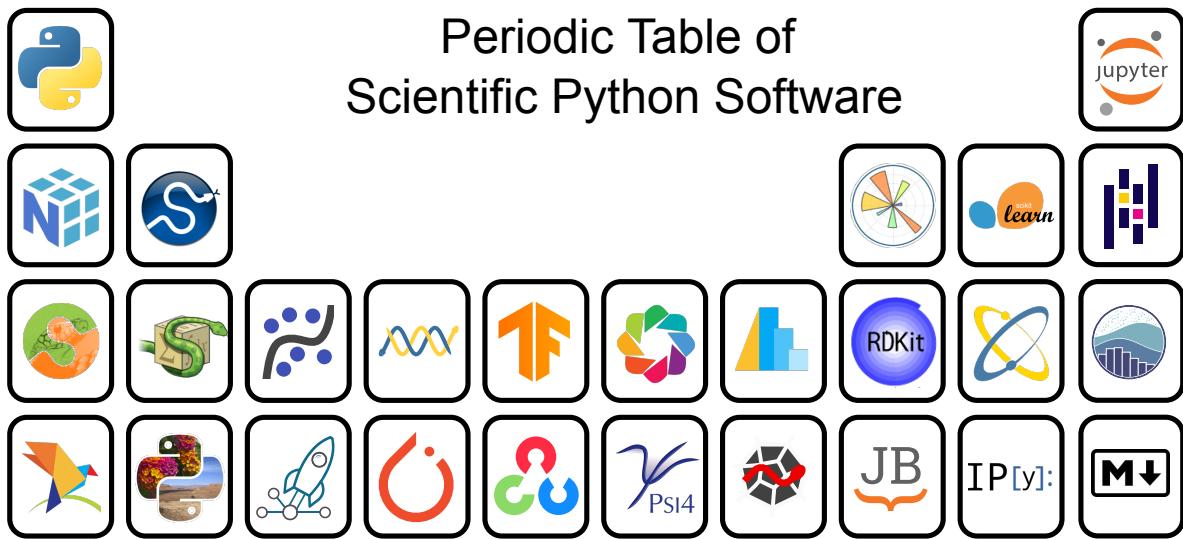


Scientific Computing for Chemists with Python

Contents

- An Introduction to Programming in Python with Chemical Applications
- Organization of Book
- Chapter and Exercise Data
- Exercise Answers
- Code and Software Versions
- Acknowledgements



An Introduction to Programming in Python with Chemical Applications

Scientific computing utilizes computers to aid in scientific tasks such as data processing

and digital simulations among others. The well-developed field of computational chemistry is part of scientific computing and focuses on utilizing computations to simulate chemical phenomena and calculate properties. However, there is less focus in the field of chemistry on the data processing side of computing, so this book strives to fill this void by introducing the reader to tools and methods for processing, visualizing, and analyzing chemical data. The tools employed in this book are the powerful and popular combination of Jupyter notebooks and the Python programming language. No background beyond first-year college chemistry and occasionally some very basic spectroscopy (for advanced chapters) is assumed for most of this book. This book starts with a brief primer on Jupyter notebooks in chapter 0 and computer programming with Python in chapters 1 and 2. If you already have background in these tools, feel free to skip ahead. The rest of the book dives into applications of Python to solving chemical problems. Python and Jupyter were chosen for a variety of reasons including that they are:

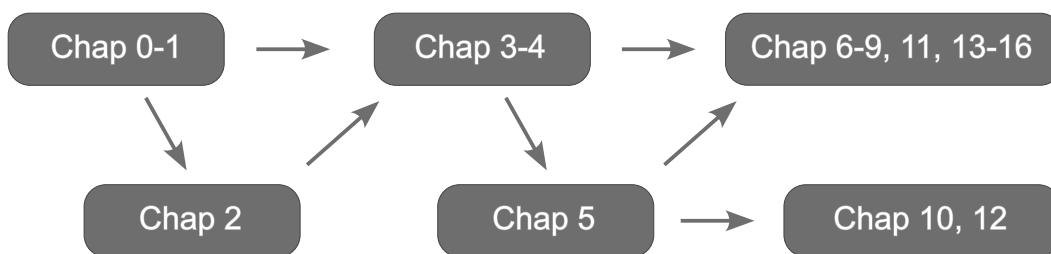
- Relatively easy to use and learn
- Powerful and well-suited for solving chemical problems
- Free, open-source software
- Cross platform (e.g., runs on Windows, macOS, and Linux)
- Supplemented with numerous, specialized libraries for handling specific types of data or problems (e.g., machine learning)
- Supported by a helpful and welcoming community

Learning to use a number of popular Python scientific libraries to solve chemical problems is one of the themes of this book. A Python library can be thought of as a tool pack with premade functions for performing common tasks in scientific data processing, analysis, and visualization. For example, the matplotlib library provides a variety of functions for creating a wide range of plots while the scikit-learn library contains functions and resources for machine learning.

Organization of Book

This book is organized in order of more fundamental topics first, but not every earlier chapter is a prerequisite for all subsequent chapters. Chapter 0 provides a quick introduction to Jupyter notebooks 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 introduces plotting and visualization, and chapter 4 covers the NumPy library. Both of these chapters are used heavily in this book and should 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 their schedule could skip this chapter, but be aware that it is heavily utilized in chapters 10 and 12. However, chapters 10 and 12 should be largely readable by someone who is not familiar with pandas or at least has read sections 5.1-5.2. Chapters beyond chapter 5 are mostly applications or cover libraries for very specific applications such as image processing, machine learning, bioinformatics, or optimization. Chapters 6-16 are designed to be modular, so after getting through chapters 0-5, these subsequent chapters can be covered in any order depending up the reader's needs and interests.



Below is a listing with brief descriptions of the chapters.

Chapter Number	Description
Chapter 0	Short 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	NumPy library which is the foundation of much of the scientific Python ecosystem
Chapter 5	Pandas data analysis library
Chapter 6	Basic signal processing in Python including finding peaks, smoothing data, and fitting/interpolation among other topics
Chapter 7	Image processing using the NumPy and scikit-image libraries
Chapter 8	Symbolic math and other more advanced mathematics in Python
Chapter 9	Simulating physical and chemical processes in Python
Chapter 10	Seaborn plotting library
Chapter 11	Interactive plotting with Altair
Chapter 12	NMR processing and simulations with nmrglue and nmrsim
Chapter 13	Machine learning using the scikit-learn library
Chapter 14	Using functions from the <code>scipy.optimize</code> module to perform minimizations, curve fitting, and root finding
Chapter 15	Cheminformatics with RDKit
Chapter 16	Bioinformatics with Biopython and nglview

Chapter Number	Description
Chapter 17	Writing Python scripts using Spyder and running them from the command line
Appendix 0	IPython widgets for interactive notebooks
Appendix 1	Remote requests for accessing online databases
Appendix 2	Visualizing atomic orbitals
Appendix 3	Uncertainty propagation made easier
Appendix 4	Regular Expressions

One of the goals of this book is to provide a streamlined introduction to Python and its scientific libraries in order to allow the reader 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 a selection 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. In an effort to make this text usable in a wide range of courses, there is little in-depth analysis of the data. This book instead focuses more on how to work with the data and leaves the analysis to the individual instructors.

Chapter and Exercise Data

Any data file(s) referred to in the chapters or end-of-chapter exercises can be found in the [data](#) folder in the same directory as the chapter's Jupyter notebook. Alternatively, you can

download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button. The latter option is recommended for those who do not use Git or GitHub.

Exercise Answers

Copies of exercise answer keys are available for **instructors** upon request. To obtain copies, please [email](#) the author using your **school email address**.

Code and Software Versions

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.

The code in this version of the book has been most recently tested with the following software versions unless otherwise noted but will likely work with other versions.

- Python – 3.12.7
- Jupyterlab – 4.4.4
- NumPy – 2.2.6
- SciPy – 1.16.0
- Pandas – 2.3.0
- Matplotlib – 3.10.3
- Seaborn – 0.13.2
- Scikit-image – 0.25.2
- Scikit-learn – 1.7.0
- Sympy – 1.14.0
- nmrglue – 0.11
- nmrsim – 0.6

- Spyder – 5.4.2
- Biopython – 1.85
- Nglview – 3.1.2
- RDKit – 2025.3.3
- pybaselines – 1.2.0
- requests – 2.32.4
- ipywidgets – 8.1.7
- uncertainties – 3.2.3
- Altair – 5.5.0

Acknowledgements

This book took a substantial time to write along with the time and effort in developing the curriculum. Thank you to those who supported and encouraged me along the way. Finally, thank you to the following people for proof reading or reporting errors. Reports of additional errors are welcome on GitHub or in an [email](#). Please do not email files to the author but rather include your error report in the body of the email.

- Wesley A. Deutscher helping collect some example data
- M. Roarke Tollar providing feedback and reporting typos in chapters 0 and 1
- Andrew Klose providing feedback and reporting typos in chapter 12 and an idea for an exercise
- Harrison Kuhn for identifying a code error in chapter 11
- nzjakemartin (GitHub handle) identifying a type in the weighted average equation
- Paul A. Craig identifying a typo in the Python code
- Patrick Coppock providing feedback
- Zachary M. Schulte for providing feedback and reporting typos in chapter 14
- Yuthana Tantirungrotechai for reporting errors in chapter 6
- @Avalanchian for reporting error in chapter 10 data/elements_data.csv file
- Matthew A. Kubasik for reporting an error in chapter 8
- Geoffrey M. Sametz for help with nmrsim
- Ryan Schulte for reporting typos

- Robert Belford for reports typos
- Filippo Muzzini for reporting a missing data file in chapter 6

Chapter 0: Python & Jupyter Notebooks

Contents

- 0.1 Python
- 0.2 Software
- 0.3 Using Jupyter Notebooks
- 0.4 Markdown
- 0.5 Comments
- 0.6 Overview of Python Scientific Libraries
- Further Reading

0.1 Python

Python is a popular programming language available on all major computer platforms including macOS, Linux, and Windows. It is a scripting language which means that 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](#). 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 offering or updating 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 scraping, 3D graphics, and more. As 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 Jupyter notebooks as a medium for running and sharing Python code. More details are give below on Jupyter notebooks, but for now, know that they provide interactive environments ideal for scientific computing. In addition, we will use a variety of free, open source libraries to provide collections of useful functions for scientific data processing, analysis, and visualization. Think of a library as an add-on or tool pack for Python, and there are many to choose from.

0.2 Software

The first step is to get access to the software which includes Python, Jupyter notebooks, and all the libraries/packages used in this book; and there are multiple options for accomplishing this. We will cover two common options below - this includes either installing the software on your own computer using Anaconda or using Google Colab to run the software from a Google server. Both are relatively simple to set up and have different advantages. Some of the major advantages of each are listed below.

Install Software	Google Colab
It's free	It's free
Faster execution of code	No software installation required
Not dependent on an internet connection	Uses Google's computing resources to run calculations, not yours
No accounts or registration required	Easier for multiple people to collaborate on the same notebook
Can have >5 notebooks open at any given time	

You only need to use one of the above options, but you can always switch later on if you want because both use the same notebook files to store your work. Go ahead and follow the instructions for **one** of the following.

0.2.1 Install Software on Computer

There are many ways to install the software on your computer. The faster and more convenient approach is to use an installer that brings nearly all the software with it. The Anaconda installer (link below) provided for free by Continuum Analytics is a very popular option that does exactly this. There are other installation options available, but the instructions in this book for launching applications assumes Anaconda. When installing the software, be sure to choose **Python 3** as this is the current version. While some applications still support Python 2, it is technically legacy with support being gradually removed. As of the time of this writing, multiple major projects in the scientific Python ecosystem no longer support Python 2, so it is likely in your interest to be on Python 3. You are strongly encouraged to install the most recent version of Python.

[Anaconda Installer](#)

The Anaconda installer above brings almost everything you need. Any software used in this book that is not installed by default through the above method is addressed in its respective chapter. 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 libraries available to be installed as shown in Figure 1. To install a library, check the box next to it and click the **Apply** button that appears on the bottom right. Anaconda will install it and anything else that is required for the new library to work properly. To update a library, select **Upgradable** from the pull-down menu, select the package(s) you want to update, and click **Apply**.

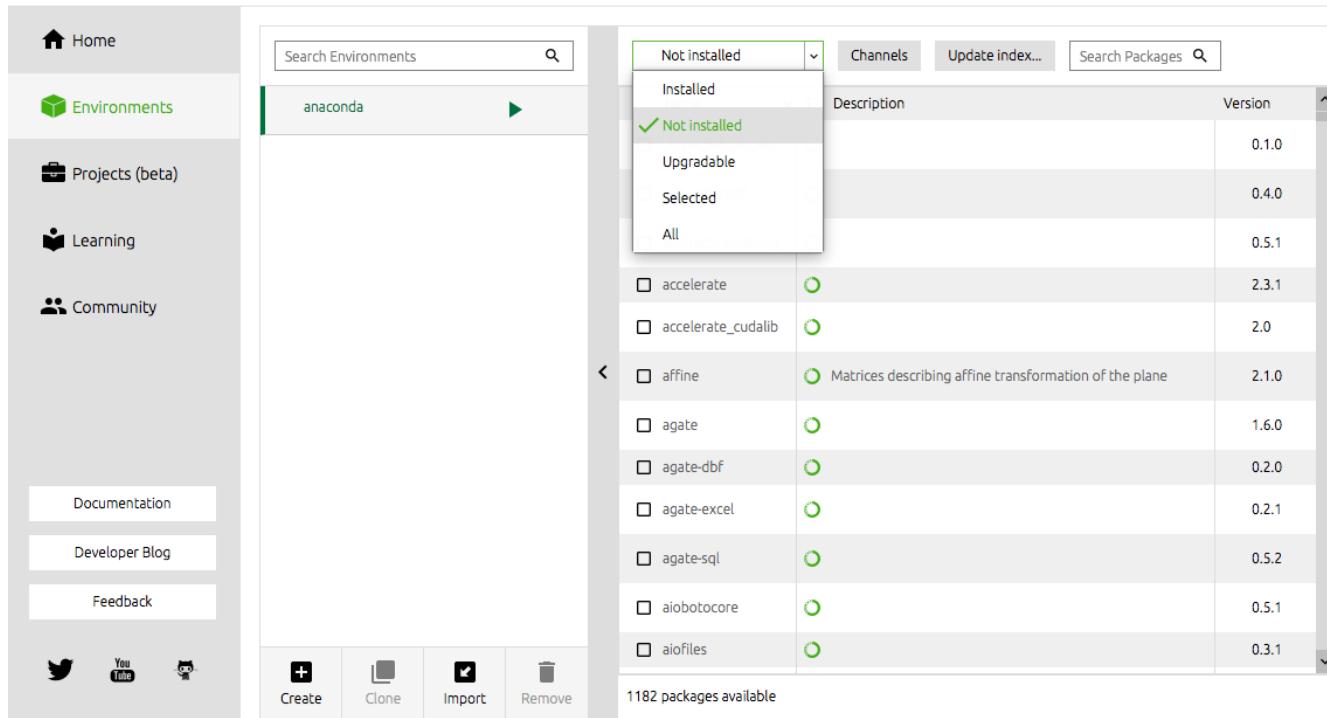


Figure 1 Installing additional libraries using Anaconda-Navigator.

Alternatively, you can install many of the libraries using the Terminal. To launch the Terminal, either use your computer's build-in terminal or launch JupyterLab (see [section 0.3.1](#)), select the **Launcher** tab in JupyterLab, and click **Terminal** (Figure 2).

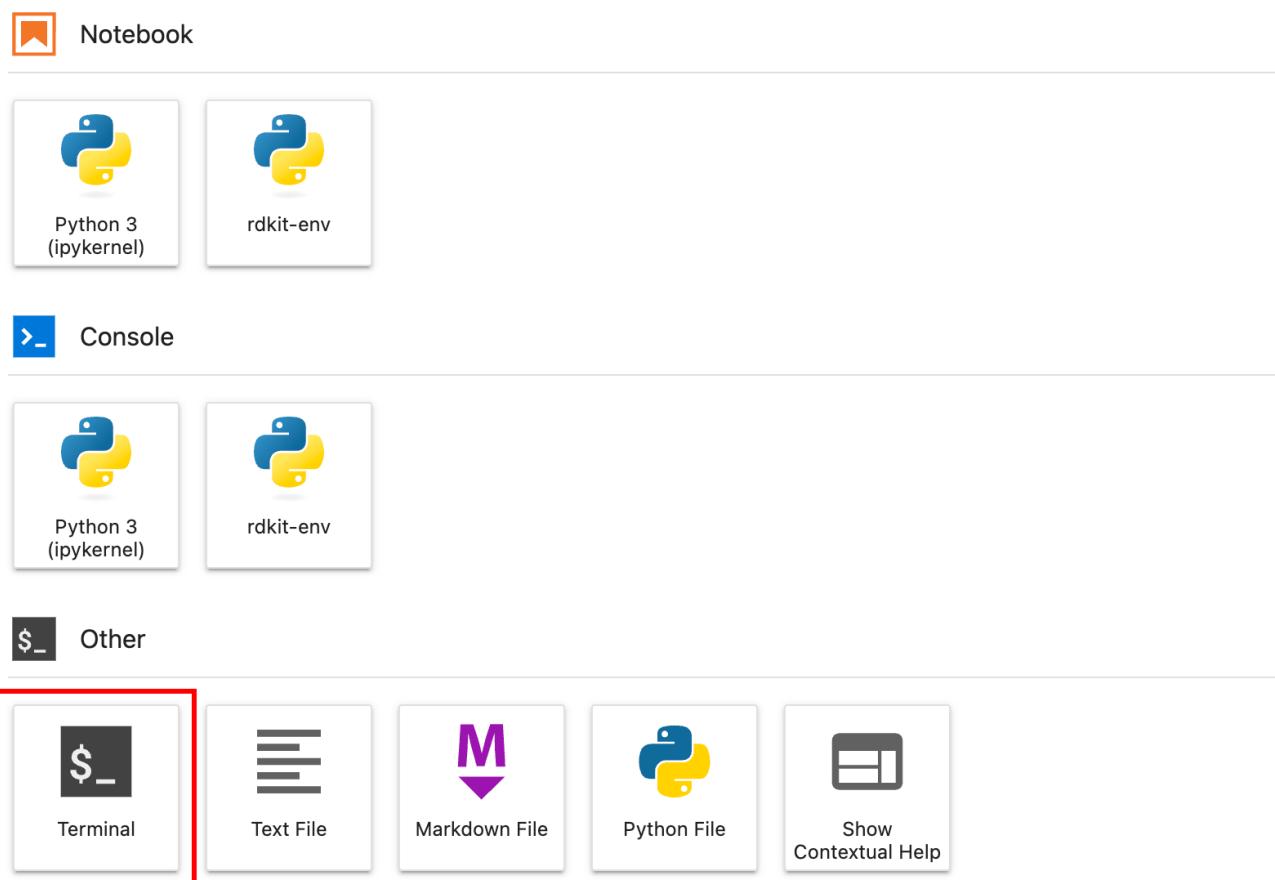


Figure 2 Launching the terminal using the JupyterLab launcher.

From here, you can install various libraries using either of the below commands where <library> is the name of the library to install.

```
pip install <library>
```

or

```
conda install <library>
```

Most libraries can be installed using either of the above commands, but a few can only be installed with one. You should do a quick internet search to see which is the preferred method for a particular library before installing it. The `pip list` or `conda list` command will display a list of all libraries currently installed with version numbers. To perform an update, the following two commands may be used for many libraries. Again, check to see which is preferred for a particular library.

```
pip install <library> --upgrade
```

```
conda update <library>
```

0.2.2 Google Colab

The other option we'll cover is to run the software on a Google server using Google Colab. You don't need to install any software for this option, but you will need a free Google account. If you have a gmail account or your institution's email is run by Google, you already have a Google account. While you could just go directly to the [Colab Page](#), we want to be able to work with data files on your Google Drive, so below are instructions for setting up Google Colab from your Google Drive.

First, log into your Google account or [create an account](#) if you don't have one already. Next, navigate to Google Drive by clicking on the **Google Apps** icon (3 × 3 grid of dots) on the top right and click **Drive** (Figure 3)

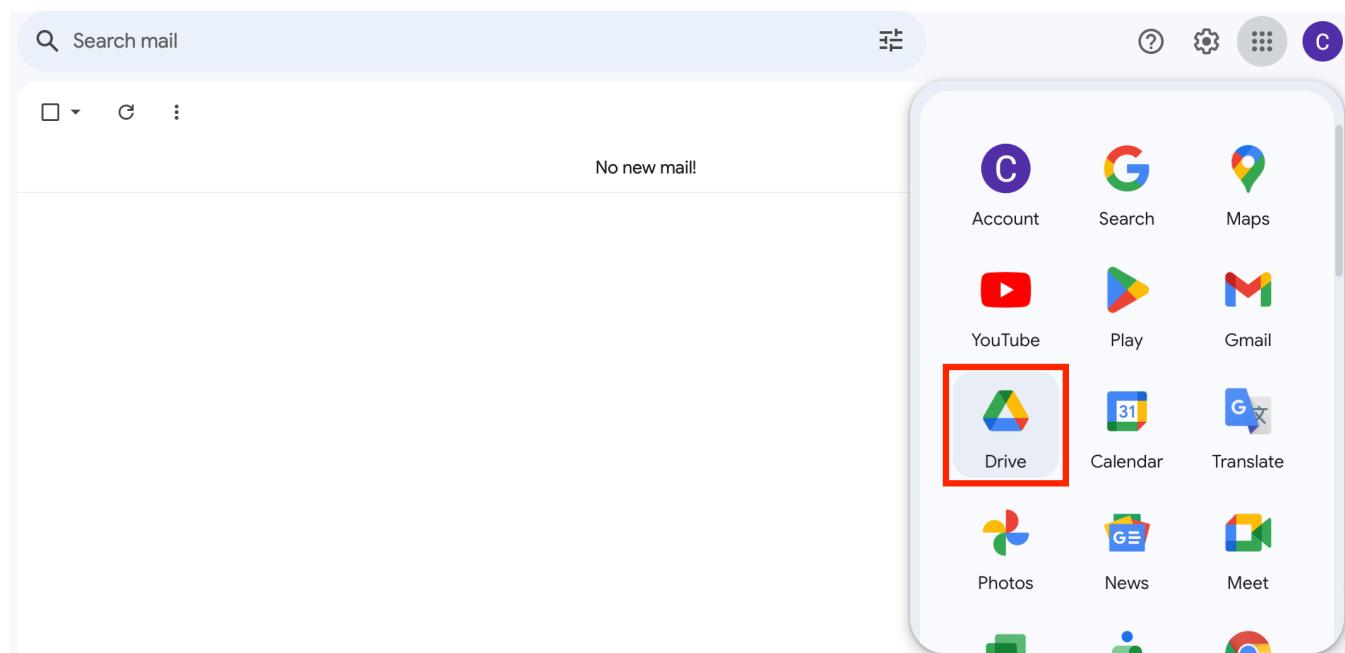


Figure 3 Accessing Google Drive

Click the **Get Add-ons** button on the center right of the window (+ icon) (Figure 4) and search for “**Google Colab**” and click **Install**.

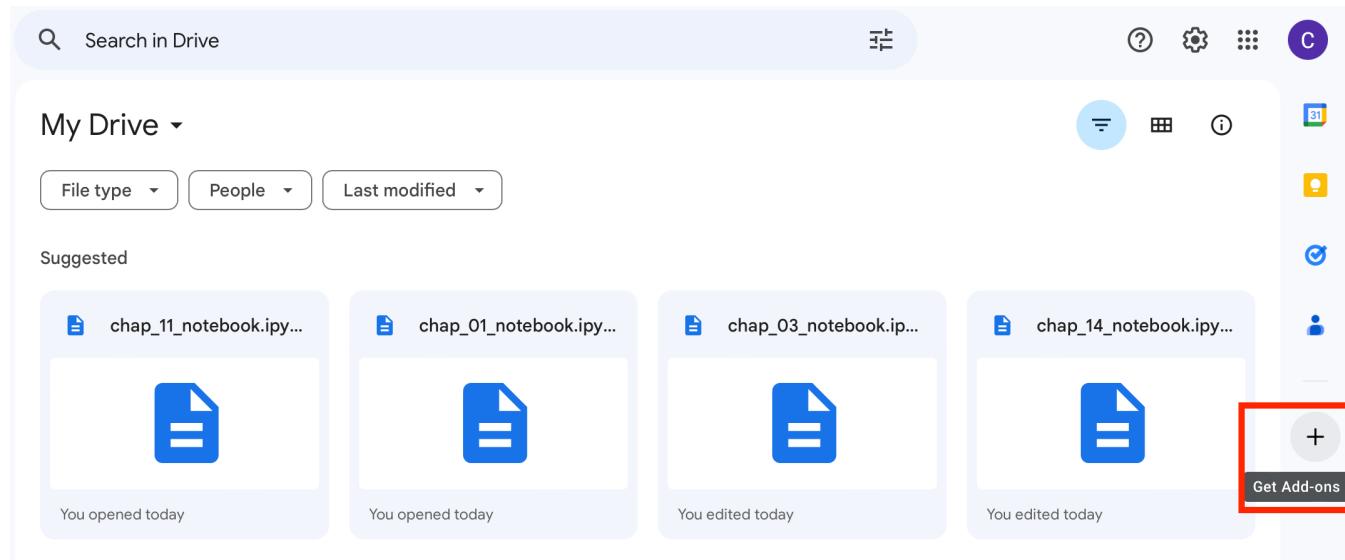


Figure 4 Accessing add-ons

Most of the libraries (see [section 0.6](#)) used in this book are already available in Google Colab by default including NumPy, SciPy, pandas, seaborn, scikit-image, and scikit-learn. If you need any additional libraries (or “packages”), you can usually install them by adding a code cell at the top of your Jupyter notebooks that looks like the following inserting the library name for `<library>`. If you need any additional libraries installed for this book, this will be addressed in the appropriate chapter.

```
!pip install <library>
```

0.3 Using Jupyter Notebooks

The Jupyter notebook (formerly known as the IPython notebook) is an electronic document designed to support interactive data processing, analysis, and visualization in a shareable 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. You can work with Jupyter notebooks either by having the Jupyter software installed on your computer or by running them on Google Colab which is Google’s implementation of Jupyter.

