assigning data
>>> s4
024    unknown
065        NaN
002    unknown
001        Nam
dtype: object
```

If the data object is a DataFrame structure, we can also proceed in a similar way:

```
>>> df5[['b', 'c']]
   b  c
0  1  2
1  4  5
2  7  8
```

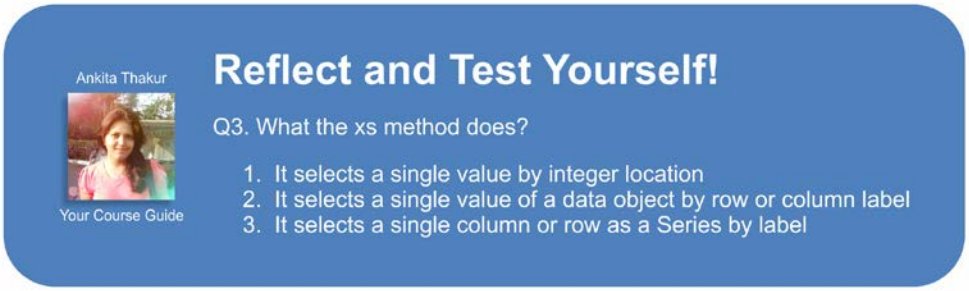
For label indexing on the rows of DataFrame, we use the `ix` function that enables us to select a set of rows and columns in the object. There are two parameters that we need to specify: the `row` and `column` labels that we want to get. By default, if we do not specify the selected column names, the function will return selected rows with all columns in the object:

```
>>> df5.ix[0]
a    0
b    1
c    2
Name: 0, dtype: int64
>>> df5.ix[0, 1:3]
b    1
c    2
Name: 0, dtype: int64
```

Moreover, we have many ways to select and edit data contained in a pandas object. We summarize these functions in the following table:

Method	Description
`icol`, `irow`	This selects a single row or column by integer location.
`get_value`, `set_value`	This selects or sets a single value of a data object by row or column label.
`xs`	This selects a single column or row as a Series by label.

pandas data objects may contain duplicate indices. In this case, when we get or set a data value via index label, it will affect all rows or columns that have the same selected index name.



Computational tools

Let's start with correlation and covariance computation between two data objects. Both the Series and DataFrame have a `cov` method. On a DataFrame object, this method will compute the covariance between the Series inside the object:

```
>>> s1 = pd.Series(np.random.rand(3))
>>> s1
0    0.460324
1    0.993279
2    0.032957
dtype: float64
>>> s2 = pd.Series(np.random.rand(3))
>>> s2
0    0.777509
1    0.573716
2    0.664212
dtype: float64
>>> s1.cov(s2)
-0.024516360159045424

>>> df8 = pd.DataFrame(np.random.rand(12).reshape(4,3),
                       columns=['a','b','c'])
>>> df8
          a         b         c
0  0.200049  0.070034  0.978615
1  0.293063  0.609812  0.788773
2  0.853431  0.243656  0.978057
0.985584  0.500765  0.481180
>>> df8.cov()
          a         b         c
a  0.155307  0.021273 -0.048449
b  0.021273  0.059925 -0.040029
c -0.048449 -0.040029  0.055067
```

Usage of the correlation method is similar to the covariance method. It computes the correlation between Series inside a data object in case the data object is a DataFrame. However, we need to specify which method will be used to compute the correlations. The available methods are `pearson`, `kendall`, and `spearman`. By default, the function applies the `spearman` method:

```
>>> df8.corr(method = 'spearman')
     a    b    c
a  1.0  0.4 -0.8
b  0.4  1.0 -0.8
c -0.8 -0.8  1.0
```

We also have the `corrwith` function that supports calculating correlations between Series that have the same label contained in different DataFrame objects:

```
>>> df9 = pd.DataFrame(np.arange(8).reshape(4,2),
                       columns=['a', 'b'])
>>> df9
   a  b
0  0  1
1  2  3
2  4  5
3  6  7
>>> df8.corrwith(df9)
a    0.955567
b    0.488370
c         NaN
dtype: float64
```

Working with missing data

In this section, we will discuss missing, `NaN`, or `null` values, in pandas data structures. It is a very common situation to arrive with missing data in an object. One such case that creates missing data is reindexing:

```
>>> df8 = pd.DataFrame(np.arange(12).reshape(4,3),
                       columns=['a', 'b', 'c'])
   a   b   c
0  0   1   2
1  3   4   5
```

```
2  6   7   8
3  9  10  11
>>> df9 = df8.reindex(columns = ['a', 'b', 'c', 'd'])
   a   b   c   d
0  0   1   2 NaN
1  3   4   5 NaN
2  6   7   8 NaN
4  9  10  11 NaN
>>> df10 = df8.reindex([3, 2, 'a', 0])
    a   b   c
3   9  10  11
2   6   7   8
a NaN NaN NaN
0   0   1   2
```

To manipulate missing values, we can use the `isnull()` or `notnull()` functions to detect the missing values in a Series object, as well as in a DataFrame object:

```
>>> df10.isnull()
       a      b      c
3  False  False  False
2  False  False  False
a   True   True   True
0  False  False  False
```

On a Series, we can drop all `null` data and index values by using the `dropna` function:

```
>>> s4 = pd.Series({'001': 'Nam', '002': 'Mary',
                    '003': 'Peter'},
                    index=['002', '001', '024', '065'])
>>> s4
002    Mary
001     Nam
024     NaN
065     NaN
dtype: object
>>> s4.dropna()    # dropping all null value of Series object
002    Mary
```

```
001     Nam
dtype: object
```

With a DataFrame object, it is a little bit more complex than with Series. We can tell which rows or columns we want to drop and also if all entries must be `null` or a single `null` value is enough. By default, the function will drop any row containing a missing value:

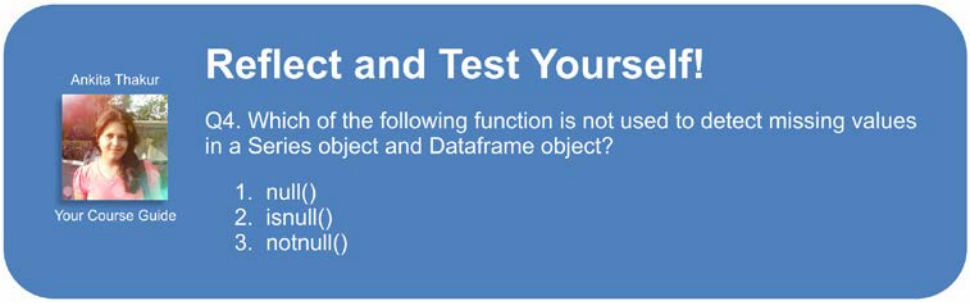
```
>>> df9.dropna()    # all rows will be dropped
Empty DataFrame
Columns: [a, b, c, d]
Index: []
>>> df9.dropna(axis=1)
   a   b   c
0  0   1   2
1  3   4   5
2  6   7   8
3  9  10  11
```

Another way to control missing values is to use the supported parameters of functions that we introduced in the previous section. They are also very useful to solve this problem. In our experience, we should assign a fixed value in missing cases when we create data objects. This will make our objects cleaner in later processing steps. For example, consider the following:

```
>>> df11 = df8.reindex([3, 2, 'a', 0], fill_value = 0)
>>> df11
   a   b   c
3  9  10  11
2  6   7   8
a  0   0   0
0  0   1   2
```

We can alse use the `fillna` function to fill a custom value in missing values:

```
>>> df9.fillna(-1)
   a   b   c  d
0  0   1   2 -1
1  3   4   5 -1
2  6   7   8 -1
3  9  10  11 -1
```



Advanced uses of pandas for data analysis

In this section we will consider some advanced pandas use cases.

Hierarchical indexing

Hierarchical indexing provides us with a way to work with higher dimensional data in a lower dimension by structuring the data object into multiple index levels on an axis:

```
>>> s8 = pd.Series(np.random.rand(8), index=[['a','a','b','b','c','c',
'd','d'], [0, 1, 0, 1, 0,1, 0, 1, ]])
>>> s8
a  0    0.721652
   1    0.297784
b  0    0.271995
   1    0.125342
c  0    0.444074
   1    0.948363
d  0    0.197565
   1    0.883776
dtype: float64
```

In the preceding example, we have a Series object that has two index levels. The object can be rearranged into a DataFrame using the `unstack` function. In an inverse situation, the `stack` function can be used:

```
>>> s8.unstack()
          0         1
a  0.549211  0.420874
b  0.051516  0.715021
c  0.503072  0.720772
d  0.373037  0.207026
```

We can also create a DataFrame to have a hierarchical index in both axes:

```
>>> df = pd.DataFrame(np.random.rand(12).reshape(4,3),
                      index=[['a', 'a', 'b', 'b'],
                             [0, 1, 0, 1]],
                      columns=[['x', 'x', 'y'], [0, 1, 0]])
>>> df
            x                   y
            0         1         0
a 0  0.636893  0.729521  0.747230
  1  0.749002  0.323388  0.259496
b 0  0.214046  0.926961  0.679686
0.013258  0.416101  0.626927
>>> df.index
MultiIndex(levels=[['a', 'b'], [0, 1]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
>>> df.columns
MultiIndex(levels=[['x', 'y'], [0, 1]],
           labels=[[0, 0, 1], [0, 1, 0]])
```

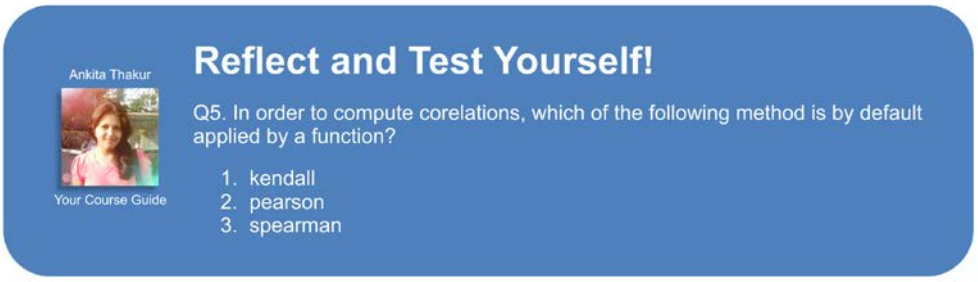
The methods for getting or setting values or subsets of the data objects with multiple index levels are similar to those of the nonhierarchical case:

```
>>> df['x']
            0         1
a 0  0.636893  0.729521
  1  0.749002  0.323388
b 0  0.214046  0.926961
```

```
0.013258  0.416101
>>> df[[0]]
            x
            0
a 0  0.636893
  1  0.749002
b 0  0.214046
0.013258
>>> df.ix['a', 'x']
          0         1
0  0.636893  0.729521
0.749002  0.323388
>>> df.ix['a','x'].ix[1]
0    0.749002
1    0.323388
Name: 1, dtype: float64
```

After grouping data into multiple index levels, we can also use most of the descriptive and statistics functions that have a level option, which can be used to specify the level we want to process:

```
>>> df.std(level=1)
          x                   y
          0         1         0
0  0.298998  0.139611  0.047761
0.520250  0.065558  0.259813
>>> df.std(level=0)
          x                   y
          0         1         0
a  0.079273  0.287180  0.344880
b  0.141979  0.361232  0.037306
```



The Panel data

The Panel is another data structure for three-dimensional data in pandas. However, it is less frequently used than the Series or the DataFrame. You can think of a Panel as a table of DataFrame objects. We can create a Panel object from a 3D `ndarray` or a dictionary of DataFrame objects:

```
# create a Panel from 3D ndarray
>>> panel = pd.Panel(np.random.rand(2, 4, 5),
                     items = ['item1', 'item2'])
>>> panel
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 5 (minor_axis)
Items axis: item1 to item2
Major_axis axis: 0 to 3
Minor_axis axis: 0 to 4

>>> df1 = pd.DataFrame(np.arange(12).reshape(4, 3),
                       columns=['a','b','c'])
>>> df1
   a   b   c
0  0   1   2
1  3   4   5
2  6   7   8
9  10  11
>>> df2 = pd.DataFrame(np.arange(9).reshape(3, 3),
                       columns=['a','b','c'])
```

```
>>> df2
   a  b  c
0  0  1  2
1  3  4  5
6  7  8
# create another Panel from a dict of DataFrame objects
>>> panel2 = pd.Panel({'item1': df1, 'item2': df2})
>>> panel2
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)
Items axis: item1 to item2
Major_axis axis: 0 to 3
Minor_axis axis: a to c
```

Each item in a Panel is a DataFrame. We can select an item, by item name:

```
>>> panel2['item1']
   a   b   c
0  0   1   2
1  3   4   5
2  6   7   8
3  9  10  11
```

Alternatively, if we want to select data via an axis or data position, we can use the `ix` method, like on Series or DataFrame:

```
>>> panel2.ix[:, 1:3, ['b', 'c']]
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: item1 to item2
Major_axis axis: 1 to 3
Minor_axis axis: b to c
>>> panel2.ix[:, 2, :]
   item1  item2
a      6      6
b      7      7
c      8      8
```

Your Coding Challenge

Ankita Thakur

Your Course Guide

1. We have US census dataset available at https://www.census.gov/2010census/csv/pop_change.csv. It has 23 columns and one row for each US state, as well as a few rows for macro regions such as North, South, and West.
   - Get this dataset into a pandas DataFrame. Hint: just skip those rows that do not seem helpful, such as comments or description.
   - While the dataset contains change metrics for each decade, we are interested in the population change during the second half of the twentieth century, that is, between 1950 and 2000. Which region has seen the biggest and the smallest population growth in this time span? Also, which US state?

2. Find more census data on the internet; not just on the US but on the world's countries. Try to find GDP data for the same time as well. Try to align this data to explore patterns. How are GDP and population growth related? Are there any special cases such as countries with high GDP but low population growth or countries with the opposite history?

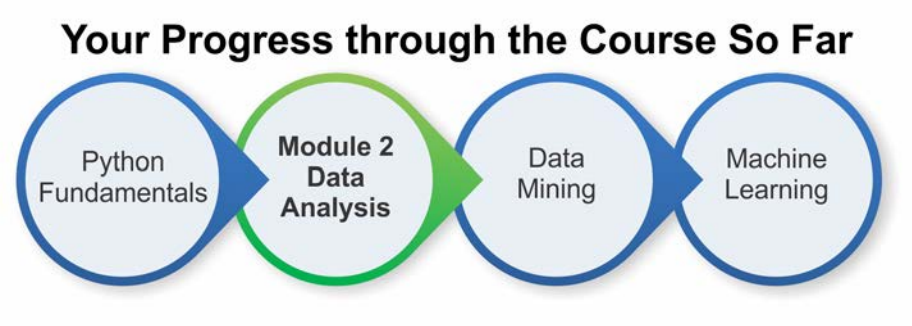
Summary of Module 2 Chapter 3

Ankita Thakur

Your Course Guide

We have finished covering the basics of the pandas data analysis library. Whenever you learn about a library for data analysis, you need to consider the three parts that we explained in this chapter. Data structures: we have two common data object types in the pandas library; Series and DataFrames. Method to access and manipulate data objects: pandas supports many way to select, set or slice subsets of data object. However, the general mechanism is using index labels or the positions of items to identify values. Functions and utilities: they are the most important part of a powerful library. In this chapter, we covered all common supported functions of pandas which allow us compute statistics on data easily. The library also has a lot of other useful functions and utilities that we could not explain in this chapter. We encourage you to start your own research, if you want to expand your experience with pandas. It helps us to process large data in an optimized way. You will see more of pandas in action later in this book.

Until now, we learned about two popular Python libraries: NumPy and pandas. pandas is built on NumPy, and as a result it allows for a bit more convenient interaction with data. However, in some situations, we can flexibly combine both of them to accomplish our goals.


Your Progress through the Course So Far
Python Fundamentals
Module 2 Data Analysis
Data Mining
Machine Learning

4
Data Visualization

Data visualization is concerned with the presentation of data in a pictorial or graphical form. It is one of the most important tasks in data analysis, since it enables us to see analytical results, detect outliers, and make decisions for model building. There are many Python libraries for visualization, of which matplotlib, seaborn, bokeh, and ggplot are among the most popular. However, in this chapter, we mainly focus on the matplotlib library that is used by many people in many different contexts.

Matplotlib produces publication-quality figures in a variety of formats, and interactive environments across Python platforms. Another advantage is that pandas comes equipped with useful wrappers around several matplotlib plotting routines, allowing for quick and handy plotting of Series and DataFrame objects.

The IPython package started as an alternative to the standard interactive Python shell, but has since evolved into an indispensable tool for data exploration, visualization, and rapid prototyping. It is possible to use the graphical capabilities offered by matplotlib from IPython through various options, of which the simplest to get started with is the `pylab` flag:

```
$ ipython --pylab
```

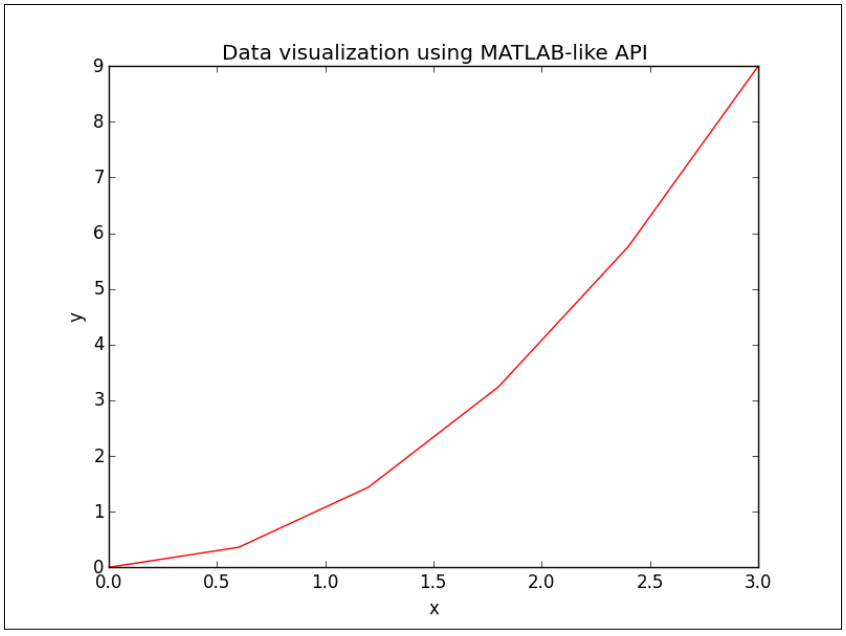
This flag will preload `matplotlib` and `numpy` for interactive use with the default matplotlib backend. IPython can run in various environments: in a terminal, as a `Qt` application, or inside a browser. These options are worth exploring, since IPython has enjoyed adoption for many use cases, such as prototyping, interactive slides for more engaging conference talks or lectures, and as a tool for sharing research.

The matplotlib API primer

The easiest way to get started with plotting using matplotlib is often by using the MATLAB API that is supported by the package:

```
>>> import matplotlib.pyplot as plt
>>> from numpy import *
>>> x = linspace(0, 3, 6)
>>> x
array([0., 0.6, 1.2, 1.8, 2.4, 3.])
>>> y = power(x,2)
>>> y
array([0., 0.36, 1.44, 3.24, 5.76, 9.])
>>> figure()
>>> plot(x, y, 'r')
>>> xlabel('x')
>>> ylabel('y')
>>> title('Data visualization in MATLAB-like API')
>>> plt.show()
```

The output for the preceding command is as follows:



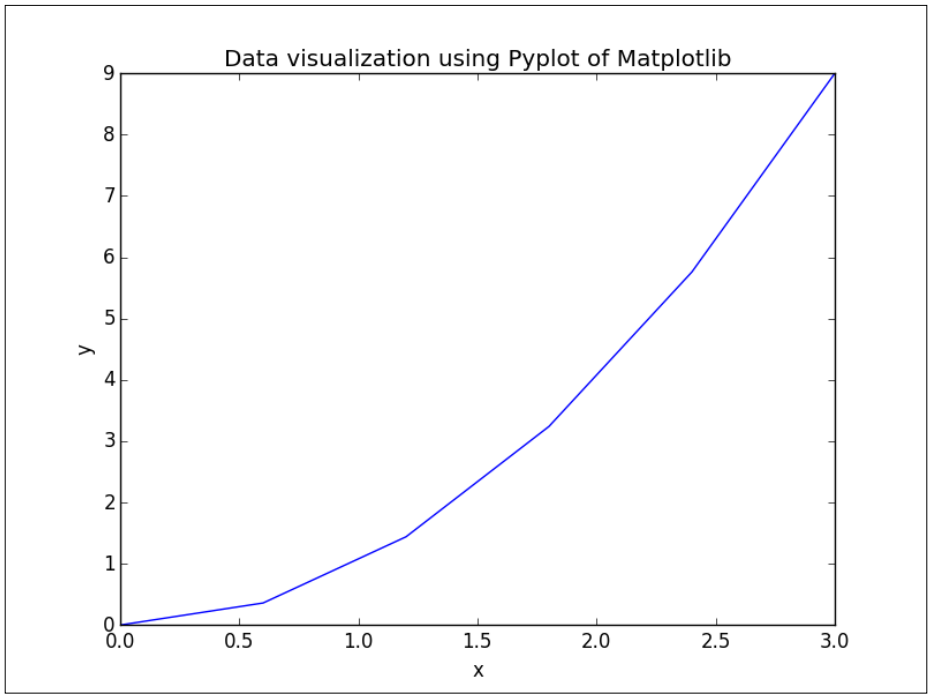
However, star imports should not be used unless there is a good reason for doing so. In the case of matplotlib, we can use the canonical import:

```
>>> import matplotlib.pyplot as plt
```

The preceding example could then be written as follows:

```
>>> plt.plot(x, y)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.title('Data visualization using Pyplot of Matplotlib')
>>> plt.show()
```

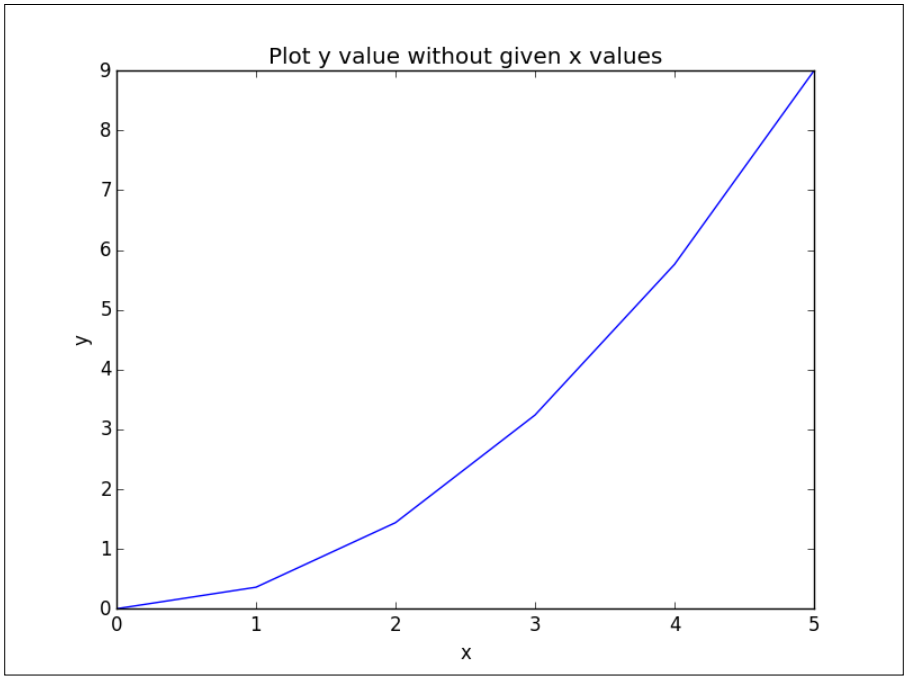
The output for the preceding command is as follows:



If we only provide a single argument to the plot function, it will automatically use it as the `y` values and generate the `x` values from `0` to `N-1`, where `N` is equal to the number of values:

```
>>> plt.plot(y)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.title('Plot y value without given x values')
>>> plt.show()
```

The output for the preceding command is as follows:



By default, the range of the axes is constrained by the range of the input `x` and `y` data. If we want to specify the `viewport` of the axes, we can use the `axis()` method to set custom ranges. For example, in the previous visualization, we could increase the range of the `x` axis from `[0, 5]` to `[0, 6]`, and that of the `y` axis from `[0, 9]` to `[0, 10]`, by writing the following command:

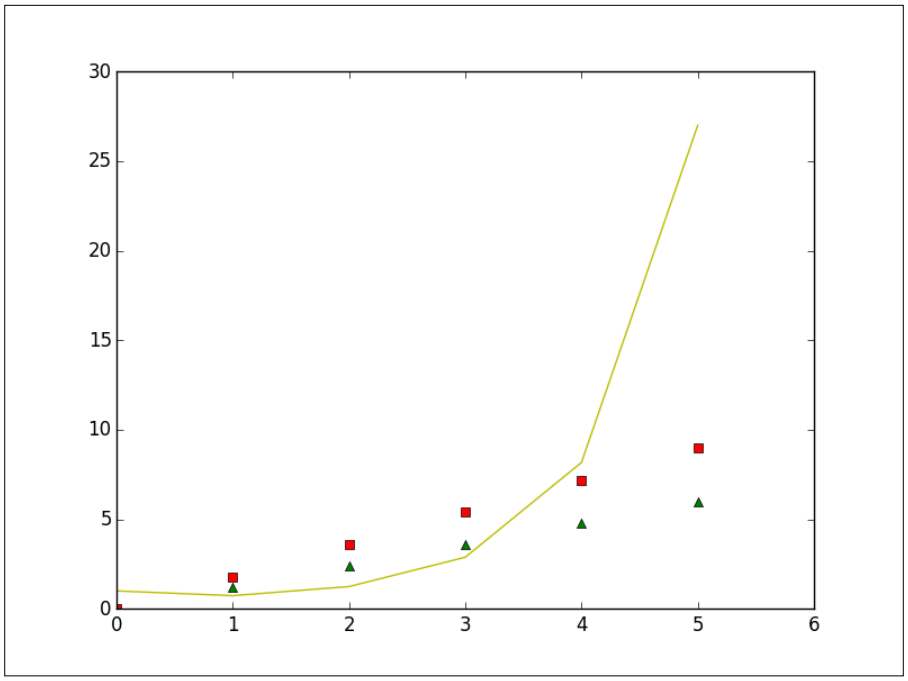
```
>>> plt.axis([0, 6, 0, 10])
```

Line properties

The default line format when we plot data in matplotlib is a solid blue line, which is abbreviated as `b-`. To change this setting, we only need to add the symbol code, which includes letters as color string and symbols as line style string, to the `plot` function. Let us consider a plot of several lines with different format styles:

```
>>> plt.plot(x*2, 'g^', x*3, 'rs', x**x, 'y-')
>>> plt.axis([0, 6, 0, 30])
>>> plt.show()
```

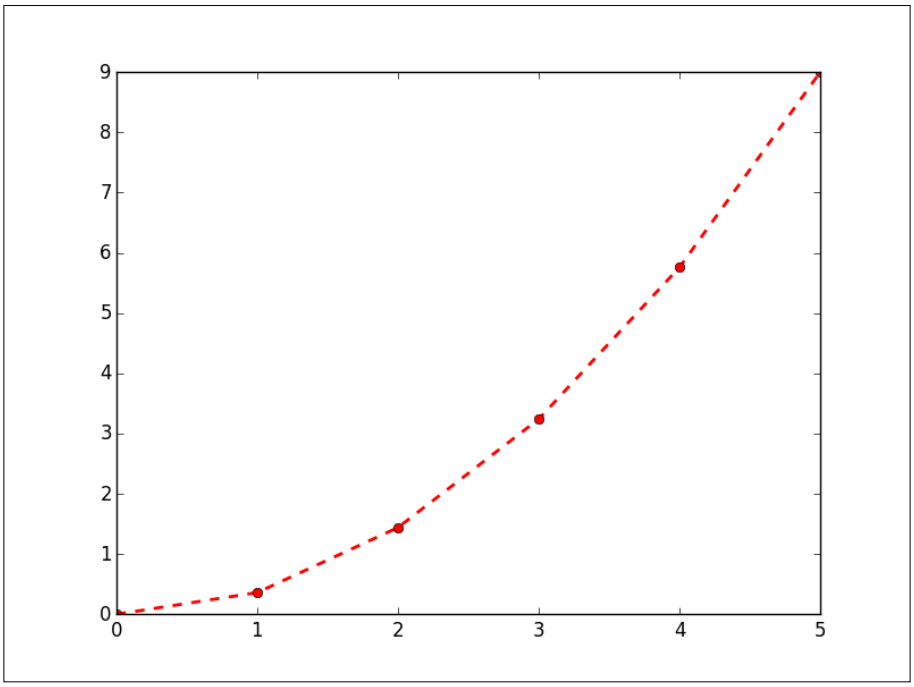
The output for the preceding command is as follows:



There are many line styles and attributes, such as color, line width, and dash style, that we can choose from to control the appearance of our plots. The following example illustrates several ways to set line properties:

```
>>> line = plt.plot(y, color='red', linewidth=2.0)
>>> line.set_linestyle('--')
>>> plt.setp(line, marker='o')
>>> plt.show()
```

The output for the preceding command is as follows:



The following table lists some common properties of the `line2d` plotting:

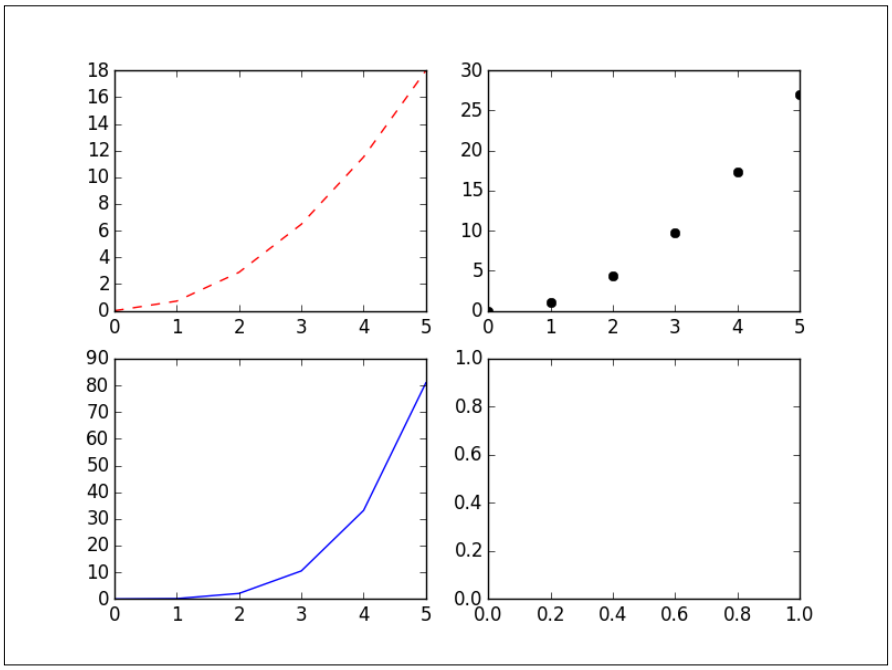
Property	Value type	Description
`color` or `c`	Any matplotlib color	This sets the color of the line in the figure
`dashes`	On/off	This sets the sequence of ink in the points
`data`	`np.array xdata`, `np.array ydata`	This sets the data used for visualization
`linestyle` or `ls`	[`'-'` | `'–'` | `'-.'` | `':'` | ...]	This sets the line style in the figure
`linewidth` or `lw`	Float value in points	This sets the width of line in the figure
`marker`	Any symbol	This sets the style at data points in the figure

Figures and subplots

By default, all plotting commands apply to the current figure and axes. In some situations, we want to visualize data in multiple figures and axes to compare different plots or to use the space on a page more efficiently. There are two steps required before we can plot the data. Firstly, we have to define which figure we want to plot. Secondly, we need to figure out the position of our subplot in the figure:

```
>>> plt.figure('a')    # define a figure, named 'a'
>>> plt.subplot(221)    # the first position of 4 subplots in 2x2 figure
>>> plt.plot(y+y, 'r--')
>>> plt.subplot(222)    # the second position of 4 subplots
>>> plt.plot(y*3, 'ko')
>>> plt.subplot(223)    # the third position of 4 subplots
>>> plt.plot(y*y, 'b^')
>>> plt.subplot(224)
>>> plt.show()
```

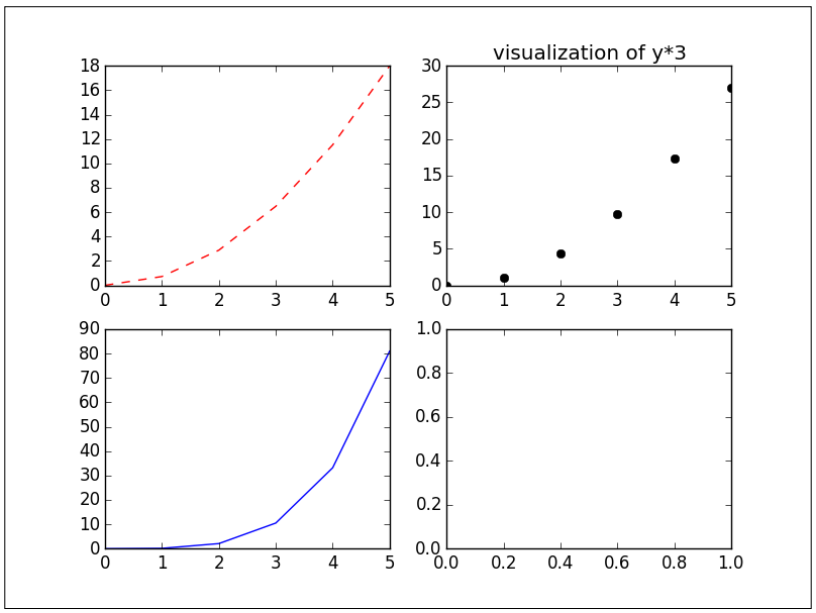
The output for the preceding command is as follows:



In this case, we currently have the figure `a`. If we want to modify any subplot in figure `a`, we first call the command to select the figure and subplot, and then execute the function to modify the subplot. Here, for example, we change the title of the second plot of our four-plot figure:

```
>>> plt.figure('a')
>>> plt.subplot(222)
>>> plt.title('visualization of y*3')
>>> plt.show()
```

The output for the preceding command is as follows:



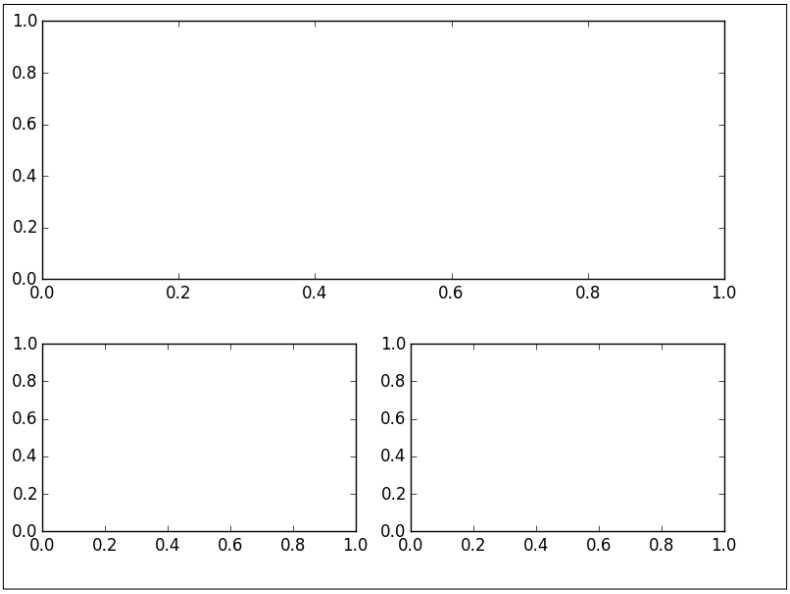
Integer subplot specification must be a three-digit number if we are not using commas to separate indices. So, `plt.subplot(221)` is equal to the `plt.subplot(2,2,1)` command.

There is a convenience method, `plt.subplots()`, to creating a figure that contains a given number of subplots. As in the previous example, we can use the `plt.subplots(2,2)` command to create a `2x2` figure that consists of four subplots.

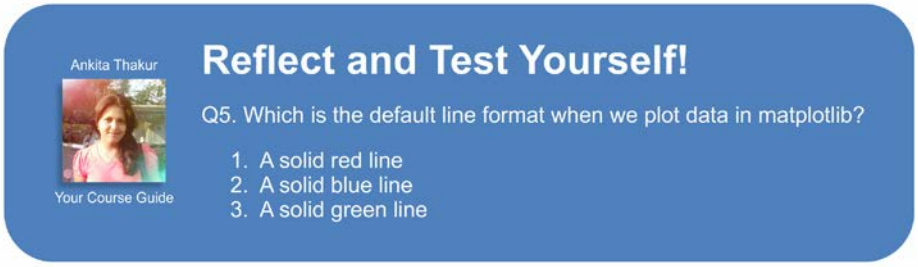
We can also create the axes manually, instead of rectangular grid, by using the `plt.axes([left, bottom, width, height])` command, where all input parameters are in the fractional `[0, 1]` coordinates:

```
>>> plt.figure('b')    # create another figure, named 'b'
>>> ax1 = plt.axes([0.05, 0.1, 0.4, 0.32])
>>> ax2 = plt.axes([0.52, 0.1, 0.4, 0.32])
>>> ax3 = plt.axes([0.05, 0.53, 0.87, 0.44])
>>> plt.show()
```

The output for the preceding command is as follows:



However, when you manually create axes, it takes more time to balance coordinates and sizes between subplots to arrive at a well-proportioned figure.

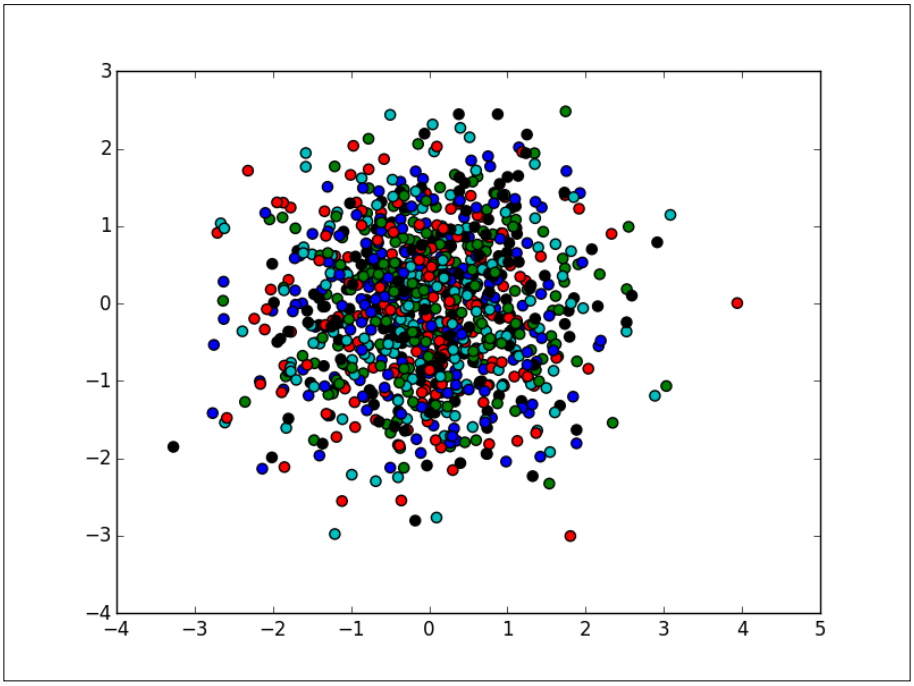


Exploring plot types

We have looked at how to create simple line plots so far. The matplotlib library supports many more plot types that are useful for data visualization. However, our goal is to provide the basic knowledge that will help you to understand and use the library for visualizing data in the most common situations. Therefore, we will only focus on four kinds of plot types: **scatter plots**, **bar plots**, **contour plots**, and **histograms**.

Scatter plots

A scatter plot is used to visualize the relationship between variables measured in the same dataset. It is easy to plot a simple scatter plot, using the `plt.scatter()` function, that requires numeric columns for both the `x` and `y` axis:

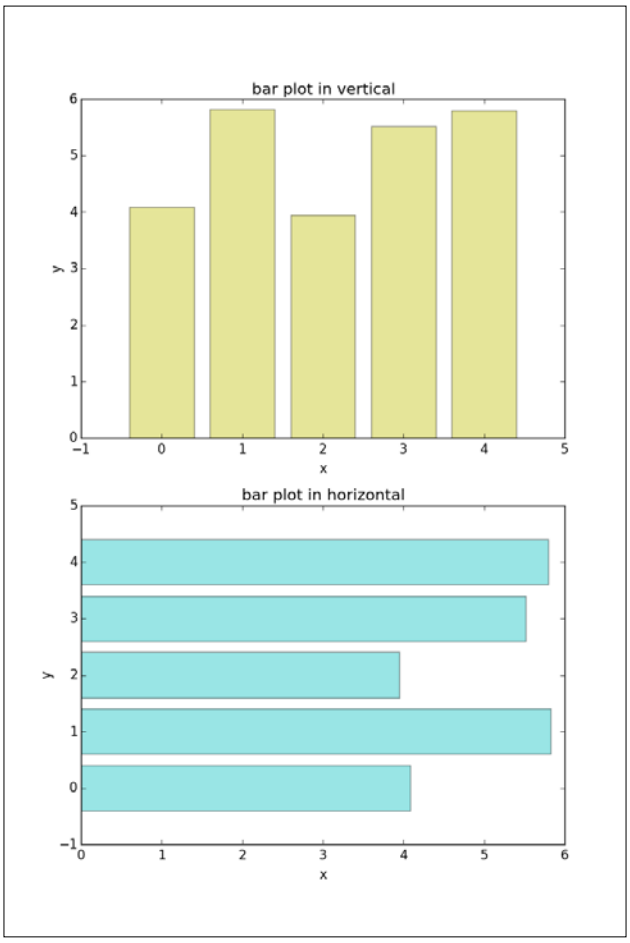


Let's take a look at the command for the preceding output:

```
>>> X = np.random.normal(0, 1, 1000)
>>> Y = np.random.normal(0, 1, 1000)
>>> plt.scatter(X, Y, c = ['b', 'g', 'k', 'r', 'c'])
>>> plt.show()
```

Bar plots

A bar plot is used to present grouped data with rectangular bars, which can be either vertical or horizontal, with the lengths of the bars corresponding to their values. We use the `plt.bar()` command to visualize a vertical bar, and the `plt.barh()` command for the other:



The command for the preceding output is as follows:

```
>>> X = np.arange(5)
>>> Y = 3.14 + 2.71 * np.random.rand(5)
>>> plt.subplots(2)
>>> # the first subplot
>>> plt.subplot(211)
```

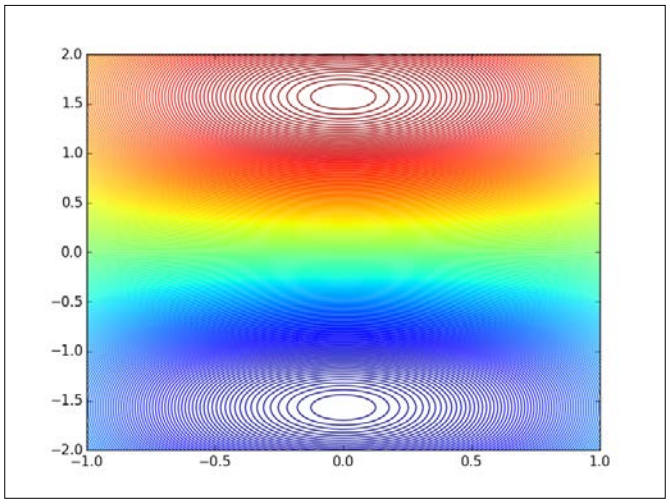
```
>>> plt.bar(X, Y, align='center', alpha=0.4, color='y')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.title('bar plot in vertical')
>>> # the second subplot
>>> plt.subplot(212)
>>> plt.barh(X, Y, align='center', alpha=0.4, color='c')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.title('bar plot in horizontal')
>>> plt.show()
```

Contour plots

We use contour plots to present the relationship between three numeric variables in two dimensions. Two variables are drawn along the `x` and `y` axes, and the third variable, `z`, is used for contour levels that are plotted as curves in different colors:

```
>>> x = np.linspace(-1, 1, 255)
>>> y = np.linspace(-2, 2, 300)
>>> z = np.sin(y[:, np.newaxis]) * np.cos(x)
>>> plt.contour(x, y, z, 255, linewidth=2)
>>> plt.show()
```

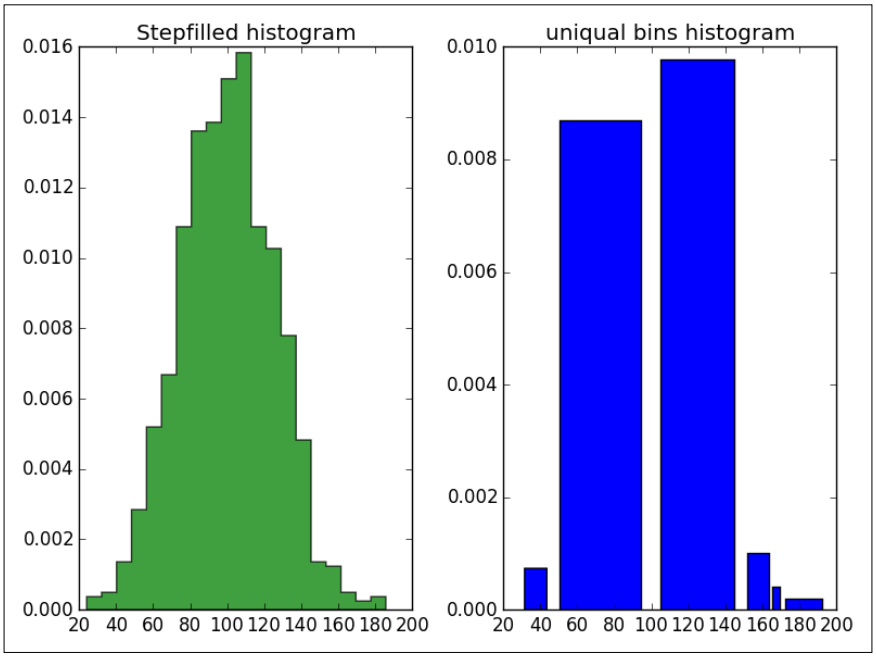
Let's take a look at the contour plot in the following image:



If we want to draw contour lines and filled contours, we can use the `plt.contourf()` method instead of `plt.contour()`. In contrast to MATLAB, matplotlib's `contourf()` will not draw the polygon edges.

Histogram plots

A histogram represents the distribution of numerical data graphically. Usually, the range of values is partitioned into bins of equal size, with the height of each bin corresponding to the frequency of values within that bin:



The command for the preceding output is as follows:

```
>>> mu, sigma = 100, 25
>>> fig, (ax0, ax1) = plt.subplots(ncols=2)
>>> x = mu + sigma * np.random.randn(1000)
>>> ax0.hist(x,20, normed=1, histtype='stepfilled',
               facecolor='g', alpha=0.75)
>>> ax0.set_title('Stepfilled histogram')
```

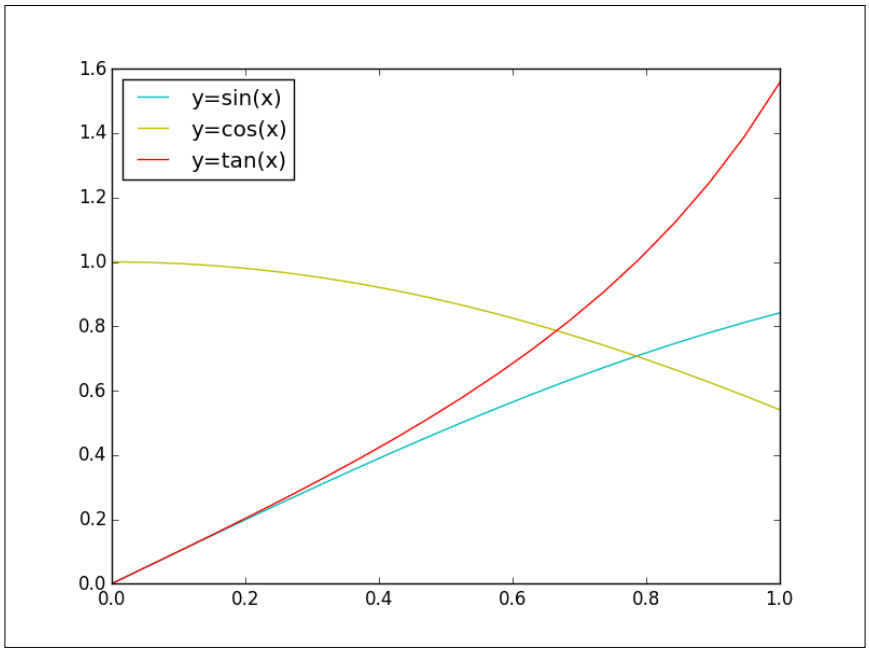
```
>>> ax1.hist(x, bins=[100,150, 165, 170, 195] normed=1,
             histtype='bar', rwidth=0.8)
>>> ax1.set_title('uniquel bins histogram')
>>> # automatically adjust subplot parameters to give specified padding
>>> plt.tight_layout()
>>> plt.show()
```

Legends and annotations

Legends are an important element that is used to identify the `plot` elements in a figure. The easiest way to show a legend inside a figure is to use the `label` argument of the `plot` function, and show the labels by calling the `plt.legend()` method:

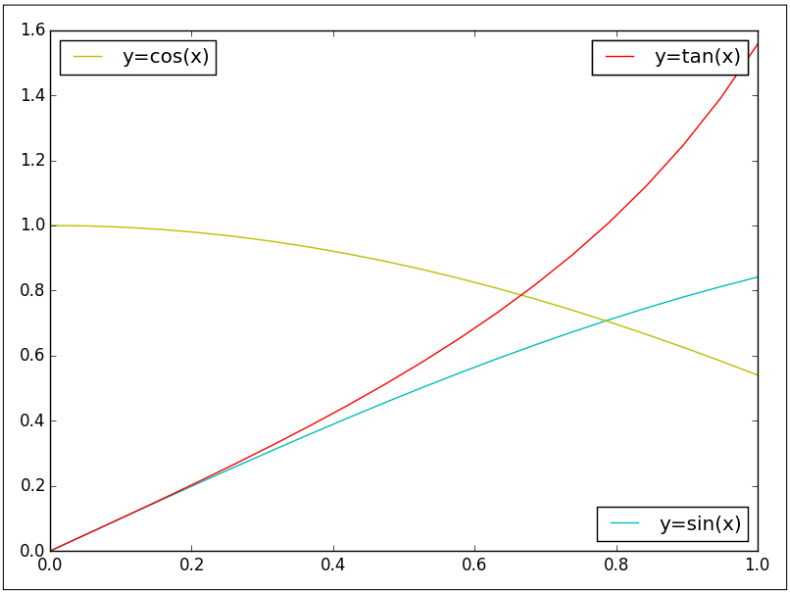
```
>>> x = np.linspace(0, 1, 20)
>>> y1 = np.sin(x)
>>> y2 = np.cos(x)
>>> y3 = np.tan(x)
>>> plt.plot(x, y1, 'c', label='y=sin(x)')
>>> plt.plot(x, y2, 'y', label='y=cos(x)')
>>> plt.plot(x, y3, 'r', label='y=tan(x)')
>>> plt.lengend(loc='upper left')
>>> plt.show()
```

The output for the preceding command as follows:



The `loc` argument in the legend command is used to figure out the position of the label box. There are several valid location options: `lower left`, `right`, `upper left`, `lower center`, `upper right`, `center`, `lower right`, `upper right`, `center right`, `best`, `upper center`, and `center left`. The default position setting is `upper right`. However, when we set an invalid location option that does not exist in the above list, the function automatically falls back to the best option.

If we want to split the legend into multiple boxes in a figure, we can manually set our expected labels for plot lines, as shown in the following image:



The output for the preceding command is as follows:

```
>>> p1 = plt.plot(x, y1, 'c', label='y=sin(x)')
>>> p2 = plt.plot(x, y2, 'y', label='y=cos(x)')
>>> p3 = plt.plot(x, y3, 'r', label='y=tan(x)')
>>> lsin = plt.legend(handles=p1, loc='lower right')
>>> lcos = plt.legend(handles=p2, loc='upper left')
>>> ltan = plt.legend(handles=p3, loc='upper right')
>>> # with above code, only 'y=tan(x)' legend appears in the figure
>>> # fix: add lsin, lcos as separate artists to the axes
>>> plt.gca().add_artist(lsin)
>>> plt.gca().add_artist(lcos)
>>> # automatically adjust subplot parameters to specified padding
>>> plt.tight_layout()
>>> plt.show()
```

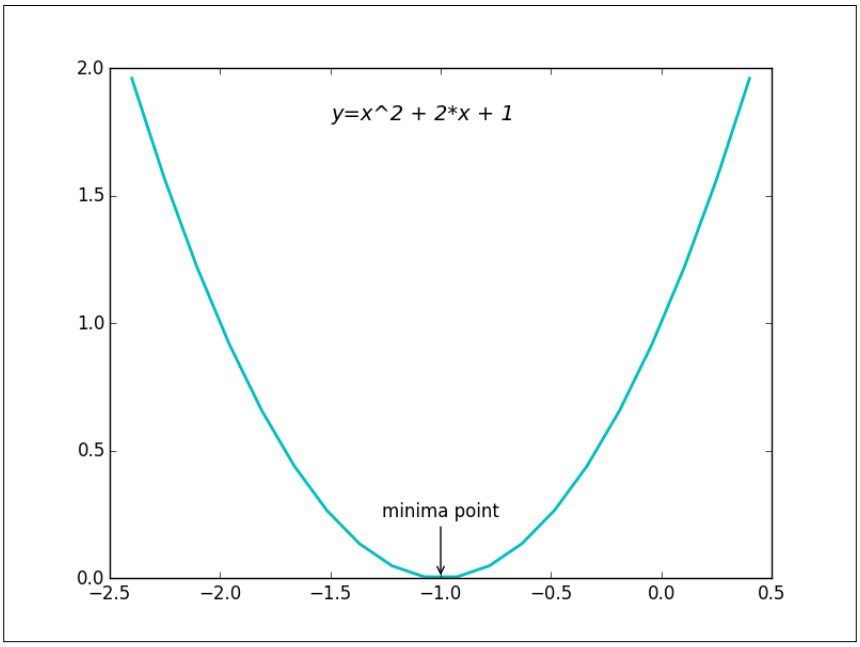
The other element in a figure that we want to introduce is the annotations which can consist of text, arrows, or other shapes to explain parts of the figure in detail, or to emphasize some special data points. There are different methods for showing annotations, such as `text`, `arrow`, and `annotation`.

- The `text` method draws text at the given coordinates `(x, y)` on the plot; optionally with custom properties. There are some common arguments in the function: `x`, `y`, label text, and font-related properties that can be passed in via `fontdict`, such as `family`, `fontsize`, and `style`.
- The `annotate` method can draw both text and arrows arranged appropriately. Arguments of this function are `s` (label text), `xy` (the position of element to annotation), `xytext` (the position of the label `s`), `xycoords` (the string that indicates what type of coordinate `xy` is), and `arrowprops` (the dictionary of line properties for the arrow that connects the annotation).

Here is a simple example to illustrate the `annotate` and `text` functions:

```
>>> x = np.linspace(-2.4, 0.4, 20)
>>> y = x*x + 2*x + 1
>>> plt.plot(x, y, 'c', linewidth=2.0)
>>> plt.text(-1.5, 1.8, 'y=x^2 + 2*x + 1',
             fontsize=14, style='italic')
>>> plt.annotate('minima point', xy=(-1, 0),
                 xytext=(-1, 0.3),
                 horizontalalignment='center',
                 verticalalignment='top',
                 arrowprops=dict(arrowstyle='->',
                 connectionstyle='arc3'))
>>> plt.show()
```

The output for the preceding command is as follows:



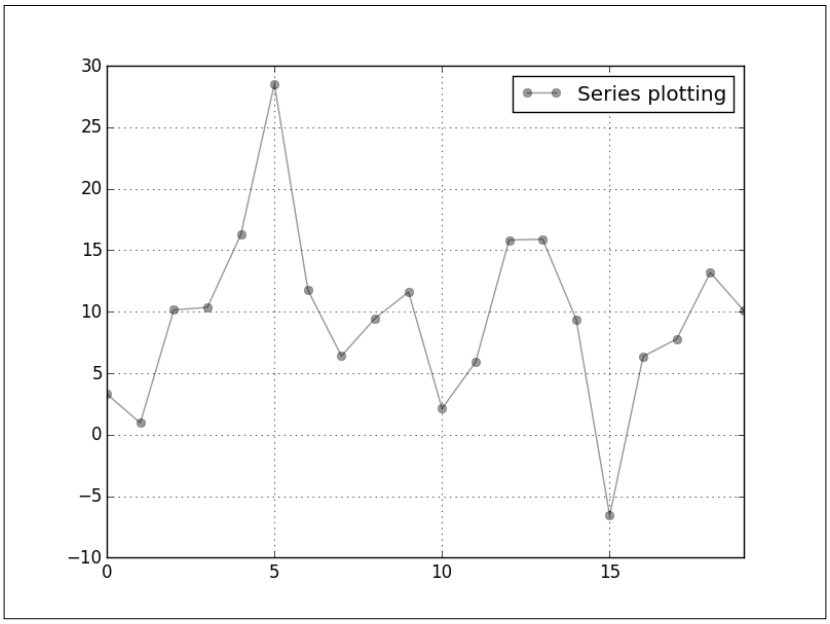
Plotting functions with pandas

We have covered most of the important components in a plot figure using matplotlib. In this section, we will introduce another powerful plotting method for directly creating standard visualization from pandas data objects that are often used to manipulate data.

For Series or DataFrame objects in pandas, most plotting types are supported, such as line, bar, box, histogram, and scatter plots, and pie charts. To select a plot type, we use the `kind` argument of the `plot` function. With no kind of plot specified, the `plot` function will generate a line style visualization by default , as in the following example:

```
>>> s = pd.Series(np.random.normal(10, 8, 20))
>>> s.plot(style='ko—', alpha=0.4, label='Series plotting')
>>> plt.legend()
>>> plt.show()
```

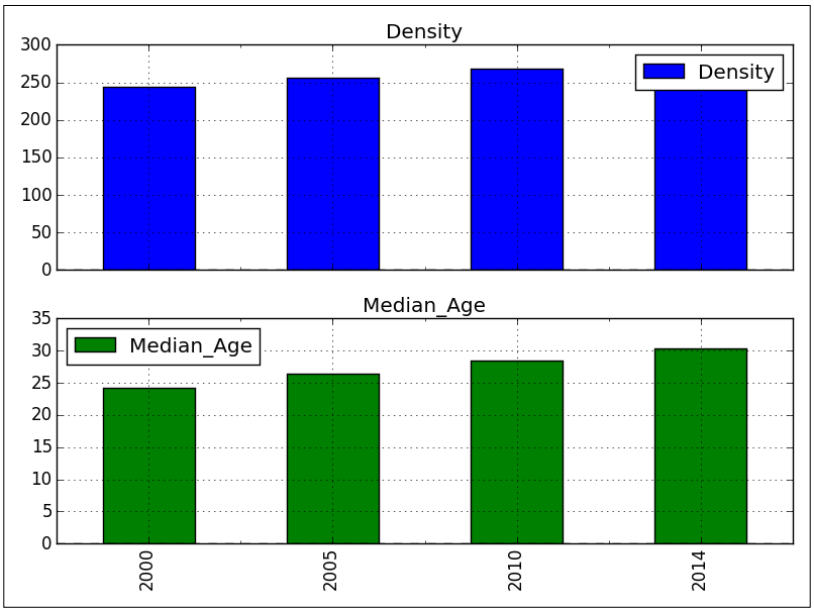
The output for the preceding command is as follows:



Another example will visualize the data of a DataFrame object consisting of multiple columns:

```
>>> data = {'Median_Age': [24.2, 26.4, 28.5, 30.3],
         'Density': [244, 256, 268, 279]}
>>> index_label = ['2000', '2005', '2010', '2014'];
>>> df1 = pd.DataFrame(data, index=index_label)
>>> df1.plot(kind='bar', subplots=True, sharex=True)
>>> plt.tight_layout();
>>> plt.show()
```

The output for the preceding command is as follows:



The plot method of the DataFrame has a number of options that allow us to handle the plotting of the columns. For example, in the preceding DataFrame visualization, we chose to plot the columns in separate subplots. The following table lists more options:

Argument	**Value**	**Description**
`subplots`	`True/False`	The plots each data column in a separate subplot
`logy`	`True/False`	The gets a log-scale `y` axis
`secondary_y`	`True/False`	The plots data on a secondary `y` axis
`sharex, sharey`	`True/False`	The shares the same `x` or `y` axis, linking sticks and limits

Additional Python data visualization tools

Besides matplotlib, there are other powerful data visualization toolkits based on Python. While we cannot dive deeper into these libraries, we would like to at least briefly introduce them in this session.

Bokeh

Bokeh is a project by Peter Wang, Hugo Shi, and others at Continuum Analytics. It aims to provide elegant and engaging visualizations in the style of `D3.js`. The library can quickly and easily create interactive plots, dashboards, and data applications. Here are a few differences between matplotlib and Bokeh:

- Bokeh achieves cross-platform ubiquity through IPython's new model of in-browser client-side rendering
- Bokeh uses a syntax familiar to R and ggplot users, while matplotlib is more familiar to Matlab users
- Bokeh has a coherent vision to build a ggplot-inspired in-browser interactive visualization tool, while Matplotlib has a coherent vision of focusing on 2D cross-platform graphics.

The basic steps for creating plots with Bokeh are as follows:

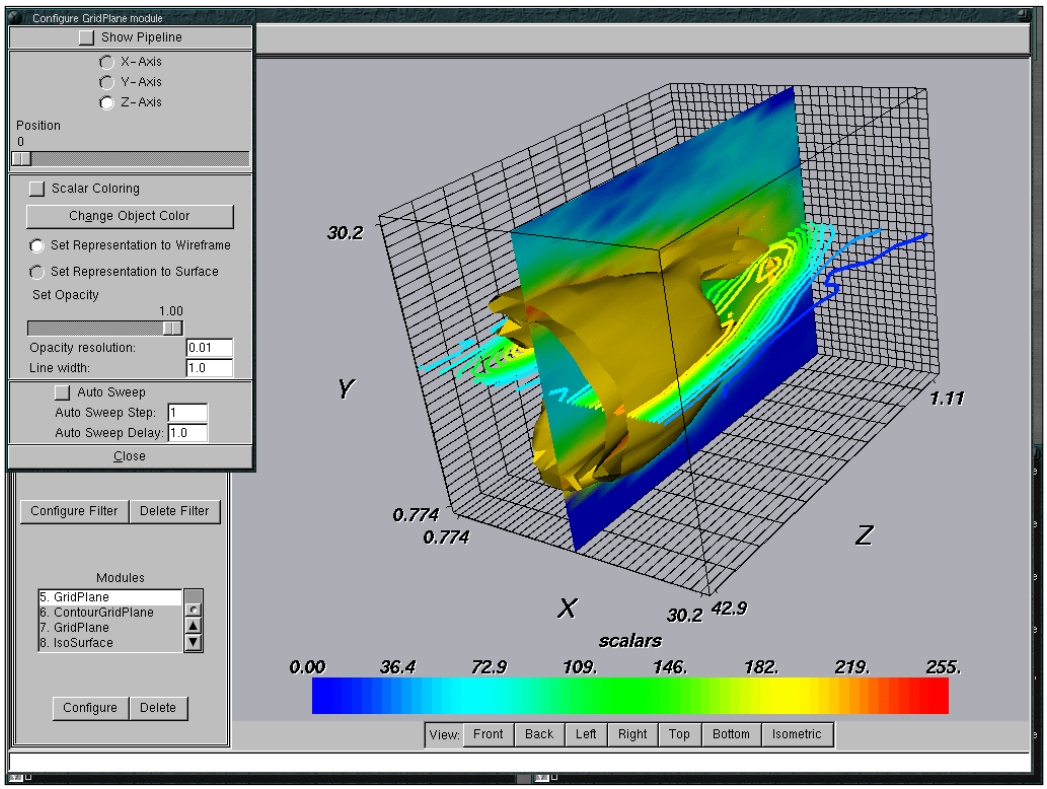
- Prepare some data in a list, series, and Dataframe
- Tell Bokeh where you want to generate the output
- Call `figure()` to create a plot with some overall options, similar to the matplotlib options discussed earlier
- Add renderers for your data, with visual customizations such as colors, legends, and width
- Ask Bokeh to `show()` or `save()` the results

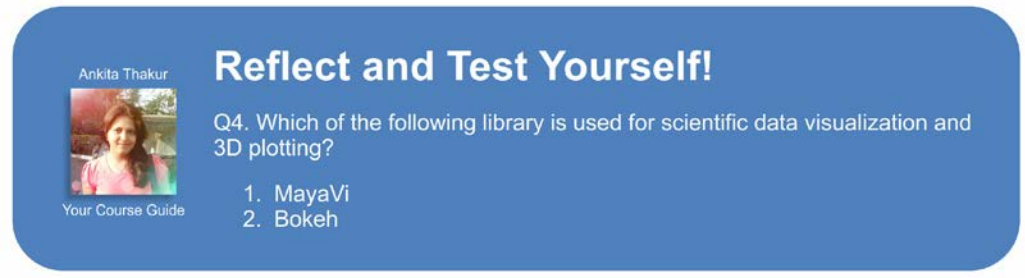
MayaVi

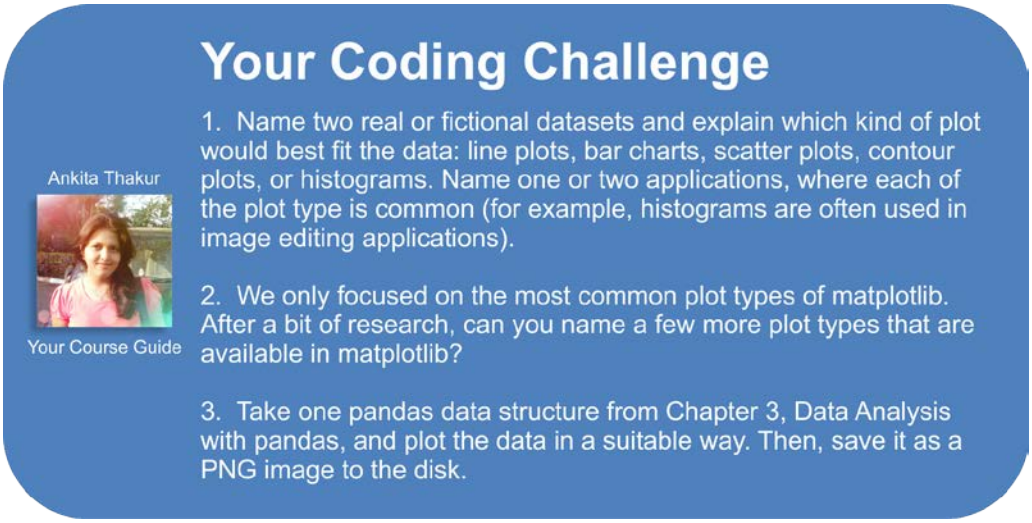
MayaVi is a library for interactive scientific data visualization and 3D plotting, built on top of the award-winning **visualization toolkit** (**VTK**), which is a traits-based wrapper for the open-source visualization library. It offers the following:

- The possibility to interact with the data and object in the visualization through dialogs.
- An interface in Python for scripting. MayaVi can work with Numpy and scipy for 3D plotting out of the box and can be used within IPython notebooks, which is similar to matplotlib.
- An abstraction over VTK that offers a simpler programming model.

Let's view an illustration made entirely using MayaVi based on VTK examples and their provided data:




Ankita Thakur
Your Course Guide
Reflect and Test Yourself!
Q4. Which of the following library is used for scientific data visualization and 3D plotting?
1. MayaVi
2. Bokeh


Ankita Thakur
Your Course Guide
Your Coding Challenge
1. Name two real or fictional datasets and explain which kind of plot would best fit the data: line plots, bar charts, scatter plots, contour plots, or histograms. Name one or two applications, where each of the plot type is common (for example, histograms are often used in image editing applications).
2. We only focused on the most common plot types of matplotlib. After a bit of research, can you name a few more plot types that are available in matplotlib?
3. Take one pandas data structure from Chapter 3, Data Analysis with pandas, and plot the data in a suitable way. Then, save it as a PNG image to the disk.

Summary of Module 2 Chapter 4

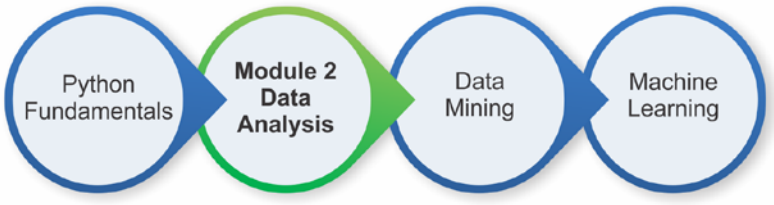
Ankita Thakur

Your Course Guide

We finished covering most of the basics, such as functions, arguments, and properties for data visualization, based on the matplotlib library. We hope that, through the examples, you will be able to understand and apply them to your own problems. In general, to visualize data, we need to consider five steps- that is, getting data into suitable Python or pandas data structures, such as lists, dictionaries, Series, or DataFrames. We explained in the previous chapters, how to accomplish this step. The second step is defining plots and subplots for the data object in question. We discussed this in the figures and subplots session. The third step is selecting a plot style and its attributes to show in the subplots such as: line, bar, histogram, scatter plot, line style, and color. The fourth step is adding extra components to the subplots, like legends, annotations and text. The fifth step is displaying or saving the results.

By now, you can do quite a few things with a dataset; for example, manipulation, cleaning, exploration, and visualization based on Python libraries such as Numpy, pandas, and matplotlib. You can now combine this knowledge and practice with these libraries to get more and more familiar with Python data analysis.

Your Progress through the Course So Far



5
Time Series

Time series typically consist of a sequence of data points coming from measurements taken over time. This kind of data is very common and occurs in a multitude of fields.

A business executive is interested in stock prices, prices of goods and services or monthly sales figures. A meteorologist takes temperature measurements several times a day and also keeps records of precipitation, humidity, wind direction and force. A neurologist can use electroencephalography to measure electrical activity of the brain along the scalp. A sociologist can use campaign contribution data to learn about political parties and their supporters and use these insights as an argumentation aid. More examples for time series data can be enumerated almost endlessly.

Time series primer

In general, time series serve two purposes. First, they help us to learn about the underlying process that generated the data. On the other hand, we would like to be able to forecast future values of the same or related series using existing data. When we measure temperature, precipitation or wind, we would like to learn more about more complex things, such as weather or the climate of a region and how various factors interact. At the same time, we might be interested in weather forecasting.

In this chapter we will explore the time series capabilities of pandas. Apart from its powerful core data structures – the series and the DataFrame – pandas comes with helper functions for dealing with time related data. With its extensive built-in optimizations, pandas is capable of handling large time series with millions of data points with ease.

We will gradually approach time series, starting with the basic building blocks of date and time objects.

Working with date and time objects

Python supports date and time handling in the `datetime` and time modules from the standard library:

```
>>> import datetime
>>> datetime.datetime(2000, 1, 1)
datetime.datetime(2000, 1, 1, 0, 0)
```

Sometimes, dates are given or expected as strings, so a conversion from or to strings is necessary, which is realized by two functions: `strptime` and `strftime`, respectively:

```
>>> datetime.datetime.strptime("2000/1/1", "%Y/%m/%d")
datetime.datetime(2000, 1, 1, 0, 0)
>>> datetime.datetime(2000, 1, 1, 0, 0).strftime("%Y%m%d")
'20000101'
```

Real-world data usually comes in all kinds of shapes and it would be great if we did not need to remember the exact date format specifies for parsing. Thankfully, pandas abstracts away a lot of the friction, when dealing with strings representing dates or time. One of these helper functions is `to_datetime`:

```
>>> import pandas as pd
>>> import numpy as np
>>> pd.to_datetime("4th of July")
Timestamp('2015-07-04 00:00:00')
>>> pd.to_datetime("13.01.2000")
Timestamp('2000-01-13 00:00:00')
>>> pd.to_datetime("7/8/2000")
Timestamp('2000-07-08 00:00:00')
```

The last can refer to August 7th or July 8th, depending on the region. To disambiguate this case, `to_datetime` can be passed a keyword argument `dayfirst`:

```
>>> pd.to_datetime("7/8/2000", dayfirst=True)
Timestamp('2000-08-07 00:00:00')
```

Timestamp objects can be seen as pandas' version of `datetime` objects and indeed, the `Timestamp` class is a subclass of `datetime`:

```
>>> issubclass(pd.Timestamp, datetime.datetime)
True
```

Which means they can be used interchangeably in many cases:

```
>>> ts = pd.to_datetime(946684800000000000)
>>> ts.year, ts.month, ts.day, ts.weekday()
(2000, 1, 1, 5)
```

Timestamp objects are an important part of time series capabilities of pandas, since timestamps are the building block of `DateTimeIndex` objects:

```
>>> index = [pd.Timestamp("2000-01-01"),
             pd.Timestamp("2000-01-02"),
             pd.Timestamp("2000-01-03")]
>>> ts = pd.Series(np.random.randn(len(index)), index=index)
>>> ts
2000-01-01    0.731897
2000-01-02    0.761540
2000-01-03   -1.316866
dtype: float64
>>> ts.indexDatetime
Index(['2000-01-01', '2000-01-02', '2000-01-03'],
dtype='datetime64[ns]', freq=None, tz=None)
```

There are a few things to note here: We create a list of timestamp objects and pass it to the series constructor as index. This list of timestamps gets converted into a `DatetimeIndex` on the fly. If we had passed only the date strings, we would not get a `DatetimeIndex`, just an `index`:

```
>>> ts = pd.Series(np.random.randn(len(index)), index=[
              "2000-01-01", "2000-01-02", "2000-01-03"])
>>> ts.index
Index([u'2000-01-01', u'2000-01-02', u'2000-01-03'], dtype='object')
```

However, the `to_datetime` function is flexible enough to be of help, if all we have is a list of date strings:

```
>>> index = pd.to_datetime(["2000-01-01", "2000-01-02", "2000-01-03"])
>>> ts = pd.Series(np.random.randn(len(index)), index=index)
>>> ts.index
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03'],
dtype='datetime64[ns]', freq=None, tz=None))
```

Another thing to note is that while we have a `DatetimeIndex`, the `freq` and `tz` attributes are both `None`. We will learn about the utility of both attributes later in this chapter.

With `to_datetime` we are able to convert a variety of strings and even lists of strings into timestamp or `DatetimeIndex` objects. Sometimes we are not explicitly given all the information about a series and we have to generate sequences of time stamps of fixed intervals ourselves.

pandas offer another great utility function for this task: `date_range`.

The `date_range` function helps to generate a fixed frequency `datetime` index between start and end dates. It is also possible to specify either the start or end date and the number of timestamps to generate.

The frequency can be specified by the `freq` parameter, which supports a number of offsets. You can use typical time intervals like hours, minutes, and seconds:

```
>>> pd.date_range(start="2000-01-01", periods=3, freq='H')
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:00:00',
  '2000-01-01 02:00:00'], dtype='datetime64[ns]', freq='H', tz=None)
>>> pd.date_range(start="2000-01-01", periods=3, freq='T')
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 00:01:00',
  '2000-01-01 00:02:00'], dtype='datetime64[ns]', freq='T', tz=None)

>>> pd.date_range(start="2000-01-01", periods=3, freq='S')
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 00:00:01',
  '2000-01-01 00:00:02'], dtype='datetime64[ns]', freq='S', tz=None)
```

The `freq` attribute allows us to specify a multitude of options. pandas has been used successfully in finance and economics, not least because it is really simple to work with business dates as well. As an example, to get an index with the first three business days of the millennium, the `B` offset alias can be used:

```
>>> pd.date_range(start="2000-01-01", periods=3, freq='B')
DatetimeIndex(['2000-01-03', '2000-01-04', '2000-01-05'],
dtype='datetime64[ns]', freq='B', tz=None)
```

The following table shows the available offset aliases and can be also be looked up in the pandas documentation on time series under `http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases`:

Alias	Description
B	Business day frequency
C	Custom business day frequency
D	Calendar day frequency
W	Weekly frequency
M	Month end frequency

Alias	Description
BM	Business month end frequency
CBM	Custom business month end frequency
MS	Month start frequency
BMS	Business month start frequency
CBMS	Custom business month start frequency
Q	Quarter end frequency
BQ	Business quarter frequency
QS	Quarter start frequency
BQS	Business quarter start frequency
A	Year end frequency
BA	Business year end frequency
AS	Year start frequency
BAS	Business year start frequency
BH	Business hour frequency
H	Hourly frequency
T	Minutely frequency
S	Secondly frequency
L	Milliseconds
U	Microseconds
N	Nanoseconds

Moreover, The offset aliases can be used in combination as well. Here, we are generating a `datetime` index with five elements, each one day, one hour, one minute and ten seconds apart:

```
>>> pd.date_range(start="2000-01-01", periods=5, freq='1D1h1min10s')
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-02 01:01:10',
               '2000-01-03 02:02:20', '2000-01-04 03:03:30',
               '2000-01-05 04:04:40'],
              dtype='datetime64[ns]', freq='90070S', tz=None)
```

If we want to index data every 12 hours of our business time, which by default starts at 9 AM and ends at 5 PM, we would simply prefix the `BH` alias:

```
>>> pd.date_range(start="2000-01-01", periods=5, freq='12BH')
DatetimeIndex(['2000-01-03 09:00:00', '2000-01-04 13:00:00',
               '2000-01-06 09:00:00', '2000-01-07 13:00:00',
               '2000-01-11 09:00:00'],
              dtype='datetime64[ns]', freq='12BH', tz=None)
```

A custom definition of what a business hour means is also possible:

```
>>> ts.index
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03'],
dtype='datetime64[ns]', freq=None, tz=None)
```

We can use this custom business hour to build indexes as well:

```
>>> pd.date_range(start="2000-01-01", periods=5, freq=12 * bh)
DatetimeIndex(['2000-01-03 07:00:00', '2000-01-03 19:00:00',
               '2000-01-04 07:00:00', '2000-01-04 19:00:00',
               '2000-01-05 07:00:00', '2000-01-05 19:00:00',
               '2000-01-06 07:00:00'],
              dtype='datetime64[ns]', freq='12BH', tz=None)
```

Some frequencies allow us to specify an anchoring suffix, which allows us to express intervals, such as every Friday or every second Tuesday of the month:

```
>>> pd.date_range(start="2000-01-01", periods=5, freq='W-FRI')
DatetimeIndex(['2000-01-07', '2000-01-14', '2000-01-21', '2000-01-28',
'2000-02-04'], dtype='datetime64[ns]', freq='W-FRI', tz=None)
>>> pd.date_range(start="2000-01-01", periods=5, freq='WOM-2TUE')
DatetimeIndex(['2000-01-11', '2000-02-08', '2000-03-14', '2000-04-11',
'2000-05-09'], dtype='datetime64[ns]', freq='WOM-2TUE', tz=None)
```

Finally, we can merge various indexes of different frequencies. The possibilities are endless. We only show one example, where we combine two indexes – each over a decade – one pointing to every first business day of a year and one to the last day of February:

```
>>> s = pd.date_range(start="2000-01-01", periods=10, freq='BAS-JAN')
>>> t = pd.date_range(start="2000-01-01", periods=10, freq='A-FEB')
>>> s.union(t)
DatetimeIndex(['2000-01-03', '2000-02-29', '2001-01-01', '2001-02-28',
               '2002-01-01', '2002-02-28', '2003-01-01', '2003-02-28',
               '2004-01-01', '2004-02-29', '2005-01-03', '2005-02-28',
               '2006-01-02', '2006-02-28', '2007-01-01', '2007-02-28',
               '2008-01-01', '2008-02-29', '2009-01-01', '2009-02-28'],
              dtype='datetime64[ns]', freq=None, tz=None)
```

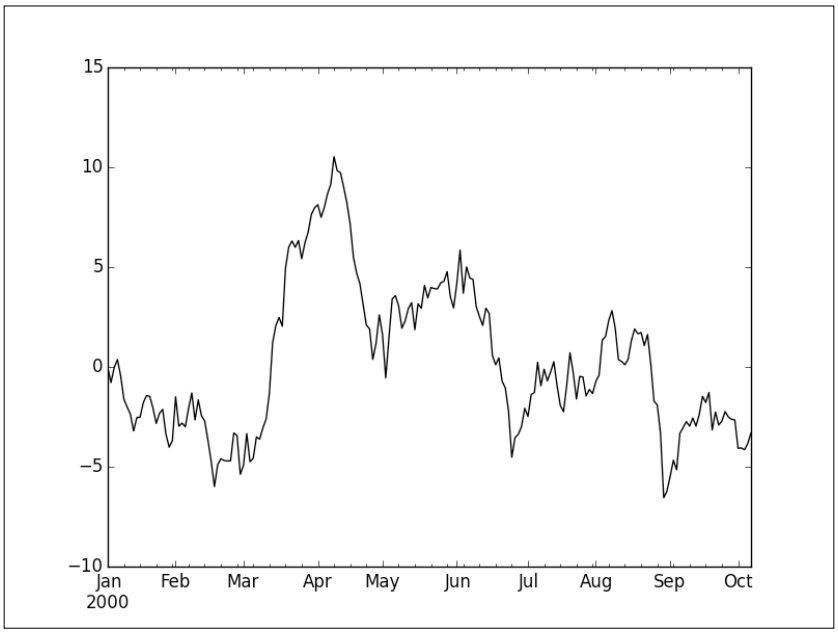
We see, that 2000 and 2005 did not start on a weekday and that 2000, 2004, and 2008 were the leap years.

We have seen two powerful functions so far, `to_datetime` and `date_range`. Now we want to dive into time series by first showing how you can create and plot time series data with only a few lines. In the rest of this section, we will show various ways to access and slice time series data.

It is easy to get started with time series data in pandas. A random walk can be created and plotted in a few lines:

```
>>> index = pd.date_range(start='2000-01-01', periods=200, freq='B')
>>> ts = pd.Series(np.random.randn(len(index)), index=index)
>>> walk = ts.cumsum()
>>> walk.plot()
```

A possible output of this plot is show in the following figure:



Just as with usual series objects, you can select parts and slice the index:

```
>>> ts.head()
2000-01-03    1.464142
2000-01-04    0.103077
2000-01-05    0.762656
2000-01-06    1.157041
2000-01-07   -0.427284
```

```
Freq: B, dtype: float64
>>> ts[0]
1.4641415817112928
>>> ts[1:3]
2000-01-04    0.103077
2000-01-05    0.762656
```

We can use date strings as keys, even though our series has a `DatetimeIndex`:

```
>>> ts['2000-01-03']
1.4641415817112928
```

Even though the `DatetimeIndex` is made of timestamp objects, we can use `datetime` objects as keys as well:

```
>>> ts[datetime.datetime(2000, 1, 3)]
1.4641415817112928
```

Access is similar to lookup in dictionaries or lists, but more powerful. We can, for example, slice with strings or even mixed objects:

```
>>> ts['2000-01-03':'2000-01-05']
2000-01-03    1.464142
2000-01-04    0.103077
2000-01-05    0.762656
Freq: B, dtype: float64
>>> ts['2000-01-03':datetime.datetime(2000, 1, 5)]
2000-01-03    1.464142
2000-01-04    0.103077
2000-01-05    0.762656
Freq: B, dtype: float64
>>> ts['2000-01-03':datetime.date(2000, 1, 5)]
2000-01-03   -0.807669
2000-01-04    0.029802
2000-01-05   -0.434855
Freq: B, dtype: float64
```

It is even possible to use partial strings to select groups of entries. If we are only interested in February, we could simply write:

```
>>> ts['2000-02']
2000-02-01    0.277544
2000-02-02   -0.844352
2000-02-03   -1.900688
```

```
2000-02-04   -0.120010
2000-02-07   -0.465916
2000-02-08   -0.575722
2000-02-09    0.426153
2000-02-10    0.720124
2000-02-11    0.213050
2000-02-14   -0.604096
2000-02-15   -1.275345
2000-02-16   -0.708486
2000-02-17   -0.262574
2000-02-18    1.898234
2000-02-21    0.772746
2000-02-22    1.142317
2000-02-23   -1.461767
2000-02-24   -2.746059
2000-02-25   -0.608201
2000-02-28    0.513832
2000-02-29   -0.132000
```

To see all entries from March until May, including:

```
>>> ts['2000-03':'2000-05']
2000-03-01    0.528070
2000-03-02    0.200661
                   ...
2000-05-30    1.206963
2000-05-31    0.230351
Freq: B, dtype: float64
```

Time series can be shifted forward or backward in time. The index stays in place, the values move:

```
>>> small_ts = ts['2000-02-01':'2000-02-05']
>>> small_ts
2000-02-01    0.277544
2000-02-02   -0.844352
2000-02-03   -1.900688
2000-02-04   -0.120010
Freq: B, dtype: float64
>>> small_ts.shift(2)
2000-02-01         NaN
```

```
2000-02-02         NaN
2000-02-03    0.277544
2000-02-04   -0.844352
Freq: B, dtype: float64
```

To shift backwards in time, we simply use negative values:

```
>>> small_ts.shift(-2)
2000-02-01   -1.900688
2000-02-02   -0.120010
2000-02-03         NaN
2000-02-04         NaN
Freq: B, dtype: float64
```

Resampling time series

Resampling describes the process of frequency conversion over time series data. It is a helpful technique in various circumstances as it fosters understanding by grouping together and aggregating data. It is possible to create a new time series from daily temperature data that shows the average temperature per week or month. On the other hand, real-world data may not be taken in uniform intervals and it is required to map observations into uniform intervals or to fill in missing values for certain points in time. These are two of the main use directions of resampling: binning and aggregation, and filling in missing data. Downsampling and upsampling occur in other fields as well, such as digital signal processing. There, the process of downsampling is often called decimation and performs a reduction of the sample rate. The inverse process is called **interpolation**, where the sample rate is increased. We will look at both directions from a data analysis angle.

Downsampling time series data

Downsampling reduces the number of samples in the data. During this reduction, we are able to apply aggregations over data points. Let's imagine a busy airport with thousands of people passing through every hour. The airport administration has installed a visitor counter in the main area, to get an impression of exactly how busy their airport is.

They are receiving data from the counter device every minute. Here are the hypothetical measurements for a day, beginning at 08:00, ending 600 minutes later at 18:00:

```
>>> rng = pd.date_range('4/29/2015 8:00', periods=600, freq='T')
>>> ts = pd.Series(np.random.randint(0, 100, len(rng)), index=rng)
>>> ts.head()
2015-04-29 08:00:00     9
2015-04-29 08:01:00    60
2015-04-29 08:02:00    65
2015-04-29 08:03:00    25
2015-04-29 08:04:00    19
```

To get a better picture of the day, we can downsample this time series to larger intervals, for example, 10 minutes. We can choose an aggregation function as well. The default aggregation is to take all the values and calculate the mean:

```
>>> ts.resample('10min').head()
2015-04-29 08:00:00    49.1
2015-04-29 08:10:00    56.0
2015-04-29 08:20:00    42.0
2015-04-29 08:30:00    51.9
2015-04-29 08:40:00    59.0
Freq: 10T, dtype: float64
```

In our airport example, we are also interested in the sum of the values, that is, the combined number of visitors for a given time frame. We can choose the aggregation function by passing a function or a function name to the `how` parameter works:

```
>>> ts.resample('10min', how='sum').head()
2015-04-29 08:00:00    442
2015-04-29 08:10:00    409
2015-04-29 08:20:00    532
2015-04-29 08:30:00    433
2015-04-29 08:40:00    470
Freq: 10T, dtype: int64
```

Or we can reduce the sampling interval even more by resampling to an hourly interval:

```
>>> ts.resample('1h', how='sum').head()
2015-04-29 08:00:00    2745
2015-04-29 09:00:00    2897
2015-04-29 10:00:00    3088
2015-04-29 11:00:00    2616
2015-04-29 12:00:00    2691
Freq: H, dtype: int64
```

We can ask for other things as well. For example, what was the maximum number of people that passed through our airport within one hour:

```
>>> ts.resample('1h', how='max').head()
2015-04-29 08:00:00    97
2015-04-29 09:00:00    98
2015-04-29 10:00:00    99
2015-04-29 11:00:00    98
2015-04-29 12:00:00    99
Freq: H, dtype: int64
```

Or we can define a custom function if we are interested in more unusual metrics. For example, we could be interested in selecting a random sample for each hour:

```
>>> import random
>>> ts.resample('1h', how=lambda m: random.choice(m)).head()
2015-04-29 08:00:00    28
2015-04-29 09:00:00    14
2015-04-29 10:00:00    68
2015-04-29 11:00:00    31
2015-04-29 12:00:00     5
```

If you specify a function by string, pandas uses highly optimized versions.

The built-in functions that can be used as argument to `how` are: `sum`, `mean`, `std`, `sem`, `max`, `min`, `median`, `first`, `last`, `ohlc`. The `ohlc` metric is popular in finance. It stands for open-high-low-close. An OHLC chart is a typical way to illustrate movements in the price of a financial instrument over time.

While in our airport this metric might not be that valuable, we can compute it nonetheless:

```
>>> ts.resample('1h', how='ohlc').head()
                     open  high  low  close
2015-04-29 08:00:00     9    97    0     14
2015-04-29 09:00:00    68    98    3     12
2015-04-29 10:00:00    71    99    1      1
2015-04-29 11:00:00    59    98    0      4
2015-04-29 12:00:00    56    99    3     55
```

Upsampling time series data

In upsampling, the frequency of the time series is increased. As a result, we have more sample points than data points. One of the main questions is how to account for the entries in the series where we have no measurement.

Let's start with hourly data for a single day:

```
>>> rng = pd.date_range('4/29/2015 8:00', periods=10, freq='H')
>>> ts = pd.Series(np.random.randint(0, 100, len(rng)), index=rng)
>>> ts.head()
2015-04-29 08:00:00    30
2015-04-29 09:00:00    27
2015-04-29 10:00:00    54
2015-04-29 11:00:00     9
2015-04-29 12:00:00    48
Freq: H, dtype: int64
```

If we upsample to data points taken every 15 minutes, our time series will be extended with `NaN` values:

```
>>> ts.resample('15min')
>>> ts.head()
2015-04-29 08:00:00    30
2015-04-29 08:15:00   NaN
2015-04-29 08:30:00   NaN
2015-04-29 08:45:00   NaN
2015-04-29 09:00:00    27
```

There are various ways to deal with missing values, which can be controlled by the `fill_method` keyword argument to resample. Values can be filled either forward or backward:

```
>>> ts.resample('15min', fill_method='ffill').head()
2015-04-29 08:00:00    30
2015-04-29 08:15:00    30
2015-04-29 08:30:00    30
2015-04-29 08:45:00    30
2015-04-29 09:00:00    27
Freq: 15T, dtype: int64
>>> ts.resample('15min', fill_method='bfill').head()
2015-04-29 08:00:00    30
2015-04-29 08:15:00    27
2015-04-29 08:30:00    27
2015-04-29 08:45:00    27
2015-04-29 09:00:00    27
```

With the `limit` parameter, it is possible to control the number of missing values to be filled:

```
>>> ts.resample('15min', fill_method='ffill', limit=2).head()
2015-04-29 08:00:00    30
2015-04-29 08:15:00    30
2015-04-29 08:30:00    30
2015-04-29 08:45:00   NaN
2015-04-29 09:00:00    27
Freq: 15T, dtype: float64
```

If you want to adjust the labels during resampling, you can use the `loffset` keyword argument:

```
>>> ts.resample('15min', fill_method='ffill', limit=2, loffset='5min').
head()
2015-04-29 08:05:00    30
2015-04-29 08:20:00    30
2015-04-29 08:35:00    30
2015-04-29 08:50:00   NaN
2015-04-29 09:05:00    27
Freq: 15T, dtype: float64
```

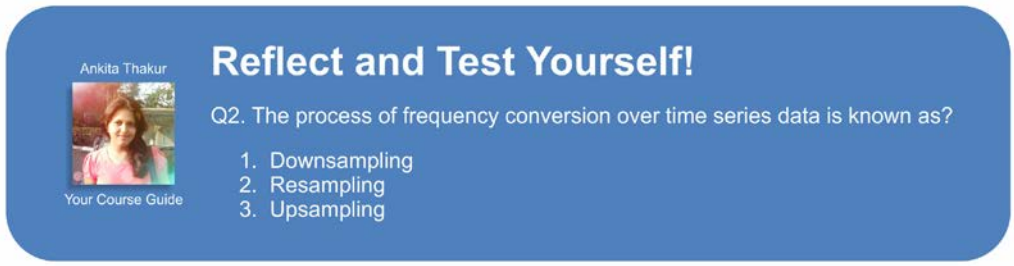
There is another way to fill in missing values. We could employ an algorithm to construct new data points that would somehow fit the existing points, for some definition of somehow. This process is called interpolation.

We can ask pandas to interpolate a time series for us:

```
>>> tsx = ts.resample('15min')
>>> tsx.interpolate().head()
2015-04-29 08:00:00    30.00
2015-04-29 08:15:00    29.25
2015-04-29 08:30:00    28.50
2015-04-29 08:45:00    27.75
2015-04-29 09:00:00    27.00
Freq: 15T, dtype: float64
```

We saw the default `interpolate` method – a linear interpolation – in action. pandas assumes a linear relationship between two existing points.

pandas supports over a dozen `interpolation` functions, some of which require the `scipy` library to be installed. We will not cover `interpolation` methods in this chapter, but we encourage you to explore the various methods yourself. The right `interpolation` method will depend on the requirements of your application.



While, by default, pandas objects are time zone unaware, many real-world applications will make use of time zones. As with working with time in general, time zones are no trivial matter: do you know which countries have daylight saving time and do you know when the time zone is switched in those countries? Thankfully, pandas builds on the time zone capabilities of two popular and proven utility libraries for time and date handling: `pytz` and `dateutil`:

```
>>> t = pd.Timestamp('2000-01-01')
>>> t.tz is None
True
```

To supply time zone information, you can use the `tz` keyword argument:

```
>>> t = pd.Timestamp('2000-01-01', tz='Europe/Berlin')
>>> t.tz
<DstTzInfo 'Europe/Berlin' CET+1:00:00 STD>
```

This works for `ranges` as well:

```
>>> rng = pd.date_range('1/1/2000 00:00', periods=10, freq='D',
tz='Europe/London')
>>> rng
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04',
               '2000-01-05', '2000-01-06', '2000-01-07', '2000-01-08',
               '2000-01-09', '2000-01-10'],
              dtype='datetime64[ns]', freq='D', tz='Europe/London')
```

Time zone objects can also be constructed beforehand:

```
>>> import pytz
>>> tz = pytz.timezone('Europe/London')
>>> rng = pd.date_range('1/1/2000 00:00', periods=10, freq='D', tz=tz)
>>> rng
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04',
               '2000-01-05', '2000-01-06', '2000-01-07', '2000-01-08',
               '2000-01-09', '2000-01-10'],
              dtype='datetime64[ns]', freq='D', tz='Europe/London')
```

Sometimes, you will already have a time zone unaware time series object that you would like to make time zone aware. The `tz_localize` function helps to switch between time zone aware and time zone unaware objects:

```
>>> rng = pd.date_range('1/1/2000 00:00', periods=10, freq='D')
>>> ts = pd.Series(np.random.randn(len(rng)), rng)
>>> ts.index.tz is None
True
>>> ts_utc = ts.tz_localize('UTC')
>>> ts_utc.index.tz
<UTC>
```

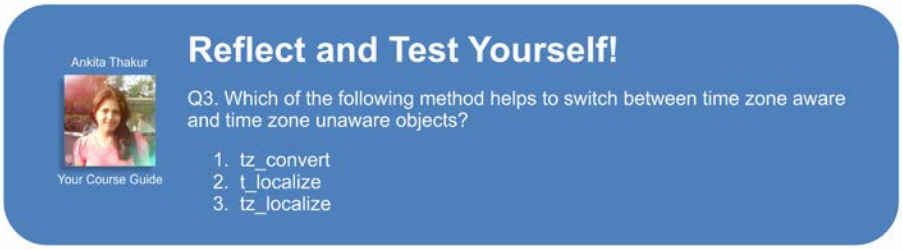
To move a time zone aware object to other time zones, you can use the `tz_convert` method:

```
>>> ts_utc.tz_convert('Europe/Berlin').index.tz
<DstTzInfo 'Europe/Berlin' LMT+0:53:00 STD>
```

Finally, to detach any time zone information from an object, it is possible to pass `None` to either `tz_convert` or `tz_localize`:

```
>>> ts_utc.tz_convert(None).index.tz is None
True
```

```
>>> ts_utc.tz_localize(None).index.tz is None
True
```



Timedeltas

Along with the powerful timestamp object, which acts as a building block for the `DatetimeIndex`, there is another useful data structure, which has been introduced in pandas 0.15 – the Timedelta. The Timedelta can serve as a basis for indices as well, in this case a `TimedeltaIndex`.

Timedeltas are differences in times, expressed in difference units. The `Timedelta` class in pandas is a subclass of `datetime.timedelta` from the Python standard library. As with other pandas data structures, the Timedelta can be constructed from a variety of inputs:

```
>>> pd.Timedelta('1 days')
Timedelta('1 days 00:00:00')
>>> pd.Timedelta('-1 days 2 min 10s 3us')
Timedelta('-2 days +23:57:49.999997')
>>> pd.Timedelta(days=1,seconds=1)
Timedelta('1 days 00:00:01')
```

As you would expect, `Timedeltas` allow basic arithmetic:

```
>>> pd.Timedelta(days=1) + pd.Timedelta(seconds=1)
Timedelta('1 days 00:00:01')
```

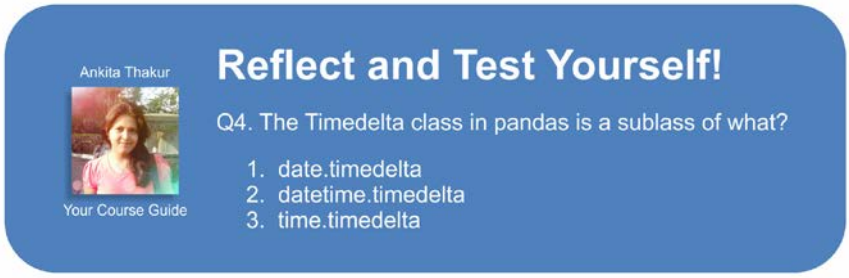
Similar to `to_datetime`, there is a `to_timedelta` function that can parse strings or lists of strings into Timedelta structures or `TimedeltaIndices`:

```
>>> pd.to_timedelta('20.1s')
Timedelta('0 days 00:00:20.100000')
```

Instead of absolute dates, we could create an index of `timedeltas`. Imagine measurements from a volcano, for example. We might want to take measurements but index it from a given date, for example the date of the last eruption. We could create a `timedelta` index that has the last seven days as entries:

```
>>> pd.to_timedelta(np.arange(7), unit='D')
TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days', '5
days', '6 days'], dtype='timedelta64[ns]', freq=None)
```

We could then work with time series data, indexed from the last eruption. If we had measurements for many eruptions (from possibly multiple volcanos), we would have an index that would make comparisons and analysis of this data easier. For example, we could ask whether there is a typical pattern that occurs between the third day and the fifth day after an eruption. This question would not be impossible to answer with a `DatetimeIndex`, but a `TimedeltaIndex` makes this kind of exploration much more convenient.



Time series plotting

pandas comes with great support for plotting, and this holds true for time series data as well.

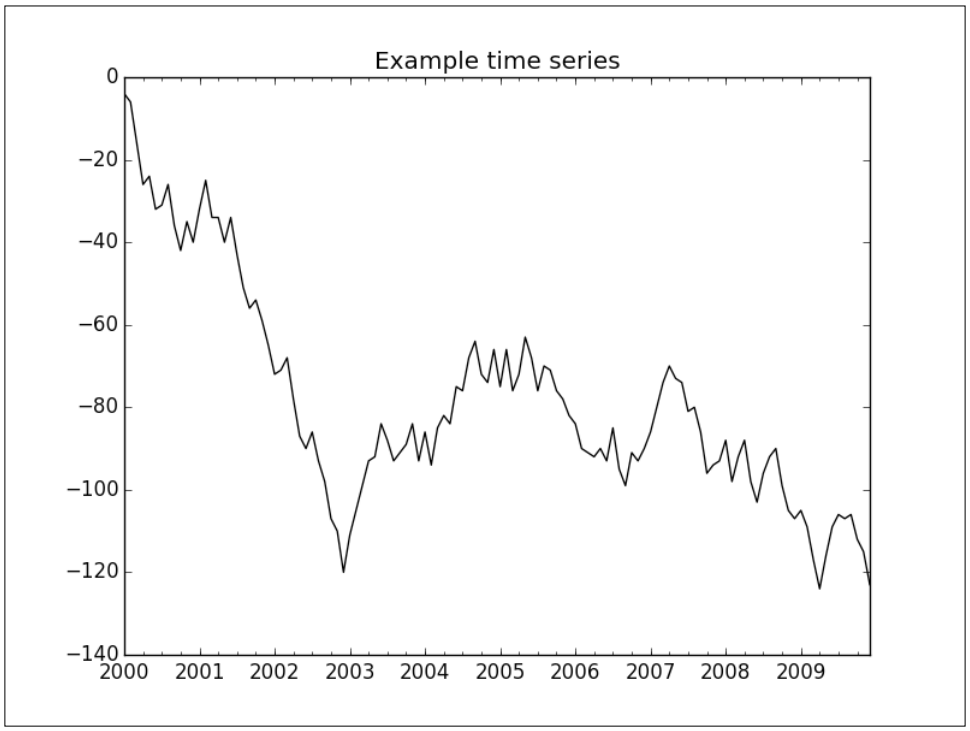
As a first example, let's take some monthly data and plot it:

```
>>> rng = pd.date_range(start='2000', periods=120, freq='MS')
>>> ts = pd.Series(np.random.randint(-10, 10, size=len(rng)), rng).
cumsum()
>>> ts.head()
2000-01-01    -4
2000-02-01    -6
2000-03-01   -16
2000-04-01   -26
2000-05-01   -24
Freq: MS, dtype: int64
```

Since matplotlib is used under the hood, we can pass a familiar parameter to plot, such as c for color, or title for the chart title:

```
>>> ts.plot(c='k', title='Example time series')
>>> plt.show()
```

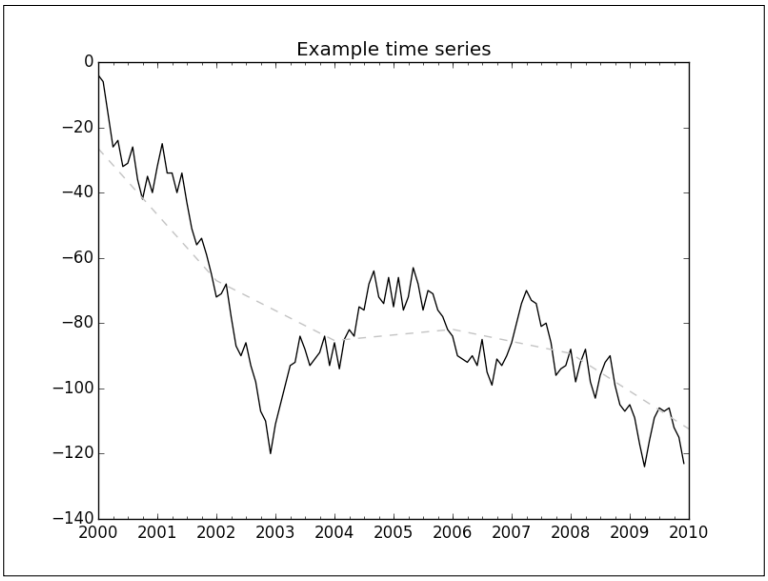
The following figure shows an example time series plot:



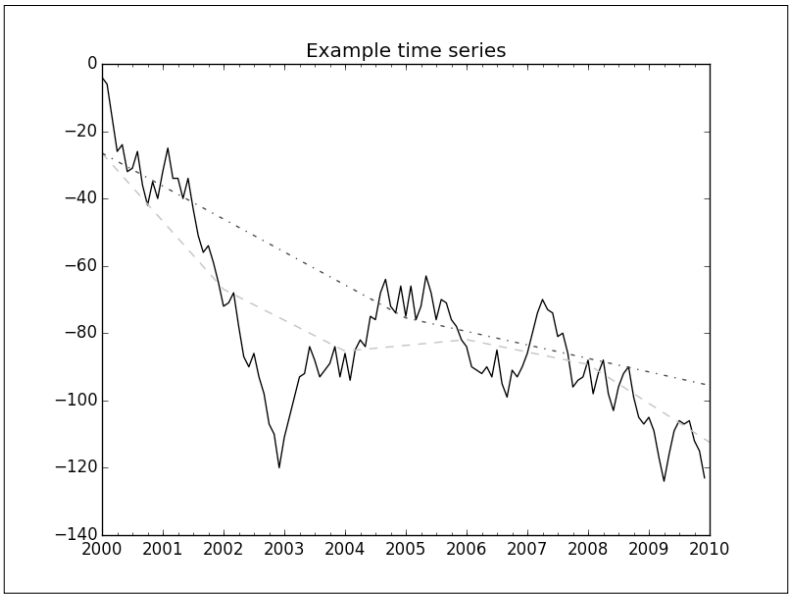
We can overlay an aggregate plot over 2 and 5 years:

```
>>> ts.resample('2A').plot(c='0.75', ls='--')
>>> ts.resample('5A').plot(c='0.25', ls='-.')
```

The following figure shows the resampled 2-year plot:

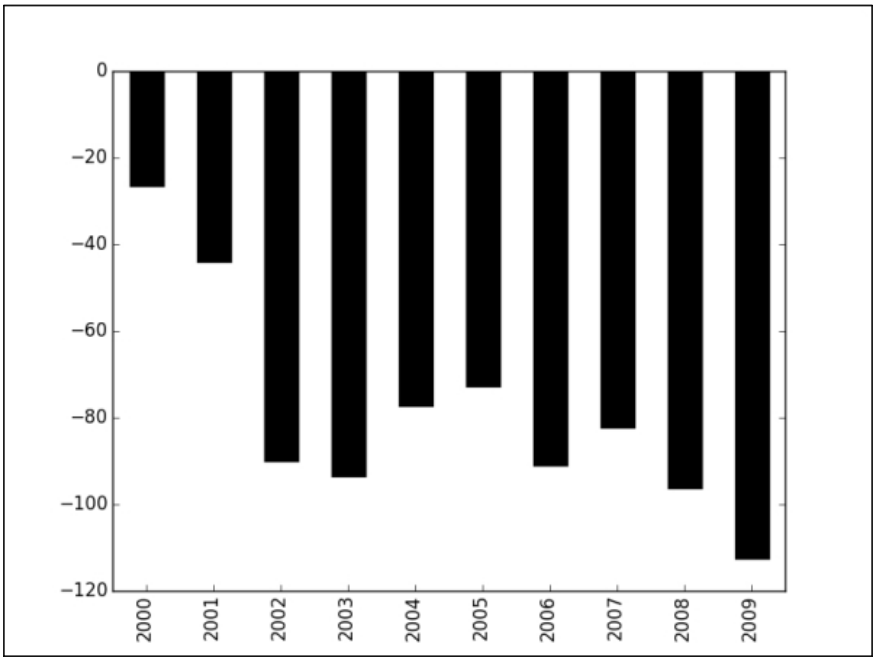


The following figure shows the resample 5-year plot:



We can pass the kind of chart to the `plot` method as well. The return value of the `plot` method is an `AxesSubplot`, which allows us to customize many aspects of the plot. Here we are setting the label values on the `X` axis to the year values from our time series:

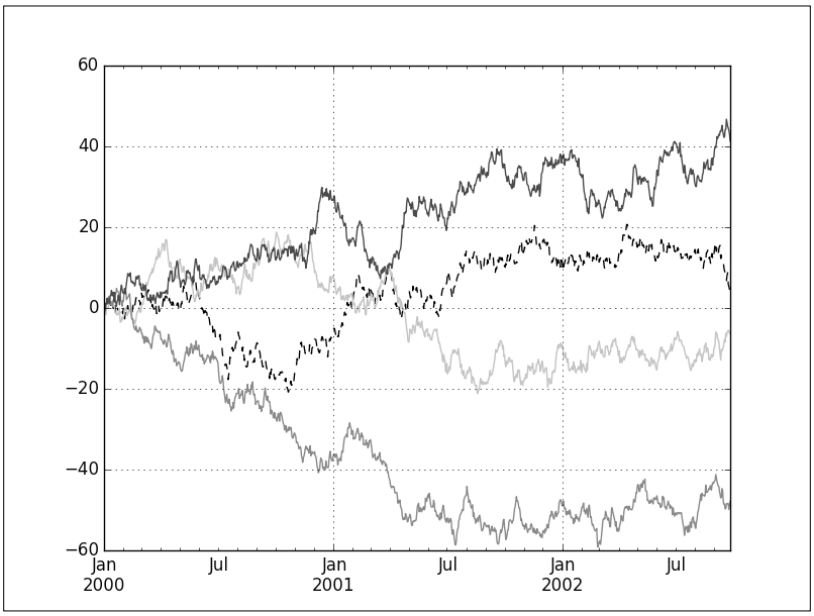
```
>>> plt.clf()
>>> tsx = ts.resample('1A')
>>> ax = tsx.plot(kind='bar', color='k')
>>> ax.set_xticklabels(tsx.index.year)
```



Let's imagine we have four time series that we would like to plot simultaneously. We generate a matrix of 1000 × 4 random values and treat each column as a separated time series:

```
>>> plt.clf()
>>> ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000',
periods=1000))
>>> df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
columns=['A', 'B', 'C', 'D'])
```

```
>>> df = df.cumsum()
>>> df.plot(color=['k', '0.75', '0.5', '0.25'], ls='--')
```



Your Coding Challenge

Ankita Thakur

Your Course Guide

1. Find one or two real-world examples for data sets, which could – in a sensible way – be assigned to the following groups:
   - Fixed frequency data
   - Variable frequency data
   - Data where frequency is usually measured in seconds
   - Data where frequency is measured in nanoseconds
   - Data, where a TimedeltaIndex would be preferable
2. Create various fixed frequency ranges:
   - Every minute between 1 AM and 2 AM on 2000-01-01
   - Every two hours for a whole week starting 2000-01-01
   - An entry for every Saturday and Sunday during the year 2000
   - An entry for every Monday of a month, if it was a business day, for the years 2000, 2001 and 2002

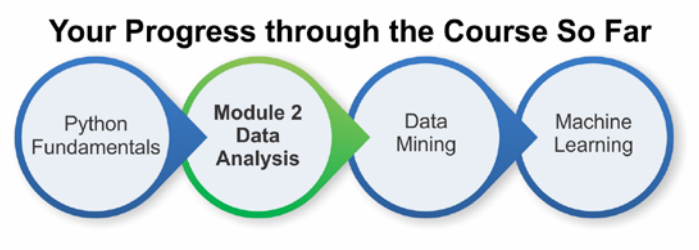
Summary of Module 2 Chapter 5

Ankita Thakur

Your Course Guide

In this chapter we showed how you can work with time series in pandas. We introduced two index types, the DatetimeIndex and the TimedeltaIndex and explored their building blocks in depth. pandas comes with versatile helper functions that take much of the pain out of parsing dates of various formats or generating fixed frequency sequences. Resampling data can help get a more condensed picture of the data, or it can help align various datasets of different frequencies to one another. One of the explicit goals of pandas is to make it easy to work with missing data, which is also relevant in the context of upsampling.

Finally, we showed how time series can be visualized. Since matplotlib and pandas are natural companions, we discovered that we can reuse our previous knowledge about matplotlib for time series data as well.
In the next chapter, we will explore ways to load and store data in text files and databases.



6
Interacting with Databases

Data analysis starts with data. It is therefore beneficial to work with data storage systems that are simple to set up, operate and where the data access does not become a problem in itself. In short, we would like to have database systems that are easy to embed into our data analysis processes and workflows. In this module, we focus mostly on the Python side of the database interaction, and we will learn how to get data into and out of pandas data structures.

There are numerous ways to store data. In this chapter, we are going to learn to interact with three main categories: text formats, binary formats and databases. We will focus on two storage solutions, MongoDB and Redis. MongoDB is a document-oriented database, which is easy to start with, since we can store JSON documents and do not need to define a schema upfront. Redis is a popular in-memory data structure store on top of which many applications can be built. It is possible to use Redis as a fast key-value store, but Redis supports lists, sets, hashes, bit arrays and even advanced data structures such as HyperLogLog out of the box as well.

Interacting with data in text format

Text is a great medium and it's a simple way to exchange information.
The following statement is taken from a quote attributed to *Doug McIlroy*:
Write programs to handle text streams, because that is the universal interface.

In this section we will start reading and writing data from and to text files.

Reading data from text format

Normally, the raw data logs of a system are stored in multiple text files, which can accumulate a large amount of information over time. Thankfully, it is simple to interact with these kinds of files in Python.

pandas supports a number of functions for reading data from a text file into a DataFrame object. The most simple one is the `read_csv()` function. Let's start with a small example file:

```
$ cat example_data/ex_06-01.txt
Name,age,major_id,sex,hometown
Nam,7,1,male,hcm
Mai,11,1,female,hcm
Lan,25,3,female,hn
Hung,42,3,male,tn
Nghia,26,3,male,dn
Vinh,39,3,male,vl
Hong,28,4,female,dn
```

The `cat` is the Unix shell command that can be used to print the content of a file to the screen.

In the preceding example file, each column is separated by comma and the first row is a header row, containing column names. To read the data file into the DataFrame object, we type the following command:

```
>>> df_ex1 = pd.read_csv('example_data/ex_06-01.txt')
>>> df_ex1
    Name  age  major_id     sex hometown
0    Nam    7         1    male      hcm
1    Mai   11         1  female      hcm
2    Lan   25         3  female       hn
3   Hung   42         3    male       tn
4  Nghia   26         3    male       dn
5   Vinh   39         3    male       vl
6   Hong   28         4  female       dn
```

We see that the `read_csv` function uses a comma as the default delimiter between columns in the text file and the first row is automatically used as a header for the columns. If we want to change this setting, we can use the `sep` parameter to change the separated symbol and set `header=None` in case the example file does not have a caption row.

See the following example:

```
$ cat example_data/ex_06-02.txt
Nam     7       1       male    hcm
Mai     11      1       female  hcm
Lan     25      3       female  hn
Hung    42      3       male    tn
Nghia   26      3       male    dn
Vinh    39      3       male    vl
Hong    28      4       female  dn

>>> df_ex2 = pd.read_csv('example_data/ex_06-02.txt',
                         sep = '\t', header=None)
>>> df_ex2
       0   1  2       3    4
0    Nam   7  1    male  hcm
1    Mai  11  1  female  hcm
2    Lan  25  3  female   hn
3   Hung  42  3    male   tn
4  Nghia  26  3    male   dn
5   Vinh  39  3    male   vl
6   Hong  28  4  female   dn
```

We can also set a specific row as the caption row by using the `header` that's equal to the index of the selected row. Similarly, when we want to use any column in the data file as the column index of DataFrame, we set `index_col` to the name or index of the column. We again use the second data file `example_data/ex_06-02.txt` to illustrate this:

```
>>> df_ex3 = pd.read_csv('example_data/ex_06-02.txt',
                         sep = '\t', header=None,
                         index_col=0)
>>> df_ex3
        1  2       3    4
0
Nam     7  1    male  hcm
Mai    11  1  female  hcm
Lan    25  3  female   hn
```

```
Hung   42  3    male   tn
Nghia  26  3    male   dn
Vinh   39  3    male   vl
Hong   28  4  female   dn
```

Apart from those parameters, we still have a lot of useful ones that can help us load data files into pandas objects more effectively. The following table shows some common parameters:

Parameter	Value	Description
`dtype`	Type name or dictionary of type of columns	Sets the data type for data or columns. By default it will try to infer the most appropriate data type.
`skiprows`	List-like or integer	The number of lines to skip (starting from 0).
`na_values`	List-like or dict, default None	Values to recognize as `NA`/`NaN`. If a dict is passed, this can be set on a per-column basis.
`true_values`	List	A list of values to be converted to Boolean True as well.
`false_values`	List	A list of values to be converted to Boolean False as well.
`keep_default_na`	`Bool, default True`	If the `na_values` parameter is present and `keep_default_na` is `False`, the default NaN values are ignored, otherwise they are appended to
`thousands`	`Str, default None`	The thousands separator
`nrows`	`Int, default None`	Limits the number of rows to read from the file.
`error_bad_lines`	`Boolean, default True`	If set to True, a DataFrame is returned, even if an error occurred during parsing.

Besides the `read_csv()` function, we also have some other parsing functions in pandas:

Function	Description
`read_table`	Read the general delimited file into DataFrame
`read_fwf`	Read a table of fixed-width formatted lines into DataFrame
`read_clipboard`	Read text from the clipboard and pass to `read_table`. It is useful for converting tables from web pages

In some situations, we cannot automatically parse data files from the disk using these functions. In that case, we can also open files and iterate through the reader, supported by the CSV module in the standard library:

```
$ cat example_data/ex_06-03.txt
Nam     7       1       male    hcm
Mai     11      1       female  hcm
Lan     25      3       female  hn
Hung    42      3       male    tn      single
Nghia   26      3       male    dn      single
Vinh    39      3       male    vl
Hong    28      4       female  dn

>>> import csv
>>> f = open('data/ex_06-03.txt')
>>> r = csv.reader(f, delimiter='\t')
>>> for line in r:
>>>    print(line)
['Nam', '7', '1', 'male', 'hcm']
['Mai', '11', '1', 'female', 'hcm']
['Lan', '25', '3', 'female', 'hn']
['Hung', '42', '3', 'male', 'tn', 'single']
['Nghia', '26', '3', 'male', 'dn', 'single']
['Vinh', '39', '3', 'male', 'vl']
['Hong', '28', '4', 'female', 'dn']
```

Writing data to text format

We saw how to load data from a text file into a pandas data structure. Now, we will learn how to export data from the data object of a program to a text file. Corresponding to the `read_csv()` function, we also have the `to_csv()` function, supported by pandas. Let's see the following example:

```
>>> df_ex3.to_csv('example_data/ex_06-02.out', sep = ';')
```

The result will look like this:

```
$ cat example_data/ex_06-02.out
0;1;2;3;4
Nam;7;1;male;hcm
Mai;11;1;female;hcm
Lan;25;3;female;hn
Hung;42;3;male;tn
Nghia;26;3;male;dn
Vinh;39;3;male;vl
Hong;28;4;female;dn
```

If we want to skip the header line or index column when writing out data into a disk file, we can set a `False` value to the header and index parameters:

```
>>> import sys
>>> df_ex3.to_csv(sys.stdout, sep='\t',
                  header=False, index=False)
7       1       male    hcm
11      1       female  hcm
25      3       female  hn
42      3       male    tn
26      3       male    dn
39      3       male    vl
28      4       female  dn
```

We can also write a subset of the columns of the DataFrame to the file by specifying them in the `columns` parameter:

```
>>> df_ex3.to_csv(sys.stdout, columns=[3,1,4],
                  header=False, sep='\t')
```

```
Nam     male    7       hcm
Mai     female  11      hcm
Lan     female  25      hn
Hung    male    42      tn
Nghia   male    26      dn
Vinh    male    39      vl
Hong    female  28      dn
```

With series objects, we can use the same function to write data into text files, with mostly the same parameters as earlier.

Interacting with data in binary format

We can read and write binary serialization of Python objects with the pickle module, which can be found in the standard library. Object serialization can be useful, if you work with objects that take a long time to create, like some machine learning models. By pickling such objects, subsequent access to this model can be made faster. It also allows you to distribute Python objects in a standardized way.

pandas includes support for pickling out of the box. The relevant methods are the `read_pickle()` and `to_pickle()` functions to read and write data from and to files easily. Those methods will write data to disk in the pickle format, which is a convenient short-term storage format:

```
>>> df_ex3.to_pickle('example_data/ex_06-03.out')
>>> pd.read_pickle('example_data/ex_06-03.out')
        1  2       3    4
0
Nam     7  1    male  hcm
Mai    11  1  female  hcm
Lan    25  3  female   hn
```

```
Hung   42  3    male   tn
Nghia  26  3    male   dn
Vinh   39  3    male   vl
Hong   28  4  female   dn
```

HDF5

HDF5 is not a database, but a data model and file format. It is suited for write-once, read-many datasets. An HDF5 file includes two kinds of objects: data sets, which are array-like collections of data, and groups, which are folder-like containers what hold data sets and other groups. There are some interfaces for interacting with HDF5 format in Python, such as `h5py` which uses familiar NumPy and Python constructs, such as dictionaries and NumPy array syntax. With `h5py`, we have high-level interface to the HDF5 API which helps us to get started. However, in this module, we will introduce another library for this kind of format called PyTables, which works well with pandas objects:

```
>>> store = pd.HDFStore('hdf5_store.h5')
>>> store
<class 'pandas.io.pytables.HDFStore'>
File path: hdf5_store.h5
Empty
```

We created an empty HDF5 file, named `hdf5_store.h5`. Now, we can write data to the file just like adding key-value pairs to a `dict`:

```
>>> store['ex3'] = df_ex3
>>> store['name'] = df_ex2[0]
>>> store['hometown'] = df_ex3[4]
>>> store
<class 'pandas.io.pytables.HDFStore'>
File path: hdf5_store.h5
/ex3                    frame        (shape->[7,4])
/hometown               series       (shape->[1])
/name                   series       (shape->[1])
```

Objects stored in the HDF5 file can be retrieved by specifying the object keys:

```
>>> store['name']
0      Nam
1      Mai
2      Lan
```

```
3     Hung
4    Nghia
5     Vinh
6     Hong
Name: 0, dtype: object
```

Once we have finished interacting with the HDF5 file, we close it to release the file handle:

```
>>> store.close()
>>> store
<class 'pandas.io.pytables.HDFStore'>
File path: hdf5_store.h5
File is CLOSED
```

There are other supported functions that are useful for working with the HDF5 format. You should explore ,in more detail, two libraries – `pytables` and `h5py` – if you need to work with huge quantities of data.

Interacting with data in MongoDB

Many applications require more robust storage systems then text files, which is why many applications use databases to store data. There are many kinds of databases, but there are two broad categories: relational databases, which support a standard declarative language called SQL, and so called NoSQL databases, which are often able to work without a predefined schema and where a data instance is more properly described as a document, rather as a row.

MongoDB is a kind of NoSQL database that stores data as documents, which are grouped together in collections. Documents are expressed as JSON objects. It is fast and scalable in storing, and also flexible in querying, data. To use MongoDB in Python, we need to import the `pymongo` package and open a connection to the database by passing a hostname and port. We suppose that we have a MongoDB instance, running on the default host (`localhost`) and port (`27017`):

```
>>> import pymongo
>>> conn = pymongo.MongoClient(host='localhost', port=27017)
```

If we do not put any parameters into the `pymongo.MongoClient()` function, it will automatically use the default host and port.

In the next step, we will interact with databases inside the MongoDB instance. We can list all databases that are available in the instance:

```
>>> conn.database_names()
['local']
>>> lc = conn.local
>>> lc
Database(MongoClient('localhost', 27017), 'local')
```

The preceding snippet says that our MongoDB instance only has one database, named 'local'. If the databases and collections we point to do not exist, MongoDB will create them as necessary:

```
>>> db = conn.db
>>> db
Database(MongoClient('localhost', 27017), 'db')
```

Each database contains groups of documents, called collections. We can understand them as tables in a relational database. To list all existing collections in a database, we use `collection_names()` function:

```
>>> lc.collection_names()
['startup_log', 'system.indexes']
>>> db.collection_names()
[]
```

Our `db` database does not have any collections yet. Let's create a collection, named `person`, and insert data from a DataFrame object to it:

```
>>> collection = db.person
>>> collection
Collection(Database(MongoClient('localhost', 27017), 'db'), 'person')
>>> # insert df_ex2 DataFrame into created collection
>>> import json
>>> records = json.load(df_ex2.T.to_json()).values()
>>> records
dict_values([{'2': 3, '3': 'male', '1': 39, '4': 'vl', '0': 'Vinh'},
{'2': 3, '3': 'male', '1': 26, '4': 'dn', '0': 'Nghia'}, {'2': 4, '3':
'female', '1': 28, '4': 'dn', '0': 'Hong'}, {'2': 3, '3': 'female', '1':
25, '4': 'hn', '0': 'Lan'}, {'2': 3, '3': 'male', '1': 42, '4': 'tn',
'0': 'Hung'}, {'2': 1, '3':'male', '1': 7, '4': 'hcm', '0': 'Nam'}, {'2':
1, '3': 'female', '1': 11, '4': 'hcm', '0': 'Mai'}])
>>> collection.insert(records)
```

```
[ObjectId('557da218f21c761d7c176a40'),
 ObjectId('557da218f21c761d7c176a41'),
 ObjectId('557da218f21c761d7c176a42'),
 ObjectId('557da218f21c761d7c176a43'),
 ObjectId('557da218f21c761d7c176a44'),
 ObjectId('557da218f21c761d7c176a45'),
 ObjectId('557da218f21c761d7c176a46')]
```

The `df_ex2` is transposed and converted to a JSON string before loading into a dictionary. The `insert()` function receives our created dictionary from `df_ex2` and saves it to the collection.

If we want to list all data inside the collection, we can execute the following commands:

```
>>> for cur in collection.find():
>>>     print(cur)
{'4': 'vl', '2': 3, '3': 'male', '1': 39, '_id':
ObjectId('557da218f21c761d7c176
a40'), '0': 'Vinh'}
{'4': 'dn', '2': 3, '3': 'male', '1': 26, '_id':
ObjectId('557da218f21c761d7c176
a41'), '0': 'Nghia'}
{'4': 'dn', '2': 4, '3': 'female', '1': 28, '_id':
ObjectId('557da218f21c761d7c1
76a42'), '0': 'Hong'}
{'4': 'hn', '2': 3, '3': 'female', '1': 25, '_id':
ObjectId('557da218f21c761d7c1
76a43'), '0': 'Lan'}
{'4': 'tn', '2': 3, '3': 'male', '1': 42, '_id':
ObjectId('557da218f21c761d7c176
a44'), '0': 'Hung'}
{'4': 'hcm', '2': 1, '3': 'male', '1': 7, '_id':
ObjectId('557da218f21c761d7c176
a45'), '0': 'Nam'}
{'4': 'hcm', '2': 1, '3': 'female', '1': 11, '_id':
ObjectId('557da218f21c761d7c
176a46'), '0': 'Mai'}
```

If we want to query data from the created collection with some conditions, we can use the `find()` function and pass in a dictionary describing the documents we want to retrieve. The returned result is a `cursor` type, which supports the iterator protocol:

```
>>> cur = collection.find({'3' : 'male'})
>>> type(cur)
pymongo.cursor.Cursor
>>> result = pd.DataFrame(list(cur))
>>> result
       0   1  2     3    4                         _id
0   Vinh  39  3  male   vl  557da218f21c761d7c176a40
1  Nghia  26  3  male   dn  557da218f21c761d7c176a41
2   Hung  42  3  male   tn  557da218f21c761d7c176a44
3    Nam   7  1  male  hcm  557da218f21c761d7c176a45
```

Sometimes, we want to delete data in MongdoDB. All we need to do is to pass a query to the `remove()` method on the collection:

```
>>> # before removing data
>>> pd.DataFrame(list(collection.find()))
       0   1  2       3    4                         _id
0   Vinh  39  3    male   vl  557da218f21c761d7c176a40
1  Nghia  26  3    male   dn  557da218f21c761d7c176a41
2   Hong  28  4  female   dn  557da218f21c761d7c176a42
3    Lan  25  3  female   hn  557da218f21c761d7c176a43
4   Hung  42  3    male   tn  557da218f21c761d7c176a44
5    Nam   7  1    male  hcm  557da218f21c761d7c176a45
6    Mai  11  1  female  hcm  557da218f21c761d7c176a46


>>> # after removing records which have '2' column as 1 and '3' column as
'male'
>>> collection.remove({'2': 1, '3': 'male'})
{'n': 1, 'ok': 1}
>>> cur_all = collection.find();
>>> pd.DataFrame(list(cur_all))
       0   1  2       3    4                         _id
0   Vinh  39  3    male   vl  557da218f21c761d7c176a40
1  Nghia  26  3    male   dn  557da218f21c761d7c176a41
```

```
2   Hong  28  4  female   dn  557da218f21c761d7c176a42
3    Lan  25  3  female   hn  557da218f21c761d7c176a43
4   Hung  42  3    male   tn  557da218f21c761d7c176a44
5    Mai  11  1  female  hcm  557da218f21c761d7c176a46
```

We learned step by step how to insert, query and delete data in a collection. Now, we will show how to update existing data in a collection in MongoDB:

```
>>> doc = collection.find_one({'1' : 42})
>>> doc['4'] = 'hcm'
>>> collection.save(doc)
ObjectId('557da218f21c761d7c176a44')
>>> pd.DataFrame(list(collection.find()))
       0   1  2       3    4                         _id
0   Vinh  39  3    male   vl  557da218f21c761d7c176a40
1  Nghia  26  3    male   dn  557da218f21c761d7c176a41
2   Hong  28  4  female   dn  557da218f21c761d7c176a42
3    Lan  25  3  female   hn  557da218f21c761d7c176a43
4   Hung  42  3    male  hcm  557da218f21c761d7c176a44
5    Mai  11  1  female  hcm  557da218f21c761d7c176a46
```

The following table shows methods that provide shortcuts to manipulate documents in MongoDB:

Update Method	Description
`inc()`	Increment a numeric field
`set()`	Set certain fields to new values
`unset()`	Remove a field from the document
`push()`	Append a value onto an array in the document
`pushAll()`	Append several values onto an array in the document
`addToSet()`	Add a value to an array, only if it does not exist
`pop()`	Remove the last value of an array
`pull()`	Remove all occurrences of a value from an array
`pullAll()`	Remove all occurrences of any set of values from an array
`rename()`	Rename a field
`bit()`	Update a value by bitwise operation

Interacting with data in Redis

Redis is an advanced kind of key-value store where the values can be of different types: string, list, set, sorted set or hash. Redis stores data in memory like memcached but it can be persisted on disk, unlike memcached, which has no such option. Redis supports fast reads and writes, in the order of 100,000 set or get operations per second.

To interact with Redis, we need to install the `Redis-py` module to Python, which is available on `pypi` and can be installed with `pip`:

```
$ pip install redis
```

Now, we can connect to Redis via the host and port of the DB server. We assume that we have already installed a Redis server, which is running with the default host (`localhost`) and port (`6379`) parameters:

```
>>> import redis
>>> r = redis.StrictRedis(host='127.0.0.1', port=6379)
>>> r
StrictRedis<ConnectionPool<Connection<host=localhost,port=6379,db=0>>>
```

As a first step to storing data in Redis, we need to define which kind of data structure is suitable for our requirements. In this section, we will introduce four commonly used data structures in Redis: simple value, list, set and ordered set. Though data is stored into Redis in many different data structures, each value must be associated with a key.

The simple value

This is the most basic kind of value in Redis. For every key in Redis, we also have a value that can have a data type, such as string, integer or double. Let's start with an example for setting and getting data to and from Redis:

```
>>> r.set('gender:An', 'male')
True
>>> r.get('gender:An')
b'male'
```

In this example we want to store the gender info of a person, named `An` into Redis. Our key is `gender:An` and our value is `male`. Both of them are a type of string.

The `set()` function receives two parameters: the key and the value. The first parameter is the key and the second parameter is value. If we want to update the value of this key, we just call the function again and change the value of the second parameter. Redis automatically updates it.

The `get()` function will retrieve the value of our key, which is passed as the parameter. In this case, we want to get gender information of the key `gender:An`.

In the second example, we show you another kind of value type, an integer:

```
>>> r.set('visited_time:An', 12)
True
>>> r.get('visited_time:An')
b'12'
>>> r.incr('visited_time:An', 1)
13
>>> r.get('visited_time:An')
b'13'
```

We saw a new function, `incr()`, which used to increment the value of key by a given amount. If our key does not exist, RedisDB will create the key with the given increment as the value.

List

We have a few methods for interacting with list values in Redis. The following example uses `rpush()` and `lrange()` functions to put and get list data to and from the DB:

```
>>> r.rpush('name_list', 'Tom')
1L
>>> r.rpush('name_list', 'John')
2L
```

```
>>> r.rpush('name_list', 'Mary')
3L
>>> r.rpush('name_list', 'Jan')
4L
>>> r.lrange('name_list', 0, -1)
[b'Tom', b'John', b'Mary', b'Jan']
>>> r.llen('name_list')
4
>>> r.lindex('name_list', 1)
b'John'
```

Besides the `rpush()` and `lrange()` functions we used in the example, we also want to introduce two others functions. First, the `llen()` function is used to get the length of our list in the Redis for a given key. The `lindex()` function is another way to retrieve an item of the list. We need to pass two parameters into the function: a key and an index of item in the list. The following table lists some other powerful functions in processing list data structure with Redis:

Function	Description
`rpushx(name, value)`	Push value onto the tail of the list name if name exists
`rpop(name)`	Remove and return the last item of the list name
`lset(name, index, value)`	Set item at the index position of the list name to input value
`lpushx(name,value)`	Push value on the head of the list name if name exists
`lpop(name)`	Remove and return the first item of the list name

Set

This data structure is also similar to the list type. However, in contrast to a list, we cannot store duplicate values in our set:

```
>>> r.sadd('country', 'USA')
1
>>> r.sadd('country', 'Italy')
1
>>> r.sadd('country', 'Singapore')
1
>>> r.sadd('country', 'Singapore')
0
```

```
>>> r.smembers('country')
{b'Italy', b'Singapore', b'USA'}
>>> r.srem('country', 'Singapore')
1
>>> r.smembers('country')
{b'Italy', b'USA'}
```

Corresponding to the list data structure, we also have a number of functions to get, set, update or delete items in the set. They are listed in the supported functions for set data structure, in the following table:

Function	Description
`sadd(name, values)`	Add value(s) to the set with key name
`scard(name)`	Return the number of element in the set with key name
`smembers(name)`	Return all members of the set with key name
`srem(name, values)`	Remove value(s) from the set with key name

Ordered set

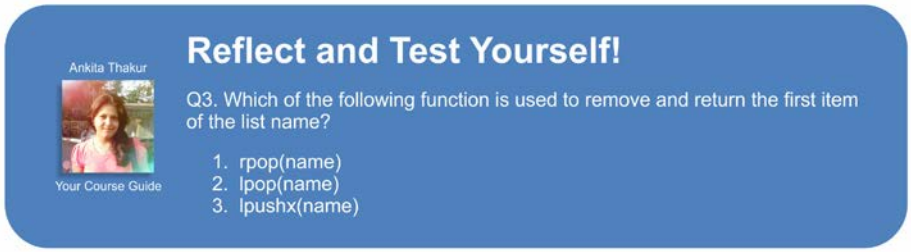
The ordered set data structure takes an extra attribute when we add data to a set called **score**. An ordered set will use the score to determine the order of the elements in the set:

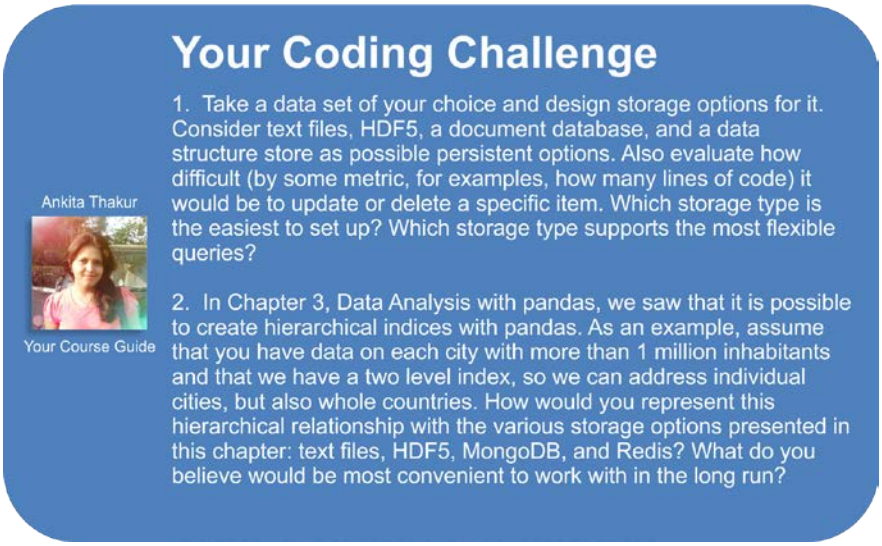
```
>>> r.zadd('person:A', 10, 'sub:Math')
1
>>> r.zadd('person:A', 7, 'sub:Bio')
1
>>> r.zadd('person:A', 8, 'sub:Chem')
1
>>> r.zrange('person:A', 0, -1)
[b'sub:Bio', b'sub:Chem', b'sub:Math']
>>> r.zrange('person:A', 0, -1, withscores=True)
[(b'sub:Bio', 7.0), (b'sub:Chem', 8.0), (b'sub:Math', 10.0)]
```

By using the `zrange(name, start, end)` function, we can get a range of values from the sorted set between the start and end score sorted in ascending order by default. If we want to change the `way` method of sorting, we can set the `desc` parameter to `True`. The `withscore` parameter is used in case we want to get the scores along with the return values. The return type is a list of (value, score) pairs as you can see in the preceding example.

See the following table for more functions available on ordered sets:

Function	Description
`zcard(name)`	Return the number of elements in the sorted set with key name
`zincrby(name, value, amount=1)`	Increment the score of value in the sorted set with key name by amount
`zrangebyscore(name, min, max, withscores=False, start=None, num=None)`	Return a range of values from the sorted set with key name with a score between min and max. If `withscores` is `true`, return the scores along with the values. If start and `num` are given, return a slice of the range
`zrank(name, value)`	Return a 0-based value indicating the rank of value in the sorted set with key name
`zrem(name, values)`	Remove member value(s) from the sorted set with key name





Summary of Module 2 Chapter 6

Ankita Thakur

Your Course Guide

We finished covering the basics of interacting with data in different commonly used storage mechanisms from the simple ones, such as text files, over more structured ones, such as HDF5, to more sophisticated data storage systems, such as MongoDB and Redis. The most suitable type of storage will depend on your use case. The choice of the data storage layer technology plays an important role in the overall design of data processing systems. Sometimes, we need to combine various database systems to store our data, such as complexity of the data, performance of the system or computation requirements.

Your Progress through the Course So Far

7
Data Analysis Application Examples

In this chapter, we want to get you acquainted with typical data preparation tasks and analysis techniques, because being fluent in preparing, grouping, and reshaping data is an important building block for successful data analysis.

While preparing data seems like a mundane task – and often it is – it is a step we cannot skip, although we can strive to simplify it by using tools such as pandas.

Why is preparation necessary at all? Because most useful data will come from the real world and will have deficiencies, contain errors or will be fragmentary.

There are more reasons why data preparation is useful: it gets you in close contact with the raw material. Knowing your input helps you to spot potential errors early and build confidence in your results.

Here are a few data preparation scenarios:

- A client hands you three files, each containing time series data about a single geological phenomenon, but the observed data is recorded on different intervals and uses different separators
- A machine learning algorithm can only work with numeric data, but your input only contains text labels
- You are handed the raw logs of a web server of an up and coming service and your task is to make suggestions on a growth strategy, based on existing visitor behavior

Data munging

The arsenal of tools for data munging is huge, and while we will focus on Python we want to mention some useful tools as well. If they are available on your system and you expect to work a lot with data, they are worth learning.

One group of tools belongs to the Unix tradition, which emphasizes text processing and as a consequence has, over the last four decades, developed many high-performance and battle-tested tools for dealing with text. Some common tools are: `sed`, `grep`, `awk`, `sort`, `uniq`, `tr`, `cut`, `tail`, and `head`. They do very elementary things, such as filtering out lines (`grep`) or columns (`cut`) from files, replacing text (`sed`, `tr`) or displaying only parts of files (`head`, `tail`).

We want to demonstrate the power of these tools with a single example only.

Imagine you are handed the log files of a web server and you are interested in the distribution of the IP addresses.

Each line of the log file contains an entry in the common log server format (you can download this data set from `http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html`:

```
$ cat epa-html.txt
wpbfl2-45.gate.net [29:23:56:12] "GET /Access/ HTTP/1.0" 200 2376ebaca.
icsi.net [30:00:22:20] "GET /Info.html HTTP/1.0" 200 884
```

For instance, we want to know how often certain users have visited our site.

We are interested in the first column only, since this is where the IP address or hostname can be found. After that, we need to count the number of occurrences of each host and finally display the results in a friendly way.

The `sort | uniq -c` stanza is our workhorse here: it sorts the data first and `uniq -c` will save the number of occurrences along with the value. The `sort -nr | head -15` is our formatting part; we sort numerically (`-n`) and in reverse (`-r`), and keep only the top 15 entries.

Putting it all together with pipes:

```
$ cut -d ' ' -f 1 epa-http.txt | sort | uniq -c | sort -nr | head -15
294 sandy.rtptok1.epa.gov
292 e659229.boeing.com
266 wicdgserv.wic.epa.gov
263 keyhole.es.dupont.com
248 dwilson.pr.mcs.net
176 oea4.r8stw56.epa.gov
174 macip26.nacion.co.cr
172 dcimsd23.dcimsd.epa.gov
167 www-b1.proxy.aol.com
158 piweba3y.prodigy.com
152 wictrn13.dcwictrn.epa.gov
151 nntp1.reach.com
151 inetg1.arco.com
149 canto04.nmsu.edu
146 weisman.metrokc.gov
```

With one command, we get to convert a sequential server log into an ordered list of the most common hosts that visited our site. We also see that we do not seem to have large differences in the number of visits among our top users.

There are more little helpful tools of which the following are just a tiny selection:

- `csvkit`: This is the suite of utilities for working with CSV, the king of tabular file formats
- `jq`: This is a lightweight and flexible command-line JSON processor
- `xmlstarlet`: This is a tool that supports XML queries with XPath, among other things
- `q`: This runs SQL on text files

Where the Unix command line ends, lightweight languages take over. You might be able to get an impression from text only, but your colleagues might appreciate visual representations, such as charts or pretty graphs, generated by matplotlib, much more.

Python and its data tools ecosystem are much more versatile than the command line, but for first explorations and simple operations the effectiveness of the command line is often unbeatable.

Cleaning data

Most real-world data will have some defects and therefore will need to go through a cleaning step first. We start with a small file. Although this file contains only four rows, it will allow us to demonstrate the process up to a cleaned data set:

```
$ cat small.csv
22,6.1
41,5.7
  18,5.3*
29,NA
```

Note that this file has a few issues. The lines that contain values are all comma-separated, but we have missing (NA) and probably unclean (5.3*) values. We can load this file into a data frame, nevertheless:

```
>>> import pandas as pd
>>> df = pd.read_csv("small.csv")
>>> df
   22   6.1
0  41   5.7
1  18  5.3*
2  29   NaN
```

pandas used the first row as `header`, but this is not what we want:

```
>>> df = pd.read_csv("small.csv", header=None)
>>> df
    0     1
0  22   6.1
1  41   5.7
2  18  5.3*
3  29   NaN
```

This is better, but instead of numeric values, we would like to supply our own column names:

```
>>> df = pd.read_csv("small.csv", names=["age", "height"])
>>> df
   age height
0   22    6.1
1   41    5.7
2   18   5.3*
3   29    NaN
```

The `age` column looks good, since pandas already inferred the intended type, but the `height` cannot be parsed into numeric values yet:

```
>>> df.age.dtype
dtype('int64')
>>> df.height.dtype
dtype('O')
```

If we try to coerce the `height` column into float values, pandas will report an exception:

```
>>> df.height.astype('float')
ValueError: invalid literal for float(): 5.3*
```

We could use whatever value is parseable as a float and throw away the rest with the `convert_objects` method:

```
>>> df.height.convert_objects(convert_numeric=True)
0    6.1
1    5.7
2    NaN
3    NaN
Name: height, dtype: float64
```

If we know in advance the undesirable characters in our data set, we can augment the `read_csv` method with a custom converter function:

```
>>> remove_stars = lambda s: s.replace("*", "")
>>> df = pd.read_csv("small.csv", names=["age", "height"],
                     converters={"height": remove_stars})
>>> df
   age height
0   22    6.1
1   41    5.7
2   18    5.3
3   29     NA
```

Now we can finally make the height column a bit more useful. We can assign it the updated version, which has the favored type:

```
>>> df.height = df.height.convert_objects(convert_numeric=True)
>>> df
   age  height
0   22     6.1
1   41     5.7
2   18     5.3
3   29     NaN
```

If we wanted to only keep the complete entries, we could drop any row that contains undefined values:

```
>>> df.dropna()
   age  height
0   22     6.1
1   41     5.7
2   18     5.3
```

We could use a default height, maybe a fixed value:

```
>>> df.fillna(5.0)
   age  height
0   22     6.1
1   41     5.7
2   18     5.3
3   29     5.0
```

On the other hand, we could also use the average of the existing values:

```
>>> df.fillna(df.height.mean())
   age  height
0   22     6.1
1   41     5.7
2   18     5.3
3   29     5.7
```

The last three data frames are complete and correct, depending on your definition of correct when dealing with missing values. Especially, the columns have the requested types and are ready for further analysis. Which of the data frames is best suited will depend on the task at hand.

Filtering

Even if we have clean and probably correct data, we might want to use only parts of it or we might want to check for outliers. An outlier is an observation point that is distant from other observations because of variability or measurement errors. In both cases, we want to reduce the number of elements in our data set to make it more relevant for further processing.

In this example, we will try to find potential outliers. We will use the Europe Brent Crude Oil Spot Price as recorded by the U.S. Energy Information Administration. The raw Excel data is available from `http://www.eia.gov/dnav/pet/hist_xls/rbrted.xls` (it can be found in the second worksheet). We cleaned the data slightly (the cleaning process is part of an exercise at the end of this chapter) and will work with the following data frame, containing 7160 entries, ranging from 1987 to 2015:

```
>>> df.head()
        date  price
0 1987-05-20  18.63
1 1987-05-21  18.45
2 1987-05-22  18.55
3 1987-05-25  18.60
4 1987-05-26  18.63
>>> df.tail()
           date  price
7155 2015-08-04  49.08
7156 2015-08-05  49.04
7157 2015-08-06  47.80
7158 2015-08-07  47.54
7159 2015-08-10  48.30
```

While many people know about oil prices – be it from the news or the filling station – let us forget anything we know about it for a minute. We could first ask for the extremes:

```
>>> df[df.price==df.price.min()]
           date  price
2937 1998-12-10    9.1
>>> df[df.price==df.price.max()]
           date   price
5373 2008-07-03  143.95
```

Another way to find potential outliers would be to ask for values that deviate most from the mean. We can use the `np.abs` function to calculate the deviation from the mean first:

```
>>> np.abs(df.price - df.price.mean())
0       26.17137  1       26.35137  7157     2.99863
7158     2.73863  7159     3.49863
```

We can now compare this deviation from a multiple – we choose 2.5 – of the standard deviation:

```
>>> import numpy as np
>>> df[np.abs(df.price - df.price.mean()) > 2.5 * df.price.std()]
       date   price
5354 2008-06-06  132.81
5355 2008-06-09  134.43
5356 2008-06-10  135.24
5357 2008-06-11  134.52
5358 2008-06-12  132.11
5359 2008-06-13  134.29
5360 2008-06-16  133.90
5361 2008-06-17  131.27
5363 2008-06-19  131.84
5364 2008-06-20  134.28
5365 2008-06-23  134.54
5366 2008-06-24  135.37
5367 2008-06-25  131.59
5368 2008-06-26  136.82
5369 2008-06-27  139.38
5370 2008-06-30  138.40
5371 2008-07-01  140.67
5372 2008-07-02  141.24
5373 2008-07-03  143.95
5374 2008-07-07  139.62
5375 2008-07-08  134.15
5376 2008-07-09  133.91
5377 2008-07-10  135.81
5378 2008-07-11  143.68
5379 2008-07-14  142.43
5380 2008-07-15  136.02
5381 2008-07-16  133.31
5382 2008-07-17  134.16
```

We see that those few days in summer 2008 must have been special. Sure enough, it is not difficult to find articles and essays with titles like *Causes and Consequences of the Oil Shock of 2007–08*. We have discovered a trace to these events solely by looking at the data.

We could ask the preceding question for each decade separately. We first make our data frame look more like a time series:

```
>>> df.index = df.date
>>> del df["date"]
>>> df.head()
            price
date
1987-05-20  18.63  1987-05-21  18.45
1987-05-22  18.55  1987-05-25  18.60
1987-05-26  18.63
```

We could filter out the eighties:

```
>>> decade = df["1980":"1989"]
>>> decade[np.abs(decade.price - decade.price.mean()) > 2.5 * decade.
price.std()]
            price
date
1988-10-03  11.60  1988-10-04  11.65  1988-10-05  11.20  1988-10-06
11.30  1988-10-07  11.35
```

We observe that within the data available (1987–1989), the fall of 1988 exhibits a slight spike in the oil prices. Similarly, during the nineties, we see that we have a larger deviation, in the fall of 1990:

```
>>> decade = df["1990":"1999"]
>>> decade[np.abs(decade.price - decade.price.mean()) > 5 * decade.price.
std()]
            price
date
1990-09-24  40.75  1990-09-26  40.85  1990-09-27  41.45  1990-09-28
41.00  1990-10-09  40.90  1990-10-10  40.20  1990-10-11  41.15
```

There are many more use cases for filtering data. Space and time are typical units: you might want to filter census data by state or city, or economical data by quarter. The possibilities are endless and will be driven by your project.

Merging data

The situation is common: you have multiple data sources, but in order to make statements about the content, you would rather combine them. Fortunately, pandas' concatenation and merge functions abstract away most of the pain, when combining, joining, or aligning data. It does so in a highly optimized manner as well.

In a case where two data frames have a similar shape, it might be useful to just append one after the other. Maybe `A` and `B` are products and one data frame contains the number of items sold per product in a store:

```
>>> df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
>>> df1
   A  B
0  1  4
1  2  5
2  3  6
>>> df2 = pd.DataFrame({'A': [4, 5, 6], 'B': [7, 8, 9]})
>>> df2
   A  B
0  4  7
1  5  8
2  6  9
>>> df1.append(df2)
   A  B
0  1  4
1  2  5
2  3  6
0  4  7
1  5  8
2  6  9
```

Sometimes, we won't care about the indices of the originating data frames:

```
>>> df1.append(df2, ignore_index=True)
   A  B
0  1  4
1  2  5
2  3  6
3  4  7
4  5  8
5  6  9
```

A more flexible way to combine objects is offered by the `pd.concat` function, which takes an arbitrary number of series, data frames, or panels as input. The default behavior resembles an append:

```
>>> pd.concat([df1, df2])
   A  B
0  1  4
1  2  5
2  3  6
0  4  7
1  5  8
2  6  9
```

The default `concat` operation appends both frames along the rows – or index, which corresponds to axis 0. To concatenate along the columns, we can pass in the axis keyword argument:

```
>>> pd.concat([df1, df2], axis=1)
   A  B  A  B
0  1  4  4  7
1  2  5  5  8
2  3  6  6  9
```

We can add keys to create a hierarchical index.

