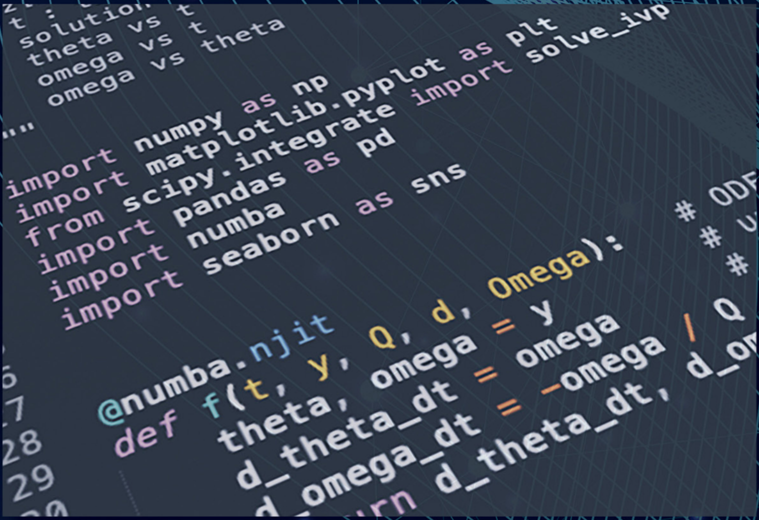
SERIES IN COMPUTATIONAL BIOPHYSICS

Introduction to Python for Science and Engineering

SECOND EDITION



```
solution vs t
theta vs t
omega vs theta
omega vs
"""
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
import pandas as pd
import numba
import seaborn as sns
                                          # OD
                                          # U
                                          #
@numba.njit
def f(t, y, Q, d, Omega):
    theta, omega = y
    d_theta_dt = omega
    d_omega_dt = -omega / Q
    return d_theta_dt, d_o
```

David J. Pine

CRC Press
Taylor & Francis Group

Introduction to Python for Science and Engineering

Introduction to Python for Science and Engineering offers a quick and incisive introduction to the Python programming language for use in any science or engineering discipline. The approach is pedagogical and "bottom up," which means starting with examples and extracting more general principles from that experience. No prior programming experience is assumed.

Readers will learn the basics of Python syntax, data structures, input and output, conditionals and loops, user-defined functions, plotting, animation, and visualization. They will also learn how to use Python for numerical analysis, including curve fitting, random numbers, linear algebra, solutions to nonlinear equations, numerical integration, solutions to differential equations, and fast Fourier transforms.

Readers learn how to interact and program with Python using JupyterLab and Spyder, two simple and widely used integrated development environments.

All the major Python libraries for science and engineering are covered, including NumPy, SciPy, Matplotlib, and Pandas. Other packages are also introduced, including Numba, which can render Python numerical calculations as fast as compiled computer languages such as C but without their complex overhead.

David J. Pine has taught physics and chemical engineering for over 40 years at four different institutions: Cornell University (as a graduate student), Haverford College, UCSB, and NYU, where he is a Professor of Physics, Mathematics, and Chemical & Biomolecular Engineering. He has taught a broad spectrum of courses, including numerical methods. He does research on optical materials and in experimental soft-matter physics, which is concerned with materials such as polymers, emulsions, and colloids.

Introduction to Python for Science and Engineering

Second Edition

David J. Pine

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

Designed cover image: David J. Pine

Second edition published 2025
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

First edition published by CRC Press 2018

ISBN: 978-1-032-65033-3 (hbk)
ISBN: 978-1-032-67390-5 (pbk)
ISBN: 978-1-032-67395-0 (ebk)

DOI: 10.1201/9781032673950

Typeset in Nimbus font
by KnowledgeWorks Global Ltd.

To Alex Pine
who introduced me to Python

Contents

Preface to First Edition xvii

Preface to Second Edition xxi

About the Author xxiii

CHAPTER 1 ▪ Introduction 1

1.1 INTRODUCTION TO PYTHON FOR SCIENCE AND ENGINEERING 1

1.2 INSTALLING PYTHON 3

CHAPTER 2 ▪ Launching Python 4

2.1 INTERACTING WITH PYTHON: THE IPYTHON SHELL 4

2.2 THE IPYTHON SHELL 6

2.3 INTERACTIVE PYTHON AS A CALCULATOR 6

2.3.1 Binary Arithmetic Operations in Python 7

2.3.2 Types of Numbers 7

2.3.3 Numbers as Objects 9

2.4 VARIABLES AND ASSIGNMENT 10

2.4.1 Names and the Assignment Operator 10

2.4.2 Legal and Recommended Variable Names 14

2.4.3 Reserved Words in Python 14

2.5 SCRIPT FILES AND PROGRAMS 15

2.5.1 Editors for Python Scripts 15

2.5.2 First Scripting Example 16

2.6 PYTHON MODULES 19

2.6.1 Python Modules and Functions: A First Look 20

2.6.2 Some NumPy Functions 22
2.6.3 Scripting Example 2 23
2.6.4 Different Ways of Importing Modules 24
2.7 GETTING HELP: DOCUMENTATION IN IPYTHON 25
2.8 PERFORMING SYSTEM TASKS WITH IPYTHON 26
2.8.1 Magic Commands 26
2.8.2 Tab Completion 29
2.8.3 Recap of Commands 30
2.9 PROGRAMMING ERRORS 30
2.9.1 Error Checking 30
2.10 EXERCISES 31

CHAPTER 3 ▪ Integrated Development Environments 33

3.1 PROGRAMMING AND INTERACTING WITH PYTHON 33
3.2 PROGRAMMING STYLE AND CODING ERRORS: PEP 8 AND LINTERS 34
3.3 THE SPYDER IDE 35
3.3.1 Autoformatting and Linting in Spyder 36
3.3.2 Running Python Code in Spyder 37
3.4 THE JUPYTERLAB IDE 39
3.4.1 Jupyter Extensions 41
3.5 JUPYTER NOTEBOOKS 43
3.6 LAUNCHING A JUPYTER NOTEBOOK 43
3.7 RUNNING PROGRAMS IN A JUPYTER NOTEBOOK 45
3.8 ANNOTATING A JUPYTER NOTEBOOK 45
3.8.1 Adding Headings and Text 46
3.8.2 Saving a Jupyter Notebook 48
3.8.3 Editing and Rerunning a Notebook 48
3.8.4 Quitting a Jupyter Notebook 49
3.8.5 Working with an Existing Jupyter Notebook 49

CHAPTER 4 ▪ Strings, Lists, Arrays, and Dictionaries 50

4.1 STRINGS 51

4.1.1 Unicode Characters 52
4.2 LISTS 53
4.2.1 Slicing Lists 55
4.2.2 Multidimensional Lists 56
4.2.3 Appending to Lists 57
4.2.4 Tuples 58
4.3 DICTIONARIES 59
4.4 NUMPY ARRAYS 61
4.4.1 Creating Arrays (1-d) 61
4.4.2 Mathematical Operations with Arrays 63
4.4.3 Slicing and Addressing Arrays 66
4.4.4 Fancy Indexing: Boolean Indexing 67
4.4.5 Multidimensional Arrays and Matrices 69
4.4.6 Broadcasting 73
4.4.7 Differences Between Lists and Arrays 75
4.5 OBJECTS 76
4.6 EXERCISES 78

CHAPTER 5 ■ Input and Output 83

5.1 KEYBOARD INPUT 83
5.2 SCREEN OUTPUT 85
5.2.1 Formatting Output with `str.format()` 85
5.2.2 Formatting with f-strings 89
5.2.3 Printing Arrays 89
5.3 FILE INPUT 91
5.3.1 Reading Data from a Text File 91
5.3.2 Reading Data from an Excel File: CSV Files 92
5.4 FILE OUTPUT 94
5.4.1 Writing Data to a Text File 94
5.4.2 Writing Data to a CSV File 96
5.5 EXERCISES 97

CHAPTER 6 ▪ Conditionals and Loops 100

6.1 CONDITIONALS 102

6.1.1 `if`, `elif`, and `else` Statements 102

6.1.2 More about Boolean Variables, Operators, and Expressions 107

6.2 LOOPS 110

6.2.1 `while` Loops 110

6.2.2 `for` Loops 113

6.2.3 Loop Control Statements 118

6.2.4 Loops and Array Operations 119

6.3 LIST COMPREHENSIONS 120

6.4 HANDLING EXCEPTIONS 122

6.5 EXERCISES 125

CHAPTER 7 ▪ Functions 127

7.1 USER-DEFINED FUNCTIONS 128

7.1.1 Looping Over Arrays in User-Defined Functions 130

7.1.2 Fast Array Processing for User-Defined Functions 131

7.1.3 Functions with More than One Input or Output 133

7.1.4 Type Hints 134

7.1.5 Positional and Keyword Arguments 134

7.1.6 Variable Number of Arguments 136

7.1.7 Passing a Function Name and Its Parameters as Arguments 137

7.2 NAMESPACE AND SCOPE IN PYTHON 140

7.2.1 Scope: Four Levels of Namespaces in Python 140

7.2.2 Variables and Arrays Created Entirely Within a Function 142

7.2.3 Passing Lists and Arrays to Functions: Mutable and Immutable Objects 143

7.3 ANONYMOUS FUNCTIONS: `LAMBDA` EXPRESSIONS 146

7.4 NUMPY OBJECT ATTRIBUTES: METHODS AND INSTANCE VARIABLES 148

7.5 EXAMPLE: LINEAR LEAST SQUARES FITTING 150
7.5.1 Linear Regression 151
7.5.2 Linear Regression with Weighting: χ^2 153
7.6 EXERCISES 155

CHAPTER 8 ■ Plotting 163

8.1 AN INTERACTIVE SESSION WITH PYPLOT 164
8.2 BASIC PLOTTING 166
8.2.1 Specifying Line and Symbol Types and Colors 171
8.2.2 Error Bars 174
8.2.3 Setting Plotting Limits and Excluding Data 176
8.2.4 Subplots 179
8.3 LOGARITHMIC PLOTS 180
8.3.1 Semi-Log Plots 181
8.3.2 Log-Log Plots 183
8.4 MORE ADVANCED GRAPHICAL OUTPUT 183
8.4.1 An Alternative Syntax for a Grid of Plots 187
8.5 PLOTS WITH MULTIPLE AXES 189
8.5.1 Plotting Quantities that Share One Axis but not the Other 189
8.5.2 Two Separate Scales for a Data Set 190
8.6 PLOTS WITH INSETS 191
8.7 MATHEMATICS AND GREEK SYMBOLS 193
8.7.1 Manual Axis Labeling 198
8.8 THE STRUCTURE OF MATPLOTLIB: OOP AND ALL THAT 199
8.8.1 The Backend Layer 200
8.8.2 The Artist Layer 202
8.8.3 The PyPlot (scripting) Layer 204
8.9 CONTOUR AND VECTOR FIELD PLOTS 206
8.9.1 Making a 2D Grid of Points 206
8.9.2 Contour Plots 207
8.9.3 Streamline Plots 211

8.9.4 Vector Field (quiver) Plots 215
8.10 THREE-DIMENSIONAL PLOTS 217
8.10.1 Cartesian Coordinates 217
8.10.2 Polar Coordinates 219
8.11 EXERCISES 220

CHAPTER 9 ■ Numerical Routines: SciPy and NumPy 225

9.1 SPECIAL FUNCTIONS 226
9.1.1 Important Note on Importing SciPy Subpackages 229
9.2 SPLINE FITTING, SMOOTHING, AND INTERPOLATION 229
9.2.1 Interpolating Splines 230
9.2.2 Smoothing Splines 234
9.2.3 Finding Roots (zero crossings) of Numerical Data 236
9.3 CURVE FITTING 236
9.3.1 Linear Fitting Functions 237
9.3.2 Polynomial Fitting Functions 240
9.3.3 Nonlinear Fitting Functions 242
9.4 RANDOM NUMBERS 247
9.4.1 Initializing NumPy's Random Number Generator 247
9.4.2 Uniformly Distributed Random Numbers 247
9.4.3 Normally Distributed Random Numbers 248
9.4.4 Random Distribution of Integers 249
9.4.5 Poisson Distribution of Random Integers 249
9.5 LINEAR ALGEBRA 250
9.5.1 Basic Computations in Linear Algebra 251
9.5.2 Solving Systems of Linear Equations 251
9.5.3 Eigenvalue Problems 253
9.6 SOLVING NONLINEAR EQUATIONS 255
9.6.1 Single Equations of a Single Variable 255
9.6.2 Solving Systems of Nonlinear Equations 258
9.7 NUMERICAL INTEGRATION 258
9.7.1 Single Integrals of Functions 259

9.7.2 Double Integrals 263
9.7.3 Integrating Numerical Data 264
9.8 SOLVING ODES 265
9.8.1 A First-Order ODE 265
9.8.2 A Second-Order ODE 269
9.9 DISCRETE (FAST) FOURIER TRANSFORMS 273
9.9.1 Continuous and Discrete Fourier Transforms 273
9.9.2 The SciPy FFT Library 274
9.10 EXERCISES 276

CHAPTER 10 ■ Python Classes: Encapsulation 284

10.1 A VERY SIMPLE CLASS 286
10.2 A BRIEF INTRODUCTION TO MODULES AND PACKAGES 290
10.2.1 Pythonpath 291
10.3 A CLASS FOR READING AND PROCESSING DATA 293
10.3.1 The Data 294
10.3.2 The Class 295
10.3.3 The Code 297
10.4 A CLASS OF RELATED FUNCTIONS 300
10.5 INHERITANCE 304
10.6 EXERCISES 305

CHAPTER 11 ■ Data Manipulation and Analysis: Pandas 306

11.1 DATA STRUCTURES: SERIES AND DATAFRAME 307
11.1.1 Series 307
11.1.2 DataFrame 310
11.2 INDEXING DATAFRAMES 315
11.2.1 Pandas `iloc` Indexing 315
11.2.2 Pandas `loc` Indexing 316
11.3 READING DATA FROM FILES USING PANDAS 317
11.3.1 Reading from Excel Files Saved as CSV Files 317
11.3.2 Reading from an Excel File 324
11.3.3 Getting Data from the Web 326

11.4 EXTRACTING INFORMATION FROM A DATAFRAME 328

11.5 PLOTTING WITH PANDAS 332

11.6 GROUPING AND AGGREGATION 336

11.6.1 The `groupby` Method 337

11.6.2 Iterating Over Groups 338

11.6.3 Reformatting DataFrames 341

11.6.4 Custom Aggregation of DataFrames 343

11.7 EXERCISES 346

CHAPTER 12 ■ Animation 351

12.1 ANIMATING A SEQUENCE OF IMAGES 351

12.1.1 Simple Image Sequence 352

12.1.2 Annotating and Embellishing Videos 355

12.2 ANIMATING FUNCTIONS 357

12.2.1 Animating for a Fixed Number of Frames 358

12.2.2 Animating until a Condition is Met 362

12.3 COMBINING VIDEOS WITH ANIMATED FUNCTIONS 368

12.3.1 Using a Single Animation Instance 368

12.3.2 Combining Multiple Animation Instances 370

12.4 EXERCISES 372

CHAPTER 13 ■ Speeding Up Numerical Calculations 374

13.1 NUMBA'S BASIC FUNCTIONS 375

13.1.1 Faster Loops and NumPy Functions 376

13.1.2 Vectorizing Functions with Numba 382

13.1.3 Numba Signatures 383

13.2 SIMULATIONS 385

13.2.1 A Brownian Dynamics Simulation 385

13.2.2 Nondimensional Simulation Variables and Parameters 387

13.2.3 Simulation with the Numba Decorator 389

13.2.4 Performance and Saving/Reading Large Data Files 392

13.2.5 Isolating Numerical Code for Numba 392

13.3 USING NUMBA WITH CLASSES 393
13.4 OTHER FEATURES OF NUMBA 396
13.5 EXERCISES 396

Appendix A ▪ Maintaining Your Python Installation 400

A.1 UPDATING PYTHON 400
A.2 TESTING YOUR PYTHON INSTALLATION 400
A.3 INSTALLING FFMPEG FOR SAVING ANIMATIONS 402
A.4 ADDING FOLDERS/DIRECTORIES TO YOUR PYTHON PATH 402
A.4.1 Spyder 403
A.4.2 macOS 403
A.4.3 Windows 404
A.4.4 Linux 405

Appendix B ▪ Glossary 406

Appendix C ▪ Python Resources 409

C.1 PYTHON PROGRAMS AND DATA FILES INTRODUCED IN THIS TEXT 409
C.2 WEB RESOURCES 409
C.3 BOOKS 410

Index 413

Preface to First Edition

The aim of this book is to provide science and engineering students a practical introduction to technical programming in Python. It grew out of notes I developed for various undergraduate physics courses I taught at NYU. While it has evolved considerably since I first put pen to paper, it retains its original purpose: to get students with no previous programming experience writing and running Python programs for scientific applications with a minimum of fuss.

The approach is pedagogical and "bottom up," which means starting with examples and extracting more general principles from that experience. This is in contrast to presenting the general principles first and then examples of how those general principles work. In my experience, the latter approach is satisfying only to the instructor. Much computer documentation takes a top-down approach, which is one of the reasons it's frequently difficult to read and understand. On the other hand, once examples have been seen, it's useful to extract the general ideas in order to develop the conceptual framework needed for further applications.

In writing this text, I assume that the reader:

- has never programmed before;
- is not familiar with programming environments;
- is familiar with how to get around a Mac or PC at a very basic level; and
- is competent in basic algebra, and for Chapters 8 and 9, calculus, linear algebra, ordinary differential equations, and Fourier analysis. The other chapters, including 10–12, require only basic algebra skills.

This book introduces, in some depth, four Python packages that are important for scientific applications:

NumPy, short for Numerical Python, provides Python with a multidimensional array object (like a vector or matrix) that is at the center of virtually all fast numerical processing in scientific Python. It is both versatile and powerful, enabling fast numerical computation that, in some cases, approaches speeds close to those of a compiled language like C, C++, or Fortran.

SciPy, short for Scientific Python, provides access through a Python interface to a very broad spectrum of scientific and numerical software written in C, C++, and Fortran. These include routines to numerically differentiate and integrate functions, solve differential equations, diagonalize matrices, take discrete Fourier transforms, perform least-squares fitting, as well as many other numerical tasks.

Matplotlib is a powerful plotting package written for Python and capable of producing publication-quality plots. While there are other Python plotting packages available, Matplotlib is the most widely used and is the *de facto* standard.

Pandas is a powerful package for manipulating and analyzing data formatted and labeled in a manner similar to a spreadsheet (think Excel). Pandas is very useful for handling data produced in experiments and is particularly adept at manipulating large data sets in different ways.

In addition, Chapter 12 provides a brief introduction to Python classes and to PyQt5, which provides Python routines for building graphical user interfaces (GUIs) that work on Macs, PCs, and Linux platforms.

Chapters 1–7 provide the basic introduction to scientific Python and should be read in order. Chapters 8–12 do not depend on each other and, with a few mild caveats, can be read in any order.

As the book's title implies, the text is focused on scientific uses of Python. Many of the topics that are of primary importance to computer scientists, such as object-oriented design, are of secondary importance here. Our focus is on learning how to harness Python's ability to perform scientific computations quickly and efficiently.

The text shows the reader how to interact with Python using IPython, which stands for Interactive Python, through one of three different interfaces, all freely available on the web: Spyder, an integrated development environment, Jupyter Notebooks, and a simple IPython terminal. Chapter 2 provides an overview of Spyder and an introduction to IPython, which is a powerful interactive environment tailored to scientific use of Python. Appendix B provides an introduction to Jupyter notebooks.

Python 3 is used exclusively throughout the text with little reference to any version of Python 2. It's been nearly 10 years since Python 3 was introduced, and there is little reason to write new code in Python 2; all the major

Python packages have been updated to Python 3. Moreover, once Python 3 has been learned, it's a simple task to learn how Python 2 differs, which may be needed to deal with legacy code. There are many lucid web sites dedicated to this sometimes necessary but otherwise mind-numbing task.

The scripts, programs, and data files introduced in this book are available at https://github.com/djpine/python-scieng-public-2, the GitHub site for this book.

Finally, I would like to thank Étienne Ducrot, Wenhai Zheng, and Stefano Sacanna for providing some of the data and images used in Chapter 12, and Mingxin He and Wenhai Zheng for their critical reading of early versions of the text.

Preface to Second Edition

The aim of the second edition remains the same as the first: to provide science and engineering students a practical introduction to technical programming in Python. This new edition adds nearly 100 pages of new material.

Among the changes, the concept of an object is developed more thoroughly, starting in Chapter 2 with the discussion of variables and assignment. This perspective is continued throughout the text as the various aspects of objects are revealed and developed. The chapter on Python classes, now Chapter 10, has been completely rewritten with new examples. Here, we emphasize the concept of encapsulation and its use in science and engineering.

Chapter 3 on the Spyder and Jupyter Lab integrated development environments (IDEs) is new. Some of the material on the Spyder IDE can be found in the First Edition, but it has been updated and expanded in this edition. The material on the Jupyter Lab IDE is entirely new, as Jupyter Lab has developed significantly since the first edition and now offers a compelling IDE.

New examples have been added to Chapter 6 on conditionals and loops. The chapter also includes a new section on exception handling.

The introduction of functions has been moved so that it now occurs before the chapter on plotting. Type hints, new to Python since the first edition, are discussed. The subtle subject of namespace and scope and its relation to functions has been expanded significantly.

The chapter on curve fitting has been eliminated. That material is now covered in Chapters 7 and 9.

New material has been added to Chapter 8 on plotting, including an introduction to the Seaborn package. New examples have been added, including using two separate scales for a single axis, plots with insets, vector field (quiver) plots, and plotting with polar coordinates.

Chapter 9 on the NumPy and SciPy packages has been expanded to include new material on interpolating and smoothing splines. Several updates in various NumPy and SciPy packages have been incorporated into the text, including changes in NumPy's random number and polynomial packages.

Chapter 13 on speeding up numerical computations is new. It focuses on the Numba package and how to effectively use it to address Python's Achilles' heel, its slow execution of long loops involving numerical code.

The programs and data files introduced in the Second Edition are available at https://github.com/djpine/python-scieng-public-2.

The paper edition is printed in grayscale to reduce costs. However, the original figures in full color are available at https://github.com/djpine/python-scieng-public-2.

In addition to those who contributed to the First Edition, I would like to thank Marc Gershow for helpful suggestions, Fan Cui for initial versions code presented in Chapter 13, and Xinhang Shen for providing data used in Chapter 8.

About the Author

David Pine has taught Physics and Chemical Engineering for more than 40 years at four different institutions: Cornell University (as a graduate student), Haverford College, UCSB, and NYU, where he is a Professor of Physics, Mathematics, and Chemical and Biomolecular Engineering. He has taught a broad spectrum of courses, including numerical methods. He does research on optical materials and soft-matter physics, which is concerned with materials such as polymers, emulsions, and colloids.

CHAPTER 1

Introduction

1.1 INTRODUCTION TO PYTHON FOR SCIENCE AND ENGINEERING

This book is meant to serve as an introduction to the Python programming language and its use for scientific computing. It's ok if you have never programmed a computer before. This book will teach you how to do it from the ground up.

Python is well suited for most scientific and engineering computing tasks. You can use it to analyze and plot data. You can also use it to numerically solve science and engineering problems that are difficult or even impossible to solve analytically.

While we want to marshal Python's powers to address scientific problems, you should know that Python is a general-purpose computer language widely used for a broad spectrum of computing tasks, from web applications to processing financial data on Wall Street and various scripting tasks for computer system management. Over the past decade, it has been increasingly used by scientists and engineers for numerical computations and graphics and as a "wrapper" for numerical software originally written in other languages, like Fortran and C.

Python is similar to MATLAB®, another computer language frequently used in science and engineering applications. Like MATLAB®, Python is an *interpreted* language, meaning you can run your code without going through an extra step of compiling, as required for the C and Fortran programming languages. It is also a *dynamically typed language*, meaning you don't have to declare variables and set aside memory before using them.

DOI: 10.1201/9781032673950-1

Don't worry if you don't know exactly what these terms mean.[1] Their primary significance for you is that you can write Python code, test it, and use it quickly with a minimum of fuss.

One advantage of Python compared to MATLAB® is that it is free. It can be downloaded from the web and is available on all the standard computer platforms, including Windows, macOS, and Linux. This also means that you can use Python without being tethered to the internet, as required for commercial software tied to a remote license server.

Another advantage is Python's clean and simple syntax, including its implementation of *object-oriented* programming. This should not be discounted; Python's rich and elegant syntax renders many tasks that are difficult or arcane in other languages more straightforward and understandable in Python.

A significant disadvantage is that Python programs can be slower than compiled languages like C. For large-scale simulations and other demanding applications, there can be a considerable speed penalty in using Python. In these cases, C, C++, or Fortran are recommended, although intelligent use of Python's array processing tools in the NumPy module can significantly speed up Python code. Alternatively, several new tools have recently appeared that can be used to speed up certain numerical computations in Python significantly, often by one or two orders of magnitude. These are discussed in Chapter 13. Another disadvantage is that, compared to MATLAB®, Python is less well-documented. This stems from the fact that it is public *open source* software and thus depends on volunteers from the community of developers and users for documentation. The documentation is freely available on the web but is scattered among a number of different sites and can be terse. This book will acquaint you with the most commonly used websites. Search engines like Google can help you find others.

You are not assumed to have had any previous programming experience. However, the purpose of this manual isn't to teach you the principles of computer programming; it's to provide a very practical guide to getting started with Python for scientific computing. Once you see some of the powerful tasks you can accomplish with Python, perhaps you will be inspired to study computational science and engineering, as well as computer programming, in greater depth.

[1] Appendix B contains a glossary of terms you may find helpful.

1.2 INSTALLING PYTHON

You need to install Python and four scientific Python libraries for scientific programming with Python: NumPy, SciPy, Matplotlib, and Pandas. You can install many other useful libraries, but these four are the most widely used and are the only ones you will need for this text.

There are several ways to install Python and the necessary scientific libraries. Some are easier than others. For most people, the simplest way to install Python and all the scientific libraries you need is to use the *Anaconda* distribution, which includes the JupyterLab and Spyder integrated development environments (IDEs) for Python. These IDEs are introduced in Chapter 3.

The Anaconda distribution package can be found at the website https://www.anaconda.com/download/. Once you download and install it, you can use the **Anaconda-Navigator** application to launch all of the applications introduced in this text, including Spyder, JupyterLab, and Qt Console.

Now, you are ready to go.

CHAPTER 2

Launching Python

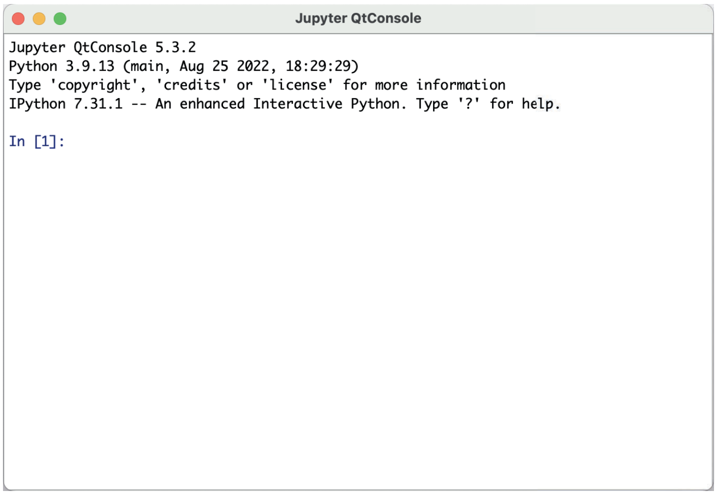
In this chapter, you learn about ***IPython****, an interface that allows you to use Python interactively with tools optimized for mathematical and computational tasks. You learn how to use IPython as a calculator to add, subtract, multiply, divide, and perform other common mathematical functions. You also learn the basic elements of the Python programming language, including* ***functions****,* ***variables****, and* ***scripts****, which are rudimentary computer programs. You are introduced to Python* ***modules****, which extend the capabilities of the core Python language and allow you to perform advanced mathematical tasks. You also learn some new ways to navigate your computer's file directories. Finally, you learn how to get help with Python commands and functions.*

2.1 INTERACTING WITH PYTHON: THE IPython SHELL

There are many different ways to interact with Python. For simple computing tasks, people typically use the *Python command shell*, which is also called the *Python interpreter* or *console*. A shell or console is just a window on your computer that you use to issue written commands from the keyboard. For scientific Python, which is the focus of this text, people generally use the *IPython* shell (or console) instead of the Python shell. The IPython shell is specifically designed for scientific and engineering use. We use the IPython shell throughout this text.

To launch a Python or IPython shell, you first need to launch a terminal application. If you are running the macOS, launch the **Terminal** application, which you can find in the **Applications/Utilities** folder on your computer. If

DOI: 10.1201/9781032673950-2



```
Jupyter QtConsole 5.3.2
Python 3.9.13 (main, Aug 25 2022, 18:29:29)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.31.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Figure 2.1 Qt console for IPython.

you are running Windows, launch the **Anaconda Powershell Prompt** application from the **Start** menu. Under Linux, you can open the **Terminal** application by pressing <**ctrl** + **alt** + **T**>.

After launching a terminal application, type `jupyter qtconsole` at the terminal prompt and press <**return**>. This launches a particularly powerful version of the IPython shell called the *Qt Console*. Alternatively, instead of launching a terminal, you can launch Qt Console directly from the **Anaconda-Launcher** app that is downloaded with the Anaconda Python Distribution. The Qt Console for IPython will be used throughout this text. It should look like the window in Figure 2.1. You should see the default input prompt of the IPython shell, which looks like this:

```
In[1]:
```

Now you are ready to go.

By the way, if you type `ipython` at the terminal prompt, you will get a similar but less powerful IPython shell. This is not what you want. If you type `python` at the terminal prompt, you get the standard Python shell with the prompt:

```
>>>
```

This is not what you want either. Type `quit()` after the >>> prompt to quit the Python shell and return to the system terminal. By the way, you can also type `quit()` to quit either of the IPython shells and return to the system terminal.

2.2 THE IPython SHELL

The IPython prompt, shown here,

```
In [1]:
```

indicates that the **IPython** shell is running and ready to receive input from the user (you!). By typing commands at the prompt, IPython can be used to perform various tasks, such as running programs, performing arithmetic, and creating and moving files around on your computer.

Before getting started, we note that, like most modern computer languages, Python is *case sensitive*. That is, Python distinguishes between upper- and lower-case letters. Thus, two words spelled the same but having different letters capitalized are treated as different names in Python. Please keep that in mind in all that follows.

2.3 INTERACTIVE PYTHON AS A CALCULATOR

Let's get started. You can use the IPython shell to perform simple arithmetic calculations. For example, to find the product 3×15, you type `3*15` (or `3 * 15`, spaces don't matter) at the `In` prompt and press <**return**>:

```
In [1]: 3 * 15
Out[1]: 45
```

Python returns the correct product, as expected. You can do more complicated calculations:

```
In [2]: 6 + 21 / 3
Out[2]: 13.0
```

Let's try some more arithmetic:

```
In [3]: (6 + 21) / 3
Out[3]: 9.0
```

Notice that the effect of the parentheses in `In[3]: (6 + 21) / 3` is to cause the addition to be performed first and then the division. Without the parentheses, Python will always perform the multiplication and division operations *before* performing the addition and subtraction operations. The order in which arithmetic operations are performed is the same as for most calculators: exponentiation first, then multiplication or division, then addition or subtraction, then left to right.

TABLE 2.1 Binary operators.

Operation	Symbol	Example	Output
Addition	`+`	`19 + 7`	`26`
Subtraction	`-`	`19 - 7`	`12`
Multiplication	`*`	`19 * 7`	`133`
Division	`/`	`19 / 7`	`2.7142857142`
Floor division	`//`	`19 // 7`	`2`
Remainder	`%`	`19 % 7`	`5`
Exponentiation	`**`	`19**7`	`893871739`

2.3.1 Binary Arithmetic Operations in Python

Table 2.1 lists the binary arithmetic operations in Python. Python has all the standard binary operators for arithmetic, plus a few you may not have seen before.

"Floor division," designated by `//`, means divide and keep only the integer part without rounding. "Remainder," designated by the symbol `%`, gives the remainder after floor division.

2.3.2 Types of Numbers

There are three different types of numbers in Python: Integers, floating point numbers, and complex numbers.

1. **Integers** in Python are simply, as their name implies, integers. They can be positive or negative and can be arbitrarily long. In Python, a number is automatically treated as an integer if it is written without a decimal point. This means that `23`, written without a decimal point, is an integer, and `23.`, written with a decimal point, is a floating point number. Here are some examples of integer arithmetic:

```
 In [4]: 12 * 3
Out [4]: 36

 In [5]: 4 + 5 * 6 - (21 * 8)
Out [5]: -134

 In [6]: 11 / 5
Out [6]: 2.2

 In [7]: 11 // 5     # floor divide
Out [7]: 2
```

```
 In [8]: 9734828*79372
Out[8]: 772672768016
```

For the binary operators +, -, *, and //, the output is an integer if the inputs are integers. The output of the division operator / is a floating point number (as of version 3 of Python). The floor division operator // must be used if an integer output is desired when dividing two integers.

2. **Floating point** numbers are essentially rational numbers and can have a fractional part; integers, by their very nature, have no fractional part. In most versions of Python, floating point numbers go between approximately $\pm 2 \times 10^{-308}$ and $\pm 2 \times 10^{308}$. Here are some examples of floating point arithmetic:

```
 In [9]: 12. * 3
Out[9]: 36.0

 In [10]: 12 / 3.
Out[10]: 4.0

 In [11]: 5 ** 0.5
Out[11]: 2.23606797749979

 In [12]: 5 ** (1/2)
Out[12]: 2.23606797749979

 In [13]: 11. / 5.
Out[13]: 2.2

 In [14]: 11. // 5.
Out[14]: 2.0

 In [15]: 11. % 5.
Out[15]: 1.0

 In [16]: 6.022e23 * 300.
Out[16]: 1.8066e+26
```

Note that the result of any operation involving only floating point numbers as inputs is another floating point number, even in cases where the floor division // or remainder % operators are used. The last output illustrates an alternative way of writing floating point numbers as a mantissa followed by e or E followed by a power of 10: so 1.23e-12 is equivalent to 1.23×10^{-12}.

Notice also that multiplying or dividing a floating point number by an integer produces a floating point number.

We used the exponentiation operator `**` to find the square root of 5 by using a fractional power of `0.5` and `1/2`. In Section 2.6.2, an alternative method is presented for finding the square root of a number.

3. **Complex numbers** are written in Python as a sum of real and imaginary parts. For example, the complex number $3-2i$ is represented as `3-2j` in Python, where `j` represents $\sqrt{-1}$. Here are some examples of complex arithmetic: `2+3j * -4+9j = 2+(3j * -4)+9j = (2-3j)`, whereas `(2+3j) * (-4+9j) = (-35+6j)`.

```
 In [17]:  (2+3j) * (-4+9j)
Out [17]:  (-35+6j)

 In [18]:  (2+3j) / (-4+9j)
Out [18]:  (0.1958762886597938-0.3092783505154639j)

 In [19]:  2.5-3j**2
Out [19]:  (11.5+0j)

 In [20]:  (2.5-3j)**2
Out [20]:  (-2.75-15j)
```

Notice that `2.5-3j**2` and `(2.5-3j)**2` give *different* results. You need to enclose the real and imaginary parts of a complex number in parentheses if you want exponentiation to operate on the entire complex number and not simply on the imaginary part. It works similarly with multiplication and division of complex numbers, so be sure to enclose the entire complex number with parentheses if you wish to multiply or divide complex numbers.

If you multiply an integer by a floating point number, the result is a floating point number. If you multiply a floating point number by a complex number, the result is a complex number. Python promotes the result to the most complex of the inputs.

2.3.3 Numbers as Objects

Everything in Python is an *object*. Thus, the numbers we introduced above are all objects. We will not fully define what an object is right now; we will explain it little by little as needed as we proceed.

The first thing to know about objects is that they are the fundamental things that Python manipulates and works with. As such, they have some interesting properties, a few of which we explore here. For example, each object has an ID, which is just its location in your computer's memory. We can determine the ID of an object using Python's `id` function.

```
 In[21]: id(52)
Out[21]: 140294301609872

 In[22]: id(241.3)
Out[22]: 140293508419248

 In[23]: id(3+7j)
Out[23]: 140293508416368
```

The next thing to know is that every object has a *type*, which can be ascertained using the function `type`. So, what are the types of the numbers we introduced above?

```
 In[24]: type(72)
Out[24]: int

 In[25]: type(-11.4)
Out[25]: float

 In[26]: type(3-36j)
Out[26]: complex
```

The results are not too surprising: `int` for integers, `float` for floating point numbers, and `complex` for complex numbers.

An object's type defines how it interacts with other objects. For example, you can freely add, subtract, multiply, and divide objects of the types `int`, `float`, and `complex`, as illustrated above. On the other hand, you can't multiply `float` and `complex` types by a string type such as "dog" (we introduce strings in Chapter 4). Trying to do so will result in an error message. Surprisingly, you can multiply strings by `int` types, but we defer that discussion to Section 4.1.

2.4 VARIABLES AND ASSIGNMENT

2.4.1 Names and the Assignment Operator

A variable is a way of associating a name with an object. Thus, when we write

```
In[1]: a = 32
```

Python binds the variable name `a` to the integer object `32`. The equals sign `=` is the *assignment operator*, and its function is to bind the variable name on the left side to the object on its right side.

Consider the following code:

```
In[2]: leg_a = 3.7
```

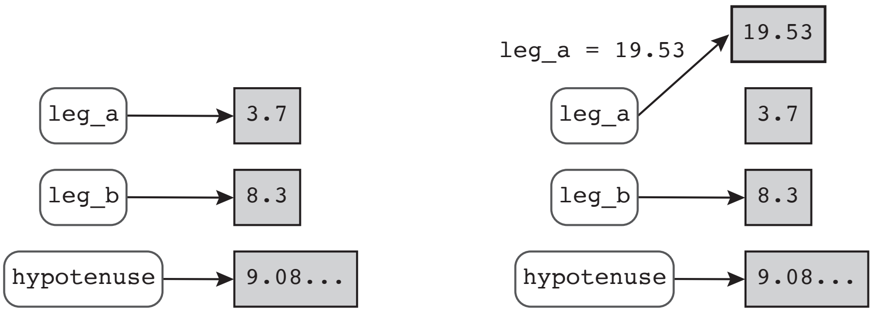


Figure 2.2 Binding variable names to objects.

```
In[3]: leg_b = 8.3

In[4]: hypotenuse = (leg_a**2 + leg_b**2)**0.5

In[5]: hypontenuse
Out[5]: 9.087353850269066
```

The first two statements bind the variable names `leg_a` and `leg_b` to the float objects `3.7` and `8.3`, respectively. The third statement performs the calculation to the right of the equals sign and then binds the variable name `hypotenuse` to the resulting float object `9.087353850269066`. Note that Python binds the result of the calculation, not the calculation itself, to the variable `hypotenuse`. Therefore, if we reassign the value of `leg_a` to a new value, the value of `hypotenuse` does not change, as demonstrated here.

```
In[6]: leg_a = 19.53

In[7]: hypotenuse
Out[7]: 9.087353850269066
```

When we write `leg_a = 19.53`, Python reassigns the variable name `leg_a` to a new float object `19.53`, as illustrated in Figure 2.2. The old object, in this case, the float `3.7`, is still in memory. Eventually, Python gets rid of it to free up memory; this process is called *garbage collection* and occurs behind the scenes so that you do not need to worry about it.

The assignment operator "=" in Python is not equivalent to the equals sign "=" you are accustomed to in algebra. Consider the following sequence of commands.

```
In[8]: a = 5
```

```
 In [9]: a = a + 2

 In [10]: a
Out[10]: 7
```

The statement `a = a + 2` makes no sense in algebra. But it makes perfect sense in Python (and in most computer languages). It means take the current value of `a`, add 2 to it, and assign the result to the variable name `a`. Python reassigns the variable name `a` to a new object, with a value of 7 in this case.

This construction appears so often in programming that there is a special set of operators dedicated to performing such changes to a variable: `+=`, `-=`, `*=`, and `/=`. For example, `a = a + 2` and `a += 2` do the same thing; they add 2 to the current value of `a`. Here are some other examples of how these operators work:

```
 In [11]: c = 4

 In [12]: c += 3

 In [13]: c
Out[13]: 7

 In [14]: c *= 3

 In [15]: c
Out[15]: 21

 In [16]: d = 7.92

 In [17]: d /= -2

 In [18]: d
Out[18]: -3.96

 In [19]: d -= 4

 In [20]: d
Out[20]: -7.96
```

By the way, `%=`, `**=`, and `//=`, are also valid operators. Verify in the IPython console that you understand how the above operations work.

Python also allows you to make multiple variable assignments in a single statement

```
 In [21]: p, q, r = 32.1, 81.6, 111.6
```

is equivalent to `p = 32.1`, `q = 81.6`, and `r = 111.6`. Having made that assignment, what do you think `p, q, r = r, p, q` does? Try it out for yourself and see if you were able to predict the correct results. The key thing to remember is that Python evaluates the right-hand side of the equation before assigning the results to the left-hand side.

Finally, please note that the same object can have multiple names. For example, in the following code, `a` and `b` point to (*i.e.*, are bound to) the same object, which you can verify by checking the ID of each of them.

```
 In[22]: a = b = 3.4

 In[23]: id(a)
Out[23]: 140293508599056

 In[24]: id(b)
Out[24]: 140293508599056

 In[25]: b = 5.8

 In[26]: id(b)
Out[26]: 140293508598928

 In[27]: id(a)
Out[27]: 140293508599056
```

Reassigning `b` to a different value creates a new object with a new ID distinct from the ID of `a`, which remains the same after the reassignment of `b`.

We also point out here that an object doesn't have to have a name associated with it. For example, when we write

```
 In[28]: 5 * 6
Out[28]: 30
```

none of the objects for 5, 6, or 30 have names. They are all integer objects. Not having variable names associated with them, we say that they are integer *literals*.

2.4.1.1 Python Variables are Dynamically Typed

By the way, suppose that in the last step above, we had written `b = 6` instead of `b = 5.8`. Before trying this out, let's first check the variable `b`'s type. Then we will reassign the value of `b` by typing `b = 6`.

```
 In[29]: type(b)
Out[29]: float
 In[30]: b = 6

 In[31]: type(b)
Out[31]: int
```

Notice that after we wrote `b = 6`, its type changed from `float` to `int`. This feature of Python is called *dynamical typing*. A variable's type can change on the fly. That's all we'll say about dynamic typing for now, but we will return to this topic.

2.4.2 Legal and Recommended Variable Names

Variable names in Python must start with a letter or an underscore "_" and can be followed by as many alphanumeric characters as you like, including the underscore character "_". Spaces are not allowed in variable names. No other character that is not a letter, number, or underscore is permitted.

Although variable names can start with the underscore character, you should avoid doing so except in special cases, which we discuss in Chapter 10.

Recall that Python is *case sensitive*, so the variable `velocity` is distinct from the variable `veLocity`. We recommend giving your variables descriptive names as in the following calculation:

```
In [32]: distance = 34.

In [33]: time_traveled = 0.59

In [34]: velocity = distance / time_traveled

In [35]: velocity
Out[35]: 57.6271186440678
```

Giving variables descriptive names serves two purposes. First, it makes the code (to some extent) self-documenting so that you or another reader of the code can get some idea about what it does. The variable names `distance`, `time_traveled`, and `velocity` immediately remind you of what is being calculated here. Second, it can help you catch errors; if we had written `velocity = time_traveled / distance`, you might be more likely to notice that something's amiss. So, using descriptive variable names is good practice. But so is keeping variable names reasonably short, so don't go nuts! When using two words for a variable name, it's considered good practice in Python to connect the two words with an underscore (*e.g.*, `time_traveled`).

2.4.3 Reserved Words in Python

Python reserves certain names or words for special purposes. These names are provided in Table 2.2 for your reference. You must avoid using these names as variables.

TABLE 2.2 Reserved names in Python.

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

2.5 SCRIPT FILES AND PROGRAMS

Performing calculations in the IPython shell is handy if the calculations are short. But calculations quickly become tedious when they are over a few lines long. If you discover that you made a mistake at some early step, for example, you may have to go back and retype all the steps subsequent to the error. Having code saved in a file means you can correct the error and rerun the code without having to retype it. Saving code can also be useful if you want to reuse it later, perhaps with different inputs.

For these and many other reasons, we save code in computer files. The sequence of Python commands stored in a file is called a *script* or a *program* or sometimes a *routine*. Programs can become quite sophisticated and complex. In this chapter, we introduce only the simplest features of programming by writing a very simple script. Later, we will introduce some of the more advanced features of programming.

2.5.1 Editors for Python Scripts

Python scripts are just plain text files. The only requirement is that the filename ends with `.py`. Because they are just plain text files, you can write Python scripts using any simple text editor. No special editor is required. Some editors, however, automatically recognize any file whose name ends in the suffix `.py` as a Python file. For example, the text editors *Notepad++* (for PCs), *BBEdit* (for Macs), and *Gedit* (for Linux) automatically recognize Python files. All three editors work very well and are available without charge on the internet. These editors are nice because they color code the Python syntax, a helpful feature called *syntax highlighting*. They also have other programming-specific features that make the files easy to read and edit. Other editors have similar features, and they also work very well.

Note, however, that word processing programs like Microsoft Word® are not suitable for this purpose because they produce files that, in addition to the visible text, contain all sorts of formatting code that is invisible to the user but not the computer (or the Python interpreter). You must use a plain text editor.

2.5.1.1 Create a Directory (Folder) for Python Scripts

You should create a directory (also known as a folder) on your computer to store your Python scripts. For example, you might create a directory called **PyScripts** inside the **Documents** directory. If possible, choose a filename with no spaces in it. Avoiding spaces in directory names is not absolutely necessary but, as you will see below, it can simplify navigation between directories.

2.5.2 First Scripting Example

Let's work through an example to see how scripting works. Suppose you are going on a road trip and would like to estimate how long the drive will take, how much electricity you will need (you're driving an EV), and the cost of the electricity. It's a simple calculation. As inputs, you will need the trip's distance, your average speed, the cost of electricity, and the mileage (average miles per kilowatt-hour) for your car.

Writing a script to do these calculations is straightforward. First, launch a text editor, which can be one of the three mentioned above, Notepad++, BBEdit, or Gedit, depending on your operating system, or some other text editor of your choosing. Enter the following code and *save the code*. Do not include the small numbers 1–10 in the left margin. These are just for reference and are not part of the Python code.

Code: my_trip.py

```
 1  """Calculates time, electrical energy used, and cost of electricity
 2  for a trip in an electric vehicle"""
 3  # Get inputs
 4  distance = 180.        # [miles]
 5  mpk = 3.9              # [miles/kilowatt-h] car mileage
 6  speed = 60.            # [miles/h] average speed
 7  cost_per_kWh = 0.22    # [$/kW-h] price of electricity
 8
 9  # Calculate outputs
10  time = distance / speed        # [hours]
11  energy = distance / mpk        # [kW-h]
12  cost = energy * cost_per_kWh   # [$]
```

Save the file with the name `my_trip.py` in the directory **PyScripts** that you created earlier (see Section 2.5.1.1). This stores your script (or program) on your computer's disk. More generally, the name of a Python file can be almost anything consistent with the computer operating system as long the name ends with the extension `.py`. The `.py` extension tells the computer this is a Python program.

The code in the program is pretty straightforward: lines 4–7 set the values of the inputs, while lines 10–12 calculate the desired information. All of the variables are floats by virtue of the decimal point included in each assignment statement. Notice that we included a blank line, line 8, between the input and output blocks of code. This is not necessary, as the blank line serves no computational purpose. Rather, it indicates to the reader that the blocks do different things, analogous to what paragraphs do in normal written text.

The text between the triple quotes at the beginning of the program is called a "docstring" and is not executed when the script is run. Everything between the triple quotes is part of the docstring, which can extend over multiple lines, as it does here. It's a good idea to include a docstring explaining what your script does at the beginning of your file.

The hash (or number) symbol # is the "comment" character in Python; anything on a line following # is ignored when the code is executed. A comment in a Python script is a brief explanation or annotation added to help people reading the program understand what the program is doing. Judicious use of comments in your code will make it much easier to understand days, weeks, or months after you write it. Use comments generously. For aesthetic reasons, the comments on different lines have been aligned. This isn't necessary. The spaces needed to align the comments have no effect on the running of the code.

Now you are ready to run the code. From a QtConsole, type

```
In[1]: run ~/Documents/PyScripts/my_trip.py
```

The string of text `~/Documents/PyScripts/` tells IPython where your script is located on your computer: the tilde `~` designates the user's home directory, which might correspond to something like **/Users/dp** on a macOS or Linux computer, or **C:\Users\dp** on a Windows computer. You don't need to write out the name of your home directory; just writing `~` will do. Next comes the hierarchy of directories, each separated by a forward slash (for any operating system, MacOs, Windows, or Linux), where the Python script is located, and then finally the name of the file `my_trip.py` containing the script. If a directory name contains one or more spaces, such as **My Files**, for example, then on Mac you should write `cd ~/Documents/My\ Files` or `cd "~/Documents/My Files"`. That is, you can either replace each space by a backslash and a space or you can enclose the entire path in quotes, either single `'` or double `"`. On a PC, you can include a space without taking any special measures.

When you run a script, Python executes the sequence of commands in the order they appear. Afterward, you can see the values of the variables calculated in the script by typing the name of the variable. IPython responds with the value of that variable. For example:

```
 In[2]: time
Out[2]:  3.0

 In[3]: energy
Out[3]: 46.15384615384615

 In[4]: cost
Out[4]: 10.153846153846153
```

Of course, you must remember that the time is in hours, and the cost is in U.S. dollars.

You can change the number of digits IPython displays using the command `%precision`. To display two digits to the right of the decimal place, enter `%precision 2`:

```
 In[5]: %precision 2
Out[5]: ' %.2f'

 In[6]: time
Out[6]: 3.00

 In[7]: energy
Out[7]: 46.15

 In[8]: cost
Out[8]: 10.15
```

Typing `%precision` returns IPython to its default state; `%precision %e` causes IPython to display numbers in exponential format (scientific notation).

2.5.2.1 Note about Printing

If you want your script to return the value of a variable (that is, print the value of the variable to your computer screen), use the `print` function. For example, at the end of our script, if we include the code

```
print(time)
print(energy)
print(cost)
```

the script will return the values of the variables `time`, `gallons`, and `cost` that the script calculated. We will discuss the `print` function in much greater detail, as well as other methods for data output, in Chapter 5.

2.6 PYTHON MODULES

The Python computer language consists of a "core" language plus a vast collection of supplementary software that is contained in **modules** (or packages, which are collections of modules—we'll not fuss about the distinction here). Many of these modules come with the standard Python distribution and provide added functionality for performing computer system tasks. Other modules provide more specialized capabilities that only some users may want. These modules are a kind of library from which you can borrow according to your needs. You gain access to a module using the `import` command, which we introduce in the next section.

We will need four Python modules that are not part of the core Python distribution but are widely used for scientific computing. The four modules are:

NumPy is the standard Python package for scientific computing with Python. It provides the all-important NumPy `array` data structure, which is at the very heart of NumPy. It also provides tools for creating and manipulating arrays, including indexing and sorting, as well as basic logical operations and element-by-element arithmetic operations like addition, subtraction, multiplication, division, and exponentiation. It includes the basic mathematical functions of trigonometry, exponentials, and logarithms, as well as a vast collection of special functions (Bessel functions, *etc.*), statistical functions, and random number generators. It also includes many linear algebra routines that overlap with those in SciPy, although the SciPy routines tend to be more comprehensive. You can find more information about NumPy at http://docs.scipy.org/doc/numpy/reference/index.html.

SciPy provides a broad spectrum of mathematical functions and numerical routines for Python. SciPy makes extensive use of NumPy arrays, so when you import SciPy, you should always import NumPy too. In addition to providing basic mathematical functions, SciPy provides Python "wrappers" for numerical software written in other languages, like Fortran, C, or C++. A "wrapper" provides a transparent easy-to-use Python interface to standard numerical software, such as routines for performing curve fitting and numerically solving differential equations. SciPy dramatically extends the power of Python and saves you the trouble of writing software in Python that someone else has already written and optimized in some other language. You can find more information about SciPy at http://docs.scipy.org/doc/scipy/reference/.

Matplotlib is the standard Python package for making two- and three-dimensional plots. Matplotlib makes extensive use of NumPy arrays. All of the plots in this book use this package. You can find more information about Matplotlib at the website http://matplotlib.sourceforge.net/.

Pandas is a Python package providing a powerful set of data analysis tools. It uses data structures similar to those used in a spreadsheet program like Excel and allows you to manipulate data in ways similar to spreadsheets. You can find more information about Pandas at http://pandas.pydata.org/.

We will use these four modules extensively and, therefore, will provide introductions to their capabilities as we develop Python. The links above provide much more extensive information; you will certainly want to refer to them occasionally.

2.6.1 Python Modules and Functions: A First Look

Because the modules listed above, NumPy, SciPy, Matplotlib, and Pandas, are not part of core Python, they must be imported before we can access their functions and data structures. Here, we show how to import the NumPy module and use some of its functions. We defer introducing NumPy arrays, mentioned in the previous section, until Section 4.4.

You gain access to the NumPy package using Python's `import` statement:

```
 In[1]: import numpy
```

After running this statement, you can access all the functions and data structures of NumPy. For example, you can now access NumPy's sine function as follows:

```
 In[2]: numpy.sin(0.5)
Out[2]: 0.479425538604203
```

In this simple example, the `sin` function has one argument, here `0.5`, and the function returns the sine of that argument, which must be expressed in radians.

Note that we had to put the prefix `numpy` dot before the name of the actual function name `sin`. This tells Python that the `sin` function is part of the NumPy module that we just imported.

Another Python module called `math` also has a sine function. We can import the math module just like we imported the NumPy module:

```
 In[3]: import math
 In[4]: math.sin(0.5)
Out[4]: 0.479425538604203
```

These two sine functions are not the same, even though they give the same answer in this case. Consider, for example, what happens if we ask each function to find the sine of a complex number:

```
 In[5]: numpy.sin(3+4j)
Out[5]: (3.853738037919377-27.016813258003932j)

 In[6]: math.sin(3+4j)
```

```
-------------------------------------------------------------
TypeError                         Traceback (most recent call last)
<ipython-input-24-b48edfeaf02a> in <module>()
----> 1 math.sin(3+4j)

TypeError: can't convert complex to float
```

The NumPy sine function works just fine and returns a complex result. By contrast, the `math` sine function returns an error message because it does not accept a complex argument. In fact, the `math` sine function accepts only a single real number as an argument while the `numpy` sine function accepts real and complex NumPy arrays, which we introduce in Section 4.4, as arguments. For single real arguments, the `math` sine function executes faster than the `numpy` function, but the difference in execution speed is not noticeable in most cases.

The important lesson here is to appreciate how Python allows you to extend its capabilities by importing additional packages, while at the same time keeping track of where these capabilities come from using the prefix dot syntax. By using different prefixes, each module maintains its own *namespace*, that is, a separate dictionary of names, so that functions with the same name in different packages do not clash.

If you are using a lot of NumPy functions, writing out `numpy` dot before each function can be a little lengthy. Python allows you to define an abbreviation for the prefix when you import a library. Here we show how to do it for NumPy:

```
 In[7]: import numpy as np

 In[8]: np.sin(0.5)
Out[8]: 0.47942553860420301
```

The statement `import numpy as np` imports and assigns the abbreviation `np` for `numpy`. In principle, you can use any abbreviation you wish. However, using `np` for the NumPy module is common practice. You are strongly encouraged to abide by this practice so that others reading your code will recognize what you are doing.

TABLE 2.3 Some `numpy` and `math` functions. Functions work for both the `numpy` and `math` packages unless explicitly indicated.

Function	Description
`sqrt(x)`	square root of x
`exp(x)`	exponential of x, *i.e.*, e^x
`log(x)`	natural log of x, *i.e.*, $\ln x$
`log10(x)`	base 10 log of x
`degrees(x)`	converts x from radians to degrees
`radians(x)`	converts x from degrees to radians
`sin(x)`	sine of x (x in radians)
`cos(x)`	cosine x (x in radians)
`tan(x)`	tangent x (x in radians)
`arcsin(x), math.asin(x)`	Arc sine (in radians) of x
`np.arccos(x), math.acos(x)`	arc cosine (in radians) of x
`arctan(x), math.atan(x)`	arc tangent (in radians) of x
`fabs(x)`	absolute value of x
`math.factorial(n)`	$n!$ of an integer
`np.round(x)`	rounds float to nearest integer
`floor(x)`	rounds float *down* to nearest integer
`ceil(x)`	rounds float *up* to nearest integer
`np.sign(x)`	-1 if $x < 0$, $+1$ if $x > 0$, 0 if $x = 0$

2.6.2 Some NumPy Functions

NumPy includes an extensive library of mathematical functions. In Table 2.3, we list some of the most useful ones. A complete list is available at https://docs.scipy.org/doc/numpy/reference/.

The argument of these functions can be a number or any expression whose output produces a number. All of the following expressions are legal and produce the expected output:

```
 In[9]: np.log(179.2)
Out[9]: 5.18850250054083

 In[10]: np.log(np.sin(0.5))
Out[10]: -0.7351666863853142

 In[11]: np.log(np.sin(0.5)+1.0)
Out[11]: 0.39165386283471759

 In[12]: np.log(5.5/1.2)
Out[12]: 1.5224265354444708
```

Here, we have demonstrated functions with one input and one output. In general, Python functions have multiple inputs and multiple outputs. We will discuss these and other features of functions later when we take up functions in the context of user-defined functions.

2.6.3 Scripting Example 2

Let's try another problem. Suppose you want to find the distance between two Cartesian coordinates (x_1, y_1, z_1) and (x_2, y_2, z_2). The distance is given by the formula

$$\Delta r = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

Let's write a script to do this calculation and save it in the subdirectory `~/Documents/PyScripts/` a file called `two_point_distance.py`.

Code: two_point_distance.py

```python
"""Calculates the distance between two 3d Cartesian coordinates"""
import numpy as np

x1, y1, z1 = 23.7, -9.2, -7.8
x2, y2, z2 = -3.5, 4.8, 8.1

dr = np.sqrt((x2 - x1)**2 + (y2 - y1)**2 + (z2 - z1)**2)
```

We have introduced extra spaces into some of the expressions to improve readability. They are not necessary; where and whether you include them is largely a matter of taste.

Because we need the square root function of NumPy, the script imports NumPy before doing anything else. If you leave out the "`import numpy as np`" line or remove the `np` dot in front of the `sqrt` function, you will get the following error message:

```
Traceback (most recent call last):
...
File ".../two_point_distance.py", line 8, in <module>
dr = sqrt((x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2)

NameError: name "sqrt" is not defined
```

Now, with the `import numpy as np` statement, we can run the script.

```
In[13]: %run ~/Documents/PyScripts/two_point_distance.py

In[14]: dr
Out[14]: 34.476803796175766
```

The script works as expected. Or does it? It might be a good idea to check it out by running it for inputs for which you know the answer. For example, suppose the first point is at $(0,0,0)$ and the second point is at $(1,1,1)$. Then we know the distance between these two points is $\sqrt{3}$. Go ahead and modify the program and verify that it gives the expected output for these two inputs. This is an excellent way to check whether the program behaves as expected.

2.6.4 Different Ways of Importing Modules

There are different ways that you can import modules in Python.

2.6.4.1 Importing an Entire Module

Usually we import entire modules using the `import` statement or the `import ...as ...` statement that we introduced for the Math and NumPy libraries:

```
        import math
        import numpy as np
```

2.6.4.2 Importing Part of a Module

You can also import a single function or subset of functions from a module without importing the entire module. For example, suppose you wanted to import just the `log` function from NumPy. You could write

```
from numpy import log
```

To use the `log` function in a script, you would write

```
a = log(5)
```

which would assign the value `1.6094379124341003` to the variable `a`. If you wanted to import the three functions, `log`, `sin`, and `cos`, you would write

```
from numpy import log, sin, cos
```

Imported in this way, you would use them without a prefix, as the functions are imported into the general namespace of the program. In general, we do not recommend using `from` *module* `import` *functions* way of importing functions. When reading code, it makes it harder to determine from which modules functions are imported, and can lead to clashes between similarly named functions from different modules. Nevertheless, we do use this form sometimes, and, more importantly, you will see the form used in programs you encounter on the web and elsewhere, so it is essential to understand the syntax.

2.6.4.3 *Blanket Importing of a Module*

There is yet another way of importing an entire module by writing

```
from numpy import *
```

This imports the entire module, in this case NumPy, into the general namespace and allows you to use all the functions in the module without a prefix. If you import two different libraries this way in the same script, then it's impossible to tell which functions come from which library by just looking at the script. You also have the aforementioned problem of clashes between libraries, so you are strongly advised not to import this way in a script or program.

There is one possible exception to this advice, however. When working in the IPython shell, you often just want to try out a function or a very small snippet of code. You usually are not saving this code in a script; it's disposable code, never to be seen or used again. In this case, it can be convenient not to write out the prefixes. If you like to operate this way, type `pylab` at the IPython prompt. This imports NumPy and Matplotlib (a plotting library that we introduce in Chapter 8) as follows:

```
from numpy import *
from matplotlib.pyplot import *
```

While you are learning Python, it's important that you learn which functions belong to which modules. After you become more expert in Python, you can decide if you want to work in an IPython shell in pylab mode.[1]

In this text, we **do not** operate our IPython shell in "pylab" mode. That way, it is always clear to you where the functions we use come from.

Whether you choose to operate your IPython shell in pylab mode or not, the NumPy and Matplotlib libraries (as well as other libraries) are not available in the scripts and programs you write with a text editor and store in a `.py` file unless you explicitly import these modules, which you would do by writing

```
import numpy as np
import matplotlib.pyplot as plt
```

2.7 GETTING HELP: DOCUMENTATION IN IPython

Help is never far away when running the IPython shell. To obtain information on any valid Python or NumPy function, and many Matplotlib functions, simply type `help(`*function*`)`, as illustrated here

[1]Some programmers consider such advice sacrilege. Others find pylab mode to be convenient for their workflow. You can decide if it suits you.

```
In[1]: help(abs)
```

```
Help on built-in function abs in module builtins:

abs(x, /)
Return the absolute value of the argument.
```

Often, the information provided can be pretty extensive, and you might find it helpful to clear the IPython window with the `%clear` command so you can quickly scroll back to see the beginning of the documentation. You may have also noticed that when you type the name of a function plus the opening parenthesis, IPython displays a small window describing the basic operation of that function. To exit `help` mode and return to the IPython prompt, type `q` (for quit).

2.8 PERFORMING SYSTEM TASKS WITH IPython

By typing commands at the prompt, IPython can be used to perform various system tasks, such as running programs, which we have already seen in Section 2.5.2 and Section 2.6.3, and for creating and moving files around on your computer. This is a different kind of computer interface than the icon-based interface (or graphical user interface, GUI) that you normally use to communicate with your computer. While it may seem more cumbersome for some tasks, it can be more powerful for other tasks, particularly those associated with programming.

Before getting started, we remind you that Python, like most modern computer languages, Python is *case sensitive*. That is, Python distinguishes between upper- and lower-case letters. Thus, two words spelled the same but having different letters capitalized are treated as different names in Python. Please keep that in mind in all that follows.

2.8.1 Magic Commands

IPython features a number of commands called "magic" commands that let you perform various useful tasks. There are two types of magic commands: line magic commands that begin with `%`—these are executed on a single line—and cell magic commands that begin with `%%`—these are executed on several lines. Here, we concern ourselves only with line magic commands.

The first thing to know about magic commands is that you can toggle (turn on and off) the need to use the `%` prefix for line magic commands by typing `%automagic`. By default, the `Automagic` switch is usually set to `ON` when you run the IPython shell, so you don't need the `%` prefix. To set `Automagic` to `OFF`, simply type `%automagic` at the IPython prompt. Cell magic commands always need the `%%` prefix.

In what follows, we assume that `Automagic` is `OFF` and thus use the `%` sign for magic commands.

2.8.1.1 The `%run` Magic Command

A very important magic command is `%run` *filename*, where *filename* is the name of a Python program you have created. we already introduced this command in Section 2.5.2. We will use it frequently.

2.8.1.2 Navigation Commands

IPython recognizes several common navigation commands that are used under the Unix/Linux operating systems. In the IPython shell, these few commands also work on PCs, as well as on Macs and Linux machines.

At the IPython prompt, type `%cd` $\sim$ (*i.e.*, "`%cd`" – "space" – "tilde," where tilde is found near the upper left corner of most keyboards).[2] This will set your computer to its home (default) directory (here "`Users/dp`" but yours will be different).

```
In[1]: %cd ~
/Users/dp
```

Next type `%pwd` (**p**rint **w**orking **d**irectory) and press <**return**>. The console should return the path of your computer's current directory. It might look like this on a Mac:

```
In[2]: %pwd
Out[2]: '/Users/dp'
```

or this on a PC:

```
In[3]: %pwd
Out[3]: C:\\Users\\dp
```

Typing `%cd ..` ("`%cd`" – "space" – two periods) moves the IPython shell up one directory in the directory tree, as illustrated by the set of commands below.

```
In[4]: %cd ..
/Users
```

[2]The default mode for most installations is to have `Automagic` set to `ON`. This is likely to be the setting for your installation. If so, you do not need to type the % prefix, so feel free to drop it as you follow along the text.

```
In [5]: %pwd
Out[5]: '/Users'
```

The directory moved up one from `/Users/dp` to `/Users`. Now type `ls` (list) and press <**return**>. The console should list the names of the files and subdirectories in the current directory.

```
In [6]: %ls
Shared/     pine/
```

In this case, there are only two directories (indicated by the slash) and no files (although the names of the files may be different for you). Type `%cd ~` again to return to your home directory and then type `pwd` to verify where you are in your directory tree.

2.8.1.3 Making a Directory

Let's create a directory within your documents directory that you can use to store your Python programs. Let's call it `programs`.[3] First, return to your home directory by typing `%cd ~`. Then type `%ls` to list the files and directories in your home directory.

```
In [7]: %cd ~
/Users/dp

In [8]: %ls
Applications/    Library/        Pictures/
Desktop/         Movies/         Public/
Documents/       Music/
Downloads/       News/
```

To create a directory called `programs`, type `%mkdir programs` (**make** **dir**ectory). Then type `%ls` to confirm that you have created `programs`.

```
In [9]: %mkdir programs

In [10]: %ls
Applications/    Library/        Pictures/
Desktop/         Movies/         Public/
Documents/       Music/          programs/
Downloads/       News/
```

You should see that a new directory named `programs` has been added to the list of directories. Next, type `%cd programs` to navigate to that new directory.

[3] You should have already made a folder to store your programs called **PyScripts** in your **Documents** folder, so this is just for practice to learn how to do it from the Qt Console. If you wish, you can delete it when you are done.

```
 In[11]: %cd programs
/Users/dp/programs
```

In Section 2.5.2, we created a script `my_trip.py` in the directory **~/Documents/PyScripts/**. We can navigate IPython to that directory by typing

```
 In[12]: %cd ~/Documents/PyScripts/
/Users/dp/Documents/PyScripts
```

Let's check that this is indeed the current directory and that `my_trip.py` is in it

```
 In[13]: %pwd
Out[13]: '/Users/dp/Documents/PyScripts'

 In[14]: %ls
my_trip.py
```

Now we can run **MyTrip.py** without specifying the full path

```
 In[15]: %run my_trip.py
```

Sometimes, the IPython shell becomes cluttered. You can clean up the shell by typing `%clear`, which will give you a fresh shell window.

There are a lot of other magic commands, most of which we don't need, and others that we will introduce as needed. If you are curious, you can get a list of them by typing `%lsmagic`.

2.8.2 Tab Completion

IPython also incorporates several shortcuts that make using the shell more efficient. One of the most useful is **tab completion**. Let's assume you have followed along and are in the directory **Documents**. To switch to the directory **PyScripts**, you could type `cd PyScripts`. Instead, type `cd PyS` and press the **<tab>** key. This will complete the command, provided there is no ambiguity about how to finish the command. In the present case, that would mean that there is no other subdirectory beginning with `PyS`. Tab completion works with any command you type into the IPython terminal.

A related shortcut involves the ↑ key. If you type a command, say `cd`, and then press the ↑ key, IPython will complete the `cd` command with the last instance of that command. Thus, when you launch IPython, you can use this shortcut to take you to the directory you used when you last ran IPython.

You can also press the ↑ key, which will recall the most recent command. Repeated application of the ↑ key scrolls through the most recent commands in reverse order. You can use the ↓ key to scroll in the other direction.

2.8.3 Recap of Commands

Let's recap the most useful commands introduced above:

`%pwd` : (**p**rint **w**orking **d**irectory) Prints the path of the current directory.

`%ls` : (**l**i**s**t) Lists the names of the files and directories located in the current directory.

`%mkdir` *filename* : (**m**a**k**e **dir**ectory) Makes a new directory *filename*.

`%cd` *directoryname* : (**c**hange **d**irectory) Changes the current directory to *directoryname*. Note: to work, *directoryname* must be a subdirectory in the current directory. Typing `%cd` changes to your computer's home directory. Typing `%cd ..` moves the console one directory up in the directory tree.

`%precision` *n* : Sets the number of digits displayed ($n = 1$ or 2 or 3, ...) when floating point numbers are displayed. Reset to display all digits by typing `%precision` (with no value of *n*.)

`%clear` : Clears the IPython screen of previous commands.

`%run` *filename* : Runs (executes) a Python script.

Tab completion: Provides convenient shortcuts, with or without the arrow keys, for executing commands in the IPython shell.

2.9 PROGRAMMING ERRORS

Now that you have a little experience with Python and computer programming, it's time for an important reminder: ***Programming is a detail-oriented activity***. To be good at computer programming and to avoid frustration when programming, you must pay attention to details. A misplaced or forgotten comma or colon can keep your code from working. Worse still, little errors can make your code give erroneous answers, where your code appears to work, but in fact, does not do what you intended it to do! So pay attention to the details!

2.9.1 Error Checking

This raises a second point: sometimes your code will run but give the wrong answer because of a programming error or because of a more subtle error in your algorithm, even though there may be nothing wrong with your Python syntax. The program runs; it just gives the wrong answer. For example, maybe you typed `sin` where you meant to use `cos`. For this reason, it is essential to test your code to ensure it behaves properly. Test it to ensure it gives the correct answers for cases where you already know the right answer or have some independent means of checking it. Test it in limiting cases, that is, at the extremes of the sets of parameters you will employ. ***Always test your code; this is a cardinal rule of programming.***

2.10 EXERCISES

1. A ball is thrown vertically up in the air from a height h_0 above the ground at an initial velocity v_0. Its subsequent height h and velocity v are given by the equations

$$h = h_0 + v_0 t - \frac{1}{2} g t^2$$
$$v = v_0 - gt$$

   where $g = 9.8$ is the acceleration due to gravity in m/s^2. Write a script that finds the height h and velocity v at a time t after the ball is thrown. Start the script by setting $h_0 = 1.6$ (meters) and $v_0 = 14.2$ (m/s). Then have your script print out the values of height and velocity after 0.5 seconds and after 2.0 seconds.

2. Write a script that defines the variables $V_0 = 10, a = 2.5$, and $z = 4\frac{1}{3}$, and then evaluates the expression

$$V = V_0 \left(1 - \frac{z}{\sqrt{a^2 + z^2}}\right).$$

   Then find V for $z = 8\frac{2}{3}$ and print it out (see *Note about printing* on page 18). Then, find V for $z = 13$ by changing the value of z in your script. Use either the `numpy` or `math` function `sqrt` to calculate the square root in this expression.

3. Write a single Python script that calculates the following expressions:

   (a) $$a = \frac{2 + e^{2.8}}{\sqrt{13} - 2}$$

   (b) $$b = \frac{1 - (1 + \ln 2)^{-3.5}}{1 + \sqrt{5}}$$

   (c) $$c = \sin\left(\frac{2 - \sqrt{2}}{2 + \sqrt{2}}\right)$$

After running your script in the IPython shell, typing `a`, `b`, or `c` at the IPython prompt should yield the value of the expressions in (a), (b), or (c), respectively. Use the either the `numpy` or `math` packages as needed.

4. A quadratic equation with the general form

$$ax^2 + bx + c = 0$$

has two solutions given by the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

   (a) Given a, b, and c as inputs, write a script that gives the numerical values of the two solutions. Use NumPy to calculate the square root in this exercise. Write the constants a, b, and c as floats, and show that your script gives the correct solutions for a few test cases when the solutions are real numbers, that is, when the discriminant $b^2 - 4ac \geq 0$. Use the `print` function in your script, discussed at the end of Section 2.5.2, to print out your two solutions.

   (b) Written this way, however, your script gives an error message when the solutions are complex. For example, see what happens when $a = 1$, $b = 2$, and $c = 3$. You can fix this using statements in your script like `a = a+0j` after setting a to some float value. Thus, you can make the script work for any set of real inputs for a, b, and c. Again, use the `print` function to print out your two solutions.

5. Write a program to calculate the perimeter p of an n-gon inscribed inside a sphere of diameter 1. Find p for n = 3, 4, 5, 100, 10,000, and 1,000,000. Your answers should be

n	p	n	p
3	2.59807621135	100	3.14107590781
4	2.82842712475	10,000	3.14159260191
5	2.93892626146	1,000,000	3.14159265358

CHAPTER 3

Integrated Development Environments

In this chapter, you learn about Integrated Development Environments, also known as IDEs. Two IDEs designed for scientific and engineering work using Python are introduced: Spyder and JupyterLab. These IDEs include many useful features, including an IPython shell, a Python program editor that provides syntax checking and formatting, Jupyter notebooks, and debugging tools.

3.1 PROGRAMMING AND INTERACTING WITH PYTHON

In the previous chapter, you learned how to launch an IPython shell. You also learned how to execute Python commands and how to write and execute short Python scripts. While this method of interacting with Python is adequate for simple tasks, it becomes tedious and error-prone for more sophisticated programming tasks. *Integrated Development Environments*, commonly known as *IDEs*, address these limitations by providing tools that make computer programming more efficient and less error-prone.

In this chapter, we introduce two IDEs that have been specifically designed for scientific programming: *Spyder* and *JupyterLab*. Both use the IPython shell introduced in Chapter 2. Both support Jupyter notebooks, which we introduce in Section 3.5. Both are free and included in the Anaconda Python distribution you were advised to download in Section 1.2. Both are highly configurable, powerful, and relatively simple to use.

Spyder is the more mature of the two IDEs. Spyder comes equipped with all the tools you'll need, at least as a beginning to intermediate

DOI: 10.1201/9781032673950-3

programmer. Configuring the IDE to suit your preferences is easier with Spyder. With JupyterLab, you need to download several extensions to get the functionality you need (we show you how to do this for the ones we think are most helpful). The upside of this approach is that you include in JupyterLab only the features you want. On the other hand, Spyder is a relatively lightweight IDE, and the parts you don't use don't get in the way. Nevertheless, JupyterLab is very popular and has a large and enthusiastic user base. Which IDE you prefer is largely a matter of taste.

3.2 PROGRAMMING STYLE AND CODING ERRORS: PEP 8 AND LINTERS

The two IDEs we introduce in this chapter include tools to help you write clear, consistent code. Guido van Rossum, the creator of Python, noted that code is read more often than it is written. To improve the readability of code and make it consistent among the multitude of Python programmers, he and several others developed a programming style guide called PEP 8 (PEP stands for *Python Enhancement Proposal*). These guidelines are not meant to be applied rigidly, but following them generally improves readability, especially for beginning programmers. Following them can also make it easier for you to spot coding errors.

To help you format your code to be consistent with PEP 8 style guidelines, people have written programs called *linters*. A linter passes over your Python code and flags style violations. More importantly, linters also flag a large number of coding syntax errors. It's like a lint brush that you pass over your clothing to eliminate the messy little fibers and other detritus that accumulate; a linting program flags messy code in a program.

You can use a linting program in two ways. First, you can write your code and then run a linting program to analyze it. Two widely used Python linters are pylint and pyflakes. Running a linter on your Python code will produce a list of style violations and coding errors, which you can then go back and fix. A good IDE will include a linter to analyze the code you write.

The second way to use a linting program is more automatic and has two parts. First, you can use an autoformatter, which should be included in any good IDE. It detects *style violations* and automatically corrects them, saving you time and work. Three of the most widely used Python autoformatters are *Autopep8*, *Yapf*, and *Black*. Each makes slightly different default formatting choices, with some flexibility for you to make adjustments.

Second, a good IDE will run a linter in the background and flag the syntax errors it finds in the code editor more-or-less where the error occurs.

This alerts you to coding errors as they occur and allows you to correct them on the spot. Errors like unmatched parentheses or brackets, which are easy for humans to miss, are reliably flagged. They can even spot inconsistently spelled variable names. They should catch all *syntax* errors. Usually, the error is flagged right on the line where it occurs. Sometimes, however, the error has occurred a line or two before the flagged line of code. So, look around.

On the other hand, linters cannot catch *logical* errors, typos, or errors that are logically wrong but syntactically correct. We will return to the question of finding and expunging logical errors later. In the meantime, use a linter; just avoiding syntax errors will save you time—hours!

Fortunately, both IDEs introduced in this chapter feature a linter and autoformatters, including those mentioned above: Autopep8, Yapf, and Black. We provide details in the sections that follow.

3.3 THE SPYDER IDE

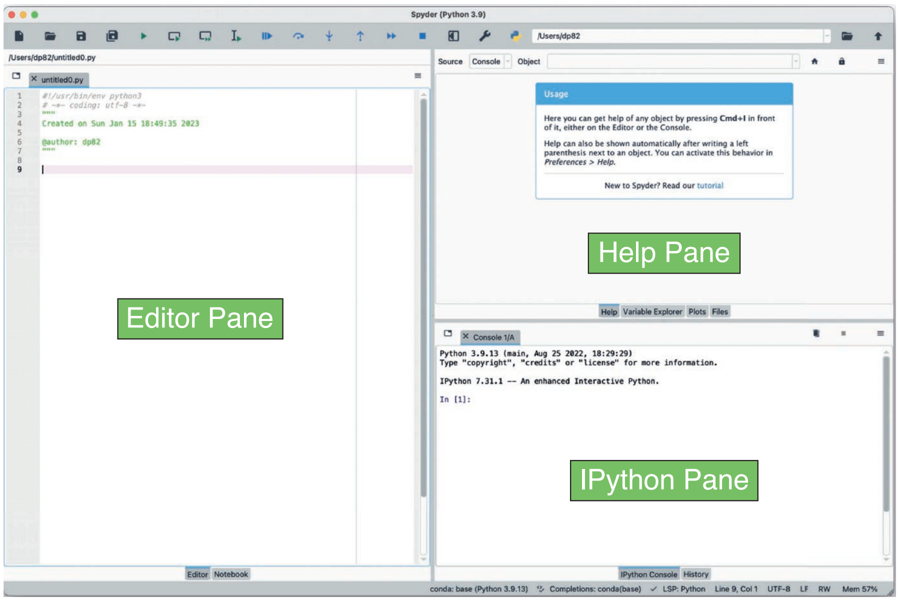
Spyder is an IDE designed for scientific programming. It runs as a standalone application on macOS, Windows, and Linux. It is open-source (free) software first developed in 2009. Since then, a team of volunteer developers has maintained and enhanced it.

To get started, launch the Spyder IDE application on your computer. One way is to launch the Terminal application (PowerShell on Windows), type `spyder`, and then press **<return>**. You can also launch the Anaconda-Navigator app, which presents a menu from which you can launch Spyder.

The default Spyder IDE window has three panes, which are shown in Figure 3.1: The IPython pane, the Editor pane, and the Help pane.

- The **IPython pane** runs the Qt Console IPython shell that you learned about in Chapter 2. You can use it to perform very simple calculations, run Python computer programs, test snippets of Python code, navigate through your computer file directories, and perform system tasks like creating, moving, and deleting files and directories.

- The **Editor Pane** is where you write and edit Python programs (or scripts). It is like the text editor applications introduced in Chapter 2. Like them, it features syntax highlighting. In addition, it can run a linter or configure an autoformatter to check the formatting of your code and improve its readability. More importantly, it can check your code for syntax errors. As noted above, this powerful feature will save you hours.



```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Jan 15 18:49:35 2023

@author: dp82
"""

```

Usage

Here you can get help of any object by pressing **Cmd+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help.*

New to Spyder? Read our tutorial

```
Python 3.9.13 (main, Aug 25 2022, 18:29:29)
Type "copyright", "credits" or "license" for more information.

IPython 7.31.1 -- An enhanced Interactive Python.

In [1]:
```

Figure 3.1 Spyder IDE window.

- The **Help Pane** in Spyder gives help on Python commands. It's formatted very nicely, similar to the formatting used on the web pages for Python and its various packages.

  The individual panes in the Spyder window are reconfigurable and detachable, but we will leave them pretty much as they are. However, you may want to adjust the window's overall size to suit your computer screen. Below, we suggest a few other customizations you are highly encouraged to make.

3.3.1 Autoformatting and Linting in Spyder

By default, basic linting is turned on in Spyder. However, we highly recommend turning on some additional linting features, including autoformatting. To do so, go to the **Spyder Preferences** menu (or select the wrench tool 🔧 at the top of the Spyder window), select the **Completion and linting** submenu, and then select the **Linting** tab. Check the **Enable basic linting** and **Underline errors and warning** boxes if they are not already checked, and then press the **Apply** button (do not forget to press the **Apply** button). Next, go to the **Code style and formatting** tab, check the **Enable code style linting** box, and press the **Apply** button. You should also see a spinbox that you can use to select either the

Autopep8, Yapf, or Black linters. Start with Autopep8, and after you gain some experience, try the others to see which you like best.

With these selections made, the Python linter will flag formatting violations and syntax errors. A red circle appears to the left of the line where the linter thinks the error occurs. Sometimes, the error actually occurs in a preceding line, so look around. A yellow triangle appears to the left of the line where the linter thinks that the coding style doesn't conform to the PEP 8 standard; it's not an error, just a coding style violation, which you can heed or ignore. Passing your mouse pointer over the red or yellow icon brings up a *Code Analysis* box with a brief message describing the error or style violation. You can get a complete list of all the errors, warnings, and style violations by bringing up the **Code Analysis** pane (see below).

You can fix formatting violations manually or do it globally for the entire file or just for a highlighted selection by selecting the **Format file or selection with Autopep8** in the **Source** menu.

3.3.2 Running Python Code in Spyder

There are different ways you can run Python code using Spyder. To illustrate, use the **File:open** menu in Spyder to open up the **my_trip.py** script you wrote in Chapter 2 (see p. 16). This will bring up the **my_trip.py** code in the **Editor** pane of Spyder. To run this code, press the right-pointing green arrow icon near the top-left of the Spyder window. This runs the Python code in the **my_trip.py** file. You should see a `runfile` command in the IPython pane, together with the complete path to that file. Typing the output variables, time, gallons, and cost, will display the values of these variables calculated in the script, just as it did in the IPython shell in Chapter 2. It should look something like this:

```
 In[1]: runfile('/Users/dp/Documents/PyScripts/my_trip.py',
wdir='/Users/dp/Documents/PyScripts')

 In[2]: time, gallons, cost
Out[2]: (6.666666667, 13.333333333, 48.666666667)
```

You can also run the file by selecting **Run** from the **Run** menu at the top of the Spyder window. Of course, you can run the file by navigating to the file containing the **my_trip.py** file and issue a run command from the IPython pane, as shown here:

```
 In[3]: %cd ~/Documents/PyScripts

 In[4]: %run  my_trip.py
```

```
In[5]: time, gallons, cost
Out[5]: (6.666666666666667, 13.333333333333334, 38.0)
```

3.3.2.1 Other Spyder Panes

The Spyder window has many more features, including a number of panes in addition to the three discussed so far. All are available by selecting **Panes** under the **View** menu. If a particular pane has been selected in the **View**→**Panes** menu, it is available by pressing the corresponding tab at the bottom of one of the panes. Here is a brief list of the different panes and what they do:

Variable Explorer displays a table showing the names, types, sizes, and values of all the currently defined variables. This information can be helpful for debugging code.

Plots displays any currently active plots. When plotting is introduced in Chapter 8, we will say more about this.

Files displays your computer's directory tree and allows you to quickly navigate between directories to access, edit, and run different files. Clicking on a file name in the directory tree brings it up in the Editor pane.

Code Analysis displays the results of running Spyder's linter on the code shown in the Editor pane or selected in the spinbox at the top of the window. You run the linter by pressing the right-pointing green arrow icon ▶ near the top right of the Code Analysis pane.

History shows you the commands you have entered in the IPython console.

Profiler shows how fast the various parts of a program run, which you can use to optimize your code and reduce the overall time it takes to run.

Spyder has several additional panes that you may find useful as you become a more experienced Python programmer. As you proceed with learning Python using Spyder, we encourage you to explore its many features. A series of tutorial videos are available on YouTube and can be accessed through Spyder's **Help: Tutorial videos** menu. These videos will teach you how to get started with Spyder's debugger, its profiler, and many other powerful and helpful features. You can also explore these features through the **Help** menu and the online documentation.

There is also a plugin that allows Spyder to run Jupyter Notebooks, which are described in Section 3.5. However, Jupyter Notebooks are easier to use within the JupyterLab IDE, which we describe next.

3.4 THE JUPYTERLAB IDE

JupyterLab is an IDE designed for scientific and technical computing. Instead of working as a standalone application, JupyterLab runs in a web browser. However, your computer does not need to be connected to the internet for JupyterLab to work; JupyterLab runs locally on your personal computer without an internet link. JupyterLab can be run using the Firefox, Chrome, or Safari browsers, so make sure you have one of these installed as the default browser on your computer if you are going to use JupyterLab.

JupyterLab is open-source software. First announced in 2018, JupyterLab grew from Jupyter Notebook, a project that started around 2014, which, in turn, developed from the IPython command shell, which dates back to about 2001. Like Spyder, JupyterLab is maintained by a community of volunteers.

To get started, launch JupyterLab on your computer. One way to do this is to launch the Terminal application (PowerShell on Windows), type `jupyter lab` (two words, lowercase), and then press <**return**>. Or, if you prefer, you can launch the **Anaconda-Navigator** app, which presents a menu from which you can launch JupyterLab. Take care that you launch JupyterLab and not Jupyter Notebook; they are different.

Upon launching JupyterLab, a window like the one shown in Figure 3.2 should appear in your default web browser. It has a **Menu Bar** at the top window (separate from the menu bar for the browser). It also has a **Left Sidebar**, a **Right Sidebar**, and a **Main Work Area**, which in Figure 3.2 has a **Launcher** tab open.

The left and right sidebars are divided into different tabs that can be selected by clicking on a tab's icon, which opens up a side pane for that tab. In Figure 3.2, the **File Browser** tab 📁 has been selected in the **Left Sidebar**, which has opened up the side pane showing the contents of the open directory. You can navigate up and down the directory tree by clicking on the appropriate icons. Section 2.5.2. The menus opened by the sidebar tabs can be set to collapse or remain open using the **View** tab in the **Menu Bar**.

The JupyterLab **Main Work Area** pane is highly configurable, providing more freedom than you may want as a beginner. We will show you how to configure the JupyterLab like the default Spyder window. Over time, as you gain experience, you will want to configure the panes to suit your workflow.

With the **File Browser** pane open in the **Left Sidebar**, navigate to the **PyScripts** directory on your computer and click on the **my_trip.py** item in the File Browser pane. This will create a new tab labeled **my_trip.py** in the **Main Work Area** pane. Right-click anywhere in the **my_trip.py** pane and select **Create Console for Editor**. This should bring up a Qt Console of the IPython shell in the

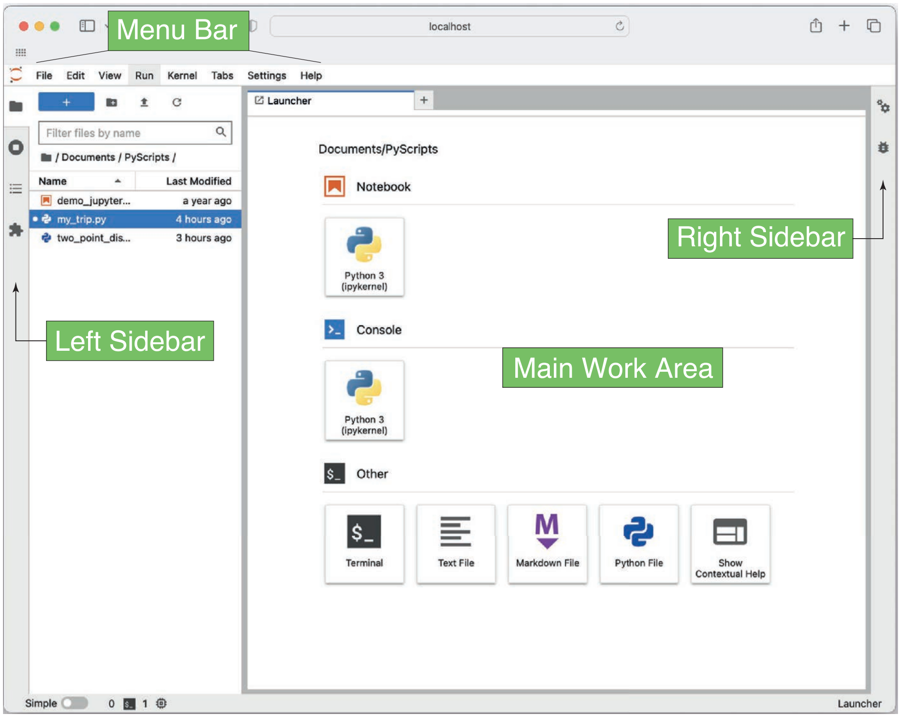


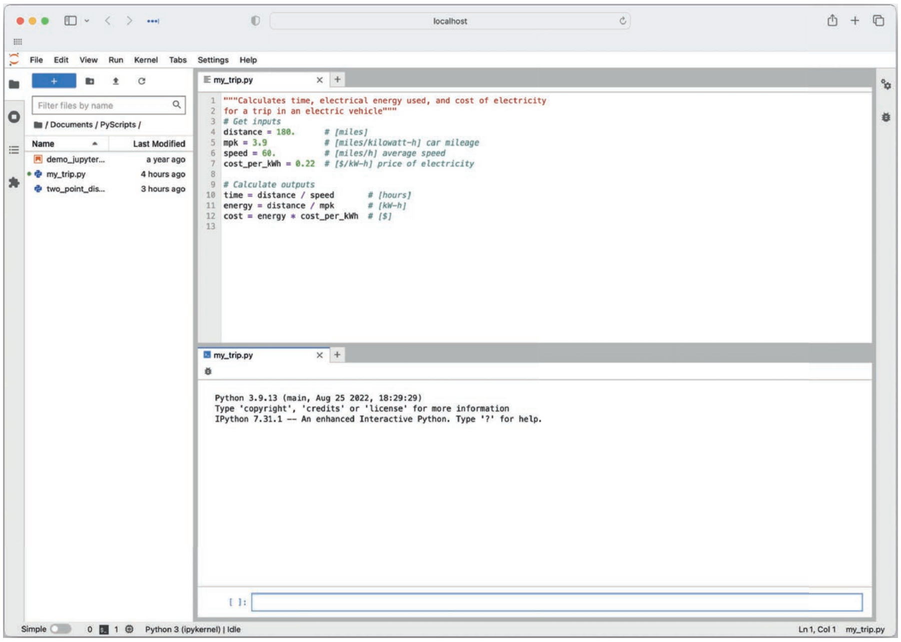
Figure 3.2 JupyterLab IDE window organization.

Main Work Area pane. Your JupyterLab window should look something like the one displayed in Figure 3.3.

Type `%pwd` at the IPython prompt in the IPython console, and then press **<shift-return>**. In JupyterLab, you need to press **<shift-return>** to execute a command written at the IPython prompt. Simply pressing **<return>** will open another line where you can write another command. You can change this to simply be **<return>** by selecting **Console Run Keystroke** from the **Settings** menu.

Next, run the `run my_trip.py` script by writing run `run my_trip.py` at the IPython prompt and press **<shift-return>**. Finally, type `time, energy, cost` at the IPython prompt to print out the output variables, `time`, `energy`, and `cost`, and verify that you get the expected answers (3.0, 46.15..., 10.15...).

This is just one way of setting up the panes in JupyterLab. Select the **Launcher** tab next to the **my_trip.py** tab to see another way. If there is no **Launcher** tab, press the + icon at the upper left of the JupyterLab window, which will create a new **Launcher** tab. Scroll up the window and press the **Python File** button. This opens up a new blank Python file ready for you to



```
"""Calculates time, electrical energy used, and cost of electricity
for a trip in an electric vehicle"""
# Get inputs
distance = 180.      # [miles]
mpk = 3.9            # [miles/kilowatt-h] car mileage
speed = 60.          # [miles/h] average speed
cost_per_kWh = 0.22  # [$/kW-h] price of electricity

# Calculate outputs
time = distance / speed        # [hours]
energy = distance / mpk        # [kW-h]
cost = energy * cost_per_kWh   # [$]
```

```
Python 3.9.13 (main, Aug 25 2022, 18:29:29)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.31.1 -- An enhanced Interactive Python. Type '?' for help.