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 (Figure 5). 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.

File Edit View Run Kernel Tabs Settings Help

Launcher Deterministic Chemical Kinetics chap_00_notebook.ipynb Python 3 (ipykernel)

Multistep Kinetics

The following is a numerical simulation of a two-step reaction of A going to B which then reacts to form P. Again, each step is asserted to be an *elementary step*.

$$A \xrightarrow{k_1} B \xrightarrow{k_2} P$$

The initial concentration of A is 1 M while the concentrations of B and P are initially zero. The rate constants for the following reaction are set to $k_1 = 0.2 \text{ s}^{-1}$ and $k_2 = 0.3 \text{ s}^{-1}$ for the first and second steps, respectively. The rate laws for each step are shown below.

$$\text{Rate}_1 = \frac{d[A]}{dt} = k_1[A]$$

$$\text{Rate}_2 = \frac{d[B]}{dt} = k_2[B]$$

```
[11]: k1 = 0.2
k2 = 0.3

A0 = 1.0
B0 = 0.0

t = np.linspace(0, 20, 20)

# dA/dt = k1[A] and dB/dt = k1[A] - k2[B]
def rate(y, t):
    A, B = y

    dAdt = -k1 * A
    dBdt = k1 * A - k2 * B

    return dAdt, dBdt

Y = A0, B0
A, B = scipy.integrate.odeint(rate, Y, t).T

P = A0 - A - B

plt.plot(t, A, 'o-', label='A')
plt.plot(t, B, '^--', label='B')
plt.plot(t, P, 'p-.', label='P')
plt.xlabel('Time, s')
plt.ylabel('[X], M')
plt.legend()
```

Deterministic Chemical Kinetics ODEINT_v2.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

RAM Disk

Files

- content
 - drive
 - MyDrive
 - notebooks
 - Deterministic Chemi...
 - Untitled0.ipynb
 - sample_data
 - datalab
 - dev
 - etc
 - home
 - lib
 - lib32
 - lib64
 - libx32
 - media
 - mnt
 - opt
 - proc
 - python-apt
 - root
 - run
 - sbin
 - srv
 - sys
 - tmp

+ Code + Text

Notice that the rate constant for the second step was set lower than the first ($k_1 = 0.2 \text{ s}^{-1}$ and $k_2 = 0.3 \text{ s}^{-1}$). This results in the second step being the kinetic "bottleneck" for the chemical reaction. As a result, intermediate B begins to build up. However, as A is depleted and more B is generated, the rates of the first and second steps decrease and increase respectively resulting in the concentration of B going back down again.

Michelis-Menten Kinetics

The following is a simulation of Michelis-Menten enzyme kinetics where a substrate (S) first coordinates to an enzyme (E) before it is converted to product (P) and dissociates from the enzyme. The coordination of the substrate to the enzyme is reversible, with rate constants of k_1 , k_{-1} , and k_2 .

$$S + E \xrightleftharpoons[k_{-1}]{k_1} ES \xrightleftharpoons[k_2]{k_1} P + E$$

```
[ ] k1 = 0.2
kn1 = 0.2
k2 = 0.3

S0 = 1.0
E0 = 0.01
ES0 = 0.0

t = np.linspace(0, 2000, 100)

def rate(y, t):
    S, E, ES = y

    dSdt = -k1 * S * E + kn1 * ES
    dEdt = -k1 * S * E + k2 * ES + kn1 * ES
    dESdt = k1 * S * E - kn1 * ES - k2 * ES
```

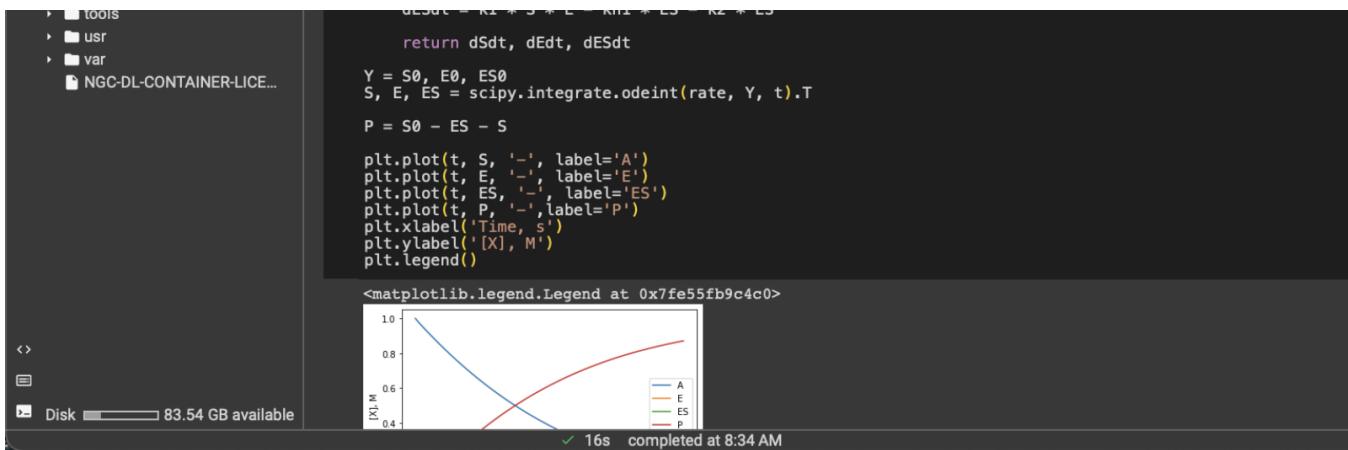


Figure 5 An example Jupyter notebook with markdown cells, code, and outputs of the code when open on Jupyter installed on a computer (top) and from Google Colab (bottom).

Jupyter notebooks can be opened and edited using Jupyter installed on your own computer or Google Colab. While the two platforms of Jupyter are similar, there are some minor differences in the location of some controls and other features. Using installed Jupyter and Google Colab are both addressed below.

0.3.1 Jupyter Installed using Anaconda

If you have Python and Jupyter installed on your computer using Anaconda ([section 0.2.1](#)), a Jupyter notebook can be launched by starting the **Navigator** application (green circle icon) and then clicking the **Launch** button under JupyterLab. Alternatively, Jupyter can be launched from the Terminal or shell by typing `jupyter-lab`. The Jupyter notebook will launch in the web browser, but this is *not* a website. An internet browser is fundamentally a fancy file viewer that displays documents and images from web servers, but it can also view files on your own computer which is what Jupyter is doing. From here, you can either select an already existing Jupyter notebook, denoted by the orange icons and `.ipynb` extension (Figure 6, left), to open it or create a new notebook by selecting **New** from the **File** menu (Figure 6, right) and selecting **Notebook**. If a popup dialogue appears titled **Select Kernel**, you should select **Python 3**.

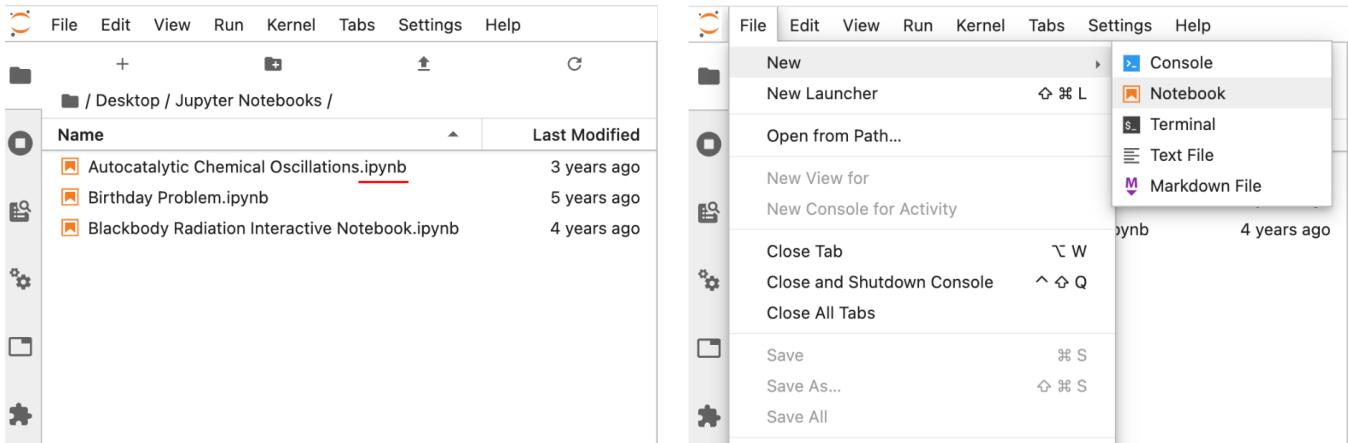


Figure 6 Launching a Jupyter notebook can be accomplished by opening a preexisting notebook from within JupyterLab (left) or launching a new Jupyter notebook from the File menu (right).

Both code and markdown cells can be run by either selecting **Run Selected Cells** in the **Run** menu, by clicking the ► button at the top of the notebook (Figure 7), 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. Markdown cells do not execute Python code and treat code like regular text.

The screenshot shows a Jupyter Notebook interface. On the left is a sidebar with a tree view of notebooks, including 'Deterministic Chemical Kin...' which is currently selected. The main area has a tab bar with 'Console 1' and 'Deterministic Chemical Kin...'. A red box highlights the play button icon (▶) at the top of the notebook area. Below it, a text cell contains a chemical reaction equation $A \rightarrow P$. A code cell [2]: contains Python code for simulating first-order decay. The output cell [2]: shows a plot of concentration [X] vs Time (s) for species A (blue circles) and P (orange squares). Species A starts at 1.0 M and decays towards zero, while species P increases from 0.0 M.

```

The concentration of starting material, A, is modeled below for the following first-order chemical reaction. The radioactive decay of an isotope is an example of a first-order reaction because the rate of decays are dependent on the amount of A. The reaction rate is described by Rate = k[A] where [A] is the concentration (M) of A, k is the rate constant (1/s), and Rate is the rate of the reaction (M/s).


$$A \rightarrow P$$


[2]: t = np.linspace(0,50,20) # time(seconds)
A_0 = 1 # starting concentration (molarity)
k = 0.1 # rate constant in 1/s

def rate_1st(A, t):
    return - k * A

A_t = scipy.integrate.odeint(rate_1st, A_0, t)

P_t = A_0 - A_t

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()

```

[2]: <matplotlib.legend.Legend at 0x151b267a20>

Figure 7 Run a selected cell in a Jupyter notebook by clicking the ▶ button at the top of the notebook or by selecting **Run Selected Cells** from the **Run menu**. The output of a code cell appears directly below the executed code cell.

To add additional cells in Jupyter, click the + above the notebook to produce another cell and then select either **Code** or **Markdown** from the pulldown menu at the top to set the cell type.

0.3.2 Jupyter using Google Colab

Google Colab is Google's flavor of Jupyter with Python. If you are using Google Colab ([section 0.2.2](#)), you can open a notebook by double clicking on the Jupyter notebook file (.ipynb extension) in your Google Drive. To create a new notebook, click the **New** button on the top left of the Google Drive window and then **More → Google Colaboratory**. (Figure 8).

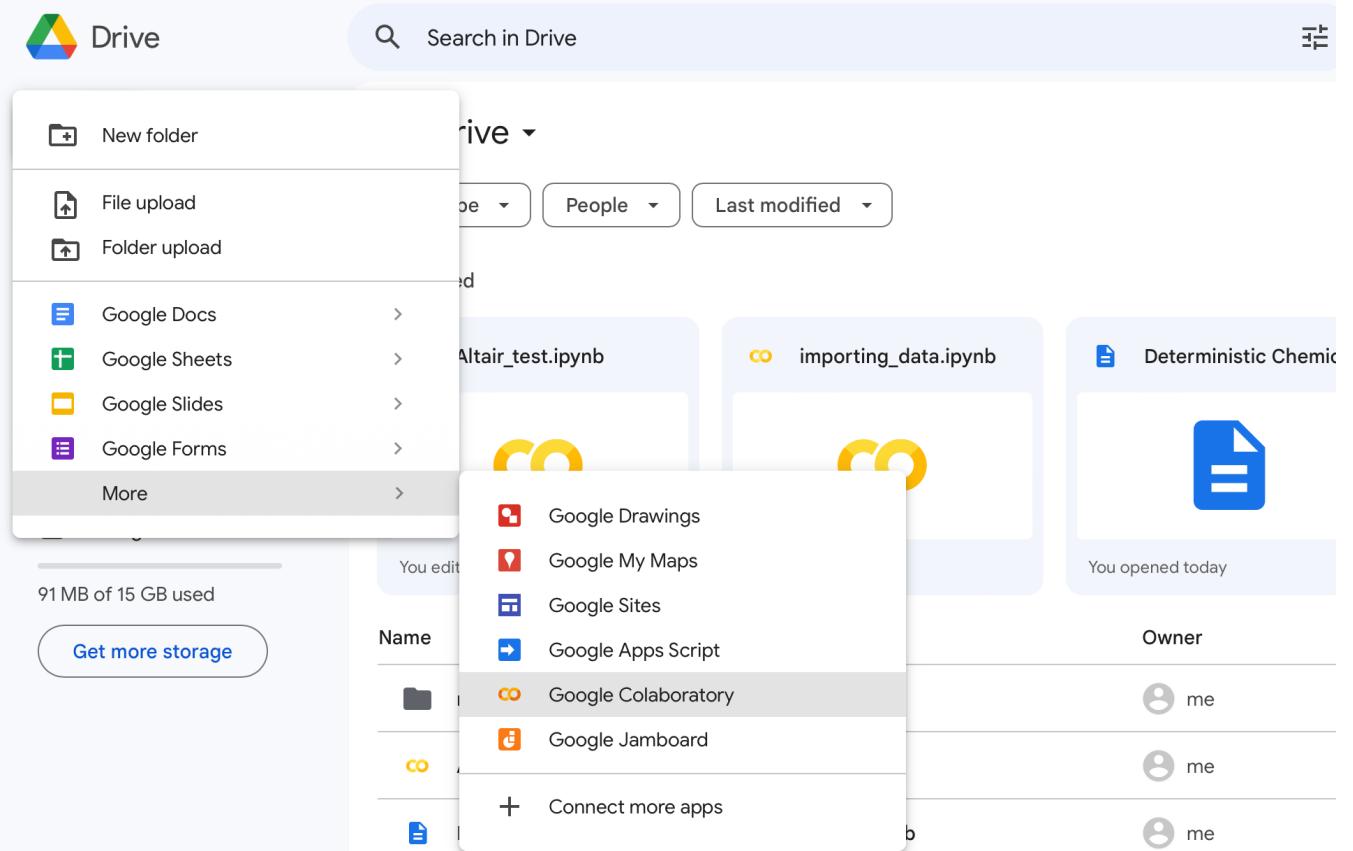


Figure 8 Launching a new notebook in Google Colab using **New** → **More** → **Google Colaboratory**.

Once your notebook is open, you can execute code or markdown cells by either selecting one of the run options (e.g., **Run all**) in the **Runtime** menu, by clicking the ► button at the left of the cell (Figure 9), or by using the **Shift + Return** shortcut.

```
t = np.linspace(0,20,20) # time(seconds)
A_0 = 1 # starting concentration (molarity)
k = 0.2 # rate constant in 1/s

# dA/dt = k[A]
def rate_1st(A, t):
    return - k * A

A_t = scipy.integrate.odeint(rate_1st, A_0, t)

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

Figure 9 A cell can be executed by clicking the ► button at the left of a cell in Google Colab among other methods.

Just like Jupyter installed on a computer, once a code cell is run, the code is executed with any output (e.g., numbers, text, or graphs) appearing directly below the code cell. 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. If code is written in a markdown cell, it is treated like regular text instead of code.

To add additional cells in Google Colab, click either the **+ Code** or **+ Text** above the notebook to produce another code or markdown cell, respectively.

The one other major difference between running the software installed on your own computer and Google Colab is that if

you want Colab to be able to interact with data or images files on your Google Drive, you need to include the three extra lines of code shown below at the top of your notebook. The first two lines grant the notebook access to read/write files on your Google Drive while the third line (`%cd /content/drive/My Drive/project`) points your notebook to where your files are located. The path should reflect the location of the folder containing your notebook and data files. For example, if your notebook is contained in a folder titled `project` on Google Drive, the path will be `/content/drive/My Drive/project`.

```
from google.colab import drive  
drive.mount('/content/drive')  
  
%cd /content/drive/My Drive/project
```

0.4 Markdown

Markdown is a light-weight markup language that allows users to make bold, italic, or monospaced text and various kinds of lists and other simple formatting. The table below 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 1 Markdown Syntax

Markdown Syntax	Result
# Header	Header
## Sub-Header	Sub-header
## Sub-Sub-Header	Sub-Sub-Header
* Italic *	Italic
** Bold **	Bold
`Monospace`	Monospace
—	Line across page
>	Indents the block of text
* Item	Bulleted item in line
1. Item	Numbered list item
[link](www.python.org)	URL link

One difference between writing code in a code cell versus a markdown cell is that code cells color the text based on the syntax or the role the text plays in the code, known as *syntax highlighting*, and markdown cells do not. It would be like if a word processor colored nouns gray, verbs orange, prepositions blue, and punctuation marks red so that reader can see the role each word or symbol plays in a sentence. If you want to include example text in a markdown cell with syntax highlighting, place `~~~python` in the line above the code and `~~~` in the line below the code.

0.5 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 (attribution uncertain) 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." 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 differently than the rest of the code.

```
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

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

0.6 Overview of Python Scientific Libraries

The Python programming language allows for add-ons known as *libraries* or *packages* 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 equations or sampled data. For scientific applications, there is a series of core libraries collectively known as the *SciPy stack* along with many other popular libraries. The table below lists some of the common libraries for scientific applications with an asterisk by those often considered part of the SciPy stack.

Table 2 Common Python Scientific Libraries

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
Seaborn	Advanced plotting library built on matplotlib
SymPy*	Symbolic mathematics (somewhat 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 processing
Biopython	Computational biology and bioinformatics
Scikit-Bio	Computational biology and bioinformatics
RDKit	General purpose cheminformatics

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> (free resource)
2. Google Colab (and Jupyter) Cheat Sheet. <https://colab.research.google.com> (free resource)
3. SciPy Website. <https://www.scipy.org> (free resource)
4. IPython Interactive Computing Website. <https://ipython.org> (free resource)
5. VanderPlas, J. Python data Science Handbook: Essential Tools for Working with Data, 1st ed.; O'Reilly: Sebastopol, CA, 2017, chapter 1. Freely available from the author at <https://jakevdp.github.io/PythonDataScienceHandbook/> (free resource)

Chapter 1: Basic Python

Contents

- 1.1 Numbers
- 1.2 Variables
- 1.3 Strings
- 1.4 Boolean Logic
- 1.5 Conditions
- 1.6 List & Tuples
- 1.7 Loops
- 1.8 File Input/Output (I/O)
- 1.9 Creating Functions
- Further Reading
- Exercises

1.1 Numbers

1.1.1 Basic Math

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 the table below.

Table 1 Python 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, so parentheses can be used to change the flow of the mathematical operations as needed.

```
8 + 3 * 2
```

14

```
(8 + 3) * 2
```

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.

```
3 / 2
```

1.5

```
3 // 2
```

1

Exponentiation is performed with a double asterisk (`**`). The carrot (`^`) means something else, so be careful not to accidentally use this.

```
2 ** 3
```

```
8
```

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.

```
10 % 3
```

```
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.

```
3 + 8
```

```
11
```

```
3.0 + 8
```

```
11.0
```

```
2 / 5
```

```
0.4
```

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

```
int(3.0)
```

3

`float(4)`

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 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 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).

`abs(-4)`

4

`round(4.5)`

4

Table 2 Common Python Functions

Function	Description
<code>abs()</code>	Returns the absolute value
<code>float()</code>	Converts a value to a float
<code>int()</code>	Converts a 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](#).

```
print(8.3145)
```

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. The `math` module comes with every installation of Python and is activated by importing it (i.e., loading it into memory) 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.

```
import math
math.sqrt(4)
```

2.0

Table 3 lists some commonly used functions in the `math` module, and a few examples are shown below. Interestingly, some functions simply provide a mathematical constant.

```
math.ceil(4.3)
```

```
5
```

```
math.pi
```

```
3.141592653589793
```

```
math.pow(2, 8)
```

```
256.0
```

Table 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 π
<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 only the `radians()` function.

```
from math import radians  
radians(4)
```

```
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 1.2](#) 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.

```
a = 5.0  
b = 3
```

```
a + b
```

```
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), and are not contained in Python's list of reserved words shown in Table 4. 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 4 Reserved Words in Python

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 generic example (like used in this book) or experiment. This will make writing and reading code significantly easier and is a good habit to start early. In the examples below, a reader might be able to determine that the first example is calculating energy using $E = mc^2$ while it is more difficult to determine what the second example is calculating.

```
# clear variables

mass = 1.6
light_speed = 3.0e8
mass * light_speed**2
```

1.44e+17

```
# not-so-great variables

x = 3.2
a = 1.77
a + x
```

4.970000000000001

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.

```
x = 5
y = x
```

y

5

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

```
x = 8
```

```
y
```

```
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 illustrates what happens in the above example.

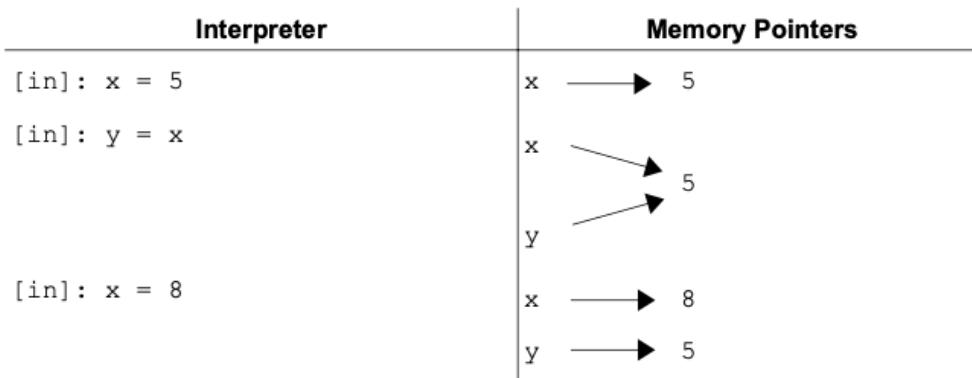


Figure 1 A representation of memory pointer during variable assignment is shown with the Python code (left) and the corresponding points (right).

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.

```
'some text'
```

```
'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, and a string can be assigned to variables just like floats and integers. To have Python print out the text, use the `print()` function.

```
text = "some text"
```

```
print(text)
```

```
some text
```

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

```
str(4)
```

```
'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.

```
'4' + '2'
```

```
'42'
```

```
'4' * 2
```

```
'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.

```
int('4') * int('2')
```

```
8
```

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

```
len(text)
```

```
9
```

The length of `'some text'` is 9 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.

```
print(str(4.0) + ' g')
```

```
4.0 g
```

```
print(4.0, 'g')
```

```
4.0 g
```

1.3.2 Indexing & 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.

```
seq = 'MSLFKIRMPE'
```

```
seq[0]
```

```
'M'
```

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

```
'MSLFKIRMPE'[1]
```

```
'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.

```
len(seq)
```

10

seq[9]

'E'

The string can also be reverse indexed from the last character to the first using negatives starting with -1 the last character.

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

seq[-1]

'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.

seq[0:5]

'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.

file = '1rxt.pdb'

file[-3:-1]

'pd'

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

file[-3:]

'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.

```
file[:-4]
```

```
'1rxt'
```

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]`.

```
seq[::-2]
```

```
'MLKRP'
```

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.

```
quote = 'anyone who has never made a mistake has never tried anything new.'
```

```
quote.capitalize()
```

```
'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.

```
quote
```

```
'anyone who has never made a mistake has never tried anything new.'
```

```
cap_quote = quote.capitalize()
```

```
cap_quote
```

'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.

```
str.isalpha(quote)
```

False

```
str.capitalize(quote)
```

'Anyone who has never made a mistake has never tried anything new.'

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

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

1.3.4 String Formatting

In [section 1.3.1](#), we were able to concatenate two strings by using the `+` operator as shown below. With this approach, it is necessary to convert any non-string into a string using the `str()` function.

