

# **Scientific Computing for Chemists**

**Charles J. Weiss**



## **Scientific Computing for Chemists**

By Charles J. Weiss

Copyright © 2020 Charles J. Weiss. This document is licensed under CC-BY-NC-SA 4.0 available at <https://creativecommons.org/licenses/by-nc-sa/4.0/>.

While great efforts have gone into ensuring that all the code in this book works as prescribed and all text and code are free of errors, some errors could exist. Additionally, some examples in this book are simplified for pedagogical reasons and may not be appropriate for research and other applications. It is the responsibility of the reader to check that their code is free of errors, behaves as required, and that the methods are appropriate for their applications.

## Preface

I first taught the *Scientific Computing for Chemists* course in the fall of 2016.<sup>1</sup> The goal of the course was to teach students with no previous programming experience to code in Python and the various scientific Python libraries (e.g., SciPy, scikit-image, etc...) and to use these skills to solve an assortment of chemical problems. Finding a book was a particular challenge because there was a paucity of chemistry-specific resources on Python, and no single book covered the range of topics I felt were important and useful for students of chemistry. I set out to write my own book in the summer of 2017 and used the first draft in class in the fall of 2017. This text has been since modified and expanded to the current text.

The emphasis of this book matches that of the course – to provide a streamlined introduction to Python and its scientific libraries in order to allow the student to start applying these new skills to chemistry as quickly as possible. As a result, not all topics covered in a typical computer science course on Python are included here. Instead, the most relevant topics to chemistry are covered along with selections of scientific libraries not likely taught in most Python courses. Another difference between this book and a typical computer science course on Python is that many computer science courses would have students write and save code as text files and run them from the command line. In contrast, this book assumes that the reader is running his or her code in a Jupyter notebook, as described in chapter 0, which is an ideal environment for scientific data analysis. The Jupyter notebook provides immediate feedback to the user, convenient graphical outputs, is shareable, and is simpler to use than running Python scripts from the command line. For those students who wish to continue on to run Python scripts from the command line, chapter 13 provides a brief introduction to this process.

This book is organized in order of more fundamental topics first, but not every earlier chapter is a prerequisite for all subsequent chapters. An outline of prerequisites are listed below. Chapter 0 provides a quick introduction to the Jupyter notebook and chapters 1-2 provide background on the Python programming language. Anyone who already knows Python can skim or skip past these two chapters. Chapter 3 is plotting and visualization and chapter 4 covers NumPy. Both of these chapters are used heavily in this book and should definitely not be bypassed. The pandas library is covered in chapter 5 which is used in some subsequent chapters but not all. This library adds functionality and extra ease-of-use to NumPy. Anyone looking to streamline the schedule could skip this chapter, but be aware this it is heavily utilized in chapters 10 and 12. Luckily, chapters 10 and 12 should be mostly readable by someone who is not familiar with pandas or at least has read sections 5.1-5.2. Beyond chapter 5 are mostly application focused topics or cover libraries for very specific applications such as image processing or machine learning. Chapters 6-13 are designed to be modular, so after getting through chapter 0-5, these subsequent chapters can be covered in any order depending up a readers needs and interests. Below is a brief outline of the chapters.

---

<sup>1</sup> An outline of the original course is provided in Weiss, C. J. “Scientific Computing for Chemists: An Undergraduate Course in Simulations, Data Processing, and Visualization.” *J. Chem. Educ.* **2017**, 94 (5), 592-597.

Chapter 0	Provides a quick introduction to installing and using Jupyter notebooks
Chapter 1	Core Python programming skills
Chapter 2	Intermediate Python programming skills... this chapter contains many useful topics but may be skipped over and returned to as needed for the impatient reader
Chapter 3	Matplotlib plotting library for visualization of data and results
Chapter 4	Introduces the NumPy library which is the foundation of much of the scientific Python ecosystem
Chapter 5	Introduces the pandas data analysis library. This library is not strictly required for much of this book, but it is a useful and popular library that students seem to really enjoy.
Chapter 6	Provides the reader to basic signal processing in Python including finding peaks, smoothing data, and fitting/interpolation among other topics.
Chapter 7	Introduces image processing using the NumPy and scikit-image libraries
Chapter 8	Provides background in doing symbolic math and other more advanced mathematics in Python.
Chapter 9	Simulations
Chapter 10	Seaborn plotting library
Chapter 11	NMR processing with NMRglue.
Chapter 12	Scikit-learn
Chapter 13	Command line

## Acknowledgements

This book took a substantial time to write along with the time and effort in developing the curriculum. I would like to thank the Department of Chemistry at Wabash College and Department of Chemistry and Biochemistry at Augustana University for being supportive of my development of this course and curriculum. I would also like to thank my students for their feedback and enthusiasm for this topic and for testing out the early versions of this book. I would also like to thank my family for the support and patience as I performed endless edits and proofreading. Finally, thank you to the following people for proof reading or reporting errors.

***Wesley A. Deutscher*** helping collect some example data

***M. Roarke Tollar*** providing feedback and reporting typos in chapters 0 and 1

***Dr. Andrew Klose*** providing feedback and reporting typos in chapter 12

## Table of Contents

<b>Chapter 0.....</b>	<b>11</b>
0.1 Python.....	11
0.2 Project Jupyter.....	13
0.3 Markdown.....	16
0.4 Comments.....	17
0.5 SciPy Stack.....	18
Further Reading.....	19
<b>Chapter 1.....</b>	<b>21</b>
1.1 Numbers.....	21
1.2 Variables.....	27
1.3 Strings.....	29
1.4 Boolean Logic.....	35
1.5 Conditions.....	39
1.6 Lists & Tuples.....	41
1.7 Loops.....	46
1.8 File I/O.....	52
1.9 Creating Functions.....	56
Further Reading.....	64
Exercises.....	64
<b>Chapter 2.....</b>	<b>69</b>
2.1 Syntactic Sugar.....	69
2.2 Dictionaries.....	73
2.3 Sets.....	75
2.4 Python Modules.....	76
2.5 Zipping and Enumeration.....	81
2.6 Encoding Numbers.....	83
2.7 Advanced Functions.....	85
Further Reading.....	89
Exercises.....	89
<b>Chapter 3.....</b>	<b>93</b>
3.1 Basic Plotting.....	94
3.2 Plotting Types.....	100
3.3 Overlaying Plots.....	107
3.4 Multifigure Plots.....	109
3.5 3D Plotting.....	113
3.6 Surface Plots.....	114
Further Reading.....	118
Exercises.....	118
<b>Chapter 4.....</b>	<b>121</b>

4.1 NumPy Arrays.....	122
4.2 Reshaping & Merging Arrays.....	125
4.3 Indexing Arrays.....	129
4.4 Vectorization & Broadcasting.....	130
4.5 Array Methods.....	134
4.6 Random Number Generation.....	137
Further Reading.....	143
Exercises.....	143
<b>Chapter 5.....</b>	<b>147</b>
5.1 Basic Pandas Objects.....	148
5.2 Reading/Writing Data.....	152
5.3 Examining Data with Pandas.....	158
5.4 Modifying DataFrames.....	161
Further Reading.....	168
Exercises.....	168
<b>Chapter 6.....</b>	<b>171</b>
6.1 Feature Detection.....	172
6.2 Smoothing Data.....	180
6.3 Fourier Transforms.....	185
6.4 Fitting & Interpolation.....	187
Further Reading.....	193
Exercises.....	193
<b>Chapter 7.....</b>	<b>197</b>
7.1 Basic Image Structure.....	199
7.2 Basic Image Manipulation.....	207
7.3 Scikit-Image Examples.....	215
Further Reading.....	222
Exercises.....	222
<b>Chapter 8.....</b>	<b>225</b>
8.1 Symbolic Mathematics.....	225
8.2 Algebra in SymPy.....	228
8.3 Matrices.....	231
8.4 Calculus.....	236
8.5 Mathematics in Python.....	245
Further Reading.....	246
Exercises.....	246
<b>Chapter 9.....</b>	<b>249</b>
9.1 Deterministic Simulations.....	250
9.2 Stochastic Simulations.....	260
Further Reading.....	266
Exercises.....	266
<b>Chapter 10.....</b>	<b>269</b>
10.1 Seaborn Plot Types.....	270

10.2 Regression Plots.....	270
10.3 Categorical Plots.....	276
10.4 Distribution Plots.....	285
10.5 Pair Plot.....	291
10.6 Heat Map.....	293
10.7 Relational Plots.....	295
10.8 Internal Data Sets.....	300
Further Reading.....	302
Exercises.....	302
<b>Chapter 11.....</b>	<b>303</b>
11.1 NMR Processing.....	304
11.2 Importing Data with NMRglue.....	305
11.3 Fourier Transforming Data.....	307
11.4 Phasing Data.....	308
11.5 Chemical Shift.....	311
11.6 Integration.....	314
11.7 Peak Picking.....	316
Further Reading.....	319
Exercises.....	319
<b>Chapter 12.....</b>	<b>321</b>
12.1 Supervised Learning.....	322
12.2 Unsupervised Learning.....	333
12.3 Final Notes.....	344
Further Reading.....	345
Exercises.....	345
<b>Chapter 13.....</b>	<b>349</b>
13.1 Bash Scripting Primer.....	349
13.2 Running Scripts.....	350
13.3 Additional Inputs.....	351
13.4 Spyder.....	353
Further Reading.....	356
Exercises.....	356



# *Chapter 0*

## *Python & Jupyter Notebooks*

### **0.1 Python**

Python is a popular programming language available on all major computer platforms including Mac, Linux, and Windows. It is a scripting language which means the moment the user presses the Return key or Run, the Python software interprets and runs the code. This is in contrast to a compiled language like C where the code must first be translated into binary (i.e., machine language) before it can be run. On-the-fly interpretation makes Python quick to use and often provides the user with rapid results. This is ideal for scientific data analysis where the user is routinely making changes to the processing and visualization of the data.

Python is free, open source software and is maintained by the non-profit Python Software Foundation.<sup>2</sup> This is appealing for two major reasons. The first is that it is widely, freely, and irrevocably available to anyone who wants to use it regardless of budget. With proprietary software, which is more and more commonly offered under a subscription model, if a company stops selling a software package, it may simply become unavailable leaving users without the software they built their work around. Second, it is open source, so anyone can inspect and modify the code. This allows anyone to review the code to ensure it does what it claims instead of relying on the assertions of the software distributor.

Another reason to use Python over other options, free or otherwise, is the power and the community support available to Python users. Python is a common and popular programming language that has been applied to a wide variety of applications including data analysis, visualization, machine learning, robotics, web scrapping, 3D graphics, and more. As

---

<sup>2</sup> For more information, see <https://www.python.org>

a result, there is a large community built around Python that provides valuable support for those who need assistance. If you are stuck on a problem or have a question, a quick internet search will likely provide the answer. Common internet forums include *stackexchange.com* or *stackoverflow.com* among others. If you have a question or need help on something, you are probably not the first person to ask that question.

Along with Python, this book uses the IPython environment and the Jupyter notebook as a medium for running and sharing Python code. More details are give below on the Jupyter notebook, but for now, know that it provides interactive environments ideal for scientific computing. In addition, we will use a variety of free, open source libraries to provide collections of useful functions in scientific data processing, analysis, and visualization. Think of a library as an add-on or tool pack for Python.

## ***Installing Software***

The first step is to install the software, if you have not done so already, and there are multiple options for software installation. The first is to install Python followed by Jupyter, IPython, and all of the libraries. This is a tedious process. The faster and more convenient approach is to use an installer that brings all the software with it. The Anaconda installer (link below) provided for free by Continuum Analytics is a very popular option. Another common option is the Canopy environment (link below) provided by Enthought.<sup>3</sup> The code for this book should work with either installation, but the instructions for launching applications assumes Anaconda. When installing the software, be sure to choose **Python 3** as this is the current version. While many applications still support Python 2, it is technically legacy and support is being gradually removed. As of the time of this writing, multiple major projects in the scientific Python ecosystem have announced that current or future releases of their software will no longer support Python 2, so it is likely in your interest to be on Python 3. Most of the examples in this book will *probably* work in Python 2 if you absolutely must use it, but you are *strongly* encouraged to install the most recent version of Python.

### **Anaconda Installer**

<https://www.anaconda.com/download/>

### **Enthought Canopy Installer**

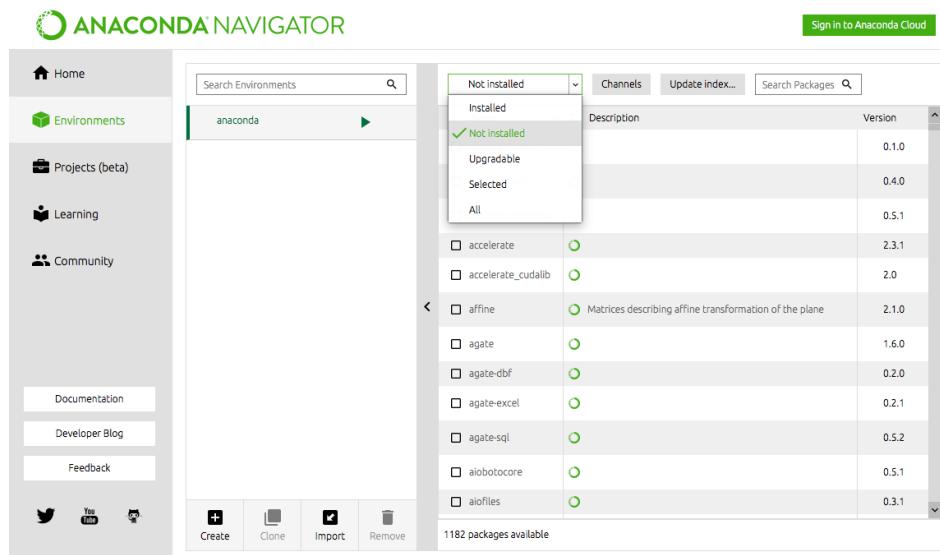
<https://www.enthought.com/products/canopy/>

The Anaconda installer above brings everything you need except for chapter 11 which requires a more specialized library that is not installed by default. If you want to install additional libraries, open the **Anaconda-Navigator** (green circle icon) and select the **Environment** tab on the left. Select **Not Installed** from the pull-down menu to see all the

---

<sup>3</sup> The links and installation instructions for Anaconda and Canopy are current as of this writing but may have since changed.

libraries available to be installed as shown in Figure 0.1. To install a library, check the box next to it and click the **Apply** button. Anaconda will install it and anything else that is required for the new library to work properly. Simple as that!



**Figure 0.1** Installing additional libraries using Anaconda-Navigator.

Any software used in this book that is not installed either by default or through the above method is covered in its respective chapter. A key example of this is NMRglue covered in chapter 11.

## 0.2 Project Jupyter

The Jupyter notebook (formerly known as the IPython notebook<sup>4</sup>... this name still persists in some places) is an electronic document designed to support interactive data processing, analysis, and visualization in an easily shared format. A Jupyter notebook can contain live code, equations, explanatory text, and the output of code such as values, text, images, and plots. The code and examples in this book are intended to be run from a Jupyter notebook but should work fine in many other environments including a basic IPython terminal.

---

<sup>4</sup> The name changed as a result of support for more programming languages than Python. The name Jupyter was forged from Julia/Python/R, the first three languages supported, and is a nod to Galileo Galilei for his notebooks where he sketched the planet Jupiter and moons as observed through his telescope. The Jupyter notebook currently supports 40+ programming languages, but for this book, we will only be addressing Python.

If you already have Python and Jupyter installed, a Jupyter notebook can be launched by either starting the **Navigator** application (green circle icon) and then clicking the **Launch** button for the Jupyter notebook. Alternatively, Jupyter can be launched from a terminal or shell by typing `jupyter notebook` or `jupyter-lab`. The Jupyter notebook will launch in the web browser, but this is *not* a website or web application. An internet browser is fundamentally a fancy image and document viewer. From here, you can either select an already existing Jupyter notebook, denoted by the `.ipynb` extension, to open it or create a new notebook by clicking the **New** menu (Figure 0.2) and selecting **Python 3**.



**Figure 0.2** Launching a new Jupyter notebook.

## Notebook Structure

The Jupyter notebook is structured as a series of cells of two main types: code and markdown. The *code cells* contain live Python code that can be run inside the notebook with any output of the code, including values, text, and plots, appearing directly below the cell. The *markdown cell* is the other common cell type and is designed to contain explanatory information on what is happening in the code cells. They can contain text, equations, and images to help the user convey information. Markdown cells support formatting in markdown, html, and Latex. These two types of cells provide the user with the ability to produce documents containing the data analysis, results, and explanations of the data and analysis along with any conclusions.

Both code and markdown cells can be run by either selecting **Run Cells** in the **Cell** menu (Figure 0.3), by clicking the ► button at the top of the notebook, or by using the **Shift + Return** shortcut. When a code cell is run, the code is executed with any output appearing directly below. When a markdown cell is executed, the text in the cell is rendered to look nicer, and any html or Latex code is rendered to generate the equation(s) or desired formatting.

jupyter Deterministic Chemical Kinetics ODEINT\_v2 (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

**First-Order**

The concentration of a first-order reaction depends on the amount of A.

d below for the following first-order chemical reaction. The radioactive decay of an isotope is an example of a first-order reaction. The reaction rate is described by  $\text{Rate} = k[A]$  where  $[A]$  is the reactant concentration (M) and Rate is the rate of the reaction (M/s).

$A \rightarrow P$

In [2]:

```
t = np.linspace(0, 20, 100)
A_0 = 1 # initial concentration (M)
k = 0.2 # rate constant (s^-1)

# dA/dt = k[A]
def rate_lst(A, t):
    return -k * A

A_t = scipy.integrate.odeint(rate_lst, A_0, t)

plt.plot(t, A_t, 'o')
plt.xlabel('Time, s')
plt.ylabel('[A], M')
```

Out[2]: <matplotlib.text.Text at 0x11b5c96a0>

Time (s)	[A] (M)
0.0	1.00
2.5	0.80
5.0	0.63
7.5	0.49
10.0	0.36
12.5	0.26
15.0	0.19
17.5	0.14
20.0	0.10

**Figure 0.3** Running a selected cell in a Jupyter notebook by selecting **Cell → Run Cells**.

## 0.3 Markdown

*Markdown* is a light-weight markup language that allows users to make text bold, italic, or monospaced text and various kinds of lists and other simple formatting. Table 0.1 provides a collection of common markdown syntax (left) with the corresponding rendered result (right). These are worth knowing to generate sharp markdown cells in your Jupyter notebooks. You will likely find that regular usage will commit them to memory.

**Table 0.1** Common Markdown Syntax

Markdown Syntax	Result
# Header	<b>Header</b>
## Sub-Header	<b>Sub-Header</b>
## Sub-Sub-Header	<b>Sub-Sub-Header</b>
* Italic *	<i>Italic</i>
** Bold **	<b>Bold</b>
` Monospace `	Monospace
---	
* Item	• Item
* Item 2	• Item 2
* Item 3	• Item 3
1. First item	1. First item
2. Second item	2. Second item
3. Third item	3. Third item
[link] (www.python.org)	<a href="http://www.python.org">www.python.org</a>

## 0.4 Comments

Along with markdown cells, it is good practice to add comments to your code. Comments are a means of describing what each section of code does and makes it easier for you and others to navigate the code. It may seem clear to you what each piece of code does as you write it, but after a week, month, or longer, it is unlikely to be as obvious. Someone once elegantly described the importance of comments in stating that the “Your closest collaborator is you six months ago, but you don’t reply to emails.”<sup>5</sup> Comment your code now so that you are not confused later.

The code comments are added directly to code cells using the hash (#) symbol. Anything in a line after a hash symbol is not executed. This means that an entire line can be a comment or a comment can be added after code as demonstrated below with comments colored green.

```
import numpy as np

particles = 10000      # number of particles
steps = 1000           # steps in simulation

# steps to iterate over
t = np.arange(0, steps)

loc = np.zeros(particles)    # particle locations

for frame in t:
    # add random value to locations
    loc += 2 * (np.random.rand(particles) - 0.5)
```

---

<sup>5</sup> This quote can be found around the internet and has been attributed to Paul Wilson from the University of Wisconsin at Madison. See [http://www2.stat.duke.edu/~cr173/Sta523\\_Fa15/git.html](http://www2.stat.duke.edu/~cr173/Sta523_Fa15/git.html) (accessed March 2019).

## 0.5 SciPy Stack

The Python programming language allows for add-ons known as *libraries* to provide extra features. Each library is a collection of *modules*, and each module is a collection of *functions*... or occasionally data. For example, the SciPy library contains a module called `integrate` which contains a collection of functions for integrating under sampled data and equations. For scientific applications, there is a series of core libraries collectively known as the *SciPy stack* along with many other popular libraries. Table 0.2 lists some of the common libraries for scientific applications with an asterisk by those often considered part of the SciPy stack.

**Table 0.2** *Python Libraries used in Scientific Computing*

Library	Description
NumPy*	Foundation of the SciPy stack and provides arrays and a large collection of mathematical functions
SciPy*	Scientific data analysis tools for common scientific data analysis tasks including signal analysis, Fourier transform, integration, linear algebra, optimization, feature identification, and others
Matplotlib*	Popular and powerful plotting library
Scikit-Image	Scientific image processing and analysis
IPython*	Interactive scientific Python environment
Seaborn	Advanced plotting library built on matplotlib
SymPy*	Symbolic mathematics (analogous to Mathematica)
Pandas*	Advanced data analysis tools
Scikit-Learn	Machine learning tools
TensorFlow	Machine learning tools for neural networks
NMRglue	Nuclear magnetic resonance data analysis
Biopython	Computational biology and bioinformatics
Scikit-Bio <sup>6</sup>	Computational biology and bioinformatics

---

<sup>6</sup> As of this writing, Scikit-Bio (<http://scikit-bio.org>) is in beta and only runs on macOS and Linux.

## Further Reading

For further reading and exploration on Jupyter notebooks, the Jupyter Project website below is a good place to see what is happening. There are also a number of books that include chapters on the Jupyter notebooks and the interactive IPython environment.

1. Jupyter Project Website. <https://jupyter.org/>.
2. SciPy Website. <https://www.scipy.org/>
3. IPython Interactive Computing Website. <https://ipython.org/>.
4. VanderPlas, J. *Python data Science Handbook: Essential Tools for Working with Data*, 1<sup>st</sup> ed.; O'Reilly: Sebastopol, CA, 2017, chapter 1. A free, online version is available from the author at  
<https://github.com/jakevdp/PythonDataScienceHandbook>.



# *Chapter 1*

## *Basic Python*

### 1.1 Numbers

#### 1.1.1 Basic Mathematics

To a degree, Python is an extremely powerful calculator that can perform both basic arithmetic and advanced mathematical calculations. Doing math in a Python interpreter is similar to using a graphing calculator – the user inputs a mathematical expression in a line and presses **Return** (or **Shift-Return** in the case of a Jupyter notebook cell), and the output appears directly below. Python includes a few basic mathematical operators shown in Table 1.1.

**Table 1.1** Basic Mathematical Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division (regular)
//	Integer division (aka. floor division)
**	Exponentiation
%	Modulus (aka. remainder)

The addition, subtraction, multiplication, and division (regular) operators work the same way as they do in most math classes. In addition, Python follows the standard order of operation,<sup>7</sup> but parentheses can be used to change the flow of the mathematical operations as needed.

```
[in]: 8 + 3 * 2
```

```
[out]: 14
```

```
[in]: (8 + 3) * 2
```

```
[out]: 22
```

You may have noticed that there are spaces around the mathematical operators in the example calculations above. Python does *not* care about spaces *within* a line, so feel free to add spaces to make your calculation more readable as is done above. Python does, however, care about spaces at the *beginning* of a line. This will be further addressed in the sections on conditions and loops.

Regular division, denoted by a single forward slash (/), is exactly what you probably expect. Three divided by two is one and a half. *Integer division*, shown with a double forward slash (//), is a little more surprising. Instead of providing the exact answer, it can be viewed as either rounding down to the nearest integer (also known as *flooring* it) or simply truncating off anything after the decimal place.<sup>8</sup>

```
[in]: 3 / 2
```

```
[out]: 1.5
```

```
[in]: 3 // 2
```

```
[out]: 1
```

Exponentiation is performed with a double asterisk (\*\*). The carrot (^) means something else,<sup>9</sup> so be careful not to accidentally use this.

```
[in]: 2 ** 3
```

```
[out]: 8
```

---

<sup>7</sup> Exponents are first, followed by multiplication/division, and finally addition/subtraction. Expressions inside parentheses are evaluated before operations outside parentheses.

<sup>8</sup> If you are using Python 2, the single slash acts as integer division if the input values are integers. In both Python 2 and Python 3, the double slash is integer division regardless to whether the input values are integers.

<sup>9</sup> The carrot is the xor logic operator. This is not covered in most math classes and will *not* be used here.

Occasionally, obtaining the *modulus* is also useful and is done using the *modulo* operator (%). This is also sometimes referred to as the *remainder* after division as it is whatever is left over that does not divide evenly into the divisor. In the example below, 3 is seen as going into 10 thrice with 1 left over. The left over portion is the modulus. This is often useful in determining if a number is even among other things.

```
[in]: 10 % 3
```

```
[out]: 1
```

### 1.1.2 Integers & Floats

There are two types of numbers in Python – floats and integers. *Floats*, short for “floating point numbers,” are values with decimals in them. They may be either whole or non-whole values such as 3.0 or 1.2, but there is always a decimal point. *Integers* are whole numbers with no decimal point such as 2 or 53.

Mathematical operations that include only integers and evaluate to a whole number will generate an integer. All other situations will generate a float. In the second example below, a float is generated because one of the inputs is a float. In the third example below, a float is generated despite only integers in the input because the operation evaluates to a fraction.

```
[in]: 3 + 8
```

```
[out]: 11
```

```
[in]: 3.0 + 8
```

```
[out]: 11.0
```

```
[in]: 2 / 5
```

```
[out]: 0.4
```

Integers and floats can be interconverted using the `int()` and `float()` functions.

```
[in]: int(3.0)
```

```
[out]: 3
```

```
[in]: float(4)
```

```
[out]: 4.0
```

The distinction between floats and integers is often a minor detail. There are times when a specific application or function will require a value as an integer or float. However, a

majority of the time, you do not need to think much about it as Python manages most of this for you in the background.

### 1.1.3 Python Functions

In addition to basic mathematical operators, Python contains a number of mathematical functions. As in mathematics, a function has a name (e.g.,  $f$ ) and the arguments are placed inside of the parentheses after the name. The *argument* is any value or piece of information fed into a function. In the case below,  $f$  requires a single argument  $x$ .

$$f(x)$$

There are a number of useful math functions in Python with Table 1.2 describing a few common ones such as the absolute value (`abs`) and round (`round`) functions. Note that the `round` function uses Banker's rounding - if a number is half way between two integers (e.g., 4.5), it will round toward the even integer (i.e., 4).

```
[in]: abs(-4)
```

```
[out]: 4
```

**Table 1.2** Common Python Functions

Function	Description
<code>abs()</code>	Returns absolute value
<code>float()</code>	Converts value to a float
<code>int()</code>	Converts value to an integer
<code>len()</code>	Returns the length of an object
<code>list()</code>	Converts an object to a list
<code>max()</code>	Returns the maximum value
<code>min()</code>	Returns the minimum value
<code>open()</code>	Opens a file
<code>print()</code>	Displays an output
<code>round()</code>	Rounds a value using banker's rounding
<code>str()</code>	Converts an object to a string
<code>sum()</code>	Returns the sum of values

<code>tuple()</code>	Converts an object to a tuple
<code>type()</code>	Returns the object type (e.g., float)
<code>zip()</code>	Zips together two lists or tuples

The `print()` function is one of the most commonly used functions that tells Python to display some text or values. While Jupyter notebooks will display the output or contents of a variable by default, the `print()` function allows for a considerable more control as you will see below in section 1.3.

```
[in]: print(8.3145)

[out]: 8.3145
```

In addition to Python's native collection of functions, python also contains a `math` module with more mathematical functions. Think of a module as an add-on or tool pack for Python just like a library. The `math` module comes with every installation of Python and is *imported* (i.e., activated) using the `import math` command. After the module has been imported, any function in the module is called using `math.function()` where `function` is the name of the function. For example, `math` contains the function `sqrt` for taking the square root of values.

```
[in]: import math
       math.sqrt(4)

[out]: 2.0
```

Table 1.3 lists some commonly used functions in the `math` module, and a few examples are shown below.

```
[in]: math.ceil(4.3)

[out]: 5

[in]: math.pi

[out]: 3.141592653589793

[in]: math.pow(2, 8)

[out]: 256.0
```

**Table 1.3** Common math Functions

Function	Description
<code>ceil(x)</code>	Rounds $x$ up to nearest integer
<code>cos(x)</code>	Returns $\cos(x)$
<code>degrees(x)</code>	Converts $x$ from radians to degrees
<code>e</code>	Returns the value $e$
<code>exp(x)</code>	Returns $e^x$
<code>factorial(x)</code>	Takes the factorial (!) of $x$
<code>floor(x)</code>	Rounds $x$ down to the nearest integer
<code>log(x)</code>	Takes the natural log (ln) of $x$
<code>log10(x)</code>	Takes the common log (base 10) of $x$
<code>pi</code>	Returns the value $\pi$
<code>pow(x, y)</code>	Returns $x^y$
<code>radians(x)</code>	Converts $x$ from degrees to radians
<code>sin(x)</code>	Returns $\sin(x)$
<code>sqrt(x)</code>	Returns the square root of $x$
<code>tan(x)</code>	Returns $\tan(x)$

There are more ways to import functions or modules in Python. If you only want to use a single function from the entire module, you can selectively import it using the `from` statement. Below is an example of importing the `radians()` function.

```
[in]: from math import radians  
       radians(4)
```

```
[out]: 0.06981317007977318
```

One advantage of importing only a single function or variable is that you do not need to use the `math.` prefix. Some Python users take this method one step further by using a wild card (\*), which imports everything from the module. That is, they type `from math import *`. This imports all functions and variables and again allows the user to use them without the `math.` prefix. The down side is that you might accidentally overwrite a variable (see following section on variables) in your code this way. Unless you are absolutely certain you know all the functions and variables in a module and that it will not overwrite any variables in your code, do not use the \* import. On second thought, just avoid using the \* import anyway.

## 1.2 Variables

When performing mathematical operations, it is often desirable to store values in *variables* for later use instead of manually typing them back in. This will save effort when writing your code and make any changes automatically propagate through your calculations.

### 1.2.1 Choosing & Assigning Variables

Attaching a value to a variable is called *assignment* and is performed using a single equal sign (=). Below, 5.0 and 3 are assigned to the variables `a` and `b`, respectively.

Mathematical operations can then be performed with the variables just as is done with numerical values.

```
[in]: a = 5.0
      b = 3

[in]: a + b

[out]: 8.0
```

Variables can be almost any string of characters as long as they start with a letter, do not contain an operator (see Table 1.1), and are not contained in Python's list of reserved words shown in Table 1.4.<sup>10</sup> It is also important to not use a variable twice as this will overwrite the first value. Modules and functions are also attached to variables, so if you have imported the `math` module, the module is attached to the variable `math`.

**Table 1.4** Python Reserved Words<sup>11</sup>

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

It is also in your best interest to create variable names that clearly indicate what it contains if it is more than a quick example or experiment. This will make writing and reading code

<sup>10</sup> For a complete list of Python reserved keywords, see:  
[https://docs.python.org/3/reference/lexical\\_analysis.html](https://docs.python.org/3/reference/lexical_analysis.html)

<sup>11</sup> The reserved words `True`, `False`, and `None` are capitalized while the others are not.

significantly easier and is a good habit to start early. In the examples below, a reader might be able to determine that the left example is calculating energy using  $E = mc^2$  while it is more difficult to determine what the example on the right is calculating.

<b>Good Variable Names</b>	<b>Poor Variable Names</b>
[in]: mass_kg = 1.6 light_speed = 3.0e8 mass * light_speed**2 [out]: 1.44e+14	[in]: x = 3.2 a = 1.77 a + x [out]: 4.97

### 1.2.2 Compound Assignment

A variable can be assigned to another variable as is shown below. When this happens, both variables are assigned to the same value, which is not particularly surprising.

```
[in]: x = 5  
y = x  
  
[in]: y  
  
[out]: 5
```

However, watch what happens if the first variable, `x`, is then assigned to a new value.

```
[in]: x = 8  
  
[in]: y  
  
[out]: 5
```

Instead of `y` updating to the new value, it still contains the first value. This is because instead of `y` being assigned to `x`, the value 5 was assigned directly to `y`. Behind the scenes, Python handles assignment by making a *pointer* that connects a variable name to a value in the computer's memory. Figure 1.1 illustrates what happens in the above example.

Interpreter	Memory Pointers
[in]: x = 5	x → 5
[in]: y = x	x → 5 y → 5
[in]: x = 8	x → 8 y → 5

**Figure 1.1** A representation of memory pointer during variable assignment is shown with the Python code (left) and the corresponding points (right).

The x pointer is directed to a new values but the y pointer is still aimed at 5.

## 1.3 Strings

Floats and integers are means of storing numerical data. The other major type of data is text which is stored as a string of characters known simply as a *string*. Strings can contain a variety of characters including letters, numbers, and symbols and are identified by single or double quotes.

```
[in]: 'some text'  
[out]: 'some text'
```

### 1.3.1 Creating a String

The simplest way to create a string is to enclose the text in either single or double quotes,<sup>12</sup> and a string can be assigned to variables just like floats and integers. To have Python print out the text, use the `print()` function.

```
[in]: text = "some text"  
[in]: print(text)  
[out]:  
some text
```

Strings can also be created by converting a float or integer into a string using the `str()` function.

<sup>12</sup> Or in a function's docstring, a triple quote is also used. For typical strings, only use single or double quotes.

```
[in]: str(4)  
[out]: '4'
```

Even though a number can be contained in a string, Python will not perform mathematical operations with it because it sees anything in a string as a series of characters and nothing more. As can be seen below, in attempting to add 4 and 2, instead of doing mathematical addition, Python concatenates the two strings. Similarly, in attempting to multiply 4 by 2, Python returns the string twice and concatenates them. These are ways of combining or lengthening strings, but no actual math is performed.

```
[in]: '4' + '2'  
[out]: '42'  
  
[in]: '4' * 2  
[out]: '44'
```

If two strings are multiplied, Python returns an error. This is an issue commonly encountered when importing numerical data from a text document. The remedy is to convert the string(s) into numbers using either the `float()` or `int()` functions.

```
[in]: '4' * '2'  
TypeError: can't multiply sequence by non-int of type 'str'  
  
[in]: int('4') * int('2')  
[out]: 8
```

If we want to know the length of a string, we can use the `len()` function as shown below.

```
[in]: len(text)  
[out]: 9
```

The length of '`some text`' is nine because a space is a valid character.

To display both text and numbers in the same message, the `print()` function is very helpful. The user can either convert the number to a string and concatenate the two or separate each object by a comma. Notice in the former method, spaces need to be included by the user.

```
[in]: print(str(4.0) + ' g')
```

```
[out]: 4.0 g  
[in]: print(4.0, 'g')  
[out]: 4.0 g
```

### 1.3.2 Indexing and Slicing

Accessing a piece or slice of a string is a common task in scientific computing among other applications. This is often encountered when importing data into Python from text files and only wanting a section of it. *Indexing* allows the user to access a single character in a string. For example, if a string contains the amino acid sequence of a peptide and we want to know the first amino acid, we can use indexing to extract this character. The key detail about indexing in Python is that *indices start from zero*. That means the first character is index zero, the second character is index one, and so on. If we have a peptide sequence of 'MSLFKIRMPE', then the indices are as shown below.

Characters	M	S	L	F	K	I	R	M	P	E
Index	0	1	2	3	4	5	6	7	8	9

To access a character, place the index in square brackets after the name of the string.

```
[in]: seq = 'MSLFKIRMPE'  
[in]: seq[0]  
[out]: 'M'
```

Interestingly, we do not have to use variables to do this; we could perform the same above operation directly on the string.

```
[in]: 'MSLFKIRMPE'[1]  
[out]: 'S'
```

What happens if you want to know the last character of a string? One method is to determine the length of a string and use that to determine the index of the last character.

```
[in]: len(seq)  
[out]: 10
```

```
[in]: seq[9]  
[out]: 'E'
```

An easier approach is to index backwards. The string can be reverse indexed from the last charter to the first using negatives starting with  $-1$  the last character.

<b>Characters</b>	M	S	L	F	K	I	R	M	P	E
<b>Reverse Index</b>	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
[in]: seq[-1]  
[out]: 'E'
```

Indexing only provides a single character, but it is common to want a series of characters from a string. *Slicing* allows us to grab a section of the string. It uses the same index values as above except requires the start and stop indices separated by a colon in the square brackets. One important detail is that the character at the starting index is included in the slice while the character at the final index is excluded from the slice.

```
[in]: seq[0:5]  
[out]: 'MSLFK'
```

If you look at the index values for each letter, you will notice that the character at index 5 (I) is not included.

What happens if you want to grab the last three characters of a string to determine the file extension (i.e., what type of file it is)? The fact that the last index is not included in the slice causes a problem as is shown below.

```
[in]: file = '1rxt.pdb'  
[in]: file[-3:-1]  
[out]: 'pd'
```

The way around this is to just leave the stop index blank. This tells Python to just go to the end.

```
[in]: file[-3:]  
[out]: 'pdb'
```

This trick also works for the start index to get the file name without the extension. Notice that the `-4` index is the period.

```
[in]: file[:-4]  
[out]: 'lrxxt'
```

Finally, we can also adjust the step size in the slice. That is, we can ask for every other character in a string by setting a step size of 2. The overall structure is `[start : stop : step]`.

```
[in]: seq[::-2]  
[out]: 'MLKRP'
```

The above slice does not include any start or stop indices, so it includes the entire length of the string.

### 1.3.3 String Methods

A *method* is a function that works with a specific type of object. String methods only work on strings, and they do not work on other objects such as floats. Later on, you will see other objects like lists and NumPy arrays which have their own methods for performing common tasks with those types of objects. If it makes it any easier, feel free to equate the term “method” with “function” in your mind, but know that there is a bit more to methods.

One example of a string method is the `capitalize()` function which returns a string with the first letter capitalized. Using a string method is referred to as *calling* the method... it is computer science lingo for executing a function. The method is called by appending `.capitalize()` to the string or a variable representing the string. For example, below is an Albert Einstein quote that needs to have the first letter capitalized.

```
[in]: quote = 'anyone who has never made a mistake has never  
tried anything new.'  
  
[in]: quote.capitalize()  
  
[out]: 'Anyone who has never made a mistake has never tried  
anything new.'
```

Notice that if we check the original quote, it is unchanged (below). This method does not change the original string but rather returns a copy with the first letter capitalized. If we want to save the capitalized version, we can assign it to a new variable... or overwrite the original.

```
[in]: quote
[out]: 'anyone who has never made a mistake has never tried
        anything new.'
[in]: cap_quote = quote.capitalize()
[in]: cap_quote
[out]: 'Anyone who has never made a mistake has never tried
        anything new.'
```

As a minor note, string methods can also be called with `str.method(string)` with `method` being the name of the string method and `string` being the string or string variable. While this works, it is used less often. The first approach with `string.method()` is preferred because any string method needs a string to act upon, so many people find it logical that a string should start the function call. It is also shorter to type, which is certainly a virtue.

```
[in]: str.isalpha(quote)
[out]: True
[in]: str.capitalize(quote)
[out]: 'Anyone who has never made a mistake has never tried
        anything new.'
```

Below are a few common string methods you may find useful.

**Table 1.5** Common String Methods

Method	Description
<code>capitalize()</code>	Capitalizes the first letter in the string
<code>center(width)</code>	Returns the string centered with spaces on both sides to have a requested total width
<code>count(charters)</code>	Returns the number of non-overlapping occurrences of a series of characters
<code>find(characters)</code>	Returns the index of the first occurrence of <code>characters</code> in a string
<code>isalnum()</code>	Determines whether a string is all alphanumeric characters and returns <code>True</code> or <code>False</code>

<code>isalpha()</code>	Determines whether a string is all letters and returns True or False
<code>isdigit()</code>	Determines whether a string is all numbers and returns True or False
<code>lstrip(characters)</code>	Returns a string with the leading characters removed; if no characters are given, it removes spaces
<code>rstrip(characters)</code>	Returns a string with the trailing characters removed; if no characters are given, it removes spaces
<code>split(sep=None)</code>	Splits a string apart based on a separator; if no separator is given, it uses white spaces
<code>startswith(prefix)</code>	Determines if the string starts with a prefix and returns True or False
<code>endswith(suffix)</code>	Determines if the string ends with a suffix and returns True or False

## 1.4 Boolean Logic

Python supports *Boolean logic* where all expressions are evaluated to either True or False. These are useful for adding conditions to scripts. For example, if you are writing code to determine if a sample is a neutral pH, you will want to test if the pH equals 7. If the `pH == 7` evaluates as True, the sample is neutral, and if this statement is False, the sample is not neutral.

### 1.4.1 Boolean Basics

There are a number of Boolean operators available in Python with the most common summarized in Table 1.6. These operators are essentially truth tests with Python returning either True or False.<sup>13</sup> Many of them work as one would expect. For example, if 8 is tested for equality with 3, a False is returned. Note that the operator for equals is a *double* equal signs, whereas a single equal sign assigns a value to a variable.

```
[in]: 8 == 3
[out]: False
```

When 8 is tested for being larger than 3, it evaluates as True.

```
[in]: 8 > 3
```

---

<sup>13</sup> True and False are always capitalized in Python.

```
[out]: True
```

**Table 1.6** Basic Boolean Comparison Operators

Operator	Description
<code>==</code>	Equal (double equal sign)
<code>!=</code>	Not equal
<code>&lt;=</code>	Less than or equal
<code>&gt;=</code>	Greater than or equal
<code>&lt;</code>	Less than
<code>&gt;</code>	Greater than
<code>is</code>	Identity
<code>is not</code>	Negative identity

The `is` and `is not` Boolean operators are not as intuitive. These two operators test to see if two objects are the *same thing* (i.e., identity) or *not the same thing*, respectively. For example, if we test 8 and 8.0 for equality, the result is `True` because they are the same quantity. However, if we test for identity, the result is `False` because 8 is an integer and 8.0 is a float.

```
[in]: 8 == 8.0
```

```
[out]: True
```

```
[in]: 8 is 8.0
```

```
[out]: False
```

### 1.4.2 Compound Comparisons

Comparisons can be concatenated together with Boolean logic operators to make compound comparisons. Common Boolean logic operators are shown in Table 1.7.

**Table 1.7** Common Boolean Logic Operators

Operator	Description
<code>and</code>	Tests for both being <code>True</code>
<code>or</code>	Tests for either being <code>True</code>
<code>not</code>	Tests for <code>False</code>

The `and` operator requires both input values to be `True` in order to return `True` while the `or` operator requires only one input value to be `True` in order to evaluate as `True`. The `not` operator is different in that it only takes a single input value and returns `True` if and only if the input value is `False`. It is essentially a test for `False`.

```
[in]: True and False
```

```
[out]: False
```

```
[in]: True or False
```

```
[out]: True
```

```
[in]: 8 > 3 or 8 < 2
```

```
[out]: True
```

```
[in]: not 8 > 3
```

```
[out]: False
```

Truth tables for the three common Boolean logic operators are shown below. Boolean logic by itself is not immensely useful, but when paired with conditions (introduced below), it is a powerful tool in programming and data analysis.

**Table 1.8** Truth Table for the `and/or` Logic Operators

<b>p</b>	<b>q</b>	<b>p and q</b>	<b>p or q</b>
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

**Table 1.9** Truth Table for the `not` Logic Operator

<b>p</b>	<b>p not q</b>
True	False
False	True

### **1.4.3 Alternative Truth Representations**

The values 1 and 0 can also be used in place of `True` and `False`, respectively, as Python recognizes them as surrogates. For Python to know that you mean these values as Booleans and not simply integers, Python sometimes requires the `bool()` function.

```
[in]: bool(1)
```

```
[out]: True
```

```
[in]: bool(0)
```

```
[out]: False
```

It is customary to use the values 1 and 0, but any non-zero value will evaluate to `True` as well.

```
[in]: bool(5)
```

```
[out]: True
```

### **1.4.4 `any()` and `all()`**

It is sometimes helpful to test if any or all values test `True` in a list or tuple covered in section 1.6. The `any()` and `all()` functions do exactly this. The former will return `True` if one or more of the values in the object test `True` while the latter will only evaluate as `True` only if all values are `True`.

```
[in]: any([True, True, False])
```

```
[out]: True
```

```
[in]: all([True, True, False])
```

```
[out]: False
```

```
[in]: all([True, True, True])
```

```
[out]: True
```

When fed numbers, both the `any()` and `all()` functions will treat them as Booleans as described in section 1.4.3.

```
[in]: any([0, 1, 0])
```

```
[out]: True
```

### 1.4.5 Test for Inclusion

Python allows for the testing of inclusion using the `in` operator. Let us say we want to test if there is nickel in a provided molecular formula. We can simply test to see if “Ni” is in the formula.

```
[in]: comp1 = 'Co(NH3)6'  
       comp2 = 'Ni(H2O)6'  
  
[in]: 'Ni' in comp1  
  
[out]: False  
  
[in]: 'Ni' in comp2  
  
[out]: True
```

The `in` operator also works for other objects beyond strings including lists and tuples which you will learn about in section 1.6.

## 1.5 Conditions

*Conditions* allow for the user to specify if and when certain lines or blocks of code are executed. Specifically, when a condition is true, the block of *indented* code directly below runs. In the example below, if pH is greater than 7, the code prints out the statements “The solution is basic” and “Neutralize with acid.”

```
if pH > 7:  
    print('The solution is basic.')  
    print('Neutralize with acid.')
```

### 1.5.1 `if` Statements

The `if` statement is a powerful way to control when a block of code is run. It is structured as shown below with the `if` statement ending in a colon and the block of code below indented by *four spaces*. In the Jupyter notebook, hitting the **Tab** key will also generate four spaces.

```
[in]: x = 7  
  
[in]: if x > 5:  
        y = x **2  
        print(y)
```

```
[out]: 42
```

If the Boolean statement is true at the top of the `if` statement, the code indented below will be run. If the statement is false, Python skips the indented code as shown below.

```
[in]: x = 3
[in]: if x > 5:
        y = x **2
        print(y)
```

Nothing is printed or returned in this code because `x` is not greater than 5.

### **1.5.2 `else` Statements**

There are times when there is an alternative block of code that you will want to be run if the `if` statement evaluates as `False`. This is accomplished using the `else` statement as shown below.

```
[in]: if pH == 7:
        print('The solution is neutral.')
    else:
        print('The solution is not neutral.)
```

If `pH` does not equal 7, then anything indented below the `else` statement is executed.

There is an additional statement called the `elif` statement, short for “else if,” which is used to add extra conditions below the first `if` statement. The block of code below an `elif` statement only runs if the `if` statement is false and the `elif` statement is true. In the example below, if `pH` is equal to 7, the first indented block is run. Otherwise, if `pH` is greater than 7, the second block is executed. In the event that the `if` and all `elif` statements are false, then the `else` block is executed.

```
[in]: if pH == 7:
        print('The solution is neutral.')
    elif pH > 7:
        print('The solution is basic.')
    else:
        print('The solution is acidic.)
```

It is worth noting that `else` statements are not required with every `if` statement, and the last condition above could have been `elif pH < 7:` and have accomplished the same result.

## 1.6 Lists & Tuples

Up to this point, we have only been dealing with single values or strings. It is common to work with a collection of values such as the average atomic masses of the chemical elements, but it is inconvenient to assign each value to its own variable. Instead, the values can be placed in a list or tuple. Lists and tuple are both collections of *elements*, such as numbers or strings, with the key difference that a list can be modified while a tuple cannot. A tuple is said to be *immutable* as it cannot be changed once created. Not surprisingly, lists are often more useful than tuples.

### 1.6.1 Creating Lists

A list is created by placing elements inside square brackets. Below, the list called mass is created containing the atomic mass of the first six elements.

```
[in]: mass = [1.01, 4.00, 6.94, 0.01, 10.81, 12.01]  
[in]: mass  
[out]: [1.01, 4.00, 6.94, 0.01, 10.81, 12.01]
```

A single list can contain a variety of different types of objects. Below a list called EN is created to store the Pauling electronegativity values for the first six elements on the periodic table. The list contains mostly floats, but being that the value for He is unavailable, an 'NA' string resides where a value would otherwise be.

```
[in]: EN = [2.1, 'NA', 1.0, 1.5, 2.0, 2.5]  
[in]: EN  
[out]: [2.1, 'NA', 1.0, 1.5, 2.0, 2.5]
```

### 1.6.2 Indexing & Slicing Lists

*Indexing* is used to access individual elements in a list, and this method is similar to indexing strings as demonstrated below. The index is the position in the list of a given object, and again, the *index numbering starts with zero*. Accessing an element of a list is done by placing the numerical index of the element we want in square brackets behind the list name. For example, if we want the first element in the electronegativity list (EN), we use EN [0], while EN [1] provides the second element and so on.

```
[in]: EN[0]
```

```
[out]: 2.1  
[in]: EN[1]  
[out]: 'NA'
```

Multiple elements can be retrieved at once by including the start and stop indices separated by a colon. Like in strings, this process is known as *slicing*. A convention that occurs throughout Python is that the first index is included but the second is not.

[included : excluded : step]

```
[in]: EN[0:3]  
[out]: [2.1, 'NA', 1.0]  
[in]: EN[3:5]  
[out]: [1.5, 2.0]
```

Just like in strings, if we want everything to the end, provide no stop index.

```
[in]: EN[3:]  
[out]: [1.5, 2.0, 2.5]
```

### 1.6.3 List Methods

Similar to strings, list objects also have a collection of methods (i.e., functions) for performing common tasks. Some of the more common and useful list methods are presented in Table 1.10, and all of these methods modify the original list except `copy()`. As is the case with methods, they only work on the object type they are designed for, so list methods only work on lists.

**Table 1.10** Common List Methods

Method	Description
<code>append(element)</code>	Adds a single <code>element</code> to the end of the list
<code>clear()</code>	Removes all elements from the list
<code>copy()</code>	Creates an independent copy of the list
<code>count(element)</code>	Returns the number of times an <code>element</code> occurs in the list
<code>extend(elements)</code>	Adds multiple <code>elements</code> to the list

<code>index(element)</code>	Returns the index of the first occurrence of <code>font</code>
<code>insert(index, element)</code>	Inserts the given <code>element</code> at the provided <code>index</code>
<code>pop(index)</code>	Removes and returns <sup>a</sup> the element from a given index; if no index is provided, it defaults to the last element
<code>remove(element)</code>	Removes the first occurrence of <code>element</code> in the list
<code>reverse()</code>	Reverses the order of the entire list
<code>sort</code>	Sorts the list in place <sup>b</sup>

<sup>a</sup> Returns means that Python outputs the value to the user. It does *not* mean that the value is returned to the list.

<sup>b</sup> That is, it modifies the original list opposed to the `sorted()` function which leaves the original list unchanged.

Below is a list containing the masses, in g/mol, of the first seven elements on the periodic table. They are clearly not in order, so they can be sorted using the `sort()` method. Unlike the `sorted()` function (Table 1.2), the `sort()` method modifies the original list.

```
[in]: mass = [4.00, 1.01, 6.94, 14.01, 10.81, 12.01, 9.01]
[in]: mass.sort()
mass
[out]: [1.01, 4.0, 6.94, 9.01, 10.81, 12.01, 14.01]
```

The list can be reversed using the `reverse()` method.

```
[in]: mass.reverse()
mass
[out]: [14.01, 12.01, 10.81, 9.01, 6.94, 4.0, 1.01]
```

Probably one of the most useful methods in Table 1.10 is the `append()` method. This is used for adding a single element to a list. The `extend()` method is related but is used to add multiple elements to the list.

```
[in]: mass.append(16.00)
mass
[out]: [14.01, 12.01, 10.81, 9.01, 6.94, 4.0, 1.01, 16.00]
[in]: mass.extend([19.00, 20.18])
mass
```

```
[out]: [14.01, 12.01, 10.81, 9.01, 6.94, 4.0, 1.01, 16.0,  
19.0, 20.18]
```

If multiple elements are added using the `append()` method, it will result in a nested list... that is, a list inside the list as demonstrated below.

```
[in]: mass.append([23.00, 24.31])  
  
[out]: [14.01, 12.01, 10.81, 9.01, 6.94, 4.0, 1.01, 16.0,  
19.0, 20.18, [23.0, 24.31]]
```

There are times when this might be what we want, but probably not here.

## 1.6.4 Range Objects

It is common to need a sequential series of values in a specific range. The user can manually type these values into a list, but computer programming is about making the computer do the hard work for you. Python includes a function called `range()` that will generate a series of values in the desired range. The `range()` function requires at least one argument to tell it how high the range should be. For example, `range(10)` generates values up to and excluding 10.

```
[in]: a = range(10)  
       print(a)  
  
[out]: range(0, 10)
```

The output of `a` is probably not what you expected. You were probably expecting a list from  $0 \rightarrow 9$ , which is what used to happen back in the Python 2 days. Now, Python generates a `range` object that stands in the place of a list because it requires less memory. If you want an actual list from it, just convert it using the `list()` function.

```
[in]: list(a)  
  
[out]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The `range()` function also takes additional arguments to further customize the range and spacing of values. A start and stop position may be provided to the `range()` function as shown below. Consistent with indexing, the range includes the start value and excludes the stop value.

```
[in]: list(range(3,12))  
  
[out]: [3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Finally, a step size can also be included. The default step size is one, but it can be increased to any integer value including negative numbers.

```
[in]: list(range(3,20,3))  
[out]: [3, 6, 9, 12, 15, 18]  
  
[in]: list(range(10,3,-1))  
[out]: [10, 9, 8, 7, 6, 5, 4]
```

While range objects may seem intimidating, they can be used in place of a list. Just pretend the range object is really a list. For example, you can index it like a list as shown below.

```
[in]: ten = range(10)  
[in]: ten[2]  
[out]: 2
```

### 1.6.5 Tuples

*Tuples* are another object type similar to lists except that they are *immutable*... that is to say, they cannot be changed once created. They look similar a list except that they use parentheses instead of square brackets. So what use is an unchangeable list-like object? There are times when you might want data inside your code, but you do not want to accidentally change it. Think of it as something similar to locking a file on your computer to avoid accidentally making modifications. While this features is not strictly necessary, it may be a prudent practice in some situation in case you make a mistake.

Below is a tuple containing the energy in joules of the first five hydrogen atomic orbitals. There is no need to change this data in your code, so fixing it in a tuple makes sense. Indexing and slicing work exactly the same in tuples as they do in string and lists, so we can use this tuple to quickly calculate the energy difference between any pair of atomic orbitals.

```
[in]: nrg = (-2.18e-18, -5.45e-19, -2.42e-19, -1.36e-19, -  
           8.72e-20)  
  
[in]: nrg[1] - nrg[0]  
[out]: 1.635e-18  
  
[in]: nrg[4] - nrg[3]  
[out]: 4.87999999999998e-20
```

That last output is worth commenting on. You may have noticed that the value returned by Python is not exactly what you probably expected based on the precision of the values in the `nrg` tuple. This is because Python does not store values to infinite precision, so this is merely a rounding error.

## 1.7 Loops

*Loops* allow programs to rerun the same block of code multiple times. This is important because there are often sections of code that need to be run numerous times... sometimes extending into the thousands. If we needed to include a separate copy of the same code for every time it is run, our scripts would be unreasonably large.

### 1.7.1 *for* Loops

The `for` loop is probably the most common loop you will encounter. It is typically used to iterate over a multi-element object like lists or tuples, and for each element, the block of indented code below is executed. For example:

```
[in]: for value in [4, 6, 2]:  
        print(2 * value)  
  
[out]:  
8  
12  
4
```

During the `for` loop, each element in the list is assigned to the variable `value` and then the code below is run. Essentially, what is happening is shown below.

```
value = 4  
print(2 * value)  
value = 6  
print(2 * value)  
value = 2  
print(2 * value)
```

This allows us to perform mathematical operations on each element of a list or tuple. If we instead try multiplying the list by two, we get a list of twice the length.

```
[in]: 2 * [4, 6, 2]
[out]: [4, 6, 2, 4, 6, 2]
```

The `for` loop does not, however, modify the original list. If we want a list containing the squares of the values in a previous list, we should first create an empty list and append the square values to the list.

```
[in]: numbers = [1, 2, 3, 4, 5, 6] # original values
      squares = [] # an empty list

[in]: for value in numbers:
      squares.append(value**2)

[in]: squares
[out]: [1, 4, 9, 16, 25, 36]
```

We can also iterate over range objects and strings using `for` loops. Remember that range objects do not actually generate a list, but we can often treating them as if though they do. As an example, we can generate the wavelengths ( $\lambda$ ) in the Balmer series by the following equation where  $R_{\infty}$  is the Rydberg constant ( $1.097 \times 10^{-2} \text{ nm}^{-1}$ ) and  $n_i$  is the initial principle quantum number.

$$\frac{1}{\lambda} = R_{\infty} \left( \frac{1}{4} - \frac{1}{n_i^2} \right)$$

The code below generates the first five wavelengths (nm) in the Balmer series.

```
[in]: for n in range(3,8):
      lam = 1 / (1.097e-2 * (0.25 - (1/n**2)))
      print(lam)

[out]:
656.3354603463993
486.1744150714068
434.084299170899
410.2096627164995
397.04243897498225
```

A `for` loop can also iterate over a string.

```
[in]: for letter in 'Linus':
      print(letter.capitalize)
```

```
[out]:
```

```
L  
I  
N  
U  
S
```

Another common use of `for` loops is to repeat a task a given number of times. It essentially acts as a counter. Imagine we want to determine how much of a  $183.2 \text{ g}^{235}\text{U}$  sample would be left after six half-lives. We can divide the quantity six times and print the result of each division. To accomplish this, we will have a `for` loop iterate over an object with a length of six, executing the division and printing each mass. The easiest way to generate an iterable object of length six is using the `range()` function.

```
[in]: U235 = 183.2  
      for x in range(6):  
          U235 = U235 / 2  
          print(str(U235) + ' g')
```

```
[out]:
```

```
91.6 g  
45.8 g  
22.9 g  
11.45 g  
5.725 g  
2.8625 g
```

In the above example, the value `x` from the range object is not used in the `for` loop. There is no rule that says it has to be. Also, you may notice that the variable names in all the above examples keep changing. Just like in the rest of your code, you are also welcome to pick your variables in the `for` loop. Some people like to use `x` as a generic variable, but it is often best to give the `for` loop variable a logical name so that it is easy to follow as your code grows more complex.

## 1.7.2 `while` Loops

The other common loop is the `while` loop. It is used to keep executing the indented block of code below until a stop condition is satisfied. As an example, the indented block of code below the `while` statement is run until `x` is no longer less than ten. The `x < 10` is

known as the *termination condition*, and it is checked each time before the indented code is executed.

```
[in]: x = 0
      while x < 10:
          print(x)
          x = x + 2 # increments by 2

[out]:
0
2
4
6
8
```

Essentially, what is going on is shown in the following pseudocode (i.e., not real code), and this continues until  $x$  is no longer greater than 10.

```
is x < 10?, if so:
    print(x)
    x = x + 2

is x < 10?, if so:
    print(x)
    x = x + 2

...

```

The `while` loops is not as common as the `for` loop and should be used with caution. This is because it is not difficult to have what is known as a *faulty termination condition* resulting in the code executing indefinitely... or until you manually stop Python or Python crashes because it ran out of memory. This happens because the termination condition is never met resulting in a run-away process.

### **Do NOT run the following code!**

```
[in]: x = 0
      while x != 10:
          x = x + 3
      print('Done')
```

In the above code, the value is incremented until it reaches 10 (remember, != means “does not equal”), and then a “Done” message is printed... at least that is the intention. No message is ever printed and the while loop keeps running. If we do the math on the values for x, we find that in incrementing by three (0, 3, 6, 9, 12,...), the value for x never equals 10, so the while loop never stops. For this reason, it is wise to avoid while loops unless you absolutely must use them. If you do use a while loop, triple check your termination condition and avoid using = or != in your termination condition. Instead, try to use <= or >. These are less likely to fail.

### 1.7.3 Continue, Pass, and Break Commands

Other ways to control the flow of code execution are the continue, pass, and break commands. These are not used often, but it is helpful to know about them on the occasions that you need them. Table 1.11 summarizes each of these statements below.

**Table 1.11** Loop Interruptions

Statement	Description
break	Breaks out of lowest containing for/while loop
continue	Starts the next iteration of the lowest containing for/while loop
pass	No action; code continues on

The break statement breaks out of the most lowest containing loop. This is useful if you want to apply a condition to completely stop the for or while loop early. For example, we can simulate the titration of 0.9 M NaOH with 1 mL increments of 1.0 M HCl. In the code below, the initial volumes of NaOH and HCl are 25 mL and 0 mL, respectively. The for loop successively checks to see if there is more or equal moles of HCl as NaOH (i.e., the *equivalence point*). If not, the volume of HCl is incremented by one milliliter.

```
[in]: vol_OH = 35
vol_H = 0

for ml in range(1, 50):
    vol_total = vol_OH + vol_H
    mol_OH = 0.9 * vol_OH / 1000
    mol_H = 1.0 * vol_H / 1000
```

```

        if mol_H >= mol_OH:
            break
        else:
            vol_H = vol_H + 1

    print('Endpoint: ' + str(vol_H) + ' mL HCl solution')

[out]: Endpoint: 32 mL HCl solution

```

If we solve this titration using the  $C_1V_1 = C_2V_2$  equation,<sup>14</sup> we expect an endpoint of 31.5 mL of HCl, so a simulated endpoint of 32 mL makes sense. The above simulation can also be written as a `while` loop. A `break` statement can often be avoided through other methods, but it is good to be able to use one in an instance where you really need it.

The `continue` statement is similar to the `break` except that instead of completely stopping a loop, it stops only the current iteration of the loop. That is, the loop immediately starts the next cycle. The script below takes the square root of even numbers only. The even number check is performed with `number % 2 == 1`. If this is true, the number is odd, and the `continue` statement causes the `for` loop to continue on to the next number.

```

[in]: numbers = [1,2,3,4,5,6,7]
for number in numbers:
    if number % 2 == 1:
        continue
    print(math.sqrt(number))

[out]:
1.4142135623730951
2.0
2.449489742783178

```

Finally, the `pass` statement does nothing. Seriously. It is merely a placeholder for code that you have not yet written by telling the Python interpreter to continue on. No completed code should contain a `pass` statement. The reason for using one is to be able to run and test code without errors occurring due to missing parts. If the following code is executed, an error will occur because there is nothing below the `else` statement.

```

[in]: pH = 5
if pH >7:
    print('Basic')

```

---

<sup>14</sup> Where C is concentration and V is volume.

```
else:  
[out]: SyntaxError: unexpected EOF while parsing
```

However, if we add a `pass` statement, no error occurs allowing us to see if the code works... aside from the missing part.

```
[in]: pH = 5  
      if pH >7:  
          print('Basic')  
      else:  
          pass
```

## 1.8 File I/O

Up to this point, we have only been dealing with computer-generated and manually typed values, strings, lists, and tuples. In research and laboratory environments, we often need to work with data stored in a file. These files may be generated from an instrument or as the result of humans typing values into a spreadsheet as they take measurements or make observations. There are two general categories of data files: text and binary files. *Text files* are those that, when opened by a text editor, can be read by humans, while *binary* files cannot. We will focus only on text files here.

There is a large variety of text files which differ simply by the way in which the information is formatted in the file. Common examples include comma separated values (CSV), protein database (PDB), and xyz coordinates (XYZ). These files have different extensions (i.e., those 3-4 letters after the period at the end of a file name), but they are all just text files. You can change the extension to `.txt` if you like. The `.csv`, `.pdb`, and `.xyz` are simply tags to help your computer decide which software application can and should open the file.

We will focus on the CSV file format as it is extremely common, and many software applications can export data in the CSV format. Comma separated value files are a way of encoding information that might otherwise be stored in a spreadsheet.<sup>15</sup> Each line of the text file is a different row, and each item in a row is separated by commas... hence the name. Below are the contents of a CSV file and how it would look in a spreadsheet. You will notice that each line ends in a `\n` known as a line break. These are how software applications know where each line ends, and some software includes these and some does not. Most software hides these symbols when displaying the file contents. Python does read them, so they sometimes need to be stripped out of the data (e.g., using `rstrip()`).

---

<sup>15</sup> Most if not all spreadsheet applications can read and write their data to and from csv files.

CSV File	Spreadsheet View
1, 1 \n	1
2, 4 \n	4
3, 9 \n	9
4, 16 \n	16
5, 25 \n	25
6, 36 \n	36
7, 49 \n	49
8, 64 \n	64
9, 81 \n	81
10, 100 \n	100

### 1.8.1 Reading Lines

The first method we will cover for reading text files is the native Python method of reading the lines of the text file one at a time. This method requires a little more effort than the other methods in this book, but it also offers much more control.

There are three general steps for this approach: open the file, read each line one at a time, and close the file. Opening the file is performed with the `open()` function. Be sure to attach the file to a variable to be accessed later. Next, the data is read a single line at a time using the `readlines()` method. Being that we need to do the same task over and over, we will use a `for` loop. Finally, it is a good habit to close the file using the `close()` command. This process is demonstrated below in opening the data shown above in a file called `squares.csv`.<sup>16</sup>

```
[in]: file = open('squares.csv')
        for line in file.readlines():
            print(line)
        file.close()

[out]:
1, 1 \n
2, 4 \n
```

---

<sup>16</sup> This code below also assumes that the csv file is in the same directory as the Jupyter notebook.

```
3, 9 \n
4, 16 \n
5, 25 \n
6, 36 \n
7, 49 \n
8, 64 \n
9, 81 \n
10, 100 \n
```

It worked! The above code read each line and printed the contents. Of course, this is not particularly useful in this form. It would much more useful in a list and without the \n characters. We can fix this by creating a couple empty lists and appending the values to them as they are read. We also need to strip off the line break using the `rstrip()` method.

```
[in]: file = open('squares.csv')

numbers = []
squares = []

for line in file.readlines():
    fields = line.split(',') # splits line at comma
    numbers.append(int(fields[0]))
    squares.append(int(fields[1].rstrip('\n')))

file.close()
```

Now the values are in two separate lists. The first values are in the `numbers` list and the squares of the numbers are in the `squares` list.

### **1.8.2 Using `np.genfromtxt()`**

The second approach uses a function from the NumPy library called `genfromtxt()`. NumPy will not be covered in depth until chapter 4, but we can still use this one function before then. Before using NumPy, we need to import it using `import numpy as np`, which can be thought of as activating it. The `genfromtxt()` function takes two required arguments for reading a text file: the file name and the delimiter. The *delimiter* is the symbol the separates values in each row and can be almost any symbol including spaces or tabs. If you encounter tab separated data, use `delimiter='\t'`, and for comma separated values (CSV) file, use `delimiter=', '`.

```
[in]: import numpy as np
[in]: file = np.genfromtxt('squares.csv', delimiter=',')
      file

[out]:
array([[ 0.,   0.],
       [ 1.,   1.],
       [ 2.,   4.],
       [ 3.,   9.],
       [ 4.,  16.],
       [ 5.,  25.],
       [ 6.,  36.],
       [ 7.,  49.],
       [ 8.,  64.],
       [ 9.,  81.],
       [10., 100.]])
```

The output of this function is something called a *NumPy array*. It is similar to a list except more powerful. You will learn to use these in chapter 4, but for now, just treat it as a nested list (i.e., a list within a list). If we want to know the square of 4, we can access that value using indexing.

```
[in]: file[4][1]
[out]: 16.0
```

Another feature of the `genfromtxt()` function is the `skiprows` optional argument. It instructs the function to disregard data until after a certain number of rows in the file. This is helpful because files often include non-data headers providing details like the instrument, date, time, and other details about the data. A data file may look like this.

```
July 7, 2017
number, square
1, 1 \n
2, 4 \n
3, 9 \n
4, 16 \n
5, 25 \n
6, 36 \n
7, 49 \n
8, 64 \n
```

```
9, 81 \n
10, 100 \n
```

In this case, we need the function to skip the first two rows as follows.

```
[in]: file = np.genfromtxt('header_file.csv', delimiter=',',
                           skiprows=2)
      file

[out]:
array([[ 0.,    0.],
       [ 1.,    1.],
       [ 2.,    4.],
       [ 3.,   9.],
       [ 4.,  16.],
       [ 5.,  25.],
       [ 6.,  36.],
       [ 7.,  49.],
       [ 8.,  64.],
       [ 9.,  81.],
       [10., 100.]])
```

## 1.9 Creating Functions

After you have been programming for a while, you will likely find yourself repeating the same tasks. For example, let us say your research has you repeatedly calculating the distance between two atoms based on their xyz coordinates. You certainly could rewrite or copy-and-paste the same code every time you need to find the distance between two atoms, but that sounds horrible. You can avoid this by creating your own function that calculates the distance. This way, every time you need to calculate the distance between a pair of atoms, you can call the function and the same section of code located in the function is executed. You only have to write the code once and then you can execute it as many times as you need whenever you need!

### 1.9.1 Basic Functions

To create your own function, you first need a name for the function. The name should be descriptive of what it does and makes sense to you and anyone who would reasonably use

it. If we want to create a function to measure the distance between two atoms, `distance` might be a good name for the function.

The first line of a function definition looks like the following: the `def` statement followed by the name of the function with whatever information, called *arguments*, that is fed into the function, and a colon at the end. In this function, we will feed it the *xyz* coordinates for both atoms as either a pair of lists or tuples. In the parentheses following the function name, place variable names you want to use to represent these coordinates. We will use `coords1` and `coords2` here.

```
def distance(coords1, coords2):
```

Everything inside a function is indented four spaces directly below the first line. The distance between two points in 3D space is described by the following equation.

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2}$$

It is now a matter of coding this into the function. Being that we will take the square root, we also need to import the `math` module.

```
[in]: import math  
  
[in]: def distance(coords1, coords2):  
        # changes along the x, y, and z coordinates  
        dx = coords1[0] - coords2[0]  
        dy = coords1[1] - coords2[1]  
        dz = coords1[2] - coords2[2]  
  
        d = math.sqrt(dx**2 + dy**2 + dz**2)  
  
        print('The distance is: ' + str(d))
```

If you run the above code, nothing happens! This is because you defined the function but never actually used it. Calling our new function is done the same way as any other function in Python.

```
[in]: distance((1,2,3), (4,5,6))  
  
[out]: The distance is: 5.196152422706632
```

It works! This function prints out a message stating the distance between the two *xyz* coordinates, and the better part is that we can use this over and over again without having to deal with the function code.

```
[in]: distance((5,2,3), (7, 5.3, 9))  
[out]: The distance is: 7.133722730804723
```

### 1.9.2 *return Statements*

The `distance` function prints out a value for the distance, but what happens if we want to use this value for a subsequent calculation? Perhaps we want to calculate the average of the distances between multiple pairs of atoms. We certainly do not want to retype these values back into Python, so instead we can have the function *return* the value. You can think of functions as little machines where the arguments in the parentheses are the input and the `return` at the end of the function is what comes out of the machine. Below is a modified version of our `distance()` function. By running the following code, it overwrites the original function.

```
[in]: def distance(coords1, coords2):  
    # changes along the x, y, and z coordinates  
    dx = coords1[0] - coords2[0]  
    dy = coords1[1] - coords2[1]  
    dz = coords1[2] - coords2[2]  
  
    d = math.sqrt(dx**2 + dy**2 + dz**2)  
  
    return d  
  
[in]: distance([5,6,7], [3,2,1])  
[out]: 7.483314773547883
```

Now the function returns a float. We can assign this to a variable or append it to a list for later use.

```
[in]: dist = distance([5,6,7], [3,2,1])  
[in]: dist  
[out]: 7.483314773547883
```

Below is code for iterating over a list of  $xyz$  coordinate pairs and calculating the distances between each pair. The values are appended to a list called `dist_list` from which the average distance is calculated.

```
[in]: pairs = (((1,2,3), (2,3,4)), ((3,7,1), (9,3,0)),
              ((0,0,1), (5,2,7)))

[in]: dist_list = []
      for pair in pairs:
          dist = distance(pair[0], pair[1])
          dist_list.append(dist)

      avg = sum(dist_list)/len(dist_list)

[in]: avg

[out]: 5.691472815049315
```

### 1.9.3 Local Variable Scope

Another advantage of using functions is that they maintain variables in a *local scope*. That is, any variable created *inside* a function is not accessible outside the function. If you look back at our `distance()` function, the variable `d` is only used inside the function. If we try to see what is attached to `d`, we get the following error message.

```
[in]: d

[out]: NameError: name 'd' is not defined
```

This is because the variable `d` can only be used or accessed inside the `distance()` function. This is often very convenient because we do not have to worry about overwriting a variable or using it twice. This means that if a collaborator sends you a function that he/she wrote, you do not need to be concerned if a variable in your code is the same as one in your collaborator's function. The function is self-contained making everything a lot simpler.

The obvious down side to variables being in a local scope inside a function is that you cannot access them directly. If you really need to access a variable in a function, place it in the `return` statement at the end of the function so that the function outputs the contents. Alternatively, you can also assign the contents of a variable inside a function to a variable that was created outside the function. For example, a function can append values to a list created outside of the function, shown below, and the list can be viewed anywhere. This

works because anything that is created outside of the function is visible everywhere and is said to have a *global scope*.

```
[in]: def roots(numbers):
        for number in numbers:
            value = math.sqrt(number)
        square_roots.append(value)

[in]: square_roots = []
root(range(10))

[in]: square_roots

[out]:
[0.0,
 1.0,
 1.4142135623730951,
 1.7320508075688772,
 2.0,
 2.23606797749979,
 2.449489742783178,
 2.6457513110645907,
 2.8284271247461903,
 3.0]
```

#### 1.9.4 Arguments

Functions take in data through *arguments* placed in the parentheses after the function name. Different functions take different numbers and types of arguments from as few as zero to potentially dozens of arguments. Function arguments are also sometimes optional. Some functions allow the user to add extra data or change the functions behavior through arguments.

The first type of argument is a *positional argument*. This is an argument that is required to be in a specific position inside the parentheses. For example, the function below takes in the number of protons and neutrons, respectively, and outputs the isotope name. This function is only written for the first ten elements on the periodic table.

```
[in]: def isotope(protons, neutrons):
        elements = ('H', 'He', 'Li', 'Be', 'B', 'C', 'N',
                   'O', 'F', 'Ne')
```

```

symbol = elements[protons - 1]
mass = protons + neutrons

print(str(mass) + symbol)

```

If we want to know the isotope contains six protons and seven neutrons, we input the values as `isotope(6, 7)` and get  $^{13}\text{C}$  as expected. However, if we switch the arguments to `isotope(7, 6)`, we get  $^{13}\text{N}$ , which is not correct. Positional arguments are extremely common, but unfortunately, the user needs to know what information goes where when calling a function.

```

[in]: isotope(6, 7)
[out]: 13C

[in]: isotope(7, 6)
[out]: 13N

```

The other common type of argument is the *keyword argument*. These arguments are attached to a keyword inside the parentheses. The advantage of a keyword argument is that the user does not need to be concerned about where the argument goes as long as it has the proper label. Below is the same isotope function redefined using keyword arguments.

```

[in]: def isotope(protons=1, neutrons=0):
        elements = ('H', 'He', 'Li', 'Be', 'B', 'C', 'N',
                    'O', 'F', 'Ne')
        symbol = elements[protons - 1]
        mass = protons + neutrons

        print(str(mass) + symbol)

[in]: isotope(protons=1, neutrons=2)
[out]: 3H

```

Now if we switch the order, we still get the same result.

```

[in]: isotope(neutrons=2, protons=1)
[out]: 3H

```

Another advantage of a keyword argument is that a default value can be easily coded in the function. Look up at the most recent version of the `isotope()` function and you will notice that `protons` was assigned to 1 and `neutrons` was assigned to 0 in the function

definition. These are the default values. If we call the function without inputting either or both of these values, the function will assume those values.

```
[in]: isotope()  
[out]: 1H  
  
[in]: isotope(neutrons=2)  
[out]: 3H
```

Functions can also take an indeterminate number of positional or keyword arguments, but this is less common and is covered in chapter 2.

### 1.9.5 Docstrings

The final component of a function is the docstring. Strictly speaking, this is not necessary for a function to work and is sometimes left out for simple functions, but it is a good habit to include them. This is especially true if you are creating the function for a much larger project or passing it to other people. A *docstring* is a string placed at the top a function definition describing what the function does, what types of data it takes, and what is returned at the end of the function. Traditionally, docstrings are enclosed in triple quotes. The first line of the docstring describes what type of data goes in the function and what comes out. In the `distance()` function above, our function takes in a pair of lists or tuples and outputs a single value, so the first line may look something like this.

```
[in]: def distance(coords1, coords2):  
    '''(list/tuple, list/tuple) -> float  
    '''
```

The subsequent lines in the docstring can include other information such as more complete descriptions of what the function does and even short examples.

```
[in]: def distance(coords1, coords2):  
    '''(list/tuple, list/tuple) -> float
```

Takes in the xyz coordinates as lists or tuples for two atoms and returns the distance between them.

```
distance((1,2,3), (4,5,6)) -> 5.196152422706632
```

```
'''
```

```
# changes along the x, y, and z coordinates
```

```
dx = coords1[0] - coords2[0]
dy = coords1[1] - coords2[1]
dz = coords1[2] - coords2[2]

d = math.sqrt(dx**2 + dy**2 + dz**2)

return d
```

Once a docstring is created, it can be accessed by typing the function name, complete with parentheses, and leaving the cursor in the parentheses. Then hit **Shift + Tab** to see the docstring. This trick works with any function in this book.

## Further Reading

There is a plethora of books and resources, free and otherwise, available on the Python programming language. Below are multiple examples. The most authoritative and up-to-date resource is the Python Software Foundation's documentation page also listed below.

1. Python Documentation Page. <https://www.python.org/doc/>.
2. Downey, Allen B. *Think Python*, Green Tea Press, 2012. <http://greenteapress.com/wp/think-dsp/>.
3. Reitz, K.; Schlusser, T. *The Hitchhiker's Guide to Python: Best Practices for Development*, O'Reilly: Sebastopol, CA, 2016.

## Exercises

Solve the following problems using Python in a Jupyter notebook.

### Numbers & Variables

1. A 1.6285 L (V) flask contains 1.220 moles (n) of ideal gas at 273.0 K (T). Remember that  $PV = nRT$  and R is 0.08206 L·atm/mol·K. Calculate the pressure (P) for the above system by assigning all values to variables (e.g., V, n, R, and T) and performing the mathematical operations on the variables.
2. Calculate the distance of point (23, 81) from the origin on an  $xy$ -plane first using the `math.hypot()` function and then by the following distance equation.

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

3. Assign `x = 12` and then *increase* the value by 32 *without* typing "`x = 32`".
4. Solve the quadratic equation using the quadratic formula below for  $a = 1$ ,  $b = 2$ , and  $c = 1$ .

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

5. Create the following variable `elements = 'NaKBrClNOUP'` and slice it to obtain the following strings.
  - a) `NaK`
  - b) `UP`

- c) KBr
  - d) NKrlOP
6. A single bond is comprised of a sigma bond while a double bond includes a sigma plus a pi bond. The following strings contain the bond energies (kJ/mol) for a typical C-C single bond and C=C double bond. Perform a mathematical operation on CC\_single and CC\_double to estimate how much energy a pi bond contributes to a C=C double bond.
- ```
CC_single = "345"
CC_double = "611"
```
7. Removing file extensions
- Write a Python script that takes the name of a PNG image (i.e., name always ends in “.png”) and removes the “.png” file extension *using a string method*.
  - Write a Python script that removes the file extension from a file name *using slicing*. You may assume that the file extensions will always be three letters long with a period (e.g., .png, .pdf, .txt, etc...).

### **Boolean Logic**

8. For DNA = 'ATTCGCCGCTTA', use Boolean logic to show that the DNA sequence is a palindrome (same forwards and backwards). *Hint:* this will require a Boolean logic operator to evaluate as True.
9. The following are the atomic numbers of lithium, carbon, and sodium. Assign each to a variable and use Python to evaluate each of the following.

Li = 3, C = 6, and Na = 11.

- Is Li greater than C?
- Is Na less than or equal to C?
- Is either Li or Na greater than C?
- Are both C and Na greater than Li?

### **Conditions**

10. Write a Python script that can take in any of the following molecular formulas as a string and print out whether the compound is an acidic, basic, or neutral compound when dissolved in water. The script should not contain pre-sorted lists of compound

but rather determine the class of molecule base on the formula. *Hint:* first look for patterns in the acid and base formulas in the following collection.

|                  |                                 |                  |                                |                     |      |
|------------------|---------------------------------|------------------|--------------------------------|---------------------|------|
| HCl              | NaOH                            | KCl              | H <sub>2</sub> SO <sub>4</sub> | Ca(OH) <sub>2</sub> | KOH  |
| HNO <sub>3</sub> | Na <sub>2</sub> SO <sub>4</sub> | KNO <sub>3</sub> | Mg(OH) <sub>2</sub>            | HCO <sub>2</sub> H  | NaBr |

11. Write a Python script that takes in the number of electrons and protons and determines if a compound is cationic, anionic, or neutral.

## ***Lists & Tuples***

12. Create a list of even numbers from 18 → 88 including 88. Using list methods, perform the following transformations in order on the same list:
  - a) Reverse the list
  - b) Remove the last value (i.e., 18)
  - c) Append 16
13. In a Jupyter notebook:
  - a) Create a tuple of even numbers from 18 → 320 including 320.
  - b) Can you reverse, remove, or append values to the tuple?
14. The following code generates a random list of integers from 0 → 20 (chapter 2 will cover this in more detail). Run the code and test to see if 7 is in the list. *Hint:* section 1.4.5 may be helpful.

```
import random
nums = [random.randint(0,20) for x in range(10)]
```

15. Write a sentence (string) attached to a variable.
  - a) Convert all letters to lowercase and split the sentence into individual words using the `split()` string method. This will generate a list of words.
  - b) Modify the list (i.e., the list itself changes) so that the words are in alphabetical order. *Hint:* use list and string methods.

## ***Loops***

16. Using a `for` loop, iterate over a `range` object and append 2× each value into a list called `double`.
17. Write a Python script that prints out “PV = nRT” twenty times.

18. Write a script that generates the following output without typing it yourself. Be sure to include unit labels.

```
1000 g
500.0 g
250.0 g
125.0 g
62.5 g
31.25 g
```

19. The isotope  $^{137}\text{Cs}$  has a half-life about 30.2 years. Using a `while` loop, determine how many half-lives until a 500.0 g sample would have to decay until there is less than 10.00 grams left. Create a counter (`counter = 0`) and add 1 to it each cycle of the `while` loop to keep count.
20. What is a *faulty termination condition* and what is one safeguard against them?... aside from not using `while` loops.

## File I/O

For the following two file I/O problems, first run the following code to generate a test file containing simulated kinetics data.

```
import math
with open('test.txt', 'a') as file:
    file.write('time, [A] \n')
    for t in range(20):
        file.write('%s, %s \n' % (t, math.exp(-0.5*t)))
```

21. Using Python's native `open()` and `read()` functions, open the `test.txt` file and print each line.
22. Using `np.genfromtxt()`, read the `test.txt` file and append the time values to one list and the concentration values to a second list. You will need to skip a line in the file.

## Functions

23. Write and test a function, complete with docstring, that solves the Ideal Gas Law for *pressure* when provided with volume, temperature, and moles of gas ( $R = 0.08206 \text{ L} \cdot \text{atm/mol} \cdot \text{K}$ ) with the following stipulations.

$$PV = nRT$$

- a) Create one version of the function that takes only positional arguments.
  - b) Create a second copy of the function that takes only keyword arguments. Try testing this function with positional arguments. Does it still work?
24. Complete a function started below that calculates the rate of a single-step chemical reaction  $nA \rightarrow P$  using the differential rate law ( $\text{Rate} = k[A]^n$ ).

