Python Fundamentals

A practical guide for learning Python, complete with real-world projects for you to explore

Packt

www.packt.com

Ryan Marvin, Mark Ng'ang'a, Amos Omondi

Python Fundamentals

A practical guide for learning Python, complete with real-world projects for you to explore

Ryan Marvin, Mark Ng'ang'a, and Amos Omondi

Packt®

Python Fundamentals

Authors: Ryan Marvin, Mark Ng'ang'a, and Amos Omondi

Managing Editor: Taabish Khan

Acquisitions Editor: Koushik Sen

Production Editor: Nitesh Thakur

First Published: October 2018

Production Reference: 1311018

ISBN: 978-1-78980-732-5

Table of Contents

Preface

About

This section briefly introduces the author, the coverage of this book, the technical skills you'll need to get started, and the hardware and software requirements required to complete all of the included activities and exercises.

About the Book

Python Fundamentals takes you from zero experience to a complete understanding of the key concepts, edge cases, and how to use Python for real-world application development. You'll move progressively from the basics to work with larger, more complex applications. After completing this book, you'll have the skills you need to dive into an existing application or start your own project.

After a brief history on the Python language and the key differences between Python 2 and Python 3, you'll learn how Python has been used in applications such as YouTube and Google App Engine. As we work with the language, you'll learn about control statements, delve into controlling program flow, and gradually work on more structured programs via functions.

As you settle into the Python ecosystem, you'll learn about data structures and study ways to correctly store and represent information. By working through specific examples, you'll learn how Python implements OOP concepts of abstraction, encapsulation of data, inheritance, and polymorphism. You'll be given an overview of how imports, modules, and packages work in Python, how you can handle errors to prevent apps from crashing, as well as file manipulation. By the time you complete this book, you'll have built up an impressive portfolio of experience and armed yourself with the skills you need to tackle Python projects in the real world.

About the Authors

Ryan Marvin is a software developer with extensive experience in Python. He has also worked with JavaScript and a bit of PHP. He has built web scrapers, built APIs, and worked on frontend apps using React and Angular. In his own time, he works on mobile applications and likes to contribute to open source. Currently, he is working with Andela and specializes in building smart water grid software for one of their partners.

Mark Ng'ang'a is a software developer who specializes in web technologies. He has a bachelor's degree in computer science from Jomo Kenyatta University of Agriculture and Technology, Kenya. He has been programming for 6 years in Python, PHP, and JavaScript. Mark runs a software development and consultancy firm, Builtapp Ltd., that designs, plans, and builds software solutions for diverse business needs.

Amos Omondi is a software developer, who specializes in building applications for the web. He has a bachelor's degree in computer science from Kenyatta University, Kenya. He has several years of experience working with Python, and he has dabbled in languages such as PHP, Java, C, and built products using the MEAN stack.

Learning Objectives

- Understand how to use control statements
- Manipulate primitive and non-primitive data structures
- Use loops to iterate over objects or data for accurate results
- Write encapsulated and succinct Python functions
- Build Python classes using object-oriented programming
- Manipulate files on the file system (open, read, write, delete)

Audience

The *Python Fundamentals* course is great for anyone who wants to start using Python to build anything from simple command-line programs to complex scripts. You do not need any prior knowledge of Python.

Approach

This book takes a hands-on approach to demonstrate how Python can be used in production environments. It contains multiple activities that use real-life business scenarios for you to practice and apply your new skills in a highly relevant context.

Minimum Hardware Requirements

For an optimal student experience, we recommend the following hardware configuration:

- Processor: Intel Core i5 or equivalent
- Memory: 8 GB RAM
- Storage: 4 GB available space
- Internet connection

Software Requirements

You'll also need the following software installed in advance:

- Operating system: Windows 7 or Windows 10
- Browser: Google Chrome (latest version)
- Python 3.6+
- A text editor

Conventions

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Create a new file named **`binary_converter.py`**."

A block of code is set as follows:

```
>>> question = "Who was the first Beatle to leave the group?"
>>> len(question)
44
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "You can then click on **Download Data**."

Installation

Before we start, we're going to have to install Python on our machines first. You can go to https://www.python.org/downloads/windows/ and can click on the latest Python 3.6.x release download link. In our case, it's the Python 3.6.4 link.

Installing the Code Bundle

Copy the code bundle for the class to the **`C:/Code`** folder.

Additional Resources

The code bundle for this book is also hosted on GitHub at: https://github.com/TrainingByPackt/Python-Fundamentals.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

1 Introducing Python

Learning Objectives

By the end of this chapter, you will be able to:

- Use the Python interactive shell to write simple programs
- Write and run simple Python scripts
- Write and run dynamic scripts that take arguments from the command line
- Use variables and describe the different types of values that variables can be assigned
- Get user input from the keyboard for your Python programs
- Explain the importance of comments and write them in Python
- Explain the importance of whitespace and indentation in Python

This lesson introduces the Python programming language. We will work with the Python interpreter and write our first Python program.

Introduction

Python is a high-level, general-purpose programming language. It is notorious for having a very simple "pseudocode-like" syntax that places emphasis on readability and expressiveness. This not only makes code simpler to write but also easier to maintain. Additionally, it features a vast standard library that is augmented by an even larger array of third-party libraries. These are all developed and supported by Python's very active community.

Development is also faster in Python, as it is an interpreted language. This means that the instructions are interpreted at runtime and there's no need to pre-compile the program into machine language instructions. This makes for quick prototyping and experimentation. Python's interpreted nature, along with its dynamic typing system, are what really set it apart from languages such as Java or C++.

Python also supports multiple paradigms, such as the following:

- Object-oriented programming
- Functional programming
- Imperative programming
- Procedural programming

This versatility, coupled with Python's ability to run on all operating system platforms from Windows and GNU/Linux to macOS, have led to its popularity. As a matter of fact, today, Python comes built-in into most GNU/Linux distributions as well as macOS.

Python can be applied for writing automation scripts, machine learning, scientific computation, big data, web applications, GUI programming, IoT devices—just about anything. It's a multipurpose language and is easy to extend. Due to this, Python has been adopted by tech companies such as Google (for YouTube), Uber, Facebook, and Mozilla, further ensuring its support and development.

In this chapter, we will write our first Python program and play with the interpreter through the use of the Python interactive shell. We will also take a look at the different ways of running a Python program.

Python 2 Versus Python 3

Before we move on to getting our hands dirty, we'll take a brief look at the history of Python. Out there in the wild, you'll find codebases that use Python 3 or the older Python 2. The two are very similar. Generally, a lot of the code written for Python 3 will run on Python 2 and vice versa, but this should not be practiced as there are a few syntactic differences that can bring about issues. However, the majority of the differences between the two are under the bonnet.

Currently, support still runs for Python 2, but Python 3 is the only one in active development, meaning any new features brought to the language are only developed for Python 3. Additionally, the majority of commonly used third-party libraries have now ported to Python 3 and are withdrawing development for their Python 2 versions. For this reason, we will be using Python 3.6 for all of the examples in this book.

Working with the Python Interactive Shell

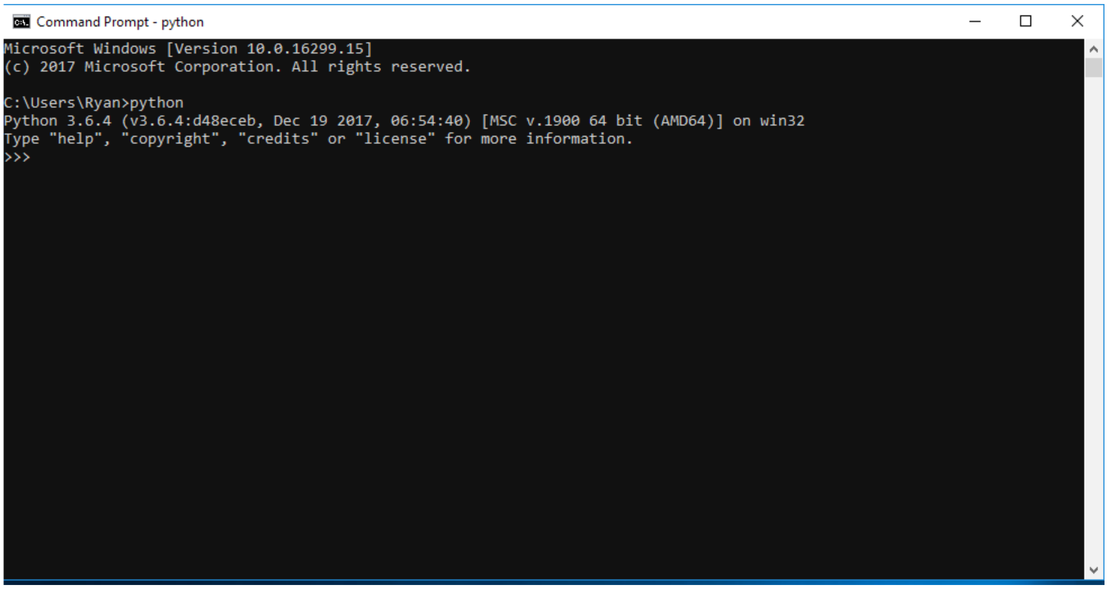
We are going to be writing our first program through the Python interactive shell. Before we begin, ensure that you have Python installed on your machine.

Exercise 1: Checking our Python Installation

To check whether Python is installed properly, perform the following steps:

1. Open the command prompt.
2. Type in the command **`python`** and press *Enter*. This should open the Python interpreter, and you should see a message containing the Python version you installed.

   Observe the output. It should be similar to what's shown in the following screenshot:



```
Microsoft Windows [Version 10.0.16299.15]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\Ryan>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 1.1: Checking the Python installation

Once the interactive shell opens, on the first line, you should see the Python version information. This includes its major version and the release date. As you can see from the preceding screenshot, it shows us that we're using Python 3, minor version 6, which was released on December 19, 2017. We can also see information on the system the interactive shell is running on. On the second line, we can see a few examples of commands we can write, and finally the prompt to enter a command **>>>** on the third line.

The Python interactive shell can be thought of as just any other shell that interfaces with the operating system (for example, Bash or CMD), but in this case, it interfaces with the Python interpreter. Through it, we can execute Python instructions. It presents a command-line interface.

Exercise 2: Working with the Python Interpreter

In this exercise, we will learn how to work with the Python interpreter:

1. At the command prompt, enter the following command to run the popular **`Hello World!`** program:

   ```
   >>> print("Hello World!")
   Hello World
   >>>
   ```

   Calling **`print`** logs the passed message to the standard output.

2. Anything you type into the shell is echoed back. Type **`"Echo"`** in the shell to check this:

   ```
   >>> "Echo"
   'Echo'
   >>> 1234
   1234
   >>>
   ```

3. You can even do some calculations (note that the order of operations is observed). Carry out a few mathematical operations in the terminal, like this:

   ```
   >>> 9 + 3
   12
   >>> 100 * 5
   20
   >>> -6 + 5 * 3
   9
   >>>
   ```

4. Exit the shell by running the **`exit`** command:

```
>>> exit
```

When you exit the Python interactive shell and relaunch it, you will notice that any variables you had defined or commands you had run in the previous session are gone. Therefore, we can't reuse them. Another way to run a Python program is by running code that has been saved in a file. This allows us to run many instructions at a time and also reuse them, as we'll see later on in the book. In the next section, we'll take a look at how to do this.

Activity 1: Working with the Python Shell

In this activity, we will perform some basic operations using the Python interactive shell.

The steps are as follows:

1. Open the terminal.
2. Open the interactive shell.
3. Print **`Happy birthday`** or any message to the standard output.
4. Run the operation **`17 + 35 * 2`**. The result should be **`87`**.
5. Print the numbers 1 to 9 on a single row.
6. Exit the shell.

> **Note**
>
> Solution for this activity can be found at page 276.

Writing and Running Simple Scripts

Running quick commands through the interactive shell is fun. It comes in handy when you have a quick hypothesis that you want to test out or when you want to check whether a specific method exists for some data type. However, you can't really write a full-fledged program through the interactive shell.

Python allows you to run your instructions from a saved file. A file containing Python instructions is called a **module**. A script is a module that can be run. Anything you can run on the interactive shell can be written and ran as a Python script.

By convention, Python scripts should have the file extension **`.py`**. The filename should be a valid filename, as defined by your operating system.

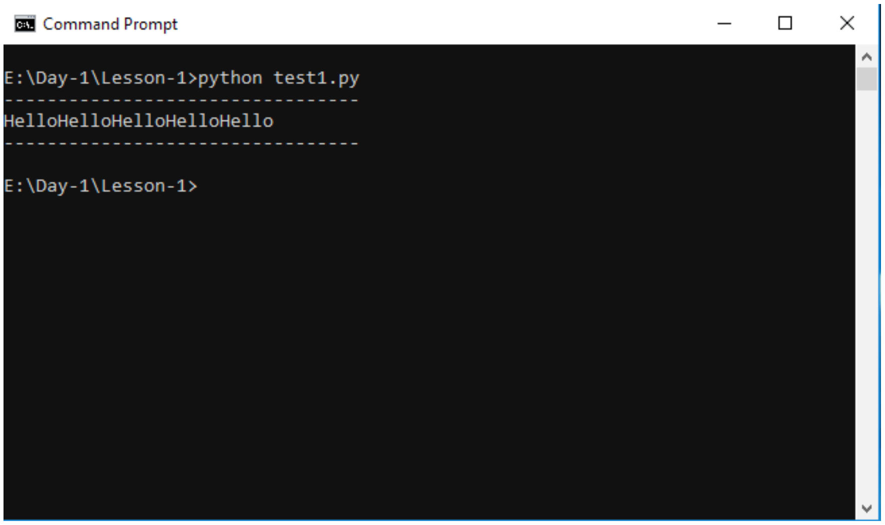
Exercise 3: Creating a Script

In this exercise, we will create a script that displays **`Hello`** five times in a single line:

1. Open your text editor.
2. Create a file called **`test1.py`** and insert the following code:

   ```
   print("--------------------------------")
   print("Hello " * 5)
   print("--------------------------------")
   ```

3. Save it to your working directory.
4. Open your terminal, change into the directory where the file is saved, and run the following command: **`python test1.py`**. You should see the following output:



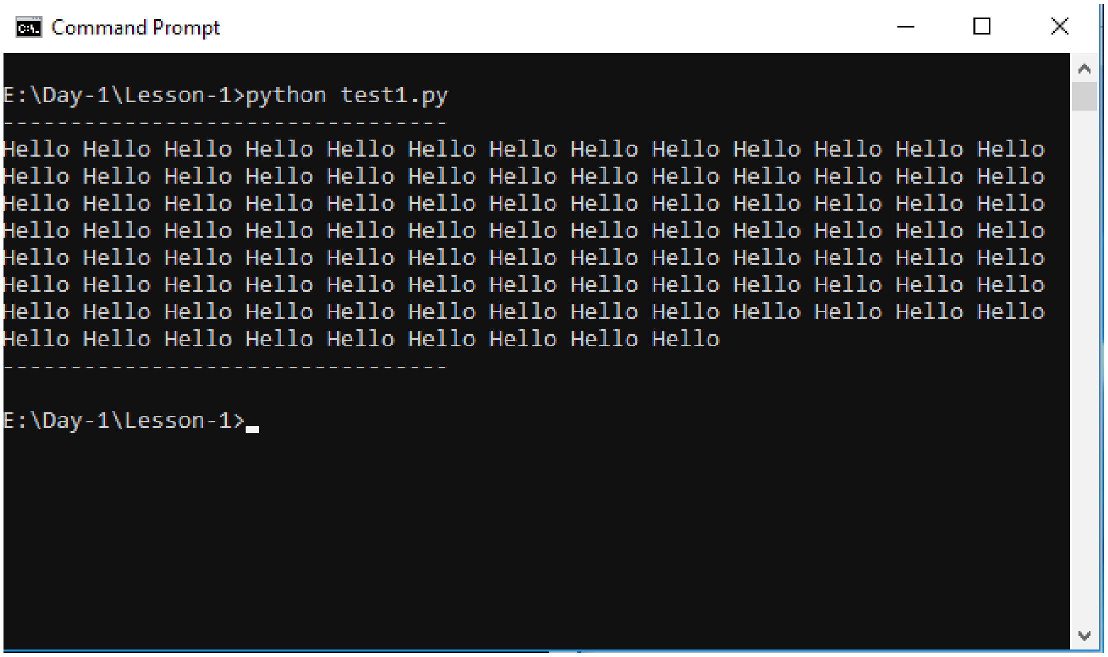
```
E:\Day-1\Lesson-1>python test1.py
--------------------------------
HelloHelloHelloHelloHello
--------------------------------

E:\Day-1\Lesson-1>
```

Figure 1.2: Creating and running a script

Python opens the file and executes each instruction, line by line. First, it runs the call to **`print`** on the first line and prints out a series of dashes. It then calls the second **`print`**, which prints our message five times, hence the **`* 5`** bit. This can be any value you want and is basically a shorthand way of saying, "repeat that string of characters an *n* number of times", *n* being **`5`** in our case. For example, if you change that **`5`** to **`100`**, it'll print **`Hello`** 100 times, as shown in the following code:

```
print("--------------------------------")
print("Hello " * 100)
print("--------------------------------")
```

This will be the output:



```
E:\Day-1\Lesson-1>python test1.py
--------------------------------
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello Hello Hello
--------------------------------

E:\Day-1\Lesson-1>
```

Figure 1.3: Creating and running the script with modified values

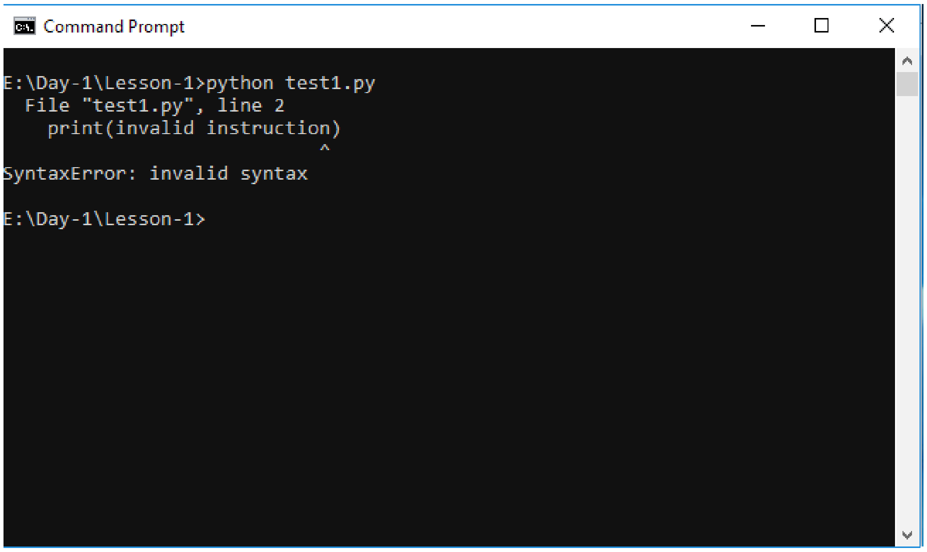
Finally, the last line is executed, just like our first, and prints out dashes. This execution is done in a blocking manner, so each line is executed after the previous line has completed running.

Running a File Containing Invalid Commands

As with the interactive shell, putting in invalid instructions also causes an error. Make the following changes to your file and run it:

```
print("--------------------------------")
print(invalid instruction)
print("--------------------------------")
```

You should see an error. This output is called a **stack trace**. It tells us useful things such as where the error happened, what kind of error it was, and what other calls were triggered along the way when we ran our command. Stack traces should be read from bottom to top. Another name for a stack trace is a **traceback**:



```
E:\Day-1\Lesson-1>python test1.py
  File "test1.py", line 2
    print(invalid instruction)
                            ^
SyntaxError: invalid syntax

E:\Day-1\Lesson-1>
```

Figure 1.4: Running the file with invalid syntax

The last line tells us what kind of error was raised, that is, a **`SyntaxError`**, meaning that our instructions were invalid. The line above it logs out the source line that caused the error, and the first line references our **`test1.py`** module where the line is. You'll be seeing different types of errors as you go through this book, and we'll have an in-depth look at errors and exception handling in one of the later chapters. For now, it is important that we understand how to read a stack trace and identify what is causing the error, and then act accordingly to fix it.

Exercise 4: Passing User Arguments to Scripts

To make a script more dynamic, you can have the user provide arguments to it when calling it:

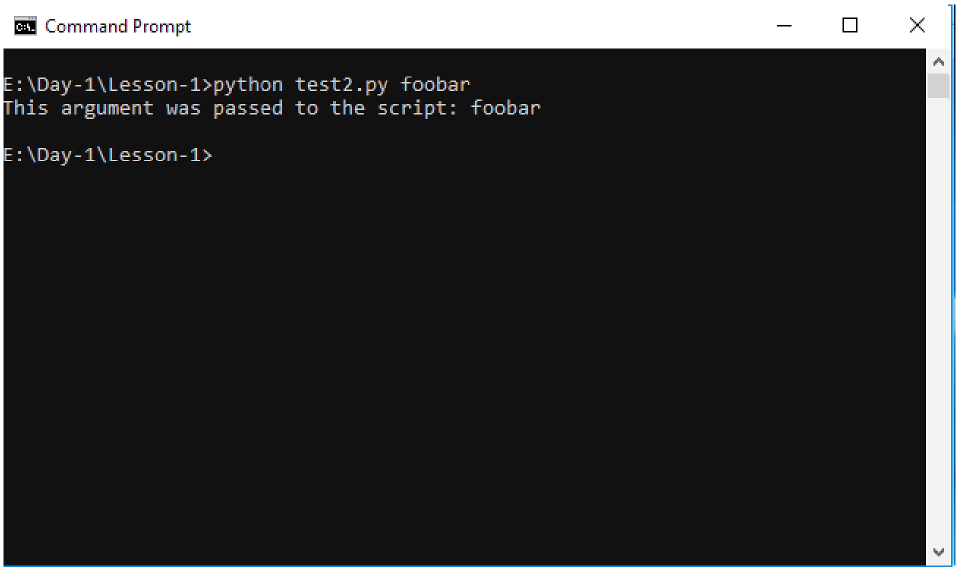
1. Create a new file called **`test2.py`** in the same directory that we created **`test1.py`** in. Then, add the following code:

   ```
   import sys

   print("This argument was passed to the script:", sys.argv[1])
   ```

2. Save and run the script as usual, passing it an argument, as illustrated here:

   ```
   python test2.py foobar
   ```

   The output should be as follows:



```
E:\Day-1\Lesson-1>python test2.py foobar
This argument was passed to the script: foobar

E:\Day-1\Lesson-1>
```

Figure 1.5: Passing arguments to the script

Note

Running this script without any arguments will raise an **`IndexError`**.

We have some new syntax in this script. We won't go over all of it in detail in this chapter, but for the purposes of this explanation, **`import sys`** imports the **`sys`** module that's built into Python into our module. This module provides access to Python interpreter functions. For our purposes, we're using it to read command-line arguments that have been passed to the interpreter when invoking our script.

> **Note**
>
> We will go over imports and the Python standard library in full detail in *chapter 8, Modules, Packages, and File Operations*.

When we call **`sys.argv[1]`**, we're asking for the first argument that's been passed when running the script. Generally, you can pass as many arguments as you like by separating each argument with a blank space.

Activity 2: Running Simple Python Scripts

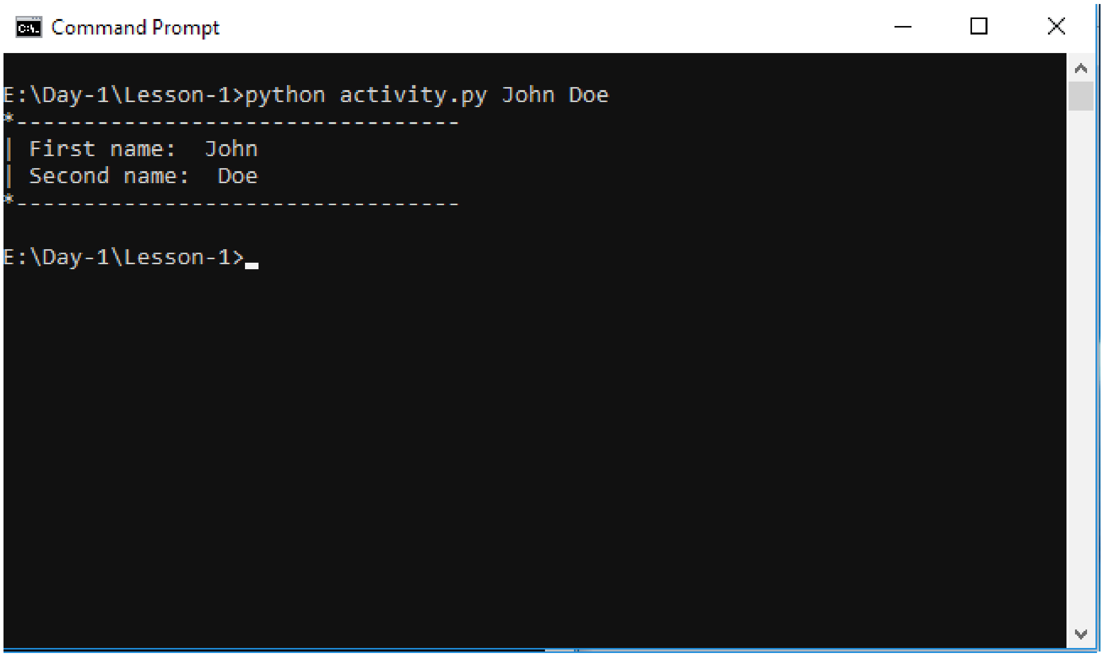
In this activity, we will create a name card generator script that, when called with a first name and last name, will generate a name card with the names.

The steps are as follows:

1. Open your editor and create a script named **`activity.py`**.
2. Use two **`print`** statements to print the **`First name`** and **`Last name`**. Also, use these **`print`** statements to print 20 underscores as borders above and below the names.
3. Two parameters should be passed with the script: one for the first name and the second for the last name.

4. Run the script by passing two string arguments.

   Your output should be similar to the following:



```
E:\Day-1\Lesson-1>python activity.py John Doe
*---------------------------------
| First name:  John
| Second name:  Doe
*---------------------------------

E:\Day-1\Lesson-1>_
```

Figure 1.6: Running simple scripts

Note

Solution for this activity can be found at page 276.

Note

Python comes with its own **IDE** (**Integrated Development Environment**) known as **IDLE**. IDLE comes with an editor and interactive shell that supports syntax highlighting, a debugger, and a handful of other practical features. Since it comes with your Python installation, it's ready to use immediately and can help improve your productivity as it offers a larger set of capabilities compared to a bare editor and shell. More information about it can be found on the Python website at https://docs.python.org/3/library/idle.html.

Python Syntax

In this section, we'll be taking a look at how the Python language expressions are structured. Essentially, we will learn what it takes to write a valid Python program.

Variables

As we know, variables are references to values in memory.

Variables in Python can reference values of different data types such as strings, integers, floating point values, Booleans, and different data types and data structures, as we'll see later on in this book. Python, in contrast to statically typed languages such as Java or C++, doesn't require you to pre-declare a variable's data type. It determines the type during runtime.

You can think of a variable as a box with a named label on it. The box on its own has no value but becomes valuable once you put something inside it. The box represents the things inside it and, similarly, a variable is used to represent the value inside it.

Additionally, a variable's value and type can change during runtime. Any variable can be used to store any data type and can be used as long as it has already been defined. Before we begin taking a look at how to assign variables to values, let's briefly go over the different types of values/data types we've encountered thus far and the ones we'll be dealing with in this chapter.

Values

Python supports several different types of values. These values are what variables can be assigned to. Thus far, we've encountered, strings and numeric values such as integers.

Numeric Values – Integers

Mathematically, integers are whole numbers that are either positive or negative. The same definition is applicable for Python integers.

Here is an example of an integer expression in Python. As we saw earlier, Python echoes whatever you write in the interactive shell:

```
Python 3.6.0 (default, Dec 24 2016, 08:01:42)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 7
7
>>>
```

We also saw the different symbols, such as **+**, *****, and **-**, that can be used to perform different arithmetic operations on the integer values:

```
>>> 5 + 4 + 6 + 9

24

>>> 5 * 5

25

>>> 42 - 16

26

>>>
```

Exercise 5: Checking the Type of a Value

We can also check the type of a value by using the **type** function that's built into Python:

1. Open the Python interactive shell.
2. Enter the following code to view the type of the numeric value **7**. Observe the output:

   ```
   >>> type(7)
   <class 'int'>
   >>>
   ```

   As you can see, it tells us that the type of 7 is **int** (short for integer).
3. Now, enter the following code, and observe the output:

   ```
   >>> type(4+3)
   <class 'int'>
   >>>
   ```
4. Enter the following code at the prompt, and observe the output:

   ```
   >>> type("7")
   <class 'str'>
   >>>
   ```

5. Enter the following code at the prompt, and observe the output:

   ```
   >>> type('7')
   <class 'str'>
   >>>
   ```

> **Note**
>
> The part in the output before **`int`** says **`class`** because everything in Python is an object.

There are a few other numeric types of values, such as floating-point numbers, but we'll be taking a look at those in the next chapter.

String Values

Another type of value that we've seen in the previous section was a string value. This is a sequence of characters that's placed in between two quotation marks, for example, **`"January"`**, **`"Chops Maloy"`**, and **`'UB40'`**. You can use both double and single quotes to denote strings. Strings can contain numbers, letters, and symbols, like so:

```
>>> type("3 Musketeers")

<class 'str'>

>>> type('First Order')

<class 'str'>

>>>
```

As you can see, it tells us the type of the value **`"3 Musketeers"`** is **`str`** (short for string).

Type Conversion

Sometimes, you may have a string with an integer inside it or an integer that you want to put in a string. The first scenario often happens with user input where everything is returned inside a string. To be able to use it, we need to convert it to the desired data type.

Python allows you to convert string type values to integer type values and vice versa. Using the built-in **`str`** function, you can convert an integer to a string:

```
>>> str(7)

'7'

>>>
```

Strings can also be converted to integers, as long as they hold a valid integer value within. This is done by use of the built-in **`int`** function:

```
>>> int("100")
100
>>>
```

An error occurs if we try converting a string that doesn't contain an integer. Here, the string **`"Foobar"`** can't be converted because it's a string of letters. **`"3.14159"`** also fails because it is a float, and not an integer:

```
>>> int("Foobar")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Foobar'
>>> int("3.14159")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '3.14159'
>>>
```

These are the basic types of values we'll be dealing with in this chapter.

Exercise 6: Assigning Variables

In this exercise, we will learn how to assign a value to a variable:

1. Assign a value to a variable in Python using the following syntax:

   ```
   >>> number = 7
   >>>
   ```

2. Print the variable to the standard output; this should reveal its value:

   ```
   >>> print(number)
   7
   >>>
   ```

3. However, if we try using a variable before assigning it a value, the Python interpreter will raise an error. Check this as follows:

   ```
   >>> del number
   >>> print(number)
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   NameError: name 'number' is not defined
   >>>
   ```

   On the first line, we're using a new statement: **`del`**. **`del`** unbinds a name/variable (Python refers to variables as names) from the current namespace. Calling **`del number`** thus deletes the variable **`number`** from the current namespace.

> **Note**
>
> A namespace is a mapping of names/variables to their values.

This means that the reference **`number`** is removed and no longer points to the value **`7`**. When we try printing out the now nonexistent variable, we get an error stating that the name **`number`** is not defined.

Exercise 7: Using Variables

We use variables when we have a value in our code that we want to use multiple times. They prevent us from having to repeat that value each time we want to use it, as variables store the values in memory.

When we store values in memory, we can reuse them as many times as we'd like:

1. Assign the value **`7`** to the **`number`** variable:

   ```
   >>> number = 7
   ```

2. We can now use this variable for any operations we'd like. Print out the value of the **`number`** variable, multiplied by **`5`**:

   ```
   >>> number * 5
   35
   ```

3. Print out **`number`** added to **`2`**:

   ```
   >>> number + 2
   9
   ```

4. Print out **number** divided by **3.5**:

```
>>> number / 3.5
2.0
```

5. Print out **number** subtracted from itself:

```
>>> number - number
0
```

6. Note that, despite having used it, the value of **number** won't change unless we reassign it. Reassigning **22** to **number** changes its value and verifies this:

```
>>> print(number)
7
>>> number = 22
>>> print(number)
22
>>>
```

7. You can also assign the resulting value of another operation to a variable. Do this as follows:

```
>>> number = 7
>>> x = number + 1
>>> x
8
>>>
```

8. String values can also be assigned and used in a similar fashion. First, set the **message** variable to the string **"I love Python"**:

```
>>> message = "I love Python"
```

9. Print out the value of the **message** variable and add an exclamation point at the end:

```
>>> message + "!"
'I love Python!'
```

10. Print out **message** plus three exclamation points:

```
>>> message + "!" * 3
'I love Python!!!'
>>>
```

Here, we can see the application of a new operation to strings: **+**. We use this whenever we want to concatenate (add together) two strings.

This only applies to strings, and thus trying to concatenate a string with any other data type will raise an error. We shall look at it in greater depth in the next chapter.

An interesting phenomenon with Python variables is that they are not deeply linked:

```
>>> x = 1
>>> y = x
>>> x = 2
>>> print(x)
2
>>> print(y)
1
>>>
```

A behavior you'd expect would be that **y** being assigned to **x** would change upon changing **x**, but it doesn't and stays the same. What do you think is happening?

Since Python variables point to values in memory, when **y** is assigned to **x**, it does not make an alias for **x** but instead points the variable **y** to where the value of **x**, 1, is. Changing **x** changes its pointer from 1 to 2, but **y** remains pointing to its initial value:

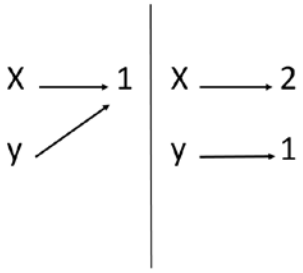


Figure 1.7: Variable assignment

Multiple Assignment

In Python, you can also assign multiple variables in one statement, like so:

```
>>> a, b, c = 1, 2, 3
>>> print(a)
1
>>> print(b)
2
>>> print(c)
3
>>>
```

The assignment works so that the first variable, **a**, is assigned the first value, **1**, after the **=** sign. The second variable, **b**, is assigned the second value, **2**, and the third variable, **c**, is assigned the third value, **3**.

What happens if we try to assign more variables than we pass values?

```
Python 3.6.0 (default, Dec 24 2016, 08:01:42)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a, b, c = 1, 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 3, got 2)
>>>
```

The Python interpreter raises an error and tells us it didn't get enough values to assign to the variables we declared in our statement.

A similar error is raised when we try to assign more values than there are variables:

```
>>> a, b = 1, 2, 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
>>>
```

Activity 3: Using Variables and Assign Statements

Write a script that will use distance in kilometers and time in hours to calculate and print out speed in kilometers per hour, miles per hour, and meters per second.

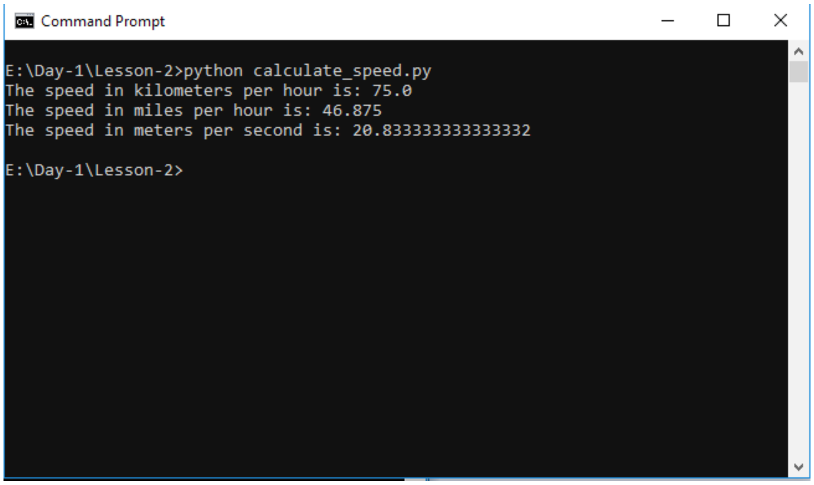
Here are some hints:

- The formula for calculating speed is *distance/time = speed.*
- To convert kilometers to miles, divide the kilometers by 1.6.
- To convert kilometers to meters, multiply the kilometers by 1,000.
- To convert hours to seconds, multiply hours by 3,600.

The steps are as follows:

1. Open your editor.
2. Create a file named **`calculate_speed.py`** and save it.
3. On the first two lines, declare two variables for the distance in kilometers and time in hours and assign the values **`150`** and **`2`**, respectively.
4. In the next two lines, calculate the distances in miles and distance in meters based on the distance in kilometers.
5. Then, calculate the time in seconds based off the time in hours.
6. Next, calculate the speed in kilometers per hour, speed in miles per hour, and speed in meters per second.
7. Finally, add **`print`** statements to print out our results.
8. Save our script and run it by using the **`python calculate_speed.py`** command.

The output should look like this:



```
E:\Day-1\Lesson-2>python calculate_speed.py
The speed in kilometers per hour is: 75.0
The speed in miles per hour is: 46.875
The speed in meters per second is: 20.833333333333332

E:\Day-1\Lesson-2>
```

Figure 1.8: Output of running the calculate_speed.py script

Note

Solution for this activity can be found at page 276.

So far, we've learned how variables are used in Python and a few of the different values you can assign to them. We've also learned how to use variables in our programs. In the next section, we will be looking at the rules for naming variables.

Naming Identifiers and Reserved Words

Python, like other languages, has a couple of rules on naming identifiers such as variable names, class names, function names, module names, and other objects. Some are strictly enforced by the interpreter, while others are simply by convention, and developers are at liberty to ignore them. The rules and conventions are designed to avoid confusion when the interpreter is parsing through code or to make the code more easily readable by humans.

We'll start off by going through some of the rules for naming variables and other identifiers:

- An identifier can consist of upper and lowercase letters of the alphabet, underscores, unicode identifiers, and digits 0 to 9.

> **Note**
>
> As per the Python documentation, you can find the list of permitted unicode identifiers here: https://www.dcl.hpi.uni-potsdam.de/home/loewis/table-3131.html.

- An identifier cannot begin with a digit; for example, **7x** is an invalid variable name.
- No other characters can be in identifiers. This means spaces or any other symbols. Spaces most notably occur in module names as some operating systems will permit filenames with spaces. This should be avoided.
- A Python keyword cannot be used in an identifier name, for example, **import**, **if**, **for**, and **lambda**.

The following are examples of valid variable definitions:

```
>>> meaning_of_life = 42
>>> COUNTRY = "Wakanda"
>>> Ω = 38.12
>>> myDiv = "<div></div>"
>>> letter6 = "f"
>>>
```

It should also be noted that Python identifier names are case sensitive. Consider this:

```
>>> foobar = 5
```

This is a different variable from the following one:

```
>>> Foobar = 5
```

Exercise 8: Python Keywords

Certain names in Python cannot be used as they are parts of the language syntax. Such words are known as **reserved words** or **keywords**. An example of a reserved word is **`import`**, which is a statement used for when you want to import a module into your code.

Python has several keywords. To get the full list, perform the following steps:

1. Open the Python interactive shell:

   ```
   Python 3.6.0 (default, Dec 24 2016, 08:01:42)
   [GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
   Type "help", "copyright", "credits" or "license" for more information.
   >>>
   ```

   In the output, you will see a line saying **`Type "help", "copyright", "credits" or "license" for more information`**.

2. Run **`help()`** to open the **`help`** utility.

3. Type **`keywords`** in the prompt:

   ```
   help> keywords
   ```

   You should see the following output:

   ```
   Here is a list of the Python keywords. Enter any keyword to get more help.

   False               def                 if                  raise
   None                del                 import              return
   True                elif                in                  try
   and                 else                is                  while
   as                  except              lambda              with
   assert              finally            nonlocal            yield
   break               for                 not
   class               from                or
   continue            global              pass

   help>
   ```

4. Quit the **`help`** utility by typing **`quit`**. This should take you back to the interpreter prompt:

   ```
   help> quit
   ```

You should not use any word from this list of keywords as an identifier name. Note that you don't have to remember them all as the Python interpreter will restrict you from using them.

In the following example, we are trying to use the keyword **`for`**, but we get a syntax error upon doing so. The same applies for all keywords:

```
>>> for = "Elise"
  File "<stdin>", line 1
    for = "Elise"
        ^
SyntaxError: invalid syntax
>>>
```

Python Naming Conventions

Python has several guidelines for naming identifiers that aren't enforced by the interpreter. These guidelines are meant for consistent code and making it more readable. Please note that these are simply guidelines and the programmer is at liberty to ignore them.

It should be counterpointed that ignoring naming conventions eventually leads to a road of regret as they provide several advantages, such as the following:

- Makes the code easier to read and understand for other programmers, since they'd find consistent and instantly recognizable patterns
- Enhances clarity and reduces ambiguity
- Makes automated refactoring easier, as the automation tools would have consistent patterns to look for
- Provides additional information about the identifiers; for example, when you see a variable name in all caps, you immediately know that it is a constant

Some of these conventions and guidelines are given here.

Compound variable names should be written in **snake_case** notation.

Prefer this:

```
>>> first_letter = "a"
```

Over this:

```
>>> firstLetter = "a"  # camelCase
>>> FirstLetter = "a"  # PascalCase
```

Naming for constants should be written in capital letters to denote that their values are not meant to change. In reality, though, Python has no way of restricting the value of a constant from being changed as they are variables, just like any other:

```
>>> NUMBER_OF_PLANETS = 8
>>> RADIUS_OF_THE_EARTH_IN_KM = 6371
```

Note that we can change a constant's value:

```
>>> NUMBER_OF_PLANETS = 9
>>>
```

Avoid lower case `l` or uppercase `O` as single character variable names, as in some fonts, these letters can be mistaken for 1 and 0, respectively, making the code harder to read.

> **Note**
>
> For a more in-depth look at Python naming conventions, visit https://www.python.org/dev/peps/pep-0008/#naming-conventions.

In the next section, we'll learn about comments, their importance, and how to write them in Python.

Activity 4: Variable Assignment and Variable Naming Conventions

Write a script that will calculate the area and circumference of a circle with a radius of 7. In this activity, we'll get better acquainted with variable assignment as well as variable naming conventions.