[ ]:
```

Figure 3.3 JupyterLab IDE window.

start coding. It's provisionally named **untitled.py**, but you can change that by selecting **Rename Python File...** from the **File** menu.

To shut down JupyterLab, go to the **file** menu and select **Shut Down**. Go ahead and do that now.

3.4.1 Jupyter Extensions

People have written several extensions for JupyterLab that add features that you might find helpful. Of the available extensions, we recommend installing three: autoformatting, linting, and variable inspection. Instructions on how to install and activate each one of the extensions are provided in the sections that follow.

3.4.1.1 Autoformatting in JupyterLab

Autoformatting is not included in the default installation of JupyterLab. However, an effective autoformatter is available through the conda package manager you used to install Python on your computer. To download the JupyterLab formatter, make sure JupyterLab is shut down and then launch the **Terminal** application (**Anaconda Prompt** on Windows). At the terminal prompt, type

```
conda install -c conda-forge jupyterlab_code_formatter
```

This installs the autoformatter. After completing the installation, launch JupyterLab. In the JupyterLab **Edit** menu, you should see four new entries: **Apply X Formatter**, where **X** is **Black**, **Autopep8**, **YAPF**, and **Isort**. If you do not see these menu items, then the autoformatter may not be enabled. To see which extensions have been enabled, type

```
jupyter lab extension list
```

If `jupyterlab-code-formatter` is not included in the list of enabled extensions, type

```
jupyter lab extension enable codeformatter
```

Test the autoformatter to see how it works. As an example, open in JupyterLab the **my_trip.py** program you created in Section 2.5.2. Remove the spaces around the = sign in the first code line; that is change "`distance = 400.`" to "`distance=400.`". With the **my_trip.py** tab of the JupyterLab editor open, select **Apply Autopep8 Formatter**. You should see the spaces around the equal sign reappear, when the autoformatter applies the PEP 8 standard.

3.4.1.2 Linting in JupyterLab

A linter is not included in the default installation of JupyterLab. However, you can download one using the conda package manager. To do so, make sure JupyterLab is shut down and then launch the **Terminal** application (**PowerShell** on Windows). At the Terminal prompt, type

```
conda install -c conda-forge jupyterlab-lsp
```

This installs the linter. After completing the installation, launch JupyterLab. The linter identifies two types of errors: (1) *critical errors*, which are underlined in red and (2) *warnings*, which are underlined in orange. You can hover the cursor over the underlined code to see detailed description of the error. Right-clicking on an error or warning brings up a panel, and selecting **Show diagnostics panel** brings up another tab with a list of all the warnings and errors found in the file. You can suppress a particular kind of error message by right-clicking on the error message in the diagnostics panel and selecting **Ignore diagnostics like this**.

3.4.1.3 Variable Inspection in JupyterLab

The default installation of JupyterLab does not include a variable inspector. A variable inspector displays a table showing the names, types, sizes, and values of all the currently defined variables in any program you have just run. This information can be helpful for debugging code. A variable inspector extension is available through the conda package manager you used to install Python on your computer. To download the JupyterLab variable inspector, make sure JupyterLab is shut down and then launch the **Terminal** application (**PowerShell** on Windows). At the terminal prompt, type

```
conda install -c conda-forge jupyterlab-variableinspector
```

This installs the variable inspector.

3.5 JUPYTER NOTEBOOKS

A Jupyter Notebook is an environment for interactive computing. It can run under both Spyder and JupyterLab. It can also run as a stand-alone program in a web environment, similar to how JupyterLab runs, but we don't recommend it, as the stand-alone Jupyter Notebook is being phased out (deprecated) in favor of JupyterLab.

To run it under Spyder, execute the following installation command from the terminal (Anaconda Prompt in Windows):

```
conda install -c conda-forge spyder-notebook
```

Once you have installed the Spyder Notebook plugin, you can open a Jupyter Notebook by pressing the Notebook tab at the bottom of the Spyder Editor Pane (see Figure 3.1). No extra installation step is required to run it under JupyterLab.

You can work in a Jupyter Notebook interactively, just as you would using the IPython shell. In addition, you can store and run programs in a Jupyter Notebook just like you would within the *Spyder* IDE. Thus, it would seem that a Jupyter Notebook and the *Spyder* IDE do essentially the same thing. Up to a point, that is true. Spyder is generally more useful for developing, storing, reusing, and running code. A Jupyter Notebook on the other hand, is excellent for logging your work in Python. For example, Jupyter Notebooks are very useful for logging, reading, and analyzing data in a laboratory setting. They are also useful for logging and turning in homework assignments. You may find them useful in other contexts for documenting and demonstrating software.

3.6 LAUNCHING A JUPYTER NOTEBOOK

To launch a Jupyter Notebook in Spyder select **Notebook** from the **View:Panes** menu. This will launch a Jupyter Notebook in the same Spyder pane as the

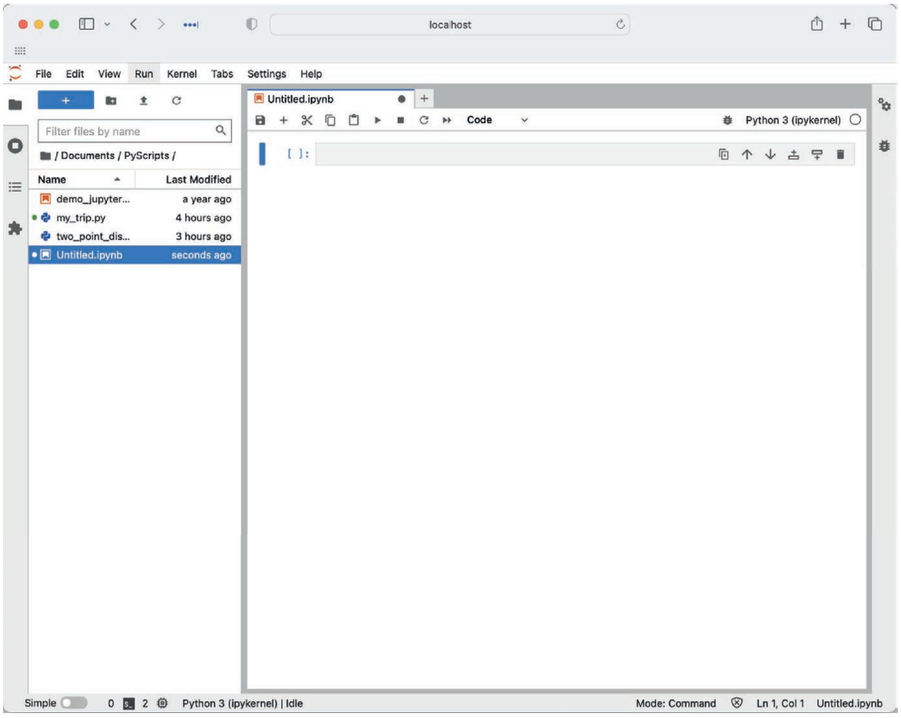

localhost
File Edit View Run Kernel Tabs Settings Help
Filter files by name
/ Documents / PyScripts /
Name
Last Modified
demo_jupyter... a year ago
my_trip.py 4 hours ago
two_point_dis... 3 hours ago
Untitled.ipynb seconds ago
Untitled.ipynb
Code
Python 3 (ipykernel)
[ ]:
Simple
0
2
Python 3 (ipykernel) | Idle
Mode: Command
Ln 1, Col 1
Untitled.ipynb

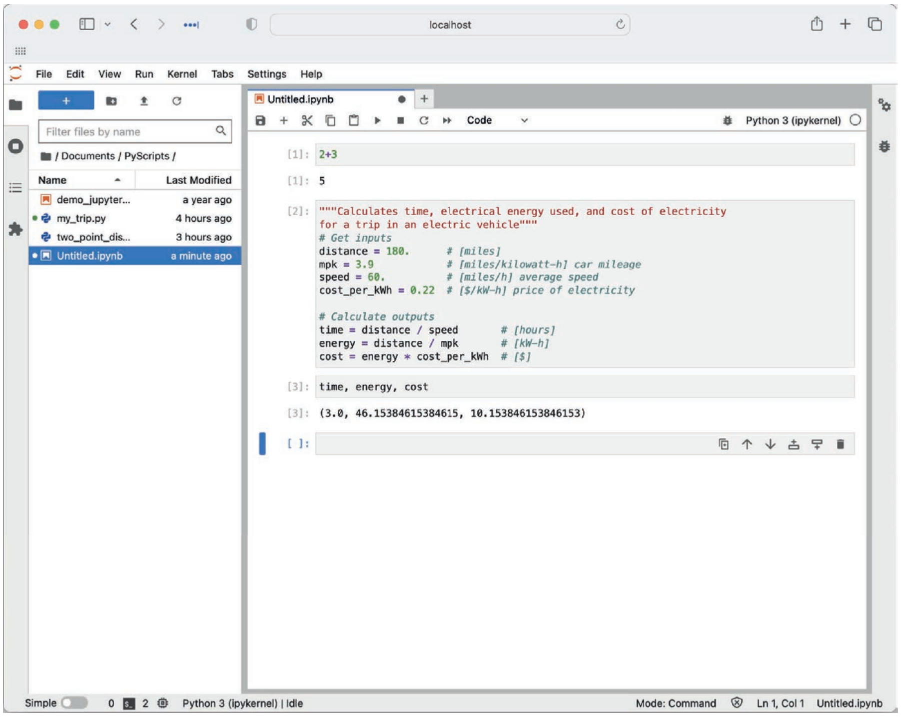
Figure 3.4 Untitled Jupyter notebook with an open cell.

Editor. A tab at the bottom of the pane allows you to toggle (switch back and forth) between the **Editor** and **Notebook**.

To launch a Jupyter Notebook in JupyterLab, open a launcher pane from the **File** menu or by pressing the + icon. It should look like the web page shown in Figure 3.2. In the **Launcher** pane, press the **Python 3 (ipykernel**) button beneath the red **Notebook** icon. This will bring up a Jupyter Notebook pane with the provisional name of **Untitled.ipynb** like the one shown in Figure 3.4. To rename the file, click on the **File** menu (in the Jupyter window, not for the web browser) and select Rename Notebook....

The Jupyter Notebook pane in Spyder should look like the one in JupyterLab shown in Figure 3.4. You may need to close other tabs to get a full page view like the one shown in Figure 3.4.

When you open a new Jupyter Notebook, an IPython interactive cell appears with the prompt `In[ ]:` to the left. You can type code into this cell just as you would in the IPython shell of the *Spyder* IDE. For example, typing `2+3` into the cell and pressing <**shift-return**> executes the cell and yields the expected result. Try it out.


localhost
File Edit View Run Kernel Tabs Settings Help
Filter files by name
/ Documents / PyScripts /
Name Last Modified
demo_jupyter... a year ago
my_trip.py 4 hours ago
two_point_dis... 3 hours ago
Untitled.ipynb a minute ago
Untitled.ipynb
Code
Python 3 (ipykernel)
Simple 0 2 Python 3 (ipykernel) | Idle
Mode: Command Ln 1, Col 1 Untitled.ipynb

```
[1]: 2+3

[1]: 5

[2]: """Calculates time, electrical energy used, and cost of electricity
     for a trip in an electric vehicle"""
     # Get inputs
     distance = 180.      # [miles]
     mpk = 3.9            # [miles/kilowatt-h] car mileage
     speed = 60.          # [miles/h] average speed
     cost_per_kWh = 0.22  # [$/kW-h] price of electricity

     # Calculate outputs
     time = distance / speed       # [hours]
     energy = distance / mpk       # [kW-h]
     cost = energy * cost_per_kWh  # [$]

[3]: time, energy, cost

[3]: (3.0, 46.15384615384615, 10.153846153846153)

[ ]:
```

Figure 3.5 Running a program in a Jupyter Notebook.

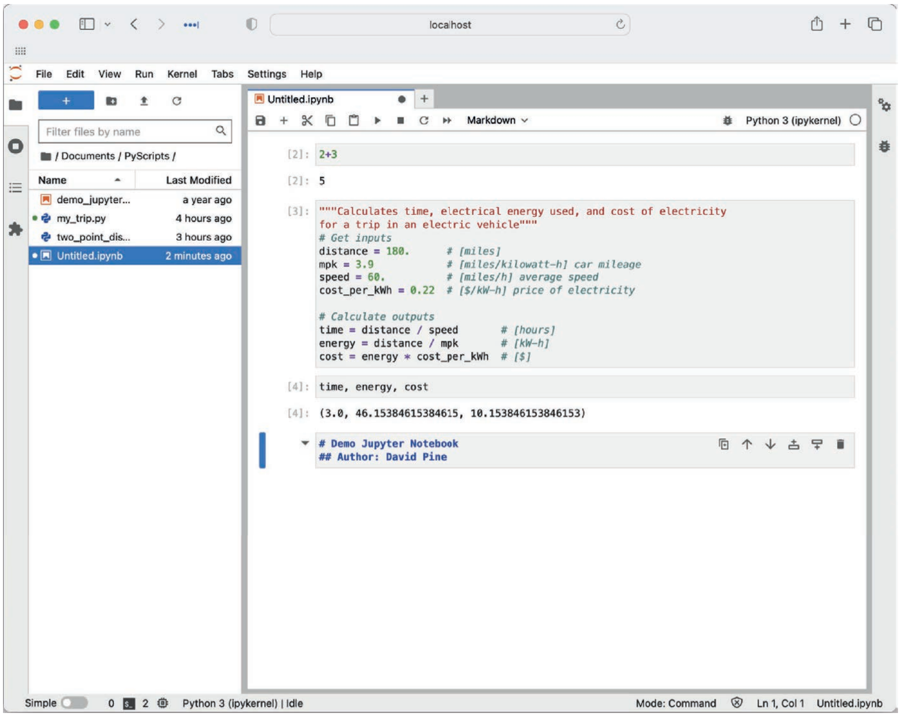
If you want to delete a cell, you can do so by clicking on the cell and then selecting **Delete Cells** from the **Edit** menu. Go ahead and delete the cell you just entered. You can also restart a notebook by selecting **Restart & Clear All Outputs...** from the **Kernel** menu. Go ahead and do this too.

3.7 RUNNING PROGRAMS IN A JUPYTER NOTEBOOK

You can run programs in a Jupyter Notebook. As an example, we run the program named **my_trip.py** introduced in Section 2.5.2. The program is input into a single notebook cell, as shown in Figure 3.5, and then executed by pressing `Shift-Enter`. Next, we type `time, gallons, distance` in the next cell, press `Shift-Enter`, and verify that the program has indeed run and produces the expected output.

3.8 ANNOTATING A JUPYTER NOTEBOOK

In addition to logging the inputs and outputs of computations, Jupyter Notebooks allow the user to embed headings, explanatory notes, mathematics, and


localhost
File Edit View Run Kernel Tabs Settings Help
Filter files by name
/ Documents / PyScripts /
Name
Last Modified
demo_jupyter... a year ago
my_trip.py 4 hours ago
two_point_dis... 3 hours ago
Untitled.ipynb 2 minutes ago
Untitled.ipynb
Markdown
Python 3 (ipykernel)
Simple 0 2 Python 3 (ipykernel) | Idle
Mode: Command Ln 1, Col 1 Untitled.ipynb

```
[2]: 2+3

[2]: 5

[3]: """Calculates time, electrical energy used, and cost of electricity
     for a trip in an electric vehicle"""
     # Get inputs
     distance = 180.      # [miles]
     mpk = 3.9            # [miles/kilowatt-h] car mileage
     speed = 60.          # [miles/h] average speed
     cost_per_kWh = 0.22  # [$/kW-h] price of electricity

     # Calculate outputs
     time = distance / speed       # [hours]
     energy = distance / mpk       # [kW-h]
     cost = energy * cost_per_kWh  # [$]

[4]: time, energy, cost

[4]: (3.0, 46.15384615384615, 10.153846153846153)

     # Demo Jupyter Notebook
     ## Author: David Pine
```

Figure 3.6 Annotating a Jupyter Notebook with title and author.

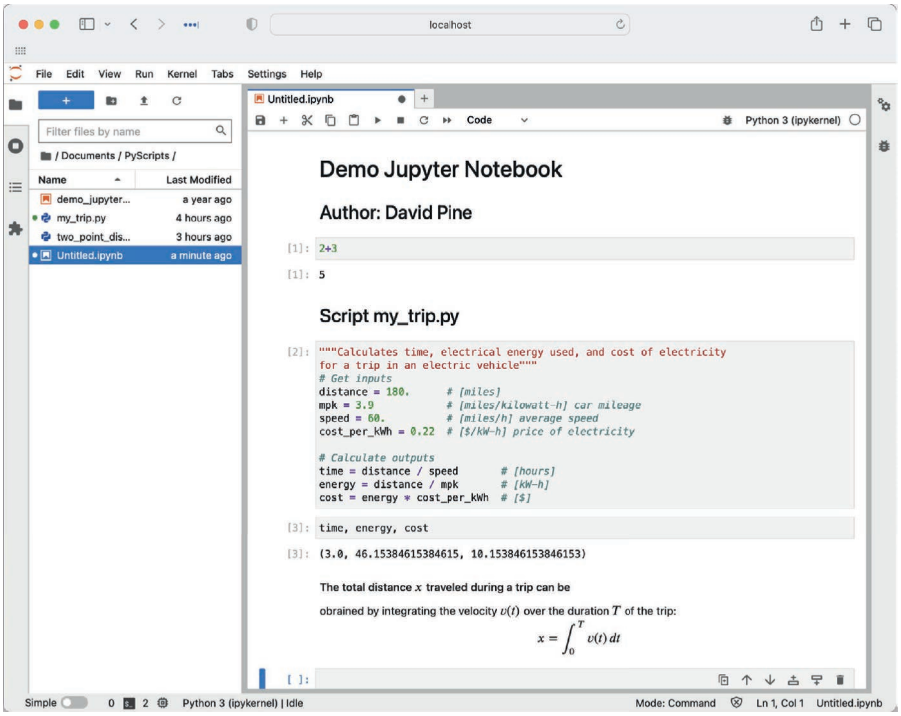
images. A Jupyter Notebook is easier to understand if it includes annotations that explain what is going on in the notebook.

3.8.1 Adding Headings and Text

Suppose, for example, that we want to have a title at the top of the Jupyter Notebook we have been working with, and we want to include the name of the author of the session. To do this, enter the following text in the next open cell:

```
# Demo Jupyter Notebook
## Author: David Pine
```

Before pressing `Shift-Enter`, go up to the top of the Main Work Area window and change the spinbox from **Code** to **Markdown**, as shown in Figure 3.6. Upon selecting **Markdown**, the `In [ ]:` prompt disappears, indicating that this box is no longer meant for inputting and executing Python code. Instead, the text is interpreted as *Markdown*, a markup language that allows various kinds



Demo Jupyter Notebook

Author: David Pine

```
[1]: 2+3

[1]: 5
```

Script my_trip.py

```
[2]: """Calculates time, electrical energy used, and cost of electricity
     for a trip in an electric vehicle"""
     # Get inputs
     distance = 180.      # [miles]
     mpk = 3.9            # [miles/kilowatt-h] car mileage
     speed = 60.          # [miles/h] average speed
     cost_per_kWh = 0.22  # [$/kW-h] price of electricity

     # Calculate outputs
     time = distance / speed       # [hours]
     energy = distance / mpk       # [kW-h]
     cost = energy * cost_per_kWh  # [$]

[3]: time, energy, cost

[3]: (3.0, 46.15384615384615, 10.153846153846153)
```

The total distance x traveled during a trip can be

obrained by integrating the velocity $v(t)$ over the duration T of the trip:

$$x = \int_0^T v(t)\,dt$$

```
[ ]:
```

Figure 3.7 An annotated Jupyter Notebook with multiple headings and formatted math.

of formatting. In this case, the single hash symbol `#` indicates that what follows is a main heading. The double hash symbols `##` on the next line indicate that what follows is a subheading. Now press `Shift-Enter` to get the formatted Markdown headings.

The only problem is that you probably want these headings to be at the top of the notebook before the first code cell. To move the textbox with the headings, use your mouse to place the cursor just to the left of the text. Then, left-click your mouse and move the headings to the top of the page. Alternatively, you can click on the up arrow in the text box to move it up one cell at a time. Figure 3.7 shows the heading placed at the top of the JupyterLab along with additional annotations discussed below.

Following the same procedure as outlined above, we have added two other subheadings and moved them to the desired locations in the Jupyter Notebook. The text for the two entries are:

```
### Script my_trip.py
```

and

```
#### Output
```

The additional hash symbols # designate other subheadings with progressively smaller typefaces. Up to six are allowed.

The final text box contains text as well as a mathematical equation. The Markdown code that produced the final text box is:

```
##### The total distance $x$ traveled during a trip can be
obtained by integrating the velocity $v(t)$ over the duration
$T$ of the trip:
\begin{equation}
x = \int_0^T v(t)\,dt
\end{equation}
```

Notice that the text is now preceded by five hash symbols, producing normal-sized text. The equation is written using LaTeX, a mathematical typesetting language. The symbols $x$, $v(t)$, and $T$ are also written using LaTeX, which Markdown recognizes. More details about LaTeX are provided in Section 8.7 in the context of using Python for plotting. If you do not already know LaTeX, you can get a brief introduction at this site: http://en.wikibooks.org/wiki/LaTeX/Mathematics.

The Markdown markup language used in JupyterLab has many more features for formatting text, which you can read about online.

3.8.2 Saving a Jupyter Notebook

By default, JupyterLab saves your Jupyter Notebook every two minutes. If you haven't already changed the name of your Jupyter Notebook from **Untitled.ipynb**, you can do so by clicking on the JupyterLab **File** menu and selecting **Rename Notebook...**. We renamed my notebook **demo_jupyter_notebook.ipynb**, which you can see (partially) in the **File Browser** pane in Figure 3.7.

3.8.3 Editing and Rerunning a Notebook

In working with a Jupyter Notebook, you may want to move some cells around, delete some cells, or simply change some cells. All of these tasks are possible. As in a standard document editor, you can cut and paste cells using the **Edit** menu. You can also edit and re-execute cells by pressing **<shift-return>**.

Sometimes, you may want to re-execute the entire notebook afresh. There are two ways to do this. The first way is to go to the **Kernel** menu and select **Restart and Run All Cells...**. A warning message will appear asking you if you really want to restart. Answering in the affirmative will cause the IPython shell to forget all the variables and rerun all the cells, calculating anew all the variables defined in the cells. The second way to re-execute the entire notebook is to go to the **Run** menu and select **Run All Cells**. This does not cause the IPython shell to forget all variables before rerunning the notebook and, in general, is not the best option. It's generally better to restart afresh.

3.8.4 Quitting a Jupyter Notebook

It goes almost without saying that before quitting a Jupyter Notebook, you should make sure you have saved the notebook by pressing the **Save Notebook** item in the **File** menu or its icon in the **Toolbar**.

When you are ready to quit working with a notebook, click on the **Shut Down** item in the **File** menu. Then, close your browser's JupyterLab tab to end the session.

Finally, return to the **Terminal** or **Anaconda Command Prompt** application. You should see the usual system prompt.

3.8.5 Working with an Existing Jupyter Notebook

To work with an existing Jupyter Notebook, open JupyterLab and use the **File Browser** pane to navigate to the directory, where the Jupyter Notebook is stored. Double-click on the file you want to open, and the Jupyter Notebook will appear in the main work area.

Note that while all the inputs and outputs from the previously saved session are visible in the notebook, the notebook has not been run. That means that none of the variables or other objects has been defined in this new session. To initialize all the objects in the file, you must rerun the file. To rerun the file, press the **Run** menu and select **Run All Cells**, which will re-execute all the cells.

CHAPTER 4

Strings, Lists, Arrays, and Dictionaries

The variables introduced in the previous chapter represent a very simple kind of data structure. This chapter introduces more sophisticated data structures, including ***strings****,* ***lists****,* ***tuples****, and* ***dictionaries****, which are all part of core Python. This chapter introduces the* ***NumPy array****, the principal structure you will use for storing and manipulating scientific data. This chapter introduces a powerful technique called* ***slicing****, which allows you to extract and manipulate sections of data contained in lists, tuples, and NumPy arrays. Finally, this chapter introduces some basic ideas about* ***objects****, which are central to Python's underlying structure and functioning.*

The most important data structure for scientific computing in Python is the **NumPy array**. NumPy arrays store lists of numerical data and are used to represent vectors, matrices, and even tensors. NumPy arrays are designed to manipulate large data sets quickly and efficiently. The NumPy library has many routines for creating, manipulating, and transforming NumPy arrays. NumPy functions, like `sqrt` and `sin`, are designed specifically to work with NumPy arrays. Core Python has an array data structure, but it's not nearly as versatile, efficient, or useful as the NumPy array. We will not be using Python arrays at all. Therefore, whenever we refer to an "array," we mean a "NumPy array." We discuss NumPy arrays in Section 4.4.

Lists are another data structure, similar to NumPy arrays, but unlike NumPy arrays, lists are a part of core Python. Lists have a variety of uses. They are used, for example, in various bookkeeping tasks that arise in computer

DOI: 10.1201/9781032673950-4

programming. Like arrays, they are sometimes used to store data. However, lists do not have the specialized properties and tools that make arrays so powerful for scientific computing. Therefore, we usually prefer arrays to lists for working with scientific data, but there are some circumstances for which using lists is preferable, even for scientific computing. And for other tasks, lists work just fine. We will use them frequently. We discuss them in Section 4.2.

Strings are lists of keyboard characters and other characters not on your keyboard. They are not particularly interesting in scientific computing but are necessary and useful. Texts on programming with Python often devote a good deal of time and space to learning about strings and how to manipulate them. However, our uses are relatively modest, so we take a minimalist approach and only introduce a few of their features. We discuss strings in Section 4.1.

Dictionaries are like lists but differ from lists in how their elements are accessed. The elements of lists and arrays are ordered sequentially, and to access an element of a list or an array, you refer to the number corresponding to its position in the sequence. The elements of dictionaries are accessed by "keys," which can be program strings or (arbitrary) integers (in no particular order). Dictionaries are an essential part of core Python. We introduce them in Section 4.3.

4.1 STRINGS

Strings are lists of characters. Any character you can type from a computer keyboard, plus various other characters, can be elements in a string. Strings are created by enclosing a sequence of characters within a pair of single or double quotes. Examples of strings include `"Marylyn"`, `'omg'`, `"good_bad_#5f>"`, `"{0:0.8g}"`, and `'We hold these truths ...'`. Caution: When defining a given string, the defining quotes must *both* be single or *both* be double. But you can use single quotes to define one string and double quotes to define the next string; it's up to you and has no consequence.

Strings can be assigned variable names

```
In[1]: a = "My dog's name is"
In[2]: b = "Bingo"
```

Note that we used double quotes to define the string `a`, so that we could use the apostrophe (single quote) in `dog's`. Strings can be concatenated using the "+" operator:

```
In[3]: c = a + " " + b
In[4]: c
Out[4]: "My dog's name is Bingo"
```

In forming the string `c`, we concatenated *three* strings, `a`, `b`, and a *string literal*, in this case, a space `" "`, which is needed to provide a space to separate string `a` from `b`.

"Wait a second?", you might say, "How can '+' be used both for adding numbers and for concatenating strings?" The short answer is "it just is." And it happens frequently enough that it's given a name: *operator overloading*. How an operator, in this case "+", gets used can depend on what data types it is operating on. For example, you can also multiply strings using the "*" operator.

```
In[5]: f = "Susie"
In[6]: 3 * f = 'SusieSusieSusie'
```

Given the definition of string addition, this definition makes sense

```
In[7]: f + f + f
Out[7]: 'SusieSusieSusie'
```

That is, `f + f + f = 3 * f`. Not all operators are overloaded in this way. For example, the division and power operators have no meaning with strings and will raise an error if you try to use them.

Finally, we note that because numbers—digits—are also alphanumeric characters, strings can include numbers:

```
In[8]: d = "927"
In[9]: e = 927
```

The variable `d` is a string, while the variable `e` is an integer. You get an error if you try to add them by writing `d + e`. However, if you type `d + str(e)` or `int(d) + e`, you get sensible results. Try them out!

You will use strings for different purposes: labeling data in data files, labeling axes in plots, formatting numerical output, requesting input for your programs, as arguments in functions, *etc*.

4.1.1 Unicode Characters

Originally, the set of characters available to computers was rather limited; there were 128. They were numbered from 0 to 127: 65–90 were used for capital letters from A to Z; 97–122 were used for lower case letters a–z. This coding is called ASCII (for American Standard Code for Information Interchange) These 128 characters represented the letters typically found on an American keyboard. To accommodate the much larger number of symbols needed to encode the world's languages, the International Standards Organization (ISO) developed the Unicode standard, which Python uses, specifically UTF-8, which you may see referenced from time to time. Numbers 0–127 are the same as the ASCII encoding. Numbers 945–970 encode the lowercase

Greek alphabet, for example. You can find the numerical value of a character using Python's `ord` (ordinal number) function. For example, `ord("a")` is 97. Conversely, the `chr` function returns the character corresponding to a particular number. For example, `chr(945)` returns the Greek letter alpha (α).

It is also common to specify the UTF-8 numerical code for Unicode characters in hexadecimal, or base 16, which uses 0–9 for the first 9 digits and then A–F for the hexadecimal ("hex" for short) digits 10–15. For example, the Unicode specification for the question mark symbol `?` is `chr(63)` in decimal format. The hexadecimal equivalent is `3F` $= 3 \times 16 + 15 = 63$. In Python, you can express this character as a 2-digit hex `\x3F` or as a 4-digit hex `\u003F` or as an 8-digit hex `\U0000003F`. Python uses `\x`, `\u`, and `\U` to express 2, 4, or 8-digit Unicode characters in hex. The hexadecimal digits A–F can be expressed as lowercase or uppercase letters. Here we demonstrate the use of unicode characters in the `print` function.

```
In[10]: print(chr(63))
        ?

In[11]: print("\x3f")
        ?

In[12]: print("\x3f\u003f\U0000003f")
        ???

In[13]: print(chr(945))
        α

In[14]: print("\u03b1")
        α
```

Note that you must include the leading zeros necessary so that `\x`, `\u`, and `\U` have precisely 2, 4, or 8 digits, respectively. Unicode can produce all kinds of characters: accented characters, subscripts, superscripts, and a host of others. You can find the Unicode encodings you want by searching on the web.

4.2 LISTS

Python has two data structures, *lists* and *tuples*, that consist of a list of one or more elements. The elements of lists or tuples can be numbers, strings, or both. Lists (we discuss tuples later in Section 4.2.4) are defined by a pair of *square* brackets on either end with individual elements separated by commas. Here are two examples of lists:

```
In[1]: a = [0, 1, 1, 2, 3, 5, 8, 13]
In[2]: b = [5., "girl", 2+0j, "horse", 21]
```

You can access individual elements of a list using the variable name for the list with an integer in square brackets:

```
 In[3]: b[0]
Out[3]: 5.0

 In[4]: b[1]
Out[4]: 'girl'

 In[5]: b[2]
Out[5]: (2+0j)
```

The first element of `b` is `b[0]`, the second is `b[1]`, the third is `b[2]`, and so on. Some computer languages index lists beginning with 0, like Python and C, while others index lists (or things more-or-less equivalent) beginning with 1 (like Fortran and MATLAB®). It's essential to remember that Python uses the former convention: lists are *zero-indexed*.

The last element of this array is `b[4]`, because `b` has 5 elements. The last element can also be accessed as `b[-1]`, no matter how many elements b has, and the next-to-last element of the list is `b[-2]`, *etc*. Try it out:

```
 In[6]: b[4]
Out[6]: 21

 In[7]: b[-1]
Out[7]: 21

 In[8]: b[-2]
Out[8]: 'horse'
```

Individual elements of lists can be changed. For example:

```
 In[9]: b
Out[9]: [5.0, 'girl', (2+0j), 'horse', 21]

 In[10]: b[0] = b[0]+2

 In[11]: b[3] = 3.14159

 In[12]: b
Out[12]: [7.0, 'girl', (2+0j), 3.14159, 21]
```

Here, you see that 2 was added to the previous value of `b[0]` and the string `'horse'` was replaced by the floating point number `3.14159`. You can also manipulate individual elements that are strings:

```
 In[13]: b[1] = b[1] + "s & boys"
 In[14]: b
Out[14]: [10.0, 'girls & boys', (2+0j), 3.14159, 21]
```

You can also add lists, but the result might surprise you:

```
 In[15]: a
Out[15]: [0, 1, 1, 2, 3, 5, 8, 13]

 In[16]: a+a
Out[16]: [0, 1, 1, 2, 3, 5, 8, 13, 0, 1, 1, 2, 3, 5, 8, 13]

 In[17]: a+b
Out[17]: [0, 1, 1, 2, 3, 5, 8, 13, 10.0, 'girls & boys',
           (2+0j), 3.14159, 21]
```

Adding lists concatenates them, just as the "+" operator concatenates strings.

4.2.1 Slicing Lists

You can access pieces of lists using the *slicing* feature of Python:

```
 In[18]: b
Out[18]: [10.0, 'girls & boys', (2+0j), 3.14159, 21]

 In[19]: b[1:4]
Out[19]: ['girls & boys', (2+0j), 3.14159]

 In[20]: b[3:5]
Out[20]: [3.14159, 21]
```

You access a subset of a list by specifying two indices separated by a colon ":". This is a powerful feature of lists that is used often. Here are a few other useful slicing shortcuts:

```
 In[21]: b[2:]
Out[21]: [(2+0j), 3.14159, 21]

 In[22]: b[:3]
Out[22]: [10.0, 'girls & boys', (2+0j)]

 In[23]: b[:]
Out[23]: [10.0, 'girls & boys', (2+0j), 3.14159, 21]
```

Thus, if the left slice index is `0`, you can leave it out; similarly, if the right slice index is the length of the list, you can leave it out also.

What does the following slice of an array give you?

```
 In[24]: b[1:-1]
```

You can get the length of a list using Python's `len` function:

```
 In[25]: len(b)
Out[25]: 5
```

Using a double colon, you can also extract every second, third, or n^{th} element of a list. Here we extract every second and third element of a list starting at different points:

```
 In[26]: b
Out[26]: [10.0, 'girls & boys', (2+0j), 3.14159, 21]

 In[27]: b[0::2]
Out[27]: [10.0, (2+0j), 21]

 In[28]: b[1::2]
Out[28]: ['girls & boys', 3.14159]

 In[29]: b[0::3]
Out[29]: [10.0, 3.14159]

 In[30]: b[1::3]
Out[30]: ['girls & boys', 21]

 In[31]: b[2::3]
Out[31]: [(2+0j)]
```

4.2.2 Multidimensional Lists

You can also make multidimensional lists or lists of lists. Consider, for example, a list of three elements, where each element in the list is itself a list:

```
 In[32]: a = [[3, 9], [8, 5], [11, 1]]
```

Here, we have introduced a three-element list, where each element consists of a two-element list. Such constructs can be useful in making tables and other structures. They also become relevant later when we introduce NumPy arrays and matrices in Section 4.4.

You can access the various elements of a list with a straightforward extension of the indexing scheme we have been using. The first element of the list a above is `a[0]`, which is `[3, 9]`; the second is `a[1]`, which is `[8, 5]`. The first element of `a[1]` is accessed as `a[1][0]`, which is 8, as illustrated below:

```
 In[33]: a[0]
Out[33]: [3, 9]

 In[34]: a[1]
Out[34]: [8, 5]
```

```
In[35]: a[1][0]
Out[35]: 8

In[36]: a[2][1]
Out[36]: 1
```

4.2.3 Appending to Lists

When you concatenate two lists, you create a new list. For example

```
In[37]: g = [9, 7, 5]

In[38]: id(g)
Out[38]:: 140316326106560

In[39]: g = g + [1, 2, 3]

In[40]: g
Out[40]: [9, 7, 5, 1, 2, 3]

In[41]: id(g)
Out[41]: 140316591106432
```

When the list `[1, 2, 3]` was added to `g`, `id(g)` changed, meaning that a new list object was created. However, if we use the `+=` operator, the `id(g)` doesn't change, as demonstrated here

```
In[42]: g += [4, 5, 6]

In[43]: g
Out[43]: [9, 7, 5, 1, 2, 3, 4, 5, 6]

In[44]: id(g)
Out[44]: 140316591106432
```

The list `g` has the same id before and after we use the `+=` operator to add `[4, 5, 6]`. We say that `g` has been changed *in place*, that is, without changing its id or memory location. By the way, the `*=` operator also makes changes in place.

The list method `append` also adds to a list *in place*.

```
In[45]: g.append(11)

In[46]: g
Out[46]: [9, 7, 5, 1, 2, 3, 4, 5, 6, 11]

In[47]: id(g)
Out[47]: 140316591106432
```

Interestingly, `append` behaves a bit differently from the operator `+=`, as illustrated here.

```
 In[48]: g.append([13, 14, 15])

 In[49]: g
Out[49]: [9, 7, 5, 1, 2, 3, 4, 5, 6, 11, [13, 14, 15]]

 In[50]: id(g)
Out[50]: 140316591106432
```

The `append` method added the list `[13, 14, 15]` to `g`, not the individual elements, 1, 2, 3, which is how the `+=` operator works.

4.2.4 Tuples

Next, a word about tuples: tuples are lists that are *immutable*. That is, once defined, the individual elements of a tuple cannot be changed. Whereas a list is written as a sequence of numbers (or characters or other objects) enclosed in *square* brackets, a tuple is written as a sequence of numbers (or characters or other objects) enclosed in *round* parentheses. Individual elements of a tuple are addressed in the same way as individual elements of lists are addressed, but those individual elements cannot be changed. All of this is illustrated by this simple example:

```
 In[51]: c = (1, 1, 2, 3, 5, 8, 13)
 In[52]: c[4]
Out[52]: 5

 In[53]: c[4] = 7
```

```
Traceback (most recent call last):

File "<ipython-input-2-7cb42185162c>", line 1,
in <module>
c[4] = 7

TypeError: 'tuple' object does not support item
assignment
```

When we tried to change `c[4]`, the system returned an error because individual elements of a tuple cannot be changed—they are immutable.

In other ways, tuples behave like lists. You can concatenate them is the addition operator and create repetitive sequences using the multiplication operator.

```
 In[54]: (3, 4, 5) + (2, 4, 6)
```

```
Out[54]: (3, 4, 5, 2, 4, 6)
 In[55]: (7, 8, 9) * 3
Out[55]: (7, 8, 9, 7, 8, 9, 7, 8, 9)
```

Note: While lists of length 1 are written as `[a]`, tuples are written as `(a,)`. The comma is needed to tell Python that this is a tuple of length 1 and not a simple variable.

Multidimensional tuples work exactly like multidimensional lists, except they are immutable.

Tuples offer some degree of safety when you want to define lists of immutable constants. As we shall see in Section 7.2.3, this becomes particularly relevant when passing tuples and lists in functions

4.3 DICTIONARIES

A Python *list* is a collection of Python objects indexed by an ordered sequence of integers starting from zero. A **dictionary** is also a collection of Python objects, just like a list, but one indexed by strings or numbers (not necessarily integers and not in any particular order) or even tuples! Dictionaries are a part of core Python, just like lists.

Suppose you want to make a dictionary of room numbers indexed by the name of the person who occupies each room. You create a dictionary using curly brackets `{...}`.

```
In[1]: room = {"Yujin":309, "Jake":582, "Ja'Marr":764}
```

The dictionary above has three entries separated by commas, each consisting of a **key**, which in this case is a string, and a **value**, which is a room number. A colon separates each key and its value. The syntax for accessing the various entries is similar to that of a list, with the key replacing the index number. For example, to find out the room number of Ja'Marr, you type

```
 In[2]: room["Ja'Marr"]
Out[2]: 764
```

The key does not need to be a string; it can be any immutable Python object. So, a key can be a string, an integer, or even a tuple, but it can't be a list. The elements accessed by their keys need not be strings but can be almost any legitimate Python object, just as for lists. Here is a weird example.

```
 In[3]: weird = {"tank":52, 846:"horse", 'bones':
   ...:             [23, 'fox', 'grass'], 'phrase': 'I am here'}

 In[4]: weird["tank"]
Out[4]: 52
```

```
 In[5]: weird[846]
Out[5]: 'horse'

 In[6]: weird["bones"]
Out[6]: [23, 'fox', 'grass']

 In[7]: weird["phrase"]
Out[7]: 'I am here'
```

You can build up and add to dictionaries in a straightforward manner

```
 In[8]: d = {}

 In[9]: d["last name"] = "Alberts"

 In[10]: d["first name"] = "Marie"

 In[11]: d["birthday"] = "January 27"

 In[12]: d
Out[12]: {'birthday': 'January 27',
    ...:   'first name': 'Marie', 'last name': 'Alberts'}
```

You can get a list of all the keys or values of a dictionary by typing the dictionary name followed by `.keys()` or `.values()`

```
 In[13]: d.keys()
Out[13]: ['last name', 'first name', 'birthday']

 In[14]: d.values()
Out[14]: ['Alberts', 'Marie', 'January 27']
```

In other languages, data types similar to Python dictionaries may be called "hashmaps" or "associative arrays," so you may see such terms used if you read about dictionaries on the web.

You can also create a dictionary from a list of tuple pairs.

```
 In[15]: g = [("Melissa", "Canada"), ("Jeana", "China"),
         ("Etienne", "France")]

 In[16]: gd = dict(g)

 In[17]: gd
Out[17]: {'Melissa': 'Canada', 'Jeana': 'China',
           'Etienne': 'France'}

 In[18]: gd['Jeana']
Out[18]: 'China'
```

4.4 NUMPY ARRAYS

The NumPy array is the real workhorse of data structures for scientific and engineering applications. The NumPy array is similar to a list but where all the elements are the same type. The NumPy array constitutes a new type, namely `ndarray`. The elements of a NumPy array, or simply an *array*, are usually numbers but can also be Booleans (*i.e.*, `True` or `False`), strings, or other objects. When the elements are numbers, they must all be the same type. For example, they might be all integers or all floating point numbers.

4.4.1 Creating Arrays (1-d)

NumPy has many functions for creating arrays. We focus on four (or five or six, depending on how you count!). The **first** of these, the `array` function, converts a list to an array:

```
 In[1]: a = [0, 0, 1, 4, 7, 16, 31, 64, 127]

 In[2]: import numpy as np

 In[3]: b = np.array(a)

 In[4]: b
Out[4]: array([ 0, 0, 1, 4, 7, 16, 31, 64, 127])

 In[5]: c = np.array([1, 4., -2, 7])

 In[6]: c
Out[6]: array([ 1., 4., -2., 7.])
```

Notice that `b` is an integer array since it was made from a list of integers. On the other hand, `c` is a floating point array even though only one of the elements of the list from which it was made was a floating point number. The `array` function automatically promotes all numbers to the type of the most general entry in the list, which in this case is a floating point number. When a list consists of numbers and strings, all the elements become strings when an array is formed from the list.

The **second** way arrays can be created is with the NumPy `linspace` or `logspace` functions. The `linspace` function creates an array of N evenly spaced points between a starting point and an ending point. The form of the function is `linspace(start, stop, N)`. If the third argument `N` is omitted, then `N=50`.

```
 In[7]: np.linspace(0, 10, 5)
Out[7]: array([ 0. , 2.5, 5. , 7.5, 10. ])
```

The `linspace` function produced five evenly spaced points between 0 and 10 inclusive. NumPy also has a closely related function `logspace` that produces evenly spaced points on a logarithmic scale. The arguments are the same as those for `linspace` except that `start` and `stop` are a power of 10. That is, the array starts at 10^{start} and ends at 10^{stop}.

```
 In[8]: %precision 1 # display 1 digit after decimal
Out[8]: '%.1f'

 In[9]: np.logspace(1, 3, 5)
Out[9]: array([ 10. , 31.6, 100. , 316.2, 1000. ])
```

The `logspace` function created an array with 5 points evenly spaced on a logarithmic axis starting at 10^1 and ending at 10^3. The `logspace` function is handy when you want to create a logarithmic plot.

A **third** way arrays can be created is using the NumPy `arange` function. The form of the function is `arange(start, stop, step)`. If the third argument is omitted `step=1`. If the first and third arguments are omitted, then `start=0` and `step=1`.

```
 In[10]: np.arange(0, 10, 2)
Out[10]: array([0, 2, 4, 6, 8])

 In[11]: np.arange(0., 10, 2)
Out[11]: array([ 0., 2., 4., 6., 8.])

 In[12]: np.arange(0, 10, 1.5)
Out[12]: array([ 0. , 1.5, 3. , 4.5, 6. , 7.5, 9. ])
```

The `arange` function produces points evenly spaced between 0 and 10, exclusive of the final point. Notice that `arange` produces an integer array in the first case but a floating point array in the other two cases. In general `arange` produces an integer array if the arguments are all integers; making any of the arguments a float causes the array created to be a float.

A **fourth** way to create an array is with the `zeros` and `ones` functions. As their names imply, they create arrays where all the elements are either zeros or ones. They each take one mandatory argument, the number of elements in the array, and one optional argument that specifies the data type of the array. Left unspecified, the data type is a float. Here are three examples

```
 In[13]: np.zeros(6)
Out[13]: array([ 0., 0., 0., 0., 0., 0.])

 In[14]: np.ones(8)
Out[14]: array([ 1., 1., 1., 1., 1., 1., 1., 1.])

 In[15]: ones(8, dtype=int)
Out[15]: np.array([1, 1, 1, 1, 1, 1, 1, 1])
```

4.4.1.1 *Recap of Ways to Create a 1-d NumPy Array*

`array(a)`: Creates an array from the list `a`.

`linspace(start, stop[, num])`: Returns `num` evenly spaced numbers over an interval from `start` to `stop` inclusive. (`num=50` if omitted.)

`logspace(start, stop[, num])`: Returns `num` logarithmically spaced numbers over an interval from 10^{start} to 10^{stop} inclusive. (`num=50` if omitted.)

`arange([start,] stop[, step,], dtype=None)`: Returns data points from `start` to `end`, exclusive, evenly spaced by `step`. (`step=1` if omitted. `start=0` and `step=1` if both are omitted.)

`zeros(num[, dtype=float])`: Returns an an array of 0s with `num` elements. An optional `dtype` argument can be used to set the data type; left unspecified, a float array is created.

`ones(num[, dtype=float])`: Returns an an array of 1s with `num` elements. Optional `dtype` argument can be used to set the data type; left unspecified, a float array is created.

4.4.2 Mathematical Operations with Arrays

The utility and power of arrays in Python come from the fact that you can process and transform all the elements of an array in one fell swoop. The best way to see how this works is to look at an example.

```
 In[16]: a = np.linspace(-1., 5, 7)

 In[17]: a
Out[17]: array([-1., 0., 1., 2., 3., 4., 5.])

 In[18]: a*6
Out[18]: array([ -6., 0., 6., 12., 18., 24., 30.])
```

Each element of the array has been multiplied by 6. This works not only for multiplication but for any other mathematical operation you can imagine: division, exponentiation, *etc.*

```
 In[19]: a/5
Out[19]: array([-0.2, 0.  , 0.2, 0.4, 0.6, 0.8, 1. ])

 In[20]: a**3
Out[20]: array([ -1., 0., 1., 8., 27., 64., 125.])

 In[21]: a+4
```

```
Out[21]: array([ 3., 4., 5., 6., 7., 8., 9.])

 In[22]: a-10
Out[22]: array([-11., -10., -9., -8., -7., -6., -5.])

 In[23]: (a+3)*2
Out[23]: array([ 4., 6., 8., 10., 12., 14., 16.])

 In[24]: np.sin(a)
Out[24]: array([-0.8415,  0.    ,  0.8415,  0.9093,  0.1411,
                -0.7568, -0.9589])
```

Here, we set `precision 4` so that only 4 digits are displayed to the right of the decimal point.[1] We will typically do this without explicitly mentioning it to have neater formatting.

```
 In[25]: np.exp(-a)
Out[25]: array([ 2.7183,  1.    ,  0.3679,  0.1353,  0.0498,
                 0.0183,  0.0067])

 In[26]: 1. + np.exp(-a)
Out[26]: array([ 3.7183,  2.    ,  1.3679,  1.1353,  1.0498,
                 1.0183,  1.0067])

 In[27]: b = 5*np.ones(8)

 In[28]: b
Out[28]: array([ 5., 5., 5., 5., 5., 5., 5., 5.])

 In[29]: b += 4

 In[30]: b
Out[30]: array([ 9., 9., 9., 9., 9., 9., 9., 9.])
```

In each case, you can see that the same mathematical operations are performed individually on each array element. Even fairly complex algebraic computations can be carried out this way.

Let's say you want to create an $x - y$ data set of $y = \cos x$ *vs.* x over the interval from -3.14 to 3.14. Here is how you might do it.

```
 In[31]: x = np.linspace(-3.14, 3.14, 21)

 In[32]: y = np.cos(x)

 In[33]: x
Out[33]: array([-3.14 , -2.826, -2.512, -2.198, -1.884,
```

[1]This works when printing from an IPython shell. To achieve the same effect within a program, use `np.set_printoptions(precision=4)`.

```
                 -1.57 , -1.256, -0.942, -0.628, -0.314,
                  0.   ,  0.314,  0.628,  0.942,  1.256,
                  1.57 ,  1.884,  2.198,  2.512,  2.826,
                  3.14 ])


 In[34]: y
Out[34]: array([ -1.0000e+00,  -9.5061e-01,  -8.0827e-01,
                  -5.8688e-01,  -3.0811e-01,   7.9633e-04,
                   3.0962e-01,   5.8817e-01,   8.0920e-01,
                   9.5111e-01,   1.0000e+00,   9.5111e-01,
                   8.0920e-01,   5.8817e-01,   3.0962e-01,
                   7.9633e-04,  -3.0811e-01,  -5.8688e-01,
                  -8.0827e-01,  -9.5061e-01,  -1.0000e+00])
```

You can use arrays as inputs for any of the functions introduced in Section 2.6.1.

You might well wonder what happens if Python encounters an illegal operation. Here is one example.

```
 In[35]: a
Out[35]: array([-1., 0., 1., 2., 3., 4., 5.])

 In[36]: np.log(a)
Out[36]: array([    nan,    -inf,  0.     ,  0.6931,  1.0986,
                  1.3863,  1.6094])
```

NumPy calculates the logarithm where it can, and returns `nan` (not a number) for an illegal operation, taking the logarithm of a negative number, and `-inf`, or $-\infty$ for the logarithm of zero. The other values in the array are correctly reported. Depending on the settings of your version of Python, NumPy may also print a warning message to let you know that something untoward has occurred.

Arrays can also be added, subtracted, multiplied, and divided by each other on an element-by-element basis, provided the two arrays have the same size. Consider adding the two arrays `a` and `b` defined below:

```
 In[37]: a = np.array([34., -12, 5.])

 In[38]: b = np.array([68., 5.0, 20.])

 In[39]: a+b
Out[39]: array([ 102., -7., 25.])
```

The result is that each element of the two arrays are added. Similar results are obtained for subtraction, multiplication, and division:

```
 In[40]: a - b
```

```
Out[40]: array([-34., -17., -15.])

 In[41]: a * b
Out[41]: array([ 2312., -60., 100.])

 In[42]: a / b
Out[42]: array([ 0.5 , -2.4 , 0.25])
```

These operations with arrays are called *vectorized* operations because the entire array, or "vector," is processed as a unit. Vectorized operations are much faster than processing each element of an array one by one. Writing code that takes advantage of these kinds of vectorized operations is almost always preferred to other means of accomplishing the same task because it is faster and syntactically simpler. You will see examples later when we discuss loops in Chapter 6.

4.4.3 Slicing and Addressing Arrays

Arrays can be sliced in the same ways that strings and lists can be sliced. Ditto for accessing individual array elements: 1-d arrays are addressed like strings and lists. Slicing and vectorized operations can lead to some pretty compact and powerful code.

Suppose, for example, that you have two arrays `y`, and `t` for position *vs.* time of a falling object, say a ball, and you want to use these data to calculate the velocity as a function of time:

```
 In[43]: y = np.array([0., 1.3, 5. , 10.9, 18.9, 28.7, 40.])

 In[44]: t = np.array([0., 0.49, 1. , 1.5 , 2.08, 2.55, 3.2])
```

You can find the average velocity for time interval i using the formula

$$v_i = \frac{y_i - y_{i-1}}{t_i - t_{i-1}} \,.$$

You can easily calculate the entire array of velocities using the slicing and vectorized subtraction properties of NumPy arrays by noting that you can create two `y` arrays displaced by one index.

```
 In[45]: y[:-1]
Out[45]: array([ 0. , 1.3, 5. , 10.9, 18.9, 28.7])

 In[46]: y[1:]
Out[46]: array([ 1.3, 5. , 10.9, 18.9, 28.7, 40. ])
```

The element-by-element difference of these two arrays is

```
 In[47]: y[1:]-y[:-1]
Out[47]: array([ 1.3, 3.7, 5.9, 8. , 9.8, 11.3])
```

The element-by-element difference of the two arrays `y[1:]-y[:-1]` divided by `t[1:]-t[:-1]` gives the entire array of velocities.

```
 In[48]: v = (y[1:]-y[:-1])/(t[1:]-t[:-1])

 In[49]: v
Out[49]: array([  2.6531,   7.2549,  11.8   ,
                13.7931,  20.8511,  17.3846])
```

Of course, these are the average velocities over each interval so the times best associated with each interval are the times halfway in between the original time array, which you can calculate using a similar trick of slicing:

```
 In[50]: tv = (t[1:]+t[:-1])/2.

 In[51]: tv
Out[51]: array([ 0.245, 0.745, 1.25 , 1.79 , 2.315, 2.875])
```

4.4.4 Fancy Indexing: Boolean Indexing

There is another way of accessing various elements of an array that is both powerful and useful. We illustrate with a simple example. Consider the following array:

```
 In[52]: b = 1.0 / np.arange(0.2, 3, 0.2)

 In[53]: b
Out[53]:
array([ 5.        ,  2.5       ,  1.66666667,  1.25      ,
        1.        ,  0.83333333,  0.71428571,  0.625     ,
        0.55555556,  0.5       ,  0.45454545,  0.41666667,
        0.38461538,  0.35714286])
```

Suppose you want just those elements of the array that are greater than one. You can get an array of those values using *Boolean indexing*. Here's how it works:

```
 In[54]: b[b > 1]
Out[54]: array([ 5.       ,  2.5      ,  1.66666667,  1.25    ])
```

Only those elements whose values meet the Boolean criterion of `b > 1` are returned.

Boolean indexing can be useful for reassigning values of an array that meet some criterion. For example, you can reassign all the elements of `b` that are greater than 1 to have a value of 1 with the following assignment:

```
 In[55]: b[b > 1] = 1

 In[56]: b
Out[56]:
array([ 1.        ,  1.        ,  1.        ,  1.        ,
        1.        ,  0.83333333,  0.71428571,  0.625     ,
        0.55555556,  0.5       ,  0.45454545,  0.41666667,
        0.38461538,  0.35714286])
```

Suppose we create another array the same size as `b`.

```
 In[57]: b.size
Out[57]: 14

 In[58]: c = np.linspace(0, 10, b.size)

 In[59]: c
Out[59]:
array([  0.        ,   0.76923077,   1.53846154,   2.30769231,
         3.07692308,   3.84615385,   4.61538462,   5.38461538,
         6.15384615,   6.92307692,   7.69230769,   8.46153846,
         9.23076923,  10.        ])
```

Now we would like for this new array `c` to be equal to 3 everywhere that `b` is equal to 1. We do that like this:

```
 In[60]: c[b == 1] = 3

 In[61]: c
Out[61]:
array([  3.        ,   3.        ,   3.        ,   3.        ,
         3.        ,   3.84615385,   4.61538462,   5.38461538,
         6.15384615,   6.92307692,   7.69230769,   8.46153846,
         9.23076923,  10.        ])
```

Here, we used the Boolean operator `==`, which returns a value of `True` if the two things it's comparing have the same value and `False` if they do not. So, a Boolean condition on one array can be used to index a different array *if the two arrays have the same size*, as in the above example.

As illustrated in the next example, the array elements selected using Boolean indexing need not be consecutive.

```
 In[62]: y = np.sin(np.linspace(0, 4*np.pi, 9))

 In[63]: y
Out[63]:
array([  0.00000000e+00,   1.00000000e+00,   1.22464680e-16,
        -1.00000000e+00,  -2.44929360e-16,   1.00000000e+00,
         3.67394040e-16,  -1.00000000e+00,  -4.89858720e-16])
```

```
In[64]: y[np.abs(y) < 1.e-15] = 0.

In[65]: y
Out[65]: array([ 0.,  1.,  0., -1.,  0.,  1.,  0., -1.,  0.])
```

Boolean indexing provides a nifty way to eliminate those tiny numbers that should be but aren't quite zero due to round-off error, even if the benefit is primarily aesthetic.

4.4.5 Multidimensional Arrays and Matrices

So far, we have examined only one-dimensional NumPy arrays, that is, arrays that consist of a simple sequence of numbers. However, NumPy arrays can be used to represent multidimensional arrays. For example, you may be familiar with the concept of a *matrix*, which consists of a series of rows and columns of numbers. Matrices can be represented using two-dimensional NumPy arrays. Higher dimension arrays can also be created as the application demands.

4.4.5.1 Creating Multidimensional NumPy Arrays

There are several ways of creating multidimensional NumPy arrays. The most straightforward way is using NumPy's `array` function, which we demonstrate here:

```
In[66]: b = np.array([[1., 4, 5], [9, 7, 4]])

In[67]: b
Out[67]: array([[ 1.,  4.,  5.],
                [ 9.,  7.,  4.]])
```

Notice the syntax used above in which two one-dimensional lists [1., 4, 5] and [9, 7, 4] are enclosed in square brackets to make a two-dimensional list. The array function converts the two-dimensional list, a structure we introduced earlier, to a two-dimensional array. When converting from a list to an array, the array function makes all the elements have the same data type as the most complex entry, in this case, a float. This reminds us of an essential difference between NumPy arrays and lists: all elements of a NumPy array must be of the same data type: floats, integers, or complex numbers, *etc*.

There are several other functions for creating multidimensional arrays. For example, a 3-row by 4-column array or 3×4 array with all the elements filled with 1 can be created using the `ones` function introduced earlier.

```
In[68]: a = np.ones((3, 4), dtype=float)

In[69]: a
```

```
Out[69]: array([[1., 1., 1., 1.],
                [1., 1., 1., 1.],
                [1., 1., 1., 1.]])
```

The first argument of the `ones` function is a tuple specifying the size and shape of the array, in this case, a two-dimensional array with 3 rows and 4 columns. The `zeros` function can be used similarly to create a multidimensional array of zeros.

The `eye(N)` function creates an $N \times N$ two-dimensional identity matrix with ones along the diagonal:

```
In[70]: np.eye(4)
Out[70]: array([[1., 0., 0., 0.],
                [0., 1., 0., 0.],
                [0., 0., 1., 0.],
                [0., 0., 0., 1.]])
```

4.4.5.2 Reshaping Arrays

Multidimensional arrays can also be created from one-dimensional arrays. For example, a 2×3 array can be created as follows:

```
In[71]: c = np.arange(6)

In[72]: c
Out[72]: array([0, 1, 2, 3, 4, 5])

In[73]: c = c.reshape(2, 3)

In[74]: c
Out[74]: array([[0, 1, 2],
                [3, 4, 5]])
```

In this example, we specified both the number of rows (2) and the number of columns (3), but we didn't have to. As the following syntax illustrates, we can simply specify the number of rows *or* the number of columns:

```
In[75]: c = np.arange(6)

In[76]: c
Out[76]: array([0, 1, 2, 3, 4, 5])

In[77]: c.reshape(2, -1)
Out[77]: array([[0, 1, 2],
                [3, 4, 5]])

In[78]: c.reshape(-1, 3)
Out[78]: array([[0, 1, 2],
                [3, 4, 5]])
```

This brings us to another point. Reshaping arrays can be used to perform useful operations on arrays efficiently. Suppose you have a 1-D array, perhaps very long, that you want to shorten by creating a new array that is the average, or perhaps the sum, of every three elements. You can do this by reshaping the array into a $3 \times n$ array and then averaging or summing the rows to create the new array you want. Let's give it a go!

```
 In[79]: d = np.array([5, 2, 8, -4, 1, -6, 8, 6, 4, 5, -9, 2])

 In[80]: d.reshape(-1, 3)
Out[80]: array([[ 5,  2,  8],
                [-4,  1, -6],
                [ 8,  6,  4],
                [ 5, -9,  2]])

 In[81]: np.mean(d.reshape(-1, 3), axis=1)
Out[81]: array([ 5.    , -3.       ,  6.       , -0.66666667])

 In[82]: np.sum(d.reshape(-1, 3), axis=1)
Out[82]: array([15, -9, 18, -2])
```

Writing `axis=1` specifies that the *rows* of the array are averaged or summed; writing `axis=0` averages or sums the *columns*.

4.4.5.3 Indexing Multidimensional Arrays

The individual elements of arrays can be accessed in the same way as for lists:

```
 In[83]: b[0][2]
Out[83]: 5.0
```

You can also use the syntax

```
 In[84]: b[0, 2]
Out[84]: 5.0
```

which gives the same result. Caution: both the `b[0][2]` and the `b[0, 2]` syntax work for NumPy arrays and give the same result; for lists, only the `b[0][2]` syntax works.

Note, however, that for very large multidimensional NumPy arrays, the `b[i,j]` syntax is strongly preferred to the `b[i][j]` syntax. This is because using the `b[i][j]` syntax first creates the `b[i]` 1D array and then looks for the j^{th} element in that array. By contrast, `b[i,j]` directly accesses element (i, j) of the 2D `b[i,j]` array.

4.4.5.4 *Array (matrix) Operations*

Addition, subtraction, multiplication, division, and exponentiation all work with multidimensional arrays the same way they work with one-dimensional arrays on an element-by-element basis, as illustrated below:

```
 In[85]: b
Out[85]: array([[ 1., 4., 5.],
                [ 9., 7., 4.]])

 In[86]: 2*b
Out[86]: array([[ 2., 8., 10.],
                [ 18., 14., 8.]])
 In[87]: b/4.
Out[87]: array([[ 0.25, 1.  , 1.25],
                [ 2.25, 1.75, 1.  ]])

 In[88]: b**2
Out[88]: array([[ 1., 16., 25.],
                [ 81., 49., 16.]])

 In[89]: b-2
Out[89]: array([[-1., 2., 3.],
                [ 7., 5., 2.]])
```

Functions also act on an element-by-element basis.

```
 In[90]: np.sin(b)
Out[90]: array([[ 0.8415, -0.7568, -0.9589],
         [ 0.4121,  0.657 , -0.7568]])
```

Binary operations like adding, subtracting, multiplying, and dividing two arrays are performed element-by-element. For example using the matrices `b` and `c` defined above, multiplying them together gives

```
 In[91]: b
Out[91]: array([[ 1., 4., 5.],
                [ 9., 7., 4.]])

 In[92]: c
Out[92]: array([[0, 1, 2],
                [3, 4, 5]])

 In[93]: b*c
Out[93]: array([[ 0., 4., 10.],
                [ 27., 28., 20.]])
```

Of course, this requires that both arrays have the same shape. Beware: array multiplication, done on an element-by-element basis, is not the same as matrix multiplication as defined in linear algebra. Therefore, in Python, we distinguish between *array* multiplication and *matrix* multiplication.

4.4.5.5 *Matrix Multiplication: Dot, Cross, and Outer Products*

Normal matrix multiplication is perfomred with NumPy's `dot` function. Here we demonstrate this capability by multiplying `c`, which is a 2×3 array by `d`, a 3×2 array:

```
In[94]: d = np.array([[4, 2], [9, 8], [-3, 6]])

In[95]: d
Out[95]: array([[ 4,  2],
                [ 9,  8],
                [-3,  6]])

In[96]: np.dot(b, d)
Out[96]: array([[25., 64.],
                [87., 98.]])
```

The cross product of two vectors $\mathbf{g} \times \mathbf{h}$ can be applied to 2 and 3-element one-dimensional arrays using NumPy's `cross` function:

```
In[97]: g = np.array([3, 5, 6])

In[98]: h = np.array([-4, 3, 7])

In[99]: np.cross(g, h)
Out[99]: array([ 17, -45,  29])
```

The outer product of two vectors $\mathbf{h} \otimes \mathbf{k}$ is calculated using NumPy's `outer` function:

```
In[100]: k = np.array([-7, 4])

In[101]: np.outer(h, k)
Out[101]: array([[ 28, -16],
                 [-21,  12],
                 [-49,  28]])
```

4.4.6 Broadcasting

In the previous section, we introduced NumPy array operations. You learned how to apply different numerical operations to every element in an array. You also learned that you can apply binary operations, like addition, subtraction, multiplication, and division, to arrays of the same size and shape on an element-by-element basis.

However, binary operations between NumPy arrays are much more versatile than these simple examples illustrate. For example, suppose you have a 4×3 array, and you want to multiply the first column by 2, the second column by -3, and the third column by 5.5. This is easy to do with NumPy arrays. First, create a 4×3 array:

```
In[102]: p = np.linspace(1, 12, 12).reshape(4, 3)

In[103]: p
Out[103]: array([[ 1.,  2.,  3.],
                 [ 4.,  5.,  6.],
                 [ 7.,  8.,  9.],
                 [10., 11., 12.]])
```

Next, create an array of the column multipliers:

```
In[104]: mc = np.array([2, -3, 5.5])
```

Now multiply the two NumPy arrays:

```
In[105]: p * mc
Out[105]: array([[  2. ,  -6. ,  16.5],
                [  8. , -15. ,  33. ],
                [ 14. , -24. ,  49.5],
                [ 20. , -33. ,  66. ]])
```

Each column of `p` is multiplied by the corresponding column of `mc`. This kind of array multiplication is commutative, so `p * mc = mc * p`.

The term *broadcasting* is used to describe this kind of behavior, where the smaller array is *broadcast* over the larger array so that their shapes are compatible. Here, the one 3-element row of `mc` was broadcast over the four 3-element rows of `p`. Of course, this only works if both `mc` and `p` have the same number of columns.

You can also perform operations on rows. For example, suppose you want to add 2 to the first row, -2 to the second row, -3.5 to the third row, and 6 to the fourth row. In this case, you need to create a 4×1 column array:

```
In[106]: ar = np.array([2, -2, -3, 5.5])

In[107]: ar.shape = (4, 1)

In[108]: ar
Out[108]: array([[ 2. ],
                 [-2. ],
                 [-3. ],
                 [ 5.5]])
```

Note that to make a NumPy column vector, we make a 2-dimensional array where the second dimension is 1. Now we add the two NumPy arrays:

```
 In[109]: p + ar
Out[109]: array([[ 3. ,   4. ,   5. ],
                 [ 2. ,   3. ,   4. ],
                 [ 4. ,   5. ,   6. ],
                 [15.5,  16.5,  17.5]])
```

Once again, we note that this kind of array addition is commutative: `p + ar = ar + p`. Note that `p - ar` and `ar - p` both give 3 × 4 arrays, but they are different, as subtraction is not commutative.

More complex broadcasting is possible but seldom necessary. If you are curious, you can investigate further in the online NumPy documentation.

4.4.7 Differences Between Lists and Arrays

While lists and arrays are superficially similar—they are both multi-element data structures—they behave quite differently in a number of circumstances. Here, we list some of the differences between Python lists and NumPy arrays, and why you might prefer to use one or the other depending on the circumstance.

- **Lists are part of the core Python programming language; arrays are a part of NumPy.** Therefore, before using NumPy arrays, you must issue the command `import numpy as np`.

- **The elements of a NumPy array must all be of the same type**, whereas the elements of a Python list can consist of different types.

- **Arrays allow Boolean indexing; lists do not.** See Section 4.4.4.

- **NumPy arrays support "vectorized" operations** like element-by-element addition and multiplication. This is made possible, in part, by the fact that all array elements have the same type, which allows array operations like element-by-element addition and multiplication to be carried out very efficiently. Such "vectorized" operations on arrays, which includes operations by NumPy functions such as `numpy.sin` and `numpy.exp`, are much faster than operations performed by loops using the core Python `math` package functions, such as `math.sin` and `math.exp`, that act only on individual elements and not on whole lists or arrays.

- **Adding one or more additional elements to a NumPy array creates a new array and destroys the old one.** Therefore, building up large arrays by appending elements one by one can be inefficient, especially if the array is large because you repeatedly create and destroy large arrays.

By contrast, elements can be added to a list without creating a whole new list. If you need to build an array element by element, it is usually better to build it as a list and then convert it to an array when the list is complete. At this point, it may not be easy to appreciate how and under what circumstances you might want to build up an array element by element. Examples are provided later on (*e.g.*, see Section 7.1.1).

4.5 OBJECTS

We have already mentioned that Python is an object-oriented programming language starting on page 2! What it means for a programming language to be object-oriented is multi-faceted and involves software and programming design principles that go beyond what you need right now. So, rather than attempt some definition that encompasses all of what an object-oriented (OO) approach means, we introduce various aspects of the OO approach as needed. In this section, we want to introduce you to some simple (perhaps deceptively so) ideas of the OO approach.

The first is the idea of an *object*. One way to think of an object is as a collection of data bundled with functions that can operate on that data. Everything you encounter in Python is an object, including the data structures discussed in this chapter: strings, lists, arrays, and dictionaries. In this case, the data are the contents of these various objects. Associated with each of these kinds of objects are *methods* that act on the data of these objects. For example, consider the string

```
In[1]: c = "My dog's name is Bingo"
```

One method associated with a string object is `split()`, which we invoke using the dot notation we've encountered before:

```
In[2]: c.split()
Out[2]: ['My', "dog's", 'name', 'is', 'Bingo']
```

The method `split()` acts on the object it's attached to by the dot. It has a matching set of parentheses to indicate that it's a function (that acts on the object `c`). Without an argument, `split()` splits the string `c` at the spaces into separate strings, five in this case, and returns the set of split strings as a list. By specifying an argument to `split()`, we can split the string elsewhere, say at the "g":

```
In[3]: c.split('g')
Out[3]: ['My do', "'s name is Bin", 'o']
```

Notice that the `split()` method dispensed with the `g` just like it got rid of all the spaces when we used it without an argument.

There are many more string methods, which we do not explore here, as our point isn't to give an exhaustive introduction to string methods and their uses. Instead, it's simply to introduce the idea of object methods.

Lists, arrays, and dictionaries are also objects with associated methods. We already introduced the `reshape` method for NumPy arrays on 70. But there are many more array methods. Consider the following 2-row array:

```
 In[4]: b = np.array([[1., 4., 5.], [9., 7., 4.]])

 In[5]: b
Out[5]: array([[1., 4., 5.],
               [9., 7., 4.]])
 In[6]: b.mean()
Out[6]: 5.0

 In[7]: b.shape
Out[7]: (2, 3)
```

The method `mean()` is a function that calculates the mean value of the elements (the data) of the array.

Writing `b.shape` returns the number of rows and columns in the array; `b.shape` is a *attribute* or the array object. Note that there are no parentheses associated with `b.shape`. That's because `b.shape` is an attribute of the array and, as such, is stored within the object. Writing `b.shape` simply looks up the attribute's value and reports it to you. An attribute of a given instance or realization of an object is also sometimes called an *instance variable*.

We stated above that methods are functions associated with objects that act on the object's data. Methods can also act on any attribute of an object. Recall the `reshape()` method for NumPy arrays introduced earlier in this chapter; it changes the `shape` attribute of an array. See, for example, page 70, where we used the `reshape()` method to change `c.shape` from `6` to `(2, 3)`.

To summarize, objects have associated with them data, *attributes* that in one way or another characterize the data, and *methods*, which are functions that act on the data or attributes. Each kind of object, such as strings, lists, dictionaries, and arrays, has its own set of attributes and methods uniquely associated with that object type. So, while `split()` is a method associated with strings, it is not a method of arrays. Thus, typing `b.split()` returns an error message, since `b` is an array.

```
 In[8]: b.split()
```

```
Traceback (most recent call last):

File "<ipython-input-11-0c30fe27ab6f>", line 1, in <module>
b.split()
```

TABLE 4.1 Some NumPy array attributes and methods.

attribute	**Output**
`.size`	number of elements in array
`.shape`	number of rows, columns, *etc.*
`.ndim`	number of array dimensions
`.real`	real part of array
`.imag`	imaginary part of array
method	**Output**
`.mean()`	average value of array elements
`.std()`	standard deviation of array elements
`.min()`	return minimum value of array
`.max()`	return maximum value of array
`.sum()`	return sum of elements of array
`.prod()`	return produce of elements of array
`.abs()`	absolute value of each element
`.conj()`	complex-conjugate each element
`.sort()`	low-to-high sorted array (in place)
`.reshape(n, m)`	Returns an `n`×`m` 2-dimensional array with same elements
`.flatten()`	Returns a copy of the (multidimensional) array collapsed into one dimension

```
AttributeError: 'numpy.ndarray' object has no attribute
'split'
```

You will often use and interact with object attributes and methods in your Python journey. Table 4.1 summarizes a few of the attributes and methods of NumPy arrays.

4.6 EXERCISES

1. This is an exercise on looking up things on the web. In the text, we mentioned that there are string methods like `split` that perform various operations on strings (see page 76). Recall that the syntax for methods is to put a dot after the object, a string in this case, followed by the method name and parentheses, which may or may not need an argument. So, consider the string `a = "chemical"`. By looking around on the web, find Python string methods that can perform the following tasks:

   (a) Capitalize the first letter.

(b) Print out the string in all uppercase.

(c) Replace instances of the substring `"cal"` with `"stry"`.

2. Make a list of integers from 0 to 12 using the following syntax:

```python
nums = list(range(13))
```

With the list `nums`, perform the following operations:

(a) Use slicing to print the numbers 5 to 9 exclusive (not including 9).

(b) Use slicing to print every third number beginning with 2.

(c) Use slicing to make a new list `numshort` consisting of all the interior elements, that is, excluding the first and last elements. Then print out `numshort`.

3. Make a dictionary named `greek` that outputs the first five letters of the Greek alphabet on your computer screen: α, β, γ, δ, ϵ. The keys to the dictionary should be the names of the letters written out in the Latin alphabet:

```python
gkeys = ["alpha", "beta", "gamma", "delta", "epsilon"]
```

Recall from page 53 that the UTF-8 encoding for the Greek alphabet begins at 945 (or in HEX: 0x3B1). It then proceeds consecutively through the Greek alphabet according to the list `gkeys` above. Demonstrate your code by showing that the following `print` statement outputs the five Greek letters:

```python
print(greek["alpha"], greek["beta"], greek["gamma"],
greek["delta"], greek["epsilon"])
```

4. Create an array of 9 evenly spaced numbers going from 1 to 29 (inclusive) and give it the variable name `r`. Print out the answers to parts (a)–(d) showing no more than two digits to the right of the decimal point.

(a) The array `r`.

(b) The cube of each array element (as simply as possible).

(c) Twice the value of each element of the array in two different ways: (*i*) using addition and (*ii*) using multiplication.

(d) The natural logarithm of each element of the array.

(e) The sums and means (averages) of each of the four arrays calculated in parts (a)–(d). Use the appropriate NumPy methods with the dot syntax. You may express your answers with an arbitrary number of digits.

5. Create the following arrays and print them showing no more than three digits to the right of the decimal point:

   (a) An array of 24 elements all equal to e, the base of the natural logarithm.

   (b) An array `b1` in 10-degree increments of all the angles in degrees from 0 to 360 degrees inclusive using the NumPy `arange` function. Then make the same array using the NumPy `linspace` function and name it `b2`. Print `b1` and `b2`.

   (c) An array `c` in 10-degree increments of all the angles in radians from 0 to 360 degrees inclusive. Verify your answers by showing that `c - b * np.pi / 180` gives an array of zeros (or nearly zeros), where `b` and `c` are the arrays you created in parts (b) and (c).

   (d) An array from 12 to 17, not including 17, in 0.2 increments; an array from 12 to 17, including 17, in 0.2 increments.

6. The position of a ball at time t dropped with zero initial velocity from a height h_0 is given by

$$y = h_0 - \frac{1}{2}gt^2$$

where $g = 9.8$ m/s^2. Suppose $h_0 = 10$ m. Find the sequence of times when the ball passes each half meter assuming the ball is dropped at $t = 0$. Hint: Create a NumPy array for y that goes from 10 to 0 in increments of -0.5 using the `arange` function. Solving the above equation for t, show that

$$t = \sqrt{\frac{2(h_0 - y)}{g}}.$$

Using this equation and the array you created, find the sequence of times when the ball passes each half meter. Save your code as a Python script and have it print the position y and time t arrays. It should yield the following results:

```
y
[10.   9.5  9.   8.5  8.   7.5  7.   6.5  6.   5.5  5.
  4.5  4.   3.5  3.   2.5  2.   1.5  1.   0.5]
t
[0.    0.319 0.452 0.553 0.639 0.714 0.782 0.845 0.904
 0.958 1.01  1.059 1.107 1.152 1.195 1.237 1.278 1.317
 1.355 1.392]
```

7. Recalling that the average velocity over an interval Δt is defined as $\bar{v} = \Delta y / \Delta t$, print the average velocity for each time interval in the previous problem using NumPy arrays. Also, print the array times, midway between the times for the ball positions, that go with each velocity. Keep in mind that the number of time intervals is one less than the number of times. Hint: What are the arrays `y[1:20]` and `y[0:19]`? What does the array `y[1:20]-y[0:19]` represent? (Try printing out the two arrays from the IPython shell to check that you get what you think you should get.) Find the average velocities using this last array and a similar one involving time. Can you think of a more elegant way of representing `y[1:20]-y[0:19]` that does not make explicit reference to the number of elements in the y array—one that would work for any length array?

   You should get the following answer for the arrays of velocities and times:

```
v =
[ -1.565  -3.779  -4.925   -5.842  -6.63   -7.334  -7.975
  -8.568  -9.123  -9.645  -10.141 -10.614 -11.066 -11.5
 -11.919 -12.323 -12.715 -13.094 -13.464]
t =
[0.16  0.386 0.503 0.596 0.677 0.748 0.814 0.874 0.931
 0.984 1.035 1.083 1.129 1.173 1.216 1.257 1.297 1.336
 1.374]
```

8. Perform the following tasks with NumPy arrays. All of them can be done (elegantly) in 1 to 3 lines.

   (a) Create a NumPy 8 × 8 array of integers with ones on all the edges and zeros everywhere else. Hint: use NumPy's ones function and then use slicing to set the interior elements of the array to zero. Print the result. It should look like this:

```
[[1 1 1 1 1 1 1 1]
 [1 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 1]
 [1 1 1 1 1 1 1 1]]
```

   (b) Create a NumPy 8×8 array of integers with a checkerboard pattern of ones and zeros. It should look like this:

```
[[0 1 0 1 0 1 0 1]
```

```
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]]
```

(c) Given the array `c = np.arange(2, 50, 5)`, use a Boolean mask to make all the numbers not divisible by 3 negative. Print the result. It should look like this:

```
[ -2  -7  12 -17 -22  27 -32 -37  42 -47]
```

(d) Use NumPy methods to find the size, shape, mean, and standard deviation of the arrays you created in parts (a)–(c).

9. Perform the cross and dot products of the following vectors:

(a) Verify that the NumPy function `np.cross` works as expected and gives $\mathbf{e}_x \times \mathbf{e}_y = \mathbf{e}_z$, where $\mathbf{e}_x = (1, 0, 0)$, $\mathbf{e}_y = (0, 1, 0)$, and $\mathbf{e}_y = (0, 0, 1)$. Then, use the NumPy `dot` function to verify that $\mathbf{e}_x \cdot \mathbf{e}_y = 0$.

(b) The magnitude of the cross produce of two vectors is given by $|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}| \sin\theta$, where θ is the angle between the two vectors $\mathbf{a}$ and $\mathbf{b}$. Use this relation to find the angle θ in degrees between $\mathbf{a}$ and $\mathbf{b}$ and print the result. Do not use the `np.abs` function to determine $|\mathbf{a}|$ and $|\mathbf{b}|$ as it takes the absolute value of each element of a NumPy array, which is not what you want. Hint: what does `(a * a).sum()` calculate?

(c) The magnitude of the dot product of two vectors is given by $|\mathbf{a} \cdot \mathbf{b}| = |\mathbf{a}||\mathbf{b}| \cos\theta$. Use this relation to find the angle θ in degrees between $\mathbf{a}$ and $\mathbf{b}$ and print the result. If you have done everything correctly, you should get the same angle that you calculated in part (b)

CHAPTER 5

Input and Output

*In this chapter, you learn how to **input (read) data** into a Python program, either from the keyboard or a computer file. You also learn how to **output (write) data** to a computer screen or a computer file.*

A good relationship depends on good communication. In this chapter, you learn how to communicate with Python. Of course, communicating is a two-way street: input and output. Generally, when you have Python perform some task, you need to feed it information—input. When finished with that task, it reports to you the results of its calculations—output.

There are two venues for input that concern us: the computer keyboard and the input data file. Similarly, there are two venues for output: the computer screen and the data file. The chapter starts with input from the keyboard and output to the computer screen. Then, data file input and output—or "io"—is discussed.

5.1 KEYBOARD INPUT

Many computer programs need input from the user. In Section 2.5.2, the program `my_trip.py` required the distance traveled as an input to determine the trip's duration and the cost of charging the battery. As you might like to reuse this same script several times to determine the cost of different trips, it would be helpful if the program requested that input when it was run from the IPython shell.

Python has a function called `input` for getting input from the user and assigning to it a variable name. It has the form

```
strname = input("prompt to user")
```

DOI: 10.1201/9781032673950-5

When executed, the `input` function prints the text in the quotes to the computer screen and waits for input from the user. The user types a string of characters and presses the **<return>** key. The `input` function then assigns that string to the variable name, `strname` in this case. Let's try it out with this snippet of code in the IPython shell.

```
In[1]: distance = input("Input trip distance (miles): ")
Input trip distance (miles):
```

Python prints out the string argument of the `input` function and waits for a response from you. Go ahead and type `250` for "250 miles" and press **<return>**. Now, type the variable name `distance` to see its value.

```
In[2]: distance
Out[2]: '250'
```

The value of the `distance` is `250` as expected, but it is a string, as you can see, because 250 is enclosed in quotes. Because you want to use `250` as a number and not as a string, you need to convert it from a string to a number. You can do that with the `eval` function by writing

```
In[3]: distance = eval(distance)

In[4]: distance
Out[4]: 250
```

The `eval` function converts *distance* to an integer. This is fine, but you might prefer that `distance` to be a float instead of an integer, as you will use it to perform floating point arithmetic. There are two ways to do this. You could assume the user is very smart and will type "`250.`" instead of "`250`", which will cause `distance` to be a float when `eval` does the conversion. The number 250 is dynamically typed to be a float or an integer, depending on whether or not the user uses a decimal point. A better solution is to use the function `float` instead of `eval`, which ensures that `distance` is a floating point variable. Thus, the code might look like this (including the user response):

```
In[5]: distance = input("Input distance of trip (miles): ")
Input distance of trip (miles): 250

In[6]: distance
Out[6]: '250'

In[7]: distance = float(distance)

In[8]: distance
Out[8]: 250.0
```

Let's incorporate these ideas into the code introduced as our first scripting example in Section 2.5.2

Code: my_trip_io.py

```
1  """Calculates time, electrical energy used, and cost of electricity
2  for a trip in an electric vehicle"""
3
4  distance = input("Input trip distance (miles): ")
5  distance = float(distance)
6
7  mpk = 3.9             # [miles/kilowatt-h] car mileage
8  speed = 60.           # [miles/h] average speed
9  cost_per_kWh = 0.22   # [$/kW-h] price of electricity
10
11 time = distance / speed        # [hours]
12 energy = distance / mpk        # [kW-h]
13 cost = energy * cost_per_kWh   # [$]
```

We can combine lines 4 and 5 into a single line, which is a little more efficient:

```
distance = float(input("Input trip distance (miles): "))
```

Whether you use `float` or `int` or `eval` depends on whether you want a float, an integer, or a dynamically typed variable. In this program, it doesn't matter very much, but in general, it's good practice to explicitly cast the variable in the type you would like it to have. Here, `distance` is used as a float, so it's best to cast it as such, as done in the example above.

Now, you run the program and then type `time`, `energy`, and `cost` to view the results of the calculations done by the program.

5.2 SCREEN OUTPUT

It would be much more convenient if the program in the previous section wrote its output to the computer screen instead of requiring the user to type `time`, `energy`, and `cost` to view the results. This can be accomplished using Python's `print` function. For example, simply including the statement `print(time, energy, cost)` after line 13, running the program would give the following result:

```
In[1]: run my_trip_io.py
What is the distance of your trip (miles)? 250
4.166666666666667 64.1025641025641 14.102564102564102
```

The program prints out the results as a tuple of time (in hours), energy used (in kilowatt-hours), and cost (in dollars). Of course, the program doesn't give the user a clue as to which quantity is which. The user has to know.

5.2.1 Formatting Output with `str.format()`

We can make the output of the above example considerably more informative and user-friendly. This program demonstrates how to do this.

Code: my_trip_nice_io.py

```python
1  """Calculates time, electrical energy used, and cost of electricity
2  for a trip in an electric vehicle"""
3  
4  distance = float(input("Input trip distance (miles): "))
5  
6  mpk = 3.9              # [miles/kilowatt-h] car mileage
7  speed = 60.            # [miles/h] average speed
8  cost_per_kWh = 0.22    # [$/kW-h] price of electricity
9  
10 time = distance / speed        # [hours]
11 energy = distance / mpk        # [kW-h]
12 cost = energy * cost_per_kWh   # [$]
13 
14 print("\nDuration of trip = {0:0.1f} hours".format(time))
15 print("Electricity used = {0:0.1f} kW-h (@ {1:0.2f} miles/kW-h)"
16       .format(energy, mpk))
17 print("Cost of electricity = ${0:0.2f} (@ ${1:0.2f}/kW-h)"
18       .format(cost, cost_per_kWh))
```

Running this program, with the distance provided by the user, gives

```
In[2]: run my_trip_nice_io.py

Input trip distance (miles): 250

Duration of trip = 4.2 hours
Electricity used = 64.1 kW-h (@ 3.90 miles/kW-h)
Cost of electricity = $14.10 (@ $0.22/kW-h)
```

Now, the output is presented in a way that is immediately understandable to the user. Moreover, the numerical output is formatted with an appropriate number of digits to the right of the decimal point. For good measure, we also included the assumed mileage ($0.22/kW-h) and the cost of the electricity. All of this is controlled by the `str.format()` method of the `print` function.

The argument of the `print` function is a string containing the text to be displayed on the screen, as well as the code contained within curly braces `{ }` that serves as format specifiers. The format specifiers are interpreted by the `format` method appended to the string using the usual dot syntax. The arguments of the format method are the variables that are to be printed:

- The `\n` at the start of the string in the `print` function on line 14 is the newline character. It creates a blank line before the output is printed.
- The positions of the curly braces specify where the variables in the `format` method at the end of the statement are printed.
- The format string inside the curly braces specifies how each variable in the `format` method is printed.

- The number before the colon in the format string specifies which variable in the list in the `format` function is printed. Remember, Python is zero-indexed, so 0 means the first variable is printed, 1 means the second variable, *etc*.

- The zero after the colon specifies the *minimum* number of spaces reserved for printing out the variable in the format function. A zero means that only as many spaces as needed will be used.

- The letter `f` specifies that a *number* is to be printed with a *fixed* number of digits. If the `f` format specifier is replaced with `e`, then the number is printed out in exponential format (scientific notation).

- The number after the period specifies the number of digits to the right of the decimal point that will be printed: 1 for `time` and `electricity` and `2` for `cost`.

In addition to `f` and `e` format types, two more are commonly used: `d` for integers (digits) and s for strings. There are, in fact, many more formatting possibilities. Python has a whole *Format Specification Mini-Language* that you can look up in the online Python documentation. It's very flexible but arcane. You might find looking at the "Format examples" in the online Python documentation helpful.

Finally, note that the code starting on lines 15 and 17 is each split into two lines. We do this so the lines fit on the page without running off the edge. Python allows you to break up code inside parentheses to improve readability. More information about line continuation in Python and other formatting guidelines can be found here: http://www.python.org/dev/peps/pep-0008/.

The program below illustrates most of the formatting you will need for writing a few variables, be they strings, integers, or floats, to screen or to data files (discussed in the next section).

Code: print_format_examples.py

```
1  string1 = 'How'
2  string2 = 'are you my friend?'
3  int1 = 34
4  int2 = 942885
5  float1 = -3.0
6  float2 = 3.141592653589793e-14
7  print(string1)
8  print(string1 + ' ' + string2)
9  print('A. {} {}'.format(string1, string2))
10 print('B. {0:s} {1:s}'.format(string1, string2))
11 print('C. {0:s} {0:s} {1:s} - {0:s} {1:s}'.format(string1, string2))
12 # Next line reserves 10 & 5 spaces, respectively, for 2 strings
```

```
13 print('D. {0:10s}{1:5s}'.format(string1, string2))
14 print(' **')
15 print(int1, int2)
16 print('E. {0:d} {1:d}'.format(int1, int2))
17 print('F. {0:8d} {1:10d}'.format(int1, int2))
18 print(' ***')
19 print('G. {0:0.3f}'.format(float1))  # 3 decimal places
20 print('H. {0:6.3f}'.format(float1))  # 6 spaces, 3 decimals
21 print('I. {0:8.3f}'.format(float1))  # 8 spaces, 3 decimals
22 print(2 * 'J.  {0:8.3f}   '.format(float1))
23 print(' ****')
24 print('K. {0:0.3e}'.format(float2))
25 print('L. {0:12.3e}'.format(float2))  # 12 spaces, 3 decimals
26 print('M. {0:12.3f}'.format(float2))  # 12 spaces, 3 decimals
27 print(' *****')
28 print('N. 12345678901234567890')
29 print('O. {0:s}--{1:8d},{2:10.3e}'.format(string2, int1, float2))
```

Here is the output:

```
How
How are you my friend?
A. How are you my friend?
B. How are you my friend?
C. How How are you my friend? - How are you my friend?
D. How       are you my friend?
**
34 942885
E. 34 942885
F.       34     942885
***
G. -3.000
H. -3.000
I.   -3.000
J.    -3.000   J.    -3.000
****
K. 3.142e-14
L.    3.142e-14
M.        0.000
*****
N. 12345678901234567890
O. are you my friend?--      34, 3.142e-14
```

Successive empty brackets `{}`, like those in line 9, will print in the order the variables appear inside the `format()` method using their default format. Starting with line 9, the number to the left of the colon inside the curly brackets specifies which of the variables, numbered starting with 0, in the `format` method is printed. The characters that appear to the right of the colon are the format specifiers with the following correspondences: `s`–string, `d`–integer, `f`–fixed floating point number, `e`–exponential floating point number. The format specifiers `6.3f` and `8.3f` in lines 20 and 21 tell the `print` statement to reserve at least 6 and 8 total spaces, respectively, with three decimal places for the output of a floating point number. Studying the output of the other lines will help you understand how formatting works.

5.2.2 Formatting with f-strings

The `print` function in Python is quite versatile. For example, you can print multiple strings and variables simply by separating them by commas in a `print` function.

```
In[3]: a, b, c = "Leon", 47, "March 25"

In[4]: print(a, "turns", b, "on", c + "th", "this year.")
Leon turns 47 on March 25th this year
```

The `print` function automatically adds a trailing space when printing literals and variables separated by commas. This is why we had to use the addition operator to add “th” to “March 25” without introducing a space.

You can also use Python’s *formatted string literals* feature—*f-strings*—to produce the same results a bit more efficiently.

```
In[5]: print(f"{a} turns {b} on {c}th this year.")
Leon turns 47 on March 25th this year.
```

The variables to be printed are once again enclosed in curly braces. You can also include format strings to control the output.

```
In[6]: d, t = 149.32, 2.77

In[7]: print(f"They drove {d:0.1f} miles in {t:0.1f} hours")
They drove 149.3 miles in 2.8 hours

In[8]: print(f"Their average speed was {d/t:0.1f} mph")
Their average speed was 53.9 mph
```

Formatting with f-string literals is done the same way as it’s done using the explicit `format` method. But instead of using numbers 0, 1, ..., within the curly braces `{ }` to index the variables that are the arguments of the `format` method, the variables themselves are used.

5.2.3 Printing Arrays

Formatting NumPy arrays for printing requires another approach. As an example, let’s create an array and then format it in various ways. From the IPython terminal:

```
In[9]: a = np.linspace(3, 19, 7)
In[10]: print(a)
```

```
[  3.     5.6667   8.3333  11.     13.6667  16.3333  19.   ]
```

Simply using the `print` function does print out the array, but perhaps not in the format you desire. To control the output format, you use the NumPy function `set_printoptions`. For example, suppose you want to see no more than two digits to the right of the decimal point. Then, you simply write.

```
 In[11]: np.set_printoptions(precision=2)
 In[12]: print(a)
[ 3.  5.67  8.33  11.  13.67  16.33  19. ]
```

The default setting is to print only as many digits as are needed to represent a number uniquely, so trailing zeros are omitted. If you wish to have all the numbers displayed with the same number of digits, then you set the `floatmode` optional argument to `"fixed"`, as illustrated here:

```
 In[13]: np.set_printoptions(precision=2, floatmode="fixed")

 In[14]: a
Out[14]: array([ 3.00,  5.67,  8.33, 11.00, 13.67, 16.33,
19.00])
```

Suppose you want to use scientific notation. The method for doing it is somewhat arcane, using something called a `lambda` function. For now, you don't need to understand how it works to use it. Just follow the examples shown below, which illustrate several different output formats using the `print` function with NumPy arrays.

```
 In[15]: np.set_printoptions(
    ...: formatter={'float': lambda x: format(x, '5.1e')})

 In[16]: print(a)
[3.0e+00 5.7e+00 8.3e+00 1.1e+01 1.4e+01 1.6e+01 1.9e+01]
```

You use the `formatter` keyword argument to specify the output format. The first entry to the right of the curly bracket is a string that can be `'float'`, as it is above, or `'int'`, or `'str'`, or several other data types that you can look up in the online NumPy documentation. The only other thing you should change is the format specifier string. In the above example, it is `'5.1e'`, specifying that Python should allocate at least five spaces, with one digit to the right of the decimal point in scientific (exponential) notation. For fixed-width floats with three digits to the right of the decimal point, use the `f` in place of the `e` format specifier, as follows:

```
 In[17]: np.set_printoptions(
    ...: formatter={'float': lambda x: format(x, '6.3f')})
 In[18]: print(a)
[ 3.000 5.667 8.333 11.000 13.667 16.333 19.000]
```

To return to the default format, type the following:

```
In[19]: np.set_printoptions(precision=8)
In[20]: print(a)
[ 3.  5.66666667 8.33333333 11.  13.66666667 16.33333333 19. ]
```

5.3 FILE INPUT

5.3.1 Reading Data from a Text File

Often, you would like to analyze data stored in a text file. Consider, for example, the data file below for an experiment measuring the free fall of a mass.

Data: mydata.txt

```
Data for falling mass experiment
Date: 16-Aug-2021
Data taken by Isabella and Martin

data point time (sec) height (mm) uncertainty (mm)
     0          0.0        180          3.5
     1          0.5        182          4.5
     2          1.0        178          4.0
     3          1.5        165          5.5
     4          2.0        160          2.5
     5          2.5        148          3.0
     6          3.0        136          2.5
     7          3.5        120          3.0
     8          4.0         99          4.0
     9          4.5         83          2.5
    10          5.0         55          3.6
    11          5.5         35          1.75
    12          6.0          5          0.75
```

You can read these data into a Python program, associating the data in each column with an appropriately named array. While there are many ways to do this in Python, the simplest is to use the NumPy `loadtxt` function. Suppose that the name of the text file is `mydata.txt`. Then, you can read the data into four different arrays with the following statement:

```
In[1]: dataPt, time, height, error = np.loadtxt(
  ...: "mydata.txt", skiprows=5 , unpack=True)
```

In this case, the `loadtxt` function takes three arguments: the first is a string that is the name of the file to be read, the second tells `loadtxt` to skip the first five lines at the top of file, sometimes called the *header*, and the third tells `loadtxt` to output the data (*unpack* the data) so that it can be directly read into arrays. `loadtxt` reads however many columns of data are present in the text file to the array names listed to the left of the "=" sign. The names labeling the columns in the text file are not used, but you are free to choose the same or similar names, of course, as long as they are legal array names.

For the above `loadtxt` call to work, the file `mydata.txt` should be in the same directory as the calling program, which should be the current working directory of the IPython shell. Otherwise, you need to specify the full directory path with the file name `mydata.txt`.

It is critically important that the data file be a *text* file. It cannot be an MS Word file, for example, an Excel file, or anything other than a plain text file. Such files can be created by text editor programs like **Notepad++** (for PCs), or **BBEdit** (for Macs), or **Gedit** (for Linux). They can also be created by MS Word and Excel provided you explicitly save the files as text files. **Beware**: You should exit any text file you make and save it with a program that allows you to save the text file using **UNIX**-type formatting, which uses a *line feed* (LF) to end a line. Some programs, like MS Word under Windows, may include a carriage return (CR) character, which can confuse `loadtxt`. Note that we give the file name a `.txt` *extension*, which indicates to most operating systems that this is a *text* file, as opposed to an Excel file, for example, which might have a `.xlsx` or `.xls` extension.

If you don't want to read in all the columns of data, you can specify which columns to read using the `usecols` keyword. For example, the call

```
In[2]: time, height = np.loadtxt("mydata.txt", skiprows=5,
usecols=(1, 2), unpack=True)
```

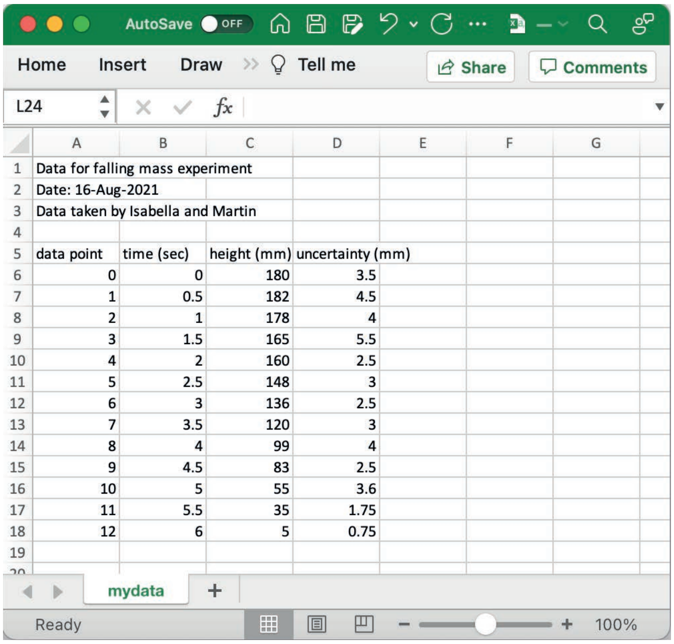
reads in only columns 1 and 2; columns 0 and 3 are skipped. Thus, only two array names are included to the left of the "=" sign, corresponding to the two columns that are read. Writing `usecols = (0,2,3)` would skip column 1 and read only the data in columns 0, 2, and 3. In this case, three array names must be provided on the left-hand side of the "=" sign.

One convenient feature of the `loadtxt` function is that it recognizes any *white space* as a column separator: spaces, tabs, *etc.*

Finally, remember that `loadtxt` is a NumPy function. So, if you are using it in a Python module, you must include an "`import numpy as np`" statement before calling "`np.loadtxt`".

5.3.2 Reading Data from an Excel File: CSV Files

Sometimes, you have data stored in a spreadsheet program like Excel that you would like to read into a Python program. The *Excel data sheet* shown in Figure 5.1 contains the same data set encountered above in a text file. While there are several different approaches one can use to read such files, the simplest and most robust is to save the spreadsheet as a CSV ("comma-separated value")



	A	B	C	D	E	F	G
1	Data for falling mass experiment						
2	Date: 16-Aug-2021						
3	Data taken by Isabella and Martin						
4							
5	data point	time (sec)	height (mm)	uncertainty (mm)			
6	0	0	180	3.5			
7	1	0.5	182	4.5			
8	2	1	178	4			
9	3	1.5	165	5.5			
10	4	2	160	2.5			
11	5	2.5	148	3			
12	6	3	136	2.5			
13	7	3.5	120	3			
14	8	4	99	4			
15	9	4.5	83	2.5			
16	10	5	55	3.6			
17	11	5.5	35	1.75			
18	12	6	5	0.75			
19							

Figure 5.1 Excel data sheet.

file, a format that all standard spreadsheet programs can create and read. For example, if your Excel spreadsheet was called `mydata.xlsx`, the CSV file saved using Excel's Save As command would be `mydata.csv` by default. It would look like this:

Data: mydata.csv

```
Data for falling mass experiment,,,
Date: 16-Aug-2021,,,
Data taken by Isabella and Martin,,,
,,,
data point,time (sec),height (mm),uncertainty (mm)
0,0,180,3.5
1,0.5,182,4.5
2,1,178,4
3,1.5,165,5.5
4,2,160,2.5
5,2.5,148,3
6,3,136,2.5
7,3.5,120,3
8,4,99,4
9,4.5,83,2.5
10,5,55,3.6
11,5.5,35,1.75
12,6,5,0.75
```

As its name suggests, the CSV file is simply a text file with the data formerly in spreadsheet columns now separated by commas. You can read the data in this file into a Python program using the `loadtxt` NumPy function once again. Here is the code

```
In[3]: dataPt, time, height, error = np.loadtxt("mydata.csv",
skiprows=5 , unpack=True, delimiter=',')
```

The form of the function is the same as before, except we added the argument `delimiter=','` that tells `loadtxt` that the columns are separated by commas instead of white space (spaces or tabs), which is the default. Once again, the `skiprows` argument is set to skip the header at the beginning of the file and to start reading at the first row of data. The data are output to the arrays to the right of the assignment operator = exactly as in the previous example.

5.4 FILE OUTPUT

5.4.1 Writing Data to a Text File

There are many ways to write data to a data file in Python. We will stick to one very simple one suitable for writing data files in text format. It uses the NumPy `savetxt` routine, the counterpart of the `loadtxt` routine introduced in the previous section. The general form of the routine is

```
savetxt(filename, array, fmt="%0.18e", delimiter=" ",
newline=" \n", header="", footer="", comments="#")
```

We illustrate `savetext` below with a script that first creates four arrays by reading in the data file `mydata.txt`, as discussed in the previous section, and then writes that same data set to another file `mydataout.txt`.

Code: read_write_mydata.py

```
import numpy as np
dataPt, time, height, error = np.loadtxt("mydata.txt", skiprows=5,
                                         unpack=True)
np.savetxt("mydataout.txt",
           list(zip(dataPt, time, height, error)),
           fmt="%12d %10.2f %12.0f %12.1f")
```

The first argument of `savetxt` is a string, the name of the data file to be created. Here, we have chosen the name `mydataout.txt`, inserted with quotes, which designates it as a literal string. Beware, if there is already a file of that name on your computer, it will be overwritten—the old file will be destroyed and a new one will be created.

The second argument is the data array to be written to the data file. Because we want to write not one but four data arrays to the file, we have to package the four data arrays as one, which we do using the `zip` function. This Python function combines the four arrays and returns a list of tuples, where the i^{th} tuple contains the i^{th} element from each of the arrays (or lists, or tuples) listed as its arguments. Since there are four arrays, each row will be a tuple with four entries, producing a table with four columns. In fact, the `zip` function is just a set of instructions to produce each tuple one after another; the `list` function is needed to construct the entire list of tuples.[1] Note that the first two arguments, the `filename` and data `array`, are regular arguments and thus must appear as the first and second arguments in the correct order. The remaining arguments are all *keyword arguments*, meaning they are optional and can appear in any order, provided you use the keyword.

The next argument is a format string that determines how the elements of the array are displayed in the data file. The argument is optional and, if left out, is the format `0.18e`, which displays numbers as 18-digit floats in exponential (scientific) notation. Here, we choose a different format, `12.1f`, a float displayed with one digit to the right of the decimal point and a minimum width of 12. By choosing 12, which is more digits than any of the numbers in the various arrays, we ensure that all the columns will have the same width. It also ensures that the decimal points in a column of numbers are aligned. This is evident in the data file below, `mydataout.txt`, produced by the above script.

Data: mydataout.txt

```
           0        0.00           180           3.5
           1        0.50           182           4.5
           2        1.00           178           4.0
           3        1.50           165           5.5
           4        2.00           160           2.5
           5        2.50           148           3.0
           6        3.00           136           2.5
           7        3.50           120           3.0
           8        4.00            99           4.0
           9        4.50            83           2.5
          10        5.00            55           3.6
          11        5.50            35           1.8
          12        6.00             5           0.8
```

If you want the different columns to be formatted differently, include a separate format statement for each column separated by spaces, for example, `fmt="%12d %10.2f %12.0f %12.1f"`.

We omitted the optional `delimiter` keyword argument, which leaves the delimiter as the default space. We also omitted the optional `header` keyword argument, a string variable allowing you to write header text above the data. For example, you might want to label the data columns and also include the

[1]Technically, the `zip` function is an *iterator*. Iterators are discussed in Section 6.2.

information that was in the header of the original data file. To do so, you need to create a string with the information you want to include and then use the `header` keyword argument. The code below illustrates how to do this.

Code: read_write_mydata_header.py

```python
import numpy as np

dataPt, time, height, error = np.loadtxt("MyData.txt", skiprows=5,
                                         unpack=True)

info = 'Data for falling mass experiment'
info += '\nDate: 16-Aug-2021'
info += '\nData taken by Lauren and John'
info += '\n\n data point time (sec) height (mm) '
info += 'uncertainty (mm)'

np.savetxt('read_write_mydata_header.txt',
           list(zip(dataPt, time, height, error)),
           header=info, fmt="%12.1f")
```

Now, the data file produced has a header preceding the data. Notice that the header rows all start with a # comment character, the default setting for the `savetxt` function. This can be changed using the keyword argument `comments`. You can find more information about `savetxt` using the IPython `help` function or from the online NumPy documentation.

Data: read_write_mydata_header.txt

```
# Data for falling mass experiment
# Date: 16-Aug-2021
# Data taken by Lauren and John
#
#  data point time (sec) height (mm) uncertainty (mm)
         0.0          0.0        180.0          3.5
         1.0          0.5        182.0          4.5
         2.0          1.0        178.0          4.0
         3.0          1.5        165.0          5.5
         4.0          2.0        160.0          2.5
         5.0          2.5        148.0          3.0
         6.0          3.0        136.0          2.5
         7.0          3.5        120.0          3.0
         8.0          4.0         99.0          4.0
         9.0          4.5         83.0          2.5
        10.0          5.0         55.0          3.6
        11.0          5.5         35.0          1.8
        12.0          6.0          5.0          0.8
```

5.4.2 Writing Data to a CSV File

To produce a CSV file, specify a comma as the delimiter. You might use the `0.1f` format specifier, which leaves no extra spaces between the comma data separators, as the file is to be read by a spreadsheet program, which will determine how the numbers are displayed. The code, which could be substituted for the `savetxt` line in the above code, reads

```
np.savetxt('mydataout.csv',
list(zip(dataPt, time, height, error)),
fmt="%0.1f", delimiter=",")
```

and produces the following data file:

Data: mydataout.csv

```
0.0,0.0,180.0,3.5
1.0,0.5,182.0,4.5
2.0,1.0,178.0,4.0
3.0,1.5,165.0,5.5
4.0,2.0,160.0,2.5
5.0,2.5,148.0,3.0
6.0,3.0,136.0,2.5
7.0,3.5,120.0,3.0
8.0,4.0,99.0,4.0
9.0,4.5,83.0,2.5
10.0,5.0,55.0,3.6
11.0,5.5,35.0,1.8
12.0,6.0,5.0,0.8
```

With a `csv` extension, a spreadsheet program like Excel can directly read this data file.

5.5 EXERCISES

1. Write a Python program that calculates how much money you can spend each day for lunch for the rest of the month based on today's date and how much money you currently have in your lunch account. The program should ask you: (1) how much money you have in your account, (2) what today's date is, and (3) how many days there are in the month. The program should return your daily allowance. The results of running your program should look like this:

```
How much money (in dollars) in your lunch account? 319
What day of the month is today? 21
How many days in this month? 30
You can spend $31.90 each day for the rest of the month.
```

Extra: Create a dictionary (see Section 4.3) that stores the number of days in each month (forget about leap years) and have your program ask what month it is rather than the number of days in the month.

2. Write a script that creates the following three NumPy arrays:

```
a = array([1, 3, 5, 7])
b = array([8, 7, 5, 4])
c = array([0, 9,-6,-8])
```

Then use the `zip` function to create a list `d` defined as

```
d = list(zip(a, b, c))
```

   (a) Print out `d`. What type of object is `d`? What type of object is `d[0]`?
   (b) One of the elements of `d` is `-8`. Show how to address and print out just that element of `d`.
   (c) From `d`, create a NumPy array and call that array `g`. Print `g`.
   (d) One of the elements of `g` is `-8`. Show how to address and print out just that element of `g`.
   (e) Have your program print out `g[1]`. What type of object is `g[1]`?

3. Create the following data file and then write a Python script to read it into three NumPy arrays with the variable names `f, a, da` for the frequency, amplitude, and amplitude error.