```
def rate(A0, k=1.0, n=1):  
    ''' (concentration (M), k = 1.0, n = 1) → rate (M/s)  
  
    Takes in the concentration of A (M), the rate  
    constant (k), and the order (n) and returns the rate  
(M/s)  
  
    '''
```

25. DNA is composed of two strands of the nucleotides adenine (A), thymine (T), guanine (G), and cytosine (C). The two strands are lined up with adenine always opposite of thymine and guanine opposite cytosine. For example, if one strand is ATGGC, then the opposite strand is TACCG. Write a function that takes in a DNA strand as a string and prints the opposite DNA strand of nucleotides.

# *Chapter 2*

## *Intermediate Python*

The contents of this chapter are intended for those who wish to dive deeper into the Python programming language. Many of the topics herein are *not* strictly required for subsequent chapters but will make you more efficient and effective as a Python programmer. Items from this chapter are occasionally used in subsequent chapters, but topics from this chapter are not absolutely necessary to reading and understanding subsequent topics in this book if you are in a rush. The sections and sometimes subsections of this chapter may also be read in any order or as needed.

### **2.1 Syntactic Sugar**

*Syntactic sugar* is a nickname given to any part of a programming language that does not extend the capabilities of the language. If any of these features were suddenly removed from the language, the language would still be just as capable, but the advantage of anything labeled “syntactic sugar” is that it makes the code quicker/shorter to write or easier to read. Below are a few examples from the Python language that you are likely to come across and find useful.

#### **2.1.1 Augmented Assignment**

Augmented assignment is a simple example of syntactic sugar that allows the user to modify the value assigned to a variable. If we want to increase a value by one, we can recursively assign the variable to itself plus one as shown below.

```
[in]: x = 5  
[in]: x = x + 1
```

```
x  
[out]: 6
```

This is certainly not difficult, but it does involve typing the variable more than once which becomes less desirable as your variable names get longer. As an alternative, we can also use *augmented assignment* shown below that accomplishes the same task. The `+=` means “increment.”

```
[in]: x += 1  
x
```

```
[out]: 7
```

Augmented assignment can be used with addition, subtraction, multiplication, and division as shown in Table 2.1.

**Table 2.1** Augmented Assignment

| Augmented Assignment | Regular Assignment     | Description          |
|----------------------|------------------------|----------------------|
| <code>x += a</code>  | <code>x = x + a</code> | Increments the value |
| <code>x -= a</code>  | <code>x = x - a</code> | Decrements the value |
| <code>x *= a</code>  | <code>x = x * a</code> | Multiplies the value |
| <code>x /= a</code>  | <code>x = x / a</code> | Divides the value    |

## 2.1.2 List Comprehension

At this point, you may have noticed that it is fairly common to generate a list populated with a series of numbers. If the values are evenly spaced integers, simply use the `range()` function and convert it to a list using `list()`. In all other scenarios, you will need to create an empty list, use a `for` loop to calculate the values, and append the values to the list as they are generated. Below is an example of generating a list of squares of all integers from  $0 \rightarrow 9$  using this method.

```
[in]: squares = []  
      for integer in range(10):  
          sqr = integer**2  
          squares.append(sqr)  
  
[in]: squares  
[out]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This whole process can be compressed down into a single line by expressing the `for` loop in square brackets resulting in the following. This is known as *list comprehension*.<sup>17</sup>

```
[in]: squares = [integer**2 for integer in range(10)]
```

To help you visualize where each part comes from, below are both methods again but with common sections in the same colors.

```
[in]: squares = []
      for integer in range(10):
          sqr = integer**2
          squares.append(sqr)
```

```
[in]: squares = [integer**2 for integer in range(10)]
```

List comprehension can take a little time to get used to, but it is well worth it. It saves both time and space and makes the code less cluttered.

### 2.1.3 Compound Assignment

At the beginning of a program or calculations, it is often necessary to populate a series of variables with values. Each variable may get its own line in the code, and if there are numerous variables, this can clutter your code. An alternative is to assign multiple variables in the same assignment as shown below with atomic masses of the first three elements.

```
[in]: H, He, Li = 1.01, 4.00, 5.39
```

```
[in]: H
```

```
[out]: 1.01
```

Each variable is assigned to the respective value. This is known as *tuple unpacking* as `H`, `He`, `Li` and `1.01, 4.00, 5.39` are automatically turned into tuples by Python (behind the scenes) as demonstrated below.

```
[in]: 1.01, 4.00, 5.39
```

```
[out]: (1.01, 4.00, 5.39)
```

Therefore, the above assignments are an equivalent of the following code.

```
[in]: (H, He, Li) = (1.01, 4.00, 5.39)
```

---

<sup>17</sup> Python also supports analogous code structures for creating other objects such as *tuple comprehension* and *dictionary comprehension* which will not be covered here.

## 2.1.4 Lambda Functions

The *lambda function* is an anonymous function for generating simple Python functions. Their value is that they can be used to generate functions in fewer lines of code than the standard `def` statement, and they do not necessarily need to be assigned to a variable... hence the anonymous part. The latter is useful in applications that require a Python function but the user does not want to clutter the namespace by assigning it to a variable or take the time to define a function normally. The lambda function is defined as shown below with the variable immediately after the lambda statement as the independent variable in the function.

```
[in]: lambda x: x**2  
[out]: <function __main__.lambda>>
```

Being that it is not attached to a variable, it needs to be used immediately. Alternatively, it can be attached to a variable as shown below and then operates like any other Python function.

```
[in]: f = lambda x: x**2  
[in]: f(9)  
[out]: 81
```

As an example looking ahead to chapter 8, the `quad` function from the `scipy.integrate` module is a general-purpose method for integrating the area under mathematical functions. Along with the upper and lower limits, the integration functions require a mathematical function in the form of a Python function (i.e., not just a mathematical expression). This would ordinarily require a formally-defined Python function, but it is often more convenient to use a lambda function as a single use Python function as shown below. In the following example, we use integration to find the probability of finding a particle in the lowest state between 0 and 0.4 in a box of length 1 by performing the following integration.

$$p = 2 \int_0^{0.4} \sin^2(\pi x) dx$$

```
[in]: from scipy.integrate import quad  
       import math  
[in]: quad(lambda x: 2 * math.sin(math.pi*x)**2, 0, 0.4)  
[out]: (0.30645107162113616, 3.402290356348383e-15)
```

The first value in the returned tuple is the result of the integration, and the second value is the estimated uncertainty. Therefore, the particle has about a 30.6% probability of being found in the region of  $0 \rightarrow 0.4$ . Performing this same calculation by defining the function with `def` is shown below. This requires more lines of code than a lambda expression.

```
[in]: def particle_box(x):
        return 2 * math.sin(math.pi*x)**2

[in]: quad(particle_box, 0, 0.4)

[out]: (0.30645107162113616, 3.402290356348383e-15)
```

## 2.2 Dictionaries

Python *dictionaries* are a multi-element Python object type that connects keys and values analogous to the way a real dictionary connects a word (the key) with a definition (the value). These are also known as *associative arrays*. Dictionaries allow the user to access the stored values using a key without knowing anything about order of items in the dictionary. One way to think of a dictionary is an object full of variables and assigned values. For example, if we are looking to write a script to calculate the molecular weight of a compound based on its molecular formula, we would need access to the atomic mass of each element based on the elemental symbol. Here the key is the symbol and the value is the atomic mass. It looks something like a list with curly brackets and each item is a `key:value` pair separated by a colon. Below is an example of a dictionary containing the atomic masses of the first ten elements on the periodic table.

```
[in]: AM = {'H':1.01, 'He':4.00, 'Li':6.94, 'Be':9.01,
           'B':10.81, 'C':12.01, 'N':14.01, 'O':16.00,
           'F':19.00, 'Ne':20.18}
```

With the dictionary in hand, we can access the mass of any element in it using the atomic symbol as the key.

```
[in]: AM['Li']

[out]: 6.94
```

Even though it is traditional to call them `key:value` pairs, the value does not need to be a numerical value. It can also be a string or other object type, and the key can also be any object type.

If you ever find yourself with a dictionary and not knowing the keys, you can find out using the `keys()` dictionary method.

```
[in]: AM.keys()
```

```
[out]: dict_keys(['H', 'He', 'Li', 'Be', 'B', 'C', 'N', 'O',
                  'F', 'Ne'])
```

We can also get a look at the key:value pairs using the `items()` method or iterate over the dictionary to get access to keys, values, or both.

```
[in]: AM.items()

[out]: dict_items([('H', 1.01), ('He', 4.0), ('Li', 6.94),
                  ('Be', 9.01), ('B', 10.81), ('C', 12.01),
                  ('N', 14.01), ('O', 16.0), ('F', 19.0),
                  ('Ne', 20.18)])
```

```
[in]: for key, values in AM.items():
      print(values)

[out]:
1.01
4.0
6.94
9.01
10.81
12.01
14.01
16.0
19.0
20.18
```

Additional key:value pairs can be added to an already existing dictionary using the `update()` method as demonstrated below.

```
[in]: AM.update({'Na', 22.99})
      AM

[out]: {'B': 10.81, 'Be': 9.01, 'C': 12.01, 'F': 19.0, 'H':
      1.01, 'He': 4.0, 'Li': 6.94, 'N': 14.01, 'Na': 22.99,
      'Ne': 20.18, 'O': 16.00}
```

Notice that after adding sodium to the atomic mass dictionary, the order of all the pairs changed. Unlike in a tuple or list, the order in a dictionary does not matter, so it is not preserved.

## 2.3 Sets

Sets are another Python object type you may encounter and use on occasions. These are multi-element objects similar to lists with the key difference that each element can appear only once in the set. This may be useful in applications where code is taking stock of what is present. For example, if we are taking inventory of the chemical stockroom to know which chemical compounds are on hand for experiments, the names of the compounds can be stored in a set. If more than one bottle of a compound is present in the stockroom, the set only contains the name once because we are only concerned with what is available, not how many are available. A set looks like a list except curly brackets are used instead of square brackets.

```
[in]: compounds = {'ethanol', 'sodium chloride', 'water',
                   'toluene', 'acetone'}
```

We can add additional items to the set using the `add()` set method.<sup>18</sup>

```
[in]: compounds.add('calcium chloride')
compounds
```

```
[out]: {'acetone', 'calcium chloride', 'ethanol', 'sodium
        chloride', 'toluene', 'water'}
```

```
[in]: compounds.add('ethanol')
compounds
```

```
[out]: {'acetone', 'calcium chloride', 'ethanol', 'sodium
        chloride', 'toluene', 'water'}
```

Notice that when ethanol is added to the set, nothing changes. This is because ethanol is already in the set, and sets do not store redundant copies of elements.

Multiple sets can be concatenated or subtracted from each other using the `|` and `-` operators, and two sets can be compared using Boolean operators. Below are two sets containing the atomic orbitals in nitrogen (N) and calcium (Ca) atoms. Even though there are three 2p orbitals in nitrogen, it only appears once telling us what types of orbitals are present but not how many.

```
[in]: N = {'1s', '2s', '2p'}
Ca = {'1s', '2s', '2p', '3s', '3p', '4s'}
```

```
[in]: N | Ca # returns orbitals in either set
```

```
[out]: {'1s', '2s', '2p', '3s', '3p', '4s'}
```

---

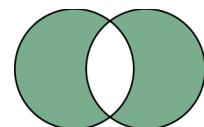
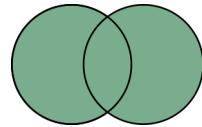
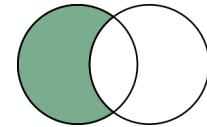
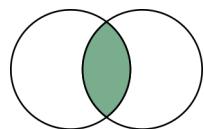
<sup>18</sup> The method is called `add()` and not `append()` like in lists because unlike lists, sets do not preserve the order of items contained within them.

```
[in]: Ca - N # returns Ca orbitals minus those in common
[out]: {'3s', '3p', '4s'}
```

```
[in]: N & Ca # returns orbitals in both sets
[out]: {'1s', '2s', '2p'}
```

**Table 2.2** Python Set Operators

| Operator | Name                 | Description                                                    |
|----------|----------------------|----------------------------------------------------------------|
| &        | Intersection         | Returns items in both sets                                     |
| -        | Difference           | Returns items in the first set minus common items in both sets |
|          | Union                | Merges both sets; redundancies are removed automatically       |
| ^        | Symmetric Difference | Merges both sets minus items in common (i.e., “exclusive or”)  |



## 2.4 Python Modules

Remember from the last chapter that a *module* is a collection of functions and data with a common theme. You have already seen the `math` module in chapter 1, but Python also contains a number of other native modules that come with every installation of Python. Table 2.3 lists a few common examples, but there are certainly many others worth exploring. You

are encouraged to visit the Python website and explore other modules.<sup>19</sup> This section will introduce a few useful modules with some examples of their uses.

**Table 2.3** Some Useful Python Modules

| Name       | Description                                    |
|------------|------------------------------------------------|
| os         | Provides access to your computer file system   |
| itertools  | Iterator and combinatorics tools               |
| random     | Functions for pseudorandom number generation   |
| datetime   | Handling of date and time information          |
| csv        | For writing and reading CSV files              |
| pickle     | Preserves Python objects on the file system    |
| timeit     | Times the execution of code                    |
| audioop    | Tools for reading and working with audio files |
| statistics | Statistics functions                           |

### 2.4.1 os Module

The `os` module provides access to the files and directories (i.e., folders) on your computer. Up to this point, we have been opening files that are in the same directory as the Jupyter notebook, so Jupyter has no difficulty finding the files. However, if you ever want to open a file somewhere else on your computer or open multiple files, this module is particularly useful. Below you will learn to use the `os` module to open files in non-local directories (i.e., not the directory your Jupyter notebook is in) and to open an entire folder of files.

**Table 2.4** Select `os` Module Functions

| Function               | Description                                                       |
|------------------------|-------------------------------------------------------------------|
| <code>chdir()</code>   | Changes the current working directory to the path provided        |
| <code>getcwd()</code>  | Returns the current working directory path                        |
| <code>listdir()</code> | Returns a list of all files in the current or indicated directory |

---

<sup>19</sup> See <https://docs.python.org/3/py-modindex.html> for a more complete listing and descriptions of built-in Python modules.

Table 2.4 provides a description of the three functions that we will be using. To open a file not in the directory of your Jupyter notebook, you will need to change the directory Python is currently looking in, known as the *current working directory*, using the `chdir()` method. It takes a single string argument of the path to the folder containing the files of interest. For example, if the files are in a folder called “my\_folder” on your computer desktop, you might use something like the following. The exact format will vary depending upon your computer and if you are using macOS, Windows, or Linux.

```
[in]: import os  
[in]: os.chdir('/Users/me/Desktop/my_folder')
```

If you are not sure which directory is the current working directory, you can use the `getcwd()` function. It does not require any arguments.

```
[in]: os.getcwd()  
[out]: /Users/me/Desktop/my_folder
```

Another useful function from the `os` module is the `listdir()` method which lists all the files and directories in a folder. It is useful not only for determining the contents a folder but also for iterating through all the files in a folder. Imagine you have not just a single CSV file with data but an entire folder of similar CSV files that you need to import into Python. Instead of handling these files one at a time, you can have Python iterate through the folder and import each CSV file it finds. Below is a demonstration of importing and printing every CSV file on the computer desktop.

```
[in]: import numpy as np  
      dir = os.chdir('/Users/me/Desktop') # changes directory  
      for file in os.listdir():  
          if file.endswith('csv'): # only open csv files  
              data = np.genfromtxt(file)  
              print(data)
```

The code above goes through every file on the computer desktop, and if the file name ends in “csv”, Python imports and prints the contents. Checking the file extension is an important step even if you have a folder that you believe only contains CSV files. This is because folders on many computers contain invisible files for use by the computer operating system. The user usually cannot see them, but Python can and will generate an error if it tries to open it as a CSV file. Checking the file extension ensures that Python only tries to open the CSV files.

## 2.4.2 *itertools* Module

The `itertools` module contains an assortment of tools for looping over data in an efficient manner. There are a number of functions that are good to know from this module, but we will focus on the combinatorics functions `combinations()` and `permutations()`.

The `combinations(collection, n)` function generates all  $n$ -sized combinations of elements from a `collection` such as a list, tuple, or range object. With `combinations()`, order does not matter, so `(1, 2)` is equivalent to `(2, 1)`. In the below code, the `combinations()` function generates all pairs of elements from `numbers`.

```
[in]: import itertools
       numbers = range(5)

[in]: itertools.combinations(numbers, 2)
[out]: <itertools.combinations at 0x10b4697c8>
```

So what just happened? Instead of returning a list, it returned a *combinations object*. You do not need to know much about these except that they can be converted into lists or iterated over to extract their elements, and they are single use. Once you have iterated over them, they need to be generated again if you need them again.<sup>20</sup>

```
[in]: for pair in itertools.combinations(numbers, 2):
       print(pair)
[out]:
(0, 1)
(0, 2)
(0, 3)
(0, 4)
(1, 2)
(1, 3)
(1, 4)
(2, 3)
(2, 4)
(3, 4)
```

---

<sup>20</sup> `combinations()` is a type of function called a *generator*. It only generates values on demand in an effort to reduce the memory usage.

Each combination is returned in a tuple, and if the combination object is converted to a list, it would be a list of tuples.

The `permutations()` function is very similar to `combinations()`, except with `permutations()`, order matters. Therefore, `(2, 1)` and `(1, 2)` are inequivalent. This is especially important in probability and statistics. Permutations of a group of items can be generated just like in the combinations examples above.

```
[in]: for pair in itertools.permutations(numbers, 2):
    print(pair)

[out]:
(0, 1)
(0, 2)
(0, 3)
(0, 4)
(1, 0)
(1, 2)
(1, 3)
(1, 4)
(2, 0)
(2, 1)
(2, 3)
(2, 4)
(3, 0)
(3, 1)
(3, 2)
(3, 4)
(4, 0)
(4, 1)
(4, 2)
(4, 3)
```

Notice how `(0, 2)` and `(2, 0)` are both present in the permutations while only one is listed in the combinations.

### 2.4.3 `random` Module

The `random` module provides a selection of functions for generating random values. Random values can be integers or floats and can be generated from a variety of ranges and distributions. A selection of common functions from the `random` module are shown in Table 2.5. We will not go into much detail here as random value generation is covered in

significantly more detail at the end of chapter 4. One key limitation of the `random` module is that the functions typically only generate a single value at a time. If you want multiple random values, you need to either use a loop or use the random value functions from NumPy presented in chapter 4.

**Table 2.5** Functions from `random` Module

| Function                     | Description*                                                               |
|------------------------------|----------------------------------------------------------------------------|
| <code>random()</code>        | Generates a value from [0, 1)                                              |
| <code>uniform(x, y)</code>   | Generates a float from the range [x, y) with a uniform probability         |
| <code>randrange(x, y)</code> | Generates an integer from the provided range [x, y)                        |
| <code>choice()</code>        | Randomly selects an item from a list, tuple, or other multi-element object |
| <code>shuffle()</code>       | Shuffles a multi-element object                                            |

One point worth noting is that square brackets mean *inclusive* while the parentheses mean *exclusive*, so [0, 9) means from 0 → 9 including 0 but not including 9.

```
[in]: random.random()
[out]: 0.05217270029554699

[in]: random.randrange(0, 10)
[out]: 7

[in]: a = [1,2,3,4,5,6]
      random.shuffle(a)
      a
[out]: [6, 3, 2, 4, 1, 5]
```

## 2.5 Zipping and Enumeration

There are times when it is necessary to iterate over two lists simultaneously. For example, let us say we have a list of the atomic numbers (`AN`) and a list of approximate atomic masses (`mass`) of the most abundant isotopes for the first six elements on the periodic table.

```
[in]: AN = [1, 2, 3, 4, 5, 6]
      mass = [1, 4, 7, 9, 11, 12]
```

If we want to calculate the number of neutrons in each isotope, we need to subtract each atomic number (equal to the number of protons) from the atomic mass. To accomplish this, it would be helpful to iterate over both lists simultaneously. Below are a couple methods of doing this.

### 2.5.1 Zipping Lists

The simplest way to iterate over two lists simultaneously is to combine both lists into a single, iterable object and iterate over it once. The `zip()` function does exactly this by merging two lists or tuples, like a zipper on a jacket, into something like a nested list of lists. However, instead of returning a list or tuple, the `zip()` function returns a single-use zip object.

```
[in]: zipped = zip(AN, mass)

[in]: for pair in zipped:
        print(pair[1] - pair[0])

[out]:
0
2
4
5
6
6
```

As noted above, these are single-use objects, so if we try to use it again, nothing happens.

```
[in]: for pair in zipped:
        print(pair[1] - pair[0])
```

If the two lists are of different length, `zip()` stops at the end of the shorter list and returns a zip object with a length of the shorter list.

### 2.5.2 Enumeration

A close relative to `zip()` is the `enumerate()` function. Instead of zipping two lists or tuples together, it zips a list or tuple to the index values for that list. Similar to `zip()`, it returns a one-time use iterable object.

```
[in]: enum = enumerate(mass)

[in]: for pair in enum:
        print(pair)
```

```
[out]:  
(0, 1)  
(1, 4)  
(2, 7)  
(3, 9)  
(4, 11)  
(5, 12)
```

The `zip()` function can be made to do the same thing by zipping a list with a range object of the same length as shown below, but `enumerate()` may be slightly more convenient.

```
[in]: zipped = zip(range(len(mass)), mass)  
for item in zipped:  
    print(item)  
  
[out]:  
(0, 1)  
(1, 4)  
(2, 7)  
(3, 9)  
(4, 11)  
(5, 12)
```

## 2.6 Encoding Numbers

During most of your work in Python, you do not need to think about how and where the values are stored as Python handles this for you. If you assign a number to a variable, Python will determine how to properly store this information. However, there are instances where you will need to understand a little about how numbers are encoded such as in gray scale images (chapter 7).

Numbers on your computer are stored in *binary* which is a base two numbering system.<sup>21</sup> That is, instead of using digits from  $0 \rightarrow 9$  to describe a number, only 0 and 1 are used. When a number is stored in memory, a fixed block of zeros/ones are allocated to storing this information, and depending upon the size or precision of the number to be stored, this block may need to be larger or smaller. By convention, the blocks are typically 8, 16, 32, 64, or 128 bits (i.e., zeros or ones) in size. Table 2.6 lists a few examples with the terms used by Python.

---

<sup>21</sup> Standard numbers used by humans are a base-ten because we describe values using combinations of ten digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9). Once we get to 9, the digit returns to 0 and a 1 is placed to the left. In a binary numbering system, we use only 0 and 1 to describe values. Analogously, once we get to 1, the digit returns to 0 and a 1 is placed to the left. Therefore, “10” is two in binary.

**Table 2.6** Python Data Types

| Data Type | Description                                        |
|-----------|----------------------------------------------------|
| uint8     | Integers from $0 \rightarrow 255^a$                |
| uint16    | Integers from $0 \rightarrow 65535$                |
| uint32    | Integers from $0 \rightarrow 4294967295$           |
| int8      | Integers from $-128 \rightarrow 127^b$             |
| int16     | Integers from $-32768 \rightarrow 32767$           |
| int32     | Integers from $-2147483648 \rightarrow 2147483647$ |
| float32   | Singe-precision floating-point numbers             |
| float64   | Double-precision floating-point numbers            |

<sup>a</sup> The u is for “unsigned”, so values can only be positive.

<sup>b</sup> The range is the same as uint8 (256 possible values) except they are centered around zero. The positive values only go up to 127 because it also includes zero.

Probably the simplest way to encode a number is an unsigned 8-bit integer. The “unsigned” means that is cannot have a negative sign while the “8-bit” means it can use eight zeros and ones to describe the number. For example, if we want to encode the number 3, it is 00000011. Even if not all the bits are strictly required, they have been allotted for the storage of this value, and with 8 bits, we can encode numbers from  $0 \rightarrow 255$  (i.e., 00000000  $\rightarrow$  11111111). If we want to encode any larger numbers, a longer block of bits such as 16 or 32 will need to be allotted.

To encode a negative integers, *signed integers* are required. The key difference between a signed and unsigned integer is that unsigned are always positive while signed can describe positive and negative values by using the first bit to describe the sign. The first bit is 0 for a positive and 1 for a negative number. Because the first bit is reserved for sign, a signed integer can describe values of only half the magnitude as an unsigned integer of the same bit length. For example, an 8-bit signed integer can describe values from  $-128 \rightarrow 127$ . All combinations of zeros/ones that start with a 0 define positive values from  $0 \rightarrow 127$  while all combinations of zeros/ones that start with a 1 define values from  $-128 \rightarrow -1$ . That is, 10000000 equals -128 while 11111111 describes -1.<sup>22</sup>

---

22 For those who wish to play with this a little, there is a function from the NumPy library (chapter 4) that will convert a number into binary. For example, the below code converts 112 and -110 into 8-bit binary.

```
[in]: import NumPy as np
       np.binary_repr(112, width=8)
[out]:
      '01110000'
[in]: np.binary_repr(-110, width=8)
[out: '10010010'
```

For non-integer values, we need floats. The number of bits used to describe a float dictates the precision of the value... or rather is the number of decimal places the float extends. The various types listed above support both positive and negative values, and the more bits, the more precision they offer.

## 2.7 Advanced Functions

Section 1.9 describes positional arguments and keyword arguments as two methods for providing functions with information and instructions, but thus far, these methods have only allowed the function to take a predetermined numbers of arguments. While some flexibility is offered by the ability to set default keyword arguments that users have the option of overriding or leaving as the default, there is still a limit on the number of parameters in the function. What do we do when we need to write a function that takes an unspecified number of arguments? This section provides two approaches to solving this problem.

### 2.7.1 Variable Positional Arguments

As a possible use case, it is common practice in lab to purify a solid compound by recrystallization, and chemists will often harvest multiple crops of crystals from the same solution to get the highest possible yield. If we want to write a function that returns the percent yield of a synthesized compound using the theoretical yield and the yields of each recrystallization crop, we are faced with the challenge of not knowing how many crops to expect. One solution is a var-positional argument.

The *var-positional argument* (often `*arg`), is a positional argument that accepts variable numbers of inputs. The arguments are then stored as a local tuple in the function attached to the `arg` variable. Even though it is extremely common in examples to see people use `arg` as the variable, you may use any non-reserved variable you like as long as you precede it with an asterisk in the function definition. For example, a function for calculating the percent yield is shown below with `g_theor` as the theoretical yield in grams and `g_crops` as the var-positional parameter storing the mass of each crop of crystals in grams.

```
[in]: def per_yield(g_theor, *g_crops):
        g_total = sum(g_crops)
        percent_yield = 100 * (g_total / g_theor)
        return percent_yield

[in]: per_yield(1.32, 0.50, 0.11, 0.27)

[out]: 66.66666666666666
```

Interestingly, depending upon how you write the internals of the function, the var-positional argument is not strictly necessary for the function to work. In this case, because `sum()` function returns 0 if no arguments are passed to it, the `per_yield` function still works with no error returned.

```
[in]: per_yield(1.32)  
[out]: 0.0
```

## 2.7.2 Variable Keyword Arguments

Similarly, an unspecified number of keyword arguments can also be accepted by a Python function using *var-keyword arguments*. In this case, the user not only dictates the number of arguments but also picks the variable names. The user-defined variables and values are stored in a local dictionary as key:value pairs. As an example, we can write a function that calculates the molar mass of a compounds based on the number and type of elements it contains. It is certainly possible to write a function with every chemical element as a keyword argument, but this gets absurd with so many chemical elements to choose from. Instead, we can use a var-keyword parameter as demonstrated below. The var-keyword argument is indicated with a `**` before the variable name. The function below is only designed to work with the first nine elements for brevity.

```
[in]: def mol_mass(**elements):  
    m = {'H':1.008, 'He':4.003, 'Li':6.94, 'Be':9.012,  
          'B':10.81, 'C':12.011, 'N':14.007, 'O':15.999,  
          'F':18.998}  
  
    masses = [] # mass total from each element  
    for key in elements.keys():  
        masses.append(elements[key] * m[key])  
  
    return sum(masses)
```

Let us test this function by calculating the molar mass of caffeine which has a molecular formula of C<sub>8</sub>H<sub>10</sub>N<sub>4</sub>O<sub>2</sub>.

```
[in]: mol_mass(C=8, H=10, N=4, O=2)  
[out]: 194.194
```

The user experience would be the same if we wrote the function to accept keyword arguments with default values of zero, but it is sometimes more convenient for the person writing the code to design the function to accept var-keyword arguments.

### 2.7.3 Recursive Functions

Functions can call other functions. This is probably not surprising as we have already seen functions call `math.sqrt()` and `append()`, but what may be surprising is that Python allows a function to call itself. This is known as a *recursive function*.

If we want to write a function that calculates the remaining mass of radioactive materials after a given number of half-lives, this can be accomplished using a `for` or `while` loop, but it can also be accomplished recursively. We start by having the function divide the provided mass (`mass`) in half and then decrement the number of half-lives (`hl`) by one. This is the core component of the function. If `hl` is zero, the function is done and returns the mass. If not, the function calls itself again with the remaining mass and number of half-lives. This is the recursive part. The second time the function is run, the mass is again halved and the half-lives decremented by one, and the number of half-lives is again checked.

```
[in]: def half_life(mass, hl=1):
    '''(float, hl=int) -> float

    Takes in mass and number of half-lives and returns
    remaining mass of material. Half-lives need to be integer
    values.
    '''

    mass /= 2
    hl -= 1

    if hl == 0:
        return mass
    else:
        return half_life(mass, hl=hl)

[in]: half_life(4.00, hl=2)
[out]: 1.00

[in]: half_life(4.00, hl=4)
[out]: 0.25
```

It works! In the second example above, the `half_life()` function is run four times because the function called itself an additional three times. What happens if we feed the function 1.5 half-lives? Like a `while` loop with a faulty termination condition, this function will keep going because `hl` never equals zero. Luckily, Python has a safeguard that stops

recursive functions from running more than a thousand iterations, but this is still a problem. We can protect against this issue by doing a check at the start of the function to ensure an integer is provided.

```
[in]: def half_life(mass, hl=1):
    '''(float, hl=int) -> float

    Takes in mass and number of half-lives and returns
    remaining mass of material. Half-lives need to be
    integer values.
    '''

    if type(hl) != int:
        print('Invalid hl. Integer required.')
        return None

    mass /= 2
    hl -= 1

    if hl <= 0:
        return mass
    else:
        return half_life(mass, hl=hl)

[in]: half_life(4.00, hl=1.5)
[out]: Invalid hl. Integer required.
```

While getting an error message is not what anyone likes to see, this is a good thing. It is better for the code to generate an error and not work than to run away uncontrollably or return an incorrect answer.

As a final note on recursive functions, you may have noticed that you could just as easily have accomplished the above task with a `while` or `for` loop. Recursive functions can usually be avoided, but once in a while a recursive function will substantially simplify your code. It is a good technique to have in your back pocket for the moment you need it, but you will not likely use them often.

## Further Reading

The official Python website is the ultimate authority for documentation on the Python programming language and is well written. There are also numerous books available on the subject both free and otherwise. Below are a few examples. There is an abundance of other free resources such as YouTube<sup>23</sup> videos and StackOverflow<sup>24</sup> boards for people looking for more information.

1. Python Documentation Page. <https://www.python.org/doc/>.
2. Reitz, K.; Schlusser, T. *The Hitchhiker's Guide to Python: Best Practices for Development*, O'Reilly: Sebastopol, CA, 2016.
3. Downey, Allen B. *Think Python* Green Tea Press 2012.  
<http://greenteapress.com/wp/think-dsp/>. (also available for purchase by O'Reilly Media)

## Exercises

### Syntactic Sugar

1. Generate a list containing the natural logs of integers from  $2 \rightarrow 23$  (including 23) using `append` and then again using *list comprehension*.
2. Write a function, using *augmented assignment*, that takes in a starting *xyz* coordinates of an atom along with how much the atom should move on each axis and returns the final coordinates. The docstring is for this function is below.

```
def trans(coord, x=0, y=0, z=0):  
    '''((x, y, z), x=0, y=0, z=0) -> (x, y, z)  
    '''
```

3. Generate a function that returns the square of a number using a *lambda function*. Assign it to a variable for reuse and test it.

### Dictionaries & Sets

4. Generate a *dictionary* called `aacid` that converts single-letter amino acid abbreviations to the three-letter abbreviations. You will need to look up the abbreviations from a textbook or online resource.

---

23 YouTube Website. <https://www.youtube.com/>.

24 StackOverflow Website. <https://stackoverflow.com/>.

5. For the following two sets:

```
acids1 = {'HCl', 'HNO3', 'HI', 'H2SO4'}
acids2 = {'HI', 'HBr', 'HClO4', 'HNO3'}
```

- a) Generate a new set with all items from acids1 and acids2.
- b) Generate a new set with the overlap between acids1 and acids2
- c) Add a new item HBrO<sub>3</sub> to acids1.
- d) Generate a new set with items from either set but not in both

## Python Modules

6. Use a `for` loop and `listdir()` method to print the name of every file in a folder on your computer. Compare what Python prints out to what you see when looking in the folder. Does Python print any files that you do not see in the file browser?
7. Use the `random` module for the following.
- a) Generate 10 random integers from 0 → 9 and calculate the mean of these values. What is the theoretical mean for this data set?
  - b) Generate 10,000 random integers from 0 → 9 and calculate the mean of these values. Is this mean closer or further than the mean from part a? Rationalize your answer. *Hint:* look up the “law of large numbers” for help.
8. The following code generates five atoms at random coordinates in 3D space. Write a Python script that calculates the distance between each pair of atoms and returns the shortest distance. The `itertools` module might be helpful here. See section 1.9.1 for help calculating distance.

```
from random import randint
atoms = []
for a in range(5):
    x, y, z = randint(0,20), randint(0,20),
    randint(0,20)
    atoms.append((x,y,z))
```

## Zip & Enumeration

9. Zipping lists
- a) Generate a list of the first ten atomic symbols on the periodic table.

- b) Convert the list from part a to (atomic number, symbol) pairs.
10. Zip together two lists containing the symbols and names of the first six elements of the periodic table and convert them to a dictionary using the `dict()` function. Test the dictionary by converting Li to its name.
  11. Write a Python script that goes through a collection of random integers from  $0 \rightarrow 20$  and returns a list of index values for all values larger than 20. Start by generating a list of random integers and combine them with their index values using either `zip()` or `enumerate()`.

## **Advanced Function Arguments**

12. Write a function that calculates the distance between the origin and a point in any dimensional space (1D, 2D, 3D, etc...) by allowing the function to take any number of coordinate values (e.g.,  $x$ ,  $xy$ ,  $xyz$ , etc...). Your function should work for the following tests.

```
[in]: dist(3)
[out]: 3

[in]: dist(1,1)
[out]: 1.4142135623730951

[in]: dist(3, 2, 1)
[out]: 3.7416573867739413
```

13. Below is a function calculates the theoretical number of remaining protons( $p$ ) and neutrons( $n$ ) remaining after  $x$  alpha decays. Convert this function to a recursive function. *Hint:* start by removing the `for` loop and replace it with an `if` statement.

```
def alpha_decay(x, p, n):
    '''(alpha decays(x), protons(int), neutrons(int)) ->
       prints protons and neutrons remaining
```

Takes in the number of alpha decays( $x$ ), protons( $p$ ), and number of neutrons( $n$ ) and all as integers and prints the final number of protons and neutrons.

```
# tests
>> alpha_decay(2, 10, 10)
```

```
6  protons and 6  neutrons remaining.  
>> alpha_decay(1, 6, 6)  
4  protons and 4  neutrons remaining.  
'''  
  
for decay in range(x):  
    p -= 2  
    n -= 2  
  
    print(str(p), ' protons and', str(n), ' neutrons  
          remaining.')  
    
```

# *Chapter 3*

## ***Plotting with Matplotlib***

Data visualization is an important part of scientific computing both in analyzing your data and in supporting your conclusions. There are a variety of plotting libraries available in Python, but the one that stands out from the rest is matplotlib. Matplotlib is a core SciPy library because it is powerful and can generate nearly any plot a user may need. The main drawback is that it is often verbose. That is to say, anything more complex than a very basic plot may require a few lines of boilerplate code to create. This chapter introduces plotting with matplotlib.

Before the first plot can be created, we must first import matplotlib using the below imports. The first line imports the `pyplot` module which does much of the basic plotting in matplotlib. While the `plt` alias is not required, it is a common convention in the SciPy community and is highly recommended as it will save you a considerable amount of typing. The second line is not really Python so much as a Jupyter notebook command. It tells Jupyter to display the matplotlib figures directly in the notebook instead of a separate window.<sup>25</sup> This text assumes that you are using inline plotting.<sup>26</sup>

```
[in]: import matplotlib.pyplot as plt  
      %matplotlib inline
```

In all the examples below, simply calling a plotting function in a Jupyter notebook will automatically make the plot appear in the Jupyter notebook. However, if you choose to use matplotlib in some other environment, it is often necessary to also execute the following

---

<sup>25</sup> If you opt to leave this inline command out, a separate window should appear displaying your plot. It often appears *behind* the Jupyter notebook window, so you will probably have to go looking for it.

<sup>26</sup> To turn off inline, type `% matplotlib` again without the `inline` part.

line to make the plot appear. This can also be done in Jupyter, but it is not shown in the rest of this chapter because Jupyter does not require it.

```
[in]: plt.show()
```

## 3.1 Basic Plotting

Before creating our first plot, we need some data to plot. The following equation defines the wave function ( $\Psi$ ) for the 3s atomic orbital of hydrogen with respect to atomic radius ( $r$ ) in Bohrs ( $a_0$ ).

$$\Psi_{3s} = 2\sqrt{3}(2r^{2/9} - 2r + 3)e^{-r/3}/27$$

We will generate points on this curve using a method called list comprehension covered in section 2.1.2. If you choose to plot something else, just make two lists of the same length containing  $x$  and  $y$  values.

```
[in]: import math

[in]: def orbital_3S(r):
        wf = 2*math.sqrt(3)*(2*r**2/9 - 2*r + 3)*math.exp(-r/3)/27
        return wf

[in]: r = [num / 4 for num in range(1, 100, 3)]
      psi_3s = [orbital_3S(num) for num in r]
```

### 3.1.1 First Plot

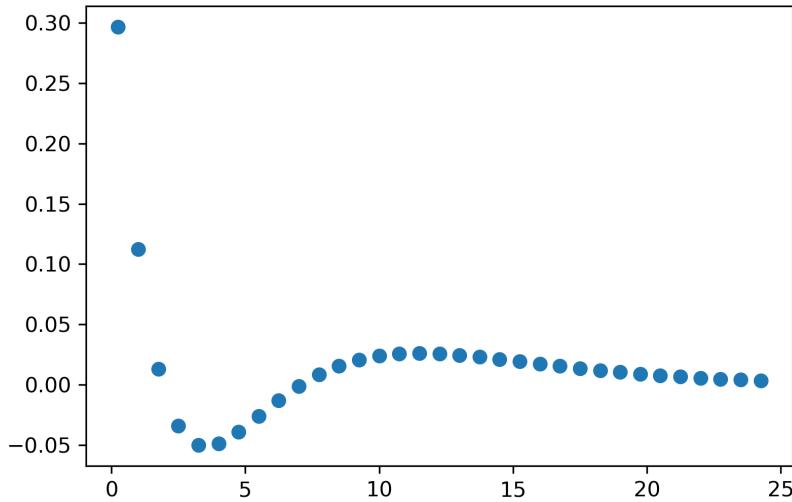
To visualize the 3s wave functions, we will call the `plot()` function, which is a general-purpose function for plotting.<sup>27</sup> The `r` and `psi` data are fed into it as positional arguments as the  $x$  and  $y$  variables, respectively. Because of the `%matplotlib inline` command, the output displays directly in the Jupyter notebook

```
[in]: plt.plot(r, psi_3s)

[out]:
```

---

<sup>27</sup> Along with a plot, matplotlib also gives a value that looks like `0x117786a20`. This is a pointer to where in computer memory the information is stored. This is not particularly important for most users as Python handles all of this behind the scenes.



By default, matplotlib creates a scatter plot using blue as the default color. This can be modified if a blue circles are not to your taste. If the `plot()` function is only provided a single argument, matplotlib assumes the data are the `y`-values and plots them against their indices.

### 3.1.2 Markers and Color

To change the color and markers, you can add a few extra arguments: `marker`, `linestyle`, and `color`. All of these are keyword arguments take strings. `marker` allows the user to choose from a list of markers (Table 3.1). The `linestyle` argument (Table 3.2) determines if a line is solid or the type of dashing that occurs, and the `color` argument (Table 3.3) allows the user to dictate the color of the plot. If an empty string is provided to `linestyle` or `marker`, no line or marker, respectively, is shown in the plot. See the matplotlib website (<https://matplotlib.org/>) for a more complete list of styles.

**Table 3.1** Common Matplotlib Marker Styles

| Argument | Description |
|----------|-------------|
| 'o'      | circle      |
| '*'      | star        |
| 'p'      | pentagon    |
| '^'      | triangle    |
| 's'      | square      |

**Table 3.2** Common Matplotlib Line Styles

| Argument | Description |
|----------|-------------|
| '-'      | solid       |
| '--'     | dashed      |
| '-.'     | dash-dot    |
| '::'     | Dotted      |

**Table 3.3** Common Matplotlib Colors

| Argument | Description |
|----------|-------------|
| 'b'      | blue        |
| 'r'      | red         |
| 'k'      | black (key) |
| 'g'      | green       |
| 'm'      | magenta     |
| 'c'      | cyan        |
| 'y'      | yellow      |

There are numerous other arguments that can be placed in the plot command. A few common, useful ones are shown below in Table 3.4.

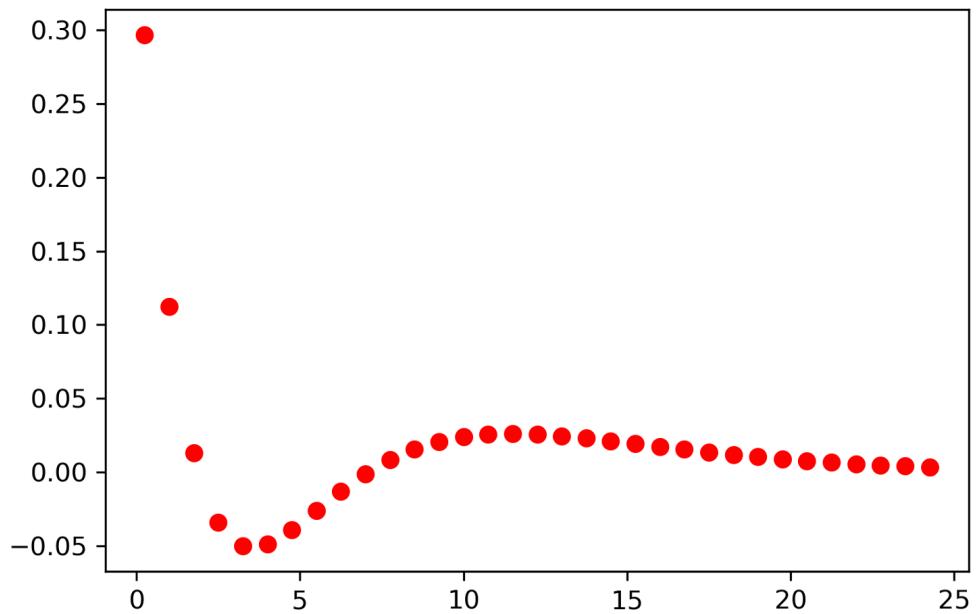
**Table 3.4** A Few Common plot Keyword Arguments

| Argument               | Description       |
|------------------------|-------------------|
| linestyle or ls        | Line style        |
| marker                 | Marker style      |
| linewidth or lw        | Line width        |
| color or c             | Line color        |
| markeredgecolor or mec | Marker edge color |
| markerfacecolor or mfc | Marker color      |
| markersize or ms       | Marker size       |

Now that you have seen the keyword argument approach which allows for the fine tuning of plots, there is a shortcut for most basic plots. The plot function can take a third, positional argument which makes plotting a lot quicker. If you place a string with a marker style and/or line style, you can adjust the color and markers without the full keyword arguments. This approach does not allow the user as much control as the keyword arguments, but it is popular because of the ease of use.

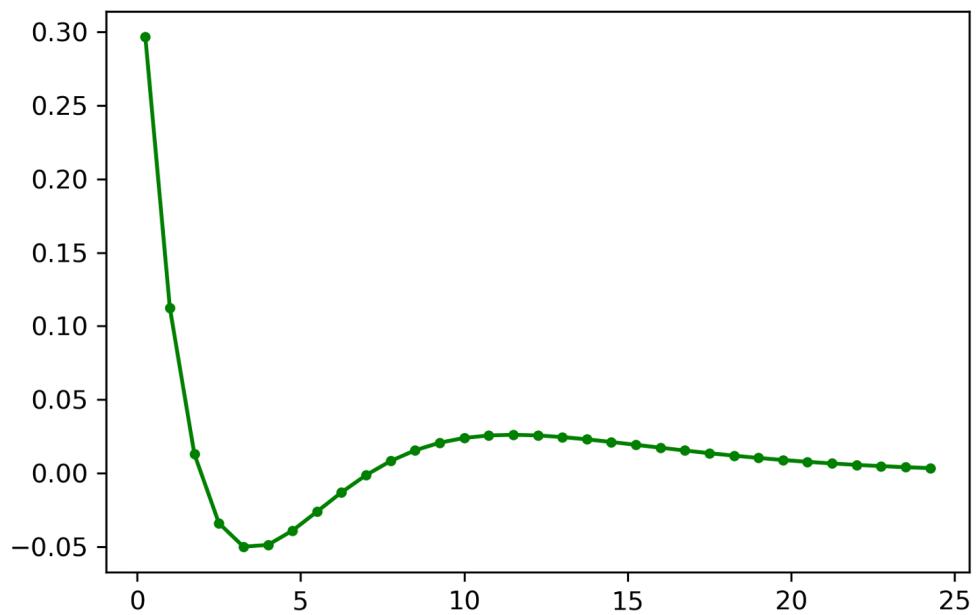
```
[in]: plt.plot(r, psi_3s, 'ro')
```

```
[out]:
```



```
[in]: plt.plot(r, psi_3s, 'g.-')
```

```
[out]:
```

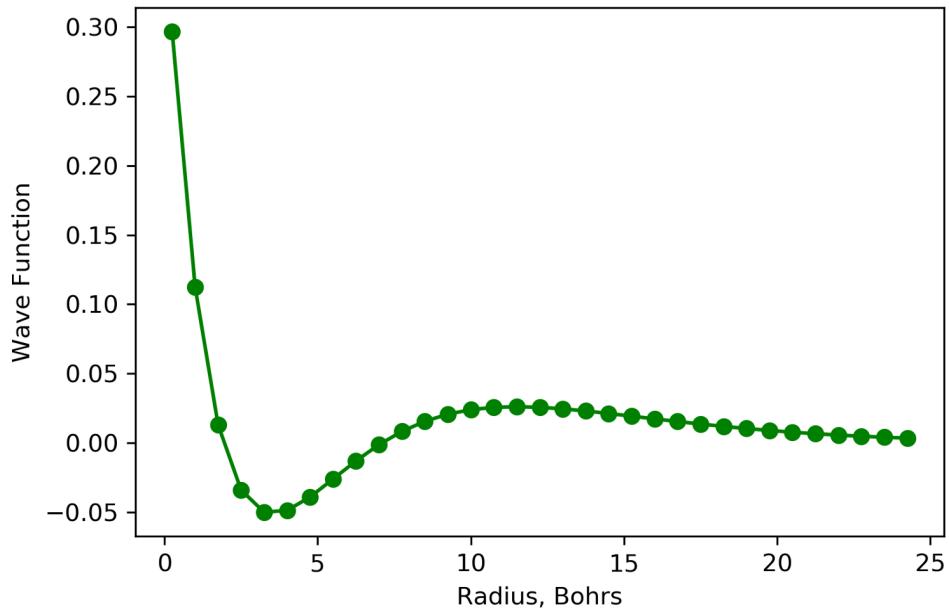


### 3.1.3 Labels

It is often important to label the axes of your plot. This is accomplished using the `plt.xlabel()` and `plt.ylabel()` functions which are placed on different lines as the `plt.plot()` function. Both functions take strings.

```
[in]: plt.plot(r, psi_3s, 'go-')
       plt.xlabel('X Values')
       plt.ylabel('Y Values')
```

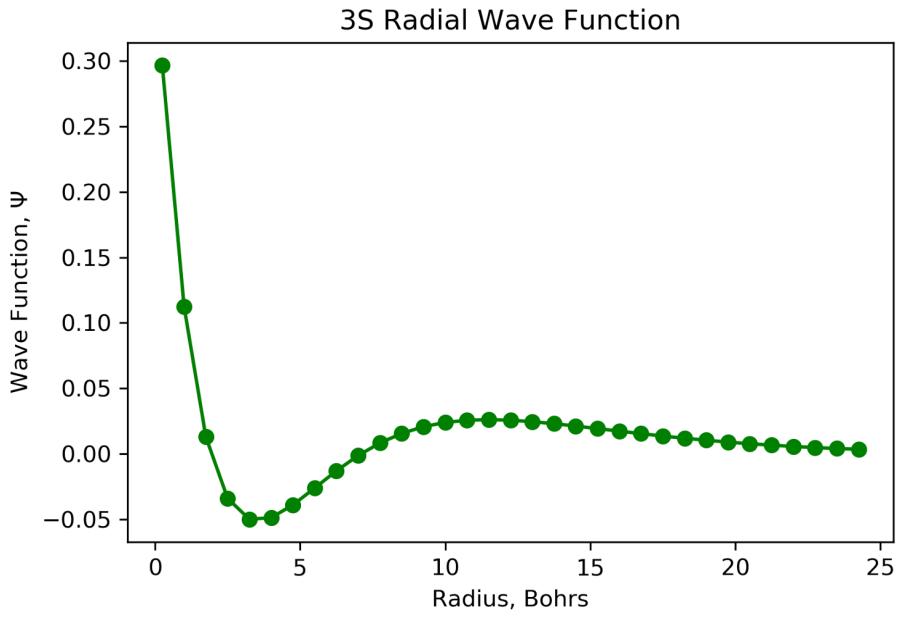
[out] :



In the event you want a title at the top of your plots, you can add one using the `plt.title()` argument. To add symbols to the axes, this can be done using Latex commands which are used below, but discussion of Latex is beyond the scope of this chapter.

```
[in]: plt.plot(r, psi_3s, 'go-')
       plt.xlabel('Radius, Rohrs')
       plt.ylabel('Wave Function, $\Psi$')
       plt.title('3S Radial Wave Function')
```

[out] :

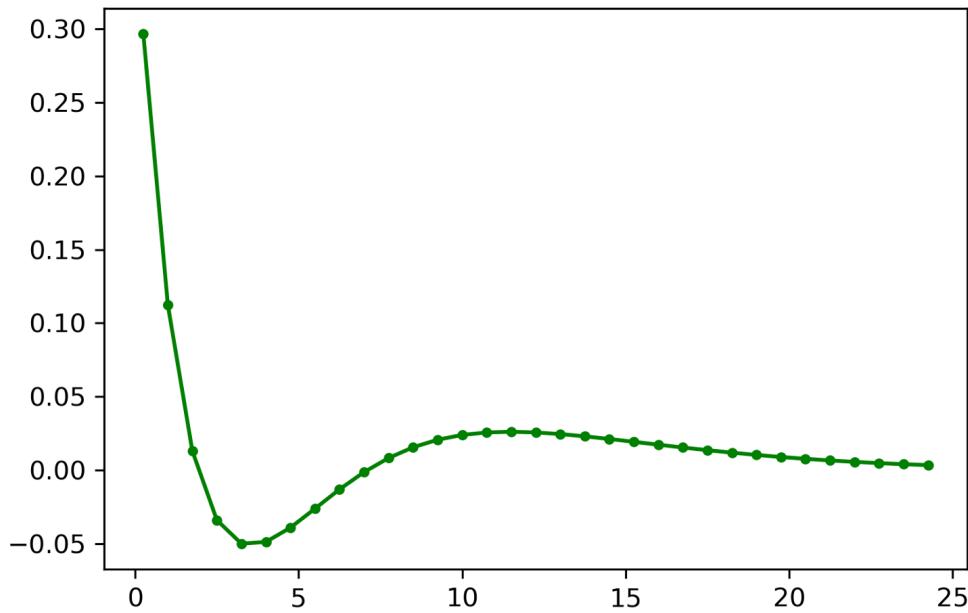


### 3.1.4 Saving Figures

A majority of matplotlib usage is to generate figures in a Jupyter notebook. However, there are times when it is necessary to save the figures to files for a manuscript, report, or presentation. In these situations, you can save your plot using the `plt.savefig()` function which takes a few arguments. The first and only required argument is the name of the output file as a string. Following this, the user can also choose the resolution in dots per inch using the `dpi` keyword argument. Finally, there are a number of file formats supported by the `plt.savefig()` functions including PNG, TIF, JPG, PDF, SVG, among others. The formats can be selected using the `format` argument which also takes a string, and if no format is explicitly chosen, matplotlib defaults to PNG.

```
[in]: plt.plot(r, psi_3s, 'g.-')
       plt.savefig('my_image.png', format='PNG', dpi=600)

[out]:
```



If you do not see your output image file, be sure that you are looking in the current working directory, which is likely the same folder as your Jupyter notebook.

## 3.2 Plotting Types

Matplotlib supports a wide variety of plotting types including scatter plots, bar plots, histograms, pie charts, stem plots, and many others. A few of the most common ones are introduced below. For additional plotting types, see the matplotlib website (<https://matplotlib.org/>).

### 3.2.1 Bar Plots

Bar plots, despite looking very different, are quite similar to scatter plots. They both show the same information except that instead of the height of a marker showing the magnitude of a  $y$ -value, it is represented by the height of a bar. Bar plots are generated using the `plt.bar()` function. Similar to the `plt.plot()` function, the bar plot takes  $x$  and  $y$  values as positional arguments, and if only one argument is given, the function assumes it is the  $y$ -variable and plots the values with respect to the index values.

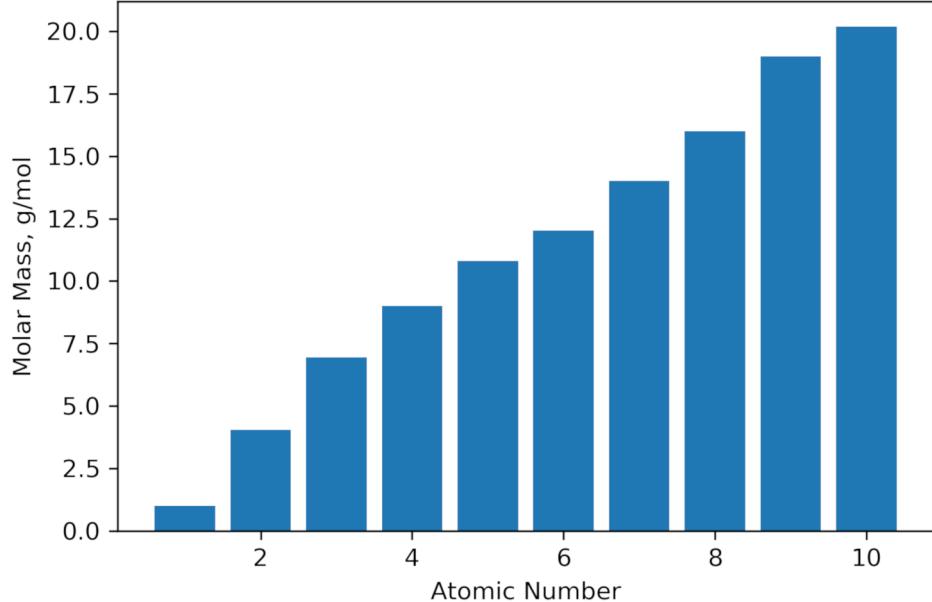
```
[in]: AN = [x + 1 for x in range(10)]
MW = [1.01, 4.04, 6.94, 9.01, 10.81, 12.01, 14.01,
       16.00, 19.00, 20.18]

[in]: plt.bar(AN, MW)
```

```

plt.xlabel('Atomic Number')
plt.ylabel('Molar Mass, g/mol')

[out]:
```



The bar plot characteristics can be adjusted like most other types of plots in matplotlib. The main arguments you will probably want to adjust are color and width, but some other arguments are provided in Table 3.5. The color arguments are consistent with the `plt.plot()` colors from earlier. The error bar arguments can take either a single value to display homogenous error bars on all data points or can take a multi-element object (e.g., a list or tuple) containing the different margins of uncertainty for each data point.

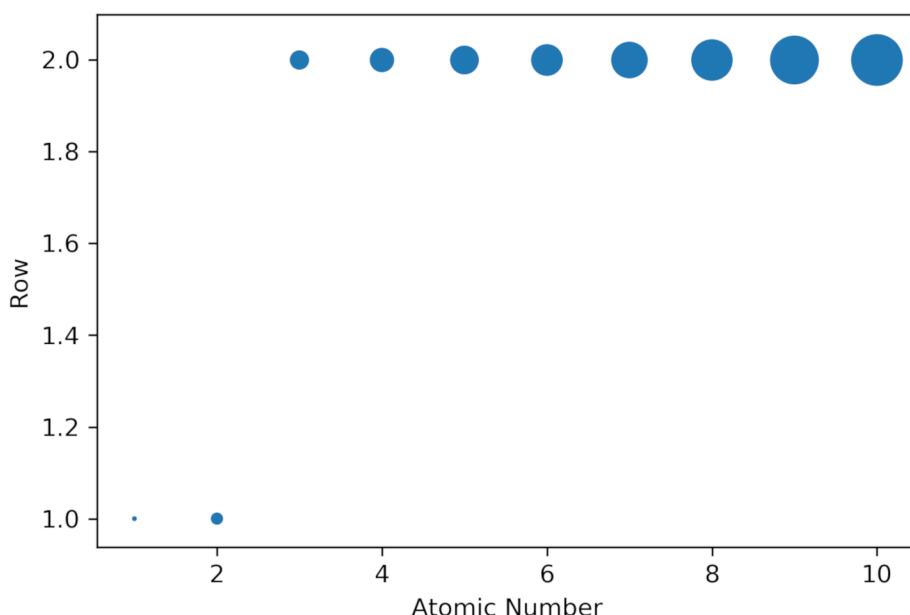
**Table 3.5** A Few Common `plot` Keyword Arguments

| Argument               | Description        |
|------------------------|--------------------|
| <code>width</code>     | Bar width          |
| <code>color</code>     | Bar color          |
| <code>edgecolor</code> | Bar edge color     |
| <code>xerr</code>      | X error bar        |
| <code>yerr</code>      | Y error bar        |
| <code>capsize</code>   | Caps on error bars |

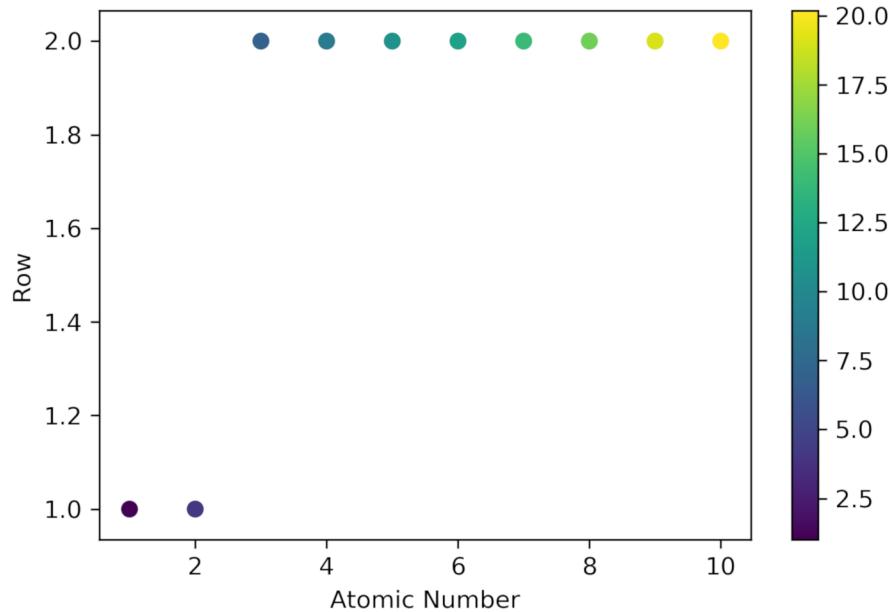
### 3.2.2 Scatter Plots

We have already generated scatter plots using the `plt.plot()` function, but they can also be created using the `plt.scatter()` function. The latter is somewhat redundant, but unlike `plt.plot()`, `plt.scatter()` allows for different size, shape, and color of individual markers.

```
[in]: row = [1, 1, 2, 2, 2, 2, 2, 2, 2, 2]  
[in]: plt.scatter(AN, row, s=[weight**2 for weight in MW])  
      plt.xlabel('Atomic Number')  
      plt.ylabel('Row')  
  
[out]:
```



```
[in]: plt.scatter(AN, row, c=MW)  
      plt.xlabel('Atomic Number')  
      plt.ylabel('Row')  
      plt.colorbar() # adds color key on the right  
  
[out]:
```

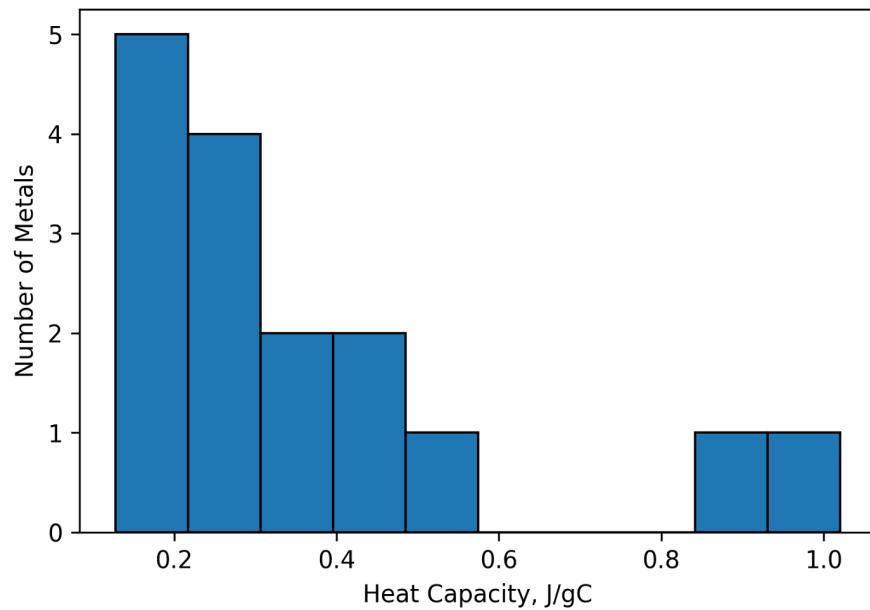


### 3.2.3 Histogram Plots

Histograms display bars representing the frequency of values in a particular data set. Unlike bar plots, the width of the bars in a histogram plot is meaningful as each bar represents the number of  $x$ -values that fall within a particular range. A histogram plot can be generated using the `plt.hist()` function which does two things. First, the function takes the data provided and sorts them into equally-spaced groups, called *bins*; and second, it plots the totals in each bin. For example, we have a list, `Cp`, of specific heat capacities for various metals in J/g·°C, and we want to visualize the distribution of the specific heat capacities.

```
[in]: Cp = [0.897, 0.207, 0.231, 0.231, 0.449, 0.385, 0.129,
          0.412, 0.128, 1.02, 0.140, 0.233, 0.227, 0.523,
          0.134, 0.387]
plt.hist(Cp, bins=10)
plt.xlabel('Heat Capacity, J/gC')
plt.ylabel('Number of Metals')

[out]:
```

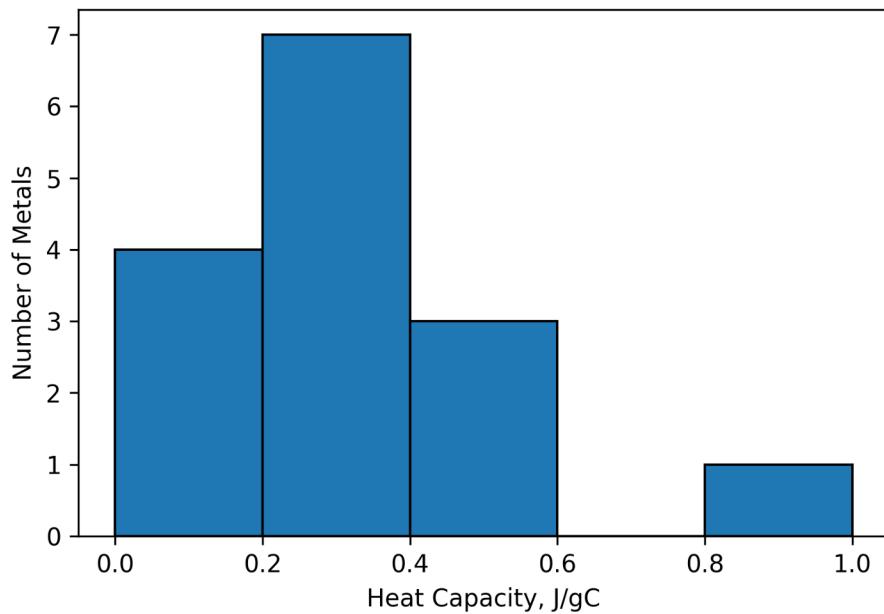


From the plot above, we can see that a large number of heat capacities reside in the area of 0.1-0.5 J/g·°C and none fall in the 0.6-0.8 J/g·°C range.

The two main arguments for the `plt.hist(data, bins=)` function are `data` and `bins`. The `bins` argument can be either a number of evenly-spaced bins in which the data is sorted, like above, or it can be a list of bin edges like below. The function automatically determines which you are providing based on your input.

```
[in]: plt.hist(Cp, bins=[0, 0.2, 0.4, 0.6, 0.8, 1.0])
       plt.xlabel('Heat Capacity, J/gC')
       plt.ylabel('Number of Metals')
```

[out] :



Providing the histogram function bin edges offers far more control to the user, but writing out a list can be tedious. As an alternative, the histogram function also accepts bin edges as `range()` objects. Unfortunately, Python's built-in `range()` function only generates values with integer steps. As an alternative, you can use list comprehension from chapter 2 or use NumPy's `np.arange()` function from chapter 4 which does allow non-integer step sizes.

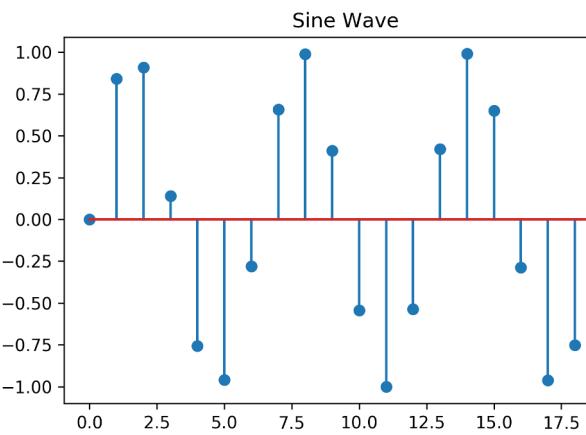
### 3.2.4 Other Plotting Types

There are a variety of other two dimensional plotting types available in the `matplotlib` library including stem, step, pie, polar,<sup>28</sup> box plots, and contour plots. Below is a table of a few worth knowing about along with the code that created them. See the `matplotlib` website for further details. Many Python library websites, including `matplotlib`'s, contain a gallery page which showcases examples of what can be done with that library. It is recommended to brows these pages when learning a new library.

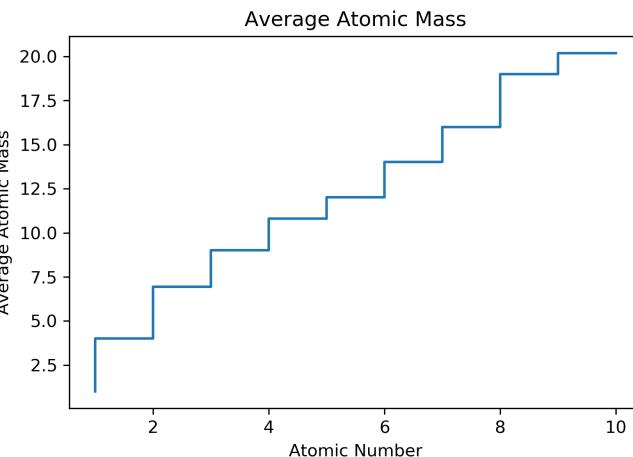
---

<sup>28</sup> The polar plot is that of the  $p_z^2$  orbital. See McQuarrie, D. A. *Quantum Chemistry*, University Science Books: Sausalito, CA, 1983, pp. 235

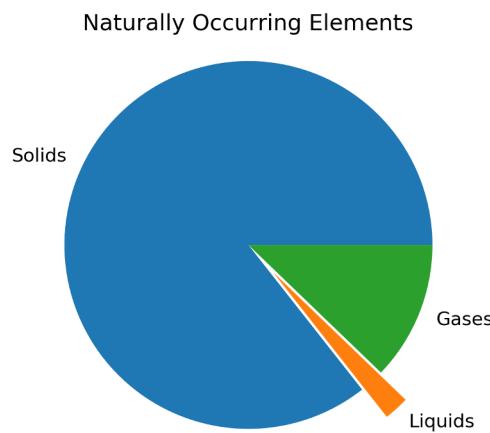
**Table 3.6 Other Matplotlib Plotting Types**



```
x = range(20)
y = [math.sin(num) for num in x]
plt.stem(x, y)
plt.title('Sine Wave')
```

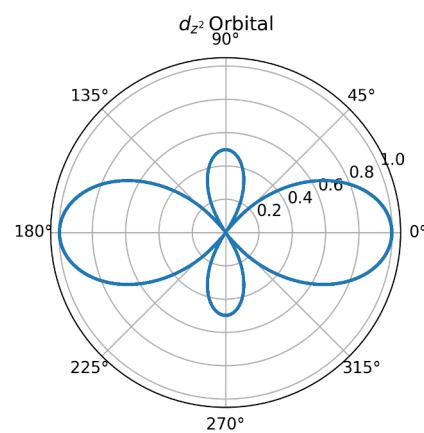


```
AN = range(1, 11)
mass_avg = [1.01, 4.00, 6.94, 9.01,
10.81, 12.01, 14.01, 16.00, 19.00,
20.18]
plt.step(AN, mass_avg)
plt.title('Average Atomic Mass')
plt.xlabel('Atomic Number')
plt.ylabel('Average Atomic Mass')
```



```
labels = ['Solids', 'Liquids',
'Gases']
percents = (85.6, 2.2, 12.2)

plt.title('Naturally Occurring
Elements')
plt.pie(percents, labels=labels,
explode=(0, 0.2, 0))
plt.axis('equal')
```



```
import numpy as np

theta = np.arange(0, 360, 0.1)
r = [abs(0.5 * (3 * math.cos(num))**2
- 1)) for num in theta]

plt.polar(theta, r)
plt.title(r'$d_{z^2}$ \ , $' +
'Orbital')
```

### 3.3 Overlaying Plots

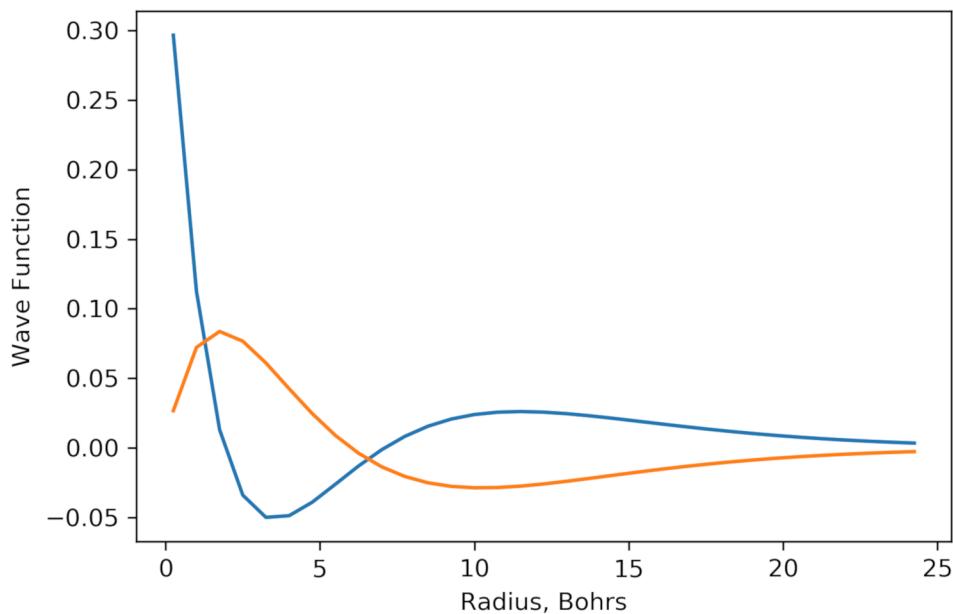
It is often necessary to plot more than one set of data on the same axes, and this can be accomplished in two ways with matplotlib. The first is to call the plotting function twice in the same Jupyter code cell. Matplotlib will automatically place both plots in the same figure and scale it appropriately to include all data. Below, data for the wave function for the 3p hydrogen orbital is generated similar to the 3s earlier, so now the wave functions for both the 3s and 3p orbitals can be plotted on the same set of axes.

```
[in]: def orbital_3P(r):
        wf = (math.sqrt(6)*r*(4-(2/3)*r)*math.e**(-r/3))/81
        return wf

[in]: r = [num / 4 for num in range(1, 100, 3)]
psi_3p = [orbital_3P(num) for num in r]

[in]: plt.plot(r, psi_3s)
plt.plot(r, psi_3p)
plt.xlabel('Radius, Bohrs')
plt.ylabel('Wave Function')

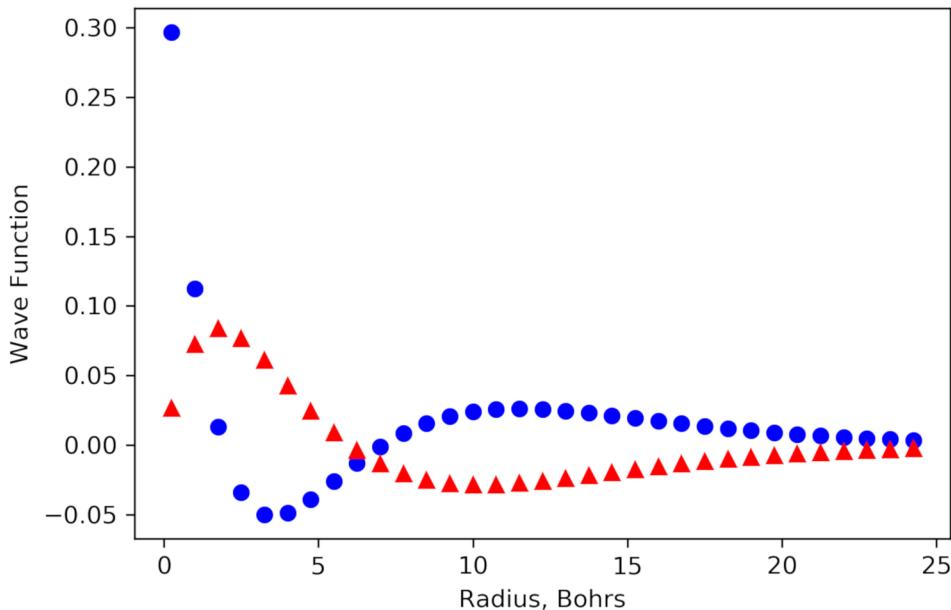
[out]:
```



The second approach is to include both sets of data in the same plotting command as is shown below. Matplotlib will assume that each new non-keyword is a new set of data and that the positional arguments are associated with the most recent data.

```
[in]: plt.plot(r, psi_3s, 'bo', r, psi_3p, 'r^')
       plt.xlabel('Radius, Bohrs')
       plt.xlabel('Radius, Bohrs')
       plt.ylabel('Wave Function')
```

[out] :



In the second plot above, `r, psi_3s, 'bo'` are the data and style for the first set of data while `r, psi_3p, 'r^'` are the data and plotting style for the second.

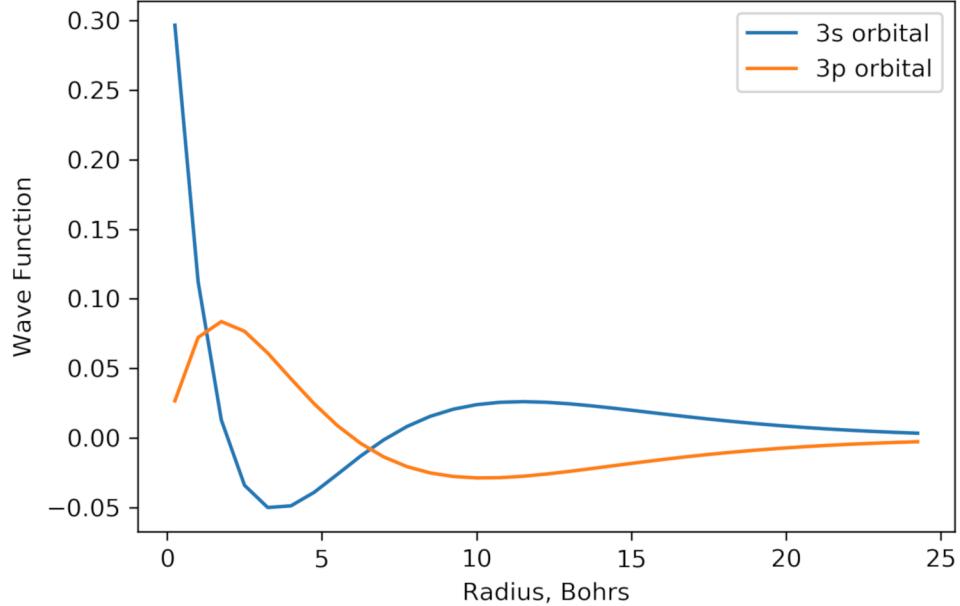
One issue that quickly arises with multifigure plots is identifying which symbols belong to which data. Matplotlib allows the user to add a legend to the plot. The user first needs to provide a label for each data set using the `label=` keyword argument. Finally, calling `plt.legend()` causes the labels to be displayed on the plot. The default is for matplotlib to place the legend where it decides is the optimal location, but this behavior can be overridden by adding a keyword `loc=` argument.<sup>29</sup>

```
[in]: plt.plot(r, psi_3s, label='3s orbital')
       plt.plot(r, psi_3p, label='3p orbital')
```

<sup>29</sup> See the matplotlib documentation page [https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.legend.html](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.legend.html) for a complete list of location arguments.

```
plt.xlabel('Radius, Bohrs')
plt.ylabel('Wave Function')
plt.legend()

[out]:
```



## 3.4 Multifigure Plots

To generate multiple, independent plots in the same figure, a few more lines of code are required to describe the dimensions of the figure and which plot goes where. Once you get used to it, it is fairly logical but requires multiple lines of code. There are two general methods for generating multifigure plots outlined below. The first is a little quicker, but the second is certainly more powerful and gives the user access to extra features. Whichever method you choose to adopt, just be aware that you will likely see the other method at times as both are common.

### 3.4.1 First Approach

In the first method, we first need to generate the figure using the `plt.figure()` command. For every subplot, we first need to call `plt.subplot(rows, columns, plot_number)`. The first two values are the number of rows and columns in the figure, and the third number is which subplot you are referring to. For example, we will generate a figure with two plots side-by-side. This is a one-by-two figure (i.e., one row and two

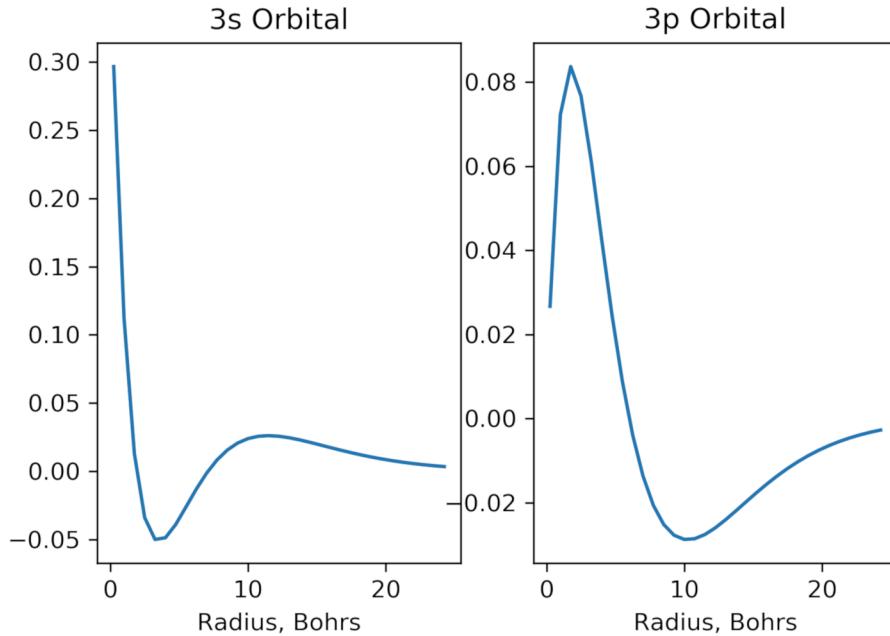
columns). Therefore, all subplots will be defined using `plt.subplot(1, 2, plot_number)`. The `plot_number` indicates the subplot with the first subplot being 1 and the second subplot being 2. The numbering always runs left-to-right and top-to-bottom.

[in]: `plt.figure()`

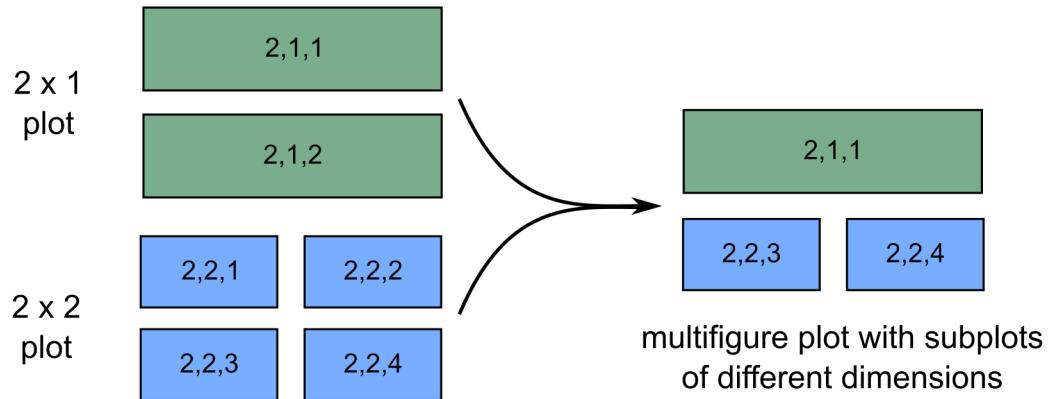
```
plt.subplot(1,2,1) # first subplot
plt.plot(r, psi_3s)
plt.xlabel('Radius, Bohrs')
plt.title('3s Orbital')

plt.subplot(1,2,2) # second subplot
plt.plot(r, psi_3p)
plt.xlabel('Radius, Bohrs')
plt.title('3p Orbital')
```

[out] :



The values in the `plt.subplot()` command may seem redundant. Why are the dimensions for the figure repeatedly defined instead of just once? The answer is that subplots with different dimensions can be created in the same figure (Figure 3.1). In this example, the top subplot dimension is created as if though it is the first subplot in a  $2 \times 1$  figure. The bottom two subplot dimensions are created as if they are the third and fourth subplots in a  $2 \times 2$  figure.