```
>>> pd.concat([df1, df2], keys=['UK', 'DE'])
      A  B
UK 0  1  4
   1  2  5
   2  3  6
DE 0  4  7
   1  5  8
   2  6  9
```

This can be useful if you want to refer back to parts of the data frame later. We use the `ix` indexer:

```
>>> df3 = pd.concat([df1, df2], keys=['UK', 'DE'])
>>> df3.ix["UK"]
   A  B
0  1  4
1  2  5
2  3  6
```

Data frames resemble database tables. It is therefore not surprising that pandas implements SQL-like join operations on them. What is positively surprising is that these operations are highly optimized and extremely fast:

```
>>> import numpy as np
>>> df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
                        'value': range(4)})
>>> df1
  key  value
0   A      0
1   B      1
2   C      2
3   D      3
>>> df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
                        'value': range(10, 14)})
>>> df2
  key  value
0   B     10
1   D     11
2   D     12
3   E     13
```

If we merge on `key`, we get an inner join. This creates a new data frame by combining the column values of the original data frames based upon the join predicate, here the `key` attribute is used:

```
>>> df1.merge(df2, on='key')
  key  value_x  value_y
0   B        1       10
1   D        3       11
2   D        3       12
```

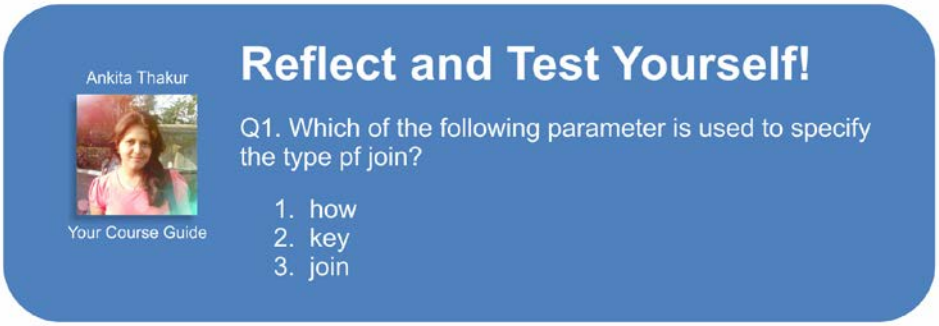
A left, right and full join can be specified by the `how` parameter:

```
>>> df1.merge(df2, on='key', how='left')
  key  value_x  value_y
0   A        0      NaN
1   B        1       10
2   C        2      NaN
3   D        3       11
4   D        3       12
```

```
>>> df1.merge(df2, on='key', how='right')
  key  value_x  value_y
0   B        1       10
1   D        3       11
2   D        3       12
3   E      NaN       13
>>> df1.merge(df2, on='key', how='outer')
  key  value_x  value_y
0   A        0      NaN
1   B        1       10
2   C        2      NaN
3   D        3       11
4   D        3       12
5   E      NaN       13
```

The merge methods can be specified with the how parameter. The following table shows the methods in comparison with SQL:

Merge Method	SQL Join Name	Description
`left`	LEFT OUTER JOIN	Use keys from the left frame only.
`right`	RIGHT OUTER JOIN	Use keys from the right frame only.
`outer`	FULL OUTER JOIN	Use a union of keys from both frames.
`inner`	INNER JOIN	Use an intersection of keys from both frames.



Reshaping data

We saw how to combine data frames but sometimes we have all the right data in a single data structure, but the format is impractical for certain tasks. We start again with some artificial weather data:

```
>>> df
          date    city  value
0   2000-01-03  London      6
1   2000-01-04  London      3
2   2000-01-05  London      4
3   2000-01-03  Mexico      3
4   2000-01-04  Mexico      9
5   2000-01-05  Mexico      8
6   2000-01-03  Mumbai     12
7   2000-01-04  Mumbai      9
8   2000-01-05  Mumbai      8
9   2000-01-03   Tokyo      5
10  2000-01-04   Tokyo      5
11  2000-01-05   Tokyo      6
```

If we want to calculate the maximum temperature per city, we could just group the data by city and then take the `max` function:

```
>>> df.groupby('city').max()
             date  value
city
London  2000-01-05      6
Mexico  2000-01-05      9
Mumbai  2000-01-05     12
Tokyo   2000-01-05      6
```

However, if we have to bring our data into form every time, we could be a little more effective, by creating a reshaped data frame first, having the dates as an index and the cities as columns.

We can create such a data frame with the `pivot` function. The arguments are the index (we use date), the columns (we use the cities), and the values (which are stored in the value column of the original data frame):

```
>>> pv = df.pivot("date", "city", "value")
>>> pv
city          London  Mexico  Mumbai  Tokyo
date
```

```
2000-01-03       6       3      12       5
2000-01-04       3       9       9       5
2000-01-05       4       8       8       6
```

We can use `max` function on this new data frame directly:

```
>>> pv.max()
city
London     6
Mexico     9
Mumbai    12
Tokyo      6
dtype: int64
```

With a more suitable shape, other operations become easier as well. For example, to find the maximum temperature per day, we can simply provide an additional axis argument:

```
>>> pv.max(axis=1)
date
2000-01-03    12
2000-01-04     9
2000-01-05     8
dtype: int64
```

Data aggregation

As a final topic, we will look at ways to get a condensed view of data with aggregations. pandas comes with a lot of aggregation functions built-in. We already saw the `describe` function in *Chapter 3*, *Data Analysis with pandas*. This works on parts of the data as well. We start with some artificial data again, containing measurements about the number of sunshine hours per city and date:

```
>>> df.head()
   country     city        date  hours
0  Germany  Hamburg  2015-06-01      8
1  Germany  Hamburg  2015-06-02     10
2  Germany  Hamburg  2015-06-03      9
3  Germany  Hamburg  2015-06-04      7
4  Germany  Hamburg  2015-06-05      3
```

To view a summary per `city`, we use the `describe` function on the grouped data set:

```
>>> df.groupby("city").describe()
                     hours
city
Berlin     count  10.000000
           mean    6.000000
           std     3.741657
           min     0.000000
           25%     4.000000
           50%     6.000000
           75%     9.750000
           max    10.000000
Birmingham count  10.000000
           mean    5.100000
           std     2.078995
           min     2.000000
           25%     4.000000
           50%     5.500000
           75%     6.750000
           max     8.000000
```

On certain data sets, it can be useful to group by more than one attribute. We can get an overview about the sunny hours per country and date by passing in two column names:

```
>>> df.groupby(["country", "date"]).describe()
                         hours
country date
France  2015-06-01 count  5.000000
                   mean   6.200000
                   std    1.095445
                   min    5.000000
                   25%    5.000000
                   50%    7.000000
                   75%    7.000000
                   max    7.000000
        2015-06-02 count  5.000000
                   mean   3.600000
```

```
                  std    3.577709
                  min    0.000000
                  25%    0.000000
                  50%    4.000000
                  75%    6.000000
                  max    8.000000
UK      2015-06-07 std    3.872983
                  min    0.000000
                  25%    2.000000
                  50%    6.000000
                  75%    8.000000
                  max    9.000000
```

We can compute single statistics as well:

```
>>> df.groupby("city").mean()
            hours
city
Berlin        6.0
Birmingham    5.1
Bordeax       4.7
Edinburgh     7.5
Frankfurt     5.8
Glasgow       4.8
Hamburg       5.5
Leipzig       5.0
London        4.8
Lyon          5.0
Manchester    5.2
Marseille     6.2
Munich        6.6
Nice          3.9
Paris         6.3
```

Finally, we can define any function to be applied on the groups with the `agg` method. The preceding code could have been written in terms of `agg` like this:

```
>>> df.groupby("city").agg(np.mean)
hours
city
Berlin        6.0
Birmingham    5.1
```

```
Bordeax       4.7
Edinburgh     7.5
Frankfurt     5.8
Glasgow       4.8
...
```

But arbitrary functions are possible. As a last example, we define a `custom` function, which takes an input of a series object and computes the difference between the smallest and the largest element:

```
>>> df.groupby("city").agg(lambda s: abs(min(s) - max(s)))
        hours
city
Berlin         10
Birmingham      6
Bordeax        10
Edinburgh       8
Frankfurt       9
Glasgow        10
Hamburg        10
Leipzig         9
London         10
Lyon            8
Manchester     10
Marseille      10
Munich          9
Nice           10
Paris           9
```

Grouping data

One typical workflow during data exploration looks as follows:

- You find a criterion that you want to use to group your data. Maybe you have GDP data for every country along with the continent and you would like to ask questions about the continents. These questions usually lead to some function applications- you might want to compute the mean GDP per continent. Finally, you want to store this data for further processing in a new data structure.

- We use a simpler example here. Imagine some fictional weather data about the number of sunny hours per day and city:

```
>>> df
          date    city  value
0   2000-01-03  London      6
1   2000-01-04  London      3
2   2000-01-05  London      4
3   2000-01-03  Mexico      3
4   2000-01-04  Mexico      9
5   2000-01-05  Mexico      8
6   2000-01-03  Mumbai     12
7   2000-01-04  Mumbai      9
8   2000-01-05  Mumbai      8
9   2000-01-03   Tokyo      5
10  2000-01-04   Tokyo      5
11  2000-01-05   Tokyo      6
```

  The `groups` attributes return a dictionary containing the unique groups and the corresponding values as axis labels:

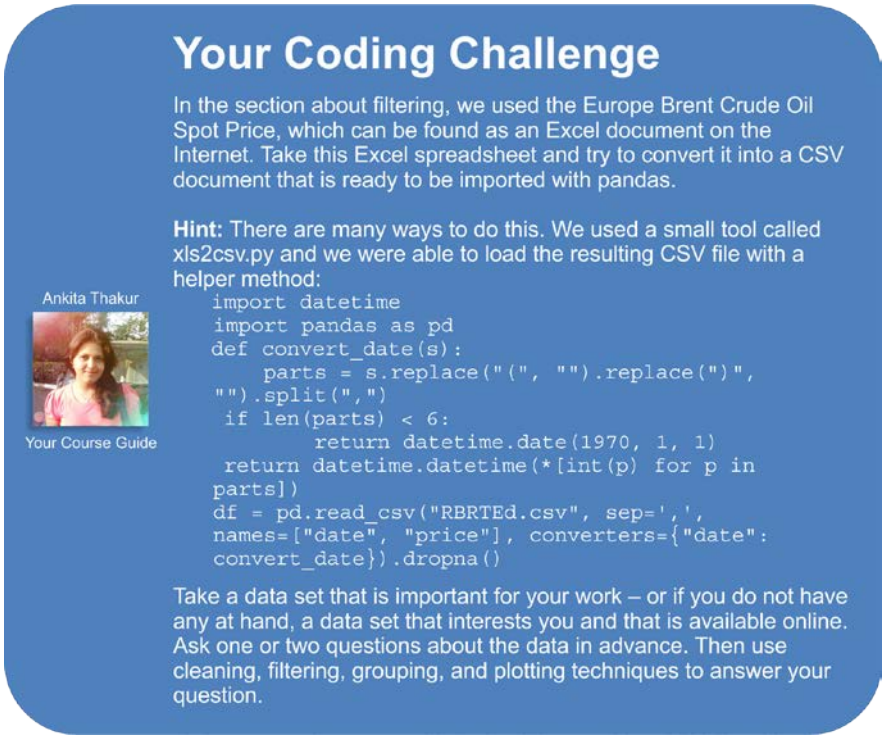
```
>>> df.groupby("city").groups
{'London': [0, 1, 2],
'Mexico': [3, 4, 5],
'Mumbai': [6, 7, 8],
'Tokyo': [9, 10, 11]}
```

- Although the result of a `groupby` is a GroupBy object, not a DataFrame, we can use the usual indexing notation to refer to columns:

```
>>> grouped = df.groupby(["city", "value"])
>>> grouped["value"].max()
city
London     6
Mexico     9
Mumbai    12
Tokyo      6
Name: value, dtype: int64
>>> grouped["value"].sum()
city
London    13
Mexico    20
Mumbai    29
Tokyo     16
Name: value, dtype: int64
```

- We see that, according to our data set, Mumbai seems to be a sunny city. An alternative – and more verbose – way to achieve the preceding code would be:

```
>>> df['value'].groupby(df['city']).sum()
city
London    13
Mexico    20
Mumbai    29
Tokyo     16
Name: value, dtype: int64
```



Summary of Module 2 Chapter 7

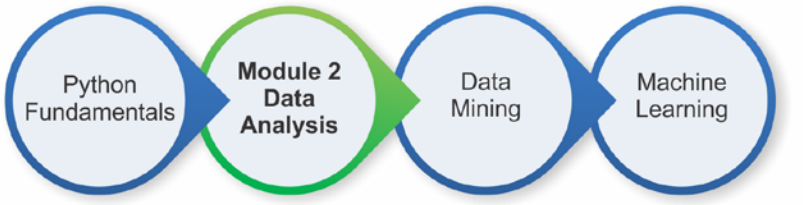
Ankita Thakur

Your Course Guide

In this chapter, we have looked at ways to manipulate data frames, from cleaning and filtering, to grouping, aggregation, and reshaping. pandas makes a lot of the common operations very easy and more complex operations, such as pivoting or grouping by multiple attributes, can often be expressed as one-liners as well. Cleaning and preparing data is an essential part of data exploration and analysis.

That's all and we've come to the end of this module. In the next module, we'll cover one of the field of data analysis—data mining. If you have ever wanted to get into data mining, but didn't know where to start, here you go!

Your Progress through the Course So Far



Course Module 3
Data Mining

Course Module 1: Python Fundamentals
Chapter 1: Introduction and First Steps – Take a Deep Breath
Chapter 2: Object-oriented Design
Chapter 3: Objects in Python
Chapter 4: When Objects are alike
Chapter 5: Expecting the Unexpected
Chapter 6: When to use Object-oriented programming
Chapter 7: Python Data Structures
Chapter 8: Python Object-oriented Shortcuts
Chapter 9: Strings and Serialization
Chapter 10: The Iterator Pattern
Chapter 11: Python Design Patterns I
Chapter 12: Python Design Patterns II
Chapter 13: Testing Object-oriented Programs
Chapter 14: Concurrency

Course Module 2: Data Analysis
Chapter 1: Introducing Data Analysis and Libraries
Chapter 2: NumPy Arrays and Vectorized Computation
Chapter 3: Data Analysis with pandas
Chapter 4: Data Visualizaiton
Chapter 5: Time Series
Chapter 6: Interacting with Databases
Chapter 7: Data Analysis Application Examples

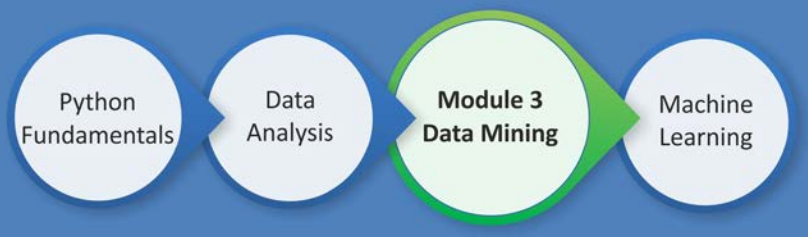
Course Module 3: Data Mining
Chapter 1: Getting Started with Data Mining
Chapter 2: Classifying with scikit-learn Estimators
Chapter 3: Predicting Sports Winners with Decision Trees
Chapter 4: Recommending Movies Using Affinity Analysis
Chapter 5: Extracting Features with Transformers
Chapter 6: Social Media Insight Using Naive Bayes
Chapter 7: Discovering Accounts to Follow Using Graph Mining
Chapter 8: Beating CAPTCHAs with Neural Networks
Chapter 9: Authorship Attribution
Chapter 10: Clustering News Articles
Chapter 11: Classifying Objects in Images Using Deep Learning
Chapter 12: Working with Big Data
Chapter 13: Next Steps…

It's time to do little mining ... Let's do this with Course Module 3, Data Mining

Course Module 4: Machine Learning

Course Module 3

The next step in the information age is to gain insights from the deluge of data coming our way. This is what we'll be learning in our third module, *Data Mining*.



Ankita Thakur

Your Course Guide

Data mining provides a way of finding this insight, and Python is one of the most popular languages for data mining, providing both power and flexibility in analysis. There is a rich and varied set of libraries available in Python for data mining.

This module will teaches us how to design and develop data mining applications using a variety of datasets, starting with basic classification and affinity analysis. Next, we move on to more complex data types including text, images, and graphs. In every Chapters, we we will be creating models to solve real-world problems.

There is a rich and varied set of libraries available in Python for data mining and this module covers most of them.

Each Chapters of this module will introduce us to new algorithms and techniques. By the end of the module, I'm sure we will gain a large insight into using Python for data mining with a good knowledge and understanding of the algorithms and implementations.

1
Getting Started with Data Mining

We are collecting information at a scale that has never been seen before in the history of mankind and placing more day-to-day importance on the use of this information in everyday life. We expect our computers to translate Web pages into other languages, predict the weather, suggest books we would like, and diagnose our health issues. These expectations will grow, both in the number of applications and also in the efficacy we expect. Data mining is a methodology that we can employ to train computers to make decisions with data and forms the backbone of many high-tech systems of today.

The Python language is fast growing in popularity, for a good reason. It gives the programmer a lot of flexibility; it has a large number of modules to perform different tasks; and Python code is usually more readable and concise than in any other languages. There is a large and an active community of researchers, practitioners, and beginners using Python for data mining.

In this chapter, we will introduce data mining with Python. We will cover the following topics:

- What is data mining and where can it be used?
- Setting up a Python-based environment to perform data mining
- An example of affinity analysis, recommending products based on purchasing habits
- An example of (a classic) classification problem, predicting the plant species based on its measurement

Introducing data mining

Data mining provides a way for a computer to learn how to make decisions with data. This decision could be predicting tomorrow's weather, blocking a spam e-mail from entering your inbox, detecting the language of a website, or finding a new romance on a dating site. There are many different applications of data mining, with new applications being discovered all the time.

Data mining is part of algorithms, statistics, engineering, optimization, and computer science. We also use concepts and knowledge from other fields such as linguistics, neuroscience, or town planning. Applying it effectively usually requires this domain-specific knowledge to be integrated with the algorithms.

Most data mining applications work with the same high-level view, although the details often change quite considerably. We start our data mining process by creating a dataset, describing an aspect of the real world. Datasets comprise of two aspects:

- Samples that are objects in the real world. This can be a book, photograph, animal, person, or any other object.
- Features that are descriptions of the samples in our dataset. Features could be the length, frequency of a given word, number of legs, date it was created, and so on.

The next step is tuning the data mining algorithm. Each data mining algorithm has parameters, either within the algorithm or supplied by the user. This tuning allows the algorithm to learn how to make decisions about the data.

As a simple example, we may wish the computer to be able to categorize people as "short" or "tall". We start by collecting our dataset, which includes the heights of different people and whether they are considered short or tall:

Person	**Height**	**Short or tall?**
1	155cm	Short
2	165cm	Short
3	175cm	Tall
4	185cm	Tall

The next step involves tuning our algorithm. As a simple algorithm; if the height is more than x, the person is tall, otherwise they are short. Our training algorithm will then look at the data and decide on a good value for x. For the preceding dataset, a reasonable value would be 170 cm. Anyone taller than 170 cm is considered tall by the algorithm. Anyone else is considered short.

In the preceding dataset, we had an obvious feature type. We wanted to know if people are short or tall, so we collected their heights. This engineering feature is an important problem in data mining. In later chapters, we will discuss methods for choosing good features to collect in your dataset. Ultimately, this step often requires some expert domain knowledge or at least some trial and error.

In this module, we will introduce data mining through Python. In some cases, we choose clarity of code and workflows, rather than the most optimized way to do this. This sometimes involves skipping some details that can improve the algorithm's speed or effectiveness.

A simple affinity analysis example

In this section, we jump into our first example. A common use case for data mining is to improve sales by asking a customer who is buying a product if he/she would like another similar product as well. This can be done through affinity analysis, which is the study of when things exist together.

What is affinity analysis?

Affinity analysis is a type of data mining that gives similarity between samples (objects). This could be the similarity between the following:

- users on a website, in order to provide varied services or targeted advertising
- items to sell to those users, in order to provide recommended movies or products
- human genes, in order to find people that share the same ancestors

We can measure affinity in a number of ways. For instance, we can record how frequently two products are purchased together. We can also record the accuracy of the statement when a person buys object 1 and also when they buy object 2. Other ways to measure affinity include computing the similarity between samples, which we will cover in later chapters.

Product recommendations

One of the issues with moving a traditional business online, such as commerce, is that tasks that used to be done by humans need to be automated in order for the online business to scale. One example of this is up-selling, or selling an extra item to a customer who is already buying. Automated product recommendations through data mining are one of the driving forces behind the e-commerce revolution that is turning billions of dollars per year into revenue.

In this example, we are going to focus on a basic product recommendation service. We design this based on the following idea: when two items are historically purchased together, they are more likely to be purchased together in the future. This sort of thinking is behind many product recommendation services, in both online and offline businesses.

A very simple algorithm for this type of product recommendation algorithm is to simply find any historical case where a user has brought an item and to recommend other items that the historical user brought. In practice, simple algorithms such as this can do well, at least better than choosing random items to recommend. However, they can be improved upon significantly, which is where data mining comes in.

To simplify the coding, we will consider only two items at a time. As an example, people may buy bread and milk at the same time at the supermarket. In this early example, we wish to find simple rules of the form:

If a person buys product X, then they are likely to purchase product Y

More complex rules involving multiple items will not be covered such as people buying sausages and burgers being more likely to buy tomato sauce.

Loading the dataset with NumPy

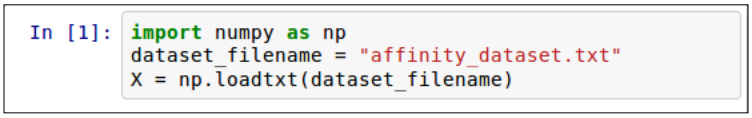
The dataset can be downloaded from the code package supplied with the course. Download this file and save it on your computer, noting the path to the dataset. For this example, I recommend that you create a new folder on your computer to put your dataset and code in. From here, open your IPython Notebook, navigate to this folder, and create a new notebook.

The dataset we are going to use for this example is a NumPy two-dimensional array, which is a format that underlies most of the examples in the rest of the module. The array looks like a table, with rows representing different samples and columns representing different features.

The cells represent the value of a particular feature of a particular sample. To illustrate, we can load the dataset with the following code:

```
import numpy as np
dataset_filename = "affinity_dataset.txt"
X = np.loadtxt(dataset_filename)
```

For this example, run the IPython Notebook and create an IPython Notebook. Enter the above code into the first cell of your Notebook. You can then run the code by pressing *Shift* + *Enter* (which will also add a new cell for the next lot of code). After the code is run, the square brackets to the left-hand side of the first cell will be assigned an incrementing number, letting you know that this cell has been completed. The first cell should look like the following:



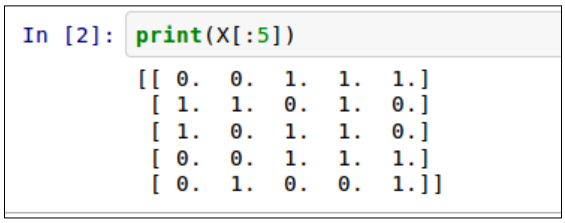
For later code that will take more time to run, an asterisk will be placed here to denote that this code is either running or scheduled to be run. This asterisk will be replaced by a number when the code has completed running.

You will need to save the dataset into the same directory as the IPython Notebook. If you choose to store it somewhere else, you will need to change the `dataset_filename` value to the new location.

Next, we can show some of the rows of the dataset to get a sense of what the dataset looks like. Enter the following line of code into the next cell and run it, in order to print the first five lines of the dataset:

```
print(X[:5])
```

The result will show you which items were bought in the first five transactions listed:



The dataset can be read by looking at each row (horizontal line) at a time. The first row `(0, 0, 1, 1, 1)` shows the items purchased in the first transaction. Each column (vertical row) represents each of the items. They are bread, milk, cheese, apples, and bananas, respectively. Therefore, in the first transaction, the person bought cheese, apples, and bananas, but not bread or milk.

Each of these features contain binary values, stating only whether the items were purchased and not how many of them were purchased. A 1 indicates that "at least 1" item was bought of this type, while a 0 indicates that absolutely none of that item was purchased.

Implementing a simple ranking of rules

We wish to find rules of the type *If a person buys product X, then they are likely to purchase product Y*. We can quite easily create a list of all of the rules in our dataset by simply finding all occasions when two products were purchased together. However, we then need a way to determine good rules from bad ones. This will allow us to choose specific products to recommend.

Rules of this type can be measured in many ways, of which we will focus on two: **support** and **confidence**.

Support is the number of times that a rule occurs in a dataset, which is computed by simply counting the number of samples that the rule is valid for. It can sometimes be normalized by dividing by the total number of times the premise of the rule is valid, but we will simply count the total for this implementation.

While the support measures how often a rule exists, confidence measures how accurate they are when they can be used. It can be computed by determining the percentage of times the rule applies when the premise applies. We first count how many times a rule applies in our dataset and divide it by the number of samples where the premise (the `if` statement) occurs.

As an example, we will compute the support and confidence for the rule *if a person buys apples, they also buy bananas*.

As the following example shows, we can tell whether someone bought apples in a transaction by checking the value of `sample[3]`, where a sample is assigned to a row of our matrix:

```
In [9]: # First, how many rows contain our premise: that a person is buying apples
        num_apple_purchases = 0
        for sample in X:
            if sample[3] == 1:  # This person bought Apples
                num_apple_purchases += 1
        print("{0} people bought Apples".format(num_apple_purchases))

        36 people bought Apples
```

Similarly, we can check if bananas were bought in a transaction by seeing if the value for `sample[4]` is equal to 1 (and so on). We can now compute the number of times our rule exists in our dataset and, from that, the confidence and support.

Now we need to compute these statistics for all rules in our database. We will do this by creating a dictionary for both *valid rules* and *invalid rules*. The key to this dictionary will be a tuple (premise and conclusion). We will store the indices, rather than the actual feature names. Therefore, we would store (3 and 4) to signify the previous rule *If a person buys Apples, they will also buy Bananas*. If the premise and conclusion are given, the rule is considered valid. While if the premise is given but the conclusion is not, the rule is considered invalid for that sample.

To compute the confidence and support for all possible rules, we first set up some dictionaries to store the results. We will use `defaultdict` for this, which sets a default value if a key is accessed that doesn't yet exist. We record the number of valid rules, invalid rules, and occurrences of each premise:

```
from collections import defaultdict
valid_rules = defaultdict(int)
invalid_rules = defaultdict(int)
num_occurances = defaultdict(int)
```

Next we compute these values in a large loop. We iterate over each sample and feature in our dataset. This first feature forms the premise of the rule—if a person buys a product premise:

```
for sample in X:
  for premise in range(4):
```

We check whether the premise exists for this sample. If not, we do not have any more processing to do on this sample/premise combination, and move to the next iteration of the loop:

```
    if sample[premise] == 0: continue
```

If the premise is valid for this sample (it has a value of 1), then we record this and check each conclusion of our rule. We skip over any conclusion that is the same as the premise—this would give us rules such as If a person buys Apples, then they buy Apples, which obviously doesn't help us much;

```
num_occurances[premise] += 1
for conclusion in range(n_features):
    if premise == conclusion: continue
```

If the conclusion exists for this sample, we increment our valid count for this rule. If not, we increment our invalid count for this rule:

```
if sample[conclusion] == 1:
  valid_rules[(premise, conclusion)] += 1
  else:
  invalid_rules[(premise, conclusion)] += 1
```

We have now completed computing the necessary statistics and can now compute the *support* and *confidence* for each rule. As before, the support is simply our `valid_rules` value:

```
support = valid_rules
```

The confidence is computed in the same way, but we must loop over each rule to compute this:

```
confidence = defaultdict(float)
for premise, conclusion in valid_rules.keys():
    rule = (premise, conclusion)
    confidence[rule] = valid_rules[rule] / num_occurances[premise]
```

We now have a dictionary with the support and confidence for each rule.
We can create a function that will print out the rules in a readable format.
The signature of the rule takes the premise and conclusion indices, the support and confidence dictionaries we just computed, and the features array that tells us what the features mean:

```
def print_rule(premise, conclusion,
               support, confidence, features):
```

We get the names of the features for the premise and conclusion and print out the rule in a readable format:

```
    premise_name = features[premise]
    conclusion_name = features[conclusion]
    print("Rule: If a person buys {0} they will also buy
      {1}".format(premise_name, conclusion_name))
```

Then we print out the `Support` and `Confidence` of this rule:

```
print(" - Support: {0}".format(support[(premise,
                                        conclusion)]))
print(" - Confidence: {0:.3f}".format(confidence[(premise,
                                                  conclusion)]))
```

We can test the code by calling it in the following way—feel free to experiment with different premises and conclusions:

```
In [31]: premise = 1
         conclusion = 3
         print_rule(premise, conclusion, support, confidence, features)

         Rule: If a person buys milk they will also buy apples
          - Confidence: 0.196
          - Support: 9
```

Ranking to find the best rules

Now that we can compute the support and confidence of all rules, we want to be able to find the *best* rules. To do this, we perform a ranking and print the ones with the highest values. We can do this for both the support and confidence values.

To find the rules with the highest support, we first sort the support dictionary. Dictionaries do not support ordering by default; the `items()` function gives us a list containing the data in the dictionary. We can sort this list using the `itemgetter` class as our key, which allows for the sorting of nested lists such as this one. Using `itemgetter(1)` allows us to sort based on the values. Setting `reverse=True` gives us the highest values first:

```
from operator import itemgetter
sorted_support = sorted(support.items(), key=itemgetter(1), re
verse=True)
```

We can then print out the top five rules:

```
for index in range(5):
    print("Rule #{0}".format(index + 1))
    premise, conclusion = sorted_support[index][0]
    print_rule(premise, conclusion, support, confidence, features)
```

The result will look like the following:

```
In [40]: for index in range(5):
             print("Rule #{0}".format(index + 1))
             (premise, conclusion) = sorted_support[index][0]
             print_rule(premise, conclusion, support, confidence, features)

Rule #1
Rule: If a person buys cheese they will also buy bananas
 - Confidence: 0.659
 - Support: 27

Rule #2
Rule: If a person buys bananas they will also buy cheese
 - Confidence: 0.458
 - Support: 27

Rule #3
Rule: If a person buys apples they will also buy cheese
 - Confidence: 0.694
 - Support: 25

Rule #4
Rule: If a person buys cheese they will also buy apples
 - Confidence: 0.610
 - Support: 25

Rule #5
Rule: If a person buys bananas they will also buy apples
 - Confidence: 0.356
 - Support: 21
```

Similarly, we can print the top rules based on confidence. First, compute the sorted confidence list:

```
sorted_confidence = sorted(confidence.items(), key=itemgetter(1),
reverse=True)
```

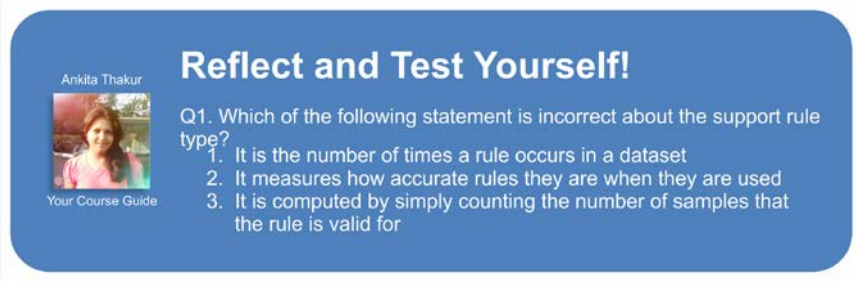
Next, print them out using the same method as before. Note the change to `sorted_confidence` on the third line;

```
for index in range(5):
    print("Rule #{0}".format(index + 1))
    premise, conclusion = sorted_confidence[index][0]
    print_rule(premise, conclusion, support, confidence, features)
```

```
In [42]: for index in range(5):
             print("Rule #{0}".format(index + 1))
             (premise, conclusion) = sorted_confidence[index][0]
             print_rule(premise, conclusion, support, confidence, features)

         Rule #1
         Rule: If a person buys apples they will also buy cheese
          - Confidence: 0.694
          - Support: 25

         Rule #2
         Rule: If a person buys cheese they will also buy bananas
          - Confidence: 0.659
          - Support: 27

         Rule #3
         Rule: If a person buys bread they will also buy bananas
          - Confidence: 0.630
          - Support: 17

         Rule #4
         Rule: If a person buys cheese they will also buy apples
          - Confidence: 0.610
          - Support: 25

         Rule #5
         Rule: If a person buys apples they will also buy bananas
          - Confidence: 0.583
          - Support: 21
```

Two rules are near the top of both lists. The first is **If a person buys apples, they will also buy cheese**, and the second is **If a person buys cheese, they will also buy bananas**. A store manager can use rules like these to organize their store. For example, if apples are on sale this week, put a display of cheeses nearby. Similarly, it would make little sense to put both bananas on sale at the same time as cheese, as nearly 66 percent of people buying cheese will buy bananas anyway—our sale won't increase banana purchases all that much.

Data mining has great exploratory power in examples like this. A person can use data mining techniques to explore relationships within their datasets to find new insights. In the next section, we will use data mining for a different purpose: prediction.



A simple classification example

In the affinity analysis example, we looked for correlations between different variables in our dataset. In classification, we instead have a single variable that we are interested in and that we call the **class** (also called the target). If, in the previous example, we were interested in how to make people buy more apples, we could set that variable to be the class and look for classification rules that obtain that goal. We would then look only for rules that relate to that goal.

What is classification?

Classification is one of the largest uses of data mining, both in practical use and in research. As before, we have a set of samples that represents objects or things we are interested in classifying. We also have a new array, the class values. These class values give us a categorization of the samples. Some examples are as follows:

- Determining the species of a plant by looking at its measurements. The class value here would be *Which species is this?*.
- Determining if an image contains a dog. The class would be *Is there a dog in this image?*.
- Determining if a patient has cancer based on the test results. The class would be *Does this patient have cancer?*.

While many of the examples above are binary (yes/no) questions, they do not have to be, as in the case of plant species classification in this section.

The goal of classification applications is to train a model on a set of samples with known classes, and then apply that model to new unseen samples with unknown classes. For example, we want to train a spam classifier on my past e-mails, which I have labeled as spam or not spam. I then want to use that classifier to determine whether my next e-mail is spam, without me needing to classify it myself.

Loading and preparing the dataset

The dataset we are going to use for this example is the famous Iris database of plant classification. In this dataset, we have 150 plant samples and four measurements of each: **sepal length**, **sepal width**, **petal length**, and **petal width** (all in centimeters). This classic dataset (first used in 1936!) is one of the classic datasets for data mining. There are three classes: **Iris Setosa**, **Iris Versicolour**, and **Iris Virginica**. The aim is to determine which type of plant a sample is, by examining its measurements.

The `scikit-learn` library contains this dataset built-in, making the loading of the dataset straightforward:

```
from sklearn.datasets import load_iris
dataset = load_iris()
X = dataset.data
y = dataset.target
```

You can also print(`dataset.DESCR`) to see an outline of the dataset, including some details about the features.

The features in this dataset are continuous values, meaning they can take any range of values. Measurements are a good example of this type of feature, where a measurement can take the value of 1, 1.2, or 1.25 and so on. Another aspect about continuous features is that feature values that are close to each other indicate similarity. A plant with a sepal length of 1.2 cm is like a plant with sepal width of 1.25 cm.

In contrast are categorical features. These features, while often represented as numbers, cannot be compared in the same way. In the Iris dataset, the class values are an example of a categorical feature. The class 0 represents Iris Setosa, class 1 represents Iris Versicolour, and class 2 represents Iris Virginica. This doesn't mean that Iris Setosa is more similar to Iris Versicolour than it is to Iris Virginica—despite the class value being more similar. The numbers here represent categories. All we can say is whether categories are the same or different.

There are other types of features too, some of which will be covered in later chapters.

While the features in this dataset are continuous, the algorithm we will use in this example requires categorical features. Turning a continuous feature into a categorical feature is a process called discretization.

A simple discretization algorithm is to choose some threshold and any values below this threshold are given a value 0. Meanwhile any above this are given the value 1. For our threshold, we will compute the mean (average) value for that feature. To start with, we compute the mean for each feature:

```
attribute_means = X.mean(axis=0)
```

This will give us an array of length 4, which is the number of features we have. The first value is the mean of the values for the first feature and so on. Next, we use this to transform our dataset from one with continuous features to one with discrete categorical features:

```
X_d = np.array(X >= attribute_means, dtype='int')
```

We will use this new `X_d` dataset (for *X discretized*) for our training and testing, rather than the original dataset (*X*).

Implementing the OneR algorithm

OneR is a simple algorithm that simply predicts the class of a sample by finding the most frequent class for the feature values. OneR is a shorthand for *One Rule*, indicating we only use a single rule for this classification by choosing the feature with the best performance. While some of the later algorithms are significantly more complex, this simple algorithm has been shown to have good performance in a number of real-world datasets.

The algorithm starts by iterating over every value of every feature. For that value, count the number of samples from each class that have that feature value. Record the most frequent class for the feature value, and the error of that prediction.

For example, if a feature has two values, *0* and *1*, we first check all samples that have the value *0*. For that value, we may have 20 in class *A*, 60 in class *B*, and a further 20 in class *C*. The most frequent class for this value is *B*, and there are 40 instances that have difference classes. The prediction for this feature value is *B* with an error of 40, as there are 40 samples that have a different class from the prediction. We then do the same procedure for the value *1* for this feature, and then for all other feature value combinations.

Once all of these combinations are computed, we compute the error for each feature by summing up the errors for all values for that feature. The feature with the lowest total error is chosen as the *One Rule* and then used to classify other instances.

In code, we will first create a function that computes the class prediction and error for a specific feature value. We have two necessary imports, `defaultdict` and `itemgetter`, that we used in earlier code:

```
from collections import defaultdict
from operator import itemgetter
```

Next, we create the function definition, which needs the dataset, classes, the index of the feature we are interested in, and the value we are computing:

```
def train_feature_value(X, y_true, feature_index, value):
```

We then iterate over all the samples in our dataset, counting the actual classes for each sample with that feature value:

```
    class_counts = defaultdict(int)
    for sample, y in zip(X, y_true):
        if sample[feature_index] == value:
            class_counts[y] += 1
```

We then find the most frequently assigned class by sorting the `class_counts` dictionary and finding the highest value:

```
    sorted_class_counts = sorted(class_counts.items(),
      key=itemgetter(1), reverse=True)
        most_frequent_class = sorted_class_counts[0][0]
```

Finally, we compute the error of this rule. In the OneR algorithm, any sample with this feature value would be predicted as being the most frequent class. Therefore, we compute the error by summing up the counts for the other classes (not the most frequent). These represent training samples that this rule does not work on:

```
incorrect_predictions = [class_count for class_value, class_count
in class_counts.items()
if class_value != most_frequent_class]
error = sum(incorrect_predictions)
```

Finally, we return both the predicted class for this feature value and the number of incorrectly classified training samples, the error, of this rule:

```
return most_frequent_class, error
```

With this function, we can now compute the error for an entire feature by looping over all the values for that feature, summing the errors, and recording the predicted classes for each value.

The function header needs the dataset, classes, and feature index we are interested in:

```
def train_on_feature(X, y_true, feature_index):
```

Next, we find all of the unique values that the given feature takes. The indexing in the next line looks at the whole column for the given feature and returns it as an array. We then use the set function to find only the unique values:

```
values = set(X[:,feature_index])
```

Next, we create our dictionary that will store the predictors. This dictionary will have feature values as the keys and classification as the value. An entry with key 1.5 and value 2 would mean that, when the feature has value set to 1.5, classify it as belonging to class 2. We also create a list storing the errors for each feature value:

```
predictors = {}
errors = []
```

As the main section of this function, we iterate over all the unique values for this feature and use our previously defined `train_feature_value()` function to find the most frequent class and the error for a given feature value. We store the results as outlined above:

```
for current_value in values:
  most_frequent_class, error = train_feature_value(X,
    y_true, feature_index, current_value)
  predictors[current_value] = most_frequent_class
  errors.append(error)
```

Finally, we compute the total errors of this rule and return the predictors along with this value:

```
total_error = sum(errors)
return predictors, total_error
```

Testing the algorithm

When we evaluated the affinity analysis algorithm of the last section, our aim was to explore the current dataset. With this classification, our problem is different. We want to build a model that will allow us to classify previously unseen samples by comparing them to what we know about the problem.

For this reason, we split our machine-learning workflow into two stages: training and testing. In training, we take a portion of the dataset and create our model. In testing, we apply that model and evaluate how effectively it worked on the dataset. As our goal is to create a model that is able to classify previously unseen samples, we cannot use our testing data for training the model. If we do, we run the risk of overfitting.

Overfitting is the problem of creating a model that classifies our training dataset very well, but performs poorly on new samples. The solution is quite simple: never use training data to test your algorithm. This simple rule has some complex variants, which we will cover in later chapters; but, for now, we can evaluate our OneR implementation by simply splitting our dataset into two small datasets: a training one and a testing one. This workflow is given in this section.

The `scikit-learn` library contains a function to split data into training and testing components:

```
from sklearn.cross_validation import train_test_split
```

This function will split the dataset into two subdatasets, according to a given ratio (which by default uses 25 percent of the dataset for testing). It does this randomly, which improves the confidence that the algorithm is being appropriately tested:

```
Xd_train, Xd_test, y_train, y_test = train_test_split(X_d, y, random_
state=14)
```

We now have two smaller datasets: `Xd_train` contains our data for training and `Xd_test` contains our data for testing. `y_train` and `y_test` give the corresponding class values for these datasets.

We also specify a specific `random_state`. Setting the random state will give the same split every time the same value is entered. It will *look* random, but the algorithm used is deterministic and the output will be consistent. For this module, I recommend setting the random state to the same value that I do, as it will give you the same results that I get, allowing you to verify your results. To get truly random results that change every time you run it, set `random_state` to `none`.

Next, we compute the predictors for all the features for our dataset. Remember to only use the training data for this process. We iterate over all the features in the dataset and use our previously defined functions to train the predictors and compute the errors:

```
all_predictors = {}
errors = {}
for feature_index in range(Xd_train.shape[1]):
  predictors, total_error = train_on_feature(Xd_train, y_train,
    feature_index)
  all_predictors[feature_index] = predictors
  errors[feature_index] = total_error
```

Next, we find the best feature to use as our "One Rule", by finding the feature with the lowest error:

```
best_feature, best_error = sorted(errors.items(), key=itemgetter(1))
[0]
```

We then create our `model` by storing the predictors for the best feature:

```
model = {'feature': best_feature,
  'predictor': all_predictors[best_feature][0]}
```

Our model is a dictionary that tells us which feature to use for our *One Rule* and the predictions that are made based on the values it has. Given this model, we can predict the class of a previously unseen sample by finding the value of the specific feature and using the appropriate predictor. The following code does this for a given sample:

```
variable = model['variable']
predictor = model['predictor']
prediction = predictor[int(sample[variable])]
```

Often we want to predict a number of new samples at one time, which we can do using the following function; we use the above code, but iterate over all the samples in a dataset, obtaining the prediction for each sample:

```
def predict(X_test, model):
    variable = model['variable']
    predictor = model['predictor']
    y_predicted = np.array([predictor[int(sample[variable])] for
      sample in X_test])
    return y_predicted
```

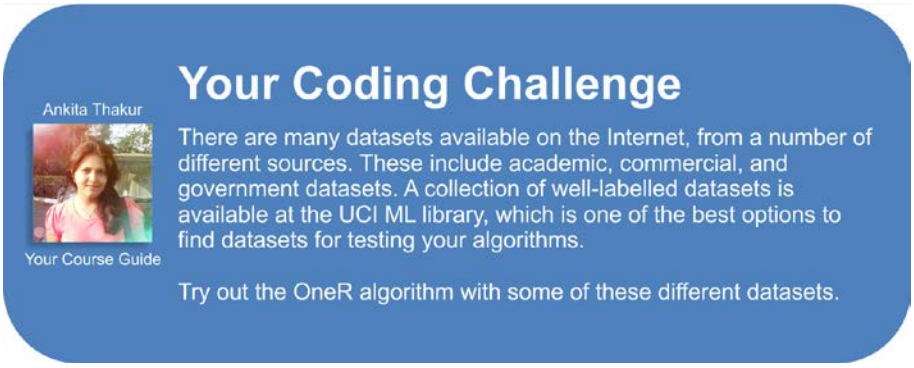
For our `testing` dataset, we get the predictions by calling the following function:

```
y_predicted = predict(X_test, model)
```

We can then compute the accuracy of this by comparing it to the known classes:

```
accuracy = np.mean(y_predicted == y_test) * 100
print("The test accuracy is {:.1f}%".format(accuracy))
```

This gives an accuracy of 68 percent, which is not bad for a single rule!



Summary of Module 3 Chapter 1

Ankita Thakur

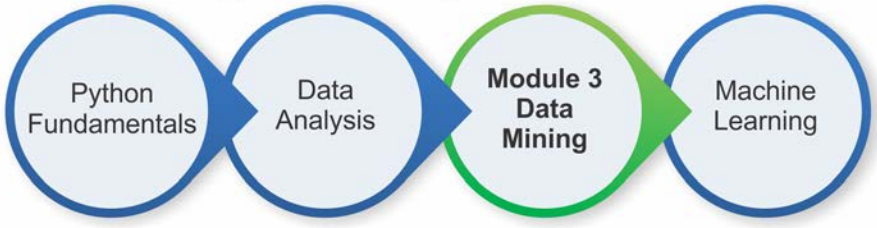
Your Course Guide

We introduced a simple affinity analysis, finding products that are purchased together. This type of exploratory analysis gives an insight into a business process, an environment, or a scenario. The information from these types of analysis can assist in business processes, finding the next big medical breakthrough, or creating the next artificial intelligence.

Also, in this chapter, there was a simple classification example using the OneR algorithm. This simple algorithm simply finds the best feature and predicts the class that most frequently had this value in the training dataset.

Over the next few chapters, we will expand on the concepts of classification and affinity analysis. We will also introduce the scikit-learn package and the algorithms it includes.

Your Progress through the Course So Far



2
Classifying with scikit-learn Estimators

The `scikit-learn` library is a collection of data mining algorithms, written in Python and using a common programming interface. This allows users to easily try different algorithms as well as utilize standard tools for doing effective testing and parameter searching. There are a large number of algorithms and utilities in scikit-learn.

In this chapter, we focus on setting up a good framework for running data mining procedures. This will be used in later chapters, which are all focused on applications and techniques to use in those situations.

The key concepts introduced in this chapter are as follows:

- **Estimators**: This is to perform classification, clustering, and regression
- **Transformers**: This is to perform preprocessing and data alterations
- **Pipelines**: This is to put together your workflow into a replicable format

scikit-learn estimators

Estimators are `scikit-learn's` abstraction, allowing for the standardized implementation of a large number of classification algorithms. Estimators are used for classification. Estimators have the following two main functions:

- `fit()`: This performs the training of the algorithm and sets internal parameters. It takes two inputs, the training sample dataset and the corresponding classes for those samples.
- `predict()`: This predicts the class of the testing samples that is given as input. This function returns an array with the predictions of each input testing sample.

Most `scikit-learn` estimators use the `NumPy` arrays or a related format for input and output.

There are a large number of estimators in scikit-learn. These include **support vector machines** (**SVM**), **random forests**, and **neural networks**. Many of these algorithms will be used in later chapters. In this chapter, we will use a different estimator from `scikit-learn`: **nearest neighbor**.

For this chapter, you will need to install a new library called `matplotlib`. The easiest way to install it is to use `pip3`, as you did in *Chapter 1*, *Getting Started with Data Mining*, to install `scikit-learn`:

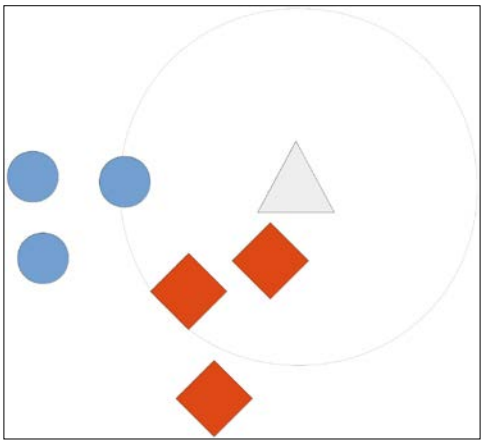
```
$pip3 install matplotlib
```

If you have any difficulty installing `matplotlib`, seek the official installation instructions at `http://matplotlib.org/users/installing.html`.

Nearest neighbors

Nearest neighbors is perhaps one of the most intuitive algorithms in the set of standard data mining algorithms. To predict the class of a new sample, we look through the training dataset for the samples that are most similar to our new sample. We take the most similar sample and predict the class that the majority of those samples have.

As an example, we wish to predict the class of the triangle, based on which class it is more similar to (represented here by having similar objects closer together). We seek the three nearest neighbors, which are two diamonds and one square. There are more diamonds than circles, and the predicted class for the triangle is, therefore, a diamond:



Nearest neighbors can be used for nearly any dataset-however, it can be very computationally expensive to compute the distance between all pairs of samples. For example if there are 10 samples in the dataset, there are 45 unique distances to compute. However, if there are 1000 samples, there are nearly 500,000! Various methods exist for improving this speed dramatically; some of which are covered in the later chapters of this module.

It can also do poorly in categorical-based datasets, and another algorithm should be used for these instead.

Distance metrics

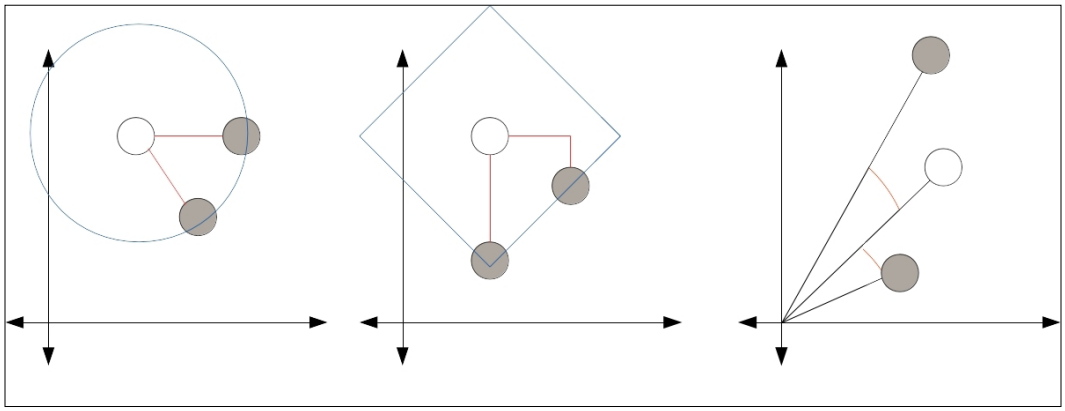
A key underlying concept in data mining is that of distance. If we have two samples, we need to know how close they are to each other. Further more, we need to answer questions such as are these two samples more similar than the other two? Answering questions like these is important to the outcome of the case.

The most common distance metric that the people are aware of is **Euclidean** distance, which is the *real-world* distance. If you were to plot the points on a graph and measure the distance with a straight ruler, the result would be the Euclidean distance. A little more formally, it is the square root of the sum of the squared distances for each feature.

Euclidean distance is intuitive, but provides poor accuracy if some features have larger values than others. It also gives poor results when lots of features have a value of 0, known as a sparse matrix. There are other distance metrics in use; two commonly employed ones are the Manhattan and Cosine distance.

The **Manhattan** distance is the sum of the absolute differences in each feature (with no use of square distances). Intuitively, it can be thought of as the number of moves a rook piece (or castle) in chess would take to move between the points, if it were limited to moving one square at a time. While the Manhattan distance does suffer if some features have larger values than others, the effect is not as dramatic as in the case of Euclidean.

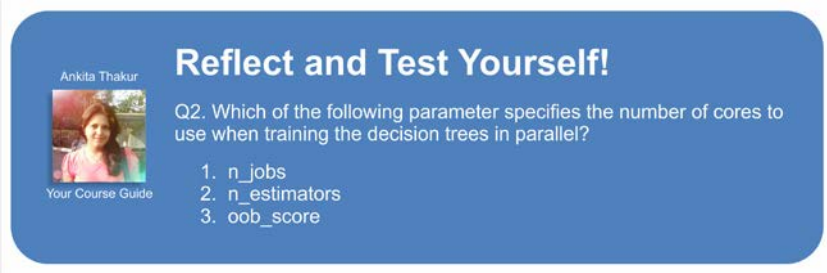
The **Cosine** distance is better suited to cases where some features are larger than others and when there are lots of zeros in the dataset. Intuitively, we draw a line from the origin to each of the samples, and measure the angle between those lines. This can be seen in the following diagram:



In this example, each of the grey circles are in the same distance from the white circle. In (a), the distances are Euclidean, and therefore, similar distances fit around a circle. This distance can be measured using a ruler. In (b), the distances are Manhattan, also called City Block. We compute the distance by moving across rows and columns, similar to how a Rook (Castle) in Chess moves. Finally, in (c), we have the Cosine distance that is measured by computing the angle between the lines drawn from the sample to the vector, and ignore the actual length of the line.

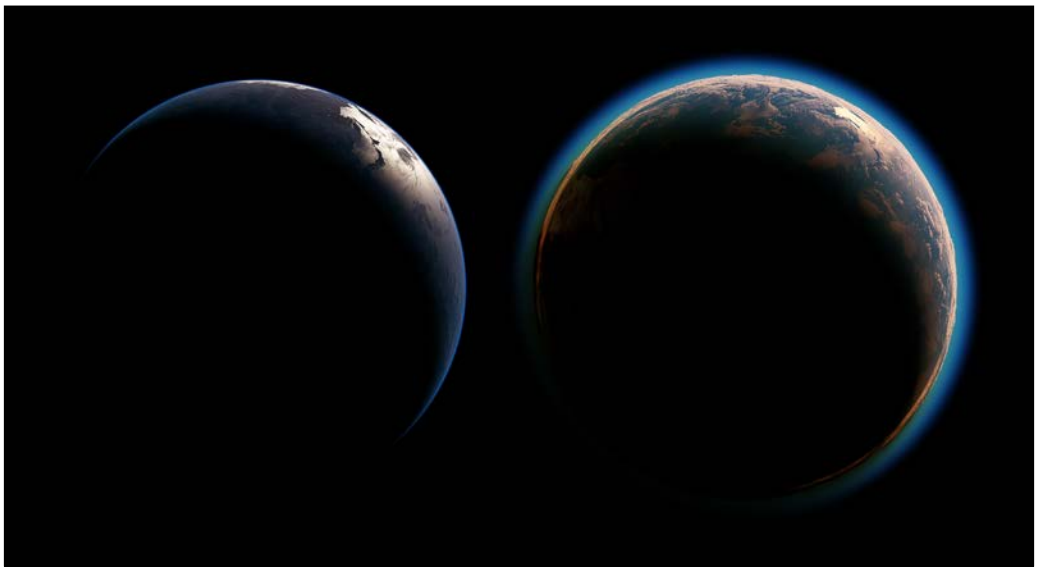
The distance metric chosen can have a large impact on the final performance. For example, if you have many features, the Euclidean distance between random samples approaches the same value. This makes it hard to compare samples as the distances are the same! Manhattan distance can be more stable in some circumstances, but if some features have very large values, this can *overrule* lots of similarities in other features. Finally, Cosine distance is a good metric for comparing items with a large number of features, but it discards some information about the length of the vector, which is useful in some circumstances.

For this chapter, we will stay with Euclidean distance, using other metrics in later chapters.



Loading the dataset

The dataset we are going to use is called *Ionosphere*, which is the recording of many high-frequency antennas. The aim of the antennas is to determine whether there is a structure in the ionosphere and a region in the upper atmosphere. Those that have a structure are deemed good, while those that do not are deemed bad. The aim of this application is to build a data mining classifier that can determine whether an image is good or bad.



(Image Credit: `https://www.flickr.com/photos/geckzilla/16149273389/`)

This can be downloaded from the UCL Machine Learning data repository, which contains a large number of datasets for different data mining applications. Go to `http://archive.ics.uci.edu/ml/datasets/Ionosphere` and click on **Data Folder**. Download the `ionosphere.data` and `ionosphere.names` files to a folder on your computer. For this example, I'll assume that you have put the dataset in a directory called Data in your home folder.

The location of your home folder depends on your operating system. For Windows, it is usually at `C:\Documents and Settings\username`. For Mac or Linux machines, it is usually at `/home/username`. You can get your home folder by running this python code:

```
import os
print(os.path.expanduser("~"))
```

For each row in the dataset, there are 35 values. The first 34 are measurements taken from the 17 antennas (two values for each antenna). The last is either 'g' or 'b'; that stands for good and bad, respectively.

Start the IPython Notebook server and create a new notebook called **Ionosphere Nearest Neighbors** for this chapter.

First, we load up the `NumPy` and `csv` libraries that we will need for our code:

```
import numpy as np
import csv
```

To load the dataset, we first get the filename of the dataset. First, get the folder the dataset is stored in from your data folder:

```
data_filename = os.path.join(data_folder, "Ionosphere",
"ionosphere.data")
```

We then create the `X` and `y` NumPy arrays to store the dataset in. The sizes of these arrays are known from the dataset. Don't worry if you don't know the size of future datasets—we will use other methods to load the dataset in future chapters and you won't need to know this size beforehand:

```
X = np.zeros((351, 34), dtype='float')
y = np.zeros((351,), dtype='bool')
```

The dataset is in a **Comma-Separated Values** (**CSV**) format, which is a commonly used format for datasets. We are going to use the `csv` module to load this file. Import it and set up a `csv` reader object:

```
with open(data_filename, 'r') as input_file:
    reader = csv.reader(input_file)
```

Next, we loop over the lines in the file. Each line represents a new set of measurements, which is a sample in this dataset. We use the enumerate function to get the line's index as well, so we can update the appropriate sample in the dataset (`X`):

```
for i, row in enumerate(reader):
```

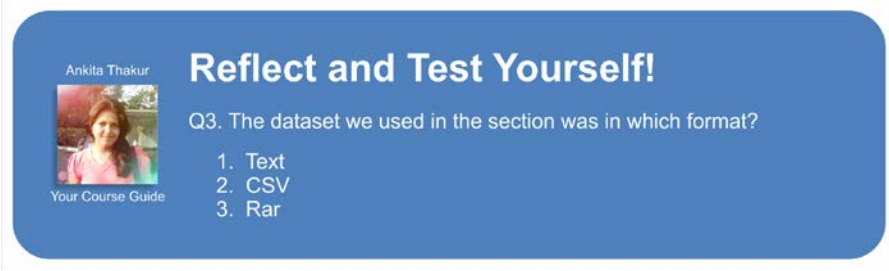
We take the first 34 values from this sample, turn each into a float, and save that to our dataset:

```
data = [float(datum) for datum in row[:-1]]
X[i] = data
```

Finally, we take the last value of the row and set the class. We set it to 1 (or `True`) if it is a good sample, and 0 if it is not:

```
y[i] = row[-1] == 'g'
```

We now have a dataset of samples and features in `X`, and the corresponding classes in `y`, as we did in the classification example in *Chapter 1, Getting Started with Data Mining*.



Moving towards a standard workflow

Estimators in `scikit-learn` have two main functions: `fit()` and `predict()`. We train the algorithm using the `fit` method and our training set. We evaluate it using the `predict` method on our testing set.

First, we need to create these training and testing sets. As before, import and run the `train_test_split` function:

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_
state=14)
```

Then, we import the nearest neighbor class and create an instance for it. We leave the parameters as defaults for now, and will choose good parameters later in this chapter. By default, the algorithm will choose the five nearest neighbors to predict the class of a testing sample:

```
from sklearn.neighbors import KNeighborsClassifier
estimator = KNeighborsClassifier()
```

After creating our estimator, we must then fit it on our training dataset. For the nearest neighbor class, this records our dataset, allowing us to find the nearest neighbor for a new data point, by comparing that point to the training dataset:

```
estimator.fit(X_train, y_train)
```

We then train the algorithm with our test set and evaluate with our testing set:

```
y_predicted = estimator.predict(X_test)
accuracy = np.mean(y_test == y_predicted) * 100
print("The accuracy is {0:.1f}%".format(accuracy))
```

This scores 86.4 percent accuracy, which is impressive for a default algorithm and just a few lines of code! Most `scikit-learn` default parameters are chosen explicitly to work well with a range of datasets. However, you should always aim to choose parameters based on knowledge of the application experiment.

Running the algorithm

In our earlier experiments, we set aside a portion of the dataset as a testing set, with the rest being the training set. We train our algorithm on the training set and evaluate how effective it will be based on the testing set. However, what happens if we get lucky and choose an easy testing set? Alternatively, what if it was particularly troublesome? We can discard a good model due to poor results resulting from such an "unlucky" split of our data.

The cross-fold validation framework is a way to address the problem of choosing a testing set and a standard methodology in data mining. The process works by doing a number of experiments with different training and testing splits, but using each sample in a testing set only once. The procedure is as follows:

1. Split the entire dataset into a number of sections called folds.
2. For each fold in the dataset, execute the following steps:
    - Set that fold aside as the current testing set
    - Train the algorithm on the remaining folds
    - Evaluate on the current testing set

3. Report on all the evaluation scores, including the average score.
4. In this process, each sample is used in the testing set only once. This reduces (but doesn't completely eliminate) the likelihood of choosing lucky testing sets.

Throughout this module, the code examples build upon each other within a chapter. Each chapter's code should be entered into the same `IPython` Notebook, unless otherwise specified.

The `scikit-learn` library contains a number of cross fold validation methods. A `helper` function is given that performs the preceding procedure. We can import it now in our IPython Notebook:

```
from sklearn.cross_validation import cross_val_score
```

By default, `cross_val_score` uses a specific methodology called **Stratified K Fold** to split the dataset into folds. This creates folds that have approximately the same proportion of classes in each fold, again reducing the likelihood of choosing poor folds. This is a great default, so we won't mess with it right now.

Next, we use this function, passing the original (full) dataset and classes:

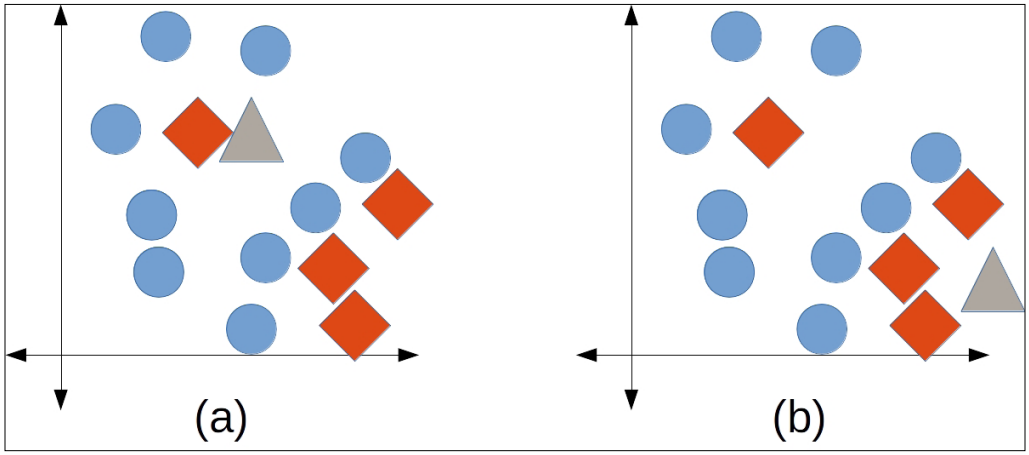
```
scores = cross_val_score(estimator, X, y, scoring='accuracy')
average_accuracy = np.mean(scores) * 100
print("The average accuracy is {0:.1f}%".format(average_accuracy))
```

This gives a slightly more modest result of 82.3 percent, but it is still quite good considering we have not yet tried setting better parameters. In the next section, we will see how we would go about changing the parameters to achieve a better outcome.

Setting parameters

Almost all data mining algorithms have parameters that the user can set. This is often a cause of generalizing an algorithm to allow it to be applicable in a wide variety of circumstances. Setting these parameters can be quite difficult, as choosing good parameter values is often highly reliant on features of the dataset.

The nearest neighbor algorithm has several parameters, but the most important one is that of the number of nearest neighbors to use when predicting the class of an unseen attribution. In `scikit-learn`, this parameter is called `n_neighbors`. In the following figure, we show that when this number is too low, a randomly labeled sample can cause an error. In contrast, when it is too high, the actual nearest neighbors have a lower effect on the result:



In the figure (a), on the left-hand side, we would usually expect the test sample (the triangle) to be classified as a circle. However, if `n_neighbors` is 1, the single red diamond in this area (likely a noisy sample) causes the sample to be predicted as being a diamond, while it appears to be in a red area. In the figure (b), on the right-hand side, we would usually expect the test sample to be classified as a diamond. However, if `n_neighbors` is 7, the three nearest neighbors (which are all diamonds) are overridden by the large number of circle samples.

If we want to test a number of values for the `n_neighbors` parameter, for example, each of the values from 1 to 20, we can rerun the experiment many times by setting `n_neighbors` and observing the result:

```
avg_scores = []
all_scores = []
parameter_values = list(range(1, 21))  # Include 20
for n_neighbors in parameter_values:
    estimator = KNeighborsClassifier(n_neighbors=n_neighbors)
    scores = cross_val_score(estimator, X, y, scoring='accuracy')
```

Compute and store the average in our list of scores. We also store the full set of scores for later analysis:

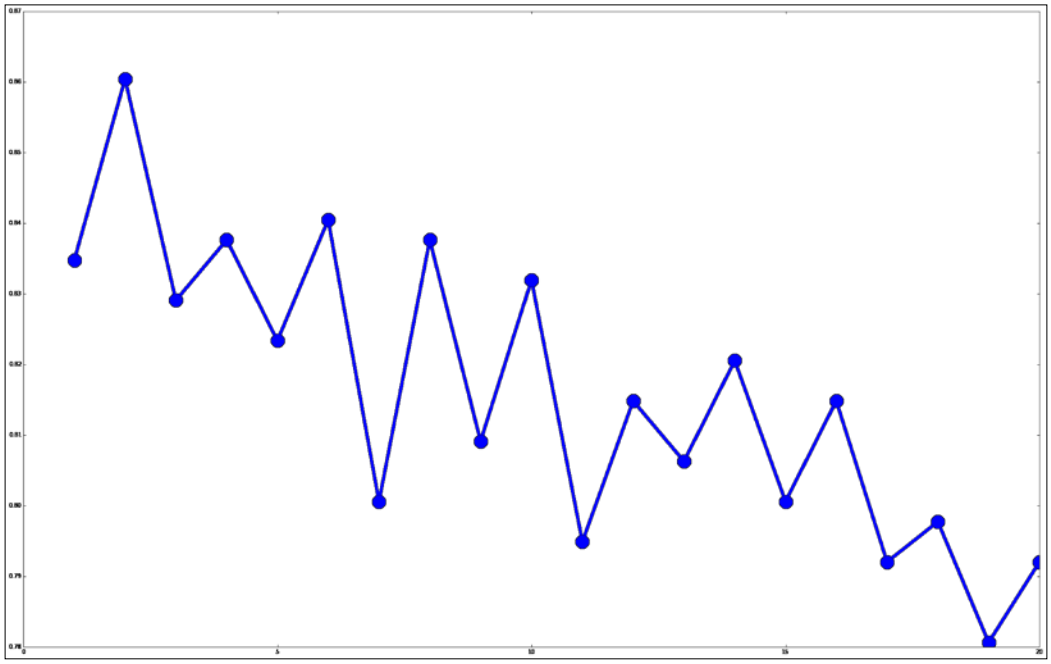
```
avg_scores.append(np.mean(scores))
all_scores.append(scores)
```

We can then plot the relationship between the value of `n_neighbors` and the accuracy. First, we tell the IPython Notebook that we want to show plots inline in the notebook itself:

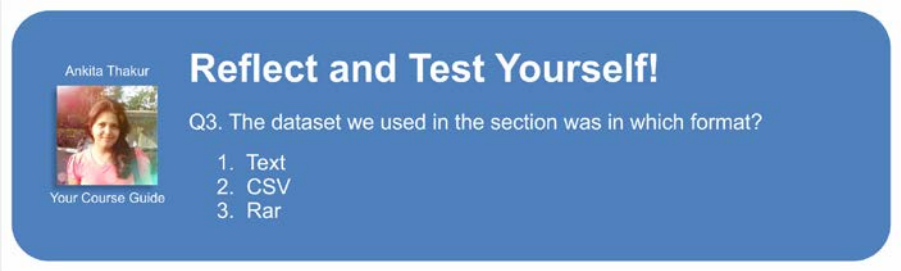
```
%matplotlib inline
```

We then import `pyplot` from the `matplotlib` library and plot the parameter values alongside average scores:

```
from matplotlib import pyplot as plt plt.plot(parameter_values,
avg_scores, '-o')
```



While there is a lot of variance, the plot shows a decreasing trend as the number of neighbors increases.



Preprocessing using pipelines

When taking measurements of real-world objects, we can often get features in very different ranges. For instance, if we are measuring the qualities of an animal, we might have several features, as follows:

- **Number of legs**: This is between the range of 0-8 for most animals, while some have many more!
- **Weight**: This is between the range of only a few micrograms, all the way to a blue whale with a weight of 190,000 kilograms!
- **Number of hearts**: This can be between zero to five, in the case of the earthworm.

For a mathematical-based algorithm to compare each of these features, the differences in the scale, range, and units can be difficult to interpret. If we used the above features in many algorithms, the weight would probably be the most influential feature due to only the larger numbers and not anything to do with the actual effectiveness of the feature.

One of the methods to overcome this is to use a process called preprocessing to *normalize* the features so that they all have the same range, or are put into categories like *small*, *medium* and *large*. Suddenly, the large difference in the types of features has less of an impact on the algorithm, and can lead to large increases in the accuracy.

Preprocessing can also be used to choose only the more effective features, create new features, and so on. Preprocessing in `scikit-learn` is done through `Transformer` objects, which take a dataset in one form and return an altered dataset after some transformation of the data. These don't have to be numerical, as Transformers are also used to extract features-however, in this section, we will stick with preprocessing.

An example

We can show an example of the problem by *breaking* the `Ionosphere` dataset. While this is only an example, many real-world datasets have problems of this form. First, we create a copy of the array so that we do not alter the original dataset:

```
X_broken = np.array(X)
```

Next, we *break* the dataset by dividing every second feature by `10`:

```
X_broken[:,::2] /= 10
```

In theory, this should not have a great effect on the result. After all, the values for these features are still relatively the same. The major issue is that the scale has changed and the odd features are now *larger* than the even features. We can see the effect of this by computing the accuracy:

```
estimator = KNeighborsClassifier()
original_scores = cross_val_score(estimator, X, y,
  scoring='accuracy')
print("The original average accuracy for is
{0:.1f}%".format(np.mean(original_scores) * 100))
broken_scores = cross_val_score(estimator, X_broken, y,
  scoring='accuracy')
print("The 'broken' average accuracy for is
  {0:.1f}%".format(np.mean(broken_scores) * 100))
```

This gives a score of 82.3 percent for the original dataset, which drops down to 71.5 percent on the broken dataset. We can fix this by scaling all the features to the range `0` to `1`.

Standard preprocessing

The preprocessing we will perform for this experiment is called feature-based normalization through the `MinMaxScaler` class. Continuing with the IPython notebook from the rest of this chapter, first, we import this class:

```
from sklearn.preprocessing import MinMaxScaler
```

This class takes each feature and scales it to the range `0` to `1`. The minimum value is replaced with `0`, the maximum with `1`, and the other values somewhere in between.

To apply our preprocessor, we run the transform function on it. While `MinMaxScaler` doesn't, some transformers need to be trained first in the same way that the classifiers do. We can combine these steps by running the `fit_transform` function instead:

```
X_transformed = MinMaxScaler().fit_transform(X)
```

Here, `X_transformed` will have the same shape as *X*. However, each column will have a maximum of 1 and a minimum of 0.

There are various other forms of normalizing in this way, which is effective for other applications and feature types:

- Ensure the sum of the values for each sample equals to 1, using `sklearn.preprocessing.Normalizer`
- Force each feature to have a zero mean and a variance of 1, using `sklearn.preprocessing.StandardScaler`, which is a commonly used starting point for normalization
- Turn numerical features into binary features, where any value above a threshold is 1 and any below is 0, using `sklearn.preprocessing.Binarizer`

We will use combinations of these preprocessors in later chapters, along with other types of `Transformers` object.

Putting it all together

We can now create a workflow by combining the code from the previous sections, using the broken dataset previously calculated:

```
X_transformed = MinMaxScaler().fit_transform(X_broken)
estimator = KNeighborsClassifier()
transformed_scores = cross_val_score(estimator, X_transformed, y,
  scoring='accuracy')
print("The average accuracy for is
  {0:.1f}%".format(np.mean(transformed_scores) * 100))
```

This gives us back our score of 82.3 percent accuracy. The `MinMaxScaler` resulted in features of the same scale, meaning that no features overpowered others by simply being bigger values. While the Nearest Neighbor algorithm can be confused with larger features, some algorithms handle scale differences better. In contrast, some are much worse!

Pipelines

As experiments grow, so does the complexity of the operations. We may split up our dataset, binarize features, perform feature-based scaling, perform sample-based scaling, and many more operations.

Keeping track of all of these operations can get quite confusing and can result in being unable to replicate the result. Problems include forgetting a step, incorrectly applying a transformation, or adding a transformation that wasn't needed.

Another issue is the order of the code. In the previous section, we created our `X_transformed` dataset and then created a new estimator for the cross validation. If we had multiple steps, we would need to track all of these changes to the dataset in the code.

Pipelines are a construct that addresses these problems (and others, which we will see in the next chapter). Pipelines store the steps in your data mining workflow. They can take your raw data in, perform all the necessary transformations, and then create a prediction. This allows us to use pipelines in functions such as `cross_val_score`, where they expect an estimator. First, import the `Pipeline` object:

```
from sklearn.pipeline import Pipeline
```

Pipelines take a list of steps as input, representing the chain of the data mining application. The last step needs to be an `Estimator`, while all previous steps are `Transformers`. The input dataset is altered by each `Transformer`, with the output of one step being the input of the next step. Finally, the samples are classified by the last step's estimator. In our pipeline, we have two steps:

1. Use `MinMaxScaler` to scale the feature values from 0 to 1
2. Use `KNeighborsClassifier` as the classification algorithms

Each step is then represented by a tuple `('name', step)`. We can then create our pipeline:

```
scaling_pipeline = Pipeline([('scale', MinMaxScaler()),
                             ('predict', KNeighborsClassifier())])
```

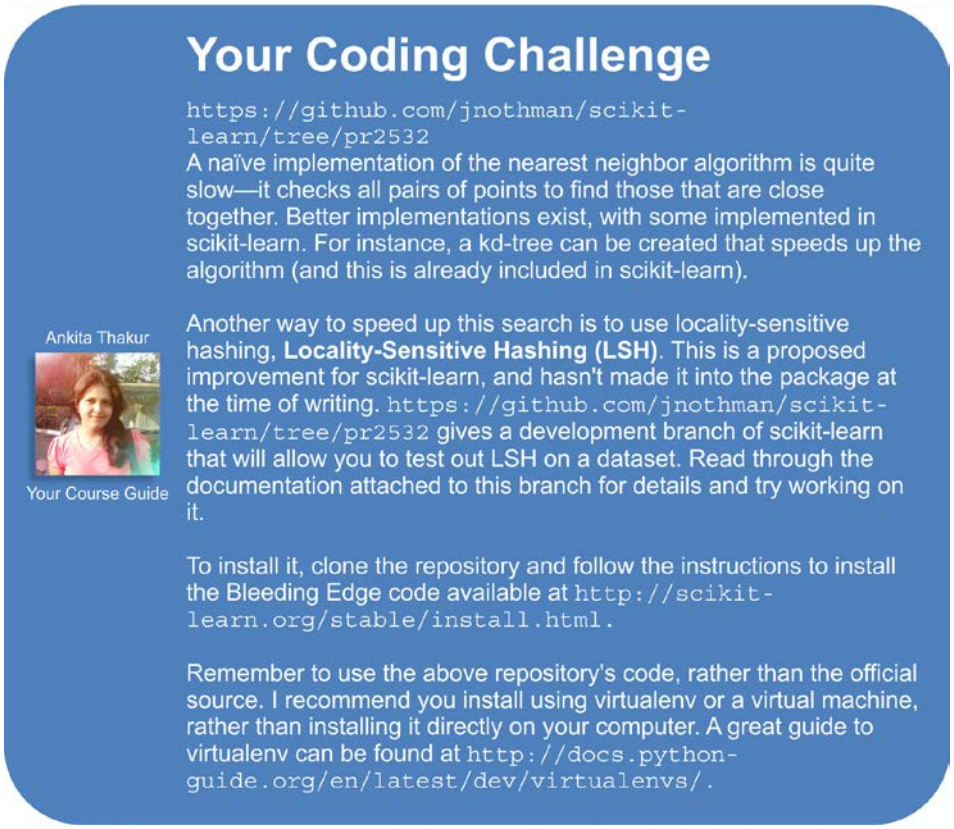
The key here is the list of tuples. The first tuple is our scaling step and the second tuple is the predicting step. We give each step a name: the first we call `scale` and the second we call `predict`, but you can choose your own names. The second part of the tuple is the actual Transformer or estimator object.

Running this pipeline is now very easy, using the cross validation code from before:

```
scores = cross_val_score(scaling_pipeline, X_broken, y,
scoring='accuracy')
print("The pipeline scored an average accuracy for is {0:.1f}%".
format(np.mean(transformed_scores) * 100))
```

This gives us the same score as before (82.3 percent), which is expected, as we are effectively running the same steps.

In later chapters, we will use more advanced testing methods, and setting up pipelines is a great way to ensure that the code complexity does not grow unmanageably.



Summary of Module 3 Chapter 2

Ankita Thakur

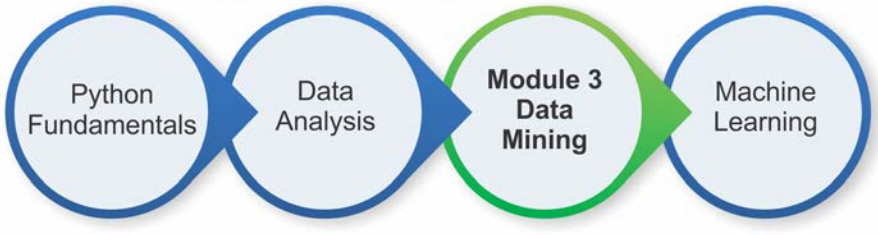
Your Course Guide

In this chapter, we used several of scikit-learn's methods for building a standard workflow to run and evaluate data mining models. We introduced the Nearest Neighbors algorithm, which is already implemented in scikit-learn as an estimator. Using this class is quite easy; first, we call the fit function on our training data, and second, we use the predict function to predict the class of testing samples.

We then looked at preprocessing by fixing poor feature scaling. This was done using a Transformer object and the MinMaxScaler class. These functions also have a fit method and then a transform, which takes a dataset as an input and returns a transformed dataset as an output.

In the next chapter, we will use these concepts in a larger example, predicting the outcome of sports matches using real-world data.

Your Progress through the Course So Far



3
Predicting Sports Winners with Decision Trees

In this chapter, we will look at predicting the winner of sports matches using a different type of classification algorithm: decision trees. These algorithms have a number of advantages over other algorithms. One of the main advantages is that they are readable by humans. In this way, decision trees can be used to learn a procedure, which could then be given to a human to perform if needed. Another advantage is that they work with a variety of features, which we will see in this chapter.

We will cover the following topics in this chapter:

- Using the pandas library for loading and manipulating data
- Decision trees
- Random forests
- Using real-world datasets in data mining
- Creating new features and testing them in a robust framework

Loading the dataset

In this chapter, we will look at predicting the winner of games of the **National Basketball Association** (**NBA**). Matches in the NBA are often close and can be decided in the last minute, making predicting the winner quite difficult. Many sports share this characteristic, whereby the *expected winner* could be beaten by another team on the right day.

Various research into predicting the winner suggests that there may be an upper limit to sports outcome prediction accuracy which, depending on the sport, is between 70 percent and 80 percent accuracy. There is a significant amount of research being performed into sports prediction, often through data mining or statistics-based methods.

Collecting the data

The data we will be using is the match history data for the NBA for the 2013-2014 season. The website `http://Basketball-Reference.com` contains a significant number of resources and statistics collected from the NBA and other leagues. To download the dataset, perform the following steps:

1. Navigate to `http://www.basketball-reference.com/leagues/NBA_2014_games.html` in your web browser.
2. Click on the **Export** button next to the **Regular Season** heading.
3. Download the file to your data folder and make a note of the path.

This will download a **CSV** (short for **Comma Separated Values**) file containing the results of the 1,230 games in the regular season for the NBA.

CSV files are simply text files where each line contains a new row and each value is separated by a comma (hence the name). CSV files can be created manually by simply typing into a text editor and saving with a `.csv` extension. They can also be opened in any program that can read text files, but can also be opened in Excel as a spreadsheet.

We will load the file with the **pandas** (short for **Python Data Analysis**) library, which is an incredibly useful library for manipulating data. Python also contains a built-in library called `csv` that supports reading and writing CSV files. However, we will use pandas, which provides more powerful functions that we will use later in the chapter for creating new features.

For this chapter, you will need to install pandas. The easiest way to install it is to use `pip3`, as you did in *Chapter 1, Getting Started with Data Mining* to install `scikit-learn`:

```
$pip3 install pandas
```

If you have difficulty in installing pandas, head to their website at `http://pandas.pydata.org/getpandas.html` and read the installation instructions for your system.

Using pandas to load the dataset

The pandas library is a library for loading, managing, and manipulating data. It handles data structures behind-the-scenes and supports analysis methods, such as computing the mean.

When doing multiple data mining experiments, you will find that you write many of the same functions again and again, such as reading files and extracting features. Each time this reimplementation happens, you run the risk of introducing bugs. Using a high-class library such as pandas significantly reduces the amount of work needed to do these functions and also gives you more confidence in using well tested code.

Throughout this module, we will be using pandas quite significantly, introducing use cases as we go.

We can load the dataset using the `read_csv` function:

```
import pandas as pd
dataset = pd.read_csv(data_filename)
```

The result of this is a pandas **Dataframe**, and it has some useful functions that we will use later on. Looking at the resulting dataset, we can see some issues. Type the following and run the code to see the first five rows of the dataset:

```
dataset.ix[:5]
```

Here's the output:

Out[47]:

Date	NaN	Visitor/Neutral	PTS	Home/Neutral	PTS	NaN	Notes
Tue Oct 29 2013	Box Score	Orlando Magic	87	Indiana Pacers	97	NaN	NaN
		Los Angeles Clippers	103	Los Angeles Lakers	116	NaN	NaN
		Chicago Bulls	95	Miami Heat	107	NaN	NaN
Wed Oct 30 2013	Box Score	Brooklyn Nets	94	Cleveland Cavaliers	98	NaN	NaN

This is actually a usable dataset, but it contains some problems that we will fix up soon.

Cleaning up the dataset

After looking at the output, we can see a number of problems:

- The date is just a string and not a date object
- The first row is blank
- From visually inspecting the results, the headings aren't complete or correct

These issues come from the data, and we could fix this by altering the data itself. However, in doing this, we could forget the steps we took or misapply them; that is, we can't replicate our results. As with the previous section where we used pipelines to track the transformations we made to a dataset, we will use pandas to apply transformations to the raw data itself.

The `pandas.read_csv` function has parameters to fix each of these issues, which we can specify when loading the file. We can also change the headings after loading the file, as shown in the following code:

```
dataset = pd.read_csv(data_filename, parse_dates=["Date"],
  skiprows=[0,])
dataset.columns = ["Date", "Score Type", "Visitor Team",
  "VisitorPts", "Home Team", "HomePts", "OT?", "Notes"]
```

The results have significantly improved, as we can see if we print out the resulting data frame:

```
dataset.ix[:5]
```

The output is as follows:

Out[48]:

	Date	Score Type	Visitor Team	VisitorPts	Home Team	HomePts	OT?	Notes
0	2013-10-29	Box Score	Orlando Magic	87	Indiana Pacers	97	NaN	NaN
1	2013-10-29	Box Score	Los Angeles Clippers	103	Los Angeles Lakers	116	NaN	NaN
2	2013-10-29	Box Score	Chicago Bulls	95	Miami Heat	107	NaN	NaN
3	2013-10-30	Box Score	Brooklyn Nets	94	Cleveland Cavaliers	98	NaN	NaN
4	2013-10-30	Box Score	Atlanta Hawks	109	Dallas Mavericks	118	NaN	NaN
5	2013-10-30	Box Score	Washington Wizards	102	Detroit Pistons	113	NaN	NaN

Even in well-compiled data sources such as this one, you need to make some adjustments. Different systems have different nuances, resulting in data files that are not quite compatible with each other.

Now that we have our dataset, we can compute a baseline. A baseline is an accuracy that indicates an easy way to get a good accuracy. Any data mining solution should beat this.

In each match, we have two teams: a home team and a visitor team. An obvious baseline, called the chance rate, is 50 percent. Choosing randomly will (over time) result in an accuracy of 50 percent.

Extracting new features

We can now extract our features from this dataset by combining and comparing the existing data. First up, we need to specify our class value, which will give our classification algorithm something to compare against to see if its prediction is correct or not. This could be encoded in a number of ways; however, for this application, we will specify our class as 1 if the home team wins and 0 if the visitor team wins. In basketball, the team with the most points wins. So, while the data set doesn't specify who wins, we can compute it easily.

We can specify the data set by the following:

```
dataset["HomeWin"] = dataset["VisitorPts"] < dataset["HomePts"]
```

We then copy those values into a `NumPy` array to use later for our `scikit-learn` classifiers. There is not currently a clean integration between pandas and scikit-learn, but they work nicely together through the use of `NumPy` arrays. While we will use pandas to extract features, we will need to extract the values to use them with scikit-learn:

```
y_true = dataset["HomeWin"].values
```

The preceding array now holds our class values in a format that scikit-learn can read.

We can also start creating some features to use in our data mining. While sometimes we just throw the raw data into our classifier, we often need to derive continuous numerical or categorical features.

The first two features we want to create to help us predict which team will win are whether either of those two teams won their last game. This would roughly approximate which team is playing well.

We will compute this feature by iterating through the rows in order and recording which team won. When we get to a new row, we look up whether the team won the last time we saw them.

We first create a (default) dictionary to store the team's last result:

```
from collections import defaultdict
won_last = defaultdict(int)
```

The key of this dictionary will be the team and the value will be whether they won their previous game. We can then iterate over all the rows and update the current row with the team's last result:

```
for index, row in dataset.iterrows():
    home_team = row["Home Team"]
    visitor_team = row["Visitor Team"]
    row["HomeLastWin"] = won_last[home_team]
    row["VisitorLastWin"] = won_last[visitor_team]
    dataset.ix[index] = row
```

Note that the preceding code relies on our dataset being in chronological order. Our dataset is in order; however, if you are using a dataset that is not in order, you will need to replace `dataset.iterrows()` with `dataset.sort("Date").iterrows()`.

We then set our dictionary with the each team's result (from this row) for the next time we see these teams. The code is as follows:

```
    won_last[home_team] = row["HomeWin"]
    won_last[visitor_team] = not row["HomeWin"]
```

After the preceding code runs, we will have two new features: `HomeLastWin` and `VisitorLastWin`. We can have a look at the dataset. There isn't much point in looking at the first five games though. Due to the way our code runs, we didn't have data for them at that point. Therefore, until a team's second game of the season, we won't know their current form. We can instead look at different places in the list. The following code will show the 20th to the 25th games of the season:

```
dataset.ix[20:25]
```

Here's the output:

Out[52]:

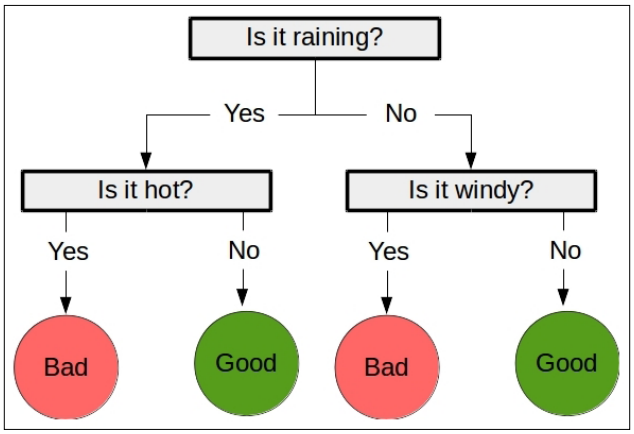
	Date	Score Type	Visitor Team	VisitorPts	Home Team	HomePts	OT?	Notes	HomeWin	HomeLastWin	VisitorLastWin
20	2013-11-01	Box Score	Milwaukee Bucks	105	Boston Celtics	98	NaN	NaN	False	False	False
21	2013-11-01	Box Score	Miami Heat	100	Brooklyn Nets	101	NaN	NaN	True	False	False
22	2013-11-01	Box Score	Cleveland Cavaliers	84	Charlotte Bobcats	90	NaN	NaN	True	False	True
23	2013-11-01	Box Score	Portland Trail Blazers	113	Denver Nuggets	98	NaN	NaN	False	False	False
24	2013-11-01	Box Score	Dallas Mavericks	105	Houston Rockets	113	NaN	NaN	True	True	True
25	2013-11-01	Box Score	San Antonio Spurs	91	Los Angeles Lakers	85	NaN	NaN	False	False	True

You can change those indices to look at other parts of the data, as there are over 1000 games in our dataset!

Currently, this gives a false value to all teams (including the previous year's champion!) when they are first seen. We could improve this feature using the previous year's data, but will not do that in this chapter.

Decision trees

Decision trees are a class of supervised learning algorithm like a flow chart that consists of a sequence of nodes, where the values for a sample are used to make a decision on the next node to go to.



As with most classification algorithms, there are two components:

- The first is the training stage, where a tree is built using training data. While the nearest neighbor algorithm from the previous chapter did not have a training phase, it is needed for decision trees. In this way, the nearest neighbor algorithm is a lazy learner, only doing any work when it needs to make a prediction. In contrast, decision trees, like most classification methods, are eager learners, undertaking work at the training stage.
- The second is the predicting stage, where the trained tree is used to predict the classification of new samples. Using the previous example tree, a data point of `["is raining", "very windy"]` would be classed as `"bad weather"`.

There are many algorithms for creating decision trees. Many of these algorithms are iterative. They start at the base node and decide the best feature to use for the first decision, then go to each node and choose the next best feature, and so on. This process is stopped at a certain point, when it is decided that nothing more can be gained from extending the tree further.

The `scikit-learn` package implements the **CART** (**Classification and Regression Trees**) algorithm as its default decision tree class, which can use both categorical and continuous features.

Parameters in decision trees

One of the most important features for a decision tree is the stopping criterion. As a tree is built, the final few decisions can often be somewhat arbitrary and rely on only a small number of samples to make their decision. Using such specific nodes can results in trees that significantly overfit the training data. Instead, a stopping criterion can be used to ensure that the decision tree does not reach this exactness.

Instead of using a stopping criterion, the tree could be created in full and then trimmed. This trimming process removes nodes that do not provide much information to the overall process. This is known as pruning.

The decision tree implementation in scikit-learn provides a method to stop the building of a tree using the following options:

- `min_samples_split`: This specifies how many samples are needed in order to create a new node in the decision tree
- `min_samples_leaf`: This specifies how many samples must be resulting from a node for it to stay

The first dictates whether a decision node will be created, while the second dictates whether a decision node will be kept.

Another parameter for decision tress is the criterion for creating a decision. Gini impurity and Information gain are two popular ones:

- **Gini impurity**: This is a measure of how often a decision node would incorrectly predict a sample's class
- **Information gain**: This uses information-theory-based entropy to indicate how much extra information is gained by the decision node

Using decision trees

We can import the `DecisionTreeClassifier` class and create a decision tree using scikit-learn:

```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(random_state=14)
```

We used 14 for our `random_state` again and will do so for most of the module. Using the same random seed allows for replication of experiments. However, with your experiments, you should mix up the random state to ensure that the algorithm's performance is not tied to the specific value.

We now need to extract the dataset from our pandas data frame in order to use it with our `scikit-learn` classifier. We do this by specifying the columns we wish to use and using the `values` parameter of a view of the data frame. The following code creates a dataset using our last win values for both the home team and the visitor team:

```
X_previouswins = dataset[["HomeLastWin", "VisitorLastWin"]].values
```

Decision trees are estimators, as introduced in *Chapter 2*, *Classifying with scikit-learn Estimators*, and therefore have `fit` and `predict` methods. We can also use the `cross_val_score` method to get the average score (as we did previously):

```
scores = cross_val_score(clf, X_previouswins, y_true,
scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

This scores 56.1 percent: we are better than choosing randomly! We should be able to do better. Feature engineering is one of the most difficult tasks in data mining, and choosing good features is key to getting good outcomes—more so than choosing the right algorithm!

Sports outcome prediction

We may be able to do better by trying other features. We have a method for testing how accurate our models are. The `cross_val_score` method allows us to try new features.

There are many possible features we could use, but we will try the following questions:

- Which team is considered better generally?
- Which team won their last encounter?

We will also try putting the raw teams into the algorithm to check whether the algorithm can learn a model that checks how different teams play against each other.

Putting it all together

For the first feature, we will create a feature that tells us if the home team is generally *better* than the visitors. To do this, we will load the standings (also called a ladder in some sports) from the NBA in the previous season. A team will be considered better if it ranked higher in 2013 than the other team.

To obtain the standings data, perform the following steps:

1. Navigate to `http://www.basketball-reference.com/leagues/NBA_2013_standings.html` in your web browser.
2. Select **Expanded Standings** to get a single list for the entire league.
3. Click on the **Export** link.
4. Save the downloaded file in your data folder.

Back in your IPython Notebook, enter the following lines into a new cell. You'll need to ensure that the file was saved into the location pointed to by the `data_folder` variable. The code is as follows:

```
standings_filename = os.path.join(data_folder,
"leagues_NBA_2013_standings_expanded-standings.csv")
standings = pd.read_csv(standings_filename, skiprows=[0,1])
```

You can view the ladder by just typing `standings` into a new cell and running the code:

```
Standings
```

The output is as follows:

Out[57]:

	Rk	Team	Overall	Home	Road	E	W	A	C	SE	...	Post	≤3	≥10	Oct	Nov	Dec	Jan	Feb	Mar	Apr
0	1	Miami Heat	66-16	37-4	29-12	41-11	25-5	14-4	12-6	15-1	...	30-2	9-3	39-8	1-0	10-3	10-5	8-5	12-1	17-1	8-1
1	2	Oklahoma City Thunder	60-22	34-7	26-15	21-9	39-13	7-3	8-2	6-4	...	21-8	3-6	44-6	NaN	13-4	11-2	11-5	7-4	12-5	6-2
2	3	San Antonio Spurs	58-24	35-6	23-18	25-5	33-19	8-2	9-1	8-2	...	16-12	9-5	31-10	1-0	12-4	12-4	12-3	8-3	10-4	3-6
3	4	Denver Nuggets	57-25	38-3	19-22	19-11	38-14	5-5	10-0	4-6	...	24-4	11-7	28-8	0-1	8-8	9-6	12-3	8-4	13-2	7-1
4	5	Los Angeles Clippers	56-26	32-9	24-17	21-9	35-17	7-3	8-2	6-4	...	17-9	3-5	38-12	1-0	8-6	16-0	9-7	8-5	7-7	7-1
5	6	Memphis Grizzlies	56-26	32-9	24-17	22-8	34-18	8-2	8-2	6-4	...	23-8	6-4	28-9	0-1	12-1	7-7	10-7	9-2	11-6	7-2
6	7	New York Knicks	54-28	31-10	23-18	37-15	17-13	10-6	12-6	15-3	...	22-10	7-5	31-12	NaN	11-4	10-5	7-6	6-5	12-6	8-2
7	8	Brooklyn Nets	49-33	26-15	23-18	36-16	13-17	11-5	13-5	12-6	...	18-11	9-4	23-17	NaN	11-4	5-11	11-4	7-5	8-7	7-2
8	9	Indiana Pacers	49-32	30-11	19-21	31-20	18-12	6-11	13-3	12-6	...	17-11	4-9	27-14	1-0	7-8	10-5	9-6	9-3	11-5	2-5
9	10	Golden State Warriors	47-35	28-13	19-22	19-11	28-24	7-3	5-5	7-3	...	17-13	5-3	20-18	1-0	8-6	12-4	8-7	4-8	9-7	5-3

Next, we create a new feature using a similar pattern to the previous feature. We iterate over the rows, looking up the standings for the home team and visitor team. The code is as follows:

```
dataset["HomeTeamRanksHigher"] = 0
for index, row in dataset.iterrows():
    home_team = row["Home Team"]
    visitor_team = row["Visitor Team"]
```

As an important adjustment to the data, a team was renamed between the 2013 and 2014 seasons (but it was still the same team). This is an example of one of the many different things that can happen when trying to integrate data! We will need to adjust the team lookup, ensuring we get the correct team's ranking:

```
    if home_team == "New Orleans Pelicans":
        home_team = "New Orleans Hornets"
    elif visitor_team == "New Orleans Pelicans":
        visitor_team = "New Orleans Hornets"
```

Now we can get the rankings for each team. We then compare them and update the feature in the row:

```
home_rank = standings[standings["Team"] ==
      home_team]["Rk"].values[0]
      visitor_rank = standings[standings["Team"] ==
      visitor_team]["Rk"].values[0]
      row["HomeTeamRanksHigher"] = int(home_rank > visitor_rank)
      dataset.ix[index] = row
```

Next, we use the `cross_val_score` function to test the result. First, we extract the dataset:

```
X_homehigher =  dataset[["HomeLastWin", "VisitorLastWin",
  "HomeTeamRanksHigher"]].values
```

Then, we create a new `DecisionTreeClassifier` and run the evaluation:

```
clf = DecisionTreeClassifier(random_state=14)
scores = cross_val_score(clf, X_homehigher, y_true,
  scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

This now scores 60.3 percent—even better than our previous result. Can we do better?

Next, let's test which of the two teams won their last match. While rankings can give some hints on who won (the higher ranked team is more likely to win), sometimes teams play better against other teams. There are many reasons for this – for example, some teams may have strategies that work against other teams really well. Following our previous pattern, we create a dictionary to store the winner of the past game and create a new feature in our data frame. The code is as follows:

```
last_match_winner = defaultdict(int)
dataset["HomeTeamWonLast"] = 0
```

Then, we iterate over each row and get the home team and visitor team:

```
for index, row in dataset.iterrows():
    home_team = row["Home Team"]
    visitor_team = row["Visitor Team"]
```

We want to see who won the last game between these two teams regardless of which team was playing *at home*. Therefore, we sort the team names alphabetically, giving us a consistent key for those two teams:

```
    teams = tuple(sorted([home_team, visitor_team]))
```

We look up in our dictionary to see who won the last encounter between the two teams. Then, we update the row in the dataset data frame:

```
    row["HomeTeamWonLast"] = 1 if last_match_winner[teams] ==
      row["Home Team"] else 0
      dataset.ix[index] = row
```

Finally, we update our dictionary with the winner of this game in order to compute the feature for the next time these two teams meet:

```
winner = row["Home Team"] if row["HomeWin"] else row
["Visitor Team"]
last_match_winner[teams] = winner
```

Next, we will create a dataset with just our two features. You could try different combinations of features to see if they obtain different results. The code is as follows:

```
X_lastwinner =  dataset[["HomeTeamRanksHigher", "HomeTeam
  WonLast"]].values
clf = DecisionTreeClassifier(random_state=14)
scores = cross_val_score(clf, X_lastwinner, y_true,
  scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

This scores 60.6 percent . Our results are getting better and better.

Finally, we will check what happens if we throw a lot of data at the decision tree, and see if it can learn an effective model anyway. We will enter the teams into the tree and check whether a decision tree can learn to incorporate that information.

While decision trees are capable of learning from categorical features, the implementation in scikit-learn requires those features to be encoded first. We can use the `LabelEncoder` transformer to convert between the string-based team names into integers. The code is as follows:

```
from sklearn.preprocessing import LabelEncoder
encoding = LabelEncoder()
```

We will fit this transformer to the home teams so that it learns an integer representation for each team:

```
encoding.fit(dataset["Home Team"].values)
```

We extract all of the labels for the home teams and visitor teams, and then join them (called *stacking* in `NumPy`) to create a matrix encoding both the home team and the visitor team for each game. The code is as follows:

```
home_teams = encoding.transform(dataset["Home Team"].values)
visitor_teams = encoding.transform(dataset["Visitor Team"].values)
X_teams = np.vstack([home_teams, visitor_teams]).T
```

These integers can be fed into the decision tree, but they will still be interpreted as continuous features by `DecisionTreeClassifier`. For example, teams may be allocated integers 0 to 16. The algorithm will see teams 1 and 2 as being similar, while teams 4 and 10 will be different—but this makes no sense as all. All of the teams are different from each other—two teams are either the same or they are not!

To fix this inconsistency, we use the `OneHotEncoder` transformer to encode these integers into a number of binary features. Each binary feature will be a single value for the feature. For example, if the NBA team Chicago Bulls is allocated as integer 7 by the `LabelEncoder`, then the seventh feature returned by the `OneHotEncoder` will be a 1 if the team is *Chicago Bulls* and 0 for all other teams. This is done for every possible value, resulting in a much larger dataset. The code is as follows:

```
from sklearn.preprocessing import OneHotEncoder
onehot = OneHotEncoder()
```

We fit and transform on the same dataset, saving the results:

```
X_teams_expanded = onehot.fit_transform(X_teams).todense()
```

Next, we run the decision tree as before on the new dataset:

```
clf = DecisionTreeClassifier(random_state=14)
scores = cross_val_score(clf, X_teams_expanded, y_true,
  scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

This scores an accuracy of 60 percent. The score is better than the baseline, but not as good as before. It is possible that the larger number of features were not handled properly by the decision trees. For this reason, we will try changing the algorithm and see if that helps. Data mining can be an iterative process of trying new algorithms and features.

Random forests

A single decision tree can learn quite complex functions. However, in many ways it will be prone to overfitting—learning rules that work only for the training set. One of the ways that we can adjust for this is to limit the number of rules that it learns. For instance, we could limit the depth of the tree to just three layers. Such a tree will learn the best rules for splitting the dataset at a global level, but won't learn highly specific rules that separate the dataset into highly accurate groups. This trade-off results in trees that may have a good generalization, but overall slightly poorer performance.

To compensate for this, we could create many decision trees and then ask each to predict the class value. We could take a majority vote and use that answer as our overall prediction. Random forests work on this principle.

There are two problems with the aforementioned procedure. The first problem is that building decision trees is largely deterministic—using the same input will result in the same output each time. We only have one training dataset, which means our input (and therefore the output) will be the same if we try build multiple trees. We can address this by choosing a random subsample of our dataset, effectively creating `new` training sets. This process is called **bagging**.

The second problem is that the features that are used for the first few decision nodes in our tree will be quite good. Even if we choose random subsamples of our training data, it is still quite possible that the decision trees built will be largely the same. To compensate for this, we also choose a random subset of the features to perform our data splits on.

Then, we have randomly built trees using randomly chosen samples, using (nearly) randomly chosen features. This is a Random Forest and, perhaps *unintuitively*, this algorithm is very effective on many datasets.

How do ensembles work?

The randomness inherent in Random forests may make it seem like we are leaving the results of the algorithm up to chance. However, we apply the benefits of averaging to nearly randomly built decision trees, resulting in an algorithm that reduces the variance of the result.

Variance is the error introduced by variations in the training dataset on the algorithm. Algorithms with a high variance (such as decision trees) can be greatly affected by variations to the training dataset. This results in models that have the problem of overfitting.

In contrast, **bias** is the error introduced by assumptions in the algorithm rather than anything to do with the dataset, that is, if we had an algorithm that presumed that all features would be normally distributed then our algorithm may have a high error if the features were not. Negative impacts from bias can be reduced by analyzing the data to see if the classifier's data model matches that of the actual data.

By averaging a large number of decision trees, this variance is greatly reduced. This results in a model with a higher overall accuracy.

In general, ensembles work on the assumption that errors in prediction are effectively random and that those errors are quite different from classifier to classifier. By averaging the results across many models, these random errors are canceled out—leaving the true prediction. We will see many more ensembles in action throughout the rest of the book.

Parameters in Random forests

The Random forest implementation in scikit-learn is called `RandomForestClassifier`, and it has a number of parameters. As Random forests use many instances of `DecisionTreeClassifier`, they share many of the same parameters such as the criterion (Gini Impurity or Entropy/Information Gain), `max_features`, and `min_samples_split`.

Also, there are some new parameters that are used in the ensemble process:

- `n_estimators`: This dictates how many decision trees should be built. A higher value will take longer to run, but will (probably) result in a higher accuracy.
- `oob_score`: If true, the method is tested using samples that aren't in the random subsamples chosen for training the decision trees.
- `n_jobs`: This specifies the number of cores to use when training the decision trees in parallel.

The `scikit-learn` package uses a library called `Joblib` for in-built parallelization. This parameter dictates how many cores to use. By default, only a single core is used—if you have more cores, you can increase this, or set it to -1 to use all cores.

Applying Random forests

Random forests in scikit-learn use the estimator interface, allowing us to use almost the exact same code as before to do cross fold validation:

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(random_state=14)
scores = cross_val_score(clf, X_teams, y_true, scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

This results in an immediate benefit of 60.6 percent, up by 0.6 points by just swapping the classifier.

Random forests, using subsets of the features, should be able to learn more effectively with more features than normal decision trees. We can test this by throwing more features at the algorithm and seeing how it goes:

```
X_all = np.hstack([X_home_higher, X_teams])
clf = RandomForestClassifier(random_state=14)
scores = cross_val_score(clf, X_all, y_true, scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

This results in 61.1 percent —even better! We can also try some other parameters using the `GridSearchCV` class as we introduced in *Chapter 2*, *Classifying with scikit-learn Estimators*:

```
parameter_space = {
  "max_features": [2, 10, 'auto'],
  "n_estimators": [100,],
  "criterion": ["gini", "entropy"],
  "min_samples_leaf": [2, 4, 6],
}
clf = RandomForestClassifier(random_state=14)
grid = GridSearchCV(clf, parameter_space)
grid.fit(X_all, y_true)
print("Accuracy: {0:.1f}%".format(grid.best_score_ * 100))
```

This has a much better accuracy of 64.2 percent!

If we wanted to see the parameters used, we can print out the best model that was found in the grid search. The code is as follows:

```
print(grid.best_estimator_)
```

The result shows the parameters that were used in the best scoring model:

```
RandomForestClassifier(bootstrap=True, compute_importances=None,
  criterion='entropy', max_depth=None, max_features=2,
    max_leaf_nodes=None, min_density=None, min_samples_leaf=6,
      min_samples_split=2, n_estimators=100, n_jobs=1,
        oob_score=False, random_state=14, verbose=0)
```

Engineering new features

In the previous few examples, we saw that changing the features can have quite a large impact on the performance of the algorithm. Through our small amount of testing, we had more than 10 percent variance just from the features.

You can create features that come from a simple function in pandas by doing something like this:

```
dataset["New Feature"] = feature_creator()
```

The `feature_creator` function must return a list of the feature's value for each sample in the dataset. A common pattern is to use the dataset as a parameter:

```
dataset["New Feature"] = feature_creator(dataset)
```

You can create those features more directly by setting all the values to a single "default" value, like 0 in the next line:

```
dataset["My New Feature"] = 0
```

You can then iterate over the dataset, computing the features as you go. We used this format in this chapter to create many of our features:

```
for index, row in dataset.iterrows():
  home_team = row["Home Team"]
  visitor_team = row["Visitor Team"]
  # Some calculation here to alter row
  dataset.ix[index] = row
```

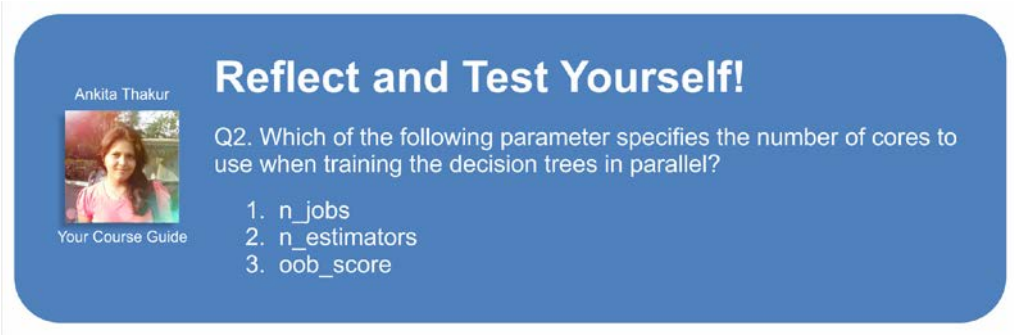
Keep in mind that this pattern isn't very efficient. If you are going to do this, try all of your features at once. A common "best practice" is to touch every sample as little as possible, preferably only once.

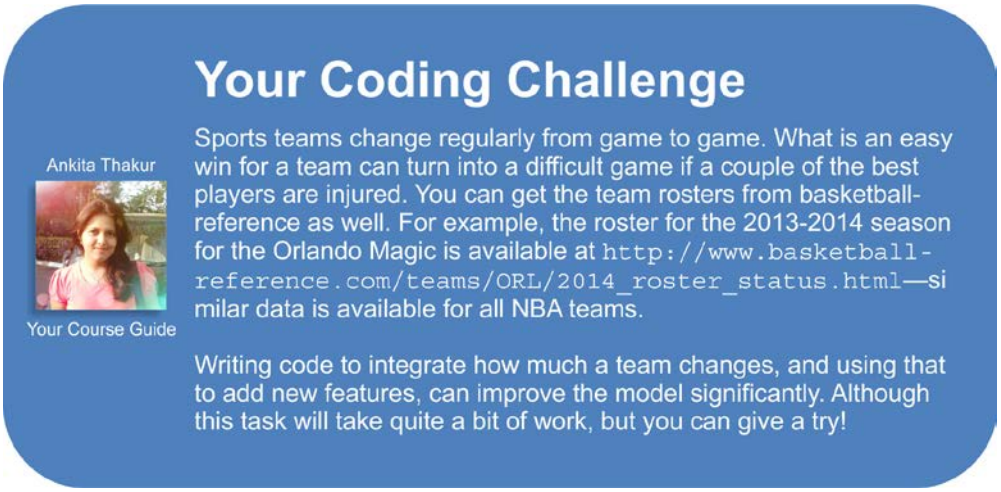
Some example features that you could try and implement are as follows:

- How many days has it been since each team's previous match? Teams may be tired if they play too many games in a short time frame.
- How many games of the last five did each team win? This will give a more stable form of the `HomeLastWin` and `VisitorLastWin` features we extracted earlier (and can be extracted in a very similar way).
- Do teams have a good record when visiting certain other teams? For instance, one team may play well in a particular stadium, even if they are the visitors.

If you are facing trouble extracting features of these types, check the pandas documentation at `http://pandas.pydata.org/pandas-docs/stable/` for help. Alternatively, you can try an online forum such as Stack Overflow for assistance.

More extreme examples could use player data to estimate the strength of each team's sides to predict who won. These types of complex features are used every day by gamblers and sports betting agencies to try to turn a profit by predicting the outcome of sports matches.





Summary of Module 3 Chapter 3

Ankita Thakur

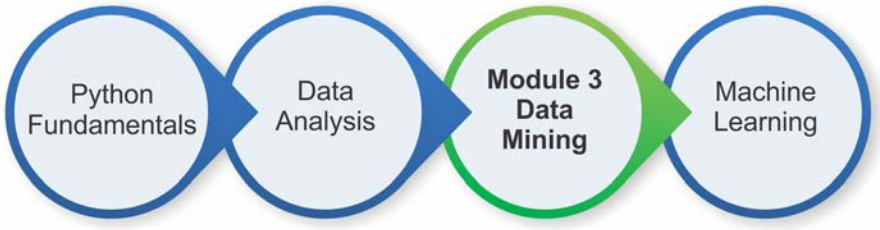
Your Course Guide

In this chapter, we extended our use of scikit-learn's classifiers to perform classification and introduced the pandas library to manage our data. We analyzed real-world data on basketball results from the NBA, saw some of the problems that even well-curated data introduces, and created new features for our analysis.

We saw the effect that good features have on performance and used an ensemble algorithm, Random forests, to further improve the accuracy.

In the next chapter, we will extend the affinity analysis that we performed in the first chapter to create a program to find similar books. We will see how to use algorithms for ranking and also use approximation to improve the scalability of data mining.

Your Progress through the Course So Far



4
Recommending Movies Using Affinity Analysis

In this chapter, we will look at affinity analysis that determines when objects occur frequently together. This is colloquially called market basket analysis, after one of the use cases of determining when items are purchased together frequently.

In *Chapter 3*, *Predicting Sports Winners with Decision Trees*, we looked at an object as a focus and used features to describe that object. In this chapter, the data has a different form. We have transactions where the objects of interest (movies, in this chapter) are used within those transactions in some way. The aim is to discover when objects occur simultaneously. In this example, we wish to work out when two movies are recommended by the same reviewers.

The key concepts of this chapter are as follows:

- Affinity analysis
- Feature association mining using the Apriori algorithm
- Movie recommendations
- Sparse data formats

Affinity analysis

Affinity analysis is the task of determining when objects are used in similar ways. In the previous chapter, we focused on whether the objects themselves are similar. The data for affinity analysis is often described in the form of a transaction. Intuitively, this comes from a transaction at a store—determining when objects are purchased together.

However, it can be applied to many processes:

- Fraud detection
- Customer segmentation
- Software optimization
- Product recommendations

Affinity analysis is usually much more exploratory than classification. We often don't have the complete dataset we expect for many classification tasks. For instance, in movie recommendation, we have reviews from different people on different movies. However, it is unlikely we have each reviewer review all of the movies in our dataset. This leaves an important and difficult question in affinity analysis. If a reviewer hasn't reviewed a movie, is that an indication that they aren't interested in the movie (and therefore wouldn't recommend it) or simply that they haven't reviewed it yet?

We won't answer that question in this chapter, but thinking about gaps in your datasets can lead to questions like this. In turn, that can lead to answers that may help improve the efficacy of your approach.

Algorithms for affinity analysis

We introduced a basic method for affinity analysis in *Chapter 1*, *Getting Started with Data Mining*, which tested all of the possible rule combinations. We computed the confidence and support for each rule, which in turn allowed us to rank them to find the best rules.

However, this approach is not efficient. Our dataset in *Chapter 1*, *Getting Started with Data Mining*, had just five items for sale. We could expect even a small store to have hundreds of items for sale, while many online stores would have thousands (or millions!). With a naive rule creation, such as our previous algorithm, the growth in time needed to compute these rules increases exponentially. As we add more items, the time it takes to compute all rules increases significantly faster. Specifically, the total possible number of rules is $2^n - 1$. For our five-item dataset, there are 31 possible rules. For 10 items, it is 1023. For just 100 items, the number has 30 digits. Even the drastic increase in computing power couldn't possibly keep up with the increases in the number of items stored online. Therefore, we need algorithms that work smarter, as opposed to computers that work harder.

The classic algorithm for affinity analysis is called the Apriori algorithm. It addresses the exponential problem of creating sets of items that occur frequently within a database, called **frequent itemsets**. Once these frequent itemsets are discovered, creating association rules is straightforward.

The intuition behind Apriori is both simple and clever. First, we ensure that a rule has sufficient *support* within the dataset. Defining a minimum support level is the key parameter for Apriori. To build a frequent itemset, for an itemset (A, B) to have a support of at least 30, both A and B must occur at least 30 times in the database. This property extends to larger sets as well. For an itemset (A, B, C, D) to be considered frequent, the set (A, B, C) must also be frequent (as must D).

These *frequent itemsets* can be built up and possible itemsets that are not frequent (of which there are many) will never be tested. This saves significant time in testing new rules.

Other example algorithms for affinity analysis include the **Eclat** and **FP-growth** algorithms. There are many improvements to these algorithms in the data mining literature that further improve the efficiency of the method. In this chapter, we will focus on the basic Apriori algorithm.

Choosing parameters

To perform association rule mining for affinity analysis, we first use the Apriori to generate frequent itemsets. Next, we create association rules (for example, if a person recommended movie *X*, they would also recommend movie *Y*) by testing combinations of premises and conclusions within those frequent itemsets.

For the first stage, the Apriori algorithm needs a value for the minimum support that an itemset needs to be considered frequent. Any itemsets with less support will not be considered. Setting this minimum support too low will cause Apriori to test a larger number of itemsets, slowing the algorithm down. Setting it too high will result in fewer itemsets being considered frequent.

In the second stage, after the frequent itemsets have been discovered, association rules are tested based on their confidence. We could choose a minimum confidence level, a number of rules to return, or simply return all of them and let the user decide what to do with them.

In this chapter, we will return only rules above a given confidence level. Therefore, we need to set our minimum confidence level. Setting this too low will result in rules that have a high support, but are not very accurate. Setting this higher will result in only more accurate rules being returned, but with fewer rules being discovered.

The movie recommendation problem

Product recommendation is big business. Online stores use it to up-sell to customers by recommending other products that they could buy. Making better recommendations leads to better sales. When online shopping is selling to millions of customers every year, there is a lot of potential money to be made by selling more items to these customers.

Product recommendations have been researched for many years; however, the field gained a significant boost when Netflix ran their Netflix Prize between 2007 and 2009. This competition aimed to determine if anyone can predict a user's rating of a film better than Netflix was currently doing. The prize went to a team that was just over 10 percent better than the current solution. While this may not seem like a large improvement, such an improvement would net millions to Netflix in revenue from better movie recommendations.

Obtaining the dataset

Since the inception of the Netflix Prize, Grouplens, a research group at the University of Minnesota, has released several datasets that are often used for testing algorithms in this area. They have released several versions of a movie rating dataset, which have different sizes. There is a version with 100,000 reviews, one with 1 million reviews and one with 10 million reviews.

The datasets are available from `http://grouplens.org/datasets/movielens/` and the dataset we are going to use in this chapter is the MovieLens 1 million dataset. Download this dataset and unzip it in your data folder. Start a new IPython Notebook and type the following code:

```
import os
import pandas as pd
data_folder = os.path.join(os.path.expanduser("~"), "Data",
  "ml-100k")
ratings_filename = os.path.join(data_folder, "u.data")
```

Ensure that `ratings_filename` points to the `u.data` file in the unzipped folder.

Loading with pandas

The MovieLens dataset is in a good shape; however, there are some changes from the default options in `pandas.read_csv` that we need to make. To start with, the data is separated by tabs, not commas. Next, there is no heading line. This means the first line in the file is actually data and we need to manually set the column names.

When loading the file, we set the delimiter parameter to the tab character, tell pandas not to read the first row as the header (with `header=None`), and set the column names. Let's look at the following code:

```
all_ratings = pd.read_csv(ratings_filename, delimiter="\t",
    header=None, names = ["UserID", "MovieID", "Rating", "Datetime"])
```

While we won't use it in this chapter, you can properly parse the date timestamp using the following line:

```
all_ratings["Datetime"] = pd.to_datetime(all_ratings['Datetime'],
  unit='s')
```

You can view the first few records by running the following in a new cell:

```
all_ratings[:5]
```

The result will come out looking something like this:

	UserID	MovieID	Rating	Datetime
0	196	242	3	1997-12-04 15:55:49
1	186	302	3	1998-04-04 19:22:22
2	22	377	1	1997-11-07 07:18:36
3	244	51	2	1997-11-27 05:02:03
4	166	346	1	1998-02-02 05:33:16

Sparse data formats

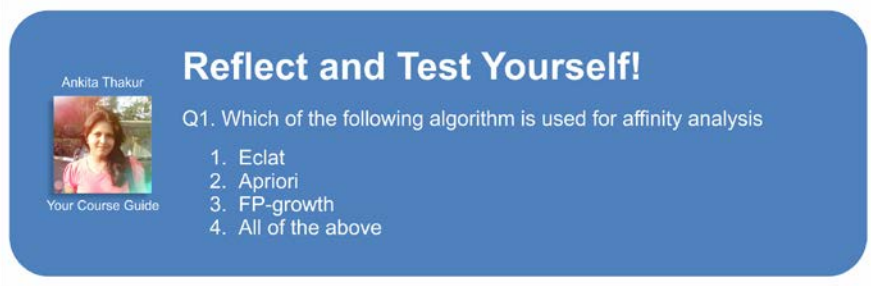
This dataset is in a sparse format. Each row can be thought of as a cell in a large feature matrix of the type used in previous chapters, where rows are users and columns are individual movies. The first column would be each user's review of the first movie, the second column would be each user's review of the second movie, and so on.

There are 1,000 users and 1,700 movies in this dataset, which means that the full matrix would be quite large. We may run into issues storing the whole matrix in memory and computing on it would be troublesome. However, this matrix has the property that most cells are empty, that is, there is no review for most movies for most users. There is no review of movie `#675` for user `#213` though, and not for most other combinations of user and movie.

The format given here represents the full matrix, but in a more compact way. The first row indicates that user `#196` reviewed movie `#242`, giving it a ranking of 3 (out of five) on the December 4, 1997.

Any combination of user and movie that isn't in this database is assumed to not exist. This saves significant space, as opposed to storing a bunch of zeroes in memory. This type of format is called a **sparse matrix** format. As a rule of thumb, if you expect about 60 percent or more of your dataset to be empty or zero, a sparse format will take less space to store.

When computing on sparse matrices, the focus isn't usually on the data we don't have—comparing all of the zeroes. We usually focus on the data we have and compare those.



The Apriori implementation

The goal of this chapter is to produce rules of the following form: *if a person recommends these movies, they will also recommend this movie*. We will also discuss extensions where a person recommends a set of movies is likely to recommend another particular movie.

To do this, we first need to determine if a person recommends a movie. We can do this by creating a new feature `Favorable`, which is `True` if the person gave a favorable review to a movie:

```
all_ratings["Favorable"] = all_ratings["Rating"] > 3
```

We can see the new feature by viewing the dataset:

```
all_ratings[10:15]
```

	UserID	MovieID	Rating	Datetime	Favorable
10	62	257	2	1997-11-12 22:07:14	False
11	286	1014	5	1997-11-17 15:38:45	True
12	200	222	5	1997-10-05 09:05:40	True
13	210	40	3	1998-03-27 21:59:54	False
14	224	29	3	1998-02-21 23:40:57	False

We will sample our dataset to form a training dataset. This also helps reduce the size of the dataset that will be searched, making the Apriori algorithm run faster. We obtain all reviews from the first 200 users:

```
ratings = all_ratings[all_ratings['UserID'].isin(range(200))]
```

Next, we can create a dataset of only the favorable reviews in our sample:

```
favorable_ratings = ratings[ratings["Favorable"]]
```

We will be searching the user's favorable reviews for our itemsets. So, the next thing we need is the movies which each user has given a favorable. We can compute this by grouping the dataset by the User ID and iterating over the movies in each group:

```
favorable_reviews_by_users = dict((k, frozenset(v.values))
                                  for k, v in favorable_ratings
                                  groupby("UserID")["MovieID"])
```

In the preceding code, we stored the values as a `frozenset`, allowing us to quickly check if a movie has been rated by a user. Sets are much faster than lists for this type of operation, and we will use them in a later code.

Finally, we can create a DataFrame that tells us how frequently each movie has been given a favorable review:

```
num_favorable_by_movie = ratings[["MovieID", "Favorable"]].
  groupby("MovieID").sum()
```

We can see the top five movies by running the following code:

```
num_favorable_by_movie.sort("Favorable", ascending=False)[:5]
```

Let's see the top five movies list:

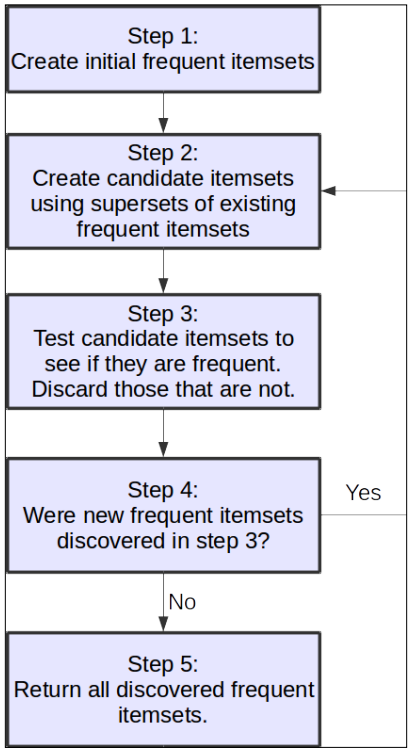
MovieID	Favorable
50	100
100	89
258	83
181	79
174	74

The Apriori algorithm

The Apriori algorithm is part of our affinity analysis and deals specifically with finding frequent itemsets within the data. The basic procedure of Apriori builds up new candidate itemsets from previously discovered frequent itemsets. These candidates are tested to see if they are frequent, and then the algorithm iterates as explained here:

1. Create initial frequent itemsets by placing each item in its own itemset. Only items with at least the minimum support are used in this step.
2. New candidate itemsets are created from the most recently discovered frequent itemsets by finding supersets of the existing frequent itemsets.
3. All candidate itemsets are tested to see if they are frequent. If a candidate is not frequent then it is discarded. If there are no new frequent itemsets from this step, go to the last step.
4. Store the newly discovered frequent itemsets and go to the second step.
5. Return all of the discovered frequent itemsets.

This process is outlined in the following workflow:



Implementation

On the first iteration of Apriori, the newly discovered itemsets will have a length of 2, as they will be supersets of the initial itemsets created in the first step. On the second iteration (after applying the fourth step), the newly discovered itemsets will have a length of 3. This allows us to quickly identify the newly discovered itemsets, as needed in second step.

We can store our discovered frequent itemsets in a dictionary, where the key is the length of the itemsets. This allows us to quickly access the itemsets of a given length, and therefore the most recently discovered frequent itemsets, with the help of the following code:

```
frequent_itemsets = {}
```

We also need to define the minimum support needed for an itemset to be considered frequent. This value is chosen based on the dataset but feel free to try different values. I recommend only changing it by 10 percent at a time though, as the time the algorithm takes to run will be significantly different! Let's apply minimum support:

```
min_support = 50
```

To implement the first step of the Apriori algorithm, we create an itemset with each movie individually and test if the itemset is frequent. We use `frozenset`, as they allow us to perform set operations later on, and they can also be used as keys in our counting dictionary (normal sets cannot). Let's look at the following code:

```
frequent_itemsets[1] = dict((frozenset((movie_id,)),
                             row["Favorable"])
                            for movie_id, row in num_favorable_
                              by_movie.iterrows()
                            if row["Favorable"] > min_support)
```

We implement the second and third steps together for efficiency by creating a function that takes the newly discovered frequent itemsets, creates the supersets, and then tests if they are frequent. First, we set up the function and the counting dictionary:

```
from collections import defaultdict
def find_frequent_itemsets(favorable_reviews_by_users, k_1_itemsets,
min_support):
    counts = defaultdict(int)
```

In keeping with our rule of thumb of reading through the data as little as possible, we iterate over the dataset once per call to this function. While this doesn't matter too much in this implementation (our dataset is relatively small), it is a good practice to get into for larger applications. We iterate over all of the users and their reviews:

```
    for user, reviews in favorable_reviews_by_users.items():
```

Next, we go through each of the previously discovered itemsets and see if it is a subset of the current set of reviews. If it is, this means that the user has reviewed each movie in the itemset. Let's look at the code:

```
        for itemset in k_1_itemsets:
            if itemset.issubset(reviews):
```

We can then go through each individual movie that the user has reviewed that isn't in the itemset, create a superset from it, and record in our counting dictionary that we saw this particular itemset. Let's look at the code:

```
                for other_reviewed_movie in reviews - itemset:
                    current_superset = itemset | frozenset((other_
                                       reviewed_movie,))
                    counts[current_superset] += 1
```

We end our function by testing which of the candidate itemsets have enough support to be considered frequent and return only those:

```
    return dict([(itemset, frequency) for itemset, frequency in
counts.items() if frequency >= min_support])
```

To run our code, we now create a loop that iterates over the steps of the Apriori algorithm, storing the new itemsets as we go. In this loop, *k* represents the length of the soon-to-be discovered frequent itemsets, allowing us to access the previously most discovered ones by looking in our `frequent_itemsets` dictionary using the key *k - 1*. We create the frequent itemsets and store them in our dictionary by their length. Let's look at the code:

```
for k in range(2, 20):
    cur_frequent_itemsets =
      find_frequent_itemsets(favorable_reviews_by_users,
        frequent_itemsets[k-1],
          min_support)
     frequent_itemsets[k] = cur_frequent_itemsets
```

We want to break out the preceding loop if we didn't find any new frequent itemsets (and also to print a message to let us know what is going on):

```
if len(cur_frequent_itemsets) == 0:
    print("Did not find any frequent itemsets of length {}".
      format(k))
    sys.stdout.flush()
    break
```

We use `sys.stdout.flush()` to ensure that the printouts happen while the code is still running. Sometimes, in large loops in particular cells, the printouts will not happen until the code has completed. Flushing the output in this way ensures that the printout happens when we want. Don't do it too much though—the flush operation carries a computational cost (as does printing) and this will slow down the program.

If we do find frequent itemsets, we print out a message to let us know the loop will be running again. This algorithm can take a while to run, so it is helpful to know that the code is still running while you wait for it to complete! Let's look at the code:

```
else:
    print("I found {} frequent itemsets of length
    {}".format(len(cur_frequent_itemsets), k))
    sys.stdout.flush()
```

Finally, after the end of the loop, we are no longer interested in the first set of itemsets anymore—these are itemsets of length one, which won't help us create association rules – we need at least two items to create association rules. Let's delete them:

```
del frequent_itemsets[1]
```

You can now run this code. It may take a few minutes, more if you have older hardware. If you find you are having trouble running any of the code samples, take a look at using an online cloud provider for additional speed. Details about using the cloud to do the work are given in *Chapter 13*, *Next Steps....*

The preceding code returns 1,718 frequent itemsets of varying lengths. You'll notice that the number of itemsets grows as the length increases before it shrinks. It grows because of the increasing number of possible rules. After a while, the large number of combinations no longer has the support necessary to be considered frequent. This results in the number shrinking. This shrinking is the benefit of the Apriori algorithm. If we search all possible itemsets (not just the supersets of frequent ones), we would be searching thousands of times more itemsets to see if they are frequent.

Extracting association rules

After the Apriori algorithm has completed, we have a list of frequent itemsets. These aren't exactly association rules, but they are similar to it. A frequent itemset is a set of items with a minimum support, while an association rule has a premise and a conclusion.

We can make an association rule from a frequent itemset by taking one of the movies in the itemset and denoting it as the conclusion. The other movies in the itemset will be the premise. This will form rules of the following form: *if a reviewer recommends all of the movies in the premise, they will also recommend the conclusion*.

For each itemset, we can generate a number of association rules by setting each movie to be the conclusion and the remaining movies as the premise.

In code, we first generate a list of all of the rules from each of the frequent itemsets, by iterating over each of the discovered frequent itemsets of each length:

```
candidate_rules = []
for itemset_length, itemset_counts in frequent_itemsets.items():
    for itemset in itemset_counts.keys():
```

We then iterate over every movie in this itemset, using it as our conclusion. The remaining movies in the itemset are the premise. We save the premise and conclusion as our candidate rule:

```
            for conclusion in itemset:
                premise = itemset - set((conclusion,))
                candidate_rules.append((premise, conclusion))
```

This returns a very large number of candidate rules. We can see some by printing out the first few rules in the list:

```
print(candidate_rules[:5])
```

The resulting output shows the rules that were obtained:

```
[(frozenset({79}), 258), (frozenset({258}), 79), (frozenset({50}),
  64), (frozenset({64}), 50), (frozenset({127}), 181)]
```

In these *rules*, the first part (the `frozenset`) is the list of movies in the premise, while the number after it is the conclusion. In the first case, if a reviewer recommends movie 79, they are also likely to recommend movie 258.

Next, we compute the confidence of each of these rules. This is performed much like in *Chapter 1*, *Getting Started with Data Mining*, with the only changes being those necessary for computing using the new data format.

The process starts by creating dictionaries to store how many times we see the premise leading to the conclusion (a *correct* example of the rule) and how many times it doesn't (an *incorrect* example). Let's look at the code:

```
correct_counts = defaultdict(int)
incorrect_counts = defaultdict(int)
```

We iterate over all of the users, their favorable reviews, and over each candidate association rule:

```
for user, reviews in favorable_reviews_by_users.items():
    for candidate_rule in candidate_rules:
        premise, conclusion = candidate_rule
```

We then test to see if the premise is applicable to this user. In other words, did the user favorably review all of the movies in the premise? Let's look at the code:

```
        if premise.issubset(reviews):
```

If the premise applies, we see if the conclusion movie was also rated favorably. If so, the rule is correct in this instance. If not, it is incorrect. Let's look at the code:

```
        if premise.issubset(reviews):
            if conclusion in reviews:
                correct_counts[candidate_rule] += 1
            else:
                incorrect_counts[candidate_rule] += 1
```

We then compute the confidence for each rule by dividing the correct count by the total number of times the rule was seen:

```
rule_confidence = {candidate_rule: correct_counts[candidate_rule]
  / float(correct_counts[candidate_rule] +
    incorrect_counts[candidate_rule])
              for candidate_rule in candidate_rules}
```

Now we can print the top five rules by sorting this confidence dictionary and printing the results:

```
from operator import itemgetter
sorted_confidence = sorted(rule_confidence.items(),
  key=itemgetter(1), reverse=True)
for index in range(5):
    print("Rule #{0}".format(index + 1))
    (premise, conclusion) = sorted_confidence[index][0]
```

```
        print("Rule: If a person recommends {0} they will also
          recommend {1}".format(premise, conclusion))
        print(" - Confidence:
          {0:.3f}".format(rule_confidence[(premise, conclusion)]))
        print("")
```

The result is as follows:

```
Rule #1
Rule: If a person recommends frozenset({64, 56, 98, 50, 7}) they will
also recommend 174
 - Confidence: 1.000

Rule #2
Rule: If a person recommends frozenset({98, 100, 172, 79, 50, 56})
they will also recommend 7
 - Confidence: 1.000

Rule #3
Rule: If a person recommends frozenset({98, 172, 181, 174, 7}) they
will also recommend 50
 - Confidence: 1.000

Rule #4
Rule: If a person recommends frozenset({64, 98, 100, 7, 172, 50}) they
will also recommend 174
 - Confidence: 1.000

Rule #5
Rule: If a person recommends frozenset({64, 1, 7, 172, 79, 50}) they
will also recommend 181
 - Confidence: 1.000
```

The resulting printout shows only the movie IDs, which isn't very helpful without the names of the movies also. The dataset came with a file called `u.items`, which stores the movie names and their corresponding `MovieID` (as well as other information, such as the genre).

We can load the titles from this file using pandas. Additional information about the file and categories is available in the **README** that came with the dataset. The data in the files is in CSV format, but with data separated by the `|` symbol; it has no header and the encoding is important to set. The column names were found in the `README` file.

```
movie_name_filename = os.path.join(data_folder, "u.item")
movie_name_data = pd.read_csv(movie_name_filename, delimiter="|",
  header=None, encoding = "mac-roman")
```

```
movie_name_data.columns = ["MovieID", "Title", "Release Date",
  "Video Release", "IMDB", "<UNK>", "Action", "Adventure",
    "Animation", "Children's", "Comedy", "Crime", "Documentary",
      "Drama", "Fantasy", "Film-Noir",
  "Horror", "Musical", "Mystery", "Romance", "Sci-Fi", "Thriller",
     "War", "Western"]
```

Getting the movie title is important, so we will create a function that will return a movie's title from its `MovieID`, saving us the trouble of looking it up each time. Let's look at the code:

```
def get_movie_name(movie_id):
```

We look up the `movie_name_data` DataFrame for the given `MovieID` and return only the title column:

```
    title_object = movie_name_data[movie_name_data["MovieID"] ==
      movie_id]["Title"]
```

We use the values parameter to get the actual value (and not the pandas `Series` object that is currently stored in `title_object`). We are only interested in the first value—there should only be one title for a given `MovieID` anyway!