```
Date: 2013-09-21
Data taken by Liam and Selena
frequency (Hz) amplitude (mm) amp error (mm)
 0.7500         13.52          0.32
 1.7885         12.11          0.92
 2.8269         14.27          0.73
 3.8654         16.60          2.06
 4.9038         22.91          1.75
 5.9423         35.28          0.91
 6.9808         60.99          0.99
 8.0192         33.38          0.36
 9.0577         17.78          2.32
10.0962         10.99          0.21
11.1346          7.47          0.48
12.1731          6.72          0.51
13.2115          4.40          0.58
14.2500          4.07          0.63
```

Show that you have correctly read in the data by having your script print out to your computer screen the three arrays. Format the printing so that it produces output like this:

```
f =
[ 0.75    1.7885  2.8269  3.8654  4.9038  5.9423  6.9808
  8.0192  9.0577 10.0962 11.1346 12.1731 13.2115 14.25  ]
a =
[13.52 12.11 14.27 16.6  22.91 35.28 60.99 33.38 17.78
 10.99  7.47  6.72  4.4   4.07]
da =
[0.32 0.92 0.73 2.06 1.75 0.91 0.99 0.36 2.32 0.21 0.48
 0.51 0.58 0.63]
```

Note that the array `f` is displayed with four digits to the right of the decimal point while the arrays `a` and `da` are displayed with only two. The columns of the displayed arrays need not line up as they do above.

4. Write a script to read the data from the previous problem into three NumPy arrays with the variable names `f`, `a`, `da` for the frequency, amplitude, and amplitude error and then, in the same script,

   (a) Write the data to a data file, including the header, with the data displayed in three columns, just as it's displayed in the problem above. It's ok if the header lines begin with the # comment character. Give your data file the name `my_data_out.txt`.

   (b) Write the data to a csv data file, without the header, with the data displayed in three columns. Use a single format specifier and set it to "`%0.16e`". If you can access a spreadsheet program (like MS Excel), try opening the file you created with your Python script and verify that the arrays are displayed in three columns. Give your data file the name `my_data_out.csv`, making sure that your file has the extension `.csv`!

CHAPTER 6

Conditionals and Loops

In this chapter, you learn how to ***control the flow of a program***. *In particular, you learn how to program a computer to make decisions based on information on different conditions it encounters as it processes data or information. You learn exception handling, a method to catch errors and gracefully process them during the execution of a program.*
You also learn how to make a computer do repetitive tasks using loops.

All the programs we have written so far have run sequentially through a series of statements. Programs become much more powerful when they can `make decisions` and *branch* based on different inputs or results they encounter. They gain even more power when they can iterate to *perform repetitive tasks*. That power is compounded when a program puts these two capabilities together.

We can illustrate how these ideas work by considering a problem that my sixth-grade teacher, Mr. Marcus, posed to our class many years ago. He said, "Suppose a farmer has some chickens and pigs. He counts a total of 34 animals and 86 legs. How many pigs and chickens does the farmer have?"

Now, I didn't know how to solve this problem, so I thought I'd make a guess and see how it worked out. So I guessed 12 pigs and 22 chickens. This gives $(12 \times 4) + (22 \times 2) = 92$ legs. Oops, too many legs, so there must be fewer pigs. Let's try 10 pigs and 24 chickens. This gives $(10 \times 4) + (24 \times 2) = 88$ legs. Closer! Much closer. I was getting excited now. Let's try 9 pigs and 25 chickens. This gives $(9 \times 4) + (25 \times 2) = 86$ legs. Eureka, I found the answer! I raised my hand and told Mr. Marcus and the class my answer.

"Correct," said Mr. Marcus. "How did you get the answer?"

So I told him I made a guess, and when I found that it was wrong, I adjusted my guess until I got the answer. "That's not how you solve the problem,"

DOI: 10.1201/9781032673950-6

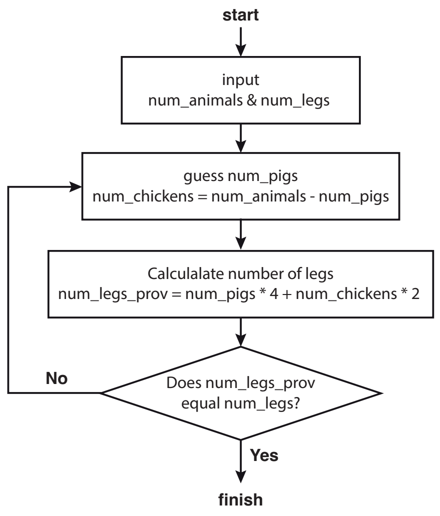

start
input
num_animals & num_legs
guess num_pigs
num_chickens = num_animals - num_pigs
Calculalate number of legs
num_legs_prov = num_pigs * 4 + num_chickens * 2
Does num_legs_prov
equal num_legs?
No
Yes
finish

Figure 6.1 Flowchart for chicken and pigs problem.

Mr. Marcus said, and he went on to show us how to pose the problem using two equations with two unknowns, ...

Ok, so I didn't know algebra and couldn't solve the problem analytically. But my method wasn't so bad. It illustrates an algorithmic approach, an approach we often employ when using computers to solve problems. Moreover, the method I employed involves both decision-making and iteration. A flowchart of the algorithm is shown in Figure 6.1. The rectangular boxes represent a processing step, and the diamond box represents a decision. A decision to stop or continue looking for a solution is made based on the success of the guess: does the calculated provisional number of legs equal the specified number of legs? An iteration occurs when the current guess fails to produce the correct number of legs. Otherwise, the answer has been found, and the process terminates.

In the problem of the chickens and pigs, it took only three iterations to find the correct answer, so a human could easily accomplish it in a short period of time. The math, engineering, and science problems we need to solve are generally more complex and typically require more iterations, many more than any human would like to attempt. That's where computers are helpful. Without complaining, getting bored, or growing tired, they can repetitively perform the same calculations with minor but important variations over and over again. Of course, we need efficient ways of telling the computer to do these repetitive tasks. This is what **loops** were made for. Python has two types of loop structures: `for` loops and `while` loops, which we introduce in this chapter.

Computer programs often need to make decisions as well. In the chicken and pigs problem, the decision was whether to stop or keep going based on a test of whether or not the current guess produced the correct number of legs. This is what **conditionals** are made for: to determine the flow of a program based on a test. Python conditional statements use the commands `if`, `elif`, and `else`, which we also introduce in this chapter.

Conditionals and loops control the flow of a program. They are essential to performing virtually any significant computational task.

6.1 CONDITIONALS

Conditional statements allow a computer program to take different actions based on whether some condition or set of conditions is true or false. In this way, the programmer can control the flow of a program.

6.1.1 `if`, `elif`, and `else` Statements

The `if`, `elif`, and `else` statements are used to define conditional statements in Python. They are used in the following format:

```
if <condition 1>:
    # indented block of code that is executed if <condition 1>
    # is True; the elif and else blocks that follow are
    # skipped if <condition 1> is True
elif <condition 2>:
    # ("else if") indented block of code that is executed if
    # <condition 1> was False and <condition 2> is True; the
    # else block that follows is skipped if <condition 2> is
    # True
else:
    # indented block of code that is executed only if both
    # <condition 1> and <condition 2> were False
```

`<condition 1>` and `<condition 2>` are Boolean expressions that can be either `True` of `False`. We illustrate their use with a few examples.

6.1.1.1 `if-elif-else` Example

Suppose you want to know if the solutions to the quadratic equation

$$ax^2 + bx + c = 0$$

are real, imaginary, or complex for a given set of coefficients a, b, and c. The solution to this quadratic equation is given by the famous formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The answer to whether the solutions are real, imaginary, or complex depends on the value of the discriminant $d = b^2 - 4ac$. The solutions are real if $d \geq 0$, imaginary if $d < 0$ and $b = 0$, and complex if $d < 0$ and $b \neq 0$. The program below implements the above logic in a Python program.

After getting the inputs from the user, the program calculates the value of the discriminant d.

Code: if_elif_else_example.py

```
1  a = float(input("What is the coefficient a? "))
2  b = float(input("What is the coefficient b? "))
3  c = float(input("What is the coefficient c? "))
4  d = b * b - 4. * a * c
5  if d >= 0.0:
6      print("Solutions are real")       # block 1
7  elif b == 0.0:
8      print("Solutions are imaginary")  # block 2
9  else:
10     print("Solutions are complex")    # block 3
11 print("Finished")
```

After d is evaluated, there are three *conditional* statements in the program, in lines 5, 7, and 9. Each conditional statement evaluates the truth value of a Boolean logical expression, ends with a colon, and is followed by an indented block of code. In this example, the indented blocks of code are just one line long, but they could be many lines—as many as you need.

We encounter the first conditional statement in line 5, an `if` statement. The `if` statement tests whether the Boolean expression `d >= 0.0` is `True` or `False`. If the expression is `True`, Python executes the indented block of statements following the `if` statement. Here, there is only one line in the indented block: `print("Solutions are real")`. Once it executes this statement, Python skips the `elif` and `else` blocks and executes the `print("Finished!")` statement.

If the expression `d >= 0.0` in line 5 is `False`, Python skips the indented block directly below the `if` statement and executes the `elif` ("else if") statement. If the expression `b == 0.0` is `True`, it executes the indented block immediately below the `elif` statement, `print("Solutions are imaginary")` and then skips the `else` statement and the indented block below it. It then executes the `print("Finished!")` statement.

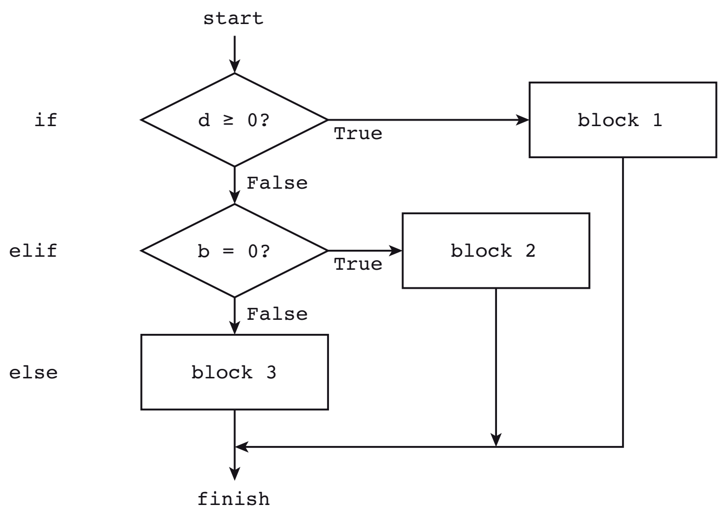


Figure 6.2 Flowchart for `if-elif-else` code.

Finally, if the expression `b == 0.0` is `False`, Python skips to the `else` statement and executes the block immediately below the `else` statement, `print("Solutions are complex")`. Once finished with that indented block, it then executes the statement `print("Finished!")`.

Each time a `False` result is obtained in an `if` or `elif` statement, Python skips the indented code block associated with that statement and drops down to the next conditional statement, that is, the next `elif` or `else`. A flowchart of the if-elif-else code is shown in Figure 6.2.

At the outset of this problem, we noted that the solutions to the quadratic equation are imaginary only if $b = 0$ and $d < 0$. The `elif b == 0.0` statement on line 7, however, only checks to see if $b = 0$. The reason that the statement doesn't have to check if $d < 0$ is that the `elif` statement is executed only if the condition `d >= 0.0` on line 5 is `False`. Similarly, the final `else` statement on line 9 is executed only if the preceding `if` and `elif` statements are `False`, so it doesn't have to check that $d < 0$ and $b \neq 0$. This illustrates a key feature of the `if`, `elif`, and `else` statements: these statements are executed sequentially until one of the `if` or `elif` statements is found to be `True`. Python reaches an `elif` or `else` statement only if all the preceding `if` and `elif` statements are `False`.

The `if-elif-else` logical structure can accommodate as many `elif` blocks as desired. This allows you to set up logic with more than the three possible outcomes in the example above. When designing the logical structure, you should keep in mind that once Python finds a true condition, it skips all subsequent `elif` and `else` statements in a given `if`, `elif`, and else block, irrespective of their truth values.

Important Note about Indentation in Python

The blocks of code that go with each of the `if`, `elif`, and `else` statements are defined by indenting the code. Indenting blocks of code is a generic feature Python and is used in various contexts to define code functionality. Python is different from many other computer languages, where indentation is optional. Indenting has the benefit, required or not, of making the functionality and structure of the code visually apparent to anyone reading the code. In Python, indenting is an essential part of the syntax and, as such, is strictly enforced. We will encounter similar uses of indentation in Python in several different contexts.

For any of these blocks, the indentation of each line has to be the same for all lines; here we use four spaces, which is the conventional choice. You should use four spaces, too; almost everybody does, and using a different number, while legal, only sews confusion when others read your code. More importantly, when indenting a block of code in Python, *every line must be indented by the same amount using the same characters*. So even if a <**tab**> character indents a line by four spaces, it is not equivalent to four <**space**> characters. Therefore, indenting some lines by four spaces and other lines by a <**tab**> character will produce an error, even when all the lines appear to have the same indentation. For this reason, virtually all Python editors translate pressing the <**tab**> key into four spaces; both Spyder and JupyterLab do. Following this convention of using four spaces and no <**tab**> characters will help ensure that your code functions correctly and is readable by others.

6.1.1.2 `if-else` Example

You will often run into situations where you want the program to execute one of only two possible blocks based on the outcome of an `if` statement. In this case, you omit the `elif` block and simply use an `if-else` structure. The following program testing whether an integer is even or odd provides a simple example.

Code: if_else_example.py

```python
a = int(input("Please input an integer: "))
if a % 2 == 0:
    print(f"{a:0d} is an even number.")
else:
    print(f"{a:0d} is an odd number.")
```

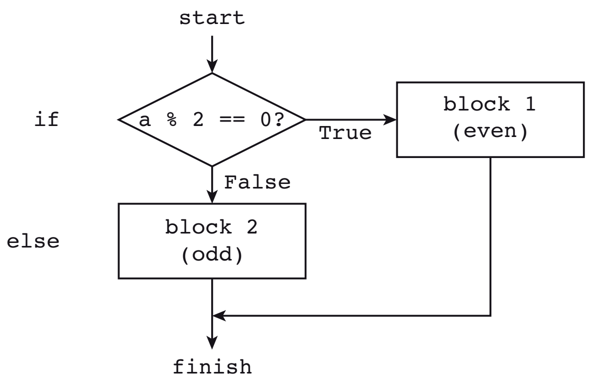


Figure 6.3 Flowchart for `if-else` code.

The flowchart in Figure 6.3 shows the logical structure of an `if-else` structure.

6.1.1.3 *`if` Example*

The simplest logical structure you can make is a simple `if` statement, which executes a block of code if some condition is met but otherwise does nothing. The program below, which takes the absolute value of a number, provides a simple example of such a case.

Code: if_example.py

```python
a = eval(input("Please input a number: "))
if a < 0:
    a = -a
print(f"The absolute value is {a}")
```

When the block of code in an `if` or `elif` statement is only one line long, you can write it on the same line as the `if` or `elif` statement. For example, the above code can be written as follows:

Code: if_example_alt.py

```python
a = eval(input("Please input a number: "))
if a < 0: a = -a
print(f"The absolute value is {a}")
```

This works exactly as the preceding code. Note, however, that if the block of code associated with an `if` or `elif` statement is more than one line long, the entire block of code must be written as indented text below the `if` or `elif` statement.

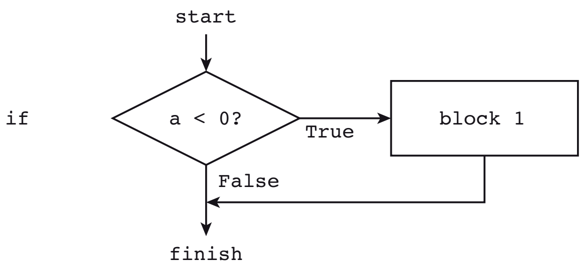


Figure 6.4 Flowchart for `if` code.

The flowchart in Figure 6.4 shows the logical structure of a simple `if` structure.

6.1.2 More about Boolean Variables, Operators, and Expressions

The preceding sections introduced Boolean operators and expressions. Before continuing our discussion of conditionals, we pause for a brief synopsis of Boolean arithmetic.

In addition to the object types we have encountered thus far (`int`, `float`, `complex`), Python has a logical (or Boolean) object type called `bool`. Boolean objects can take on only two values: `True` or `False`.

```
 In[1]: a = True

 In[2]: type(a)
Out[2]: bool

 In[3]: b = False

 In[4]: type(b)
Out[4]: bool
```

As such, Boolean variables aren't terribly interesting. But Boolean expressions are! As we have seen above, they are at the heart of conditional statements.

Most Boolean expressions involve comparisons between two objects using a Boolean operator. The values of Boolean expressions are either `True` or `False` depending on what is being tested, as these examples show.

```
 In[5]: 3 < 5
Out[5]: True

 In[6]: 3 >= 5
Out[6]: False
```

TABLE 6.1 Logical operators in Python.

Operator	Function
compare values of two objects	
`a < b`	`a` is less than `b`
`a <= b`	`a` is less than or equal to `b`
`a > b`	`a` is greater than `b`
`a >= b`	`a` is greater than or equal to `b`
`a == b`	`a` is equal to `b`
`a != b`	`a` is not equal to `b`
compare identities of two objects	
`a is b`	`a` and `b` point to the same object
`a is not b`	`a` and `b` point to different objects
act on Booleans	
`a and b`	both `a` and `b` are true
`a or b`	one or both of `a` and `b` are true
`not a`	reverses the truth value of `a`

```
 In[7]: 3 == 5
Out[7]: False

 In[8]: 3 != 5
Out[8]: True

 In[9]: type(3 != 5)
Out[9]: bool
```

By now, it should be clear that the Boolean operator ==, which tests if the values of two objects are equal, is completely different from the assignment operator =. Don't confuse them!

Table 6.1 summarizes some of the more common and useful Boolean operators. Compound Boolean expressions can be constructed using the `and` and `or` operators listed in Table 6.1. For example, consider the following code that tests to see if a number is divisible by 3 but not by 9:

Code: div_by_3not9.py

```
x = int(input("Input an integer divisible by 3 but not by 9: "))
if x % 3 == 0 and x % 9 != 0:
    print(x, "is divisible by 3 but not by 9")
else:
    print("Not ok")
```

6.1.2.1 *Finding the Maximum from a List of Numbers*

As an application of conditionals, let's consider the problem of finding the maximum from a list of numbers. Let's start with a simple example: finding the maximum in a list of three integers. Consider the following code:

Code: max0.py

```python
a, b, c = input("Input 3 comma-separated integers: ").split(",")
a, b, c = int(a), int(b), int(c)
if a > b and a > c:
    max = a
elif b > a and b > c:
    max = b
else:
    max = c
print(max)
```

The program logic is simple (after making sure you understand how the input statement works!). But does it work? Before jumping to conclusions, let's be sure we've tried different situations. What happens if all three numbers are equal? No problem, the last `else` statement will set `max = c`, which is fine. But if a and b are equal and c is smaller, this routine selects c as the maximum. This problem is easily remedied by changing > to >= in all the comparison statements.

This little vignette highlights the importance of checking your code for different cases, particularly limiting cases where unusual things sometimes occur. "Limiting cases" can mean very large or very small values or equal values; it's often a good idea to check inputs of zero and one, but testing is a bit of an art that requires continual vigilance and skepticism about your logic.

A problem with the code in `max0.py`, even after changing > to >=, is that it contains redundant comparisons; it compares `a` to `b` twice, in line 3 and again in line 5. Interestingly, look at what happens if we remove the redundant piece:

Code: max1.py

```python
1  a, b, c = input("Input 3 comma-separated integers: ").split(",")
2  a, b, c = int(a), int(b), int(c)
3  if a > b and a > c:
4      max = a
5  elif b > c:
6      max = b
7  else:
8      max = c
9  print(max)
```

Now it works again for all cases, as you can check. This code works even better if, once again, we change > to >= because the loop will terminate sooner in some cases (you should figure out which!).

None of the proposed solutions scales very well if there are more than three numbers. For example, when there are four numbers to be checked, then compound conditional statements are needed for the first *two* comparisons. The number of compound comparisons grows as the number of numbers grows, which makes the code increasingly complex. Instead of using compound `if` statements, a better strategy is to process the numbers sequentially. Consider the following code:

Code: max2.py

```
a, b, c = input("Input 3 comma-separated integers: ").split(",")
a, b, c = int(a), int(b), int(c)
max = a
if b > max:
    max = b
if c > max:
    max = c
print(max)
```

The logic used here is exceedingly simple and obviously correct. This program employs a pattern frequently used in computer programming. It begins by setting a provisional answer and then updates the answer if a better one is found. In this algorithem, every number is checked only once, the minimum that's required. Moreover, it is easily scaled to handle cases with many more numbers, which can be coded elegantly and simply using a `for` loop, as we will see in the next section.

6.2 LOOPS

In computer programming, a *loop* is a statement or block of statements that is executed repeatedly. Python has two kinds of loops, a `while` loop and a `for` loop. We first introduce the `while` loop and then the `for` loop.

6.2.1 `while` Loops

The general form of a `while` loop in Python is

```
while <condition>:
<body>
```

where `<condition>` is a Boolean expression that can be either `True` or `False`. The `<body>` is a block of indented code that is repeatedly executed as long as the `<condition>` is `True`. The while loop terminates when `<condition>` becomes `False` or when a `break` statement is issued from within the block. In most cases, this means that somewhere in `<body>`, the truth value of `<condition>` is changed

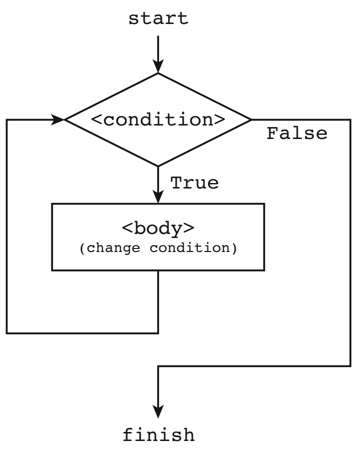


Figure 6.5 Flowchart for `while` loop.

so that it becomes `False` after a finite number of iterations. Figure 6.5 shows the flowchart for a `while` loop.

6.2.1.1 *Chickens and Pigs Problem*

Since this is the logic used in the chickens and pigs problem posed at the beginning of this chapter, let's see if we can codify it in a Python program. Consider the following code:

Code: chickens_pigs.py

```python
legs = int(input("Enter the total number of legs: "))
animals = int(input("Enter the total number of animals: "))

pigs = legs // 4  # maximum possible number of pigs
chickens = animals - pigs

while legs != 4 * pigs + 2 * chickens:
    pigs -= 1
    chickens += 1
    if pigs < 0:
        raise ValueError("There is no solution for these inputs.")

print("\nNumber of chickens =", chickens)
print("Number of pigs =", pigs)
print("Number of legs =", 4 * pigs + 2 * chickens)
print("Number of animals =", pigs + chickens)
```

The strategy employed in this program is to search for the correct answer by trying solutions with different numbers of pigs. It starts with the maximum number of pigs consistent with the prescribed number of legs. Then, it calculates the corresponding number of chickens (which will be the minimum number: 0 or 1) needed to give the prescribed number of animals. The `while` loop tests to see if the initial guess is wrong; if it is wrong, then its Boolean expression is `True`, and the code in the indented block is executed. Within the indented block, the number of pigs is decremented by 1, the number of chickens incremented by 1, after which the `while` statement at the beginning of the loop is reexecuted. This process repeats until the Boolean expression in the `while` statement becomes `False`, which means that the algorithm has found the correct number of pigs and chickens. Then, the indented block is skipped, the `while` loop terminates, and the program prints out the results.

Prior to running the `while` loop, we set our initial guesses for the number of pigs and chickens. You will find that this is a pattern frequently used in setting up loops; variables often need to be initialized prior to starting the loop.

One hazard of using `while` loops is that it's possible for the loop to go on forever without terminating: an infinite loop! Indeed, the algorithm used in this routine is subject to this problem, for example, if we start with an odd number of legs. Therefore it's wise to make sure that any `while` loop you use will always terminate. Here we do just that by including an `if` statement within the `while` loop that terminates the loop if the number of pigs is less than zero. This can only happen if all the possible values for the number of pigs have been tried and have failed. If this happens, the program raises an exception, which stops the program and issues an error message. For example, if we input 85 legs, the program fails to find a solution (because none exists) and returns the following message:

```
File "/Users/dp/Documents/PyScripts/chickens_pigs.py",
line 11, in <module>
raise ValueError("There is no solution for these inputs.")

ValueError: There is no solution for these inputs.
```

Whenever you use a while loop, consider the possibility of encountering an infinite loop and mitigate against it. Here, we handled the problem by using Python's `raise ValueError` statement, which stops the execution of the program. You can (and should!) include an explanatory string as an argument of `ValueError`, as done here. This statement is printed just before Python stops execution of the program. Python provides a means to stop the execution of a program if the program encounters an error through the `raise Exception` statement, where "Exception" can be any one of a number of exceptions, such as `ValueError` used here. We will return to the subject of exceptions in Section 6.4 and how to handle them more adroitly than by stopping the execution of the program, as we have done here.

By the way, if you inadvertently execute code that has an infinite loop, you can often terminate the program from the keyboard by typing **<ctrl-C>** a couple of times. If that doesn't work, you may have to terminate and restart Python.

6.2.1.2 Fibonacci Numbers

Suppose you want to calculate all the Fibonacci numbers smaller than 1000. The Fibonacci numbers are determined by starting with the integers 0 and 1. The next number in the sequence is the sum of the previous two. So, starting with 0 and 1, the next Fibonacci number is 0 + 1 = 1, giving the sequence 0, 1, 1. Continuing this process gives 0, 1, 1, 2, 3, 5, 8, ... where each element in the list is the sum of the previous two. Using a `for` loop to calculate the Fibonacci numbers is impractical because one does not know in advance how many Fibonacci numbers there are smaller than 1000. By contrast, a `while` loop is perfect for calculating all the Fibonacci numbers because it keeps calculating Fibonacci numbers until it reaches the desired goal, in this case, 1000. Here is the code using a `while` loop.

Code: fibonacci.py

```
x, y = 0, 1
while x < 1000:
    print(x)
    x, y = y, x + y
```

We have used the multiple assignment feature of Python in this code. Recall, especially for the assignment inside the `while` loop, that all the values on the right are set first (using the current values of `x` and `y`) and then assigned to the variables `x` and `y` on the left.

Note that the `while` loop is controlled by a conditional that involves a comparison rather than an equality. It is generally a safer way to set up the conditional compared to using an equality. Keep that in mind when using `while` loops.

For work done in science and engineering, the `for` loop is generally more useful than the `while` loop. Nevertheless, there are times when a `while` loop is better suited to a task at hand.

6.2.2 `for` Loops

The general form of a `for` loop in Python is

```
for <itervar> in <sequence>:
<body>
```

where `<intervar>` is a dummy variable, `<sequence>` is a sequence such as a list or string or array, and `<body>` is a series of Python commands to be executed repeatedly for each element in the `<sequence>`. The `<body>` is indented from the rest of the text, defining the loop's extent. Let's look at a few examples.

First, let's extend the program `max2.py` that we considered in Section 6.1.1.1 on page 110 so that it can find the maximum value from a list of integers of arbitrary length. Here is the code:

Code: max3.py

```
1  import random
2  # Make a list of k integers randomly chosen from range
3  nums = random.choices(range(-1000, 1000), k=10)
4  print(nums)
5
6  max = nums[0]
7  for n in nums:
8      if n > max:
9          max = n
10 print("The maximum value is", max)
```

In line 3, the program makes a list `nums` that consists of 10 integers randomly selected between -1000 and 1000 using Python's `random` module (which is distinct from NumPy's `random` module described in Section 9.4). The program uses the first integer in the list to set a provisional value `max = nums[0]` for the maximum (line 6).

The `for` loop begins on line 7. In the `for` statement, the variable `n` is the *interation variable*. The first time through the `for` loop, `n` is set equal to the first element in the list `nums`.

The body of the `for` loop, defined by the indented block (here, just two lines), is then executed using this value of `n`.

Once the two lines of the indented block have been executed, the program returns to the `for` loop, which then sets `n` equal to the second element in the list `nums`, and the indented block is executed again.

Every time the indented block runs, the `if` statement checks to see if the current value of `n` is larger than the value of `max`. If it is larger, the value of `max` is updated to the current value of `n`. If not, it does nothing, and the value of `max` remains unchanged.

This process is repeated until `n` reaches the last element in the list and executes the indented block for the last time.

Then, the loop terminates and goes on to the next (unindented) statement, which is to print out the value of `max`, which should be the maximum value in `nums`. Figure 6.6 show the generic flowchart for a `for` loop.

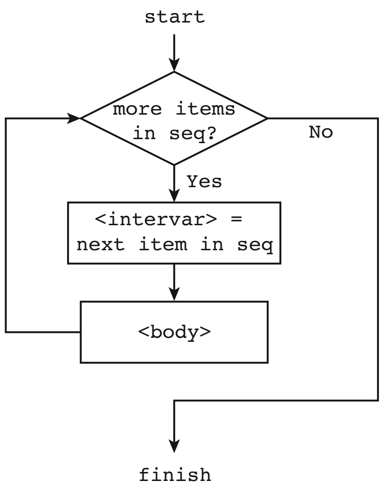


Figure 6.6 Flowchart for a `for`-loop.

Let's run the program:

```
In[1]: %run max3.py
[89, -773, -467, 136, -19, 61, 53, -752, -239, -81]
The maximum value is 136
```

An important feature of this program is that it is scalable. In this example, it finds the maximum value in a sequence of 10 numbers, but it could equally well be 1000 numbers or even 10^6 numbers (provided we supplied it with a sufficiently large pool of numbers to choose from).

The sequence over which the a `for` loop loops can be any kind of list (or list-like object, such as a tuple or an array). Here is a fairly ridiculous loop:

Code: doggy_loop.py

```
for dogname in ["Molly", "Max", "Buster", "Lucy"]:
    print(dogname)
    print(" Arf, arf!")
print("All done.")
```

Running this program, stored in file `doggy_loop.py`, produces the following output:

```
In[2]: run doggy_loop.py
Molly
Arf, arf!
Max
Arf, arf!
Buster
```

```
Arf, arf!
Lucy
Arf, arf!
All done.
```

Let's review the flowchart for a `for` loop (see Figure 6.6). It starts with an implicit conditional asking if there are any more elements in the sequence. If there are, it sets the iteration variable equal to the next element in the sequence and then executes the body—the indented text—using that value of the iteration variable (one of the dog names in the list). It then returns to the beginning to see if there are more elements in the sequence and continues the loop until none remains.

6.2.2.1 Using an Accumulator to Calculate a Sum

Let's look at another application of Python's `for` loop. Suppose you want to calculate the sum of all the odd numbers between 1 and 100. Before writing a program to do this, let's think about how you would do it by hand. You might start by adding 1+3=4. Then, take the result 4 and add the next odd integer, 5, to get 4+5=9; then 9+7=16, then 16+9=25, and so forth. You are doing repeated additions, starting with 1+3, while keeping track of the running sum, until you reach the last number 99.

In developing an algorithm for having the computer sum the series of numbers, we are going to do the same thing: add the numbers one at a time while keeping track of the running sum until we reach the last number. We will keep track of the running sum with the variable `s`, which is called the *accumulator*. Initially, `s = 0`, since we haven't added any numbers yet. Then we add the first number, 1, to `s`, and `s` becomes 1. Then we add the next number, 3, in our sequence of odd numbers to `s`, and `s` becomes 4. We continue doing this repeatedly using a `for` loop while the variable `s` accumulates the running sum until we reach the final number. The code below illustrates how to do this.

Code: odd_sum100.py

```python
s = 0
for i in range(1, 100, 2):
    print(i, end=' ')
    s += i
print(f"\n{s}")
```

The `range` function defines a *sequence* of odd numbers 1, 3, 5, ..., 97, 99. The `for` loop successively adds each number in the list to the running sum until it reaches the last element in the list and the sum is complete. Once the `for` loop finishes, the program exits the loop and prints the final value of `s`, which is the sum of the odd numbers from 1 to 99, is printed out. Line 3 is not needed, of course, and is included only to verify that the odd numbers between 1 and 100 are being summed. The `end=' '` argument causes a space to be printed out between each value of `i` instead of the default new line character `\n`. Copy the above program and run it. You should get an answer of 2500.

The `range` function produces an *iterable sequence*, a set of instructions that yields the next value in a sequence of integers each time it is accessed.

The example above provides a simple example of how a `for` loop works, but it's not the recommended way to perform the task. A faster and more efficient way to sum the odd integers between 1 and 100 is to use a NumPy array:

```
 In[3]: np.arange(1, 100, 2).sum()
Out[3]: 2500
```

We will have more to say about loops and array operations in Section 6.2.4.

6.2.2.2 *Iterating Over Sequences*

You have seen that `for` loops can iterate over elements in a list, such as random numbers, the names of dogs, or over a sequence of numbers produced by the `range` function. In fact, Python `for` loops are extremely versatile and can be used with any object that consists of a sequence of elements. Some of the ways it works might surprise you. For example, suppose we have the string

```
 In[4]: lyrics = "There are places I'll remember all my life"
```

The string `lyrics` is a sequence of characters and thus can be looped over, as illustrated here:

```
 In[5]: for letter in lyrics:
   ...:     print(letter)
   ...:
T
h
e
r
e

a
r
.
.
.
```

Suppose we wanted to print out every third letter of this string. One way to do it would be to set up a counter, as follows:

```
 In[6]: i = 0
```

```
   ...: for letter in lyrics:
   ...:     if i % 3 == 0:
   ...:         print(letter, end=' ')
   ...:     i += 1
   ...:
T r a   a s '   m b   l y i
```

While this approach works fine, Python has a function called `enumerate` that does it for you. It works like this:

```
 In[7]: for i, letter in enumerate(lyrics):
    ...:     if i % 3 == 0:
    ...:         print(letter, end=' ')
    ...:
T r a   a s '   m b   l y i
```

The `enumerate` function takes two inputs, a counter (`i` in this case) and a sequence (the string `lyrics`). In general, any kind of sequence can be used in place of `lyrics`, a list or a NumPy array, for example. The counter `i` starts with a value of zero and `letter`, the iteration variable, is set equal to the first element in the list `lyrics`. The loop is then run with these values. Then, the loop is rerun with `i` incremented by one and `letter` set equal to the next element in the list. This process continues until all the elements of the list have been processed by the loop. Pretty slick. We will find plenty of opportunities to use the `enumerate` function.

6.2.3 Loop Control Statements

There are times when you might need to end a loop before it is finished. Or perhaps you may want to skip one or more of the iterations of a loop. Loop control statements let you do this.

6.2.3.1 The `break` Statement

The `break` statement terminates a loop and moves execution of the program to the first statement after the loop. The program below illustrates its use. It prints out the first ten powers of the variable `a` but stops if the numbers in the sequence become greater than 1,000,000.

Code: breakdemo.py

```
a = 3
for i in range(10):
    x = a ** i
    if x > 1000000:
        break
    print(x, end=' ')
print('finished')
```

If you run the program as written, with `a = 3`, the `if` statement on line 4 is always false, and the following sequence of 10 numbers is printed out: `1 3 9 27 81 243 729 2187 6561 19683 finished`. However, changing line 1 to `a = 9` produces a sequence consisting of only 7 numbers: `1 9 81 729 6561 59049 531441 finished`. When the loop reaches the 8th element of the sequence, the result is 4,782,969, which is greater than 1,000,000, so the break statement in line 5 is executed. This causes the program to end the loop and proceed to the first statement after the loop, which prints `finished`.

6.2.3.2 *The `continue` Statement*

The `continue` statement skips the rest of the current iteration of a loop and returns to the beginning of the loop for the next iteration. The program below illustrates its use.

Code: continuedemo.py

```python
for x in range(4, -4, -1):
    if x==0:
        continue
    print('{0:6.3f}'.format(1/x), end=' ')
```

The `if-continue` block skips the case of `x=0` in order to avoid division by zero in the `print` function and produces the following output: `0.250 0.333 0.500 1.000 -1.000 -0.500 -0.333`.

6.2.4 Loops and Array Operations

Loops are often used to modify the elements of an array sequentially. For example, suppose you want to square each element of the array `a = np.linspace(0, 32, 1e7)`. This is a hefty array with 10 million elements. Nevertheless, the following loop does the trick:

Code: slow_loops.py

```python
import numpy as np
import time

a = np.linspace(0.0, 32.0, 10000000)  # 10 million
print(a)
startTime = time.process_time()
for i in range(len(a)):
    a[i] = a[i] * a[i]
endTime = time.process_time()
print(a)
print(f"Run time = {endTime - startTime} seconds")
```

Running this on a 2018 MacBook Pro returns the result in about 3.0 seconds—not bad for having performed 10 million multiplications. Notice that we have introduced the `time` module, which we use to measure how long (in seconds) it takes the computer to perform the 10 million multiplications.

Of course, we can perform the same calculation using the array multiplication introduced in Chapter 4. To do so, replace the `for` loop in lines 7–8 above with the simple array multiplication code in line 7 below. Here it is:

Code: fast_array.py

```
1  import numpy as np
2  import time
3
4  a = np.linspace(0.0, 32.0, 10000000)  # 10 million
5  print(a)
6  startTime = time.process_time()
7  a = a * a
8  endTime = time.process_time()
9  print(a)
10 print(f"Run time = {endTime - startTime} seconds")
```

Running this on the same computer returns the results in about 1/50 of a second, more than 100 times faster than obtained using a loop. This illustrates an important point: **`for` loops are slow**. Array operations run much faster and are preferred in any case where you have a choice. Sometimes, finding an array operation equivalent to a loop can be difficult, especially for a novice. Nevertheless, doing so pays rich rewards in execution time. Moreover, the array notation is usually simpler and clearer, providing further reasons to prefer array operations over loops.

On the other hand, there are times when no array-based operation is possible, and a loop must be used, even if it's slow. This can occur when one operation in a loop depends on the outcome of a previous operation in a loop.[1] We will not delve into this topic here but you will see examples of it later on.

6.3 LIST COMPREHENSIONS

List comprehensions are a unique feature of core Python for processing and constructing lists. We introduce them here because they use a looping process. They are commonly used in Python coding and often provide elegant compact solutions to common computing tasks. Rather than write down a general form, we provide a few examples illustrating their form and use. Consider, for example, the 3 ×3 matrix

```
In[1]: A = [[1, 2, 3],
  ...:      [4, 5, 6],
  ...:      [7, 8, 9]]
```

[1]The Numba package can help in such cases, but we defer discussion of Numba to Chapter 13.

Suppose you want to construct a vector from the diagonal elements of this matrix. You could do so with a `for` loop with an accumulator as follows:

```
In[2]: diag = []
In[3]: for i in [0, 1, 2]:
  ...:     diag.append(A[i][i])
  ...:

In[4]: diag
Out[4]: [1, 5, 9]
```

Here, we used the `append()` list method to add elements to the list `diag` one at a time.

List comprehensions provide a simpler, cleaner, and faster way to build a list of the diagonal elements of `A`:

```
In[5]: diagLC = [A[i][i] for i in [0, 1, 2]]

In[6]: diagLC
Out[6]: [1, 5, 9]
```

A one-line list comprehension replaces a three-line accumulator plus loop code. Suppose you now want the square of this list:

```
In[7]: [y*y for y in diagLC]
Out[7]: [1, 25, 81]
```

Notice here how `y` serves as a dummy variable accessing the various elements of the list `diagLC`.

Extracting a row from a 2-dimensional array such as `A` is quite easy. For example, the second row is obtained quite simply in the following fashion:

```
In[8]: A[1]
Out[8]: [4, 5, 6]
```

Obtaining a column is not as simple, but a list comprehension makes it quite straightforward:

```
In[9]: c1 = [a[1] for a in A]
In[10]: c1
Out[10]: [2, 5, 8]
```

Another slightly less elegant way to accomplish the same thing is

```
In[11]: [A[i][1] for i in range(3)]
Out[11]: [2, 5, 8]
```

Suppose you have a list of numbers and you want to extract all the elements of the list that are divisible by three. A slightly fancier list comprehension accomplishes the task quite simply and demonstrates a new feature:

```
 In[12]: y = [-5, -3, 1, 7, 4, 23, 27, -9, 11, 41]
 In[13]: [x for x in y if x%3==0]
Out[13]: [-3, 27, -9]
```

As you see in this example, a conditional statement can be added to a list comprehension. Here, it serves as a filter to select only those elements that are divisible by three.

6.4 HANDLING EXCEPTIONS

When a program crashes, it usually issues an error message when it terminates; that is, it *raises an exception* that lets you know what is wrong in the program and at what point. Usually, this exception is caused by a programming error, and you proceed to fix the problem. But in other cases, one can anticipate that an exception might arise under certain circumstances. In those cases, Python provides a way to smoothly handle an anticipated exception using its `try-except` syntax, a kind of conditional structure specifically designed to handle exceptions. The `try-except` syntax has the following form:

```
try:
# indented block of code that is supposed
# to run if there is no exception
except:
# code that runs when an exception occurs
```

A common error is dividing by zero, as we illustrate here with a routine that calculates the ratio of girls to boys on a team. Here, we use the try-except feature to capture this possibility and print an appropriate response.

Code: ratio_girls_to_boys0.py

```
girls = int(input("Enter number of girls on team: "))
boys = int(input("Enter number of boys on the team: "))
try:
    print(f"Ratio of girls to boys: {girls / boys:0.2f}")
except ZeroDivisionError:
    print("There are no boys on the team so the ratio is undefined")
```

Let's run this program for a couple of different inputs

```
 In[1]: %run ratio_girls_to_boys0.py
Enter number of girls on the team: 5
Enter number of boys on the team: 9
Ratio of girls to boys: 0.56

 In[2]: %run ratio_girls_to_boys0.py
Enter number of girls on the team: 11
Enter number of boys on the team: 0
There are no boys on the team, so the ratio is undefined
```

In the first running of the code, the code in the `try` block runs properly without raising an exception, so the `except` block is skipped. In the second run, the program raises an exception, which makes the flow move to the except block, where a message is printed. After the error is thus handled, the program moves on.

The `except` statement can be written with or without a keyword specifying the anticipated error. In this case, the `except` statement is written with the keyword `ZeroDivisionError`. This is the preferred mode as it catches only the anticipated type of error. If a different error is encountered, we want the code to crash—raise an unanticipated exception and halt execution—so that we can learn about and deal with the problem. You can include two or more keywords in the exception statement so that two or more exceptions are caught by the same `except` statement. For example, the statement

```
except (ZeroDivisionError, ValueError)
```

catches both types of errors listed and processes them in the same except block. Alternatively, the program could have two separate except blocks that process the different exceptions in different blocks.

The program `ratio_girls_to_boys0.py` will raise a ValueError exception if the user enters a non-numeric string, such as `9p` or `xy`. These kinds of errors can be caught using an `except-try` block together with a while statement that asks the user to reenter the numbers of boys and girls:

Code: ratio_girls_to_boys1.py

```python
while True:
    try:
        girls = int(input("Enter number of girls on team: "))
        boys = int(input("Enter number of boys on the team: "))
        break
    except ValueError:
        print("Your entry is not an integer, try again\n")
try:
    print(f"Ratio of girls to boys: {girls / boys:0.2f}")
except ZeroDivisionError:
    print("There are no boys on the team so the ratio is undefined")
```

Let's see how this works:

```
 In[3]: %run ratio_girls_to_boys1.py
Enter number of girls on team: 9p
Your entry is not an integer, try again

Enter number of girls on team: 9
Enter number of boys on the team: 4
Ratio of girls to boys: 2.25
```

The code uses a `break` statement to exit the `while` loop if proper numbers are input inside the `try` block.

The `chickens_pigs.py` program we discussed at the beginning of this chapter raised a `ValueError` if the user input numbers for which there was no solution. Instead of raising an error and terminating the program, we can use `try-except` statements to alert the user to input numbers for which an answer exists. The following code uses `assert` statements to ensure that the total number of legs input is an even number, thus avoid one potential problem, and another to ensure that the number of animals entered is neither too big nor too small for an answer to exist. An `assert` statement merely checks that a logical expression is True. If it is True, the `assert` statement does nothing; if it's False, it raises an `AssertionError`, which in the code below alerts the user and asks for new inputs.

Code: chickens_pigs_screen.py

```python
while True:
    try:
        legs = int(input("Enter the total number of legs: "))
        assert legs % 2 == 0
        break
    except AssertionError:
        print("Total number of legs must be an even number")
while True:
    try:
        animals = int(input("Enter the total number of animals: "))
        assert 4 * animals >= legs and 2 * animals <= legs
        break
    except AssertionError:
        print("Number of animals must be >= {} and <= {}"
              .format((legs + 2) // 4, legs // 2))

pigs = legs // 4  # maximum possible number of pigs
chickens = animals - pigs

while legs != 4 * pigs + 2 * chickens:
    pigs -= 1
    chickens += 1
    if pigs < 0:
        raise ValueError("There is no solution for these inputs.")

print("\nNumber of chickens =", chickens)
print("Number of pigs =", pigs)
print("Number of legs =", 4 * pigs + 2 * chickens)
print("Number of animals =", pigs + chickens)
```

The `assert` statement is also frequently used for debugging code. You can enter it at various points in your code to check if the anticipated result at a certain point is indeed found. If not, it raises an `AssertionError` and stops the program, which can help locate coding or logic errors.

6.5 EXERCISES

1. Redo Exercise 3 (page 79) from Chapter 4 but this time make a dictionary named `greek` that does the entire Greek alphabet. Use a `for-enumerate` loop to populate your `greek` dictionary. Demonstrate that your code works by using your dictionary to print out the entire Greek alphabet on a single line on your computer screen. The keys to the dictionary should be the names of the letters written out in the Latin alphabet:

```
gkeys = ["alpha", "beta", "gamma", "delta", "epsilon",
         "zeta", "eta", "theta", "iota", "kappa", "lamda",
         "mu", "nu", "xi", "omicron", "pi", "rho",
         "sigma_alt", "sigma", "tau", "upsilon", "phi",
         "chi", "psi", "omega"]
```

   Recall from page 53 that the UTF-8 encoding for the Greek alphabet begins at 945. It proceeds consecutively through the Greek alphabet according to the list `gkeys` above. Note that the key for the Greek letter λ is spelled `lamda` instead of `lambda` because `lambda` is a reserved word in Python (see Table 2.2).

2. Write a program to calculate the factorial of a positive integer input by the user. Recall that the factorial function is given by $x! = x(x-1)(x-2)\ldots(2)(1)$ so that $1! = 1$, $2! = 2$, $3! = 6$, $4! = 24$, $5! = 120$, …

   (a) Write the factorial function using a Python `while` loop.

   (b) Write the factorial function using a Python `for` loop.

   Check your programs to ensure they work for 1, 2, 3, 5, and beyond, but especially for the first five integers.

3. The following Python program finds the smallest non-trivial (not 1) prime factor of a positive integer.

```
n = int(input("Input an integer > 1: "))
i = 2
while (n % i) != 0:
    i += 1
print(f"The smallest prime factor of {n} is {i}")
```

   (a) Type this program into your computer and verify that it works as advertised. Then, briefly explain how it works and why the `while` loop always terminates.

   (b) Modify the program to tell whether the integer input is a prime number. If it is not a prime number, write your program so that it prints out the smallest prime factor. Using your program, verify that the following integers are prime numbers: 2, 31, 101, 8191, 94811, 947431.

4. Consider the matrix list `x = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`. Write a list comprehension to extract the last column of the matrix [3, 6, 9]. Write another list comprehension to create a vector of twice the square of the middle column `[8, 50, 128]`.

5. Write a program *that uses loops* to calculate the value of an investment after some number of years specified by the user if:

   (a) The principal is compounded annually

   (b) The principle is compounded monthly

   (c) The principle is compounded daily

   Your program should ask the user for the initial investment (principal), the interest rate in percent, and the number of years the money will be invested (whole years only). For an initial investment of $1000 at an interest rate of 6%, after ten years we get $1790.85 when compounded annually, $1819.40 when compounded monthly, and $1822.03 when compounded daily, assuming 12 months in a year and 365.24 days in a year, where the monthly interest rate is the annual rate divided by 12. The daily rate is the annual rate divided by 365 (don't worry about leap years).

6. Given a dictionary of people's names and ages like this:

```
ps={"Adele": 35, "Elton": 76, "Taylor": 34, "Billie": 22}
```

   write a program that prints out the names of the youngest and oldest members in the dictionary. Test your program using the above dictionary. Write your program to be scalable to dictionaries with more than four entries.

CHAPTER 7

Functions

In this chapter, you learn how to write your own ***functions****, similar to the functions provided by Python and NumPy. You learn how to write functions that* ***process NumPy arrays*** *efficiently. You learn how to write functions with variable numbers of arguments and how to pass function names (and the arguments of those functions) as an argument of a function you write. You learn about the concept of* ***namespace****, which isolates the names of variables and functions created inside a function from those created outside the function, with particular attention given to the subtle subject of passing mutable and immutable objects. You learn about anonymous functions (****lambda functions*** *in Python) and their uses. Objects and their associated methods and instance variables are discussed in the context of NumPy arrays. Finally, various features of Python functions are illustrated with least squares fitting routines.*

As you develop more complex computer code, organizing your code into modular blocks becomes increasingly important. One important means for doing so is *user-defined* Python functions. User-defined functions are a lot like built-in functions that we have encountered in core Python and in NumPy. The main difference is that user-defined functions are written by you. The idea is to define functions to simplify your code, improve its readability, and allow you to reuse the same code in different contexts.

The number of ways that functions are used in programming is so varied that we cannot enumerate all the possibilities. As our use of Python functions in scientific programming is somewhat specialized, we introduce only a few of the possible uses of Python functions, ones that are the most common in scientific programming.

DOI: 10.1201/9781032673950-7

7.1 USER-DEFINED FUNCTIONS

The NumPy package contains a vast number of mathematical functions. You can find a listing of them at http://docs.scipy.org/doc/numpy/reference/routines.math.html. While the list may seem pretty exhaustive, you will inevitably find that you need a function not available in the NumPy Python library. In those cases, you will want to write your own function.

In studies of optics and signal processing, one often runs into the sinc function, which is defined as

$$\operatorname{sinc} x \equiv \frac{\sin x}{x} .$$

Let's write a Python function for the sinc function. Here is our first attempt:

Code: sinc0.py

```
import numpy as np

def sinc(x):
    y = np.sin(x) / x
    return y
```

Every function definition begins with the word `def` followed by the name you want to give to the function, `sinc` in this case, then a list of arguments enclosed in parentheses, and finally terminated with a colon. In this case, there is only one argument, `x`, but generally, there can be as many arguments as you want, including no arguments. For the moment, we will consider the case of just a single argument.

The indented block of code following `def sinc(x):` defines what the function does. In this case, the first line calculates $\operatorname{sinc} x = \sin x/x$ and sets it equal to `y`. The `return` statement of the last line tells Python to return the value of y to the user.

We can try it out in the IPython shell. You can either `run` the program above that you wrote into a python file or you can type it in—it's only three lines long—into the IPython shell:

```
In[1]: def sinc(x):
  ...:     y = np.sin(x)/x
  ...:     return y
```

We assume you have already imported NumPy. Now, the function $\operatorname{sinc} x$ can be used from the IPython shell.

```
In[2]: sinc(4)
Out[2]: -0.18920062382698205
```

```
 In[3]: a = sinc(1.2)

 In[4]: a
Out[4]: 0.7766992383060219 4

 In[5]: np.sin(1.2)/1.2
Out[5]: 0.77669923830602194
```

Inputs and outputs 4 and 5 verify that the function does indeed give the same result as an explicit calculation of $\sin x/x$.

You may have noticed a problem with our definition of $\operatorname{sinc} x$ when $x = 0$. Let's try it out and see what happens

```
 In[6]: sinc(0.0)
Out[6]: nan
```

IPython returns `nan` or "not a number," which occurs when Python attempts to divide zero by zero. This is not the desired result as $\operatorname{sinc} x$ is, in fact, perfectly well defined for $x = 0$. You can verify this using L'Hopital's rule, which you may have learned in your study of calculus, or you can ascertain the correct answer by calculating the Taylor series for $\operatorname{sinc} x$. Here is what we get:

$$\operatorname{sinc} x = \frac{\sin x}{x} = \frac{x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots}{x} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} + \ldots .$$

From the Taylor series, it is clear that $\operatorname{sinc} x$ is well-defined at and near $x = 0$ and that, in fact, $\operatorname{sinc}(0) = 1$. Let's modify our function so that it gives the correct value for $x = 0$.

```
 In[7]:  def sinc(x):
   ...:      if x == 0.0:
   ...:          y = 1.0
   ...:      else:
   ...:          y = np.sin(x) / x
   ...:      return y

 In[8]:  sinc(0)
Out[8]:  1.0

 In[9]:  sinc(1.2)
Out[9]:  0.7766992383060219
```

Now, our function gives the correct value for $x = 0$ as well as for values different from zero.

By the way, we can also write the program a bit more efficiently as

```
 In[10]: def sinc(x):
    ...:     if x == 0.0:
```

```
   ...:         return 1.0
   ...:     else:
   ...:         return np.sin(x) / x
```

7.1.1 Looping Over Arrays in User-Defined Functions

The code for sinc *x* works just fine when the argument is a scalar (single) number. However, if the argument is a NumPy array, we run into a problem, as illustrated below.

```
In[11]: x = np.linspace(0., 5., 11)

In[12]: x
Out[12]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. ,
                4.5, 5. ])
In[13]: sinc(x)
Out[13]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. ,
                4.5, 5. ])
In[14]: sinc(x)
ValueError: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()
```

The `if` statement in Python evaluates the truth value of a single scalar variable, not a multi-element array. When Python tries to evaluate the truth value of a multi-element array, it doesn't know what to do and, therefore, returns an error.

An obvious way to handle this problem is to write the code so that it processes the array one element at a time, which you could do using a `for` loop, as illustrated below.

Code: sinc2.py

```
import numpy as np

def sinc(x):
    y = []  # empty list to store results
    for xx in x:  # loops over in x array
        if xx == 0.0:  # appends result of 1.0 to
            y += [1.0]  # y list if xx is zero
        else:  # appends result of sin(xx)/xx to y
            y += [np.sin(xx) / xx]  # list if xx is not zero
    # converts y to array and returns array
    return np.array(y)
```

The `for` loop evaluates the elements of the `x` array one by one and appends the results to the list `y` one by one. When the loop finishes, the function converts the list to an array and returns the array. Figure 7.1 shows the plot of the `sinc(x)` function generated by the above code.

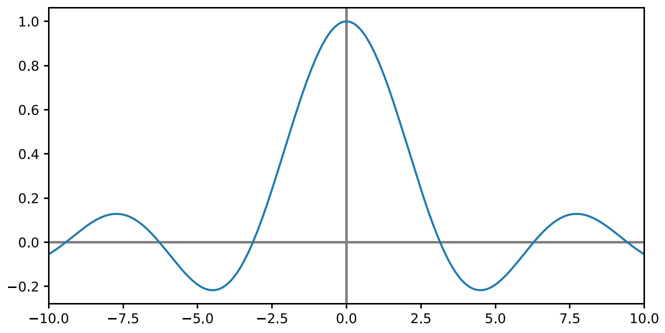


Figure 7.1 Plot of user-defined `sinc(x)` function.

7.1.2 Fast Array Processing for User-Defined Functions

While using loops to process arrays works, there are better ways to accomplish a task in Python. The reason is that loops in Python are executed rather slowly, as we saw in Section 6.2.4. To deal with this problem, the developers of NumPy introduced functions designed to process arrays quickly and efficiently. For the present case, we need a conditional statement or function that can process arrays directly. The function we want is called `where` and is part of the NumPy library. The `where` function has the form

```
where(condition, output if True, output if False)
```

The first argument of the `where` function is a conditional statement involving an array. The `where` function applies the condition to the array element by element, returns the second argument for those array elements for which the condition is `True`, and returns the third argument for those array elements that are `False`. We can apply it to the `sinc(x)` function as follows:

Code: sinc_test0.py

```python
import numpy as np

def sinc(x):
    z = np.where(x == 0.0, 1.0, np.sin(x) / x)
    return z
```

The `where` function creates the array `y` and sets the elements of `y` equal to 1.0, where the corresponding elements of `x` are zero, and otherwise sets the corresponding elements to `sin(x)/x`. Let's try it out.

```
In[15]: x = np.linspace(0.0, 5.0, 6)

In[16]: x
```

```
Out[16]: array([0., 1., 2., 3., 4., 5.])

 In[17]: sinc(x)
/Users/dp/python/functions/sinc_test0.py:4:
RuntimeWarning: invalid value encountered in true_divide
z = np.where(x == 0.0, 1.0, np.sin(x) / x)
Out[17]:
array([ 1.        ,  0.84147098,  0.45464871,  0.04704   ,
-0.18920062, -0.19178485])
```

This program produces the correct answer but issues a `RuntimeWarning`. The reason for the warning is that the `np.where` function evaluates the function, in this case `sinc(x)`, for all values in the `x` array, which includes a division by zero for the first element of `x`. However, it returns only the values specified by the condition in the `where` function, and thus discards the erroneous result. Thus, the `RuntimeWarning` can be safely ignored because it does not figure in the final output of the function.

The following code suppresses the annoying `RuntimeWarning`

Code: sinc_test1.py

```
1 import numpy as np
2 import warnings
3
4
5 def sinc(x):
6     warnings.filterwarnings("ignore", category=RuntimeWarning)
7     z = np.where(x == 0.0, 1.0, np.sin(x) / x)
8     warnings.filterwarnings("always", category=RuntimeWarning)
9     return z
```

Line 6 suppresses the `RuntimeWarning`. Line 8 restores the `RuntimeWarning` *after* the `where` function is run. Otherwise, the `RuntimeWarning` would be turned off for all other code that runs subsequently.

This code executes much faster than the code using a `for` loop by 100 times or more, depending on the array size. Moreover, the new code is much simpler to write and read. An additional benefit of the `where` function is that it can handle single variables and arrays equally well. The code we wrote for the sinc function with the `for` loop cannot handle single variables. Of course we could rewrite the code so that it did, but the code becomes even more clunky. It's better just to use NumPy's `where` function.

7.1.2.1 The Moral of the Story

The moral of the story is that you should avoid using `for` and `while` loops to process arrays in Python programs whenever an array-processing method is available. As a beginning Python programmer, you may not always see how to avoid loops, and indeed, avoiding them is not always possible. But you should look for ways to avoid them, especially loops that iterate many times. As you become more experienced, you will find that using array-processing methods in Python becomes more natural. Using them can significantly speed up the execution of your code, especially when working with large arrays.

7.1.2.2 Vectorized Code and Ufuncs

Finally, a word about jargon. Programmers sometimes refer to using array-processing methods as *vectorizing* code. The jargon comes from the idea that an array of N elements can be regarded as an N-dimensional vector. Computer code that processes *vectors* as the basic unit rather than individual data elements is said to be *vectorized*.

Within the NumPy world, functions that operate on NumPy arrays on an element-by-element basis are called *universal functions* or *ufuncs* for short. NumPy ufuncs are vectorized. They also support array broadcasting, type casting, and other features associated with NumPy arrays.

Don't worry too much about the jargon or even its origin. But it's useful to understand when reading from different sources, online or otherwise, about Python code.

7.1.3 Functions with More than One Input or Output

Python functions can have any number of input arguments and can return any number of outputs (variables). For example, suppose you want a function that outputs (x, y)-coordinates of n points evenly distributed around a circle of radius r centered at the point (x_0, y_0). The inputs to the function would be r, x_0, y_0, and n. The outputs would be the (x, y)-coordinates. The following code implements this function.

Code: circleN.py

```python
import numpy as np


def circle(r, x0, y0, n):
    theta = np.linspace(0.0, 2.0 * np.pi, n, endpoint=False)
    x, y = r * np.cos(theta), r * np.sin(theta)
    return x0 + x, y0 + y
```

This function has four inputs and two outputs. In this case, the four inputs are simple numeric variables, and the two outputs are NumPy arrays. The inputs and outputs can be any combination of data types: arrays, lists, strings, *etc*. Of course, the body of the function must be written to be consistent with the prescribed data types.

Note that we have set `endpoint=False` in the call to `linspace` in order to avoid generating a value of `theta=2*np.pi`, which would duplicate the *x*-*y* data point generated by `theta=0`.

7.1.4 Type Hints

As of Python 3.5 and NumPy 1.20, you can indicate which types are preferred for Python function arguments (often called *parameters*) and returned objects. For example, the previous program can be written with type hints:

Code: circleNtypes.py

```python
import numpy as np


def circle(r: float, x0: float, y0: float, n: int
           ) -> (np.ndarray, np.ndarray):
    theta = np.linspace(0.0, 2.0 * np.pi, n, endpoint=False)
    x, y = r * np.cos(theta), r * np.sin(theta)
    return x0 + x, y0 + y
```

The input types are specified by adding a colon followed by the argument type for each argument, as indicated. The returned types are specified by an arrow (->) followed by the data types, as indicated above. The code typing in this function has *no effect* on how Python runs the code. You should think of it merely as a way of documenting your code, that is, as a way of indicating to a user what the function expects as inputs and what it returns as outputs. More information on this feature can be found here https://docs.python.org/3/library/typing.html and here https://numpy.org/devdocs/reference/typing.html#numpy.typing.ArrayLike.

7.1.5 Positional and Keyword Arguments

It is often useful to have function arguments that have some default setting. This happens when you want an input to a function to have some standard value or setting most of the time, but you would like to reserve the possibility of giving it some value other than the default value.

For example, in the program `circle` from the previous section, you might decide that under most circumstances, you want `n=12` points around the circle, like the points on a clock face, and you want the circle to be centered at the origin. In this case, you would rewrite the code to read

Code: circleKW.py

```python
import numpy as np


def circle(r, x0=0.0, y0=0.0, n=12):
    theta = np.linspace(0., 2. * np.pi, n, endpoint=False)
    x, y = r * np.cos(theta), r * np.sin(theta)
    return x0 + x, y0 + y
```

The default values of the arguments `x0`, `y0`, and `n` are specified in the argument of the function definition in the `def` line. Arguments whose default values are specified in this manner are called *keyword arguments*, and they can be omitted from the function call if the user is content using those values. For example, writing `circle(4)` is now a perfectly legal way to call the `circle` function, and it would produce 12 (x, y) coordinates centered about the origin $(x, y) = (0, 0)$. On the other hand, if you want the values of `x0`, `y0`, and `n` to be something different from the default values, you can specify their values as you would have before.

If you want to change only some of the keyword arguments, you can do so by using the keywords in the function call. For example, suppose you are content with having the circle centered on $(x, y) = (0, 0)$, but you want only 6 points around the circle rather than 12. Then you would call the `circle` function as follows:

```
        circle(2, n=6)
```

The unspecified keyword arguments keep their default values of zero, but the number of points `n` around the circle is now six instead of the default value of 12.

The normal arguments without keywords are called *positional arguments*; they have to appear *before* any keyword arguments and, when the function is called, must appear in the same order as specified in the function definition. The keyword arguments, if supplied, can appear in any order provided they appear with their keywords. If supplied without their keywords, they must also appear in the order they appear in the function definition. The following two function calls to `circle` give the same output.

```
 In[18]: circle(3, n=3, y0=4, x0=-2)
Out[18]:
(array([ 1. , -3.5,  -3.5]),
 array([ 4. , 6.59807621, 1.40192379]))

 In[19]: circle(3, -2, 4, 3) # w/o keywords, arguments
# supplied in order
Out[19]:
(array([ 1. , -3.5, -3.5]),
 array([ 4. , 6.59807621, 1.40192379]))
```

We pointed out previously that we set `endpoint=False` in the call to `linspace` in our definition of the `circle` function. The default value of `endpoint` is `True`. Thus, we see that `endpoint` is a keyword argument of NumPy's `linspace` function.

7.1.6 Variable Number of Arguments

While it may seem odd, leaving the number of arguments unspecified is sometimes useful. A simple example is a function that computes the product of an arbitrary number of numbers:

```
In[20]: def product(*args):
   ...:     print("args = {}".format(args))
   ...:     p = 1
   ...:     for num in args:
   ...:         p *= num
   ...:     return p
```

Placing the "*" before the `args` argument tells Python that `args` can have any number of entries. For example, here we give it three entries:

```
In[21]:  product(11., -2, 3)
args = (11.0, -2, 3)
Out[21]: -66.0
```

Here, we give only two arguments:

```
In[22]: product(2.31, 7)
args = (2.31, 7)
Out[22]: 16.17
```

The `print("args...)` statement in the function definition is not necessary, of course, but is put in to show that the argument `args` is a tuple inside the function. Here, the `*args` tuple argument is used because one does not know ahead of time how many numbers are to be multiplied together.

By the way, there is nothing special about the name `args`. We could have equally well have used `*nums` or `*items` or `*params`; it's the * before the argument name that is important here!

7.1.6.1 Use of Star Syntax with `zip` Function

One useful application of this syntax involves printing the output from zipped lists or arrays. Consider the following program:

Code: zipstar.py

```
import numpy as np

x = [23.1, 45.9, 38.4, 29.7]
```

```
y = np.sin(np.array(x))
z = y ** 4

print("\nUsing explicit referencing")
for data in zip(x, y, z):
    print(f"{data[0]:6.1f}, {data[1]:8.3f}, {data[2]:7.2f}")

print("\nUsing implicit referencing")
for data in zip(x, y, z):
    print("{0:6.1f}, {1:8.3f}, {2:7.2f}".format(*data))
```

The program prints the elements of the zipped list and two arrays in two different ways: first by explicitly referencing the three variables of the tuples formed by the `zip` function, and second by implicitly referencing the three variables using the star (asterisk) syntax introduced above. The results are identical, but the star syntax is a bit slicker.

```
In[23]: run zipstar.py

Using explicit referencing
  23.1,   -0.895,    0.64
  45.9,    0.940,    0.78
  38.4,    0.645,    0.17
  29.7,   -0.989,    0.96

Using implicit referencing
  23.1,   -0.895,    0.64
  45.9,    0.940,    0.78
  38.4,    0.645,    0.17
  29.7,   -0.989,    0.96
```

7.1.7 Passing a Function Name and Its Parameters as Arguments

The `*args` tuple argument is also quite useful in another context: when passing the name of a function as an argument in another function. In many cases, the function name that is passed may have several parameters that must also be passed but aren't known ahead of time. If this all sounds a bit confusing—functions calling other functions with *arbitrary* parameters—a concrete example will help you understand.

Suppose we have the following function that numerically computes the value of the derivative of an arbitrary function $f(x)$:

Code: derivA.py

```
def deriv(f, x, h=1.e-9, *params):
    return (f(x + h, *params) - f(x - h, *params)) / (2. * h)
```

The argument `*params` is an optional *positional* argument. We begin by demonstrating the use of the function `deriv` without using the optional `*params` argument. Suppose we want to compute the derivative of the function $f_0(x) = 4x^5$. First, we define the function:

```
In[24]: def f0(x):
   ...:     return 4 * x**5
```

Now let's find the derivative of $f_0(x) = 4x^5$ at $x = 3$ using the function `deriv`:

```
In[25]: deriv(f0, 3)
Out[25]: 1620.0001482502557
```

The exact result is 1620, so our function to numerically calculate the derivative works pretty well (it's accurate to about 1 part in 10^7).

Suppose we had defined a more general function $f_1(x) = ax^p$ as follows:

```
In[26]: def f1(x, a, p):
   ...:     return a * x**p
```

Suppose we want to calculate the derivative of this function for a set of numerical values `x` for a particular set of the parameters a and p. Now, we face a problem because it might seem that there is no way to pass the values of the parameters a and p to the `deriv` function. Moreover, this is a generic problem for functions such as `deriv` that use a function as an input because different functions you might want to use as inputs generally come with a different number of parameters. Therefore, we would like to write our program `deriv` so that it works, irrespective of how many parameters are needed to specify a particular function.

This is what the optional positional argument `*params` defined in `deriv` is for: to pass parameters of `f1`, like a and p, through `deriv`. To see how this works, let's set a and p to be 4 and 5, respectively, the same values we used in the definition of `f0`, so that we can compare the results:

```
In[27]: deriv(f1, 3, 1.e-9, *(4, 5))
Out[27]: 1620.0001482502557
```

We get the same answer as before, but this time we have used `deriv` with a more general form of the function $f_1(x) = ax^p$.

The order of the parameters a and p is important. The function `deriv` uses `x`, the first argument of `f1`, as its principal argument and then uses `a` and `p`, in the same order that they are defined in the function `f1`, to fill in the additional arguments—the parameters—of the function `f1`.

Beware, the `params` argument *must* be a tuple. If there is only one parameter, as there is for the function $g(x) = (x + a)/(x - a)$, then the call to the derivative function would work like this:

```
In[28]: def g(x, a):
   ...:     return (x + a) / (x - a)

In[29]: a = 1.0

In[30]: x = np.linspace(0, 2, 6)

In[31]: deriv(g, x, 1.e-9, *(a, ))
Out[31]:
array([ -2.00000011,  -5.55555557, -49.99999792, -50.00000414,
-5.55555602,   -2.00000017])
```

The comma following `a` in the argument `*(a, )` is needed so that `(a, )` is understood by Python to be a tuple.

Optional arguments must appear after the regular positional and keyword arguments in a function call. The order of the arguments must adhere to the following convention:

```
def func(pos1, pos2, ..., keywd1, keywd2,
..., *args, **kwargs):
```

That is, the order of arguments is: positional arguments first, then keyword arguments, then optional positional arguments (`*args`), then optional keyword arguments (`**kwargs`). Note that to use the `*params` argument, we had to explicitly include the keyword argument `h` even though we didn't need to change it from its default value.

Python also allows for a variable number of keyword arguments—`**kwargs`—in a function call, that is, an argument preceded by `**`. While `args` is a tuple, `kwargs` is a dictionary, so the value of an optional keyword argument is accessed through its dictionary key. To use the `**kwargs` format, we rewrite our `deriv` function using two stars (`**params`):

Code: derivK.py

```
def deriv(f, x, h=1.e-9, **params):
    return (f(x + h, **params) - f(x - h, **params)) / (2. * h)
```

Next, we define a dictionary:

```
In[32]: d = {'a': 4, 'p': 5}
```

And finally, we input our optional keyword arguments using `**d`:

```
In[33]: deriv(f1, 3, **d)
Out[33]:1620.0001482502557
```

We can also include our optional keyword arguments as a dictionary literal:

```
In[34]: deriv(f1, 3, **{'a': 4, 'p': 5})
Out[34]:1620.0001482502557
```

Note that when using `**kwargs`, you can omit keyword arguments, in this case `h`, if you want to use the default value(s).

7.2 NAMESPACE AND SCOPE IN PYTHON

Functions are like mini-programs within the programs that call them. Each function has a set of variables with specific names that are, to some degree or other, isolated from the calling program. We will get more specific about just how isolated those variables are below, but before we do, we introduce the concept of a *namespace*.

Each function has its own namespace, which is a mapping of variable names to objects, like numerics, strings, lists, and arrays. It's a kind of dictionary that maps variable names to objects.

The calling program has its own namespace that is distinct from the namespace of any functions it calls. The distinctiveness of these namespaces plays a vital role in how functions work.

7.2.1 Scope: Four Levels of Namespaces in Python

One generally distinguishes four different levels of namespaces in Python: built-in, global, enclosing, and local. Each of these namespaces has its own *scope*.

1. The *built-in namespace* consists of all of Python's built-in objects. These are always available whenever a Python program is running. Among other things, they include Python functions we have encountered like `len`, `list`, `range`, `zip`, *etc.*, as well as exception names like `ValueError` and `ZeroDivistionError`.

2. The *global namespace* consists of all the objects defined by the main program (that you write). All the names in this namespace remain in existence for as long as the program runs.

3. The *local namespace*, when the main program calls a function, a new *local namespace* is created for that function; it remains in existence until the function terminates.

4. The *enclosing namespace*, it's also possible to call a function from within a function. A new *local namespace* is created for the new function; in that case, the namespace of the first function is called the *enclosing namespace*.

This hierarchy of namespaces, local, enclosing, global, and built-in, often referred to as *LEGB*, defines how Python looks for names of objects (*i.e.*, variables, functions, *etc.*).

The *scope* of a namespace defines where Python looks for names. If the program is executing a local function, the interpreter first looks in the local namespace for any name, let's say `x`, that it references. Thus, the scope of the local namespace is limited to only within the function. If `x` isn't defined in the local namespace, the interpreter looks for `x` in the enclosing namespace. If it's there, it uses it. Thus, the scope of this namespace is the defining function and any functions defined within that function. If the interpreter doesn't find the name in the enclosing namespace, it looks in the global namespace. Finally, if `x` isn't in any of these namespaces, it looks for it in the built-in namespace. So, the scope of the names in the global and built-in namespaces are the entire program.

The program below illustrates how the scopes of the different namespaces work.

Code: scope.py

```
1  def f():
2      y = "enclosing"
3      z = "enclosing"
4      print(f"(2) inside f: x={x}, y={y}, z={z}")
5
6      def g():
7          z = "local"
8          print(f"(3) inside g: x={x}, y={y}, z={z}")
9
10     g()
11     print(f"(4) inside f: x={x}, y={y}, z={z}")
12
13
14 x = "global"
15 y = "global"
16 z = "global"
17 print(f"(1)  in main: x={x}, y={y},    z={z}")
18 f()
19 print(f"(5)  in main: x={x}, y={y},    z={z}")
```

The program begins on line 14 where it defines three strings `x`, `y`, and `z`, setting each to `"global"`. In line 18, the main program calls the function `f()` with no arguments. The first two lines in `f()`, lines 2 and 3 in the program listing, redefine the variables `y` and `z` to be `"enclosing"`. Then when `x`, `y`, and `z` are printed in line 4, we see from the program output below that `y` and `z` have the values assigned to them within the function `f()`, but `x`, which isn't redefined in `f()` takes its value from the main program that called `f()`. When the function `g()` is called from within `f()`, `z` is set equal to `"local"`. Because only z is modified, when x, y, and z are printed out from within g(), the value of z is the value assigned in g(), but the value y is the value it was assigned in f(), and x is the value it was assigned in the main program, because neither f() nor g() reassigned x to another object. After returning to f(), printing x, y, and z gives the same answers it gave before g() was called. And finally, after returning to the main program, printing x, y, and z gives the same answers it gave before f() was called.

```
In[1]: run scope.py
(1)  in main: x=global, y=global,    z=global
(2) inside f: x=global, y=enclosing, z=enclosing
(3) inside g: x=global, y=enclosing, z=local
(4) inside f: x=global, y=enclosing, z=enclosing
(5)  in main: x=global, y=global,    z=global
```

7.2.2 Variables and Arrays Created Entirely Within a Function

One thing the LEGB hierarchy means is that names *created entirely within* a function cannot be seen by the program that calls the function. This is important because it means you can create and manipulate variables and arrays, giving them any name you please, without affecting any variables or arrays outside the function, even if the variables and arrays inside and outside a function share the same name. It also means that any objects created within a function are not available to the calling function unless the object is explicitly passed to the calling program in the return statement.

To see how this works, let's rewrite our program to plot the sinc function using the sinc function definition that uses the where function.

Code: sinc_test0.py

```python
import numpy as np

def sinc(x):
    z = np.where(x == 0.0, 1.0, np.sin(x) / x)
    return z
```

We save this program in a file named sinc_test.py. Running this program by typing run sinc_test0.py in the IPython terminal creates the x and y arrays. Notice that the array variable z is only defined within the function definition of sinc. If we ask IPython to print out the arrays, x, y, and z, we get some interesting and informative results, as shown below.

```
In[2]: run sinc_test.py

In[3]: x
Out[3]: array([-10.  , -9.99969482, -9.99938964,
   ...:          9.9993864,  9.99969482, 10.  ])

 In[4]: y
Out[4]: array([-0.05440211, -0.05437816, -0.0543542 ,
   ...:        -0.0543542 , -0.05437816, -0.05440211])

 In[5]: z

------------------------------------------------------------
NameError         Traceback (most recent call last)
NameError: name 'z' is not defined
```

When we type in `x` at the `In` [3]: prompt, IPython prints out the array `x` (some of the output is suppressed because the array `x` has many elements); similarly for `y`. But when we type `z` at the In [5]: prompt, IPython raises a `NameError` because `z` is not defined. The IPython terminal is working in the same *namespace* as the main program, the global namespace. But the namespace of the `sinc` function is isolated from the namespace of the program that calls it and therefore isolated from IPython.

This also means that when the `sinc` function ends with `return z`, it doesn't return the name `z`, but instead returns only the array object to the main program in line 11, where the name `y` is bound to the object created within the function. The name `z` along with the local namespace for `sinc` is discarded when the `sinc` function terminates.

7.2.3 Passing Lists and Arrays to Functions: Mutable and Immutable Objects

What happens to a variable or an array passed to a function when the variable or array is *changed* within the function? It turns out that the answers depend on whether the object is mutable, like lists, dictionaries, and NumPy arrays, or immutable, like integers, floats, strings, Booleans, and tuples.

If an object is immutable, any attempt to change the object will create a new object so that the original object is left unchanged.

If an object is mutable, such as a list, dictionary, or NumPy array, then changes to the elements of the object will be reflected in the object in the calling program. However, if the whole object is redefined within the function, a new object will be created, even if the name of the new object is the same, and the original object will be left unchanged in the calling program.

The program below illustrates how Python handles single variables *vs.* how it handles lists and arrays.

Code: passing_vars.py

```python
import numpy as np


def f(stg, flt, tup, dct, lis, arr):
    stg = "I am doing fine"
    flt = np.pi ** 2
    tup = (1.1, 2.9)
    dct["Dave"] = 70.1
    lis[-1] = 'end'
    arr[0] = 963.2
    return stg, flt, tup, dct, lis, arr


stg = "How do you do?"
flt = 5.0
tup = (97.5, 82.9, 66.7)
dct = {"Lucy": 3.2e6, "Ardi": 4.4e6}
lis = [3.9, 5.7, 7.5, 9.3]
arr = np.array(lis)

print('******************* Before function call ******************')
print(f"stg = {stg}")
print(f"flt = {flt:4.2f}")
print(f"tup = {tup}")
print(f"dct = {dct}")
print(f"lis = {lis}")
print(f"arr = {arr}")
print('***********************************************************')
print('********************** function call **********************')

stg1, flt1, tup1, dct1, lis1, arr1 = f(stg, flt, tup, dct, lis, arr)

print('************** Variables returned by function *************')
print(f"stg1 = {stg1}")
print(f"flt1 = {flt1:4.2f}")
print(f"tup1 = {tup1}")
print(f"dct1 = {dct1}")
print(f"lis1 = {lis1}")
print(f"arr1 = {arr1}")
print('**********  Original variables after function call ********')
print(f"stg = {stg}")
print(f"flt = {flt:4.2f}")
print(f"tup = {tup}")
print(f"dct = {dct}")
print(f"lis = {lis}")
print(f"arr = {arr}")
print('***********************************************************')
```

The function `f` has six arguments: a string `stg`, a dictionary `dct`, a float `flt`, a tuple `tup`, a list `lis`, and a NumPy array `arr`. The function `f` modifies each of these arguments and then returns the modified `stg, dct, flt, tup, lis, arr` to the calling program as `stg1, dct1, flt1, tup1, lis1, arr1`. Running the program produces the following output:

```
In[6]: run passingVars.py
****************** Before function call *****************
stg = How do you do?
flt = 5.00
tup = (97.5, 82.9, 66.7)
dct = {'Lucy': 3200000.0, 'Ardi': 4400000.0}
lis = [3.9, 5.7, 7.5, 9.3]
arr = [3.9 5.7 7.5 9.3]
**********************************************************
********************* function call *********************
************* Variables returned by function ************
stg1 = I am doing fine
flt1 = 9.87
tup1 = (1.1, 2.9)
dct1 = {'Lucy': 3200000.0, 'Ardi': 4400000.0, 'Dave': 70.1}
lis1 = [3.9, 5.7, 7.5, 'end']
arr1 = [963.2   5.7   7.5   9.3]
**********  Original variables after function call *******
stg = How do you do?
flt = 5.00
tup = (97.5, 82.9, 66.7)
dct = {'Lucy': 3200000.0, 'Ardi': 4400000.0, 'Dave': 70.1}
lis = [3.9, 5.7, 7.5, 'end']
arr = [963.2   5.7   7.5   9.3]
**********************************************************
```

The program prints out three blocks of variables separated by asterisks. The first block merely verifies that the contents of `stg`, `flt`, `tup`, `dct`, `lis`, and `arr` are those assigned in lines 14–19. Then the function `f` is called. The next block prints the output of the call to the function `f`, namely the variables `stg1`, `flt1`, `tup1`, `dct1`, `lis1`, and `arr1`. The results verify that the function modified the inputs as directed by the `f` function.

The third block prints out the variables `stg`, `flt`, `tup`, `dct`, `lis`, and `arr` from the calling program *after* the function `f` was called. These variables served as the inputs to the function `f`. Examining the output from the third printing block, we see that the string `stg`, the float `flt`, and the tuple `tup` are unchanged after the function call. This is probably what you would expect. On the other hand, we see that the dictionary `dct`, the list `lis`, and the array `arr` are changed after the function call. This might surprise you! But these are important points to remember, so we summarize them in two bullet points here:

- Changes to immutable arguments of a function, such as strings, floats, and tuples, within the function do not affect their values in the calling program.

- Changes to the elements of mutable arguments of a function, such as the elements of lists and arrays, are reflected in the values of the same list and array elements in the calling function.

The point is that simple numerics, strings, and tuples are immutable, while dictionaries, lists, and arrays are mutable. Because immutable objects can't be changed, changing them within a function creates new objects with the same name inside of the function, but the old immutable objects used as arguments in the function call remain unchanged in the calling program. On the other hand, if elements of mutable objects like those in lists or arrays are changed, then those elements that are changed inside the function are also changed in the calling program.

7.3 ANONYMOUS FUNCTIONS: LAMBDA EXPRESSIONS

Python provides another way to generate functions called *lambda* expressions. A lambda expression is an in-line function that can be generated on the fly to accomplish some small task, often where a function name is needed as input to another function and thus is used only once.

You can assign a lambda expression a name, but you don't need to; hence, they are sometimes called *anonymous* functions.

A lambda expression uses the keyword `lambda` and has the general form

```
lambda arg1, arg2, ... : output
```

The arguments `arg1, arg2, ...` are inputs to a lambda, just as for a function, and the output is an expression using the arguments.

While lambda expressions need not be named, we illustrate their use by comparing a conventional Python function definition to a lambda expression to which we give a name. First, we define a conventional Python function:

```
In[1]: def f(a, b):
  ...:     return 3 * a + b**2

In[2]: f(2, 3)
Out[2]: 15
```

Next, we define a lambda expression that does the same thing:

```
In[3]: g = lambda a, b: 3 * a + b**2

In[4]: g(2, 3)
Out[4]: 15
```

The lambda expression defined by g does the same thing as the function f. Such lambda expressions are useful when you need a very short function definition, usually used locally only once or perhaps a few times.

Lambda expressions can be useful as function arguments, particularly when extra parameters need to be passed with the function. In Section 7.1.7, we saw how Python functions can do this using optional arguments, *args and **kwargs. Lambda expressions provide another means for accomplishing the same task. To see how this works, recall our definition of the function to take the derivative of another function:

Code: derivA.py

```
def deriv(f, x, h=1.e-9, *params):
    return (f(x + h, *params) - f(x - h, *params)) / (2. * h)
```

and our definition of the function

```
In[5]: def f1(x, a, p):
  ...:     return a*x**p
```

Instead of using the *params optional argument to pass the values of the parameters a and p, we can define a lambda expression that is a function of x alone, with a and p set in the lambda expression.

```
In[6]: g = lambda x: f1(x, 4, 5)
```

The function g defined by the lambda expression is the same as f1(x, a, p) but with a and p set to 4 and 5, respectively. Now we can use deriv to calculate the derivative at $x = 3$ using the lambda function g

```
In[7]: deriv(g, 3)
Out[7]: 1620.0001482502557
```

Of course, we get the same answer as we did using the other methods.

You might wonder why we can't just insert f1(x, 4, 5) as the argument to deriv. The reason is that you need to pass the *name* of the function, not the function itself. We assign the name g to our lambda expression and then pass that name through the argument of deriv.

Alternatively, we can insert the whole lambda expression in the argument of deriv where the function name goes:

```
In[8]: deriv(lambda x: f1(x, 4, 5), 3)
Out[8]: 1620.0001482502557
```

This works too. In this case, however, we never defined a function name for our lambda expression. Our lambda expression is indeed an *anonymous function*.

You may recall that we already used lambda expressions in Section 5.2.3, where we discussed how to print formatted arrays. Several nifty programming tricks can be realized using `lambda` expressions, but we will not go into them here. Look up lambdas on the web if you are curious about their more exotic uses.

7.4 NUMPY OBJECT ATTRIBUTES: METHODS AND INSTANCE VARIABLES

You have already encountered quite a number of functions that are part of either NumPy or Python. But there is another way in which Python implements things that act like functions: these are the *methods* associated with an object that we introduced in Section 4.5. Recall from Section 4.5 that strings, arrays, lists, and other such data structures in Python are not merely the numbers or strings we have defined them to be. They are *objects*. In general, an object in Python has associated with it a number of *attributes*, which are either *instance variables* associated with the object or specialized functions called *methods* that act on the object.

Let's start with the NumPy array. A NumPy array is a Python object and has associated with it many attributes: instance variables and methods. Suppose, for example, we create a NumPy array `a = np.sin(np.exp(np.arange(10)))`, which creates an array of 10 numbers between -1. and 1. An example of an instance variable associated with an array is the size or number of elements in the array. An instance variable of an object in Python is accessed by typing the object name followed by a period followed by the variable name. The code below illustrates how to access two different instance variables of an array: its size and data type.

```
 In[1]: a = np.sin(np.exp(np.arange(10)))

 In[2]: a.size
Out[2]: 10

 In[3]: a.dtype
Out[3]: dtype('float64')
```

Any object in Python can, and in general does, have a number of instance variables that are accessed in just the way demonstrated above, with a period and the instance variable name following the name of the particular object. In general, instance variables involve properties of the object that are stored by Python with the object and require no computation. Python just looks up the attribute and returns its value.

Objects in Python also have associated with them a number of specialized functions called *methods* that act on the object or its attributes. Methods generally involve Python performing a computation. Methods are accessed in a fashion similar to instance variables, by appending a period followed the method's name, which is followed by a pair of open-close parentheses, consistent with a method being a function. Often, methods are used with no arguments, as methods, by default, act on the object whose name they follow. In some cases. however, methods can take arguments. Examples of methods for NumPy arrays are sorting, calculating the mean, or standard deviation of the array. The code below illustrates a few array methods.

```
 In[4]: a
Out[4]:
array([ 0.84147098,  0.41078129,  0.89385495, 0.94447101,
-0.92876794, -0.68769141,  0.96486625, -0.21561571,
0.40176297, -0.79346054])
 In[5]: a.sum()                # sum
Out[5]: 1.8316718643908008

 In[6]: a.mean()               # mean or average
Out[6]: 0.1831671864390801

 In[7]: a.var()                # variance
Out[7]: 0.5336294146306282

 In[8]: a.std()                # standard deviation
Out[8]: 0.7304994282206032

 In[9]: a.sort()               # sort small to large

 In[10]: a
Out[10]:
array([-0.92876794, -0.79346054, -0.68769141, -0.21561571,
0.40176297,  0.41078129,  0.84147098,  0.89385495,
0.94447101,  0.96486625])
```

Notice that the `sort()` method has permanently changed the order of the elements of the array.

```
 In[11]: a.clip(-0.3, 0.8)
Out[11]:
array([-0.3, -0.3, -0.3, -0.21561571,  0.40176297, 0.41078129,
0.8,   0.8,   0.8,   0.8])
```

The `clip()` method provides an example of a method that takes an argument; in this case, the arguments are the lower and upper values to which array elements are clipped if their values are outside the range set by these values.

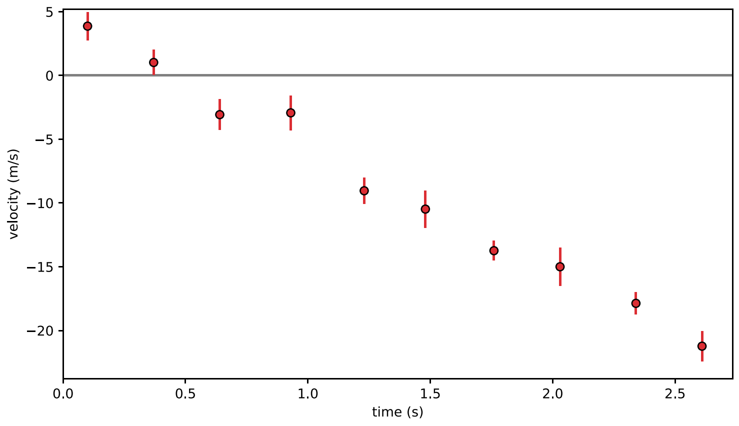


Figure 7.2 Velocity *vs.* time for falling mass.

7.5 EXAMPLE: LINEAR LEAST SQUARES FITTING

In this section, we illustrate how to use functions and methods to model experimental data.

In science and engineering, we often have some theoretical curve or *fitting function* that we would like to fit to experimental data. In general, the fitting function is of the form $f(x; a, b, c, ...)$, where x is the independent variable and $a, b, c, ...$ are parameters to be adjusted so that the function $f(x; a, b, c, ...)$ best fits the experimental data. For example, suppose we had some data of the velocity *vs.* time for a falling mass. If the mass falls only a short distance, such that its velocity remains well below its terminal velocity, we can ignore air resistance. In this case, we expect the acceleration to be constant and the velocity to change linearly in time according to the equation

$$v(t) = v_0 - gt, \tag{7.1}$$

where g is the local gravitational acceleration. We can fit the data graphically by plotting it as shown in Figure 7.2, and then drawing a line through the data. When we draw a straight line through the data, we try to minimize the distance between the points and the line, globally averaged over the whole data set.

While this can give a reasonable estimate of the best fit to the data, the procedure is rather *ad hoc*. We would prefer a more well-defined analytical method for determining what constitutes a "best fit." One way to do that is to consider the sum

$$S = \sum_i^n [y_i - f(x_i; a, b, c, ...)]^2, \tag{7.2}$$

where y_i and $f(x_i; a, b, c, ...)$ are the values of the experimental data and the fitting function, respectively, at x_i, and S is the square of their difference summed over all n data points. The quantity S is a sort of global measure of how much the fit $f(x_i; a, b, c, ...)$ differs from the experimental data y_i.

Notice that for a given set of data points $\{x_i, y_i\}$, S is a function only of the fitting parameters $a, b, ...$, that is, $S = S(a, b, c, ...)$. One way of defining a *best* fit, then, is to find the set of values of the fitting parameters $a, b, \ldots$ that minimize the value of S.

In principle, finding the values of the fitting parameters $a, b, ...$ that minimize S is a simple matter. Just set the partial derivatives of S with respect to the fitting parameter equal to zero and solve the resulting system of equations:

$$\frac{\partial S}{\partial a} = 0 \,, \quad \frac{\partial S}{\partial b} = 0 \,, \ldots \tag{7.3}$$

Because there are as many equations as there are fitting parameters, we should be able to solve the system of equations and find the values of the fitting parameters that minimize S. Solving those systems of equations is straightforward if the fitting function $f(x; a, b, ...)$ is linear in the fitting parameters. Some examples of fitting functions linear in the fitting parameters are:

$$\begin{aligned} f(x; a, b) &= a + bx \\ f(x; a, b, c) &= a + bx + cx^2 \\ f(x; a, b, c) &= a \sin x + be^x + ce^{-x^2} . \end{aligned} \tag{7.4}$$

For fitting functions such as these, taking the partial derivatives with respect to the fitting parameters, as proposed in Eq. (7.3), results in a set of algebraic equations that are linear in the fitting parameters $a, b, ...$ Because they are linear, these equations can be solved in a straightforward manner.

For cases where the fitting function is not linear in the fitting parameters, one can generally still find the values of the fitting parameters that minimize S, but finding them requires more work, which goes beyond our immediate interests here. See Section 9.3.3 for more on nonlinear fitting.

7.5.1 Linear Regression

We start by considering the simplest case, fitting a straight line to a set of $\{x_i, y_i\}$ data, such as the data set shown in Figure 7.2. Here, the fitting function is $f(x) = a + bx$, which is linear in the fitting parameters a and b. For a straight line, the sum in Eq. (7.2) becomes

$$S(a, b) = \sum_i (y_i - a - bx_i)^2 \,, \tag{7.5}$$

where the sum is over all the points in the $\{x_i, y_i\}$ data set. Finding the best fit in this case corresponds to finding the values of the fitting parameters a and b for which $S(a, b)$ is a minimum. To find the minimum, we set the derivatives of $S(a, b)$ equal to zero:

$$\begin{aligned}\frac{\partial S}{\partial a} &= \sum_i -2(y_i - a - bx_i) &&= 2\left(na + b\textstyle\sum_i x_i - \sum_i y_i\right) = 0 \\ \frac{\partial S}{\partial b} &= \sum_i -2(y_i - a - bx_i)x_i &&= 2\left(a\textstyle\sum_i x_i + b\sum_i x_i^2 - \sum_i x_1 y_i\right) = 0\end{aligned} \tag{7.6}$$

Dividing both equations by $2n$ leads to the equations

$$\begin{aligned}a + b\bar{x} &= \bar{y} \\ a\bar{x} + b\frac{1}{n}\sum_i x_i^2 &= \frac{1}{n}\sum_i x_i y_i\end{aligned} \tag{7.7}$$

where

$$\bar{x} = \frac{1}{n}\sum_i x_i\,, \quad \bar{y} = \frac{1}{n}\sum_i y_i\,. \tag{7.8}$$

Solving Eq. (7.7) for the fitting parameters gives

$$b = \frac{\sum_i x_i y_i - n\bar{x}\bar{y}}{\sum_i x_i^2 - n\bar{x}^2}\,, \quad a = \bar{y} - b\bar{x}\,. \tag{7.9}$$

Noting that $n\bar{y} = \sum_i y$ and $n\bar{x} = \sum_i x$, the results can be written as

$$b = \frac{\sum_i (x_i - \bar{x})y_i}{\sum_i (x_i - \bar{x})x_i}\,, \quad a = \bar{y} - b\bar{x}\,. \tag{7.10}$$

While Eqs. (7.9) and (7.10) are equivalent analytically, Eq. (7.10) is preferred for numerical calculations because Eq. (7.10) is less sensitive to roundoff errors. Here is a Python function implementing this algorithm:

Code: linefit.py

```python
def linefit(x, y):
    """Returns slope and y-intercept of linear fit to (x,y)
    data set"""
    xavg = x.mean()
    slope = (y * (x - xavg)).sum() / (x * (x - xavg)).sum()
    yint = y.mean() - slope * xavg
    return yint, slope
```

It's hard to imagine a simpler implementation of the linear regression algorithm.

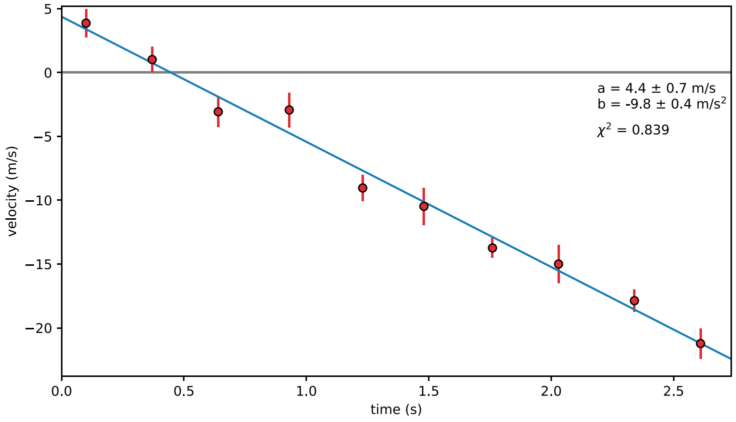


Figure 7.3 Fit using χ^2 least squares fitting routine with data weighted by error bars.

7.5.2 Linear Regression with Weighting: χ^2

The linear regression routine of the previous section weights all data points equally. That is fine if the absolute uncertainty is the same for all data points. In many cases, however, the uncertainty is different for different points in a data set. In such cases, we would like to weight the data with smaller uncertainty more heavily than those with greater uncertainty. For this case, there is a standard method of weighting and fitting data that is known as χ^2 (or *chi-squared*) fitting. In this method, we suppose that associated with each (x_i, y_i) data point is an uncertainty in the value of y_i of $\pm\sigma_i$. In this case, the "best fit" is defined as the one with the set of fitting parameters that minimizes the sum

$$\chi^2 = \sum_i \left(\frac{y_i - f(x_i)}{\sigma_i} \right)^2 . \tag{7.11}$$

Setting the uncertainties $\sigma_i = 1$ for all data points yields the same sum S we introduced in the previous section. In this case, all data points are weighted equally. However, if σ_i varies from point to point, it is clear that those points with large σ_i contribute less to the sum than those with small σ_i. Thus, data points with large σ_i are weighted less than those with small σ_i.

To fit data to a straight line, we set $f(x) = a + bx$ and write

$$\chi^2(a, b) = \sum_i \left(\frac{y_i - a - bx_i}{\sigma_i} \right)^2 . \tag{7.12}$$

Finding the minimum for $\chi^2(a, b)$ follows the same procedure used for finding the minimum of $S(a, b)$ in the previous section. The result is

$$b = \frac{\sum_i (x_i - \hat{x}) y_i / \sigma_i^2}{\sum_i (x_i - \hat{x}) x_i / \sigma_i^2} \,, \quad a = \hat{y} - b\hat{x} \,. \tag{7.13}$$

where

$$\hat{x} = \frac{\sum_i x_i / \sigma_i^2}{\sum_i 1/\sigma_i^2} \,, \quad \hat{y} = \frac{\sum_i y_i / \sigma_i^2}{\sum_i 1/\sigma_i^2} \,. \tag{7.14}$$

Figure 7.3 shows the fit to a straight line using this algorithm.

For a fit to a straight line, the overall quality of the fit can be measured by the reduced chi-squared parameter

$$\chi_r^2 = \frac{\chi^2}{n-2} \,, \tag{7.15}$$

where χ^2 is given by Eq. (7.11) evaluated at the optimal values of a and b given by Eq. (7.13). A good fit is characterized by $\chi_r^2 \approx 1$. This makes sense because if the uncertainties σ_i have been properly estimated, then $[y_i - f(x_i)]^2$ should, on average, be roughly equal to σ_i^2, so that the sum in Eq. (7.11) should consist of n terms (one for each of the n data point) approximately equal to 1. Of course, if there were only 2 data points ($n = 2$), then χ^2 would be zero as the best straight line fit to two points is a perfect fit. That is essentially why χ_r^2 is normalized using $n - 2$ instead of n. If χ_r^2 is significantly greater than 1, this indicates a poor fit to the fitting function (or an underestimation of the uncertainties σ_i). If χ^2 is significantly less than 1, then the uncertainties were probably overestimated (the fit and fitting function may or may not be good).

We can also get estimates of the uncertainties in our determination of the fitting parameters a and b, although deriving the formulas is a bit more involved than we want to get into here. Therefore, we just give the results:

$$\sigma_b^2 = \frac{1}{\sum_i (x_i - \hat{x}) x_i / \sigma_i^2} \,, \quad \sigma_a^2 = \sigma_b^2 \frac{\sum_i x_i^2 / \sigma_i^2}{\sum_i 1/\sigma_i^2} . \tag{7.16}$$

The estimates of uncertainties in the fitting parameters depend explicitly on $\{\sigma_i\}$ and will only be meaningful if (*i*) $\chi_r^2 \approx 1$ and (*ii*) the estimates of the uncertainties σ_i are accurate.

You can find more information, including a derivation of Eq. (7.16), in *Data Reduction and Error Analysis for the Physical Sciences, 3rd ed* by P. R. Bevington & D. K. Robinson, McGraw-Hill, New York, 2003.

7.6 EXERCISES

1. Write Python functions for the following:

   (a) A function named `sphere_area` that returns the area of a sphere given the sphere's diameter.

   (b) A function named `sphere_volume` that returns the volume of a sphere given the sphere's diameter.

   The output of both functions should be floats. Incorporate these two functions into a program that makes three arrays, one of sphere diameters ranging from one to ten with increments of one, and two more with the corresponding areas and volumes. Using these arrays, have your program print a table like the one below. Hint: Use the `zip` function and loop over its rows to print out your table.

```
diameter  area      volume
1      3.14      0.52
2     12.57      4.19
3     28.27     14.14
4     50.27     33.51
5     78.54     65.45
6    113.10    113.10
7    153.94    179.59
8    201.06    268.08
9    254.47    381.70
10    314.16    523.60
```

2. Write a function to estimate the price of a Manhattan apartment. The positional inputs should be the total area of the apartment in square feet and the floor (story) of the apartment. Assume that the cost per square foot of an apartment is $1000 with an additional cost of $12,000 to the total price for each floor above the second floor. The price per square foot and the additional cost per floor should be keyword arguments (set equal to 1000 and 12,000, respectively).

   Using your function, write a program to print out the total cost of an apartment on the 46th floor with a total area of 1200 sq. ft. and the total cost of an apartment on the 5th floor with a total area of 728 sq. ft.

   Extra challenge: Look up online how to print out the answers with commas as thousands separators. For good measure, put a dollar sign in front of the numbers.

3. Write a function that returns the first n spherical Bessel functions $j_n(x)$ up to $n = 2$:

$$j_0(x) = \frac{\sin x}{x}$$
$$j_1(x) = \frac{\sin x}{x^2} - \frac{\cos x}{x}$$
$$j_2(x) = \left(\frac{3}{x^2} - 1\right)\frac{\sin x}{x} - \frac{3\cos x}{x^2}$$

Your function should take as arguments a NumPy array x and the order n, and should return $j_0(x)$, or $j_0(x)$ and $j_1(x)$, or $j_0(x)$, $j_1(x)$, and $j_2(x)$, depending on whether n is 0, 1, or 2. Take care to make sure that your functions behave properly at $x = 0$.

Using your function, write a program to reproduce the following results:

```
x      j0       j1       j2
 0.0    1.000    0.000    0.000
 1.0    0.841    0.301    0.062
 2.0    0.455    0.435    0.198
 3.0    0.047    0.346    0.299
 4.0   -0.189    0.116    0.276
 5.0   -0.192   -0.095    0.135
 6.0   -0.047   -0.168   -0.037
 7.0    0.094   -0.094   -0.134
 8.0    0.124    0.034   -0.111
 9.0    0.046    0.106   -0.010
10.0   -0.054    0.078    0.078
```

Something to think about: You might note that $j_0(x)$ can be used in the calculation of $j_1(x)$, and that $j_1(x)$ can be used in the calculation of $j_2(x)$. Can you use this to write a more efficient function for the calculations of $j_1(x)$ and $j_2(x)$?

4. Write a function that simulates the rolling of n dice. You can generate a random integer between 1 and 6 with equal probability with the following code:

```
 In[1]                       In[2]: import random

 In[3]                       In[4]: random.randint(1, 6)
Out[4]                      Out[4]: 4
```

The function `random.randint(1, 6)` returns a random integer between 1 and 6 inclusive. For more about Python's module random, see https://docs.python.org/3/library/random.html.

The input of your function should be the number of dice thrown for each roll, and the output should be the sum of the n dice. Your function should print the result of each rolled die and return the sum of all the dice.

Write a pair of nested `for` loops to run your function three times for each value of n from 2 to 5. The output should look something like this:

```
number of dice = 2
[3, 6]
9
[1, 6]
7
[6, 5]
11
number of dice = 3
[3, 1, 2]
6
.
.
.
```

5. In Section 7.5, we showed that the best fit of a line $y = a + bx$ to a set of data $\{x_i, y_i\}$ is obtained for the values of a and b given by Eq. (7.10). Those formulas were obtained by finding the values of a and b that minimized the sum in Eq. (7.5). This approach and these formulas are valid when the uncertainties in the data are the same for all data points. The Python function `linefit(x, y)` in Section 7.5.1 implements Eq. (7.10).

   (a) Write a new fitting function `linefitWt(x, y, dy)` that implements the formulas given in Eqs. (7.13) and (7.14)) that minimize the χ^2 function give by Eq. (7.12). This more general approach is valid when the individual data points have different weightings *or* when they all have the same weighting; these are input to `linefitWt(x, y, dy)` with the new argument `dy`. In addition to returning the fitting parameters a and b, `linefitWt(x, y, dy)` should also return the uncertainties in the fitting parameters σ_a and σ_b using Eq. (7.16).

   You should also write a separate function to calculate the reduced chi-squared χ_r^2 defined by Eq. (7.15).

(b) Write a Python program that reads in the data below and fits it using the two fitting functions `linefit(x, y)` and `linefitWt(x, y, dy)`. Your program should report the results for both fits.

```
Velocity \vs\ time data for a falling mass
time (s) velocity (m/s) uncertainty (m/s)
 2.23         139              16
 4.78         123              16
 7.21         115               4
 9.37          96               9
11.64          62              17
14.23          54              17
16.55          10              12
18.70          -3              15
21.05         -13              18
23.21         -55              10
```

6. Consider the two functions

$$p(x) = a + bx + cx^2$$
$$V(t) = V_0\left(1 - e^{-t/RC}\right)$$

(a) Write a Python function for $p(x)$ and then use the functions `derivA` on page 137 and `derivK` on page 139 to numerically determine the derivatives of $p(x)$ at $x = -3, -2, -1, 0, 1, 2, 3$. `derivA` and `derivK` should give identical results. Use $a = 2.1$, $b = -3.2$, and $c = 5.8$ for the constants.

(b) Do the same for $V(t)$ for $t = 0.0, 0.2, 0.4, 0.6, 0.8, 1.0$. For the constants, use $V_0 = 5.0$, $R = 100,000$, and $C = 1 \times 10^{-6}$.

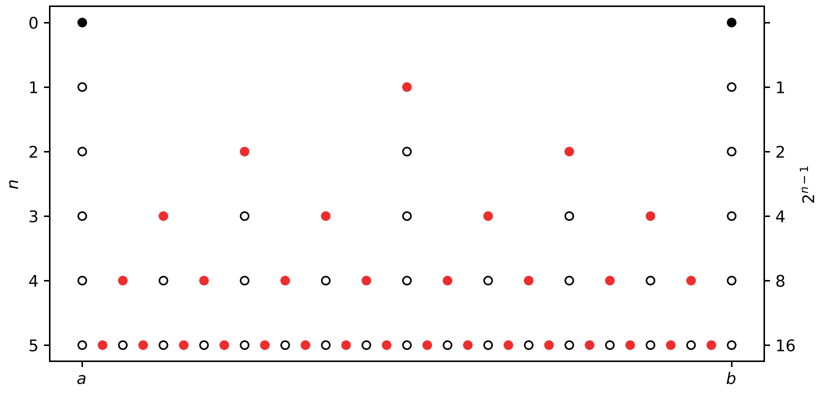
7. Write a function that numerically estimates the integral

$$A = \int_a^b f(x)\,dx$$

using the *trapezoid rule*. The simplest version of the trapezoid rule, which generally gives a very crude estimate, is

$$A_0 = \tfrac{1}{2}h_0[f(a) + f(b)]\ , \quad h_0 = b - a\ .$$

This estimate for the integral can be refined by dividing the interval from a to b in two and performing the trapezoid rule on each interval. The



refined estimate A_1 is given by

$$\begin{aligned} A_1 &= \tfrac{1}{2}h_1[f(a) + f(a+h_1)] + \tfrac{1}{2}h_1[f(a+h_1) + f(b)]\,, \quad h_1 = \tfrac{1}{2}h_0 \\ &= \tfrac{1}{2}h_1[f(a) + 2f(a+h_1) + f(b)] \\ &= \tfrac{1}{2}A_0 + h_1 f(a+h_1) \end{aligned}$$

Notice that the first estimate of the integral A_0 can be reused in the calculation of the refined estimate A_1, thereby saving some computational effort.

This process of dividing the intervals in two can be continued to obtain a sequence of ever more refined estimates $A_0, A_1, A_2, \ldots$of the integral. With each iteration, the result of the previous estimate can be reused in the calculation of the next more refined estimate. Repeatedly application of the trapezoid rule gives the following sequence of ever more refined estimates:

$$\begin{aligned} A_2 &= \tfrac{1}{2}A_1 + h_2[f(a+h_2) + f(a+3h_2)]\,, \quad h_2 = \tfrac{1}{2}h_1 \\ A_3 &= \tfrac{1}{2}A_2 + h_3[f(a+h_3) + f(a+3h_3) + f(a+5h_3) + f(a+7h_3)]\,, \\ &\qquad h_3 = \tfrac{1}{2}h_2 \\ &\vdots \\ A_n &= \tfrac{1}{2}A_{n-1} + h_n \sum_{i=1,3,\ldots}^{2^{n-1}} f(a+ih_n)\,, \quad h_n = \tfrac{1}{2}h_{n-1}\,, \text{ for } n \geq 1 \end{aligned}$$

This process can be repeated as many times as needed until the desired precision is obtained, which can be estimated by requiring that the fractional difference between successive estimates $|(A_i - A_{i-1})/A_i| < \epsilon$, where ϵ might be some small number like 10^{-8}.

Write a function that implements the trapezoid rule by first evaluating A_0, then A_1, ...until ϵ is less than some preset tolerance. By using the previous result A_{i-1} to calculate A_i, you only need to evaluate the function to be integrated $f(x)$ at the open circles in the preceding diagram, saving a great deal of computation.

Your function implementing the trapezoid rule should be of the form

```
trapz(f, a, b, tol=1.0e-8, **fparams)
```

where `f` is the function to be integrated from `a` to `b` to within a tolerance `tol` ($=\epsilon$). The function definition should include `**fparams` to pass any additional parameters of the function `f`.

Your function `trapz()` should return the following values:

- A_n: the sum that represents the numerical approximation of the integral.
- $\epsilon_n = |(A_n - A_{n-1})/A_n|$: the estimated error,which should be less than `tol`.
- n: the number of iterations required to achieve the desired accuracy for A_n.

A reasonably efficient function implementing the adaptive trapezoid method shouldn't be very long: on the order of 16 or fewer lines should suffice.

Try your trapezoid integration function on the following integrals and show that you get an answer within the specified tolerance of the exact value. Define one function for each of the three parts below and test each function for the two specific cases provided in each part. Use the `**{}` syntax to pass extra parameters of the integrand function.

(a) $\int_a^b cx^p\,dx$
 (i) $\int_2^5 x^2\,dx = 39$
 (ii) $\int_1^5 2x^3\,dx = 312$

(b) $\int_a^b A\sin kx\,dx$
 (i) $\int_0^\pi \sin x\,dx = 2$
 (ii) $\int_{\pi/2}^{2\pi} 7\sin\frac{x}{3}\,dx = \frac{21}{2}(1+\sqrt{3}) \simeq 28.686533479473$

(c) $\int_s^t Be^{-(x/w)^2}\,dx$
 (i) $\int_0^{3.5} e^{-x^2}\,dx = \frac{\sqrt{\pi}}{2}\mathrm{erf}(3.5) \simeq 0.8862262668989$
 (ii) $\int_1^5 4e^{-(x/2)^2}\,dx = 4\sqrt{\pi}\left[\mathrm{erf}\left(\frac{5}{2}\right) - \mathrm{erf}\left(\frac{1}{2}\right)\right] \simeq 3.3966821376379$

8. If an integer is divisible by 9, then the sum of its digits produces a number that is also divisible by 9. The same is true for integers divisible by 3.

   (a) Write a function called `add` that sums the digits of an integer. For example, summing the digits of the integer 3820488 gives 33. HINT: One way to do this is to use the `str` function.

   (b) Using a `while` loop, write another function that recursively calls `add` until you get a number that is 9 or smaller. Use this result to print out one of three outputs depending on which is true:
      - number is divisible by 3 and 9
      - number is divisible by 3 but not by 9
      - number is not divisible by 3 or 9

   (c) Typically, how many calls to `add` are needed for the routine you wrote in part (b)? for a number with 5 digits? 10 digits? 30 digits? (Recall that Avogadro's number is 6.022×10^{23} particles/mole so you are unlikely to run into meaningful integers too much longer than 30 digits.)

9. Write a function to determine if a year is a leap year. Leap years occur every year divisible by 4, except for years divisible by 100 but not by 400. For example, 1900 was not a leap year, but 2000 was a leap year. Write a program incorporating your function that correctly tells whether the following year years are leap years or not: 1900, 2000, 1964, 2046, 3000.

10. (a) Write a Python function `day_of_week(y, m, d)` that returns the day of the week for any given calendar date after January 1, 1700, which was a Friday. In the function `day_of_week(y, m, d)`, `y` is the year ($\geq$ 1700), `m` is the month (1–12), and `d` is the day (1–31...). Your function will need to consider leap years, which occur in every year divisible by 4, except for years divisible by 100 but not divisible by 400. For example, 1900 was not a leap year, but 2000 was a leap year. Therefore, your function should include another function that creates a list of all leap years between 1700 and the year of the date in question. Test that your program returns the answers tabulated below.

    (b) Write a function inside `day_of_week(y, m, d)` that returns a `ValueError` if the date input to the function is not valid. The text of the `ValueError` should indicate the nature of the error (year before 1700, too many days in month, *etc.*).

Date	**Weekday**
January 1, 1700	Friday
January 12, 1701	Wednesday
July 4, 1776	Thursday
April 30, 1777	Wednesday
January 28, 1813	Thursday
March 4, 1861	Monday
March 14, 1879	Friday
May 11, 1918	Saturday
July 20, 1969	Sunday
February 29, 1980	Friday
November 9, 1989	Thursday
January 20, 2009	Tuesday

CHAPTER 8

Plotting

This chapter introduces ***plotting*** *using the* ***Matplotlib*** *package. You learn how to make simple 2D x-y plots and fancy plots suitable for publication, complete with legends, annotations, logarithmic axes, subplots, and insets. You learn how to produce* ***Greek letters and mathematical symbols*** *using a powerful markup language called* ***LaTeX****. You also learn how to make various kinds of contour plots and vector field plots. 3D plotting with Matplotlib is introduced, although its 3D capabilities are somewhat limited. You learn about two interfaces for Matplotlib, a simple one known as* ***PyPlot*** *and a more powerful object-oriented interface. An overview of the object-oriented structure of Matplotlib is presented, including the important but sometimes confusing topic of* ***backends****.*

The graphical representation of data—plotting—is one of the most important tools for evaluating and understanding scientific data and theoretical predictions. Plotting is not a part of core Python but is provided through one of several different library modules. The most highly developed and widely used plotting package for Python is Matplotlib (http://matplotlib.sourceforge.net/). It is a powerful and flexible program that has become the *de facto* standard for 2D plotting with Python.

Because Matplotlib is an external library—in fact, a collection of libraries—it must be imported into any routine that uses it. Matplotlib makes extensive use of NumPy, so the two should be imported together. Therefore, for any program that is to produce 2D plots, you should include the lines

```
import numpy as np
import matplotlib.pyplot as plt
```

DOI: 10.1201/9781032673950-8

There are other Matplotlib sub-libraries, but the `pyplot` library provides nearly everything you need for 2D plotting. The standard prefix for it is `plt`. On some installations, `pyplot` is automatically loaded in the IPython shell, so you do not need to use `import matplotlib.pyplot` nor do you need to use the `plt` prefix when working in the IPython shell.[1] By contrast, you always need to explicitly import `pyplot` when writing a program or script.

One final word before getting started: This introduction only scratches the surface of what is possible using Matplotlib. As you become familiar with it, you will surely want to do more than this book describes. In that case, you need to consult the web for more information. An excellent place to start is http://matplotlib.org/index.html.

8.1 AN INTERACTIVE SESSION WITH PyPlot

Let's begin with an interactive plotting session that illustrates the basic features of Matplotlib. Working in the IPython console of Spyder, JupyterLab, Jupyter Notebook, or QtConsole, import NumPy and Matplotlib. Then enter the Matplotlib function call `plt.plot([1, 3, 2, 4, 3, 5])`. Take care to follow the exact syntax. Typing the magic command `%matplotlib qt` at the IPython console ensures that plots are generated in a separate interactive plot window and may be unnecessary depending on the settings of your IPython console.