**Figure 3.1** Multifigure plots with subplots of different dimensions (right) describe each subplot dimension as if it were part of a plot with equally sized subplots (left).

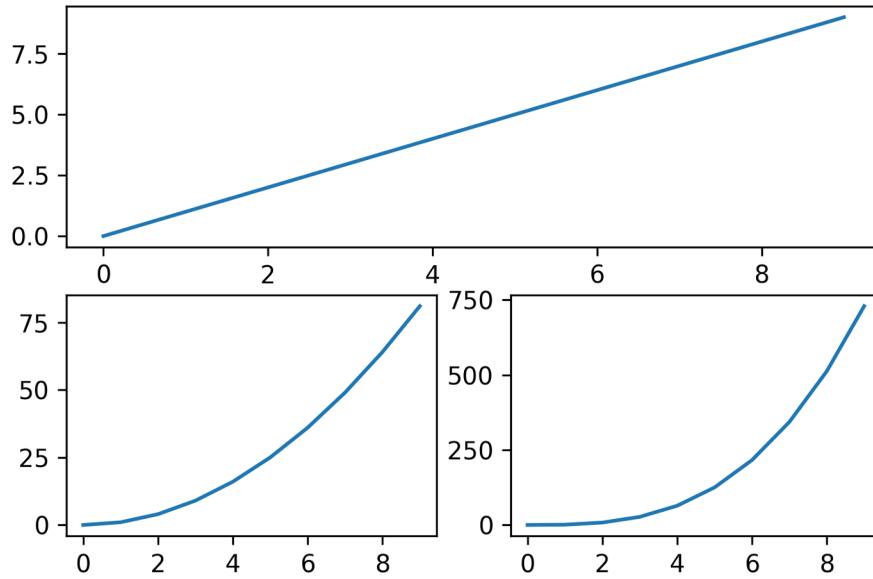
```
[in]: plt.figure()

plt.subplot(2,1,1) # top subplot
plt.plot(x, x)

plt.subplot(2,2,3) # bottom left subplot
plt.plot(x, y)

plt.subplot(2,2,4) # bottom right subplot
plt.plot(x, y2)

[out]:
```



Lastly, there are times when the axis labels for multiple subplots will overlap. If this happens, simply add `plt.tight_layout()` at the very end to fix this.

### 3.4.2 Second Approach

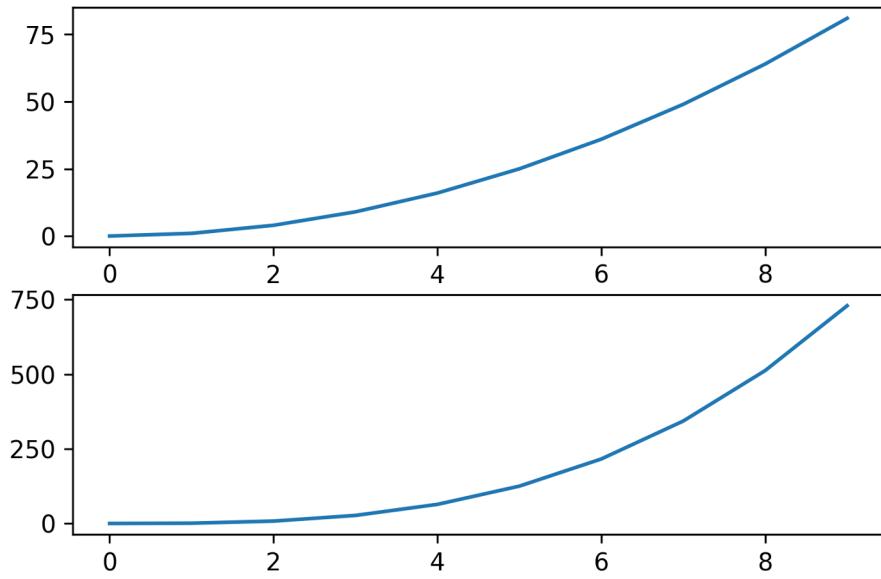
The second method is somewhat similar to the first except that it more explicitly creates and links subplots, called *axes*. To create a figure with subplots, we first need to generate the overall figure using the `plt.figure()` command again, and we also need to attach it to a variable so that we can explicitly assign axes to it. To create each subplot, use the `plt.add_subplot(rows, columns, plot_number)` command. The arguments in the `plt.add_subplot` command are the same as `plt.subplot()` in section 3.4.1. After an axis has been created as part of the figure, call the `plot()` function preceded by the axis variable name as demonstrated below.

```
[in]: fig = plt.figure()

# top plot
ax1 = fig.add_subplot(2,1,1)
ax1.plot(x, y)

# bottom plot
ax2 = fig.add_subplot(2,1,2)
ax2.plot(x, y2)
```

[out]:



### 3.5 3D Plotting

To plot in 3D, we will use the approach outlined in section 3.4.2 with two additions. First, import `Axes3D` from `mpl_toolkits.mplot3d` as shown below. Second, make the plot 3D by adding `projection='3D'` to the `plt.figure()` command. After that, it is analogous to the two dimensional plots above except `x`, `y`, and `z` data are provided.

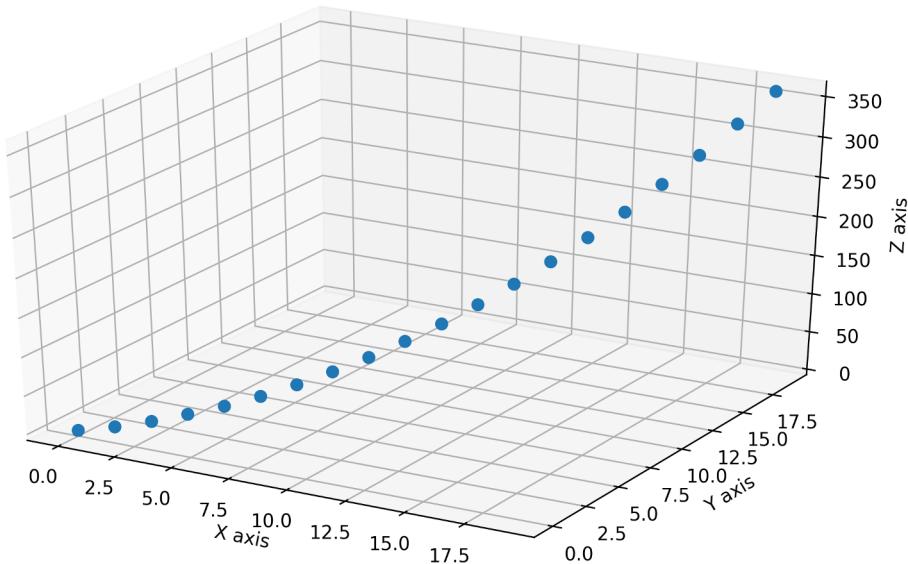
```
[in]: from mpl_toolkits.mplot3d import Axes3D
[in]: x = range(20)
       y = x
       z = [num**2 for num in x]

[in]: fig = plt.figure(figsize = (10,6))

       ax = fig.add_subplot(1,1,1, projection='3d')
       ax.plot(x, y, z, 'o')

       ax.set_xlabel('X axis')
       ax.set_ylabel('Y axis')
       ax.set_zlabel('Z axis')

[out]:
```



## 3.6 Surface Plots

The above 3D plots are simply scatter plots in a three-dimensional space. It is often useful to connect these points to describe surfaces in 3D space which can be used for energy surfaces or atomic orbital shapes among other applications. We again will import `Axes3D` from `mpl_toolkits.mplot3d` as we did in section 3.5, but we also need to generate a mesh grid to create a surface plot. Mesh grids are simply the  $x$  and  $y$  axes values extended into a 2D array. A simple example is shown below where the  $x$  and  $y$  axes are integers from 0 → 8. In the left grid, the values represent where each point is with respect to the  $x$ -axis, and the right grid is likewise where each point is located with respect to the  $y$ -axis.

| X Values Grid |   |   |   |   |   |   |   |   | Y Values Grid |   |   |   |   |   |   |   |   |
|---------------|---|---|---|---|---|---|---|---|---------------|---|---|---|---|---|---|---|---|
| 0             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 0             | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 1             | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 2             | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 0             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 3             | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 0             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 4             | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 0             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 5             | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 0             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 6             | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 0             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 7             | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| 0             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 8             | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |

We will use NumPy to generate these grids as NumPy arrays. If you have not yet seen NumPy, you can still follow along in this example without understanding how arrays operate, or you can read chapter 4 and come back to this topic later. For those who are familiar with NumPy, being that the two grids/arrays are of the same dimension, all math is done on a position-by-position basis to generate a third array of the same dimensions as the first two. For example, if we were to take the sum of the squares of the two grids above, we would get the following grid.

$$z = x^2 + y^2$$

| <b>Z Values Grid</b> |    |    |    |    |    |     |     |     |  |
|----------------------|----|----|----|----|----|-----|-----|-----|--|
| 0                    | 1  | 4  | 9  | 16 | 25 | 36  | 49  | 64  |  |
| 1                    | 2  | 5  | 10 | 17 | 26 | 37  | 50  | 65  |  |
| 4                    | 5  | 8  | 13 | 20 | 29 | 40  | 53  | 68  |  |
| 9                    | 10 | 13 | 18 | 25 | 34 | 45  | 58  | 73  |  |
| 16                   | 17 | 20 | 25 | 32 | 41 | 52  | 65  | 80  |  |
| 25                   | 26 | 29 | 34 | 41 | 50 | 61  | 74  | 89  |  |
| 36                   | 37 | 40 | 45 | 52 | 61 | 72  | 85  | 100 |  |
| 49                   | 50 | 53 | 58 | 65 | 74 | 85  | 98  | 113 |  |
| 64                   | 65 | 68 | 73 | 80 | 89 | 100 | 113 | 128 |  |

Notice that each value on the  $z$  grid is the sum of the squared values from the equivalent positions on the  $x$  and  $y$  grids... so the bottom left value is 64 because it is the sum of  $8^2$  and  $0^2$ .

To generate mesh grids, we will use the `np.meshgrid()` function from NumPy. It requires the input of the desired values from the  $x$  and  $y$  axes as a list, range object, or NumPy array. The output of the `np.meshgrid()` function is two arrays – the  $x$  grid and  $y$  grid, respectively.

```
[in]: import numpy as np
x = np.arange(-10, 10)
y = np.arange(-10, 10)
X, Y = np.meshgrid(x, y)

[in]: Z = 1 - X**2 - Y**2
```

Now to plot the surface. We will use the `plot_surface()` function which requires the  $X$ ,  $Y$ , and  $Z$  mesh grids as arguments. As an optional argument, you can designate a color map. Color maps are a series of colors or shades of a color that represents values. The default for matplotlib is `viridis`, but you can change this to anything from a wide selection of

color maps provided by matplotlib. For more information on color maps, see the matplotlib website.<sup>30</sup>

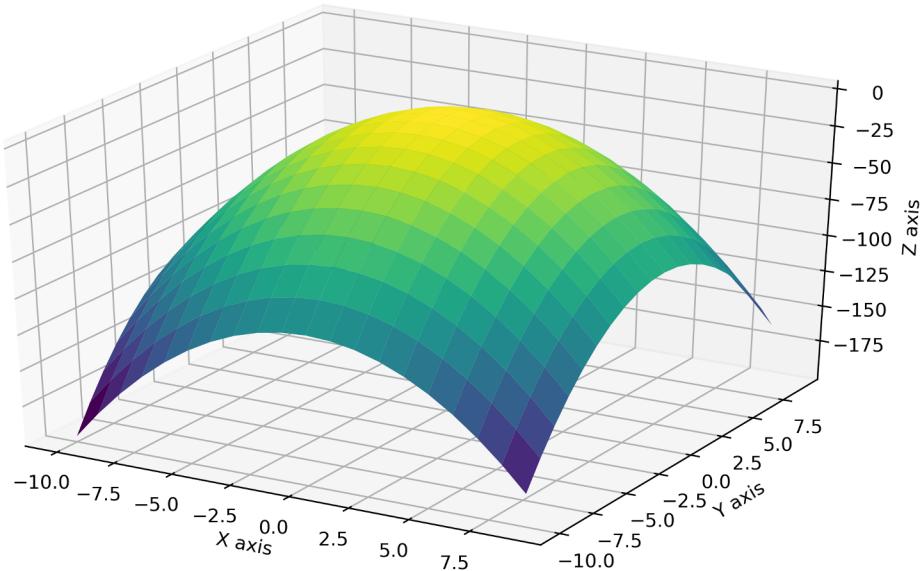
```
[in]: from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(10,6))

ax = fig.add_subplot(1,1,1, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis')

ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
```

[out] :



As a more chemical example, we can plot the standing waves for a 2D particle in a box by the following equation where  $n_x$  and  $n_y$  are the principle quantum numbers along each axis and  $L$  is the length of the box.<sup>31</sup>

$$\psi(x, y) = (2/L) \sin(n_x \pi x/L) \sin(n_y \pi y/L)$$

30 Matplotlib Colormap Reference. [https://matplotlib.org/examples/color/colormaps\\_reference.html](https://matplotlib.org/examples/color/colormaps_reference.html)

31 Lang, L. P.; Towns, M. H. Visualization of Wave Functions Using Mathematica *Chem. Educ.*, **1998**, 75 (4), 506-509.

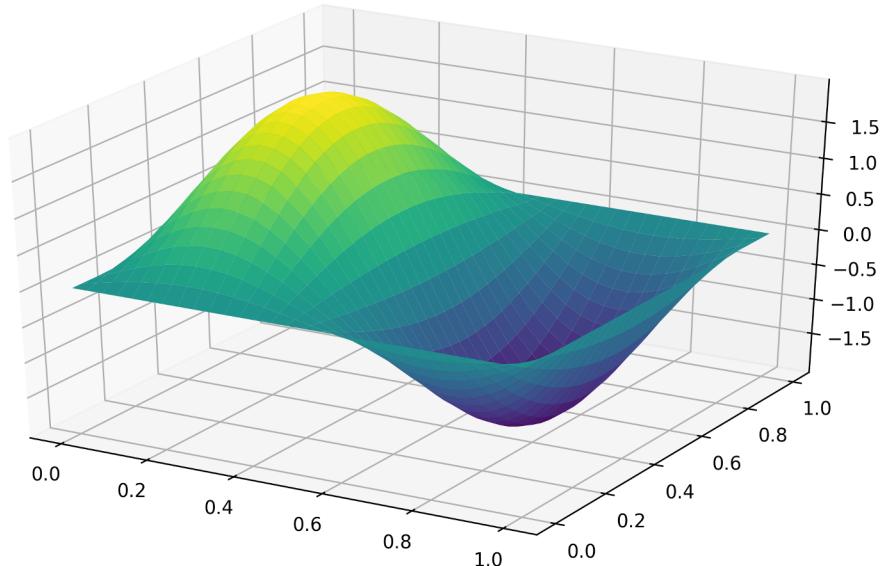
We will select  $L = 1$ ,  $n_x = 2$ , and  $n_y = 1$ . Again, a meshgrid is generated and a height value is calculated from the  $x$  and  $y$  values.

```
[in]: L = 1
nx = 2
ny = 1
x = np.linspace(0, L, 20)
y = np.linspace(0, L, 20)
X, Y = np.meshgrid(x,y)

[in]: def wave(x, y):
    psi = (2/L) * np.sin(nx*np.pi*X/L) *
          np.sin(ny*np.pi*Y/L)
    return psi

[in]: fig = plt.figure(figsize=(10,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, wave(X, Y), cmap='viridis')

[out]:
```



You are encouraged to increase the values for  $n_x$  and  $n_y$  and see how the surface plot changes.

## Further Reading

The matplotlib website is an excellent place to learn more about plotting in Python. Similar to some other Python library websites, there is a gallery page that showcases many of the capabilities of the matplotlib library. It is often worth browsing to get ideas and a sense of what the library can do.

1. Matplotlib Website. <https://matplotlib.org/>.

## Exercises

Complete the following exercises using the matplotlib library.

### Scatter & Line Plots

1. Visualize the relationship between pressure and volume for 1.00 mol of He(g) at 298 K in an expandable vessel as it increases from 1 L → 20 L.  $R = 0.08206 \text{ L}\cdot\text{atm}/\text{mol}\cdot\text{K}$ .

$$PV = nRT$$

2. Plot the electronegativity versus atomic number for the first five halogens, and make the size or color of the markers based on the atomic radii of the element.

You will need to look up the values which should be available in most general chemistry textbooks. If you do not have one available, you can also find these values in the free, open chemistry textbook available on OpenStax (<https://openstax.org/>) among other online resources.

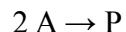
3. The following functions are an example of the sandwich theorem which aids in determining limits of function  $g(x)$  by knowing its range is between  $f(x)$  and  $h(x)$  in the relevant domain. Plot all three functions on the same axes to show that  $f(x) \leq g(x) \leq h(x)$  for  $x$  of -50 → 50. Be sure to include a legend.

$$f(x) = x^2$$

$$g(x) = x^2 \sin(x)$$

$$h(x) = -x^2$$

4. Plot the concentration of A with respect to time for the following elementary step if  $k = 0.12 \text{ M}^{-1}\text{s}^{-1}$  using the appropriate integrated rate law.



5. Import the `gc_trace.csv` file containing a gas chromatography (GC) trace and plot the intensity (y-axis) versus  $m/z$  (x-axis) using a line plot. Be sure to label the axes.

## Additional Plotting Types

6. Earth's atmosphere is composed of 78% N<sub>2</sub>, 21% O<sub>2</sub>, and 1% other gases. Represent this data with a pie chart, and make the last 1% slice stick out of the pie like in Table 3.6.
7. Create a histogram plot to examine the distribution of values generated below.<sup>32</sup>

```
import random
rdn = [random.random() for value in range(1000)]
```

8. The <sup>1</sup>H NMR spectrum of caffeine in CDCl<sub>3</sub> is composed of four singlets with the following chemical shifts and relative intensities. Visualize this data using a stem plot. Hint: the dots on the top of the lines can be removed using markerfmt=' ' (i.e., feed it a blank space).

```
ppm = [7.52, 4.00, 3.60, 3.44]
intensity = [1.52, 3.90, 5.74, 5.78]
```

9. Import the mass spectra file *ms\_bromobenzene.csv* and visualize it using a stem plot where *m/z* is on the *x*-axis and intensity is on the *y*-axis. Hint: the dots on the top of the lines can be removed using markerfmt=' ' (i.e., feed it a blank space).
10. The following table presents the calculated free energies for each step in the binding and splitting of H<sub>2</sub>(g) by a nickel phosphine catalyst.<sup>33</sup> Visualize the energies over the course of the reaction using a plotting type other than a line or scatter plot.

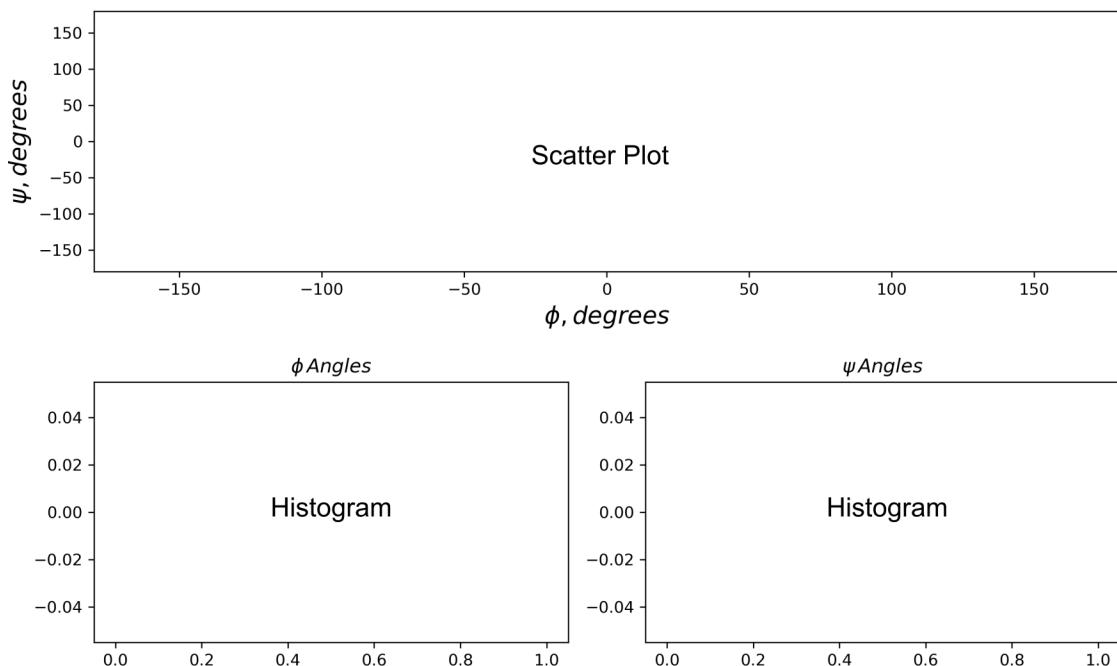
| Step | Relative Free Energy<br>(kcal/mol) |
|------|------------------------------------|
| 1    | 0.0                                |
| 2    | 11.6                               |
| 3    | 9.8                                |
| 4    | 13.4                               |
| 5    | 5.8                                |
| 6    | 8.3                                |
| 7    | 2.7                                |

<sup>32</sup> See section 2.1.2 on list comprehension for an explanation of the following code.

<sup>33</sup> Data from Raugei , S; Helm, M. L.; Hammes-Schiffer, S.; Appel, A. M.; O'Hagan, M.; Wiedner, E. S.; Bullock, R. M. *Inorg. Chem.* **2016**, 55, 445–460.

## Multifigure Plots

11. Generate two side-by-side plots that show the atomic radii and first ionization energies versus atomic number for the first ten elements on the periodic table. This data should be available on the internet or any general chemistry textbook in the periodic trends chapter. Include titles on both plots along with appropriate axis labels.
12. Import the dihedral angle data for the NiFe-hydrogenase enzyme<sup>34</sup> from the file titled *hydrogenase\_5a4m\_phipsi.csv*. Generate a single figure containing a scatter plot of the psi versus phi angles (top) and two histograms showing the distributions of phi and psi dihedral angles (bottom) as shown below.



## 3D Plotting

13. Generate a standing wave surface plot (similar to the one at the end of section 3.6) using the following equation and parameters:  $L = 1$ ,  $n_x = 2$ ,  $n_y = 2$ .

$$\psi(x, y) = (2/L)\sin(n_x \pi x/L)\sin(n_y \pi y/L)$$

<sup>34</sup> The dihedral angles were extracted from the crystal structure reported in Evans, R.M.; Brooke, E.J.; Wehlin, S.A.M.; Nomerotskaia, E.; Sergent, F.; Carr, S.B.; Philips, S.E.V.; Armstrong, F.A. Mechanism of Hydrogen Activation by [NiFe] Hydrogenases. *Nat.Chem.Biol.* **2016**, 12, 46-50.

# *Chapter 4*

## **NumPy**

NumPy is a popular library in the Python ecosystem and a critical component of the SciPy stack. So much so that NumPy is included in Apple's default installation of Python and in other Python-powered applications such as Blender.<sup>35</sup> While it may be tempting to work with NumPy's objects as lists or to circumnavigate the NumPy library altogether, the time it takes to learn NumPy's powerful features is well worth it! It will often allow you to solve problems with less effort and time and with shorter and faster-executing code. This is due to:

- NumPy automatically propagating operations to all values in an arrays instead of requiring `for` loops
- A massive collection of functions for working with numerical data
- Many of NumPy's functions are Python-wrapped C code making them run more quickly

The NumPy package can be imported by `import numpy`, but the scientific Python community has developed an unofficial, but strong, convention of importing NumPy using the `np` alias. It is a matter of personal preference whether to use the alias or not, but it is strongly encouraged for consistency with the rest of the community. Instead of `numpy.function()`, the function is then called by the shorter `np.function()`. All of the NumPy code in this and subsequent chapters assumes the following import.

```
[in]: import numpy as np
```

---

<sup>35</sup> Blender is a free, open source 3D graphics and animation software application available at <https://www.blender.org/>.

## 4.1 NumPy Arrays

One of the main contributions of NumPy is the *ndarray* (i.e., "n-dimensional array") or just *array* for short. This is an object similar to a list or nested list of lists except that mathematical operations and NumPy functions automatically propagate to each element instead of requiring a `for` loop to iterate over it. Because of their power and convenience, arrays are the default object type for any operation performed with NumPy and many scientific libraries that are built on NumPy (e.g., SciPy, pandas,<sup>36</sup> scikit-learn, etc...).

### 4.1.1 Basic Arrays

The NumPy array looks like a Python list wrapped in `array()`. It is an iterable object, so you *could* iterate over it using a `for` loop if you really want to. However, because NumPy arrays automatically propagate operations through the array, `for` loops are typically unnecessary. For example, let us say you want to multiply a list of numbers by 2. Doing this with a list would likely look like the following.

```
[in]: nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
       for value in nums:
           print(2 * value)
[out]:
0
2
4
6
8
10
12
14
16
18
```

In contrast, performing this same operation using a NumPy array only requires multiplying the array by 2.

```
[in]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
       print(2 * arr)
[out]: array([ 0  2  4  6  8 10 12 14 16 18])
```

---

<sup>36</sup> While pandas (chapter 5) uses its own Series and DataFrames objects, they are built on NumPy arrays.

## 4.1.2 Type Conversion to Arrays

There are three common ways to generate a NumPy array that we will cover in the beginning of this chapter. The first is simply to convert a list or tuple to an array using the `np.array()` function.

```
[in]: a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]    # list
       arr = np.array(a)
       print(arr)
[out]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

The fact that the object is an ndarray is denoted by the `array()`.

## 4.1.3 Array from Sequence

We can also create an array using NumPy sequence-generating functions. There are two common functions in NumPy for this task: `np.arange()` and `np.linspace()`. The `np.arange()` function behaves similarly to the native Python `range()` function with the key difference that it outputs an array. Another minor difference is that while `range()` generates a range object, `np.arange()` generates a sequences of values immediately. The arguments for `np.arange()` are similar to that of Python's `range()` function where `start` is inclusive and `stop` is exclusive, but unlike `range()`, the step size for `np.arange()` does not need to be an integer value.

```
np.arange(start, stop, step)
```

The `np.linspace()` function is related to `np.arange()` except that instead of defining the sequence based on step size, it generates a sequence based on *how many* evenly distributed points to generate in the given span of numbers. Additionally, `np.arange()` excludes the stop values while `np.linspace()` includes it. The difference between these two functions is somewhat subtle, and the use of one over the other often comes down to user preference or convenience.

```
np.linspace(start, stop, number of points)
[in]: arr = np.arange(0, 10, 0.5)
       arr
[out]:
```

```
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  
 4.5,  5. ,  5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  
 9.5])  
  
[in]: arr = np.linspace(0,10, 20)  
      arr  
  
[out]:  
  
array([ 0.          ,  0.52631579,  1.05263158,  1.57894737,  
       2.10526316,  2.63157895,  3.15789474,  3.68421053,  
       4.21052632,  4.73684211,  5.26315789,  5.78947368,  
       6.31578947,  6.84210526,  7.36842105,  7.89473684,  
       8.42105263,  8.94736842,  9.47368421,  10.        ])
```

Two other useful functions for generating arrays are `np.zeros()` and `np.ones()` which generate arrays populated with exclusively zeros and ones, respectively. The functions accept the `shape` argument as tuple of the array dimensions in the form `(rows, columns)`.

```
[in]: np.zeros((2,4))  
  
[out]:  
  
array([[ 0.,  0.,  0.,  0.],  
       [ 0.,  0.,  0.,  0.]])  
  
[in]: np.ones(10)  
  
[out]:  
  
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

You should commit to remembering `np.arange()` and `np.linspace()`, as these are used often. The `np.zeros()` and `np.ones()` functions are not as common but are useful in particular applications. For example, to generate an array of threes, an array of zeros can be generated and then incremented by 3.

```
[in]: arr = np.zeros((2,4))  
      arr += 3  
      print(arr)  
  
[out]: array([[ 3.,  3.,  3.,  3.],  
              [ 3.,  3.,  3.,  3.]])
```

#### 4.1.4 Arrays from Functions

A third approach is to generate an array from a function using `np.fromfunction()` which generates an array of values using the array indices as inputs. This function requires a function as an argument.

```
np.fromfunction(function, shape)
```

Let us make an array of the dimensions (3,3) where each element is the product of the row and column indices.

```
[in]: def prod(x, y):
        return x * y

[in]: np.fromfunction(prod, (3,3))

[out]: array([[ 0.,  0.,  0.],
              [ 0.,  1.,  2.],
              [ 0.,  2.,  4.]])
```

## 4.2 Reshaping & Merging Arrays

Modifying the dimensions of one or more arrays is a common task in NumPy. This may involve changing the number of columns and rows or merging multiple arrays into a larger array. The `size` and `shape` of an array are the number of elements and dimensions, respectively. These can be determined using the `size` and `shape` NumPy methods.

```
[in]: counting = np.array([[1, 2, 3], [4, 5, 6]])

[in]: counting.size

[out]: 6

[in]: counting.shape

[out]: (2, 3)
```

### 4.2.1 Reshaping Arrays

The dimensions of arrays can be modified using the `np.reshape()` method. This method maintains the number of elements and order of elements in the array but repacks them into a different number of row and columns. Because the number of elements is maintained, the new size array needs to contain the same number of elements as the original.

```
np.reshape(array, dimensions)
```

In this function, `array` is the NumPy array being reshaped and `dimensions` is a tuple containing the desired number of rows and columns in that order. The original array must fit exactly into the new dimensions or else NumPy will refuse to change it. This method does not change the original array in place but rather returns a modified copy. This is a good time to note that because this and other NumPy functions are methods for NumPy arrays, they can also be called by listing the array up front like list and string methods presented in chapter 1. For example, the `reshape()` function can be called with `array.reshape(dimensions)`.

```
[in]: array_1D = np.linspace(0, 9.5, 20)
      array_1D

[out]:
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,
       4.5,  5. ,  5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,
       9.5])
```

The following code reshapes the array into a  $4 \times 5$  array.

```
[in]: array_2D = np.reshape(array_1D, (4, 5))
      array_2D

[out]:
array([[ 0. ,  0.5,  1. ,  1.5,  2. ],
       [ 2.5,  3. ,  3.5,  4. ,  4.5],
       [ 5. ,  5.5,  6. ,  6.5,  7. ],
       [ 7.5,  8. ,  8.5,  9. ,  9.5]])
```

If you need to reshape an array with only one new dimension known, place a `-1` in the other. This signals to NumPy that it should choose the second dimension to make the data fit.

## 4.2.2 Flatten Arrays

Flattening an array takes a higher-dimensional array and squishes it into a one-dimensional array. To flatten out an array, the `np.flatten()` method is often the most convenient way.

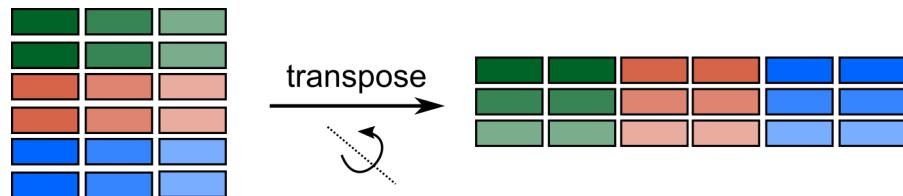
```
[in]: array_2D.flatten()

[out]:
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,
       4.5,  5. ,  5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,
       9.5])
```

The format of the output makes it look like it is still a 2D array, but notice that there is a comma instead of square a bracket at the end of the first row. The dimensions of this array are  $1 \times 20$ .

### 4.2.3 Transpose Arrays

Transposing an array rotates the array around the diagonal (Figure 4.1).



**Figure 4.1** The `np.transpose()` or `array.T` method transposes the NumPy array effectively flipping the rows and columns.

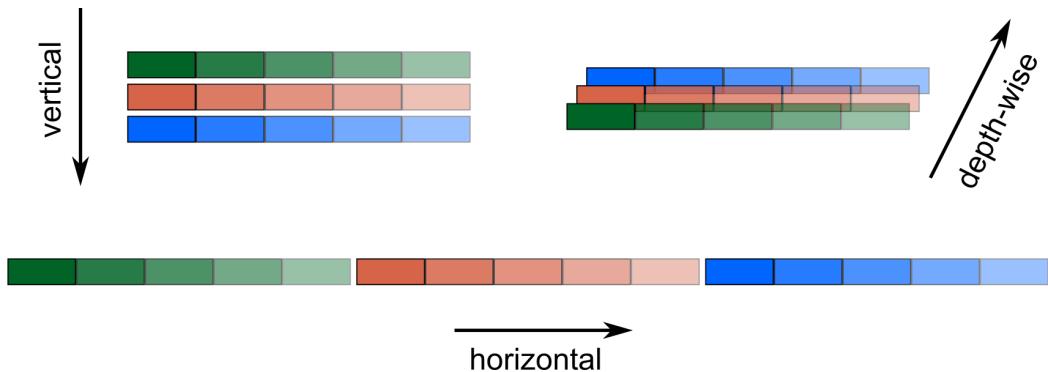
The `np.transpose()` method flips the rows and columns. NumPy also provides an alias/shortcut of `array.T` to accomplish the same outcome. The latter is far more common, so it is the method used here.

```
[in]: array_2D
[out]:
array([[ 0. ,  0.5,  1. ,  1.5,  2. ],
       [ 2.5,  3. ,  3.5,  4. ,  4.5],
       [ 5. ,  5.5,  6. ,  6.5,  7. ],
       [ 7.5,  8. ,  8.5,  9. ,  9.5]])
```

```
[in]: array_2D.T
[out]:
array([[ 0. ,  2.5,  5. ,  7.5],
       [ 0.5,  3. ,  5.5,  8. ],
       [ 1. ,  3.5,  6. ,  8.5],
       [ 1.5,  4. ,  6.5,  9. ],
       [ 2. ,  4.5,  7. ,  9.5]])
```

#### 4.2.4 Merge Arrays

Merging arrays can be done in multiple ways. NumPy provides convenient methods for merging arrays using `np.vstack`, `np.hstack`, and `np.dstack` which merge arrays along the vertically, horizontally, and depth-wise axes, respectively (Figure 4.2).



**Figure 4.2** NumPy arrays can be stacked vertically (top left), depth-wise (top right), or horizontally (bottom) using the `np.vstack()`, `np.dstack()`, and `np.hstack()` functions, respectively.

```
[in]: a = np.arange(0, 5)
[in]: b = np.arange(5, 10)
[in]: np.vstack((a, b))
[out]: array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]])
[in]: np.hstack((a, b))
[out]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
[in]: np.dstack((a, b))
[out]: array([[[0, 5],
               [1, 6],
               [2, 7],
               [3, 8],
               [4, 9]]])
```

## 4.3 Indexing Arrays

Similar to lists, it is often useful to be able to index and slice ndarrays. Because arrays are often higher dimensional, there are some differences in indexing that provide extra convenience.

### 4.3.1 One-Dimensional Arrays

Indexing one-dimensional arrays is done in an identical fashion to lists. Simply include the index value(s) or range in square brackets behind the array name.

```
[in]: array_1D  
[out]: array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  
4. ,  4.5,  5. ,  5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  
9. ,  9.5])  
  
[in]: array_1D[5]  
[out]: 2.5
```

### 4.3.2 Two-Dimensional Arrays

Two-dimensional arrays can be also indexed in a similar fashion to nested lists, but because arrays are often multidimensional, there is also a shortcut below to make work with arrays more convenient. To access the entire second row of an array, provide the row index in square brackets behind the array name just like indexing in lists.

```
[in]: array_2D  
[out]:  
  
array([[ 0. ,  0.5,  1. ,  1.5,  2. ],  
      [ 2.5,  3. ,  3.5,  4. ,  4.5],  
      [ 5. ,  5.5,  6. ,  6.5,  7. ],  
      [ 7.5,  8. ,  8.5,  9. ,  9.5]])  
  
[in]: array_2D[1]  
[out]: array([ 2.5,  3. ,  3.5,  4. ,  4.5])
```

To access the first element in the second row, it is perfectly valid to use two adjacent square brackets just as one would use in a nested list of lists. However, to make work more convenient, these square brackets are often combined with the row and column indices separated by commas.

```

array_name[rows, columns]

[in]: array_2D[1][0]
[out]: 2.5

[in]: array_2D[1, 0]
[out]: 2.5

```

Ranges of values can also be accessed in arrays by using slicing. The following array input generates a slice of the second row of the array.

```

[in]: array_2D[1, 1:]
[out]: array([3., 3.5, 4., 4.5])

```

As seen above, if you want to access an entire row, it is not necessary to indicate the columns. It is implicitly understood that all columns are requested. However, if you want to access the first column, something needs to be placed before the column. The easiest solution is the use a colon to explicitly indicate all rows.

```

[in]: array_2D[0]          # implicitly understood all columns
[out]: array([ 0., 0.5, 1., 1.5, 2. ])
[in]: array_2D[0, :]       # explicit indicating all columns
[out]: array([ 0., 0.5, 1., 1.5, 2. ])
[in]: array_2D[:, 0]        # all rows
[out]:
array([ 0., 2.5, 5., 7.5])

```

## 4.4 Vectorization & Broadcasting

One of the major advantages of NumPy arrays over lists is that operations automatically *vectorize* across the arrays. That is, mathematical operations propagate through the array(s) instead of requiring for loops. This both speeds up the calculations and makes the code easier to read and write.

### 4.4.1 NumPy Functions

Let us take the square root of squares using NumPy's `np.sqrt()` function. The square root is taken of each elements automatically.

```
[in]: squares = np.array([1, 4, 9, 16, 25])
```

```
[in]: np.sqrt(squares)

[out]: array([ 1.,  2.,  3.,  4.,  5.])
```

Performing this operation requires NumPy's `sqrt()` function. If this is attempted with the math module's `sqrt()` function, an error is returned because this function cannot take the square root of a multi-element object.

```
[in]: import math
      math.sqrt(squares)

[out]: TypeError: only length-1 arrays can be converted to
          Python scalars
```

#### 4.4.2 Scalars & Arrays

When performing mathematical operations between a scalar and an array, the same operation is performed across each elements of the array returning an array of the same dimension. Below, an array is multiplied by the scalar 3 which results in every element in the array being multiplied by this value.

$$3 \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 15 & 18 \\ 21 & 24 \end{bmatrix}$$

```
[in]: 3 * np.array([[5, 6], [7, 8]])

[out]: array([[15, 18], [21, 24]])
```

The same outcome arises when performing a similar operation between a  $1 \times 1$  array and a larger array.

$$[2] + [10 \ 20] = [12 \ 22]$$

```
[in]: np.array([2]) + np.array([10, 20])

[out]: array([12, 22])
```

#### 4.4.3 Arrays of the Same Dimensions

If a mathematical operation is performed between two arrays of the same dimensions, then the mathematical operation is performed between corresponding elements in the two arrays. For example, if a pair of  $2 \times 2$  arrays are added to one another, then the corresponding elements are added to one another. This means that the top left elements are added together and so on.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

```
[in]: a = np.array([[1,2], [3,4]])

b = np.array([[5,6], [7,8]])

a + b

[out]:
array([[ 6,  8],
       [10, 12]])
```

#### 4.4.4 Arrays of Different Dimensions

*Broadcasting* is another form of vectorization that is a series of rules for dealing with mathematical operations between two arrays of *different dimensions*. In broadcasting, the dimensions of the two arrays must be either identical or one-dimensional, otherwise nothing happens except an error message. To deal with the different dimensions, NumPy pads or clones the array with fewer dimensions out so that it had the same dimensions as the other array.<sup>37</sup> For example, below is the addition between a  $2\times 2$  and a  $1\times 2$  array.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + [2 \ 2] = ?$$

To make the two arrays the same size, the smaller array is cloned along the smaller dimension until the two arrays are the same size as shown below. We are then left with simple corresponding element by corresponding element mathematical operations described in section 4.4.3.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$$

```
[in]: a = np.array([[1,2], [3,4]])

b = np.array([2,2])

a + b
```

---

<sup>37</sup> It should be noted that NumPy does not really clone or pad out the array in the background. Its behavior *acts* as if it does, and it is a convenient way of thinking about the behavior and results.

```
[out]:
array([[3, 4],
       [5, 6]])
```

What happens if a mathematical operation is performed between an array of higher dimensions with a scalar or a  $1 \times 1$  array as shown below? You already probably know the answer from section 4.4.2, but here is how to rationalize the behavior. In this case, no dimensions are the same, but being that one of the arrays has dimensions of one where the two arrays differ, the arrays still broadcast.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times [2] = ?$$

Again, the smaller array is padded out or cloned until the two arrays are the same size.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

```
[in]: a = np.array([[1,2], [3,4]])
       b = np.array([2])
       a * b

[out]:
array([[2, 4],
       [6, 8]])
```

Finally, if we attempt to perform a mathematical operation between two arrays with different dimensions and none of the arrays have a dimension of one where the two arrays are different, an error is raised, and no operation is performed.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} = ?$$

```
[in]: a = np.array([[1,2], [3,4]])
       b = np.array([[1,1,1], [2,2,2], [3,3,3]])
       a + b

[out]: ValueError: operands could not be broadcast together
       with shapes (2,2) (3,3)
```

#### 4.4.5 Vectorizing Python Functions

Standard Python functions are often designed to perform a calculation a single time and output Python objects and not NumPy arrays. As an example, the following function calculates the rate of a first-order reaction given the rate constant ( $k$ ) and concentration of reactant ( $\text{conc}$ ).

```
[in]: def rate(k, conc):  
        return r*conc  
  
[in]: rate(1.2, 0.80)  
  
[out]: 0.96
```

What happens if we attempt the above calculation using a list of concentration values?

```
[in]: concs = [0.1, 0.5, 1.0, 1.5, 2.0]  
  
[in]: rate(1.2, concs)  
  
[out]: TypeError: can't multiply sequence by non-int of type  
       'float'
```

We get an error because Python cannot multiply a list by a value the way NumPy can. However, the above function can be converted to a NumPy function using `np.vectorize()` which will allow the function to perform the calculation on a series of values and returns a NumPy array.

```
[in]: vrate = np.vectorize(rate)  
  
[in]: vrate(1.2, concs)  
  
[out]: array([0.12, 0.6 , 1.2 , 1.8 , 2.4 ])
```

### 4.5 Array Methods

Technically, NumPy array methods have already been employed in this chapter. The functions above are NumPy methods specifically for working with NumPy arrays. If an array is fed to many non-NumPy functions, an error will result because they cannot handle multi-element objects or arrays specifically. Interestingly, if a float or integer is fed into a NumPy method, it will still work. As an example, the integer 4 can be inserted into the `np.sqrt()` function as well as an array of values.

```
[in]: np.sqrt(4)  
  
[out]: 2
```

```
[in]: np.sqrt(array([1, 4, 9]))
[out]: array([1, 2, 3])
```

NumPy contains an extensive listing of methods for working with arrays... so much so that it would be impractical to list them all here. However, below are tables of some common and useful methods. It is worth browsing and being aware of them; most are worth committing to memory. If you ever find yourself needing to manipulate an array in some complex fashion, it is worth doing a quick internet search and including “numpy” in your search. You will likely either find additional NumPy methods that will help or advice on how others solved a similar problem.

**Table 4.1** Common Methods for Generating Arrays

| Method            | Description                                                    |
|-------------------|----------------------------------------------------------------|
| np.array()        | Generates an array from another object                         |
| np.arange()       | Creates an array from [start, stop) with a given step size     |
| np.linspace()     | Creates an array from [start, stop] with given number of steps |
| np.empty()        | Creates an “empty” array (actually filled with garbage)        |
| np.zeros()        | Generates an array of a given dimensions filled with zeros     |
| np.ones()         | Generates an array of a given dimensions filled with ones      |
| np.fromfunction() | Generates an array using a Python function                     |
| np.genfromtxt()   | Loads text file data into an array                             |

**Table 4.2** Array Attribute Methods

| Method          | Description                                              |
|-----------------|----------------------------------------------------------|
| np.shape(array) | Returns the dimensions of an array                       |
| np.ndim(array)  | Returns the number of dimensions (e.g., a 2D array is 2) |
| np.size(array)  | Returns the number of elements in an array               |

**Table 4.3** Array Modification Methods

| Method                      | Description                                                             |
|-----------------------------|-------------------------------------------------------------------------|
| <code>np.flatten()</code>   | Flattens an array in place                                              |
| <code>np.ravel()</code>     | Returns a flattened <i>view</i> of the array without changing the array |
| <code>np.reshape()</code>   | Reshapes an array in place                                              |
| <code>np.resize()</code>    | Returns a resized view of an array without modifying the original       |
| <code>np.transpose()</code> | Returns a <i>view</i> of transposed array                               |
| <code>np.vstack()</code>    | Vertically stacks an arrays into a new array                            |
| <code>np.hstack()</code>    | Horizontally stacks an arrays into a new array                          |
| <code>np.dstack()</code>    | Depth-wise stacks an arrays into a new array                            |
| <code>np.vsplit()</code>    | Splits an array vertically                                              |
| <code>np.hsplit()</code>    | Splits an array horizontally                                            |
| <code>np.dsplits()</code>   | Splits an array depth-wise                                              |
| <code>np.meshgrid()</code>  | Creates a meshgrid (see chapter 3 for an example)                       |
| <code>np.sort()</code>      | Sorts elements in array; defaults along last axis                       |
| <code>np.argsort()</code>   | Returns index values of sorted array                                    |
| <code>np.fill(x)</code>     | Sets all values in an array to <code>x</code>                           |

**Table 4.4** Array Measurement Methods

| Method                      | Description                                            |
|-----------------------------|--------------------------------------------------------|
| <code>np.min()</code>       | Returns the minimum value in the array                 |
| <code>np.max()</code>       | Returns the maximum value in the array                 |
| <code>np.argmin()</code>    | Returns argument (i.e., index) of min                  |
| <code>np.argmax()</code>    | Returns argument (i.e., index) of max                  |
| <code>np.argrelmax()</code> | Returns argument (i.e., index) of the <i>local</i> max |
| <code>np.fmin()</code>      | Returns the min between two arrays of the same size    |
| <code>np.fmax()</code>      | Returns the max between two arrays of the same size    |

|                              |                                                                                                       |
|------------------------------|-------------------------------------------------------------------------------------------------------|
| <code>np.percentile()</code> | Returns the specified percentile                                                                      |
| <code>np.mean()</code>       | Returns the mean                                                                                      |
| <code>np.median()</code>     | Returns the median                                                                                    |
| <code>np.std()</code>        | Returns the standard deviation; be sure to include <code>ddof=1</code> <sup>38</sup>                  |
| <code>np.histogram()</code>  | Returns counts and bins for a histogram                                                               |
| <code>np.cumprod()</code>    | Returns the cumulative product                                                                        |
| <code>np.cumsum()</code>     | Returns the cumulative sum                                                                            |
| <code>np.sum()</code>        | Returns the sum of all elements                                                                       |
| <code>np.prod()</code>       | Returns the product of all elements                                                                   |
| <code>np.ptp()</code>        | Returns the peak-to-peak separation of max and min values                                             |
| <code>np.floor()</code>      | Returns the floor (i.e., rounds down) of all elements in an array <sup>39</sup>                       |
| <code>np.roll()</code>       | Rolls the array along the given axis; elements that fall off one end of the array appear at the other |

## 4.6 Random Number Generation

Stochastic simulations, addressed in chapter 9, are a common tool in the sciences and rely on a series of random numbers.<sup>40</sup> it is worth addressing their generation using NumPy. Depending upon the requirements of the simulation, random numbers may be a series of floats or integers, and they may be generated from various ranges of values. The numbers may also be generated as a uniform or biased distribution where some values are more probable than others. Below are random number functions<sup>41</sup> from the NumPy `random` module<sup>42</sup> useful in generating random number distributions to suit the needs of your simulations.

---

<sup>38</sup> Many software applications, such as Excel, will calculate the standard deviation with a delta degree of freedom of one. The `np.std()` function defaults to zero, so to get the same value as other software, set `ddof=1`.

<sup>39</sup> See section 11.5.2 of the NMRglue chapter for an example of this function in use.

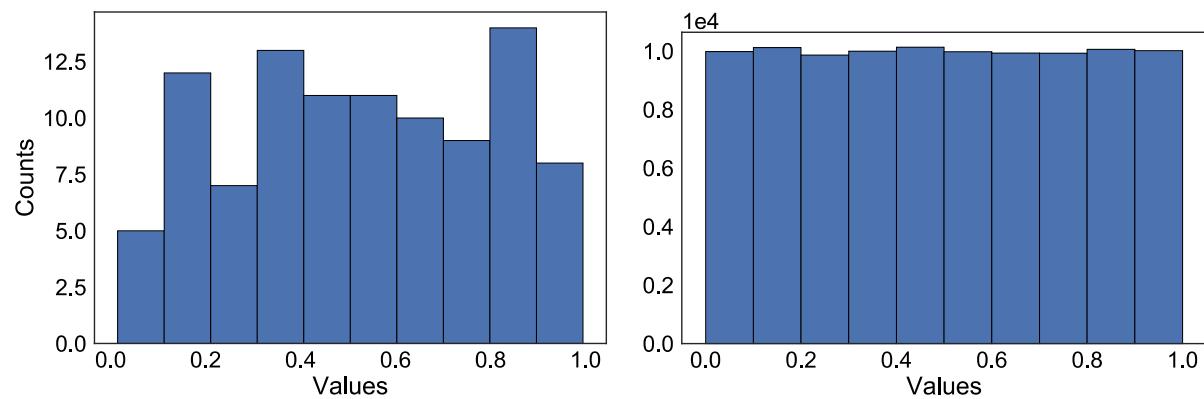
<sup>40</sup> Software generated random numbers are really *pseudorandom* numbers. However, they are close enough to random for most chemical simulations and will be referred to as “random numbers” herein.

<sup>41</sup> The term “random number generator” is used to be consistent with common terminology for these types of functions and not to imply that these functions are Python *generator functions*.

<sup>42</sup> NumPy Random Module (`numpy.random`).  
<https://docs.scipy.org/doc/numpy/reference/routines.random.html>.

### 4.6.1 Uniform Distribution

The simplest distribution is the *uniform distribution* of random numbers where every number in the range has an equal probability of occurring. The distribution may not always appear even with small sample sizes due to the random nature of the number generation, but as a larger population of samples is generated, the relative distribution will appear more even. The histograms below (Figure 4.3) are of a hundred (left) and a hundred thousand (right) randomly generated floats from the [0,1) range in an even distribution. While the plot on the right appears more even, this is mostly an effect of the different scales.



**Figure 4.3** Histograms of a hundred (left) and a hundred thousand (right) randomly generated floats from the [0,1) range in an even distribution using the `np.random.rand()` function.

NumPy has multiple functions available for generating evenly-distributed random numbers including the following two functions where  $n$  is the number of random values to be generated. The `np.random.rand()` function generates random floats from the range  $[0,1)$ <sup>43</sup> and can generate one or more values. The `np.random.randint()` function generates random integers in the range  $[\text{low}, \text{high})$ <sup>44</sup> and can generate multiple values using the `size` argument.

```
np.random.rand(n)  
np.random.randint(low, high=None, size=n)
```

<sup>43</sup> Randomly generated numbers from NumPy are generated from a range that includes the lower limit and excludes the upper limit. For example, `np.random.rand()` generates values from zero to one, including zero but excluding one.

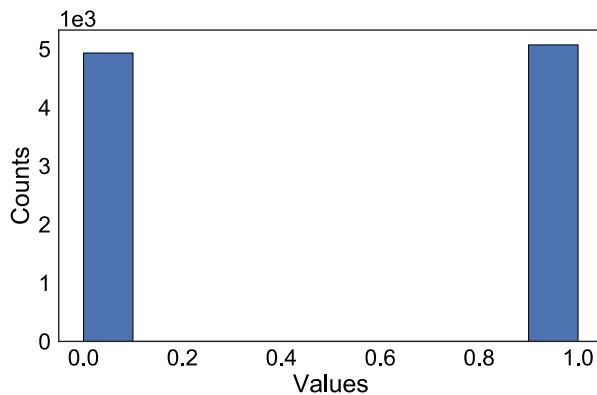
<sup>44</sup> If no `high` value is given, the value provided is assumed to be the `high` value and integers are generated from the range  $[0, \text{high})$ . See <https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randint.html> for more information.

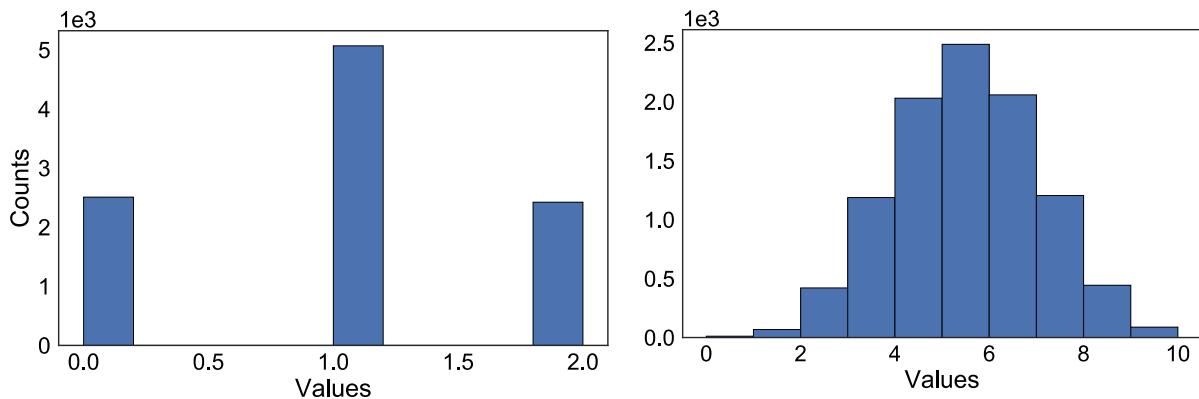
## 4.6.2 Binomial Distribution

A *binomial distribution* results when values are generated from two possible outcomes. This is useful for applications such as deciding if a simulated molecule reacts or not and whether a polymer chain terminates or propagates. The two outcomes are represented by a 0 or 1 with the probability,  $p$ , of a 1 being generated. Binomial distributions are generated by the NumPy random module using the `np.random.binomial()` function which has a default probability of 0.5.

```
np.random.binomial(t, p=0.5, size=n)
```

The  $t$  argument is the number of trials while the `size` argument is the number of generated binomial values. For example, if  $t = 2$ , two binomial values are generated and the sum is returned, which may be 0, 1, or 2. Basic probability predicts that these sums will occur in a 1:2:1 ratio, respectively. If  $t$  is increased to 10, a shape more closely representing a bell curve is obtained. A *Bernoulli distribution* is the specific instance of a binomial distribution where  $t = 1$ . The histograms below (Figure 4.4) are of a hundred randomly generated numbers in a binomial distribution with  $p = 0.5$  and where  $t = 1$  (top),  $t = 2$  (bottom left), and  $t = 10$  (bottom right).





**Figure 4.4** Histograms of a hundred randomly generated numbers in a binomial distribution with  $p = 0.5$  and  $t = 1$  (top),  $t = 2$  (bottom left), and  $t = 10$  (bottom right).

### 4.6.3 Poisson Distribution

A *Poisson distribution* is a probability distribution of how likely it is for independent events to occur in a given time interval with a known average frequency ( $\lambda$ ). Each sample in a poisson distribution is a count of how many events have occurred in the time interval, so they are always integers. NumPy can generate integers in a Poisson distribution using the `np.random.poisson()` function, which accepts two arguments.

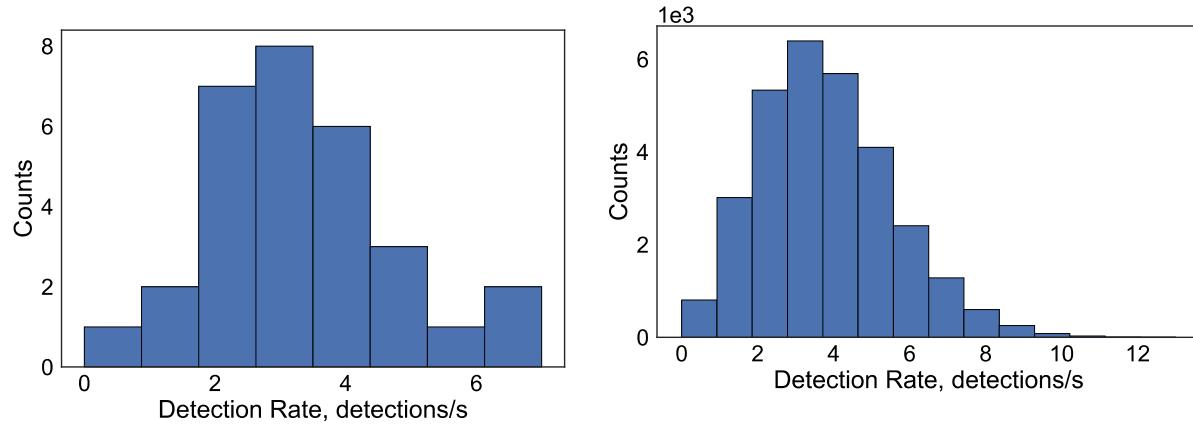
```
np.random.poisson(lam=1.0, size=n)
```

The first argument,  $\lambda$  (`lam`), is the statistical mean for the values generated, and the second argument, `size`, is the requested number of values. For example, a Geiger counter can be simulated detecting background radiation in a location that is known to have an average of 3.6 radiation counts per second with the following function call.

```
[in]: np.random.poisson(lam=3.6, size=30)
[out]: array([4, 2, 4, 3, 6, 7, 4, 4, 2, 1, 3, 1, 7, 7, 5, 3,
   2, 4, 2, 7, 4, 2, 3, 2, 1, 4, 5, 8, 4, 3])
```

The returned array of values are the total radiation detections for each second for thirty seconds, and the mean value is 3.8 counts. While not precisely the target of 3.6 counts, it is close and larger sample sizes are statistically more likely to generate results closer to the target value. A histogram of these values is shown in below (Figure 4.5, left). When this simulation was repeated with thirty thousand samples (Figure 4.5, right), a mean of 3.61 counts is obtained. In addition, the larger number of values results in a classic Poisson

distribution curve which appears something like a bell curve with more tapering on the high end.



**Figure 4.5** Histograms of thirty (left) and thirty thousand (right) randomly generated integers in a Poisson distribution with a target mean ( $\lambda$ ) of 3.6.

Alternative distributions of random number can be generated by manipulating the output of the above functions. For example, random numbers in a [-1,1] distribution, which is useful in a 2D diffusion simulation, can be generated by subtracting 0.5 from values in the range [0, 1) and multiplying by two.

```
rand_float = 2 * (np.random.rand() - 0.5)
```

#### 4.6.4 Other Functions

The `random` module in NumPy also includes a large variety of other random number and sequence generators. This includes `np.random.randn()` which generates values centered around zero in a *normal* distribution. The `np.random.choice()` function selects a random value from a provided array of values, while the `np.random.shuffle()` function randomizes the order of values for a given array. Other random distribution functions can be found on the SciPy website (see Further Reading). A summary of common `np.random` functions are in Table 4.5.

**Table 4.5** Summary of Common NumPy np.random Functions

| Function <sup>a</sup> | Description                                                                            | Example                                                                                                                    |
|-----------------------|----------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| rand()                | Generates random floats in the range [0,1) in an even distribution                     | [in]: np.random.rand(1)<br>[out]: array([0.8895507])                                                                       |
| randint()             | Generates random integers from a given range in an even distribution <sup>b</sup>      | [in]: np.random.randint(0, high=100)<br>[out]: 78                                                                          |
| randn()               | Generates random floats in a normal distribution centered around zero                  | [in]: np.random.randn(3)<br>[out]: array([-0.061095, -1.26774427, -0.67311476])                                            |
| binomial()            | Generates random integers in a binomial distribution; takes a probability argument (p) | [in]: np.random.binomial(2, p=0.5, size=3)<br>[out]: array([2, 2, 1])                                                      |
| poisson()             | Generates random floats in a Poisson distribution; takes a target mean argument (p)    | [in]: np.random.poisson(lam=2.0, size=5)<br>[out]: array([1, 3, 2, 2, 1])                                                  |
| choice()              | Selects random values taken from a 1-D array or range                                  | [in]: np.random.choice(20, size=3)<br>[out]: array([19, 4, 6])                                                             |
| shuffle()             | Randomizes the order of an array                                                       | [in]: arr = np.array([0, 1, 2, 3, 4])<br>[in]: np.random.shuffle(arr)<br>[in]: print(arr)<br>[out]: array([1, 2, 4, 0, 3]) |

<sup>a</sup> All functions can be called by np.random.function() where function is the function name listed above. <sup>b</sup> A NumPy array is generated if more than one integer is generated using the size argument.

## Further Reading

While NumPy has its own website, the documentation is hosted on the SciPy website demonstrating the close ties between the two libraries. The NumPy documentation is well written and a good resource. Because NumPy is the foundation of the SciPy ecosystem, if you find a Python book on scientific computing, odds are that it will discuss or use NumPy at some level.

1. NumPy Website. <http://www.numpy.org/>.
2. NumPy Documentation. <https://docs.scipy.org/doc/>.

## Exercises

Complete the following using NumPy arrays and methods. Avoid using `for` loops whenever possible.

### NumPy Arrays

1. Generate an array of atomic numbers for the first 26 elements.
2. The following equation defines the relationship between energy ( $E$ ) in J of a photon and its wavelength ( $\lambda$ ) in m. The  $h$  is Plank's constant ( $6.626 \times 10^{-34}$  J·s) and  $c$  is the speed of light in a vacuum ( $2.998 \times 10^8$  m/s).

$$E = \frac{hc}{\lambda}$$

- a) Generate an array containing the wavelengths of visible light ( $4.00 \times 10^{-7}$  m →  $8.00 \times 10^{-7}$  m) in  $5 \times 10^{-8}$  m increments.
- b) Generate a second array containing the energy of each wavelength of light from part a.
3. Generate an array containing the value 101.325 a hundred times.
4. The following array contains temperatures in Fahrenheit. Convert these values to °C without using a `for` loop.

```
T_F = array([0, 32, 100, 212, 451])
```

5. Generate an array ( $x$ ) containing values from 0 → 10 with at least a hundred data points. Using this array, create two other arrays ( $y_1$  and  $y_2$ ) containing the results from the following sine functions.

$$y_1 = \sin(x) \quad y_2 = 1.1\sin(x) + 0.5$$

- a) Plot both sine waves  $y_1$  and  $y_2$  with respect to  $x$  on the same plot.
  - b) Add the two arrays together (i.e.,  $y1 + y2$ ) and plot the result against  $x$ .
  - c) Explain why the signal in part *b* is smaller in the center and larger on both ends.  
*Hint:* look at your plot for part *a* to see how the two original sine waves relate to each other.
6. The numerical relationship between  $\Delta G^\circ$  and  $K$  (equilibrium constant) is shown below. Plot  $\Delta G^\circ$  versus  $K$  at standard temperature and pressure for  $K$  values of  $0.001 \rightarrow 1000$ . Use NumPy arrays and do not use any `for` loops.

$$\Delta G^\circ = -RT \ln(K)$$

7. The numerical relationship between  $k$  (rate constant) and  $E_a$  is shown below. Plot  $k$  versus  $E_a$  at standard temperature and pressure for activation energies of  $1 \rightarrow 20$  kJ/mol. Use NumPy arrays, do not use any `for` loops, and use  $A = 1$  and  $R = 8.314 \text{ J/K}\cdot\text{mol}$ .

$$k = Ae^{-E_a/RT}$$

## **Array Transformations**

8. Generate an array containing integers  $0 \rightarrow 14$  (inclusive).
- a) Reshape the array to be a  $3 \times 5$  array.
  - b) Transpose the array from part *a*, so now it should be a  $5 \times 3$  array.
  - c) Make the array from part *b* one dimensional.
9. Combining arrays – Bohr hydrogen atom
- a) Create an array containing the principle quantum numbers ( $n$ ) for the first eight orbits of a hydrogen atom (e.g.,  $1 \rightarrow 8$ ).
  - b) Generate a second array containing the energy ( $E_n$ ) of each orbit in part *a* for the Bohr model of a hydrogen atom using the equation below.

$$E_n = -2.18 \times 10^{-18} J \frac{1}{n^2}$$

- c) Combine the two arrays from parts *a* and *b* into a new  $8 \times 2$  array with the first column containing the principle quantum numbers and the second containing the energies. It should look like the following when complete.

1       $-2.18 \times 10^{-18}$

|   |             |
|---|-------------|
| 2 | -5.45e10-19 |
| 3 | -2.42e10-19 |
| 4 | -1.36e10-19 |
| 5 | -8.72e10-20 |
| 6 | -6.06e10-20 |
| 7 | -4.45e10-20 |
| 8 | -3.41e10-20 |

## Array Indexing

10. Generate a one-dimensional array with the following code and index the 5<sup>th</sup> element of the array

```
arr = np.random.randint(0, high=10, size=10)
```

11. Generate a two-dimensional array with the following code.

```
arr2 = np.random.randint(0, high=10, size=15).reshape(5,  
3)
```

- a) Index the second element of the third column.
- b) Slice the array to get the entire third row.
- c) Slice the array to access the entire first column.
- d) Slice the array to get the last two elements of the first row.

## Vectorization & Broadcasting

12. Predict the outcome of the following operation between two NumPy arrays. Test your prediction.

$$\begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} + [1] = ?$$

13. Predict the outcome of the following operation between two NumPy arrays. Test your prediction.

$$\begin{bmatrix} 1 & 8 & 9 \\ 8 & 1 & 9 \\ 1 & 8 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = ?$$

14. Predict the outcome of the following operation between two NumPy arrays. Test your prediction.

$$\begin{bmatrix} 1 & 8 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = ?$$

## Array Methods

15. For the following randomly-generated array:

```
arr = np.random.rand(20)
```

- a) Find the *index* of the largest values in the following array.
- b) Calculate the mean value of the array.
- c) Calculate the cumulative sum of the array.
- d) Sort the array from smallest to largest values

## Random Numbers

- 16. Generate a random array of fifty values from  $-1 \rightarrow 1$  (exclusive) and calculate its median value. *Hint:* Start with an array of values  $0 \rightarrow 1$  (exclusive) and manipulate it.
- 17. Generate an array of fifty integers from  $0 \rightarrow 35$  (inclusive) and then sort it.
- 18. Hydrogen nuclei can have a spin of  $+\frac{1}{2}$  and  $-\frac{1}{2}$  and occur in approximately a 1:1 ratio. Simulate the number of  $+\frac{1}{2}$  hydrogen nuclei in a molecule containing six hydrogen atoms and plot it in a histogram. *Hint:* Being that there are two possible outcomes, this is an example of a *binomial distribution*. See section 4.6.2.

# *Chapter 5*

## *Pandas*

While NumPy is the foundation of much of the SciPy ecosystem and provides very capable ndarray objects, it has a few shortcomings. The first is that NumPy arrays cannot hold different types of objects in a single array. For example, if we attempt to convert the following list containing integers, floats, and strings into an array, NumPy converts all elements into strings as a way of making the object types uniform.

```
[in]: nums = [1, 2, 3, 'four', 5, 'six', 7.0]

[in]: import numpy as np
      np.array(nums)

[out]: array(['1', '2', '3', 'four', '5', 'six', '7.0'],
            dtype='|<U21')
```

The second shortcoming is that NumPy arrays do not support labels in the data. That is, columns cannot have labels describing what they contain like you might see in a well constructed spreadsheet. This burdens the user to keep track of which column contains what information. Finally, while NumPy contains a wealth of basic tools for working with data, there are still many operations that it does not support like grouping data based on the content of a particular column or the ability to merge two data sets with automatic alignment of analogous data.

A data scientist named Wes McKinney created the pandas library which provides a wealth of additional tools for working with data, and possibly the most endearing feature, the ability to call data based on labels. Data columns and rows can contain human-readable labels that are used to access the data. Pandas still supports accessing data using indices if the

user wishes to go that route, but the user can now access data without knowing which column it is in as long as the user knows the column label.

By popular convention,<sup>45</sup> the pandas library is imported with the `pd` alias, which is used here. This chapter also assumes the following imports.

```
[in]: import pandas as pd  
       import matplotlib.pyplot as plt  
       %matplotlib inline
```

## 5.1 Basic Pandas Objects

To support the wealth of features, pandas uses its own objects to hold data called a Series and a DataFrame, which are built on NumPy arrays. Because they are built on NumPy, many of the NumPy functions (e.g., `np.mean()`) work on pandas objects. The key difference between the a Series and DataFrame is that a Series is one-dimensional while a DataFrame is two-dimensional because unlike a NumPy array, pandas objects have fixed dimensionality. There is a three-dimensional object called a Panel, but this will not be covered here as it is not often used.

### 5.1.1 Series

While the pandas *Series* is restricted to being a single dimension, it can be as long as necessary to hold the data. A Series containing the atomic masses of the first five elements on the periodic table is generated below using the `pd.Series()` function. This function is always capitalized.

```
[in]: mass = pd.Series([1.01, 4.00, 6.94, 9.01, 10.81])  
       mass  
[out]:  
0      1.01  
1      4.00  
2      6.94  
3      9.01  
4     10.81  
dtype: int64
```

---

<sup>45</sup> This import convention, like NumPy's `np`, is certainly not required and only saves a few key strokes per usage. However, you will find yourself using it enough times that it will ultimately save plenty of typing and makes your code more readable by other users of pandas. This convention is strongly encouraged.

The right column is the actual data in the Series while the values on the left are the assigned indices for each value in the Series. The index column is not part of the dimensionality of the Series; it is metadata (i.e., data about the data).

Consistent with lists, tuples, and ndarrays, values in a Series can be accessed using indexing with square brackets as demonstrated below.

```
[in]: mass[2]  
[out]: 6.94
```

Unlike other multi-element objects seen so far, data in a Series can be accessed using indices different from the default (i.e., 0, 1, 2, etc...) values. That is, custom indices can be assigned using the `index` argument shown below.

```
[in]: mass2 = pd.Series([1.01, 4.00, 6.94, 9.01, 10.81],  
index=['H', 'He', 'Li', 'Be', 'B'])  
mass2  
[out]:  
H      1.01  
He     4.00  
Li     6.94  
Be     9.01  
B     10.81  
dtype: float64
```

The custom indices can now be used to access an element in a Series. This makes a Series behave something like a dictionary (section 2.2).

```
[in]: mass2['He']  
[out]: 4.00
```

The indices can be accessed by using `mass2.index`. Series indices can also be modified after a Series has been created by using `.index` and assignment as demonstrated below.

```
[in]: mass.index =['H', 'He', 'Li', 'Be', 'B']  
mass  
[out]:  
H      1.01  
He     4.00  
Li     6.94  
Be     9.01  
B     10.81  
dtype: float64
```

Even if we create or modify a Series to have custom indices, we can still access the elements using the traditional numerical indices using the `iloc` method. This method allows the user to access elements the same way as in a NumPy array regardless of assigned index values.

```
[in]: mass2.iloc[2]  
[out]: 6.94
```

### 5.1.2 DataFrame

Most data you will find yourself working with will be best placed in a two-dimensional pandas object called a *DataFrame* which is always written in camelcase.<sup>46</sup> The DataFrame is similar to a Series except that now there are also columns with names. The columns can be accessed by column names and rows can be accessed by indices. You might think of a DataFrame as a collection of Series objects. Below, a DataFrame is constructed to hold the names, atomic numbers, masses, and ionization energies of the first five elements.

```
[in]: name = ['hydrogen', 'helium', 'lithium', 'beryllium',  
            'boron']  
      AN = [1, 2, 3, 4, 5]  
      mass = [1.01, 4.00, 6.94, 9.01, 10.81]  
      IE = [13.6, 24.6, 5.4, 9.3, 8.3]  
  
[in]: elements = pd.DataFrame([name, AN, mass, IE],  
                           columns=['H', 'He', 'Li', 'Be',  
                                     'B'],  
                           index=['name', 'AN', 'mass',  
                                  'IE'])  
  
elements  
  
[out]:
```

|      | H        | He     | Li      | Be        | B     |
|------|----------|--------|---------|-----------|-------|
| name | hydrogen | helium | lithium | beryllium | boron |
| AN   | 1        | 2      | 3       | 4         | 5     |
| mass | 1.01     | 4.00   | 6.94    | 9.01      | 10.81 |
| IE   | 13.6     | 24.6   | 5.4     | 9.3       | 8.3   |

---

<sup>46</sup> Camelcase is when separate words are capitalized and the spaces removed between them. This is sometimes used for variable or file names.

To access data in a DataFrame, place the column name in square brackets.

```
[in]: elements['Li']

[out]:
name      lithium
AN          3
mass       6.94
IE         5.4
Name: Li, dtype: object
```

Essentially what we get out of a column is a Series with the indices shown on the lefthand side.

To indicate a row, instead use the `loc` method. We again get a Series with indices derived from the column names in the source DataFrame. This Series can be placed in a variable and indexed just like in section 5.1.1.

```
[in]: elements.loc['IE']

[out]:
H      13.6
He     24.6
Li      5.4
Be      9.3
B       8.3
Name: IE, dtype: object

[in]: atomic_number = elements.loc['AN']

[in]: atomic_number['B']

[out]: 5
```

Alternatively, we can use the DataFrame directly and index it with the `loc` method as `[row, column]`.

```
[in]: elements.loc['IE', 'Li']

[out]: 5.4
```

Numerical index values can also be used with the `iloc` method. This reduces indexing to how NumPy arrays are indexed.

```
[in]: elements.iloc[2:, 2]

[out]:
```

```

mass      6.94
IE        5.4
Name: Li, dtype: object

```

A summary of the methods of indexing pandas Series and DataFrames is presented below in Table 5.1.

**Table 5.1** Summary of Pandas Indexing

| Index Method         | Description                                                        |
|----------------------|--------------------------------------------------------------------|
| s[index]             | Index Series with assigned index values                            |
| s.iloc[index]        | Index Series with default numerical index values                   |
| df[column]           | Index DataFrame with column name                                   |
| df.loc[row]          | Index DataFrame with row name                                      |
| df.loc[row, column]  | Index DataFrame with row and column names                          |
| df.iloc[row, column] | Index DataFrame with row and column default numerical index values |

## 5.2 Reading/Writing Data

Similar to NumPy, pandas contains multiple, convenient functions for reading/writing data directly to and from its own object types, and each function is suited to a specific file format. This includes CSV, HTML, JSON, SQL, Excel, and HDF5 files among others.<sup>47</sup>

**Table 5.2** Import/Export Functions in Pandas

| Argument                           | Description                                                    |
|------------------------------------|----------------------------------------------------------------|
| read_csv()<br>to_csv()             | Imports/Exports data from/to a CSV file                        |
| read_table()<br>to_table()         | General-purpose importer/exporter                              |
| read_hdf5()<br>to_hdf5()           | Imports/Exports data from/to an HDF5 file                      |
| read_clipboard()<br>to_clipboard() | Transfers data to/from the clipboard* to a Series or DataFrame |

---

<sup>47</sup> For more information on pandas IO tools, see: <https://pandas.pydata.org/pandas-docs/stable/io.html>.

`read_excel()`      Reads/writes an Excel file  
`to_excel()`

\* i.e., the copy and paste clipboard from your computer

### 5.2.1 General-Purpose Delimited File Reader

Before we start with more well-defined file formats, pandas provides a general purpose file reader `pd.read_table()`. This function imports text files where lines represent rows and the data in each row is separated by characters or spaces. The user can designate what character(s) separate the data by using the `delimiter` or `sep` arguments (they do the same thing), or as an easy way of breaking up data based on spaces, set the `delim_whitespace` equal to `True`. The function also includes a series of other arguments listed below in Table 5.3.

**Table 5.3** More `pd.read_table()` Arguments

| Argument                      | Description                                                                                              |
|-------------------------------|----------------------------------------------------------------------------------------------------------|
| <code>delimiter</code>        | Data separator; default is tab                                                                           |
| <code>sep</code>              | Data separator; default is tab                                                                           |
| <code>skiprows</code>         | Number of rows in file to skip before reading data                                                       |
| <code>skipfooter</code>       | Number of rows at the bottom of the file to skip                                                         |
| <code>skip_blank_lines</code> | If <code>True</code> , skips blank lines in file; default is <code>False</code>                          |
| <code>header</code>           | Row number to use for a data header; also accepts <code>None</code> if no header is provided in the file |
| <code>delim_whitespace</code> | Boolean argument indicating that data is separated by white space; default is <code>False</code>         |
| <code>skipinitialspace</code> | If <code>True</code> , skips white space after delimiter                                                 |

As an example, we can use this function to read a calculated PDB file of benzene and extract the *xyz* coordinates for each atom. This particular file type, shown below, is strictly formatted based on the number of spaces,<sup>48</sup> but being that all the data columns here have spaces between them, we can use space delimitation by setting `delim_whitespace=True`. Because the data do not start until the third line and we do not need the last thirteen lines of the file, we should exclude these rows. We set `header=None` because we do not want the function to treat the first line of data as a header or data label.

48 wwPDB Foundation PDB file documentation page. <http://www.wwpdb.org/documentation/file-format>

```

HEADER
REMARK
HETATM    1   H    UNK   0001      0.000   0.000  -0.020
HETATM    2   C    UNK   0001      0.000   0.000   1.067
HETATM    3   C    UNK   0001      0.000   0.000   3.857
HETATM    4   C    UNK   0001      0.000  -1.208   1.764
HETATM    5   C    UNK   0001      0.000   1.208   1.764
HETATM    6   C    UNK   0001      0.000   1.208   3.159
HETATM    7   C    UNK   0001      0.000  -1.208   3.159
HETATM    8   H    UNK   0001      0.000  -2.149   1.221
HETATM    9   H    UNK   0001      0.000   2.149   1.221
HETATM   10   H    UNK   0001      0.000   2.149   3.703
HETATM   11   H    UNK   0001      0.000  -2.149   3.703
HETATM   12   H    UNK   0001      0.000   0.000   4.943
CONECT    1     2
CONECT    2     1     5     4
CONECT    3     6     7    12
CONECT    4     7     2     8
CONECT    5     2     6     9
CONECT    6     5     3    10
CONECT    7     3     4    11
CONECT    8     4
CONECT    9     5
CONECT   10     6
CONECT   11     7
CONECT   12     3
END

```

```
[in]: benz = pd.read_table('benzene.pdb',
                           delim_whitespace=True, skiprows=2, skipfooter=13,
                           header=None)
```

```
[in]: benz
```

```
[out]:
```

|          |        | <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> |
|----------|--------|----------|----------|----------|----------|----------|----------|----------|----------|
| <b>0</b> | HETATM | 1        | H        | UNK      | 1        | 0.0      | 0.000    | -0.020   |          |
| <b>1</b> | HETATM | 2        | C        | UNK      | 1        | 0.0      | 0.000    | 1.067    |          |
| <b>2</b> | HETATM | 3        | C        | UNK      | 1        | 0.0      | 0.000    | 3.857    |          |
| <b>3</b> | HETATM | 4        | C        | UNK      | 1        | 0.0      | -1.208   | 1.764    |          |

|    |        |    |   |     |   |     |        |       |
|----|--------|----|---|-----|---|-----|--------|-------|
| 4  | HETATM | 5  | C | UNK | 1 | 0.0 | 1.208  | 1.764 |
| 5  | HETATM | 6  | C | UNK | 1 | 0.0 | 1.208  | 3.159 |
| 6  | HETATM | 7  | C | UNK | 1 | 0.0 | -1.208 | 3.159 |
| 7  | HETATM | 8  | H | UNK | 1 | 0.0 | -2.149 | 1.221 |
| 8  | HETATM | 9  | H | UNK | 1 | 0.0 | 2.149  | 1.221 |
| 9  | HETATM | 10 | H | UNK | 1 | 0.0 | 2.149  | 3.703 |
| 10 | HETATM | 11 | H | UNK | 1 | 0.0 | -2.149 | 3.703 |
| 11 | HETATM | 12 | H | UNK | 1 | 0.0 | 0.000  | 4.943 |

The  $x$ ,  $y$ , and  $z$  data are in columns 5, 6, and 7, respectively and can be extracted by indexing as discussed in section 5.1.2.

### 5.2.2 Comma Separated Values Files

Pandas provides a collection of more format-specific functions for reading/writing files. The most popular is possibly the CSV file because it is simple and many scientific instruments support exporting data in this format. To import a CSV file, we will use the `read_csv()` function. This function is very similar to the `read_table()` function except that a default value for the separator/delimiter is set to a comma. To create a CSV file, use the `to_csv()` method which at a minimum requires the file name and a pandas object with the data.

We can write the above chemical element data assembled in section 5.1 as shown below. Because we are starting from a pandas object and are using a pandas method, the `df.to_csv()` format is used where `df` is a DataFrame.

```
[in]: elements.to_csv('elements.csv')
```

If we check the directory containing the Jupyter notebook, there is a file titled `elements.csv` that looks like the following. Each row in the DataFrame is a different line in the file, and every column is separated by a comma.

```
,H,He,Li,Be,B
name,hydrogen,helium,lithium,beryllium,boron
AN,1,2,3,4,5
mass,1.01,4.0,6.94,9.01,10.81
IE,13.6,24.6,5.4,9.3,8.3
```

To read the data back in from the file, use `pd.read_csv()`. Because we are not starting with a pandas object, the function is called using the `pd.function()` format.

```
[in]: pd.read_csv('elements.csv')
```

| Unnamed:0 | H        | He     | Li      | Be        | B     |
|-----------|----------|--------|---------|-----------|-------|
| name      | hydrogen | helium | lithium | beryllium | boron |
| AN        | 1        | 2      | 3       | 4         | 5     |
| mass      | 1.01     | 4.00   | 6.94    | 9.01      | 10.81 |
| IE        | 13.6     | 24.6   | 5.4     | 9.3       | 8.3   |

### 5.2.3 Excel Notebook Files

Pandas provides another useful function for importing Excel notebook files (i.e., `.xls` or `.xlsx`). Because Excel files can contain multiple sheets, this function is a little more complicated to use. The simplest way to import an excel file is to use `pd.read_excel()` and provide it with the Excel file name.

```
[in]: pd.read_excel('test.xls')
```

```
[out]:
```

```
    x      y
0  1      1
1  2      4
2  3      9
3  4     16
4  5     25
5  6     36
6  7     49
```

In the above example, pandas assumes the first sheet in the file. If you want to access a different sheet in the file, you can specify this by using the `sheet_name` keyword argument. If you do not know the sheet name, the `sheetname` argument also accepts integer index values (i.e., 0 for the first sheet and so on).

```
[in]: data = pd.read_excel('test.xls', sheet_name='Sheet2')
      data
```

```
[out]:
```

```
    a      b
0  1  0.841471
1  2  0.909297
```

```
2 3 0.141120
3 4 -0.756802
4 5 -0.958924
5 6 -0.279415
6 7 0.656987
7 8 0.989358
8 9 0.412118
```

Alternatively, if you want to extract the sheet names, you can use the `sheets_names` method with the `ExcelFile` class as demonstrated below.

```
[in]: xl = pd.ExcelFile('test.xlsx')
       xl.sheet_names

[out]: ['Sheet1']
```

Writing to an Excel file requires two steps – generate an `ExcelWriter` engine and then write each sheet. The Excel writer offers more power in generating Excel files including embedding charts, conditional formatting, coloring cells, and other tasks; but we will stick to the basics here.

```
[in]: writer = pd.ExcelWriter('new_file.xls')
       data.to_excel(writer, 'First Sheet')
       writer.save()
```

#### 5.2.4 Computer Clipboard

Pandas will also accept data from the computer's copy and paste clipboard. Start by highlighting some data from a webpage or a spreadsheet, select copy,<sup>49</sup> and then use the `pd.read_clipboard()` function to convert it to a pandas DataFrame.

```
[in]: pd.read_clipboard()

[out]:
```

|   | numbers | squares |
|---|---------|---------|
| 0 | 1       | 1       |
| 1 | 2       | 4       |
| 2 | 3       | 9       |
| 3 | 4       | 16      |
| 4 | 5       | 25      |

<sup>49</sup> This is typically located under the Edit menu of most software applications. Alternatively, you can type Command + C on a Mac or Control + C on Windows and Linux.

|          |   |    |
|----------|---|----|
| <b>5</b> | 6 | 36 |
| <b>6</b> | 7 | 49 |

Loading data from the clipboard is not a robust and efficient way to do much of your automated data analysis, but it is a very convenient method to experiment with data or to quickly grab some data off a website to experiment with.