```
MW = 63.21
"Molar mass = " + str(MW) + " g/mol."
```

```
'Molar mass = 63.21 g/mol.'
```

While this approach usually works fine, it can get messy or unwieldy as you are combining more strings. In this section, we will cover a couple other methods for merging strings. Which you choose to use is a matter of personal preference, but it is good to be aware of them as you may see them around.

`str.format()` Method

The first method we will address is using the `str.format()` method. In this approach, the string (i.e., `str`) includes curly brackets `{}` where you want to insert additional strings, and these additional strings are provided as arguments in the `str.format()` function. As an example, below we are generating a sentence providing the name and molecular weight of a compound. Notice how `compound` is inserted in the sentence where the first `{}` is located while `MW` is inserted in the location of the second `{}`.

```
compound = 'ammonia'
MW = 17.03

'The molar mass of {} is {} g/mol.'.format(compound, MW)
```

```
'The molar mass of ammonia is 17.03 g/mol.'
```

If we assign the `compound` and `MW` variable to other values, the `str.format()` function dutifully inserts these new strings into our sentence. Also notice that the `format()` function automatically converts non-string objects into strings for us.

```
compound = 'urea'
MW = 60.06

'The molar mass of {} is {} g/mol.'.format(compound, MW)
```

```
'The molar mass of urea is 60.06 g/mol.'
```

A variation of the above approach is to include an index value inside the curly brackets indicating which string provided to the `str.format()` function is inserted where in the sentence. In the example below, `compound` is provided to the `str.format()` function first, so it replaces `{0}` while `MW` is second, so it replaces `{1}`. Remember that Python index values start with zero.

```
compound = 'urea'
MW = 60.06

'The molar mass of {0} is {1} g/mol.'.format(compound, MW)
```

```
'The molar mass of urea is 60.06 g/mol.'
```

Because we are explicitly providing index values, we can insert strings into the sentence in any order. Notice in the example below that the `MW` and `compound` variables are provided to the function in a different order.

```
'The molar mass of {1} is {0} g/mol.'.format(MW, compound)
```

```
'The molar mass of urea is 60.06 g/mol.'
```

We can also insert strings into our sentence multiple times as shown below.

```
'The compound {0} is a molecular compound \
and {0} has a molar mass of {1} g/mol.'.format(compound, MW)
```

```
'The compound urea is a molecular compound and urea has a molar mass of 60.06 g/mol.'
```

F-Strings

The next approach to combining strings is using *f-strings*. In this approach, the string is preceded with `f`, and any inserted strings are denoted using `{}` with the variable name inside the curly brackets as demonstrated below. The appeal of this approach is that it is simple, versatile, and relatively easy to follow.

```
f'The molar mass of {compound} is {MW} g/mol.'
```

```
'The molar mass of urea is 60.06 g/mol.'
```

We can also modify the strings by placing additional Python code inside the brackets like below where the first letter of the compound is capitalized.

```
f'The molar mass of {compound.capitalize()} is {MW} g/mol.'
```

```
'The molar mass of Urea is 60.06 g/mol.'
```

1.4 Boolean Logic

Python supports *Boolean logic* where all expressions are evaluated as 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 6. These operators are essentially truth tests with Python returning either `True` or `False`. 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.

```
8 == 3
```

```
False
```

Table 6 Basic Boolean Comparison Operators

Operator	Description
<code>==</code>	Equal (double equal sign)
<code>!=</code>	Not equal
<code><=</code>	Less than or equal
<code>>=</code>	Greater than or equal
<code><</code>	Less than
<code>></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.

```
8 > 3
```

```
True
```

```
8 == 8.0
```

```
True
```

```
8 is 8.0
```

```
<=>1: SyntaxWarning: "is" with 'int' literal. Did you mean "=="?
<=>1: SyntaxWarning: "is" with 'int' literal. Did you mean "=="?
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4426/1557232650.py:1: SyntaxWarning: "is" wi
8 is 8.0
```

False

In the last example, Python generates a warning because the user probably meant to use `==` instead of `is`.

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 7.

Table 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`.

True and False

False

True or False

True

`8 > 3 or 8 < 2`

True

`not 8 > 3`

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 8 Truth Table for the `and` / `or` Logic Operators

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

Table 9 Truth Table for the `not` Logic Operator

<code>p</code>	<code>p not q</code>
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.

```
bool(1)
```

```
True
```

```
bool(0)
```

```
False
```

Python also accepts any non-zero value as `True`.

```
bool(5)
```

```
True
```

You can perform some of the above Boolean operations from [section 1.4.2](#) with `1` and `0`, but Python will return the result in terms of `1` and `0`.

```
1 or 0
```

```
1
```

```
1 and 0
```

```
0
```

1.4.4 `any()` & `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`.

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

```
True
```

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

```
False
```

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

```
True
```

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

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

```
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.

```
comp1 = 'Co(NH3)6'  
comp2 = 'Ni(H2O)6'
```

```
'Ni' in comp1
```

```
False
```

```
'Ni' in comp2
```

```
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.

```
x = 7
```

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

```
49
```

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.

```
x = 3
```

```
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 when the `if` statement evaluates as `False`. This is accomplished using the `else` statement as shown below.

```
pH = 9
```

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

```
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.

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

```
The solution is basic.
```

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 List & 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 chemical elements.

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

```
mass
```

```
[1.01, 4.0, 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 in this example, an `'NA'` string resides where a value would otherwise be.

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

```
EN
```

```
[2.1, 'NA', 1.0, 1.5, 2.0, 2.5]
```

1.6.2 Indexing & Slicing List

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.

```
EN[0]
```

```
2.1
```

```
EN[1]
```

```
'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]`.

```
EN[0:3]
```

```
[2.1, 'NA', 1.0]
```

```
EN[3:5]
```

```
[1.5, 2.0]
```

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

```
EN[3:]
```

```
[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 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 10 Common List Methods

Method	Description
<code>append(element)</code>	Adds a single element 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 element occurs in the list
<code>extend(elements)</code>	Adds multiple elements to the list
<code>index(element)</code>	Returns the index of the first occurrence of font
<code>insert(index, element)</code>	Inserts the given element at the provided index
<code>pop(index)</code>	Removes and returns 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 element in the list
<code>reverse()</code>	Reverses the order of the entire list
<code>sort()</code>	Sorts the list in place

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 2), the `sort()` method modifies the original list.

```
mass = [4.00, 1.01, 6.94, 14.01, 10.81, 12.01, 9.01]
```

```
mass.sort()  
mass
```

```
[1.01, 4.0, 6.94, 9.01, 10.81, 12.01, 14.01]
```

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

```
mass.reverse()  
mass
```

```
[14.01, 12.01, 10.81, 9.01, 6.94, 4.0, 1.01]
```

Probably one of the most useful methods in Table 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.

```
mass.append(16.00)  
mass
```

```
[14.01, 12.01, 10.81, 9.01, 6.94, 4.0, 1.01, 16.0]
```

```
mass.extend([19.00, 20.18])  
mass
```

```
[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.

```
mass.append([23.00, 24.31])  
mass
```

```
[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.

Tip

The `append()` method is frequently used as a means of storing values in a list as they are generated like the following calculation of the wavelnegths in the Balmer series. The `for` loop is explained in [section 1.7.1](#).

```
wavelengths = []
for n in range(3,6):
    wl = 1 / (1.097E-2 * (0.25 - 1/n**2))
    wavelengths.append(wl)
```

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.

```
a = range(10)
print(a)
```

```
range(0, 10)
```

The output of `a` is probably not what you expected. You were likely 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.

```
list(a)
```

```
[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.

```
list(range(3, 12))
```

```
[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.

```
list(range(3, 20, 3))
```

```
[3, 6, 9, 12, 15, 18]
```

```
list(range(10, 3, -1))
```

```
[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.

```
ten_nums = range(10)
```

```
ten_nums[2]
```

```
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 to 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.

```
nrg = (-2.18e-18, -5.45e-19, -2.42e-19, -1.36e-19, -8.72e-20)
```

```
nrg[1] - nrg[0]
```

```
1.635e-18
```

```
nrg[4] - nrg[3]
```

```
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 often 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:

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

```
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.

```
2 * [4, 6, 2]
```

```
[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.

```
numbers = [1, 2, 3, 4, 5, 6] # original values  
squares = [] # an empty list  
  
for value in numbers:  
    squares.append(value**2)
```

```
squares
```

```
[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 (λ)

in the Balmer series by the following equation where R_{∞} 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.

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

```
656.3354603463993
486.1744150714068
434.084299170899
410.2096627164995
397.04243897498225
```

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

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

```
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 g ^{235}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.

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

```
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 an intutive 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.

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

```
0
2
4
6
8
```

Essentially, what is going on is shown in the following example, and this continues until `x` is no longer greater than 10.

```
if x < 10:
    print(x)
    x = x + 2
if x < 10:
    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 runaway process.

⚠ Warning

Do **not** run the following code! It may result in Python crashing.

```
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, & Break Commands

Other ways to control the flow of code execution are the `continue`, `pass`, and `break` commands. These are not used heavily, but it is helpful to know about them on the occasions that you need them. Table 11 summarizes each of these statements below.

Table 11 Loop Interruptions

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

The `break` statement breaks out of the most immediate 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.

```
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(f'Endpoint: {vol_H} mL HCl solution')
```

Endpoint: 32 mL HCl solution

If we solve this titration using the $C_1V_1 = C_2V_2$ equation where C is concentration and V is volume, 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 for instances 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 and 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.

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

```
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.

```
pH = 5
if pH > 7:
    print('Basic')
else:
```

```
Cell In[127], line 4
  else:_
SyntaxError: incomplete input
```

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

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

1.8 File Input/Output (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. The reading of binary files requires other specialized software, such as demonstrated in [chapter 12](#), and text files are very common for storing data, so we will focus only on text files here.

There are 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 and open them in any text editor or word processor. 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, and spreadsheet applications are able to easily read and write CSV files. 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. In some files, you may see a `\n` at the end of each line. This is a line terminator character telling some software applications where a line ends.

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

1.8.1. Reading Lines with Python

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 practice 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`.

Note

One major difference between running the software installed on your own computer and Google Colab is that if you want Colab to be able to interact with data or images files on your Google Drive, you need to include the three extra lines of code shown below at the top of your notebook. The first two lines grant the notebook access to read/write files on your Google Drive while the third line (`%cd /content/drive/My Drive/project`) points your notebook to where your files are located. The path should reflect the location of the folder containing your notebook and data files. For example, if your notebook is contained in a folder titled `project` on Google Drive, the path will be `/content/drive/My Drive/project`.

```
from google.colab import drive
drive.mount('/content/drive')

%cd /content/drive/My Drive/project
```

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

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

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 lists. We can fix this by creating a couple empty lists and appending the values to the lists as the file is read.

```
file = open('data/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]))  
  
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.

```
numbers
```

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

```
squares
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

While the above methods work fine, it is considered best practice to read a file inside a `context` so that even if an error occurs, the file will still be closed properly. This is done as shown below using a `with` statement. There is no need to explicitly close the file because it is done automatically.

```
with open('data/squares.csv') as file:  
    for line in file.readlines():  
        print(line)
```

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

1.8.2 Writing Data with Python

Python can also write data to a file using the `write()` function which takes a string and writes it to a file. Before this can be done, the file needs to be opened using the `open()` function which requires the name of the file to write to, and if the file does not already exist, it creates a new file with this name. There is an optional second argument for the `open()` function that sets the mode in which the file is opened. There are a number of modes, but common modes include `'w'` for write only mode, `'r'` for read only mode, and `'a'` for append mode. The latter adds any new text to the end of an already-existing file.

In the example below, a list, `angular`, containing nested lists of angular quantum numbers and shapes is written to a new file. Following each nested list (i.e., angular quantum number and shape pair) is a line terminator character `\n`. Because the following code opens the file in a context using a `with` statement, there is no need to explicitly close the file as this is done automatically.

```
angular = [['l', 'shape'], [0, 's'], [1, 'p'],  
           [2, 'd'], [3, 'f']]  
  
with open('new_file.csv', 'w') as file:  
    for row in angular:  
        file.write('{0}, {1}\n'.format(row[0], row[1]))
```

1.8.3 Reading Data with `np.genfromtxt()`

The second approach to reading data from files uses a function from the NumPy library called `genfromtxt()`. NumPy will not be covered in depth until [chapter 4](#), but we can still use a couple functions before then. Before using NumPy, we need to import it using `import numpy as np`, which can be thought of as activating the library. The `np.genfromtxt()` function takes two required arguments for reading a text file: the file name and the delimiter.

```
np.genfromtxt('file_name', delimiter='')
```

The *delimiter* is the symbol that 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=','`.

```
import numpy as np

file = np.genfromtxt('data/squares.csv', delimiter=',')
file

array([[ 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 list. If we want to know the square of 4, we can access that value using indexing. In the example below, the first index identifies the nested list inside the main list, and the second index indicates the second value inside that list.

```
file[4][1]

np.float64(25.0)
```

Another feature of the `np.genfromtxt()` function is the `skip_header=` 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
2, 4
3, 9
4, 16
5, 25
6, 36
7, 49
8, 64
9, 81
10, 100
```

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

```
file = np.genfromtxt('data/header_file.csv', delimiter=',', skip_header=2)
file
```

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

1.8.4 Writing Data with `np.savetxt()`

One of the easiest approaches to writing data back to a file is to again use a NumPy function, `np.savetxt()`, which requires both a file name as a string and the data. It is also recommended to include a delimiter as a string using the `delimiter=` keyword argument. This function can write a file from a list, tuple, or NumPy array (introduced in [section 4.1](#)), and if a list or tuple is nested, each inner list/tuple is a row in the written file.

```
np.savetxt('file_name', data, delimiter=',')
```

As an example, below is a nested list of temperatures ($^{\circ}\text{C}$) and the density of water at each temperature (g/mL). These data are saved to a file `water_density.csv` with each value separated by a comma.

```
# temp(C), density(g/mL)
H2O_dens = [[10, 0.999], [20, 0.998], [30, 0.996],
             [40, 0.992], [60, 0.983], [80, 0.972]]

np.savetxt('water_density.csv', H2O_dens, delimiter=',')
```

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 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.

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

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.

```
import math
```

```
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(f'The distance is: {d}')
```

If you run the above code, nothing seems to happen. 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.

```
distance((1, 2, 3), (4, 5, 6))
```

```
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.

```
distance((5, 2, 3), (7, 5.3, 9))
```

```
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 with a `return` statement instead of printing the value. By running the following code, it overwrites the original function.

```
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
```

```
distance([5, 6, 7], [3, 2, 1])
```

```
7.483314773547883
```

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

```
dist = distance([5, 6, 7], [3, 2, 1])
dist
```

```
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.

```
pairs = ((1, 2, 3), (2, 3, 4),
          ((3, 7, 1), (9, 3, 0)),
          ((0, 0, 1), (5, 2, 7)))
```

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

avg = sum(dist_list) / len(dist_list)
avg
```

```
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.

```
d
```

```
NameError
Cell In[149], line 1
----> 1 d

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*.

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

```
square_roots = []
roots(range(10))

square_roots
```

```
[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.

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

    print(f'{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 `13C` as expected. However, if we switch the arguments to `isotope(7,6)`, we get `13N`, which is not correct. Positional arguments are extremely common, but the user needs to know what information goes where when calling a function.

```
isotope(6, 7)
```

13C

```
isotope(7, 6)
```

13N

The other common type of argument is the *keyword argument*. These arguments are attached to a variable inside the parentheses. The advantage of a keyword argument is that the user does not need to be concerned about argument order as long as the arguments have the proper labels. Below is the same `isotope()` function redefined using keyword arguments.

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

    print(f'{mass}{symbol}')
```

```
isotope(protons=1, neutrons=2)
```

3H

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

```
isotope(neutrons=2, protons=1)
```

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.

```
isotope()
```

```
1H
```

```
isotope(neutrons=2)
```

```
3H
```

Functions can also take an indeterminate number of positional or keyword arguments, but this is less common and is covered in [section 2.7](#) as an optional topic for those who are interested.

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.

```
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.

```
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 are 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/> (free resource)
2. Downey, Allen B. *Think Python*, Green Tea Press, 2012. <https://greenteapress.com/wp/think-python-2e/> (free resource)
3. Reitz, K.; Schlusser, T. *The Hitchhiker's Guide to Python: Best Practices for Development*, O'Reilly: Sebastopol, CA, 2016.
4. Das, U; Lawson, A.; Mayfield, C.; Norouzi, N.; Rajasekhar, Y.; Kanemaru, R. *Introduction to Python Programming*, OpenStax: Houston, TX, 2024. <https://openstax.org/details/books/introduction-python-programming>. (free resource)

Exercises

Complete the following exercises in a Jupyter notebook. Any data file(s) referred to in the problems can be found in the [data](#) folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

1. A 1.6285 L (V) flask contains 1.220 moles (n) of ideal gas at 273.0 K (T). Calculate the pressure (P) for the above system by assigning all values to variables and performing the mathematical operations on the variables. Remember that $PV = nRT$ describes the relationship between V , n , P , and T where R is 0.08206 L·atm/mol·K.
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.

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

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.

```
elements = 'NaKBrClNOUP'
```

- 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...).

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`.

```
DNA = 'ATTCGCCGCTTA'
```

9. The following are the atomic numbers of lithium, carbon, and sodium. Assign each to a variable and use Python Boolean logic operators to evaluate each of the following.

```
Li, C, Na = 3, 6, 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?
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 ₂ SO ₄	Ca(OH) ₂	KOH
HNO ₃	Na ₂ SO ₄	KNO ₃	Mg(OH) ₂	HCO ₂ 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.
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:
- Reverse the list
 - Remove the last value (i.e., 18)
 - Append 16
13. In a Jupyter notebook:
- 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 ([section 2.4.3](#) 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.
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 with the space.

1000 g

500.0 g

250.0 g

125.0 g

62.5 g

31.25 g

19. The isotope ^{137}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. To accomplish this, create a counter (`counter = 0`) and add 1 to it each cycle of a `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.

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.csv', '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 `readlines()` 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.
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.
- 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

Contents

- 2.1 Syntactic Sugar
- 2.2 Dictionaries
- 2.3 Set
- 2.4 Python Modules
- 2.5 Zipping and Enumeration
- 2.6 Encoding Numbers
- 2.7 Advanced Functions
- 2.8 Error Handling
- 2.9 Date and Time Information
- Further Reading
- Exercises

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 most 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 you should still be able to follow along without having read this chapter. If you are in a rush, you can bypass this chapter and circle back as needed. 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.

```
x = 5
x = x + 1
x
```

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."

```
x += 1  
x
```

```
7
```

Augmented assignment can also be used with addition, subtraction, multiplication, and division as shown in Table 1.

Table 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 converts 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 → 9 using this method.

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

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This whole process can be condensed down into a single line using *list comprehension* demonstrated below.

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

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

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.

Note

In addition to list comprehension, there are the related dictionary comprehension and set comprehension shown below that can be used for [dictionary](#) and [set](#) objects introduced in the following two sections.

```
[1]: {n: 2*n**2 for n in range(5)}
[1]: {0: 0, 1: 2, 2: 8, 3: 18, 4: 32, 5: 50}
```

```
[2]: {n: 2*n**2 for n in range(5)}
[2]: {(0, 0), (1, 2), (2, 8), (3, 18), (4, 32)}
```

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.

```
H, He, Li = 1.01, 4.00, 5.39
```

```
H
```

```
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.

```
1.01, 4.00, 5.39
```

```
(1.01, 4.0, 5.39)
```

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

```
(H, He, Li) = (1.01, 4.00, 5.39)
```

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. This 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 other words, the variable to the left of the `:` is the variable that goes in the parentheses in a normal function definition, and everything to the right of the `:` is what is indented in a normal function definition.

```
lambda x: x**2
```

```
<function __main__.<lambda>(x)>
```

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.

```
f = lambda x: x**2
```

```
f(9)
```

```
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 `quad()` function requires 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.

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

```
from scipy.integrate import quad
import math
```

```
quad(lambda x: 2 * math.sin(math.pi * x)**2, 0, 0.4)
```

```
(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.

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

```
quad(particle_box, 0, 0.4)
```

```
(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.

```
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.

```
AM['Li']
```

```
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.

```
AM.keys()
```

```
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.

```
AM.items()
```

```
dict_items([('H', 1.01), ('He', 4.0), ('Li', 6.94), ('Be', 9.01), ('B', 10.81), ('C', 12.01), ('N', 14.
```

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

```
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 by calling the key and assigning it to a value as demonstrated below. Instead of giving an error, the dictionary inserts that key: value pair.

```
AM['Na'] = 22.99
AM
```

```
{'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,
 'Na': 22.99}
```

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

Another method for generating a dictionary is the `dict()` function which takes in pair for nested lists or tuples and generates key:value pairs as follows.

```
dict([(1, 'H'), (2, 'He'), (3, 'Li')])
```

```
{'H': 1, 'He': 2, 'Li': 3}
```

Not only can dictionaries be used to store data for calculations, such as atomic masses, they can also be used to store changing data as we perform calculations or operations. For example, let's say we want to count how often each base (i.e., A, T, C, and G) appears in the following DNA sequence `DNA`. For this, we create a dictionary `dna_bases` to hold the totals for each base and add one to each value as we iterate along the DNA sequence.

```
DNA = 'GGGCTCCATTGTCTGCCCGGGCGGGTGTAGTCTAAGGTT'

dna_bases = {'A':0, 'T':0, 'C':0, 'G':0}
for base in DNA:
    dna_bases[base] += 1

dna_bases
```

```
{'A': 4, 'T': 11, 'C': 10, 'G': 15}
```

2.3 Set

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.

```
compounds = {'ethanol', 'sodium chloride', 'water',
             'toluene', 'acetone'}
```

We can add additional items to the set using the `add()` set method.

```
compounds.add('calcium chloride')
compounds
```

```
{'acetone',
 'calcium chloride',
 'ethanol',
 'sodium chloride',
 'toluene',
 'water'}
```

```
compounds.add('ethanol')
compounds
```

```
{'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.

```
N = {'1s', '2s', '2p'}  
Ca = {'1s', '2s', '2p', '3s', '3p', '4s'}
```

```
N | Ca # returns orbitals in either set
```

```
{'1s', '2p', '2s', '3p', '3s', '4s'}
```

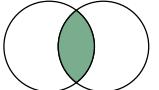
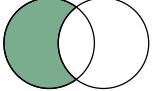
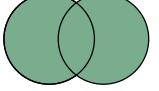
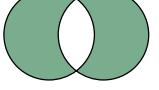
```
Ca - N # returns Ca orbitals minus those in common
```

```
{'3p', '3s', '4s'}
```

```
N & Ca # returns orbitals in both sets
```

```
{'1s', '2p', '2s'}
```

Table 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 [section 1.1.3](#), but Python also contains a number of other native modules that come with every installation of Python. Table 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. This section will introduce a few useful modules with some examples of their uses.

Table 3 Some Useful Python Modules

Name	Description
<code>os</code>	Provides access to your computer file system
<code>itertools</code>	Iterator and combinatorics tools
<code>random</code>	Functions for pseudorandom number generation
<code>datetime</code>	Handling of date and time information (see section 2.9)
<code>csv</code>	For writing and reading CSV files
<code>pickle</code>	Preserves Python objects on the file system
<code>timeit</code>	Times the execution of code
<code>audioop</code>	Tools for reading and working with audio files
<code>statistics</code>	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 4 Select os Module Functions

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

Table 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 in string format 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.

```
import os
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.

```
os.getcwd()
```

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.

```
import numpy as np
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 actual CSV files. See [section 13.2.5](#) for an example of this.

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.

```
import itertools
numbers = range(5)
itertools.combinations(numbers, 2)
```

```
<itertools.combinations at 0x10aaaf6b0>
```

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.

```
for pair in itertools.combinations(numbers, 2):
    print(pair)
```

```
(0, 1)
(0, 2)
(0, 3)
(0, 4)
(1, 2)
(1, 3)
(1, 4)
(2, 3)
(2, 4)
(3, 4)
```

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 example above.

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

```
(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 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 5 Functions from random Module

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

```
import random
random.random()
```

0.1536240063320783

```
random.randrange(0, 10)
```

3

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

[4, 6, 2, 5, 3, 1]

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.

```
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

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.

```
zipped = zip(AN, mass)
```

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

```
0  
2  
4  
5  
6  
6
```

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

```
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.

```
enum = enumerate(mass)
```

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

```
(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.

```
zipped = zip(range(len(mass)), mass)  
for item in zipped:  
    print(item)
```

```
(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 because 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. 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 6 lists a few examples with the terms used by Python.

Table 6 Python Data Types

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

Probably the simplest way to encode a number is an unsigned 8-bit integer. The "unsigned" means that it 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.

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.

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

```
per_yield(1.32, 0.50, 0.11, 0.27)
```

```
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 the `sum()` function returns `0` if no arguments are passed to it, the `per_yield()` function still works with no error returned.

```
per_yield(1.32)
```

```
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.

```
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 C₈H₁₀N₄O₂.