```
 In[1]: import numpy as np

 In[2]: import matplotlib.pyplot as plt

 In[3]: %matplotlib qt

 In[4]: plt.plot([1, 3, 2, 4, 3, 5])
Out[4]: [<matplotlib.lines.Line2D at 0x1280eadd8>]

 In[5]: plt.show()
```

A window should appear with a plot that looks like the interactive plot window shown in Figure 8.1.[2] By default, the `plot` function draws a line between the data points that were entered. You can save this figure to an image file by clicking on the floppy disk (Save the figure) icon 🖫 in the border of

[1]This depends on the settings in your particular installation. You may want to set the Preferences for the IPython console to automatically load PyLab, which loads the NumPy and Matplotlib modules.

[2]Here, we have assumed that Matplotlib's *interactive mode* is turned off. If it's turned on, you don't need the `plt.show()` function when plotting from the command line of the IPython shell. Type `plt.ion()` at the IPython prompt to turn on interactive mode. Type `plt.ioff()` to turn it off. Whether or not you work with interactive mode turned on in IPython is a matter of taste.

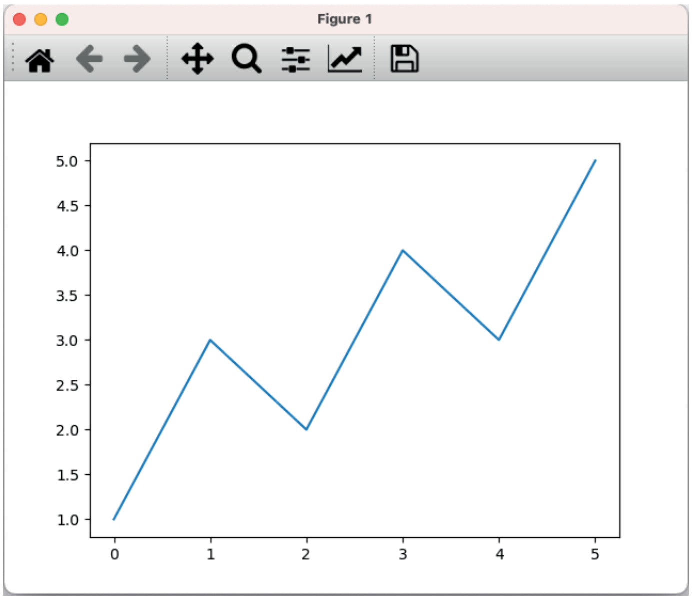


Figure 8.1 Interactive plot window.

the plot window. You can also zoom 🔍, pan ✥, scroll through the plot, and return to the original view 🏠 using other icons in the plot window. Experimenting with them reveals their functions. See page 183 for information about the configure subplots icon ☰. When you finish, close the plot window, which will return control to the IPython console.

By the way, executing the magic command `%matplotlib inline` instead of `%matplotlib qt` above causes the plot to be generated in the IPython console you are using. This can be convenient if you want to keep a record of your work in the IPython console. This is the default mode when working with Jupyter Notebooks. However, you sacrifice that ability to work interactively with your plot. On the other hand, you can toggle (switch back and forth) between the two modes using the above magic commands depending on how you want to display and interact with a plot.

Let's take a closer look at the `plot` function. It is used to plot *x*-*y* data sets and is written like this

```
plot(x, y)
```

where `x` and `y` are arrays (or lists) that have the same size. If the `x` array is omitted, that is, if there is only a single array, as in our example above, the `plot` function uses `0, 1, ..., N-1` for the `x` array, where `N` is the size of the `y` array. Thus, the `plot` function provides a quick graphical way of examining a data set.

More typically, you supply two arrays, `x` and a `y`, as a data set to plot. Taking things further, you may also want to plot several data sets on the same graph, use symbols as well as lines, label the axes, create a title and a legend, and control the color of symbols and lines. All of this is possible but requires calling several plotting functions. For this reason, plotting is best done using a Python script or program.

8.2 BASIC PLOTTING

The quickest way to learn how to plot using the Matplotlib library is by example. Let's start by plotting the sine function from 0 to 4π. The main plotting function `plot` in Matplotlib does not plot functions *per se*; it plots (x, y) data sets. As you will see, you can instruct the function `plot` either to just draw points—or dots—at each data point, or you can instruct it to draw straight lines between the data points. To create the illusion of the smooth function that the sine function is, you need to create enough (x, y) data points so that when `plot` draws straight lines between the data points, the function appears smooth. The sine function undergoes two complete oscillations with two maxima and two minima between 0 and 4π. So, let's start by creating an array with 33 data points between 0 and 4π, and then let Matplotlib draw a straight line between them. Our code consists of four parts:

- Import the NumPy and Matplotlib modules (lines 1-2 below).
- Create the (x, y) data arrays (lines 3-4 below).
- Have `plot` draw straight lines between the (x, y) data points (line 5 below).
- Display the plot in a figure window using the `show` function (line 6 below).

Here is our code, which consists of only six lines:

Code: sine_plot33.py

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 x = np.linspace(0, 4.0 * np.pi, 33)
4 y = np.sin(x)
5 plt.plot(x, y)
6 plt.show()
```

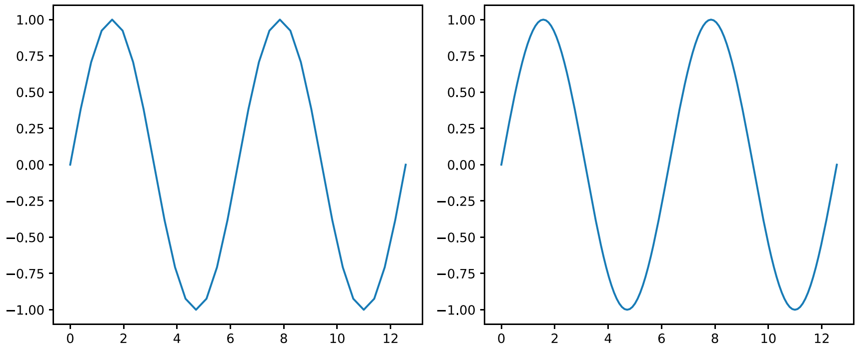


Figure 8.2 Sine function: LEFT, 33 data points; RIGHT, 129 data points.

Only six lines suffice to create the plot, which is shown on the left side of Figure 8.2. It consists of the sine function plotted over the interval from 0 to 4π, as advertised, and axes annotated with nice whole numbers over the appropriate interval. Matplotlib automatically scales the axes to include the plotted points. It's a pretty nice plot made with very little code.

One problem, however, is that while the plot oscillates like a sine wave, it is not smooth (look at the peaks). This is because we did not create the (x, y) arrays with enough data points. To correct this, we need more data points. The plot on the right side of Figure 8.2 was created using the same program shown above but with 129 (x, y) data points instead of 33. Try it out yourself by copying the above program and replacing 33 in line 3 with 129 (a few more or less is ok) so that the function `linspace` creates an array with 129 data points instead of 33.

The above script illustrates how plots can be made with very little code using the Matplotlib module. In making this plot, Matplotlib has made several choices, such as the size of the figure, the color of the line, and even the fact that by default a line is drawn between successive data points in the (x, y) arrays. All of these choices can be changed by explicitly instructing Matplotlib to do so. This involves including more arguments in our function calls and using new functions that control other plot properties. The next example illustrates a few of the simpler embellishments that are possible.

In Figure 8.3, we plot two (x, y) data sets: a data set of discrete data points read in from a data file, which are plotted as red circles, and a theoretical model, which is a smooth blue line. In this plot, we label the x and y axes, create a legend, and draw lines to indicate where x and y are zero. The code that creates Figure 8.3 is listed here:

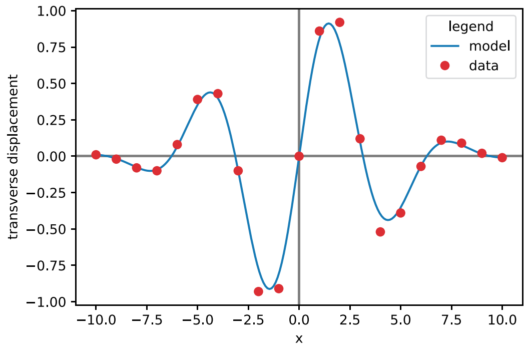


Figure 8.3 Wavy pulse.

Code: wavy_pulse.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # read data from file
5  xdata, ydata = np.loadtxt("wavy_pulse_data.txt", unpack=True)
6
7  # create x and y arrays for smooth curve
8  x = np.linspace(-10.0, 10.0, 200)
9  y = np.sin(x) * np.exp(-((x / 5.0) ** 2))
10
11 # create plot
12 plt.figure(1, figsize=(6, 4))
13
14 plt.plot(x, y, "-C0", label="model")  # blue line
15 plt.plot(xdata, ydata, "oC3", label="data")  # red circle
16 # label x & y axes
17 plt.xlabel("x")
18 plt.ylabel("transverse displacement")
19 # display legend using labels set in plot function calls
20 plt.legend(loc="upper right", title="legend")
21 # draw gray lines behind plotted data for y=0 and x=0
22 plt.axhline(color="gray", zorder=-1)
23 plt.axvline(color="gray", zorder=-1)
24
25 # save plot to file
26 plt.savefig("figures/wavy_pulse.pdf")
27
28 # display plot on screen
29 plt.show()
```

If you have read the first seven chapters, the code in lines 1–9 in the above script should be familiar. First, the script loads the NumPy and Matplotlib modules, then reads data from a data file into two arrays, `xdata` and `ydata`, and then creates two more arrays, `x` and `y`. The first pair or arrays, `xdata` and `ydata`, contain the x-y data that are plotted as blue circles in Figure 8.3; the arrays created in lines 8 and 9 contain the x-y data that are plotted as a blue line.

The functions that create the plot begin on line 12. Let's go through them one by one and see what they do. To follow and better understand what the various plotting calls do, we strongly recommend that you open up a Qt Console and enter the `wavy_pulse.py` program line by line, more-or-less, in particular lines 12–23. To do so, you will need the `wavy_pulse_data.txt`, which you can download from the GitHub site for this textbook. You will notice that *keyword arguments* (`kwargs`) are used in several cases.

`figure()` creates a blank figure window. If it has no arguments, it creates a window that is 6.4 × 4.8 inches (about 16.2 × 12.2 cm) by default, although the size that appears on your computer depends on your screen's resolution. For most computers, it will be somewhat smaller. You can create a window whose size differs from the default using the optional keyword argument `figsize`, as we have done here. If you use `figsize`, set it equal to a 2-element tuple where the elements, expressed in inches,[3] are the width and height, respectively, of the plot. Multiple calls to `figure()` opens multiple windows: `figure(1)` opens up one window for plotting, `figure(2)` another, and `figure(3)` yet another.

`plot(x, y, *optional arguments*)` graphs the x-y data in the arrays `x` and `y`. The third argument is a format string that specifies the color and the type of line or symbol used to plot the data. The string `'oC3'` specifies a red (`C3`) circle (`o`). The string `'-C0'` specifies a blue (`C0`) solid line (`-`). The keyword argument `label` is set equal to a string that labels the data for the `legend` function if it is called subsequently.

`xlabel(*string*)` takes a string argument that specifies the label for the graph's x-axis.

`ylabel(*string*)` takes a string argument that specifies the label for the graph's y-axis.

`legend()` makes a legend for the data plotted. Each x-y data set is labeled using the string supplied by the `label` keyword in the `plot` function that graphed the data set. The `loc` keyword argument specifies the location of the legend. The `title` keyword is used to give the legend a title.

[3] 1 inch = 2.54 cm

`axhline()` draws a horizontal line across the width of the plot at `y=0`. Writing `axhline(y=a)` draws a horizontal line at `y=a`, where `a` can be any numerical value. The optional keyword argument `color` is a string that specifies the line's color. The default color is black. The optional keyword argument `zorder` is an integer that specifies which plotting elements are in front of or behind others. By default, new plotting elements appear *on top of* previously plotted elements and have a value of `zorder=0`. By specifying `zorder=-1`, the horizontal line is plotted *behind* all existing plot elements that have not been assigned an explicit `zorder` less than -1. The keyword `zorder` can also be used as an argument for the `plot` function to specify the order of lines and symbols. Normally, symbols are placed on top of lines that pass through them.

`axvline()` draws a vertical line from the top to the bottom of the plot at `x=0`. See `axhline()` for an explanation of the arguments.

`savefig(`*`string`*`)` saves the figure to a file with a name specified by the string argument. The string argument can also contain path information to save the file somewhere other than the default directory. Here, we save the figure to a subdirectory named `figures` of the default directory. The extension of the filename determines the format of the figure file. The following formats are supported: png, pdf, ps, eps, and svg.

`show()` displays the plot on the computer screen. No screen output is produced before this function is called.

To plot the solid blue line, the code uses the `"-C0"` format specifier in the `plot` function call. It is important to understand that Matplotlib draws *straight lines* between data points. Therefore, the curve will appear smooth only if the data in the NumPy arrays are sufficiently dense. If the space between data points is too large, the straight lines the `plot` function draws between data points will be visible. For plotting a typical function, something on the order of 100–200 data points usually produces a smooth curve, depending on how curvy the function is. On the other hand, only two points are required to draw a smooth, straight line.

Detailed information about the Matplotlib plotting functions is available online. The main Matplotlib site is http://matplotlib.org/.

TABLE 8.1 Line and symbol type designations for plotting.

character	description	character	description
-	solid line style	3	tri_left marker
--	dashed line style	4	tri_right marker
-.	dash-dot line style	s	square marker
:	dotted line style	p	pentagon marker
.	point marker	*	star marker
,	pixel marker	h	hexagon1 marker
o	circle marker	H	hexagon2 marker
v	triangle_down marker	+	plus marker
^	triangle_up marker	x	x marker
<	triangle_left marker	D	diamond marker
>	triangle_right marker	d	thin_diamond marker
1	tri_down marker	|	vline marker
2	tri_up marker	_	hline marker

8.2.1 Specifying Line and Symbol Types and Colors

In the above example, we illustrated how to draw one line type (solid), one symbol type (circle), and two colors (blue and red). There are many more possibilities.

Table 8.1 shows the characters used to specify the line or symbol type that is used. If a line type is chosen, the lines are drawn between the data points. If a marker type is chosen, the marker is plotted at each data point.

Color is specified using the codes in Figure 8.4: single letters for primary colors and codes `C0`, `C2`, …, `C9` for a standard Matplotlib color palette of ten colors designed to be pleasing to the eye.

These format specifiers give rudimentary control of the plotting symbols and lines. Matplotlib provides much more precise control of the plotting symbol size, line types, and colors using optional keyword arguments instead of the plotting format strings introduced above. For example, the following command creates a plot of large yellow diamond symbols with orange edges connected by a green dashed line:

```
plt.plot(x, y, color='green', linestyle='dashed', marker='D',
         markerfacecolor='yellow', markersize=7,
         markeredgecolor='C1')
```

Try it out! Another useful keyword is `fillstyle`, with self-explanatory keywords `full` (the default), `left`, `right`, `bottom`, `top`, `none`. The online Matplotlib

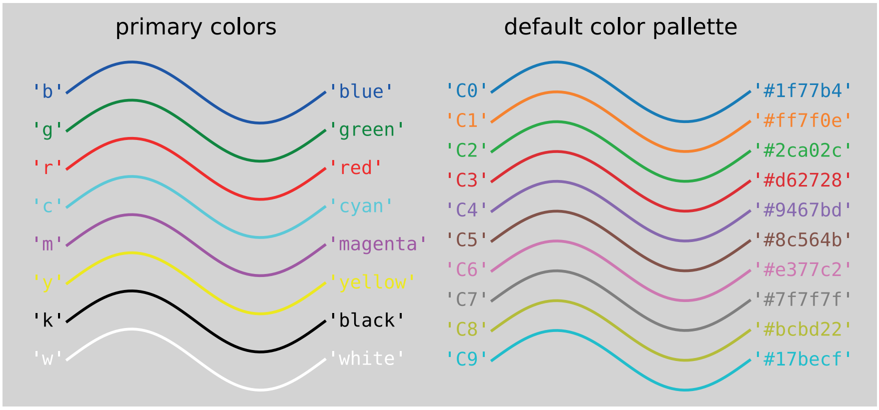


Figure 8.4 Some Matplotlib colors. The one- or two-letter strings to the left of each curve, or the longer strings to the right of each curve, can be used to specify a designated color. However, the shades of cyan, magenta, and yellow for the one-letter codes are different from the full-word codes (shown).

documentation provides all the plotting format keyword arguments and their possible values.

Specifying Line and Symbol Colors Using Seaborn

Seaborn is a Python data visualization library that augments Matplotlib in several ways, too many to detail here. One of its many convenient features is its function `color_palette()`, which, as its name implies, allows you to specify one of a large variety of color palettes.

The Seaborn library is imported with the statement

```
import seaborn as sns
```

with `sns` being the standard prefix abbreviation for Seaborn. The default Matplotlib palette is selected as follows:

```
c0 = sns.color_palette()  # no arguments gives default palette
```

where `color_palette()` returns a list of 3-element tuples that specify the sequence of colors: `c0[0]` is the `C0` (blue) color, `c0[1]` is the `C1` (orange) color, `c0[2]` is the `C2` (green) color, *etc.* The concave up parabolas above the x-axis in Figure 8.5 show the sequence of colors specified using the `color=c0[i]` keyword argument in `plot()` calls for `i=1,2,3,4`. Line 6 in the program listing `snscolors.py` below creates the `c0` list, which is subsequently used in `plot()` calls in lines 13 and 16.

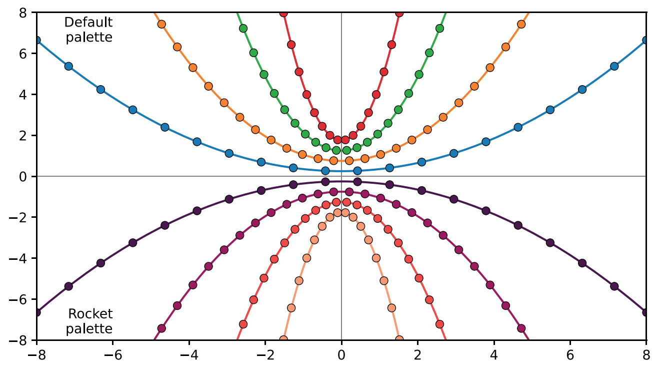


Figure 8.5 Seaborn color palettes.

Seaborn has a rich set of color palettes, at least 85 (depending on how you count, as there are several variations on each palette). You select a particular palette by specifying its name, for example, `"Set2"`, as shown here:

```
c = sns.color_palette("Set2")   # Seaborn "Set2" palette
```

where `c` is a list of 3-element tuples that specify the sequence of colors in the Seaborn `"Set2"` palette. The `"Set2"` is a discrete palette with eight colors, so `c` is an 8-element list. Other discrete palettes may have a different number of elements.

There are also continuous palettes, such as `"rocket"`. For plotting, you can select a discrete number of colors from a continuous palette as follows:

```
c1 = sns.color_palette("rocket", 4)
```

This selects four colors that more-or-less span Seaborn's `"rocket"` palette, avoiding the most extreme ends of palette. The concave down parabolas below the *x*-axis in Figure 8.5 show the sequence of colors from the `rocket` palette specified using the `color=c1[i]` keyword argument in `plot()` calls for `i=1,2,3,4`. Line 6 in the program listing `snscolors.py` below creates the `c1` list, which is subsequently used in `plot()` calls in lines 14 and 17.

The Seaborn library has a host of other useful enhancements to Matplotlib, which you are encouraged to explore after you become more familiar with Matplotlib.

Code: snscolors.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import seaborn as sns
4
5  c0 = sns.color_palette()   # default color palette (matplotlib)
```

```python
c1 = sns.color_palette("rocket", 4)  # a Seaborn palette

xl = np.linspace(-8.0, 8.0, 200)
plt.figure(figsize=(7, 4))
for i, curvature in enumerate([0.1, 0.3, 0.9, 2.7]):
    xp = np.linspace(-8.0, 8.0, 20 * (i + 1))
    yy = 0.25 + curvature * xl**2
    plt.plot(xl, yy + 0.5 * i, color=c0[i])
    plt.plot(xl, -yy - 0.5 * i, color=c1[i])
    yy = 0.25 + curvature * xp**2
    plt.plot(xp, yy + i / 2, "o", color=c0[i], mec="black", mew=0.5)
    plt.plot(xp, -yy - i / 2, "o", color=c1[i], mec="black", mew=0.5)

plt.axhline(color="gray", lw=0.5, zorder=-2)  # draw x-axis
plt.axvline(color="gray", lw=0.5, zorder=-2)  # draw y-axis
plt.xlim(-8.0, 8.0)
plt.ylim(-8.0, 8.0)
plt.text(-6.0, 7.8, "Default\n  palette", va="top", ha="right")
plt.text(-6.0, -7.8, "Rocket\npalette", va="bottom", ha="right")

plt.tight_layout()
plt.savefig("figures/snscolors.pdf")
plt.show()
```

8.2.2 Error Bars

When plotting experimental data it is customary to include error bars that indicate graphically the degree of uncertainty in the measurement of each data point. The Matplotlib function `errorbar` plots data with error bars attached. You can use it to either replace or augment the `plot` function. Both vertical and horizontal error bars can be displayed. Figure 8.6 illustrates the use of error bars.

When error bars are desired, you typically replace the `plot` function with the `errorbar` function. The first two arguments of the `errorbar` function are the `x` and `y` arrays to be plotted, just as for the `plot` function. The keyword `fmt` *must be used* to specify the format of the points to be plotted; the format specifiers are the same as for `plot`. The keywords `xerr` and `yerr` are used to specify the x and y error bars. Setting one or both to a constant specifies one size for all the error bars. Alternatively, setting one or both of them equal to an array with the same length as the `x` and `y` arrays allows you to give each data point an error bar with a different value. If you only want y error bars, you should specify only the `yerr` keyword and omit the `xerr` keyword. The color of the error bars is set with the keyword `ecolor`.

The code below illustrates how to make error bars and was used to make the plot in Figure 8.6. Lines 14 and 15 contain the call to the `errorbar` function. The x error bars are all set to a constant value of 1.25, meaning that the error

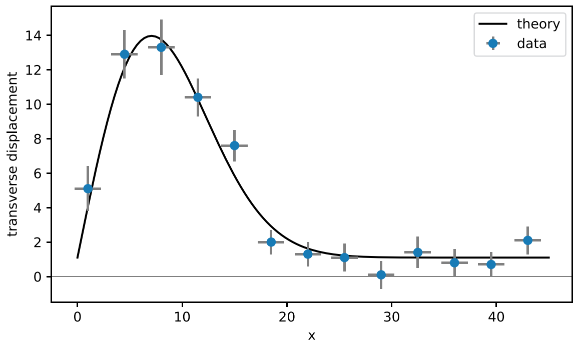


Figure 8.6 Data with error bars.

bars extend 1.25 to the left and 1.25 to the right of each data point. The *y* error bars are set equal to an array, which was read in from the data file containing the data, so each data point has a different *y* error bar. By the way, leaving out the `xerr` keyword argument in the `errorbar` function call below would mean that only the *y* error bars would be plotted.

Code: error_bar_plot.py

```python
import numpy as np
import matplotlib.pyplot as plt

# read data from file
xdata, ydata, yerror = np.loadtxt('expDecayData.txt', unpack=True)

# create theoretical fitting curve
x = np.linspace(0, 45, 128)
y = 1.1 + 3.0 * x * np.exp(-(x / 10.0) ** 2)

# create plot
plt.figure(1, figsize=(7, 4))
plt.plot(x, y, '-k', label="theory")
plt.errorbar(xdata, ydata, fmt='oC0', label="data",
             xerr=1.25, yerr=yerror, ecolor='gray')
plt.axhline(color="gray", linewidth=0.5)  # draws line at y=0
plt.xlabel('x')
plt.ylabel('transverse displacement')
plt.legend(loc='upper right')

# save plot to file
plt.savefig('figures/ExpDecay.pdf')

# display plot on screen
plt.show()
```

We have more to say about the `errorbar` function in the sections on logarithmic plots.

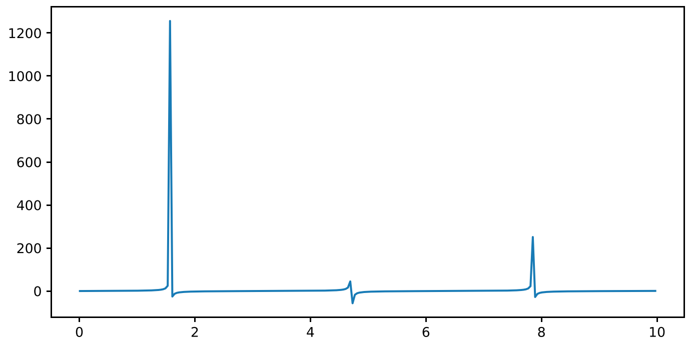


Figure 8.7 Trial tangent plot.

8.2.3 Setting Plotting Limits and Excluding Data

You often want to restrict the range of numerical values over which you plot data or functions. In these cases, you may need to manually specify the plotting window; alternatively, you may wish to exclude data points outside some set of limits. Here, we demonstrate methods for doing this.

8.2.3.1 Setting Plotting Limits

Suppose you want to plot the tangent function over the interval from 0 to 10. The following script offers a straightforward first attempt.

Code: tan_plot0.py

```python
import numpy as np
import matplotlib.pyplot as plt

theta = np.arange(0.01, 10., 0.04)
ytan = np.tan(theta)

plt.figure(figsize=(8.5, 4.2))
plt.plot(theta, ytan)
plt.savefig("figures/tanPlot0.pdf")
plt.show()
```

The resulting plot, shown in Figure 8.7, doesn't quite look like what you might have expected for $\tan\theta$ *vs.* θ. The problem is that $\tan\theta$ diverges at $\theta = \pi/2, 3\pi/2, 5\pi/2, ...$, which leads to large spikes in the plots as values in the `theta` array come near those values. Of course, we don't want the plot to extend out to $\pm\infty$ in the y direction, nor can it. Instead, we would like the plot to extend far enough that we get the idea of what is going on as $y \to \pm\infty$, but we would still like to see the behavior of the graph near $y = 0$. We can restrict

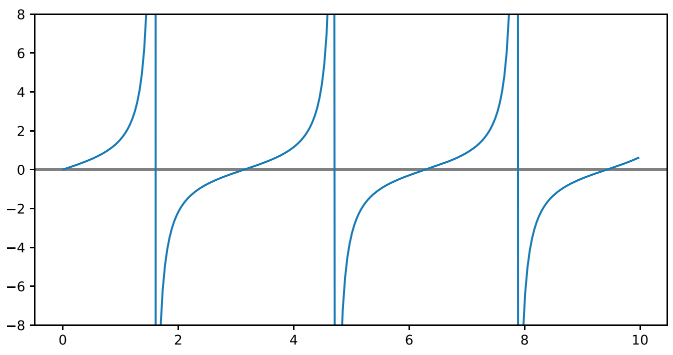


Figure 8.8 Tangent function (with spurious vertical lines).

the range of `ytan` values that are plotted using the Matplotlib function `ylim`, as we demonstrate in the script below.

Code: tan_plot1.py

```
import numpy as np
import matplotlib.pyplot as plt

theta = np.arange(0.01, 10., 0.04)
ytan = np.tan(theta)

plt.figure(figsize=(8.5, 4.2))
plt.plot(theta, ytan)
plt.ylim(-8, 8)  # restricts range of y axis from -8 to +8
plt.axhline(color="gray", zorder=-1)
plt.savefig("figures/tanPlot1.pdf")
plt.show()
```

Figure 8.8 shows the plot produced by this script, which now looks much more like the familiar $\tan\theta$ function we all know. We have also included a call to the `axline` function to create an x axis.

Recall that for $\theta = \pi/2$, $\tan\theta^- \to +\infty$ and $\tan\theta^+ \to -\infty$; in fact, $\tan\theta$ diverges to $\pm\infty$ at every odd half integral value of θ. Therefore, the vertical blue lines at $\theta = \pi/2, 3\pi/2, 5\pi/2$ should not appear in a proper plot of $\tan\theta$ *vs.* θ. However, they do appear because the `plot` function simply draws lines between the data points in the x-y arrays provided the `plot` function's arguments. Thus, plot draws a line between the very large positive and negative ytan values corresponding to the theta values on either side of $\pi/2$, where tan θ diverges to $\pm\infty$. It would be nice to exclude that line.

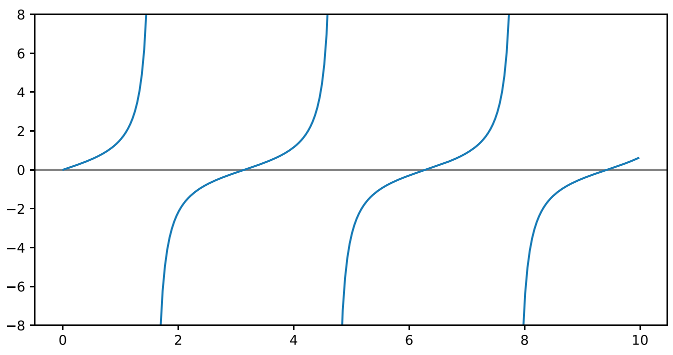


Figure 8.9 Tangent function (without spurious vertical lines).

8.2.3.2 Masked Arrays

You can exclude the data points near $\theta = \pi/2$, $3\pi/2$, and $5\pi/2$ in the above plot, and thus avoid drawing the nearly vertical lines at those points, using NumPy's *masked array* feature. The code below shows how to do this and produces Figure 8.10. The masked array feature is implemented in line 6 with a call to NumPy's `masked_where` function in the sub-module `ma` (masked array). It is called by writing `np.ma.masked_where`. The `masked_where` function works as follows: The first argument sets the condition for masking elements of the array; the second argument specifies the array. In this case, the function says to mask all elements of the array `ytan` (the second argument), where the absolute value of `ytan` is greater than 20. The result is set equal to `ytanM`. When `ytanM` is plotted, Matplotlib's `plot` function omits all masked points from the plot. You can think of it as the `plot` function lifting the pen that draws the line in the plot when it comes to the masked points in the array `ytanM`.

Code: tan_plot_masked.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  
4  theta = np.arange(0.01, 10., 0.04)
5  ytan = np.tan(theta)
6  ytanM = np.ma.masked_where(np.abs(ytan) > 20., ytan)
7  
8  plt.figure(figsize=(8.5, 4.2))
9  plt.plot(theta, ytanM)
10 plt.ylim(-8, 8)  # restricts y-axis range from -8 to +8
11 plt.axhline(color="gray", zorder=-1)
12 plt.savefig("figures/tanPlotMasked.pdf")
13 plt.show()
```

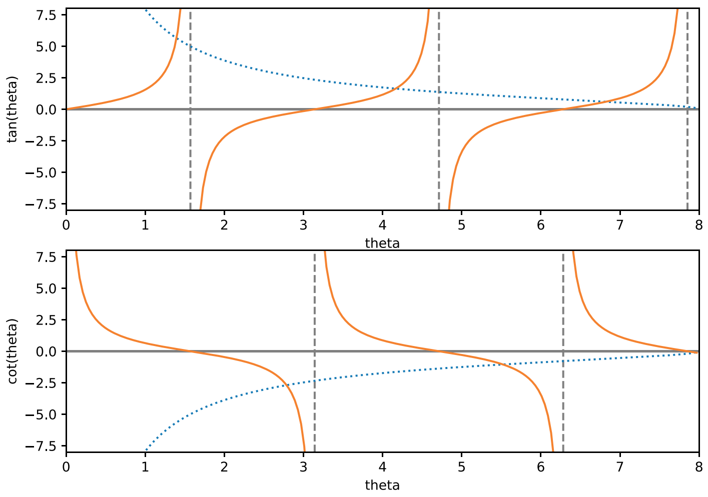


Figure 8.10 Plotting window with two subplots.

8.2.4 Subplots

Often, you want to create two or more graphs and place them next to one another, generally because they are related in some way. Figure 8.10 shows an example of such a plot. In the top graph, $\tan\theta$ and $\sqrt{(8/\theta)^2 - 1}$ *vs.* θ are plotted. The two curves cross each other at the points where $\tan\theta = \sqrt{(8/\theta)^2 - 1}$. In the bottom $\cot\,\theta$ and $-\sqrt{(8/\theta)^2 - 1}$ *vs.* θ are plotted. These two curves cross each other at the points where $\cot\theta = -\sqrt{(8/\theta)^2 - 1}$.

The code that produces this plot is provided below.

Code: subplot_demo.py

```
import numpy as np
import matplotlib.pyplot as plt

theta = np.arange(0.01, 8.0, 0.04)
y = np.sqrt((8.0 / theta) ** 2 - 1.0)
ytan = np.tan(theta)
ytan = np.ma.masked_where(np.abs(ytan) > 20.0, ytan)
ycot = 1.0 / np.tan(theta)
ycot = np.ma.masked_where(np.abs(ycot) > 20.0, ycot)

plt.figure(figsize=(8.5, 6))

plt.subplot(2, 1, 1)
plt.plot(theta, y, linestyle=":")
plt.plot(theta, ytan)
plt.xlim(0, 8)
plt.ylim(-8, 8)
```

```
18 plt.axhline(color="gray", zorder=-1)
19 plt.axvline(x=0.5 * np.pi, color="gray", linestyle="--", zorder=-1)
20 plt.axvline(x=1.5 * np.pi, color="gray", linestyle="--", zorder=-1)
21 plt.axvline(x=2.5 * np.pi, color="gray", linestyle="--", zorder=-1)
22 plt.xlabel("theta")
23 plt.ylabel("tan(theta)")
24 
25 plt.subplot(2, 1, 2)
26 plt.plot(theta, -y, linestyle=":")
27 plt.plot(theta, ycot)
28 plt.xlim(0, 8)
29 plt.ylim(-8, 8)
30 plt.axhline(color="gray", zorder=-1)
31 plt.axvline(x=np.pi, color="gray", linestyle="--", zorder=-1)
32 plt.axvline(x=2.0 * np.pi, color="gray", linestyle="--", zorder=-1)
33 plt.xlabel("theta")
34 plt.ylabel("cot(theta)")
35 
36 plt.savefig("figures/subplot_demo.pdf")
37 plt.show()
```

The function `subplot`, called on lines 13 and 25, creates the two subplots in the above figure. `subplot` has three arguments. The first specifies the number of rows into which the figure window will be divided: in line 13, it's 2. The second specifies the number of columns into which the figure window will be divided; in line 13, it's 1. The third argument specifies which rectangle will contain the plot specified by the following function calls. Line 13 specifies that the plotting commands that follow will act on the first box. Line 25 specifies that the plotting commands that follow will act on the second box. As a convenience, the commas separating the three arguments in the `subplot` routine can be omitted, provided they are all single-digit arguments (less than or equal to 9). For example, lines 13 and 25 can be written as

```
plt.subplot(211)
.
.
plt.subplot(212)
```

Finally, we labeled the axes and included dashed vertical lines at the values of θ where $\tan\theta$ and $\cot\theta$ diverge.

8.3 LOGARITHMIC PLOTS

Data sets can span many orders of magnitude, from fractional quantities much smaller than unity to values much larger than unity. In such cases, plotting the data on logarithmic axes is often useful.

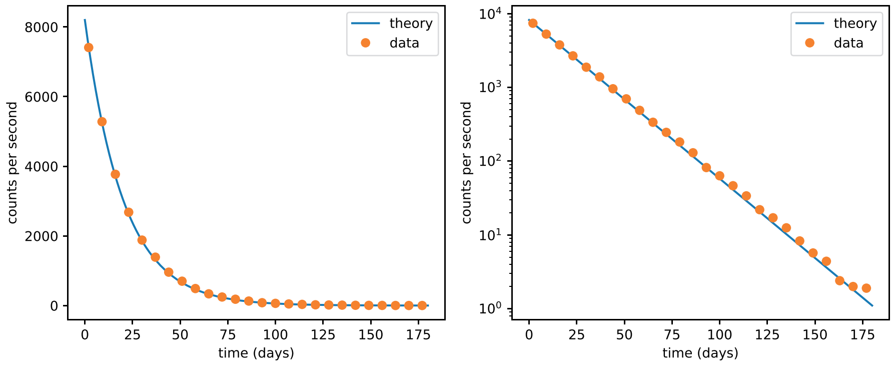


Figure 8.11 Semi-log plotting.

8.3.1 Semi-Log Plots

For data sets that vary exponentially in the independent variable, using one or more logarithmic axes is often useful. Radioactive decay of unstable nuclei, for example, exhibits an exponential decrease in the number of particles emitted from the nuclei as a function of time. Figure 8.11, for example, shows the decay of the radioactive isotope Phosphorus-32 over six months, where the radioactivity is measured once each week. The decay rate starts at nearly 10^4 electrons (counts) per second and diminishes to only about one count per second after about six months or 180 days. If we plot counts per second as a function of time on a standard plot, as we have done in the left panel of Figure 8.11, then the count rate is indistinguishable from zero after about 100 days. On the other hand, if we use a logarithmic axis for the count rate, as we have done in the right panel of Figure 8.11, then we can follow the count rate well past 100 days and can readily distinguish it from zero. Moreover, if the data vary exponentially in time, they will fall along a straight line, as they do in the case of radioactive decay.

Matplotlib provides two functions for making semi-logarithmic plots, `semilogx` and `semilogy`, for creating plots with logarithmic x and y axes, with linear y and x axes, respectively. We illustrate their use in the program below, which made the above plots.

Code: semilog_demo.py

```python
import numpy as np
import matplotlib.pyplot as plt

# read data from file
time, counts, unc = np.loadtxt("semilog_demo.txt", unpack=True)
```

```
6
7  # create theoretical fitting curve
8  t_half = 14.0  # P-32 half life = 14 days
9  tau = t_half / np.log(2)  # exponential tau
10 N0 = 8200.0  # Initial count rate (per sec)
11 t = np.linspace(0, 180, 128)
12 N = N0 * np.exp(-t / tau)
13
14 # create plot
15 plt.figure(1, figsize=(9.5, 4))
16
17 plt.subplot(1, 2, 1)
18 plt.plot(t, N, color="C0", label="theory")
19 plt.plot(time, counts, "oC1", label="data")
20 plt.xlabel("time (days)")
21 plt.ylabel("counts per second")
22 plt.legend(loc="upper right")
23
24 plt.subplot(1, 2, 2)
25 plt.semilogy(t, N, color="C0", label="theory")
26 plt.semilogy(time, counts, "oC1", label="data")
27 plt.xlabel("time (days)")
28 plt.ylabel("counts per second")
29 plt.legend(loc="upper right")
30
31 plt.tight_layout()
32
33 plt.savefig("figures/semilog_demo.pdf")
34 plt.show()
```

The `semilogx` and `semilogy` functions work like the `plot` function. You use one or the other depending on which axis you want to be logarithmic.

8.3.1.1 Adjusting Spacing Around Subplots

You may have noticed the `tight_layout()` function, called without arguments on line 31 of the program. This convenience function adjusts the space around the subplots to make room for the axes labels. If it is not called, the *y*-axis label of the right plot runs into the left plot. In Figure 8.10, you can see the consequence of not using the `tight_layout()` function, where the lower plot runs into the *x*-axis label of the upper plot. The `tight_layout()` function can also be useful in graphics windows with only one plot sometimes.

If you want direct control over how much space is allocated around subplots, use the function

```
plt.subplots_adjust(left=None, bottom=None, right=None,
                    top=None, wspace=None, hspace=None)
```

The keyword arguments `wspace` and `hspace` control the width and height of the space between plots, while the other arguments control the space to the left, bottom, right, and top. You can see and adjust the parameters of the `subplots_adjust()` routine by clicking on the configure subplots icon in the figure window. Once you have adjusted the parameters to obtain the desired effect, you can use them in your script.

8.3.2 Log-Log Plots

Matplotlib can also make log-log or double-logarithmic plots using the function `loglog`. It is useful when both the x and y data span many orders of magnitude. Data described by a power law $y = Ax^b$, where A and b are constants, appear as straight lines when plotted on a log-log plot. Again, the `loglog` function works just like the `plot` function but with logarithmic axes.

In the next section, we describe a more advanced syntax for creating plots and, with it, an alternative syntax for making logarithmic axes. For a description, see page 186.

8.4 MORE ADVANCED GRAPHICAL OUTPUT

The plotting methods introduced in the previous sections are adequate for basic plotting but are recommended only for the simplest graphical output. Here, we introduce a more advanced syntax that harnesses the full power of Matplotlib. It gives you more options and greater control.

An efficient way to learn this new syntax is simply to look at an example. Figure 8.12, which shows multiple plots laid out in the same window, is produced by the following code:

Code: multiple_plots1window.py

```python
import numpy as np
import matplotlib.pyplot as plt


# Define the sinc function, with output for x=0
# defined as a special case to avoid division by zero
def sinc(x):
    a = np.where(x == 0.0, 1.0, np.sin(x) / x)
    return a


x = np.arange(0.0, 10.0, 0.1)
y = np.exp(x)

t = np.linspace(-15.0, 15.0, 150)
z = sinc(t)

# create a figure window
fig = plt.figure(figsize=(9, 7))

# subplot: linear plot of exponential
```

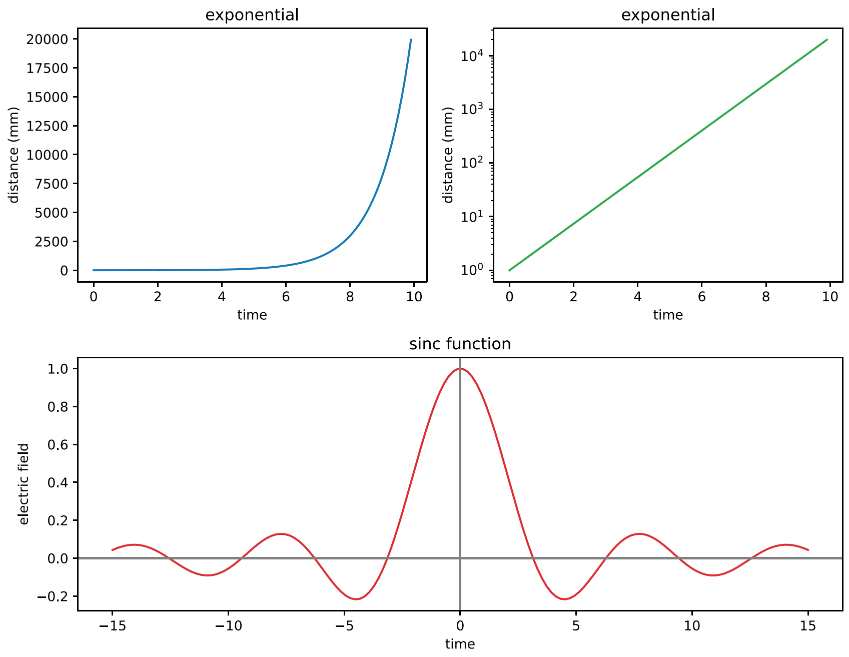


Figure 8.12 Mulitple plots in the same window.

```
ax1 = fig.add_subplot(2, 2, 1)
ax1.plot(x, y, "C0")
ax1.set_ylabel("distance (mm)")
ax1.set_title("exponential")

# subplot: semi-log plot of exponential
ax2 = fig.add_subplot(2, 2, 2)
ax2.plot(x, y, "C2")
ax2.set_yscale("log")
# ax2.semilogy(x, y, 'C2')  # same as 2 previous lines
ax2.set_ylabel("distance (mm)")
ax2.set_title("exponential")

# subplot: wide subplot of sinc function
ax3 = fig.add_subplot(2, 1, 2)
ax3.plot(t, z, "C3")
ax3.axhline(color="gray")
ax3.axvline(color="gray")
ax3.set_ylabel("electric field")
ax3.set_title("sinc function")

for xl in [ax1, ax2, ax3]:
    xl.set_xlabel("time")

# fig.tight_layout() adjusts white space to
# avoid collisions between subplots
fig.tight_layout()
fig.savefig("figures/multiple_plots1window.pdf")
plt.show()
```

After defining several arrays for plotting, the above program opens a figure window in line 19 with the statement

```
fig = plt.figure(figsize=(9, 7))
```

The Matplotlib statement above creates a **Figure** object, assigns it the name `fig`, and opens a blank figure window. We can use the `figure` function to open up multiple figure objects, each opening up a different blank figure window. The statements

```
fig1 = plt.figure()
fig2 = plt.figure()
```

open up two separate windows, one named `fig1` and the other `fig2`. We can then use the names `fig1` and `fig2` to create plot in either window. The `figure` function need not take any arguments if you are satisfied with the default settings, such as the figure size and the background color. On the other hand, by supplying one or more keyword arguments, you can customize the figure size, the background color, and several other properties. For example, in the program listing (line 19), the keyword argument `figsize` sets the width and height of the figure window. The default size is `(8, 6)`; in our program we set it to `(9, 7)`, which is a bit wider and higher. In the example above, we open only a single window, hence the single `figure` call.

The `fig.add_subplot(2, 2, 1)` in line 22 is a Matplotlib function that divides the figure window into 2 rows (the first argument) and 2 columns (the second argument). The third argument creates a subplot in the first of the 4 subregions (*i.e.*, of the 2 rows $\times$ 2 columns) created by the `fig.add_subplot(2, 2, 1)` call. To see how this works, type the following code into a Python module and run it:

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(9, 8))
ax1 = fig.add_subplot(2,2,1)

fig.show()
```

These commands open a figure window with axes drawn in the upper left quadrant. The `fig` prefix used with the `add_subplot(2, 2, 1)` function directs Python to draw these axes in the figure window named `fig`. If we had opened two figure windows, changing the prefix to correspond to the name of the other figure window would direct the axes to be drawn there. Writing `ax1 = fig.add_subplot(2, 2, 1)` assigns the name `ax1` to the axes in the upper left quadrant of the figure window.

The `ax1.plot(x, y, 'C0')` in line 23 directs Python to plot the previously defined `x` and `y` arrays onto the axes named `ax1`. The statement `ax2 = fig.add_subplot(2, 2, 2)` draws axes in the second, or upper right, quadrant of the figure window. The statement `ax3 = fig.add_subplot(2, 1, 2)` divides the figure window into two rows (first argument) and 1 column (second argument), creates axes in the second or these two sections, and assigns those axes (*i.e.*, that subplot) the name `ax3`. It divides the figure window into two halves, top and bottom, and then draws axes in the half number 2 (the third argument), the lower half of the figure window.

You may have noticed in the above code that some of the function calls are a bit different from those used before, so:

`xlabel('time (ms)')` $\longrightarrow$ `set_xlabel('time (ms)')`

`title('exponential')` $\longrightarrow$ `set_title('exponential')`

etc.

The call `ax2.set_yscale('log')` sets the y-axes in the second plot to be logarithmic, thus creating a semi-log plot. Alternatively, you can also do this with a `ax2.semilogy(x, y, 'C2')` call.

Using the prefixes `ax1`, `ax2`, or `ax3`, directs graphical instructions to their respective subplots. By creating and specifying names for the different figure windows and subplots within them, you access the different plot windows more efficiently. For example, the following code makes four identical subplots in a single figure window using a `for` loop (see Section 6.2.2):

```
In[1]: fig = figure()

In[2]: ax1 = fig.add_subplot(221)

In[3]: ax2 = fig.add_subplot(222)

In[4]: ax3 = fig.add_subplot(223)

In[5]: ax4 = fig.add_subplot(224)

In[6]: for ax in [ax1, ax2, ax3, ax4]:
  ...:     ax.plot([3,5,8],[6,3,1])

In[7]: fig.show()
```

8.4.1 An Alternative Syntax for a Grid of Plots

The syntax introduced above for defining a `Figure` window and opening *a grid* of several subplots can be a bit cumbersome, so an alternative, more compact syntax was developed. We illustrate its use with the program below:

Code: multiple_plots_grid.py

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2 * np.pi, 2 * np.pi, 200)
sin, cos, tan = np.sin(x), np.cos(x), np.tan(x)
csc, sec, cot = 1.0 / sin, 1.0 / cos, 1.0 / tan

fig, ax = plt.subplots(2, 3, figsize=(9.5, 6), sharex=True,
                       sharey=True)
ax[0, 0].plot(x, sin, color="red")
ax[0, 1].plot(x, cos, color="orange")
ax[0, 2].plot(x, np.ma.masked_where(np.abs(tan) > 20.0, tan),
              color="yellow")
ax[1, 0].plot(x, np.ma.masked_where(np.abs(csc) > 20.0, csc),
              color="green")
ax[1, 1].plot(x, np.ma.masked_where(np.abs(sec) > 20.0, sec),
              color="blue")
ax[1, 2].plot(x, np.ma.masked_where(np.abs(cot) > 20.0, cot),
              color="violet")
ax[0, 0].set_xlim(-2 * np.pi, 2 * np.pi)
ax[0, 0].set_ylim(-5, 5)
ax[0, 0].set_xticks(np.pi * np.array([-2, -1, 0, 1, 2]))
ax[0, 0].set_xticklabels([r"-2$\pi$", r"-$\pi$", "0", r"$\pi$",
                          r"2$\pi$"])
ax[0, 2].patch.set_facecolor("lightgray")

ylab = [["sin", "cos", "tan"], ["csc", "sec", "cot"]]
for i in range(2):
    for j in range(3):
        ax[i, j].axhline(color="gray", zorder=-1)
        ax[i, j].set_ylabel(ylab[i][j])

fig.savefig("figures/multiple_plots_grid.pdf")
plt.show()
```

This program generates a 2-row × 3-column grid of plots, as shown in Figure 8.13, using the function `subplots`. The first two arguments of `subplots` specify, respectively, the number of rows and columns in the plot grid. The other arguments are optional; we will return to them after discussing the output of the function `subplots`.

The output of `subplots` is a two-element list, which we name `fig` and `ax`. The first element `fig` is the name given to the figure object that contains all of the subplots. The second element `ax` is the name given to a 2 × 3 list of axes

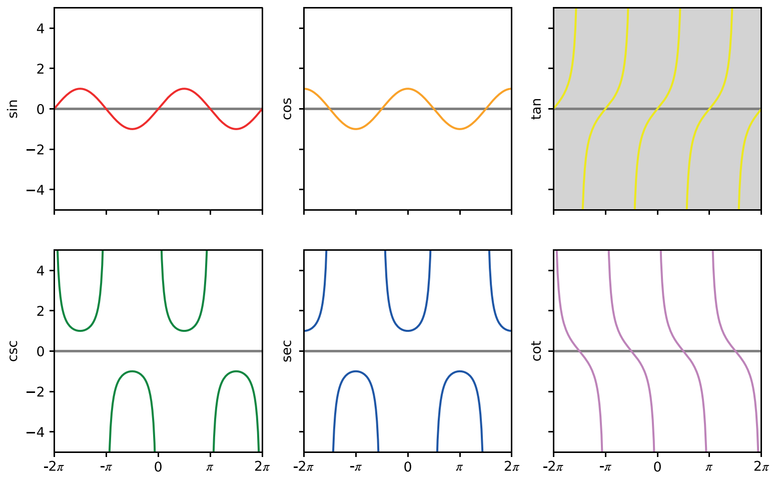


Figure 8.13 Grid of plots.

objects, one entry for each subplot. These subplots are indexed as you might expect: `ax[0, 0]`, `ax[0, 1]`, `ax[0, 2]`, ...

Returning to the arguments of `subplots`, the first keyword argument `figsize` sets the overall size of the figure window. The next keyword argument, `sharex=True`, instructs Matplotlib to create identical *x* axes for all six subplots; `sharey=True` does the same for the *y* axes. Thus, in lines 20 and 21, when we set the limits for the *x* and *y* axes for only the first subplot, `[0, 0]`, these instructions are applied to all six subplots because the keyword arguments instruct Matplotlib to make all the *x* axes the same and all the *y* axes the same. This also applies even to the tick placement and labels set in lines 23 and 24.

You may have noticed in lines 23 and 24 that Matplotlib can print Greek letters, in this case the letter π. Indeed, Matplotlib can output the Greek alphabet and virtually any mathematical equations you can imagine using the LaTeX typesetting system. The LaTeX string is enclosed by $ symbols, which are inside of quotes (double or single) because it's a string. The LaTeX capabilities of Matplotlib are discussed in Section 8.7.

By contrast, the subplot background is set to `'lightgray'` only in plot `[0, 2]` in line 25.

The nested `for` loops in lines 28–31 place a gray line at $y = 0$ and labels the *y*-axis in each subplot.

Finally, we note that writing

```
fig, ax = plt.subplots()
```

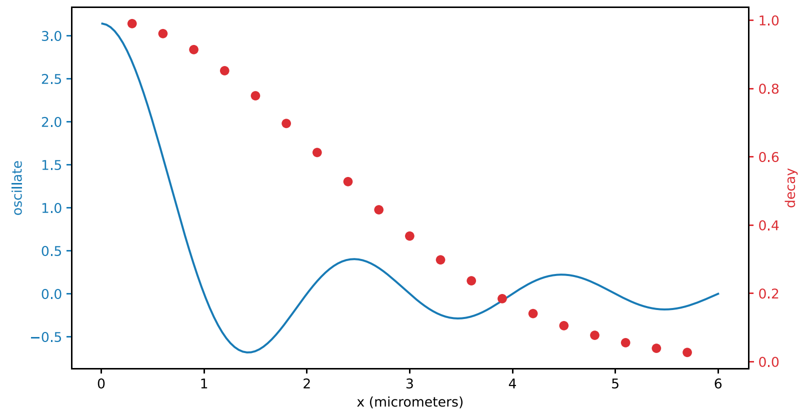


Figure 8.14 Figure with two *y* axes.

without any arguments in the `subplots` function opens a figure window with a single subplot. It is equivalent to

```
fig = plt.figure()
ax = fig.add_subplot(111)
```

It's a handy way to save a line of code.

8.5 PLOTS WITH MULTIPLE AXES

8.5.1 Plotting Quantities that Share One Axis but not the Other

Plotting two different quantities that share a common independent variable on the same graph can be a compelling way to compare and visualize data. Figure 8.14 shows an example of such a plot, where the blue curve is linked to the left blue *y*-axis and the red data points are linked to the right red *y*-axis. The code below shows how this is done with Matplotlib using the function `twinx()`.

Code: two_axes.py

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax1 = plt.subplots(figsize=(8.5, 4.5))
xa = np.linspace(0.01, 6.0, 150)
ya = np.sin(np.pi * xa) / xa
ax1.plot(xa, ya, "-C0")
ax1.set_xlabel("x (micrometers)")
# Make y-axis label, ticks and numbers match line color.
ax1.set_ylabel("oscillate", color="C0")
ax1.tick_params("y", colors="C0")
```

```

ax2 = ax1.twinx()  # use same x-axis for a 2nd (right) y-axis
xb = np.arange(0.3, 6.0, 0.3)
yb = np.exp(-xb * xb / 9.0)
ax2.plot(xb, yb, "oC3")
ax2.set_ylabel("decay", color="C3")  # axis label
ax2.tick_params("y", colors="C3")  # ticks & numbers

fig.tight_layout()
fig.savefig("figures/two_axes.pdf")
plt.show()
```

After plotting the first set of data using the axes `ax1`, calling `twinx()` instructs Matplotlib to use the same x-axis for a second x-y set of data, which we set up with a new set of axes `ax2`. The `set_ylabel` and `tick_parameters` functions are used to harmonize the colors of the y-axes with the different data sets.

There is an equivalent function `twiny()`, illustrated in the next section, that allows two sets of data to share a common y-axis and then have separate (top and bottom) x-axes.

8.5.2 Two Separate Scales for a Data Set

Sometimes plotting the same data set (or sets) on two different but related scales can be helpful. For example, infrared spectroscopic data are often plotted as a function of wavenumber $\tilde{\nu}$, which is defined as the inverse wavelength $\tilde{\nu} = 1/\lambda$, usually expressed in inverse centimeters. But you might also like to see the wavelength at a given wavenumber without having to calculate the reciprocal wavenumber mentally. In this case, we can provide a second x-axis at the top of the plot that displays the wavelength λ, say in micrometers, corresponding to each wavenumber $\tilde{\nu}$, as shown in Figure 8.15. The code below shows how to do this.

Code: two_scales.py

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

dataFile = ["ref_wavnum_1.txt", "ref_wavnum_2.txt",
            "ref_wavnum_3.txt", "ref_wavnum_4.txt", ]

fig, ax = plt.subplots(figsize=(8.5, 4.5))
c = sns.color_palette("flare_r", 6)  # a Seaborn palette
for i, fname in enumerate(dataFile):
    nu, sig = np.loadtxt(
        "twoScalesData/" + fname, unpack=True, delimiter="\t")
    ax.plot(nu, sig, color=c[i])
ax.set_xlabel("wavenumber [cm$^{-1}$]")
```

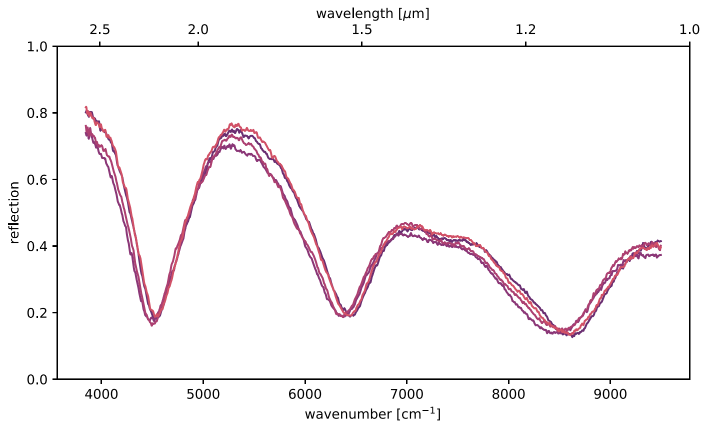


Figure 8.15 Figure with two x scales.

```
15 ax.set_ylabel("reflection")
16 ax.set_ylim(0.0, 1.0)
17 axtop = ax.twiny()
18 axtop.set_xlim(ax.get_xlim())
19 wavelen = np.array([1.0, 1.2, 1.5, 2.0, 2.5])  # microns
20 axtop.set_xticks(1e4 / wavelen)  # place ticks on top axis
21 axtop.set_xticklabels(wavelen)
22 axtop.set_xlabel(r"wavelength [$\mu$m]")
23 
24 fig.savefig("figures/two_scales.pdf")
25 plt.show()
```

Four data sets are loaded and plotted *vs.* wavenumber $\tilde{\nu}$ using semi-log axes `ax` in a `for` loop (lines 10–13). A second scale for all four data sets is set up by calling `twiny()`, which instructs Matplotlib to use the same y-axis for the new x axis `axtop` at the top of the plot. In contrast to the second set of axes we set up in Section 8.5.1, where we set up the two axes for two different data sets, here the second (top) axis is for the same data set: we simply want to *label* the (second) axis differently. Therefore, the second (`axtop`) axis needs to have the same x range and data limits as the first (`ax`) axis. This is done with the `set_xlim` command in line 18. Lines 19–21 set up the scale for the top axis. Because wavenumber is specified in cm^{-1} and wavelength is specified in $\mu$m, the conversion from $\mu$m to cm^{-1} is `1e4/wavelen`. Line 20–22 specifies the ticks and label in micrometers on the top axis.

8.6 PLOTS WITH INSETS

Sometimes, you would like to include a plot or an image as an inset within another plot. Figure 8.16 shows an example of two inset plots and an image being

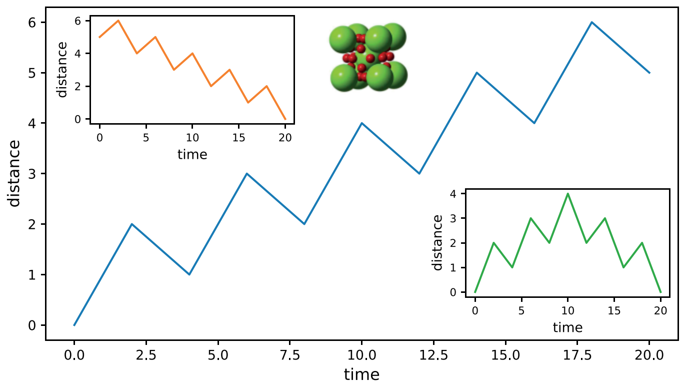


Figure 8.16 Figure with two inset plots and an inset image.

included within a larger plot. There are several ways to do this in Matplotlib. You might be tempted to use the `inset_axes` method from the `mpl_toolkits` package. Don't do it. While it's been written precisely for this purpose, it has too many parameters that interact with each other in complex ways, and you are likely to spend excessive time tweaking them to get the desired result. Instead, use `add_axes`, a method attached to Matplotlib's `figure` class, the class you use to create a figure. Using `add_axes` will also require some tweaking to get the desired result. Still, there are fewer parameters to adjust, and they interact minimally with each other, meaning that you will arrive at a satisfactory result more quickly and with less frustration.

The plot in Figure 8.16 was created using the program `inset.py` shown below. The insets are sized and positioned in lines 20, 26, and 37 using `add_axes(left, bottom, width, height)`, where `left` and `bottom` set the position of the left and bottom positions of the rectangle that contains each inset and `width` and `height` set the width and height of each rectangle. These four distances are fractions of the overall figure size. Once an inset is set and sized according to your purposes, you can adjust the sizes of the axes and tick labels to fit the inset size or, in the case of an image, remove them altogether. The image is displayed in line 39 using `imshow` from the Python Imaging Library (PIL), which is imported at the top of the program. In order not to distort the displayed image, the *ratio* of `width` and `height` used in line 37 should correspond to the actual ratio of the image displayed; in this case, `cs6c60` is a square image so `width` and `height` are equal. On the other hand, the absolute sizes of `width` and `height` can be set to get the overall image size you want for the plot.

Code: inset.py

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

yup = np.array([0, 2, 1, 3, 2, 4, 3, 5, 4, 6, 5])
ydn = np.array([5, 6, 4, 5, 3, 4, 2, 3, 1, 2, 0])
ypk = np.array([0, 2, 1, 3, 2, 4, 2, 3, 1, 2, 0])
x = np.linspace(0, 20, len(yup))

fig, ax = plt.subplots(figsize=(8.5, 4.5))

# Main plot
ax.plot(x, yup, color="C0")
ax.set_xlabel("time", fontsize=12)
ax.set_ylabel("distance", fontsize=12)

# Upper left inset plot
# Inset boundaries are fractions of figure size. (0,0 is bottom left)
left, bottom, width, height = [0.18, 0.61, 0.25, 0.25]
axUL = fig.add_axes([left, bottom, width, height])
axUL.plot(x, ydn, color="C1")

# Lower right inset plot
# Inset boundaries are fractions of figure size. (0,0 is bottom left)
left, bottom, width, height = [0.64, 0.21, 0.25, 0.25]
axLR = fig.add_axes([left, bottom, width, height])
axLR.plot(x, ypk, color="C2")

for axx in [axUL, axLR]:
    axx.set_xlabel("time", fontsize=10)
    axx.set_ylabel("distance", fontsize=10)
    axx.tick_params(labelsize=8)

# Upper middle inset image (large green & small red spheres)
# Inset boundaries are fractions of figure size. (0,0 is bottom left)
left, bottom, width, height = [0.42, 0.66, 0.2, 0.2]
axPIC = fig.add_axes([left, bottom, width, height])
axPIC.axis("off")
axPIC.imshow(Image.open("figures/cs6c60.jpg"))

fig.savefig("figures/inset.pdf")
plt.show()
```

8.7 MATHEMATICS AND GREEK SYMBOLS

Matplotlib can display mathematical formulas, Greek letters, and symbols using a math rendering module known as *mathtext*. The mathtext module parses a subset of Donald Knuth's TeX mathematical typesetting language and provides basic mathematical typesetting without any software other than Matplotlib.

If, in addition, you have TEX (and/or LATEX) as a separate stand-alone program (such as MacTex [TexShop] or MiKTeX), then you can do even more. In what follows, we will assume that you are using the native Matplotlib mathtext but will make a few comments applicable to those with a separate installation of LATEX.

Matplotlib's mathtext can display Greek letters and mathematical symbols using the syntax of TEX. If you are familiar with TEX or LATEX, you have little to learn. Even if you are unfamiliar with them, the syntax is simple enough to employ with little effort in most cases.

You designate text as mathtext by placing dollar signs ($) in a text string at the beginning and end of any part of the string that you want to be rendered as math text. You should also use raw strings in most cases, meaning you should precede a string's quotes with the letter `r`. For example, the following commands produce a plot titled "$\pi > 3$."

```
 In[1]: plot([1, 2, 3, 2, 3, 4, 3, 4, 5])
Out[1]: [<matplotlib.lines.Line2D at 0x11d5c4780>]

 In[2]:  title(r'$\pi > 3$')
Out[2]:  <matplotlib.text.Text at 0x11d59f390>
```

Where the Matplotlib function normally takes a string as input, you simply input the mathtext string. Note the `r` before the string in the `title` argument and the dollar signs (`$`) inside the quotes of the title string.

Subscripts and superscripts are designated using the underline "_" and caret "^" characters, respectively. Multiple characters to be grouped together in a subscript or superscript should be enclosed in a pair of curly braces `{...}`. All of this and more is illustrated in the plot shown in Figure 8.17, produced by the Python code below.

Code: mpl_latex_demo.py

```
import numpy as np
import matplotlib.pyplot as plt


def f0(t, omega, gamma, tau):
    wt = omega * t
    f1 = np.sin(wt) + (np.cos(wt) - 1.0) / wt
    f2 = 1.0 + (gamma / omega) * f1
    return np.exp(-t * f2 / tau)


omega = 12.0
gamma = 8.0
tau = 1.0
t = np.linspace(0.01, 10.0, 500)
f = f0(t, omega, gamma, tau)
```

```
17
18 plt.rc("mathtext", fontset="stix")   # Use with mathtext
19 # plt.rc("text", usetex=True)        # Use with Latex
20 # plt.rc("font", family="serif")     # Use with Latex
21
22 fig, ax = plt.subplots(figsize=(7.5, 4.5), tight_layout=True)
23 ax.plot(t, f, color="C0")
24 ax.set_ylabel(r"$f_0(t)$", fontsize=14)
25 ax.set_xlabel(r'$t/\tau\quad\rm{(ms)}$', fontsize=14)
26 ax.text(0.45, 0.95,
27         r"$\Gamma(z)=\int_0^\infty x^{z-1}e^{-x}dx$",
28         fontsize=16, ha="right", va="top",
29         transform=ax.transAxes)
30 ax.text(0.45, 0.75,
31         r"$e^x=\sum_{n=0}^\infty\frac{x^n}{n!}$",
32         fontsize=16, ha="right", va="top",
33         transform=ax.transAxes)
34 ax.text(0.45, 0.55,
35         r"$\zeta(z)=\prod_{k=0}^\infty \frac{1}{1-p_k^{-z}}$",
36         fontsize=16, ha="left", va="top",
37         transform=ax.transAxes)
38 ax.text(0.95, 0.80,
39         r"$\omega={0:0.1f},\;\gamma={1:0.1f},\;\tau={2:0.1f}$"
40         .format(omega, gamma, tau),
41         fontsize=14, ha="right", va="top",
42         transform=ax.transAxes)
43 ax.text(0.85, 0.35,
44         r"$e=\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$",
45         fontsize=14, ha="right", va="top",
46         transform=ax.transAxes)
47
48 fig.savefig("./figures/mplLatexDemo.pdf")
49 plt.show()
```

Line 18 sets the font to be used by mathtext, in this case `stix`. If you omit such a statement, mathtext uses its default font `dejavusans`. Other options include `dejavuserif`, `cm` (Computer Modern), `stix`, and `stixsans`. Try them out!

Let's see what LaTeX can do. First, the y-axis label in Figure 8.17 is $f_0(t)$, which has a subscript formatted using mathtext in line 24. The x-axis label is also typeset using mathtext, but the effects are more subtle. The variable t is italicized, as is proper for a mathematical variable, but the units (ms) are not, which is also standard practice. The math mode italics are turned off with the `\rm` (Roman) switch, which acts on text until the next closing curly brace (}).

Lines 26–29 provide the code to produce the expression for the Gamma function $\Gamma(z)$, lines 30–33 produce the expression for the Taylor series for the exponential function, and lines 34–37 produce the product that gives the zeta function. Lines 38–42 provide the code that produces the expressions for ω, γ, and τ. Lines 43–46 provide the code that produces the limit expression for the natural logarithm base e. The mathtext (TEX) code that produces the four typeset equations is contained in strings in lines 27, 31, 35, 39, and 44. The

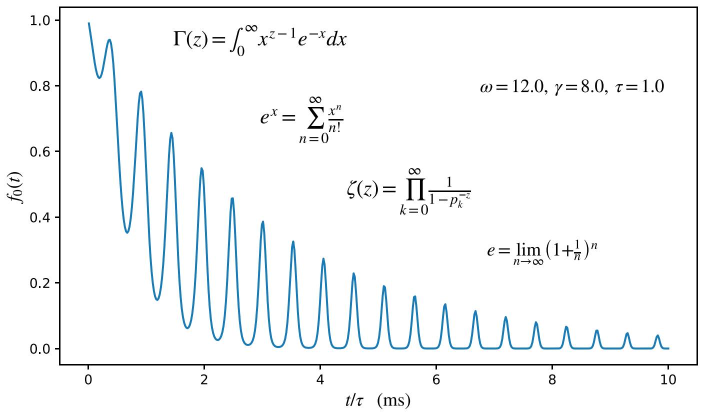


Figure 8.17 Plot using Matplotlib's mathtext for Greek letters and mathematical symbols.

TABLE 8.2 LaTeX (mathtext) codes for Greek letters.

α	`\alpha`	β	`\beta`	γ	`\gamma`	δ	`\delta`
ϵ	`\epsilon`	ε	`\varepsilon`	ζ	`\zeta`	η	`\eta`
θ	`\theta`	ι	`\iota`	κ	`\kappa`	λ	`\lambda`
μ	`\mu`	ν	`\nu`	ξ	`\xi`	π	`\pi`
ρ	`\rho`	ϱ	`\varrho`	σ	`\sigma`	τ	`\tau`
υ	`\upsilon`	ϕ	`\phi`	φ	`\varphi`	χ	`\chi`
ψ	`\psi`	ω	`\omega`	Γ	`\Gamma`	Δ	`\Delta`
Θ	`\Theta`	Λ	`\Lambda`	Ξ	`\Xi`	Π	`\Pi`
Σ	`\Sigma`	Υ	`\Upsilon`	Φ	`\Phi`	Ψ	`\Psi`
Ω	`\Omega`						

strings begin and end with `$` symbols, which activate and deactivate mathtext's math mode.

Special commands and symbols begin with a backslash in mathtext, which follows the convention of TeX. However, Python strings also use backslashes for certain formatting commands, such as `\t` for TAB or `\n` for a new line. The `r` preceding the strings in lines 27, 31, 35, 39, and 44 makes those strings *raw* strings, which turns off Python's special backslash formatting commands so that backslash commands are interpreted as mathtext.

Table 8.2 provides the LaTeX (mathtext) codes for Greek letters; Table 8.3 gives the code for miscellaneous mathematical expressions.

TABLE 8.3 Mathtext (LaTeX) codes for miscellaneous mathematical expressions. Search "latex math symbols" on the internet for more extensive lists.

$\sum$	`\sum`	$\sum_{i=0}^{\infty}$	`\sum_{i=0}^{\infty}`
$\prod$	`\prod`	$\prod_{i=0}^{\infty}$	`\prod_{i=0}^{\infty}`
$\int$	`\int`	$\int_{a}^{b}f(x)\,dx$	`\int_{a}^{b}f(x)\,dx`
$\sqrt{q}$	`\sqrt{q}`	$\lim_{x\to\infty}\ f(x)$	`\lim_{x\to\infty}\ f(x)`
∇	`\nabla`	$e^{i\pi}+1=0$	`e^{i\pi}+1=0`
$\sin\,\phi$	`\sin\,\phi`	$\sinh\,z$	`\sinh\,z`
$\cos\,\theta$	`\cos\,\theta`	$\cosh\,y$	`\cosh\,y`
$\tan\,x$	`\tan\,x`	$\tanh\,x$	`\tanh\,x`
$\mathbf{B}$	`\mathbf{B}`	$\vec{W} = \vec{F}\cdot\vec{x}$	`\vec{W} = \vec{F}\cdot\vec{x}`
23°C	`$23^\circ$C`	$\vec{L} = \vec{r}\times\vec{p}$	`\vec{L} = \vec{r}\times\vec{p}`
$a \leq b$	`a \leq b`	$\frac{a^{1/3}b}{x+y}$	`\frac{a^{1/3}b}{x+y}`
$a \geq b$	`a \geq b`	$\langle x \rangle$	`\langle x \rangle`
$a \equiv b$	`a \equiv b`	$a^\dagger\|n\rangle$	`a^\dagger|n\rangle`

The mathtext codes are the same as LaTeX codes but are sometimes rendered slightly differently from what you might use in standard LaTeX. For example, the code `$\cos\theta$` produces $\cos\theta$ in LaTeX, but produces $\mathrm{cos}\theta$ using Matplotlib's mathtext: there is too little space between cos and θ in the mathtext expression. For this reason, you may need to insert some extra space in your mathtext code where you wouldn't need to do so using LaTeX. Spaces of varying length can be inserted using `\,` `\:` `\;` and `\␣` for shorter spaces (of increasing length); `\quad` and `\qquad` provide longer spaces equal to one and two character widths, respectively.

While extra spacing in rendered mathtext equations matters, extra spaces in mathtext *code* makes no difference in the output. In fact, spaces in the code are not needed if the meaning of the code is unambiguous. Thus, `$\cos \, \theta$` and `$\cos\,\theta$` produce exactly the same output, namely $\cos\,\theta$.

Matplotlib can produce even more beautiful mathematical expressions if you have a separate stand-alone version of TeX (and/or LaTeX) on your computer. In this case, you can access a broader selection of fonts—all the fonts you already installed with your TeX installation. You also can write LaTeX code using the `\displaystyle` switch, which produces more nicely proportioned expressions. To do this, each expression in lines 27, 31, 35, and 44 should have

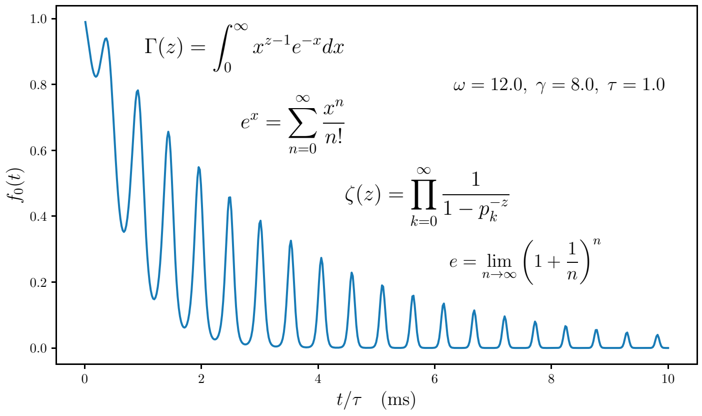


Figure 8.18 Plot using LaTeX for Greek letters and mathematical symbols.

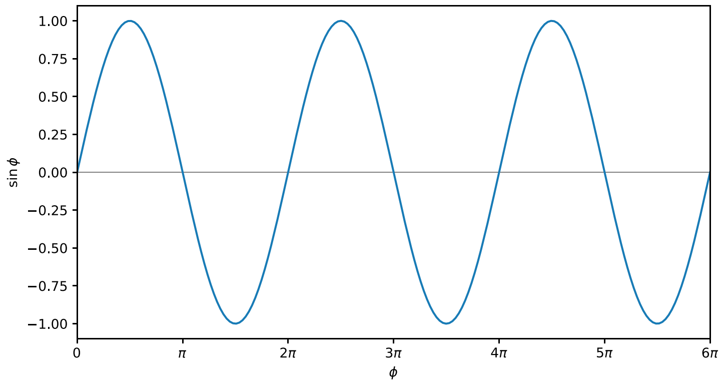


Figure 8.19 Manual ticks.

`\displaystyle` prepended to each mathtext string. You would also uncomment lines 19–20 and comment out line 18. The result is shown in Figure 8.18.[4]

8.7.1 Manual Axis Labeling

For some plots it can be useful to manually set where the ticks and tick labels occur on the axes. For example, if you plot trigonometric functions, you might want to place the tick in units of π as shown in Figure 8.19.

Matplotlib has a sophisticated set of tools for creating all kinds of ticks,

[4]For LaTeX experts, Matplotlib's mathtext does not recognize TeX's `$$...$$` syntax for displayed equations, but you can get the same result with the `\displaystyle` switch.

both major and minor. Here, we focus on the most straightforward of those tools as they are sufficient for the task at hand. The code to produce Figure 8.19 is given here

Code: manual_ticks.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  phi = np.linspace(0.0, 6.0*np.pi, 200)
5  wave = np.sin(phi)
6
7  fig, ax = plt.subplots(figsize=(8.5, 4.5))
8  ax.plot(phi, wave)
9  ax.axhline(color="gray", lw=0.5, zorder=-1)
10 ax.set_xlim(0.0, 6.0)
11 ax.set_xticks(np.linspace(0.0, 6.0*np.pi, 7))
12 labels = ["$0$", r"$\pi$", r"$2\pi$", r"$3\pi$", r"$4\pi$", r"$5\pi$",
13           r"$6\pi$"]
14 ax.set_xticklabels(labels)
15 ax.set_xlabel(r"$\phi$")
16 ax.set_ylabel(r"$\sin\phi$")
17
18 plt.savefig("./figures/manual_ticks.pdf")
19 plt.show()
```

The location of the ticks on the x-axis is specified by Matplotlib's `set_ticks` function on line 11 in the program above. Lines 12–14 specify the labels; make sure that there are as many labels specified as ticks.

8.8 THE STRUCTURE OF Matplotlib: OOP AND ALL THAT

This section provides an overview of the logical structure of Matplotlib. On a first pass, you can skip this section, but you may find it useful for various reasons. First, it will help you better understand the Matplotlib syntax introduced in Section 8.4. Second, it should improve your ability to read and understand the online documentation and other online resources such as our favorite, stack**overflow**.[5] The writing in these and other web resources is replete with jargon and ideas that can be frustratingly obscure to a novice. Much of it is the jargon of object-oriented programming. Other parts pertain to the jargon of graphical user interfaces or GUIs. This section introduces the basic structure of Matplotlib and explains its lexicon.

Matplotlib is a Python module for generating graphical output to your computer screen or a computer file. Fundamentally, its job is to translate Python scripts into graphical instructions your computer can understand. It does this using two different layers of software, the *backend* layer and the *artist*

[5]https://stackoverflow.com/tags/matplotlib/info

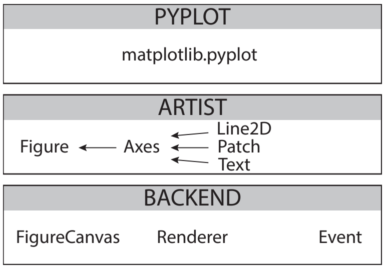


Figure 8.20 Matplotlib software layers, from the low-level backend to the high level PyPlot.

layer. To these two layers, it adds a scripting layer, PyPlot, which we have met already (`import matplotlib.pyplot as plt`). PyPlot is a convenience layer and not really necessary, but it facilitates rapid scripting and aids portability. As you shall see, we advocate using a hybrid of the scripting and artist layers for most programming. Figure 8.20 portrays the Matplotlib software hierarchy.

8.8.1 The Backend Layer

To be concrete, we'll start by considering the task of creating a figure on your computer screen. Matplotlib must be able to generate graphics using different computer platforms: Linux, Microsoft Windows, and macOS. Of course, we want Matplotlib to do this in a way that is transportable from one platform to another and transparent to you, the user. Ideally, the Python Matplotlib code you write on your PC should work without modification on your friend's Mac or Linux computer.

To accomplish this, Matplotlib uses open-source cross-platform software toolkits written by third parties. There are several different toolkits available. Most of these toolkits can write graphics to computer screens on different platforms, including Windows, macOS, and Linux. They are written mainly in C++ and are very efficient and powerful. To harness their power and versatility, Matplotlib provides several Python "wrappers"—Python functions—that call the C++ functions of these toolkits to send graphics instructions to your computer.[6] Matplotlib calls these wrappers *backends*.

Several backends have been written for Matplotlib. They fall into two categories: those written for output to files—hardcopy backends—and those for

[6] A Python "wrapper" provides a Python interface to software written in a different computer language, such as C++ or Fortran. You call a Python function, and it calls a C++ or Fortran program.

output to a computer screen—user interface backends, also known as interactive backends.

For output to a computer screen, the qt5Agg backend is among the most versatile and widely used, so we will use it as an example of what an interactive backend does. The qt5Agg backend is made from two C++ toolkits: Qt[7] and Agg.[8] Qt can define windows on your computer screen, create buttons, scrollbars, and other *widgets*, that is to say, elements of a GUI. It can also process *events*, actions like clicking a mouse, pressing a keyboard key, moving a scroll bar, or pressing a button. This includes processing events generated by the computer, rather than the user, like an alarm clock going off or a process finishing. The Agg toolkit is a rendering program that produces pixel images in memory from vectorial data. For example, you provide Agg with the equation for a circle, and it determines which pixels to activate on the computer screen. To produce faithful high-resolution graphics, it employs advanced rendering techniques like *anti-aliasing*[9].

The job of a Matplotlib backend is to provide a Python interface to the functionality of the underlying C++ (or other language) toolkits, which for qt5Agg are qt5 and Agg. The Python-facing parts of all Matplotlib backends have three basic classes:

`FigureCanvas` defines the canvas—a figure window or a graphics file—and transfers the output from the `Renderer` onto this canvas. It also translates Qt events into the Matplotlib `Event` framework (see below).

`Renderer` does the drawing. It connects Matplotlib to the Agg library described above.

`Event` handles user inputs such as keyboard and mouse events for Matplotlib.

Besides the qt5Agg backend, there are several other commonly used interactive backends: TkAgg, GTK3Agg, GTK3Cairo, WXAgg, and macOSX, to name a few. Why all the different backends? Part of the reason is historical. Early in the development of Matplotlib, many toolkits worked on only one platform, meaning that a separate backend had to be developed for each. As time has passed, most toolkits became cross-platform. As better cross-platform graphical tools, generally written in C++, were developed, programmers in the Python world wanted access to their functionality. Hence, the different backends.

[7] https://en.wikipedia.org/wiki/Qt_(software)
[8] https://en.wikipedia.org/wiki/Anti-Grain_Geometry
[9] See https://en.wikipedia.org/wiki/Spatial_anti-aliasing.

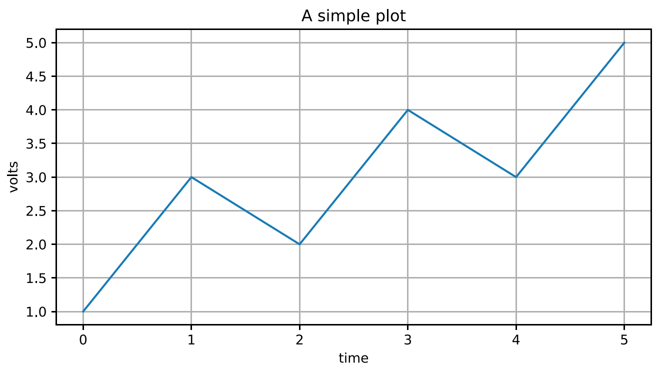


Figure 8.21 Plotting without PyPlot using the pure OO-interface.

As mentioned earlier, there are also hardcopy backends that only produce graphical output to files. These include Agg (which we already met as part of qt5Agg), PDF (to produce the Adobe portable document format), SVG (scalable vector graphics), and Cairo (png, ps, pdf, and svg).

In the end, the idea of a Matplotlib backend is to provide the software machinery for setting up a canvas to draw on and the low-level tools for creating graphical output: plots and images. The drawing tools of the backend layer, while sufficient for producing any output you might want, work at too low of a level to be useful for everyday programming. For this, Matplotlib provides another layer, the artist layer, which provides the software tools you will use for creating and managing graphical output.

8.8.2 The Artist Layer

The artist layer consists of a hierarchy of Python classes that facilitate creating a figure and embellishing it with any of the features we might desire: axes, data points, curves, axes labels, legends, titles, annotations, and everything else. The first step is to create a figure and place it on a canvas (figure window, whether on a computer screen or a computer file). To the figure, we add axes, data, labels, *etc.* When we have everything the way we want, we send it to the screen to display and/or save it to a file.

To make this concrete, consider the program below that creates the plot shown in Figure 8.21.

Code: oop_test.py

```python
from matplotlib.backends.backend_qt5agg \
    import FigureCanvasQTAgg as FigureCanvas
```

```
3  from matplotlib.figure import Figure
4
5  fig = Figure(figsize=(8, 4))
6  canvas = FigureCanvas(fig)
7  ax = fig.add_subplot(111)
8  ax.plot([1, 3, 2, 4, 3, 5])
9  ax.set_title("A simple plot")
10 ax.grid(True)
11 ax.set_xlabel("time")
12 ax.set_ylabel("volts")
13 fig.savefig("figures/oop_test.pdf")
14 canvas.show()
```

First, we import the `FigureCanvas` from the qt5Agg backend.[10] Then, we import `Figure` from `matplotlib.figure`. After finishing our imports, the first step is to define and name (`fig`) the `Figure` object that will serve as a container for our plot. The next step is to attach the figure to an instance of `FigureCanvas`, which we name `canvas`, from the `qt5Agg` backend. This places the canvas on the screen and connects all the Matplotlib routines to the Qt and Agg C++ routines that write to the screen. Next, in line 7, we create a set of axes on our figure, making a single subplot that takes up the entire frame of the figure. Lines 7–12 should be familiar to you, as they use the syntax introduced in Section 8.4. Finally, we write the plot to a file (line 13) and the screen (line 14) from the canvas, which makes sense because that connects the Matplotlib routines to the hardware.

The code in this example is native Matplotlib object-oriented (OO) code. In its purest form, it's how Matplotlib code should be written. The code is entirely transportable from one computer to another irrespective of the operating system, so long as the qt5Agg backend is included in the local machine's Python installation. In this case, the output should look the same, whether on the screen or in a file, on Microsoft Windows, macOS, or any of the different flavors of Linux. By the way, if another machine does not have the qt5Agg backend installed, you can change lines 1 and 2 to a different backend that is installed, and the program should work as expected, with no discernible differences.

Before moving on to the next layer of Matplotlib, it's useful to introduce some Matplotlib terminology—jargon—for describing the artist layer. The routines in lines 5 and 7–12 are part of the Artist module and the eponymous Artist class of Matplotlib. Collectively and individually, all the routines that get attached to `fig` and its descendant `ax` are known as *Artists*: `add_subplot`, `plot`, `title`, `grid`, `xlabel`, *etc.* Artists are those Matplotlib objects that draw on

[10]Lines 1–2 can be shortened to read:

```
from matplotlib.backends.backend_qt5Agg import FigureCanvas
```

the canvas, including `figure`. You will see the term Artist employed liberally in online documentation and commentary on Matplotlib. It's simply a part of the Matplotlib lexicon, along with backend, PyPlot, and the yet-to-be-mentioned, PyLab.

8.8.3 The PyPlot (scripting) Layer

For newcomers to Python, learning about backends and the OOP syntax can create a barrier to code development, particularly engineers and scientists who come to Python after first being exposed to MATLAB®. This is why the PyPlot module was developed, to provide a more familiar interface to those coming from software packages like MATLAB®, or simply for those new to Python programming.

Consider, for example, our first plotting script of a sine function, which we reproduce here from page 167:

Code: sine_plot33.py

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 4.0 * np.pi, 33)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```

After importing NumPy and PyPlot, we define the *x*-*y* arrays in two lines, make a plot in the next line, and then display it in the last line: the epitome of simplicity.

8.8.3.1 PyPlot and Backends

There is no mention of a backend in this syntax. Nevertheless, one must be used so that Matplotlib can communicate with the computer. So, how does a backend get loaded using PyPlot? And which backend? First, a backend is loaded when PyPlot is imported. Second, which backend is loaded is determined by how you launch IPython. If you launch it from a terminal or Jupyter Lab, the default backend is set by a Matplotlib configuration file named `matplotlibrc` on your computer. It is but one of many files installed on your computer when Matplotlib was installed, for example, using the Anaconda installation of Python. You can find out where this file is from an IPython prompt by typing

```
 In[1]: import matplotlib

 In[2]: matplotlib.matplotlib_fname()
Out[2]: '/Users/dp/.matplotlib/matplotlibrc'
```

Alternatively, if you use a Python IDE like Spyder, the backend is set in a Preferences menu and loaded when the IDE is launched. In any case, you can find out which backend was installed in your IPython shell using a IPython magic command

```
In [3]: %matplotlib
Out[3]: Using matplotlib backend: Qt5Agg
```

Thus, using PyPlot frees you up from thinking about the backend—how Matplotlib is connected to your computer's hardware—when writing Matplotlib routines. It also makes your code more portable. That's a good thing.

8.8.3.2 PyPlot's "State-Machine" Environment

For *simple plots*, you can proceed using the syntax introduced in Section 8.2 and Section 8.3. The only Matplotlib prefix used in this syntax is `plt`. You can do most of the things you want to do with Matplotlib working in this way. You can make single plots or plots with multiple subplots, as demonstrated in Section 8.2.4.

When working in this mode with PyPlot, we are working in a "state-machine" environment, meaning that where on a canvas PyPlot adds features (*i.e.*, Artists) depends on the program's state. For example, in the program starting on page 179 that produces the plot in Figure 8.10, you make the first subplot by including `plt.subplot(2, 1, 1)` in the program. Everything after that statement affects that subplot until the program comes to the line `plt.subplot(2, 1, 2)`, which opens up a second subplot. After that statement, everything affects the second subplot. You've changed the "state" of the "machine" (*i.e.*, the FigureCanvas object). Which subplot is affected by commands in the program depends on which state the machine is in.

8.8.3.3 PyPlot's Hybrid OOP Environment

Contrast operating in this state-machine mode with the syntax introduced in Section 8.4. In those programs, we load PyPlot, but then employ the OOP syntax for creating figures and making subplots. Each subplot object is referenced by a different object name, *e.g.*, `ax1`, `ax2`, or `ax[i, j]`, where `i` and `j` cycle through different values. These object names identify the target of a function, not the machine's state. It's a more powerful way of programming and provides more versatility in writing code. For example, in making the plots shown in Figure 8.13, we could conveniently combine the labeling of the y-axes by cycling through object names of each subplot toward the end of the program (see page 187). This would be hard to do with the state-machine approach.

Most programming with Matplotlib should be done using this hybrid (but basically OOP) approach introduced in Section 8.4. The state-machine approach we employed in earlier sections should be reserved for short snippets of code.

8.9 CONTOUR AND VECTOR FIELD PLOTS

Matplotlib has extensive tools for creating and annotating two-dimensional contour and vector field plots. A *contour plot* is used to visualize two-dimensional scalar functions, such as the electric potential $V(x,y)$ or elevations $h(x,y)$ over some physical terrain. Vector field plots come in different varieties. Field line plots, which in some contexts are called *streamline plots*, show the direction of a vector field over some 2D (x,y) range. There are also *quiver plots*, which consist of a 2D grid of arrows that give the direction and magnitude of a vector field over some 2D (x,y) range.

8.9.1 Making a 2D Grid of Points

When plotting a function $f(x)$ of a single variable, the first step is usually to create a one-dimensional x array of points and then to evaluate and plot the function $f(x)$ at those points, often drawing lines between the points to create a continuous curve. Similarly, when making a two-dimensional plot, we usually need to make a two-dimensional x-y array of points and then evaluate and plot the function $f(x,y)$, be it a scaler or vector function, at those points, perhaps with continuous curves to indicate the value of the function over the 2D surface.

Thus, instead of having a line of evenly spaced x points, we need a *grid* of evenly spaced x-y points. Fortunately, NumPy has a function `np.meshgrid` for doing just that. The procedure is first to make an x-array at even intervals over the range of x to be covered and then to do the same for y. These two one-dimensional arrays are input to the `np.meshgrid` function, which makes a two-dimensional mesh. Here is how it works:

```
In[1]: x = linspace(-1, 1, 5)

In[2]: x
Out[2]: array([-1. , -0.5,  0. ,  0.5,  1. ])

In[3]: y = linspace(2, 6, 5)

In[4]: y
Out[4]: array([ 2.,  3.,  4.,  5.,  6.])
```

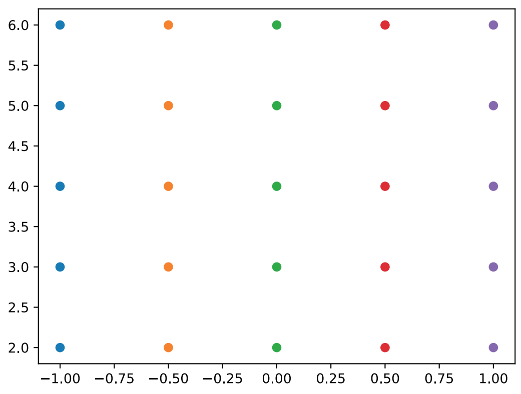


Figure 8.22 Point pattern produced by `np.gridmesh(X, Y)`.

```
In[5]: X, Y = np.meshgrid(x, y)

In[6]: X
Out[6]:
array([[-1. , -0.5,  0. ,  0.5,  1. ],
       [-1. , -0.5,  0. ,  0.5,  1. ],
       [-1. , -0.5,  0. ,  0.5,  1. ],
       [-1. , -0.5,  0. ,  0.5,  1. ],
       [-1. , -0.5,  0. ,  0.5,  1. ]])

In[7]: Y
Out[7]:
array([[ 2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.,  4.],
       [ 5.,  5.,  5.,  5.,  5.],
       [ 6.,  6.,  6.,  6.,  6.]])

In[8]: plot(X, Y, 'o')
```

The output of `plot(X, Y, 'o')` is a 2D grid of points, as shown in Figure 8.22. Matplotlib's functions for making contour plots and vector field plots use the output of `gridmesh` as the 2D input for the functions to be plotted.

8.9.2 Contour Plots

The principal Matplotlib routines for creating contour plots are `contour` and `contourf`. Sometimes, you would like to make a contour plot of a function of two variables; other times, you may wish to make a contour plot of some data

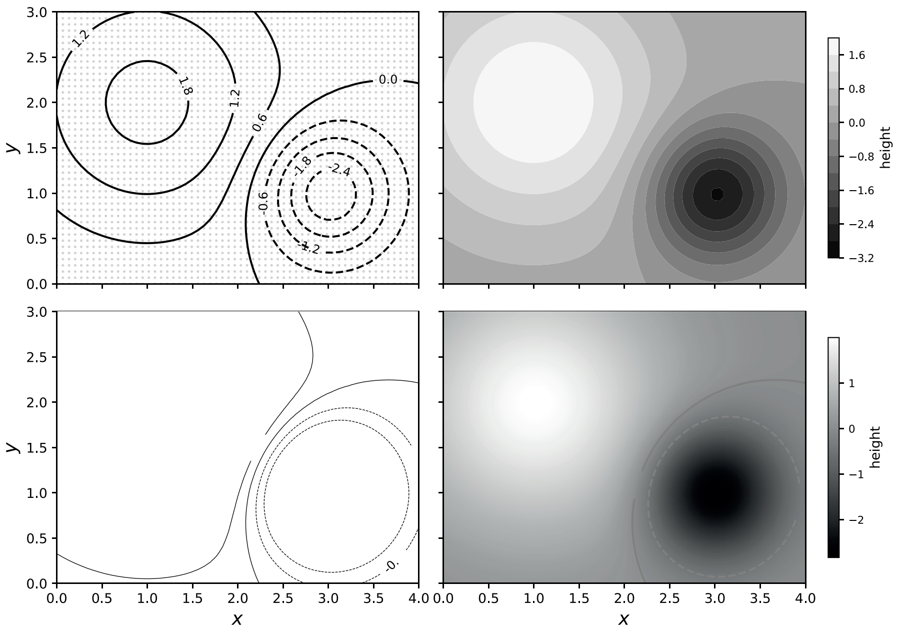


Figure 8.23 Contour plots.

you have. Of the two, making a contour plot of a function is simpler, which is all we cover here.

8.9.2.1 Contour Plots of Functions

Figure 8.23 shows four different contour plots. All were produced using `contour` except the upper left plot, which was produced using `contourf`. All plot the same function, which is the sum of a pair of Gaussians, one positive and the other negative:

$$f(x,y) = 2e^{-\frac{1}{2}[(x-2)^2+(y-1)^2]} - 3e^{-2[(x-1)^2+(y-2)^2]} \tag{8.1}$$

The code that produces Figure 8.23 is given below.

Code: contour4.py

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm  # color maps
import matplotlib


def pmgauss(x, y):
 8      r1 = (x - 1) ** 2 + (y - 2) ** 2
 9      r2 = (x - 3) ** 2 + (y - 1) ** 2
10      return 2 * np.exp(-0.5 * r1) - 3 * np.exp(-2 * r2)
11
12
13  a, b = 4, 3
14
15  x = np.linspace(0, a, 60)
16  y = np.linspace(0, b, 45)
17
18  X, Y = np.meshgrid(x, y)
19  Z = pmgauss(X, Y)
20
21  fig, ax = plt.subplots(2, 2, figsize=(9.4, 6.5),
22                         sharex=True, sharey=True,
23                         gridspec_kw={"width_ratios": [4, 5]})
24
25  CS0 = ax[0, 0].contour(X, Y, Z, 8, colors="k")
26  ax[0, 0].clabel(CS0, fontsize=9, fmt="%0.1f")
27  matplotlib.rcParams["contour.negative_linestyle"] = "dashed"
28  ax[0, 0].plot(X, Y, "o", ms=1, color="lightgray", zorder=-1)
29
30  CS1 = ax[0, 1].contourf(X, Y, Z, 12, cmap=cm.gray, zorder=0)
31  cbar1 = fig.colorbar(CS1, shrink=0.8, ax=ax[0, 1])
32  cbar1.set_label(label="height", fontsize=10)
33  plt.setp(cbar1.ax.yaxis.get_ticklabels(), fontsize=8)
34
35  lev2 = np.arange(-3, 2, 0.3)
36  CS2 = ax[1, 0].contour(X, Y, Z, levels=lev2, colors="k",
37                         linewidths=0.5)
38  ax[1, 0].clabel(CS2, lev2[1::2], fontsize=9, fmt="%0.1f")
39
40  CS3 = ax[1, 1].contour(X, Y, Z, 10, colors="gray")
41  ax[1, 1].clabel(CS3, fontsize=9, fmt="%0.1f")
42  im = ax[1, 1].imshow(Z, interpolation="bilinear",
43                       origin="lower", cmap=cm.gray,
44                       extent=(0, a, 0, b))
45  cbar2 = fig.colorbar(im, shrink=0.8, ax=ax[1, 1])
46  cbar2.set_label(label="height", fontsize=10)
47  plt.setp(cbar2.ax.yaxis.get_ticklabels(), fontsize=8)
48
49  for i in range(2):
50      ax[1, i].set_xlabel(r"$x$", fontsize=14)
51      ax[i, 0].set_ylabel(r"$y$", fontsize=14)
52      for j in range(2):
53          ax[i, j].set_aspect("equal")
54          ax[i, j].set_xlim(0, a)
55          ax[i, j].set_ylim(0, b)
56  plt.subplots_adjust(left=0.06, bottom=0.07, right=0.99,
57                      top=0.99, wspace=0.06, hspace=0.09)
58  fig.savefig("./figures/contour4.pdf")
59  plt.show()
```

After defining the function to be plotted in lines 7–10, the next step is to create the *x*-*y* array of points at which the function will be evaluated using `np.meshgrid`. We use `np.linspace` rather than `np.arange` to define the extent of the *x*-*y* mesh because we want the *x* range to go precisely from 0 to `a=4` and the *y* range to go precisely from 0 to `b=3`. We use `np.linspace` for two reasons. First, if we use `np.arange`, the array of data points does not include the upper bound, while `np.linspace` does. This is important for producing the grayscale (or color) background that extends to the upper limits of the *x*-*y* ranges in the upper-right plot, produced by `contourf`, of Figure 8.23. Second, to produce smooth-looking contours, one generally needs about 40–200 points in each direction across the plot, irrespective of the absolute magnitude of the numbers being plotted. The number of points is directly specified by `np.linspace` but must be calculated for `np.arange`. We follow the convention that the `meshgrid` variables are capitalized, which seems to be a standard followed by many programmers. It's certainly not necessary.