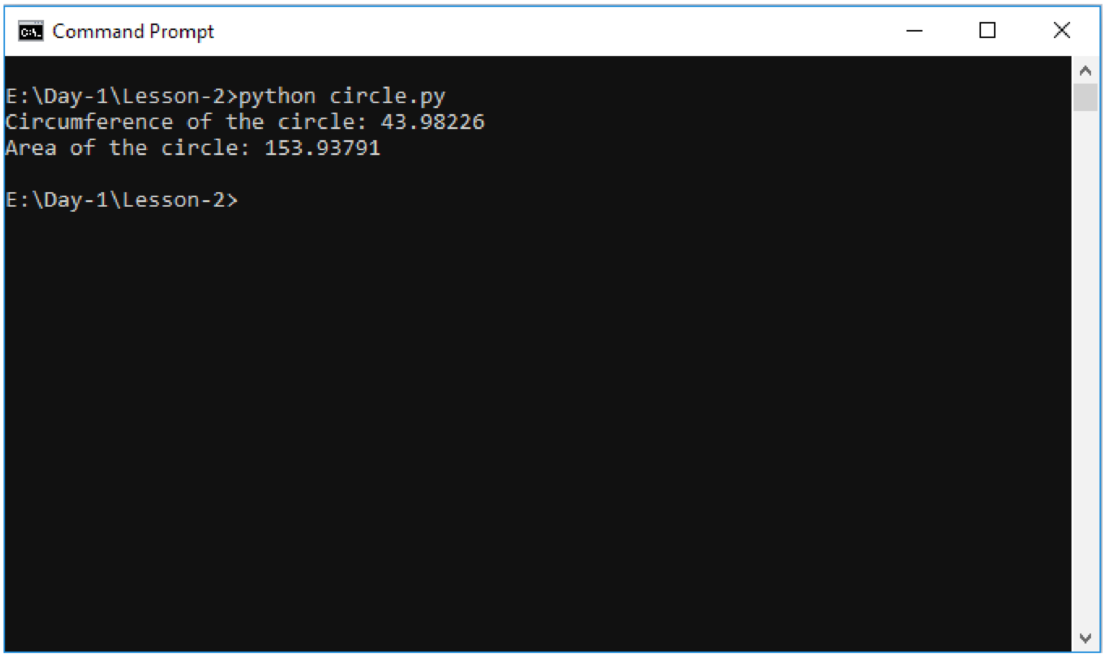
Here are some hints:

- The formula for calculating area is $\pi * r^2$, r being the radius.
- The formula for calculating circumference is $2 * \pi * r$.
- π can be approximated to 3.14159.

The steps are as follows:

1. Open your editor.
2. Create a file named `circle.py` and save it.
3. On the first two lines, declare our constant, π (`PI`), and the radius of the circle with a value of `7`.
4. Write the lines to run our calculations.
5. Display the results by using `print` statements.
6. Save the script and run it by using the `python circle.py` command.

The output should look like this:



```
E:\Day-1\Lesson-2>python circle.py
Circumference of the circle: 43.98226
Area of the circle: 153.93791

E:\Day-1\Lesson-2>
```

Figure 1.9: Output of running the circle.py script

Note

Solution for this activity can be found at page 277.

User Input, Comments, and Indentations

In this section, we will look at how we can take user input from the keyboard, how to write comments, and the importance of indentation while writing Python code.

User Input from the Keyboard

Python has a very handy function for obtaining user keyboard input from the CLI called **`input()`**. When called, this function allows the user to type input into your program using their keyboard. The execution of your program pauses until the user presses the *Enter* key after completing the input. The user's input is then passed to your program as a string that you can use. The following is an example of this:

Declare the following variable:

```
>>> message = input()
```

The program execution will halt until you input a value and hit the *Enter* key:

```
Peter Piper picked a peck of pickled peppers

>>>
```

The **`message`** variable was assigned to the value that we passed. Let's print it out:

```
>>> print(message)

Peter Piper picked a peck of pickled peppers

>>>
```

Passing in a Prompt to the input Function

You may have noted that there is no cue that lets us know when to type or what to type. You can pass a prompt to the **`input`** function to do this. The prompt is written to standard output without a trailing newline and serves as a cue for the user to pass in their input. Let's view this in action.

Declare the following variable, passing in the prompt **`Enter a tongue twister:`** to the **`input`** function. This will be displayed to the user:

```
>>> tongue_twister = input("Enter a tongue twister: ")

Enter a tongue twister:
```

Let's type in our tongue twister and hit the *Enter* key. The **tongue_twister** variable will then be set, and printing it out should reveal the value we passed:

```
Enter a tongue twister: She sells seashells by the seashore
>>> print(tongue_twister)
She sells seashells by the seashore
>>>
```

Using Different Input Data Types in your Program

The values returned by the **input** function are always strings. Sometimes, a string may not be the data type you're expecting in your program. As we saw earlier, you can convert strings to integers using the built-in **int** function. To retrieve integer values from user input, all your program needs to do is cast the string that is returned as a value by the **input** function. Let's take a look at an example of this:

```
>>> number = int(input("Find the square root of: "))
Find the square root of: 49
>>> print("The square root of", number, "is", number ** 0.5 )
The square root of 49 is 7.0
```

Here, we take the user's input which is a string, convert it to an integer using the **int** function, and print out its square root.

Exercise 9: Fetching and Using User Input

In this exercise, we will aim to get better acquainted with fetching and using user input as well as practice multiple assignment:

1. Open your interpreter.
2. Declare the following variables:

   ```
   >>> name, hobbies = input("What is your name? "), input("What are your hobbies? ")
   ```

3. Type in the values when prompted:

   ```
   What is your name? John Doe
   What are your hobbies? swimming, reading, playing instruments, drawing, programming
   >>>
   ```

4. Print out the values of the variables:

   ```
   >>> print("Your name is", name, "and your hobbies are", hobbies)
   Your name is John Doe and your hobbies are swimming, reading, playing
   instruments, drawing, programming
   ```

We can get user input to use in our program in Python by using the built-in **`input`** function. You can pass a prompt argument to the **`input`** function that'll act as a cue for the user to pass their input. The **`input`** function always returns values as strings that we can optionally cast to the data type we desire.

Comments

Comments are an integral part of programming.

Comments can be thought of as notes on the code that give us more contextual information about it. They can tell us why certain decisions were made, some improvements that can be made in future, and also explain the business logic. In short, they make the code easier to understand for humans.

There are three different ways to write Python comments, **documentation strings** (**docstrings** for short), inline comments, and block comments.

Block and Inline Comments

Block and inline comments start with a pound sign, **`#`**. A **block comment** comes in the line before the statement it annotates and is placed at the same indentation level:

```
# increment counter

counter = counter + 1
```

Inline comments are placed on the same line as the statement it annotates:

```
>>> print(foobar)  # this will raise an error since foobar isn't defined
```

Block comments are applied to the code that follows them and should be indented at the same level as the code it's meant for. While they serve the same purpose as inline comments, block comments are to be preferred by convention as they are more immediately noticeable and unambiguous as to the code they are annotating.

Documentation Strings

A documentation string, or docstring for short, is a literal string used as a Python comment. It is written and wrapped within triple quotation marks; `"""` or `'''`. Docstrings are often used to document modules, functions, and class definitions. The following is an example module that's been documented with a docstring. Module docstrings should be put at the beginning of the file:

```
"""
This script can be called with two integer arguments to return their sum
"""
import sys
num_1  = int(sys.argv[1])
num_2  = int(sys.argv[2])


print(num_1, "+", num_2, "=", num_1 + num_2)
```

Furthermore, you can also use docstrings to write multiline comments:

```
"""
This loop goes through all the numbers from 1 to 100
printing each out
"""
for i in range(1, 101):
    print(i)
```

Sufficiently commenting our code leads to easily maintainable and easier-to-read code. Since we often spend more time reading code than writing it, it is important that we write comments, but even more important is that we write code readable enough to need as few explanations as possible.

Indentation

A **block** is a group of statements that are meant to be executed together. Blocks are a fundamental aspect of modern programming languages since flow of control structures (structures that determine the flow of execution of code) are formed from blocks. Blocks allow a set of statements to be executed as though they were a single statement. Different languages represent blocks differently, but most commonly, the following syntax is used:

```
if (true) {
  // execute this block of statements
} else {
  // execute other block of statements
}
```

> **Note**
>
> **`if`** is a common control flow structure that evaluates Boolean expressions and determines where the program should go.

In the preceding code snippet, a block is denoted by whatever is inside the curly braces. If the condition is true, then execute a certain block of statements; otherwise, execute the other block. Often, blocks can be nested within other blocks.

In Python, statements are grouped using whitespace, that is, blocks are indented within other blocks instead of using curly braces.

> **Note**
>
> Whitespace is any character in a piece of text that occupies space but doesn't correspond to a visible marking.

Observe the following, which is the Python equivalent of the previous code snippet:

```
if True:
    # execute this block of statements
    print("Block 1")
else:
    # execute other block of statements
    print("Block 2")
```

Python uses whitespace to denote blocks. Just as blocks are denoted by statements enclosed in curly braces in several other languages, any statement that's indented in Python forms a new block, becoming a child of the previous statement with the parent statement being suffixed with a colon, `:`. In place of a closing curly brace, a statement that would come after that block would be outdented outside it, as we've seen with the **`else`** statement that came after the **`if`** block.

Generally, you can use any number of spaces as long as every statement in the block has an equal amount of indentation, although the standard is four spaces:

```
if True:
    x = 5
    y = x
    print(x, y ** 2)
```

Exercise 10: The Importance of Proper Indentation

If we try running a piece of code without proper indentation, Python will raise an error. Let's take a look at this:

1. Open your interpreter and type the following:

   ```
   >>> if True:
   ```

2. Press the *Shift* + *Enter* keys to create a new line. The interpreter will show three dots **…** in place of the usual **>>>** prompt, which means you're on another line. Type four spaces and then write your statement:

   ```
   >>> if True:
   ...     print(1)
   ```

3. We'll then type our last statement with an unequal number of spaces. This should raise an error when we execute it. Create another new line, type three spaces, and write the final statement as follows. Then, press the *Enter* key to execute the whole snippet:

   ```
   >>> if True:
   ...     print(1)
   ...    print(2)
   ```

   Executing this should raise the following indentation error:

   ```
     File "<stdin>", line 3
       print(2)
              ^
   IndentationError: unindent does not match any outer indentation level
   ```

4. This error tells us that the indentation for our last line, which was three spaces, didn't match our first line's indentation, which was four spaces. Indentation for statements in a block is dictated by the first statement's indentation level. To fix this, indent that line in the same way as the first, and it should run normally now:

   ```
   >>> if True:
   ...     print(1)
   ...     print(2)
   ...
   1
   2
   >>>
   ```

Note

While you can use any number of spaces to indent a statement, the general convention is to use four spaces.

Activity 5: Fixing Indentations in a Code Block

Fix the code snippet in the **`incorrect_indentation.py`** file from the code bundle to have the appropriate indentation in each block. Once you fix it, run the script in the terminal and verify that you get the output. Don't worry if you don't understand what the code is doing; the goal is to get comfortable with indentations. The incorrect code is also shown here:

```
>>> if 5 > 2:
...     print("Greater than")
...    x = 5
...   print(x * 2)
... else:
...     print("Less than")
...    print(2)
```

Here's a hint: Indentation for statements in a block is dictated by the first statement's indentation level.

The output of the correct syntax should be as follows:

```
Greater than
10
```

> **Note**
>
> Solution for this activity can be found at page 278.

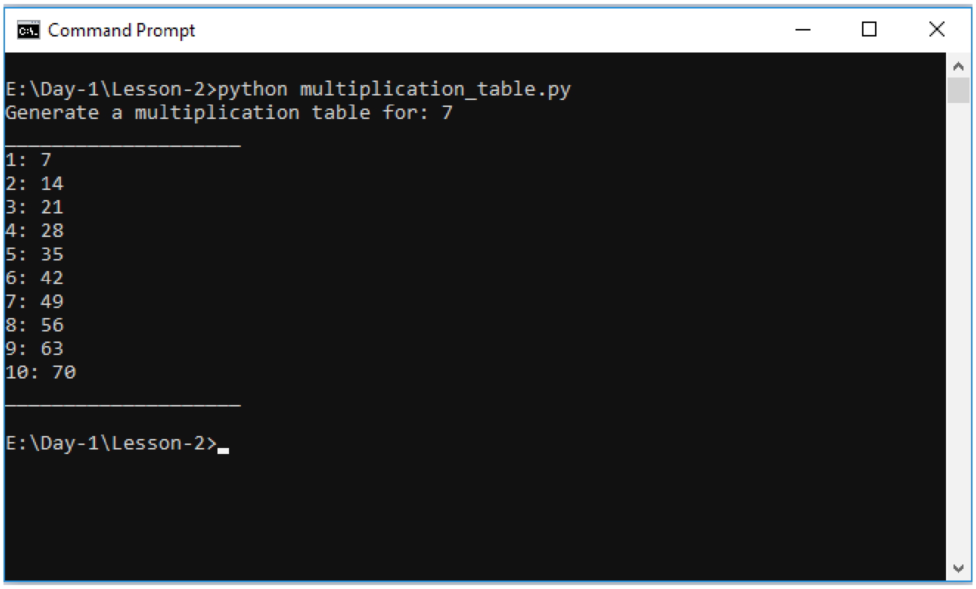
Activity 6: Implementing User Input and Comments in a Script

Write a script that takes a number from a user's input and prints out its multiplication table from 1 to 10.

The steps are as follows:

1. Open your editor.
2. Create a file named `multiplication_table.py` and save it.
3. On the first line, add a docstring explaining what our file does. Then, we'll assign a variable called `number` to the user's input and cast it to an integer.
4. Next, print 20 underscores as the top border of the table.
5. Multiply our number with each number from 1 to 10 and print that out.
6. Finally, print 20 underscores again for the bottom border of the table, as in step 4.
7. Save the file and run it by using the `python multiplication_table.py` command.

The output should look like this:



```
E:\Day-1\Lesson-2>python multiplication_table.py
Generate a multiplication table for: 7
____________________
1: 7
2: 14
3: 21
4: 28
5: 35
6: 42
7: 49
8: 56
9: 63
10: 70
____________________

E:\Day-1\Lesson-2>_
```

Figure 1.10: Output of running the multiplication_table.py script

> **Note**
>
> Solution for this activity can be found at page 278.

Summary

In this chapter, we have looked at two ways of running Python programs. We can run commands through the Python interactive shell or by running saved scripts. While you'll mostly find yourself running programs from saved scripts, you will find the freedom and quick gratification of using the interactive shell to run a quick check very convenient. These two methods will come in handy on your Python journey.

We have also covered the Python syntax in detail in this chapter. We started with variable assignment in Python. We looked at the different types of values Python variables can be assigned to, the syntax for assigning them, as well as the importance of reserved keywords in Python.

We then looked at the built-in **`input`** function and how it enables us to take input from a user keyboard. We looked at the different ways of writing comments in Python code, and then we finished the chapter by looking at the importance of indentation in writing readable, maintainable Python code.

In the next chapter, we will look at data types such as integers, strings, Booleans, and more.

2

Data Types

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the different numerical data types
- Use operators on numerical data types
- Explain strings and implement string operations, such as indexing, slicing, and string formatting
- Describe escape sequences
- Explain lists and perform simple operations on them
- Use Boolean expressions and Boolean operators

This lesson introduces the data types available to us. We look at integers, strings, lists, and Booleans.

Introduction

In the previous chapter, we learned about variables and looked at a few of the values/types of data that can be assigned to them. Specifically, we dealt with data of the string and integer types. In this chapter, we will look at the other data types that Python supports. Data types classify data, to tell the interpreter how the program intends to utilize that data. Data types define the different operations that can be performed on the data, how the data is stored, and the meaning of the data.

Numerical Data

Let's begin with numerical data types.

Types of Numbers

Integers

Integers, as we saw in the previous chapter, are numerical data types that are comprised of whole numbers. Whole numbers can be either negative or positive. In the following example, we will see how Python represents integers, and then, we can check their types:

```
>>> integer = 49

>>> negative_integer = -35

>>> print(type(integer), integer)

<class 'int'> 49

>>> print(type(negative_integer), negative_integer)

<class 'int'> -35

>>>
```

Additionally, Python integers have unlimited precision. This means that there are no limits to how large they can be (save for the amount of available memory):

```
>>> large_integer = 34567898327463893216847532149022563647754227885439016662
145553364327889985421

>>> print(large_integer)

34567898327463893216847532149022563647754227885439016662145553364327889985421
1

>>>
```

Floating Point Numbers

Another numerical type supported by Python is floating point numbers. The type for this kind of value is **`float`**. As we saw in the previous chapter, these are represented in the following way:

```
>>> n = 3.3333
>>> print(n)
3.3333
>>> import math
>>> print(type(math.pi), math.pi)
<class 'float'>, 3.141592653589793
>>> print(type(math.e), math.e)
<class 'float'>, 2.718281828459045
>>>
```

Here, we import **`math`**, which is a built-in Python library that provides access to different mathematical functions and constants. We print out the type and the values of the constants Pi (**`math.pi`**) and Euler's number (**`math.e`**). The displayed values are approximated.

Furthermore, you can convert an integer to a floating point value by using the **`float`** function:

```
>>> float(23)
23.0
>>>
```

Binary, Hexadecimal, and Octal Numbers

Binary, hexadecimal, and octal numbers are alternative number systems, as opposed to the common decimal number system that we're accustomed to. Binary numbers are numbers expressed in the *base* 2 system, which uses only 0s and 1s to represent numbers.

To write binary numbers in Python, write the number and prefix it with **`0b`**. Typing a binary number on the interactive shell outputs its decimal equivalent:

```
>>> 0b111

7

>>> 0b10

2

>>> 0b1000

8

>>>
```

Hexadecimal numbers are numbers that are expressed in the *base* 16 system. The symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, and f are used to represent hexadecimal numbers. Hexadecimal numbers should be prefixed with **`0x`**. Typing a hexadecimal number in the interpreter outputs its decimal equivalent:

```
>>> 0xf

15

>>> 0x9ac

2476

>>> 0xaf

175

>>>
```

Finally, we have octal numbers, which are numbers written in the *base* 8 numbering system. This system uses the digits from 0 to 7 to represent numbers. In Python, they should be prefixed with **`0o`**. Typing an octal number in the interpreter outputs its decimal equivalent:

```
>>> 0o20

16

>>> 0o200

128

>>> 0o113

75

>>>
```

To convert any decimal (*base* 10) number to binary, hexadecimal, or octal numbers, you can use the built-in **bin**, **hex**, and **oct** Python functions, respectively, as shown here:

```
>>> bin(7)
'0b111'
>>> hex(700)
'0x2bc'
>>> oct(70)
'0o106'
>>>
```

Exercise 11: Converting Between Different Types of Number Systems

In this exercise, we'll practice converting between different types of number systems. We will create a script that takes the user's input and converts it into a binary number.

The steps are as follows:

1. Create a new file named **binary_converter.py**.
2. On the first line, define the **number** variable that takes a user input:

   ```
   number = input("Convert to binary: ")
   ```

3. Convert the input to an integer:

   ```
   # convert number to integer
   integer = int(number)
   ```

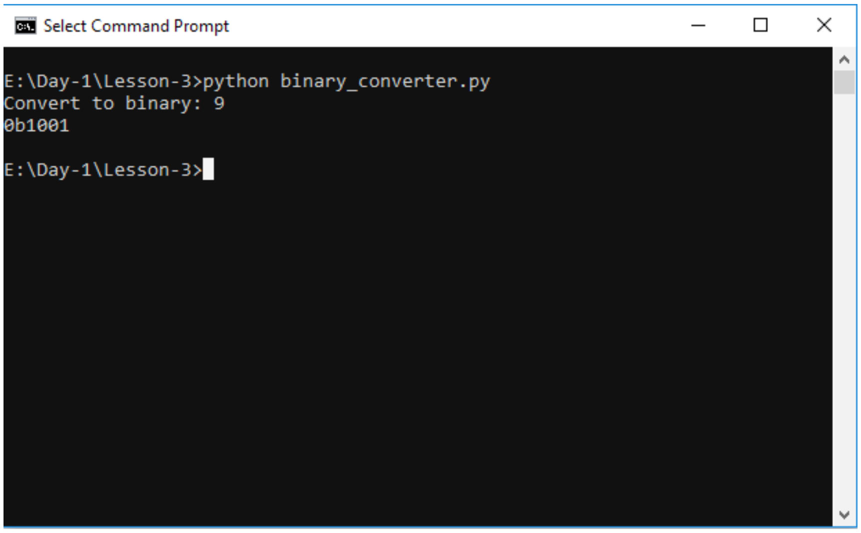
4. Then, convert it to a binary number:

   ```
   # convert integer to binary
   binary = bin(integer)
   ```

5. Finally, print out the value:

   ```
   print(binary)
   ```

6. We can then run the script with the **`python binary_converter.py`** command and pass in a value to be converted. It should look something like this:



```
E:\Day-1\Lesson-3>python binary_converter.py
Convert to binary: 9
0b1001

E:\Day-1\Lesson-3>
```

Figure 2.1: Output of running the binary_converter.py script

Besides these types, Python also has support for complex numbers, which comes in handy for scientific calculations that require them, such as Fourier transforms, among others. Python offers several different numeric types and provides a straightforward, intuitive process for converting between them.

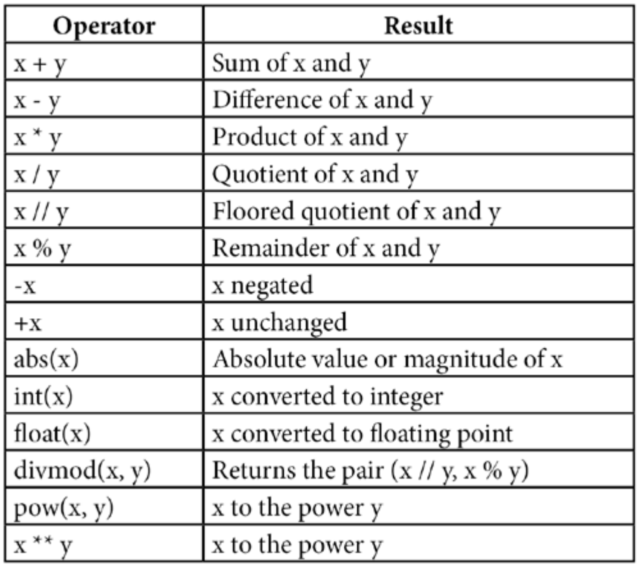
Operators

In this section, we will discuss the different operators that Python makes available to us.

Arithmetic Operators

Arithmetic operators are mathematical functions that take numerical values and perform calculations on them.

Numerical data types are only as valuable as the operations that you can carry out on them. All of the Python numeric types support the following operations:



Operator	Result
x + y	Sum of x and y
x - y	Difference of x and y
x * y	Product of x and y
x / y	Quotient of x and y
x // y	Floored quotient of x and y
x % y	Remainder of x and y
-x	x negated
+x	x unchanged
abs(x)	Absolute value or magnitude of x
int(x)	x converted to integer
float(x)	x converted to floating point
divmod(x, y)	Returns the pair (x // y, x % y)
pow(x, y)	x to the power y
x ** y	x to the power y

Figure 2.2: Arithmetic Operators

We will be demonstrating by using decimal numbers, but these operators can work on operands of any numeric type. As you've already seen, you can add numbers, as follows:

```
>>> 5 + 8 + 7
20
>>>
```

You can also carry out subtraction:

```
>>> 20 - 5
15
>>>
```

And, you can also perform multiplication:

```
>>> 4 * 3
12
>>>
```

Finally, you can perform division:

```
>>> 12 / 3
4.0
>>>
```

> **Note**
>
> The division of two numbers, regardless of their value types in Python, will always yield a floating point number.

Floor division is different from classic division, in that it always yields a whole integer. It is a division of two numbers, but the value yielded has any fractional parts discarded. Floor division is also referred to as integer division:

```
>>> 13 // 2  # classic division would yield 6.5
6
>>>
```

The modulo operation finds the remainder after the division of one number by another:

```
>>> 5 % 2
1
>>> 20 % 3
2
>>>
```

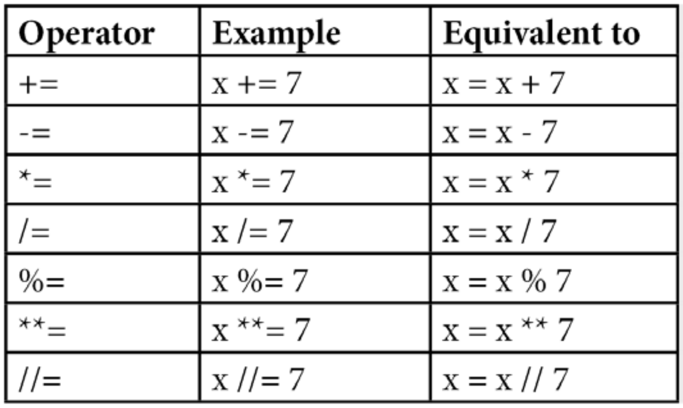
Finally, we have the exponentiation operation, which raises a number to a specified power:

```
>>> 5 ** 3
125
>>> 10 ** 4
10000
>>>
```

The difference between this method of exponentiation and using the `pow` function is that the `pow` function allows you to pass in a third argument, a divisor, which can be used to find the remainder after dividing the result (the exponentiated value) and the divisor.

Assignment Operators

Aside from the **=** simple assignment operator, Python has other assignment operators. These are shorthand variations of simple operators, in that they not only do an arithmetic operation but also reassign the variable. The following table lists all of the assignment operators:



Operator	Example	Equivalent to
+=	x += 7	x = x + 7
-=	x -= 7	x = x - 7
*=	x *= 7	x = x * 7
/=	x /= 7	x = x / 7
%=	x %= 7	x = x % 7
**=	x **= 7	x = x ** 7
//=	x //= 7	x = x // 7

Figure 2.3: Assignment Operators

The following is an example of these operators in action. **x** is initially assigned to **10**. We add **1** and then reassign **x** to the result of that operation, **11**:

```
>>> x = 10
>>> x += 1
>>> print(x)
11
>>>
```

The preceding code is equivalent to the following:

```
>>> x = 10
>>> x = x + 1
>>> print(x)
11
>>>
```

The same principle is applicable for all of the operators that are listed in the preceding table.

You can perform all arithmetic operations in Python. All operators can be applied to all numeric types. Python also provides assignment operators as a shorthand way of performing an operation and assignment in one statement.

Order of Operations

The **order of operations** is the collection of rules about which procedures should be evaluated first when evaluating an expression.

In Python, the order in which operators are evaluated is just as it is mathematically: **PEMDAS**.

Parentheses have the highest precedence. Expressions inside parenthesis are evaluated first:

```
>>> (9 + 2) * 2
22
>>>
```

Next, the exponentiation operator is given the second highest precedence:

```
>>> 2 ** 3 + 2
10
>>>
```

Multiplication and division (including floor division and the modulo operation) have the same precedence. Addition and subtraction come next:

```
>>> 8 * 3 + 1
25
>>> 24 / 6 - 2
2
>>> 7 + 5 - 3
9
```

In cases where two operators have the same precedence (for example, addition and subtraction), statements are evaluated left to right:

```
>>> 7 - 5 + 4
6
>>> 10 / 5 * 3
6.0
```

The exception to the preceding rule is with exponents, which are evaluated from the right-most value. In the following example, the evaluation is equivalent to 2^(3^2):

```
>>> 2**3**2
512
```

Let's go over what we have learned in this section before moving ahead:

- Python supports different numeric types: integers, floating point numbers, and binary, hexadecimal, and octal numbers, to name but a few.
- Python also supports complex numbers.
- Multiple operators can be applied to these different types of numbers.
- Assignment operators carry out the operation and reassign the variable to the result.
- Arithmetic operators in Python follow the standard order of operations in mathematics: PEMDAS.

Activity 7: Order of Operations

In this activity, we will try to get conversant with the order of arithmetic operators in Python. Rewrite the following equation as a Python expression and get the result of the equation:

$$5(4-2)+\left(\frac{100}{\frac{5}{2}}\right)2$$

Figure 2.4: Activity Equation

Python follows the mathematical rules that you're accustomed to. A lot of what you'd expect to work mathematically can be intuitively tried out in Python, and will often work.

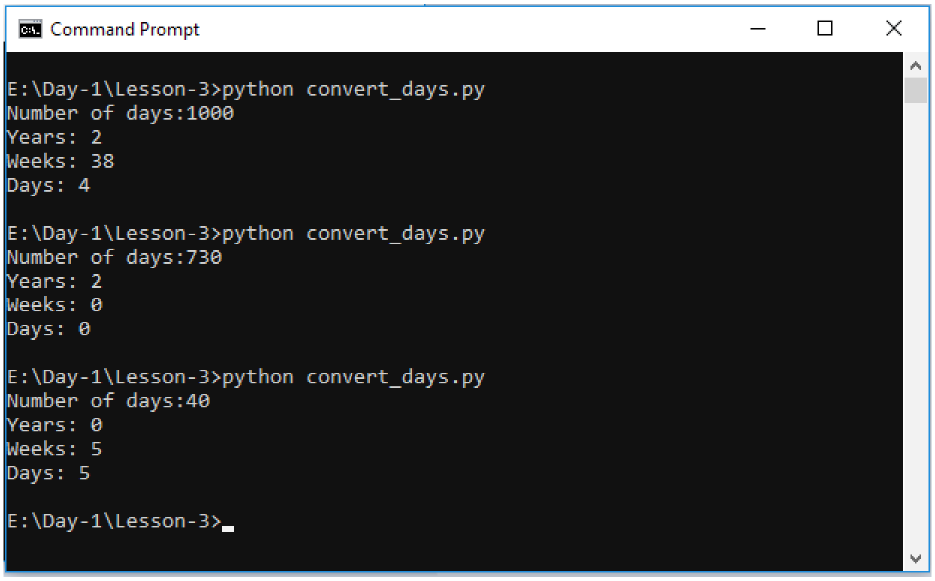
> **Note**
>
> Solution for this activity can be found at page 279.

Activity 8: Using Different Arithmetic Operators

Write a script that takes user input as days and converts the days into years, weeks, and days, and then prints them out. We can ignore leap years. The aim of this activity is to use different arithmetic operators to split days into years, weeks, and days.

The steps are as follows:

1. Create a file named **`convert_days.py`**.
2. On the first line, declare the user input. It's an integer, so cast the string.
3. Then calculate the number of years in that set of days.
4. Next, convert the remaining days that weren't converted to years into weeks.
5. Then, get any remaining days that weren't converted to weeks.
6. Finally, print everything out.
7. We can then save and run the script.
8. The output should look something like this:



```
E:\Day-1\Lesson-3>python convert_days.py
Number of days:1000
Years:  2
Weeks:  38
Days:  4

E:\Day-1\Lesson-3>python convert_days.py
Number of days:730
Years:  2
Weeks:  0
Days:  0

E:\Day-1\Lesson-3>python convert_days.py
Number of days:40
Years:  0
Weeks:  5
Days:  5

E:\Day-1\Lesson-3>
```

Figure 2.5: Output of running the convert_days.py script

> **Note**
>
> Solution for this activity can be found at page 279.

Strings

In this section, we will look at strings in detail.

String Operations and Methods

As we mentioned in the previous chapter, strings are a sequence of characters. The characters in a string can be enclosed in either single (`'`) or double (`"`) quotes. This does not make a difference. A string enclosed in single quotes is completely identical to one enclosed in double quotes:

```
>>> "a string"
'a string'
>>> 'foobar'
'foobar'
>>>
```

A double-quoted string can contain single quotes:

```
>>> "Susan's"
"Susan's"
>>>
```

A single-quoted string can also contain double quotes:

```
>>> '"Help!", he exclaimed.'
'"Help!", he exclaimed.'
>>>
```

You can also build a multiline string by enclosing the characters in triple quotes (`'''` or `"""`):

```
>>> s = """A multiline
string"""
>>> print(s)
A multiline
string
>>> s2 = '''Also a
```

```
multiline string
'''
>>> print(s2)
Also a
multiline string
>>>
```

Also, as you saw in the first chapter, you can use the `*` operator to repeat strings:

```
>>> print('Alibaba and the', 'thieves ' * 40)
Alibaba and the thieves thieves thieves thieves thieves thieves thieves
thieves thieves thieves thieves thieves thieves thieves thieves thieves
thieves thieves thieves thieves thieves thieves thieves thieves thieves
thieves thieves thieves thieves thieves thieves thieves thieves thieves
thieves thieves thieves thieves thieves thieves
>>>
```

And, you can use the `+` operator to concatenate strings:

```
>>> "I " + "love " + "Python"
'I love Python'
>>>
```

> **Note**
>
> Concatenation will join the two strings just as they are, and will not add spaces; thus, we add a space at the end for each string that forms a word in the preceding example.

Python strings are *immutable*. This means that once they are assigned to a variable, their value cannot be changed. Consider the following:

```
>>> string = "flip flop"
>>> string * 8  # a spider wearing slippers
'flip flop flip flop flip flop flip flop flip flop flip flop flip flop flip flop '
```

Once we print the original value, we will see that the string's original value remains unchanged:

```
>>> print(string)
flip flop
```

The same is applicable for all string operations; they do not change any part of the string:

```
>>> hello = "Hello "
>>> world = "World"
>>> hello + world
'Hello World'
>>> hello
'Hello '
```

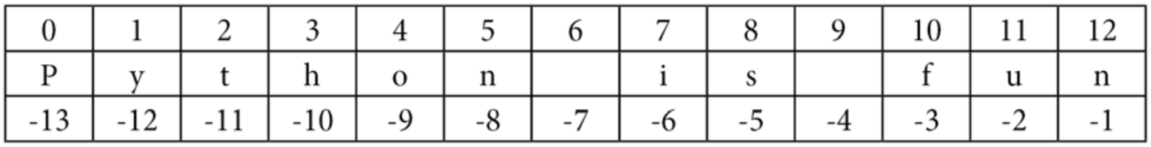
To *change* the preceding string, we'd have to reassign the variable to the new string:

```
>>> hello = hello + world
>>> hello
'Hello World'
```

Indexing

Python strings can be indexed. Like most languages, the first character in the sequence in the string is at the index 0.

Consider the string **`Python is fun`**. The following is a table showing the index of each character in the string. Characters in the string are indexed in two ways - left to right, which starts at 0, and right to left, which starts at -1:



0	1	2	3	4	5	6	7	8	9	10	11	12
P	y	t	h	o	n		i	s		f	u	n
-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Figure 2.6: String indices

To get a character from a string, you can use the standard **`[]`** syntax:

```
>>> s = "Python is fun"
>>> s[0]
'P'
>>> print(s[7], s[8])
i s
>>> s[-1]
'n'
>>> s[-13]
'P'
```

If we try to get a character from an index that doesn't exist, Python will raise an **`IndexError`**. Here, we are trying to get a character in an index that's larger than the size of the string itself:

```
>>> s = "foobar"
>>> s[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

Slicing

Additionally, you can access characters within a range of indices in a string and get a slice/substring of that string. Slicing syntax is in the following format: **`string[start_index : end_index]`**.

Note that the returned substring doesn't include the character at the end index, but instead, every character up to it:

```
>>> string = "championships"
>>> string[0:5]
'champ'
>>> string[5:9]
'ions'
>>> string[-5:-1]
'ship'
```

Python allows you to omit the start or end index when slicing a string:

```
>>> string = "foobar"
>>> string[3:]
'bar'
>>> string[:3]
'foo'
```

Here, we can see that when we pass just the start index while slicing, Python automatically slices the string up to the last index. If we pass only the end index, it slices every character, from the start of the string up to that end index.

Activity 9: String Slicing

Given the following statements, predict what the output will be:

1. Execute the following statement:

   ```
   >>> "Living in a silent film"[5]
   ```

2. Execute the following statement:

   ```
   >>> "[8, 9, 10]"[4]
   ```

3. Execute the following statement:

   ```
   >>> "Don't try to frighten us with your sorcerer's ways"[13:19]
   ```

4. Execute the following statement:

   ```
   >>> "Testing 1, 2, 3"[7:]
   ```

5. Execute the following statement:

   ```
   >>> "A man, a plan, a canal: Panama."[:-1]
   ```

> **Note**
>
> Solution for this activity can be found at page 280.

Length

The length of a string is determined by the number of characters there are inside of it. In Python, you can get the length of a string by using the built-in **`len()`** function, which takes a string as its parameter and returns an integer. In the following example, we can see that the length of the string is 44 characters:

```
>>> question = "Who was the first Beatle to leave the group?"
>>> len(question)
44
>>>
```

An empty string would have a length of 0:

```
>>> empty = ""
>>> len(empty)
0
>>>
```

String Formatting

String formatting is important when you want to build new strings that are using existing values. Python provides several ways to format text strings. The most popular of these are string interpolation, the **`str.format()`** method, and % formatting.

String Interpolation

In Python 3.6, support for string interpolation was added. String interpolation is the process of evaluating a string that has placeholders. These placeholders can hold expressions that yield a value, which is then placed inside the string. Special kinds of strings, known as **f-strings** (**formatted strings**), are used during string interpolation. These strings are prefixed with an **`f`** to denote how they're meant to be interpreted:

```
>>> pie = 3.14
>>> f"I ate some {pie} and it was yummy!"
'I ate some 3.14 and it was yummy!'
>>>
```

> **Note**
>
> If you omit the **`f`** prefix, the string will be interpreted literally, as-is.

To insert the variable, we need to place curly braces that contain the expression we want to put inside the string. This can be any valid Python expression:

```
>>> number = 7
>>> f"{number+1} is just a number."
'8 is just a number.'
>>>
```

Python string interpolation provides powerful, declarative, and more intuitive formatting of your strings, compared to the other methods that Python offers. This should be the de facto way to format strings when using Python 3.6+.

> **Note**
>
> To read more on string interpolation, visit https://www.python.org/dev/peps/pep-0498/.

The str.format() Method

The **`format()`** method can be found on every string instance. It allows you to insert different values in positions within the string. This method works similarly to interpolation, save for the fact that you can't put expressions into the placeholders, and you have to pass in the values for insertion in the method call. The syntax for this is as follows:

```
>>> fruit = "bananas"
>>> "I love {}".format(fruit)
'I love bananas'
>>>
```

In the preceding string, we put curly braces in the positions where we want to put our values. When we call the **`format`** method, it takes that first argument (our variable, **`fruit`**), and replaces the curly braces with its value. You can also pass multiple values in.

The values can be any kind of object:

```
>>> age = 40
>>> years = 10
>>> string = "In {} years, I'll be {}"
>>> string.format(years, age)
"In 10 years I'll be 40"
>>>
```

If the Python version you're using doesn't support string interpolation, this should be the method that you use.

% Formatting

An old, deprecated way of formatting strings, which you might end up seeing in old code, is the C language style % formatting. In this method, you use the % operator to pass in values. Inside the string, you use the **`%`** character, followed by a format specifier, to declare how the value should be inserted; for example, **`%s`** for string, or **`%d`** for integers:

```
>>> number = 3
>>> pets = "cats"
>>> "They have %d %s" % (number, pets)
'They have 3 cats'
```

This method is inflexible and is harder to use correctly, and thus, it should generally be avoided.

String Methods

Aside from the **`format`** method, string instances have a couple of useful methods that can be used to transform and inspect strings. We will demonstrate a few of the common ones. You can read through the Python documentation for more information on string methods.

str.capitalize()

The **`str.capitalize()`** method returns a copy of the string with the first letter capitalized and the rest in lowercase:

```
>>> "HELLO".capitalize()
'Hello'
>>> "hello".capitalize()
'Hello'
```

str.lower()

The **`str.lower()`** method returns a copy of the string with all characters converted to lowercase:

```
>>> "WORLD".lower()
'world'
>>> "wOrLd".lower()
'world'
```

str.upper()

The **`str.upper()`** method returns a copy of the string with all characters converted to uppercase:

```
>>> "abcd".upper()
'ABCD'
>>> "EfGhi".upper()
'EFGHI'
>>>
```

str.startswith()

The **`str.startswith()`** method checks whether a string starts with the specified prefix. The prefix can contain one or more characters, and is case-sensitive. The method returns a Boolean, **`True`** or **`False`**:

```
>>> "Python".startswith("Py")
True
>>>
```

str.endswith()

The **`str.endswith()`** method is just like the **`startswith`** method, but it checks that the string ends with the specified suffix:

```
>>> "Python".endswith("on")
True
>>>
```

str.strip()

The **`str.strip()`** method returns a copy of the string with the leading and trailing characters removed. The method also takes an argument that is a string, specifying the set of characters to be removed. This method is also case-sensitive. If no arguments are passed to it, it removes all of the trailing and leading whitespaces.

This can be useful when sanitizing data:

```
>>> "Championship".strip("ship")
'Champion'
>>> "repair".strip("r")
'epai'
>>> "   John Doe  ".strip()
'John Doe'
>>>
```

str.replace()

The **`str.replace()`** method takes two substrings as arguments (old and new), then returns a copy of the string with all of the occurrences of the old substring replaced with the new one. Note that the method is case-sensitive:

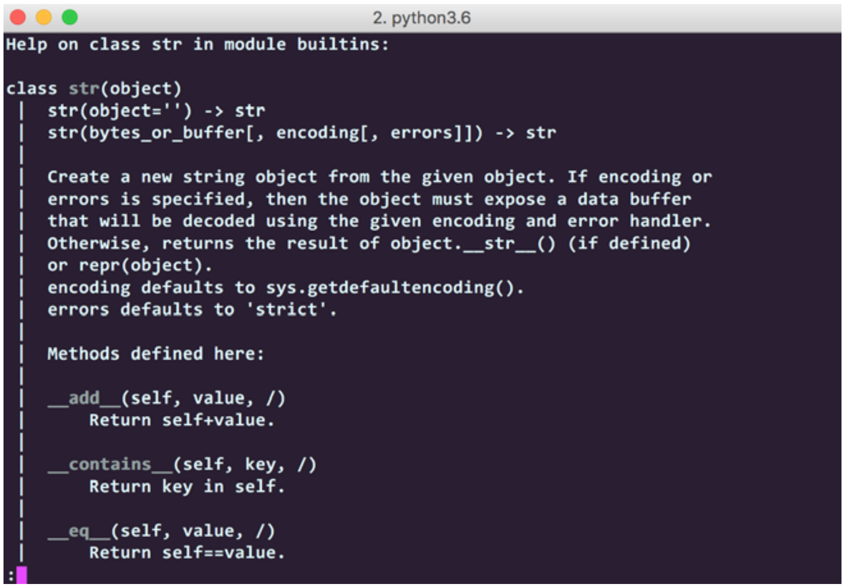
```
>>> "Cashewnuts".replace("Cashew", "Coco")
'Coconuts'
>>> "Emacs".replace("Emacs", "Vim")
"Vim"
>>>
```

> **Note**
>
> There are a lot more string methods available for use. You can read through the Python documentation for more information on string methods at https://docs.python.org/3/library/stdtypes.html#str.

You don't have to remember all of the string methods off the top of your head, as you can always refer to the documentation to see what methods strings support. To do this in the interpreter, run the following command:

```
>>> help(str)
```

You should see the following output, which you can browse through:



```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
:
```

Figure 2.7: Output of help(str)

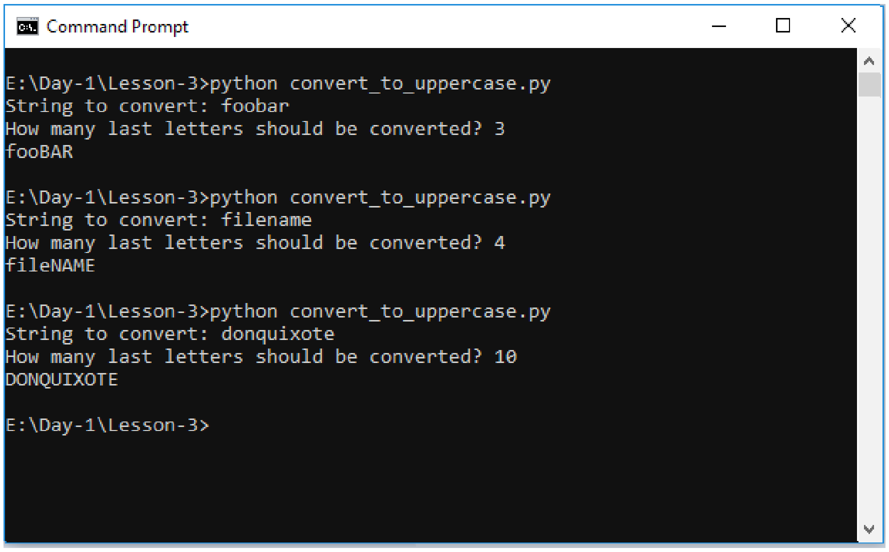
Activity 10: Working with Strings

Write a script that converts the last *n* letters of a given string to uppercase. The script should take the string to convert and an integer, specifying the last *n* letters to convert as input from the user. You can assume that *n* will be a positive number.

The steps are as follows:

1. Create a file named **`convert_to_uppercase.py`**.
2. On the first line, request the string to convert from the user.
3. On the next line, request for the number of last letters to convert.
4. Next, get the first part of the string.
5. Then, get the last part of the string, that is, the one we'll be converting.
6. Then, concatenate the first and last part back together, with the last substring transformed.
7. Finally, run the script with the **`python convert_to_uppercase.py`** command.

The output should look something like this:



```
E:\Day-1\Lesson-3>python convert_to_uppercase.py
String to convert: foobar
How many last letters should be converted? 3
fooBAR

E:\Day-1\Lesson-3>python convert_to_uppercase.py
String to convert: filename
How many last letters should be converted? 4
fileNAME

E:\Day-1\Lesson-3>python convert_to_uppercase.py
String to convert: donquixote
How many last letters should be converted? 10
DONQUIXOTE

E:\Day-1\Lesson-3>
```

Figure 2.8: Output of running the convert_to_uppercase.py script

> **Note**
>
> Solution for this activity can be found at page 280.

Reviewing what we have learned about strings, we should remember that the characters in a string are indexed, and you can access characters in each index. Python string indices start at 0. You can also access characters within a range of indices by slicing the string. Of the different string formatting methods that Python allows for, you should use string interpolation when using Python versions 3.6+, and the **`str.format()`** method otherwise.

Escape Sequences

An escape sequence is a sequence of characters that does not represent its literal meaning when inside of a string. An escape character tells the interpreter/compiler to interpret the next character(s) in a special way and ignore its usual meaning, thus creating an escape sequence.

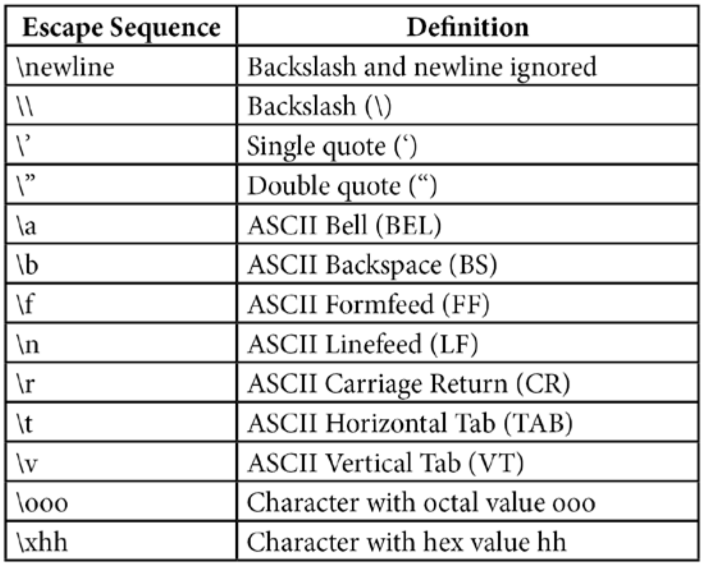
In Python, the escape character is the backslash **(\)**. For example, adding **\n** inside a string will tell the interpreter to interpret a new line inside the string, instead of the literal letter **n**:

```
>>> print("Hello\nWorld")
Hello
World
>>>
```

You can escape quotes inside a string, so that they are not interpreted as closing quotes:

```
>>> 'Weekend at Bernie's'
  File "<stdin>", line 1
    'Weekend at Bernie's'
                       ^
SyntaxError: invalid syntax
>>> 'Weekend at Bernie\'s'
"Weekend at Bernie's"
>>>
```

Here is the full list of valid escape sequences in Python:



Escape Sequence	Definition
\newline	Backslash and newline ignored
\\	Backslash (\)
\’	Single quote (‘)
\”	Double quote (“)
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

Figure 2.9: Escape sequences

Exercise 12: Using Escape Sequences

In this exercise, we will write a Python script that breaks down all the words of a sentence and prints each word on its own line, and will ring a bell when it's done. The goal of this exercise is to get familiar with escape sequences:

1. Create a file named **`split_to_lines.py`**.
2. On the first line, request the sentences to split from the user:

   ```
   sentence = input("Sentence: ")
   ```

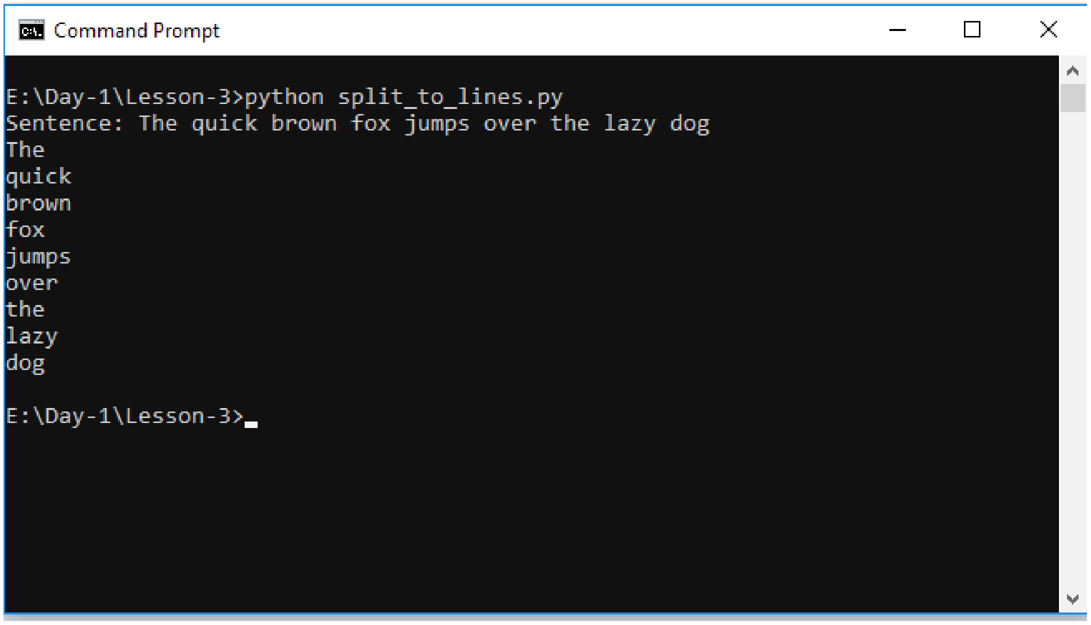
3. Next, replace all of the spaces in the sentence with newline characters:

   ```
   sentence = sentence.replace(" ", "\n")
   ```

4. Finally, print out the sentence and ring the terminal bell when you're done. Each word should appear on a new line:

   ```
   print(sentence, "\a")
   ```

5. You can run the script by using the **`python split_to_lines.py`** command. The output should look like this:



Command Prompt

```
E:\Day-1\Lesson-3>python split_to_lines.py
Sentence: The quick brown fox jumps over the lazy dog
The
quick
brown
fox
jumps
over
the
lazy
dog

E:\Day-1\Lesson-3>_
```

Figure 2.10: Output of running the split_to_lines.py script

Escape sequences in strings tell the program to perform a function or command, such as inserting a new line, ignore quotes occurring in strings, prompting the terminal to emit an audible signal, and several other functions.

Let's review what we have learned about strings:

- Strings are a sequence of characters.
- Strings can be enclosed in either single quotes (`'`) or double quotes (`"`).
- Multiline strings can be enclosed in either triple single quotes (`'''`) or triple double quotes (`"""`).
- Strings are immutable
- Characters in a string are indexed, and you can access each character by index.
- The first element in a string is at the index **`0`**.
- Substrings in a string can be accessed by slicing.
- Formatting strings allows you to insert values into a string.
- Strings come with several handy built-in methods for transforming or inspecting the string.
- Escape sequences in strings tell the program to perform a function or command.

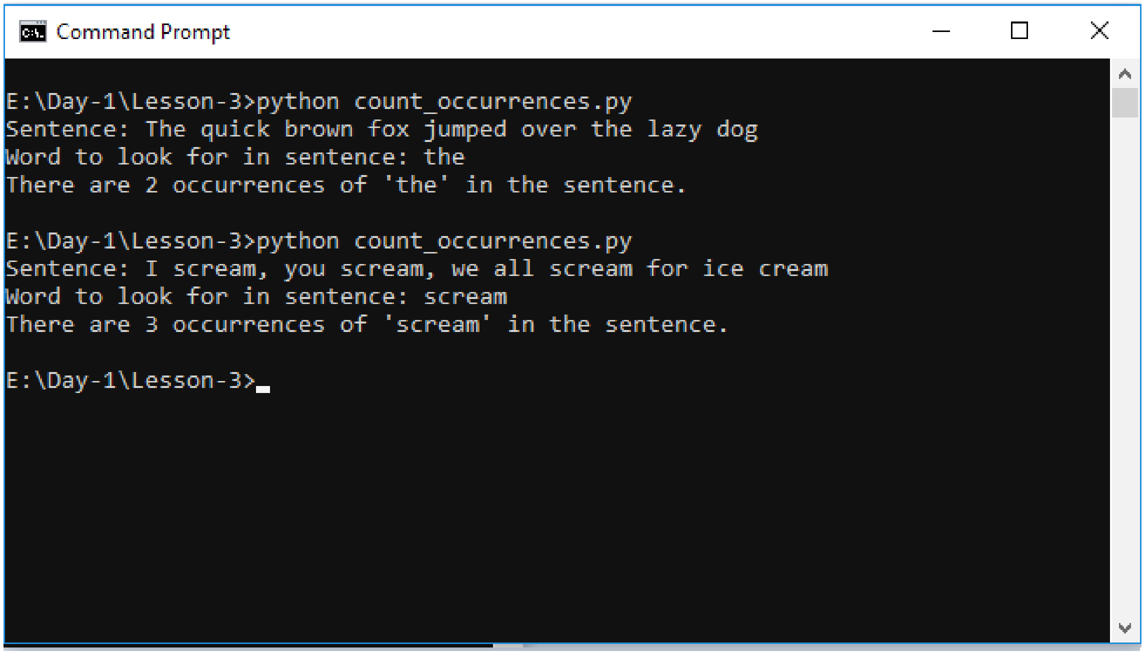
Activity 11: Manipulating Strings

Write a script that counts and displays the number of occurrences of a specified word in a given excerpt. The script should request two input values from the user, that is, the excerpt and the word to search for. You can assume that the word will not occur as a substring in other words.

The steps are as follows:

1. Create a file named **`count_occurrences.py`**.
2. Take in the user input for the sentence and the query.
3. Next, sanitize and format the input by removing the whitespace and converting it to lowercase.
4. Count the occurrences of the substring.
5. Print the results.
6. Run the script by using the **`python count_occurrences.py`** command.

The output should look like this:



```
E:\Day-1\Lesson-3>python count_occurrences.py
Sentence: The quick brown fox jumped over the lazy dog
Word to look for in sentence: the
There are 2 occurrences of 'the' in the sentence.

E:\Day-1\Lesson-3>python count_occurrences.py
Sentence: I scream, you scream, we all scream for ice cream
Word to look for in sentence: scream
There are 3 occurrences of 'scream' in the sentence.

E:\Day-1\Lesson-3>_
```

Figure 2.11: Output of running the count_occurrences.py script

Note

Solution for this activity can be found at page 281.

Lists

This is part one of two regarding lists, which we will be going through in this book. This part will act as an introduction, and will not cover the various methods that list objects have, such as **`extend()`**, **`remove()`**, **`pop()`**, and several others. We will go through the second section on lists in a later chapter.

List Operations

In Python, arrays (or the closest abstraction of them) are known as **lists**. Lists are an aggregate data type, meaning that they are composed of other data types. Lists are similar to strings, in that the values inside them are indexed, and they have a length property and a count of the objects inside of them. In Python, lists are heterogeneous, in that they can hold values of different types. In contrast to how arrays are in most languages, Python lists are also mutable, meaning that you can change the values inside of them, adding and removing items on the go.

Lists can be likened to a wardrobe. Wardrobes can hold multiple items of clothing, clothes of different kinds, and even shoes. Wardrobes provide a convenient storage space for the easy retrieval of your clothes, and make it so that you don't have to look for them all around the house. If we didn't have a list, we'd have to keep track of dozens of separate variables. Like wardrobes, lists provide a convenient collection of related objects.

Lists are made with is comma-separated elements enclosed in square brackets; for example:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> digits
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> letters = ["a", "b", "c", "d"]
>>> letters
['a', 'b', 'c', 'd']
>>> mixed_list = [1, 3.14159, "Spring", "Summer", [1, 2, 3, 4]]
>>> mixed_list
[1, 3.14159, 'Spring', 'Summer', [1, 2, 3, 4]]
>>>
```

As you can see here, lists can also contain other lists within them. You can also get the number of elements in a list by using the **`len()`** function:

```
>>> len(["a", "b", "c", "d"])
4
>>>
```

Indexing

Like strings, lists can also be indexed. The first element in a list starts at the index 0:

```
>>> fruits = ["apples", "bananas", "strawberries", "mangoes", "pears"]
>>> fruits[3]
'mangoes'
```

Negative indices can be used, as well:

```
>>> fruits[-1]
'pears'
>>>
```

Slicing

Lists can also be sliced. The slicing operation always returns a new list that's been derived from the old list. The syntax remains as **`list[start_index : end_index ]`**. As with string slicing, the element at the end index isn't included in the result:

```
>>> my_list = [10, 20, 30, 40, 50, 60, 70]
>>> my_list[4:5]
[50]
>>> my_list[5:]
[60, 70]
>>>
```

Omitting the end index and providing only the start index will slice everything from the start to the end of the list, while omitting the start index and giving only the end index will slice everything from the start index to the end index:

```
>>> my_list = [10, 20, 30, 40, 50, 60, 70]
>>> my_list[5:]
[60, 70]
>>> my_list[:4]
[10, 20, 30, 40]
>>>
```

Concatenation

Additionally, you can add two lists together by using the + operator. The elements of all of the lists being concatenated are brought together inside one list:

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> ["a", "b", "c"] + [1, 2.0, 3]
['a', 'b', 'c', 1, 2.0, 3]
>>>
```

Changing Values in a List

Since lists are mutable, you can change the value in a list by assigning whatever is at that index:

```
>>> names = ["Eva", "Keziah", "John", "Diana"]
>>> names[2] = "Jean"
>>> names
['Eva', Keziah, 'Jean', 'Diana']
>>>
```

Note that it's possible to add any type of value to a list, regardless of what types of values it contains. For example, you can add an integer to a list of strings or a string to a list of integers, and so on.

You can also use the **`list.append()`** method to insert a value at the end of a list:

```
>>> planets = ["Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn",
"Uranus", "Neptune"]
>>> planets.append("Planet X")
>>> planets
['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus',
'Neptune', 'Planet X']
```

Finally, you can assign slices of a list. This replaces the target slice with whatever you assign, regardless of the initial size of the list:

```
>>> alphanumeric_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> alphanumeric_list[4:7]
[5, 6, 7]
>>> alphanumeric_list[4:7] = ["a", "b", "c"]
>>> alphanumeric_list
[1, 2, 3, 4, 'a', 'b', 'c', 8, 9, 0]
```

An important thing to note is that when you assign a list, it points it to an object in the memory. If you assign another variable to the variable that references that list, the new variable also references that same list object in the memory. Any changes made using either reference will always change the same list object in the memory.

Exercise 13: List References

In this exercise, we will see how list references work:

1. Create a new list, as follows:

   ```
   >>> list_1 = [1, 2, 3]
   ```

2. Then, assign a new variable, **`list_2`**, to **`list_1`**:

   ```
   >>> list_2 = list_1
   ```

3. Any changes that we make to **`list_2`** will be applied to **`list_1`**. Append **`4`** to **`list_2`** and check the contents of **`list_1`**:

   ```
   >>> list_2.append(4)
   >>> list_1
   [1, 2, 3, 4]
   ```

4. Any changes that we make to **`list_1`** will be applied to **`list_2`**. Insert the value **`a`** at index **`0`** of **`list_1`**, and check the contents of **`list_2`**:

   ```
   >>> list_1[0] = "a"
   >>> list_2
   ['a', 2, 3, 4]
   >>>
   ```

This is because both variables reference the same object.

> **Note**
>
> We will be covering lists in further depth in *Chapter 5, Lists and Tuples*.

We've seen that lists are collections of values. Python lists are mutable. There are multiple operations that you can carry out on lists, such as accessing elements by index, slicing elements, getting the count of elements inside a list, concatenation, and changing values, either by index, appending, or replacing slices of the list.

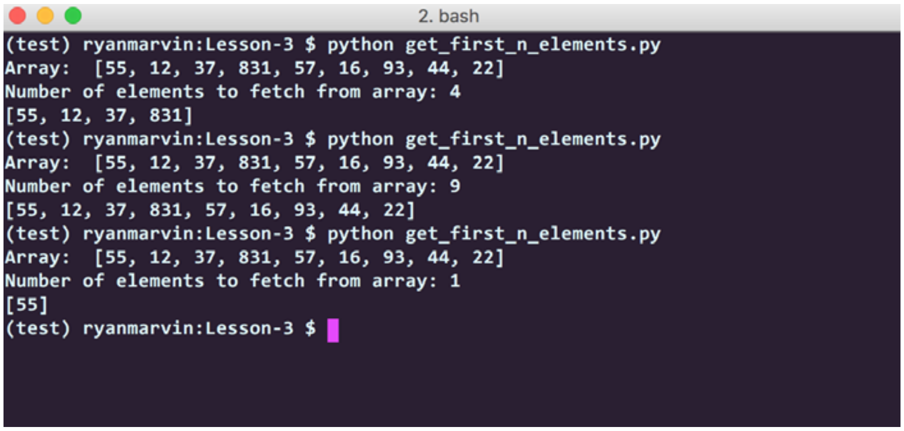
Activity 12: Working with Lists

Write a program that fetches the first *n* elements of a list.

The steps are as follows:

1. Create a script named **`get_first_n_elements.py`**.
2. On the first line, create the array.
3. Next, print the array out and fetch the user input for the number of elements to fetch from the array.
4. Finally, print out the slice of the array from the first element to the *nth* element.
5. Then, run the script by using the **`python get_first_n_elements.py`** command.

The output should look like this:



```
(test) ryanmarvin:Lesson-3 $ python get_first_n_elements.py
Array:  [55, 12, 37, 831, 57, 16, 93, 44, 22]
Number of elements to fetch from array: 4
[55, 12, 37, 831]
(test) ryanmarvin:Lesson-3 $ python get_first_n_elements.py
Array:  [55, 12, 37, 831, 57, 16, 93, 44, 22]
Number of elements to fetch from array: 9
[55, 12, 37, 831, 57, 16, 93, 44, 22]
(test) ryanmarvin:Lesson-3 $ python get_first_n_elements.py
Array:  [55, 12, 37, 831, 57, 16, 93, 44, 22]
Number of elements to fetch from array: 1
[55]
(test) ryanmarvin:Lesson-3 $
```

Figure 2.12: Output of running the get_first_n_elements.py script

> **Note**
>
> Solution for this activity can be found at page 282.

Booleans

Boolean data types are values that can only be one of two values, **`True`** or **`False`**. For example, the proposition *100 is more than* 5 is *True*, and thus, it would have a **`True`** Boolean value. On the other hand, the proposition *The sky is green* is *False*, and thus, it would have a **`False`** Boolean value.

Booleans are largely associated with control statements, as they change the flow of the program, depending on the truthfulness of the specified quantities.

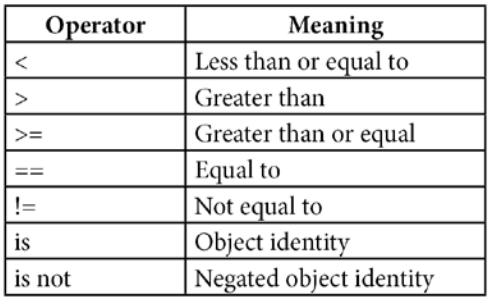
In Python, **`True`** and **`False`** are used to represent the two Boolean constants:

```
>>> True
True
>>> False
False
>>> print(type(True), type(False))
<class 'bool'> <class 'bool'>
```

We can see that the type of each expression is **bool** (short for Boolean). Like all other types, Booleans have operators that you can apply.

Comparison Operators

Comparison operators compare the values of objects or the objects, identities themselves. The objects don't need to be of the same type. There are eight comparison operators in Python:



Operator	Meaning
<	Less than or equal to
>	Greater than
>=	Greater than or equal
==	Equal to
!=	Not equal to
is	Object identity
is not	Negated object identity

Figure 2.13: Comparison operators

The following are some example uses:

```
>>> 10 < 1
False
>>> len("open") <= 4
True
>>> 10 > 1
True
>>> len(["banana"]) >= 0
True
>>> "Foobar" == "Foobar"
True
>>> "Foobar" != "Foobar"
False
>>>
```

Now, consider the following code:

```
>>> l =  [1, 2, 3]
>>> l2 = l
>>> l is l2
True
>>> l is not None
True
>>>
```

Here, we create a list, **`l`**, and then assign the variable **`l2`** to that same list. This creates a reference for the list. Thus, the statement **`l`** is **`l2`** is **`True`**, since both variables reference the same object in the memory.

The statement **`l is not None`** evaluates to **`True`**, as well, since **`l`** points to something in the memory, and therefore, it isn't null. **`None`** is the Python equivalent of null.

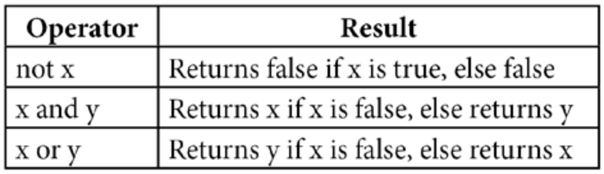
Logical Operators

We use logic in everyday life. Consider the following statements:

- *I'll have juice* OR *water if there isn't any juice.*
- *The knife has to be sharpened* AND *polished for the chef to use it.*
- *I am* NOT *tired; therefore, I will stay awake.*

Each of these statements has a condition. The condition for having water is if there isn't any juice available. The chef will only use the knife if two conditions are met, that is, that it is sharpened and that it is polished. The condition for staying awake is only if the condition of being tired has not been met.

In the same way, we have logical operators that combine Boolean expressions in Python: **`not`**, **`and`**, and **`or`**, as described in the following table:



Operator	Result
not x	Returns false if x is true, else false
x and y	Returns x if x is false, else returns y
x or y	Returns y if x is false, else returns x

Figure 2.14: Logical operators

and is a short-circuit operator, in that it only evaluates the second argument if the first one is **True**. **or** is also a short-circuit operator, in that it will only evaluate the second argument if the first one is **False**.

The following is an example of **and**:

```
>>> fruits = ["banana", "mangoes", "apples"]
>>> wants_fruits = True
>>> len(fruits) > 0 and wants_fruits
True
>>>
```

The following code shows **or** in action:

```
>>> value_1 = 5
>>> value_2 = 0
>>> value_1 > 0 or value_2 > 0
True
>>>
```

Finally, the following is an example of **not**:

```
>>> not True
False
>>> not False
True
>>>
```

Membership Operators

The operators **in** and **not in** test for membership. All sequences (for example, lists and strings), support this operator. For lists, these operators go through each element to see whether the element being searched for is within the list. For strings, the operators check whether the substring can be found within the string. The return values for these operators are **True** or **False**.

Let's see how they work:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> 3 in numbers
True
>>> 100 in numbers
False
>>> sentence =  "I like beef, mutton and pork"
>>> "chicken" not in sentence
True
>>> "beef" not in sentence
False
>>>
```

Boolean data types are values that can be either **`True`** or **`False`**. You can compare the values of two objects by using comparison operators, and you can combine or alter Boolean expressions by using logical operators. Membership operators are used to assert whether an element can be found in a sequence or container type object.

Activity 13: Using Boolean Operators

Insert the appropriate Boolean operators in the following condition statements, so that the code in the block executes. This will help us practice using Boolean operators:

1. Consider the following code block:

   ```
   n = 124
   if n % 2 ? 0:
       print("Even")
   ```

2. Consider the following code block:

   ```
   age = 25
   if age ? 18:
       print("Here is your legal pass.")
   ```

3. Consider the following code block:

   ```
   letter = "b"
   if letter ? ["a", "e", "i", "o", "u"]:
     print(f"'{letter}' is not a vowel.")
   ```

> **Note**
>
> Solution for this activity can be found at page 282.

Summary

In this chapter, we took an in-depth look at the basic data types that Python supports. We started with numerical data types and their related operators. We then covered strings and looked at string indexing, slicing, and formatting. Then, we moved on and took a brief look at lists (also known as arrays) and Booleans, as well as Boolean operators.

In the next chapter, we will begin our journey into learning how to control the flow of our programs by using control statements and loops.

3

Control Statements

Learning Objectives

By the end of this chapter, you will be able to:

- Describe the different control statements in Python
- Control program execution flow using control statements such as **`if`** and **`while`**
- Use looping structures in your Python programs
- Implement branching within looping structures such as **`for`** and **`range`**
- Implement breaking out of loops

This lesson describes the Python program flow and how we can change the flow of execution using control statements such as if, while, for, and range.

Introduction

Previously in this book, we covered the following topics:

- The Python interpreter
- Python syntax
- Values and data types

In this chapter, we are going to build on the knowledge that we have acquired so far to dive deeper into the beautiful language that is Python. In this chapter, we will explore how Python handles control statements–in simple terms, how Python handles decision making, for instance, resulting to *True* if 2 + 3 = 5.

In this chapter, we will also dive deeper into program flow control. In particular, we will look at how we can run code repeatedly or in a loop.

Specifically, we will cover the following topics:

- Python program flow
- Python control statements, that is, `if` and `while`
- The differences between `if` and `while`
- The `for` loop
- The `range` function
- Nesting loops
- Breaking out of loops

Control Statements

Like most programming languages, Python supports a number of ways to control the execution of a program by using control statements. Some of them might already be familiar, while others are unique to Python either in terms of syntax or execution.

In this chapter, we will delve into controlling program flow and start working on building more structured programs. This should also prepare us for learning various kinds of loops later in this chapter. To start us off, we are going to define some terms:

- Program flow
- Control statement

Program Flow

Program flow describes the way in which statements in code are executed. This also includes the priority given to different elements.

Python uses a simple top-down program flow. This is to say that code is executed in sequence from the top of the file to the very bottom. Each line has to wait until all of the lines that come before it have completed execution before their own execution can begin.

This top-down program flow makes it easy to understand and debug Python programs, as you can visually step through the code and see where things are failing.

In a top-down scenario, a problem is broken down into simple modules, each responsible for a part of the solution, closely related to one another. For instance, consider a salary calculation. There would be a module responsible for each of the following:

- Tax computation
- Debt computation (if needed)
- Net amount computation

Control Statement

Having defined the program flow, we can now understand what a control statement is.

A control statement is a structure in code that conditionally changes the program flow. A control statement achieves this by conditionally executing different parts of code. A control statement can also be used to repeatedly and conditionally execute some code.

You can think of a control statement as a traffic police officer at a junction who only lets traffic through if the exit is clear. Checking whether the exit is clear would be the condition, in this case. The officer will only let cars through the junction when the exit is clear.

The two main control statements in Python are:

- **`if`**
- **`while`**

The if Statement

An **if** statement allows you to execute a block of code if a condition is true. Otherwise, it can run an alternative block of code in its **else** clause.

The **else** clause of an **if** statement is optional.

You can chain multiple **if** statements that check for multiple conditions one after the other and execute a different block of code when the various conditions are true.

The basic syntax of an if statement is shown here:

```
if condition:

  # Run this code if the condition evaluates to True

else:

  # Run this code if the condition evaluates to False
```

As you can see, the **if** statement allows you to branch the execution of code based on a condition. If the condition evaluates to true, we execute the code in the **if** block. If the condition evaluates to false, we execute the code in the **else** block.

Exercise 14: Using the if Statement

In this exercise, we will see a practical application of the **if** statement:

1. First, declare a variable of type string called **release_year** and assign the value **1991** to it:

   ```
   release_year = '1991'
   ```

2. Then, declare another variable called **answer** and assign it to an **input** function call:

   ```
   answer = input('When was Python first released?')
   ```

3. Next, use an **if** statement to check whether the answer entered by the user is correct. If the answer is correct, we print out the success message **Congratulations! That is correct.**:

   ```
   if answer == release_year:
     print('Congratulations! That is correct.')
   ```

4. Then, use an **elif** statement to check whether the answer entered by the user is greater than the correct answer. **elif** is a combination of **else** and **if**, and enables a broader comparison scope through chaining multiple **if** statements.

If the answer is greater than the correct answer, we tell the user that the guess was too high:

```
elif answer > release_year:
  print('No, that\'s too late')
```

5. Next, use an **`elif`** to check whether the answer entered by the user is less than the correct answer.

   If the answer is less than the correct answer, we tell the user that the guess was too low:

```
elif answer < release_year:
  print('No, that\'s too early')
```

6. Finally, print the exit message:

```
print('Bye!')
```

The final code is as follows:

```
# Set release_year to 1991
release_year = '1991'
# Prompt the user to enter their answer to the question
answer = input('When was Python first released?')

if answer == release_year:
  # If the answer is correct, show the success message
  print('Congratulations! That is correct.')
elif answer > release_year:
  # If the answer is greater that release_year, tell the user the guess
was too high
  print('No, that\'s too late')
elif answer < release_year:
  # If the answer is less that release_year, tell the user the guess was
too low
  print('No, that\'s too early')

# Finally, print the exit message
print('Bye!')
```

Now let's look at some sample output from running this program and giving it various responses:

7. First, we'll look at what happens when you give the program an answer that is less than the expected answer. Run the script in the terminal and on being prompted, enter an incorrect value, such as **1969**:

   ```
   Python 3.6.1 (default, Dec 2015, 13:05:11)
   [GCC 4.8.2] on linux

   When was Python first released? 1969
   No, that's too early
   Bye!
   ```

8. The following output shows what happens when you provide the correct answer to the program. Run the script again and enter **1991** as the input:

   ```
   Python 3.6.1 (default, Dec 2015, 13:05:11)
   [GCC 4.8.2] on linux

   When was Python first released? 1991
   Congratulations! That is correct.
   Bye!
   ```

An **if** statement is used when you want to conditionally execute different blocks of code. The **if** statement is especially useful when there are multiple different blocks of code that could be executed, depending on multiple conditions.

Activity 14: Working with the if Statement

We have recently walked through a simple **if** statement block that results to a value based on the input given. This is given here:

```
release_year = 1991
answer = input('When was Python first released?')


if answer == release_year:
  print('Congratulations! That is correct.')
elif answer > release_year:
  print('No, that\'s too late')
elif answer < release_year:
  print('No, that\'s too early')


print('Bye!')
```

Modify this block as follows:

- The user prompt should be **Return TRUE or FALSE: Python was released in 1991**.
- If the user inputs `TRUE`, print out **Correct**.
- If the user inputs `FALSE`, print out **Wrong**.
- Any other user input should result in **Please answer TRUE or FALSE**.

Create a script called `if_python.py` and add your code to it. Then, run it in a terminal. The output should be as follows:

```
>python if_python.py

Return TRUE or FALSE: Python was released in 1991:

TRUE

Correct

Bye!

>python if_python.py

Return TRUE or FALSE: Python was released in 1991:

FALSE

Wrong

Bye!
```

> **Note**
>
> Solution for this activity can be found at page 283.

The while Statement

A `while` statement allows you to execute a block of code repeatedly, as long as a condition remains true. That is to say, *as long as condition* X *remains true, execute this code.*

A `while` statement can also have an `else` clause that will be executed exactly once when the condition, X, that's mentioned is no longer true. This can be read as follows: *As long as* X *remains true, execute this block of code, else, execute this other block of code immediately so that the condition is no longer true.*

For instance, consider the traffic police officer we mentioned earlier, who could be letting traffic through while the exit is clear, and as soon as it is not clear, they stop the drivers from exiting.

Here is the basic syntax of a **while** statement:

```
while condition:
  # Run this code while condition is true
  # Replace the "condition" above with an actual condition
  # This code keeps running as long as the condition evaluates to True
else:
  # Run the code in here once the condition is no longer true
  # This code only runs one time unlike the code in the while block
```

As mentioned earlier, this can be read as: *As long as the condition is true, execute the first block of code, and if the condition is not true, execute the second block of code.*

Exercise 15: Using the while Statement

Now we will look at a practical use of the **while** statement:

1. First, declare a variable called **current** and set its value to **1**:

   ```
   current = 1
   ```

2. Declare the **end** variable and set its value to **10**:

   ```
   end = 10
   ```

3. Then, write a **while** statement.

   The condition of this **while** statement is that the **current** number is less than or equal to the **end** number:

   ```
   while current <= end:
   ```

4. For every **current** number, print it out and then increment it by 1:

   ```
     print(current)
     current += 1
   ```

5. If immediately the condition is not true, print the statement **You have reached the end**:

   ```
   else:
     print('You have reached the end')
   ```

Note that the statement **`You have reached the end`** is printed out only once. This demonstrates how to implement the **`else`** clause with a **`while`** statement.

The final code is as follows:

```
# Set the starting value
current = 1
# Set the end value
end = 10

# While the current number is less than or equal to the end number
while current <= end:
  # Print the current number
  print(current)
  # Increment the current number by one
  current += 1
else:
  """
  Immediately the current number is not less than or equal to the end
  number, print this statement.
  Note that the statement is only printed out once
  """
  print('You have reached the end')
```

6. Run the script. The output of this code is as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
1
2
3
4
5
6
7
8
9
10
You have reached the end
```

As expected, Python loops through all numbers from 1 to 10 and prints them out to standard output. At the end of the loop, the program prints the line **`You have reached the end`** as expected. This marks the end of program execution, and the program exits.

Exercise 16: Using while to Keep a Program Running

To show the power of **while** loops, let's look at an example that uses a **while** loop to keep an interactive Python program running until a condition is met. To do this, we will rewrite the program we wrote earlier and add a **while** statement:

1. Set **release_year** to **1991**. This is the correct answer to the question that is going to be asked:

   ```
   release_year = '1991'
   ```

2. Next, set the **correct** condition to **False**. We will be using this condition to check whether we should break out of the **while** statement:

   ```
   correct = False
   ```

3. Now go into the body of the **while** statement. While the answer provided is not correct, keep the program running. Notice that we use a negative condition, where we check whether the provided answer is incorrect. Implement this as follows:

   ```
   while not correct:
   ```

4. Print out the question to the terminal:

   ```
       answer = input('When was Python first released?')
   ```

   Note the use of **input()**. This tells the terminal to wait for user keyboard input.

5. Use an **if** statement to check that the provided answer is correct:

   ```
       if answer == release_year:
   ```

6. If the answer is correct, print a success message to the terminal:

   ```
           print('Congratulations! That is correct.')
   ```

7. After printing the message, set **correct** to **True**. This will cause the **while** loop to stop executing:

   ```
           correct = True
   ```

8. If the answer is incorrect, encourage the user to try again:

   ```
       else:
           print('No, that\'s not it. Try again\n')
   ```

9. Finally, print the exit message:

    ```
    print('Bye!')
    ```

    The final code is as follows:

    ```
    # Set release_year to 1991. This is the correct answer to the question to
    be asked
    # Note that this is a string as user input is automatically set to type
    str
    release_year = '1991'
    # Set the condition "correct" to False
    correct = False

    # While the answer provided is not correct, keep the program running
    while not correct:
      # Print out the question to stdout
      # Note the use of input(). This tells the terminal to wait for user
    keyboard input
      answer = input('When was Python first released?')

      # Use an if statement to check that the provided answer is correct
      if answer == release_year:
        # If the answer is correct, print success message to stdout
        print('Congratulations! That is correct.')
        # After printing message, set correct to True
        # This will cause the while loop to stop executing
        correct = True
      else:
        # If the answer is incorrect, encourage user to try again
        print('No, that\'s not it. Try again\n')

    # Finally, print the exit message
    # Note that this is only printed just before the program exits
    print('Bye!')
    ```

10. Run the script. Here's the output with incorrect answers first, and then the correct answer:

    ```
    Python 3.6.1 (default, Dec 2015, 13:05:11)
    [GCC 4.8.2] on linux

    When was Python first released? 1841
    No, that's not it. Try again

    When was Python first released? 2001
    No, that's not it. Try again

    When was Python first released? today
    No, that's not it. Try again

    When was Python first released? 1991
    Congratulations! That is correct.
    Bye!
    ```

The **while** loop is used in the following cases:

- When we must wait for a condition to be satisfied before continuing execution
- When a user's input is required—as seen in the second example code snippet

Activity 15: Working with the while Statement

The aim of this activity is to create a password authentication feature using a **while** loop.

You have been asked to add a simple password authentication feature for a prototype that your team is setting up. One of your team members has advised the use of a **while** loop.

The steps are as follows:

1. Create a script called **while_python.py**.
2. Define **random** as the password and the Boolean validator first.
3. Initiate the **while** loop and ask for the user's input.
4. Validate the password and return an error if the input is invalid.
5. Run the script in the terminal.

The output should be as follows:

```
>python while_python.py
please enter your password: random
Welcome back user!

>python while_python.py
please enter your password: none
invalid password, try again...
please enter your password:
```

> **Note**
>
> Solution for this activity can be found at page 283.

while Versus if

The main difference between an **`if`** and a **`while`** statement is that an **`if`** statement gives you the opportunity to branch the execution of code once based on a condition. The code in the **`if`** block is only executed once. For instance, if a value is greater than another, an **`if`** statement will branch out and execute a computation, and then proceed with the program's flow or exit.

A **`while`** statement, however, gives you the opportunity to run a block of code multiple times as long as a condition evaluates to true. This means that a **`while`** statement will, for example, execute a computation as long as value A is greater than value B and only proceed with the program flow when A is no longer greater than B.

In this sense, a **`while`** statement can be considered a loop. We'll look at looping structures next.

Loops

In Python, looks (just as in any other language) are a way to execute a specific block of code several times. In particular, loops are used to iterate or loop over what we call iterables.

For the purposes of this chapter, we can define an iterable as follows:

- Anything that can be looped over (that is, you can loop over a string or a file)
- Anything that can appear on the right-hand side of a **`for`** loop, for example, **`for x in iterable`**

A few examples of common iterables include the following:

- Strings
- Lists
- Dictionaries
- Files

You can think of an iterable as a collection of homogeneous things that have been grouped together to form a large collective. The individuals in the group have the same properties, and when they are combined, they form something new.

Consider the example of cars in a car yard. We can consider the car yard as the collection or iterable and the individual cars as the constituent members of the car yard. If you were shopping for a car, you would probably have a couple of qualities that you are looking for. You walk into the car yard and start going from car to car looking for one that satisfies your conditions or comes as close as possible to satisfying your conditions. The act of going from car to car and repeating the aforementioned inspection operation is basically what looping is.

Loops allow us to deconstruct iterables and perform operations on their constituent members or even convert them into new data structures. The possibilities are endless once you start using loops.

The for Loop

The **`for`** loop in Python is also referred to as the **`for…in`** loop. This is due to its unique syntax that differs a bit from **`for`** loops in other languages.

A **`for`** loop is used when you have a block of code that you would like to execute repeatedly a given number of times. For example, multiplying an iterable value, or dividing the value by another if the iterable value is still present in the loop.

The loop contrasts and differs from a **while** statement in that in a **for** loop, the repeated code block is ran a predetermined number of times, while in a **while** statement, the code is ran an arbitrary number of times as long as a condition is satisfied.

The basic syntax of a **for** loop is shown here:

```
# Iterable here can be anything that can be looped over e.g. a list
# Member here is a single constituent of the iterable e.g. an entry in a
list
for member in iterable:
  # Execute this code for each constituent member of the iterable
  pass
```

As shown in the preceding code, the **for** loop allows you to go through each constituent member of an iterable and run a block of code for each member. This code could be anything from a simple summation of the values to more complex manipulations and analysis. Again, the possibilities are endless, and having the ability to easily access iterables like this will prove invaluable as you start building more complex programs in Python.

Exercise 17: Using the for Loop

A practical use of the **for** loop is shown here:

1. First, declare a variable called **numbers**. We initialize **numbers** to a list of integers from 1 to 10:

   ```
   numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
   ```

2. Next, loop through the list we just created.

   Create a new variable called **num** in the **for** loop. We will use this variable to represent and access the individual numbers in the list as we loop through it:

   ```
   for num in numbers:
   ```

3. Inside the loop, calculate the square of **num** by multiplying it by itself. Note that there are other ways to calculate the square, but this rudimentary method will suffice for our example here.

   We assign the square of **num** to the variable **square**:

   ```
     square = num * num
   ```

4. Then, print out the string that tells us that **num** squared is **square**.

    ```
    print(num ,' squared is ', square)
    ```

    This loop is repeated for all ten numbers in the list.

    The final code is as follows:

    ```
    # Create a list with numbers 1 through 10
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

    # Loop through the list of numbers
    for num in numbers:
      # Calculate the square of the number
      square = num * num
      # Print out a string showing the number and its calculated square
      print(num ,' squared is ', square)
    ```

5. Run the script. The output of our example is as follows:

    ```
    Python 3.6.1 (default, Dec 2015, 13:05:11)
    [GCC 4.8.2] on linux

    1  squared is  1
    2  squared is  4
    3  squared is  9
    4  squared is  16
    5  squared is  25
    6  squared is  36
    7  squared is  49
    8  squared is  64
    9  squared is  81
    10  squared is  100
    ```

As expected, the loop iterates over the list numbers, calculates the square of each number, and prints out the result in a readable format.

Using else

As with a **while** statement, an **else** statement can also be optionally used with a **for** loop. In this case, the code inside the **else** block will be executed exactly once when the loop exits cleanly. Exiting cleanly means that the loop went through all the members of the iterable without breaking.

Here is that same code snippet from earlier with an **`else`** clause added:

```
# Create a list with number 1 through 10
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Loop through the list of numbers
for num in numbers:
  # Calculate the square of the number
  square = num * num
  # Print out a string showing the number and its calculated square
  print(num ,' squared is ', square)
else:
  print('The last number was ', num)
```

The **`else`** clause is added to the very bottom of the loop.

The output from this code will be as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux

1  squared is  1
2  squared is  4
3  squared is  9
4  squared is  16
5  squared is  25
6  squared is  36
7  squared is  49
8  squared is  64
9  squared is  81
10  squared is  100
The last number was  10
```

As we can see, the **else** clause is executed only once and only after the rest of the code has been ran successfully. It is also important to note that although it may not seem like it, the **else** clause has access to all the variables created within the **for** loop. This can prove very valuable for debugging and handling error conditions in our code.