## 5.3 Examining Data with Pandas

Once you load data into pandas, you will likely want to get an idea of what the data look like before you proceed to calculations and in-depth analyses. This section covers a few methods provided in pandas to gain a preliminary understanding of your data.

### 5.3.1 Descriptive Functions

Pandas provides a few simple functions to view and describe new data. The first two are `head()` and `tail()` which allow you to see the top and bottom of the DataFrame, respectively. These are particularly useful when dealing with very large DataFrames. Below, a DataFrame containing random values in an even, normal, and poisson distribution ( $\lambda = 3.0$ ) demonstrates these functions.

```
[in]: random = pd.DataFrame({'even': np.random.rand(1000),
                             'normal': np.random.randn(1000),
                             'poisson':
                                np.random.poisson(lam=3.0, size=1000)})

[in]: radom.head()

[out]:
```

|          | <b>even</b> | <b>normal</b> | <b>poisson</b> |
|----------|-------------|---------------|----------------|
| <b>0</b> | 0.050967    | 0.99457       | 4              |
| <b>1</b> | 0.040849    | 0.751054      | 2              |
| <b>2</b> | 0.858292    | -1.864487     | 3              |
| <b>3</b> | 0.846110    | -0.139636     | 0              |
| <b>4</b> | 0.768479    | 0.363213      | 1              |

```
[in]: random.tail()
```

|            | <b>even</b> | <b>normal</b> | <b>poisson</b> |
|------------|-------------|---------------|----------------|
| <b>994</b> | 0.547927    | -0.003141     | 3              |
| <b>995</b> | 0.254816    | 0.707050      | 5              |
| <b>997</b> | 0.809388    | -0.244430     | 4              |
| <b>998</b> | 0.339278    | -0.624394     | 3              |
| <b>999</b> | 0.322744    | 0.046268      | 3              |

Pandas also contains a `describe()` function that returns a variety of statistics on each column. For example, the mean is provided which are approximately 0.5, 0.0, and 3.0 for the even, normal, and poisson distributions, respectively. This is not surprising being that the even distribution is centered around 0.5, the normal around 0.0, and the poisson distribution is generated for an average of 3.0. The user is also provided with the minimum, maximum, standard deviation, and the quartile boundaries.

[in]: `random.describe()`

[out]:

|              | <b>even</b> | <b>normal</b> | <b>poisson</b> |
|--------------|-------------|---------------|----------------|
| <b>count</b> | 1000.00000  | 1000.00000    | 1000.00000     |
| <b>mean</b>  | 0.511100    | 0.039563      | 2.947000       |
| <b>std</b>   | 0.293089    | 0.986080      | 1.772945       |
| <b>min</b>   | 0.000528    | -4.827215     | 0.000000       |
| <b>25%</b>   | 0.251197    | -0.608324     | 2.000000       |
| <b>50%</b>   | 0.522237    | 0.047438      | 3.000000       |
| <b>75%</b>   | 0.773007    | 0.712473      | 4.000000       |
| <b>max</b>   | 0.999466    | 3.087794      | 10.000000      |

Another useful function is the `value_counts()` method which returns all unique values in a Series (or DataFrame column or row). Below, it is demonstrated on the poisson column being that the other two columns will have a relatively large number of unique values.

[in]: `counts = random['poisson'].value_counts()`  
`counts`

[out]:

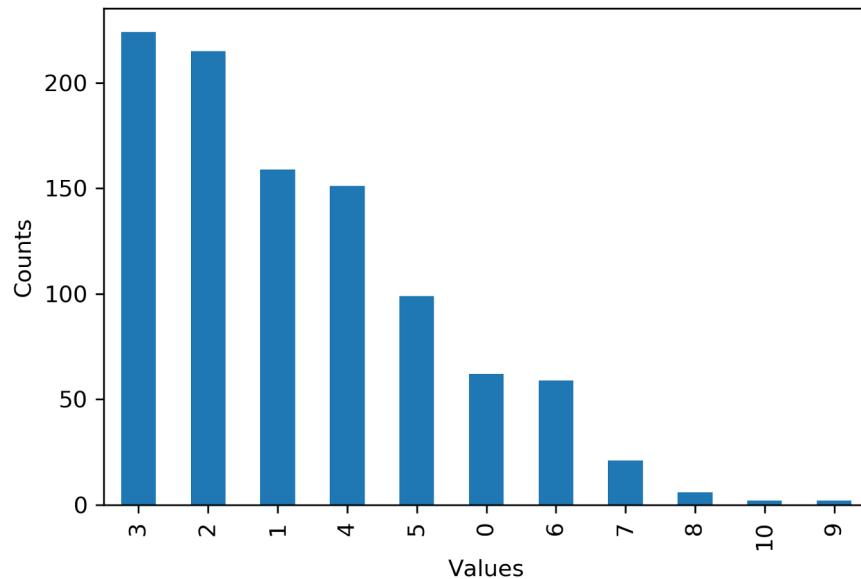
3 224

```
2      215
1      159
4      151
5      99
0      62
6      59
7      21
8      6
10     2
9      2
Name: poisson, dtype: int64
```

Data in DataFrames can be plotting by calling the desired columns of data and feeding them into plotting functions like `plt.scatter()`. The data can also be visualized by using the `df.plot(kind=)` format where `df` is the DataFrame and `kind` is the plot type (e.g., 'bar', 'hist', 'scatter', 'line', 'pie', etc...). However, this is just matplotlib doing the plotting and is largely redundant with other methods already covered. Below is a quick example of the `counts` data generated above.

```
[in]: counts.plot(kind='bar')
```

```
[out]:
```



### 5.3.2 Broadcasted Mathematical Operations

Because pandas is built upon NumPy arrays, mathematical operations are propagated through Series and DataFrames. The user is able to use NumPy methods on pandas objects,

and there are a number of other mathematical operations to chose from such as those listed below.

**Table 5.4** Broadcasted Pandas Methods

| Function | Description                     |
|----------|---------------------------------|
| abs      | Absolute value                  |
| count    | Counts items                    |
| cumsum   | Cumulative sum                  |
| cumprod  | Cumulative product              |
| mad      | Mean absolute deviation         |
| max      | Maximum                         |
| min      | Minimum                         |
| mean     | Mean                            |
| median   | Median                          |
| mode     | Mode                            |
| std      | Standard deviation <sup>a</sup> |

<sup>a</sup> Here the default delta degree of freedom (`ddof`) equals one unlike NumPy where the default is zero.

## 5.4 Modifying DataFrames

Now that you are able to generate DataFrames, it is useful to be able to modify them as you clean your data or perform calculations. This can be done through methods such as assignment, dropping rows and columns, and combining DataFrames or Series.

### 5.4.1 Insert Columns via Assignment

Possibly the easiest method of adding a new column is through assignment. If a *nonexistent* column is called and assigned values, instead of returning an error, pandas creates a new column with the given name and populates it with the data. For example, the `elements` DataFrame below does not contain a carbon column, so the column is added when assigned to a Series with the data.

```
[in]: elements  
[out]:
```

|      | H        | He     | Li      | Be        | B     |
|------|----------|--------|---------|-----------|-------|
| name | hydrogen | helium | lithium | beryllium | boron |
| AN   | 1        | 2      | 3       | 4         | 5     |
| mass | 1.01     | 4.00   | 6.94    | 9.01      | 10.81 |
| IE   | 13.6     | 24.6   | 5.4     | 9.3       | 8.3   |

```
[in]: elements['C'] = ['carbon', 6, 12.01, 11.3]
elements
```

[out] :

|      | H        | He     | Li      | Be        | B     | C      |
|------|----------|--------|---------|-----------|-------|--------|
| name | hydrogen | helium | lithium | beryllium | boron | carbon |
| AN   | 1        | 2      | 3       | 4         | 5     | 6      |
| mass | 1.01     | 4.00   | 6.94    | 9.01      | 10.81 | 12.01  |
| IE   | 13.6     | 24.6   | 5.4     | 9.3       | 8.3   | 11.3   |

### 5.4.2 Automatic Alignment

Another important feature of pandas is the ability to automatically align data based on labels. In the above example, carbon is added to the DataFrame with the name, atomic number, atomic mass, and ionization energy in the same order as in the DataFrame. What happens if the new data is not in the correct order? If we are using NumPy, this would require additional effort on the part of the user to reorder the data. However, if each value is labeled, pandas will see to it that they are placed in the correct location.

```
[in]: nitrogen = pd.Series([7, 14.01, 'nitrogen', 14.5],
                           index=['AN', 'mass', 'name', 'IE'])
nitrogen
[out] :
```

```
AN          7
mass      14.01
name    nitrogen
IE        14.5
dtype: object
```

Data for nitrogen is placed in a Series above. Notice that the values are out of order with respect to the data in `elements`. There are index labels (i.e., row labels) that tell pandas

what each piece of data is, and pandas will use them to determine where to place the new information.

```
[in]: elements['N'] = nitrogen  
elements
```

```
[out]:
```

|      | H        | He     | Li      | Be        | B     | C      | N        |
|------|----------|--------|---------|-----------|-------|--------|----------|
| name | hydrogen | helium | lithium | beryllium | boron | carbon | nitrogen |
| AN   | 1        | 2      | 3       | 4         | 5     | 6      | 7        |
| mass | 1.01     | 4.00   | 6.94    | 9.01      | 10.81 | 12.01  | 14.01    |
| IE   | 13.6     | 24.6   | 5.4     | 9.3       | 8.3   | 11.3   | 14.5     |

The new column of nitrogen data has been added to `elements` with all pieces of data residing in the correct row.

### 5.4.3 Dropping Columns

When cleaning up data, you may wish to drop a column or row. Pandas provides the `drop()` method for this purpose. It requires the name of the column or row to be dropped, and by default, it assumes a row, `axis=0`, is to be dropped. If you want to drop a column, change the axis using the `axis=1` argument. Below, the hydrogen column is dropped from the `elements` DataFrame.

```
[in]: elements.drop('H', axis=1)
```

|      | He     | Li      | Be        | B     | C      | N        |
|------|--------|---------|-----------|-------|--------|----------|
| name | helium | lithium | beryllium | boron | carbon | nitrogen |
| AN   | 2      | 3       | 4         | 5     | 6      | 7        |
| mass | 4.00   | 6.94    | 9.01      | 10.81 | 12.01  | 14.01    |
| IE   | 24.6   | 5.4     | 9.3       | 8.3   | 11.3   | 14.5     |

Next, the ionization energy (`IE`) row is dropped.

```
[in]: elements.drop('IE', axis=0)
```

|      | He     | Li      | Be        | B     | C      | N        |
|------|--------|---------|-----------|-------|--------|----------|
| name | helium | lithium | beryllium | boron | carbon | nitrogen |
| AN   | 2      | 3       | 4         | 5     | 6      | 7        |

|             |      |      |      |       |       |       |
|-------------|------|------|------|-------|-------|-------|
| <b>mass</b> | 4.00 | 6.94 | 9.01 | 10.81 | 12.01 | 14.01 |
|-------------|------|------|------|-------|-------|-------|

#### 5.4.4 Merge

To merge multiple DataFrames, pandas provides a `merge()` method. Similar to above, the `merge()` function will properly align data, but because DataFrames have multiple columns and index values to choose from, the `merge()` function can align data based on any of these values. The default behavior for `merge()` is to check for common columns between the two DataFrames and align the data based on those columns. As an example, below are two DataFrames containing data from various chemical compounds.

```
[in]: chmdata1 = [['MW', 58.08, 32.04], ['dipole', 2.91, 1.69], ['formula', 'C3H6O', 'CH3OH']]
chmdf1 = pd.DataFrame(chmdata1, columns=['property', 'acetone', 'methanol'])
```

```
[in]: chmdf1
```

|          | <b>property</b> | <b>acetone</b> | <b>methanol</b> |
|----------|-----------------|----------------|-----------------|
| <b>0</b> | MW              | 58.08          | 32.04           |
| <b>1</b> | dipole          | 2.91           | 1.69            |
| <b>2</b> | formula         | C3H6O          | CH3OH           |

```
[in]: chmdata2 = [['formula', 'C6H6', 'H2O'], ['dipole', 0.00, 1.85], ['MW', 78.11, 18.02]]
chmf2 = pd.DataFrame(chmdata2, columns=['property', 'benzene', 'water'])
```

```
[in]: chmdata2
```

|          | <b>property</b> | <b>benzene</b> | <b>water</b> |
|----------|-----------------|----------------|--------------|
| <b>0</b> | formula         | C6H6           | H2O          |
| <b>1</b> | dipole          | 0.00           | 1.85         |
| <b>2</b> | MW              | 78.11          | 18.02        |

Both DataFrames above have a `property` column, so the `merge()` function uses this common column to align all the data into a new DataFrame.

```
[in]: chmdf1.merge(chmf2)
```

```
[out]:
```

|   | property | acetone | methanol | benzene | water |
|---|----------|---------|----------|---------|-------|
| 0 | MW       | 58.08   | 32.04    | 78.11   | 18.02 |
| 1 | dipole   | 2.91    | 1.69     | 0.00    | 1.85  |
| 2 | formula  | C3H5O   | CH3OH    | C6H6    | H2O   |

If there are multiple columns with the same name, the user can specify which to use with the `on` keyword argument (e.g., `on='property'`). Alternatively, if the two DataFrames contain columns with different names that the user wants used for alignment, the user can specify which columns to use with the `left_on` and `right_on` keyword arguments.

```
[in]: comps1 = pd.DataFrame({'element': ['Co', 'Fe', 'Cr', 'Ni'],
                           'protons': [27, 26, 24, 28]})

comps2 = pd.DataFrame({'metal': ['Fe', 'Co', 'Cr', 'Ni'],
                       'IE': [7.90, 7.88, 6.79, 7.64]}))
```

In the two DataFrames above, each contains data on cobalt, iron, chromium, and nickel; but the first DataFrame labels metals as `element` while the second labels the metals as `metal`. The following merges the two DataFrames based on values in these two columns.

```
[in]: comps1.merge(comps2, left_on='element', right_on='metal')

[out]:
```

|   | element | protons | metal | IE   |
|---|---------|---------|-------|------|
| 0 | Co      | 27      | Co    | 7.88 |
| 1 | Fe      | 26      | Fe    | 7.90 |
| 2 | Cr      | 24      | Cr    | 6.79 |
| 3 | Ni      | 28      | Ni    | 7.64 |

Notice that the values in the `element` and `metal` columns were aligned in the resulting DataFrame. To get rid of one of the redundant columns, just use the `drop()` method described in section 5.4.3.

## 5.4.5 Concatenation

Concatenation is the process of splicing two DataFrames along a given axis. This is different from the `merge()` method above in that `merge()` merges and aligns common

data between the two DataFrames while `pd.concat()` blindly appends one DataFrame to another. As an example, imagine two lab groups measure the densities of magnesium, aluminum, titanium, and iron and load their results into DataFrames below.

```
[in]: group1 = pd.DataFrame({'metal':['Mg', 'Al', 'Ti', 'Fe'],
                             'density': [1.77, 2.73, 4.55,
   7.88]})

group2 = pd.DataFrame({'metal':['Al', 'Mg', 'Ti', 'Fe'],
                       'density': [2.90, 1.54, 4.12,
                                   8.10]})
```

```
[in]: group1
```

```
[out]:
```

|   | metal | density |
|---|-------|---------|
| 0 | Mg    | 1.77    |
| 1 | Al    | 2.73    |
| 2 | Ti    | 4.55    |
| 3 | Fe    | 7.88    |

See what happens when these two DataFrames are concatenated.

```
[in]: pd.concat((group1, group2))

[out]:
```

|   | metal | density |
|---|-------|---------|
| 0 | Mg    | 1.77    |
| 1 | Al    | 2.73    |
| 2 | Ti    | 4.55    |
| 3 | Fe    | 7.88    |
| 0 | Al    | 2.90    |
| 1 | Mg    | 1.54    |
| 2 | Ti    | 4.12    |
| 3 | Fe    | 8.10    |

Notice how the two DataFrames are appended with no consideration for common values in the `metal` column. The default behavior is to concatenate along the first axis (`axis=0`), but

this behavior can be modified with the `axis=` keyword argument. Again, the metals are not all aligned below because they were not in the same order in the original DataFrames.

```
[in]: pd.concat((group1, group2), axis=1)
```

```
[out]:
```

|   | metal | density | Metal | density |
|---|-------|---------|-------|---------|
| 0 | Mg    | 1.77    | Al    | 2.90    |
| 1 | Al    | 2.73    | Mg    | 1.54    |
| 2 | Ti    | 4.55    | Ti    | 4.12    |
| 3 | Fe    | 7.88    | Fe    | 8.10    |

For comparison, if the two DataFrames are merged instead of concatenating them, pandas will align the data based on the `metal` as demonstrated below. Because `density` appears twice as a column header, pandas deals with this by adding a suffix to differentiate between the two data sets.

```
[in]: pd.merge(group1, group2, on='metal')
```

```
[out]:
```

|   | metal | density_x | density_y |
|---|-------|-----------|-----------|
| 0 | Mg    | 1.77      | 1.54      |
| 1 | Al    | 2.73      | 2.90      |
| 2 | Ti    | 4.55      | 4.12      |
| 3 | Fe    | 7.88      | 8.10      |

## Further Reading

For further resources on the pandas library, see the following. The value of the pandas website cannot be emphasized enough as it contains a large quantity of high quality documentation and illustrative examples on using pandas for data analysis and processing.

1. Pandas Website. <http://pandas.pydata.org/>.
2. VanderPlas, J. *Python data Science Handbook: Essential Tools for Working with Data*, 1<sup>st</sup> ed.; O'Reilly: Sebastopol, CA, 2017, chapter 3. A free, online version is available by the author at <https://github.com/jakevdp/PythonDataScienceHandbook>.
3. McKinney, W. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, 2nd ed.; O'Reilly: Sebastopol, CA, 2018.

## Exercises

Complete the following using the pandas library. Avoid using `for` loops unless absolutely necessary.

1. Below is a table containing the melting points and boiling points of multiple common chemical solvents.

| Solvent  | bp  | mp   |
|----------|-----|------|
| benzene  | 80  | 6    |
| acetone  | 56  | -95  |
| toluene  | 111 | -95  |
| pentane  | 36  | -130 |
| ether    | 35  | -116 |
| ethanol  | 78  | -114 |
| methanol | 65  | -98  |

- a) Create a Series containing the boiling points of the above solvents with the solvent names as the indices. Call the Series to look up the boiling point of ethanol.
- b) Create a DataFrame that contains both the boiling points and melting points with the solvent names as the indices. Call the DataFrame to look up the melting point of benzene.

- c) Access the boiling point of pentane in the DataFrame from part *b* using numerical indices.
2. Import the attached file *blue1.csv* containing the absorption spectrum of Blue 1 food dye using pandas.
- Set the wavelengths as the index values.
  - Plot the absorption versus wavelength.
  - Determine the absorbance of Blue 1 at 620 nm.
3. Chemical Kinetics: Import the file *kinetics.csv* containing time series data for the conversion of A → Product using pandas IO tools. Generate new columns for  $\ln[A]$ ,  $[A]^{-1}$ , and  $[A]^{0.5}$  and determine the order of the reaction.<sup>50,51</sup>
4. Import the *ROH\_data.csv* file containing data on various simple alcohols to a DataFrame. Notice that this data is missing densities for some of the compounds.
- Use pandas to remove any rows with incomplete information in the density column using the `pd.dropna()` function. Check the DataFrame to see if it has changed.
  - Again using the `pd.dropna()` function, drop incomplete row with the parameter `inplace=True`. Check to see if the DataFrame has changed.
5. Import the following four files containing UV-vis spectra of four food dyes with the first column listing the wavelengths (nm) and the second column containing the absorbances. Each file contains data in from 400-850 nm in 1 nm increments.
- red40.csv*    *green3.csv*    *blue1.csv*    *yellow6.csv*
- Concatenation the files into a single DataFrame with the first column as the wavelength (nm) and the other four columns as the absorbances for each dye.
  - Replace the column headers with meaningful labels.
6. Import the two files *alcohols.csv* and *alkanes.csv* containing the boiling points of the two classes of organic compounds with respect to the number of carbons in each compound.

50 Half-order reactions have an integrated rate law shown here.  $[A]_t^{0.5} = [A]_i^{0.5} - \frac{1}{2}kt$

51 Meek, S. J.; Pitman, C. L.; Miller, A. J. M. Deducing Reaction Mechanism: A Guide for Students, Researchers, and Instructors. *J. Chem. Educ.* **2016**, 93 (2), 275–286.

- a) Drop the columns containing the names of the compounds.
- b) Merge the two DataFrames allowing pandas to align the two DataFrames based on carbon number.

# *Chapter 6*

## *Signal & Noise*

When collecting data from a scientific instrument, a measurement is returned as a value or series of values, and these values are composed of both signal and noise. The signal is the component of interest while the noise is random instrument response resulting from a variety of sources that can include the instrument itself, the sample holder, and even the tiny vibrations of the building. For the most interpretable data, you want the largest signal-to-noise ratio possible in order to reliably identify the features in the data.

This chapter introduces the processing of signal data including detecting features, removing noise from the data, and fitting the data to mathematical models. We will be using the NumPy library in this chapter and also start to use modules from the SciPy library. SciPy, short for “scientific python,” is one of the core libraries in the scientific python ecosystem. This library includes a variety of modules for dealing with signal data, performing Fourier transforms, and integrating sampled data among other common tasks for scientific data analysis. Table 6.1 summarizes some of the key modules in the SciPy library.

**Table 6.1** Common SciPy Modules

| Module                      | Description                                     |
|-----------------------------|-------------------------------------------------|
| <code>constants ()</code>   | Compilation of scientific constants             |
| <code>fftpack ()</code>     | Fourier transform functions                     |
| <code>integrate ()</code>   | Integration for both functions and sampled data |
| <code>interpolate ()</code> | Data interpolation                              |

|                         |                              |
|-------------------------|------------------------------|
| <code>io()</code>       | File importers and exporters |
| <code>linalg()</code>   | Linear algebra functions     |
| <code>optimize()</code> | Optimization algorithms      |
| <code>signal()</code>   | Signal processing functions  |

In contrast to NumPy, each module from SciPy needs to be imported individually,<sup>52</sup> so `import scipy` is not going help you much. Instead, you need to do the following to import a module.

```
[in]: from scipy import module
```

Alternatively, you can import a single function from a module.

```
[in]: from scipy.module import function
```

Because NumPy and plotting are used heavily in signal processing, the examples in this chapter assume the following NumPy and matplotlib imports.

```
[in]: import numpy as np
      import matplotlib.pyplot as plt
      %matplotlib inline
```

## 6.1 Feature Detection

When analyzing experimental data, there are typically key features in the signal that you are most interested in. Often, they are peaks or a series of peaks, but they can also be negative peaks (i.e., low point), the slopes, or an inflection points. This section covers extracting feature information from signal data.

### 6.1.1 Global Maxima & Minima

The simplest and probably most commonly sought after features in signal data are peaks and negative peaks. These are known as the *maxima* and *minima*, respectively, or collectively known as the *extrema*. In the simplest data, there may be only one peak or negative peak, so finding it is a matter of finding the maximum or minimum value in the data. For this, we can use NumPy's `np.maximum()` and `np.minimum()` functions, and

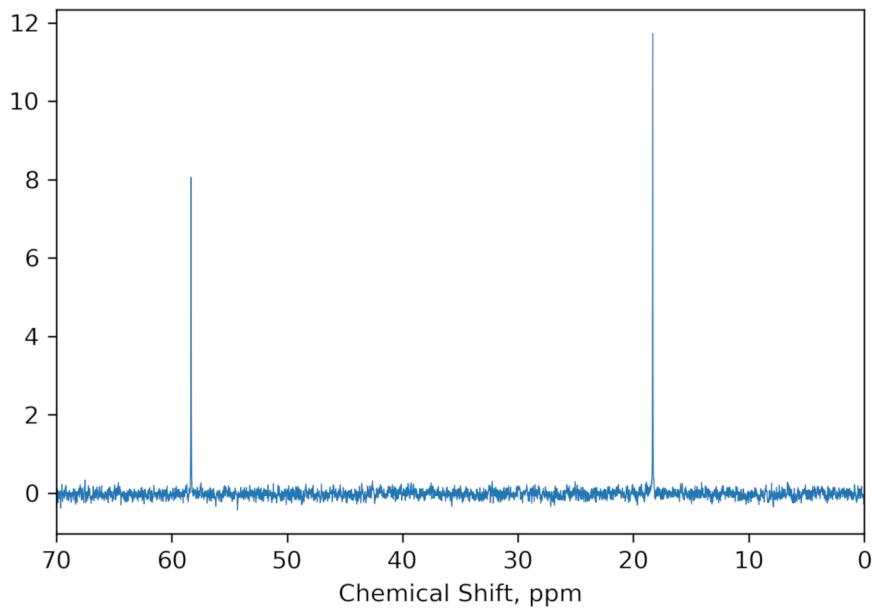
---

<sup>52</sup> Most libraries require that each module be imported explicitly like SciPy. NumPy is a major exception to this rule.

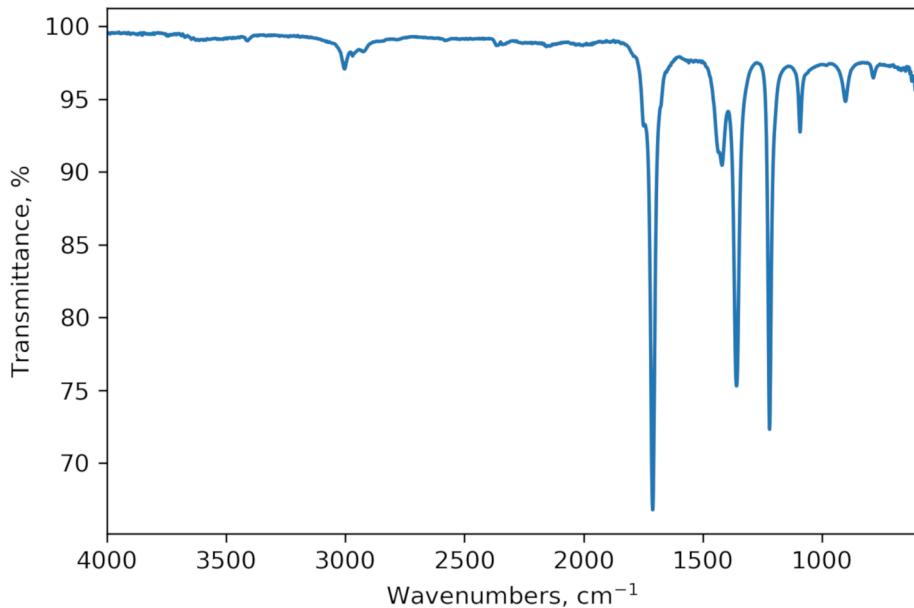
these functions can also be called using the shorter `np.max()` and `np.min()` function calls, respectively.

To demonstrate peak finding, we will use both a  $^{13}\text{C}\{^1\text{H}\}$  Nuclear Magnetic Resonance (NMR) spectrum and an infrared (IR) spectrum. These data are imported below using NumPy.

```
[in]: nmr = np.genfromtxt('13C_ethanol.csv', delimiter=',',  
                         skiprows=1)  
  
[in]: plt.plot(nmr[:,0], nmr[:,1], lw=0.5)  
      plt.xlabel('Chemical Shift, ppm')  
      plt.xlim(70, 0)  
  
[out]:
```



```
[in]: ir = np.genfromtxt('IR_acetone.csv', delimiter=',')  
  
[in]: plt.plot(ir[:,0], ir[:,1])  
      plt.xlabel('Wavenumbers, cm$^{-1}$')  
      plt.ylabel('Transmittance, %')  
      plt.xlim(4000, 600)
```



NMR resonances are positive peaks while IR stretches are represented here as negative peaks, so we can find the largest features in both spectra by finding the maximum value in the NMR spectrum and the smallest value in the IR spectrum.

```
[in]: np.max(nmr[:,1])
[out]: 11.7279863357544
[in]: np.min(ir[:,1])
[out]: 66.80017
```

These functions output the max and min values of the independent variable (*y*-axis). If we want to know the location on the *x*-axes, we need to use the NumPy functions `np.argmax()` and `np.argmin()` which return the indices of the max or min values instead of the actual value (“arg” is short for *argument*).

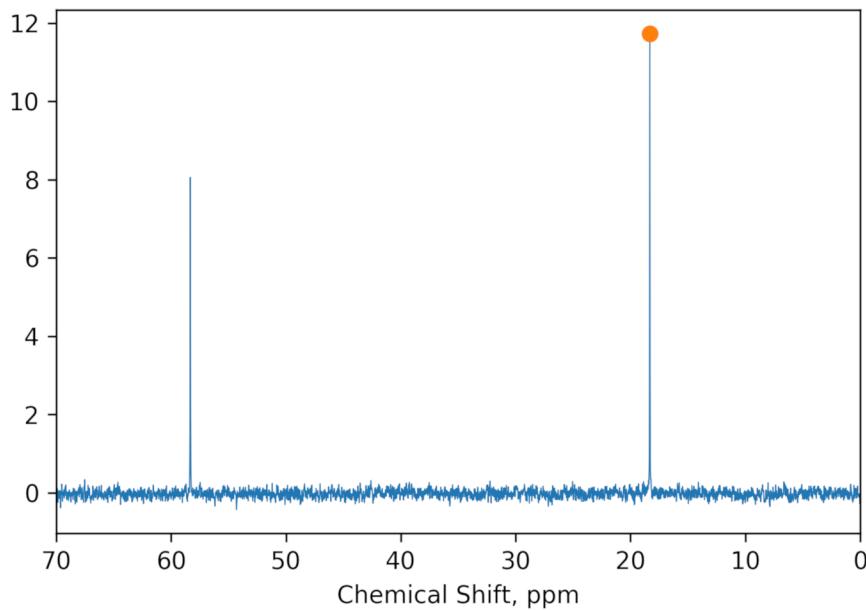
```
[in]: imax = np.argmax(nmr[:,1])
      imax
[out]: 5395
[in]: imin = np.argmin(ir[:,1])
      imin
[out]: 2302
```

With the indices, we can extract the desired information using indexing of the  $x$ -axes. Below, the largest peak in the NMR spectrum is at 18.3 ppm while the smallest transmittance (i.e., largest absorbance) is at 1710  $\text{cm}^{-1}$  in the IR spectrum.

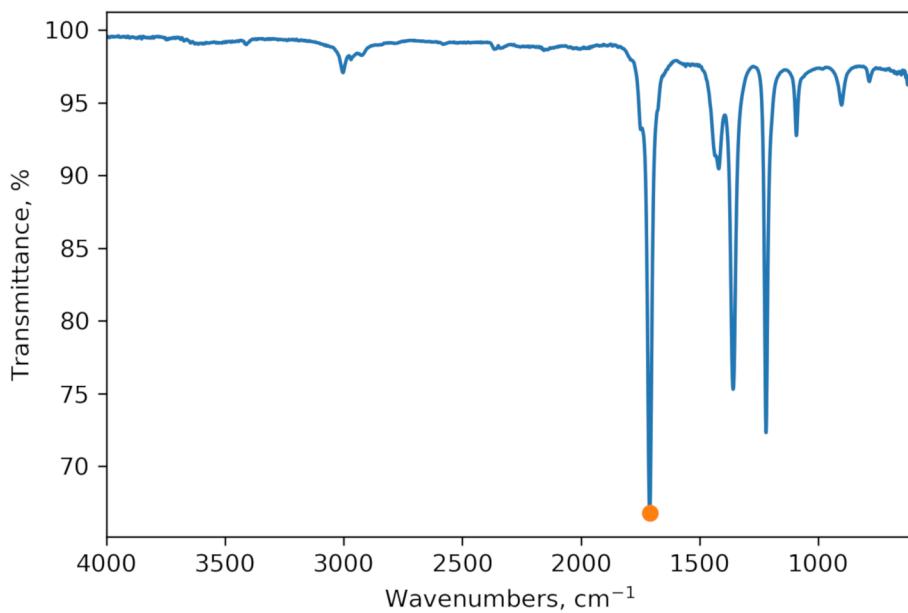
```
[in]: nmr[imax, 0]  
[out]: 18.312606267778  
  
[in]: ir[imin, 0]  
[out]: 1710.068
```

Below, these values are plotted on the spectra as orange dots to validate that they are indeed the largest features in the spectra.

```
[in]: plt.plot(nmr[:,0], nmr[:,1], lw=0.5)  
      plt.plot(nmr[imax,0], nmr[imax,1], 'o')  
      plt.xlabel('Chemical Shift, ppm')  
      plt.xlim(70,0)  
  
[out]:
```



```
[in]: plt.plot(ir[:,0], ir[:,1])  
      plt.plot(ir[imin, 0], ir[imin, 1], 'o')  
      plt.xlim(4000, 600)  
      plt.xlabel('Wavenumbers, cm$^{-1}$')  
      plt.ylabel('Transmittance, %')
```



Both of these functions find the *global extremes* (or *global extrema*). If all you need is the largest feature in a spectrum, this works just fine. To find multiple features, we will need to find the local extrema addressed in the following section.

### 6.1.2 Local Maximums & Minimums

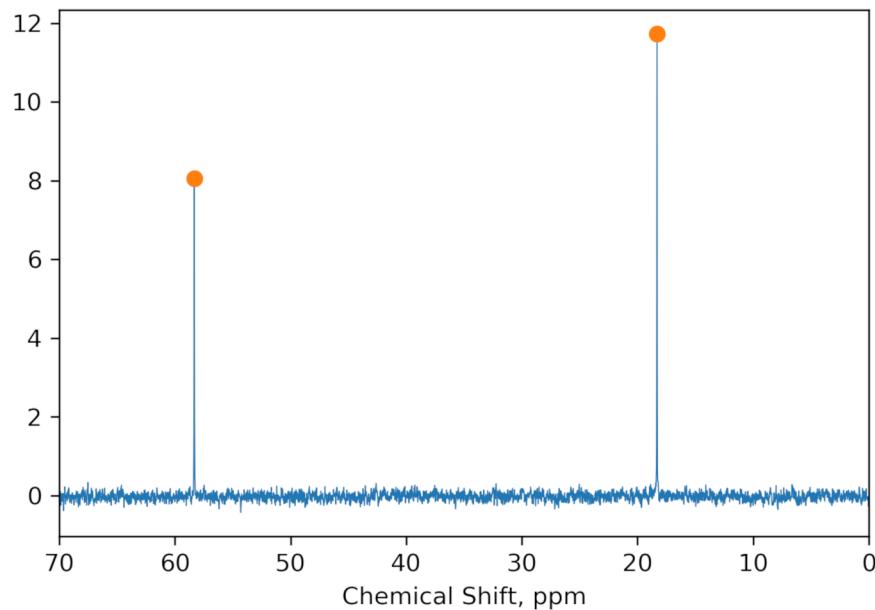
A considerable amount of data in science contain numerous peaks and negative peaks which are called *local extrema*. To locate the multiple max and min values, we will use SciPy's relative max/min functions `argrelextrema()` and `argrelmin()`. These functions determine if a point is a max/min by checking a range of data points on both sides to see if the point is the largest/smallest. The range of data points examined is known as the window, and the window can be modified using the `order` argument. Instead of the actual max/min values, these functions return the indices as the “arg” part of the name suggests.

```
[in]: from scipy.signal import argrelextrema, argrelmin
[in]: imax = argrelextrema(nmr[:,1], order=2000)
      imax
[out]: (array([1219, 5395]),)
```

The `argrelextrema()` function returned two indices as an array wrapped in a tuple.<sup>53</sup> If we plot the maxima marked with dots, we see that the function correctly identified both peaks.

```
[in]: plt.plot(nmr[:,0], nmr[:,1], lw=0.5)
plt.plot(nmr[imax, 0], nmr[imax, 1], 'C1o')
plt.xlabel('Chemical Shift, ppm')
plt.xlim(70,0)

[out]:
```



There are times when the `argrelextrema()` function will identify an edge or a point in a flat region as a local maximum because there is nothing larger near it. There are multiple ways to mitigate these erroneous peaks. First, we can increase the window for which the function checks to see if a point is the largest value in its neighborhood. Unfortunately, making the window too large can also prevent the identification of multiple extrema near each other. The second mitigation is to change the function's mode from the default 'clip' to 'wrap'. This makes the function treat the data as wrapped around on itself instead of stopping at the edge. That is, both edges of the data are treated as being connected. This makes it more likely that an extrema value is in the neighborhood.

---

<sup>53</sup> This may seem odd and annoying, but the function wraps the array in a tuple because the function may also be used on higher-dimensional data and return the local extrema along different dimensions as a series of arrays wrapped in the tuple.

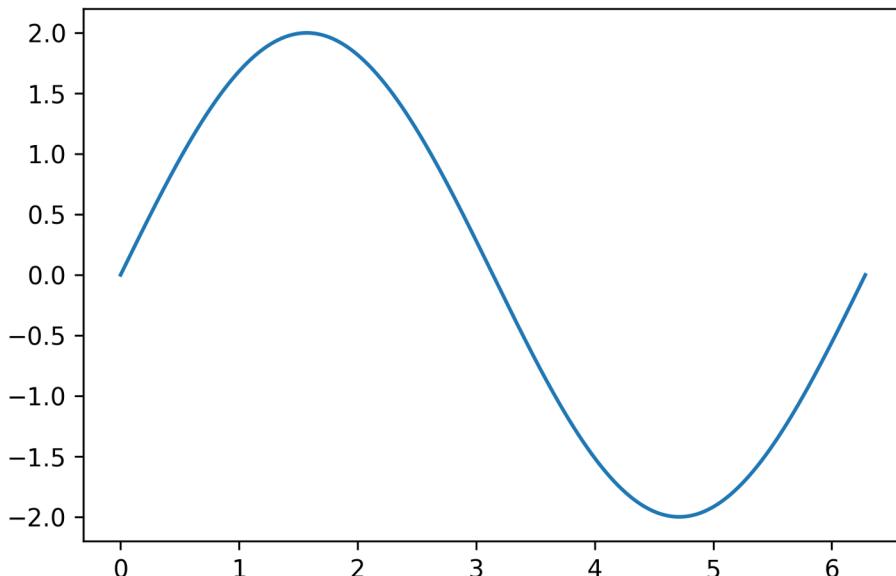
### 6.1.3 Slopes & Inflection Points

The slope is a useful feature as it can be used to identify inflection points, edges, and make subtle features in a curve more obvious. Noisy data can make it challenging to examine the slope as it causes the slope to fluctuate so much that it sometimes dwarfs the overall signal. It is recommended that the noise be first removed by signal smoothing covered in a section 6.2. To demonstrate the challenges, we will generate both noise-free and noisy synthetic data below and calculate the slopes.

```
[in]: x = np.linspace(0, 2*np.pi, 1000)
y_smooth = np.sin(x)
y_noisy = np.sin(x) + 0.07 * np.random.rand(len(x))

[in]: plt.plot(x, y_smooth)

[out]:
```

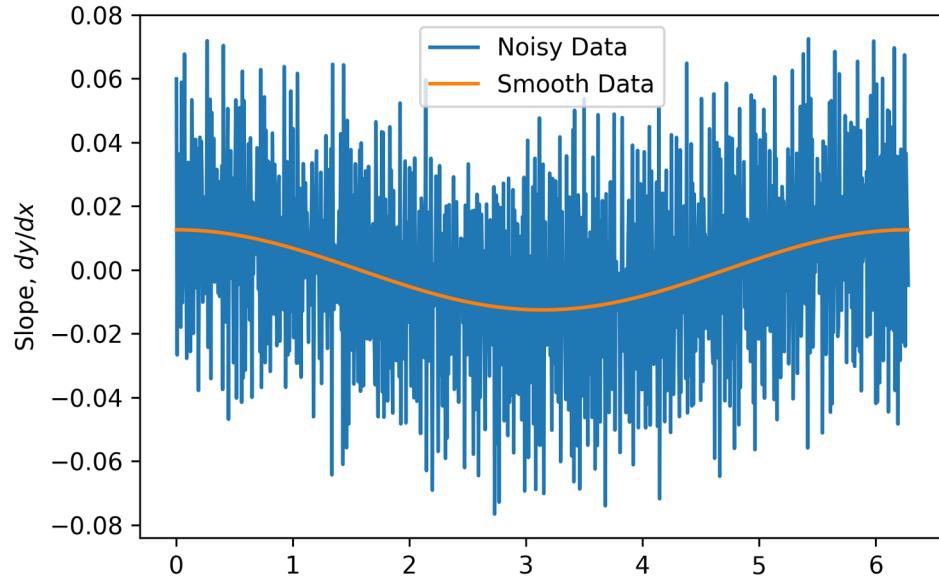


We will use NumPy to calculate the slope using the `np.diff()` function which calculates the derivative of a user defined order (`n`). Because the slope is the  $dy/dx$  between every pair of adjacent points, the resulting slope data is one data point shorter than the original data. This is important when plotting the data because the length of the `x` and `y` values must be the same.

```
[in]: dy_smooth = np.diff(y_smooth, n=1)
dy_noisy = np.diff(y_noisy, n=1)
x2 = (x[:-1] + x[1:]) / 2 # x values one shorter
```

```
[in]: plt.plot(x2, dy_noisy, label='Noisy Data')
plt.plot(x2, dy_smooth, label='Smooth Data')
plt.ylabel('Slope, dx/dy')
plt.legend()
```

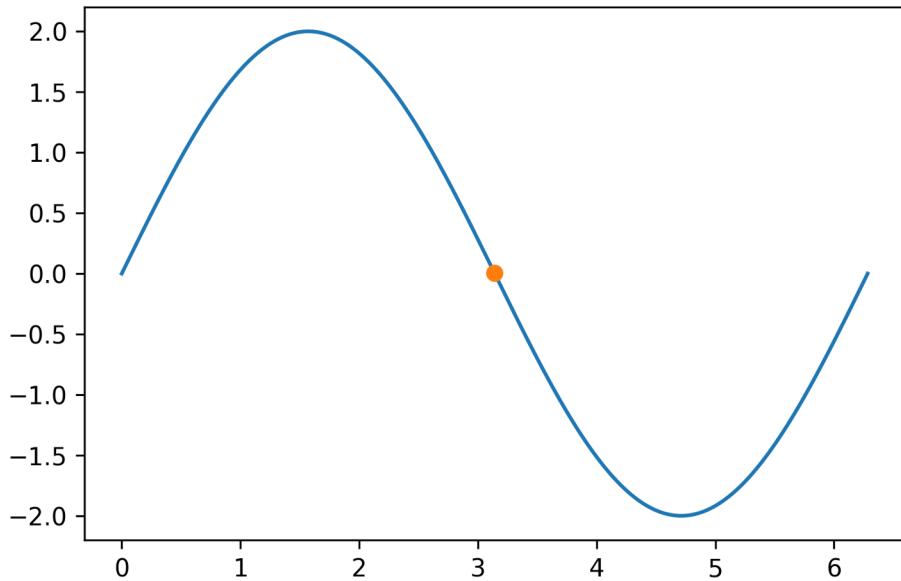
[out]:



As you can see, the slope from the noise dwarfs that of the main signal. We will use the derivative of the smooth data to find the inflection point below. Because the inflection point in the center of the data has a negative slope, we will need to find the minimum slope. This may not always be the case with other data.

```
[in]: i = np.argmin(dy_smooth) # finds min slope
plt.plot(x, y_smooth)
plt.plot(x[i], y_smooth[i], 'o')
```

[out]:



## 6.2 Smoothing Data

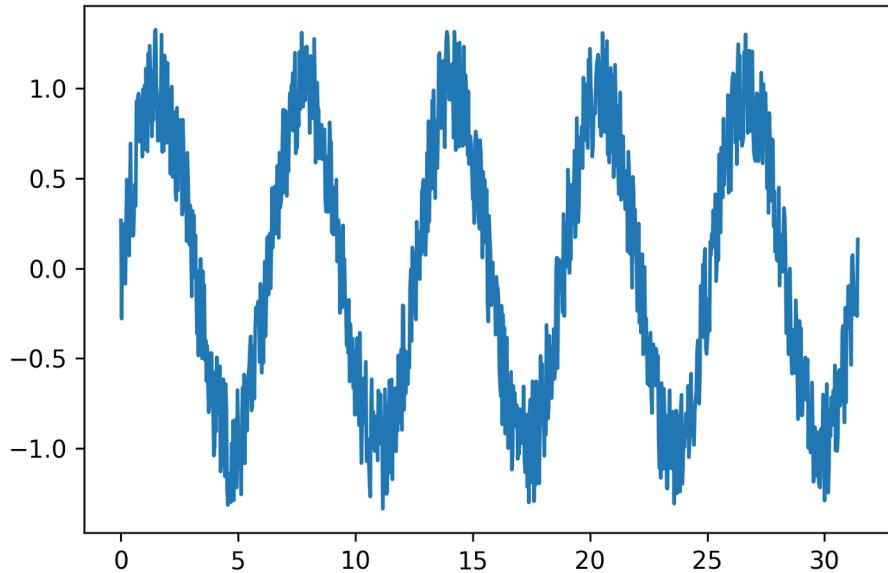
It is not uncommon to collect signal data that has a considerable amount of noise in it. Smoothing the data can help in the processing and analysis of the data such as making it easier to identify peaks or preventing the noise from hiding the extremes in the derivative of the data. Smoothing alters the actual data, so it is important to be transparent to others that the data were smoothed and how they were smoothed.

There are a variety of ways to smooth data including moving averages, band filters, and the Savitzky-Golay filter. We will focus on moving averages and Savitzky-Golay here. For this section, we will work with synthetic data with random noise generated below.

```
[in]: t = np.linspace(0, 10 * np.pi, 1000)
       signal = np.sin(t)
       noise = np.random.rand(1000)
       data = signal + 0.7 * (noise - 0.5)

[in]: plt.plot(t, data)

[out]:
```



### 6.2.1 Unweighted Average

The first and simplest way to smooth data is to *moving average* each data point with its immediate neighbors. This is an *unweighted sliding average smooth* or a *rectangular boxcar smooth*. From noisy data point  $D_j$ , we get smoothed data point  $S_j$  by the following equation where  $D_{j-1}$  and  $D_{j+1}$  are the points immediately preceding and following a data point  $D_j$ , respectively.

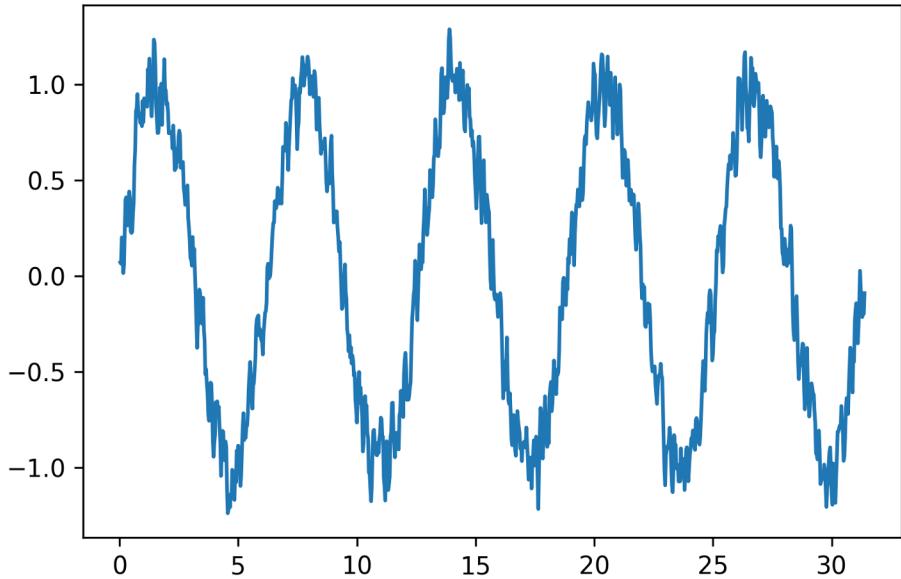
$$S_j = \frac{D_{j-1} + D_j + D_{j+1}}{3}$$

One thing to note about this smoothing method is that it is only valid for all points except the first and last because there are no data points both before and after them to take the average with. As a result, the smoothed data is two data points shorter. There are approximations that can be used to maintain the length of the data, but for simplicity, we will allow the data to shorten.

```
[in]: sum = data[:-2] + data[1:-1] + data[2:]
rect_smooth = sum / 3

[in]: plt.plot(t[1:-1], rect_smooth)

[out]:
```



The data is smoothed relative to the original data, but there is still a considerable amount of noise present.

### 6.2.2 Weighted Averages

The above method treats each point equally and only takes the average with the immediately adjacent data points. The *triangular smooth* approach averages extra data points with the points closer to the original point weighted more heavily than those further away. For example, if we take the average using five data points, this is described by the following equation.

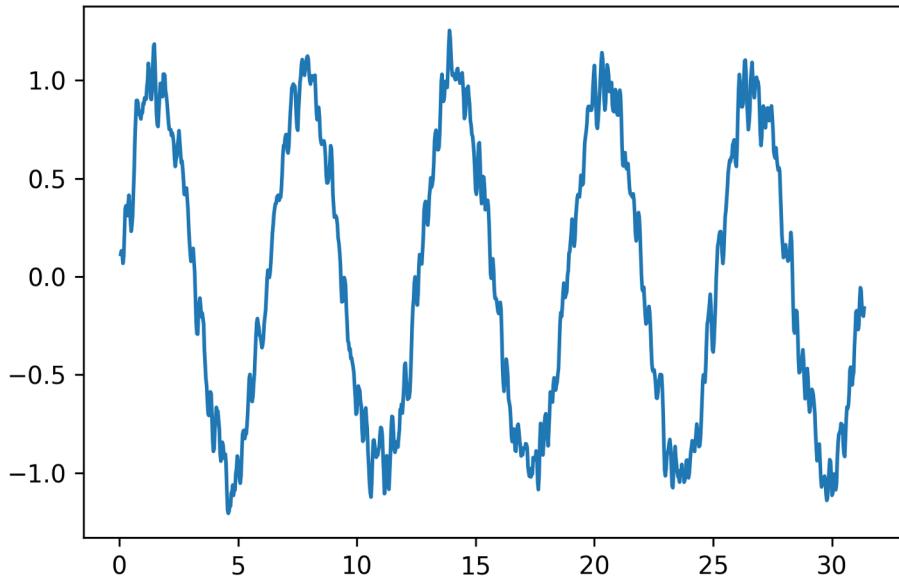
$$S_j = \frac{D_{j-2} + 2D_{j-1} + 3D_j + 2D_{j+1} + D_{j+2}}{5}$$

The resulting data is shortened by four points as the end points have insufficient neighbors to be averaged.

```
[in]: sum = data[:-4] + 2*data[1:-3] + 3*data[2:-2] +
        2*data[3:-1] + data[4:]
tri_smooth = sum / 9

[in]: plt.plot(t[2:-2], tri_smooth)

[out]:
```



The triangular smooth results in a smoother data set than the rectangular smooth. This is not surprising as applying the triangular smooth above is mathematically equivalent to applying the rectangular smooth twice.<sup>54</sup>

### 6.2.3 Median Smoothing

While the above filters take some form of the mean of the surrounding data points, a median filter takes the median. This filter is sometimes applied to images because it reduces noise while maintaining sharp edges.<sup>54</sup>

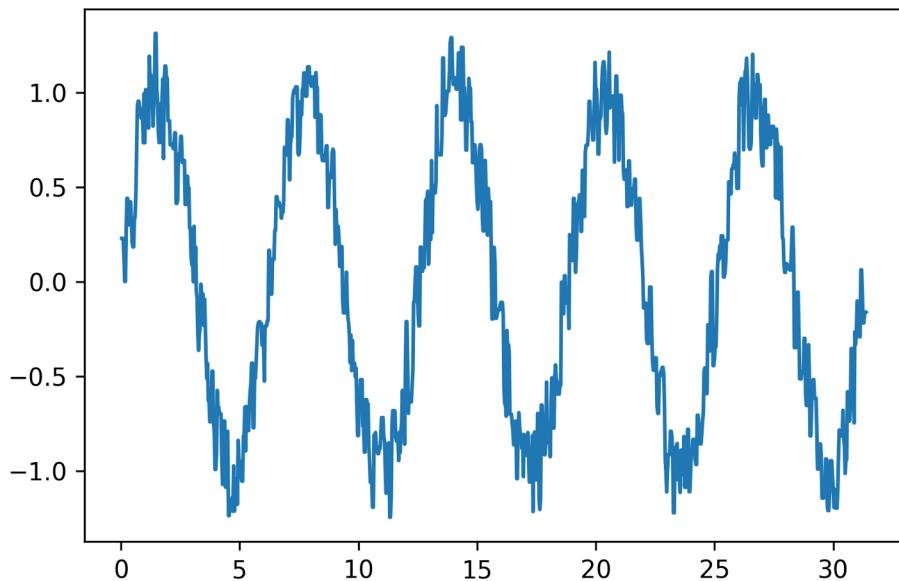
```
[in]: array2d = np.vstack((data[2:], data[1:-1], data[:-2]))
median_smooth = np.median(array2d, axis=0)

[in]: plt.plot(t[1:-1], median_smooth)

[out]:
```

---

<sup>54</sup> Paranjape, R. B. (2009). *Fundamental Enhancement Techniques*. In Bankman, I. N. (Ed.). *Handbook of Medical Image Processing and Analysis (section 1.4.2)*. Cambridge, MA: Academic Press.



#### 6.2.4 Savitzky–Golay

Another approach is the Savitzky–Golay filter which incrementally moves along the noisy data and fits sections (i.e., windows) of data points to a polynomial using least square minimization. While this approach had been previously described in the mathematical literature, Abraham Savitzky and M. J. E. Golay are known for applying it to spectroscopy.<sup>55</sup> Conveniently, SciPy contains a built-in function for this called `savgol_filter` from the `signal` module shown below.

```
scipy.signal.savgo_lfilter(data, window, polyorder)
```

This function requires three arguments which include the original data as a NumPy array, `window` which is the width of the moving window the savgol algorithm fits to a polynomial, and `polyorder` which is the order of polynomial used for the moving data fit. You are encouraged to play with the `window` and `polyorder` arguments to see what works the best for your application. However, `polyorder` must be less than the window size, and the `window` must be an odd integer.

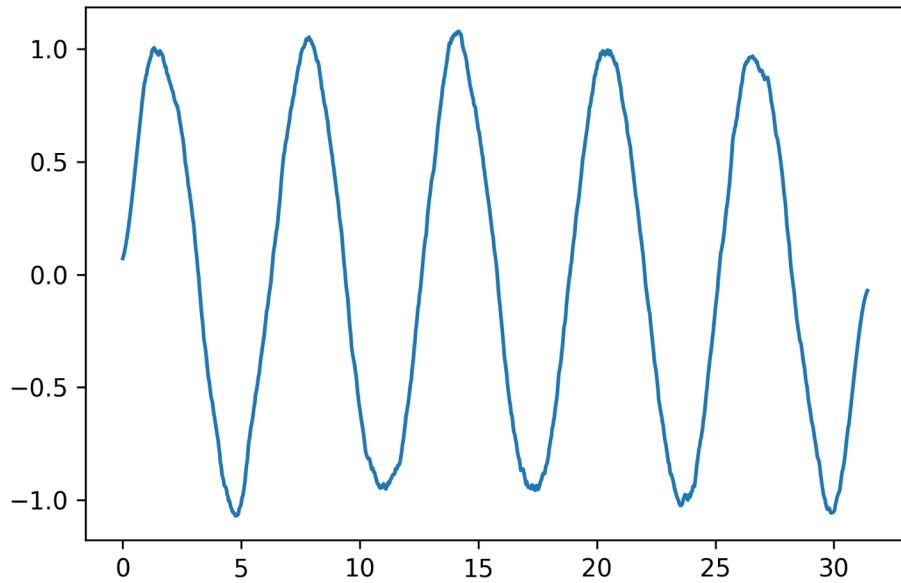
```
[in]: from scipy.signal import savgo_lfilter
      sg_smooth = savgo_lfilter(data, 51, 3)

[in]: plt.plot(t, sg_smooth)

[out]:
```

---

<sup>55</sup> The seminal paper being: Savitzky, A.; Golay, M.J.E. Smoothing and Differentiation of Data by Simplified Least Squares Procedures. *Anal. Chem.* **1964**, 36 (8) 1627–1639.



## 6.3 Fourier Transforms

Another approach to filtering noise is to filter based on frequency. Many times, random noise in data occurs at a different frequency than the data itself, and the noise can be reduced by filtering noise frequency ranges while maintaining signal frequencies. Often times, the random noise is higher frequency than the signal, so filtering out higher frequency noise is known as a *low-pass* filter. Alternatively, filtering out low-frequency noise is known as a *high-pass* filter, and filtering out noise above and below the signal frequency is known as a *band-pass* filter. Frequency filtering is somewhat involved being that we need to use window functions which are covered in the *Think DSP* book by Allen Downey listed at the end of this chapter. Instead, we will just look at the distribution of signal and noise frequencies in synthetic data. This is useful for analyzing the noise in data and also is used routinely in nuclear magnetic resonance (NMR) spectroscopy<sup>56</sup> and Fourier Transform infrared spectroscopy (FTIR).

To convert the data from the time domain to the frequency domain, we will use the fast *Fourier transform (FFT)* algorithm. This algorithm is only for data that is periodic. Below, synthetic data is generated oscillating at 62.0 Hz with some random noise to make it more interesting.<sup>57</sup>

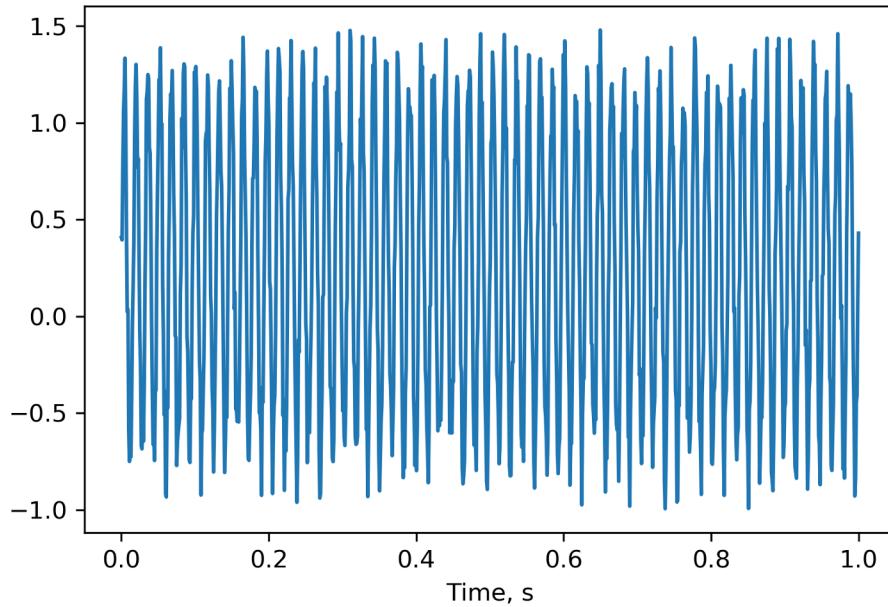
---

<sup>56</sup> Python has a library for working with NMR data called NMRglue covered in chapter 11. Much of the processing and visualization of the data is done with NumPy, SciPy, and matplotlib; but NMRglue helps with importing the NMR binary files into NumPy arrays for easy processing.

```
[in]: t = np.linspace(0,1,1000)
freq = 62.0 # Hz
signal = np.sin(freq*2*np.pi*t)
noise = np.random.rand(1000)
data = signal + 0.5 * noise

[in]: plt.plot(t, data)

[out]:
```



SciPy contains an entire module called `fftpack` dedicated to Fourier transforms and reverse Fourier transforms. We will use the basic `fft()` function for our synthetic data which returns a mixture of real and imaginary values. For plotting, we will simply look at the real component of the result using `.real`.<sup>58</sup>

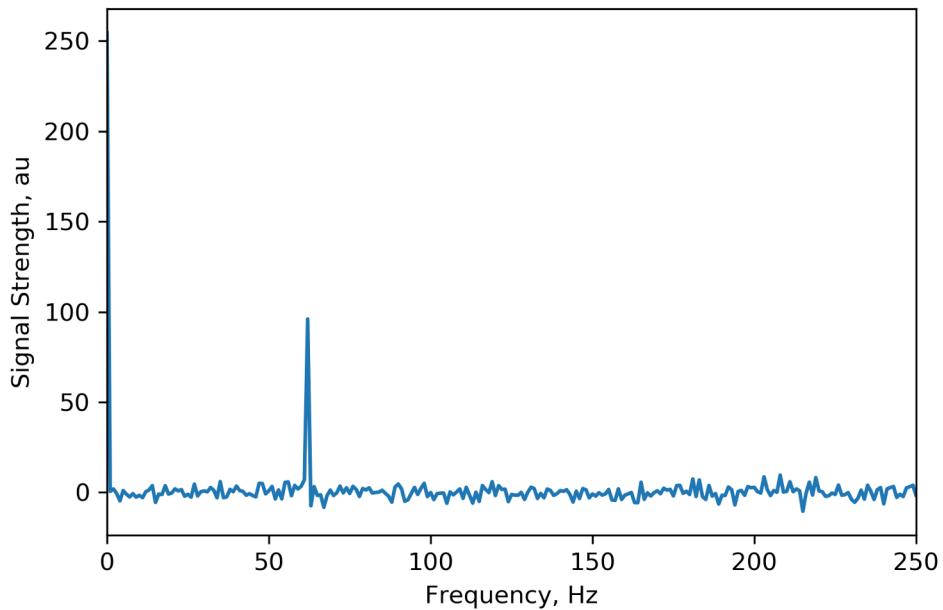
```
[in]: from scipy.fftpack import fft
fdata = fft(data)
plt.plot(fdata.real)
plt.xlim(0,500/2)
plt.xlabel('Frequency, Hz')
```

---

<sup>57</sup> The following example is inspired by an example from the SciPy website. <https://docs.scipy.org/doc/scipy-1.1.0/reference/tutorial/fftpack.html>. For the application of FFT to NMR spectroscopy, see chapter 11.

<sup>58</sup> The `.real` returns the real component of a complex number or data set while the `.imag` returns the imaginary component. If you are not familiar with complex numbers, this is not a critical concept in the present coverage of this topic.

```
plt.ylabel('Signal Strength, au')  
[out]:
```



Only the first half of the Fourier transform output is plotted above because the second half is a mirror image of the first. A single peak at 62.0 Hz is represent from our signal. The rest of the baseline of the plot is not smooth because there is noise present at a variety of frequencies. It is important to note that the erratic variations in the baseline of the frequency plot is not the noise itself but more like a histogram of all the frequencies present in the original data.

## 6.4 Fitting & Interpolation

Signal data or information taken from signal data often conforms to linear, polynomial, or other mathematical trends, and fitting data is important because it allows scientists to determine the equation describing the physical or chemical behavior of the data. In *data fitting*, the user provides the data and the general class of equation expected, and the software returns the *coefficients* for the equation. *Interpolation* is the method of predicting values in regions among known data points. The calculation of values where no data was collected can be accomplished by either using the coefficients derived from a curve fit or using a special interpolation function that generates a callable function to calculate the new data points. Both approaches are demonstrated below.

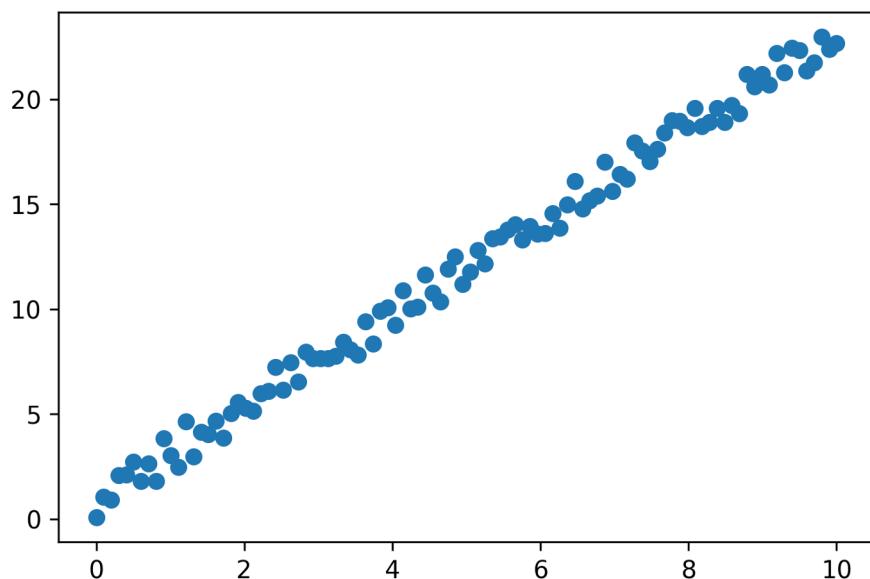
### 6.4.1 Polynomial Fit

Before we can do our fitting, we need some new, noisy data to examine. A linear set of data with added noise is generated below along with a second-order curve with the same noise.

```
[in]: x = np.linspace(0,10,100)
y = 2.2 * x
y2 = 3.4 * x**2 + 4 * x + 7
noise = np.random.rand(100)
y_noisy = y + 10 * noise
```

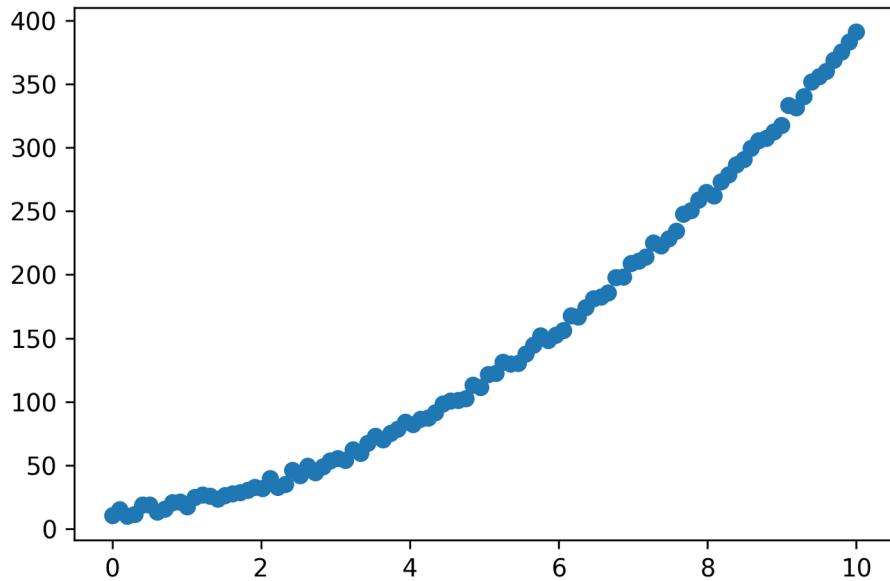
```
[in]: plt.scatter(x, y_noisy)
```

```
[out]:
```



```
[in]: plt.scatter(x, y2)
```

```
[out]:
```



Now we can fit the noisy data with a line using the NumPy `polyfit(x, y, degree)` function. The function takes the `x` and `y` data along with the degree of the polynomial.

A line is a first-degree polynomial, and the function returns an array containing the coefficients for the fit with the highest order coefficients first.

```
[in]: np.polyfit(x, y_noisy, 1)
[out]: array([ 2.20323392,  1.05364069])
```

For a linear equation of the form  $y = ax + b$ , we get an array of the form `array([a, b])`, so the fitted equation above is  $y = 2.2x + 1$ . The positive shift of the  $y$ -intercept above zero is not surprising being that we added random noise not centered around zero; the average of our `np.random.rand()` noise should be around 0.5, not zero.<sup>59</sup> We can also obtain the statistics for our fit using the `linregress` function from the SciPy `stats` module. Note that this does not return the  $r^2$  value but instead the  $r$ -value which can be squared to generate the  $r^2$  value.

```
[in]: from scipy import stats
       stats.linregress(x, y_noisy)
[out]: LinregressResult(slope=2.2342978321182261,
                       intercept=0.85347929923196553,
```

---

<sup>59</sup> This can be remedied either by subtracting 0.5 from the noise or using another random number generator such as the normal distribution (`randn()`) which is centered around zero.

```
rvalue=0.9959183667295306,
pvalue=3.506071633967805e-104,
stderr=0.020454678724776052)
```

Fitting to a polynomial of a higher order works the same way except that the order is above one. Below, the `polyfit()` function determines the equation to be  $y = 3.4x^2 + 3.7x + 13$ .

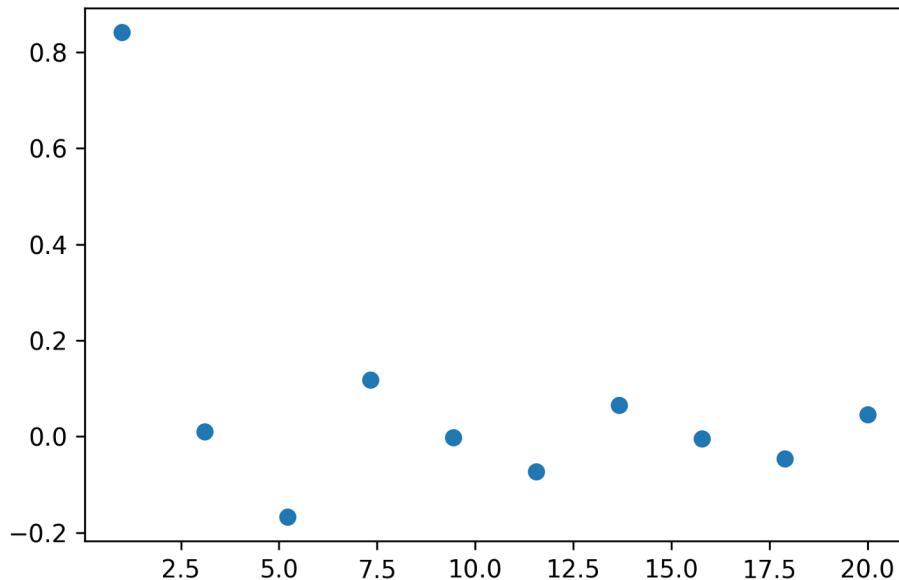
```
[in]: np.polyfit(x, y, 2)
[out]: array([ 3.41596168,  3.67943579, 13.00301123])
```

### 6.4.3 Interpolation

The practical difference between the `np.polyfit` function and the interpolation functions in SciPy is that the former returns coefficients for the equation while the interpolation functions return a Python function that can be used to calculate values. There are times when one is more desirable than the other depending upon your application. Below we will use the interpolation function to interpolate a one dimensional function.

Below is a dampening sine wave that we will interpolate from ten data points.

```
[in]: x = np.linspace(1,20, 10)
y = np.sin(x)/x
plt.plot(x,y, 'o')
```



To interpolate this one-dimensional function, we will use the `interp1d()` method from SciPy. Along with the `x` and `y` values, `interp1d()` requires a mode of interpolation using the `kind` keyword which can include the items listed in Table 6.2.

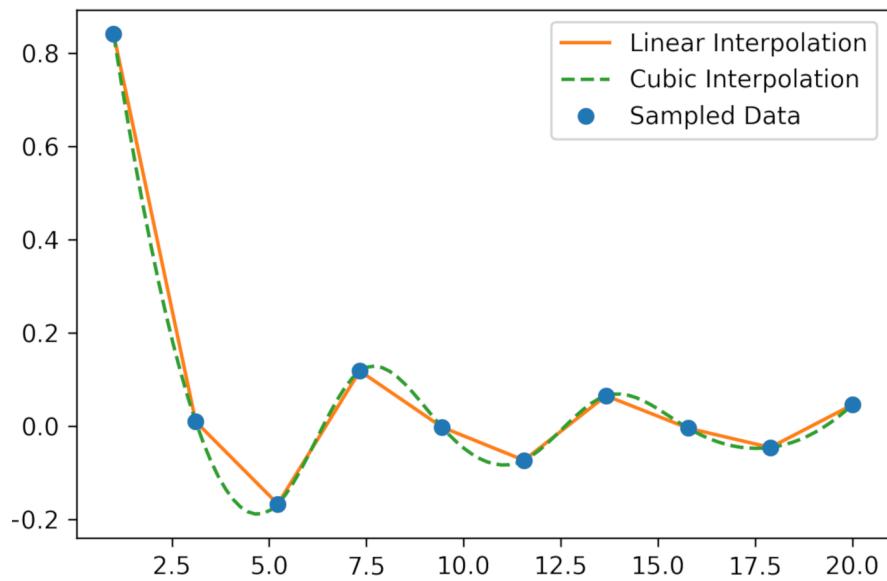
**Table 6.2** Modes for `interp1d()` Method

| Kind        | Description                                     |
|-------------|-------------------------------------------------|
| 'linear'    | Linear interpolation between data points        |
| 'zero'      | Constant value until the next data point        |
| 'nearest'   | Predicts values equaling the closest data point |
| 'quadratic' | Interpolates with a second-order spline         |
| 'cubic'     | Interpolates with a third-order spline          |

Below is a demonstration of both linear and cubic interpolation. The two functions `f()` and `f2()` are generated and can be used like any other Python function to calculate values.

```
[in]: from scipy import interpolate
       f = interpolate.interp1d(x, y, kind='linear')
       f2 = interpolate.interp1d(x, y, kind='cubic')

[in]: xnew = np.linspace(1,20, 100)
       plt.plot(xnew, f(xnew), 'C1-', label='Linear
                           Interpolation')
       plt.plot(xnew, f2(xnew), 'C2-', label='Cubic
                           Interpolation')
       plt.plot(x, y, 'C0o', label='Sampled Data')
       plt.legend()
```



## Further Reading

The ultimate authority on NumPy and SciPy are the Numpy & SciPy Documentation page listed below. As changes and improvements occur in these libraries, this is one of the best places to find information. For information on digital signal processing (DSP), there are numerous sources such as Allen Downey's *Think DSP* book or articles such as those listed below.

1. Numpy and Scipy Documentation. <https://docs.scipy.org/doc/>
2. Downey, Allen B. *Think DSP*, Green Tea Press, 2016.  
<http://greenteapress.com/wp/think-dsp/>.
3. O'Haver, T. C. An Introduction to Signal Processing in Chemical Measurement. *J. Chem. Educ.* **1991**, 68 (6), A147-A150.
4. Savitzky, A.; Golay, M.J.E. Smoothing and Differentiation of Data by Simplified Least Squares Procedures. *Anal. Chem.* **1964**, 36 (8) 1627–1639.

## Exercises

### Feature Detection

1. Import the file *CV\_K3Fe(CN)6.csv* which contains a cyclic voltammogram for potassium cyanoferrate. Plot the data with the green dots on the highest point(s) and red triangles on the lowest point(s).
2. Import the file titled *CV\_K3Fe(CN)6.csv* and determine the inflection point. Plot the data with a marker on the inflection point.

### Signal & Noise

3. Generate noisy synthetic data from the following code.

```
from scipy.signal import sawtooth
import numpy as np
t = np.linspace(0, 4, 1000)
sig = sawtooth(2 * np.pi * t) + np.random.rand(1000)
```

- a) Smooth the data using moving averages and plot the smoothed signal. Feel free to use the moving averages code from this chapter.
- b) Smooth the same data using a Savitzky–Golay filter. Plot the smoothed signal.

4. Import the  $^{31}\text{P}$  NMR file titled *fid\_31P.csv* and determine the number of major frequencies are in this wave. Keep in mind that there will be a second echo for each peak.

## **Fitting & Interpolation**

5. The wavelength of emitted light ( $\lambda$ ) from hydrogen is related to the electron energy level transitions by the following equation where  $R_{\infty}$  is the Rydberg constant,  $n_i$  is the initial principle quantum number of the electron, and  $n_f$  is the final principle quantum number of the electron.

$$\frac{1}{\lambda} = R_{\infty} \left( \frac{1}{n_f^2} - \frac{1}{n_i^2} \right)$$

The following is experimental data of the wavelengths for five different transitions from the Balmer series (i.e.,  $n_f = 2$ ).

| Transition ( $n_i \rightarrow n_f$ ) | Wavelength (nm) |
|--------------------------------------|-----------------|
| $3 \rightarrow 2$                    | 656.1           |
| $4 \rightarrow 2$                    | 485.2           |
| $5 \rightarrow 2$                    | 433.2           |
| $6 \rightarrow 2$                    | 409.1           |
| $7 \rightarrow 2$                    | 396.4           |

Calculate a value for the Rydberg constant ( $R_{\infty}$ ) using a linear fit of the above data. The data will need to be first linearized.

6. The following data is for the initial rate of a chemical reaction for different concentrations of starting material (A). Calculate a rate constant ( $k$ ) for this reaction using a nonlinear fit.

| [A] <sub>i</sub> (M) | Rate (M/s) |
|----------------------|------------|
| 0.10                 | 0.0034     |
| 0.16                 | 0.0087     |
| 0.20                 | 0.014      |
| 0.25                 | 0.021      |
| 0.41                 | 0.057      |
| 0.55                 | 0.10       |

7. A colorimeter exhibits the following absorbances for known concentrations of Red 40 food dye. Generate a calibration curve using the data below and then calculate the concentration of Red 40 dye in a pink soft drink with an absorbance of 0.181.

| Absorbance<br>(@ 504 nm) | [Red 40]<br>( $10^{-5}$ M) |
|--------------------------|----------------------------|
| 0.125                    | 0.150                      |
| 0.940                    | 1.13                       |
| 2.36                     | 2.84                       |
| 2.63                     | 3.16                       |
| 3.31                     | 3.98                       |
| 3.77                     | 4.53                       |

8. The following are points on the 2s radial wave function ( $\psi$ ) for a hydrogen atom with respect to the radial distance from the nucleus in Bohrs ( $a_0$ ). Visualize the radial wave function as a smooth curve by interpolating the following data points.

| Radius ( $a_0$ ) | $\psi$   |
|------------------|----------|
| 1.0              | 0.21     |
| 5.0              | -0.087   |
| 9.0              | -0.027   |
| 13.0             | -0.0058  |
| 17.0             | -0.00108 |



# *Chapter 7*

## *Image Processing & Analysis*

Images are a major data format in chemistry. They can be electron microscope images of a surface, photos of a reaction, or images from fluorescence microscopy. Image processing and analysis can be performed using software like Photoshop or GIMP,<sup>60</sup> but this can be tedious and subjective when done manually. A better alternative is to have software automate the entire process to provide consistent, precise, and objective processing of images and taking measurements of their features.

Among the more popular Python library for performing scientific image analysis is scikit-image. This is a library specifically designed for scientific image analysis and includes a wide variety of tools for the processing and extracting information from images. Examples of tools in scikit-image include functions for boundary detection, object counting, entropy quantification, color space conversion, image comparison, and many others. Even though there are other Python libraries for working with images, such as pillow,<sup>61</sup> scikit-image is designed for scientific image analysis while pillow is intended for more fundamental operations such as image rotation and cropping.

Unlike NumPy which automatically imports all modules with the library, scikit-image requires each module be imported explicitly. For example, if the user wants to import the color module, it is imported using the following code.

```
[in]: from skimage import color
```

---

<sup>60</sup> GIMP (GNU Image Manipulation Program) is a free, open-source image manipulation program available for free at <https://www.gimp.org/>.

<sup>61</sup> See Further Reading

Multiple modules can also be imported in a single import such as below. A list of modules and their description are shown in Table 7.1, and additional information can be found on the project website at <http://scikit-image.org/>.

```
[in]: from skimage import color, data, io
```

Because we will be doing some plotting, this chapter assumes the following matplotlib import and that inline plotting is enabled.

```
[in]: import matplotlib.pyplot as plt  
%matplotlib inline
```

**Table 7.1** Scikit-image Modules and Descriptions

| Module            | Description                                                                       |
|-------------------|-----------------------------------------------------------------------------------|
| color             | Covers images between color spaces                                                |
| data              | Provides sample images                                                            |
| draw              | Generates coordinates of geometric shapes                                         |
| exposure          | Examines and modifies image exposure levels                                       |
| external.tifffile | Handles reading, writing, and visualizing TIFF files <sup>62</sup>                |
| feature           | Feature detection and calculation                                                 |
| filters           | Contains various image filters and functions for calculating threshold values     |
| filters.rank      | Returns localized measurements in the image.                                      |
| graph             | Finds optimized paths across the image                                            |
| io                | Supports reading and writing images and contains a function for displaying images |
| measure           | Performs a variety of measurements and calculations on or between two images      |
| morphology        | Generates objects of a specified morphology                                       |
| novice            | Provides simple image functions for beginners                                     |
| restoration       | Includes image restoration tools                                                  |
| segmentation      | Identifies boundaries in an image                                                 |
| transform         | Performs image transformations including scaling and image rotation               |

---

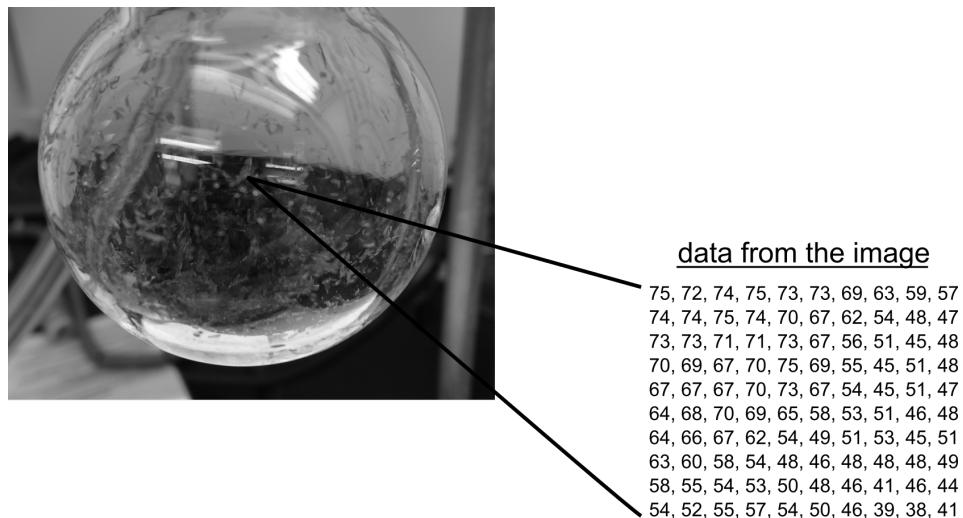
<sup>62</sup> TIFF files can be more complex than other images files because they are capable of holding multiple images and other metadata.

|        |                                                                                                                                                                    |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| util   | Converts images into different encodings (e.g., floats to integers) and other modifications such as inverting the image values and adding random noise to an image |
| viewer | Image viewer tools                                                                                                                                                 |

Despite the power and utility of the scikit-image library, there is a significant amount of image processing and analysis that can be performed using NumPy functionality. This is especially true being that scikit-image imports/stores images as NumPy arras.