The upper-left contour plot takes the `X`-`Y` 2D arrays made using `gridspec` as its first two arguments and `Z` as its third argument. The third argument tells `contour` to make approximately 5 different levels in `Z`. We give the `contour` object a name, as it is needed by the `clabel` call in the next line, which sets the font size and the format of the numbers that label the contours. The line style of the negative contours is set globally to be "dashed" by a call to Matplotlib's `rcparams`. We also plot the location of the `X`-`Y` grid created by `gridspec` just for the sake of illustrating its function; normally, these would not be plotted.

The upper-right contour plot is made using `contourf` with 12 different `Z` layers indicated by the different gray levels. The gray color scheme is set by the keyword argument `cmap`, which here is set to the `matplotlib.cm` color scheme `cm.gray`. Other color schemes can be found in the Matplotlib documentation by an internet search on "matplotlib choosing colormaps." The color bar legend on the right is created by the `colorbar` method, which is attached to `fig`. It is associated with the upper right plot by the name `CS1` of the `contourf` method *and* by the keyword argument `ax=ax[0, 1]`. Its size relative to the plot is determined by the `shrink` keyword. The font size of the color bar label is set using the generic set property method `setp` using a somewhat arcane but compact syntax.

For the lower-left contour plot `CS2`, we manually specify the levels of the contours with the keyword argument `levels=lev2`. We specify that only every other contour will be labeled numerically with `lev2[1::2]` as the second argument of the `clabel` call in line 38; `lev2[0::2]` would also label every other contour, but the even ones instead of the odd ones.

The lower-right contour plot `CS3` has ten contour levels and a continuously varying grayscale background created using `imshow`. The `imshow` method uses only the `Z` array to determine the gray levels. The *x*-*y* extent of the grayscale background is determined by the keyword argument `extent`. By default, `imshow`

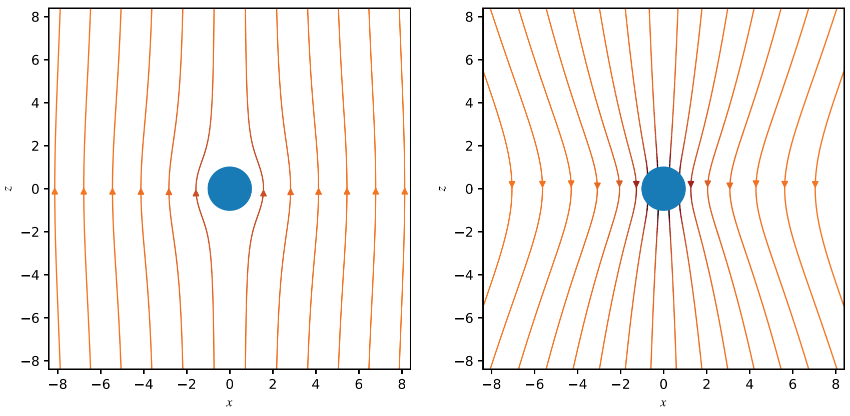


Figure 8.24 Streamlines of flow around a sphere falling in a viscous fluid.

uses the upper-left corner as its origin. We override the default using the `imshow` keyword argument `origin='lower'` so that the grayscale is consistent with the data. The keyword argument `iterpolation` tells `imshow` how to interpolate the grayscale between different `z` levels.

8.9.3 Streamline Plots

Matplotlib can also make streamline plots, sometimes called field line plots. The Matplotlib function call to make such plots is `streamplot`, and its use is illustrated in Figure 8.24 to plot the streamlines of the velocity field of a viscous liquid around a sphere falling through it at constant velocity u. The left plot is in the reference frame of the falling sphere, and the right plot is in the laboratory frame, where the liquid is very far from the sphere and is at rest. The program that produces Figure 8.24 is given below.

Code: stokes_flow_stream.py

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Circle


def v(u, a, x, z):
    """Return the velocity vector field v = (vx, vy)
    around sphere at r=0."""
    r = np.sqrt(x * x + z * z)
    R = a / r
    RR = R * R
    cs, sn = z / r, x / r
    vr = u * cs * (1.0 - 0.5 * R * (3.0 - RR))
    vtheta = -u * sn * (1.0 - 0.25 * R * (3.0 + RR))
    vx = vr * sn + vtheta * cs
    vz = vr * cs - vtheta * sn
    return vx, vz


# Grid of x, y points
xlim, zlim = 12, 12
nx, nz = 100, 100
x = np.linspace(-xlim, xlim, nx)
z = np.linspace(-zlim, zlim, nz)
X, Z = np.meshgrid(x, z)

# Set particle radius and velocity
a, u = 1.0, 1.0

# Velocity field vector, V=(Vx, Vz) as separate components
Vx, Vz = v(u, a, X, Z)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 4.5))

# Plot the streamlines using colormap and arrow style
color = np.log(np.sqrt(Vx * Vx + Vz * Vz))
seedx = np.linspace(-xlim, xlim, 18)  # Seed streamlines
seedz = -zlim * np.ones(len(seedx))  # evenly in far field
seed = np.array([seedx, seedz])
ax1.streamplot(x, z, Vx, Vz, color=color, linewidth=1,
               cmap="afmhot", density=5, arrowstyle="-|>",
               arrowsize=1.0, minlength=0.4,
               start_points=seed.T)
ax2.streamplot(x, z, Vx, Vz - u, color=color, linewidth=1,
               cmap="afmhot", density=5, arrowstyle="-|>",
               arrowsize=1.0, minlength=0.4,
               start_points=seed.T)
for ax in (ax1, ax2):
    # Add filled circle for sphere
    ax.add_patch(Circle((0, 0), a, color="C0", zorder=2))
    ax.set_xlabel("$x$")
    ax.set_ylabel("$z$")
    ax.set_aspect("equal")
    ax.set_xlim(-0.7 * xlim, 0.7 * xlim)
    ax.set_ylim(-0.7 * zlim, 0.7 * zlim)
plt.subplots_adjust(left=0.06, right=0.98, top=0.98, bottom=0.05)
plt.savefig("./figures/stokes_flow_stream.pdf")
plt.show()
```

The program starts by defining a function that calculates the velocity field as a function of the lateral distance $x$ and the vertical distance $z$. The function is a solution to the Stokes equation, which describes flow in viscous liquids at very low (zero) Reynolds number. The velocity field is the primary input into the Matplotlib `streamplot` function.

The next step is to use NumPy's `meshgrid` program to define the 2D grid of points at which the velocity field will be calculated, just as we did for the

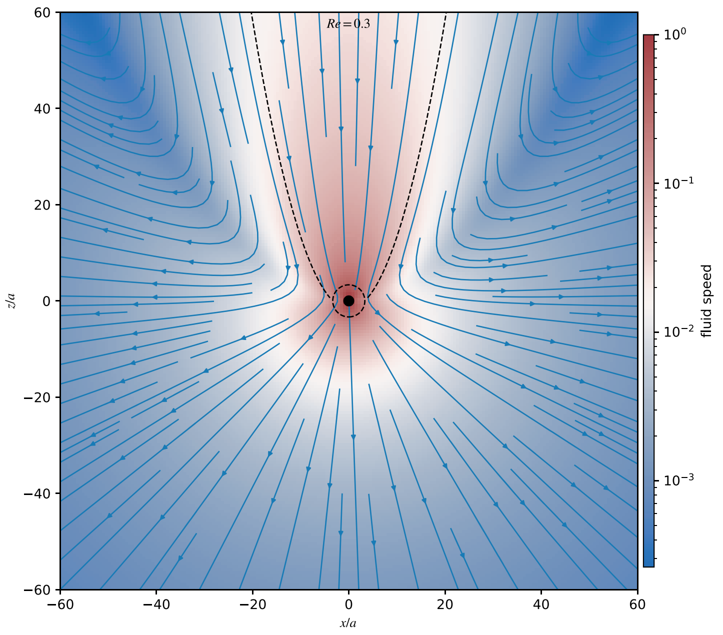


Figure 8.25 Streamlines of flow around a sphere falling in a fluid.

contour plots. After setting up the `meshgrid` arrays `X` and `Z`, we call the function we defined `v(u, a, X, Z)` to calculate the velocity field (line 31).

The `streamplot` functions are set up in lines 36–39 and called in lines 40–47. Note that for the `streamplot` function the input `x-z` coordinate arrays are 1D arrays but the velocity arrays `Vx-Vz` are 2D arrays. The arrays `seedx` and `seedx` set up the starting points (seeds) for the streamlines. You can leave them out and `streamplot` will make its own choices based on the values you set for the `density` and `minlength` keywords. Here, we have chosen them, along with the seed settings, so that all the streamlines are continuous across the plot. The other keywords set the properties for the arrow size and style, the streamlines' width, and the streamlines' coloring, in this case, according to the speed at a given point.

Let's look at another streamline plot, which illustrates some other possibilities for customizing streamline plots. The plot in Figure 8.25 shows the streamlines for a faster-moving sphere and makes different choices than the plot above. The code to make this plot, `stokes_oseen_flow.py`, is provided on the next page. The most noticeable difference is the use of the Matplotlib function `pcolor` in line 55 that adds background coloring to the plot keyed to the local speed of the liquid. A logarithmic color scale is used with a logarithmic color bar, which is set up by setting the `pcolor` keyword `norm=LogNorm(vmin=speed.min(), vmax=1)` in line 55.

Code: stokes_oseen_flow.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from matplotlib.patches import Circle
4  from matplotlib.colors import LogNorm
5  import seaborn as sns
6  
7  def v(u, a, x, z, Re):
8      """Return the velocity vector field v = (vx, vy)
9      around sphere at r=0."""
10     theta = np.arctan2(-x, -z)
11     cs, sn = np.cos(theta), np.sin(theta)
12     R = a / np.sqrt(x * x + z * z)
13     if Re > 0:  # Oseen solution
14         ex = np.exp(-0.5 * Re * (1.0 + cs) / R)
15         vr = 0.5 * u * R * (1.5 * (1.0 - cs) *
16                             ex - R * (3.0 * (1 - ex) / Re - R * cs))
17         vtheta = 0.25 * u * R * sn * (3.0 * ex - R * R)
18     else:  # Stokes solution
19         RR = R * R
20         vr = 0.5 * u * cs * R * (RR - 3.0)
21         vtheta = 0.25 * u * sn * R * (RR + 3.0)
22     vx = vr * sn + vtheta * cs
23     vz = vr * cs - vtheta * sn
24     return vx, vz
25 
26 
27 def stokesWake(x, Re):
28     """Return parabola r[1+cos(theta)]=xi of Stokes wake"""
29     z = -0.5 * (1.0 / Re - x * x * Re)
30     return np.ma.masked_where(x * x + z * z < 1.0 / Re ** 2, z)
31 
32 
33 # Set particle radius and velocity
34 a, u = 1.0, 1.0  # normalizes radius & velocity
35 Re = 0.3  # Reynolds number (depends on viscosity)
36 
37 # Grid of x, z points
38 xlim, zlim = 60, 60
39 nx, nz = 200, 200
40 x = np.linspace(-xlim, xlim, nx)
41 z = np.linspace(-zlim, zlim, nz)
42 X, Z = np.meshgrid(x, z)
43 
44 # Velocity field vector, v=(Vx, Vz) as separate components
45 Vx, Vz = v(u, a, X, Z, Re)
46 R = np.sqrt(X * X + Z * Z)
47 speed = np.sqrt(Vx * Vx + Vz * Vz)
48 speed[R < a] = u  # set particle speed to u
49 
50 fig, ax = plt.subplots(figsize=(8, 8))
```

```

# Plot the streamlines with an bwr colormap and arrow style
ax.streamplot(x, z, Vx, Vz, linewidth=1, density=[1, 2],
              arrowstyle="-|>", arrowsize=0.7, color="C0")
cntr = ax.pcolor(X, Z, speed, norm=LogNorm(vmin=speed.min(), vmax=1),
                 cmap=sns.color_palette("vlag", as_cmap=True))
if Re > 0:
    ax.add_patch(
        Circle((0, 0), 1 / Re, color="black", fill=False,
               ls="dashed", zorder=2))
    ax.plot(x, stokesWake(x, Re), color="black", lw=1, ls="dashed",
            zorder=2)
cbar = fig.colorbar(cntr, ax=ax, aspect=50, fraction=0.02,
                    shrink=0.9, pad=0.01)
cbar.set_label(label="fluid speed", fontsize=10)
plt.setp(cbar.ax.yaxis.get_ticklabels(), fontsize=10)
cbar.mappable.set_clim(vmin=speed.min(), vmax=1.0)

# Add filled circle for sphere
ax.add_patch(Circle((0, 0), a, color="black", zorder=2))
ax.set_xlabel("$x/a$")
ax.set_ylabel("$z/a$")
ax.set_aspect(1)
ax.set_xlim(-xlim, xlim)
ax.set_ylim(-zlim, zlim)
ax.text(0.5, 0.99, r"$Re = {0:g}$".format(Re), ha="center", va="top",
        transform=ax.transAxes)
fig.savefig("./figures/stokes_oseen_flow.pdf")
plt.show()
```

8.9.4 Vector Field (quiver) Plots

As an alternative to streamline plots, Matplotlib can also make vector-field plots, which are created by the function `quiver`. Figure 8.26 shows a vector-field plot and a streamline plot for an electric dipole. In a vector-field plot, arrows are drawn on a grid showing the direction of the field at each grid point. The length of each arrow is proportional to the magnitude of the local field. This presents a problem when the field strength varies by more than about an order of magnitude over the field of view. In the case of the electric dipole, the field diverges as r^{-2} so the arrows indicated the vectors get too long near $r = 0$. To cope with this problem, the data within a certain radius, indicated by the gray circle in Figure 8.26, are excluded from the vector field plot. This done by creating a mask in line 21 that excludes all grid points with a radius less than that specified by `rmask` in line 20 (recall that `~` means logical not). The mask is deployed in lines 24 and 28. An alternative is to make each arrow proportional to the logarithm of the field magnitude at each point.

Code: elec_dipole_vec_field.py

```
1  import matplotlib.pyplot as plt
2  from matplotlib.patches import Circle
```

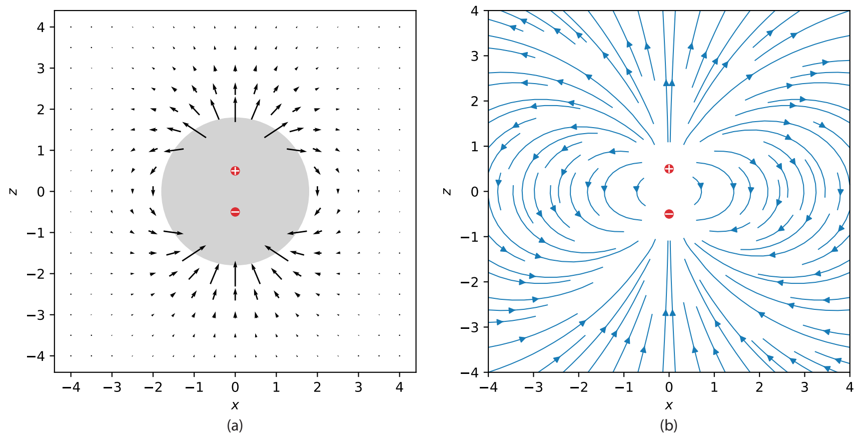


Figure 8.26 Electric dipole field. (a) Vector-field quiver plot. (b) Streamline plot.

```python
import numpy as np


def e_field(x, z, d):
    # Return electric dipole vector field E = (ex, ez) at (x, z)
    zp, zn = z - 0.5 * d, z + 0.5 * d  # z-coords of pos & neg charges
    rp2 = x * x + zp * zp  # distance squared from positive charge
    rn2 = x * x + zn * zn  # distance squared from negative charge
    emagpl = 1.0 / rp2  # E-field magnitude from positive charge
    emagmi = 1.0 / rn2  # E-field magnitude from positive charge
    ex = (emagpl / np.sqrt(rp2) - emagmi / np.sqrt(rn2)) * x
    ez = emagpl * zp / np.sqrt(rp2) - emagmi * zn / np.sqrt(rn2)
    return ex, ez


X, Z = np.meshgrid(np.arange(-4.0, 4.01, 0.5),
                   np.arange(-4.0, 4.01, 0.5))
rmask = 1.8  # radius of meshgrid mask
mask = ~(X**2 + Z**2 < rmask**2)  # meshgrid mask

d = 1.0
Ex, Ez = e_field(X[mask], Z[mask], d)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 4.5))
ax1.add_patch(Circle((0, 0), radius=rmask, fc="lightgray", zorder=1))
ax1.quiver(X[mask], Z[mask], Ex, Ez, scale=4.0, pivot="mid", zorder=2)

Ex, Ez = e_field(X, Z, d)
ax2.streamplot(X, Z, Ex, Ez, linewidth=0.75)

for ax in [ax1, ax2]:
    ax.set_aspect("equal")
    ax.plot([0.0, 0.0], [0.5 * d, -0.5 * d], "oC3", zorder=1)
    ax.plot([0.0], [0.5 * d], "+w", zorder=2)
    ax.plot([0.0], [-0.5 * d], "_w", zorder=2)
    ax.set_xlabel("$x$")
    ax.set_ylabel("$z$")

plt.subplots_adjust(left=0.06, right=0.98, top=0.98, bottom=0.05)
plt.savefig("./figures/elec_dipole_vec_field.pdf")
plt.show()
```

8.10 THREE-DIMENSIONAL PLOTS

While Matplotlib is primarily a 2D plotting package, it does have basic 3D plotting capabilities. To create a 3D plot, we need to import `Axes3D` from `mpl_toolkits.mplot3d` and then set the keyword `projection` to `'3d'` in a subplot call as follows:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

Different 2D and 3D subplots can be mixed within the same figure window by setting `projection='3d'` only in those subplots where 3D plotting is desired. Alternatively, *all* the subplots in a figure can be set to be 3D plots using the `subplots` function:

```
fig, ax = plt.subplots(subplot_kw={'projection': '3d'})
```

As you might expect, the third axis in a 3D plot is called the $z$-axis, and the same commands for labeling and setting the limits that work for the $x$ and y axes also work for the $z$-axis.

8.10.1 Cartesian Coordinates

As a demonstration of Matplotlib's 3D plotting capabilities, Figure 8.27 shows a wireframe and a surface plot of Eq. (8.1), the same equation plotted with contour plots in Figure 8.23. The code used to make Figure 8.27 is given below.

Code: wireframe_surface_plots.py

```
import numpy as np
import matplotlib.pyplot as plt


def pmgauss(x, y):
    r1 = (x-1)**2 + (y-2)**2
    r2 = (x-3)**2 + (y-1)**2
    return 2*np.exp(-0.5*r1) - 3*np.exp(-2*r2)

```

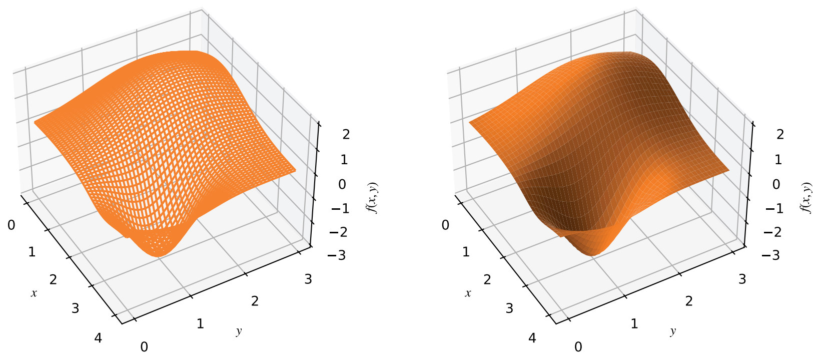


Figure 8.27 Wireframe and surface plots.

```python

a, b = 4, 3

# Create an x,y mesh
x = np.linspace(0, a, 60)
y = np.linspace(0, b, 45)

X, Y = np.meshgrid(x, y)
Z = pmgauss(X, Y)

fig, ax = plt.subplots(1, 2, figsize=(9.2, 4),
                       subplot_kw={"projection": "3d"})
for i in range(2):
    ax[i].set_zlim(-3, 2)
    ax[i].xaxis.set_ticks(range(a+1))  # manually set ticks
    ax[i].yaxis.set_ticks(range(b+1))
    ax[i].set_xlabel(r"$x$")
    ax[i].set_ylabel(r"$y$")
    ax[i].set_zlabel(r"$f(x,y)$")
    ax[i].view_init(40, -30)

# Plot wireframe and surface plots.
plt.subplots_adjust(left=0.0, bottom=0.08, right=0.96,
                    top=0.96, wspace=0.05)
p0 = ax[0].plot_wireframe(X, Y, Z, rcount=80, ccount=80,
                          color="C1")
p1 = ax[1].plot_surface(X, Y, Z, rcount=50, ccount=50,
                        color="C1")
fig.savefig("./figures/wireframe_surface_plots.pdf")
fig.show()
```

The 3D wireframe and surface plots use the same `meshgrid` function to set up the $x$-$y$ 2D arrays. The `rcount` and `ccount` keywords set the maximum number of rows and columns used to sample the input data to generate the graph.

The viewing angles elevation and azimuth for 3D plots are set with the `view_init` function. If the default values do not give the desired view, you can use your mouse to adjust the view in the plot window manually and then read off the values of the elevation and azimuth, which are displayed in the plot window.

8.10.2 Polar Coordinates

Wireframe and surface plots can also be made using polar coordinates. The first step is to create a polar coordinate mesh, as shown in lines 11–13 in `surface_polar.py` below. This polar mesh is then used to define the height `Z` of the surface plot. Then, the Cartesian coordinates are calculated from the polar coordinate and used for plotting in line 17.

Code: surface_polar.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import seaborn as sns
4
5
6  def f(r, theta):
7      return r**2 * np.exp(-4.0 * ((r * (r - 1.0))**2)) * np.cos(theta)
8
9
10 # Make polar mesh
11 r = np.linspace(0.0, 2.0, 50)
12 th = np.linspace(0.0, 2.0 * np.pi, 50)
13 R, TH = np.meshgrid(r, th)
14
15 Z = f(R, TH)
16 # Translate polar grid to Cartesian grid for plotting
17 X, Y = R * np.cos(TH), R * np.sin(TH)
18
19 fig, ax = plt.subplots(1, 2, figsize=(9.2, 4),
20                        subplot_kw={"projection": "3d"})
21 for i in range(2):
22     ax[i].set_xlabel(r"$x$")
23     ax[i].set_ylabel(r"$y$")
24     ax[i].set_zlabel(r"$f(x,y)$")
25     ax[i].view_init(20, -120)
26
27 # Plot wireframe and surface plots.
28 plt.subplots_adjust(left=0.04, bottom=0.04, right=0.96, top=0.96,
29                     wspace=0.05)
30 p0 = ax[0].plot_wireframe(X, Y, Z, rcount=45, ccount=45, color="C0")
31 # cc = sns.color_palette("YlOrBr_r", as_cmap=True)
32 cc = sns.color_palette("dark:salmon", as_cmap=True)
33 p1 = ax[1].plot_surface(X, Y, Z, rcount=45, ccount=45, cmap=cc)
34 plt.savefig("./figures/surface_polar.pdf")
35 plt.show()
```

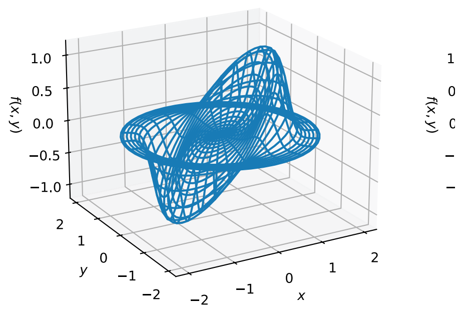


Figure 8.28 Polar surface plots.

These plotting examples are just a sample of many kinds of plots that can be made using Matplotlib. Our purpose is not to exhaustively list all the possibilities here but to introduce you to the Matplotlib package. Online documentation and examples allow you to explore the full range of possibilities.

8.11 EXERCISES

1. Plot the function $y = 3x^2$ for $-1 \le x \le 3$ as a continuous line. Include enough points so that the curve you plot appears smooth. Label the axes x and y.

2. Plot the following function for $-15 \le x \le 15$:

$$y = \frac{\cos x}{1 + \frac{1}{5}x^2}$$

Include enough points so that the curve you plot appears smooth. Draw thin gray lines, one horizontal at $y = 0$ and the other vertical at $x = 0$. Both lines should appear behind the function. Label the axes x and y.

3. Plot the functions $\sin x$ and $\cos x$ *vs.* x on the same plot with x going from $-\pi$ to π. Make sure the limits of the x-axis do not extend beyond the limits of the data. Plot $\sin x$ in orange and $\cos x$ in green, and include a legend to label the two curves. Place the legend within the plot, but such that it does not cover either of the sine or cosine traces. Draw thin gray lines behind the curves, one horizontal at $y = 0$ and the other vertical at $x = 0$.

4. For the data from the data file shown below.

   (a) Read the data into the Python program and plot `y` *vs.* `time` using circles for data points with error bars. Use the data in the `dy` column as the error estimates for the `y` data. Label the horizontal and vertical axes "time (s)" and "position (cm)." Create your plot using the `fig, ax = plt.subplots()` syntax.

   (b) The following function is used to model the data that you were asked to plot in part (a):

$$f(t) = \left(3 + \frac{1}{2}\sin\frac{\pi t}{5}\right) te^{-t/10}$$

   To ascertain how well $f(t)$ models the `y` *vs.* `time` data, plot $f(t)$ as a smooth line on the same plot. To plot $f(t)$ as a smooth line, create a new NumPy t array separate from the `time` data. Make the line for $f(t)$ pass *behind* the data points.

Data: lauren_john_data.txt

```
Data for Exercise 4
Date: 16-Aug-2022
Data taken by Lauren and John

time    y      dy
 1.0    2.94   0.3
 4.5    8.49   0.8
 8.0    9.36   0.4
11.5   11.60   1.0
15.0    9.32   0.9
18.5    7.75   0.5
22.0    8.06   0.4
25.5    5.60   0.3
29.0    4.50   0.3
32.5    4.01   0.4
36.0    2.62   0.4
39.5    1.70   0.6
43.0    2.03   0.4
```

5. Use Matplotlib's function `hist` along with NumPy's functions `random.rand` and `random.randn` to create the histogram graphs shown in Figure 9.7. See Section 9.4 for a description of NumPy's random number functions.

6. The data file below shows data obtained for the displacement (position) *vs.* time of a falling object, together with the estimated uncertainty in the displacement.

Data: falling_ball_data.txt

```
Measurements of fall velocity vs time
Taken by A.P. Crawford and S.M. Torres
19-Sep-22
time (s)    position (m)    uncertainty (m)
 0.0            0.0             0.04
 0.5            1.3             0.12
 1.0            5.1             0.2
 1.5           10.9             0.3
 2.0           18.9             0.4
 2.5           28.7             0.4
 3.0           40.3             0.5
 3.5           53.1             0.6
 4.0           67.5             0.6
 4.5           82.3             0.6
 5.0           97.6             0.7
 5.5          113.8             0.7
 6.0          131.2             0.7
 6.5          148.5             0.7
 7.0          166.2             0.7
 7.5          184.2             0.7
 8.0          201.6             0.7
 8.5          220.1             0.7
 9.0          238.3             0.7
 9.5          256.5             0.7
10.0          275.6             0.8
```

(a) Use these data to calculate the velocity and acceleration (in a Python program `.py` file), together with their uncertainties propagated from the displacement *vs.* time uncertainties. Be sure to calculate time arrays corresponding to the midpoint in time between the two displacements or velocities for the velocity and acceleration arrays, respectively, as described in Section 4.4.3. When adding or subtracting two quantities a and b with uncertainties Δa and Δb, respectively, the uncertainty in either $a + b$ or $a - b$ is $\sqrt{(\Delta a)^2 + (\Delta b)^2}$.

(b) In a single window frame, make three vertically stacked plots of the displacement, velocity, and acceleration *vs.* time. Show the error bars on the different plots. Ensure that all three plots' time axes cover the same range of times (use `sharex`). Why do the relative sizes of the error bars grow progressively greater as one progresses from displacement to velocity to acceleration?

7. Starting from the code that produced Figure 8.10, write a program to produce the plot in Figure 8.29. You will need to use the `sharex` feature introduced in Section 8.2.4, the `subplots_adjust` function to adjust the

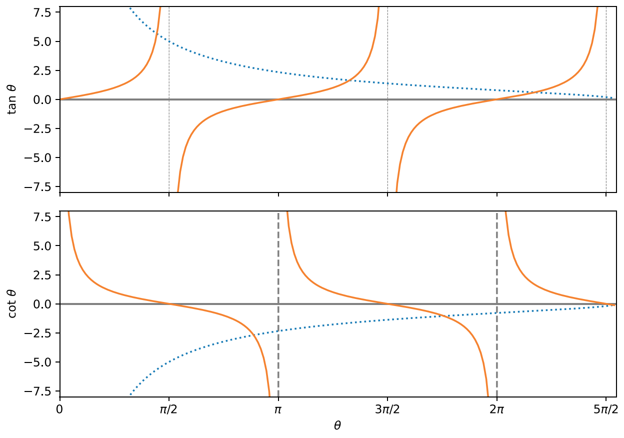


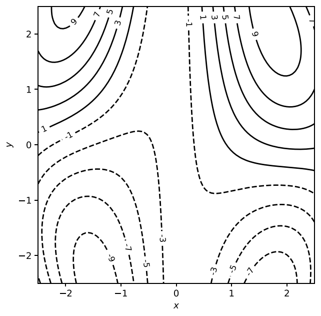
Figure 8.29 Figure for Exercise 7.

space between the two subplots, and the LaTeX syntax introduced in Section 8.7 to produce the math and Greek symbols. Draw the ticks for the x-axes at intervals of $\pi/2$ and label them as shown below. Use `for` loops where possible to avoid repetitive code.

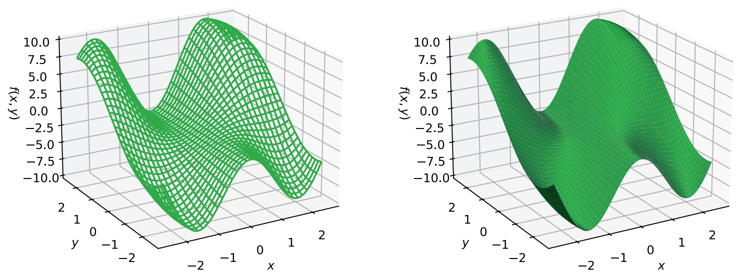
8. Write a program to make a contour plot of the following function over the area defined by $-2.5 < x < 2.5$ and $-2.5 < y < 2.5$.

$$f(x) = 10\cos(\tfrac{1}{3}xy + 5)\sin x - 2\cos x$$

Make the contours at the intervals and label them as shown below. You should also ensure that the x and y axes have the same scaling.



9. Write a program to create the 3D surface plots below of the function $f(x)$ from the previous exercise.



10. Make quiver and streamline plots of the following vector field for a flowing fluid:

$$v_x = by\,, \quad v_y = bx\,.$$

Set $b = 1$. The output should look like Figure 8.30.

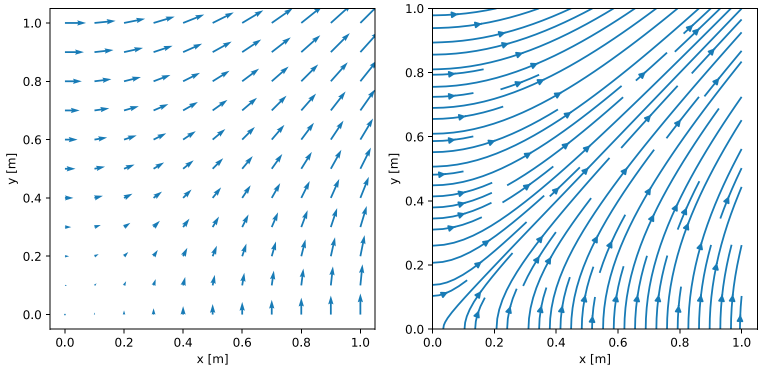


Figure 8.30 Sample output for Exercise 10.

CHAPTER 9

Numerical Routines: SciPy and NumPy

This chapter describes some of the more useful numerical routines available in the SciPy and NumPy packages, most of which are wrappers to well-established numerical routines written in Fortran, C, and C++. ***Special functions*** *like Bessel, Gamma, Error, and many others are covered. Routines for* ***curve fitting*** *to linear, polynomial, and nonlinear functions are introduced.* ***Random number generators*** *are covered. Linear algebra routines are covered, including ones that solve* ***systems of linear equations*** *and* ***eigenvalue problems****. Routines for obtaining* ***solutions to nonlinear equations*** *are introduced, as are routines to perform* ***numerical integration*** *of single and multiple integrals. Routines for obtaining* ***solutions to ODEs*** *(and systems of ODEs) are introduced. Finally, you learn about routines to perform* ***discrete Fourier transforms*** *(FFT algorithm).*

SciPy is a Python library of mathematical routines. Many of the SciPy routines are Python "wrappers," that is, Python routines that provide a Python interface for numerical libraries and routines written in Fortran, C, or C++. Thus, SciPy lets you take advantage of the decades of work that has gone into creating and optimizing numerical routines for science and engineering. Because the Fortran, C, or C++ code that Python accesses is compiled, these routines typically run very fast. Therefore, there is no real downside—no speed penalty—for using Python in these cases.

This chapter introduces many but not all of the SciPy packages. Those covered include special functions, spline fitting, least-squares fitting, random

DOI: 10.1201/9781032673950-9

numbers, linear algebra, finding roots of scalar functions, numerical integration, routines for numerically solving ordinary differential equations (ODEs), and discrete Fourier transforms. This introduction does not include extensive background on the numerical methods employed; that is a topic for another text. You may want to explore more of SciPy's capabilities after having read this introduction.

One final note: SciPy makes extensive use of NumPy arrays, so NumPy should be imported with SciPy.

9.1 SPECIAL FUNCTIONS

SciPy provides many special functions, including Bessel functions (and routines for finding their zeros, derivatives, and integrals), error functions, the gamma function, Mathieu functions, many statistical functions, and many others. Most are contained in the `scipy.special` library, and each has its own special arguments and syntax, depending on the vagaries of the particular function. Polynomial functions, such as Legendre, Laguerre, Hermite, *etc.*, exist in both the SciPy and NumPy libraries. The NumPy versions, found in `numpy.polynomial`, are more numerically efficient and stable than their SciPy equivalents. They also offer more comprehensive capabilities, so we introduce only the NumPy versions. We demonstrate several of them in the code below that produces a plot of the different functions. For more information, you should consult the SciPy website on the `scipy.special` and `numpy.polynomial` libraries.

Code: special_functions.py

```python
import numpy as np
import scipy.special
import matplotlib.pyplot as plt

# create a figure window with subplots
fig, ax = plt.subplots(3, 2, figsize=(9.4, 8.1))

# create arrays for a few Bessel functions and plot them
x = np.linspace(0, 20, 256)
j0 = scipy.special.jv(0, x)  # J_0(x)
j1 = scipy.special.jv(1, x)  # J_1(x)
y0 = scipy.special.yv(0, x)  # Y_0(x)
y1 = scipy.special.yv(1, x)  # Y_1(x)
j0_zeros = scipy.special.jn_zeros(0, 6)
ax[0, 0].plot(x, j0, color="C0", label=r"$J_0(x)$")
ax[0, 0].plot(x, j1, color="C1", dashes=(5, 2), label=r"$J_1(x)$")
ax[0, 0].plot(j0_zeros, np.zeros(j0_zeros.size), "oC3", ms=3)
ax[0, 0].plot(x, y0, color="C2", dashes=(3, 2), label=r"$Y_0(x)$")
ax[0, 0].plot(x, y1, color="C3", dashes=(1, 2), label=r"$Y_1(x)$")
ax[0, 0].axhline(color="grey", lw=0.5, zorder=-1)
```

```
ax[0, 0].set_xlim(0, 20)
ax[0, 0].set_ylim(-1, 1)
ax[0, 0].text(0.5, 0.95, "Bessel", ha="center", va="top",
              transform=ax[0, 0].transAxes)
ax[0, 0].legend(loc="lower right", ncol=2)

# gamma function
x = np.linspace(-3.5, 6., 3601)
g = scipy.special.gamma(x)
g = np.ma.masked_outside(g, -100, 400)
ax[0, 1].plot(x, g, color="C0")
ax[0, 1].set_xlim(-3.5, 6)
ax[0, 1].axhline(color="grey", lw=0.5, zorder=-1)
ax[0, 1].axvline(color="grey", lw=0.5, zorder=-1)
ax[0, 1].set_ylim(-20, 100)
ax[0, 1].text(0.5, 0.95, "Gamma", ha="center",
              va="top", transform=ax[0, 1].transAxes)

# error function
x = np.linspace(0, 2.5, 256)
ef = scipy.special.erf(x)
ax[1, 0].plot(x, ef, color="C0")
ax[1, 0].set_xlim(0, 2.0)
ax[1, 0].set_ylim(0, 1.1)
ax[1, 0].axhline(y=1., color="grey", lw=0.5, dashes=(5, 2), zorder=-1)
ax[1, 0].text(0.5, 0.97, "Error", ha="center",
              va="top", transform=ax[1, 0].transAxes)

# Airy function
x = np.linspace(-15, 4, 256)
ai, aip, bi, bip = scipy.special.airy(x)
ax[1, 1].plot(x, ai, color="C0", label=r"$Ai(x)$")
ax[1, 1].plot(x, bi, color="C1", dashes=(5, 2), label=r"$Bi(x)$")
ax[1, 1].axhline(color="grey", lw=0.5, zorder=-1)
ax[1, 1].axvline(color="grey", lw=0.5, zorder=-1)
ax[1, 1].set_xlim(-15, 4)
ax[1, 1].set_ylim(-0.5, 0.8)
ax[1, 1].text(0.5, 0.95, "Airy", ha="center",
              va="top", transform=ax[1, 1].transAxes)
ax[1, 1].legend(loc="upper left")

# Legendre polynomials
x = np.linspace(-1, 1, 256)
lp0 = np.polynomial.Legendre.basis(0)(x)  # P_0(x)
lp1 = np.polynomial.Legendre.basis(1)(x)  # P_1(x)
lp2 = np.polynomial.Legendre.basis(2)(x)  # P_2(x)
lp3 = np.polynomial.Legendre.basis(3)(x)  # P_3(x)
ax[2, 0].plot(x, lp0, color="C0", label=r"$P_0(x)$")
ax[2, 0].plot(x, lp1, color="C1", dashes=(5, 2), label=r"$P_1(x)$")
ax[2, 0].plot(x, lp2, color="C2", dashes=(3, 2), label=r"$P_2(x)$")
ax[2, 0].plot(x, lp3, color="C3", dashes=(1, 2), label=r"$P_3(x)$")
ax[2, 0].axhline(color="grey", lw=0.5, zorder=-1)
ax[2, 0].axvline(color="grey", lw=0.5, zorder=-1)
ax[2, 0].set_xlim(-1, 1.)
ax[2, 0].set_ylim(-1, 1.1)
ax[2, 0].text(0.5, 0.9, "Legendre", ha="center",
77               va="top", transform=ax[2, 0].transAxes)
78 ax[2, 0].legend(loc="lower right", ncol=2)
79
80 # Laguerre polynomials
81 x = np.linspace(-5, 8, 256)
82 lg0 = np.polynomial.Laguerre.basis(0)(x)  # L_0(x)
83 lg1 = np.polynomial.Laguerre.basis(1)(x)  # L_1(x)
84 lg2 = np.polynomial.Laguerre.basis(2)(x)  # L_2(x)
85 lg3 = np.polynomial.Laguerre.basis(3)(x)  # L_3(x)
86 ax[2, 1].plot(x, lg0, color="C0", label=r"$L_0(x)$")
87 ax[2, 1].plot(x, lg1, color="C1", dashes=(5, 2), label=r"$L_1(x)$")
88 ax[2, 1].plot(x, lg2, color="C2", dashes=(3, 2), label=r"$L_2(x)$")
89 ax[2, 1].plot(x, lg3, color="C3", dashes=(1, 2), label=r"$L_3(x)$")
90 ax[2, 1].axhline(color="grey", lw=0.5, zorder=-1)
91 ax[2, 1].axvline(color="grey", lw=0.5, zorder=-1)
92 ax[2, 1].set_xlim(-5, 7.2)
93 ax[2, 1].set_ylim(-5, 10)
94 ax[2, 1].text(0.5, 0.9, "Laguerre", ha="center",
95               va="top", transform=ax[2, 1].transAxes)
96 ax[2, 1].legend(loc="lower left", ncol=2)
97 plt.tight_layout()
98 plt.savefig("./figures/special_functions.pdf")
99 plt.show()
```

The arguments of the different functions depend, of course, on the nature of the particular function. For example, the first argument of the two types of Bessel functions called in lines 10–13 is the *order* of the Bessel function, and the second argument is the independent variable. For many problems, it is important to know the zeros of different Bessel functions. The SciPy function `scipy.special.jn_zeros(n, m)` returns the first m zeros of the n^{th}-order Bessel function. For example, the first three zeros are obtained as follows:

```
 In[1]: from scipy.special import jn, jn_zeros

 In[2]: jn_zeros(0, 3)
Out[2]: array([2.40482556, 5.52007811, 8.65372791])
```

Line 14 of `special_functions.py` gets the first six zeros of $J_0(x)$ and line 17 plots them as solid red circles in the Bessel functions plot.

In contrast to the Bessel functions, the Gamma and Error functions take one argument each and produce one output. The Airy function takes only one input argument but returns four outputs, which correspond to the two Airy functions, commonly designated $\mathrm{Ai}(x)$ and $\mathrm{Bi}(x)$, and their derivatives $\mathrm{Ai}'(x)$ and $\mathrm{Bi}'(x)$. The plot in Figure 9.1 shows only $\mathrm{Ai}(x)$ and $\mathrm{Bi}(x)$.

The Legendre and Laguerre polynomials are calculated using the `basis` method of the `Legendre` and `Laguerre` classes, respectively, which are part of the `numpy.polynomial` module. The argument of the `basis` method specifies the order n of the polynomial, where $n = 0, 1, 2, \ldots$. The method returns the

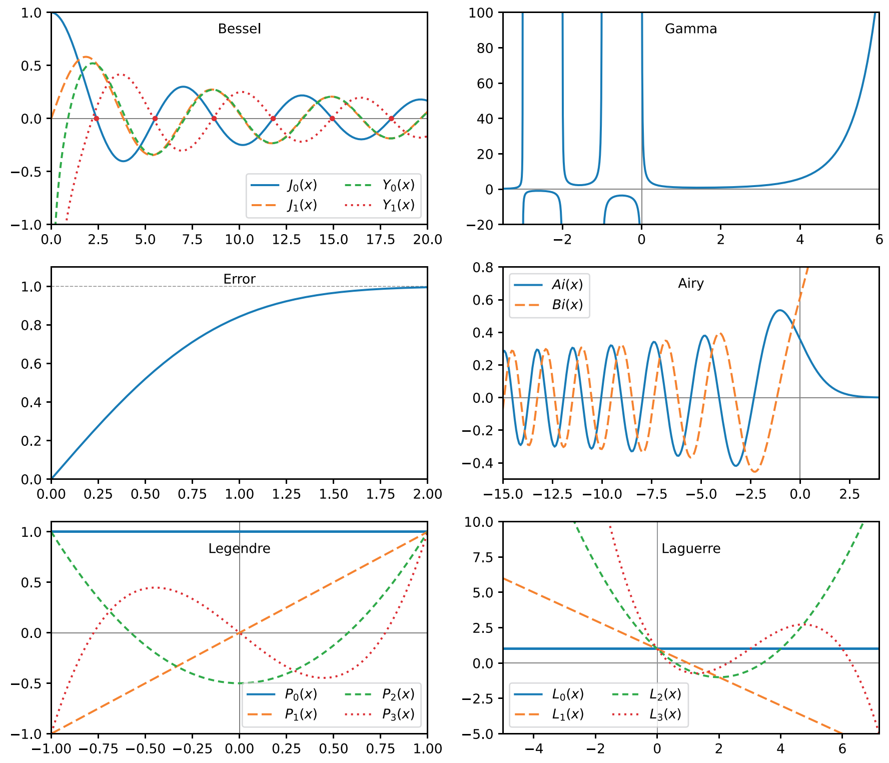


Figure 9.1 Plots of special functions.

values of the polynomial by adding another argument enclosed in its own set of parentheses.

9.1.1 Important Note on Importing SciPy Subpackages

Writing `import scipy` does not import any SciPy subpackage. You must import each subpackage separately, for example, by writing `import scipy.special` or `import scipy.optimize` (see Section 9.3 below). Alternatively, you can write `from scipy import special` or `from scipy import optimize`, *etc*.

9.2 SPLINE FITTING, SMOOTHING, AND INTERPOLATION

In the presentation and analysis of data, spline fitting is used primarily for smoothing data, either to make a visually pleasing guide to the eye or as a prelude to calculating the numerical derivative of a noisy data set. It is also useful for interpolating between data points and for numerically estimating roots (zero crossings) of data.

9.2.1 Interpolating Splines

Interpolating splines connect neighboring data points with a piecewise continuous polynomial of a given order. A linear spline draws a straight line between neighboring data points. A quadratic spline draws a second-order polynomial (quadratic) line between data neighboring points. An n^{th}-order spline curve connects the interval between neighboring data points with an n^{th}-order polynomial. Done properly, an n^{th}-order spline and its first $n-1$ derivatives are piecewise continuous across the entire range of the input data. Be warned, however, that if the data are noisy, the derivatives will be even noisier, so a simple spline may not be what you want if your purpose is to take derivatives of numerical data; you may prefer a smoothing spline, which we discuss in Section 9.2.2. Nevertheless, even if a smoothing spline is what you want to use, it's instructive to understand first how simple splines work.

To see how spline fitting works mathematically, consider the case of a cubic (third order) spline for a set of $m+1$ data points $\{x_i, y_i\}$ between a and b such that $a = x_0 < x_1 \cdots < x_m = b$. In a cubic spline fit, the i^{th} subinterval's endpoints x_i and x_{i+1} are connected by a third-order polynomial of the form

$$S_i(t_i) = a_i + b_i t_i + c_i t_i^2 + d_i t_i^3 \;, \quad \text{for } 0 \le t_i \le 1 \;, \; i = 0, 1, ..., m-1 \;,$$

where $t_i = (x - x_i)/(x_{i+1} - x_i)$ and $x_i \le x \le x_{i+1}$. Each of the m subintervals, with indices $i = 0, \ldots, m-1$, is described by a different cubic polynomial $S_i(t_i)$. As each of the m polynomials has 4 unknown constants, a_i, b_i, c_i, and d_i, there are $4m$ unknowns to be determined to fully determine all the splines. By demanding that $S_i(0) = y_i$ and $S_i(1) = y_{i+1}$ for each subinterval, that is, by demanding that the starting and ending points of each interval give the correct y values, one obtains $2m$ equations for the $4m$ coefficients $\{a_i, b_i, c_i, d_i\}$. By demanding that the first and second derivatives match at the $m-1$ interior points, *i.e.*, $S'_{i-1}(1) = S'_i(0)$ and $S''_{i-1}(1) = S''_i(0)$ for $i = 1, \ldots, m-1$, one obtains another $2(m-1)$ equations, giving a total of $4m-2$ equations for the $4m$ unknown coefficients.[1] The final two conditions needed to obtain a closed set of $4m$ equations are provided by imposing boundary conditions, *i.e.* conditions at the beginning and end of the interval. A common choice is to set the second derivative at the two endpoints to zero, $S''_0(0) = S''_{m-1}(1) = 0$. This

[1] Please note that the derivatives $S', S'' \ldots$ are derivatives with respect to x, not t. For example, $S'_i(0)$ and $S'_i(1)$ are $(dS_i/dx)_{t_i=0}$ and $(dS_i/dx)_{t_i=1}$ evaluated, respectively, at the beginning and end of the i^{th} interval.

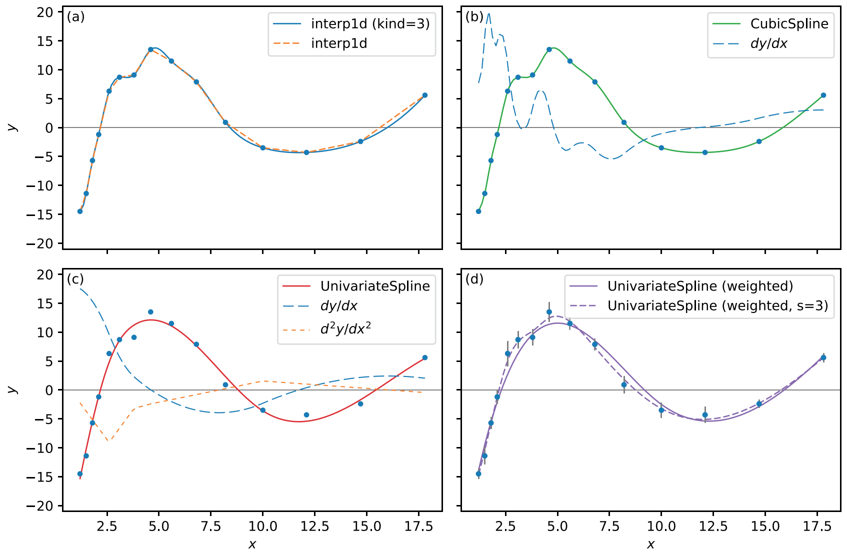


Figure 9.2 Polynomial (cubic) least squares fit to data.

is often called the *natural* boundary condition. But there are other choices as well, such as setting the third derivatives equal to each other at the first and last internal boundaries, *i.e.* $S_0'''(1) = S_1'''(0)$ and $S_{m-2}'''(1) = S_{m-1}'''(0)$. This is often called the *not-a-knot* boundary condition. Periodic boundary conditions and clamping the first or second derivatives at the boundaries are sometimes used, depending on the problem.

To summarize, a cubic spline draws a piecewise continuous curve with continuous first and second derivatives everywhere through an ordered set of $m + 1$ data points (*i.e.*, $x_0 \leq x_1, \ldots, \leq x_{m+1}$). An n^{th}-order spline draws a piecewise continuous curve with continuous derivatives up to the $(n-1)^{\text{th}}$ order through an ordered set of data points.

9.2.1.1 Interpolation

If your purpose in spline fitting is to interpolate between data points, you can use the `interp1d` from the `scipy.interpolate` module. It can perform linear, quadratic, cubic, ...polynomial interpolation between data points in your data set. Please note that using higher than cubic order splines for interpolation is seldom of any value, as we discuss in the next section on cubic splines.

Figure 9.2(a) shows a data set fit with both a cubic (solid blue line) and linear (dashed orange line) spline using the `interp1d` function. The spline fit and plot are made using the program `fit_spline_demo.py` listed below. A spline fit is initialized (*instantiated*, in OOP jargon) for a linear spline on line 9 with the statement `od1 = interp1d(xdata, ydata)`. The first two (positional) arguments of `interp1d` are the arrays `xdata` and `ydata`, which contain the data to be fit. If no other arguments are specified, as in line 9, `interp1d` performs a linear interpolation between data points. To find the interpolated value at any point within the lower and upper limits of the x-data, here from 1.2 to 17.8, you write `od1(x)`, where `x` can be a single number or an array. If you ask it to extrapolate to points outside the data limits, it returns a `ValueError`. As described in the online documentation, you can specify that it returns numerical results for points outside the data limits using the keyword argument `fill_value`. Extrapolating outside the range of the data set is fraught with problems, so beware if you choose to use this feature.

A cubic spline fit is instantiated on line 10 for the same data set but with an additional keyword argument `kind`, which is used to specify the polynomial order of the spline fit. Setting it to a numerical value of 3 (or a string argument of `"cubic"`) tells it to perform a cubic spline.

9.2.1.2 Cubic Splines

Cubic splines are the workhorse of spline fitting. Higher-order spline fits are generally too sensitive to small fluctuations in the data, which can lead to wild fluctuations in the interpolations. The lower-order quadratic splines can have difficulties capturing points of inflection. Linear splines, of course, are not smooth but have the advantage of confining all interpolated values of y to be within the measured values of y.

The `scipy.interpolate` module has a dedicated `CubicSpline` function that is more versatile than the `interp1d` function. It allows the user to specify the spline boundary conditions and can provide the first three derivatives of the spline fit (although only the first and second derivatives are continuous). The `CubicSpline` function is instantiated and used similarly to `scipy.interpolate`. The first two positional arguments of `CubicSpline` are the arrays `xdata` and `ydata`, which contain the data to be fit. The keyword argument `bc_type` sets the boundary conditions. Setting it equal to `"natural"`, `"not-a-knot"`, or `"periodic"` gives the boundary conditions of the same names as described on page 231. Clamped and other boundary conditions can also be specified, as detailed in the online documentation. The `CubicSpline` function is instantiated on line 11 of `fit_spline_demo.py`, which is listed below.

`CubicSpline` has a method `derivative(n)` to find the n^{th} derivative of the spline fit, where `n` can be 1, 2, or 3 (1 is the default). Figure 9.2(b) shows a plot of the cubic spline (solid green curve) and its first derivative (dashed blue curve). The calculated first derivative (slope) fluctuates quite significantly, showing how sensitive it is to any noise in the original data. Because of this sensitivity, people usually use smoothing splines, discussed in the next section, to numerically determine derivatives from noisy data.

Code: fit_spline_demo.py

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d, CubicSpline, UnivariateSpline

# load data from file
xdata, ydata, yunc = np.loadtxt("fit_spline_demo.txt", skiprows=5,
                                unpack=True)
# calculate splines
od1 = interp1d(xdata, ydata)
od3 = interp1d(xdata, ydata, kind=3)
cs = CubicSpline(xdata, ydata, bc_type="natural")
us = UnivariateSpline(xdata, ydata)
usw0 = UnivariateSpline(xdata, ydata, w=1.0 / yunc)
usw3 = UnivariateSpline(xdata, ydata, w=1.0 / yunc, s=3)
# array for plotting splines (many points for smooth plot)
xs = np.linspace(xdata.min(), xdata.max(), 100)

fig, ax = plt.subplots(2, 2, figsize=(9, 6), sharex=True, sharey=True)
ax[0, 0].plot(xs, od3(xs), "-C0", lw=1, zorder=-2,
              label="interp1d (kind=3)")
ax[0, 0].plot(xs, od1(xs), "-C1", lw=1, zorder=-2, dashes=(5, 2),
              label="interp1d")
ax[0, 1].plot(xs, cs(xs), "-C2", lw=1, zorder=-2,
              label="CubicSpline")
ax[1, 0].plot(xs, us(xs), "-C3", lw=1, zorder=-2,
              label="UnivariateSpline")
ax[1, 1].plot(xs, usw0(xs), "-C4", lw=1, zorder=-2,
              label="UnivariateSpline (weighted)")
ax[1, 1].plot(xs, usw3(xs), "-C4", lw=1, zorder=-2, dashes=(5, 2),
              label="UnivariateSpline (weighted, s=3)")
# Calculate and plot derivatives of Cubic and Univariate splines
ax[0, 1].plot(xs, cs.derivative(1)(xs), lw=0.8, zorder=-2,
              dashes=(10, 4), label=r"$dy/dx$")
ax[1, 0].plot(xs, us.derivative(1)(xs), lw=0.8, zorder=-2,
              dashes=(10, 4), label=r"$dy/dx$")
ax[1, 0].plot(xs, us.derivative(2)(xs), lw=0.8, zorder=-2,
              dashes=(4, 4), label=r"$d^2y/dx^2$")
# plot data & annotations on top of spline fits plotted above
abcd = [["(a)", "(b)"], ["(c)", "(d)"]]
for i in range(2):
    for j in range(2):
        ax[i, j].plot(xdata, ydata, "oC0", ms=3, lw=1)
        ax[i, j].legend()
        ax[i, j].text(0.01, 0.98, abcd[i][j], ha="left", va="top",
                      transform=ax[i, j].transAxes)
        ax[i, j].axhline(color="gray", lw=0.5, zorder=-3)
        ax[i, j].set_ylim(-21., 21.)
        if i == 1: ax[i, j].set_xlabel(r"$x$")
        if j == 0: ax[i, j].set_ylabel(r"$y$")
# include error bars for lower right plot (d)
ax[1, 1].errorbar(xdata, ydata, fmt="oC0", ms=3, yerr=yunc,
                  ecolor="gray", elinewidth=1, zorder=-1)

fig.tight_layout()
fig.savefig("figures/fit_spline_demo.pdf")
plt.show()
```

9.2.2 Smoothing Splines

Smoothing splines are often used to process data, sometimes just to draw a smooth curve through noisy data as a guide to the eye. Smoothing splines can also be used to provide estimates of numerical derivatives of data, as taking derivatives with standard interrpolating splines can be very sensitive to even a small amount of noise.

The solid red curve in Figure 9.2(c) shows a cubic smoothing spline for the same data set used for the linear and cubic interpolating splines in panels (a) and (b) of Figure 9.2. The smoothing spline is created in line 12 of `fit_spline_demo.py` with a call to the `scipy.interpolate` routine `UnivariateSpline`. The only arguments are `xdata` and `ydata`. The smoothing spline passes quite close to all the data points but no longer goes through them, yielding a satisfactory smoothing of the data. You need go no further in many cases, and your smoothing task is done. It is possible to exert more control over the smoothing, but you need to understand more about how spline smoothing works, which we discuss next.

Smoothing splines work in part by reducing the number of splines over the data interval, usually quite dramatically. In standard interpolating spline fits, there are $m+1$ data points and m splines, one spline (polynomial) for each interval between neighboring data points. The place where two splines meet is called a *knot*. In an interpolating spline, there is a knot at every data point. By contrast, the smooth spline fit to 15 data points in Figure 9.2(c) has six knots. The positions (and number) of knots can be determined using the `get_knots()` method of `UnivariateSpline` after running `fit_spline_demo.py`:

```
 In[1]: us.get_knots()
Out[1]: array([ 1.2,  2.6,  3.8,  4.6, 10. , 17.8])
```

The `derivative()` method of `UnivariateSpline` can be used to find the first three derivatives of the spline fit. The first and second derivatives, shown by a long dashed blue line and a short dashed orange line, respectively, in Figure 9.2(c), are determined and plotted in lines 34–35 and 36–37 of `fit_spline_demo.py`. As the second derivative of a cubic polynomial is a straight line, the second spline derivative is a piecewise continuous set of straight lines (short dashed orange trace). The knots at $x = 2.6$, 3.8, and 10.0 are evident by the breaks in the slope of the second derivative, which means that the third derivative (not shown) is discontinuous at those knots.

`UnivariateSpline` uses de Boor's algorithm[2] for smoothing splines, which seeks to minimize the objective function

$$\chi_s^2 = p \sum_{i=0}^{m-1} \left(\frac{y_i - \hat{f}(x_i)}{\sigma_i} \right)^2 + (1-p) \int_a^b \left(\hat{f}''(x) \right)^2 dx \,, \tag{9.1}$$

where $\hat{f}(x)$ is the piecewise continuous set of splines that is sought, $\{x_i, y_i\}$ are the data, and σ_i the uncertainty in the y-data. The sum is just the familiar χ^2 objective function Eq. (7.11) used in least squares fitting, which is minimized when $\hat{f}(x)$ most closely approximates $\{y_i\}$, while the second term is minimized when $\hat{f}(x)$ is smoothest, that is, when it has the smallest squared second derivative integrated over the data interval $[a, b]$, which would be straight line. Thus, p sets the trade-off between the closeness and smoothness of the fit. The task is to find the optimal spline coefficients $\{a_i, b_i, c_i, d_i\}$ and value of p that minimize χ_s^2 for a given number and placement of knots.

The de Boor method splits the task of minimizing χ_s^2 into two parts: first, you decide on the number and placement of the knots, and then you adjust p and the spline coefficients to minimize χ_s^2. To determine the number of knots, you specify the approximate value of χ^2 you want. To get $\chi^2 = 0$, you would need m knots, which you can achieve with a normal interpolating (unsmoothed) spline fit with $\hat{f}(x_i) = y_i$ for all the data points. You can get a smoother spline by decreasing the number of knots, increasing the value of $\chi^2 > 0$. So, the procedure is for you to specify χ^2, and the algorithm will choose the number and placement of knots needed to achieve it; this sets the desired degree of smoothness. Once this is known, the routine adjusts the value of p to minimize χ_s^2. Then it recalculates the spline coefficients to minimize χ^2 and χ_s^2 further. This is repeated with p and the spline coefficients being recalculated until the minimum of χ_s^2 is found. If the resulting spline doesn't have the level of smoothness you desire, you can increase or decrease the value of χ^2 that you specify and then rerun the routine. The value of the χ^2 sum that you want to achieve is set by the keyword argument `s` (for *smoothing factor*). If you do not specify `s`, `UnivariateSpline` sets it to $m+1$, the number of

[2] de Boor, Carl. *A practical guide to splines*, New York, Springer, 2001.

data points. The weights are specified by the `w` keyword argument and should be set to the inverse of the uncertainty for each data point: `w = 1.0 / unc`.

For the smoothing fit shown in Figure 9.2(c), only `xdata` and `ydata` are specified in the function call (line 12). Since the weights (`w = 1./unc`) are not specified, `UnivariateSpline` sets them all equal. Figure 9.2(d) shows the results of specifying the weighting (see lines 13 and 14). In this case, when the smoothing factor is not specified, the smoothing factor `s` is set to its default value of `len(w)`, here 15, and a fairly satisfactory smoothing spline is obtained (solid purple line). Reducing the smoothing factor to `s=3` gives a spline (dashed purple curve) that follows the data more closely.

9.2.3 Finding Roots (zero crossings) of Numerical Data

Spline fits can also be used to find roots (zero crossings) of numerical data. The two functions `CubicSpline` and `UnivariateSpline` both have a `root()` method, which can be run after the `fit_spline_demo.py` is run

```
 In[2]: cs.roots()
Out[2]: array([ 2.17780146,  8.41469048, 15.84466433,
26.48818529])
 In[3]: us.roots()
Out[3]: array([ 2.17942427,  8.81132025, 15.4409352 ])
```

Interestingly, the `CubicSpline` routine finds four roots, one outside the domain over which the data are defined. By contrast, the `UnivariateSpline` routine is more disciplined and returns only the three roots within the domain over which the data are defined.

9.3 CURVE FITTING

One of the most important tasks in any experimental science is modeling data and determining how well some theoretical function describes experimental data. In Section 7.5, you saw how this can be done when the theoretical model is a simple straight line. In this section, we explore what NumPy and SciPy have to offer in the way of linear, polynomial, and nonlinear fitting functions.

SciPy has excellent routines for fitting to nonlinear fitting functions, as discussed in Section 9.3.

NumPy and SciPy also have routines for fitting to linear and polynomial functions, which we discuss below. Unfortunately, the organization of these routines can be confusing. Nevertheless, the tools are there, so you can generally get the job done.

9.3.1 Linear Fitting Functions

Suppose you want to fit some data set $\{x_i, y_i\}$ to a linear function

$$f(x) = a + bx \,. \tag{9.2}$$

Your task is to find the values of the y-intercept a and slope b that give the best fit to the data. In general this means finding the values of a and b that minimize $\chi^2(a, b)$, as given by Eq. (7.12), which is reproduced here for reference

$$\chi^2(a, b) = \sum_i \left(\frac{y_i - a - bx_i}{\sigma_i}\right)^2 \,.$$

The values of a and b that minimize $\chi^2(a, b)$ are given by Eqs. (7.10) and (7.14). In addition, you generally would like to know the uncertainty in the values of a and b. These are given by Eq. (7.16). All that is needed is a program that evaluates these equations for a given data set $\{x_i, y_i\}$. Finally, you might like the program to return the value of $\chi^2(a, b)$ or χ_r^2, given by Eq. (7.15); χ_r^2 should be close to 1 if the set of $\{\sigma_i\}$ are known well.

You have many choices for fitting to a straight line using Python. If your background is in the physical sciences and you know the uncertainties $\{\sigma_i\}$ for your data set, you can use the routine `numpy.polynomial.polynomial.polyfit`, which can fit data to a polynomial of any order, including a linear function, a polynomial of order 1. We discuss this routine in the next section on fitting to polynomials. As `numpy.polynomial.polynomial.polyfit` is a bit of overkill for simply fitting a linear function, you might want to write your own routine. We show how this can be done in the program `fit_linear_demo.py` below.

On the other hand, if you come from a background in the social sciences, where you typically do not know the uncertainties $\{\sigma_i\}$ for your data, you can use the routine `scipy.stats.linregress`.

All three of these approaches are illustrated below in the program `fit_linear_demo.py`. A plot showing the fits to data using these approaches is shown in Figure 9.3.

Code: fit_linear_demo.py

```python
import numpy as np
import matplotlib.pyplot as plt
from numpy.polynomial import polynomial as P
import scipy.stats as ss


def linfit(x, y, w=None, full=False):
    """
    Fit to straight line: f(x) =  a + b x
```

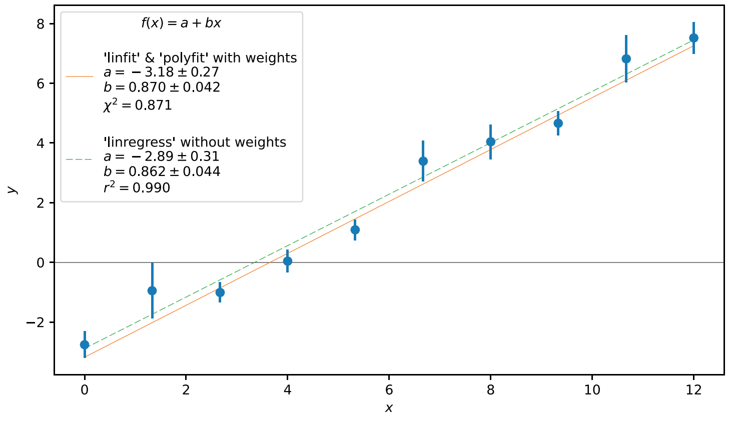


Figure 9.3 Linear least squares fit to data.

```
    Inputs: x, y, and w (1/uncertainty) data arrays with length > 2.
    Ouputs: y-intercept a and slope b of best fit to data.
    """
    if w is None:
        w = w2 = 1.0
        s = x.size
    else:
        w2 = w * w
        s = w2.sum()
    sx = (x * w2).sum()
    sy = (y * w2).sum()
    sxx = (x * x * w2).sum()
    sxy = (x * y * w2).sum()
    d = s * sxx - sx * sx
    a = (sxx * sy - sx * sxy) / d  # y-intercept
    b = (s * sxy - sx * sy) / d  # slope
    if full:
        sig2_slope = s / d
        sig2_yint = sxx / d
        sig2_cross = -sx / d
        cov = np.array([[sig2_yint, sig2_cross],
                        [sig2_cross, sig2_slope]])
        residuals = (((y - a - b * x) * w) ** 2).sum()
        return (a, b), cov, residuals
    else:
        return (a, b)


# loads data from text file
x, y, unc = np.loadtxt("fit_linear_demo.txt", skiprows=5, unpack=True)

# define weighting function from uncertainties
w = 1.0 / unc
# fit data to straight line using linfit fitting routine above
```

```
44 coefs, cov, residuals = linfit(x, y, w=w, full=True)
45 # make y data from fit for plotting
46 lfit = coefs[0] + coefs[1] * x
47 # fit data to using numpy.polynomial.polynomial.polyfit routine
48 coefsP, statsP = P.polyfit(x, y, deg=1, w=w, full=True)
49 # get covariance matrix from deprecated np.polyfit routine
50 c0, cov0 = np.polyfit(x, y, deg=1, w=w, cov="unscaled")
51 # fit data with scipy.stats.linregress routine (no weighting possible)
52 lrout = ss.linregress(x, y)
53 regfit = lrout.intercept + lrout.slope * x
54 
55 # linfit output for display in plot legend
56 a, da = coefs[0], np.sqrt(cov[0, 0])
57 b, db = coefs[1], np.sqrt(cov[1, 1])
58 ltxt = "\n'linfit' & 'polyfit' with weights"
59 ltxt += "\n" + r"$a = {0:0.2f} \pm {1:0.2f}$".format(a, da)
60 ltxt += "\n" + r"$b = {0:0.3f} \pm {1:0.3f}$".format(b, db)
61 ltxt += "\n" + r"$\chi^2 = {0:0.3f}$".format(residuals / (y.size-2))
62 
63 # linregress output for display in plot legend
64 a, da = lrout.intercept, lrout.intercept_stderr
65 b, db = lrout.slope, lrout.stderr
66 rtxt = "\n'linregress' without weights"
67 rtxt += "\n" + r"$a = {0:0.2f} \pm {1:0.2f}$".format(a, da)
68 rtxt += "\n" + r"$b = {0:0.3f} \pm {1:0.3f}$".format(b, db)
69 rtxt += "\n" + r"$r^2 = {0:0.3f}$".format(lrout.rvalue)
70 
71 fig, ax = plt.subplots(figsize=(9, 5))
72 ax.errorbar(x, y, yerr=unc, fmt="o")
73 ax.plot(x, lfit, "-", lw=0.5, zorder=-1, label=ltxt)
74 ax.plot(x, regfit, "-", lw=0.5, zorder=-1, dashes=(10, 4), label=rtxt)
75 ax.axhline(color="gray", lw=0.5, zorder=-2)
76 ax.legend(title=r"$f(x) = a + bx$")
77 ax.set_xlabel(r"$x$")
78 ax.set_ylabel(r"$y$")
79 
80 plt.savefig("figures/fit_linear_demo.pdf")
81 plt.show()
```

We define a function `linfit(x, y, w=None, full=False)` in lines 7–35. The first three inputs are the *x*-*y* data arrays and the *inverse* of the *y*-uncertainties `w`. By default, `w=None`, which means that no weighting is used. If you want to use weighting, you must provide the `w` array. When the fourth argument of the function is `full=False`, the program returns only the optimal fitting parameters `a` and `b`. Setting `full=True` causes the program to return additional parameters described below.

After loading data from a data file in line 39, the weighting array `w` is defined by the reciprocal of `unc`, the *y*-value uncertainties of the data specified in the data file. The `linfit` routine returns three objects: a tuple `coefs`, a 2×2 NumPy array `cov`, and a float `residuals`. `coefs` contains to fitting parameters *a* and *b*. `cov` is the covariance matrix. The square roots of the diagonal elements of the matrix are the uncertainties in the fitting parameters a and b. `residuals` is χ^2 (*not* χ_r^2!!) given by Eq. (7.12). The name `residuals` is chosen for consistency with `numpy.polynomial.polynomial.polyfit`.

The same data are fit with `numpy.polynomial.polynomial.polyfit` in line 48. The arguments are the same as for `linfit` but with one additional argument `deg` that gives the degree of the polynomial to be fit, here 1 for a first-order polynomial or linear fit. There are two outputs, `coefsP` and `statsP`. The first, `coefsP`, gives the fitting parameters a and b, just like `linfit`; the same values are obtained, as expected.

The same data are fit with `scipy.stats.linregress` in line 52. `linregress` returns a dictionary, here given the variable name `lrout`. The intercept a and slope b of the fit are given by `lrout.intercept` and `lrout.slope`; their uncertainties by `lrout.intercept_stderr` and `lrout.stderr`. The fit returned by `linregress` uses unweighted data so the values of a and b and their uncertainties are not the same as for the other routines.

Figure 9.3 shows the data and the fits. For this data set, the main effect of the weighting of data is to pull down the fit, which reflects the relatively small error bars of the data points at $x = 2.7, 4.0, 5.3$, and 9.3.

9.3.2 Polynomial Fitting Functions

As noted above, `numpy.polynomial.polynomial.polyfit`[3] is available for fitting to polynomials of the form

$$p(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n \ . \tag{9.3}$$

Because the fitting function $p(x)$ is linear in the fitting parameters $\{c_j\}$, the values of the coefficients can be directly calculated as long as the problem isn't singular. Thus, the routine returns the coefficients $\{c_j\}$ that minimize $\chi^2(\{c_j\})$, where

$$\chi^2(\{c_j\}) = \sum_i \left(\frac{y_i - p(x_i; \{c_j\})}{\sigma_i} \right)^2 \ . \tag{9.4}$$

You supply the routine with a data set $\{x_i, y_i\}$ and optionally the weight of each data point $w_i = 1/\sigma_i$, where σ_i are the uncertainties in the data. It returns the optimal coefficients $\{c_j\}$ that minimize $\chi^2(\{c_j\})$.

[3]Please be aware that `numpy.polynomial.polynomial.polyfit` is different from the older `numpy.polyfit` routine, which should no longer be used.

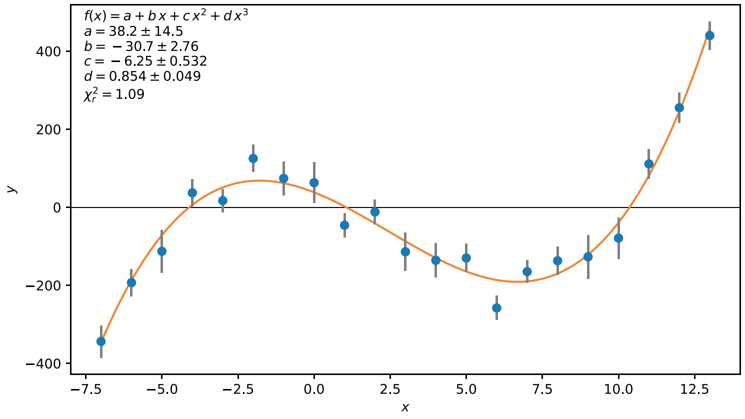


Figure 9.4 Polynomial (cubic) least squares fit to data.

Code: fit_poly_demo.py

```
import numpy as np
import matplotlib.pyplot as plt
import numpy.polynomial.polynomial as poly

# loads data from text file
xdata, ydata, yunc = np.loadtxt("fit_poly_demo.txt", skiprows=5,
                                unpack=True)
# performs cubic (deg=3) polynomial fit to data
coefs, stats = poly.polyfit(xdata, ydata, deg=3, w=1./yunc, full=True)
# use old polyfit to get covariance matrix that poly.polyfit lacks
c, cov = np.polyfit(xdata, ydata, deg=3, w=1./yunc, cov=True)
# array for plotting fit
xfit = np.linspace(xdata.min(), xdata.max(), 100)

fig, ax = plt.subplots(figsize=(9, 5))
ax.errorbar(xdata, ydata, fmt="oC0", yerr=yunc, ecolor="gray")
ax.plot(xfit, poly.polyval(xfit, coefs), "-C1", zorder=-1)
ax.axhline(lw=0.5, color="k", zorder=-2)
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$y$")
# get uncertainties from covariance matrix
coef_unc = [np.sqrt(cov[j, j]) for j in range(cov.shape[0])]
# invert order of coefficients for consistency with new polynomials
coef_unc = coef_unc[::-1]
# Print fit results on plot
alphabet = "abcd"
fitxt = r"$f(x)=a+b\,x+c\,x^2+d\,x^3$"
for i in range(stats[1]):
    st = "\n" + r"${0} = {1:0.3g} \pm {2:0.3g}$"
    fitxt += st.format(alphabet[i], coefs[i], coef_unc[i])
chisq_red = stats[0][0]/(ydata.size-stats[1])
```

The routine `fit_poly_demo.py` illustrates the use of `polyfit` to fit $\{x_i, y_i\}$ data, which include uncertainties $\{\sigma_i\}$, to a cubic third order polynomial of the form $p(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3$. The result of the fit is shown in Figure 9.4. The routine, called on line 9, returns the fitting parameters, here c_0, c_1, c_2, and c_3 in the array `coefs`. Setting the argument `full=True` causes another argument, `stats`, to be returned. `stats` contains, in order, `residuals`, which is the value of χ^2, the rank of the scaled Vandermonde matrix, here 4, as there are four coefficients for a third order polynomial, and two more diagnostic parameters. The reduced value of chi-squared is given by χ^2 divided by the number of degrees of freedom in the fit, corresponding to the number of data points minus the number of fitting parameters. This is calculated in line 31 from `stats` and displayed on the plot.

Note that the `numpy.polynomial.polynomial.polyval()` function is used to evaluate the fitted polynomial on line 17. The first argument is the array of x values, and the second is the coefficients of the fitted polynomial.

9.3.3 Nonlinear Fitting Functions

When χ^2 depends linearly on the fitting parameters, as it does for the linear and polynomial functions considered in Section 9.3.1 and Section 9.3.2, there is a solution that can calculated using straightforward linear algebra, provided the problem isn't singular. This is what the routines that were introduced in these previous sections do. However, when χ^2 depends nonlinearly on the fitting parameters, no algorithm is guaranteed to find the optimal set of fitting parameters. Instead of calculating the solution, one must *search* for the solution.

Consider, for example, the problem of fitting a function $f(x_i; a, b)$ that is nonlinear in the fitting parameters a and b to a data set $\{x_i, y_i\}$, with uncertainties in $\{y_i\}$ of $\{\sigma_i\}$. To do so, we look for the minimum in

$$\chi^2(a, b) = \sum_i \left(\frac{y_i - f(x_i)}{\sigma_i} \right)^2 . \tag{9.5}$$

Once the data set, uncertainties, and fitting function are specified, $\chi^2(a, b)$ is simply a function of a and b. You can picture the function $\chi^2(a, b)$ as a landscape with peaks and valleys: as a and b vary, $\chi^2(a, b)$ rises and falls. The basic idea of all nonlinear fitting routines is to start with some initial guesses for the fitting parameters, here a and b, and by scanning the local landscape of $\chi^2(a, b)$, move toward and find values of a and b that minimize $\chi^2(a, b)$.

There are several different methods for finding the minimum in χ^2 for nonlinear fitting problems. The most widely used method is the

Levenberg-Marquardt method. In fact, the Levenberg-Marquardt method is a combination of two other methods, the *steepest descent* (or gradient) method and *parabolic extrapolation*. Roughly speaking, when the values of a and b are not too near their optimal values, the gradient descent method determines in which direction in (a, b)-space the function $\chi^2(a, b)$ decreases most quickly—the direction of steepest descent—and then changes a and b to move in that direction. This method is very efficient unless a and b are very near their optimal values. Parabolic extrapolation is more efficient near the optimal values of a and b. Therefore, as a and b approach their optimal values, the Levenberg-Marquardt method gradually changes to the parabolic extrapolation method, which approximates $\chi^2(a, b)$ by a Taylor series second-order in a and b and then computes directly the analytical minimum of the Taylor series approximation of $\chi^2(a, b)$. This method is only good if the second-order Taylor series provides a good approximation of the local minimum of $\chi^2(a, b)$. That is why parabolic extrapolation only works well very near the minimum in $\chi^2(a, b)$.

Before illustrating the Levenberg-Marquardt method, we need to make one cautionary remark: the Levenberg-Marquardt method can fail if the initial guesses of the fitting parameters are too far from the desired solution. This problem becomes more serious the greater the number of fitting parameters. Thus, providing reasonable initial guesses for the fitting parameters is essential. Usually, this is not a problem, as it is clear from the physical situation of a particular experiment what reasonable values of the fitting parameters are. But beware!

The `scipy.optimize` module provides routines that implement the Levenberg-Marquardt nonlinear fitting method. The most useful of these is called `scipy.optimize.curve_fit`, which we demonstrate here. The function call is

```
import scipy.optimize
[...insert code here defining the data & fitting function...]
scipy.optimize.curve_fit(f, xdata, ydata, p0=None,
sigma=None, **kwargs)
```

The arguments of `curve_fit` are as follows:

`f(xdata, a, b, ...)`: is the fitting function, where `xdata` is the data for the independent variable and `a, b, ...` are the fitting parameters, however many there are, listed as separate arguments. Obviously, `f(xdata, a, b, ...)` should return the y value of the fitting function.

`xdata`: is the array containing the x data.

`ydata`: is the array containing the y data.

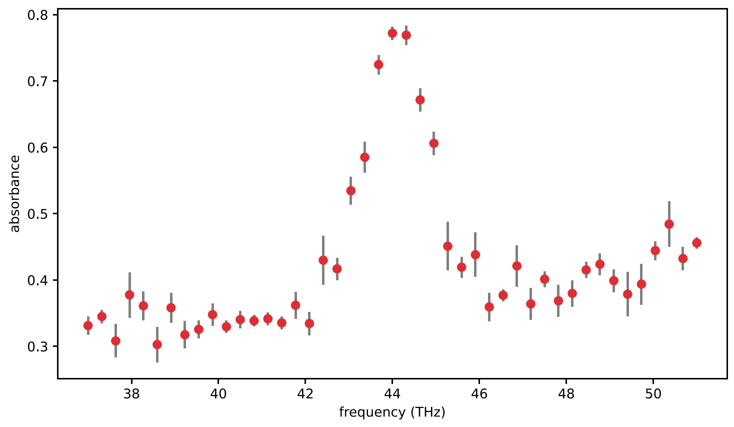


Figure 9.5 Data to be fit with nonlinear fitting function.

`p0`: is a tuple containing the initial guesses for the fitting parameters. The guesses for the fitting parameters are set equal to 1 if they are left unspecified. It is almost never a good idea not to specify the initial guesses for the fitting parameters. Failing to specify good guesses for the fitting parameters can cause the routine to converge to a local minimum of χ^2 with incorrect values of the fitting parameters.

`sigma`: is the array containing the uncertainties in the y data.

`**kwargs`: are keyword arguments that can be passed to the fitting routine `scipy.optimize.leastsq` that `curve_fit` calls. These are usually left unspecified.

We demonstrate the use of `curve_fit` to fit the data plotted in Figure 9.5. We model the data with the fitting function that consists of a quadratic polynomial background with a Gaussian peak:

$$s(f) = a + bf + cf^2 + Pe^{-\frac{1}{2}[(f-f_p)/f_w]^2} \tag{9.6}$$

Lines 7–9 define the fitting function. Note that the independent variable `f` is the first argument, which is followed by the six fitting parameters a, b, c, P, f_p, and f_w.

To fit the data with $s(f)$, we need good estimates of the fitting parameters. Setting $f = 0$, the data suggests that $a \approx 0.3$. Examining the data in Figure 9.5 suggests that the slope b and curvature c of the baseline are small, so we'll set them both to zero as an initial guess. The amplitude of the peak above the

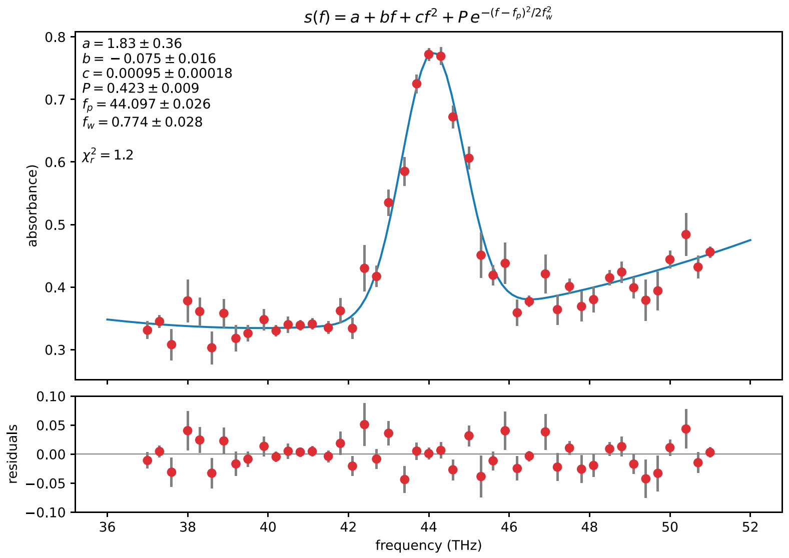


Figure 9.6 Fit to Gaussian with quadratic polynomial background.

baseline is $P \approx 0.5$. The peak is centered at $f_p \approx 44$ THz, while the width of the peak is about $f_w \approx 1$ THz. We use these estimates to set the initial guesses of the fitting parameters in lines 17 and 18 in the following code.

The function that performs the Levenberg-Marquardt algorithm, `scipy.optimize.curve_fit`, is called in lines 21–22 with the output set equal to the one- and two-dimensional arrays `nlfit` and `nlpcov`, respectively. The array `nlfit`, which gives the optimal values of the fitting parameters, is unpacked in line 24. The square root of the diagonal of the two-dimensional array `nlpcov`, which estimates the uncertainties in the fitting parameters, is unpacked in lines 27–28 using a list comprehension.

The rest of the code plots the data, the fitting function using the optimal values of the fitting parameters found by `scipy.optimize.curve_fit`, and the values of the fitting parameters and their uncertainties. See Figure 9.6.

Code: fit_nonlin_demo.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import scipy.optimize
4
5
6  # define fitting function
7  def gauss_poly_base(f, a, b, c, P, fp, fw):
8      baseline = a + f * (b + f * c)   # quadratic polynomial
```

```
    return baseline + P * np.exp(-0.5 * ((f - fp) / fw) ** 2)


# read in spectrum from data file
# f=frequency, s=signal, ds=s uncertainty
f, s, ds = np.loadtxt("fit_nonlin_demo.txt", skiprows=5, unpack=True)

# initial guesses for fitting parameters
a0, b0, c0 = 0.3, 0.0, 0.0
P0, fp0, fw0 = 0.5, 44.0, 1.0

# fit data using SciPy"s Levenberg Marquart method
nlfit, nlpcov = scipy.optimize.curve_fit(
    gauss_poly_base, f, s, p0=[a0, b0, c0, P0, fp0, fw0], sigma=ds)
# unpack fitting parameters
a, b, c, P, fp, fw = nlfit
# unpack uncertainties in fitting parameters from
# the diagonal of the covariance matrix
da, db, dc, dP, dfp, dfw = [np.sqrt(nlpcov[j, j])
                            for j in range(nlfit.size)]

# create fitting function from fitted parameters
f_fit = np.linspace(36.0, 52.0, 128)
s_fit = gauss_poly_base(f_fit, a, b, c, P, fp, fw)

# Calculate residuals and reduced chi squared
resids = s - gauss_poly_base(f, a, b, c, P, fp, fw)
redchisqr = ((resids / ds) ** 2).sum() / float(f.size - 6)

# Create figure window to plot data
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(9.5, 6.5), sharex=True,
    gridspec_kw={"height_ratios": [6, 2], "hspace": 0.07})
# Top plot: data and fit
ax1.plot(f_fit, s_fit, "-C0")
ax1.errorbar(f, s, yerr=ds, fmt="oC3", ecolor="gray")
ax1.set_ylabel("absorbance)")
# Write values of fitting parameters on plot
tx = r"$a = {0:0.2f} \pm {1:0.2f}$".format(a, da)
tx += "\n" + r"$b = {0:0.3f} \pm {1:0.3f}$".format(b, db)
tx += "\n" + r"$c = {0:0.5f} \pm {1:0.5f}$".format(c, dc)
tx += "\n" + r"$P = {0:0.3f} \pm {1:0.3f}$".format(P, dP)
tx += "\n" + r"$f_p = {0:0.3f} \pm {1:0.3f}$".format(fp, dfp)
tx += "\n" + r"$f_w = {0:0.3f} \pm {1:0.3f}$".format(fw, dfw)
tx += "\n\n" + r"$\chi_r^2 = {0:0.2g}$".format(redchisqr)
ax1.text(0.01, 0.98, tx, va="top", ha="left", transform=ax1.transAxes)
ax1.set_title(r"$s(f)=a+bf+cf^2+P\,e^{-(f-f_p)^2/2f_w^2}$")
# Bottom plot: residuals
ax2.errorbar(f, resids, yerr=ds, ecolor="gray", fmt="oC3")
ax2.axhline(color="gray", lw=0.5, zorder=-1)
ax2.set_xlabel("frequency (THz)")
ax2.set_ylabel("residuals")
ax2.set_ylim(-0.1, 0.1)
fig.savefig("figures/fit_nonlin_demo.pdf")
plt.show()
```

The above code also plots the difference between the data and fit, known as the *residuals*, in the subplot below the plot of the data and fit. Plotting the residuals in this way gives a graphical representation of the goodness of the fit. The fit would seem to be a good fit to the extent that the residuals vary randomly about zero and do not show any overall upward or downward curvature or any long wavelength oscillations.

Finally, note that we invoked the Matplotlib `gridspec` function using the `gridspec_kw` keyword argument in line 40 to create the two subplots with different heights; we also use it to specify the space between the two plots. More details about the `gridspec` package can be found at the Matplotlib website.

9.4 RANDOM NUMBERS

Random numbers are widely used in science and engineering computations. They can be used to simulate noisy data, to model physical phenomena like the distribution of velocities of molecules in a gas, or to act like the roll of dice in a game. There are even methods for numerically evaluating multi-dimensional integrals using random numbers.

The basic idea of a random number generator is that it should be able to produce a sequence of numbers that are distributed according to some predetermined distribution function. NumPy provides many such random number generators in its library. Here, we introduce four functions from the `numpy.random` library: `random`, `standard_normal`, `poisson`, and `integers`.

Python also has a module on random numbers, called `random`, that is distinct from NumPy's module. Indeed, we have used it a few times earlier in this text. However, for most scientific numerical work, you should use the NumPy module as it interfaces better with the rest of NumPy and SciPy.

9.4.1 Initializing NumPy's Random Number Generator

Before NumPy can generate any random numbers, you need to initialize its random number generator, which you can do like this:

```
In[1]: import numpy as np
In[2]: rng = np.random.default_rng()
```

Once this is done, you can generate sequences of random numbers with well-defined distributions using various functions in the `numpy.random` library.

9.4.2 Uniformly Distributed Random Numbers

The `random(n)` function creates an array of `n` floats uniformly distributed on the interval from 0 to 1.

```
In[3]: rng.random()
```

```
Out[3]: 0.5742573549114448

 In[4]: rng.random(5)
Out[4]: array([0.39217609, 0.49224774, 0.63899968,
                0.85669782, 0.18048322])
```

A single random number is generated if `random` has no argument. Otherwise, the argument specifies the number of random numbers created and the size of the array that holds them.

If you want random numbers uniformly distributed over some other interval, say from a to b, you can do that simply by stretching the interval to have a width of $b - a$ and displacing the lower limit from 0 to a. The following statements produce an array of 12 random numbers uniformly distributed from 20 to 30:

```
 In[5]: a, b = 20, 30

 In[6]: (b-a) * rng.random(12) + a
Out[6]: array([21.24773546, 26.95406295, 24.92558953,
                20.70140004, 27.65198469, 29.86126406,
                29.73395443, 27.45396695, 22.49217888,
                23.50130919, 24.8918542 , 29.64653862])
```

9.4.3 Normally Distributed Random Numbers

The function `standard_normal(n)` produces a *normal* or *Gaussian* distribution of `n` random numbers with a mean of 0 and a standard deviation of 1. That is, they are distributed according to

$$P(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} .$$

The top two panels of Figure 9.7 show histograms for the distributions of 10,000 random numbers generated by the `rng.random()` and `rng.standard_normal()` functions. As advertised, the `rng.random()` function produces an array of random numbers uniformly distributed between 0 and 1, while the `rng.standard_normal()` function produces an array of random numbers that follows a distribution of mean 0 and standard deviation 1.

If you want random numbers with a Gaussian distribution of width σ centered about x_0, you can stretch the interval by a factor of σ and displace it by x_0. The following code produces 12 random numbers normally distributed around 15 with a width of 10:

```
 In[7]: x0, sigma = 15, 10
```

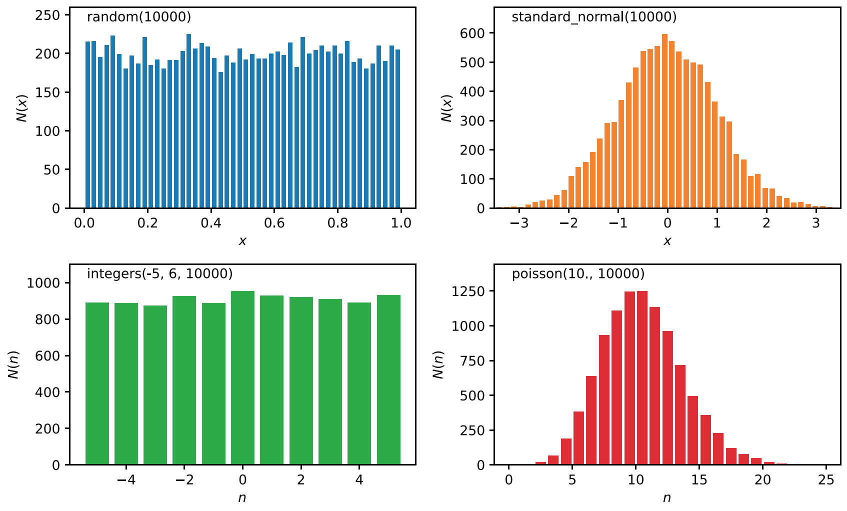


Figure 9.7 Random number distributions (unnormalized).

```
In [8]: sigma * rng.standard_normal(12) + x0
Out[8]: array([12.06094094, 17.74044565,  9.88128787,
               22.01051136,  6.69699366, 13.46305006,
               26.54998687, 35.63815862,  3.26284585,
               26.05653199, 24.54614562,  7.37724636])
```

9.4.4 Random Distribution of Integers

The function `rng.integers(low, high, n)` produces a uniform random distribution of `n` integers between `low` (inclusive) and `high` (exclusive). For example, we can simulate a dozen rolls of a single die with the following statement:

```
In [9]: rng.integers(1, 7, 12)
Out[9]: array([6, 2, 1, 5, 4, 6, 3, 6, 5, 4, 6, 2])
```

9.4.5 Poisson Distribution of Random Integers

The function `poisson(lam, n)` produces a *Poisson* distribution of `n` random integers with a mean of `lam`. That is, they are distributed according to

$$P(n) = \frac{\lambda^n e^{-\lambda}}{n!} \,,$$

where λ is the mean and variance of the distribution. For example, you can generate a Poisson distribution of 24 integers with a mean of 5.5 as follows:

TABLE 9.1 Selected random number functions from `numpy.random`.

Function call	Output
`random(n)`	`n` random numbers uniformly distributed from 0 to 1
`standard_normal(n)`	`n` random numbers normally distributed with 0 mean and width 1
`poisson(lam, n)`	`n` random integers Poisson distributed with `lam` mean and variance
`integers(low, high, n)`	`n` random integers from low (inclusive) to high (exclusive)
`exponential(beta, n)`	`n` random numbers exponentially distributed with `beta` mean
`lognormal(avg, sig, n)`	`n` random numbers lognormally distributed with an underlying normal distribution of mean `avg` and standard deviation `sig`
`shuffle(a)`	shuffles (randomly reorders) the elements of an array in place

```
 In[10]: rng.poisson(5.5, 24)
Out[10]: array([9, 3, 5, 3, 6, 4, 4, 7, 4, 7, 8, 5,
                9, 2, 2, 4, 8, 5, 4, 5, 4, 7, 8, 5])
```

The bottom two panels of Figure 9.7 show histograms for distributions of 10,000 random numbers generated by the `rng.integers()` and `rng.poisson()` functions.

The `numpy.random` library has a plethora of other routines for dealing with random numbers and you are encouraged to check them out. A few of them, including those introduced above, are listed in Table 9.1.

9.5 LINEAR ALGEBRA

Python's mathematical libraries, NumPy and SciPy, have extensive tools for numerically solving problems in linear algebra. Here, we focus on two common problems in scientific and engineering settings: (1) solving a system of linear equations and (2) eigenvalue problems. In addition, we show how to perform several other basic computations, such as finding the determinant of a matrix, matrix inversion, and *LU* decomposition. The SciPy package for linear algebra is called `scipy.linalg`.

9.5.1 Basic Computations in Linear Algebra

SciPy has several routines for performing basic operations with matrices. The determinant of a matrix is computed using the `scipy.linalg.det` function:

```
 In[1]: import numpy as np
 In[2]: import scipy.linalg
 In[3]: a = np.array([[-2, 3], [4, 5]])
 In[4]: a
Out[4]: array([[-2,  3],
               [ 4,  5]])

 In[5]: scipy.linalg.det(a)
Out[5]: -22.0
```

The inverse of a matrix is computed using the `scipy.linalg.inv` function, while the product of two matrices is calculated using the NumPy `dot` function:

```
 In[6]: b = scipy.linalg.inv(a)

 In[7]: b
Out[7]: array([[-0.22727273, 0.13636364],
               [ 0.18181818, 0.09090909]])

 In[8]: np.dot(a, b)
Out[8]: array([[ 1.000000000e+00,  2.775557562e-17],
               [-5.551115123e-17,  1.000000000e+00]])
```

9.5.2 Solving Systems of Linear Equations

Solving systems of equations is nearly as simple as constructing a coefficient matrix and a column vector. Suppose you have the following system of linear equations to solve:

$$\begin{aligned} 2x_1 + 4x_2 + 6x_3 &= 4 \\ x_1 - 3x_2 - 9x_3 &= -11 \\ 8x_1 + 5x_2 - 7x_3 &= 1 \end{aligned} \tag{9.7}$$

The first task is to recast this set of equations as a matrix equation of the form $\mathsf{A}\mathbf{x} = \mathbf{b}$. In this case, we have:

$$\mathsf{A} = \begin{pmatrix} 2 & 4 & 6 \\ 1 & -3 & -9 \\ 8 & 5 & -7 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 4 \\ -11 \\ 1 \end{pmatrix}. \tag{9.8}$$

Next we construct the NumPy arrays reprenting the matrix A and the vector **b**:

```
In[9]: A = array([[2, 4, 6], [1, -3, -9], [8, 5, -7]])
In[10]: b = array([4, -11, 2])
```

Finally we use the SciPy function `scipy.linalg.solve` to find x_1, x_2, and x_3:

```
In[11]: scipy.linalg.solve(A, b)
Out[11]: array([ -8.91304348, 10.2173913 , -3.17391304])
```

which gives the results: $x_1 = -8.91304348$, $x_2 = 10.2173913$, and $x_3 = -3.17391304$. Of course, you can get the same answer by noting that $\mathbf{x} = \mathsf{A}^{-1}\mathbf{b}$. Following this approach, we can use *scipy.linalg.inv* introduced in the previous section:

```
In[12]: Ainv = scipy.linalg.inv(A)

In[13]: dot(Ainv, b)
Out[13]: array([ -8.91304348, 10.2173913 , -3.17391304])
```

which is the same answer we obtained using `scipy.linalg.solve`. Using `scipy.linalg.solve` is faster and numerically more stable than using $\mathbf{x}= \mathsf{A}^{-1}\mathbf{b}$, so it is the preferred method for solving systems of equations.

What happens if the equations are not all linearly independent? For example, if the matrix A is given by

$$\mathsf{A} = \begin{pmatrix} 2 & 4 & 6 \\ 1 & -3 & -9 \\ 1 & 2 & 3 \end{pmatrix} \tag{9.9}$$

where the third row is a multiple of the first row. Let's try it out and see what happens. First, we change the bottom row of the matrix A and then try to solve the system as before.

```
In[14]: A[2] = array([1, 2, 3])

In[15]: A
Out[15]: array([[ 2,  4,  6],
                [ 1, -3, -9],
                [ 1,  2,  3]])

In[16]: scipy.linalg.solve(A,b)
LinAlgError: Singular matrix

In[17]: Ainv = scipy.linalg.inv(A)
LinAlgError: Singular matrix
```

Whether we use `scipy.linalg.solve` or `scipy.linalg.inv`, SciPy raises an error because the matrix is singular.

9.5.3 Eigenvalue Problems

One of the most common problems in science and engineering is the eigenvalue problem, which, in matrix form, is written as

$$\mathsf{A}\mathbf{x} = \lambda\mathbf{x} \tag{9.10}$$

where `A` is a square matrix, $\mathbf{x}$ is a column vector, and λ is a scalar (number). Given the matrix `A`, the problem is to find the set of eigenvectors $\mathbf{x}$ and their corresponding eigenvalues λ that solve this equation.

We can solve eigenvalue equations like this using the SciPy routine `scipy.linalg.eig`. The output of this function is an array whose entries are the eigenvalues and a matrix whose rows are the eigenvectors. Let's return to the matrix we were using previously and find its eigenvalues and eigenvectors.

```
 In [18]: A
Out[18]: array([[ 2,  4,  6],
                [ 1, -3, -9],
                [ 8,  5, -7]])

 In [19]: lam, evec = scipy.linalg.eig(A)

 In [20]: lam
Out[20]: array([ 2.40995356+0.j, -8.03416016+0.j,
                 -2.37579340+0.j])

 In [21]: evec
Out[21]: array([[-0.77167559, -0.52633654,  0.57513303],
                [ 0.50360249,  0.76565448, -0.80920669],
                [-0.38846018,  0.36978786,  0.12002724]])
```

The first eigenvalue and its corresponding eigenvector are given by

```
 In [22]: lam[0]
Out[22]: (2.4099535647625494+0j)

 In [23]: evec[:,0]
Out[23]: array([-0.77167559,  0.50360249, -0.38846018])
```

We can check that they satisfy the $\mathsf{A}\mathbf{x} = \lambda\mathbf{x}$:

```
 In [24]: dot(A, evec[:,0])
Out[24]: array([-1.85970234,  1.21365861, -0.93617101])

 In [25]: lam[0] * evec[:,0]
Out[25]: array([-1.85970234+0.j,  1.21365861+0.j,
                 -0.93617101+0.j])
```

Thus we see by direct substitution that the left and right sides of $A\mathbf{x} = \lambda\mathbf{x}$: are equal. In general, the eigenvalues can be complex, so their values are reported as complex numbers.

9.5.3.1 Generalized Eigenvalue Problem

The `scipy.linalg.eig` function can also solve the *generalized* eigenvalue problem

$$A\mathbf{x} = \lambda B\mathbf{x} \tag{9.11}$$

where B is a square matrix with the same size as A. Suppose, for example, that we have

```
In[26]: A = array([[2, 4, 6], [1, -3, -9], [8, 5, -7]])
Out[26]: B = array([[5, 9, 1], [-3, 1, 6], [4, 2, 8]])
```

Then, we can solve the generalized eigenvalue problem by entering `B` as the optional second argument to `scipy.linalg.eig`

```
In[27]: lam, evec = scipy.linalg.eig(A, B)
```

The solutions are returned in the same fashion as before, as an array `lam` whose entries are the eigenvalues and a matrix `evac` whose rows are the eigenvectors.