```
    title = title_object.values[0]
```

We end the function by returning the title as needed. Let's look at the code:

```
    return title
```

In a new IPython Notebook cell, we adjust our previous code for printing out the top rules to also include the titles:

```
for index in range(5):
    print("Rule #{0}".format(index + 1))
    (premise, conclusion) = sorted_confidence[index][0]
    premise_names = ", ".join(get_movie_name(idx) for idx
      in premise)
    conclusion_name = get_movie_name(conclusion)
    print("Rule: If a person recommends {0} they will
      also recommend {1}".format(premise_names, conclusion_name))
    print(" - Confidence: {0:.3f}".format(confidence[(premise,
      conclusion)]))
    print("")
```

The result is much more readable (there are still some issues, but we can ignore them for now):

```
Rule #1
Rule: If a person recommends Shawshank Redemption, The (1994), Pulp
Fiction (1994), Silence of the Lambs, The (1991), Star Wars (1977),
Twelve Monkeys (1995) they will also recommend Raiders of the Lost Ark
(1981)
 - Confidence: 1.000

Rule #2
Rule: If a person recommends Silence of the Lambs, The (1991), Fargo
(1996), Empire Strikes Back, The (1980), Fugitive, The (1993), Star
Wars (1977), Pulp Fiction (1994) they will also recommend Twelve
Monkeys (1995)
 - Confidence: 1.000

Rule #3
Rule: If a person recommends Silence of the Lambs, The (1991), Empire
Strikes Back, The (1980), Return of the Jedi (1983), Raiders of the
Lost Ark (1981), Twelve Monkeys (1995) they will also recommend Star
Wars (1977)
 - Confidence: 1.000

Rule #4
Rule: If a person recommends Shawshank Redemption, The (1994), Silence
of the Lambs, The (1991), Fargo (1996), Twelve Monkeys (1995), Empire
Strikes Back, The (1980), Star Wars (1977) they will also recommend
Raiders of the Lost Ark (1981)
 - Confidence: 1.000

Rule #5
Rule: If a person recommends Shawshank Redemption, The (1994), Toy
Story (1995), Twelve Monkeys (1995), Empire Strikes Back, The (1980),
Fugitive, The (1993), Star Wars (1977) they will also recommend Return
of the Jedi (1983)
 - Confidence: 1.000
```

Evaluation

In a broad sense, we can evaluate the association rules using the same concept as for classification. We use a test set of data that was not used for training, and evaluate our discovered rules based on their performance in this test set.

To do this, we will compute the test set confidence, that is, the confidence of each rule on the testing set.

We won't apply a formal evaluation metric in this case; we simply examine the rules and look for good examples.

First, we extract the test dataset, which is all of the records we didn't use in the training set. We used the first 200 users (by ID value) for the training set, and we will use all of the rest for the testing dataset. As with the training set, we will also get the favorable reviews for each of the users in this dataset as well. Let's look at the code:

```
test_dataset =
all_ratings[~all_ratings['UserID'].isin(range(200))]
test_favorable = test_dataset[test_dataset["Favorable"]]
test_favorable_by_users = dict((k, frozenset(v.values)) for k, v
  in test_favorable.groupby("UserID")["MovieID"])
```

We then count the correct instances where the premise leads to the conclusion, in the same way we did before. The only change here is the use of the test data instead of the training data. Let's look at the code:

```
correct_counts = defaultdict(int)
incorrect_counts = defaultdict(int)
for user, reviews in test_favorable_by_users.items():
    for candidate_rule in candidate_rules:
        premise, conclusion = candidate_rule
        if premise.issubset(reviews):
            if conclusion in reviews:
                correct_counts[candidate_rule] += 1
            else:
                incorrect_counts[candidate_rule] += 1
```

Next, we compute the confidence of each rule from the correct counts. Let's look at the code:

```
test_confidence = {candidate_rule: correct_counts[candidate_rule]
  / float(correct_counts[candidate_rule] + incorrect_counts
    [candidate_rule])
  for candidate_rule in rule_confidence}
```

Finally, we print out the best association rules with the titles instead of the movie IDs.

```
for index in range(5):
    print("Rule #{0}".format(index + 1))
```

```
    (premise, conclusion) = sorted_confidence[index][0]
    premise_names = ", ".join(get_movie_name(idx) for idx in
      premise)
    conclusion_name = get_movie_name(conclusion)
    print("Rule: If a person recommends {0} they will also
      recommend {1}".format(premise_names, conclusion_name))
    print(" - Train Confidence:
      {0:.3f}".format(rule_confidence.get((premise, conclusion),
        -1)))
    print(" - Test Confidence:
     {0:.3f}".format(test_confidence.get((premise, conclusion),
        -1)))
    print("")
```

We can now see which rules are most applicable in new unseen data:

```
Rule #1
Rule: If a person recommends Shawshank Redemption, The (1994), Pulp
Fiction (1994), Silence of the Lambs, The (1991), Star Wars (1977),
Twelve Monkeys (1995) they will also recommend Raiders of the Lost Ark
(1981)
 - Train Confidence: 1.000
 - Test Confidence: 0.909

Rule #2
Rule: If a person recommends Silence of the Lambs, The (1991), Fargo
(1996), Empire Strikes Back, The (1980), Fugitive, The (1993), Star
Wars (1977), Pulp Fiction (1994) they will also recommend Twelve
Monkeys (1995)
 - Train Confidence: 1.000
 - Test Confidence: 0.609

Rule #3
Rule: If a person recommends Silence of the Lambs, The (1991), Empire
Strikes Back, The (1980), Return of the Jedi (1983), Raiders of the
Lost Ark (1981), Twelve Monkeys (1995) they will also recommend Star
Wars (1977)
 - Train Confidence: 1.000
 - Test Confidence: 0.946

Rule #4
Rule: If a person recommends Shawshank Redemption, The (1994), Silence
of the Lambs, The (1991), Fargo (1996), Twelve Monkeys (1995), Empire
Strikes Back, The (1980), Star Wars (1977) they will also recommend
Raiders of the Lost Ark (1981)
```

```
 - Train Confidence: 1.000  - Test Confidence: 0.971

Rule #5
Rule: If a person recommends Shawshank Redemption, The (1994), Toy
Story (1995), Twelve Monkeys (1995), Empire Strikes Back, The (1980),
Fugitive, The (1993), Star Wars (1977) they will also recommend Return
of the Jedi (1983)
 - Train Confidence: 1.000
 - Test Confidence: 0.900
```

The second rule, for instance, has a perfect confidence in the training data, but it is only accurate in 60 percent of cases for the test data. Many of the other rules in the top 10 have high confidences in test data though, making them good rules for making recommendations.

If you are looking through the rest of the rules, some will have a test confidence of -1. Confidence values are always between 0 and 1. This value indicates that the particular rule wasn't found in the test dataset at all.

Your Coding Challenge

There are many recommendation-based datasets that are worth investigating, each with its own issues. For example, the Book-Crossing dataset contains more than 278,000 users and over a million ratings. Some of these ratings are explicit (the user did give a rating), while others are more implicit. The weighting to these implicit ratings probably shouldn't be as high as for explicit ratings.

The music website `www.last.fm` has released a great dataset for music recommendation: `http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/`.

There is also a joke recommendation dataset! See here: `http://eigentaste.berkeley.edu/dataset/`.

Try experimenting around with these datasets!

Summary of Module 3 Chapter 4

In this chapter, we performed affinity analysis in order to recommend movies based on a large set of reviewers. We did this in two stages. First, we found frequent itemsets in the data using the Apriori algorithm. Then, we created association rules from those itemsets.

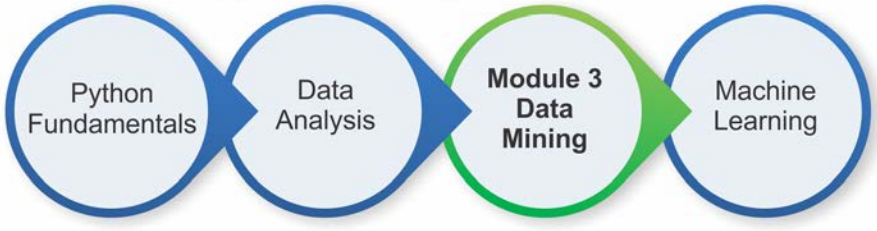
Ankita Thakur

Your Course Guide

The use of the Apriori algorithm was necessary due to the size of the dataset. While in *chapter 1, Getting Started With Data Mining*, we used a brute-force approach, the exponential growth in the time needed to compute those rules required a smarter approach. This is a common pattern for data mining: we can solve many problems in a brute force manner, but smarter algorithms allow us to apply the concepts to larger datasets.

We performed training on a subset of our data in order to find the association rules, and then tested those rules on the rest of the data—a testing set. From what we discussed in the previous chapters, we could extend this concept to use cross-fold validation to better evaluate the rules. This would lead to a more robust evaluation of the quality of each rule.

Your Progress through the Course So Far



5
Extracting Features with Transformers

The datasets we have used so far have been described in terms of features. In the previous chapter, we used a transaction-centric dataset. However, ultimately this was just a different format for representing feature-based data.

There are many other types of datasets, including text, images, sounds, movies, or even real objects. Most data mining algorithms, however, rely on having numerical or categorical features. This means we need a way to represent these types before we input them into the data mining algorithm.

In this chapter, we will discuss how to extract numerical and categorical features, and choose the best features when we do have them. We will discuss some common patterns and techniques for extracting features.

The key concepts introduced in this chapter include:

- Extracting features from datasets
- Creating new features
- Selecting good features
- Creating your own transformer for custom datasets

Feature extraction

Extracting features is one of the most critical tasks in data mining, and it generally affects your end result more than the choice of data mining algorithm. Unfortunately, there are no hard and fast rules for choosing features that will result in high performance data mining. In many ways, this is where the science of data mining becomes more of an art. Creating good features relies on intuition, domain expertise, data mining experience, trial and error, and sometimes a little luck.

Representing reality in models

Not all datasets are presented in terms of features. Sometimes, a dataset consists of nothing more than all of the books that have been written by a given author. Sometimes, it is the film of each of the movies released in 1979. At other times, it is a library collection of interesting historical artifacts.

From these datasets, we may want to perform a data mining task. For the books, we may want to know the different categories that the author writes. In the films, we may wish to see how women are portrayed. In the historical artifacts, we may want to know whether they are from one country or another. It isn't possible to just pass these raw datasets into a decision tree and see what the result is.

For a data mining algorithm to assist us here, we need to represent these as features. Features are a way to create a model and the model provides an approximation of reality in a way that data mining algorithms can understand. Therefore, a model is just a simplified version of some aspect of the real world. As an example, the game of chess is a simplified model for historical warfare.

Selecting features has another advantage: they reduce the complexity of the real world into a more manageable model. Imagine how much information it would take to properly, accurately, and fully describe a real-world object to someone that has no background knowledge of the item. You would need to describe the size, weight, texture, composition, age, flaws, purpose, origin, and so on.

The complexity of real objects is too much for current algorithms, so we use these simpler models instead.

This simplification also focuses our intent in the data mining application. In later chapters, we will look at clustering and where it is critically important. If you put random features in, you will get random results out.

However, there is a downside as this simplification reduces the detail, or may remove good indicators of the things we wish to perform data mining on.

Thought should always be given to how to represent reality in the form of a model. Rather than just using what has been used in the past, you need to consider the goal of the data mining exercise. What are you trying to achieve? In *Chapter 3*, *Predicting Sports Winners with Decision Trees*, we created features by thinking about the goal (predicting winners) and used a little domain knowledge to come up with ideas for new features.

Not all features need to be numeric or categorical. Algorithms have been developed that work directly on text, graphs, and other data structures. Unfortunately, those algorithms are outside the scope of this module. In this module, we mainly use numeric or categorical features.

The `Adult` dataset is a great example of taking a complex reality and attempting to model it using features. In this dataset, the aim is to estimate if someone earns more than $50,000 per year. To download the dataset, navigate to `http://archive.ics.uci.edu/ml/datasets/Adult` and click on the **Data Folder** link. Download the `adult.data` and `adult.names` into a directory named `Adult` in your data folder.

This dataset takes a complex task and describes it in features. These features describe the person, their environment, their background, and their life status.

Open a new IPython Notebook for this chapter and set the data's filename and import pandas to load the file:

```
import os
import pandas as pd
data_folder = os.path.join(os.path.expanduser("~"), "Data",
  "Adult")
adult_filename = os.path.join(data_folder, "adult.data")
Using pandas as before, we load the file with read_csv:
adult = pd.read_csv(adult_filename, header=None,
    names=["Age", "Work-Class", "fnlwgt",
    "Education", "Education-Num",
    "Marital-Status", "Occupation",
    "Relationship", "Race", "Sex",
    "Capital-gain", "Capital-loss",
    "Hours-per-week", "Native-Country",
    "Earnings-Raw"])
```

Most of the code is the same as in the previous chapters.

The `adult` file itself contains two blank lines at the end of the file. By default, pandas will interpret the penultimate new line to be an empty (but valid) row. To remove this, we remove any line with invalid numbers (the use of `inplace` just makes sure the same Dataframe is affected, rather than creating a new one):

```
adult.dropna(how='all', inplace=True)
```

Having a look at the dataset, we can see a variety of features from `adult.columns`:

```
adult.columns
```

The results show each of the feature names that are stored inside an `Index` object from `pandas`:

```
Index(['Age', 'Work-Class', 'fnlwgt', 'Education',
'Education-Num', 'Marital-Status', 'Occupation', 'Relationship',
'Race', 'Sex', 'Capital-gain', 'Capital-loss', 'Hours-per-week',
'Native-Country', 'Earnings-Raw'], dtype='object')
```

Common feature patterns

While there are millions of ways to create features, there are some common patterns that are employed across different disciplines. However, choosing appropriate features is tricky and it is worth considering how a feature might correlate to the end result. As the adage says, don't judge a book by its cover—it is probably not worth considering the size of a book if you are interested in the message contained within.

Some commonly used features focus on the physical properties of the real world objects being studied, for example:

- Spatial properties such as the length, width, and height of an object
- Weight and/or density of the object
- Age of an object or its components
- The type of the object
- The quality of the object

Other features might rely on the usage or history of the object:

- The producer, publisher, or creator of the object
- The year of manufacturing
- The use of the object

Other features describe a dataset in terms of its components:

- Frequency of a given subcomponent, such as a word in a book
- Number of subcomponents and/or the number of different subcomponents
- Average size of the subcomponents, such as the average sentence length

Ordinal features allow us to perform ranking, sorting, and grouping of similar values. As we have seen in previous chapters, features can be numerical or categorical. Numerical features are often described as being **ordinal**. For example, three people, Alice, Bob and Charlie, may have heights of 1.5 m, 1.6 m and 1.7 m. We would say that Alice and Bob are more similar in height than are Alice and Charlie.

The Adult dataset that we loaded in the last section contains examples of continuous, ordinal features. For example, the `Hours-per-week` feature tracks how many hours per week people work. Certain operations make sense on a feature like this. They include computing the mean, standard deviation, minimum and maximum. There is a function in pandas for giving some basic summary stats of this type:

```
adult["Hours-per-week"].describe()
```

The result tells us a little about this feature.

```
count    32561.000000
mean        40.437456
std         12.347429
min          1.000000
25%         40.000000
50%         40.000000
75%         45.000000
max         99.000000
dtype: float64
```

Some of these operations do not make sense for other features. For example, it doesn't make sense to compute the sum of the education statuses.

There are also features that are not numerical, but still ordinal. The `Education` feature in the Adult dataset is an example of this. For example, a Bachelor's degree is a higher education status than finishing high school, which is a higher status than not completing high school. It doesn't quite make sense to compute the mean of these values, but we can create an approximation by taking the median value. The dataset gives a helpful feature `Education-Num`, which assigns a number that is basically equivalent to the number of years of education completed. This allows us to quickly compute the median:

```
adult["Education-Num"].median()
```

The result is 10, or finishing one year past high school. If we didn't have this, we could compute the median by creating an ordering over the education values.

Features can also be categorical. For instance, a ball can be a *tennis ball*, *cricket ball*, *football*, or any other type of ball. Categorical features are also referred to as nominal features. For nominal features, the values are either the same or they are different. While we could rank balls by size or weight, just the category alone isn't enough to compare things. A tennis ball is not a cricket ball, and it is also not a football. We could argue that a tennis ball is more similar to a cricket ball (say, in size), but the category alone doesn't differentiate this—they are the same, or they are not.

We can convert categorical features to numerical features using the one-hot encoding, as we saw in *Chapter 3*, *Predicting Sports Winners with Decision Trees*. For the aforementioned categories of balls, we can create three new binary features: `is a tennis ball`, `is a cricket ball`, and `is a football`. For a tennis ball, the vector would be [1, 0, 0]. A cricket ball has the values [0, 1, 0], while a football has the values [0, 0, 1]. These features are binary, but can be used as continuous features by many algorithms. One key reason for doing this is that it easily allows for direct numerical comparison (such as computing the distance between samples).

The Adult dataset contains several categorical features, with `Work-Class` being one example. While we could argue that some values are of higher rank than others (for instance, a person with a job is likely to have a better income than a person without), it doesn't make sense for all values. For example, a person working for the state government is not more or less likely to have a higher income than someone working in the private sector.

We can view the unique values for this feature in the dataset using the `unique()` function:

```
adult["Work-Class"].unique()
```

The result shows the unique values in this column:

```
array([' State-gov', ' Self-emp-not-inc', ' Private', ' Federal-gov',
       ' Local-gov', ' ?', ' Self-emp-inc', ' Without-pay',
       ' Never-worked', nan], dtype=object)
```

There are some missing values in the preceding dataset, but they won't affect our computations in this example.

Similarly, we can convert numerical features to categorical features through a process called **discretization**, as we saw in *Chapter 4*, *Recommending Movies Using Affiity Analysis*. We can call any person who is taller than 1.7 m tall, and any person shorter than 1.7 m short. This gives us a categorical feature (although still an ordinal one). We do lose some data here. For instance, two people, one 1.69 m tall and one 1.71 m, will be in two different categories and considered *drastically* different from each other. In contrast, a person 1.2 m tall will be considered "of roughly the same height" as the person 1.69 m tall! This loss of detail is a side effect of discretization, and it is an issue that we deal with when creating models.

In the Adult dataset, we can create a `LongHours` feature, which tells us if a person works more than 40 hours per week. This turns our continuous feature (`Hours-per-week`) into a categorical one:

```
adult["LongHours"] = adult["Hours-per-week"] > 40
```

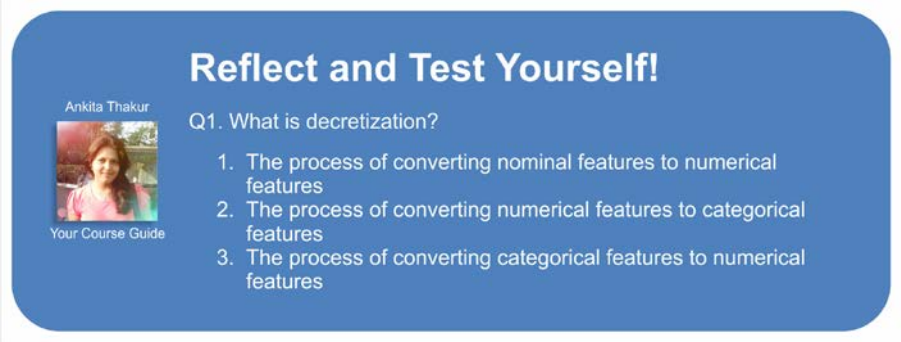
Creating good features

Modeling, and the loss of information that the simplification causes, are the reasons why we do not have data mining methods that can just be applied to any dataset. A good data mining practitioner will have, or obtain, domain knowledge in the area they are applying data mining to. They will look at the problem, the available data, and come up with a model that represents what they are trying to achieve.

For instance, a *height* feature may describe one component of a person, but may not describe their academic performance well. If we were attempting to predict a person's grade, we may not bother measuring each person's height.

This is where data mining becomes more art than science. Extracting good features is difficult and is the topic of significant and ongoing research. Choosing better classification algorithms can improve the performance of a data mining application, but choosing better features is often a better option.

In all data mining applications, you should first outline what you are looking for before you start designing the methodology that will find it. This will dictate the types of features you are aiming for, the types of algorithms that you can use, and the expectations on the final result.



Feature selection

We will often have a large number of features to choose from, but we wish to select only a small subset. There are many possible reasons for this:

- **Reducing complexity**: Many data mining algorithms need more time and resources with increase in the number of features. Reducing the number of features is a great way to make an algorithm run faster or with fewer resources.
- **Reducing noise**: Adding extra features doesn't always lead to better performance. Extra features may confuse the algorithm, finding correlations and patterns that don't have meaning (this is common in smaller datasets). Choosing only the appropriate features is a good way to reduce the chance of random correlations that have no real meaning.
- **Creating readable models**: While many data mining algorithms will happily compute an answer for models with thousands of features, the results may be difficult to interpret for a human. In these cases, it may be worth using fewer features and creating a model that a human can understand.

Some classification algorithms can handle data with issues such as these. Getting the data right and getting the features to effectively describe the dataset you are modeling can still assist algorithms.

There are some basic tests we can perform, such as ensuring that the features are at least different. If a feature's values are all same, it can't give us extra information to perform our data mining.

The `VarianceThreshold` transformer in scikit-learn, for instance, will remove any feature that doesn't have at least a minimum level of variance in the values. To show how this works, we first create a simple matrix using NumPy:

```
import numpy as np
X = np.arange(30).reshape((10, 3))
```

The result is the numbers zero to 29, in three columns and 10 rows. This represents a synthetic dataset with 10 samples and three features:

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23],
       [24, 25, 26],
       [27, 28, 29]])
```

Then, we set the entire second column/feature to the value 1:

```
X[:,1] = 1
```

The result has lots of variance in the first and third rows, but no variance in the second row:

```
array([[ 0,  1,  2],
       [ 3,  1,  5],
       [ 6,  1,  8],
       [ 9,  1, 11],
       [12,  1, 14],
       [15,  1, 17],
       [18,  1, 20],
       [21,  1, 23],
       [24,  1, 26],
       [27,  1, 29]])
```

We can now create a `VarianceThreshold` transformer and apply it to our dataset:

```
from sklearn.feature_selection import VarianceThreshold
vt = VarianceThreshold()
Xt = vt.fit_transform(X)
```

Now, the result `Xt` does not have the second column:

```
array([[ 0,  2],
       [ 3,  5],
       [ 6,  8],
       [ 9, 11],
       [12, 14],
       [15, 17],
       [18, 20],
       [21, 23],
       [24, 26],
       [27, 29]])
```

We can observe the variances for each column by printing the `vt.variances_` attribute:

```
print(vt.variances_)
```

The result shows that while the first and third column contains at least some information, the second column had no variance:

```
array([ 74.25,   0.  ,  74.25])
```

A simple and obvious test like this is always good to run when seeing data for the first time. Features with no variance do not add any value to a data mining application; however, they can slow down the performance of the algorithm.

Selecting the best individual features

If we have a number of features, the problem of finding the best subset is a difficult task. It relates to solving the data mining problem itself, multiple times. As we saw in *Chapter 4*, *Recommending Movies Using Affinity Analysis*, subset-based tasks increase exponentially as the number of features increase. This exponential growth in time needed is also true for finding the best subset of features.

A workaround to this problem is not to look for a subset that works well together, rather than just finding the best individual features. This **univariate** feature selection gives us a score based on how well a feature performs by itself. This is usually done for classification tasks, and we generally measure some type of correlation between a variable and the target class.

The `scikit-learn` package has a number of transformers for performing univariate feature selection. They include `SelectKBest`, which returns the *k* best performing features, and `SelectPercentile`, which returns the top *r%* of features. In both cases, there are a number of methods of computing the quality of a feature.

There are many different methods to compute how effectively a single feature correlates with a class value. A commonly used method is the chi-squared ($\chi 2$) test. Other methods include mutual information and entropy.

We can observe single-feature tests in action using our `Adult` dataset. First, we extract a dataset and class values from our pandas `DataFrame`. We get a selection of the features:

```
X = adult[["Age", "Education-Num", "Capital-gain", "Capital-loss",
  "Hours-per-week"]].values
```

We will also create a target class array by testing whether the `Earnings-Raw` value is above $50,000 or not. If it is, the class will be `True`. Otherwise, it will be `False`. Let's look at the code:

```
y = (adult["Earnings-Raw"] == ' >50K').values
```

Next, we create our transformer using the `chi2` function and a `SelectKBest` transformer:

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
transformer = SelectKBest(score_func=chi2, k=3)
```

Running `fit_transform` will call fit and then transform with the same dataset. The result will create a new dataset, choosing only the best three features. Let's look at the code:

```
Xt_chi2 = transformer.fit_transform(X, y)
```

The resulting matrix now only contains three features. We can also get the scores for each column, allowing us to find out which features were used. Let's look at the code:

```
print(transformer.scores_)
```

The printed results give us these scores:

```
[  8.60061182e+03   2.40142178e+03   8.21924671e+07   1.37214589e+06
   6.47640900e+03]
```

The highest values are for the first, third, and fourth columns correlates to the `Age`, `Capital-Gain`, and `Capital-Loss` features. Based on a univariate feature selection, these are the best features to choose.

If you'd like to find out more about the features in the Adult dataset, take a look at the `adult.names` file that comes with the dataset and the academic paper it references.

We could also implement other correlations, such as the Pearson's correlation coefficient. This is implemented in SciPy, a library used for scientific computing (scikit-learn uses it as a base).

If scikit-learn is working on your computer, so is SciPy. You do not need to install anything further to get this sample working.

First, we import the `pearsonr` function from SciPy:

```
from scipy.stats import pearsonr
```

The preceding function almost fits the interface needed to be used in scikit-learn's univariate transformers. The function needs to accept two arrays (`x` and `y` in our example) as parameters and returns two arrays, the scores for each feature and the corresponding p-values. The `chi2` function we used earlier only uses the required interface, which allowed us to just pass it directly to SelectKBest.

The `pearsonr` function in SciPy accepts two arrays; however, the X array it accepts is only one dimension. We will write a wrapper function that allows us to use this for multivariate arrays like the one we have. Let's look at the code:

```
def multivariate_pearsonr(X, y):
```

We create our `scores` and `pvalues` arrays, and then iterate over each column of the dataset:

```
    scores, pvalues = [], []
    for column in range(X.shape[1]):
```

We compute the Pearson correlation for this column only and the record both the score and p-value.

```
        cur_score, cur_p = pearsonr(X[:,column], y)
        scores.append(abs(cur_score))
        pvalues.append(cur_p)
```

The Pearson value could be between `-1` and `1`. A value of `1` implies a perfect correlation between two variables, while a value of -1 implies a perfect negative correlation, that is, high values in one variable give low values in the other and vice versa. Such features are really useful to have, but would be discarded. For this reason, we have stored the absolute value in the `scores` array, rather than the original signed value.

Finally, we return the scores and `p-values` in a tuple:

```
    return (np.array(scores), np.array(pvalues))
```

Now, we can use the transformer class as before to rank the features using the Pearson correlation coefficient:

```
transformer = SelectKBest(score_func=multivariate_pearsonr, k=3)
Xt_pearson = transformer.fit_transform(X, y)
print(transformer.scores_)
```

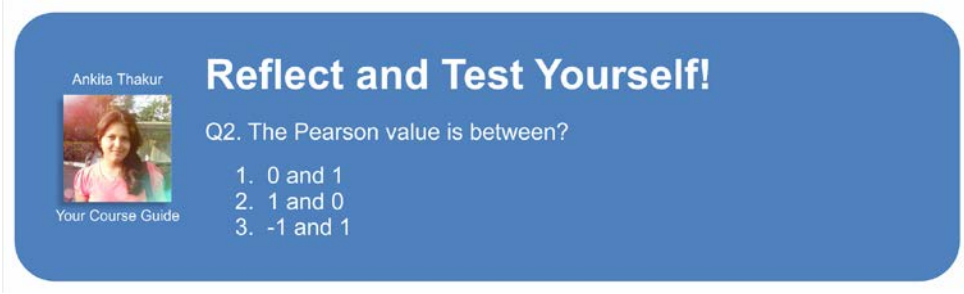
This returns a different set of features! The features chosen this way are the first, second, and fifth columns: the `Age`, `Education`, and `Hours-per-week` worked. This shows that there is not a definitive answer to what the best features are— it depends on the metric.

We can see which feature set is better by running them through a classifier. Keep in mind that the results only indicate which subset is better for a particular classifier and/or feature combination—there is rarely a case in data mining where one method is strictly better than another in all cases! Let's look at the code:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.cross_validation import cross_val_score
clf = DecisionTreeClassifier(random_state=14)
scores_chi2 = cross_val_score(clf, Xt_chi2, y, scoring='accuracy')
scores_pearson = cross_val_score(clf, Xt_pearson, y,
  scoing='accuracy')
```

The `chi2` average here is 0.83, while the Pearson score is lower at 0.77. For this combination, `chi2` returns better results!

It is worth remembering the goal of this data mining activity: predicting wealth. Using a combination of good features and feature selection, we can achieve 83 percent accuracy using just three features of a person!



Feature creation

Sometimes, just selecting features from what we have isn't enough. We can create features in different ways from features we already have. The one-hot encoding method we saw previously is an example of this. Instead of having a category features with options *A*, *B* and *C*, we would create three new features *Is it A?*, *Is it B?*, *Is it C?*.

Creating new features may seem unnecessary and to have no clear benefit—after all, the information is already in the dataset and we just need to use it. However, some algorithms struggle when features correlate significantly, or if there are redundant features. They may also struggle if there are redundant features.

For this reason, there are various ways to create new features from the features we already have.

We are going to load a new dataset, so now is a good time to start a new IPython Notebook. Download the Advertisements dataset from `http://archive.ics.uci.edu/ml/datasets/Internet+Advertisements` and save it to your `Data` folder.

Next, we need to load the dataset with pandas. First, we set the data's filename as always:

```
import os
import numpy as np
import pandas as pd
data_folder = os.path.join(os.path.expanduser("~"), "Data")
data_filename = os.path.join(data_folder, "Ads", "ad.data")
```

There are a couple of issues with this dataset that stop us from loading it easily. First, the first few features are numerical, but pandas will load them as strings. To fix this, we need to write a converting function that will convert strings to numbers if possible. Otherwise, we will get a **NaN** (which is short for **Not a Number**), which is a special value that indicates that the value could not be interpreted as a number. It is similar to *none* or *null* in other programming languages.

Another issue with this dataset is that some values are missing. These are represented in the dataset using the string `?`. Luckily, the question mark doesn't convert to a float, so we can convert those to NaNs using the same concept. In further chapters, we will look at other ways of dealing with missing values like this.

We will create a function that will do this conversion for us:

```
def convert_number(x):
```

First, we want to convert the string to a number and see if that fails. Then, we will surround the conversion in a `try/except` block, catching a `ValueError` exception (which is what is thrown if a string cannot be converted into a number this way):

```
    try:
        return float(x)
    except ValueError:
```

Finally, if the conversion failed, we get a NaN that comes from the NumPy library we imported previously:

```
        return np.nan
```

Now, we create a dictionary for the conversion. We want to convert all of the features to floats:

```
converters = defaultdict(convert_number
```

Also, we want to set the final column (column index #1558), which is the class, to a binary feature. In the Adult dataset, we created a new feature for this. In the dataset, we will convert the feature while we load it.

```
converters[1558] = lambda x: 1 if x.strip() == "ad." else 0
```

Now we can load the dataset using `read_csv`. We use the converters parameter to pass our custom conversion into pandas:

```
ads = pd.read_csv(data_filename, header=None, converters=converters)
```

The resulting dataset is quite large, with 1,559 features and more than 2,000 rows. Here are some of the feature values the first five, printed by inserting `ads[:5]` into a new cell:

	0	1	2	3	4	5	6	7	8	9	...	1549	1550	1551	1552	1553	1554	1555	1556	1557	1558
0	125	125	1.0000	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
1	57	468	8.2105	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
2	33	230	6.9696	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
3	60	468	7.8000	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
4	60	468	7.8000	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1

This dataset describes images on websites, with the goal of determining whether a given image is an advertisement or not.

The features in this dataset are not described well by their headings. There are two files accompanying the `ad.data` file that have more information: `ad.DOCUMENTATION` and `ad.names`. The first three features are the height, width, and ratio of the image size. The final feature is 1 if it is an advertisement and 0 if it is not.

The other features are 1 for the presence of certain words in the URL, alt text, or caption of the image. These words, such as the word *sponsor*, are used to determine if the image is likely to be an advertisement. Many of the features overlap considerably, as they are combinations of other features. Therefore, this dataset has a lot of redundant information.

With our dataset loaded in pandas, we will now extract the `x` and `y` data for our classification algorithms. The `x` matrix will be all of the columns in our Dataframe, except for the last column. In contrast, the y array will be only that last column, feature #1558. Let's look at the code:

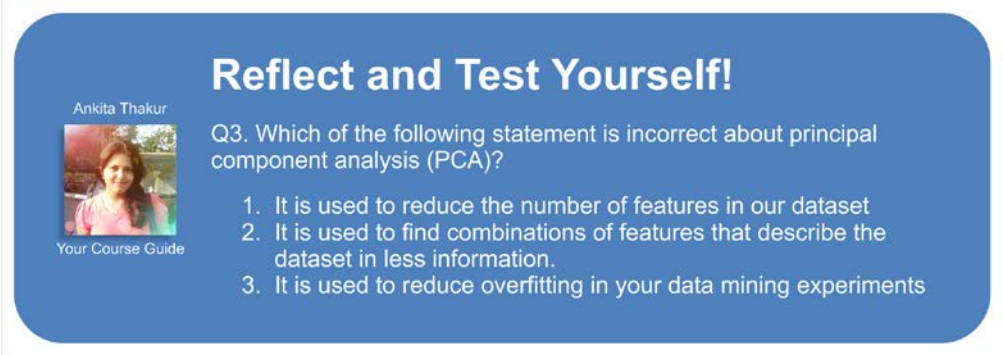
```
X = ads.drop(1558, axis=1).values
y = ads[1558]
```

Creating your own transformer

As the complexity and type of dataset changes, you might find that you can't find an existing feature extraction transformer that fits your needs. We will see an example of this in *Chapter 7*, *Discovering Accounts to Follow Using Graph Mining*, where we create new features from graphs.

A transformer is akin to a converting function. It takes data of one form as input and returns data of another form as output. Transformers can be trained using some training dataset, and these trained parameters can be used to convert testing data.

The transformer API is quite simple. It takes data of a specific format as input and returns data of another format (either the same as the input or different) as output. Not much else is required of the programmer.



The transformer API

Transformers have two key functions:

- `fit()`: This takes a training set of data as input and sets internal parameters
- `transform()`: This performs the transformation itself. This can take either the training dataset, or a new dataset of the same format

Both `fit()` and `transform()` fuction should take the same data type as input, but `transform()` can return data of a different type.

We are going to create a trivial transformer to show the API in action. The transformer will take a NumPy array as input, and discretize it based on the mean. Any value higher than the mean (of the training data) will be given the value 1 and any value lower or equal to the mean will be given the value 0.

We did a similar transformation with the Adult dataset using pandas: we took the `Hours-per-week` feature and created a `LongHours` feature if the value was more than 40 hours per week. This transformer is different for two reasons. First, the code will conform to the scikit-learn API, allowing us to use it in a pipeline. Second, the code will *learn* the mean, rather than taking it as a fixed value (such as 40 in the `LongHours` example).

Implementation details

To start, open up the IPython Notebook that we used for the Adult dataset. Then, click on the **Cell menu** item and choose **Run All**. This will rerun all of the cells and ensure that the notebook is up to date.

First, we import the `TransformerMixin`, which sets the API for us. While Python doesn't have strict interfaces (as opposed to languages like Java), using a `mixin` like this allows scikit-learn to determine that the class is actually a transformer. We also need to import a function that checks the input is of a valid type. We will use that soon.

Let's look at the code:

```
from sklearn.base import TransformerMixin
from sklearn.utils import as_float_array
```

Now, create a new class that subclasses from our `mixin`:

```
class MeanDiscrete(TransformerMixin):
```

We need to define both a fit and transform function to conform to the API. In our `fit` function, we find the mean of the dataset and set an internal variable to remember that value. Let's look at the code:

```
    def fit(self, X):
```

First, we ensure that `X` is a dataset that we can work with, using the `as_float_array` function (which will also convert `X` if it can, for example, if `X` is a list of floats):

```
        X = as_float_array(X)
```

Next, we compute the mean of the array and set an internal parameter to remember this value. When `X` is a multivariate array, `self.mean` will be an array that contains the mean of each feature:

```
        self.mean = X.mean(axis=0)
```

The `fit` function also needs to return the class itself. This requirement ensures that we can perform chaining of functionality in transformers (such as calling `transformer.fit(X).transform(X)`). Let's look at the code:

```
        return self
```

Next, we define the transform function, this takes a dataset of the same type as the fit function, so we need to check we got the right input:

```
    def transform(self, X):
        X = as_float_array(X)
```

We should perform another check here too. While we need the input to be a NumPy array (or an equivalent data structure), the shape needs to be consistent too. The number of features in this array needs to be the same as the number of features the class was trained on.

```
        assert X.shape[1] == self.mean.shape[0]
```

Now, we perform the actual transformation by simply testing if the values in `X` are higher than the stored mean.

```
        return X > self.mean
```

We can then create an instance of this class and use it to transform our `X` array:

```
mean_discrete = MeanDiscrete()
X_mean = mean_discrete.fit_transform(X)
```

Unit testing

When creating your own functions and classes, it is always a good idea to do unit testing. Unit testing aims to test a single unit of your code. In this case, we want to test that our transformer does as it needs to do.

Good tests should be independently verifiable. A good way to confirm the legitimacy of your tests is by using another computer language or method to perform the calculations. In this case, I used Excel to create a dataset, and then computed the mean for each cell. Those values were then transferred here.

Unit tests should also be small and quick to run. Therefore, any data used should be of a small size. The dataset I used for creating the tests is stored in the `Xt` variable from earlier, which we will recreate in our test. The mean of these two features is 13.5 and 15.5, respectively.

To create our unit test, we import the `assert_array_equal` function from NumPy's testing, which checks whether two arrays are equal:

```
from numpy.testing import assert_array_equal
```

Next, we create our function. It is important that the test's name starts with `test_`, as this nomenclature is used for tools that automatically find and run tests. We also set up our testing data:

```
def test_meandiscrete():
    X_test = np.array([[ 0,  2],
                       [ 3,  5],
                       [ 6,  8],
                       [ 9, 11],
                       [12, 14],
                       [15, 17],
                       [18, 20],
                       [21, 23],
                       [24, 26],
                       [27, 29]])
```

We then create our transformer instance and fit it using this test data:

```
    mean_discrete = MeanDiscrete()
    mean_discrete.fit(X_test)
```

Next, we check whether the internal mean parameter was correctly set by comparing it with our independently verified result:

```
    assert_array_equal(mean_discrete.mean, np.array([13.5, 15.5]))
```

We then run the transform to create the transformed dataset. We also create an (independently computed) array with the expected values for the output:

```
    X_transformed = mean_discrete.transform(X_test)
    X_expected = np.array([[ 0,  0],
                           [ 0, 0],
                           [ 0, 0],
                           [ 0, 0],
                           [ 0, 0],
                           [ 1, 1],
                           [ 1, 1],
                           [ 1, 1],
                           [ 1, 1],
                           [ 1, 1]])
```

Finally, we test that our returned result is indeed what we expected:

```
    assert_array_equal(X_transformed, X_expected)
```

We can run the test by simply running the function itself:

```
test_meandiscrete()
```

If there was no error, then the test ran without an issue! You can verify this by changing some of the tests to deliberately incorrect values, and seeing that the test fails. Remember to change them back so that the test passes.

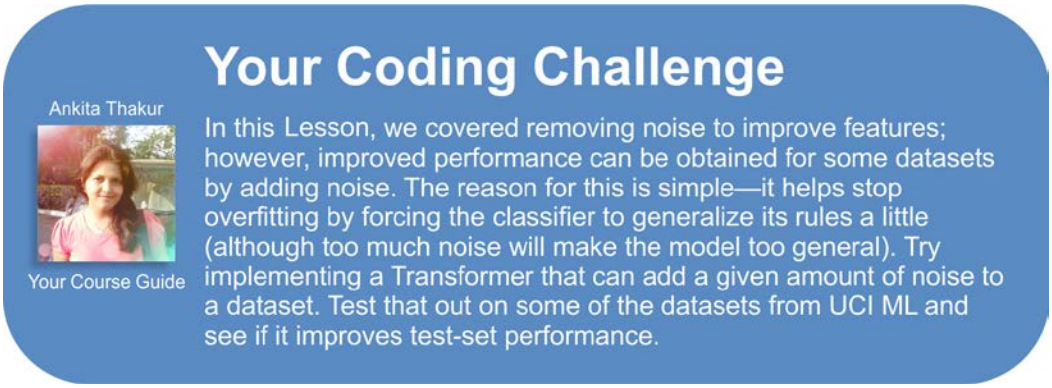
If we had multiple tests, it would be worth using a testing framework called `nose` to run our tests.

Putting it all together

Now that we have a tested transformer, it is time to put it into action. Using what we have learned so far, we create a `Pipeline`, set the first step to the `MeanDiscrete` transformer, and the second step to a Decision Tree Classifier. We then run a cross validation and print out the result. Let's look at the code:

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([('mean_discrete', MeanDiscrete()),
  ('classifier', DecisionTreeClassifier(random_state=14))])
  scores_mean_discrete = cross_val_score(pipeline, X, y,
    scoring='accuracy')
  print("Mean Discrete performance:
{0:.3f}".format(scores_mean_discrete.mean()))
```

The result is 0.803, which is not as good as before, but not bad for simple binary features.



Summary of Module 3 Chapter 5

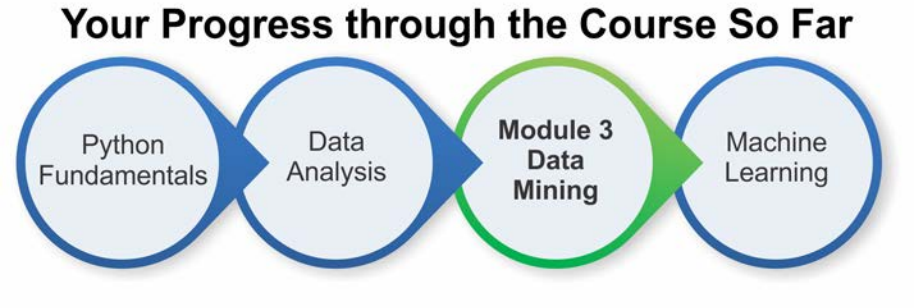
Ankita Thakur

Your Course Guide

In this chapter, we looked at features and transformers and how they can be used in the data mining pipeline. We discussed what makes a good feature and how to algorithmically choose good features from a standard set. However, creating good features is more art than science and often requires domain knowledge and experience.

We then created our own transformer using an interface that allows us to use it in scikit-learn's helper functions. We will be creating more transformers in later chapters so that we can perform effective testing using existing functions.

In the next chapter, we use feature extraction on a corpus of text documents. There are many transformers and feature types for text, each with their advantages and disadvantages.



6
Social Media Insight Using Naive Bayes

Text-based datasets contain a lot of information, whether they are books, historical documents, social media, e-mail, or any of the other ways we communicate via writing. Extracting features from text-based datasets and using them for classification is a difficult problem. There are, however, some common patterns for text mining.

We look at disambiguating terms in social media using the Naive Bayes algorithm, which is a powerful and surprisingly simple algorithm. Naive Bayes takes a few shortcuts to properly compute the probabilities for classification, hence the term *naive* in the name. It can also be extended to other types of datasets quite easily and doesn't rely on numerical features. The model in this chapter is a baseline for text mining studies, as the process can work reasonably well for a variety of datasets.

We will cover the following topics in this chapter:

- Downloading data from social network APIs
- Transformers for text
- Naive Bayes classifier
- Using JSON for saving and loading datasets
- The NLTK library for extracting features from text
- The F-measure for evaluation

Disambiguation

Text is often called an **unstructured** format. There is a lot of information there, but it is just there; no headings, no required format, loose syntax and other problems prohibit the easy extraction of information from text. The data is also highly connected, with lots of mentions and cross-references—just not in a format that allows us to easily extract it!

We can compare the information stored in a book with that stored in a large database to see the difference. In the book, there are characters, themes, places, and lots of information. However, the book needs to be read and, more importantly, interpreted to gain this information. The database sits on your server with column names and data types. All the information is there and the level of interpretation needed is quite low. Information about the data, such as its type or meaning is called **metadata**, and text lacks it. A book also contains some metadata in the form of a table of contents and index but the degree is significantly lower than that of a database.

One of the problems is the term **disambiguation**. When a person uses the word bank, is this a financial message or an environmental message (such as river bank)? This type of disambiguation is quite easy in many circumstances for humans (although there are still troubles), but much harder for computers to do.

In this chapter, we will look at disambiguating the use of the term Python on Twitter's stream. A message on Twitter is called a **tweet** and is limited to 140 characters. This means there is little room for context. There isn't much metadata available although hashtags are often used to denote the topic of the tweet.

When people talk about Python, they could be talking about the following things:

- The programming language Python
- Monty Python, the classic comedy group
- The snake Python
- A make of shoe called Python

There can be many other things called Python. The aim of our experiment is to take a tweet mentioning Python and determine whether it is talking about the programming language, based only on the content of the tweet.

Downloading data from a social network

We are going to download a corpus of data from Twitter and use it to sort out spam from useful content. Twitter provides a robust API for collecting information from its servers and this API is free for small-scale usage. It is, however, subject to some conditions that you'll need to be aware of if you start using Twitter's data in a commercial setting.

First, you'll need to sign up for a Twitter account (which is free). Go to `http://twitter.com` and register an account if you do not already have one.

Next, you'll need to ensure that you only make a certain number of requests per minute. This limit is currently 180 requests per hour. It can be tricky ensuring that you don't breach this limit, so it is highly recommended that you use a library to talk to Twitter's API.

You will need a key to access Twitter's data. Go to `http://twitter.com` and sign in to your account.

When you are logged in, go to `https://apps.twitter.com/` and click on **Create New App**.

Create a name and description for your app, along with a website address. If you don't have a website to use, insert a placeholder. Leave the **Callback URL** field blank for this app—we won't need it. Agree to the terms of use (if you do) and click on **Create your Twitter application**.

Keep the resulting website open—you'll need the **access keys** that are on this page. Next, we need a library to talk to Twitter. There are many options; the one I like is simply called `twitter`, and is the *official* Twitter Python library.

You can install `twitter` using `pip3 install twitter` if you are using `pip` to install your packages. If you are using another system, check the documentation at `https://github.com/sixohsix/twitter`.

Create a new IPython Notebook to download the data. We will create several notebooks in this chapter for various different purposes, so it might be a good idea to also create a folder to keep track of them. This first notebook, `ch6_get_twitter`, is specifically for downloading new Twitter data.

First, we import the `twitter` library and set our authorization tokens. The **consumer key**, **consumer secret** will be available on the **Keys and Access Tokens** tab on your Twitter app's page. To get the access tokens, you'll need to click on the **Create my access token** button, which is on the same page. Enter the keys into the appropriate places in the following code:

```
import twitter
consumer_key = "<Your Consumer Key Here>"
consumer_secret = "<Your Consumer Secret Here>"
access_token = "<Your Access Token Here>"
access_token_secret = "<Your Access Token Secret Here>"
authorization = twitter.OAuth(access_token, access_token_secret,
consumer_key, consumer_secret)
```

We are going to get our tweets from Twitter's `search` function. We will create a reader that connects to `twitter` using our authorization, and then use that reader to perform searches. In the Notebook, we set the filename where the tweets will be stored:

```
import os
output_filename = os.path.join(os.path.expanduser("~"),
  "Data", "twitter", "python_tweets.json")
```

We also need the `json` library for saving our tweets:

```
import json
```

Next, create an object that can read from Twitter. We create this object with our authorization object that we set up earlier:

```
t = twitter.Twitter(auth=authorization)
```

We then open our output file for writing. We open it for appending—this allows us to rerun the script to obtain more tweets. We then use our Twitter connection to perform a search for the word Python. We only want the statuses that are returned for our dataset. This code takes the tweet, uses the `json` library to create a string representation using the `dumps` function, and then writes it to the file. It then creates a blank line under the tweet so that we can easily distinguish where one tweet starts and ends in our file:

```
with open(output_filename, 'a') as output_file:
    search_results = t.search.tweets(q="python", count=100)['statuses']
    for tweet in search_results:
        if 'text' in tweet:
            output_file.write(json.dumps(tweet))
            output_file.write("\n\n")
```

In the preceding loop, we also perform a check to see whether there is text in the tweet or not. Not all of the objects returned by twitter will be actual tweets (some will be actions to delete tweets and others). The key difference is the inclusion of text as a key, which we test for.

Running this for a few minutes will result in 100 tweets being added to the output file.

You can keep rerunning this script to add more tweets to your dataset, keeping in mind that you may get some duplicates in the output file if you rerun it too fast (that is, before Twitter gets new tweets to return!).

Loading and classifying the dataset

After we have collected a set of tweets (our dataset), we need labels to perform classification. We are going to label the dataset by setting up a form in an IPython Notebook to allow us to enter the labels.

The dataset we have stored is *nearly* in a **JSON** format. JSON is a format for data that doesn't impose much structure and is directly readable in JavaScript (hence the name, JavaScript Object Notation). JSON defines basic objects such as numbers, strings, lists and dictionaries, making it a good format for storing datasets if they contain data that isn't numerical. If your dataset is fully numerical, you would save space and time using a matrix-based format like in NumPy.

A key difference between our dataset and *real* JSON is that we included new lines between tweets. The reason for this was to allow us to easily append new tweets (the actual JSON format doesn't allow this easily). Our format is a JSON representation of a tweet, followed by a newline, followed by the next tweet, and so on.

To parse it, we can use the `json` library but we will have to first split the file by newlines to get the actual tweet objects themselves.

Set up a new IPython Notebook (I called mine `ch6_label_twitter`) and enter the dataset's filename. This is the same filename in which we saved the data in the previous section. We also define the filename that we will use to save the labels to. The code is as follows:

```
import os
input_filename = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "python_classes.json")
```

As stated, we will use the `json` library, so import that too:

```
import json
```

We create a list that will store the tweets we received from the file:

```
tweets = []
```

We then iterate over each line in the file. We aren't interested in lines with no information (they separate the tweets for us), so check if the length of the line (minus any whitespace characters) is zero. If it is, ignore it and move to the next line. Otherwise, load the tweet using `json.loads` (which loads a JSON object from a string) and add it to our list of tweets. The code is as follows:

```
with open(input_filename) as inf:
    for line in inf:
        if len(line.strip()) == 0:
            continue
        tweets.append(json.loads(line))
```

We are now interested in classifying whether an item is relevant to us or not (in this case, *relevant* means *refers to the programming language Python*). We will use the IPython Notebook's ability to embed HTML and talk between JavaScript and Python to create a viewer of tweets to allow us to easily and quickly classify the tweets as spam or not.

The code will present a new tweet to the user (you) and ask for a label: is it relevant or not? It will then store the input and present the next tweet to be labeled.

First, we create a list for storing the labels. These labels will be stored whether or not the given tweet refers to the programming language Python, and it will allow our classifier to learn how to differentiate between meanings.

We also check if we have any labels already and load them. This helps if you need to close the notebook down midway through labeling. This code will load the labels from where you left off. It is generally a good idea to consider how to save at midpoints for tasks like this. Nothing hurts quite like losing an hour of work because your computer crashed before you saved the labels! The code is as follows:

```
labels = []
if os.path.exists(labels_filename):
    with open(labels_filename) as inf:
        labels = json.load(inf)
```

Next, we create a simple function that will return the next tweet that needs to be labeled. We can work out which is the next tweet by finding the first one that hasn't yet been labeled. The code is as follows:

```
def get_next_tweet():
    return tweet_sample[len(labels)]['text']
```

The next step in our experiment is to collect information from the user (you!) on which tweets are referring to Python (the programming language) and which are not. As of yet, there is not a good, straightforward way to get interactive feedback with pure Python in IPython Notebooks. For this reason, we will use some JavaScript and HTML to get this input from the user.

Next we create some JavaScript in the IPython Notebook to run our input. Notebooks allow us to use magic functions to embed HTML and JavaScript (among other things) directly into the Notebook itself. Start a new cell with the following line at the top:

```
%%javascript
```

The code in here will be in JavaScript, hence the curly braces that are coming up. Don't worry, we will get back to Python soon. Keep in mind here that the following code must be in the same cell as the `%%javascript` magic function.

The first function we will define in JavaScript shows how easy it is to talk to your Python code from JavaScript in IPython Notebooks. This function, if called, will add a label to the `labels` array (which is in `python` code). To do this, we load the IPython **kernel** as a JavaScript object and give it a Python command to execute. The code is as follows:

```
function set_label(label){
    var kernel = IPython.notebook.kernel;
    kernel.execute("labels.append(" + label + ")");
    load_next_tweet();
}
```

At the end of that function, we call the `load_next_tweet` function. This function loads the next tweet to be labeled. It runs on the same principle; we load the IPython kernel and give it a command to execute (calling the `get_next_tweet` function we defined earlier).

However, in this case we want to get the result. This is a little more difficult. We need to define a `callback`, which is a function that is called when the data is returned. The format for defining `callback` is outside the scope of this module. If you are interested in more advanced JavaScript/Python integration, consult the IPython documentation.

The code is as follows:

```
function load_next_tweet(){
   var code_input = "get_next_tweet()";
   var kernel = IPython.notebook.kernel;
   var callbacks = { 'iopub' : {'output' : handle_output}};
   kernel.execute(code_input, callbacks, {silent:false});
}
```

The callback function is called `handle_output`, which we will define now. This function gets called when the Python function that `kernel.execute` calls returns a value. As before, the full format of this is outside the scope of this module. However, for our purposes the result is returned as data of the type text/plain, which we extract and show in the `#tweet_text div` of the form we are going to create in the next cell. The code is as follows:

```
function handle_output(out){
   var res = out.content.data["text/plain"];
   $("div#tweet_text").html(res);
}
```

Our form will have a `div` that shows the next tweet to be labeled, which we will give the `ID #tweet_text`. We also create a textbox to enable us to capture key presses (otherwise, the Notebook will capture them and JavaScript won't do anything). This allows us to use the keyboard to set labels of *1* or *0*, which is faster than using the mouse to click buttons—given that we will need to label at least 100 tweets.

Run the previous cell to embed some JavaScript into the page, although nothing will be shown to you in the results section.

We are going to use a different magic function now, `%%html`. Unsurprisingly, this magic function allows us to directly embed HTML into our Notebook. In a new cell, start with this line:

```
%%html
```

For this cell, we will be coding in HTML and a little JavaScript. First, define a `div` element to store our current tweet to be labeled. I've also added some instructions for using this form. Then, create the `#tweet_text` div that will store the text of the next tweet to be labeled. As stated before, we need to create a textbox to be able to capture key presses. The code is as follows:

```
<div name="tweetbox">
    Instructions: Click in textbox. Enter a 1 if the tweet is
relevant, enter 0 otherwise.<br>
Tweet: <div id="tweet_text" value="text"></div><br>
<input type=text id="capture"></input><br>
</div>
```

Don't run the cell just yet!

We create the JavaScript for capturing the key presses. This has to be defined after creating the form, as the `#tweet_text` div doesn't exist until the above code runs. We use the **JQuery** library (which IPython is already using, so we don't need to include the JavaScript file) to add a function that is called when key presses are made on the `#capture` textbox we defined. However, keep in mind that this is a `%%html` cell and not a JavaScript cell, so we need to enclose this JavaScript in the `<script>` tags.

We are only interested in key presses if the user presses the **0** or the **1**, in which case the relevant label is added. We can determine which key was pressed by the ASCII value stored in `e.which`. If the user presses 0 or 1, we append the label and clear out the textbox. The code is as follows:

```
<script>
$("input#capture").keypress(function(e) {
if(e.which == 48) {
    set_label(0);
    $("input#capture").val("");
}else if (e.which == 49){
    set_label(1);
    $("input#capture").val("");
  }
});
```

All other key presses are ignored.

As a last bit of JavaScript for this chapter (I promise), we call the `load_next_tweet()` function. This will set the first tweet to be labeled and then close off the JavaScript. The code is as follows:

```
load_next_tweet();
</script>
```

After you run this cell, you will get an HTML textbox, alongside the first tweet's text. Click in the textbox and enter `1` if it is relevant to our goal (in this case, it means *is the tweet related to the programming language Python*) and a 0 if it is not. After you do this, the next tweet will load. Enter the label and the next one will load. This continues until the tweets run out.

When you finish all of this, simply save the labels to the output filename we defined earlier for the class values:

```
with open(labels_filename, 'w') as outf:
    json.dump(labels, outf)
```

You can call the preceding code even if you haven't finished. Any labeling you have done to that point will be saved. Running this Notebook again will pick up where you left off and you can keep labeling your tweets.

This might take a while to do this! If you have a lot of tweets in your dataset, you'll need to classify all of them. If you are pushed for time, you can download the same dataset I used, which contains classifications.

Creating a replicable dataset from Twitter

In data mining, there are lots of variables. These aren't just in the data mining algorithms—they also appear in the data collection, environment, and many other factors. Being able to replicate your results is important as it enables you to verify or improve upon your results.

Getting 80 percent accuracy on one dataset with algorithm `X`, and 90 percent accuracy on another dataset with algorithm `Y` doesn't mean that Y is better. We need to be able to test on the same dataset in the same conditions to be able to properly compare.

On running the preceding code, you will get a different dataset to the one I created and used. The main reasons are that Twitter will return different search results for you than me based on the time you performed the search. Even after that, your labeling of tweets might be different from what I do. While there are obvious examples where *a given tweet relates to the python programming language*, there will always be gray areas where the labeling isn't obvious. One tough gray area I ran into was tweets in non-English languages that I couldn't read. In this specific instance, there are options in Twitter's API for setting the language, but even these aren't going to be perfect.

Due to these factors, it is difficult to replicate experiments on databases that are extracted from social media, and Twitter is no exception. Twitter explicitly disallows sharing datasets directly.

One solution to this is to share tweet IDs only, which you can share freely. In this section, we will first create a tweet ID dataset that we can freely share. Then, we will see how to download the original tweets from this file to recreate the original dataset.

First, we save the replicable dataset of tweet IDs. Creating another new IPython Notebook, first set up the filenames. This is done in the same way we did labeling but there is a new filename where we can store the replicable dataset. The code is as follows:

```
import os
input_filename = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "python_classes.json")
replicable_dataset = os.path.join(os.path.expanduser("~"),
  "Data", "twitter", "replicable_dataset.json")
```

We load the tweets and labels as we did in the previous notebook:

```
import json
tweets = []
with open(input_filename) as inf:
    for line in inf:
        if len(line.strip()) == 0:
            continue
        tweets.append(json.loads(line))
if os.path.exists(labels_filename):
    with open(classes_filename) as inf:
        labels = json.load(inf)
```

Now we create a dataset by looping over both the tweets and labels at the same time and saving those in a list:

```
dataset = [(tweet['id'], label) for tweet, label in zip(tweets,
  labels)]
```

Finally, we save the results in our file:

```
with open(replicable_dataset, 'w') as outf:
    json.dump(dataset, outf)
```

Now that we have the tweet IDs and labels saved, we can recreate the original dataset. If you are looking to recreate the dataset I used for this chapter, it can be found in the code bundle that comes with this course.

Loading the preceding dataset is not difficult but it can take some time. Start a new IPython Notebook and set the dataset, label, and tweet ID filenames as before. I've adjusted the filenames here to ensure that you don't overwrite your previously collected dataset, but feel free to change these if you want. The code is as follows:

```
import os
tweet_filename = os.path.join(os.path.expanduser("~"), "Data",
  "twitter", "replicable_python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data",
  "twitter", "replicable_python_classes.json")
replicable_dataset = os.path.join(os.path.expanduser("~"),
  "Data", "twitter", "replicable_dataset.json")
```

Then load the tweet IDs from the file using JSON:

```
import json
with open(replicable_dataset) as inf:
    tweet_ids = json.load(inf)
```

Saving the labels is very easy. We just iterate through this dataset and extract the IDs. We could do this quite easily with just two lines of code (open file and save tweets). However, we can't guarantee that we will get all the tweets we are after (for example, some may have been changed to private since collecting the dataset) and therefore the labels will be incorrectly indexed against the data.

As an example, I tried to recreate the dataset just one day after collecting them and already two of the tweets were missing (they might be deleted or made private by the user). For this reason, it is important to only print out the labels that we need. To do this, we first create an empty `actual labels` list to store the labels for tweets that we actually recover from `twitter`, and then create a dictionary mapping the tweet IDs to the labels.

The code is as follows:

```
actual_labels = []
label_mapping = dict(tweet_ids)
```

Next, we are going to create a `twitter` server to collect all of these tweets. This is going to take a little longer. Import the `twitter` library that we used before, creating an authorization token and using that to create the `twitter` object:

```
import twitter
consumer_key = "<Your Consumer Key Here>"
consumer_secret = "<Your Consumer Secret Here>"
access_token = "<Your Access Token Here>"
access_token_secret = "<Your Access Token Secret Here>"
authorization = twitter.OAuth(access_token, access_token_secret,
  consumer_key, consumer_secret)
t = twitter.Twitter(auth=authorization)
```

Iterate over each of the twitter IDs by extracting the IDs into a list using the following command:

```
all_ids = [tweet_id for tweet_id, label in tweet_ids]
```

Then, we open our output file to save the tweets:

```
with open(tweets_filename, 'a') as output_file:
```

The Twitter API allows us get 100 tweets at a time. Therefore, we iterate over each batch of 100 tweets:

```
    for start_index in range(0, len(tweet_ids), 100):
```

To search by ID, we first create a string that joins all of the IDs (in this batch) together:

```
        id_string = ",".join(str(i) for i in
          all_ids[start_index:start_index+100])
```

Next, we perform a **statuses/lookup** API call, which is defined by Twitter. We pass our list of IDs (which we turned into a string) into the API call in order to have those tweets returned to us:

```
        search_results = t.statuses.lookup(_id=id_string)
```

Then for each tweet in the search results, we save it to our file in the same way we did when we were collecting the dataset originally:

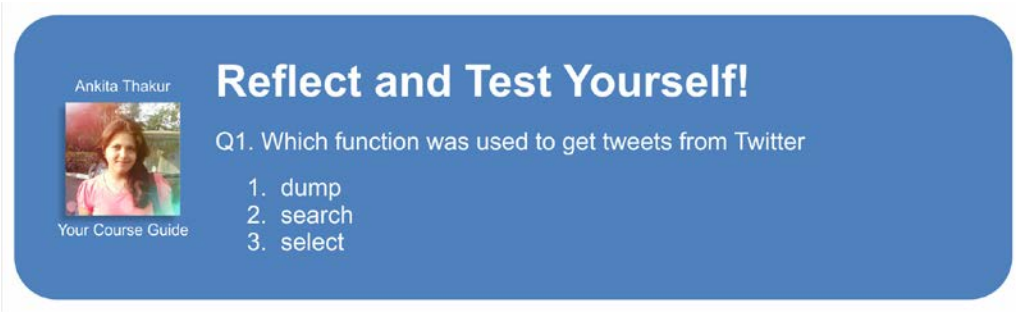
```
for tweet in search_results:
    if 'text' in tweet:
        output_file.write(json.dumps(tweet))
        output_file.write("\n\n")
```

As a final step here (and still under the preceding `if` block), we want to store the labeling of this tweet. We can do this using the `label_mapping` dictionary we created before, looking up the tweet ID. The code is as follows:

```
        actual_labels.append(label_mapping[tweet['id']])
```

Run the previous cell and the code will collect all of the tweets for you. If you created a really big dataset, this may take a while—Twitter does rate-limit requests. As a final step here, save the `actual_labels` to our `classes` file:

```
with open(labels_filename, 'w') as outf:
    json.dump(actual_labels, outf)
```



Text transformers

Now that we have our dataset, how are we going to perform data mining on it?

Text-based datasets include books, essays, websites, manuscripts, programming code, and other forms of written expression. All of the algorithms we have seen so far deal with numerical or categorical features, so how do we convert our text into a format that the algorithm can deal with?

There are a number of measurements that could be taken. For instance, average word and average sentence length are used to predict the readability of a document. However, there are lots of feature types such as word occurrence which we will now investigate.

Bag-of-words

One of the simplest but highly effective models is to simply count each word in the dataset. We create a matrix, where each row represents a document in our dataset and each column represents a word. The value of the cell is the frequency of that word in the document.

Here's an excerpt from *The Lord of the Rings*, *J.R.R. Tolkien*:

> *Three Rings for the Elven-kings under the sky,*
>
> *Seven for the Dwarf-lords in halls of stone,*
>
> *Nine for Mortal Men, doomed to die,*
>
> *One for the Dark Lord on his dark throne*
>
> *In the Land of Mordor where the Shadows lie.*
>
> *One Ring to rule them all, One Ring to find them,*
>
> *One Ring to bring them all and in the darkness bind them.*
>
> *In the Land of Mordor where the Shadows lie.*
>
> *- J.R.R. Tolkien's epigraph to The Lord of The Rings*

The word *the* appears nine times in this quote, while the words *in*, *for*, *to*, and *one* each appear four times. The word *ring* appears three times, as does the word *of*.

We can create a dataset from this, choosing a subset of words and counting the frequency:

Word	the	one	ring	to
Frequency	9	4	3	4

We can use the `counter` class to do a simple count for a given string. When counting words, it is normal to convert all letters to lowercase, which we do when creating the string. The code is as follows:

```
s = """Three Rings for the Elven-kings under the sky,
Seven for the Dwarf-lords in halls of stone,
Nine for Mortal Men, doomed to die,
One for the Dark Lord on his dark throne
In the Land of Mordor where the Shadows lie.
One Ring to rule them all, One Ring to find them,
```

```
One Ring to bring them all and in the darkness bind them.
In the Land of Mordor where the Shadows lie. """.lower()
words = s.split()
from collections import Counter
c = Counter(words)
```

Printing `c.most_common(5)` gives the list of the top five most frequently occurring words. Ties are not handled well as only five are given and a very large number of words all share a tie for fifth place.

The bag-of-words model has three major types. The first is to use the raw frequencies, as shown in the preceding example. This does have a drawback when documents vary in size from fewer words to many words, as the overall values will be very different. The second model is to use the normalized frequency, where each document's sum equals 1. This is a much better solution as the length of the document doesn't matter as much. The third type is to simply use binary features—a value is 1 if the word occurs *at all* and 0 if it doesn't. We will use binary representation in this chapter.

Another popular (arguably more popular) method for performing normalization is called **term frequency - inverse document frequency**, or **tf-idf**. In this weighting scheme, term counts are first normalized to frequencies and then divided by the number of documents in which it appears in the corpus. We will use tf-idf in *Chapter 10*, *Clustering News Articles*.

There are a number of libraries for working with text data in Python. We will use a major one, called **Natural Language ToolKit** (**NLTK**). The `scikit-learn` library also has the `CountVectorizer` class that performs a similar action, and it is recommended you take a look at it (we will use it in *Chapter 9*, *Authorship Attribution*). However the NLTK version has more options for word tokenization. If you are doing natural language processing in python, NLTK is a great library to use.

N-grams

A step up from single bag-of-words features is that of **n-grams**. An n-gram is a subsequence of *n* consecutive tokens. In this context, a word n-gram is a set of *n* words that appear in a row.

They are counted the same way, with the n-grams forming a *word* that is put in the *bag*. The value of a cell in this dataset is the frequency that a particular n-gram appears in the given document.

The value of *n* is a parameter. For English, setting it to between 2 to 5 is a good start, although some applications call for higher values.

As an example, for *n=3*, we extract the first few n-grams in the following quote:

Always look on the bright side of life.

The first n-gram (of size 3) is *Always look on*, the second is *look on the*, the third is *on the bright*. As you can see, the n-grams overlap and cover three words.

Word n-grams have advantages over using single words. This simple concept introduces some context to word use by considering its local environment, without a large overhead of understanding the language computationally. A disadvantage of using n-grams is that the matrix becomes even sparser—word n-grams are unlikely to appear twice (especially in tweets and other short documents!).

Specially for social media and other short documents, word n-grams are unlikely to appear in too many different tweets, unless it is a retweet. However, in larger documents, word n-grams are quite effective for many applications.

Another form of n-gram for text documents is that of a character n-gram. Rather than using sets of words, we simply use sets of characters (although character n-grams have lots of options for how they are computed!). This type of dataset can pick up words that are misspelled, as well as providing other benefits. We will test character n-grams in this chapter and see them again in *Chapter 9*, *Authorship Attribution*.

Other features

There are other features that can be extracted too. These include syntactic features, such as the usage of particular words in sentences. Part-of-speech tags are also popular for data mining applications that need to understand meaning in text. Such feature types won't be covered in this module. If you are interested in learning more, I recommend *Python 3 Text Processing with NLTK 3 Cookbook*, *Jacob Perkins*, *Packt Publishing*.

Naive Bayes

Naive Bayes is a probabilistic model that is unsurprisingly built upon a naive interpretation of Bayesian statistics. Despite the naive aspect, the method performs very well in a large number of contexts. It can be used for classification of many different feature types and formats, but we will focus on one in this chapter: binary features in the bag-of-words model.

Bayes' theorem

For most of us, when we were taught statistics, we started from a frequentist approach. In this approach, we assume the data comes from some distribution and we aim to determine what the parameters are for that distribution. However, those parameters are (perhaps incorrectly) assumed to be fixed. We use our model to describe the data, even testing to ensure the data fits our model.

Bayesian statistics instead model how people (non-statisticians) actually reason. We have some data and we use that data to update our model about how likely something is to occur. In Bayesian statistics, we use the data to describe the model rather than using a model and confirming it with data (as per the frequentist approach).

Bayes' theorem computes the value of *P(A | B)*, that is, knowing that *B* has occurred, what is the probability of *A*. In most cases, *B* is an observed event such as *it rained yesterday*, and *A* is a prediction *it will rain today*. For data mining, *B* is usually *we observed this sample* and *A* is *it belongs to this class*. We will see how to use Bayes' theorem for data mining in the next section.

The equation for Bayes' theorem is given as follows:

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}.$$

As an example, we want to determine the probability that an e-mail containing the word *drugs* is spam (as we believe that such a tweet may be a pharmaceutical spam).

A, in this context, is the probability that *this tweet is spam*. We can compute *P(A)*, called the *prior belief* directly from a training dataset by computing the percentage of tweets in our dataset that are spam. If our dataset contains 30 spam messages for every 100 e-mails, *P(A)* is *30/100* or *0.3*.

B, in this context, is *this tweet contains the word 'drugs'*. Likewise, we can compute *P(B)* by computing the percentage of tweets in our dataset containing the word *drugs*. If 10 e-mails in every 100 of our training dataset contain the word *drugs*, *P(B)* is *10/100* or 0.1. Note that we don't care if the e-mail is spam or not when computing this value.

`P(B|A)` is the probability that an e-mail contains the word *drugs* if it is spam. It is also easy to compute from our training dataset. We look through our training set for spam e-mails and compute the percentage of them that contain the word *drugs*. Of our 30 spam e-mails, if 6 contain the word *drugs*, then *P(B | A)* is calculated as 6/30 or 0.2.

From here, we use Bayes' theorem to compute *P(A | B)*, which is the probability that a tweet containing the word *drugs* is spam. Using the previous equation, we see the result is 0.6. This indicates that if an e-mail has the word *drugs* in it, there is a 60 percent chance that it is spam.

Note the empirical nature of the preceding example—we use evidence directly from our training dataset, not from some preconceived distribution. In contrast, a frequentist view of this would rely on us creating a distribution of the probability of words in tweets to compute similar equations.

Naive Bayes algorithm

Looking back at our Bayes' theorem equation, we can use it to compute the probability that a given sample belongs to a given class. This allows the equation to be used as a classification algorithm.

With *C* as a given class and *D* as a sample in our dataset, we create the elements necessary for Bayes' theorem, and subsequently Naive Bayes. Naive Bayes is a classification algorithm that utilizes Bayes' theorem to compute the probability that a new data sample belongs to a particular class.

P(C) is the probability of a class, which is computed from the training dataset itself (as we did with the spam example). We simply compute the percentage of samples in our training dataset that belong to the given class.

P(D) is the probability of a given data sample. It can be difficult to compute this, as the sample is a complex interaction between different features, but luckily it is a constant across all classes. Therefore, we don't need to compute it at all. We will see later how to get around this issue.

P(D | C) is the probability of the data point belonging to the class. This could also be difficult to compute due to the different features. However, this is where we introduce the *naive* part of the Naive Bayes algorithm. We naively assume that each feature is independent of each other. Rather than computing the full probability of *P(D | C)*, we compute the probability of each feature *D1, D2, D3*, … and so on. Then, we multiply them together:

```
P(D|C) = P(D1|C) x P(D2|C).... x P(Dn|C)
```

Each of these values is relatively easy to compute with binary features; we simply compute the percentage of times it is equal in our sample dataset.

In contrast, if we were to perform a non-naive Bayes version of this part, we would need to compute the correlations between different features for each class. Such computation is infeasible at best, and nearly impossible without vast amounts of data or adequate language analysis models.

From here, the algorithm is straightforward. We compute *P(C|D)* for each possible class, ignoring the *P(D)* term. Then we choose the class with the highest probability. As the *P(D)* term is consistent across each of the classes, ignoring it has no impact on the final prediction.

How it works

As an example, suppose we have the following (binary) feature values from a sample in our dataset: `[0, 0, 0, 1]`.

Our training dataset contains two classes with 75 percent of samples belonging to the class `0`, and 25 percent belonging to the class `1`. The likelihood of the feature values for each class are as follows:

For class 0: `[0.3, 0.4, 0.4, 0.7]`

For class 1: `[0.7, 0.3, 0.4, 0.9]`

These values are to be interpreted as: *for feature 1, it is a 1 in 30 percent of cases for class 0*.

We can now compute the probability that this sample should belong to the class `0`. *P(C=0) = 0.75* which is the probability that the class is `0`.

P(D) isn't needed for the Naive Bayes algorithm. Let's take a look at the calculation:

```
P(D|C=0) = P(D1|C=0) x P(D2|C=0) x P(D3|C=0) x P(D4|C=0)
= 0.3 x 0.6 x 0.6 x 0.7
= 0.0756
```

The second and third values are 0.6, because the value of that feature in the sample was `0`. The listed probabilities are for values of `1` for each feature. Therefore, the probability of a `0` is its inverse: *P(0) = 1 – P(1)*.

Now, we can compute the probability of the data point belonging to this class. An important point to note is that we haven't computed *P(D)*, so this isn't a real probability. However, it is good enough to compare against the same value for the probability of the class 1. Let's take a look at the calculation:

```
P(C=0|D) = P(C=0) P(D|C=0)
= 0.75 * 0.0756
= 0.0567
```

Now, we compute the same values for the class 1:

```
P(C=1) = 0.25
```

`P(D)` isn't needed for naive Bayes. Let's take a look at the calculation:

```
P(D|C=1) = P(D1|C=1) x P(D2|C=1) x P(D3|C=1) x P(D4|C=1)
= 0.7 x 0.7 x 0.6 x 0.9
= 0.2646
P(C=1|D) = P(C=1)P(D|C=1)
= 0.25 * 0.2646
= 0.06615
```

Normally, *P(C=0|D) + P(C=1|D)* should equal to 1. After all, those are the only two possible options! However, the probabilities are not 1 due to the fact we haven't included the computation of *P(D)* in our equations here.

The data point should be classified as belonging to the class `1`. You may have guessed this while going through the equations anyway; however, you may have been a bit surprised that the final decision was so close. After all, the probabilities in computing *P(D|C)* were much, much higher for the class `1`. This is because we introduced a prior belief that most samples generally belong to the class `0`.

If the classes had been equal sizes, the resulting probabilities would be much different. Try it yourself by changing both *P(C=0)* and *P(C=1)* to 0.5 for equal class sizes and computing the result again.

Application

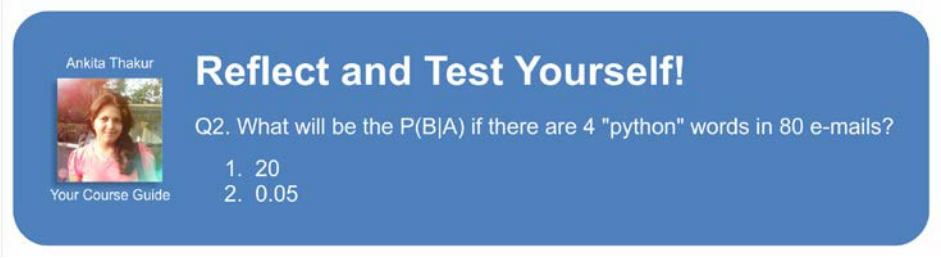
We will now create a pipeline that takes a tweet and determines whether it is relevant or not, based only on the content of that tweet.

To perform the word extraction, we will be using the NLTK, a library that contains a large number of tools for performing analysis on natural language. We will use NLTK in future chapters as well.

To get NLTK on your computer, use `pip` to install the package:
`pip3 install nltk`
If that doesn't work, see the NLTK installation instructions at `www.nltk.org/install.html`.

We are going to create a pipeline to extract the word features and classify the tweets using Naive Bayes. Our pipeline has the following steps:

1. Transform the original text documents into a dictionary of counts using NLTK's `word_tokenize` function.
2. Transform those dictionaries into a vector matrix using the `DictVectorizer` transformer in `scikit-learn`. This is necessary to enable the Naive Bayes classifier to read the feature values extracted in the first step.
3. Train the Naive Bayes classifier, as we have seen in previous chapters.
4. We will need to create another Notebook (last one for the chapter!) called `ch6_classify_twitter` for performing the classification.



Extracting word counts

We are going to use NLTK to extract our word counts. We still want to use it in a pipeline, but NLTK doesn't conform to our transformer interface. We will therefore need to create a basic transformer to do this to obtain both `fit` and `transform` methods, enabling us to use this in a pipeline.

First, set up the `transformer` class. We don't need to fit anything in this class, as this transformer simply extracts the words in the document. Therefore, our fit is an empty function, except that it returns `self` which is necessary for transformer objects.

Our transform is a little more complicated. We want to extract each word from each document and record `True` if it was discovered. We are only using the binary features here—`True` if in the document, `False` otherwise. If we wanted to use the frequency we would set up counting dictionaries, as we have done in several of the past chapters.

Let's take a look at the code:

```
from sklearn.base import TransformerMixin
class NLTKBOW(TransformerMixin):
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return [{word: True for word in word_tokenize(document)}
                for document in X]
```

The result is a list of dictionaries, where the first dictionary is the list of words in the first tweet, and so on. Each dictionary has a word as key and the value `true` to indicate this word was discovered. Any word not in the dictionary will be assumed to have not occurred in the tweet. Explicitly stating that a word's occurrence is `False` will also work, but will take up needless space to store.

Converting dictionaries to a matrix

This step converts the dictionaries built as per the previous step into a matrix that can be used with a classifier. This step is made quite simple through the `DictVectorizer` transformer.

The `DictVectorizer` class simply takes a list of dictionaries and converts them into a matrix. The features in this matrix are the keys in each of the dictionaries, and the values correspond to the occurrence of those features in each sample. Dictionaries are easy to create in code, but many data algorithm implementations prefer matrices. This makes `DictVectorizer` a very useful class.

In our dataset, each dictionary has words as keys and only occurs if the word actually occurs in the tweet. Therefore, our matrix will have each word as a feature and a value of `True` in the cell if the word occurred in the tweet.

To use `DictVectorizer`, simply import it using the following command:

```
from sklearn.feature_extraction import DictVectorizer
```

Training the Naive Bayes classifier

Finally, we need to set up a classifier and we are using Naive Bayes for this chapter. As our dataset contains only binary features, we use the `BernoulliNB` classifier that is designed for binary features. As a classifier, it is very easy to use. As with `DictVectorizer`, we simply import it and add it to our pipeline:

```
from sklearn.naive_bayes import BernoulliNB
```

Putting it all together

Now comes the moment to put all of these pieces together. In our IPython Notebook, set the filenames and load the dataset and classes as we have done before. Set the filenames for both the tweets themselves (not the IDs!) and the labels that we assigned to them. The code is as follows:

```
import os
input_filename = os.path.join(os.path.expanduser("~"), "Data",
  "twitter", "python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data",
  "twitter", "python_classes.json")
```

Load the tweets themselves. We are only interested in the content of the tweets, so we extract the `text` value and store only that. The code is as follows:

```
tweets = []
with open(input_filename) as inf:
    for line in inf:
        if len(line.strip()) == 0:
            continue
        tweets.append(json.loads(line)['text'])
```

Load the labels for each of the tweets:

```
with open(classes_filename) as inf:
    labels = json.load(inf)
```

Now, create a pipeline putting together the components from before. Our pipeline has three parts:

- The `NLTKBOW` transformer we created
- A `DictVectorizer` transformer
- A `BernoulliNB` classifier

The code is as follows:

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([('bag-of-words', NLTKBOW()),
                     ('vectorizer', DictVectorizer()),
                     ('naive-bayes', BernoulliNB())
                     ])
```

We can nearly run our pipeline now, which we will do with `cross_val_score` as we have done many times before. Before *that* though, we will introduce a better evaluation metric than the accuracy metric we used before. As we will see, the use of accuracy is not adequate for datasets when the number of samples in each class is different.

Evaluation using the F1-score

When choosing an evaluation metric, it is always important to consider cases where that evaluation metric is not useful. Accuracy is a good evaluation metric in many cases, as it is easy to understand and simple to compute. However, it can be easily faked. In other words, in many cases you can create algorithms that have a high accuracy by poor utility.

While our dataset of tweets (typically, your results may vary) contains about 50 percent programming-related and 50 percent nonprogramming, many datasets aren't as **balanced** as this.

As an example, an e-mail spam filter may expect to see more than 80 percent of incoming e-mails be spam. A spam *filter* that simply labels everything as spam is quite useless; however, it will obtain an accuracy of 80 percent!

To get around this problem, we can use other evaluation metrics. One of the most commonly employed is called an *f1-score* (also called f-score, f-measure, or one of many other variations on this term).

The f1-score is defined on a *per-class* basis and is based on two concepts: the *precision* and *recall*. The *precision* is the percentage of all the samples that were predicted as belonging to a specific class that were actually from that class. The *recall* is the percentage of samples in the dataset that are in a class and actually labeled as belonging to that class.

In the case of our application, we could compute the value for both classes (relevant and not relevant). However, we are really interested in the spam. Therefore, our precision computation becomes the question: *of all the tweets that were predicted as being relevant, what percentage were actually relevant?* Likewise, the recall becomes the question: *of all the relevant tweets in the dataset, how many were predicted as being relevant?*

After you compute both the precision and recall, the f1-score is the harmonic mean of the precision and recall:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

To use the f1-score in `scikit-learn` methods, simply set the scoring parameter to *f1*. By default, this will return the f1-score of the class with label 1. Running the code on our dataset, we simply use the following line of code:

```
scores = cross_val_score(pipeline, tweets, labels, scoring='f1')
```

We then print out the average of the scores:

```
import numpy as np
print("Score: {:.3f}".format(np.mean(scores)))
```

The result is 0.798, which means we can accurately determine if a tweet using Python relates to the programing language nearly 80 percent of the time. This is using a dataset with only 200 tweets in it. Go back and collect more data and you will find that the results increase!

More data usually means a better accuracy, but it is not guaranteed!

Getting useful features from models

One question you may ask is *what are the best features for determining if a tweet is relevant or not?* We can extract this information from of our Naive Bayes model and find out which features are the best individually, according to Naive Bayes.

First we fit a new model. While the `cross_val_score` gives us a score across different folds of cross-validated testing data, it doesn't easily give us the trained models themselves. To do this, we simply fit our pipeline with the tweets, creating a new model. The code is as follows:

```
model = pipeline.fit(tweets, labels)
```

Note that we aren't really evaluating the model here, so we don't need to be as careful with the training/testing split. However, before you put these features into practice, you should evaluate on a separate test split. We skip over that here for the sake of clarity.

A pipeline gives you access to the individual steps through the `named_steps` attribute and the name of the step (we defined these names ourselves when we created the pipeline object itself). For instance, we can get the Naive Bayes model:

```
nb = model.named_steps['naive-bayes']
```

From this model, we can extract the probabilities for each word. These are stored as log probabilities, which is simply *log(P(A | f))*, where *f* is a given feature.

The reason these are stored as log probabilities is because the actual values are very low. For instance, the first value is -3.486, which correlates to a probability under 0.03 percent. Logarithm probabilities are used in computation involving small probabilities like this as they stop underflow errors where very small values are just rounded to zeros. Given that all of the probabilities are multiplied together, a single value of 0 will result in the whole answer always being 0! Regardless, the relationship between values is still the same; the higher the value, the more useful that feature is.

We can get the most useful features by sorting the array of logarithm probabilities. We want descending order, so we simply negate the values first. The code is as follows:

```
top_features = np.argsort(-feature_probabilities[1])[:50]
```

The preceding code will just give us the indices and not the actual feature values. This isn't very useful, so we will map the feature's indices to the actual values. The key is the `DictVectorizer` step of the pipeline, which created the matrices for us. Luckily this also records the mapping, allowing us to find the feature names that correlate to different columns. We can extract the features from that part of the pipeline:

```
dv = model.named_steps['vectorizer']
```

From here, we can print out the names of the top features by looking them up in the `feature_names_` attribute of `DictVectorizer`. Enter the following lines into a new cell and run it to print out a list of the top features:

```
for i, feature_index in enumerate(top_features):
    print(i, dv.feature_names_[feature_index],
      np.exp(feature_probabilities[1][feature_index]))
```

The first few features include `:`, `http`, `#` and `@`. These are likely to be noise (although the use of a colon is not very common outside programming), based on the data we collected. Collecting more data is critical to smoothing out these issues. Looking through the list though, we get a number of more obvious programming features:

```
7 for 0.188679245283
11 with 0.141509433962
28 installing 0.0660377358491
29 Top 0.0660377358491
34 Developer 0.0566037735849
35 library 0.0566037735849
```

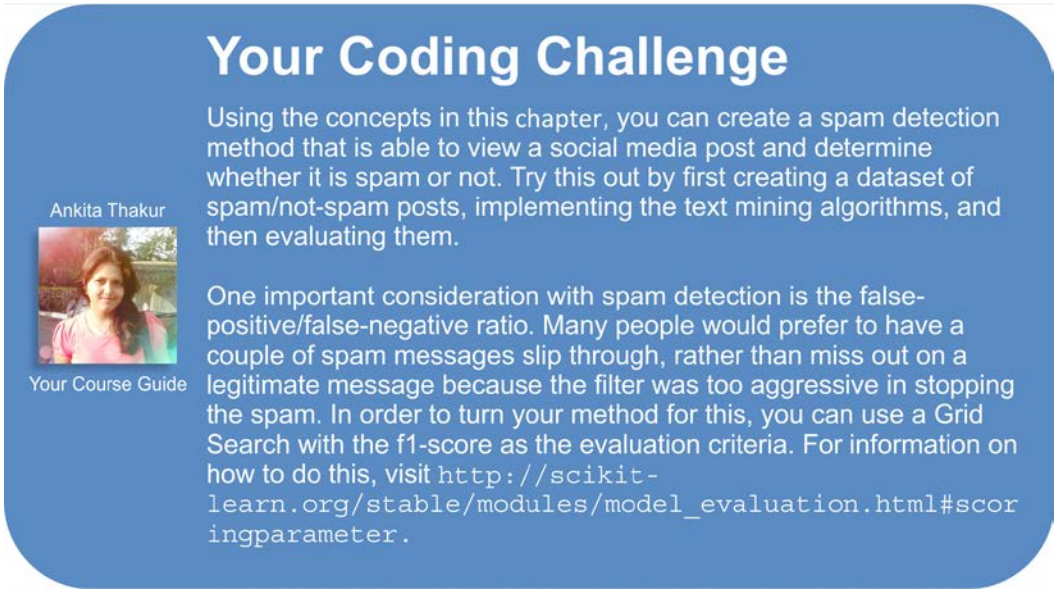
```
36 ] 0.0566037735849
37 [ 0.0566037735849
41 version 0.0471698113208
43 error 0.0471698113208
```

There are some others too that refer to Python in a work context, and therefore might be referring to the programming language (although freelance snake handlers may also use similar terms, they are less common on Twitter):

```
22 jobs 0.0660377358491
30 looking 0.0566037735849
31 Job 0.0566037735849
34 Developer 0.0566037735849
38 Freelancer 0.0471698113208
40 projects 0.0471698113208
47 We're 0.0471698113208
```

That last one is usually in the format: *We're looking for a candidate for this job*.

Looking through these features gives us quite a few benefits. We could train people to recognize these tweets, look for commonalities (which give insight into a topic), or even get rid of features that make no sense. For example, the word *RT* appears quite high in this list; however, this is a common Twitter phrase for retweet (that is, forwarding on someone else's tweet). An expert could decide to remove this word from the list, making the classifier less prone to the noise we introduced by having a small dataset.



Summary of Module 3 Chapter 6

In this chapter, we looked at text mining—how to extract features from text, how to use those features, and ways of extending those features. In doing this, we looked at putting a tweet in context—was this tweet mentioning python referring to the programming language? We downloaded data from a web-based API, getting tweets from the popular microblogging website Twitter. This gave us a dataset that we labeled using a form we built directly in the IPython Notebook.

Ankita Thakur

Your Course Guide

We also looked at reproducibility of experiments. While Twitter doesn't allow you to send copies of your data to others, it allows you to send the tweet's IDs. Using this, we created code that saved the IDs and recreated most of the original dataset. Not all tweets were returned; some had been deleted in the time since the ID list was created and the dataset was reproduced.

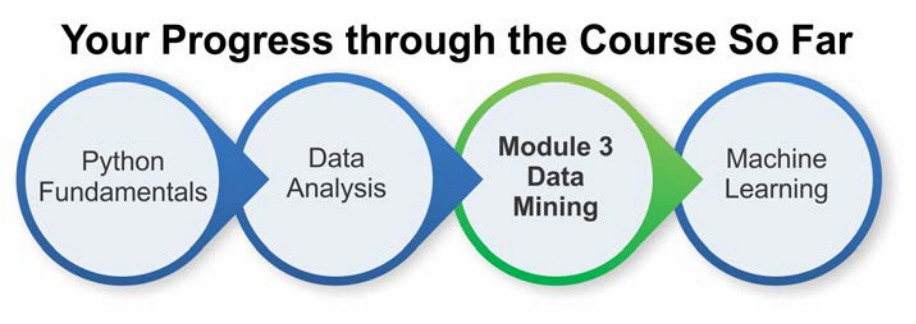
We used a Naive Bayes classifier to perform our text classification. This is built upon the Bayes' theorem that uses data to update the model, unlike the frequentist method that often starts with the model first. This allows the model to incorporate and update new data, and incorporate a prior belief. In addition, the naive part allows to easily compute the frequencies without dealing with complex correlations between features. The features we extracted were word occurrences—did this word occur in this tweet? This model is called bag-of-words. While this discards information about where a word was used, it still achieves a high accuracy on many datasets.

Ankita Thakur

Your Course Guide

This entire pipeline of using the bag-of-words model with Naive Bayes is quite robust. You will find that it can achieve quite good scores on most text-based tasks. It is a great baseline for you, before trying more advanced models. As another advantage, the Naive Bayes classifier doesn't have any parameters that need to be set (although there are some if you wish to do some tinkering).

In the next chapter, we will look at extracting features from another type of data, graphs, in order to make recommendations on who to follow on social media.



7
Discovering Accounts to Follow Using Graph Mining

Lots of things can be represented as graphs. This is particularly true in this day of Big Data, online social networks, and the Internet of Things. In particular, online social networks are big business, with sites such as Facebook that have over 500 million active users (50 percent of them log in each day). These sites often monetize themselves by targeted advertising. However, for users to be engaged with a website, they often need to follow interesting people or pages.

In this chapter, we will look at the concept of similarity and how we can create graphs based on it. We will also see how to split this graph up into meaningful subgraphs using connected components. This simple algorithm introduces the concept of cluster analysis—splitting a dataset into subsets based on similarity. We will investigate cluster analysis in more depth in *Chapter 10*, *Clustering News Articles*.

The topics covered in this chapter include:

- Creating graphs from social networks
- Loading and saving built classifiers
- The NetworkX package
- Converting graphs to matrices
- Distance and similarity
- Optimizing parameters based on scoring functions
- Loss functions and scoring functions

Loading the dataset

In this chapter, our task is to recommend users on online social networks based on shared connections. Our logic is that *if two users have the same friends, they are highly similar and worth recommending to each other*.

We are going to create a small social graph from Twitter using the API we introduced in the previous chapter. The data we are looking for is a subset of users interested in a similar topic (again, the Python programming language) and a list of all of their friends (people they follow). With this data, we will check how similar two users are, based on how many friends they have in common.

There are many other online social networks apart from Twitter. The reason we have chosen Twitter for this experiment is that their API makes it quite easy to get this sort of information. The information is available from other sites, such as Facebook, LinkedIn, and Instagram, as well. However, getting this information is more difficult.

To start collecting data, set up a new IPython Notebook and an instance of the `twitter` connection, as we did in the previous chapter. You can reuse the app information from the previous chapter or create a new one:

```
import twitter
consumer_key = "<Your Consumer Key Here>"
consumer_secret = "<Your Consumer Secret Here>"
access_token = "<Your Access Token Here>"
access_token_secret = "<Your Access Token Secret Here>"
authorization = twitter.OAuth(access_token, access_token_secret,
  consumer_key, consumer_secret)
t = twitter.Twitter(auth=authorization, retry=True)
```

Also, create the output filename:

```
import os
data_folder = os.path.join(os.path.expanduser("~"), "Data",
  "twitter")
output_filename = os.path.join(data_folder, "python_tweets.json")
```

We will also need the `json` library to save our data:

```
import json
```

Next, we will need a list of users. We will do a search for tweets, as we did in the previous chapter, and look for those mentioning the word *python*. First, create two lists for storing the tweet's text and the corresponding users. We will need the user IDs later, so we create a dictionary mapping that now. The code is as follows:

```
original_users = []
tweets = []
user_ids = {}
```

We will now perform a search for the word *python*, as we did in the previous chapter, and iterate over the search results:

```
search_results = t.search.tweets(q="python",
  count=100)['statuses']
for tweet in search_results:
```

We are only interested in tweets, not in other messages Twitter can pass along. So, we check whether there is text in the results:

```
    if 'text' in tweet:
```

If so, we record the screen name of the user, the tweet's text, and the mapping of the screen name to the user ID. The code is as follows:

```
        original_users.append(tweet['user']['screen_name'])
        user_ids[tweet['user']['screen_name']] =
          tweet['user']['id']
        tweets.append(tweet['text'])
```

Running this code will get about 100 tweets, maybe a little fewer in some cases. Not all of them will be related to the programming language, though.

Classifying with an existing model

As we learned in the previous chapter, not all tweets that mention the word *python* are going to be relating to the programming language. To do that, we will use the classifier we used in the previous chapter to get tweets based on the programming language. Our classifier wasn't perfect, but it will result in a better specialization than just doing the search alone.

In this case, we are only interested in users who are tweeting about Python, the programming language. We will use our classifier from the last chapter to determine which tweets are related to the programming language. From there, we will select only those users who were tweeting about the programming language.

To do this, we first need to save the model. Open the IPython Notebook we made in the last chapter, the one in which we built the classifier. If you have closed it, then the IPython Notebook won't remember what you did, and you will need to run the cells again. To do this, click on the **Cell** menu in the notebook and choose **Run All**.

After all of the cells have computed, choose the final blank cell. If your notebook doesn't have a blank cell at the end, choose the last cell, select the **Insert** menu, and select the **Insert Cell Below** option.

We are going to use the `joblib` library to save our model and load it.

`joblib` is included with the `scikit-learn` package.

First, import the library and create an output filename for our model (make sure the directories exist, or else they won't be created). I've stored this model in my **Models** directory, but you could choose to store them somewhere else. The code is as follows:

```
from sklearn.externals import joblib
output_filename = os.path.join(os.path.expanduser("~"), "Models",
  "twitter", "python_context.pkl")
```

Next, we use the `dump` function in `joblib`, which works like in the `json` library. We pass the model itself (which, if you have forgotten, is simply called `model`) and the output filename:

```
joblib.dump(model, output_filename)
```

Running this code will save our model to the given filename. Next, go back to the new IPython Notebook you created in the last subsection and load this model.

You will need to set the model's filename again in this Notebook by copying the following code:

```
model_filename = os.path.join(os.path.expanduser("~"), "Models",
  "twitter", "python_context.pkl")
```

Make sure the filename is the one you used just before to save the model.
Next, we need to recreate our `NLTKBOW` class, as it was a custom-built class and can't be loaded directly by `joblib`. In later chapters, we will see some better ways around this problem. For now, simply copy the entire `NLTKBOW` class from the previous chapter's code, including its dependencies:

```
from sklearn.base import TransformerMixin
from nltk import word_tokenize
```

```
class NLTKBOW(TransformerMixin):
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        return [{word: True for word in word_tokenize(document)}
                for document in X]
```

Loading the model now just requires a call to the `load` function of `joblib`:

```
from sklearn.externals import joblib
context_classifier = joblib.load(model_filename)
```

Our `context_classifier` works exactly like the `model` object of the notebook we saw in *Chapter 6*, *Social Media Insight Using Naive Bayes*, It is an instance of a `Pipeline`, with the same three steps as before (`NLTKBOW`, `DictVectorizer`, and a `BernoulliNB` classifier).

Calling the `predict` function on this model gives us a prediction as to whether our tweets are relevant to the programming language. The code is as follows:

```
y_pred = context_classifier.predict(tweets)
```

The `ith` item in `y_pred` will be 1 if the `ith` tweet is (predicted to be) related to the programming language, or else it will be 0. From here, we can get just the tweets that are relevant and their relevant users:

```
relevant_tweets = [tweets[i] for i in range(len(tweets)) if y_pred[i]
  == 1]
relevant_users = [original_users[i] for i in range(len(tweets)) if
  y_pred[i] == 1]
```

Using my data, this comes up to 46 relevant users. A little lower than our 100 tweets/users from before, but now we have a basis for building our social network.

Getting follower information from Twitter

Next, we need to get the *friends* of each of these users. A friend is a person whom the user is following. The API for this is called `friends/ids`, and it has both good and bad points. The good news is that it returns up to 5,000 friend IDs in a single API call. The bad news is that you can only make 15 calls every 15 minutes, which means it will take you at least 1 minute per user to get all followers—more if they have more than 5,000 friends (which happens more often than you may think).

However, the code is relatively easy. We will package it as a function, as we will use this code in the next two sections. First, we will create the function signature that takes our Twitter connection and a user's ID. The function will return all of the followers for that user, so we will also create a list to store these in. We will also need the time module, so we import that as well. We will first go through the composition of the function, but then I'll give you the unbroken function in its entirety. The code is as follows:

```
import time
def get_friends(t, user_id):
    friends = []
```

While it may be surprising, many Twitter users have more than 5,000 friends. So, we will need to use Twitter's **pagination**. Twitter manages multiple *pages* of data through the use of a cursor. When you ask Twitter for information, it gives that information along with a *cursor*, which is an integer that Twitter users to track your request. If there is no more information, this cursor is 0; otherwise, you can use the supplied cursor to get the next page of results. To start with, we set the cursor to -1, indicating the start of the results:

```
    cursor = -1
```

Next, we keep looping while this cursor is not equal to 0 (as, when it is, there is no more data to collect). We then perform a request for the user's followers and add them to our list. We do this in a `try` block, as there are possible errors that can happen that we can handle. The follower's IDs are stored in the `ids` key of the `results` dictionary. After obtaining that information, we update the cursor. It will be used in the next iteration of the loop. Finally, we check if we have more than 10,000 friends. If so, we break out of the loop. The code is as follows:

```
    while cursor != 0:
        try:
            results = t.friends.ids(user_id= user_id,
              cursor=cursor, count=5000)
            friends.extend([friend for friend in results['ids']])
            cursor = results['next_cursor']
            if len(friends) >= 10000:
                break
```

It is worth inserting a warning here. We are dealing with data from the Internet, which means weird things can and do happen regularly. A problem I ran into when developing this code was that some users have many, many, many thousands of friends. As a fix for this issue, we will put a failsafe here, exiting if we reach more than 10,000 users. If you want to collect the full dataset, you can remove these lines, but beware that it may get stuck on a particular user for a very long time.

We now handle the errors that can happen. The most likely error that can occur happens if we accidentally reached our API limit (while we have a `sleep` to stop that, it can occur if you stop and run your code before this `sleep` finishes). In this case, `results is None` and our code will fail with a `TypeError`. In this case, we wait for 5 minutes and try again, hoping that we have reached our next 15-minute window. There may be another `TypeError` that occurs at this time. If one of them does, we raise it and will need to handle it separately. The code is as follows:

```
except TypeError as e:
    if results is None:
        print("You probably reached your API limit,
          waiting for 5 minutes")
        sys.stdout.flush()
        time.sleep(5*60) # 5 minute wait
    else:
        raise e
```

The second error that can happen occurs at Twitter's end, such as asking for a user that doesn't exist or some other data-based error. In this case, don't try this user anymore and just return any followers we did get (which, in this case, is likely to be 0). The code is as follows:

```
except twitter.TwitterHTTPError as e:
    break
```

Now, we will handle our API limit. Twitter only lets us ask for follower information 15 times every 15 minutes, so we will wait for 1 minute before continuing. We do this in a `finally` block so that it happens even if an error occurs:

```
finally:
    time.sleep(60)
```

We complete our function by returning the friends we collected:

```
return friends
```

The full function is given as follows:

```
import time
def get_friends(t, user_id):
    friends = []
    cursor = -1
    while cursor != 0:
        try:
            results = t.friends.ids(user_id= user_id,
              cursor=cursor, count=5000)
            friends.extend([friend for friend in
              results['ids']])
            cursor = results['next_cursor']
            if len(friends) >= 10000:
                break
        except TypeError as e:
            if results is None:
                print("You probably reached your API limit,
                  waiting for 5 minutes")
                sys.stdout.flush()
                time.sleep(5*60) # 5 minute wait
            else:
                raise e
        except twitter.TwitterHTTPError as e:
                break
               finally:
                   time.sleep(60)
    return friends
```

Building the network

Now we are going to build our network. Starting with our original users, we will get the friends for each of them and store them in a dictionary (after obtaining the user's ID from our `user_id` dictionary):

```
friends = {}
for screen_name in relevant_users:
    user_id = user_ids[screen_name]
    friends[user_id] = get_friends(t, user_id)
```

Next, we are going to remove any user who doesn't have any friends. For these users, we can't really make a recommendation in this way. Instead, we might have to look at their content or people who follow them. We will leave that out of the scope of this chapter, though, so let's just remove these users. The code is as follows:

```
friends = {user_id:friends[user_id] for user_id in friends
             if len(friends[user_id]) > 0}
```

We now have between 30 and 50 users, depending on your initial search results. We are now going to increase that amount to 150. The following code will take quite a long time to run—given the limits on the API, we can only get the friends for a user once every minute. Simple math will tell us that 150 users will take 150 minutes, or 2.5 hours. Given the time we are going to be spending on getting this data, it pays to ensure we get only *good* users.

What makes a good user, though? Given that we will be looking to make recommendations based on shared connections, we will search for users based on shared connections. We will get the friends of our existing users, starting with those users who are better connected to our existing users. To do that, we maintain a count of all the times a user is in one of our `friends` lists. It is worth considering the goals of the application when considering your sampling strategy. For this purpose, getting lots of similar users enables the recommendations to be more regularly applicable.

To do this, we simply iterate over all the `friends` lists we have and then count each time a friend occurs.

```
from collections import defaultdict
def count_friends(friends):
    friend_count = defaultdict(int)
    for friend_list in friends.values():
        for friend in friend_list:
            friend_count[friend] += 1
    return friend_count
```

Computing our current friend count, we can then get the most connected (that is, most friends from our existing list) person from our sample. The code is as follows:

```
friend_count
reverse=True) = count_friends(friends)
from operator import itemgetter
best_friends = sorted(friend_count.items(), key=itemgetter(1),
```

From here, we set up a loop that continues until we have the friends of 150 users. We then iterate over all of our best friends (which happens in order of the number of people who have them as friends) until we find a user whose friends we haven't already got. We then get the friends of that user and update the `friends` counts. Finally, we work out who is the most connected user who we haven't already got in our list:

```
while len(friends) < 150:
    for user_id, count in best_friends:
        if user_id not in friends:
            break
        friends[user_id] = get_friends(t, user_id)
    for friend in friends[user_id]:
        friend_count[friend] += 1
    best_friends = sorted(friend_count.items(),
      key=itemgetter(1), reverse=True)
```

The codes will then loop and continue until we reach 150 users.

You may want to set these value lower, such as 40 or 50 users (or even just skip this bit of code temporarily). Then, complete the chapter's code and get a feel for how the results work. After that, reset the number of users in this loop to 150, leave the code to run for a few hours, and then come back and rerun the later code.

Given that collecting that data probably took over 2 hours, it would be a good idea to save it in case we have to turn our computer off. Using the `json` library, we can easily save our friends dictionary to a file:

```
import json
friends_filename = os.path.join(data_folder, "python_friends.json")
with open(friends_filename, 'w') as outf:
    json.dump(friends, outf)
```

If you need to load the file, use the `json.load` function:

```
with open(friends_filename) as inf:
    friends = json.load(inf)
```

Creating a graph

Now, we have a list of users and their friends and many of these users are taken from friends of other users. This gives us a graph where some users are friends of other users (although not necessarily the other way around).

A graph is a set of nodes and edges. Nodes are usually objects—in this case, they are our users. The edges in this initial graph indicate that *user A is a friend of user B*. We call this a directed graph, as the order of the nodes matters. Just because user A is a friend of user B, that doesn't imply that user B is a friend of user A. We can visualize this graph using the **NetworkX** package.

Once again, you can use `pip` to install NetworkX: `pip3 install networkx`.

First, we create a directed graph using NetworkX. By convention, when importing NetworkX, we use the abbreviation `nx` (although this isn't necessary). The code is as follows:

```
import networkx as nx
G = nx.DiGraph()
```

We will only visualize our key users, not all of the friends (as there are many thousands of these and it is hard to visualize). We get our main users and then add them to our graph as nodes. The code is as follows:

```
main_users = friends.keys()
G.add_nodes_from(main_users)
```

Next we set up the edges. We create an edge from a user to another user if the second user is a friend of the first user. To do this, we iterate through all of the friends:

```
for user_id in friends:
    for friend in friends[user_id]:
```

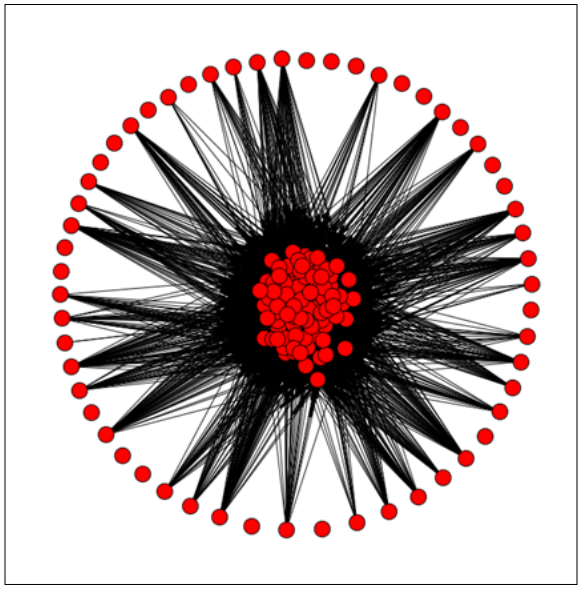
We ensure that the friend is one of our main users (as we currently aren't interested in the other ones), and add the edge if they are. The code is as follows:

```
            if friend in main_users:
               G.add_edge(user_id, friend)
```

We can now visualize network using NetworkX's `draw` function, which uses `matplotlib`. To get the image in our notebook, we use the `inline` function on `matplotlib` and then call the `draw` function. The code is as follows:

```
%matplotlib inline
nx.draw(G)
```

The results are a bit hard to make sense of; they show that there are some nodes with few connections but many nodes with many connections:

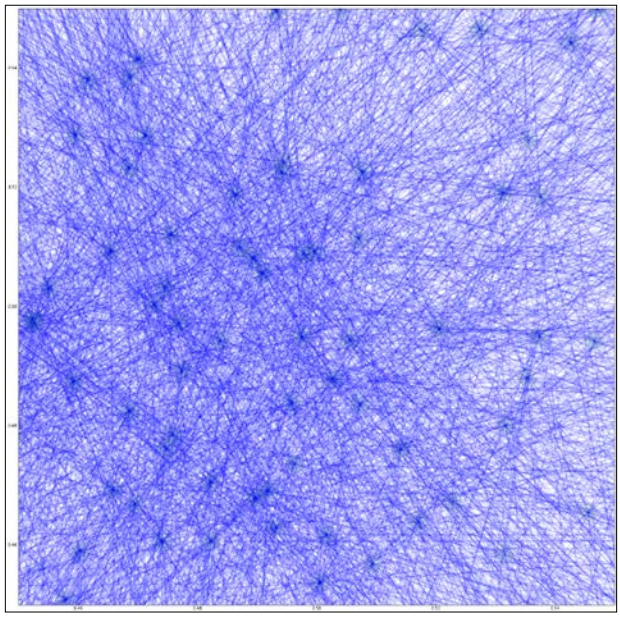


We can make the graph a bit bigger by using `pyplot` to handle the creation of the figure. To do that, we import `pyplot`, create a large figure, and then call NetworkX's `draw` function (NetworkX uses `pyplot` to draw its figures):

```
from matplotlib import pyplot as plt
plt.figure(3,figsize=(20,20))
nx.draw(G, alpha=0.1, edge_color='b')
```

The results are too big for a page here, but by making the graph bigger, an outline of how the graph appears can now be seen. In my graph, there was a major group of users all highly connected to each other, and most other users didn't have many connections at all. I've zoomed in on just the center of the network here and set the edge color to blue with a low `alpha` in the preceding code.

As you can see, it is very well connected in the center!



This is actually a property of our method of choosing new users—we choose those who are already well linked in our graph, so it is likely they will just make this group larger. For social networks, generally the number of connections a user has follows a power law. A small percentage of users have many connections, and others have only a few. The shape of the graph is often described as having a *long tail*. Our dataset doesn't follow this pattern, as we collected our data by getting friends of users we already had.

Creating a similarity graph

Our task in this chapter is recommendation through shared friends. As mentioned previously, our logic is that, *if two users have the same friends, they are highly similar*. We could recommend one user to the other on this basis.

We are therefore going to take our existing graph (which has edges relating to friendship) and create a new graph. The nodes are still users, but the edges are going to be weighted edges. A weighted edge is simply an edge with a `weight` property. The logic is that a higher weight indicates more similarity between the two nodes than a lower weight. This is context-dependent. If the weights represent distance, then the lower weights indicate more similarity.

For our application, the weight will be the similarity of the two users connected by that edge (based on the number of friends they share). This graph also has the property that it is not directed. This is due to our similarity computation, where the similarity of user A to user B is the same as the similarity of user B to user A.

There are many ways to compute the similarity between two lists like this. For example, we could compute the number of friends the two have in common. However, this measure is always going to be higher for people with more friends. Instead, we can normalize it by dividing by the total number of distinct friends the two have. This is called the **Jaccard Similarity**.

The Jaccard Similarity, always between 0 and 1, represents the percentage overlap of the two. As we saw in *Chapter 2*, *Classifying with scikit-learn Estimators*, normalization is an important part of data mining exercises and generally a good thing to do (unless you have a specific reason not to).

To compute this Jaccard similarity, we divide the intersection of the two sets of followers by the union of the two. These are `set` operations and we have `lists`, so we will need to convert the `friends` lists to sets first. The code is as follows:

```
friends = {user: set(friends[user]) for user in friends}
```

We then create a function that computes the similarity of two sets of `friends` lists. The code is as follows:

```
def compute_similarity(friends1, friends2):
return len(friends1 & friends2) / len(friends1 | friends2)
```

From here, we can create our weighted graph of the similarity between users. We will use this quite a lot in the rest of the chapter, so we will create a function to perform this action. Let's take a look at the threshold parameter:

```
def create_graph(followers, threshold=0):
    G = nx.Graph()
```

We iterate over all combinations of users, ignoring instances where we are comparing a user with themselves:

```
    for user1 in friends.keys():
        for user2 in friends.keys():
            if user1 == user2:
                continue
```

We compute the weight of the edge between the two users:

```
            weight = compute_similarity(friends[user1],
friends[user2])
```

Next, we will only add the edge if it is above a certain threshold. This stops us from adding edges we don't care about—for example, edges with weight 0. By default, our threshold is 0, so we will be including all edges right now. However, we will use this parameter later in the chapter. The code is as follows:

```
            if weight >= threshold:
```

If the weight is above the threshold, we add the two users to the graph (they won't be added as a duplicate if they are already in the graph):

```
                G.add_node(user1)
                G.add_node(user2)
```

We then add the edge between them, setting the weight to be the computed similarity:

```
                G.add_edge(user1, user2, weight=weight)
```

Once the loops have finished, we have a completed graph and we return it from the function:

```
    return G
```

We can now create a graph by calling this function. We start with no threshold, which means all links are created. The code is as follows:

```
G = create_graph(friends)
```

The result is a very strongly connected graph—all nodes have edges, although many of those will have a weight of 0. We will see the weight of the edges by drawing the graph with line widths relative to the weight of the edge—thicker lines indicate higher weights.

Due to the number of nodes, it makes sense to make the figure larger to get a clearer sense of the connections:

```
plt.figure(figsize=(10,10))
```

We are going to draw the edges with a weight, so we need to draw the nodes first. NetworkX uses layouts to determine where to put the nodes and edges, based on certain criteria. Visualizing networks is a very difficult problem, especially as the number of nodes grows. Various techniques exist for visualizing networks, but the degree to which they work depends heavily on your dataset, personal preferences, and the aim of the visualization. I found that the `spring_layout` worked quite well, but other options such as `circular_layout` (which is a good default if nothing else works), `random_layout`, `shell_layout`, and `spectral_layout` also exist.

Visit `http://networkx.lanl.gov/reference/drawing.html` for more details on layouts in NetworkX. Although it adds some complexity, the `draw_graphviz` option works quite well and is worth investigating for better visualizations. It is well worth considering in real-world uses.

Let's use `spring_layout` for visualization:

```
pos = nx.spring_layout(G)
```

Using our `pos` layout, we can then position the nodes:

```
nx.draw_networkx_nodes(G, pos)
```

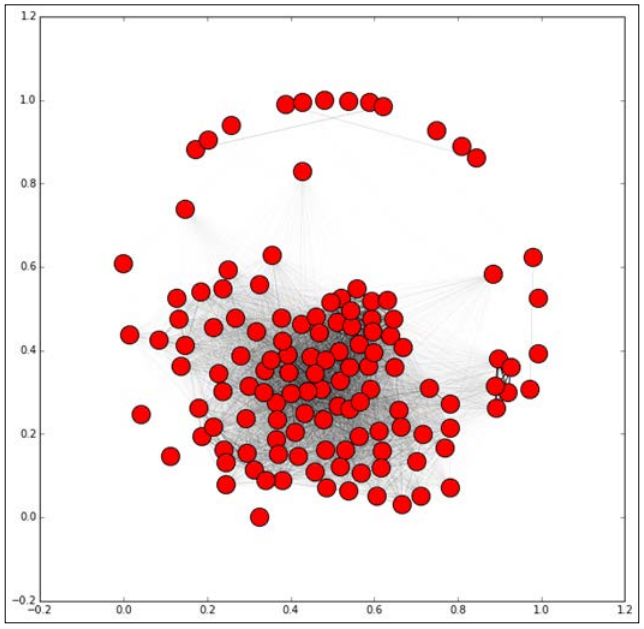
Next, we draw the edges. To get the weights, we iterate over the edges in the graph (in a specific order) and collect the weights:

```
edgewidth = [ d['weight'] for (u,v,d) in G.edges(data=True)]
```

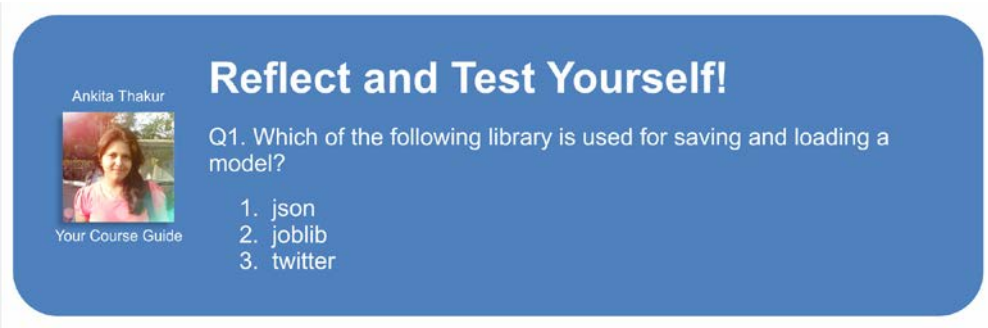
We then draw the edges:

```
nx.draw_networkx_edges(G, pos, width=edgewidth)
```

The result will depend on your data, but it will typically show a graph with a large set of nodes connected quite strongly and a few nodes poorly connected to the rest of the network.



The difference in this graph compared to the previous graph is that the edges determine the similarity between the nodes based on our similarity metric and not on whether one is a friend of another (although there are similarities between the two!). We can now start extracting information from this graph in order to make our recommendations.



Finding subgraphs

From our similarity function, we could simply rank the results for each user, returning the most similar user as a recommendation—as we did with our product recommendations. Instead, we might want to find clusters of users that are all similar to each other. We could advise these users to start a group, create advertising targeting this segment, or even just use those clusters to do the recommendations themselves.

Finding these clusters of similar users is a task called cluster analysis. It is a difficult task, with complications that classification tasks do not typically have. For example, evaluating classification results is relatively easy—we compare our results to the ground truth (from our training set) and see what percentage we got right. With cluster analysis, though, there isn't typically a ground truth. Evaluation usually comes down to seeing if the clusters *make sense*, based on some preconceived notion we have of what the cluster should look like. Another complication with cluster analysis is that the model can't be trained against the expected result to learn—it has to use some approximation based on a mathematical model of a cluster, not what the user is hoping to achieve from the analysis.

Connected components

One of the simplest methods for clustering is to find the **connected components** in a graph. A connected component is a set of nodes in a graph that are connected via edges. Not all nodes need to be connected to each other to be a connected component. However, for two nodes to be in the same connected component, there needs to be a way to travel from one node to another in that connected component.

Connected components do not consider edge weights when being computed; they only check for the presence of an edge. For that reason, the code that follows will remove any edge with a low weight.

NetworkX has a function for computing connected components that we can call on our graph. First, we create a new graph using our `create_graph` function, but this time we pass a threshold of 0.1 to get only those edges that have a weight of at least 0.1.

```
G = create_graph(friends, 0.1)
```

We then use NetworkX to find the connected components in the graph:

```
sub_graphs = nx.connected_component_subgraphs(G)
```

To get a sense of the sizes of the graph, we can iterate over the groups and print out some basic information:

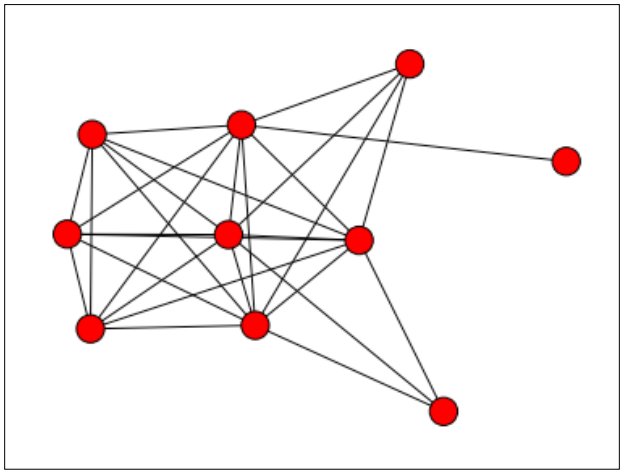
```
for i, sub_graph in enumerate(sub_graphs):
    n_nodes = len(sub_graph.nodes())
    print("Subgraph {0} has {1} nodes".format(i, n_nodes))
```

The results will tell you how big each of the connected components is. My results had one big subgraph of 62 users and lots of little ones with a dozen or fewer users.

We can alter the threshold to alter the connected components. This is because a higher threshold has fewer edges connecting nodes, and therefore will have smaller connected components and more of them. We can see this by running the preceding code with a higher threshold:

```
G = create_graph(friends, 0.25)
sub_graphs = nx.connected_component_subgraphs(G)
for i, sub_graph in enumerate(sub_graphs):
    n_nodes = len(sub_graph.nodes())
    print("Subgraph {0} has {1} nodes".format(i, n_nodes))
```

The preceding code gives us much smaller nodes and more of them. My largest cluster was broken into at least three parts and none of the clusters had more than 10 users. An example cluster is shown in the following figure, and the connections within this cluster are also shown. Note that, as it is a connected component, there were no edges from nodes in this component to other nodes in the graph (at least, with the threshold set at 0.25):



We can graph the entire set too, showing each connected component in a different color. As these connected components are not connected to each other, it actually makes little sense to plot these on a single graph. This is because the positioning of the nodes and components is arbitrary, and it can confuse the visualization. Instead, we can plot each separately on a separate subfigure.

In a new cell, obtain the connected components and also the count of the connected components:

```
sub_graphs = nx.connected_component_subgraphs(G)
n_subgraphs = nx.number_connected_components(G)
```

`sub_graphs` is a generator, not a list of the connected components. For this reason, use `nx.number_connected_components` to find out how many connected components there are; don't use `len`, as it doesn't work due to the way that NetworkX stores this information. This is why we need to recompute the connected components here.

Create a new `pyplot` figure and give enough room to show all of our connected components. For this reason, we allow the graph to increase in size with the number of connected components:

```
fig = plt.figure(figsize=(20, (n_subgraphs * 3)))
```

Next, iterate over each connected component and add a subplot for each. The parameters to `add_subplot` are the number of rows of subplots, the number of columns, and the index of the subplot we are interested in. My visualization uses three columns, but you can try other values instead of three (just remember to change both values):

```
for i, sub_graph in enumerate(sub_graphs):
    ax = fig.add_subplot(int(n_subgraphs / 3), 3, i)
```

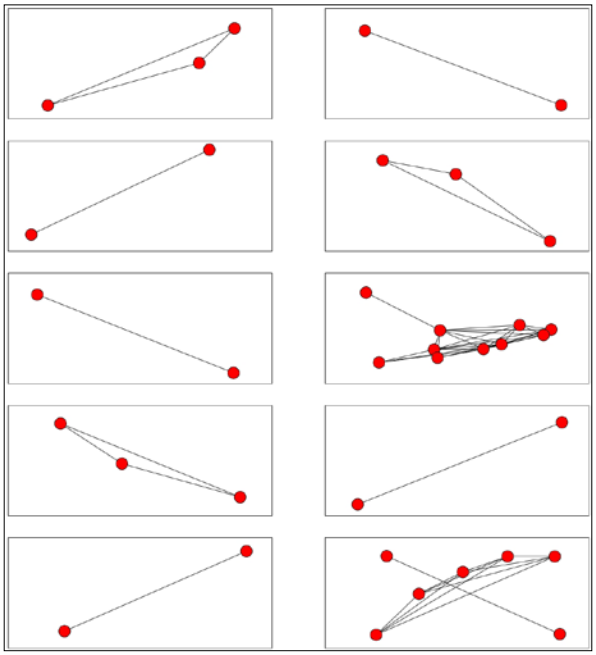
By default, `pyplot` shows plots with axis labels, which are meaningless in this context. For that reason, we turn labels off:

```
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```

Then we plot the nodes and edges (using the `ax` parameter to plot to the correct subplot). To do this, we also need to set up a layout first:

```
    pos = nx.spring_layout(G)
    nx.draw_networkx_nodes(G, pos, sub_graph.nodes(), ax=ax,
      node_size=500)
    nx.draw_networkx_edges(G, pos, sub_graph.edges(), ax=ax)
```

The results visualize each connected component, giving us a sense of the number of nodes in each and also how connected they are.



Optimizing criteria

Our algorithm for finding these connected components relies on the threshold parameter, which dictates whether edges are added to the graph or not. In turn, this directly dictates how many connected components we discover and how big they are. From here, we probably want to settle on some notion of which is the *best* threshold to use. This is a very subjective problem, and there is no definitive answer. This is a major problem with any cluster analysis task.

We can, however, determine what we think a good solution should look like and define a metric based on that idea. As a general rule, we usually want a solution where:

- Samples in the same cluster (connected components) are highly *similar* to each other
- Samples in different clusters are highly *dissimilar* to each other

The `Silhouette Coefficient` is a metric that quantifies these points. Given a single sample, we define the Silhouette Coefficient as follows:

$$s = \frac{b - a}{\max(a, b)}$$

Where *a* is the **intra-cluster distance** or the average distance to the other samples in the sample's cluster, and *b* is the inter-cluster distance or the average distance to the other samples in the *next-nearest* cluster.

To compute the overall Silhouette Coefficient, we take the mean of the Silhouettes for each sample. A clustering that provides a Silhouette Coefficient close to the maximum of 1 has clusters that have samples all similar to each other, and these clusters are very spread apart. Values near 0 indicate that the clusters all overlap and there is little distinction between clusters. Values close to the minimum of -1 indicate that samples are probably in the wrong cluster, that is, they would be better off in other clusters.

Using this metric, we want to find a solution (that is, a value for the threshold) that maximizes the Silhouette Coefficient by altering the threshold parameter. To do that, we create a function that takes the threshold as a parameter and computes the Silhouette Coefficient.

We then pass this into the `optimize` module of SciPy, which contains the `minimize` function that is used to find the minimum value of a function by altering one of the parameters. While we are interested in maximizing the Silhouette Coefficient, SciPy doesn't have a maximize function. Instead, we minimize the inverse of the Silhouette (which is basically the same thing).

The scikit-learn library has a function for computing the Silhouette Coefficient, `sklearn.metrics.silhouette_score`; however, it doesn't fix the function format that is required by the SciPy `minimize` function. The minimize function requires the variable parameter to be first (in our case, the threshold value), and any arguments to be after it. In our case, we need to pass the `friends` dictionary as an argument in order to compute the graph. The code is as follows:

```
def compute_silhouette(threshold, friends):
```

We then create the graph using the threshold parameter, and check it has at least some nodes:

```
    G = create_graph(friends, threshold=threshold)
    if len(G.nodes()) < 2:
```

The Silhouette Coefficient is not defined unless there are at least two nodes (in order for distance to be computed at all). In this case, we define the problem scope as invalid. There are a few ways to handle this, but the easiest is to return a very poor score. In our case, the minimum value that the Silhouette Coefficient can take is -1, and we will return -99 to indicate an invalid problem. Any valid solution will score higher than this. The code is as follows:

```
        return -99
```

We then extract the connected components:

```
    sub_graphs = nx.connected_component_subgraphs(G)
```

The Silhouette is also only defined if we have at least two connected components (in order to compute the inter-cluster distance), and at least one of these connected components has two members (to compute the intra-cluster distance). We test for these conditions and return our invalid problem score if it doesn't fit. The code is as follows:

```
    if not (2 <= nx.number_connected_components() < len(G.nodes())
      - 1):
        return -99
```

Next, we need to get the labels that indicate which connected component each sample was placed in. We iterate over all the connected components, noting in a dictionary which user belonged to which connected component. The code is as follows:

```
label_dict = {}
for i, sub_graph in enumerate(sub_graphs):
    for node in sub_graph.nodes():
        label_dict[node] = i
```

Then we iterate over the nodes in the graph to get the label for each node in order. We need to do this two-step process, as nodes are not clearly ordered within a graph but they do maintain their order as long as no changes are made to the graph. What this means is that, until we change the graph, we can call `.nodes()` on the graph to get the same ordering. The code is as follows:

```
labels = np.array([label_dict[node] for node in G.nodes()])
```

Next the Silhouette Coefficient function takes a *distance matrix*, not a *graph*. Addressing this is another two-step process. First, NetworkX provides a handy function `to_scipy_sparse_matrix`, which returns the graph in a matrix format that we can use:

```
X = nx.to_scipy_sparse_matrix(G).todense()
```

The Silhouette Coefficient implementation in scikit-learn, at the time of writing, doesn't support sparse matrices. For this reason, we need to call the `todense()` function. Typically, this is a bad idea—sparse matrices are usually used because the data typically shouldn't be in a dense format. In this case, it will be fine because our dataset is relatively small; however, don't try this for larger datasets.

For evaluating sparse datasets, I recommended that you look into V-Measure or Adjusted Mutual Information. These are both implemented in scikit-learn, but they have very different parameters for performing their evaluation.

However, the values are based on our weights, which are a similarity and not a distance. For a distance, higher values indicate more difference. We can convert from similarity to distance by subtracting the value from the maximum possible value, which for our weights was 1:

```
X = 1 - X
```

Now we have our distance matrix and labels, so we have all the information we need to compute the Silhouette Coefficient. We pass the metric as `precomputed`; otherwise, the matrix X will be considered a feature matrix, not a distance matrix (feature matrices are used by default nearly everywhere in scikit-learn). The code is as follows:

```
return silhouette_score(X, labels, metric='precomputed')
```

We have two forms of inversion happening here. The first is taking the inverse of the similarity to compute a distance function; this is needed, as the Silhouette Coefficient only accepts distances. The second is the inverting of the Silhouette Coefficient score so that we can minimize with SciPy's `optimize` module.

We have one small problem, though. This function returns the Silhouette Coefficient, which is a score where higher values are considered better. Scipy's `optimize` module only defines a `minimize` function, which works off a `loss` function where lower scores are better. We can fix this by inverting the value, which takes our `score` function and returns a `loss` function.

```
def inverted_silhouette(threshold, friends):
    return -compute_silhouette(threshold, friends)
```

This function creates a new function from an original function. When the new function is called, all of the same arguments and keywords are passed onto the original function and the return value is returned, except that this returned value is negated before it is returned.

Now we can do our actual optimization. We call the `minimize` function on the inverted `compute_silhouette` function we defined:

```
result = minimize(inverted_silhouette, 0.1, args=(friends,))
```

The parameters are as follows:

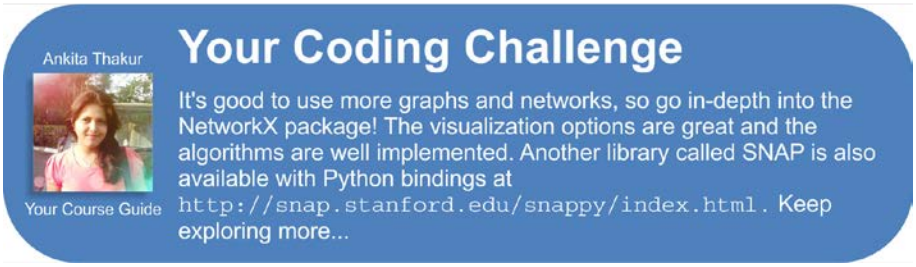
- `invert(compute_silhouette)`: This is the function we are trying to minimize (remembering that we invert it to turn it into a loss function)
- `0.1`: This is an initial guess at a threshold that will minimize the function
- `options={'maxiter':10}`: This dictates that only 10 iterations are to be performed (increasing this will probably get a better result, but will take longer to run)
- `method='nelder-mead'`: This is used to select the Nelder-Mead optimize routing (SciPy supports quite a number of different options)
- `args=(friends,)`: This passes the `friends` dictionary to the function that is being minimized

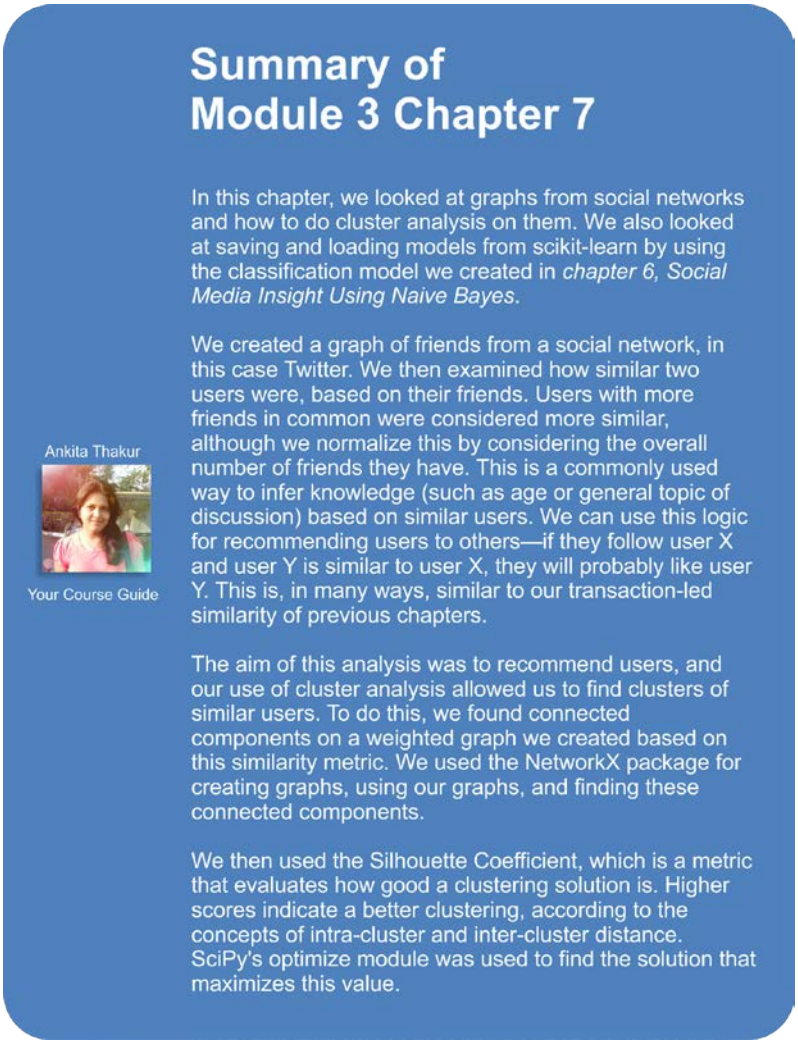
This function will take quite a while to run. Our graph creation function isn't that fast, nor is the function that computes the Silhouette Coefficient. Decreasing the `maxiter` value will result in fewer iterations being performed, but we run the risk of finding a suboptimal solution.

Running this function, I got a threshold of 0.135 that returns 10 components. The score returned by the minimize function was -0.192. However, we must remember that we negated this value. This means our score was actually 0.192. The value is positive, which indicates that the clusters tend to be more separated than not (a good thing). We could run other models and check whether it results in a better score, which means that the clusters are better separated.

We could use this result to recommend users—if a user is in a connected component, then we can recommend other users in that component. This recommendation follows our use of the Jaccard Similarity to find good connections between users, our use of connected components to split them up into clusters, and our use of the optimization technique to find the best model in this setting.

However, a large number of users may not be connected at all, so we will use a different algorithm to find clusters for them.



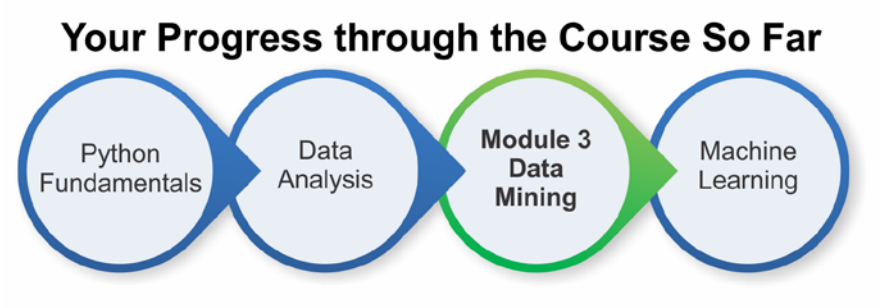


Ankita Thakur

Your Course Guide

In this chapter, we compared a few opposites too. **Similarity** is a measure between two objects, where *higher* values indicate more similarity between those objects. In contrast, **distance** is a measure where *lower* values indicate more similarity. Another contrast we saw was a loss function, where lower scores are considered better (that is, we lost less). Its opposite is the score function, where *higher* scores are considered better.

In the next chapter, we will see how to extract features from another new type of data: images. We will discuss how to use neural networks to identify numbers in images and develop a program to automatically beat CAPTCHA images.



8
Beating CAPTCHAs with Neural Networks

Interpreting information contained in images has long been a difficult problem in data mining, but it is one that is really starting to be addressed. The latest research is providing algorithms to detect and understand images to the point where automated commercial surveillance systems are now being used—in real-world scenarios—by major vendors. These systems are capable of understanding and recognizing objects and people in video footage.

It is difficult to extract information from images. There is lots of raw data in an image, and the standard method for encoding images—pixels—isn't that informative by itself. Images—particularly photos—can be blurry, too close to the targets, too dark, too light, scaled, cropped, skewed, or any other of a variety of problems that cause havoc for a computer system trying to extract useful information.

In this chapter, we look at extracting text from images by using neural networks for predicting each letter. The problem we are trying to solve is to automatically understand CAPTCHA messages. CAPTCHAs are images designed to be easy for humans to solve and hard for a computer to solve, as per the acronym: Completely Automated Public Turing test to tell Computers and Humans Apart. Many websites use them for registration and commenting systems to stop automated programs flooding their site with fake accounts and spam comments.

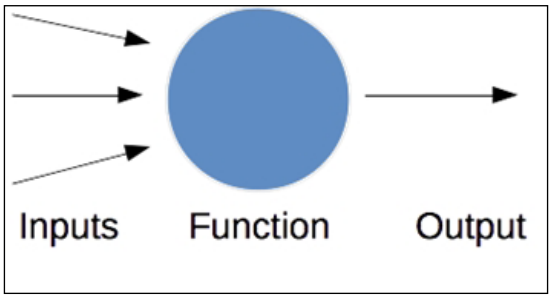
The topics covered in this chapter include:

- Neural networks
- Creating our own dataset of CAPTCHAs and letters
- The scikit-image library for working with image data
- The PyBrain library for neural networks

- Extracting basic features from images
- Using neural networks for larger-scale classification tasks
- Improving performance using postprocessing

Artificial neural networks

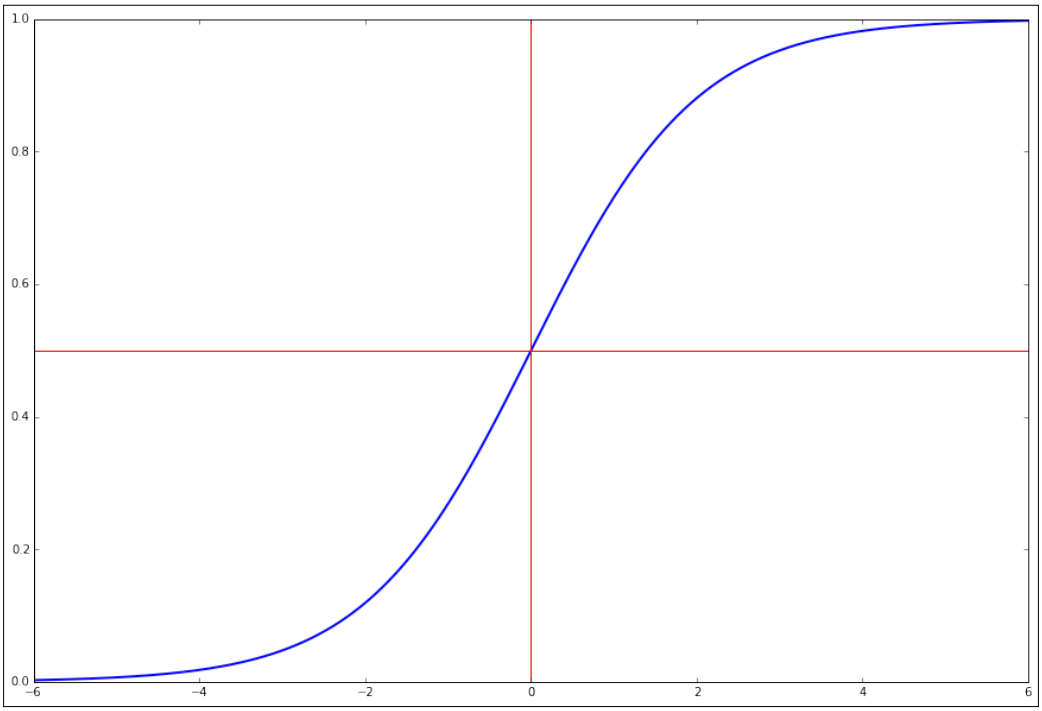
Neural networks are a class of algorithm that was originally designed based on the way that human brains work. However, modern advances are generally based on mathematics rather than biological insights. A neural network is a collection of **neurons** that are connected together. Each neuron is a simple function of its inputs, which generates an output:



The functions that define a neuron's processing can be any standard function, such as a linear combination of the inputs, and are called the **activation function**. For the commonly used learning algorithms to work, we need the activation function to be derivable and smooth. A frequently used activation function is the logistic function, which is defined by the following equation (*k* is often simply 1, *x* is the inputs into the neuron, and *L* is normally 1, that is, the maximum value of the function):

$$f(x) = \frac{L}{1 + \mathrm{e}^{-k(x - x_0)}}$$

The value of this graph, from -6 to +6, is shown as follows:



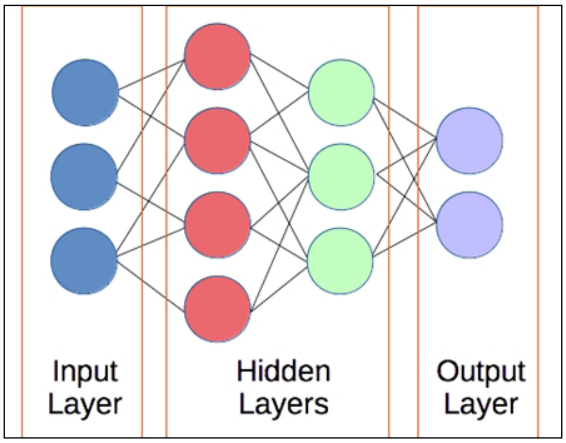
The red lines indicate that the value is 0.5 when x is zero.

Each individual neuron receives its inputs and then computes the output based on these values. Neural networks are simply networks of these neurons connected together, and they can be very powerful for data mining applications. The combinations of these neurons, how they fit together, and how they combine to learn a model are one of the most powerful concepts in machine learning.

An introduction to neural networks

For data mining applications, the arrangement of neurons is usually in layers. The first layer, the **input layer**, takes the inputs from the dataset. The outputs of each of these neurons are computed and then passed along to the neurons in the next layer. This is called a **feed-forward neural network**. We will refer to these simply as neural networks for this chapter. There are other types of neural networks too that are used for different applications. We will see another type of network in *Chapter 11*, *Classifying Objects in Images Using Deep Learning*.

The outputs of one layer become the inputs of the next layer, continuing until we reach the final layer: the, **output layer**. These outputs represent the predictions of the neural network as the classification. Any layer of neurons between the input layer and the output layer is referred to as a **hidden layer**, as they learn a representation of the data not intuitively interpretable by humans. Most neural networks have at least three layers, although most modern applications use networks with many more layers than that.



Typically, we consider fully connected layers. The outputs of each neuron in a layer go to all neurons in the next layer. While we do define a fully connected network, many of the weights will be set to zero during the training process, effectively removing these links. Fully connected neural networks are also simpler and more efficient to program than other connection patterns.

As the function of the neurons is normally the logistic function, and the neurons are fully connected to the next layer, the parameters for building and training a neural network must be other factors. The first factor for neural networks is in the building phase: the size of the neural network. This includes how many layers the neural network has and how many neurons it has in each hidden layer (the size of the input and output layers is usually dictated by the dataset).

The second parameter for neural networks is determined in the training phase: the weight of the connections between neurons. When one neuron connects to another, this connection has an associated **weight** that is multiplied by the signal (the output of the first neuron). If the connection has a weight of 0.8, the neuron is activated, and it outputs a value of 1, the resulting input to the next neuron is 0.8. If the first neuron is not activated and has a value of 0, this stays at 0.

The combination of an appropriately sized network and well-trained weights determines how accurate the neural network can be when making classifications. The word "appropriately" also doesn't necessarily mean bigger, as neural networks that are too large can take a long time to train and can more easily overfit the training data.

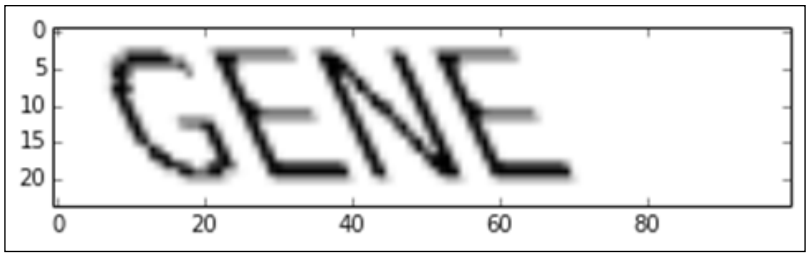
Weights are normally set randomly to start with, but are then updated during the training phase.

We now have a classifier that has initial parameters to set (the size of the network) and parameters to train from the dataset. The classifier can then be used to predict the target of a data sample based on the inputs, much like the classification algorithms we have used in previous chapters. But first, we need a dataset to train and test with.

Creating the dataset

In this chapter, we will take on the role of the bad guy. We want to create a program that can beat CAPTCHAs, allowing our comment spam program to advertise on someone's website. It should be noted that our CAPTCHAs will be a little easier that those used on the web today and that spamming isn't a very nice thing to do.

Our CAPTCHAs will be individual English words of four letters only, as shown in the following image:



Our goal will be to create a program that can recover the word from images like this. To do this, we will use four steps:

1. Break the image into individual letters.
2. Classify each individual letter.
3. Recombine the letters to form a word.
4. Rank words with a dictionary to try to fix errors.

Our CAPTCHA-busting algorithm will make the following assumptions. First, the word will be a whole and valid four-character English word (in fact, we use the same dictionary for creating and busting CAPTCHAs). Second, the word will only contain uppercase letters. No symbols, numbers, or spaces will be used. We are going to make the problem slightly harder: we are going to perform a *shear* transform to the text, along with varying rates of shearing.

Drawing basic CAPTCHAs

Next, we develop our function for creating our CAPTCHA. Our goal here is to draw an image with a word on it, along with a shear transform. We are going to use the `PIL` library to draw our CAPTCHAs and the `scikit-image` library to perform the shear transform. The `scikit-image` library can read images in a NumPy array format that `PIL` can export to, allowing us to use both libraries.

Both `PIL` and `scikit-image` can be installed via `pip`:

```
pip install PIL
pip install scikit-image
```

First, we import the necessary libraries and modules. We import NumPy and the `Image` drawing functions as follows:

```
import numpy as np
from PIL import Image, ImageDraw, ImageFont
from skimage import transform as tf
```

Then we create our base function for generating CAPTCHAs. This function takes a word and a shear value (which is normally between 0 and 0.5) to return an image in a NumPy array format. We allow the user to set the size of the resulting image, as we will use this function for single-letter training samples as well. The code is as follows:

```
def create_captcha(text, shear=0, size=(100, 24)):
```

We create a new image using L for the format, which means black-and-white pixels only, and create an instance of the `ImageDraw` class. This allows us to draw on this image using `PIL`. The code is as follows:

```
    im = Image.new("L", size, "black")
    draw = ImageDraw.Draw(im)
```

Next we set the font of the CAPTCHA we will use. You will need a font file and the filename in the following code (`Coval.otf`) should point to it (I just placed the file in the `Notebook's` directory.