## 7.1 Basic Image Structure

Most images are *raster* images, which are essentially a grid of pixels where each location on the grid is a number describing that pixel. If the image is a grayscale image, these values represent how light or dark each pixel is; and if it is a color image, the value(s) at each location describe the color. Figure 7.1 shows a grayscale photo of a flask containing crystals, with a  $10 \times 10$  pixel excerpt showing the brightness values from the photo.



**Figure 7.1** An excerpts of values from a grayscale image showing values representing the brightness of each pixel.

### 7.1.1 Loading Images

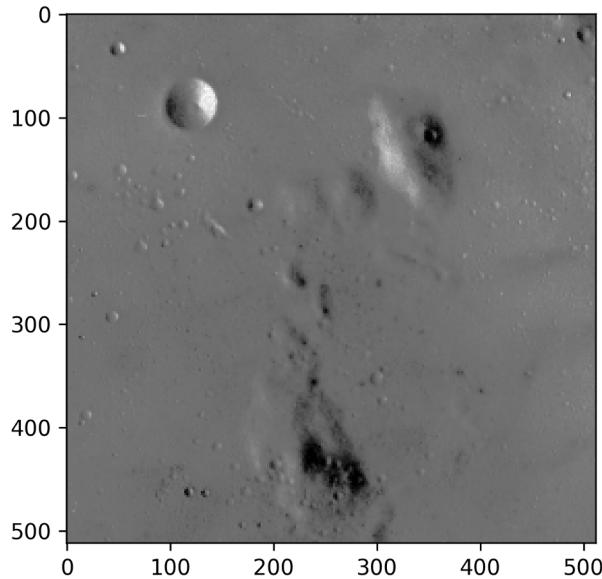
The scikit-image library includes a `data` module containing a series of images for the user to experiment with.<sup>63</sup> To display images in the notebook, use the `io.imshow()`

---

<sup>63</sup> It is a common practice for many specialized libraries is to include a `data` module with the library. These modules include example data for users of the library to experiment with and are usually a good place to start when learning a new library.

function from the `io` module. Each image in the `data` module has a function for fetching the image, and you can find a complete list of images/functions in the `data` module by typing `help(data)`. We will open and view the image of a lunar surface using the `data.moon()` function.

```
[in]: from skimage import data, io  
  
[in]: moon = data.moon()  
      io.imshow(moon)  
  
[out]:
```



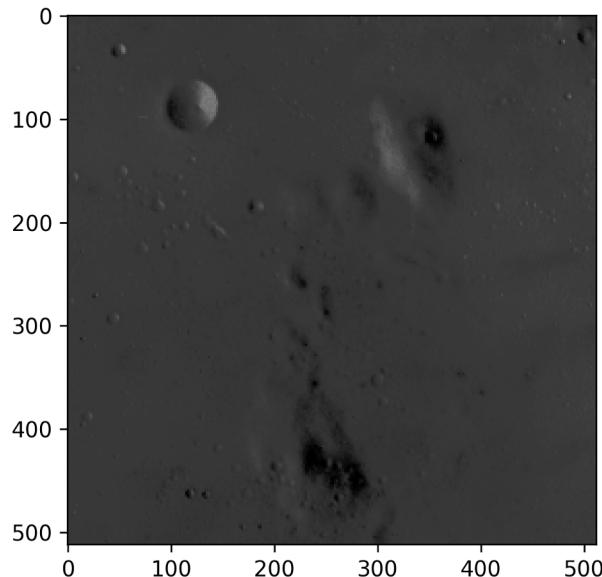
If we take a closer look at the data contained inside the lunar surface image, we find a two-dimensional NumPy array filled with integers ranging from  $0 \rightarrow 255$ .

```
[in]: moon  
  
[out]: array([[116, 116, 122, ..., 93, 96, 96],  
              [116, 116, 122, ..., 93, 96, 96],  
              [116, 116, 122, ..., 93, 96, 96],  
              ...,  
              [109, 109, 112, ..., 117, 116, 116],  
              [114, 114, 113, ..., 118, 118, 118],  
              [114, 114, 113, ..., 118, 118, 118]],  
             dtype=uint8)
```

Each of these values represents a lightness value where 0 is black, 255 is pure white, and all other values are various shades of gray. To manipulate the image, we can use NumPy methods being that scikit-image stores images as ndarrays. For example, the image can be darkened by dividing all the values by two. Because this array is designated to contain integers (`dtype = uint8`), integer division (`//`) is used to avoid floats.

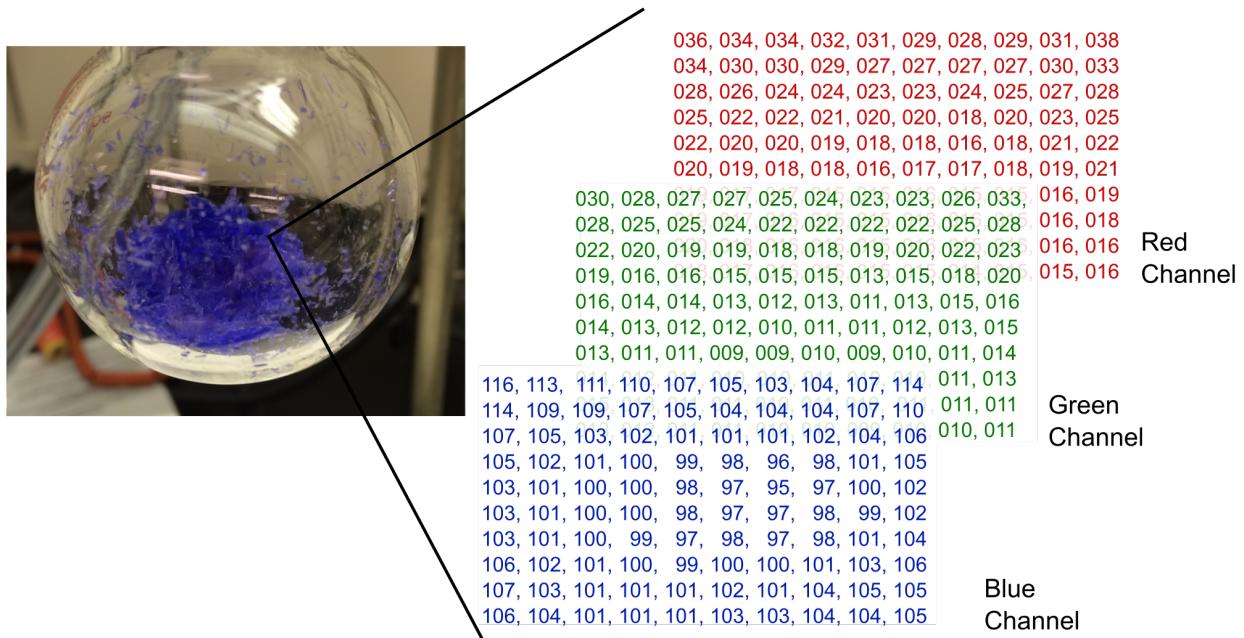
```
[in]: moon_dark = moon // 2
      io.imshow(moon_dark)
```

[out]:



### 7.1.2 Color Images

Color images are slightly more complicated to represent because all colors cannot be represented by single integers from  $0 \rightarrow 255$ . Probably the most popular way to digitally encode colors is **RGB** which describes every color as a combination of red, green, and blue (Figure 7.2). These are also known as **color channels**, and this is typically how computer monitors display colors. If you look close enough at the screen, which may require a magnifying glass for high resolution displays, you can see that every pixel is really made up of three lights: a red, a green, and a blue. You combine them to generate a net perceived color. Being that every pixel now has three values to describe it, a NumPy array that defines a color image is three dimensional. The first two dimensions are the height and width of the image, and the third dimension contains each of the three color channels.



**Figure 7.2** An excerpt of the red, green, and blue color channels for a small portion of a color image. The values in each channel represent the brightness of that color in each pixel.

By scikit-image convention, the encoding of color images in arrays is shown below, and the colors are in the order red, green, and then blue.

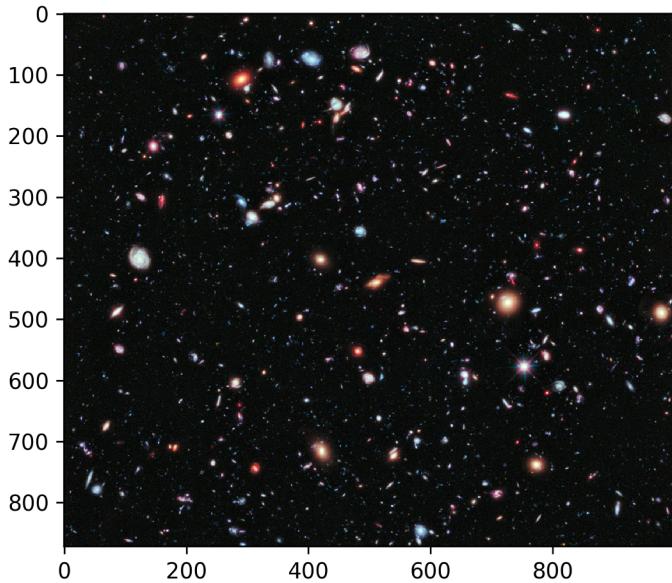
```
[row, column, channel]
```

This may be somewhat counter-intuitive knowing that three-dimensional NumPy arrays are encoded [depth, row, column].

We can look at an example of a color photo by loading an image from the Hubble space telescope.

```
[in]: hubble = data.hubble_deep_field()
      io.imshow(hubble)
```

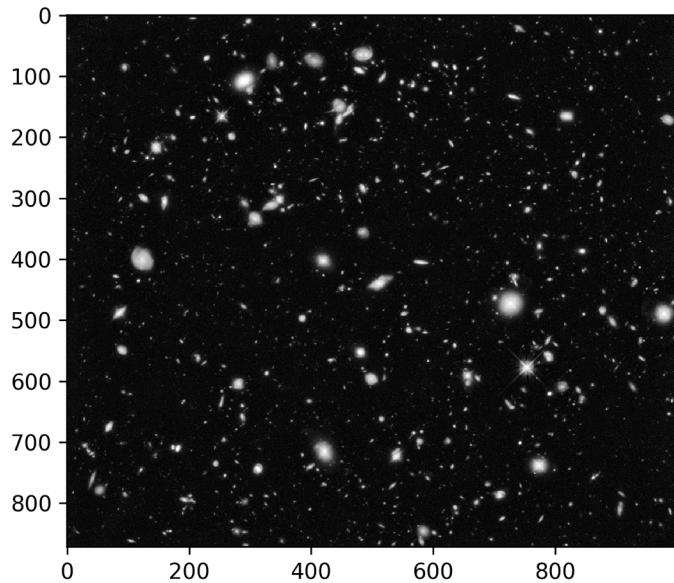
```
[out]:
```



```
[in]: hubble
[out]: array([[[15,  7,  4],
   [15,  9,  9],
   [ 9,  4,  8],
   ...,
   [11, 15, 14],
   [ 9, 18, 15],
   [ 7, 18, 12]]], dtype=uint8)
```

Looking at the array, you will notice that it is indeed three-dimensional with values residing in triplets. You may also notice that the numbers are rather small because most pixels in this particular image are near black. If we want to look at just the red values of the image, this can be accomplished by slicing the array. The red is the first layer in the third dimension, so we should slice it `hubble[:, :, 0]`. The brighter a group of pixels in the red channel image, the more red color that is present in that region.

```
[in]: io.imshow(hubble[:, :, 0])
[out]:
```



### 7.1.3 External Images

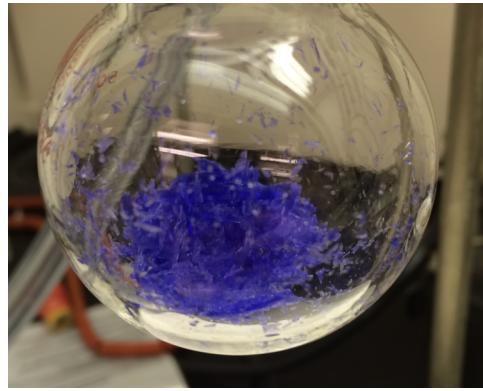
Alternatively, images can be loaded from an external source using the `io.imread()` function provided by scikit-image. This function requires one argument to tell scikit-image which image the user wants to load. If your Jupyter notebook is in the same directory as the image you want to load,<sup>64</sup> you can simply input the full file name, including the extension, as a string. Otherwise, you will need to include the full path to the file in addition to the name. Below an image showing a flask full of hexakis(acetonitrile)Ni(II) bis(tetrafluoroborate) crystals is read into Python.

```
[in]: flask = io.imread('flask.png')
       io.imshow(flask)

[out]:
```

---

<sup>64</sup> Or if you have already navigated to the directory of the image using the `os` module.



If we look at the array for the flask image below, you will notice that this is a three-dimensional array with four color channels. This can happen in some file types such as PNG where a fourth *alpha color channel* is supported making the coding RGBA. This channel measures opacity... that is, how non-transparent a pixel is. All of the pixels in this image are fully opaque which is represented by all 255. If the image was fully transparent, the third values would be all zeros, and anything in between would be translucent. Portable Network Graphics (PNG) images support an alpha channel as do many image formats, but JPG/JPEG images do not support this feature.

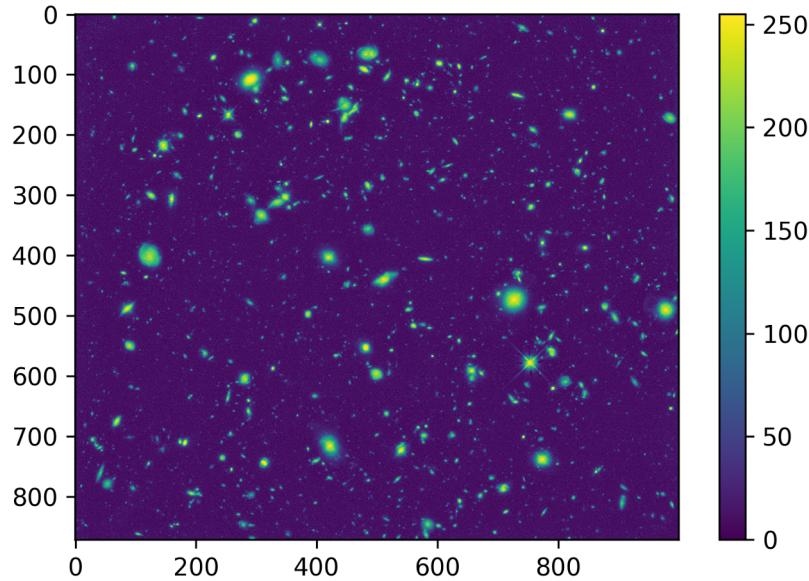
```
[in]: flask  
[out]:  
array([[[101,    85,    59,  255],  
       [101,    85,    59,  255],  
       [104,    87,    61,  255],  
       ...,  
       [ 18,    15,     8,  255],  
       [ 20,    17,    10,  255],  
       [ 22,    19,    12,  255]]], dtype=uint8)
```

### 7.1.4 Colormaps

So far, we having been displaying images using the scikit-image `io.imshow()` function which is intended specifically for images. Matplotlib also provides a `plt.imshow()` function that does roughly the same thing but with some important differences. Below is the red color channel from the `hubble` image displayed using the `plt.imshow()` function. The first thing you probably notice about the image is that the colors are likely not what you expected because when matplotlib deals with a NumPy array, it treats it as generic data, not an image. The human mind does not effectively handle data on this scale, so to make it easier for humans to interpret, `matplotlib` maps the values to

colors according the colormap on the right. By default, the colormap *viridis* is used,<sup>65</sup> but there are many others available to choose from in matplotlib.

```
[in]: import matplotlib.pyplot as plt  
plt.imshow(hubble[:, :, 0])  
plt.colorbar()  
  
[out]:
```

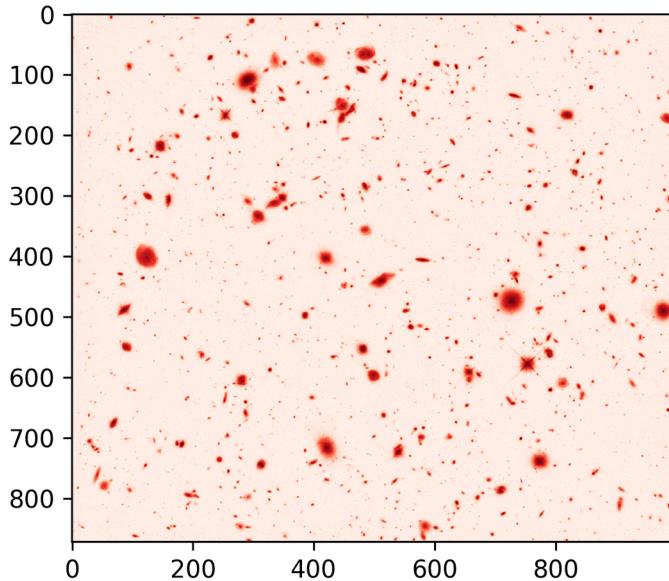


To change colormaps, input the name of a different colormap as a string in the optional `cmap` argument (e.g., `plt.imshow(hubble[:, :, 0], cmap='magma')`). See [https://matplotlib.org/examples/color/colormaps\\_reference.html](https://matplotlib.org/examples/color/colormaps_reference.html) for a list of available colormaps. It is strongly encouraged to use one of the *perceptually uniform* colormaps because they are more accurately interpreted by humans and also show up as a smooth, interpretable gradient when printed on a black-and-white printer. Below is the display of the Hubble image red channel using the `Reds` colormap.

```
[in]: plt.imshow(hubble[:, :, 0], cmap='Reds')  
  
[out]:
```

---

<sup>65</sup> For more information on this colormap, there is a 20 minutes talk by Nathaniel Smith and Stéfan van der Walt from the SciPy 2015 conference that explains the development of this colormap. It is a highly recommended watch at <https://www.youtube.com/watch?v=xAoljeRJ3lU>.



## 7.15 Saving Images

After processing an image, it is sometimes helpful to save the image to disk for records, reports, and presentations. Being that the images are ndarrays, the `plt.savefig()` function works just fine if executed in the same Jupyter cell as the `imshow()` function. Alternatively, scikit-image provides an image saving function `io.imsave(file_name, array)` that operates similarly as `plt.savefig()` except with a couple image-specific arguments. One key difference is that `plt.savefig()` does not take an array argument but instead assumes you want the recently displayed image saved while `io.imsave(file_name, array)` takes an array and can save an image even if it has not been displayed in the Jupyter notebook. Check the directory containing the Jupyter notebook, and there should be a new file titled `new_image.png`.

```
[in]: io.imsave('new_img.png', hubble)
```

## 7.2 Basic Image Manipulation

The scikit-image library along with NumPy also provide a variety of basic image manipulation functions such as adjusting the color, managing how the data is numerically represented, and establishing threshold cutoff values.

## 7.2.1 Colors

There are numerous ways to represent colors in digital data. The RGB color space is undoubtably one of the most popular color spaces, but there are others that you may encounter such as HSV (hue, saturation, value) or XYZ. Scikit-image provides functions in the `color` module for easily converting between these color spaces, and Table 7.2 lists some common functions. See the scikit-image website for a more complete list

**Table 7.2** Common Functions from the `color` Module

| Function              | Description                                                                              |
|-----------------------|------------------------------------------------------------------------------------------|
| <code>rgb2gray</code> | Converts from RGB to grayscale                                                           |
| <code>gray2rgb</code> | Converts grayscale to RGB; by just replicating the gray values into three color channels |
| <code>hsv2rgb</code>  | HSV to RGB conversion                                                                    |
| <code>xyz2rgb</code>  | XYZ to RGB conversion                                                                    |

```
[in]: hubble_gray = color.rgb2gray(hubble)
       hubble_gray

[out]:
array([[ 0.03326941,  0.04029412,  0.02098392, ...,
         0.04727412,
         0.0694651 ,  0.04225137],
       [ 0.0213051 ,  0.03700627,  0.06894431, ...,
         0.03863529,
         0.06444235,  0.04005686],
       [ 0.05296039,  0.06529059,  0.08322392, ...,
         0.00644431,
         0.05657647,  0.04877098],
       ...,
       [ 0.07590157,  0.05825804,  0.05098039, ...,
         0.01991333,
         0.05295255,  0.03334471],
       [ 0.04030196,  0.04062235,  0.06502275, ...,
         0.03167804,
         0.04149333,  0.04484941],
       [ 0.06578078,  0.04454392,  0.06813373, ...,
         0.05520745,  0.06224   ,  0.0597251 ]])
```

You will notice that scikit-image takes a three-dimensional data structure, the third dimension being the color channels, and converted it to a two-dimensional, grayscale structure as expected. One detail that may strike you as different is that the values are decimals. Up to this point, grayscale images were represented as two-dimensional arrays of integers from 0 → 255. There is no rule that says lightness and darkness values need to be represented as integers. Above, they are presented as floats from 0 → 1. This brings us to the next topic of encoding values.

### 7.2.2 Encoding

*Encoding* is how the values are presented in the image array. The two most common are integers from 0 → 255 or floats from 0 → 1. However, there are other ranges outlined in Table 7.3. Though you may never encounter some, it is good to be aware that they exist in case you need to deal with them. The difference between signed integers (`int`) and unsigned integers (`uint`) is that unsigned integers are only positive integers starting with zero while signed integers are both positive and negative centered approximately around zero.<sup>66</sup> To determine what the range of values is for an image, scikit-image provides the function `skimage.dtype.limits()`.

Scikit-image also provides some convenient functions for converting to various value ranges described in Table 7.3. These functions are not contained in a module, so you will need to just do an `import skimage` to get access to them. The one format that probably needs commenting on is the Boolean format. In this encoding, every pixel is a `True` or `False` value, which is equivalent to saying 1 or 0. This is for black-and-white images where each pixel is one of two possible values.

**Table 7.3** Scikit-image Functions for Converting Data Types

| Functions                                                   | Description                                                                 |
|-------------------------------------------------------------|-----------------------------------------------------------------------------|
| <code>img_as_ubyte</code>                                   | Converts to integers from 0 → 255                                           |
| <code>img_as_uint</code>                                    | Converts to integers from 0 → 65535                                         |
| <code>img_as_int</code>                                     | Converts to integers from -32768 → 32767                                    |
| <code>img_as_bool</code>                                    | Converts to boolean (i.e., <code>True</code> or <code>False</code> ) format |
| <code>img_as_float32</code>                                 | Converts to floats from 0 → 1 with 32-bit precision                         |
| <code>img_as_float64</code><br>or <code>img_as_float</code> | Converts to floats from 0 → 1 with 64-bit precision                         |

---

<sup>66</sup> There are equal numbers of positive and negative integers, and being that zero is a positive integer, zero is not the exact center.

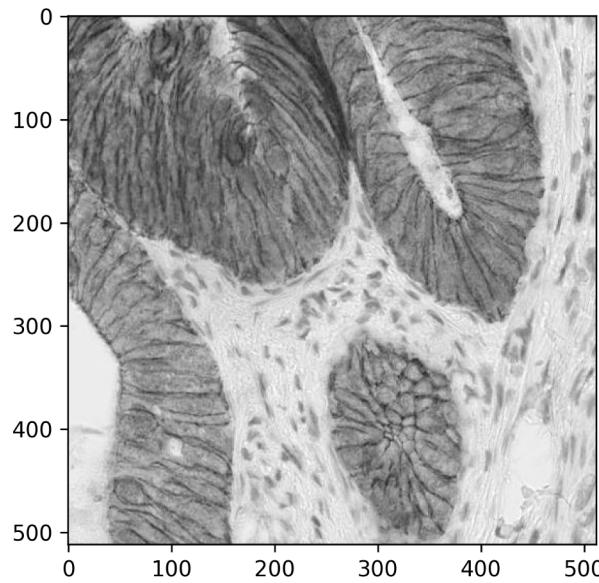
```
[in]: import skimage  
       skimage.dtype.limits(hubble_gray)  
[out]: (0, 1)  
  
[in]: hubble_gray_uint8 = skimage.img_as_ubyte(hubble_gray)  
       skimage.dtype.limits(hubble_gray_uint8)  
[out]: (0, 255)
```

### 7.2.2 Image Contrast

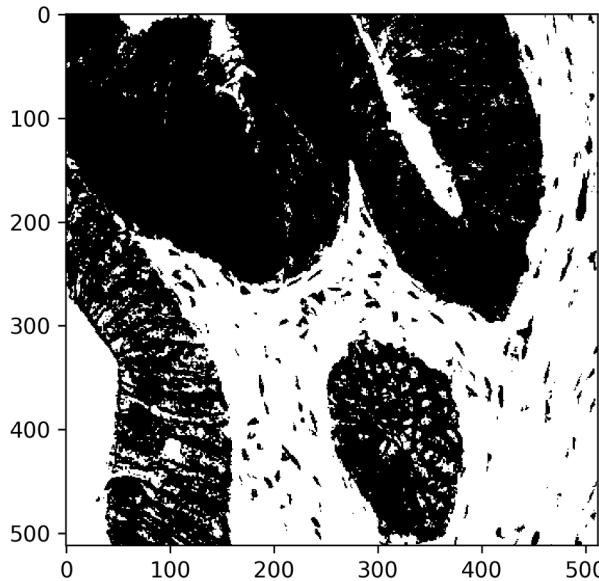
Before trying to extract certain types of information or identify features in an image, it is sometimes helpful to first increase the contrast of an image. There are a number of ways of doing this including thresholding and modification of the image histogram. Some approaches can be performed using NumPy array manipulation, but scikit-image also provides convenient functions designed for these task.

*Thresholding* can be used to generate a black-and-white image (not grayscale) by converting gray values at or below a brightness threshold to black and above the threshold to white. The threshold can be set manually or by an algorithm that chooses an optimal value customized to each image. We will start with manually setting a threshold. The grayscale image generated from `rgb2gray()` is encoded with floats from  $0 \rightarrow 1$ , so a threshold of 0.65 is chosen using trial-and-error. A black-and-white image is then generated as a Boolean. The resulting black-and-white image is shown below.

```
[in]: chem = data.immunohistochemistry()  
       chem_gray = color.rgb2gray(chem)  
       io.imshow(chem_gray)  
  
[out]:
```



```
[in]: chem_bw = chem_gray > 0.65 # generates a boolean  
encoding  
io.imshow(chem_bw)  
[out]:
```



The appropriate threshold may vary from image to image, so manually setting a value is sometimes not practical. Scikit-image provides a number of functions in the `filters` module for automatically choosing a threshold shown below in Table 7.4. If you are not sure

which of the functions below to use, there is a `try_all_filters()` function in the `filters` module that will try seven of them and plot the results for easy comparison.

**Table 7.4** Threshold Functions from the `filters` Module

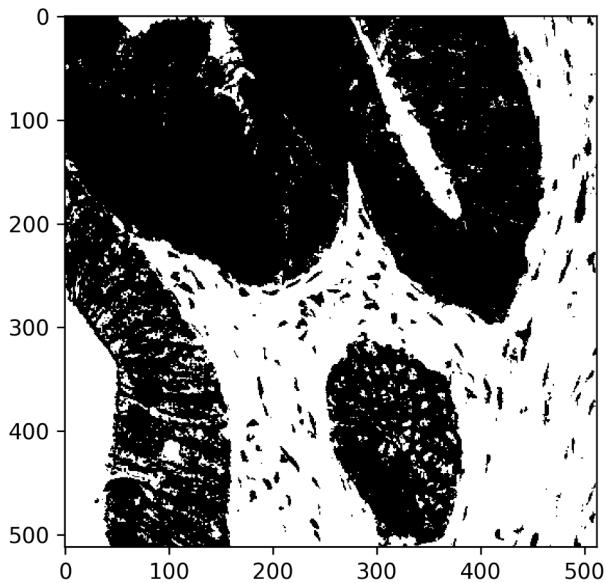
| Functions                       | Description*                                           |
|---------------------------------|--------------------------------------------------------|
| <code>threshold_isodata</code>  | Threshold value from ISODATA method                    |
| <code>threshold_li</code>       | Threshold value from Li's minimum cross entropy method |
| <code>threshold_local</code>    | Threshold mask (array) from local neighborhoods.       |
| <code>threshold_mean</code>     | Threshold value from mean grayscale value              |
| <code>threshold_minimum</code>  | Threshold value from minimum method                    |
| <code>threshold_niblack</code>  | Threshold mask (array) from the Niblack method         |
| <code>threshold_otsu</code>     | Threshold value from Otsu's method                     |
| <code>threshold_sauvola</code>  | Threshold mask (array) from Sauvola method             |
| <code>threshold_triangle</code> | Threshold value from triangle method                   |
| <code>threshold_yen</code>      | Threshold value from Yen method                        |

\*Threshold value functions provides a single threshold value while threshold masks provides arrays of values the size of the image. They are used in the same fashion except that the latter provides a per-pixel threshold.

Below, the Otsu filter is demonstrated.

```
[in]: from skimage import filters
       threshold = filters.threshold_otsu(chem_gray)
       chem_otsu = chem_gray > threshold
       io.imshow(chem_otsu)

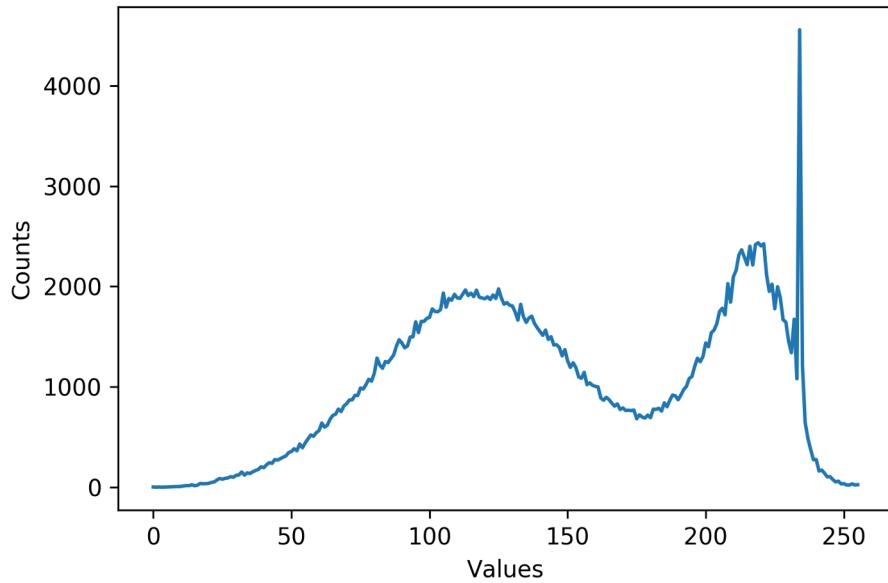
[out]:
```



Another method for increasing contrast is by modifying the image histogram. If the values from an image are plotted in a histogram, you will see something that looks like the following.

```
[in]: from skimage import exposure  
hist = exposure.histogram(chem_gray) # must be  
grayscale image  
plt.plot(hist[0])  
plt.xlabel('Values')  
plt.ylabel('Counts')
```

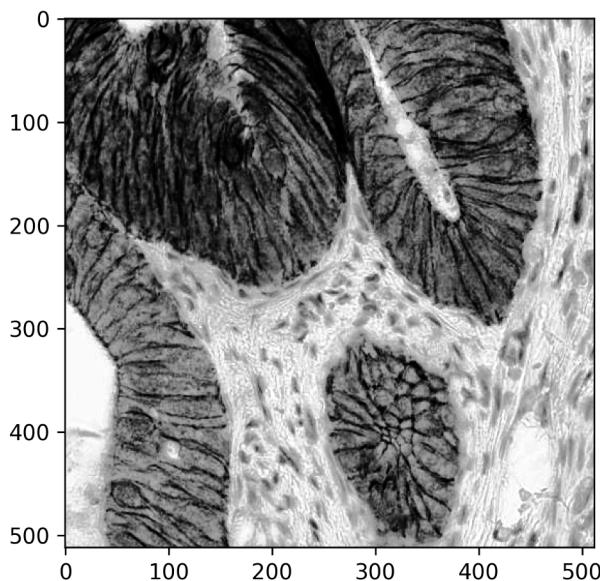
```
[out]:
```



This is a plot of how many of each type of brightness value is present in the image. There are practically no pixels in the image that are black (value 0) or completely white (value 255), but there are two main collections of gray values. The contrast of this image can be increased by performing histogram equalization, which spreads these values out more evenly. The `exposure` module provides an `equalize_hist()` function for this task.

```
[in]: chem_eq = exposure.equalize_hist(chem_gray)
       io.imshow(chem_eq)
```

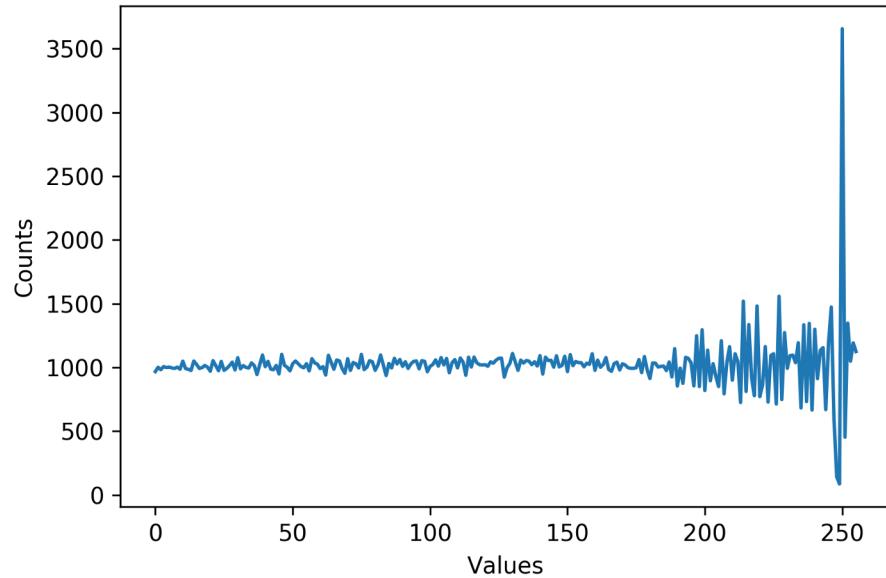
[out] :



Histogram equalization does not produce a black-and-white image, but it does make the dark values darker and the light values lighter. If we look at the histogram for this image, it will be more even as shown below.

```
[in]: hist = exposure.histogram(chem_eq)
plt.plot(hist[0])
plt.xlabel('Values')
plt.ylabel('Counts')

[out]:
```



## 7.3 Scikit-Image Examples

The scikit-image library contains numerous functions for performing various scientific analyses... so many that they cannot be comprehensively covered here. Below is a selection of some interesting examples that are relevant to science including counting objects in images, entropy analysis, and measuring eccentricity of objects. The examples below use mostly synthetic data to represent various data you might encounter in the lab. Real data can be easily extracted from publications but are not used here for copyright reasons.

### 7.3.1 Blob Detection

A classic problem that translates across many scientific fields is to count spots in a photograph. A biologist may need to quantifying the number of bacteria colonies in a petrie dish over the course of an experiment, while an astronomer may want to count the number of stars in a large cluster. In chemistry, this problem may occur as a need to quantify the number

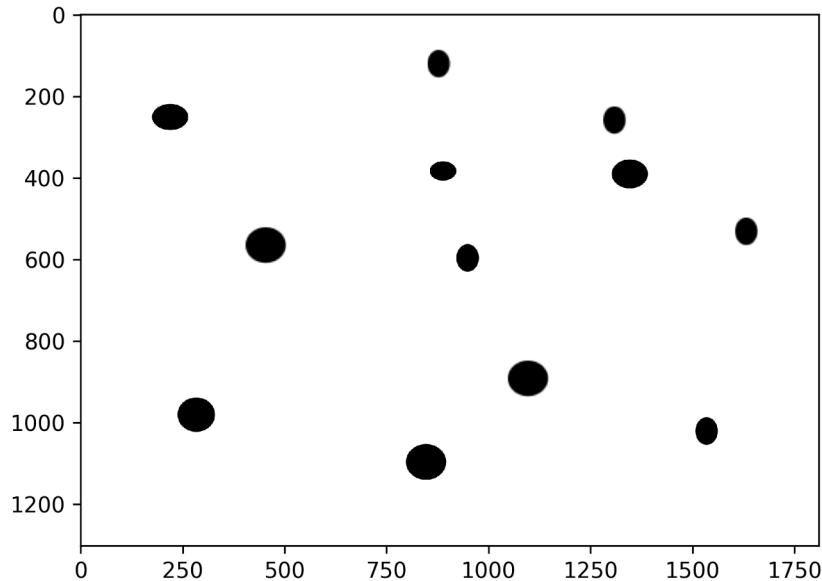
of nanoparticles in a photograph or using the locations to calculate the average distances between the particles.

The good news is that the scikit-image library provides three functions that will take a photograph and return an array of  $x, y, z$  coordinates indicating where the blobs are located in the image. If all you care about is the number of blobs, simply find the length of the returned array. There are three functions listed below which include Laplacian of Gaussian (LoG), Difference of Gaussian (DoG), and Determinant of Hessian (DoH). The LoG algorithm is the most accurate but the slowest while the DoH algorithm is the fastest.<sup>67</sup> These functions only accept two-dimensional images, so if it is a color image, you will need to either convert it to grayscale or select a single color channel to work with.

```
skimage.feature.blob_log(image, threshold=)  
skimage.feature.blob_dog(image, threshold=)  
skimage.feature.blob_doh(image, threshold=)
```

```
[in]: dots = io.imread('dots.png')  
io.imshow(dots)
```

```
[out] :
```



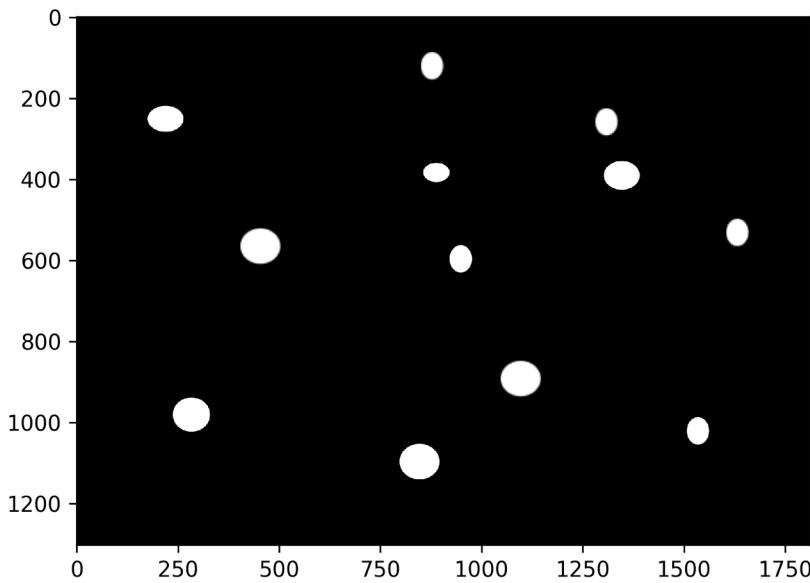
---

<sup>67</sup> Scikit-Image Blob Detection. [http://scikit-image.org/docs/dev/auto\\_examples/features\\_detection/plot\\_blob.html](http://scikit-image.org/docs/dev/auto_examples/features_detection/plot_blob.html).

An image of black dots on a white background is imported above, but the blob detection algorithms work best with light colors on a dark background. We will invert the image below by subtracting the values from the maximum value.<sup>68</sup>

```
[in]: dots_invert = color.rgb2gray(255 - dots)
      io.imshow(dots_invert)
```

[out]:



To detect the blobs, we will use the `blob_dog` function as demonstrated below. The function allows for a `threshold` argument to be set to adjust the sensitivity of the algorithm in finding blobs. A lower threshold results in smaller or less intense blobs included in the returned array.

```
[in]: from skimage import feature
```

```
blobs = feature.blob_dog(dots_inverted, threshold=0.5)
blobs
```

[out]:

```
array([[ 1096.          ,   847.          ,   42.94967296],
       [ 1021.          ,  1534.          ,   26.8435456 ],
       [  980.          ,   283.          ,   42.94967296],
       [  892.          ,  1097.          ,   42.94967296],
```

---

<sup>68</sup> Inverting the image is accomplished here by subtracting the image from the maximum *possible* value. This can also be accomplished using the `invert()` function provided in the `util` module of scikit-image.

```
[ 596. , 949. , 26.8435456 ],
[ 565. , 453. , 42.94967296],
[ 531. , 1632. , 26.8435456 ],
[ 391. , 1346. , 26.8435456 ],
[ 383. , 888. , 26.8435456 ],
[ 258. , 1308. , 26.8435456 ],
[ 251. , 219. , 26.8435456 ],
[ 120. , 877. , 26.8435456 ]])
```

The returned array includes three columns corresponding to the  $y$  position,  $x$  position, and intensity of each spot, respectively. The  $x$  and  $y$  coordinates for an image starts at the top left corner while typical plots start at the bottom left. Keep this in mind when comparing the coordinates to the image. To find the number of spots, determine length of the array.

```
[in]: len(blobs)
[out]: 12
```

### 7.3.2 Entropy Analysis

The term *entropy* outside of the physical sciences is used to simply represent a quantification of disorder or irregularity. In image analysis, this disorder is the amount of brightness variation within a region of the image. As you will see below, entropy is the highest near the boundaries and in noisy areas of the photograph. This makes an entropy analysis useful for edge detection, checking for image quality, and detecting alterations to an image.

The `filters.rank` module contains the entropy function shown below. It works by going through the image pixel-by-pixel and calculating the entropy in the neighborhood. The *neighborhood* is the area around each pixel. An entropy value is recorded in the new array at each location and can be plotted to generate an entropy map. The entropy function takes two required arguments: the image (`img`) and a description of the neighborhood called a *structured element* (`selem`).

```
filters.rank.entropy(img, selem)

[in]: from skimage.morphology import disk
      from skimage.filters.rank import entropy
      selem = disk(5)
      selem

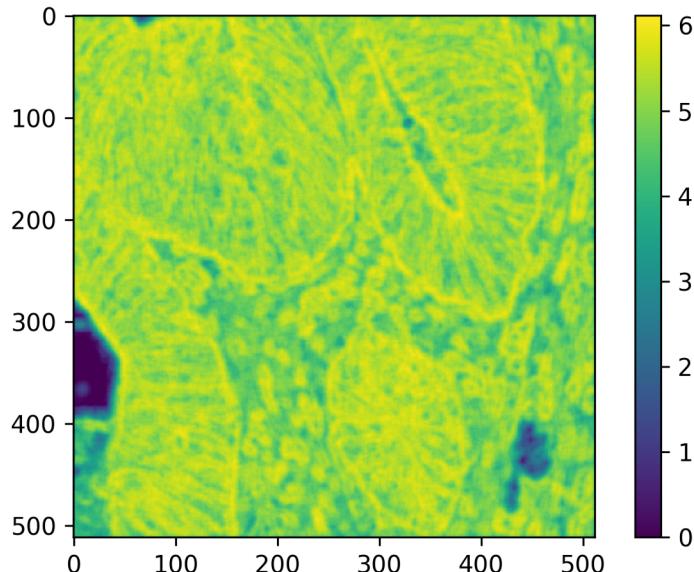
[out]:
```

```
array([[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],  
       [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],  
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0],  
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0],  
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0],  
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],  
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0],  
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0],  
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0],  
       [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0],  
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

The neighborhood is defined as an array of ones and zeros. In this case, it is a disk of radius 5. The user can adjust this value to the needs of the analysis.

```
[in]: S = entropy(chem_gray, selem)  
plt.imshow(S)  
plt.colorbar()
```

[out]:



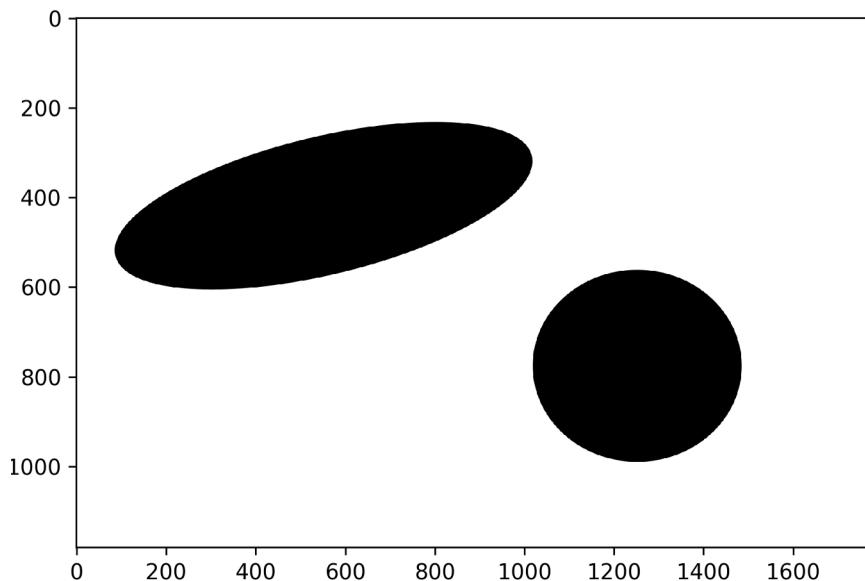
Examination of the image shows that there is an increase in entropy near the edges of the features in the image as expected. There are two regions (blue) that contain unusually low entropy. If you look back at the original image, these regions are comparatively homogeneous in color.

### 7.3.3 Eccentricity

*Eccentricity* is the measurement of how non-circular an object is. It runs from  $0 \rightarrow 1$  with zero being a perfect circle and large values representing more eccentric objects. This can be useful for quantifying the shape of nanoparticles or droplets of liquid. The `measure` module from scikit-image provides an easy method of measuring eccentricity. First, let us first import an image of ovals for an example. Alternatively, you are welcome to use the `coins` image from the `data` module, but this will require some preprocessing such as increasing the contrast.

```
[in]: ovals = io.imread('oval.png')
       io.imshow(ovals)
```

[out] :



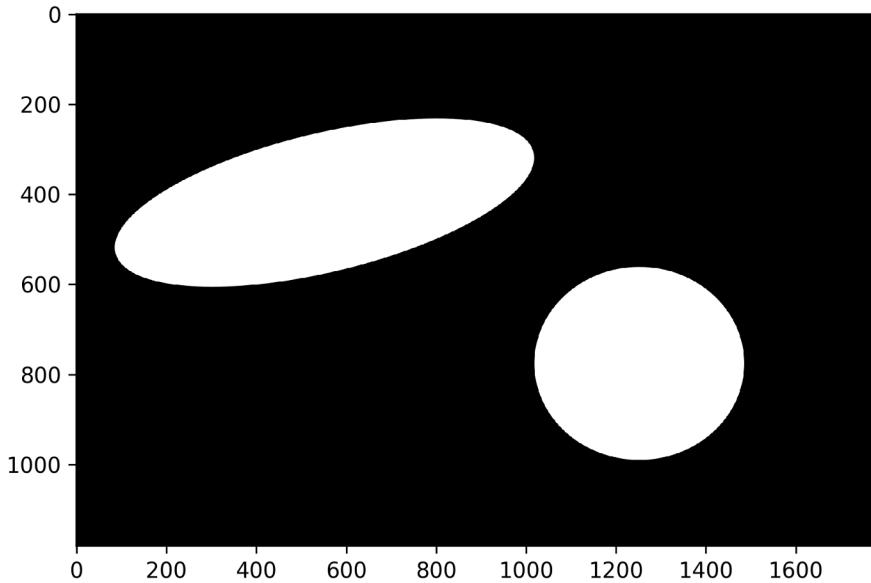
The main function for measuring eccentricity is the `regionsprops` function, but this function by itself cannot find the objects. Luckily, there is another function in the `measure` module called `label` that will do exactly this, and this function requires the regions to be light with dark backgrounds.<sup>69</sup>

```
[in]: ovals_invert = color.rgb2gray(255 - ovals)
       io.imshow(ovals_invert)
```

[out] :

---

<sup>69</sup> There are various functions in the scikit-image library that prefer light objects with black backgrounds. The colors can always be flipped if needed.



```
[in]: from skimage.measure import label, regionprops  
       lbl = label(ovals_invert)  
       props = regionprops(lbl)
```

The `regionprops` function returns the properties on the two ovals in a list of lists. The first list corresponds to the first object and so on. Each list contains an extensive collection of properties, so it is worth visiting the scikit-image website to see the complete documentation. We are only concerned with eccentricity right now, so we can access the eccentricity of the first object with `props[0].eccentricity`, which gives a value of about 0.95 while the second object has a much lower values of about 0.40. This makes sense being that the first object is very eccentric while the second object is much more circular.

```
[in]: props[0].eccentricity  
[out]: 0.9469273936534153  
  
[in]: props[1].eccentricity  
[out]: 0.39666071911268475
```

## Further Reading

The scikit-image library with NumPy are likely all you will need for a vast majority of your scientific image processing, and the scikit-image project webpage is an excellent course of information and examples. The gallery page is particularly worth checking out as it provides a large number of examples highlighting the library's capabilities. In the event there is an edge case the scikit-image cannot do, the pillow library may be of some use. Pillow provides more fundamental image processing functionality such as extracting metadata from the original file.

1. Scikit-image Website. <http://scikit-image.org/>.
2. Pillow Documentation Page. <https://pillow.readthedocs.io/en/4.2.x/>.

## Exercises

Solve the following problems using Python in a Jupyter notebook.

### ***Basic Image Operations***

1. Import the image titled `NaK_THF.jpg` using scikit-image.
  - a) Convert the image to grayscale using a scikit-image function.
  - b) Save the grayscale image using the `io` module.
2. Load the `chelsea` image from the scikit-image `data` module and convert it to grayscale. Display the image using the scikit-image plotting function and display it a second time using a matplotlib plotting function. Why do they look different?
3. Save a  $1000 \times 1000$  pixel image containing random noise generated by the NumPy `np.random.randn()` function. Display the image in a Jupyter notebook along with a histogram of the pixel values.
4. Write your own Python function for converting a color image to grayscale. Then find the source code for the scikit-image `rgb2gray()` function available on the scikit-image website and compare it to your own function. Are there any major differences between your function and the scikit-image function?
5. Import an image of your choice either from the `data` module or of your own and convert it to a grayscale image.
  - a) Invert the grayscale image using NumPy by subtracting all values from the maximum possible value

- b) Invert the original grayscale image using the `invert()` function in the scikit-image module
6. Import a *color image* of your choice either from the `data` module or of your own and calculate the sum of all pixels from each of the three color channels (RGB). Which color (red, green, or blue) is most prevalent in your image?

## ***Image Analysis***

7. The folder titled `glow_stick` contains a series of images taken of a glow stick over the course of approximately thirteen hours along with a CSV file containing the times at which each image was taken in numerical order. Quantify the brightness of each image and generate a plot of brightness versus time.
8. The JPG image file format commonly used for photographs degrades images during the saving process due to the *lossy compression* algorithm while the PNG image file format does not degrade images with its *lossless compression* algorithm.
- To view how JPG distorts images, import the `nmr.png` and `nmr.jpg` images of the same NMR spectrum. Subtract the two images from each other and visualize this difference to see the image distortions caused by JPG compression.
  - Which of the above file formats is better for image-based data in terms of data integrity?
9. Import the image `spots.png` and determine the number of spots in the image using scikit-image. Plot the coordinates of the spots you find with red x's over the image to confirm your results. If your script missed any spots, speculate as to why those spots were missed.
10. The image `test_tube_altered.png` has been altered using photo editing software. Generate and plot an entropy map of the image to identify the altered regions.
11. *Steganography* is the practice of hiding information in an image or digital file to avoid detection. The file `hidden_image.png` was created by combining an image with pseudorandom noise to mask the original image. Perform an entropy analysis on the image to reveal the original image. You may need to adjust the size of the selection element (`selem`) to detect the hidden image.



# *Chapter 8*

## ***Mathematics***

We have already been doing math throughout this book as Python is fundamentally performing mathematical operations through arithmetic, calculus, algebra, and Boolean logic among others, but this chapter will dive deeper into symbolic mathematics, matrix operations, and integration. Some of this chapter will rely on SciPy and NumPy, but for the symbolic mathematics, we will use the popular SymPy<sup>70</sup> library.

SymPy is the main library in the SciPy ecosystem for performing symbolic mathematics, and it is suitable for a wide audience from high school students to scientific researchers. It is something like a free, open source Mathematica substitute that is built on Python and is arguably more accessible in terms of cost and ease of acquisition. All of the following SymPy code relies on the following import which makes all of the SymPy modules available.

```
[in]: import sympy
```

### **8.1 Symbolic Mathematics**

SymPy differentiates itself from the rest of Python and SciPy stack in that it returns exact or symbolic results whereas Python, SciPy, and NumPy will generate numerical answers which may not be exact. That is to say, not only does SymPy perform symbolic mathematical operations, but even if the result of an operation has a numerical answer, SymPy will return the value in exact form. For example, if we take the square root of 2 using the `math` module, we get a numerical value.

---

<sup>70</sup> SymPy Website. <http://www.sympy.org/en/index.html>.

```
[in]: import math  
       math.sqrt(2)  
  
[out]: 1.4142135623730951
```

The value returned is a rounded approximation of the true answer. In contrast, if the same operation is performed using SymPy, we get a different result.

```
[in]: sympy.sqrt(2)  
  
[out]: sqrt(2)
```

Because the square root of two is an irrational number, it cannot be represented exactly by a decimal, so SymPy leaves it in the exact form of `sqrt(2)`. If we absolutely need a numerical value, SymPy can be instructed to evaluate an imprecise, numerical value using the `evalf()` method.

```
[in]: sympy.sqrt(2).evalf()  
  
[out]: 1.41421356237310
```

One of the advantages of `evalf()` is that it also accepts a `significant figures` argument.

```
[in]: sympy.sqrt(2).evalf(30)  
  
[out]: 1.41421356237309504880168872421  
  
[in]: sympy.pi.evalf(40)  
  
[out]: 3.141592653589793238462643383279502884197
```

### 8.1.1 Symbols

Before SymPy will accept a variable as a symbol, the variable must first be defined as a SymPy symbol using the `symbols()` function. It takes one or more symbols at a time and attaches them to variables.

```
[in]: x, c, m = sympy.symbols('x c m')  
  
[in]: x  
  
[out]: x
```

There is no value attached to `x` as it is a symbol, so now it can be used to generate symbolic mathematical expressions.

```
[in]: E = m*c**2  
  
[in]: E
```

```
[out]: m*C**2  
[in]: E**2  
[out]: c**4*m**2
```

### 8.1.2 Pretty Printing

At this point, you might be thinking that the output of SymPy is not the easiest to read or at the very least, not very aesthetically appealing. As an option, you can turn on *pretty printing*, which will instruct SymPy to output the expressions in more traditional mathematical representations.

```
[in]: from sympy import init_printing  
sympy.init_printing()  
  
[in]: E**2  
[out]:  $c^4m^2$   
  
[in]: sympy.sqrt(x)  
[out]:  $\sqrt{x}$ 
```

While you are free to use this output mode, the remainder of this chapter will have pretty printing turned off.

### 8.1.3 SymPy Mathematical Functions

Similar to the `math` Python module, SymPy contains an assortment of standard mathematical operators such as square root and trigonometric functions. A table of common functions is below. Some of the functions start with a capital letter such as `Abs()`. This is important so that they do not collide with native Python functions if SymPy is imported into the global namespace.<sup>71</sup>

**Table 8.1** Common SymPy Functions

|                        |                      |                     |                     |                     |
|------------------------|----------------------|---------------------|---------------------|---------------------|
| <code>Abs()</code>     | <code>sin()</code>   | <code>cos()</code>  | <code>tan()</code>  | <code>cot()</code>  |
| <code>sec()</code>     | <code>csc()</code>   | <code>asin()</code> | <code>acos()</code> | <code>atan()</code> |
| <code>ceiling()</code> | <code>floor()</code> | <code>Min()</code>  | <code>Max()</code>  | <code>sqrt()</code> |

---

<sup>71</sup> That is, if SymPy is imported with `from sympy import *`... which is discouraged as it can cause unexpected behavior if variables collide.

It is important to note that any mathematical function operating on a symbol needs to be from the SymPy library. For example, using a cosine function from the `math` library will result in an error.

```
[in]: math.cos(x)  
[out]: TypeError: can't convert expression to float  
  
[in]: sympy.cos(x)  
[out]: cos(x)
```

## 8.2 Algebra in SymPy

SymPy is quite capable at algebraic operations and is knowledgeable of common identities such as  $\sin(x)^2 + \cos(x)^2 = 1$ , but before we proceed with doing algebra in SymPy, we need to cover some basic algebraic methods. These are provided in table 8.2 which includes polynomial expansion and factoring, expression simplification, and solving equations. The subsequent sections demonstrate each of these.

**Table 8.2** Common Algebraic Methods

| Method                | Description                                                          |
|-----------------------|----------------------------------------------------------------------|
| <code>expand</code>   | Expand polynomials                                                   |
| <code>factor</code>   | Factors polynomials                                                  |
| <code>simplify</code> | Simplifies the expression                                            |
| <code>solve</code>    | Equates the expression to zero and solves for the requested variable |

### 8.2.1 Polynomial Expansion and Factoring

When dealing with polynomials, expansion and factoring are common operations that can be tedious and time-consuming by hand. SymPy makes these quick and easy. For example, we can expand the expression  $(x - 1)(3x + 2)$  as demonstrated below.

```
[in]: expr = (x - 1)*(3*x + 2)  
[in]: sympy.expand(expr)  
[out]: 3*x**2 - x - 2
```

The process can be reversed by factoring the polynomial.

```
[in]: sympy.factor(3*x**2 - x - 2)
[out]: (x - 1)*(3*x + 2)
```

## 8.2.2 Simplification

SymPy may not always return a mathematical expression in the simplest form. Below is an expression with a simpler form, and if we feed this to SymPy, it is not automatically simplified.

```
[in]: 3*x**2 - 4*x - 15 / (x - 3)
[out]: 3*x**2 - 4*x - 15 / (x - 3)
```

However, if we instruct SymPy to simplify the expression using the `simplify()` method, it will make a best attempt at finding a simpler form.

```
[in]: sympy.simplify((3*x**2 - 4*x - 15) / (x - 3))
[out]: 3*x + 5
```

## 8.2.3 Solving Equations

SymPy can also solve equations for an unknown variable using the `solve()` function. The function requires a single expression that is equal to zero. For example, the following solves for  $x$  in  $x^2 + 1.4x - 5.76 = 0$ .

```
[in]: sympy.solve(x**2 + 1.4*x - 5.76)
[out]: [-3.20000000000000, 1.80000000000000]
```

The two mathematically valid solutions for  $x$  are returned as a Python list. Just because the math allows for two values, however, does not mean that both solutions are chemically valid. For example, if we are solving for concentration, a negative concentration does not make sense leaving 1.8 as the only chemically valid answer.

## 8.2.4 Equilibrium ICE Table

A common chemical application of the above algebraic operations is solving equilibrium problems using the ICE (Initial, Change, and Equilibrium) method. As a penultimate step, the mathematical expressions are inserted into the equilibrium expression and often result in a polynomial equation. Below is an example problem with completed ICE table and equilibrium expression.

|                                    |                         |                      |                            |   |                          |
|------------------------------------|-------------------------|----------------------|----------------------------|---|--------------------------|
|                                    | <b>2 NH<sub>3</sub></b> | $\rightleftharpoons$ | <b>3 H<sub>2</sub> (g)</b> | + | <b>N<sub>2</sub> (g)</b> |
| <b>Initial</b>                     | 0.60 M                  |                      | 0.60 M                     |   | 0.80 M                   |
| <b>Change, <math>\Delta</math></b> | -2x                     |                      | +3x                        |   | +x                       |
| <b>Equilibrium</b>                 | 0.60 - 2x               |                      | 0.60 + 3x                  |   | 0.80 + x                 |

$$K_c = 3.44 = \frac{[N_2][H_2]^3}{[NH_3]^2} = \frac{(0.80+x)(0.60+3x)^3}{(0.60-2x)^2}$$

To expand the right portion of the equation, we can use the `expand()` method.

```
[in]: expr = (0.80 + x) * (0.60 + 3*x)**3 / (0.60 - 2*x)**2
[in]: sympy.expand(expr)
[out]: 27*x**4/(4*x**2 - 2.4*x + 0.36) + 37.8*x**3/(4*x**2 -
2.4*x + 0.36) + 16.2*x**2/(4*x**2 - 2.4*x + 0.36) +
2.808*x/(4*x**2 - 2.4*x + 0.36) + 0.1728/(4*x**2 -
2.4*x + 0.36)
```

This is probably not what you were expecting or hoping for. The polynomial has been expanded for the numerator and denominator, but the result is still a fraction. We can instruct SymPy to simplify the results.

```
[in]: sympy.simplify(sympy.expand(expr))
[out]: (27*x**4 + 37.8*x**3 + 16.2*x**2 + 2.808*x +
0.1728)/(4*x**2 - 2.4*x + 0.36)
```

This is much better. Ultimately, we want to solve for  $x$ , but the `solve()` function requires an expression that equals zero. We can achieve this by subtracting 3.44.

```
[in]: sympy.solve(expr - 3.44)
[out]: [-1.52752978294276,
 0.0916567216237442,
 0.0179365306595075 - 0.530628104891225*I,
 0.0179365306595075 + 0.530628104891225*I]
```

The fourth-order polynomial returns four solutions, but only one will make physical sense for our application. The second value, 0.09166, is the only non-imaginary value that does not generate negative concentrations, so this is the solution for  $x$ .

## 8.3 Matrices

Matrices are an efficient method of working with larger amounts of data. When done by hand, as is the case in many classroom environments, it is likely slow and painful. The beauty and power of matrices is when they are used with computers because they simplify bulk calculations. SymPy, SciPy, and NumPy all support matrix operations. If you need to do symbolic math, SymPy should be your go-to, but for the numerical calculations that we will do here, we will use NumPy's `linalg` module.

SciPy and NumPy both offer a matrix object, but the SciPy official documentation discourages their use as they offer little advantage over a standard NumPy array.<sup>72</sup> We will stick with NumPy arrays here, but below demonstrates creating a matrix object if you feel that you absolutely must use them. See the NumPy documentation page for further details on attributes and methods for this class of object as they will not be covered here.<sup>73</sup>

```
[in]: mat = np.matrix([[1, 8], [3, 2]])  
[in]: mat  
[out]:  
matrix([[1, 8],  
       [3, 2]])
```

### 8.3.1 Mathematical Operations with Arrays

Being that we are using NumPy arrays, the standard mathematical operations use the `+`, `-`, `*`, `/`, and `**` operators as demonstrated in chapter 4. There are a few other operations and methods, however, that are important for matrices such as calculating the inverse, determinant, transpose, and the dot product. For these operations, we have the following methods provided by NumPy's `linalg` module (Table 8.3) which are demonstrated in the following sections.

---

72 Linear Algebra (`scipy.linalg`) Documentation.  
<https://docs.scipy.org/doc/scipy-1.1.0/reference/tutorial/linalg.html>.

73 NumPy's Matrix Class Documentation Page.  
<https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.matrix.html>.

**Table 8.3** Common np.linalg Methods

| Method   | Description                                       |
|----------|---------------------------------------------------|
| dot ()   | Calculates the dot product                        |
| inv ()   | Returns the inverse of an array                   |
| det ()   | Returns the determinant of an array <sup>74</sup> |
| solve () | Solves a system of linear equations               |

In addition, it is worth reiterating that there is a general NumPy array method transpose () that will transpose or rotate the array around the diagonal. There is a convenient array.T shortcut that is often used. See section 4.2.3 for details.

### 8.3.2 Solving Systems of Equations

Solving systems of equations can be a tedious process by hand, but solving them using matrices can save time and effort. Let us say we want to solve the following system of equations for  $x$ ,  $y$ , and  $z$ .

$$\begin{aligned} 6x + 10y - 5z &= 21 \\ 2x + 7y + z &= 13 \\ -10x - 11y + 11z &= -21 \end{aligned}$$

These equations can be rewritten in matrix or array form as follows.

$$\begin{bmatrix} 6 & 10 & -5 \\ 2 & 7 & 1 \\ -10 & -11 & 11 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 21 \\ 13 \\ -21 \end{bmatrix}$$

We will call the first array  $M$ , the second  $X$ , and the third  $y$ , so we get

$$M \cdot X = y$$

We can solve for  $X$  by multiplying (dot product) both sides by the inverse of  $M$  (i.e.,  $M^{-1}$ ). Anything multiplied by its inverse is the identity, so  $M^{-1} \cdot M = 1$ .

$$\begin{aligned} M^{-1} \cdot M \cdot X &= M^{-1} \cdot y \\ X &= M^{-1} \cdot y \end{aligned}$$

---

<sup>74</sup> This is useful to determine if an array has an inverse by returning a value other than zero.

To get the inverse of a matrix or array, we can use the `np.linalg.inv()` function provided by NumPy's linear algebra module and use the `dot()` method to take the dot product.

```
[in]: M = np.array([[6, 10, -5],
                   [-2, 7, 1],
                   [-10, -11, 11]])
y = np.array([21, 13, -21])

[in]: np.linalg.inv(M).dot(y)

[out]: array([1., 2., 1.])
```

This means that  $x = 1$ ,  $y = 2$ , and  $z = 1$ .

As a chemical example, we can use the above mathematics and Beer's law to determine the concentration of three light absorbing analytes in a solution. The mathematical representation of the law is written below where  $A$  is absorbance (unitless),  $b$  is path length (cm),  $C$  is concentration (M), and  $\varepsilon$  is the molar absorptivity ( $\text{cm}^{-1}\text{M}^{-1}$ ). The latter value is analyte dependent.

$$A = \varepsilon b C$$

For a path length of 1.0 cm, which is quite common, the equation simplifies down to:

$$A = \varepsilon C$$

When there are three analytes,  $x$ ,  $y$ , and  $z$ , the absorption of light at a given wavelength equals the sum of the individual absorptions.

$$A = \varepsilon_x C_x + \varepsilon_y C_y + \varepsilon_z C_z$$

If we measure the absorbance of a three analyte solution at three different wavelengths ( $\lambda$ ), we get the following three equations.

$$\begin{aligned} A_{\lambda 1} &= \varepsilon_{x \lambda 1} C_x + \varepsilon_{y \lambda 1} C_y + \varepsilon_{z \lambda 1} C_z \\ A_{\lambda 2} &= \varepsilon_{x \lambda 2} C_x + \varepsilon_{y \lambda 2} C_y + \varepsilon_{z \lambda 2} C_z \\ A_{\lambda 3} &= \varepsilon_{x \lambda 3} C_x + \varepsilon_{y \lambda 3} C_y + \varepsilon_{z \lambda 3} C_z \end{aligned}$$

As long as we know the molar absorptivity of each analyte at each wavelength from pure samples, we have three unknowns and three equations, so we can calculate the concentration of each component. The above equations can be represented as matrices shown below.

$$\begin{bmatrix} \varepsilon_{x\lambda 1} & \varepsilon_{y\lambda 2} & \varepsilon_{z\lambda 3} \\ \varepsilon_{x\lambda 1} & \varepsilon_{y\lambda 2} & \varepsilon_{z\lambda 3} \\ \varepsilon_{x\lambda 1} & \varepsilon_{y\lambda 2} & \varepsilon_{z\lambda 3} \end{bmatrix} \cdot \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix} = \begin{bmatrix} A_{\lambda 1} \\ A_{\lambda 2} \\ A_{\lambda 3} \end{bmatrix}$$

If the absorbances at the three wavelengths are 0.6469, 0.2823, and 0.2221, respectively, and we know the molar absorptivities, we get the following matrices.

$$\begin{bmatrix} 7.8 & 1.1 & 2.0 \\ 2.6 & 3.2 & 0.89 \\ 1.8 & 1.0 & 8.9 \end{bmatrix} \cdot \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix} = \begin{bmatrix} 0.6469 \\ 0.2823 \\ 0.2221 \end{bmatrix}$$

We simply solve for the concentration matrix as was done earlier. Again, this is solvable using NumPy as shown below.

```
[in]: E = np.array([[7.8, 1.1, 2.0],
                   [2.6, 3.2, 0.89],
                   [1.8, 1.0, 8.9]])
A = np.array([0.6469, 0.282274, 0.22214])
np.linalg.inv(E).dot(A)

[out]: array([0.07800008, 0.02300962, 0.00659441])
```

The concentrations are  $C_x = 0.078$  M,  $C_y = 0.023$  M, and  $C_z = 0.0066$  M.

### 8.3.3 Least-Square Minimization by the Normal Equation

Finding the line of best fit through data points can be accomplished by least-square minimization. What we are essentially looking for is an equation of the form  $y = mx + b$  that is as close as possible to the data points, and the mean square error determines what qualifies as “close.” If we rewrite this problem in matrix or array form, it will look like the following for a series of four point  $(x_n, y_n)$  on a two-dimensional plane. The first array contains a column of ones to multiply with  $b$ , so for the first row, we get  $mx_0 + b = y_0$ .

$$\begin{bmatrix} x_0 & 1 \\ x_1 & 1 \\ x_2 & 1 \\ x_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

We will call the leftmost matrix  $X$ , the center matrix  $\theta$ , and the rightmost matrix  $y$ .

$$X \cdot \theta = y$$

Ultimately, we are looking for the values of  $m$  and  $b$ , so we need to find matrix  $\theta$ . This can be accomplished through optimization algorithms, or in the case of linear regression, there is a direct solution known as the *normal equation* shown below where  $X^T$  is the transpose of  $X$ .

$$(X^T X)^{-1} X^T y = \theta$$

As an example, below is a table of synthetic data for copper cuprizone absorbances at various concentrations at 591 nm.<sup>75</sup> We can use a linear fit to create a calibration curve from this data.

**Table 8.4** Beer-Lambert Law Data for Copper Cuprizone

| Concentration ( $10^{-6}$ M) | Absorbance |
|------------------------------|------------|
| 1.0                          | 0.0154     |
| 3.0                          | 0.0467     |
| 6.0                          | 0.0930     |
| 15                           | 0.2311     |
| 25                           | 0.3925     |
| 35                           | 0.5413     |

```
[in]: C = np.array([1.0e-06, 3.0e-06, 6.0e-06, 1.5e-05, 2.5e-05, 3.5e-05])
A = np.array([0.0154, 0.0467, 0.0930 , 0.2311, 0.3975, 0.5413])

[in]: y = A
      X = np.vstack((C, np.ones(6))).T
```

For the sake of readability, the calculation using the normal equation has been split in half.

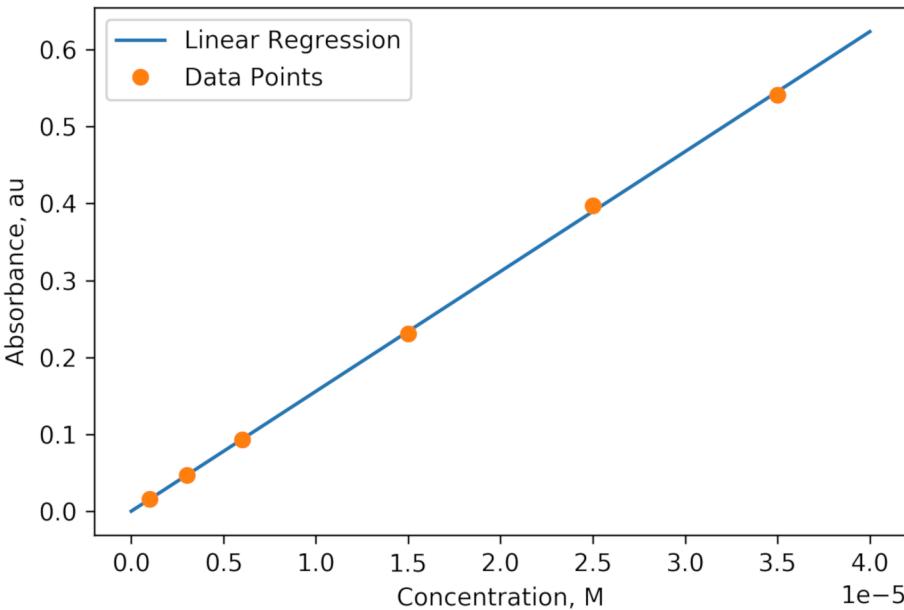
```
[in]: u = np.linalg.inv(X.T.dot(X))
      v = X.T.dot(y)
      theta = u.dot(v)

[in]: theta

[out]: array([ 1.55886203e+04, -5.45355390e-06])
```

---

<sup>75</sup> Values were calculated using the reported molar absorptivity from Porter, L. A.; Washer, B. M.; Hakim, M. H.; Dallinger, R. F. *J. Chem. Educ.* **2016**, 93 (7), 1305-1309.



A plot of the linear regression and the data points is shown above, and the linear regression returned a molar absorptivity of  $1.55 \times 10^4 \text{ cm}^{-1}\text{M}^{-1}$ . The regression also returned a  $y$ -intercept value of  $-5.45 \times 10^{-6}$ , which is below the detection limits making it practically zero. This makes sense because the  $y$ -intercept should always be approximately zero if the background is subtracted.

## 8.4 Calculus

Sympy and SciPy both contain functionality for performing calculus operations. We will start with SymPy for the symbolic math and switch over to SciPy for the strictly numerical work in section 8.4.3. In this section (8.4), we will be working with the radial density functions ( $\psi$ ) for hydrogen atomic orbitals. The squares of these functions ( $\psi^2$ ) provide the probability of finding an electron with respect to distance from the nucleus. While these equations are available in various textbooks, SymPy provides a `physics` module with a `R_nl()` function for generating these equations based on the principle ( $n$ ) quantum number, angular ( $l$ ) quantum number, and the atomic number ( $Z$ ). For example, to generate the function for the 2p orbital of hydrogen,  $n = 2$ ,  $l = 1$ , and  $Z = 1$ .

```
[in]: from sympy.physics.hydrogen import R_nl
[in]: r = sympy.symbols('r')
R_21 = R_nl(2, 1, r, Z=1)
[in]: R_21
[out]: sqrt(6)*r*exp(-r/2)/12
```

This provides the wavefunction equation with respect to the radius,  $r$ . We can also convert it to a Python function using the `sympy.lambdify()` method.

```
[in]: f = sympy.lambdify(r, R_21, modules='numpy')
```

This function is now callable by providing a value for  $r$ .

```
[in]: f(0.5)
```

```
[out]: 0.07948602207520471
```

### 8.4.1 Differentiation

SymPy can take the derivative of mathematical expression using the `sympy.diff()` function. This function requires a mathematical expression, the variable with respect the derivative is taken from, and the degree. The default behavior is to take the first derivative if a degree is not specified.

```
sympy.diff(expr, r, deg)
```

As an example problem, the radius of maximum density can be found by taking the first derivative of the radial equation and solving for zero slope.

```
[in]: dR_21 = sympy.diff(R_21, r, 1)
dR_21
```

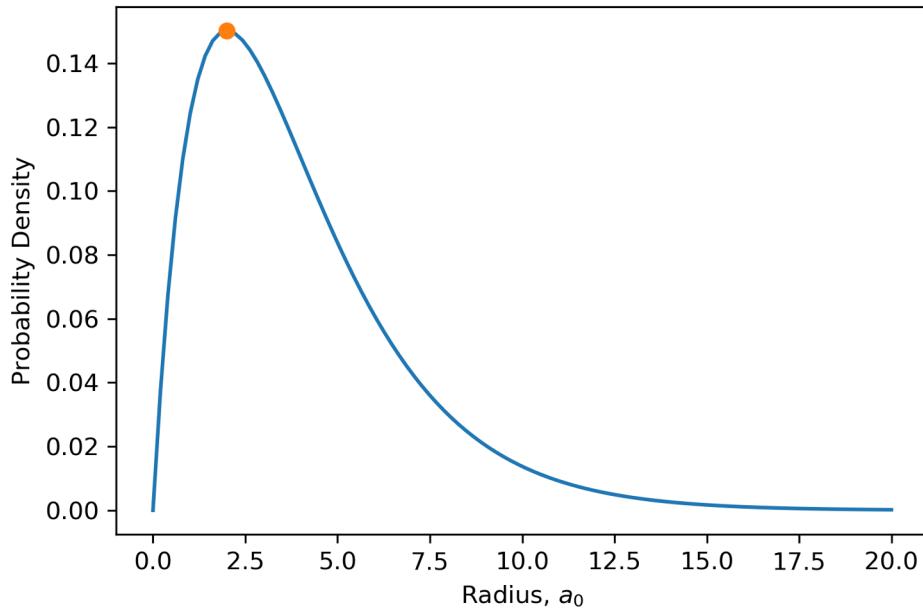
```
[out]: -sqrt(6)*r*exp(-r/2)/24 + sqrt(6)*exp(-r/2)/12
```

```
[in]: mx = float(sympy.solve(dR_21)[0])
```

The `solve()` function returns an array, so we need to index it to get the single value out. We can plot the radial density and the maximum density point to see if it worked.

```
[in]: R = np.linspace(0,20,100)
plt.plot(R, f(R))
plt.plot(mx, f(mx), 'o')
plt.xlabel('Radius, $a_0$')
plt.ylabel('Probability Density')

[out]:
```



The radius is in Bohrs ( $a_0$ ) which is equal to approximately 0.53 angstroms.

### 8.4.2 Integration of Functions

Sympy can also integrate expressions using the `sympy.integrate()` function which takes the mathematical expression and the variable plus integration range in the form of a tuple. If the integration range is omitted, then SymPy will return a symbolic expression.

The normalized (i.e., totals to one) density function is the squared wave function times  $r^2$  (i.e,  $\psi^2 r^2$ ). We can use this to determine the probability of finding an electron in a particular range of distances from the radius. Below, we integrate from the nucleus to the radius of maximum density.

```
[in]: sympy.integrate(R_21**2 * r**2, (r,0, mx)).evalf()
[out]: 0.0526530173437111
```

There is a 5.27% probability finding an electron between the nucleus and the radius of maximum probability. This is probably a bit surprising. Let's see the probability of finding an electron between 0 and 10 Bohrs from the nucleus.

```
[in]: sympy.integrate(R_21**2 * r**2, (r,0,10)).evalf()
[out]: 0.970747311923039
```

There is a 97.1% chance of finding the electron between 0 and 10 angstroms.

The SciPy library also includes functions in the `integrate` module for integrating mathematical functions. Information can be found on the SciPy documentation page listed at the end of this chapter under Further Reading.

### 8.4.3 Integrating Sampled Data

The above integration assumes a mathematical function is known. There are times when there is no known function to describe the data such as spectra. This is common in NMR spectroscopy and gas chromatography (GC) among many other applications where integrations of peak areas are used to quantify different components of a spectrum.

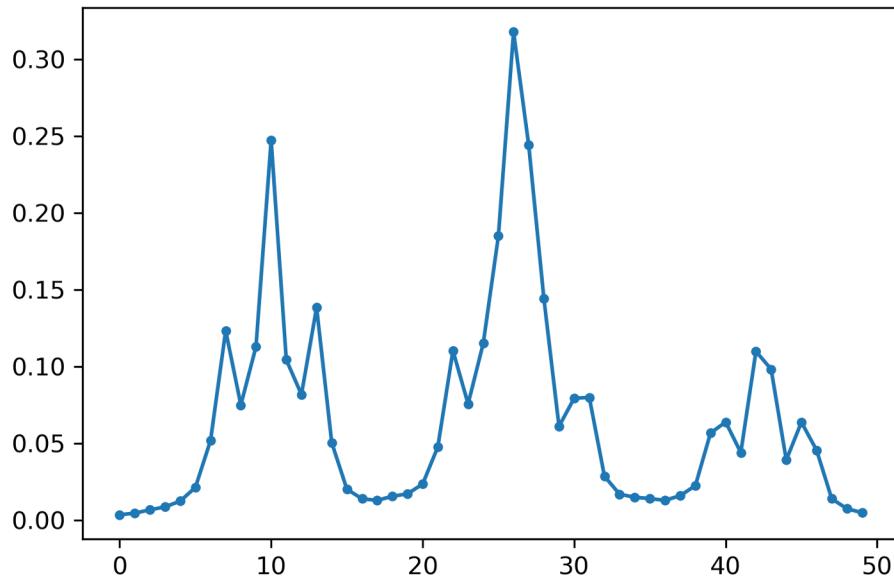
In the following example, we will use a section of a  $^1\text{H}$  NMR spectrum where we want to determine the ratio of the three triplet peaks via integration. NMR spectra are typically stored in binary files that require a special library to read, which is covered in chapter 11. For simplicity in this example, the data for a section of the NMR spectrum has been converted to a CSV file titled *Ar\_NMR.csv*.

```
[in]: nmr = np.genfromtxt('Ar_NMR.csv', delimiter=',')
      nmr

[out]: array([[0.0000000e+00, 3.42490660e-03],
              [1.0000000e+00, 4.52560300e-03],
              [2.0000000e+00, 6.67372160e-03],
              [3.0000000e+00, 8.58410100e-03],
              [4.0000000e+00, 1.23892580e-02],
              [5.0000000e+00, 2.12517060e-02],
              [6.0000000e+00, 5.18062560e-02],
              [7.0000000e+00, 1.23403220e-01],
              [8.0000000e+00, 7.49717060e-02],
              [9.0000000e+00, 1.12987520e-01],
              ...]
```

The imported data are stored in an array where the first column contains the index values and the second column contains the amplitudes.

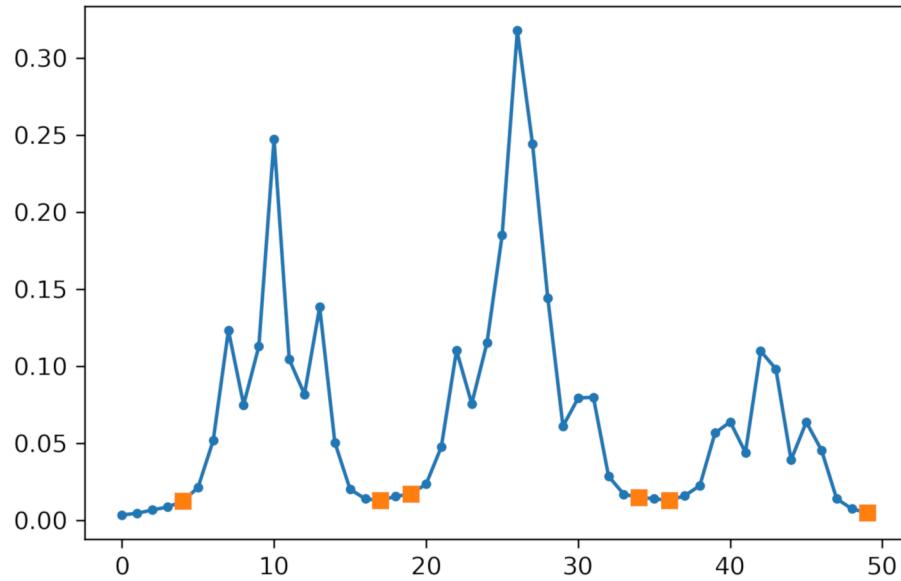
```
[in]: plt.plot(nmr[:,0], nmr[:,1], '-.')
      