```
In[28]: lam
Out[28]: array([-1.36087907+0.j, 0.83252442+0.j,
                -0.10099858+0.j])

In[29]: evec
Out[29]: array([[-0.0419907 , -1.  , 0.93037493],
                [-0.43028153, 0.17751302, -1. ],
                [ 1.  , -0.29852465, 0.4226201 ]])
```

9.5.3.2 Hermitian and Banded Matrices

SciPy has a specialized routine for solving eigenvalue problems for Hermitian (or real symmetric) matrices. The routine for Hermitian matrices is `scipy.linalg.eigh`. It is more efficient (faster and uses less memory) than `scipy.linalg.eig`. The basic syntax of the two routines is the same, although some of the *optional* arguments are different. Both routines can solve generalized as well as standard eigenvalue problems.

SciPy has a specialized routine `scipy.linalg.eig_banded` for solving eigenvalue problems for real symmetric or complex Hermitian banded matrices. When there is a specialized routine for handling a particular kind of matrix, you should use it; it is almost certain to run faster, use less memory, and give more accurate results.

9.6 SOLVING NONLINEAR EQUATIONS

SciPy has many different routines for numerically solving nonlinear equations or systems of nonlinear equations. Here, we will introduce only a few of the simpler routines suitable for the most common types of nonlinear equations.

9.6.1 Single Equations of a Single Variable

Solving a single nonlinear equation is enormously simpler than solving a system of nonlinear equations, so that is where we start. Solving nonlinear equations can be tricky, so you need to have a good sense of the behavior of the function you are trying to solve. A good way to do this is to plot the function over the domain of interest before trying to find the solutions. This will assist you in finding the solutions you seek and avoiding spurious solutions.

We begin with a concrete example. Suppose we want to find the solutions to the equation

$$\tan x = \sqrt{(8/x)^2 - 1}\,. \tag{9.12}$$

Plots of $\tan x$ and $\sqrt{(8/x)^2 - 1}$ *vs.* x are shown in the top plot of Figure 8.10, albeit with x replaced by θ. The solutions to this equation are those x values where the two curves $\tan x$ and $\sqrt{(8/x)^2 - 1}$ cross each other. The first step toward obtaining a numerical solution is to rewrite the equation to be solved in the form $f(x) = 0$. Doing so, the above equation becomes

$$f(x) = \tan x - \sqrt{(8/x)^2 - 1} = 0\,. \tag{9.13}$$

Clearly, the two equations above have the same solutions for x. Parenthetically, we mention that the problem of finding the solutions to equations of the form $f(x) = 0$ is often referred to as *finding the roots* of $f(x)$.

In Figure 9.8, we plot $f(x)$ over the domain of interest, in this case from $x = 0$ to 8. For $x > 8$, the equation has no real solutions as the argument of the square root becomes negative. The solutions, points where $f(x) = 0$, are indicated by open green circles; there are three of them. Another notable feature of the function is that it diverges to $\pm\infty$ at $x = 0, \pi/2, 3\pi/2$, and $5\pi/2$.

9.6.1.1 Brent Method

One of the workhorses for finding solutions to a single variable nonlinear equation is Brent's method, discussed in many texts on numerical methods. SciPy's implementation of the Brent algorithm is the function `scipy.optimize.brentq(f, a, b)`, which has three required arguments. The

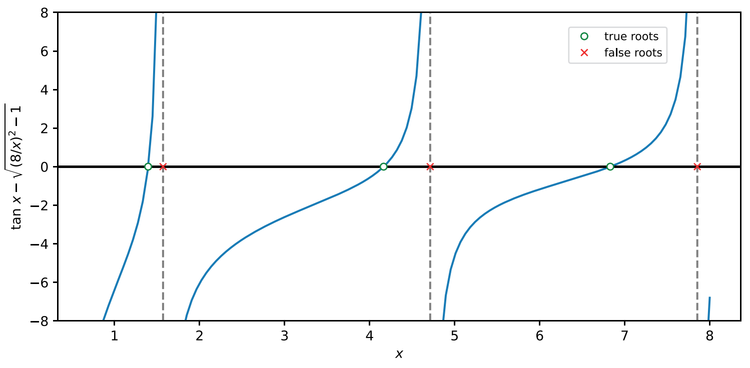


Figure 9.8 Roots of a nonlinear function.

first argument `f` is the name of the user-defined function to be solved. The next two, `a` and `b`, are the x values that bracket the solution you are looking for. You should choose `a` and `b` so that there is only one solution in the interval between `a` and `b`. Brent's method also requires that `f(a)` and `f(b)` have opposite signs; an error message is returned if they do not. To find the three solutions to $\tan x - \sqrt{(8/x)^2 - 1} = 0$, we need to run `scipy.optimize.brentq(f, a, b)` three times using three different values of `a` and `b` that bracket each of the three solutions. The program below illustrates how to use `scipy.optimize.brentq`.

Code: root_brentq.py

```python
import numpy as np
import scipy.optimize
import matplotlib.pyplot as plt

def tdl(x):
    y = 8.0 / x
    return np.tan(x) - np.sqrt(y * y - 1.0)

# Find true roots
rx1 = scipy.optimize.brentq(tdl, 0.5, 0.49 * np.pi)
rx2 = scipy.optimize.brentq(tdl, 0.51 * np.pi, 1.49 * np.pi)
rx3 = scipy.optimize.brentq(tdl, 1.51 * np.pi, 2.49 * np.pi)
rx = np.array([rx1, rx2, rx3])
ry = np.zeros(3)
# print true roots using a list comprehension
print("\nTrue roots:")
print("\n".join("f({0:0.5f}) = {1:0.2e}"
                .format(x, tdl(x)) for x in rx))

# Find false roots
rx1f = scipy.optimize.brentq(tdl, 0.49 * np.pi, 0.51 * np.pi)
rx2f = scipy.optimize.brentq(tdl, 1.49 * np.pi, 1.51 * np.pi)
rx3f = scipy.optimize.brentq(tdl, 2.49 * np.pi, 2.51 * np.pi)
```

```
rxf = np.array([rx1f, rx2f, rx3f])
# print false roots using a list comprehension
print("\nFalse roots:")
print("\n".join("f({0:0.5f}) = {1:0.2e}"
                .format(x, tdl(x)) for x in rxf))

# Plot function and various roots
x = np.linspace(0.7, 8, 128)
y = tdl(x)
# Create masked array for plotting
ymask = np.ma.masked_where(np.abs(y) > 20.0, y)

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(x, ymask)
ax.axhline(color="black")
ax.axvline(x=np.pi / 2.0, color="gray", linestyle="--", zorder=-1)
ax.axvline(x=1.5 * np.pi, color="gray", linestyle="--", zorder=-1)
ax.axvline(x=2.5 * np.pi, color="gray", linestyle="--", zorder=-1)
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$\tan\,x - \sqrt{(8/x)^2-1}$")
ax.set_ylim(-8, 8)

ax.plot(rx, ry, "og", ms=5, mfc="white", label="true roots")

ax.plot(rxf, ry, "xr", ms=5, label="false roots")
ax.legend(numpoints=1, fontsize="small", loc="upper right",
          bbox_to_anchor=(0.9, 0.97))
fig.tight_layout()
fig.savefig("./figures/root_brentq.pdf")
plt.show()
```

Running this code generates the following output:

```
 In[1]: run root_brentq.py

True roots:
f(1.39547) = -6.39e-14
f(4.16483) = -7.95e-14
f(6.83067) = -1.11e-15

False roots:
f(1.57080) = -1.61e+12
f(4.71239) = -1.56e+12
f(7.85398) = 1.17e+12
```

The Brent method finds the three true roots of the equation quickly and accurately when you provide values for the brackets a and b that are valid. However, like many numerical methods for finding roots, the Brent method can produce spurious roots as it does in the above example when a and b bracket singularities like those at $x = \pi/2$, $3\pi/2$, and $5\pi/2$. Here we evaluated the function at the purported roots found by `brentq` to verify that the values of $x$ found were indeed roots. For the true roots, the values of the function were very near zero, to within an acceptable roundoff error of less than 10^{-13}. For the false roots, very large numbers on the order of 10^{12} were obtained, indicating a possible problem. These results and the plots allow you to identify the true solutions to this nonlinear function unambiguously.

The `brentq` function has several optional keyword arguments that you may find useful. One keyword argument causes `brentq` to return the solution and the value of the function evaluated at the solution. Other arguments allow you to specify a tolerance to which the solution is found and a few other parameters of interest. Most of the time, you can leave the keyword arguments at their default values. See the `brentq` entry online on the SciPy website for more information.

9.6.1.2 *Other Methods for Solving Equations of a Single Variable*

SciPy provides several other methods for solving nonlinear equations of a single variable. It has an implementation of the Newton-Raphson method called `scipy.optimize.newton`. It's the race car of such methods; it's super fast but less stable than the Brent method. To fully realize its speed, you need to specify the function to be solved and its first derivative, which is often more trouble than it's worth. You can also specify its second derivative, which may further speed up finding the solution. If you do not specify the first or second derivatives, the method uses the secant method, which is usually slower than the Brent method.

Other methods are also available, including the Ridder and bisection methods, but the Brent method is generally superior. SciPy lets you choose your favorite.

9.6.2 Solving Systems of Nonlinear Equations

Solving systems of nonlinear equations is not for the faint of heart. These are difficult problems that lack any general-purpose solutions. Nevertheless, SciPy provides quite an assortment of numerical solvers for nonlinear systems of equations. However, because of the complexity and subtleties of this class of problems, we do not discuss their use here.

9.7 NUMERICAL INTEGRATION

When a function cannot be integrated analytically or is very difficult to integrate analytically, one generally turns to numerical integration methods. SciPy has several routines for performing numerical integration. Most of them are found in the same `scipy.integrate` library where the ODE solvers are found. We list them in Table 9.2 for reference.

TABLE 9.2 Some integrating routines from `scipy.integrate` unless otherwise noted.

Function	**Description**
Integration of functions (`scipy.integrate`)	
`quad`	single integration
`dblquad`	double integration
`tplquad`	triple integration
`nquad`	*n*-fold multiple integration
`fixed_quad`	Gaussian quadrature, order n
`quadrature`	Gaussian quadrature to tolerance
`romberg`	Romberg integration
Analytical integration of polynomial functions	
	(`numpy.polynomial.polynomial`)
`polyint`	Polynomial integration
`poly1d`	Helper function for polyint
Integration of data sampled at fixed intervals	
`trapz`	trapezoidal rule
`cumtrapz`	trapezoidal rule to cumulatively compute integral
`simps`	Simpson's rule
`romb`	Romberg integration

9.7.1 Single Integrals of Functions

The function `quad` is the workhorse of SciPy's integration functions. Numerical integration is sometimes called *quadrature*, hence the name. The function `quad` is the default choice for performing single integrals of a function $f(x)$ over a given fixed range from a to b

$$\int_a^b f(x)\,dx\,. \tag{9.14}$$

The general form of `quad` is `scipy.integrate.quad(f, a, b)`, where `f` is the *name* of the function to be integrated and `a` and `b` are the lower and upper limits, respectively. The routine uses *adaptive quadrature* methods to numerically evaluate integrals, meaning it successively refines the subintervals (makes them smaller) until a specified level of numerical precision is achieved. For the `quad` routine, this is about 10^{-8}, although it often does even better.

As an example, let's integrate a Gaussian function over the range from 0 to 1:

$$\int_0^1 e^{-x^2}\,dx \tag{9.15}$$

We first need to define the function $f(x) = e^{-x^2}$, which we do using a lambda expression, and then we call the function `quad` to perform the integration.

```
 In[1]: import numpy as np

 In[2]: f = lambda x : np.exp(-x**2)

 In[3]: from scipy.integrate import quad

 In[4]: quad(f, 0, 1)
Out[4]: (0.7468241328124271, 8.291413475940725e-15)
```

The function call `scipy.integrate.quad(f, 0, 1)` returns two numbers. The first is `0.7468...`, which is the value of the integral, and the second is `8.29...e-15`, which is an estimate of the absolute error in the value of the integral, which we see is quite small compared to `0.7468`.

For its first argument, `quad` requires a function *name*. We used a lambda expression to define the function name, `f`. Alternatively, we could have defined the function using the usual `def` construction:

```
def f(x):
return np.exp(-x**2)
```

But here, it is simpler to use a lambda expression. Even simpler, we can just put the lambda expression directly into the first argument of `quad`, as illustrated here:

```
 In[5]: quad(lambda x : np.exp(-x**2), 0, 1)
Out[5]: (0.7468241328124271, 8.291413475940725e-15)
```

That works too! Thus, we see a `lambda` expression used as an *anonymous function*, a function with no name, as promised in Section 7.3.

Interestingly, the `quad` function accepts positive and negative infinity as limits.

```
 In[6]: quad(lambda x : np.exp(-x**2), 0, inf)
Out[6]: (0.8862269254527579, 7.101318390472462e-09)

 In[7]: scipy.integrate.quad(lambda x : np.exp(-x**2), -inf, 1)
Out[7]: (1.6330510582651852, 3.669607414547701e-11)
```

The quad function handles infinite limits just fine. The absolute errors are larger but still well within acceptable bounds for practical work. Note that inf is a NumPy object and should be written as np.inf within a Python program.

The quad function can integrate standard predefined NumPy functions of a single variable, like exp, sin, and cos.

```
 In[8]: quad(np.exp, 0, 1)
Out[8]: (1.7182818284590453, 1.9076760487502457e-14)

 In[9]: quad(np.sin, -0.5, 0.5)
Out[9]: (0.0, 2.707864644566304e-15)

 In[10]: quad(np.cos, -0.5, 0.5)
Out[10]: (0.9588510772084061, 1.0645385431034061e-14)
```

Suppose we want to integrate a function such as Ae^{-cx^2} defined as a normal Python function:

```
 In[11]: def gauss(x, A, c):
  ....:      return A * np.exp(-c*x**2)
```

Of course, we will need to pass the values of A and c to gauss via quad to perform the integral numerically. This can be done using args, one of the optional keyword arguments of quad. The code below shows how to do this

```
 In[12]: A, c = 2.0, 0.5

 In[13]: intgrl1 = quad(gauss, 0.0, 5.0, args=(A, c))

 In[14]: intgrl1
Out[14]: (2.5066268375731307, 2.1728257867977207e-09)
```

Note that the order of the additional parameters in args=(A, c) must be in the same order as they appear in the function definition of gauss.

We can also do this using a lambda expression:

```
 In[15]: intgrl2 = quad(lambda x: gauss(x, A, c), 0.0, 5.0)

 In[16]: intgrl2
Out[16]: (2.5066268375731307, 2.1728257867977207e-09)
```

Either way, we get the same answer.

Let's do one last example. Let's integrate the first-order Bessel function of the first kind, usually denoted $J_1(x)$, over the interval from 0 to 5. $J_1(x)$ is available in the special functions library of SciPy as scipy.special.jn(v, x), where v is the (real) order of the Bessel function (see Section 9.1). Note that x is the *second* argument of scipy.special.jn(v, x), which means that we cannot use the args keyword function because the integration routine quad *assumes* that the independent variable is the first argument of the function to be integrated. Here, the first argument is `v`, which we wish to fix to be 1. Therefore, we use a lambda expression to fix the parameters `A` and `c`, assign 1 to the value of `v`, and define the function to be integrated. Here is now it works:

```
In[17]: import scipy.special

In[18]: quad(lambda x: scipy.special.jn(1,x), 0, 5)
Out[18]: (1.177596771314338, 1.8083362065765924e-14)
```

Because the SciPy function `scipy.special.jn(v, x)` is a function of two variables, `v` and `x`, and we want to use the second variable `x` as the independent variable, we cannot use the function name `scipy.special.jn` together with the `args` argument of `quad`. So we use a `lambda` expression, which is a function of only one variable, `x`, and set the `v` argument equal to 1.

9.7.1.1 *Integrating Polynomial Functions*

The `numpy.polynomial.polynomial` library has a function `polyint` that can be used to integrate polynomial functions. The function `polyint` analytically takes the n^{th} antiderivative of a polynomial, which can then be used to evaluate definite integrals. First, let's import the modules we will need.

```
In[19]: import numpy as np
In[20]: import numpy.polynomial.polynomial as poly
```

As an example of how the `numpy.polynomial.polynomial` functions work, let's consider the polynomial function $p(x) = 1 - 5x + 2x^2 - 4x^3$. The polynomial can be evaluated using the `polyval` function; its syntax is `poly.polyval(x, c)`, where `x` is the array of x values at which the polynomial is evaluated and `c` is a Python list or array of the polynomial coefficients from the lowest power of x to the highest. For the polynomial $p(x) = 1 - 5x + 2x^2 - 4x^3$,

```
In[21]: c = [1, -5, 2, -4]
```

Below, we evaluate the polynomial $p(x)$ at three different values of `x`.

```
In[22]: x = np.array([1.2, 5.6, 9.0])

In[23]: poly.polyval(x, c)
Out[23]: array([   -9.032,  -666.744, -2798.   ])
```

Next, we use the `polyint` function, which returns the coefficients of the antiderivative of $p(x)$.

```
In[24]: c_ad = poly.polyint(c)

In[25]: c_ad
Out[25]: array([ 0.       , 1.       , -2.5      , 0.66666667, -1.])
```

Indeed, this gives the coefficients of the antiderivative of $p(x)$

$$P(x) = \int p(x)\,dx = C + x - \frac{5}{2}x + \frac{2}{3}x^3 - x^4$$

where C is the integration constant. By default, the `polyint` function sets the integration constant C to 0, as seen in the first coefficient of `c_ad` shown above. An optional keyword argument of `polyint` can set C to a nonzero value, but we will not need to use it here.

With the coefficients of the antiderivative polynomial in hand, it is then easy to determine the definite integral of the polynomial $p(x) = 1-5x+2x^2-4x^3$ between any two limits.

$$q \equiv \int_a^b p(x)\,dx = P(b) - P(a)\,. \tag{9.16}$$

For example, if $a = 1$ and $b = 4$,

```
In[26]: q = poly.polyval(4, c_ad) - poly.polyval(1, c_ad)

In[27]: q
Out[27]: -247.5
```

or

$$\int_1^5 \left(1 - 5x + 2x^2 - 4x^3\right)\,dx = -247\tfrac{1}{2}\,. \tag{9.17}$$

9.7.2 Double Integrals

The `scipy.integrate` function `dblquad` can be used to numerically evaluate double integrals of the form

$$\int_{y=a}^{y=b} dy \int_{x=g(y)}^{x=h(y)} dx\, f(x,y)\,. \tag{9.18}$$

The general form of `dblquad` is

```
scipy.integrate.dblquad(func, a, b, gfun, hfun)
```

where `func` is the name of the function to be integrated, `a` and `b` are the lower and upper limits of the `y` variable, respectively, and `gfun` and `hfun` are the *names* of the functions that define the lower and upper limits of the `x` variable. As an example, let's perform the double integral

$$\int_0^{1/2} dy \int_0^{\sqrt{1-4y^2}} 16xy\,dx\,. \tag{9.19}$$

We define the functions f, g, and h, using lambda expressions. Note that even if g and h are constants, as they may be in many cases, they must be defined as functions, as we have done here for the lower limit.

```
 In[28]: f = lambda x, y : 16.0 * x * y

 In[29]: h = lambda y : np.sqrt(1-4*y**2)

 In[30]: from scipy.integrate import dblquad

 In[31]: dblquad(f, 0.0, 0.5, 0.0, h)
Out[31]: (0.5, 1.7092350012594845e-14)
```

Once again, there are two outputs: the first is the value of the integral, and the second is its absolute uncertainty.

Of course, the lower limit can also be a function of y, as we demonstrate here by performing the integral

$$\int_0^{1/2} dy \int_{1-2y}^{\sqrt{1-4y^2}} 16xy\, dx\,. \tag{9.20}$$

The code for this is given by

```
 In[32]: g = lambda y : 1.0 - 2.0 * y

 In[33]: dblquad(f, 0.0, 0.5, g, h)
Out[33]: (0.33333333333333326, 1.3125184411111567e-14)
```

9.7.2.1 *Other Function Integration Routines*

In addition to the routines described above, `scipy.integrate` has several other integration routines, including `nquad`, which performs n-fold multiple integration, as well as other routines that implement other integration algorithms. You will find, however, that `quad` and `dblquad` meet most of your needs for numerical integration.

9.7.3 Integrating Numerical Data

Sometimes, you have data, perhaps from a set of measurements rather than a function that you would like to integrate numerically. `scipy.integrate` has several routines for accomplishing this task, provided your data is sampled at regular intervals. Suppose, for example, that you have a set of y data points $\{y_0, y_1, \ldots, y_{N-1}\}$ with corresponding x values of $\{x_0, x_1, \ldots, x_{N-1}\}$. If the data points are evenly spaced so that $x_i - x_{i-1} = \Delta x$ is constant, then the routines listed at the bottom of Table 9.2 can be used to find the numerical integral of the data.

9.8 SOLVING ODES

Initial value problems of linear ODEs can be solved using analytical techniques. Nonlinear ODEs, however, cannot be solved using analytical techniques except in exceptional cases. When analytical techniques do not work or are inconvenient, one generally turns to numerical methods.

The `scipy.integrate` library has a powerful routine, `solve_ivp`, for numerically solving initial-value problems of ordinary differential equations (ODEs). While `solve_ivp` can solve n^{th}-order ODEs, it doesn't do so directly. Instead, it solves a system of first-order ODEs. Fortunately, an n^{th}-order ODE can generally be rewritten as a system of n coupled first-order ODEs. We show you how to do this in an example below. Once this is done, the problem takes the following form:

$$\begin{aligned}\frac{dy_1}{dt} &= f_1(t, y_1, ..., y_n)\\ \frac{dy_2}{dt} &= f_2(t, y_1, ..., y_n)\\ \vdots \;&=\; \vdots\\ \frac{dy_n}{dt} &= f_n(t, y_1, ..., y_n).\end{aligned} \tag{9.21}$$

The n equations require n initial conditions, one for each variable y_i. Once cast in this form, the routine can call any one of several different ODE solvers, some for stiff and others for non-stiff problems.

9.8.1 A First-Order ODE

Before tackling n^{th}-order ODEs, let's look at a simple first-order ODE to get familiar with the `solve_ivp` routine and some of its features. Consider the simple first-order ODE describing the change of concentration $c(t)$ as a function of time

$$\frac{dc}{dt} = -\frac{1}{\tau}c\,, \tag{9.22}$$

with the initial condition that the concentration is $c(0) = c_0$. The analytical solution to this initial value problem is

$$c(t) = c_0 e^{-t/\tau}\,. \tag{9.23}$$

We can use this solution to check the accuracy of the numerical solution we obtain using `solve_ivp`.

The function `solve_ivp` has three mandatory arguments and several optional arguments. The mandatory arguments are:

fun: a function that returns the derivative functions $f_n(t, y_1, \ldots, y_n)$

t_span: a two-element list (or array) specifying the starting and ending times of the integration

y0: a list of the initial values of the dependent variables $y_1, \ldots, y_n$.

The first items of business are importing the `solve_ivp` function and defining a function that returns the derivative functions for the problem of interest. Because we are working with a first-order ODE, there is only one derivative function, which is given by Eq. (9.22).

```
In[1]: from scipy.integrate import solve_ivp

In[2]: def c_deriv(t, c, tau):
  ...:     return -c / tau
```

The first argument of the function is the time (a float) at which the derivative(s) $dy_1/dt, \ldots, dy_n/dt$ will be evaluated. The second argument is a list of the dependent variables, generically $y_1, \ldots, y_n$, but here just the single dependent variable `c` representing the concentration. The third (and fourth, fifth, ...) argument(s) is/are any parameter(s) that are needed to evaluate the derivatives. With these inputs, the function returns a list of the derivatives for each dependent variable evaluated at the time `t`. In this case, it returns a single derivative, given by the right-hand side of Eq. (9.22).

Next, we need to specify the starting and stopping times for the integration, the initial concentration, and the value of any parameters, here `tau`, needed to evaluate the derivative function.

```
In[3]: t_start, t_stop = 0.0, 10.0
In[4]: c0 = 1.6  # initial concentration
In[5]: tau = 2.0  # time constant (a parameter of ODE)
```

The solution is obtained by calling `solve_ivp`

```
In[6]: csoln = solve_ivp(c_deriv, [t_start, t_stop], [c0],
  ...: args=[tau])
```

Note that the third argument, `[c0]`, is a list, even if it has only one element, as it does in this case. The last argument `args`, also a list, is an optional argument and is used to specify the values of any parameters needed to evaluate the derivative function. The order of the elements in the list must be the same as in the definition of the derivative function. Here, there is only one parameter, `tau`. The output of `solve_ivp` is a dictionary:

```
In[7]: csoln
Out[7]:
```

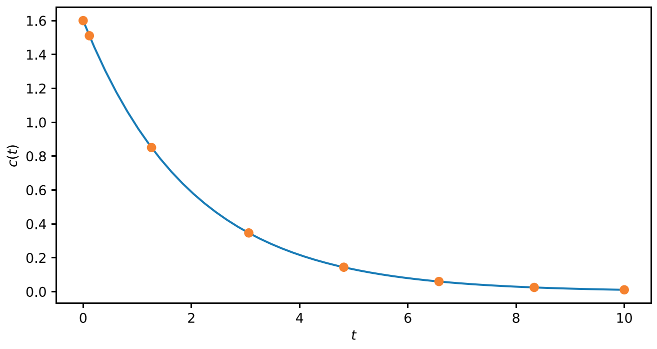


Figure 9.9 Solution to first-order ODE.

```
message: 'The solver successfully reached the end of the
integration interval.'
nfev: 44
njev: 0
nlu: 0
sol: None
status: 0
success: True
t: array([ 0.        ,  0.11488419,  1.2637261 ,  3.06082378,
           4.8165295 ,  6.57540382,  8.33550038, 10.        ])
t_events: None
y: array([[1.6       , 1.5106825 , 0.85058161, 0.34651891,
           0.14410797, 0.05983617, 0.02482991, 0.01080626]])
y_events: None
```

For the moment, we are interested only in the solution, which is given by `csoln.t` and `csoln.y`. The array of times where the solution is calculated is given by `csoln.t`. The dependent variables are given by `csoln.y[0,:]`, `csoln.y[1,:]`, ... In this case, there is only one dependent variable, the concentration, so only `csoln.y[0,:]` is specified.

By default, `solve_ivp` uses an adaptive-step-size Runge-Kutta RK4(5) method and thus returns a solution at just enough points to achieve the desired accuracy. The accuracy is determined by two optional parameters, `rtol` and `atol`, the relative and absolute tolerances, which are by default set to 10^{-3} and 10^{-6}, respectively. In this case, this produces an unevenly spaced solution of only eight points on the specified interval $[0, 10]$. Figure 9.9 shows a plot of the calculated points and the analytical solution as a continuous line.

The difference between the result obtained by `solve_ivp` and the analytical result is given by:

```
In[8]: csoln.y[0,:] - c0 * np.exp(-csoln.t / tau)
```

```
Out[8]:
array([0.00000000e+00, 1.66164860e-11, 2.08427320e-05,
       2.04511241e-04, 1.53923236e-04, 9.28685010e-05,
       5.06076616e-05, 2.55425445e-05])
```

If greater accuracy is desired, the optional parameters `rtol` and `atol` can be decreased from their default values. In doing so, the adaptive-step-size Runge-Kutta RK4(5) solver of `solve_ivp` will need a higher density of points, as demonstrated here.

```
In[9]: csoln2 = solve_ivp(c_deriv, [t_start, t_stop], [c0],
  ...: args=[tau], rtol=1e-4, atol=1e-7)
In[10]: csoln2.y[0,:] - c0 * np.exp(-csoln2.t / tau)
Out[10]:
array([0.00000000e+00, 1.03339559e-12, 1.21209786e-06,
       1.57616524e-05, 1.66117434e-05, 1.37928617e-05,
       1.03031920e-05, 7.25170757e-06, 4.91258229e-06,
       3.24053792e-06, 2.09602897e-06, 1.99706238e-06])
```

You can ask `solve_ivp` to generate a denser, evenly spaced solution data set in one of two ways. The most versatile way is to set the optional parameter `dense_output=True`.

```
In[11]: csoln2 = solve_ivp(c_deriv, [t_start, t_stop], [c0],
   ...: args=[tau], dense_output=True)
```

This produces the same solution obtained in `csoln` above, because it uses the same parameters, notably the same values of `rtol` and `atol`. However, setting the optional parameter `dense_output=True` allows you to generate a denser data set from this solution. To do so, you specify a time array over the time domain defined by `[t_start, t_stop]` at the desired (higher) temporal density.

```
In[12]: t = np.linspace(t_start, t_stop, 41)
In[13]: z = csoln2.sol(t)
```

The statement `z = csoln2.sol(t)` uses the `sol` method of `solve_ivp` to generate values of the solution in an array `z` for each time in the array `t`. It generates `z` using an interpolating polynomial between the points in the original solution, not by resolving the ODE with a denser set of times. This procedure saves time and should maintain the accuracy specified by `rtol` and `atol`.

Alternatively, you can specify a time array in the original call to `solve_ivp`.

```
In[14]: csoln3 = solve_ivp(c_deriv, [t_start, t_stop], [c0],
   ...: args=[tau], t_eval=t)
```

The time array is available as `csoln3.t` and the solution as `csoln3.y`. The solutions are identical to those obtained using `dense_output=True`. However, you lose the ability to generate another solution using a time array with a different density of times. To obtain a different density of times, you need to rerun `solve_ivp` and specify a different time array.

9.8.2 A Second-Order ODE

Here, we show how to use `solve_ivp` to solve the equation for a driven damped pendulum. The equation of motion for the angle θ the pendulum makes with the equilibrium position is given by

$$\frac{d^2\theta}{dt^2} = -\frac{1}{Q}\frac{d\theta}{dt} - \sin\theta + d\cos\Omega t \tag{9.24}$$

where t is time, Q is the quality factor that defines the damping, d is the forcing amplitude, and Ω is the frequency of the forcing. Reduced variables are used here so that the natural (angular) oscillation frequency is 1. The ODE is nonlinear owing to the $\sin\theta$ term.

Equation (9.24) is a second-order ODE. The first step is to transform it into two coupled first-order ODEs. The transformation is readily accomplished by defining a new variable $\omega \equiv d\theta/dt$. With this definition, there are two dependent variables, $y_1 = \theta$ and $y_2 = \omega$, and we can rewrite this second-order ODE as two coupled first-order ODEs:

$$\frac{d\theta}{dt} = \omega \tag{9.25}$$
$$\frac{d\omega}{dt} = -\frac{1}{Q}\omega - \sin\theta + d\cos\Omega t. \tag{9.26}$$

In this case, the functions on the right-hand side of Eq. (9.21) of the equations are

$$f_1(t,\theta,\omega) = \omega \tag{9.27}$$
$$f_2(t,\theta,\omega) = -\frac{1}{Q}\omega - \sin\theta + d\cos\Omega t. \tag{9.28}$$

Note that there are no explicit derivatives on the right-hand side of the functions f_i; they are all functions of t and the various y_i, in this case, θ and ω. Specifying the initial conditions of the dependent variables θ and ω at $t = 0$ completes the statement of the problem.

Having written the n^{th}-order ODE as a system of n first-order ODEs, we can now use `solve_ivp` to solve the problem numerically.

The first task is to write the function `fun`, which in this example we call `f`.

This is done in lines 7–11 of the program `odePend.py` given below. As in the previous example, the first argument of the function `f` is the current time `t`. The second argument is the list (or array) of current `y` values. The third and following arguments pass the parameters needed to evaluate `f`. In this case, there are three parameters: Q, d, and Ω, or `Q`, `d`, and `Omega`.

The function `f` returns the values of the derivatives $dy_i/dt = f_i(t, y_1, ..., y_n)$ as a list. Here, there are two dependent variables, θ and ω, and two derivatives, $d\theta/dt$ and $d\omega/dt$. Lines 9–10 calculate the derivatives that are returned by `f` as a tuple in the *same order* as they appear in `y`.

In the main program, the parameters `Q`, `d`, and `Omega` are defined in lines 15–17. The initial conditions are defined in lines 20–21, and the start and stop times are defined in line 24.

The only remaining task before running `solve_ivp` is to decide which solver to use. In the previous example, no solver was specified in the `solve_ivp` call so the default solver Runge-Kutta RK4(5) was used. The RK4(5) is an *explicit* solver and an excellent choice for most problems. Here, an *implicit* solver gives better results, so the Radau solver is used. The optional argument `method` is used to specify the solver. The `solve_ivp` routine also offers several other solvers, which you can learn about by reading the SciPy documentation.

The `solve_ivp` routine is called in line 28 and assigned to the variable `psoln`, which stores the output. Notice that for the keyword argument `args`, the parameters are specified in the same order as they appear in the derivative function `f`, as they must!

The arguments `rtol` and `atol` specify the relative and absolute tolerances, and thus the accuracy of the numerical solution. Setting them appropriately can be delicate. `rtol` should be larger than `atol`, typically by a factor of 10 to 10^3. For the nonlinear ODE considered here, solutions can be very sensitive to the initial conditions so high accuracy is required for reproducable results: setting `rtol` to 10^{-8} and `atol` to 10^{-10} seems to work well for most initial conditions. Smaller values can be used at the expense of greater computational time.

The argument `dense_output` is set to `True` so that the solution can be plotted with an arbitrary density of points. The array of times where the solution is calculated by `solve_ivp` is given by `psoln.t`. The two dependent variables, θ and ω are given at these times by `psoln.y[0,:]` and `psoln.y[1,:]`, respectively.

In this case, it turns out that the average spacing between points returned in `psoln.t` by `solve_ivp` is about 0.25, but for purposes of plotting, it's more visually appealing to have a temporal spacing of about 0.1. Setting the optional argument `dense_output` set to `True`, we obtain a denser data set using `sol` method of `solve_ivp`. The denser array of times is created in lines 33–34, and the denser set of solutions is generated in line 35 using the `sol` method. The two dependent variables, $\theta$ and ω for the denser set of times are given by `psoln.z[0,:]` and `psoln.z[1,:]`, respectively. These data are used in the plot generated by `odePend.py` and shown in Figure 9.10

Code: ode_pend.py

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
import seaborn as sns

def f(t, y, Q, d, Omega):
    theta, omega = y       # unpack current values of dep variables
    d_theta_dt = omega     # calculate derivatives
    d_omega_dt = -omega / Q - np.sin(theta) + d * np.cos(Omega * t)
    return d_theta_dt, d_omega_dt

# Parameters
Q = 2.0            # quality factor (inverse damping)
d = 1.091          # forcing amplitude
Omega = 0.67       # drive frequency

# Initial values
theta0 = 0.0       # initial angular displacement
omega0 = 0.0       # initial angular velocity

# Make time array for solution
t_start, t_stop = 0.0, 300.0

# Call the ODE solver
mthd = "Radau"
psoln = solve_ivp(f, [t_start, t_stop], [theta0, omega0],
                  args=(Q, d, Omega), method=mthd,
                  rtol=1e-8, atol=1e-10, dense_output=True)

# Calculate dense solution for plotting
t_increment = 0.1
t = np.arange(t_start, t_stop, t_increment)
z = psoln.sol(t)

# Plot results
fig = plt.figure(figsize=(9.5, 4.5))
c = sns.color_palette("icefire_r", 3)  # a Seaborn palette

# Plot theta as a function of time
ax1 = fig.add_subplot(221)
ax1.plot(t, z[0, :], color=c[0])
ax1.set_xlabel("time", fontsize=14)
ax1.set_ylabel(r"$\theta$", fontsize=14)
ax1.text(0.98, 0.98, "{0:s}".format(mthd), ha="right", va="top",
         transform=ax1.transAxes)

# Plot omega as a function of time
ax2 = fig.add_subplot(223)
ax2.plot(t, z[1, :], color=c[1])
```

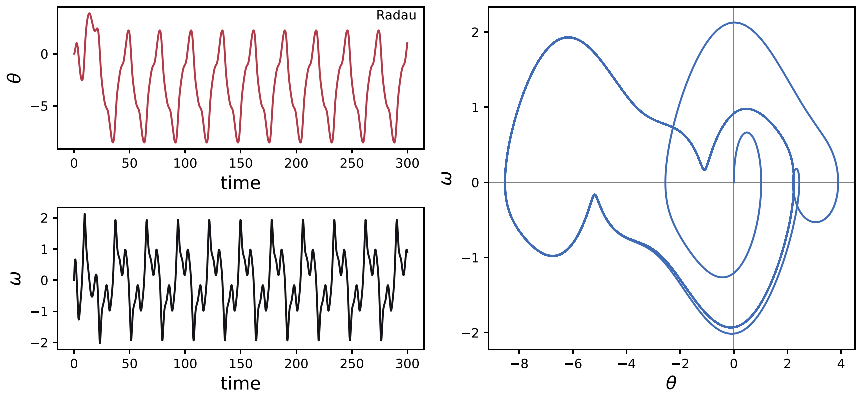


Figure 9.10 Pendulum trajectory.

```
ax2.set_xlabel("time", fontsize=14)
ax2.set_ylabel(r"$\omega$", fontsize=14)

# Plot omega vs theta
ax3 = fig.add_subplot(122)
pi, twopi = np.pi, 2.0 * np.pi
ax3.plot((z[0, :]), z[1, :], "-", ms=1, color=c[2])
ax3.set_xlabel(r"$\theta$", fontsize=14)
ax3.set_ylabel(r"$\omega$", fontsize=14)
ax3.axhline(lw=0.5, color="gray", zorder=-1)
ax3.axvline(lw=0.5, color="gray", zorder=-1)
fig.tight_layout()
fig.savefig("./figures/ode_pend.pdf")
plt.show()
```

The plots in Figure 9.10 reveal that for the particular set of input parameters chosen, `Q = 2.0`, `d = 1.091`, and `Omega = 0.67`, the pendulum trajectories are chaotic. Weaker forcing (smaller d) leads to the more familiar behavior of sinusoidal oscillations with a fixed frequency, which, at long times, is equal to the driving frequency.

In this example, the Jacobian matrix, defined as $\partial f_i / \partial y_j$, is determined numerically by `solve_ivp`. Alternatively, the Jacobian can be specified by an auxiliary function and implemented by providing the additional keyword argument `jac=jacobian` in the `solve_ivp` call. Here, the Jacobian function is

```
def jacobian(t, y, Q, f0, Omega):  # Calculates Jacobian
    phi, vphi = y
    j11, j12 = 0.0, 1.0
    j21, j22 = -np.cos(phi), -1.0 / Q
    return np.array([[j11, j12], [j21, j22]])
```

9.9 DISCRETE (FAST) FOURIER TRANSFORMS

The SciPy library has several routines for performing discrete Fourier transforms. Before delving into them, we briefly review Fourier transforms and discrete Fourier transforms.

9.9.1 Continuous and Discrete Fourier Transforms

The Fourier transform of a function $g(t)$ is given by

$$G(f) = \int_{-\infty}^{\infty} g(t) e^{-i2\pi ft} dt \,, \tag{9.29}$$

where f is the Fourier transform variable; if t is time, then f is frequency. The inverse transform is given by

$$g(t) = \int_{-\infty}^{\infty} G(f) e^{i2\pi ft} dt \,. \tag{9.30}$$

Here we define the Fourier transform in terms of the frequency f rather than the angular frequency $\omega = 2\pi f$.

The conventional Fourier transform is defined for continuous functions and thus has an infinite number of data points. When doing numerical analysis, however, you work with *discrete* data sets, that is, data sets defined for a finite number of points. The discrete Fourier transform (DFT) is defined for a function g_n consisting of a set of N discrete data points. Those N data points must be defined at *equally spaced* times $t_n = n\Delta t$ where Δt is the time between successive data points and n runs from 0 to $N - 1$. The discrete Fourier transform (DFT) of g_n is defined as

$$G_l = \sum_{n=0}^{N-1} g_n e^{-i(2\pi/N)ln} \tag{9.31}$$

where l runs from 0 to $N - 1$. The inverse discrete Fourier transform (iDFT) is defined as

$$g_n = \frac{1}{N} \sum_{l=0}^{N-1} G_l e^{i(2\pi/N)ln} . \tag{9.32}$$

The DFT is usually implemented on computers using the well-known fast Fourier transform (FFT) algorithm, generally credited to Cooley and Tukey who developed it at AT&T Bell Laboratories during the 1960s. However, their algorithm is one of many independent rediscoveries of the basic algorithm dating back to Gauss, who described it as early as 1805.

9.9.2 The SciPy FFT Library

The SciPy library `scipy.fftpack` has routines that implement a souped-up version of the FFT algorithm along with many ancillary routines that support working with DFTs. The basic FFT routine in `scipy.fftpack` is appropriately named `fft`. The program below illustrates its use, along with the plots that follow.

Code: fft_example.py

```python
import numpy as np
from scipy import fftpack
import matplotlib.pyplot as plt

width = 2.0
freq = 0.5

t = np.linspace(-10, 10, 128)
g = np.exp(-np.abs(t) / width) * np.sin(2.0 * np.pi * freq * t)
dt = t[1] - t[0]   # increment between times in time array

G = fftpack.fft(g)   # FFT of g
f = fftpack.fftfreq(g.size, d=dt)   # FFT frequenies
f = fftpack.fftshift(f)   # shift freqs from min to max
G = fftpack.fftshift(G)   # shift G order to match f

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(9, 6))
ax1.plot(t, g)
ax1.set_xlabel(r"$t$")
ax1.set_ylabel(r"$g(t)$")
ax1.set_ylim(-1, 1)
ax2.plot(f, np.real(G), color="C0", label="real part")
ax2.plot(f, np.imag(G), color="C1", label="imaginary part")
ax2.legend()
ax2.set_xlabel(r"$f$")
ax2.set_ylabel(r"$G(f)$")
for ax in (ax1, ax2):
    ax.axhline(color="gray", lw=0.5, zorder=-1)
    ax.axvline(color="gray", lw=0.5, zorder=-1)
plt.tight_layout()
plt.savefig("figures/fft_example.pdf")
```

The DFT has real and imaginary parts, which are plotted in Figure 9.11.

The `fft` function returns the N Fourier components of G_n starting with the zero-frequency component G_0 and progressing to the maximum positive frequency component $G_{(N/2)-1}$ (or $G_{(N-1)/2}$ if N is odd). From there, `fft` returns the maximum *negative* component $G_{N/2}$ (or $G_{(N-1)/2}$ if N is odd) and continues upward in frequency until it reaches the minimum negative frequency component G_{N-1}. This is the standard way that most numerical DFT packages order DFTs. The `scipy.fftpack` function `fftfreq` creates the array of frequencies in this non-intuitive order such that `f[n]` in the above routine

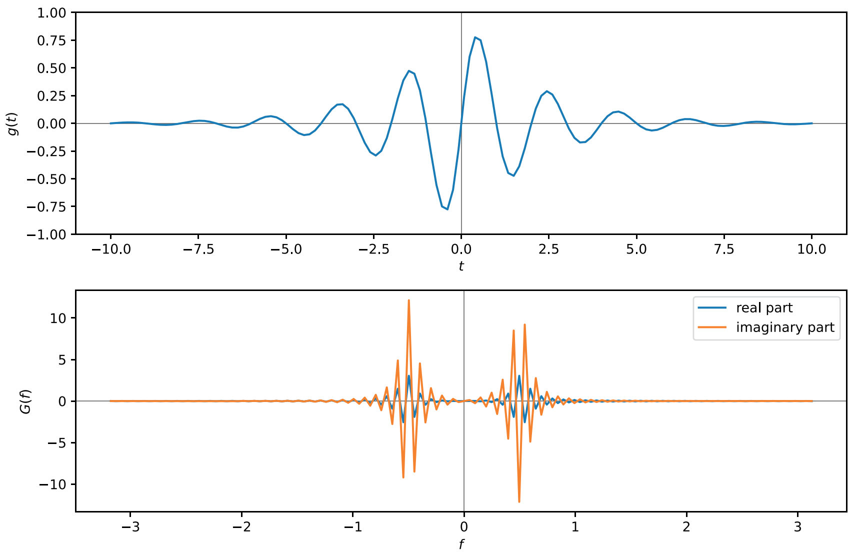


Figure 9.11 Function $g(t)$ and its DFT $G(f)$.

is the correct frequency for the Fourier component `G[n]`. The arguments of `fftfreq` are the size of the original array g and the keyword argument d that is the spacing between the (equally spaced) elements of the time array (`d=1` if left unspecified). The package `scipy.fftpack` provides the convenience function `fftshift` that reorders the frequency array so that the zero-frequency occurs at the middle of the array, that is, so the frequencies proceed monotonically from smallest (most negative) to largest (most positive). Applying `fftshift` to both `f` and `G` puts the frequencies `f` in ascending order and shifts `G` so that the frequency of `G[n]` is given by the shifted `f[n]`.

The `scipy.fftpack` module also contains routines for performing 2-dimensional and n-dimensional DFTs, named `fft2` and `fftn`, respectively, using the FFT algorithm.

As for most FFT routines, the `scipy.fftpack` FFT routines are most efficient if N is a power of 2. Nevertheless, the FFT routines can handle data sets where N is not a power of 2.

`scipy.fftpack` also supplies an inverse DFT function `ifft`. It is written to act on the *unshifted* FFT so take care! Note also that `ifft` returns a *complex* array. Because of machine roundoff error, the imaginary part of the function returned by `ifft` will generally be very near zero but not exactly zero, even when the original function is a purely real function.

9.10 EXERCISES

1. Plot the following functions using the following NumPy methods: `polynomial.Chebyshev.basis` and `polynomial.Hermite.basis`.

   (a) The first four Chebyshev polynomials of the first kind over the interval from -1 to $+1$. Consult the documentation about `numpy.polynomial.Chebyshev` on the NumPy web site.

   (b) The first four four wave functions $\psi_n(x)$ of the quantum mechanical simple harmonic oscillator are given by

$$\frac{e^{-x^2/2}}{\left(2^n n!\sqrt{\pi}\right)^{1/2}} H_n(x) \tag{9.33}$$

   where $H_n(x)$ is the n^{th} (physicist's) Hermite polynomial and $n = 0, 1, 2, 3, \ldots$. Plot these on the interval from -5 to $+5$. Consult the documentation about `numpy.polynomial.Hermite` on the NumPy website.

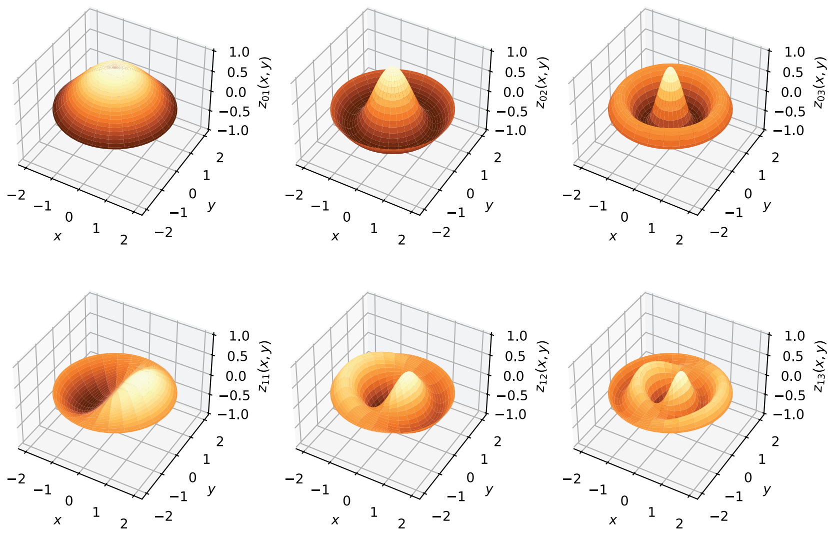


Figure 9.12 Low-order modes of a vibrating drumhead using the Seaborn color palette `YlOrBr_r`.

2. The possible shapes (eigenmodes) of a vibrating drumhead (*i.e.*, a circular vibrating membrane whose perimeter is fixed so that it does not

move) are given by

$$z_{nm}(r) = J_n(\alpha_{nm} r/R)\ [a\cos(n\theta) + b\sin(n\theta)]$$

where $z_{nm}(r,\theta)$ is the vertical height of the membrane with radial coordinates (r,θ), a and b are constants determined by the initial conditions, R is the radius of the drumhead, J_n is the n^{th} order Bessel function, and α_{nm} is the m^{th} positive root of J_n. For each value of $m = 1, 2, 3, \ldots$, the index n can take on the values $n = 0, 1, \ldots, m$. Each unique pair of integers (n,m) corresponds to a single vibrational mode of the drumhead.

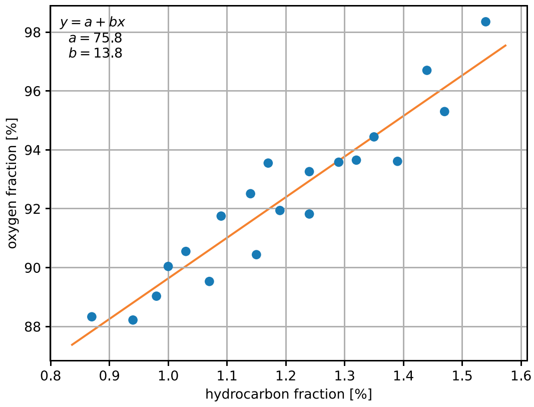
Make a 3D surface plot of each of the low-order vibrational modes: $(n,m) = (0,1), (0,2), (0,3), (1,1), (1,2), (1,3)$. For the plots, set $R = 2$, $a = 1$, and $b = 0$. The resulting plot should look something like that shown in Figure 9.12.

3. The data below show the purity of oxygen produced in a chemical distillation process against the percentage of hydrocarbons present in the main condenser of the distillation unit.

Data: distillation.txt

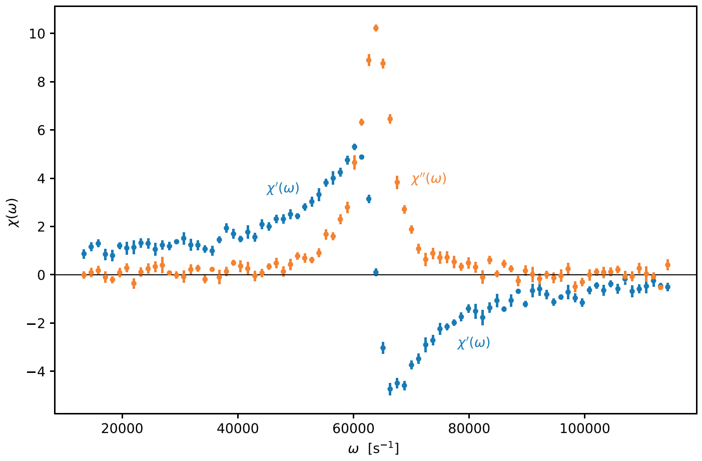
```
hydrocarbon    oxygen
 level (%)    level (%)
   1.44         96.70
   1.35         94.44
   0.87         88.33
   1.24         91.82
   1.19         91.94
   1.54         98.35
   1.39         93.61
   1.29         93.58
   1.17         93.55
   1.00         90.04
   1.14         92.51
   1.03         90.55
   1.07         89.53
   1.09         91.75
   1.15         90.44
   1.24         93.26
   1.32         93.65
   0.98         89.03
   1.47         95.30
   0.94         88.22
```

Plot the oxygen fraction *vs.* the hydrocarbon fraction and fit the data to a straight line using no weighting. Use the `linfit` routine from the file `fit_linear_demo.py` to fit the data and then draw a straight line representing the fit through the data. Report the results of the fit on the plot



as shown below In addition, fit the data using `scipy.stats.linregress` routine and show that you get the same results.

4. The text file `res.txt` contains data obtained from a small mechanical resonator. Data for both the in-phase $\chi'(\omega)$ and out-of-phase $\chi''(\omega)$ response of the resonator are listed, along with their uncertainties, as a function of the angular frequency ω, and are plotted below.



These data can be modeled by the equations:

$$\chi'(\omega) = A_1 \frac{\omega_0^2 - \omega^2}{(\omega_0^2 - \omega^2)^2 + (\gamma\,\omega)^2}\,, \quad \chi''(\omega) = A_2 \frac{\gamma\,\omega}{(\omega_0^2 - \omega^2)^2 + (\gamma\,\omega)^2}$$

Fit each of the two data sets in `res.txt` using the SciPy routine `scipy.optimize.curve_fit` using the appropriate fitting function above. Note that in each case, there are three fitting parameters, A_1 or A_2, ω_0, and γ (determining good starting values for A_1 and A_2 will require some thought!). You will need to provide estimates of each value, which you should determine by examining the above plots and equations as inputs to the fitting routine. Use the uncertainties for the data in `res.txt` to determine the weighting. Make a plot like the one above, together with continuous lines representing the fits to the data. Report the values of the two fits on the plot like Figure 9.6.

5. Numerically solve the following system of equations using the `solve()` routine from SciPy's `linalg` module. That is, find the numerical values of x_1, x_2, x_3, x_4 that simultaneously satisfy these four equations:

$$\begin{aligned} x_1 - 2x_2 + 9x_3 + 13x_4 &= 1 \\ -5x_1 + x_2 + 6x_3 - 7x_4 &= -3 \\ 4x_1 + 8x_2 - 4x_3 - 2x_4 &= -2 \\ 8x_1 + 5x_2 - 7x_3 + x_4 &= 5 \end{aligned}$$

Verify that you get the same results by matrix inversion using SciPy's `linalg` module.

6. Numerically integrate the following integrals and compare the results to the exact value of each one.

$$\text{(a)} \quad \int_{-1}^{1} \frac{dx}{1+x^2} = \frac{\pi}{2} \qquad \text{(b)} \quad \int_{-\infty}^{\infty} \frac{dx}{(e^x + x + 1)^2 + \pi^2} = \frac{2}{3}$$

You may encounter difficulties evaluating the integral in part (b). If you do, try evaluating the integral in two parts, from $-\infty$ to 0 and from 0 to ∞, to see if you can locate the source of the problem. Once you locate the source of the problem, find a practical strategy to get an answer and demonstrate that it is accurate.

7. Numerically integrate the following double integrals using the `scipy.integrate.dblquad` routine.

$$\text{(a)} \int_0^3 dy \int_{y^2}^{9} dx\, x^3 e^{-y^3} \approx 1400 \quad \text{(b)} \int_0^2 dy \int_{\sqrt{y}}^{9y} dx\, \sqrt{x^4+2} \approx 970\,.$$

Specify answers to at least 6 digits (the first two are provided above).

8. Fit the following data set with different kinds of splines:

   (a) Fit with a linear interpolating spline.

   (b) Fit with a cubic spline.

   (c) Fit with a cubic smoothing spline using the `UnivariateSpline` routine from the `scipy.interpolate module`. Choose a value of the smoothing parameter `s` that results in 6 knots (this will require some trial and error). Your code should print out the value of the smoothing parameter `s` you end up using and the number of knots.

   Plot the data and fits on a single graph. Include a legend to indicate which curves correspond to which kind of spline fit.

Data: sdata.txt

```
xdata      ydata
 0.1         1.1
 0.8         3.4
 1.5         5.9
 2.2        15.0
 2.9        24.6
 3.6        23.3
 4.3        21.7
 5.0        18.7
 5.6        13.2
 6.3         7.4
 7.0         3.6
 7.7         0.5
 8.4         1.5
 9.1         0.7
 9.8         0.1
```

9. Use `scipy.integrate.solve_ivp` to solve the following set of nonlinear ODEs:

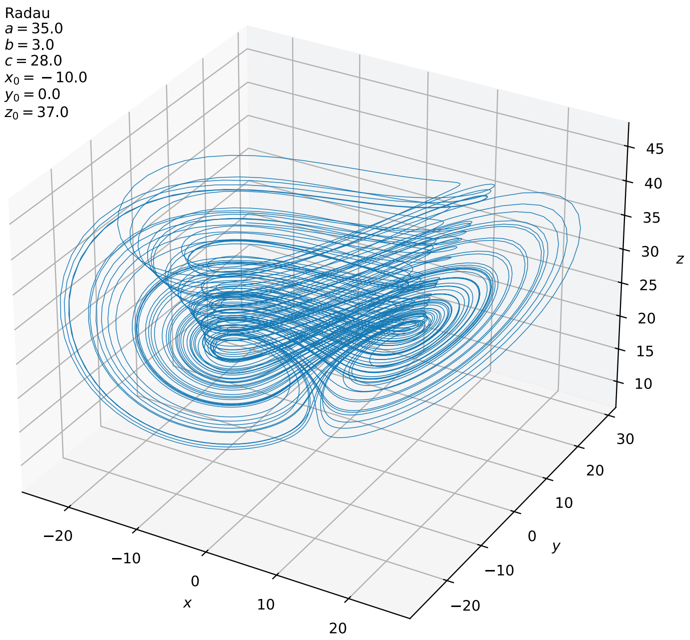
$$\frac{dx}{dt} = a(y-x)\,, \qquad \frac{dy}{dt} = (c-a)x - xz + cy\,, \qquad \frac{dz}{dt} = xy - bz$$

   Use $x_0 = -10$, $y_0 = 0$, $z_0 = 37$ for the initial conditions.

   (a) Set the initial parameters to $a = 35$, $b = 3$, $c = 28$ and plot the 3D trajectory. Use the Radau implicit integration method in `solve_ivp`. Integrate from time $t = 0$ out to $t = 50$. You should get a solution that looks like the trajectory shown below, which is an *attractor*. Take care to choose a small enough time step (but not too small!).

(b) Set $b = 10$ while keeping $a = 40$ and $c = 35$ and plot the result. You should see the trajectory approach a periodic solution after a short time.

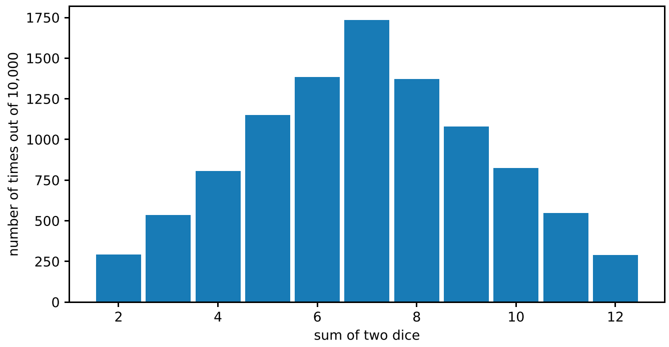
HINTS: Use `ax.plot3D(x, y, z, lw=0.5)` to plot the 3D trajectory shown above. Use `ax.text2D()` to write the text on the plot.



10. Use NumPy's random number generator to simulate the rolling of a pair of dice 10,000 times and then make a histogram of the number of times each of the results from 2 to 12 are obtained. The result should look something like the figure below.

11. In this exercise, you explore using discrete Fourier transforms to filter noisy signals. As a first step, use the following function to create a noisy Gaussian waveform:

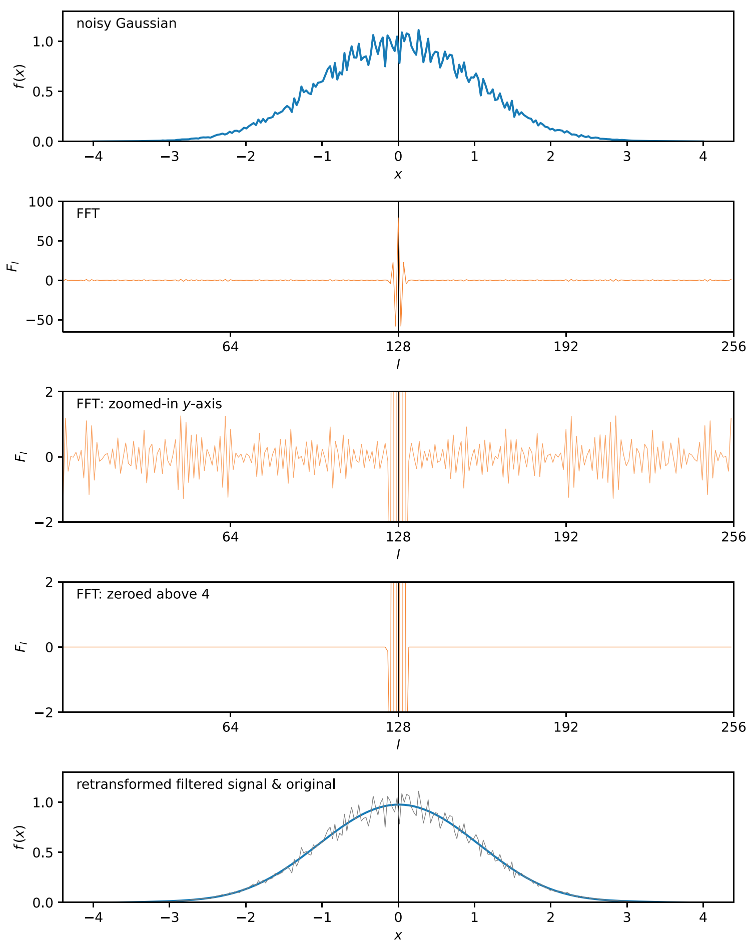
```
def gaussNoisy(x, noiseAmp):
noise = noiseAmp * (np.random.randn(len(x)))
return np.exp(-0.5 * x * x) * (1.0 + noise)
N = 256
x = np.linspace(-4.0, 4.0, N)
y = gaussNoisy(x, 0.1)
```



(a) Calculate the discrete Fourier transform using NumPy's `fft` and `fftshift` routines so that you have a properly ordered Fourier transform.

(b) Plot the noisy Gaussian and its DFT on two separate panes in the same figure window. Set the limits of the *y*-axis of the DFT plot so that you can see the noise at the high frequencies (limits of ± 2 should suffice).

(c) Next, set all the frequencies higher than a certain cutoff equal to zero (real and imaginary parts) to obtain a filtered DFT (take care to do this right!). Inverse Fourier transform the filtered DFT. Then, add a third frame to your figure and plot the inverse transform of the filtered DFT and the original. If it works well, you should observe a smooth Gaussian that goes through the original noisy one. Experiment with different cutoffs, say all the frequencies above 2, 4, 8, 16, 32. The figure below shows an example of what the third frame of your plot might look like.


noisy Gaussian
$f(x)$
x
FFT
F_l
l
FFT: zoomed-in y-axis
FFT: zeroed above 4
retransformed filtered signal & original

CHAPTER 10

Python Classes: Encapsulation

In this chapter, you learn how to create and use ***Python classes****, which are central to what is known as* ***object-oriented programming*** *(OOP). You learn about* ***encapsulation****, one of the most useful features of classes, especially for scientific and engineering work. You will learn how to design a Python class with methods (functions) that act on data read by the class.*

This chapter provides an introduction to object-oriented programming—OOP. As its name implies, OOP involves programming with objects, a concept introduced in Section 4.5. While using Python objects is relatively straightforward, object-oriented programming is a big subject, and covering it thoroughly requires much more space than we can give it here. On the other hand, the basic machinery of OOP is relatively simple, especially in Python. This chapter introduces this machinery and provides a few examples of how it might be used, particularly for working with scientific data. We start by reviewing the essential properties of a Python object.

An object, we learned (see Section 4.5), is a collection of data, together with *attributes* that characterize the data, and functions, called *methods*, that can operate on the data or attributes.

This idea of bundling data with its attributes and methods in a single package—as an object—is called *encapsulation*. Encapsulation allows you to provide the functionality you need to process data while isolating the code that

DOI: 10.1201/9781032673950-10

provides that functionality from other parts of your code. This helps avoid unintended clashes between different parts of a software package. It can also help you organize your code conceptually. These features become particularly useful when many programmers are developing large software packages. But they can also provide useful functionality even for relatively small programs.

A NumPy array provides an illustrative example of an object. It contains data in the form of the elements of the array, and it has several attributes, such as the array size and shape, which can be accessed using the dot syntax. The size and shape of a NumPy array are determined for a particular array when it is created or, in the jargon of OOP, *instantiated*:

```
 In[1]: w = np.array([[2, -5, 6], [-10, 9, 7]])

 In[2]: w.size     # size is an array instance variable
Out[2]: 6

 In[3]: w.shape    # shape is another array instance variable
Out[3]: (2, 3)
```

NumPy arrays also have methods associated with them, functions that act on a NumPy array, such as the methods that calculate the mean and standard deviation of the array:

```
 In[4]: w.mean()   # mean() is an array method
Out[4]: 1.5

 In[5]: w.std()    # std() is another array method
Out[5]: 6.8495741960115053
```

Object methods always have parentheses, which may or may not take an argument. By contrast, instance variables do not have parentheses or arguments.

In the language of OOP, we created an *instance* of the NumPy array *class* and named it `w` when we wrote `w = np.array(...)` above. Writing `x = np.numpy([8, -4, -6, 3])` creates another instance of the NumPy array class, with different attributes, but with the same set of methods (although using them on `x` would give different results than using them on `w`). `w` and `x` are two *objects* that belong to the same NumPy array class. Once we have instantiated an array, it is available for further queries or processing, which might involve interacting with other objects.

In Python, we can define new kinds of objects by writing classes to augment Python's classes, much like we can define our own functions to augment Python's functions. This chapter guides you through designing and coding several different Python classes.

10.1 A VERY SIMPLE CLASS

As a first step in learning how to program a Python class, let's go through the process of making a very simple (but not very useful) class. The name of the class is `Point`. In Python, class names usually start with a capital letter, a practice we follow here. As input, `Point` takes the (x, y) coordinates of a point in the x-y plane. It has three methods: one named `radius()` that returns the distance of the (x, y) point from the origin, a second named `rect_area()` that returns the area of the rectangle centered about the origin with one of its vertices at (x, y), and a third named `circ_area()` that returns the area of a circle centered about the origin that goes through the point (x, y).

Before looking at the code that defines the class is `Point`, let's first see how it works. The definition of the `Point` class is stored in a file called `pmod.py`. From the IPython console we first navigate to the directory in which the `pmod.py` file is located. Recall that you can determine the current working directory by typing `pwd` at the IPython command prompt. Once we have done this, we type

```
In [1]: %run pmod.py
```

This loads `pmod` into Python and makes any classes or functions defined within `pmod` available from the IPython console.

Next, we create an instance of the class for the (x, y) coordinates of $(6, 8)$ and then use the three methods of Point to calculate the radius (distance to the origin), the area of the rectangle, and the area of the circle.

```
In [2]: p1 = Point(6, 8)

In [3]: p1.radius()
Out[3]: 10.0

In [4]: p1.rect_area()
Out[4]: 192.0

In [5]: p1.circ_area()
Out[5]: 314.1592653589793
```

We can create a second instance of the class for different (x, y) coordinates and repeat the same calculations

```
In [6]: p2 = Point(3, 4)

In [7]: p2.radius()
Out[7]: 5.0

In [8]: p2.rect_area()
Out[8]: 48.0
```

```
In [9]: p2.circ_area()
Out[9]: 78.53981633974483
```

Here is the code for the `Point` class. We examine it below to see how it works.

Code: pmod.py

```
1   from math import pi
2
3
4   class Point:
5       """A simple class about (x, y) data points"""
6
7       def __init__(self, x, y):
8           """Input an (x, y) data point"""
9           self.x = x
10          self.y = y
11
12      def radius(self):
13          """returns distance from (x, y) to origin"""
14          return (self.x**2 + self.y**2) ** 0.5
15
16      def rect_area(self):
17          """returns area of rectangle with a vertex at (x, y) centered
18          around origin"""
19          return 4.0 * abs(self.x * self.y)
20
21      def circ_area(self):
22          """returns area of circle through (x, y) centered around
23          origin"""
24          return pi * self.radius() ** 2
```

Before defining the Point class, we import `pi` from the `math` module because it will be needed to determine the area of a circle.

The class definition starts on line 4 with the keyword `class` followed by the name of the class, `Point`, and a colon. Parentheses are not needed after the class name `Point` nor are they recommended for this simple class, although the class will still work if you include them.[1]

The `Point` class has four methods (functions): a method called `__init__()`, and the three class methods whose functionalities were introduced above.

Let's look at each of these methods starting with `__init__()`. The `__init__()` method starts and ends with two underscores and is sometimes called the *constructor*. It is part of every class and is called when the class is instantiated. The arguments of the `__init__()` method are `self` plus the two arguments specified when the class is instantiated. So when we wrote

[1]When looking at Python code on the internet, you may encounter class definitions not only with parenthesis but also with the argument `object`. This is a leftover from Python 2. Including it in Python 3 code does no harm, but it's completely superfluous unless you want your code to run under Python 2 (which you don't).

```
p1 = Point(6, 8)
```

the (x, y) values of $(6, 8)$ are the `x` and `y` arguments of `__init__()`.

But what about the `self` argument? What does it do, and why is it there? The `self` argument represents a particular instance of the class. In the examples above, the two `Point` objects, `p1` and `p2` each have their own `x` and `y` attributes. The `self` argument is how Python keeps track of the different instances of the same class.

Within the class definition, the `self` argument serves another important purpose. It makes all variables defined with a `self` prefix available to any method within that class that has `self` as an argument. Thus, each method defined in the class has `self` as their first argument. Note that the first thing we do in the `__init__()` method is to define the `self.x = x` and the `self.y = y` variables. This makes their values available anywhere within the class. Once this is done, `self.x` and `self.y` become *instance attributes* or *instance variables*. As instance attributes, they can be accessed outside the class using the dot syntax.

```
 In[10]: p1.x, p1.y
Out[10]: (6, 8)
 In[11]: p2.x, p2.y
Out[11]: (3, 4)
```

With this in mind, let's look at the definition of the `radius()` method. It calculates the radius, or distance to the origin, in the usual way using the instance variables `self.x` and `self.y`. The only argument `radius` needs is the `self` argument so that it knows which instance of the class to act on. The same holds for the `rect_area()` method.

The last method, `circ_area()`, needs the radius to determine the area of the circle centered about the origin that goes through the points (x, y). Note that when calling the class's `radius()` method, it uses the `self` prefix. This is a general rule: the `self` prefix must be used when calling a class method from within the class.

The docstrings provide a brief description of different parts of the class and provide help to the user. For example, after importing `pmod`, a user can get help on the `radius` method by typing

```
 In[12]: help(Point.radius)
Help on function radius in module __main__:

radius(self)
returns distance from (x, y) to origin
```

In the example above, the code for the class `Point` was loaded into the IPython console by running the file `pmod.py`. There is another, perhaps more familiar, way of loading the code into the IPython console. From the console, you can type

```
In[13]: import pmod
```

This treats the file `pmod.py` as a Python *module*. Once this is done, you can access the class definition in file `pmod.py` using the familiar dot syntax:

```
In[14]: p1 = pmod.Point(6, 8)
```

Alternatively, you could have loaded the code by typing

```
In[15]: import pmod as pm
```

Then you would access the class definition in file `pmod.py` using the familiar dot syntax with the prefix `pm`:

```
In[16]: p1 = pm.Point(6, 8)
```

After instantiating `p1` in either of these ways, you proceed as before.

Note that for any of these `import` statements above to work, the file `pmod.py` must be in Python's *path*, which is just the set of directories where Python looks for files. By default, Python always looks in the current directory first. Thus, for simplicity, we have put `pmod.py` in the same directory from which we run my IPython console. In Section 10.2, we discuss Python paths and how to store modules so they are accessible from any directory on your computer.

One final note on annotating a module: Sometimes, it is desirable to include some code at the end of a module to illustrate how it works. If you want to do this, you can include the following statement at the end of the module:

```
if __name__ == "__main__":
```

Then, you include the code you would like to run. What does the statement `if __name__ =="__main__":` mean? This statement allows you to run code within the `if` block when the file runs as a script but not when it is imported as a module. For example, suppose we include the following block at the end of `pmod.py`:

```
if __name__ == "__main__":
    p1 = Point(6, 8)
    print("p1.radius() =", p1.radius())
    print("p1.rect_area() =", p1.rect_area())
    print("p1.circ_area() =", p1.circ_area())
    p2 = Point(3, 4)
    print("p2.radius() =", p2.radius())
    print("p2.rect_area() =", p2.rect_area())
    print("p2.circ_area() =", p2.circ_area())
    print("p1.x = {0}, p1.y = {1}".format(p1.x, p1.y))
    print("p2.x = {0}, p2.y = {1}".format(p2.x, p2.y))
    help(Point.radius)
```

If we load `pmod.py` as a module, the `if __name__ == "__main__":` statement will return false, and the code below it will not be executed. On the other hand if we type `%run pmod.py` from the IPython console, the code after the `if` will execute as shown here:

```
%run pmod.py
p1.radius() = 10.0
p1.rect_area() = 192.0
p1.circ_area() = 314.1592653589793
p2.radius() = 5.0
p2.rect_area() = 48.0
p2.circ_area() = 78.53981633974483
p1.x = 6, p1.y = 8
p2.x = 3, p2.y = 4
Help on function radius in module __main__:

radius(self)
    returns distance from (x, y) to origin
```

This serves to show how the module works.

Before moving on, now might be a good time to go back to page 286 and review how the Point class and its methods are used.

10.2 A BRIEF INTRODUCTION TO MODULES AND PACKAGES

In the previous section, we created a class called `Point` and stored in the file `pmod.py`. As demonstrated above, one way to access the `Point` class is to import `pmod.py`, which is a Python file, as a Python *module*.

For a module to be imported into a Python program, it must be in a location where Python knows to look. By default, Python always looks in the same directory as the Python program that called it. That's why the program we wrote could find the `pmod` module, because the `pmod.py` file was in the same directory that we ran the IPython QtConsole from (this means that writing `ls` from the IPython prompt should list `pmod.py`). If you are content to use modules in this way, by keeping them in the same directory from which they are called, you can skip the rest of this section. However, keep reading if you want to learn more about storing Python modules and organizing them into packages.

10.2.1 Pythonpath

If Python doesn't find a module in the calling directory, it looks in a list of directory names stored in a system environmental variable called `PYTHONPATH` (all capital letters). You can get a list of these directories for your computer by importing the `sys` module and then typing `sys.path`.

```
In[1]: import sys

In[2]: sys.path
Out[2]:
['/Users/dp',
'/Users/dp/anaconda3/lib/python311.zip',
'/Users/dp/anaconda3/lib/python3.11',
'/Users/dp/anaconda3/lib/python3.11/lib-dynload',
'',
'/Users/dp/anaconda3/lib/python3.11/site-packages',
'/Users/dp/anaconda3/lib/python3.11/site-packages/aeosa']
```

The command `sys.path` produces a list of all the directory names (with the full path) contained in `PYTHONPATH`. You can add directory names to this list and then use those directories to store modules that you write. However, you need to be organized about how you do this.

To illustrate, we will add a single directory that we call `mypy`, which we will use to store any custom modules we write. While the directory `mypy` can go almost anywhere in my computer's directory tree, we choose to make it a subdirectory of `/Users/dp`, the home directory, which also contains the `/anaconda3` directory.

Table 10.1 shows an example of a directory tree for custom modules:

TABLE 10.1 Directory structure for storing custom Python modules.

`/Users/dp/mypy/`	directory containing packages
└── `pythonbook/`	first package
└── `__init__.py`	empty file
└── `pmod.py`	pmod module
└── `shapes.py`	shapes module
└── `measurement.py`	measurement module
└── `simulations.py`	simulations module
└── `apak/`	another package
└── `__init__.py`	empty file
└── `fill.py`	fill module
└── `randomize.py`	randomize module

The topmost directories in `/Users/dp/mypy/`, `pythonbook` and `apak`, are Python *packages*. Each package contains several modules, which are Python files. In addition, each package directory contains an `__init__.py` (with two underline characters before and after `init`), which lets Python know to treat these directories as packages. In this case, the `__init__.py` files are empty, although they can contain commands that initialize the package, something we will not concern ourselves with here. The top-level directory `mypy` is not a package and thus does not have an `__init__.py` file.

Each Python file comprising a module contains one or more classes or function definitions. Once the directory `mypy` is added to `PYTHONPATH`,[2] these modules are available to be used in programs you write using the usual syntax for imported modules. For example, the Point class contained in pmod.py could be accessed in any one of the usual ways:

```
 In[3]: from pythonbook import pmod as pm

 In[4]: p1 = pm.Point(6, 8)

 In[5]: p1.radius()
Out[5]: 10.0
```

Alternatively, you can use the familiar dot syntax:

```
 In[6]: import pythonbook.pmod as pm

 In[7]: p1 = pm.Point(6, 8)

 In[8]: p1.radius()
Out[8]: 10.0
```

or:

```
 In[9]: import pythonbook as pb

 In[10]: p1 = pb.pmod.Point(6, 8)

 In[11]: p1.radius()
Out[11]: 10.0
```

You can make as many packages as you want, each with its own set of modules. You can even make subpackages using subdirectories, although we will leave this to you to explore on your own by reading the appropriate online Python documentation.

[2]See Appendix A.4 for instructions about how to add the directory `mypy` to `PYTHONPATH`. How to do this depends on which operating system you are using.

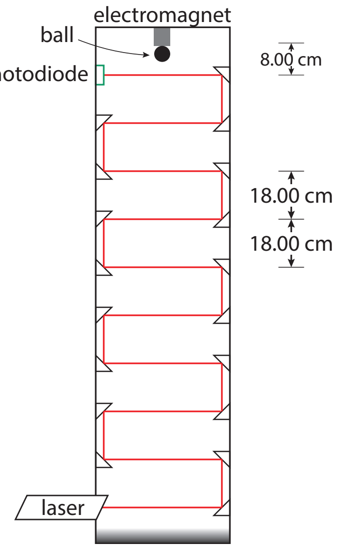


Figure 10.1 Apparatus for measuring free fall with air resistance.

10.3 A CLASS FOR READING AND PROCESSING DATA

Now that you've seen how a very simple class works, let's make a more useful class. The purpose of the class is to read numerical data from text files and process the data in various ways. This might involve performing calculations to characterize, query, transform, plot, or model the data. A Python class is well suited to such a task and can be very useful, particularly if you need to process many similar data sets.

For this example, we consider an experiment designed to measure the effect of air resistance on a freely falling spherical ball. In the experiment, a spherical ball is released by an electromagnet so that it falls freely nearly two meters until it falls onto a piece of foam rubber, as illustrated in Figure 10.1. As it falls, the ball periodically breaks a laser beam that crisscrosses the ball's path using a series of mirrors. The mirrors are precisely placed a fixed distance apart, 18.00 cm in this case. A computer-based timing circuit records the times when the laser path is broken and when it is restored as the front and back sides of the falling ball pass through the laser beam. These crossing times are recorded with an estimated uncertainty of ±0.5 ms.

The equation governing the (positive) distance y a spherical ball falls in a time t is given by

$$y = \frac{v_t^2}{g} \ln \left(\cosh \frac{gt}{v_t} + \frac{v_0}{v_t} \sinh \frac{gt}{v_t} \right) , \tag{10.1}$$

where g is the acceleration due to gravity, v_0 is the velocity at $t = 0$, and v_t is the terminal velocity. In the ideal limit where air resistance is negligible, $v_t \to \infty$, and this formula reduces to the familiar equation

$$y = v_0 t + \tfrac{1}{2} g t^2 . \tag{10.2}$$

The terminal velocity is given by

$$v_t = \frac{2mg}{C_d \pi R^2 \rho_a} , \tag{10.3}$$

where m is the mass of the sphere, R is its radius, ρ_a is the density of air, and C_d is an empirical drag coefficient, which for spheres is 0.47.

10.3.1 The Data

First, let's consider how the data will be stored. We use two files. The first is a text data file containing a column of numbers corresponding to the measured crossing times in seconds.

Data: plastic1.txt

```
0.000000
0.044784
0.126177
0.147594
0.200606
0.217874
0.260726
0.273778
0.311248
0.322659
0.354883
0.366229
0.396670
0.405874
0.434361
0.443185
0.469306
0.477919
0.502953
0.511296
```

The second file stores metadata about this experiment. It contains information about things like who acquired the data, the date and time it was acquired, and information about the measured sample, for example, the ball's mass and radius. The metadata file is also a text file, here named `plastic1.yaml`; its `.yaml` extension designates it as a YAML file, a type of text file whose properties we describe below.

Data: plastic1.yaml

```
---
data_aquired_by: Jeana Cui
date: 2022-07-21
time: "16:07"
material: plastic
mass_gm: 14.40
radius_cm: 2.29
magnet_to_laser_cm: 8.00
gravity_acceleration_si: 9.795
density_air_si: 1.1839
drag_coefficient: 0.47
laser_spacing_cm: 18.0
time_uncertainty_s: 0.0005
...
```

Both the text data file (`plastic1.txt`) and the metadata text YAML file (`plastic1.yaml`) have the same root name. Thus, we know that both files refer to the same experimental measurement.

A YAML file starts with three dashes and ends with three periods. The lines in between are used to build a Python dictionary. In this case, each line is a new entry in the dictionary. The string to the left of the colon defines the keys of the dictionary; the text to the right defines the values. YAML files can have other features as well, but the above description will suffice for our purposes. The appeal of such a YAML file is its simplicity. The virtue of using a Python dictionary to store information is its versatility and adaptability. You can add entries to your YAML file without ruining (breaking) how your code works.

10.3.2 The Class

Before learning how to write the code for our new class, let's see how the finished class works. The name of our class is `FallingBall`; it is stored in a module called `measurement`. The `FallingBall` class takes one argument, a string corresponding to the base name of the text and YAML files containing the data and metadata.

```
In[1]: import measurement as m

In[2]: p1 = m.FallingBall("plastic1")
```

The first statement above imports the module `measurement` and gives it the abbreviation of `m`. Again we note that for the `import` statement to work, the file `measurement.py` must be in Python's path (see Section 10.2.1).

The second statement instantiates the class and gives this instance the tag `p1`. The class takes one argument, `"plastic1"`, the base name of both the text data file and the YAML metadata file discussed above. The `plastic1.txt` and `plastic1.yaml` data files are read, and their contents are stored in memory when the class is instantiated.

The `FallingBall` class has a few attributes, such as the initial and terminal velocities, that are determined from the data and metadata when they are read in:

```
 In[3]: p1.v0   # initial velocity
Out[3]: 0.8185218384380468

 In[4]: p1.vt   #terminal velocity
Out[4]: 17.542100957438386
```

The base filename is also an attribute

```
 In[5]: p1.filename
Out[5]: 'plastic1'
```

The data read in from the data (TXT) file, the laser crossing times, are contained in the attribute `crossing_times` and can be accessed like any other attribute:

```
 In[6]: p1.crossing_times
Out[6]:
array([0.       , 0.044784, 0.126177, 0.147594, 0.200606,
       0.217874, 0.260726, 0.273778, 0.311248, 0.322659,
       0.354883, 0.366229, 0.39667 , 0.405874, 0.434361,
       0.443185, 0.469306, 0.477919, 0.502953, 0.511296])
```

You can get a list of the attributes of a user-defined class using the (so-called) "magic" class method `__dict__`. To get a listing of all the attribute keys of a class, type the following:

```
 In[7]: print(p1.__dict__.keys())
dict_keys(['filename', 'meta', 'crossing_times', 'v0', 'vt',
'y_laser', 'yp', 'y_nofric', 'dy', 'dy_err'])
```

You can list all the attribute values by typing `print(p1.__dict__.values())`.

The `FallingBall` class has two other methods. The `mdata()` method prints out the metadata read in from the YAML file:

```
 In[8]: p1.mdata()
Out[8]:
data_aquired_by: Jeana Cui
```

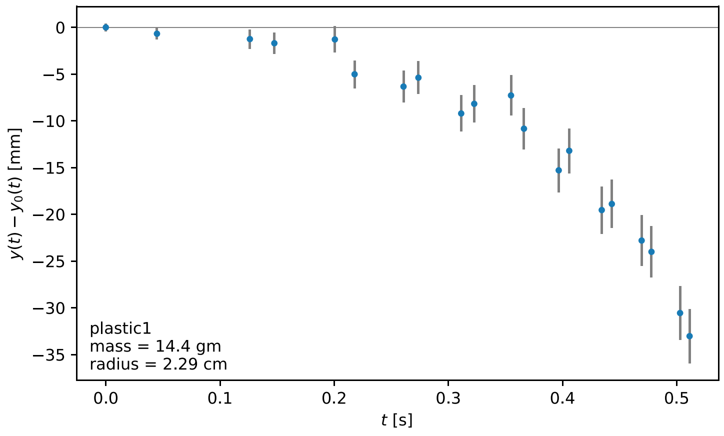


Figure 10.2 Plot produced by the `plot()` method of the `FallingBall` class.

```
date: 2022-07-21
time: 16:07
material: plastic
mass_gm: 14.4
radius_cm: 2.29
magnet_to_laser_cm: 8.0
gravity_acceleration_si: 9.795
density_air_si: 1.1839
drag_coefficient: 0.47
laser_spacing_cm: 18.0
time_error_s: 0.0005
```

The `plot` method plots $y(t) - y_0(t)$, the distance the ball has fallen minus the distance it would be expected to fall if there were no air resistance as a function of time.

```
In[9]: p1.plot()
```

The result is shown in Figure 10.2. By default, the plot is displayed on the screen but not saved to disk. To save the file, the `plot()` method takes an argument, either `plot(True)` or `plot(save=True)`, which causes the method to save the plot to disk with the name `plastic1.pdf`.

10.3.3 The Code

The `FallingBall` class described above has one input, the base filename of the text and YAML files on which the class acts. It has three methods: the constructor `__init__()`, which is called when the class is instantiated, and the two methods associated with the class, `mdata` and `plot()`. Thus, the basic structure of the class looks like this:

Code: measurement_shell.py

```python
class FallingBall:

    def __init__(self, filename):
        pass

    def mdata(self):
        pass

    def plot(self, save=False):
        pass
```

Next, we need to fill in the code. The code is contained in a Python file `measurement.py` (see below). At the top of the file, NumPy, Matplotlib, and Yaml are imported as they will be used in the `FallingBall` class code. Importing them outside the class definition makes these packages available to any code within the `measurement.py` file, including the `FallingBall` class.

Code: measurement.py

```python
import numpy as np
import matplotlib.pyplot as plt
import yaml


class FallingBall:
    """
    A class to analyze data from falling ball experiment with air
    resistance
    """

    def __init__(self, filename):
        """Read data from & metadata from txt and yaml files"""
        self.filename = filename

        # open YAML file and read in metadata dictionary
        fmeta = open(filename + ".yaml", "r")
        self.meta = yaml.safe_load(fmeta)
        fmeta.close()

        # open text file and read in crossing times data
        self.crossing_times = np.loadtxt(filename + ".txt",
                                         unpack=True, delimiter=",")

        # determine initial velocity v0
        radius = self.meta["radius_cm"] / 100.0  # [m] ball radius
        y00 = self.meta["magnet_to_laser_cm"] / 100.0  # [m]
        g = self.meta["gravity_acceleration_si"]  # [m/s^2]
        mass = self.meta["mass_gm"] / 1000.0  # [kg]
        y0 = y00 - 2.0 * radius  # [m] fall to first laser
        self.v0 = np.sqrt(2.0 * g * y0)  # [m/s] initial velocity

        # determine terminal velocity vt
        cdrag = self.meta["drag_coefficient"]
        density_air = self.meta["density_air_si"]  # [kg/m^3]
        drag = 0.5 * cdrag * np.pi * radius**2 * density_air
        self.vt = np.sqrt(mass * g / drag)  # [m/s] terminal velocity

        # determine laser y-coordinates
        dy = self.meta["laser_spacing_cm"] / 100.0  # [m]
        nlines = self.crossing_times.size // 2
        self.y_laser = np.linspace(0.0, nlines-1, nlines) * dy

        # determine particle y-positions
        y2 = self.y_laser + 2.0 * radius
        yp = np.concatenate((self.y_laser, y2))
        self.yp = np.sort(yp)

        # determine ideal particle y-positions with no friction
        self.y_nofric = self.crossing_times * (self.v0 + 0.5 * g *
                                               self.crossing_times)

        # determine difference from ideal no-friction displacements
        self.dy = self.yp - self.y_nofric

        # determine uncertainties in difference displacements
        v = self.v0 + g * self.crossing_times
        self.dy_err = np.abs(v * self.meta["time_uncertainty_s"])

    def mdata(self):
        """Print out nicely-formatted metadata"""
        for key, value in self.meta.items():
            print(key + ": " + str(value))

    def plot(self, save=False):
        fig, ax = plt.subplots(figsize=(6.5, 4.0))
        ax.errorbar(self.crossing_times, 1.0e3 * self.dy, fmt="oC0",
                    yerr=1.0e3 * self.dy_err, ecolor="gray", ms=3)
        ax.axhline(color="gray", lw=0.5, zorder=-1)
        ax.set_xlabel(r"$t$ [s]")
        ax.set_ylabel(r"$y(t) - y_0(t)$ [mm]")
        txt = self.filename
        txt += "\n" + f"mass = {self.meta['mass_gm']:0.1f} gm"
        txt += "\n" + f"radius = {self.meta['radius_cm']:0.2f} cm"
        ax.text(0.02, 0.02, txt, ha="left", va="bottom",
                transform=ax.transAxes)
        fig.tight_layout()
        if save:
            fig.savefig("figures/" + self.filename + ".pdf")
        fig.show()
```

The `FallingBall` class is defined with the `class` statement on line 6, followed by a brief docstring.

The `__init__()` method reads in the metadata and data, and it needs to create the method's attributes, which include the laser crossing times, the initial and terminal velocities, the root name of the data files, and several other items.

The metadata is read from the YAML file with the base filename specified when the `FallingBall` class is instantiated. Note that the YAML `safe_load()` method is used rather than the alternative YAML `load()` method. It is unsafe to call `yaml.load()` with data received from an untrusted source, as a YAML file can call any Python function, including those that might damage your computer. The YAML `safe_load()` method limits this feature and should always be used unless you trust the source of the YAML file *and* you have a reason to use YAML's advanced features. Here, we need to load a dictionary containing metadata, so we don't need YAML's advanced features. Be sure to close the YAML file once you have finished reading in its metadata.

Next, the `__init__()` method reads the data file, which in this case consists of a single column of data (the crossing times). We use NumPy's `loadtxt` function, which we introduced in Section 5.3.

The remainder of the `__init__` method processes the data and metadata to put all of it in a useful form for analysis and display. From the spacing of the laser beam crossing the path of the falling mass and the crossing times, it determines the coordinates of the mass at different times as it falls. It also uses the metadata to determine where the particle would be at these times without air resistance. From the uncertainty of the crossing-time measurements, it also calculates the uncertainties in the measured particle positions.

The `__init__()` is run when the `FallingBall` class is instantiated.

Once the class is instantiated, two other methods are available that act on the data. The `mdata` method displays the metadata nicely formatted for examination by the user. It takes no arguments.

The `plot` method plots $y(t) - y_0(t)$, the position of the falling mass minus $y(t)$ what its position would be $y_0(t)$ if there were no air resistance. The `plot` method has one argument, the keyword argument `save`. By default, it is set to be `False` and does not save the plot to your computer's drive. However, setting the keyword argument `save=True`, the plot is saved to a PDF file with the same base name as the YAML and TXT data files that were read with the class was instantiated.

10.4 A CLASS OF RELATED FUNCTIONS

Occasionally, you may want to code a number of related functions, perhaps pertaining to some model or set of models you wish to examine and use. In such a case, the functions may share at least some data and parameters. While

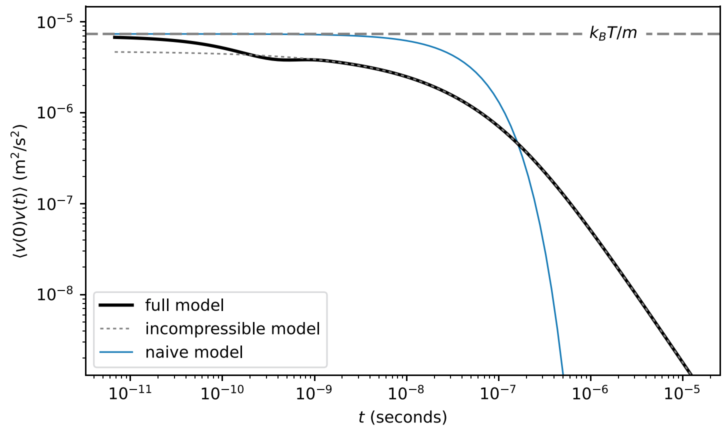


Figure 10.3 Velocity autocorrelation functions.

you can define sets of data and variables along with a set of appropriate functions, it is often useful to bundle the data and functions within a class.

Doing so has several advantages. First, it organizes the related set of functions and the data they use into a single easily recognizable unit, a class. Second, it organizes the necessary data inputs into a convenient list contained in the arguments of the class (*i.e.*, the arguments of its `__init__()` method). This makes recognizing and understanding the inputs easier for you and others. Finally, it isolates the data and functions from other code and thus protects the internal functioning of the data and functions (methods) that operate on the data from the other software you may need.

These are the classic elements of *encapsulation*, which the class structure nicely provides.

We illustrate this kind of functionality with a class that calculates the velocity autocorrelation function, which is used in statistical mechanics to characterize the Brownian motion of microscopic particles suspended in a liquid. The exact nature of the function is not important here. The essential thing you need to know is that it is a function of time and that various theoretical models result in different functional forms for the velocity autocorrelation function. In addition, the velocity autocorrelation function depends on a number of physical parameters: the particle diameter (the particles are assumed to be spheres), the mass densities of the particle and liquid, the viscosity of the liquid, the velocity of sound in the liquid, and the temperature. The three models of the velocity autocorrelation function we wish to consider are plotted in Figure 10.3.

The code for the class, which we name `VVautocorr`, is provided below. The class consists of the constructor `__init__()` and three methods, one for each model. As noted above, the arguments of the class (*i.e.*, of `__init__()`) consist of a list of all the parameters on which the models depend. All of these inputs are made attributes of the class by `__init__()`. In addition, `__init__()` creates a few other attributes, including three characteristic time scales, `self.tauVort`, `self.tauSound`, and `self.tauVisc`, as well as the mass of the particle `self.mass_p` and of the fluid it displaces `self.mass_f`.

The class has three methods `hinchVV()`, `langevinVV()`, and `zwanzigVV()`. Besides the `self` argument, each method has only one argument, `time`, which can be a NumPy array or a single value of time. The `self` argument ensures that all the attributes are available to each method.

While the attributes are freely available within the `VVautocorr` class, they are isolated from other software outside of the class and are available only through the usual dot syntax for a particular instance of the class.

Code: vv.py

```python
import numpy as np
import scipy.special
import matplotlib.pyplot as plt
from scipy.constants import Boltzmann as kB


class VVautocorr():
    """
    Calculates velocity autocorrelation function of colloidal particle
    suspended in a liquid.  All units are in SI.
    """

    def __init__(self, diameter=1.0e-6, dens_p=1050.0,  dens_f=998.2,
                 viscosity=1.0016e-3, vel_sound=1484.0, tempC=20.0):
        self.diameter = diameter           # particle diameter
        self.radius = 0.5 * self.diameter
        self.dens_p = dens_p               # particle mass density
        self.dens_f = dens_f               # fluid mass density
        self.viscosity = viscosity         # fluid viscostiy
        self.vel_sound = vel_sound         # sound velocity in fluid
        self.tempC = tempC                 # temperature degrees C
        self.tempK = tempC + 273.15
        self.mass_p = 4.0 * np.pi * self.radius**3 / 3.0 * self.dens_p
        self.mass_f = 4.0 * np.pi * self.radius**3 / 3.0 * self.dens_f
        self.mstar = self.mass_p + 0.5 * self.mass_f
        zeta = 6.0 * np.pi * self.viscosity * self.radius
        self.tauVort = dens_f * self.radius**2 / self.viscosity
        self.tauSound = self.radius / self.vel_sound
        self.tauVisc = self.mass_p / zeta

    def hinchVV(self, time):  # [seconds] NumPy array or single value
        # Velocity autocorrelation function from
        # Hinch, J. Fluid Mech. 72, 499 (1975)
        rhoRatio = self.dens_p / self. dens_f
        alpha = 1.5 / (np.sqrt(self.tauVort) * (1.0 + 2.0 * rhoRatio))
        dsc = np.sqrt((5.0 - 8.0 * rhoRatio) + 0j)
        alphaPlus = alpha * (3.0 + dsc)
        alphaMinus = alpha * (3.0 - dsc)
        A = kB * self.tempK / (2.0 * np.pi * self.radius**3 *
                            self.dens_p * dsc) * np.sqrt(self.tauVort)
        roott = np.sqrt(time)
        RPlus = alphaPlus * np.exp(alphaPlus * alphaPlus * time) \
                          * scipy.special.erfc(alphaPlus * roott)
        RMinus = alphaMinus * np.exp(alphaMinus * alphaMinus * time) \
                            * scipy.special.erfc(alphaMinus * roott)
        # Imaginary part is zero
        return np.real(A * (RPlus - RMinus))

    def langevinVV(self, time):
        # Velocity autocorrelation function from naive Langevin eqn
        vsqrd = kB * self.tempK / self.mass_p
        return vsqrd * np.exp(-time / self.tauVisc)

    def zwanzigVV(self, time):
        # Compressibility effect on velocity autocorrelation function
        # Zwanzig & Bixon, J. Fluid Mech. 69, 21 (1975)
        t = time / self.tauSound
        mratio = self.mstar / self.mass_p
        msqrt = np.sqrt(0j + 1.0 - (0.5 * self.mass_f/self.mass_p)**2)
        mrati = 1j * mratio / msqrt
        x1 = -1j * mratio + msqrt
        x2 = -1j * mratio - msqrt
        w1 = (1.0 - mrati) * np.exp(-1j * x1 * t)
        w2 = (1.0 + mrati) * np.exp(-1j * x2 * t)
        w = (0.25 * self.mass_f / self.mstar) * np.real(w1 + w2)
        return (self.mass_p / self.mstar) + w


if __name__ == "__main__":
    a = VVautocorr()  # instantiate with default variable values

    npts = 100
    tmin = 0.02 * a.tauSound
    tmax = 50.0 * a.tauVort
    t = np.logspace(np.log10(tmin), np.log10(tmax), npts)
    vvHinch = a.hinchVV(t)
    vvZwanzig = a.zwanzigVV(t)
    full = (a.mstar / a.mass_p) * vvHinch * vvZwanzig
    vvLangevin = a.langevinVV(t)

    fig, ax = plt.subplots(figsize=(6.5, 4.0))
    ax.set_xscale("log")
    ax.set_yscale("log")
    ax.plot(t, full, lw=2, color="k", label="full model")
    ax.plot(t, vvHinch, lw=1, color="gray",
            dashes=(2, 2), label="incompressible model")
    ax.plot(t, vvLangevin, lw=1, color="C0",
            zorder=-2, label="naive model")
    vSqrd = kB * a.tempK / a.mass_p
    ax.set_ylim(vvHinch[-1], 2.0 * vSqrd)
    ax.set_xlabel(r"$t\ \mathrm{(seconds)}$")
    ax.set_ylabel(r"$\langle v(0)v(t) \rangle\ \mathrm{(m^2/s^2)}$")
    # Equipartition line
    ax.axhline(vSqrd, color="gray", dashes=(5, 2))
    ax.text(t[-10], vSqrd, "$k_BT/m$", va="center", ha="right",
            bbox=dict(fc="white", ec="white"))
    ax.legend()
    plt.tight_layout()
    plt.savefig("figures/vv.pdf")
    plt.show()
```

The use of the class is illustrated by the code following the `if __name__ == "__main__":` statement. This code following this statement is executed only if the file `vv.py` containing the code is run as a standard Python file. It is not run, however, if `vv.py` is imported as a module. This allows you to write code testing a module without affecting the module's normal use.

The `VVautocorr()` class is instantiated on line 70 without any arguments, that is, using the default values of the arguments specified in the class constructor `__init__()`.

10.5 INHERITANCE

The previous sections in this chapter provide an introduction to classes, particularly to the essential idea of how classes can be used to bundle data with methods (functions) that act on those data. This concept is called encapsulation. The next most significant concept of classes is *inheritance*, which provides a mechanism for building up a hierarchy of related classes that are part of a larger system.

In a laboratory, for example, you might have a set of instruments used to characterize various materials. Each instrument has its own unique capabilities, but the whole set of instruments is needed to characterize a physical, chemical, or biological system. Software interacting with each of the various instruments might look pretty similar and might interact with the data from each in similar ways. Moreover, you might like to share information from the different instruments and analyze them together. Inheritance provides a convenient set of software tools to create a hierarchy of classes for these related instruments. The highest level of the hierarchy sets the basic structure of software that interacts with each instrument. Software lower in the hierarchy *inherits* the basic structure from the higher levels and customizes it for each instrument.

Describing how to build up such a system goes well beyond the scope of this text. However, if you start developing a system of related classes, you will want to investigate inheritance and the broader subject of software system design.

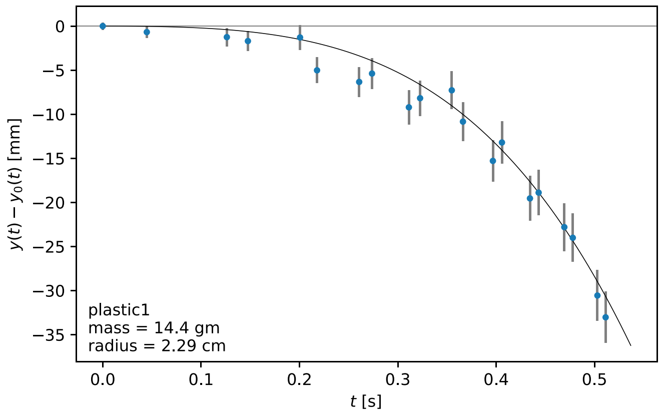
10.6 EXERCISES

1. Write a class called `Sphere` that determines the geometrical properties of a sphere. The single input should be the radius. Create methods `get_radius`, `surface_area`, and `volume` that return the radius, surface area, and volume of a sphere. Then, demonstrate that your new class works for spheres of radius 1, 3, and 10.

2. Add a method to the class `FallingBall` that returns the *y* coordinate of a falling ball given by Eq. (10.1). Other than `self`, the method's only argument should be the time. Your new method should access All other necessary parameters through the class attributes.

   Then, add an argument to the `FallingBall` `plot` method that, if `True`, plots the fit of the model to the data using the new method. The default value should be `False`.

   Demonstrate the use of your class by importing it and plotting the data from the `plastic1` data and metadata files. Your output should look like this plot.



CHAPTER 11

Data Manipulation and Analysis: Pandas

This chapter introduces Pandas, a powerful Python package for manipulating and analyzing large (and small) data sets. You first learn how to ***read data*** *from external files, e.g., Excel or text files, into Pandas. You learn about different* ***data structures*** *for storing dates and times,* ***time series****, and data organized into rows and columns in a spreadsheet-like structure called a* ***DataFrame****. You then learn how to manipulate data, extract subsets of data, and plot those data using Matplotlib, but with some new syntax introduced by Pandas that facilitates working with data the structures of Pandas.*

This chapter introduces Pandas, a versatile Python package for handling large data sets. It has a spreadsheet-like character and has become a standard tool for *data scientists*, people working across a wide range of disciplines who collect, analyze, and interpret data from a variety of inputs. While Pandas was developed by people working in the financial industry, many of its features are generally useful to scientists and engineers. We can't cover all of its capabilities in one short chapter, but we will show you a few of the things it can do. With that introduction, we hope that you will have learned enough to adapt Pandas to your own applications.

Pandas is installed with the standard Python distributions such as Anaconda. In an IPython shell or in a Python program, you access the many routines available in Pandas by writing:

```
import pandas as pd
```

The abbreviation universally used for Pandas is `pd`.

DOI: 10.1201/9781032673950-11

11.1 DATA STRUCTURES: SERIES AND DataFrame

Pandas has two principal data structures: *Series* and *DataFrame*. They form the basis for most activities using Pandas.

Both Series and DataFrames use NumPy arrays extensively, but introduce more versatile ways of indexing and manipulating different data types. This added functionality can come at a performance price, a topic we briefly address below. You will find that NumPy is more adept at numerical work while Pandas excels at managing large complex data sets.

11.1.1 Series

A Pandas Series is a one-dimensional array-like data structure comprising a NumPy array, and an associated array of data labels called the *index*. We can create a Series using the Pandas `Series` function, which turns a list, dictionary, or NumPy array into a Pandas Series. Here, we use it to turn a list into a Series:

```
 In[1]: lst = [160.0-4.9*t*t for t in range(6)]

 In[2]: lst
Out[2]: [160.0, 155.1, 140.4, 115.9, 81.6, 37.5]

 In[3]: ht = pd.Series(lst)

 In[4]: ht
Out[4]:
0     160.0
1     155.1
2     140.4
3     115.9
4      81.6
5      37.5
dtype: float64
```

The IPython output displayed is in two columns: the index column on the left and the values of the Series on the right. The argument of the `Series` function can be a list, an iterator, or a NumPy array. In this case, the values of the Series are floating point numbers. The index goes from 0 to $N - 1$, where N is the number of data points. Individual elements and slices are accessed in the same way as for lists and NumPy arrays.

```
 In[5]: ht[2]
Out[5]: 140.4

 In[6]: ht[1:4]
Out[6]:
1     155.1
```

```
2     140.4
3     115.9
dtype: float64
```

The entire array of values and indices can be accessed using the `values` and `index` attributes.

```
 In[7]: ht.values
Out[7]: array([ 160. , 155.1, 140.4, 115.9,  81.6,  37.5])

 In[8]: ht.index
Out[8]: RangeIndex(start=0, stop=6, step=1)
```

Here, as can be seen from the outputs, the `ht.values` is a NumPy array and `ht.index` is an iterator.