```
mol_mass(C=8, H=10, N=4, O=2)
```

```
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.

```
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)
```

```
half_life(4.00, hl=2)
```

1.0

```
half_life(4.00, hl=4)
```

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 using the `isinstance()` function which takes two arguments, the variable and the object type.

```
isinstance(x, type)
```

```
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 not isinstance(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)
```

```
half_life(4.00, hl=1.5)
```

```
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.

2.8 Error Handling

It doesn't take long to realize that error messages are an inevitable part of computer programming, so it is helpful to know what the different type of errors messages mean and how to deal with them. This section provides a quick overview of major types of error messages and how to get Python to work past them when appropriate.

2.8.1 Types of Errors

Whenever you encounter an error message, it includes the type of error followed by more details. There are numerous types of errors, but there are a few errors types that are more prevalent and worth being familiar with. Below is a short list of some of these common error types.

Table 7 A Selected List of Python Error Types

Type of Error	Description
<code>NameError</code>	A variable or name being used has not been defined
<code>SyntaxError</code>	Invalid syntax in code
<code>TypeError</code>	Incorrect object type is being used
<code>ValueError</code>	A value is being used that is not accepted by a function or for a particular application
<code>ZeroDivisionError</code>	Attempting to divide by zero
<code>IndentationError</code>	Invalid indentations are present
<code>IndexError</code>	Invalid index or indicies are being used
<code>KeyError</code>	Invalid key(s) for a dictionary or DataFrame are present
<code>DeprecationWarning</code>	Code uses a function or feature that will change in a future version

Examples and further details of each of these are provided below.

`NameError`

The `NameError` means the code uses a variable or function name that does not exist because it has not been defined.

This is often the result of mistyping a variable name but can have other causes like running code cells in a Jupyter notebook without first running necessary earlier code cells. If you just opened a Jupyter notebook, it is often worth selecting **Run → Run All Cells** from the top menu to ensure the latter doesn't happen.

```
print(root)
```

```
NameError
Cell In[54], line 1
----> 1 print(root)

NameError: name 'root' is not defined
```

SyntaxError

A programming language's syntax is the set of rules that dictate how the code is formatted, the appropriate symbols, valid values and variables, etc... It's all the rules that we've been learning about in the past couple of chapters. A **SyntaxError** indicates that your code violated one of these rules. To be helpful, the error message shows the line of code with the invalid syntax and points to where in the line the problem seems to be occurring.

In the first example below, the error occurred because `<>` is not a valid operator in Python.

```
5 <> 6
```

```
Cell In[55], line 1
  5 <> 6
  ^
SyntaxError: invalid syntax
```

The below example generates a **SyntaxError** because variable names cannot start with a number.

```
5sdq = 52
```

```
Cell In[56], line 1
  5sdq = 52
  ^
SyntaxError: invalid decimal literal
```

TypeError

A **TypeError** occurs when using the wrong object type for a particular function or application. For example, Python cannot take the absolute value of the letter, so this generates a **TypeError**.

```
abs('a')
```

```
-----  
TypeError                                 Traceback (most recent call last)  
Cell In[57], line 1  
----> 1 abs('a')  
  
TypeError: bad operand type for abs(): 'str'
```

A `TypeError` is encountered below because a boolean operation cannot be performed on a list - at least not without a `for` loop or NumPy (introduced in [chapter 4](#)).

```
[1,2,3] > 5
```

```
-----  
TypeError                                 Traceback (most recent call last)  
Cell In[58], line 1  
----> 1 [1,2,3] > 5  
  
TypeError: '>' not supported between instances of 'list' and 'int'
```

ValueError

The `ValueError` is somewhat similar to a `TypeError` except in this case it indicates that a numerical value is not valid or appropriate for a particular function. Some functions require that their arguments be within a certain range such as the `math.sqrt()` which does not accept negative numbers. As a result, taking the square root of -1 with this function generates a `ValueError`.

```
import math  
math.sqrt(-1)
```

```
-----  
ValueError                                Traceback (most recent call last)  
Cell In[59], line 2  
      1 import math  
----> 2 math.sqrt(-1)  
  
ValueError: math domain error
```

ZeroDivisionError

The `ZeroDivisionError` error is what the name says - the code attempted to divide by zero.

```
4 / 0
```

```
-----  
ZeroDivisionError                           Traceback (most recent call last)  
Cell In[60], line 1  
----> 1 4 / 0  
  
ZeroDivisionError: division by zero
```

IndentationError

Python does not care about spaces except those at the start of a line as these spaces or indentations have meaning. In the example below, the `print(x)` should be indented below the start of the `for` loop, so it generates an `IndentationError`.

```
for x in range(5):
    print(x)
```

```
Cell In[61], line 2
    print(x)
    ^
```

```
IndentationError: expected an indented block after 'for' statement on line 1
```

IndexError and KeyError

When indexing a composite object like a list, an index value that is outside the range results in an `IndexError`. In the list below, the indices run from `0` to `4`, so using an index of `5` returns an `IndexError`.

```
lst = [1,5,7,4,3]
lst[5]
```

```
-----
IndexError                                                 Traceback (most recent call last)
Cell In[62], line 2
  1 lst = [1,5,7,4,3]
----> 2 lst[5]

IndexError: list index out of range
```

Similarly, if the code tries to look up a value using a key not present in a dictionary, it returns a `KeyError` as shown below.

```
elements = {'H':1, 'He':2, 'Li':3, 'Be':4, 'B':5, 'C':6}
elements['Li']
```

```
3
```

```
elements['N']
```

```
-----
KeyError                                                 Traceback (most recent call last)
Cell In[64], line 1
----> 1 elements['N']

KeyError: 'N'
```

DeprecationWarning

A `DeprecationWarning` occurs when code uses a feature that will be removed or changed in a future release of Python or a third-party library. This error does not stop your code and is a friendly heads up that your code may not work in the future.

💡 Tip

Python error messages indicate the line where the error occurs, but on occasions you may find no error in that line of code. In these instances, the error is likely in the previous line. This can happen because Python provides means for continuing a line of code onto subsequent lines such as using a left parentheses, `(`, on the first line but not closing the parentheses with a right parentheses, `)`, until a later line. As an example, the following is executed by Python as if it were all on the same line.

```
V = (n * R * T_K  
     / P_atm)
```

2.8.2 Workout Around Errors with `try` and `except`

While this may seem like a bad idea at first glance, there are times when you may want Python to not come to a grinding halt in the face of an error. One common situation is when importing a large number of data files from different sources. Not every data source may have formatted data or files the same, and some files may be malformed or there may be other unexpected edge cases. To get Python to not stop at an error message, you can use a `try / except` block.

The general structure of a `try / except` block is to include the code you originally intend to run under the `try` statement, and under the following `except` statement, include what Python should do in the event of a specific error. The general structure looks like the following.

```
try:  
    regular code  
    regular code  
except ErrorType:  
    contingency code
```

As an example, let's say we are iterating through a list of numbers and appending the square root to a second list. Because one item in the original list of numbers is `'four'`, this causes a `TypeError`.

```
import math  
  
sqr_nums = [4, 25, 9, 81, 144, 'four', 49]  
sqr_root = []  
  
for num in sqr_nums:  
    sqr_root.append(math.sqrt(num))
```

```
TypeError
Cell In[66], line 5
  2     sqr_root = []
  4     for num in sqr_nums:
----> 5         sqr_root.append(math.sqrt(num))

TypeError: must be real number, not str
```

Instead, the `for` loop has been placed under a `try:` telling Python to make a best attempt at running the code. The code under the `except TypeError:` tells Python to run the following code in the event of a `TypeError`.

```
sqr_nums = [4, 25, 9, 81, 144, 'four', 49]
sqr_root = []

for num in sqr_nums:
    try:
        sqr_root.append(math.sqrt(num))
    except TypeError:
        print(f'{num} is not a float or int')
```

```
four is not a float or int
```

In the above example, nothing is done with the string except to inform the user that there was a problem. It is a prudent practice to not let unsolved errors pass by silently. If you have a good idea of where errors may turn up and have a solution to them, you can include that code under the `except:` as well.

Being that we know the above error is caused by a string, we can convert it to a float using a dictionary like below.

```
sqr_nums = [4, 25, 9, 81, 144, 'four', 49]
sqr_root = []

txt_to_int = {'one':1, 'two':2, 'three':3, 'four':4, 'five':5, 'six':6}

for num in sqr_nums:
    try:
        sqr_root.append(math.sqrt(num))
    except TypeError:
        integer = txt_to_int[num]
        sqr_root.append(math.sqrt(integer))
```

```
sqr_root
```

```
[2.0, 5.0, 3.0, 9.0, 12.0, 2.0, 7.0]
```

It is worth noting that `try/except` blocks can be avoided using `if/else` blocks like below.

```
sqr_nums = [4, 25, 9, 81, 144, 'four', 49]
sqr_root = []

for num in sqr_nums:
    if type(num) in [float, int]:
        sqr_root.append(math.sqrt(num))
    else:
        print(f'{num} is not a float or int')
```

```
four is not a float or int
```

So when should you use `try/except` versus `if/else`? If you anticipate exceptions to occur frequently, `if/else` is likely to be more efficient, but if exceptions are rare, it may be more efficient to use `try/except`.

2.8.3 Raising Exceptions

One thing worse than code not running is code running and producing incorrect outputs. At least when code fails to run, the user knows something is wrong whereas code that fails silently can lull the user into false conclusions. It is a prudent practice in coding to include checks that important conditions are met, and when these conditions are not met, the code should stop and produce an error known as *raising an exception*. To include checks in your code, you can use a condition with a `raise` statement followed by some form of error from Table 7 and an error message. The more specific you can be in your error type and message, the better.

As an example, we will write a function below which quantifies the differences between two DNA sequences. The *Hamming distance* is one possible metric for determining how different two sequences are and is simply the number of locations where two sequences of *the same length* are different. For example, `AATGC` and `AATGT` have a Hamming distance of 1 because they are identical except for the last base position. Because it is critical that the two DNA sequences be the same length, this should be checked before any further calculations, and if the sequences have different lengths, the function should not proceed and provide a helpful error message.

```
if len(seq1) != len(seq2):
    raise ValueError('Sequences must be of equal length')
```

Because the two sequences have the wrong number of bases, this qualifies as a `ValueError` (see [Table 7](#)). Inside the parentheses behind `ValueError`, a more detailed message can and should be provided.

```
dna1 = 'AACCT'
dna2 = 'ATCCA'
dna3 = 'ATCCTA'
```

```
def hamming(seq1, seq2):
    if len(seq1) != len(seq2):
        raise ValueError('Sequences must be of equal length')

    sequences = zip(seq1, seq2)
    distance = 0
    for position in sequences:
        if position[0] != position[1]:
            distance += 1

    return distance
```

When we compare the first two DNA sequences that are the same length, the function returns a numerical value. However, when comparing the second two sequences that are not the same length, the error message appears instead of a number.

```
hamming(dna1, dna2)
```

```
hamming(dna2, dna3)
```

```
-----
ValueError                                     Traceback (most recent call last)
Cell In[74], line 1
----> 1 hamming(dna2, dna3)

Cell In[72], line 4, in hamming(seq1, seq2)
    1 def hamming(seq1, seq2):
    2     if len(seq1) != len(seq2):
----> 4         raise ValueError('Sequences must be of equal length')
    5     sequences = zip(seq1, seq2)
    6     distance = 0

ValueError: Sequences must be of equal length
```

2.9 Date and Time Information

It is often necessary to know when data were collected such as in chemical kinetics. This information may be stored in the file itself or as a timestamp at the end of the file name. Not only is it necessary to extract this date and time information, it is often also necessary to calculate the times since the start of the experiment or between data points. This section covers Python's native `datetime` module useful for working with date and time information and extracting this information from files. The four object types covered here are listed in **Table 8**. The first three tell us when the data were collected while the third, `timedelta`, tells us the amount of time between two times or dates.

Table 8 Common `datetime` Objects

Object Type	Description
<code>date</code>	Contains date information ignoring time
<code>time</code>	Contains time information ignoring date
<code>datetime</code>	Contains date and time information
<code>timedelta</code>	Contains change in date and time information

We will start with what these objects are and how to work with them followed by how to use `datetime` to extract date and time information from data files. First, we need to import the `datetime` module.

```
import datetime
```

2.9.1 Date and Time Data

The `datetime` module often stores date and time information in a `datetime` object. A `datetime` object can be created multiple ways such as explicitly indicating a specific date and time using the `datetime()` method. For example, below we indicate noon on Pi Day 2025. The `datetime()` method takes the year, month, day, hour, minutes, seconds, and

microseconds as optional positional arguments in this order.

```
datetime.datetime(year, month, day, hour, minutes, seconds, microseconds)
```

```
pi_day = datetime.datetime(2025, 3, 14, 12, 0, 0, 0)
```

```
datetime.datetime(2025, 3, 14, 12, 0)
```

The date and time information can also be provided to `datetime()` using keyword arguments like below.

```
mario_day = datetime.datetime(year=2025, month=3, day=10, hour=8, minute=10, second=0, microsecond=0)
```

The current date and time can be accessed using the `now()` method for the `datetime` module. This function also accepts an optional timezone (`tz=`) argument (not discussed here). If no argument is provided, then `tz=None`. There is also a `datetime.datetime.today()` function that is equivalent to the `now()` function when no timezone is provided or `tz=None`. The `now()` function is recommended by the Python [Python datetime documentation](#).

```
now = datetime.datetime.now()  
now
```

```
datetime.datetime(2025, 8, 6, 15, 35, 53, 686964)
```

The hours, minutes, seconds, and microseconds can be accessed individually using the `hour`, `minute`, `second`, and `microsecond` attributes, respectively.

```
now.hour
```

```
15
```

A `datetime` object can be modified in place using the `replace` method like below.

```
now.replace(hour=3)
```

```
datetime.datetime(2025, 8, 6, 3, 35, 53, 686964)
```

The `datetime` module also has a `time` object that is similar to the `datetime` object except that it restricts itself to time information. The `time()` function used to create a `time` object accepts the time as optional positional arguments.

```
datetime.time(hour, minutes, seconds, microseconds)
```

```
time = datetime.time(5, 3, 32)  
time
```

```
datetime.time(5, 3, 32)
```

Like `datetime` objects, the hours, minutes, seconds, and microseconds can be accessed individually or modified in place.

```
time.second
```

```
32
```

```
time.replace(second=42)
```

```
datetime.time(5, 3, 42)
```

2.9.2 Changes in Date and Time

The differences between two `datetime` objects can also be calculated by subtracting the two objects. The result is returned as a `timedelta` object.

```
delta = pi_day - mario_day  
delta
```

```
datetime.timedelta(days=4, seconds=13800)
```

The days or seconds in the `timedelta` object can be accessed using the `days` or `seconds` attributes, respectively.

```
delta.days
```

```
4
```

```
delta.seconds
```

```
13800
```

To condense a `timedelta` into seconds, use the `total_seconds()` method.

```
delta.total_seconds()
```

```
359400.0
```

2.9.3 Extracting Date and Time Information

Extracting date and time from a file or file name can be accomplished using the 'string-parsed time' `strptime()` function and formatting codes shown below. Additional codes can be found on the [Python website](#).

Table 2 Formatting Codes for Parsing Date and Time Strings

Code	Example	Description	Length
%y	01	Year without century	Two digits
%Y	2001	Year with century	Four digits
%b	Jan	Month abbreviation	Three letters
%B	January	Month full name	Varies
%m	01	Month as zero padded number	Two digits
%d	05	Day of the month with zero padding	Two digits
%H	14	Hour in 24 hour time with zero padding	Two digits
%p	AM	AM or PM	Two letters
%l	02	Hour in 12 hour time with zero padding	Two digits
%M	16	Minute with zero padding	Two digits
%S	09	Second with zero padding	Two digits
%f	090000	Microseconds with zero padding	Six digits

These codes will allow you to parse strings into the `datetime` module by providing the `strptime()` function with both the string from the data file and a description of how the date and time information is organized. For example, below is a file where the collection time is included in the file name as hour, minutes, seconds separated by hyphens.

```
file_name_1 = 'Absorbance_12-03-48.txt'
timestamp = datetime.datetime.strptime(file_name_1[-12:-4], '%H-%M-%S')
```

```
datetime.datetime(1900, 1, 1, 12, 3, 48)
```

Because the date (i.e., year, month, and day) information were not provided, default values of January 1, 1900 is chosen for the `datetime` object. If you only want the date or time information, you can access them using the `date()` or `time()` functions, respectively.

```
timestamp.date()
```

```
datetime.date(1900, 1, 1)
```

```
timestamp.time()
```

```
datetime.time(12, 3, 48)
```

If the values are not formatted like Python assumes, a little extra effort may be required. For example, below the time is formatted at hours-minutes-seconds-microseconds, but microseconds is not represented as six digits with zero padding like Python assumed. To deal with this, the microseconds are sliced out of the file name and added to the `datetime` object using the `replace()` method.

```
file_name_2 = 'glucose_Absorbance_12-03-48-215.txt'  
time = datetime.datetime.strptime(file_name_2[-16:-8], '%H-%M-%S')  
time.replace(microsecond = int(file_name_2[-7:-4]))
```

```
datetime.datetime(1900, 1, 1, 12, 3, 48, 215)
```

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 videos and <https://stackoverflow.com/> boards for people looking for more information.

1. Python Documentation Page. <https://www.python.org/doc/> (free resource)
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/>. (free resource)
4. Das, U; Lawson, A.; Mayfield, C.; Norouzi, N.; Rajasekhar, Y.; Kanemaru, R. *Introduction to Python Programming*, OpenStax: Houston, TX, 2024. <https://openstax.org/details/books/introduction-python-programming>. (free resource)

Exercises

Complete the following exercises in a Jupyter notebook. Any data file(s) referred to in the problems can be found in the `data` folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

1. Generate a list containing the natural logs of integers from 2 → 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 translate along each axis and returns the final coordinates. The docstring 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.

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.
5. For the following two [sets](#): `acids1 = {'HCl', 'HNO3', 'HI', 'H2SO4'}` `acids2 = {'HI', 'HBr', 'HClO4', 'HNO3'}`
- Generate a new set with all items from acids1 and acids2.
 - Generate a new set with the overlap between acids1 and acids2
 - Add a new item `HBrO3` to acids1.
 - Generate a new set with items from either set but not in both
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 using the file browser. Does Python print any files that you do not see in the file browser?
7. Use the [random](#) module for the following.
- Generate 10 random integers from $0 \rightarrow 9$ and calculate the mean of these values. What is the theoretical mean for this dataset?
 - Generate 10,000 random integers from $0 \rightarrow 9$ 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))
```

9. Combining lists using [zip](#)
- Generate a list of the first ten atomic symbols on the periodic table.
 - 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 10. Start by generating a list of random integers and combine them with their index values using either [zip\(\)](#) or [enumerate\(\)](#).
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 p and n 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(f'{str(p)} protons and {str(n)} neutrons remaining.')
```

14. DNA strands contain sequences of nucleotide bases, and for DNA, these bases are adenine (A), thymine (T), guanine (G), and cytosine (C). When comparing two DNA strands of the same length, the Hamming distance is the number of places strand where the two DNA strands contain a different base. For example, the ATTG and ATCG sequences have a Hamming distance of 1 because they differ only by the third base position. Write a Python function that calculates the Hamming distance between two DNA sequences by zipping the two sequences. Your function should first check that the two sequences are of the same length and return an error message if they are not. Test the function on the following two DNA sequences.

```
dna1 = 'ATCCTGCATTAGGGAGCTTTATTGCCAATAGCTA'
dna2 = 'ATCCTGGATTAGGGAGCATTATTGCCAATAGGTA'
```

15. Chap 02: DNA sequences often do not contain equal quantities of GC versus AT bases, and the percentage of GC is known as the *GC-content*.

- Write a Python function that generates a random DNA sequence of a user defined number bases long with an average GC-content of 40%. The `random.choice()` function may be helpful here. Execute your function for a 50 bases DNA strand. Note: because your function generates a random sequence, the GC-content may not always be 40%, but the generated sequences GC-content should average to near 40% over a very large number of sequences generated.
- Write and test a separate Python function from above that calculates the GC-content of a user provided DNA sequence.

Chapter 3: Plotting with Matplotlib

Contents

- 3.1 Plotting Basics
- 3.2 Plotting Types
- 3.3 Overlaying Plots
- 3.4 Multifigure Plots
- 3.5 3D Scatter Plots
- 3.6 Surface & Wireframe Plots
- 3.7 3D Data on a 2D Surface
- Further Reading
- Exercises

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 scientific Python 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 code. This 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. You may sometimes also see a `%matplotlib inline` line. This used to be required to ensure the plots appeared in the notebook but is now typically not necessary.

```
import matplotlib.pyplot as plt
```

In all the examples below, simply calling a plotting function in a Jupyter notebook will automatically make the plot appear in the notebook below the plotting function. However, if you choose to use matplotlib in some other environment, it is often necessary to also execute the following `plt.show()` function to make the plot appear. This can also be done in Jupyter, but it is not shown in the rest of this chapter as Jupyter does not require it.

```
plt.show()
```

3.1 Plotting Basics

Before creating our first plot, we need some data to plot, so we will generate data points from orbital radial wave functions. The following equation defines the wave function (ψ) for the 3s atomic orbital of hydrogen with respect to

atomic radius (r) in Bohrs (a_0).

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

We will generate points on this curve using a method called list comprehension covered in section [2.1.2](#). In the examples below, `r` is the distance from the nucleus and `psi_3s` is the wave function. If you choose to plot something else, just make two lists or tuples of the same length containing the x - and y -values.

```
# create Python function for generating 3s radial wave function
import math

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

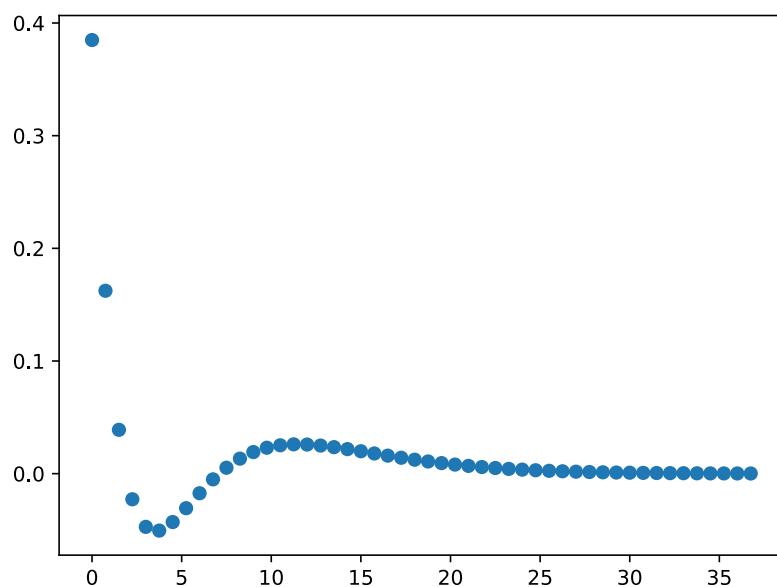


```
# generate data to plot
r = [num / 4 for num in range(0, 150, 3)]
psi_3s = [orbital_3S(num) for num in r]
```

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. The `r` and `psi_3s` data are fed into it as positional arguments as the x - and y -variables, respectively.

```
plt.plot(r, psi_3s, 'o');
```



By default, matplotlib creates a scatter plot using blue as the default color. This can be modified if 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 keyword arguments take strings. The `marker` argument allows the user to choose from a list of markers (Table 1). The `linestyle` argument (Table 2) determines if a line is solid or the type of dashing that occurs, and the `color` argument (Table 3) allows the user to dictate the color of the line/markers. If an empty string is provided to `linestyle` or `marker`, no line or marker, respectively, is included in the plot. See the [matplotlib website](#) for a more complete list of styles.

Table 1 Common Matplotlib Marker Styles

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

Table 2 Common Matplotlib Line Styles

Argument	Description
'-'	solid
'--'	dashed
'-. '	dash-dot
'.'	dotted

Table 3 Common Matplotlib Colors

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

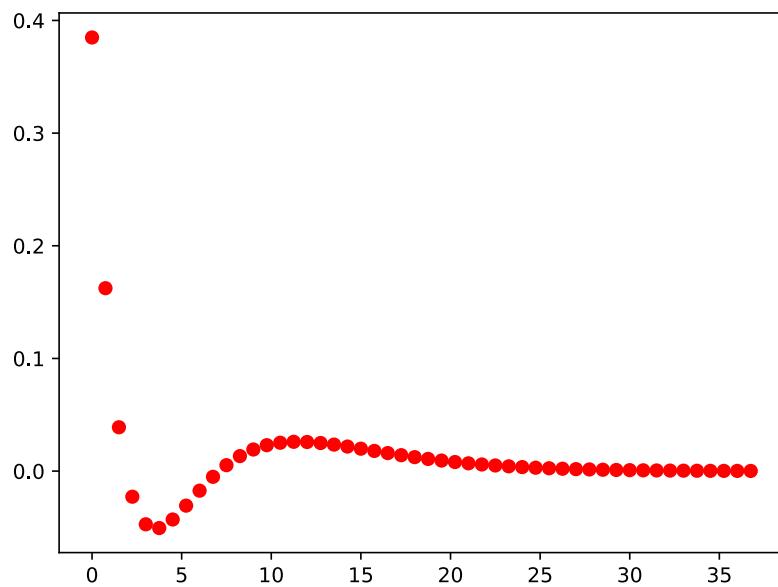
There are numerous other arguments that can be placed in the plot command. A few common, useful ones are shown below in Table 4.