```



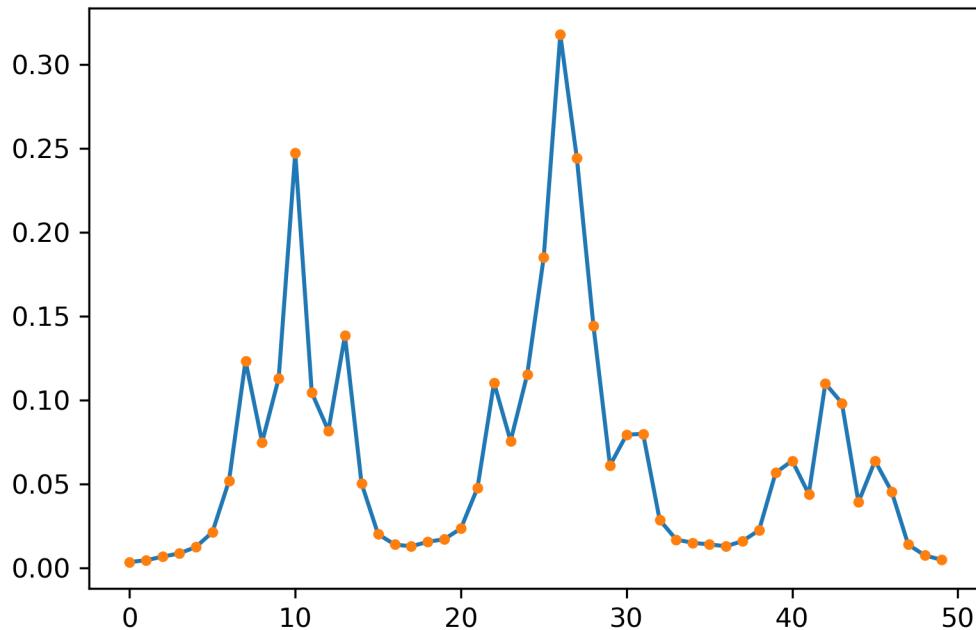
Above is a plot of the peaks with respect to the index values (not ppm). To integrate under each of the triplet peaks, first we need the index values for the edges of each peak. Below is a list, *i*, that provides reasonable boundaries, and a plot is below with these edges marked in orange squares.

```
[in]: i = [(4, 17), (19, 34), (36, 49)]
      plt.plot(nmr[:,1], '.-')
      for pair in i:
          for point in pair:
              plt.plot(point, nmr[point,1], 'C1s')

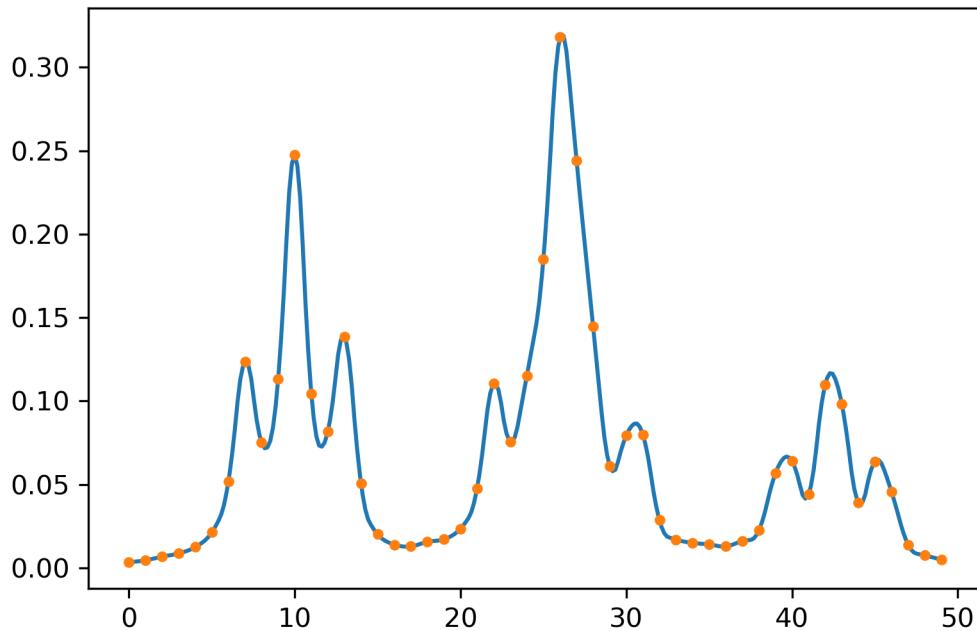
[out]:
```



Integrations under sampled data do not include the values between data points, so these regions are estimated based on assumptions. The `trapz` function assumes that any data point between known points lies directly between the known data points (i.e., linear interpolation) as shown below by the blue lines.



Alternatively, the `simps` function uses the Simpson's rule which estimates the data between known points using quadratic interpolation shown below.



Both functions take the y and x values in this order.

```
[in]: for peak in i:
    x = nmr[peak[0]:peak[1], 0]
    y = nmr[peak[0]:peak[1], 1]
    print(trapz(y, x))

[out]: 1.0401881535
       1.529880057
       0.5834871775
```

The three peaks have areas of approximately 2:3:1 ratio. Using Simpson's rule here gives approximately the same result.

#### **8.4.4 Integrating Ordinary Differential Equations**

Ordinary differential equations (ODE) mathematically describe the change of one or more dependent variables with respect to an independent variable. Common chemical applications include chemical kinetics, diffusion, electric current, among others. The SciPy `integrate` module provides an ODE integrator called `odeint()` which can integrate ordinary differential equations. This is useful for, among other things, integrating under kinetic differential equations to determine the concentration of reactants and products over the course of a chemical reaction.

For example, the following is a first-order chemical reaction with starting material, A, and product, P.<sup>76</sup>



The decay of a radioactive isotope is an example of a first-order reaction because the rate of decay is proportional to the amount of A. First-order reaction rates are described by

$$\text{Rate} = \frac{d[A]}{dt} = -k[A]$$

where [A] is the concentration (M) of A,  $k$  is the rate constant (1/s), and rate is the change in [A] versus time (M/s). The `odeint()` function below takes a differential equation in the form of a Python function (`func`), the initial values for A (`A0`), and a list or array of the times (`t`) to calculate the [A].

```
scipy.integrate.odeint(func, A0, t)
```

The Python function can be defined by a `def` statement or a lambda expression. The former is used below.

```
[in]: def rate_1st(A, t):
        return -k * A
```

The function should take the dependent variable(s) as the first positional argument and the independent variable as the second positional argument. In this example, A is the dependent variable and time, `t`, is the independent variable. If there are multiple dependent variables, they need to be provided inside a composite object like a list or tuple which can be unpacked through indexing or tuple unpacking once inside the function.<sup>77</sup> You may also notice that `t` is an unused argument in our Python function. It is included and required to signal to `odeint()` that the independent variable is `t`. The function is integrated below at times defined by `t`, and the initial concentration of A and rate constant are `A0` and `k`, respectively.

```
[in]: t = np.arange(0, 50, 4)    # time (seconds)

A0 = 1    # starting concentration (molarity)
k = 0.1   # rate constant in 1/s

A_t = scipy.integrate.odeint(rate_1st, A0, t)
P_t = A0 - A_t      # concentration of product
```

---

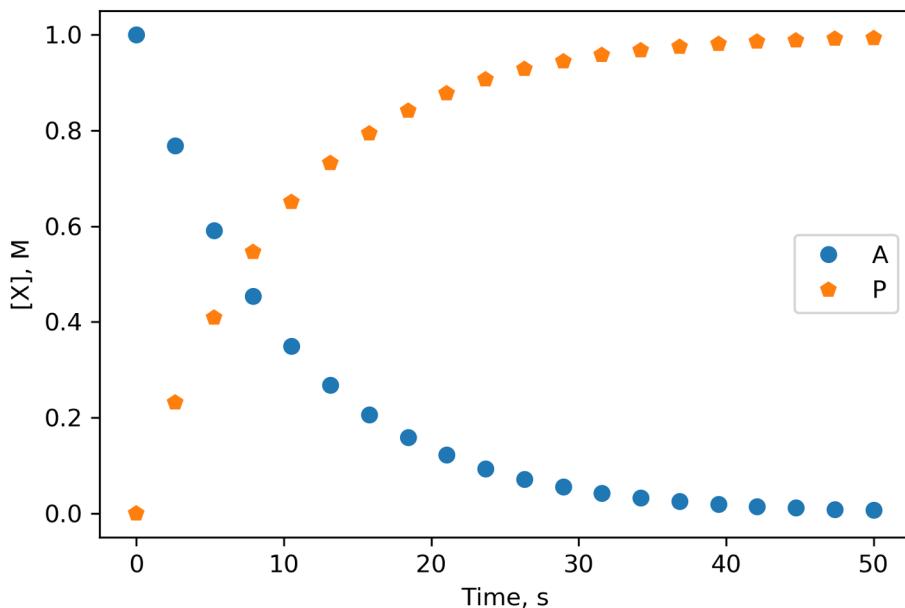
<sup>76</sup> We are assuming that this is an elementary step, so the reaction order is derivable from the stoichiometry.

<sup>77</sup> See chapter 9 for an example of multiple dependent variables.

The concentration of product ( $P_t$ ) is calculated through the difference between the initial concentration of starting material and the current concentration. That is, we assume that whatever starting material was consumed has become product. The results of the simulation have been visualized below.

```
[in]: plt.plot(t, A_t, 'o', label='A')
        plt.plot(t, P_t, 'p', label='P')
        plt.xlabel('Time, s')
        plt.ylabel('[X], M')
        plt.legend()
```

```
[out]:
```



This approach to kinetic simulations can be adapted to even more complex reactions which are demonstrated in chapter 9.

## 8.5 Mathematics in Python

Between SymPy, NumPy, SciPy, and Python's built-in functionality, there is often more than one way to carry out calculations in Python. For example, finding roots and derivatives of polynomials can be, along with the approaches demonstrated in this chapter, calculated by creating a NumPy `Polynomial` object and using NumPy's `roots()` and `deriv()` methods, respectively. How you carry out a calculation can often come down to matter of person preference, though there are differences in terms of speed and the output format. Find what works for you and do not necessarily worry if others are doing the same calculations through a different library or set of functions.

## Further Reading

1. SymPy Website. <http://www.sympy.org/en/index.html>.
2. SciPy and NumPy Documentation Pages. <https://docs.scipy.org/doc/>.

## Exercises

### Symbolic Mathematics

1. Factor the following polynomial using SymPy:  $x^2 + x - 6$
2. The following expression describes the rate of consumption of a starting material (x) that undergoes unimolecular decomposition, dimerization, and second-order reaction with another starting material (y). The rate constants are 3, 1, and 2, respectively.

$$\text{Rate} = 3x + x^{**2} + 2*x*y$$

Simplify the above mathematical expression using SymPy.

3. Expand the following expression using SymPy:  $(x - 2)(x + 5)(x)$
4. A 53.2 g block of lead ( $C_p = 0.128 \text{ J/g}\cdot\text{^\circ C}$ ) at 128 °C is dropped into a 238.1 g water ( $C_p = 4.18 \text{ J/g}\cdot\text{^\circ C}$ ) at 25.0 °C. What is the final temperature of both the lead and water?

*Hint:* Assume this is an isolated system, so  $q_{\text{lead}} + q_{\text{water}} = 0$ . We also know that  $q = mC_p\Delta T$ .

5. The following equation relates the  $\Delta G$  with respect to the equilibrium constant  $K$ .

$$\Delta G = \Delta G^\circ - RT \ln(K)$$

If  $\Delta G^\circ = -1.22 \text{ kJ/mol}$  for a chemical reaction, what is the value for  $K$  for this reaction at 298 K? Use the `sympy.solve()` function to solve this problem. Remember that equilibrium is when  $\Delta G = 0 \text{ kJ/mol}$ , and watch your energy units. ( $R = 8.314 \text{ J/mol}\cdot\text{K}$ )

### Matrices

6. A matrix or array of  $x,y$  coordinates can be rotated on a two-dimensional plane around the origin by multiplying by the following rotation matrix ( $M_R$ ). The angle ( $\theta$ ) is in radians, and the coordinates are rotated clockwise around the origin.

$$M_R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Below is an example using three generic points on the  $x,y$  plane.

$$\begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \end{bmatrix} \cdot \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} = \begin{bmatrix} x'_0 & y'_0 \\ x'_1 & y'_1 \\ x'_2 & y'_2 \end{bmatrix}$$

- a) Given the following coordinates for the four atoms in carbonate ( $\text{CO}_3^{2-}$ ) measured in angstroms, rotate them  $90^\circ$  clockwise. Plot the initial and rotated points in different colors to show that it worked.

C: (2.00, 2.00) O1: (2.00, 3.28) O2: (0.27, 1.50) O3: (3.73, 1.50)

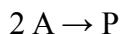
- b) Package the above code into a function that takes an array of points and an angle and performs the above rotation.
7. Using the rotation matrix described in the above problem, write a function that rotates the carbonate anion around its own center of mass. The suggested steps to complete this task are listed below.
- Calculate the center of mass
  - Subtract the center of mass from all points to shift the cluster to the origin.
  - Rotate the cluster of points.
  - Add the center of mass back the cluster the shift the points back.

## ***Calculus***

8. The following is the equation for the work performed by a reversible, isothermal (i.e., constant T) expansion of a piston by a fixed quantity of gas.

$$w = \int_{V_i}^{V_f} -nRT \frac{1}{V} dV$$

- a) Using SymPy, integrate this expression symbolically for  $V_i \rightarrow V_f$ . Try having SymPy simplify the answer to see if there is a simpler form.
- b) Integrate the same express above for the expansion of 2.44 mol of He gas from 0.552 L to 1.32 L at 298 K. Feel free to use either SymPy or SciPy.
9. Using `odeint()`, simulate the concentration of starting material for the second-order reaction below and overlay it with the second-order integrated rate law to show that they agree.





# *Chapter 9*

## *Simulations*

Simulations are a major component of modern chemical research either in conjunction with experimental work or by itself. A *digital chemical simulation* is a representation or mimic of a physical or chemical process using a computer with enough detail that the results provide meaningful and useful insights into the real process. Simulations do not need represent every aspect of the real world as long as the omitted details do not reduce the accuracy or precision to a level that the simulation is no longer useful.

Modern chemical simulations are often quite complex and are performed with a range of free or commercial software that regrettably can obfuscate the underlying methods. This chapter aims to introduce simulations with simple methodologies that can be easily coded in Python, NumPy, and SciPy. These simulations are not designed for use in a research setting due to the low level of sophistication and do not represent the current state-of-art in the field of chemical simulations. Some of these simulations are also not as computationally efficient as they could be because efficiency is sometimes sacrificed for simplicity and accessibility.

The simulations in this chapter assume the following imports from NumPy, SciPy, and matplotlib.

```
[in]: import numpy as np
       import scipy.integrate
       import matplotlib.pyplot as plt
       %matplotlib inline
```

## 9.1 Deterministic Simulations

Simulations with no random variables have fixed outcomes dictated by the code and input parameters. If these simulations are run multiple times using the same parameters, the outcomes of the simulations will be exactly identical. This is a category of simulations known as *deterministic simulations*. Even though many physical and chemical processes are driven by randomness, such as the random movements and collisions of molecules, they can often still be simulated deterministically because with a large number of molecules, the randomness often conforms to predictable patterns.<sup>78</sup> This is the case with Nuclear Magnetic Resonance (NMR) splitting patterns and chemical kinetics among many others.

### 9.1.1 Nuclear Magnetic Resonance Splitting

The splitting patterns observed in  $^1\text{H}$  NMR spectra are typically generated by neighboring protons possessing spins of  $+1/2$  or  $-1/2$  which alter the magnetic field around the observed proton. Even though the signs of the neighboring protons are random, the sample contains such a large number of molecules that the ratio should be quite close to the theoretical value of approximately 1:1. As a result, we can simulate the splitting patterns generated in  $^1\text{H}$  NMR spectra deterministically by splitting all peaks into 1:1 doublets for every neighboring proton.

A recursive function<sup>79</sup> is defined below that generates the splitting pattern generated by equivalent protons. The function takes in the chemical shift of the peak(s) (`peaks`), the number of equivalent neighboring protons (`n`), the coupling constant (`J`) in Hz, and the frequency of observation (`freq`) in MHz; and it returns a list of the split peaks in ppm. Each time the function is called, it splits the existing peak(s) into doublets, and the function is then called again if more splits are necessary due to multiple equivalent neighboring protons. The function below also includes validity checks to ensure the user-provided parameters are what the function expects.

```
def split(peaks, n, J, freq=400):  
    '''(list, int, float, freq=num) -> list
```

Takes in a list of peak ppm values for a single resonance(`peaks`), the number of identical neighboring protons(`n`), the coupling constant (`J`) in Hz, and the frequency of observation (`freq`) in MHz and returns a list of ppm values for all peaks in the splitting pattern.

<sup>78</sup> For more on this topic, look for information on the *law of large numbers*.

<sup>79</sup> See section 2.7.3 for a review of recursive functions.

```

    '''

# check validity of input values
if type(peaks) != list:
    peaks = list([peaks])
if type(n) != int:
    print('Error: n must be an integer.')
    return None

# split the peak(s)
J_ppm = J / freq

new_peaks = []

for peak in peaks:
    new_peaks.extend([peak + 0.5 * J_ppm, peak - 0.5 * J_ppm])

n = n - 1

# perform next split or return result
if n > 0:
    return split(new_peaks, n, J, freq=freq)
else:
    return new_peaks

[in]: split(1.00, 2, J=3.4, freq=400)
[out]: [1.0085000000000002, 1.0, 1.0, 0.9915]

```

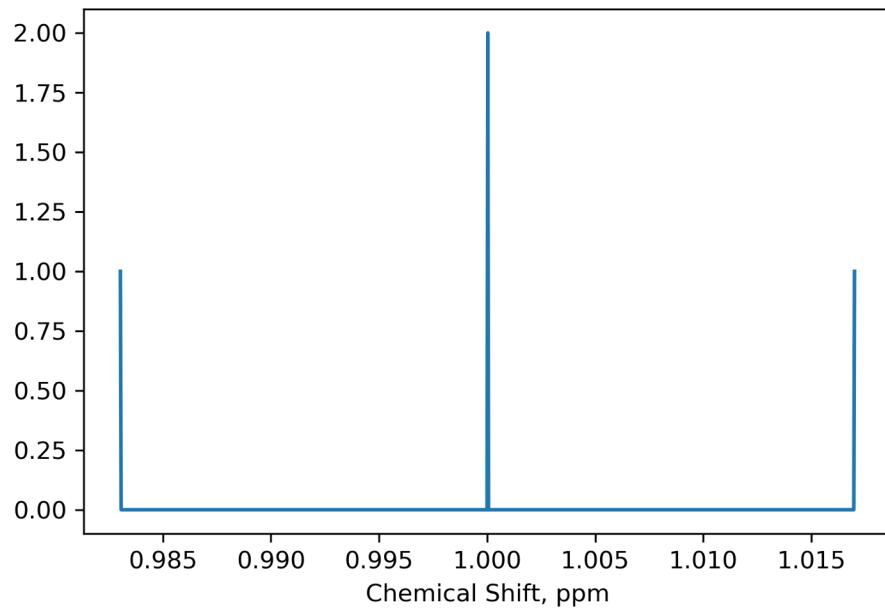
In the above example, a peak at 1.00 ppm has two neighboring protons that couple with it at 3.4 Hz, and the sample is observed at 400 MHz. There are four resulting peaks in the output list, but two peaks are at the same chemical shift of 1.00 ppm. This results in three peaks with the one at 1.00 ppm being twice the magnitude as the other two. We can visualize this by binning the peaks and generating a line plot.

```

[in]: signal, ppm = np.histogram(split([1.00], 2, J=6.8),
                                 bins=1000)
plt.plot(ppm[1:], signal)

[out]:

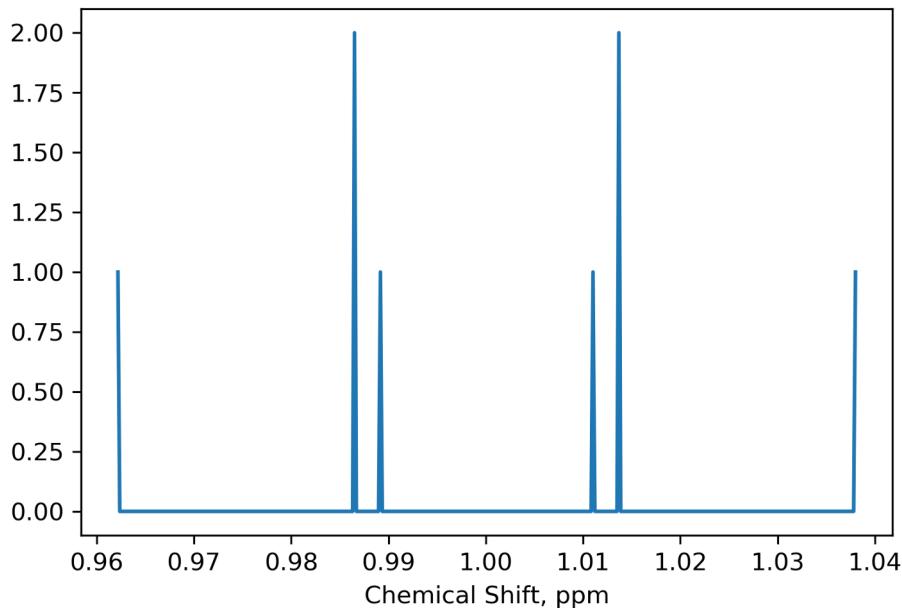
```



If there are multiple inequivalent groups of neighboring proton, this often results in more complex splitting patterns due to additional protons and additional coupling constants. This can be simulated by nesting the `split()` function and providing the different coupling constants. Below, we simulate a splitting pattern for a proton coupled with two protons with  $J = 9.8$  Hz and another proton with  $J = 10.8$ . This generates a doublet of triplets.

```
[in]: signal, ppm = np.histogram(split(split([1.00], 1,
  J=10.8), 2, J=9.8), bins=400)
      plt.plot(ppm[1:], signal)
      plt.xlabel('Chemical Shift, ppm')

[out]:
```



### 9.1.2 Single-Step Stepwise Chemical Kinetics

Another phenomenon that can be simulated deterministically<sup>80</sup> is the progress of a chemical reaction with respect to time. Many chemical reactions slow over the course of the reaction as the result of concentrations of reactants diminishing. This is because many reaction rates are dependent on the concentration of at least one starting material, and as the reaction progresses, starting material is consumed slowing the reaction.

One method for simulating this phenomenon is to incrementally calculate the rate of the chemical reaction at various points in the reaction based on the current concentrations. That is, at each small time step of the reaction, use the concentration(s) to calculate the current reaction rate and then increase/decrease the reaction concentrations by the amount calculated.

For example, we can simulate the following single-step chemical reaction of  $A \rightarrow P$ . Because this is an elementary step, the rate law is derivable from the stoichiometry where rate is M/s,  $k_{rxn}$  is the rate constant, and  $[A]$  is the concentration of A in molarity (M).

$$\text{Rate} = k_{rxn}[A]$$

To keep the math simple, we will make each step in the reaction one second. That way, if the rate is 0.1 M/s, we can simply subtract 0.1 M for one second of reaction. Let us choose a  $k =$

---

<sup>80</sup> Once again, chemical kinetics is dictated by a random process such as described in section 9.2.1 or due to random collisions between molecules, but because of the law of large numbers, it can often be simulated deterministically.

$0.05 \text{ s}^{-1}$  and an initial  $[A] = 1.00 \text{ M}$ . Therefore, the rate =  $(0.05 \text{ s}^{-1})(1.00 \text{ M}) = 0.05 \text{ M/s}$ , so the concentration of A should decrease by  $0.05 \text{ M}$  in the first second giving us  $0.95 \text{ M}$ . Now the rate of reaction is  $(0.05 \text{ s}^{-1})(0.95 \text{ M}) = 0.0475 \text{ M/s}$ , so we now subtract  $0.0475 \text{ M}$  from  $[A]$  for the next second of reaction to get  $0.903 \text{ M}$ . This continues for the entire duration of the simulation. Code for executing this process is shown below. A `for` loop runs the above process for each second of the simulation and records the new concentrations of A and P in NumPy arrays via assignment.

```
[in]: A, P = 1.00, 0.00 # molarity, M
k = 0.05             # 1/s for a first-order reaction
length = 100          # length of simulation in seconds
time = range(length + 1)

# create arrays to hold calculated concentrations
A_conc = np.empty(length + 1)
P_conc = np.empty(length + 1)

# simulation
for sec in time:
    # record concentration
    A_conc[sec] = A
    P_conc[sec] = P

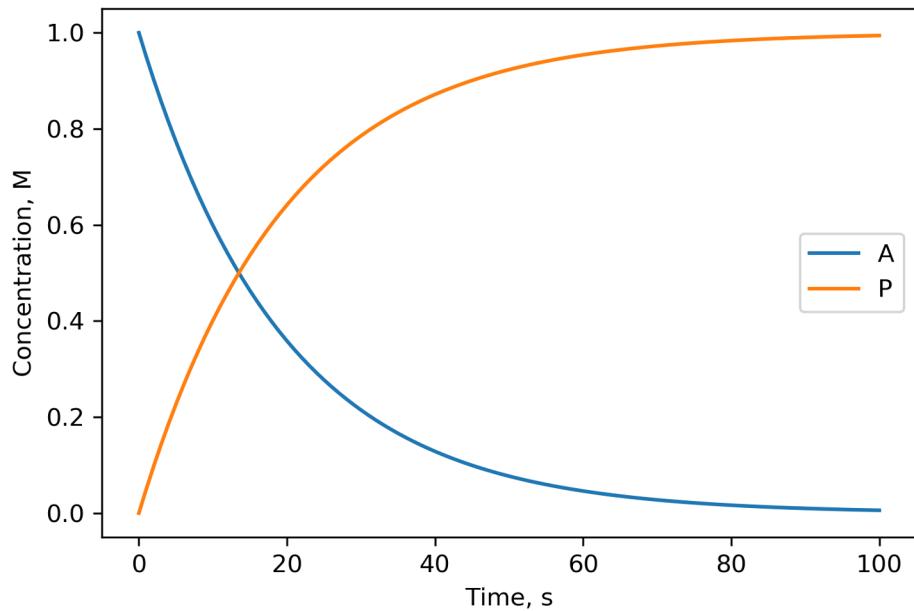
    # calculate rate
    rate = k * A

    # calculate new concentration
    A -= rate
    P += rate
```

You may be wondering why the first lines of code in the `for` loop record the concentrations instead of first decreasing them. This is because we need to record the initial concentration first before recalculating them. The next iteration will record the new concentrations before again recalculating rates and concentrations. Below is a plot of the simulation results.

```
[in]: plt.plot(time, A_conc, label='A')
plt.plot(time, P_conc, label='P')
plt.xlabel('Time, s')
plt.ylabel('Concentration, M')
plt.legend()
```

[out]:



We can overlay this plot with the theoretical values using the integrated first-order rate law below.

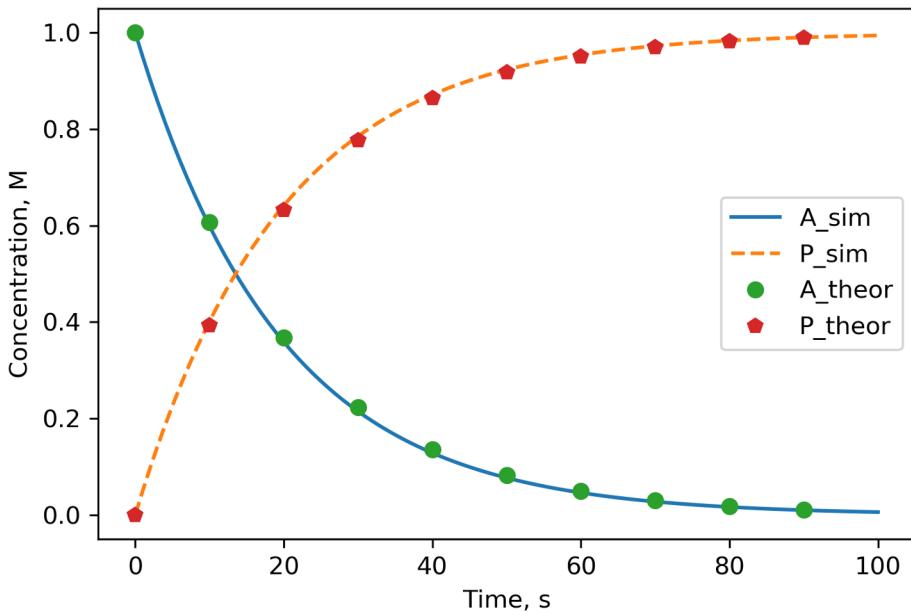
```
[in]: t = np.arange(0,100, 10)
A_theor = 1.0*np.exp(-k*t)
P_theor = np.ones(10) - A_theor

[in]: plt.plot(time, A_conc, '--', label='A_sim')
plt.plot(time, P_conc, '--', label='P_sim')
plt.xlabel('Time, s')
plt.ylabel('Concentration, M')

plt.plot(t, A_theor, 'o', label='A_theor')
plt.plot(t, P_theor, 'p', label='P_theor')

plt.legend()
```

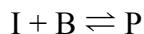
[out]:



The theoretical equation and simulation results are in good agreement. A closer inspection of the two shows a slight discrepancy between the two which is most noticeable earlier in the simulation. This is because the simulation only adjusts the rate every second while the theoretical equation can be thought of as recalculating the rate for infinitely small increments making the theoretical model more accurate.

### 9.1.3 Multistep Stepwise Chemical Kinetics

If we have a well-established theoretical equation for the above reaction of  $A \rightarrow P$ , why do we need the simulation? With this methodology, we can simulate more complicated reaction mechanisms, such as the multistep reaction below, even if we do not have the theoretical rate law.



In this reaction, starting material A converts to intermediate I in the first step followed by starting material B combining with I to form the product P. Both of these steps are reversible, so there are four rate constants. The code and output of the simulation are below. Unlike the previous simulation, the simulation below appends values to lists (e.g., `A_conc`).

```
[in]: A_conc, B_conc, I_conc, P_ conc = [], [], [], []
```

```

A, B, I, P = 1.0, 0.6, 0.0, 0.0 # initial conc, M
k1, k2, kr1, kr2 = 0.091, 0.1, 0.03, 0.01 # rate const
length = 200

for sec in range(length):
    A_conc.append(A)
    I_conc.append(I)
    B_conc.append(B)
    P_conc.append(P)

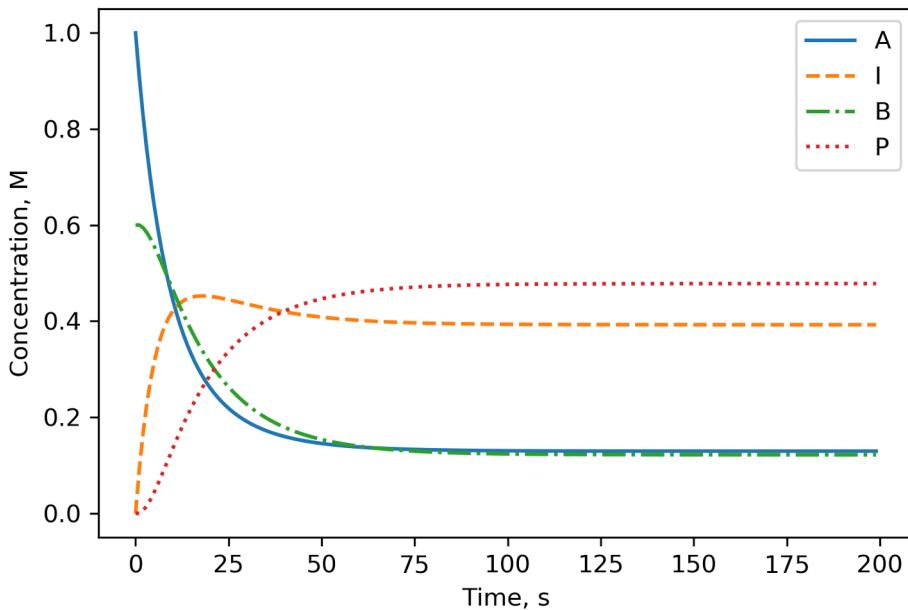
    rate_1 = k1 * A
    rate_r1 = kr1 * I
    rate_2 = k2 * B * I
    rate_r2 = kr2 * P

    A = A - rate_1 + rate_r1
    I = I + rate_1 - rate_2 - rate_r1 + rate_r2
    B = B - rate_2 + rate_r2
    P = P + rate_2 - rate_r2

[in]: plt.plot(range(length), A_conc, label='A', ls='--')
       plt.plot(range(length), I_conc, label='I', ls='---')
       plt.plot(range(length), B_conc, label='B', ls='-.')
       plt.plot(range(length), P_conc, label='P', ls=':')
       plt.xlabel('Time, s')
       plt.ylabel('Concentration, M')
       plt.legend()

[out]:

```



A word of caution regarding the above simulations: if the rate constants are increased enough, oscillating behavior and negative concentrations will be observed... the latter of which is clearly wrong.<sup>81</sup> This is because the simulation fails to recalculate the rates fast enough for the simulation, but this can be remedied by decreasing the step size.

#### 9.1.4 Chemical Kinetics and ODEINT

Another approach to performing the above kinetic simulations is to integrate the differential equations. For an introduction to integrating differential equations, see section 8.4.4. Below we will simulate a two step reaction where the first step is reversible. Because the following are the elementary steps, the rate equations can be inferred from the reaction stoichiometry.



The three differential equations tracking the concentrations of A, B, and P are shown below.

$$\frac{d[A]}{dt} = -k_1[A] + k_{r1}[B]$$

$$\frac{d[B]}{dt} = k_1[A] - k_2[B] - k_{r2}[B]$$

---

<sup>81</sup> Oscillating behavior is usually a sign of an error but not always. There are instances of oscillating “clock” reactions in the chemical literature.

$$\frac{d[P]}{dt} = k_2[B]$$

As is done in section 8.4.4, a Python function is created containing the differential equations, but in contrast to chapter 8, the differential equation for  $d[P]/dt$  is also included in the Python function instead of calculating  $[P]$  after the integration.<sup>82</sup>

```
[in]: k1, kr1, k2 = 0.2, 0.6, 0.3
      A0, B0, P0 = 1.0, 0.0, 0.0

      t = np.linspace(0, 50, 25)

      # define diff equation as Python function
      def rates(conc, t):
          A, B, P = conc
          dAdt = -k1 * A + kr1 * B
          dBdt = k1 * A - k2 * B - kr1 * B
          dPdt = k2 * B

          return dAdt, dBdt, dPdt
```

Because the `odeint()` function only takes the initial concentration (`A0`, `B0`, and `P0`) as a single argument, they need to be placed in a tuple.

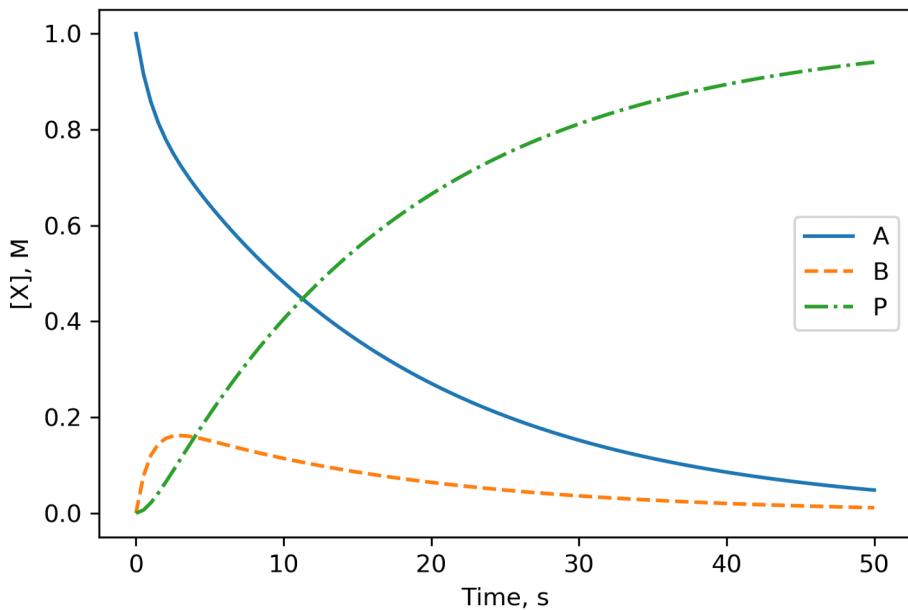
```
[in]: A_t, B_t, P_t = scipy.integrate.odeint(rates, (A0, B0,
P0), t).T

[in]: plt.plot(t, A_t, 'o', label='A')
      plt.plot(t, B_t, '^', label='B')
      plt.plot(t, P_t, 'p', label='P')
      plt.xlabel('Time, s')
      plt.ylabel('[X], M')
      plt.legend()

[out]:
```

---

<sup>82</sup> This is accomplished by calculating how much starting material (and intermediate if there is one) has been consumed (e.g., here it would be  $P_t = A0 - A_t - B_t$ ).



## 9.2 Stochastic Simulations

Unlike the deterministic simulations above, if the same code for a stochastic simulation is run multiple times, the results will vary at least slightly, though the overall patterns should be similar. This is because the outcome of stochastic simulations is determined by (pseudo) random number generators.<sup>83</sup> It is as if the results of the simulation are dictated by the flip of a coin or roll of a die. This analogy is so good that rolling dice repeatedly can simulate radioactive decay kinetics<sup>84</sup> among other things. We will use NumPy's `random` module to generate random values for the simulations.

### 9.2.1 Radioactive Decay

Radioactive decay is a random process, so logically it can be simulated as such. Every radioactive atom has a fixed probability of decaying each second just like a die has a fixed probability of rolling a one. A `for` loop is used for each second or step of the simulation, and a random number generator is used in each step to decide how many atoms decay. The `np.random.binomial()` function is used here to generate a series of zeros and ones

<sup>83</sup> “Random” numbers generated in software such as those by Python’s or NumPy’s `random` modules are never truly random, so they are really pseudo random numbers. However, they are random enough for our simulations. These would fail in applications such as cryptography where truly random number often requires for security.

<sup>84</sup> For example, Shultz, E. Dice Shaking as an Analogy for Radioactive Decay and First Order Kinetics *J. Chem. Educ.* **1997**, 74 (5), 505-507.

with a set probability of generating a one. In this simulation, a one signifies a decaying atom. These decayed atoms are tallied and subtracted from the current number of remaining atoms, and this value is recorded.

```
[in]: starting_atoms = 1000
       length = 10000 # length of simulation

       num_atoms = starting_atoms

       record = []

       for x in range(length):
           record.append(num_atoms)

           # "rolls" dice and tallies up number of zeros
           rolls = np.random.binomial(1, p=0.001,
                                       size=num_atoms)
           decayed_count = np.sum(rolls)

           # deduct decayed nuclei from the total
           num_atoms -= decayed_count

       record = np.array(record)
```

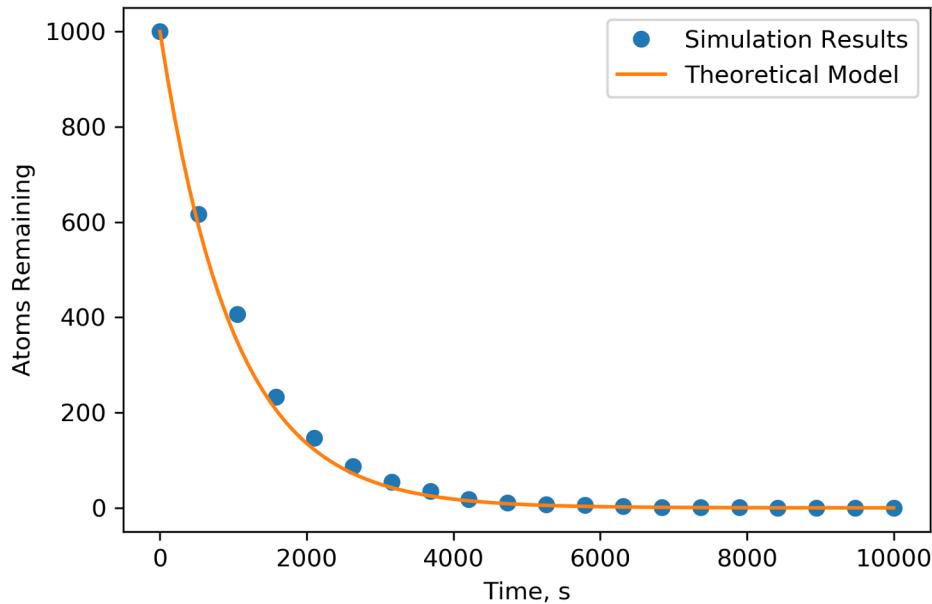
The results stored are in an array called `record` which can be plotted along with the first-order integrated rate law to see how the two compare. Being that there is a 1/1000 probability in the above simulation of each atom generating a one (decay), the rate constant ( $k$ ) is  $0.001 \text{ step}^{-1}$ . For ease of viewing, only twenty data points from the simulation are plotting below.

```
[in]: # plot of simulation
       step = np.linspace(0, length, 20)
       plt.plot(step, record[::500], 'o', label='Simulation
   Results')

       # plot of theoretical rate law
       t = np.linspace(0, length, 100)
       plt.plot(t, starting_atoms*np.exp(-1/1000 * t),
                 label='Theoretical Model')

       plt.xlabel('Time, s')
       plt.ylabel('Atoms Remaining')
```

```
plt.legend()
```



The simulation and theoretical model are in good but not perfect agreement. The deviation is a result of the simulation using random numbers and only simulating a relatively small number of molecules. If this simulation were run with increasingly larger number of molecules, the results are expected to converge on the theoretical prediction.

### 9.2.2 Confidence Intervals

Uncertainty is a part of all data, and uncertainty around a repeatedly measured and calculated value is often represented in the form of a 95% confidence interval (CI). This is the interval around the mean that has a 95% chance of containing the true value. Another way of describing 95% CI is that if we were to repeatedly collect a data set and calculate the 95% CI, the true value should be, statistically speaking, inside the confidence interval 95% of the time. Performing these experiments would be tedious, but this can be simulated in Python relatively easily.

The equation for calculating the 95% CI is shown below where  $\bar{x}$  is the average value in a set of repeated<sup>85</sup> measurements,  $s$  is the standard deviation (corrected),  $t$  is the statistical  $t$  value from a table, and  $N$  is the degrees of freedom. For 20 samples per set,  $t = 2.09$  and  $N = 19$ .

<sup>85</sup> That is, the same measurement was performed repeatedly.

$$95\% CI = \bar{x} \pm \frac{ts}{\sqrt{N}}$$

We can simulate the data collection by picking a true value and generating twenty samples by adding random error to twenty copies of the true value. Using the simulated data set, the 95% CI can be calculated, and we can test whether or not the true value is inside the CI. If we repeat this procedure numerous times recording the success or failure of the true value being inside the CI, we can calculate the success rate as demonstrated below.

```
[in]: trials = 100000
N = 20
t = 2.09
true = 6.2 # true value

# list contain a one for each time true inside 95% CI
in_interval = []

for trial in range(trials):
    # create synthetic data
    error = np.random.rand(N)
    data = np.ones(N) * true + (error - 0.5)

    # calculate the 95% CI
    avg = np.mean(data)
    CI_95 = t * np.std(data, ddof=1) / np.sqrt(N)
    lower = avg - CI_95
    upper = avg + CI_95

    # determine if true values is inside 95% CI
    if lower <= true <= upper:
        in_interval.append(1)

[in]: 100 * np.sum(in_interval) / trials
[out]: 94.769
```

The above simulation finds that 94.768% of the time, the true value is inside the 95% CI which is pretty close to what we expected. If this simulation is repeated, you will likely observe that the values are very often slightly below the expected 95%. This is the result of smaller data sets and should be closer to the theoretical value with increasing set size.

### 9.2.3 Random Flight Polymer

Polymers are long chains of repeating units called monomers. These chains can easily go for thousands of monomers and wind around in 3D space in seemingly random fashions. A single polymer chain can be made of a single type of monomer or multiple types and can be of varying lengths, but for the following polymer simulation, we will work with polymers of a fixed number of monomers and ignore the monomer types.

One model for polymer conformation is a *random flight polymer* which assumes that the conformation of the polymer is entirely random. We can simulate a random flight polymer through a *random walk* by making each subsequent segment of polymer extend in a random direction and distance. For simplicity, we will simulate the polymer in only two dimensions, but this simulation can be expanded to a third dimension. The random element of the simulation is provided by a NumPy random number generator which generates a random length and direction for each new segment.

The general procedure for the following simulation is to start the polymer chain at coordinate (0, 0), and for each new segment, add a random value to the  $x$ -coordinate of the previous polymer end and another random value to the  $y$ -coordinate. Each new coordinate is then appended to a list of coordinates (`coords`) for analysis and visualization. This simulation is coded below. The random values are floats from  $-1 \rightarrow 1$ . NumPy does not provide a function for generating this range, so we can modify the  $0 \rightarrow 1$  range from the `np.random.rand()` function by subtracting 0.5 and multiplying by 2.

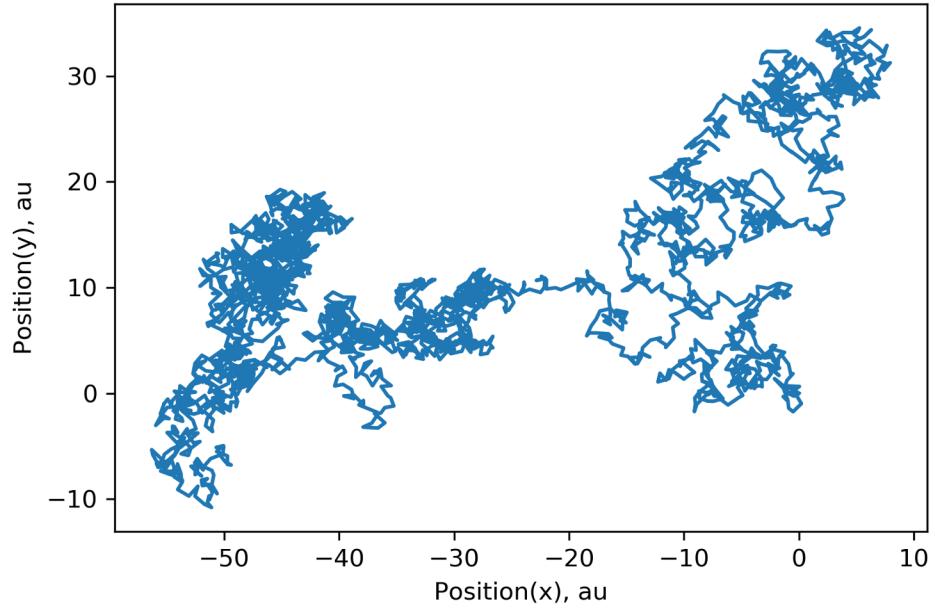
```
[in]: segments = 3000
       coords = [[0,0]]

       for step in range(segments):
           x = coords[step][0] + 2 * (np.random.rand() - 0.5)
           y = coords[step][1] + 2 * (np.random.rand() - 0.5)

           coords.append([x, y])

       coords = np.array(coords)

[in]: plt.plot(coords[:,0], coords[:,1])
      plt.xlabel('Position(x), au')
      plt.ylabel('Position(y), au')
```



The results of the simulation show a polymer strand winding around in an apparently random fashion. If we rerun the above simulation, a different looking polymer conformation will be generated.

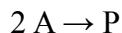
## Further Reading

1. Downey, Allen *Modeling and Simulation in Python*. <https://github.com/AllenDowney/ModSimPy> (accessed August 2020).
2. Weiss, C. J. Introduction to Stochastic Simulations for Chemical and Physical Processes: Principles and Applications. *J. Chem. Educ.* **2017**, 94 (12), 1904–1910.
3. For examples of chemical kinetics scenarios to model, see: Bentenitis, N. A Convenient Tool for the Stochastic Simulation of Reaction Mechanisms. *J. Chem. Educ.* **2008**, 85 (8), 1146–1150.

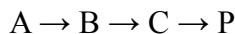
## Exercises

### Kinetic Simulations

1. Using `scipy.integrate.odeint()` and a differential equation, plot the concentration of starting material A with respect to time for a third-order reaction.
2. Create a simulation of the following single-step reaction and overlay it with the appropriate integrated rate law. The rate constant is  $0.28 \text{ M}^{-1}\text{s}^{-1}$ . Feel free to start with code from this chapter and modify it as needed.



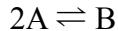
3. Plot the concentrations of A, B, C, and P with respect to time for the following three-step, non-reversible mechanism.



The initial concentrations and rate constants are in the table below.

| Step | Specie | [Specie] <sub>0</sub> , M | Rate Constant, s <sup>-1</sup> |
|------|--------|---------------------------|--------------------------------|
| 1    | A      | 1.50                      | 0.8                            |
| 2    | B      | 0.00                      | 0.4                            |
| 3    | C      | 0.00                      | 0.3                            |
| —    | P      | 0.00                      | —                              |

4. Simulate the following chemical equilibrium where the forward rate is described by  $\text{Rate}_f = (1.3 \times 10^{-2} \text{ M}^{-1}\text{s}^{-1})[\text{A}]^2$  and the reverse rate is described by  $\text{Rate}_r = (6.2 \times 10^{-3} \text{ s}^{-1})[\text{B}]$ .



Use a `for` loop to simulate each second of reaction by calculating the rates and increasing/decreasing each concentration appropriately. Record the concentrations in lists and plot the results. Start with 2.20 M of A and 1.72 M B and run the simulation for at least 200 seconds. Notice that the rates are in M/s.

5. In section 9.1.3, a two-step, reversible reaction is simulated. If the rate constant  $k_{r1}$  is decreased to 0.01 s<sup>-1</sup>, what effect on the reaction do you anticipate? Simulate this to see if your prediction is correct.
6. Simulate two competing, first-order reactions of starting material A forming product P<sub>1</sub> and P<sub>2</sub> and plot the resulting concentrations of both products versus time. Use  $k_1 = 0.02$  M/s and  $k_2 = 0.04$  M/s and start with 2.00 M A. What do you predict the plot of concentration versus time to look like and the ratio of products to be? Does your simulation agree?

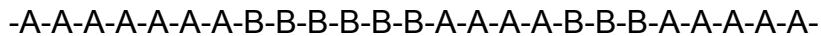


## Polymer Simulations

7. Polymers that consist of two or more different monomers are known as copolymers. Simulate an addition copolymer consisting of two monomers: ethylene (28.06 g/mol) and styrene (104.16 g/mol) with a fixed length of a thousand units. Given the molecular weights of the two monomers above, calculate the weights for a thousand simulated polymer strands and generate a histograms of the frequency versus weight.

*Hint:* try using the `np.random.binomial()` function with  $p=0.5$  and treat a zero as one monomer and a one is the other.

8. *Block copolymers* are polymers where multiple monomer types are clustered along the polymer chain instead of being randomly dispersed. These clusters are called blocks which may be of random lengths as the polymer switches between monomer types. An example is shown below.



Simulate a block copolymer consisting of two monomers with a total length of a hundred monomer units.

*Hint:* Append monomers (0 or 1) to a list inside a `for` loop, and use a function from `np.random.binomial()` to decide when to toggle between monomers types. Use `mono = 1 - mono` to make the switch.

9. The random flight polymer simulation presented in section 9.2.3 uses a `for` loop. As discussed in chapter 4, one of the virtues of NumPy is that it often avoids the computationally inefficient `for` loops. Below is the same simulation written in a single line of code leveraging the power of NumPy arrays. Briefly explain what it is doing and why it works.

```
loc = np.cumsum(np.random.randint(-1, high=2,  
size=(3000,2)), axis=0)
```

10. Proteins are nature polymers consisting of twenty common monomers called amino acids. Simulate a random protein strand of a thousand units long using the `np.random.randint()` function and a Python dictionary containing the single letter amino acid codes.

## Other Simulations

11. Confidence intervals
- Convert the code for calculating a 95% confidence interval in section 9.2.2 to a Python function that accepts a number of samples as the one argument and returns the percentage of the time the true value is inside the confidence interval. You will need to look up  $t$  values and generate a dictionary that converts degrees of freedom ( $N$ ) to  $t$  values.
  - Using a `for` loop, calculate the percentage of the time the true value is in the 95% confidence interval for each of the sample sizes in the above dictionary and plot the results. Describe the trend.
12. Simulate the diffusion of molecules along a single axis. Start all molecules at zero, and for each step of the simulation, add a random number, positive or negative, to each value in the array. Plot the results in a histogram.
13. Using the function from 9.1.1, simulate the splitting pattern for the tertiary proton in isopropyl alcohol ((CH<sub>3</sub>)<sub>2</sub>CHOH). In CDCl<sub>3</sub>, this proton is observed at 3.82 ppm with a coupling constant of 6 Hz.<sup>86</sup> Assume no coupling with the hydroxyl proton is observed.

---

<sup>86</sup> Data from: Fulmer, G. R.; Miller, A. J. M.; Sherden, N. H.; Gottlieb, H. E.; Nudelman, A.; Stoltz, B. M.; Bercaw, J. E.; Goldberg, K. I. NMR Chemical Shifts of Trace Impurities: Common Laboratory Solvents, Organics, and Gases in Deuterated Solvents Relevant to the Organometallic Chemist. *Organometallics* **2010**, 29, 2176–2179.

# *Chapter 10*

## *Plotting with Seaborn*

There are a number of plotting libraries available for Python including Bokeh, Plotly, and MayaVi; but the most prevalent library is still probably matplotlib. It is often the first plotting library a Python user will learn, and for good reason. It is stable, well supported, and there are few plots that matplotlib cannot generate. Despite its popularity, there are some drawbacks... namely, it is quite verbose. That is, you may be able to generate nearly any plot, but it will take at least a few lines of code if not dozens to create and customize your figure.

One very attractive alternative is the seaborn plotting library. While seaborn cannot generate the same variety of plots as matplotlib, it is good at generating a few common plots that people use regularly, and here is the key detail... it often does what would take matplotlib 10+ lines of code in only one or two lines. To make things even better, seaborn is built on top of matplotlib. This means that if you are not completely happy with what seaborn creates, you can fine tune it with the same matplotlib commands you already know! In addition, seaborn is designed to work closely with the pandas library. For example, think of all the lines of code you have typed to simply add labels to your *x*- and *y*-axes. Instead, seaborn often pulls the labels from the DataFrame column headers. Again, if you do not like this default behavior, you can still override it with `plt.xlabel()` and other commands that you already know.

By convention, seaborn is imported with the `sns` alias, but being that this is a relatively young library, it is unclear how strong this convention is. The official seaborn website<sup>87</sup> uses it, so we will as well. All code in this chapter assumes the following import.

```
[in]: import seaborn as sns
```

---

<sup>87</sup> Seaborn Website. <https://seaborn.pydata.org/index.html>

## 10.1 Seaborn Plot Types

A map of the seaborn plotting library is mainly a series of the different types of plots that it can generate. Below is table of the main categories. The rest of this chapter is a more in-depth survey of select plotting functions, but it is not a complete list.

**Table 10.1** Seaborn Plotting Type Categories Covered Herein

| Category     | Description                                                  |
|--------------|--------------------------------------------------------------|
| Regression   | Draws a regression line through the data                     |
| Categorical  | Plots frequency versus category                              |
| Distribution | Plots frequency versus a continuous value                    |
| Matrix       | Displays the data as a colored grid                          |
| Relational   | Visualizes the relationship between two continuous variables |

## 10.2 Regression Plots

Generating a regression line through data is a common task in science, and seaborn includes multiple plotting types that perform this task. All of the plots discussed below use a least square best fit and include a confidence interval for the regression line as a shaded region. Remember that there is uncertainty in both the slope and  $y$ -intercept for a regression line. If we were to plot all the possible variations of the regression line within the slope and intercept uncertainties, we get the regression confidence interval. By default, seaborn displays the 95% confidence interval, but this can be changed.

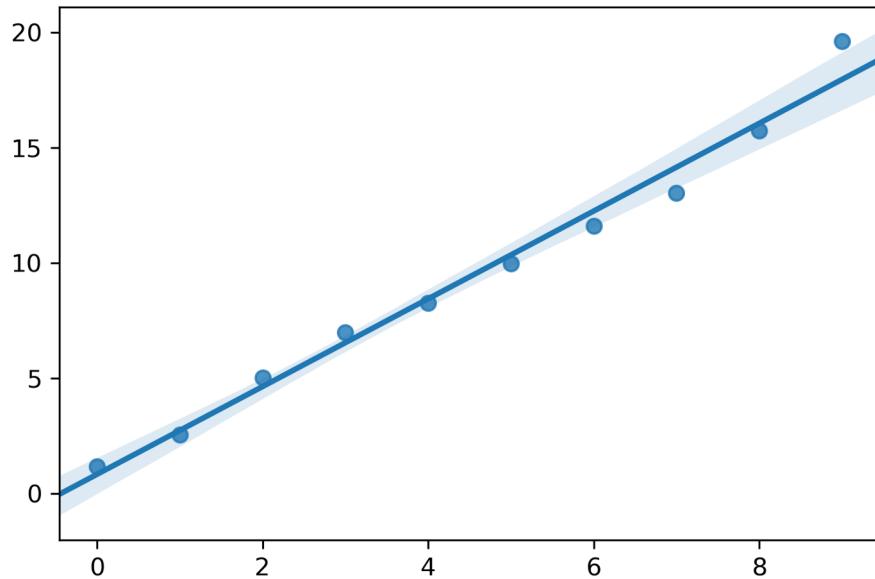
### 10.2.1 *regplot*

*regplot* generates a single scatter plot of data with a linear regression through the data points complete with a 95% confidence interval. The `sns.regplot()` function can take `x` and `y` positional arguments just like `plt.plot()`, but it also can take the `x` and `y` column names from a pandas DataFrame. Both approaches are demonstrated below.

```
[in]: x = np.arange(10)
       y = 2*x + np.random.randn(10)

[in]: sns.regplot(x, y)

[out]:
```



If the data is in a DataFrame, the  $x$  and  $y$  values can be provided as the column names, and seaborn will automatically add the column names as  $x$  and  $y$  labels. Below is a series of boiling point and molecular weights for various organic compounds.

```
[in]: bp = pd.read_csv('org_bp.csv')
      bp
```

[out] :

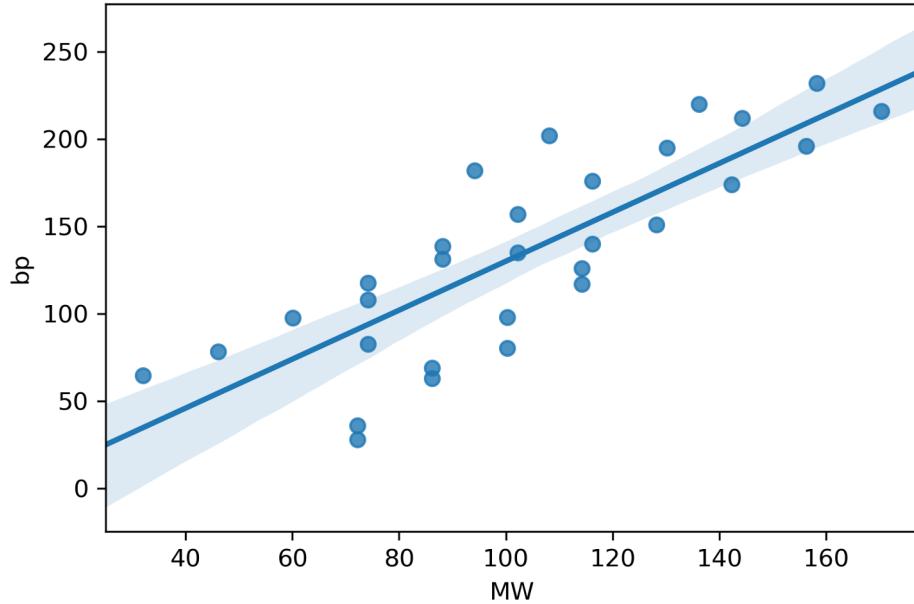
|           | <b>bp</b> | <b>MW</b> | <b>type</b> |
|-----------|-----------|-----------|-------------|
| <b>0</b>  | 66        | 32.04     | alcohol     |
| <b>1</b>  | 78        | 46.07     | alcohol     |
| <b>2</b>  | 98        | 60.1      | alcohol     |
| <b>3</b>  | 118       | 74.12     | alcohol     |
| <b>4</b>  | 139       | 88.15     | alcohol     |
| <b>5</b>  | 157       | 102.18    | alcohol     |
| <b>6</b>  | 176       | 116.2     | alcohol     |
| <b>7</b>  | 195       | 130.23    | alcohol     |
| <b>8</b>  | 212       | 144.25    | alcohol     |
| <b>9</b>  | 232       | 158.28    | alcohol     |
| <b>10</b> | 36        | 72.15     | alkane      |

|    |     |        |         |
|----|-----|--------|---------|
| 11 | 69  | 86.18  | alkane  |
| 12 | 98  | 100.21 | alkane  |
| 13 | 126 | 114.23 | alkane  |
| 14 | 151 | 128.26 | alkane  |
| 15 | 174 | 142.29 | alkane  |
| 16 | 196 | 156.31 | alkane  |
| 17 | 216 | 170.34 | alkane  |
| 18 | 63  | 86.18  | alkane  |
| 19 | 117 | 114.23 | alkane  |
| 20 | 28  | 72.15  | alkane  |
| 21 | 80  | 100.21 | alkane  |
| 22 | 108 | 74.12  | alcohol |
| 23 | 83  | 74.12  | alcohol |
| 24 | 131 | 88.15  | alcohol |
| 25 | 135 | 102.18 | alcohol |
| 26 | 140 | 116.2  | alcohol |
| 27 | 182 | 94.11  | alcohol |
| 28 | 202 | 108.14 | alcohol |
| 29 | 220 | 136.19 | alcohol |

If you choose to provide column names from a pandas DataFrame, you must also provide the name of the DataFrame using the `data` keyword argument.

```
[in]: sns.regplot(x='MW', y='bp', data=bp)
```

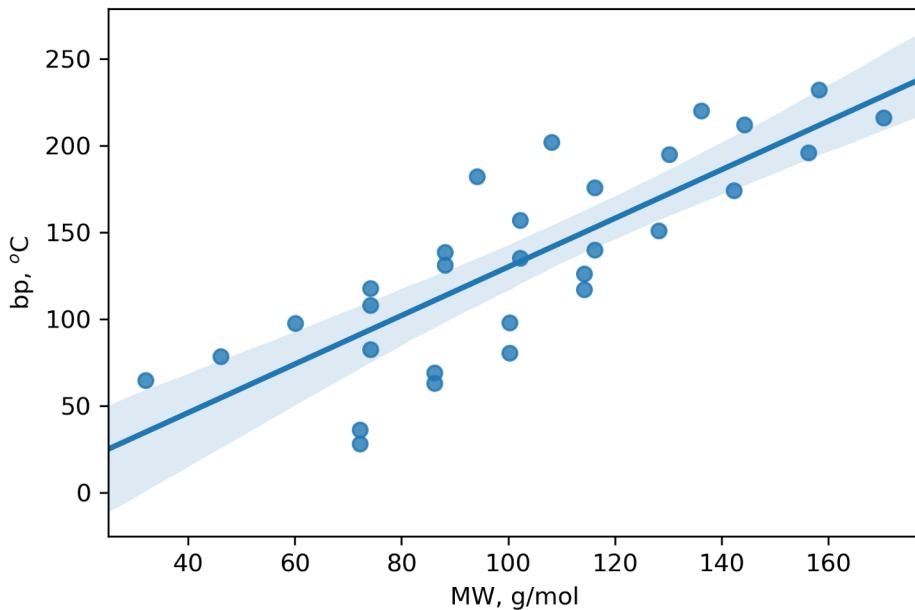
```
[out]:
```



While the DataFrame column names provide accurate axis labels, the units are missing. We can use matplotlib commands from chapter 3 to modify the axis labels.

```
[in]: sns.regplot(x='MW', y='bp', data=bp)
    plt.xlabel('MW, g/mol')
    plt.ylabel('bp, °C')
```

```
[out]:
```



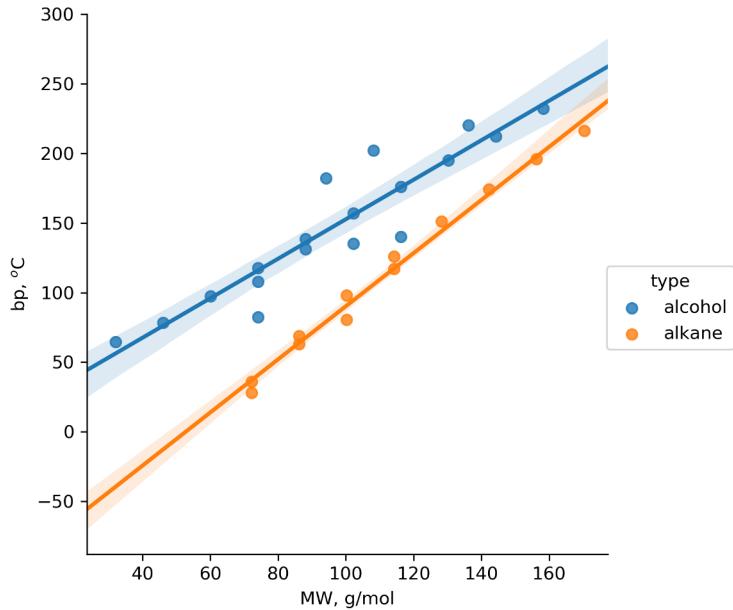
### 10.2.2 lmplot

An `lmplot` is very similar to the `regplot()` function except that an `lmplot` also allows for multiple regressions based on additional pieces of information about each data point. For example, the `org_bp.csv` file above contains the boiling points of various alcohols and alkanes along with their molecular weights. Chemical intuition might bring one to expect two independent boiling point trends between the alcohol and alkanes, so we need two independent regression lines for the two classes of organic molecule. The `lmplot` can do exactly this.

The `lmplot` function takes the `x` and `y` variables and the DataFrame name as either positional or keyword arguments, so the below function call could also be `sns.lmplot('MW', 'bp', bp, hue='type')`. The `hue` argument is the column name that dictates the color of the markers, so in this example, it will be the type of organic molecule.

```
[in]: sns.lmplot(x='MW', y='bp', hue='type', data=bp)
       plt.xlabel('MW, g/mol')
       plt.ylabel('bp, °C')
```

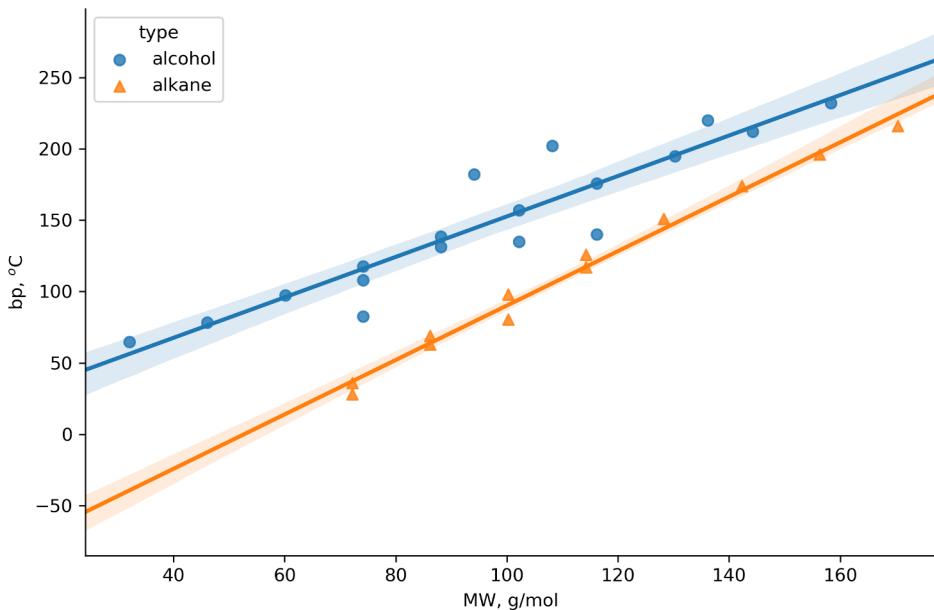
[out] :



The `lmplot()` function also provides arguments for modifying the appearance of the plot. Below is a demonstration of a few extra tweaks to the plot. The `legend_out` argument controls whether the legend is inside or outside the plot's boundaries, and the `aspect` argument sets the ratio of the *x*-axis versus the *y*-axis. The marker shapes can also be modified using the `markers` argument with matplotlib conventions from chapter 3.

```
[in]: sns.lmplot(x='MW', y='bp', hue='type', data=bp,
                  markers=['o', '^'],
                  legend_out=False, aspect=1.5)
      plt.xlabel('MW, g/mol')
      plt.ylabel('bp, $^{\circ}\text{C}$')

[out]:
```



## 10.3 Categorical Plots

Categorical plots contain one axis of continuous values and one axis of discrete or categorical values. For example, if the density of three metals were measured repeatedly in lab, we would want to plot measured density (continuous) with respect to metal identity (categorical). Below are a few fictitious laboratory measurements for the densities of copper, iron, and zinc.

**Table 10.2** Density (g/mL) Measurements for Different Metals

| Cu   | Fe   | Zn   |
|------|------|------|
| 8.51 | 7.95 | 6.79 |
| 9.49 | 7.53 | 7.06 |
| 8.48 | 8.09 | 7.96 |
| 9.40 | 7.44 | 7.06 |
| 8.83 | 8.38 | 6.69 |
| 9.45 | 7.83 | 7.21 |
| 8.73 | 6.88 | 7.35 |
| 9.00 | 7.90 | 6.65 |
| 8.84 | 8.51 | 7.41 |
| 9.32 | 7.89 | 7.89 |

If we want to compare these values, the density can be plotted on the  $y$ -axis and metal on the  $x$ -axis.

```
[in]: labels = ['Cu', 'Fe', 'Zn']
densities = [[8.51, 7.95, 6.79],
             [9.49, 7.53, 7.06],
             [8.48, 8.09, 7.96],
             [9.40, 7.44, 7.06],
             [8.83, 8.38, 6.69],
             [9.45, 7.83, 7.21],
             [8.73, 6.88, 7.35],
             [9.00, 7.90, 6.65],
             [8.84, 8.51, 7.41],
             [9.32, 7.89, 7.89]]
```

```
[in]: df = pd.DataFrame(densities, columns=labels)
df.head()
```

```
[out]:
```

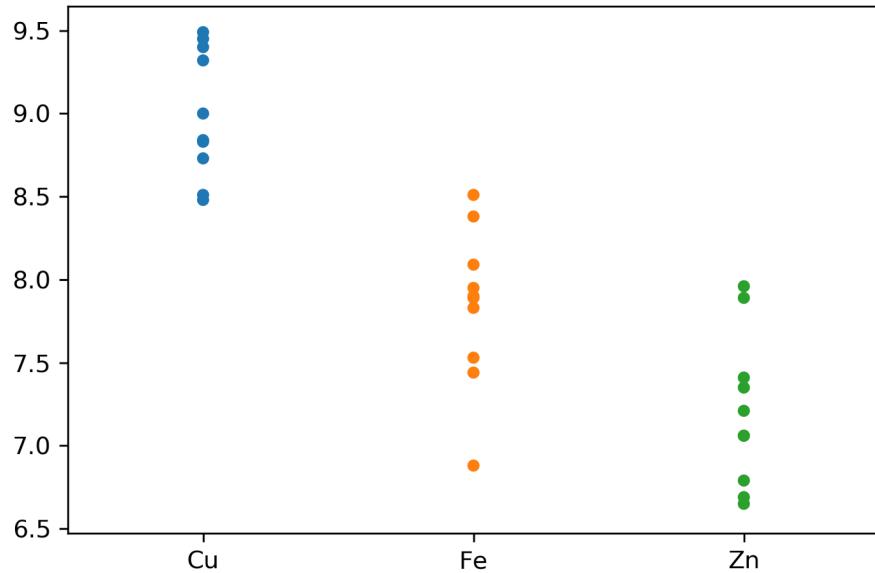
|   | Cu   | Fe   | Zn   |
|---|------|------|------|
| 0 | 8.51 | 7.95 | 6.79 |
| 1 | 9.49 | 7.53 | 7.06 |
| 2 | 8.48 | 8.09 | 7.96 |
| 3 | 9.40 | 7.44 | 7.06 |
| 4 | 8.83 | 8.38 | 6.69 |

### 10.3.1 Strip Plot

The simplest categorical plot function is `stripplot()` which generates a scatter plot with the  $x$ -axis as the categorical dimension and the  $y$ -axis as the continuous value dimension. By providing the function with the DataFrame, it will assume the columns are the categories.

```
[in]: sns.stripplot(data=df, columns=labels)
```

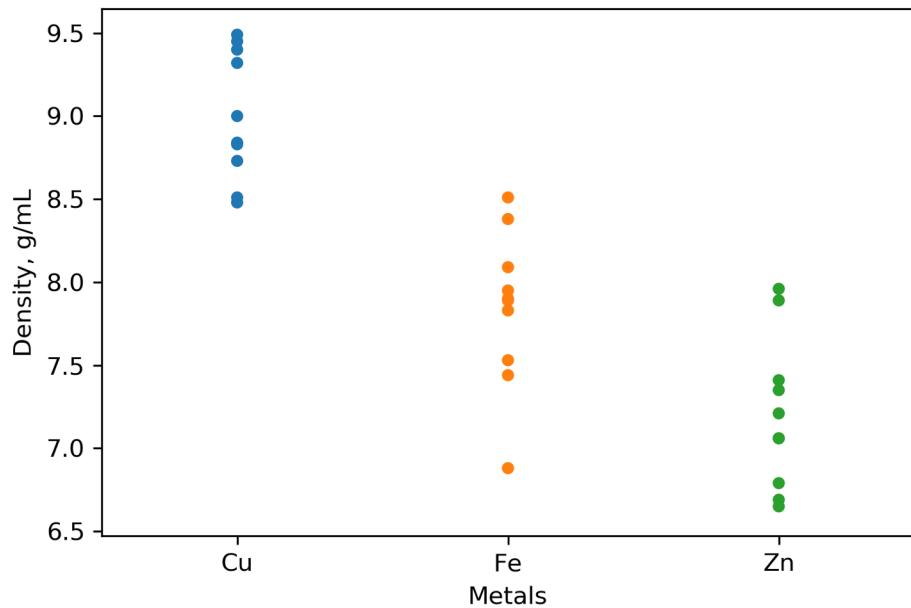
```
[out]:
```



By default, the *x*-axis contains the column labels from the DataFrame, but the *y*-axis is without any label. Again, one of the conveniences of the seaborn library is that it is built on top of matplotlib, so any plot created by seaborn can be further modified by matplotlib commands as shown below.

```
[in]: sns.stripplot(data=df, columns=labels)
plt.ylabel('Density, g/mL')
plt.xlabel('Metals')
```

```
[out] :
```

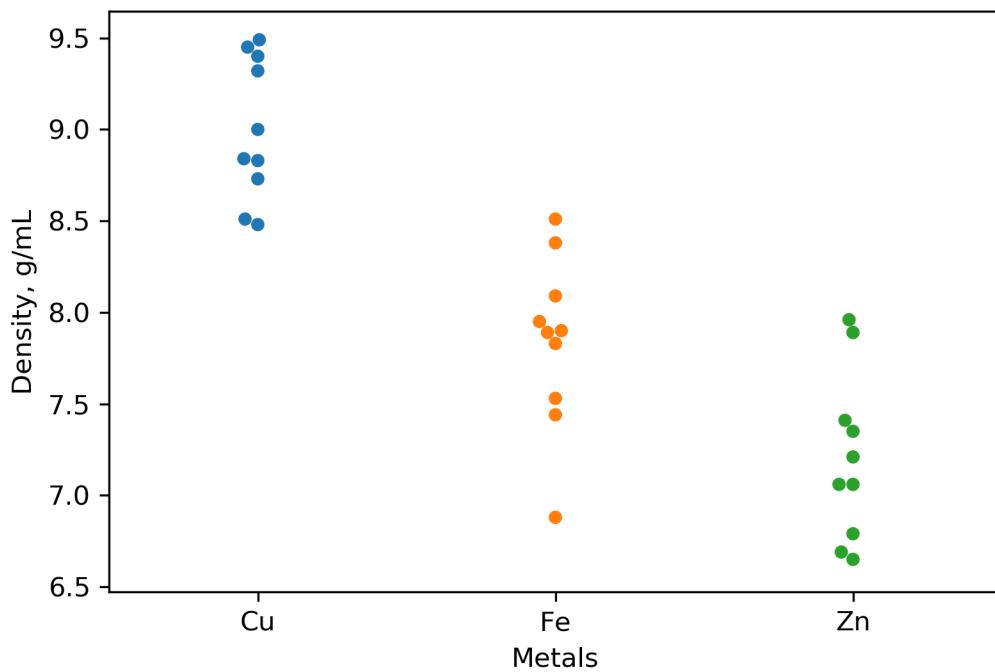


### 10.3.2 Swarm Plot

While the plots above are elegantly simple, they can make it difficult to accurately interpret the data when multiple data points are overlapping. This obscures the quantity of points in various regions. One plot that alleviates this issue is the swarm plot which is almost identical to the strip plot except that it spreads out the points horizontally to make the quantity more apparent.

```
[in]: sns.swarmplot(data=df, columns=labels)
plt.ylabel('Density, g/mL')
plt.xlabel('Metals')
```

[out]:

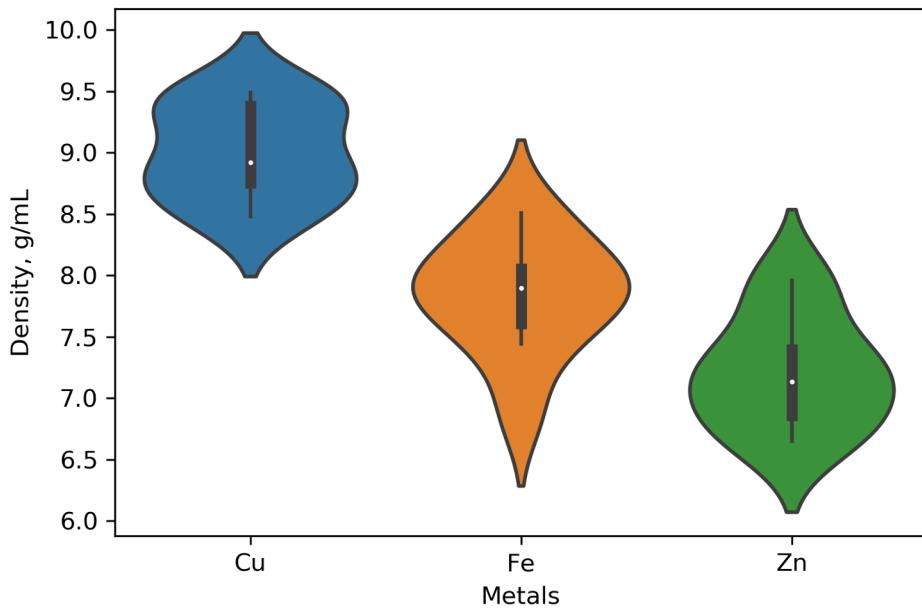


### 10.3.3 Violin Plot

An additional option for understanding the density of points is the violin plot. By default, this plot renders a blob with the width representing the density of points at various regions. Inside the blob are miniature box plots (discussed in the next section) that provide more information about the distribution of data points.

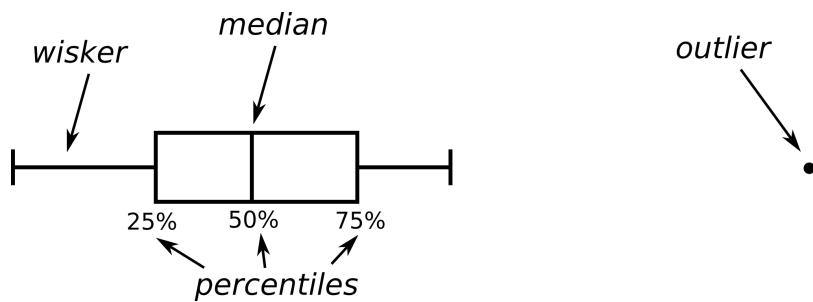
```
[in]: sns.violinplot(data=df, columns=labels)
plt.ylabel('Density, g/mL')
plt.xlabel('Metals')
```

[out] :



#### 10.3.4 Box Plot

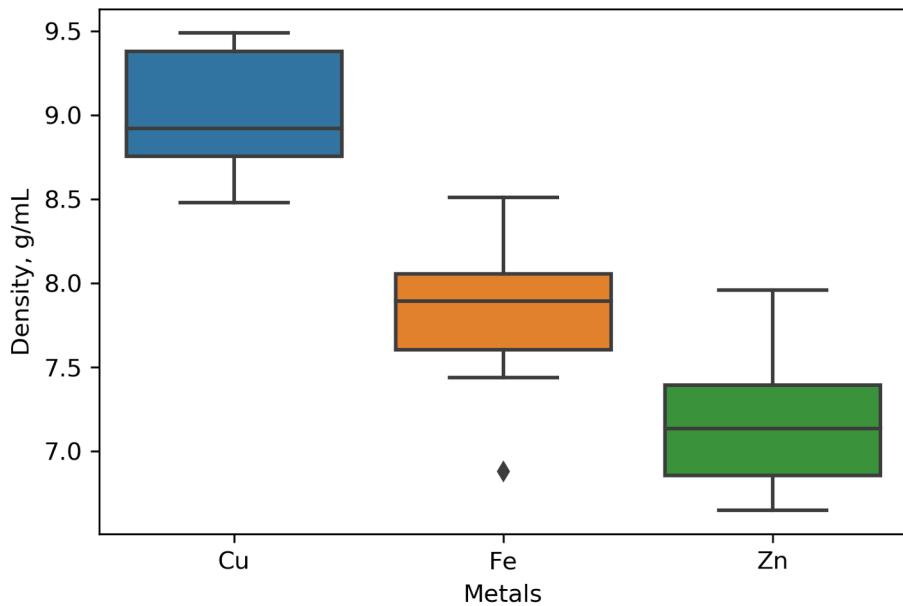
The box plot is a classic plot in statistics for representing the distribution of data and can be easily generated in seaborn using the `boxplot()` function which works much the same way as the above categorical plots. There are three main components to a box plot. The center box contains lines marking the 25, 50, and 75 percentile regions. For example, the 75 percentile line is where 75% of the data points are below. The 50 percentile is also known as the median. The length of the box (i.e., from 25 percentile to 75 percentile) is known as the *inner quartile range (IQR)*. Beyond the box are the bars known as *whiskers* which mark the range of the rest of the data points up to 1.5x the IQR. If a data point is beyond 1.5x the IQR, it is an *outlier* and is explicitly represented with a spot (Figure 10.1)



**Figure 10.1** A box plot is composed of a box with lines at the 25<sup>th</sup>, 50<sup>th</sup>, and 75<sup>th</sup> percentiles and whiskers that extend out to the rest of the non-outlier data points. If a data point is greater than 1.5× the inner quartile range from the 25<sup>th</sup> or 75<sup>th</sup> percentiles, it is an outlier represented by a dot.