The range Function

Python's **range** function is a built-in function that generates a list of numbers. This list is mostly used to iterate over using a **for** loop.

This function is used when you want to perform an action a predetermined number of times, where you may or may not care about the index, for instance, finding or calculating all even numbers between 0 and 100, where Python will list or print out all even numbers in that range, excluding 100, even though it is an even number. You can also use it to iterate over a list (or another iterable) while keeping track of the index.

The basic syntax of the range function is as follows:

```
range([start], stop, [step])
```

Here is a breakdown of what each parameter does:

- **start**: This is the starting number of the sequence.
- **stop**: This means generate numbers up to but not including this number.
- **step**: This is the difference between each number in the sequence.

As a general rule, when a parameter is enclosed in square brackets **[]** in the function definition, it means that that particular parameter is optional when you are calling the function.

This means that the only required parameter when calling the range function is the **stop** parameter, and the default call to the function can have just that one parameter.

Now, let's look at some calls to the **range** function and their corresponding output. First off, perform a call with just the **stop** parameter included:

```
range(10)
```

The **range(10)** class basically tells the function to generate numbers from 0 to 10, but not including 10. To view the numbers, we would have to cast the **range** object into a **list** object. The output of this call is shown here:

```
Python 3.7.0 (default, Jun 29 2018, 20:13:13)
[Clang 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> range(10)
range(0, 10)


>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Please note that the **range** function starts at zero if a start point is not defined as with this form of the function call.

What if we wanted to create a list from 1 to 10? To achieve this, you would have to include the **start** parameter of the **range** function. You would also have to change the **stop** parameter, too. Remember that the **stop** parameter is not included in the final generated list.

The pseudo-code for what you want to achieve is: *Generate a list of all numbers from 1 to 10.*

Here is how we do it:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

How is the **step** parameter used? The **step** parameter defines the difference between each number in the generated list. If you set **step** to 5, the difference between each number on the list will be a constant of 5. This can be useful when you want to generate a very particular set of numbers between any given two points.

Let's write a piece of code that generates a list of all even numbers between 2 and 20. Here, we can see the application of the **step** parameter. We will use it to make sure that the difference between each number is 2, thus ensuring that the final list will only contain even numbers.

Here is how we can do it:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
> list(range(2, 21, 2))
> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Conversely, you can generate a list of all odd numbers between 1 and 20 by tweaking the code a bit and changing that first parameter **2** to **1** so that the function starts at **1** instead of **2**:

```
> list(range(1, 21, 2))
```

Now that we are familiar with how **range()** works, we can look at some practical applications of **range** in a real program. Consider the following code:

```
for num in range(1, 11):
  print(num ,' squared is ', num * num)
```

This is a rewrite of the code that we used earlier with the **for** loop. The **for** loop code is shown here for comparison:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


for num in numbers:
  square = num * num
  print(num ,' squared is ', square)
```

It is evident that we are able to achieve the same result using far less code and without having to initialize variables that may not be used outside of this loop. This helps keep our code nice and clean.

Activity 16: The for loop and the range Function

Using a **`for`** loop and a **`range`** function, you have been asked to find the even numbers between 2 and 100 and then find their sum.

The steps are as follows:

1. Define a counter for the sum.
2. Define a **`for`** loop with an even range for numbers between 2 and 101.
3. Add each looped number to the sum.
4. Outside the loop, print out the total sum.

The output should be as follows:

```
>>> print(total)
2550
```

> **Note**
>
> Solution for this activity can be found at page 284.

Nesting Loops

Nesting can be defined as the practice of placing loops inside other loops. Although it is frowned upon in some applications, sometimes, it is necessary to nest loops to achieve the desired effect.

One of the use cases for nesting loops is when you need to access data inside a complex data structure. It could also be as a result of a comparison of two data structures. For instance, a computation that requires values in two lists would have you loop through both lists and execute the result. It is sometimes necessary to use one or more loops inside another loop to get to that level of granularity.

Exercise 18: Using Nested Loops

In this exercise, we will utilize nested loops:

1. First, create a variable called **groups**, which is a list that contains three other lists of three integers each:

   ```
   groups = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
   ```

   From the structure of the list, it is clear that just looping over it will not get us to the individual integers. To get to the individual integers, we have to loop over the groups and then loop over each group.

2. So, first, loop over **groups**:

   ```
   for group in groups:
   ```

3. For each **group**, loop over it to reach the individual integers, which we call **num**:

   ```
     for num in group:
   ```

4. Then, proceed to calculate the square of each number:

   ```
       square = num * num
   ```

5. Finally, print out the statement that shows the number's square:

   ```
       print(num ,' squared is ', square)
   ```

   The final code is as follows:

   ```
   # Create a list with three groups of numbers 1 through 9
   groups = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

   # Loop through the list of number groups
   for group in groups:
     # Loop through numbers in group
     for num in group:
       # Calculate the square of the number
       square = num * num
       # Print out a string showing the number and it's calculated square
       print(num ,' squared is ', square)
   ```

6. Run the script. The output of the preceding code is as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux

1  squared is  1
2  squared is  4
3  squared is  9
4  squared is  16
5  squared is  25
6  squared is  36
7  squared is  49
8  squared is  64
9  squared is  81
```

As you can see, we are able to loop over the individual groups of integers and perform calculations on the individual integers by nesting the **`for`** loops. This is the power of nesting: it can allow us to unpack complex structures.

There is no limit to how far you can nest loops, though you should keep code readability in mind when writing nested loops. You don't want to nest so much that you cannot easily deduce what the code does or the expected results of running the code at a glance. Remember, you are writing your code not just for the computer but for future developers who might need to work on it. Oftentimes, this future developer is you in a couple of months. You don't want to look at some code that you wrote six months from now and not be able to comprehend it. Do yourself a favor: keep it simple.

Activity 17: Nested Loops

Write a simple nested loop example that sums the even and odd numbers between 1 and 11 and prints out the computation.

The steps are as follows:

1. Write a **`for`** loop with a **`range`** function for **`even`** numbers.
2. Write a **`for`** loop with a **`range`** function for **`odd`** numbers.
3. Use a variable called **`val`** to calculate the sum of **`even`** and **`odd`**.
4. Print **`val`** to the terminal.

The output will be as follows:

```
2 + 1 = 3
2 + 3 = 5
2 + 5 = 7
2 + 7 = 9
2 + 9 = 11
4 + 1 = 5
4 + 3 = 7
4 + 5 = 9
4 + 7 = 11
4 + 9 = 13
6 + 1 = 7
6 + 3 = 9
6 + 5 = 11
6 + 7 = 13
6 + 9 = 15
8 + 1 = 9
8 + 3 = 11
8 + 5 = 13
8 + 7 = 15
8 + 9 = 17
10 + 1 = 11
10 + 3 = 13
10 + 5 = 15
10 + 7 = 17
10 + 9 = 19
```

Note

Solution for this activity can be found at page 284.

Breaking Out of Loops

When running loops, sometimes, we might want to interrupt or intervene in the execution of the loops before it runs its full course due to an external factor. For instance, when writing a function looping though a list of numbers, you may want to break when a defined condition external to the program flow is met. We will demonstrate this further.

Python provides us with three statements that can be used to achieve this:

- **`break`**
- **`continue`**
- **`pass`**

The break Statement

The **`break`** statement allows you to exit a loop based on an external trigger. This means that you can exit the loop based on a condition external to the loop. This statement is usually used in conjunction with a conditional **`if`** statement.

The following is an example program that shows the **`break`** statement in action:

```
# Loop over all numbers from 1 to 10
for number in range(1,11):
  # If the number is 4, exit the loop
  if number == 4:
    break

  # Calculate the product of number and 2
  product = number * 2
  # Print out the product in a friendly way
  print(number, '* 2 = ', product)

print('Loop completed')
```

The output for this code is as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux

1 * 2 =  2
2 * 2 =  4
3 * 2 =  6
Loop completed
```

As shown by the output, the loop exits when **`number`** is equal to **`4`** because of the conditional **`if number == 4`** statement.

The continue Statement

The **`continue`** statement allows you to skip over the part of a loop where an external condition is triggered, but then goes on to complete the rest of the loop. This means that the current run of the loop will be interrupted, but the program will return to the top of the loop and continue execution from there.

As with the **`break`** statement, the **`continue`** statement is usually used in conjunction with a conditional **`if`** statement.

Here is the same code from earlier with the **`break`** statement replaced with a **`continue`** statement:

```
# Loop over all numbers from 1 to 10
for number in range(1,11):
  # If the number is 4, continue the loop from the top
  if number == 4:
    continue

  # Calculate the product of number and 2
  product = number * 2
  # Print out the product in a friendly way
  print(number, '* 2 = ', product)


print('Loop completed')
```

The output then becomes the following:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux


1 * 2 =  2
2 * 2 =  4
3 * 2 =  6
5 * 2 =  10
6 * 2 =  12
7 * 2 =  14
8 * 2 =  16
9 * 2 =  18
10 * 2 =  20
Loop completed
```

Note the difference between this and the output of the **break** statement. This loop, instead of exiting when it reached **4**, simply skips over the rest of the loop when **number** is **4** and resumes execution at the top of the loop. The product of **4** is therefore never printed to the screen, but the rest of the output is printed normally.

The pass Statement

The **pass** statement allows you to handle an external trigger condition without affecting the execution of the loop. This is to say that the loop will continue to execute as normal unless it hits a **break** or **continue** statement somewhere later in the code.

As with the other statements, the **pass** statement is usually used in conjunction with a conditional **if** statement.

Here is the code from the **break** statement example, with **break** replaced with **pass**:

```
# Loop over all numbers from 1 to 10
for number in range(1,11):
  # If the number is 4, proceed as normal
  if number == 4:
    pass

  # Calculate the product of number and 2
```

```
    product = number * 2
    # Print out the product in a friendly way
    print(number, '* 2 = ', product)

print('Loop completed')
```

The output of this code will be as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux

1 * 2 =  2
2 * 2 =  4
3 * 2 =  6
4 * 2 =  8
5 * 2 =  10
6 * 2 =  12
7 * 2 =  14
8 * 2 =  16
9 * 2 =  18
10 * 2 =  20
Loop completed
```

As evident, the code encounters the condition but does nothing based on it. The **pass** statement simply tells the program to proceed as normal. The **pass** statement is also mostly used as a placeholder.

Activity 18: Breaking out of Loops

You have been asked to write a loop that outputs values in a database column ranging between 0 and 200. Zero, any number that is not divisible by 3, and any value that is not an integer, should be ignored. One of your team members has advised the use of **break**, **continue**, and **pass** statements.

The steps are as follows:

1. Define a `for` loop.
2. Define a condition that checks for zero.
3. Define a condition that checks whether the number is divisible by 3.
4. Define a condition that checks the data type.

The output will be as follows:

```
3
6
9
12
15
…
…
174
177
180
183
186
189
192
195
198
```

> **Note**
>
> Solution for this activity can be found at page 284.

Summary

In this chapter, we have learned about how programs in Python flow. We also learned how to control and branch the flow of a Python program by using the two main control statements, that is, **`if`** and **`while`**. We have also looked at some practical applications of the two control statements and have seen how they differ in implementation and syntax.

In this chapter, we have also increased our knowledge of looping structures. We have seen the structure of a **`for`** loop and looked at practical examples. We have also looked into the **`range`** function and how it comes in handy when you need to quickly iterate over a list.

Apart from that, we have also covered how and when to nest loops and how to break out of loops prematurely under different conditions and with differing results by using the **`break`**, **`continue`**, and **`pass`** statements.

Armed with this knowledge, you can now start incorporating more complex structures into your programs.

In the next chapter, we will look at functions and how to define them, the various types of functions, and how they can help us compartmentalize our code.

4 Functions

Learning Objectives

By the end of this chapter, you will be able to:

- Describe the various function types in Python
- Define global and local variables
- Define a function that takes in a variable number of arguments

This lesson introduces functions in Python. We look at the various types of functions and define our own.

Introduction

In the previous chapter, we covered the following topics:

- How to use looping structures
- How to branch within looping structures
- How to break out of loops

We will continue to build on this knowledge by implementing what we have learned, to build functions in Python.

Functions are an integral part of the Python programming language, and a lot of languages, really. Throughout this book, you have already encountered some built-in functions, especially when dealing with certain data structures.

Functions are an easy way to group a few lines of code that implement a functionality together. This is especially useful if the code in question will be used several times in different parts of your program. You may want to use functions to abstract away some complex code that you need in your programs. You can think of functions as mini-programs within your bigger program that implement specific tasks.

It is important to remember that while it is tempting to tuck a lot of functionality into a single function, it is better to write functions that only perform one specific task. This makes it easier to modularize your code, and, in the long run, it is more maintainable and easier to debug.

Functions may take optional inputs to work with and may optionally return a value or values.

The three main types of functions in Python are as follows:

- Built-in functions
- User-defined functions
- Anonymous functions

Built-In Functions

The Python interpreter has a number of built-in functions and types that are always available. These are called **built-in functions**, and they can be used anywhere in your code, without the need of any importation.

Some of the built-in functions that we have already encountered in this book are as follows:

- **`input([prompt])`**: This optionally prints the prompt to the terminal. It then reads a line from the input and returns that line.
- **`print()`**: Prints objects to the text stream file or the terminal.
- **`map()`**: Returns an iterator that applies a function to every item of the iterable, yielding the results.

For example, we recently used the built-in **`print()`** function to output results; the following is another simple demonstration:

```
print("Hello world")
```

This results in the following:

```
Hello world
```

User-Defined Functions

As the name suggests, these are functions that are written by the user, to aid them in achieving a specific goal. The main use of functions is to help us organize our programs into logical fragments that work together to solve a specific part of our problem.

The syntax of a Python function looks like this:

```
def function_name( parameter_one, parameter_two, parameter_n ):
    # Logic goes here
    return
```

To define a function, we can use the following steps:

1. Use the **`def`** keyword, followed by the function name.
2. Add parameters (if any) to the function within the parentheses. End the function definition with a full colon.
3. Write the logic of the function.
4. Finally, use the **`return`** keyword to return the output of the function. This is optional, and if it is not included, the function automatically returns **`None`**.

A user-defined function must have a name. You can give it any name that you like, and it is a good practice to make the name as descriptive of the task that the function achieves as possible. For example, if you are writing a function that calculates the monthly average rainfall from a list of values, it is better to name the function **`calculate_monthly_average_rainfall()`** than **`calculate()`**. *Remember, code is written to be read by humans, not computers*. Make it easy for other humans to immediately understand what a function does, just by looking at the name:

```
def calculate_monthly_average_rainfall(list_of_annual_values):

  # Loop over list and calculate average here

  return average
```

Parameters are the information that needs to be passed to the function, in order for it to do its work. Parameters are optional, and they are separated by commas and placed between parentheses after the function name. A function can have any number of parameters, and there is really no limit to this.

Calling a Function

Calling a function means executing the logic that is defined inside of the function. This can be done from the Python interactive prompt, or from within some other part of your code. Functions are often called from other functions.

If we were to call the function that we defined earlier (the one that calculates the monthly rainfall average), we would just do something like this:

```
annual_values = []

calculate_monthly_average_rainfall(annual_values)
```

Note that the names of the arguments that we pass when we are calling the function in this instance do not have to match the parameter names that the function expects. What is important is that no matter what you pass to the function when calling it, the function will refer to this parameter as **`list_of_annual_values`** internally.

This transitions us smoothly to our next sub-topic: global and local variables.

Global and Local Variables

Variables that are defined inside of a function body are called **local variables**, as they are only accessible inside the function. They are said to have a local scope.

Variables that are defined outside of a function body are called **global variables**, as they are accessible both outside and inside of the functions. They are said to have a global scope because of this.

Exercise 19: Defining Global and Local Variables

In this exercise, we will define global and local variables. We will also demonstrate the difference between global and local variables:

1. First, define a global variable, **number**, and initialize it to **5**:

   ```
   number = 5
   ```

2. Then, define a function called **summation** that takes two named parameters- **first** and **second**:

   ```
   def summation(first, second):
   ```

3. Inside of the function, add up the two parameters that were passed in and the global variable **number**:

   ```
     total = first + second + number
   ```

4. Return the final **total**:

   ```
     return total
   ```

5. Call the **summation** function with two parameters, as expected (**10** and **20**):

   ```
   summation(10, 20)
   ```

6. Print out the initial value of the **number**:

   ```
   print("The first number we initialised was " + str(number))
   ```

7. Try to access the local variable **total**:

   ```
   print("The total after summation is " + str(total))
   ```

   Note the use of the built-in function, **str()**, which returns the string version of an object.

   The final code is as follows:

   ```
   # Initialise global variable "number" to 5
   number = 5

   """
   Define function "summation" that takes two parameters
   Note that the function accesses the global variable "number"
   """
   def summation(first, second):
     # Add the parameters and global number together
     total = first + second + number
     # Return result
   ```

```
    return total

  # Call the "summation" function with two parameters as expected
  summation(10, 20)

  # Print out the initial value of "number"
  print("The first number we initialised was " + str(number))

  # Try to access the local variable "total"
  print("The total after summation is " + str(total))
```

8. Run this script. The output of running this code is as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux

The first number we initialised was 5
Traceback (most recent call last):
  File "main.py", line 21, in <module>
    print("The total after summation is " + str(total))
NameError: name 'total' is not defined
```

   As you can see, this results in an error, because we are trying to access the local variable **`total`** from the global scope. This is just to demonstrate that we cannot access local variables globally.

9. Now, change the code to the following; we will get different output upon calling the function:

```
# Initialise global variable "number" to 5
number = 5

"""
Define function "summation" that takes two parameters
Note that the function accesses the global variable "number"
"""
def summation(first, second):
  # Add the parameters and global number together
  total = first + second + number
  # Return result
  return total

# Call the "summation" function with two parameters as excepted
# Assign the result of "summation" to the variable "outer_total"
```

```
outer_total = summation(10, 20)

# Print out the initial value of "number"
print("The first number we initialised was " + str(number))

# Try to access the local variable "total"
print("The total after summation is " + str(outer_total))
```

Notice that we are now assigning the result of the **`summation`** function to the **`outer_total`** variable.

10. Run the script again. The output now changes to what we expect, without errors, and it looks as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux

The first number we initialised was 5
The total after summation is 35
```

Function Return

The **`return`** statement in Python is used within functions, to actually return something to the caller of the function. Without a **`return`** statement, every function will return **None**.

Consider the following function:

```
def summation(first, second):
  total = first + second
  print("The total is " + str(total))


summation(10, 20)


The output of this code is:


Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux


The total is 30
```

Note that the **summation** function, in this case, does not have a **return** statement. As mentioned previously, a **return** statement is not necessary for all functions. The purpose of this function in particular is to print out the total, and in this case, it is not necessary to return anything, as the printing can be done within the function.

However, a **return** statement is required if you need to use the result of calling a function for any further processing in your code. Consider the following variation:

```
def summation(first, second):
  total = first + second
  return total


outer_total = summation(10, 20) * 2


print("Double the total is " + str(outer_total))
```

In this variation, we can see that the **summation** function returns the sum of the passed-in values. This returned value is then multiplied by two and assigned to the **outer_total** variable. This way, the function has, in essence, abstracted away the operation of summing the two numbers.

Of course, this is just a rudimentary example, and with this knowledge, you can begin to build more complex functions and programs.

Using main()

Most other programming languages (for example, Java and C++) require a special function, called **main()**, which tells the operating system what code to execute when a program is invoked. This is not necessary in Python, but you will find that it is a good and logical way to structure a program.

Before the Python interpreter executes our program, it defines a few special variables. One of them is **__name__**, and it will automatically be set to **__main__** if our program will be executed by itself, in a standalone fashion.

However, if our program will be imported by another program, then **__name__** will be set to the name of that other program. We can easily determine whether the program is standalone or is being used by another program as an import. Based on that, we can decide to either execute or exclude some of the code in a program.

The following is an example that uses the **`main()`** function:

```
def summation(first, second):
  total = first + second
  return total


def main():
  outer_total = summation(10, 20) * 2
  print("Double the total is " + str(outer_total))


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

Without the **`if __name_ == __main__`** check and declaration, our script would still be executable. All we would have to do is declare or call our **`summation()`** function.

In Python, there is nothing special about the name **`main`**. We could have called this function anything that we wanted. We chose **`main`** to be consistent with some of the other languages.

Function Arguments

As mentioned earlier, parameters are the information that need to be passed to the function for it to do its work. Although parameters are also commonly referred to as arguments, arguments are thought of more as the actual values or references assigned to the parameter variables when a function is called at runtime. In simpler terms, arguments are to functions as ingredients are to a recipe.

Python supports several types of arguments; namely:

- Required arguments
- Keyword arguments
- Default arguments
- A variable number of arguments

Required Arguments

Required arguments are the types of arguments that have to be present when calling a function. These types of arguments also need to be in the correct order for the function to work as expected.

Consider the following code snippet:

```
def division(first, second):
  return first/second
```

You have to pass the arguments **`first`** and **`second`** for the function to work. You also have to pass the arguments in the correct order, as switching them will yield completely different results.

You would then call the function like this:

```
quotient = division(10, 2)
```

The result, as expected, would be as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux

5.0
```

Keyword Arguments

If it is necessary that you call all of the parameters in the right order, you can use keyword arguments in your function call. You can use these to identify the arguments by their parameter names. Let's consider the previous example to make this a bit clearer:

```
def division(first, second):
  return first/second

quotient = division(second=2, first=10)
print(quotient)
```

As you can see, we have intentionally passed arguments in the wrong order by swapping their positions. The difference here is that we used the names of the arguments when passing them. By doing this, we corrected our *mistake*, and the function will receive its parameters in the right order and give us the correct (and expected) output, which is as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux

5.0
```

Keyword arguments are very powerful, and they ensure that no matter which order we pass arguments in, the function will always know which argument goes where.

Default Arguments

Default arguments are those that take a default value if no argument value is passed during the function call. You can assign this default value with the assignment operator, **=**, just like in the following example:

```
def division(first, second=2):
  return first/second

quotient = division(10)
print(quotient)
```

The output for this example is the same as that of the previous examples.

Note that even if the argument named **second** has a default value, you can still pass a value to it, and this passed value will override the default value. This means that the function will promptly ignore the default value and use whatever value you passed to it.

The following is an example with the default argument where a value is passed:

```
def division(first, second=2):
  return first/second

quotient = division(10, 5)
print(quotient)
```

As expected, the output of this snippet is:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux

2.0
```

Variable Number of Arguments

It might so happen that you want to allow a function to receive any number of variables, and then process them. Wouldn't it be convenient if you could pass a variable number of arguments to this function? Well, you're in luck! This is possible in Python by using the special **`*`** (asterisk) syntax.

The following is an example of using **`*args`**:

```
def addition(*args):
  total = 0
  for i in args:
    total += i
  return total

answer = addition(20, 10, 5, 1)
print(answer)
```

Note that you don't have to name the variable **`*args`**. You could have named it **`*numbers`**, and the function would have worked just as well.

Activity 19: Function Arguments

Write a function that receives *n* number of arguments; using a **`continue`** statement, skip integers and print out all other values.

The steps are as follows:

1. Define a function named **`print_arguments`**, with a variable number of arguments.
2. Use a **`for`** loop to iterate over the arguments.
3. Use a check to see whether the value passed is of the integer type. If it is, use the **`continue`** statement to ignore it.
4. Print the arguments.

The output should be as follows:

```
>>> print_arguments(2, 3.4, "s", 1.0)
3.4
s
1.0
```

> **Note**
>
> Solution for this activity can be found at page 285.

Anonymous Functions

Anonymous functions in Python are also called **lambda functions**. This is because they use the keyword **`lambda`** in their definition.

Anonymous functions are so called because, unlike all of the other functions that we have looked at up to this point, they do not require to be named in their definition. The functions are usually throwaway, meaning that they are only required where they are defined, and are not to be called in other parts of the codebase.

The syntax of an anonymous function is as follows:

```
lambda argument_list: expression
```

The argument list consists of a comma-separated list of arguments, and the expression is an arithmetic expression that uses these arguments. You can assign the function to a variable to give it a name.

Exercise 20: Creating a Lambda Function

Let's look at a practical application of a lambda function. Suppose that you want to simply sum up two numbers in a function. There is no need to define a whole user-defined function for this if you are only going to use it once:

1. Create a lambda function that takes two parameters - **`first`** and **`second`**:

   ```
   answer = lambda first, second : first + second
   print(answer(6, 9))
   ```

   Here, we have defined a lambda function that takes two parameters (**`first`** and **`second`**) and adds them up. We have then assigned the function to the **`answer`** variable, so that we have a way to refer to it.

2. Now, call the function and print out the output, which looks like this:

    ```
    Python 3.6.1 (default, Dec 2015, 13:05:11)
    [GCC 4.8.2] on linux

    15
    ```

The true power of anonymous functions can be seen when they are used in combination with the **map()**, **reduce()**, or **filter()** functions.

The syntax of a **map()** function is as follows:

```
map(func, iterable)
```

The first argument, **func**, is the name of a function, and the second, **iterable**, is a sequence (for example, a list). **map()** applies the **func** function to all of the elements of the **iterable** sequence. It returns a new list, with the elements changed by **func**.

Let's suppose that you have the following list, and want to generate a new list with the squares of every item in the list:

```
numbers = [2, 4, 6, 8, 10]
```

One way to implement this would be:

```
numbers = [2, 4, 6, 8, 10]

squared = []


for num in numbers:

  squared.append(num**2)
```

The **for** loop, in this case, can be replaced with a lambda function that serves the same purpose, like this:

```
lambda num: num ** 2
```

We can then use the **map()** function to apply this lambda function to each of the items in the list, in order to get their squares, like this:

```
squared = map(lambda num: num ** 2, numbers)
```

This will yield a **map** object, and, to cast this to a list, we use the **list()** function, like this:

```
squared = list(map(lambda num: num ** 2, numbers))
```

Our whole program then reduces to just three lines:

```
numbers = [2, 4, 6, 8, 10]
squared = list(map(lambda num: num ** 2, numbers))


print(squared)
```

The output of these lines is as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux


[4, 16, 36, 64, 100]
```

This definitely shows the versatility of lambda/anonymous functions, and their ability to make our code more concise.

Activity 20: Using Lambda Functions

Write a lambda function that takes in two numerical values and returns the first value, raised to the power of the second value:

1. Create a lambda function that takes in **number** and **power** and returns the value of the *number raised to power*.
2. Assign it to a variable called **answer**.
3. Print the **answer** variable.

The output should be as follows:

```
>>> print(answer(2, 4))
16
```

> **Note**
>
> Solution for this activity can be found at page 285.

Summary

In this chapter, we learned about the various types of functions in Python, as well as their differences, syntax, and use cases. We covered how and where to apply the different types of functions, and how they can be used to help break your programs into smaller sub-programs that achieve a specific purpose. We also saw how the use of functions can help use reuse functionality in our code and avoid repeating the same blocks of code.

With this knowledge, you should be able to build all sorts of well-structured programs that will be easy to read and understand, and which will make optimal use of repetitive functionality.

In the next chapter, we will take a look at lists and tuples; it will be our first chapter regarding the various data structures that Python offers.

5

Lists and Tuples

Learning Objectives

By the end of this chapter, you will be able to:

- Create and access lists in Python
- Describe the various methods that are available in lists, and use them in your Python programs
- Create and access tuples in Python
- Identify the differences between tuples and lists
- Implement the various built-in methods that are available with tuples

This lesson introduces lists and tuples in Python. We also look at the various list and tuple methods.

Introduction

A **list** is a data structure that holds ordered collections of related data. Lists are known as **arrays** in other programming languages, like Java, C, and C++. Python lists, however, are more flexible and powerful than the traditional arrays of other languages.

An example of this power is that the items in a list do not have to all be of the same type. In other words, we can have a list whose items are strings, integers, or even other lists. The items in a list can be of any of the Python types.

The main properties of Python lists are as follows:

- They are ordered.
- They contain objects of arbitrary types.
- The elements of a list can be accessed by an index.
- They are arbitrarily nestable, that is, they can contain other lists as sublists.
- They have variable sizes.
- They are mutable, that is, the elements of a list can be changed.

Tuples are used to hold together multiple related objects. They are similar to the lists discussed previously, in that they are also sequence data types, but differ in that they don't have all the functionality afforded by lists. The key difference between lists and tuples is that you cannot change the elements of a tuple once they are set. This property of tuples is called **immutability**.

List Syntax

As you saw previously, lists can be created in several ways. The simplest of them is to enclose the elements of the list in square brackets `[]`.

Here, you can see several flavors of lists:

```
# Empty list
[]


# List containing numbers
[2, 4, 6, 8, 10]


# List with mixed types
["one", 2, "three", ["five", 6]]
```

The first is an initialization of an empty list. The second is a list whose elements are numbers. The last one is an example of a list that contains several types. It has numbers, strings, and sublists inside of it.

List Methods

The list data type has some built-in methods that can be used with it. These methods are as follows:

- **`list.append(item)`**
- **`list.extend(iterable)`**
- **`list.insert(index, item)`**
- **`list.remove(item)`**
- **`list.pop([index])`**
- **`list.clear()`**
- **`list.index(item [, start [, end]])`**
- **`list.count(item)`**
- **`list.sort(key=None, reverse=False)`**
- **`list.reverse()`**
- **`list.copy()`**

Let's take a closer look at what these methods can do.

list.append(item)

The **`list.append(item)`** method adds a single item to the end of a list. This doesn't return a new list – it only modifies the original. The following is an example of this in use:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   things = ["first"]

   things.append("another thing")

   things

=> ['first', 'another thing']
```

list.extend(iterable)

The **`list.extend(iterable)`** method takes one argument, which should be an iterable data type. It then extends the list by appending all of the items from the iterable to the list. What would happen is we would extend a list with another list:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   things

=> ['first', 'another thing']
   others = ["third", "fourth"]

   things.extend(others)

   things

=> ['first', 'another thing', 'third', 'fourth']
```

The following code shows what would happen if you passed a string to the **`extend`** method:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)

[GCC 4.8.2] on linux

   things = ["first"]


   things.append("another thing")


   things


=> ['first', 'another thing']


   things.extend("another thing")


   things


=> ['first', 'another thing', 'a', 'n', 'o', 't', 'h', 'e', 'r', ' ', 't',
'h', 'i', 'n', 'g']
```

Note that the **`extend`** method used the string as an iterable. This means that it looped through every character in the string (including the space) and appended those characters to the list.

list.insert(index, item)

The **`list.insert(index, item)`** method (as the name suggests) inserts an item at a given position in a list. The method takes two arguments, **`index`** and **`item`**. The **`index`** is the position in the list before which to insert the item defined in the second argument. Both arguments are required.

To insert an item at the beginning of a list, you would do the following:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   things = ["second"]

   things

=> ['second']

   things.insert(0, "first")

   things

=> ['first', 'second']
```

list.remove(item)

The **`list.remove(item)`** method removes from the list the first item whose value matches the argument item that's passed in:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   things

=> ['first', 'second']

   things.remove("second")

   things

=> ['first']
```

list.pop([index])

The **`list.pop([index])`** method removes the item at the given index of the list, and returns it. The **`index`** is an optional argument, and if it isn't passed in, the method will remove the last item in the list.

The following code shows how it works:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   things

=> ['first', 'second', 'third']
   second_item = things.pop(1)

   second_item

=> 'second'
   things

=> ['first', 'third']
```

The argument that is passed in is the index of the item that you would like to pop, and not the position. Python list indices start at zero.

list.clear()

As the name suggests, the **`list.clear()`** method removes all items from a list. An alternative to this method would be **`del a[:]`**.

list.index(item [, start [, end]])

The **`list.index(item [, start [, end]])`** method returns the index of the first item in the list whose value is **`item`**.

The parameters **`start`** and **`end`** are optional, and they indicate the position in the list at which the search for said item should start and end, respectively. Again, the very beginning of the list is index zero. The index at the end of the list is one less than the length of the list.

To search the entire list, simply pass in the **`item`** parameter, like this:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   things = ['first', 'second', 'third']

   things.index('second')

=> 1
```

An example of where the **`start`** and **`end`** parameters are specified is as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   things = ['first', 'second', 'third', 'fourth']

   things.index('third', 1, 3)

=> 2
```

If the specified item is not found in the list, Python raises a **`ValueError`**, as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   things = ['first', 'second', 'third', 'fourth']

   things.index('fifth')

Traceback (most recent call last):
  File "python", line 1, in <module>
ValueError: 'fifth' is not in list

   things.index('fourth', 0, 2)

Traceback (most recent call last):
  File "python", line 1, in <module>
ValueError: 'fourth' is not in list
```

list.count(item)

The **`list.count(item)`** method returns the number of times the given item occurs in the list.

list.sort(key=None, reverse=False)

The **`list.sort(key=None, reverse=False)`** method sorts the items of the list.

list.reverse()

The **`list.reverse()`** method reverses the elements of the list.

list.copy()

The **`list.copy()`** method returns a shallow copy of a list.

Activity 21: Using the List Methods

Using your recently gained knowledge of how to use various list methods, work out the following tasks. The aim of this activity is to get acquainted with list methods. Suppose that we have the following list:

```
Lion, Zebra, Panther, Antelope
```

We need to change the contents of this list using the various available methods:

1. Using the built-in **`append()`** method, add the value **`Elephant`** to a list named **`wild`** that is comprised of the following elements:

   ```
   Lion, Zebra, Panther, Antelope
   ```

2. Using the built-in **`extend()`** method, extend an empty **`animals`** list from the **`wild`** list that you created previously.
3. Insert **`Cheetah`** into the **`animals`** list at index **`2`** and replace the element on index **`1`** with **`Giraffe`**, using the built-in methods **`insert()`** and **`pop()`**.
4. Finally, sort the elements in the **`animals`** list by using the built-in method **`sort()`**.

> **Note**
>
> Solution for this activity can be found at page 285.

List Comprehensions

List comprehensions are a feature of Python that give us a clean, concise way to create lists.

A common use case would be when you need to create a list where each element is the result of some operations applied to each member of another sequence or iterable object.

The syntax for a list comprehension is pretty similar to creating a list the *traditional* way. It consists of square brackets **`[]`** containing an expression followed by a **`for`** clause, then zero or more **`if`** clauses.

Let's go back to the example we looked at previously, where we needed to calculate the squares of numbers. To calculate the squares of all numbers from one to ten using a **`for`** loop, we could do the following:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   for num in range(1,11):
     print(num**2)


1
4
9
16
25
36
49
64
81
100
```

The same can be achieved by using list comprehensions, as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   squares = [num**2 for num in range(1, 11)]


   squares


=> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

We can also use **`if`** statements within list comprehensions. There are really no limitations in what can be done with comprehensions. The following is an example of the preceding code, using an **`if`** statement:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   squares = [num**2 for num in range(1, 11) if num%2 == 0]

   squares

=> [4, 16, 36, 64, 100]
```

This particular snippet only adds to the list those squares whose roots are even numbers.

Tuple Syntax

The main advantages of using tuples, rather than lists, are as follows:

- They are better suited for use with different (heterogeneous) data types.
- Tuples can be used as a key for a dictionary (we'll see dictionaries in the next chapter). This is due to the immutable nature of tuples.
- Iterating over tuples is much faster than iterating over lists.
- They are better for passing around data that you don't want changed.

A tuple consists of a number of individual values, separated by commas (just like lists). As with lists, a tuple can contain elements of different types. You create a tuple by placing all of the comma-separated values in parentheses, **`()`**, like this:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   pets = ('dog', 'cat', 'parrot')
   pets
=> ('dog', 'cat', 'parrot')
   type(pets)
=> <class 'tuple'>
```

The parentheses are optional, and you might as well create a tuple using just the comma-separated values, as follows:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   pets = 'dog', 'cat', 'parrot'
   pets
=> ('dog', 'cat', 'parrot')
   type(pets)
=> <class 'tuple'>
```

Note that the output version of a tuple will always be enclosed in parentheses, no matter which method you used to create it. This prevents confusion and allows us to better interpret tuples visually.

In this example of a nested tuple, we can see why outputting in this format is important:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   pets = ('dog', 'cat'), 'parrot'
   pets
=> (('dog', 'cat'), 'parrot')
```

Here, we can see that there is a nested tuple, **`('dog', 'cat')`**, inside of the main tuple, and the use of multiple parentheses makes this abundantly clear. This also displays another property of tuples; they can be nested with no limit. Be careful when nesting tuples, though, as several nesting levels may make it difficult to understand and navigate the structure.

What if we wanted to create a tuple with one element? We could just enclose it in parentheses, right? Consider the following code:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   one = ('thing')
   one
=> 'thing'
   type(one)
=> <class 'str'>
```

Wrong! As you can see in the preceding example, creating a tuple in this way will only result in a string being created.

You have to recall one very important part of the definition of a tuple: it is a collection of comma-separated values. The comma is very important; therefore, you would have to do something like the following to create a tuple with one element:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)

[GCC 4.8.2] on linux

   one = 'thing',

   one

=> ('thing',)

   type(one)

=> <class 'tuple'>
```

Exercise 21: Creating a Tuple

In this exercise, we will create a tuple of our own. We will create a **`vehicle`** tuple, add the elements listed as follows, and verify the object type.

The elements are as follows:

- **`Toyota`**
- **`BMW`**
- **`Benz`**

1. Create the tuple and populate the elements:

   ```
   >>> vehicle = ('Toyota', 'BMW', 'Benz')
   ```

2. Verify the object type:

   ```
   >>> type(vehicle)
   <class 'tuple'>
   ```

Accessing Tuple Elements

Tuples give us various ways to access their elements. These are as follows:

- Indexing
- Slicing

Indexing

Similar to lists, we can use the index operator **[]** to access an element in a tuple by using its index. Tuple indices start at zero, just like those of lists.

Exercise 22: Accessing Tuple Elements Using Indexing

In this exercise, you will see how to use indexing to access tuples:

1. Create a new **pets** tuple with the elements **dog**, **cat**, and **parrot**:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   pets = 'dog', 'cat', 'parrot'
```

2. Run **pets[1]** to access the second index:

```
   pets[1]
=> 'cat'
```

3. Try to access an index that is not in the tuple. Python will raise an **IndexError**, as shown here:

```
   pets[3]
Traceback (most recent call last):
  File "python", line 1, in <module>
IndexError: tuple index out of range
```

4. Indices can also be negative. If you use a negative index, **-1** will reference the last element in the tuple, **-2** will refer to the second from last element in the tuple, and so on. Use the following code to access the tuple:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   pets = 'dog', 'cat', 'parrot', 'gerbil'
   pets[-1]
=> 'gerbil'
   pets[-2]
=> 'parrot'
   pets[-3]
=> 'cat'
```

5. As with list indices, tuple indices must always be integers, and trying to use other types will result in a **TypeError**, as shown in the following code. Type a string and a float value as an index:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
   pets = 'dog', 'cat', 'parrot'
   pets['1']
Traceback (most recent call last):
  File "python", line 1, in <module>
TypeError: tuple indices must be integers or slices, not str
   pets[1.5]
Traceback (most recent call last):
  File "python", line 1, in <module>
TypeError: tuple indices must be integers or slices, not float
```

The **error** message presented here mentions **slices**, and you might be wondering what slices are. This brings us to the alternative way in which you can access tuple elements: slicing.

Slicing

While indexing allows you to access an individual element in a tuple, slicing allows you to access a subset, or a slice, of the tuple.