Table 4 A Few Common plot Keyword Arguments

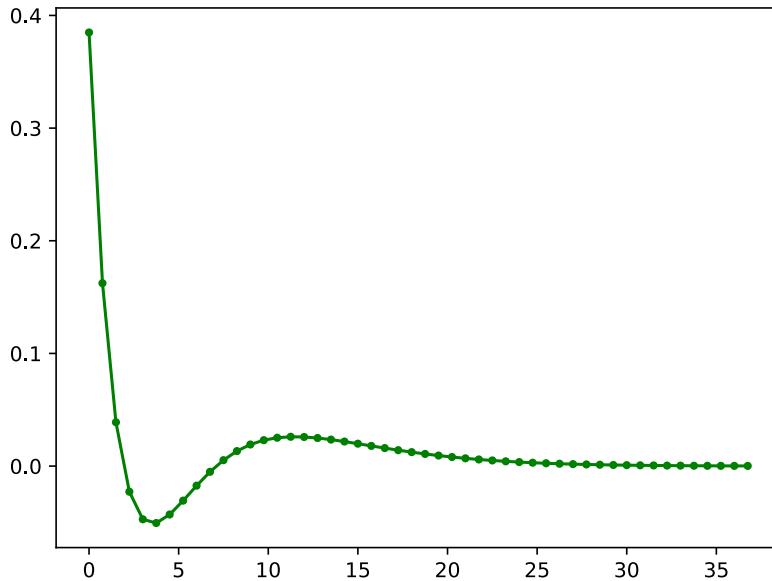
Argument	Description
<code>linestyle</code> or <code>ls</code>	line style
<code>marker</code>	marker style
<code>linewidth</code> or <code>lw</code>	line width
<code>color</code> or <code>c</code>	line color
<code>markeredgecolor</code> or <code>mec</code>	marker edge color
<code>markerfacecolor</code> or <code>mfc</code>	marker color
<code>markersize</code> or <code>ms</code>	marker size

Now that you have seen the keyword argument approach which allows for the fine tuning of plots, there is also a shortcut useful for 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 brevity.

```
# ro = red circle
plt.plot(r, psi_3s, 'ro');
```



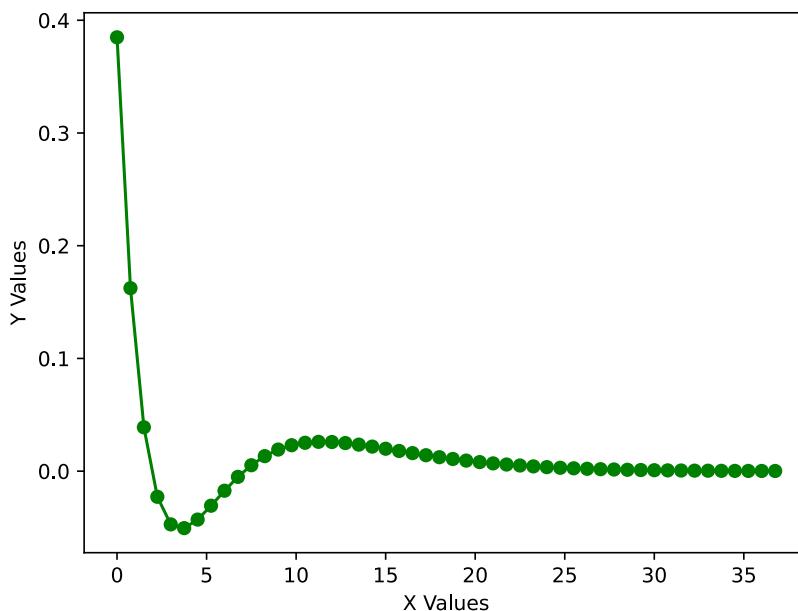
```
# g.- = green solid line with dots along it
plt.plot(r, psi_3s, 'g.-');
```



3.1.3 Labels

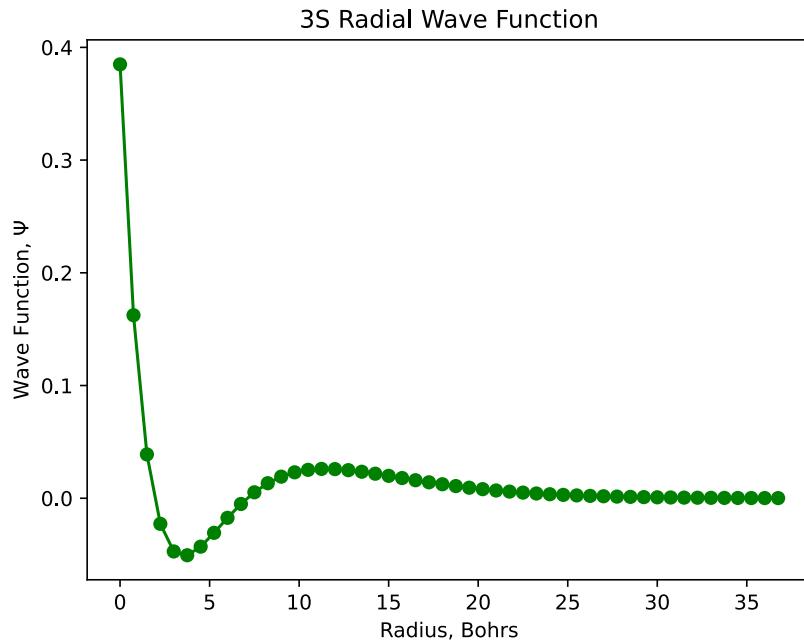
It is often important to label the axes of your plot. This is accomplished using the `plt.xlabel()` and `plt.ylabel()` functions which are placed on different lines as the `plt.plot()` function. Both functions take strings.

```
plt.plot(r, psi_3s, 'go-')
plt.xlabel('X Values')
plt.ylabel('Y Values');
```



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.

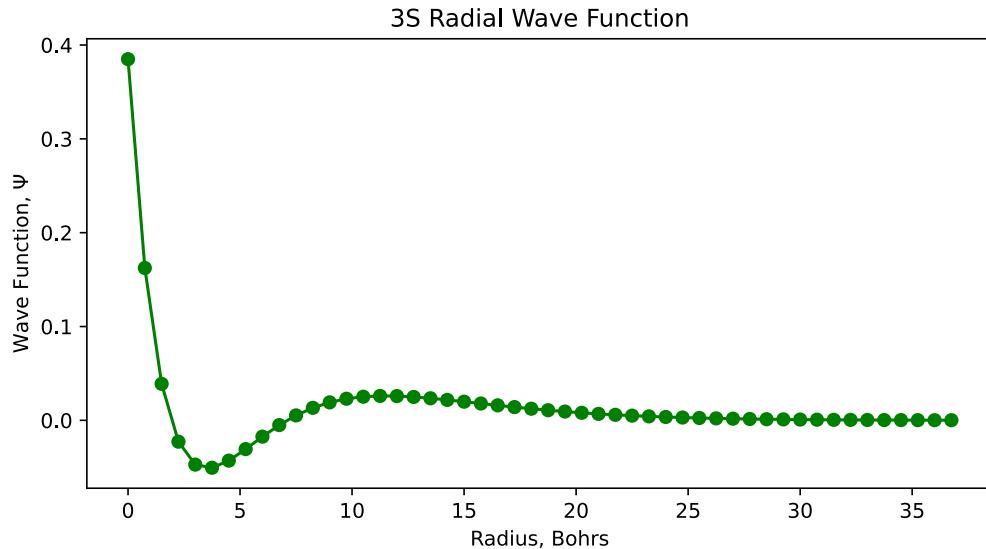
```
plt.plot(r, psi_3s, 'go-')
plt.xlabel('Radius, Bohrs')
plt.ylabel('Wave Function, $\Psi$')
plt.title('3S Radial Wave Function');
```



3.1.4 Figure Size

If you want to change the size or dimensions of the figure in the Jupyter notebook, this can be accomplished by `plt.figure(figsize=(width, height))`. It is important that this function be *above* the the actual plotting function and not below for it to modify the figure.

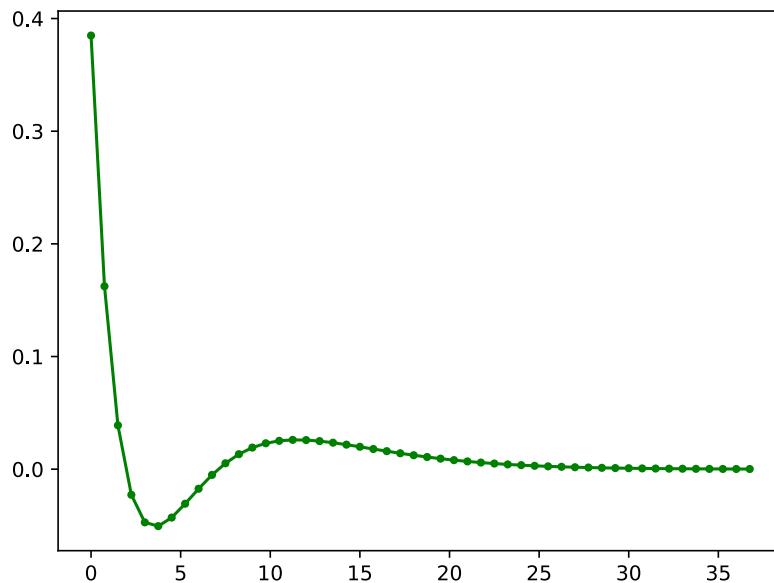
```
plt.figure(figsize=(8,4))
plt.plot(r, psi_3s, 'go-')
plt.xlabel('Radius, Bohrs')
plt.ylabel('Wave Function, $\Psi$')
plt.title('3S Radial Wave Function');
```



3.1.5 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.

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



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](#).

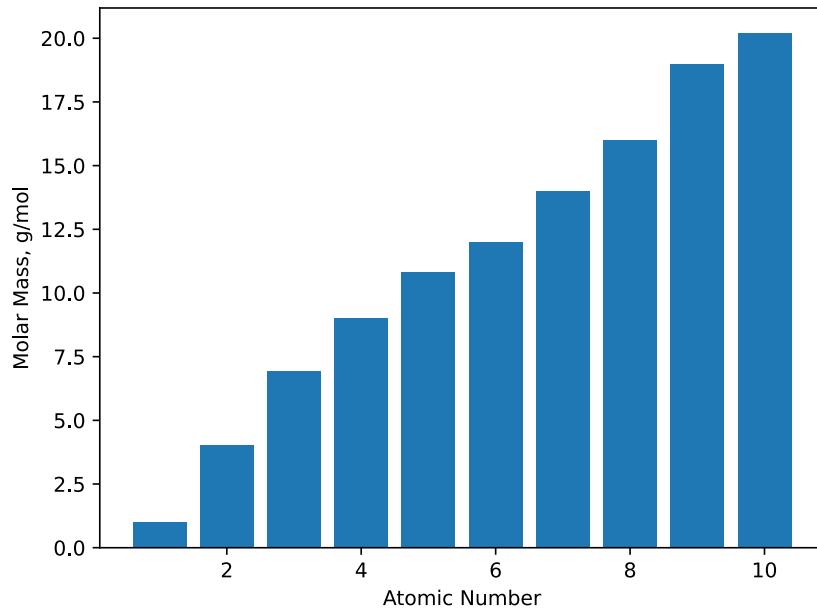
3.2.1 Bar Plots

Bar plots, despite looking very different, are quite similar to scatter plots. They both show the same information except that instead of the vertical position 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 -variables and plots the values with respect to the index values.

The atomic numbers ([AN](#)) for the first ten chemical elements are generated below using list comprehension introduced in [section 2.1.2](#) to be plotted with the molecular weights ([MW](#)).

```
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]
```

```
plt.bar(AN, MW)
plt.xlabel('Atomic Number')
plt.ylabel('Molar Mass, g/mol');
```



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 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 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

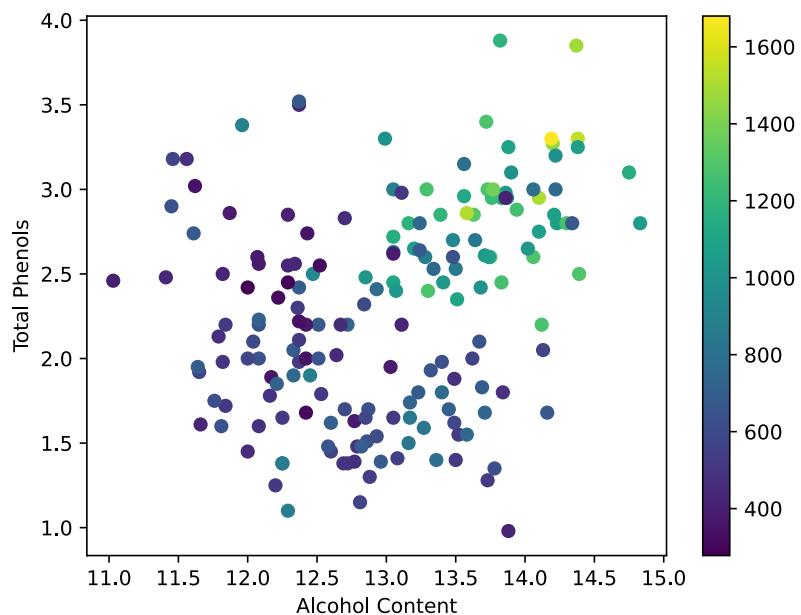
We have already generated scatter plots using the `plt.plot()` function, but they can also be created using the

`plt.scatter()` function. The latter is partially redundant, but unlike `plt.plot()`, `plt.scatter()` allows for different sizes, shapes, and colors of individual markers using the `s=`, `marker=`, and `c=` keyword arguments, respectively. See [section 3.1.2](#) for a short list of some of the marker shapes and colors available. Links to more complete lists can be found in the [Further Reading](#) section.

In the example below, we are loading the famous [wine dataset](#) that describes wine samples through a number of measurements including alcohol content, magnesium levels, color, etc... For convenience, we will load the dataset using the scikit-learn library introduced in [section 13.2.2](#). We then plot it and include a third attribute to the color `c=` argument.

```
from sklearn.datasets import load_wine
wine = load_wine()
wine = wine.data
```

```
plt.scatter(wine[:,0], wine[:,5], c=wine[:,12])
plt.xlabel('Alcohol Content')
plt.ylabel('Total Phenols')
plt.colorbar();
```

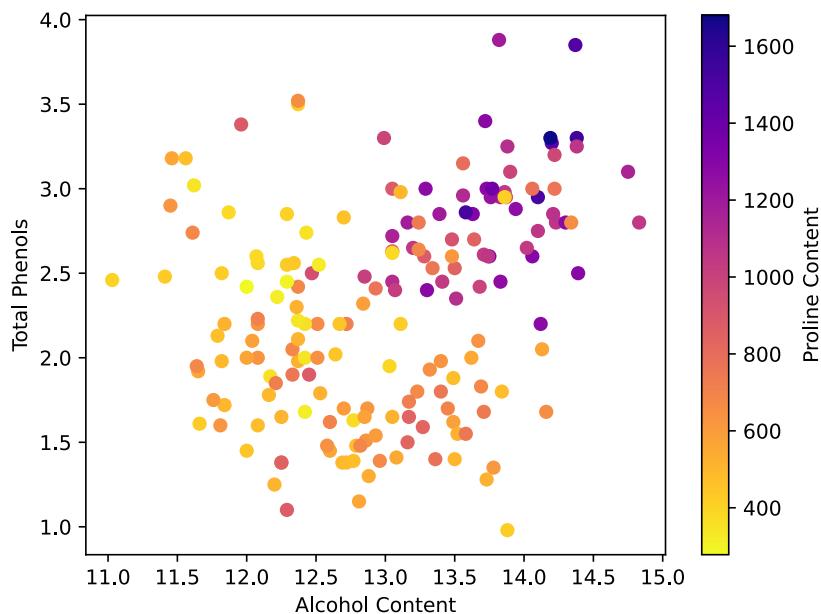


In the example above, the alcohol content is represented on the x -axis, the alkalinity is represented on the y -axis, and the proline content is shown using the color of the markers. The spectrum of colors that represent the values is called the *colormap*, and this can be changed using an optional `cmap=` argument. See the [matplotlib colormap page](#) for a list of available colormaps.

The `plt.colorbar()` provides a guide as to the meaning of the colors, but it would be nice to also have a text label on the color bar just like the axes. This can be accomplished by assigning the color bar to a variable and then using the `set_label()` attribute to add a label as demonstrated below.

```
plt.scatter(wine[:,0], wine[:,5], c=wine[:,12], cmap='plasma_r')
plt.xlabel('Alcohol Content')
plt.ylabel('Total Phenols')

cbar = plt.colorbar()
cbar.set_label('Proline Content');
```

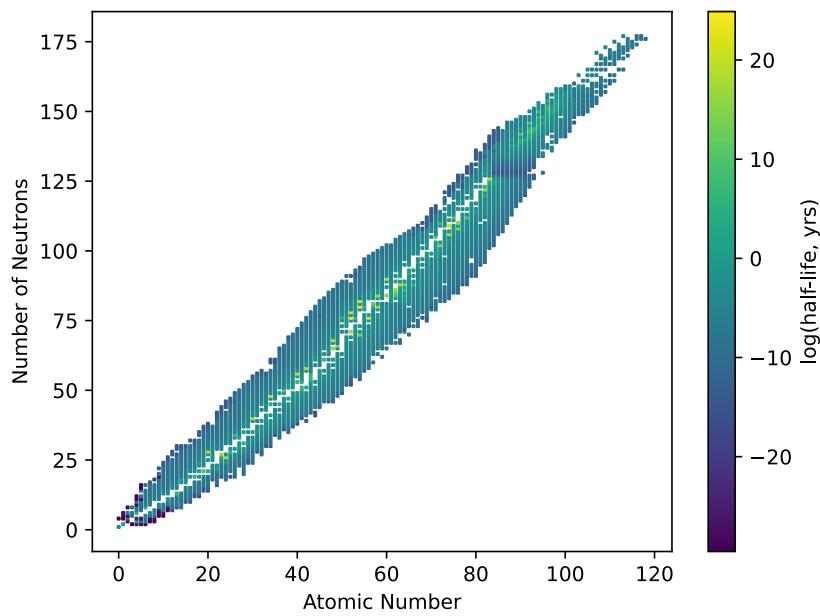


As an additional example, we can generate a plot of nuclide atomic numbers versus the number of neutrons and color the markers with the log of the half-life, in years, of each nuclide.

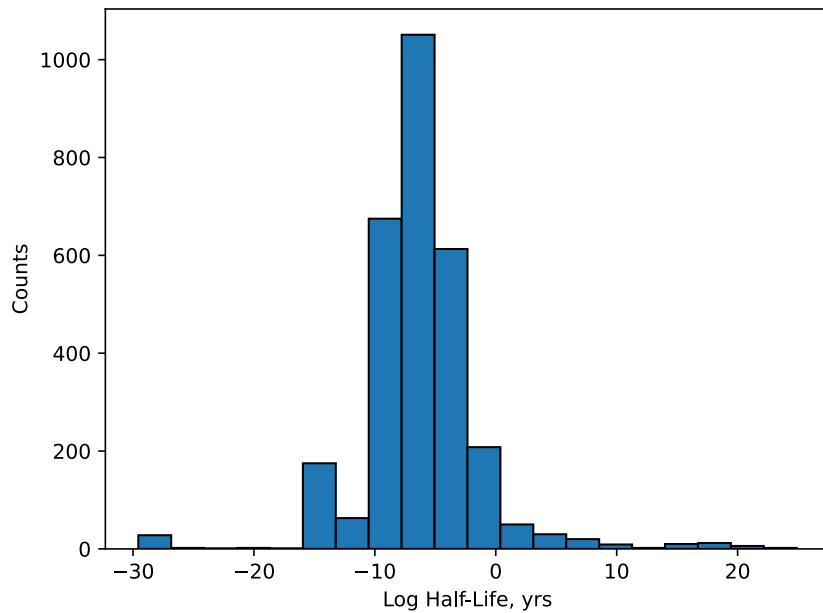
```
import numpy as np
nuc = np.genfromtxt('data/nuclide.csv', delimiter=',', skip_header=1)
nuc
```

```
array([[ 0.        ,  1.        , -4.71070897],
       [ 0.        ,  4.        , -29.25458877],
       [ 1.        ,  2.        ,  1.09089879],
       ...,
       [117.        , 176.        , -9.35267857],
       [117.        , 177.        , -8.79123643],
       [118.        , 176.        , -10.73537861]], shape=(2960, 3))
```

```
plt.scatter(nuc[:,0], nuc[:,1], s=1, marker='s', c=nuc[:,2], cmap='viridis')
plt.xlabel('Atomic Number')
plt.ylabel('Number of Neutrons')
cbar = plt.colorbar()
cbar.set_label('log(half-life, yrs)');
```

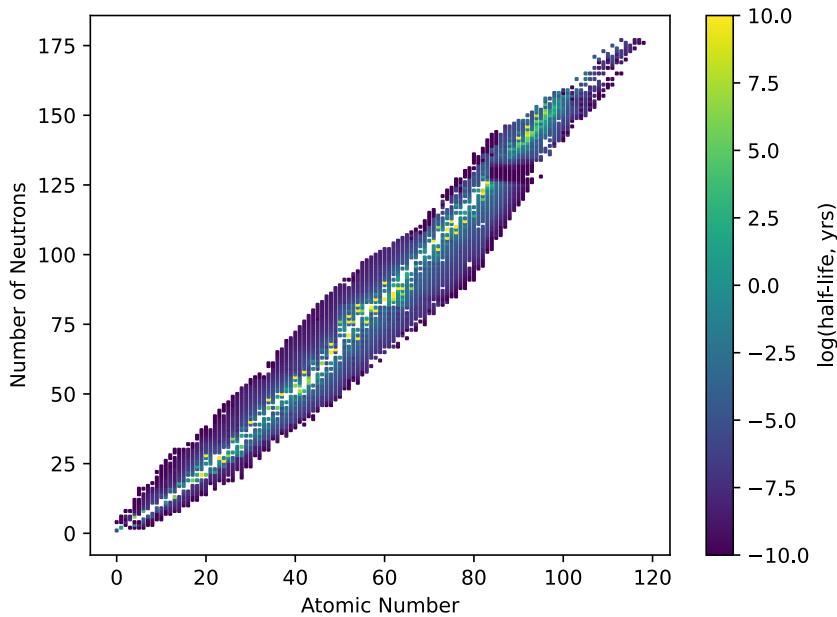


One of the issues we encounter in the above plot is that the range of half-lives is large with relatively few points in the extreme ends. We can see this in the histogram plot of these log half-life values shown below (see [section 3.2.3](#)).



In order to prevent the few values at the extremes from effectively washing out the color and making it difficult to see the differences, we can use the `plt.scatter()` arguments `vmax=` and `vmin=` to narrow the colormap range like shown below. By doing this, any values above the `vmax=` value will be a fixed color, and any values below the `vmin=` value will be a fixed color.

```
plt.scatter(nuc[:,0], nuc[:,1], s=1, marker='s', c=nuc[:,2],
            cmap='viridis', vmax=10, vmin=-10)
plt.xlabel('Atomic Number')
plt.ylabel('Number of Neutrons')
cbar = plt.colorbar()
cbar.set_label('log(half-life, yrs)');
```

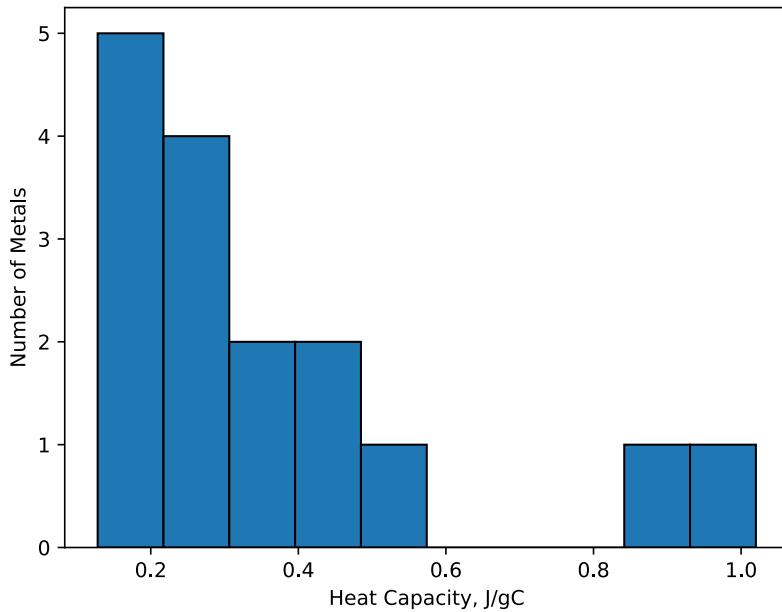


3.2.3 Histogram Plots

Histograms display bars representing the frequency of values in a particular dataset. 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.

```
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]

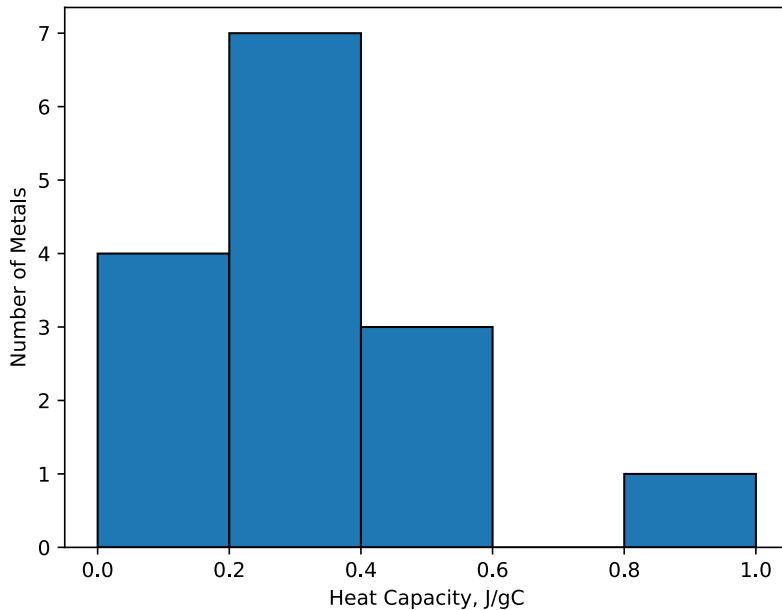
plt.hist(Cp, bins=10, edgecolor='k')
plt.xlabel('Heat Capacity, J/gC')
plt.ylabel('Number of Metals');
```



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.