```
font = ImageFont.truetype(r"Coval.otf", 22)
draw.text((2, 2), text, fill=1, font=font)
```

You can get the Coval font I used from the Open Font Library at `http://openfontlibrary.org/en/font/bretan`.

We convert the PIL image to a NumPy array, which allows us to use `scikit-image` to perform a shear on it. The `scikit-image` library tends to use NumPy arrays for most of its computation. The code is as follows:

```
image = np.array(im)
```

We then apply the shear transform and return the image:

```
affine_tf = tf.AffineTransform(shear=shear)
image = tf.warp(image, affine_tf)
return image / image.max()
```

In the last line, we normalize by dividing by the maximum value, ensuring our feature values are in the range 0 to 1. This normalization can happen in the data preprocessing stage, the classification stage, or somewhere else.

From here, we can now generate images quite easily and use `pyplot` to display them. First, we use our inline display for the `matplotlib` graphs and import `pyplot`. The code is as follows:

```
%matplotlib inline
from matplotlib import pyplot as plt
```

Then we create our first CAPTCHA and show it:

```
image = create_captcha("GENE", shear=0.5)
plt.imshow(image, cmap='Greys')
```

The result is the image shown at the start of this section: our CAPTCHA.

Splitting the image into individual letters

Our CAPTCHAs are words. Instead of building a classifier that can identify the thousands and thousands of possible words, we will break the problem down into a smaller problem: predicting letters.

The next step in our algorithm for beating these CAPTCHAs involves segmenting the word to discover each of the letters within it. To do this, we are going to create a function that finds contiguous sections of black pixels on the image and extract them as sub-images. These are (or at least should be) our letters.

First we import the `label` and `regionprops` functions, which we will use in this function:

```
from skimage.measure import label, regionprops
```

Our function will take an image, and return a list of subimages, where each sub-image is a letter from the original word in the image:

```
def segment_image(image):
```

The first thing we need to do is to detect where each letter is. To do this, we will use the label function in `scikit-image`, which finds connected sets of pixels that have the same value. This has analogies to our connected component discovery in *Chapter 7*, *Discovering Accounts to Follow Using Graph Mining*.

The `label` function takes an image and returns an array of the same shape as the original. However, each *connected region* has a different number in the array and pixels that are not in a connected region have the value 0. The code is as follows:

```
    labeled_image = label(image > 0)
```

We will extract each of these sub-images and place them into a list:

```
    subimages = []
```

The `scikit-image` library also contains a function for extracting information about these regions: `regionprops`. We can iterate over these regions and work on each individually:

```
    for region in regionprops(labeled_image):
```

From here, we can query the `region` object for information about the current region. For our algorithm, we need to obtain the starting and ending coordinates of the current region:

```
        start_x, start_y, end_x, end_y = region.bbox
```

We can then extract the sub-images by indexing the image (remember it is represented as a simple NumPy array, so we can easily index it) using the starting and ending positions of the sub-image, and adding the selected sub-image to our list. The code is as follows:

```
subimages.append(image[start_x:end_x,start_y:end_y])
```

Finally (and outside the loop) we return the discovered sub-images, each (hopefully) containing the section of the image with an individual letter in it. However, if we didn't find any sub-images, we just return the original image as our only sub-image. The code is as follows:

```
if len(subimages) == 0:
    return [image,]
return subimages
```

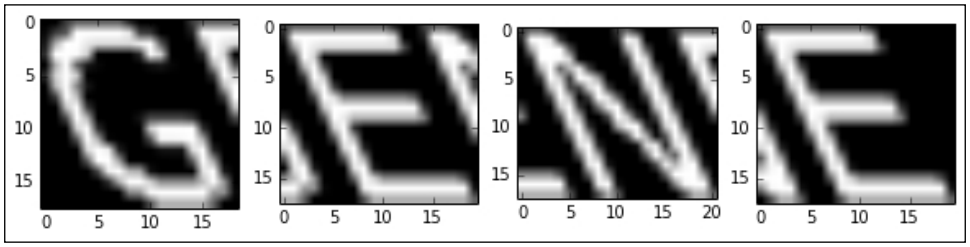
We can then get the sub-images from the example CAPTCHA using this function:

```
subimages = segment_image(image)
```

We can also view each of these sub-images:

```
f, axes = plt.subplots(1, len(subimages), figsize=(10, 3))
for i in range(len(subimages)):
    axes[i].imshow(subimages[i], cmap="gray")
```

The result will look something like this:



As you can see, our image segmentation does a reasonable job, but the results are still quite messy, with bits of previous letters showing.

Creating a training dataset

Using this function, we can now create a dataset of letters, each with different shear values. From this, we will train a neural network to recognize each letter from the image.

We first set up our random state and an array that holds the options for letters and shear values that we will randomly select from. There isn't much surprise here, but if you haven't used NumPy's `arange` function before, it is similar to Python's `range` function—except this one works with NumPy arrays and allows the step to be a float. The code is as follows:

```
from sklearn.utils import check_random_state
random_state = check_random_state(14)
letters = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
shear_values = np.arange(0, 0.5, 0.05)
```

We then create a function (for generating a single sample in our training dataset) that randomly selects a letter and a shear value from the available options. The code is as follows:

```
def generate_sample(random_state=None):
    random_state = check_random_state(random_state)
    letter = random_state.choice(letters)
    shear = random_state.choice(shear_values)
```

We then return the image of the letter, along with the target value representing the letter in the image. Our classes will be 0 for A, 1 for B, 2 for C, and so on. The code is as follows:

```
    return create_captcha(letter, shear=shear, size=(20, 20)),
      letters.index(letter)
```

Outside the function block, we can now call this code to generate a new sample and then show it using `pyplot`:

```
image, target = generate_sample(random_state)
plt.imshow(image, cmap="Greys")
print("The target for this image is: {0}".format(target))
```

We can now generate all of our dataset by calling this several thousand times. We then put the data into NumPy arrays, as they are easier to work with than lists. The code is as follows:

```
dataset, targets = zip(*(generate_sample(random_state) for i in
range(3000)))
dataset = np.array(dataset, dtype='float')
targets =  np.array(targets)
```

Our targets are integer values between 0 and 26, with each representing a letter of the alphabet. Neural networks don't usually support multiple values from a single neuron, instead preferring to have multiple outputs, each with values 0 or 1. We therefore perform one hot-encoding of the targets, giving us a target array that has 26 outputs per sample, using values near 1 if that letter is likely and near 0 otherwise. The code is as follows:

```
from sklearn.preprocessing import OneHotEncoder
onehot = OneHotEncoder()
y = onehot.fit_transform(targets.reshape(targets.shape[0],1))
```

The library we are going to use doesn't support sparse arrays, so we need to turn our sparse matrix into a dense NumPy array. The code is as follows:

```
y = y.todense()
```

Adjusting our training dataset to our methodology

Our training dataset differs from our final methodology quite significantly. Our dataset here is nicely created individual letters, fitting the 20-pixel by 20-pixel image. The methodology involves extracting the letters from words, which may squash them, move them away from the center, or create other problems.

Ideally, the data you train your classifier on should mimic the environment it will be used in. In practice, we make concessions, but aim to minimize the differences as much as possible.

For this experiment, we would ideally extract letters from actual CAPTCHAs and label those. In the interests of speeding up the process a bit, we will just run our segmentation function on the training dataset and return those letters instead.

We will need the `resize` function from `scikit-image`, as our sub-images won't always be 20 pixels by 20 pixels. The code is as follows:

```
from skimage.transform import resize
```

From here, we can run our `segment_image` function on each sample and then resize them to 20 pixels by 20 pixels. The code is as follows:

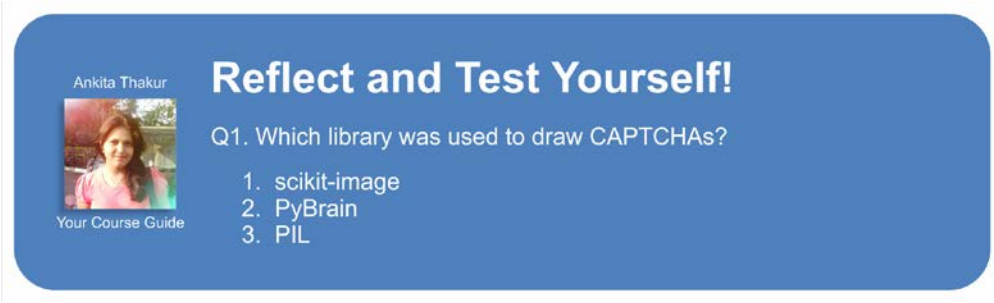
```
dataset = np.array([resize(segment_image(sample)[0], (20, 20)) for
sample in dataset])
```

Finally, we will create our dataset. This `dataset` array is three-dimensional, as it is an array of two-dimensional images. Our classifier will need a two-dimensional array, so we simply flatten the last two dimensions:

```
X = dataset.reshape((dataset.shape[0], dataset.shape[1] * dataset.
shape[2]))
```

Finally, using the `train_test_split` function of scikit-learn, we create a set of data for training and one for testing. The code is as follows:

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, train_size=0.9)
```



Training and classifying

We are now going to build a neural network that will take an image as input and try to predict which (single) letter is in the image.

We will use the training set of single letters we created earlier. The dataset itself is quite simple. We have a 20 by 20 pixel image, each pixel 1 (black) or 0 (white). These represent the 400 features that we will use as inputs into the neural network. The outputs will be 26 values between 0 and 1, where higher values indicate a higher likelihood that the associated letter (the first neuron is A, the second is B, and so on) is the letter represented by the input image.

We are going to use the PyBrain library for our neural network.

As with all the libraries we have seen so far, PyBrain can be installed from `pip`: `pip install pybrain`.

The `PyBrain` library uses its own dataset format, but luckily it isn't too difficult to create training and testing datasets using this format. The code is as follows:

```
from pybrain.datasets import SupervisedDataSet
```

First, we iterate over our training dataset and add each as a sample into a new `SupervisedDataSet` instance. The code is as follows:

```
training = SupervisedDataSet(X.shape[1], y.shape[1])
for i in range(X_train.shape[0]):
    training.addSample(X_train[i], y_train[i])
```

Then we iterate over our testing dataset and add each as a sample into a new `SupervisedDataSet` instance for testing. The code is as follows:

```
testing = SupervisedDataSet(X.shape[1], y.shape[1])
for i in range(X_test.shape[0]):
    testing.addSample(X_test[i], y_test[i])
```

Now we can build a neural network. We will create a basic three-layer network that consists of an input layer, an output layer, and a single hidden layer between them. The number of neurons in the input and output layers is fixed. 400 features in our dataset dictates that we need 400 neurons in the first layer, and 26 possible targets dictate that we need 26 output neurons.

Determining the number of neurons in the hidden layers can be quite difficult. Having too many results in a sparse network and means it is difficult to train enough neurons to properly represent the data. This usually results in overfitting the training data. If there are too few results in neurons that try to do too much of the classification each and again don't train properly, underfitting the data is the problem. I have found that creating a funnel shape, where the middle layer is between the size of the inputs and the size of the outputs, is a good starting place. For this chapter, we will use 100 neurons in the hidden layer, but playing with this value may yield better results.

We import the `buildNetwork` function and tell it to build a network based on our necessary dimensions. The first value, `X.shape[1]`, is the number of neurons in the input layer and it is set to the number of features (which is the number of columns in `X`). The second feature is our decided value of 100 neurons in the hidden layer. The third value is the number of outputs, which is based on the shape of the target array `y`. Finally, we set network to use a bias neuron to each layer (except for the output layer), effectively a neuron that always activates (but still has connections with a weight that are trained). The code is as follows:

```
from pybrain.tools.shortcuts import buildNetwork
net = buildNetwork(X.shape[1], 100, y.shape[1], bias=True)
```

From here, we can now train the network and determine good values for the weights. But how do we train a neural network?

Back propagation

The back propagation (**backprop**) algorithm is a way of assigning blame to each neuron for incorrect predictions. Starting from the output layer, we compute which neurons were incorrect in their prediction, and adjust the weights into those neurons by a small amount to attempt to fix the incorrect prediction.

These neurons made their mistake because of the neurons giving them input, but more specifically due to the weights on the connections between the neuron and its inputs. We then alter these weights by altering them by a small amount. The amount of change is based on two aspects: the partial derivative of the error function of the neuron's individual weights and the *learning rate*, which is a parameter to the algorithm (usually set at a very low value). We compute the gradient of the error of the function, multiply it by the learning rate, and subtract that from our weights. This is shown in the following example. The gradient will be positive or negative, depending on the error, and subtracting the weight will always attempt to correct the weight *towards* the correct prediction. In some cases, though, the correction will move towards something called a **local optima**, which is better than similar weights but not the best possible set of weights.

This process starts at the output layer and goes back each layer until we reach the input layer. At this point, the weights on all connections have been updated.

PyBrain contains an implementation of the backprop algorithm, which is called on the neural network through a `trainer` class. The code is as follows:

```
from pybrain.supervised.trainers import BackpropTrainer
trainer = BackpropTrainer(net, training, learningrate=0.01,
weightdecay=0.01)
```

The backprop algorithm is run iteratively using the training dataset, and each time the weights are adjusted a little. We can stop running backprop when the error reduces by a very small amount, indicating that the algorithm isn't improving the error much more and it isn't worth continuing the training. In theory, we would run the algorithm until the error doesn't change at all. This is called convergence, but in practice this takes a very long time for little gain.

Alternatively, and much more simply, we can just run the algorithm a fixed number of times, called **epochs**. The higher the number of epochs, the longer the algorithm will take and the better the results will be (with a declining improvement for each epoch). We will train for 20 epochs for this code, but trying larger values will increase the performance (if only slightly). The code is as follows:

```
trainer.trainEpochs(epochs=20)
```

After running the previous code, which may take a number of minutes depending on the hardware, we can then perform predictions of samples in our testing dataset. PyBrain contains a function for this, and it is called on the `trainer` instance:

```
predictions = trainer.testOnClassData(dataset=testing)
```

From these predictions, we can use `scikit-learn` to compute the F1 score:

```
from sklearn.metrics import f1_score
print("F-score: {0:.2f}".format(f1_score(predictions,
                                         y_test.argmax(axis=1) )))
```

The score here is 0.97, which is a great result for such a relatively simple model. Recall that our features were simple pixel values only; the neural network worked out how to use them.

Now that we have a classifier with good accuracy on letter prediction, we can start putting together words for our CAPTCHAs.

Predicting words

We want to predict each letter from each of these segments, and put those predictions together to form the predicted word from a given CAPTCHA.

Our function will accept a CAPTCHA and the trained neural network, and it will return the predicted word:

```
def predict_captcha(captcha_image, neural_network):
```

We first extract the sub-images using the `segment_image` function we created earlier:

```
    subimages = segment_image(captcha_image)
```

We will be building our word from each of the letters. The sub-images are ordered according to their location, so usually this will place the letters in the correct order:

```
    predicted_word = ""
```

Next we iterate over the sub-images:

```
    for subimage in subimages:
```

Each sub-image is unlikely to be exactly 20 pixels by 20 pixels, so we will need to resize it in order to have the correct size for our neural network.

```
        subimage = resize(subimage, (20, 20))
```

We will activate our neural network by sending the sub-image data into the input layer. This propagates through our neural network and returns the given output. All this happened in our testing of the neural network earlier, but we didn't have to explicitly call it. The code is as follows:

```
        outputs = net.activate(subimage.flatten())
```

The output of the neural network is 26 numbers, each relative to the likelihood that the letter at the given index is the predicted letter. To get the actual prediction, we get the index of the maximum value of these outputs and look up our letters list from before for the actual letter. For example, if the value is highest for the fifth output, the predicted letter will be *E*. The code is as follows:

```
        prediction = np.argmax(outputs)
```

We then append the predicted letter to the predicted word we are building:

```
        predicted_word += letters[prediction]
```

After the loop completes, we have gone through each of the letters and formed our predicted word:

```
    return predicted_word
```

We can now test on a word using the following code. Try different words and see what sorts of errors you get, but keep in mind that our neural network only knows about capital letters.

```
word = "GENE"
captcha = create_captcha(word, shear=0.2)
print(predict_captcha(captcha, net))
```

We can codify this into a function, allowing us to perform predictions more easily. We also leverage our assumption that the words will be only four-characters long to make prediction a little easier. Try it without the `prediction = prediction[:4]` line and see what types of errors you get. The code is as follows:

```
def test_prediction(word, net, shear=0.2):
    captcha = create_captcha(word, shear=shear)
```

```
    prediction = predict_captcha(captcha, net)
    prediction = prediction[:4]
    return word == prediction, word, prediction
```

The returned results specify whether the prediction is correct, the original word, and the predicted word.

This code correctly predicts the word *GENE*, but makes mistakes with other words. How accurate is it? To test, we will create a dataset with a whole bunch of four-letter English words from NLTK. The code is as follows:

```
from nltk.corpus import words
```

The `words` instance here is actually a corpus object, so we need to call `words()` on it to extract the individual words from this corpus. We also filter to get only four-letter words from this list. The code is as follows:

```
valid_words = [word.upper() for word in words.words() if len(word) ==
4]
```

We can then iterate over all of the words to see how many we get correct by simply counting the correct and incorrect predictions:

```
num_correct = 0
num_incorrect = 0
for word in valid_words:
    correct, word, prediction = test_prediction(word, net,
                                                shear=0.2)
if correct:
        num_correct += 1
    else:
        num_incorrect += 1
print("Number correct is {0}".format(num_correct))
print("Number incorrect is {0}".format(num_incorrect))
```

The results we get are 2,832 correct and 2,681 incorrect for an accuracy of just over 51 percent. From our original 97 percent per-letter accuracy, this is a big decline. What happened?

The first factor to impact is our accuracy. All other things being equal, if we have four letters, and 97 percent accuracy per-letter, then we can expect about an 88 percent success rate (all other things being equal) getting four letters in a row ($0.88 \approx 0.974$). A single error in a single letter's prediction results in the wrong word being predicted.

The second impact is the shear value. Our dataset chose randomly between shear values of 0 to 0.5. The previous test used a shear of 0.2. For a value of 0, I get 75 percent accuracy; for a shear of 0.5, the result is much worse at 2.5 percent. The higher the shear, the lower the performance.

The next impact is that our letters were randomly chosen for the dataset. In reality, this is not true at all. Letters, such as E, appear much more frequently than other letters, such as Q. Letters that appear reasonably commonly but are frequently mistaken for each other, will also contribute to the error.

We can table which letters are frequently mistaken for each other using a confusion matrix, which is a two-dimensional array. Its rows and columns each represent an individual class.

Each cell represents the number of times that a sample is actually from one class (represented by the row) and predicted to be in the second class (represented by the column). For example, if the value of the cell `(4,2)` is 6, it means that there were six cases where a sample with the letter D was predicted as being a letter B.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(np.argmax(y_test, axis=1), predictions)
```

Ideally, a confusion matrix should only have values along the diagonal. The cells `(i, i)` have values, but any other cell has a value of zero. This indicates that the predicted classes are exactly the same as the actual classes. Values that aren't on the diagonal represent errors in the classification.

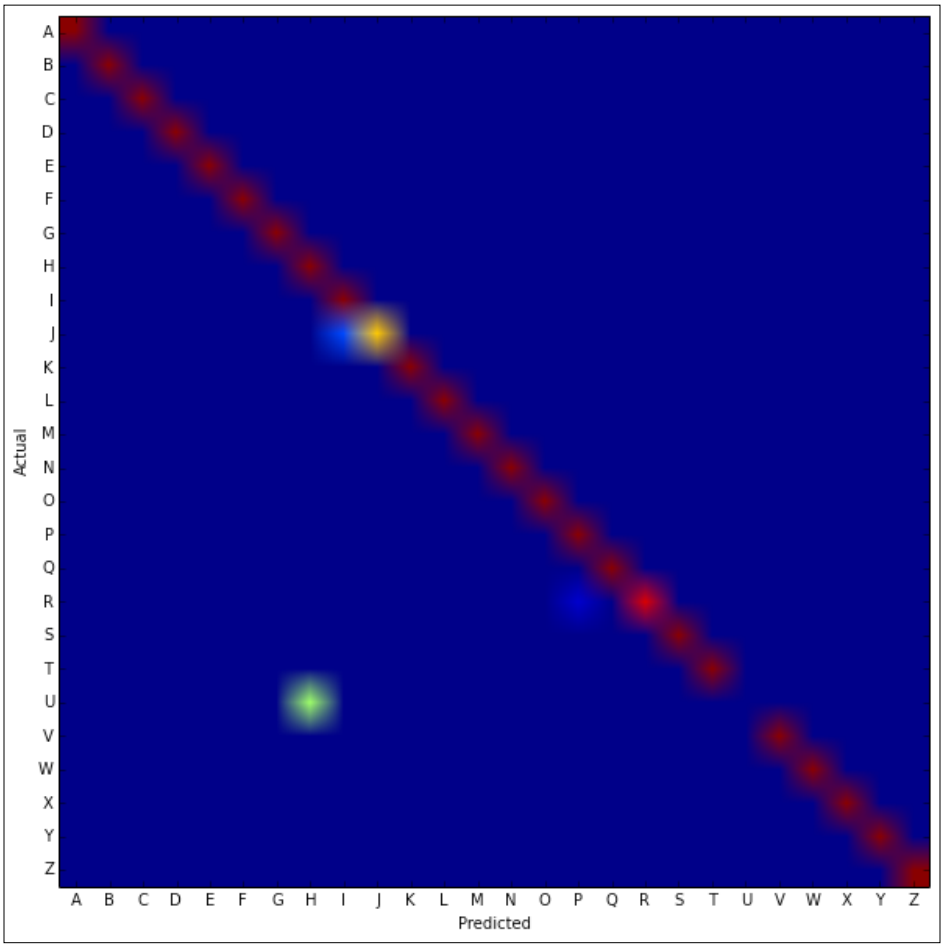
We can also plot this using `pyplot`, showing graphically which letters are confused with each other. The code is as follows:

```
plt.figure(figsize=(10, 10))
plt.imshow(cm)
```

We set the axis and tick marks to easily reference the letters each index corresponds to:

```
tick_marks = np.arange(len(letters))
plt.xticks(tick_marks, letters)
plt.yticks(tick_marks, letters)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

The result is shown in the next graph. It can be quite clearly seen that the main source of error is U being mistaken for an H nearly every single time!



The letter U appears in 17 percent of words in our list. For each word that a U appears in, we can expect this to be wrong. U actually appears more often than H (which is in around 11 percent of words), indicating we could get a cheap (although possibly not a robust) boost in accuracy by changing any H prediction into a U.

In the next section, we will do something a bit smarter and actually use the dictionary to search for similar words.

Improving accuracy using a dictionary

Rather than just returning the given prediction, we can check whether the word actually exists in our dictionary. If it does, then that is our prediction. If it isn't in the dictionary, we can try and find a word that is similar to it and predict that instead. Note that this strategy relies on our assumption that all CAPTCHA words will be valid English words, and therefore this strategy wouldn't work for a random sequence of characters. This is one reason why some CAPTCHAs don't use words.

There is one issue here—how do we determine the closest word? There are many ways to do this. For instance, we can compare the lengths of words. Two words that have a similar length could be considered more similar. However, we commonly consider words to be similar if they have the same letters in the same positions. This is where the edit distance comes in.

Ranking mechanisms for words

The **Levenshtein edit distance** is a commonly used method for comparing two short strings to see how similar they are. It isn't very scalable, so it isn't commonly used for very long strings. The edit distance computes the number of steps it takes to go from one word to another. The steps can be one of the following three actions:

1. Insert a new letter into the word at any position.
2. Delete any letter from the word.
3. Substitute a letter for another one.

The minimum number of actions needed to transform the first word into the second is given as the distance. Higher values indicate that the words are less similar.

This distance is available in NLTK as `nltk.metrics.edit_distance`. We can call it using on two strings and it returns the edit distance:

```
from nltk.metrics import edit_distance
steps = edit_distance("STEP", "STOP")
print("The number of steps needed is: {0}".format(steps))
```

When used with different words, the edit distance is quite a good approximation to what many people would intuitively feel are similar words. The edit distance is great for testing spelling mistakes, dictation errors, and name matching (where you can mix up your Marc and Mark spelling quite easily).

However, it isn't very good. We don't really expect letters to be moved around, just individual letter comparisons to be wrong. For this reason, we will create a different distance metric, which is simply the number of letters in the same positions that are incorrect. The code is as follows:

```
def compute_distance(prediction, word):
    return len(prediction) - sum(prediction[i] == word[i] for i in
range(len(prediction)))
```

We subtract the value from the length of the prediction word (which is four) to make it a distance metric where lower values indicate more similarity between the words.

Putting it all together

We can now test our improved prediction function using similar code to before. First we define a prediction, which also takes our list of valid words:

```
from operator import itemgetter
def improved_prediction(word, net, dictionary, shear=0.2):
    captcha = create_captcha(word, shear=shear)
    prediction = predict_captcha(captcha, net)
    prediction = prediction[:4]
```

Up to this point, the code is as before. We do our prediction and limit it to the first four characters. However, we now check if the word is in the dictionary or not. If it is, we return that as our prediction. If it is not, we find the next nearest word. The code is as follows:

```
    if prediction not in dictionary:
```

We compute the distance between our predicted word and each other word in the dictionary, and sort it by distance (lowest first). The code is as follows:

```
        distances = sorted([(word, compute_distance(prediction, word))
                            for word in dictionary],
key=itemgetter(1))
```

We then get the best matching word—that is, the one with the lowest distance—and predict that word:

```
        best_word = distances[0]
        prediction = best_word[0]
```

We then return the correctness, word, and prediction as before:

```
    return word == prediction, word, prediction
```

The changes in our testing code are highlighted in the following code:

```
num_correct = 0
num_incorrect = 0
for word in valid_words:
    correct, word, prediction = improved_prediction(word, net, valid_
words, shear=0.2)
    if correct:
        num_correct += 1
    else:
        num_incorrect += 1
print("Number correct is {0}".format(num_correct))
print("Number incorrect is {0}".format(num_incorrect))
```

The preceding code will take a while to run (computing all of the distances will take some time) but the net result is 3,037 samples correct and 2,476 samples incorrect. This is an accuracy of 55 percent for a boost of 4 percentage points. The reason this improvement is so low is that multiple words all have the same similarity, and the algorithm is choosing the *best* one randomly between this set of most similar words. For example, the first word in the list, AANI (I just chose the first word in the list, which is a dog-headed ape from Egyptian mythology), has 44 candidate words that are all the same distance from the word. This gives just a 1/44 chance of choosing the correct word from this list.

If we were to cheat and count the prediction as correct if the actual word was any one of the best candidates, we would rate 78 percent of predictions as correct (to see this code, check out the code in the bundle).

To further improve the results, we can work on our distance metric, perhaps using information from our confusion matrix to find *commonly confused letters* or some other improvement upon this. This iterative improvement is a feature of many data mining methodologies, and it mimics the scientific method—have an idea, test it out, analyze the results, and use that to improve the next idea.

Your Coding Challenge

Ankita Thakur

Your Course Guide

The CAPTCHAs we beat in this example were not as complex as those normally used today. You can create more complex variants using a number of techniques as follows:

- Applying different transformations such as the ones in scikit-image (refer to `http://scikit-image.org/docs/dev/auto_examples/applications/plot_geometric.html`)
- Using different colors and colors that don't translate well to grayscale
- Adding lines or other shapes to the image: `http://scikit-image.org/docs/dev/api/skimage.draw.html`

Summary of Module 3 Chapter 8

Ankita Thakur

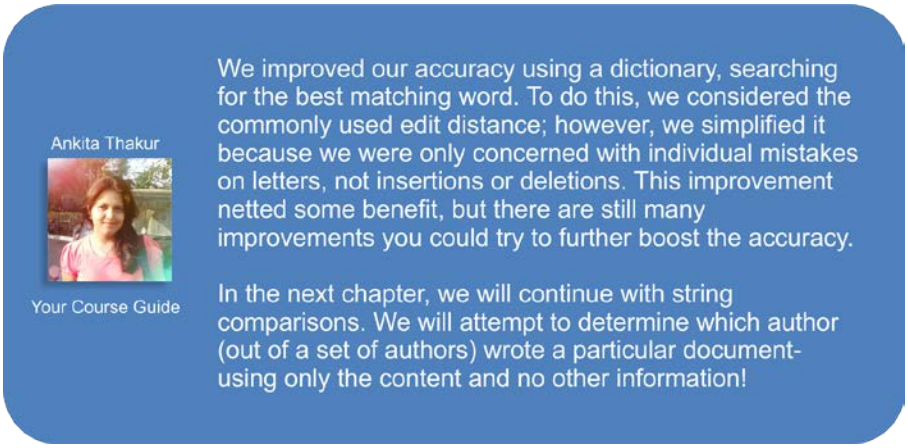
Your Course Guide

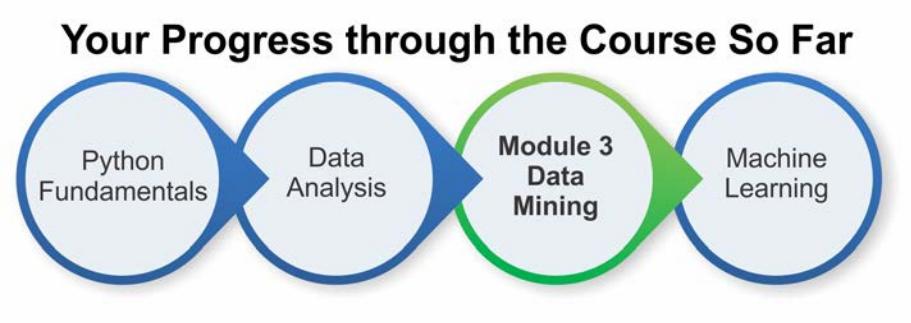
In this chapter, we worked with images in order to use simple pixel values to predict the letter being portrayed in a CAPTCHA. Our CAPTCHAs were a bit simplified; we only used complete four-letter English words. In practice, the problem is much harder—as it should be! With some improvements, it would be possible to solve much harder CAPTCHAs with neural networks and a methodology similar to what we discussed. The scikit-image library contains lots of useful functions for extracting shapes from images, functions for improving contrast, and other image tools that will help.

We took our larger problem of predicting words, and created a smaller and simple problem of predicting letters. From here, we were able to create a feed-forward neural network to accurately predict which letter was in the image. At this stage, our results were very good with 97 percent accuracy.

Neural networks are simply connected sets of neurons, which are basic computation devices consisting of a single function. However, when you connect these together, they can solve incredibly complex problems. Neural networks are the basis for deep learning, which is one of the most effective areas of data mining at the moment.

Despite our great per-letter accuracy, the performance when predicting a word drops to just over 50 percent when trying to predict a whole word. There were several factors for this, representing the difficulty of taking a problem from an experiment to the real world.


Ankita Thakur
Your Course Guide
We improved our accuracy using a dictionary, searching for the best matching word. To do this, we considered the commonly used edit distance; however, we simplified it because we were only concerned with individual mistakes on letters, not insertions or deletions. This improvement netted some benefit, but there are still many improvements you could try to further boost the accuracy.
In the next chapter, we will continue with string comparisons. We will attempt to determine which author (out of a set of authors) wrote a particular document-using only the content and no other information!


Your Progress through the Course So Far
Python Fundamentals
Data Analysis
Module 3 Data Mining
Machine Learning

9
Authorship Attribution

Authorship analysis is, predominately, a text mining task that aims to identify certain aspects about an author, based only on the content of their writings. This could include characteristics such as age, gender, or background. In the specific **authorship attribution** task, we aim to identify who out of a set of authors wrote a particular document. This is a classic case of a classification task. In many ways, authorship analysis tasks are performed using standard data mining methodologies, such as cross fold validation, feature extraction, and classification algorithms.

In this chapter, we will use the problem of authorship attribution to piece together the parts of the data mining methodology we developed in the previous chapters. We identify the problem and discuss the background and knowledge of the problem. This lets us choose features to extract, which we will build a pipeline for achieving. We will test two different types of features: function words and character n-grams. Finally, we will perform an in-depth analysis of the results. We will work with a book dataset, and then a very messy real-world corpus of e-mails.

The topics we will cover in this chapter are as follows:

- Feature engineering and how the features differ based on application
- Revisiting the bag-of-words model with a specific goal in mind
- Feature types and the character n-grams model
- Support vector machines
- Cleaning up a messy dataset for data mining

Attributing documents to authors

Authorship analysis has a background in **stylometry**, which is the study of an author's style of writing. The concept is based on the idea that everyone learns language slightly differently, and measuring the nuances in people's writing will enable us to tell them apart using only the content of their writing.

The problem has been historically performed using manual analysis and statistics, which is a good indication that it could be automated with data mining. Modern authorship analysis studies are almost entirely data mining-based, although quite a significant amount of work is still done with more manually driven analysis using linguistic styles.

Authorship analysis has many subproblems, and the main ones are as follows:

- **Authorship profiling**: This determines the age, gender, or other traits of the author based on the writing. For example, we can detect the first language of a person speaking English by looking for specific ways in which they speak the language.
- **Authorship verification**: This checks *whether the author of this document also wrote the other document*. This problem is what you would normally think about in a legal court setting. For instance, the suspect's writing style (content-wise) would be analyzed to see if it matched the ransom note.
- **Authorship clustering**: This is an extension of authorship verification, where we use cluster analysis to group documents from a big set into clusters, and each cluster is written by the same author.

However, the most common form of authorship analysis study is that of authorship attribution, a classification task where we attempt to predict which of a set of authors wrote a given document.

Applications and use cases

Authorship analysis has a number of use cases. Many use cases are concerned with problems such as verifying authorship, proving shared authorship/provenance, or linking social media profiles with real-world users.

In a historical sense, we can use authorship analysis to verify whether certain documents were indeed written by their supposed authors. Controversial authorship claims include some of Shakespeare's plays, the Federalist papers from the USA's foundation period, and other historical texts.

Authorship studies alone cannot prove authorship, but can provide evidence for or against a given theory. For example, we can analyze Shakespeare's plays to determine his writing style, before testing whether a given sonnet actually does originate from him.

A more modern use case is that of linking social network accounts. For example, a malicious online user could set up accounts on multiple online social networks. Being able to link them allows authorities to track down the user of a given account—for example, if it is harassing other online users.

Another example used in the past is to be a backbone to provide expert testimony in court to determine whether a given person wrote a document. For instance, the suspect could be accused of writing an e-mail harassing another person. The use of authorship analysis could determine whether it is likely that person did in fact write the document. Another court-based use is to settle claims of stolen authorship. For example, two authors may claim to have written a book, and authorship analysis could provide evidence on which is the likely author.

Authorship analysis is not foolproof though. A recent study found that attributing documents to authors can be made considerably harder by simply asking people, who are otherwise untrained, to hide their writing style. This study also looked at a framing exercise where people were asked to write in the style of another person. This framing of another person proved quite reliable, with the faked document commonly attributed to the person being framed.

Despite these issues, authorship analysis is proving useful in a growing number of areas and is an interesting data mining problem to investigate.

Attributing authorship

Authorship attribution is a classification task by which we have a set of candidate authors, a set of documents from each of those authors (the training set), and a set of documents of unknown authorship (the test set). If the documents of unknown authorship definitely belong to one of the candidates, we call this a **closed problem**.

If we cannot be sure of that, we call this an open problem. This distinction isn't just specific to authorship attribution though—any data mining application where the actual class may not be in the training set is considered an open problem, with the task being to find the candidate author or to select none of them.

In authorship attribution, we typically have two restrictions on the tasks. First, we only use content information from the documents and not metadata about time of writing, delivery, handwriting style, and so on. There are ways to combine models from these different types of information, but that isn't generally considered authorship attribution and is more a data fusion application.

The second restriction is that we don't look at the topic of the documents; instead, we look for more salient features such as word usage, punctuation, and other text-based features. The reasoning here is that a person can write on many different topics, so worrying about the topic of their writing isn't going to model their actual authorship style. Looking at topic words can also lead to overfitting on the training data—our model may train on documents from the same author and also on the same topic. For instance, if you were to model my authorship style by looking at this module, you might conclude the words *data mining* are indicative of my style, when in fact I write on other topics as well.

From here, the pipeline for performing authorship attribution looks a lot like the one we developed in *Chapter 6*, *Social Media Insight Using Naive Bayes*. First, we extract features from our text. Then, we perform some feature selection on those features. Finally, we train a classification algorithm to fit a model, which we can then use to predict the class (in this case, the author) of a document.

There are some differences, mostly having to do with which features are used, that we will cover in this chapter. But first, we will define the scope of the problem.

Getting the data

The data we will use for this chapter is a set of books from Project Gutenberg at `www.gutenberg.org`, which is a repository of public domain literature works. The books I used for these experiments come from a variety of authors:

- Booth Tarkington (22 titles)
- Charles Dickens (44 titles)
- Edith Nesbit (10 titles)
- Arthur Conan Doyle (51 titles)
- Mark Twain (29 titles)
- Sir Richard Francis Burton (11 titles)
- Emile Gaboriau (10 titles)

Overall, there are 177 documents from 7 authors, giving a significant amount of text to work with. A full list of the titles, along with download links and a script to automatically fetch them, is given in the code bundle.

To download these books, we use the requests library to download the files into our data directory. First, set up the data directory and ensure the following code links to it:

```
import os
import sys
data_folder = os.path.join(os.path.expanduser("~"), "Data", "books")
```

Next, run the script from the code bundle to download each of the books from Project Gutenberg. This will place them in the appropriate subfolders of this data folder.

To run the script, download the `getdata.py` script from the `Chapter 9` folder in the code bundle. Save it to your `notebooks` folder and enter the following into a new cell:

```
!load getdata.py
```

Then, from inside your IPython Notebook, press *Shift* + *Enter* to run the cell. This will load the script into the cell. Then click the code again and press *Shift* + *Enter* to run the script itself. This will take a while, but it will print a message to let you know it is complete.

After taking a look at these files, you will see that many of them are quite messy—at least from a data analysis point of view. There is a large project Gutenberg disclaimer at the start of the files. This needs to be removed before we do our analysis.

We could alter the individual files on disk to remove this stuff. However, what happens if we were to lose our data? We would lose our changes and potentially be unable to replicate the study. For that reason, we will perform the preprocessing as we load the files—this allows us to be sure our results will be replicable (as long as the data source stays the same). The code is as follows:

```
def clean_book(document):
```

We first split the document into lines, as we can identify the start and end of the disclaimer by the starting and ending lines:

```
    lines = document.split("\n")
```

We are going to iterate through each line. We look for the line that indicates the start of the book, and the line that indicates the end of the book. We will then take the text in between as the book itself. The code is as follows:

```
    start = 0
    end = len(lines)
    for i in range(len(lines)):
        line = lines[i]
        if line.startswith("*** START OF THIS PROJECT GUTENBERG"):
            start = i + 1
        elif line.startswith("*** END OF THIS PROJECT GUTENBERG"):
            end = i - 1
```

Finally, we join those lines together with a newline character to recreate the book without the disclaimers:

```
    return "\n".join(lines[start:end])
```

From here, we can now create a function that loads all of the books, performs the preprocessing, and returns them along with a class number for each author. The code is as follows:

```
import numpy as np
```

By default, our function signature takes the parent folder containing each of the subfolders that contain the actual books. The code is as follows:

```
def load_books_data(folder=data_folder):
```

We create lists for storing the documents themselves and the author classes:

```
documents = []
authors = []
```

We then create a list of each of the subfolders in the parent directly, as the script creates a subfolder for each author. The code is as follows:

```
subfolders = [subfolder for subfolder in os.listdir(folder)
              if os.path.isdir(os.path.join(folder,
                 subfolder))]
```

Next we iterate over these subfolders, assigning each subfolder a number using `enumerate`:

```
for author_number, subfolder in enumerate(subfolders):
```

We then create the full subfolder path and look for all documents within that subfolder:

```
    full_subfolder_path = os.path.join(folder, subfolder)
    for document_name in os.listdir(full_subfolder_path):
```

For each of those files, we open it, read the contents, preprocess those contents, and append it to our documents list. The code is as follows:

```
        with open(os.path.join(full_subfolder_path,
          document_name)) as inf:
            documents.append(clean_book(inf.read()))
```

We also append the number we assigned to this author to our authors list, which will form our classes:

```
            authors.append(author_number)
```

We then return the documents and classes (which we transform into a NumPy array for each indexing later on):

```
return documents, np.array(authors, dtype='int')
```

We can now get our documents and classes using the following function call:

```
documents, classes = load_books_data(data_folder)
```

This dataset fits into memory quite easily, so we can load all of the text at once. In cases where the whole dataset doesn't fit, a better solution is to extract the features from each document one-at-a-time (or in batches) and save the resulting values to a file or in-memory matrix.

Function words

One of the earliest types of features, and one that still works quite well for authorship analysis, is to use function words in a bag-of-words model. Function words are words that have little meaning on their own, but are required for creating (English) sentences. For example, the words *this* and *which* are words that are really only defined by what they do within a sentence, rather than their meaning in themselves. Contrast this with a content word such as *tiger*, which has an explicit meaning and invokes imagery of a large cat when used in a sentence.

Function words are not always clearly clarified. A good rule of thumb is to choose the most frequent words in usage (over all possible documents, not just ones from the same author). Typically, the more frequently a word is used, the better it is for authorship analysis. In contrast, the less frequently a word is used, the better it is for content-based text mining, such as in the next chapter, where we look at the topic of different documents.

The use of function words is less defined by the content of the document and more by the decisions made by the author. This makes them good candidates for separating the authorship traits between different users. For instance, while many Americans are particular about the different in usage between *that* and *which* in a sentence, people from other countries, such as Australia, are less particular about this. This means that some Australians will lean towards almost exclusively using one word or the other, while others may use *which* much more. This difference, combined with thousands of other nuanced differences, makes a model of authorship.

Counting function words

We can count function words using the `CountVectorizer` class we used in *Chapter 6*, *Social Media Insight Using Naive Bayes*. This class can be passed a **vocabulary**, which is the set of words it will look for. If a vocabulary is not passed (we didn't pass one in the code of `Chapter 6`), then it will learn this vocabulary from the dataset. All the words are in the training set of documents (depending on the other parameters of course).

First, we set up our vocabulary of function words, which is just a list containing each of them. Exactly which words are function words and which are not is up for debate. I've found this list, from published research, to be quite good:

```
function_words = ["a", "able", "aboard", "about", "above", "absent",
"according" , "accordingly", "across", "after", "against",
"ahead", "albeit", "all", "along", "alongside", "although",
"am", "amid", "amidst", "among", "amongst", "amount", "an",
"and", "another", "anti", "any", "anybody", "anyone",
"anything", "are", "around", "as", "aside", "astraddle",
"astride", "at", "away", "bar", "barring", "be", "because",
"been", "before", "behind", "being", "below", "beneath",
"beside", "besides", "better", "between", "beyond", "bit",
"both", "but", "by", "can", "certain", "circa", "close",
"concerning", "consequently", "considering", "could",
"couple", "dare", "deal", "despite", "down", "due", "during",
"each", "eight", "eighth", "either", "enough", "every",
"everybody", "everyone", "everything", "except", "excepting",
"excluding", "failing", "few", "fewer", "fifth", "first",
"five", "following", "for", "four", "fourth", "from", "front",
"given", "good", "great", "had", "half", "have", "he",
"heaps", "hence", "her", "hers", "herself", "him", "himself",
"his", "however", "i", "if", "in", "including", "inside",
"instead", "into", "is", "it", "its", "itself", "keeping",
"lack", "less", "like", "little", "loads", "lots", "majority",
"many", "masses", "may", "me", "might", "mine", "minority",
"minus", "more", "most", "much", "must", "my", "myself",
"near", "need", "neither", "nevertheless", "next", "nine",
"ninth", "no", "nobody", "none", "nor", "nothing",
"notwithstanding", "number", "numbers", "of", "off", "on",
"once", "one", "onto", "opposite", "or", "other", "ought",
"our", "ours", "ourselves", "out", "outside", "over", "part",
"past", "pending", "per", "pertaining", "place", "plenty",
"plethora", "plus", "quantities", "quantity", "quarter",
"regarding", "remainder", "respecting", "rest", "round",
```

```
"save", "saving", "second", "seven", "seventh", "several",
"shall", "she", "should", "similar", "since", "six", "sixth",
"so", "some", "somebody", "someone", "something", "spite",
"such", "ten", "tenth", "than", "thanks", "that", "the",
"their", "theirs", "them", "themselves", "then", "thence",
"therefore", "these", "they", "third", "this", "those",
"though", "three", "through", "throughout", "thru", "thus",
"till", "time", "to", "tons", "top", "toward", "towards",
"two", "under", "underneath", "unless", "unlike", "until",
"unto", "up", "upon", "us", "used", "various", "versus",
"via", "view", "wanting", "was", "we", "were", "what",
"whatever", "when", "whenever", "where", "whereas",
"wherever", "whether", "which", "whichever", "while",
"whilst", "who", "whoever", "whole", "whom", "whomever",
"whose", "will", "with", "within", "without", "would", "yet",
"you", "your", "yours", "yourself", "yourselves"]
```

Now, we can set up an extractor to get the counts of these function words. We will fit this using a pipeline later:

```
from sklearn.feature_extraction.text import CountVectorizer
extractor = CountVectorizer(vocabulary=function_words)
```

Classifying with function words

Next, we import our classes. The only new thing here is the support vector machines, which we will cover in the next section (for now, just consider it a standard classification algorithm). We import the SVC class, an SVM for classification, as well as the other standard workflow tools we have seen before:

```
from sklearn.svm import SVC
from sklearn.cross_validation import cross_val_score
from sklearn.pipeline import Pipeline
from sklearn import grid_search
```

Support vector machines take a number of parameters. As I said, we will use one blindly here, before going into detail in the next section. We then use a dictionary to set which parameters we are going to search. For the `kernel` parameter, we will try `linear` and `rbf`. For `C`, we will try values of 1 and 10 (descriptions of these parameters are covered in the next section). We then create a grid search to search these parameters for the best choices:

```
parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
svr = SVC()
grid = grid_search.GridSearchCV(svr, parameters)
```

Gaussian kernels (such as `rbf`) only work for reasonably sized datasets, such as when the number of features is fewer than about 10,000.

Next, we set up a pipeline that takes the feature extraction step using the `CountVectorizer` (only using function words), along with our grid search using SVM. The code is as follows:

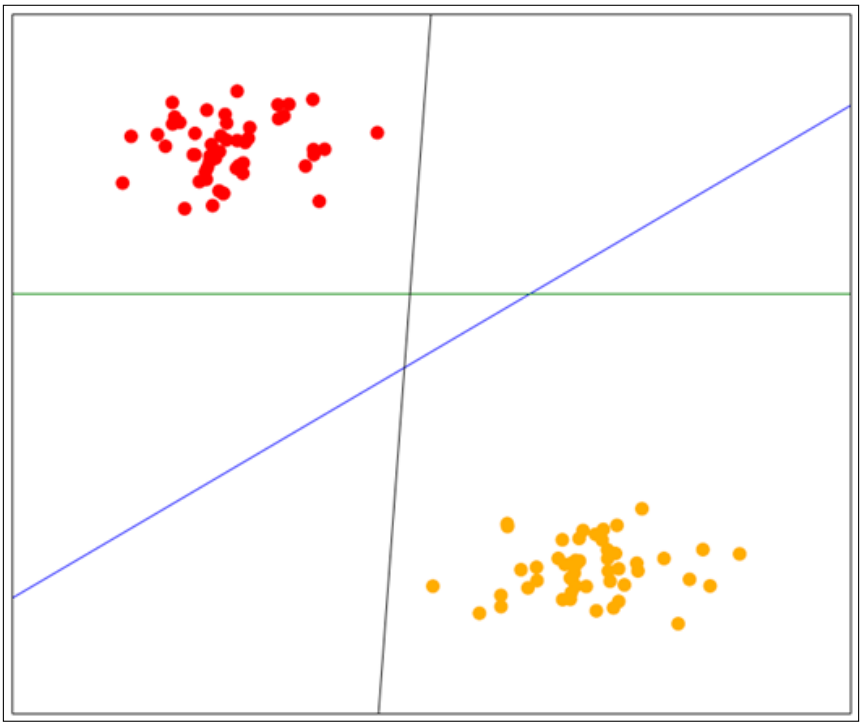
```
pipeline1 = Pipeline([('feature_extraction', extractor),
                      ('clf', grid)
                      ])
```

Next, we apply `cross_val_score` to get our cross validated score for this pipeline. The result is 0.811, which means we approximately get 80 percent of the predictions correct. For 7 authors, this is a good result!

Support vector machines

Support vector machines (**SVMs**) are classification algorithms based on a simple and intuitive idea. It performs classification between only two classes (although we can extend it to more classes). Suppose that our two classes can be separated by a line such that any points above the line belong to one class and any below the line belong to the other class. SVMs find this line and use it for prediction, much the same way as linear regression works. SVMs, however, find the best *line* for separating the dataset.

In the following figure, we have three lines that separate the dataset: blue, black, and green. Which would you say is the best option?



Intuitively, a person would normally choose the blue line as the *best* option, as this separates the data the most. That is, it has the maximum distance from any point in each class.

Finding this line is an optimization problem, based on finding the lines of margin with the maximum distance between them.

The derivation of these equations is outside the scope of this module, but I recommend interested readers to go through the derivations at `http://en.wikibooks.org/wiki/Support_Vector_Machines` for the details. Alternatively, you can visit `http://docs.opencv.org/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html`.

Classifying with SVMs

After training the model, we have a line of maximum margin. The classification of new samples is then simply asking the question: *does it fall above the line, or below it?* If it falls above the line, it is predicted as one class. If it is below the line, it is predicted as the other class.

For multiple classes, we create multiple SVMs—each a binary classifier. We then connect them using any one of a variety of strategies. A basic strategy is to create a **one-versus-all** classifier for each class, where we train using two classes—the given class and all other samples. We do this for each class and run each classifier on a new sample, choosing the best match from each of these. This process is performed automatically in most SVM implementations.

We saw two parameters in our previous code: `C` and the `kernel`. We will cover the `kernel` parameter in the next section, but the `C` parameter is an important parameter for fitting SVMs. The `C` parameter relates to how much the classifier should aim to predict all training samples correctly, at the risk of overfitting. Selecting a higher `C` value will find a line of separation with a smaller margin, aiming to classify all training samples correctly. Choosing a lower `C` value will result in a line of separation with a larger margin—even if that means that some training samples are incorrectly classified. In this case, a lower `C` value presents a lower chance of overfitting, at the risk of choosing a generally poorer line of separation.

One limitation with SVMs (in their basic form) is that they only separate data that is linearly separable. What happens if the data isn't? For that problem, we use kernels.

Kernels

When the data cannot be separated linearly, the trick is to embed it onto a higher dimensional space. What this means, with a lot of hand-waving about the details, is to add pseudo-features until the data is linearly separable (which will always happen if you add enough of the right kinds of features).

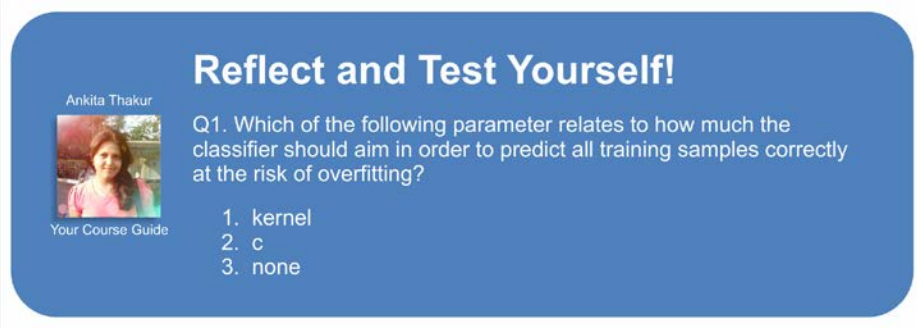
The trick is that we often compute the inner-produce of the samples when finding the best line to separate the dataset. Given a function that uses the dot product, we effectively manufacture new features without having to actually define those new features. This is handy because we don't know what those features were going to be anyway. We now define a `kernel` as a function that itself is the dot product of the function of two samples from the dataset, rather than based on the samples (and the made-up features) themselves.

We can now compute what that dot product is (or approximate it) and then just use that.

There are a number of kernels in common use. The `linear` kernel is the most straightforward and is simply the dot product of the two sample feature vectors, the weight feature, and a bias value. There is also a polynomial kernel, which raises the dot product to a given degree (for instance, 2). Others include the Gaussian (`rbf`) and Sigmoidal functions. In our previous code sample, we tested between the `linear` kernel and the `rbf` kernels.

The end result from all this derivation is that these kernels effectively define a distance between two samples that is used in the classification of new samples in SVMs. In theory, any distance could be used, although it may not share the same characteristics that enable easy optimization of the SVM training.

In scikit-learn's implementation of SVMs, we can define the `kernel` parameter to change which kernel function is used in computations, as we saw in the previous code sample.



Character n-grams

We saw how function words can be used as features to predict the author of a document. Another feature type is **character n-grams**. An n-gram is a sequence of `n` objects, where `n` is a value (for text, generally between 2 and 6). *Word n-grams* have been used in many studies, usually relating to the topic of the documents. However, character n-grams have proven to be of high quality for authorship attribution.

Character n-grams are found in text documents by representing the document as a sequence of characters. These n-grams are then extracted from this sequence and a model is trained. There are a number of different models for this, but a standard one is very similar to the bag-of-words model we have used earlier.

For each distinct n-gram in the training corpus, we create a feature for it. An example of an n-gram is `<e t>`, which is the letter *e*, a space, and then the letter *t* (the angle brackets are used to denote the start and end of the n-gram and aren't part of it). We then train our model using the frequency of each n-gram in the training documents and train the classifier using the created feature matrix.

Character n-grams are defined in many ways. For instance, some applications only choose within-word characters, ignoring whitespace and punctuation. Some use this information (like our implementation in this chapter).

A common theory for why character n-grams work is that people more typically write words they can easily say and character n-grams (at least when n is between 2 and 6) are a good approximation for phonemes—the sounds we make when saying words. In this sense, using character n-grams approximates the sounds of words, which approximates your writing style. This is a common pattern when creating new features. First we have a theory on what concepts will impact the end result (authorship style) and then create features to approximate or measure those concepts.

A main feature of a character n-gram matrix is that it is sparse and increases in sparsity with higher n-values quite quickly. For an *n*-value of 2, approximately 75 percent of our feature matrix is zeros. For an *n*-value of 5, over 93 percent is zeros. This is typically less sparse than a word n-gram matrix of the same type though and shouldn't cause many issues using a classifier that is used for word-based classifications.

Extracting character n-grams

We are going to use our `CountVectorizer` class to extract character n-grams. To do that, we set the `analyzer` parameter and specify a value for *n* to extract n-grams with.

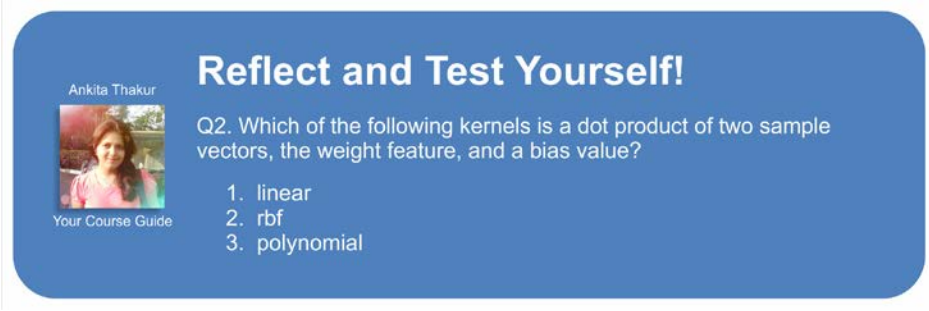
The implementation in scikit-learn uses an n-gram *range*, allowing you to extract n-grams of multiple sizes at the same time. We won't delve into different *n*-values in this experiment, so we just set the values the same. To extract n-grams of size 3, you need to specify (3, 3) as the value for the n-gram range.

We can reuse the grid search from our previous code. All we need to do is specify the new feature extractor in a new pipeline:

```
pipeline = Pipeline([('feature_extraction', CountVectorizer(analyzer='
char', ngram_range=(3, 3))),
                     ('classifier', grid)
```

```
                        ])
scores = cross_val_score(pipeline, documents, classes, scoring='f1')
print("Score: {:.3f}".format(np.mean(scores)))
```

There is a lot of implicit overlap between function words and character n-grams, as character sequences in function words are more likely to appear. However, the actual features are very different and character n-grams capture punctuation, which function words do not. For example, a character n-gram includes the full stop at the end of a sentence, while a function word-based method would only use the preceding word itself.



Using the Enron dataset

Enron was one of the largest energy companies in the world in the late 1990s, reporting revenue over $100 billion. It has over 20,000 staff and—as of the year 2000—there seemed to be no indications that something was very wrong.

In 2001, the *Enron Scandal* occurred, where it was discovered that Enron was undertaking systematic, fraudulent accounting practices. This fraud was deliberate, wide-ranging across the company, and for significant amounts of money. After this was publicly discovered, its share price dropped from more than $90 in 2000 to less than $1 in 2001. Enron shortly filed for bankruptcy in a mess that would take more than 5 years to finally be resolved.

As part of the investigation into Enron, the Federal Energy Regulatory Commission in the United States made more than 600,000 e-mails publicly available. Since then, this dataset has been used for everything from social network analysis to fraud analysis. It is also a great dataset for authorship analysis, as we are able to extract e-mails from the sent folder of individual users. This allows us to create a dataset much larger than many previous datasets.

Accessing the Enron dataset

The full set of Enron e-mails is available at `https://www.cs.cmu.edu/~./enron/`.

The full dataset is 423 MB in a compression format called `gzip`. If you don't have a Linux-based machine to decompress (unzip) this file, get an alternative program, such as 7-zip (`http://www.7-zip.org/`).

Download the full corpus and decompress it into your data folder. By default, this will decompress into a folder called `enron_mail_20110402`.

As we are looking for authorship information, we only want the e-mails we can attribute to a specific author. For that reason, we will look in each user's sent folder—that is, e-mails they have sent.

In the Notebook, setup the data folder for the Enron dataset:

```
enron_data_folder = os.path.join(os.path.expanduser("~"), "Data",
  "enron_mail_20110402", "maildir")
```

Creating a dataset loader

We can now create a function that will choose a couple of authors at random and return each of the e-mails in their sent folder. Specifically, we are looking for the payloads—that is, the content rather than the e-mails themselves. For that, we will need an e-mail parser. The code is as follows:

```
from email.parser import Parser
p = Parser()
```

We will be using this later to extract the payloads from the e-mail files that are in the data folder.

We will be choosing authors at random, so we will be using a random state that allows us to replicate the results if we want:

```
from sklearn.utils import check_random_state
```

With our data loading function, we are going to have a lot of options. Most of these ensure that our dataset is relatively balanced. Some authors will have thousands of e-mails in their sent mail, while others will have only a few dozen. We limit our search to only authors with at least 10 e-mails using `min_docs_author` and take a maximum of 100 e-mails from each author using the `max_docs_author` parameter. We also specify how many authors we want to get—10 by default using the `num_authors` parameter. The code is as follows:

```
def get_enron_corpus(num_authors=10, data_folder=data_folder,
                     min_docs_author=10, max_docs_author=100,
                     random_state=None):
    random_state = check_random_state(random_state)
```

Next, we list all of the folders in the data folder, which are separate e-mail addresses of Enron employees. We when randomly shuffle them, allowing us to choose a new set every time the code is run. Remember that setting the random state will allow us to replicate this result:

```
    email_addresses = sorted(os.listdir(data_folder))
    random_state.shuffle(email_addresses)
```

It may seem odd that we sort the e-mail addresses, only to shuffle them around. The `os.listdir` function doesn't always return the same results, so we sort it first to get some stability. We then shuffle using a random state, which means our shuffling can reproduce a past result if needed.

We then set up our documents and class lists. We also create an `author_num`, which will tell us which class to use for each new author. We won't use the `enumerate` trick we used earlier, as it is possible that we won't choose some authors. For example, if an author doesn't have 10 sent e-mails, we will not use it. The code is as follows:

```
    documents = []
    classes = []
    author_num = 0
```

We are also going to record which authors we used and which class number we assigned to them. This isn't for the data mining, but will be used in the visualization so we can identify the authors more easily. The dictionary will simply map e-mail usernames to class values. The code is as follows:

```
    authors = {}
```

Next, we iterate through each of the e-mail addresses and look for all subfolders with "sent" in the name, indicating a sent mail box. The code is as follows:

```
for user in email_addresses:
  users_email_folder = os.path.join(data_folder, user)
  mail_folders = [os.path.join(users_email_folder,
    subfolder) for subfolder in os.listdir(users_email_folder)
                  if "sent" in subfolder]
```

We then get each of the e-mails that are in this folder. I've surrounded this call in a try-except block, as some of the authors have subdirectories in their sent mail. We could use some more detailed code to get all of these e-mails, but for now we will just continue and ignore these users. The code is as follows:

```
try:
  authored_emails = [open(os.path.join(mail_folder,
    email_filename), encoding='cp1252').read()
  for mail_folder in mail_folders
  for email_filename in os.listdir(mail_folder)]
except IsADirectoryError:
    continue
```

Next we check we have at least 10 e-mails (or whatever `min_docs_author` is set to):

```
if len(authored_emails) < min_docs_author:
    continue
```

As a next step, if we have too many e-mails from this author, only take the first 100 (from `max_docs_author`):

```
if len(authored_emails) > max_docs_author:
    authored_emails = authored_emails[:max_docs_author]
```

Next, we parse the e-mail to extract the contents. We aren't interested in the headers—the author has little control over what goes here, so it doesn't make for good data for authorship analysis. We then add those e-mail payloads to our dataset:

```
contents = [p.parsestr(email)._payload for email in
  authored_emails]
documents.extend(contents)
```

We then append a class value for this author, for each of the e-mails we added to our dataset:

```
classes.extend([author_num] * len(authored_emails))
```

We then record the class number we used for this author and *then* increment it:

```
authors[user] = author_num
author_num += 1
```

We then check if we have enough authors and, if so, we break out of the loop to return the dataset. The code is as follows:

```
if author_num >= num_authors or author_num >=
  len(email_addresses):
    break
```

We then return the datatset's documents and classes, along with our author mapping. The code is as follows:

```
return documents, np.array(classes), authors
```

Outside this function, we can now get a dataset by making the following function call. We are going to use a random state of 14 here (as always in this module), but you can try other values or set it to *none* to get a random set each time the function is called:

```
documents, classes, authors = get_enron_corpus(data_folder=enron_data_
folder, random_state=14)
```

If you have a look at the dataset, there is still a further preprocessing set we need to undertake. Our e-mails are quite messy, but one of the worst bits (from a data analysis perspective) is that these e-mails contain writings from other authors, in the form of attached replies. Take the following e-mail, which is `documents[100]`, for instance:

I am disappointed on the timing but I understand. Thanks. Mark

-----Original Message-----

From: Greenberg, Mark

Sent: Friday, September 28, 2001 4:19 PM

To: Haedicke, Mark E.

Subject: Web Site

Mark -

FYI - I have attached below a screen shot of the proposed new look and feel for the site. We have a couple of tweaks to make, but I believe this is a much cleaner look than what we have now.

This document contains another e-mail attached to the bottom as a reply, a common e-mail pattern. The first part of the e-mail is from *Mark Haedicke*, while the second is a previous e-mail written to Mark Haedicke by *Mark Greenberg*. Only the preceding text (the first instance of **-----Original Message-----**) could be attributed to the author, and this is the only bit we are actually worried about.

Extracting this information generally is not easy. E-mail is a notoriously badly used format. Different e-mail clients add their own headers, define replies in different ways, and just do things however they want. It is really surprising that e-mail works at all in the current environment.

There are some commonly used patterns that we can look for. The `quotequail` package looks for these and can find the new part of the e-mail, discarding replies and other information.

You can install `quotequail` using pip: `pip3 install quotequail`.

We are going to write a simple function to wrap the `quotequail` functionality, allowing us to easily call it on all of our documents. First we import `quotequail` and set up the function definition:

```
import quotequail
def remove_replies(email_contents):
```

Next, we use `quotequail` to unwrap the e-mail, which returns a dictionary containing the different parts of the e-mail. The code is as follows:

```
    r = quotequail.unwrap(email_contents)
```

In some cases, `r` can be none. This happens if the e-mail couldn't be parsed. In this case, we just return the full e-mail contents. This kind of messy solution is often necessary when working with real world datasets. The code is as follows:

```
    if r is None:
        return email_contents
```

The actual part of the e-mail we are interested in is called (by `quotequail`) the `text_top`. If this exists, we return this as our interesting part of the e-mail. The code is as follows:

```
    if 'text_top' in r:
        return r['text_top']
```

If it doesn't exist, `quotequail` couldn't find it. It is possible it found other text in the e-mail. If that exists, we return only that text. The code is as follows:

```
elif 'text' in r:
    return r['text']
```

Finally, if we couldn't get a result, we just return the e-mail contents, hoping they offer some benefit to our data analysis:

```
return email_contents
```

We can now preprocess all of our documents by running this function on each of them:

```
documents = [remove_replies(document) for document in documents]
```

Our preceding e-mail sample is greatly clarified now and contains only the e-mail written by *Mark Greenberg*:

I am disappointed on the timing but I understand. Thanks. Mark

Putting it all together

We can use the existing parameter space and classifier from our previous experiments—all we need to do is refit it on our new data. By default, training in scikit-learn is done from scratch—subsequent calls to `fit()` will discard any previous information.

There is a class of algorithms called online learning that update the training with new samples and don't restart their training each time. We will see online learning in action later in this module, including the next chapter, *Chapter 10*, *Clustering News Articles*.

As before, we can compute our scores by using `cross_val_score` and print the results. The code is as follows:

```
scores = cross_val_score(pipeline, documents, classes, scoring='f1')
print("Score: {:.3f}".format(np.mean(scores)))
```

The result is 0.523, which is a reasonable result for such a messy dataset. Adding more data (such as increasing `max_docs_author` in the dataset loading) can improve these results.

Evaluation

It is generally never a good idea to base an assessment on a single number. In the case of the f-score, it is usually more robust than *tricks* that give good scores despite not being useful. An example of this is accuracy. As we said in our previous chapter, a spam classifier could predict everything as being spam and get over 80 percent accuracy, although that solution is not useful at all. For that reason, it is usually worth going more in-depth on the results.

To start with, we will look at the confusion matrix, as we did in *Chapter 8*, *Beating CAPTCHAs with Neural Networks*. Before we can do that, we need to predict a testing set. The previous code uses `cross_val_score`, which doesn't actually give us a trained model we can use. So, we will need to refit one. To do that, we need training and testing subsets:

```
from sklearn.cross_validation import train_test_split
training_documents, testing_documents, y_train, y_test =
train_test_split(documents, classes, random_state=14)
```

Next, we fit the pipeline to our training documents and create our predictions for the testing set:

```
pipeline.fit(training_documents, y_train)
y_pred = pipeline.predict(testing_documents)
```

At this point, you might be wondering what the best combination of parameters actually was. We can extract this quite easily from our grid search object (which is the `classifier` step of our pipeline):

```
print(pipeline.named_steps['classifier'].best_params_)
```

The results give you all of the parameters for the classifier. However, most of the parameters are the defaults that we didn't touch. The ones we did search for were `C` and `kernel`, which were set to `1` and `linear`, respectively.

Now we can create a confusion matrix:

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_pred, y_test)
cm = cm / cm.astype(np.float).sum(axis=1)
```

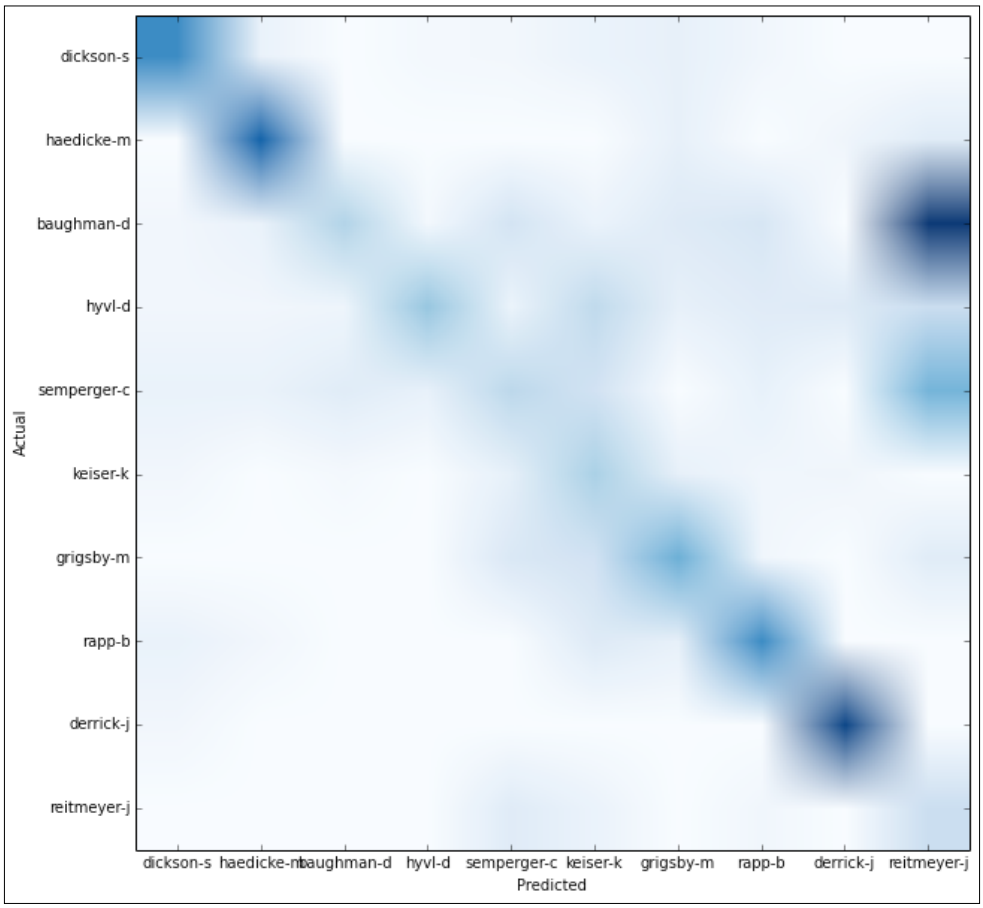
Next we get our authors so that we can label the axis correctly. For this purpose, we use the `authors` dictionary that our Enron dataset loaded. The code is as follows:

```
sorted_authors = sorted(authors.keys(), key=lambda x:authors[x])
```

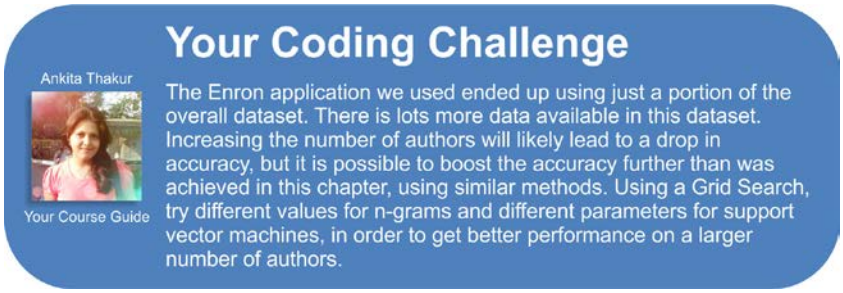
Finally, we show the confusion matrix using `matplotlib`. The only changes from the last chapter are highlighted below; just replace the letter labels with the authors from this chapter's experiments:

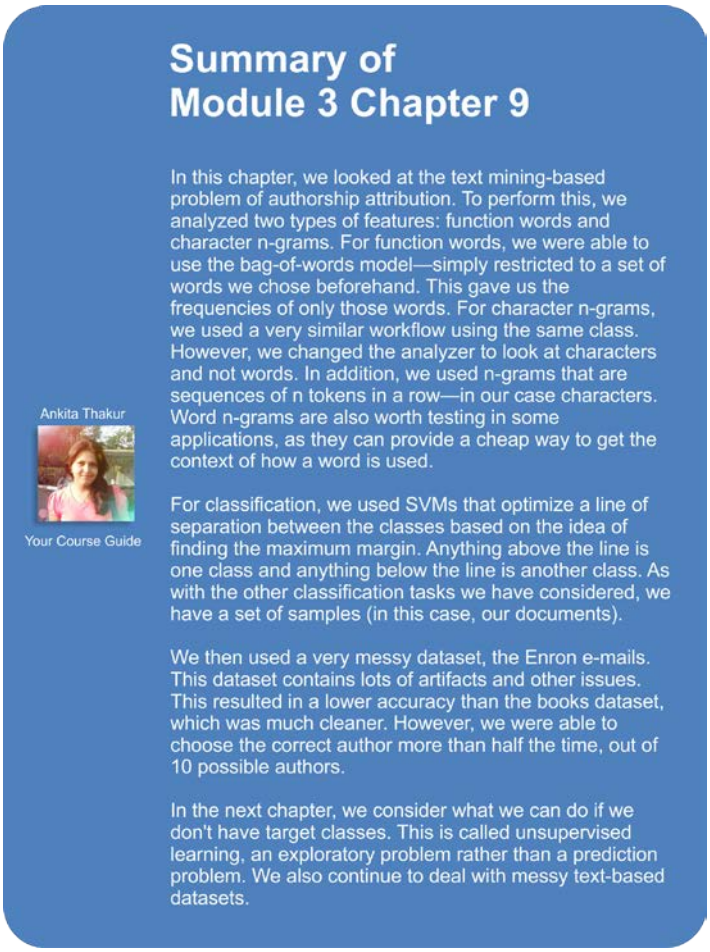
```
%matplotlib inline
from matplotlib import pyplot as plt
plt.figure(figsize=(10,10))
plt.imshow(cm, cmap='Blues')
tick_marks = np.arange(len(sorted_authors))
plt.xticks(tick_marks, sorted_authors)
plt.yticks(tick_marks, sorted_authors)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

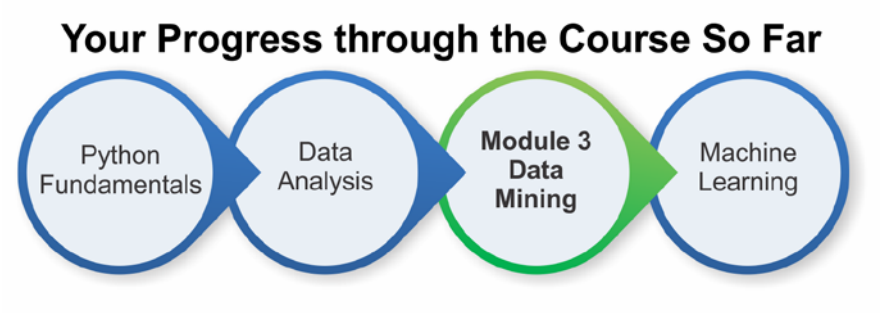
The results are shown in the following figure:



We can see that authors are predicted correctly in most cases—there is a clear diagonal line with high values. There are some large sources of error though (darker values are larger): e-mails from user `baughman-d` are typically predicted as being from `reitmeyer-j` for instance.






Your Progress through the Course So Far
Python Fundamentals
Data Analysis
Module 3 Data Mining
Machine Learning

10
Clustering News Articles

In most of the previous chapters, we performed data mining knowing what we were looking for. Our use of target classes allowed us to learn how our variables model those targets during the training phase. This type of learning, where we have targets to train against, is called **supervised learning**. In this chapter, we consider what we do without those targets. This is **unsupervised learning** and is much more of an exploratory task. Rather than wanting to classify with our model, the goal in unsupervised learning is more about exploring the data to find insights.

In this chapter, we look at clustering news articles to find trends and patterns in the data. We look at how we can extract data from different websites using a link aggregation website to show a variety of news stories.

The key concepts covered in this chapter include:

- Obtaining text from arbitrary websites
- Using the reddit API to collect interesting news stories
- Cluster analysis for unsupervised data mining
- Extracting topics from documents
- Online learning for updating a model without retraining it
- Cluster ensembling to combine different models

Obtaining news articles

In this chapter, we will build a system that takes a live feed of news articles and groups them together, where the groups have similar topics. You could run the system over several weeks (or longer) to see how trends change over that time.

Our system will start with the popular link aggregation website **reddit**, which stores lists of links to other websites, as well as a comments section for discussion. Links on reddit are broken into several categories of links, called **subreddits**. There are subreddits devoted to particular TV shows, funny images, and many other things. What we are interested in is the subreddits for news. We will use the `/r/worldnews` subreddit in this chapter, but the code should work with any other subreddit.

In this chapter, our goal is to download popular stories, and then cluster them to see any major themes or concepts that occur. This will give us an insight into the popular focus, without having to manually analyze hundreds of individual stories.

Using a Web API to get data

We have used web-based APIs to extract data in several of our previous chapters. For instance, in *Chapter 7*, *Discovering Accounts to Follow Using Graph Mining*, we used Twitter's API to extract data. Collecting data is a critical part of the data mining pipeline, and web-based APIs are a fantastic way to collect data on a variety of topics.

There are three things you need to consider when using a web-based API for collecting data: authorization methods, rate limiting, and API endpoints.

Authorization methods allow the data provider to know who is collecting the data, in order to ensure that they are being appropriately rate-limited and that data access can be tracked. For most websites, a personal account is often enough to start collecting data, but some websites will ask you to create a formal developer account to get this access.

Rate limiting is applied to data collection, particularly free services. It is important to be aware of the rules when using APIs, as they can and do change from website to website. Twitter's API limit is 180 requests per 15 minutes (depending on the particular API call). Reddit, as we will see later, allows 30 requests per minute. Other websites impose daily limits, while others limit on a per-second basis. Even within websites, there are drastic differences for different API calls. For example, Google Maps has smaller limits and different API limits per-resource, with different allowances for the number of requests per hour.

If you find you are creating an app or running an experiment that needs more requests and faster responses, most API providers have commercial plans that allow for more calls.

API Endpoints are the actual URLs that you use to extract information. These vary from website to website. Most often, web-based APIs will follow a **RESTful** interface (short for **Representational State Transfer**). RESTful interfaces often use the same actions that HTTP does: GET, POST, and DELETE are the most common. For instance, to retrieve information on a resource, we might use the following API endpoint: `www.dataprovider.com/api/resource_type/resource_id/`.

To *get* the information, we just send a HTTP GET request to this URL. This will return information on the resource with the given type and ID. Most APIs follow this structure, although there are some differences in the implementation. Most websites with APIs will have them appropriately documented, giving you details of all the APIs that you can retrieve.

First, we set up the parameters to connect to the service. To do this, you will need a developer key for reddit. In order to get this key, log in to the `https://www.reddit.com/login` website and go to `https://www.reddit.com/prefs/apps`. From here, click on **are you a developer? create an app…** and fill out the form, setting the type as *script*. You will get your client ID and a secret, which you can add to a new IPython Notebook:

```
CLIENT_ID = "<Enter your Client ID here>"
CLIENT_SECRET = "<Enter your Client Secret here>"
```

Reddit also asks that, when you use their API, you set the user agent to a unique string that includes your username. Create a user agent string that uniquely identifies your application. I used the name of the book, `chapter 10`, and a version number of 0.1 to create my user agent, but it can be any string you like. Note that not doing this will result in your connection being heavily rate-limited:

```
 USER_AGENT = "python:<your unique user agent> (by /u/<your reddit
username>)"
```

In addition, you will need to log into reddit using your username and password. If you don't have one already, sign up for a new one (it is free and you don't need to verify with personal information either).

You will need your password to complete the next step, so be careful before sharing your code to others to remove it. If you don't put your password in, set it to `none` and you will be prompted to enter it. However, due to the way IPython Notebooks work, you'll need to enter it into the command-line terminal that started the IPython server, not the notebook itself. If you can't do this, you'll need to set it in the script. The developers of the IPython Notebook are working on a plugin to fix this, but it was not yet available at the time of writing.

Now let's create the username and password:

```
USERNAME = "<your reddit username>"
PASSWORD = "<your reddit password>"
```

Next, we are going to create a function to log with this information. The reddit login API will return a token that you can use for further connections, which will be the result of this function. The code is as follows:

```
def login(username, password):
```

First, if you don't want to add your password to the script, you can set it to `None` and you will be prompted, as explained previously. The code is as follows:

```
    if password is None:
        password = getpass.getpass("Enter reddit password for user {}:
".format(username))
```

It is very important that you set the user agent to a unique value, or your connection might be severely restricted. The code is as follows:

```
    headers = {"User-Agent": USER_AGENT}
```

Next, we set up a HTTP authorization object to allow us to login at reddit's servers:

```
    client_auth = requests.auth.HTTPBasicAuth(CLIENT_ID, CLIENT_
SECRET)
```

To login, we make a POST request to the `access_token` endpoint. The data we send is our username and password, along with the grant type that is set to `password` for this example:

```
    post_data = {"grant_type": "password", "username": username,
"password": password}
```

Finally, we use the `requests` library to make the login request (this is done via a HTTP POST request) and return the result, which is a dictionary of values. One of these values is the token we will need for future requests. The code is as follows:

```
    response = requests.post("https://www.reddit.com/api/v1/access_
token", auth=client_auth, data=post_data, headers=headers)
    return response.json()
```

We can call now our function to get a token:

```
token = login(USERNAME, PASSWORD)
```

The `token` object is just a dictionary, but it contains the `access_token` string that we will pass with future requests. It also contains other information such as the scope of the token (which would be everything) and the time in which it expires—for example:

```
{'access_token': '<semi-random string>',  'expires_in': 3600,
'scope': '*',  'token_type': 'bearer'}
```

Reddit as a data source

Reddit (`www.reddit.com`) is a link aggregation website used by millions worldwide, although the English versions are US-centric. Any user can contribute a link to a website they found interesting, along with a title for that link. Other users can then *upvote* it, indicating that they liked the link, or *downvote* it, indicating they didn't like the link. The highest voted links are moved to the top of the page, while the lower ones are not shown. Older links are removed over time (depending on how many upvotes it has). Users who have stories upvoted earn points called *karma*, providing an incentive to submit only good stories.

Reddit also allows nonlink content, called self-posts. These contain a title and some text that the submitter enters. These are used for asking questions and starting discussions, but do not count towards a person's karma. For this chapter, we will be considering only link-based posts, and not comment-based posts.

Posts are separated into different sections of the website called *subreddits*. A subreddit is a collection of posts that are related. When a user submits a link to reddit, they choose which subreddit it goes into. Subreddits have their own administrators, and have their own rules about what is valid content for that subreddit.

By default, posts are sorted by *Hot*, which is a function of the age of a post, the number of upvotes, and the number of downvotes it has received. There is also *New*, which just gives you the most recently posted stories (and therefore contains lots of spam and bad posts), and *Top*, which is the highest voted stories for a given time period. In this chapter, we will be using *Hot*, which will give us recent, higher-quality stories (there really are a lot of poor-quality links in *New*).

Using the token we previously created, we can now obtain sets of links from a subreddit. To do that, we will use the `/r/<subredditname>` API endpoint that, by default, returns the Hot stories. We will use the `/r/worldnews` subreddit:

```
subreddit = "worldnews"
```

The URL for the previous end-point lets us create the full URL, which we can set using string formatting:

```
url = "https://oauth.reddit.com/r/{}".format(subreddit)
```

Next, we need to set the headers. This is needed for two reasons: to allow us to use the authorization token we received earlier and to set the user agent to stop our requests from being heavily restricted. The code is as follows:

```
headers = {"Authorization": "bearer {}".format(token['access_token']),
           "User-Agent": USER_AGENT}
```

Then, as before, we use the `requests` library to make the call, ensuring that we set the headers:

```
response = requests.get(url, headers=headers)
```

Calling `json()` on this will result in a Python dictionary containing the information returned by Reddit. It will contain the top 25 results from the given subreddit. We can get the title by iterating over the stories in this response. The stories themselves are stored under the dictionary's `data` key. The code is as follows:

```
for story in result['data']['children']:
    print(story['data']['title'])
```

Getting the data

Our dataset is going to consist of posts from the Hot list of the `/r/worldnews` subreddit. We saw in the previous section how to connect to reddit and how to download links. To put it all together, we will create a function that will extract the titles, links, and score for each item in a given subreddit.

We will iterate through the subreddit, getting a maximum of 100 stories at a time. We can also do pagination to get more results. We can read a large number of pages before reddit will stop us, but we will limit it to 5 pages.

As our code will be making repeated calls to an API, it is important to remember to rate-limit our calls. To do so, we will need the `sleep` function:

```
from time import sleep
```

Our function will accept a subreddit name and an authorization token. We will also accept a number of pages to read, although we will set a default of 5:

```
def get_links(subreddit, token, n_pages=5):
```

We then create a list to store the stories in:

```
stories = []
```

We saw in *Chapter 7*, *Discovering Accounts to Follow Using Graph Mining*, how pagination works for the Twitter API. We get a cursor with our returned results, which we send with our request. Twitter will then use this cursor to get the next page of results. The reddit API does almost exactly the same thing, except it calls the parameter `after`. We don't need it for the first page, so we initially set it to `none`. We will set it to a meaningful value after our first page of results. The code is as follows:

```
after = None
```

We then iterate for the number of pages we want to return:

```
for page_number in range(n_pages):
```

Inside the loop, we initialize our URL structure as we did before:

```
    headers = {"Authorization": "bearer
    {}".format(token['access_token']),
        "User-Agent": USER_AGENT}
    url = "https://oauth.reddit.com/r/{}?limit=100".
        format(subreddit)
```

From the second loop onwards, we need to set the `after` parameter (otherwise, we will just get multiple copies of the same page of results). This value will be set in the previous iteration of the loop—the first loop sets the after parameter for the second loop and so on. If present, we append it to the end of our URL, telling reddit to get us the next page of data. The code is as follows:

```
    if after:
        url += "&after={}".format(after)
```

Then, as before, we use the requests library to make the call and turn the result into a Python dictionary using `json()`:

```
    response = requests.get(url, headers=headers)
    result = response.json()
```

This result will give us the `after` parameter for the next time the loop iterates, which we can now set as follows:

```
    after = result['data']['after']
```

We then sleep for 2 seconds to avoid exceeding the API limit:

```
    sleep(2)
```

As the last action inside the loop, we get each of the stories from the returned result and add them to our `stories` list. We don't need all of the data—we only get the title, URL, and score. The code is as follows:

```
            stories.extend([(story['data']['title'], story['data']['url'],
story['data']['score'])
                            for story in result['data']['children']])
```

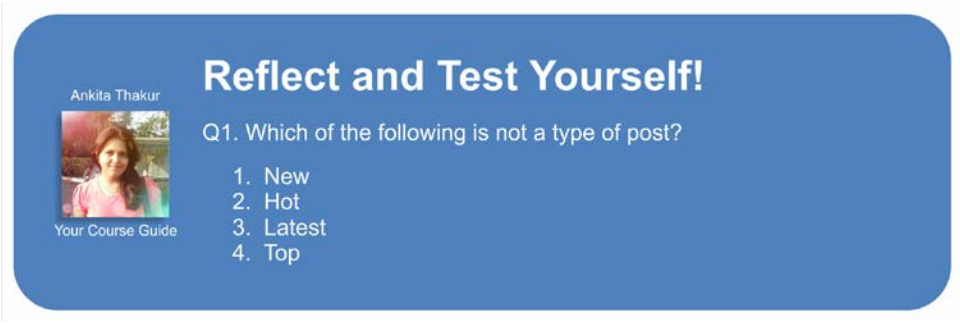
Finally (and outside the loop), we return all the stories we have found:

```
        return stories
```

Calling the `stories` function is a simple case of passing the authorization token and the subreddit name:

```
stories = get_links("worldnews", token)
```

The returned results should contain the title, URL, and 500 stories, which we will now use to extract the actual text from the resulting websites.



Extracting text from arbitrary websites

The links that we get from reddit go to arbitrary websites run by many different organizations. To make it harder, those pages were designed to be read by a human, not a computer program. This can cause a problem when trying to get the actual content/story of those results, as modern websites have a lot going on in the background. JavaScript libraries are called, style sheets are applied, advertisements are loaded using AJAX, extra content is added to sidebars, and various other things are done to make the modern webpage a complex document. These features make the modern Web what it is, but make it difficult to automatically get good information from!

Finding the stories in arbitrary websites

To start with, we will download the full webpage from each of these links and store them in our data folder, under a `raw` subfolder. We will process these to extract the useful information later on. This caching of results ensures that we don't have to continuously download the websites while we are working. First, we set up the data folder path:

```
import os
data_folder = os.path.join(os.path.expanduser("~"), "Data",
"websites", "raw")
```

We are going to use MD5 hashing to create unique filenames for our articles, so we will import `hashlib` to do that. A `hash` function is a function that converts some input (in our case a string containing the title) into a string that is seemingly random. The same input will always return the same output, but slightly different inputs will return drastically different outputs. It is also impossible to go from a hash value to the original value, making it a one-way function. The code is as follows:

```
import hashlib
```

We are going to simply skip any website downloads that fail. In order to make sure we don't lose too much information doing this, we maintain a simple counter of the number of errors that occur. We are going to suppress any error that occurs, which could result in a systematic problem prohibiting downloads. If this error counter is too high, we can look at what those errors were and try to fix them. For example, if the computer has no Internet access, all 500 of the downloads will fail and you should probably fix that before continuing!

If there is no error in the download, zero should be the output:

```
number_errors = 0
```

Next, we iterate through each of our stories:

```
for title, url, score in stories:
```

We then create a unique output filename for our article by hashing the title. Titles in reddit don't need to be unique, which means there is a possibility of two stories having the same title and, therefore, clashing in our dataset. To get our unique filename, we simply hash the URL of the article using the MD5 algorithm. While MD5 is known to have some problems, it is unlikely that a problem (a collision) will occur in our scenario, and we don't need to worry too much even if it does and we don't need to worry too much about collisions if they do occur.

```
    output_filename = hashlib.md5(url.encode()).hexdigest()
    fullpath = os.path.join(data_folder, output_filename + ".txt")
```

Next, we download the actual page and save it to our `output` folder:

```
try:
    response = requests.get(url)
    data = response.text
    with open(fullpath, 'w') as outf:
        outf.write(data)
```

If there is an error in obtaining the website, we simply skip this website and keep going. This code will work on 95 percent of websites and that is good enough for our application, as we are looking for general trends and not exactness. Note that sometimes you do care about getting 100 percent of responses, and you should adjust your code to accommodate more errors. The code to get those final 5 to 10 percent of websites will be significantly more complex. We then catch any error that could occur (it is the Internet, lots of things could go wrong), increment our error count, and continue.

```
except Exception as e:
    number_errors += 1
    print(e)
```

If you find that too many errors occur, change the `print(e)` line to just type `raise` instead. This will cause the exception to be called, allowing you to debug the problem.

Now, we have a bunch of websites in our `raw` subfolder. After taking a look at these pages (open the created files in a text editor), you can see that the content is there but there are HTML, JavaScript, CSS code, as well as other content. As we are only interested in the story itself, we now need a way to extract this information from these different websites.

Putting it all together

After we get the raw data, we need to find the story in each. There are a few online sources that use data mining to achieve this. You can find them listed in *Chapter 13*. It is rarely needed to use such complex algorithms, although you can get better accuracy using them. This is part of data mining—knowing when to use it, and when not to.

First, we get a list of each of the filenames in our `raw` subfolder:

```
filenames = [os.path.join(data_folder, filename)
             for filename in os.listdir(data_folder)]
```

Next, we create an `output` folder for the text only versions that we will extract:

```
text_output_folder = os.path.join(os.path.expanduser("~"), "Data",
                                  "websites", "textonly")
```

Next, we develop the code that will extract the text from the files. We will use the `lxml` library to parse the HTML files, as it has a good HTML parser that deals with some badly formed expressions. The code is as follows:

```
from lxml import etree
```

The actual code for extracting text is based on three steps. First, we iterate through each of the nodes in the HTML file and extract the text in it. Second, we skip any node that is JavaScript, styling, or a comment, as this is unlikely to contain information of interest to us. Third, we ensure that the content has at least 100 characters. This is a good baseline, but it could be improved upon for more accurate results.

As we said before, we aren't interested in scripts, styles, or comments. So, we create a list to ignore nodes of those types. Any node that has a type in this list will not be considered as containing the story. The code is as follows:

```
skip_node_types = ["script", "head", "style", etree.Comment]
```

We will now create a function that parses an HTML file into an `lxml etree`, and then we will create another function that parses this tree looking for text. This first function is pretty straightforward; simply open the file and create a tree using the `lxml` library's parsing function for HTML files. The code is as follows:

```
def get_text_from_file(filename):
    with open(filename) as inf:
        html_tree = lxml.html.parse(inf)
    return get_text_from_node(html_tree.getroot())
```

In the last line of that function, we call the `getroot()` function to get the root node of the tree, rather than the full `etree`. This allows us to write our text extraction function to accept any node, and therefore write a recursive function.

This function will call itself on any child nodes to extract the text from them, and then return the concatenation of any child nodes text.

If the node this function is passed doesn't have any child nodes, we just return the text from it. If it doesn't have any text, we just return an empty string. Note that we also check here for our third condition—that the text is at least 100 characters long. The code is as follows:

```
def get_text_from_node(node):
    if len(node) == 0:
```

```
        # No children, just return text from this item
        if node.text and len(node.text) > 100:
            return node.text
        else:
            return ""
```

At this point, we know that the node has child nodes, so we recursively call this function on each of those child nodes and then join the results when they return. The code is as follows:

```
    results = (get_text_from_node(child) for child in node
                    if child.tag not in skip_node_types)
    return "\n".join(r for r in results if len(r) > 1)
```

The final condition on the return result stops blank lines being returned (for example, when a node has no children and no text).

We can now run this code on all of the raw HTML pages by iterating through them, calling the text extraction function on each, and saving the results to the text-only subfolder:

```
for filename in os.listdir(data_folder):
    text = get_text_from_file(os.path.join(data_folder, filename))
    with open(os.path.join(text_output_folder, filename), 'w')
      as outf:
        outf.write(text)
```

You can evaluate the results manually by opening each of the files in the text only subfolder and checking their content. If you find too many of the results have nonstory content, try increasing the minimum 100 character limit. If you still can't get good results, or need better results for your application, try the more complex methods listed in *Chapter 13*.

Grouping news articles

The aim of this chapter is to discover trends in news articles by clustering, or grouping, them together. To do that, we will use the k-means algorithm, a classic machine-learning algorithm originally developed in 1957.

Clustering is an unsupervised learning technique and we use clustering algorithms for exploring data. Our dataset contains approximately 500 stories, and it would be quite arduous to examine each of those stories individually. Even if we used summary statistics, that is still a lot of data. Using clustering allows us to group similar stories together, and we can explore the themes in each cluster independently.

We use clustering techniques when we don't have a clear set of target classes for our data. In that sense, clustering algorithms have little direction in their learning. They learn according to some function, regardless of the underlying meaning of the data. For this reason, it is critical to choose good features. In supervised learning, if you choose poor features, the learning algorithm can choose to not use those features. For instance, support vector machines will give little weight to features that aren't useful in classification. However, with clustering, all features are used in the final result—even if those features don't provide us with the answer we were looking for.

When performing cluster analysis on real-world data, it is always a good idea to have a sense of what sorts of features will work for your scenario. In this chapter, we will use the bag-of-words model. We are looking for topic-based groups, so we will use topic-based features to model the documents. We know those features work because of the work others have done in supervised versions of our problem. In contrast, if we were to perform an authorship-based clustering, we would use features such as those found in the *Chapter 9*, *Authorship Attribution* experiment.

The k-means algorithm

The k-means clustering algorithm finds centroids that best represent the data using an iterative process. The algorithm starts with a predefined set of centroids, which are normally data points taken from the training data. The *k* in k-means is the number of centroids to look for and how many clusters the algorithm will find. For instance, setting *k* to *3* will find three clusters in the dataset.

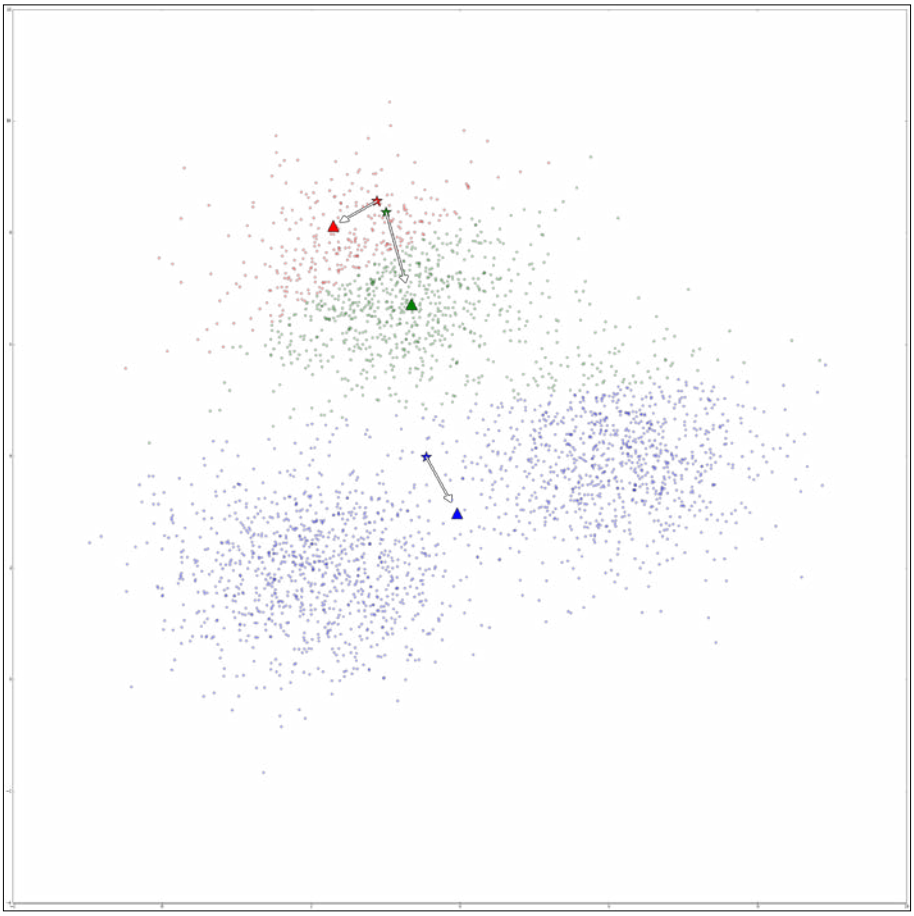
There are two phases to the k-means: assignment and updating.

In the assignment step, we set a label to every sample in the dataset linking it to the nearest centroid. For each sample nearest to centroid 1, we assign the label 1. For each sample nearest to centroid 2, we assign a label 2 and so on for each of the *k* centroids. These labels form the clusters, so we say that each data point with the label 1 is in cluster 1 (at this time only, as assignments can change as the algorithm runs).

In the updating step, we take each of the clusters and compute the centroid, which is the mean of all of the samples in that cluster.

The algorithm then iterates between the assignment step and the updating step; each time the updating step occurs, each of the centroids moves a small amount. This causes the assignments to change slightly, causing the centroids to move a small amount in the next iteration. This repeats until some stopping criterion is reached. It is common to stop after a certain number of iterations, or when the total movement of the centroids is very low. The algorithm can also complete in some scenarios, which means that the clusters are stable—the assignments do not change and neither do the centroids.

In the following figure, k-means was performed over a dataset created randomly, but with three clusters in the data. The stars represent the starting location of the centroids, which were chosen randomly by picking a random sample from the dataset. Over 5 iterations of the k-means algorithm, the centroids move to the locations represented by the triangles.



The k-means algorithm is fascinating for its mathematical properties and historical significance. It is an algorithm that (roughly) only has a single parameter, and is quite effective and frequently used, even more than 50 years after its discovery.

There is a k-means algorithm in scikit-learn, which we import from the `cluster` subpackage:

```
from sklearn.cluster import KMeans
```

We also import the `CountVectorizer` class's close cousin, `TfidfVectorizer`. This vectorizer applies a weighting to each term's counts, depending on how many documents it appears in. Terms that appear in many documents are weighted lower (by dividing the value by the log of the number of documents it appears in). For many text mining applications, using this type of weighting scheme can improve performance quite reliably. The code is as follows:

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

We then set up our pipeline for our analysis. This has two steps. The first is to apply our vectorizer, and the second is to apply our k-means algorithm. The code is as follows:

```
from sklearn.pipeline import Pipeline
n_clusters = 10
pipeline = Pipeline([('feature_extraction', TfidfVectorizer(max_
df=0.4)),
                     ('clusterer', KMeans(n_clusters=n_clusters))
                     ])
```

The `max_df` parameter is set to a low value of 0.4, which says *ignore any word that occurs in more than 40 percent of documents*. This parameter is invaluable for removing function words that give little topic-based meaning on their own.

Removing any word that occurs in more than 40 percent of documents will remove function words, making this type of preprocessing quite useless for the work we saw in *Chapter 9, Authorship Attribution*.

We then fit and predict this pipeline. We have followed this process a number of times in this module so far for classification tasks, but there is a difference here—we do not give the target classes for our dataset to the fit function. This is what makes this an unsupervised learning task! The code is as follows:

```
pipeline.fit(documents)
labels = pipeline.predict(documents)
```

The `labels` variable now contains the cluster numbers for each sample. Samples with the same label are said to belong in the same cluster. It should be noted that the cluster labels themselves are meaningless: clusters 1 and 2 are no more similar than clusters 1 and 3.

We can see how many samples were placed in each cluster using the `Counter` class:

```
from collections import Counter
c = Counter(labels)
for cluster_number in range(n_clusters):
    print("Cluster {} contains {} samples".format(cluster_number,
c[cluster_number]))
```

Many of the results (keeping in mind that your dataset will be quite different to mine) consist of a large cluster with the majority of instances, several medium clusters, and some clusters with only one or two instances. This imbalance is quite normal in many clustering applications.

Evaluating the results

Clustering is mainly an exploratory analysis, and therefore it is difficult to evaluate a clustering algorithm's results effectively. A straightforward way is to evaluate the algorithm based on the criteria the algorithm tries to learn from.

If you have a test set, you can evaluate clustering against it. For more details, visit `http://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html`.

In the case of the k-means algorithm, the criterion that it uses when developing the centroids is to minimize the distance from each sample to its nearest centroid. This is called the inertia of the algorithm and can be retrieved from any `KMeans` instance that has had fit called on it:

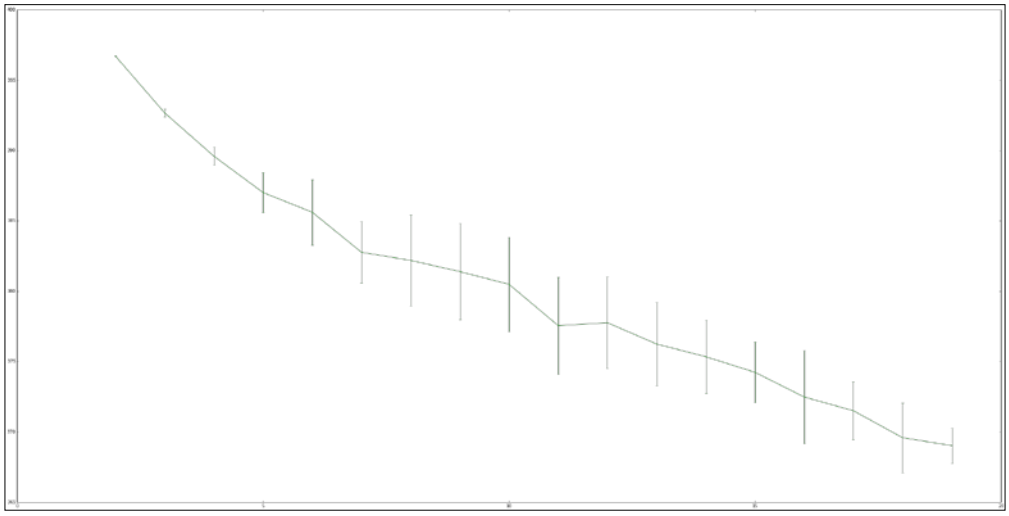
```
pipeline.named_steps['clusterer'].inertia_
```

The result on my dataset was 343.94. Unfortunately, this value is quite meaningless by itself, but we can use it to determine how many clusters we should use. In the preceding example, we set `n_clusters` to 10, but is this the best value? The following code runs the k-means algorithm 10 times with each value of `n_clusters` from 2 to 20. For each run, it records the inertia of the result.

We only fit the `X` matrix once per value of `n_clusters` to (drastically) improve the speed of this code:

```
inertia_scores = []
n_cluster_values = list(range(2, 20))
for n_clusters in n_cluster_values:
    cur_inertia_scores = []
    X = TfidfVectorizer(max_df=0.4).fit_transform(documents)
    for i in range(10):
        km = KMeans(n_clusters=n_clusters).fit(X)
        cur_inertia_scores.append(km.inertia_)
    inertia_scores.append(cur_inertia_scores)
```

The `inertia_scores` variable now contains a list of inertia scores for each `n_clusters` value between 2 and 20. We can plot this to get a sense of how this value interacts with `n_clusters`:



Overall, the value of the inertia should decrease with reducing improvement as the number of clusters improves, which we can broadly see from these results. The increase between values of 6 to 7 is due only to the randomness in selecting the centroids, which directly affect how good the final results are. Despite this, there is a general trend (in these results; your results may vary) that about 6 clusters is the last time a major improvement in the inertia occurred.

After this point, only slight improvements are made to the inertia, although it is hard to be specific about vague criteria such as this. Looking for this type of pattern is called the elbow rule, in that we are looking for an elbow-esque bend in the graph. Some datasets have more pronounced elbows, but this feature isn't guaranteed to even appear (some graphs may be smooth!).

Based on this analysis, we set `n_clusters` to be 6 and then rerun the algorithm:

```
n_clusters = 6
pipeline = Pipeline([('feature_extraction',
  TfidfVectorizer(max_df=0.4)),
                     ('clusterer', KMeans(n_clusters=n_clusters))
                     ])
pipeline.fit(documents)
labels = pipeline.predict(documents)
```

Extracting topic information from clusters

Now we set our sights on the clusters in an attempt to discover the topics in each. We first extract the `term` list from our feature extraction step:

```
terms = pipeline.named_steps['feature_extraction'].get_feature_names()
```

We also set up another counter for counting the size of each of our classes:

```
c = Counter(labels)
```

Iterating over each cluster, we print the size of the cluster as before. It is important to keep in mind the sizes of the clusters when evaluating the results—some of the clusters will only have one sample, and are therefore not indicative of a general trend. The code is as follows:

```
for cluster_number in range(n_clusters):
    print("Cluster {} contains {} samples".format(cluster_number,
c[cluster_number]))
```

Next (and still in the loop), we iterate over the most important terms for this cluster. To do this, we take the five largest values from the centroid, which we get by finding the features that have the highest values in the centroid itself. The code is as follows:

```
    print("  Most important terms")
    centroid = pipeline.named_steps['clusterer'].cluster_centers_
[cluster_number]
    most_important = centroid.argsort()
```

We then print out the most important five terms:

```
for i in range(5):
```

We use the negation of `i` in this line, as our `most_important` array is sorted with lowest values first:

```
        term_index = most_important[-(i+1)]
```

We then print the rank, term, and score for this value:

```
        print("  {0}) {1} (score: {2:.4f})".format(i+1, terms[term_
index], centroid[term_index]))
```

The results can be quite indicative of current trends. In my results (March 2015), the clusters correspond to health matters, Middle East tensions, Korean tensions, and Russian affairs. These were the main topics frequenting news around this time—although this has hardly changed for a number of years!

Using clustering algorithms as transformers

As a side note, one interesting property about the k-means algorithm (and any clustering algorithm) is that you can use it for feature reduction. There are many methods to reduce the number of features (or create new features to embed the dataset on), such as Principle Component Analysis, Latent Semantic Indexing, and many others. One issue with many of these algorithms is that they often need lots of computing power.

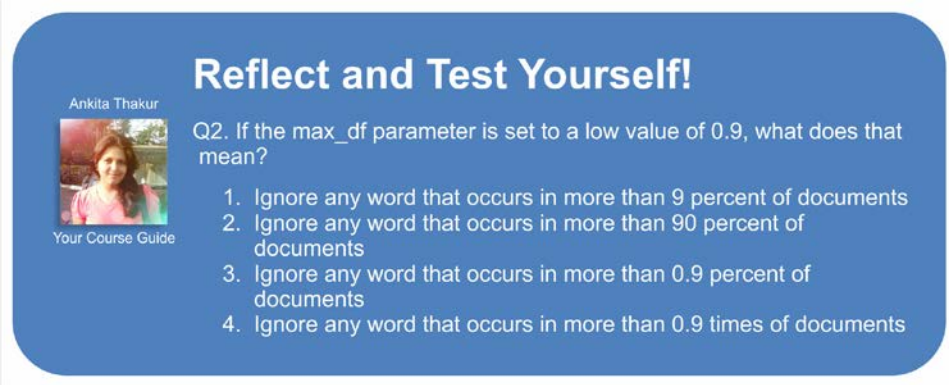
In the preceding example, the terms list had more than 14,000 entries in it—it is quite a large dataset. Our k-means algorithm transformed these into just six clusters. We can then create a dataset with a much lower number of features by taking the distance to each centroid as a feature. The code is as follows:

To do this, we call the transform function on a KMeans instance. Our pipeline is fit for this purpose, as it has a k-means instance at the end:

```
X = pipeline.transform(documents)
```

This calls the transform method on the final step of the pipeline, which is an instance of k-means. This results in a matrix that has six features and the number of samples is the same as the length of documents.

You can then perform your own second-level clustering on the result, or use it for classification if you have the target values. A possible workflow for this would be to perform some feature selection using the supervised data, use clustering to reduce the number of features to a more manageable number, and then use the results in a classification algorithm such as SVMs.



Clustering ensembles

In *Chapter 3*, *Predicting Sports Winners with Decision Trees*, we looked at a classification ensemble using the random forest algorithm, which is an ensemble of many low-quality, tree-based classifiers. Ensembling can also be performed using clustering algorithms. One of the key reasons for doing this is to smooth the results from many runs of an algorithm. As we saw before, the results from running k-means are varied, depending on the selection of the initial centroids. Variation can be reduced by running the algorithm many times and then combining the results.

Ensembling also reduces the effects of choosing parameters on the final result. Most clustering algorithms are quite sensitive to the parameter values chosen for the algorithm. Choosing slightly different parameters results in different clusters.

Evidence accumulation

As a basic ensemble, we can first cluster the data many times and record the labels from each run. We then record how many times each pair of samples was clustered together in a new matrix. This is the essence of the **Evidence Accumulation Clustering** (**EAC**) algorithm.

EAC has two major steps. The first step is to cluster the data many times using a lower-level clustering algorithm such as k-means and record the frequency that samples were in the same cluster, in each iteration. This is stored in a `coassociation` matrix. The second step is to perform a cluster analysis on the resulting `coassociation` matrix, which is performed using another type of clustering algorithm called hierarchical clustering. This has an interesting property, as it is mathematically the same as finding a tree that links all the nodes together and removing weak links.

We can create a `coassociation` matrix from an array of labels by iterating over each of the labels and recording where two samples have the same label. We use SciPy's `csr_matrix`, which is a type of sparse matrix:

```
from scipy.sparse import csr_matrix
```

Our function definition takes a set of labels:

```
def create_coassociation_matrix(labels):
```

We then record the rows and columns of each match. We do these in a list. Sparse matrices are commonly just sets of lists recording the positions of nonzero values, and `csr_matrix` is an example of this type of sparse matrix:

```
    rows = []
    cols = []
```

We then iterate over each of the individual labels:

```
    unique_labels = set(labels)
    for label in unique_labels:
```

We look for all samples that have this label:

```
        indices = np.where(labels == label)[0]
```

For each pair of samples with the preceding label, we record the position of both samples in our list. The code is as follows:

```
        for index1 in indices:
            for index2 in indices:
                rows.append(index1)
                cols.append(index2)
```

Outside all loops, we then create the data, which is simply the value 1 for every time two samples were listed together. We get the number of 1 to place by noting how many matches we had in our labels set altogether. The code is as follows:

```
    data = np.ones((len(rows),))
    return csr_matrix((data, (rows, cols)), dtype='float')
```

To get the coassociation matrix from the labels, we simply call this function:

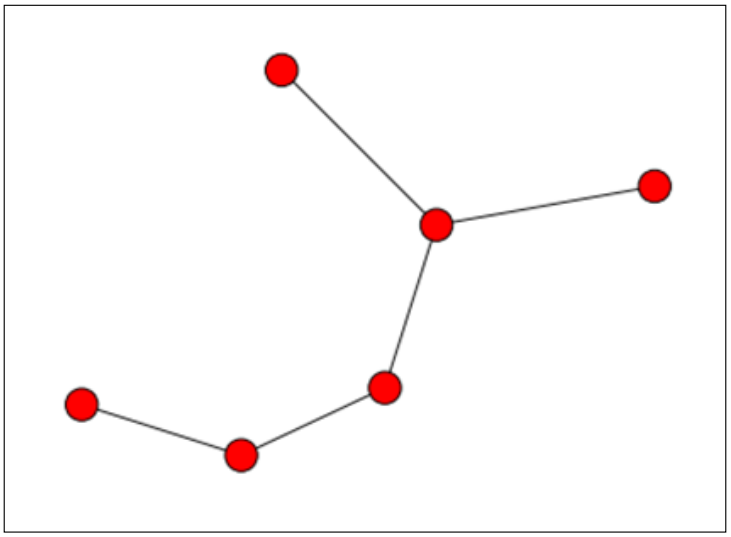
```
C = create_coassociation_matrix(labels)
```

From here, we can add multiple instances of these matrices together. This allows us to combine the results from multiple runs of k-means. Printing out C (just enter C into a new cell and run it) will tell you how many cells have nonzero values in them. In my case, about half of the cells had values in them, as my clustering result had a large cluster (the more even the clusters, the lower the number of nonzero values).

The next step involves the hierarchical clustering of the coassociation matrix. We will do this by finding minimum spanning trees on this matrix and removing edges with a weight lower than a given threshold.

In graph theory, a spanning tree is a set of edges on a graph that connects all of the nodes together. The **Minimum Spanning Tree** (**MST**) is simply the spanning tree with the lowest total weight. For our application, the nodes in our graph are samples from our dataset, and the edge weights are the number of times those two samples were clustered together—that is, the value from our coassociation matrix.

In the following figure, a MST on a graph of six nodes is shown. Nodes on the graph can be used more than once in the MST. The only criterion for a spanning tree is that all nodes should be connected together.



To compute the MST, we use SciPy's `minimum_spanning_tree` function, which is found in the `sparse` package:

```
from scipy.sparse.csgraph import minimum_spanning_tree
```

The `mst` function can be called directly on the sparse matrix returned by our coassociation function:

```
mst = minimum_spanning_tree(C)
```

However, in our coassociation matrix `C`, higher values are indicative of samples that are clustered together more often—a similarity value. In contrast, `minimum_spanning_tree` sees the input as a distance, with higher scores penalized. For this reason, we compute the minimum spanning tree on the negation of the coassociation matrix instead:

```
mst = minimum_spanning_tree(-C)
```

The result from the preceding function is a matrix the same size as the coassociation matrix (the number of rows and columns is the same as the number of samples in our dataset), with only the edges in the MST kept and all others removed.

We then remove any node with a weight less than a predefined threshold. To do this, we iterate over the edges in the MST matrix, removing any that are less than a specific value. We can't test this out with just a single iteration in a coassociation matrix (the values will be either 1 or 0, so there isn't much to work with). So, we will create extra labels first, create the coassociation matrix, and then add the two matrices together. The code is as follows:

```
pipeline.fit(documents)
labels2 = pipeline.predict(documents)
C2 = create_coassociation_matrix(labels2)
C_sum = (C + C2) / 2
```

We then compute the MST and remove any edge that didn't occur in both of these labels:

```
mst = minimum_spanning_tree(-C_sum)
mst.data[mst.data > -1] = 0
```

The threshold we wanted to cut off was any edge not in both clusterings—that is, with a value of 1. However, as we negated the coassociation matrix, we had to negate the threshold value too.

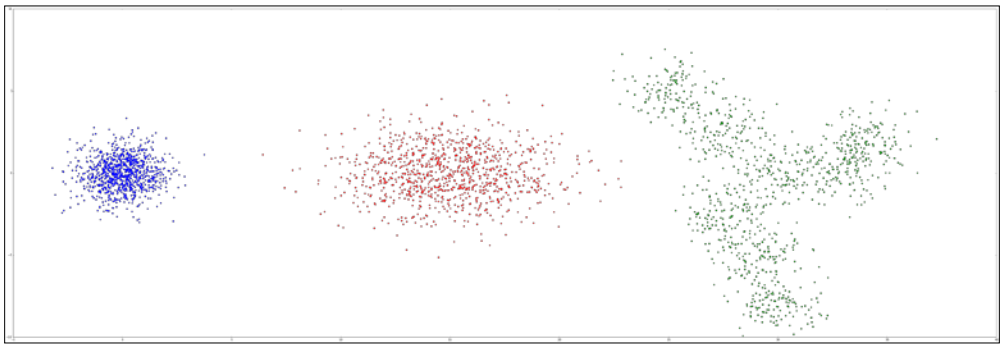
Lastly, we find all of the connected components, which is simply a way to find all of the samples that are still connected by edges after we removed the edges with low weights. The first returned value is the number of connected components (that is, the number of clusters) and the second is the labels for each sample. The code is as follows:

```
from scipy.sparse.csgraph import connected_components
number_of_clusters, labels = connected_components(mst)
```

In my dataset, I obtained eight clusters, with the clusters being approximately the same as before. This is hardly a surprise, given we only used two iterations of k-means; using more iterations of k-means (as we do in the next section) will result in more variance.

How it works

In the k-means algorithm, each feature is used without any regard to its weight. In essence, all features are assumed to be on the same scale. We saw the problems with not scaling features in *Chapter 2*, *Classifying with scikit-learn Estimators*. The result of this is that k-means is looking for circular clusters, as shown in the following screenshot:



As we can see in the preceding screenshot, not all clusters have this shape. The blue cluster is circular and is of the type that k-means is very good at picking up. The red cluster is an ellipse. The k-means algorithm can pick up clusters of this shape with some feature scaling. The third cluster isn't even convex—it is an odd shape that k-means will have trouble discovering.

The EAC algorithm works by remapping the features onto a new space, in essence turning each run of the k-means algorithm into a transformer using the same principles we saw the previous section using k-means for feature reduction. In this case, though, we only use the actual label and not the distance to each centroid. This is the data that is recorded in the co-association matrix.

The result is that EAC now only cares about how close things are to each other, not how they are placed in the original feature space. There are still issues around unscaled features. Feature scaling is important and should be done anyway (we did it using `tf-idf` in this chapter, which results in feature values having the same scale).

We saw a similar type of transformation in *Chapter 9*, *Authorship Attribution*, through the use of kernels in SVMs. These transformations are very powerful and should be kept in mind for complex datasets.

Implementation

Putting all this altogether, we can now create a simply clustering algorithm fitting the scikit-learn interface that performs all of the steps in EAC. First, we create the basic structure of the class using scikit-learn's `ClusterMixin`:

```
from sklearn.base import BaseEstimator, ClusterMixin
class EAC(BaseEstimator, ClusterMixin):
```

Our parameters are the number of k-means clusterings to perform in the first step (to create the coassociation matrix), the threshold to cut off at, and the number of clusters to find in each k-means clustering. We set a range of `n_clusters` in order to get lots of variance in our k-means iterations. Generally, in ensemble terms, variance is a good thing; without it, the solution can be no better than the individual clusterings (that said, high variance is not an indicator that the ensemble will be better). The code is as follows:

```
    def __init__(self, n_clusterings=10, cut_threshold=0.5, n_
clusters_range=(3, 10)):
        self.n_clusterings = n_clusterings
        self.cut_threshold = cut_threshold
        self.n_clusters_range = n_clusters_range
```

Next up is the `fit` function for our `EAC` class:

```
    def fit(self, X, y=None):
```

We then perform our low-level clustering using k-means and sum the resulting coassociation matrices from each iteration. We do this in a generator to save memory, creating only the coassociation matrices when we need them. In each iteration of this generator, we create a new single k-means run with our dataset and then create the coassociation matrix for it. We use `sum` to add these together. The code is as follows:

```
        C = sum((create_coassociation_matrix(self._single_
clustering(X))
                for i in range(self.n_clusterings)))
```

As before, we create the MST, remove any edges less than the given threshold (properly negating values as explained earlier), and find the connected components. As with any `fit` function in scikit-learn, we need to return self in order for the class to work in pipelines effectively. The code is as follows:

```
        mst = minimum_spanning_tree(-C)
        mst.data[mst.data > -self.cut_threshold] = 0
        self.n_components, self.labels_ = connected_components(mst)
        return self
```

We then write the function to cluster a single iteration. To do this, we randomly choose a number of clusters to find using NumPy's `randint` function and our `n_clusters_range` parameter, which sets the range of possible values. We then cluster and predict the dataset using k-means. The return value here will be the labels coming from k-means. The code is as follows:

```
    def _single_clustering(self, X):
        n_clusters = np.random.randint(*self.n_clusters_range)
        km = KMeans(n_clusters=n_clusters)
        return km.fit_predict(X)
```

We can now run this on our previous code by setting up a pipeline as before and using EAC where we previously used a `KMeans` instance as our final stage of the pipeline. The code is as follows:

```
pipeline = Pipeline([('feature_extraction', TfidfVectorizer(max_
df=0.4)),
                     ('clusterer', EAC())
                     ])
```

Online learning

In some cases, we don't have all of the data we need for training before we start our learning. Sometimes, we are waiting for new data to arrive, perhaps the data we have is too large to fit into memory, or we receive extra data after a prediction has been made. In cases like these, online learning is an option for training models over time.

An introduction to online learning

Online learning is the incremental updating of a model as new data arrives. Algorithms that support online learning can be trained on one or a few samples at a time, and updated as new samples arrive. In contrast, algorithms that are not online require access to all of the data at once. The standard k-means algorithm is like this, as are most of the algorithms we have seen so far in this module.

Online versions of algorithms have a means to partially update their model with only a few samples. Neural networks are a standard example of an algorithm that works in an online fashion. As a new sample is given to the neural network, the weights in the network are updated according to a learning rate, which is often a very small value such as 0.01. This means that any single instance only makes a small (but hopefully improving) change to the model.

Neural networks can also be trained in batch mode, where a group of samples are given at once and the training is done in one step. Algorithms are faster in batch mode, but use more memory.

In this same vein, we can slightly update the k-means centroids after a single or small batch of samples. To do this, we apply a learning rate to the centroid movement in the updating step of the k-means algorithm. Assuming that samples are randomly chosen from the population, the centroids should tend to move towards the positions they would have in the standard, offline, and k-means algorithm.

Online learning is related to streaming-based learning; however, there are some important differences. Online learning is capable of reviewing older samples after they have been used in the model, while a streaming-based machine learning algorithm typically only gets *one pass*—that is, one opportunity to look at each sample.

Implementation

The scikit-learn package contains the `MiniBatchKMeans` algorithm, which allows online learning. This class implements a `partial_fit` function, which takes a set of samples and updates the model. In contrast, calling `fit()` will remove any previous training and refit the model only on the new data.

`MiniBatchKMeans` follows the same clustering format as other algorithms in scikit-learn, so creating and using it is much the same as other algorithms.

Therefore, we can create a matrix `X` by extracting features from our dataset using `TfIDFVectorizer`, and then sample from this to incrementally update our model. The code is as follows:

```
vec = TfidfVectorizer(max_df=0.4)
X = vec.fit_transform(documents)
```

We then import `MiniBatchKMeans` and create an instance of it:

```
from sklearn.cluster import MiniBatchKMeans
mbkm = MiniBatchKMeans(random_state=14, n_clusters=3)
```

Next, we will randomly sample from our `X` matrix to simulate data coming in from an external source. Each time we get some data in, we update the model:

```
batch_size = 10
for iteration in range(int(X.shape[0] / batch_size)):
    start = batch_size * iteration
    end = batch_size * (iteration + 1)
    mbkm.partial_fit(X[start:end])
```

We can then get the labels for the original dataset by asking the instance to predict:

```
labels  = mbkm.predict(X)
```

At this stage, though, we can't do this in a pipeline as `TfIDFVectorizer` is not an online algorithm. To get over this, we use a `HashingVectorizer`. The `HashingVectorizer` class is a clever use of hashing algorithms to drastically reduce the memory of computing the bag-of-words model. Instead of recording the feature names, such as words found in documents, we record only hashes of those names. This allows us to *know* our *features* before we even look at the dataset, as it is the set of all possible hashes. This is a very large number, usually of the order 2^18. Using sparse matrices, we can quite easily store and compute even a matrix of this size, as a very large proportion of the matrix will have the value 0.

Currently, the `Pipeline` class doesn't allow for its use in online learning. There are some nuances in different applications that mean there isn't an obvious one-size-fits-all approach that could be implemented. Instead, we can create our own subclass of `Pipeline` that allows us to use it for online learning. We first derive our class from `Pipeline`, as we only need to implement a single function:

```
class PartialFitPipeline(Pipeline):
```

We create a class function `partial_fit`, which accepts an input matrix, and an optional set of classes (we don't need those for this experiment though):

```
    def partial_fit(self, X, y=None):
```

A pipeline, which we introduced before, is a set of transformations where the input to one step is the output of the previous step. To do this, we set the first input to our `X` matrix, and then go over each of the transformers to transform this data:

```
        Xt = X
        for name, transform in self.steps[:-1]:
```

We then transform our current dataset and continue iterating until we hit the final step (which, in our case will be the clustering algorithm):

```
            Xt = transform.transform(Xt)
```

We then call the `partial_fit` function on the final step and return the results:

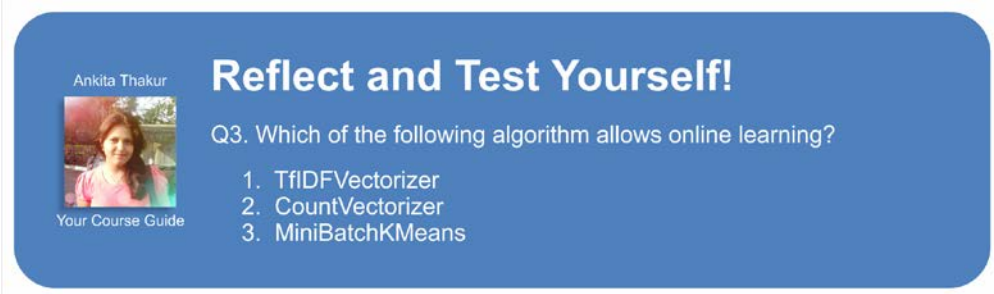
```
return self.steps[-1][1].partial_fit(Xt, y=y)
```

We can now create a pipeline to use our `MiniBatchKMeans` in online learning, alongside our `HashingVectorizer`. Other than using our new classes `PartialFitPipeline` and `HashingVectorizer`, this is the same process as used in the rest of this chapter, except we only fit on a few documents at a time. The code is as follows:

```
pipeline = PartialFitPipeline([('feature_extraction',
HashingVectorizer()),
                               ('clusterer', MiniBatchKMeans(random_
state=14, n_clusters=3))
                               ])
batch_size = 10
for iteration in range(int(len(documents) / batch_size)):
    start = batch_size * iteration
    end = batch_size * (iteration + 1)
    pipeline.partial_fit(documents[start:end])
labels = pipeline.predict(documents)
```

There are some downsides to this approach though. For one, we can't easily find out which words are most important for each cluster. We can get around this by fitting another `CountVectorizer` and taking the hash of each word. We then look up values by hash rather than word. This is a bit cumbersome and defeats the memory gains from using `HashingVectorizer`. Further, we can't use the `max_df` parameter that we used earlier, as it requires us to know what the features mean and to count them over time.

We also can't use `tf-idf` weighting when performing training online. It would be possible to approximate this and apply such weighting, but again this is a cumbersome approach. `HashingVectorizer` is still a very useful algorithm and a great use of hashing algorithms.



Your Coding Challenge

The evaluation of clustering algorithms is a difficult problem—on the one hand, we can sort of tell what good clusters look like; on the other hand, if we really know that, we should label some instances and use a supervised classifier! Much has been written on this topic. One slideshow on the topic that is a good introduction to the challenges is available at `http://www.cs.kent.edu/~jin/DM08/ClusterValidation.pdf`.

In addition, a very comprehensive (although now a little dated) paper on this topic is here: `http://web.itu.edu.tr/sgunduz/courses/verimaden/paper/validity_survey.pdf`.

Ankita Thakur

Your Course Guide

The scikit-learn package does implement a number of the metrics described in those links, with an overview here: `http://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation`.

Using some of these, you can start evaluating which parameters need to be used for better clusterings. Using a Grid Search, we can find parameters that maximize a metric—just like in classification.

Second challenge

The code we developed in this Lesson can be rerun over many months. By adding some tags to each cluster, you can track which topics stay active over time, getting a longitudinal viewpoint of what is being discussed in the world news.

To compare the clusters, consider a metric such as the adjusted mutual information score, which was linked to the scikit-learn documentation earlier. See how the clusters change after one month, two months, six months, and a year.

Summary of Module 3 Chapter 10

In this chapter, we looked at clustering, which is an unsupervised learning approach. We use unsupervised learning to explore data, rather than for classification and prediction purposes. In the experiment here, we didn't have topics for the news items we found on reddit, so we were unable to perform classification. We used k-means clustering to group together these news stories to find common topics and trends in the data.

In pulling data from reddit, we had to extract data from arbitrary websites. This was performed by looking for large text segments, rather than a full-blown machine learning approach. There are some interesting approaches to machine learning for this task that may improve upon these results. In the last chapter of this module, I've listed, for each chapter, avenues for going beyond the scope of the chapter and improving upon the results. This includes references to other sources of information and more difficult applications of the approaches in each chapter.

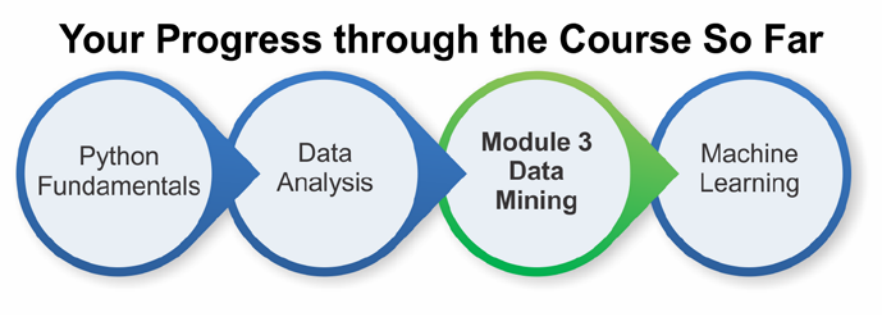
Ankita Thakur

Your Course Guide

We also looked at a straightforward ensemble algorithm, ECA. An ensemble is often a good way to deal with variance in the results, especially if you don't know how to choose good parameters (which is especially difficult with clustering).

Finally, we introduced online learning. This is a gateway to larger learning exercises, including Big data, which will be discussed in the final two chapters of this module. These final experiments are quite large and require management of data as well as learning a model from them.

In the next chapter, we step away from unsupervised learning and go back to classification. We will look at deep learning, which is a classification method built on complex neural networks.


Your Progress through the Course So Far
Python Fundamentals
Data Analysis
Module 3 Data Mining
Machine Learning

11
Classifying Objects in Images Using Deep Learning

We used basic neural networks in *Chapter 8*, *Beating CAPTCHAs with Neural Networks*. A recent flood of research in the area has led to a number of significant advances to that base design. Today, research in neural networks is creating some of the most advanced and accurate classification algorithms in many areas.

These advances have come on the back of improvements in computational power, allowing us to train larger and more complex networks. However, the advances are much more than simply throwing more computational power at the problem. New algorithms and layer types have drastically improved performance, outside computational power.

In this chapter, we will look at determining what object is represented in an image. The pixel values will be used as input, and the neural network will then automatically find useful combinations of pixels to form higher-level features. These will then be used for the actual classification. Overall, in this chapter, we will examine the following:

- Classifying objects in images
- The different types of deep neural networks
- Theano, Lasagne, and nolearn; libraries to build and train neural networks
- Using a GPU to improve the speed of the algorithms

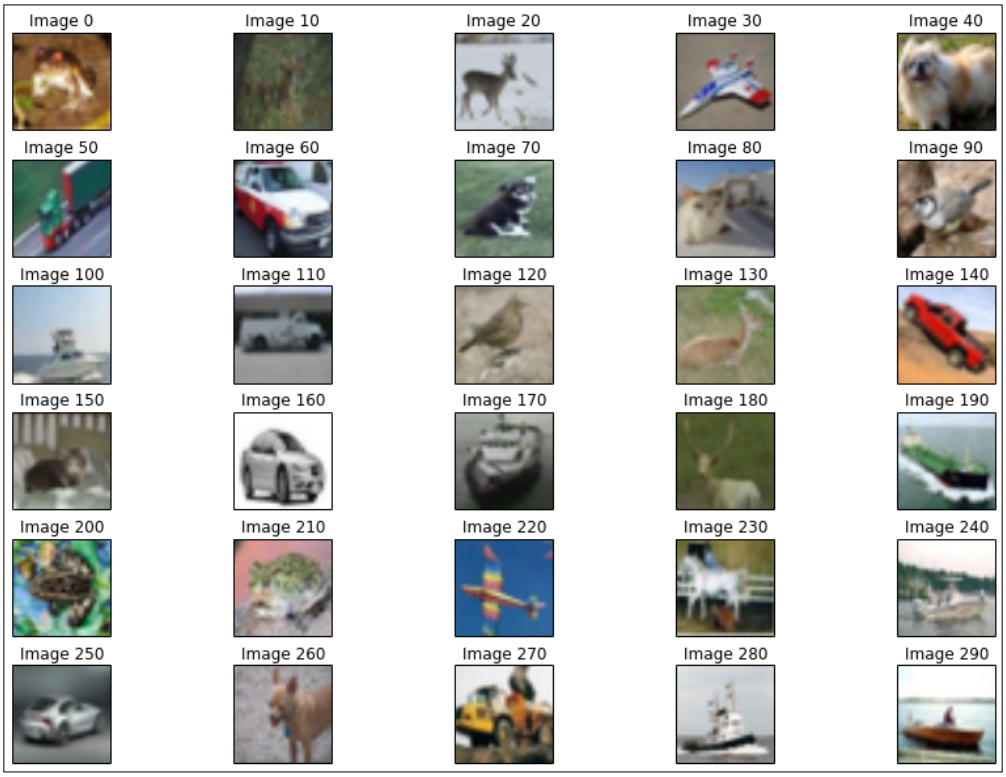
Object classification

Computer vision is becoming an important part of future technology. For example, we will have access to self-driving cars in the next five years (possibly much sooner, if some rumors are to be believed). In order to achieve this, the car's computer needs to be able to see around it: obstacles, other traffic, and weather conditions.

While we can easily detect whether there is an obstacle, for example using radar, it is also important we know what that object is. If it is an animal, it may move out of the way; if it is a building, it won't move at all and we need to go around it.

Application scenario and goals

In this chapter, we will build a system that will take an image as an input and give a prediction on what the object in it is. We will take on the role of a vision system for a car, looking around at any obstacles in the way or on the side of the road. Images are of the following form:



This dataset comes from a popular dataset called **CIFAR-10**. It contains 60,000 images that are 32 pixels wide and 32 pixels high, with each pixel having a red-green-blue (RGB) value. The dataset is already split into training and testing, although we will not use the testing dataset until after we complete our training.

The CIFAR-10 dataset is available for download at: `http://www.cs.toronto.edu/~kriz/cifar.html`. Download the python version, which has already been converted to NumPy arrays.

Opening a new IPython Notebook, we can see what the data looks like. First, we set up the data filenames. We will only worry about the first batch to start with, and scale up to the full dataset size towards the end;

```
import os
data_folder = os.path.join(os.path.expanduser("~"), "Data", "cifar-10-
batches-py")
batch1_filename = os.path.join(data_folder, "data_batch_1")
```

Next, we create a function that can read the data stored in the batches. The batches have been saved using pickle, which is a python library to save objects. Usually, we can just call `pickle.load` on the file to get the object. However, there is a small issue with this data: it was saved in Python 2, but we need to open it in Python 3. In order to address this, we set the encoding to *latin* (even though we are opening it in byte mode):

```
import pickle
# Bigfix thanks to: http://stackoverflow.com/questions/11305790/
pickle-incompatability-of-numpy-arrays-between-python-2-and-3
def unpickle(filename):
    with open(filename, 'rb') as fo:
        return pickle.load(fo, encoding='latin1')
```

Using this function, we can now load the batch dataset:

```
batch1 = unpickle(batch1_filename)
```

This batch is a dictionary, containing the actual data in NumPy arrays, the corresponding labels and filenames, and finally a note to say which batch it is (this is *training batch 1 of 5*, for instance).

We can extract an image by using its index in the batch's data key:

```
image_index = 100
image = batch1['data'][image_index]
```

The image array is a NumPy array with 3,072 entries, from 0 to 255. Each value is the red, green, or blue intensity at a specific location in the image.

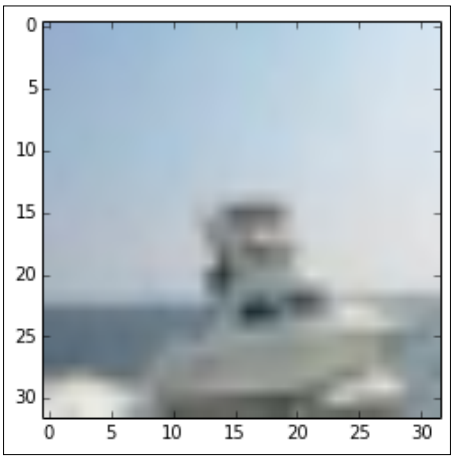
The images are in a different format than what matplotlib usually uses (to display images), so to show the image we first need to reshape the array and rotate the matrix. This doesn't matter so much to train our neural network (we will define our network in a way that fits with the data), but we do need to convert it for matplotlib's sake:

```
image = image.reshape((32,32, 3), order='F')
import numpy as np
image = np.rot90(image, -1)
```

Now we can show the image using matplotlib:

```
%matplotlib inline

from matplotlib import pyplot as plt
plt.imshow(image)
```

The resulting image, a boat, is displayed:



The resolution on this image is quite poor—it is only 32 pixels wide and 32 pixels high. Despite that, most people will look at the image and see a boat. Can we get a computer to do the same?

You can change the image index to show different images, getting a feel for the dataset's properties.

The aim of our project, in this chapter, is to build a classification system that can take an image like this and predict what the object in it is.

Use cases

Computer vision is used in many scenarios.

Online map websites, such as Google Maps, use computer vision for a number of reasons. One reason is to automatically blur any faces that they find, in order to give some privacy to the people being photographed as part of their Street View feature.

Face detection is also used in many industries. Modern cameras automatically detect faces, as a means to improve the quality of photos taken (the user most often wants to focus on a visible face). Face detection can also be used for identification. For example, Facebook automatically recognizes people in photos, allowing for easy tagging of friends.

As we stated before, autonomous vehicles are highly dependent on computer vision to recognize their path and avoid obstacles. Computer vision is one of the key problems that is being addressed not only in research into autonomous vehicles, not just for consumer use, but also in mining and other industries.

Other industries are using computer vision too, including warehouses examining goods automatically for defects.

The space industry is also using computer vision, helping to automate the collection of data. This is critical for effective use of spacecraft, as sending a signal from Earth to a rover on Mars can take a long time and is not possible at certain times (for instance, when the two planets are not facing each other). As we start dealing with space-based vehicles more frequently, and from a greater distance, increasing the autonomy of these spacecrafts is absolutely necessary.