Slicing uses the slicing operator, `:`, and the general syntax is as follows:

```
tupleToSlice[Start index (included):Stop index (excluded):Increment]
```

You have to recall one very important part of the definition of a tuple: it is a collection of comma-separated values. The comma is very important; therefore, you would have to do something like the following to create a tuple with one element:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)

[GCC 4.8.2] on linux

   one = 'thing',

   one

=> ('thing',)

   type(one)

=> <class 'tuple'>
```

Exercise 21: Creating a Tuple

In this exercise, we will create a tuple of our own. We will create a **`vehicle`** tuple, add the elements listed as follows, and verify the object type.

The elements are as follows:

- **`Toyota`**
- **`BMW`**
- **`Benz`**

1. Create the tuple and populate the elements:

   ```
   >>> vehicle = ('Toyota', 'BMW', 'Benz')
   ```

2. Verify the object type:

   ```
   >>> type(vehicle)
   <class 'tuple'>
   ```

Accessing Tuple Elements

Tuples give us various ways to access their elements. These are as follows:

- Indexing
- Slicing

Indexing

Similar to lists, we can use the index operator `[]` to access an element in a tuple by using its index. Tuple indices start at zero, just like those of lists.

All of these parameters are optional, and this is how they work:

- **`Start index`**: The index at which to start the slicing. The element at this index is included in the slice. If this parameter is absent, it is assumed to be zero, and thus, the slicing starts at the beginning of the tuple.
- **`Stop index`**: The index at which to stop slicing. The element at this index is not included in the slice. This means that the last item in this slice will be the one just before the stop index. If this parameter is absent, the slice ends at the very end of the tuple.
- **`Increment`**: This determines how many steps to take in the tuple when creating the slice. If this parameter is absent, it is assumed to be one.

Exercise 23: Accessing Tuple Elements Using Slicing

In this exercise, we will take a look at some practical examples of slicing. Consider a tuple of numbers from one through ten:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)

[GCC 4.8.2] on linux

   numbers = tuple(range(1,11)])

   numbers

=> (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

We will create a list using list comprehensions and cast it to a tuple by using the **`tuple()`** method.

To demonstrate tuple slicing, we will to do two things:

- Create a new tuple containing only the even numbers in the tuple
- Create a new tuple containing only the odd numbers in the tuple

1. To slice the tuple so that we have only the even numbers, start at the second index and finish at the end, while skipping one place every step.

   In code, that looks something like this:

   ```
   Python 3.6.1 (default, Dec 2015, 13:05:11)
   [GCC 4.8.2] on linux
      numbers = tuple([i for i in range(1,11)])
      numbers
   => (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
      even = numbers[1::2]
      even
   => (2, 4, 6, 8, 10)
   ```

   For this operation, we set the start index to **`1`** (the second element of the tuple) and the increment to **`2`**, so that it skips one element every time.

2. To slice the tuple so that we have only the odd numbers, start at the beginning of the tuple and finish at the end, while skipping one place every step.

   In code, that looks something like this:

   ```
   Python 3.6.1 (default, Dec 2015, 13:05:11)
   [GCC 4.8.2] on linux
      numbers = tuple([i for i in range(1,11)])
      odd = numbers[::2]
      odd
   => (1, 3, 5, 7, 9)
   ```

   For this operation, we start at index zero (the first element of the tuple) and set the increment to **`2`**, so that it skips one element every time.

Tuple Methods

Python has the following methods that work with tuples:

- **`any()`**: This method can be used to discover whether any element of a tuple is an iterable:

  ```
  >>> pets = ('cat', 'dog', 'horse')
  >>> any(pets)
  >>> TRUE
  ```

- **`count()`**: This method returns the number of occurrences of an item in a tuple.

  This is also the only bound method, as the syntax describes: **`tuple.count(element)`**. The other methods are unbound:

  ```
  >>> pets = ('cat', 'dog', 'horse')
  >>> pets.count("cat")
  >>> 1
  ```

- **`min()`**: This method returns the smallest element in a tuple:

  ```
  >>> pets = ('cat', 'dog', 'horse')
  >>> min(pets)
  >>> 'cat'
  ```

- **`max()`**: This method returns the largest element in a tuple:

  ```
  >>> pets = ('cat', 'dog', 'horse')
  >>> max(pets)
  >>> 'horse'
  ```

- **`len()`**: This method returns the total number of elements in a tuple:

  ```
  >>> pets = ('cat', 'dog', 'horse')
  >>> len(pets)
  >>> 3
  ```

- Tuples, just like strings, can be concatenated. This is shown here:

  ```
  >>> pets = ('cat', 'dog', 'horse')
  >>> wild = ('lion', 'zebra', 'antelope')
  >>> animals = pets + wild
  >>> print(animals)
  >>> ('cat', 'dog', 'horse', 'lion', 'zebra', 'antelope')
  ```

Activity 22: Using Tuple Methods

Write a script that uses tuple methods to count the number of elements in a tuple and the number of times a particular element appears in the tuple. Based on these counts, the script will also print a message to the terminal.

The steps are as follows:

1. Initialize a tuple, as follows:

   ```
   pets = ('cat', 'cat', 'cat', 'dog', 'horse')
   ```

2. Use a tuple method to count the number of times the **`cat`** element appears in the tuple **`pets`**, and assign it to a variable, **`c`**.
3. Use a tuple method to calculate the length of the tuple **`pets`**, and assign it to a variable, **`d`**.
4. If the number of times the **`cat`** element appears in the tuple **`pets`** is more than 50%, print **`There are too many cats here`** to the terminal. If not, print **`Everything is good`** to the terminal.

> **Note**
>
> Solution for this activity can be found at page 286.

Summary

In this chapter, we expanded our knowledge on Python lists. We have looked at the various methods that are available for lists, and how to use them in practical applications. We have also seen how we can use list comprehensions to make the task of building lists programmatically easier.

Next, we covered Python tuples. We looked at the various methods that are available for tuple operations, and how to use them in practical applications. We have also seen how we can access tuple elements.

In the next chapter, we will cover sets and dictionaries, which are the other data structures that Python offers.

6

Dictionaries and Sets

Learning Objectives

By the end of this chapter, you will be able to:

- Create and use dictionaries
- Use methods and attributes associated with dictionaries
- Describe and use ordered dictionaries to store and retrieve data in a predictable order
- Create sets, as well as add, read, and remove data from them
- Describe the attributes defined on set objects
- Describe frozen sets

This lesson describes dictionaries and sets. We cover creating, reading, and writing data to these data structures.

Introduction

You have already seen lists that hold values that you can access by using indexes. However, what if you wanted to name each value, instead of using an index? For example, suppose that you want to access a list of cake ingredients, but you do not know where in the array it is. In that case, a dictionary would come in handy.

Dictionaries, sometimes referred to as associative arrays in other languages, are data structures that hold data or information in a key-value order. Dictionaries allow you to access whatever value you want, using the much easier to remember key.

Dictionaries, unlike lists, are indexed using keys, which are usually strings. There are two kinds of dictionaries that you can use in Python: the default **`dict`**, which is unordered, and a special kind of dictionary called an **`OrderedDict`**. The difference is that the keys in the default dictionary are stored in an unordered manner, whereas an **`OrderedDict`** stores key-value pairs in the order of insertion.

A **set** is a collection of data items that are unordered and unique, that is, items cannot be repeated. For example, *[1, 1]* is a valid list, but not a valid set. With no duplicates in sets, we are able to perform mathematical operations, such as unions and intersection. You can store any kind of valid string or integer in a set, so long as it is unique.

For background information, sets are very important objects in mathematics, as well. A whole section of mathematics, called *Set Theory*, has been dedicated to the study of sets and their properties. We will not get into a lot of the mathematics around sets, but we will discuss some basic set operations.

Suppose that we have a set, A = {1,2,3}, and another set, B = {3,4,5}.

The **union** of set A and B, mathematically denoted as A *u* B, will be {1,2,3,4,5}. The union is simply the set of everything in A and B. Remember, there are no duplicates, so 3 will appear only once.

On the other hand, the **intersection** of A and B, denoted as A *n* B, will be the set of everything that is common in both A and B, which, in our case, is just {3}.

Working with Dictionaries

You can create a dictionary in two ways. The first way is to simply assign an empty dictionary to a variable by using curly brackets, like so:

```
dictionary = {}
```

The second way is to use the **`dict()`** function to return a new dictionary:

```
dictionary = dict()
```

In both cases, a **dictionary** object will be created. We can inspect the attributes and properties defined on the **dictionary** object by using the built-in **dir()** function:

```
>>> dir(dictionary)
['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__
doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__',
'__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__
subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop',
'popitem', 'setdefault', 'update', 'values']
```

We can also confirm that we have an actual dictionary by using the built-in **isinstance()** function to check that **dictionary** is an instance of the **dict** class:

```
>>> isinstance(dictionary, dict)
True
```

The **isinstance()** function is used to check the type of an object. It takes two arguments, the first one being the object being inspected, and the second being the class that we want to type-check against; for example, **int**, **str**, **dict**, **list**, and so on.

Activity 23: Creating a Dictionary

In this activity, we will create a dictionary and verify its type. The steps are as follows:

1. Use the **dict()** or curly bracket notation to create a dictionary and assign it to a variable, **d**.
2. Use the built-in **type()** function on **d** to see if it is an instance of the **dict** class.

> **Note**
>
> Solution for this activity can be found at page 286.

A typical example of a dictionary is as follows:

```
d = dict(
    state="NY",
    city="New York"
  )
```

In this example, **state** and **city** are **keys**, while **NY** and **New York** are the respective **values** assigned to them.

Exercise 24: Adding Data to a Dictionary

In this exercise, we will see how to add data to a dictionary:

1. First, add data to a dictionary during its creation by using the **`dict`** function, like this:

   ```
   dictionary1 = dict(
       state="NY",
       city="New York"
     )

   print(dictionary1)

   dictionary2 = {
     "state": "Maryland",
     "city": "Baltimore"
   }

   print(dictionary2)
   ```

   Notice that when we use the **`dict()`** function, we assign values to keys using the **`=`** operator. When using **`{}`**, we separate the keys (which are strings) from the values by using **`:`**. Running this code will print the following:

   ```
   {'state': 'NY', 'city': 'New York'}
   {'state': 'Maryland', 'city': 'Baltimore'}
   ```

2. Assign values to existing dictionaries using keys, like this:

   ```
   dictionary2['bird'] = 'Baltimore oriole'
   ```

   This will add a new key to **`dictionary2`**, with the name **`bird`** and the value **`Baltimore oriole`**.

   ```
   >>> print(dictionary2)
   {'state': 'Maryland', 'city': 'Baltimore', 'bird': 'Baltimore oriole'}
   ```

3. On the other hand, using the preceding format with an existing key will reassign that key to a new value. Assign a new value to the key **`state`** of **`dictionary1`**, like this:

   ```
   dictionary1['state'] = 'New York'
   ```

   This will change the value of **`state`** from **`NY`** to **`New York`**, like so:

   ```
   >>> print(dictionary1)
   {'state': 'New York', 'city': 'New York'}
   ```

Exercise 25: Reading Data from a Dictionary

In this exercise, we will look at how to read data from a dictionary:

1. You can access dictionary values via their keys, like so:

    ```
    print(dictionary1['state'])
    >>> NY
    ```

    Using this format, if the key does not exist, we will get a **`KeyError`**. For example:

    ```
    print(dictionary1['age'])
    Traceback (most recent call last):
      File "python", line 13, in <module>
    KeyError: 'age'
    ```

2. To avoid this, we can also access values from dictionaries by using their **`get()`** function.

    The **`get()`** function returns **`None`** if an item does not exist. You can also use the **`get()`** function to specify what should be returned when no value exists. Use the **`get()`** function, as shown here:

    ```
    print(dictionary1.get('state'))
    print(dictionary1.get('age'))
    print(dictionary1.get('age', 'Key age is not defined'))
    ```

    This will output the following:

    ```
    NY
    None
    Key age is not defined
    ```

Exercise 26: Iterating Through Dictionaries

In this exercise, we will look at how to iterate through dictionary values by using various methods:

1. The simplest way to iterate through dictionaries is to use a **`for`** loop. Use a **`for`** loop as follows, in order to get a dictionary's keys:

    ```
    dictionary1 = dict(
        state="NY",
        city="New York"
      )

    for item in dictionary1:
      print(item)
    ```

This code, by default, iterates through the dictionary's keys, and will print the following:

```
state
city
```

2. You can also explicitly iterate through only the keys or the values of a dictionary by calling the **`keys()`** method, which returns a list of keys, or the **`values()`** method, which returns a list of values in the dictionary. Use the **`keys()`** method as follows, in order to print the keys:

```
dictionary1 = dict(
    state="NY",
    city="New York"
  )

for item in dictionary1.keys():
  print(item)
```

This will also output the following:

```
state
city
```

3. Use the **`values()`** method as follows, in order to print the values:

```
dictionary1 = dict(
    state="NY",
    city="New York"
  )

for item in dictionary1.values():
  print(item)
```

The output for this will be as follows:

```
NY
New York
```

4. You can also iterate through both keys and values at the same time. Use a **for** loop, as follows, in order to print both keys and values:

```
dictionary1 = dict(
    state="NY",
    city="New York"
  )

for key, value in dictionary1.items():
  print(key, value)
```

This will output the following:

```
state NY
city New York
```

Checking for the Existence of Particular Keys

You can use the **in** keyword to check whether a particular key exists in a dictionary, without iterating through it. This works the same way as it does in lists, and you will get back a Boolean value of **True** if the key exists, and **False** if it doesn't:

```
a = {
  "size": "10 feet",
  "weight": "16 pounds"
}


print("size" in a)
print("length" in a)
```

This example outputs the following:

```
True
False
```

Additional Dictionary Attributes

If you run the **dir()** function on a dictionary, you will see a few more attributes defined that we have not yet touched upon. The following is a sample output:

```
['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__
doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__
iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '__
sizeof__', '__str__', '__subclasshook__', 'clear', 'copy',
'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update',
'values']
```

Let's go through some of these attributes and see what they can do.

dict.update()

The **.update()** method on dictionaries, is used to insert new key-value pairs into a dictionary, or update the value of an existing one.

For example, if we have an empty dictionary, calling **.update()** will insert a new entry:

```
>>> a = {}
>>> a.update({"name": "Dan Brown"})
>>> a
{'name': 'Dan Brown'}
```

What if we update the **name** key again? Consider the following code:

```
>>> a.update({"name": "Dan Brown Xavier"})
>>> a
{'name': 'Dan Brown Xavier'}
```

As you can see, calling **`.update()`** with an existing key replaces the value of that key. Also, note that the **`.update()`** function takes a dictionary with the key-value pairs defined, in order to update the existing dictionary. This means that the **`.update()`** function would come in handy if you had two dictionaries with different keys that you wanted to combine into one.

dict.clear() and dict.pop()

The **`clear`** method is used to remove all keys from a dictionary. For example:

```
>>> a
{'name': 'Dan Brown Xavier'}
>>> a.clear()
>>> a
{}
```

If you only want to remove one key-value pair, you can use the **`del`** keyword, like so:

```
>>> a = {"name": "Skandar Keynes", "age": "24"}
>>> del a["name"]
>>> a
{'age': '24'}
```

What if you want to remove a key-value pair from the dictionary and do something with the value? In that case, you can use **`pop()`**, which will delete the entry from the dictionary and return the value:

```
>>> a = {"name": "Skandar Keynes", "age": "24"}
>>> b = a.pop("name")
>>> a
{'age': '24'}
>>> b
'Skandar Keynes'
```

dict.copy()

The **copy** method is used to create shallow copies of dictionaries. For example:

```
>>> a = {"name": "Skandar Keynes", "age": "24"}
>>> b = a.copy()
>>> b
{'name': 'Skandar Keynes', 'age': '24'}
>>> a["name"] = "Janet Jackson"
>>> a
{'name': 'Janet Jackson', 'age': '24'}
>>> b
{'name': 'Skandar Keynes', 'age': '24'}
```

In this example, you can see that **b** is a **shallow copy** of **a**, and has all of the exact key-value pairs found in **a**. However, updating **a** will not update **b**. They exist as two different entities.

This is different from using the **=** operator to make a **deep copy**, where **a** and **b** will refer to the same object, and updating one will update the other:

```
>>> a = {"name": "Skandar Keynes", "age": "24"}
>>> b = a
>>> a["name"] = "Janet Jackson"
>>> b["age"] = 16
>>> a
{'name': 'Janet Jackson', 'age': 16}
>>> b
{'name': 'Janet Jackson', 'age': 16}
```

dict.popitem()

The **`popitem()`** method pops and returns a random item from the dictionary. That item will no longer exist in the dictionary after that. For example:

```
>>> a = {"name": "Skandar Keynes", "age": "24", "sex": "male"}
>>> a.popitem()
('sex', 'male')
>>> a.popitem()
('age', '24')
>>> a
{'name': 'Skandar Keynes'}
```

dict.setdefault()

The **`setdefault()`** method takes two arguments: a key to be searched for in the dictionary, and a value. If the key exists in the dictionary, its value will be returned. If the key does not exist, it will be inserted with the value provided in the second argument. If no second argument was passed, any insertion will be done with the value **None**.

Let's look at an example where the key exists in the dictionary:

```
>>> a = {"name": "Skandar Keynes", "age": "24", "sex": "male"}
>>> b = a.setdefault("name")
>>> a
{'name': 'Skandar Keynes', 'age': '24', 'sex': 'male'}
>>> b
'Skandar Keynes'
```

In this case, the value is returned as-is, and the dictionary is left untouched. Passing the second argument in this case will have no effect, since a value already exists.

Let's look at another example, where the key does not exist in the dictionary, and a value was passed. In this case, the key-value pair will be added to the dictionary, and the value will be returned, as well:

```
>>> a = {"name": "Skandar Keynes", "age": "24", "sex": "male"}
>>> b = a.setdefault("planet", "Earth")
>>> a
{'name': 'Skandar Keynes', 'age': '24', 'sex': 'male', 'planet': 'Earth'}
>>> b
'Earth'
```

Now, let's look at a final example, where the key does not exist in the dictionary, and no value was passed. In this case, the key will be added with a value of **None**. Nothing will be returned:

```
>>> a = {"name": "Skandar Keynes", "age": "24", "sex": "male"}
>>> b = a.setdefault("planet")
>>> a
{'name': 'Skandar Keynes', 'age': '24', 'sex': 'male', 'planet': None}
>>> b
>>>
```

dict.fromkeys()

The **dict.fromkeys()** method is used to create a dictionary from an iterable of keys, with whatever value is provided by the user. An iterable is anything that you can iterate over (for example, using a **for** loop).

Let's look at an example of this:

```
>>> a = dict.fromkeys(["name", "age"], "Nothing here yet")
>>> a
{'name': 'Nothing here yet', 'age': 'Nothing here yet'}
```

Note that if you do not provide a second argument, the values will be auto-set to **None**:

```
>>> a = dict.fromkeys(["name", "age"])
>>> a
{'name': None, 'age': None}
```

Activity 24: Arranging and Presenting Data Using Dictionaries

Due to its key-value pair format, dictionaries can be used to present some forms of structured data in an easily readable way. This activity will help you write code to present frequency data for characters in strings.

Dictionaries are very good data structures for organizing data in a readable format. Your aim is to write a function called **sentence_analyzer**, which will take in a line of text and output a dictionary with each letter as a key, the value being the frequency of appearance of the letter.

The following is an example of what the output would look like:

```
sentence_analyzer("Pythonn")
>>> {
  "P": 1,
  "y": 1,
  "t": 1,
  "h": 1,
  "o": 1
  "n": 2
}
```

The steps are as follows:

1. Define the function and have it take a string argument.
2. Iterate through the string and count the occurrences of each character.
3. Output each character as a key in the output dictionary, with the frequency as the value.

The following are some hints to help you out:

- Treat different uppercase and lowercase letters separately. For example, **`P`** and **`p`** should have different counts.
- Drop empty spaces.
- Assume that sentences will not have any special characters.

> **Note**
>
> Solution for this activity can be found at page 287.

Ordered Dictionaries

So far, the dictionaries that we have created do not maintain the insertion order of the key-value pairs that are added. **Ordered dictionaries** are dictionaries that maintain the insertion order of keys. This means that when you are iterating through them, you will always access the keys in the order in which they were inserted.

The **`OrderedDict`** class is a **`dict`** subclass defined in the **`collections`** package that Python ships with. We will use ordered dictionaries when it is vitally important to store and retrieve data in a predictable order; for example, when reading database entries.

The following section will describe how to work with them.

Creating an ordered dictionary is as easy as creating an instance of the **`OrderedDict`** class and passing in key-value pairs:

```
>>> from collections import OrderedDict
>>> a = OrderedDict(name="Zeus", role="god")
>>> a
OrderedDict([('name', 'Zeus'), ('role', 'god')])
```

Everything about **`OrderedDict`**, except for it maintaining an internal key order, is similar to normal dictionaries. However, if you inspect the attributes defined on it using **`dir()`**, you may see some new ones that were not in the normal **`dict()`** class that we looked at previously:

```
['__class__', '__contains__', '__delattr__', '__delitem__',
'__dict__','__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__geta ttribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__
init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__
new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subcla sshook__', 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys',
'move_to_end', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

One of the new attributes is **`move_to_end`**, which moves a key contained in the **`OrderedDict`** from its current position to the very end of the dictionary.

Note

You can look up more information about these attributes at https://docs.python.org/3/library/collections.html#collections.

Note that when you are checking whether two **`OrderedDict`** are equal, the order of keys is also considered. Although for a normal **`dict`**, having the same key-value pairs is enough to declare equality, in **`OrderedDict`**, if they are not in the same order, those two objects are not equal.

Activity 25: Combining Dictionaries

Suppose that you have data in two different places and you need to work with both at the same time. Python gives you the ability to extend dictionaries with data from each other. This activity will help you learn how to combine dictionaries, or extend one dictionary with the contents of another.

Write a function called **`dictionary_masher`** that will take two dictionaries and return a single dictionary with non-duplicated keys.

The following shows an example output:

```
>>> dictionary_masher({"name": "Amos"}, {"age": 100})
>>>
{
   "name": "Amos",
   "age": 100
}
```

> **Note**
>
> Solution for this activity can be found at page 287.

The Basics of Sets

In this section, we are going to cover sets, which are unique data structures with interesting properties.

Let's begin our journey into sets by looking at how to create sets, how to read data from them, and how to remove data from them.

Exercise 27: Creating Sets

In this exercise, we will create a set. We can do so by using the **set** method or the curly bracket notation:

1. The first way to create a set is to use the **set** method. In Python, you can use the built-in **set** function to create a set. The function takes either an iterable (like a list or a tuple) or a sequence (lists, tuples, and strings are all sequences). Use the **set** method to define the sets named **a** and **b**, like this:

   ```
   >>> a = set([1,2,3])
   >>> a
   {1, 2, 3}
   >>> b = set((1,2,2,3,4))
   >>> b
   {1, 2, 3, 4}
   ```

   Note that in set **b**, all duplicated values in the original tuple were dropped.

2. Another way is to use the curly bracket notation. Create a set directly, by using the curly bracket notation, like this:

```
>>> c = {'a', 'b', 'c'}
>>> c
{'c', 'b', 'a'}
```

3. Pass a sequence to the **`set()`** method, like this. As you can see, this builds a set of the sequence's constituent elements:

```
>>> a = set("A random string")
>>> a
{'A', 'n', 'm', ' ', 's', 'a', 'r', 'o', 'g', 'd', 't', 'i'}
```

> **Note**
>
> Passing a dictionary to the **`set()`** method will create a set of its keys.

4. Inspect the set by using the **`dir()`** function; you should see the following output:

```
['__and__', '__class__', '__contains__', '__delattr__', '__dir__', '__
doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__iand__', '__init__', '__init_subclass__', '__ior__', '__
isub__', '__iter__', '__ixor__', '__le__', '__len__', '__lt__', '__ne__',
'__new__', '__or__', '__rand__', '__reduce__', '__reduce_ex__', '__
repr__', '__ror__', '__rsub__', '__rxor__', '__setattr__', '__sizeof__',
'__str__', '__sub__', '__subclasshook__', '__xor__', 'add', 'clear',
'copy', 'difference', 'difference_update', 'discard', 'intersection',
'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop',
'remove', 'symmetric_difference', 'symmetric_difference_update', 'union',
'update']
```

We will look at some of the attributes defined on the **`set`** object later on in this chapter.

Exercise 28: Adding Data to a Set

In this exercise, we will look at how to add data to a set by using the **`set.add()`** and **`update()`** methods:

1. Add data to an existing set by using the **`set.add()`** method:

   ```
   >>> a = {1,2,3}
   >>> a
   {1, 2, 3}
   >>> a.add(4)
   >>> a
   {1, 2, 3, 4}
   ```

2. Of course, since this is a set, adding a value more than once will not change the set. The item will only appear once. Add **`4`** to set **`a`** again, in order to verify this:

   ```
   >>> a
   {1, 2, 3, 4}
   >>> a.add(4)
   >>> a
   {1, 2, 3, 4}
   ```

3. You can also use the set's **`update()`** method to add items to the set, by using iterables. Use the **`update()`** method to pass in a list of values, like this:

   ```
   >>> a = {1,2,3}
   >>> a
   {1, 2, 3}
   >>> a.update([3,4,5,6])
   >>> a
   {1, 2, 3, 4, 5, 6}
   ```

Exercise 29: Reading Data from a Set

Set objects do not support indexes, so you cannot access values from them using indexes, like you can in a list or a tuple. In this exercise, we will look at what other methods we can use to read data from a set:

1. Iterate through the set by using a **`for`** loop:

   ```
   a = {1,2,3,4}

   for num in a:
     print(num)
   ```

This will output the following:

```
1
2
3
4
```

2. Use the set's **`pop()`** method to remove and return an item from the set. Items are removed from the beginning of the set, as follows:

```
>>> a
{1, 2, 3, 4}
>>> a.pop()
1
>>> a
{2, 3, 4}
>>> a.pop()
2
>>> a
{3, 4}
```

Sets have more utility in the actions that we can perform on them than as a store of data. This means that finding things like the union and intersection of items in sets gives us more insight into the data they hold.

Activity 26: Building a Set

Imagine that you have some data in a list that contains a lot of duplicates that you want to remove. However, you think writing a **`for`** loop and building a new list without duplicates is too much overkill. Can you find a more efficient way to remove the duplicated values from the list?

The aim of this activity is to build a set out of a random set of items. Write a function called **`set_maker`** that takes a list and turns it into a set. Do not use a **`for`** loop or **`while`** loop to do so.

The following is an example output:

```
>>> set_maker([1, 1, 2, 2, 2, 3, 4, 6, 5, 5])
{1, 2, 3, 4, 5, 6}
```

> **Note**
>
> Solution for this activity can be found at page 287.

Exercise 30: Removing Data from a Set

There are other ways to remove data from sets without using **pop()**, especially if you just want to remove the data and not return it. In this exercise, we will look at how we can do this:

1. Use the **remove()** method, like this:

```
>>> a = {1,2,3}
>>> a.remove(3)
>>> a
{1, 2}
```

2. As you can see, **remove()** drops the item from the set and does not return it. If you try to remove a non-existing item, a **KeyError** will be raised. For set **a**, remove the value **3**:

```
>>> a
{1, 2}
>>> a.remove(3)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    a.remove(3)
KeyError: 3
```

3. You can also use the **discard()** method. For comparison, **discard()** does not raise a **KeyError** if the item to be discarded does not exist. Use the **discard** method, like this:

```
>>> a = {1,2,3}
>>> a.discard(2)
>>> a
{1, 3}
>>> a.discard("nonexistent item")
>>> a
{1, 3}
```

4. Lastly, use the **`clear()`** method to remove all of the data from the set object:

```
>>> a = {1,2,3,4,5,6}
>>> a
{1, 2, 3, 4, 5, 6}
>>> a.clear()
>>> a
set()
```

> **Note**
>
> Mathematics has a section called *Set Theory*, which is dedicated to the study of sets and their properties. You can read more about the mathematical aspects of sets at http://www.math-only-math.com/sets.html.

Set Operations

In this section, we will look at all of the different operations we can perform on a set. Let's begin.

Union

As we stated earlier, a union between sets is the set of all items/elements in both sets.

A union can be represented by the following Venn diagram:

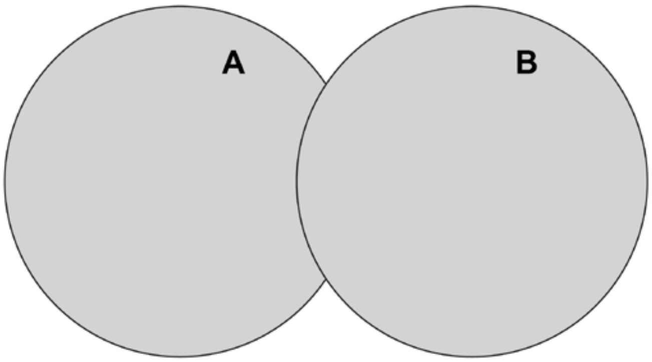


Figure 6.1: Union of sets A and B

For example, if set A = {1,2,3,4,5,6} and set B = {1,2,3,7,8,9,10}, then A *u* B will be {1,2,3,4,5,6,7,8,9,10}.

To achieve union between sets in Python, we can use the **union** method, which is defined on set objects:

```
>>> a = {1,2,3,4,5,6}
>>> b = {1,2,3,7,8,9,10}
>>> a.union(b)
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
>>> b.union(a)
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Another way to achieve union between sets in Python is to use the **|** operator:

```
>>> a = {1,2,3,4,5,6}
>>> b = {1,2,3,7,8,9,10}
>>> a | b
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Intersection

An intersection of sets is the set of all items that appear in all of the sets, that is, what they have in common. An intersection can be represented by the following Venn diagram, with the purple area being the intersection:

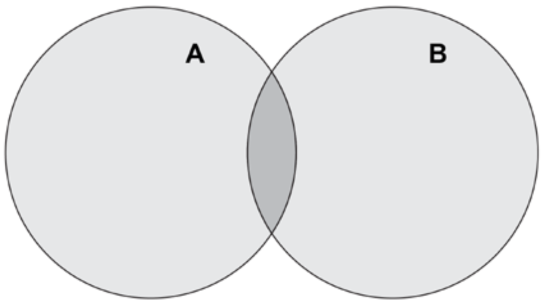


Figure 6.2: Intersection of sets A and B

As in our previous example, if set A = {1,2,3,4,5,6} and set B = {1,2,3,7,8,9,10}, then A *u* B will be {1,2,3}.

To find the intersection between sets, we can use the **`intersection`** method, which is defined on set objects:

```
>>> a = {1,2,3,4,5,6}
>>> b = {1,2,3,7,8,9,10}
>>> a.intersection(b)
{1, 2, 3}
>>> b.intersection(a)
{1, 2, 3}
```

To find the intersection between sets, you can also use the **`&`** operator:

```
>>> a = {1,2,3,4,5,6}
>>> b = {1,2,3,7,8,9,10}
>>> a & b
{1, 2, 3}
```

Difference

The difference between two sets is basically what is in one set and not in the other.

If set A = {1,2,3,4,5,6} and set B = {1,2,3,7,8,9,10}, A – B will be the set of items that are only in A, that is, {4,5,6}. Similarly, B – A will be the set {7,8,9,10}.

The following diagram illustrates A - B:

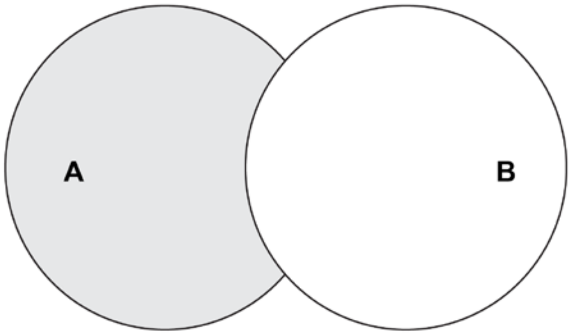


Figure 6.3: Difference of sets A and B (A - B)

Programmatically, in Python, we can use the **-** operator or the **`difference()`** method:

```
>>> a = {1,2,3,4,5,6}
>>> b = {1,2,3,7,8,9,10}
>>> a - b
{4, 5, 6}
>>> b - a
{8, 9, 10, 7}
>>> a.difference(b)
{4, 5, 6}
>>> b.difference(a)
{8, 9, 10, 7}
```

You can also get the symmetric difference between sets, which is the set of everything that is not in the intersection of the sets:

```
>>> a = {1,2,3,4,5,6}
>>> b = {1,2,3,7,8,9,10}
>>> a.symmetric_difference(b)
{4, 5, 6, 7, 8, 9, 10}
```

Subsets

The **`issubset()`** method can be used to check whether all of one set's elements exist in another set (that is, whether the set is a subset of another):

```
>>> a = {1,2,3,4,5,6,7,8,9,10}
>>> b = {5,2,10}
>>> a.issubset(b)
False
>>> b.issubset(a)
True
```

In this example, all of the elements in **`b`** are a small part of what is in **`a`**. Therefore, **`b`** is a subset of **`a`**. We call **`a`** a superset of **`b`**:

```
>>> a.issuperset(b)
True
```

Equality

You can check whether the two sets are equivalent by using the **==** operator, and whether they are not equivalent by using the **!=** operator.

The following is an example:

```
>>> a = {1,2,3}
>>> b = a.copy()
>>> c = {"money", "fame"}
>>> a == b
True
>>> a == c
False
>>> c != a
True
```

> **Note**
>
> The **copy()** method, as used on sets, produces a shallow copy of a set, much like the dictionary's **copy** method. A shallow copy means that only references to values are copied, not the values themselves.

Update Methods

You can update a set with values from the results of set operations by using the special update operations defined on the set.

These methods are as follows:

- **difference_update()**: This method removes all of the values of the other set from the set it is called on:

  ```
  >>> a = {1,2,3}
  >>> b = {3,4,5}
  >>> a - b
  {1, 2}
  >>> a.difference_update(b)
  >>> a
  {1, 2}
  ```

- **`intersection_update()`**: This method updates the set it is called on with the intersection of itself and another set that is passed as an argument:

  ```
  >>> a = {1,2,3}
  >>> b = {3,4,5}
  >>> a.intersection(b)
  {3}
  >>> a.intersection_update(b)
  >>> a
  {3}
  ```

- **`symmetric_difference_update()`**: This method updates a set that it is called on with the symmetric difference between it and the set passed as an argument:

  ```
  >>> a = {1,2,3}
  >>> b = {3,4,5}
  >>> a.symmetric_difference(b)
  {1, 2, 4, 5}
  >>> a.symmetric_difference_update(b)
  >>> a
  {1, 2, 4, 5}
  ```

Frozen Sets

Frozen sets are just like sets, and they support all other set operations. However, they are immutable, and they do not support adding or removing items. Frozen sets are useful for holding items that do not need to change; for example, a set containing the names of states in the United States.

To create a frozen set, you can call the built-in **`frozenset()`** method with an iterable:

```
>>> a = frozenset([1,2,3])

>>> a

frozenset({1, 2, 3})

>>> dir(a)

['__and__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__ne__', '__new__', '__or__', '__rand__

', '__reduce__', '__reduce_ex__', '__repr__', '__ror__', '__rsub__',
'__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__
subclasshook__', '__xor__', 'copy', 'difference', 'intersection',
'isdisjoint', 'issubset', 'issuperset', 'symmetric_difference', 'union']
```

As you can see from the output of **`dir()`**, the **`add`**, **`update`**, **`pop`**, **`discard`**, and other methods that modify the structure of the frozen set, are not defined.

Activity 27: Creating Unions of Elements in a Collection

The aim of this activity is to implement an algorithm that returns the union of elements in a collection.

We will create a function called **`find_union()`**, which takes two lists and returns a list of all of the elements in both lists, with no duplicates. Do not use the built-in **`set`** function.

The steps are as follows:

1. Define a function named **`find_union()`**.
2. Using a **`for`** statement, iterate through the two lists, while finding the common elements in both lists without duplicates.

The following shows an example output:

```
>>> find_union([1, 2, 3, 4], [3, 4, 5, 6])
[1, 2, 3, 4, 5, 6]
```

> **Note**
>
> Solution for this activity can be found at page 288.

Summary

In this chapter, we covered dictionaries and their types (the default, unordered **`dict`**, and the specialized **`OrderedDict`**). We also looked at attributes defined on dictionary objects and their use cases; for example, **`update`** and **`setdefault`**. Using these attributes, we learned how to iterate through dictionaries and modify them to achieve particular goals.

We also covered sets in this chapter, which are collections of unique and unordered items. We covered operations that you can perform on sets, such as finding unions and intersections, and other specialized operations, such as finding the difference and symmetric difference. We also looked at what frozen sets are, and the potential uses for them.

In the next chapter, we will begin our journey into object-oriented programming with Python, and we will look at how Python implements OOP concepts, such as classes and inheritance.

7

Object-Oriented Programming

Learning Objectives

By the end of this chapter, you will be able to:

- Explain different OOP concepts and the importance of OOP
- Instantiate a class
- Describe how to define instance methods and pass arguments to them
- Declare class attributes and class methods
- Describe how to override methods
- Implement multiple inheritance

This lesson introduces object-oriented programming as implemented in Python. We also cover classes and methods, as well as overriding methods and inheritance.

Introduction

A programming paradigm is a style of reasoning about programming problems. Problems, in general, can often be solved in multiple ways; for example, to calculate the sum of 2 and 3, you can use a calculator, you can use your fingers, you can use a tally mark, and so on. Similarly, in programming, you can solve problems in different ways.

At the beginning of this book, we mentioned that Python is multi-paradigm, as it supports solving problems in a functional, imperative, procedural, and object-oriented way. In this chapter, we will be diving into object-oriented programming in Python.

A First Look at OOP

Object-oriented Programming (**OOP**) is a programming paradigm based on the concept of objects. Objects can be thought of as capsules of properties and procedures/ methods. In an interview with *Rolling Stone* magazine, Steve Jobs, co-founder of Apple, once explained OOP in the following way:

"Objects are like people. They're living, breathing things that have knowledge inside them about how to do things and have memory inside them so that they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction..."

Steve Jobs; Rolling Stone; June 16, 1994

An example of an object you can consider is a car. A car has multiple different attributes. It has a number of doors, a color, and a transmission type (for example, manual or automatic). A car, regardless of the type, also has specific behaviors: it can start, accelerate, decelerate, and change gears. Regardless of how these behaviors are implemented, the only thing we, the users of the car, care about, is that the aforementioned behaviors, such as acceleration, actually work.

In OOP, reasoning about data as objects allows us to abstract the actual code and think more about the attributes of the data and the operations around the data. OOP offers the following advantages:

- It makes code reusable.
- It makes it easier to design software as you can model it in terms of real-world objects.
- It makes it easier to test, debug, and maintain.
- The data is secure due to abstraction and data hiding.

With the benefits it confers, OOP is a powerful tool in a programmer's tool box.
In the next section, we'll be looking at how OOP is used in Python.

OOP in Python

Classes are a fundamental building block of object-oriented programming. They can be likened to blueprints for an object, as they define what properties and methods/behaviors an object should have.

For example, when building a house, you'd follow a blueprint that tells you things such as how many rooms the house has, where the rooms are positioned relative to one another, or how the plumbing and electrical circuitry is laid out. In OOP, this building blueprint would be the class, while the house would be the instance/object.

In the earlier lessons, we mentioned that everything in Python is an object. Every data type and data structure you've encountered thus far, from lists and strings to integers, functions, and others, are objects. This is why when we run the **`type`** function on any object, it will have the following output:

```
>>> type([1, 2, 3])
<class 'list'>
>>> type("foobar")
<class 'str'>
>>> type({"a": 1, "b": 2})
<class 'dict'>
>>> def func(): return True
...
>>> type(func)
<class 'function'>
>>>
```

You'll note that calling the **`type`** function on each object prints out that it is an instance of a specific class. Lists are instances of the **`list`** class, strings are instances of the **`str`** class, dictionaries are instances of the **`dict`** class, and so on and so forth.

Each class is a blueprint that defines what behaviors and attributes objects will contain and how they'll behave; for example, all of the lists that you create will have the **`lists.append()`** method, which allows you to add elements to the list.

Here, we are creating an instance of the **list** class and printing out the **append** and **remove** methods. It tells us that they are methods of the list object we've instantiated:

```
>>> l = [1, 2, 3, 4, 5]  # create a list object
>>> print(l.append)
<built-in method append of list object at 0x10dd36a08>
>>> print(l.remove)
<built-in method remove of list object at 0x10dd36a08>
>>>
```

> **Note**
>
> In this chapter, we will use the terms instance and object synonymously.

Defining a Class in Python

In our example, we'll be creating the blueprint for a person. Compared to most languages, the syntax for defining a simple class is very minimal in Python.

Exercise 31: Creating a Class

In this exercise, we will create our first class, called **Person**. The steps are as follows:

1. Declare the class using the Python keyword **class**, followed by the class name **Person**. In the block, we have the Python keyword **pass**, which is used as a placeholder for where the rest of our class definition will go:

```
>>> class Person:
...     pass
...
>>>
```

2. Run the **type** function on this class we've created; it will yield the type **type**:

```
>>> type(Person)
<class 'type'>
>>>
```

This is a bit confusing, but what this means is just as there are data structures of type **`list`** or **`dict`**, we've also, in a sense, extended the Python language to include a new kind of data structure called **`Person`**. In Python, a **`class`** and a **`type`** are synonymous. This **`Person`** structure can encapsulate different attributes and methods that will be specific to that object. We'll look at this in more depth further down the line.

Instantiating an Object

Having a blueprint for building something is a great first step. However, blueprints aren't very useful if you can't build what they describe. Instantiating an object of a class is the act of building what the blueprint/class describes.

Exercise 32: Instantiating a Person Object

From the **`Person`** class we've defined, we'll instantiate a **`Person`** object. The steps are as follows:

1. Create a **`Person`** object and assign it to the **`jack`** variable:

   ```
   >>> jack = Person()
   ```

2. Create another object and assign it to the **`jill`** variable:

   ```
   >>> jill = Person()
   ```

3. Make a comparison between **`jack`** and **`jill`** to check whether they are different objects:

   ```
   >>> jack is jill
   False
   ```

   You will find that they are. This is because whenever we instantiate an object, it creates a brand-new object.

4. Assign **`jack2`** to **`jack`**:

   ```
   >>> jack2 = jack
   >>> jack2 is jack
   True
   ```

   Assigning another variable to **`jack`** simply points it to whatever object **`jack`** is pointing to, and so they are the same object and thus identical.

Adding Attributes to an Object

An attribute is a specific characteristic of an object.

In Python, you can add attributes dynamically to an already instantiated object by writing the name of the object followed by a dot (`.`) and the name of the attribute you want to add, and assigning it to a value:

```
>>> person1 = Person()
>>> person1.name = "Gol D. Roger"
```

However, setting attributes in this manner is a bad practice, since it leads to hard-to-read code that's hard to debug. We'll see the appropriate way of setting attributes in the next section.

You can get the value of an attribute by using a similar syntax:

```
>>> person1.name
'Gol D. Roger'
>>>
```

Every object in Python comes with built-in attributes, such as **`__dict__`**, which is a dictionary that holds all of the attributes of the object:

```
>>> person1 = Person()
>>> person1.__dict__
{}
>>> person1.name = "Gol D. Roger"
>>> person1.age = 53
>>> person1.height_in_cm = 180
>>> person.__dict__
{'age': 53, 'height_in_cm': 180, 'name': 'Gol D. Roger'}
>>> print(person1.name, person1.age, person1.height_in_cm)
Gol D. Roger 53 180
>>>
```

The __init__ Method

The appropriate way to add attributes to an object is by defining them in the object's constructor method. A constructor method resides in the class and is called to create an object. It often takes arguments that are used in setting attributes of that instantiated object.

In Python, the constructor method for an object is named **`__init__()`**. As its name suggests, it is called when initializing an object of a class. Because of this, you can use it to pass the initial attributes you want your object to be constructed with.

The **`hasattr()`** function checks whether an object has a specific attribute or method. When we call the **`hasattr()`** function on the **`Person`** class to check whether it has an **`__init__`** method, it returns **`True`**. This applies for all instances of the class, too:

```
>>> hasattr(Person, '__init__')
True
>>> person1 = Person()
>>> hasattr(person1, '__init__')
True
>>>
```

This method is here because it is inherited. We'll be taking a closer look at what inheritance is in later in this chapter.

We can define this constructor method in our class just like a function and specify attributes that will need to be passed in when instantiating an object:

```
>>> class Person:
...     def __init__(self, name):
...         self.name = name
...
>>>
```

Earlier on, we likened classes to blueprints for objects. In the preceding example, we're adding more details to that blueprint, and stating that every **`Person`** object that will be created should have a **`name`** attribute.

We have the arguments **self** and **name** in the **__init__** method signature. The **name** argument refers to the person's name, while **self** refers to the object we're currently in the process of creating.

Remember, the **__init__** method is called when instantiating objects of the **Person** class, and since every person has a different name, we need to be able to assign different values to different instances. Therefore, we attach our current object's **name** attribute in the line **self.name = name**.

Let's test this out.

Exercise 33: Adding Attributes to a Class

In this exercise, we will add attributes to our **Person** class. The steps are as follows:

1. First, instantiate an object without passing any arguments:

```
>>> person1 = Person()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 1 required positional argument: 'name'
>>>
```

   Python throws us an error since now we need to pass in a **name** argument when instantiating a **Person** object. This argument is passed to the **__init__** method when the object is being instantiated.