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

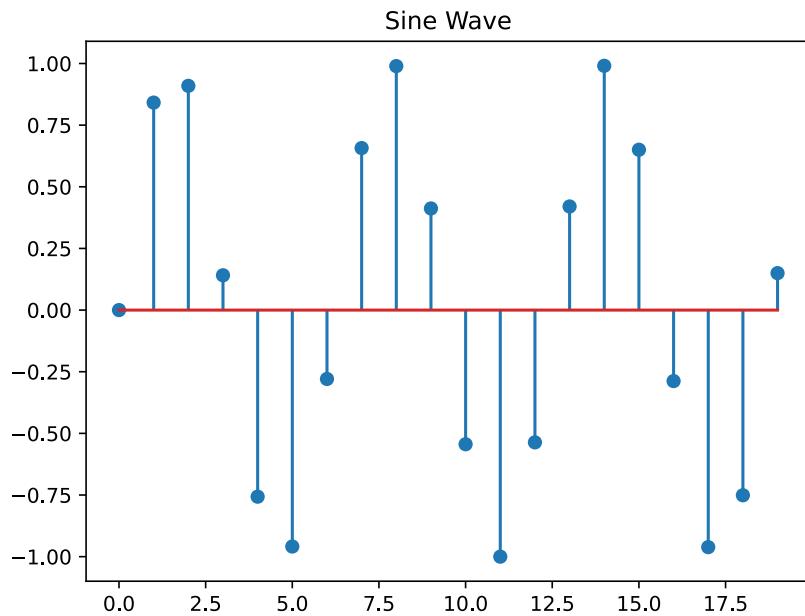


Providing the histogram function bin edges offers far more control to the user, but writing out a list can be tedious. As an alternative, the histogram function also accepts bin edges as `range()` objects. Unfortunately, Python's built-in `range()` function only generates values with integer steps. As an alternative, you can use list comprehension from chapter 2 or use NumPy's `np.arange()` function from [section 4.1.3](#) which does allow non-integer step sizes.

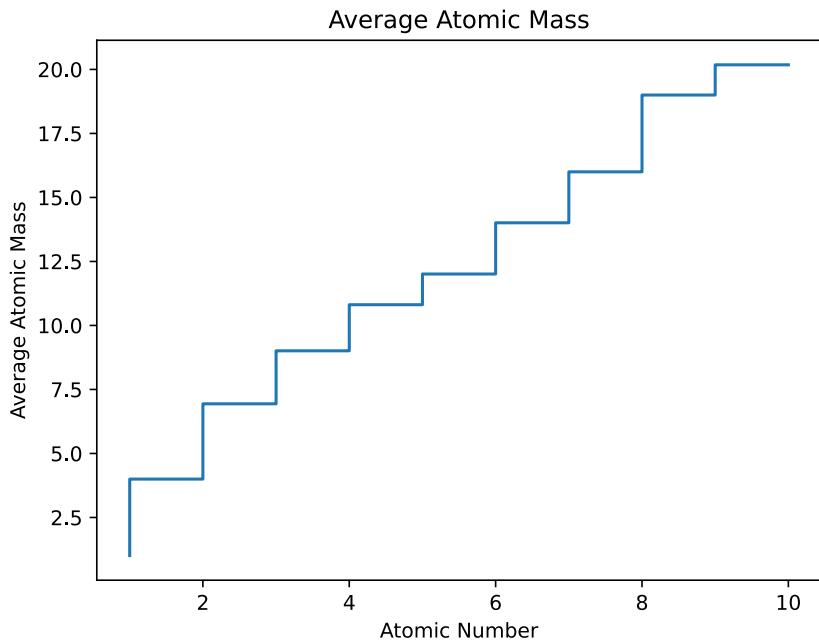
3.2.4 Other Plotting Types

There are a variety of other two dimensional plotting types available in the matplotlib library including stem, step, pie, polar, 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.

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

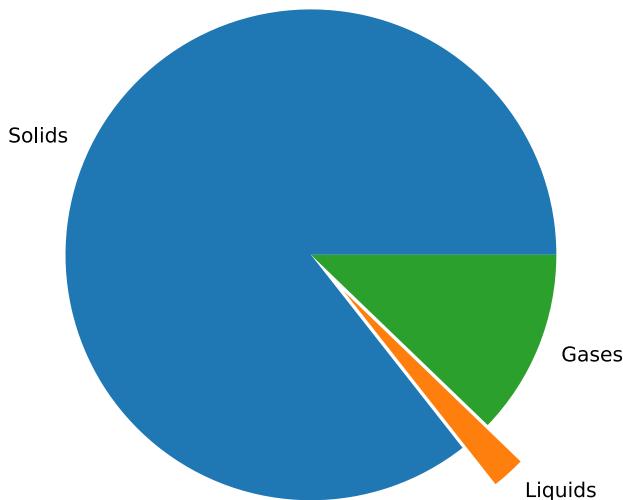


```
AN = range(1, 11)
mass_avg = [1.01, 4.00, 6.94, 9.01,
10.81, 12.01, 14.01, 16.00, 19.00,
20.18]
plt.step(AN, mass_avg)
plt.title('Average Atomic Mass')
plt.xlabel('Atomic Number')
plt.ylabel('Average Atomic Mass');
```

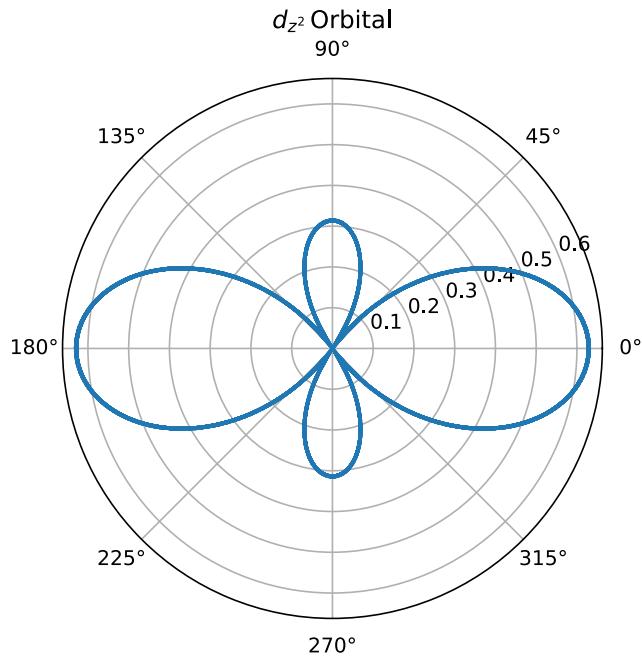


```
labels = ['Solids', 'Liquids', 'Gases']
percents = (85.6, 2.2, 12.2)
plt.title('Naturally Occurring Elements')
plt.pie(percents, labels=labels,
explode=(0, 0.2, 0))
plt.axis('equal');
```

Naturally Occurring Elements



```
import numpy as np
theta = np.arange(0, 360, 0.1)
r = [abs(math.sqrt(5 / (16 * math.pi)) *
         (3 * math.cos(num)**2 - 1)) for num in theta]
plt.polar(theta, r)
plt.title(r'$d_{z^2}$ ,,$ + 'Orbital');
```



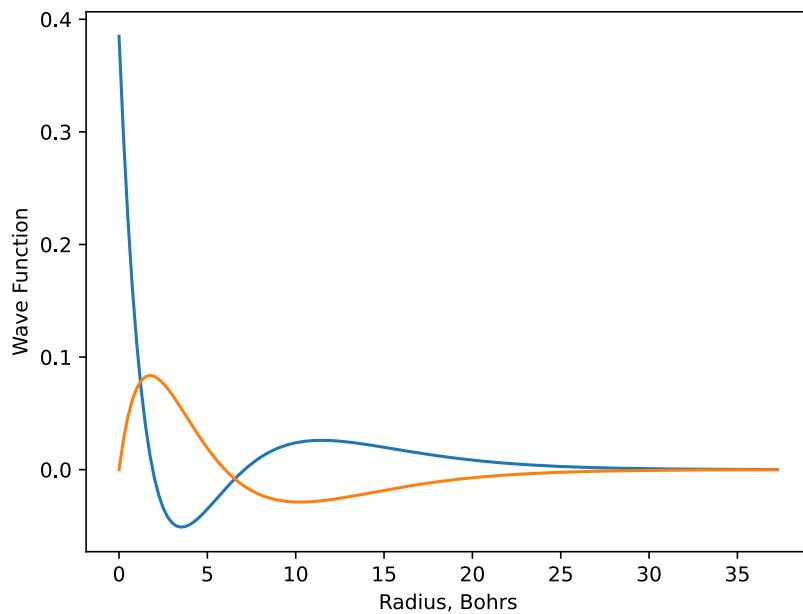
3.3 Overlaying Plots

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 3p and 3s orbitals can be plotted on the same set of axes.

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

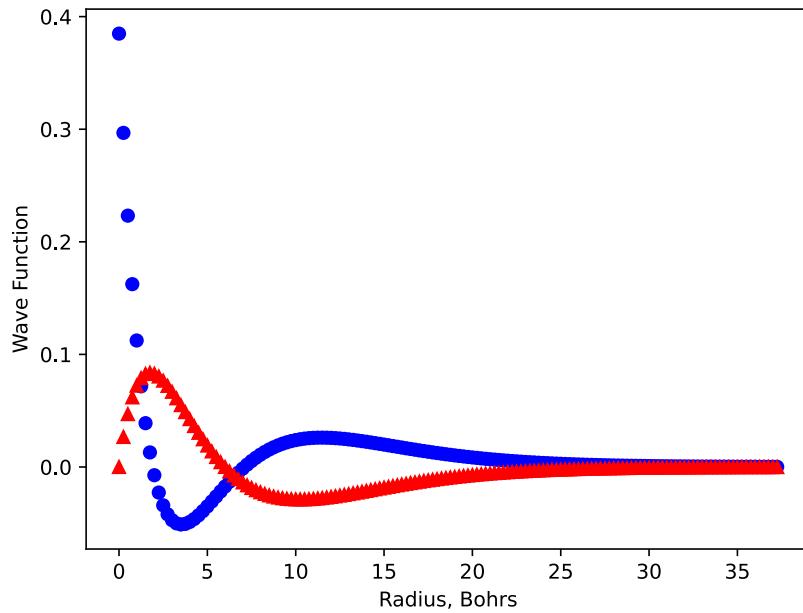
r = [num / 4 for num in range(0, 150)]
psi_3p = [orbital_3P(num) for num in r]
psi_3s = [orbital_3S(num) for num in r]
```

```
plt.plot(r, psi_3s)
plt.plot(r, psi_3p)
plt.xlabel('Radius, Bohrs')
plt.ylabel('Wave Function');
```



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.

```
plt.plot(r, psi_3s, 'bo', r, psi_3p,'r^')
plt.xlabel('Radius, Bohrs')
plt.xlabel('Radius, Bohrs')
plt.ylabel('Wave Function');
```



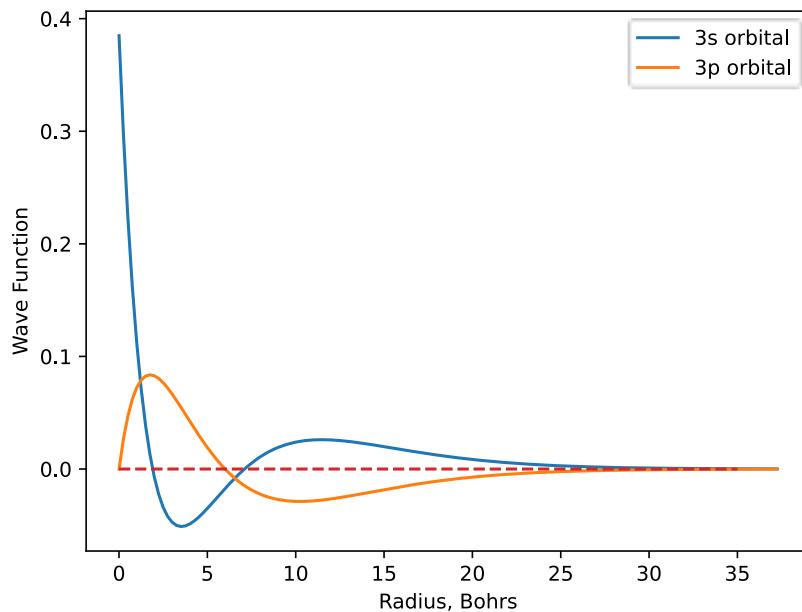
In the second plot above, `r`, `psi_3s`, `'bo'` are the data and style for the first set of data while `r`, `psi_3p`, `'r^'` are the data and plotting style for the second.

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 dataset 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. A complete list of location arguments are available on the [matplotlib website](#).

It would also be helpful to include a horizontal line at zero as a guide to the eye. Matplotlib includes a `plt.hlines(y, xmin, xmax)` function for just this purpose, and this function takes similar arguments for color and style.

```
plt.plot(r, psi_3s, label='3s orbital')
plt.plot(r, psi_3p, label='3p orbital')
plt.hlines(0, 0, 35, linestyle='dashed', color='C3')
plt.xlabel('Radius, Bohrs')
plt.ylabel('Wave Function')
plt.legend();
```



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. 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.

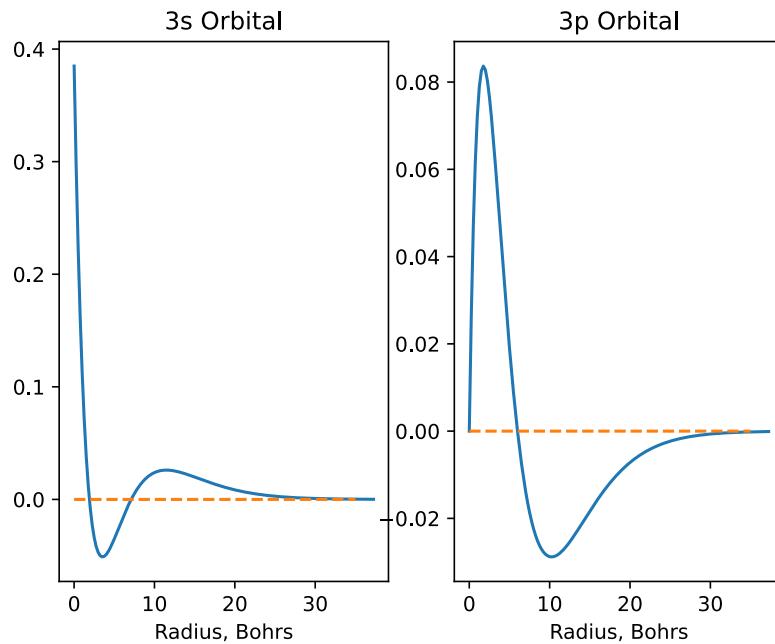
```

plt.figure()

plt.subplot(1,2,1) # first subplot
plt.plot(r, psi_3s)
plt.hlines(0, 0, 35, linestyle='dashed', color='C1')
plt.xlabel('Radius, Bohrs')
plt.title('3s Orbital')

plt.subplot(1,2,2) # second subplot
plt.plot(r, psi_3p)
plt.hlines(0, 0, 35, linestyle='dashed', color='C1')
plt.xlabel('Radius, Bohrs')
plt.title('3p Orbital');

```



If you don't like dimensions of your plot, you can still change them using a `figsize=(width, height)` argument in `figure()` function like the following.

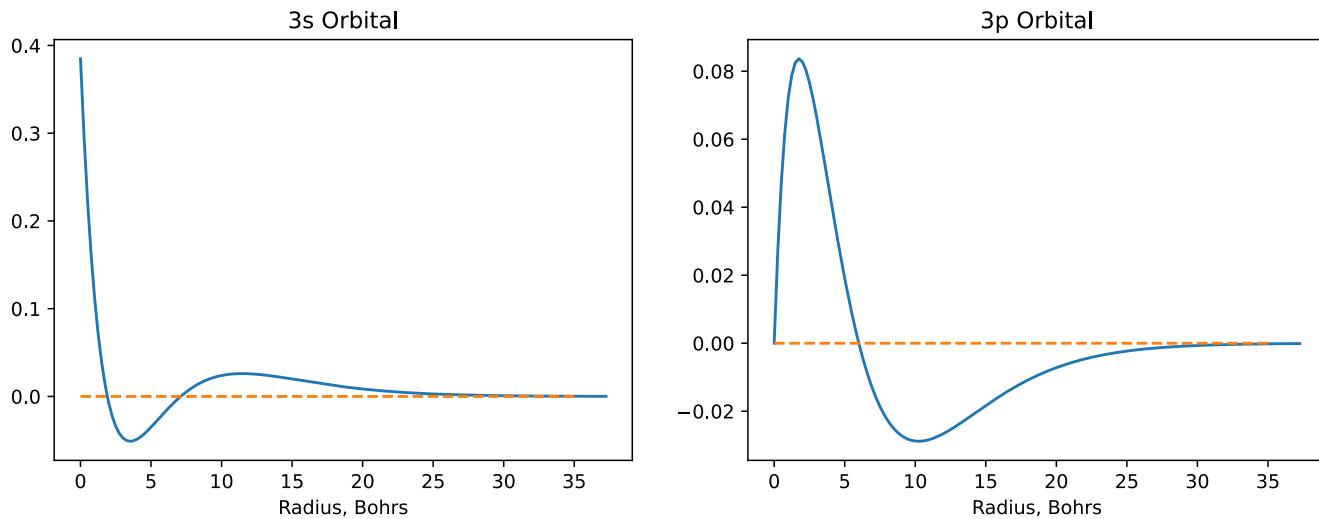
```

plt.figure(figsize=(12,4))

plt.subplot(1,2,1) # first subplot
plt.plot(r, psi_3s)
plt.hlines(0, 0, 35, linestyle='dashed', color='C1')
plt.xlabel('Radius, Bohrs')
plt.title('3s Orbital')

plt.subplot(1,2,2) # second subplot
plt.plot(r, psi_3p)
plt.hlines(0, 0, 35, linestyle='dashed', color='C1')
plt.xlabel('Radius, Bohrs')
plt.title('3p Orbital');

```



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 1). In this example, the top subplot dimension is created as if though it is the first subplot in a 2×1 figure. The bottom two subplot dimensions are created as if they are the third and fourth subplots in a 2×2 figure.

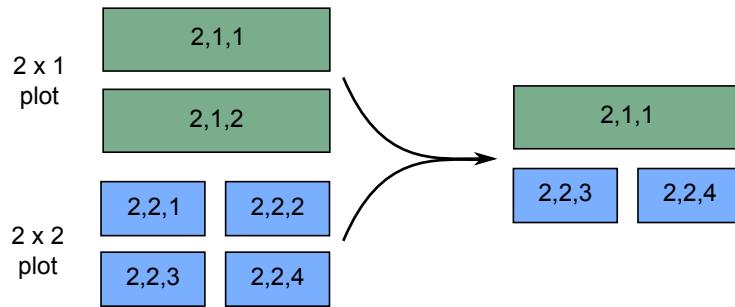


Figure 1 Multifigure plots with subplots of different dimensions (right) describe each subplot dimension as if it were part of a plot with equally sized subplots (left).

In the following example, dihedral angle data contained in a hydrogenase enzyme from [Nat. Chem. Biol. 2016, 12, 46-50](#) is important and displayed. The top plot shows the relationship between the psi (ψ) and phi (ϕ) angles while the bottom two plots show the distribution of psi and phi angles using histogram plots.

```
rama = np.genfromtxt('data/hydrogenase_5a4m_phipsi.csv',
                     delimiter=',', skip_header=1)

psi = rama[:,0]
phi = rama[:,1]
```

```

plt.figure(figsize=(10,8))

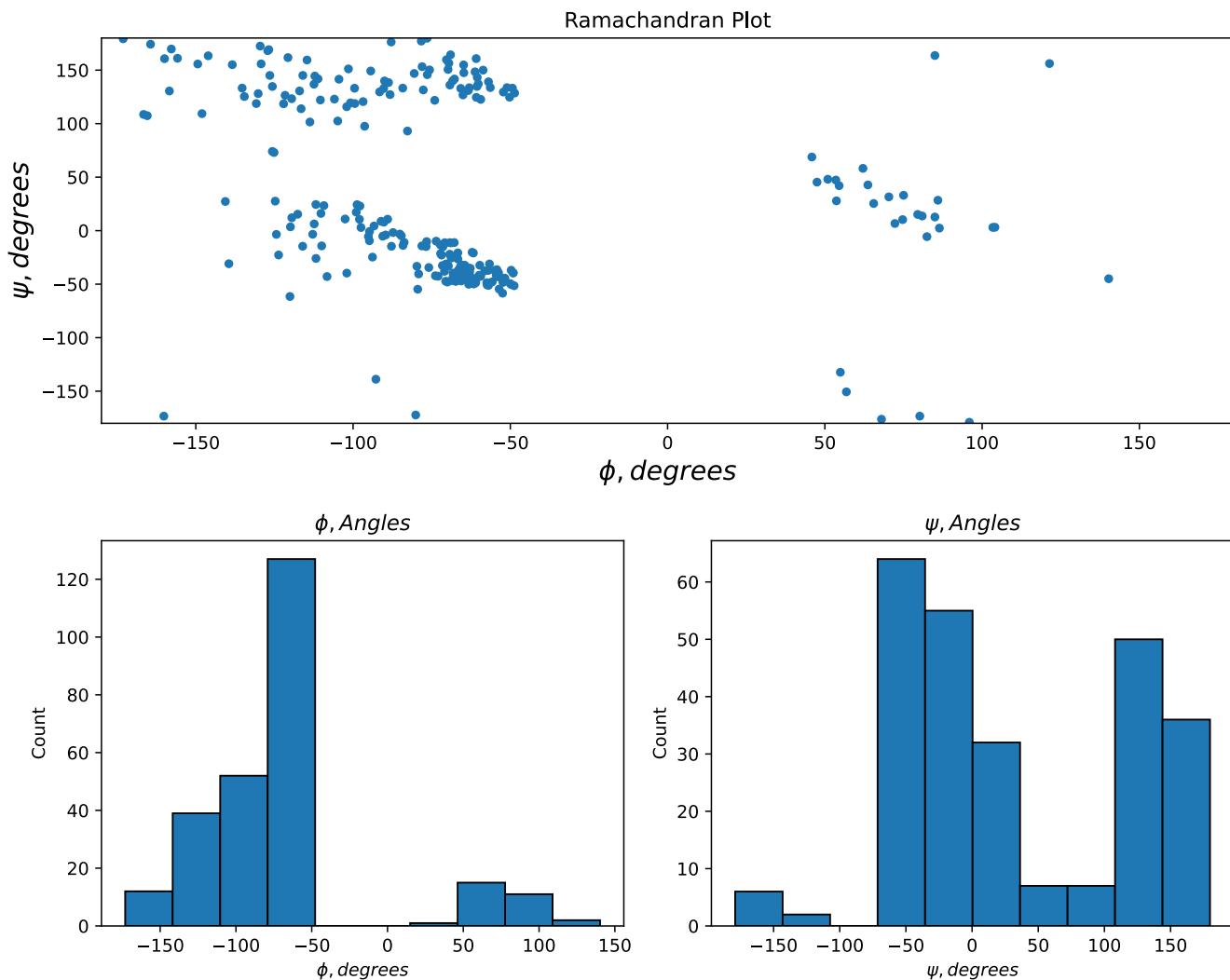
plt.subplot(2,1,1)
plt.plot(phi, psi, '.', markersize=8)
plt.xlim(-180, 180)
plt.ylim(-180, 180)
plt.xlabel('$\phi$, degrees', fontsize=15)
plt.ylabel('$\psi$, degrees', fontsize=15)
plt.title('Ramachandran Plot')

plt.subplot(2,2,3)
plt.hist(phi[1:], edgecolor='k')
plt.xlabel('$\phi$, degrees')
plt.ylabel('Count')
plt.title('$\phi$ , Angles')

plt.subplot(2,2,4)
plt.hist(psi[:-1], edgecolor='k')
plt.xlabel('$\psi$, degrees')
plt.ylabel('Count')
plt.title('$\psi$ , Angles')

plt.tight_layout();

```



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 `add_subplot(rows, columns, plot_number)` command. The arguments in the `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 your plotting function preceded by the axis variable name as demonstrated below.

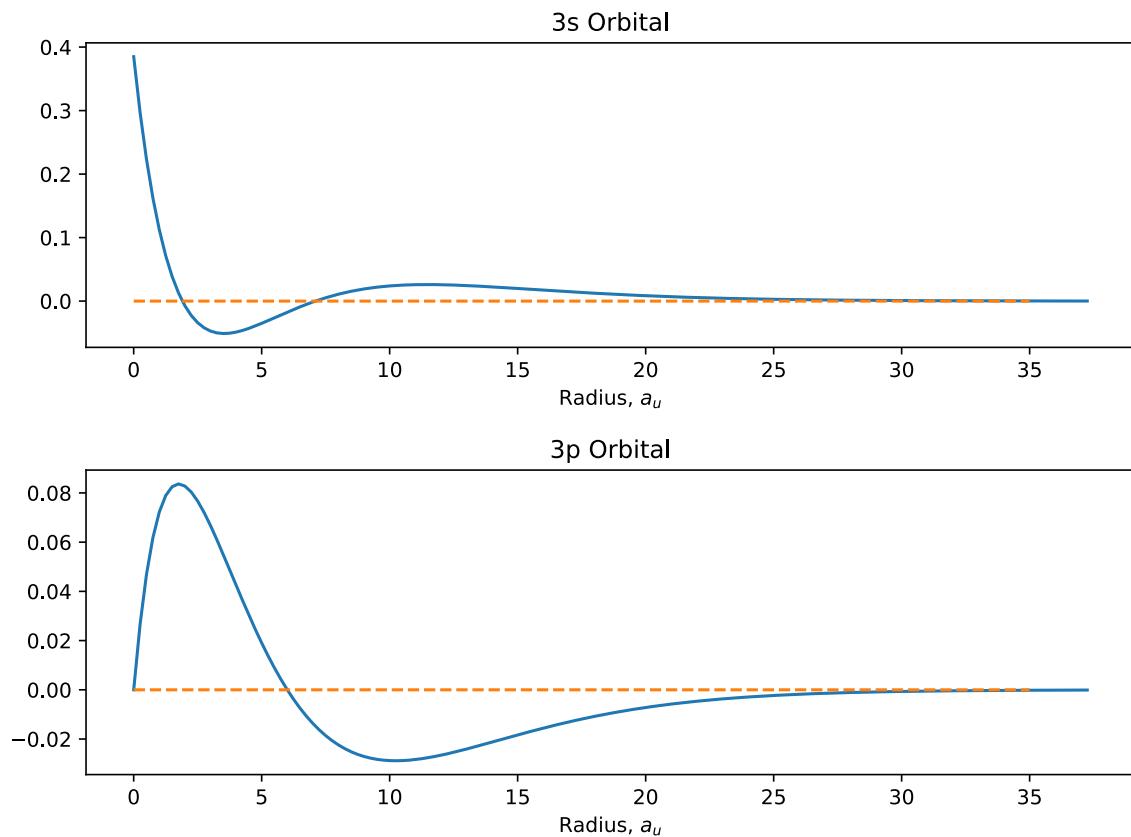
One noticeable difference in this method is the functions for customizing the plots are typically preceded with `set_` such as `set_title()`, `set_xlim()`, or `set_ylabel()`.