The following screenshot shows the Mars rover designed and used by NASA; it made significant use of computer vision:

Deep neural networks

The neural networks we used in *Chapter 8*, *Beating CAPTCHAs with Neural Networks*, have some fantastic theoretical properties. For example, only a single hidden layer is needed to learn any mapping (although the size of the middle layer may need to be very, very big). Neural networks were a very active area of research in the 1970s and 1980s, and then these networks were no longer used, particularly compared to other classification algorithms such as support vector machines. One of the main issues was that the computational power needed to run many neural networks was more than other algorithms and more than what many people had access to.

Another issue was training the networks. While the back propagation algorithm has been known about for some time, it has issues with larger networks, requiring a very large amount of training before the weights settle.

Each of these issues has been addressed in recent times, leading to a resurgence in popularity of neural networks. Computational power is now much more easily available than 30 years ago, and advances in algorithms for training mean that we can now readily use that power.

Intuition

The aspect that differentiates deep neural networks from the more basic neural network we saw in *Chapter 8*, *Beating CAPTCHAs with Neural Networks*, is size. A neural network is considered deep when it has two or more hidden layers. In practice, a deep neural network is often much larger, both in the number of nodes in each layer and also the number of layers. While some of the research of the mid-2000s focused on very large numbers of layers, smarter algorithms are reducing the actual number of layers needed.

A neural network basically takes very basic features as inputs—in the case of computer vision, it is simple pixel values. Then, as that data is combined and pushed through the network, these basic features combine into more complex features. Sometimes, these features have little meaning to humans, but they represent the aspects of the sample that the computer looks for to make its classification.

Implementation

Implementing these deep neural networks can be quite challenging due to their size. A bad implementation will take significantly longer to run than a good one, and may not even run at all due to memory usage.

A basic implementation of a neural network might start by creating a node class and collecting a set of these into a layer class. Each node is then connected to a node in the next layer using an instance of an Edge class. This type of implementation, a class-based one, is good to show how networks work, but is too inefficient for larger networks.

Neural networks are, at their core, simply mathematical expressions on matrices. The weights of the connections between one network and the next can be represented as a matrix of values, where the rows represent nodes in the first layer and the columns represent the nodes in the second layer (the transpose of this matrix is used sometimes too). The value is the weight of the edge between one layer and the next. A network can then be defined as a set of these weight matrices. In addition to the nodes, we add a bias term to each layer, which is basically a node that is always on and connected to each neuron in the next layer.

This insight allows us to use mathematical operations to build, train, and use neural networks, as opposed to creating a class-based implementation. These mathematical operations are great, as many great libraries of highly optimized code have been written that we can use to perform these computations as efficiently as we can.

The `PyBrain` library that we used in *Chapter 8*, *Beating CAPTCHAs with Neural Networks*, does contain a simple convolutional layer for a neural network. However, it doesn't offer us some of the features that we need for this application. For larger and more customized networks, though, we need a library that gives us a bit more power. For this reason, we will be using the `Lasagne` and `nolearn` libraries. This library works on the `Theano` library, which is a useful tool for mathematical expressions.

In this chapter, we will start by implementing a basic neural network with `Lasagne` to introduce the concepts. We will then use `nolearn` to replicate our experiment in *Chapter 8*, *Beating CAPTCHAs with Neural Networks,* on predicting which letter is in an image. Finally, we will use a much more complex convolution neural network to perform image classification on the CIFAR dataset, which will also include running this on GPUs rather than CPUs to improve the performance.

An introduction to Theano

Theano is a library that allows you to build mathematical expressions and run them. While this may immediately not seem that different to what we normally do to write a program, in Theano, we define the function we want to perform, not the way in which it is computed. This allows Theano to optimize the evaluation of the expression and also to perform lazy computation—expressions are only actually computed when they are needed, not when they are defined.

Many programmers don't use this type of programming day-to-day, but most of them interact with a related system that does. Relational databases, specifically SQL-based ones, use this concept called the declarative paradigm. While a programmer might define a *SELECT* query on a database with a *WHERE* clause, the database interprets that and creates an optimized query based on a number of factors, such as whether the *WHERE* clause is on a primary key, the format the data is stored in, and other factors. The programmer defines what they want and the system determines how to do it.

You can install Theano using pip: `pip3 install Theano`.

Using Theano, we can define many types of functions working on scalars, arrays, and matrices, as well as other mathematical expressions. For instance, we can create a function that computes the length of the hypotenuse of a right-angled triangle:

```
import theano
from theano import tensor as T
```

First, we define the two inputs, a and b. These are simple numerical values, so we define them as scalars:

```
a = T.dscalar()
b = T.dscalar()
```

Then, we define the output, `c`. This is an expression based on the values of `a` and `b`:

```
c = T.sqrt(a ** 2 + b ** 2)
```

Note that `c` isn't a function or a value here—it is simply an expression, given `a` and `b`. Note also that `a` and `b` don't have actual values—this is an algebraic expression, not an absolute one. In order to compute on this, we define a function:

```
f = theano.function([a,b], c)
```

This basically tells Theano to create a function that takes values for `a` and `b` as inputs, and returns `c` as an output, computed on the values given. For example, `f(3, 4)` returns `5`.

While this simple example may not seem much more powerful than what we can already do with Python, we can now use our function or our mathematical expression `c` in other parts of code and the remaining mappings. In addition, while we defined `c` before the function was defined, no actual computation was done until we called the function.

An introduction to Lasagne

Theano isn't a library to build neural networks. In a similar way, NumPy isn't a library to perform machine learning; it just does the heavy lifting and is generally used from another library. Lasagne is such a library, designed specifically around building neural networks, using Theano to perform the computation.

Lasagne implements a number of modern types of neural network layers, and the building blocks for building them.

These include the following:

- **Network-in-network layers**: These are small neural networks that are easier to interpret than traditional neural network layers.
- **Dropout layers**: These randomly drop units during training, preventing overfitting, which is a major problem in neural networks.
- **Noise layers**: These introduce noise into the neurons; again, addressing the overfitting problem.

In this chapter, we will use `convolution layers` (layers that are organized to mimic the way in which human vision works). They use small collections of connected neurons that analyze only a segment of the input values (in this case, an image). This allows the network to deal with standard alterations such as dealing with translations of images. In the case of vision-based experiments, an example of an alteration dealt with by convolution layers is translating the image.

In contrast, a traditional neural network is often heavily connected—all neurons from one layer connect to all neurons in the next layer.

Convolutional networks are implemented in the `lasagne.layers.Conv1DLayer` and `lasagne.layers.Conv2DLayer` classes.

At the time of writing, Lasagne hasn't had a formal release and is not on `pip`. You can install it from `github`. In a new folder, download the source code repository using the following:

```
git clone https://github.com/Lasagne/Lasagne.git
```

From within the created Lasagne folder, you can then install the library using the following:

```
sudo python3 setup.py install
```

See `http://lasagne.readthedocs.org/en/latest/user/installation.html` for installation instructions.

Neural networks use convolutional layers (generally, just *Convolutional Neural Networks*) and also the `pooling` layers, which take the maximum output for a certain region. This reduces noise caused by small variations in the image, and reduces (or down-samples) the amount of information. This has the added benefit of reducing the amount of work needed to be done in later layers.

Lasagne also implements these pooling layers—for example in the `lasagne.layers.MaxPool2DLayer` class. Together with the convolution layers, we have all the tools needed to build a convolution neural network.

Building a neural network in Lasagne is easier than building it using just Theano. To show the principles, we will implement a basic network to lean on the Iris dataset, which we saw in *Chapter 1*, *Getting Started with Data Mining*. The Iris dataset is great for testing new algorithms, even complex ones such as deep neural networks.

First, open a new IPython Notebook. We will come back to the Notebook we loaded the CIFAR dataset with, later in the chapter.

First, we load the dataset:

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data.astype(np.float32)
y_true = iris.target.astype(np.int32)
```

Due to the way Lasagne works, we need to be a bit more explicit about the data types. This is why we converted the classes to `int32` (they are stored as `int64` in the original dataset).

We then split into training and testing datasets:

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y_true, random_
state=14)
```

Next, we build our network by creating the different layers. Our dataset contains four input variables and three output classes. This gives us the size of the first and last layer, but not the layers in between. Playing around with this figure will give different results, and it is worth trailing different values to see what happens.

We start by creating an input layer, which has the same number of nodes as the dataset. We can specify a batch size (where the value is 10), which allows Lasagne to do some optimizations in training:

```
import lasagne
input_layer = lasagne.layers.InputLayer(shape=(10, X.shape[1]))
```

Next, we create our hidden layer. This layer takes its input from our input layer (specified as the first argument), which has 12 nodes, and uses the sigmoid nonlinearity, which we saw in *Chapter 8*, *Beating CAPTCHAs with Neural Networks*;

```
hidden_layer = lasagne.layers.DenseLayer(input_layer, num_units=12,
nonlinearity=lasagne.nonlinearities.sigmoid)
```

Next, we have our output layer that takes its input from the hidden layer, which has three nodes (which is the same as the number of classes), and uses the **softmax** nonlinearity. Softmax is more typically used in the final layer of neural networks:

```
output_layer = lasagne.layers.DenseLayer(hidden_layer, num_units=3,
                                         nonlinearity=lasagne.
nonlinearities.softmax)
```

In Lasagne's usage, this output layer is our `network`. When we enter a sample into it, it looks at this output layer and obtains the layer that is inputted into it (the first argument). This continues recursively until we reach an input layer, which applies the samples to itself, as it doesn't have an input layer to it. The activations of the neurons in the input layer are then fed into its calling layer (in our case, the `hidden_layer`), and that is then propagated up all the way to the output layer.

In order to train our network, we now need to define some training functions, which are Theano-based functions. In order to do this, we need to define a Theano expression and a function for the training. We start by creating variables for the input samples, the output given by the network, and the actual output:

```
import theano.tensor as T
net_input = T.matrix('net_input')
net_output = output_layer.get_output(net_input)
true_output = T.ivector('true_output')
```

We can now define our loss function, which tells the training function how to improve the network—it attempts to train the network to minimize the loss according to this function. The loss we will use is the categorical cross entropy, a metric on categorical data such as ours. This is a function of the output given by the network and the actual output we expected:

```
loss = T.mean(T.nnet.categorical_crossentropy(net_output,
  true_output))
```

Next, we define the function that will change the weights in our network. In order to do this, we obtain all of the parameters from the network and create a function (using a helper function given by Lasagne) that adjusts the weights to minimize our loss;

```
all_params = lasagne.layers.get_all_params(output_layer)
updates = lasagne.updates.sgd(loss, all_params, learning_rate=0.1)
```

Finally, we create Theano-based functions that perform this training and also obtain the output of the network for testing purposes:

```
import theano
train = theano.function([net_input, true_output], loss,
updates=updates)
get_output = theano.function([net_input], net_output)
```

We can then call our train function, on our training data, to perform one iteration of training the network. This involves taking each sample, computing the predicted class of it, comparing those predictions to the expected classes, and updating the weights to minimize the loss function. We then perform this 1,000 times, incrementally training our network over those iterations:

```
for n in range(1000):
    train(X_train, y_train)
```

Next, we can evaluate by computing the F-score on the outputs. First, we obtain those outputs:

```
y_output = get_output(X_test)
```

Note that `get_output` is a Theano function we obtained from our neural network, which is why we didn't need to add our network as a parameter to this line of code.

This result, `y_output`, is the activation of each of the neurons in the final output layer. The actual prediction itself is created by finding which neuron has the highest activation:

```
import numpy as np
y_pred = np.argmax(y_output, axis=1)
```

Now, `y_pred` is an array of class predictions, like we are used to in classification tasks. We can now compute the F-score using these predictions:

```
from sklearn.metrics import f1_score
print(f1_score(y_test, y_pred))
```

The result is impressively perfect—1.0! This means all the classifications were correct in the test data: a great result (although this is a simpler dataset).

As we can see, while it is possible to develop and train a network using just Lasagne, it can be a little awkward. To address this, we will be using `nolearn`, which is a package that further wraps this process in code that is conveniently convertible with the scikit-learn API.

Implementing neural networks with nolearn

The `nolearn` package provides wrappers for Lasagne. We lose some of the fine-tuning that can go with building a neural network by hand in Lasagne, but the code is much more readable and much easier to manage.

The `nolearn` package implements the normal sorts of complex neural networks you are likely to want to build. If you want more control than `nolearn` gives you, you can revert to using Lasagne, but at the cost of having to manage a bit more of the training and building process.

To get started with `nolearn`, we are going to reimplement the example we used in *Chapter 8*, *Beating CAPTCHAs with Neural Networks*, to predict which letter was represented in an image. We will recreate the dense neural network we used in *Chapter 8*, *Beating CAPTCHAs with Neural Networks*. To start with, we need to enter our dataset building code again in our notebook. For a description of what this code does, refer to *Chapter 8*, *Beating CAPTCHAs with Neural Networks*:

```
import numpy as np
from PIL import Image, ImageDraw, ImageFont
from skimage.transform import resize
from skimage import transform as tf
from skimage.measure import label, regionprops
from sklearn.utils import check_random_state
from sklearn.preprocessing import OneHotEncoder
from sklearn.cross_validation import train_test_split

def create_captcha(text, shear=0, size=(100, 24)):
    im = Image.new("L", size, "black")
    draw = ImageDraw.Draw(im)
    font = ImageFont.truetype(r"Coval.otf", 22)
    draw.text((2, 2), text, fill=1, font=font)
    image = np.array(im)
    affine_tf = tf.AffineTransform(shear=shear)
    image = tf.warp(image, affine_tf)
    return image / image.max()

def segment_image(image):
    labeled_image = label(image > 0)
    subimages = []
    for region in regionprops(labeled_image):
        start_x, start_y, end_x, end_y = region.bbox
        subimages.append(image[start_x:end_x,start_y:end_y])
    if len(subimages) == 0:
```

```
            return [image,]
        return subimages

    random_state = check_random_state(14)
    letters = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
    shear_values = np.arange(0, 0.5, 0.05)

    def generate_sample(random_state=None):
        random_state = check_random_state(random_state)
        letter = random_state.choice(letters)
        shear = random_state.choice(shear_values)
        return create_captcha(letter, shear=shear, size=(20, 20)),
    letters.index(letter)
    dataset, targets = zip(*(generate_sample(random_state) for i in
    range(3000)))
    dataset = np.array(dataset, dtype='float')
    targets =  np.array(targets)

    onehot = OneHotEncoder()
    y = onehot.fit_transform(targets.reshape(targets.shape[0],1))
    y = y.todense().astype(np.float32)

    dataset = np.array([resize(segment_image(sample)[0], (20, 20)) for
    sample in dataset])
    X = dataset.reshape((dataset.shape[0], dataset.shape[1] * dataset.
    shape[2]))
    X = X / X.max()
    X = X.astype(np.float32)

    X_train, X_test, y_train, y_test = \
        train_test_split(X, y, train_size=0.9, random_state=14)
```

A neural network is a collection of layers. Implementing one in `nolearn` is a case of organizing what those layers will look like, much as it was with PyBrain. The neural network we used in *Chapter 8, Beating CAPTCHAs with Neural Networks*, used fully connected dense layers. These are implemented in `nolearn`, meaning we can replicate our basic network structure here. First, we create the layers consisting of an input layer, our dense hidden layer, and our dense output layer:

```
from lasagne import layers
layers=[
        ('input', layers.InputLayer),
        ('hidden', layers.DenseLayer),
        ('output', layers.DenseLayer),
        ]
```

We then import some requirements, which we will explain as we use them:

```
from lasagne import updates
from nolearn.lasagne import NeuralNet
from lasagne.nonlinearities import sigmoid, softmax
```

Next we define the neural network, which is represented as a scikit-learn-compatible estimator:

```
net1 = NeuralNet(layers=layers,
```

Note that we haven't closed off the parenthesis—this is deliberate. At this point, we enter the parameters for the neural network, starting with the size of each layer:

```
    input_shape=X.shape,
    hidden_num_units=100,
    output_num_units=26,
```

The parameters here match the layers. In other words, the `input_shape` parameter first finds the layer in our layers that has given the name input, working much the same way as setting parameters in pipelines.

Next, we define the nonlinearities. Again, we will use `sigmoid` for the hidden layer and `softmax` for the output layer:

```
    hidden_nonlinearity=sigmoid,
    output_nonlinearity=softmax,
```

Next, we will use bias nodes, which are nodes that are always turned on in the hidden layer. Bias nodes are important to train a network, as they allow for the activations of neurons to train more specifically to their problems. As an oversimplified example, if our prediction is always off by 4, we can add a bias of -4 to remove this bias. Our bias nodes allow for this, and the training of the weights dictates the amount of bias that is used.

The biases are given as a set of weights, meaning that it needs to be the same size as the layer the bias is attaching to:

```
    hidden_b=np.zeros((100,), dtype=np.float32),
```

Next, we define how the network will train. The `nolearn` package doesn't have the exact same training mechanism as we used in *Chapter 8*, *Beating CAPTCHAs with Neural Networks*, as it doesn't have a way to decay weights. However, it does have momentum, which we will use, along with a high learning rate and low momentum value:

```
        update=updates.momentum,
        update_learning_rate=0.9,
        update_momentum=0.1,
```

Next, we define the problem as a regression problem. This may seem odd, as we are performing a classification task. However, the outputs are real-valued, and optimizing them as a regression problem appears to do much better in training than trying to optimize on classification:

```
        regression=True,
```

Finally, we set the maximum number of epochs for training at 1,000, which is a good fit between good training and not taking a long time to train (for this dataset; other datasets may require more or less training):

```
        max_epochs=1000,
```

We can now close off the parenthesis for the neural network constructor;

```
    )
```

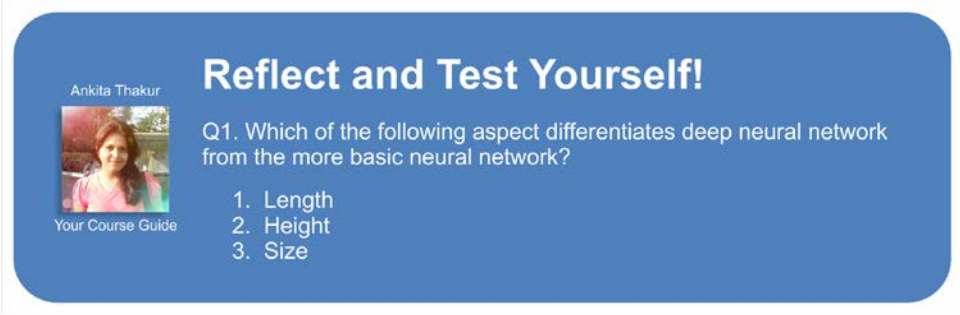
Next, we train the network on our training dataset:

```
    net1.fit(X_train, y_train)
```

Now we can evaluate the trained network. To do this, we get the output of our network and, as with the Iris example, we need to perform an `argmax` to get the actual classification by choosing the highest activation:

```
    y_pred = net1.predict(X_test)
    y_pred = y_pred.argmax(axis=1)
    assert len(y_pred) == len(X_test)
    if len(y_test.shape) > 1:
        y_test = y_test.argmax(axis=1)
    print(f1_score(y_test, y_pred))
```

The results are equally impressive—another perfect score on my machine. However, your results may vary as the `nolearn` package has some randomness that can't be directly controlled at this stage.



GPU optimization

Neural networks can grow quite large in size. This has some implications for memory use; however, efficient structures such as sparse matrices mean that we don't generally run into problems fitting a neural network in memory.

The main issue when neural networks grow large is that they take a very long time to compute. In addition, some datasets and neural networks will need to run many epochs of training to get a good fit for the dataset. The neural network we will train in this chapter takes more than 8 minutes per epoch on my reasonably powerful computer, and we expect to run dozens, potentially hundreds, of epochs. Some larger networks can take hours to train a single epoch. To get the best performance, you may be considering thousands of training cycles.

The math obviously doesn't give a nice result here.

One positive is that neural networks are, at their core, full of floating point operations. There are also a large number of operations that can be performed in parallel, as neural network training is composed of mainly matrix operations. These factors mean that computing on GPUs is an attractive option to speed up this training.

When to use GPUs for computation

GPUs were originally designed to render graphics for display. These graphics are represented using matrices and mathematical equations on those matrices, which are then converted into the pixels that we see on our screen. This process involves lots of computation in parallel. While modern CPUs may have a number of cores (your computer may have 2, 4, or even 16—or more!), GPUs have thousands of small cores designed specifically for graphics.

A CPU is therefore better for sequential tasks, as the cores tend to be individually faster and tasks such as accessing the computer's memory are more efficient. It is also, honestly, easier to just let the CPU do the heavy lifting. Almost every machine learning library defaults to using the CPU, and there is extra work involved before you can use the GPU for computing. The benefits though, can be quite significant.

GPUs are therefore better suited for tasks in which there are lots of small operations on numbers that can be performed at the same time. Many machine learning tasks are like this, lending themselves to efficiency improvements through the use of a GPU.

Getting your code to run on a GPU can be a frustrating experience. It depends greatly on what type of GPU you have, how it is configured, your operating system, and whether you are prepared to make some low-level changes to your computer.

There are three main avenues to take:

- The first is to look at your computer, search for tools and drivers for your GPU and operating system, explore some of the many tutorials out there, and find one that fits your scenario. Whether this works depends on what your system is like. That said, this scenario is much easier than it was a few years ago, with better tools and drivers available to perform GPU-enabled computation.
- The second avenue is to choose a system, find good documentation on setting it up, and buy a system to match. This will work better, but can be fairly expensive—in most modern computers, the GPU is one of the most expensive parts. This is especially true if you want to get great performance out of the system—you'll need a really good GPU, which can be very expensive.

- The third avenue is to use a virtual machine, which is already configured for such a purpose. For example, Markus Beissinger has created such a system that runs on Amazon's Web Services. The system will cost you money to run, but the price is much less than that of a new computer. Depending on your location, the exact system you get and how much you use it, you are probably looking at less than $1 an hour, and often much, much less. If you use spot instances in Amazon's Web Services, you can run them for just a few cents per hour (although, you will need to develop your code to run on spot instances separately).

If you aren't able to afford the running costs of a virtual machine, I recommend that you look into the first avenue, with your current system. You may also be able to pick up a good secondhand GPU from family or a friend who constantly updates their computer (gamer friends are great for this!).

Running our code on a GPU

We are going to take the third avenue in this chapter and create a virtual machine based on *Markus Beissinger's* base system. This will run on an Amazon's EC2 service. There are many other Web services to use, and the procedure will be slightly different for each. In this section, I'll outline the procedure for Amazon.

If you want to use your own computer and have it configured to run GPU-enabled computation, feel free to skip this section.

You can get more information on how this was set up, which may also provide information on setting it up on another computer, at `http://markus.com/install-theano-on-aws/`.

To start with, go to the AWS console at:

`https://console.aws.amazon.com/console/home?region=us-east-1`

Log in with your Amazon account. If you don't have one, you will be prompted to create one, which you will need to do in order to continue.

Next, go to the EC2 service console at: `https://console.aws.amazon.com/ec2/v2/home?region=us-east-1`.

Click on **Launch Instance** and choose **N. California** as your location in the drop-down menu at the top-right.

Click on **Community AMIs** and search for `ami-b141a2f5`, which is the machine created by Markus Beissinger. Then, click on **Select**. On the next screen, choose **g2.2xlarge** as the machine type and click on **Review and Launch**. On the next screen, click on **Launch**.

At this point, you will be charged, so please remember to shut down your machines when you are done with them. You can go to the EC2 service, select the machine, and stop it. You won't be charged for machines that are not running.

You'll be prompted with some information on how to connect to your instance. If you haven't used AWS before, you will probably need to create a new key pair to securely connect to your instance. In this case, give your key pair a name, download the `pem` file, and store it in a safe place—if lost, you will not be able to connect to your instance again!

Click on **Connect** for information on using the `pem` file to connect to your instance. The most likely scenario is that you will use `ssh` with the following command:

```
ssh -i <certificante_name>.pem ubuntu@<server_ip_address>
```

Setting up the environment

When you have connected to the instance, you can install the updated `Lasagne` and `nolearn` packages.

First, clone the `git` repository for `Lasagne`, as was outlined earlier in this chapter:

```
git clone https://github.com/Lasagne/Lasagne.git
```

In order to build this library on this machine, we will need `setuptools` for Python 3, which we can install via `apt-get`, which is Ubuntu's method of installing applications and libraries; we also need the development library for NumPy. Run the following in the command line of the virtual machine:

```
sudo apt-get install python3-pip python3-numpy-dev
```

Next, we install Lasagne. First, we change to the source code directory and then run `setup.py` to build and install it:

```
cd Lasagne
sudo python3 setup.py install
```

We have installed `Lasagne` and will install `nolearn` as system-wide packages for simplicity. For those wanting a more portable solution, I recommend using `virtualenv` to install these packages. It will allow you to use different python and library versions on the same computer, and make moving the code to a new computer much easier. For more information, see `http://docs.python-guide.org/en/latest/dev/virtualenvs/`.

After Lasagne is built, we can now install `nolearn`. Change to the home directory and follow the same procedure, except for the `nolearn` package:

```
cd ~/
git clone https://github.com/dnouri/nolearn.git
cd nolearn
sudo python3 setup.py install
```

Our system is nearly set up. We need to install scikit-learn and scikit-image, as well as matplotlib. We can do all of this using `pip3`. As a dependency on these, we need the `scipy` and `matplotlib` packages as well, which aren't currently installed on this machine. I recommend using scipy and matplotlib from `apt-get` rather than `pip3`, as it can be painful in some cases to install it using `pip3`:

```
sudo apt-get install python3-scipy python3-matplotlib
sudo pip3 install scikit-learn scikit-image
```

Next, we need to get our code onto the machine. There are many ways to get this file onto your computer, but one of the easiest is to just copy-and-paste the contents.

To start with, open the IPython Notebook we used before (on your computer, not on the Amazon Virtual Machine). On the Notebook itself is a menu. Click on **File** and then **Download as**. Select **Python** and save it to your computer. This procedure downloads the code in the IPython Notebook as a python script that you can run from the command line.

Open this file (on some systems, you may need to right-click and open with a text editor). Select all of the contents and copy them to your clipboard.

On the Amazon Virtual Machine, move to the home directory and open `nano` with a new filename:

```
cd ~/
nano chapter11script.py
```

The `nano` program will open, which is a command-line text editor.

With this program open, paste the contents of your clipboard into this file. On some systems, you may need to use a file option of the ssh program, rather than pressing *Ctrl* + *V* to paste.

In `nano`, press *Ctrl* + *O* to save the file on the disk and then *Ctrl* + *X* to exit the program.

You'll also need the font file. The easiest way to do this is to download it again from the original location. To do this, enter the following:

```
wget http://openfontlibrary.org/assets/downloads/bretan/680bc56bbeeca9535
3ede363a3744fdf/bretan.zip
sudo apt-get install unzip
unzip -p bretan.zip Coval.otf > Coval.otf
```

This will unzip only one `Coval.otf` file (there are lots of files in this zip folder that we don't need).

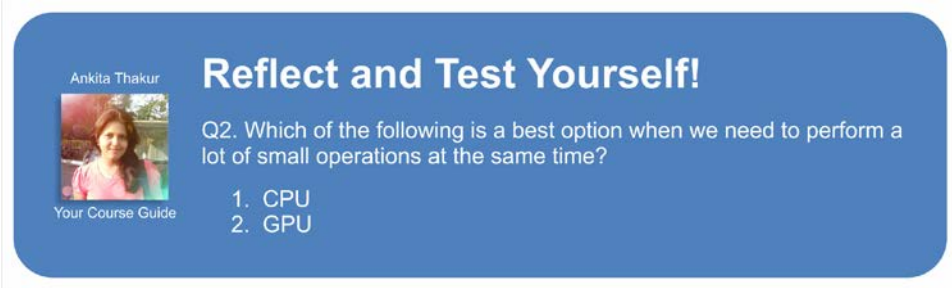
While still in the virtual machine, you can run the program with the following command:

```
python3 chapter11script.py
```

The program will run through as it would in the IPython Notebook and the results will print to the command line.

The results should be the same as before, but the actual training and testing of the neural network will be much faster. Note that it won't be that much faster in the other aspects of the program—we didn't write the CAPTCHA dataset creation to use a GPU, so we will not obtain a speedup there.

You may wish to shut down the Amazon virtual machine to save some money; we will be using it at the end of this chapter to run our main experiment, but will be developing the code on your main computer first.



Application

Back on your main computer now, open the first IPython Notebook we created in this chapter—the one that we loaded the CIFAR dataset with. In this major experiment, we will take the CIFAR dataset, create a deep convolution neural network, and then run it on our GPU-based virtual machine.

Getting the data

To start with, we will take our CIFAR images and create a dataset with them. Unlike previously, we are going to preserve the pixel structure—that is,. in rows and columns. First, load all the batches into a list:

```
import numpy as np
batches = []
for i in range(1, 6):
    batch_filename = os.path.join(data_folder, "data_batch_{}".
format(i))
    batches.append(unpickle(batch1_filename))
    break
```

The last line, the break, is to test the code—this will drastically reduce the number of training examples, allowing you to quickly see if your code is working. I'll prompt you later to remove this line, after you have tested that the code works.

Next, create a dataset by stacking these batches on top of each other. We use NumPy's `vstack`, which can be visualized as adding rows to the end of the array:

```
X = np.vstack([batch['data'] for batch in batches])
```

We then normalize the dataset to the range 0 to 1 and then force the type to be a 32-bit float (this is the only datatype the GPU-enabled virtual machine can run with):

```
X = np.array(X) / X.max()
X = X.astype(np.float32)
```

We then do the same with the classes, except we perform a `hstack`, which is similar to adding columns to the end of the array. We then use the `OneHotEncoder` to turn this into a one-hot array:

```
from sklearn.preprocessing import OneHotEncoder
y = np.hstack(batch['labels'] for batch in batches).flatten()
y = OneHotEncoder().fit_transform(y.reshape(y.shape[0],1)).todense()
y = y.astype(np.float32)
```

Next, we split the dataset into training and testing sets:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_
size=0.2)
```

Next, we reshape the arrays to preserve the original data structure. The original data was 32 by 32 pixel images, with 3 values per pixel (for the red, green, and blue values);

```
X_train = X_train.reshape(-1, 3, 32, 32)
X_test = X_test.reshape(-1, 3, 32, 32)
```

We now have a familiar training and testing dataset, along with the target classes for each. We can now build the classifier.

Creating the neural network

We will be using the `nolearn` package to build the neural network, and therefore will follow a pattern that is similar to our replication experiment in *Chapter 8*, *Beating CAPTCHAs with Neural Networks* replication.

First we create the layers of our neural network:

```
from lasagne import layers
layers=[
        ('input', layers.InputLayer),
        ('conv1', layers.Conv2DLayer),
        ('pool1', layers.MaxPool2DLayer),
        ('conv2', layers.Conv2DLayer),
        ('pool2', layers.MaxPool2DLayer),
        ('conv3', layers.Conv2DLayer),
        ('pool3', layers.MaxPool2DLayer),
        ('hidden4', layers.DenseLayer),
        ('hidden5', layers.DenseLayer),
        ('output', layers.DenseLayer),
        ]
```

We use dense layers for the last three layers, but before that we use convolution layers combined with pooling layers. We have three sets of these. In addition, we start (as we must) with an input layer. This gives us a total of 10 layers. As before, the size of the first and last layers is easy to work out from the dataset, although our input size will have the same shape as the dataset rather than just the same number of nodes/inputs.

Start building our neural network (remember to not close the parentheses):

```
from nolearn.lasagne import NeuralNet
nnet = NeuralNet(layers=layers,
```

Add the input shape. The shape here resembles the shape of the dataset (three values per pixel and a 32 by 32 pixel image). The first value, None, is the default batch size used by `nolearn`—it will train on this number of samples at once, decreasing the running time of the algorithm. Setting it to None removes this hard-coded value, giving us more flexibility in running our algorithm:

```
input_shape=(None, 3, 32, 32),
```

To change the batch size, you will need to create a `BatchIterator` instance. Those who are interested in this parameter can view the source of the file at `https://github.com/dnouri/nolearn/tree/master/nolearn/lasagne`, track the `batch_iterator_train` and `batch_iterator_test` parameters, and see how they are set in the `NeuralNet` class in this file.

Next we set the size of the convolution layers. There are no strict rules here, but I found the following values to be good starting points;

```
conv1_num_filters=32,
conv1_filter_size=(3, 3),
conv2_num_filters=64,
conv2_filter_size=(2, 2),
conv3_num_filters=128,
conv3_filter_size=(2, 2),
```

The `filter_size` parameter dictates the size of the window of the image that the convolution layer looks at. In addition, we set the size of the pooling layers:

```
pool1_ds=(2,2),
pool2_ds=(2,2),
pool3_ds=(2,2),
```

We then set the size of the two hidden dense layers (the third-last and second-last layers) and also the size of the output layer, which is just the number of classes in our dataset;

```
hidden4_num_units=500,
hidden5_num_units=500,
output_num_units=10,
```

We also set a nonlinearity for the final layer, again using `softmax`;

```
output_nonlinearity=softmax,
```

We also set the learning rate and momentum. As a rule of thumb, as the number of samples increase, the learning rate should decrease:

```
update_learning_rate=0.01,
update_momentum=0.9,
```

We set regression to be `True`, as we did before, and set the number of training epochs to be low as this network will take a long time to run. After a successful run, increasing the number of epochs will result in a much better model, but you may need to wait for a day or two (or more!) for it to train:

```
regression=True,
max_epochs=3,
```

Finally, we set the verbosity as equal to 1, which will give us a printout of the results of each epoch. This allows us to know the progress of the model and also that it is still running. Another feature is that it tells us the time it takes for each epoch to run. This is pretty consistent, so you can compute the time left in training by multiplying this value by the number of remaining epochs, giving a good estimate on how long you need to wait for the training to complete:

```
verbose=1)
```

Putting it all together

Now that we have our network, we can train it with our training dataset:

```
nnet.fit(X_train, y_train)
```

This will take quite a while to run, even with the reduced dataset size and the reduced number of epochs. Once the code completes, you can test it as we did before:

```
from sklearn.metrics import f1_score
y_pred = nnet.predict(X_test)
print(f1_score(y_test.argmax(axis=1), y_pred.argmax(axis=1)))
```

The results will be terrible—as they should be! We haven't trained the network very much—only for a few iterations and only on one fifth of the data.

First, go back and remove the break line we put in when creating the dataset (it is in the batches loop). This will allow the code to train on all of the samples, not just some of them.

Next, change the number of epochs to 100 in the neural network definition.

Now, we upload the script to our virtual machine. As with before, click on **File | Download** as, `Python`, and save the script somewhere on your computer. Launch and connect to the virtual machine and upload the script as you did earlier (I called my script `chapter11cifar.py`—if you named yours differently, just update the following code).

The next thing we need is for the dataset to be on the virtual machine. The easiest way to do this is to go to the virtual machine and type:

```
wget http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
```

This will download the dataset. Once that has downloaded, you can extract the data to the `Data` folder by first creating that folder and then unzipping the data there:

```
mkdir Data
tar -zxf cifar-10-python.tar.gz -C Data
```

Finally, we can run our example with the following:

```
python3 chapter11cifar.py
```

The first thing you'll notice is a drastic speedup. On my home computer, each epoch took over 100 seconds to run. On the GPU-enabled virtual machine, each epoch takes just 16 seconds! If we tried running 100 epochs on my computer, it would take nearly three hours, compared to just 26 minutes on the virtual machine.

This drastic speedup makes trailing different models much faster. Often with trialing machine learning algorithms, the computational complexity of a single algorithm doesn't matter too much. An algorithm might take a few seconds, minutes, or hours to run. If you are only running one model, it is unlikely that this training time will matter too much—especially as prediction with most machine learning algorithms is quite quick, and that is where a machine learning model is mostly used.

However, when you have many parameters to run, you will suddenly need to train thousands of models with slightly different parameters—suddenly, these speed increases matter much more.

After 100 epochs of training, taking a whole 26 minutes, you will get a printout of the final result:

```
0.8497
```

Not too bad! We can increase the number of epochs of training to improve this further or we might try changing the parameters instead; perhaps, more hidden nodes, more convolution layers, or an additional dense layer. There are other types of layers in Lasagne that could be tried too; although generally, convolution layers are better for vision.

Your Coding Challenge

Ankita Thakur

Your Course Guide

There are many datasets of images available from a number of academic and industry-based sources. `http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html` lists a bunch of datasets and some of the best algorithms to use on them. Implementing some of the better algorithms will require significant amounts of custom code, but the payoff can be well worth the pain. Try for it!

Summary of Module 3 Chapter 11

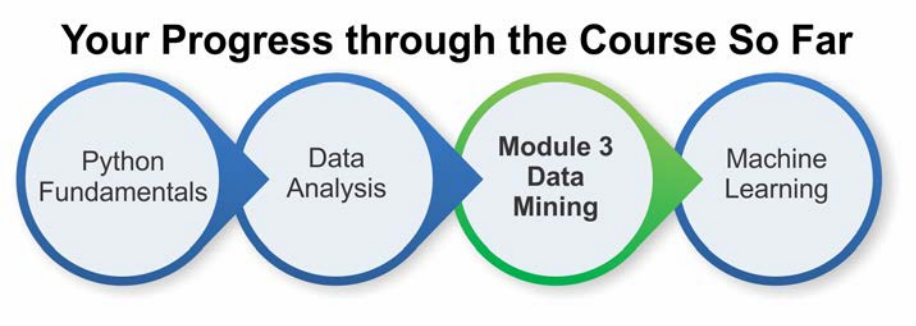
In this chapter, we looked at using deep neural networks, specifically convolution networks, in order to perform computer vision. We did this through the Lasagne and nolearn packages, which work off Theano. The networks were relatively easy to build with nolearn's helper functions.

The convolution networks were designed for computer vision, so it shouldn't be a surprise that the result was quite accurate. The final result shows that computer vision is indeed an effective application using today's algorithms and computational power.
We also used a GPU-enabled virtual machine to drastically speed up the process, by a factor of almost 10 for my machine. If you need extra power to run some of these algorithms, virtual machines by cloud providers can be an effective way to do this (usually for less than a dollar per hour)—just remember to turn them off when you are done!

Ankita Thakur

Your Course Guide

This chapter's focus was on a very complex algorithm. Convolution networks take a long time to train and have many parameters to train. Ultimately, the size of the data was small in comparison; although it was a large dataset, we can load it all in memory without even using sparse matrices. In the next chapter, we go for a much simpler algorithm, but a much, much larger dataset that can't fit in memory. This is the basis of Big Data and it underpins applications of data mining in many large industries such as mining and social networks.


Your Progress through the Course So Far
Python Fundamentals
Data Analysis
Module 3 Data Mining
Machine Learning

12
Working with Big Data

The amount of data is increasing at exponential rates. Today's systems are generating and recording information on customer behavior, distributed systems, network analysis, sensors and many, many more sources. While the current big trend of mobile data is pushing the current growth, the next big thing—the **Internet of Things** (**IoT**)—is going to further increase the rate of growth.

What this means for data mining is a new way of thinking. The complex algorithms with high run times need to be improved or discarded, while simpler algorithms that can deal with more samples are becoming more popular to use. As an example, while support vector machines are great classifiers, some variants are difficult to use on very large datasets. In contrast, simpler algorithms such as logistic regression can manage more easily in these scenarios.

In this chapter, we will investigate the following:

- Big data challenges and applications
- The MapReduce paradigm
- Hadoop MapReduce
- mrjob, a python library to run MapReduce programs on Amazon's infrastructure

Big data

What makes big data different? Most big-data proponents talk about the four Vs of big data:

1. **Volume**: The amount of data that we generate and store is growing at an increasing rate, and predictions of the future generally only suggest further increases. Today's multi-gigabyte sized hard drives will turn into exabyte hard drives in a few years, and network throughput traffic will be increasing as well. The signal to noise ratio can be quite difficult, with important data being lost in the mountain of non-important data.
2. **Velocity**: While related to volume, the velocity of data is increasing too. Modern cars have hundreds of sensors that stream data into their computers, and the information from these sensors needs to be analyzed at a subsecond level to operate the car. It isn't just a case of finding answers in the volume of data; those answers often need to come quickly.
3. **Variety**: Nice datasets with clearly defined columns are only a small part of the dataset that we have these days. Consider a social media post, which may have text, photos, user mentions, likes, comments, videos, geographic information, and other fields. Simply ignoring parts of this data that don't fit your model will lead to a loss of information, but integrating that information itself can be very difficult.
4. **Veracity**: With the increase in the amount of data, it can be hard to determine whether the data is being correctly collected—whether it is outdated, noisy, contains outliers, or generally whether it is useful at all. Being able to trust the data is hard when a human can't reliably verify the data itself. External datasets are being increasingly merged into internal ones too, giving rise to more troubles relating to the veracity of the data.

These main four Vs (others have proposed additional Vs) outline why big data is different to just lots-of-data. At these scales, the engineering problem of working with the data is often more difficult—let alone the analysis. While there are lots of snake oil salesmen that overstate the ability to use big data, it is hard to deny the engineering challenges and the potential of big-data analytics.

The algorithms we have used are to date load the dataset into memory and then to work on the in-memory version. This gives a large benefit in terms of speed of computation, as it is much faster to compute on in-memory data than having to load a sample before we use it. In addition, in-memory data allows us to iterate over the data many times, improving our model.

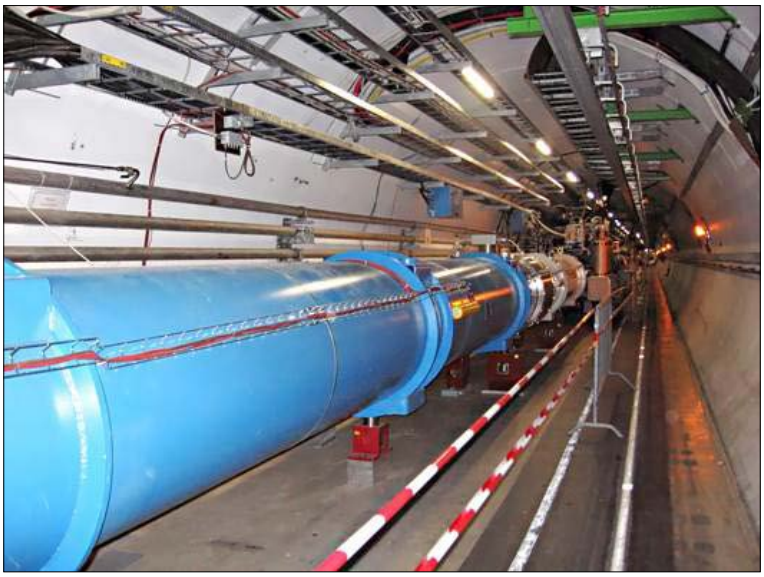
In big data, we can't load our data into memory, In many ways, this is a good definition for whether a problem is big data or not—if the data can fit in the memory on your computer, you aren't dealing with a big data problem.

Application scenario and goals

There are many use cases for big data, in the public and private sectors.

The most common experience people have using a big-data-based system is in Internet search, such as Google. To run these systems, a search needs to be carried out over billions of websites in a fraction of a second. Doing a basic text-based search would be inadequate to deal with such a problem. Simply storing the text of all those websites is a large problem. In order to deal with queries, new data structures and data mining methods need to be created and implemented specifically for this application.

Big data is also used in many other scientific experiments such as the Large Hadron Collider, part of which is pictured below, that stretches over 17 kilometers and contains 150 million sensors monitoring hundreds of millions of particle collisions per second. The data from this experiment is massive, with 25 petabytes created daily, after a filtering process (if filtering was not used, there would be 150 million petabytes per year). Analysis on data this big has led to amazing insights about our universe, but has been a significant engineering and analytics challenge.



Governments are increasingly using big data too, to track populations, businesses, and other aspects about their country. Tracking millions of people and billions of interactions (such as business transactions or health spending) has led to a need for big data analytics in many government organizations.

Traffic management is a particular focus of many governments around the world, who are tracking traffic using millions of sensors to determine which roads are most congested and predicting the impact of new roads on traffic levels.

Large retail organizations are using big data to improve the customer experience and reduce costs. This involves predicting customer demand in order to have the correct level of inventory, upselling customers with products they may like to purchase, and tracking transactions to look for trends, patterns, and potential frauds.

Other large businesses are also leveraging big data to automate aspects of their business and improve their offering. This includes leveraging analytics to predict future trends in their sector and track external competitors. Large businesses also use analytics to manage their own employees—tracking employees to look for signs that an employee may leave the company, in order to intervene before they do.

The information security sector is also leveraging big data in order to look for malware infections in large networks, by monitoring network traffic. This can include looking for odd traffic patterns, evidence of malware spreading, and other oddities. **Advanced Persistent Threats** (**APTs**) is another problem, where a motivated attacker will hide their code within a large network to steal information or cause damage over a long period of time. Finding APTs is often a case of forensically examining many computers, a task which simply takes too long for a human to effectively perform themselves. Analytics helps automate and analyze these forensic images to find infections.

Big data is being used in an increasing number of sectors and applications, and this trend is likely to only continue.

MapReduce

There are a number of concepts to perform data mining and general computation on big data. One of the most popular is the MapReduce model, which can be used for general computation on arbitrarily large datasets.

MapReduce originates from Google, where it was developed with distributed computing in mind. It also introduces fault tolerance and scalability improvements. The "original" research for MapReduce was published in 2004, and since then there have been thousands of projects, implementations, and applications using it.

While the concept is similar to many previous concepts, MapReduce has become a staple in big data analytics.

Intuition

MapReduce has two main steps: the Map step and the Reduce step. These are built on the functional programming concepts of mapping a function to a list and reducing the result. To explain the concept, we will develop code that will iterate over a list of lists and produce the sum of all numbers in those lists.

There are also shuffle and combine steps in the MapReduce paradigm, which we will see later.

To start with, the Map step takes a function and applies it to each element in a list. The returned result is a list of the same size, with the results of the function applied to each element.

To open a new IPython Notebook, start by creating a list of lists with numbers in each sublist:

```
a = [[1,2,1], [3,2], [4,9,1,0,2]]
```

Next, we can perform a map, using the sum function. This step will apply the sum function to each element of `a`:

```
sums = map(sum, a)
```

While sums is a generator (the actual value isn't computed until we ask for it), the above step is approximately equal to the following code:

```
sums = []
for sublist in a:
    results = sum(sublist)
    sums.append(results)
```

The reduce step is a little more complicated. It involves applying a function to each element of the returned result, to some starting value. We start with an initial value, and then apply a given function to that initial value and the first value. We then apply the given function to the result and the next value, and so on.

We start by creating a function that takes two numbers and adds them together.

```
def add(a, b):
  return a + b
```

We then perform the reduce. The signature of reduce is reduce(function, sequence, and initial), where the function is applied at each step to the sequence. In the first step, the initial value is used as the first value, rather than the first element of the list:

```
from functools import reduce
print(reduce(add, sums, 0))
```

The result, *25*, is the sum of each of the values in the sums list and is consequently the sum of each of the elements in the original array.

The preceding code is equal to the following:

```
initial = 0
current_result = initial
for element in sums:
    current_result = add(current_result, element)
```

In this trivial example, our code can be greatly simplified, but the real gains come from distributing the computation. For instance, if we have a million sublists and each of those sublists contained a million elements, we can distribute this computation over many computers.

In order to do this, we distribute the map step. For each of the elements in our list, we send it, along with a description of our function, to a computer. This computer then returns the result to our main computer (the master).

The master then sends the result to a computer for the reduce step. In our example of a million sublists, we would send a million jobs to different computers (the same computer may be reused after it completes our first job). The returned result would be just a single list of a million numbers, which we then compute the sum of.

The result is that no computer ever needed to store more than a million numbers, despite our original data having a trillion numbers in it.

A word count example

The implementation of MapReduce is a little more complex than just using a map and reduce step. Both steps are invoked using keys, which allows for the separation of data and tracking of values.

The map function takes a key and value pair and returns a list of *key+value* pairs. The keys for the input and output don't necessarily relate to each other. For example, for a MapReduce program that performs a word count, the input key might be a sample document's ID value, while the output key would be a given word. The input value would be the text of the document and the output value would be the frequency of each word:

```
from collections import defaultdict
def map_word_count(document_id, document):
```

We first count the frequency of each word. In this simplified example, we split the document on whitespace to obtain the words, although there are better options:

```
    counts = defaultdict(int)
    for word in document.split():
        counts[word] += 1
```

We then yield each of the word, count pairs. The word here is the key, with the count being the value in MapReduce terms:

```
    for word in counts:
        yield (word, counts[word])
```

By using the word as the key, we can then perform a **shuffle** step, which groups all of the values for each key:

```
def shuffle_words(results):
```

First, we aggregate the resulting counts for each word into a list of counts:

```
    records = defaultdict(list)
```

We then iterate over all the results that were returned by the map function;

```
    for results in results_generators:
        for word, count in results:
            records[word].append(count)
```

Next, we yield each of the words along with all the counts that were obtained in our dataset:

```
        for word in records:
            yield (word, records[word])
```

The final step is the reduce step, which takes a key value pair (the value in this case is always a list) and produces a key value pair as a result. In our example, the key is the word, the input list is the list of counts produced in the shuffle step, and the output value is the sum of the counts:

```
def reduce_counts(word, list_of_counts):
    return (word, sum(list_of_counts))
```

To see this in action, we can use the 20 newsgroups dataset, which is provided in scikit-learn:

```
from sklearn.datasets import fetch_20newsgroups
dataset = fetch_20newsgroups(subset='train')
documents = dataset.data
```

We then apply our map step. We use enumerate here to automatically generate document IDs for us. While they aren't important in this application, these keys are important in other applications;

```
map_results = map(map_word_count, enumerate(documents))
```

The actual result here is just a generator, no actual counts have been produced. That said, it is a generator that emits (word, count) pairs.

Next, we perform the shuffle step to sort these word counts:

```
shuffle_results = shuffle_words(map_results)
```

This, in essence is a MapReduce job; however, it is only running on a single thread, meaning we aren't getting any benefit from the MapReduce data format. In the next section, we will start using Hadoop, an open source provider of MapReduce, to start to get the benefits from this type of paradigm.

Hadoop MapReduce

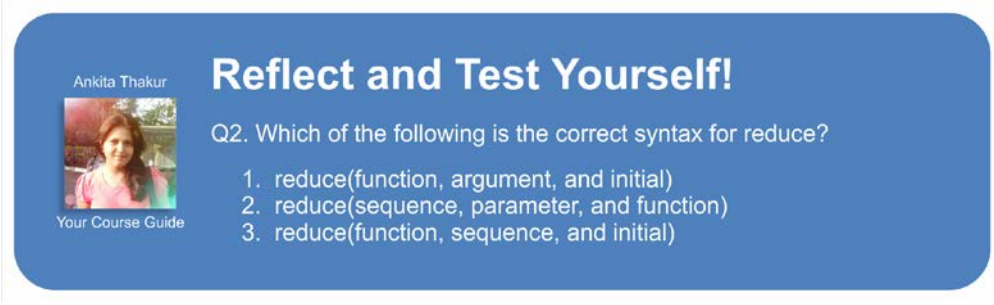
Hadoop is a set of open source tools from Apache that includes an implementation of MapReduce. In many cases, it is the de facto implementation used by many. The project is managed by the Apache group (who are responsible for the famous web server).

The Hadoop ecosystem is quite complex, with a large number of tools. The main component we will use is Hadoop MapReduce. Other tools for working with big data that are included in Hadoop are as follows:

- **Hadoop Distributed File System (HDFS)**: This is a file system that can store files over many computers, with the goal of being robust against hardware failure while providing high bandwidth.
- **YARN**: This is a method for scheduling applications and managing clusters of computers.
- **Pig**: This is a higher level programming language for MapReduce. Hadoop MapReduce is implemented in Java, and Pig sits on top of the Java implementation, allowing you to write programs in other languages—including Python.
- **Hive**: This is for managing data warehouses and performing queries.
- **HBase**: This is an implementation of Google's BigTable, a distributed database.

These tools all solve different issues that come up when doing big data experiments, including data analytics.

There are also non-Hadoop-based implementations of MapReduce, as well as other projects with similar goals. In addition, many cloud providers have MapReduce-based systems.



Application

In this application, we will look at predicting the gender of a writer based on their use of different words. We will use a Naive Bayes method for this, trained in MapReduce. The final model doesn't need MapReduce, although we can use the Map step to do so—that is, run the prediction model on each document in a list. This is a common Map operation for data mining in MapReduce, with the reduce step simply organizing the list of predictions so they can be tracked back to the original document.

We will be using Amazon's infrastructure to run our application, allowing us to leverage their computing resources.

Getting the data

The data we are going to use is a set of blog posts that are labeled for age, gender, industry (that is, work) and, funnily enough, star sign. This data was collected from `http://blogger.com` in August 2004 and has over 140 million words in more than 600,000 posts. Each blog is probably written by just one person, with some work put into verifying this (although, we can never be really sure). Posts are also matched with the date of posting, making this a very rich dataset.

To get the data, go to `http://u.cs.biu.ac.il/~koppel/BlogCorpus.htm` and click on **Download Corpus**. From there, unzip the file to a directory on your computer.

The dataset is organized with a single blog to a file, with the filename giving the classes. For instance, one of the filenames is as follows:

```
1005545.male.25.Engineering.Sagittarius.xml
```

The filename is separated by periods, and the fields are as follows:

- **Blogger ID**: This a simple ID value to organize the identities.
- **Gender**: This is either male or female, and all the blogs are identified as one of these two options (no other options are included in this dataset).
- **Age**: The exact ages are given, but some gaps are deliberately present. Ages present are in the (inclusive) ranges of 13-17, 23-27, and 33-48. The reason for the gaps is to allow for splitting the blogs into age ranges with gaps, as it would be quite difficult to separate an 18 year old's writing from a 19 year old, and it is possible that the age itself is a little outdated.
- **Industry**: In one of 40 different industries including science, engineering, arts, and real estate. Also, included is indUnk, for unknown industry.
- **Star Sign**: This is one of the 12 astrological star signs.

All values are self-reported, meaning there may be errors or inconsistencies with labeling, but are assumed to be mostly reliable—people had the option of not setting values if they wanted to preserve their privacy in those ways.

A single file is in a `pseudo-XML` format, containing a `<Blog>` tag and then a sequence of `<post>` tags. Each of the `<post>` tag is proceeded by a `<date>` tag as well. While we can parse this as XML, it is much simpler to parse it on a line-by-line basis as the files are not exactly well-formed XML, with some errors (mostly encoding problems). To read the posts in the file, we can use a loop to iterate over the lines.

We set a test filename so we can see this in action:

```
import os
filename = os.path.join(os.path.expanduser("~"), "Data", "blogs",
"1005545.male.25.Engineering.Sagittarius.xml")
```

First, we create a list that will let us store each of the posts:

```
all_posts = []
```

Then, we open the file to read:

```
with open(filename) as inf:
```

We then set a flag indicating whether we are currently in a post. We will set this to `True` when we find a `<post>` tag indicating the start of a post and set it to `False` when we find the closing `</post>` tag;

```
    post_start = False
```

We then create a list that stores the current post's lines:

```
    post = []
```

We then iterate over each line of the file and remove white space:

```
    for line in inf:
        line = line.strip()
```

As stated before, if we find the opening `<post>` tag, we indicate that we are in a new post. Likewise, with the close `</post>` tag:

```
        if line == "<post>":
            post_start = True
        elif line == "</post>":
            post_start = False
```

When we do find the closing `</post>` tag, we also then record the full post that we have found so far. We also then start a new "current" post. This code is on the same indentation level as the previous line:

```
            all_posts.append("\n".join(post))
            post = []
```

Finally, when the line isn't a start of end tag, but we are in a post, we add the text of the current line to our current post:

```
        elif post_start:
            post.append(line)
```

If we aren't in a current post, we simply ignore the line.

We can then grab the text of each post:

```
print(all_posts[0])
```

We can also find out how many posts this author created:

```
print(len(all_posts))
```

Naive Bayes prediction

We are now going to implement the Naive Bayes algorithm (technically, a reduced version of it, without many of the features that more complex implementations have) that is able to process our dataset.

The mrjob package

The mrjob package allows us to create MapReduce jobs that can easily be transported to Amazon's infrastructure. While mrjob sounds like a sedulous addition to the Mr. Men series of children's books, it actually stands for *Map Reduce Job*. It is a great package; however, as of the time of writing, Python 3 support is still not mature yet, which is true for the Amazon EMR service that we will discuss later on.

You can install mrjob for Python 2 versions using the following:

```
sudo pip2 install mrjob
```

Note that `pip` is used for version 2, not for version 3.

In essence, mrjob provides the standard functionality that most MapReduce jobs need. Its most amazing feature is that you can write the same code, test on your local machine without Hadoop, and then push to Amazon's EMR service or another Hadoop server.

This makes testing the code significantly easier, although it can't magically make a big problem small—note that any local testing uses a subset of the dataset, rather than the whole, big dataset.

Extracting the blog posts

We are first going to create a MapReduce program that will extract each of the posts from each blog file and store them as separate entries. As we are interested in the gender of the author of the posts, we will extract that too and store it with the post.

We can't do this in an IPython Notebook, so instead open a Python IDE for development. If you don't have a Python IDE (such as PyCharm), you can use a text editor. I recommend looking for an IDE that has syntax highlighting.

If you still can't find a good IDE, you can write the code in an IPython Notebook and then click on **File** | **Download As** | **Python**. Save this file to a directory and run it as we outlined in *Chapter 11*, *Classifying Objects in Images using Deep Learning*.

To do this, we will need the `os` and `re` libraries as we will be obtaining environment variables and we will also use a regular expression for word separation:

```
import os
import re
```

We then import the `MRJob` class, which we will inherit from our MapReduce job:

```
from mrjob.job import MRJob
```

We then create a new class that subclasses MRJob:

```
class ExtractPosts(MRJob):
```

We will use a similar loop, as before, to extract blog posts from the file. The mapping function we will define next will work off each line, meaning we have to track different posts outside of the mapping function. For this reason, we make `post_start` and `post` class variables, rather than variables inside the function:

```
    post_start = False
    post = []
```

We then define our mapper function—this takes a line from a file as input and yields blog posts. The lines are guaranteed to be ordered from the same per-job file. This allows us to use the above class variables to record current post data:

```
    def mapper(self, key, line):
```

Before we start collecting blog posts, we need to get the gender of the author of the blog. While we don't normally use the filename as part of MapReduce jobs, there is a strong need for it (as in this case) so the functionality is available. The current file is stored as an environment variable, which we can obtain using the following line of code:

```
        filename = os.environ["map_input_file"]
```

We then split the filename to get the gender (which is the second token):

```
gender = filename.split(".")[1]
```

We remove whitespace from the start and end of the line (there is a lot of whitespace in these documents) and then do our post-based tracking as before;

```
line = line.strip()
if line == "<post>":
    self.post_start = True
elif line == "</post>":
    self.post_start = False
```

Rather than storing the posts in a list, as we did earlier, we yield them. This allows mrjob to track the output. We yield both the gender and the post so that we can keep a record of which gender each record matches. The rest of this function is defined in the same way as our loop above:

```
    yield gender, repr("\n".join(self.post))
    self.post = []
elif self.post_start:
    self.post.append(line)
```

Finally, outside the function and class, we set the script to run this MapReduce job when it is called from the command line:

```
if __name__ == '__main__':
    ExtractPosts.run()
```

Now, we can run this MapReduce job using the following shell command. Note that we are using Python 2, and not Python 3 to run this;

```
python extract_posts.py <your_data_folder>/blogs/51* --output-
  dir=<your_data_folder>/blogposts -no-output
```

The first parameter, `<your_data_folder>/blogs/51*` (just remember to change `<your_data_folder>` to the full path to your data folder), obtains a sample of the data (all files starting with 51, which is only 11 documents). We then set the output directory to a new folder, which we put in the data folder, and specify not to output the streamed data. Without the last option, the output data is shown to the command line when we run it—which isn't very helpful to us and slows down the computer quite a lot.

Run the script, and quite quickly each of the blog posts will be extracted and stored in our `output` folder. This script only ran on a single thread on the local computer so we didn't get a speedup at all, but we know the code runs.

We can now look in the `output` folder for the results. A bunch of files are created and each file contains each blog post on a separate line, preceded by the gender of the author of the blog.

Training Naive Bayes

Now that we have extracted the blog posts, we can train our Naive Bayes model on them. The intuition is that we record the probability of a word being written by a particular gender. To classify a new sample, we would multiply the probabilities and find the most likely gender.

The aim of this code is to output a file that lists each word in the corpus, along with the frequencies of that word for each gender. The output file will look something like this:

```
"'ailleurs"  {"female": 0.003205128205128205}
"'air"  {"female": 0.003205128205128205}
"'an"  {"male": 0.0030581039755351682, "female": 0.004273504273504274}
"'angoisse"  {"female": 0.003205128205128205}
"'apprendra"  {"male": 0.0013047113868622459, "female":
0.0014172668603481887}
"'attendent"  {"female": 0.00641025641025641}
"'autistic"  {"male": 0.002150537634408602}
"'auto"  {"female": 0.003205128205128205}
"'avais"  {"female": 0.00641025641025641}
"'avait"  {"female": 0.004273504273504274}
"'behind"  {"male": 0.0024390243902439024}
"'bout"  {"female": 0.002034152292059272}
```

The first value is the word and the second is a dictionary mapping the genders to the frequency of that word in that gender's writings.

Open a new file in your Python IDE or text editor. We will again need the `os` and `re` libraries, as well as `NumPy` and `MRJob` from mrjob. We also need itemgetter, as we will be sorting a dictionary:

```
import os
import re
import numpy as np
from mrjob.job import MRJob
from operator import itemgetter
```

We will also need MRStep, which outlines a step in a MapReduce job. Our previous job only had a single step, which is defined as a mapping function and then as a reducing function. This job will have three steps where we Map, Reduce, and then Map and Reduce again. The intuition is the same as the pipelines we used in earlier chapters, where the output of one step is the input to the next step:

```
from mrjob.step import MRStep
```

We then create our word search regular expression and compile it, allowing us to find word boundaries. This type of regular expression is much more powerful than the simple split we used in some previous chapters, but if you are looking for a more accurate word splitter, I recommend using NLTK as we did in *Chapter 6*, *Social Media Insight using Naive Bayes*:

```
word_search_re = re.compile(r"[\w']+")
```

We define a new class for our training:

```
class NaiveBayesTrainer(MRJob):
```

We define the steps of our `MapReduce` job. There are two steps. The first step will extract the word occurrence probabilities. The second step will compare the two genders and output the probabilities for each to our output file. In each MRStep, we define the mapper and reducer functions, which are class functions in this `NaiveBayesTrainer` class (we will write those functions next):

```
    def steps(self):
        return [
            MRStep(mapper=self.extract_words_mapping,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.compare_words_reducer),
            ]
```

The first function is the mapper function for the first step. The goal of this function is to take each blog post, get all the words in that post, and then note the occurrence. We want the frequencies of the words, so we will return `1 / len(all_words)`, which allows us to later sum the values for frequencies. The computation here isn't exactly correct—we need to also normalize for the number of documents. In this dataset, however, the class sizes are the same, so we can conveniently ignore this with little impact on our final version.

We also output the gender of the post's author, as we will need that later:

```
def extract_words_mapping(self, key, value):
    tokens = value.split()
    gender = eval(tokens[0])
    blog_post = eval(" ".join(tokens[1:]))
    all_words = word_search_re.findall(blog_post)
    all_words = [word.lower() for word in all_words]
        all_words = word_search_re.findall(blog_post)
        all_words = [word.lower() for word in all_words]
        for word in all_words:
            yield (gender, word), 1. / len(all_words)
```

We used `eval` in the preceding code to simplify the parsing of the blog posts from the file, for this example. This is not recommended. Instead, use a format such as JSON to properly store and parse the data from the files. A malicious use with access to the dataset can insert code into these tokens and have that code run on your server.

In the reducer for the first step, we sum the frequencies for each gender and word pair. We also change the key to be the word, rather than the combination, as this allows us to search by word when we use the final trained model (although, we still need to output the gender for later use);

```
def reducer_count_words(self, key, frequencies):
    s = sum(frequencies)
    gender, word = key
    yield word, (gender, s)
```

The final step doesn't need a mapper function, so we don't add one. The data will pass straight through as a type of `identity mapper`. The reducer, however, will combine frequencies for each gender under the given word and then output the word and frequency dictionary.

This gives us the information we needed for our Naive Bayes implementation:

```
    def compare_words_reducer(self, word, values):
        per_gender = {}
        for value in values:
            gender, s = value
            per_gender[gender] = s
        yield word, per_gender
```

Finally, we set the code to run this model when the file is run as a script;

```
if __name__ == '__main__':
    NaiveBayesTrainer.run()
```

We can then run this script. The input to this script is the output of the previous post-extractor script (we can actually have them as different steps in the same MapReduce job if you are so inclined);

```
python nb_train.py <your_data_folder>/blogposts/
  --output-dir=<your_data_folder>/models/
--no-output
```

The output directory is a folder that will store a file containing the output from this MapReduce job, which will be the probabilities we need to run our Naive Bayes classifier.

Putting it all together

We can now actually run the Naive Bayes classifier using these probabilities. We will do this in an IPython Notebook, and can go back to using Python 3 (phew!).

First, take a look at the models folder that was specified in the last MapReduce job. If the output was more than one file, we can merge the files by just appending them to each other using a command line function from within the models directory:

```
cat * > model.txt
```

If you do this, you'll need to update the following code with `model.txt` as the model filename.

Back to our Notebook, we first import some standard imports we need for our script:

```
import os
import re
import numpy as np
from collections import defaultdict
from operator import itemgetter
```

We again redefine our word search regular expression—if you were doing this in a real application, I recommend centralizing this. It is important that words are extracted in the same way for training and testing:

```
word_search_re = re.compile(r"[\w']+")
```

Next, we create the function that loads our model from a given filename:

```
def load_model(model_filename):
```

The model parameters will take the form of a dictionary of dictionaries, where the first key is a word, and the inner dictionary maps each gender to a probability. We use `defaultdicts`, which will return zero if a value isn't present;

```
    model = defaultdict(lambda: defaultdict(float))
```

We then open the model and parse each line;

```
    with open(model_filename) as inf:
        for line in inf:
```

The line is split into two sections, separated by whitespace. The first is the word itself and the second is a dictionary of probabilities. For each, we run `eval` on them to get the actual value, which was stored using `repr` in the previous code:

```
            word, values = line.split(maxsplit=1)
            word = eval(word)
            values = eval(values)
```

We then track the values to the word in our model:

```
            model[word] = values
    return model
```

Next, we load our actual model. You may need to change the model filename—it will be in the output `dir` of the last MapReduce job;

```
model_filename = os.path.join(os.path.expanduser("~"), "models",
"part-00000")
model = load_model(model_filename)
```

As an example, we can see the difference in usage of the word **i** (all words are turned into lowercase in the MapReduce jobs) between males and females:

```
model["i"]["male"], model["i"]["female"]
```

Next, we create a function that can use this model for prediction. We won't use the scikit-learn interface for this example, and just create a function instead. Our function takes the model and a document as the parameters and returns the most likely gender:

```
def nb_predict(model, document):
```

We start by creating a dictionary to map each gender to the computed probability:

```
    probabilities = defaultdict(lambda : 1)
```

We extract each of the words from the document:

```
    words = word_search_re.findall(document)
```

We then iterate over the words and find the probability for each gender in the dataset:

```
    for word in set(words):
        probabilities["male"] += np.log(model[word].get("male", 1e-
15))
        probabilities["female"] += np.log(model[word].get("female",
1e-15))
```

We then sort the genders by their value, get the highest value, and return that as our prediction:

```
    most_likely_genders = sorted(probabilities.items(),
key=itemgetter(1), reverse=True)
    return most_likely_genders[0][0]
```

It is important to note that we used `np.log` to compute the probabilities. Probabilities in Naive Bayes models are often quite small. Multiplying small values, which is necessary in many statistical values, can lead to an underflow error where the computer's precision isn't good enough and just makes the whole value 0. In this case, it would cause the likelihoods for both genders to be zero, leading to incorrect predictions.

To get around this, we use log probabilities. For two values a and b, `log(a,b)` is equal to `log(a) + log(b)`. The log of a small probability is a negative value, but a relatively large one. For instance, `log(0.00001)` is about `-11.5`. This means that rather than multiplying actual probabilities and risking an underflow error, we can sum the log probabilities and compare the values in the same way (higher numbers still indicate a higher likelihood).

One problem with using log probabilities is that they don't handle zero values well (although, neither does multiplying by zero probabilities). This is due to the fact that log(0) is undefined. In some implementations of Naive Bayes, a 1 is added to all counts to get rid of this, but there are other ways to address this. This is a simple form of smoothing of the values. In our code, we just return a very small value if the word hasn't been seen for our given gender.

Back to our prediction function, we can test this by copying a post from our dataset:

```
new_post = """ Every day should be a half day.  Took the afternoon
off to hit the dentist, and while I was out I managed to get my oil
changed, too.  Remember that business with my car dealership this
winter?  Well, consider this the epilogue.  The friendly fellas at the
Valvoline Instant Oil Change on Snelling were nice enough to notice
that my dipstick was broken, and the metal piece was too far down in
its little dipstick tube to pull out.  Looks like I'm going to need a
magnet.   Damn you, Kline Nissan, daaaaaaammmnnn yooouuuu....   Today
I let my boss know that I've submitted my Corps application.  The news
has been greeted by everyone in the company with a level of enthusiasm
that really floors me.     The back deck has finally been cleared off
by the construction company working on the place.  This company, for
anyone who's interested, consists mainly of one guy who spends his
days cursing at his crew of Spanish-speaking laborers.  Construction
of my deck began around the time Nixon was getting out of office.
"""
```

We then predict with the following code:

```
nb_predict(model, new_post)
```

The resulting prediction, male, is correct for this example. Of course, we never test a model on a single sample. We used the file starting with 51 for training this model. It wasn't many samples, so we can't expect too high of an accuracy.

The first thing we should do is train on more samples. We will test on any file that starts with a 6 or 7 and train on the rest of the files.

In the command line and in your data folder (`cd <your_data_folder`), where the blogs folder exists, create a copy of the blogs data into a new folder.

Make a folder for our training set:

```
mkdir blogs_train
```

Move any file starting with a 6 or 7 into the test set, from the train set:

```
cp blogs/4* blogs_train/
cp blogs/8* blogs_train/
```

Then, make a folder for our test set:

```
mkdir blogs_test
```

Move any file starting with a `6` or `7` into the test set, from the train set:

```
cp blogs/6* blogs_test/
cp blogs/7* blogs_test/
```

We will rerun the blog extraction on all files in the training set. However, this is a large computation that is better suited to cloud infrastructure than our system. For this reason, we will now move the parsing job to Amazon's infrastructure.

Run the following on the command line, as you did before. The only difference is that we train on a different folder of input files. Before you run the following code, delete all files in the blog posts and models folders:

```
python extract_posts.py ~/Data/blogs_train --output-dir=/home/bob/
Data/blogposts –no-output
python nb_train.py ~/Data/blogposts/ --output-dir=/home/bob/models/
--no-output
```

The code here will take quite a bit longer to run.

We will test on any blog file in our test set. To get the files, we need to extract them. We will use the `extract_posts.py` MapReduce job, but store the files in a separate folder:

```
python extract_posts.py ~/Data/blogs_test --output-dir=/home/bob/Data/
blogposts_testing –no-output
```

Back in the IPython Notebook, we list all the outputted testing files:

```
testing_folder = os.path.join(os.path.expanduser("~"), "Data",
"blogposts_testing")
testing_filenames = []
for filename in os.listdir(testing_folder):
    testing_filenames.append(os.path.join(testing_folder, filename))
```

For each of these files, we extract the gender and document and then call the predict function. We do this in a generator, as there are a lot of documents, and we don't want to use too much memory. The generator yields the actual gender and the predicted gender:

```
def nb_predict_many(model, input_filename):
    with open(input_filename) as inf:
        # remove leading and trailing whitespace
```

```
        for line in inf:
            tokens = line.split()
            actual_gender = eval(tokens[0])
            blog_post = eval(" ".join(tokens[1:]))
            yield actual_gender, nb_predict(model, blog_post)
```

We then record the predictions and actual genders across our entire dataset. Our predictions here are either male or female. In order to use the `f1_score` function from scikit-learn, we need to turn these into ones and zeroes. In order to do that, we record a 0 if the gender is male and 1 if it is female. To do this, we use a Boolean test, seeing if the gender is female. We then convert these Boolean values to `int` using `NumPy`:

```
y_true = []
y_pred = []
for actual_gender, predicted_gender in nb_predict_many(model, testing_
filenames[0]):
    y_true.append(actual_gender == "female")
    y_pred.append(predicted_gender == "female")
y_true = np.array(y_true, dtype='int')
y_pred = np.array(y_pred, dtype='int')
```

Now, we test the quality of this result using the `F1` score in scikit-learn:

```
from sklearn.metrics import f1_score
print("f1={:.4f}".format(f1_score(y_true, y_pred, pos_label=None)))
```

The result of 0.78 is not bad. We can probably improve this by using more data, but to do that, we need to move to a more powerful infrastructure that can handle it.

Training on Amazon's EMR infrastructure

We are going to use Amazon's **Elastic Map Reduce** (**EMR**) infrastructure to run our parsing and model building jobs.

In order to do that, we first need to create a bucket in Amazon's storage cloud. To do this, open the Amazon S3 console in your web browser by going to `http://console.aws.amazon.com/s3` and click on **Create Bucket**. Remember the name of the bucket, as we will need it later.

Right-click on the new bucket and select `Properties`. Then, change the permissions, granting everyone full access. This is not a good security practice in general, and I recommend that you change the access permissions after you complete this chapter.

Left-click the bucket to open it and click on **Create Folder**. Name the folder `blogs_train`. We are going to upload our training data to this folder for processing on the cloud.

On your computer, we are going to use Amazon's AWS CLI, a command-line interface for processing on Amazon's cloud.

To install it, use the following:

```
sudo pip2 install awscli
```

Follow the instructions at `http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-set-up.html` to set the credentials for this program.

We now want to upload our data to our new bucket. First, we want to create our dataset, which is all the blogs not starting with a 6 or 7. There are more graceful ways to do this copy, but none are cross-platform enough to recommend. Instead, simply copy all the files and then delete the ones that start with a 6 or 7, from the training dataset:

```
cp -R ~/Data/blogs ~/Data/blogs_train_large
rm ~/Data/blogs_train_large/6*
rm ~/Data/blogs_train_large/7*
```

Next, upload the data to your Amazon S3 bucket. Note that this will take some time and use quite a lot of upload data (several hundred megabytes). For those with slower Internet connections, it may be worth doing this at a location with a faster connection;

```
 aws s3 cp  ~/Data/blogs_train_large/ s3://ch12/blogs_train_large
--recursive --exclude "*" --include "*.xml"
```

We are going to connect to Amazon's EMR using mrjob—it handles the whole thing for us; it only needs our credentials to do so. Follow the instructions at `https://pythonhosted.org/mrjob/guides/emr-quickstart.html` to setup mrjob with your Amazon credentials.

After this is done, we alter our mrjob run, only slightly, to run on Amazon EMR. We just tell mrjob to use `emr` using the `-r` switch and then set our `s3` containers as the input and output directories. Even though this will be run on Amazon's infrastructure, it will still take quite a long time to run.

```
python extract_posts.py -r emr s3://ch12gender/blogs_train_large/
--output-dir=s3://ch12/blogposts_train/ --no-output
python nb_train.py -r emr s3://ch12/blogposts_train/ --output-dir=s3://
ch12/model/ --o-output
```

You will also be charged for the usage. This will only be a few dollars, but keep this in mind if you are going to keep running the jobs or doing other jobs on bigger datasets. I ran a very large number of jobs and was charged about $20 all up. Running just these few should be less than $4. However, you can check your balance and set up pricing alerts, by going to `https://console.aws.amazon.com/billing/home`.

It isn't necessary for the `blogposts_train` and `model` folders to exist—they will be created by EMR. In fact, if they exist, you will get an error. If you are rerunning this, just change the names of these folders to something new, but remember to change both commands to the same names (that is, the output directory of the first command is the input directory of the second command).

If you are getting impatient, you can always stop the first job after a while and just use the training data gathered so far. I recommend leaving the job for an absolute minimum of 15 minutes and probably at least an hour. You can't stop the second job and get good results though; the second job will probably take about two to three times as long as the first job did.

You can now go back to the `s3` console and download the output model from your bucket. Saving it locally, we can go back to our IPython Notebook and use the new model. We reenter the code here—only the differences are highlighted, just to update to our new model:

```
aws_model_filename = os.path.join(os.path.expanduser("~"), "models",
"aws_model")
aws_model = load_model(aws_model_filename)
y_true = []
y_pred = []
for actual_gender, predicted_gender in nb_predict_many(aws_model,
testing_filenames[0]):
    y_true.append(actual_gender == "female")
    y_pred.append(predicted_gender == "female")
y_true = np.array(y_true, dtype='int')
y_pred = np.array(y_pred, dtype='int')
print("f1={:.4f}".format(f1_score(y_true, y_pred, pos_label=None)))
```

The result is much better with the extra data, at 0.81.

If everything went as planned, you may want to remove the bucket from Amazon S3—you will be charged for the storage.

Ankita Thakur

Your Course Guide

Your Coding Challenge

The dataset used in this Lesson provides authorship-based classes (each blogger ID is a separate author). This dataset can be tested using this kind of method as well. In addition, there are the other classes of gender, age, industry, and star sign that can be tested—are authorship-based methods good for these classification tasks?

Summary of Module 3 Chapter 12

In this chapter, we looked at running jobs on big data. By most standards, our dataset is quite small—only a few hundred megabytes. Many industrial datasets are much bigger, so extra processing power is needed to perform the computation.

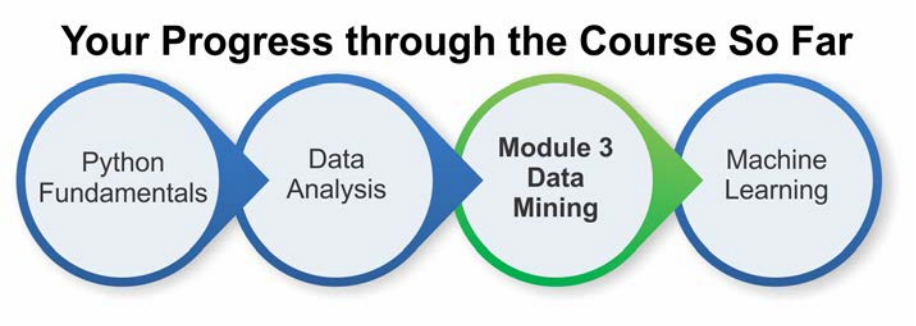
Ankita Thakur

Your Course Guide

In addition, the algorithms we used can be optimized for different tasks to further increase the scalability.

Our approach extracted word frequencies from blog posts, in order to predict the gender of the author of a document. We extracted the blogs and word frequencies using MapReduce-based projects in mrjob. With those extracted, we can then perform a Naive Bayes-esque computation to predict the gender of a new document.

We can use the mrjob library to test locally and then automatically set up and use Amazon's EMR cloud infrastructure. You can use other cloud infrastructure or even a custom built Amazon EMR cluster to run these MapReduce jobs, but there is a bit more tinkering needed to get them running.


Your Progress through the Course So Far
Python Fundamentals
Data Analysis
Module 3 Data Mining
Machine Learning

13
Next Steps…

During the course of this book, there were lots of avenues not taken, options not presented, and subjects not fully explored. In this Appendix, I've created a collection of next steps for those wishing to undertake extra learning and progress their data mining with Python. Consider this Hero mode, the second question, of the book.

This appendix is broken up by chapter, with articles, books, and other resources for learning more about data mining. Also included are some challenges to extend the work performed in the chapter. Some of these will be small improvements; some will be quite a bit more work—I've made a note on those tasks that are noticeably more extensive than the others.

Chapter 1 – Getting Started with Data Mining

Scikit-learn tutorials

`http://scikit-learn.org/stable/tutorial/index.html`

Included in the scikit-learn documentation is a series of tutorials on data mining. The tutorials range from basic introductions to toy datasets, all the way through to comprehensive tutorials on techniques used in recent research.

The tutorials here will take quite a while to get through—they are very comprehensive—but are well worth the effort to learn.

Extending the IPython Notebook

`http://ipython.org/ipython-doc/1/interactive/public_server.html`

The IPython Notebook is a powerful tool. It can be extended in many ways, and one of those is to create a server to run your Notebooks, separately from your main computer. This is very useful if you use a low-power main computer, such as a small laptop, but have more powerful computers at your disposal. In addition, you can set up nodes to perform parallelized computations.More datasets are available at `http://archive.ics.uci.edu/ml/`.

Chapter 2 – Classifying with scikit-learn Estimators

More complex pipelines

`http://scikit-learn.org/stable/modules/pipeline.html#featureunion-composite-feature-spaces`

The Pipelines we have used in the module follow a single stream—the output of one step is the input of another step.

Pipelines follow the transformer and estimator interfaces as well—this allows us to embed Pipelines within Pipelines. This is a useful construct for very complex models, but becomes very powerful when combined with Feature Unions, as shown in the preceding link.

This allows us to extract multiple types of features at a time and then combine them to form a single dataset. For more details, see the example at `http://scikit-learn.org/stable/auto_examples/feature_stacker.html`.

Comparing classifiers

There are lots of classifiers in scikit-learn that are ready to use. The one you choose for a particular task is going to be based on a variety of factors. You can compare the f1-score to see which method is better, and you can investigate the deviation of those scores to see if that result is statistically significant.

An important factor is that they are trained and tested on the same data—that is, the test set for one classifier is the test set for all classifiers. Our use of random states allows us to ensure this is the case—an important factor for replicating experiments.

Chapter 3: Predicting Sports Winners with Decision Trees

More on pandas

The pandas library is a great package—anything you normally write to do data loading is probably already implemented in pandas. You can learn more about it from their tutorial at `http://pandas.pydata.org/pandas-docs/stable/tutorials.html`

There is also a great blog post written by Chris Moffitt that overviews common tasks people do in Excel and how to do them in pandas: `http://pbpython.com/excel-pandas-comp.html`

You can also handle large datasets with pandas; see the answer, from user Jeff (the top answer at the time of writing), to this StackOverflow question for an extensive overview of the process: `http://stackoverflow.com/questions/14262433/large-data-work-flows-using-pandas`.

Another great tutorial on pandas is written by Brian Connelly: `http://bconnelly.net/2013/10/summarizing-data-in-python-with-pandas/`

Chapter 4 – Recommending Movies Using Affinity Analysis

The Eclat algorithm

`http://www.borgelt.net/eclat.html`

The **Apriori** algorithm implemented in this chapter is easily the most famous of the association rule mining graphs, but isn't necessarily the best. Eclat is a more modern algorithm that can be implemented relatively easily.

Chapter 5 – Extracting Features with Transformers

Vowpal Wabbit

`http://hunch.net/~vw/`

Vowpal Wabbit is a great project, providing very fast feature extraction for text-based problems. It comes with a Python wrapper, allowing you to call it from with Python code. Test it out on large datasets, such as the one we used in *Chapter 12*, *Working with Big Data*.

Chapter 6 – Social Media Insight Using Naive Bayes

Natural language processing and part-of-speech tagging

The techniques we used in this chapter were quite lightweight compared to some of the linguistic models employed in other areas. For example, part-of-speech tagging can help disambiguate word forms, allowing for higher accuracy. The book that comes with NLTK has a chapter on this (`http://www.nltk.org/book/ch05.html`). The whole book is well worth reading too.

Chapter 7 – Discovering Accounts to Follow Using Graph Mining

More complex algorithms

There has been extensive research on predicting links in graphs, including for social networks. For instance, David Liben-Nowell and Jon Kleinberg published a paper on this topic that would serve as a great place for more complex algorithms, linked above. It is available at `https://www.cs.cornell.edu/home/kleinber/link-pred.pdf`.

Chapter 8 – Beating CAPTCHAs with Neural Networks

Deeper networks

These techniques will probably fool our current implementation, so improvements will need to be made to make the method better. Try some of the deeper networks we used in *Chapter 11*, *Classifying Objects in Images Using Deep Learning*.

Larger networks need more data, though, so you will probably need to generate more than the few thousand samples we did in this chapter in order to get good performance. Generating these datasets is a good candidate for parallelization—lots of small tasks that can be performed independently.

Reinforcement learning

`http://pybrain.org/docs/tutorial/reinforcement-learning.html`

Reinforcement learning is gaining traction as the next big thing in data mining—although it has been around a long time! PyBrain has some reinforcement learning algorithms that are worth checking out with this dataset (and others!).

Chapter 9 – Authorship Attribution

Local n-grams

Another form of classifier is local n-gram, which involves choosing the best features per-author, not globally for the entire dataset. I wrote a tutorial on using local n-grams for authorship attribution, which is available at `https://github.com/robertlayton/authorship_tutorials/blob/master/LNGTutorial.ipynb`

Chapter 10 – Clustering News Articles

Real-time clusterings

The k-means algorithm can be iteratively trained and updated over time, rather than discrete analyses at given time frames. Cluster movement can be tracked in a number of ways—for instance, you can track which words are popular in each cluster and how much the centroids move per day. Keep the API limits in mind—you probably only need to do one check every few hours to keep your algorithm up-to-date.

Chapter 11 – Classifying Objects in Images Using Deep Learning

Keras and Pylearn2

Other deep learning libraries that are worth looking at, if you are going further with deep learning in Python, are Keras and Pylearn2. They are both based on Theano and have different usages and features.

Keras can be found here: `https://github.com/fchollet/keras/`.

Pylearn2 can be found here: `http://deeplearning.net/software/pylearn2/`.

Both are not stable platforms at the time of writing, although Pylearn2 is the more stable of the two. That said, they both do what they do very well and are worth investigating for future projects.

Another library called Torch is very popular but, at the time of writing, it doesn't have python bindings (see `http://torch.ch/`).

Mahotas

Another package for image processing is Mahotas, including better and more complex image processing techniques that can help achieve better accuracy, although they may come at a high computational cost. However, many image processing tasks are good candidates for parallelization. More techniques on image classification can be found in the research literature, with this survey paper as a good start: `http://luispedro.org/software/mahotas/`.

Chapter 12 – Working with Big Data

Courses on Hadoop

Both Yahoo and Google have great tutorials on Hadoop, which go from beginner to quite advanced levels. They don't specifically address using Python, but learning the Hadoop concepts and then applying them in Pydoop or a similar library can yield great results.

Yahoo's tutorial: `https://developer.yahoo.com/hadoop/tutorial/`

Google's tutorial: `https://cloud.google.com/hadoop/what-is-hadoop`

Pydoop

Pydoop is a python library to run Hadoop jobs—it also has a great tutorial that can be found here: `http://crs4.github.io/pydoop/tutorial/index.html`.

Pydoop also works with HDFS, the Hadoop File System, although you can get that functionality in mrjob as well. Pydoop will give you a bit more control over running some jobs.

Recommendation engine

Building a large recommendation engine is a good test of your Big data skills. A great blog post by Mark Litwintschik covers an engine using Apache Spark, a big data technology: `http://tech.marksblogg.com/recommendation-engine-spark-python.html`.

More resources

Kaggle competitions:

`www.kaggle.com/`

Kaggle runs data mining competitions regularly, often with monetary prizes. Testing your skills on Kaggle competitions is a fast and great way to learn to work with real-world data mining problems. The forums are nice and share environments—often, you will see code released for a top-10 entry during the competition!

Course Module 4
Machine Learning

Course Module 1: Python Fundamentals
Chapter 1: Introduction and First Steps – Take a Deep Breath
Chapter 2: Object-oriented Design
Chapter 3: Objects in Python
Chapter 4: When Objects are alike
Chapter 5: Expecting the Unexpected
Chapter 6: When to use Object-oriented programming
Chapter 7: Python Data Structures
Chapter 8: Python Object-oriented Shortcuts
Chapter 9: Strings and Serialization
Chapter 10: The Iterator Pattern
Chapter 11: Python Design Patterns I
Chapter 12: Python Design Patterns II
Chapter 13: Testing Object-oriented Programs
Chapter 14: Concurrency

Course Module 2: Data Analysis
Chapter 1: Introducing Data Analysis and Libraries
Chapter 2: NumPy Arrays and Vectorized Computation
Chapter 3: Data Analysis with pandas
Chapter 4: Data Visualizaiton
Chapter 5: Time Series
Chapter 6: Interacting with Databases
Chapter 7: Data Analysis Application Examples

Course Module 3: Data Mining
Chapter 1: Getting Started with Data Mining
Chapter 2: Classifying with scikit-learn Estimators
Chapter 3: Predicting Sports Winners with Decision Trees
Chapter 4: Recommending Movies Using Affinity Analysis
Chapter 5: Extracting Features with Transformers
Chapter 6: Social Media Insight Using Naive Bayes
Chapter 7: Discovering Accounts to Follow Using Graph Mining
Chapter 8: Beating CAPTCHAs with Neural Networks
Chapter 9: Authorship Attribution
Chapter 10: Clustering News Articles
Chapter 11: Classifying Objects in Images Using Deep Learning
Chapter 12: Working with Big Data
Chapter 13: Next Steps…

Course Module 4: Machine Learning

Roll up your sleeves and let's get started with advanced machine learning techniques with Course Module 4, Machine Learning

Course Module 4

After exploring data analysis and data mining, let's move ahead to explore one more branch of the data science field, which is machine learning. We've reached our final module, *Machine Learning*. Well, I must say you've been doing well, good job!

Ankita Thakur

Your Course Guide

There is no need to tell you that machine learning has become one of the most exciting technologies of our time and age. Big companies, such as Google, Facebook, Apple, Amazon, IBM, and many more, heavily invest in machine learning research and applications for good reasons. Although it may seem that machine learning has become the buzzword of our time and age, it is certainly not a hype. This exciting field opens the way to new possibilities and has become indispensable to our daily lives. Talking to the voice assistant on our smart phones, recommending the right product for our customers, stopping credit card fraud, filtering out spam from our e-mail inboxes, detecting and diagnosing medical diseases, the list goes on and on. Machine learning and predictive analytics are transforming the way businesses and other organizations operate.

As I've already said that if you thinking to become a machine-learning practitioner, a better problem solver, or maybe even considering a career in machine learning research, then this module is for you. However, for a novice, the theoretical concepts behind machine learning can be quite overwhelming. Yet, many practical books that have been published in recent years will help you get started in machine learning by implementing powerful learning algorithms. In my opinion, the use of practical code examples serve an important purpose. They illustrate the concepts by putting the learned material directly into action. However, remember that *with great power comes great responsibility*! The concepts behind machine learning are too beautiful and important to be hidden in a black box.

Ankita Thakur

Your Course Guide

This module is designed in a way to give you access to the world of predictive analysis and demonstrates why Python is one of the world's leading data science languages. So if you're excited to find out how to use Python to start answering critical questions of your data, pick up this book-whether you want to start from scratch or want to extend your data science knowledge.

1
Giving Computers the Ability to Learn from Data

In my opinion, *machine learning*, the application and science of algorithms that makes sense of data, is the most exciting field of all the computer sciences! We are living in an age where data comes in abundance; using the self-learning algorithms from the field of machine learning, we can turn this data into knowledge. Thanks to the many powerful open source libraries that have been developed in recent years, there has probably never been a better time to break into the machine learning field and learn how to utilize powerful algorithms to spot patterns in data and make predictions about future events.

In this chapter, we will learn about the main concepts and different types of machine learning. Together with a basic introduction to the relevant terminology, we will lay the groundwork for successfully using machine learning techniques for practical problem solving.

In this chapter, we will cover the following topics:

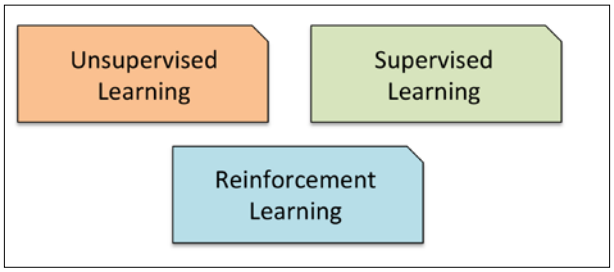
- The general concepts of machine learning
- The three types of learning and basic terminology
- The building blocks for successfully designing machine learning systems

How to transform data into knowledge

In this age of modern technology, there is one resource that we have in abundance: a large amount of structured and unstructured data. In the second half of the twentieth century, machine learning evolved as a subfield of *artificial intelligence* that involved the development of self-learning algorithms to gain knowledge from that data in order to make predictions. Instead of requiring humans to manually derive rules and build models from analyzing large amounts of data, machine learning offers a more efficient alternative for capturing the knowledge in data to gradually improve the performance of predictive models, and make data-driven decisions. Not only is machine learning becoming increasingly important in computer science research but it also plays an ever greater role in our everyday life. Thanks to machine learning, we enjoy robust e-mail spam filters, convenient text and voice recognition software, reliable Web search engines, challenging chess players, and, hopefully soon, safe and efficient self-driving cars.

The three different types of machine learning

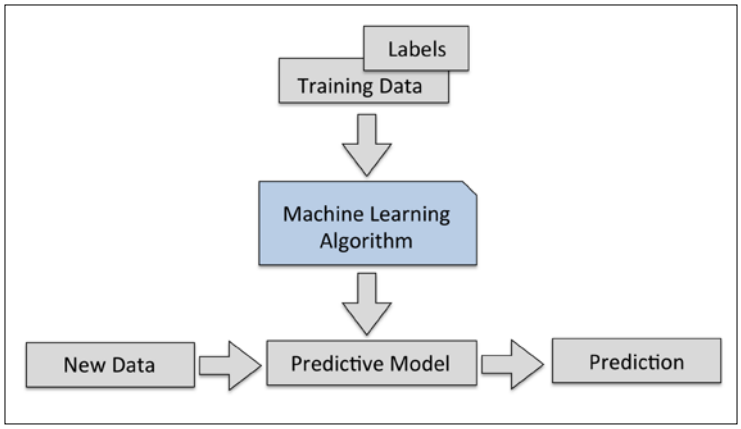
In this section, we will take a look at the three types of machine learning: *supervised learning*, *unsupervised learning*, and *reinforcement learning*. We will learn about the fundamental differences between the three different learning types and, using conceptual examples, we will develop an intuition for the practical problem domains where these can be applied:



Making predictions about the future with supervised learning

The main goal in supervised learning is to learn a model from labeled *training* data that allows us to make predictions about unseen or future data. Here, the term *supervised* refers to a set of samples where the desired output signals (labels) are already known.

Considering the example of e-mail spam filtering, we can train a model using a supervised machine learning algorithm on a corpus of labeled e-mail, e-mail that are correctly marked as spam or not-spam, to predict whether a new e-mail belongs to either of the two categories. A supervised learning task with discrete *class labels*, such as in the previous e-mail spam-filtering example, is also called a *classification* task. Another subcategory of supervised learning is *regression*, where the outcome signal is a continuous value:

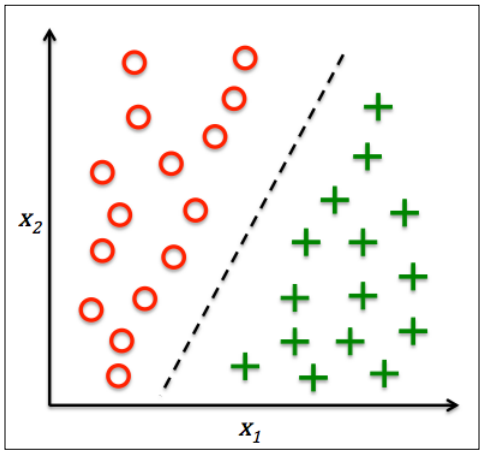


Classification for predicting class labels

Classification is a subcategory of supervised learning where the goal is to predict the categorical class labels of new instances based on past observations. Those class labels are discrete, unordered values that can be understood as the *group memberships* of the instances. The previously mentioned example of e-mail-spam detection represents a typical example of a *binary classification* task, where the machine learning algorithm learns a set of rules in order to distinguish between two possible classes: spam and non-spam e-mail.

However, the set of class labels does not have to be of a binary nature. The predictive model learned by a supervised learning algorithm can assign any class label that was presented in the training dataset to a new, unlabeled instance. A typical example of a *multi-class classification* task is handwritten character recognition. Here, we could collect a training dataset that consists of multiple handwritten examples of each letter in the alphabet. Now, if a user provides a new handwritten character via an input device, our predictive model will be able to predict the correct letter in the alphabet with certain accuracy. However, our machine learning system would be unable to correctly recognize any of the digits zero to nine, for example, if they were not part of our training dataset.

The following figure illustrates the concept of a binary classification task given 30 training samples: 15 training samples are labeled as *negative class* (circles) and 15 training samples are labeled as *positive class* (plus signs). In this scenario, our dataset is two-dimensional, which means that each sample has two values associated with it: x_1 and x_2. Now, we can use a supervised machine learning algorithm to learn a rule—the decision boundary represented as a black dashed line—that can separate those two classes and classify new data into each of those two categories given its x_1 and x_2 values:



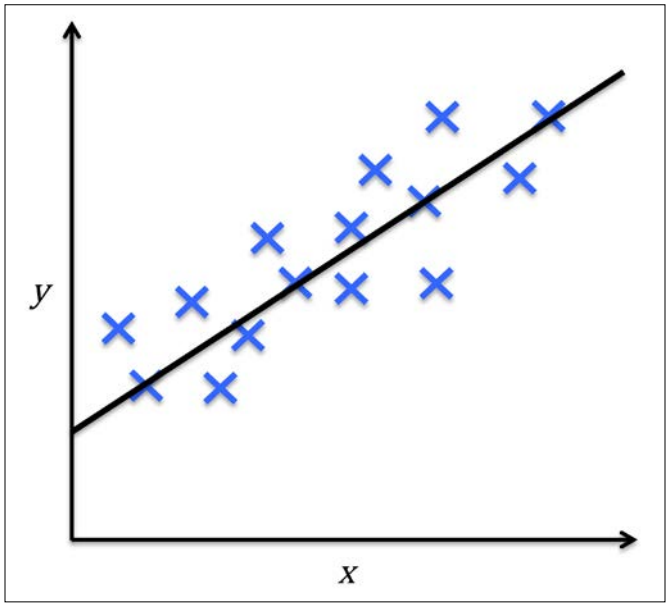
Regression for predicting continuous outcomes

We learned in the previous section that the task of classification is to assign categorical, unordered labels to instances. A second type of supervised learning is the prediction of continuous outcomes, which is also called regression analysis. In *regression analysis*, we are given a number of *predictor* (explanatory) variables and a continuous response variable (outcome), and we try to find a relationship between those variables that allows us to predict an outcome.

For example, let's assume that we are interested in predicting the Math SAT scores of our students. If there is a relationship between the time spent studying for the test and the final scores, we could use it as training data to learn a model that uses the study time to predict the test scores of future students who are planning to take this test.

The term *regression* was devised by Francis Galton in his article *Regression Towards Mediocrity in Hereditary Stature* in 1886. Galton described the biological phenomenon that the variance of *height* in a population does not increase over time. He observed that the height of parents is not passed on to their children but the children's height is regressing towards the population mean.

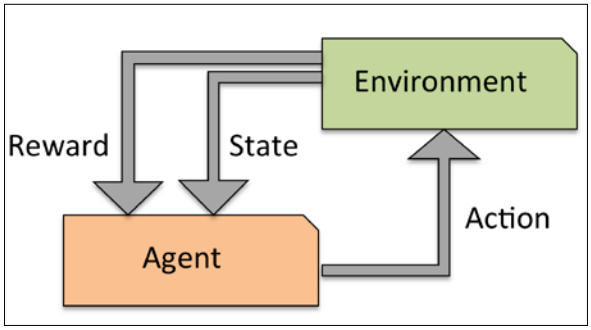
The following figure illustrates the concept of *linear regression*. Given a predictor variable x and a response variable y, we fit a straight line to this data that minimizes the distance—most commonly the average squared distance—between the sample points and the fitted line. We can now use the intercept and slope learned from this data to predict the outcome variable of new data:



Solving interactive problems with reinforcement learning

Another type of machine learning is reinforcement learning. In reinforcement learning, the goal is to develop a system (*agent*) that improves its performance based on interactions with the *environment*. Since the information about the current state of the environment typically also includes a so-called *reward* signal, we can think of reinforcement learning as a field related to *supervised* learning. However, in reinforcement learning this feedback is not the correct ground truth label or value, but a measure of how well the action was measured by a *reward* function. Through the interaction with the environment, an agent can then use reinforcement learning to learn a series of actions that maximizes this reward via an exploratory trial-and-error approach or deliberative planning.

A popular example of reinforcement learning is a chess engine. Here, the agent decides upon a series of moves depending on the state of the board (the environment), and the reward can be defined as *win* or *lose* at the end of the game:



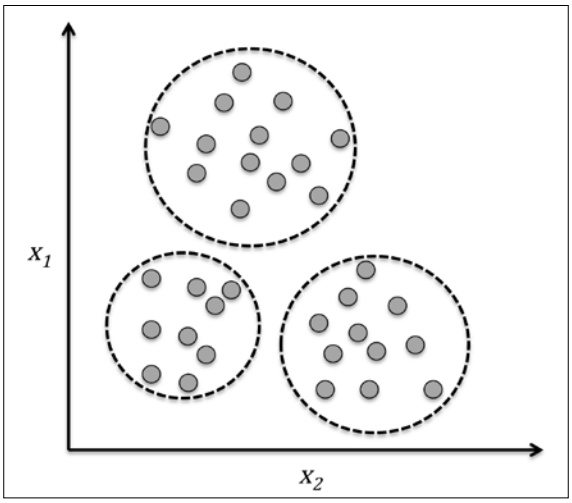
Discovering hidden structures with unsupervised learning

In supervised learning, we know the *right answer* beforehand when we train our model, and in reinforcement learning, we define a measure of *reward* for particular actions by the agent. In unsupervised learning, however, we are dealing with unlabeled data or data of *unknown structure*. Using unsupervised learning techniques, we are able to explore the structure of our data to extract meaningful information without the guidance of a known outcome variable or reward function.

Finding subgroups with clustering

Clustering is an exploratory data analysis technique that allows us to organize a pile of information into meaningful subgroups (*clusters*) without having any prior knowledge of their group memberships. Each cluster that may arise during the analysis defines a group of objects that share a certain degree of similarity but are more dissimilar to objects in other clusters, which is why clustering is also sometimes called "unsupervised classification." Clustering is a great technique for structuring information and deriving meaningful relationships among data, For example, it allows marketers to discover customer groups based on their interests in order to develop distinct marketing programs.

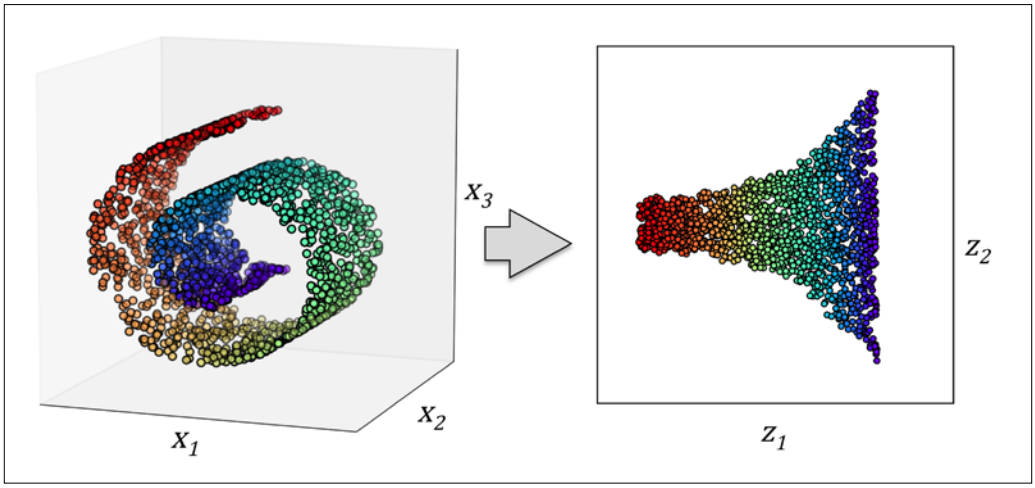
The figure below illustrates how clustering can be applied to organizing unlabeled data into three distinct groups based on the similarity of their features x_1 and x_2:

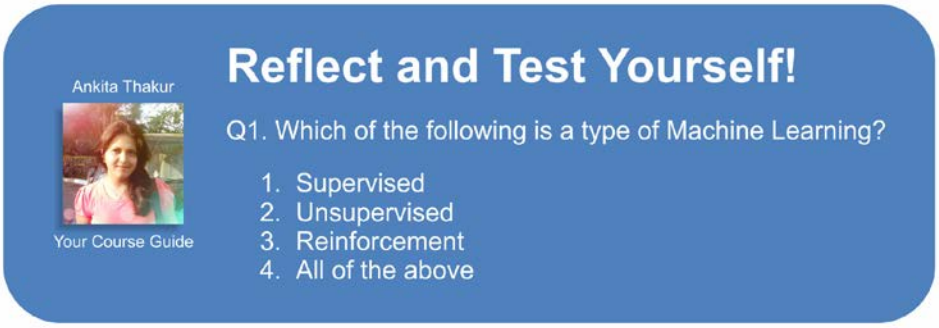


Dimensionality reduction for data compression

Another subfield of unsupervised learning is *dimensionality reduction*. Often we are working with data of high dimensionality—each observation comes with a high number of measurements—that can present a challenge for limited storage space and the computational performance of machine learning algorithms. Unsupervised dimensionality reduction is a commonly used approach in feature preprocessing to remove noise from data, which can also degrade the predictive performance of certain algorithms, and compress the data onto a smaller dimensional subspace while retaining most of the relevant information.

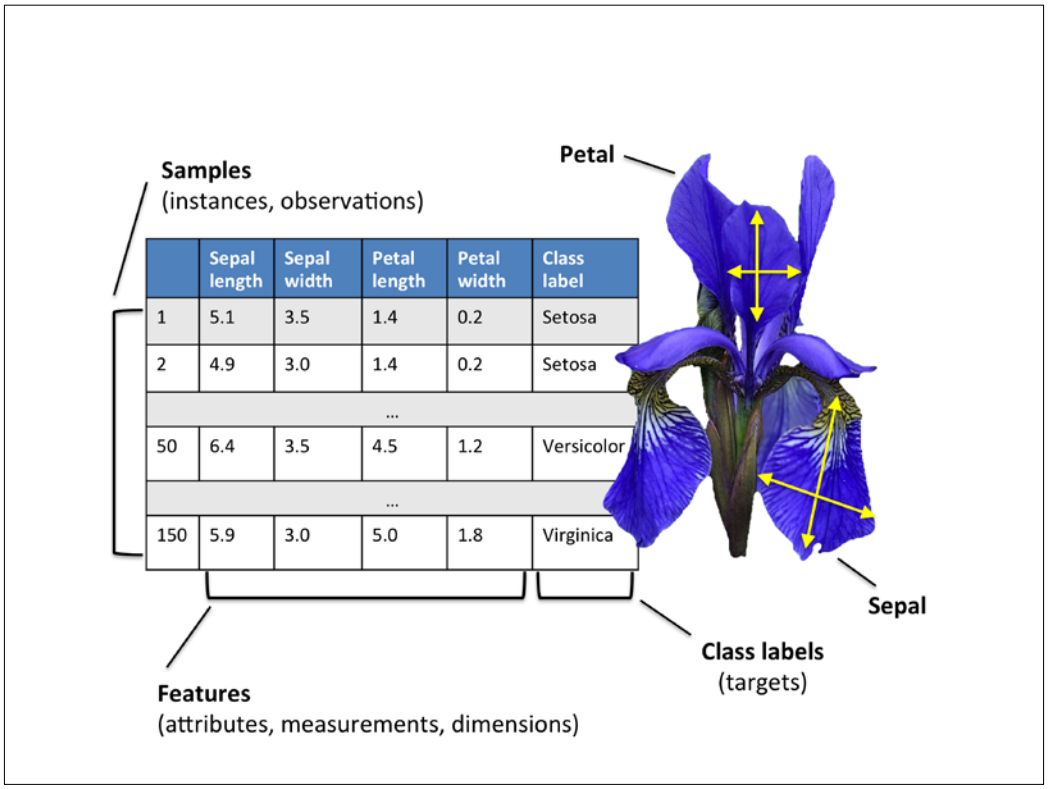
Sometimes, dimensionality reduction can also be useful for visualizing data—for example, a high-dimensional feature set can be projected onto one-, two-, or three-dimensional feature spaces in order to visualize it via 3D- or 2D-scatterplots or histograms. The figure below shows an example where non-linear dimensionality reduction was applied to compress a 3D *Swiss Roll* onto a new 2D feature subspace:





An introduction to the basic terminology and notations

Now that we have discussed the three broad categories of machine learning—supervised, unsupervised, and reinforcement learning—let us have a look at the basic terminology that we will be using in the next chapters. The following table depicts an excerpt of the *Iris* dataset, which is a classic example in the field of machine learning. The Iris dataset contains the measurements of 150 iris flowers from three different species: *Setosa*, *Versicolor*, and *Virginica*. Here, each flower sample represents one row in our data set, and the flower measurements in centimeters are stored as columns, which we also call the features of the dataset:



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

To keep the notation and implementation simple yet efficient, we will make use of some of the basics of *linear algebra*. In the following chapters, we will use a *matrix* and *vector* notation to refer to our data. We will follow the common convention to represent each sample as separate row in a feature matrix X, where each feature is stored as a separate column.

The Iris dataset, consisting of 150 samples and 4 features, can then be written as a 150×4 matrix $\boldsymbol{X} \in \mathbb{R}^{\mathbf{150 \times 4}}$:

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

For the rest of this module, we will use the superscript *(i)* to refer to the *i*th training sample, and the subscript *j* to refer to the *j*th dimension of the training dataset.

We use lower-case, bold-face letters to refer to vectors $\left(\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{n \times 1}}\right)$ and upper-case, bold-face letters to refer to matrices, respectively $(\boldsymbol{X} \in \mathbb{R}^{\boldsymbol{n \times m}})$. To refer to single elements in a vector or matrix, we write the letters in italics ($x^{(n)}$ or $x_{(m)}^{(n)}$, respectively).

For example, x_1^{150} refers to the first dimension of flower sample 150, the *sepal length*. Thus, each row in this feature matrix represents one flower instance and can be written as four-dimensional row vector $\boldsymbol{x}^{(i)} \in \mathbb{R}^{\mathbf{1 \times 4}}$, $\boldsymbol{x}^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix}$.

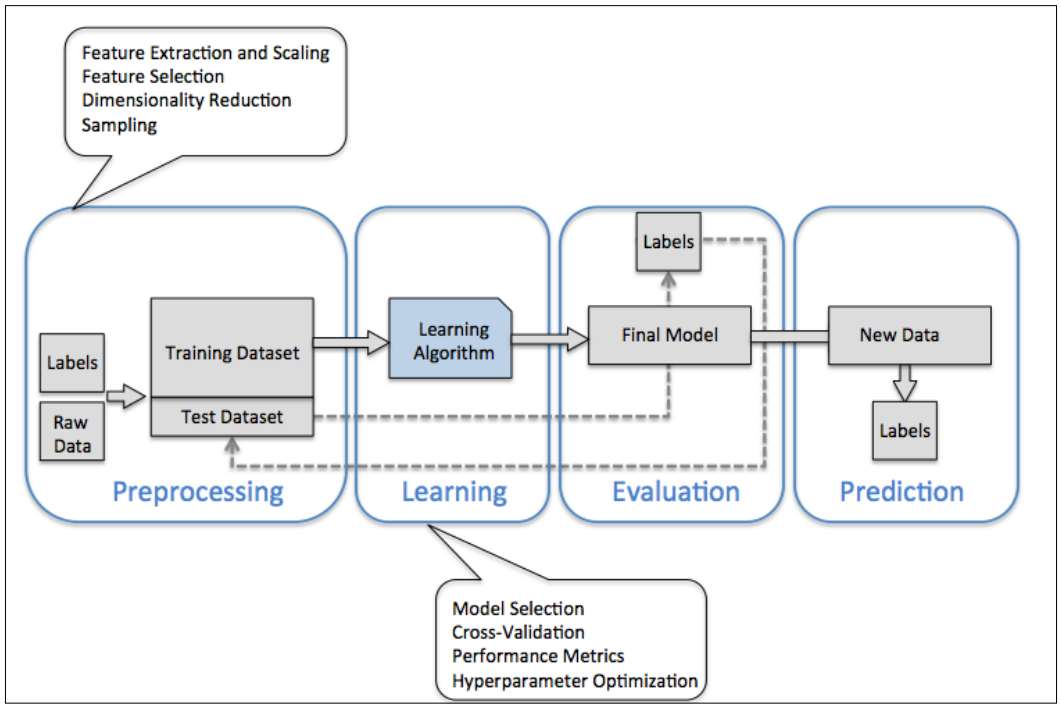
Each feature dimension is a 150-dimensional column vector $\boldsymbol{x}_j \in \mathbb{R}^{\mathbf{150 \times 1}}$, for example:

$$\boldsymbol{x}_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix}.$$

Similarly, we store the target variables (here: class labels) as a 150-dimensional column vector $\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(150)} \end{bmatrix} \left(y \in \{\text{Setosa, Versicolor, Virginica}\}\right)$.

A roadmap for building machine learning systems

In the previous sections, we discussed the basic concepts of machine learning and the three different types of learning. In this section, we will discuss other important parts of a machine learning system accompanying the learning algorithm. The diagram below shows a typical workflow diagram for using machine learning in *predictive modeling*, which we will discuss in the following subsections:



Preprocessing – getting data into shape

Raw data rarely comes in the form and shape that is necessary for the optimal performance of a learning algorithm. Thus, the *preprocessing* of the data is one of the most crucial steps in any machine learning application. If we take the Iris flower dataset from the previous section as an example, we could think of the raw data as a series of flower images from which we want to extract meaningful features. Useful features could be the color, the hue, the intensity of the flowers, the height, and the flower lengths and widths. Many machine learning algorithms also require that the selected features are on the same scale for optimal performance, which is often achieved by transforming the features in the range [0, 1] or a standard normal distribution with zero mean and unit variance, as we will see in the later chapters.

Some of the selected features may be highly correlated and therefore redundant to a certain degree. In those cases, dimensionality reduction techniques are useful for compressing the features onto a lower dimensional subspace. Reducing the dimensionality of our feature space has the advantage that less storage space is required, and the learning algorithm can run much faster.

To determine whether our machine learning algorithm not only performs well on the training set but also generalizes well to new data, we also want to randomly divide the dataset into a separate training and test set. We use the training set to train and optimize our machine learning model, while we keep the test set until the very end to evaluate the final model.

Training and selecting a predictive model

As we will see in later chapters, many different machine learning algorithms have been developed to solve different problem tasks. An important point that can be summarized from David Wolpert's famous *No Free Lunch Theorems* is that we can't get learning "for free" (*The Lack of A Priori Distinctions Between Learning Algorithms*, D.H. Wolpert 1996; *No Free Lunch Theorems for Optimization*, D.H. Wolpert and W.G. Macready, 1997). Intuitively, we can relate this concept to the popular saying, "*I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail*" (Abraham Maslow, 1966). For example, each classification algorithm has its inherent biases, and no single classification model enjoys superiority if we don't make any assumptions about the task. In practice, it is therefore essential to compare at least a handful of different algorithms in order to train and select the best performing model. But before we can compare different models, we first have to decide upon a metric to measure performance. One commonly used metric is classification accuracy, which is defined as the proportion of correctly classified instances.

One legitimate question to ask is: *how do we know which model performs well on the final test dataset and real-world data if we don't use this test set for the model selection but keep it for the final model evaluation?* In order to address the issue embedded in this question, different cross-validation techniques can be used where the training dataset is further divided into training and *validation subsets* in order to estimate the *generalization performance* of the model. Finally, we also cannot expect that the default parameters of the different learning algorithms provided by software libraries are optimal for our specific problem task. Therefore, we will make frequent use of hyperparameter *optimization techniques* that help us to fine-tune the performance of our model in later chapters. Intuitively, we can think of those hyperparameters as parameters that are not learned from the data but represent the knobs of a model that we can turn to improve its performance, which will become much clearer in later chapters when we see actual examples.

Evaluating models and predicting unseen data instances

After we have selected a model that has been fitted on the training dataset, we can use the test dataset to estimate how well it performs on this unseen data to estimate the generalization error. If we are satisfied with its performance, we can now use this model to predict new, future data. It is important to note that the parameters for the previously mentioned procedures—such as feature scaling and dimensionality reduction—are solely obtained from the training dataset, and the same parameters are later re-applied to transform the test dataset, as well as any new data samples—the performance measured on the test data may be overoptimistic otherwise.

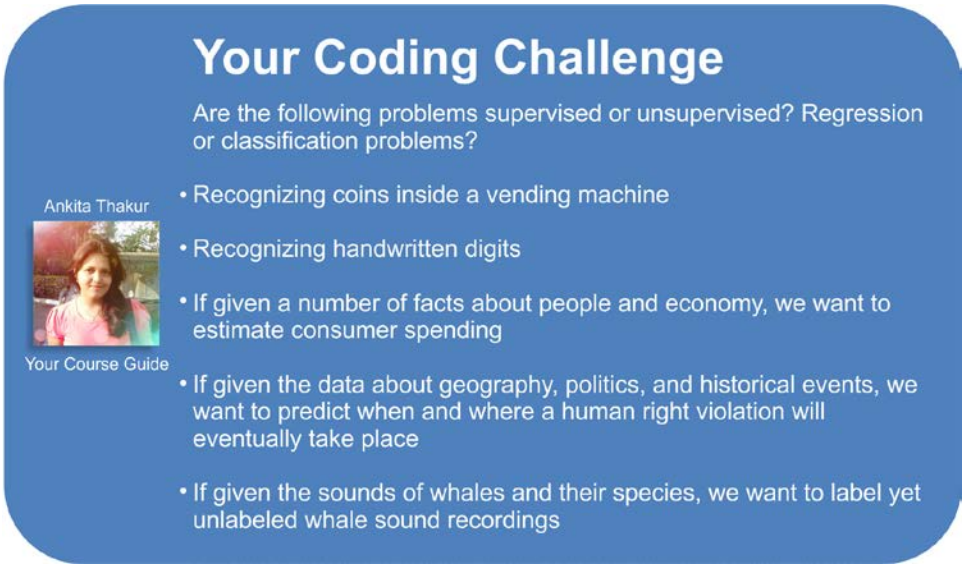
Using Python for machine learning

Python is one of the most popular programming languages for data science and therefore enjoys a large number of useful add-on libraries developed by its great community.

Although the performance of interpreted languages, such as Python, for computation-intensive tasks is inferior to lower-level programming languages, extension libraries such as *NumPy* and *SciPy* have been developed that build upon lower layer Fortran and C implementations for fast and vectorized operations on multidimensional arrays.

For machine learning programming tasks, we will mostly refer to the *scikit-learn* library, which is one of the most popular and accessible open source machine learning libraries as of today.


Reflect and Test Yourself!
Ankita Thakur
Your Course Guide
Q2. Which of the following is the proper order to prepare a machine learning application?
1. Evaluation, Learning, Preprocessing, and Prediction
2. Learning, Prediction, Evaluation, and Preprocessing
3. Preprocessing, Learning, Evaluation, and Prediction
4. Preprocessing, Evaluation, Learning, and Prediction


Your Coding Challenge
Are the following problems supervised or unsupervised? Regression or classification problems?
Ankita Thakur
Your Course Guide
• Recognizing coins inside a vending machine
• Recognizing handwritten digits
• If given a number of facts about people and economy, we want to estimate consumer spending
• If given the data about geography, politics, and historical events, we want to predict when and where a human right violation will eventually take place
• If given the sounds of whales and their species, we want to label yet unlabeled whale sound recordings

Summary of Module 4 Chapter 1

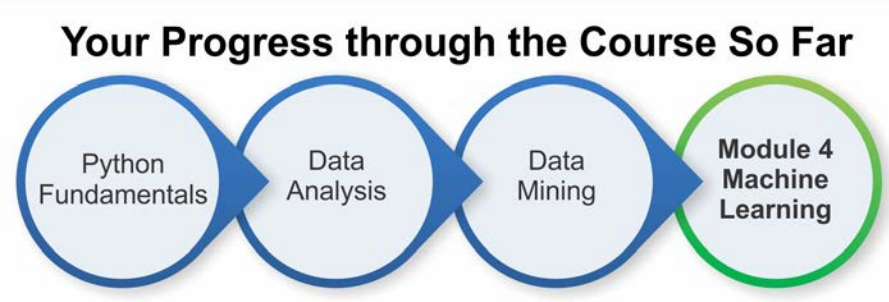
In this chapter, we explored machine learning on a very high level and familiarized ourselves with the big picture and major concepts that we are going to explore in the next chapters in more detail.

We learned that supervised learning is composed of two important subfields: classification and regression. While classification models allow us to categorize objects into known classes, we can use regression analysis to predict the continuous outcomes of target variables. Unsupervised learning not only offers useful techniques for discovering structures in unlabeled data, but it can also be useful for data compression in feature preprocessing steps.

We briefly went over the typical roadmap for applying machine learning to problem tasks, which we will use as a foundation for deeper discussions and hands-on examples in the following chapters.

Ankita Thakur

Your Course Guide



2
Training Machine Learning Algorithms for Classification

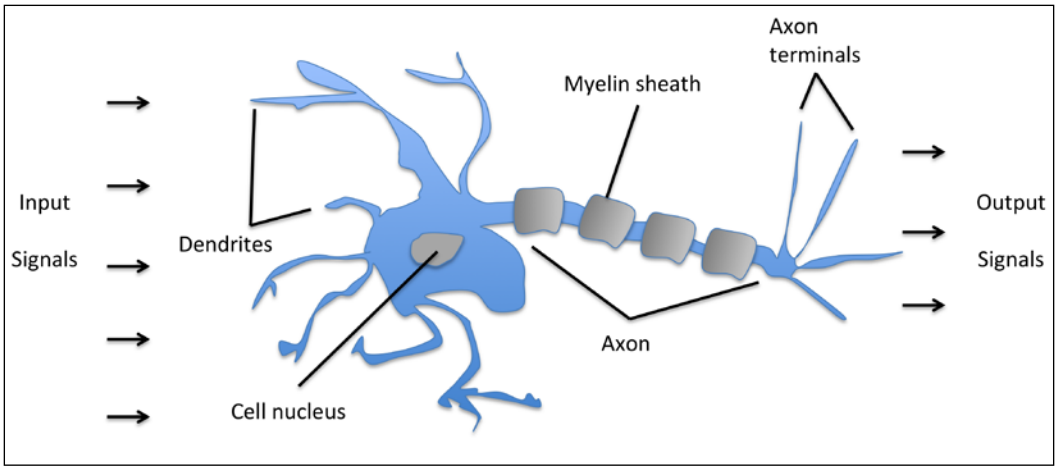
In this chapter, we will make use of one of the first algorithmically described machine learning algorithms for classification, the *perceptron* and *adaptive linear neurons*. We will start by implementing a perceptron step by step in Python and training it to classify different flower species in the Iris dataset. This will help us to understand the concept of machine learning algorithms for classification and how they can be efficiently implemented in Python. Discussing the basics of optimization using adaptive linear neurons will then lay the groundwork for using more powerful classifiers via the scikit-learn machine-learning library in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*.

The topics that we will cover in this chapter are as follows:

- Building an intuition for machine learning algorithms
- Using pandas, NumPy, and matplotlib to read in, process, and visualize data
- Implementing linear classification algorithms in Python

Artificial neurons – a brief glimpse into the early history of machine learning

Before we discuss the perceptron and related algorithms in more detail, let us take a brief tour through the early beginnings of machine learning. Trying to understand how the biological brain works to design artificial intelligence, Warren McCullock and Walter Pitts published the first concept of a simplified brain cell, the so-called *McCullock-Pitts (MCP) neuron*, in 1943 (W. S. McCulloch and W. Pitts. *A Logical Calculus of the Ideas Immanent in Nervous Activity*. The bulletin of mathematical biophysics, 5(4):115–133, 1943). Neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals, which is illustrated in the following figure:



McCullock and Pitts described such a nerve cell as a simple logic gate with binary outputs; multiple signals arrive at the dendrites, are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

Only a few years later, Frank Rosenblatt published the first concept of the perceptron learning rule based on the MCP neuron model (F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton*. Cornell Aeronautical Laboratory, 1957). With his perceptron rule, Rosenblatt proposed an algorithm that would automatically learn the optimal weight coefficients that are then multiplied with the input features in order to make the decision of whether a neuron fires or not. In the context of supervised learning and classification, such an algorithm could then be used to predict if a sample belonged to one class or the other.

More formally, we can pose this problem as a binary classification task where we refer to our two classes as `1` (positive class) and `-1` (negative class) for simplicity. We can then define an *activation function* $\phi(z)$ that takes a linear combination of certain input values $\boldsymbol{x}$ and a corresponding weight vector $\boldsymbol{w}$, where z is the so-called net input ($z = w_1x_1 + \ldots + w_mx_m$):

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \ \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

Now, if the activation of a particular sample $\boldsymbol{x}^{(i)}$, that is, the output of $\phi(z)$, is greater than a defined threshold θ, we predict class 1 and class -1, otherwise. In the perceptron algorithm, the activation function $\phi(\cdot)$ is a simple *unit step function*, which is sometimes also called the *Heaviside step function*:

$$\phi(z) = \begin{cases} 1 \ \ if \ z \geq \theta \\ -1 \ \ otherwise \end{cases}$$

For simplicity, we can bring the threshold θ to the left side of the equation and define a weight-zero as $w_0 = -\theta$ and $x_0 = 1$, so that we write $\mathbf{z}$ in a more compact form $z = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \boldsymbol{w}^T \boldsymbol{x}$ and $\phi(z) = \begin{cases} 1 \ \textit{if } z \geq 0 \\ -1 \ \textit{otherwise} \end{cases}$.

In the following sections, we will often make use of basic notations from linear algebra. For example, we will abbreviate the sum of the products of the values in $\boldsymbol{x}$ and $\boldsymbol{w}$ using a *vector dot product*, whereas superscript **T** stands for *transpose*, which is an operation that transforms a column vector into a row vector and vice versa:

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} \boldsymbol{x}_j \boldsymbol{w}_j = \boldsymbol{w}^T \boldsymbol{x}$$

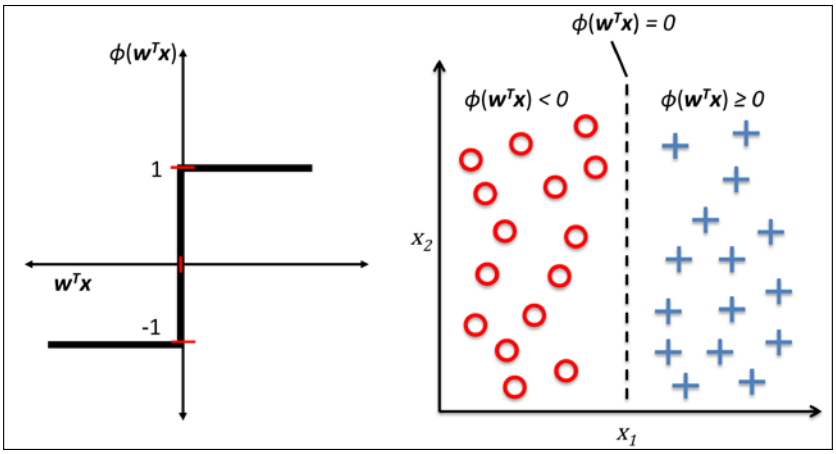
For example: $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$.

Furthermore, the transpose operation can also be applied to a matrix to reflect it over its diagonal, for example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

In this book, we will only use the very basic concepts from linear algebra. However, if you need a quick refresher, please take a look at Zico Kolter's excellent Linear Algebra Review and Reference, which is freely available at `http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf`.

The following figure illustrates how the net input $z = \boldsymbol{w}^T \boldsymbol{x}$ is squashed into a binary output (-1 or 1) by the activation function of the perceptron (left subfigure) and how it can be used to discriminate between two linearly separable classes (right subfigure):



The whole idea behind the MCP neuron and Rosenblatt's *thresholded* perceptron model is to use a reductionist approach to mimic how a single neuron in the brain works: it either *fires* or it doesn't. Thus, Rosenblatt's initial perceptron rule is fairly simple and can be summarized by the following steps:

1. Initialize the weights to 0 or small random numbers.
2. For each training sample $\boldsymbol{x}^{(i)}$ perform the following steps:
    1. Compute the output value $\hat{y}$.
    2. Update the weights.

Here, the output value is the class label predicted by the unit step function that we defined earlier, and the simultaneous update of each weight w_j in the weight vector $\boldsymbol{w}$ can be more formally written as:

$$w_j := w_j + \Delta w_j$$

The value of Δw_j, which is used to update the weight w_j, is calculated by the perceptron learning rule:

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$

Where η is the learning rate (a constant between 0.0 and 1.0), $y^{(i)}$ is the true class label of the i th training sample, and $\hat{y}^{(i)}$ is the predicted class label. It is important to note that all weights in the weight vector are being updated simultaneously, which means that we don't recompute the $\hat{y}^{(i)}$ before all of the weights Δw_j were updated. Concretely, for a 2D dataset, we would write the update as follows:

$$\Delta w_0 = \eta\left(y^{(i)} - output^{(i)}\right)$$

$$\Delta w_1 = \eta\left(y^{(i)} - output^{(i)}\right)x_1^{(i)}$$

$$\Delta w_2 = \eta\left(y^{(i)} - output^{(i)}\right)x_2^{(i)}$$

Before we implement the perceptron rule in Python, let us make a simple thought experiment to illustrate how beautifully simple this learning rule really is. In the two scenarios where the perceptron predicts the class label correctly, the weights remain unchanged:

$$\Delta w_j = \eta\left(-1--1\right)x_j^{(i)} = 0$$

$$\Delta w_j = \eta\left(1-1\right)x_j^{(i)} = 0$$

However, in the case of a wrong prediction, the weights are being pushed towards the direction of the positive or negative target class, respectively:

$$\Delta w_j = \eta\left(1--1\right)x_j^{(i)} = \eta\left(2\right)x_j^{(i)}$$

$$\Delta w_j = \eta\left(-1-1\right)x_j^{(i)} = \eta\left(-2\right)x_j^{(i)}$$

To get a better intuition for the multiplicative factor $x_j^{(i)}$, let us go through another simple example, where:

$$y^{(i)} = +1,\ \hat{y}_j^{(i)} = -1,\ \eta = 1$$

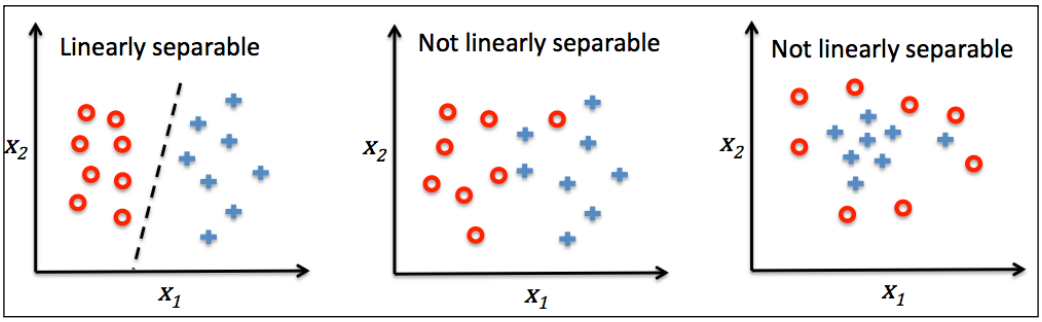
Let's assume that $x_j^{(i)} = 0.5$, and we misclassify this sample as -1. In this case, we would increase the corresponding weight by 1 so that the activation $x_j^{(i)} \times w_j^{(i)}$ will be more positive the next time we encounter this sample and thus will be more likely to be above the threshold of the unit step function to classify the sample as +1:

$$\Delta w_j^{(i)} = \left(1 - -1\right) 0.5 = \left(2\right) 0.5 = 1$$

The weight update is proportional to the value of $x_j^{(i)}$. For example, if we have another sample $x_j^{(i)} = 2$ that is incorrectly classified as -1, we'd push the decision boundary by an even larger extent to classify this sample correctly the next time:

$$\Delta w_j = \left(1 - -1\right) 2 = \left(2\right) 2 = 4$$

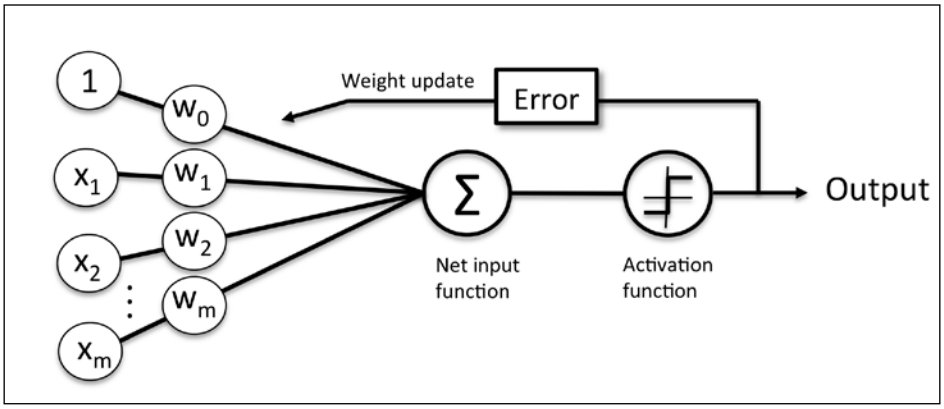
It is important to note that the convergence of the perceptron is only guaranteed if the two classes are linearly separable and the learning rate is sufficiently small. If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (*epochs*) and/or a threshold for the number of tolerated misclassifications—the perceptron would never stop updating the weights otherwise:



Downloading the example code

The code files for all the four parts of the course are available at `https://github.com/PacktPublishing/Data-Science-With-Python`.

Now, before we jump into the implementation in the next section, let us summarize what we just learned in a simple figure that illustrates the general concept of the perceptron:



The preceding figure illustrates how the perceptron receives the inputs of a sample $\boldsymbol{x}$ and combines them with the weights $\boldsymbol{w}$ to compute the net input. The net input is then passed on to the activation function (here: the unit step function), which generates a binary output -1 or +1—the predicted class label of the sample. During the learning phase, this output is used to calculate the error of the prediction and update the weights.

Implementing a perceptron learning algorithm in Python

In the previous section, we learned how Rosenblatt's perceptron rule works; let us now go ahead and implement it in Python and apply it to the Iris dataset that we introduced in *Chapter 1, Giving Computers the Ability to Learn from Data*. We will take an objected-oriented approach to define the perceptron interface as a Python `Class`, which allows us to initialize new perceptron objects that can learn from data via a `fit` method, and make predictions via a separate `predict` method. As a convention, we add an underscore to attributes that are not being created upon the initialization of the object but by calling the object's other methods—for example, `self.w_`.

If you are not yet familiar with Python's scientific libraries or need a refresher, please see the following resources:

NumPy: `http://wiki.scipy.org/Tentative_NumPy_Tutorial`

Pandas: `http://pandas.pydata.org/pandas-docs/stable/tutorials.html`

Matplotlib: `http://matplotlib.org/users/beginner.html`

Also, to better follow the code examples, I recommend you download the IPython notebooks from the Packt website. For a general introduction to IPython notebooks, please visit `https://ipython.org/ipython-doc/3/notebook/index.html`.

```
import numpy as np
class Perceptron(object):
    """Perceptron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclassifications in every epoch.

    """
    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples
            is the number of samples and
```

```
            n_features is the number of features.
        y : array-like, shape = [n_samples]
            Target values.

        Returns
        -------
        self : object

        """
        self.w_ = np.zeros(1 + X.shape[1])
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Using this perceptron implementation, we can now initialize new `Perceptron` objects with a given learning rate `eta` and `n_iter`, which is the number of epochs (passes over the training set). Via the `fit` method we initialize the weights in `self.w_` to a zero-vector $\mathbb{R}^{m+1}$ where m stands for the number of dimensions (features) in the dataset where we add 1 for the zero-weight (that is, the threshold).

NumPy indexing for one-dimensional arrays works similarly to Python lists using the square-bracket (`[]`) notation. For two-dimensional arrays, the first indexer refers to the row number, and the second indexer to the column number. For example, we would use `X[2, 3]` to select the third row and fourth column of a 2D array `X`.

After the weights have been initialized, the `fit` method loops over all individual samples in the training set and updates the weights according to the perceptron learning rule that we discussed in the previous section. The class labels are predicted by the `predict` method, which is also called in the `fit` method to predict the class label for the weight update, but `predict` can also be used to predict the class labels of new data after we have fitted our model. Furthermore, we also collect the number of misclassifications during each epoch in the list `self.errors_` so that we can later analyze how well our perceptron performed during the training. The `np.dot` function that is used in the `net_input` method simply calculates the vector dot product $\boldsymbol{w}^T\boldsymbol{x}$.

Instead of using NumPy to calculate the vector dot product between two arrays a and b via `a.dot(b)` or `np.dot(a, b)`, we could also perform the calculation in pure Python via `sum([i*j for i,j in zip(a, b)]`. However, the advantage of using NumPy over classic Python for-loop structures is that its arithmetic operations are vectorized. **Vectorization** means that an elemental arithmetic operation is automatically applied to all elements in an array. By formulating our arithmetic operations as a sequence of instructions on an array rather than performing a set of operations for each element one at a time, we can make better use of our modern CPU architectures with **Single Instruction, Multiple Data** (**SIMD**) support. Furthermore, NumPy uses highly optimized linear algebra libraries, such as **Basic Linear Algebra Subprograms** (**BLAS**) and **Linear Algebra Package** (**LAPACK**) that have been written in C or Fortran. Lastly, NumPy also allows us to write our code in a more compact and intuitive way using the basics of linear algebra, such as vector and matrix dot products.

Training a perceptron model on the Iris dataset

To test our perceptron implementation, we will load the two flower classes *Setosa* and *Versicolor* from the Iris dataset. Although, the perceptron rule is not restricted to two dimensions, we will only consider the two features *sepal length* and *petal length* for visualization purposes. Also, we only chose the two flower classes *Setosa* and *Versicolor* for practical reasons. However, the perceptron algorithm can be extended to multi-class classification—for example, through the *One-vs.-All* technique.

One-vs.-All (**OvA**), or sometimes also called **One-vs.-Rest** (**OvR**), is a technique used to extend a binary classifier to multi-class problems. Using OvA, we can train one classifier per class, where the particular class is treated as the positive class and the samples from all other classes are considered as the negative class. If we were to classify a new data sample, we would use our n classifiers, where n is the number of class labels, and assign the class label with the highest confidence to the particular sample. In the case of the perceptron, we would use OvA to choose the class label that is associated with the largest absolute net input value.

First, we will use the *pandas* library to load the Iris dataset directly from the *UCI Machine Learning Repository* into a `DataFrame` object and print the last five lines via the `tail` method to check that the data was loaded correctly:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'
...   'machine-learning-databases/iris/iris.data', header=None)
>>> df.tail()
```

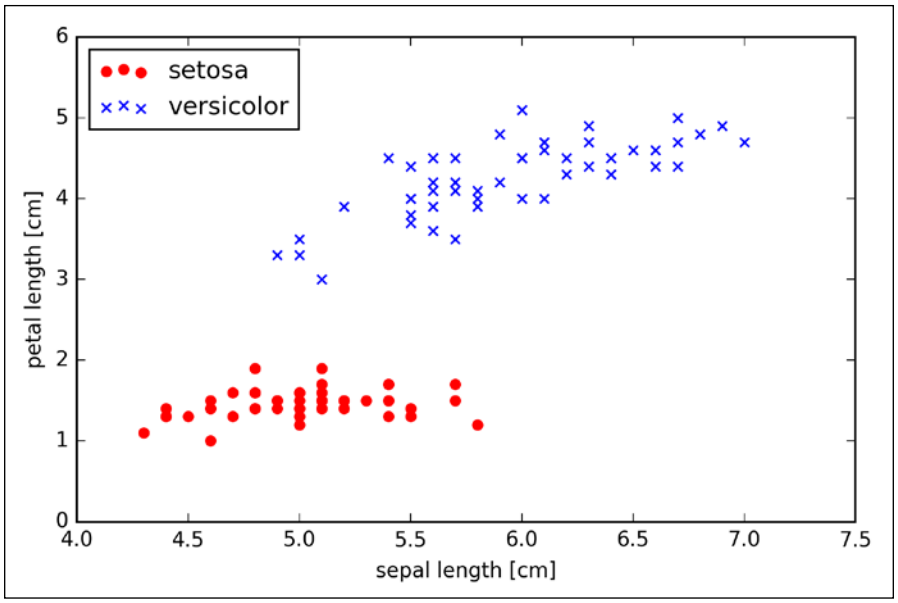
	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Next, we extract the first 100 class labels that correspond to the 50 *Iris-Setosa* and 50 *Iris-Versicolor* flowers, respectively, and convert the class labels into the two integer class labels `1` (*Versicolor*) and `-1` (*Setosa*) that we assign to a vector `y` where the values method of a pandas `DataFrame` yields the corresponding NumPy representation. Similarly, we extract the first feature column (*sepal length*) and the third feature column (*petal length*) of those 100 training samples and assign them to a feature matrix `X`, which we can visualize via a two-dimensional scatter plot:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np

>>> y = df.iloc[0:100, 4].values
```

```
>>> y = np.where(y == 'Iris-setosa', -1, 1)
>>> X = df.iloc[0:100, [0, 2]].values
>>> plt.scatter(X[:50, 0], X[:50, 1],
...             color='red', marker='o', label='setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...             color='blue', marker='x', label='versicolor')
>>> plt.xlabel('sepal length')
>>> plt.ylabel('petal length')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example we should now see the following scatterplot:

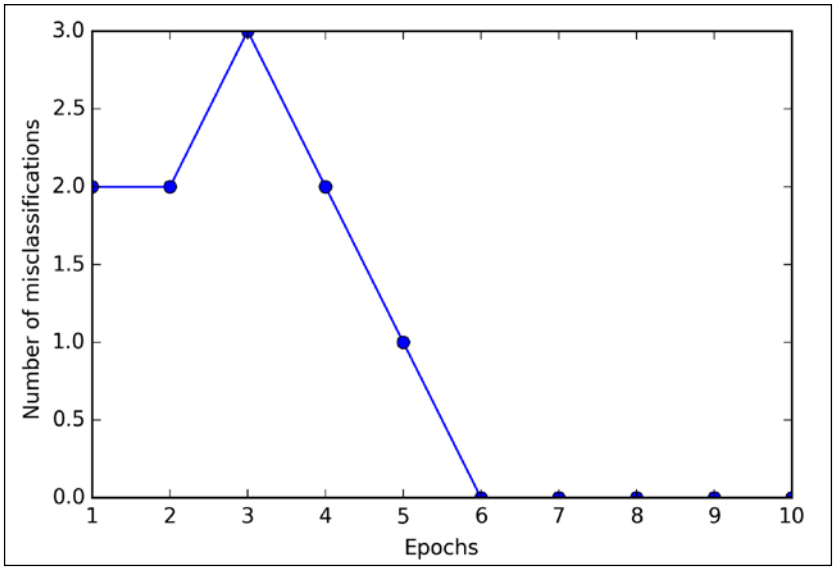


Now it's time to train our perceptron algorithm on the Iris data subset that we just extracted. Also, we will plot the `misclassification error` for each epoch to check if the algorithm converged and found a decision boundary that separates the two Iris flower classes:

```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_,
```

```
...          marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Number of misclassifications')
>>> plt.show()
```

After executing the preceding code, we should see the plot of the misclassification errors versus the number of epochs, as shown next:



As we can see in the preceding plot, our perceptron already converged after the sixth epoch and should now be able to classify the training samples perfectly. Let us implement a small convenience function to visualize the decision boundaries for 2D datasets:

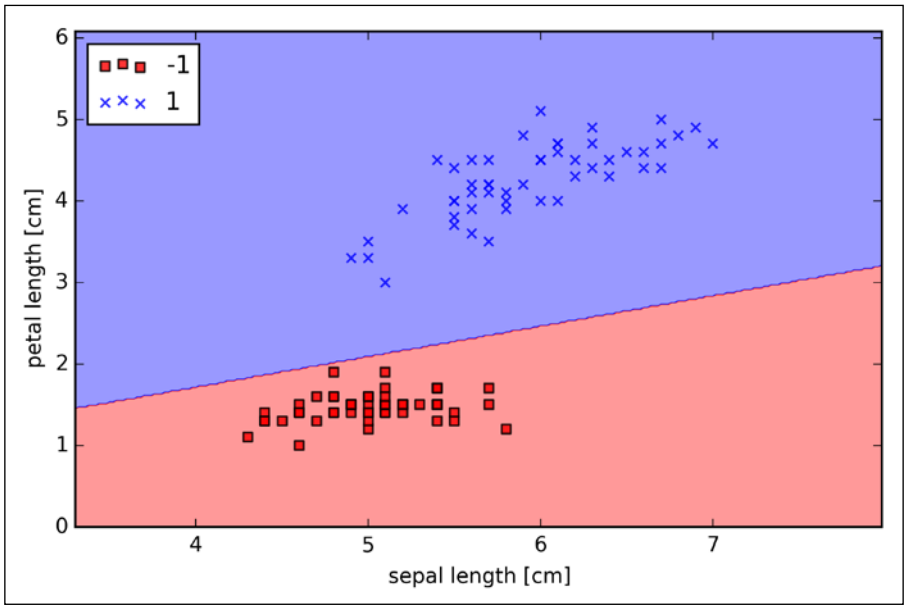
```
from matplotlib.colors import ListedColormap


def plot_decision_regions(X, y, classifier, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
```

```
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)
```

First, we define a number of `colors` and `markers` and create a color map from the list of colors via `ListedColormap`. Then, we determine the minimum and maximum values for the two features and use those feature vectors to create a pair of grid arrays `xx1` and `xx2` via the NumPy `meshgrid` function. Since we trained our perceptron classifier on two feature dimensions, we need to flatten the grid arrays and create a matrix that has the same number of columns as the Iris training subset so that we can use the `predict` method to predict the class labels `Z` of the corresponding grid points. After reshaping the predicted class labels `Z` into a grid with the same dimensions as `xx1` and `xx2`, we can now draw a contour plot via matplotlib's `contourf` function that maps the different decision regions to different colors for each predicted class in the grid array:

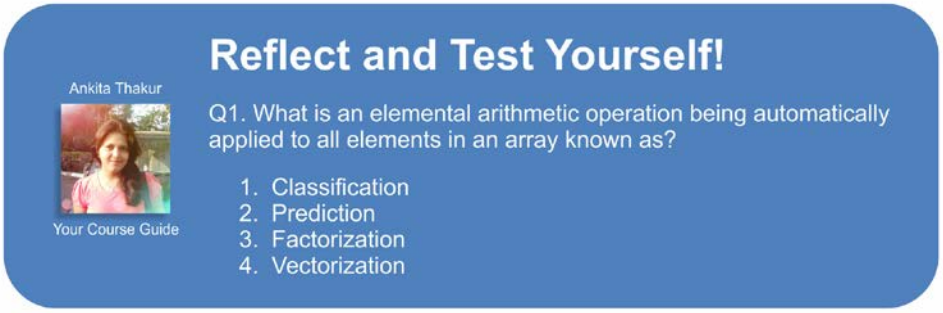
```
>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('sepal length [cm]')
>>> plt.ylabel('petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example, we should now see a plot of the decision regions, as shown in the following figure:



As we can see in the preceding plot, the perceptron learned a decision boundary that was able to classify all flower samples in the Iris training subset perfectly.