2. Instantiate a **Person** object, passing an argument for **name**:

```
>>> person1 = Person("Bon Clay")
```

3. Now try to access the attribute from our instance:

```
>>> person1.name
'Bon Clay'
>>>
```

4. Redefine the **Person** class so that it is defining more attributes that instances should be initialized with, for example, name, age, and height in centimeters:

```
>>> class Person:
...     def __init__(self, name, age, height_in_cm):
...         self.name = name
...         self.age = age
...         self.height_in_cm = height_in_cm
...
>>>
```

5. Now, when instantiating a **Person** object, we'll need to pass in the three arguments: **name**, **age**, and **height_in_cm**. Pass in the three values, as shown here:

   ```
   >>> person1 = Person("Cubert", 62, 180)
   >>> print(person1.name, person1.age, person1.height_in_cm)
   Cubert 62 180
   >>>
   ```

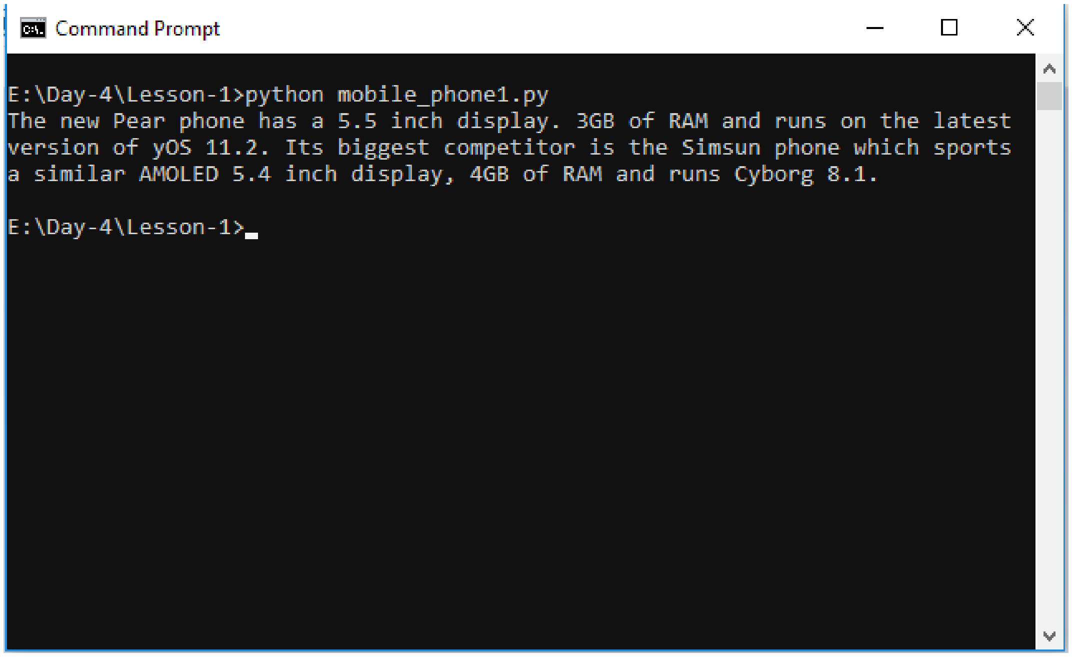
Activity 28: Defining a Class and Objects

Suppose you are a backend developer for a tech news platform. You have been asked to design a templating system for their news articles. To do this, you will need to run some proof of concepts.

Define the **MobilePhone** class in a file named **mobile_phone1.py** so that the following code runs without error:

```
# Class definition goes here


pearphone = MobilePhone(5.5, "3GB", "yOS 11.2")

simsun = MobilePhone(5.4, "4GB", "Cyborg 8.1")


print(f"The new Pear phone has a {pearphone.display_size}"

      f" inch display. {pearphone.ram} of RAM and runs on "

      f"the latest version of {pearphone.os}. Its biggest competitor is "

      f"the Simsun phone which sports a similar AMOLED {simsun.display_size}
"

      f"inch display, {simsun.ram} of RAM and runs {simsun.os}."

      )
```

After defining the class, running the preceding code should yield the following output:



```
E:\Day-4\Lesson-1>python mobile_phone1.py
The new Pear phone has a 5.5 inch display. 3GB of RAM and runs on the latest
version of yOS 11.2. Its biggest competitor is the Simsun phone which sports
a similar AMOLED 5.4 inch display, 4GB of RAM and runs Cyborg 8.1.

E:\Day-4\Lesson-1>
```

Figure 7.1: Output of running the mobile_phone1.py script

> **Note**
>
> Solution for this activity can be found at page 288.

Methods in a Class

In this topic, we will look at class methods in detail.

Defining Methods in a Class

So far, we've seen how to add attributes to an object. As we mentioned earlier, objects are also comprised of behaviors known as methods. Now we will take a look at how to add our own methods to classes.

Exercise 34: Creating a Method for our Class

We'll rewrite our original **Person** class to include a **speak()** method. The steps are as follows:

1. Create a **speak()** method in our **Person** class, as follows:

```
class Person:
    def __init__(self, name, age, height_in_cm):
        self.name = name
        self.age = age
        self.height_in_cm = height_in_cm

    def speak(self):
        print("Hello!")
```

   The syntax for defining an instance method is familiar. We pass the argument **self** which, as in the **__init__** method, refers to the current object at hand. Passing **self** will allow us to get or set the object's attributes inside our function. It is always the first argument of an instance method.

2. Instantiate an object and call the method we've defined:

```
>>> adam = Person("Adam", 47, 193)
>>> adam.speak()
Hello!
>>>
```

3. Access instance attributes and use them inside our method by using **self**, as follows:

```
class Person:
    def __init__(self, name, age, height_in_cm):
        self.name = name
        self.age = age
        self.height_in_cm = height_in_cm

    def speak(self):
        print(f"Hello! My name is {self.name}. I am {self.age} years
old.")
>>> adam = Person("adam", 47, 193)
>>> lovelace = Person("Lovelace", 24, 178)
>>> lucre = Person("Lucre", 13, 154)
>>> adam.speak()
Hello! My name is Adam. I am 47 years old.
>>> lovelace.speak()
```

```
Hello! My name is Lovelace. I am 24 years old.
>>> lucre.speak()
Hello! My name is Lucre. I am 13 years old.
>>>
```

As you can see, the output is dependent on the object we're calling the method on.

Passing Arguments to Instance Methods

Just as with normal functions, you can pass arguments to methods in a class. Let's now learn how to pass arguments to instance methods.

Exercise 35: Passing Arguments to Instance Methods

In this exercise, we'll create a new method called **`greet()`** that takes in an argument, **`person`**, which is a **`Person`** object.

1. Define the **`greet()`** method in the **`Person`** class:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def speak(self):
        print(f"Hello! My name is {self.name}. I am {self.age} years
old.")

    def greet(self, person):
        print(f"Hi {person.name}")
```

> **Note**
>
> We do not have to specify the method return type as you would in statically typed languages such as Java.

2. Instantiate two new **`Person`** objects and call the **`greet()`** method to test this out:

```
>>> joe = Person("Josef", 31)
>>> gabby = Person("Gabriela", 32)
>>> joe.greet(gabby)
Hi Gabriela
>>>
```

3. Add more logic to the method that checks for the person's name, and print out a different message if the person is named **`Rogers`**:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def speak(self):
        print(f"Hello! My name is {self.name}. I am {self.age} years
old.")

    def greet(self, person):
        if person.name == "Rogers":
            print("Hey neighbour!")
        else:
            print(f"Hi {person.name}")
```

4. Test out the new implementation of the **`greet`** method:

```
>>> joe = Person("Josef", 31)
>>> john = Person("John", 5)
>>> rogers = Person("Rogers", 46)
>>> john.greet(rogers)
Hey neighbour!
>>> john.greet(joe)
Hi Josef
```

Exercise 36: Setting Instance Attributes within Instance Methods

We'll create a **`birthday()`** method that increments the person's age.

1. First, implement the **`birthday()`** method, which takes the age and increments it by 1:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def speak(self):
        print(f"Hello! My name is {self.name}. I am {self.age} years 
old.")

    def birthday(self):
        self.age += 1
```

2. Create a person instance and check the age:

```
>>> diana = Person("Diana", 28)
>>> diana.age
28
```

3. Call the **`birthday`** method and check the age again:

```
>>> diana.birthday()
>>> diana.age
29
>>>
```

Congratulations, you can now define and use classes. You can add methods and attributes to them, as well as instantiate objects and use them. While there's more to learn, you have the necessary tools to build basic object-oriented programs.

Activity 29: Defining Methods in a Class

You are part of a team building a program to help children learn math. Currently, you're building a module on shapes, more specifically, calculating the circumference and area of circles.

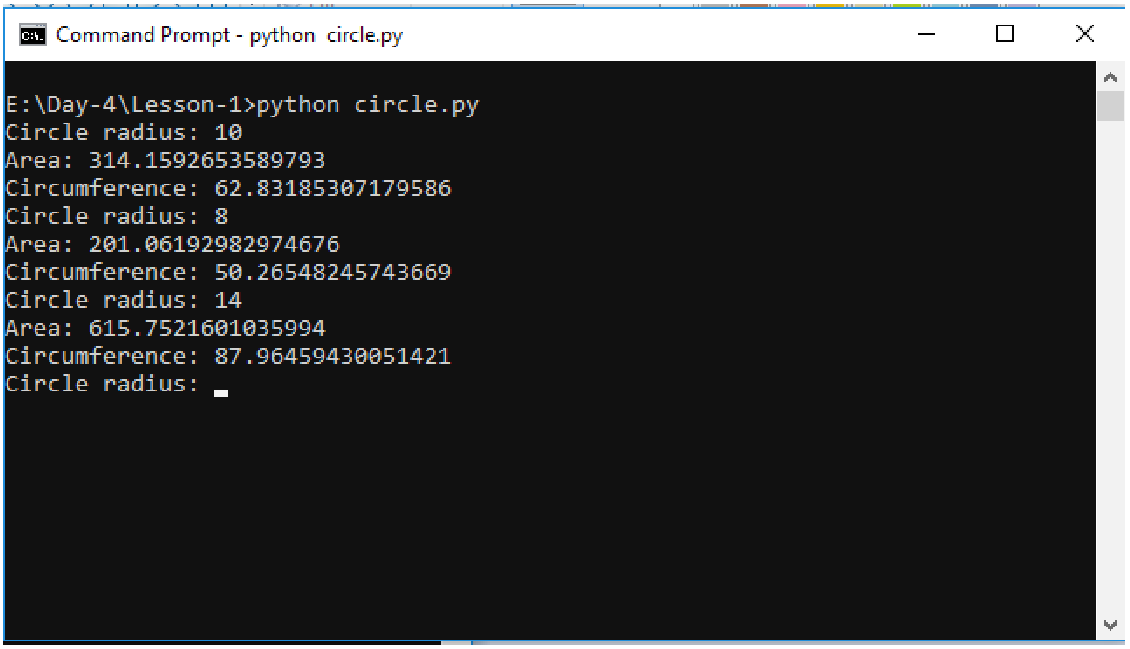
The formula for calculating the circumference of a circle is $2*\pi*r$. The formula for calculating the area of a circle is $\pi*r*r$.

Write a Python class named **`Circle`**, constructed by a radius and two methods, which will calculate the circumference and the area of a circle. The script should ask for the user's input for the radius, create a **`Circle`** object, and print out its area and circumference. It should ask for input again after it prints the area and circumference each time. Our aim here is to practice defining methods in a class.

The steps are as follows:

1. Create a file named **`circle.py`**.
2. Define the **`Circle`** class and the **`Circle`** constructor method.
3. Create the **`area`** calculation method, which returns the circle's area.
4. Create the **`circumference`** method, which returns the circle's circumference.
5. After the class definition, add the code that requests for user input for the radius.
6. We'll create a **`while`** loop so that the request for user input runs multiple times. In the **`while`** loop, we'll request the user input, change the **`circle`** object's radius, and then print out the area and circumference.

7. Once you have saved the script, you can run it by using **`python circle.py`**. The script will ask for input, calculate the area and circumference, print that out, and ask for your input again. The output should look as follows:



```
E:\Day-4\Lesson-1>python circle.py
Circle radius: 10
Area: 314.1592653589793
Circumference: 62.83185307179586
Circle radius: 8
Area: 201.06192982974676
Circumference: 50.26548245743669
Circle radius: 14
Area: 615.7521601035994
Circumference: 87.96459430051421
Circle radius: _
```

Figure 7.2: Output of running the circle.py script

> **Note**
>
> Solution for this activity can be found at page 288.

Class Versus Instance Attributes

In the previous section, we had an introduction to classes and attributes. The attributes we've seen defined up until this point are **instance attributes**. This means that they are bound to a specific instance. Initializing an object with specific attributes applies/binds those attributes to only that object, but not to any other object initialized from that class.

Exercise 37: Declaring a Class with Instance Attributes

In this exercise, we'll declare a **WebBrowser** class that has the attributes for history, the current page, and a flag that shows whether it's incognito or not. It can be initialized with a page.

> **Note**
>
> The attributes that we will declare inside the constructor will be added as instance attributes The binding of the attributes to the instance happens in the **__init__** method, where we add attributes to **self**.

1. Define the **WebBrowser** class as follows:

```
class WebBrowser:
    def __init__(self, page):
        self.history = [page]
        self.current_page = page
        self.is_incognito = False
```

2. Then, initialize the objects from the class:

```
>>> firefox = WebBrowser("google.com")
>>> chrome = WebBrowser("facebook.com")
>>>
```

3. Every **WebBrowser** instance will have a different **current_page** attribute. This happens because these attributes are bound to the instance and not to the class; they are instance attributes. Check this by getting the **current_page** attribute on different **WebBrowser** instances:

```
>>> firefox.current_page
'google.com'
>>> chrome.current_page
'facebook.com'
>>>
```

Class Attributes

We can also define attributes at the class level. **Class attributes** are bound to the class itself and are shared by all instances as opposed to being bound to each instance.

Exercise 38: Extending our Class with Class Attributes

In this exercise, we'll add a class attribute to our **`WebBrowser`** class. The syntax for this is just like defining a variable. You simply define it in the class body. The steps are as follows:

1. Add the **`connected`** attribute to our class. This is a Boolean showing whether the web browser has an active internet connection:

```
class WebBrowser:
    connected = True

    def __init__(self, page):
        self.history = [page]
        self.current_page = page
        self.is_incognito = False
```

2. Then, instantiate a **`WebBrowser`** object. We can see that the **`connected`** attribute is **`True`** for all instances:

```
>>> firefox = WebBrowser("google.com")
>>> iceweasel = WebBrowser("facebook.com")
>>> firefox.connected
True
>>> iceweasel.connected
True
>>>
```

3. Since a class attribute is bound to the class and not the instance, we can access class attributes via the class itself. Do this as follows:

```
>>> WebBrowser.connected
True
>>>
```

4. Print out our instances' **`__dict__`** attributes; we'll see that they do not have the **`connected`** attribute:

```
>>> iceweasel.__dict__
{'history': ['facebook.com'], 'current_page': 'facebook.com', 'is_
incognito': False}
>>> firefox.__dict__
{'history': ['google.com'], 'current_page': 'google.com', 'is_incognito':
False}
>>>
```

5. Why, then, don't we get an **`AttributeError`** when we try to retrieve this attribute? This is because when we access a class attribute from an instance, it retrieves it from the class itself. Here, we can see that the **`WebBrowser`** class's **`__dict__`** contains the **`connected`** attribute:

```
>>> WebBrowser.__dict__
mappingproxy({'__module__': '__main__', 'connected': True, '__init__':
<function WebBrowser.__init__ at 0x10cc6ad08>, '__dict__': <attribute '__
dict__' of 'WebBrowser' objects>, '__weakref__': <attribute '__weakref__'
of 'WebBrowser' objects>, '__doc__': None})
>>>
```

> **Note**
>
> Since instances retrieve the attribute from the class, when we change this class attribute through the class, it'll reflect on all existing instances.

6. Therefore, we need to be cautious when changing a class attribute through an instance because doing so will create an instance attribute and will no longer retrieve the attribute from the class. Check this, as follows:

```
>>> firefox.connected = False
>>>
```

7. Print out the **`__dict__`** attribute of the object; we'll see that it now has a new instance attribute, **`connected`**:

```
>>> firefox.__dict__
{'history': ['google.com'], 'current_page': 'google.com', 'is_incognito':
False, 'connected': False}
>>>
```

8. This means that when we try to get the **connected** attribute, it will no longer try retrieving it from the class, but will instead retrieve the attribute bound to the object. Despite this change we've made, the **WebBrowser** class attribute remains the same. Check this, as follows:

```
>>> WebBrowser.connected
True
>>>
```

Exercise 39: Implementing a Counter for Instances of a Class

Our aim here is to thoroughly understand class attributes. In this exercise, we're going to create a counter that will be incremented each time a new **WebBrowser** object is instantiated:

1. Add the class attribute **number_of_web_browsers**, which will serve as the counter and will start at 0:

```
class WebBrowser:
    number_of_web_browsers = 0
    connected = True

    def __init__(self, page):
        self.history = [page]
        self.current_page = page
        self.is_incognito = False
```

2. Modify the constructor to increment the counter each time a new instance is created by adding the line **WebBrowser.number_of_web_browsers += 1**. This increments the **number_of_web_browsers** attribute of our class by 1 and will be called each time a new instance is initialized:

```
class WebBrowser:
    number_of_web_browsers = 0
    connected = True

    def __init__(self, page):
        self.history = [page]
        self.current_page = page
        self.is_incognito = False
        WebBrowser.number_of_web_browsers += 1
```

Let's test it out:

3. First, check that the **number_of_web_browsers** counter is at 0:

```
>>> WebBrowser.number_of_web_browsers
0
>>>
```

4. Next, instantiate a new object and check the counter:

```
>>> opera = WebBrowser("opera.com")
>>> WebBrowser.number_of_web_browsers
1
>>>
```

5. The counter increments with every other instance we create. Check this, as follows:

```
>>> edge = WebBrowser("microsoft.com")
>>> WebBrowser.number_of_web_browsers
2
>>>
```

Besides the use cases we've seen, class attributes should be used when you have variables that are common to all instances, such as constants for the class.

Activity 30: Creating Class Attributes

Suppose you are designing a piece of software for an elevator company. A part of the software involves a safety mechanism to prevent the elevator from being used when it's filled past its capacity.

Define a class called **Elevator**, which will have a maximum occupancy of 8. The elevator should be initialized with the number of occupants. If the number of occupants exceeds the limit during initialization, it should print out a message indicating that the limit has been exceeded and only initialize how many occupants should step off the elevator.

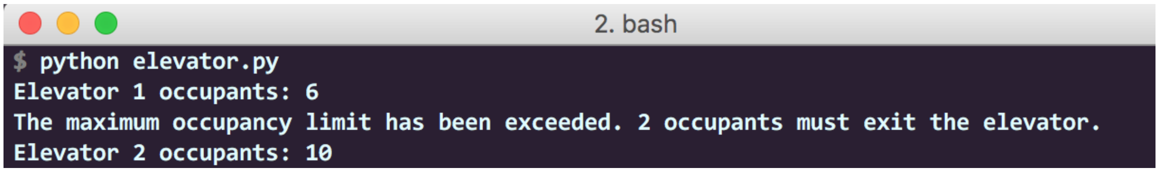
The steps are as follows:

1. Create a file named **elevator.py**.
2. Declare the **Elevator** class by adding an occupancy limit class attribute.
3. Add the initializer, which will check whether the occupancy limit will be exceeded, and print a message indicating how many people should alight.

4. Finally, create a few instances after the class declaration to test the class out:

```
elevator1 = Elevator(6)
print("Elevator 1 occupants:", elevator1.occupants)
elevator2 = Elevator(10)
print("Elevator 2 occupants:", elevator2.occupants)
```

We can then test out our script by running **`python elevator.py`** in the terminal. The output should look like this:



```
$ python elevator.py
Elevator 1 occupants: 6
The maximum occupancy limit has been exceeded. 2 occupants must exit the elevator.
Elevator 2 occupants: 10
```

Figure 7.3: Output of running the elevator.py script

> **Note**
>
> Solution for this activity can be found at page 289.

Class Versus Instance Methods

In this section, we will take a brief look at instance methods and cover class methods in detail.

Exercise 40: Creating Instance Methods

In this exercise, we will implement the **`navigate()`** and **`clear_history()`** methods for the **`WebBrowser`** class we defined in the previous section:

1. Add the **`navigate()`** method to the **`WebBrowser`** class:

```
class WebBrowser:
    def __init__(self, page):
        self.history = [page]
        self.current_page = page
        self.is_incognito = False

    def navigate(self, new_page):
        self.current_page = new_page
        if not self.is_incognito:
            self.history.append(new_page)
```

Any call to **navigate** will the set the browser's current page to the **new_page** argument and then add it to the history if we're not in incognito mode (incognito mode in browsers prevents browsing history from being recorded).

2. Calling **navigate()** on an instance should change **current_page**:

```
>>> vivaldi = WebBrowser("gocampaign.org")
>>> vivaldi.current_page
'gocampaign.org'
>>> vivaldi.navigate("reddit.com")
>>> vivaldi.current_page
'reddit.com'
>>> vivaldi.history
['gocampaign.org', 'reddit.com']
>>>
```

3. Create the **clear_history** method, which will delete the browser's history:

```
class WebBrowser:
    def __init__(self, page):
        self.history = [page]
        self.current_page = page
        self.is_incognito = False

    def navigate(self, new_page):
        self.current_page = new_page
        if not self.is_incognito:
            self.history.append(new_page)

    def clear_history(self):
        self.history[:-1] = []
```

The **clear_history** method removes everything from the history list up to the last element, which is our current page. This leaves only our current page on the list.

4. Add to the browser history by navigating to a couple of pages and then call the **clear_history()** method to see whether it works:

```
>>> chrome = WebBrowser("example.net")
>>> chrome.navigate("example2.net")
>>> chrome.navigate("example3.net")
>>> chrome.history
['example.net', 'example2.net', 'example3.net']
>>> chrome.current_page
'example3.net'
```

```
>>> chrome.clear_history()
>>> chrome.history
['example3.net']
>>>
```

We mentioned in the previous chapter that instance methods must receive **self** as the first argument. This is because **self** refers to the current instance in the context. Despite not passing it as an argument when calling instance methods, our method calls execute without any error. How does this work?

Python passes the **self** argument implicitly. In the preceding code snippet, when we call **chrome.clear_history()**, Python essentially passes **chrome** in as an argument to the method; therefore, we don't need to explicitly pass in a value for **self**.

Such a method, one that takes an instance (**self**) as the first parameter, is referred to as a **bound method**. They are bound to that specific instance when it is created. In a sense, it can be thought of as every instance of a class having its own copy of the method that was defined in the class. If we print out the instance method of any instance, we'll see the following output:

```
>>> chrome.navigate

<bound method WebBrowser.navigate of <__main__.WebBrowser object at
0x107a9a390>>

>>> opera = WebBrowser("foobar.com")

>>> opera.navigate

<bound method WebBrowser.navigate of <__main__.WebBrowser object at
0x107a9a400>>

>>>
```

The output for **chrome.navigate** tells us that it is a bound method of an object in the memory location **0x107a9a390**. The output of **opera.navigate** tells us that it is a bound method of an object at a different object at memory location **0x107a9a400**. This shows us that the two instance methods are tied/bound to different objects.

Class Methods

This brings us to class methods. **Class methods** differ from instance methods in that they are bound to the class itself and not the instance. As such, they don't have access to instance attributes. Additionally, they can be called through the class itself and don't require the creation of an instance of the class.

Regarding instance methods, we saw that the first parameter is always an instance; with class methods, the first parameter is always the class itself, as we'll see in our examples.

One common use case for class methods is when you're making **factory methods**. A factory method is one that returns objects. They can be used for returning objects of a different type or with different attributes. Let's add a class method called **`with_incognito()`** to our **`WebBrowser`** class that initializes a web browser object in incognito mode:

```
class WebBrowser:
    def __init__(self, page):
        self.history = [page]
        self.current_page = page
        self.is_incognito = False

    def navigate(self, new_page):
        self.current_page = new_page
        if not self.is_incognito:
            self.history.append(new_page)

    def clear_history(self):
        self.history[:-1] = []

    @classmethod
    def with_incognito(cls, page):
        instance = cls(page)
        instance.is_incognito = True
        instance.history = []
        return instance
```

Our function definition begins with a peculiar piece of syntax, **`@classmethod`**. We won't go into the details on it, but all we need to know right now is that it tells Python to add the function below it as a class method. On the next line, we declare our function, which takes two arguments, **`cls`** and **`page`**. The **`cls`** argument refers to our **`WebBrowser`** class in this context. All class methods must have the class as the first argument. The name can be **`cls`**, which is the convention, or anything else, whether it be **`class_`** or **`foobar`**. All that matters is that the first argument of the class method is *reserved*.

We pass the **`page`** argument during the instantiation of our **`WebBrowser`** object. In the function's body, we instantiate an object which we assign the name **`instance`**. We then change the incognito value of that instance to **`True`** and clear the history list. Finally, we return the instance we've created.

Exercise 41: Testing our Factory Method

In this exercise, we'll try out our factory method:

1. Print out the class method. The output tells us that it is a bound method of the **`WebBrowser`** class. This illustrates what we said earlier regarding class methods and how they are bound to the class itself:

```
>>> WebBrowser.with_incognito
<bound method WebBrowser.with_incognito of <class '__main__.WebBrowser'>>
```

2. Create a **`WebBrowser`** instance that starts off in incognito mode. Note that we call **`with_incognito`** through the class. Despite not passing the **`cls`** argument in this call, Python implicitly passes the **`WebBrowser`** class to the function. All we need to pass in is the **`page`** parameter.

```
>>> chrome = WebBrowser.with_incognito("shady-website.com")
>>> chrome.is_incognito
True
```

3. Print out the current page of our instance to check whether it was set:

```
>>> chrome.current_page
'shady-website.com'
```

4. Confirm that the history was not tracked:

```
>>> chrome.history
[]
>>>
```

5. Additionally, you can call class methods through instances for the same effect.

```
>>> opera = WebBrowser("foobar.com")
>>> netscape = opera.with_incognito("secret.net")
>>> netscape.current_page
'secret.net'
>>> netscape.is_incognito
True
>>>
```

> **Caution**
>
> You should only call class methods through an instance in situations where it won't raise any confusion as to what kind of method it is you're calling (instance or class method).

Exercise 42: Accessing Class Attributes from within Class Methods

Class methods also have access to class attributes. They can get and set class attributes. Most browsers today have a geolocation API. We will simulate this functionality in our class.

In this exercise, we will create a **geo_coordinates** attribute in the **WebBrowser** class that holds the current latitude and longitude. We will also add a class method called **change_geo_coordinates()** that will change the coordinates when called:

1. Add the **geo_coordinates** class attribute and change the **geo_coordinates()** class method, like so:

```
class WebBrowser:
    geo_coordinates = {"lat": -4.764813, "lng": 16.131331 }
    def __init__(self, page):
        self.history = [page]
        self.current_page = page
        self.is_incognito = False

    def navigate(self, new_page):
        self.current_page = new_page
        if not self.is_incognito:
            self.history.append(new_page)
    def clear_history(self):
        self.history[:-1] = []

    @classmethod
    def with_incognito(cls, page):
        instance = cls(page)
        instance.is_incognito = True
        instance.history = []
        return instance

    @classmethod
    def change_geo_coordinates(cls, new_coordinates):
        if new_coordinates["lat"] > 90 or new_coordinates["lat"] < -90:
            print("Invalid value for latitude. Should be within the"
                  " range from -90 to 90 degrees.")
            return None
        if new_coordinates["lng"] > 180 or new_coordinates["lng"] < -180:
            print("Invalid value for longitude. Should be within the"
                  " range from -180 to 180 degrees.")
            return None
        cls.geo_coordinates = new_coordinates
```

Our class method, **change_geo_coordinates**, takes the **new_coordinates** parameter, which is a dictionary. It checks whether the latitude and longitude provided in the parameters are valid and then changes the class attribute **geo_coordinates** to reflect the new coordinates that have been provided. We can test this out.

2. Create a **WebBrowser** instance, **firefox**, and check its geocoordinates. It fetches the attribute from the class:

```
>>> firefox = WebBrowser("www.org")
>>> firefox.geo_coordinates
{'lat': -4.764813, 'lng': 16.131331}
```

3. Calling **`change_geo_coordinates`** on the class as we do in the next line changes the **`geo_coordinates`** attribute for all of the class's instances (since they fetch the attribute from the class), and hence this change reflects for the **`firefox`** instance:

```
>>> WebBrowser.change_geo_coordinates({"lat": 31, "lng": 123})
>>> firefox.geo_coordinates
{'lat': 31, 'lng': 123}
>>> WebBrowser.change_geo_coordinates({"lat": 31, "lng": 190})
Invalid value for longitude. Should be within the range from -180 to 180
degrees.
>>> WebBrowser.change_geo_coordinates({"lat": -100, "lng": 123})
Invalid value for latitude. Should be within the range from -90 to 90
degrees.
>>>
```

Encapsulation and Information Hiding

One of the key concepts of OOP is **encapsulation**. Encapsulation is the bundling of data with the methods that operate on that data. It's used to hide the internal state of an object by bundling together and providing methods that can get and set the object state through an interface. This hiding of the internal state of an object is what we refer to as **information hiding**.

With our **`WebBrowser`** class, when we called the **`navigate`** method as users, all we cared about was that it changed the current page. The class was a bundle of data and logic that gave us access to a uniform browser interface. The same is true for a real web browser. As users, we simply type in the URL and hit the *Enter* key, and it takes us to the new page. We don't care to know that the browser had to make a request to the server, wait for the response, render the resulting markup, apply styling, and download accompanying media along with it. The browser acts as a simple interface that allows us to interact with the internet. The processes behind all the steps it takes are hidden away from the users.

We use information hiding to abstract away irrelevant details about the class from users to prevent them from changing them, which would affect the functionality of our class.

In Python, information hiding is accomplished by marking attributes as **private** or **protected**:

- **private** attributes should only be used inside the class definition and shouldn't be accessed externally.
- **protected** attributes are similar to **private** ones, but can only be used in very specific contexts.

By default, all attributes in Python are **public**.

In most languages, these attribute access modifiers are denoted by the keyword **private**, **public**, or **protected**. Python, however, simply implements these in the attribute names themselves.

All Python attributes are **public** by default and need no special naming or declaration:

```
class Car:
    def __init__(self):
        self.speed = 300
        self.color = "black"
```

For **protected** attributes, we prefix the attribute name with an underscore, **_**, to show that it's **protected**:

```
class Car:
    def __init__(self):
        self._speed = 300
        self._color = "black"
```

Doing this doesn't change the class user's ability to change the attribute. It's simply a marker letting them know not to access or change the attribute from outside the class or its children. The interpreter enforces no actual restrictions to enforce this. You can still change **protected** attributes:

```
>>> car = Car()
>>> car._speed
300
>>> car._speed = 400
>>> car._speed
400
>>>
```

While it may seem that marking attribute names as **protected** is useless since it doesn't impose any restrictions, it is good practice to do it to let the users of the class know it's a **protected** attribute that is only meant to be used internally. It is up to them to follow convention and not assign or access **protected** attributes.

For **private** attributes, we prefix the attribute name with a double underscore **__**. This renders the attribute inaccessible from outside the class. The attribute can only be gotten and set from within the class:

```
class Car:
    def __init__(self):
        self.__speed = 300
        self.__color = "black"
    def change_speed(self, new_speed):
        self.__speed = new_speed
    def get_speed(self):
        return self.__speed
```

If we try accessing any of these attributes from outside the class, we'll get an error:

```
>>> car = Car()
>>> car.__speed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Car' object has no attribute '__speed'
>>>
```

To change the **private** attribute **__speed**, we need to use the defined setter method **change_speed**. Similarly, we can use the **get_speed** getter method to get the **speed** attribute from outside the class if need be:

```
>>> car.get_speed()
300
>>> car.change_speed(120)
>>> car.get_speed()
120
>>>
```

Activity 31: Creating Class Methods and Using Information Hiding

Suppose you work for an electronics company that has a new *MusicPlayer* device that it wants to release to the market. The software for this device needs to support over-the-air updates that enables users to listen to their favorite tunes painlessly.

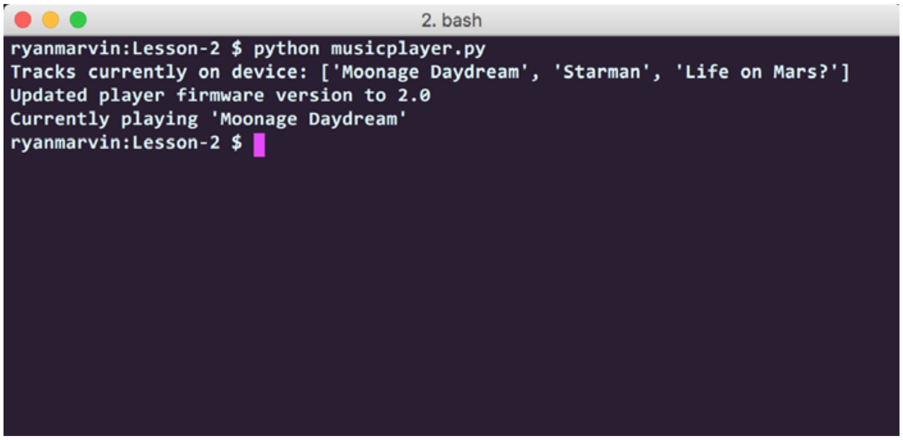
Create a class that represents a portable music player, *MusicPlayer*. The **`MusicPlayer`** class should have a **`play`** method, which sets the first track from the list of tracks as currently playing. The list of tracks should be a **`private`** attribute. Additionally, it should have a firmware version attribute and an update firmware class method that updates the firmware version.

The steps are as follows:

1. Create a file named **`musicplayer.py`**.
2. Define the **`MusicPlayer`** class by adding the firmware version class attribute.
3. Define the initializer method and pre-populate the track list with a few songs. Make sure the music track's store is **`private`**.
4. Define the **`play`** method, which sets the **`current_track`** attribute to the first item in the track's list.
5. Define the list tracks method, which returns the list of tracks in the **`MusicPlayer`**.
6. Finally, we'll add the update firmware version method, which checks for whether the new version being provided is more recent than the current firmware version before updating.
7. We can then add a few test lines and run the script:

```
player = MusicPlayer()
print("Tracks currently on device:", player.list_tracks())

MusicPlayer.update_firmware(2.0)
print("Updated player firmware version to", player.firmware_version)

player.play()
print("Currently playing", f"'{player.current_track}'")
```

We can run the script by running **`python musicplayer.py`** in the terminal. The output should look like this:

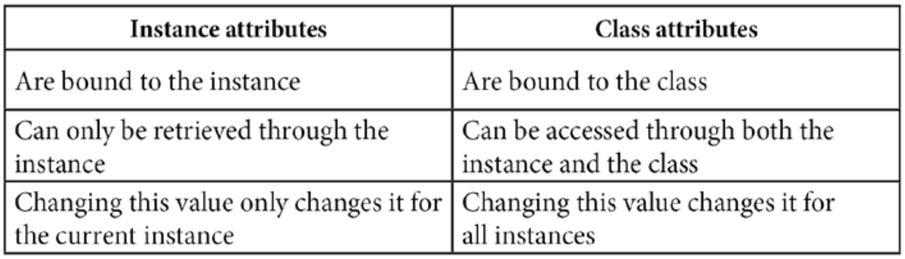


```
ryanmarvin:Lesson-2 $ python musicplayer.py
Tracks currently on device: ['Moonage Daydream', 'Starman', 'Life on Mars?']
Updated player firmware version to 2.0
Currently playing 'Moonage Daydream'
ryanmarvin:Lesson-2 $
```

Figure 7.4: Output of running the musicplayer.py script

> **Note**
>
> Solution for this activity can be found at page 290.

The following table compares instance attributes with class attributes:



Instance attributes	Class attributes
Are bound to the instance	Are bound to the class
Can only be retrieved through the instance	Can be accessed through both the instance and the class
Changing this value only changes it for the current instance	Changing this value changes it for all instances

Figure 7.5: Instance versus class attributes

The following table compares instance methods with class methods:

Instance methods	Class methods
Are bound to the instance	Are bound to the class
Can only be called through an instance	Can be called through both the instance and the class
Take the self instance as the first argument	Take the class as the first argument
Have access to both instance and class attributes	Only have access to class attributes

Figure 7.6: Instance versus class methods

Class Inheritance

A key feature of object-oriented programming is **inheritance**. Inheritance is a mechanism that allows for a class's implementation to be derived from another class's implementation. This subclass/derived/child class inherits all of the attributes and methods of the superclass/base/parent class:

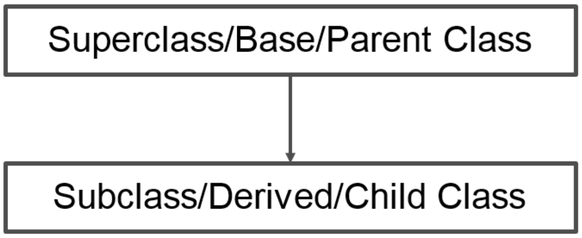


Figure 7.7: Inheritance in classes

A practical real-world example of inheritance can be thought of with big cats. Cheetahs, leopards, tigers, and lions are all cats. They all share the same properties that are common to cats such as mass, lifespan, speed, and behaviors such as making vocalizations and hunting, among others. If we were to implement a `Leopard`, `Cheetah`, or `Lion` class, we would define one `Cat` class that has all of these properties and then derive the `Leopard`, `Lion`, and `Cheetah` classes from this `Cat` class since they all share these same properties. This would be inheritance.

We use inheritance because it confers the following benefits:

- It makes our code more reusable. For example, with our **`Cat`** class example, we don't have to repeat the properties that each of the **`Lion`**, **`Cheetah`**, and **`Leopard`** classes possess; we can simply define them once in the **`Cat`** class and inherit the functionality in the derived classes. This also reduces code duplication.
- Inheritance also makes it easier to extend functionality since a method or attribute added to a base class automatically gets applied to all of its subclasses. For example, defining a **`spots`** attribute in the **`Cat`** class automatically avails cheetah and leopard subclasses with the same attribute.
- Inheritance adds flexibility to our code. Any place where a superclass instance is being used, a subclass instance can be substituted for the same effect. For example, at any place where a **`Cat`** instance would be used in our code, a **`Leopard`**, **`Lion`**, or **`Cheetah`** instance can be substituted since they're all cats.

The Python syntax for inheritance is very minimal. You define the class as usual, but then you can pass in the base class as a parameter. As we'll see later on, you can pass multiple base classes for cases where you want multiple inheritance, like so:

```
class Subclass(Superclass):
    pass
```

Exercise 43: Implementing Class Inheritance

In this exercise, we'll define the **`Cat`** class from which we'll derive our other big cats. The class will have the methods **`vocalize`** and **`print_facts`**, and the attributes **`mass`**, **`lifespan`**, and **`speed`**.

The constructor method will take the arguments **`mass`**, **`lifespan`**, and **`speed`** from which it will add the attributes **`mass_in_kg`**, **`lifespan_in_years`**, and **`speed_in_kph`** to the object.

The **vocalize** method will print out **Chuff**, a non-threatening vocalization that's common to several big cats. The **print_facts** method will print out facts about the cat:

1. Define the **Cat** class:

```
class Cat:
    def __init__(self, mass, lifespan, speed):
        self.mass_in_kg = mass
        self.lifespan_in_years = lifespan
        self.speed_in_kph = speed

    def vocalize(self):
        print("Chuff")

    def print_facts(self):
        print(f"The {type(self).__name__.lower()} "
              f"weighs {self.mass_in_kg}kg,"
              f" has a lifespan of {self.lifespan_in_years} years and "
              f"can run at a maximum speed of {self.speed_in_kph}kph.")
```

> **Note**
>
> The line **type(self).__name__** means that we want the name of the current class of the object, in this case, **Cat**. We then call **str.lower()** on the name in our example.

2. Instantiate a **cat** instance and interact with the different methods and attributes it has:

```
>>> cat = Cat(4, 18, 48)
>>> cat.vocalize()
Chuff
>>> cat.print_facts()
The cat weighs 4kg, has a lifespan of 18 years and can run at a maximum
speed of 48kph.
>>>
```

3. Create the subclasses **Leopard**, **Cheetah**, and **Lion**, which will inherit from the **Cat** class:

```
class Cat:
    def __init__(self, mass, lifespan, speed):
        self.mass_in_kg = mass
        self.lifespan_in_years = lifespan
        self.speed_in_kph = speed

    def vocalize(self):
        print("Chuff")

    def print_facts(self):
        print(f"The {type(self).__name__.lower()} "
              f"weighs {self.mass_in_kg}kg,"
              f" has a lifespan of {self.lifespan_in_years} years and "
              f"can run at a maximum speed of {self.speed_in_kph}kph.")

class Cheetah(Cat):
    pass

class Lion(Cat):
    pass

class Leopard(Cat):
    pass
```

4. Instantiate the new **Leopard**, **Cheetah**, and **Lion** classes.

   Despite not adding any methods or attributes to these new classes, if we instantiate them, we'll need to pass in the same arguments that we do when instantiating the **Cat** class. The methods and attributes our instance will have will be identical to a **Cat** class instance:

```
>>> cheetah = Cheetah(72, 12, 120)
>>> lion = Lion(190, 14, 80)
>>> leopard = Leopard(90, 17, 58)
>>> cheetah.print_facts()
```

```
The cheetah weighs 72kg, has a lifespan of 12 years and can run at a
maximum speed of 120kph.
>>> lion.print_facts()
The lion weighs 190kg, has a lifespan of 14 years and can run at a maximum
speed of 80kph.
>>> leopard.print_facts()
The leopard weighs 90kg, has a lifespan of 17 years and can run at a
maximum speed of 58kph.
>>>
```

As you can see, our subclasses have automatically inherited all of the attributes and methods of the **Cat** class. We have a slight issue on our hands, though.

5. If we call **vocalize** on our instances, they all have the same behavior:

```
>>> cheetah.vocalize()
Chuff
>>> lion.vocalize()
Chuff
>>> leopard.vocalize()
Chuff
>>>
```

In reality, cheetahs make a *chirrup*, bird-like sound, while lions and leopards roar. We can rectify this by overriding the method in our class. **Overriding** means redefining the implementation of a method defined in a superclass to add or change a subclass's functionality.

6. Override the **vocalize** method for our subclasses:

```
class Cat:
    def __init__(self, mass, lifespan, speed):
        self.mass_in_kg = mass
        self.lifespan_in_years = lifespan
        self.speed_in_kph = speed

    def vocalize(self):
        print("Chuff")

    def print_facts(self):
        print(f"The {type(self).__name__.lower()} "
                      f"weighs {self.mass_in_kg}kg,"
                      f" has a lifespan of {self.lifespan_in_years} years and "
                      f"can run at a maximum speed of {self.speed_in_kph}kph.")