```
fig = plt.figure(figsize=(8,6))

ax1 = fig.add_subplot(2,1,1)
ax1.plot(r, psi_3s)
ax1.hlines(0, 0, 35, linestyle='dashed', color='C1')
ax1.set_title('3s Orbital')
ax1.set_xlabel('Radius, $a_u$')

ax2 = fig.add_subplot(2,1,2)
ax2.plot(r, psi_3p)
ax2.hlines(0, 0, 35, linestyle='dashed', color='C1')
ax2.set_title('3p Orbital')
ax2.set_xlabel('Radius, $a_u$')

plt.tight_layout();
```



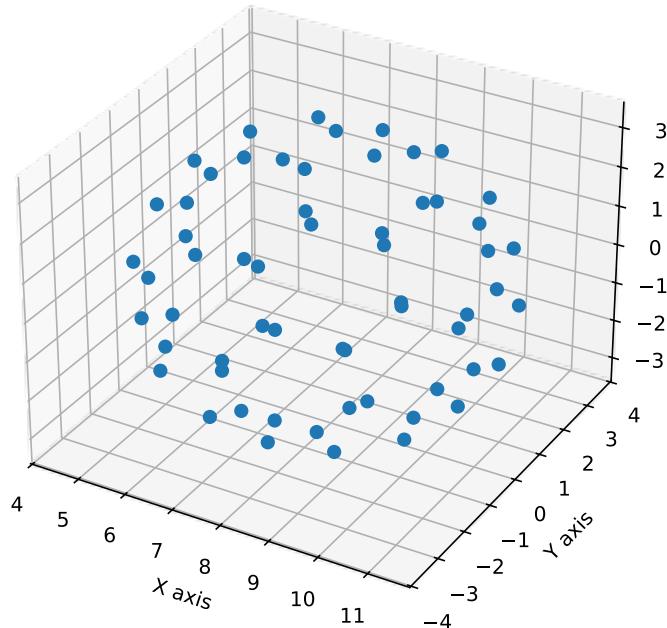
3.5 3D Scatter Plots

To plot in 3D, we will use the approach outlined in [section 3.4.2](#) with two additions. First, add `from mpl_toolkits.mplot3d import Axes3D` 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 the following example, we will import xyz -coordinates for a C_{60} buckyball molecule and plot the carbon atom positions in 3D.

```
from mpl_toolkits.mplot3d import Axes3D
C60 = np.genfromtxt('data/C60.csv', delimiter=',', skip_header=1)
x, y, z = C60[:,0], C60[:,1], C60[:,2]
```

```
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');
```



3.6 Surface & Wireframe Plots

The above 3D plot is simply a scatter plot in a three-dimensional space, but it is often useful to connect these points to describe surfaces in 3D space which can be used for displaying energy surfaces, chemical spectra, or atomic orbital shapes among other applications. We again will import `Axes3D` from `mpl_toolkits.mplot3d` as we did in [section 3.5](#). When choosing matplotlib functions below, it depends not only on what you want your surface to look like but also upon the format of the data. Specifically, your data may be in a grid or xyz format. Below addressed both scenarios.

3.6.1 Gridded Data

If the height data are formated as a grid, we will need to generate a mesh grid of the x - and y -axis locations to create a surface plot. Mesh grids are simply the x - and y -axes values extended into a 2D array. An 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.

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

Z Values Grid								
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 for example, the bottom left value is 64 because it is the sum of 64 and 0.

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.

```

import numpy as np

x = np.arange(-10, 10)
y = np.arange(-10, 10)

X, Y = np.meshgrid(x, y)
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 (`cmap`). 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](#).

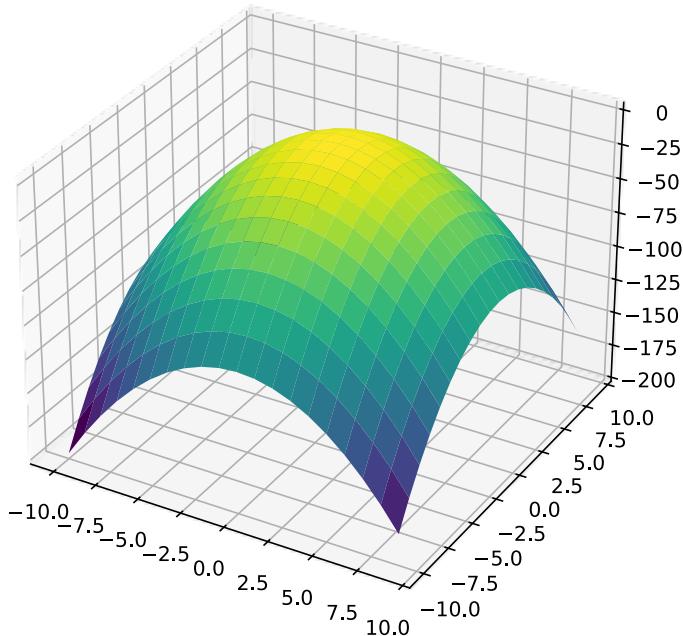
```

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');

```



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.

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

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.

```

L = 1
nx = 2
ny = 1

x = np.linspace(0, L, 20)
y = np.linspace(0, L, 20)
X, Y = np.meshgrid(x,y)

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

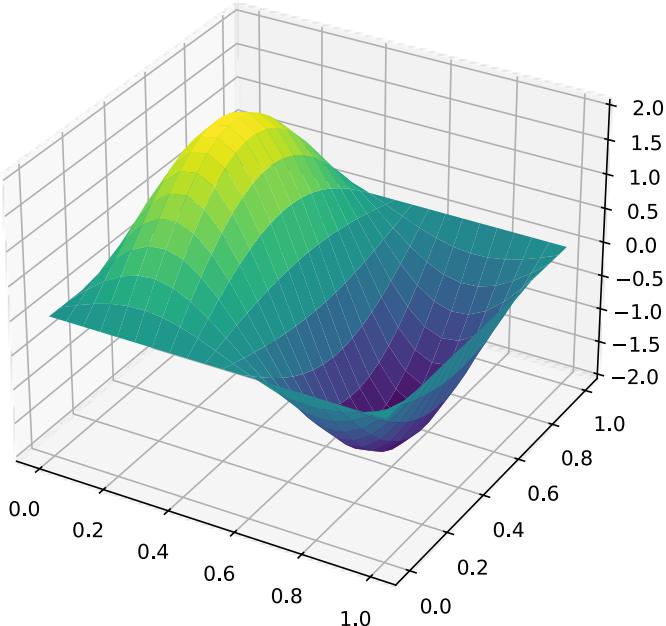
Z = wave(x, y, nx, ny)

```

```

fig = plt.figure(figsize=(10,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis');

```



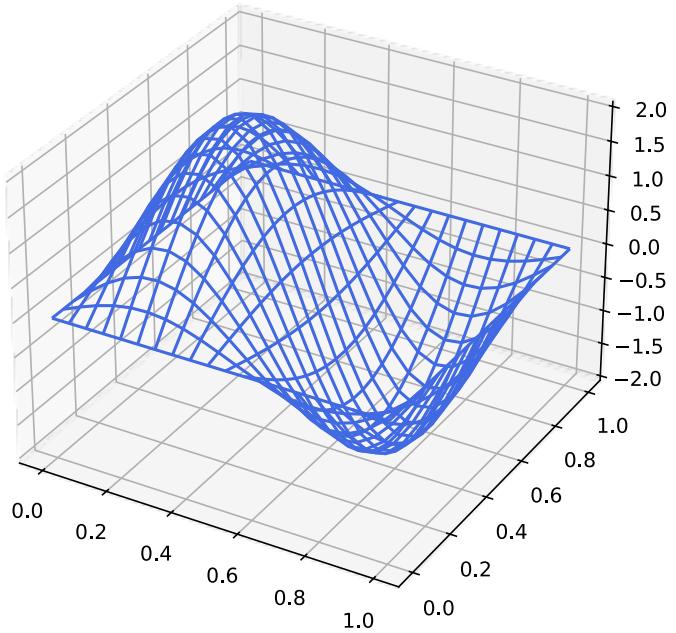
You are encouraged to increase the values for n_x and n_y and see how the surface plot changes.

Alternatively, a surface can be represented with a wireframe using the `plt.plot_wireframe()` function which operates similarly to the `plt.plot_surface()` function.

```

fig = plt.figure(figsize=(12,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_wireframe(X, Y, Z, linewidths=1.5, colors='royalblue');

```



3.6.2 Trigonal Plots

If the data are formatted as three columns containing x , y , and z values, matplotlib provides triangulated grid function, `plt.plot_trisurf()`, that can work with these data. Because the function cannot guarantee that the data points are arranged in rectangular grids, the surface mesh is instead composed of triangular faces. The function takes the x , y , and z values as the required arguments. As an example, the data from the above standing wave are repacked below as a series of xyz vector coordinates and plotted using the `plt.plot_trisurface()`.

```
# repack data in xyz vectors for example
wave_d = np.dstack((X, Y, Z))
wave_xyz = []
for layer in wave_d:
    for vect in layer:
        wave_xyz.append(vect)

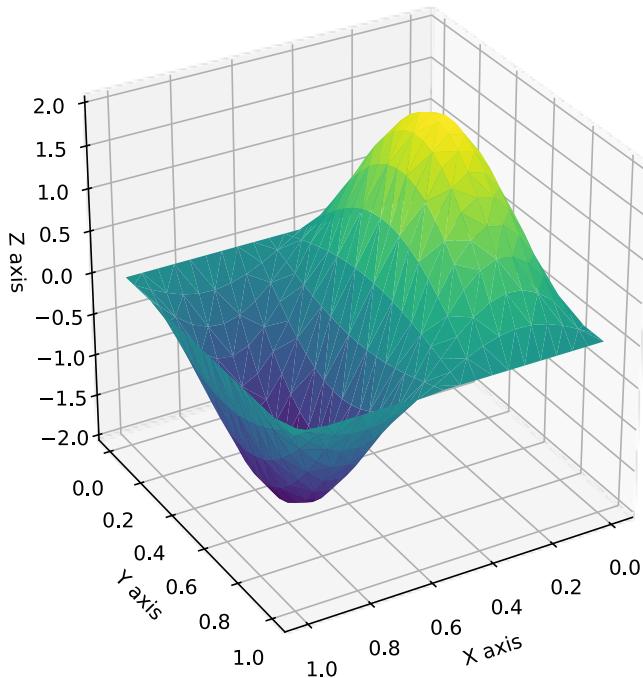
wave_xyz = np.array(wave_xyz)
```

```
x, y, z = wave_xyz[:,0], wave_xyz[:,1], wave_xyz[:,2]

fig = plt.figure(figsize=(14,6))
ax = fig.add_subplot(1,1,1, projection='3d', )
ax.plot_trisurf(x, y, z, cmap='viridis')

# adjusts view
ax.view_init(azim=60, elev=30)
# prevents z label from being cut off
ax.set_box_aspect(aspect=(1,1,1))

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



3.6.3 3D Surfaces

Matplotlib supports the ability to plot 3D surfaces and wireframes which is useful for molecular orbitals among other applications. We will start with a basic sphere and then morph it into the angular component of an atomic orbital. We are going to again use the `plt.plot_surface()` and `plt.plot_wireframe()` functions, so we first need a mesh grid using the `np.meshgrid()` function to yield the theta (θ) and phi (ϕ) values. There are multiple conventions for these angles, but here we will follow the [SciPy convention](#) which treats phi as the azimuthal angle (i.e., direction on xy -plane) and theta as the polar angle (i.e., angle off the positive z -axis). The values for phi do a full circle, ranging from $0 \rightarrow 2\pi$, while theta here swings from the north pole to the south pole, ranging from $0 \rightarrow \pi$. These angle are then converted to xyz-coordinates using the trigonometric equations shown below. In this example, we are plotting a unit sphere, so $r = 1$. Finally, the x , y , and z values are provided to either the `plt.plot_surface()` or `plt.plot_wireframe()` functions to plot a sphere. It is important here to set the aspect ratio to `equal` using `ax.set_aspect('equal')` so that equal changes in value are represented with equal distances along all axes. Otherwise, the z -axis will be compressed here making the sphere look squished or oblate.

$$x = r \sin(\theta) \sin(\phi)$$

$$y = r \sin(\theta) \cos(\phi)$$

$$z = r \cos(\theta)$$

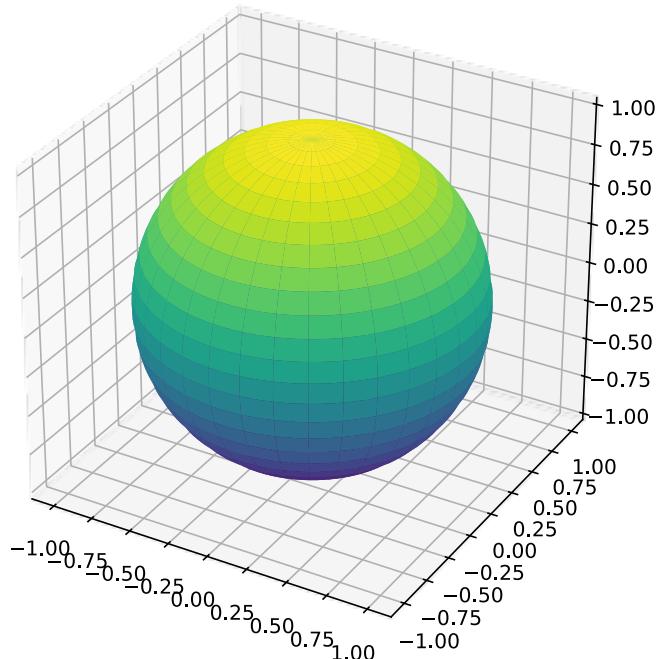
```

# generate mesh grid of theta and phi angles
th, ph = np.meshgrid(np.linspace(0, np.pi, 51),
                      np.linspace(0, 2 * np.pi, 101))

# convert angles to xyz coordinates for r = 1
x = np.sin(th) * np.sin(ph)
y = np.sin(th) * np.cos(ph)
z = np.cos(th)

# plotting
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_aspect('equal') # sets aspect ratio to equal

```



To plot orbital angular components, we can modify or warp our sphere by multiplying the xyz -coordinates by the orbital's angular wave function. We are essentially changing the radius at different angles in the trigonometric equations above. For example, below is the angular wave function for the d_{z^2} orbital.

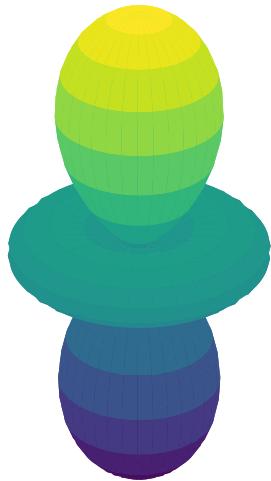
$$Y_{d_{z^2}} = \left(\frac{5}{16\pi} \right)^{1/2} (3 \cos^2 \theta - 1)$$

```

# multiply xyz values by angular wave function
dz2 = np.sqrt((5 / 16) * np.pi) * (3 * np.cos(th)**2 - 1)
X, Y, Z = x * dz2, y * dz2, z * dz2

# plotting
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_aspect('equal') # sets aspect ratio to equal
ax.set_axis_off() # turns off axes and background

```



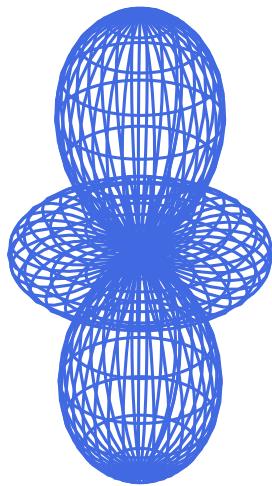
While the angular wave functions can be coded manually, the SciPy library includes a [spherical harmonics](#) `sph_harm_y(l, m, theta, phi)` function that will calculate the angular wave function for any combination of the angular (l) and magnetic (m_l) quantum numbers. We only want the positive, real results, so we will take the absolute value of the real component.

```
from scipy.special import sph_harm_y

# calculate spherical harmonic
l, m = 2, 0
harm = sph_harm_y(l, m, th, ph)
f = np.abs(harm)

# multiply xyz values by wave function
X, Y, Z = x * f, y * f, z * f

# plotting
fig = plt.figure(figsize = (10, 6))
ax = fig.add_subplot(1, 1, 1, projection='3d')
ax.plot_wireframe(X, Y, Z, colors='royalblue')
ax.set_aspect('equal') # sets aspect ratio to equal
ax.set_axis_off() # turns off axes and background
```



3.7 3D Data on a 2D Surface

There are times when it is useful to represent 3D data on a 2D surface requiring the third dimension to be represented by color or contour lines. This can be useful for representing an energy surface, 3D fluorescence spectra where the x - and y -axes are absorption and emission wavelengths, or 2D NMR spectra. This section demonstrates a number of plotting functions in matplotlib to generate 2D histograms and contour plots.

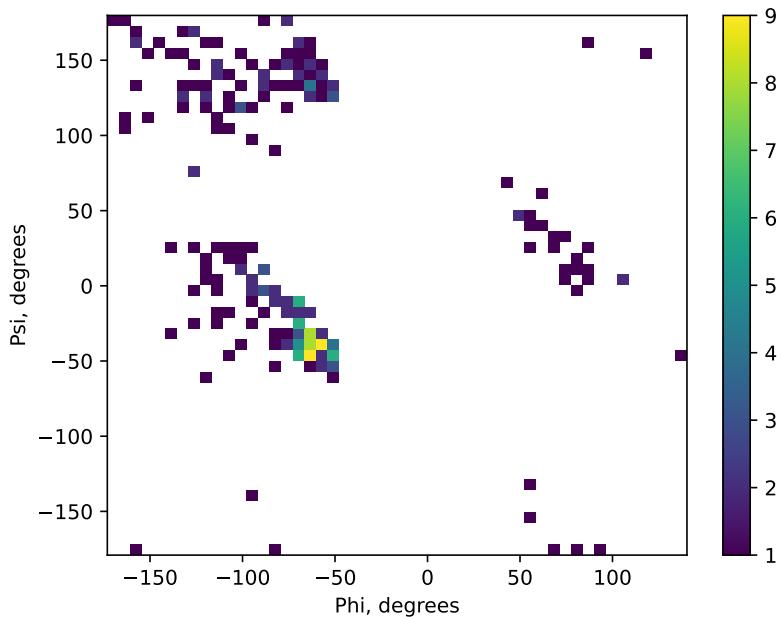
3.7.1 2D Histograms

The first plot we will cover is the 2D histogram. This is similar to the standard histogram except that the bins are 2D and the quantity in a bin is represented by color instead of a bar height. There are two functions available in matplotlib for this task listed below. Each of these functions requires the x - and y -coordinates as the two required arguments, and like the previously seen histogram function, these functions total the counts in each bin for the user. For this example, we will again use the Ramachandran data from [section 3.4.1](#).

```
plt.hist2d(x, y)
plt.hexbin(x, y)
```

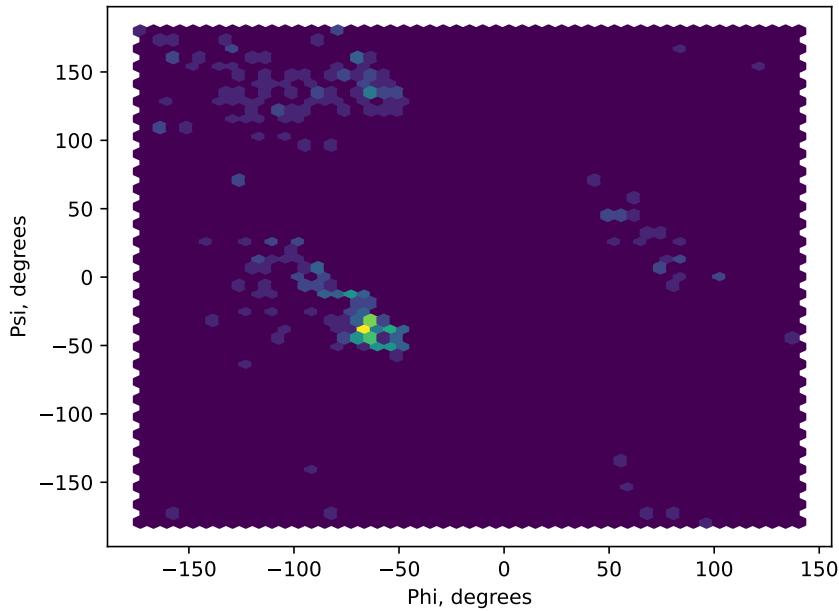
The `plt.hist2d()` function, like the regular histogram function, can accept additional arguments such as the number or position of the bins (`bins=`) or minimum or maximum values for bins to be displayed (`cmin=` and `cmax=`, respectively). In the example below, there are 50 bins on each axis, and any bin with fewer than 1 count is not displayed.

```
plt.hist2d(phi, psi, bins=50, cmin=1)
plt.xlabel('Phi, degrees')
plt.ylabel('Psi, degrees')
plt.colorbar();
```



The `plt.hexbin()` function in its basic form is like the `plt.hist2d()` function except that the bins are hexagons instead of rectangles.

```
plt.hexbin(phi, psi, gridsize=50, vmax=10)
plt.xlabel('Phi, degrees')
plt.ylabel('Psi, degrees');
```



3.7.2 Contour Plots

We will next look at contour plots which show the z values using color or lines. When lines are used, this is similar to a topographic map where the closer the lines, the steeper the change in z values. The lines are also colored to show the values. Like plotting 3D surfaces in [section 3.6](#), the data may be represented as either three grids or a series of xyz values.

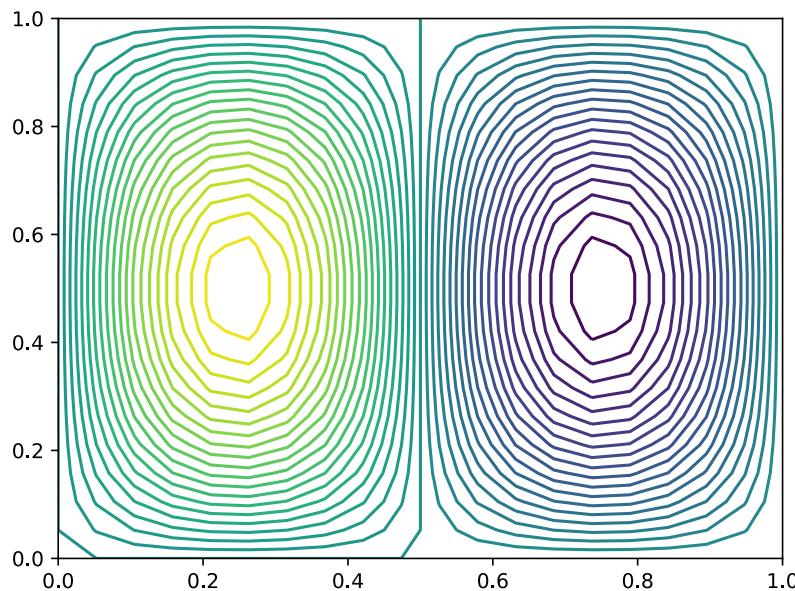
For our gridded example, we will again visualize our standing wave function from [sections 3.6](#). The `plt.contour()` plot accepts x, y, and z grids as the required arguments, but it can also accept the number of levels (`levels=`) and a colormap (`cmap=`).