Although the perceptron classified the two Iris flower classes perfectly, convergence is one of the biggest problems of the perceptron. Frank Rosenblatt proved mathematically that the perceptron learning rule converges if the two classes can be separated by a linear hyperplane. However, if classes cannot be separated perfectly by such a linear decision boundary, the weights will never stop updating unless we set a maximum number of epochs.

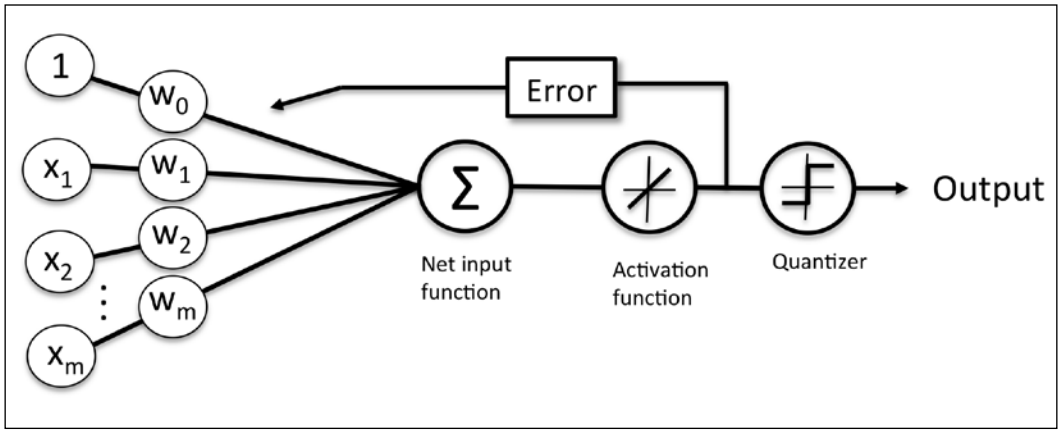


Adaptive linear neurons and the convergence of learning

In this section, we will take a look at another type of single-layer neural network: **ADAptive LInear NEuron** (**Adaline**). Adaline was published, only a few years after Frank Rosenblatt's perceptron algorithm, by Bernard Widrow and his doctoral student Tedd Hoff, and can be considered as an improvement on the latter (B. Widrow et al. Adaptive "*Adaline*" *neuron using chemical* "*memistors*". Number Technical Report 1553-2. Stanford Electron. Labs. Stanford, CA, October 1960). The Adaline algorithm is particularly interesting because it illustrates the key concept of defining and minimizing cost functions, which will lay the groundwork for understanding more advanced machine learning algorithms for classification, such as logistic regression and support vector machines, as well as regression models that we will discuss in future chapters.

The key difference between the Adaline rule (also known as the *Widrow-Hoff rule*) and Rosenblatt's perceptron is that the weights are updated based on a linear activation function rather than a unit step function like in the perceptron. In Adaline, this linear activation function $\phi(z)$ is simply the identity function of the net input so that $\phi(\boldsymbol{w}^T\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$.

While the linear activation function is used for learning the weights, a *quantizer*, which is similar to the unit step function that we have seen before, can then be used to predict the class labels, as illustrated in the following figure:



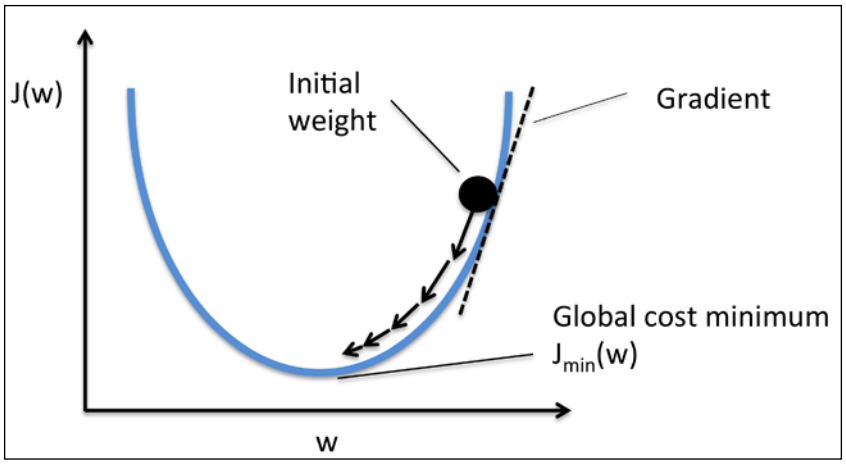
If we compare the preceding figure to the illustration of the perceptron algorithm that we saw earlier, the difference is that we know to use the continuous valued output from the linear activation function to compute the model error and update the weights, rather than the binary class labels.

Minimizing cost functions with gradient descent

One of the key ingredients of supervised machine learning algorithms is to define an *objective function* that is to be optimized during the learning process. This objective function is often a *cost function* that we want to minimize. In the case of Adaline, we can define the cost function J to learn the weights as the **Sum of Squared Errors** (**SSE**) between the calculated outcomes and the true class labels $J(\boldsymbol{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$.

The term $\frac{1}{2}$ is just added for our convenience; it will make it easier to derive the gradient, as we will see in the following paragraphs. The main advantage of this continuous linear activation function is—in contrast to the unit step function—that the cost function becomes differentiable. Another nice property of this cost function is that it is convex; thus, we can use a simple, yet powerful, optimization algorithm called *gradient descent* to find the weights that minimize our cost function to classify the samples in the Iris dataset.

As illustrated in the following figure, we can describe the principle behind gradient descent as *climbing down a hill* until a local or global cost minimum is reached. In each iteration, we take a step away from the gradient where the step size is determined by the value of the learning rate as well as the slope of the gradient:



Using gradient descent, we can now update the weights by taking a step away from the gradient $\nabla J(\boldsymbol{w})$ of our cost function $J(\boldsymbol{w})$:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}$$

Here, the weight change $\Delta \boldsymbol{w}$ is defined as the negative gradient multiplied by the learning rate η:

$$\Delta \boldsymbol{w} = -\eta \nabla J(\boldsymbol{w})$$

To compute the gradient of the cost function, we need to compute the partial derivative of the cost function with respect to each weight w_j, $\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)}$, so that we can write the update of weight w_j as $\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)}$.

Since we update all weights simultaneously, our Adaline learning rule becomes $\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}$.

For those who are familiar with calculus, the partial derivative of the SSE cost function with respect to the *j*th weight in can be obtained as follows:

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

$$= \frac{1}{2}\frac{\partial}{\partial w_j}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

$$= \frac{1}{2}\sum_i 2\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\frac{\partial}{\partial w_j}\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\frac{\partial}{\partial w_j}\left(y^{(i)} - \sum_i \left(w_j^{(i)} x_j^{(i)}\right)\right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\left(-x_j^{(i)}\right)$$

$$= -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)x_j^{(i)}$$

Although the Adaline learning rule looks identical to the perceptron rule, the $\phi\left(z^{(i)}\right)$ with $z^{(i)} = \boldsymbol{w}^T \boldsymbol{x}^{(i)}$ is a real number and not an integer class label. Furthermore, the weight update is calculated based on all samples in the training set (instead of updating the weights incrementally after each sample), which is why this approach is also referred to as "batch" gradient descent.

Implementing an Adaptive Linear Neuron in Python

Since the perceptron rule and Adaline are very similar, we will take the perceptron implementation that we defined earlier and change the `fit` method so that the weights are updated by minimizing the cost function via gradient descent:

```
class AdalineGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
```

```
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclassifications in every epoch.

    """
    def __init__(self, eta=0.01, n_iter=50):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors,
            where n_samples is the number of samples and
            n_features is the number of features.
        y : array-like, shape = [n_samples]
            Target values.

        Returns
        -------
        self : object

        """
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
```

```
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Compute linear activation"""
        return self.net_input(X)

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

Instead of updating the weights after evaluating each individual training sample, as in the perceptron, we calculate the gradient based on the whole training dataset via `self.eta * errors.sum()` for the zero-weight and via `self.eta * X.T.dot(errors)` for the weights 1 to m where `X.T.dot(errors)` is a *matrix-vector multiplication* between our feature matrix and the error vector. Similar to the previous perceptron implementation, we collect the cost values in a list `self.cost_` to check if the algorithm converged after training.

Performing a matrix-vector multiplication is similar to calculating a vector dot product where each row in the matrix is treated as a single row vector. This vectorized approach represents a more compact notation and results in a more efficient computation using NumPy. For example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 \times 7 + 2 \times 8 + 3 \times 9 \\ 4 \times 7 + 5 \times 8 + 6 \times 9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

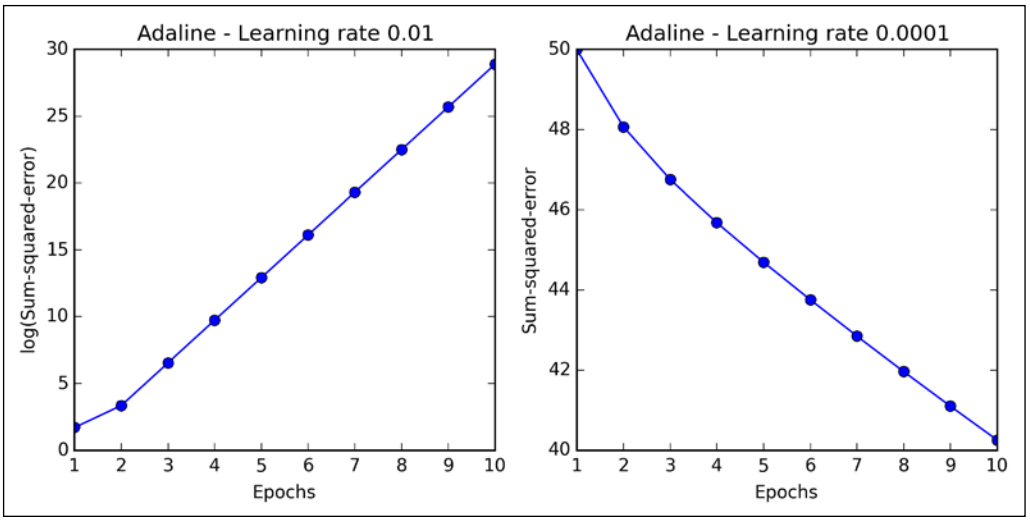
In practice, it often requires some experimentation to find a good learning rate η for optimal convergence. So, let's choose two different learning rates $\eta = 0.1$ and $\eta = 0.0001$ to start with and plot the cost functions versus the number of epochs to see how well the Adaline implementation learns from the training data.

The learning rate η, as well as the number of epochs `n_iter`, are the so-called *hyperparameters* of the perceptron and Adaline learning algorithms. In *Chapter 4, Building Good Training Sets – Data Preprocessing,* we will take a look at different techniques to automatically find the values of different hyperparameters that yield optimal performance of the classification model.

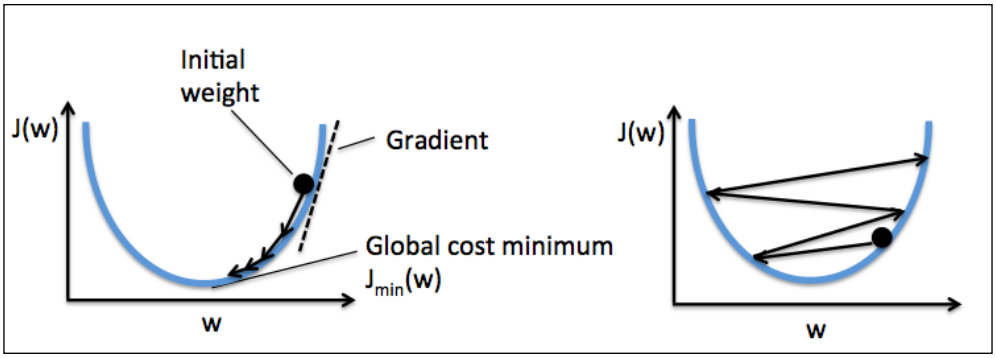
Let us now plot the cost against the number of epochs for the two different learning rates:

```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))
>>> ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
>>> ax[0].plot(range(1, len(ada1.cost_) + 1),
...            np.log10(ada1.cost_), marker='o')
>>> ax[0].set_xlabel('Epochs')
>>> ax[0].set_ylabel('log(Sum-squared-error)')
>>> ax[0].set_title('Adaline - Learning rate 0.01')
>>> ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
>>> ax[1].plot(range(1, len(ada2.cost_) + 1),
...            ada2.cost_, marker='o')
>>> ax[1].set_xlabel('Epochs')
>>> ax[1].set_ylabel('Sum-squared-error')
>>> ax[1].set_title('Adaline - Learning rate 0.0001')
>>> plt.show()
```

As we can see in the resulting cost function plots next, we encountered two different types of problems. The left chart shows what could happen if we choose a learning rate that is too large—instead of minimizing the cost function, the error becomes larger in every epoch because we *overshoot* the global minimum:



Although we can see that the cost decreases when we look at the right plot, the chosen learning rate $\eta = 0.0001$ is so small that the algorithm would require a very large number of epochs to converge. The following figure illustrates how we change the value of a particular weight parameter to minimize the cost function J (left subfigure). The subfigure on the right illustrates what happens if we choose a learning rate that is too large, we overshoot the global minimum:



Many machine learning algorithms that we will encounter throughout this book require some sort of feature scaling for optimal performance, which we will discuss in more detail in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*. Gradient descent is one of the many algorithms that benefit from feature scaling. Here, we will use a feature scaling method called *standardization*, which gives our data the property of a standard normal distribution. The mean of each feature is centered at value 0 and the feature column has a standard deviation of 1. For example, to standardize the j th feature, we simply need to subtract the sample mean μ_j from every training sample and divide it by its standard deviation σ_j:

$$\boldsymbol{x}'_j = \frac{\boldsymbol{x}_j - \mu_j}{\sigma_j}$$

Here $\boldsymbol{x}_j$ is a vector consisting of the j th feature values of all training samples n.

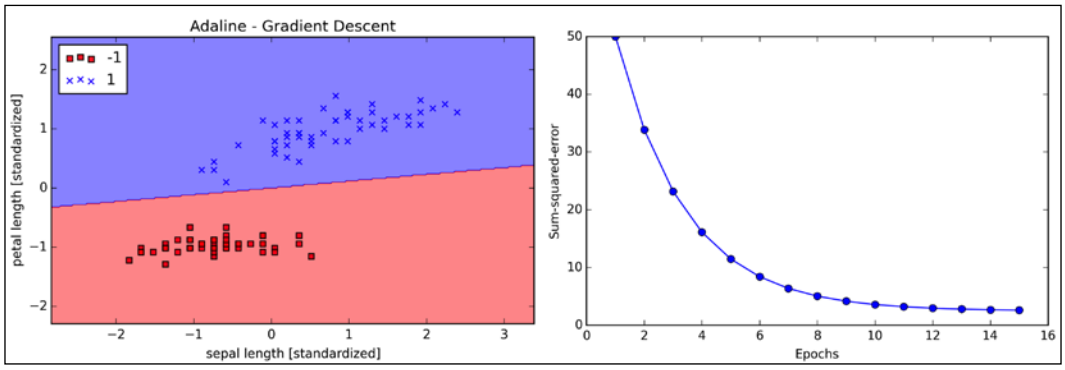
Standardization can easily be achieved using the NumPy methods `mean` and `std`:

```
>>> X_std = np.copy(X)
>>> X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
>>> X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

After standardization, we will train the Adaline again and see that it now converges using a learning rate $\eta = 0.01$:

```
>>> ada = AdalineGD(n_iter=15, eta=0.01)
>>> ada.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada)
>>> plt.title('Adaline - Gradient Descent')
>>> plt.xlabel('sepal length [standardized]')
>>> plt.ylabel('petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
>>> plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Sum-squared-error')
>>> plt.show()
```

After executing the preceding code, we should see a figure of the decision regions as well as a plot of the declining cost, as shown in the following figure:



As we can see in the preceding plots, the Adaline now converges after training on the standardized features using a learning rate $\eta = 0.01$. However, note that the SSE remains non-zero even though all samples were classified correctly.

Large scale machine learning and stochastic gradient descent

In the previous section, we learned how to minimize a cost function by taking a step into the opposite direction of a gradient that is calculated from the whole training set; this is why this approach is sometimes also referred to as *batch* gradient descent. Now imagine we have a very large dataset with millions of data points, which is not uncommon in many machine learning applications. Running batch gradient descent can be computationally quite costly in such scenarios since we need to reevaluate the whole training dataset each time we take one step towards the global minimum.

A popular alternative to the batch gradient descent algorithm is *stochastic gradient descent*, sometimes also called *iterative* or *on-line* gradient descent. Instead of updating the weights based on the sum of the accumulated errors over all samples $\boldsymbol{x}^{(i)}$:

$$\Delta \boldsymbol{w} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \boldsymbol{x}^{(i)},$$

We update the weights incrementally for each training sample:

$$\eta\left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \boldsymbol{x}^{(i)}$$

Although stochastic gradient descent can be considered as an approximation of gradient descent, it typically reaches convergence much faster because of the more frequent weight updates. Since each gradient is calculated based on a single training example, the error surface is noisier than in gradient descent, which can also have the advantage that stochastic gradient descent can escape shallow local minima more readily. To obtain accurate results via stochastic gradient descent, it is important to present it with data in a random order, which is why we want to shuffle the training set for every epoch to prevent cycles.

In stochastic gradient descent implementations, the fixed learning rate η is often replaced by an adaptive learning rate that decreases over time, for example, $\frac{c_1}{[number\ of\ iterations]+c_2}$ where c_1 and c_2 are constants. Note that stochastic gradient descent does not reach the global minimum but an area very close to it. By using an adaptive learning rate, we can achieve further annealing to a better global minimum

Another advantage of stochastic gradient descent is that we can use it for *online learning*. In online learning, our model is trained on-the-fly as new training data arrives. This is especially useful if we are accumulating large amounts of data—for example, customer data in typical web applications. Using online learning, the system can immediately adapt to changes and the training data can be discarded after updating the model if storage space in an issue.

A compromise between batch gradient descent and stochastic gradient descent is the so-called *mini-batch learning*. Mini-batch learning can be understood as applying batch gradient descent to smaller subsets of the training data—for example, 50 samples at a time. The advantage over batch gradient descent is that convergence is reached faster via mini-batches because of the more frequent weight updates. Furthermore, mini-batch learning allows us to replace the for-loop over the training samples in **Stochastic Gradient Descent** (**SGD**) by vectorized operations, which can further improve the computational efficiency of our learning algorithm.

Since we already implemented the Adaline learning rule using gradient descent, we only need to make a few adjustments to modify the learning algorithm to update the weights via stochastic gradient descent. Inside the `fit` method, we will now update the weights after each training sample. Furthermore, we will implement an additional `partial_fit` method, which does not reinitialize the weights, for on-line learning. In order to check if our algorithm converged after training, we will calculate the cost as the average cost of the training samples in each epoch. Furthermore, we will add an option to `shuffle` the training data before each epoch to avoid cycles when we are optimizing the cost function; via the `random_state` parameter, we allow the specification of a random seed for consistency:

```
from numpy.random import seed

class AdalineSGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclassifications in every epoch.
    shuffle : bool (default: True)
        Shuffles training data every epoch
        if True to prevent cycles.
    random_state : int (default: None)
        Set random state for shuffling
        and initializing the weights.

    """
    def __init__(self, eta=0.01, n_iter=10,
                 shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
```

```
        if random_state:
            seed(random_state)

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples
            is the number of samples and
            n_features is the number of features.
        y : array-like, shape = [n_samples]
            Target values.

        Returns
        -------
        self : object

         """
        self._initialize_weights(X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            cost = []
            for xi, target in zip(X, y):
                cost.append(self._update_weights(xi, target))
            avg_cost = sum(cost)/len(y)
            self.cost_.append(avg_cost)
        return self

    def partial_fit(self, X, y):
        """Fit training data without reinitializing the weights"""
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
```

```
            """Shuffle training data"""
            r = np.random.permutation(len(y))
            return X[r], y[r]

        def _initialize_weights(self, m):
            """Initialize weights to zeros"""
            self.w_ = np.zeros(1 + m)
            self.w_initialized = True

        def _update_weights(self, xi, target):
            """Apply Adaline learning rule to update the weights"""
            output = self.net_input(xi)
            error = (target - output)
            self.w_[1:] += self.eta * xi.dot(error)
            self.w_[0] += self.eta * error
            cost = 0.5 * error**2
            return cost

        def net_input(self, X):
            """Calculate net input"""
            return np.dot(X, self.w_[1:]) + self.w_[0]

        def activation(self, X):
            """Compute linear activation"""
            return self.net_input(X)

        def predict(self, X):
            """Return class label after unit step"""
            return np.where(self.activation(X) >= 0.0, 1, -1)
```

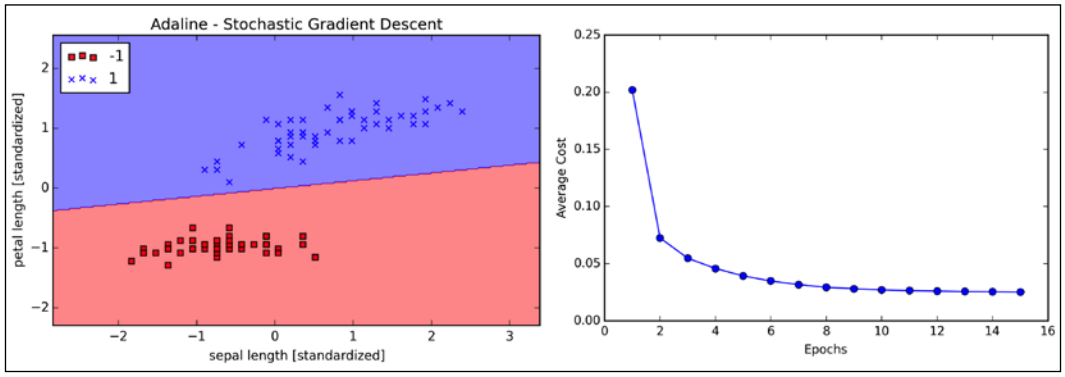
The `_shuffle` method that we are now using in the `AdalineSGD` classifier works as follows: via the `permutation` function in `numpy.random`, we generate a random sequence of unique numbers in the range 0 to 100. Those numbers can then be used as indices to shuffle our feature matrix and class label vector.

We can then use the fit method to train the `AdalineSGD` classifier and use our `plot_decision_regions` to plot our training results:

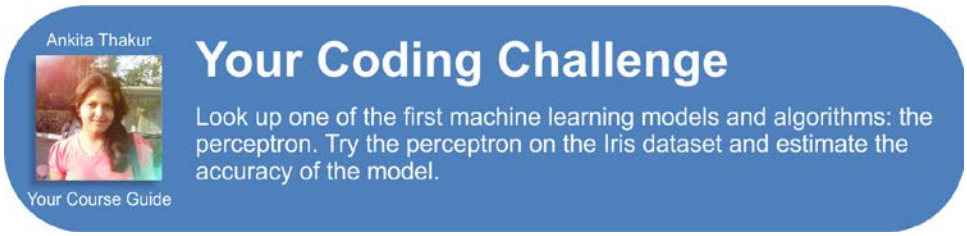
```
>>> ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada)
>>> plt.title('Adaline - Stochastic Gradient Descent')
>>> plt.xlabel('sepal length [standardized]')
>>> plt.ylabel('petal length [standardized]')
```

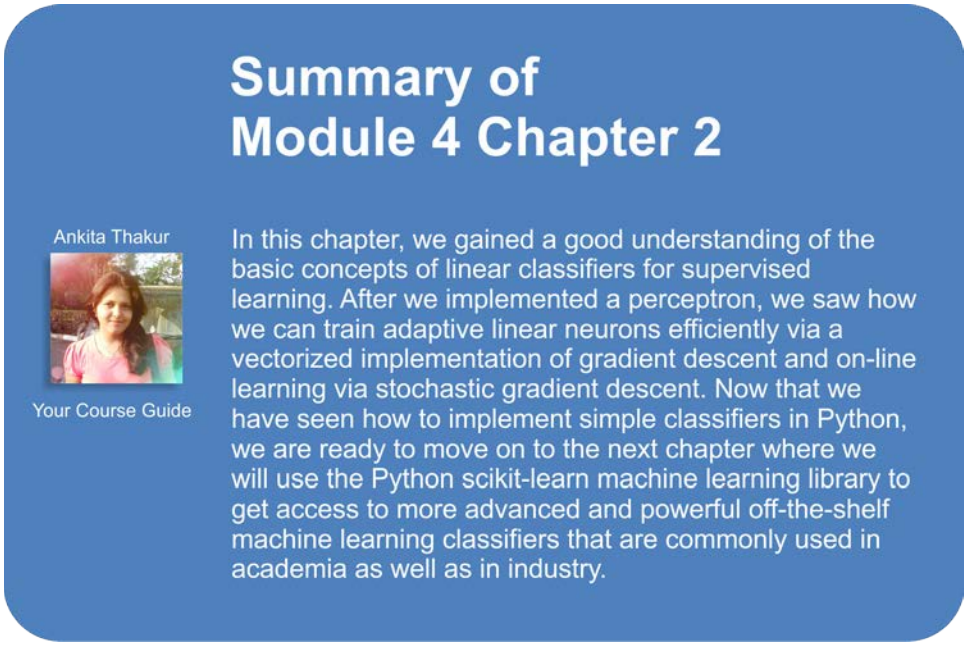
```
>>> plt.legend(loc='upper left')
>>> plt.show()
>>> plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Average Cost')
>>> plt.show()
```

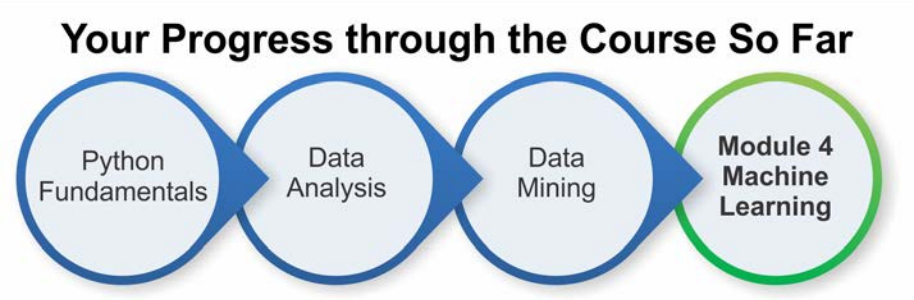
The two plots that we obtain from executing the preceding code example are shown in the following figure:



As we can see, the average cost goes down pretty quickly, and the final decision boundary after 15 epochs looks similar to the batch gradient descent with Adaline. If we want to update our model—for example, in an on-line learning scenario with streaming data—we could simply call the `partial_fit` method on individual samples—for instance, `ada.partial_fit(X_std[0, :], y[0])`.




Summary of
Module 4 Chapter 2
Ankita Thakur
Your Course Guide
In this chapter, we gained a good understanding of the basic concepts of linear classifiers for supervised learning. After we implemented a perceptron, we saw how we can train adaptive linear neurons efficiently via a vectorized implementation of gradient descent and on-line learning via stochastic gradient descent. Now that we have seen how to implement simple classifiers in Python, we are ready to move on to the next chapter where we will use the Python scikit-learn machine learning library to get access to more advanced and powerful off-the-shelf machine learning classifiers that are commonly used in academia as well as in industry.


Your Progress through the Course So Far
Python Fundamentals
Data Analysis
Data Mining
Module 4 Machine Learning

3
A Tour of Machine Learning Classifiers Using scikit-learn

In this chapter, we will take a tour through a selection of popular and powerful machine learning algorithms that are commonly used in academia as well as in the industry. While learning about the differences between several supervised learning algorithms for classification, we will also develop an intuitive appreciation of their individual strengths and weaknesses. Also, we will take our first steps with the scikit-learn library, which offers a user-friendly interface for using those algorithms efficiently and productively.

The topics that we will learn about throughout this chapter are as follows:

- Introduction to the concepts of popular classification algorithms
- Using the scikit-learn machine learning library
- Questions to ask when selecting a machine learning algorithm

Choosing a classification algorithm

Choosing an appropriate classification algorithm for a particular problem task requires practice: each algorithm has its own quirks and is based on certain assumptions. To restate the "No Free Lunch" theorem: no single classifier works best across all possible scenarios. In practice, it is always recommended that you compare the performance of at least a handful of different learning algorithms to select the best model for the particular problem; these may differ in the number of features or samples, the amount of noise in a dataset, and whether the classes are linearly separable or not.

Eventually, the performance of a classifier, computational power as well as predictive power, depends heavily on the underlying data that are available for learning. The five main steps that are involved in training a machine learning algorithm can be summarized as follows:

1. Selection of features.
2. Choosing a performance metric.
3. Choosing a classifier and optimization algorithm.
4. Evaluating the performance of the model.
5. Tuning the algorithm.

Since the approach of this book is to build machine learning knowledge step by step, we will mainly focus on the principal concepts of the different algorithms in this chapter and revisit topics such as feature selection and preprocessing, performance metrics, and hyperparameter tuning for more detailed discussions later in this book.

First steps with scikit-learn

In *Chapter 2*, *Training Machine Learning Algorithms for Classification*, you learned about two related learning algorithms for classification: the **perceptron** rule and **Adaline**, which we implemented in Python by ourselves. Now we will take a look at the scikit-learn API, which combines a user-friendly interface with a highly optimized implementation of several classification algorithms. However, the scikit-learn library offers not only a large variety of learning algorithms, but also many convenient functions to preprocess data and to fine-tune and evaluate our models. We will discuss this in more detail together with the underlying concepts in *Chapter 4*, *Building Good Training Sets – Data Preprocessing*, and *Chapter 5*, *Compressing Data via Dimensionality Reduction*.

Training a perceptron via scikit-learn

To get started with the scikit-learn library, we will train a perceptron model similar to the one that we implemented in *Chapter 2*, *Training Machine Learning Algorithms for Classification*. For simplicity, we will use the already familiar **Iris** dataset throughout the following sections. Conveniently, the Iris dataset is already available via scikit-learn, since it is a simple yet popular dataset that is frequently used for testing and experimenting with algorithms. Also, we will only use two features from the **Iris flower** dataset for visualization purposes.

We will assign the *petal length* and *petal width* of the 150 flower samples to the feature matrix `X` and the corresponding class labels of the flower species to the vector `y`:

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
```

If we executed `np.unique(y)` to return the different class labels stored in `iris.target`, we would see that the Iris flower class names, *Iris-Setosa*, *Iris-Versicolor*, and *Iris-Virginica*, are already stored as integers (`0`, `1`, `2`), which is recommended for the optimal performance of many machine learning libraries.

To evaluate how well a trained model performs on unseen data, we will further split the dataset into separate training and test datasets. Later in *Chapter 5*, *Compressing Data via Dimensionality Reduction*, we will discuss the best practices around model evaluation in more detail:

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...           X, y, test_size=0.3, random_state=0)
```

Using the `train_test_split` function from scikit-learn's `cross_validation` module, we randomly split the `X` and `y` arrays into 30 percent test data (45 samples) and 70 percent training data (105 samples).

Many machine learning and optimization algorithms also require feature scaling for optimal performance, as we remember from the **gradient descent** example in *Chapter 2*, *Training Machine Learning Algorithms for Classification*. Here, we will standardize the features using the `StandardScaler` class from scikit-learn's `preprocessing` module:

```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Using the preceding code, we loaded the `StandardScaler` class from the preprocessing module and initialized a new `StandardScaler` object that we assigned to the variable `sc`. Using the `fit` method, `StandardScaler` estimated the parameters μ (sample mean) and σ (standard deviation) for each feature dimension from the training data. By calling the `transform` method, we then standardized the training data using those estimated parameters μ and σ. Note that we used the same scaling parameters to standardize the test set so that both the values in the training and test dataset are comparable to each other.

Having standardized the training data, we can now train a perceptron model. Most algorithms in scikit-learn already support multiclass classification by default via the **One-vs.-Rest** (**OvR**) method, which allows us to feed the three flower classes to the perceptron all at once. The code is as follows:

```
>>> from sklearn.linear_model import Perceptron
>>> ppn = Perceptron(n_iter=40, eta0=0.1, random_state=0)
>>> ppn.fit(X_train_std, y_train)
```

The scikit-learn interface reminds us of our perceptron implementation in *Chapter 2*, *Training Machine Learning Algorithms for Classification*: after loading the `Perceptron` class from the `linear_model` module, we initialized a new `Perceptron` object and trained the model via the `fit` method. Here, the model parameter `eta0` is equivalent to the learning rate `eta` that we used in our own perceptron implementation, and the parameter `n_iter` defines the number of epochs (passes over the training set). As we remember from *Chapter 2*, *Training Machine Learning Algorithms for Classification*, finding an appropriate learning rate requires some experimentation. If the learning rate is too large, the algorithm will overshoot the global cost minimum. If the learning rate is too small, the algorithm requires more epochs until convergence, which can make the learning slow—especially for large datasets. Also, we used the `random_state` parameter for reproducibility of the initial shuffling of the training dataset after each epoch.

Having trained a model in scikit-learn, we can make predictions via the `predict` method, just like in our own perceptron implementation in *Chapter 2*, *Training Machine Learning Algorithms for Classification*. The code is as follows:

```
>>> y_pred = ppn.predict(X_test_std)
>>> print('Misclassified samples: %d' % (y_test != y_pred).sum())
Misclassified samples: 4
```

On executing the preceding code, we see that the perceptron misclassifies 4 out of the 45 flower samples. Thus, the misclassification error on the test dataset is 0.089 or 8.9 percent $(4 / 45 \approx 0.089)$.

Instead of the misclassification **error**, many machine learning practitioners report the classification **accuracy** of a model, which is simply calculated as follows:

1 - *misclassification error* = 0.911 or 91.1 percent.

Scikit-learn also implements a large variety of different performance metrics that are available via the `metrics` module. For example, we can calculate the classification accuracy of the perceptron on the test set as follows:

```
>>> from sklearn.metrics import accuracy_score
>>> print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
0.91
```

Here, `y_test` are the true class labels and `y_pred` are the class labels that we predicted previously.

Note that we evaluate the performance of our models based on the test set in this chapter. In *Chapter 5*, *Compressing Data via Dimensionality Reduction*, you will learn about useful techniques, including graphical analysis such as learning curves, to detect and prevent **overfitting**. Overfitting means that the model captures the patterns in the training data well, but fails to generalize well to unseen data.

Finally, we can use our `plot_decision_regions` function from *Chapter 2*, *Training Machine Learning Algorithms for Classification*, to plot the **decision regions** of our newly trained perceptron model and visualize how well it separates the different flower samples. However, let's add a small modification to highlight the samples from the test dataset via small circles:

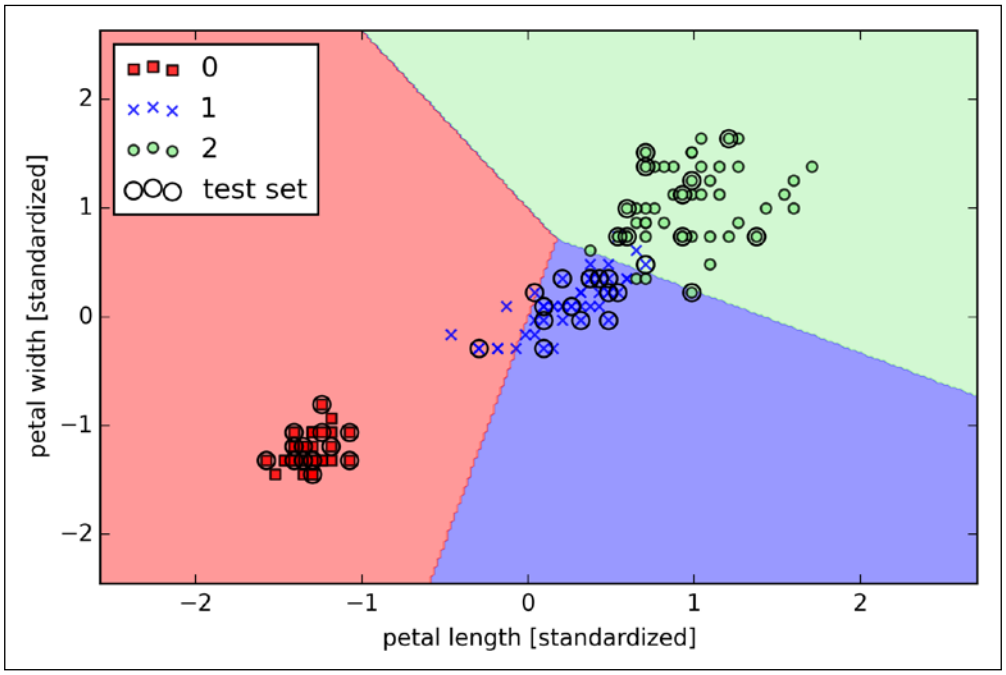
```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt

def plot_decision_regions(X, y, classifier,
                          test_idx=None, resolution=0.02):

    # setup marker generator and color map
```

```
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot all samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)

    # highlight test samples
    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1], c='',
                alpha=1.0, linewidths=1, marker='o',
                s=55, label='test set')
```

With the slight modification that we made to the `plot_decision_regions` function (highlighted in the preceding code), we can now specify the indices of the samples that we want to mark on the resulting plots. The code is as follows:

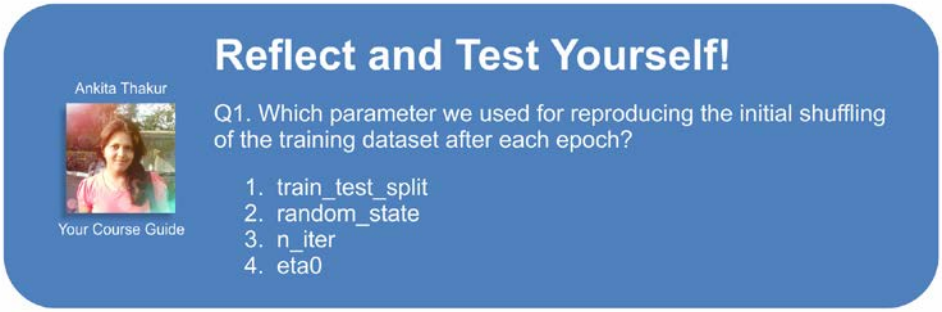
```
>>> X_combined_std = np.vstack((X_train_std, X_test_std))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X=X_combined_std,
...                       y=y_combined,
...                       classifier=ppn,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

As we can see in the resulting plot, the three flower classes cannot be perfectly separated by a linear decision boundaries:



We remember from our discussion in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, that the perceptron algorithm never converges on datasets that aren't perfectly linearly separable, which is why the use of the perceptron algorithm is typically not recommended in practice. In the following sections, we will look at more powerful linear classifiers that converge to a cost minimum even if the classes are not perfectly linearly separable.

The `Perceptron` as well as other scikit-learn functions and classes have additional parameters that we omit for clarity. You can read more about those parameters using the `help` function in Python (for example, `help(Perceptron)`) or by going through the excellent scikit-learn online documentation at `http://scikit-learn.org/stable/`.



Modeling class probabilities via logistic regression

Although the perceptron rule offers a nice and easygoing introduction to machine learning algorithms for classification, its biggest disadvantage is that it never converges if the classes are not perfectly linearly separable. The classification task in the previous section would be an example of such a scenario. Intuitively, we can think of the reason as the weights are continuously being updated since there is always at least one misclassified sample present in each epoch. Of course, you can change the learning rate and increase the number of epochs, but be warned that the perceptron will never converge on this dataset. To make better use of our time, we will now take a look at another simple yet more powerful algorithm for linear and binary classification problems: **logistic regression**. Note that, in spite of its name, logistic regression is a model for classification, not regression.

Logistic regression intuition and conditional probabilities

Logistic regression is a classification model that is very easy to implement but performs very well on linearly separable classes. It is one of the most widely used algorithms for classification in industry. Similar to the perceptron and Adaline, the logistic regression model in this chapter is also a linear model for binary classification that can be extended to multiclass classification via the OvR technique.

To explain the idea behind logistic regression as a probabilistic model, let's first introduce the **odds ratio**, which is the odds in favor of a particular event. The odds ratio can be written as $\frac{p}{(1-p)}$, where p stands for the probability of the positive event. The term *positive event* does not necessarily mean *good*, but refers to the event that we want to predict, for example, the probability that a patient has a certain disease; we can think of the positive event as class label $y = 1$. We can then further define the **logit** function, which is simply the logarithm of the odds ratio (log-odds):

$$logit\left(p\right) = log\frac{p}{\left(1-p\right)}$$

The logit function takes input values in the range 0 to 1 and transforms them to values over the entire real number range, which we can use to express a linear relationship between feature values and the log-odds:

$$logit\left(p\left(y=1 \mid \boldsymbol{x}\right)\right) = w_{\mathbf{0}}x_{\mathbf{0}} + w_{\mathbf{1}}x_{\mathbf{1}} + \cdots + w_m x_m = \sum_{i=0}^{m} w_i x_i = \boldsymbol{w}^T \boldsymbol{x}$$

Here, $p\left(y=1 \mid \boldsymbol{x}\right)$ is the conditional probability that a particular sample belongs to class 1 given its features *x*.

Now what we are actually interested in is predicting the probability that a certain sample belongs to a particular class, which is the inverse form of the logit function. It is also called the *logistic* function, sometimes simply abbreviated as *sigmoid* function due to its characteristic S-shape.

$$\phi\left(z\right) = \frac{1}{1+e^{-z}}$$

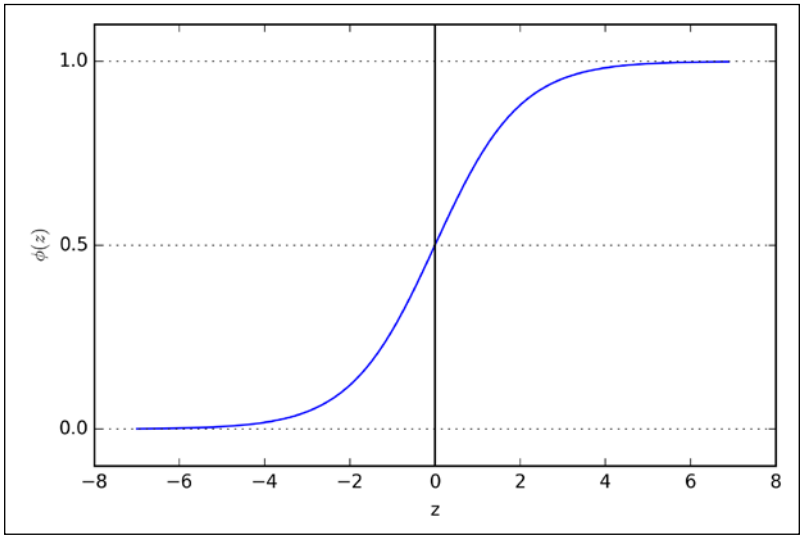
Here, *z* is the net input, that is, the linear combination of weights and sample features and can be calculated as $z = \boldsymbol{w}^T \boldsymbol{x} = w_0 + w_{\mathbf{1}}x_{\mathbf{1}} + \cdots + w_{\boldsymbol{m}}x_{\boldsymbol{m}}$.

Now let's simply plot the sigmoid function for some values in the range -7 to 7 to see what it looks like:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def sigmoid(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> z = np.arange(-7, 7, 0.1)
```

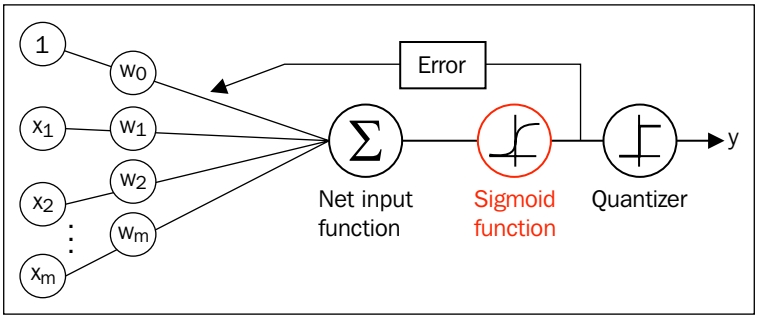
```
>>> phi_z = sigmoid(z)
>>> plt.plot(z, phi_z)
>>> plt.axvline(0.0, color='k')
>>> plt.axhspan(0.0, 1.0, facecolor='1.0', alpha=1.0, ls='dotted')
>>> plt.axhline(y=0.5, ls='dotted', color='k')
>>> plt.yticks([0.0, 0.5, 1.0])
>>> plt.ylim(-0.1, 1.1)
>>> plt.xlabel('z')
>>> plt.ylabel('$\phi (z)$')
>>> plt.show()
```

As a result of executing the previous code example, we should now see the **S-shaped** (sigmoidal) curve:



We can see that $\phi(z)$ approaches 1 if z goes towards infinity ($z \to \infty$), since e^{-z} becomes very small for large values of z. Similarly, $\phi(z)$ goes towards 0 for $z \to -\infty$ as the result of an increasingly large denominator. Thus, we conclude that this sigmoid function takes real number values as input and transforms them to values in the range [0, 1] with an intercept at $\phi(z) = 0.5$.

To build some intuition for the logistic regression model, we can relate it to our previous Adaline implementation in *Chapter 2*, *Training Machine Learning Algorithms for Classification*. In Adaline, we used the identity function $\phi(z) = z$ as the activation function. In logistic regression, this activation function simply becomes the sigmoid function that we defined earlier, which is illustrated in the following figure:



The output of the sigmoid function is then interpreted as the probability of particular sample belonging to class 1 $\phi(z) = P(y = 1 | \boldsymbol{x}; \boldsymbol{w})$, given its features x parameterized by the weights w. For example, if we compute $\phi(z) = 0.8$ for a particular flower sample, it means that the chance that this sample is an Iris-Versicolor flower is 80 percent. Similarly, the probability that this flower is an Iris-Setosa flower can be calculated as $P(y = 0 | \boldsymbol{x}; \boldsymbol{w}) = 1 - P(y = 1 | \boldsymbol{x}; \boldsymbol{w}) = 0.2$ or 20 percent. The predicted probability can then simply be converted into a binary outcome via a quantizer (unit step function):

$$\hat{y} = \begin{cases} 1 & \text{if } \phi(z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

If we look at the preceding sigmoid plot, this is equivalent to the following:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0.0 \\ 0 & \text{otherwise} \end{cases}$$

In fact, there are many applications where we are not only interested in the predicted class labels, but where estimating the class-membership probability is particularly useful. Logistic regression is used in weather forecasting, for example, to not only predict if it will rain on a particular day but also to report the chance of rain. Similarly, logistic regression can be used to predict the chance that a patient has a particular disease given certain symptoms, which is why logistic regression enjoys wide popularity in the field of medicine.

Learning the weights of the logistic cost function

You learned how we could use the logistic regression model to predict probabilities and class labels. Now let's briefly talk about the parameters of the model, for example, weights *w*. In the previous chapter, we defined the sum-squared-error cost function:

$$J(\boldsymbol{w}) = \sum_i \frac{1}{2}\left(\phi\left(z^{(i)}\right) - y^{(i)}\right)^2$$

We minimized this in order to learn the weights *w* for our Adaline classification model. To explain how we can derive the cost function for logistic regression, let's first define the likelihood *L* that we want to maximize when we build a logistic regression model, assuming that the individual samples in our dataset are independent of one another. The formula is as follows:

$$L(\boldsymbol{w}) = P(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{w}) = \prod_{i=1}^{n} P\left(y^{(i)} \mid x^{(i)}; \boldsymbol{w}\right) = \prod_{i=1}^{n} \left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

In practice, it is easier to maximize the (natural) log of this equation, which is called the log-likelihood function:

$$l(\boldsymbol{w}) = \log L(\boldsymbol{w}) = \sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)\right]$$

Firstly, applying the log function reduces the potential for numerical underflow, which can occur if the likelihoods are very small. Secondly, we can convert the product of factors into a summation of factors, which makes it easier to obtain the derivative of this function via the addition trick, as you may remember from calculus.

Now we could use an optimization algorithm such as gradient ascent to maximize this log-likelihood function. Alternatively, let's rewrite the log-likelihood as a cost function J that can be minimized using gradient descent as in *Chapter 2*, *Training Machine Learning Algorithms for Classification*:

$$J(\boldsymbol{w}) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)\right]$$

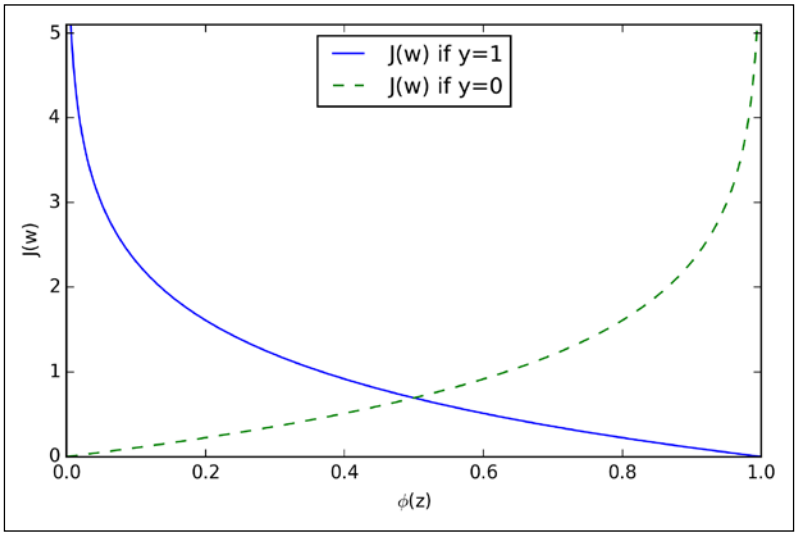
To get a better grasp on this cost function, let's take a look at the cost that we calculate for one single-sample instance:

$$J\left(\phi\left(z\right), y; \boldsymbol{w}\right) = -y \log\left(\phi\left(z\right)\right) - \left(1-y\right) \log\left(1-\phi\left(z\right)\right)$$

Looking at the preceding equation, we can see that the first term becomes zero if $y=0$, and the second term becomes zero if $y=1$, respectively:

$$J\left(\phi\left(z\right), y; \boldsymbol{w}\right) = \begin{cases} -\log\left(\phi\left(z\right)\right) & \textit{if } y=1 \\ -\log\left(1-\phi\left(z\right)\right) & \textit{if } y=0 \end{cases}$$

The following plot illustrates the cost for the classification of a single-sample instance for different values of $\phi(z)$:



We can see that the cost approaches 0 (plain blue line) if we correctly predict that a sample belongs to class 1. Similarly, we can see on the *y* axis that the cost also approaches 0 if we correctly predict $y=0$ (dashed line). However, if the prediction is wrong, the cost goes towards infinity. The moral is that we penalize wrong predictions with an increasingly larger cost.

Training a logistic regression model with scikit-learn

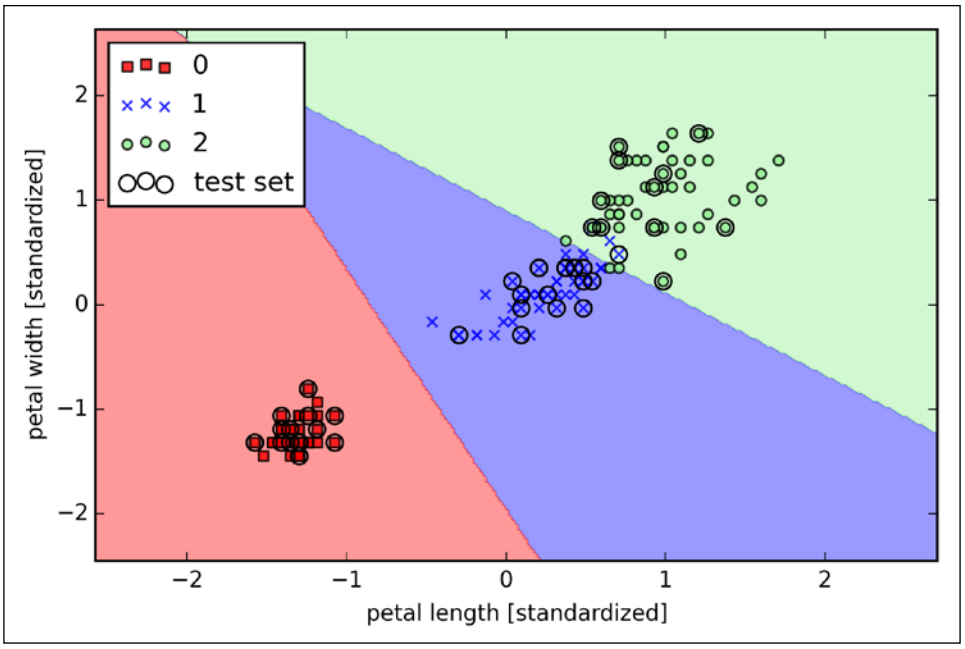
If we were to implement logistic regression ourselves, we could simply substitute the cost function J in our Adaline implementation from *Chapter 2*, *Training Machine Learning Algorithms for Classification*, by the new cost function:

$$J(\boldsymbol{w}) = -\sum_i y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)$$

This would compute the cost of classifying all training samples per epoch and we would end up with a working logistic regression model. However, since scikit-learn implements a highly optimized version of logistic regression that also supports multiclass settings off-the-shelf, we will skip the implementation and use the `sklearn.linear_model.LogisticRegression` class as well as the familiar `fit` method to train the model on the standardized flower training dataset:

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=1000.0, random_state=0)
>>> lr.fit(X_train_std, y_train)
   >>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=lr,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After fitting the model on the training data, we plotted the decision regions, training samples and test samples, as shown here:



Looking at the preceding code that we used to train the `LogisticRegression` model, you might now be wondering, "What is this mysterious parameter `C`?" We will get to this in a second, but let's briefly go over the concept of overfitting and regularization in the next subsection first.

Furthermore, we can predict the class-membership probability of the samples via the `predict_proba` method. For example, we can predict the probabilities of the first Iris-Setosa sample:

```
>>> lr.predict_proba(X_test_std[0,:])
```

This returns the following array:

```
array([[  0.000,    0.063,    0.937]])
```

The preceding array tells us that the model predicts a chance of 93.7 percent that the sample belongs to the Iris-Virginica class, and a 6.3 percent chance that the sample is a Iris-Versicolor flower.

We can show that the weight update in logistic regression via gradient descent is indeed equal to the equation that we used in Adaline in *Chapter 2*, *Training Machine Learning Algorithms for Classification*. Let's start by calculating the partial derivative of the log-likelihood function with respect to the *j*th weight:

$$\frac{\partial}{\partial w_j} l(\boldsymbol{w}) = \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z)$$

Before we continue, let's calculate the partial derivative of the sigmoid function first:

$$\begin{aligned} \frac{\partial}{\partial z} \phi(z) &= \frac{\partial}{\partial z} \frac{1}{1+e^{-z}} = \frac{1}{\left(1+e^{-z}\right)^2} e^{-z} = \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}} \right) \\ &= \phi(z)\left(1-\phi(z)\right) \end{aligned}$$

Now we can resubstitute $\frac{\partial}{\partial z}\phi(z) = \phi(z)(1-\phi(z))$ in our first equation to obtain the following:

$$\begin{aligned} & \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z) \\ &= \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \phi(z)\left(1-\phi(z)\right) \frac{\partial}{\partial w_j} z \\ &= \left(y\left(1-\phi(z)\right) - (1-y)\phi(z) \right) x_j \\ &= \left(y - \phi(z) \right) x_j \end{aligned}$$

Remember that the goal is to find the weights that maximize the log-likelihood so that we would perform the update for each weight as follows:

$$w_j := w_j + \eta \sum_{i=1}^{n} \left(y^{(i)} - \phi\left(z^{(i)} \right) \right) x_j^{(i)}$$

Since we update all weights simultaneously, we can write the general update rule as follows:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}$$

We define $\Delta \boldsymbol{w}$ as follows:

$$\Delta \boldsymbol{w} = \eta \nabla l\left(\boldsymbol{w} \right)$$

Since maximizing the log-likelihood is equal to minimizing the cost function J that we defined earlier, we can write the gradient descent update rule as follows:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_{i=1}^{n} \left(y^{(i)} - \phi\left(z^{(i)} \right) \right) x_j^{(i)}$$

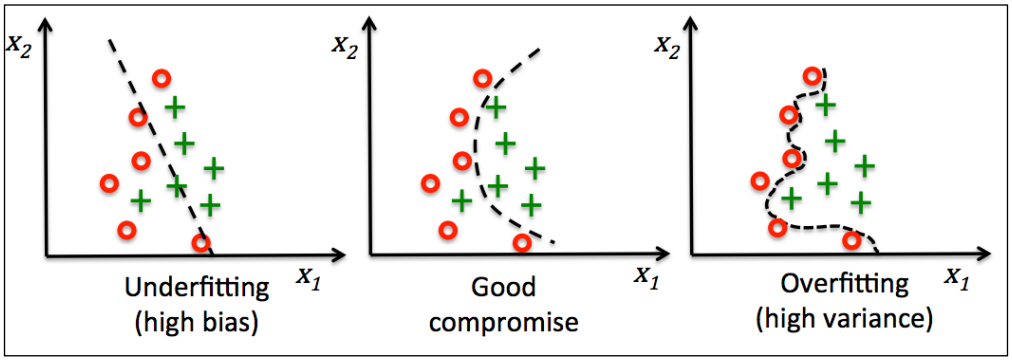
$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w},\ \Delta \boldsymbol{w} = -\eta \nabla J\left(\boldsymbol{w} \right)$$

This is equal to the gradient descent rule in Adaline in *Chapter 2*, *Training Machine Learning Algorithms for Classification*.

Tackling overfitting via regularization

Overfitting is a common problem in machine learning, where a model performs well on training data but does not generalize well to unseen data (test data). If a model suffers from overfitting, we also say that the model has a high variance, which can be caused by having too many parameters that lead to a model that is too complex given the underlying data. Similarly, our model can also suffer from **underfitting** (high bias), which means that our model is not complex enough to capture the pattern in the training data well and therefore also suffers from low performance on unseen data.

Although we have only encountered linear models for classification so far, the problem of overfitting and underfitting can be best illustrated by using a more complex, nonlinear decision boundary as shown in the following figure:



Variance measures the consistency (or variability) of the model prediction for a particular sample instance if we would retrain the model multiple times, for example, on different subsets of the training dataset. We can say that the model is sensitive to the randomness in the training data. In contrast, bias measures how far off the predictions are from the correct values in general if we rebuild the model multiple times on different training datasets; bias is the measure of the systematic error that is not due to randomness.

One way of finding a good bias-variance tradeoff is to tune the complexity of the model via regularization. Regularization is a very useful method to handle collinearity (high correlation among features), filter out noise from data, and eventually prevent overfitting. The concept behind regularization is to introduce additional information (bias) to penalize extreme parameter weights. The most common form of regularization is the so-called **L2 regularization** (sometimes also called L2 shrinkage or weight decay), which can be written as follows:

$$\frac{\lambda}{2}\|\boldsymbol{w}\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$$

Here, λ is the so-called regularization parameter.

Regularization is another reason why feature scaling such as standardization is important. For regularization to work properly, we need to ensure that all our features are on comparable scales.

In order to apply regularization, we just need to add the regularization term to the cost function that we defined for logistic regression to shrink the weights:

$$J(\boldsymbol{w}) = \sum_{i=1}^{n}\left[-y^{(i)}\log\left(\phi\left(z^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-\phi\left(z^{(i)}\right)\right)\right] + \frac{\lambda}{2}\|w\|^2$$

Via the regularization parameter λ, we can then control how well we fit the training data while keeping the weights small. By increasing the value of λ, we increase the regularization strength.

The parameter `C` that is implemented for the `LogisticRegression` class in scikit-learn comes from a convention in support vector machines, which will be the topic of the next section. `C` is directly related to the regularization parameter λ, which is its inverse:

$$C = \frac{1}{\lambda}$$

So we can rewrite the regularized cost function of logistic regression as follows:

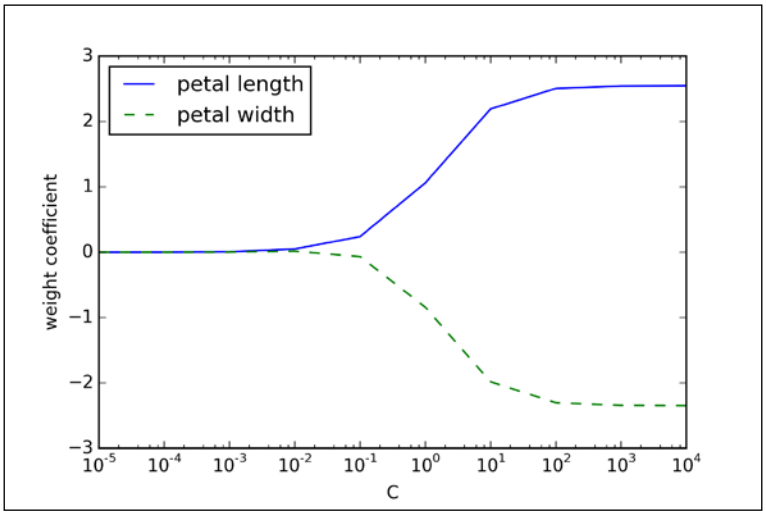
$$J(\boldsymbol{w}) = C\left[\sum_{i=1}^{n}\left(-y^{(i)}\log\left(\phi\left(z^{(i)}\right)\right) - \left(1-y^{(i)}\right)\right)\log\left(1-\phi\left(z^{(i)}\right)\right)\right] + \frac{1}{2}\|\boldsymbol{w}\|^2$$

Consequently, decreasing the value of the inverse regularization parameter `C` means that we are increasing the regularization strength, which we can visualize by plotting the L2 regularization path for the two weight coefficients:

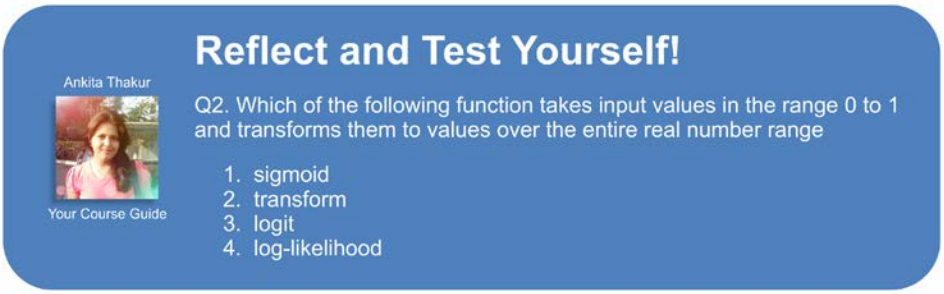
```
>>> weights, params = [], []
>>> for c in np.arange(-5, 5):
...     lr = LogisticRegression(C=10**c, random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
>>> weights = np.array(weights)
>>> plt.plot(params, weights[:, 0],
...          label='petal length')
>>> plt.plot(params, weights[:, 1], linestyle='--',
...          label='petal width')
>>> plt.ylabel('weight coefficient')
>>> plt.xlabel('C')
>>> plt.legend(loc='upper left')
>>> plt.xscale('log')
>>> plt.show()
```

By executing the preceding code, we fitted ten logistic regression models with different values for the inverse-regularization parameter `C`. For the purposes of illustration, we only collected the weight coefficients of the class 2 vs. all classifier. Remember that we are using the OvR technique for multiclass classification.

As we can see in the resulting plot, the weight coefficients shrink if we decrease the parameter **C**, that is, if we increase the regularization strength:

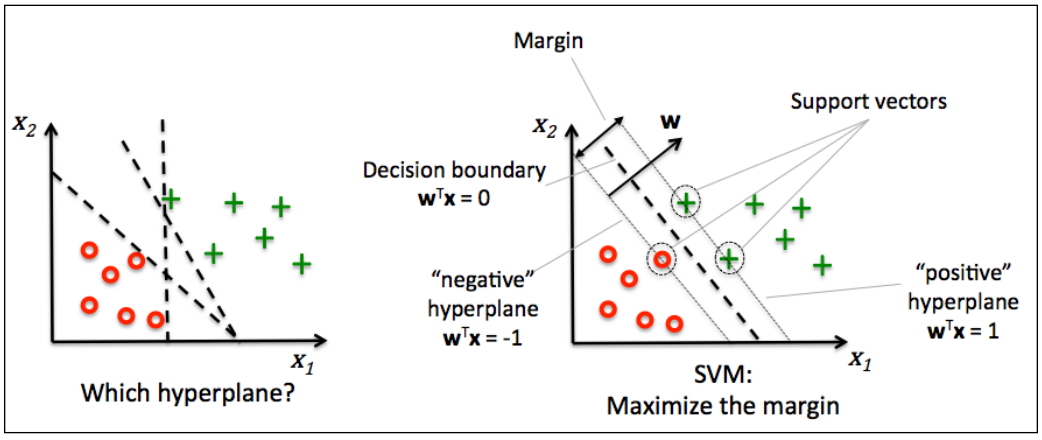


Since an in-depth coverage of the individual classification algorithms exceeds the scope of this book, I warmly recommend Dr. Scott Menard's *Logistic Regression: From Introductory to Advanced Concepts and Applications*, *Sage Publications*, to readers who want to learn more about logistic regression.



Maximum margin classification with support vector machines

Another powerful and widely used learning algorithm is the **support vector machine** (**SVM**), which can be considered as an extension of the perceptron. Using the perceptron algorithm, we minimized misclassification errors. However, in SVMs, our optimization objective is to maximize the **margin**. The margin is defined as the distance between the separating hyperplane (decision boundary) and the training samples that are closest to this hyperplane, which are the so-called **support vectors**. This is illustrated in the following figure:



Maximum margin intuition

The rationale behind having decision boundaries with large margins is that they tend to have a lower generalization error whereas models with small margins are more prone to overfitting. To get an intuition for the margin maximization, let's take a closer look at those *positive* and *negative* hyperplanes that are parallel to the decision boundary, which can be expressed as follows:

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{pos} = 1 \qquad (1)$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{neg} = -1 \qquad (2)$$

If we subtract those two linear equations (**1**) and (**2**) from each other, we get:

$$\Rightarrow \boldsymbol{w}^T \left(\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg} \right) = 2$$

We can normalize this by the length of the vector w, which is defined as follows:

$$\|\boldsymbol{w}\| = \sqrt{\sum_{j=1}^{m} w_j^2}$$

So we arrive at the following equation:

$$\frac{\boldsymbol{w}^T \left(\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg} \right)}{\|\boldsymbol{w}\|} = \frac{2}{\|\boldsymbol{w}\|}$$

The left side of the preceding equation can then be interpreted as the distance between the positive and negative hyperplane, which is the so-called margin that we want to maximize.

Now the objective function of the SVM becomes the maximization of this margin by maximizing $\frac{2}{\|\boldsymbol{w}\|}$ under the constraint that the samples are classified correctly, which can be written as follows:

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1 \; if \; y^{(i)} = 1$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} < -1 \; if \; y^{(i)} = -1$$

These two equations basically say that all negative samples should fall on one side of the negative hyperplane, whereas all the positive samples should fall behind the positive hyperplane. This can also be written more compactly as follows:

$$y^{(i)} \left(w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \right) \geq 1 \; \forall_i$$

In practice, though, it is easier to minimize the reciprocal term $\frac{1}{2}\|\boldsymbol{w}\|^2$, which can be solved by quadratic programming. However, a detailed discussion about quadratic programming is beyond the scope of this book, but if you are interested, you can learn more about **Support Vector Machines** (**SVM**) in Vladimir Vapnik's *The Nature of Statistical Learning Theory*, *Springer Science & Business Media*, or Chris J.C. Burges' excellent explanation in *A Tutorial on Support Vector Machines for Pattern Recognition* (Data mining and knowledge discovery, 2(2):121–167, 1998).

Dealing with the nonlinearly separable case using slack variables

Although we don't want to dive much deeper into the more involved mathematical concepts behind the margin classification, let's briefly mention the slack variable ξ. It was introduced by Vladimir Vapnik in 1995 and led to the so-called soft-margin classification. The motivation for introducing the slack variable ξ was that the linear constraints need to be relaxed for nonlinearly separable data to allow convergence of the optimization in the presence of misclassifications under the appropriate cost penalization.

The positive-values slack variable is simply added to the linear constraints:

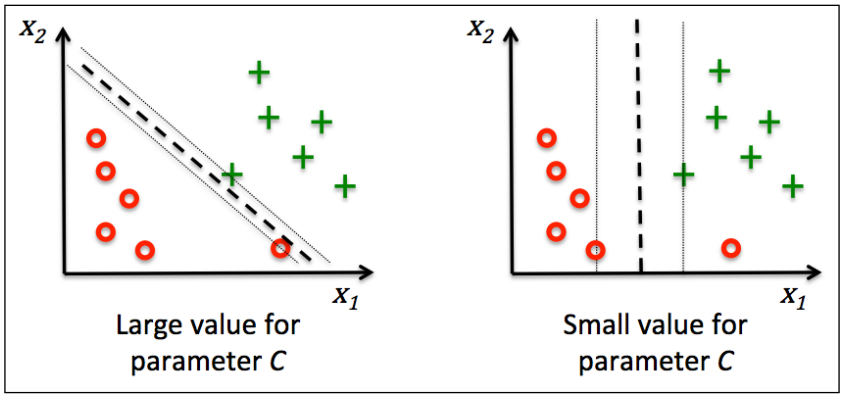
$$\boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1 - \xi^{(i)} \; if \; \; y^{(i)} = 1$$

$$\boldsymbol{w}^T \boldsymbol{x}^{(i)} \leq -1 + \xi^{(i)} \; if \; \; y^{(i)} = -1$$

So the new objective to be minimized (subject to the preceding constraints) becomes:

$$\frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

Using the variable `C`, we can then control the penalty for misclassification. Large values of `C` correspond to large error penalties whereas we are less strict about misclassification errors if we choose smaller values for `C`. We can then we use the parameter `C` to control the width of the margin and therefore tune the bias-variance trade-off as illustrated in the following figure:

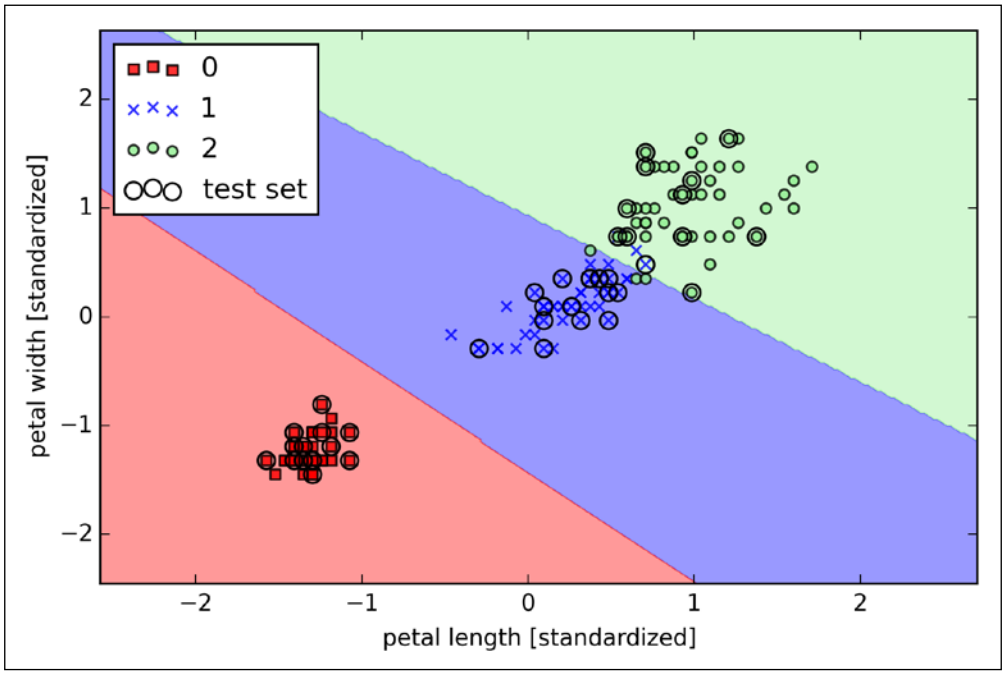


This concept is related to regularization, which we discussed previously in the context of regularized regression where increasing the value of `C` increases the bias and lowers the variance of the model.

Now that we learned the basic concepts behind the linear SVM, let's train a SVM model to classify the different flowers in our Iris dataset:

```
>>> from sklearn.svm import SVC
   >>> svm = SVC(kernel='linear', C=1.0, random_state=0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

The decision regions of the SVM visualized after executing the preceding code example are shown in the following plot:



Logistic regression versus SVM

In practical classification tasks, linear logistic regression and linear SVMs often yield very similar results. Logistic regression tries to maximize the conditional likelihoods of the training data, which makes it more prone to outliers than SVMs. The SVMs mostly care about the points that are closest to the decision boundary (support vectors). On the other hand, logistic regression has the advantage that it is a simpler model that can be implemented more easily. Furthermore, logistic regression models can be easily updated, which is attractive when working with streaming data.

Alternative implementations in scikit-learn

The `Perceptron` and `LogisticRegression` classes that we used in the previous sections via scikit-learn make use of the LIBLINEAR library, which is a highly optimized C/C++ library developed at the National Taiwan University (`http://www.csie.ntu.edu.tw/~cjlin/liblinear/`). Similarly, the `SVC` class that we used to train an SVM makes use of LIBSVM, which is an equivalent C/C++ library specialized for SVMs (`http://www.csie.ntu.edu.tw/~cjlin/libsvm/`).

The advantage of using LIBLINEAR and LIBSVM over native Python implementations is that they allow an extremely quick training of large amounts of linear classifiers. However, sometimes our datasets are too large to fit into computer memory. Thus, scikit-learn also offers alternative implementations via the `SGDClassifier` class, which also supports online learning via the `partial_fit` method. The concept behind the `SGDClassifier` class is similar to the stochastic gradient algorithm that we implemented in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, for Adaline. We could initialize the stochastic gradient descent version of the perceptron, logistic regression, and support vector machine with default parameters as follows:

```
>>> from sklearn.linear_model import SGDClassifier
>>> ppn = SGDClassifier(loss='perceptron')
>>> lr = SGDClassifier(loss='log')
>>> svm = SGDClassifier(loss='hinge')
```

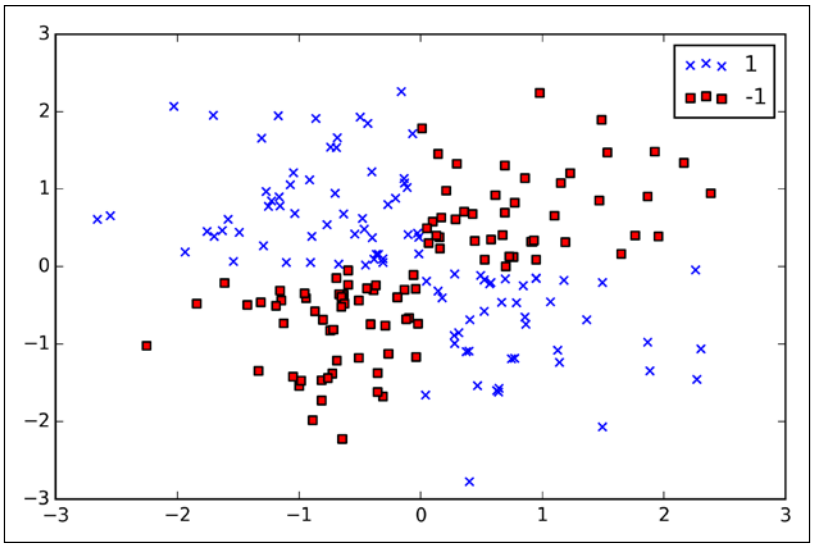
Solving nonlinear problems using a kernel SVM

Another reason why SVMs enjoy high popularity among machine learning practitioners is that they can be easily *kernelized* to solve nonlinear classification problems. Before we discuss the main concept behind **kernel SVM**, let's first define and create a sample dataset to see how such a nonlinear classification problem may look.

Using the following code, we will create a simple dataset that has the form of an XOR gate using the `logical_xor` function from NumPy, where 100 samples will be assigned the class label 1 and 100 samples will be assigned the class label -1, respectively:

```
>>> np.random.seed(0)
>>> X_xor = np.random.randn(200, 2)
>>> y_xor = np.logical_xor(X_xor[:, 0] > 0, X_xor[:, 1] > 0)
>>> y_xor = np.where(y_xor, 1, -1)

>>> plt.scatter(X_xor[y_xor==1, 0], X_xor[y_xor==1, 1],
...             c='b', marker='x', label='1')
>>> plt.scatter(X_xor[y_xor==-1, 0], X_xor[y_xor==-1, 1],
...             c='r', marker='s', label='-1')
>>> plt.ylim(-3.0)
>>> plt.legend()
>>> plt.show()
```

After executing the code, we will have an XOR dataset with random noise, as shown in the following figure:

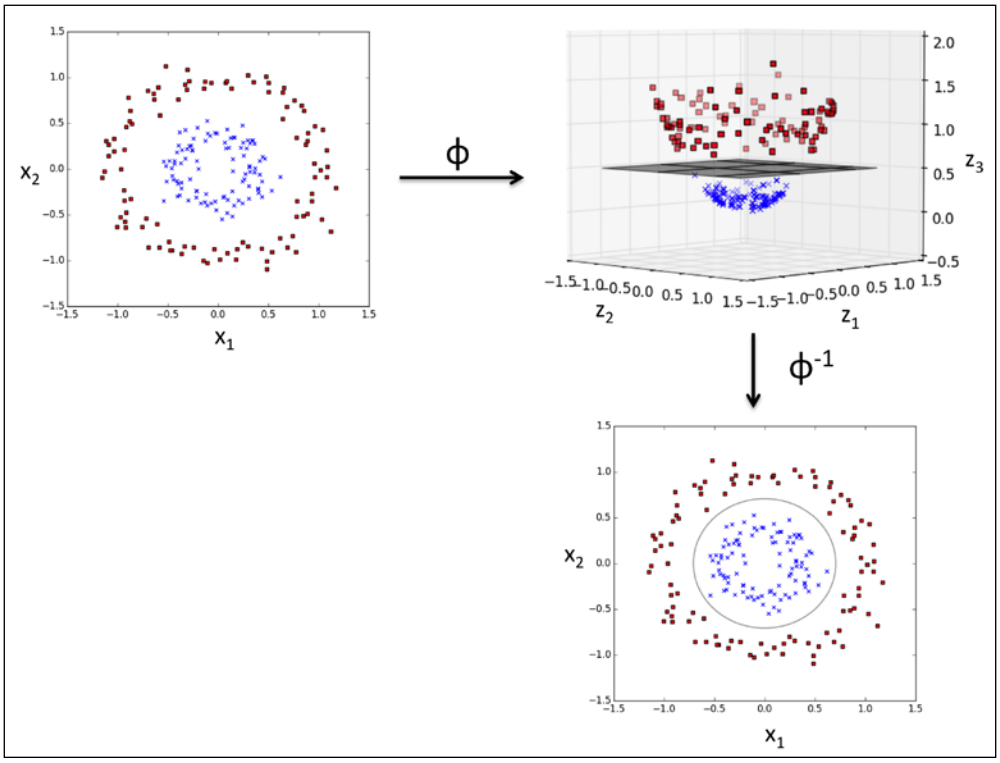


Obviously, we would not be able to separate samples from the positive and negative class very well using a linear hyperplane as the decision boundary via the linear logistic regression or linear SVM model that we discussed in earlier sections.

The basic idea behind kernel methods to deal with such linearly inseparable data is to create nonlinear combinations of the original features to project them onto a higher dimensional space via a mapping function $\phi(\cdot)$ where it becomes linearly separable. As shown in the next figure, we can transform a two-dimensional dataset onto a new three-dimensional feature space where the classes become separable via the following projection:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

This allows us to separate the two classes shown in the plot via a linear hyperplane that becomes a nonlinear decision boundary if we project it back onto the original feature space:



Using the kernel trick to find separating hyperplanes in higher dimensional space

To solve a nonlinear problem using an SVM, we transform the training data onto a higher dimensional feature space via a mapping function $\phi(\cdot)$ and train a linear SVM model to classify the data in this new feature space. Then we can use the same mapping function $\phi(\cdot)$ to transform new, unseen data to classify it using the linear SVM model.

However, one problem with this mapping approach is that the construction of the new features is computationally very expensive, especially if we are dealing with high-dimensional data. This is where the so-called kernel trick comes into play. Although we didn't go into much detail about how to solve the quadratic programming task to train an SVM, in practice all we need is to replace the dot product $\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}$ by $\phi\left(\boldsymbol{x}^{(i)}\right)^T\phi\left(\boldsymbol{x}^{(j)}\right)$. In order to save the expensive step of calculating this dot product between two points explicitly, we define a so-called kernel function: $k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right) = \phi\left(\boldsymbol{x}^{(i)}\right)^T\phi\left(\boldsymbol{x}^{(j)}\right)$.

One of the most widely used kernels is the **Radial Basis Function kernel** (**RBF kernel**) or Gaussian kernel:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right) = \exp\left(-\frac{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^2}{2\sigma^2}\right)$$

This is often simplified to:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right) = \exp\left(-\gamma\left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^2\right)$$

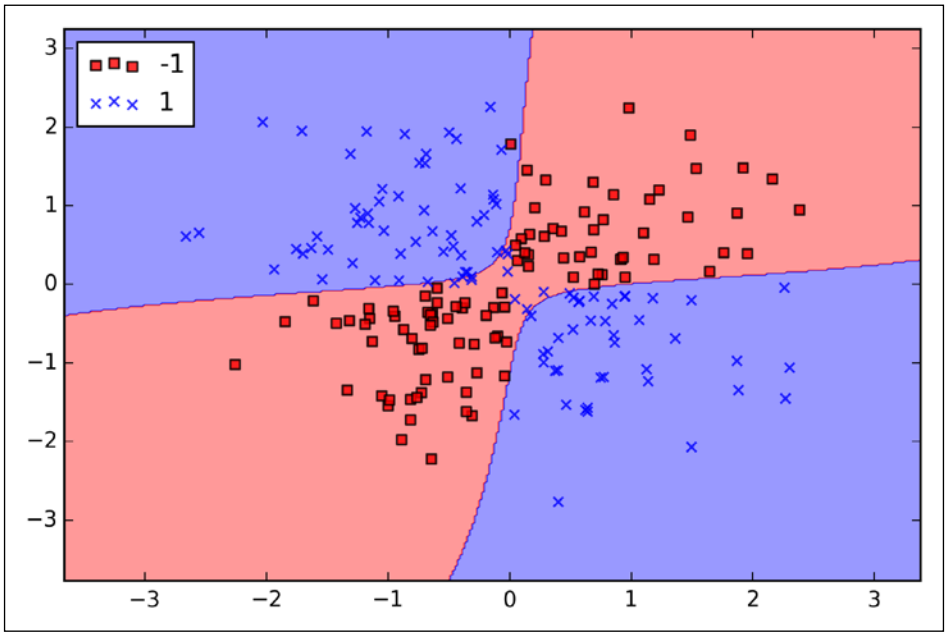
Here, $\gamma = \frac{1}{2\sigma^2}$ is a free parameter that is to be optimized.

Roughly speaking, the term *kernel* can be interpreted as a *similarity function* between a pair of samples. The minus sign inverts the distance measure into a similarity score and, due to the exponential term, the resulting similarity score will fall into a range between 1 (for exactly similar samples) and 0 (for very dissimilar samples).

Now that we defined the big picture behind the kernel trick, let's see if we can train a kernel SVM that is able to draw a nonlinear decision boundary that separates the XOR data well. Here, we simply use the `SVC` class from scikit-learn that we imported earlier and replace the parameter `kernel='linear'` with `kernel='rbf'`:

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=0.10, C=10.0)
>>> svm.fit(X_xor, y_xor)
>>> plot_decision_regions(X_xor, y_xor, classifier=svm)
>>> plt.legend(loc='upper left')
>>> plt.show()
```

As we can see in the resulting plot, the kernel SVM separates the XOR data relatively well:

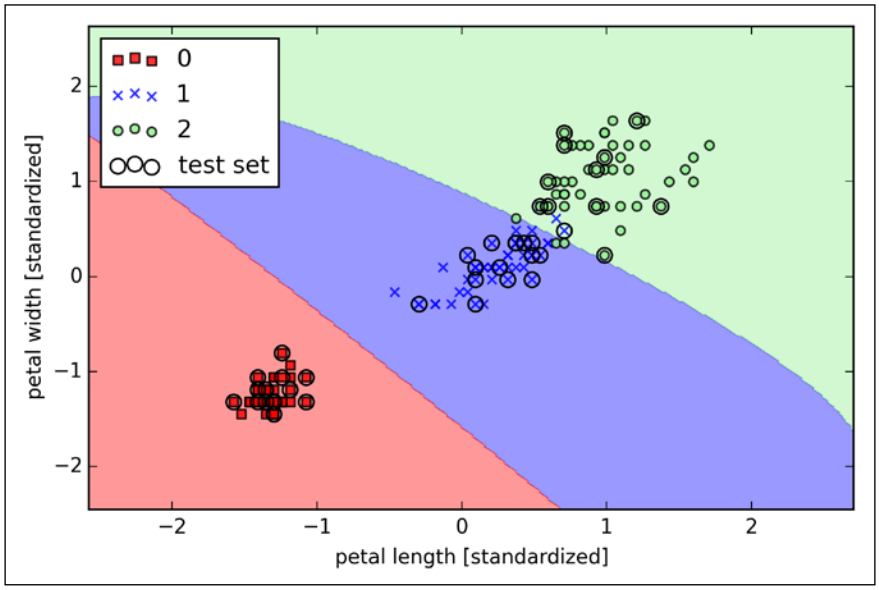


The γ parameter, which we set to `gamma=0.1`, can be understood as a *cut-off* parameter for the Gaussian sphere. If we increase the value for γ, we increase the influence or reach of the training samples, which leads to a softer decision boundary. To get a better intuition for γ, let's apply RBF kernel SVM to our Iris flower dataset:

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=0.2, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
```

```
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

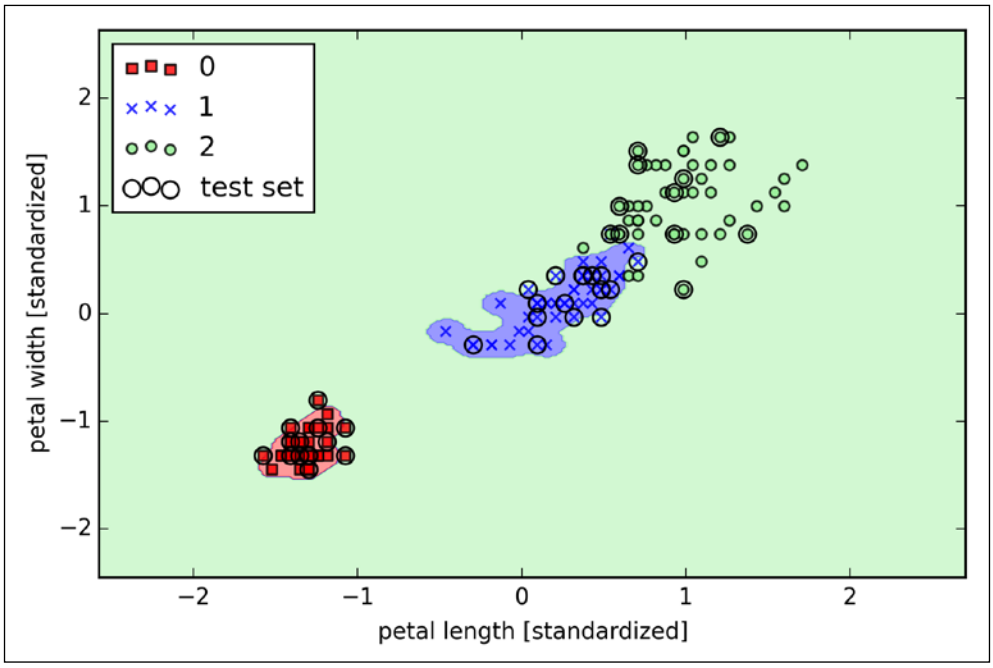
Since we chose a relatively small value for γ, the resulting decision boundary of the RBF kernel SVM model will be relatively soft, as shown in the following figure:



Now let's increase the value of γ and observe the effect on the decision boundary:

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=100.0, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

In the resulting plot, we can now see that the decision boundary around the classes 0 and 1 is much tighter using a relatively large value of γ:

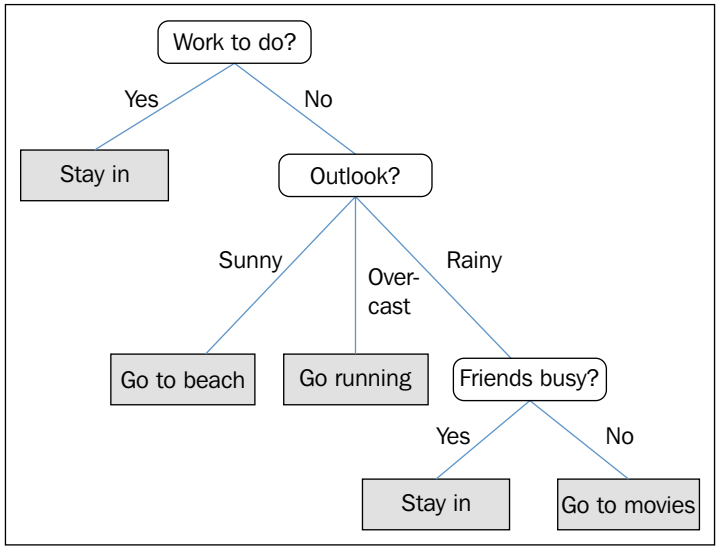


Although the model fits the training dataset very well, such a classifier will likely have a high generalization error on unseen data, which illustrates that the optimization of γ also plays an important role in controlling overfitting.

Decision tree learning

Although we've seen what is decision tree in the previous module, let's dive more deeper. **Decision tree** classifiers are attractive models if we care about interpretability. Like the name *decision tree* suggests, we can think of this model as breaking down our data by making decisions based on asking a series of questions.

Let's consider the following example where we use a decision tree to decide upon an activity on a particular day:



Based on the features in our training set, the decision tree model learns a series of questions to infer the class labels of the samples. Although the preceding figure illustrated the concept of a decision tree based on categorical variables, the same concept applies if our features are real numbers like in the Iris dataset. For example, we could simply define a cut-off value along the **sepal width** feature axis and ask a binary question "sepal width ≥ 2.8 cm?"

Using the decision algorithm, we start at the tree root and split the data on the feature that results in the largest **information gain** (**IG**), which will be explained in more detail in the following section. In an iterative process, we can then repeat this splitting procedure at each child node until the leaves are pure. This means that the samples at each node all belong to the same class. In practice, this can result in a very deep tree with many nodes, which can easily lead to overfitting. Thus, we typically want to *prune* the tree by setting a limit for the maximal depth of the tree.

Maximizing information gain – getting the most bang for the buck

In order to split the nodes at the most informative features, we need to define an objective function that we want to optimize via the tree learning algorithm. Here, our objective function is to maximize the information gain at each split, which we define as follows:

$$IG\left(D_p, f\right) = I\left(D_p\right) - \sum_{j=1}^{m} \frac{N_j}{N_p} I\left(D_j\right)$$

Here, *f* is the feature to perform the split, D_p and D_j are the dataset of the parent and *j*th child node, *I* is our impurity measure, N_p is the total number of samples at the parent node, and N_j is the number of samples in the *j*th child node. As we can see, the information gain is simply the difference between the impurity of the parent node and the sum of the child node impurities—the lower the impurity of the child nodes, the larger the information gain. However, for simplicity and to reduce the combinatorial search space, most libraries (including scikit-learn) implement binary decision trees. This means that each parent node is split into two child nodes, D_{left} and D_{right}:

$$IG\left(D_p, f\right) = I\left(D_p\right) - \frac{N_{left}}{N_p} I\left(D_{left}\right) - \frac{N_{right}}{N_p} I\left(D_{right}\right)$$

Now, the three impurity measures or splitting criteria that are commonly used in binary decision trees are **Gini impurity** (I_G), **entropy** (I_H), and the **classification error** (I_E). Let's start with the definition of entropy for all **non-empty** classes $p\left(i \mid t\right) \neq 0$:

$$I_H\left(t\right) = -\sum_{i=1}^{c} p\left(i \mid t\right) \log_2 p\left(i \mid t\right)$$

Here, $p(i|t)$ is the proportion of the samples that belongs to class *i* for a particular node *t*. The entropy is therefore 0 if all samples at a node belong to the same class, and the entropy is maximal if we have a uniform class distribution. For example, in a binary class setting, the entropy is 0 if $p(i=1|t)=1$ or $p(i=0|t)=0$. If the classes are distributed uniformly with $p(i=1|t)=0.5$ and $p(i=0|t)=0.5$, the entropy is 1. Therefore, we can say that the entropy criterion attempts to maximize the mutual information in the tree.

Intuitively, the Gini impurity can be understood as a criterion to minimize the probability of misclassification:

$$I_G(t) = \sum_{i=1}^{c} p(i|t)\big(1 - p(i|t)\big) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

Similar to entropy, the Gini impurity is maximal if the classes are perfectly mixed, for example, in a binary class setting ($c=2$):

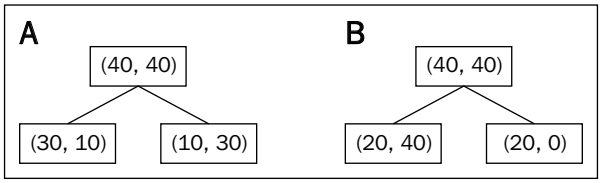
$$1 - \sum_{i=1}^{c} 0.5^2 = 0.5$$

However, in practice both the Gini impurity and entropy typically yield very similar results and it is often not worth spending much time on evaluating trees using different impurity criteria rather than experimenting with different pruning cut-offs.

Another impurity measure is the classification error:

$$I_E = 1 - \max\{p(i|t)\}$$

This is a useful criterion for pruning but not recommended for growing a decision tree, since it is less sensitive to changes in the class probabilities of the nodes. We can illustrate this by looking at the two possible splitting scenarios shown in the following figure:



We start with a dataset D_p at the parent node D_p that consists of 40 samples from class 1 and 40 samples from class 2 that we split into two datasets D_{left} and D_{right}, respectively. The information gain using the classification error as a splitting criterion would be the same ($IG_E = 0.25$) in both scenario A and B:

$$I_E\left(D_p\right) = 1 - 0.5 = 0.5$$

$$A: I_E\left(D_{left}\right) = 1 - \frac{3}{4} = 0.25$$

$$A: I_E\left(D_{right}\right) = 1 - \frac{3}{4} = 0.25$$

$$A: IG_E = 0.5 - \frac{4}{8}0.25 - \frac{4}{8}0.25 = 0.25$$

$$B: I_E\left(D_{left}\right) = 1 - \frac{4}{6} = \frac{1}{3}$$

$$B: I_E\left(D_{right}\right) = 1 - 1 = 0$$

$$B: IG_E = 0.5 - \frac{6}{8} \times \frac{1}{3} - 0 = 0.25$$

However, the Gini impurity would favor the split in scenario $B\left(IG_G = 0.1\overline{6}\right)$ over scenario $A\left(IG_G = 0.125\right)$, which is indeed more *pure*:

$$I_G\left(D_p\right) = 1 - \left(0.5^2 + 0.5^2\right) = 0.5$$

$$A: I_G\left(D_{left}\right) = 1 - \left(\left(\frac{3}{4}\right)^2 + \left(\frac{1}{4}\right)^2\right) = \frac{3}{8} = 0.375$$

$$A: I_G\left(D_{right}\right) = 1 - \left(\left(\frac{1}{4}\right)^2 + \left(\frac{3}{4}\right)^2\right) = \frac{3}{8} = 0.375$$

$$A: IG_G = 0.5 - \frac{4}{8}0.375 - \frac{4}{8}0.375 = 0.125$$

$$B: I_G\left(D_{left}\right) = 1 - \left(\left(\frac{2}{6}\right)^2 + \left(\frac{4}{6}\right)^2\right) = \frac{4}{9} = 0.\overline{4}$$

$$B: I_G\left(D_{right}\right) = 1 - \left(1^2 + 0^2\right) = 0$$

$$B: IG_G = 0.5 - \frac{6}{8}0.\overline{4} - 0 = 0.\overline{16}$$

Similarly, the entropy criterion would favor scenario $B\left(IG_H = 0.31\right)$ over scenario $A\left(IG_H = 0.19\right)$:

$$I_H\left(D_p\right) = -\left(0.5\ \log_2\left(0.5\right) + 0.5\ \log_2\left(0.5\right)\right) = 1$$

$$A: I_H\left(D_{left}\right) = -\left(\frac{3}{4}\log_2\left(\frac{3}{4}\right) + \frac{1}{4}\log_2\left(\frac{1}{4}\right)\right) = 0.81$$

$$A: I_H\left(D_{right}\right) = -\left(\frac{1}{4}\log_2\left(\frac{1}{4}\right) + \frac{3}{4}\log_2\left(\frac{3}{4}\right)\right) = 0.81$$

$$A: IG_H = 1 - \frac{4}{8}0.81 - \frac{4}{8}0.81 = 0.19$$

$$B: I_H\left(D_{left}\right) = -\left(\frac{2}{6}\log_2\left(\frac{2}{6}\right) + \frac{4}{6}\log_2\left(\frac{4}{6}\right)\right) = 0.92$$

$$B: I_H\left(D_{right}\right) = 0$$

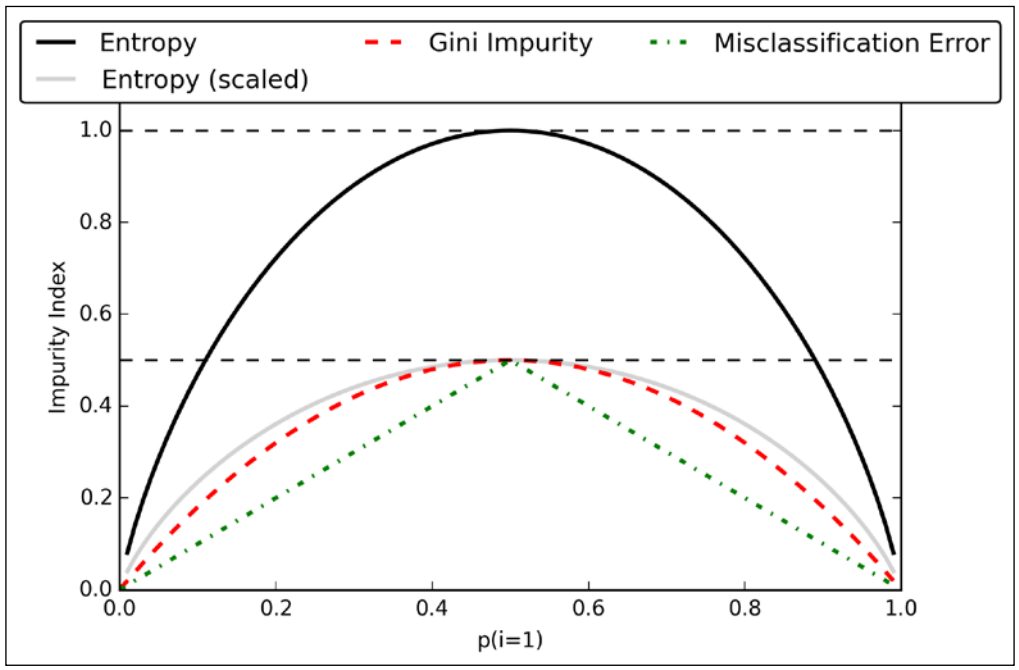
$$B: IG_H = 1 - \frac{6}{8}0.92 - 0 = 0.31$$

For a more visual comparison of the three different impurity criteria that we discussed previously, let's plot the impurity indices for the probability range [0, 1] for class 1. Note that we will also add in a scaled version of the entropy (*entropy/2*) to observe that the Gini impurity is an intermediate measure between entropy and the classification error. The code is as follows:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def gini(p):
...     return (p)*(1 - (p)) + (1 - p)*(1 - (1-p))
>>> def entropy(p):
...     return - p*np.log2(p) - (1 - p)*np.log2((1 - p))
>>> def error(p):
...     return 1 - np.max([p, 1 - p])
>>> x = np.arange(0.0, 1.0, 0.01)
>>> ent = [entropy(p) if p != 0 else None for p in x]
>>> sc_ent = [e*0.5 if e else None for e in ent]
>>> err = [error(i) for i in x]
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
...                   ['Entropy', 'Entropy (scaled)',
...                   'Gini Impurity',
```

```
...                    'Misclassification Error'],
...                    ['-', '-', '--', '-.'],
...                    ['black', 'lightgray',
...                       'red', 'green', 'cyan']):
...     line = ax.plot(x, i, label=lab,
...                    linestyle=ls, lw=2, color=c)
>>> ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
...           ncol=3, fancybox=True, shadow=False)
>>> ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
>>> ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
>>> plt.ylim([0, 1.1])
>>> plt.xlabel('p(i=1)')
>>> plt.ylabel('Impurity Index')
>>> plt.show()
```

The plot produced by the preceding code example is as follows:

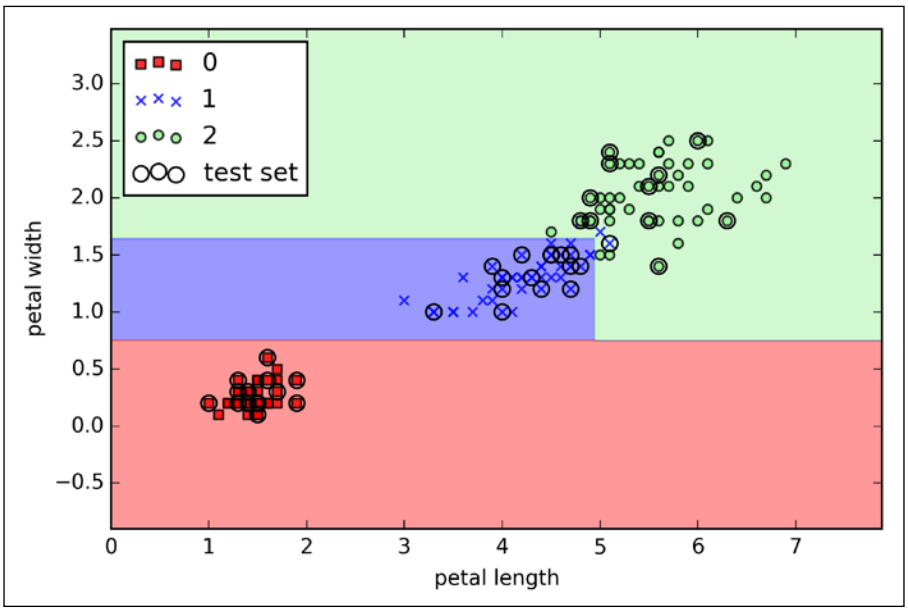


Building a decision tree

Decision trees can build complex decision boundaries by dividing the feature space into rectangles. However, we have to be careful since the deeper the decision tree, the more complex the decision boundary becomes, which can easily result in overfitting. Using scikit-learn, we will now train a decision tree with a maximum depth of 3 using entropy as a criterion for impurity. Although feature scaling may be desired for visualization purposes, note that feature scaling is not a requirement for decision tree algorithms. The code is as follows:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=3, random_state=0)
>>> tree.fit(X_train, y_train)
>>> X_combined = np.vstack((X_train, X_test))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X_combined, y_combined,
...                       classifier=tree, test_idx=range(105,150))
>>>plt.xlabel('petal length [cm]')
>>>plt.ylabel('petal width [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example, we get the typical axis-parallel decision boundaries of the decision tree:



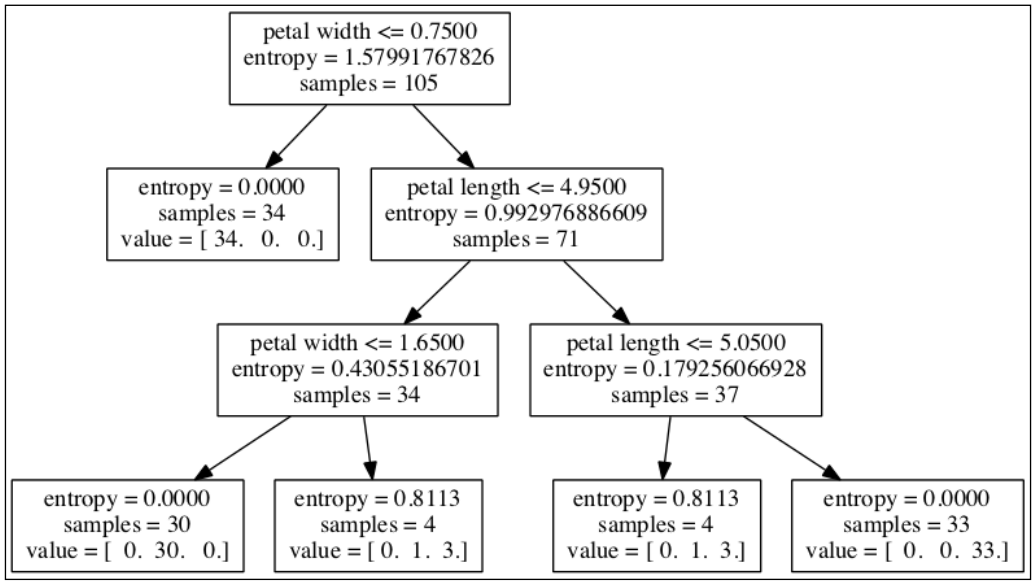
A nice feature in scikit-learn is that it allows us to export the decision tree as a `.dot` file after training, which we can visualize using the GraphViz program. This program is freely available at `http://www.graphviz.org` and supported by Linux, Windows, and Mac OS X.

First, we create the `.dot` file via scikit-learn using the `export_graphviz` function from the `tree` submodule, as follows:

```
>>> from sklearn.tree import export_graphviz
>>> export_graphviz(tree,
...                 out_file='tree.dot',
...                 feature_names=['petal length', 'petal width'])
```

After we have installed GraphViz on our computer, we can convert the `tree.dot` file into a PNG file by executing the following command from the command line in the location where we saved the `tree.dot` file:

```
> dot -Tpng tree.dot -o tree.png
```



Looking at the decision tree figure that we created via GraphViz, we can now nicely trace back the splits that the decision tree determined from our training dataset. We started with 105 samples at the root and split it into two child nodes with 34 and 71 samples each using the **petal width** cut-off ≤ 0.75 cm. After the first split, we can see that the left child node is already pure and only contains samples from the Iris-Setosa class (entropy = 0). The further splits on the right are then used to separate the samples from the Iris-Versicolor and Iris-Virginica classes.

Combining weak to strong learners via random forests

Random forests have gained huge popularity in applications of machine learning during the last decade due to their good classification performance, scalability, and ease of use. Intuitively, a random forest can be considered as an *ensemble* of decision trees. The idea behind ensemble learning is to combine **weak learners** to build a more robust model, a **strong learner**, that has a better generalization error and is less susceptible to overfitting. The random forest algorithm can be summarized in four simple steps:

1. Draw a random **bootstrap** sample of size *n* (randomly choose *n* samples from the training set with replacement).
2. Grow a decision tree from the bootstrap sample. At each node:
    1. Randomly select *d* features without replacement.
    2. Split the node using the feature that provides the best split according to the objective function, for instance, by maximizing the information gain.
3. Repeat the steps 1 to 2 *k* times.
4. Aggregate the prediction by each tree to assign the class label by **majority vote**. Majority voting will be discussed in more detail in *Chapter 7*, *Combining Different Models for Ensemble Learning*.

There is a slight modification in step 2 when we are training the individual decision trees: instead of evaluating all features to determine the best split at each node, we only consider a random subset of those.

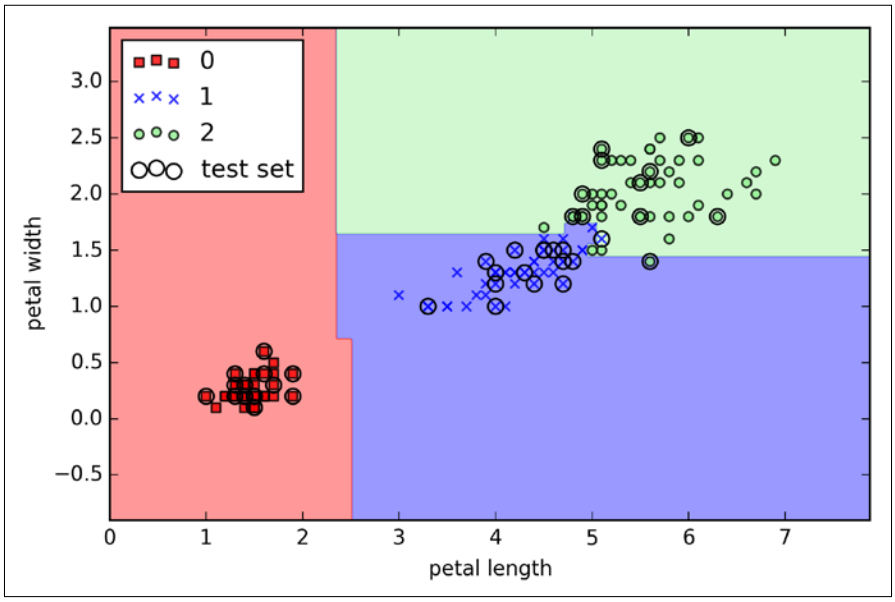
Although random forests don't offer the same level of interpretability as decision trees, a big advantage of random forests is that we don't have to worry so much about choosing good hyperparameter values. We typically don't need to prune the random forest since the ensemble model is quite robust to noise from the individual decision trees. The only parameter that we really need to care about in practice is the number of trees *k* (step 3) that we choose for the random forest. Typically, the larger the number of trees, the better the performance of the random forest classifier at the expense of an increased computational cost.

Although it is less common in practice, other hyperparameters of the random forest classifier that can be optimized—using techniques we will discuss in *Chapter 5*, *Compressing Data via Dimensionality Reduction*—are the size *n* of the bootstrap sample (step 1) and the number of features *d* that is randomly chosen for each split (step 2.1), respectively. Via the sample size *n* of the bootstrap sample, we control the bias-variance tradeoff of the random forest. By choosing a larger value for *n*, we decrease the randomness and thus the forest is more likely to overfit. On the other hand, we can reduce the degree of overfitting by choosing smaller values for *n* at the expense of the model performance. In most implementations, including the `RandomForestClassifier` implementation in scikit-learn, the sample size of the bootstrap sample is chosen to be equal to the number of samples in the original training set, which usually provides a good bias-variance tradeoff. For the number of features *d* at each split, we want to choose a value that is smaller than the total number of features in the training set. A reasonable default that is used in scikit-learn and other implementations is $d = \sqrt{m}$, where *m* is the number of features in the training set.

Conveniently, we don't have to construct the random forest classifier from individual decision trees by ourselves; there is already an implementation in scikit-learn that we can use:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> forest = RandomForestClassifier(criterion='entropy',
...                                 n_estimators=10,
...                                 random_state=1,
...                                 n_jobs=2)
>>> forest.fit(X_train, y_train)
>>> plot_decision_regions(X_combined, y_combined,
...                       classifier=forest, test_idx=range(105,150))
>>> plt.xlabel('petal length')
>>> plt.ylabel('petal width')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code, we should see the decision regions formed by the ensemble of trees in the random forest, as shown in the following figure:



Using the preceding code, we trained a random forest from 10 decision trees via the `n_estimators` parameter and used the entropy criterion as an impurity measure to split the nodes. Although we are growing a very small random forest from a very small training dataset, we used the `n_jobs` parameter for demonstration purposes, which allows us to parallelize the model training using multiple cores of our computer (here, two).

K-nearest neighbors – a lazy learning algorithm

The last supervised learning algorithm that we want to discuss in this chapter is the **k-nearest neighbor classifier** (**KNN**), which is particularly interesting because it is fundamentally different from the learning algorithms that we have discussed so far.

KNN is a typical example of a **lazy learner**. It is called *lazy* not because of its apparent simplicity, but because it doesn't learn a discriminative function from the training data but memorizes the training dataset instead.

Parametric versus nonparametric models

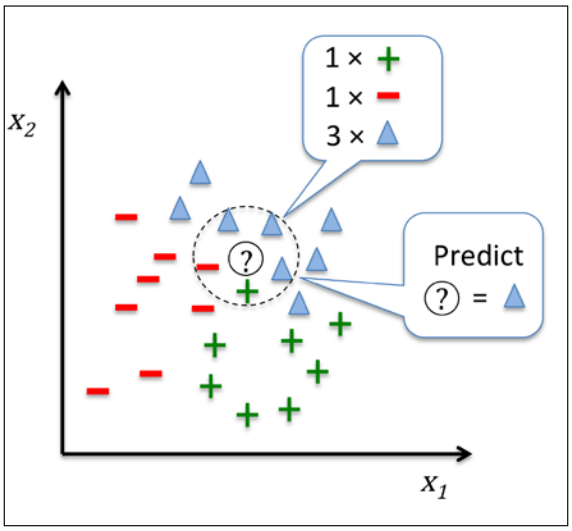
Machine learning algorithms can be grouped into **parametric** and **nonparametric** models. Using parametric models, we estimate parameters from the training dataset to learn a function that can classify new data points without requiring the original training dataset anymore. Typical examples of parametric models are the perceptron, logistic regression, and the linear SVM. In contrast, nonparametric models can't be characterized by a fixed set of parameters, and the number of parameters grows with the training data. Two examples of nonparametric models that we have seen so far are the decision tree classifier/random forest and the kernel SVM.

KNN belongs to a subcategory of nonparametric models that is described as **instance-based learning**. Models based on instance-based learning are characterized by memorizing the training dataset, and lazy learning is a special case of instance-based learning that is associated with no (zero) cost during the learning process.

The KNN algorithm itself is fairly straightforward and can be summarized by the following steps:

1. Choose the number of *k* and a distance metric.
2. Find the *k* nearest neighbors of the sample that we want to classify.
3. Assign the class label by majority vote.

The following figure illustrates how a new data point (**?**) is assigned the triangle class label based on majority voting among its five nearest neighbors.



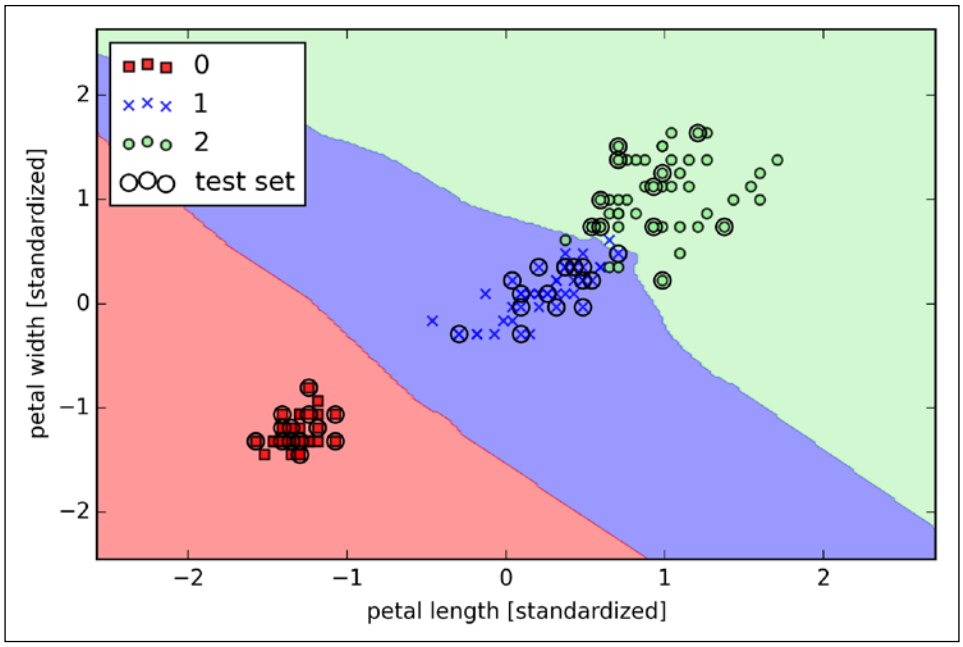
Based on the chosen distance metric, the KNN algorithm finds the *k* samples in the training dataset that are closest (most similar) to the point that we want to classify. The class label of the new data point is then determined by a majority vote among its *k* nearest neighbors.

The main advantage of such a memory-based approach is that the classifier immediately adapts as we collect new training data. However, the downside is that the computational complexity for classifying new samples grows linearly with the number of samples in the training dataset in the worst-case scenario—unless the dataset has very few dimensions (features) and the algorithm has been implemented using efficient data structures such as KD-trees. (J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software (TOMS), 3(3):209–226, 1977.) Furthermore, we can't discard training samples since no *training* step is involved. Thus, storage space can become a challenge if we are working with large datasets.

By executing the following code, we will now implement a KNN model in scikit-learn using an Euclidean distance metric:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                            metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                       classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.show()
```

By specifying five neighbors in the KNN model for this dataset, we obtain a relatively smooth decision boundary, as shown in the following figure:



In the case of a tie, the scikit-learn implementation of the KNN algorithm will prefer the neighbors with a closer distance to the sample. If the neighbors have a similar distance, the algorithm will choose the class label that comes first in the training dataset.

The *right* choice of *k* is crucial to find a good balance between over- and underfitting. We also have to make sure that we choose a distance metric that is appropriate for the features in the dataset. Often, a simple Euclidean distance measure is used for real-valued samples, for example, the flowers in our Iris dataset, which have features measured in centimeters. However, if we are using a Euclidean distance measure, it is also important to standardize the data so that each feature contributes equally to the distance. The `'minkowski'` distance that we used in the previous code is just a generalization of the Euclidean and Manhattan distance that can be written as follows:

$$d\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \sqrt[p]{\sum_k \left|x_k^{(i)} - x_k^{(j)}\right|^p}$$

It becomes the Euclidean distance if we set the parameter `p=2` or the Manhattan distance at `p=1`, respectively. Many other distance metrics are available in scikit-learn and can be provided to the `metric` parameter. They are listed at `http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html`.

The curse of dimensionality

It is important to mention that KNN is very susceptible to overfitting due to the **curse of dimensionality**. The curse of dimensionality describes the phenomenon where the feature space becomes increasingly sparse for an increasing number of dimensions of a fixed-size training dataset. Intuitively, we can think of even the closest neighbors being too far away in a high-dimensional space to give a good estimate.

We have discussed the concept of regularization in the section about logistic regression as one way to avoid overfitting. However, in models where regularization is not applicable such as decision trees and KNN, we can use feature selection and dimensionality reduction techniques to help us avoid the curse of dimensionality. This will be discussed in more detail in the next chapter.

Summary of Module 4 Chapter 3

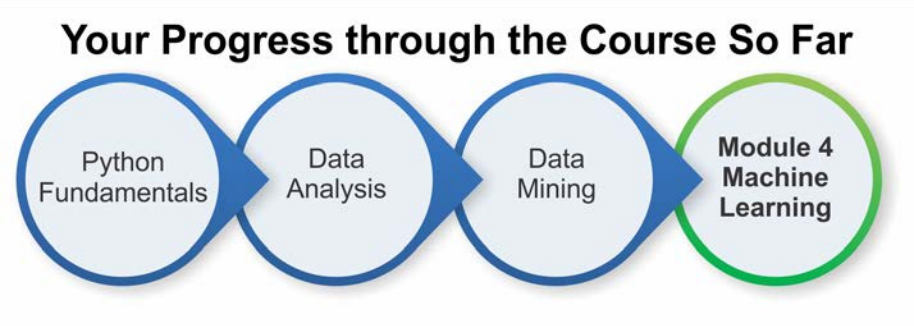
Ankita Thakur

Your Course Guide

In this chapter, you learned about many different machine algorithms that are used to tackle linear and nonlinear problems. We have seen that decision trees are particularly attractive if we care about interpretability. Logistic regression is not only a useful model for online learning via stochastic gradient descent, but also allows us to predict the probability of a particular event. Although support vector machines are powerful linear models that can be extended to nonlinear problems via the kernel trick, they have many parameters that have to be tuned in order to make good predictions. In contrast, ensemble methods such as random forests don't require much parameter tuning and don't overfit so easily as decision trees, which makes it an attractive model for many practical problem domains. The K-nearest neighbor classifier offers an alternative approach to classification via lazy learning that allows us to make predictions without any model training but with a more computationally expensive prediction step.

However, even more important than the choice of an appropriate learning algorithm is the available data in our training dataset. No algorithm will be able to make good predictions without informative and discriminatory features.

In the next chapter, we will discuss important topics regarding the preprocessing of data, feature selection, and dimensionality reduction, which we will need to build powerful machine learning models. Later in chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning, we will see how we can evaluate and compare the performance of our models and learn useful tricks to fine-tune the different algorithms.


Your Progress through the Course So Far
Python Fundamentals
Data Analysis
Data Mining
Module 4 Machine Learning

4
Building Good Training Sets – Data Preprocessing

The quality of the data and the amount of useful information that it contains are key factors that determine how well a machine learning algorithm can learn. Therefore, it is absolutely critical that we make sure to examine and preprocess a dataset before we feed it to a learning algorithm. In this chapter, we will discuss the essential data preprocessing techniques that will help us to build good machine learning models.

The topics that we will cover in this chapter are as follows:

- Removing and imputing missing values from the dataset
- Getting categorical data into shape for machine learning algorithms
- Selecting relevant features for the model construction

Dealing with missing data

It is not uncommon in real-world applications that our samples are missing one or more values for various reasons. There could have been an error in the data collection process, certain measurements are not applicable, particular fields could have been simply left blank in a survey, for example. We typically see *missing values* as the blank spaces in our data table or as placeholder strings such as `NaN` (Not A Number).

Unfortunately, most computational tools are unable to handle such missing values or would produce unpredictable results if we simply ignored them. Therefore, it is crucial that we take care of those missing values before we proceed with further analyses. But before we discuss several techniques for dealing with missing values, let's create a simple example data frame from a **CSV** (**comma-separated values**) file to get a better grasp of the problem:

```
>>> import pandas as pd
>>> from io import StringIO
>>> csv_data = '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,'''
>>> # If you are using Python 2.7, you need
>>> # to convert the string to unicode:
>>> # csv_data = unicode(csv_data)
>>> df = pd.read_csv(StringIO(csv_data))
>>> df
    A   B   C   D
0   1   2   3   4
1   5   6 NaN   8
2  10  11  12 NaN
```

Using the preceding code, we read CSV-formatted data into a pandas `DataFrame` via the `read_csv` function and noticed that the two missing cells were replaced by `NaN`. The `StringIO` function in the preceding code example was simply used for the purposes of illustration. It allows us to read the string assigned to `csv_data` into a pandas `DataFrame` as if it was a regular CSV file on our hard drive.

For a larger `DataFrame`, it can be tedious to look for missing values manually; in this case, we can use the `isnull` method to return a `DataFrame` with Boolean values that indicate whether a cell contains a numeric value (`False`) or if data is missing (`True`). Using the `sum` method, we can then return the number of missing values per column as follows:

```
>>> df.isnull().sum()
A    0
B    0
C    1
D    1
dtype: int64
```

This way, we can count the number of missing values per column; in the following subsections, we will take a look at different strategies for how to deal with this missing data.

Although scikit-learn was developed for working with NumPy arrays, it can sometimes be more convenient to preprocess data using pandas' `DataFrame`. We can always access the underlying NumPy array of the `DataFrame` via the `values` attribute before we feed it into a scikit-learn estimator:

```
>>> df.values
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  nan,   8.],
       [ 10.,  11.,  12.,  nan]])
```

Eliminating samples or features with missing values

One of the easiest ways to deal with missing data is to simply remove the corresponding features (columns) or samples (rows) from the dataset entirely; rows with missing values can be easily dropped via the `dropna` method:

```
>>> df.dropna()
   A  B  C  D
0  1  2  3  4
```

Similarly, we can drop columns that have at least one `NaN` in any row by setting the `axis` argument to `1`:

```
>>> df.dropna(axis=1)
   A   B
0  1   2
1  5   6
2  10  11
```

The `dropna` method supports several additional parameters that can come in handy:

```
# only drop rows where all columns are NaN
>>> df.dropna(how='all')

# drop rows that have not at least 4 non-NaN values
>>> df.dropna(thresh=4)

# only drop rows where NaN appear in specific columns (here: 'C')
>>> df.dropna(subset=['C'])
```

Although the removal of missing data seems to be a convenient approach, it also comes with certain disadvantages; for example, we may end up removing too many samples, which will make a reliable analysis impossible. Or, if we remove too many feature columns, we will run the risk of losing valuable information that our classifier needs to discriminate between classes. In the next section, we will thus look at one of the most commonly used alternatives for dealing with missing values: interpolation techniques.

Imputing missing values

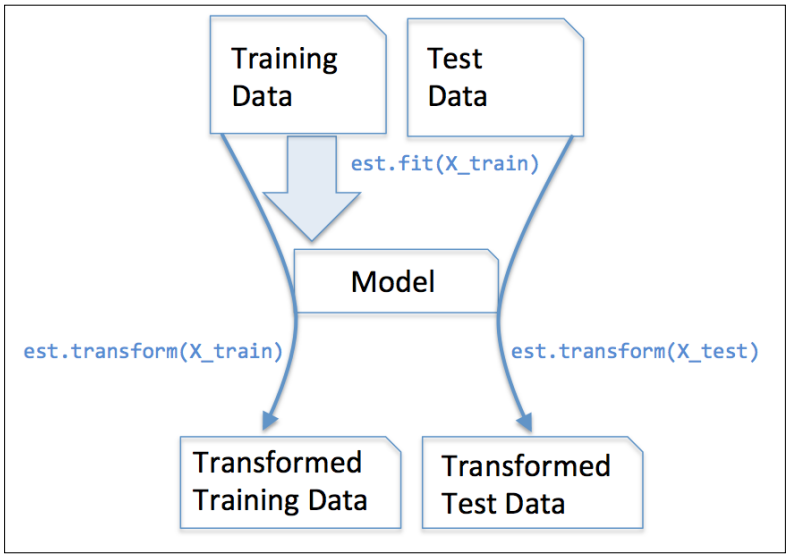
Often, the removal of samples or dropping of entire feature columns is simply not feasible, because we might lose too much valuable data. In this case, we can use different interpolation techniques to estimate the missing values from the other training samples in our dataset. One of the most common interpolation techniques is **mean imputation**, where we simply replace the missing value by the mean value of the entire feature column. A convenient way to achieve this is by using the `Imputer` class from scikit-learn, as shown in the following code:

```
>>> from sklearn.preprocessing import Imputer
>>> imr = Imputer(missing_values='NaN', strategy='mean', axis=0)
>>> imr = imr.fit(df)
>>> imputed_data = imr.transform(df.values)
>>> imputed_data
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  7.5,   8.],
       [ 10.,  11.,  12.,   6.]])
```

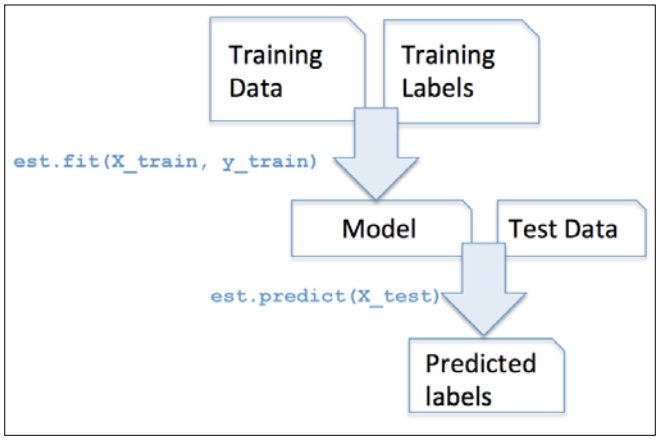
Here, we replaced each `NaN` value by the corresponding mean, which is separately calculated for each feature column. If we changed the setting `axis=0` to `axis=1`, we'd calculate the row means. Other options for the `strategy` parameter are `median` or `most_frequent`, where the latter replaces the missing values by the most frequent values. This is useful for imputing categorical feature values.

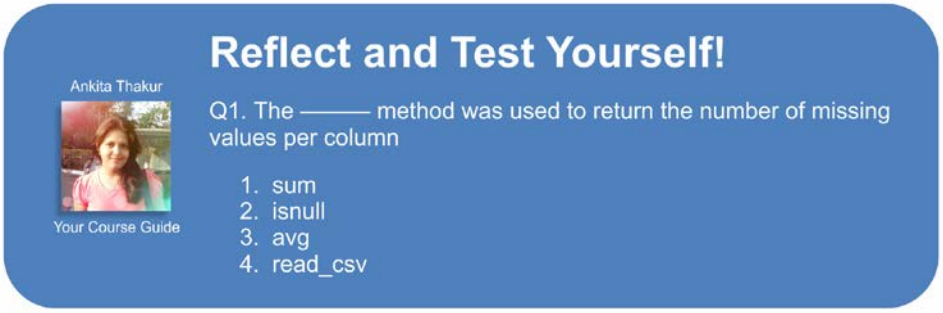
Understanding the scikit-learn estimator API

In the previous section, we used the `Imputer` class from scikit-learn to impute missing values in our dataset. The `Imputer` class belongs to the so-called **transformer** classes in scikit-learn that are used for data transformation. The two essential methods of those estimators are `fit` and `transform`. The `fit` method is used to learn the parameters from the training data, and the `transform` method uses those parameters to transform the data. Any data array that is to be transformed needs to have the same number of features as the data array that was used to fit the model. The following figure illustrates how a transformer fitted on the training data is used to transform a training dataset as well as a new test dataset:



The classifiers that we used in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*, belong to the so-called estimators in scikit-learn with an API that is conceptually very similar to the transformer class. Estimators have a `predict` method but can also have a `transform` method, as we will see later. As you may recall, we also used the `fit` method to learn the parameters of a model when we trained those estimators for classification. However, in supervised learning tasks, we additionally provide the class labels for fitting the model, which can then be used to make predictions about new data samples via the `predict` method, as illustrated in the following figure:





Handling categorical data

So far, we have only been working with numerical values. However, it is not uncommon that real-world datasets contain one or more categorical feature columns. When we are talking about categorical data, we have to further distinguish between **nominal** and **ordinal** features. Ordinal features can be understood as categorical values that can be sorted or ordered. For example, *T-shirt size* would be an ordinal feature, because we can define an order *XL* > *L* > *M*. In contrast, nominal features don't imply any order and, to continue with the previous example, we could think of *T-shirt color* as a nominal feature since it typically doesn't make sense to say that, for example, *red* is larger than *blue*.

Before we explore different techniques to handle such categorical data, let's create a new data frame to illustrate the problem:

```
>>> import pandas as pd
>>> df = pd.DataFrame([
...             ['green', 'M', 10.1, 'class1'],
...             ['red', 'L', 13.5, 'class2'],
...             ['blue', 'XL', 15.3, 'class1']])
>>> df.columns = ['color', 'size', 'price', 'classlabel']
>>> df
   color size  price classlabel
0  green    M   10.1     class1
1    red    L   13.5     class2
2   blue   XL   15.3     class1
```

As we can see in the preceding output, the newly created `DataFrame` contains a nominal feature (`color`), an ordinal feature (`size`), and a numerical feature (`price`) column. The class labels (assuming that we created a dataset for a supervised learning task) are stored in the last column. The learning algorithms for classification that we discuss in this book do not use ordinal information in class labels.

Mapping ordinal features

To make sure that the learning algorithm interprets the ordinal features correctly, we need to convert the categorical string values into integers. Unfortunately, there is no convenient function that can automatically derive the correct order of the labels of our `size` feature. Thus, we have to define the mapping manually. In the following simple example, let's assume that we know the difference between features, for example, $XL = L + 1 = M + 2$.

```
>>> size_mapping = {
...                 'XL': 3,
...                 'L': 2,
...                 'M': 1}
>>> df['size'] = df['size'].map(size_mapping)
>>> df
   color  size  price classlabel
0  green     1   10.1     class1
1    red     2   13.5     class2
2   blue     3   15.3     class1
```

If we want to transform the integer values back to the original string representation at a later stage, we can simply define a reverse-mapping dictionary `inv_size_mapping = {v: k for k, v in size_mapping.items()}` that can then be used via the pandas' `map` method on the transformed feature column similar to the `size_mapping` dictionary that we used previously.

Encoding class labels

Many machine learning libraries require that class labels are encoded as integer values. Although most estimators for classification in scikit-learn convert class labels to integers internally, it is considered good practice to provide class labels as integer arrays to avoid technical glitches. To encode the class labels, we can use an approach similar to the mapping of ordinal features discussed previously. We need to remember that class labels are *not* ordinal, and it doesn't matter which integer number we assign to a particular string-label. Thus, we can simply enumerate the class labels starting at 0:

```
>>> import numpy as np
>>> class_mapping = {label:idx for idx,label in
...                  enumerate(np.unique(df['classlabel']))}
>>> class_mapping
{'class1': 0, 'class2': 1}
```

Next we can use the mapping dictionary to transform the class labels into integers:

```
>>> df['classlabel'] = df['classlabel'].map(class_mapping)
>>> df
   color  size  price  classlabel
0  green     1   10.1           0
1    red     2   13.5           1
2   blue     3   15.3           0
```

We can reverse the key-value pairs in the mapping dictionary as follows to map the converted class labels back to the original string representation:

```
>>> inv_class_mapping = {v: k for k, v in class_mapping.items()}
>>> df['classlabel'] = df['classlabel'].map(inv_class_mapping)
>>> df
   color  size  price classlabel
0  green     1   10.1     class1
1    red     2   13.5     class2
2   blue     3   15.3     class1
```

Alternatively, there is a convenient `LabelEncoder` class directly implemented in scikit-learn to achieve the same:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> class_le = LabelEncoder()
>>> y = class_le.fit_transform(df['classlabel'].values)
>>> y
array([0, 1, 0])
```

Note that the `fit_transform` method is just a shortcut for calling `fit` and `transform` separately, and we can use the `inverse_transform` method to transform the integer class labels back into their original string representation:

```
>>> class_le.inverse_transform(y)
array(['class1', 'class2', 'class1'], dtype=object)
```

Performing one-hot encoding on nominal features

In the previous section, we used a simple dictionary-mapping approach to convert the ordinal size feature into integers. Since scikit-learn's estimators treat class labels without any order, we used the convenient `LabelEncoder` class to encode the string labels into integers. It may appear that we could use a similar approach to transform the nominal `color` column of our dataset, as follows:

```
>>> X = df[['color', 'size', 'price']].values
>>> color_le = LabelEncoder()
>>> X[:, 0] = color_le.fit_transform(X[:, 0])
>>> X
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

After executing the preceding code, the first column of the NumPy array `X` now holds the new `color` values, which are encoded as follows:

- blue → 0
- green → 1
- red → 2

If we stop at this point and feed the array to our classifier, we will make one of the most common mistakes in dealing with categorical data. Can you spot the problem? Although the color values don't come in any particular order, a learning algorithm will now assume that *green* is larger than *blue*, and *red* is larger than *green*. Although this assumption is incorrect, the algorithm could still produce useful results. However, those results would not be optimal.

A common workaround for this problem is to use a technique called **one-hot encoding**. The idea behind this approach is to create a new **dummy feature** for each unique value in the nominal feature column. Here, we would convert the `color` feature into three new features: `blue`, `green`, and `red`. Binary values can then be used to indicate the particular color of a sample; for example, a blue sample can be encoded as `blue=1`, `green=0`, `red=0`. To perform this transformation, we can use the `OneHotEncoder` that is implemented in the `scikit-learn.preprocessing` module:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> ohe = OneHotEncoder(categorical_features=[0])
>>> ohe.fit_transform(X).toarray()
array([[  0. ,   1. ,   0. ,   1. ,  10.1],
       [  0. ,   0. ,   1. ,   2. ,  13.5],
       [  1. ,   0. ,   0. ,   3. ,  15.3]])
```

When we initialized the `OneHotEncoder`, we defined the column position of the variable that we want to transform via the `categorical_features` parameter (note that `color` is the first column in the feature matrix `X`). By default, the `OneHotEncoder` returns a sparse matrix when we use the `transform` method, and we converted the sparse matrix representation into a regular (*dense*) NumPy array for the purposes of visualization via the `toarray` method. Sparse matrices are simply a more efficient way of storing large datasets, and one that is supported by many scikit-learn functions, which is especially useful if it contains a lot of zeros. To omit the `toarray` step, we could initialize the encoder as `OneHotEncoder(…, sparse=False)` to return a regular NumPy array.

An even more convenient way to create those dummy features via one-hot encoding is to use the `get_dummies` method implemented in pandas. Applied on a `DataFrame`, the `get_dummies` method will only convert string columns and leave all other columns unchanged:

```
>>> pd.get_dummies(df[['price', 'color', 'size']])
   price  size  color_blue  color_green  color_red
0   10.1     1           0            1          0
1   13.5     2           0            0          1
2   15.3     3           1            0          0
```

Partitioning a dataset in training and test sets

We briefly introduced the concept of partitioning a dataset into separate datasets for training and testing in *Chapter 1*, *Giving Computers the Ability to Learn from Data*, and *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*. Remember that the test set can be understood as the *ultimate test* of our model before we let it loose on the real world. In this section, we will prepare a new dataset, the **Wine** dataset. After we have preprocessed the dataset, we will explore different techniques for feature selection to reduce the dimensionality of a dataset.

The Wine dataset is another open-source dataset that is available from the UCI machine learning repository (`https://archive.ics.uci.edu/ml/datasets/Wine`); it consists of 178 wine samples with 13 features describing their different chemical properties.

Using the pandas library, we will directly read in the open source Wine dataset from the UCI machine learning repository:

```
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/wine/wine.data', header=None)
```

```
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash', 'Magnesium',
...                    'Total phenols', 'Flavanoids',
...                    'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>>  print('Class labels', np.unique(df_wine['Class label']))
Class labels [1 2 3]
>>> df_wine.head()
```

The 13 different features in the **Wine** dataset, describing the chemical properties of the 178 wine samples, are listed in the following table:

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

The samples belong to one of three different classes, 1, 2, and 3, which refer to the three different types of grapes that have been grown in different regions in Italy.

A convenient way to randomly partition this dataset into a separate *test* and *training* dataset is to use the `train_test_split` function from scikit-learn's `cross_validation` submodule:

```
>>> from sklearn.cross_validation import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...         train_test_split(X, y, test_size=0.3, random_state=0)
```

First, we assigned the NumPy array representation of feature columns 1-13 to the variable `X`, and we assigned the class labels from the first column to the variable `y`. Then, we used the `train_test_split` function to randomly split `X` and `y` into separate training and test datasets. By setting `test_size=0.3` we assigned 30 percent of the wine samples to `X_test` and `y_test`, and the remaining 70 percent of the samples were assigned to `X_train` and `y_train`, respectively.

If we are dividing a dataset into training and test datasets, we have to keep in mind that we are withholding valuable information that the learning algorithm could benefit from. Thus, we don't want to allocate too much information to the test set. However, the smaller the test set, the more inaccurate the estimation of the generalization error. Dividing a dataset into training and test sets is all about balancing this trade-off. In practice, the most commonly used splits are 60:40, 70:30, or 80:20, depending on the size of the initial dataset. However, for large datasets, 90:10 or 99:1 splits into training and test subsets are also common and appropriate. Instead of discarding the allocated test data after model training and evaluation, it is a good idea to retrain a classifier on the entire dataset for optimal performance.

Bringing features onto the same scale

Feature scaling is a crucial step in our **preprocessing** pipeline that can easily be forgotten. Decision trees and random forests are one of the very few machine learning algorithms where we don't need to worry about feature scaling. However, the majority of machine learning and optimization algorithms behave much better if features are on the same scale, as we saw in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, when we implemented the **gradient descent** optimization algorithm.

The importance of feature scaling can be illustrated by a simple example. Let's assume that we have two features where one feature is measured on a scale from 1 to 10 and the second feature is measured on a scale from 1 to 100,000. When we think of the squared error function in **Adaline** in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, it is intuitive to say that the algorithm will mostly be busy optimizing the weights according to the larger errors in the second feature. Another example is the **k-nearest neighbors** (**KNN**) algorithm with a Euclidean distance measure; the computed distances between samples will be dominated by the second feature axis.