class Cheetah(Cat):
    def vocalize(self):
        print("Chirrup")

class Lion(Cat):
    def vocalize(self):
        print("ROAR")


class Leopard(Cat):
    def vocalize(self):
        print("Roar")
```

7. If we call the **vocalize** method now, we should get different outputs:

```
>>> cheetah.vocalize()
Chirrup
>>> lion.vocalize()
ROAR
>>> leopard.vocalize()
Roar
>>>
```

We'll be taking a look at overriding in more depth in the next section.

Overriding __init__()

In the previous topic, we overrode the **vocalize()** method of our **Cat** base class in our **Cheetah**, **Lion**, and **Leopard** subclasses. In this topic, we'll see how to override the **__init__()** method.

A lot of big cats have a pattern in their coat; they have spots or stripes. Let's add this to our **Cheetah** subclass.

Exercise 44: Overriding the `__init__` Method to Add an Attribute

In this exercise, we'll override the **`__init__`** method and add a **`spotted_coat`** attribute:

1. Override the initializer method and add the **`spotted_coat`** attribute:

```
class Cat:
    def __init__(self, mass, lifespan, speed):
        self.mass_in_kg = mass
        self.lifespan_in_years = lifespan
        self.speed_in_kph = speed

    def vocalize(self):
        print("Chuff")

    def print_facts(self):
        print(f"The {type(self).__name__.lower()} "
              f"weighs {self.mass_in_kg}kg,"
              f" has a lifespan of {self.lifespan_in_years} years and "
              f"can run at a maximum speed of {self.speed_in_kph}kph.")

class Cheetah(Cat):
    def __init__(self, mass, lifespan, speed):
        self.spotted_coat = True

    def vocalize(self):
        print("Chirrup")
```

   Unfortunately, this overwrites the previous implementation and replaces it with our new one; so, when we initialize the **`Cheetah`** subclass, it won't add the **`mass_in_kg`**, **`lifespan_in_years`**, and **`speed_in_kph`** attributes. It will only add the **`spotted_coat`** attribute to the instance.

2. Initialize the newly modified **`Cheetah`** class. It should raise an error upon trying to access the original attributes that it had before we overrode the **`__init__()`** method:

```
>>> cheetah = Cheetah(72, 12, 120)
>>> cheetah.mass_in_kg
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Cheetah' object has no attribute 'mass_in_kg'
>>> cheetah.lifespan_in_years
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Cheetah' object has no attribute 'lifespan_in_years'
>>> cheetah.speed_in_kph
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Cheetah' object has no attribute 'speed_in_kph'
>>> cheetah.spotted_coat
True
>>>
```

What we can do is invoke the **`__init__`** method of the **`Cat`** class inside the **`Cheetah`** subclass's **`__init__`** method before adding the **`spotted_coat`** attribute. To do this, we can call **`Cat.__init__(self, mass, lifespan, speed)`**, which calls the superclass's initializer with the required arguments.

3. Call the superclass's initializer method in the **`Cheetah`** subclass initializer:

```
class Cheetah(Cat):
    def __init__(self, mass, lifespan, speed):
        Cat.__init__(self, mass, lifespan, speed)
        self.spotted_coat = True

    def vocalize(self):
        print("Chirrup")
```

However, doing this hardcodes the superclass name, and in case we need to change the name of the **`Cat`** class, we'd have to change it in multiple places. Python provides a cleaner way of doing this through the built-in **`super()`** method. We use **`super()`** to access inherited methods from a parent class that has been overwritten in the child class.

4. Call the superclass's initializer method by using the **`super()`** method:

```
class Cheetah(Cat):
    def __init__(self, mass, lifespan, speed):
        super().__init__(mass, lifespan, speed)
        self.spotted_coat = True

    def vocalize(self):
        print("Chirrup")
```

5. When we instantiate a **Cheetah** instance, we see that our **Cat** superclass implementation is preserved:

    ```
    >>> cheetah = Cheetah(72,12,120)
    >>> cheetah.print_facts()
    The cheetah weighs 72kg, has a lifespan of 12 years and can run at a
    maximum speed of 120kph.
    >>>
    ```

    At the same time, our new implementation is also added:

    ```
    >>> cheetah.spotted_coat
    True
    >>>
    ```

Commonly Overridden Methods

As you may have noticed, special methods in Python classes are always prefixed and suffixed with double underscores, for example, **__init__**. They are known as **Dunder** (**double underscore**) or **magic methods**.

Besides the **__init__** method, there are other magic methods in Python that you can override to customize your class and add custom functionality, such as changing what the output of your printed object looks like or how your classes are compared.

We will only be going over the method that defines what is output when print is called on your object, **__str__()**, and the method that's called when an object is destroyed, **__del__()**.

> **Note**
>
> Special methods in Python classes are always prefixed and suffixed with double underscores. You can find the documentation for the rest of the special methods Python provides at https://docs.python.org/3/reference/datamodel.html.

The `__str__()` Method

Every object in Python has the **`__str__()`** method by default. It is called every time **`print()`** is called on an object in Python to retrieve the string containing the readable representation of the object.

Let's replace the **`print_facts()`** method of the **`Cat`** class with this method:

```
class Cat:
    def __init__(self, mass, lifespan, speed):
        self.mass_in_kg = mass
        self.lifespan_in_years = lifespan
        self.speed_in_kph = speed

    def vocalize(self):
        print("Chuff")

    def __str__(self):
        return f"The {type(self).__name__.lower()} "\
            f"weighs {self.mass_in_kg}kg,"\
            f" has a lifespan of {self.lifespan_in_years} years and "\
            f"can run at a maximum speed of {self.speed_in_kph}kph."
```

Now, when we call **`print()`** on any **`Cat`** instance or **`Cat`** subclass instance, it should have the same result as when we were calling **`print_facts()`**:

```
>>> cheetah = Cheetah(72, 12, 120)
>>> print(cheetah)
The cheetah weighs 72kg, has a lifespan of 12 years and can run at a maximum
speed of 120kph.
>>>
```

The `__del__()` Method

The **`__del__()`** method is the destructor method. The destructor method is called whenever an object gets destroyed:

```
class Cheetah(Cat):
    def __init__(self, mass, lifespan, speed):
        super().__init__(mass, lifespan, speed)
        self.spotted_coat = True

    def vocalize(self):
        print("Chirrup")

    def __del__(self):
        print("No animals were harmed in the deletion of this instance")
```

If we call **`del`** on a **`Cheetah`** instance, it should print out that message:

```
>>> cheetah = Cheetah(72, 12, 120)
>>> del cheetah
No animals were harmed in the deletion of this instance
>>> cheetah
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cheetah' is not defined
>>>
```

Activity 32: Overriding Methods

Suppose you work for a wildlife conservation organization. You are working on creating a system to educate the general public about different animals and get them more interested in conservation.

Create a **`Tiger`** class that inherits from the **`Cat`** class and has a new coat pattern attribute. Change the behavior of instances of the **`Tiger`** class to include this coat pattern fact when they're printed.

The steps are as follows:

1. Create the **`tiger.py`** file.
2. Define the **`Cat`** class.
3. Define the **`Tiger`** class that inherits from the **`Cat`** class. Override its initializer and add a **`coat_pattern`** attribute.
4. Override the **`__str__()`** method and modify it to include mention of the coat pattern.
5. Initialize an instance of the **`Tiger`** class. Calling **`print`** on the instance should display facts about the tiger. It should look like this:

```
The tiger weighs 310kg, has a lifespan of 26 years and can run at a
maximum speed of 65kph. It also has a striped coat.
```

> **Note**
>
> Solution for this activity can be found at page 291.

Multiple Inheritance

Multiple inheritance is a feature that allows you to inherit attributes and methods from more than one class. The most common use case for multiple inheritance is for **mixins**. Mixins are classes that have methods/attributes that are meant to be used by other functions. For example, a **`Logger`** class would have a **`log()`** method that writes to a logfile, and when added to your classes as a mixin, would give them that same capability.

The following is the syntax for multiple inheritance:

```
class Subclass(Superclass1, Superclass2):
    pass
```

The subclass inherits all of the features of both superclasses.

Exercise 45: Implementing Multiple Inheritance

In the real world, lions and tigers can naturally mate to create a hybrid known as a liger or a tigon. Ligers are much larger than either lions or tigers, they are social like lions, have stripes, and, just like tigers, they like swimming. We're going to create a **`Liger`** class that inherits from both the **`Lion`** and **`Tiger`** class we're going to define.

In this exercise, we will learn how to implement multiple inheritance:

1. Define the **`Lion`** and **`Tiger`** classes. For simplicity, we'll hardcode the **`mass`**, **`lifespan`**, and **`speed`** attributes. They'll both inherit from the **`Cat`** class:

```
class Cat:
    def __init__(self, mass, lifespan, speed):
        self.mass_in_kg = mass
        self.lifespan_in_years = lifespan
        self.speed_in_kph = speed

    def vocalize(self):
        print("Chuff")

    def __str__(self):
        return f"The {type(self).__name__.lower()} "\
            f"weighs {self.mass_in_kg}kg,"\
            f" has a lifespan of {self.lifespan_in_years} years and "\
            f"can run at a maximum speed of {self.speed_in_kph}kph."

class Lion(Cat):
    def __init__(self, mass=190, lifespan=14, speed=80):
        super().__init__(mass, lifespan, speed)
        self.is_social = True

    def vocalize(self):
        print("ROAR")

class Tiger(Cat):
    def __init__(self, mass=310, lifespan=26, speed=65):
        super().__init__(mass, lifespan, speed)
        self.coat_pattern = "striped"
    def swim(self):
        print("Splash splash")

    def vocalize(self):
        print("ROAR")
```

2. Then, define the **Liger** class, which will inherit from both the **Tiger** and **Lion** classes:

```
    class Liger(Lion, Tiger):
        pass
```

3. On testing it out, we should see that the **Liger** class has inherited attributes from both the **Lion** and **Tiger** classes. The **Liger** class has both the **coat_pattern** attribute and **swim()** method of the **Tiger** class and the **is_social** attribute of the **Lion** class:

```
    >>> liger = Liger()
    >>> liger.swim()
    Splash splash
    >>> liger.is_social
    True
    >>> liger.coat_pattern
    'striped'
    >>>
```

Activity 33: Practicing Multiple Inheritance

It's the year 2000. You're working for a mobile phone company and have been tasked with modeling out the software for a cutting-edge phone that will have a built-in camera: a camera phone.

Create a class called **Camera** and a class called **MobilePhone** that will be the base classes of a derived class called **CameraPhone**. The **CameraPhone** class should be initialized with the **memory** attribute and should have a **take_picture()** method that prints out the message, **Say cheese!**.

The steps are as follows:

1. Create a **Camera** class that has a **take_picture()** method.
2. Create a **MobilePhone** class that will be initialized with a **memory** attribute.
3. Create a **CameraPhone** class that inherits from both the **MobilePhone** and **Camera** classes.
4. Initialize an instance of the **CameraPhone** class. Calling the **take_picture()** method on the instance should have an output that looks like this. Also, print the **memory** attribute:

```
Say cheese!
200KB
```

> **Note**
>
> Solution for this activity can be found at page 292.

Summary

In this chapter, we have begun our journey into OOP. OOP makes code more reusable; it makes it easier to design software; it makes code easier to test, debug, and maintain; and it adds a form of security to the data in an application. The behaviors of an object are known as methods, and you can add a method to a class by defining a function inside it. To be bound to your objects, this function needs to take in the argument **self**. We also covered class attributes and class methods in detail. We also took a look at encapsulation and the keywords that enable information hiding in Python. Information hiding is used to abstract away irrelevant details about the class from users. This chapter also covered inheritance in detail. We saw how to have a derived class inherit from a single base class, as well as multiple base classes. We also saw how to override methods: specifically, the **__init__()**, **__str__()**, and **__del__()** methods. This chapter completes our journey into object-oriented programming with Python.

In the next chapter, we will cover Python modules and packages in detail. We will also take a look at how to handle different types of files and related file operations.

8

Modules, Packages, and File Operations

Learning Objectives

By the end of this chapter, you will be able to:

- Describe what Python modules are and create your own
- Describe what Python packages are and create your own
- Work with the built-in Python modules
- Describe the **`file`** object in Python
- Read and write to Python files
- Work with structured data in Python files

This lesson introduces Python modules and packages. We also cover file operations available to us in Python.

Introduction

In the previous chapter, we have covered object-oriented programming in depth. We have covered important OOP concepts such as classes, methods, and inheritance. In this chapter, we will take a look at modules and file operations.

When you are working on any project, for example, a Word document, you may have a folder with the name of your project, and inside it you have the project files themselves. This helps you know which files are associated with which project. Next time you want to look at the project files, you won't have to search all over your computer for them. You can simply go to the project folder and find the files you need.

The concept of arranging work into files and folders also applies when programming in Python. You can arrange your code into pieces called **modules**, which makes it easier to group related functionality together. Once you have created a module, it becomes very easy to refer to that collection of functionalities again and also share and reuse the functionality defined in that module.

Specifically, a module in Python is any file with a `.py` extension. A Python module can contain any valid Python code you want, including classes, functions, or just variable definitions. Once you have created a module, you can then go ahead and import it elsewhere to use whatever resources are defined in the module.

A module can also be a directory containing Python files. Adding an `__init__.py` file inside a directory will tell the Python interpreter that the indicated directory is a module and that it will be registered in Python's module namespace. Note that it is no longer a requirement to have an `__init__.py` file, but it is a good practice to have one in case you need to run some custom code during module initialization.

Following the same trend, a **package** is therefore defined as a collection of modules. Packages are a good way of separating your modules from other people's modules to avoid name clashes.

Arranging your code into modules and packages makes it extremely easy for other developers to work with your code. They can easily see what resources are defined in each package or module and import them as needed.

Real-life applications will need to read input from files or write output to files at one point or another. Every time you open a document, be it a PDF file, a JPEG image, or even a `.py` file in your application, some code is running behind the scenes to process that file and output the data to you.

Imagine a payroll system. The system may be implemented in a way where user data is input into an Excel sheet containing the name of the person and the hours worked in a specific time period. The program will then read this data and calculate the amount of wages each person is owed and output a new file with the name of the person and their wages.

Does this sound interesting? This is just an example of what you can do once you can read input from a file. Files are useful for aggregating huge amounts of data that might be too cumbersome to enter through the keyboard line by line.

Imagine if the aforementioned payroll system needed input to be entered on the keyboard by hand. This would make payroll processing, for whatever company that used it, very cumbersome and prone to mistakes.

On the other hand, a file could contain thousands of lines, and the execution of the payroll would be faster because the input is not blocked by data entry. In fact, we shall build a rudimentary version of such a payroll system at the end of this chapter once you have a deeper understanding of how to work with files in Python.

Defining Modules

Following the definition of a module that was given earlier, you can now see that you have in fact been working with modules all along. Any valid `.py` file that you have created in this book is more or less a valid module.

In this section, though, we are going to be a bit more deliberate in how we create modules so that you can see how to define and import resources.

To recap, a valid Python module is any `.py` file containing valid Python code. This code could be variable definitions, functions, classes, methods, and so on. We are going to practice with a simple module that contains just one function.

Let's go ahead and create our first module.

Exercise 46: Creating Modules

In this exercise, we will create a module named **calculator**:

1. Create a file named **calculator.py**. This is where our module resources are going to live.

2. Inside the file, add the following simple function code in order to add and return the sum of two numbers:

```
def add(x, y):
    return x + y
```

   Remember, the **def** keyword is used to define functions in Python. In our example, we are defining a function called **add** that takes two parameters, **x** and **y**, which will be both integers. The function body simply returns the sum of the two numbers by using the **+** operator.

3. Next, run the Python interpreter using the **Python** command on your terminal in the same folder you created the **calculator** module in.

   You can now see and use the function you defined by importing the **calculator** module:

```
>>> import calculator
>>> calculator.add(8, 9)
17
```

Module names should follow normal Python variable naming conventions.
These include the following:

- They should follow snake_case (**lower_case_with_underscore**). Some good module names would be **module** and **another_module**. The following are examples of bad module names: **Amodule** and **AnotherModule**.
- Names should be descriptive. This means that the module name should reflect the purpose of the resources defined inside it; for example, a module containing mathematical functions can be named **math**, but not **string**.
- Only use underscores if it improves readability.

> **Note**
>
> These rules and other such rules are defined in **Python Enhancement Proposal 8** (**PEP8**), which is the go-to guide for formatting your Python code. You can take a look at it at https://www.python.org/dev/peps/pep-0008/.

When importing a module, the code in the module is executed exactly once, even if you have another statement in the code importing the exact same module elsewhere. Upon import, Python compiles the module's code, executes the code, and then populates the module's namespace with the resource names defined in the module. This makes the resources that have been imported accessible in the scope in which they have been imported.

Imports and Import Statements

There are several ways we can import and use the resources defined in a module. Taking the **`calculator`** module we defined previously, you have already seen one way to go about it, which is importing the whole module by using the **`import`** keyword and then calling a resource inside it by using the dot (`.`) notation, like so:

```
>>> import calculator
>>> calculator.add(8, 9)
17
```

Another way of accessing a module's resources is to use the **`from…import…`** syntax:

```
>>> from calculator import *
>>> add(8, 9)
17
```

Note that the **`*`** in the preceding example tells Python to import everything in the module named **`calculator`**. We can then use any resource defined in the **`calculator`** module by referring to the resource name (in our case, the **`add`** function) directly. You should note that using **`*`** is not a good practice. You should always strive to import exactly what you need.

What if you want to only import a few resources from the module? In that case, you can name them directly, as shown here:

```
>>> from calculator import add
>>> add(8, 9)
17
```

Another useful thing you can do is name your imports by using the **`as`** keyword:

```
>>> from calculator import add as a
>>> a(8, 9)
17
>>> import calculator as calc
>>> calc.add(8,9)
17
```

This can come in handy when you have a module or resource with a long name that you will be referring to in many places.

You can also group many imports by using brackets for easier readability:

```
>>> from random import (choice, choices)
```

Exercise 47: Importing Modules

In this exercise, we will practice how to import modules and familiarize ourselves with what successful imports look like and how non-successful imports will fail:

1. Open the Python interpreter on your machine.
2. Importing an existing library, such as **`string`**:

   ```
   >>> import string
   ```

   A successful import should not produce any output.
3. Next, import an undefined library, such as **`packt`**:

   ```
   >>> import packt
   Traceback (most recent call last):
       File "<input>", line 1, in <module>
           import packt
   ModuleNotFoundError: No module named 'packt'
   >>>
   ```

   You should get a **`ModuleNotFoundError`**. We will cover errors in the next chapter.

Modules and Packages

In this section, we will turn our focus toward the standard Python modules and Python packages.

The Module Search Path

When you import any module, Python will first check whether there is a built-in module with the specified name. An example of a built-in module is the **`string`** module.

If no built-in module is found, the interpreter will look for a file with the name of the module and the `.py` extension in the directories given by the **`sys.path`** variable. This variable is simply a list of strings which specifies where to search for modules.

How this variable is built is beyond the scope of this book. However, it is partly dependent on your defined **`PYTHONPATH`** and can be modified if necessary. You can read more about it in the Python documentation at https://docs.python.org/3/library/sys.html#sys.path.

For your interest, though, you can inspect the **`sys.path`** in your current environment by running the following commands on your terminal:

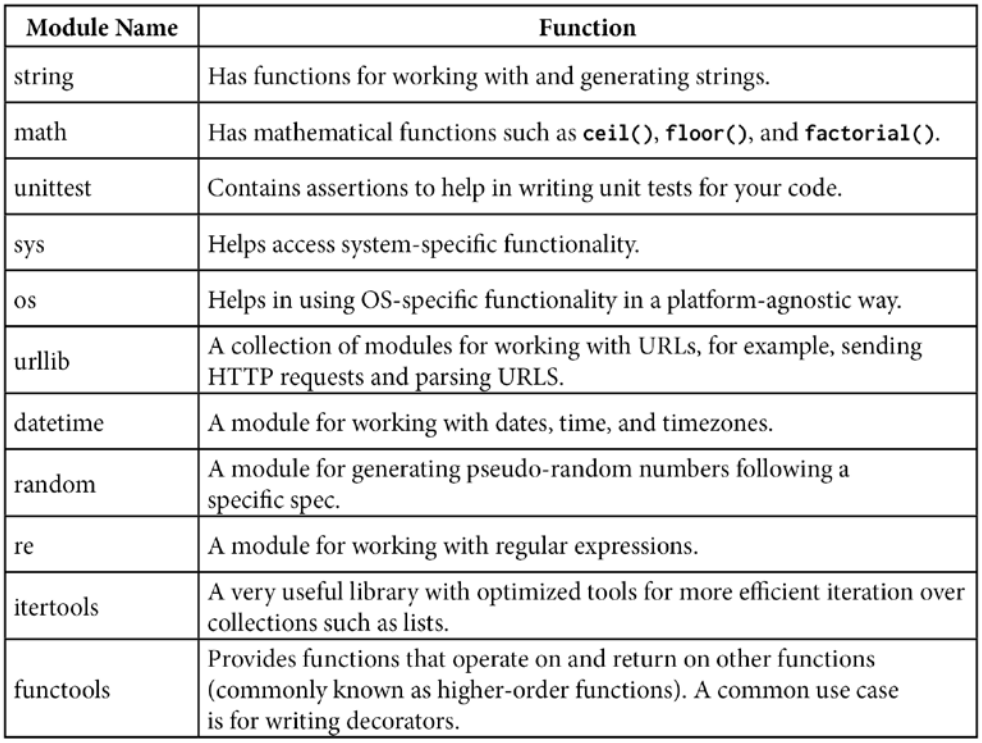
```
>>> import sys
>>> sys.path
['', '/usr/local/bin', '/usr/local/Cellar/python3/3.6.4_2/Frameworks/Python.
framework/Versions/3.6/lib/python36.zip', '/usr/local/Cellar/python3/3.6.4_2/
Frameworks/Python.framework/
Versions/3.6/lib/python3.6', '/usr/local/Cellar/python3/3.6.4_2/Frameworks/
Python.framework/Versions/3.6/lib/python3.6/lib-dynload', '/usr/local/Cellar/
python3/3.6.4_2/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-
packages']
>>>
```

Standard Python Modules

Python ships with a lot of ready-made modules and packages. These modules and packages are grouped into libraries and provided for use in the standard library.

An example is the **`math`** library, which provides utility functions for math operations.

Here are a couple of common modules that you are likely to interact with and what they do:



Module Name	Function
string	Has functions for working with and generating strings.
math	Has mathematical functions such as **`ceil()`**, **`floor()`**, and **`factorial()`**.
unittest	Contains assertions to help in writing unit tests for your code.
sys	Helps access system-specific functionality.
os	Helps in using OS-specific functionality in a platform-agnostic way.
urllib	A collection of modules for working with URLs, for example, sending HTTP requests and parsing URLS.
datetime	A module for working with dates, time, and timezones.
random	A module for generating pseudo-random numbers following a specific spec.
re	A module for working with regular expressions.
itertools	A very useful library with optimized tools for more efficient iteration over collections such as lists.
functools	Provides functions that operate on and return on other functions (commonly known as higher-order functions). A common use case is for writing decorators.

Figure 8.1: Common Python modules

You can import each module and use the **`dir()`** function to see a list of the methods implemented in each:

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__interactivehook__', '__
loader__', '__name__', '__package__', '__spec__', '__stderr__', '__stdin__',
'__stdout__', '_clear_type_cache', '_current_frames', '_debugmallocstats',
'_getframe', '_git', '_home', '_xoptions', 'abiflags', 'api_version',
'argv', 'base_exec_prefix', 'base_prefix', 'builtin_module_names',
'byteorder', 'call_tracing', 'callstats', 'copyright', 'displayhook',
'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable',
'exit', 'flags', 'float_info','float_repr_style', 'get_asyncgen_hooks',
'get_coroutine_wrapper', 'getallocatedblocks', 'getcheckinterval',
'getdefaultencoding', 'getdlopenflags', 'getfilesystemencodeerrors',
'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount',
'getsizeof', 'getswitchinterval', 'gettrace', 'hash_info', 'hexversion',
'implementation', 'int_info', 'intern', 'is_finalizing', 'last_traceback',
'last_type', 'last_value', 'maxsize', 'maxunicode', 'meta_path', 'modules',
'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix',
 'set_asyncgen_hooks', 'set_coroutine_wrapper', 'setcheckinterval',
'setdlopenflags', 'setprofile', 'setrecursionlimit', 'setswitchinterval',
'settrace', 'stderr', 'stdin', 'stdout', 'thread_info', 'version', 'version_
info', 'warnoptions']
>>>
```

You can also use the **help()** function to find more information about what a specific module does. You can use it in the same way as **dir()**. The **help** function shows the content of **documentation strings** (**docstrings**) defined on the resource.

For example, we can add a docstring to our previous **calculator** function so that future users can find out what the function does:

```
def add(x, y):
    """Return the sum of x and y."""
     return x + y
```

You can then use **help()** on it:

```
>>> help(add)
```

This will output the following:

```
Help on function add in module calculator:


add(x, y)
    Return the sum of x and y.
/var/folders/lb/k8ws21qn1x5gxq9zbrlqxp9w0000gn/T/tmphhbdd2y_ (END)
```

This is a best practice, and you should always strive to ensure that all your resources have docstring comments.

For a more detailed look at the built-in Python modules, you can check out the **Python Module Index** at https://docs.python.org/3/py-modindex.html.

Activity 34: Inspecting Modules

In this activity, we will try and use built-in Python functions to inspect modules. The steps are as follows:

1. Open the Python interpreter on your machine.
2. Import the **`itertools`** library.
3. Use the relevant built-in function to inspect the resources defined in it.
4. Use the relevant built-in function to read about how to use the **`itertools`** library.

> **Note**
>
> Solution for this activity can be found at page 293.

Packages

Packages are collections of modules. Each package must contain an **`__init__.py`** in the root of its directory, which indicates to the Python interpreter that the directory is a module.

The file doesn't need to contain anything, but you can use it for some added functionality (for example, to specify exactly what can and cannot be imported from the package).

When importing a package, the code in the **`__init__.py`** file is executed by the Python interpreter to build the package and the package namespace.

A sample package could have a file structure such as this:

```
mytools
|
├── init.py
├── calculator.py
├── tokenizer.py
```

In our example, the **`mytools`** package has two modules, the **`calculator`** and **`tokenizer`** modules.

Sample imports would look like this:

```
# import both modules
from mytools import calculator, tokenizer


# import one of the modules
from mytools import calculator


# import specific resources from a module
from mytools.calculator import add, another_function
```

Absolute Imports

An **absolute import** is an import that uses the full path from a project to the module being imported. An example of this is importing a built-in library:

```
import string
```

Relative Imports

A **relative import** is one that uses the relative path from the current directory to the directory where a module is. Imports of the following nature are relative:

```
from . import calculator
```

The preceding statement means import **`calculator`** from the current directory, which is represented by the dot (`.`).

> **Note**
>
> Always prefer absolute imports where possible.

Activity 35: Listing the Resources Defined in a Package or Module

You are implementing a program at work and you need to use a package that you have never interacted with before. You find that you need to write a script to list all resources that are available in that package.

Our aim here is to list the resources defined in a package or module.

Define a function called **`package_enumerator`**, which will take a package or module name and print out the names of all the resources defined in the package or module:

1. Define a function called **`package_enumerator`**.
2. Inside the function, use a **`for`** statement to list the package's resources.

A sample output is as follows:

```
>>> package_enumerator(string)

Formatter

Template

_ChainMap

_TemplateMetaclass

__all__

__builtins__

__cached__

__doc__

__file__

__loader__

__name__

__package__

__spec__

_re

_string

ascii_letters

ascii_lowercase

ascii_uppercase

capwords

digits
```

```
hexdigits
octdigits
printable
punctuation
whitespace
```

> **Note**
>
> Solution for this activity can be found at page 294.

Activity 36: Using Resources in a Module

To attempt this activity, you should have completed the previous chapters of this book. Our aim here is to practice using the resources defined in a module to implement a solution to a problem.

Write a function called **`random_number_generator`** whose purpose is to generate random integers between 0 and 1,000.

The function should use the resources defined in the built-in random module. It should take one argument, **`l`**, which is an integer, and return a list containing **`l`** random numbers.

For example, calling the function with **`3`** returns a list of three random numbers:

```
random_number_generator(3) -> [54,13,2]
```

The steps are as follows:

1. Import the **`random`** module.
2. Define a function named **`random_number_generator`**.
3. Inside the function, use a looping statement and the resources from our imported module to generate random numbers.

Here is the output when **`l`** is **`5`**:

```
>>> [3, 618, 298, 168, 701]
```

> **Note**
>
> Solution for this activity can be found at page 295.

File Operations

Files could be of many types, for example, text files with `.txt` extensions, data files in a `.csv` extension, or even executable files with `.exe` extensions. Not all file types are immediately readable. Some file types are proprietary and require specialized software to read them. An example of a proprietary file type is the PSD file that's generated by Adobe Photoshop.

We will be working with non-proprietary extensions which are easy to work with, such as `.txt`, `.csv`, and `.json` files for the purposes of this chapter. The skills that you will learn about will be transferrable to any other file you need to read with Python in the future.

The file Object

The `file` object is the default and easiest way to manipulate files in Python. It includes a couple of methods and attributes which make it easier for developers to read from, and write to, files in the filesystem.

There are two major `file` object types that are recognized in Python:

- Binary `file` objects: These can read and write byte-like objects.
- Text `file` objects: These can read and write strings objects.

The `open()` function, which we will be looking at later, is the easiest way to create a `file` object. Depending on the mode passed to the `open()` function, you will get back either a binary or text file object. We will be specifically working with text `file` objects.

The file Object Methods

The `file` object has several methods to make it easy to work with the underlying file. They include the following:

- `file.read()`: This method loads the entire file into memory.
- `file.readline()`: This method reads a single line from the file into memory.
- `file.readlines()`: This method reads all of the lines of a file into a list.
- `file.write()`: This method writes output to the file.
- `file.seek()`: This method is used to move the `file` object position to a certain location in the file.

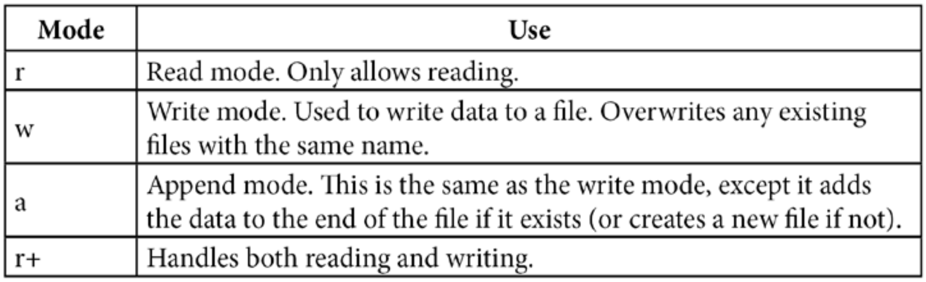
Reading and Writing to Files

In the previous section, we mentioned that to create a **`file`** object, you can use the built-in **`open()`** function with a filename and mode parameter. But how exactly does that work? Let's talk a bit more about the **`open()`** method now.

The open() Method

The **`open()`** method is a built-in function in Python that you can use to read a file. It takes two arguments, the filename and a mode, for example, **`open(name_of_file, mode)`**, and returns a **`file`** object that you can manipulate.

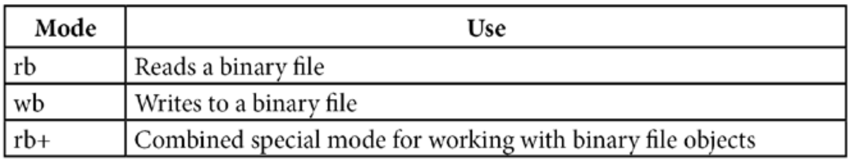
The mode passed to the function determines what kind of file object you will get back and what you will be able to do with it. Some available modes are as follows:



Mode	Use
r	Read mode. Only allows reading.
w	Write mode. Used to write data to a file. Overwrites any existing files with the same name.
a	Append mode. This is the same as the write mode, except it adds the data to the end of the file if it exists (or creates a new file if not).
r+	Handles both reading and writing.

Figure 8.2: Available modes with open()

The preceding table lists the most important modes you will encounter. Some additional specialized modes are as follows:



Mode	Use
rb	Reads a binary file
wb	Writes to a binary file
rb+	Combined special mode for working with binary file objects

Figure 8.3: Specialized modes with open()

Note

We will not be working with the binary modes for now, but feel free to play around with them in your own time. A good place to start is http://www.diveintopython3.net/files.html#binary.

Exercise 48: Creating, Reading, and Writing to Files

In this exercise, we will learn how to create, read, and write to files.

1. First, we will focus on creating a file. To create a file, open your terminal and start the Python interpreter.
2. Then, use the **`open()`** method to create a file:

   ```
   f = open('myfile.txt', 'w')
   ```

   You should not see any output after running the preceding command. What it does is simple; we open a file called **`myfile.txt`** in the write mode. Since that file does not exist, a blank file with that name will be created on your filesystem. The **`file`** object that's created is assigned to the variable **`f`**, which you can then use to manipulate the file.
3. Next, write something to the file, like so:

   ```
   >>> f.write("Hello, world\n")
   13
   ```

   We used the **`write()`** method on the **`file`** object to write the string **`"Hello, world"`** and a new line character to the file. You can see that **`13`** is the output. The **`write()`** method returns the number of characters written. The contents of **`myfile.txt`** is as follows:

   ```
   Hello, world
   ```
4. Write something else again:

   ```
   >>> f.write("Hello, world again")
   18
   ```
5. Now close the file. Remember to always do this after you have finished working with a file:

   ```
   f.close()
   ```
6. Check the contents of **`myfile.txt`**; you should see the two strings we wrote:

   ```
   Hello, world
   Hello, world again
   ```

7. To add more content to the file, open it in append mode:

   ```
   >>> f = open('myfile.txt', 'a')
   >>> f.write("More content")
   12
   >>> f.close()
   ```

   myfile.txt should now have the following content:

   ```
   Hello, world
   Hello, world againMore content
   ```

   The last two lines are joined because we did not write any newline character between them to the file.

8. Now, let's read a file. Use the **r** mode to read the file:

   ```
   >>> f = open('myfile.txt', 'r')
   >>> f.read()
   'Hello, world\nHello, world againMore content'
   ```

   You cannot write in this mode:

   ```
   >>> f.write("Some content")
   Traceback (most recent call last):
     File "<input>", line 1, in <module>
       f.write("Some content")
   io.UnsupportedOperation: not writable
   >>> f.close()
   ```

   Similarly, you cannot read in write (**w**) or append (**a**) modes:

   ```
   >>> f = open('myfile.txt', 'a')
   >>> f.read()
   Traceback (most recent call last):
     File "<input>", line 1, in <module>
       f.read()
   io.UnsupportedOperation: not readable
   >>> f.close()
   >>> f = open('myfile.txt', 'w')
   >>> f.read()
   Traceback (most recent call last):
     File "<input>", line 1, in <module>
       f.read()
   io.UnsupportedOperation: not readable
   >>> f.close()
   ```

9. If you need to combine both reading and writing, use the **r+** mode:

    ```
    >>> f = open('myfile.txt', 'r+')
    >>> f.read()
    ''
    >>> f.write("Some  content")
    13
    ```

    Note that the **r+** mode appends to the existing file when writing, so it is safe to use on existing files.

The with Context Manager

Now that you understand the different file reading and writing modes, let's talk about automating some of the processes around working with files—more specifically, auto-closing files after you have finished using them. Remember, we need to close files after using them so that they can be removed from memory and so the memory is then freed up for other processes.

Python comes with a handy context manager for working with files, which helps in writing neater code and freeing resources automatically when the code exits so that you do not have to. The syntax is as follows:

```
with open("myfile.txt", "r+") as f:

  content = f.read()


  print(content)
```

This code will read the specified file for us, print the content, and exit while automatically closing the file.

You can also process each line individually (this can be done with or without the context manager, so long as you have a **file** object):

```
with open("myfile.txt") as f:

  for line in f:

    print(line)
```

Activity 37: Performing File Operations

In this activity, we will create a file and add data to it.

The steps are as follows:

1. Use the **`with`** context manager or the **`open()`** function to create a **`file`** object.
2. Write **`I love Python`** to the file.

   Remember to close the **`file`** object when you're only using **`open()`**.

> **Note**
>
> Solution for this activity can be found at page 295.

Handling Structured Data

Now that you have a good handle on reading and writing to files, let's talk about how to deal with more structured data. In real-world applications, you will most likely have to read data in a structured format.

Do you remember the payroll application we talked about in the beginning of this chapter? Such data could be represented in a **CSV file**. A CSV file is a file with **comma-separated values**, usually arranged in columns.

Such data can then be easily read into a spreadsheet application, such as Excel, and manipulated there. Python provides a utility to work with CSV files, which we will cover in this topic.

Here is an example of a CSV file:

```
Name,City
"Nash, Colorado B.",Milton Keynes
"Herrera, Chase E.",Gentbrugge
"Hubbard, Leilani I.",Bremen
"Vinson, Marsden H.",Lakeland County
"Macias, Lawrence E.",Noisy-le-Grand
…
…
"Phelps, Amity V.",Morena
"Woods, Jaden V.",Portland
```

```
"Hyde, Duncan P.",Schellebelle
"Hendricks, Yoshio V.",Sperlinga
"Delgado, Emma T.",Reyhanlı
"Oneil, Orson B.",Rotello
"Sims, Noah C.",Selkirk
```

CSV files are popular because they have some advantages over files such as spreadsheets; for example:

- They are human readable and easy to parse.
- They are smaller, compact, and faster to work with.
- CSV files are easy to generate and have a standard format.

Another form of structured data is **JSON**. JSON is very useful, especially on the internet, for easily transferring data in the form of key-value pairs. As shown in the following sample, JSON data looks like this:

```
{
    "data": [
        {
            "name": "Rodriguez, Evangeline U.",
            "city": "Lobbes"
        },
        {
            "name": "Herman, Leandra P.",
            "city": "Tramonti di Sopra"
        }
    ]
}
```

Other formats used to transfer data over the internet include:

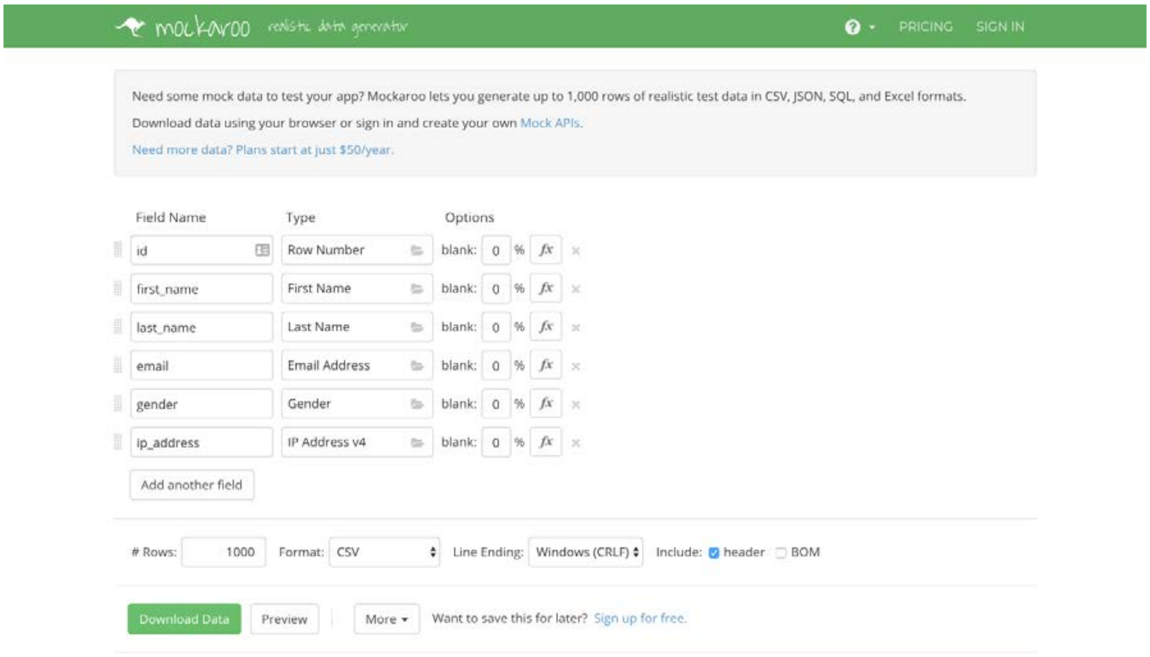
- XML (you can learn more about XML at https://www.w3schools.com/xml/)
- Protocol buffers (you can learn more about protocol buffers at https://developers.google.com/protocol-buffers/).

Working with CSV Data

Python includes the very handy `csv` module to help us work with CSV files. It makes it very easy to read the tabular data defined in a CSV file and operate on it by creating reader or writer objects.

For our examples, we are going to be using a dummy CSV file that we generated from Mockaroo (https://mockaroo.com/). Our download file is called `MOCK_DATA.csv` and is available in the accompanying code bundle.

If you want different mock data, you can go to the site and generate a CSV file, as illustrated in the following screenshot. This can easily be done on the home page without creating an account:



Need some mock data to test your app? Mockaroo lets you generate up to 1,000 rows of realistic test data in CSV, JSON, SQL, and Excel formats.
Download data using your browser or sign in and create your own Mock APIs.
Need more data? Plans start at just $50/year.

Field Name	Type	Options
id	Row Number	blank: 0 %
first_name	First Name	blank: 0 %
last_name	Last Name	blank: 0 %
email	Email Address	blank: 0 %
gender	Gender	blank: 0 %
ip_address	IP Address v4	blank: 0 %

Add another field

Rows: 1000 Format: CSV Line Ending: Windows (CRLF) Include: ✓ header ☐ BOM

Download Data Preview More ▾ Want to save this for later? Sign up for free.

Figure 8.4: Mockaroo home page

Make sure to change the format at the bottom to **CSV** and choose **Windows (CRLF)** as the option for **Line Ending**. Everything else can remain as is. You can then click on **Download Data**.

Reading a CSV File

Reading a CSV file requires us to import the **csv** module and create a reader object:

```
import csv


with open('MOCK_DATA.csv', 'r') as f:

    mock_data_reader = csv.reader(f)


line_count = 1


for row in mock_data_reader:


    if line_count > 1: # skipping line 1 which is the header row

        print(row)


     line_count += 1:
```

As you can see, we use the standard file reading **open()** function within a context manager with the read mode. We then create a reader object to help us read the CSV. The reader object returns each row as a list of strings.

Running the preceding code, you should see an output like the following:

```
['10', 'Sherwin', 'Lydall', 'slydall9@eventbrite.com', 'Male',
'9.153.155.182']

['11', 'Liz', 'Linthead', 'llintheada@skyrock.com', 'Female',
'232.213.9.139']

['12', 'Connie', 'Moreby', 'cmorebyb@csmonitor.com', 'Male', '145.75.39.34']

['13', 'Christie', 'Gerasch', 'cgeraschc@typepad.com', 'Male',
'195.178.145.92']

['14', 'Ellen', 'Bocke', 'ebocked@foxnews.com', 'Female', '94.217.132.103']

['15', 'Cecile', 'Maginn', 'cmaginne@omniture.com', 'Female',
'253.72.149.37']

['16', 'Beverly', 'Hendrich', 'bhendrichf@google.co.uk', 'Female',
'224.43.75.87']
```

Since the first row contains field names, we can extract those from the output.