```
[in]: sns.boxplot(data=df, columns=labels)
    plt.ylabel('Density, g/mL')
    plt.xlabel('Metals')
```

```
[out] :
```



### 10.3.5 Count Plot

The count plot represents the frequency of values for different categories. This is very similar to a histogram plot except that a histogram's *x*-axis is a continuous set of values while a count plot's *x*-axis is made up of discrete categories. The `countplot()` function accepts a raw collection of responses, tallies them up, and plots them as a labeled bar plot. For example, if we have a dataset of all the chemical elements up to rutherfordium (Rf) and their physical state under standard conditions, the function accepts the list of their physical states, counts them, and generates the plot.

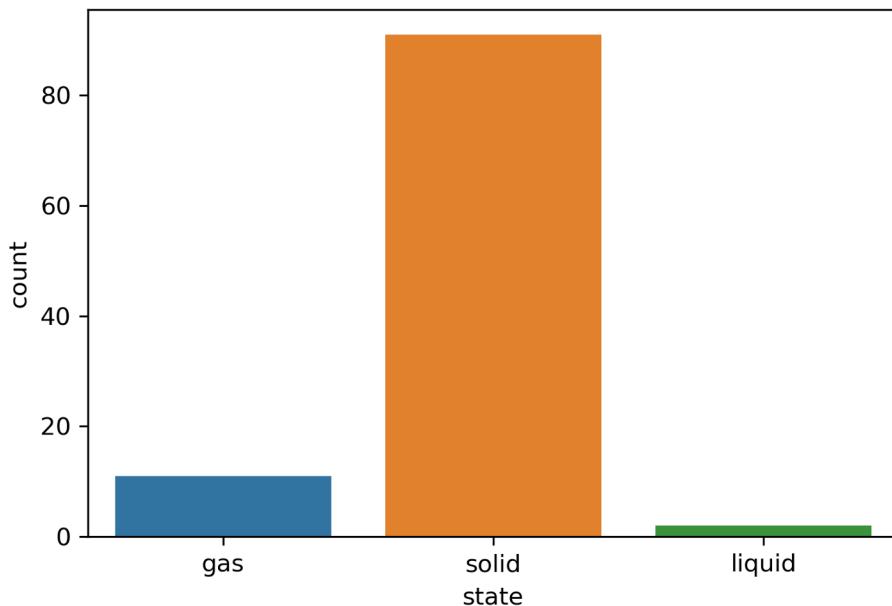
```
[in]: elem = pd.read_csv('elements_data.csv')
      elem.head()
```

```
[out] :
```

| <b>symbol</b> | <b>AN</b> | <b>row</b> | <b>block</b> | <b>state</b> |       |
|---------------|-----------|------------|--------------|--------------|-------|
| <b>0</b>      | H         | 1          | 1            | s            | gas   |
| <b>1</b>      | He        | 2          | 1            | s            | gas   |
| <b>2</b>      | Li        | 3          | 2            | s            | solid |
| <b>3</b>      | Be        | 4          | 2            | s            | solid |
| <b>4</b>      | B         | 5          | 2            | p            | solid |

```
[in]: sns.countplot(x='state', data=elem)
```

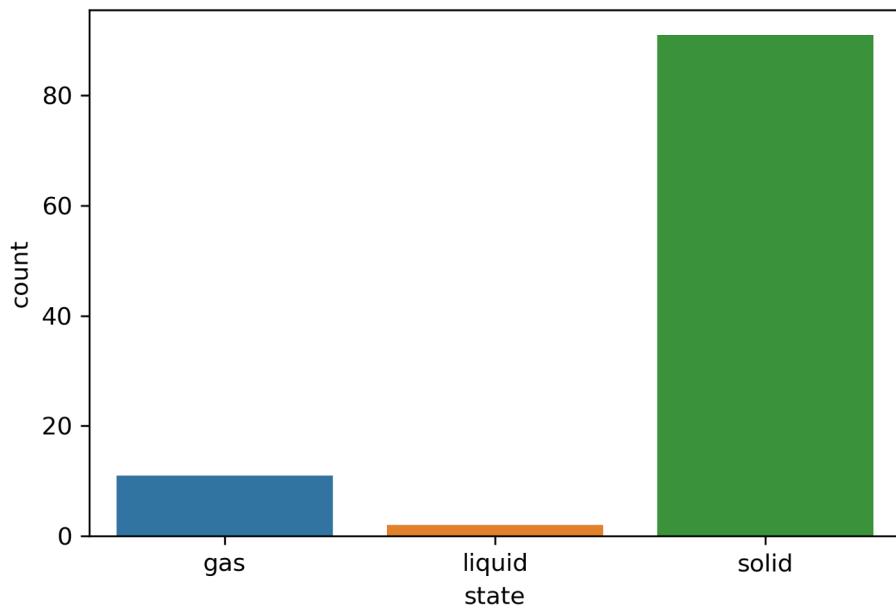
[out]:



Like many plotting types in seaborn, the count plot can be further customized through keyword arguments and using other available data. One shortcoming of the above plot is that the states are listed in the order they first appear in the dataset instead of based on disorder. We can assert a different order by providing the `order` argument as a list of how the states should appear.

```
[in]: sns.countplot(x='state', data=elem, order=['gas',
   'liquid', 'solid'])
```

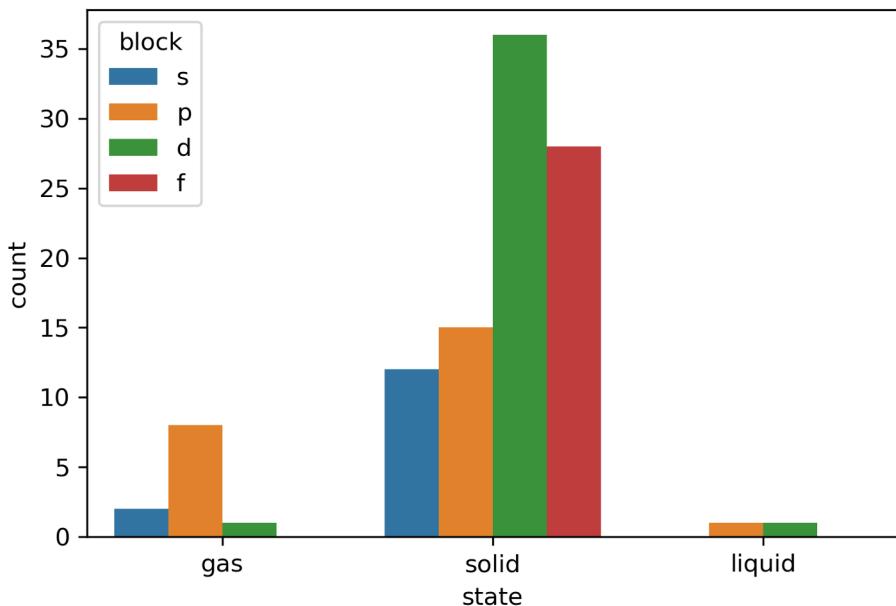
[out]:



We can also set the color of each bar based on the valence orbital block by providing the `hue` argument with the name of the column.

```
[in]: sns.countplot(x='state', hue='block', data=elem)
```

```
[out] :
```



## 10.4 Distribution Plots

Seaborn provides a set of plotting types that represent the distribution of data. These are essentially extensions of the histogram plot but with extra features like additional dimensions, kernel density estimates, and generating grids of histogram plots.

### 10.4.1 *dist Plot*

The `dist` plot is one of the most basic distribution plotting functions in seaborn. It is almost just a histogram plotting function in seaborn except that it also fits the histogram with a kernel density estimates (kde) curve. This curve is basically just a smoothed curve over the data to help visualize the overall trend.

To demonstrate this, we will use the results of a one dimensional stochastic diffusion simulation. During the individual steps of this simulation, each of a thousand simulated molecules are either moved to the right one unit, to the left one unit, or not moved at all. A random number generator dictates this movement as demonstrated below.<sup>88</sup>

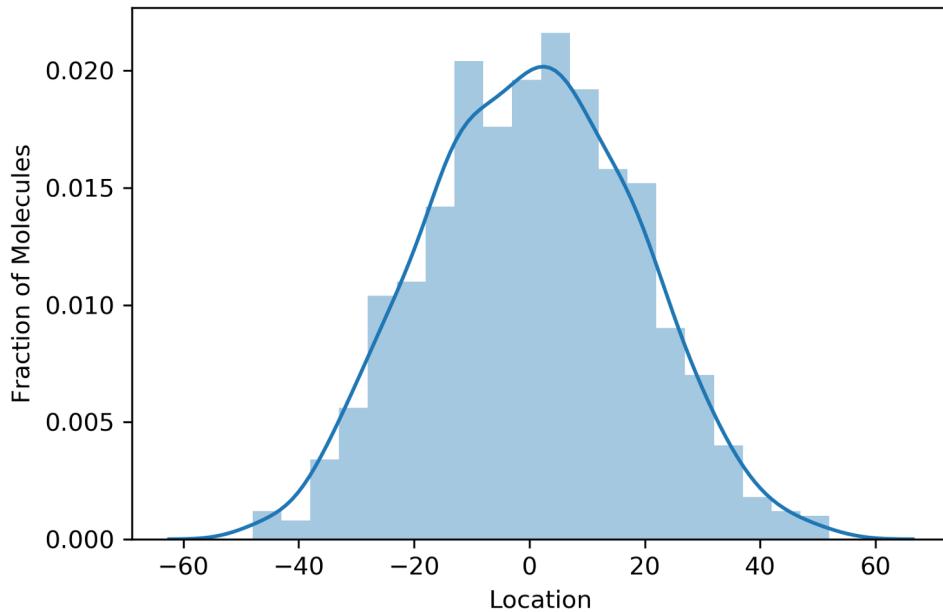
```
[in]: loc = np.zeros(1000)    # locations of molecules
      for step in range(1000):
          loc += np.random.randint(-1, high=2, size=1000)

[in]: sns.distplot(loc)
      plt.xlabel('Location')
      plt.ylabel('Fraction of Molecules')

[out]:
```

---

<sup>88</sup> For a description of the diffusion simulation methodology, see: Weiss, C. J. “Introduction to Stochastic Simulations for Chemical and Physical Processes: Principles and Applications.” *J. Chem. Educ.* **2017**, 94 (12), 1904-1910.

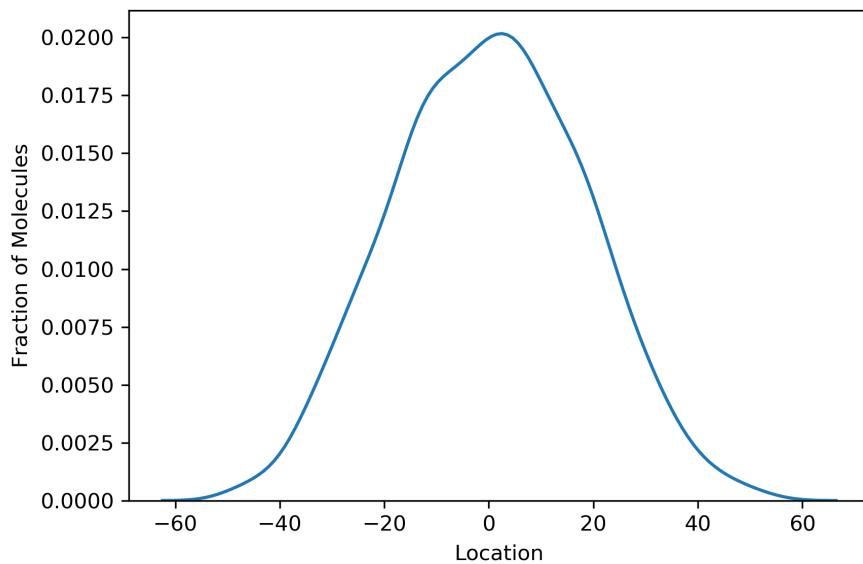


#### 10.4.2 kde Plot

The `kde` plot type is very similar to the `dist` plot except that it only shows the kernel density estimates and not the histogram itself.

```
[in]: sns.distplot(loc)
       plt.xlabel('Location')
       plt.ylabel('Fraction of Molecules')
       plt.tight_layout()
```

```
[out]:
```



### **10.4.3 jointplot (diffusion simulation)**

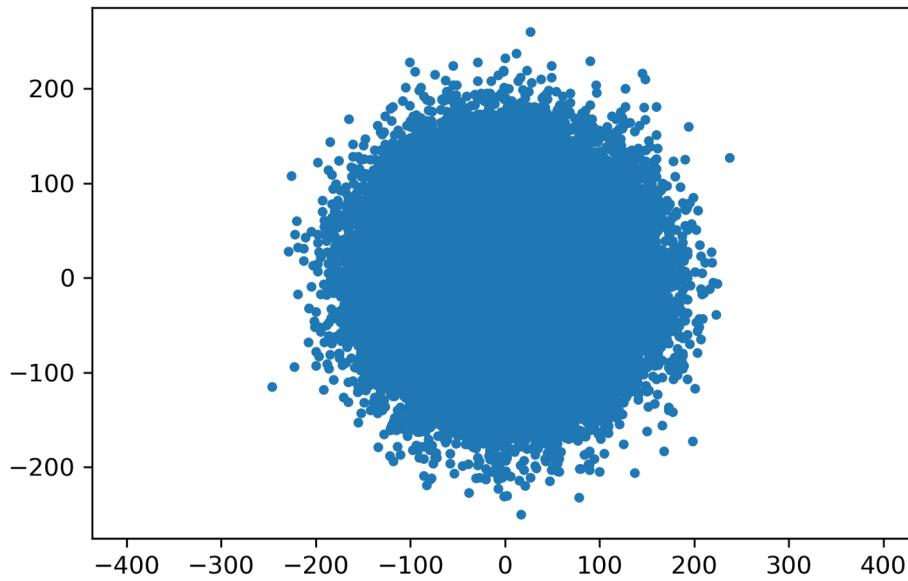
A joint plot can be described as a scatter plot with histograms on the sides providing additional information or clarification on the density of the data points. To demonstrate this, below is a two dimensional stochastic diffusion simulation<sup>88</sup> and the results. The principles are the same as above except applied to two dimensions.

```
[in]: x = np.sum(np.random.randint(-1, high=2, size=(5000,
   100000)), axis=0)
y = np.sum(np.random.randint(-1, high=2, size=(5000,
   100000)), axis=0)
```

Now to plot the results.

```
[in]: plt.plot(x, y, '.')
plt.axis('equal')

[out]:
```

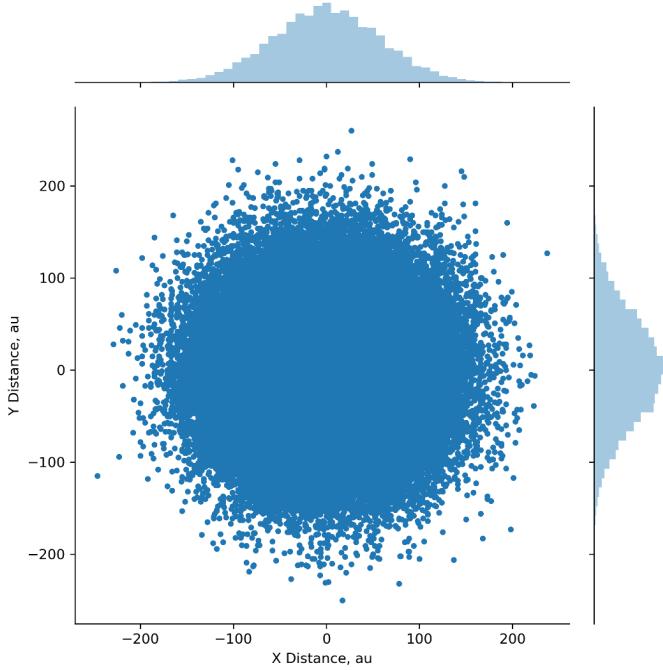


One of the issues with this plot is that there are so many data points in the plot that it is difficult to determine the distribution/density inside the blanket of solid dots. The seaborn joint plot adds histograms to the side to help the viewer recognize where most of the data points reside.

The joint plot function, `sns.jointplot()`, takes two required arguments of the *x* and *y* variables. While this function does not require the use of pandas or a DataFrame, it is convenient because the axis labels are pulled directly from the column headers.

```
[in]: df = pd.DataFrame(data={'X Distance', 'au': loc_x, 'Y
                               Distance, au':
                               loc_y})
      sns.jointplot(df['X Distance, au'], df['Y Distance,
                               au'], size=7, color='C0',
                               stat_func=None,
                               joint_kws={'s':10})
      plt.tight_layout()

[out]:
```

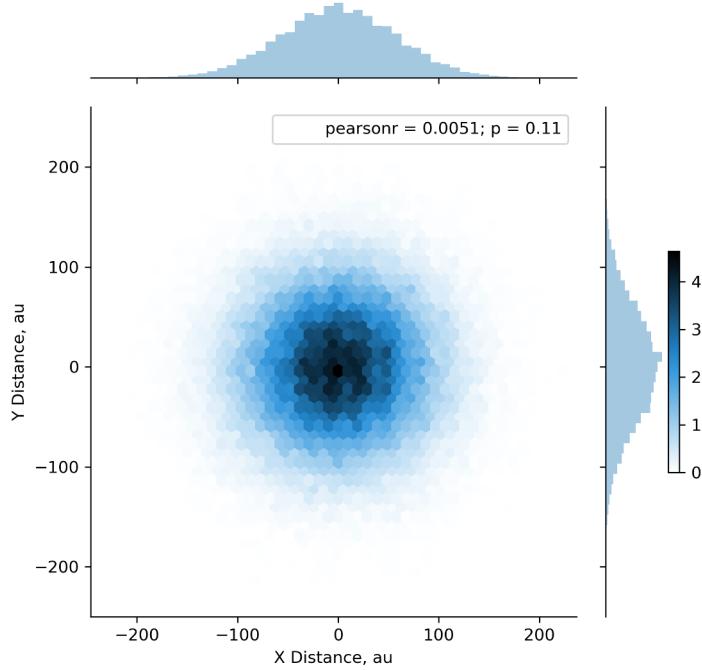


There are numerous arguments to fine tune the joint plot. For example, the joint plot does not need to be a scatter plot with histograms. The density of the data points can be represented with hexagonal patches or kernel *density estimates* (kde). The latter represents the density of points through contours and is a reoccurring option in other plotting functions in the seaborn library. It is worth noting that the kde plotting types take a little time to calculate, so expect a brief delay in generating these plots.

Below, the hexagonal joint plot was generated with a color bar on the side using the `matplotlib plt.colorbar()` command. As with all seaborn plots, the plot can be customized using `matplotlib` commands.

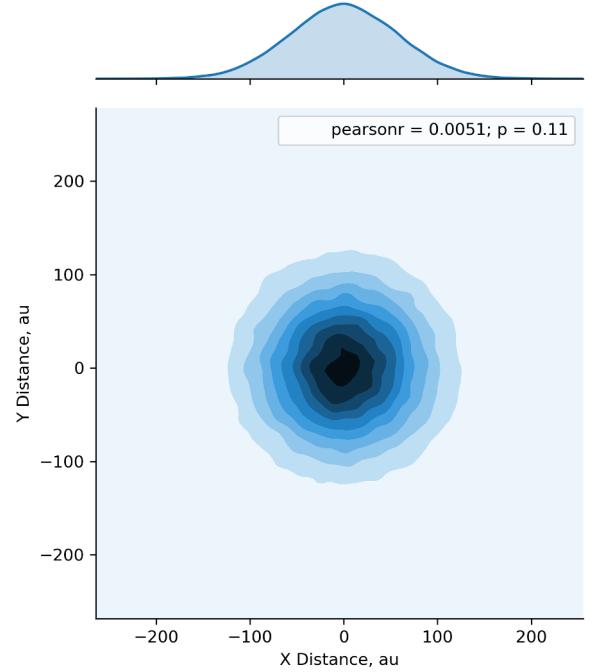
```
[in]: sns.jointplot(df['X Distance, au'], df['Y Distance, au'], kind='hex')
       plt.colorbar()

[out]:
```



```
[in]: sns.jointplot(df['X Distance, au'], df['Y Distance, au'], kind='kde')
```

```
[out] :
```



## 10.5 Pair Plot

The pair plot belongs to the category of distribution plots, but it is different enough to be worth addressing separately. A pair plot is designed to show the relationship among multiple variables by generating a grid of plots in a single figure. Each plot in the grid is a scatter plot showing the relationship between two of the variables on either axis with the exception of the plots in the diagonals. Because the diagonal plots are the intersection between a variable and itself, these are histograms showing the distributions of values for that variable. Pair plots are particularly useful for looking at new data to see if there are any trends worth investigating because this entire grid can be easily generated with a single `sns.pairplot()` function.

To demonstrate a pair plot, the file `periodic_trends.csv` contains physical data on non-noble gas elements in the first three rows of the periodic table. To quickly see how each of the columns of data relate to each other, we will generate a pair plot.

```
[in]: per = pd.read_csv('periodic_trends.csv')
per.head()
```

[out]:

|          | <b>symbol</b> | <b>AN</b> | <b>row</b> | <b>IE_kJ</b> | <b>radius_pm</b> |
|----------|---------------|-----------|------------|--------------|------------------|
| <b>0</b> | H             | 1         | 1          | 1310         | 38               |
| <b>1</b> | Li            | 3         | 1          | 520          | 34               |
| <b>2</b> | Be            | 4         | 2          | 900          | 90               |
| <b>3</b> | B             | 5         | 2          | 800          | 82               |
| <b>4</b> | C             | 6         | 2          | 1090         | 77               |

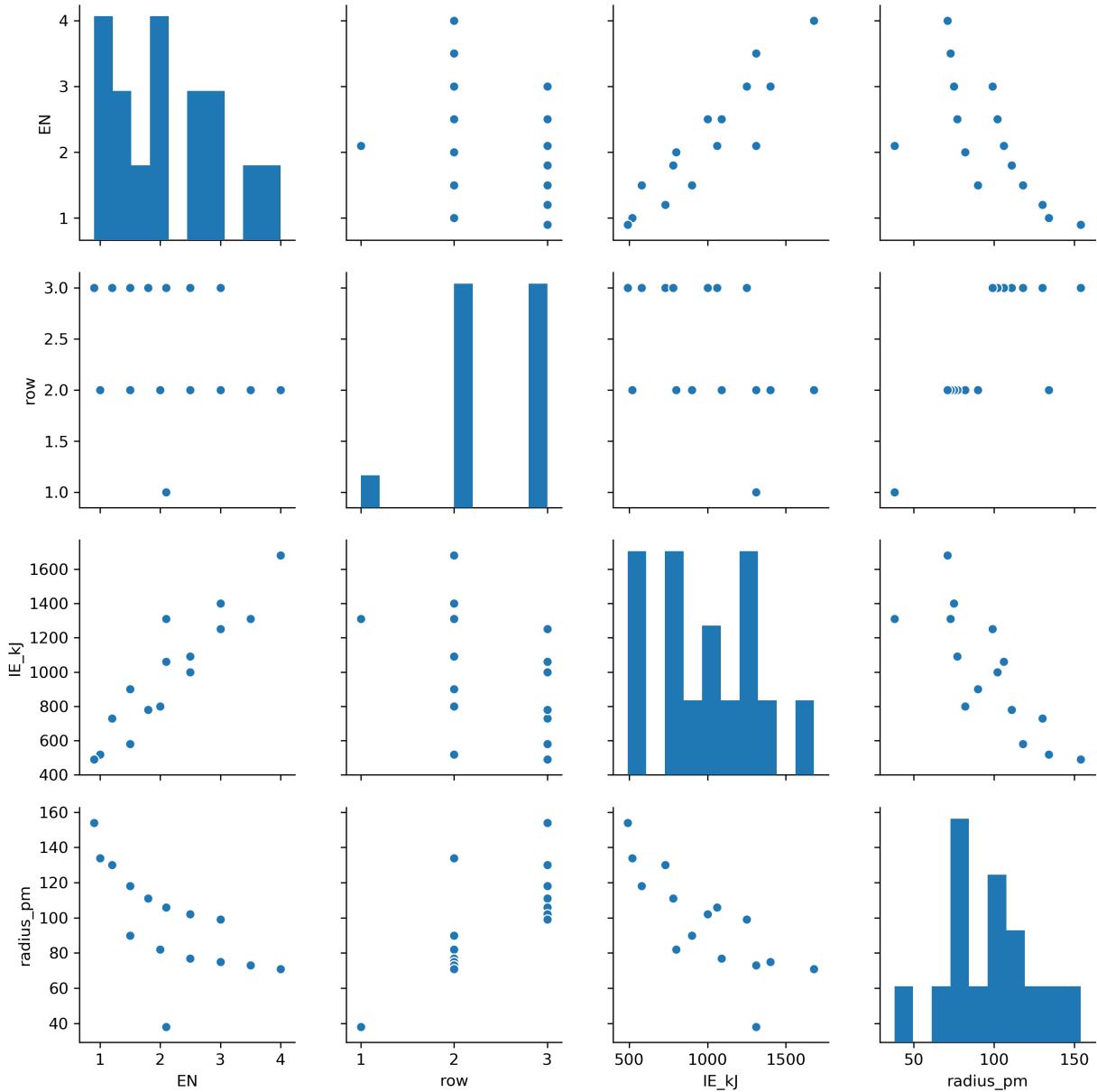
```
[in]: per.drop(['AN', 'symbol'], axis=1, inplace=True)
per.head()
```

[out]:

|          | <b>row</b> | <b>IE_kJ</b> | <b>radius_pm</b> |
|----------|------------|--------------|------------------|
| <b>0</b> | 1          | 1310         | 38               |
| <b>1</b> | 1          | 520          | 34               |
| <b>2</b> | 2          | 900          | 90               |
| <b>3</b> | 2          | 800          | 82               |
| <b>4</b> | 2          | 1090         | 77               |

```
[in]: sns.pairplot(per)
```

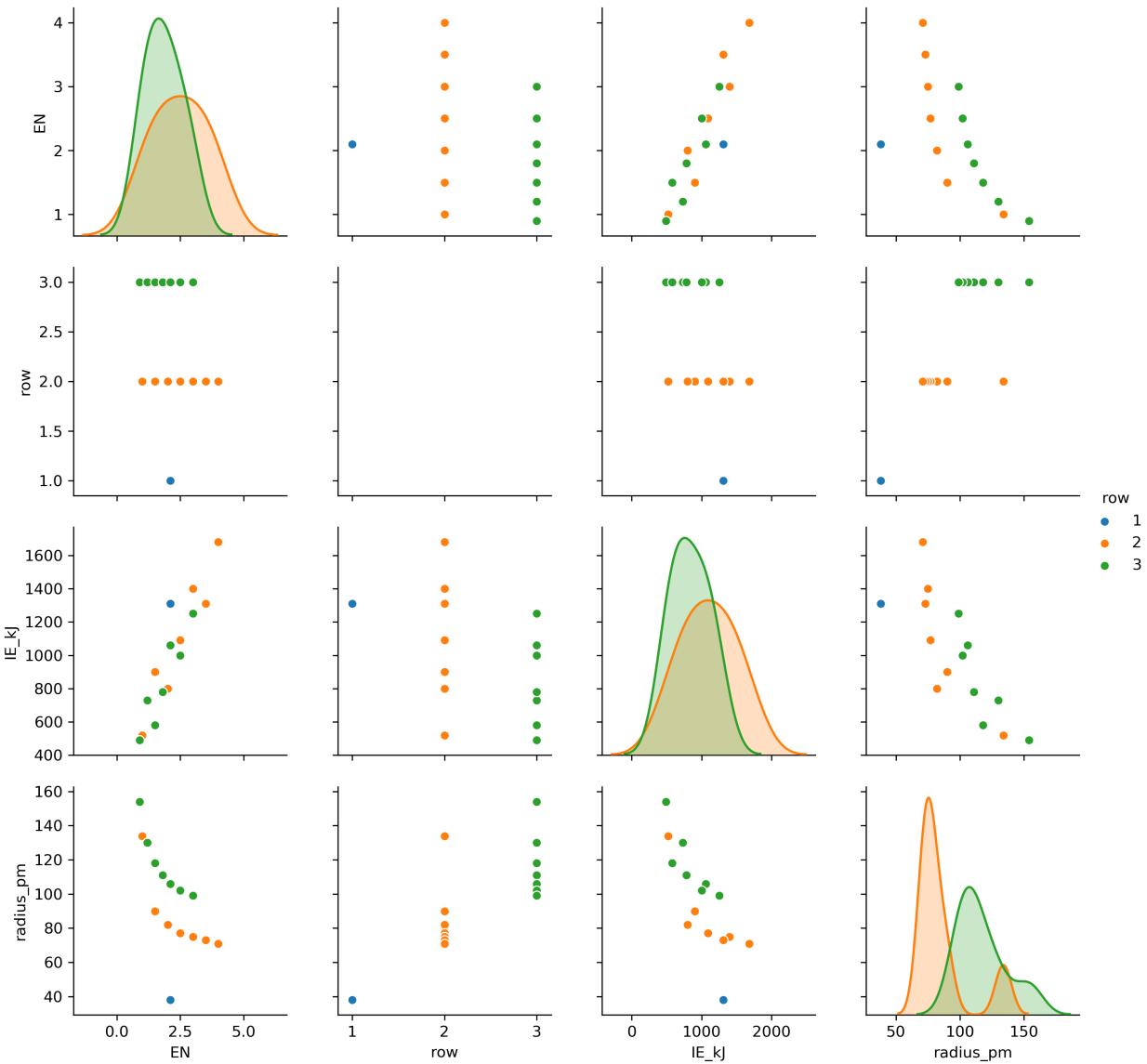
```
[out] :
```



The color can also be set based on any piece of information. Below, the row is used to dictate the color of each data point.

```
[in]: sns.pairplot(per, hue='row')
```

[out]:



## 10.6 Heat Map

Heat maps are color representations of 2D grids of numerical data and are ideal for making large tables of values easily interpretable. As an example, we can import a table of bond dissociation energies (in kJ/mol) and visualize these data as a heat map. In the following pandas function call, the `index_col=0` tells pandas to apply the first column as column headers as well.

```
[in]: bde = pd.read_csv('bond_enthalpy_kJmol.csv',
index_col=0)
      bde

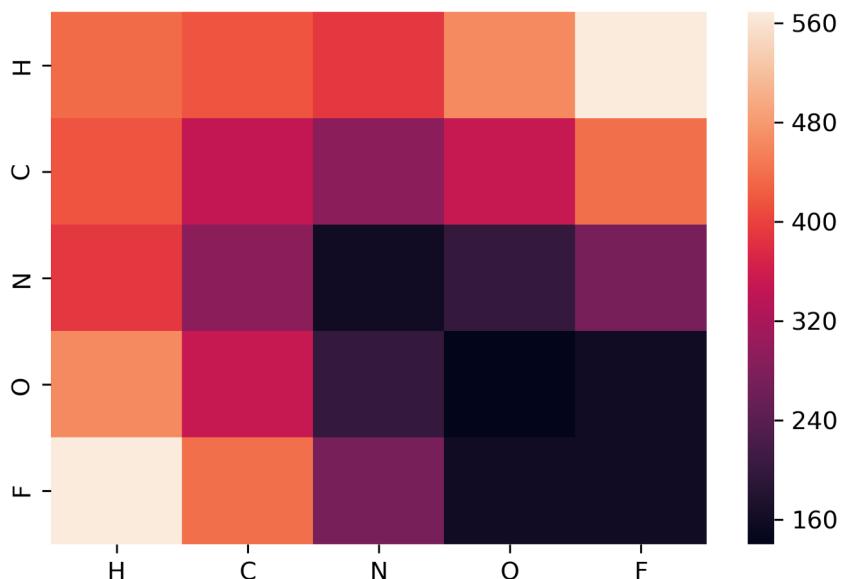
[out]:
```

|   | H   | C   | N   | O   | F   |
|---|-----|-----|-----|-----|-----|
| H | 436 | 415 | 390 | 464 | 569 |
| C | 415 | 345 | 290 | 350 | 439 |
| N | 390 | 290 | 160 | 200 | 270 |
| O | 464 | 350 | 200 | 140 | 160 |
| F | 569 | 439 | 270 | 160 | 160 |

This grid of numerical values is difficult to quickly interpret, and if it were a larger table of data, it could become almost impossible to interpret in this form. We can plot the heat map using the `heatmap()` function and feeding it the DataFrame. The function also accepts NumPy arrays, but without the index and column labels of a DataFrame, the axes will not be automatically labeled.

```
[in]: sns.heatmap(bde)
```

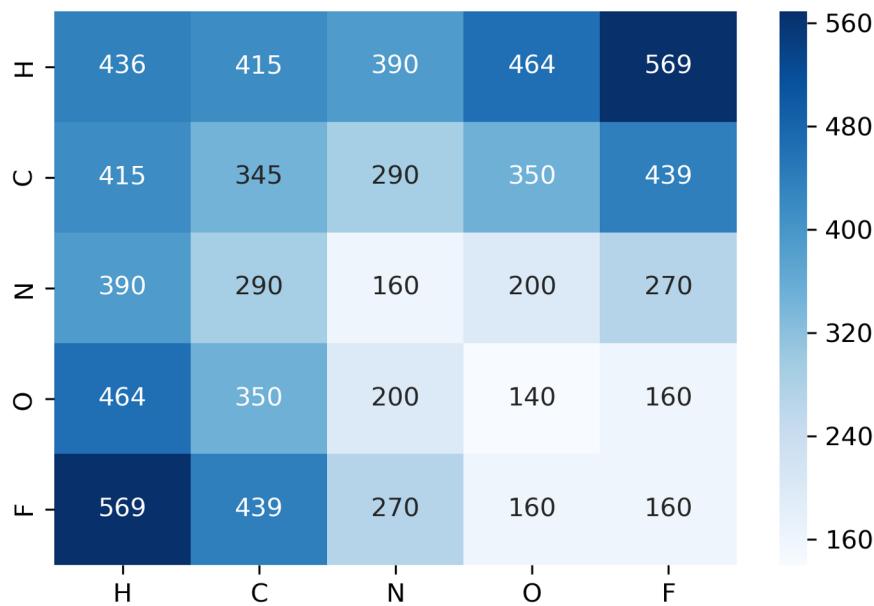
```
[out]:
```



Now we have a color grid where the colors represent numerical values defined in a colorbar automatically displayed on the righthand side. This default color map can easily be customized through various arguments in the `heatmap()` function. One nice addition is to display the numerical values on the heat map by setting `annot=True`. If you choose to annotate the rectangles, you may need to use the `fmt=` parameter to dictate the format of the annotation labels. Some common formats are `d` for decimal, `f` for floating point, and `.2f` gives two places after the decimal place in a floating point number. If you want a different color map, this can be set using the `cmap` argument and any matplotlib colormap you want. Below, the annotation is turned on with the perceptually uniform viridis colormap.

```
[in]: sns.heatmap(bde, annot=True, fmt='d', cmap='Blues')
```

```
[out]:
```



## 10.7 Relational Plots

Relational plots are a new addition to the seaborn library as of version 0.9 and include seaborn's functions for scatter and line plots. Of course, matplotlib does a nice job making scatter and line plots reasonably easy, but seaborn offers a few extra ease-of-use improvements upon matplotlib that may be worth something to you depending upon your needs.

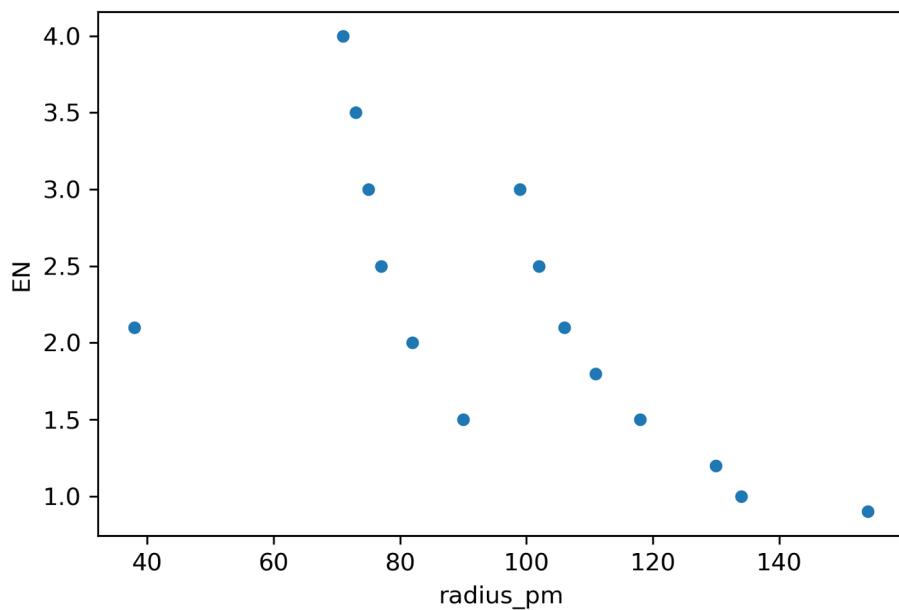
### 10.7.1 Scatter Plots

One difference between seaborn and matplotlib in generating scatter and line plots is that seaborn allows the user to change the color, size, and marker styles of individual markers based on numerical values or text data. Matplotlib can also change the color and size of the markers but only based on numerical values, and to change the marker style, the `plt.scatter()` function needs to be called a second time. Seaborn allows this whole process in a single function call.

Below, we are using the periodic trends data (`per`) imported in section 10.5. We can start with plotting the electronegativity (`EN`) versus the atomic radius (`radius_pm`) using the `sns.scatterplot()` function which takes many of the same basic arguments as plots we have seen so far with seaborn.

```
[in]: sns.scatterplot(x='radius_pm', y='EN', data=per)
```

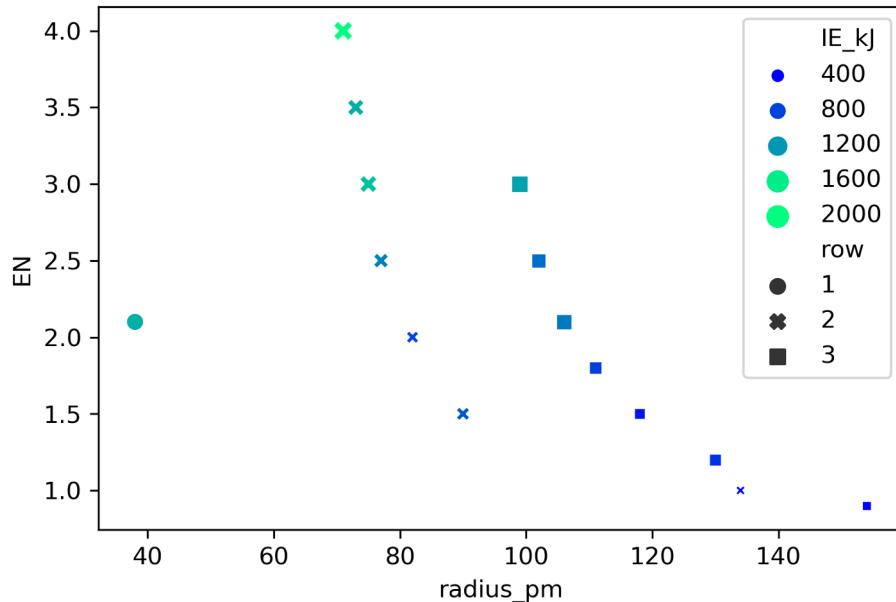
```
[out] :
```



To modify the color, size, and marker style of the data points, use the `hue`, `size`, and `marker` arguments. This allows additional information to be infused into a single plot. Note that the legend automatically appears on the plot. In addition, the colormap for the plot can be modified using the `palette` keyword argument and the name of any matplotlib colormap.

```
[in]: sns.scatterplot(x='radius_pm', y='EN', data=per,
                     hue='IE_kJ',
                     size='IE_kJ', style='row',
                     palette='winter')

[out]:
```



## 10.7.2 Line Plots

The `lineplot()` function in seaborn is somewhat similar to the `plt.plot()` function in matplotlib except it also includes a number of extra features similar to those seen in other seaborn plotting functions. This includes the ability to change the plotting color and style based on additional information, easy visualization of confidence intervals, automatic generation of a legend, and others. To demonstrate the `lineplot()` function, we will import simulated kinetic data for a first-order chemical reaction run repeatedly.

```
[in]: kinetics = pd.read_csv('kinetic_runs.csv')
kinetics.head()

[out]:
```

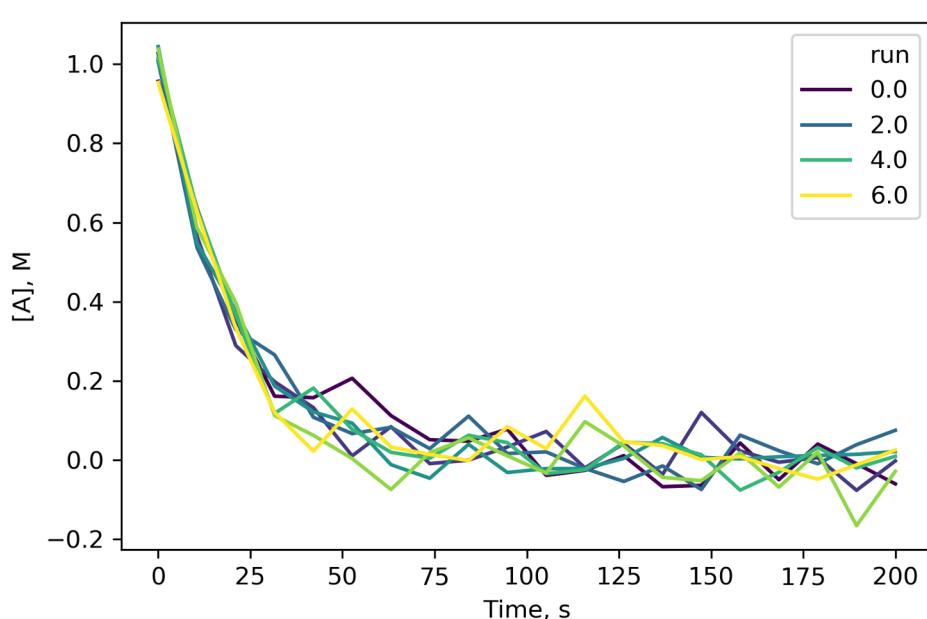
|   | Time      | [A]      | Run | [P]      |
|---|-----------|----------|-----|----------|
| 0 | 0.000000  | 0.956279 | 0.0 | 0.043721 |
| 1 | 10.526316 | 0.636978 | 0.0 | 0.363022 |

|          |           |          |     |          |
|----------|-----------|----------|-----|----------|
| <b>2</b> | 21.052632 | 0.355690 | 0.0 | 0.644310 |
| <b>3</b> | 31.578947 | 0.161173 | 0.0 | 0.838827 |
| <b>4</b> | 42.105263 | 0.157420 | 0.0 | 0.842580 |

The [A] and [P] columns contain the molar concentrations of reactant and product for the reaction  $A \rightarrow B$  while Time and Run provide the time in seconds and experiment number.

```
[in]: sns.lineplot(x='time', y='[A]', data=kinetics,
                  hue='run',
                  palette='viridis')
      plt.xlabel('Time, s')
      plt.ylabel('[A], M')
```

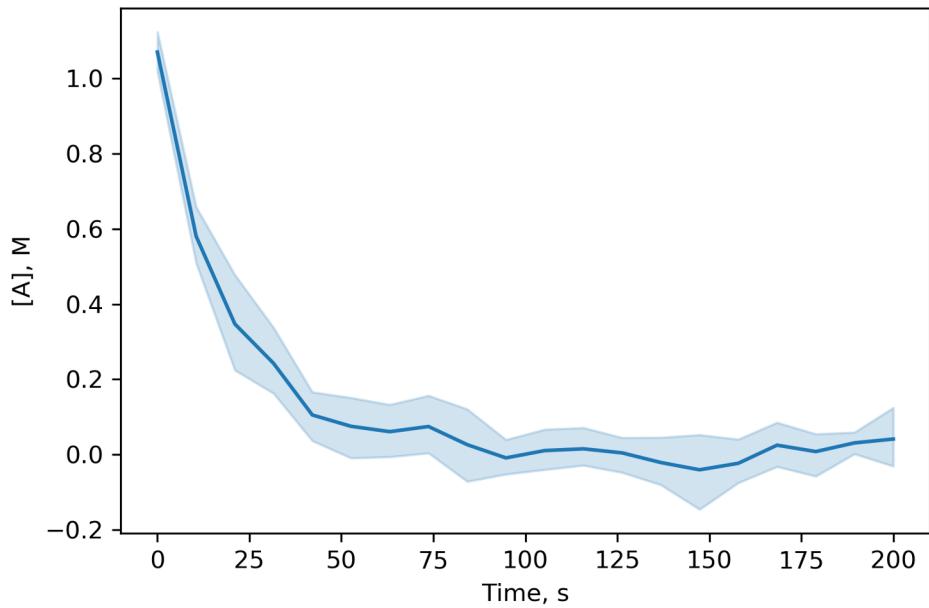
[out] :



The [A] was plotted versus Time, and the hue of each line was set to the Run number. The result is that each kinetic run is shown in a separate color. If the user is not concerned so much with seeing the individual runs but instead wants to see an average of each run, the lineplot() function provides a default 95% confidence interval as is shown below.

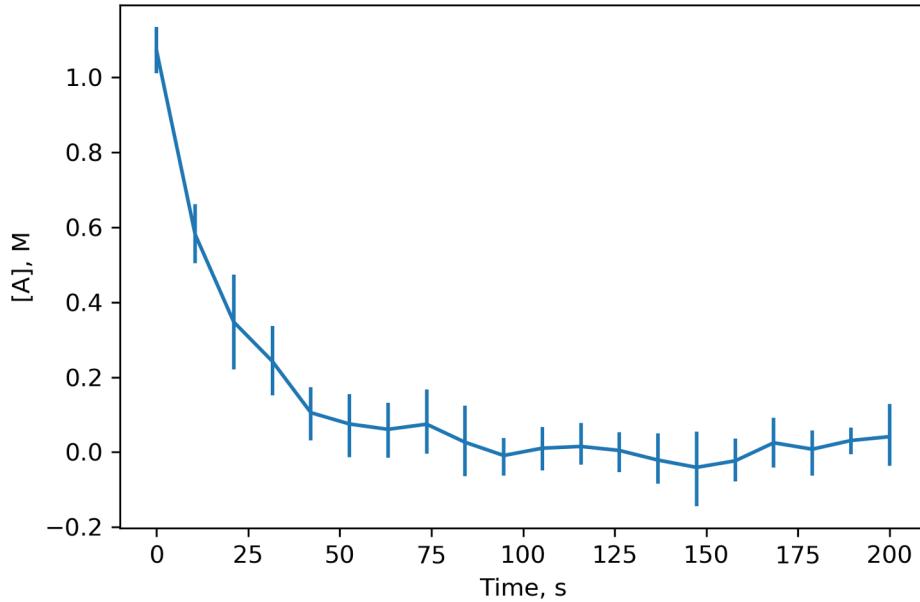
```
[in]: sns.lineplot(x='time', y='[A]', data=kinetics)
      plt.xlabel('Time, s')
      plt.ylabel('[A], M')
```

[out] :



A confidence interval is only shown if there are multiple data points for each time. The confidence intervals can also be represented with error bars by setting `err_style = 'bars'`.

```
[in]: sns.lineplot(x='time', y='[A]', data=kinetics,  
err_style='bars')  
      plt.xlabel('Time, s')  
      plt.ylabel('[A], M')  
  
[out]:
```



## 10.8 Internal Data Sets

Similar to a number of other Python libraries, seaborn brings with it data sets for users to experiment with. These are callable using the `sns.load_datasets()` function with the name of the data set as the argument. Below is a table describing a few of the available Seaborn data sets. This list may change, so you can use the `sns.get_dataset_names()` to see the most current list.

**Table 10.3** A Few Data Sets Available in Seaborn

| Name                     | Description                                                                                                                                                                   |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>anscombe</code>    | Anscombe's quartet <sup>89</sup> data with four artificial datasets that exhibit the same mean, standard deviation, and linear regression among other statistical descriptors |
| <code>car_crashes</code> | Data on car crashes including mph above the speed limit among other information                                                                                               |
| <code>exercise</code>    | Diet and exercise data                                                                                                                                                        |
| <code>flights</code>     | Aircraft flight information including year, month, and number of passengers                                                                                                   |
| <code>iris</code>        | Ronald Fisher's famous iris dataset <sup>90</sup> used frequently in machine learning classification examples                                                                 |

<sup>89</sup> Anscombe, F. J. "Graphs in Statistical Analysis". *Amer. Statist.* **1973**, 27(1), 17–21.

|         |                                                                                    |
|---------|------------------------------------------------------------------------------------|
| planets | Information on discovered planets                                                  |
| tips    | Restaurant information including bill total, tip, and information about the client |
| titanic | Titanic survivor dataset                                                           |

---

90 Fisher, R. A. "The use of multiple measurements in taxonomic problems". *Ann. Eugen.* **1936**, 7(2), 179–188.

## Further Reading

1. Seaborn Website. <https://seaborn.pydata.org/>

## Exercises

1. Import the file *linear\_data.csv* and visualize it using a regression plot.
2. Import the file titled *ir\_carbonyl.csv* and visualize the carbonyl stretching frequencies using a seaborn categorical plot. Represent the different molecules with different colors.
3. Import the file titled *ir\_carbonyl.csv* containing carbonyl stretches of ketones and aldehydes.<sup>91</sup>
  - a) Separate the ketones and aldehydes values into individual Series.
  - b) Visualize the distribution of both ketone and aldehyde carbonyl stretches using `sns.kdeplot()`.
4. Import the *elements\_data.csv* file and generate a count plots showing the number of elements in each block of the periodic table (i.e., s, p, d, f).
5. The following equation is Plank's law which describes the relationship between the radiation intensity( $M$ )<sup>92</sup> with respect to wavelength ( $\lambda$ ) and temperature (T).
$$M = \frac{2\pi hc^2}{\lambda^5(e^{(hc/\lambda kT)} - 1)}$$

Import the data called *blackbody.csv* containing intensities at various temperatures and wavelengths based on Plank's law. Generate a plot of intensity versus wavelength using the `lineplot()` function, and display the different temperatures as different colors.

6. Import the file *ionization\_energies.csv* showing the first four ionization energies for a number of elements. Plot this grid of data as a heat map. Include labels in each cell using the `annot=` argument.
7. Import the file *ROH\_data\_small.csv* and plot visualize how boiling point (bp), molecular weight (MW), degree, and whether a compound is aliphatic are correlated using a pairplot.

<sup>91</sup> Values from: National Institute of Advanced Industrial Science and Technology (AIST). <https://sdbs.db.aist.go.jp> (accessed August 2019)

<sup>92</sup> More precisely it is *excitance*.

# *Chapter 11*

## ***Nuclear Magnetic Resonance with NMRglue***

Nuclear magnetic resonance (NMR) spectroscopy is one of the most common and powerful analytical methods used in modern chemistry. Up to this point, we having been primarily dealing with text-based data files – that is, files that can be opened with a text editor and still contain human comprehensible information. If you open most files that come out of an NMR instrument in a text editor, it will look more like gibberish than anything a human should be able to read. This is because they are *binary* files; they are written in computer language rather than human language.

We need a specialized module to be able to import and read these data, and luckily a Python module called NMRglue to does exactly this. The module contains submodules for dealing with data from each of the major NMR spectroscopy file types which includes Bruker, Pipe, Sparky, and Varian. It does not read JOEL files, but as of this writing, JOEL spectrometers support exporting data into at least one of the above file types supported by NMRglue.

Currently, NMRglue is not included with Anaconda, so you will need to fetch NMRglue and install it yourself. Instructions are included on the following website. NMRglue requires you have NumPy and SciPy already installed, and matplotlib should also be installed for visualization.

<https://nmrglue.readthedocs.io/en/latest/install.html>

All use of NMRglue below assumes the following import with alias. NMRglue is not a major library in the SciPy ecosystem, so the `ng` alias is not a strong convention but is used here for convenience and to be consistent with the online documentation.<sup>93</sup>

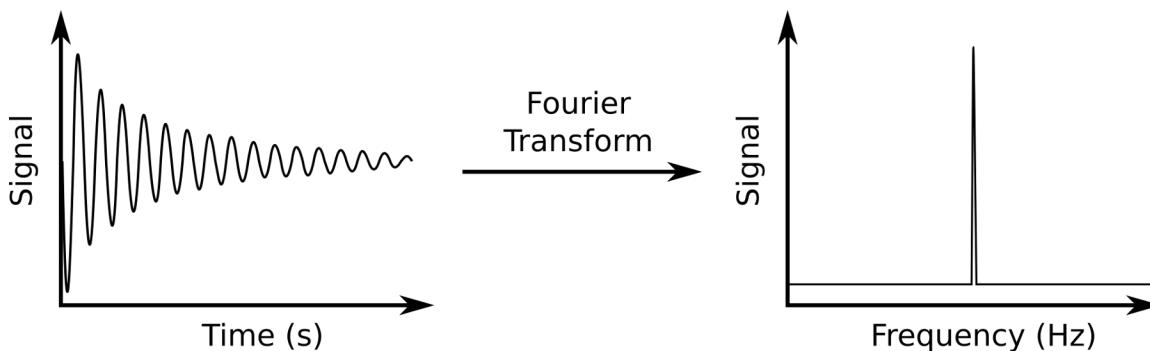
---

<sup>93</sup> See Further Reading at the end of this chapter for a link to the NMRglue documentation.

```
[in]: import nmrglue as ng  
import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline
```

## 11.1 NMR Processing

The general procedure for collecting NMR data is to excite a given type of NMR-active nuclei with a radio-frequency pulse and allow them to relax. As they precess, their rotation leads to a voltage oscillation in the instrument at characteristic frequencies, and the spectrometer records these oscillations as a *free induction decay (fid)* depicted below (Figure 11.1, left). It is the frequency of these oscillations that we are interested in because they are informative to a trained chemist as to the chemical environment of the nuclei. One challenge is that all the different signals from each of the nuclei are stacked on top of each other making it difficult to distinguish one from the other or determine the wave frequency. This is similar to the problem of a computer discerning a single instrument in an entire orchestra playing at once. Fortunately, there is a mathematical equation called the Fourier transform that converts the above *fid* into a graph showing all of the different frequencies (Figure 11.1, right). This is what is known as converting the *time domain* to the *frequency domain*.



**Figure 11.1** Raw NMR spectroscopy data is converted from the time domain (left) to the frequency domains (right) using a Fourier transform.

The general steps for dealing with NMR spectroscopic data in Python are outlined below.

1. Load the fid data into a NumPy array using NMRglue
2. Fourier transform the data to the frequency domain
3. Phase the spectrum
4. Reference the spectrum

5. Measure the chemical shifts and integrals of the peaks

## 11.2 Importing Data with NMRglue

The importing of data using NMRglue is performed by the `read` function from one of four submodules shown below. The choice of module is dictated by the data file type.

**Table 11.1** Four NMRglue Modules

| Module | Description                                                    |
|--------|----------------------------------------------------------------|
| bruker | Bruker data as a single file                                   |
| pipe   | Pipe data as a single file with an .fid extension              |
| sparky | Sparky NMR file format with .ucsf extension                    |
| varian | Varian/Agilent data as a folder of data with an .fid extension |

The `read` function loads the NMR file and returns a tuple containing a dictionary of metadata<sup>94</sup> and data in an ndarray. The dictionary includes information required to complete the processing of the NMR date. Looking at the NMR data shown below, you may noticed each point includes both both real and imaginary components (i.e., the mathematical terms with ' $j$ '). Both are necessary for phasing the spectrum later on, so don't discard any of this data.

```
[in]: dic, data = ng.pipe.read('EtPh_1H_NMR_CDCl3.fid')

[out]: data
array([-0.00175312-0.00272448j, -0.00165303-0.00273172j,
       -0.00161139-0.00266785j, ..., -0.00172259+0.00730833j,
       -0.00175674+0.00729703j, -0.00166400+0.0072923j ],
      dtype=complex64)
```

The dictionary contains a very long list of values, and the dictionary keys can be different among different file formats. To maintain a shorter, more useful, and more consistent dictionary of metadata, NMRglue provides the `guess_udic` function for generating a *universal dictionary* among all file formats.

```
[in]: udic = ng.pipe.guess_udic(dic, data)
```

---

<sup>94</sup> Metadata is information about the data. In this case, the dictionary includes information such as the nucleus, spectrometer field strength, instrument temperature, data, resolution, carrier frequency, etc...

```

udic

[out]:
{'ndim': 1,
 0: {'car': 1998.9109802246094,
 'complex': True,
 'encoding': 'direct',
 'freq': True,
 'label': 'Proton',
 'obs': 399.7821960449219,
 'size': 13107,
 'sw': 5994.65478515625,
 'time': False} }

```

The universal dictionary is a nested dictionary. The first key is `ndim` which provides the number of dimensions in the NMR spectrum. Most NMR spectra are one dimensional, but two dimensional is also fairly common. Subsequent key(s) are for each dimension in the NMR spectrum with the value as a nested dictionary of metadata. Because the data for the above spectrum is one dimensional, there is only one nested dictionary. Table 11.2 below provides a description of each piece of metadata contained in the universal dictionary.

**Table 11.2** udic Dictionary Keys for Single Dimensions\*

| Key                   | Description                                       | Data Type |
|-----------------------|---------------------------------------------------|-----------|
| <code>car</code>      | Carrier frequency (Hz)                            | Float     |
| <code>complex</code>  | Indicates if the data contain complex values      | Boolean   |
| <code>encoding</code> | Encoding format                                   | String    |
| <code>freq</code>     | Indicates if the data are in the frequency domain | Boolean   |
| <code>label</code>    | Observed nucleus                                  | String    |
| <code>obs</code>      | Observed frequency (MHz)                          | Float     |
| <code>size</code>     | Number of data points in spectrum                 | Integer   |
| <code>sw</code>       | Spectral width (Hz)                               | Float     |
| <code>time</code>     | Indicates if the data are in the time domain**    | Boolean   |

\* That is, it is assumed that we are looking at single dimensions from the NMR data, so for example, we are looking at `udic[0]`.

\*\* Being that the data must be in either the frequency or time domain, the `freq` and `time` keywords effectively provide the same information.

## 11.3 Fourier Transforming Data

When the data is first imported, it is often in the time domain. You can confirm this by checking that the `time` value in the `udic` is set to `True`.

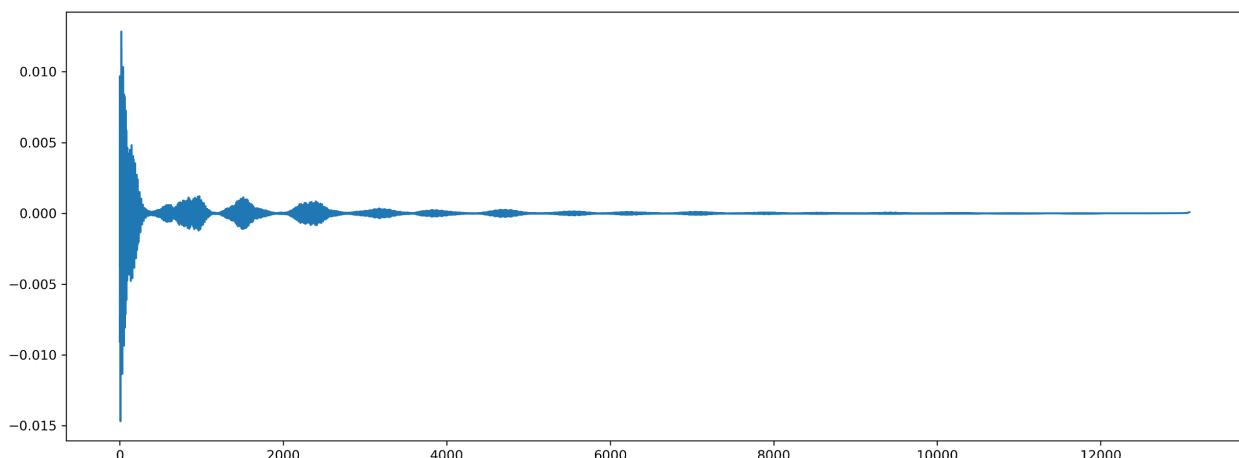
```
[in]: udic[0]['time']
```

```
[out]: True
```

We can also view the data by plotting with `matplotlib`.

```
[in]: fig0 = plt.figure(figsize=(16, 6))
       ax0 = fig0.add_subplot(1,1,1)
       ax0.plot(data.real)
```

```
[out]:
```



To convert the data to the frequency domain, we will use the fast Fourier transform function (`fft`) from the `fft` SciPy module. NMRglue also contains Fourier transform functions, but we will use SciPy here. The plot below inverts the `x-axis` with

`plt.gca().invert_xaxis()` in order to conform with NMR plotting conventions.<sup>95</sup>

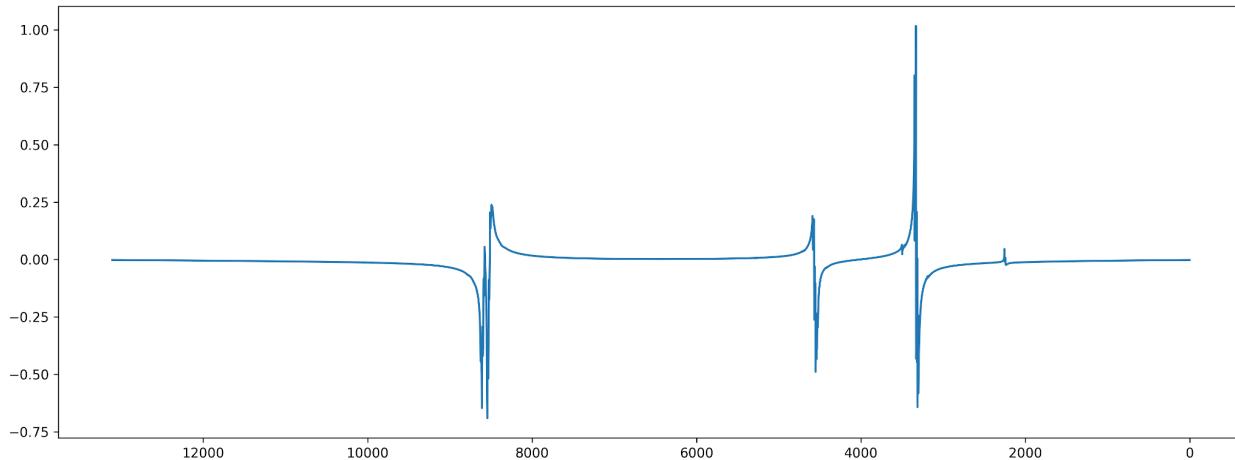
```
[in]: from scipy.fft import fft    # imports only fft function
       fdata = fft(data)
```

```
[in]: fig1 = plt.figure(figsize=(16, 6))
       ax1 = fig1.add_subplot(1,1,1)
       ax1.plot(fdata.real)
```

---

<sup>95</sup> That is, the higher frequency signals are on the left. NMR spectra are unusual in that the numbers increase going left instead of right.

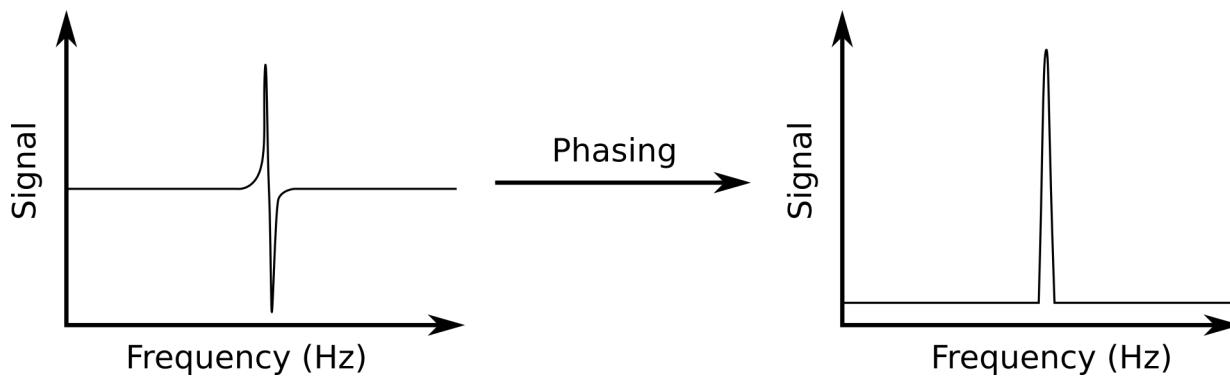
```
plt.gca().invert_xaxis()  
[out]:
```



When you plot the Fourier transformed data, you may get a `ComplexWarning` error message because the Fourier transform will return complex values (i.e., values with real and imaginary components). To only work with the real components, use the `.real` method as is done above. The plot now looks more like an NMR spectrum, but most of the resonances are out of phase. The next step is to phase the spectrum.

## 11.4 Phasing Data

Phasing is the post-processing procedure for making all peaks point upward as shown in Figure 11.2. There is more to it than taking the absolute value as that would not always generate a single peak, so NMRglue contains a series of functions for phasing spectra.



**Figure 11.2** Phasing an NMR spectrum results in all the signals pointing in the positive direction.

### 11.4.1 Autophasing

The simplest method to phase your NMR spectrum is to allow the autophasing function to handle it for you. Below is the function which takes the data and the phasing algorithm as the arguments.

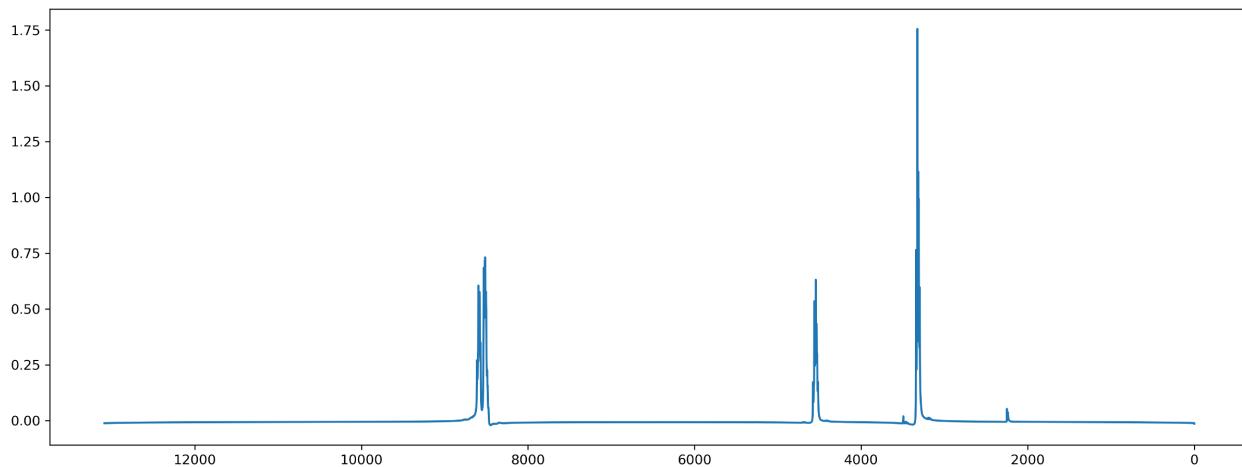
```
ng.process.proc_autophase.autops(data, algorithm)
```

The permitted phasing algorithms can be either 'acme' or 'peak\_minima'. It is important to feed the `autops()` function the data array with *both* the real and imaginary components.

```
[in]: phased_data =
       ng.process.proc_autophase.autops(fdata, 'acme')

[in]: fig2 = plt.figure(figsize=(16, 6))
       ax2 = fig2.add_subplot(1,1,1)
       ax2.plot(phased_data.real)
       plt.gca().invert_xaxis()

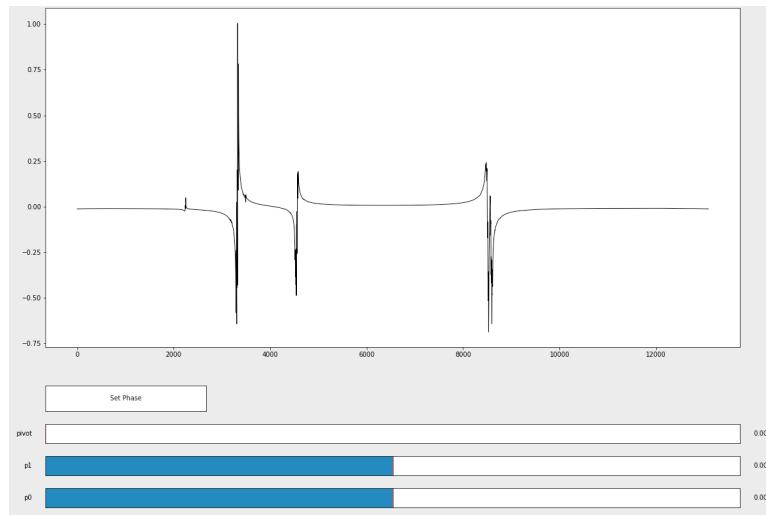
[out]:
```



You should try both algorithms to see which works best for you. The above spectrum is the result of the `acme` autophasing algorithm which is close but still slightly off. If neither of the provided autophasing algorithms work for you, you will need to instead manually phase the NMR spectrum as discussed below.

## 11.4.2 Manual Phasing

Manually phasing the NMR spectrum is a two-step process. First, you need to call the `manual_ps` phasing function and experiment with the `p0` and `p1` sliders until the spectrum appears phased.



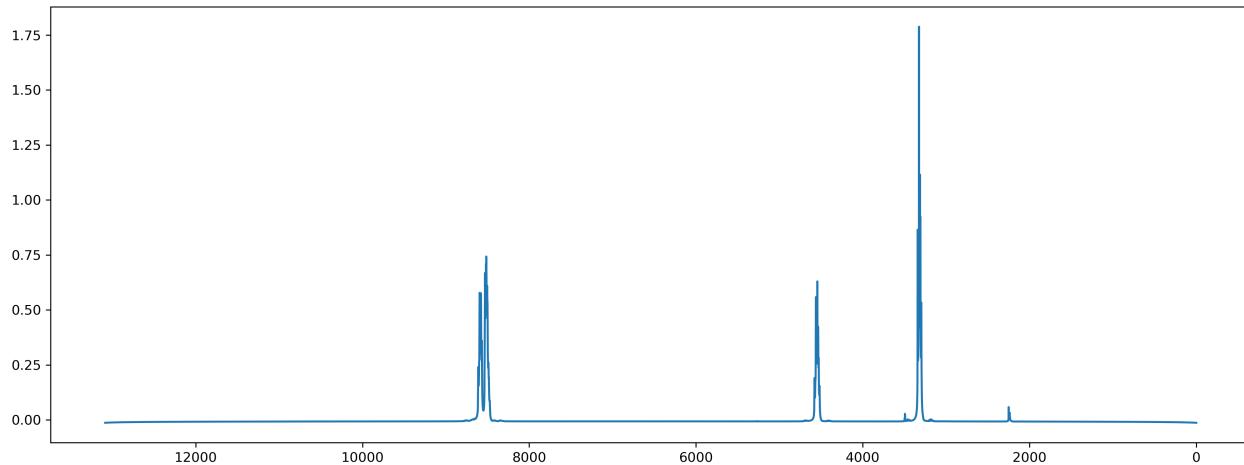
After closing the window, the function will return values for `p0` and `p1` that you found to properly phase the spectrum. Second, input those `p0` and `p1` values into the `ps()` phasing function to actually phase the spectrum.

```
[in]: %matplotlib # exists inline plotting
        p0, p1 = ng.proc_base.manual_ps(fdata.real)

[in]: phased_data = ng.proc_base.ps(fdata, p0=p0, p1=p1)

[in]: %matplotlib inline # reinstates inline plotting
        fig3 = plt.figure(figsize=(16, 6))
        ax3 = fig3.add_subplot(1, 1, 1)
        ax3.plot(phased_data.real)
        plt.gca().invert_xaxis()
```

You can then plot the `phased_data` to get your NMR spectrum with all the peaks pointing upward as shown below.



## 11.5 Chemical Shift

Even though the NMR spectrum is now phased, it is unlikely to be properly *referenced*. That is, the peaks are not currently located at the correct *chemical shift*. Referencing is often performed by knowing the accepted chemical shifts of the solvent resonances or an internal standard (e.g., tetramethylsilane, TMS) and adjusting the spectrum by a correction factor. Currently, we are plotting our data against the index of each data point, so first we need to create a frequency scaled *x*-axis as an array followed by adjusting the location of the spectrum so that it is properly referenced.

### 11.5.1 Generate the X-Axis

The *x*-axis is the frequency scale, so this axis is sometimes presented in hertz (Hz). However, because the frequency of NMR resonances depends upon the instrument field strength, the same sample will exhibit different frequencies in different instruments. To make the frequency axis independent of the spectrometer field strength, NMR spectra are often presented on a *ppm scale* which is the ratio of the frequency of the spectrum (Hz) by the frequency at which the spectrum is collected at (MHz).<sup>96</sup>

$$ppm = \frac{\text{Spectral Frequency (Hz)}}{\text{Observed Frequency (MHz)}}$$

This makes the locations of the peaks consistent from spectrometer to spectrometer no matter the strength of the magnet. This is where the `udi_c` from section 11.2 is important because we can obtain the observed frequency width (Hz) of the spectrum, and the resolution of the

---

<sup>96</sup> Ppm stands for “parts per million” which is what we get if we divide Hz/MHz (i.e., hertz divided by a million hertz).

data. The latter is how many data points are in the spectrum which is important so that we avoid a plotting error (you know the one: “`ValueError: x and y must have same first dimension, ...`”). If any of the values from the `udic` are 999.99, this means the spectrometer did not record this piece of information and you will need to find it elsewhere.

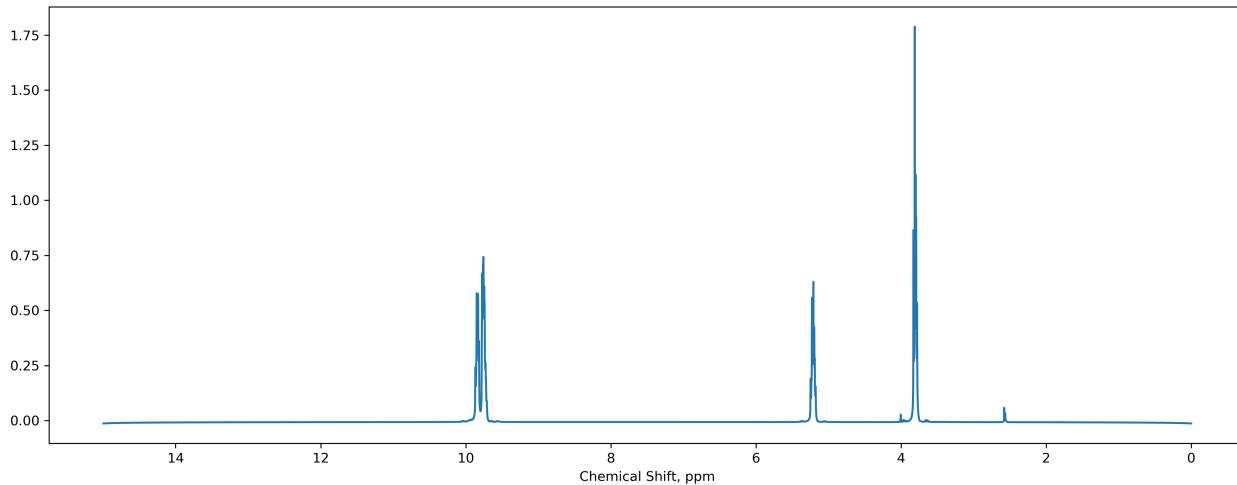
```
[in]: size = udic[0]['size']           # points in data
       sw = udic[0]['sw']              # width in Hz
       obs = udic[0]['obs']            # carrier frequency

[in]: hz = np.linspace(0, floor(sw), size)) # x-axis in Hz
      ppm = hz / freq                 # x-axis in ppm
```

Now if we plot the spectrum, we see it in a ppm scale.

```
[in]: fig4 = plt.figure(figsize=(16, 6))
       ax4 = fig4.add_subplot(1,1,1)
       ax4.plot(ppm, phased_data.real)
       ax4.set_xlabel('Chemical Shift, ppm')
       plt.gca().invert_xaxis()
```

[out] :



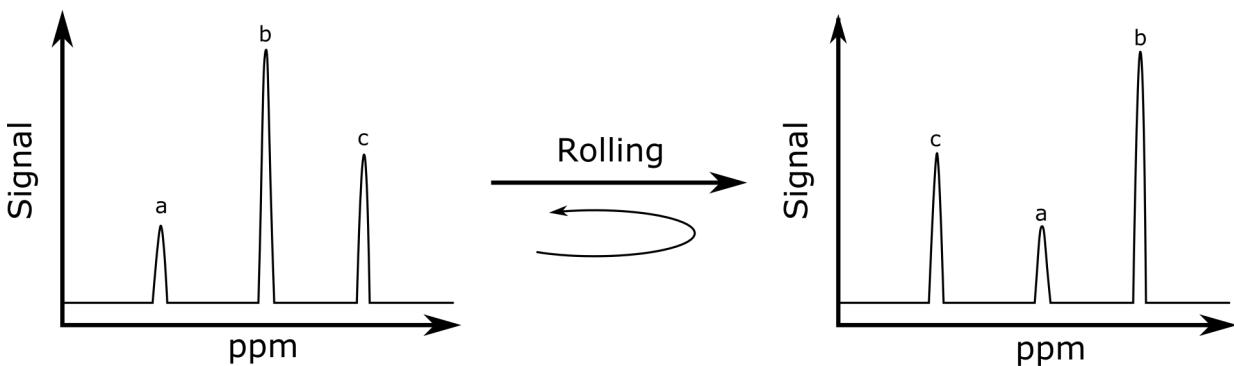
Alternatively, NMRglue contains an object called a `unit conversion object` that can be created and used to convert between ppm, Hz, and point index values for any position in an NMR spectrum. To create a unit conversion object, use the `make_uc` function which takes two arguments – the dictionary, `dic`, and the original data array, `data`, generated from reading the NMR file in section 11.2. Note: if you are using a different NMR file format than pipe, change `pipe` to the appropriate format from Table 11.1.

```
[in]: unit_conv = ng.pipe.make_uc(dic, data)
[in]: ppm = unit_conv.ppm_scale()
```

The last line of the above code generates an array of ppm values required for the *x*-axis to plot the NMR data.

### 11.5.2 Referencing the Data

In the above spectrum, the small resonance at 2.58 ppm is internal TMS (tetramethylsilane) standard which should be located at 0.00 ppm. The temptation is to subtract 2.58 ppm from the *x*-axis, but the spectrum is not simply moved over but instead is *rolled*. That is, as the spectrum is moved, some of it disappears off one end and reappears on the other (Figure 11.3).



**Figure 11.3** Referencing an NMR spectrum is performing by rolling it until the peaks reside at the correct shifts. As a signal falls off one end of the spectrum, it reappears at the other end.

Conveniently for us, NumPy has a function `np.roll()` that does exactly this to array data.

```
np.roll(array, shift)
```

The `np.roll()` function takes two required arguments. The first is the array containing the data and the second is the amount to shift or roll the data. The shift is not in ppm but rather positions in the data array. If you know your referencing correction in ppm ( $\Delta_{ppm}$ ), use the following equation which describes the relationship between the correction in ppm ( $\Delta_{ppm}$ ) and the correction in number of data points ( $\Delta_{points}$ ). The `size` is the number of point in a spectrum, `obs` is the observed carrier frequency, and `sw` is the sweep width in Hz. These values are all available from the universal dictionary.

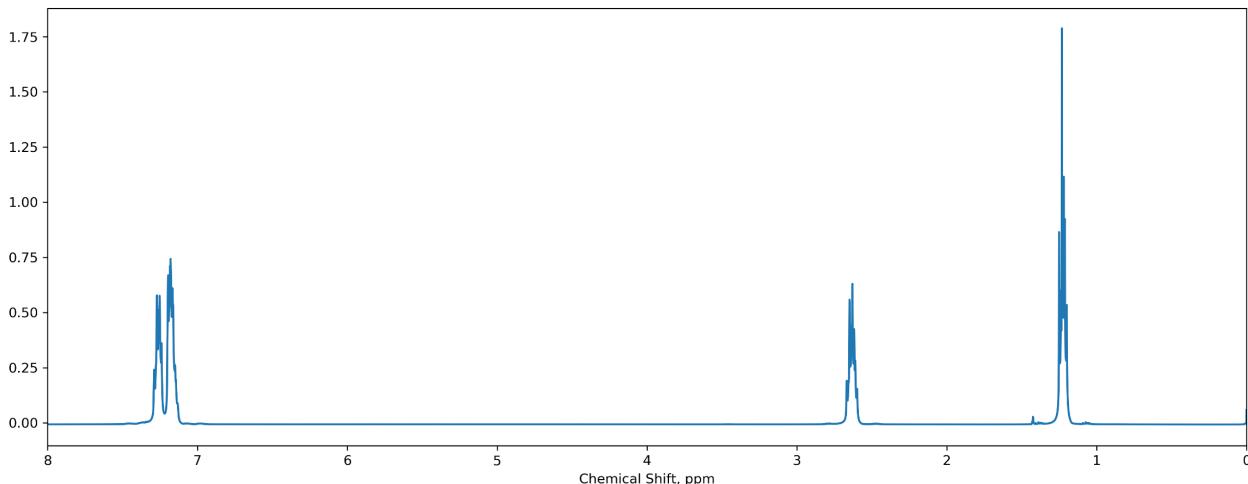
$$\Delta_{points} = \frac{\Delta_{ppm} \times \text{size} \times \text{obs}}{\text{sw}}$$

The example below requires the spectrum to be shifted by -2.58 ppm.

```
[in]: point_shift = int( (-2.58 * size * obs) / sw )
    ref_data = np.roll(phased_data, point_shift)

[in]: fig5 = plt.figure(figsize=(16, 6))
    ax5 = fig5.add_subplot(1,1,1)
    ax5.plot(ppm, ref_data.real)
    ax5.set_xlabel('Chemical Shift, ppm')
    plt.xlim(8,0)
    plt.xticks(np.arange(0,8,1))
    plt.show()

[out]:
```



If you want to narrow the plot to where the resonances are located, you can use the `plt.xlim(8,0)` function. Notice that 8 is first to indicate that the plot is from 8 ppm → 0 ppm. The use of `plt.xlim(8,0)` removes the need to use `plt.gca().invert_xaxis()` to flip the x-axis.

## 11.6 Integration

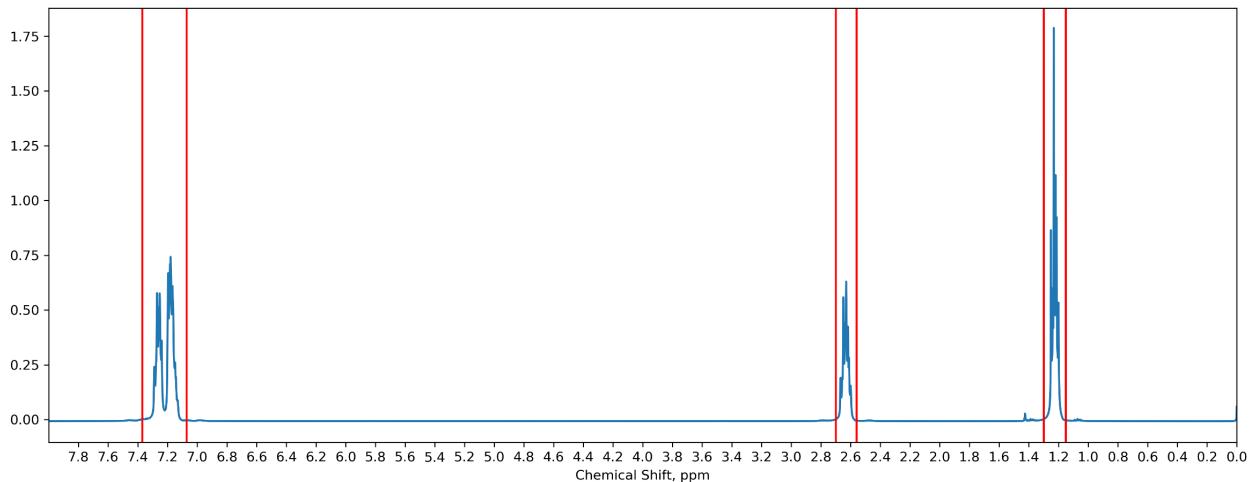
Integration of the area under the peaks can be performed using either integration functions from the `scipy.integrate` module or through NMRglue's integration function(s). Because the integration function in NMRglue supports limit values in the ppm scale, it is probably the most convenient and is demonstrated below.

The integration is performed using the `integrate()` function below where `data` is your NMR data as a NumPy array, the `conv_obj` is an NMRglue unit conversion object (see section 11.5.1), and `limits` is a list or array of limits for integration.