Now, there are two common approaches to bringing different features onto the same scale: **normalization** and **standardization**. Those terms are often used quite loosely in different fields, and the meaning has to be derived from the context. Most often, normalization refers to the rescaling of the features to a range of [0, 1], which is a special case of min-max scaling. To normalize our data, we can simply apply the min-max scaling to each feature column, where the new value $x_{norm}^{(i)}$ of a sample $x^{(i)}$ can be calculated as follows:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{\min}}{x_{\max} - x_{\min}}$$

Here, $x^{(i)}$ is a particular sample, $x_{\min}$ is the smallest value in a feature column, and $x_{\max}$ the largest value, respectively.

The min-max scaling procedure is implemented in scikit-learn and can be used as follows:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> mms = MinMaxScaler()
>>> X_train_norm = mms.fit_transform(X_train)
>>> X_test_norm = mms.transform(X_test)
```

Although normalization via min-max scaling is a commonly used technique that is useful when we need values in a bounded interval, standardization can be more practical for many machine learning algorithms. The reason is that many linear models, such as the logistic regression and SVM that we remember from *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, initialize the weights to 0 or small random values close to 0. Using standardization, we center the feature columns at mean 0 with standard deviation 1 so that the feature columns take the form of a normal distribution, which makes it easier to learn the weights. Furthermore, standardization maintains useful information about outliers and makes the algorithm less sensitive to them in contrast to min-max scaling, which scales the data to a limited range of values.

The procedure of standardization can be expressed by the following equation:

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

Here, μ_x is the sample mean of a particular feature column and σ_x the corresponding standard deviation, respectively.

The following table illustrates the difference between the two commonly used feature scaling techniques, standardization and normalization on a simple sample dataset consisting of numbers 0 to 5:

input	standardized	normalized
0.0	-1.336306	0.0
1.0	-0.801784	0.2
2.0	-0.267261	0.4
3.0	0.267261	0.6
4.0	0.801784	0.8
5.0	1.336306	1.0

Similar to `MinMaxScaler`, scikit-learn also implements a class for standardization:

```
>>> from sklearn.preprocessing import StandardScaler
>>> stdsc = StandardScaler()
>>> X_train_std = stdsc.fit_transform(X_train)
>>> X_test_std = stdsc.transform(X_test)
```

Again, it is also important to highlight that we fit the `StandardScaler` only once on the training data and use those parameters to transform the test set or any new data point.

Selecting meaningful features

If we notice that a model performs much better on a training dataset than on the test dataset, this observation is a strong indicator for **overfitting**. Overfitting means that model fits the parameters too closely to the particular observations in the training dataset but does not generalize well to real data—we say that the model has a *high variance*. A reason for overfitting is that our model is too complex for the given training data and common solutions to reduce the generalization error are listed as follows:

- Collect more training data
- Introduce a penalty for complexity via regularization
- Choose a simpler model with fewer parameters
- Reduce the dimensionality of the data

Collecting more training data is often not applicable. In the next chapter, we will learn about a useful technique to check whether more training data is helpful at all. In the following sections and subsections, we will look at common ways to reduce overfitting by regularization and dimensionality reduction via feature selection.

Sparse solutions with L1 regularization

We recall from *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, that **L2 regularization** is one approach to reduce the complexity of a model by penalizing large individual weights, where we defined the L2 norm of our weight vector w as follows:

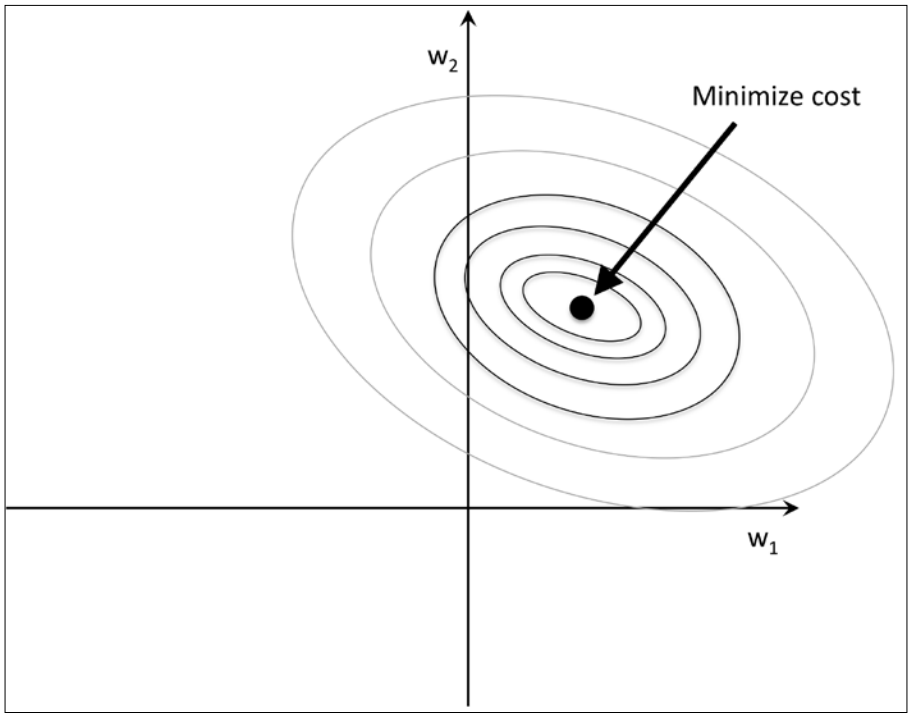
$$L2: \|\boldsymbol{w}\|_2^2 = \sum_{j=1}^{m} w_j^2$$

Another approach to reduce the model complexity is the related **L1 regularization**:

$$L1: \|\boldsymbol{w}\|_1 = \sum_{j=1}^{m} |w_j|$$

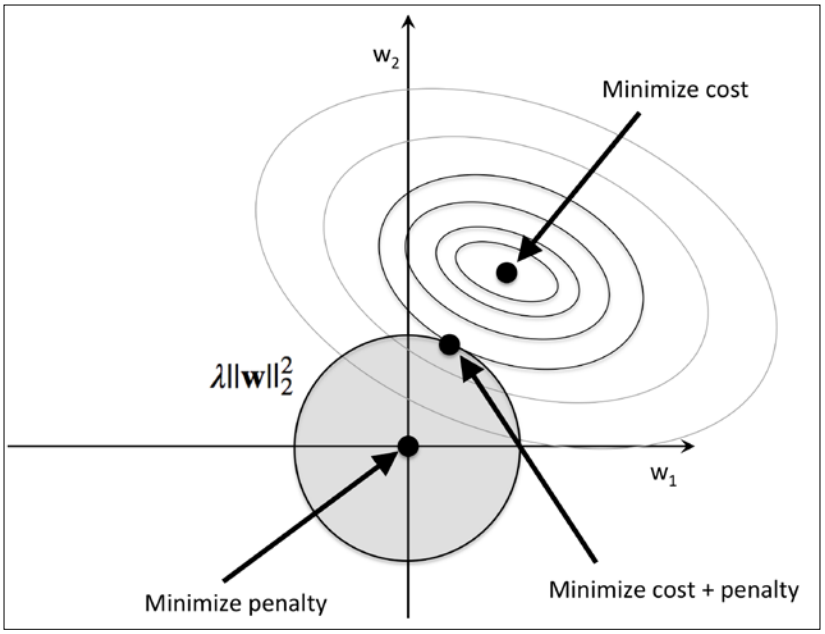
Here, we simply replaced the square of the weights by the sum of the absolute values of the weights. In contrast to L2 regularization, L1 regularization yields sparse feature vectors; most feature weights will be zero. Sparsity can be useful in practice if we have a high-dimensional dataset with many features that are irrelevant, especially cases where we have more irrelevant dimensions than samples. In this sense, L1 regularization can be understood as a technique for feature selection.

To better understand how L1 regularization encourages sparsity, let's take a step back and take a look at a geometrical interpretation of regularization. Let's plot the contours of a convex cost function for two weight coefficients w_1 and w_2. Here, we will consider the **sum of the squared errors** (**SSE**) cost function that we used for Adaline in *Chapter 2*, *Training Machine Learning Algorithms for Classification*, since it is symmetrical and easier to draw than the cost function of logistic regression; however, the same concepts apply to the latter. Remember that our goal is to find the combination of weight coefficients that minimize the cost function for the training data, as shown in the following figure (the point in the middle of the ellipses):



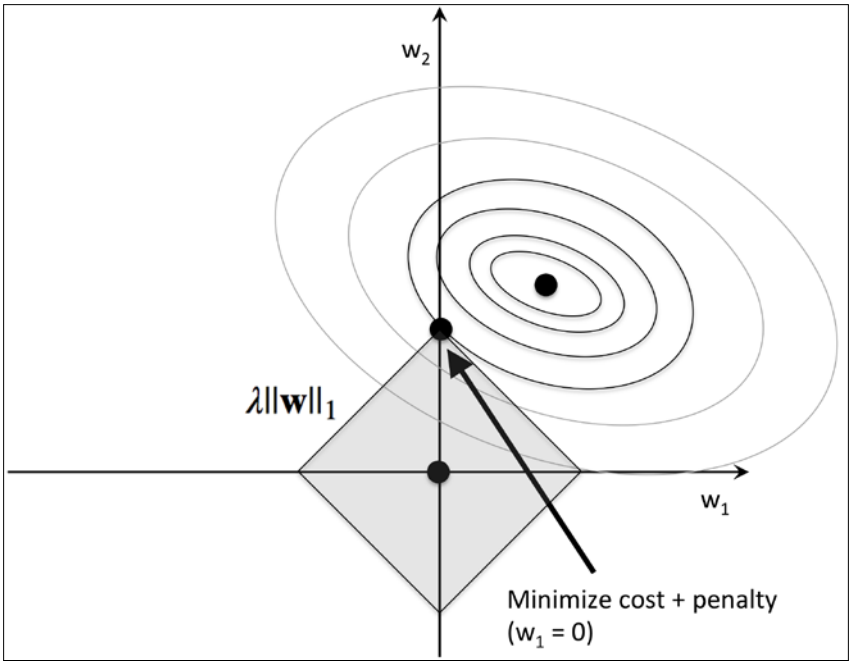
Now, we can think of regularization as adding a penalty term to the cost function to encourage smaller weights; or, in other words, we penalize large weights.

Thus, by increasing the regularization strength via the regularization parameter λ, we shrink the weights towards zero and decrease the dependence of our model on the training data. Let's illustrate this concept in the following figure for the L2 penalty term.



The quadratic L2 regularization term is represented by the shaded ball. Here, our weight coefficients cannot exceed our regularization *budget*—the combination of the weight coefficients cannot fall outside the shaded area. On the other hand, we still want to minimize the cost function. Under the penalty constraint, our best effort is to choose the point where the L2 ball intersects with the contours of the unpenalized cost function. The larger the value of the regularization parameter λ gets, the faster the penalized cost function grows, which leads to a narrower L2 ball. For example, if we increase the regularization parameter towards infinity, the weight coefficients will become effectively zero, denoted by the center of the L2 ball. To summarize the main message of the example: our goal is to minimize the sum of the unpenalized cost function plus the penalty term, which can be understood as adding bias and preferring a simpler model to reduce the variance in the absence of sufficient training data to fit the model.

Now let's discuss L1 regularization and sparsity. The main concept behind L1 regularization is similar to what we have discussed here. However, since the L1 penalty is the sum of the absolute weight coefficients (remember that the L2 term is quadratic), we can represent it as a diamond shape *budget*, as shown in the following figure:



In the preceding figure, we can see that the contour of the cost function touches the L1 diamond at $w_1 = 0$. Since the contours of an L1 regularized system are sharp, it is more likely that the optimum—that is, the intersection between the ellipses of the cost function and the boundary of the L1 diamond—is located on the axes, which encourages sparsity. The mathematical details of why L1 regularization can lead to sparse solutions are beyond the scope of this book. If you are interested, an excellent section on L2 versus L1 regularization can be found in section 3.4 of *The Elements of Statistical Learning*, *Trevor Hastie, Robert Tibshirani, and Jerome Friedman*, *Springer*.

For regularized models in scikit-learn that support L1 regularization, we can simply set the `penalty` parameter to `'l1'` to yield the sparse solution:

```
>>> from sklearn.linear_model import LogisticRegression
>>> LogisticRegression(penalty='l1')
```

Applied to the standardized Wine data, the L1 regularized logistic regression would yield the following sparse solution:

```
>>> lr = LogisticRegression(penalty='l1', C=0.1)
>>> lr.fit(X_train_std, y_train)
>>> print('Training accuracy:', lr.score(X_train_std, y_train))
Training accuracy: 0.983870967742
>>> print('Test accuracy:', lr.score(X_test_std, y_test))
Test accuracy: 0.981481481481
```

Both training and test accuracies (both 98 percent) do not indicate any overfitting of our model. When we access the intercept terms via the `lr.intercept_` attribute, we can see that the array returns three values:

```
>>> lr.intercept_
array([-0.38379237, -0.1580855 , -0.70047966])
```

Since we the fit the `LogisticRegression` object on a multiclass dataset, it uses the **One-vs-Rest** (**OvR**) approach by default where the first intercept belongs to the model that fits class 1 versus class 2 and 3; the second value is the intercept of the model that fits class 2 versus class 1 and 3; and the third value is the intercept of the model that fits class 3 versus class 1 and 2, respectively:

```
>>> lr.coef_
array([[ 0.280, 0.000, 0.000, -0.0282, 0.000,
         0.000, 0.710, 0.000, 0.000, 0.000,
         0.000, 0.000, 1.236],
       [-0.644, -0.0688 , -0.0572, 0.000, 0.000,
         0.000, 0.000, 0.000, 0.000, -0.927,
         0.060, 0.000, -0.371],
       [ 0.000, 0.061, 0.000, 0.000, 0.000,
         0.000, -0.637, 0.000, 0.000, 0.499,
        -0.358, -0.570, 0.000
       ]])
```

The weight array that we accessed via the `lr.coef_` attribute contains three rows of weight coefficients, one weight vector for each class. Each row consists of 13 weights where each weight is multiplied by the respective feature in the 13-dimensional Wine dataset to calculate the net input:

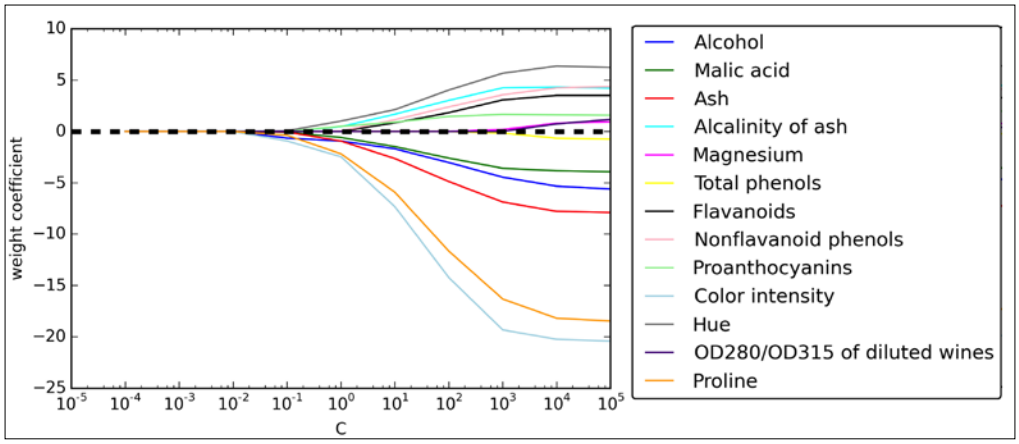
$$z = w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \boldsymbol{w}^T \boldsymbol{x}$$

We notice that the weight vectors are sparse, which means that they only have a few non-zero entries. As a result of the L1 regularization, which serves as a method for feature selection, we just trained a model that is robust to the potentially irrelevant features in this dataset.

Lastly, let's plot the regularization path, which is the weight coefficients of the different features for different regularization strengths:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> colors = ['blue', 'green', 'red', 'cyan',
...           'magenta', 'yellow', 'black',
...           'pink', 'lightgreen', 'lightblue',
...           'gray', 'indigo', 'orange']
>>> weights, params = [], []
>>> for c in np.arange(-4, 6):
...     lr = LogisticRegression(penalty='l1',
...                             C=10**c,
...                             random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
>>> weights = np.array(weights)
>>> for column, color in zip(range(weights.shape[1]), colors):
...     plt.plot(params, weights[:, column],
...              label=df_wine.columns[column+1],
...              color=color)
>>> plt.axhline(0, color='black', linestyle='--', linewidth=3)
>>> plt.xlim([10**(-5), 10**5])
>>> plt.ylabel('weight coefficient')
>>> plt.xlabel('C')
>>> plt.xscale('log')
>>> plt.legend(loc='upper left')
>>> ax.legend(loc='upper center',
...           bbox_to_anchor=(1.38, 1.03),
...           ncol=1, fancybox=True)
>>> plt.show()
```

The resulting plot provides us with further insights about the behavior of L1 regularization. As we can see, all features weights will be zero if we penalize the model with a strong regularization parameter ($C < 0.1$); C is the inverse of the regularization parameter λ.



Sequential feature selection algorithms

An alternative way to reduce the complexity of the model and avoid overfitting is **dimensionality reduction** via feature selection, which is especially useful for unregularized models. There are two main categories of dimensionality reduction techniques: **feature selection** and **feature extraction**. Using feature selection, we select a subset of the original features. In feature extraction, we derive information from the feature set to construct a new feature subspace. In this section, we will take a look at a classic family of feature selection algorithms. In the next chapter, *Chapter 5*, *Compressing Data via Dimensionality Reduction*, we will learn about different feature extraction techniques to compress a dataset onto a lower dimensional feature subspace.

Sequential feature selection algorithms are a family of greedy search algorithms that are used to reduce an initial *d-dimensional* feature space to a *k-dimensional* feature subspace where $k < d$. The motivation behind feature selection algorithms is to automatically select a subset of features that are most relevant to the problem to improve computational efficiency or reduce the generalization error of the model by removing irrelevant features or noise, which can be useful for algorithms that don't support regularization. A classic sequential feature selection algorithm is **Sequential Backward Selection** (**SBS**), which aims to reduce the dimensionality of the initial feature subspace with a minimum decay in performance of the classifier to improve upon computational efficiency. In certain cases, SBS can even improve the predictive power of the model if a model suffers from overfitting.

Greedy algorithms make locally optimal choices at each stage of a combinatorial search problem and generally yield a suboptimal solution to the problem in contrast to exhaustive search algorithms, which evaluate all possible combinations and are guaranteed to find the optimal solution. However, in practice, an exhaustive search is often computationally not feasible, whereas greedy algorithms allow for a less complex, computationally more efficient solution.

The idea behind the SBS algorithm is quite simple: SBS sequentially removes features from the full feature subset until the new feature subspace contains the desired number of features. In order to determine which feature is to be removed at each stage, we need to define criterion function J that we want to minimize. The criterion calculated by the criterion function can simply be the difference in performance of the classifier after and before the removal of a particular feature. Then the feature to be removed at each stage can simply be defined as the feature that maximizes this criterion; or, in more intuitive terms, at each stage we eliminate the feature that causes the least performance loss after removal. Based on the preceding definition of SBS, we can outline the algorithm in 4 simple steps:

1. Initialize the algorithm with $k = d$, where *d* is the dimensionality of the full feature space $\mathbf{X}_d$.
2. Determine the feature x^- that maximizes the criterion $x^- = \arg\max J(\boldsymbol{X}_k - \boldsymbol{x})$ where $\boldsymbol{x} \in \boldsymbol{X}_k$.
3. Remove the feature x^- from the feature set: $\boldsymbol{X}_{k-1} := \boldsymbol{X}_k - \boldsymbol{x}^-; k := k-1$.
4. Terminate if *k* equals the number of desired features, if not, go to step 2.

You can find a detailed evaluation of several sequential feature algorithms in *Comparative Study of Techniques for Large Scale Feature Selection, F. Ferri, P. Pudil, M. Hatef, and J. Kittler. Comparative study of techniques for large-scale feature selection. Pattern Recognition in Practice IV, pages 403–413, 1994.*

Unfortunately, the SBS algorithm is not implemented in scikit-learn, yet. But since it is so simple, let's go ahead and implement it in Python from scratch:

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.cross_validation import train_test_split
from sklearn.metrics import accuracy_score
```

```
class SBS():
    def __init__(self, estimator, k_features,
        scoring=accuracy_score,
        test_size=0.25, random_state=1):
        self.scoring = scoring
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state

    def fit(self, X, y):
        X_train, X_test, y_train, y_test = \
                train_test_split(X, y, test_size=self.test_size,
                                 random_state=self.random_state)

        dim = X_train.shape[1]
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train,
                                 X_test, y_test, self.indices_)
        self.scores_ = [score]

        while dim > self.k_features:
            scores = []
            subsets = []

            for p in combinations(self.indices_, r=dim-1):
                score = self._calc_score(X_train, y_train,
                                         X_test, y_test, p)
                scores.append(score)
                subsets.append(p)

            best = np.argmax(scores)
            self.indices_ = subsets[best]
            self.subsets_.append(self.indices_)
            dim -= 1

            self.scores_.append(scores[best])
        self.k_score_ = self.scores_[-1]

        return self

    def transform(self, X):
        return X[:, self.indices_]
```

```
    def _calc_score(self, X_train, y_train,
                            X_test, y_test, indices):
        self.estimator.fit(X_train[:, indices], y_train)
        y_pred = self.estimator.predict(X_test[:, indices])
        score = self.scoring(y_test, y_pred)
        return score
```

In the preceding implementation, we defined the `k_features` parameter to specify the desired number of features we want to return. By default, we use the `accuracy_score` from scikit-learn to evaluate the performance of a model and estimator for classification on the feature subsets. Inside the `while` loop of the `fit` method, the feature subsets created by the `itertools.combination` function are evaluated and reduced until the feature subset has the desired dimensionality. In each iteration, the accuracy score of the best subset is collected in a list `self.scores_` based on the internally created test dataset `X_test`. We will use those scores later to evaluate the results. The column indices of the final feature subset are assigned to `self.indices_`, which we can use via the `transform` method to return a new data array with the selected feature columns. Note that, instead of calculating the criterion explicitly inside the `fit` method, we simply removed the feature that is not contained in the best performing feature subset.

Now, let's see our SBS implementation in action using the KNN classifier from scikit-learn:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> import matplotlib.pyplot as plt
>>> knn = KNeighborsClassifier(n_neighbors=2)
>>> sbs = SBS(knn, k_features=1)
>>> sbs.fit(X_train_std, y_train)
```

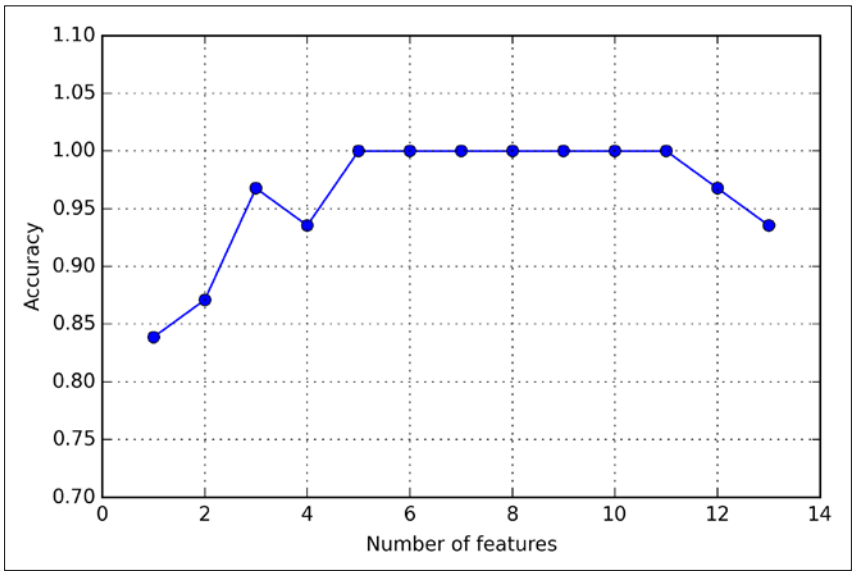
Although our SBS implementation already splits the dataset into a test and training dataset inside the `fit` function, we still fed the training dataset `X_train` to the algorithm. The SBS `fit` method will then create new training-subsets for testing (validation) and training, which is why this test set is also called **validation dataset**. *This approach is necessary to prevent our original test set becoming part of the training data.*

Remember that our SBS algorithm collects the scores of the best feature subset at each stage, so let's move on to the more exciting part of our implementation and plot the classification accuracy of the KNN classifier that was calculated on the validation dataset. The code is as follows:

```
>>> k_feat = [len(k) for k in sbs.subsets_]
>>> plt.plot(k_feat, sbs.scores_, marker='o')
>>> plt.ylim([0.7, 1.1])
```

```
>>> plt.ylabel('Accuracy')
>>> plt.xlabel('Number of features')
>>> plt.grid()
>>> plt.show()
```

As we can see in the following plot, the accuracy of the KNN classifier improved on the validation dataset as we reduced the number of features, which is likely due to a decrease of **the curse of dimensionality** that we discussed in the context of the KNN algorithm in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*. Also, we can see in the following plot that the classifier achieved 100 percent accuracy for *k={5, 6, 7, 8, 9, 10}*:



To satisfy our own curiosity, let's see what those five features are that yielded such a good performance on the validation dataset:

```
>>> k5 = list(sbs.subsets_[8])
>>> print(df_wine.columns[1:][k5])
Index(['Alcohol', 'Malic acid', 'Alcalinity of ash', 'Hue',
'Proline'], dtype='object')
```

Using the preceding code, we obtained the column indices of the 5-feature subset from the 9th position in the `sbs.subsets_` attribute and returned the corresponding feature names from the column-index of the pandas Wine `DataFrame`.

Next let's evaluate the performance of the KNN classifier on the original test set:

```
>>> knn.fit(X_train_std, y_train)
>>> print('Training accuracy:', knn.score(X_train_std, y_train))
Training accuracy: 0.983870967742
>>> print('Test accuracy:', knn.score(X_test_std, y_test))
Test accuracy: 0.944444444444
```

In the preceding code, we used the complete feature set and obtained ~98.4 percent accuracy on the training dataset. However, the accuracy on the test dataset was slightly lower (~94.4 percent), which is an indicator of a slight degree of overfitting. Now let's use the selected 5-feature subset and see how well KNN performs:

```
>>> knn.fit(X_train_std[:, k5], y_train)
>>> print('Training accuracy:',
...       knn.score(X_train_std[:, k5], y_train))
Training accuracy: 0.959677419355
>>> print('Test accuracy:',
...       knn.score(X_test_std[:, k5], y_test))
Test accuracy: 0.962962962963
```

Using fewer than half of the original features in the Wine dataset, the prediction accuracy on the test set improved by almost 2 percent. Also, we reduced overfitting, which we can tell from the small gap between test (~96.3 percent) and training (~96.0 percent) accuracy.

Feature selection algorithms in scikit-learn

There are many more feature selection algorithms available via scikit-learn. Those include recursive backward elimination based on feature weights, tree-based methods to select features by importance, and univariate statistical tests. A comprehensive discussion of the different feature selection methods is beyond the scope of this book, but a good summary with illustrative examples can be found at `http://scikit-learn.org/stable/modules/feature_selection.html`.

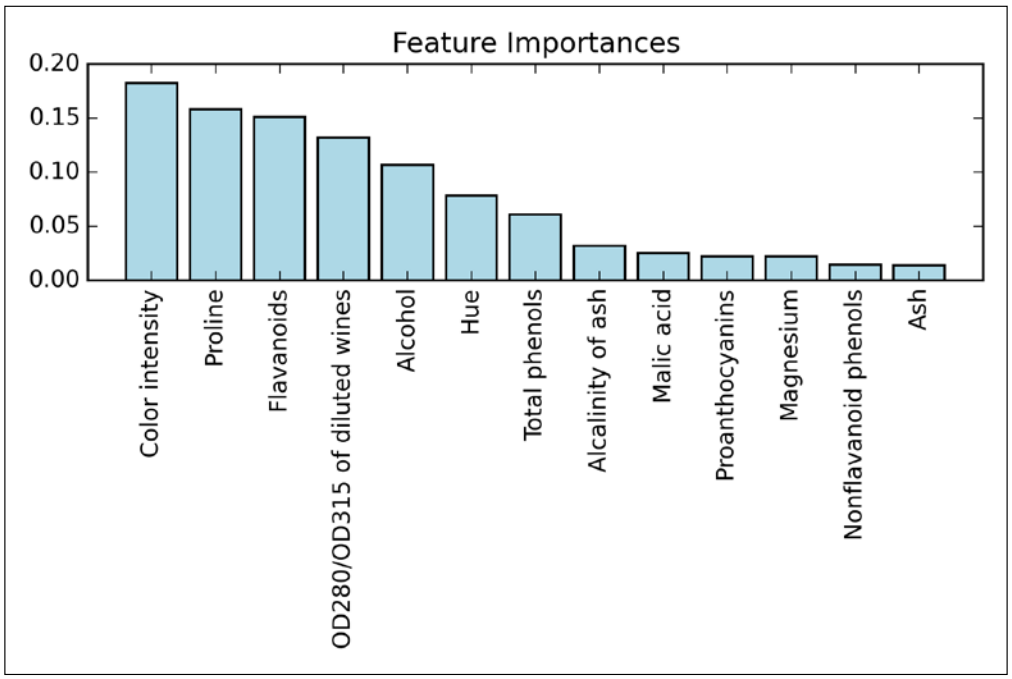
Assessing feature importance with random forests

In the previous sections, you learned how to use L1 regularization to zero out irrelevant features via logistic regression and use the SBS algorithm for feature selection. Another useful approach to select relevant features from a dataset is to use a random forest, an ensemble technique that we introduced in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*. Using a random forest, we can measure feature importance as the averaged impurity decrease computed from all decision trees in the forest without making any assumptions whether our data is linearly separable or not. Conveniently, the random forest implementation in scikit-learn already collects feature importances for us so that we can access them via the `feature_importances_` attribute after fitting a `RandomForestClassifier`. By executing the following code, we will now train a forest of 10,000 trees on the Wine dataset and rank the 13 features by their respective importance measures. Remember (from our discussion in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*) that we don't need to use standardized or normalized tree-based models. The code is as follows:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> feat_labels = df_wine.columns[1:]
>>> forest = RandomForestClassifier(n_estimators=10000,
...                                 random_state=0,
...                                 n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> importances = forest.feature_importances_
>>> indices = np.argsort(importances)[::-1]
>>> for f in range(X_train.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                             feat_labels[indices[f]],
...                             importances[indices[f]]))
1) Color intensity                0.182483
2) Proline                        0.158610
3) Flavanoids                     0.150948
4) OD280/OD315 of diluted wines   0.131987
5) Alcohol                        0.106589
6) Hue                            0.078243
7) Total phenols                  0.060718
8) Alcalinity of ash              0.032033
9) Malic acid                     0.025400
10) Proanthocyanins               0.022351
11) Magnesium                     0.022078
```

```
12) Nonflavanoid phenols           0.014645
13) Ash                            0.013916
>>> plt.title('Feature Importances')
>>> plt.bar(range(X_train.shape[1]),
...         importances[indices],
...         color='lightblue',
...         align='center')
>>> plt.xticks(range(X_train.shape[1]),
...            feat_labels[indices], rotation=90)
>>> plt.xlim([-1, X_train.shape[1]])
>>> plt.tight_layout()
>>> plt.show()
```

After executing the preceding code, we created a plot that ranks the different features in the Wine dataset by their relative importance; note that the feature importances are normalized so that they sum up to 1.0.



We can conclude that the color intensity of wine is the most discriminative feature in the dataset based on the average impurity decrease in the 10,000 decision trees. Interestingly, the three top-ranked features in the preceding plot are also among the top five features in the selection by the SBS algorithm that we implemented in the previous section. However, as far as interpretability is concerned, the random forest technique comes with an important *gotcha* that is worth mentioning. For instance, if two or more features are highly correlated, one feature may be ranked very highly while the information of the other feature(s) may not be fully captured. On the other hand, we don't need to be concerned about this problem if we are merely interested in the predictive performance of a model rather than the interpretation of feature importances. To conclude this section about feature importances and random forests, it is worth mentioning that scikit-learn also implements a `transform` method that selects features based on a user-specified threshold after model fitting, which is useful if we want to use the `RandomForestClassifier` as a feature selector and intermediate step in a scikit-learn pipeline, which allows us to connect different preprocessing steps with an estimator, as we will see in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*. For example, we could set the threshold to 0.15 to reduce the dataset to the 3 most important features, **Color intensity**, **Proline**, and **Flavonoids** using the following code:

```
>>> X_selected = forest.transform(X_train, threshold=0.15)
>>> X_selected.shape
(124, 3)
```

Summary of Module 4 Chapter 4

Ankita Thakur

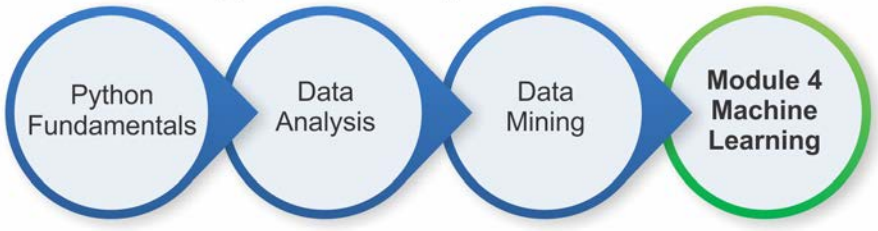
Your Course Guide

We started this chapter by looking at useful techniques to make sure that we handle missing data correctly. Before we feed data to a machine learning algorithm, we also have to make sure that we encode categorical variables correctly, and we have seen how we can map ordinal and nominal features values to integer representations.

Moreover, we briefly discussed L1 regularization, which can help us to avoid overfitting by reducing the complexity of a model. As an alternative approach for removing irrelevant features, we used a sequential feature selection algorithm to select meaningful features from a dataset.

In the next chapter, you will learn about yet another useful approach to dimensionality reduction: feature extraction. It allows us to compress features onto a lower dimensional subspace rather than removing features entirely as in feature selection.

Your Progress through the Course So Far



5
Compressing Data via Dimensionality Reduction

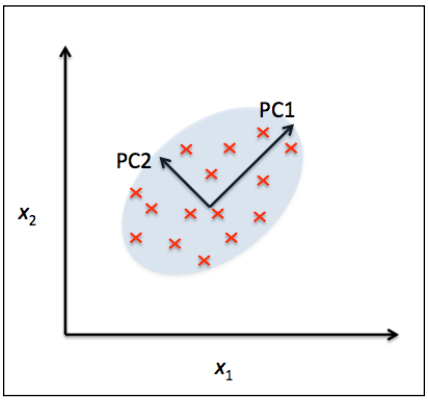
In *Chapter 4*, *Building Good Training Sets – Data Preprocessing*, you learned about the different approaches for reducing the dimensionality of a dataset using different feature selection techniques. An alternative approach to feature selection for dimensionality reduction is *feature extraction*. In this chapter, you will learn about three fundamental techniques that will help us to summarize the information content of a dataset by transforming it onto a new feature subspace of lower dimensionality than the original one. Data compression is an important topic in machine learning, and it helps us to store and analyze the increasing amounts of data that are produced and collected in the modern age of technology. In this chapter, we will cover the following topics:

- **Principal component analysis** (**PCA**) for unsupervised data compression
- **Linear Discriminant Analysis** (**LDA**) as a supervised dimensionality reduction technique for maximizing class separability
- Nonlinear dimensionality reduction via **kernel principal component analysis**

Unsupervised dimensionality reduction via principal component analysis

Similar to feature selection, we can use feature extraction to reduce the number of features in a dataset. However, while we maintained the original features when we used feature selection algorithms, such as *sequential backward selection*, we use feature extraction to transform or project the data onto a new feature space. In the context of dimensionality reduction, feature extraction can be understood as an approach to data compression with the goal of maintaining most of the relevant information. Feature extraction is typically used to improve computational efficiency but can also help to reduce the *curse of dimensionality*—especially if we are working with nonregularized models.

Principal component analysis (**PCA**) is an unsupervised linear transformation technique that is widely used across different fields, most prominently for dimensionality reduction. Other popular applications of PCA include exploratory data analyses and de-noising of signals in stock market trading, and the analysis of genome data and gene expression levels in the field of bioinformatics. PCA helps us to identify patterns in data based on the correlation between features. In a nutshell, PCA aims to find the directions of maximum variance in high-dimensional data and projects it onto a new subspace with equal or fewer dimensions that the original one. The orthogonal axes (principal components) of the new subspace can be interpreted as the directions of maximum variance given the constraint that the new feature axes are orthogonal to each other as illustrated in the following figure. Here, x_1 and x_2 are the original feature axes, and **PC1** and **PC2** are the principal components:



If we use PCA for dimensionality reduction, we construct a $d \times k$-dimensional transformation matrix $\boldsymbol{W}$ that allows us to map a sample vector $\boldsymbol{x}$ onto a new k-dimensional feature subspace that has fewer dimensions than the original d-dimensional feature space:

$$\boldsymbol{x} = \left[x_1, x_2, \ldots, x_d\right], \quad \boldsymbol{x} \in \mathbb{R}^{\boldsymbol{d}}$$

$$\downarrow \boldsymbol{xW}, \quad \boldsymbol{W} \in \mathbb{R}^{\boldsymbol{d} \times \boldsymbol{k}}$$

$$\mathbf{z} = \left[z_1, z_2, \ldots, z_k\right], \quad \mathbf{z} \in \mathbb{R}^{\boldsymbol{k}}$$

As a result of transforming the original d-dimensional data onto this new k-dimensional subspace (typically $k << d$), the first principal component will have the largest possible variance, and all consequent principal components will have the largest possible variance given that they are uncorrelated (orthogonal) to the other principal components. Note that the PCA directions are highly sensitive to data scaling, and we need to standardize the features *prior* to PCA if the features were measured on different scales and we want to assign equal importance to all features.

Before looking at the PCA algorithm for dimensionality reduction in more detail, let's summarize the approach in a few simple steps:

1. Standardize the d-dimensional dataset.
2. Construct the covariance matrix.
3. Decompose the covariance matrix into its eigenvectors and eigenvalues.
4. Select k eigenvectors that correspond to the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \leq d$).
5. Construct a projection matrix $\boldsymbol{W}$ from the "top" k eigenvectors.
6. Transform the d-dimensional input dataset $\boldsymbol{X}$ using the projection matrix $\boldsymbol{W}$ to obtain the new k-dimensional feature subspace.

Total and explained variance

In this subsection, we will tackle the first four steps of a principal component analysis: standardizing the data, constructing the covariance matrix, obtaining the eigenvalues and eigenvectors of the covariance matrix, and sorting the eigenvalues by decreasing order to rank the eigenvectors.

First, we will start by loading the *Wine* dataset that we have been working with in *Chapter 4*, *Building Good Training Sets – Data Preprocessing*:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/wine/wine.data', header=None)
```

Next, we will process the *Wine* data into separate training and test sets—using 70 percent and 30 percent of the data, respectively—and standardize it to unit variance.

```
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...              train_test_split(X, y,
...              test_size=0.3, random_state=0)
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

After completing the mandatory preprocessing steps by executing the preceding code, let's advance to the second step: constructing the covariance matrix. The symmetric $d \times d$-dimensional covariance matrix, where d is the number of dimensions in the dataset, stores the pairwise covariances between the different features. For example, the covariance between two features $\boldsymbol{x}_j$ and x_k on the population level can be calculated via the following equation:

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n}\left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$$

Here, μ_j and μ_k are the sample means of feature j and k, respectively. Note that the sample means are zero if we standardize the dataset. A positive covariance between two features indicates that the features increase or decrease together, whereas a negative covariance indicates that the features vary in opposite directions. For example, a covariance matrix of three features can then be written as (note that Σ stands for the Greek letter *sigma*, which is not to be confused with the *sum* symbol):

$$\sum = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

The eigenvectors of the covariance matrix represent the principal components (the directions of maximum variance), whereas the corresponding eigenvalues will define their magnitude. In the case of the *Wine* dataset, we would obtain 13 eigenvectors and eigenvalues from the 13×13-dimensional covariance matrix.

Now, let's obtain the eigenpairs of the covariance matrix. As we surely remember from our introductory linear algebra or calculus classes, an eigen vector $\boldsymbol{v}$ satisfies the following condition:

$$\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}$$

Here, λ is a scalar: the eigenvalue. Since the manual computation of eigenvectors and eigenvalues is a somewhat tedious and elaborate task, we will use the `linalg.eig` function from NumPy to obtain the eigenpairs of the *Wine* covariance matrix:

```
>>> import numpy as np
>>> cov_mat = np.cov(X_train_std.T)
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
>>> print('\nEigenvalues \n%s' % eigen_vals)
Eigenvalues
[ 4.8923083   2.46635032  1.42809973  1.01233462  0.84906459
0.60181514
0.52251546  0.08414846  0.33051429  0.29595018  0.16831254  0.21432212
0.2399553 ]
```

Using the `numpy.cov` function, we computed the covariance matrix of the standardized training dataset. Using the `linalg.eig` function, we performed the eigendecomposition that yielded a vector (`eigen_vals`) consisting of 13 eigenvalues and the corresponding eigenvectors stored as columns in a 13×13-dimensional matrix (`eigen_vecs`).

Although the `numpy.linalg.eig` function was designed to decompose nonsymmetric square matrices, you may find that it returns complex eigenvalues in certain cases.

A related function, `numpy.linalg.eigh`, has been implemented to decompose Hermetian matrices, which is a numerically more stable approach to work with symmetric matrices such as the covariance matrix; `numpy.linalg.eigh` always returns real eigenvalues.

Since we want to reduce the dimensionality of our dataset by compressing it onto a new feature subspace, we only select the subset of the eigenvectors (principal components) that contains most of the information (variance). Since the eigenvalues define the magnitude of the eigenvectors, we have to sort the eigenvalues by decreasing magnitude; we are interested in the top k eigenvectors based on the values of their corresponding eigenvalues. But before we collect those k most informative eigenvectors, let's plot the *variance explained ratios* of the eigenvalues.

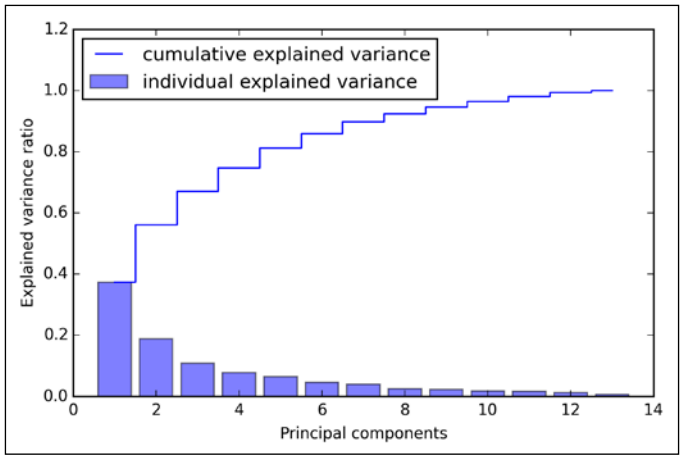
The variance explained ratio of an eigenvalue λ_j is simply the fraction of an eigenvalue λ_j and the total sum of the eigenvalues:

$$\frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$$

Using the NumPy `cumsum` function, we can then calculate the cumulative sum of explained variances, which we will plot via matplotlib's `step` function:

```
>>> tot = sum(eigen_vals)
>>> var_exp = [(i / tot) for i in
...            sorted(eigen_vals, reverse=True)]
>>> cum_var_exp = np.cumsum(var_exp)

>>> import matplotlib.pyplot as plt
>>> plt.bar(range(1,14), var_exp, alpha=0.5, align='center',
...         label='individual explained variance')
>>> plt.step(range(1,14), cum_var_exp, where='mid',
...         label='cumulative explained variance')
>>> plt.ylabel('Explained variance ratio')
>>> plt.xlabel('Principal components')
>>> plt.legend(loc='best')
>>> plt.show()
```

The resulting plot indicates that the first principal component alone accounts for 40 percent of the variance. Also, we can see that the first two principal components combined explain almost 60 percent of the variance in the data:



Although the explained variance plot reminds us of the feature importance that we computed in *Chapter 4*, *Building Good Training Sets – Data Preprocessing*, via random forests, we shall remind ourselves that PCA is an unsupervised method, which means that information about the class labels is ignored. Whereas a random forest uses the class membership information to compute the node impurities, variance measures the spread of values along a feature axis.

Feature transformation

After we have successfully decomposed the covariance matrix into eigenpairs, let's now proceed with the last three steps to transform the *Wine* dataset onto the new principal component axes. In this section, we will sort the eigenpairs by descending order of the eigenvalues, construct a projection matrix from the selected eigenvectors, and use the projection matrix to transform the data onto the lower-dimensional subspace.

We start by sorting the eigenpairs by decreasing order of the eigenvalues:

```
>>> eigen_pairs =[(np.abs(eigen_vals[i]),eigen_vecs[:,i])
...               for i inrange(len(eigen_vals))]
>>> eigen_pairs.sort(reverse=True)
```

Next, we collect the two eigenvectors that correspond to the two largest values to capture about 60 percent of the variance in this dataset. Note that we only chose two eigenvectors for the purpose of illustration, since we are going to plot the data via a two-dimensional scatter plot later in this subsection. In practice, the number of principal components has to be determined from a trade-off between computational efficiency and the performance of the classifier:

```
>>> w= np.hstack((eigen_pairs[0][1][:, np.newaxis],
...                eigen_pairs[1][1][:, np.newaxis]))
>>> print('Matrix W:\n',w)
Matrix W:
[[ 0.14669811  0.50417079]
[-0.24224554  0.24216889]
[-0.02993442  0.28698484]
[-0.25519002 -0.06468718]
[ 0.12079772  0.22995385]
[ 0.38934455  0.09363991]
[ 0.42326486  0.01088622]
[-0.30634956  0.01870216]
[ 0.30572219  0.03040352]
[-0.09869191  0.54527081]
```

```
[ 0.30032535 -0.27924322]
[ 0.36821154 -0.174365  ]
[ 0.29259713  0.36315461]]
```

By executing the preceding code, we have created a 13×2-dimensional projection matrix $\boldsymbol{W}$ from the top two eigenvectors. Using the projection matrix, we can now transform a sample $\boldsymbol{x}$ (represented as 1×13-dimensional row vector) onto the PCA subspace obtaining $\boldsymbol{x}'$, a now two-dimensional sample vector consisting of two new features:

$$\boldsymbol{x}' = \boldsymbol{x}\boldsymbol{W}$$

```
>>> X_train_std[0].dot(w)
array([ 2.59891628,  0.00484089])
```

Similarly, we can transform the entire 124×13-dimensional training dataset onto the two principal components by calculating the matrix dot product:

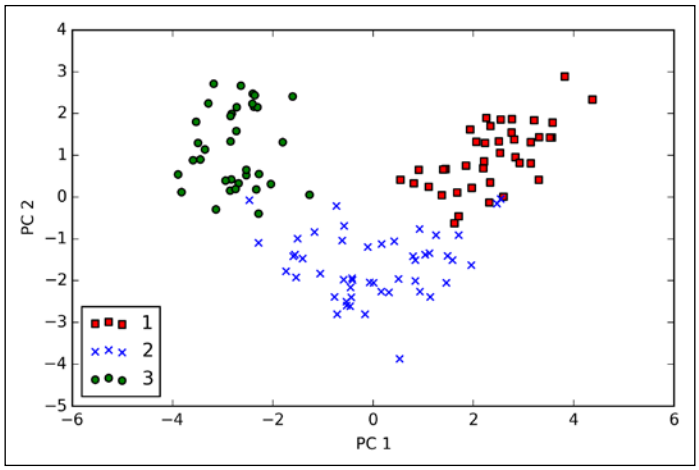
$$\boldsymbol{X}' = \boldsymbol{X}\boldsymbol{W}$$

```
>>> X_train_pca = X_train_std.dot(w)
```

Lastly, let's visualize the transformed *Wine* training set, now stored as an 124×2-dimensional matrix, in a two-dimensional scatterplot:

```
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_pca[y_train==l, 0],
...                 X_train_pca[y_train==l, 1],
...                 c=c, label=l, marker=m)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

As we can see in the resulting plot (shown in the next figure), the data is more spread along the *x*-axis—the first principal component—than the second principal component (*y*-axis), which is consistent with the explained variance ratio plot that we created in the previous subsection. However, we can intuitively see that a linear classifier will likely be able to separate the classes well:



Although we encoded the class labels information for the purpose of illustration in the preceding scatter plot, we have to keep in mind that PCA is an unsupervised technique that doesn't use class label information.

Principal component analysis in scikit-learn

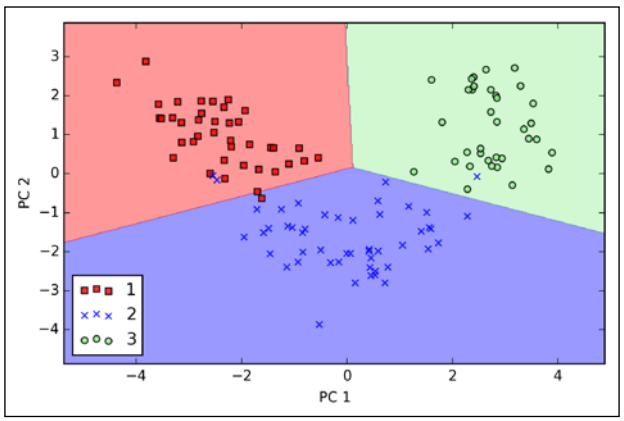
Although the verbose approach in the previous subsection helped us to follow the inner workings of PCA, we will now discuss how to use the `PCA` class implemented in scikit-learn. `PCA` is another one of scikit-learn's transformer classes, where we first fit the model using the training data before we transform both the training data and the test data using the same model parameters. Now, let's use the `PCA` from scikit-learn on the *Wine* training dataset, classify the transformed samples via logistic regression, and visualize the decision regions via the `plot_decision_region` function that we defined in *Chapter 2*, *Training Machine Learning Algorithms for Classification*:

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
```

```
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)
```

```
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)
>>> lr = LogisticRegression()
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> X_test_pca = pca.transform(X_test_std)
>>> lr.fit(X_train_pca, y_train)
>>> plot_decision_regions(X_train_pca, y_train, classifier=lr)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

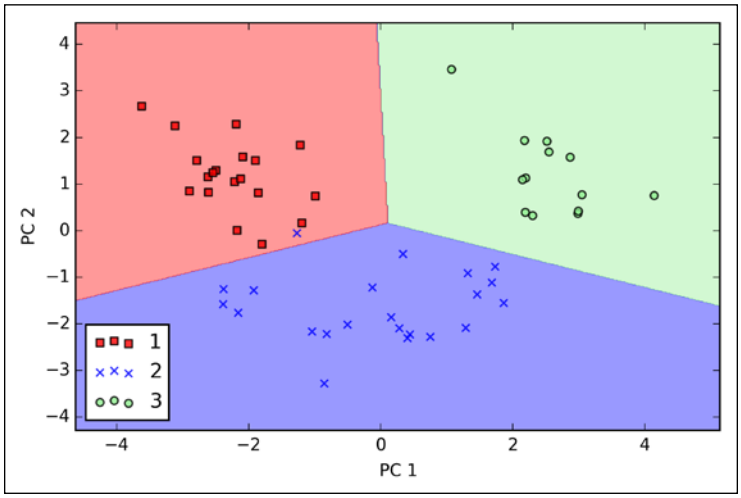
By executing the preceding code, we should now see the decision regions for the training model reduced to the two principal component axes.



If we compare the PCA projection via scikit-learn with our own PCA implementation, we notice that the plot above is a mirror image of the previous PCA via our step-by-step approach. Note that this is not due to an error in any of those two implementations, but the reason for this difference is that, depending on the eigensolver, eigenvectors can have either negative or positive signs. Not that it matters, but we could simply revert the mirror image by multiplying the data with `-1` if we wanted to; note that eigenvectors are typically scaled to unit length `1`. For the sake of completeness, let's plot the decision regions of the logistic regression on the transformed test dataset to see if it can separate the classes well:

```
>>> plot_decision_regions(X_test_pca, y_test, classifier=lr)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

After we plot the decision regions for the test set by executing the preceding code, we can see that logistic regression performs quite well on this small two-dimensional feature subspace and only misclassifies one sample in the test dataset.



If we are interested in the explained variance ratios of the different principal components, we can simply initialize the `PCA` class with the `n_components` parameter set to `None`, so all principal components are kept and the explained variance ratio can then be accessed via the `explained_variance_ratio_` attribute:

```
>>> pca = PCA(n_components=None)
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> pca.explained_variance_ratio_
```

```
array([ 0.37329648,  0.18818926,  0.10896791,  0.07724389,
0.06478595,
0.04592014,  0.03986936,  0.02521914,  0.02258181,  0.01830924,
0.01635336,  0.01284271,  0.00642076])
```

Note that we set `n_components=None` when we initialized the PCA class so that it would return all principal components in sorted order instead of performing a dimensionality reduction.

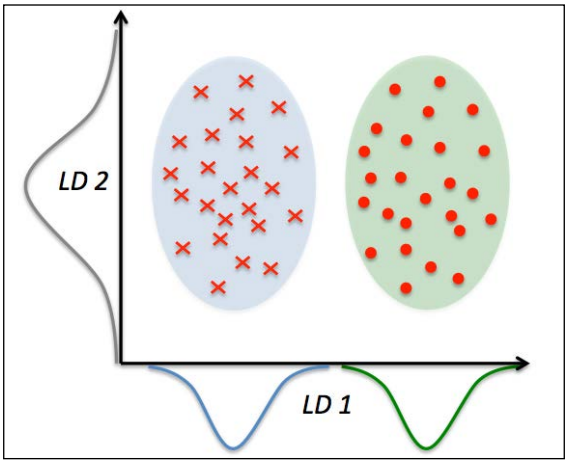
Supervised data compression via linear discriminant analysis

Linear Discriminant Analysis (**LDA**) can be used as a technique for feature extraction to increase the computational efficiency and reduce the degree of over-fitting due to the curse of dimensionality in nonregularized models.

The general concept behind LDA is very similar to PCA, whereas PCA attempts to find the orthogonal component axes of maximum variance in a dataset; the goal in LDA is to find the feature subspace that optimizes class separability. Both LDA and PCA are linear transformation techniques that can be used to reduce the number of dimensions in a dataset; the former is an unsupervised algorithm, whereas the latter is supervised. Thus, we might intuitively think that LDA is a superior feature extraction technique for classification tasks compared to PCA. However, A.M. Martinez reported that preprocessing via PCA tends to result in better classification results in an image recognition task in certain cases, for instance, if each class consists of only a small number of samples (A. M. Martinez and A. C. Kak. *PCA Versus LDA*. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 23(2):228–233, 2001).

Although LDA is sometimes also called Fisher's LDA, Ronald A. Fisher initially formulated *Fisher's Linear Discriminant* for two-class classification problems in 1936 (R. A. Fisher. *The Use of Multiple Measurements in Taxonomic Problems*. Annals of Eugenics, 7(2):179–188, 1936). Fisher's Linear Discriminant was later generalized for multi-class problems by C. Radhakrishna Rao under the assumption of equal class covariances and normally distributed classes in 1948, which we now call LDA (C. R. Rao. *The Utilization of Multiple Measurements in Problems of Biological Classification*. Journal of the Royal Statistical Society. Series B (Methodological), 10(2):159–203, 1948).

The following figure summarizes the concept of LDA for a two-class problem. Samples from class 1 are shown as crosses and samples from class 2 are shown as circles, respectively:



A linear discriminant, as shown on the *x*-axis (LD 1), would separate the two normally distributed classes well. Although the exemplary linear discriminant shown on the *y*-axis (LD 2) captures a lot of the variance in the dataset, it would fail as a good linear discriminant since it does not capture any of the class-discriminatory information.

One assumption in LDA is that the data is normally distributed. Also, we assume that the classes have identical covariance matrices and that the features are statistically independent of each other. However, even if one or more of those assumptions are slightly violated, LDA for dimensionality reduction can still work reasonably well (R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. 2nd. Edition. New York, 2001).

Before we take a look into the inner workings of LDA in the following subsections, let's summarize the key steps of the LDA approach:

1. Standardize the d-dimensional dataset (d is the number of features).
2. For each class, compute the d-dimensional mean vector.
3. Construct the between-class scatter matrix $\boldsymbol{S}_B$ and the within-class scatter matrix $\boldsymbol{S}_w$.

4. Compute the eigenvectors and corresponding eigenvalues of the matrix $\boldsymbol{S}_w^{-1}\boldsymbol{S}_B$.
5. Choose the k eigenvectors that correspond to the k largest eigenvalues to construct a $d \times k$-dimensional transformation matrix $\boldsymbol{W}$; the eigenvectors are the columns of this matrix.
6. Project the samples onto the new feature subspace using the transformation matrix $\boldsymbol{W}$.

The assumptions that we make when we are using LDA are that the features are normally distributed and independent of each other. Also, the LDA algorithm assumes that the covariance matrices for the individual classes are identical. However, even if we violate those assumptions to a certain extent, LDA may still work reasonably well in dimensionality reduction and classification tasks (R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. 2nd. Edition. New York, 2001).

Computing the scatter matrices

Since we have already standardized the features of the *Wine* dataset in the PCA section at the beginning of this chapter, we can skip the first step and proceed with the calculation of the mean vectors, which we will use to construct the within-class scatter matrix and between-class scatter matrix, respectively. Each mean vector $\boldsymbol{m}_i$ stores the mean feature value μ_m with respect to the samples of class i:

$$\boldsymbol{m}_i = \frac{1}{n_i} \sum_{\boldsymbol{x} \in D_i}^{c} \boldsymbol{x}_m$$

This results in three mean vectors:

$$\boldsymbol{m}_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \vdots \\ \mu_{i,proline} \end{bmatrix} \quad i \in \{1,2,3\}$$

```
>>> np.set_printoptions(precision=4)
>>> mean_vecs = []
>>> for label in range(1,4):
...     mean_vecs.append(np.mean(
...                X_train_std[y_train==label], axis=0))
...     print('MV %s: %s\n' %(label, mean_vecs[label-1]))
MV 1: [ 0.9259 -0.3091  0.2592 -0.7989  0.3039  0.9608  1.0515 -0.6306
0.5354
  0.2209  0.4855  0.798   1.2017]

MV 2: [-0.8727 -0.3854 -0.4437  0.2481 -0.2409 -0.1059  0.0187 -0.0164
0.1095
 -0.8796  0.4392  0.2776 -0.7016]

MV 3: [ 0.1637  0.8929  0.3249  0.5658 -0.01   -0.9499 -1.228   0.7436
-0.7652
  0.979  -1.1698 -1.3007 -0.3912]
```

Using the mean vectors, we can now compute the within-class scatter matrix $\boldsymbol{S}_W$:

$$\boldsymbol{S}_W = \sum_{i=1}^{c} \boldsymbol{S}_i$$

This is calculated by summing up the individual scatter matrices $\boldsymbol{s}_i$ of each individual class i:

$$\boldsymbol{S}_i = \sum_{\boldsymbol{x} \in D_i}^{c} \left(\boldsymbol{x} - \boldsymbol{m}_i\right)\left(\boldsymbol{x} - \boldsymbol{m}_i\right)^T$$

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1,4), mean_vecs):
...     class_scatter = np.zeros((d, d))
...     for row in X_train[y_train == label]:
...         row, mv = row.reshape(d, 1), mv.reshape(d, 1)
...         class_scatter += (row-mv).dot((row-mv).T)
...     S_W += class_scatter
>>> print('Within-class scatter matrix: %sx%s'
...        % (S_W.shape[0], S_W.shape[1]))
Within-class scatter matrix: 13x13
```

The assumption that we are making when we are computing the scatter matrices is that the class labels in the training set are uniformly distributed. However, if we print the number of class labels, we see that this assumption is violated:

```
>>> print('Class label distribution: %s'
...       % np.bincount(y_train)[1:])
Class label distribution: [40 49 35]
```

Thus, we want to scale the individual scatter matrices $\boldsymbol{S}_i$ before we sum them up as scatter matrix $\boldsymbol{S}_w$. When we divide the scatter matrices by the number of class samples N_i, we can see that computing the scatter matrix is in fact the same as computing the covariance matrix Σ_i. The covariance matrix is a normalized version of the scatter matrix:

$$\Sigma_i = \frac{1}{N_i}\boldsymbol{S}_W = \frac{1}{N_i}\sum_{\boldsymbol{x}\in D_i}^{c}(\boldsymbol{x}-\boldsymbol{m}_i)(\boldsymbol{x}-\boldsymbol{m}_i)^T$$

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.cov(X_train_std[y_train==label].T)
...     S_W += class_scatter
>>> print('Scaled within-class scatter matrix: %sx%s'
...       % (S_W.shape[0], S_W.shape[1]))
Scaled within-class scatter matrix: 13x13
```

After we have computed the scaled within-class scatter matrix (or covariance matrix), we can move on to the next step and compute the between-class scatter matrix $\boldsymbol{S}_B$:

$$\boldsymbol{S}_B = \sum_{i=1}^{c} N_i(\boldsymbol{m}_i-\boldsymbol{m})(\boldsymbol{m}_i-\boldsymbol{m})^T$$

Here, $\boldsymbol{m}$ is the overall mean that is computed, including samples from all classes.

```
>>> mean_overall = np.mean(X_train_std, axis=0)
>>> d = 13 # number of features
>>> S_B = np.zeros((d, d))
>>> for i,mean_vec in enumerate(mean_vecs):
...     n = X_train[y_train==i+1, :].shape[0]
...     mean_vec = mean_vec.reshape(d, 1)
...     mean_overall = mean_overall.reshape(d, 1)
    S_B += n * (mean_vec - mean_overall).dot(
```

```
...                (mean_vec - mean_overall).T)
print('Between-class scatter matrix: %sx%s'
...    % (S_B.shape[0], S_B.shape[1]))
Between-class scatter matrix: 13x13
```

Selecting linear discriminants for the new feature subspace

The remaining steps of the LDA are similar to the steps of the PCA. However, instead of performing the eigendecomposition on the covariance matrix, we solve the generalized eigenvalue problem of the matrix $\boldsymbol{S}_w^{-1}\boldsymbol{S}_B$:

```
>>>eigen_vals, eigen_vecs =\
...np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

After we computed the eigenpairs, we can now sort the eigenvalues in descending order:

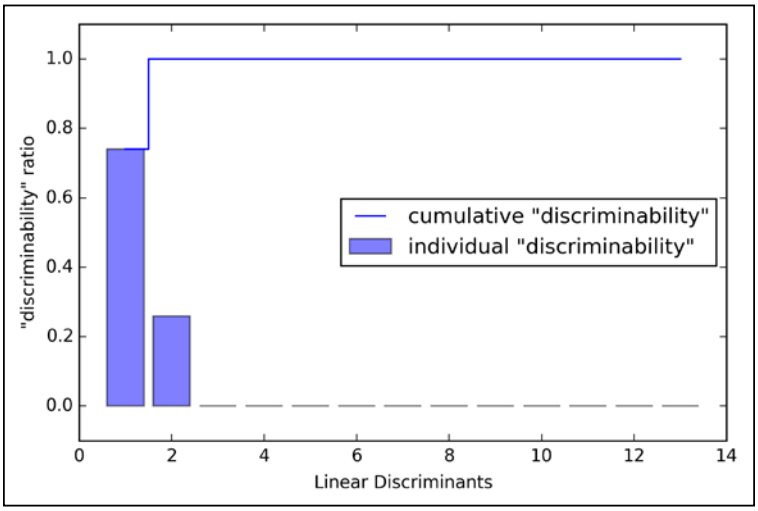
```
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
...               for i in range(len(eigen_vals))]
>>> eigen_pairs = sorted(eigen_pairs,
...                key=lambda k: k[0], reverse=True)
>>> print('Eigenvalues in decreasing order:\n')
>>> for eigen_val in eigen_pairs:
...     print(eigen_val[0])


Eigenvalues in decreasing order:

452.721581245
156.43636122
8.11327596465e-14
2.78687384543e-14
2.78687384543e-14
2.27622032758e-14
2.27622032758e-14
1.97162599817e-14
1.32484714652e-14
1.32484714652e-14
1.03791501611e-14
5.94140664834e-15
2.12636975748e-16
```

In LDA, the number of linear discriminants is at most $c-1$ where c is the number of class labels, since the in-between class scatter matrix $\boldsymbol{S}_B$ is the sum of c matrices with rank 1 or less. We can indeed see that we only have two nonzero eigenvalues (the eigenvalues 3-13 are not exactly zero, but this is due to the floating point arithmetic in NumPy). Note that in the rare case of perfect collinearity (all aligned sample points fall on a straight line), the covariance matrix would have rank one, which would result in only one eigenvector with a nonzero eigenvalue.

To measure how much of the class-discriminatory information is captured by the linear discriminants (eigenvectors), let's plot the linear discriminants by decreasing eigenvalues similar to the explained variance plot that we created in the PCA section. For simplicity, we will call the content of the class-discriminatory information *discriminability*.

```
>>> tot = sum(eigen_vals.real)
>>> discr = [(i / tot) for i in sorted(eigen_vals.real, reverse=True)]
>>> cum_discr = np.cumsum(discr)
>>> plt.bar(range(1, 14), discr, alpha=0.5, align='center',
...         label='individual "discriminability"')
>>> plt.step(range(1, 14), cum_discr, where='mid',
...          label='cumulative "discriminability"')
>>> plt.ylabel('"discriminability" ratio')
>>> plt.xlabel('Linear Discriminants')
>>> plt.ylim([-0.1, 1.1])
>>> plt.legend(loc='best')
>>> plt.show()
```

As we can see in the resulting figure, the first two linear discriminants capture about 100 percent of the useful information in the *Wine* training dataset:



Let's now stack the two most discriminative eigenvector columns to create the transformation matrix W:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Matrix W:\n', w)
Matrix W:
[[ 0.0662 -0.3797]
[-0.0386 -0.2206]
[ 0.0217 -0.3816]
[-0.184 0.3018]
[ 0.0034 0.0141]
[-0.2326 0.0234]
[ 0.7747 0.1869]
[ 0.0811 0.0696]
[-0.0875 0.1796]
[-0.185 -0.284 ]
[ 0.066 0.2349]
[ 0.3805 0.073 ]
[ 0.3285 -0.5971]]
```

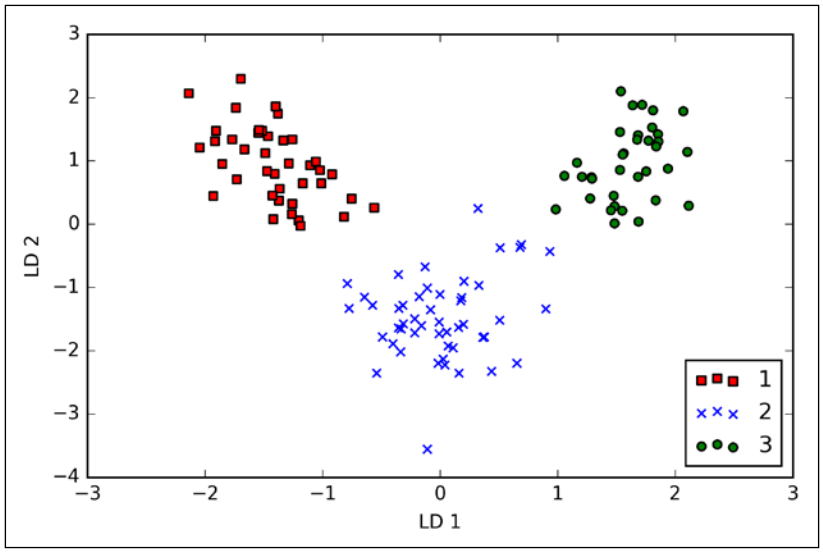
Projecting samples onto the new feature space

Using the transformation matrix W that we created in the previous subsection, we can now transform the training data set by multiplying the matrices:

$$X' = XW$$

```
>>> X_train_lda = X_train_std.dot(w)
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_lda[y_train==l, 0]*(-1)
...                 X_train_lda[y_train==l, 1]*(-1)
...                 c=c, label=l, marker=m)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower right')
>>> plt.show()
```

As we can see in the resulting plot, the three wine classes are now linearly separable in the new feature subspace:



LDA via scikit-learn

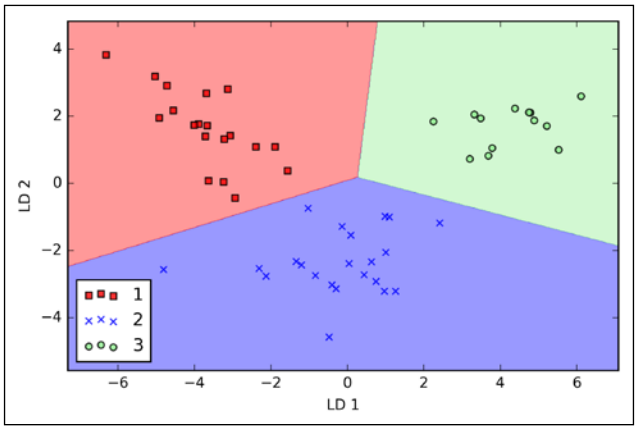
The step-by-step implementation was a good exercise for understanding the inner workings of LDA and understanding the differences between LDA and PCA. Now, let's take a look at the `LDA` class implemented in scikit-learn:

```
>>> from sklearn.lda import LDA
>>> lda = LDA(n_components=2)
>>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

Next, let's see how the logistic regression classifier handles the lower-dimensional training dataset after the LDA transformation:

```
>>> lr = LogisticRegression()
>>> lr = lr.fit(X_train_lda, y_train)
>>> plot_decision_regions(X_train_lda, y_train, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

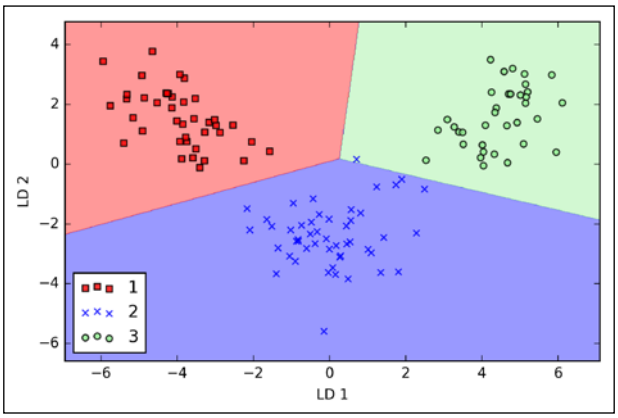
Looking at the resulting plot, we see that the logistic regression model misclassifies one of the samples from class 2:



By lowering the regularization strength, we could probably shift the decision boundaries so that the logistic regression models classify all samples in the training dataset correctly. However, let's take a look at the results on the test set:

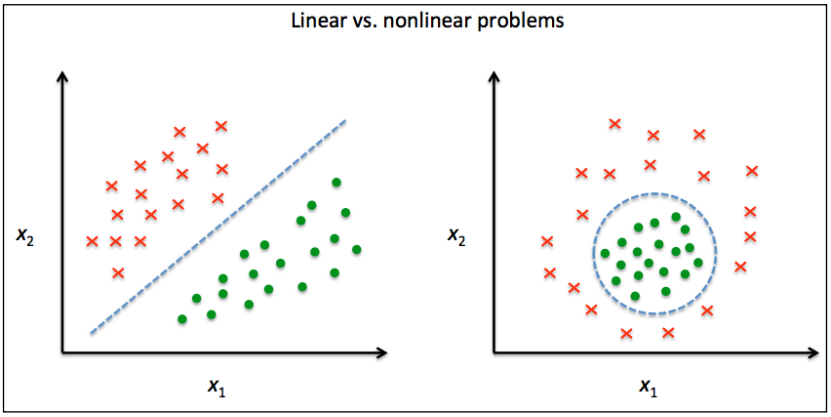
```
>>> X_test_lda = lda.transform(X_test_std)
>>> plot_decision_regions(X_test_lda, y_test, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

As we can see in the resulting plot, the logistic regression classifier is able to get a perfect accuracy score for classifying the samples in the test dataset by only using a two-dimensional feature subspace instead of the original 13 *Wine* features:



Using kernel principal component analysis for nonlinear mappings

Many machine learning algorithms make assumptions about the linear separability of the input data. You learned that the perceptron even requires perfectly linearly separable training data to converge. Other algorithms that we have covered so far assume that the lack of perfect linear separability is due to noise: Adaline, logistic regression, and the (standard) **support vector machine** (**SVM**) to just name a few. However, if we are dealing with nonlinear problems, which we may encounter rather frequently in real-world applications, linear transformation techniques for dimensionality reduction, such as PCA and LDA, may not be the best choice. In this section, we will take a look at a kernelized version of PCA, or *kernel PCA*, which relates to the concepts of kernel SVM that we remember from *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*. Using kernel PCA, we will learn how to transform data that is not linearly separable onto a new, lower-dimensional subspace that is suitable for linear classifiers.



Kernel functions and the kernel trick

As we remember from our discussion about kernel SVMs in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, we can tackle nonlinear problems by projecting them onto a new feature space of higher dimensionality where the classes become linearly separable. To transform the samples $\boldsymbol{x} \in \mathbb{R}^d$ onto this higher k-dimensional subspace, we defined a nonlinear mapping function ϕ:

$$\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k \quad (k >> d)$$

We can think of ϕ as a function that creates nonlinear combinations of the original features to map the original d-dimensional dataset onto a larger, k-dimensional feature space. For example, if we had feature vector $\boldsymbol{x} \in \mathbb{R}^d$ ($\boldsymbol{x}$ is a column vector consisting of d features) with two dimensions $(d = 2)$, a potential mapping onto a 3D space could be as follows:

$$\boldsymbol{x} = \begin{bmatrix} x_1, & x_2 \end{bmatrix}^T$$

$$\downarrow \phi$$

$$\mathbf{z} = \begin{bmatrix} x_1^2, \sqrt{2x_1x_2}, x_2^2 \end{bmatrix}^T$$

In other words, via kernel PCA we perform a nonlinear mapping that transforms the data onto a higher-dimensional space and use standard PCA in this higher-dimensional space to project the data back onto a lower-dimensional space where the samples can be separated by a linear classifier (under the condition that the samples can be separated by density in the input space). However, one downside of this approach is that it is computationally very expensive, and this is where we use the *kernel trick*. Using the kernel trick, we can compute the similarity between two high-dimension feature vectors in the original feature space.

Before we proceed with more details about using the kernel trick to tackle this computationally expensive problem, let's look back at the *standard* PCA approach that we implemented at the beginning of this chapter. We computed the covariance between two features k and j as follows:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^{n} \left(x_j^{(i)} - \mu_j \right) \left(x_k^{(i)} - \mu_k \right)$$

Since the standardizing of features centers them at mean zero, for instance, $\mu_j = 0$ and $\mu_k = 0$, we can simplify this equation as follows:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^{n} x_j^{(i)} x_k^{(i)}$$

Note that the preceding equation refers to the covariance between two features; now, let's write the general equation to calculate the covariance *matrix* Σ:

$$\sum = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}^{(i)}\, \boldsymbol{x}^{(i)T}$$

Bernhard Scholkopf generalized this approach (B. Scholkopf, A. Smola, and K.-R. Muller. *Kernel Principal Component Analysis*. pages 583–588, 1997) so that we can replace the dot products between samples in the original feature space by the nonlinear feature combinations via ϕ:

$$\sum = \frac{1}{n}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi(\boldsymbol{x}^{(i)})^{T}$$

To obtain the eigenvectors—the principal components—from this covariance matrix, we have to solve the following equation:

$$\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}$$

$$\Rightarrow \frac{1}{n}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T} \boldsymbol{v} = \lambda \boldsymbol{v}$$

$$\Rightarrow v = \frac{1}{n\lambda}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T} \boldsymbol{v} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{a}^{(i)}\phi\left(\boldsymbol{x}^{(i)}\right)$$

Here, λ and $\boldsymbol{v}$ are the eigenvalues and eigenvectors of the covariance matrix Σ, and $\boldsymbol{a}$ can be obtained by extracting the eigenvectors of the kernel (similarity) matrix $\boldsymbol{K}$ as we will see in the following paragraphs.

The derivation of the kernel matrix is as follows:

First, let's write the covariance matrix as in matrix notation, where $\phi(X)$ is an $n \times k$-dimensional matrix:

$$\sum = \frac{1}{n}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T} = \frac{1}{n}\phi(\boldsymbol{X})^{T}\,\phi(\boldsymbol{X})$$

Now, we can write the eigenvector equation as follows:

$$v = \frac{1}{n}\sum_{i=1}^{n} a^{(i)}\phi\left(\boldsymbol{x}^{(i)}\right) = \lambda\phi(\boldsymbol{X})^{T}\,\boldsymbol{a}$$

Since $\Sigma\boldsymbol{v} = \lambda\boldsymbol{v}$, we get:

$$\frac{1}{n}\phi(\boldsymbol{X})^{T}\,\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\,\boldsymbol{a} = \lambda\phi(\boldsymbol{X})^{T}\,\boldsymbol{a}$$

Multiplying it by $\phi(\boldsymbol{X})$ on both sides yields the following result:

$$\frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\,\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\,\boldsymbol{a} = \lambda\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\,\boldsymbol{a}$$

$$\Rightarrow \frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\,\boldsymbol{a} = \lambda\boldsymbol{a}$$

$$\Rightarrow \frac{1}{n}\boldsymbol{K}\boldsymbol{a} = \lambda\boldsymbol{a}$$

Here, $\boldsymbol{K}$ is the similarity (kernel) matrix:

$$\boldsymbol{K} = \phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}$$

As we recall from the SVM section in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, we use the kernel trick to avoid calculating the pairwise dot products of the samples $\boldsymbol{x}$ under ϕ explicitly by using a kernel function $\boldsymbol{K}$ so that we don't need to calculate the eigenvectors explicitly:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right) = \phi\left(\boldsymbol{x}^{(i)}\right)^{T}\,\phi\left(\boldsymbol{x}^{(j)}\right)$$

In other words, what we obtain after kernel PCA are the samples already projected onto the respective components rather than constructing a transformation matrix as in the standard PCA approach. Basically, the kernel function (or simply *kernel*) can be understood as a function that calculates a dot product between two vectors—a measure of similarity.

The most commonly used kernels are as follows:

- The polynomial kernel:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\left(\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}+\theta\right)^{p}$$

  Here, θ is the threshold and p is the power that has to be specified by the user.

- The hyperbolic tangent (sigmoid) kernel:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\tanh\left(\eta\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}+\theta\right)$$

- The **Radial Basis Function** (**RBF**) or Gaussian kernel that we will use in the following examples in the next subsection:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\frac{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^{2}}{2\sigma^{2}}\right)$$

  It is also written as follows:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\gamma\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^{2}\right)$$

To summarize what we have discussed so far, we can define the following three steps to implement an RBF kernel PCA:

1. We compute the kernel (similarity) matrix k, where we need to calculate the following:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\gamma\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^{2}\right)$$

We do this for each pair of samples:

$$\boldsymbol{K} = \begin{bmatrix} \kappa\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(1)}\right) & \kappa\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(n)}\right) \\ \kappa\left(\boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right) & \left(\boldsymbol{x}^{(2)}, \boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(2)}, \boldsymbol{x}^{(n)}\right) \\ \vdots & \vdots & \ddots & \vdots \\ \kappa\left(\boldsymbol{x}^{(n)}, \boldsymbol{x}^{(1)}\right) & \kappa\left(\boldsymbol{x}^{(n)}, \boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(n)}, \boldsymbol{x}^{(n)}\right) \end{bmatrix}$$

For example, if our dataset contains 100 training samples, the symmetric kernel matrix of the pair-wise similarities would be 100×100 dimensional.

2. We center the kernel matrix k using the following equation:

$$\boldsymbol{K}' = \boldsymbol{K} - 1_{\boldsymbol{n}}\boldsymbol{K} - \boldsymbol{K}1_{\boldsymbol{n}} + 1_{\boldsymbol{n}}\boldsymbol{K}1_{\boldsymbol{n}}$$

Here, $1_{\boldsymbol{n}}$ is an $n \times n$ - dimensional matrix (the same dimensions as the kernel matrix) where all values are equal to $\frac{1}{n}$.

3. We collect the top k eigenvectors of the centered kernel matrix based on their corresponding eigenvalues, which are ranked by decreasing magnitude. In contrast to standard PCA, the eigenvectors are not the principal component axes but the samples projected onto those axes.

At this point, you may be wondering why we need to center the kernel matrix in the second step. We previously assumed that we are working with standardized data, where all features have mean zero when we formulated the covariance matrix and replaced the dot products by the nonlinear feature combinations via ϕ.Thus, the centering of the kernel matrix in the second step becomes necessary, since we do not compute the new feature space explicitly and we cannot guarantee that the new feature space is also centered at zero.

In the next section, we will put those three steps into action by implementing a kernel PCA in Python.

Implementing a kernel principal component analysis in Python

In the previous subsection, we discussed the core concepts behind kernel PCA. Now, we are going to implement an RBF kernel PCA in Python following the three steps that summarized the kernel PCA approach. Using the SciPy and NumPy helper functions, we will see that implementing a kernel PCA is actually really simple:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.

    Parameters
    ------------
    X: {NumPy ndarray}, shape = [n_samples, n_features]

    gamma: float
      Tuning parameter of the RBF kernel

    n_components: int
      Number of principal components to return

    Returns
    ------------
     X_pc: {NumPy ndarray}, shape = [n_samples, k_features]
       Projected dataset

    """
    # Calculate pairwise squared Euclidean distances
    # in the MxN dimensional dataset.
    sq_dists = pdist(X, 'sqeuclidean')

    # Convert pairwise distances into a square matrix.
    mat_sq_dists = squareform(sq_dists)

    # Compute the symmetric kernel matrix.
    K = exp(-gamma * mat_sq_dists)
```

```
    # Center the kernel matrix.
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    # Obtaining eigenpairs from the centered kernel matrix
    # numpy.eigh returns them in sorted order
    eigvals, eigvecs = eigh(K)

    # Collect the top k eigenvectors (projected samples)
    X_pc = np.column_stack((eigvecs[:, -i]
                            for i in range(1, n_components + 1)))

    return X_pc
```

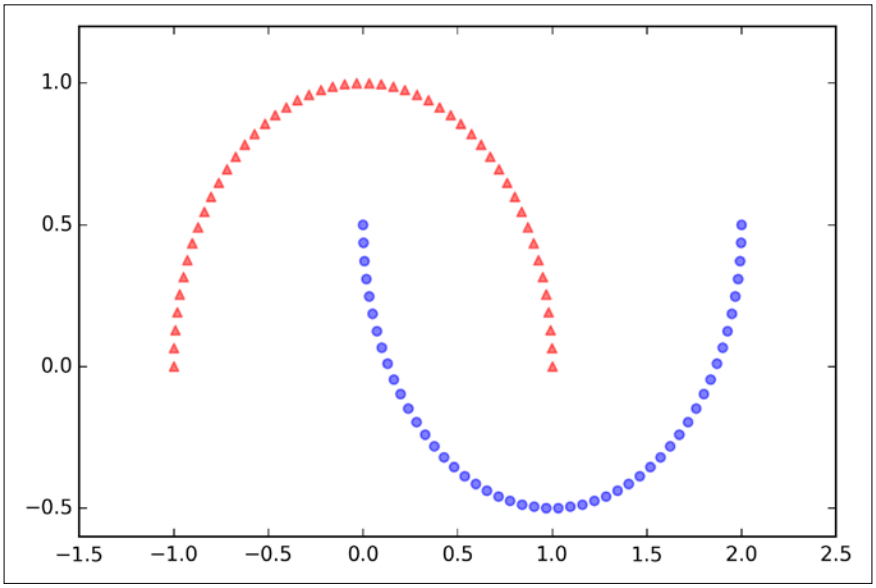
One downside of using an RBF kernel PCA for dimensionality reduction is that we have to specify the parameter γ a priori. Finding an appropriate value for γ requires experimentation and is best done using algorithms for parameter tuning, for example, grid search, which we will discuss in more detail in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

Example 1 – separating half-moon shapes

Now, let's apply our `rbf_kernel_pca` on some nonlinear example datasets. We will start by creating a two-dimensional dataset of 100 sample points representing two half-moon shapes:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.show()
```

For the purposes of illustration, the half-moon of triangular symbols shall represent one class and the half-moon depicted by the circular symbols represent the samples from another class:

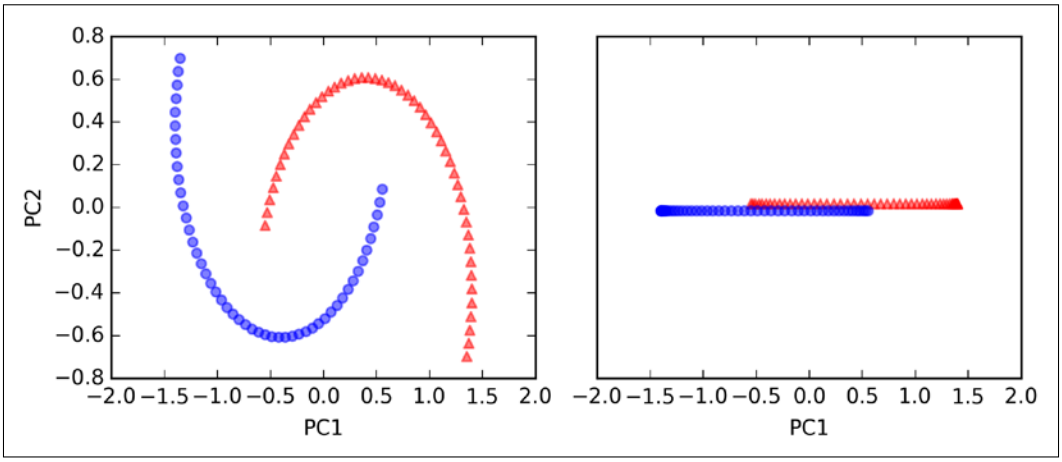


Clearly, these two half-moon shapes are not linearly separable and our goal is to *unfold* the half-moons via kernel PCA so that the dataset can serve as a suitable input for a linear classifier. But first, let's see what the dataset looks like if we project it onto the principal components via standard PCA:

```
>>> from sklearn.decomposition import PCA
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((50,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((50,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
```

```
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

Clearly, we can see in the resulting figure that a linear classifier would be unable to perform well on the dataset transformed via standard PCA:



Note that when we plotted the first principal component only (right subplot), we shifted the triangular samples slightly upwards and the circular samples slightly downwards to better visualize the class overlap.

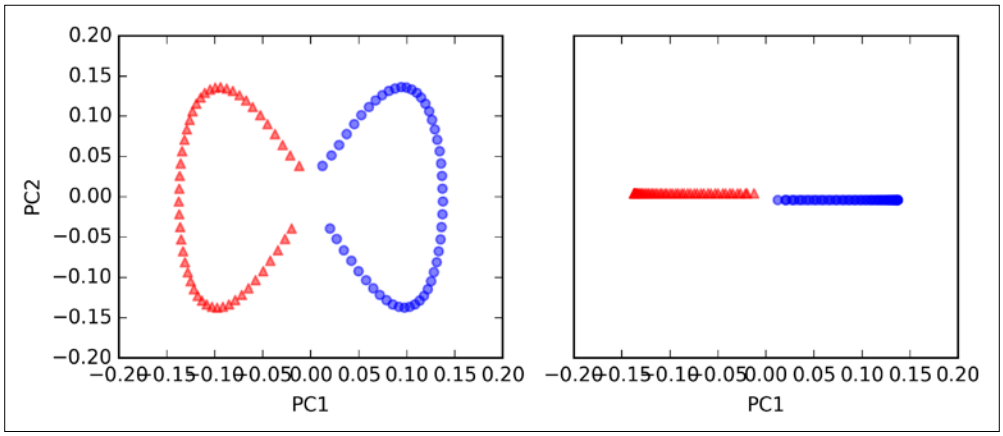
Please remember that PCA is an unsupervised method and does not use class label information in order to maximize the variance in contrast to LDA. Here, the triangular and circular symbols were just added for visualization purposes to indicate the degree of separation.

Now, let's try out our kernel PCA function `rbf_kernel_pca`, which we implemented in the previous subsection:

```
>>> from matplotlib.ticker import FormatStrFormatter
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
```

```
...                color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
...                color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
...                color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> ax[0].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
>>> ax[1].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
>>> plt.show()
```

We can now see that the two classes (circles and triangles) are linearly well separated so that it becomes a suitable training dataset for linear classifiers:



Unfortunately, there is no universal value for the tuning parameter γ that works well for different datasets. To find a γ value that is appropriate for a given problem requires experimentation. In *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will discuss techniques that can help us to automate the task of optimizing such tuning parameters. Here, I will use values for γ that I found produce *good* results.

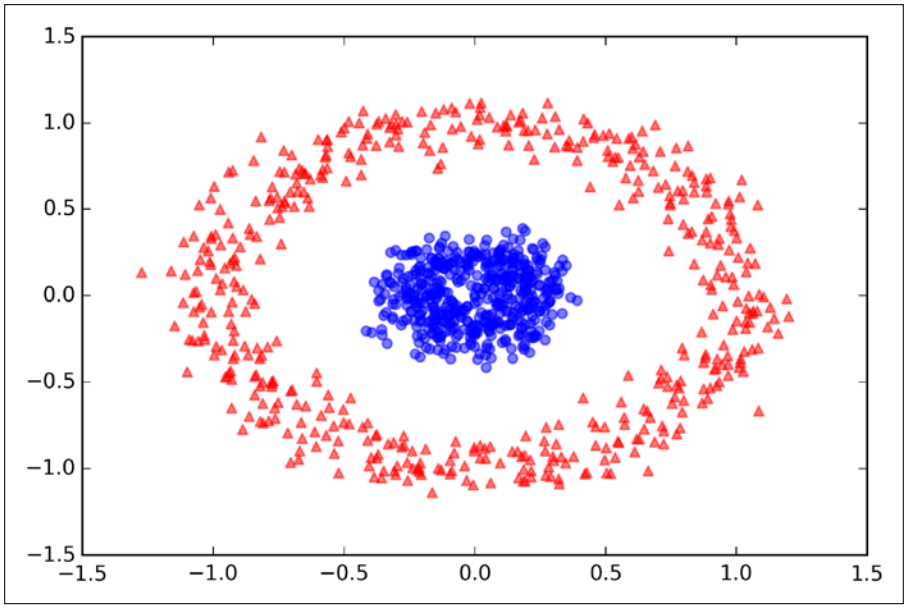
Example 2 – separating concentric circles

In the previous subsection, we showed you how to separate half-moon shapes via kernel PCA. Since we put so much effort into understanding the concepts of kernel PCA, let's take a look at another interesting example of a nonlinear problem: concentric circles.

The code is as follows:

```
>>> from sklearn.datasets import make_circles
>>> X, y = make_circles(n_samples=1000,
...                     random_state=123, noise=0.1, factor=0.2)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.show()
```

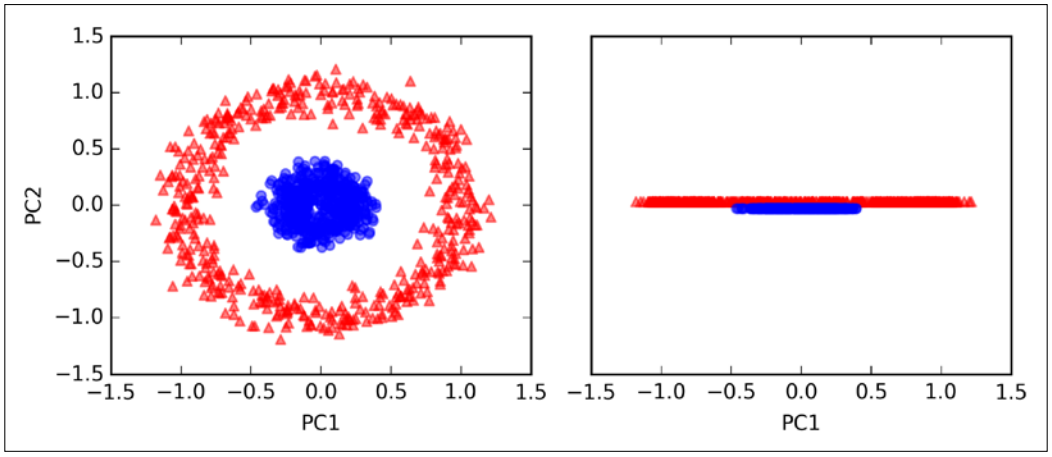
Again, we assume a two-class problem where the triangle shapes represent one class and the circle shapes represent another class, respectively:



Let's start with the standard PCA approach to compare it with the results of the RBF kernel PCA:

```
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((500,1))+0.02,
...              color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((500,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

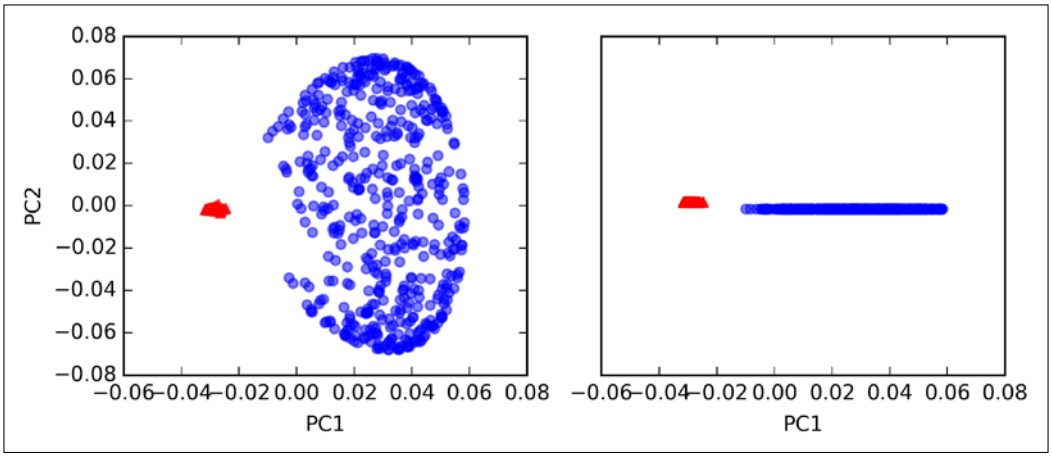
Again, we can see that standard PCA is not able to produce results suitable for training a linear classifier:



Given an appropriate value for γ, let's see if we are luckier using the RBF kernel PCA implementation:

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((500,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((500,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

Again, the RBF kernel PCA projected the data onto a new subspace where the two classes become linearly separable:



Projecting new data points

In the two previous example applications of kernel PCA, the half-moon shapes and the concentric circles, we projected a single dataset onto a new feature. In real applications, however, we may have more than one dataset that we want to transform, for example, training and test data, and typically also new samples we will collect after the model building and evaluation. In this section, you will learn how to project data points that were not part of the training dataset.

As we remember from the standard PCA approach at the beginning of this chapter, we project data by calculating the dot product between a transformation matrix and the input samples; the columns of the projection matrix are the top k eigenvectors ($\boldsymbol{v}$) that we obtained from the covariance matrix. Now, the question is how can we transfer this concept to kernel PCA? If we think back to the idea behind kernel PCA, we remember that we obtained an eigenvector ($\boldsymbol{a}$) of the centered kernel matrix (not the covariance matrix), which means that those are the samples that are already projected onto the principal component axis $\boldsymbol{v}$. Thus, if we want to project a new sample $\boldsymbol{x}'$ onto this principal component axis, we'd need to compute the following:

$$\phi(\boldsymbol{x}')^T \boldsymbol{v}$$

Fortunately, we can use the kernel trick so that we don't have to calculate the projection $\phi(\boldsymbol{x}')^T \boldsymbol{v}$ explicitly. However, it is worth noting that kernel PCA, in contrast to standard PCA, is a memory-based method, which means that we have to reuse the original training set each time to project new samples. We have to calculate the pairwise RBF kernel (similarity) between each i th sample in the training dataset and the new sample $\boldsymbol{x}'$:

$$\phi(\boldsymbol{x}')^T \boldsymbol{v} = \sum_i a^{(i)} \phi(\boldsymbol{x}')^T \phi\left(\boldsymbol{x}^{(i)}\right)$$

$$= \sum_i a^{(i)} k\left(\boldsymbol{x}', \boldsymbol{x}^{(i)}\right)^T$$

Here, eigenvectors $\boldsymbol{a}$ and eigenvalues λ of the Kernel matrix $\boldsymbol{K}$ satisfy the following condition in the equation:

$$\boldsymbol{K}\boldsymbol{a} = \lambda \boldsymbol{a}$$

After calculating the similarity between the new samples and the samples in the training set, we have to normalize the eigenvector $\boldsymbol{a}$ by its eigenvalue. Thus, let's modify the `rbf_kernel_pca` function that we implemented earlier so that it also returns the eigenvalues of the kernel matrix:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.

    Parameters
    ------------
    X: {NumPy ndarray}, shape = [n_samples, n_features]

    gamma: float
      Tuning parameter of the RBF kernel

    n_components: int
      Number of principal components to return

    Returns
    ------------
     X_pc: {NumPy ndarray}, shape = [n_samples, k_features]
       Projected dataset

     lambdas: list
       Eigenvalues

    """
    # Calculate pairwise squared Euclidean distances
    # in the MxN dimensional dataset.
    sq_dists = pdist(X, 'sqeuclidean')

    # Convert pairwise distances into a square matrix.
    mat_sq_dists = squareform(sq_dists)

    # Compute the symmetric kernel matrix.
    K = exp(-gamma * mat_sq_dists)
```

```
    # Center the kernel matrix.
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    # Obtaining eigenpairs from the centered kernel matrix
    # numpy.eigh returns them in sorted order
    eigvals, eigvecs = eigh(K)

    # Collect the top k eigenvectors (projected samples)
    alphas = np.column_stack((eigvecs[:,-i]
                    for i in range(1,n_components+1)))

    # Collect the corresponding eigenvalues
    lambdas = [eigvals[-i] for i in range(1,n_components+1)]

    return alphas, lambdas
```

Now, let's create a new half-moon dataset and project it onto a one-dimensional subspace using the updated RBF kernel PCA implementation:

```
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> alphas, lambdas =rbf_kernel_pca(X, gamma=15, n_components=1)
```

To make sure that we implement the code for projecting new samples, let's assume that the 26th point from the half-moon dataset is a new data point $\boldsymbol{x}'$, and our task is to project it onto this new subspace:

```
>>> x_new = X[25]
>>> x_new
array([ 1.8713187 ,  0.00928245])
>>> x_proj = alphas[25] # original projection
>>> x_proj
array([ 0.07877284])
>>> def project_x(x_new, X, gamma, alphas, lambdas):
...     pair_dist = np.array([np.sum(
...                     (x_new-row)**2) for row in X])
...     k = np.exp(-gamma * pair_dist)
... return k.dot(alphas / lambdas)
```

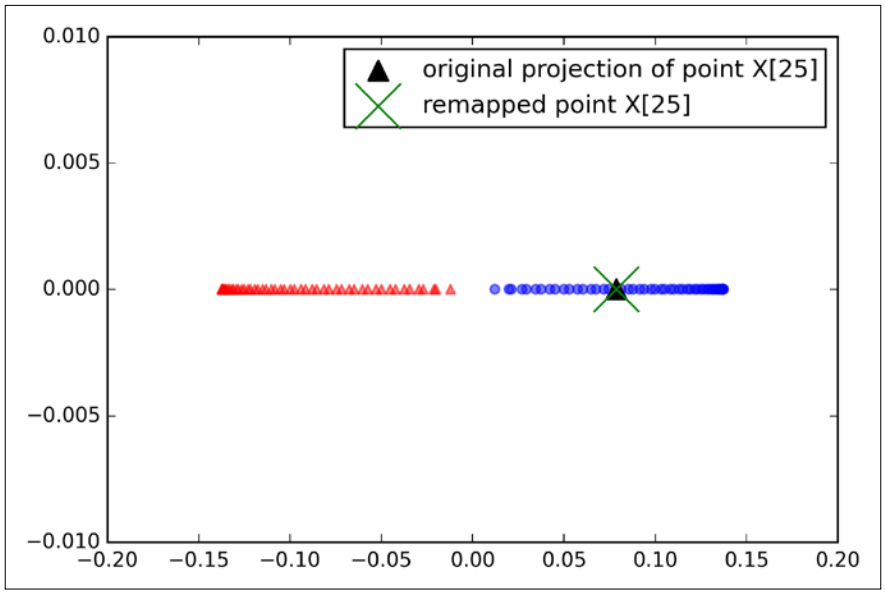
By executing the following code, we are able to reproduce the original projection. Using the `project_x` function, we will be able to project any new data samples as well. The code is as follows:

```
>>> x_reproj = project_x(x_new, X,
...       gamma=15, alphas=alphas, lambdas=lambdas)
>>> x_reproj
array([ 0.07877284])
```

Lastly, let's visualize the projection on the first principal component:

```
>>> plt.scatter(alphas[y==0, 0], np.zeros((50)),
...             color='red', marker='^',alpha=0.5)
>>> plt.scatter(alphas[y==1, 0], np.zeros((50)),
...             color='blue', marker='o', alpha=0.5)
>>> plt.scatter(x_proj, 0, color='black',
...             label='original projection of point X[25]',
...             marker='^', s=100)
>>> plt.scatter(x_reproj, 0, color='green',
...             label='remapped point X[25]',
...             marker='x', s=500)
>>> plt.legend(scatterpoints=1)
>>> plt.show()
```

As we can see in the following scatterplot, we mapped the sample $\boldsymbol{x}'$ onto the first principal component correctly:



Kernel principal component analysis in scikit-learn

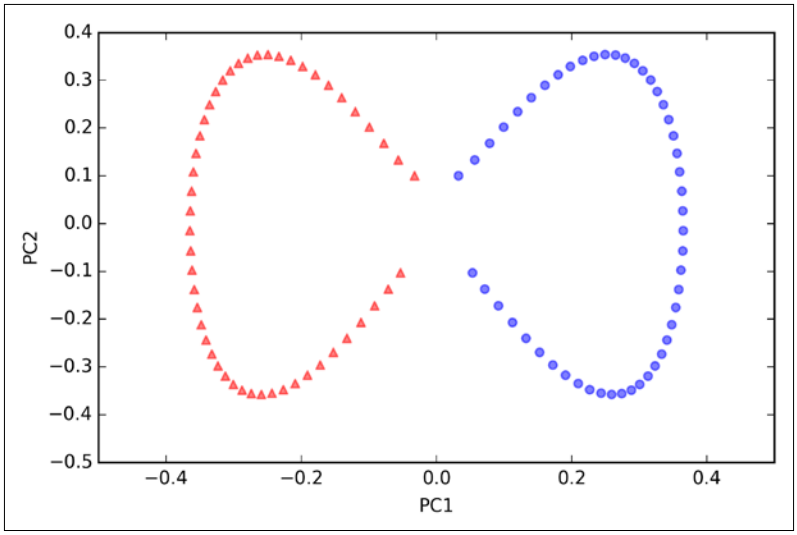
For our convenience, scikit-learn implements a kernel PCA class in the `sklearn.decomposition` submodule. The usage is similar to the standard PCA class, and we can specify the kernel via the `kernel` parameter:

```
>>> from sklearn.decomposition import KernelPCA
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> scikit_kpca = KernelPCA(n_components=2,
...                         kernel='rbf', gamma=15)
>>> X_skernpca = scikit_kpca.fit_transform(X)
```

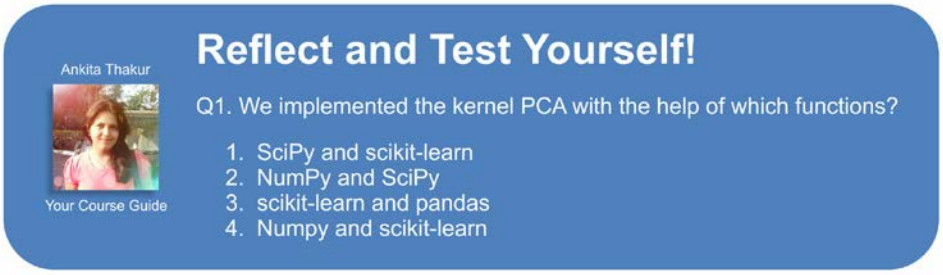
To see if we get results that are consistent with our own kernel PCA implementation, let's plot the transformed half-moon shape data onto the first two principal components:

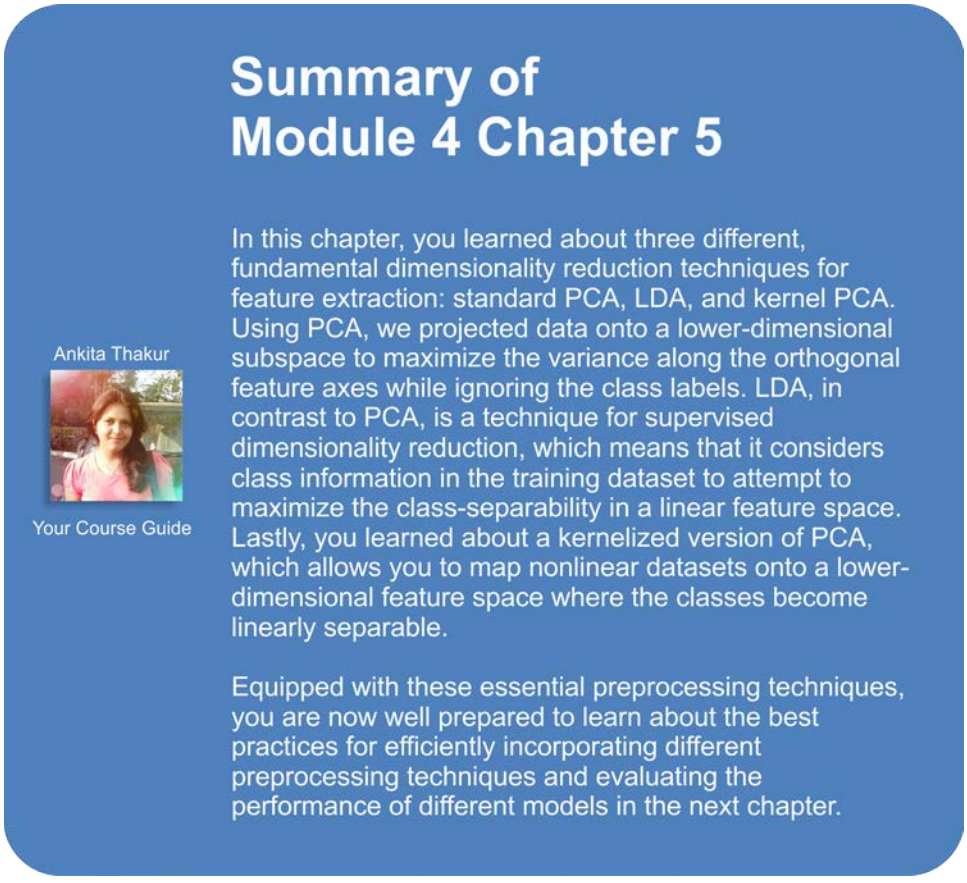
```
>>> plt.scatter(X_skernpca[y==0, 0], X_skernpca[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X_skernpca[y==1, 0], X_skernpca[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.show()
```

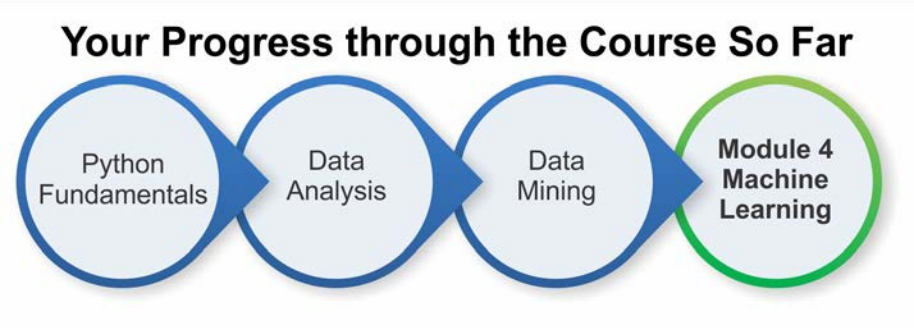
As we can see, the results of the scikit-learn `KernelPCA` are consistent with our own implementation:



Scikit-learn also implements advanced techniques for nonlinear dimensionality reduction that are beyond the scope of this book. You can find a nice overview of the current implementations in scikit-learn complemented with illustrative examples at `http://scikit-learn.org/stable/modules/manifold.html`.






Your Progress through the Course So Far
Python Fundamentals
Data Analysis
Data Mining
Module 4 Machine Learning

6
Learning Best Practices for Model Evaluation and Hyperparameter Tuning

In the previous chapters, you learned about the essential machine learning algorithms for classification and how to get our data into shape before we feed it into those algorithms. Now, it's time to learn about the best practices of building good machine learning models by fine-tuning the algorithms and evaluating the model's performance! In this chapter, we will learn how to:

- Obtain unbiased estimates of a model's performance
- Diagnose the common problems of machine learning algorithms
- Fine-tune machine learning models
- Evaluate predictive models using different performance metrics

Streamlining workflows with pipelines

When we applied different preprocessing techniques in the previous chapters, such as **standardization** for feature scaling in *Chapter 4*, *Building Good Training Sets – Data Preprocessing*, or **principal component analysis** for data compression in *Chapter 5*, *Compressing Data via Dimensionality Reduction*, you learned that we have to reuse the parameters that were obtained during the fitting of the training data to scale and compress any new data, for example, the samples in the separate test dataset.
In this section, you will learn about an extremely handy tool, the `Pipeline` class in scikit-learn. It allows us to fit a model including an arbitrary number of transformation steps and apply it to make predictions about new data.

Loading the Breast Cancer Wisconsin dataset

In this chapter, we will be working with the **Breast Cancer Wisconsin** dataset, which contains 569 samples of **malignant** and **benign** tumor cells. The first two columns in the dataset store the unique ID numbers of the samples and the corresponding diagnosis (*M*=*malignant*, *B*=*benign*), respectively. The columns 3-32 contain 30 real-value features that have been computed from digitized images of the cell nuclei, which can be used to build a model to predict whether a tumor is benign or malignant. The Breast Cancer Wisconsin dataset has been deposited on the *UCI machine learning repository* and more detailed information about this dataset can be found at `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)`.

In this section we will read in the dataset, and split it into training and test datasets in three simple steps:

1. We will start by reading in the dataset directly from the UCI website using pandas:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/breast-cancer-wisconsin/wdbc.data',
header=None)
```

2. Next, we assign the 30 features to a NumPy array `X`. Using `LabelEncoder`, we transform the class labels from their original string representation (`M` and `B`) into integers:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> X = df.loc[:, 2:].values
>>> y = df.loc[:, 1].values
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

   After encoding the class labels (diagnosis) in an array `y`, the malignant tumors are now represented as class `1`, and the benign tumors are represented as class `0`, respectively, which we can illustrate by calling the `transform` method of `LabelEncoder` on two dummy class labels:

```
>>> le.transform(['M', 'B'])
array([1, 0])
```

3. Before we construct our first model pipeline in the following subsection, let's divide the dataset into a separate training dataset (80 percent of the data) and a separate test dataset (20 percent of the data):

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test = \
...      train_test_split(X, y, test_size=0.20, random_state=1)
```

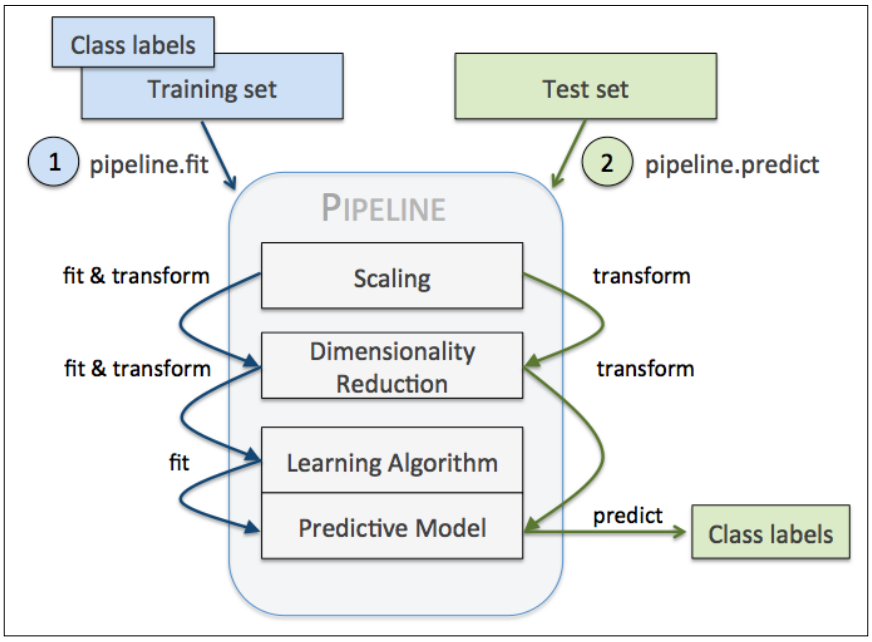
Combining transformers and estimators in a pipeline

In the previous chapter, you learned that many learning algorithms require input features on the same scale for optimal performance. Thus, we need to standardize the columns in the Breast Cancer Wisconsin dataset before we can feed them to a linear classifier, such as logistic regression. Furthermore, let's assume that we want to compress our data from the initial 30 dimensions onto a lower two-dimensional subspace via **principal component analysis** (**PCA**), a feature extraction technique for dimensionality reduction that we introduced in *Chapter 5*, *Compressing Data via Dimensionality Reduction*. Instead of going through the fitting and transformation steps for the training and test dataset separately, we can chain the `StandardScaler`, `PCA`, and `LogisticRegression` objects in a pipeline:

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.pipeline import Pipeline
>>> pipe_lr = Pipeline([('scl', StandardScaler()),
...             ('pca', PCA(n_components=2)),
...             ('clf', LogisticRegression(random_state=1))])
>>> pipe_lr.fit(X_train, y_train)
>>> print('Test Accuracy: %.3f' % pipe_lr.score(X_test, y_test))
Test Accuracy: 0.947
```

The `Pipeline` object takes a list of tuples as input, where the first value in each tuple is an arbitrary identifier string that we can use to access the individual elements in the pipeline, as we will see later in this chapter, and the second element in every tuple is a scikit-learn transformer or estimator.

The intermediate steps in a pipeline constitute scikit-learn transformers, and the last step is an estimator. In the preceding code example, we built a pipeline that consisted of two intermediate steps, a `StandardScaler` and a `PCA` transformer, and a logistic regression classifier as a final estimator. When we executed the `fit` method on the pipeline `pipe_lr`, the `StandardScaler` performed `fit` and `transform` on the training data, and the transformed training data was then passed onto the next object in the pipeline, the `PCA`. Similar to the previous step, `PCA` also executed `fit` and `transform` on the scaled input data and passed it to the final element of the pipeline, the estimator. We should note that there is no limit to the number of intermediate steps in this pipeline. The concept of how pipelines work is summarized in the following figure:



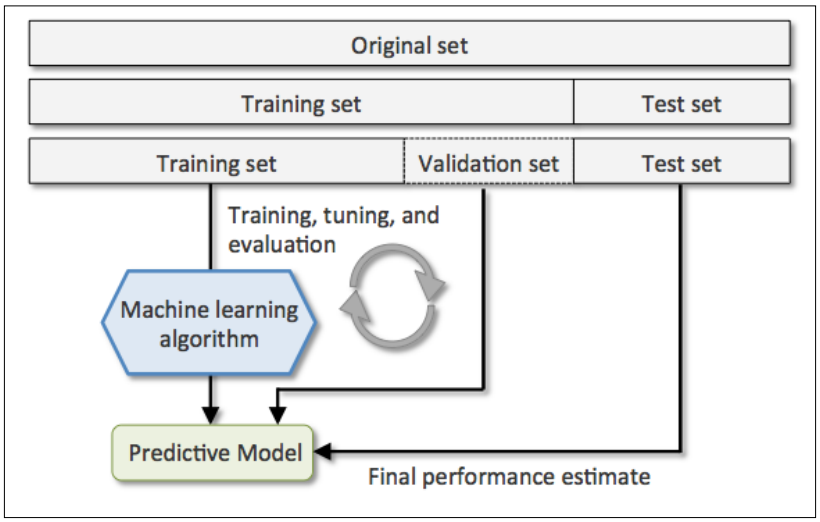
Using k-fold cross-validation to assess model performance

One of the key steps in building a machine learning model is to estimate its performance on data that the model hasn't seen before. Let's assume that we fit our model on a training dataset and use the same data to estimate how well it performs in practice. We remember from the *Tackling overfitting via regularization* section in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, that a model can either suffer from underfitting (high bias) if the model is too simple, or it can overfit the training data (high variance) if the model is too complex for the underlying training data. To find an acceptable bias-variance trade-off, we need to evaluate our model carefully. In this section, you will learn about the useful cross-validation techniques **holdout cross-validation** and **k-fold cross-validation**, which can help us to obtain reliable estimates of the model's generalization error, that is, how well the model performs on unseen data.

The holdout method

A classic and popular approach for estimating the generalization performance of machine learning models is holdout cross-validation. Using the holdout method, we split our initial dataset into a separate training and test dataset—the former is used for model training, and the latter is used to estimate its performance. However, in typical machine learning applications, we are also interested in tuning and comparing different parameter settings to further improve the performance for making predictions on unseen data. This process is called **model selection**, where the term model selection refers to a given classification problem for which we want to select the *optimal* values of tuning parameters (also called **hyperparameters**). However, if we reuse the same test dataset over and over again during model selection, it will become part of our training data and thus the model will be more likely to overfit. Despite this issue, many people still use the test set for model selection, which is not a good machine learning practice.

A better way of using the holdout method for model selection is to separate the data into three parts: a training set, a validation set, and a test set. The training set is used to fit the different models, and the performance on the validation set is then used for the model selection. The advantage of having a test set that the model hasn't seen before during the training and model selection steps is that we can obtain a less biased estimate of its ability to generalize to new data. The following figure illustrates the concept of holdout cross-validation where we use a validation set to repeatedly evaluate the performance of the model after training using different parameter values. Once we are satisfied with the tuning of parameter values, we estimate the models' generalization error on the test dataset:



A disadvantage of the holdout method is that the performance estimate is sensitive to how we partition the training set into the training and validation subsets; the estimate will vary for different samples of the data. In the next subsection, we will take a look at a more robust technique for performance estimation, k-fold cross-validation, where we repeat the holdout method *k* times on *k* subsets of the training data.

K-fold cross-validation

In k-fold cross-validation, we randomly split the training dataset into *k* folds without replacement, where $k-1$ folds are used for the model training and one fold is used for testing. This procedure is repeated *k* times so that we obtain *k* models and performance estimates.

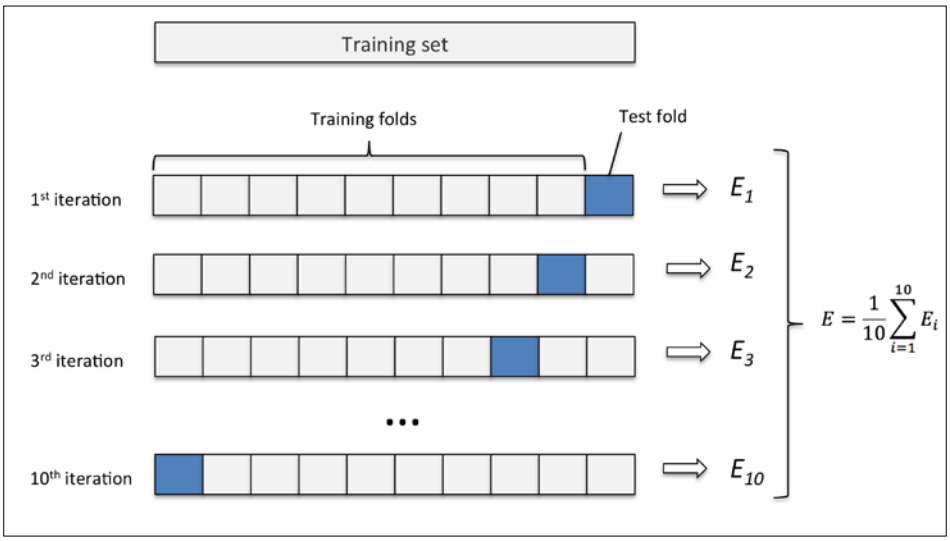
In case you are not familiar with the terms sampling *with* and *without* replacement, let's walk through a simple thought experiment. Let's assume we are playing a lottery game where we randomly draw numbers from an urn. We start with an urn that holds five unique numbers 0, 1, 2, 3, and 4, and we draw exactly one number each turn. In the first round, the chance of drawing a particular number from the urn would be 1/5. Now, in sampling without replacement, we do not put the number back into the urn after each turn. Consequently, the probability of drawing a particular number from the set of remaining numbers in the next round depends on the previous round. For example, if we have a remaining set of numbers 0, 1, 2, and 4, the chance of drawing number 0 would become 1/4 in the next turn.

However, in random sampling with replacement, we always return the drawn number to the urn so that the probabilities of drawing a particular number at each turn does not change; we can draw the same number more than once. In other words, in sampling with replacement, the samples (numbers) are independent and have a covariance zero. For example, the results from five rounds of drawing random numbers could look like this:

- Random sampling without replacement: 2, 1, 3, 4, 0
- Random sampling with replacement: 1, 3, 3, 4, 1

We then calculate the average performance of the models based on the different, independent folds to obtain a performance estimate that is less sensitive to the subpartitioning of the training data compared to the holdout method. Typically, we use k-fold cross-validation for model tuning, that is, finding the optimal hyperparameter values that yield a satisfying generalization performance. Once we have found satisfactory hyperparameter values, we can retrain the model on the complete training set and obtain a final performance estimate using the independent test set.

Since k-fold cross-validation is a resampling technique without replacement, the advantage of this approach is that each sample point will be part of a training and test dataset exactly once, which yields a lower-variance estimate of the model performance than the holdout method. The following figure summarizes the concept behind k-fold cross-validation with $k = 10$. The training data set is divided into 10 folds, and during the 10 iterations, 9 folds are used for training, and 1 fold will be used as the test set for the model evaluation. Also, the estimated performances E_i (for example, classification accuracy or error) for each fold are then used to calculate the estimated average performance E of the model:



The standard value for *k* in k-fold cross-validation is 10, which is typically a reasonable choice for most applications. However, if we are working with relatively small training sets, it can be useful to increase the number of folds. If we increase the value of *k*, more training data will be used in each iteration, which results in a lower bias towards estimating the generalization performance by averaging the individual model estimates. However, large values of *k* will also increase the runtime of the cross-validation algorithm and yield estimates with higher variance since the training folds will be more similar to each other. On the other hand, if we are working with large datasets, we can choose a smaller value for *k*, for example, $k = 5$, and still obtain an accurate estimate of the average performance of the model while reducing the computational cost of refitting and evaluating the model on the different folds.

A special case of k-fold cross validation is the **leave-one-out** (**LOO**) cross-validation method. In LOO, we set the number of folds equal to the number of training samples ($k = n$) so that only one training sample is used for testing during each iteration. This is a recommended approach for working with very small datasets.

A slight improvement over the standard k-fold cross-validation approach is stratified k-fold cross-validation, which can yield better bias and variance estimates, especially in cases of unequal class proportions, as it has been shown in a study by R. Kohavi et al. (R. Kohavi et al. *A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection*. In Ijcai, volume 14, pages 1137–1145, 1995). In stratified cross-validation, the class proportions are preserved in each fold to ensure that each fold is representative of the class proportions in the training dataset, which we will illustrate by using the `StratifiedKFold` iterator in scikit-learn:

```
>>> import numpy as np
>>> from sklearn.cross_validation import StratifiedKFold
>>> kfold = StratifiedKFold(y=y_train,
...                         n_folds=10,
...                         random_state=1)
>>> scores = []
>>> for k, (train, test) in enumerate(kfold):
...    pipe_lr.fit(X_train[train], y_train[train])
...    score = pipe_lr.score(X_train[test], y_train[test])
...    scores.append(score)
...    print('Fold: %s, Class dist.: %s, Acc: %.3f' % (k+1,
...                np.bincount(y_train[train]), score))
Fold: 1, Class dist.: [256 153], Acc: 0.891
Fold: 2, Class dist.: [256 153], Acc: 0.978
Fold: 3, Class dist.: [256 153], Acc: 0.978
Fold: 4, Class dist.: [256 153], Acc: 0.913
Fold: 5, Class dist.: [256 153], Acc: 0.935
Fold: 6, Class dist.: [257 153], Acc: 0.978
Fold: 7, Class dist.: [257 153], Acc: 0.933
Fold: 8, Class dist.: [257 153], Acc: 0.956
Fold: 9, Class dist.: [257 153], Acc: 0.978
Fold: 10, Class dist.: [257 153], Acc: 0.956
>>> print('CV accuracy: %.3f +/- %.3f' % (
...                np.mean(scores), np.std(scores)))
CV accuracy: 0.950 +/- 0.029
```

First, we initialized the `StratifiedKfold` iterator from the `sklearn.cross_validation` module with the class labels `y_train` in the training set, and specified the number of folds via the `n_folds` parameter. When we used the `kfold` iterator to loop through the `k` folds, we used the returned indices in `train` to fit the logistic regression pipeline that we set up at the beginning of this chapter. Using the `pile_lr` pipeline, we ensured that the samples were scaled properly (for instance, standardized) in each iteration. We then used the `test` indices to calculate the accuracy score of the model, which we collected in the `scores` list to calculate the average accuracy and the standard deviation of the estimate.

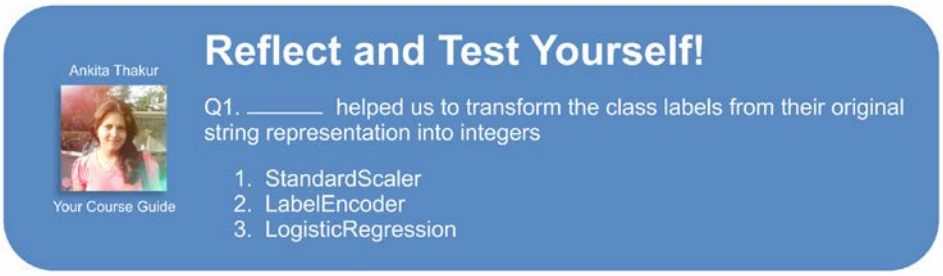
Although the previous code example was useful to illustrate how k-fold cross-validation works, scikit-learn also implements a k-fold cross-validation scorer, which allows us to evaluate our model using stratified k-fold cross-validation more efficiently:

```
>>> from sklearn.cross_validation import cross_val_score
>>> scores = cross_val_score(estimator=pipe_lr,
...                          X=X_train,
...                          y=y_train,
...                          cv=10,
...                          n_jobs=1)
>>> print('CV accuracy scores: %s' % scores)
CV accuracy scores: [ 0.89130435  0.97826087  0.97826087
                      0.91304348  0.93478261  0.97777778
                      0.93333333  0.95555556  0.97777778
                         0.95555556]
>>> print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
np.std(scores)))
CV accuracy: 0.950 +/- 0.029
```

An extremely useful feature of the `cross_val_score` approach is that we can distribute the evaluation of the different folds across multiple CPUs on our machine. If we set the `n_jobs` parameter to `1`, only one CPU will be used to evaluate the performances just like in our `StratifiedKFold` example previously. However, by setting `n_jobs=2` we could distribute the 10 rounds of cross-validation to two CPUs (if available on our machine), and by setting `n_jobs=-1`, we can use all available CPUs on our machine to do the computation in parallel.

Please note that a detailed discussion of how the variance of the generalization performance is estimated in cross-validation is beyond the scope of this book, but you can find a detailed discussion in this excellent article by M. Markatou et al (M. Markatou, H. Tian, S. Biswas, and G. M. Hripcsak. *Analysis of Variance of Cross-validation Estimators of the Generalization Error. Journal of Machine Learning Research*, 6:1127–1168, 2005).

You can also read about alternative cross-validation techniques, such as the .632 Bootstrap cross-validation method (B. Efron and R. Tibshirani. *Improvements on Cross-validation: The 632+ Bootstrap Method. Journal of the American Statistical Association*, 92(438):548–560, 1997).

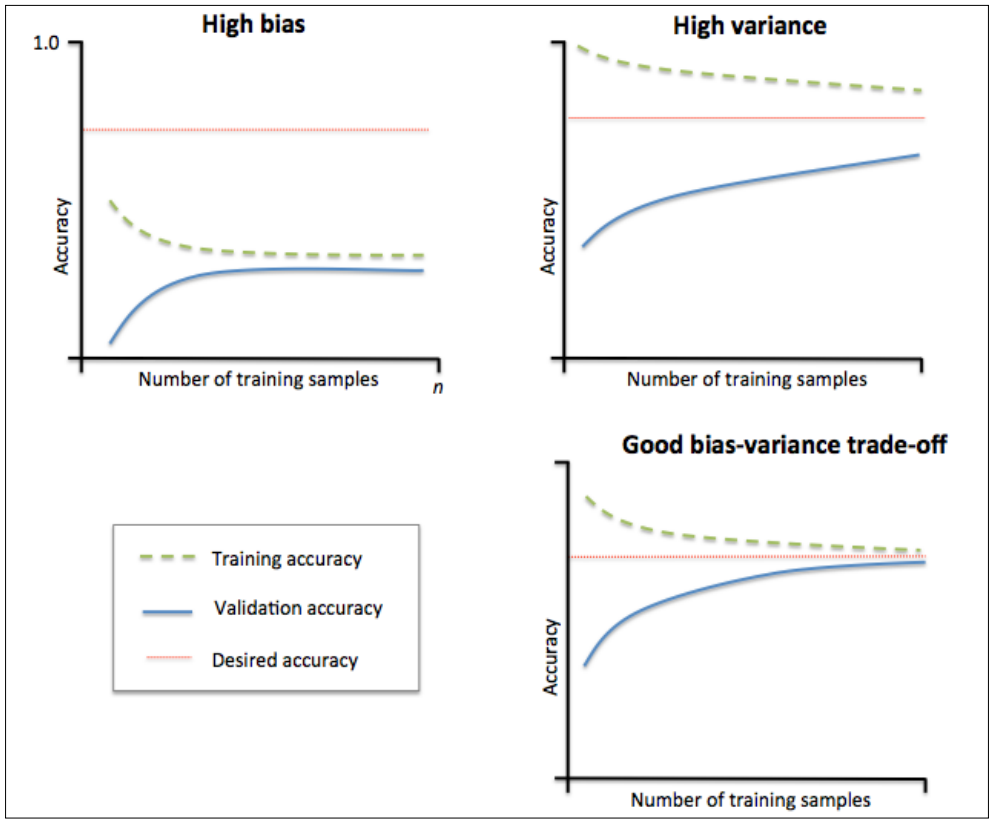


Debugging algorithms with learning and validation curves

In this section, we will take a look at two very simple yet powerful diagnostic tools that can help us to improve the performance of a learning algorithm: **learning curves** and **validation curves**. In the next subsections, we will discuss how we can use learning curves to diagnose if a learning algorithm has a problem with overfitting (high variance) or underfitting (high bias). Furthermore, we will take a look at validation curves that can help us address the common issues of a learning algorithm.

Diagnosing bias and variance problems with learning curves

If a model is too complex for a given training dataset—there are too many degrees of freedom or parameters in this model—the model tends to overfit the training data and does not generalize well to unseen data. Often, it can help to collect more training samples to reduce the degree of overfitting. However, in practice, it can often be very expensive or simply not feasible to collect more data. By plotting the model training and validation accuracies as functions of the training set size, we can easily detect whether the model suffers from high variance or high bias, and whether the collection of more data could help to address this problem. But before we discuss how to plot learning curves in sckit-learn, let's discuss those two common model issues by walking through the following illustration:



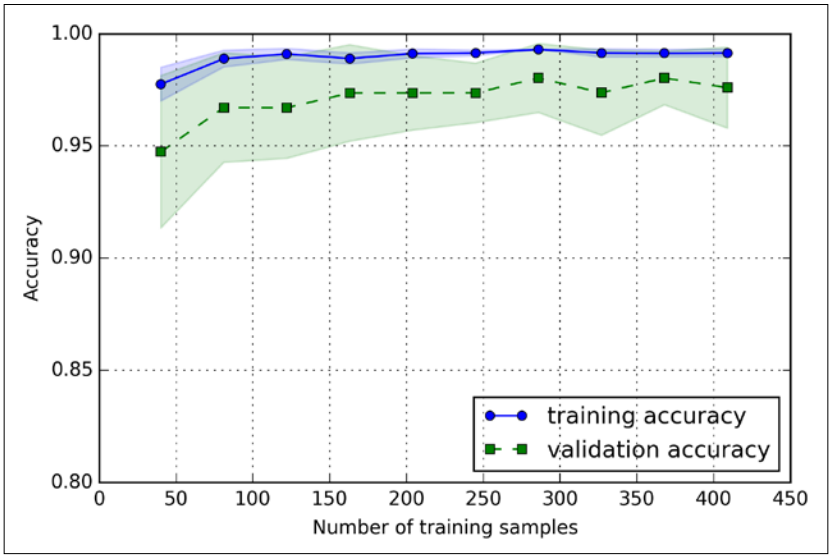
The graph in the upper-left shows a model with high bias. This model has both low training and cross-validation accuracy, which indicates that it underfits the training data. Common ways to address this issue are to increase the number of parameters of the model, for example, by collecting or constructing additional features, or by decreasing the degree of regularization, for example, in SVM or logistic regression classifiers. The graph in the upper-right shows a model that suffers from high variance, which is indicated by the large gap between the training and cross-validation accuracy. To address this problem of overfitting, we can collect more training data or reduce the complexity of the model, for example, by increasing the regularization parameter; for unregularized models, it can also help to decrease the number of features via feature selection (*Chapter 4*, *Building Good Training Sets – Data Preprocessing*) or feature extraction (*Chapter 5*, *Compressing Data via Dimensionality Reduction*). We shall note that collecting more training data decreases the chance of overfitting. However, it may not always help, for example, when the training data is extremely noisy or the model is already very close to optimal.

In the next subsection, we will see how to address those model issues using validation curves, but let's first see how we can use the learning curve function from scikit-learn to evaluate the model:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.learning_curve import learning_curve
>>> pipe_lr = Pipeline([
...           ('scl', StandardScaler()),
...           ('clf', LogisticRegression(
...                        penalty='l2', random_state=0))])
>>> train_sizes, train_scores, test_scores =\
...        learning_curve(estimator=pipe_lr,
...                       X=X_train,
...                       y=y_train,
...                       train_sizes=np.linspace(0.1, 1.0, 10),
...                       cv=10,
...                       n_jobs=1)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(train_sizes, train_mean,
...          color='blue', marker='o',
...          markersize=5,
...          label='training accuracy')
>>> plt.fill_between(train_sizes,
...                  train_mean + train_std,
...                  train_mean - train_std,
```

```
...                  alpha=0.15, color='blue')
>>> plt.plot(train_sizes, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='validation accuracy')
>>> plt.fill_between(train_sizes,
...                  test_mean + test_std,
...                  test_mean - test_std,
...                  alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xlabel('Number of training samples')
>>> plt.ylabel('Accuracy')
>>> plt.legend(loc='lower right')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

After we have successfully executed the preceding code, we will obtain the following learning curve plot:



Via the `train_sizes` parameter in the `learning_curve` function, we can control the absolute or relative number of training samples that are used to generate the learning curves. Here, we set `train_sizes=np.linspace(0.1, 1.0, 10)` to use 10 evenly spaced relative intervals for the training set sizes. By default, the `learning_curve` function uses stratified k-fold cross-validation to calculate the cross-validation accuracy, and we set $k = 10$ via the `cv` parameter. Then, we simply calculate the average accuracies from the returned cross-validated training and test scores for the different sizes of the training set, which we plotted using matplotlib's `plot` function. Furthermore, we add the standard deviation of the average accuracies to the plot using the `fill_between` function to indicate the variance of the estimate.

As we can see in the preceding learning curve plot, our model performs quite well on the test dataset. However, it may be slightly overfitting the training data indicated by a relatively small, but visible, gap between the training and cross-validation accuracy curves.

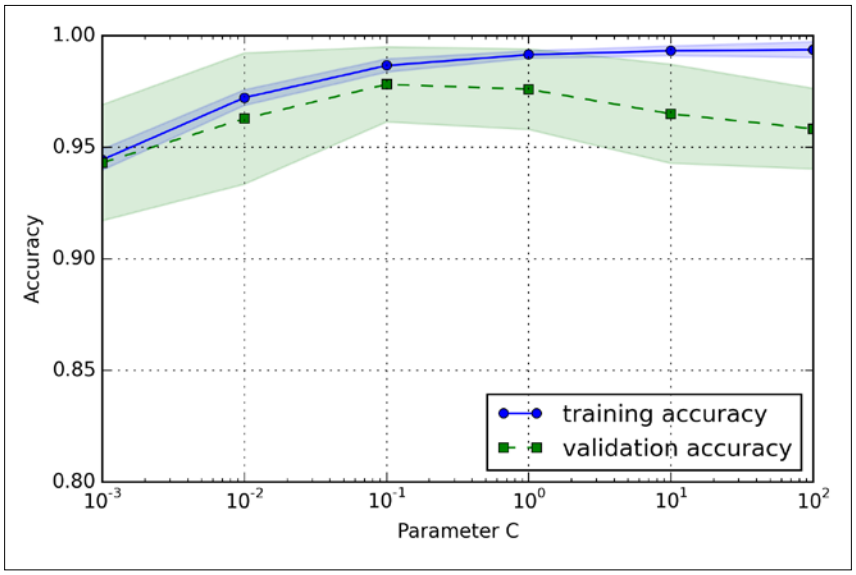
Addressing overfitting and underfitting with validation curves

Validation curves are a useful tool for improving the performance of a model by addressing issues such as overfitting or underfitting. Validation curves are related to learning curves, but instead of plotting the training and test accuracies as functions of the sample size, we vary the values of the model parameters, for example, the inverse regularization parameter `C` in logistic regression. Let's go ahead and see how we create validation curves via sckit-learn:

```
>>> from sklearn.learning_curve import validation_curve
>>> param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
>>> train_scores, test_scores = validation_curve(
...                 estimator=pipe_lr,
...                  X=X_train,
...                  y=y_train,
...                  param_name='clf__C',
...                  param_range=param_range,
...                  cv=10)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
```

```
>>> plt.plot(param_range, train_mean,
...          color='blue', marker='o',
...          markersize=5,
...          label='training accuracy')
>>> plt.fill_between(param_range, train_mean + train_std,
...                  train_mean - train_std, alpha=0.15,
...                  color='blue')
>>> plt.plot(param_range, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='validation accuracy')
>>> plt.fill_between(param_range,
...                  test_mean + test_std,
...                  test_mean - test_std,
...                  alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xscale('log')
>>> plt.legend(loc='lower right')
>>> plt.xlabel('Parameter C')
>>> plt.ylabel('Accuracy')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

Using the preceding code, we obtained the validation curve plot for the parameter C:



Similar to the `learning_curve` function, the `validation_curve` function uses stratified k-fold cross-validation by default to estimate the performance of the model if we are using algorithms for classification. Inside the `validation_curve` function, we specified the parameter that we wanted to evaluate. In this case, it is `C`, the inverse regularization parameter of the `LogisticRegression` classifier, which we wrote as `'clf__C'` to access the `LogisticRegression` object inside the scikit-learn pipeline for a specified value range that we set via the `param_range` parameter. Similar to the learning curve example in the previous section, we plotted the average training and cross-validation accuracies and the corresponding standard deviations.

Although the differences in the accuracy for varying values of `C` are subtle, we can see that the model slightly underfits the data when we increase the regularization strength (small values of `C`). However, for large values of `C`, it means lowering the strength of regularization, so the model tends to slightly overfit the data. In this case, the sweet spot appears to be around `C=0.1`.

Fine-tuning machine learning models via grid search

In machine learning, we have two types of parameters: those that are learned from the training data, for example, the weights in logistic regression, and the parameters of a learning algorithm that are optimized separately. The latter are the tuning parameters, also called hyperparameters, of a model, for example, the **regularization** parameter in logistic regression or the **depth** parameter of a decision tree.

In the previous section, we used validation curves to improve the performance of a model by tuning one of its hyperparameters. In this section, we will take a look at a powerful hyperparameter optimization technique called **grid search** that can further help to improve the performance of a model by finding the *optimal* combination of hyperparameter values.

Tuning hyperparameters via grid search

The approach of grid search is quite simple, it's a brute-force exhaustive search paradigm where we specify a list of values for different hyperparameters, and the computer evaluates the model performance for each combination of those to obtain the optimal set:

```
>>> from sklearn.grid_search import GridSearchCV
>>> from sklearn.svm import SVC
>>> pipe_svc = Pipeline([('scl', StandardScaler()),
...                      ('clf', SVC(random_state=1))])
>>> param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
>>> param_grid = [{'clf__C': param_range,
...                'clf__kernel': ['linear']},
...               {'clf__C': param_range,
...                'clf__gamma': param_range,
...                'clf__kernel': ['rbf']}]
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=10,
...                   n_jobs=-1)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.978021978022
>>> print(gs.best_params_)
{'clf__C': 0.1, 'clf__kernel': 'linear'}
```

Using the preceding code, we initialized a `GridSearchCV` object from the `sklearn.grid_search` module to train and tune a **support vector machine** (**SVM**) pipeline. We set the `param_grid` parameter of `GridSearchCV` to a list of dictionaries to specify the parameters that we'd want to tune. For the linear SVM, we only evaluated the inverse regularization parameter `C`; for the RBF kernel SVM, we tuned both the `C` and `gamma` parameter. Note that the `gamma` parameter is specific to kernel SVMs. After we used the training data to perform the grid search, we obtained the score of the best-performing model via the `best_score_` attribute and looked at its parameters, that can be accessed via the `best_params_` attribute. In this particular case, the linear SVM model with `'clf__C'= 0.1'` yielded the best k-fold cross-validation accuracy: 97.8 percent.

Finally, we will use the independent test dataset to estimate the performance of the best selected model, which is available via the `best_estimator_` attribute of the `GridSearchCV` object:

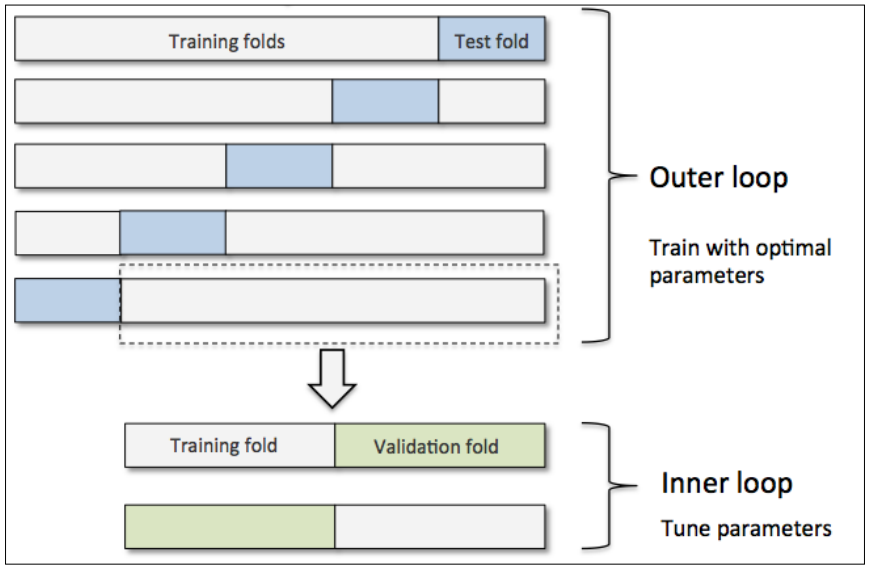
```
>>> clf = gs.best_estimator_
>>> clf.fit(X_train, y_train)
>>> print('Test accuracy: %.3f' % clf.score(X_test, y_test))
Test accuracy: 0.965
```

Although grid search is a powerful approach for finding the optimal set of parameters, the evaluation of all possible parameter combinations is also computationally very expensive. An alternative approach to sampling different parameter combinations using scikit-learn is randomized search. Using the `RandomizedSearchCV` class in scikit-learn, we can draw random parameter combinations from sampling distributions with a specified budget. More details and examples for its usage can be found at `http://scikit-learn.org/stable/modules/grid_search.html#randomized-parameter-optimization`.

Algorithm selection with nested cross-validation

Using k-fold cross-validation in combination with grid search is a useful approach for fine-tuning the performance of a machine learning model by varying its hyperparameters values as we saw in the previous subsection. If we want to select among different machine learning algorithms though, another recommended approach is nested cross-validation, and in a nice study on the bias in error estimation, Varma and Simon concluded that the true error of the estimate is almost unbiased relative to the test set when nested cross-validation is used (S. Varma and R. Simon. *Bias in Error Estimation When Using Cross-validation for Model Selection*. BMC bioinformatics, 7(1):91, 2006).

In nested cross-validation, we have an outer k-fold cross-validation loop to split the data into training and test folds, and an inner loop is used to select the model using k-fold cross-validation on the training fold. After model selection, the test fold is then used to evaluate the model performance. The following figure explains the concept of nested cross-validation with five outer and two inner folds, which can be useful for large data sets where computational performance is important; this particular type of nested cross-validation is also known as **5x2 cross-validation**:



In scikit-learn, we can perform nested cross-validation as follows:

```
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=2,
...                   n_jobs=-1)
>>> scores = cross_val_score(gs, X_train, y_train, scoring='accuracy',
cv=5)
>>> print('CV accuracy: %.3f +/- %.3f' % (
...               np.mean(scores), np.std(scores)))
CV accuracy: 0.965 +/- 0.025
```

The returned average cross-validation accuracy gives us a good estimate of what to expect if we tune the hyperparameters of a model and then use it on unseen data. For example, we can use the nested cross-validation approach to compare an SVM model to a simple decision tree classifier; for simplicity, we will only tune its depth parameter:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> gs = GridSearchCV(
...       estimator=DecisionTreeClassifier(random_state=0),
...       param_grid=[
...            {'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
...       scoring='accuracy',
...       cv=5)
>>> scores = cross_val_score(gs,
...                          X_train,
...                          y_train,
...                          scoring='accuracy',
...                          cv=2)
>>> print('CV accuracy: %.3f +/- %.3f' % (
...                      np.mean(scores), np.std(scores)))
CV accuracy: 0.921 +/- 0.029
```

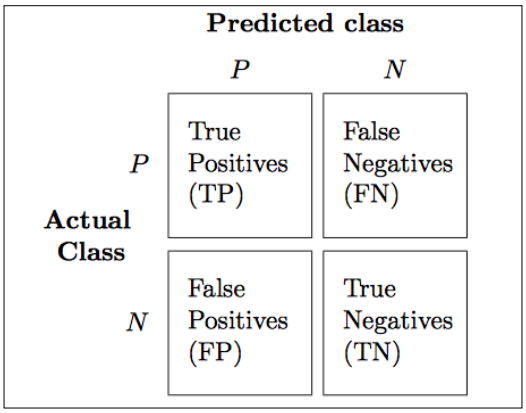
As we can see here, the nested cross-validation performance of the SVM model (97.8 percent) is notably better than the performance of the decision tree (90.8 percent). Thus, we'd expect that it might be the better choice for classifying new data that comes from the same population as this particular dataset.