11.1.1.1 *Alternative Indexing Schemes*

Unless otherwise specified, the index default is to go from 0 to $N-1$ where N is the size of the Series. However, there are other ways of indexing. Consider a Series that stores the heights of different individuals:

```
 In[9]: height = pd.Series([188, 157, 173, 169, 155],
   ...:                    index=['Jake', 'Sarah', 'Maya',
   ...:                           'Chris', 'Alex'])

 In[10]: height
Out[10]:
Jake      188
Sarah     157
Maya      173
Chris     169
Alex      155
dtype: int64
```

Here the `index` keyword argument specifies that the heights are indexed by people's names.

```
 In[11]: height['Maya']
Out[11]: 173
```

The old indexing scheme still works in this case:

```
 In[12]: height[2]
Out[12]: 173
```

The Series `height` bears a striking resemblance to a Python dictionary. Indeed, a Pandas Series can be converted to a dictionary using the `to_dict` method:

```
 In[13]: htd = height.to_dict()

 In[14]: htd
Out[14]: {'Jake': 188, 'Sarah': 157, 'Maya': 173,
         'Chris': 169, 'Alex': 155}
```

Coming full circle, we see that a Python dictionary can be converted to a Pandas Series using the `Series` function:

```
 In[15]: pd.Series(htd)
Out[15]:
Jake      188
Sarah     157
Maya      173
Chris     169
Alex      155
dtype: int64
```

11.1.1.2 Time Series

One of the most common uses of Pandas Series involves a *time* Series in which the Series is indexed by timestamps. This is facilitated by another Python library called datetime, which is distinct from Pandas and defines, among other things, a useful *datetime* object. A datetime object stores, as its name implies, a precise moment in time. To see how this works, let's import the datetime library and then get the current value of `datetime`:

```
 In[16]: import datetime as dt

 In[17]: t0 = dt.datetime.now()

 In[18]: t0
Out[18]: datetime.datetime(2017, 7, 21, 8, 17, 24, 241916)
```

The datetime object returns the year, month, day, hour, minute, second, and microsecond. You can format a datetime object for printing using the strftime method of the datetime library in various ways:

```
 In[19]: t0.strftime('%Y-%m-%d')
Out[19]: '2017-07-21'

 In[20]: t0.strftime('%d-%m-%Y')
Out[20]: '21-07-2017'

 In[21]: t0.strftime('%d-%b-%Y')
Out[21]: '21-Jul-2017'

 In[22]: t0.strftime('%H:%M:%S')
```

```
Out[22]: '09:00:15'

 In[23]: t0.strftime('%Y-%B-%d %H:%M:%S')
Out[23]: '2017-July-21 09:00:15'
```

You can construct almost any format you want.

Datetime objects can be used to index Pandas Series. For example, we can change the index of the series `ht` to consecutive days. First, we create a time sequence:

```
 In[24]: dtr = pd.date_range('2017-07-22', periods=6)
```

Then we set the index of `ht` to the time sequence:

```
 In[25]: ht.index = dtr

 In[26]: ht
Out[26]:
2017-07-22    160.0
2017-07-23    155.1
2017-07-24    140.4
2017-07-25    115.9
2017-07-26     81.6
2017-07-27     37.5
Freq: D, dtype: float64
```

```
 In[27]: ht['2017-07-25']
Out[27]: 115.90000000000001

 In[28]: ht['2017-07-23':'2017-07-26']
Out[28]:
2017-07-23    155.1
2017-07-24    140.4
2017-07-25    115.9
2017-07-26     81.6
Freq: D, dtype: float64
```

Note that you can slice time Series indexed by dates, and that the slice range includes both the starting and ending dates.

11.1.2 DataFrame

A Pandas DataFrame is a two-dimensional spreadsheet-like data structure. It consists of an index column and two or more data columns; in effect, it's a Series with more than one data column.

One simple way to generate a DataFrame is from tabular data stored in a text file. Consider, for example, the following data about the planets tabulated in a simple text file. The quantities in the table are referenced to Earth.

Data: planet_data.txt

```
planet      distance      mass   gravity  diameter     year
Mercury         0.39     0.055      0.38      0.38     0.24
Venus           0.72      0.82      0.91      0.95     0.62
Earth           1.00      1.00      1.00      1.00     1.00
Mars            1.52      0.11      0.38      0.53     1.88
Jupiter         5.20       318      2.36      11.2     11.9
Saturn          9.58        95      0.92      9.45       29
Uranus          19.2        15      0.89      4.01       84
Neptune         30.0        17      1.12      3.88      164
Pluto           39.5    0.0024     0.071      0.19      248
```

We can read this table into a DataFrame using the Pandas function `read_table()`. Notice below that we use the keyword argument `sep`, which specifies the character or characters that Pandas uses to separate the columns when the table is read. In this case, `sep="\s+"`, which sets the column separator to by one or more spaces. You would use `sep=","` for comma-separated columns, although there also exists a dedicated Pandas function `read_csv()` for that purpose.

```
In[1]: planets = pd.read_table('planet_data.txt', sep='\s+')

In[2]: planets
Out[2]:
planet  distance      mass  gravity  diameter    year
0  Mercury      0.39    0.0550    0.380      0.38    0.24
1    Venus      0.72    0.8200    0.910      0.95    0.62
2    Earth      1.00    1.0000    1.000      1.00    1.00
3     Mars      1.52    0.1100    0.380      0.53    1.88
4  Jupiter      5.20  318.0000    2.360     11.20   11.90
5   Saturn      9.58   95.0000    0.920      9.45   29.00
6   Uranus     19.20   15.0000    0.890      4.01   84.00
7  Neptune     30.00   17.0000    1.120      3.88  164.00
8    Pluto     39.50    0.0024    0.071      0.19  248.00
```

The `pd.read_table` function created a Pandas DataFrame to which we assigned the name `planets`. Notice that Pandas added a numerical index (the first column) to designate the rows, just as it does for Series. These numbers are used for indexing the rows of the DataFrame.

By default, the Pandas function `pd.read_table` uses the names in the top row of the data file as names by which the respective columns are indexed. For example, the `mass` column is indexed as follows:

```
In[3]: planets["mass"]
Out[3]:
0       0.0550
1       0.8200
2       1.0000
3       0.1100
```

```
4    318.0000
5     95.0000
6     15.0000
7     17.0000
8      0.0024
Name: mass, dtype: float64

 In[4]: type(planets["mass"])
Out[4]: pandas.core.series.Series
```

Notice that a single column of the DataFrame, in this case `planets["mass"]`, is a Series. As such, its elements can be indexed in the same way as for stand-alone Series.

```
 In[5]: planets["mass"][4]
Out[5]: 318.0
```

Slicing also works.

```
planets["mass"][2:5]
Out[5]37:
2      1.00
3      0.11
4    318.00
Name: mass, dtype: float64
```

Slicing does not work on the columns. However, in Section 11.2, we will learn more efficient ways of indexing that will get around this problem.

Using the following code, you can designate the `planet` column to be the index instead of the numbers.

```
 In[6]: planets = planets.set_index("planet")

 In[7]: planets
Out[7]:
distance       mass   gravity  diameter    year
planet
Mercury       0.39    0.0550    0.380      0.38    0.24
Venus         0.72    0.8200    0.910      0.95    0.62
Earth         1.00    1.0000    1.000      1.00    1.00
Mars          1.52    0.1100    0.380      0.53    1.88
Jupiter       5.20  318.0000    2.360     11.20   11.90
Saturn        9.58   95.0000    0.920      9.45   29.00
Uranus       19.20   15.0000    0.890      4.01   84.00
Neptune      30.00   17.0000    1.120      3.88  164.00
Pluto        39.50    0.0024    0.071      0.19  248.00
```

With this change, the planet names index the various rows. Here, we display the diameter of the planets from Venus to Neptune:

```
In[8]: planets["diameter"]["Venus":"Neptune"]
Out[8]:
planet
Venus       0.95
Earth       1.00
Mars        0.53
Jupiter    11.20
Saturn      9.45
Uranus      4.01
Neptune     3.88
```

Interestingly, we can use slicing on the row index names (but not on the column names, as noted above).

Alternatively, we could have specified that the `planet` column be used as the index when we read the tabular data from the text data file:

```
In[9]: planets = pd.read_table('planet_data.txt', sep='\s+',
  ...:                          index_col='planet')
```

This yields the same result that we obtained when we read in the data file without designating an index column but then later set the index column with the Pandas `set_index` method (see `In[6]` above).

11.1.2.1 Creating DataFrames from Dictionaries

In the previous section, we created a DataFrame by reading tabular data from a text file. Alternatively, you can create a DataFrame using the Pandas `DataFrame` routine. As input, you can use nearly any list-like object, including a list, a NumPy array, or a dictionary. Perhaps the simplest way is using a dictionary.

```
In[10]: optmat = {'mat': ['silica', 'titania', 'PMMA', 'PS'],
   ...:           'RI': [1.46, 2.40, 1.49, 1.59],
   ...:           'density': [2.03, 4.2, 1.19, 1.05]}

In[11]: omdf = pd.DataFrame(optmat)
Out[11]:
mat     RI  density
0   silica  1.46     2.03
1  titania  2.40     4.20
2     PMMA  1.49     1.19
3       PS  1.59     1.05
```

You can coerce the columns to appear in any desired order using the `columns` keyword argument.

```
In[12]: omdf = pd.DataFrame(optmat,
   ...:         columns=['mat', 'density', 'RI'])

In[13]: omdf
```

```
Out[13]:
mat  density    RI
0    silica     2.03  1.46
1   titania     4.20  2.40
2      PMMA     1.19  1.49
3        PS     1.05  1.59
```

You can also create a DataFrame with empty columns and fill in the data later.

```
In[14]: omdf1 = pd.DataFrame(index=['silica', 'titania',
   ...:            'PMMA', 'PS'], columns=('density', 'RI'))
In[15]: omdf1
Out[15]:
density   RI
silica      NaN  NaN
titania     NaN  NaN
PMMA        NaN  NaN
PS          NaN  NaN
```

The index and column names are indicated, but there is no data. The empty data columns are indicated by `NaN` (not-a-number). We can fill in the empty entries as follows:

```
In[16]: omdf1.loc['PS', ('RI', 'density')] = (1.05, 1.59)

In[17]: omdf1
Out[17]:
density    RI
silica      NaN   NaN
titania     NaN   NaN
PMMA        NaN   NaN
PS         1.59  1.05
```

Here, we used the `loc` method, which we discuss in greater detail in the next section.

Let's check the data types in our DataFrame.

```
In[18]: omdf1.dtypes
Out[18]:
density    object
RI         object
dtype: object
```

The data types for the `index` and `density` columns were set to be `object` when the DataFrame was created because we gave these columns no data. Now that we have entered the data, we would prefer that the `index` and `density` columns be the `float` data type. To do so, we explicitly set the data type.

```
In[19]: omdf1[['RI', 'density']] = omdf1[
   ...:        ['RI', 'density']].apply(pd.to_numeric)

 In[20]: omdf1.dtypes
Out[20]:
density     float64
RI          float64
dtype: object
```

The columns are correctly typed as floating point numbers while the overall DataFrame is an object.

11.2 INDEXING DataFrames

In the previous section, we introduced a method for selecting an entire column of a data frame. We also described methods for selecting particular entries in DataFrames and slices over DataFrame rows. This latter set of indexing methods is limited (no slicing over columns) and inefficient.

In this section, we introduce two Pandas properties, `iloc` and `loc`, that provide both powerful and efficient indexing over data. We will demonstrate how these two properties work using the `planets` DataFrame we considered in the previous section

```
planets
Out[20]:
distance      mass  gravity  diameter    year
planet
Mercury      0.39    0.0550    0.380     0.38    0.24
Venus        0.72    0.8200    0.910     0.95    0.62
Earth        1.00    1.0000    1.000     1.00    1.00
Mars         1.52    0.1100    0.380     0.53    1.88
Jupiter      5.20  318.0000    2.360    11.20   11.90
Saturn       9.58   95.0000    0.920     9.45   29.00
Uranus      19.20   15.0000    0.890     4.01   84.00
Neptune     30.00   17.0000    1.120     3.88  164.00
Pluto       39.50    0.0024    0.071     0.19  248.00
```

11.2.1 Pandas `iloc` Indexing

The Pandas `iloc` property indexes DataFrames by row and column *number* (note the order: [row, column]—it's opposite to what we used before):

```
 In[1]: planets.iloc[0, 1]
Out[1]: 0.055
```

The row and column numbers use the usual Python zero-based indexing scheme. Note that `Mercury` is row 0 and `mass` in column 1; the index column doesn't figure in the `iloc` indexing scheme. The usual slicing syntax applies:

```
In[2]: planets.iloc[3:5, 1:4]
Out[2]:
mass  gravity  diameter
planet
Mars        0.11     0.38      0.53
Jupiter   318.00     2.36     11.20
```

As with lists and NumPy arrays, an index of −1 signifies the last element, −2 is the next to the last element, and so on.

11.2.2 Pandas `loc` Indexing

The Pandas `loc` property is an extremely versatile and powerful tool for indexing DataFrames. Amongst other things, it allows you to use the row and column names as indices (row first, then column, as for `iloc`).

```
In[3]: planets.loc["Mars":"Jupiter", "mass":"diameter"]
Out[3]:
mass  gravity  diameter
planet
Mars        0.11     0.38      0.53
Jupiter   318.00     2.36     11.20
```

At first look, it seems simply like an alternative syntax for accomplishing the same thing you might have done with `iloc` indexing. But it can do much more. Suppose, for example, we wanted a list of planets whose mass was larger than the Earth's. As a part of its functionality, `loc` indexing can select data based on conditions.

```
In[4]: planets.loc[planets.mass > 1.0]
Out[4]:
distance      mass   gravity  diameter   year
planet
Jupiter       5.20   318.0      2.36     11.20   11.9
Saturn        9.58    95.0      0.92      9.45   29.0
Uranus       19.20    15.0      0.89      4.01   84.0
Neptune      30.00    17.0      1.12      3.88  164.0
```

Here, we set `loc` to select data based on a *condition* on the value of the planet's mass, whether it is greater than the mass of the Earth, which is 1. This is essentially the same *Boolean indexing* we encountered in Section 4.4.4 for NumPy arrays.

You can specify quite complicated conditions if you wish. For example, suppose we wanted a list of all the planets that are more massive than the Earth but have a smaller gravitational force at their surface. We could list those planets as follows:

```
 In[5]: planets.loc[(planets["mass"] > 1.0) &
   ...:                (planets["gravity"] < 1.0)]
Out[5]:
distance  mass  gravity  diameter  year
planet
Saturn      9.58  95.0     0.92       9.45  29.0
Uranus     19.20  15.0     0.89       4.01  84.0
```

The parentheses within `loc` are needed to define the order in which the logical operations are applied.

If we don't want to see all of the columns, we can specify the columns as well as the rows:

```
 In[6]: planets.loc[(planets.mass > 1.0) &
   ...:                (planets.gravity < 1.0),
   ...:                'mass':'gravity']

Out[6]:
mass  gravity
planet
Saturn  95.0     0.92
Uranus  15.0     0.89
```

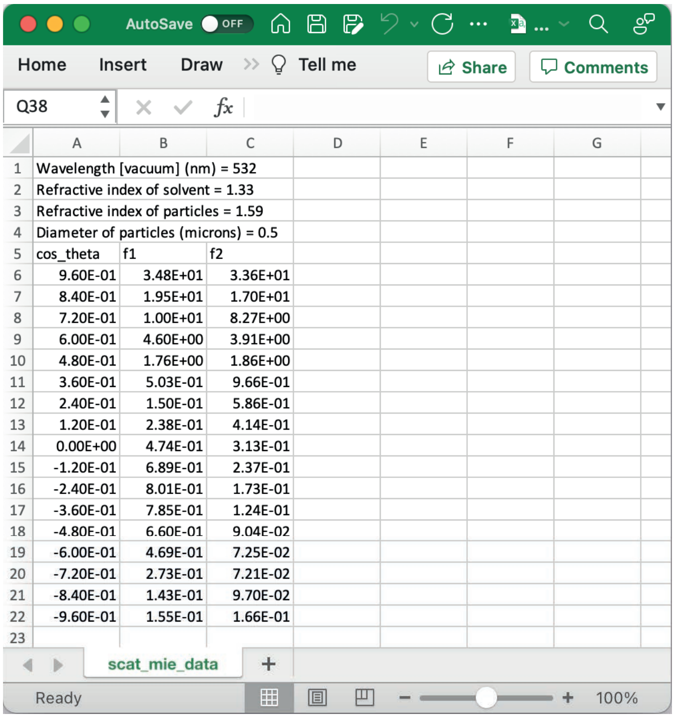
Note that we can use either `planets.mass` and `planets.gravity` or `planets["mass"]` and `planets["gravity"]`; either syntax works. Either works.

11.3 READING DATA FROM FILES USING Pandas

Pandas can read data from files written in many different formats, including the following: text, csv, Excel, JSON (JavaScript Object Notation), fixed-width text tables, HTML (web pages), and more that you can define. Our purpose here, however, is not to exhaust all the possibilities. Instead, we show you some of the more common methods and a few tricks. The idea is to illustrate, with some well-chosen examples (we hope!), how you can use Pandas, so that you get the idea of how Pandas works. After finishing this chapter, you should be able to use Pandas to read in and manipulate data, and then also be able to read the appropriate online Pandas documentation to extend your knowledge and adapt Pandas for your own applications.

11.3.1 Reading from Excel Files Saved as CSV Files

Excel files are commonly used to store data. As you learned in Chapter 5, one simple way to read in data from an Excel file is to save it as a CSV file, which is a text file of tabular data with different columns of data separated by commas (hence the name csv: comma-separated values, see Section 5.3.2).



	A	B	C
1	Wavelength [vacuum] (nm) = 532		
2	Refractive index of solvent = 1.33		
3	Refractive index of particles = 1.59		
4	Diameter of particles (microns) = 0.5		
5	cos_theta	f1	f2
6	9.60E-01	3.48E+01	3.36E+01
7	8.40E-01	1.95E+01	1.70E+01
8	7.20E-01	1.00E+01	8.27E+00
9	6.00E-01	4.60E+00	3.91E+00
10	4.80E-01	1.76E+00	1.86E+00
11	3.60E-01	5.03E-01	9.66E-01
12	2.40E-01	1.50E-01	5.86E-01
13	1.20E-01	2.38E-01	4.14E-01
14	0.00E+00	4.74E-01	3.13E-01
15	-1.20E-01	6.89E-01	2.37E-01
16	-2.40E-01	8.01E-01	1.73E-01
17	-3.60E-01	7.85E-01	1.24E-01
18	-4.80E-01	6.60E-01	9.04E-02
19	-6.00E-01	4.69E-01	7.25E-02
20	-7.20E-01	2.73E-01	7.21E-02
21	-8.40E-01	1.43E-01	9.70E-02
22	-9.60E-01	1.55E-01	1.66E-01
23			

Figure 11.1 Excel spreadsheet.

Let's start with the Excel file shown in Figure 11.1. The Excel application can save the spreadsheet as a csv text file. It looks like this:

Data: scat_mie_data.csv

```
Wavelength [vacuum] (nm) = 532,,
Refractive index of solvent = 1.33,,
Refractive index of particles = 1.59,,
Diameter of particles (microns) = 0.5,,
cos_theta,f1,f2
9.60E-01,3.48E+01,3.36E+01
8.40E-01,1.95E+01,1.70E+01
7.20E-01,1.00E+01,8.27E+00
6.00E-01,4.60E+00,3.91E+00
4.80E-01,1.76E+00,1.86E+00
3.60E-01,5.03E-01,9.66E-01
2.40E-01,1.50E-01,5.86E-01
1.20E-01,2.38E-01,4.14E-01
0.00E+00,4.74E-01,3.13E-01
-1.20E-01,6.89E-01,2.37E-01
-2.40E-01,8.01E-01,1.73E-01
-3.60E-01,7.85E-01,1.24E-01
-4.80E-01,6.60E-01,9.04E-02
-6.00E-01,4.69E-01,7.25E-02
```

```
-7.20E-01,2.73E-01,7.21E-02
-8.40E-01,1.43E-01,9.70E-02
-9.60E-01,1.55E-01,1.66E-01
```

This particular file has a header that provides information about the data, a header row specifying the name of each column of data, `cos_theta`,[1] `f1`, and `f2`, followed by three columns of data.

Let's use Pandas to read the data in this file. To start, we skip the header information contained in the top 4 lines of the file using the `skiprows` keyword argument:

```
In[1]: scat = pd.read_csv('scat_mie_data.csv', skiprows=4)
```

Note that we did not have `pd.read_csv` skip the row containing the column labels, `cos_theta`, `f1`, and `f2`, as Pandas will use those as labels for the different columns of data.

The Pandas function `pd.read_csv()` reads the data into a DataFrame, to which we give the name `scat` in the code above. We can examine the DataFrame by typing `scat` at the IPython prompt:

```
In[2]: scat
Out[2]:
cos_theta        f1        f2
0        1.000   70.0000   70.0000
1        0.875   27.1000   23.5000
2        0.750    8.5800    6.8000
3        0.625    1.8700    1.7200
4        0.500    0.2250    0.5210
5        0.375    0.3040    0.3110
6        0.250    0.6540    0.2360
7        0.125    0.7980    0.1490
8        0.000    0.7040    0.0763
9       -0.125    0.4850    0.0406
10      -0.250    0.2650    0.0364
11      -0.375    0.1170    0.0459
12      -0.500    0.0623    0.0579
13      -0.625    0.0851    0.0763
14      -0.750    0.1560    0.1200
15      -0.875    0.2590    0.2180
16      -1.000    0.4100    0.4100
```

A DataFrame is a tabular data structure similar to a spreadsheet.

```
In[3]: type(scat)
Out[3]: pandas.core.frame.DataFrame
```

[1]Generally, it's preferable to use column names with no spaces, which is why we have used an underline here. Pandas can handle headers with spaces, although in some cases, it can be limiting.

DataFrames consist of an index column and two or more data columns; the DataFrame `scat` has three data columns that were read in from the `scat_mie_data.csv` data file. The index column is added by Pandas and runs from 0 to $N - 1$, where N is the number of data points in the file. The three data columns are labeled with the names given in the fifth line of the data file, which was the first line read by `pd.read_csv()`, as the keyword argument `skiprows` was set equal to 4. By default, Pandas assumes that the first line read gives the names of the data columns that follow.

The data in the DataFrame can be accessed and sliced in different ways. To access the data in the column, you use the column labels:

```
 In[4]: scat.cos_theta
Out[4]:
0      1.000
1      0.875
2      0.750
3      0.625
4      0.500
5      0.375
6      0.250
7      0.125
8      0.000
9     -0.125
10    -0.250
11    -0.375
12    -0.500
13    -0.625
14    -0.750
15    -0.875
16    -1.000
Name: cos_theta, dtype: float64
```

A single row of a DataFrame is an example of the other central data structure of Pandas: a Series. We discuss Series and DataFrames more systematically in Section 11.1.

```
 In[5]: type(scat.cos_theta)
Out[5]: pandas.core.series.Series
```

Typing `scat['cos_theta']`, a syntax similar to the one used for dictionaries, yields the same result. Individual elements and slices can be accessed by indexing as for NumPy arrays:

```
 In[6]: scat.cos_theta[2]
Out[6]: 0.75

 In[7]: scat.cos_theta[2:5]
Out[7]:
```

```
2     0.750
3     0.625
4     0.500
Name: cos_theta, dtype: float64

 In[8]: scat['cos_theta'][2:5]
Out[8]:
2     0.750
3     0.625
4     0.500
Name: cos_theta, dtype: float64
```

Similarly, `scat.f1` and `scat.f2` give the data in the columns labeled `f1` and `f2`.

11.3.1.1 *Munging: Cleaning up Messy Data*

In the example above, we ignored the header data in the first four lines by setting `skiprows=4`. But suppose we want to read in the information in those four rows. How would we do it? Let's try the routine `read_csv()` once again.

```
 In[1]: head = pd.read_csv('scat_mie_data.csv', nrows=4,
   ...: header=None)
```

We use the keyword `nrows` and set it equal to 4 so that Pandas reads only the first four lines of the file, which comprise the header information. We also set `head=None` as there is no separate header information for these four rows. Let's examine the result by typing `head`, the name we assigned to these data.

```
 In[2]: head
Out[2]:
0    1    2
0          Wavelength [vacuum] (nm) = 532 NaN  NaN
1      Refractive index of solvent = 1.33 NaN  NaN
2    Refractive index of particles = 1.59 NaN  NaN
3   Diameter of particles (microns) = 0.5 NaN  NaN
```

The four rows are indexed from 0 to 3, as expected. The three columns are indexed from 0 to 2. Pandas introduced the column indices 0 to 2 because we set `header=None` in the `read_csv()` calling function instead of inferring names for these columns from the first row read from the CSV file as it did above. Individual elements of the `head` DataFrame can be indexed by their column and row, respectively:

```
 In[3]: head[0][1]
Out[3]: 'Refractive index of solvent = 1.33'
```

The quotes tell us that the output of `head[0][1]` is a string. In general, Pandas infers the data types for the different columns and assigns them correctly for numeric, string, or Boolean data types. But in this case, the information in the first column of the spreadsheet contains both string and numeric data, so `read_csv()` interprets the column as a string.

The data in the other two columns, which were empty in the original Excel spreadsheet and are commas with nothing between them in the CSV file, become `NaN` ("not a number") in the DataFrame. Accessing the datum in one cell as before gives the expected result:

```
In[4]: head[1][1]
Out[4]: nan
```

Pandas fills in missing data with `NaN`, a feature we discuss in greater detail in Section 11.1.2.

The appearance of quotes in the output `Out[3]:` above indicates that the data read from the header information are stored as strings. But suppose we want to separate out the numeric data from the strings that describe them. While Python has routines for stripping off numeric information from strings, performing this task is more efficient when reading the file. To do this, we use the Pandas routine `read_table()`, which reads in data from a text file like `read_csv()`. With `read_table()`, however, the user can specify the symbol that will be used to separate the columns of data: a symbol other than a comma can be used. The following call to `read_table()` does just that.

```
In[5]: head = pd.read_table('scat_mie_data.csv', sep='=',
  ...: nrows=4, header=None)
```

The keyword `sep`, which specifies the symbol that separates columns, is set equal to the string `'='`, as the equals sign delimits the string from the numeric data in this file. Printing out `head` reveals that there are now two columns.

```
In[6]:  head
Out[6]:
0         1
0          Wavelength [vacuum] (nm)     532,,
1       Refractive index of solvent    1.33,,
2     Refractive index of particles    1.59,,
3   Diameter of particles (microns)     0.5,,
```

This still isn't quite what we want, as the second column consists of numbers followed by two commas, unwanted remnants of the CSV file. We eliminate the commas by declaring the comma to be a "comment" character (the symbol `#` is the default comment character in Python). We do this by introducing the keyword `comment`, as illustrated here:

```
In[7]:  head = pd.read_table('scat_mie_data.csv', sep='=',
nrows=4, comment=',',
header=None)
```

Now typing `head` gives numbers without the trailing commas:

```
 In[8]:   head
Out[8]:
0       1
0          Wavelength [vacuum] (nm)   532.00
1       Refractive index of solvent     1.33
2     Refractive index of particles     1.59
3   Diameter of particles (microns)     0.50
```

Printing out individual elements of the two columns shows that the elements of column 0 are strings while the elements of column 1 are floating point numbers, which is the desired result.

```
 In[9]:  head[0][0]
Out[9]: 'Wavelength [vacuum] (nm) '

 In[10]:  head[1][:]
Out[10]:
0    532.00
1      1.33
2      1.59
3      0.50
Name: 1, dtype: float64
```

11.3.1.2 *Naming Columns Manually*

If you prefer for the columns to be labeled by descriptive names instead of numbers, you can use the keyword `names` to provide names for the columns.

```
 In[11]:  head = pd.read_table('scat_mie_data.csv', sep='=',
   ...:                        nrows=4, comment=',',
   ...:                        names=['property', 'value'])

Out[11]: head
property   value
0          Wavelength [vacuum] (nm)   532.00
1       Refractive index of solvent     1.33
2     Refractive index of particles     1.59
3   Diameter of particles (microns)     0.50

 In[12]: head['property'][2]
Out[12]: 'Refractive index of particles '

 In[13]: head['value'][2]
Out[13]: 1.5900000000000001
```

We can use what we have learned here to read data from the data file and then plot it, as shown in Figure 11.2. Here is the code that produces the plot shown in Figure 11.2.

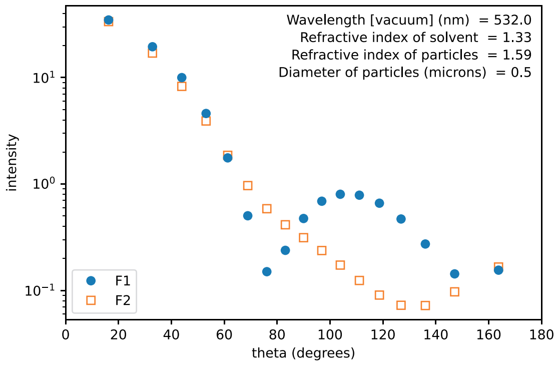


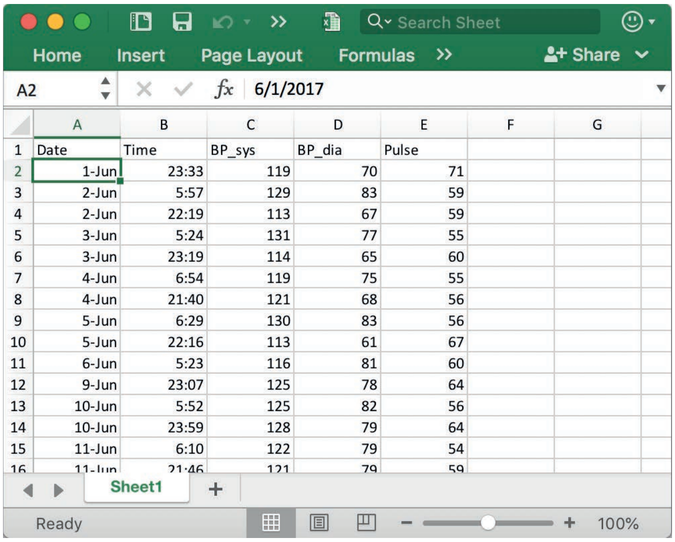
Figure 11.2 Plotting data from CSV file read by Pandas routines.

Code: scat_mie_plot.py

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Read in data
head = pd.read_table("scat_mie_data.csv", sep="=", nrows=4,
                     comment=",", header=None)
scat = pd.read_csv("scat_mie_data.csv", skiprows=4)

theta = (180. / np.pi) * np.arccos(scat.cos_theta)

fig, ax = plt.subplots(figsize=(6, 4))

ax.semilogy(theta, scat.f1, "o", color="C0", label="F1")
ax.semilogy(theta, scat.f2, "s", mec="C1", mfc="white", zorder=-1,
            label="F2")
ax.set_xlim(0., 180.)
ax.legend(loc="lower left")
ax.set_xlabel("theta (degrees)")
ax.set_ylabel("intensity")
for i in range(4):
    ax.text(0.98, 0.94-i/18, f"{head[0][i]} = {head[1][i]}",
            fontsize=10, ha="right", transform=ax.transAxes)
fig.tight_layout()
fig.savefig("figures/scat_mie_plot.pdf")
fig.show()
```

11.3.2 Reading from an Excel File

Pandas can also read directly from Excel files (*i.e.*, with .xls or .xlsx extensions). Let's consider an Excel file containing blood pressure and pulse data taken twice per day, early in the morning and late in the evening, over several weeks. The top of the Excel file is shown in Figure 11.3 (there are many more rows,



	A	B	C	D	E	F	G
1	Date	Time	BP_sys	BP_dia	Pulse		
2	1-Jun	23:33	119	70	71		
3	2-Jun	5:57	129	83	59		
4	2-Jun	22:19	113	67	59		
5	3-Jun	5:24	131	77	55		
6	3-Jun	23:19	114	65	60		
7	4-Jun	6:54	119	75	55		
8	4-Jun	21:40	121	68	56		
9	5-Jun	6:29	130	83	56		
10	5-Jun	22:16	113	61	67		
11	6-Jun	5:23	116	81	60		
12	9-Jun	23:07	125	78	64		
13	10-Jun	5:52	125	82	56		
14	10-Jun	23:59	128	79	64		
15	11-Jun	6:10	122	79	54		
16	11-Jun	21:46	121	79	59		

Figure 11.3 Excel file containing blood pressure data.

which are not shown). The file contains five columns: the date, time, systolic blood pressure, diastolic blood pressure, and pulse. The blood pressures are reported in mm-Hg, and the pulse rate is in heartbeats/minute. The name of the Excel file is `BloodPressure.xlsx`.

Reading data from an Excel file using Pandas is simple:

```
In[1]: bp = pd.read_excel('blood_pressure.xlsx',
  ...: usecols='A:E')
```

The keyword argument `usecols='A:E'` tells Pandas to read in only columns A through E; data in any other columns are ignored. Had we wanted to read in only the pulse and not the blood pressure data, we could have written `usecols='A:B, E'` for the keyword argument. But as written, Pandas reads columns A through E into a DataFrame object named `bp`, whose structure we can see by typing `bp`:

```
In[2]: bp
Out[2]:
Date        Time   BP_sys  BP_dia  Pulse
0   2017-06-01   23:33:00     119       70      71
1   2017-06-02   05:57:00     129       83      59
2   2017-06-02   22:19:00     113       67      59
3   2017-06-03   05:24:00     131       77      55
4   2017-06-03   23:19:00     114       65      60
5   2017-06-04   06:54:00     119       75      55
6   2017-06-04   21:40:00     121       68      56
```

TABLE 11.1 Summary of Pandas functions to read tabular data files.

Function	Description
read_table	workhorse: read tabular data from a text file
read_csv	read tabular data from a comma separated file
read_excel	read tabular data from an Excel file
read_clipboard	read data copied from web page to clipboard
read_fwf	read data in fixed-width columns w/o delimiters

```
7   2017-06-05   06:29:00      130       83      56
8   2017-06-05   22:16:00      113       61      67
9   2017-06-06   05:23:00      116       81      60
10  2017-06-09   23:07:00      125       78      64
.
.
.
```

Table 11.1 summarizes several Pandas functions for reading data files, including the three discussed above.

11.3.3 Getting Data from the Web

Pandas has extensive tools for scraping data from the web. Here, we illustrate one of the simpler cases, reading a CSV file from a website. The Bank of Canada publishes the daily exchange rates between the Canadian dollar and a couple dozen international currencies. We want to download these data and print the results as a simple table. Here we employ Pandas's usual `read_csv` function using its `url` keyword argument to specify the web address of the CSV file we want to read. To follow the code, download the CSV file manually using the URL defined in line 6 of the program `url_read.py` listed below and then open it using a spreadsheet program like Excel.

To obtain all the data we want, we read the CSV file twice. In line 9 of `url_read.py`, we call `read_csv` to read into a DataFrame `rates` the exchange rates for the different currencies over a range of dates that extends from any start date after 2017-01-03, the earliest date for which the site supplies data, up to the most recent business day. The rates are indexed by date (*e.g.*, `'2018-04-23'`) with each column corresponding to a different currency.

The header for the exchange rates, which consists of codes for each exchange rate, begins on line 40 of the CSV file, so we skip the first 39 rows. In line 13 of `url_read.py`, we extract from the shape of the DataFrame, the number of days and the number of currencies downloaded.

We reread the CSV file on lines 16–17 of `url_read.py` to get keys for the codes for the various currencies used in the DataFrame. We use the number of currencies determined in line 13 to determine the number of lines to read. Lines 18–19 strip off some extraneous verbiage in the keys.

Code: url_read.py

```
 1  import pandas as pd
 2  
 3  url1 = "http://www.bankofcanada.ca/"
 4  url2 = "valet/observations/group/FX_RATES_DAILY/csv?start_date="
 5  start_date = "2017-01-03"  # Earliest start date is 2017-01-03
 6  url = url1 + url2 + start_date  # Complete url to download csv file
 7  
 8  # Read in rates for different currencies for a range of dates
 9  rates = pd.read_csv(url, skiprows=39, index_col="date")
10  rates.index = pd.to_datetime(rates.index)  # assures data type
11  
12  # Get number of days & number of currences from shape of rates
13  days, currencies = rates.shape
14  
15  # Read in the currency codes & strip off extraneous part
16  codes = pd.read_csv(url, skiprows=10, usecols=[0, 2],
17                      nrows=currencies)
18  for i in range(currencies):
19      codes.iloc[i, 1] = codes.iloc[i, 1].split(" to Canadian")[0]
20  
21  # Report exchange rates for the most most recent date available
22  date = rates.index[-1]  # most recent date available
23  print("\nCurrency values on {0}".format(date))
24  for (code, rate) in zip(codes.iloc[:, 1], rates.loc[date]):
25      print(f"{code:20s}  Can$ {rate:8.6g}")
```

Using the `index` attribute for Pandas DataFrames, line 18 sets the date for which the currency exchange data will be displayed, in this case, the most recent date in the file. Running the program produces the desired output:

```
In[3]: run urlRead.py
Currency values on 2024-01-12 00:00:00
Australian dollar     Can$    0.8965
Brazilian real        Can$    0.2758
Chinese renminbi      Can$    0.1868
European euro         Can$    1.4673
Hong Kong dollar      Can$    0.1712
Indian rupee          Can$   0.01616
Indonesian rupiah     Can$   8.6e-05
Japanese yen          Can$   0.00924
Malaysian ringgit     Can$       nan
Mexican peso          Can$   0.07939
New Zealand dollar    Can$    0.8369
Norwegian krone       Can$    0.1302
Peruvian new sol      Can$    0.3622
Russian ruble         Can$   0.01514
```

```
Saudi riyal             Can$     0.357
Singapore dollar        Can$    1.0056
South African rand      Can$   0.07186
South Korean won        Can$  0.001021
Swedish krona           Can$    0.1303
Swiss franc             Can$     1.571
Taiwanese dollar        Can$   0.04306
Thai baht               Can$       nan
Turkish lira            Can$    0.0445
UK pound sterling       Can$    1.7071
US dollar               Can$    1.3387
Vietnamese dong         Can$       nan
```

What we have done here illustrates only one simple feature of Pandas for scraping data from the web. Many more web-scraping tools exist within Pandas. They are extensive and powerful and can be used with other packages, such as `urllib3`, to extract almost any data on the web.

11.4 EXTRACTING INFORMATION FROM A DataFrame

Once we have our data organized in a DataFrame, we can employ the tools of Pandas to examine and process the data it contains in various ways. Let's start with the `planets` DataFrame introduced in Section 11.1.2. We read it in again for good measure:

```
In[1]: planets = pd.read_table('planet_data.txt', sep='\s+',
  ...:                         index_col='planet')

In[2]: planets
Out[2]:
distance       mass   gravity  diameter    year
planet
Mercury       0.39    0.0550    0.380      0.38     0.24
Venus         0.72    0.8200    0.910      0.95     0.62
Earth         1.00    1.0000    1.000      1.00     1.00
Mars          1.52    0.1100    0.380      0.53     1.88
Jupiter       5.20  318.0000    2.360     11.20    11.90
Saturn        9.58   95.0000    0.920      9.45    29.00
Uranus       19.20   15.0000    0.890      4.01    84.00
Neptune      30.00   17.0000    1.120      3.88   164.00
Pluto        39.50    0.0024    0.071      0.19   248.00
```

Note that we have set the `planet` column as the index variable in the `planets` DataFrame.

Pandas can readily sort data. For example, to list the planets in order of increasing mass, we write:

```
In[3]: planets.sort_values(by='mass')
```

```
Out[3]:
distance       mass   gravity  diameter    year
planet
Pluto         39.50    0.0024    0.071      0.19   248.00
Mercury        0.39    0.0550    0.380      0.38     0.24
Mars           1.52    0.1100    0.380      0.53     1.88
Venus          0.72    0.8200    0.910      0.95     0.62
Earth          1.00    1.0000    1.000      1.00     1.00
Uranus        19.20   15.0000    0.890      4.01    84.00
Neptune       30.00   17.0000    1.120      3.88   164.00
Saturn         9.58   95.0000    0.920      9.45    29.00
Jupiter        5.20  318.0000    2.360     11.20    11.90
```

To produce the same table ordered from highest to lowest mass, use the keyword argument `ascending=False`.

We can use Boolean indexing to get a list of all the planets with gravitational acceleration larger than Earth's.

```
In[4]: planets[planets['gravity']>1]
Out[4]:
distance    mass  gravity  diameter    year
planet
Jupiter        5.2   318.0      2.36     11.20   11.9
Neptune       30.0    17.0      1.12      3.88  164.0
```

It's instructive to parse `In [4]` to understand better how the Boolean indexing works. Suppose we had typed just what is inside the outermost brackets:

```
In[5]: planets['gravity']>1
Out[5]:
planet
Mercury     False
Venus       False
Earth       False
Mars        False
Jupiter      True
Saturn      False
Uranus      False
Neptune      True
Pluto       False
Name: gravity, dtype: bool
```

We get the logical (Boolean) truth values for each entry. Thus, writing `planets[planets['gravity']>1]` lists the DataFrame only for those entries where the Boolean value is `True`.

Suppose we would like to find the volume V of each of the planets (normalized by the volume of the Earth) and add the result to our `planets` DataFrame. Using the formula $V = \frac{1}{6}\pi d^3 / \frac{1}{6}\pi d_{\text{Earth}}^3 = d^3$, where d is the diameter of the planet and $d_{\text{Earth}} = 1$, we simply write

```
 In[6]: planets['volume'] = planets['diameter']**3

 In[7]: planets
Out[7]:
distance      mass  gravity diameter    year     volume
planet
Mercury     0.39    0.0550    0.380    0.38    0.24     0.0549
Venus       0.72    0.8200    0.910    0.95    0.62     0.8574
Earth       1.00    1.0000    1.000    1.00    1.00     1.0000
Mars        1.52    0.1100    0.380    0.53    1.88     0.1489
Jupiter     5.20  318.0000    2.360   11.20   11.90  1404.9280
Saturn      9.58   95.0000    0.920    9.45   29.00   843.9086
Uranus     19.20   15.0000    0.890    4.01   84.00    64.4812
Neptune    30.00   17.0000    1.120    3.88  164.00    58.4111
Pluto      39.50    0.0024    0.071    0.19  248.00     0.0069
```

This maneuver added an extra column to the DataFrame with the desired data.

Next, we revisit the blood pressure DataFrame introduced in Section 11.3.2 to explore some other features of DataFrames.

```
 In[8]: bp = pd.read_excel('blood_pressure.xlsx',
   ...:                     usecols='A:E',
   ...:                     parse_dates=[['Date', 'Time']])

 In[9]: bp = bp.set_index('Date_Time')
```

Here, we used the keyword argument `parse_dates`, which combines the `Date` and `Time` columns into a single datetime column. We then set the `Date_Time` column as the index variable in the `bp` DataFrame.

```
 In[10]: bp.head()
Out[10]9:
BP_sys  BP_dia  Pulse
Date_Time
2017-06-01 23:33:00     119      70     71
2017-06-02 05:57:00     129      83     59
2017-06-02 22:19:00     113      67     59
2017-06-03 05:24:00     131      77     55
2017-06-03 23:19:00     114      65     60
```

Pandas can calculate standard statistical quantities for the data in a DataFrame.

```
 In[11]: bp['BP_sys'].mean()       # average systolic pressure
Out[11]: 119.27083333333333

 In[12]: bp['BP_sys'].max()        # maximum systolic pressure
Out[12]: 131

 In[13]: bp['BP_sys'].min()        # minimum systolic pressure
```

```
Out[13]: 105

 In[14]: bp['BP_sys'].count()      # num (non-null) of entries
Out[14]: 48
```

The statistical methods can even act on dates if doing so makes sense.

```
 In[15]: bp.index.min()            # starting datetime
Out[15]: Timestamp('2017-06-01 23:33:00')

 In[16]: bp.index.max()             # ending datetime
Out[16]: Timestamp('2017-07-17 06:22:00')
```

Note that we used `bp.index` and not `bp.['Date_Time']`, as we previously set `'Date_Time'` to be the index of `bp`. Time differences can also be calculated:

```
 In[17]: bp.index.max()-bp.index.min()
Out[17]: Timedelta('45 days 06:49:00')
```

We can combine these methods with conditional indexing to answer some interesting questions. For example, are there systematic differences in the blood pressure and pulse readings in the morning and the evening? Let's use what we've learned to find out. First, we separate the morning and evening readings into two different Series using Boolean indexing.

```
 In[18]: PulseAM = bp.loc[bp.index.hour<12, 'Pulse']

 In[19]: PulsePM = bp.loc[bp.index.hour>=12, 'Pulse']
```

Now let's look at some averages and fluctuations about the mean:

```
 In[20]: precision 3
Out[20]: '%.3f'

 In[21]: PulseAM.mean(), PulseAM.std(), PulseAM.sem()
Out[21]:: (57.586, 5.791, 1.075)

 In[22]: PulsePM.mean(), PulsePM.std(), PulsePM.sem()
Out[22]:: (61.789, 4.939, 1.133)
```

The average morning pulse of 57.6 is lower than the average evening pulse of 61.8. The difference of 4.2 is greater than the standard error of the mean of about 1.1, which means the difference is significant, even though the morning and evening pulse distributions overlap each other to a significant extent, as indicated by the standard deviations of around 5.

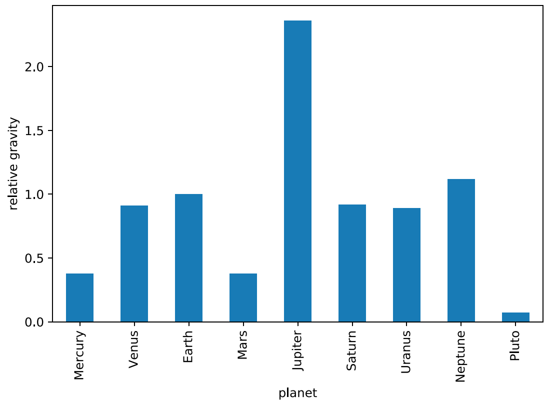


Figure 11.4 Relative gravity of different planets.

11.5 PLOTTING WITH Pandas

In Chapter 8, we introduced the Matplotlib plotting package, which provides an extensive framework for plotting within Python. Pandas builds on the Matplotlib package, adding some functionality peculiar to Pandas.

One notable change is that when plotting data from a Pandas Series or DataFrame, Matplotlib's `plot` function will use the index as the x data if the x data is not otherwise specified. For example, we can get a graphical display, shown in Figure 11.4, of the relative gravity of each planet from the `planets` DataFrame with the following simple commands:

```
 In[1]: planets['gravity'].plot.bar(color='C0')
Out[1]: <matplotlib.axes._subplots.AxesSubplot at 0x11de29400>

 In[2]: ylabel('relative gravity')
Out[2]: Text(42.5972,0.5,'relative gravity')

 In[3]: tight_layout()
```

Pandas allows us to write plotting commands differently, where a Matplotlib plotting function is now a DataFrame method. Here, we use `plot` as a method of `planets['gravity']`. We further specify a `bar` (histogram) plot with `bar` as a method `plot`. The y-axis is specified by choosing the desired column(s) of the DataFrame, in this case, `gravity`, and the x-axis is taken to be the DataFrame index unless otherwise specified. Notice how each bar is neatly labeled with its corresponding planet index. We could have made a plot with horizontal instead of vertical bars using the `barh` method instead of `bar`. Try it out!

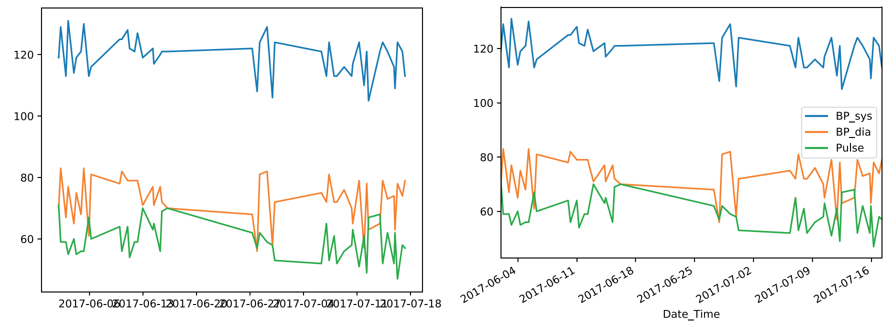


Figure 11.5 Crude plots of the bp DataFrame.

Let's look at another example, this time using our bp DataFrame. First, let's plot it using the conventional Matplotlib syntax,

```
In [4]: plot(bp)
Out[4]:
[<matplotlib.lines.Line2D at 0x13667b278>,
<matplotlib.lines.Line2D at 0x1366aecf8>,
<matplotlib.lines.Line2D at 0x1366b7080>]
```

which produces the graph on the left in Figure 11.5. The three traces correspond to the systolic pressure, the diastolic pressure, and the pulse, and are plotted as a function of the time (date), which is the index of the bp DataFrame. Since the x-array is not specified, the index variable, the date, is used. However, the dates are not so nicely formatted and run into each other.

Alternatively, we can graph this using plot as a DataFrame method:

```
In [5]: bp.plot()
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x1527ddd240>
```

The result, shown on the right in Figure 11.5, is a more nicely formatted plot, where the dates labeling the x-axis are automatically tilted so that they don't run into each other, and a legend is produced, which identifies the different traces.

Figure 11.6 shows these same data in a more compelling and refined graph, bringing together much of the analysis we have already developed using Pandas. Measurements made early in the morning and late in the evening are distinguished using open and closed symbols. The morning and evening averages are indicated by horizontal lines annotated with the numerical averages and indicated using arrows. A more complete legend is supplied.

The code shows how Pandas and conventional Matplotlib syntax can be used together. The blood pressure and pulse data are plotted on separate

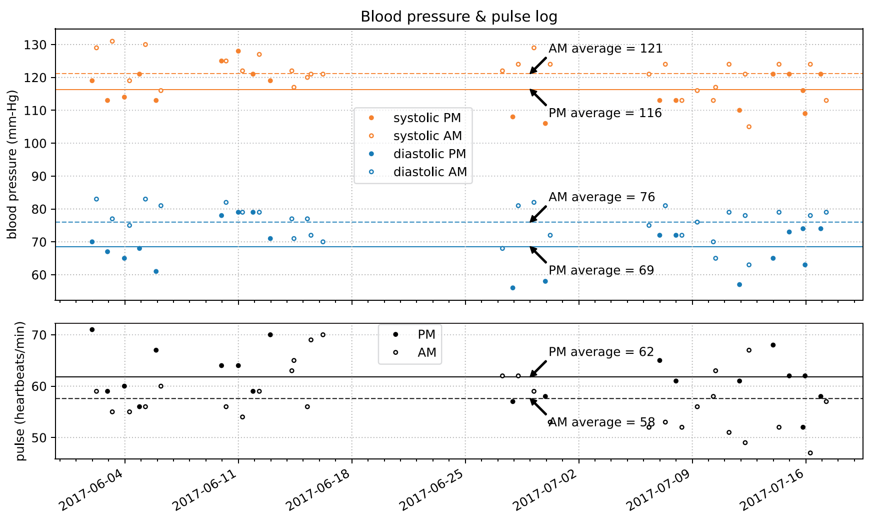


Figure 11.6 Blood pressure data from an Excel file.

graphs sharing a common time axis. The code that produces Figure 11.6 is listed below. Note that which plot is chosen, `ax1` or `ax2`, is indicated using the keyword argument `ax` within the `plot` method belonging to the various data sets, `sysPM`, ..., `PulsePM`. Finally, Matplotlib's dates package is used to format the x-axis.

Much more information is available at the Pandas website, which gives details about all of p's plotting commands.

Code: blood_pressure.py

```python
import matplotlib.pyplot as plt
import pandas as pd
import matplotlib.dates as mdates
from datetime import datetime

# Read in data
bp = pd.read_excel('blood_pressure.xlsx', usecols='A:E',
                   parse_dates=[['Date', 'Time']])
bp = bp.set_index('Date_Time')
# Divide data into AM and PM sets
diaAM = bp.loc[bp.index.hour < 12, 'BP_dia']
diaPM = bp.loc[bp.index.hour >= 12, 'BP_dia']
sysAM = bp.loc[bp.index.hour < 12, 'BP_sys']
sysPM = bp.loc[bp.index.hour >= 12, 'BP_sys']
PulseAM = bp.loc[bp.index.hour < 12, 'Pulse']
PulsePM = bp.loc[bp.index.hour >= 12, 'Pulse']
# Set up figure with 2 subplots and plot BP data

```

```
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True,
                              gridspec_kw={'height_ratios': [2, 1]},
                              figsize=(10, 6))
fig.subplots_adjust(left=0.065, right=0.99, hspace=0.06)
sysPM.plot(ax=ax1, marker='o', ms=3, lw=0, color='C1',
           label='systolic PM')
sysAM.plot(ax=ax1, marker='o', ms=3, lw=0, color='C1',
           mfc='white', label='systolic AM')
diaPM.plot(ax=ax1, marker='o', ms=3, lw=0, color='C0',
           label='diastolic PM')
diaAM.plot(ax=ax1, marker='o', ms=3, lw=0, color='C0',
           mfc='white', label='diastolic AM')
# Average values of blood pressures with arrows labeling them
dtlab = datetime(2017, 6, 29)
bpavgs = (sysAM.mean(), sysPM.mean(), diaAM.mean(),
          diaPM.mean())
ytext = ('bottom', 'top')
tavgs = ('AM average = {0:0.0f}'.format(bpavgs[0]),
         'PM average = {0:0.0f}'.format(bpavgs[1]),
         'AM average = {0:0.0f}'.format(bpavgs[2]),
         'PM average = {0:0.0f}'.format(bpavgs[3]))
aprops = dict(facecolor='black', width=1, headlength=5,
              headwidth=5)
for i, bpa in enumerate(bpavgs):
    ax1.annotate(tavgs[i], xy=(dtlab, bpa),
                 xytext=((15, (-1)**(i % 2)*15)),
                 textcoords='offset points',
                 arrowprops=aprops, ha='left',
                 va=ytext[i % 2])
# Lines indicating average blood pressures
ax1.axhline(y=sysPM.mean(), color='C1', lw=0.75, zorder=-1)
ax1.axhline(y=sysAM.mean(), color='C1', dashes=(5, 2),
            lw=0.75, zorder=-1)
ax1.axhline(y=diaPM.mean(), color='C0', lw=0.75, zorder=-1)
ax1.axhline(y=diaAM.mean(), color='C0', dashes=(5, 2),
            lw=0.75, zorder=-1)
# Formatting top graph
ax1.set_title('Blood pressure & pulse log')
ax1.set_ylabel('blood pressure (mm-Hg)')
ax1.legend(loc=(0.37, 0.43))
ax1.grid(dashes=(1, 2))
# Plot pulse
PulsePM.plot(ax=ax2, marker='o', ms=3, lw=0, color='k',
             label='PM')
PulseAM.plot(ax=ax2, marker='o', ms=3, lw=0, color='k',
             mfc='white', label='AM')
# Average values of pulse with arrows labeling them
Pulseavgs = (PulseAM.mean(), PulsePM.mean())
tavgs = ('AM average = {0:0.0f}'.format(Pulseavgs[0]),
         'PM average = {0:0.0f}'.format(Pulseavgs[1]))
for i, pulse in enumerate(Pulseavgs):
    ax2.annotate(tavgs[i], xy=(dtlab, pulse),
                 xytext=((15, -(-1)**(i)*15)),
                 textcoords='offset points',
                 arrowprops=aprops, ha='left',
                 va=ytext[-i-1])
```

```python

ax2.axhline(y=PulsePM.mean(), color='k', lw=0.75, zorder=-1)
ax2.axhline(y=PulseAM.mean(), color='k', dashes=(5, 2),
            lw=0.75, zorder=-1)
# Formatting bottom graph
week = mdates.WeekdayLocator(byweekday=mdates.SU)
day = mdates.DayLocator()
ax2.xaxis.set_major_locator(week)
ax2.xaxis.set_minor_locator(day)
ax2.set_xlabel('')
ax2.set_ylabel('pulse (heartbeats/min)')
ax2.legend(loc=(0.4, 0.7))
ax2.grid(dashes=(1, 2))

fig.tight_layout()
fig.show()
fig.savefig('./figures/blood_pressure.pdf')
```

11.6 GROUPING AND AGGREGATION

Pandas allows you to group data and analyze the subgroups in useful and powerful ways. The best way to understand what you can do is to work with an example. Here, we will work with a data set that lists all Newark Liberty International Airport (EWR) departures on a particular (stormy) day. The data is stored in a CSV file named `ewrFlights20180516.csv`, which we read into a DataFrame that we call `ewr`.

```
 In[1]: ewr = pd.read_csv('ewrFlights20180516.csv')

 In[2]: ewr.head()
Out[2]:
Destination                Airline    Flight Departure \
0  Baltimore (BWI)     Southwest Airlines  WN 8512  12:09 AM
1  Baltimore (BWI)     Mountain Air Cargo  C2 7304  12:10 AM
2      Paris (ORY)  Norwegian Air Shuttle  DY 7192  12:30 AM
3      Paris (ORY)   euroAtlantic Airways  YU 7192  12:30 AM
4   Rockford (RFD)                    UPS   5X 108  12:48 AM

Terminal   Status Arrival_time  A_day Scheduled  S_day
0       NaN   Landed          NaN    NaN        NaN    NaN
1       NaN  Unknown          NaN    NaN        NaN    NaN
2         B   Landed      1:48 PM    NaN   1:35 PM    NaN
3         B   Landed      1:48 PM    NaN   1:35 PM    NaN
4       NaN  Unknown          NaN    NaN        NaN    NaN
 In[3]: ewr.shape
Out[3]:  (1555, 10)
```

Notice the line continuation symbol \, which is used because the DataFrame is wider than the page width. There are 1555 flights listed and 10 column headings: Destination, Airline, Flight, Departure, Terminal, Status, Arrival_time, A_day, Scheduled, and S_day. We will explain the headings as we go.

Let's get familiar with the ewr DataFrame. You might wonder what the possibilities are for the status of a flight. You can find out and get some additional information using the value_counts() method.

```
 In[4]: ewr['Status'].value_counts()
Out[4]:
Landed - On-time       757
Landed - Delayed       720
Canceled                41
Landed                  18
En Route - Delayed      10
Unknown                  4
Scheduled - Delayed      2
En Route - On-time       1
En Route                 1
Diverted                 1
Name: Status, dtype: int64

 In[5]: ewr['Status'].value_counts().sum()
Out[5]: 1555
```

The value_counts() method is quite useful. It finds all the unique entries in a Series (or DataFrame column) and reports the number of times each entry appears. We also checked to confirm that the categories' counts summed to the total number of entries.

Newark Airport has three terminals: A, B, and C. Let's find out how many departures there were from each terminal.

```
 In[6]: ewr['Terminal'].value_counts()
Out[6]:
C    826
A    471
B    191
```

11.6.1 The groupby Method

Now, suppose we would like to know the status of each flight broken down by terminal. For this, we need a more sophisticated tool: groupby. Here is how it works:

```
 In[7]: ewr['Status'].groupby(ewr['Terminal']).value_counts()
Out[7]:
Terminal  Status
A         Landed - On-time       229
Landed - Delayed       218
```

```
Canceled                21
Landed                   3
B          Landed - On-time        104
Landed - Delayed        70
En Route - Delayed       6
Canceled                 4
Landed                   4
Scheduled - Delayed      2
En Route - On-time       1
C          Landed - Delayed        413
Landed - On-time       395
Canceled                14
En Route - Delayed       4
Name: Status, dtype: int64
```

In this case, we want to know the status of each flight, so `ewr['Status']` comes first in our command above. Next, we want the status broken down by terminal, so we add the method `groupby` with the argument `ewr['Terminal']`. Finally, we want to know how many flights fall into each category so we add the method `value_counts()`.

Alternatively, we could have written

```
In[8]: ewr_statterm = ewr['Status'].groupby(ewr['Terminal'])
```

which creates a `groupby` object that we can subsequently process. For example, we can get the total number of flights from each terminal:

```
In[9]: ewr_statterm.count()
Out[9]:
Terminal
Terminal
A    471
B    191
C    826
Name: Status, dtype: int64
```

Or we can write `ewr_statterm.value_counts()`, which gives the same output as above.

11.6.2 Iterating Over Groups

Sometimes, it is useful to iterate over groups to perform a calculation. For example, suppose that for each airline, we want to determine what fraction of the flights arriving at their destination arrived on time.

The information about on-time arrivals is contained in the `Status` column of the `ewr` DataFrame. It has, amongst other things, entries `Landed - On-time` and `Landed - Delayed`. We will want to use these entries to perform the calculation.

To do this, we use a `for` loop with the following construction:

```
for name, group in grouped:
```

where `grouped` is a `groupby` object, and `name` and `group` are the individual names and groups within the `groupby` object that are looped over.

To perform our calculation, we need to iterate over each airline, so our `groupby` object should group by `ewr['Airline']`. Before actually doing the calculations, however, we illustrate how the loop works for our `groupby` object with a little demonstration program. In this program, the loop doesn't do any calculations; it simply prints out the `name` and `group` for each iteration with the following code:

Code: ewr_groupby_elements.py

```python
import pandas as pd

ewr = pd.read_csv("ewrFlights20180516.csv")

for airln, grp in ewr.groupby(ewr["Airline"]):
    print("\nairln = {}: \ngrp:".format(airln))
    print(grp)
```

The output of this program is:

```
airln = ANA:
grp:
Destination Airline   Flight Departure Terminal
134   San Francisco (SFO)     ANA  NH 7007   7:00 AM        C
189     Los Angeles (LAX)     ANA  NH 7229   7:59 AM        C
303         Chicago (ORD)     ANA  NH 7469   8:59 AM        C
438           Tokyo (NRT)     ANA  NH 6453  11:00 AM        C
562         Chicago (ORD)     ANA  NH 7569   1:20 PM        C
1140    Los Angeles (LAX)     ANA  NH 7235   6:43 PM        C
1533      Sao Paulo (GRU)     ANA  NH 7214  10:05 PM        C

Status Arrival_time  A_day Scheduled  S_day
134   Landed - Delayed     11:13 AM    NaN  10:18 AM    NaN
189   Landed - On-time     10:57 AM    NaN  11:05 AM    NaN
303   Landed - On-time     10:39 AM    NaN  10:25 AM    NaN
438   Landed - On-time      1:20 PM    NaN   1:55 PM    NaN
562   Landed - Delayed      3:16 PM    NaN   2:44 PM    NaN
1140  Landed - Delayed      9:54 PM    NaN   9:41 PM    NaN
1533  Landed - Delayed     10:06 AM    1.0   8:50 AM    1.0

airln = AVIANCA:
grp:
Destination  Airline   Flight Departure Terminal
81         Dulles (IAD)  AVIANCA  AV 2135   6:05 AM        A
367        Dulles (IAD)  AVIANCA  AV 2233  10:00 AM        A
```

```
422         Miami (MIA)  AVIANCA  AV 2002  10:44 AM        C
805  San Salvador (SAL)  AVIANCA   AV 399   3:55 PM        B
890        Bogota (BOG)  AVIANCA  AV 2245   4:45 PM        C

Status Arrival_time  A_day Scheduled  S_day
81       Landed - On-time      7:17 AM    NaN   7:25 AM    NaN
367      Landed - On-time     11:10 AM    NaN  11:20 AM    NaN
422      Landed - On-time      1:30 PM    NaN   1:46 PM    NaN
805   Scheduled - Delayed          NaN    NaN   7:05 PM    NaN
890      Landed - Delayed     12:42 AM    1.0   9:35 PM    NaN
.
.
.
```

By examining this output, the form of the data structures being looped over should become clear to you.

Now, let's do our calculation. To keep things manageable, let's say we only care about those airlines that landed 12 or more flights. Grouping the data by airline, we perform the calculation using a `for` loop, accumulating the results about on-time and late flights in a list of lists, which we convert to a DataFrame at the end of the calculations.

```
In[10]: ot = []  # create an empty list to accumulate results

In[11]: for airln, grp in ewr.groupby(ewr['Airline']):
   ...:     ontime = grp.Status[grp.Status ==
                     'Landed - On-time'].count()
   ...:     delayd = grp.Status[grp.Status ==
                     'Landed - Delayed'].count()
   ...:     totl = ontime+delayd
   ...:     if totl >= 12:
   ...:         ot.append([airln, totl, ontime/totl])
```

The code output is a list called `ot`. We convert it to a DataFrame using the Pandas function `DataFrame.from_records`.

```
In[12]: t = pd.DataFrame.from_records(ot, columns=['Airline',
   ...:               'Flights Landed', 'On-time fraction'])
```

We choose to print out the results sorted by on-time fraction, from largest to smallest.

```
In[13]: t.sort_values(by='On-time fraction', ascending=False)
Out[13]:
Airline  Flights Landed  On-time fraction
0          Air Canada             129          0.472868
1            Air China              24          0.750000
2      Air New Zealand              34          0.617647
3      Alaska Airlines              20          0.500000
```

```
4     American Airlines              27           0.592593
5              Austrian              28           0.428571
6     Brussels Airlines              21           0.523810
7             CommutAir              47           0.531915
8         Copa Airlines              12           0.333333
9       Delta Air Lines              33           0.606061
10           ExpressJet              64           0.531250
11                FedEx              27           0.555556
12      JetBlue Airways              24           0.416667
13            Lufthansa             119           0.403361
14    Republic Airlines              66           0.606061
15                  SAS              94           0.414894
16                SWISS              20           0.450000
17   Southwest Airlines              18           0.500000
18         TAP Portugal              38           0.315789
19      United Airlines             417           0.529976
20      Virgin Atlantic              26           0.615385
```

11.6.3 Reformatting DataFrames

We often create DataFrames that need to be reformatted for processing. The range of reformatting issues one can run across is enormous, so we can't even hope to cover all the eventualities. But we can illustrate a few to give you a sense of how this works.

In this example, we would like to analyze on-time arrivals. To do so, we will need to work with the `Departure`, `Arrival_time`, and `Scheduled` columns. Let's take a look at them.

```
 In[14]: ewr[['Departure', 'Arrival_time', 'Scheduled']].head()
Out[14]:
Departure Arrival_time Scheduled
0   12:09 AM          NaN       NaN
1   12:10 AM          NaN       NaN
2   12:30 AM      1:48 PM   1:35 PM
3   12:30 AM      1:48 PM   1:35 PM
4   12:48 AM          NaN       NaN
```

Some of the times are missing, represented by `NaN` in a DataFrame, but these are generally not much of a concern as Pandas handles them in an orderly manner. More worrisome are the times, which do not contain a date. This can be a problem if a flight departs on one day but arrives the next day. The `ewr` columns `S_day` and `A_day` for a particular row have entries of 1, respectively, if the flight is scheduled to arrive or if it actually arrives on the next day.

Before addressing these problems, let's examine the data types of the different columns of the `ewr` DataFrame.

```
 In[15]: ewr.dtypes
Out[15]:
Destination      object
Airline          object
Flight           object
Departure        object
Terminal         object
Status           object
Arrival_time     object
A_day           float64
Scheduled        object
S_day           float64
dtype: object
```

We note that `Departure`, `Arrival_time`, and `Scheduled` are not formatted as datetime objects. To convert them to datetime objects, we use Pandas' `apply` method, which applies a function to a column (the default) or a row (by setting the keyword `axis=0`) of a DataFrame. Here, we use the Pandas function `pd.to_datetime`.

```
 In[16]: ewr[['Departure', 'Arrival_time', 'Scheduled']] = \
    ...:     ewr[['Departure','Arrival_time', 'Scheduled']] \
    ...:     .apply(pd.to_datetime)

 In[17]: ewr.dtypes
Out[17]: ewr.dtypes
Destination                 object
Airline                     object
Flight                      object
Departure       datetime64[ns]
Terminal                    object
Status                      object
Arrival_time    datetime64[ns]
A_day                      float64
Scheduled       datetime64[ns]
S_day                      float64
dtype: object
```

Next, we set the dates. First, we use the datetime `replace` method to reset the year, month, and day of all dates to the departure date for all the flights: 2018-05-16.

```
 In[18]: for s in ['Departure', 'Arrival_time', 'Scheduled']:
    ...:      ewr[s] = ewr[s].apply(lambda dt:
    ...:                dt.replace(year=2018, month=5, day=16))
```

Finally, we add a day to those dates in the `Scheduled` and `Arrival_time` columns that have a 1 in the corresponding `S_day` and `A_day` columns with this code snippet.

Code: add_day_snippet.py

```python
from datetime import timedelta
i_A = ewr.columns.get_loc("Arrival_time")  # i_A  = 6
i_Ap = ewr.columns.get_loc("A_day")        # i_Ap = 7
i_S = ewr.columns.get_loc("Scheduled")     # i_S  = 8
i_Sp = ewr.columns.get_loc("S_day")        # i_Sp = 9
for i in range(ewr.shape[0]):
    if ewr.iloc[i, i_Ap] >= 1:
        ewr.iloc[i, i_A] += timedelta(days=ewr.iloc[i, i_Ap])
    if ewr.iloc[i, i_Sp] >= 1:
        ewr.iloc[i, i_S] += timedelta(days=ewr.iloc[i, i_Sp])
```

After running this, the datetime stamps are correct for all the datetime entries, which we check by printing out some times for flights that departed late in the day.

```
 In[19]: ewr[['Departure', 'Arrival_time', 'Scheduled']][-45:-40]
Out[19]:
Departure             Arrival_time           Scheduled
1510 2018-05-16 21:55:00 2018-05-17 11:25:00 2018-05-17 11:40:00
1511 2018-05-16 21:57:00 2018-05-17 00:53:00 2018-05-16 23:05:00
1512 2018-05-16 21:57:00 2018-05-17 00:53:00 2018-05-16 23:05:00
1513 2018-05-16 21:59:00 2018-05-16 23:29:00 2018-05-16 23:42:00
1514 2018-05-16 21:59:00 2018-05-16 23:29:00 2018-05-16 23:42:00
```

Let's calculate the difference between the actual `Arrival_time` and the `Scheduled` arrival time in minutes.

```
 In[20]: late = (ewr['Arrival_time']
- ewr['Scheduled']).dt.total_seconds()/60

 In[21]: late.hist(bins=range(-50, 300, 10))
Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x1245b39b0>
```

Note that instead of setting the number of bins, as we have done previously, we specify the widths of the bins and their precise placement using the `range` function.

Let's go ahead and add axis labels to our plot, which is displayed in Figure 11.7.

```
 In[22]: xlabel('minutes late')
Out[22]: Text(0.5,23.5222,'minutes late')

 In[23]: ylabel('number of flights')
Out[23]: Text(38.2222,0.5,'number of flights')
```

11.6.4 Custom Aggregation of DataFrames

Pandas has several built-in functions and methods for extracting useful information from Pandas Series and DataFrames, some of which are listed in Table

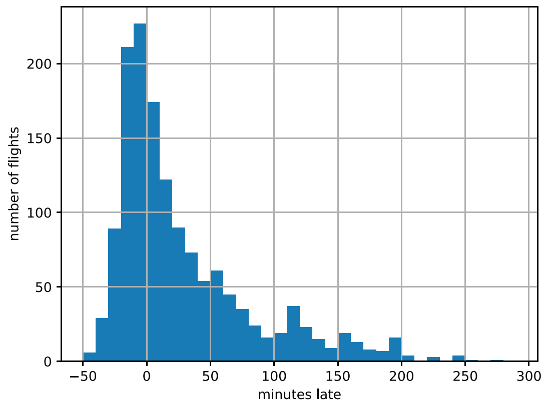


Figure 11.7 Histogram of late arrival times.

TABLE 11.2 Statistical methods for Pandas DataFrame and Series.

Function	Description	Function	Description
`min`	minimum	`cummin`	cumulative minimum
`max`	maximum	`cummax`	cumulative maximum
`mean`	mean	`skew`	skewness
`median`	median	`kurt`	kurtosis
`mode`	mode	`quantile`	quantile
`var`	variance	`mad`	mean abs deviation
`std`	standard deviation	`sem`	standard error of mean
`abs`	absolute value	`count`	num non-null entries
`sum`	sum	`cumsum`	cumulative sum
`prod`	product	`cumprod`	cumulative product
`describe`	count, mean, std, min, max, & percentiles		

11.2. These can be thought of as aggregation functions because they aggregate a set of data into some scalar quantity, like the minimum or maximum of a data set.

In addition to these built-in methods, Pandas has a method `agg` that allows you to implement your own aggregation functions. Suppose, for example, we would like to characterize the distribution of late arrival times plotted in Figure 11.7. We could use the `std` method to characterize the width of distribution, but that would miss the fact that the distribution is obviously wider on the late (positive) side than it is on the early (negative) side.

To take this asymmetry into account, we devise our own function `siglohi` that calculates two one-sided measures of the width of the distribution.

Code: siglohi.py

```python
def siglohi(x, x0=0, n=2):
    xplus = x[x > x0] - x0
    xminus = x0 - x[x < x0]
    sigplus = ((xplus**n).mean())**(1/n)
    sigminus = ((xminus**n).mean())**(1/n)
    return sigminus, sigplus
```

By default, the function calculates the square root of the negative and positive second moments about zero. Using its optional keyword arguments, it can calculate the n^{th} root of the n^{th} moment and can center the calculation of the moments around any value, not just zero.

We demonstrate how `siglohi` works on the Series `late` of the distribution of flight arrival times that we developed in the previous section. We use the Pandas `agg` method on the Series `late` with our function `siglohi` as the argument of `agg`.

```
In [24]: late.agg(siglohi)
Out[24]: (16.613569037283458, 78.8155571711229)
```

As expected, the width is much smaller on the early (nagative) side than it is on the late (positive) side.

The optional keyword arguments of `siglohi` are passed in the usual way (see Section 7.1.7). For example, to calculate the cube root of the third moment, we set the optional argument `n` equal to 3 as follows:

```
In [25]: late.agg(siglohi, *(0, 0, 3))
Out[25]: (18.936255193664774, 96.23261210488258)
```

Note that there are *three*, not two, optional arguments. The first is the `axis`, which is an optional argument (the only one) for the `agg` method, and the second and third are `x0` and `n`, the optional arguments of `siglohi`. Alternatively, we can call `agg` with the `axis` argument of `agg` set to `zero` as a positional argument as follows:

```
In [26]: late.agg(siglohi, 0, *(0, 3))
Out[26]: (18.936255193664774, 96.23261210488258)
```

Either way, the result is the same.

Finally, we note that `siglohi` can be used on `late` as a function in the usual way with `late` as an explicit argument of `siglohi`.

```
In [27]: siglohi(late, n=3)
Out[27]: (18.936255193664774, 96.23261210488258)
```

11.7 EXERCISES

1. Read the planetary data in the text file `planetData.txt` into a Pandas DataFrame and perform the following tasks:

   (a) Print out the DataFrame.

   (b) Based on the data read from the file, find the average density of each planet relative to that of the Earth and add the results as a column in your DataFrame. Then, print out the new DataFrame in order from the most dense to the least dense.

   (c) Print out your DataFrame sorted from the largest-diameter to smallest-diameter planet.

   (d) Print out your DataFrame with only those planets with masses greater than Earth's, sorted from least to most massive planet.

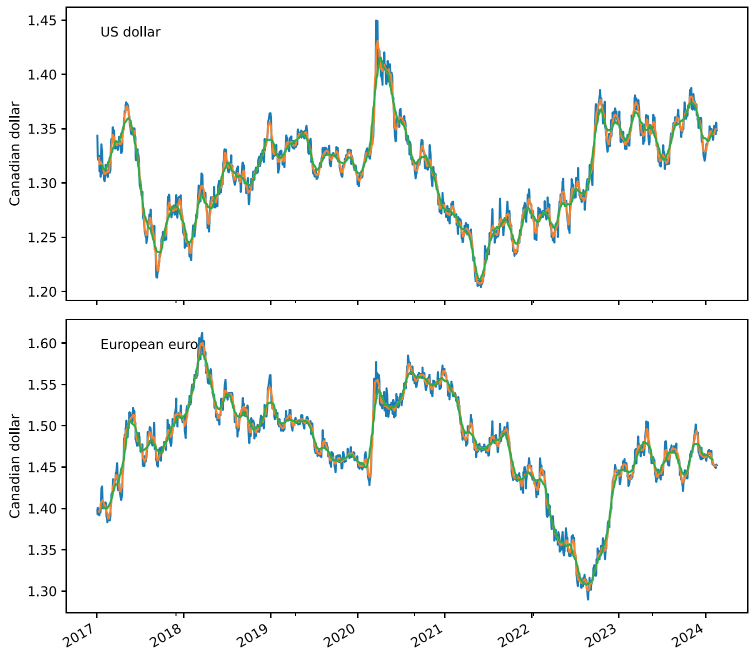
2. Starting from the program `urlRead.py` on page 327 (*i.e.*, start with same code in lines 1–19), write a program that calculates how much different currencies have fluctuated relative to the US dollar (or some other currency of your choosing) since January 3, 2017.

   Before writing your code, download the CSV file and load it into Excel or a similar spreadsheet program to see the data's structure. You can download the CSV file by going to the url given by line 6 of `urlRead.py` on page 327.

   Once you examine the CSV file's structure, make sure you understand exactly what lines 1–19 of `urlRead.py` are doing.

   The first thing the code you write should do is find the average a, maximum m, and standard deviation s of the value of each currency relative to the US dollar over the period of time starting from the first business day of 2017, January 3rd. Then create a new DataFrame with columns that list `av` $= a$, `mx` $= m/a$, and `sd` $= s/a$ along with the name of each currency, as shown in the listing below. The DataFrame should be sorted from the largest fluctuation (standard deviation) `sd` to the smallest. Print out the DataFrame; it should start like this:

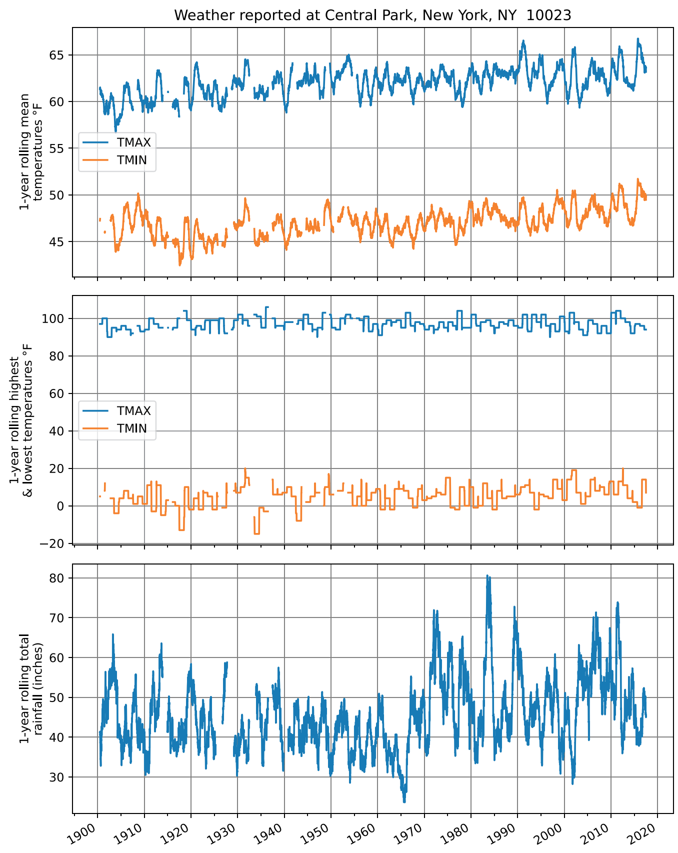
```
description     av      mx       sd
id
FXTRYCAD        Turkish lira  0.159  1.8488  0.46087
FXBRLCAD      Brazilian real  0.234  1.3935  0.21171
FXRUBCAD       Russian ruble  0.015  1.2505  0.11391
FXZARCAD  South African rand  0.068  1.2748  0.10972
```



3. Starting from the program `urlRead.py` on page 327, extend the code to make a plot like the one above. The three traces in each of the two plots give, respectively, the daily exchange rate and the daily exchange rate as a centered running average over 10 (business) days and over 30 days. Look up the Pandas rolling averages routine `pd.Series.rolling`, which you should find useful in making this plot. Write your program so you can switch the currency plotted by changing a single variable.

4. Go to the website https://www.ncdc.noaa.gov/cdo-web/search and make a request to download weather data for some place that interests you. We requested weather data as a CSV file for Central Park in New York City (zip code 10023), as it dates back from the 19th century, although we chose to receive data dating from January 1, 1900.

   (a) Read the weather data you download into a DataFrame, ensuring the date is formatted as a datetime object and set to be the DataFrame index. Print out a table with the date as the first column and the daily precipitation, maximum temperature, and minimum temperature for one month of your choosing. The headings for those data are `PRCP`, `TMIN`, and `TMAX`, respectively.



(b) Get a list of the dates when more than 5 inches of rain fell and the rainfall on those dates. If this does not yield any results for your data, reduce the number of inches until you get a few dates.

(c) Make a plot like the one on the previous page from the data set you downloaded. The top graph plots the 1-year running averages (centered) of the daily high and low temperatures. The middle graph plots the running 1-year high and low temperatures, which are remarkably stable around 0°F and 100°F. The bottom graph plots the 1-year running total rainfall. Write the program that makes these graphs so that the 1-year running quantities can be changed to 2, 3, or some other number of years. Look up the Pandas rolling averages routine `pd.Series.rolling`, which you should find useful in making this plot.

5. In this problem, you characterize chromosomes of the human genome, working with a CSV file that was downloaded from the National Center for Biotechnology Information (NCBI) of the US National Institutes of Health: https://www.ncbi.nlm.nih.gov/datasets/gene/taxon/9606/. The data file is called `ncbi_dataset.csv`. Read this data file into a Pandas DataFrame and perform the following analyses:

   (a) Compute and print out the number of genes listed for the human genome (there is one per row).

   (b) Compute and print out the minimum, maximum, average, and median number of known isoforms per gene (consider the `transcript_count` column as a Series).

   (c) Plot a histogram of the number of known isoforms per gene. As these numbers vary over a wide range, use a logarithmic y-axis, as shown in the upper right plot in the figure below.

   (d) Compute and print out the number of different gene types.

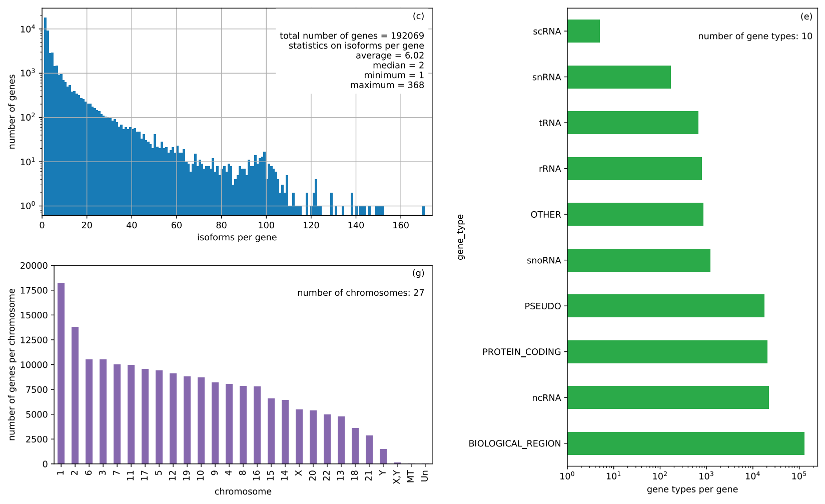
   (e) Compute and print out the total number of genes and the number of genes for each `gene_type`. Make a horizontal bar graph that shows the number of genes for each type associated with each gene in decreasing order of gene type.

   (f) Compute and print out the number of different chromosomes.

   (g) Compute and print out the number of genes for each chromosome. Make a vertical bar plot of the number of genes for each chromosome in decreasing order.

   (h) Compute and print out the percentage of genes with the plus orientation for each chromosome.

   (i) Compute and print out the average number of transcripts associated with each gene type.


(c)
total number of genes = 192069
statistics on isoforms per gene
average = 6.02
median = 2
minimum = 1
maximum = 368
number of genes
isoforms per gene
(g)
number of chromosomes: 27
number of genes per chromosome
chromosome
(e)
number of gene types: 10
gene_type
gene types per gene
scRNA
snRNA
tRNA
rRNA
OTHER
snoRNA
PSEUDO
PROTEIN_CODING
ncRNA
BIOLOGICAL_REGION

CHAPTER 12

Animation

This chapter teaches you how to use Matplotlib's Animation package. You learn how to ***animate a sequence of images*** *to make a video and then how to add text and other features to your videos. You learn how to* ***animate functions****. You also learn how to combine movies and animated plots side-by-side. You learn how to animate a fixed number of frames and how to animate an arbitrary number of frames until some condition is met.*

It's often not enough to plot our data; we want to see it move! Simulations, dynamical systems, wave propagation, explosions—they all involve time evolution. Moreover, the human brain is well-adapted to extract and understand spatial information in motion. Therefore, we want to animate our representations of information.

While not strictly necessary, we use the *animation* library of Matplotlib to make animations. It is powerful, relatively easy to use, and well-suited for animating functions, data, and images.

12.1 ANIMATING A SEQUENCE OF IMAGES

One of the most basic animation tasks is to make a movie from a sequence of images stored in a set of image files. If the size and number of images are not too large, you can read all the images into your program (*i.e.*, into memory) and then use the function `ArtistAnimation` of Matplotlib's `Animation` class to play a movie. You can also save the movie you make to an external file using the `save` function from the Animation module.

DOI: 10.1201/9781032673950-12

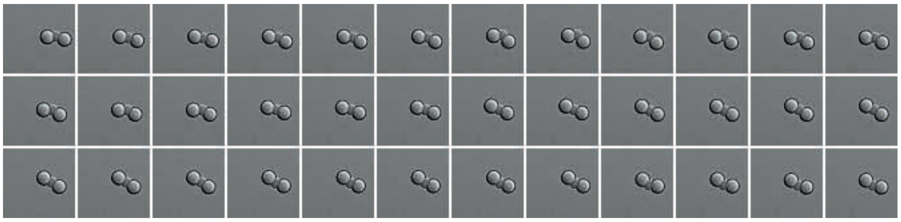


Figure 12.1 Partial sequence of images for animation.

12.1.1 Simple Image Sequence

First, we make a video from a sequence of images. Later, we will show you how to add text and other animated features to your movie.

We will make a movie from a sequence of images of micrometer-size particles suspended in water that are undergoing Brownian motion. Figure 12.1 shows a selection of the sequence of images.

The names of the sequence of image files to be animated should consist of a base alphanumeric string—any legal filename—followed by an *n*-digit integer, including leading zeros so that every file has a name with the same number of characters. The images will be animated from the smallest to the largest numbers. As an example, suppose we want to animate a sequence of 100 image files named `s000.png`, `s001.png`, `s002.png`, ..., `s099.png`.

Here is our program. Below, we explain how it works.

Code: movie_from_images.py

```python
import matplotlib.pyplot as plt
import matplotlib.animation as anim
from PIL import Image
from glob import glob

fig, ax = plt.subplots(figsize=(3.6, 3.5))
fig.subplots_adjust(bottom=0, top=1, left=0, right=1)
ax.axis("off")

ims = []
for fname in sorted(glob("pacb/s0*.png")):
    # print(fname)  # uncomment to follow loading of images
    im = ax.imshow(Image.open(fname), animated=True)
    ims.append([im])

ani = anim.ArtistAnimation(fig, artists=ims, interval=33,
                           repeat=False)
# Uncomment to save as mp4 movie file.  Need ffmpeg.
# ani.save("movies/movie_from_images.mp4", writer="ffmpeg")

fig.show()
```

We animate the file sequence using the `ArtistAnimation` function, which is part of the `matplotlib.animation` library. It's called in lines 16- 17, and, in this example, it has four arguments.

The first argument is the name of the figure window, in this case, `fig`, where the animation will be rendered.

The second argument, with the keyword `artists`, must be a list of lists (or tuples) that contain the images to be animated. Below, we explain how to put together such a list.

The third argument, `interval`, specifies the time in milliseconds between successive animation frames. In this example, it's 30 ms, corresponding to $1000/30 = 33.3$ frames per second.

The fourth argument, `repeat`, tells the animation to play through one time when it's set to `False`, rather than repeating in a loop over and over again.

The `ArtistAnimation` call (line 16) must be assigned a variable name, which here is `ani`. Otherwise, the animation will be deleted before it can display the sequence of images. Do not forget to give any `ArtistAnimation` call a name!

Aside from calling the function `ArtistAnimation`, the main tasks for the program are to set up the figure window and then assemble the list `ims` that contains the images to be rendered for the animation.

Lines 6–8 set up the figure window. The argument `figsize` is set to have the same aspect ratio as the movie frames we want to animate. Then, the function `subplots_adjust` is set so that frames take up the entire figure window. In line 8, we turn off all the axes labels, as we do not want them for our animation.

In line 10, we create an empty list, `ims`, that will contain the images to be animated.

The `for` loop starting at line 11 reads in an image frame from a sequence of image files, formats it for animation and then adds it to the list of images to be animated.

To read the names of our data files, we use the function `glob` from the module of the same name. The function `glob` returns a list of paths on your computer matching a pathname pattern. The asterisk symbol * acts as a wildcard for any symbol. Typing `glob('pacb/s*.png')` returns the list of all files on my computer matching this pattern, which turns out to be a sequence of 100 image files named `s000.png`, `s002.png`, `s003.png}, \ldots, \texttt{s099.png` that are located in the `pacb` subdirectory of the directory where our program is stored. To ensure that the list of filenames is read in numerical order, we use the Python function `sorted` in line 11 (which may not be necessary if the file timestamps are in the correct order). We can restrict the wildcard using square brackets with entries that specify which characters must appear at least once in the wildcard string. For example `glob('pacb/s00*[0-2].png')` returns the list `['pacb/s000.png', 'pacb/s001.png', 'pacb/s002.png']`. You can experiment independently to understand better how `glob()` works with wildcards. Uncommenting line 12 prints out the names of the data files read and parsed by `glob()`, which can serve as a check that `glob()` is working as expected.

Two functions are used in line 13: `PIL.Image.open()` from the Python Image Library (PIL) reads an image from a file into a NumPy array; `imshow()` from the Matplotlib library displays an image stored as a NumPy array, on the figure axes. The image is not displayed immediately but is stored with the name `im`. Note that in each iteration of the `for` loop, we read in one frame from the sequence of frames that make up the animation clip we are putting together.

In the final line of the `for` loop, we append `[im]` to the list `ims` that we defined in line 10 just before the `for` loop. Note that `im` is entered with square brackets around it so that `[im]` is a one-item list. Thus, it is added as a list to the list `ims`, so that `ims` is a list of lists. This is the format that the function `ArtistAnimation` needs for the `artists` argument.

Finally, we save the movie[1] as an `mp4` movie so that it can be played independently of the program that produced it (and without running Python). Alternatively, changing the file name to `'pacb.avi'` saves the movie as an `avi` file. The `mp4` and `avi` movies and the Python code above that produced them are available at https://github.com/djpine/python-scieng-public-2.

As an alternative to the program provided and discussed above, we offer one that is self-contained so that you do not need to load a sequence of images from files. Instead, this program makes the images on the fly purely for demonstration purposes. The program is adapted from an example on the Matplotlib website.[2]

Code: movie_from_images_alt.py

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as anim


def f(x, y):
    return np.sin(x) + np.cos(y)
```

[1] To save a movie to your computer, you must install a third-party MovieWriter that Matplotlib recognizes, such as FFmpeg. See Section A.3 for instructions on downloading and installing FFmpeg. Alternatively, you can comment out the `ani.save` call (line 27) so that the program runs without saving the movie.

[2] See https://matplotlib.org/examples/animation/dynamic_image2.html.

```
8
9
10  x = np.linspace(0, 2 * np.pi, 120)
11  y = np.linspace(0, 2 * np.pi, 120).reshape(-1, 1)
12
13  fig, ax = plt.subplots(figsize=(3.5, 3.5))
14  fig.subplots_adjust(bottom=0, top=1, left=0, right=1)
15  ax.axis("off")
16  ims = []
17  for i in range(120):
18      x += np.pi / 20.
19      y += np.pi / 20.
20      im = ax.imshow(f(x, y), cmap=plt.get_cmap("plasma"),
21                     animated=True)
22      ims.append([im])
23
24  ani = anim.ArtistAnimation(fig, artists=ims, interval=33,
25                             repeat_delay=0)
26  # Uncomment to save as mp4 movie file.  Need ffmpeg.
27  # ani.save("movies/movie_from_images_alt.mp4", writer="ffmpeg")
28
29  fig.show()
```

The 2D NumPy array is created with `f(x, y)` in lines 20–21, in place of reading in image files from disk. The only other notable difference is that here, we let the animation repeat over and over. We set the delay between repetitions to be 0 ms, so the animation appears as an endless repeating clip without interruption.

12.1.2 Annotating and Embellishing Videos

Adding dynamic text or highlighting various features in a video is often useful. In the sequence of images animated in the program `movie_from_images.py` (see page 352), two outer particles rotate around a central particle, forming a kind of ball-and-socket joint. We want to highlight the angle of the joint and display its value in degrees as the system evolves over time. We do this by adding some Matplotlib Artists to each frame. Figure 12.2 shows one frame of what our program will eventually display.

To start, we need data that gives the positions of the three particles as a function of time. These data are provided in an Excel spreadsheet called trajectories.xlsx, which is read by the program in line 20.

Next, we construct a list, `ims`, that will contain a set of lists the animation routine will display. In a previous example, each element of the list `ims` was a one-item list `[im]` of PNG images (see line 14 in `movie_from_images.py` listed on page 352). Adding dynamic text and other features, this one-item list becomes a three-item list in the program below.

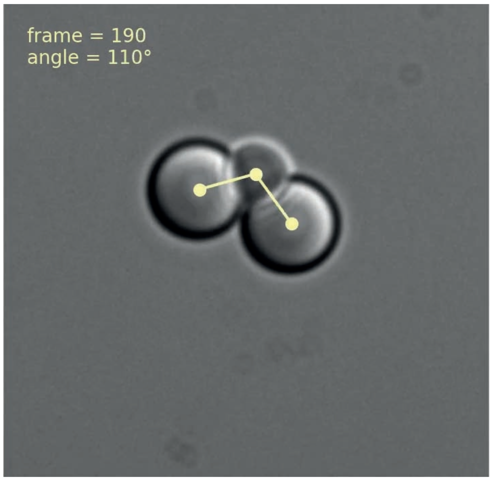


Figure 12.2 Annotated frame highlighting particle positions and displaying angle.

Code: movie_from_images_annotated.py

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as anim
import pandas as pd
from PIL import Image
from glob import glob


def angle(x, y):
    a = np.array([x[0] - x[1], y[0] - y[1]])
    b = np.array([x[2] - x[1], y[2] - y[1]])
    cs = np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))
    if cs > 1.0:
        cs = 1.0
    elif cs < -1.0:
        cs = -1.0
    return np.rad2deg(np.arccos(cs))


r = pd.read_excel("trajectories.xlsx", usecols="A:F")

fig, ax = plt.subplots(figsize=(3.6, 3.5))
fig.subplots_adjust(bottom=0, top=1, left=0, right=1)
ax.axis("off")

ims = []
angles = []
for i, fname in enumerate(sorted(glob("pacb/s[0-2]*.png"))):
    # print(fname)  # uncomment to follow loading of images
    im = ax.imshow(Image.open(fname), animated=True)
```

```
31      # Make 3 solid points connect by two bars
32      x = np.array([r["x1"][i], r["xc"][i], r["x2"][i]])
33      y = np.array([r["y1"][i], r["yc"][i], r["y2"][i]])
34      ima, = ax.plot(x, y, "o-", color=[1, 1, 0.7])
35      # Get angle between bars & write on movie frames
36      theta = angle(x, y)
37      angles.append(theta)
38      imb = ax.text(0.05, 0.95,
39                    "frame = {0:d}\nangle = {1:0.0f}\u00B0"
40                    .format(i, theta), va="top", ha="left",
41                    color=[1, 1, 0.7], transform=ax.transAxes)
42      ims.append([im, ima, imb])
43
44  ani = anim.ArtistAnimation(fig, artists=ims, interval=33,
45                             repeat=False)
46  # Uncomment to save as mp4 movie file.  Need ffmpeg.
47  # ani.save("movies/movie_from_images_annotated.mp4", writer="ffmpeg")
```

The first item of the list is `im`, a list of the same PNG images we used before. This element is created in line 30.

The second item in the list is `ima`, a line plot connecting the centers of the three particles, where each center is indicated by a circular data point. This Artist is created in line 34. Note that a comma is used in defining `ima` because the `plot` function creates a one-element list, and we want the element itself, not the list.

The third item in the list is `imb`, a text Artist that displays the frame number and the angle of the ball-and-socket joint, calculated by the function `angle`. This Artist is created in lines 38–41.

The three items become the elements of a list `[im, ima, imb]` that represents one frame of our video: a PNG file, a plot, and text. Each frame, `[im, ima, imb]`, becomes an element in the list `ims`, which represents all the frames of the entire video.

The function `ArtistAnimation` is called with essentially the same inputs that we used previously. This time, we choose not to have the video loop, but instead, we have it stop after it plays through one time.

Finally, you may have noticed that in line 28, we changed the argument of `glob`. The `[0-2]` is a wildcard that specifies that only 0, 1, and 2 will be accepted as the first character in the file name. In this way, a movie from `000` to `299` is made.

12.2 ANIMATING FUNCTIONS

Suppose you want to visualize the nonlinear single pendulum whose solution we calculated in Chapter 9. While it might not seem obvious, the simplest way to do these animations is with the *function animation* routine, called

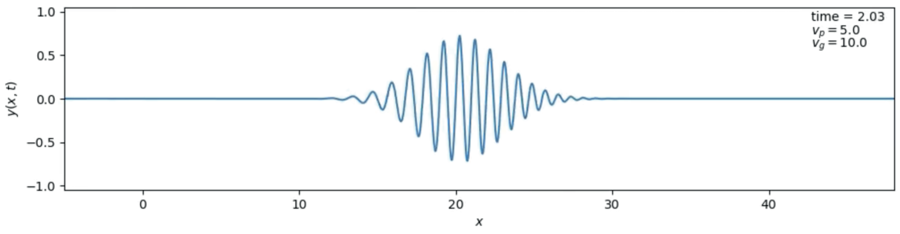


Figure 12.3 One frame from the propagating wave packet movie.

`FuncAnimation`, of Matplotlib's mpl's `Animation` library. As its name implies, `FuncAnimation` can animate functions, but it turns out that animating functions encompasses a wide spectrum of animation tasks, more than you might have imagined.

12.2.1 Animating for a Fixed Number of Frames

We start by writing a program to animate a propagating wave packet with an initial width a_0 that spreads with time. The equation for the wave packet is given by the real part of

$$u(x,t) = \frac{1}{\sqrt{\alpha + i\beta t}} e^{ik_0(x-v_p t)} e^{-(x-v_g t)^2/4(\alpha+i\beta t)} ,$$

where $i \equiv \sqrt{-1}$, $\alpha = a_0^2$, and $\beta = v_g/2k_0$. The phase and group velocities are v_p and v_g, respectively, and $k_0 = 2\pi/\lambda_0$, where λ_0 is the initial wavelength of the wave packet. Figure 12.3 shows the wave packet at a particular moment.

Here is the program for animating the propagation of the wave packet. An explanation of how it works follows the listing.

Code: wave_packet_spreads.py

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as anim


def ww(x, t, k0, a0, vp, vg):
    tc = a0 * a0 + 1j * (0.5 * vg / k0) * t
    u = np.exp(1.0j * k0 * (x - vp * t) - 0.25 * (x - vg * t)**2 / tc)
    return np.real(u / np.sqrt(tc))


wavelength = 1.0
a0 = 1.0
k0 = 2 * np.pi / wavelength
vp, vg = 5.0, 10.0
```

```
16 period = wavelength / vp
17 runtime = 40 * period               # total time to follow wave
18 rundistance = 0.6 * vg * runtime    # total distance to plot wave
19 dt = period / 12.0                  # time between frames
20 tsteps = int(runtime / dt)          # total number of times wave
21 
22 fig, ax = plt.subplots(figsize=(12, 3))
23 fig.subplots_adjust(bottom=0.2)  # allow room for axis label
24 x = np.arange(-5 * a0, rundistance, wavelength / 20.0)
25 ax.text(0.9, 0.91, r"$v_p = {0:0.1f}$".format(vp),
26         ha="left", va="top", transform=ax.transAxes)
27 ax.text(0.9, 0.84, r"$v_g = {0:0.1f}$".format(vg),
28         ha="left", va="top", transform=ax.transAxes)
29 ax.set_xlabel(r"$x$")
30 ax.set_ylabel(r"$y(x,t)$")
31 ax.set_xlim(-5 * a0, rundistance)
32 ax.set_ylim(-1.05, 1.05)
33 # Define containers for dynamic elements
34 line, = ax.plot(x, np.ma.array(x, mask=True), color="C0")
35 timeText = ax.text(0.9, 0.98, "", ha="left", va="top",
36                    transform=ax.transAxes)
37 timeString = "time = {0:0.2f}"
38 
39 
40 def animate(i):
41     t = float(i) * dt
42     line.set_ydata(ww(x, t, k0, a0, vp, vg))  # update y-data
43     timeText.set_text(timeString.format(t))
44     return line, timeText
45 
46 
47 ani = anim.FuncAnimation(fig, func=animate,
48                          frames=range(tsteps),
49                          interval=30.0, blit=True)
50 # Uncomment to save as mp4 movie file.  Need ffmpeg.
51 # ani.save("movies/wave_packet_spreads.mp4", writer="ffmpeg")
52 fig.show()
```

After importing the relevant libraries, lines 6–9 define the wave packet using complex algebra, with line 9 returning only the real part. The physical parameters defining various properties of the wave are initialized in lines 12–15. The range of times and distances over which the waveform will be calculated are determined in lines 16–18.

The frame time interval and the total number of frames are set in lines 19–20.

12.2.1.1 *Setting up Static Elements of an Animation*

Lines 22–32 set up the *static* elements of the figure—everything about the figure *that does not change* during the animation. The $x$-array is defined in line 24, which remains fixed throughout the animation—it does not change with time. The distance between points along the x-axis is set to be small enough, $1/20^{\text{th}}$ of a wavelength, to make the waveform appear smooth.

Lines 25–28 set up two static text elements that give the values of the phase and group velocities of the wave packet.[3] These text elements appear at the upper right of the plot. The keyword argument `transform=ax.transAxes` is used to specify the text in axis coordinates, where `0, 0` is lower-left and `1, 1` is upper-right; without this argument, data coordinates are used.

Lines 31–32 fix the limits of the x and y axes. Setting the plot limits to fixed values is generally recommended for an animation.

Lines 34–37 set up static *containers* for the *dynamic* elements of the routine: a container for the plot of the moving wavepacket function and a container for the dynamic text that updates the time displayed at the upper right of the plot.

Line 34 merits special attention. Here, the program sets up a *container* for the animated plot of the waveform. Note that the x-data are loaded but not the y-data. The reason is that the y-data will *change* as the wave propagates; at this point in the program, only the *fixed* unchanging elements of the plot are set up. A fully masked x-array is used as a placeholder for the y-array, which is guaranteed to have the correct number of elements but will not plot. You could also put an appropriately initialized version of the y-array here without consequence since the `plot` command is not rendered until the `show()` command is called at the end of the routine. However, if you are running the program from IPython *and* you have interactive plotting turned on (`plt.ion()`), the `plot` command will be rendered immediately, which spoils the animation. Using a masked array for the y-data in the `plot` command avoids this problem so that the animation is rendered correctly irrespective of whether interactive mode is on or off.

Lines 35–36 set up a text container `timeString` used to display the current time in the animation.

12.2.1.2 Why is there a Comma after an Assigned Variable Name?

Notice that line 34 has a comma after the name `line`. This is important. To understand why, consider the following command issued from the IPython shell:

```
In[1]: plot([1, 2, 3, 2, 3, 4])
Out[1]: [<matplotlib.lines.Line2D at 0x181dca2e48>]
```

[3]The phase velocity is the speed with which the crests in the wave packet moves; the group velocity is the overall speed of the wave packet. Don't worry if you're not familiar with these terms.

Notice that the `plot` command returns a one-item *list*, which is indicated by the square brackets around the Matplotlib line object `<matplotlib.lines.Line2D at 0x181dca2e48>`. Writing `[line,] = plot(…)`, or equivalently writing `line, = plot(…)`, sets `line` equal to the first (and only) element of the list, which is the line object `<matplotlib.lines.Line2D at 0x181dca2e48>`, rather than the list. It is the line object that the `FuncAnimation` needs, not a list, so writing `line, = plot(…)` is the right thing to do. By the way, you could also write `line = plot(…)[0]`. That works too!

Note that in contrast to the `line` object, `timeString` set up in lines 35–36 is a simple object, not a list, so a trailing comma is not needed.

12.2.1.3 Animating a Function (and text)

The animation is done in lines 47–49 by `FuncAnimation` from Matplotlib's Animation package. The first argument is the name of the figure window, `fig` in this example, where the animation is to be rendered.

The second argument, `func`, specifies the function's name, here `animate`, that updates each animation frame. We defer explaining how the function `animate` works until after the discussion of the other arguments of `FuncAninmation()`.