```
ng.analysis.integration.integrate(data, conv_obj, limits)
```

```
[in]: unit_conv = ng.pipe.make_uc(dic, data)
[in]: limits = np.array([[7.07,7.37], [1.15, 1.3],
[2.56,2.7]])
[in]: fig6 = plt.figure(figsize=(16,6))
ax6 = fig6.add_subplot(1,1,1)
ax6.plot(ppm, data_ref.real)
ax6.set_xlabel('Chemical Shift, ppm')
plt.xlim(8,0)
plt.xticks(np.arange(0,8,0.2))
for lim in limits.flatten():
    plt.axvline(lim, c='r')
```

The limits are in ppm, so take a look at the spectrum above and decide where you want to put the integration limits. An NMR spectrum with the chosen integration limits are shown below as vertical red lines.



Now to integrate our NMR spectrum.

```
[in]: area = ng.analysis.integration.integrate(ref_data.real,
  uc, limits)

area

[out]: array([-0.00200144, -0.00126142, -0.00103802],
            dtype=float32)
```

These values are probably not what you expected, but if we divide all of them by the smallest value, we are left with the relative areas under the peaks like you would get from most NMR processing software.

```
[in]: area / np.min(area)

[out]: array([ 1.          ,  0.63025647,  0.51863462],
            dtype=float32)
```

The spectrum above is the  $^1\text{H}$  NMR of ethylbenzene (in  $\text{CDCl}_3$ ) which has five aromatic protons, and the other two resonances should have three and two protons. If we do some math to make the integrations total to ten protons and round to the nearest integer, we get 5:3:2. There is a small amount of error likely due to the solvent resonance ( $\text{CHCl}_3$ , 7.27 ppm) being included in the integration of the aromatic protons.

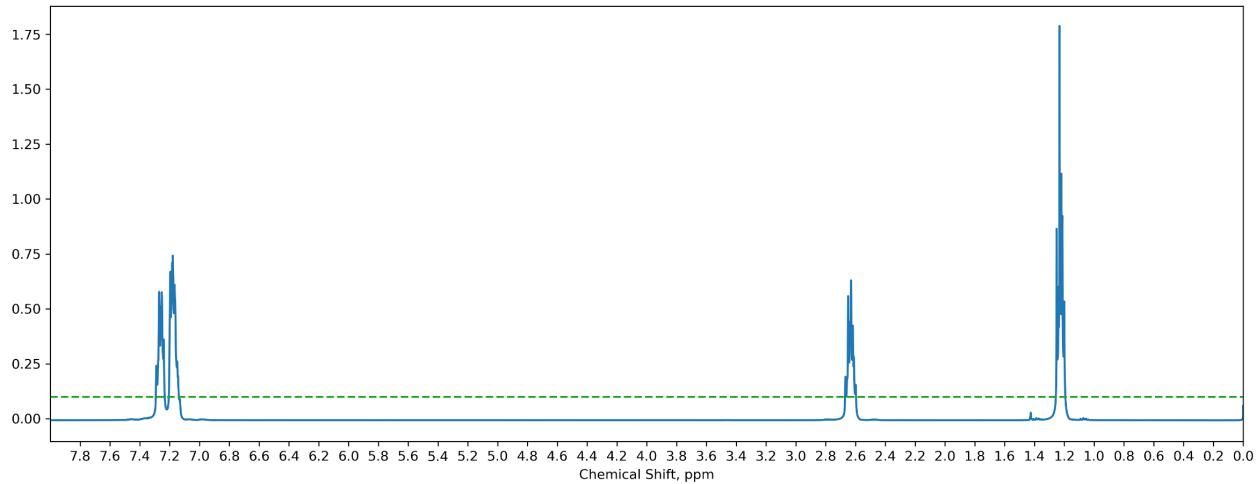
## 11.7 Peak Picking

Another piece of information that is commonly extracted from NMR spectra is the chemical shift of the resonances. Similar to integration, SciPy contains functions such as `scipy.signal.argrelmax()` that can find peaks in spectra, but again, NMRglue contains a function, below, designed for the task of locating peaks in NMR spectra.

```
ng.analysis.peakpick.pick(data, pthres=)
```

There are numerous optional arguments for the peak picking function, but the two mandatory pieces of information required are the `data` array and a positive threshold (`pthres`) above which any peak will be identified. Glancing at the spectrum below, all peaks are above 0.1 (green dotted line) and the baseline is below 0.1, so this seems like a reasonable threshold.

```
[in]: fig7 = plt.figure(figsize=(16, 6))
       ax7 = fig7.add_subplot(1,1,1)
       ax7.plot(ppm, data_ref.real)
       ax7.set_xlabel('Chemical Shift, ppm')
       plt.xlim(8,0)
       plt.xticks(np.arange(0,8,0.2))
       plt.axhline(0.1, c='C2', ls='--')
```



```
[in]: peaks = ng.analysis.peakpick.pick(ref_data.real,
   pthres=0.1)
peaks

[out]:
rec.array([( 1075., 1, 18.18991513, 27.13726807),
            ( 2295., 2, 31.24499905, 17.63588905),
            ( 6265., 3, 35.81783027, 25.40891266),
            ( 6345., 4, 32.05903832, 17.21949005)],
           dtype=[('X_AXIS', '<f8'), ('CID', '<i8'), ('X_LW',
           '<f8'), ('VOL', '<f8')])
```

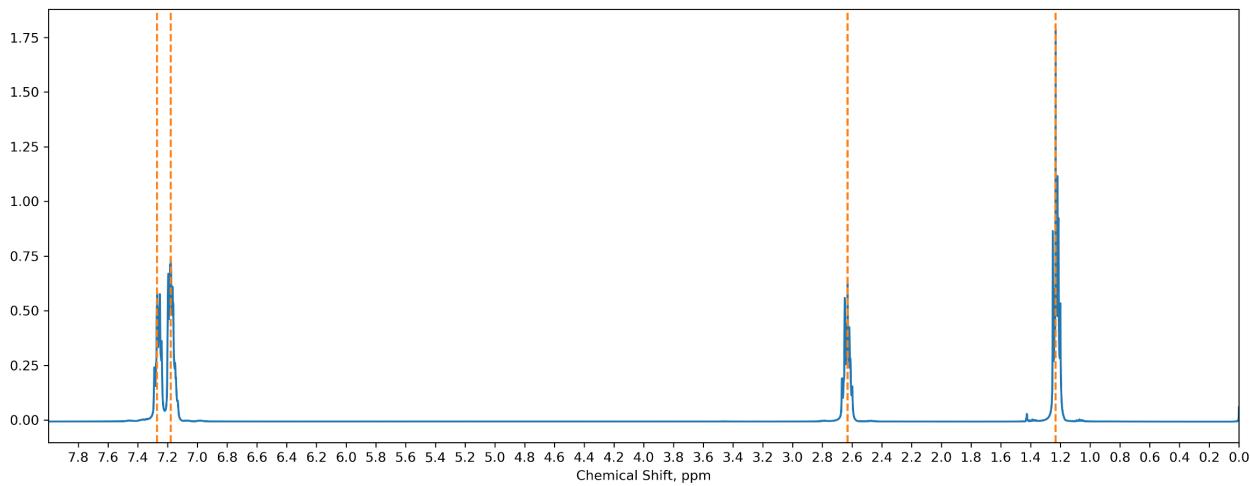
The output of this function is an array of tuples with each tuple containing information about an identified peak. From this, we can already tell there are four peaks identified. Each tuple contains an index for the peak, a peak number, a line width of the peak, and a rough estimate of the volumes of each peak. We can use the index values to index the ppm array for the chemical shifts.

```
[in]: peak_loc = []
for x in peaks:
    peak_loc.append(ppm[int(x[0])])
print(peak_loc)

[out]:
```

```
[1.2318058928838895, 2.6297623480637458, 7.1788501571326222,  
7.2705194328821214]
```

We can plot the NMR spectrum with these chemical shifts marked with vertical dotted lines shown below. Looks like it did a pretty good job locating the resonances! If NMRglue fails to properly identify the peaks, there are a number of parameters described in the documentation (see Further Reading) that can be adjusted.



## Further Reading

1. NMRglue Website. <https://www.nmrglue.com/>.
2. NMRglue Documentation Page. <http://nmrglue.readthedocs.io/en/latest/tutorial.html>.

## Exercises

1. Open the  $^1\text{H}$  NMR spectrum of ethanol, *EtOH\_1H\_NMR.fid*, taken in  $\text{CDCl}_3$  with TMS using NMRglue.
  - a) Plot the resulting spectrum and be sure to properly reference it.
  - b) Integrate the methyl ( $\text{CH}_3$ ) versus the methylene (- $\text{CH}_2-$ ) resonances and calculate the ratio.
2. Open the  $^1\text{H}$  and  $^{13}\text{C}$  NMR spectra of 2-ethyl-1-hexanol, *2-ethyl-1-hexanol\_1H\_NMR\_CDCl3.fid* and *2-ethyl-1-hexanol\_13C\_NMR\_CDCl3.fid*, in  $\text{CDCl}_3$  with TMS and plot them on a ppm scale. Be sure to properly phase and reference the spectra.



# *Chapter 12*

## ***Machine Learning using Scikit-Learn***

Machine learning is a hot topic with popular applications in driverless cars, internet search engines, and data analysis among many others. Numerous fields are utilizing machine learning, and chemistry is certainly no exception with papers using machine learning methods being published regularly.<sup>97</sup> There is a considerable amount of hype around the topic along with debate about whether the field will live up to this hype. However, there is little doubt that machine learning is making a significant impact and is a powerful tool when used properly.

*Machine learning* occurs when a program exhibits behavior that is not explicitly programmed but rather is “learned” from data. This definition may seem somewhat unsatisfying because it is so broad that it is vague and only mildly informative. Perhaps a better way of explaining machine learning is through an example. In section 12.1, we are faced with the challenge of writing a program that can accurately predict the boiling point of simple alcohols when provided with information about the alcohols such as the molecular weight, number of carbon atoms, degree,<sup>98</sup> etc... These pieces of information about each alcohol are known as *features* while the answer we aim to predict (i.e., boiling point) is the *target*. How can each feature be used to predict the target? To generate a program for predicting boiling points, we would need to pour over the data to see how each feature affects the boiling point. Next, we would need to write a script that somehow uses these trends to calculate the boiling points of new alcohols. This probably appears like a daunting task.

---

<sup>97</sup> Machine learning articles are spread out in a wide range of chemical journals, but if you are looking for examples, the American Chemical Society's *Journal of Chemical Information and Modeling* is a good place to start as it contains a higher than average concentration of machine learning publications. The Further Reading section at the end of this chapter also contains multiple simple examples.

<sup>98</sup> That is, primary, secondary, or tertiary.

Instead, we can use machine learning to solve this task by allowing the machine learning algorithms to figure out how to use the data and make predictions. Simply provide the machine learning algorithm with the features and targets on a number of alcohols and allow the machine learning algorithm to quantify the trends and develop a function to predict the boiling point of alcohols. In simple situations, this entire task can be completed in just a few minutes!

The sections in this chapter are broken down by types of machine learning. There are three major branches of machine learning: supervised, unsupervised, and reinforcement learning. This chapter will focus on the first two, which are the most applicable to chemistry and data science, while the latter relates more to robotics and is not as commonly employed in chemistry.<sup>99</sup>

There are multiple machine learning libraries for Python, but one of the most common, general-purpose machine learning libraries is scikit-learn.<sup>100</sup> This library is simple to use, offers a wide array of common machine learning algorithms, and is installed by default with Anaconda. As you advance in machine learning, you may find it necessary to branch out to other libraries, but you will probably find that scikit-learn does almost everything you need it to do during your first year or two of using machine learning. In addition, scikit-learn includes functions for preprocessing data and evaluating the efficacy of models.

The scikit-learn library is abbreviated `sklearn` during imports. Each module needs to be imported individually, so you will see them imported throughout this chapter. We will be working with data and visualizing our results, so we will also be utilizing pandas, NumPy, and matplotlib. This chapter assumes the following imports.

```
[in]: import pandas as pd  
       import numpy as np  
       import matplotlib.pyplot as plt  
       %matplotlib inline
```

## 12.1 Supervised Learning

*Supervised learning* is where the machine learning algorithms are provided with both feature and target information with the goal of developing a model to predict targets based on the features. When the supervised machine learning predictions are looking to categorize an

<sup>99</sup> Reinforcement learning can be described as conditioning for computers through positive and negative reinforcement analogous to the psychological conditioning of lab rats to solve puzzles. For an example of reinforcement in chemistry, see: Zhou, Z.; Li, X.; Zare, R. N. Optimizing Chemical Reactions with Deep Reinforcement Learning. *ACS Cent. Sci.* **2017**, 3, 1337–1344 (DOI: 10.1021/acscentsci.7b00492). This is an open-access article, so it is free to download from the <https://pubs.acs.org> website.

<sup>100</sup> Scikit-Learn Website. <http://scikit-learn.org/stable/>.

item like a photo or type of metal complex, it is known as *classification*; and when the predictions are seeking a numerical value from a continuous range, it is a *regression* problem. Some machine learning algorithms are designed for only classification or only regression while others can do either.

There are numerous algorithms for supervised learning; below are simple examples employing some well known and common algorithms. For a more in depth coverage of the different machine learning algorithms and scikit-learn, see the Further Reading section at the end of this chapter.

### 12.1.1 Features and Information

The file titled *ROH\_data.csv* contains information on over seventy simple alcohols (i.e., a single -OH with no other non-hydrocarbon function groups) including their boiling points.<sup>101</sup> Our goal is to generate a function or algorithm to predict the boiling points of the alcohols based on the information on the alcohols, so here the target is the boiling point and features are the other information about the alcohols.

```
[in]: ROH = pd.read_csv('ROH_data.csv', sep=', ')
      ROH.head()
```

```
[out]:
```

|          | <b>bp</b> | <b>MW</b> | <b>carbons</b> | <b>degree</b> | <b>aliphatic</b> | <b>avg_aryl_position</b> | <b>cyclic</b> |
|----------|-----------|-----------|----------------|---------------|------------------|--------------------------|---------------|
| <b>0</b> | 338       | 32.04     | 1              | 1             | 1                | 0                        | 0             |
| <b>1</b> | 352       | 46.07     | 2              | 1             | 1                | 0                        | 0             |
| <b>2</b> | 370       | 60.10     | 3              | 1             | 1                | 0                        | 0             |
| <b>3</b> | 356       | 60.10     | 3              | 2             | 1                | 0                        | 0             |
| <b>4</b> | 391       | 74.12     | 4              | 1             | 1                | 0                        | 0             |

The dataset includes the boiling point (K), molecular weight (g/mol), number of carbon atoms, whether or not it is aliphatic, degree, whether it is cyclic, and the average position of any aryl substituents. Scikit-learn requires that all features be represented numerically, so for the last three features 1 represents True and 0 represents False.

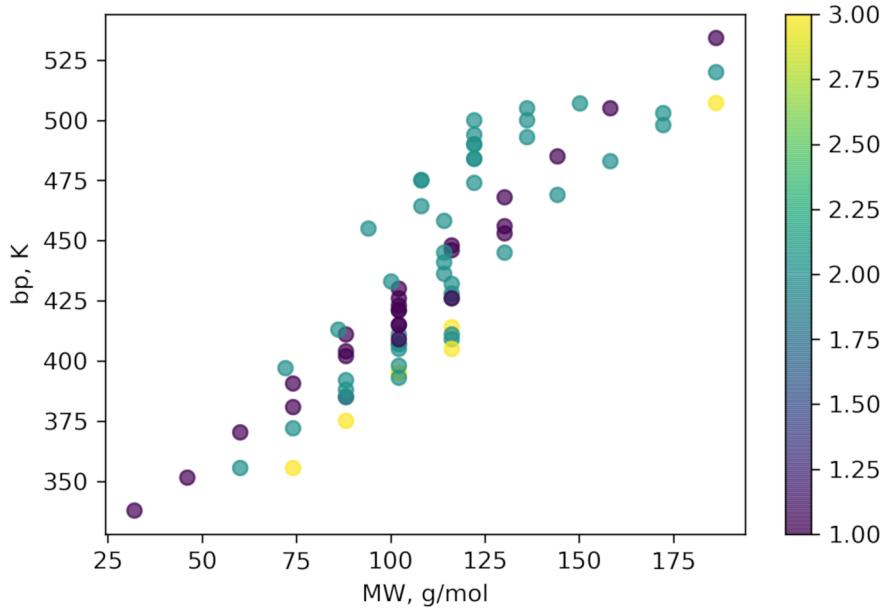
Not every feature will be equally helpful in predicting the boiling points. Chemical intuition may lead someone to propose that the molecular weight will have a relatively large impact on the boiling points, and the scatter plot below supports this prediction with boiling

---

<sup>101</sup> The boiling points were compiled from the NIST Chemistry WebBook at <https://webbook.nist.gov/chemistry/>. When multiple differing boiling points are reported, the most recent, mode, or mean is included in this dataset.

points increasing with molecular weight. However, the molecular weight alone is not enough to obtain a good boiling point prediction as there is as much as a one hundred degree variation in boiling points at around the same molecular weight. The color of the markers indicates the degree of the alcohol, and it is pretty clear that tertiary alcohols tend to have lower boiling points than primary and secondary which means there is a small amount of information in the degree that can be used to improve a boiling point prediction. If all the small amounts of information from each feature are combined, there is potential to produce a better boiling point prediction, and machine learning algorithms do exactly this.

```
[in]: plt.scatter(ROH['MW'], ROH['bp'], alpha=0.5,
                 c=data['degree'], cmap='viridis')
plt.colorbar()
plt.xlabel('MW, g/mol')
plt.ylabel('bp, K')
```



### 12.1.2 Train Test Split

Whenever training a machine learning model to make predictions, it is important to evaluate the accuracy of the predictions. It is unfair to test an algorithm on data it has already seen, so before training a model, first split the dataset into a training subset and testing subset. It is also important to shuffle the data set before splitting it as many datasets are at least partially ordered. The alcohol dataset is roughly in order of molecular weight, so if an algorithm is trained on the first three-quarters of the data set and then tested on the last

quarter, training occurs on smaller alcohols and testing on larger alcohols. This could result in poorer predictions as the machine learning algorithm is not familiar with the trends of larger alcohols. The good news is that scikit-learn provides a built-in function for shuffling and splitting the dataset known as `train_test_split()`. The arguments are the features, target, and the fraction of the dataset to be used for testing. Below, a quarter of the dataset is allotted for testing (`test_size=0.25`).

```
[in]: from sklearn.model_selection import train_test_split  
  
[in]: target = ROH['bp']  
      features = ROH[['MW', 'carbons', 'degree', 'aliphatic',  
                      'avg_aryl_position', 'cyclic']]  
  
[in]: x_train, x_test, y_train, y_test =  
      train_test_split(features, target, test_size=0.25)
```

The output includes four values containing the training/testing features and targets. By convention, `X` contains the features and `y` are the target values because they are the independent and dependent variables, respectively; and the `features` variable is capitalized because it contains multiple values per alcohol.

### **12.1.3 Training a Linear Regression Model**

Now for some machine learning using a very simple *linear regression model*. This model treats the target value as a linear combination or weighted sum of the features where  $x$  are the features and  $w$  are the weights.

$$target = w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5$$

The general procedure for supervised machine learning, regardless of model, usually includes three steps.

1. Create a model and attach it to a variable
2. Train the model with the training data
3. Evaluate the model using the testing data or use it to make predictions.

To implement these steps, the linear model from the `linear_model` module is first created with the `LinearRegression()` function and assigned the variable `reg`. Next, it is trained using the `fit()` method and the training data from above.

```
[in]: from sklearn import linear_model  
      reg = linear_model.LinearRegression()
```

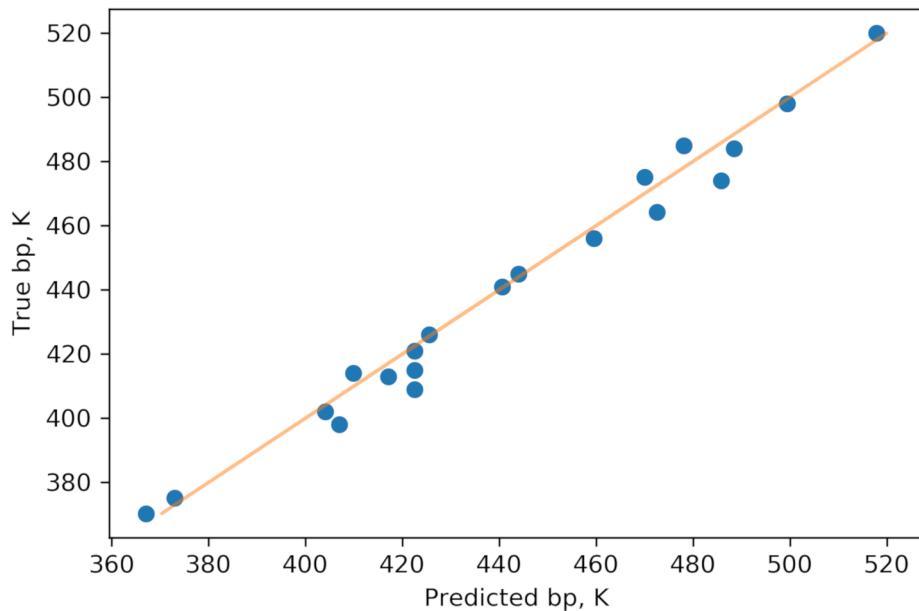
```
[in]: reg.fit(X_train, y_train)
```

Finally, the trained model can make predictions using the `predict()` method.

```
[in]: prediction = reg.predict(X_test)
```

Remember that the algorithm has been trained only on the features for the testing subset; it has never seen the `y_test` target data. The performance can be assessed by plotting the predictions against the true values.

```
[in]: plt.plot(prediction, y_test, 'o')
plt.plot(y_test, y_test, '--', lw=1.3, alpha=0.5)
plt.xlabel('Predicted bp, K')
plt.ylabel('True bp, K')
```



This is a substantial improvement from using only the molecular weight to make predictions! If the above code is run again, the results will likely vary because the `train_test_split()` function randomly splits the dataset, so each time the above code is run, the algorithm is trained and tested on different portions of the original dataset.

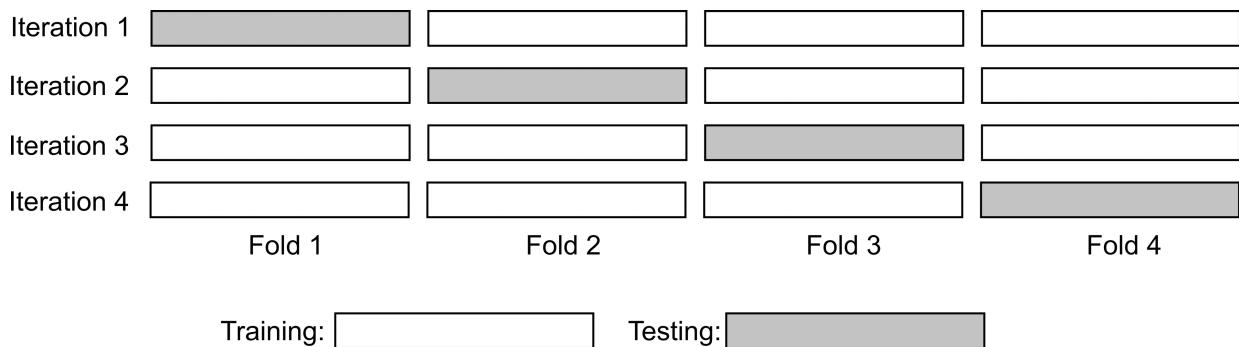
#### 12.1.4 Model Evaluation

It is important to evaluate the effectiveness of trained machine learning models before rolling them out for widespread use, and scikit-learn provides multiple built-in functions to help in this task. The first is the `score()` method. Instead of making predictions using the testing features and then plotting the predictions against the known values, the `score()`

method takes in the testing features and target values and returns the  $r^2$ . The closer the  $r^2$  value is to one, the better the predictions are.

```
[in]: reg.score(X_test, y_test)  
[out]: 0.9786372189524518
```

Another tool for evaluating the efficacy of a machine learning algorithm is *k-fold cross-validation*. The prediction results will vary depending upon how the dataset is randomly split into training and testing data. *K*-fold cross-validation compensates for this randomness by splitting the entire dataset into *k* chunks called *folds*. It then reserves one fold as the testing fold and trains the algorithm on the rest. The algorithm is tested using the testing fold and the process is repeated with a different fold reserved for testing (Figure 12.1). Each iteration trains a fresh algorithm, so it does not remember anything from the previous train/test iteration. The results for each iteration are provided at the end of this process.



**Figure 12.1** In each iteration of *k*-fold cross-validation, different folds of data are used for training and testing the algorithm.

A demonstration of *k*-fold cross validation is show below. First, a cross-validation generator is created using the `ShuffleSplit()` function. This function shuffles the data to avoid having all similar alcohols in any particular fold. The linear model is then provided to the `cross_val_score()` function along with the feature and target data and the cross-validation generator.

```
[in]: from sklearn.model_selection import cross_val_score,  
        ShuffleSplit  
  
[in]: splitter = ShuffleSplit(n_splits=5)  
[in]: reg = linear_model.LinearRegression()
```

```
scores = cross_val_score(reg, features, target,  
                        cv=splitter)  
  
[in]: scores  
  
[out]: array([0.98364425, 0.98380668, 0.98569942, 0.98706059,  
             0.98264135])
```

The scores are the  $r^2$  values for each iteration. The average  $r^2$  is a pretty reasonable assessment of the efficacy of the model and can be found through the `mean()` function.

```
[in]: scores.mean()  
  
[out]: 0.9845704567350738
```

### 12.1.5 Linear Models and Coefficients

Recall that the linear model calculates the boiling point based on a weighted sum of the features, so it can be informative to know the weights to see which features are the most influential in making the predictions. The `LinearRegression()` method contains the attribute `coef_` which provides these coefficients in a NumPy array.

```
[in]: reg = linear_model.LinearRegression()  
      reg.fit(X_train, y_train)  
      reg.coef_  
  
[out]: array([-6.15050535, 104.74786254, -15.58509361,  
             13.40265618, -2.61340234, 16.14686839])
```

These coefficients correspond to molecular weight, number of carbons, degree, whether or not it is aliphatic, average aryl position, and whether or not it is cyclic, respectively. While some coefficients are larger than others, we cannot yet distinguish which features are more important than the others because the values for each feature occur in different ranges. This is because the coefficients are not only proportional to the predictive value of a feature but also inversely proportional to the magnitude of feature values. For example, while the molecular mass has greater predictive value than the degree, the degree has a larger coefficient because it occurs in a smaller range ( $1 \rightarrow 3$ ) than the molecular weights ( $32.04 \rightarrow 186.33$  g/mol).

To address this issue, the scikit-learn `sklearn.preprocessing` module provides a selection of functions for scaling the features to the same range. Three common feature scaling functions are described in Table 12.1, but others are detailed on the scikit-learn website.

**Table 12.1** Preprocessing Data Scaling Functions

| Scaler         | Description                                                                                                                                          |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| MinMaxScaler   | Scales the features to a designated range; defaults to [0, 1]                                                                                        |
| StandardScaler | Centers the features around zero and scales them to a variance of one                                                                                |
| RobustScaler   | Centers the features around zero using the median and sets the range using the quartiles; similar to StandardScaler except less affected by outliers |

For this data, we will use the `MinMaxScaler` with the default scaling of values from 0 → 1. This process parallels the `fit/predict` procedure above except that instead of predicting the target, the algorithm transforms it. That is, first the algorithm learns about the data using the `fit` method followed by scaling the data using the `transform` method. Once the scaling model is trained, it can be used to scale any new data by the same amount as the original data.

```
[in]: scaler = MinMaxScaler()
scaler.fit(features)
scaled_features = scaler.transform(features)
```

With the features now scaled, we can proceed through training the linear regression model as we have done previously and examine the coefficients.

```
[in]: X_train, X_test, y_train, y_test =
       train_test_split(scaled_features, target)

[in]: reg = linear_model.LinearRegression()
reg.fit(X_train, y_train)

[in]: reg.coef_

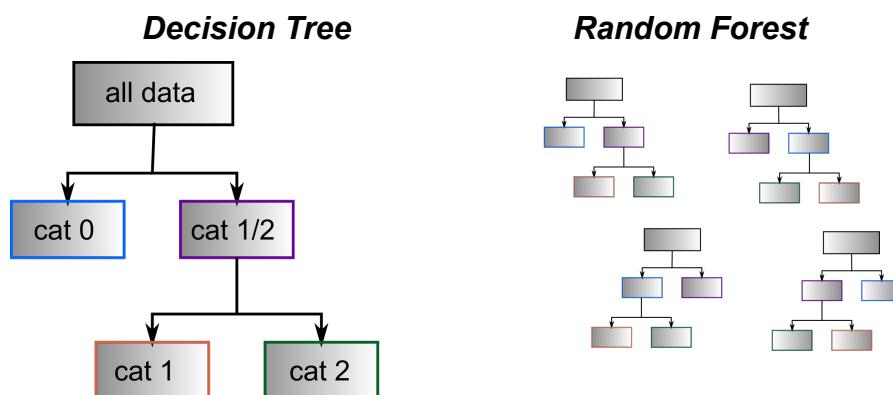
[out]: array([-915.94709534, 1122.93622447, -31.09690156,
           12.90844675, -14.24631555, 16.06530556])
```

It is quite clear from the coefficients that the molecular weight and number of carbons are both by far the most important features to predicting the boiling points of alcohols. This makes chemical sense being that larger molecules have greater London dispersion forces thus increasing the boiling points.

### 12.1.5 Classification using Random Forests

Classification involves sorting items into discrete categories such as sorting alcohols, aldehydes/ketones, and amines by type based on features. Scikit-learn provides a number of

algorithms designed for this type of task. One method is known as a *decision tree* (Figure 12.2, left) which sorts items into categories based on a series of conditions. For example, it might first sort chemicals based on which have degrees of unsaturation greater than zero<sup>102</sup> because these are most likely to be the aldehydes and ketones. It will then take the samples with zero degrees of unsaturation, which are the alcohols and amines, and separate them through another condition based on other information about the chemical compounds. Decision trees are relatively simple and easily interpreted, but they tend not to perform particularly well in practice. An extension of the decision tree is the *random forest* (Figure 12.2, right) which trains a larger number of decision trees using different subsets of the training data resulting in large numbers of different decision trees. Each decision tree is used to predict the category, and the final prediction is based on the majority prediction of all the trees. Random forests tend to be more accurate than a single decision tree because even if every tree is only slightly better than random at making an accurate prediction, large numbers of decision trees have a much higher probability of making a correct prediction because of the law of large numbers.



**Figure 12.2** An illustration of a single decision tree (left) and a random forest (right) composed of numerous decision trees generated with different subsections of data.

### 12.1.6 Classify Chemical Compounds

To demonstrate classification, we will use a small dataset containing 122 monofunctional organic compounds from three different categories: alcohols (category 0), ketones/aldehydes (category 1), and amines (category 2). The features provided are the molecular weight, number of carbons, boiling point, whether it is cyclic, whether it is aromatic, and the unsaturation number. All the data is represented numerically, so the data is ready to be used.

<sup>102</sup> Often due to double or triple bonds, though this can also be the result of ring structures.

```
[in]: data = pd.read_csv('org_comp.csv')
      data.head
```

```
[out]:
```

|   | class | bp  | MW     | C | cyclic | unsaturation |
|---|-------|-----|--------|---|--------|--------------|
| 0 | 0     | 455 | 94.11  | 6 | 0      | 0            |
| 1 | 0     | 475 | 108.14 | 7 | 0      | 0            |
| 2 | 0     | 475 | 108.14 | 7 | 0      | 0            |
| 3 | 0     | 464 | 108.14 | 7 | 0      | 0            |
| 4 | 0     | 474 | 122.17 | 8 | 0      | 0            |

```
[in]: target = pd.data['class']
      features = data.drop('class', axis=1)
```

Now that we have our data, the classification process is similar to the regression example above: first perform a train/test split, initiate the model, train the model, and then test it.

```
[in]: X_train, X_test, y_train, y_test =
      train_test_split(features, target, test_size=0.25)

[in]: from sklearn.ensemble import RandomForestClassifier
      rf = RandomForestClassifier()
      rf.fit(X_train, y_train)
      rf.predict(X_test)
```

```
[out]: array([2, 0, 1, 0, 2, 2, 0, 2, 0, 0, 1, 0, 0, 0, 0, 2,
           1, 2, 0, 0, 0, 0, 0, 0, 1, 0, 2, 2, 2, 1])
```

We now have predictions for our testing data, but it would be helpful to know how accurate these predictions are. Again, there is the `score` method that can calculate the fraction of accurately predicted functional groups.

```
[in]: rf.score(X_test, y_test)
[out]: 0.8709677419354839
```

### 12.1.7 Confusion Matrix

The above score shows that the predictions are about 87% accurate. However, with three possible categories, this number does not tell the whole story because it does not inform

us as to where the errors are occurring. For this, we will use a *confusion matrix* which is a grid of predicted categories versus true categories.

```
[in]: from sklearn.metrics import confusion_matrix  
  
[in]: conf_matrix = confusion_matrix(y_test,  
                                     rf.predict(X_test))  
      conf_matrix  
[out]:  
  
array([[13,  0,  0],  
       [ 1,  5,  0],  
       [ 3,  0,  9]])
```

Each row is a predicted category and each column is the true category, but it is difficult to interpret the confusion matrix without labels. We can use seaborn's `heatmap` function (see section 10.6) to produce a clearer representation.

```
[in]: import seaborn as sns  
      sns.heatmap(conf_matrix, annot=True, cmap='Blues')  
      plt.xlabel('True Value')  
      plt.ylabel('Predicted Value')  
  
[out]:
```



Every value in the diagonal has the same predicted category as the true value, making them correct predictions, whereas anything off diagonal are incorrect predictions. For example, the bottom left corner shows that three instance were predicted as category 2 but really belong to category 0. Examination of the confusion matrix shows that the most common erroneous prediction is a category 0. This could be due to, for example, the fact that alcohols and amines both tend to have degrees of unsaturation of zero in this dataset.

## 12.2 Unsupervised Learning

Another major class of machine learning is *unsupervised learning* where no target value is provided to the machine learning algorithm. Unsupervised learning seeks to find patterns in the data instead of making predictions. One form of unsupervised problem is *dimensionality reduction* where the number of features is condensed down to typically two or three features while maintaining as much information as possible. Another unsupervised learning task is *clustering* where the algorithm attempts to group similar items in a dataset. Because no target label is available, the algorithm does not know what each group contains; it only knows that the data fall into a pattern of cohesive groups. *Blind signal separation* (BSS) is a third unsupervised task introduced below where the algorithm attempts at pulling apart mixed signals into its components without knowledge of the components. One application of BSS is extracting the spectra of pure compounds from spectra containing a mixture of chemical compounds.

### 12.2.1 Dimensional Reduction

We will first address dimensionality reduction which typically condenses features down to two or three dimensions because it is often used in the visualization complex data. To demonstrate this task, we will use scikit-learn's `datasets` module which contains datasets along with data-generating functions. We will use the wine classification dataset that includes 178 samples of three different types of wines which we will classify based on features such as alcohol content, hue, malic acid, etc....

### 12.2.2 Load Wine Dataset

To load the wine dataset, we first need to import the `load_wine` function and then call the function.

```
[in]: from sklearn.datasets import load_wine  
       wine = load_wine()
```

The data is now stored as a dictionary-style object in the variable `wine` with the features stored under the key `data` and targets stored under `target`.

```
[in]: wine.data  
[out]:  
  
array([[1.423e+01, 1.710e+00, 2.430e+00, ..., 1.040e+00,  
       3.920e+00, 1.065e+03],  
      [1.320e+01, 1.780e+00, 2.140e+00, ..., 1.050e+00,  
       3.400e+00, 1.050e+03],  
      [1.316e+01, 2.360e+00, 2.670e+00, ..., 1.030e+00,  
       3.170e+00, 1.185e+03],  
      ...,  
      [1.327e+01, 4.280e+00, 2.260e+00, ..., 5.900e-01,  
       1.560e+00, 8.350e+02],  
      [1.317e+01, 2.590e+00, 2.370e+00, ..., 6.000e-01,  
       1.620e+00, 8.400e+02],  
      [1.413e+01, 4.100e+00, 2.740e+00, ..., 6.100e-01,  
       1.600e+00, 5.600e+02]])
```

```
[in]: wine.target
```

```
[out]:
```

```
array([0,  
0,  
0, 1,  
1,  
1,  
1,  
1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,  
2,  
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Notice again that every data point is a number including the category because scikit-learn requires that all data are numerically encoded. We can get a full listing of the keys using the `keys()` method shown below. Most keys are self-explanatory except for the `DESCR` which provides a description of the dataset for those who are interested.

```
[in]: wine.keys()
```

```
[out]: dict_keys(['data', 'target', 'target_names', 'DESCR',  
                 'feature_names'])
```

We will store the features and target values in variables for use in the next section.

```
[in]: features = wine.data
```

```
target = wine.target
```

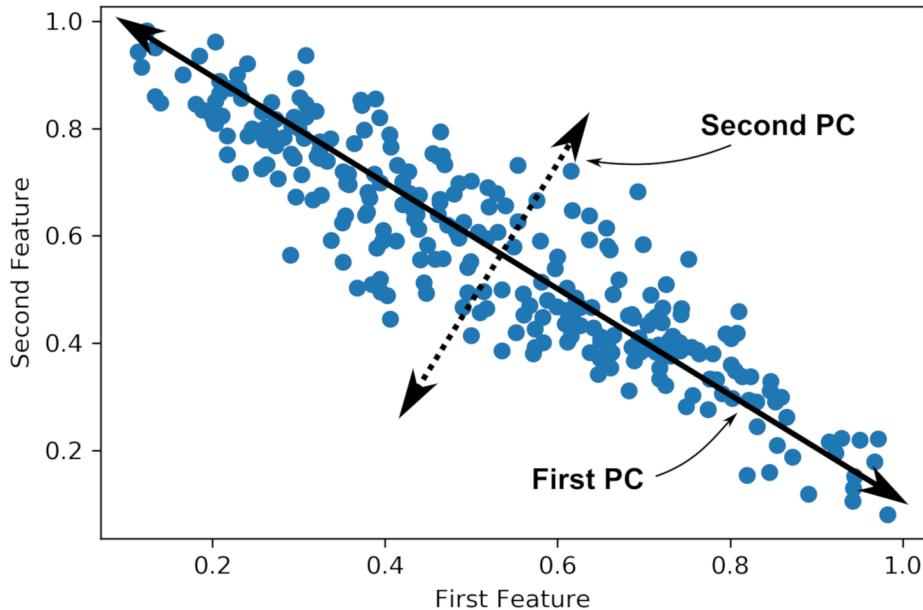
### 12.2.3 Reduce Dimensionality of Wine Dataset

Below is a list of thirteen features in the wine dataset which is too many to represent in a single plot, so it needs to be pared down to two or three.

```
[in]: wine.feature_names  
[out]: ['alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash',  
       'magnesium', 'total_phenols', 'flavanoids',  
       'nonflavanoid_phenols', 'proanthocyanins',  
       'color_intensity', 'hue',  
       'od280/od315_of_diluted_wines', 'proline']
```

Inevitably, some information will be lost by representing high-dimensionality data in lower dimensions, but the algorithms in scikit-learn are designed to preserve as much information as possible. Among the most common algorithms is *principle component analysis (PCA)* which determines the axes of greatest variation in the dataset known as principle components. The first principle component is the axis of greatest variation, the second principle component is the axis of the second greatest variation, and so on. Every subsequent principle component is also orthogonal to the previous principle components.

As a simplified example, below is a dataset containing only two features. The axis of greatest variation slopes down and to the right, shown with a longer solid line, making this the first principle component. The second principle component is the axis of second greatest variation perpendicular to the first axis shown as a dotted line. If the data had a third dimension, the third principle component would come directly out of the page orthogonal to the first two principle components. Each data point is then represented by its relationship to the principle component axes. That is, the principle components are the new Cartesian axes. This may seem trivial with only two features, but it allows high-dimensional data to be reasonably represented in only two or three dimensions while preserving as much information as possible.



**Figure 12.3** Principle components are axes of greatest variation of a dataset in feature space. The first principle component is the axis of greatest variation while the second principle component is the axis of second greatest variation orthogonal to the first.

The PCA algorithm is provided in the `decomposition` module of scikit-learn. Unsupervised learning procedures are similar to those of supervised learning except that there is no reason to split the data into training and testing sets, and instead of making predictions, the trained algorithm is used to transform the data. The general process is outlined below.

1. Create a model attached to a variable
2. Train the model with the `fit` method using all of the data
3. Modify the data using the `transform` method

Principle component analysis is sensitive to the scale of features, so before we proceed, we will scale the features using the `StandardScaler()` function introduced in section 12.1.5.

```
[in]: from sklearn.preprocessing import StandardScaler
[in]: SS = StandardScaler()
       features_ss = SS.fit_transform(features)
```

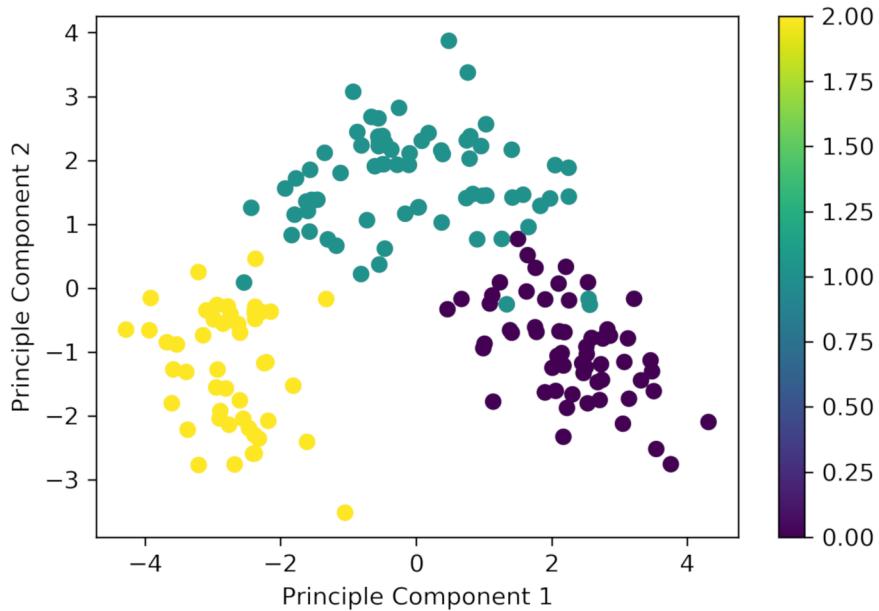
When training the PCA model, it can take a number of arguments. Most are beyond the scope of this chapter, but the one you should focus on is `n_components=` where the user

provides the number of principle components desired. In this case, we will obtain two principle components because it is the easiest to visualize.

```
[in]: from sklearn.decomposition import PCA  
  
[in]: pca = PCA(n_components=2)  
      trans_data = pca.fit_transform(features_ss)  
      trans_data.shape  
  
[out]: (178, 2)
```

The result is a two-dimensional array where each column represents a principle component. We can plot these components against each other and color the markers based on the class.

```
[in]: plt.scatter(trans_data[:,0], trans_data[:,1], c=target)  
  
[out]:
```



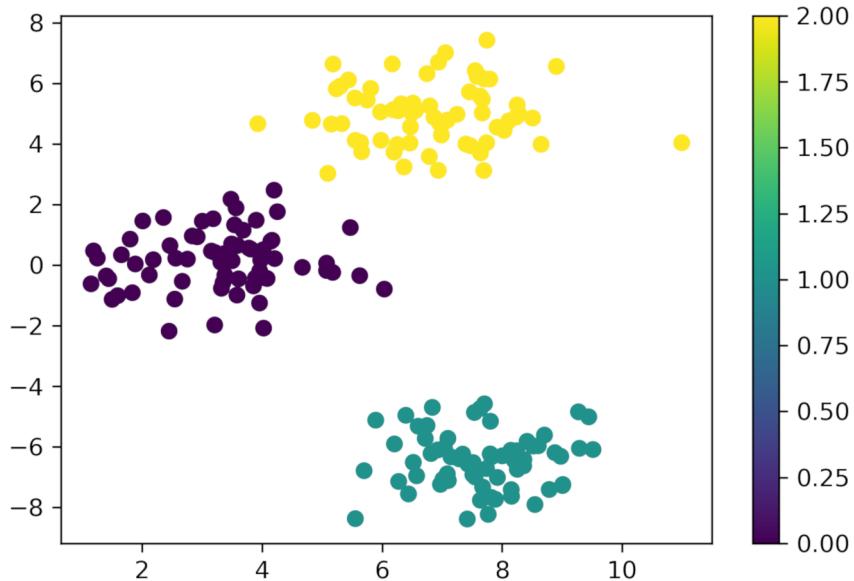
We can see that the three categories of wine all form cohesive clusters with class 0 and 2 being well resolved and class 1 exhibiting slight overlap with the other two classes of wine. This suggests that we should have better luck distinguishing between class 0 and 2 than between these two classes and class 1.

## 12.2.4 Clustering

Clustering involves grouping similar items in a dataset, and this can be performed with a number of algorithms including  $k$ -means, agglomerative clustering, and Density Based Spacial Clustering Application with Noise (DBSCAN) among others. This process is somewhat similar to classification except that no labels are provided, so the algorithm does not know anything about the groups and must rely on the similarity of samples. Here we will use the DBSCAN clustering algorithm. This algorithm works by assigning items in a dataset as *core* data points if they are within a minimum distance (`eps`) of a minimum number of other samples in a dataset (`min_samples`). Clusters are built around these core data points, and any data point not within `eps` distance from a core data point is designated as *noise*, which means it is not assigned to any cluster. The larger the minimum distance and smaller minimum number of samples, the fewer clusters that are likely to be predicted by DBSCAN. One notable attribute of this algorithm versus some of the others mentioned above is that DBSCAN does not require the user to provide a requested number of clusters; it determines the number of clusters based on the other parameters mentioned above.

To demonstrate clustering, we will generate a random, synthetic dataset using the `make_blobs` function from the `sklearn.datasets` module. This function takes a number of arguments including the number of samples (`n_samples`), number of features (`n_features`), number of clusters (`centers`), and the standard deviation of the clusters (`cluster_std`). We will only generate two features to make this example easy to visualize. The output of `make_blobs` is a NumPy array containing the features (`X`) and a second NumPy array containing the labels (`y`).

```
[in]: from sklearn.datasets import make_blobs  
  
[in]: X, y = make_blobs(n_samples=200, n_features=2,  
                      centers=3, cluster_std=1)  
      plt.scatter(X[:,0], X[:,1], c=y)  
  
[out]:
```



We can see three distinct clusters with the cluster on the bottom being more distinct than the two at the top. Also notice that the scales of the two features are different by roughly a factor of two. Before we can use this data, we will need to normalize the scale of both features as clustering algorithms are sensitive to scale.<sup>103</sup> For this task, we will use the `StandardScaler` function introduced in section 12.2.5.

```
[in]: SS = StandardScaler()
X_ss = SS.fit_transform(X)
```

Now that the data is scaled, we will initiate our model, train it using the `fit` method, and examine the predictions using the `labels_` attribute.

```
[in]: DB = DBSCAN(eps=0.4, min_samples=5)
DB.fit(X_ss)

[in]: DB.labels_

[out]:
array([0, 0, 1, 1, 2, 0, 2, 1, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 1, 0,
2, 0, 2, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 2, 2, 1, 0, 0,
0, 1, 0, 0, 1, 1, 2, 0, 2, -1, 1, 0, 1, 1, 1, 0, 0, 1, 2, 1,
2, 0, 2, 2, 0, 1, 0, 2, 2, 2, 0, 2, 1, 1, 0, 2, 1, 0, 2, 0, 1,
```

---

<sup>103</sup>If the scales of the features are substantially different, the larger feature will have a greater impact on the calculation of distance between data points. This in turn affects which data points are assessed at being close to each other in feature space.

```

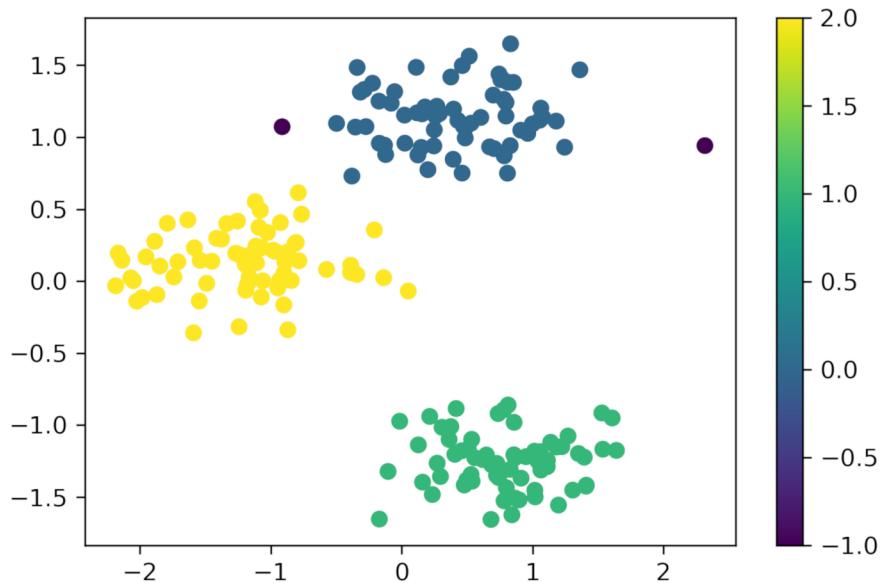
0, 2, 0, 2, 0, 2, 1, 1, 2, 1, 0, 1, 0, 0, 1, 1, 2, 0, 2,
1, 2, 2, 1, 2, 0, 1, 2, 2, 0, 2, 2, 2, 1, 1, 0, 0, 1, 0, 2, 2,
1, 1, 1, 2, 2, 1, 0, 0, 1, 1, 2, 2, 0, 2, 0, 1, 1, 1, 1, 2, 1,
1, 2, 1, 1, 1, 2, 2, 1, 1, 2, 2, 1, 0, 1, 1, 2, 2, 2, 1, 2,
0, 0, 0, 2, -1, 2, 2, 2, 1, 2, 0, 0, 2, 1, 0, 1, 1, 2, 0, 2,
1, 1, 2, 2, 1, 0, 0, 1, 1, 0, 0, 0, 0, 2, 2]

```

The DBSCAN algorithm has designated which cluster each data point belongs to by assigning them an integer labels. Notice in the plot below that the labels assigned to each cluster are not the same as those in the previous plot. Clustering labels are not classes but rather are merely to indicate which data points belong to the same cluster. The values themselves do not matter. Two data points have been assigned values of  $-1$  which means these data points are noise. The  $k$ -means and agglomerative clustering algorithms would have assigned all data points, including outliers, to a cluster, but DBSCAN is willing to label outliers as noise.

```
[in]: plt.scatter(X_ss[:,0], X_ss[:,1], c=DB.labels_)
```

```
[out]:
```

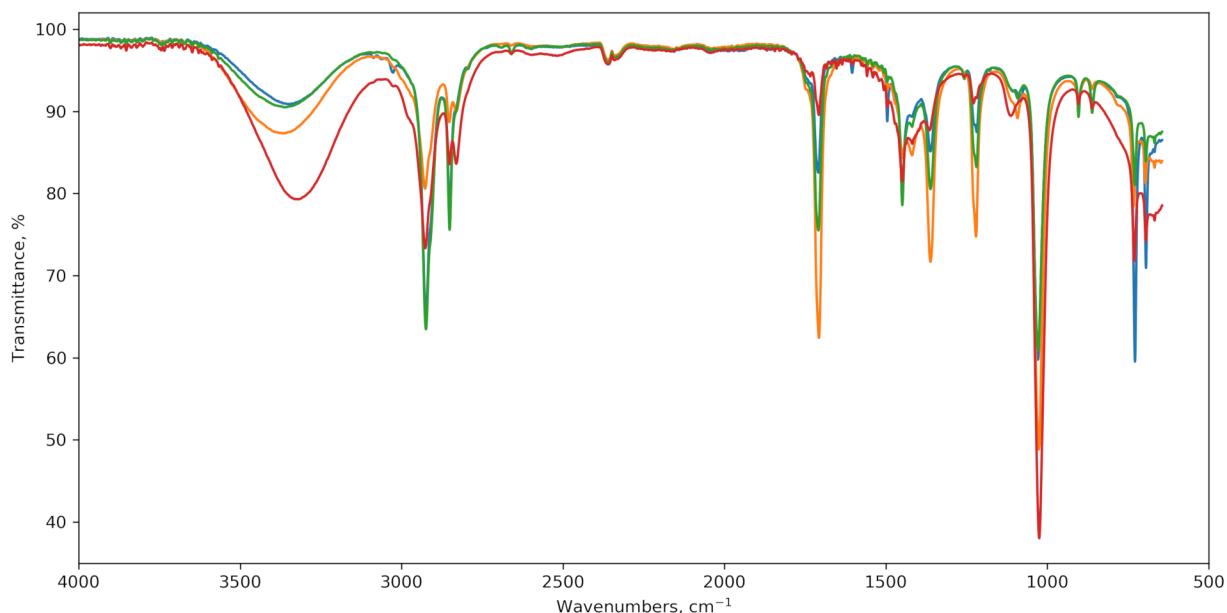


### 12.2.5 Blind Signal Separation

Blind signal separation is the processes of separating independent component signals from a mixed signal. One application is in chemical spectroscopy where a spectrum may include signals from multiple chemical compounds in a mixture. If we provide the BSS

algorithm multiple spectra of chemical mixtures where each mixture contains varying amounts of each chemical, the BSS algorithm should be able to separate the signals for each chemical component.

To demonstrate this process, we will use infrared (IR) spectroscopy data containing mixtures of acetone, cyclohexane, toluene, and methanol in random ratios.<sup>104</sup> Below are plots of four mixtures. We can see that, for example, the bands at  $\sim 3400\text{ cm}^{-1}$  and  $\sim 1000\text{ cm}^{-1}$  increase together suggesting that they originate from the same compound; this type of information can be used to discriminate which band belongs to which compound. However, instead of doing this manually, we can allow the machine learning algorithms to pick apart the spectra, and even better yet, yield complete spectra of each component.



For this task, we will use the *independent component analysis (ICA)* function called `fastICA` available in scikit-learn. The process parallels the other unsupervised learning processes above of first training the algorithm using the `fit` method followed by transforming the data using the `transform` method. First we will load the data from the files and stack them into an array where each column contains the data from a spectrum.

```
[in]: data_mix = []
```

---

<sup>104</sup> The IR spectra do not include frequencies below around  $600\text{ cm}^{-1}$  because it is often of minimal importance in chemistry and there is a disproportionate amount of noise in that region resulting in poorer performance by the BSS algorithm.

```

for file in os.listdir():
    if file.lower().endswith('csv') and
       file.lower().startswith('mix'):
        data_mix.append(np.genfromtxt(file,
                                      delimiter=',',)[:,1])
# load x-asix from the last data file
wavenumbers = np.genfromtxt(file, delimiter=',',)[:,0]

data_array_mix = np.vstack(data_mix).T
signals = np.ndarray.astype(data_array_mix, float)

```

The next step is to train and transform the data. When generating the fastICA model, it requires the number of components (`n_components`) which is four in this case. One minor drawback of this algorithm is that the user must first know the number of components in the mixed signal.

```

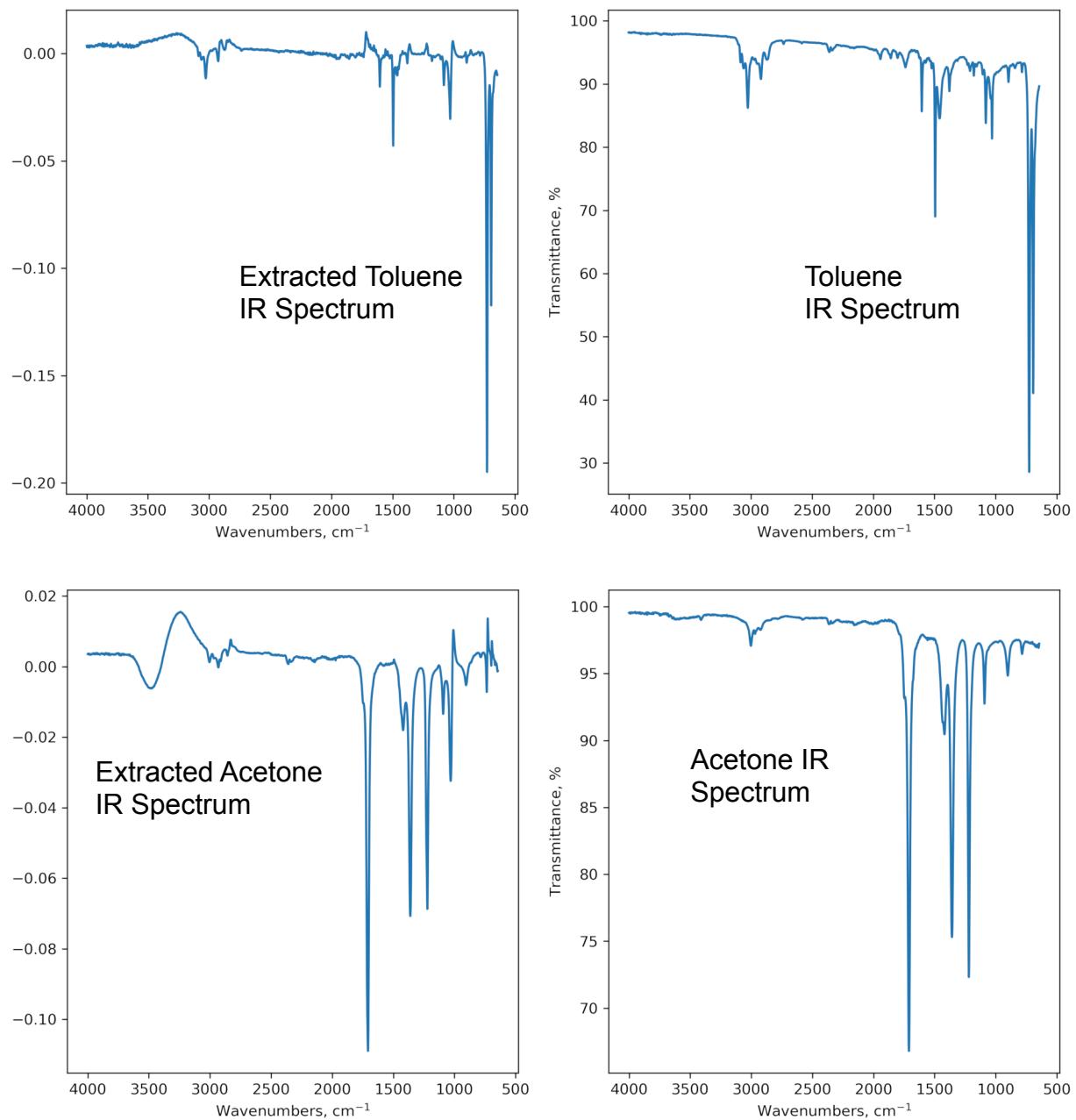
[in]: ica = FastICA(n_components=4)
       comp_sig = ica.fit_transform(S)

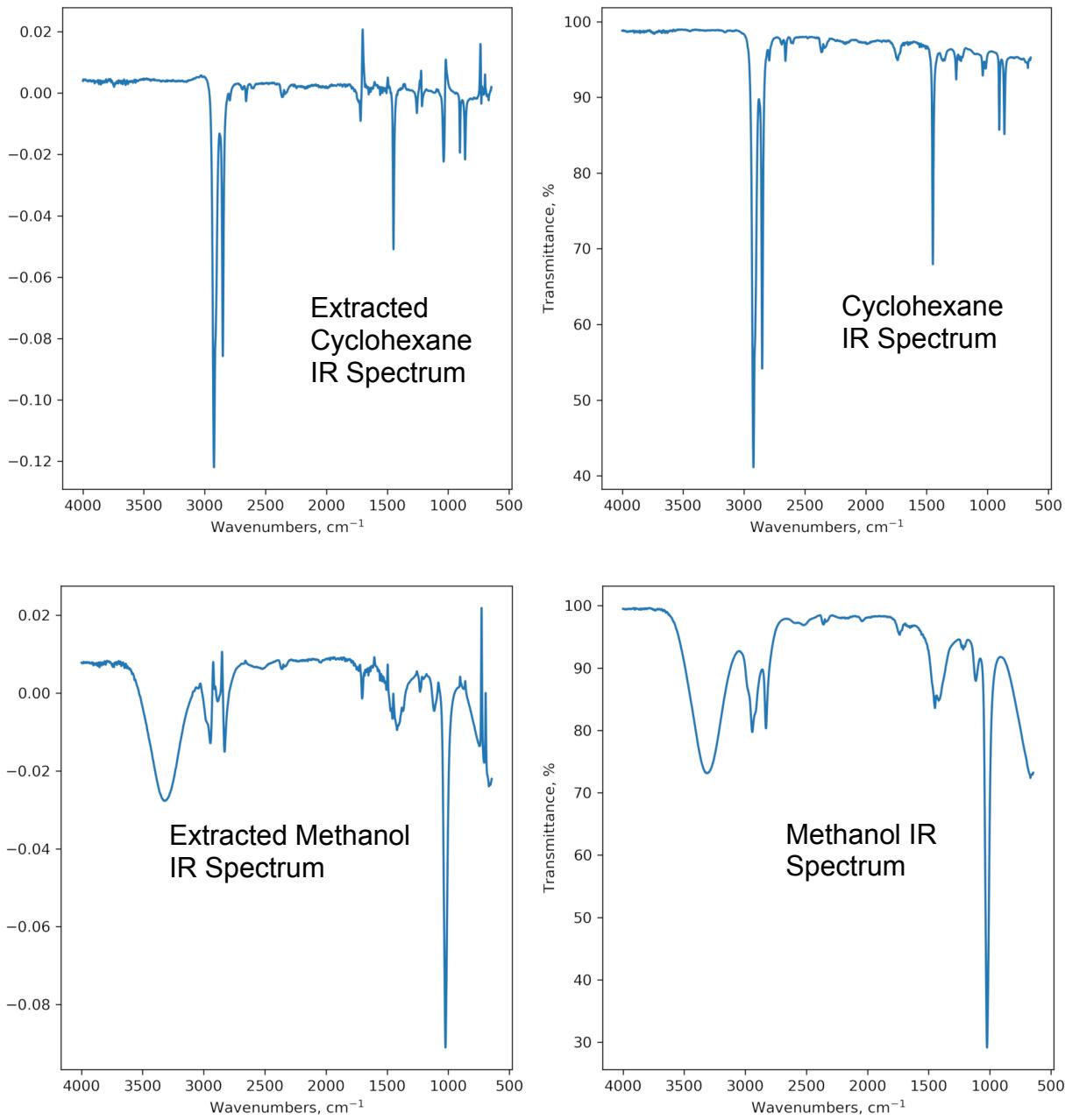
[in]: comp_sig.shape

[out]: (6961, 4)

```

You may have noticed that instead of doing the `fit` and `transform` in two steps, we used a `fit_transform` method. This method is present in many unsupervised algorithms allowing the user to perform both steps in a single function call. The resulting array `comp_sig` contains the four extracted components where each column of the array is a component. We can plot each component next to IR spectra of pure compounds collected separately to see how it performed. Remember that the BSS algorithm does not know anything about what these components are, so interpreting them or matching them to real chemical compounds is left to the user.





Overall, the fastICA algorithm did a decent job... sometimes even impressive job of picking out small features, but there are some discrepancies between the extracted and pure IR spectra. The first is that there are peaks that extend above the extracted spectra. A transmittance over 100% is not possible, but the algorithm does not know this. The y-axis scales of the extracted IR spectra also do not match the percent transmittance. While it is not shown here, sometimes the extracted components are also upside down. This is because the

mixtures are assumed to be weighted sums of the components, and a component can be negative. If this bothers you, there is a related BSS algorithm called non-negative matrix factorization (NMF) supported in scikit-learn which enforces each component to be non-negative. Finally, you may notice that there is a broad feature at around  $3400\text{ cm}^{-1}$  in the second extracted component that is not in the pure compound. This is an O-H stretch from the methanol IR spectrum showing up in the acetone spectrum. This may be the result of hydrogen-bonding between methanol and acetone breaking down the assumption that the spectra of mixtures are purely additive.

## 12.3 Final Notes

There is a saying that there is no task so simple it cannot be done wrong, and machine learning is no exception. Machine learning, like any tool, can be used incorrectly leading to erroneous or error-prone results. One particular source of error in machine learning is making predictions outside the scope of the training dataset. That is, if we train an algorithm to predict the boiling points using aliphatic alcohols, there is no reason to expect that the algorithm should be able to accurately predict the boiling points of aromatic alcohols. Another risk in machine learning is overtraining an algorithm. Some algorithms provide numerous parameters which customize the behavior, and these parameters are often used to optimize the accuracy of the predictions. The parameters can be over optimized for the training data so that the algorithm then performs worse in predicts for non-training data. This is known as *overtraining* the algorithm. In all of the excitement about how powerful and useful machine learning is, we should always keep the sources of error in mind and always remember that just because a machine learning algorithm makes a prediction does not make it true.

## Further Reading

1. Scikit-Learn Website. <https://scikit-learn.org/stable/> This is a great resource both on using scikit-learn and about machine learning algorithms implemented within.
2. VanderPlas, J. *Python data Science Handbook: Essential Tools for Working with Data*, 1st ed.; O'Reilly: Sebastopol, CA, 2017, chapter 5. A free, online version available is available by the author at <https://github.com/jakevdp/PythonDataScienceHandbook>.
3. Nallon, E. C.; Schnee, V. P.; Bright, C.; Polcha, M. P.; Li, Q. Chemical Discrimination with an Unmodified Graphene Chemical Sensor. *ACS Sens.* **2016**, 1, 26–31. This is a relatively approachable chemistry article that applies scikit-learn to a chemical problem using both supervised and unsupervised techniques.

## Exercises

1. Import the data file *ROH\_data.csv* containing data on simple alcohols and train a random forest algorithm to predict whether or not an alcohol is aliphatic. Remember to split the dataset using `train_test_split()` and evaluate the quality of the predictions.

Note: Some densities are missing in this dataset, so you will need to decide how to deal with this missing data.
2. Open the file titled *mixed\_NMR.csv* which contains three  $^1\text{H}$  NMR spectra. Each spectrum (columns) is a mixture of three chemical compounds in different ratios (artificially generated). Use fastICA to separate out three pure  $^1\text{H}$  NMR spectra of each component. Compare your separated spectra to the pure NMR spectra in *pure\_NMR.csv*.
3. Import the file titled *clusters.csv* containing unlabeled data with two features.
  - a) Use the DBSCAN algorithm to predict clusters for each datapoint in the set. Plot the data points using color to represent each cluster.
  - b) Use the k-means algorithm (`sklearn.cluster.KMeans`) to predict clusters for each datapoint in the set. This may require you to visit the Scikit-Learn website to view the documentation for this algorithm and function. Plot the data points using color to represent each cluster.

4. Load the handwritten digits dataset using the `sklearn.datasets.load_digits()` function.
  - a) Reduce the dimensionality of the dataset to two principle components and visualize it. Color the markers based on the category. You will need to import PCA from `sklearn.decomposition`.
  - b) Train the Gaussian Naive Bays algorithm to classify the digits. Be sure to evaluate the effectiveness using a testing dataset. Import GaussianNB from `sklearn.naive_bayes`.



# *Chapter 13*

## *Command Line & Spyder*

Up to this point, we have been running all of our Python scripts through the IPython environment from either a Jupyter notebook or a Python interpreter. A third way to run Python code is to save it as text files and run the code from the computer's command prompt. The advantage of this approach is that it is more practical for larger scripts and more convenient for doing repetitive tasks like reformatting instrument data. You will need access to the command prompt to run your Python script. If you are using Linux or Mac, launch the Terminal to get access to the Bash command line, and if you are on Windows 10, you will first need to activate the Bash command line before proceeding.

### **13.1 Bash Scripting Primer**

Bash is the name for the command line terminal used in macOS and unix-like systems such as Linux and the BSD family. It is also now available for Windows 10 but must first be turned on. The Bash command line allows users to perform a wide array of tasks from installing and running software to file management. In section 13.2, you will learn to run Python scripts from the command line, but before you can run a script, you need to be able to navigate your file system and find your Python scripts. This section is a short primer on navigating the file system through the Bash command line.

#### **13.1.1 Directory Name & Contents**

When you open the terminal, you are greeted with a line that looks something like the following where Comp is your computer name and Me is your account's user name. After the \$ sign is where you type your commands.

Comp : ~Me\$

From here, you can navigate your file system. The first thing you will want to know is where on the file system you are currently looking. This is known as the *current working directory*, which can be determined with the command `pwd` (print working directory).

```
$ pwd  
/Users/Me
```

This means that we are currently in the home directory for the user Me.

To determine what is in the directory, we can have its contents listed using the `ls` command.

```
$ ls  
Applications Documents Movies Public  
Downloads Music anaconda  
Desktop Library Pictures seaborn-data
```

You may see files listed in the terminal that you cannot see when manually looking in a folder. This is normal. Computers often contain invisible files for items like icons, and it is best not to alter or delete these invisible files.

### **13.1.2 Changing Directory**

To change the current working directory, use the `cd` command. This can be used either incrementally by stepping one directory at a time or by providing the full path name such as `/Users/Me/Documents/Scripts/`.

```
$ cd Desktop
```

This only allows the user to navigate into folders. To back out of a folder, `cd..` (with two periods) is used.

```
$ cd..
```

There is certainly much more that can be done with Bash scripting, but this is enough of a foundation for you to find and run scripts as we will do below.

## **13.2 Running Scripts**

Now that you know the basics of the Bash command line, we can now run our first script. Open a text editor of your choice. It is recommended not to use your regular word processor (e.g., Word, LibreOffice, Pages, etc...) as it may save extra formatting in any text file generated. Most computer operating systems come with a very basic text editor suited for

writing code. Save the following as a text file titled `first_script.py`. The `.py` extension does not do anything to the file; it just indicates to other software that this text file is a Python script.

```
import random
rdn = random.randint(0,100)
print(rdn)
```

Next, open the terminal and navigate to the directory (i.e., folder) containing the above script file and type the following into the terminal.

```
$ python first_script.py
66
```

You just ran your first script from the command line! The output only includes what you print in the Python script. One key difference between a script run in the command line and Python code run in a Jupyter notebook is that when running from the command line, if you want something displayed, you need to explicitly instruct this action using the `print()` function. In contrast, the Jupyter notebook automatically prints the output of calculations that are not assigned to variables.

An alternative way to run the above file without having to navigate to the folder is to provide the file with the full (absolute) path like is shown below

```
$ python /Users/Me/Desktop/first_script.py
98
```

This might seem like a lot of typing. One handy shortcut is to type `python` and then drag-and-drop the file into the Bash window. This will result in the file path and name being automatically pasted into the command line.

```
$ python /Users/Me/Desktop/first_script.py
65
```

### 13.3 Additional Inputs

There are often times when running a script from the command line that you want to be able to include addition inputs or information to the Python script. This may come in the form of a user input or extra files. Below are ways to accomplish this making your script more interactive.

### **13.3.1 User Inputs**

In the event you want to user to be able to input values, Python includes an `input()` function that prompts the user to provide information. For example, if we want to write a script to calculate molecular weights of simple hydrocarbon molecules based on the number of hydrogen and carbon atoms, it would be helpful to allow the user to input the number of hydrogen and carbon atoms instead of altering the script itself. The argument inside the `input()` function is what is displayed in front of the user to prompt an input. It is important to note that the `input()` function provides the user input as a string. Being that we are expecting integers, we need to convert these strings to integers before calculating the molecular weight of the molecule as has been done below.

```
H = input('H = ')
C = input('C = ')

MW = int(H) * 1.01 + int(C) * 12.01
print(MW)
```

Save the above script in a text file named `MW.py` and run it. You are prompted to provide the number of hydrogens and carbons before a molecular weight is calculated and printed.

```
$ python MW.py
H = 4
C = 1
16.05
```

### **13.3.2 `sys.argv`**

Another approach to allowing the user to provide additional information is to provide all the required information in the same line as calling the script. For example, when running the above hydrocarbon molecular weight script above, you might expect it to look like the following.

```
$ python MW.py 4 1
16.05
```

We can instruct Python to grab the information behind the script file name using the `argv` function from the `sys` module. This function brings all information after `python` as a list which can be accessed using indexing. The above input generates the following list from `sys.argv`.

```
['MW.py', '4', '1']
```

Now it is just a matter of indexing and converting strings to integers as is done below.

```
import sys  
  
H = sys.argv[1]  
C = sys.argv[2]  
  
MW = int(H) * 1.01 + int(C) * 12.01  
print(MW)
```

Now we can run the script as follows.

```
$ python MW.py 8 3  
44.11
```

The above method is ideal from accepting file names and extension as they can be dragged into the terminal more easily than typed. The down side to this approach is that the user needs to be aware of what information to provide the script and in what order. This is analogous to the difference between a keyword argument and positional argument in a function.

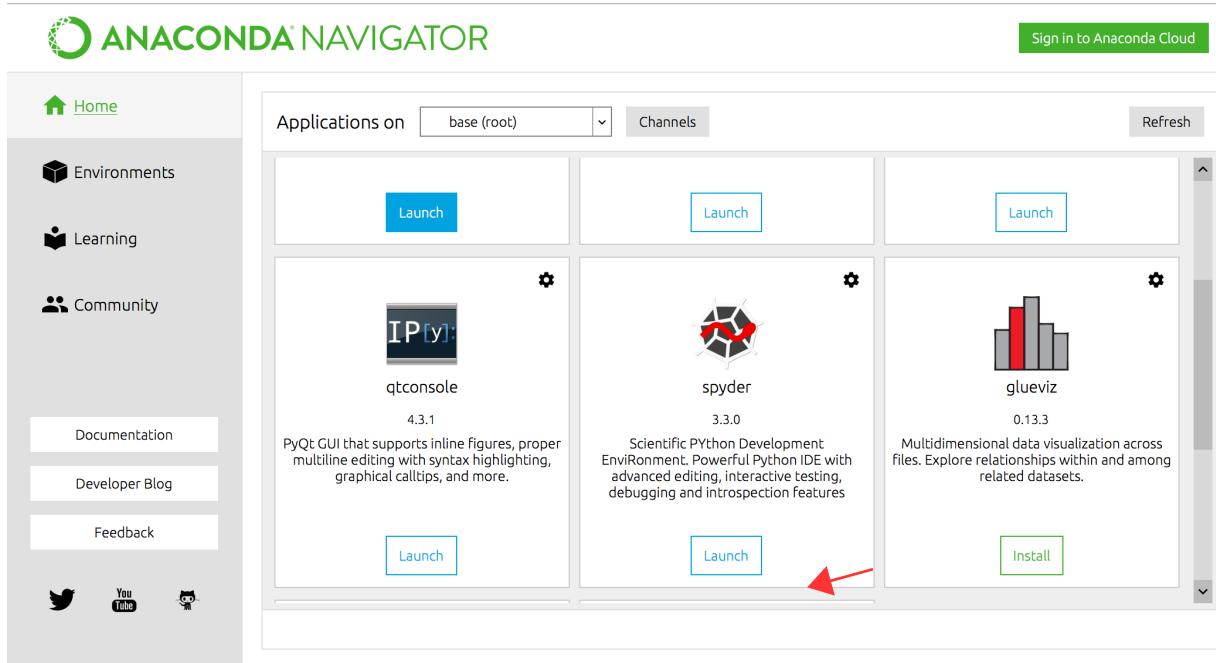
## 13.4 Spyder

While using a text editor to write your scripts works just fine, you may long for some of the features of Jupyter notebooks like how it automatically color codes text based on syntax and provides easy access to function docstrings. To get some of these features back, you can use an Integrated Development Environment (IDE). There are many to choose from, but here we will address Spyder (Scientific Python Development Environment) as it is specifically tailored to scientific applications and comes with the Anaconda installation of Python.

There are two methods of launching Spyder. The first is to type `spyder` in the terminal.

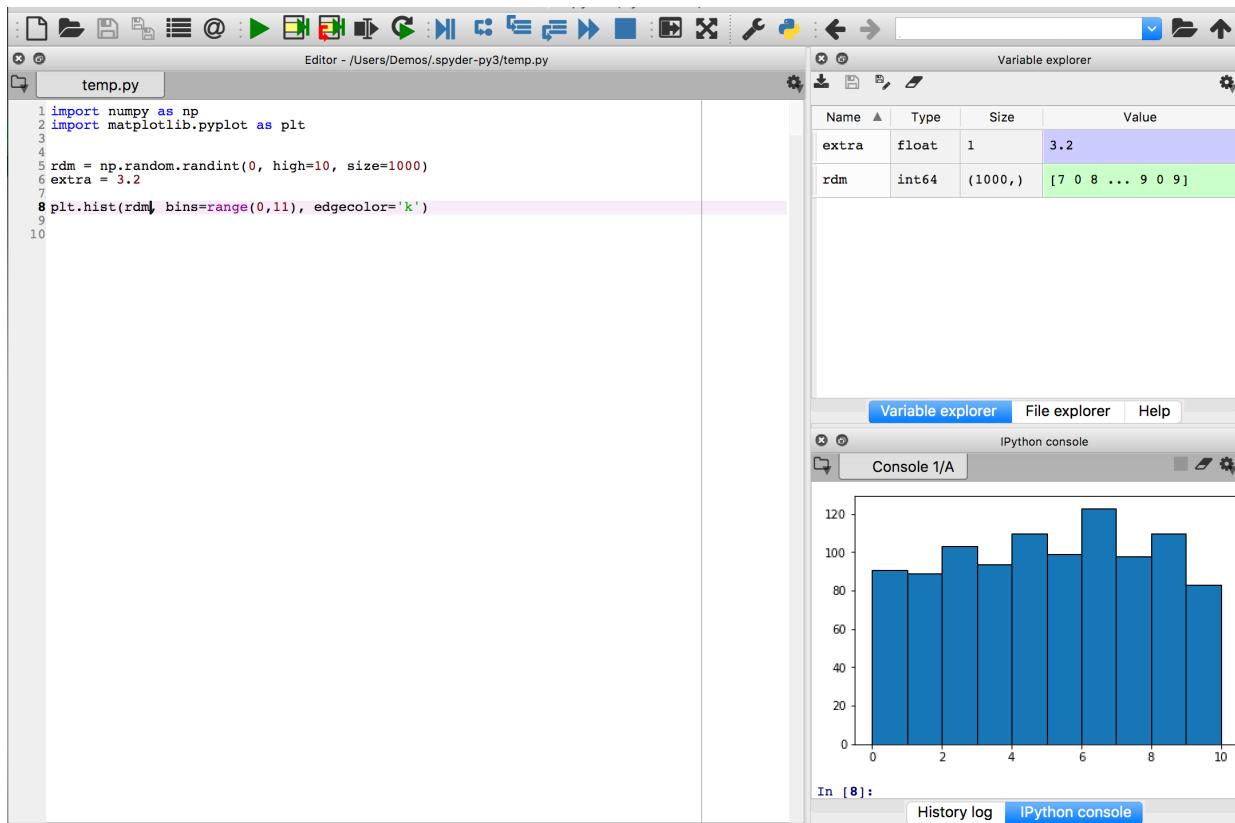
```
$ spyder
```

The second method is to press the launch button for Spyder in Anaconda Navigator (Figure 13.1). The latter method is often slower because it requires that Navigator be first launched.



**Figure 13.1** A screenshot of the Anaconda Navigator application launcher window.

Once Spyder has launched, you will be greeted by an interface divided into three windows (Figure 13.2). The left window is a text editor where code is written. Like the Jupyter notebook, it color codes your Python code based on syntax and provides docstrings and helpful notices. To run the code written here, you can either save it as a text file and run it as described above, or you can press the run button (►) at the top of the window. The latter approach is particularly handy during the development phase of a script as it allows you to quickly test and modify your script without having to jump between Spyder and the terminal. The smaller window on the bottom right is a Python terminal where you can test out code and see the output of your code if you run your code inside Spyder. The top right window is useful as a file navigator and as a variable explorer depending upon the tabs you choose. In Figure 13.2, it is a variable explorer which shows each variable in memory and what it contains. This is powerful tool when debugging code as it allows you to quickly see what the code is doing and where things are not working.



**Figure 13.2** The Spyder interface with the text editor (left), variable explorer (top right), and interpreter (bottom right).

So when should you use a Jupyter notebook and when should you use Spyder? The decision is often a matter of preference, but if you are doing interactive data analysis, Jupyter notebooks are typically the better choice. This is particularly true if you need to share your analysis and results with others. If you are writing large blocks of code, Spyder is likely a better choice of environment. As an example, if you wish to perform complex mining of information from an external dataset and then analyze the resulting information, you might want to write the data mining code in Spyder and then run the data analysis in a Jupyter notebook.

## Further Reading

1. Spyder Website. <https://www.spyder-ide.org/>

## Exercises

Complete the following problems by writing a Python scripts either in a text editor or Spyder and run them from the terminal. Jupyter Lab (newest version of Jupyter) includes a text editor if you wish to use it, but do not use a Jupyter notebook for any of these problems!

1. When an electron in a hydrogen atom relaxes from a higher to a lower energy orbital, a photon is released with the energy described by the below equation. Write and run a Python script that prompts the user to input the initial and final principle quantum numbers ( $n$ ) and outputs the wavelength ( $\lambda$ ) of light emitted.

$$E = 2.18 \times 10^{-18} J \left( \frac{1}{n_i^2} - \frac{1}{n_f^2} \right) \quad E = \frac{hc}{\lambda}$$

2. In the folder titled *Problem 2 Kinetic Data* you will find synthetic data for the conversion of A → P. Both datasets are for first-order reactions.
  - a) Write a Python script that accepts the name of a single data file like below and outputs the rate constant ( $k$ ) for the data. Test it on both datasets. For the script to find the file, it needs to either be in the same directory as the data file or be provided the absolute path to the file.

```
$ python script.py kinetic_data_1.csv
```

or

```
$ python script.py  
/Users/Me/Desktop/kinetic_data_1.csv
```

- b) Modify the above script to print out the rate constant for all data sets in the folder. This script will accept the folder name instead of the file name. Remember to use the `os` module described in section 2.4.1.