Looking at different performance evaluation metrics

In the previous sections and chapters, we evaluated our models using the model accuracy, which is a useful metric to quantify the performance of a model in general. However, there are several other performance metrics that can be used to measure a model's relevance, such as **precision**, **recall**, and the **F1-score**.

Reading a confusion matrix

Before we get into the details of different scoring metrics, let's print a so-called **confusion matrix**, a matrix that lays out the performance of a learning algorithm. The confusion matrix is simply a square matrix that reports the counts of the **true positive**, **true negative**, **false positive**, and **false negative** predictions of a classifier, as shown in the following figure:



Although these metrics can be easily computed manually by comparing the true and predicted class labels, scikit-learn provides a convenient `confusion_matrix` function that we can use as follows:

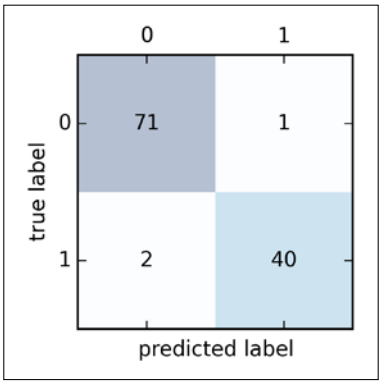
```
>>> from sklearn.metrics import confusion_matrix
>>> pipe_svc.fit(X_train, y_train)
>>> y_pred = pipe_svc.predict(X_test)
>>> confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
>>> print(confmat)
[[71  1]
 [ 2 40]]
```

The array that was returned after executing the preceding code provides us with information about the different types of errors the classifier made on the test dataset that we can map onto the confusion matrix illustration in the previous figure using matplotlib's `matshow` function:

```
>>> fig, ax = plt.subplots(figsize=(2.5, 2.5))
>>> ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
>>> for i in range(confmat.shape[0]):
...     for j in range(confmat.shape[1]):
...         ax.text(x=j, y=i,
...                 s=confmat[i, j],
...                 va='center', ha='center')
```

```
>>> plt.xlabel('predicted label')
>>> plt.ylabel('true label')
>>> plt.show()
```

Now, the confusion matrix plot as shown here should make the results a little bit easier to interpret:



Assuming that class 1 (malignant) is the positive class in this example, our model correctly classified 71 of the samples that belong to class 0 (true negatives) and 40 samples that belong to class 1 (true positives), respectively. However, our model also incorrectly misclassified 1 sample from class 0 as class 1 (false positive), and it predicted that 2 samples are benign although it is a malignant tumor (false negatives). In the next section, we will learn how we can use this information to calculate various different error metrics.

Optimizing the precision and recall of a classification model

Both the prediction **error** (**ERR**) and **accuracy** (**ACC**) provide general information about how many samples are misclassified. The error can be understood as the sum of all false predictions divided by the number of total predictions, and the accuracy is calculated as the sum of correct predictions divided by the total number of predictions, respectively:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}$$

The prediction accuracy can then be calculated directly from the error:

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$$

The **true positive rate** (**TPR**) and **false positive rate** (**FPR**) are performance metrics that are especially useful for imbalanced class problems:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

In tumor diagnosis, for example, we are more concerned about the detection of malignant tumors in order to help a patient with the appropriate treatment. However, it is also important to decrease the number of benign tumors that were incorrectly classified as malignant (false positives) to not unnecessarily concern a patient. In contrast to the FPR, the true positive rate provides useful information about the fraction of positive (or relevant) samples that were correctly identified out of the total pool of positives (**P**).

Precision (**PRE**) and **recall** (**REC**) are performance metrics that are related to those true positive and true negative rates, and in fact, recall is synonymous to the true positive rate:

$$PRE = \frac{TP}{TP + FP}$$

$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

In practice, often a combination of precision and recall is used, the so-called **F1-score**:

$$F1 = 2\frac{PRE \times REC}{PRE + REC}$$

These scoring metrics are all implemented in scikit-learn and can be imported from the `sklearn.metrics` module, as shown in the following snippet:

```
>>> from sklearn.metrics import precision_score
>>> from sklearn.metrics  import recall_score, f1_score
>>> print('Precision: %.3f' % precision_score(
...              y_true=y_test, y_pred=y_pred))
Precision: 0.976
>>> print('Recall: %.3f' % recall_score(
...              y_true=y_test, y_pred=y_pred))
Recall: 0.952
>>> print('F1: %.3f' % f1_score(
...              y_true=y_test, y_pred=y_pred))
F1: 0.964
```

Furthermore, we can use a different scoring metric other than accuracy in `GridSearch` via the scoring parameter. A complete list of the different values that are accepted by the scoring parameter can be found at `http://scikit-learn.org/stable/modules/model_evaluation.html`.

Remember that the positive class in scikit-learn is the class that is labeled as class 1. If we want to specify a different *positive label*, we can construct our own scorer via the `make_scorer` function, which we can then directly provide as an argument to the scoring parameter in `GridSearchCV`:

```
>>> from sklearn.metrics import make_scorer, f1_score
>>> scorer = make_scorer(f1_score, pos_label=0)
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring=scorer,
...                   cv=10)
```

Plotting a receiver operating characteristic

Receiver operator characteristic (**ROC**) graphs are useful tools for selecting models for classification based on their performance with respect to the false positive and true positive rates, which are computed by shifting the decision threshold of the classifier. The diagonal of an ROC graph can be interpreted as random guessing, and classification models that fall below the diagonal are considered as worse than random guessing. A perfect classifier would fall into the top-left corner of the graph with a true positive rate of 1 and a false positive rate of 0. Based on the ROC curve, we can then compute the so-called **area under the curve** (**AUC**) to characterize the performance of a classification model.

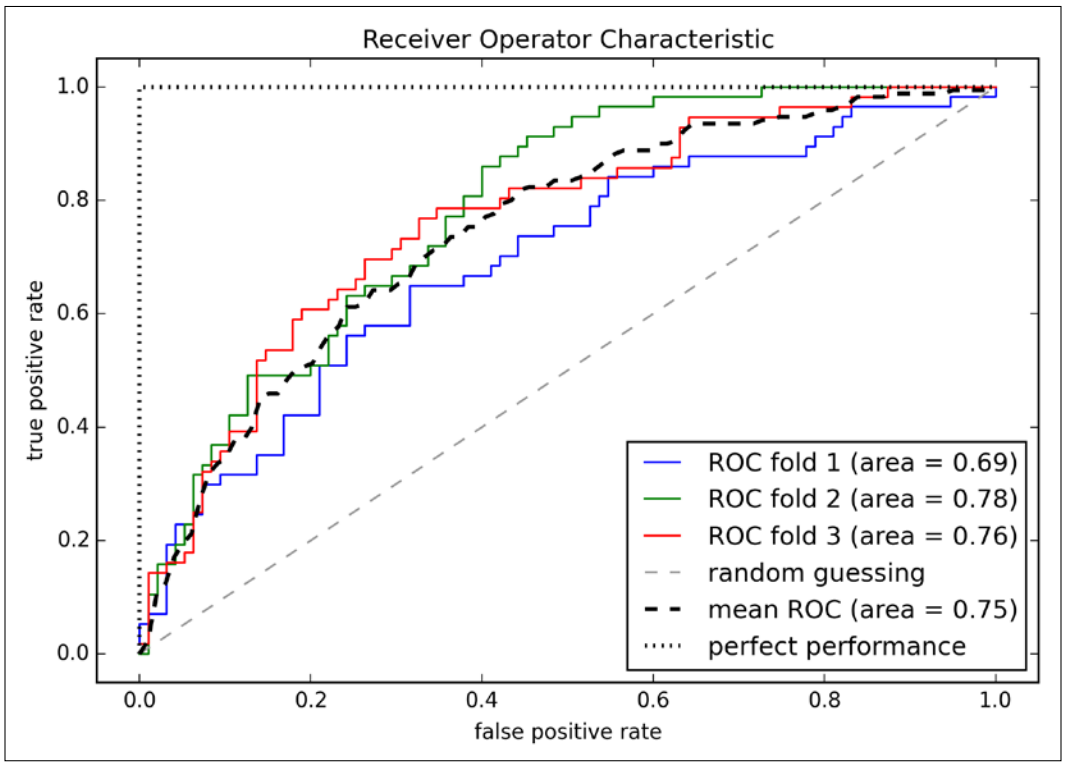
Similar to ROC curves, we can compute **precision-recall curves** for the different probability thresholds of a classifier. A function for plotting those precision-recall curves is also implemented in scikit-learn and is documented at `http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html`.

By executing the following code example, we will plot an ROC curve of a classifier that only uses two features from the Breast Cancer Wisconsin dataset to predict whether a tumor is benign or malignant. Although we are going to use the same logistic regression pipeline that we defined previously, we are making the classification task more challenging for the classifier so that the resulting ROC curve becomes visually more interesting. For similar reasons, we are also reducing the number of folds in the `StratifiedKFold` validator to three. The code is as follows:

```
>>> from sklearn.metrics import roc_curve, auc
>>> from scipy import interp
>>> pipe_lr = Pipeline([('scl', StandardScaler()),
...                     ('pca', PCA(n_components=2)),
...                     ('clf', LogisticRegression(penalty='l2',
...                                                random_state=0,
...                                                C=100.0))])
>>> X_train2 = X_train[:, [4, 14]]
>>> cv = StratifiedKFold(y_train,
...                      n_folds=3,
...                      random_state=1)
>>> fig = plt.figure(figsize=(7, 5))
>>> mean_tpr = 0.0
>>> mean_fpr = np.linspace(0, 1, 100)
>>> all_tpr = []

>>> for i, (train, test) in enumerate(cv):
...     probas = pipe_lr.fit(X_train2[train],
>>> y_train[train]).predict_proba(X_train2[test])
...     fpr, tpr, thresholds = roc_curve(y_train[test],
```

```
...                                     probas[:, 1],
...                                     pos_label=1)
...     mean_tpr += interp(mean_fpr, fpr, tpr)
...     mean_tpr[0] = 0.0
...     roc_auc = auc(fpr, tpr)
...     plt.plot(fpr,
...              tpr,
...              lw=1,
...              label='ROC fold %d (area = %0.2f)'
...                    % (i+1, roc_auc))
>>> plt.plot([0, 1],
...          [0, 1],
...          linestyle='--',
...          color=(0.6, 0.6, 0.6),
...          label='random guessing')
>>> mean_tpr /= len(cv)
>>> mean_tpr[-1] = 1.0
>>> mean_auc = auc(mean_fpr, mean_tpr)
>>> plt.plot(mean_fpr, mean_tpr, 'k--',
...          label='mean ROC (area = %0.2f)' % mean_auc, lw=2)
>>> plt.plot([0, 0, 1],
...          [0, 1, 1],
...          lw=2,
...          linestyle=':',
...          color='black',
...          label='perfect performance')
>>> plt.xlim([-0.05, 1.05])
>>> plt.ylim([-0.05, 1.05])
>>> plt.xlabel('false positive rate')
>>> plt.ylabel('true positive rate')
>>> plt.title('Receiver Operator Characteristic')
>>> plt.legend(loc="lower right")
>>> plt.show()
```

In the preceding code example, we used the already familiar `StratifiedKFold` class from scikit-learn and calculated the ROC performance of the `LogisticRegression` classifier in our `pipe_lr` pipeline using the `roc_curve` function from the `sklearn.metrics` module separately for each iteration. Furthermore, we interpolated the average ROC curve from the three folds via the `interp` function that we imported from SciPy and calculated the area under the curve via the `auc` function. The resulting ROC curve indicates that there is a certain degree of variance between the different folds, and the average ROC AUC (0.75) falls between a perfect score (1.0) and random guessing (0.5):



If we are just interested in the ROC AUC score, we could also directly import the `roc_auc_score` function from the `sklearn.metrics` submodule. The following code calculates the classifier's ROC AUC score on the independent test dataset after fitting it on the two-feature training set:

```
>>> pipe_lr = pipe_lr.fit(X_train2, y_train)
>>> y_pred2 = pipe_lr.predict(X_test[:, [4, 14]])
```

```
>>> from sklearn.metrics import roc_auc_score
>>> from sklearn.metrics import accuracy_score
>>> print('ROC AUC: %.3f' % roc_auc_score(
...         y_true=y_test, y_score=y_pred2))
ROC AUC: 0.662

>>> print('Accuracy: %.3f' % accuracy_score(
...         y_true=y_test, y_pred=y_pred2))
Accuracy: 0.711
```

Reporting the performance of a classifier as the ROC AUC can yield further insights in a classifier's performance with respect to imbalanced samples. However, while the accuracy score can be interpreted as a single cut-off point on a ROC curve, A. P. Bradley showed that the ROC AUC and accuracy metrics mostly agree with each other (A. P. Bradley. *The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms*. Pattern recognition, 30(7):1145–1159, 1997).

The scoring metrics for multiclass classification

The scoring metrics that we discussed in this section are specific to binary classification systems. However, scikit-learn also implements **macro** and **micro** averaging methods to extend those scoring metrics to multiclass problems via **One vs. All** (**OvA**) classification. The micro-average is calculated from the individual true positives, true negatives, false positives, and false negatives of the system. For example, the micro-average of the precision score in a k-class system can be calculated as follows:

$$PRE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}$$

The macro-average is simply calculated as the average scores of the different systems:

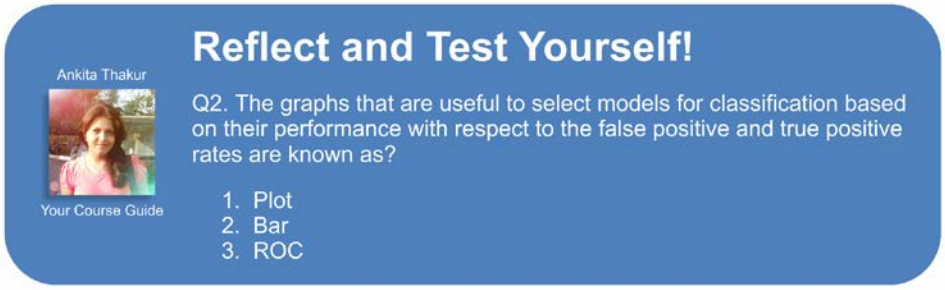
$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$

Micro-averaging is useful if we want to weight each instance or prediction equally, whereas macro-averaging weights all classes equally to evaluate the overall performance of a classifier with regard to the most frequent class labels.

If we are using binary performance metrics to evaluate multiclass classification models in scikit-learn, a normalized or weighted variant of the macro-average is used by default. The weighted macro-average is calculated by weighting the score of each class label by the number of true instances when calculating the average. The weighted macro-average is useful if we are dealing with class imbalances, that is, different numbers of instances for each label.

While the weighted macro-average is the default for multiclass problems in scikit-learn, we can specify the averaging method via the `average` parameter inside the different scoring functions that we import from the `sklearn.metrics` module, for example, the `precision_score` or `make_scorer` functions:

```
>>> pre_scorer = make_scorer(score_func=precision_score,
...                          pos_label=1,
...                          greater_is_better=True,
...                          average='micro')
```



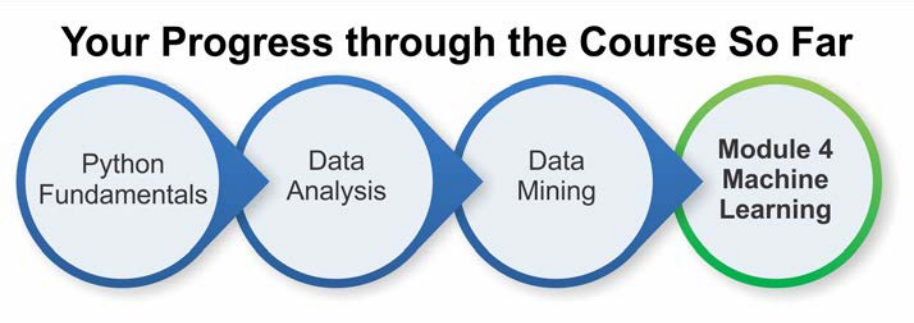
Summary of Module 4 Chapter 6

Ankita Thakur

Your Course Guide

In the beginning of this chapter, we discussed how to chain different transformation techniques and classifiers in convenient model pipelines that helped us to train and evaluate machine learning models more efficiently. We then used those pipelines to perform k-fold cross-validation, one of the essential techniques for model selection and evaluation. Using k-fold cross-validation, we plotted learning and validation curves to diagnose the common problems of learning algorithms, such as overfitting and underfitting. Using grid search, we further fine-tuned our model. We concluded this chapter by looking at a confusion matrix and various different performance metrics that can be useful to further optimize a model's performance for a specific problem task. Now, we should be well-equipped with the essential techniques to build supervised machine learning models for classification successfully.

In the next chapter, we will take a look at ensemble methods, methods that allow us to combine multiple models and classification algorithms to boost the predictive performance of a machine learning system even further.



7
Combining Different Models for Ensemble Learning

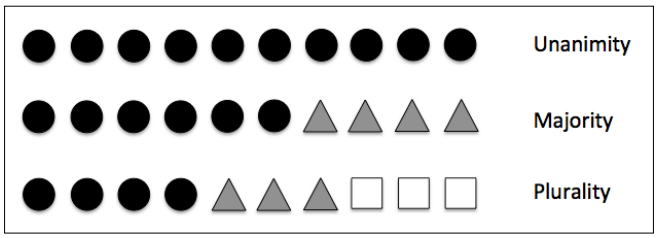
In the previous chapter, we focused on the best practices for tuning and evaluating different models for classification. In this chapter, we will build upon these techniques and explore different methods for constructing a set of classifiers that can often have a better predictive performance than any of its individual members. You will learn how to:

- Make predictions based on majority voting
- Reduce overfitting by drawing random combinations of the training set with repetition
- Build powerful models from *weak learners* that learn from their mistakes

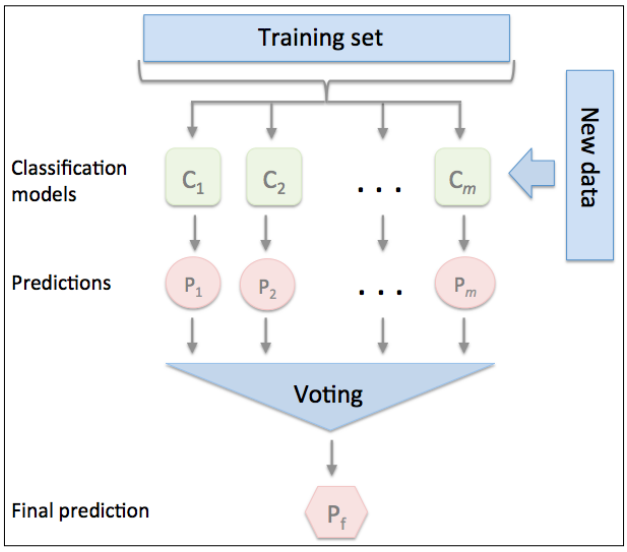
Learning with ensembles

The goal behind **ensemble methods** is to combine different classifiers into a meta-classifier that has a better generalization performance than each individual classifier alone. For example, assuming that we collected predictions from 10 experts, ensemble methods would allow us to strategically combine these predictions by the 10 experts to come up with a prediction that is more accurate and robust than the predictions by each individual expert. As we will see later in this chapter, there are several different approaches for creating an ensemble of classifiers. In this section, we will introduce a basic perception about how ensembles work and why they are typically recognized for yielding a good generalization performance.

In this chapter, we will focus on the most popular ensemble methods that use the **majority voting** principle. Majority voting simply means that we select the class label that has been predicted by the majority of classifiers, that is, received more than 50 percent of the votes. Strictly speaking, the term **majority vote** refers to binary class settings only. However, it is easy to generalize the majority voting principle to multi-class settings, which is called **plurality voting**. Here, we select the class label that received the most votes (mode). The following diagram illustrates the concept of majority and plurality voting for an ensemble of 10 classifiers where each unique symbol (triangle, square, and circle) represents a unique class label:



Using the training set, we start by training m different classifiers ($C_1,\ldots,C_m$). Depending on the technique, the ensemble can be built from different classification algorithms, for example, decision trees, support vector machines, logistic regression classifiers, and so on. Alternatively, we can also use the same base classification algorithm fitting different subsets of the training set. One prominent example of this approach would be the random forest algorithm, which combines different decision tree classifiers. The following diagram illustrates the concept of a general ensemble approach using majority voting:



To predict a class label via a simple majority or plurality voting, we combine the predicted class labels of each individual classifier C_j and select the class label $\hat{y}$ that received the most votes:

$$\hat{y} = mode\{C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \ldots, C_m(\boldsymbol{x})\}$$

For example, in a binary classification task where $class1 = -1$ and $class2 = +1$, we can write the majority vote prediction as follows:

$$C(\boldsymbol{x}) = sign\left[\sum_j^m C_j(\boldsymbol{x})\right] = \begin{cases} 1 \ \ if \sum_i C_j(\boldsymbol{x}) \geq 0 \\ -1 \ otherwise \end{cases}$$

To illustrate why ensemble methods can work better than individual classifiers alone, let's apply the simple concepts of combinatorics. For the following example, we make the assumption that all *n* base classifiers for a binary classification task have an equal error rate ε. Furthermore, we assume that the classifiers are independent and the error rates are not correlated. Under those assumptions, we can simply express the error probability of an ensemble of base classifiers as a probability mass function of a binomial distribution:

$$P(y \geq k) = \sum_k^n \left\langle {n \atop k} \right\rangle \varepsilon^k (1-\varepsilon)^{n-k} = \varepsilon_{ensemble}$$

Here, $\left\langle {n \atop k} \right\rangle$ is the binomial coefficient *n choose k*. In other words, we compute the probability that the prediction of the ensemble is wrong. Now let's take a look at a more concrete example of 11 base classifiers ($n = 11$) with an error rate of 0.25 ($\varepsilon = 0.25$):

$$P(y \geq k) = \sum_{k=6}^{11} \left\langle {11 \atop k} \right\rangle 0.25^k (1-\varepsilon)^{11-k} = 0.034$$

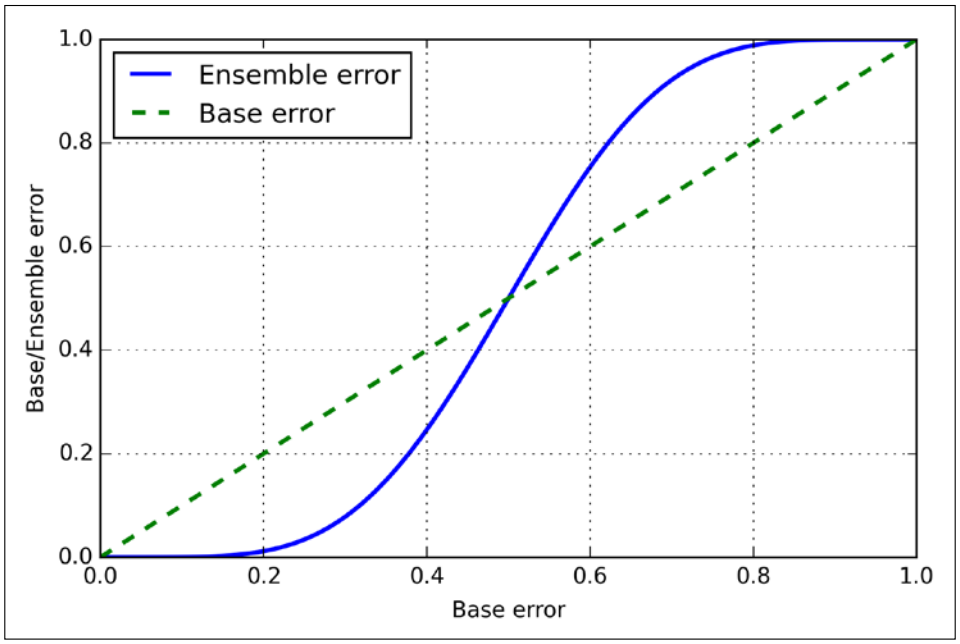
As we can see, the error rate of the ensemble (0.034) is much lower than the error rate of each individual classifier (0.25) if all the assumptions are met. Note that, in this simplified illustration, a 50-50 split by an even number of classifiers *n* is treated as an error, whereas this is only true half of the time. To compare such an idealistic ensemble classifier to a base classifier over a range of different base error rates, let's implement the probability mass function in Python:

```
>>> from scipy.misc import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
...     k_start = math.ceil(n_classifier / 2.0)
...     probs = [comb(n_classifier, k) *
...              error**k *
...              (1-error)**(n_classifier - k)
...              for k in range(k_start, n_classifier + 1)]
...     return sum(probs)
>>> ensemble_error(n_classifier=11, error=0.25)
0.034327507019042969
```

After we've implemented the `ensemble_error` function, we can compute the ensemble error rates for a range of different base errors from 0.0 to 1.0 to visualize the relationship between ensemble and base errors in a line graph:

```
>>> import numpy as np
>>> error_range = np.arange(0.0, 1.01, 0.01)
>>> ens_errors = [ensemble_error(n_classifier=11, error=error)
...               for error in error_range]
>>> import matplotlib.pyplot as plt
>>> plt.plot(error_range, ens_errors,
...          label='Ensemble error',
...          linewidth=2)
>>> plt.plot(error_range, error_range,
...          linestyle='--', label='Base error',
...          linewidth=2)
>>> plt.xlabel('Base error')
>>> plt.ylabel('Base/Ensemble error')
>>> plt.legend(loc='upper left')
>>> plt.grid()
>>> plt.show()
```

As we can see in the resulting plot, the error probability of an ensemble is always better than the error of an individual base classifier as long as the base classifiers perform better than random guessing ($\varepsilon < 0.5$). Note that the *y*-axis depicts the base error (dotted line) as well as the ensemble error (continuous line):



Implementing a simple majority vote classifier

After the short introduction to ensemble learning in the previous section, let's start with a warm-up exercise and implement a simple ensemble classifier for majority voting in Python. Although the following algorithm also generalizes to multi-class settings via plurality voting, we will use the term *majority voting* for simplicity as is also often done in literature.

The algorithm that we are going to implement will allow us to combine different classification algorithms associated with individual weights for confidence. Our goal is to build a stronger meta-classifier that balances out the individual classifiers' weaknesses on a particular dataset. In more precise mathematical terms, we can write the weighted majority vote as follows:

$$\hat{y} = \arg\max_{i} \sum_{j=1}^{m} w_j \chi_A \left(C_j \left(\boldsymbol{x} \right) = i \right)$$

Here, w_j is a weight associated with a base classifier, C_j, $\hat{y}$ is the predicted class label of the ensemble, χ_A (Greek *chi*) is the characteristic function $[C_j(\boldsymbol{x}) = i \in A]$, and A is the set of unique class labels. For equal weights, we can simplify this equation and write it as follows:

$$\hat{y} = mode\{C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \ldots, C_m(\boldsymbol{x})\}$$

To better understand the concept of *weighting*, we will now take a look at a more concrete example. Let's assume that we have an ensemble of three base classifiers C_j ($j \in \{0,1\}$) and want to predict the class label of a given sample instance x. Two out of three base classifiers predict the class label 0, and one C_3 predicts that the sample belongs to class 1. If we weight the predictions of each base classifier equally, the majority vote will predict that the sample belongs to class 0:

$$C_1(x) \rightarrow 0,\ C_2(x) \rightarrow 0,\ C_3(x) \rightarrow 1$$

$$\hat{y} = mode\{0, 0, 1\} = 0$$

Now let's assign a weight of 0.6 to C_3 and weight C_1 and C_2 by a coefficient of 0.2, respectively.

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j \chi_A (C_j(\boldsymbol{x}) = i)$$

$$= \arg\max_i [0.2 \times i_0 + 0.2 \times i_0 + 0.6 \times i_1] = 1$$

More intuitively, since $3 \times 0.2 = 0.6$, we can say that the prediction made by C_3 has three times more weight than the predictions by C_1 or C_2, respectively. We can write this as follows:

$$\hat{y} = mode\{0, 0, 1, 1, 1\} = 1$$

To translate the concept of the weighted majority vote into Python code, we can use NumPy's convenient `argmax` and `bincount` functions:

```
>>> import numpy as np
>>> np.argmax(np.bincount([0, 0, 1],
...           weights=[0.2, 0.2, 0.6]))
1
```

As discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, certain classifiers in scikit-learn can also return the probability of a predicted class label via the `predict_proba` method. Using the predicted class probabilities instead of the class labels for majority voting can be useful if the classifiers in our ensemble are well calibrated. The modified version of the majority vote for predicting class labels from probabilities can be written as follows:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij}$$

Here, p_{ij} is the predicted probability of the *jth* classifier for class label *i*.

To continue with our previous example, let's assume that we have a binary classification problem with class labels $i \in \{0,1\}$ and an ensemble of three classifiers C_j ($j \in \{1,2,3\}$). Let's assume that the classifier C_j returns the following class membership probabilities for a particular sample $\boldsymbol{x}$:

$$C_1(\boldsymbol{x}) \rightarrow [0.9, 0.1],\ C_2(\boldsymbol{x}) \rightarrow [0.8, 0.2],\ C_3(\boldsymbol{x}) \rightarrow [0.4, 0.6]$$

We can then calculate the individual class probabilities as follows:

$$p(i_0 \mid \boldsymbol{x}) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$

$$p(i_1 \mid \boldsymbol{x}) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.06 = 0.42$$

$$\hat{y} = \arg\max_i \left[p(i_0 \mid \boldsymbol{x}), p(i_1 \mid \boldsymbol{x}) \right] = 0$$

To implement the weighted majority vote based on class probabilities, we can again make use of NumPy using `numpy.average` and `np.argmax`:

```
>>> ex = np.array([[0.9, 0.1],
...                [0.8, 0.2],
...                [0.4, 0.6]])
>>> p = np.average(ex, axis=0, weights=[0.2, 0.2, 0.6])
>>> p
array([ 0.58,  0.42])
>>> np.argmax(p)
0
```

Putting everything together, let's now implement a `MajorityVoteClassifier` in Python:

```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.externals import six
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator


class MajorityVoteClassifier(BaseEstimator,
                             ClassifierMixin):
    """ A majority vote ensemble classifier

    Parameters
    ----------
    classifiers : array-like, shape = [n_classifiers]
      Different classifiers for the ensemble

    vote : str, {'classlabel', 'probability'}
      Default: 'classlabel'
      If 'classlabel' the prediction is based on
      the argmax of class labels. Else if
      'probability', the argmax of the sum of
      probabilities is used to predict the class label
      (recommended for calibrated classifiers).

    weights : array-like, shape = [n_classifiers]
      Optional, default: None
      If a list of `int` or `float` values are
```

```
  provided, the classifiers are weighted by
  importance; Uses uniform weights if `weights=None`.

"""
def __init__(self, classifiers,
             vote='classlabel', weights=None):

    self.classifiers = classifiers
    self.named_classifiers = {key: value for
                              key, value in
                              _name_estimators(classifiers)}
    self.vote = vote
    self.weights = weights

def fit(self, X, y):
    """ Fit classifiers.

    Parameters
    ----------
    X : {array-like, sparse matrix},
        shape = [n_samples, n_features]
        Matrix of training samples.

    y : array-like, shape = [n_samples]
        Vector of target class labels.

    Returns
    -------
    self : object

    """
    # Use LabelEncoder to ensure class labels start
    # with 0, which is important for np.argmax
    # call in self.predict
    self.lablenc_ = LabelEncoder()
    self.lablenc_.fit(y)
    self.classes_ = self.lablenc_.classes_
    self.classifiers_ = []
    for clf in self.classifiers:
        fitted_clf = clone(clf).fit(X,
                          self.lablenc_.transform(y))
        self.classifiers_.append(fitted_clf)
    return self
```

I added a lot of comments to the code to better understand the individual parts. However, before we implement the remaining methods, let's take a quick break and discuss some of the code that may look confusing at first. We used the parent classes `BaseEstimator` and `ClassifierMixin` to get some base functionality *for free*, including the methods `get_params` and `set_params` to set and return the classifier's parameters as well as the `score` method to calculate the prediction accuracy, respectively. Also note that we imported `six` to make the `MajorityVoteClassifier` compatible with Python 2.7.

Next we will add the `predict` method to predict the class label via majority vote based on the class labels if we initialize a new `MajorityVoteClassifier` object with `vote='classlabel'`. Alternatively, we will be able to initialize the ensemble classifier with `vote='probability'` to predict the class label based on the class membership probabilities. Furthermore, we will also add a `predict_proba` method to return the average probabilities, which is useful to compute the **Receiver Operator Characteristic area under the curve** (**ROC AUC**).

```
    def predict(self, X):
        """ Predict class labels for X.

        Parameters
        ----------
        X : {array-like, sparse matrix},
            Shape = [n_samples, n_features]
            Matrix of training samples.

        Returns
        ----------
        maj_vote : array-like, shape = [n_samples]
            Predicted class labels.

        """
        if self.vote == 'probability':
            maj_vote = np.argmax(self.predict_proba(X),
                                 axis=1)
        else:  # 'classlabel' vote

            #  Collect results from clf.predict calls
            predictions = np.asarray([clf.predict(X)
                                      for clf in
                                      self.classifiers_]).T

            maj_vote = np.apply_along_axis(
                           lambda x:
                           np.argmax(np.bincount(x,
```

```
                                        weights=self.weights)),
                        axis=1,
                        arr=predictions)
        maj_vote = self.lablenc_.inverse_transform(maj_vote)
        return maj_vote

    def predict_proba(self, X):
        """ Predict class probabilities for X.

        Parameters
        ----------
        X : {array-like, sparse matrix},
            shape = [n_samples, n_features]
            Training vectors, where n_samples is
            the number of samples and
            n_features is the number of features.

        Returns
        ----------
        avg_proba : array-like,
            shape = [n_samples, n_classes]
            Weighted average probability for
            each class per sample.

        """
        probas = np.asarray([clf.predict_proba(X)
                             for clf in self.classifiers_])
        avg_proba = np.average(probas,
                               axis=0, weights=self.weights)
        return avg_proba

    def get_params(self, deep=True):
        """ Get classifier parameter names for GridSearch"""
        if not deep:
            return super(MajorityVoteClassifier,
                         self).get_params(deep=False)
        else:
            out = self.named_classifiers.copy()
            for name, step in\
                    six.iteritems(self.named_classifiers):
                for key, value in six.iteritems(
                        step.get_params(deep=True)):
                    out['%s__%s' % (name, key)] = value
            return out
```

Also, note that we defined our own modified version of the `get_params` methods to use the `_name_estimators` function in order to access the parameters of individual classifiers in the ensemble. This may look a little bit complicated at first, but it will make perfect sense when we use grid search for hyperparameter-tuning in later sections.

Although our `MajorityVoteClassifier` implementation is very useful for demonstration purposes, I also implemented a more sophisticated version of the majority vote classifier in scikit-learn. It will become available as `sklearn.ensemble.VotingClassifier` in the next release version (v0.17).

Combining different algorithms for classification with majority vote

Now it is about time to put the `MajorityVoteClassifier` that we implemented in the previous section into action. But first, let's prepare a dataset that we can test it on. Since we are already familiar with techniques to load datasets from CSV files, we will take a shortcut and load the **Iris** dataset from scikit-learn's dataset module. Furthermore, we will only select two features, **sepal width** and **petal length**, to make the classification task more challenging. Although our `MajorityVoteClassifier` generalizes to multiclass problems, we will only classify flower samples from the two classes, **Iris-Versicolor** and **Iris-Virginica**, to compute the **ROC AUC**. The code is as follows:

```
>>> from sklearn import datasets
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.preprocessing import LabelEncoder
>>> iris = datasets.load_iris()
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

Note that scikit-learn uses the `predict_proba` method (if applicable) to compute the ROC AUC score. In *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, we saw how the class probabilities are computed in logistic regression models. In decision trees, the probabilities are calculated from a frequency vector that is created for each node at training time. The vector collects the frequency values of each class label computed from the class label distribution at that node. Then the frequencies are normalized so that they sum up to 1. Similarly, the class labels of the k-nearest neighbors are aggregated to return the normalized class label frequencies in the k-nearest neighbors algorithm. Although the normalized probabilities returned by both the decision tree and k-nearest neighbors classifier may look similar to the probabilities obtained from a logistic regression model, we have to be aware that these are actually not derived from probability mass functions.

Next we split the Iris samples into 50 percent training and 50 percent test data:

```
>>> X_train, X_test, y_train, y_test =\
...        train_test_split(X, y,
...                         test_size=0.5,
...                         random_state=1)
```

Using the training dataset, we now will train three different classifiers—a logistic regression classifier, a decision tree classifier, and a k-nearest neighbors classifier—and look at their individual performances via a 10-fold cross-validation on the training dataset before we combine them into an ensemble classifier:

```
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> clf1 = LogisticRegression(penalty='l2',
...                           C=0.001,
...                           random_state=0)
>>> clf2 = DecisionTreeClassifier(max_depth=1,
...                               criterion='entropy',
...                               random_state=0)
>>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                             p=2,
...                             metric='minkowski')
>>> pipe1 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf1]])
```

```
>>> pipe3 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf3]])
>>> clf_labels = ['Logistic Regression', 'Decision Tree', 'KNN']
>>> print('10-fold cross validation:\n')
>>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
>>>                              X=X_train,
>>>                              y=y_train,
>>>                              cv=10,
>>>                              scoring='roc_auc')
>>>     print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
...                % (scores.mean(), scores.std(), label))
```

The output that we receive, as shown in the following snippet, shows that the predictive performances of the individual classifiers are almost equal:

```
10-fold cross validation:

ROC AUC: 0.92 (+/- 0.20) [Logistic Regression]
ROC AUC: 0.92 (+/- 0.15) [Decision Tree]
ROC AUC: 0.93 (+/- 0.10) [KNN]
```

You may be wondering why we trained the logistic regression and k-nearest neighbors classifier as part of a **pipeline**. The reason behind it is that, as discussed in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, both the logistic regression and k-nearest neighbors algorithms (using the Euclidean distance metric) are not scale-invariant in contrast with decision trees. Although the Iris features are all measured on the same scale (cm), it is a good habit to work with standardized features.

Now let's move on to the more exciting part and combine the individual classifiers for majority rule voting in our `MajorityVoteClassifier`:

```
>>> mv_clf = MajorityVoteClassifier(
...                  classifiers=[pipe1, clf2, pipe3])
>>> clf_labels += ['Majority Voting']
>>> all_clf = [pipe1, clf2, pipe3, mv_clf]
>>> for clf, label in zip(all_clf, clf_labels):
...     scores = cross_val_score(estimator=clf,
...                              X=X_train,
...                              y=y_train,
...                              cv=10,
...                              scoring='roc_auc')
...     print("Accuracy: %0.2f (+/- %0.2f) [%s]"
...                % (scores.mean(), scores.std(), label))
```

```
ROC AUC: 0.92 (+/- 0.20) [Logistic Regression]
ROC AUC: 0.92 (+/- 0.15) [Decision Tree]
ROC AUC: 0.93 (+/- 0.10) [KNN]
ROC AUC: 0.97 (+/- 0.10) [Majority Voting]
```

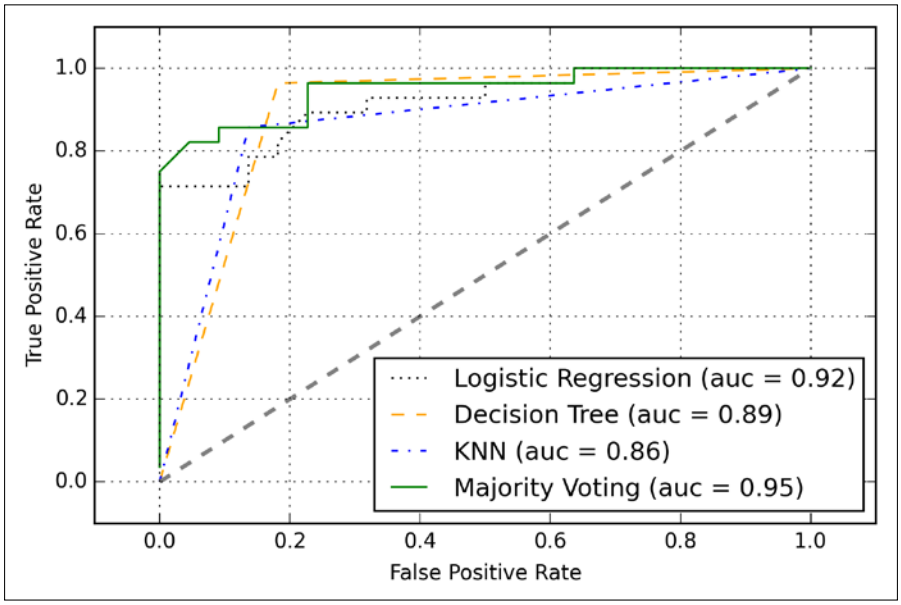
As we can see, the performance of the `MajorityVotingClassifier` has substantially improved over the individual classifiers in the 10-fold cross-validation evaluation.

Evaluating and tuning the ensemble classifier

In this section, we are going to compute the ROC curves from the test set to check if the `MajorityVoteClassifier` generalizes well to unseen data. We should remember that the test set is not to be used for model selection; its only purpose is to report an unbiased estimate of the generalization performance of a classifier system. The code is as follows:

```
>>> from sklearn.metrics import roc_curve
>>> from sklearn.metrics import auc
>>> colors = ['black', 'orange', 'blue', 'green']
>>> linestyles = [':', '--', '-.', '-']
>>> for clf, label, clr, ls \
...         in zip(all_clf, clf_labels, colors, linestyles):
...     # assuming the label of the positive class is 1
...     y_pred = clf.fit(X_train,
...                      y_train).predict_proba(X_test)[:, 1]
...     fpr, tpr, thresholds = roc_curve(y_true=y_test,
...                                      y_score=y_pred)
...     roc_auc = auc(x=fpr, y=tpr)
...     plt.plot(fpr, tpr,
...              color=clr,
...              linestyle=ls,
...              label='%s (auc = %0.2f)' % (label, roc_auc))
>>> plt.legend(loc='lower right')
>>> plt.plot([0, 1], [0, 1],
...          linestyle='--',
...          color='gray',
...          linewidth=2)
>>> plt.xlim([-0.1, 1.1])
>>> plt.ylim([-0.1, 1.1])
>>> plt.grid()
>>> plt.xlabel('False Positive Rate')
>>> plt.ylabel('True Positive Rate')
>>> plt.show()
```

As we can see in the resulting ROC, the ensemble classifier also performs well on the test set (*ROC AUC = 0.95*), whereas the k-nearest neighbors classifier seems to be overfitting the training data (training *ROC AUC = 0.93*, test *ROC AUC = 0.86*):

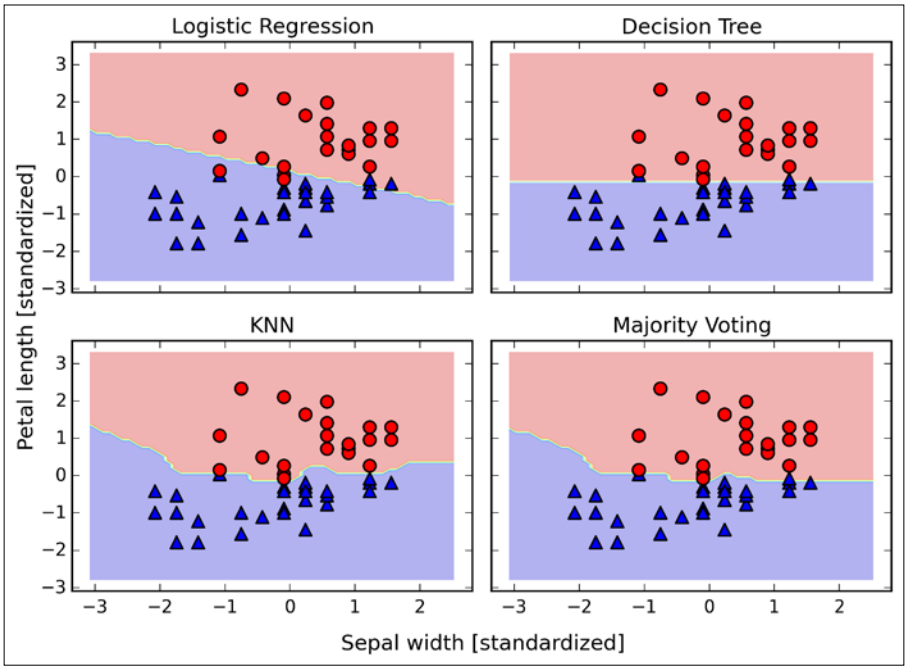


Since we only selected two features for the classification examples, it would be interesting to see what the decision region of the ensemble classifier actually looks like. Although it is not necessary to standardize the training features prior to model fitting because our logistic regression and k-nearest neighbors pipelines will automatically take care of this, we will standardize the training set so that the decision regions of the decision tree will be on the same scale for visual purposes. The code is as follows:

```
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> from itertools import product
>>> x_min = X_train_std[:, 0].min() - 1
>>> x_max = X_train_std[:, 0].max() + 1
>>> y_min = X_train_std[:, 1].min() - 1
>>> y_max = X_train_std[:, 1].max() + 1
```

```
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=2, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(7, 5))
>>> for idx, clf, tt in zip(product([0, 1], [0, 1]),
...                         all_clf, clf_labels):
...     clf.fit(X_train_std, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
...                                   X_train_std[y_train==0, 1],
...                                   c='blue',
...                                   marker='^',
...                                   s=50)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
...                                   X_train_std[y_train==1, 1],
...                                   c='red',
...                                   marker='o',
...                                   s=50)
...     axarr[idx[0], idx[1]].set_title(tt)
>>> plt.text(-3.5, -4.5,
...          s='Sepal width [standardized]',
...          ha='center', va='center', fontsize=12)
>>> plt.text(-10.5, 4.5,
...          s='Petal length [standardized]',
...          ha='center', va='center',
...          fontsize=12, rotation=90)
>>> plt.show()
```

Interestingly but also as expected, the decision regions of the ensemble classifier seem to be a hybrid of the decision regions from the individual classifiers. At first glance, the majority vote decision boundary looks a lot like the decision boundary of the k-nearest neighbor classifier. However, we can see that it is orthogonal to the y axis for $\textit{sepal width} \geq 1$, just like the decision tree stump:



Before you learn how to tune the individual classifier parameters for ensemble classification, let's call the `get_params` method to get a basic idea of how we can access the individual parameters inside a `GridSearch` object:

```
>>> mv_clf.get_params()
{'decisiontreeclassifier': DecisionTreeClassifier(class_weight=None,
criterion='entropy', max_depth=1,
            max_features=None, max_leaf_nodes=None, min_samples_
leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            random_state=0, splitter='best'),
 'decisiontreeclassifier__class_weight': None,
 'decisiontreeclassifier__criterion': 'entropy',
 [...]
 'decisiontreeclassifier__random_state': 0,
 'decisiontreeclassifier__splitter': 'best',
```

```
 'pipeline-1': Pipeline(steps=[('sc', StandardScaler(copy=True, with_
mean=True, with_std=True)), ('clf', LogisticRegression(C=0.001, class_
weight=None, dual=False, fit_intercept=True,
           intercept_scaling=1, max_iter=100, multi_class='ovr',
           penalty='l2', random_state=0, solver='liblinear',
tol=0.0001,
           verbose=0))]),
 'pipeline-1__clf': LogisticRegression(C=0.001, class_weight=None,
dual=False, fit_intercept=True,
           intercept_scaling=1, max_iter=100, multi_class='ovr',
           penalty='l2', random_state=0, solver='liblinear',
tol=0.0001,
           verbose=0),
 'pipeline-1__clf__C': 0.001,
 'pipeline-1__clf__class_weight': None,
 'pipeline-1__clf__dual': False,
 [...]
 'pipeline-1__sc__with_std': True,
 'pipeline-2': Pipeline(steps=[('sc', StandardScaler(copy=True, with_
mean=True, with_std=True)), ('clf', KNeighborsClassifier(algorithm='au
to', leaf_size=30, metric='minkowski',
            metric_params=None, n_neighbors=1, p=2,
weights='uniform'))]),
 'pipeline-2__clf': KNeighborsClassifier(algorithm='auto', leaf_
size=30, metric='minkowski',
            metric_params=None, n_neighbors=1, p=2,
weights='uniform'),
 'pipeline-2__clf__algorithm': 'auto',
 [...]
 'pipeline-2__sc__with_std': True}
```

Based on the values returned by the `get_params` method, we now know how to access the individual classifier's attributes. Let's now tune the inverse regularization parameter `C` of the logistic regression classifier and the decision tree depth via a grid search for demonstration purposes. The code is as follows:

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = {'decisiontreeclassifier__max_depth': [1, 2],
...           'pipeline-1__clf__C': [0.001, 0.1, 100.0]}
>>> grid = GridSearchCV(estimator=mv_clf,
...                     param_grid=params,
...                     cv=10,
...                     scoring='roc_auc')
>>> grid.fit(X_train, y_train)
```

After the grid search has completed, we can print the different hyperparameter value combinations and the average ROC AUC scores computed via 10-fold cross-validation. The code is as follows:

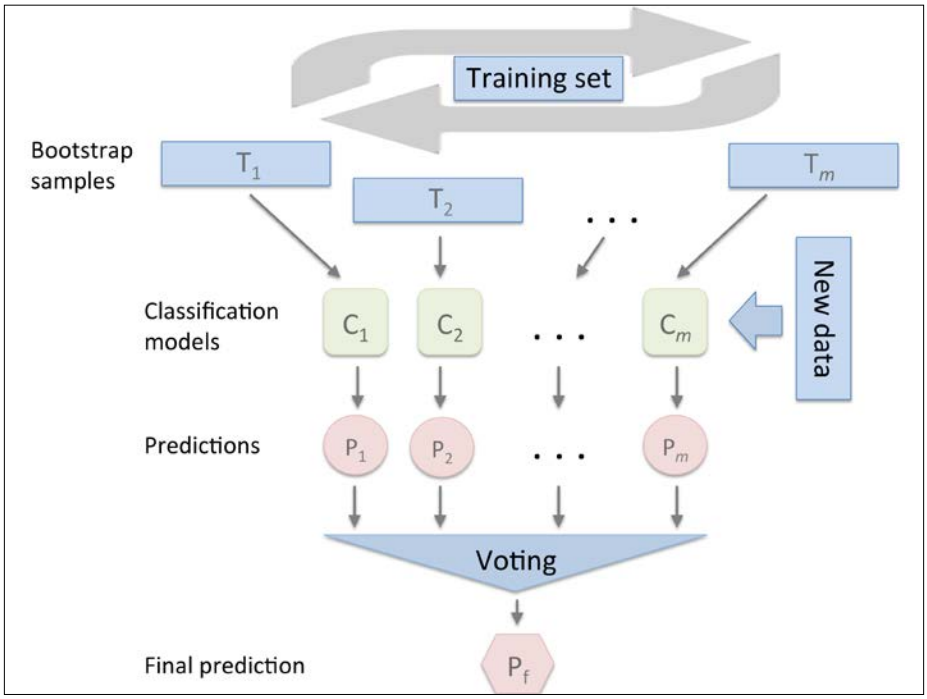
```
>>> for params, mean_score, scores in grid.grid_scores_:
...     print("%0.3f+/-%0.2f %r"
...           % (mean_score, scores.std() / 2, params))
0.967+/-0.05 {'pipeline-1__clf__C': 0.001, 'decisiontreeclassifier__
max_depth': 1}
0.967+/-0.05 {'pipeline-1__clf__C': 0.1, 'decisiontreeclassifier__max_
depth': 1}
1.000+/-0.00 {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__
max_depth': 1}
0.967+/-0.05 {'pipeline-1__clf__C': 0.001, 'decisiontreeclassifier__
max_depth': 2}
0.967+/-0.05 {'pipeline-1__clf__C': 0.1, 'decisiontreeclassifier__max_
depth': 2}
1.000+/-0.00 {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__
max_depth': 2}

>>> print('Best parameters: %s' % grid.best_params_)
Best parameters: {'pipeline-1__clf__C': 100.0,
'decisiontreeclassifier__max_depth': 1}

>>> print('Accuracy: %.2f' % grid.best_score_)
Accuracy: 1.00
```

As we can see, we get the best cross-validation results when we choose a lower regularization strength (`C = 100.0`) whereas the tree depth does not seem to affect the performance at all, suggesting that a decision stump is sufficient to separate the data. To remind ourselves that it is a bad practice to use the test dataset more than once for model evaluation, we are not going to estimate the generalization performance of the tuned hyperparameters in this section. We will move on swiftly to an alternative approach for ensemble learning: **bagging**.

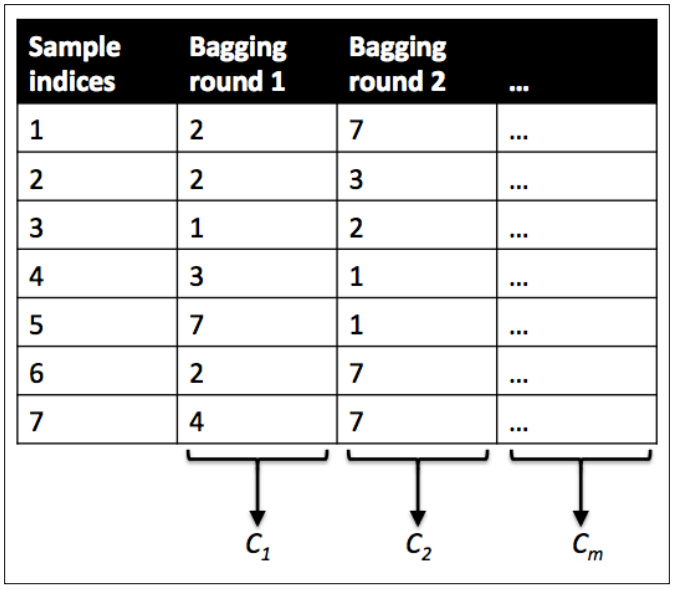
The majority vote approach we implemented in this section is sometimes also referred to as **stacking**. However, the stacking algorithm is more typically used in combination with a logistic regression model that predicts the final class label using the predictions of the individual classifiers in the ensemble as input, which has been described in more detail by David H. Wolpert in D. H. Wolpert. *Stacked generalization*. Neural networks, 5(2):241–259, 1992.

Bagging – building an ensemble of classifiers from bootstrap samples

Bagging is an ensemble learning technique that is closely related to the `MajorityVoteClassifier` that we implemented in the previous section, as illustrated in the following diagram:



However, instead of using the same training set to fit the individual classifiers in the ensemble, we draw bootstrap samples (random samples with replacement) from the initial training set, which is why bagging is also known as **bootstrap aggregating**. To provide a more concrete example of how bootstrapping works, let's consider the example shown in the following figure. Here, we have seven different training instances (denoted as indices 1-7) that are sampled randomly with replacement in each round of bagging. Each bootstrap sample is then used to fit a classifier C_j, which is most typically an unpruned decision tree:



Bagging is also related to the random forest classifier that we introduced in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*. In fact, random forests are a special case of bagging where we also use random feature subsets to fit the individual decision trees. Bagging was first proposed by Leo Breiman in a technical report in 1994; he also showed that bagging can improve the accuracy of unstable models and decrease the degree of overfitting. I highly recommend you read about his research in L. Breiman. *Bagging Predictors*. Machine Learning, 24(2):123–140, 1996, which is freely available online, to learn more about bagging.

To see bagging in action, let's create a more complex classification problem using the **Wine** dataset that we introduced in *Chapter 4*, *Building Good Training Sets – Data Preprocessing*. Here, we will only consider the Wine classes 2 and 3, and we select two features: **Alcohol** and **Hue**.

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/wine/wine.data', header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash',
...                    'Magnesium', 'Total phenols',
...                    'Flavanoids', 'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>> df_wine = df_wine[df_wine['Class label'] != 1]
>>> y = df_wine['Class label'].values
>>> X = df_wine[['Alcohol', 'Hue']].values
```

Next we encode the class labels into binary format and split the dataset into 60 percent training and 40 percent test set, respectively:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> from sklearn.cross_validation import train_test_split
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> X_train, X_test, y_train, y_test =\
...            train_test_split(X, y,
...                             test_size=0.40,
...                             random_state=1)
```

A `BaggingClassifier` algorithm is already implemented in scikit-learn, which we can import from the `ensemble` submodule. Here, we will use an unpruned decision tree as the base classifier and create an ensemble of 500 decision trees fitted on different bootstrap samples of the training dataset:

```
>>> from sklearn.ensemble import BaggingClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=None,
...                               random_state=1)
>>> bag = BaggingClassifier(base_estimator=tree,
```

```
...                          n_estimators=500,
...                          max_samples=1.0,
...                          max_features=1.0,
...                          bootstrap=True,
...                          bootstrap_features=False,
...                          n_jobs=1,
...                          random_state=1)
```

Next we will calculate the accuracy score of the prediction on the training and test dataset to compare the performance of the bagging classifier to the performance of a single unpruned decision tree:

```
>>> from sklearn.metrics import accuracy_score
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...        % (tree_train, tree_test))
Decision tree train/test accuracies 1.000/0.833
```

Based on the accuracy values that we printed by executing the preceding code snippet, the unpruned decision tree predicts all class labels of the training samples correctly; however, the substantially lower test accuracy indicates high variance (overfitting) of the model:

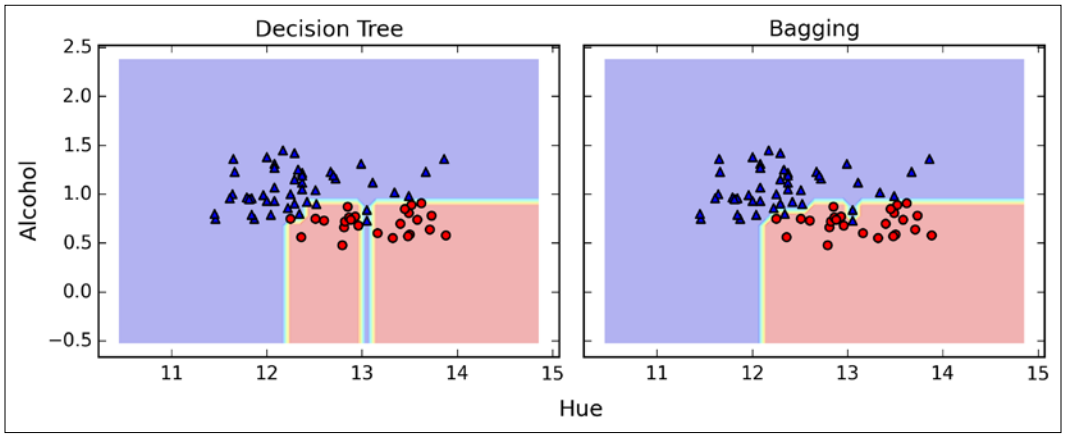
```
>>> bag = bag.fit(X_train, y_train)
>>> y_train_pred = bag.predict(X_train)
>>> y_test_pred = bag.predict(X_test)
>>> bag_train = accuracy_score(y_train, y_train_pred)
>>> bag_test = accuracy_score(y_test, y_test_pred)
>>> print('Bagging train/test accuracies %.3f/%.3f'
...        % (bag_train, bag_test))
Bagging train/test accuracies 1.000/0.896
```

Although the training accuracies of the decision tree and bagging classifier are similar on the training set (both 1.0), we can see that the bagging classifier has a slightly better generalization performance as estimated on the test set. Next let's compare the decision regions between the decision tree and bagging classifier:

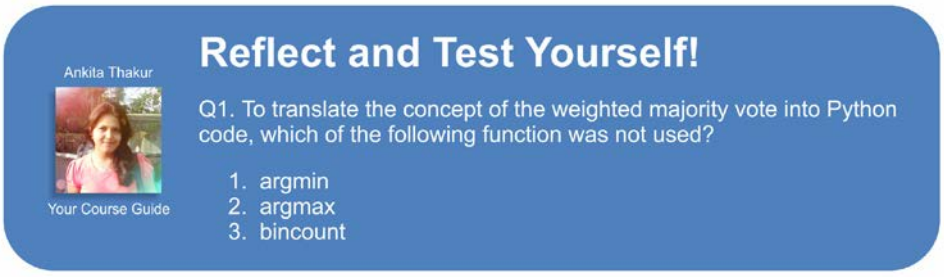
```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
```

```
>>> f, axarr = plt.subplots(nrows=1, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, bag],
...                         ['Decision Tree', 'Bagging']):
...     clf.fit(X_train, y_train)
...
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue', marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='red', marker='o')
...     axarr[idx].set_title(tt)
>>> axarr[0].set_ylabel(Alcohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...          s=Hue',
...          ha='center', va='center', fontsize=12)
>>> plt.show()
```

As we can see in the resulting plot, the piece-wise linear decision boundary of the three-node deep decision tree looks smoother in the bagging ensemble:



We only looked at a very simple bagging example in this section. In practice, more complex classification tasks and datasets' high dimensionality can easily lead to overfitting in single decision trees and this is where the bagging algorithm can really play out its strengths. Finally, we shall note that the bagging algorithm can be an effective approach to reduce the variance of a model. However, bagging is ineffective in reducing model bias, which is why we want to choose an ensemble of classifiers with low bias, for example, unpruned decision trees.



Leveraging weak learners via adaptive boosting

In this section about ensemble methods, we will discuss **boosting** with a special focus on its most common implementation, **AdaBoost** (short for **Adaptive Boosting**).

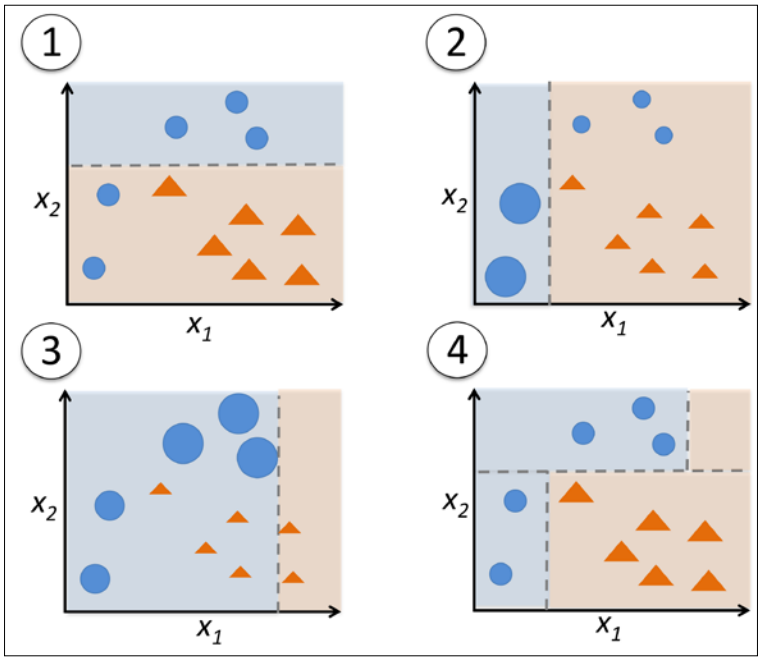
The original idea behind AdaBoost was formulated by Robert Schapire in 1990 (R. E. Schapire. *The Strength of Weak Learnability*. Machine learning, 5(2):197–227, 1990). After Robert Schapire and Yoav Freund presented the AdaBoost algorithm in the Proceedings of the Thirteenth International Conference (ICML 1996), AdaBoost became one of the most widely used ensemble methods in the years that followed (Y. Freund, R. E. Schapire, et al. *Experiments with a New Boosting Algorithm*. In ICML, volume 96, pages 148–156, 1996). In 2003, Freund and Schapire received the *Goedel Prize* for their groundbreaking work, which is a prestigious prize for the most outstanding publications in the computer science field.

In boosting, the ensemble consists of very simple base classifiers, also often referred to as **weak learners**, that have only a slight performance advantage over random guessing. A typical example of a weak learner would be a decision tree stump. The key concept behind boosting is to focus on training samples that are hard to classify, that is, to let the weak learners subsequently learn from misclassified training samples to improve the performance of the ensemble. In contrast to bagging, the initial formulation of boosting, the algorithm uses random subsets of training samples drawn from the training dataset without replacement. The original boosting procedure is summarized in four key steps as follows:

1. Draw a random subset of training samples d_1 without replacement from the training set D to train a weak learner C_1.
2. Draw second random training subset d_2 without replacement from the training set and add 50 percent of the samples that were previously misclassified to train a weak learner C_2.
3. Find the training samples d_3 in the training set D on which C_1 and C_2 disagree to train a third weak learner C_3.
4. Combine the weak learners C_1, C_2, and C_3 via majority voting.

As discussed by Leo Breiman (L. Breiman. *Bias, Variance, and Arcing Classifiers*. 1996), boosting can lead to a decrease in bias as well as variance compared to bagging models. In practice, however, boosting algorithms such as AdaBoost are also known for their high variance, that is, the tendency to overfit the training data (G. Raetsch, T. Onoda, and K. R. Mueller. *An Improvement of Adaboost to Avoid Overfitting*. In Proc. of the Int. Conf. on Neural Information Processing. Citeseer, 1998).

In contrast to the original boosting procedure as described here, AdaBoost uses the complete training set to train the weak learners where the training samples are reweighted in each iteration to build a strong classifier that learns from the mistakes of the previous weak learners in the ensemble. Before we dive deeper into the specific details of the AdaBoost algorithm, let's take a look at the following figure to get a better grasp of the basic concept behind AdaBoost:



To walk through the AdaBoost illustration step by step, we start with subfigure **1**, which represents a training set for binary classification where all training samples are assigned equal weights. Based on this training set, we train a decision stump (shown as a dashed line) that tries to classify the samples of the two classes (triangles and circles) as well as possible by minimizing the cost function (or the impurity score in the special case of decision tree ensembles). For the next round (subfigure **2**), we assign a larger weight to the two previously misclassified samples (circles). Furthermore, we lower the weight of the correctly classified samples. The next decision stump will now be more focused on the training samples that have the largest weights, that is, the training samples that are supposedly hard to classify. The weak learner shown in subfigure **2** misclassifies three different samples from the circle-class, which are then assigned a larger weight as shown in subfigure **3**. Assuming that our AdaBoost ensemble only consists of three rounds of boosting, we would then combine the three weak learners trained on different reweighted training subsets by a weighted majority vote, as shown in subfigure **4**.

Now that have a better understanding behind the basic concept of AdaBoost, let's take a more detailed look at the algorithm using pseudo code. For clarity, we will denote element-wise multiplication by the cross symbol $(\times)$ and the dot product between two vectors by a dot symbol $(\cdot)$, respectively. The steps are as follows:

1. Set weight vector $\boldsymbol{w}$ to uniform weights where $\sum_i w_i = 1$
2. For *j* in *m* boosting rounds, do the following:
3. Train a weighted weak learner: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.
4. Predict class labels: $\hat{y} = \text{predict}(C_j, \boldsymbol{X})$.
5. Compute weighted error rate: $\varepsilon = \boldsymbol{w} \cdot (\hat{\boldsymbol{y}} == \boldsymbol{y})$.
6. Compute coefficient: $\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$.
7. Update weights: $\boldsymbol{w} := \boldsymbol{w} \times \exp(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y})$.
8. Normalize weights to sum to 1: $\boldsymbol{w} := \boldsymbol{w} / \sum_i w_i$.
9. Compute final prediction: $\hat{\boldsymbol{y}} = \left(\sum_{j=1}^{m} (\boldsymbol{\alpha}_j \times \text{predict}(C_j, \boldsymbol{X})) > 0\right)$.

Note that the expression $(\hat{\boldsymbol{y}} == \boldsymbol{y})$ in step 5 refers to a vector of 1s and 0s, where a 1 is assigned if the prediction is incorrect and 0 is assigned otherwise.

Although the AdaBoost algorithm seems to be pretty straightforward, let's walk through a more concrete example using a training set consisting of 10 training samples as illustrated in the following table:

Sample indices	x	y	Weights	$\hat{y}$(x <= 3.0)?	Correct?	Updated weights
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	1	0.1	1	Yes	0.072
4	4.0	-1	0.1	-1	Yes	0.072
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	-1	0.1	-1	Yes	0.072
7	7.0	1	0.1	-1	No	0.167
8	8.0	1	0.1	-1	No	0.167
9	9.0	1	0.1	-1	No	0.167
10	10.0	-1	0.1	-1	Yes	0.072

The first column of the table depicts the sample indices of the training samples 1 to 10. In the second column, we see the feature values of the individual samples assuming this is a one-dimensional dataset. The third column shows the true class label y_i for each training sample x_i, where $y_i \in \{1, -1\}$. The initial weights are shown in the fourth column; we initialize the weights to uniform and normalize them to sum to one. In the case of the 10 sample training set, we therefore assign the 0.1 to each weight w_i in the weight vector $\boldsymbol{w}$. The predicted class labels $\hat{\boldsymbol{y}}$ are shown in the fifth column, assuming that our splitting criterion is $x \leq 3.0$. The last column of the table then shows the updated weights based on the update rules that we defined in the pseudocode.

Since the computation of the weight updates may look a little bit complicated at first, we will now follow the calculation step by step. We start by computing the weighted error rate ε as described in step 5:

$$\varepsilon = 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 1 + 0.1 \times 1 + 0.1 \times 1 + 0.1 \times 0 = \frac{3}{10} = 0.3$$

Next we compute the coefficient α_j (shown in step 6), which is later used in step 7 to update the weights as well as for the weights in majority vote prediction (step 10):

$$\alpha_j = 0.5 \log\left(\frac{1-\varepsilon}{\varepsilon}\right) \approx 0.424$$

After we have computed the coefficient α_j we can now update the weight vector using the following equation:

$$\boldsymbol{w} := \boldsymbol{w} \times \exp\left(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y}\right)$$

Here, $\hat{\boldsymbol{y}} \times \boldsymbol{y}$ is an element-wise multiplication between the vectors of the predicted and true class labels, respectively. Thus, if a prediction $\hat{y}_i$ is correct, $\hat{y}_i \times y_i$ will have a positive sign so that we decrease the *ith* weight since α_j is a positive number as well:

$$0.1 \times \exp\left(-0.424 \times 1 \times 1\right) \approx 0.065$$

Similarly, we will increase the *ith* weight if $\hat{y}_i$ predicted the label incorrectly like this:

$$0.1 \times \exp\left(-0.424 \times 1 \times \left(-1\right)\right) \approx 0.153$$

Or like this:

$$0.1 \times \exp\left(-0.424 \times \left(-1\right) \times \left(1\right)\right) \approx 0.153$$

After we update each weight in the weight vector, we normalize the weights so that they sum up to 1 (step 8):

$$\boldsymbol{w} := \frac{\boldsymbol{w}}{\sum_i w_i}$$

Here, $\sum_i w_i = 7 \times 0.065 + 3 \times 0.153 = 0.914$.

Thus, each weight that corresponds to a correctly classified sample will be reduced from the initial value of 0.1 to $0.065 / 0.914 \approx 0.071$ for the next round of boosting. Similarly, the weights of each incorrectly classified sample will increase from 0.1 to $0.153 / 0.914 \approx 0.167$.

This was AdaBoost in a nutshell. Skipping to the more practical part, let's now train an AdaBoost ensemble classifier via scikit-learn. We will use the same Wine subset that we used in the previous section to train the bagging meta-classifier. Via the `base_estimator` attribute, we will train the `AdaBoostClassifier` on 500 decision tree stumps:

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=None,
...                               random_state=0)
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                          n_estimators=500,
...                          learning_rate=0.1,
...                          random_state=0)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
```

```
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 0.845/0.854
```

As we can see, the decision tree stump tends to underfit the training data in contrast with the unpruned decision tree that we saw in the previous section:

```
>>> ada = ada.fit(X_train, y_train)
>>> y_train_pred = ada.predict(X_train)
>>> y_test_pred = ada.predict(X_test)
>>> ada_train = accuracy_score(y_train, y_train_pred)
>>> ada_test = accuracy_score(y_test, y_test_pred)
>>> print('AdaBoost train/test accuracies %.3f/%.3f'
...       % (ada_train, ada_test))
AdaBoost train/test accuracies 1.000/0.875
```

As we can see, the AdaBoost model predicts all class labels of the training set correctly and also shows a slightly improved test set performance compared to the decision tree stump. However, we also see that we introduced additional variance by our attempt to reduce the model bias.

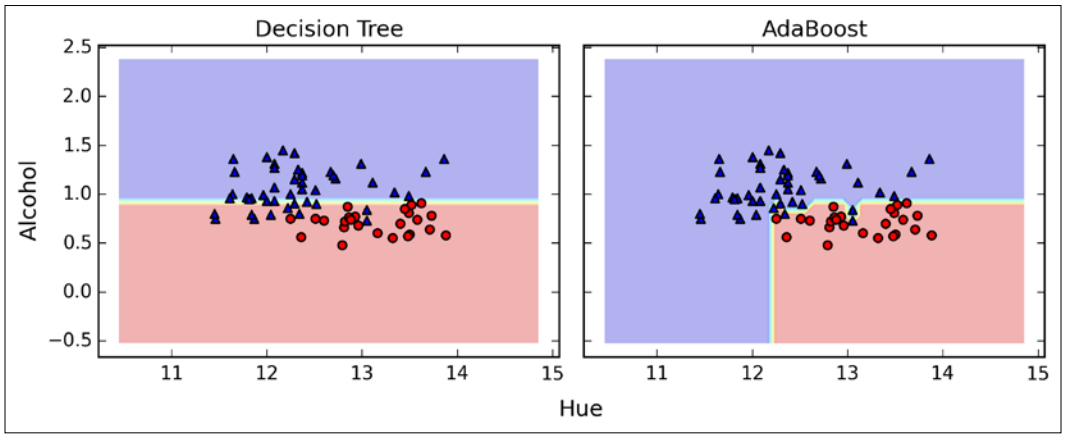
Although we used another simple example for demonstration purposes, we can see that the performance of the AdaBoost classifier is slightly improved compared to the decision stump and achieved very similar accuracy scores to the bagging classifier that we trained in the previous section. However, we should note that it is considered bad practice to select a model based on the repeated usage of the test set. The estimate of the generalization performance may be too optimistic, which we discussed in more detail in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

Finally, let's check what the decision regions look like:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(1, 2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, ada],
```

```
...                           ['Decision Tree', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue',
...                        marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='red',
...                        marker='o')
... axarr[idx].set_title(tt)
... axarr[0].set_ylabel('Alcohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...          s=Hue',
...          ha='center',
...          va='center',
...          fontsize=12)
>>> plt.show()
```

By looking at the decision regions, we can see that the decision boundary of the AdaBoost model is substantially more complex than the decision boundary of the decision stump. In addition, we note that the AdaBoost model separates the feature space very similarly to the bagging classifier that we trained in the previous section.



As concluding remarks about ensemble techniques, it is worth noting that ensemble learning increases the computational complexity compared to individual classifiers. In practice, we need to think carefully whether we want to pay the price of increased computational costs for an often relatively modest improvement of predictive performance.

An often-cited example of this trade-off is the famous *$1 Million Netflix Prize*, which was won using ensemble techniques. The details about the algorithm were published in A. Toescher, M. Jahrer, and R. M. Bell. *The Bigchaos Solution to the Netflix Grand Prize*. Netflix prize documentation, 2009 (which is available at `http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BigChaos.pdf`). Although the winning team received the $1 million prize money, Netflix never implemented their model due to its complexity, which made it unfeasible for a real-world application. To quote their exact words (`http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html`):

> *"[…] additional accuracy gains that we measured did not seem to justify the engineering effort needed to bring them into a production environment."*

Summary of Module 4 Chapter 7

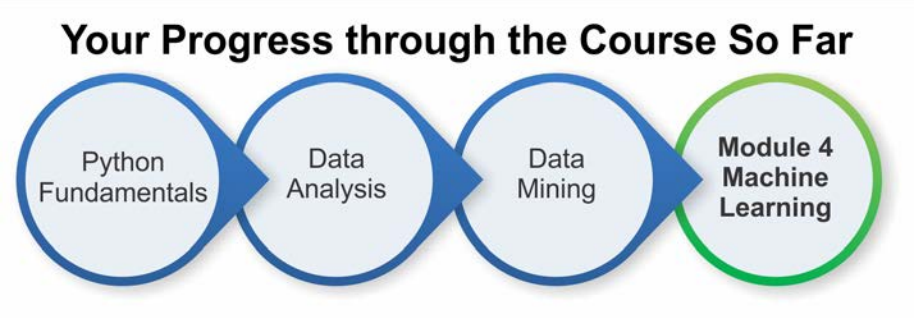
Ankita Thakur

Your Course Guide

In this chapter, we looked at some of the most popular and widely used techniques for ensemble learning. Ensemble methods combine different classification models to cancel out their individual weakness, which often results in stable and well-performing models that are very attractive for industrial applications as well as machine learning competitions.

In the beginning of this chapter, we implemented a MajorityVoteClassifier in Python that allows us to combine different algorithm for classification. We then looked at bagging, a useful technique to reduce the variance of a model by drawing random bootstrap samples from the training set and combining the individually trained classifiers via majority vote. Then we discussed AdaBoost, which is an algorithm that is based on weak learners that subsequently learn from mistakes.

Throughout the previous chapters, we discussed different learning algorithms, tuning, and evaluation techniques. In the next chapter, we will take a look at a subcategory of supervised learning, regression analysis, which lets us predict outcome variables on a continuous scale, in contrast to the categorical class labels of the classification models that we have been working with so far.



8

Predicting Continuous Target Variables with Regression Analysis

Throughout the previous chapters, you learned a lot about the main concepts behind *supervised learning* and trained many different models for classification tasks to predict group memberships or categorical variables. In this chapter, we will take a dive into another subcategory of supervised learning: *regression analysis*.

Regression models are used to predict target variables on a *continuous* scale, which makes them attractive for addressing many questions in science as well as applications in industry, such as understanding relationships between variables, evaluating trends, or making forecasts. One example would be predicting the sales of a company in future months.

In this chapter, we will discuss the main concepts of regression models and cover the following topics:

- Exploring and visualizing datasets
- Looking at different approaches to implement linear regression models
- Training regression models that are robust to outliers
- Evaluating regression models and diagnosing common problems
- Fitting regression models to nonlinear data

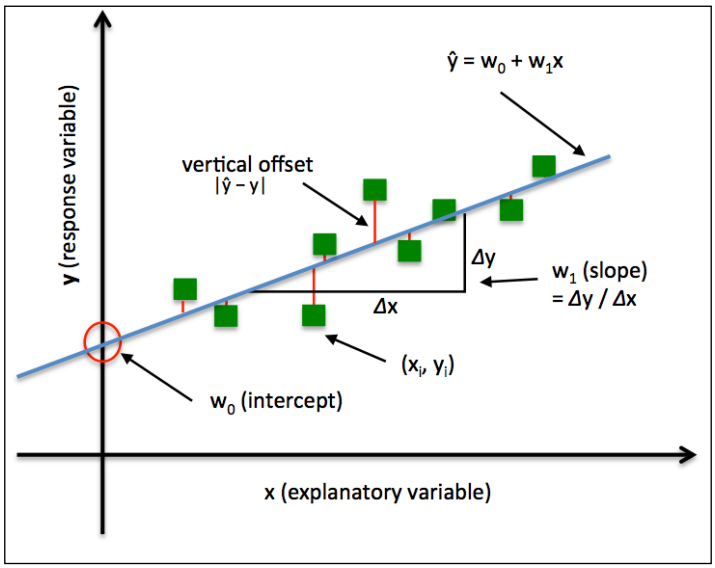
Introducing a simple linear regression model

The goal of simple (*univariate*) linear regression is to model the relationship between a single feature (explanatory variable *x*) and a continuous valued *response* (target variable *y*). The equation of a linear model with one explanatory variable is defined as follows:

$$y = w_0 + w_1 x$$

Here, the weight w_0 represents the *y* axis intercepts and w_1 is the coefficient of the explanatory variable. Our goal is to learn the weights of the linear equation to describe the relationship between the explanatory variable and the target variable, which can then be used to predict the responses of new explanatory variables that were not part of the training dataset.

Based on the linear equation that we defined previously, linear regression can be understood as finding the best-fitting straight line through the sample points, as shown in the following figure:



This best-fitting line is also called the **regression line**, and the vertical lines from the regression line to the sample points are the so-called **offsets** or **residuals**—the errors of our prediction.

The special case of one explanatory variable is also called **simple linear regression**, but of course we can also generalize the linear regression model to multiple explanatory variables. Hence, this process is called **multiple linear regression**:

$$y = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \sum_{i=0}^{m} w_i x_i = w^T x$$

Here, w_0 is the *y* axis intercept with $x_0 = 1$.

Exploring the Housing Dataset

Before we implement our first linear regression model, we will introduce a new dataset, the **Housing Dataset**, which contains information about houses in the suburbs of Boston collected by D. Harrison and D.L. Rubinfeld in 1978. The *Housing Dataset* has been made freely available and can be downloaded from the *UCI machine learning repository* at `https://archive.ics.uci.edu/ml/datasets/Housing`.

The features of the 506 samples may be summarized as shown in the excerpt of the dataset description:

- **CRIM**: This is the per capita crime rate by town
- **ZN**: This is the proportion of residential land zoned for lots larger than 25,000 sq.ft.
- **INDUS**: This is the proportion of non-retail business acres per town
- **CHAS**: This is the Charles River dummy variable (this is equal to 1 if tract bounds river; 0 otherwise)
- **NOX**: This is the nitric oxides concentration (parts per 10 million)
- **RM**: This is the average number of rooms per dwelling
- **AGE**: This is the proportion of owner-occupied units built prior to 1940
- **DIS**: This is the weighted distances to five Boston employment centers
- **RAD**: This is the index of accessibility to radial highways
- **TAX**: This is the full-value property-tax rate per $10,000
- **PTRATIO**: This is the pupil-teacher ratio by town
- **B**: This is calculated as *1000(Bk - 0.63)^2*, where Bk is the proportion of people of African American descent by town
- **LSTAT**: This is the percentage lower status of the population
- **MEDV**: This is the median value of owner-occupied homes in $1000s

For the rest of this chapter, we will regard the housing prices (MEDV) as our target variable—the variable that we want to predict using one or more of the 13 explanatory variables. Before we explore this dataset further, let's fetch it from the UCI repository into a pandas `DataFrame`:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/housing/housing.data',
...                  header=None, sep='\s+')
>>> df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS',
...               'NOX', 'RM', 'AGE', 'DIS', 'RAD',
...               'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
>>> df.head()
```

To confirm that the dataset was loaded successfully, we displayed the first five lines of the dataset, as shown in the following screenshot:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

Visualizing the important characteristics of a dataset

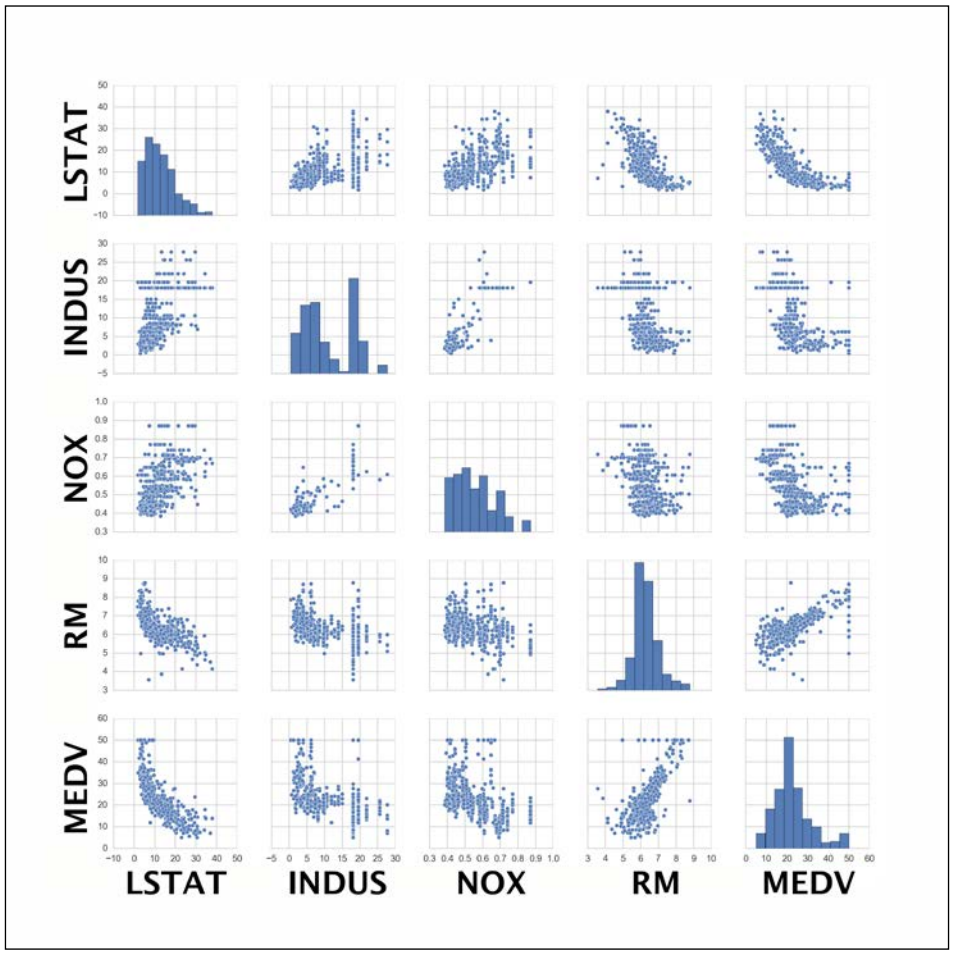
Exploratory Data Analysis (**EDA**) is an important and recommended first step prior to the training of a machine learning model. In the rest of this section, we will use some simple yet useful techniques from the graphical EDA toolbox that may help us to visually detect the presence of outliers, the distribution of the data, and the relationships between features.

First, we will create a *scatterplot matrix* that allows us to visualize the pair-wise correlations between the different features in this dataset in one place. To plot the scatterplot matrix, we will use the `pairplot` function from the `seaborn` library (`http://stanford.edu/~mwaskom/software/seaborn/`), which is a Python library for drawing statistical plots based on matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> sns.set(style='whitegrid', context='notebook')
```

```
>>> cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']
>>> sns.pairplot(df[cols], size=2.5)
>>> plt.show()
```

As we can see in the following figure, the scatterplot matrix provides us with a useful graphical summary of the relationships in a dataset:



Importing the seaborn library modifies the default aesthetics of matplotlib for the current Python session. If you do not want to use seaborn's style settings, you can reset the matplotlib settings by executing the following command:

```
>>> sns.reset_orig()
```

Due to space constraints and for purposes of readability, we only plotted five columns from the dataset: **LSTAT**, **INDUS**, **NOX**, **RM**, and **MEDV**. However, you are encouraged to create a scatterplot matrix of the whole `DataFrame` to further explore the data.

Using this scatterplot matrix, we can now quickly eyeball how the data is distributed and whether it contains outliers. For example, we can see that there is a linear relationship between **RM** and the housing prices **MEDV** (the fifth column of the fourth row). Furthermore, we can see in the histogram (the lower right subplot in the scatter plot matrix) that the **MEDV** variable seems to be normally distributed but contains several outliers.

Note that in contrast to common belief, training a linear regression model does not require that the explanatory or target variables are normally distributed. The normality assumption is only a requirement for certain statistical tests and hypothesis tests that are beyond the scope of this book (Montgomery, D. C., Peck, E. A., and Vining, G. G. *Introduction to linear regression analysis*. John Wiley and Sons, 2012, pp.318–319).

To quantify the linear relationship between the features, we will now create a correlation matrix. A correlation matrix is closely related to the covariance matrix that we have seen in the section about **principal component analysis** (**PCA**) in *Chapter 4*, *Building Good Training Sets – Data Preprocessing*. Intuitively, we can interpret the correlation matrix as a rescaled version of the covariance matrix. In fact, the correlation matrix is identical to a covariance matrix computed from standardized data.

The correlation matrix is a square matrix that contains the **Pearson product-moment correlation coefficients** (often abbreviated as **Pearson's r**), which measure the linear dependence between pairs of features. The correlation coefficients are bounded to the range -1 and 1. Two features have a perfect positive correlation if $r = 1$, no correlation if $r = 0$, and a perfect negative correlation if $r = -1$, respectively. As mentioned previously, Pearson's correlation coefficient can simply be calculated as the covariance between two features x and y (numerator) divided by the product of their standard deviations (denominator):

$$r = \frac{\sum_{i=1}^{n}\left[\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)\right]}{\sqrt{\sum_{i=1}^{n}\left(x^{(i)} - \mu_x\right)^2}\sqrt{\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Here, μ denotes the sample mean of the corresponding feature, σ_{xy} is the covariance between the features x and y, and σ_x and σ_y are the features' standard deviations, respectively.

We can show that the covariance between standardized features is in fact equal to their linear correlation coefficient.

Let's first standardize the features x and y, to obtain their z-scores which we will denote as x' and y', respectively:

$$x' = \frac{x - \mu_x}{\sigma_x}, y' = \frac{y - \mu_y}{\sigma_y}$$

Remember that we calculate the (population) covariance between two features as follows:

$$\sigma_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

Since standardization centers a feature variable at mean 0, we can now calculate the covariance between the scaled features as follows:

$$\sigma'_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x' - 0\right)\left(y' - 0\right)$$

Through resubstitution, we get the following result:

$$\frac{1}{n}\sum_{i}^{n}\left(\frac{x - \mu_x}{\sigma_x}\right)\left(\frac{y - \mu_y}{\sigma_y}\right)$$

$$\frac{1}{n \cdot \sigma_x \sigma_y}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

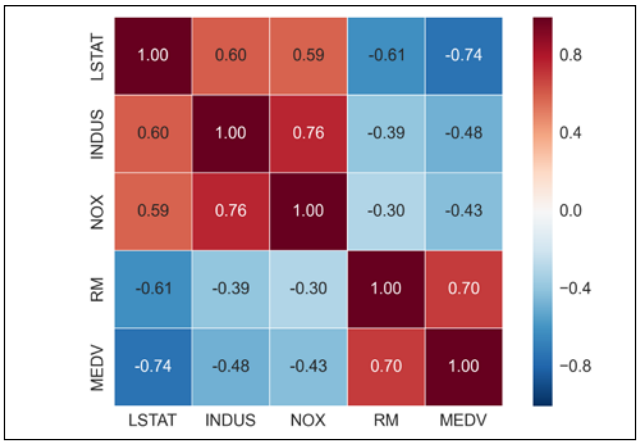
We can simplify it as follows:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

In the following code example, we will use NumPy's `corrcoef` function on the five feature columns that we previously visualized in the scatterplot matrix, and we will use seaborn's `heatmap` function to plot the correlation matrix array as a heat map:

```
>>> import numpy as np
>>> cm = np.corrcoef(df[cols].values.T)
>>> sns.set(font_scale=1.5)
>>> hm = sns.heatmap(cm,
...                  cbar=True,
...                  annot=True,
...                  square=True,
...                  fmt='.2f',
...                  annot_kws={'size': 15},
...                  yticklabels=cols,
...                  xticklabels=cols)
>>> plt.show()
```

As we can see in the resulting figure, the correlation matrix provides us with another useful summary graphic that can help us to select features based on their respective linear correlations:



To fit a linear regression model, we are interested in those features that have a high correlation with our target variable **MEDV**. Looking at the preceding correlation matrix, we see that our target variable **MEDV** shows the largest correlation with the **LSTAT** variable (-0.74). However, as you might remember from the scatterplot matrix, there is a clear nonlinear relationship between **LSTAT** and **MEDV**. On the other hand, the correlation between **RM** and **MEDV** is also relatively high (0.70) and given the linear relationship between those two variables that we observed in the scatterplot, **RM** seems to be a good choice for an explanatory variable to introduce the concepts of a simple linear regression model in the following section.

Implementing an ordinary least squares linear regression model

At the beginning of this chapter, we discussed that linear regression can be understood as finding the best-fitting straight line through the sample points of our training data. However, we have neither defined the term *best-fitting* nor have we discussed the different techniques of fitting such a model. In the following subsections, we will fill in the missing pieces of this puzzle using the **Ordinary Least Squares** (**OLS**) method to estimate the parameters of the regression line that minimizes the sum of the squared vertical distances (residuals or errors) to the sample points.

Solving regression for regression parameters with gradient descent

Consider our implementation of the **ADAptive LInear NEuron** (**Adaline**) from *Chapter 2*, *Training Machine Learning Algorithms for Classification*; we remember that the artificial neuron uses a linear activation function and we defined a cost function $J(\cdot)$, which we minimized to learn the weights via optimization algorithms, such as **Gradient Descent** (**GD**) and **Stochastic Gradient Descent** (**SGD**). This cost function in Adaline is the **Sum of Squared Errors** (**SSE**). This is identical to the OLS cost function that we defined:

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Here, $\hat{y}$ is the predicted value $\hat{y} = w^T x$ (note that the term 1/2 is just used for convenience to derive the update rule of GD). Essentially, OLS linear regression can be understood as Adaline without the unit step function so that we obtain continuous target values instead of the class labels -1 and 1. To demonstrate the similarity, let's take the GD implementation of *Adaline* from *Chapter 2*, *Training Machine Learning Algorithms for Classification*, and remove the unit step function to implement our first linear regression model:

```
class LinearRegressionGD(object):

    def __init__(self, eta=0.001, n_iter=20):
        self.eta = eta
```

```
            self.n_iter = n_iter

        def fit(self, X, y):
            self.w_ = np.zeros(1 + X.shape[1])
            self.cost_ = []

            for i in range(self.n_iter):
                output = self.net_input(X)
                errors = (y - output)
                self.w_[1:] += self.eta * X.T.dot(errors)
                self.w_[0] += self.eta * errors.sum()
                cost = (errors**2).sum() / 2.0
                self.cost_.append(cost)
            return self

        def net_input(self, X):
            return np.dot(X, self.w_[1:]) + self.w_[0]

        def predict(self, X):
            return self.net_input(X)
```

If you need a refresher about how the weights are being updated—taking a step in the opposite direction of the gradient—please revisit the Adaline section in *Chapter 2*, *Training Machine Learning Algorithms for Classification*.

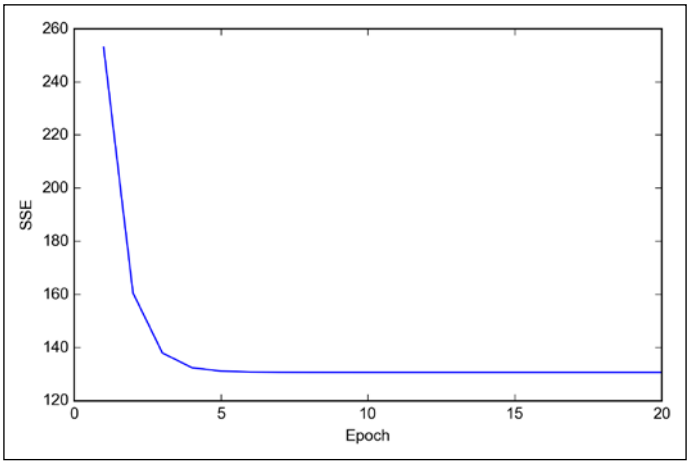
To see our `LinearRegressionGD` regressor in action, let's use the RM (number of rooms) variable from the Housing Data Set as the explanatory variable to train a model that can predict MEDV (the housing prices). Furthermore, we will standardize the variables for better convergence of the GD algorithm. The code is as follows:

```
>>> X = df[['RM']].values
>>> y = df['MEDV'].values
>>> from sklearn.preprocessing import StandardScaler
>>> sc_x = StandardScaler()
>>> sc_y = StandardScaler()
>>> X_std = sc_x.fit_transform(X)
>>> y_std = sc_y.fit_transform(y)
>>> lr = LinearRegressionGD()
>>> lr.fit(X_std, y_std)
```

We discussed in *Chapter 2, Training Machine Learning Algorithms for Classification*, that it is always a good idea to plot the cost as a function of the number of epochs (passes over the training dataset) when we are using optimization algorithms, such as gradient descent, to check for convergence. To cut a long story short, let's plot the cost against the number of epochs to check if the linear regression has converged:

```
>>> plt.plot(range(1, lr.n_iter+1), lr.cost_)
>>> plt.ylabel('SSE')
>>> plt.xlabel('Epoch')
>>> plt.show()
```

As we can see in the following plot, the GD algorithm converged after the fifth epoch:



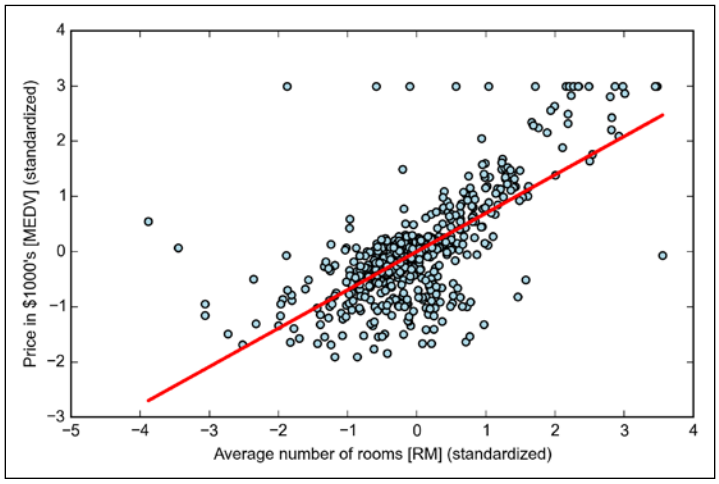
Next, let's visualize how well the linear regression line fits the training data. To do so, we will define a simple helper function that will plot a scatterplot of the training samples and add the regression line:

```
>>> def lin_regplot(X, y, model):
...     plt.scatter(X, y, c='blue')
...     plt.plot(X, model.predict(X), color='red')
...     return None
```

Now, we will use this `lin_regplot` function to plot the number of rooms against house prices:

```
>>> lin_regplot(X_std, y_std, lr)
>>> plt.xlabel('Average number of rooms [RM] (standardized)')
>>> plt.ylabel('Price in $1000\'s [MEDV] (standardized)')
>>> plt.show()
```

As we can see in the following plot, the linear regression line reflects the general trend that house prices tend to increase with the number of rooms:



Although this observation makes intuitive sense, the data also tells us that the number of rooms does not explain the house prices very well in many cases. Later in this chapter, we will discuss how to quantify the performance of a regression model. Interestingly, we also observe a curious line $y = 3$, which suggests that the prices may have been clipped. In certain applications, it may also be important to report the predicted outcome variables on their original scale. To scale the predicted price outcome back on the **Price in $1000's** axes, we can simply apply the `inverse_transform` method of the `StandardScaler`:

```
>>> num_rooms_std = sc_x.transform([5.0])
>>> price_std = lr.predict(num_rooms_std)
>>> print("Price in $1000's: %.3f" % \
...       sc_y.inverse_transform(price_std))
Price in $1000's: 10.840
```

In the preceding code example, we used the previously trained linear regression model to predict the price of a house with five rooms. According to our model, such a house is worth $10,840.

On a side note, it is also worth mentioning that we technically don't have to update the weights of the intercept if we are working with standardized variables since the *y* axis intercept is always 0 in those cases. We can quickly confirm this by printing the weights:

```
>>> print('Slope: %.3f' % lr.w_[1])
Slope: 0.695
>>> print('Intercept: %.3f' % lr.w_[0])
Intercept: -0.000
```

Estimating the coefficient of a regression model via scikit-learn

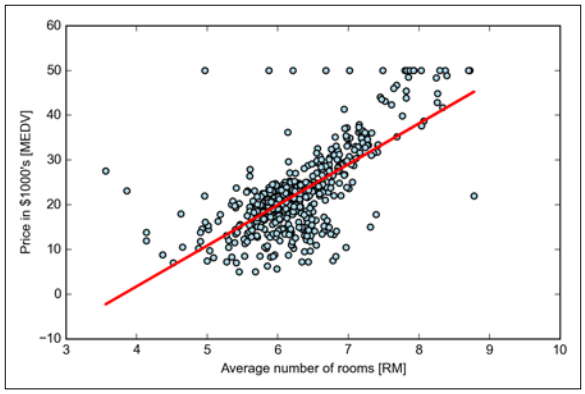
In the previous section, we implemented a working model for regression analysis. However, in a real-world application, we may be interested in more efficient implementations, for example, scikit-learn's `LinearRegression` object that makes use of the **LIBLINEAR** library and advanced optimization algorithms that work better with unstandardized variables. This is sometimes desirable for certain applications:

```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
>>> slr.fit(X, y)
>>> print('Slope: %.3f' % slr.coef_[0])
Slope: 9.102
>>> print('Intercept: %.3f' % slr.intercept_)
Intercept: -34.671
```

As we can see by executing the preceding code, scikit-learn's `LinearRegression` model fitted with the unstandardized **RM** and **MEDV** variables yielded different model coefficients. Let's compare it to our own GD implementation by plotting MEDV against RM:

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Average number of rooms [RM]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.show()w
```

Now, when we plot the training data and our fitted model by executing the code above, we can see that the overall result looks identical to our GD implementation:



As an alternative to using machine learning libraries, there is also a closed-form solution for solving OLS involving a system of linear equations that can be found in most introductory statistics textbooks:

$$w = \left(X^T X\right)^{-1} X^T y$$

We can implement it in Python as follows:

```
# adding a column vector of "ones"
>>> Xb = np.hstack((np.ones((X.shape[0], 1)), X))
>>> w = np.zeros(X.shape[1])
>>> z = np.linalg.inv(np.dot(Xb.T, Xb))
>>> w = np.dot(z, np.dot(Xb.T, y))
>>> print('Slope: %.3f' % w[1])
Slope: 9.102
>>> print('Intercept: %.3f' % w[0])
Intercept: -34.671
```

The advantage of this method is that it is guaranteed to find the optimal solution analytically. However, if we are working with very large datasets, it can be computationally too expensive to invert the matrix in this formula (sometimes also called the **normal equation**) or the sample matrix may be singular (non-invertible), which is why we may prefer iterative methods in certain cases.

If you are interested in more information on how to obtain the normal equations, I recommend you take a look at Dr. Stephen Pollock's chapter, *The Classical Linear Regression Model* from his lectures at the University of Leicester, which are available for free at `http://www.le.ac.uk/users/dsgp1/COURSES/MESOMET/ECMETXT/06mesmet.pdf`.

Fitting a robust regression model using RANSAC

Linear regression models can be heavily impacted by the presence of outliers. In certain situations, a very small subset of our data can have a big effect on the estimated model coefficients. There are many statistical tests that can be used to detect outliers, which are beyond the scope of the book. However, removing outliers always requires our own judgment as a data scientist, as well as our domain knowledge.

As an alternative to throwing out outliers, we will look at a robust method of regression using the **RANdom SAmple Consensus** (**RANSAC**) algorithm, which fits a regression model to a subset of the data, the so-called *inliers*.

We can summarize the iterative RANSAC algorithm as follows:

1. Select a random number of samples to be inliers and fit the model.
2. Test all other data points against the fitted model and add those points that fall within a user-given tolerance to the inliers.
3. Refit the model using all inliers.
4. Estimate the error of the fitted model versus the inliers.
5. Terminate the algorithm if the performance meets a certain user-defined threshold or if a fixed number of iterations has been reached; go back to step 1 otherwise.

Let's now wrap our linear model in the RANSAC algorithm using scikit-learn's `RANSACRegressor` object:

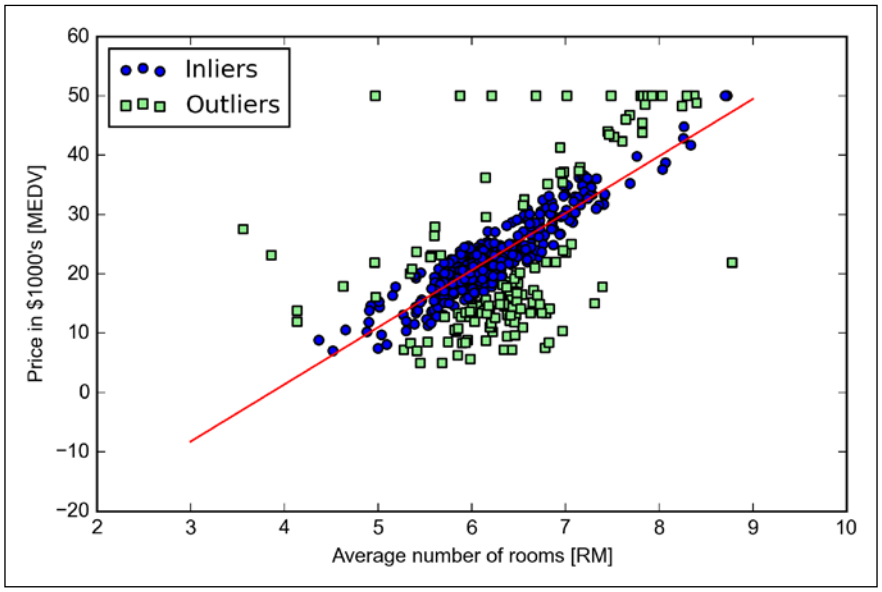
```
>>> from sklearn.linear_model import RANSACRegressor
>>> ransac = RANSACRegressor(LinearRegression(),
...            max_trials=100,
...            min_samples=50,
...            residual_metric=lambda x: np.sum(np.abs(x), axis=1),
...            residual_threshold=5.0,
...            random_state=0)
>>> ransac.fit(X, y)
```

We set the maximum number of iterations of the `RANSACRegressor` to 100, and using `min_samples=50`, we set the minimum number of the randomly chosen samples to be at least 50. Using the `residual_metric` parameter, we provided a callable `lambda` function that simply calculates the absolute vertical distances between the fitted line and the sample points. By setting the `residual_threshold` parameter to `5.0`, we only allowed samples to be included in the inlier set if their vertical distance to the fitted line is within 5 distance units, which works well on this particular dataset. By default, scikit-learn uses the MAD estimate to select the inlier threshold, where **MAD** stands for the **Median Absolute Deviation** of the target values `y`. However, the choice of an appropriate value for the inlier threshold is problem-specific, which is one disadvantage of RANSAC. Many different approaches have been developed over the recent years to select a good inlier threshold automatically. You can find a detailed discussion in R. Toldo and A. Fusiello's. *Automatic Estimation of the Inlier Threshold in Robust Multiple Structures Fitting* (in Image Analysis and Processing–ICIAP 2009, pages 123–131. Springer, 2009).

After we have fitted the RANSAC model, let's obtain the inliers and outliers from the fitted RANSAC linear regression model and plot them together with the linear fit:

```
>>> inlier_mask = ransac.inlier_mask_
>>> outlier_mask = np.logical_not(inlier_mask)
>>> line_X = np.arange(3, 10, 1)
>>> line_y_ransac = ransac.predict(line_X[:, np.newaxis])
>>> plt.scatter(X[inlier_mask], y[inlier_mask],
...             c='blue', marker='o', label='Inliers')
>>> plt.scatter(X[outlier_mask], y[outlier_mask],
...             c='lightgreen', marker='s', label='Outliers')
>>> plt.plot(line_X, line_y_ransac, color='red')
>>> plt.xlabel('Average number of rooms [RM]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

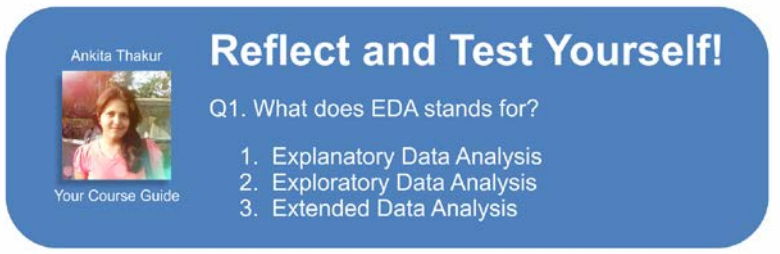
As we can see in the following scatterplot, the linear regression model was fitted on the detected set of inliers shown as circles:



When we print the slope and intercept of the model executing the following code, we can see that the linear regression line is slightly different from the fit that we obtained in the previous section without RANSAC:

```
>>> print('Slope: %.3f' % ransac.estimator_.coef_[0])
Slope: 9.621
>>> print('Intercept: %.3f' % ransac.estimator_.intercept_)
Intercept: -37.137
```

Using RANSAC, we reduced the potential effect of the outliers in this dataset, but we don't know if this approach has a positive effect on the predictive performance for unseen data. Thus, in the next section we will discuss how to evaluate a regression model for different approaches, which is a crucial part of building systems for predictive modeling.



Evaluating the performance of linear regression models

In the previous section, we discussed how to fit a regression model on training data. However, you learned in previous chapters that it is crucial to test the model on data that it hasn't seen during training to obtain an unbiased estimate of its performance.

As we remember from *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we want to split our dataset into separate training and test datasets where we use the former to fit the model and the latter to evaluate its performance to generalize to unseen data. Instead of proceeding with the simple regression model, we will now use all variables in the dataset and train a multiple regression model:

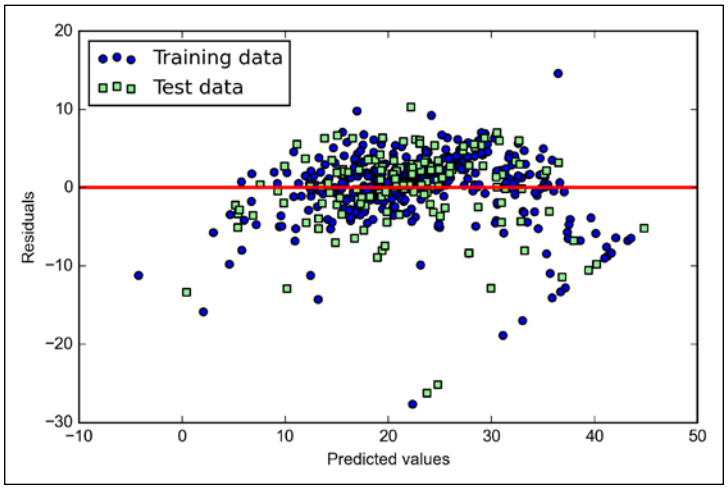
```
>>> from sklearn.cross_validation import train_test_split
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test = train_test_split(
...       X, y, test_size=0.3, random_state=0)
>>> slr = LinearRegression()
>>> slr.fit(X_train, y_train)
>>> y_train_pred = slr.predict(X_train)
>>> y_test_pred = slr.predict(X_test)
```

Since our model uses multiple explanatory variables, we can't visualize the linear regression line (or hyperplane to be precise) in a two-dimensional plot, but we can plot the residuals (the differences or vertical distances between the actual and predicted values) versus the predicted values to diagnose our regression model. Those **residual plots** are a commonly used graphical analysis for diagnosing regression models to detect nonlinearity and outliers, and to check if the errors are randomly distributed.

Using the following code, we will now plot a residual plot where we simply subtract the true target variables from our predicted responses:

```
>>> plt.scatter(y_train_pred, y_train_pred - y_train,
...             c='blue', marker='o', label='Training data')
>>> plt.scatter(y_test_pred,  y_test_pred - y_test,
...             c='lightgreen', marker='s', label='Test data')
>>> plt.xlabel('Predicted values')
>>> plt.ylabel('Residuals')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
>>> plt.xlim([-10, 50])
>>> plt.show()
```

After executing the code, we should see a residual plot with a line passing through the *x* axis origin as shown here:



In the case of a perfect prediction, the residuals would be exactly zero, which we will probably never encounter in realistic and practical applications. However, for a good regression model, we would expect that the errors are randomly distributed and the residuals should be randomly scattered around the centerline. If we see patterns in a residual plot, it means that our model is unable to capture some explanatory information, which is leaked into the residuals as we can slightly see in our preceding residual plot. Furthermore, we can also use residual plots to detect outliers, which are represented by the points with a large deviation from the centerline.

Another useful quantitative measure of a model's performance is the so-called **Mean Squared Error** (**MSE**), which is simply the average value of the SSE cost function that we minimize to fit the linear regression model. The MSE is useful to for comparing different regression models or for tuning their parameters via a grid search and cross-validation:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Execute the following code:

```
>>> from sklearn.metrics import mean_squared_error
>>> print('MSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train, y_train_pred),
        mean_squared_error(y_test, y_test_pred)))
```

We will see that the MSE on the training set is 19.96, and the MSE of the test set is much larger with a value of 27.20, which is an indicator that our model is overfitting the training data.

Sometimes it may be more useful to report the coefficient of determination (R^2), which can be understood as a standardized version of the MSE, for better interpretability of the model performance. In other words, R^2 is the fraction of response variance that is captured by the model. The R^2 value is defined as follows:

$$R^2 = 1 - \frac{SSE}{SST}$$

Here, SSE is the sum of squared errors and SST is the total sum of squares $SST = \sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2$, or in other words, it is simply the variance of the response.

Let's quickly show that R^2 is indeed just a rescaled version of the MSE:

$$R^2 = 1 - \frac{SSE}{SST}$$

$$1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2}$$

$$1 - \frac{MSE}{Var(y)}$$

For the training dataset, R^2 is bounded between 0 and 1, but it can become negative for the test set. If $R^2 = 1$, the model fits the data perfectly with a corresponding $MSE = 0$.

Evaluated on the training data, the R^2 of our model is 0.765, which doesn't sound too bad. However, the R^2 on the test dataset is only 0.673, which we can compute by executing the following code:

```
>>> from sklearn.metrics import r2_score
>>> print('R^2 train: %.3f, test: %.3f' %
...       (r2_score(y_train, y_train_pred),
...        r2_score(y_test, y_test_pred)))
```

Using regularized methods for regression

As we discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, regularization is one approach to tackle the problem of overfitting by adding additional information, and thereby shrinking the parameter values of the model to induce a penalty against complexity. The most popular approaches to regularized linear regression are the so-called **Ridge Regression**, **Least Absolute Shrinkage and Selection Operator** (**LASSO**) and **Elastic Net** method.

Ridge regression is an L2 penalized model where we simply add the squared sum of the weights to our least-squares cost function:

$$J(w)_{Ridge} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda \| w \|_2^2$$

Here:

$$L2: \quad \lambda \| w \|_2^2 = \lambda \sum_{j=1}^{m} w_j^2$$

By increasing the value of the hyperparameter λ, we increase the regularization strength and shrink the weights of our model. Please note that we don't regularize the intercept term w_0.

An alternative approach that can lead to sparse models is the LASSO. Depending on the regularization strength, certain weights can become zero, which makes the LASSO also useful as a supervised feature selection technique:

$$J(w)_{LASSO} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \parallel w \parallel_1$$

Here:

$$L1: \lambda \parallel w \parallel_1 = \lambda \sum_{j=1}^{m} \left| w_j \right|$$

However, a limitation of the LASSO is that it selects at most n variables if $m > n$. A compromise between Ridge regression and the LASSO is the Elastic Net, which has a L1 penalty to generate sparsity and a L2 penalty to overcome some of the limitations of the LASSO, such as the number of selected variables.

$$J(w)_{ElasticNet} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda_1 \sum_{j=1}^{m} w_j^{\,2} + \lambda_2 \sum_{j=1}^{m} \left| w_j \right|$$

Those regularized regression models are all available via scikit-learn, and the usage is similar to the regular regression model except that we have to specify the regularization strength via the parameter λ, for example, optimized via k-fold cross-validation.

A Ridge Regression model can be initialized as follows:

```
>>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

Note that the regularization strength is regulated by the parameter `alpha`, which is similar to the parameter λ. Likewise, we can initialize a LASSO regressor from the `linear_model` submodule:

```
>>> from sklearn.linear_model import Lasso
>>> lasso = Lasso(alpha=1.0)
```

Lastly, the `ElasticNet` implementation allows us to vary the L1 to L2 ratio:

```
>>> from sklearn.linear_model import ElasticNet
>>> lasso = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

For example, if we set `l1_ratio` to `1.0`, the `ElasticNet` regressor would be equal to LASSO regression. For more detailed information about the different implementations of linear regression, please see the documentation at `http://scikit-learn.org/stable/modules/linear_model.html`.

Turning a linear regression model into a curve – polynomial regression

In the previous sections, we assumed a linear relationship between explanatory and response variables. One way to account for the violation of linearity assumption is to use a polynomial regression model by adding polynomial terms:

$$y = w_0 + w_1 x + w_2 x^2 x^2 + \ldots + w_d x^d$$

Here, d denotes the degree of the polynomial. Although we can use polynomial regression to model a nonlinear relationship, it is still considered a multiple linear regression model because of the linear regression coefficients w.

We will now discuss how to use the `PolynomialFeatures` transformer class from scikit-learn to add a quadratic term ($d = 2$) to a simple regression problem with one explanatory variable, and compare the polynomial to the linear fit. The steps are as follows:

1. Add a second degree polynomial term:

```
from sklearn.preprocessing import PolynomialFeatures
>>> X = np.array([258.0, 270.0, 294.0,
…                          320.0, 342.0, 368.0,
…                          396.0, 446.0, 480.0,
…                          586.0])[:, np.newaxis]

>>> y = np.array([236.4, 234.4, 252.8,
…                         298.6, 314.2, 342.2,
…                         360.8, 368.0, 391.2,
…                         390.8])
>>> lr = LinearRegression()
>>> pr = LinearRegression()
>>> quadratic = PolynomialFeatures(degree=2)
>>> X_quad = quadratic.fit_transform(X)
```

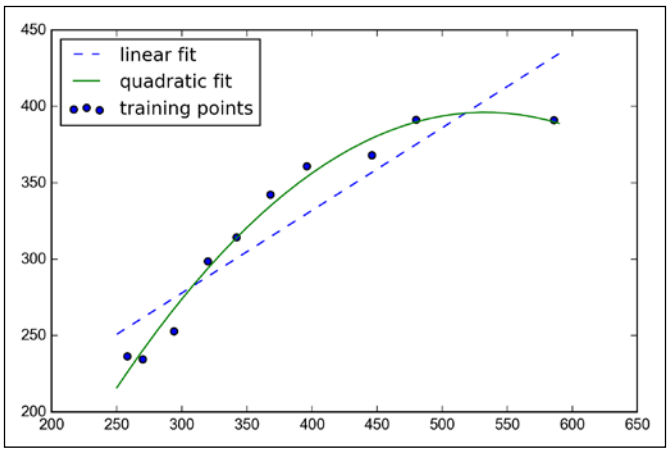
2. Fit a simple linear regression model for comparison:

```
>>> lr.fit(X, y)
>>> X_fit = np.arange(250,600,10)[:, np.newaxis]
>>> y_lin_fit = lr.predict(X_fit)
```

3. Fit a multiple regression model on the transformed features for polynomial regression:

```
>>> pr.fit(X_quad, y)
>>> y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
Plot the results:
>>> plt.scatter(X, y, label='training points')
>>> plt.plot(X_fit, y_lin_fit,
...          label='linear fit', linestyle='--')
>>> plt.plot(X_fit, y_quad_fit,
...          label='quadratic fit')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

In the resulting plot, we can see that the polynomial fit captures the relationship between the response and explanatory variable much better than the linear fit:




```
>>> y_lin_pred = lr.predict(X)
>>> y_quad_pred = pr.predict(X_quad)
>>> print('Training MSE linear: %.3f, quadratic: %.3f' % (
...         mean_squared_error(y, y_lin_pred),
...         mean_squared_error(y, y_quad_pred)))
Training MSE linear: 569.780, quadratic: 61.330
>>> print('Training R^2 linear: %.3f, quadratic: %.3f' % (
```

```
...          r2_score(y, y_lin_pred),
...          r2_score(y, y_quad_pred)))
Training R^2 linear: 0.832, quadratic: 0.982
```

As we can see after executing the preceding code, the MSE decreased from 570 (linear fit) to 61 (quadratic fit), and the coefficient of determination reflects a closer fit to the quadratic model ($R^2 = 0.982$) as opposed to the linear fit ($R^2 = 0.832$) in this particular toy problem.

Modeling nonlinear relationships in the Housing Dataset

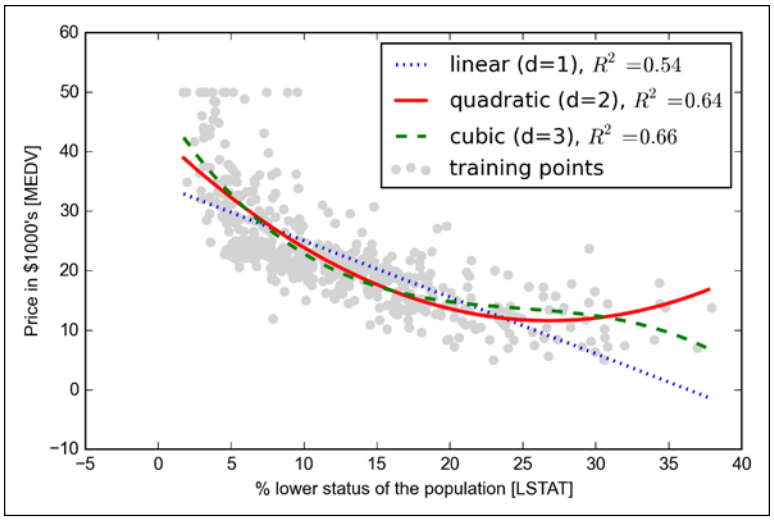
After we discussed how to construct polynomial features to fit nonlinear relationships in a toy problem, let's now take a look at a more concrete example and apply those concepts to the data in the *Housing Dataset*. By executing the following code, we will model the relationship between house prices and LSTAT (percent lower status of the population) using second degree (quadratic) and third degree (cubic) polynomials and compare it to a linear fit.

The code is as follows:

```
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> regr = LinearRegression()

# create polynomial features
>>> quadratic = PolynomialFeatures(degree=2)
>>> cubic = PolynomialFeatures(degree=3)
>>> X_quad = quadratic.fit_transform(X)
>>> X_cubic = cubic.fit_transform(X)

# linear fit
>>> X_fit = np.arange(X.min(), X.max(), 1)[:, np.newaxis]
>>> regr = regr.fit(X, y)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y, regr.predict(X))

# quadratic fit
>>> regr = regr.fit(X_quad, y)
>>> y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
>>> quadratic_r2 = r2_score(y, regr.predict(X_quad))

# cubic fit
```

```
>>> regr = regr.fit(X_cubic, y)
>>> y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
>>> cubic_r2 = r2_score(y, regr.predict(X_cubic))

# plot results
>>> plt.scatter(X, y,
...             label='training points',
...             color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...          label='linear (d=1), $R^2=%.2f$'
...            % linear_r2,
...          color='blue',
...          lw=2,
...          linestyle=':')
>>> plt.plot(X_fit, y_quad_fit,
...          label='quadratic (d=2), $R^2=%.2f$'
...            % quadratic_r2,
...          color='red',
...          lw=2,
...          linestyle='-')
>>> plt.plot(X_fit, y_cubic_fit,
...          label='cubic (d=3), $R^2=%.2f$'
...            % cubic_r2,
...          color='green',
...          lw=2,
...          linestyle='--')
>>> plt.xlabel('% lower status of the population [LSTAT]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.legend(loc='upper right')
>>> plt.show()
```

As we can see in the resulting plot, the cubic fit captures the relationship between the house prices and **LSTAT** better than the linear and quadratic fit. However, we should be aware that adding more and more polynomial features increases the complexity of a model and therefore increases the chance of overfitting. Thus, in practice, it is always recommended that you evaluate the performance of the model on a separate test dataset to estimate the generalization performance:

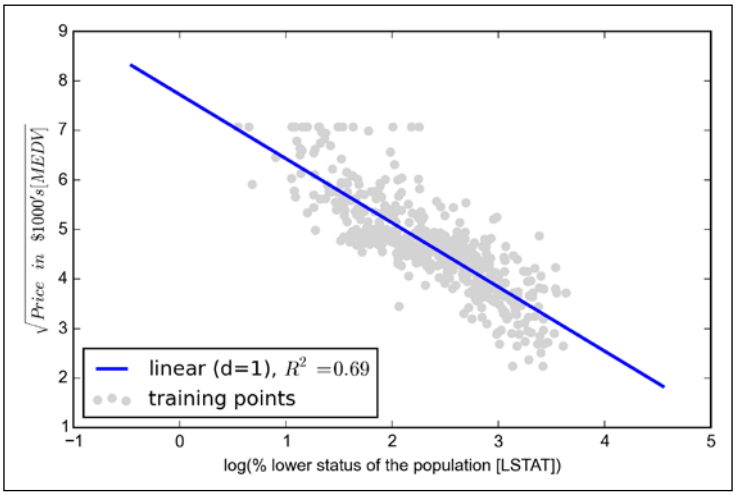


In addition, polynomial features are not always the best choice for modeling nonlinear relationships. For example, just by looking at the **MEDV-LSTAT** scatterplot, we could propose that a log transformation of the **LSTAT** feature variable and the square root of **MEDV** may project the data onto a linear feature space suitable for a linear regression fit. Let's test this hypothesis by executing the following code:

```
# transform features
>>> X_log = np.log(X)
>>> y_sqrt = np.sqrt(y)

# fit features
>>> X_fit = np.arange(X_log.min()-1,
...                   X_log.max()+1, 1)[:, np.newaxis]
>>> regr = regr.fit(X_log, y_sqrt)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y_sqrt, regr.predict(X_log))

# plot results
>>> plt.scatter(X_log, y_sqrt,
...             label='training points',
...             color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...          label='linear (d=1), $R^2=%.2f$' % linear_r2,
...          color='blue',
...          lw=2)
>>> plt.xlabel('log(% lower status of the population [LSTAT])')
```

```
>>> plt.ylabel('$\sqrt{Price \; in \; \$1000\'s [MEDV]}$')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

After transforming the explanatory onto the log space and taking the square root of the target variables, we were able to capture the relationship between the two variables with a linear regression line that seems to fit the data better ($R^2 = 0.69$) than any of the polynomial feature transformations previously:



Dealing with nonlinear relationships using random forests

In this section, we are going to take a look at **random forest** regression, which is conceptually different from the previous regression models in this chapter. A random forest, which is an ensemble of multiple **decision trees**, can be understood as the sum of piecewise linear functions in contrast to the global linear and polynomial regression models that we discussed previously. In other words, via the decision tree algorithm, we are subdividing the input space into smaller regions that become more *manageable*.

Decision tree regression

An advantage of the decision tree algorithm is that it does not require any transformation of the features if we are dealing with nonlinear data. We remember from *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, that we grow a decision tree by iteratively splitting its nodes until the leaves are pure or a stopping criterion is satisfied. When we used decision trees for classification, we defined entropy as a measure of impurity to determine which feature split maximizes the **Information Gain** (**IG**), which can be defined as follows for a binary split:

$$IG\left(D_p, x_i\right) = I\left(D_p\right) - \frac{N_{left}}{N_p} I\left(D_{left}\right) - \frac{N_{right}}{N_p} I\left(D_{right}\right)$$

Here, x is the feature to perform the split, N_p is the number of samples in the parent node, I is the impurity function, D_p is the subset of training samples in the parent node, and D_{left} and D_{right} are the subsets of training samples in the left and right child node after the split. Remember that our goal is to find the feature split that maximizes the information gain, or in other words, we want to find the feature split that reduces the impurities in the child nodes. In *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, we used *entropy* as a measure of impurity, which is a useful criterion for classification. To use a decision tree for regression, we will replace entropy as the impurity measure of a node t by the MSE:

$$I\left(t\right) = MSE\left(t\right) = \frac{1}{N_t} \sum_{i \in D_t} \left(y^{(i)} - \hat{y}_t\right)^2$$

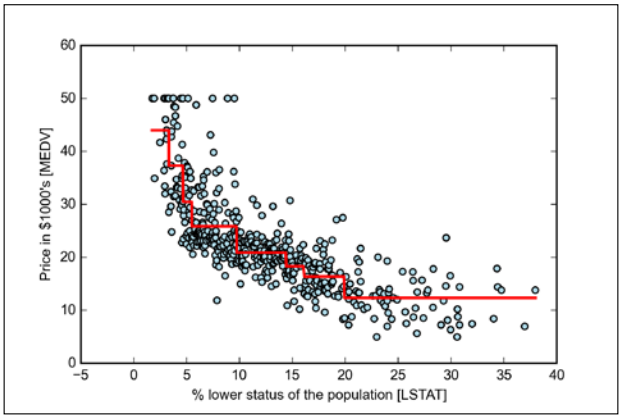
Here, N_t is the number of training samples at node t, D_t is the training subset at node t, $y^{(i)}$ is the true target value, and $\hat{y}_t$ is the predicted target value (sample mean):

$$\hat{y}_t = \frac{1}{N} \sum_{i \in D_t} y^{(i)}$$

In the context of decision tree regression, the MSE is often also referred to as within-node variance, which is why the splitting criterion is also better known as *variance reduction*. To see what the line fit of a decision tree looks like, let's use the `DecisionTreeRegressor` implemented in scikit-learn to model the nonlinear relationship between the **MEDV** and **LSTAT** variables:

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> tree = DecisionTreeRegressor(max_depth=3)
>>> tree.fit(X, y)
>>> sort_idx = X.flatten().argsort()
   >>> lin_regplot(X[sort_idx], y[sort_idx], tree)
>>> plt.xlabel('% lower status of the population [LSTAT]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.show()
```

As we can see from the resulting plot, the decision tree captures the general trend in the data. However, a limitation of this model is that it does not capture the continuity and differentiability of the desired prediction. In addition, we need to be careful about choosing an appropriate value for the depth of the tree to not overfit or underfit the data; here, a depth of 3 seems to be a good choice:



In the next section, we will take a look at a more robust way for fitting regression trees: random forests.

Random forest regression

As we discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, the random forest algorithm is an ensemble technique that combines multiple decision trees. A random forest usually has a better generalization performance than an individual decision tree due to randomness that helps to decrease the model variance. Other advantages of random forests are that they are less sensitive to outliers in the dataset and don't require much parameter tuning. The only parameter in random forests that we typically need to experiment with is the number of trees in the ensemble. The basic random forests algorithm for regression is almost identical to the random forest algorithm for classification that we discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*. The only difference is that we use the MSE criterion to grow the individual decision trees, and the predicted target variable is calculated as the average prediction over all decision trees.

Now, let's use all the features in the Housing Dataset to fit a random forest regression model on 60 percent of the samples and evaluate its performance on the remaining 40 percent. The code is as follows:

```
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test =\
...       train_test_split(X, y,
...                        test_size=0.4,
...                        random_state=1)

>>> from sklearn.ensemble import RandomForestRegressor
>>> forest = RandomForestRegressor(                              ..
.                                  n_estimators=1000,
...                                  criterion='mse',
...                                  random_state=1,
...                                  n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> y_train_pred = forest.predict(X_train)
>>> y_test_pred = forest.predict(X_test)
```

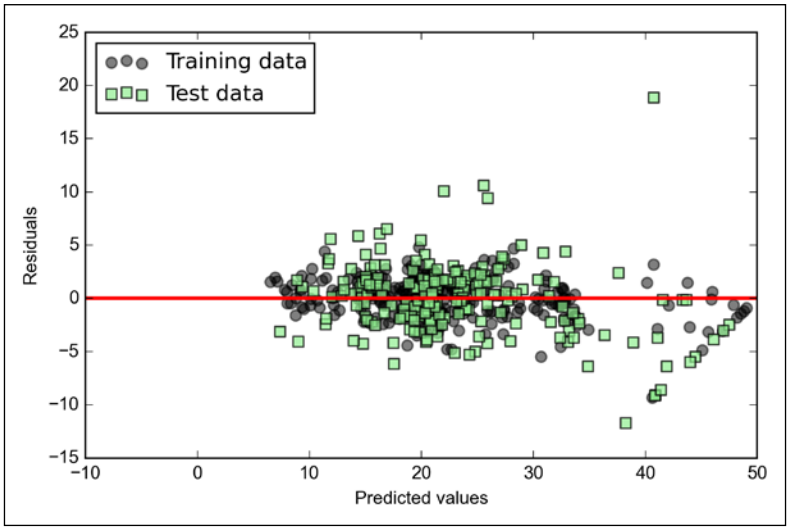
```
>>> print('MSE train: %.3f, test: %.3f' % (
...        mean_squared_error(y_train, y_train_pred),
...        mean_squared_error(y_test, y_test_pred)))
>>> print('R^2 train: %.3f, test: %.3f' % (
...        r2_score(y_train, y_train_pred),
...        r2_score(y_test, y_test_pred)))
MSE train: 1.642, test: 11.635
R^2 train: 0.960, test: 0.871
```

Unfortunately, we see that the random forest tends to overfit the training data. However, it's still able to explain the relationship between the target and explanatory variables relatively well ($R^2 = 0.871$ on the test dataset).

Lastly, let's also take a look at the residuals of the prediction:

```
>>> plt.scatter(y_train_pred,
...             y_train_pred - y_train,
...             c='black',
...             marker='o',
...             s=35,
...             alpha=0.5,
...             label='Training data')
>>> plt.scatter(y_test_pred,
...             y_test_pred - y_test,
...             c='lightgreen',
...             marker='s',
...             s=35,
...             alpha=0.7,
...             label='Test data')
>>> plt.xlabel('Predicted values')
>>> plt.ylabel('Residuals')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
>>> plt.xlim([-10, 50])
>>> plt.show()
```

As it was already summarized by the R^2 coefficient, we can see that the model fits the training data better than the test data, as indicated by the outliers in the y axis direction. Also, the distribution of the residuals does not seem to be completely random around the zero center point, indicating that the model is not able to capture all the exploratory information. However, the residual plot indicates a large improvement over the residual plot of the linear model that we plotted earlier in this chapter:



In *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, we also discussed the kernel trick that can be used in combination with **support vector machine** (**SVM**) for classification, which is useful if we are dealing with nonlinear problems. Although a discussion is beyond of the scope of this book, SVMs can also be used in nonlinear regression tasks. The interested reader can find more information about Support Vector Machines for regression in an excellent report by S. R. Gunn: S. R. Gunn et al. *Support Vector Machines for Classification and Regression*. (ISIS technical report, 14, 1998). An SVM regressor is also implemented in scikit-learn, and more information about its usage can be found at `http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR`.

Summary of Module 4 Chapter 8

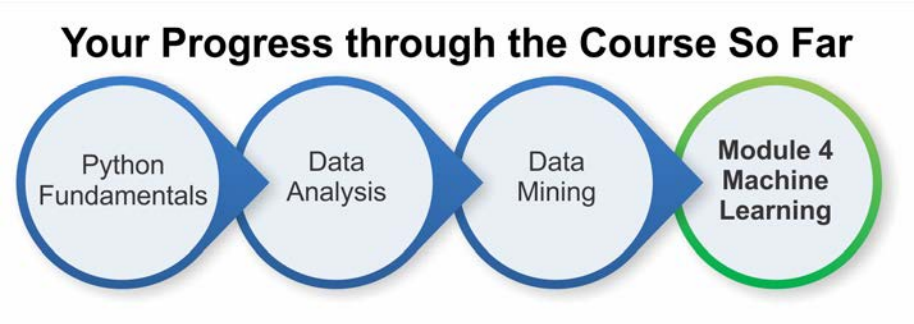
Ankita Thakur

Your Course Guide

At the beginning of this chapter, you learned about using simple linear regression analysis to model the relationship between a single explanatory variable and a continuous response variable. We then discussed a useful explanatory data analysis technique to look at patterns and anomalies in data, which is an important first step in predictive modeling tasks.

We built our first model by implementing linear regression using a gradient-based optimization approach. We then saw how to utilize scikit-learn's linear models for regression and also implement a robust regression technique (RANSAC) as an approach for dealing with outliers. To assess the predictive performance of regression models, we computed the mean sum of squared errors and the related 2 Rmetric. Furthermore, we also discussed a useful graphical approach to diagnose the problems of regression models: the residual plot.

After we discussed how regularization can be applied to regression models to reduce the model complexity and avoid overfitting, we also introduced several approaches to model nonlinear relationships, including polynomial feature transformation and random forest regressors.



A Final Run-Through

Here, we come to the end of our learning journey. Congratulations, you've done well so far! I hope you had a smooth journey and gained a lot of knowledge on Python and Data Science. If you ever wanted to get started with Python or you already knew about Python but wanted to explore more, this course was designed to help you achieve it. I'm sure you must have gained a lot of information and you'll start implementing it.

Now, let's have a small recap of what we have learned through this course. We covered four modules:

Ankita Thakur

Your Course Guide

In the first module, we discussed what classes and objects are and how they are used in Python. We learned about attributes and behaviors on Python objects, and also the organization of classes into packages and modules. We also saw how to protect our data. Then, we covered how to create our own exceptions and how to use exceptions for program flow control. Next, we covered tuples, dictionaries, lists, and sets, as well as a few more advanced collections. We also saw how to extend these standard objects. We looked at many useful built-in functions such as method overloading using default arguments. Finally, we covered design patterns, testing, and concurrency concepts.

Moving on to the next module, we saw what data analysis is and which libraries help us to perform the analysis task. We explored two powerful libraries of Python, NumPy and pandas. We also saw the common plotting functions for visualization using the Matplotlib API.

After this, we saw what data mining is and how to develop and design data mining applications with the help of Python. The best part was that in each chapter, we learned new algorithms and techniques and we created models to solve the real-world problems.

Ankita Thakur

Your Course Guide

Lastly, we explored the machine learning field. We introduced the main subareas of machine learning to tackle various problem tasks. We explored how to use different machine learning models to answer different questions about our data. Then, there was a gentle introduction to the fundamentals of pattern classification and we focused on the interplay of optimization algorithms and machine learning. After this, we described the essential machine learning algorithms for classification and provided practical examples using one of the most popular and comprehensive open source machine learning libraries, scikit-learn. We also discussed how to deal with missing data, one of the most common problems in unprocessed datasets.

Also, we had interesting challenges and quizzes throughout this course. How did you find them? Hope it was interesting. Keep writing to us in case you have any feedback or queries. I wish you all the best for your future projects.

Keep learning and exploring until we meet again!

Reflect and Test Yourself! Answers

Module 2: Data Analysis

Chapter 1: Introducing Data Analysis and Libraries

Q1	3
Q2	1
Q3	2

Chapter 2: Object-oriented Design

Q1	5
Q2	2
Q3	1
Q4	3

Chapter 3: Data Analysis with pandas

Q1	1
Q2	2
Q3	3
Q4	1
Q5	3

Chapter 4: Data Visualization

Q1	2
Q2	4
Q3	2
Q4	1

Chapter 5: Time Series

Q1	3
Q2	2
Q3	3
Q4	2

Chapter 6: Interacting with Databases

Q1	3
Q2	4
Q3	2

Chapter 7: Data Analysis Application Examples

Q1	1
Q2	2

Module 3: Data Mining

Chapter 1: Getting Started with Data Mining

Q1	2
Q2	1
Q3	4

Chapter 2: Classifying with scikit-learn Estimators

Q1	3
Q2	2
Q3	2
Q4	3

Chapter 3: Predicting Sports Winners with Decision Trees

Q1	2
Q2	1

Chapter 4: Recommending Movies Using Affinity Analysis

Q1	4

Chapter 5: Extracting Features with Transformers

Q1	2
Q2	3
Q3	3

Chapter 6: Social Media Insight Using Naive Bayes

Q1	2
Q2	2

Chapter 7: Discovering Accounts to Follow Using Graph Mining

Q1	2

Chapter 8: Beating CAPTCHAs with Neural Networks

Q1	3

Chapter 9: Authorship Attribution

Q1	2
Q2	1

Chapter 10: Clustering News Articles

Q1	3
Q2	2
Q3	3

Chapter 11: Classifying Objects in Images Using Deep Learning

Q1	3
Q2	2

Chapter 12: Working with Big Data

Q1	2
Q2	3
Q3	1

Module 4: Machine Learning

Chapter 1: Giving Computers the Ability to Learn from Data

Q1	4
Q2	2

Chapter 2: Training Machine Learning

Q1	3

Chapter 3: A Tour of Machine Learning Classifiers Using scikit-learn

Q1	2
Q2	3

Chapter 4: Building Good Training Sets – Data Preprocessing

Q1	1

Chapter 5: Compressing Data via Dimensionality Reduction

Q1	2

Chapter 6: Learning Best Practices for Model Evaluation and Hyperparameter Tuning

Q1	2
Q2	3

Chapter 7: Combining Different Models for Ensemble Learning

Q1	1

Chapter 8: Predicting Continuous Target Variables with Regression Analysis

Q1	2

Bibliography

This course is a blend of text and quizzes, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Python 3 Object-oriented Programming, Second Edition, Dusty Phillips*
- *Learning Python, Fabrizio Romano*
- *Getting Started with Python Data Analysis, Phuong Vo.T.H* and *Martin Czygan*
- *Learning Data Mining with Python, Robert Layton*
- *Python Machine Learning, Sebastian Raschka*

Thank you for buying

Python

Real-World Data Science

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.