In that way, you can isolate the column names from your data and process each separately in code:

```
['id', 'first_name', 'last_name', 'email', 'gender', 'ip_address', '']
['1', 'Skipton', 'Mattiacci', 'smattiacci0@berkeley.edu', 'Male',
'196.4.235.14', '']
['2', 'Keith', 'Rosenfelder', 'krosenfelder1@icq.com', 'Male',
'26.145.67.148', '']
['3', 'Helaina', 'Ind', 'hind2@japanpost.jp', 'Female', '32.201.244.49', '']
['4', 'Joelly', 'Milesap', 'jmilesap3@dot.gov', 'Female', '74.199.54.89',
'']
['5', 'Portie', 'MacCoughan', 'pmaccoughan4@mapquest.com', 'Male',
'61.150.82.117', '']
['6', 'Fletcher', 'Reynold', 'freynold5@blogs.com', 'Male',
'254.202.181.187', '']
['7', 'Arnaldo', 'Batch', 'abatch6@umich.edu', 'Male', '137.231.46.255', '']
['8', 'Galven', 'Turban', 'gturban7@google.es', 'Male', '180.52.139.226',
'']
['9', 'Cele', 'La Vigne', 'clavigne8@house.gov', 'Female', '125.202.213.224',
'']
```

Once you have your data in the reader object, you can proceed to manipulate it as you want, just like any other list of strings.

Writing to the CSV File

Writing to a CSV file is done by using the writer object. The writer object accepts a list of items to write for each row:

```
import csv


with open('example.csv', 'w') as f:
    example_data_writer = csv.writer(f)


    example_data_writer.writerow(['name', 'age'])
    example_data_writer.writerow(['Steven', 25])
```

Running this code will create a file called **`example.csv`** with the following contents:

```
name,age
Steven,25
```

Writing a dict to the CSV File

Instead of writing rows using lists, you can write dictionaries to CSV files. To do that, you would have to use a **`DictWriter`** object instead of the usual writer object, even though their behaviors are pretty similar. However, you will also have to define a list of fieldnames which will specify the order in which values will be written to the file. Here is an example:

```
import csv


with open('people.csv', 'w') as f:
    fields = ['name', 'age']
    people_writer = csv.DictWriter(f, fieldnames=fields)


    people_writer.writeheader() # writes the fields as the first row
    people_writer.writerow({'name': 'Santa Claus', 'age': 1000})
```

This will create a file called **`people.csv`** with the following contents:

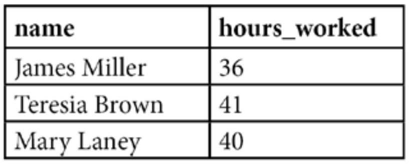
```
name,age
Santa Claus,1000
```

Activity 38: Working with Files

Suppose you are an HR manager and you need to create a file containing wages for your employees. We will read in a CSV file with the employee names and the hours worked, and then output another CSV file with their wages calculated.

The steps are as follows:

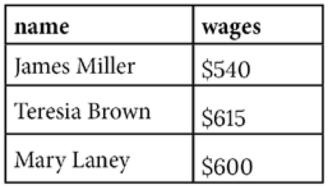
1. Write a script to read in a CSV file of the following format:



name	hours_worked
James Miller	36
Teresia Brown	41
Mary Laney	40

Figure 8.5: Input file format

2. Output a new CSV file of the following format:



name	wages
James Miller	\$540
Teresia Brown	\$615
Mary Laney	\$600

Figure 8.6: Output file format

   Here, wages are calculated using the formula *hours_worked* * 15.

> **Note**
>
> Solution for this activity can be found at page 295.

Working with JSON Data

JSON stands for **JavaScript Object Notation**. It's a data format that was built for exchanging data in a human-readable way. Most APIs you will interact with on the internet today will be using JSON as the data-exchange format. It is therefore important that we talk about it a little before we move on.

Python includes a `json` module with a few functions to assist in parsing JSON and converting some Python objects, for example, dictionaries into JSON objects.

There are two critical methods from the `json` module that you need to know about to use JSON in Python.

json.dumps()

The **`json.dumps()`** method is used for JSON encoding, for example, converting dictionaries into JSON objects.

Here is an example:

```
import json


sample = {
  "name": "Bert Bertie",
  "age": 24
}


sample_json = json.dumps(sample)
print(sample_json)
print(type(sample_json))
```

This code defines a dictionary called **`sample`** and then encodes it to JSON using **`json.dumps()`**. The **`dumps`** function will return a JSON string of the object. Here is the output:

```
'{"name": "Bert Bertie", "age": 24}'
<class 'str'>
```

You can see from the preceding output that the sample **`dict`** was encoded into a **`string`** JSON object.

json.loads()

If you want to decode the JSON object, you can use **`json.loads()`** to do so. We are going to add some code in our original code to illustrate this:

```
original_sample = json.loads(sample_json)
print(original_sample)
print(type(original_sample))
```

This will output the following additional lines:

```
{'name': 'Bert Bertie', 'age': 24}
<class 'dict'>
```

As you can see, the **`json.loads()`** function directly reverses what the **`json.dumps()`** function does – or rather, it creates Python objects out of JSON objects.

> **Note**
>
> We will not go into too much detail about working with JSON. However, feel free to read more on encoding and decoding JSON objects in the official Python documentation at https://docs.python.org/3.6/library/json.html.

Summary

Congratulations! You now have a good understanding of modules and packages and how they function in Python. You can now go ahead and use the structuring methods you have learned to build even larger applications.

Large frameworks written in Python, such as Django and Flask, use the same principles of modules and packages that you have learned here. For reference, check out the Flask project source code on GitHub to see how files are arranged in a real-life project. You will notice that the same concepts that you just learned about are in use.

As we said in the introduction, working with data in files is a fact of any developer's life. Python, as a general-purpose scripting language, comes with a lot of built-in tools to help you read, manipulate, and write to files very easily in your program. You should now have a good grasp of how to use these tools to not only manipulate text files but also to work with more complex data such as CSV files. In our final chapter activity, you had the chance to put those skills to good use by calculating wages for the employees of a fictional company.

In our next and final chapter, we will cover error handling in detail. We will also look at the built-in exception classes and create our own custom exceptions.

9

Error Handling

Learning Objectives

By the end of this chapter, you will be able to:

- Describe what errors and exceptions are
- Handle errors and exceptions when they occur
- Define and use your own custom exceptions

This lesson describes error handling in Python. We look at the try…except clause and its modified types. Lastly, we cover custom exceptions. Solutions for the activities in this lesson can be found on page 289.

Introduction

In life, things sometimes do not go according to plan. You may find, for example, that you have budgeted to buy some things for when you go to the store. But when you actually arrive at the store, you see some items that are not on your list are on sale and you buy them! That is an incident where the initial plan was not executed well and did not produce the expected results.

A similar scenario can arise while programming. When you write some code and run it, unexpected situations can occur, which may cause the code not to be executed correctly or not produce the expected results. For example, there could be a problem with syntax, an undefined variable you are trying to use, or even a completely unforeseen scenario. When code does not execute as intended, we say that an error has occurred.

Some errors can be logical errors. These can occur when specifications are not followed. For example, this could be a function that is supposed to return the sum of two numbers but actually returns the product. This is a logical error and is not the kind of error we will be tackling in this module. Instead, we will focus more on the runtime errors you are likely to see when the Python interpreter encounters problems during code execution.

> **Note**
>
> For more information on how to handle logical errors, look at this excellent guide on unit testing at https://docs.python-guide.org/writing/tests/.

By now, you will have probably encountered many errors while coding in Python. This chapter aims to equip you with a better understanding of why errors occur and what to do about them when they do. This helps prevent scenarios where, for example, an error occurs on your application and because it is not handled well, brings the whole application down.

Errors and Exceptions in Python

You may have noticed that we mention both errors and exceptions and use them in an almost interchangeable way. Why is that? Are they the same thing? Usually, they are. But in Python there are some slight differences in the meanings of the two words.

Exceptions are errors that occur when your program is running, while **errors**, for example, syntax errors, occur before program execution happens.

How to Raise Exceptions

You can raise exceptions yourself for one reason or another by using the **`raise`** keyword. You will want to raise exceptions when something occurs and you want to inform users of your app, for example, the input given is incorrect. Let's try out an example:

```
def raise_an_error(error):
    raise error


raise_an_error(ValueError)
```

In our example, we define a function called **`raise_an_error()`**, which takes the **`error`** class name and raises it. We then try out the code by calling it with the built-in **`ValueError`** exception class.

If all goes well, you should see an output like this when running the script:

```
Traceback (most recent call last):
  File "python", line 4, in <module>
  File "python", line 2, in raise_an_error
ValueError
```

Built-In Exceptions

Python ships with a ton of built-in exception classes to cover a lot of error situations so that you do not have to define your own. These classes are divided into **`Base`** error classes, from which other error classes are defined, and **`Concrete`** error classes, which define the exceptions you are more likely to see from time to time.

> **Note**
>
> Built-in exception classes cover many error situations so that you do not have to define your own. For more information on built-in exceptions, visit https://docs.python.org/3/library/exceptions.html.

We shall cover more on the **`Exception`** base class and its uses later on in this chapter. For now, let's take a look at some common error and exception classes and understand what they mean.

SyntaxError

A **SyntaxError** is very common, especially if you are new to Python. It occurs when you type a line of code which the Python interpreter is unable to parse.

Here is an example:

```
def raise_an_error(error)
    raise error


raise_an_error(ValueError)
```

In this implementation of our previous **raise_an_error()** method, we made a deliberate syntax error. Can you guess what it is?

Running the previous script will output the following:

```
Traceback (most recent call last):
  File "python", line 1
    def raise_an_error(error)
                            ^
SyntaxError: invalid syntax
```

You can see that a **SyntaxError** was raised with the message **invalid syntax**. You can also see from the stack trace the exact line the error occurred on and a small **^** pointing to the source of the error, in this case, the omission of **:** in the function signature. Adding it will enable the interpreter to parse the line successfully and move on with execution.

ImportError

An **ImportError** occurs when an import cannot be resolved.

For example, importing a non-existent module will raise a **ModuleNotFoundError**, that is a subclass of the **ImportError** class:

```
>>> import nonexistentmodule


Traceback (most recent call last):
  File "python", line 1, in <module>
ModuleNotFoundError: No module named 'nonexistentmodule'
```

KeyError

A **`KeyError`** occurs when a dictionary key is not found while trying to access it. Here's an example:

```
person = {
    "name": "Rich Brown",
    "age": 56
}


print(person["gender"])
```

The **`person`** dictionary defined here has only two keys: **`name`** and **`age`**. Attempting to read a key called **`gender`** raises a **`KeyError`**:

```
Traceback (most recent call last):
  File "python", line 6, in <module>
KeyError: 'gender'
```

A simple way of mitigating a **`KeyError`** is to use the **`get()`** method that is defined on dictionaries, when accessing keys, which will return **`None`** or a custom value if the key is non-existent:

```
person = {
    "name": "Rich Brown",
    "age": 56
}


print(person.get("gender"))
```

This will output **`None`**.

TypeError

A **TypeError** will occur if you attempt to do an operation on a value or object of the wrong type.

Here are a few examples:

Adding a **string** to an **int** results in the following error:

```
>>> "string" + 8
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    "string" + 8
TypeError: must be str, not int
```

This also occurs when passing wrong arguments (for example, passing an integer when we expect a list):

```
a = 6

for index, value in enumerate(a):
    print(value)
```

This will also result in a **TypeError**:

```
Traceback (most recent call last):
  File "python", line 3, in <module>
TypeError: 'int' object is not iterable
```

The error displayed means that we cannot loop over an **int** object.

AttributeError

An **AttributeError** is raised when assigning or referencing an attribute fails.

Here is an example. We are going to try and call a method called **push** on a **list** object:

```
a = [1,2,3]

a.push(4)
```

An **AttributeError** is thrown because the **list** object has no attribute called **push**:

```
Traceback (most recent call last):
  File "python", line 3, in <module>
AttributeError: 'list' object has no attribute 'push'
```

> **Note**
>
> Remember, to add a value to the end of a list, use the **append()** method.

IndexError

An **IndexError** occurs if you are trying to access an index (for example, in a list) which does not exist.

Here is an example:

```
a = [1,2,3]

print(a[3])
```

The list, **a**, only has three indexes: **0**, **1**, and **2**. Attempting to access index **3** will cause the interpreter to throw an **IndexError**:

```
Traceback (most recent call last):
  File "python", line 3, in <module>
IndexError: list index out of range
```

NameError

A **NameError** occurs when a specified name cannot be found either locally or globally. This usually happens because the name or variable is not defined.

For example, printing any undefined variable should throw a **NameError**:

```
>>> print(age)

Traceback (most recent call last):
  File "python", line 1, in <module>
NameError: name 'age' is not defined
```

FileNotFoundError

The last **`Exception`** class we will cover in this section is the **`FileNotFoundError`**.

This error is raised if a file you are attempting to read or write is not found.

Here is an example:

```
with open('input.txt', 'r') as myinputfile:

    for line in myinputfile:

        print(line)
```

The preceding code attempts to read a file called **`input.txt`**. Since we have deliberately not created any such file on our environment, we get a **`FileNotFoundError`**:

```
Traceback (most recent call last):

  File "python", line 1, in <module>

FileNotFoundError: [Errno 2] No such file or directory: 'input.txt'
```

Activity 39: Identifying Error Scenarios

During programming, errors often occur. But how can we anticipate when something will cause an error? By thoroughly understanding error scenarios and causes. In this activity, we will create error scenarios. Let's write some code that will cause the following errors:

The steps are as follows:

`KeyError`

1. Create the following dictionary:

   ```
   building = dict(
       name="XYZ Towers",
       type="Business Premises"
   )
   ```

2. Write a **`print`** statement that uses a key that's not defined in this dictionary.

`AttributeError`

1. Import the **`string`** module.
2. Use an attribute that's not defined in the **`string`** module.

> **Note**
>
> Solution for this activity can be found at page 296.

Handling Errors and Exceptions

Handling errors and exceptions starts long before you get to running your code. Right from the planning phase, you should have contingencies in place to avoid running into errors, especially logical errors that may be harder to catch in some cases.

Practices such as **defensive programming** can help mitigate future errors in some cases.

According to Wikipedia:

"Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances. Defensive programming practices are often used where high availability, safety, or security is needed."

Defensive programming is an approach that's used to improve software and source code, in terms of the following:

- General quality—by reducing the number of software bugs and problems.
- Making the source code comprehensible—the source code should be readable and understandable, so that it is approved in a code audit.
- Making the software behave in a predictable manner, despite unexpected inputs or user actions.

In this section, we aim to show you some basic error handling in Python so that the next time errors occur, they do not bring your program to a crashing halt.

Exercise 49: Implementing the try…except Block

The simplest way to handle errors is to use the **`try…except`** block. The code in the **`try`** section is executed and if an error, which is specified in the **`except`** block, is thrown, the code in the **`except`** block is executed.

Once the block finishes executing, the rest of the code executes as well. This prevents errors from causing your program to crash.

Let's see an example. We are going to use the code example that we used to describe the **`FileNotFoundError`** in the previous section to demonstrate the **`try…except`** block:

1. Write the following script:

   ```
   with open('input.txt', 'r') as myinputfile:
       for line in myinputfile:
           print(line)

   print("Execution never gets here")
   ```

2. Run this script to get the following output:

   ```
   Traceback (most recent call last):
     File "python", line 1, in <module>
   FileNotFoundError: [Errno 2] No such file or directory: 'input.txt'
   ```

   As you can see, the error caused the execution to stop before the last line.

3. Rewrite the same code with the **`FileNotFoundError`** handled using a **`try…except`** block:

   ```
   try:
       with open('input.txt', 'r') as myinputfile:
           for line in myinputfile:
               print(line)
   except FileNotFoundError:
       print("Whoops! File does not exist.")

   print("Execution will continue to here.")
   ```

   After wrapping the code in a **`try…except`** block, instead of crashing, the script executes the code in the **`except`** block and continues to next line:

   ```
   Whoops! File does not exist.
   Execution will continue to here.
   ```

4. Note that the way we have written our code means that if any other exception occurs, it will be unhandled and the code will still crash.

You can handle more than one exception by creating a tuple, like this:

```
try:
    with open('input.txt', 'r') as myinputfile:
        for line in myinputfile:
            print(line)
except (FileNotFoundError, ValueError):
    print("Whoops! File does not exist.")

print("Execution will continue to here.")
```

5. A better way is to handle them individually and do something different for each error you get. Implement this as follows:

```
try:
    with open('input.txt', 'r') as myinputfile:
        for line in myinputfile:
            print(line)
except FileNotFoundError:
    print("Whoops! File does not exist.")
except ValueError:
    print("A value error occurred")
```

In our case, only the **except** clause for the **FileNotFoundError** will be executed. However, if a **ValueError** also occurs, both **except** clauses will be executed.

6. If you are not quite sure which exception will be thrown, you can catch the generic **Exception**, which will catch any exception that's thrown. It is a good practice to catch the generic **Exception** at the end of more specific **except** clauses and not by itself.

Implement it like this:

```
try:
    with open('input.txt', 'r') as myinputfile:
        for line in myinputfile:
            print(line)
except FileNotFoundError:
    print("Whoops! File does not exist.")
except ValueError:
    print("A value error occurred")
except Exception:
    print("Something unforeseen happened")

print("Execution will continue to here.")
```

However, doing this is a bad practice:

```
try:
    with open('input.txt', 'r') as myinputfile:
        for line in myinputfile:
            print(line)
except Exception:
    print("Something unforeseen happened")

print("Execution will continue to here.")
```

However, both approaches are valid syntax and will work just fine.

> **Note**
>
> Python will not allow you to catch syntax errors. These should always be fixed before your code can run at all.

Exercise 50: Implementing the try…except…else Block

In this exercise, we will implement the **`try…except`** block with an additional **`else`** statement.

The **`try…except…else`** block is a minor modification of the traditional **`try…except`** block so that it can include an **`else`** block. The code in the **`else`** block is always executed if no error has occurred.

1. Implement the **`try…except…else`** block as follows:

```
try:
    with open('input.txt', 'r') as myinputfile:
        for line in myinputfile:
            print(line)
except FileNotFoundError:
    print("Whoops! File does not exist.")
except ValueError:
    print("A value error occurred")
except Exception:
    print("Something unforeseen happened")
else:
    print("No error because file exists")

print("Execution will continue to here.")
```

2. Run the preceding script; if an error is thrown, the output will be as follows:

```
Whoops! File does not exist.
Execution will continue to here.
```

The output in the case of no error being thrown will be as follows:

```
No error because file exists
Execution will continue to here.
```

> **Note**
>
> The output shown here ignores the contents of **`input.txt`** printed by **`print(line)`** since it is not relevant to the **`try…except`** logic.

Exercise 51: Implementing the finally Keyword

The **`finally`** keyword defines a code block that *must* execute before the **`try…except`** block exits, irrespective of whether any exception occurred.

It is usually the last block in the **`try…except`** block after all the exception handling logic and will always be executed:

1. Continuing with our file reading example, the **`finally`** keyword can be implemented like this:

```
try:
    with open('input.txt', 'r') as myinputfile:
        for line in myinputfile:
            print(line)
except FileNotFoundError:
    print("Whoops! File does not exist.")
except ValueError:
    print("A value error occurred")
except Exception:
    print("Something unforeseen happened")
finally:
    print("I will always show up")

print("Execution will continue to here.")
```

2. Run this script; the output when the **FileNotFoundError** occurs will look like this:

   ```
   Whoops! File does not exist.
   I will always show up
   Execution will continue to here.
   ```

   If the file exists, the output will look like this:

   ```
   I will always show up
   Execution will continue to here.
   ```

The **finally** keyword is useful, for example, in cases where some clean-up logic needs to happen. This might be closing files, closing database connections, or releasing system resources.

Activity 40: Handling Errors

You are completing some code, but you have an unhandled error. What do you do to make sure that the error doesn't stop your program prematurely? In this activity, we will practice handling errors.

The following code throws an error:

```
import random


print(random.randinteger(1,10))
```

Identify and handle the error so that when it occurs, the message **Oops! Something went wrong** is printed to the terminal.

> **Note**
>
> Solution for this activity can be found at page 297.

Custom Exceptions

Built-in exceptions cover a wide range of situations. Sometimes, however, you may need to define a custom exception to fit your specific application situation; for example, a **RecipeNotValidError** exception for when a recipe is not valid in your cooking app.

In this case, Python contains the ability to add custom errors by extending the base **Exception** class.

Implementing Your Own Exception Class

Exceptions should be named with names ending with the word **Error**. Let's create the **RecipeNotValidError** we talked about previously as a custom exception:

```
class RecipeNotValidError(Exception):
    def __init__(self):
        self.message = "Your recipe is not valid"


try:
    raise RecipeNotValidError
except RecipeNotValidError as e:
    print(e.message)
```

The custom exception class should just contain a few attributes that will help the user get more information about what error occurred. Our sample implementation has the **message** attribute, which we have used to get details on the error message. This is the output you would get if you run the preceding snippet:

```
Your recipe is not valid
```

Activity 41: Creating Your Own Custom Exception Class

As a developer, you may need to create exception classes that handle custom exceptions that are not defined in the standard library in a certain way. In this activity, we will practice creating custom exception classes.

The steps are as follows:

1. Subclass the **Exception** class and create a new **Exception** class of your choosing.
2. Write some code where you use the **try…except** flow to capture and handle your custom exception.

Note

Solution for this activity can be found at page 297.

Summary

In this chapter, we talked about errors and exceptions, what they are, and how to avoid them. We looked at a few built-in errors and the scenarios that would cause them to be raised. We then moved on to handling them by using **`try…except`** and **`finally`** blocks. We also covered how to implement our own custom exceptions by using the **`Exception`** base class.

This should give you the ability to make your programs more robust by handling both seen and unforeseen issues that may arise during code execution. Handling errors should also help prevent unpleasant usability or security issues from cropping up when your code is in the wild.

This book is an introduction to the general-purpose language Python. We have covered topics such as variable names; working with functions; modules; data structures such as lists, tuples, and dictionaries; and even working with files. This book is designed to get you from beginner level to intermediate Python developer level. We hope that you have enjoyed this experience. Feel free to go back to the chapter activities to refresh your memory regarding what you have learned.

Appendix A

About

This section is included to assist the students to perform the activities present in the book. It includes detailed steps that are to be performed by the students to complete and achieve the objectives of the book.

Chapter 1: Introducing Python

Activity 1: Working with the Python Shell

Solution:

```
>>> print("Happy birthday")
Happy birthday
>>> 17 + 35 * 2
87
>>> print(1, 2, 3, 4, 5, 6, 7, 8, 9)
1 2 3 4 5 6 7 8 9
>>>
```

Activity 2: Running Simple Python Scripts

Solution:

```
import sys


print("*-------------------------------")
print("|", "First name: ", sys.argv[1])
print("|", "Second name: ", sys.argv[2])
print("*-------------------------------")
```

Activity 3: Using Variables and Assign Statements

Solution:

1. Open your editor.
2. Create a file named **`calculate_speed.py`** and save it.
3. On the first two lines, we'll declare our two variables for the distance in kilometers and the time in hours:

   ```
   distance_in_km = 150
   time_in_hours = 2
   ```

4. In the next two lines, we'll calculate the distances in miles and the distance in meters based on the distance in kilometers:

   ```
   distance_in_mi = distance_in_km / 1.6
   distance_in_mtrs = distance_in_km * 1000
   ```

   We'll then calculate the time in seconds based on the time in hours:

   ```
   time_in_seconds = time_in_hours * 3600
   ```

5. Next, we'll calculate the speed in kilometers per hour, the speed in miles per hour, and the speed in miles per second:

   ```
   speed_in_kph = distance_in_km / time_in_hours
   speed_in_mph = distance_in_mi / time_in_hours
   speed_in_mps = distance_in_mtrs / time_in_seconds
   ```

6. Finally, we'll print out our results:

   ```
   print("The speed in kilometers per hour is:", speed_in_kph)
   print("The speed in miles per hour is:", speed_in_mph)
   print("The speed in meters per second is:", speed_in_mps)
   ```

7. We can now save our script and run it by using the **`python calculate_speed.py`** command.

Activity 4: Variable Assignment and Variable Naming Conventions

Solution:

1. Open your editor.
2. Create a file named **`circle.py`** and save it.
3. On the first two lines, we'll declare our constant, π (**`PI`**), and the radius of the circle:

   ```
   PI = 3.14159
   radius = 7
   ```

4. On the next two lines, we'll run our calculations:

   ```
   area = PI * radius**2
   circumference = 2 * PI * radius
   ```

   The **`**`** operator raises the value to the specified exponent; in this case, the exponent is **`2`**.

5. Lastly, we'll display the results:

   ```
   print("Circumference of the circle:", circumference)
   print("Area of the circle:", area)
   ```

6. We can now save our script and run it by using the **`python circle.py`** command.

Activity 5: Fixing Indentations in a Code Block

Solution:

```
>>> if 5 > 2:
...     print("Greater than")
...     x = 5
...     print(x * 2)
... else:
...     print("Less than")
...     print(2)
```

Activity 6: Implementing User Input and Comments in a Script

Solution:

1. Open your editor.
2. Create a file named **`multiplication_table.py`** and save it.
3. On the first line, we'll add a docstring explaining what our file does. Then, we'll assign **`number`** to the user's input and cast it to an integer:

   ```
   """
   This script generates a multiplication table from 1 to 10 for
   any given number.
   """
   number = int(input("Generate a multiplication table for: "))
   ```

4. Next, we'll print 20 underscores as the top border of the table:

   ```
   print("_" * 20)
   ```

5. We'll then multiply our number with each number from 1 to 10 and print that out:

   ```
   print("1:", number)
   print("2:", number * 2)
   print("3:", number * 3)
   print("4:", number * 4)
   print("5:", number * 5)
   print("6:", number * 6)
   print("7:", number * 7)
   print("8:", number * 8)
   print("9:", number * 9)
   print("10:", number * 10)
   ```

6. Finally, we'll print 20 underscores again for the bottom border of the table, as in step 4.

   ```
   print("_" * 20)
   ```

7. Save the file and run it by using the **`python multiplication_table.py`** command.

Chapter 2: Data Types

Activity 7: Order of Operations

Solution:

```
>>> 5 * (4 - 2) + 100 / ( 5/2 ) * 2
90.0
```

Activity 8: Using Different Arithmetic Operators

Solution:

1. Create a file named **`convert_days.py`**.
2. On the first line, let's declare the user input. It's an integer, so we cast the string we get from the input function:

   ```
   days = int(input("Number of days:"))
   ```

3. We can then calculate the number of years in that set of days. We floor divide to get an integer:

   ```
   years = days // 365
   ```

4. Next, we convert the remaining days that weren't converted to years into weeks:

   ```
   weeks = (days % 365) // 7
   ```

5. Then, we get any remaining days that weren't converted to weeks:

   ```
   days = days - ((years * 365) + (weeks * 7))
   ```

6. Finally, we'll print everything out:

   ```
   print("Years:", years)
   print("Weeks:", weeks)
   print("Days:", days)
   ```

7. We can then save and run the script by using the **`python convert_days.py`** command.

Activity 9: String Slicing

Solution:

1. **`'g'`**
2. **`'9'`**
3. **`'fright'`**
4. **`' 1, 2, 3'`**
5. **`'A man, a plan, a canal: Panama'`**

Activity 10: Working with Strings

Solution:

1. Create a file named **`convert_to_uppercase.py`**.
2. On the first line, we'll request the user for the string to convert:

   ```
   string = input("String to convert: ")
   ```

3. On the next line, we'll request the number of last letters to convert:

   ```
   n = int(input("How many last letters should be converted? "))
   ```

4. Next, we'll get the first part of the string:

   ```
   # First part of the string
   start = string[:len(string) - n]
   ```

5. Then, we'll get the last part of the string, that is, the one we'll be converting:

   ```
   # last part of the string that we're converting.
   end = string[len(string) - n:]
   ```

6. Then, we will concatenate the first and last part back together with the last substring transformed:

   ```
   print(start + end.upper())
   ```

7. Finally, we can run the script with the **`python convert_to_uppercase.py`** command.

Activity 11: Manipulating Strings

Solution:

1. Create a file named **`count_occurrences.py`**.
2. We'll take in the user inputs for the sentence and the query:

   ```
   sentence = input("Sentence: ")
   query = input("Word to look for in sentence: ")
   ```

3. Next, we'll sanitize and format our inputs by removing the whitespace and converting them to lowercase:

   ```
   # sanitize our inputs
   sentence = sentence.lower().strip()
   query = query.lower().strip()
   ```

4. We'll count the occurrences of the substring by using the **`str.count()`** method:

   ```
   num_occurrences = sentence.count(query)
   ```

   Then, we will print the results:

   ```
   print(f"There are {num_occurrences} occurrences of '{query}' in the
   sentence.")
   ```

5. You can run the script by using the **`python count_occurrences.py`** command.

Activity 12: Working with Lists

Solution:

1. Create a file named **`get_first_n_elements.py`**.
2. On the first line, we'll create the array:

   ```
   array = [55, 12, 37, 831, 57, 16, 93, 44, 22]
   ```

3. Next, we'll print the array out and fetch user input for the number of elements to fetch from the array:

   ```
   print("Array: ", array)
   n = int(input("Number of elements to fetch from array: "))
   ```

4. Finally, we'll print out the slice of the array from the first element to the *nth* element:

   ```
   print(array[0: n])
   ```

5. Then, we'll run the script by using the **`python get_first_n_elements.py`** command.

Activity 13: Using Boolean Operators

Solutions:

1. The code block with the missing Boolean operator is added:

   ```
   n = 124
   if n % 2 == 0:
       print("Even")
   ```

2. The code block with the missing Boolean operator is added:

   ```
   age = 25
   if age >= 18:
       print("Here is your legal pass.")
   ```

3. The code block with the missing Boolean operator is added:

   ```
   letter = "b"
   if letter not in ["a", "e", "i", "o", "u"]:
     print(f"'{letter}' is not a vowel.")
   ```

Chapter 3: Control Statements

Activity 14: Working with the if Statement

Solution:

```
answer = input("Return TRUE or FALSE: Python was released in 1991:\n")


if answer == "TRUE":

  print('Correct')

elif answer == "FALSE":

  print('Wrong')

elif answer != ("TRUE" or "FALSE"):

  print('Please answer TRUE or FALSE')


print('Bye!')
```

Activity 15: Working with the while Statement

Solution:

1. Define the password and the Boolean validator first:

   ```
   user_pass = "random"
   valid = False
   ```

2. Initiate the **`while`** loop and ask for the user's input:

   ```
   while not valid:
        password = input("please enter your password: ")
   ```

3. Validate the password and return an error if the input is invalid:

   ```
        if password == user_pass:
          print("Welcome back user!")
          valid =  True
        else:
          print("invalid password, try again... ")
   ```

Your block of code should look as follows; take note of the indentation:

```
user_pass = "random"
valid = False
while not valid:
     password = input("please enter your password: ")
     if password == user_pass:
       print("Welcome back user!")
       valid =  True
     else:
       print("invalid password, try again... ")
```

Activity 16: The for loop and range Function

Solution:

```
>>> total = 0
>>> for number in range(2,101,2):
    total += number
>>> print(total)
```

Activity 17: Nested Loops

Solution:

```
for even in range(2,11,2):
    for odd in range(1,11,2):
        val = even + odd
        print(even, "+", odd, "=", val)
```

Activity 18: Breaking out of Loops

Solution:

```
for number in range(0,200):
    if number == 0:
        continue
    elif number % 3 != 0:
        continue
    elif type(number) != int:
        continue
    else:
        pass
    print(number)
```

Chapter 4: Functions

Activity 19: Function Arguments

Solution:

```
def print_arguments(*args):
    for value in args:
        if type(value) == int:
            continue
        print(value)
```

Activity 20: Using Lambda Functions

Solution:

```
>>> answer = lambda number, power : number ** power
```

Chapter 5: Lists and Tuples

Activity 21: Using the List Methods

Solution:

```
>>> wild = ["Lion", "Zebra", "Panther", "Antelope"]
>>> wild
['Lion', 'Zebra', 'Panther', 'Antelope']
>>> wild.append("Elephant")
>>> wild
['Lion', 'Zebra', 'Panther', 'Antelope', 'Elephant']
>>> animals = []
>>> animals.extend(wild)
>>> animals
['Lion', 'Zebra', 'Panther', 'Antelope', 'Elephant']
>>> animals.insert(2, "Cheetah")
>>> animals
['Lion', 'Zebra', 'Cheetah', 'Panther', 'Antelope', 'Elephant']
>>> animals.pop(1)
'Zebra'
>>> animals.insert(1, "Giraffe")
>>> animals
['Lion', 'Giraffe', 'Cheetah', 'Panther', 'Antelope', 'Elephant']
>>> animals.sort(key=None, reverse=False)
>>> animals
['Antelope', 'Cheetah', 'Elephant', 'Giraffe', 'Lion', 'Panther']
```

Activity 22: Using Tuple Methods

Solution:

```
pets = ('cat', 'cat', 'cat', 'dog', 'horse')
c = pets.count("cat")
d = len(pets)
if (c/d)*100 > 50.0:
    print("There are too many cats here")
else:
    print("Everything is good")
```

Chapter 6: Dictionaries and Sets

Activity 23: Creating a Dictionary

Solution:

```
>>> d = dict()
>>> type(d)
<class 'dict'>
```

Activity 24: Arranging and Presenting Data Using Dictionaries

Solution:

```
def sentence_analyzer(sentence):
  solution = {}

  for char in sentence:
    if char is not ' ':
      if char in solution:
        solution[char] += 1
      else:
        solution[char] = 1

  return solution
```

Activity 25: Combining Dictionaries

Solution:

```
def dictionary_masher(dict_a, dict_b):
  for key, value in dict_b.items():
    if key not in dict_a:
      dict_a[key] = value

  return dict_a
```

Activity 26: Building a Set

Solution:

The **set** function can help us create our function:

```
def set_maker(the_list):
  return set(the_list)
```

Activity 27: Creating Unions of Elements in a Collection

Solution:

```
def find_union(list_a, list_b):
  union = []

  for element in list_a + list_b:
    if element not in union:
      union.append(element)

  return union
```

Chapter 7: Object-Oriented Programming

Activity 28: Defining a Class and Objects

Solution:

This is the **MobilePhone** class definition:

```
class MobilePhone:
    def __init__(self, display_size, ram, os):
        self.display_size = display_size
        self.ram = ram
        self.os = os
```

The solution can also be found in the code files for this chapter in the file named **mobile_phone1.py**.

Activity 29: Defining Methods in a Class

Solution:

```
import math


class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * self.radius ** 2

    def circumference(self):
        return 2 * math.pi * self.radius

    def change_radius(self, new_radius):
        self.radius = new_radius



circle = Circle(7)

while True:
    radius = float(input("Circle radius: "))
    circle.change_radius(radius)
    print("Area:", circle.area())
    print("Circumference:", circle.circumference())
```

The solution can also be found in the code files for this chapter in the file named **`circle.py`**.

Activity 30: Creating Class Attributes

Solution:

```
class Elevator:
    occupancy_limit = 8

    def __init__(self, occupants):
        if occupants > self.occupancy_limit:
            print("The maximum occupancy limit has been exceeded."
                  f" {occupants - self.occupancy_limit} occupants must exit 
the elevator.")
        self.occupants = occupants
```

```
elevator1 = Elevator(6)
print("Elevator 1 occupants:", elevator1.occupants)
elevator2 = Elevator(10)
print("Elevator 2 occupants:", elevator2.occupants)
```

The solution can also be found in the code files for this book in the file named **`elevator.py`**.

Activity 31: Creating Class Methods and Using Information Hiding

Solution:

```
class MusicPlayer:
    firmware_version = 1.0

    def __init__(self):
        self.__tracks = ["Moonage Daydream", "Starman", "Life on Mars?"]
        self.current_track = None

    def play(self):
        self.current_track = self.__tracks[0]

    def list_tracks(self):
        return self.__tracks

    @classmethod
    def update_firmware(cls, new_version):
        if new_version > cls.firmware_version:
            cls.firmware_version = new_version

player = MusicPlayer()
print("Tracks currently on device:", player.list_tracks())
```

```
MusicPlayer.update_firmware(2.0)
print("Updated player firmware version to", player.firmware_version)
player.play()
print("Currently playing", f"'{player.current_track}'")
```

The solution can also be found in the code files for this course in the file named **`musicplayer.py`**.

Activity 32: Overriding Methods

Solution:

```
class Cat:
    def __init__(self, mass, lifespan, speed):
        self.mass_in_kg = mass
        self.lifespan_in_years = lifespan
        self.speed_in_kph = speed

    def vocalize(self):
        print("Chuff")

    def __str__(self):
        return f"The {type(self).__name__.lower()} "\
            f"weighs {self.mass_in_kg}kg,"\
            f" has a lifespan of {self.lifespan_in_years} years and "\
            f"can run at a maximum speed of {self.speed_in_kph}kph."



class Tiger(Cat):
    def __init__(self, mass, lifespan, speed):
        super().__init__(mass, lifespan, speed)
        self.coat_pattern = "striped"
    def __str__(self):
        facts = super().__str__()
        facts = f"{facts} It also has a {self.coat_pattern} coat."
        return facts


tiger = Tiger(310, 26, 65)
print(tiger)
```

The solution for this activity can also be found in the **`tiger.py`** file in the attached code files for this chapter.

Activity 33: Practicing Multiple Inheritance

Solution:

```
class MobilePhone:
    def __init__(self, memory):
        self.memory = memory



class Camera:
    def take_picture(self):
        print("Say cheese!")


class CameraPhone(MobilePhone, Camera):
    pass



cameraphone = CameraPhone("200KB")
cameraphone.take_picture()
print(cameraphone.memory)
```

The solution for this activity can also be found in the **`cameraphone.py`** file in the attached code files for this chapter.

Chapter 8: Modules, Packages, and File Operations

Activity 34: Inspecting Modules

Solution:

1. Inspect the resources defined in the **`itertools`** library by using the **`dir()`** function:

```
>>> import itertools
>>> dir(itertools)
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'_grouper', '_tee', '_tee_dataobject', 'accumulate', 'chain',
'combinations', 'combinations_with_replacement', 'comp
ress', 'count', 'cycle', 'dropwhile', 'filterfalse', 'groupby', 'islice',
'permutations', 'product', 'repeat', 'starmap', 'takewhile', 'tee', 'zip_
longest']
```

2. Use the **`help()`** function to read about how to use the **`itertools`** library:

```
>>> help(itertools)
```

You should see an output such as this:

```
Help on built-in module itertools:

NAME
    itertools - Functional tools for creating and using iterators.
DESCRIPTION
    Infinite iterators:
    count(start=0, step=1) --> start, start+step, start+2*step, ...
    cycle(p) --> p0, p1, ... plast, p0, p1, ...
    repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times

    Iterators terminating on the shortest input sequence:
    accumulate(p[, func]) --> p0, p0+p1, p0+p1+p2
    chain(p, q, ...) --> p0, p1, ... plast, q0, q1, ...
    chain.from_iterable([p, q, ...]) --> p0, p1, ... plast, q0, q1, ...
    compress(data, selectors) --> (d[0] if s[0]), (d[1] if s[1]), ...
    dropwhile(pred, seq) --> seq[n], seq[n+1], starting when pred fails
    groupby(iterable[, keyfunc]) --> sub-iterators grouped by value of
keyfunc(v)
```

```
        filterfalse(pred, seq) --> elements of seq where pred(elem) is False
        islice(seq, [start,] stop [, step]) --> elements from
               seq[start:stop:step]
        starmap(fun, seq) --> fun(*seq[0]), fun(*seq[1]), ...
        tee(it, n=2) --> (it1, it2 , ... itn) splits one iterator into n
        takewhile(pred, seq) --> seq[0], seq[1], until pred fails
        zip_longest(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...

        Combinatoric generators:
        product(p, q, ... [repeat=1]) --> cartesian product
        permutations(p[, r])
        combinations(p, r)
        combinations_with_replacement(p, r)

    CLASSES
        builtins.object
            accumulate
            chain
            combinations
            combinations_with_replacement
            compress
            count
            cycle
            dropwhile
            filterfalse
            groupby
            islice
    /var/folders/lb/k8ws21qn1x5gxq9zbrlqxp9w0000gn/T/tmp__ayalkd
```

Activity 35: Listing the Resources Defined in a Package or Module

Solution:

```
def package_enumerator(package):
  """
  List the resources defined in a package or module.
  """
  resources = dir(package)
  for resource in resources:
      print(resource)
```

```
import string
package_enumerator(string)
```

Activity 36: Using Resources in a Module

Solution:

```
import random


def random_number_generator(l):
    """
    Generate a list of random numbers of length l.
    """
    output = []

    for i in range(l):
        output.append(random.randint(0, 1000))

    return output
```

Activity 37: Performing File Operations

Solution:

```
with open("myfile.txt", 'w') as f:
  f.write("I love Python")
```

Activity 38: Working with Files

Solution:

```
import csv


output_data = []


with open('input.csv', 'r') as f:
    mock_data_reader = csv.reader(f)
    output_data = []
    line_count = 1
    for row in mock_data_reader:
      if line_count != 1:
        row[1] = int(row[1]) * 15
        output_data.append(row)
      line_count += 1

with open('output.csv', 'w') as f:
    fields = ['name', 'wages']
    output_writer = csv.DictWriter(f, fieldnames=fields)

    output_writer.writeheader()

    for line in output_data:
      output_writer.writerow(
        {
          'name': line[0],
          'wages': line[1]
        }
      )
```

This sample solution is also present in the **wage_calculator.py** file in the accompanying code files.

Chapter 9: Error Handling

Activity 39: Identifying Error Scenarios

Solution:

KeyError:

```
building = dict(
    name="XYZ Towers",
    type="Business Premises"
)
```

```
print(f"I went to visit {building['name']} yesterday at
{building['street']}.")
```

`AttributeError`:

```
import string


letters = string.asciiletters
```

Activity 40: Handling Errors

Solution:

```
import random


try:
    print(random.randinteger(1,10))
except AttributeError:
    print("Oops! Something went wrong")
```

Activity 41: Creating Your Own Custom Exception Class

Solution:

```
class RecipeNotValidError(Exception):
    def __init__(self):
        self.message = "Your recipe is not valid"


try:
    raise RecipeNotValidError
except RecipeNotValidError as e:
    print(e.message)
```

>

Index

About

All major keywords used in this book are captured alphabetically in this section. Each one is accompanied by the page number of where they appear.