The third argument is an iterator, in this case `range(tsteps)`, that provides the frame number of the animation. The iterator serves two functions: (1) it provides data to `func` (here `animate`), in this case a single integer that is incremented by 1 each time a new frame is rendered; (2) it signals `FuncAnimation` to keep calling the routine `animate` until the iterator has run through all its values. Then, `FuncAnimation` will restart the animation from the beginning unless an additional keyword argument, `repeat`, not used here, is set equal to `False`.

The `interval` argument sets the time in milliseconds between successive frames. Here, the value of `dt` is set to 30 ms, fast enough for a human brain to perceive the animation as continuous in time.

The last argument, `blit=True`, turns on *blitting*. Blitting is the practice of redrawing only those elements in an animation that have changed from the previous frame, instead of redrawing the entire frame each time. This can save a great deal of time and allow an animation to run significantly faster.

The function `animate` updates each frame of the animation. In this example, updating means calculating the y-values of the wave packet for the next time step and providing those values to the 2D line object—the wave packet—that was named `line` in line 34. This is done using the `set_ydata()` function, which is attached via the *dot* syntax to the name `line`. The argument of `set_ydata()` is simply the array of updated y-values, which is calculated by the function `ww()`. The `set_text` command updates the text container that was set up in lines 35–36.

`FuncAnimation` does the rest of the work, updating the animation frame by frame at the specified rate until the animation finishes. Once finished, the animation starts again since we did not set the keyword argument `repeat=False`.[4]

The next statement `ani.save('wavepacket.mp4')` saves the animation (one iteration only) in the current directory as an mp4 video that can be played with third-party applications on any platform (in principle).

12.2.2 Animating until a Condition is Met

In the previous section, our animation of the propagating wave packet ran for a preset number of steps (`tsteps`). This is a sensible way to make an animation when you know or can calculate ahead of time how long you want the animation to run. In other cases, however, you may want the animation to run until some condition is met, and exactly how long this takes is not known in advance. This is generally the case when the animation involves some random process. A simple but powerful way to do this is to write a *generator* function, which is used as the `frames` keyword argument in `FuncAnimation`.

To illustrate this kind of animation, we introduce an algorithm known as *random organization*, which first appeared in the context of a physics problem.[5] In its simplest form, we consider a set of N spheres with a diameter of 1 randomly placed along the circumference of a circle of length $L > N$, as shown in Figure 12.4. Nearby spheres overlap if the distance between their centers is less than 1. To aid visibility, spheres that overlap are colored differently from spheres that do not. The time evolution of the system proceeds as follows: Each time step, the subroutine `move` checks all N spheres to see which spheres, if any, overlap each other. Any sphere that overlaps with one or more spheres is then given a kick that moves it a random distance between $-\epsilon$ and $+\epsilon$, where ϵ is typically about $^1/_4$ or less. Spheres that do not overlap with any of their neighbors do not move. That ends one time step. The number of overlapping spheres may increase, decrease, or remain the same for any given time step. In the next time step, the process repeats. The algorithm continues as long as there are still spheres that overlap. In practice, it is found that all spheres eventually find a position at which they do not overlap with any other

[4]If you consult the online documentation on `FuncAnimation`, you will see that there is a keyword argument `init_func`, which the documentation states is used to draw a clear frame at the beginning of an animation. As far as we can tell, this function is redundant, so we don't use it, even though many web tutorials suggest it is necessary.

[5]Corté *et al.*, Nature Physics **4**, 420–424, (2008).

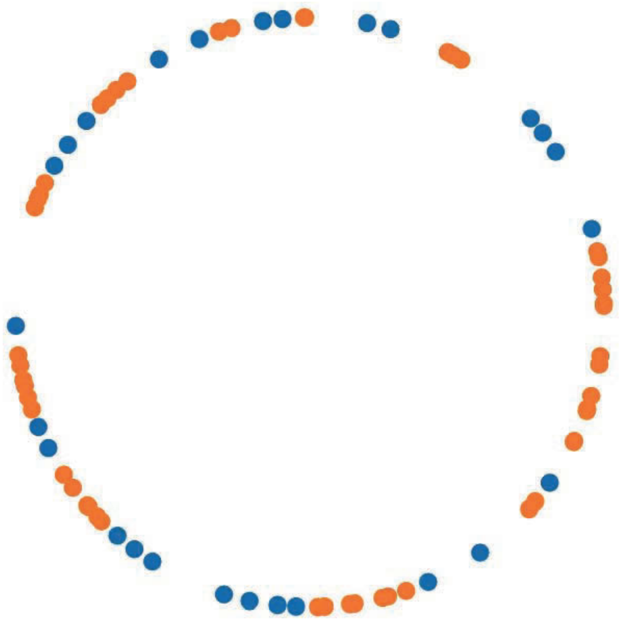


Figure 12.4 Random organization.

sphere if the number of spheres is not too high. For the conditions $L = 100$ and $\epsilon = 0.25$, the system eventually settles into a state where no spheres move if $N \leq 86$.

The random organization algorithm is implemented in the generator function `move` below. In this implementation, we use periodic boundary conditions, equivalent to bending the line on which the spheres move into a circle so that spheres at the very end of the line interact with spheres at the beginning of the line.

The generator function `move` returns two arrays: `x`, which gives the updated positions of the N spheres as a floating point number between 0 and L, and `changes`, an integer array of length N where the i^{th} entry is 1 if the i^{th} sphere has moved in the most recent time step and 0 if it hasn't.

Code: rand_org.py

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as anim


```

```python
def move(L, N, eps):  # generator for updating
    x = np.sort(L * np.random.rand(N))  # speeds up algorithm
    changes = np.zeros(N, dtype="int")
    moves = 1
    while moves > 0:
        changes.fill(0)  # tally changes starting with zero
        xc = np.copy(x)
        for i in range(N - 1):
            j = i + 1
            while x[j] - x[i] < 1.0:
                rr = 2.0 * (np.random.rand(2) - 0.5)
                xc[i] += eps * rr[0]
                xc[j] += eps * rr[1]
                changes[i] = 1
                changes[j] = 1
                if j < N - 1:
                    j += 1
                else:
                    break  # terminates while loop when j=N-1
            if x[i] < 1.0:  # periodic boundary conditions
                k = -1
                while x[i] + L - x[k] < 1.0:
                    rr = 2.0 * (np.random.rand(2) - 0.5)
                    xc[i] += eps * rr[0]
                    xc[k] += eps * rr[1]
                    changes[i] = 1
                    changes[k] = 1
                    k -= 1
        x = np.sort(xc % L)  # sort data for algorithm to work
        moves = np.sum(changes)
        yield x, changes


N, L, eps = 70, 100, 0.25  # inputs for algorithm

circumference = float(L)
radius = circumference / (2.0 * np.pi)
R = radius * np.ones(N)

fig, ax = plt.subplots(figsize=(8, 8),
                       subplot_kw=dict(polar=True))
pStill, = ax.plot(np.ma.array(R, mask=True), R,
                  "o", ms=12, color="C0")
pActiv, = ax.plot(np.ma.array(R, mask=True), R,
                  "o", ms=12, color="C1")
ax.set_rmax(1.1 * radius)
ax.axis("off")


def updatePlot(mv):
    x, changes = mv
    angle = 2.0 * np.pi * x / L
    active = np.ma.masked_where(changes != 1, angle)
    inactive = np.ma.masked_where(changes == 1, angle)
    pStill.set_xdata(inactive)
    pActiv.set_xdata(active)
```

```
62      return pStill, pActiv
63
64
65  ani = anim.FuncAnimation(fig=fig, func=updatePlot,
66                           frames=move(L, N, eps),
67                           interval=10, blit=True,
68                           save_count=10000,
69                           repeat=False)
70  # Uncomment to save as mp4 movie file.  Need ffmpeg.
71  # ani.save("movies/rand_org.mp4", writer="ffmpeg", dpi=200)
72  plt.show()
```

The output of `move(L, N, eps)` provides the input to the function `update(mv)`, which updates the animation. First, it unpacks `mv` in line 56, then converts the `x` array into angles for display purposes, and then creates two masked arrays, one for active and the other for inactive particles using the array `changes` that tracks which spheres moved in the most recent time step. These updated masked arrays, created in lines 58 and 59, are fed into the plots that were set up in lines 47–50, where the static data for the *y*-array—the unchanging radius of the polar plot—was already entered. The still and active data sets are updated and returned to `FuncAnimation`, which plots the next frame, with the moving particles shown in orange (`color='C1'`) and the stationary particles in blue (`color='C0'`).

A key feature of this approach is using a generator function, here `move`. The function returns two arrays, `x` and `changes`, using a `yield` statement, which is what makes the function a generator. Note that `move` yields these two arrays *inside* a while loop. A key feature of a generator function is that it remembers its current state between calls. In particular, `move` remembers the value of the positions `x` of all the spheres, it remembers that it is in a while loop, and it remembers the variable `move`, which keeps track of how many spheres were moved in the most recent time step. When `move` is zero, the while loop terminates, which signals `FuncAnimation` that the animation is finished.

In line 51, we use `set_rmax` to set the maximum radius of the polar plot. It is important to do this after the `plot` calls in lines 47 and 49, as they can reset the plot limits in unanticipated ways.

You can save the plot as an mp4 movie by uncommenting line 70. However, you need to have installed FFmpeg, as discussed in a footnote on page 354. Since, in this case, the length of the movie is unknown *a priori*, `FuncAnimation` has a keyword argument `save_count` that you can set to limit the number of frames that are recorded in the movie. Its default value is 100, so if you want to record more frames than that, you need to include it in the `FuncAnimation` call and set it to some other value. If the keyword `frames` is an iterable with a definite length (not the case here), it will override the `save_count` keyword argument value.

When running the program, be aware that the animation will only begin displaying once the movie is recorded for whatever number of frames you set, so if the number is large you may have to wait awhile before the animation appears.

12.2.2.1 Gilding the Lily

As nice as this animation is, plotting the number of active particles as a function of time to give a better sense of how the system evolves would be helpful. Therefore, in the program below, we add a plot inside the circular animation of the spheres. Aside from three additional lines discussed below, the program below is the same as the program starting on page 363 up to the line `ax.axis('off')`.

Code: rand_org_lily.py

```
ax.axis("off")

gs = gridspec.GridSpec(3, 3, width_ratios=[1, 4, 1],
                       height_ratios=[1, 2, 1])
ax2 = fig.add_subplot(gs[4], xlim=(0, 250), ylim=(0, L))
ax2.set_xlabel("time")
ax2.set_ylabel("active particles")
activity, = ax2.plot([], [], "-", color="C1")
tm, number_active = [], []


def updatePlot(mv):
    t, moves, x, changes = mv
    tm.append(t)
    number_active.append(moves)
    tmin, tmax = ax2.get_xlim()
    if t > tmax:
        ax2.set_xlim(0, 2 * tmax)
    angle = 2.0 * np.pi * x / L
    active = np.ma.masked_where(changes != 1, angle)
    inactive = np.ma.masked_where(changes == 1, angle)
    pStill.set_xdata(inactive)
    pActiv.set_xdata(active)
    activity.set_data(tm, number_active)
    return pStill, pActiv, activity


ani = anim.FuncAnimation(fig=fig, func=updatePlot,
                         frames=move(L, N, eps),
                         interval=10, blit=True,
                         save_count=10000,
                         repeat=False)
# Uncomment to save as mp4 movie file.  Need ffmpeg.
```

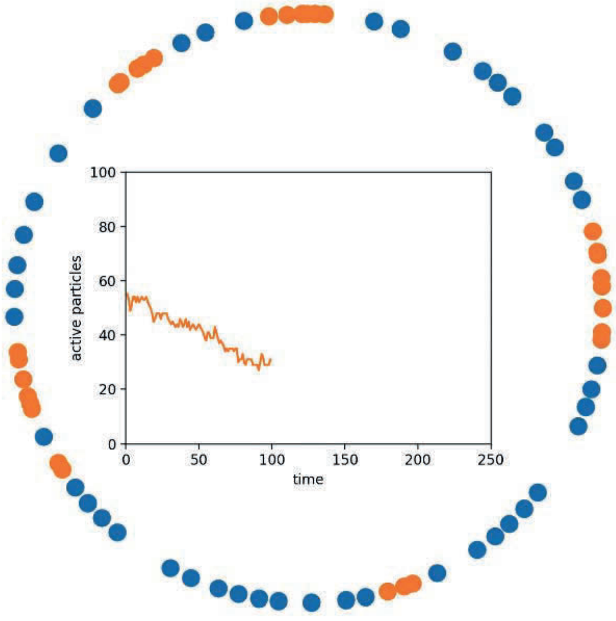


Figure 12.5 Random organization with a plot of the number of active particles *vs.* time.

```python
# ani.save("movies/rand_org_lily.mp4", writer="ffmpeg", dpi=200)
plt.show()
```

Lines 57–58 set up the area for the additional plot using the `gridspec` module from the Matplotlib library. To add this library, we include this line with the other import statements at the beginning of the program (the first of the three additional lines before line 56):

```python
import matplotlib.gridspec as gridspec
```

Returning to line 57, the first two arguments of `GridSpec()` set up a 3×3 grid in the figure frame. You can set the relative widths of the columns and rows using the `width_ratios` and `height_ratios` keyword arguments. Then in line 59, we select grid rectangle number 4, which is at the center of a grid, by setting the first argument of `add_subplot` to `gs[4]`. The rectangles are numbered starting at zero from left to right and top to bottom. The other arguments fix the range of the x and y axes. The next few lines set up the plot of the number of active particles—the *activity*—as well as lists for the time `t` and a number of active particles `move`.

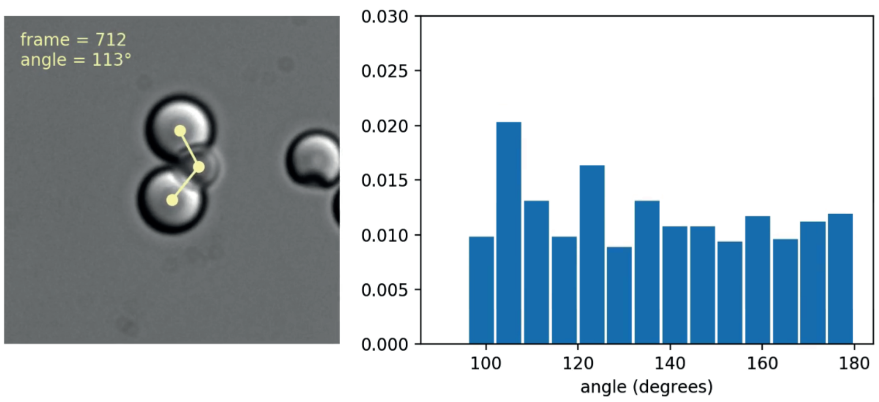


Figure 12.6 movie with animated histogram.

The only difference in the generator function `move` is that it returns two more variables than previously. Now, the final line reads:

```
yield t, moves, x, changes
```

We've inserted the current time `t` and the number of particles `moves` that moved in the most recent cycle. In the function `update`, the number `t` and the number `moves` are appended to the lists `tm` and `number_active` that keep a record of the number of active particles as a function of time. These data are then transmitted to the `activity` plot using the `set_data` function in line 78.

12.3 COMBINING VIDEOS WITH ANIMATED FUNCTIONS

When presenting or analyzing scientific data, it's often useful to display a video with an animated plot that highlights or analyzes the evolution of some features of the video. Consider, for example, the sequence of images we animated in Section 12.1.2. We want to show the movie and, next to it, display the evolving distribution of the angles swept out by the ball-and-socket joint. Figure 12.6 shows what we are aiming for.

12.3.1 Using a Single Animation Instance

Starting from the program on page 357, we modify line 22 by expanding the figure size and creating two `Axes` objects, one for the video and another for the plot of the distribution of angles. The distribution of angles is calculated using the NumPy histogram function on lines 42–43. Lines 44–45 set up the plot of the distribution, which is then appended to the list of Artists in line 46 that will be animated. The call to `ArtistAnimation` is the same as before.

Alternatively, we can make a histogram for the distribution of angles by commenting out lines 45–46 and uncommenting lines 48–49. Matplotlib's `bar` function returns a special bar container that needs to be turned into a list for incorporation in the list of lists for the animation, which is done in line 49.

Code: movie_from_images_histp.py

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3   import matplotlib.animation as anim
4   import pandas as pd
5   from PIL import Image
6   from glob import glob
7
8
9   def angle(x, y):
10      a = np.array([x[0] - x[1], y[0] - y[1]])
11      b = np.array([x[2] - x[1], y[2] - y[1]])
12      cs = np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))
13      if cs > 1.0:
14          cs = 1.0
15      elif cs < -1.0:
16          cs = -1.0
17      return np.rad2deg(np.arccos(cs))
18
19
20  r = pd.read_excel("trajectories.xlsx", usecols="A:F")
21
22  fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3.5))
23  ax1.axis("off")
24  ax2.set_xlim(90, 180)
25  ax2.set_ylim(0, 0.03)
26  ax2.set_xlabel("angle (degrees)")
27
28  ims = []
29  angles = []
30  for i, fname in enumerate(sorted(glob("pacb/s0*.png"))):
31      # print(fname)  # uncomment to follow loading of image frames
32      im = ax1.imshow(Image.open(fname), animated=True)  # image
33      x = np.array([r["x1"][i], r["xc"][i], r["x2"][i]])  # 3 balls
34      y = np.array([r["y1"][i], r["yc"][i], r["y2"][i]])  # joined by
35      ima, = ax1.plot(x, y, "o-", color=[1, 1, 0.7])  # 2 lines
36      theta = angle(x, y)
37      angles.append(theta)
38      imb = ax1.text(0.05, 0.95,
39                     "frame = {0:d}\nangle = {1:0.0f}\u00B0"
40                     .format(i, theta), va="top", ha="left",
41                     color=[1, 1, 0.7], transform=ax1.transAxes)
42      a, b = np.histogram(angles, bins=15, range=(90, 180),
43                          density=True)
44      xx = 0.5 * (b[:-1] + b[1:])
45      im2, = ax2.plot(xx, a, "-oC0")
46      ims.append([im, ima, imb, im2])
```

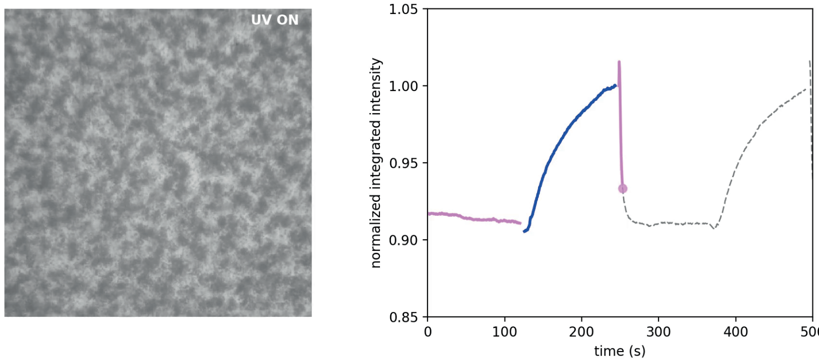


Figure 12.7 movie with animated plot.

```
    # plot histogram
    # im2 = ax2.bar(xx, a, width=0.9*(b[1]-b[0]), color="C0")
    # ims.append([im, ima, imb] + list(im2))
plt.tight_layout()

ani = anim.ArtistAnimation(fig, artists=ims, interval=33,
                           repeat=False, blit=False)
# Uncomment to save as mp4 movie file.  Need ffmpeg.
# ani.save("movies/movie_from_images_histp.mp4", writer="ffmpeg")
```

12.3.2 Combining Multiple Animation Instances

Let's look at another example of a movie combined with a dynamic plot. Our previous example showed how to render many plotting elements using a single instance of `ArtistAnimation`. While `ArtistAnimation` is the natural choice for animating sequences of images, `FuncAnimation` is the more natural choice for animating a dynamic plot. So, in this section, we use two different animation instances, one using `ArtistAnimation` to animate an image sequence and the other using `FuncAnimation` to animate the plot. We then show how to combine them into a single animation.

One frame of the result is shown in Figure 12.7. As the movie progresses, the illumination switches from ultraviolet (UV) to blue, designated on the image in the upper right corner and reflected in a change in color from violet to blue in the trace on the left.

After reading the sequence of frames to make the movie, the program reads data associated with each frame from a CSV file.

The static part of the plot, which is not animated, is rendered first. Lines 23–28 set up containers for the animated lines and circle, which change color according to whether UV or blue light is used to illuminate the image sequence.

Before plotting the data, the UV and blue data are masked in lines 31–32 so that only one of the two traces is displayed at any given time.

The `for` loop starting at line 36 puts together the list `ims` of the frames to be animated. Each frame is itself a list of the separate elements to be rendered in each frame: an image and the `UV ON`/`UV OFF` text. The `if-else` block increases the brightness of the frames when UV light is off so that all the movie frames have nearly the same brightness. The loop is completed when the list of plot elements is appended to the `ims`.

Next, the routine `animate` is defined, which is called by the `FuncAnimation` routine to animate the line plot.

`ArtistAnimation` is called to animate the movie frames, and `FuncAnimation` is called to animate the line plot. The second animation `ani2` is synchronized to the event clock of `ani1` using the keyword argument `event_source=ani1.event_source`. This assures that a single clock updates both animations.

Finally, to save both animations to the same file, we set the keyword argument, which takes a list (or tuple), `extra_anim=[ani1]` in the `ani2.save` call.

Code: movie_sync_plot1.py

```
 1  import numpy as np
 2  import matplotlib.pyplot as plt
 3  import matplotlib.animation as anim
 4  from PIL import Image, ImageEnhance
 5  from glob import glob
 6
 7  framesDir = "movie_from_frames"  # movie frames directory
 8  framesData = "movie_sync_data.csv"  # data file with intensities
 9  time, uv, blue = np.loadtxt(framesData, skiprows=1,
10                              unpack=True, delimiter=",")
11
12  # Static parts of plot come first
13  fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 4))
14  fig.subplots_adjust(bottom=0.15, top=0.95, left=0, right=0.98)
15  ax1.axis("off")
16  ax2.set_xlim([0, time.max()])
17  ax2.set_ylim([0.85, 1.05])
18  ax2.plot(time, uv + blue, dashes=(5, 2), color="gray", lw=1)
19  ax2.set_xlabel("time (s)")
20  ax2.set_ylabel("normalized integrated intensity")
21  ax2.set_yticks([0.85, 0.9, 0.95, 1., 1.05])
22  # Set up plot containers for ax2
23  plotdotUV, = ax2.plot(np.nan, np.nan, "o", color="violet",
24                        ms=6, alpha=0.7)
25  plotdotBlue, = ax2.plot(np.nan, np.nan, "o", color="blue",
26                          ms=6, alpha=0.7)
27  plotlineB, = ax2.plot(np.nan, np.nan, "-", color="blue", lw=2)
28  plotlineU, = ax2.plot(np.nan, np.nan, "-", color="violet", lw=2)
```

```python

# Mask data you do not want to plot
uvM = np.where(uv > 0.9, uv, np.nan)
blueM = np.where(blue > 0.9, blue, np.nan)

# Dynamic parts of plot come next
ims = []
imagelist = sorted(glob(framesDir + "/sp00*.png"))
for i, fname in enumerate(imagelist):
    # print(i, fname)  # uncomment to follow loading of image frames
    if uv[i] >= blue[i]:
        im = ax1.imshow(Image.open(fname), animated=True)
        textUV = ax1.text(320, 20, "UV ON", color="white",
                          weight="bold")
    else:
        img0 = Image.open(fname)
        # Increase brightness of uv-illuminated images
        img0 = ImageEnhance.Brightness(img0).enhance(2.5)
        im = ax1.imshow(img0, animated=True)
        textUV = ax1.text(320, 20, "UV OFF", color="yellow",
                          weight="bold")
    ims.append([im, textUV])


def animate(i):
    plotdotUV.set_data([time[i]], [uvM[i]])
    plotdotBlue.set_data([time[i]], [blueM[i]])
    plotlineB.set_data(time[0:i], blueM[0:i])
    plotlineU.set_data(time[0:i], uvM[0:i])
    return plotdotUV, plotdotBlue, plotlineB, plotlineU


ani1 = anim.ArtistAnimation(fig, artists=ims, interval=33,
                            repeat=False)
ani2 = anim.FuncAnimation(fig, func=animate,
                          frames=range(time.size), interval=33,
                          repeat=False, blit=False,
                          event_source=ani1.event_source)
# Uncomment to save as mp4 movie file.  Need ffmpeg.
ani2.save("movies/movie_sync_plot1.mp4", extra_anim=[ani1],
          writer="ffmpeg", dpi=200)
fig.show()
```

12.4 EXERCISES

1. Write a program to animate a 2-dimensional random walk for a fixed number of steps. Midway through the animation, your animation should look something like Figure 12.8(a).

   Start the random walk at $x = y = 0$. Show the leading edge of the random walk as a red circle and the rest as a line. Ensure the line extends from the starting point (0, 0) to the red circle at the end. The x and y axes should span the same distance in an equal-aspect-ratio square plot.

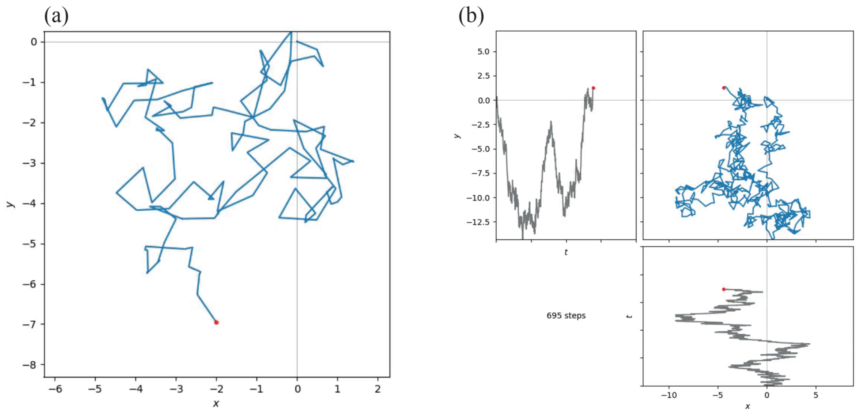


Figure 12.8 (a) Frame from Exercise 1. (b) Frame from Exercise 2.

The following code gets you started by creating a 2D random walk of $N = 200$ steps.

Code: diffusion.py

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as anim

N = 200
# Create arrays of random jump lengths in random directions
rng = np.random.default_rng()
dr = rng.random(N-1)   # random number 0-1
angle = 2.0 * np.pi * rng.random(N-1)
dx = dr * np.cos(angle)
dy = dr * np.sin(angle)
# Add up the random jumps to make a random walk
x = np.insert(np.cumsum(dx), 0, 0.)   # insert 0 as
y = np.insert(np.cumsum(dy), 0, 0.)   # first point
```

2. Embellish the animation of Exercise 1 by adding *x vs. t* and *y vs. t* panels along with animated text that gives the number of steps so far. Take *t* to be equal to the running number of steps. Midway through the animation, your animation should look something like Figure 12.8(b).

3. Rewrite the program that produces the animation associated with Figure 12.6 but use separate animation instances for the movie on the left side and the line (not a bar) histogram on the right.

4. Rewrite the program that produces the animation associated with Figure 12.7 but use a single animation instance for the movie on the left side and the animated plot on the right.

CHAPTER 13

Speeding Up Numerical Calculations

In this chapter, you learn how to speed up numerical calculations in Python using the Numba library with its just-in-time compiler. For numerical code involving long loops, Numba can speed up code execution by one or two orders of magnitude.

One of the significant advantages of an interpreted language like Python is that you can run code without going through the extra step of compiling, as required for languages like C and Fortran. Moreover, since Python is a dynamically typed language, you don't have to declare variables and set aside memory before using them. While these features make it easy to write code and quickly get it to run, they come with a price, most notably in execution speed.

The speed penalty incurred using Python is often inconsequential, and Python's convenience greatly outweighs the associated costs. For routine data analysis and plotting, for example, you usually get your results without noticeable delay, especially if you write code that takes advantage of NumPy's array processing capabilities…but not always.

For example, numerically simulating the temporal evolution of a physical system typically involves loops, because the solution at the current time step depends on the result from the previous step. As noted previously, loops execute very slowly in Python (see Section 6.2.4), so these dynamical simulations can be very slow. Indeed, any computation involving extensive use of loops of significant length, especially if there are nested loops, is likely to incur a substantial speed penalty.

DOI: 10.1201/9781032673950-13

An obvious way to deal with the problem of slow execution speed is to program using a compiled language, like C or Fortran. This is the right choice for demanding computational projects like molecular dynamics simulations, computational fluid dynamics, general relativity, computational chemistry, and many others. However, considerable overhead can be incurred, which may not be justified for projects of intermediate complexity.

Recently, a great deal of activity has focused on developing tools that can, for certain kinds of problems, significantly speed up numerical computations in Python. Some popular ones include CuPy, PyCUDA, Cython, f2py, and Numba. Of these, Numba is the easiest to use and the focus of this chapter. However, you should be aware of these other tools, which we summarize briefly here.

The first two tools, CuPy and PyCUDA, require the computer you are using to have one or more Nvidia® graphics processing units (GPUs) in addition to the standard central processing unit (CPU) that does the regular computational work of a computer. GPUs can significantly speed up programs involving image processing and similar computational tasks that can be performed in parallel. You should consider using them if you have the GPU hardware and computational tasks that can take advantage of their powerful capabilities.

The next two tools, Cython and f2py, allow you to incorporate compiled code, C or Fortran, respectively, into your Python programs. The idea is to write those parts of your programs that are computationally demanding in C or Fortran and then compile them into modules that Python can import. That way, the computationally demanding part of the code is performed by a compiled module that runs quickly, while the more routine tasks, like reading and saving data from and to files, changing the parameters of a calculation or simulation, and plotting data, are performed by standard Python code.

This brings us to Numba, which requires neither special hardware nor programming in another language. Thus, Numba is generally easier to implement and use than these other tools. Just as importantly, Numba can significantly increase the speed of numerical computations. The remainder of this chapter introduces the Python package Numba.

13.1 NUMBA'S BASIC FUNCTIONS

Numba is a Python package that uses a *just-in-time* (JIT) compiler to selectively compile parts of your Python code at runtime, that is, when you run your Python routines. The just-in-time compiler works automatically when you run your Python program, so no separate compilation step is required. Numba is part of the standard Anaconda distribution, so you don't need to download any additional packages to use it. It can speed up the execution of Python code, often by factors of 10 to 100 or more, if the code involves one or more of the following:

- loops (especially nested and long loops)
- a lot of math

Numba generally works well with NumPy; most NumPy functions are implemented in Numba.[1]

Numba is implemented using a standard Python tool called a *decorator*. A decorator is a Python object that modifies the behavior of a user-defined Python function. In this case, Numba uses a decorator to instruct Numba to compile a function and to use its compiled form whenever it is encountered in a Python program. This means that any numerical code you want to speed up must be coded within a Python function or a class method. To see how it works, let's consider a few examples.

13.1.1 Faster Loops and NumPy Functions

Numba excels at speeding up code with loops. Consider the Python function `gaussian_blur(image, pxblur)` below, which performs a Gaussian blur on an array `image` over a radius of `pxblur` pixels. Performing the Gaussian blur entails four nested `for` loops, with two of them looping over all the pixels of the image while the other two loop over the dimensions of the Gaussian blurring filter. For an image with a size of 1024 × 1024 pixels and a Gaussian blur over a radius of 8 pixels, nearly 300 million calculations are required, each entailing another loop iteration. Running this function without Numba takes about 72 seconds on a MacBook Pro M1 laptop. Figure 13.1 shows how it works on an image of random pixel values.

We invoke Numba for the function `gaussian_blur()` with a single statement, a decorator, on line 7: `@numba.jit(nopython=True)` (plus the `import numba` statement online 2). Decorators begin with the at sign @ and are placed just before the function definition whose behavior they modify. The decorator used here is defined within the Numba package. It tells Python to use the just-in-time compiler employed by Numba to compile the function starting on the next line and then to use the compiled version of the function whenever it is called within the Python program.

[1]A list of NumPy functions that have not yet been implemented in Numba is provided at https://github.com/numba/numba/issues/4074.

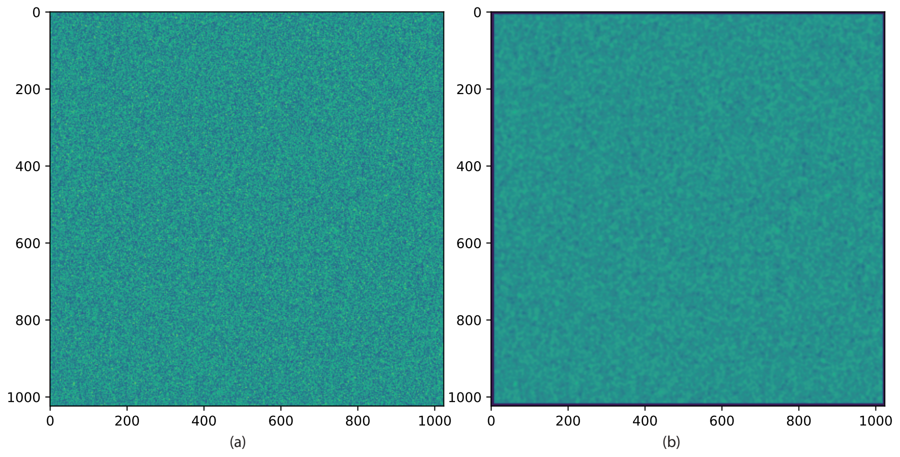


Figure 13.1 (a) Original image of random pixel values. (b) Blurred image of image on the left.

The program below runs the `gaussian_blur()` twice in a loop that starts on line 34. The first run takes 0.4322 seconds while the second takes 0.2561 seconds, representing speedups by factors of 166 and 273, respectively, over the 72-second time logged without Numba! Both times represent substantial improvements in performance.

Code: gaussian_blur.py

```
1  import numpy as np
2  import numba
3  import time
4  import matplotlib.pyplot as plt
5
6
7  @numba.jit(nopython=True)
8  def gaussian_blur(image, pxblur):
9      # make gaussian filter of radius pxblur
10     x = np.exp(-np.linspace(-2.0, 2.0, 2 * pxblur + 1) ** 2)
11     fltr = np.outer(x, x)
12     fltr /= fltr.mean() * fltr.size  # normalize filter
13     # Apply filter (adapted from Numba Read the Docs)
14     m, n = image.shape
15     mf, nf = fltr.shape
16     mf2 = mf // 2
17     nf2 = nf // 2
18     result = np.zeros(image.shape)
19     for i in range(mf2, m - mf2):
20         for j in range(nf2, n - nf2):
21             num = 0.0
22             for ii in range(mf):
23                 for jj in range(nf):
```

```
                    num += (fltr[mf - 1 - ii, nf - 1 - jj] *
                            image[i - mf2 + ii, j - nf2 + jj])
            result[i, j] = num
    # Return blurred image
    return result


# make image of random pixel values and set blur radius (pixels)
rng = np.random.default_rng()
image = rng.random((1024, 1024))
r_blur = 8

# Run calculation twice to determine jit compilation time
for i in range(2):
    start_pd = time.perf_counter()
    res = gaussian_blur(image, r_blur)
    end_pd = time.perf_counter()
    runtime = end_pd - start_pd
    print("\nrun time = {0:0.4g} seconds".format(runtime))

# Plot original image and Gaussian blurred image
fig, ax = plt.subplots(1, 2, figsize=(9.25, 4.6))
ax[0].imshow(image, vmin=0, vmax=image.max(), cmap="viridis")
ax[1].imshow(res, vmin=0, vmax=image.max(), cmap="viridis")
fig.subplots_adjust(top=1.0, bottom=0.02, left=0.05, right=0.99,
                    hspace=0.2, wspace=0.12)

fig.savefig("figures/gaussian_blur.pdf")
fig.show()
```

The time of 0.4322 seconds includes the time it took the just-in-time Numba compiler to compile the `gaussian_blur()` function when it was run the first time. Numba caches (saves) the compiled version and reuses it the second time it is called, so it runs faster. Evidently, it took the just-in-time compiler $0.4322 - 0.2561 = 0.1761$ seconds to compile the `gaussian_blur()` function. In this case, even if we include the compilation time, better performance is obtained using Numba. However, this is not always true, so it may not always make sense to use Numba. We will return to this question shortly.

13.1.1.1 NumPy Functions and Numba

Numba is designed to work with NumPy arrays and most NumPy functions. The function `gaussian_blur()`, introduced above and compiled with the Numba decorator `@numba.jit()`, used four NumPy functions: `np.exp()`, `np.linspace()`, `outer()`, and `np.zeros()`. It also used several NumPy attributes and methods, including `size`, `shape`, and `mean()`.

Most of the time, you can use NumPy functions with Numba just as you would without Numba. However, subtle differences can sometimes trip you up. For example, substituting `np.zeros([1024, 1024])` for `np.zeros` `(image.shape)` causes Numba to throw a `TypingError`, while `np.zeros((1024, 1024))` does not. All three of these calls work with NumPy. The problem here is that Numba requires that the shape of an array be strictly specified as a tuple and not as a list; NumPy is more relaxed and accepts both tuples and lists.

You can avoid getting such an error when using Numba by changing the decorator to `@numba.jit(nopython=False)`. This will run the Numba JIT compiler in *object mode* instead of in *nopython mode*. In object mode Numba identifies loops that it can compile and compiles them, but runs the rest of the code using the Python interpreter. Therefore, the code runs a bit slower than in nopython mode. When the JIT compiler falls back into object mode, Numba issues a warning. In general, we recommend running in nopython mode, as it runs faster. If Numba raises an error, correct the error and rerun in nopython mode. This is the best practice.

The Numba implementation of NumPy functions can differ in other ways. Consider the `circle()` function introduced on page 135. Let's see what happens when we try to compile it using Numba.

```
In[1]: import numpy as np
In[2]: import numba

In[3]: @numba.jit(nopython=True)
In[4]: def circle(r, x0=0.0, y0=0.0, n=12):
  ...:   theta = np.linspace(0., 2. * np.pi, n,
  ...:                       endpoint=False)
  ...:   x = r * np.cos(theta)
  ...:   y = r * np.sin(theta)
  ...:   return x0 + x, y0 + y

In[5]: circle(5.0)
Traceback (most recent call last):
...
TypingError: got an unexpected keyword argument 'endpoint'
...
```

The `circle()` routine runs perfectly well without Numba, but calling the Numba version throws an error. The error message tells us that Numba does not recognize the keyword argument `endpoint`. Apparently, it's not implemented in the Numba version of `linspace()`. We can create a workaround by eliminating the `endpoint` keyword, adding an element to the `theta` array, and then omitting the last point of the `theta` array so that the function returns the same arrays as it does without Numba.

```
In[6]: @numba.jit(nopython=True)
In[7]: def circle(r, x0=0.0, y0=0.0, n=12):
  ...:   theta = np.linspace(0., 2. * np.pi, n + 1)
```

```
   ...:     x = r * np.cos(theta[:-1])
   ...:     y = r * np.sin(theta[:-1])
   ...:     return x0 + x, y0 + y

In [8]: circle(5.0)
Out[8]:
(array([ 5.000,  4.330,  2.500,  0.000, -2.500, -4.330,
        -5.000, -4.330, -2.500,  0.000,  2.500,  4.330]),
 array([ 0.000,  2.500,  4.330,  5.000,  4.330,  2.500,
         0.000, -2.500, -4.330, -5.000, -4.330, -2.500]))
```

Thus, the Numba implementation of a NumPy function can differ from the NumPy implementation, which may require some changes in your code. Additionally, some of the less commonly used NumPy functions are not (yet) available in Numba (see Footnote 1 on p. 376.). When one of these is needed, you can often find a workaround. Sometimes, you can manually code the desired functionality without difficulty. In other cases, you can use the desired NumPy function without Numba but still use Numba for the rest of the code. This strategy is explored in Exercise 3.

The decorator `@numba.jit(nopython=True)` can also be written equivalently as `@numba.njit`.

13.1.1.2 Are NumPy Functions Compiled with Numba Faster?

Suppose you want to calculate the sine of a NumPy array. Will the calculation be faster using NumPy or Numba? Most NumPy functions are already vectorized and compiled so they run very fast. Nevertheless, NumPy functions compiled using Numba generally run somewhat faster than with NumPy alone, *if you don't include the compilation time*. If you include the compilation time, Numba can be substantially slower or somewhat faster, depending on the size of the NumPy array. The program below calculates the sine of an array of N elements, once using NumPy alone and twice using Numba.

Code: sine_numba.py

```
import numpy as np
import numba
import time


@numba.jit(nopython=True)
def trig(x):
    a = np.sin(x)
    return a


# Make NumPy arrayd with N elements
```

```python
N = int(100)
x = np.linspace(-1000.0, 1000.0, N)

# Compute sine of NumPy array using NumPy sine function
start_pd = time.perf_counter()
a = np.sin(x)
end_pd = time.perf_counter()
runtime = end_pd - start_pd
print("\nNumPy run time = {0:0.4g} secs".format(runtime))

# Compute sine of NumPy array using Numba
start_pd = time.perf_counter()
a = trig(x)
end_pd = time.perf_counter()
runtime = end_pd - start_pd
print("\nNumba 1st run time = {0:0.4g} secs (incl. compilation time)"
      .format(runtime))
# Compute sine of NumPy array using Numba a 2nd time
start_pd = time.perf_counter()
a = trig(x)
end_pd = time.perf_counter()
runtime = end_pd - start_pd
print("\nNumba 2nd run time = {0:0.4g} secs (w/o compilation time)"
      .format(runtime))
```

Here are the results obtained using $N = 100$:

```
NumPy run time = 1.097e-05 secs

Numba 1st run time = 0.07795 secs (incl. compilation time)

Numba 2nd run time = 4.806e-06 secs (w/o compilation time)
```

If the compilation time is included, the Numba calculation is $0.07795/1.097 \times 10^{-5} = 7106$ times slower than the NumPy calculation for this case. If the compilation time is not included, the Numba calculation is $1.097 \times 10^{-5}/4.806 \times 10^{-6} \times 10^{-6} = 2.3$ times faster than the NumPy calculation, a modest gain in speed.

What happens if we increase the size of N? For the program `sine_numba.py` above, Numba, including the compilation time, runs slower than simple NumPy for $N \lesssim 10^8$. Moreover, even for $N \sim 10^8$, the runtime is on the order of seconds or less. Most other NumPy functions yield similar results. Therefore, for routine computation with NumPy functions, using moderately sized arrays, using Numba is ill-advised. On the other hand, when long loops are used, as in `gaussian_blur.py` above, the compilation time is made up for by the gain in execution time of the loops, and any compiled NumPy functions will run at least as fast as they do in simple NumPy.

13.1.2 Vectorizing Functions with Numba

Numba can vectorize user-defined functions written to act only on scalars and thus make them work like NumPy ufuncs (see Section 7.1.2.2). Consider the following user-defined `sinc()` function (introduced in Section 7.1 and plotted in Figure 7.1):

Code: sinc0.py

```python
import math


def sinc(x):
    if x == 0.0:
        return 1.0
    else:
        return math.sin(x) / x
```

This function works on scalar inputs but not on arrays. It is manifestly not a ufunc as it uses the Python `math` library, which does not process arrays, to calculate the `sin()` function. Even if we rewrite the function so that it uses the NumPy library to calculate the `sin()` function, the rewritten `sinc()` function would still not work for NumPy arrays as the Python `if` statement cannot process arrays (see Section 7.1.1).

Nevertheless, Numba can vectorize this function using the decorator `@numba.vectorize()`.

Code: sinc_mn.py

```python
import math
import numba


@numba.vectorize(nopython=True)
def sinc(x):
    if x == 0.0:
        return 1.0
    else:
        return math.sin(x) / x
```

Running this program from the IPython console, we can then test it:

```
 In[9]: x = np.linspace(0.0, 32.0, 5)
 In[10]: x
Out[10]: array([ 0.,  8., 16., 24., 32.])
 In[11]: sinc(x)
Out[11]: array([ 1.     ,  0.1237, -0.018 , -0.0377,  0.0172])
```

The `sinc()` now acts like a NumPy ufunc.

We note that the `sinc()` function above cannot handle complex arguments because Python's `math` library does not handle complex arguments. However, if we substitute NumPy's `np.sin()` function, which accepts complex arguments, for `math.sin()`, the `sinc()` function handles complex arguments.

13.1.3 Numba Signatures

A `sinc()` function written with NumPy will accept either real numbers (floats) or complex numbers and will (try to) infer the output type. If you want more control over the input and output types, you can include a list of the desired types, called signatures, in the `@numba.vectorize()` decorator. Doing so can also reduce the compilation time.

Code: sinc_nn.py

```python
import numpy as np
import numba


@numba.vectorize([numba.float64(numba.float64),
                  numba.complex128(numba.complex128)],
                 nopython=True)
def sinc(x):
    if x == 0.0:
        return 1.0
    else:
        return np.sin(x) / x
```

This function will return an array of floats for an input array of floats and an array of complex numbers for an input array of complex numbers. This simpler data type, in this case, `numba.float64`, must appear in the list before the more complicated data type `numba.complex128`.

The scalar function can have multiple inputs and outputs. Here, for example, is a two-dimensional version of the `sinc()` function, which takes two inputs and produces one output.

Code: sinc_2dnn.py

```python
import math
import numba


@numba.vectorize([numba.float64(numba.float64, numba.float64)],
                 nopython=True)
def sinc(x, y):
    r = (x ** 2 + y ** 2) ** 0.5
    if r == 0.0:
        return 1.0
    else:
        return math.sin(r) / r


if __name__ == "__main__":
```

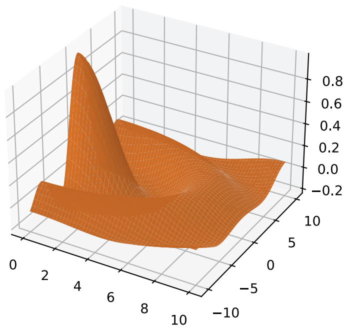


Figure 13.2 Plot of 2D `sinc()` function.

```python
import numpy as np
import matplotlib.pyplot as plt
n = 100
x = np.linspace(0.0, 10.0, n)
y = np.linspace(-10.0, 10.0, n)
f = sinc(x, y)

X, Y = np.meshgrid(x, y)
Z = sinc(X, Y)

fig, ax = plt.subplots(figsize=(5, 4),
                       subplot_kw={"projection": "3d"})
# Make surface plot
fig.subplots_adjust(left=0.0, bottom=0.08, right=0.96,
                    top=0.96, wspace=0.05)
p1 = ax.plot_surface(X, Y, Z, rcount=50, ccount=50, color="C1")

fig.savefig("figures/sinc_2dnn.pdf")
fig.show()
```

The list specifying the data types in the `@numba.vectorize()` decorator stipulates one output and two inputs. An important caveat is that the two input arrays must have the same dimension. The program also plots the two-dimensional function, which is shown in Figure 13.2.

13.1.3.1 *Signatures for Arrays*

The signatures specified in the example above are *scalars* because the original unjitted function acts on scalars. How do you specify signatures when the unjitted functions act on or return arrays? A 1D float array is specified as `numba.float64[:]`, a 2D float array as `numba.float64[:, :]`, *etc.* The same notation works for integer and other data types. For example, the following decorator applies signatures to the `gaussian_blur.py` program introduced in Section 13.1.1:

```
from numba import int64, float64


@numba.jit(float64[:, :](float64[:, :], int64), nopython=True)
```

In this case, introducing signatures reduces the compilation time so that it is completely negligible compared to the run time.

13.2 SIMULATIONS

Dynamical simulations model the behavior of physical objects in space and time according to an equation of motion or a set of equations of motion. They work by assuming the initial positions of the physical objects are known and then use the equations of motion to calculate where these objects will be a short time Δt later. Once the new positions are obtained, this process is repeated to obtain the particle positions at the next time, again a short time Δt later. This process is repeated to obtain the trajectories of the physical objects over some desired duration of time.

Because calculating what occurs in the next time step depends on what happened in the last time step, a dynamical simulation will necessarily involve a loop and, thus, is likely to be quite slow using Python. This is where Numba can help by speeding up the execution of all loops in the simulation.

To pique your interest, we report up front the acceleration in time obtained by using Numba for the simulation we introduce below. Without Numba, the simulation runs in 3 minutes and 15 seconds (195 seconds) using a MacBook Pro M1 laptop. Using Numba, the simulation runs in 5.3 seconds!

13.2.1 A Brownian Dynamics Simulation

How this works can best be understood using an example. To make things interesting, we consider the height $z(t)$ as a function of time for a micrometer-size particle—a colloidal sphere of radius R—suspended in water. We measure the height from the bottom of the water container.

The height $z(t)$ fluctuates due to random collisions with the water molecules. The impacts of these collisions are strong enough to keep the particle from settling to the bottom of the container under the influence of gravity. Thus, the particle executes a vertical random walk as the water molecules batter it about. The particle also moves horizontally from side to side, but we will not concern ourselves with its horizontal motion.

The particle is also subject to other forces, which we describe below. The simulation models the vertical movement of the particle as a function of time with a finite difference equation. The idea is to numerically solve the finite difference equation to determine the height of the particle z as a function of time.

The finite difference equation is:[2]

$$z(t+\Delta t) = z(t) + \frac{dD(z)}{dz}\Delta t + \frac{D(z)\,F(z)}{k_B T}\Delta t + Z_r(\Delta t)\,, \tag{13.1}$$

where $z(t)$ is the height of the particle at time t and $z(t+\Delta t)$ is its height a short time Δt later. $D(z)$ is a height-dependent diffusion coefficient,

$$D(z) = D_0 \lambda(x)\,, \tag{13.2}$$

where to a good approximation $\lambda(x)$ is given by

$$\lambda(x) = \frac{2x + 6x^2}{2 + 9x + 6x^2}\,, \tag{13.3}$$

where $x = z/R$. Note that $\lambda(x) \to 1$ for $x \gg 1$. Thus, far from the wall when $z \gg R$, $D \to D_0$. The diffusion constant $D_0 = k_B T/6\pi\eta R$, where η is the viscosity of the water, T is the absolute temperature in Kelvins, and $k_B = 1.380649 \times 10^{-23}$ J/K is Boltzmann's constant

The last two terms in Eq. (13.1) arise from different forces acting on the colloidal particle. $Z_r(\Delta t)$ is the random displacement of the particle in the z direction caused by collisions with the water molecules. It has a Gaussian distribution with a zero mean and a variance of

$$\langle Z_r^2 \rangle = 2D(z)\Delta t\,. \tag{13.4}$$

Two forces contribute to F, one due to gravity and another from an interaction between the particle and the flat horizontal bottom of the container. The gravitational force is given by

$$F_g = -\Delta m g\,, \tag{13.5}$$

where Δm is the sphere's mass less the mass of the liquid it displaces (Archimedes principle). The Morse potential models the interaction energy between the particle and the bottom of the container

$$U_M(z) = \epsilon\left[e^{-2(z-z_{min})/z_w} - 2e^{-(z-z_{min})/z_w}\right]\,, \tag{13.6}$$

[2]For more information on this model, see D. S. Sholl, M. K. Fenwick, E. Atman, & D. C. Prieve, *J. Chem. Phys.* **113**, 9268–9278 (2000).

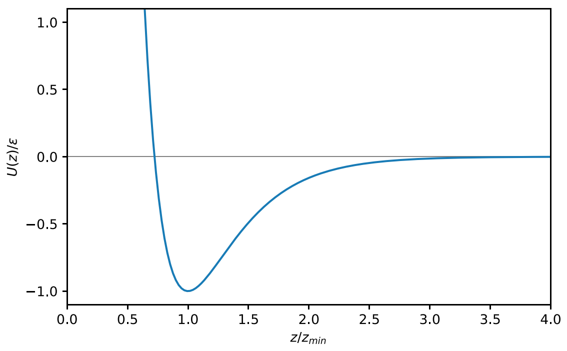


Figure 13.3 Plot of Morse potential.

which is plotted in Figure 13.3. The minimum of the potential well is located at $z = z_{min}$ and its depth is $-\epsilon$. The width of the potential well is z_w. The force on the particle due to the Morse potential is

$$F_M(z) = -\frac{dU_M(z)}{dz} = \frac{2\epsilon}{z_w}\left[e^{-2(z-z_{min})/z_w} - e^{-(z-z_{min})/z_w}\right] \tag{13.7}$$

This force is repulsive (it repels the particle from the bottom of the container) for $z < z_{min}$ and attractive for $z > z_{min}$. The force F is given by the sum $F_g + F_M$.

13.2.2 Nondimensional Simulation Variables and Parameters

When simulating a physical system, it's generally a good idea to transform the equations of motion into dimensionless form. Using the dimensionless form in a simulation keeps the numbers closer to unity and tends to avoid intermediate results during the computation that underflow or overflow the range covered by floating point numbers (recall from Section 2.3.2 that floating point numbers go between approximately $\pm 2 \times 10^{-308}$ and $\pm 2 \times 10^{308}$). An added benefit is that nondimensionalizing an equation usually reduces the number of parameters, as different parameters get consolidated.

Since our equation of motion Eq. (13.1) involves distance and time, the first step is to determine the characteristic distance and time scales of the problem. For the system we are considering, the characteristic length scale is set by the particle radius R, and the characteristic time scale is set by the time it takes a particle to diffuse its radius, which is R^2/D_0 (the SI units of D_0 are m^2/s). Thus, we define the dimensionless length as $x = z/R$ and the dimensionless time as $\tau = D_0 t/R^2$. Setting $z = Rx$ and $t = R^2\tau/D$, Eq. (13.1) becomes

$$Rx(\tau + \Delta\tau) = Rx(\tau) + \frac{dD(x)}{Rdx}\frac{R^2\Delta\tau}{D_0} + \frac{D(x)\,F(x)}{k_BT}\frac{R^2\Delta\tau}{D_0} + RX_r(\Delta\tau)\;.$$

Canceling common factors or R and D_0 gives

$$x(\tau + \Delta\tau) = x(\tau) + \frac{d\lambda(x)}{dx}\Delta\tau + \lambda(x)\frac{RF(x)}{k_BT}\Delta\tau + X_r(\Delta\tau)\;.$$

where $\lambda(x)$ is given by Eq. (13.3). This suggests that we define a dimensionless force $f = FR/k_BT$, with k_BT/R setting the characteristic force scale. Thus, we finally arrive at a nondimensional form of the equation of motion

$$x(\tau + \Delta\tau) = x(\tau) + \frac{d\lambda(x)}{dx}\Delta\tau + \lambda(x) f(x)\,\Delta\tau + X_r(\Delta\tau)\;. \tag{13.8}$$

Note that the parameters k_BT and D_0, which appeared in the dimensional equation of motion Eq. (13.1), do not appear in the dimensionless equation Eq. (13.8).

The nondimensional random displacement is defined by $X_r(\Delta\tau) = Z_r(\Delta t)/R$. The non-dimensional version of Eq. (13.4) is thus given by

$$\langle X_r^2\rangle = 2\lambda(x)\Delta\tau\;. \tag{13.9}$$

Finally, we need dimensionless forms of the forces given by Eqs. (13.5) and (13.7). The dimensionless gravitational force, which we call G, is obtained by dividing Δmg by the characteristic force k_BT/R introduced above, which gives

$$G = \Delta mgR/k_BT\;. \tag{13.10}$$

Similarly, the dimensionless force from the Morse potential is given by dividing Eq. (13.7) by k_BT/R, which gives

$$f_M(x) = \frac{R}{k_BT}\frac{2\epsilon}{z_w}\left[e^{-2(x-x_{min})/x_w} - e^{-(x-x_{min})/x_w}\right] \tag{13.11}$$

$$= \frac{2\epsilon_{nd}}{x_w}\left[e^{-2(x-x_{min})/x_w} - e^{-(x-x_{min})/x_w}\right]\;, \tag{13.12}$$

where the nondimensional Morse well depth is given by $\epsilon_{nd} = \epsilon/k_BT$.

13.2.3 Simulation with the Numba Decorator

The program below implements the simulation.

Code: bdsimf.py

```python
import numpy as np
import numba
from scipy.constants import Boltzmann as kB
import yaml


@numba.jit(nopython=True)
def run_simulation(sim_steps, x0):
    x = np.zeros(sim_steps, dtype=np.float64)
    x[0] = x0  # initial particle height
    for i in range(1, sim_steps):
        x[i] = x[i - 1] + dh(x[i - 1])
    return x


@numba.jit(nopython=True)
def dh(xi):
    """
    x(t+dt) - x(t)
    """
    # Calculate dimensionless D
    denom = 1.0 + xi * (4.5 + 3.0 * xi)
    D = xi * (1.0 + 3.0 * xi) / denom
    # Calculate dD/dx
    dDdx = (1.0 + xi * (6.0 + 10.5 * xi)) / denom ** 2
    # Calcuate random displacement
    xrand = np.sqrt(2.0 * D * dt) * np.random.standard_normal()
    # Return full displacement
    return (dDdx + D * F(xi)) * dt + xrand


@numba.jit(nopython=True)
def F(xi):
    """
    Force from potentials on sphere = -dU/dx - G
    """
    ex = np.exp(-(xi - x_min) / x_width)
    return (2.0 * pot_depth / x_width) * (ex * (ex - 1.0)) - G


def read_params(yaml_data_file_name):
    """Reads simulation parameters from yaml file"""
    # Open yaml file, read, and close
    fmeta = open(yaml_data_file_name, "r")
    meta = yaml.safe_load(fmeta)
    fmeta.close()
    # extract parameters read from yaml file
    temp_C = float(meta["temp_C"])
    temp_K = temp_C + 273.15
    kT = kB * temp_K  # energy scale
    diameter = float(meta["diameter"])
    radius = 0.5 * diameter  # length scale
    viscosity = float(meta["viscosity"])
    D0 = kT / (6.0 * np.pi * viscosity * radius)

    dt = float(meta["dt"]) * (D0 / radius ** 2)  # dimensionless time
    x_min = float(meta['z_min_nm']) * 1.0e-9 / radius  # ht of pot min
    x_width = float(meta['z_width_nm']) * 1.0e-9 / radius  # pot width
    pot_depth = float(meta['pot_depth_kT'])  # pot depth

    density_sphere = float(meta["density_sphere"])
    density_fluid = float(meta["density_fluid"])
    volume = (4.0 / 3.0) * np.pi * radius ** 3
    g = float(meta["g"])
    G = (density_sphere - density_fluid) * volume * g * (radius / kT)
    sim_steps = int(float(meta["sim_steps"]))
    return sim_steps, dt, x_min, x_width, pot_depth, G


if __name__ == "__main__":
    import time
    # Read in simultaion parameters from yaml file
    data_file_name = "sim_data01"
    sim_steps, dt, x_min, x_width, pot_depth, G = read_params(
        data_file_name + ".yaml")
    # Initialize particle height (x) array
    x_start = x_min

    # Do simulation: get height x as a function of time
    for i in range(2):
        start_pd = time.perf_counter()
        x = run_simulation(sim_steps, x_start)
        end_pd = time.perf_counter()
        runtime = end_pd - start_pd
        print("\nrun time = {0:0.4g} seconds".format(runtime))
```

13.2.3.1 *The Input Parameters*

As in previous cases, we specify the parameters the simulation needs using a YAML file:

Data: sim_data01.yaml

```yaml
---
z_min_nm: 10.0
z_width_nm: 5.0
pot_depth_kT: 5.0
sim_steps: 1e8
dt: 1e-3
diameter: 5e-6
density_sphere: 1055.0
density_fluid: 997.0
temp_C: 22.0
viscosity: 0.89e-3
g: 9.802
...
```

All the parameters in the YAML file are provided in SI units unless otherwise specified in the variable name. The file provides the parameters needed by the Morse potential, z_{min}, z_w, and the depth of the well ϵ in units of k_BT. The number of simulation steps can be provided as an integer or a float. Next, the YAML file provides the simulation step size Δt (in seconds), the sphere and fluid densities (in kg/m^3), the temperature (in °C), the fluid viscosity (in Pa-s=N-s/m^2), and the acceleration due to gravity g (in m/s^2).

Next, we describe what the four different functions defined in the program `bdsimf.py` do.

`read_params()`: This function reads the parameters from the YAML file and returns the five dimensionless parameters needed as input to the simulation. It has one argument, the base name of the YAML file, that is, the file name without the `.yaml` extension.

`run_simulation()`: This function executes the loop that calculates the values of the dimensionless height $x(t)$ at a sequence of times spaced by the dimensionless time interval $\Delta\tau$. Before starting the loop, it creates the NumPy array `x` that will store the simulation results. It then sets the starting position of the particle by setting the first element of the array `x[0] = x0`, where `x0` is the second argument of the function. While `x0` can be almost anything, when we call `run_simulation()`, we will set it to the minimum of the Morse potential well x_{min}. The first argument of `run_simulation()` is the number of simulation steps `sim_steps`, which is the number of time steps in the simulation and the size of the `x` array.

The `for` loop calculates the next value of `x` at a time increment of $\Delta\tau$ by calling the function `dh()`, which we describe next.

`dh()`: This function calculates the last three terms of Eq. (13.8). It has one *scalar* (not an array) argument, the current nondimensional particle position.

First, it calculates the nondimensional diffusion coefficient $D(x)/D_0 = \lambda(x)$, to which the program assigns the variable name `D` rather than `lambda`, as `D` is more expressive and all variables are understood to be nondimensional.

Next, it calculates the nondimensional derivative $(d\lambda(x)/dx)$ to which it assigns the variable name `dDdx`. Then it calculates the random force, the last term of Eq. (13.8) using NumPy's normalized random number generator for a Gaussian distribution and setting the amplitude according to Eq. (13.9).

Finally, in the `return` statement, it calculates the force $f(x)$ by calling another function `F()`, described below, and then pulls all the terms together to return the sum of the last three terms in Eq. (13.8).

`F()`: This function has one argument, the current particle position, and returns the forces due to the Morse potential and gravity.

Of the four functions, only the first three are compiled using the `@numba.jit(nopython=True)` decorator. The first three use only NumPy functions and thus can be compiled by Numba's JIT compiler. The first function contains the simulation loop, which is typically run 10^6 to 10^8 times. Therefore, it has the most to gain by being compiled. The fourth function calls YAML functions, which cannot be compiled, and will thus throw an error if run with the decorator `@numba.jit(nopython=True)`. Moreover, the fourth function is only run once so there is little speed to gain by compiling it.

13.2.4 Performance and Saving/Reading Large Data Files

To obtain good statistics on a simulation like this, about 10^6–10^8 steps are needed. As noted above, running the simulation without the decorators with 10^8 simulations steps takes about 190 seconds on a MacBook Pro M1. Running with the decorators takes about 5.3 seconds, representing a speedup by a factor of 40.

Note that we have used the `np.save()` and `np.load()` to save and load the 10^8 x values generated by this run. This produces an 800 MB `npy` *binary* data file. The save and load times are 0.16 and 0.44 seconds, respectively.

If instead we use the `np.savetxt()` and `np.loadtxt()` to save and load the 10^8 z values, a 2.5 GB *text* file is produced with save and load times of 78 and 19 seconds, respectively.

For large files, there are significant time and storage space penalties associated with writing and reading text files. The only disadvantage of using the `npy` binary file format is that the data file is not readable by humans or programs like Excel. On the other hand, it's hard to imagine wanting to visually inspect 10^8 numbers or manipulate them in an Excel file.

13.2.5 Isolating Numerical Code for Numba

Numba speeds up numerical code, particularly code involving loops and NumPy arrays and functions. It does not work with most functions from Pandas, SciPy, Matplotlib, or other packages. It does not speed up reading from or writing to data files. Therefore, to make optimal use of Numba, you should isolate the code that does number crunching, which Numba can speed up, from other parts of your program that read, write, plot, and otherwise manipulate data.

This is why the `read_params()` function in `dfsimf.py` is not compiled: it contains code that reads YAML files, code that Numba does not recognize.

Moreover, `read_params()` has no loops and is only run once, so there is little to gain by compiling the code.

13.3 USING NUMBA WITH CLASSES

The need to isolate numerical code for processing by Numba from other code suggests that it can make sense to encapsulate the numerical code by writing it into a Python class. Indeed, Numba provides the `@numba.experimental.jitclass()` decorator for classes. The module is still under development but it works fine for many purposes in its current form.

The program below runs the same simulation introduced in Section 13.2.3. The class decorator `@numba.experimental.jitclass(spec)` takes the name of a list as an argument, `spec` in this case. Each entry in the list is 2-element tuple that specifies is name and *Numba data type* of one of the fields initialized in the `__init__()` methods. All of the fields must be included in the list. Note that the one array in the list is specified using square brackets: `[:]`.

All of the class methods are compiled as nopython functions. In this case, the methods are reiterations of the same functions that were compiled in the simulation `bdsimf.py` in Section 13.2.3.

To run the simulation, the class is first instantiated with the appropriate dimensionless inputs. Then the `run_simulation(x0)` method is called with the initial dimensionless starting position `x0` of the particle as the argument. This simulation fills into the array `x` the values the dimensionless particle positions, which is not explicitly returned but instead is available as an instance variable through the dot syntax.

Code: bdsimc.py

```python
import numpy as np
from scipy.constants import Boltzmann as kB
import numba
import yaml


spec = [("sim_steps", numba.int32),
        ("dt", numba.float64),
        ("x_min", numba.float64),
        ("x_width", numba.float64),
        ("pot_depth", numba.float64),
        ("G", numba.float64),
        ("x", numba.float64[:])]


@numba.experimental.jitclass(spec)
class BDsim:
    """Simulates vertical motion of Brownian particle."""

    def __init__(self, sim_steps, dt, x_min, x_width, pot_depth, G):
        self.sim_steps = sim_steps
        self.dt = dt
        self.x_min = x_min
        self.x_width = x_width
        self.pot_depth = pot_depth
        self.G = G
        self.x = np.zeros(self.sim_steps)

    def run_simulation(self, x0):
        self.x[0] = x0  # initial particle height
        for i in range(1, self.sim_steps):
            self.x[i] = self.x[i - 1] + self.dh(self.x[i - 1])

    def dh(self, xi):
        """
        x(t+dt) - x(t)
        """
        # Calculate dimensionless D
        denom = 1.0 + xi * (4.5 + 3.0 * xi)
        D = xi * (1.0 + 3.0 * xi) / denom
        # Calculate dD/dx
        dDdx = (1.0 + xi * (6.0 + 10.5 * xi)) / denom ** 2
        # Calcuate random displacement
        xrand = np.sqrt(2.0 * D * self.dt) * \
            np.random.standard_normal()
        # Return full displacement
        return (dDdx + D * self.F(xi)) * self.dt + xrand

    def F(self, xi):
        """
        Force from potentials on sphere = -dU/dx - G
        """
        ex = np.exp(-(xi - self.x_min) / self.x_width)
        return (2.0 * self.pot_depth /
                self.x_width) * (ex * (ex - 1.0)) - self.G


class read_params:
    """Reads simulation parameters from yaml file"""

    def __init__(self, yaml_data_file_name):
        # Open yaml file, read, and close
        fmeta = open(yaml_data_file_name + ".yaml", "r")
        meta = yaml.safe_load(fmeta)
        fmeta.close()
        # extract parameters read from yaml file
        self.temp_C = float(meta["temp_C"])
        self.temp_K = self.temp_C + 273.15
        self.kT = kB * self.temp_K  # [J] energy scale
        self.diameter = float(meta["diameter"])
        self.radius = 0.5 * self.diameter  # length scale
        self.viscosity = float(meta["viscosity"])
        self.D0 = self.kT / (6.0 * np.pi *
                             self.viscosity * self.radius)
        self.dt = float(meta["dt"])  # [s] time increment
        self.z_min_nm = float(meta['z_min_nm'])  # height of pot min
        self.z_width_nm = float(meta['z_width_nm'])  # pot width
        self.pot_depth_kT = float(meta['pot_depth_kT'])  # pot depth
        self.sim_steps = int(float(meta["sim_steps"]))
        self.density_sphere = float(meta["density_sphere"])
        self.density_fluid = float(meta["density_fluid"])
        self.g = float(meta["g"])
        volume = (4.0 / 3.0) * np.pi * self.radius ** 3
        self.delta_mg = (self.density_sphere -
                         self.density_fluid) * volume * self.g
        # Form dimensionless parameters needed for simulation
        self.dt_nd = self.dt * self.D0 / self.radius ** 2
        self.x_min = self.z_min_nm * 1.0e-9 / self.radius
        self.x_width = self.z_width_nm * 1.0e-9 / self.radius
        self.G = self.delta_mg * (self.radius / self.kT)


if __name__ == "__main__":
    import matplotlib.pyplot as plt
    import time
    # Read in simultaion parameters from yaml file
    data_file_name = "sim_data01"
    param = read_params(data_file_name)
    # Initialize particle height (x) array
    x_start = param.x_min

    # Do simulation: get dimensionless height x as a function of time
    sim01 = BDsim(param.sim_steps, param.dt_nd, param.x_min,
                  param.x_width, param.pot_depth_kT, param.G)
    start_pd = time.perf_counter()
    sim01.run_simulation(x_start)
    end_pd = time.perf_counter()
    runtime = end_pd - start_pd
    print("\nrun time = {0:0.4g} seconds".format(runtime))

    # Save data to NumPy npy file
    start_pd = time.perf_counter()
    np.save(data_file_name + ".npy", sim01.x)
    end_pd = time.perf_counter()
    runtime = end_pd - start_pd
    print("\nsave npy time = {0:0.4g} seconds".format(runtime))
```

Once again, we specify the parameters the simulation needs using a YAML file. This time we use a class to read the YAML file, which has the advantage of making all the parameters available to the user through the dot syntax, including both original dimensional parameters and the dimensionless parameters needed for the simulation.

The `bdsimc.py` class-based simulation runs in 7.5 seconds on a MacBook Pro MI processor, a bit slower than the 5.3 time for the function-based `bdsimf.py` simulation. You can decide if the functionality provided by the class structure is worth this modest speed penalty. Both programs run in about 190 seconds without Numba.

13.4 OTHER FEATURES OF NUMBA

Numba has several other features that can speed up certain kinds of Python code. Instead of just-in-time compilation, Numba can compile Python functions ahead of time and cache (save) the result for later use. Numba can automatically parallelize code, simultaneously running independent parts of your code on multiple CPUs. Numba also supports CUDA GPU programming for systems with Nvidia GPUs. Each of these comes with unique constraints so you will need to read the documentation carefully and test your code to see if any of these tools can improve the performance of your code.

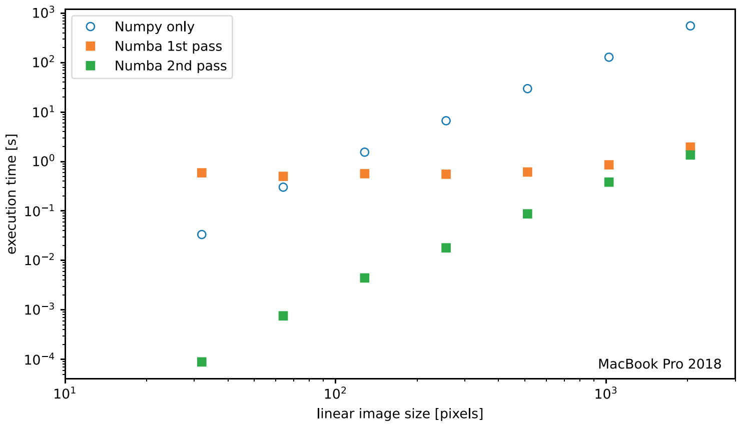
13.5 EXERCISES

1. (a) Make a table of the execution speed of the function `gaussian_blur()` for the following input image sizes both with and without Numba: 32×32, 64×64, 128×128, 256×256, 512×512, and 1024×1024. When using Numba, run `gaussian_blur()` twice and record the times for both the first and second runs (the second should be faster, as discussed in Section 13.1.1).

TABLE 13.1 Fill in this table.

	Execution time (seconds)						
image size ($h = w$)	32	64	128	256	512	1024	2048
NumPy only							
Numba (1st run)							
Numba (2nd run)							

   (b) Plot the execution times tabulated in part (a) *vs.* the linear image size. You should get a plot similar to the one below but with faster times, most likely. These data were obtained using a 2018 MacBook Pro laptop. From the results you obtained from your computer, what is the mean ratio of the execution time without Numba to the execution time with Numba, the second pass?



2. Monte Carlo integration is a versatile way of computing the value of a definite integral using random numbers. In this exercise, you will use it to get an estimate of π. The algorithm works like this. Choose n random points in the unit square defined by $0 \leq x \leq 1$ and $0 \leq y \leq 1$. Count the number m of those points with a radius $r \leq 1$. In the limit that n is very large, the ratio m/n should be equal to the area of one-quarter of a unit circle $A_c = \frac{1}{4}\pi$ divided by the area of the unit square, which is 1. Therefore $4m/n \to \pi$ as $n \to \infty$. Write a program to determine *pi* using Monte Carlo integration. Determine the time it takes to execute for N random points for $N = 10^4, 10^5, 10^6, 10^7, 10^8$, and 10^9 with and without Numba.

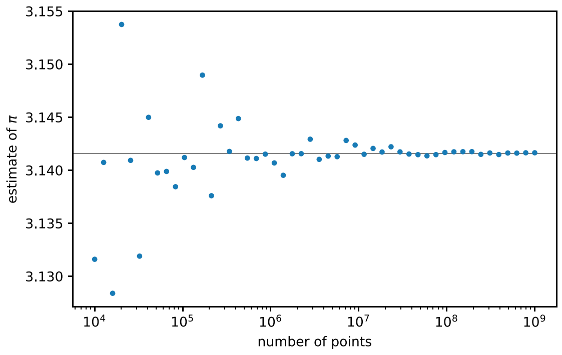


Figure 13.4 Estimates of π using Monte Carlo integration for different numbers of points. Gray line indicates the precise value of π.

3. The program `gaussian_blur.py` listed on page 377 calculates the Gaussian blur of an image over a radius of `pxblur` pixels. However, it leaves an edge around the perimeter of the blurred image `pxblur` pixels wide, which is visible as the dark strip around the perimeter of the blurred image in Figure 13.1.

   (a) With Numba off, modify `gaussian_blur.py` so that it blurs the input image to the edge. Do this by using the NumPy function `np.pad()` within the function `gaussian_blur()` to pad the image array with an extra `pxblur` pixels around the perimeter. Set the keyword argument `np.pad()` `mode="mean"` and calculate the Gaussian blur over the pixels of the *entire original image*. The function `gaussian_blur()` should return a blurred image the same size as the original array. In developing and testing your code, reduce the size of the random image generated by the program to 128 × 128 pixels so that the code runs quickly without using Numba. When finished, the output plot should show the original image blurred to the image edge.

   (b) Rerun the program you wrote in part (a), but this time using the decorator `@numba.jit(nopython=True)`. Unless Numba has been updated since the publication of this book to include the NumPy function `np.pad()`, you should get an error message.

   (c) Modify `gaussian_blur.py` so that it can take advantage of Numba's ability to speed up the execution of loops while still using NumPy's `np.pad()` function. You can do this by padding the original image before sending it to the `gaussian_blur()` function. For this to work, you must also alter `gaussian_blur()` so that it expects the padded image file. Nevertheless, the function should return the unpadded blurred image. You should be able to obtain an execution speed comparable to that obtained for the original version of `gaussian_blur.py` using Numba.

4. Starting from the program `bdsimc.py`, write code that plots a histogram of the height z (in nanometers). Your plot should use the z coordinates instead of the dimensionless x coordinates. The scaling of the z coordinate should be done automatically using the parameters from the `read_params` class.

   - Your plotting code will need to call the `BDsim` class but should not be included within the class as Numba does not complile Matplotlib routines.

- To calculate the histogram, use the NumPy routine `np.histogram()` rather than the Matplotlib routine `plt.hist()`, as you can write a class method that includes the NumPy version in the class `BDsim` and have it compiled by Numba. Numba will not compile the Matplotlib version.
- How much time do you save by running the histogram with Numba *vs.* running it outside the class without Numba? Is compiling `histogram()` justified?

5. Adapt the program `gaussian_blur.py` listed on page 377 to read in the grayscale PNG image below and blur it. Show that your program works on the following image using pixel blur radii of 4 and 8. Use `np.asarray(Image.open(image_filename).convert('L')` to properly read in the grayscale image (`image_filename` is the image filename, including the path). After opening the image, you will need to convert it to a NumPy array. Compare the run times with and without Numba for a blur radius of 8. Bonus: See Exercise 3 if, for aesthetic reasons, you want to eliminate the black boarder made using `gaussian_blur.py` routine.

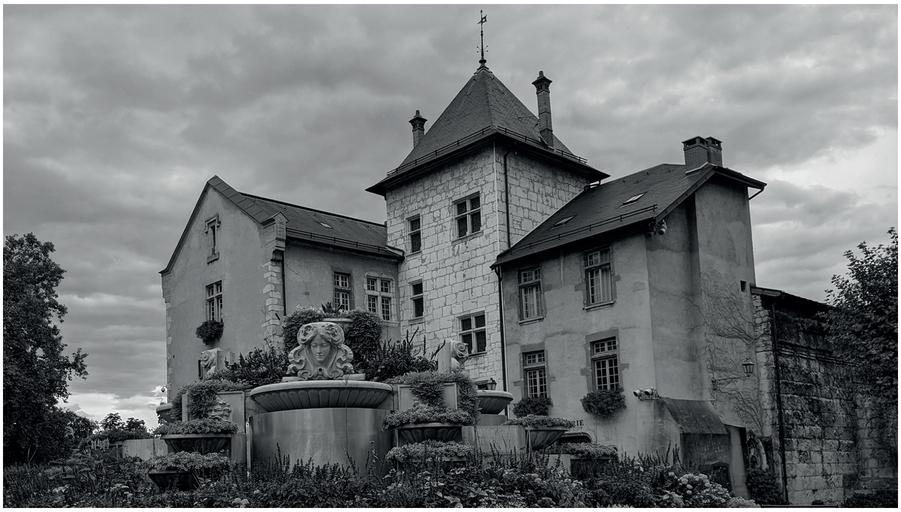


Figure 13.5 Original grayscale image without blurring for Problem 5.

APPENDIX A

Maintaining Your Python Installation

You need to install Python and four scientific Python libraries for scientific programming with Python: NumPy, SciPy, Matplotlib, and Pandas. You can install many other useful libraries, but these four are the most widely used and are the only ones you will need for this text. We recommend using the *Anaconda* distribution for your work in this text. See Section 1.2 for instructions about how to do this.

A.1 UPDATING PYTHON

Python and its many packages are regularly updated, including NumPy, SciPy, Matplotlib, and Pandas. Updating your installation periodically is generally a good idea. With the *Anaconda* distribution, you can do this simply by opening the terminal application on your computer (see Section 2.1), typing `conda update conda` at the terminal prompt, and pressing RETURN. Answer `[y]es` to the question about installing updates, and you're all set. You should do this every month or so.

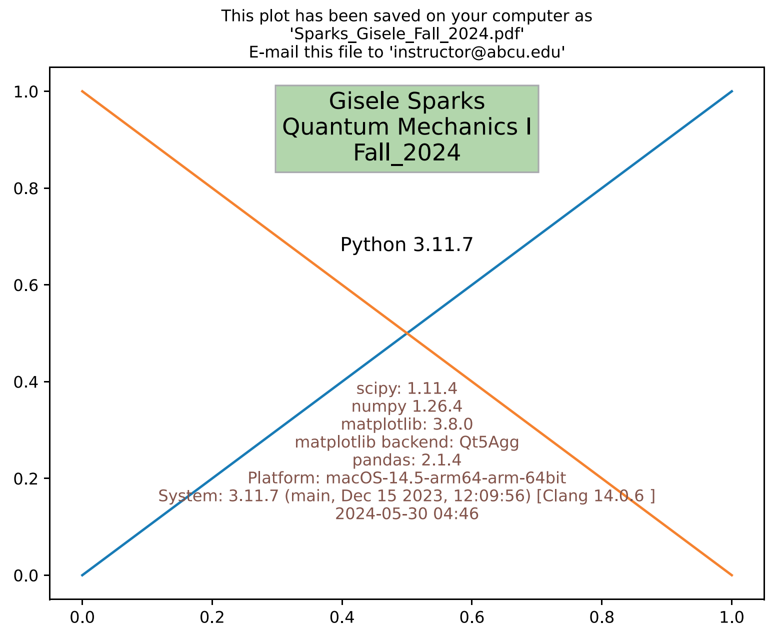
A.2 TESTING YOUR PYTHON INSTALLATION

Running the program testInsatllation.py below tests your installation of Python and records information about the installed versions of various packages used in this manual. If you are a student, you should input your first and last names inside the single quotes on lines 10 and 11, respectively. Instructors should modify the course information inside the double quotes in lines 15–17.

DOI: 10.1201/9781032673950-A

Code: testInstallation.py

```python
""" Checks Python installation and generates a pdf image file that
    reports the versions of Python and selected installed packages.
    Students can sent output file to instructor."""
import scipy, numpy, matplotlib, pandas, datetime, platform, sys
import matplotlib.pyplot as plt

# If you are a student, please fill in your first and last names
# inside the single quotes in the two lines below. You do not need to
# modify anything else in this file.
student_first_name = 'Gisele'
student_last_name = 'Sparks'
# If you are an instructor, modify the text between the double quotes
# on the next 3 lines. You do not need to  modify anything else in
# this file.
classname = "Quantum Mechanics I"
term = "Fall_2024"  # must contain no spaces
email = "instructor@abcu.edu"
timestamp = datetime.datetime.now()
tm = "{0:s}".format(timestamp.strftime("%Y-%m-%d %H:%M"))
plt.figure(figsize=(8, 6))
plt.plot([0, 1], "C0", [1, 0], 'C1')
plt.text(0.5, 1.0, "{0:s} {1:s}\n{2:s}\n{3:s}"
         .format(student_first_name, student_last_name, classname,
                 term),
         ha="center", va="top", size='x-large',
         bbox=dict(facecolor="C2", alpha=0.4))
```



```
plt.text(0.5, 0.7, "Python {0:s}".format(platform.python_version()),
         ha="center", va="top", size="large")
pkgstr = "scipy: {0:s}\nnumpy {1:s}\nmatplotlib: {2:s}"
pkgstr += "\nmatplotlib backend: {3:s}\npandas: {4:s}"
pkgstr += "\nPlatform: {5:s}\nSystem: {6:s}"
pkgstr += "\n{7:s}"
plt.text(0.5, 0.4, pkgstr.format(scipy.__version__,
         numpy.__version__, matplotlib.__version__,
         matplotlib.get_backend(), pandas.__version__,
         platform.platform(), sys.version, tm),
         ha="center", va="top", color="C5")
filename = student_last_name + "_" + student_first_name
filename += "_" + term + ".pdf"
ttlstr = "This plot has been saved on your computer as"
ttlstr += "\n'{0:s}'\nE-mail this file to '{1:s}'"
plt.title(ttlstr.format(filename, email), fontsize=10)
plt.savefig(filename)
plt.show()
```

A.3 INSTALLING FFMPEG FOR SAVING ANIMATIONS

You must install additional software to record animations in an independent movie file. We suggest using FFmpeg, which works nicely with Matplotlib.

Installing FFmpeg is a fairly simple matter. Go to your computer's terminal application (Terminal on a Macs and Linux or Anaconda Powershell Prompt on a PC) and type:

```
conda install -c anaconda ffmpeg
```

Respond [y]es when asked to proceed. The Anaconda utility `conda` will install several new packages and probably update others. When finished, FFmpeg should work with the Matplotlib animation programs discussed in Chapter 12.

A.4 ADDING FOLDERS/DIRECTORIES TO YOUR PYTHON PATH

For a given Python session, you can add directories to PYTHONPATH from the IPython command line. Suppose, for example, you want to add the `/Users/dp/mypy` to your PYTHONPATH. From the Terminal, you can simply type:

```
In[1]: import sys

In[2]: sys.path.append("/Users/dp/mypy")
```

That does the trick. The only problem is that it's only good for the current session. If you relaunch the IPython kernel, `/Users/dp/mypy` will not be in your PYTHONPATH. If you want to add a directory permanently to your PYTHONPATH, keep reading.

How you permanently add directories (folders) to your PYTHONPATH depends on which operating system you are using and on which application you use for running Python.

A.4.1 Spyder

If you use Spyder, the directories can be added to PYTHONPATH from the "Tools>
PYTHONPATH manager" menu. This works for all operating systems but only sets PYTHONPATH for the Spyder application.

For all other applications, how you permanently add directories to your PYTHONPATH depends on which operating system you are using. The instructions below show you how to do it for macOS, Windows, and Linux. Once this is done, the directories you add are, by default, a part of PYTHONPATH for all applications except Spyder.

A.4.2 macOS

If you are using a Mac operating under **macOS** 10.15 (Catalina, introduced in October 2019) *or later*, then you need to edit a file called `.zshrc`. If you are using an earlier macOS, you need to edit a file called `.bashrc`. Installing the Anaconda distribution should have created it if it didn't already exist. You can find it in your home directory (the one you go to in the **Terminal** app when you type `cd ~`). For me, it's in `/Users/dp`; for your Mac, it will be in `/Users/XXX`, where `XXX` is the name of your home directory. Launch the text editor **BBEdit**. Use the `File:Open...` menu to open the file and make sure the `Show hidden items` box under the `Options` button is ticked. Then you should see `.zshrc` (or `.bashrc`) in the list of files. Open it. Then, on a new line type

```
# add directory to PYTHONPATH for my modules and repositories
export PYTHONPATH="${PYTHONPATH}:/Users/dp/mypy"
```

where you replace `/Users/dp/mypy` with `/Users/XXX/mypy`. Add a comment if you wish (always a good idea). Notice that we added a comment before the line adding the new directory to PYTHONPATH. Comment lines begin with `#`, as in a Python file. Save your edits and exit.

You can check that your directory has been added to PYTHONPATH by opening the **Terminal** application and, at the prompt, typing

```
        echo $PYTHONPATH
```

It should return something like `/Users/dp/mypy`.

You will need to relaunch Python (*e.g.*, QtConsole or Jupyter Lab) for the change in PYTHONPATH to take effect.

A.4.3 Windows

If you are using a Windows PC, you must set a new environment variable. This is how to do it.

1. Bring up the **System** menu by pressing WIN + I
2. Scroll to the bottom of the page and press **About**
3. Find "Device Specifications" and press the link "Advanced System Settings."
4. In the "System Properties" dialogue box, press the "Environmental Variables" button.
5. Under "User Variables…", press the **New…** button.
6. A **New User Variable menu** should come up.
7. Under **Variable name**, PYTHONPATH.
8. Be sure to use all capital letters.
9. Then press the button **Browse Directory**.
10. In the Browse For Folder menu that appears, navigate to `mypy` (or whichever directory you want to add to PYTHONPATH) and select it.
11. The directory, together with its full path, should get copied into the **Variable value**. In my case, it is `C:\Users\dp\mypy`. Then press **OK**.
12. In the **Environment Variables**, you should see a new Variable PYTHONPATH with a value of the path to the directory you added; in my case, this is `C:\Users\dp\mypy`.
13. Press OK at the bottom of the **Environment Variables** menu and then OK at the bottom of the **System Properties** menu.

You will need to relaunch Python for the change in PYTHONPATH to take effect.

A.4.4 Linux

If you are using Linux, then the procedure is largely the same as it is for the Mac. Most Linux installations use the Bash shell; some may have migrated to the Z shell. Depending on whether your computer uses the Bash or Z shell, open up the file `.bashrc` or `.zshrc` and add the lines

```
# add directory to PYTHONPATH for my modules and repositories
export PYTHONPATH="${PYTHONPATH}:/home/dp/mypy"
```

where you replace `/home/dp/mypy` with `/home/XXX/mypy`, where `XXX` is the name of your home directory.

You can check that your directory has been added to PYTHONPATH by opening the **Terminal** application and, at the prompt, typing

```
        echo $PYTHONPATH
```

It should return something like `/home/dp/mypy`.

You will need to relaunch Python (*e.g.*, QtConsole, Jupyter Lab, or Spyder) for the change in PYTHONPATH to take effect.

Glossary

Many terms introduced in the text are defined below for your convenience. The page number where the term is first used or defined is provided in parentheses.

Artist (202) Artists in Matplotlib are the routines that define the objects that are drawn in a plot: lines, circles, rectangles, axes, data points, legends, *etc.* The artist layer consists of the hierarchy of Python objects (or classes) that facilitate creating a figure and embellishing it with any of the features listed above.

Attributes (77) The methods and instance variables of an object.

Backend (200) A Matplotlib backend translates plotting code into useful output. There are two types: hardcopy backends and interactive (alternatively called interface) backends. A hardcopy backend translates Matplotlib code into image files such as PDF, PNG, PS, or SVG. An interactive backend translates Matplotlib code into instructions your computer screen can understand. It does this using third-party cross-platform software, usually written in C or C++, that can produce instructions that are sent to your computer screen. The net result is Matplotlib code that is platform-independent, working equally well under Windows, macOS, and Linux.

Blitting (361) Blitting most generally refers to the transfer of a block of pixel data from memory to your computer screen. However, in the context of animations, it refers to updating only those regions of the screen that change from one animation frame to the next. For example, in an animated plot, such as the one displayed in Fig. 12.3, blitting means only

DOI: 10.1201/9781032673950-B

updating the plot itself since it is the only thing changing. The axes' labels and tick marks are not redrawn since they are not changing. Not redrawing static features like the axes, axes labels, and ticks can dramatically speed up an animation.

Dynamically typed language (1) In a *statically typed language*, variable names are declared to be of a certain type—say a floating point number *or* an integer—before they are ever used. A variable's type is fixed at the beginning of a program and cannot be changed during the execution of the program. In a *dynamically typed language*, variable names are not generally declared to be of a certain type. Instead, their type is determined on the fly as the program runs. Moreover, the type can change during the execution of the program.

Instance variables (77) Data stored with an object that can be accessed by appending a period (.) followed by the name of the instance variable (without parentheses).

Instantiate (285) Create a new *instance* (realization) of a class by calling the class and providing the arguments required by the class constructor (`__init__` method).

Method (76) Function associated with an object that acts on the object or its attributes. A method is invoked by appending a period (.) followed by the method's name and an open-close set of parentheses, which can but need not take an argument.

Object (2) In object-oriented programming, an object is generally thought of as a collection of data together with methods that act on the data and instance variables that characterize various aspects of the data.

Object-oriented programming (OOP) (2) Object-oriented programming refers to programming based on the concept of objects that interact with each other in a modular manner. Scientific programming is typically procedural. However, object-oriented design becomes increasingly useful as the size and complexity of a scientific programming task increases.

Universal function or **ufunc** (133) A function that operates on NumPy arrays (ndarrays) in an element-by-element fashion. Such a function is said to be "vectorized." A NumPy ufunc also respects specific rules about handling arrays of different sizes and shapes.

Vectorized code (201) Computer code that processes *vectors* (ndarrays in NumPy) as the basic unit rather than individual data elements.

Wrapper (19) A Python program that provides a Python interface to a program written in another language, usually C, C++, or Fortran.

Python Resources

This text introduces Python for science and engineering applications but is hardly exhaustive. There are many other resources that you will want to tap. Here we point out several that you may find useful.

C.1 PYTHON PROGRAMS AND DATA FILES INTRODUCED IN THIS TEXT

Throughout the text, various Python programs and data files are introduced. All are freely available at https://github.com/djpine/python-scieng-public.

C.2 WEB RESOURCES

The best web resource for Python is a good search engine like Google. Nevertheless, I list a few websites here that you might find useful.

Python home page: https://www.python.org/.
The official Python website. I almost never look here.

Python 3 language reference: https://docs.python.org/3/reference/.
I look here sometimes for detailed information about Python 3, which is the version used in this text.

NumPy Reference: https://docs.scipy.org/doc/numpy/reference/.
I usually start here when I need information about NumPy. It has links to just about all the NumPy documentation I need. By the way, I say "num-pee," which rhymes with "bumpy" —a lot of people say "num-pie," which doesn't sound like English to me.

DOI: 10.1201/9781032673950-C

SciPy Reference: https://docs.scipy.org/doc/scipy/reference/.
I start here when I need information about SciPy, its various packages and their functions. I say "psy-pi" for SciPy, like everyone else. Who says I have to be consistent? (See Emerson.)

Pyplot: https://matplotlib.org/stable/api/pyplot_summary.html.
The *Plotting Commands Summary* page for Matplotlib. You can go to the main Matplotlib page, http://matplotlib.org/, but frankly, it's less useful.

Stack Overflow: https://stackoverflow.com/questions/.
stack**overflow** is a set of websites that lets people pose and answer questions related to computer programming in every computer language imaginable. Nearly every question or problem you can think of has already been asked and answered. If it isn't and you need to ask a question, pose it as precisely as possible and find the solution to your problem. Take advantage of this valuable resource.

Pandas: http://pandas.pydata.org/pandas-docs/stable/10min.html.
There is probably some useful information here, but frankly, I recommend stack**overflow** for nearly all your questions about Pandas: https://stackoverflow.com/tags/pandas/info.

Jupyter Lab: https://jupyterlab.readthedocs.io/en/stable/.
I go to this page to learn about Jupyter Lab.

Anaconda/Continuum: https://www.anaconda.com/.
Get virtually all the Python packages you want here from Continuum Analytics. Anaconda is available on all platforms: Macs, PCs, and Linux machines.

Mailing lists: Some software packages have mailing lists to which you can subscribe or pose questions about a specific package. They give you access to a community of developers and users that can often provide expert help. Remember to be polite and respectful of those helping you and those posting questions. The URL for the SciPy mailing list is http://www.scipy.org/Mailing_Lists/. The URL for the Matplotlib mailing list is https://lists.sourceforge.net/lists/listinfo/matplotlib-users/.

C.3 BOOKS

There are a lot of books on Python and there is no way I can provide reviews for all of them. The book by Mark Lutz, *Learning Python*, published by

O'Reilly Media, provides a fairly comprehensive, if verbose, introduction for non-scientific Python.

For Pandas, the book by its originator, Wes McKinney, called *Python for Data Analysis* provides a thorough but terse treatment and slanted toward applications in finance. Some well-written tutorials on the web cover many aspects of Pandas, but it takes some digging to find useful ones.

Index

addition operator (+), 7
adjust space around plots, *see* subplots_adjust
animation, 351
  annotating animations, 355
  blitting, 361
  combining multiple animation instances, 370
  combining videos with animated functions, 368
  dynamic text, 355
  fixed number of frames, 358
  function animation, 357
  function call, 361
  histogram, 369
  indeterminate number of frames, 362
  save to file, *see* FFmpeg
  sequence of images, 351
  static elements, 359
  until a condition is met, 362
anonymous functions, *see* lambda expressions
append, 121, 352, 355, 357
argument, *see* function
arithmetic, 7
  order of operations, 6
array (NumPy), 61
  multidimensional, 69
    indexing, 71
  array operations, 72
  attributes, 77, 148
  Boolean indexing, 67
  broadcasting, 73
  conditional (where), 131
  creating, 61, 69
  differences with lists, 75
  indexing, 66, 71
    b[i,j] vs b[i][j], 71
  instance variables, 148
  masked, 178, 187
  mathematical operations, 63
  matrices, 69
    matrix operations, 72
  methods, 77, 148
  printing, 89
  reshaping, 70
  slicing, 66
assert statement, 124
assignment operator (=), 10
attribute, *see* object
axes
  manual ticks
    labels, 198
    placement, 198

backend, 200
binary arithmetic operations, 7
blitting, *see* animation
Boolean operators, 107
break, *see* loop control statements
broadcasting, *see* array (NumPy)

case sensitive
  commands, 6, 26
  variable names, 14
class, 285, 297
  attributes, 284

__dict__ method, 296
methods, 284
comma after assigned variable, 360
comma after variable name, 360
comma-separated value file, *see* CSV file
comments, 17
comment character #, 17
comparisons, 107
complex, *see* numeric types
conditional statements, 102
where function, if-else for NumPy arrays, 131
if, 106
if-elif-else, 102
if-else, 105
configure subplots icon, 183
continue, *see* loop control statements
CSV file, 92
curve fitting, 236
linear, 150
with weighting (χ^2), 153

DataFrame, 320
creating, 313
index attribute, 327
dates and times, 309
datetime
replace, 342
decorator, 375
dictionaries, 59
differential equations, *see* ODEs
discrete (Fast) Fourier transforms, *see* FFT
division operators
floor division (//), 7
normal division (/), 7
docstring, 17
documentation
books, 410
help, 25
online, 409
dynamical typing, 13

encapsulation, 284
enumerate, 118
exceptions, 112, 122
handling, 122
exponentiation operator (**), 7

FFmpeg, 354, 355
installing, 402
FFT, 273
figure
navigation toolbar, 164
figure size
figsize, 185
file names
spaces, 17
floating point, *see* numeric types
Floating point numbers, 8
function, 127
anonymous functions, *see* lambda expressions
argument
passing function names, 137
variable number (*args, **kwargs), 136
derivative (deriv), 138
keyword argument (kwarg), 95, 134
namespace, 140
passing (immutable) strings, variables, and tuples, 143
passing (mutable) lists and arrays, 143
passing function names and parameters, 137
passing mutable & immutable objects, 146

passing mutable and immutable objects, 144
positional argument, 134
universal function (ufunc), 133
unnamed arguments
**kwargs, 139
*args, 137
user defined, 128

garbage collection, 11
generator function, 362, 365
glob, 353
graphical user interface, *see* GUI
Greek letters in plots, *see* LaTeX in plots
GUI, 284

id function, 10
IDEs
JupyterLab, 39
if __name__ == __main__:, 304
imageopenImage.open(), 354
importing modules, 24
Matplotlib, 24
NumPy, 20
Pandas, 306
SciPy, 243
imshow(), 354
indentation, 105
input function, 83, 85
type casting, 85
installing Python, 3
instance variable, *see* object
instantiate, 285
integers, *see* numeric types
integrals, *see* numerical integration
Integrated Development Environments, *see* IDEs
interactive mode, plt.ion(), plt.ioff(), 164, 360

IPython, 5, 6
JupyterLab, 39
launching shell from Terminal, 5
magic commands, 26, 27
navigation commands, 27
Qt Console, 5
Spyder, 35
tab completion, 29
IPython pane, 6
iterable sequence, 117

JupyterLab IDE, 39

keyboard input, 83, *see also* input function
keyword argument, *see* function
kwarg, *see* function

lambda expressions, 146
set_printoptions, 90
LaTeX in plots, 193
fonts, 195
Greek letters, 196
math symbols, 197
non-italicized text in math mode, 195
least squares fitting
linear, 150
with weighting (χ^2), 153
legend, 167, 169
linear algebra, 251
eigenvalue problems, 253
banded matrices, 254
Hermitian matrices, 254
matrix determinant, 251
matrix inverse, 251
matrix multiplication, 251
systems of equations, 251
linear equations, 251

list comprehensions, 120

lists, 53
 adding in place, 57
 appending, 57
 differences with NumPy arrays, 75
 multidimensional, 56
 slicing, 55
 zero-indexed, 54
literals, 13
logarithmic plots, 180
 `set_xscale`, 183
 `set_yscale`, 183, 186
 log-log, 183
 semi-log, 181, 186
logical operators, 107
loop control statements, 118
 `break`, 118
 `continue`, 119
loops
 array operations, 119
 `for`, 113
  `enumerate`, 118
  slow execution, 120
 `while`, 110
  infinite loop, 112, 113

masked arrays, 178, 187
masking data, 215
math text in plots, *see* LaTeX in plots
Matplotlib, 20
 artist layer, 202
 backend, 200
  Qt5Agg, 201
 PyPlot scripting layer, 204
  backends, 204
  state machine, 205
 software layers, 200
matrix operations, 72
 dot, cross, and outer products, 73

method, *see* object
modules, 19
 creating, 290
 importing, 20
multiplication operator (∗), 7

namespace, 21, 140
nonlinear equations, 255
 bisection, 258
 Brent method, 255
 Newton-Raphson, 258
 Ridder, 258
 systems, 258
Numba
 classes, 393
 isolating numerical code, 392
 `@jit`, 376
 `@njit`, 380
 `@vectorize`, 382
 nopython mode *vs.* object mode, 379
 signatures, 383
numeric types, 7
 complex, 9
 floating point, 8
 integers, 7
numerical integration, 258
 single, 259
 double, 263
 double integrals, 263
 methods, 258
 numerical data, 264
 polynomials, 262
 single integrals, 259
NumPy, 19, 22
 array, *see* array (NumPy), *see* array (NumPy)
 functions, 22

object, 9, 148

attribute, 77
attributes, 77, 148
instance variable, 77
instance variables, 148
methods, 76, 148, 285
object-oriented programming, 2, 76, 284
objects, 76
ODEs, 265
solve_ivp, 265
OOP, *see* object-oriented programming
operator overloading, *see* overloading
order of arithmatic operations, 6
overloading
operator, 52

packages
creating, 290
Pandas, 20, 306
agg method, 344
apply method, 342
axis, 342
Boolean indexing, 316
conditional indexing, 316, 329
data from web, 326
DataFrame
indexing, 315
indexing with iloc, 315
indexing with loc, 316
dates and times, 309
dt.total_seconds, 343
dtypes, 314
groupby method, 337
groupby object, 338
iloc method, 315
loc method, 316
pd.to_datetime, 342
plotting, 332
reading data, 317
CSV files, 317
Excel files, 324
selecting data, 329
Series, 307
sorting data, 328
statistical methods, 330
table, 343
time Series, 309
PEP 8, 34
plotting, 163
tight_layout, 182
3D plots, 217, 280
adjusting space around subplots, 182
color codes, 171
contour plots, 208
error bars, 174, 246
excluding points, 178
insets, 191
layering (order) of plot elements: zorder, 170
line and symbol types, 170
log-log, 183, 186
masked arrays, 178
meshgrid, 206
multiple axes, 189
OO interface, 183, 200
PyPlot, 166, 200
semi-log, 181, 186
set axis limits, 177, 187
streamline plots, 211
subplots, 179, 187
grid, 187
unequal grid, 247, 334
subplots, 187
two x axes, 190
polar plot, 365
polynomial functions
numpy.polynomial.polynomial

polyfit, 240
polyint, 262
polyval, 240, 262
Laguerre, 226
Legendre, 226
polynomials, 242
positional argument, *see* function
power operator (**), 7
PowerShell
launching under Windows, 5
print() function, 18, 85
f-strings, 89
formatted, 85
with arrays, 90
print function
suppress new line, 117
program, 15
Python
launching shell from Terminal, 5
Python Console, 5
PYTHONPATH, 291
updating, 402

Qt Console for IPython, 5
How to launch, 5
JupyterLab, 39
Spyder, 35

random numbers, 247
integers, 249
normally distributed, 248
Poisson, 249
uniformly distributed, 247
reading data from a file
binary file (fast), 392
CSV file, 92
Pandas, 317
Excel file (Pandas), 324
text file, 91
glob, 353
remainder operator (%), 7
reserved words, 14
routine, *see* program
RuntimeWarning
supress, 132

save animation to movie file, *see* FFmpeg
saving data to a file
binary file (fast), 392
CSV file, 96
text file, 94
SciPy, 19, 225
eigenvalue problems, 253
FFT, 273
importing subpackages, 229
linear algebra, 250
nonlinear equations, 255
numerical integration, 258
random numbers, 247
solving ODEs, 265
special functions, 226
splines, 230
systems of equations, 251
script, *see* program
Seaborn, 172
line & symbol colors, 172
sort data
NumPy arrays, 77, 149
using Pandas, 328
special functions (SciPy), 226, 228
Airy, 226
Bessel, 226
zeros, 228
error, 226
gamma, 226
random numbers, 247
splines, 230
cubic, 232
finding roots, 236

linear, 231
smoothing, 234
Spyder IDE, 35
strings, 51
concatenation (+), 51
quotes, single & double, 51
`split()`, 76
`subplots_adjust`, 182, 209
subtraction operator (-), 7
syntax checkers, 34
syntax highlighting, 15

tab completion, *see* IPython
Terminal
launching under Linux, 5
launching under macOS, 4
launching under Windows, 5
terminating a program while it's still running, 112
TeX in plots, *see* LaTeX in plots
text editors for programming, 15, 16
three-dimensional plots, 217
ticks
color (parameters), 190
manual labels, 188
manual placement, 188, 218, 246
`time` module, 119
timing code execution, *see* `time` module, 120
tuples, 53, 58
`type()` function, 10
type hints, 134

ufuncs, 133
Unicode characters, 53
unicode characters, 52
universal function (`ufunc`), 133
universal functions, *see* ufuncs
updating Python, 400

variables, 10
legal names, 14
vectorized code, 133

wrapper, 1, 19, 200
writing data to a file
binary file (fast), 392
CSV file, 96
text file, 94

YAML files, 295
yield statement, 365, 368

`zip()` function, 95, 98