```
L = 1
nx = 2
ny = 1

x = np.linspace(0, L, 20)
y = np.linspace(0, L, 20)
X, Y = np.meshgrid(x,y)

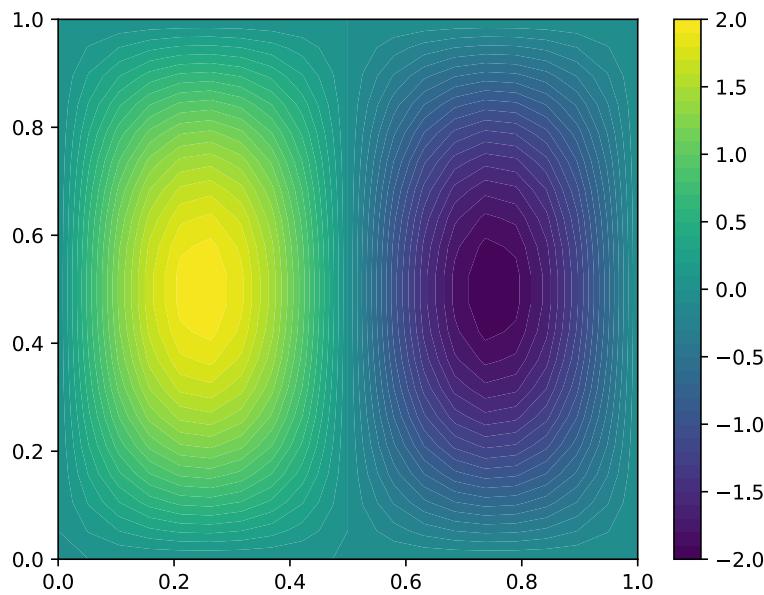
def wave(x, y):
    psi = (2/L) * np.sin(nx*np.pi*X/L) * np.sin(ny*np.pi*Y/L)
    return psi
```

```
plt.contour(X, Y, wave(X, Y), cmap='viridis', levels=40);
```



We can also generate a contour plot where the space between the lines is filled using the `plt.contourf()` function. The "f" is for "filled".

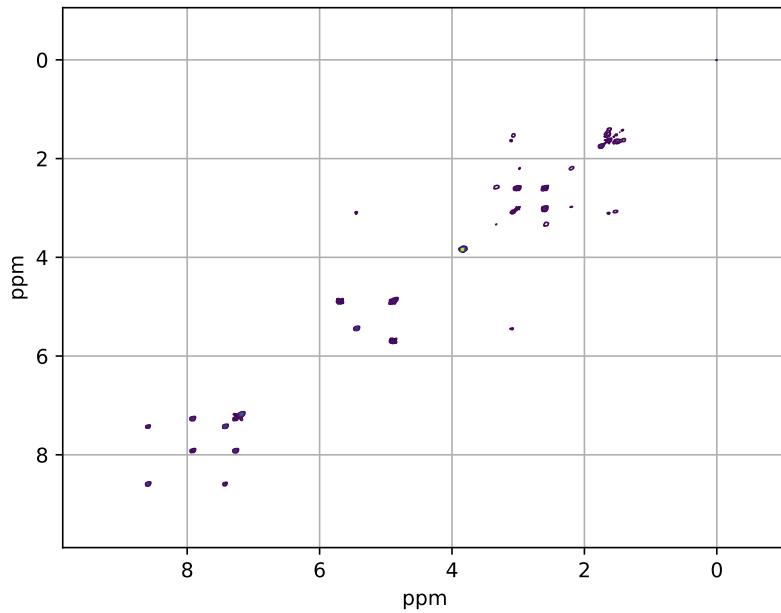
```
plt.contourf(X, Y, wave(X, Y), cmap='viridis', levels=40)
plt.colorbar();
```



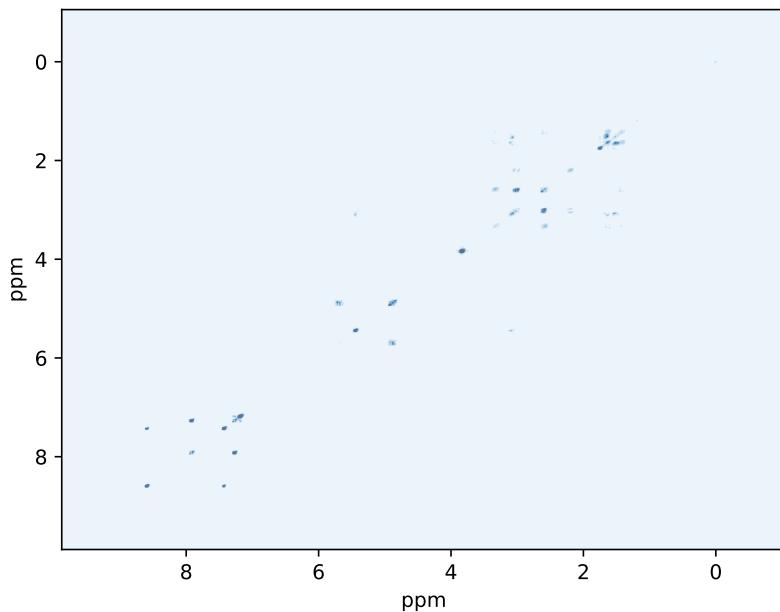
If the data are in xyz coordinate format, we will instead use the `plt.tricontour()` or `plt.tricontourf()` functions as demonstrated below with COSY NMR data of quinine.

```
COSY = np.genfromtxt('data/Quinine_CDCl3_COSY.csv', delimiter=',', skip_header=1)
x, y, z = COSY[:,0], COSY[:,1], COSY[:,2]

plt.tricontour(x, y, z, levels=100, linewidths=0.8)
plt.gca().invert_xaxis()
plt.gca().invert_yaxis()
plt.xlabel('ppm')
plt.ylabel('ppm')
plt.grid(which='major');
```



```
plt.tricontourf(x, y, z, levels=200, vmax=0.05, cmap='Blues')
plt.gca().invert_xaxis()
plt.gca().invert_yaxis()
plt.xlabel('ppm')
plt.ylabel('ppm');
```

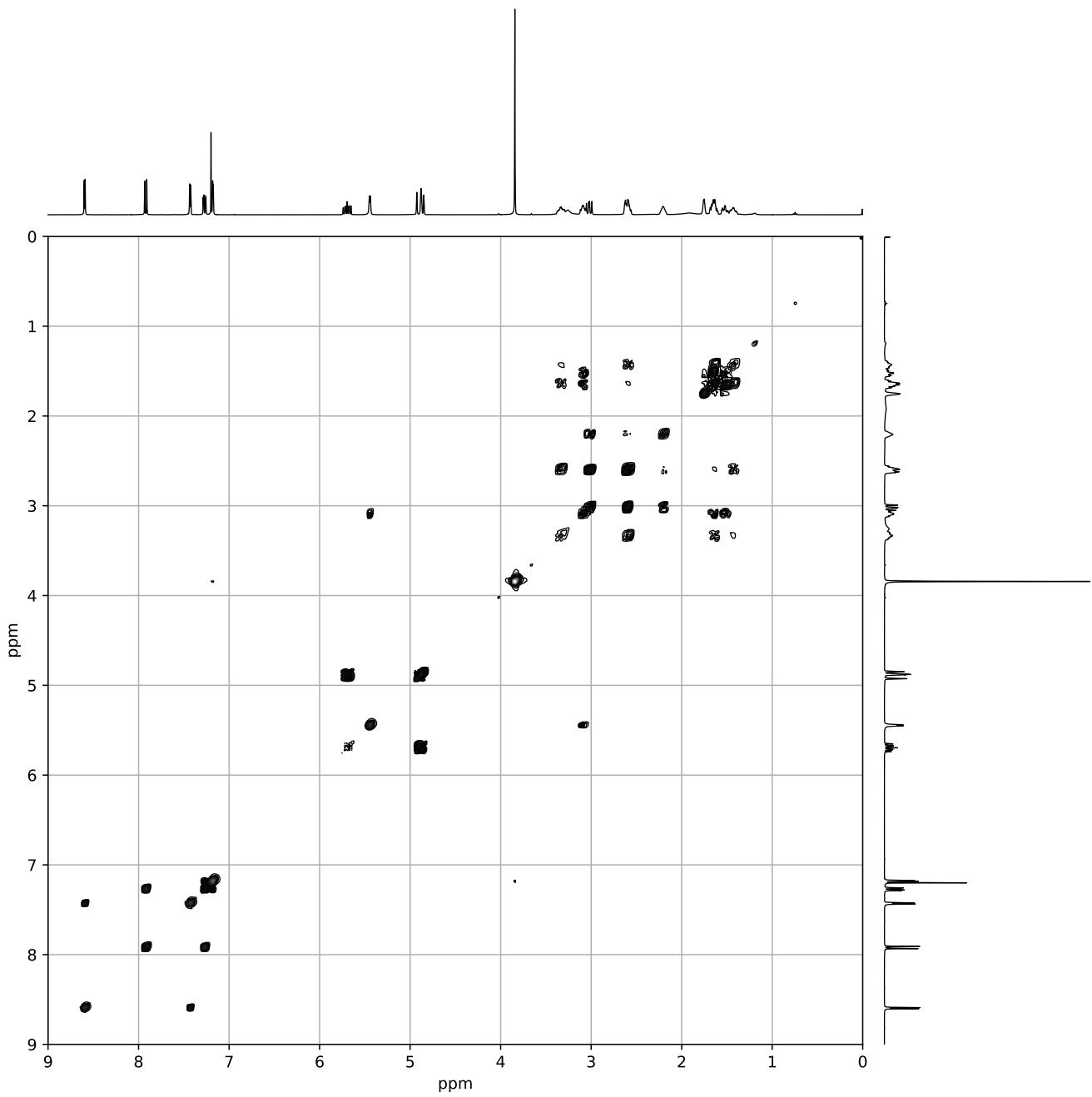


As a final example, it is possible to merge a contour plot with a line plot. This is useful for representing 2D NMR spectra such as COSY NMR where the COSY NMR data is represented by the contour plot while the ^1H NMR spectrum is located on the margins of the contour plot. Below, a function `plot_2d_nmr()` is defined (in collapsed cell) to generate such a plot.

► Show code cell source

```
proton = np.genfromtxt('data/Quinine_CDCl3_1HNMR.csv', delimiter=',', skip_header=1)
cosy = np.genfromtxt('data/Quinine_CDCl3_COSY.csv', delimiter=',', skip_header=1)

plot_2d_nmr((cosy[:,0], cosy[:,1], cosy[:,2]), (proton[:,0], proton[:,1]),
             limits=(9,0), levels=300, linewidths=0.7, grayscale=True)
```



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. The matplotlib website also provides free cheatsheets summarizing key features and functions.

1. Matplotlib Website. <https://matplotlib.org> (free resource)
2. Matplotlib Cheatsheets <https://matplotlib.org/cheatsheets/> (free resource)
3. VanderPlas, J. Python data Science Handbook: Essential Tools for Working with Data, 1st ed.; O'Reilly: Sebastopol, CA, 2017, chapter 4. Freely available from the author at <https://jakevdp.github.io/PythonDataScienceHandbook/> (free resource)

4. Matplotlib Colormap Reference https://matplotlib.org/stable/gallery/color/colormap_reference.html (free resource)
5. Matplotlib Marker Reference https://matplotlib.org/stable/api/markers_api.html (free resource)

Exercises

Complete the following exercises in a Jupyter notebook using the matplotlib library and be sure to **label axes and include units** when appropriate. Any data file(s) referred to in the problems can be found in the [data](#) folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

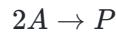
1. Visualize the relationship between pressure and volume for 1.00 mol of He(g) at 298 K in an expandable vessel as it increases from 1 L → 20 L. $R = 0.08206 \text{ L}\cdot\text{atm/mol}\cdot\text{K}$. This will require you to generate values and perform the calculating using the equation below.

$$PV = nRT$$

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](#) 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 $-50 \rightarrow 50$. Be sure to include a legend.

$$f(x) = x^2 \quad g(x) = x^2 \sin(x) \quad h(x) = -x^2$$

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 time (x-axis) using a line plot. Be sure to label the axes.
6. 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='None'`.
7. Earth's atmosphere is composed of 78% N₂, 21% O₂, and 1% other gases. Represent this data with a pie chart, and make the last 1% slice stick out of the pie like in [section 3.2.4](#).
8. Create a histogram plot to examine the distribution of values generated below.

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

9. The ¹H NMR spectrum of caffeine in CDCl₃ 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='None'`.

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

10. The following table presents the calculated free energies for each step in the binding and splitting of H₂(g) by a nickel phosphine catalyst. Visualize the energies over the course of the reaction using a plotting type other than a line or scatter plot. Data from [Inorg. Chem. 2016, 55, 445–460](#).

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

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, including [OpenStax](#) in the periodic trends chapter. Include titles on both plots along with appropriate axis labels.
12. Generate a standing wave surface plot (similar to the one at the end of [section 3.6](#)) using the following equation and parameters: $L = 1$, $n_x = 2$, $n_y = 2$.
- $$\Psi(x, y) = (2/L) \sin(n_x\pi x/L) \sin(n_y\pi y/L)$$
13. Load the **amine_bp.csv** file in the data folder which contains the boiling points of primary, secondary, and tertiary amines and the number of carbons in each amine. Plot the boiling point (*x*-axis) versus number of carbons (*y*-axis) for each degree of amine. Your plot should have three distinct trends, one for each degree, represented both in different colors and with different markers. Include a legend on your plot indicating which data points represent which degree of amine.
14. Visualize the angular component of a *d*-orbital other than d_{z^2} and identify which *d*-orbital you visualized the angular component for. You will need to find a [table of the real components of spherical harmonics](#) for this task.

Chapter 4: NumPy

Contents

- 4.1 NumPy Arrays
- 4.2 Reshaping & Merging Arrays
- 4.3 Indexing Arrays
- 4.4 Vectorization & Broadcasting
- 4.5 Array Methods
- 4.6 Missing Data
- 4.7 Random Number Generation
- Further Reading
- Exercises

NumPy is a popular library in the Python ecosystem and a critical component of the SciPy stack. So much so that NumPy is even included in Apple's default installation of Python and in other Python-powered applications such as [Blender](#). 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 array 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 faster

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.

```
import numpy as np
```

4.1 NumPy Arrays

One of the main contributions of NumPy is the *ndarray* (i.e., "n-dimensional array"), *NumPy 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](#), [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 automatically propagates 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.

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

```
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.

```
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print(2 * arr)
```

```
[ 0  2  4  6  8 10 12 14 16 18]
```

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.

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

```
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)
```

```
arr = np.arange(0, 10, 0.5)
arr
```

```
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])
```

```
arr = np.linspace(0, 10, 20)
arr
```

```
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 a tuple of the array dimensions in the form `(rows, columns)`.

```
np.zeros((2, 4))
```

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

```
np.ones(10)
```

```
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. They can also be used to generate arrays filled with other values. For example, to generate an array of threes, an array of zeros can be generated and then incremented by 3.

```
arr = np.zeros((2, 4))
arr += 3
print(arr)
```

```
[[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.

```
def prod(x, y):
    return x * y
```

```
np.fromfunction(prod, (3, 3))
```

```
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.

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

```
counting.size
```

```
6
```

```
counting.shape
```

```
(2, 3)
```

The NumPy convention is to provide the dimensions of a two-dimensional array as (**row, columns**).

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 rows and columns. Because the number of elements is maintained, the new array size needs to be able 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)`.

```
array_1D = np.linspace(0, 9.5, 20)
array_1D
```

```
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 4×5 array.

```
array_2D = np.reshape(array_1D, (4, 5))
array_2D
```

```
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]])
```

As an alternative and preferred way to reshape an array, the `reshape()` function can be used as an array method. Start with the original array and follow it with `.reshape((rows, columns))` like below. This format is often preferred and will be used often herein.

```
array_1D.reshape((4, 5))
```

```
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.

```
array_2D.flatten()
```

```
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 a square bracket at the end of the first row. The dimensions of this array are 1×20 .

4.2.3 Transpose Arrays

Transposing an array rotates the array around the diagonal (Figure 1).

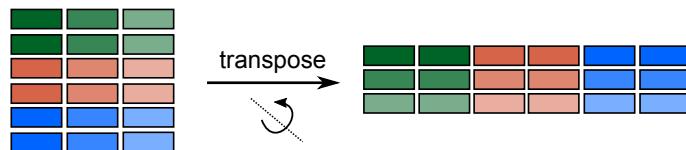


Figure 1 The `np.transpose()` or `array.T` method transposes the NumPy array effectively flipping the rows and columns.

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.

```
array_2D
```

```
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]])
```

```
array_2D.T
```

```
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 2).

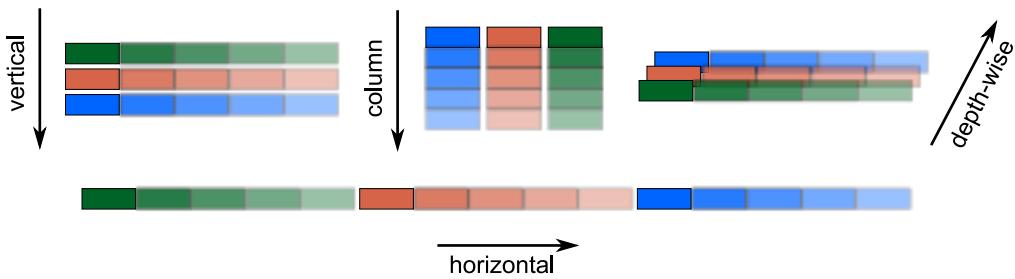


Figure 2 NumPy arrays can be stacked vertically (top left), as columns (top center), depth-wise (top right), or horizontally (bottom) using the `np.vstack()`, `np.column_stack()`, `np.dstack()`, and `np.hstack()` functions, respectively.

```
a = np.arange(0, 5)
b = np.arange(5, 10)
```

```
np.vstack((a, b))
```

```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

```
np.hstack((a, b))
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.dstack((a, b))
```

```
array([[[0, 5],
       [1, 6],
       [2, 7],
       [3, 8],
       [4, 9]]])
```

A related function is the `np.column_stack()` function that stacks the corresponding elements in column lists in a column arrangement.

```
np.column_stack((a, b))
```

```
array([[0, 5],
       [1, 6],
       [2, 7],
       [3, 8],
       [4, 9]])
```

The outcome of the `np.column_stack()` function can also be accomplished by transposing the output of the `np.vstack()` function.

```
np.vstack((a, b)).T
```

```
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.

```
array_1D
```

```
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])
```

```
array_1D[5]
```

```
np.float64(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.

```
array_2D
```

```
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]])
```

```
array_2D[1]
```

```
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]
```

```
array_2D[1][0]
```

```
np.float64(2.5)
```

```
array_2D[1, 0]
```

```
np.float64(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.

```
array_2D[1, 1:]
```

```
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 to use a colon to explicitly indicate all rows.

```
array_2D[0] # implicitly understood all columns
```

```
array([0. , 0.5, 1. , 1.5, 2. ])
```

```
array_2D[0, :] # explicit indicating all columns
```

```
array([0. , 0.5, 1. , 1.5, 2. ])
```

```
array_2D[:, 0] # all rows
```

```
array([0. , 2.5, 5. , 7.5])
```

4.3.3 Advanced Indexing

In the event you have a multidimensional array, you can access elements in the array using multiple collections of values (i.e., ndarrays, lists, or tuples) where each collection indicates the location along a different dimension. This is an instance of *fancy indexing*. For example, if we want to select the following bolded, orange elements from `array_2D`, we can create two lists - the first list contains the row indices for each element and the second list likewise contains the column indices.

$$\begin{bmatrix} 0. & 0.5 & 1. & \textbf{1.5} & 2. \\ 2.5 & 3. & 3.5 & 4. & 4.5 \\ \textbf{5.} & \textbf{5.5} & 6. & 6.5 & 7. \\ 7.5 & 8. & 8.5 & 9. & 9.5 \end{bmatrix}$$

```
row = [2, 2, 0]
col = [0, 1, 3]
```

```
array_2D[row, col]
```

```
array([5. , 5.5, 1.5])
```

Another feature of indexing ndarrays is that the returned array will have the same dimensions as the array containing the indicies. In the following example, we have two index arrays where `i_flat` is a 1×4 array while `i_square` is a 2×2 array resulting in 1×4 and 2×2 arrays, respectively

```
threes = np.arange(3, 30, 3)
i_flat = np.array([0, 3, 1, 5])
i_square = np.array([[0, 3],
                    [1, 5]])
```

```
threes[i_flat]
```

```
array([ 3, 12,  6, 18])
```

```
threes[i_square]
```

```
array([[ 3, 12],
       [ 6, 18]])
```

The latter result can also be accomplished by indexing using a flat (i.e., one-dimensional) array followed by reshaping it to the desired dimensions as demonstrated below.

```
i = np.array([0, 3, 1, 5])
threes[i].reshape((2, 2))
```

```
array([[ 3, 12],  
       [ 6, 18]])
```

4.3.4 Masking

Elements in a NumPy array can also be selected using a boolean array through a process known as *masking*. The masking array is a boolean array filled with either `1` and `0` or `True` and `False` and has the same dimensions as the original array. Any element in the original array that has a `1` or `True` in the corresponding position of the masking array is returned.

For example,

```
orig_array = np.array([[5, 7, 1],  
                      [3, 4, 2],  
                      [0, 9, 8]])  
  
mask = np.array([[0, 1, 0],  
                 [1, 1, 1],  
                 [1, 0, 1]], dtype=bool)
```

```
orig_array[mask]
```

```
array([7, 3, 4, 2, 0, 8])
```

It's important to note that if you use `1` and `0` in the masking array, it is required that you include `dtype=bool` or else NumPy will treat the `1` and `0` as indices instead of booleans and attempt indexing.

```
mask = np.array([[0, 1, 0],  
                 [1, 1, 1],  
                 [1, 0, 1]])  
  
orig_array[mask]
```

```
array([[5, 7, 1],  
       [3, 4, 2],  
       [5, 7, 1]],  
  
      [[3, 4, 2],  
       [3, 4, 2],  
       [3, 4, 2]],  
  
      [[3, 4, 2],  
       [5, 7, 1],  
       [3, 4, 2]])
```

The true power of masking is when the masking array is generated through boolean logic such as `>`, `<=`, or `==`. This enables the user to select elements of an array through conditions as demonstrated below where we select all elements of the `orig_array` that are greater than 5.

```
cond = orig_array > 5  
cond
```

```
array([[False,  True, False],  
       [False, False, False],  
       [False,  True,  True]])
```

```
orig_array[cond]
```

```
array([7, 9, 8])
```

We can also include the condition directly in the square brackets to save a step as shown below.

```
orig_array[orig_array > 5]
```

```
array([7, 9, 8])
```

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's take the square root of numbers using NumPy's `np.sqrt()` function. The square root is taken of each elements automatically.

```
squares = np.array([1, 4, 9, 16, 25])  
np.sqrt(squares)
```

```
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 without loops.

```
import math  
math.sqrt(squares)
```

```
-----  
TypeError                                 Traceback (most recent call last)  
Cell In[51], line 2  
      1 import math  
----> 2 math.sqrt(squares)
```

```
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 as the starting array. Below, an array is multiplied by the scalar 3 which results in every element in the array being multiplied by this value.

$$3 \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 15 & 18 \\ 21 & 24 \end{bmatrix}$$

```
3 * np.array([[5, 6], [7, 8]])
```

```
array([[15, 18],
       [21, 24]])
```

The same outcome arises when performing a similar operation between a 1×1 array and a larger array.

$$[2] + [10 \ 20] = [12 \ 22]$$

```
np.array([2]) + np.array([10, 20])
```

```
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 2×2 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.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
a + b
```

```
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 one of 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 clones the array with fewer dimensions out so that it has the same dimensions as the other array. It should be noted that NumPy does not really clone out the array in the background; its behavior acts as if it does. It is a convenient way of thinking about the behavior and results. For example, below is the addition between a 2×2 and a 1×2 array.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + [2 \quad 5] = ?$$

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.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 2 & 5 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 3 & 7 \\ 5 & 9 \end{bmatrix}$$

```
a = np.array([[1, 2], [3, 4]])
b = np.array([2, 5])
a + b
```

```
array([[3, 7],
       [5, 9]])
```

What happens if a mathematical operation is performed between an array of higher dimensions with a scalar or a 1×1 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 cloned until the two arrays are the same size.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

```
a = np.array([[1, 2], [3, 4]])
b = np.array([2])
a * b
```

```
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.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} = ?$$

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[1, 1, 1],
              [2, 2, 2],
              [3, 3, 3]])
a + b
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[57], line 5
      1 a = np.array([[1, 2], [3, 4]])
      2 b = np.array([[1, 1, 1],
      3             [2, 2, 2],
      4             [3, 3, 3]])
----> 5 a + b

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 (`conc`).

```
def rate(k, conc):
    return k * conc
```

```
rate(1.2, 0.80)
```

```
0.96
```

What happens if we attempt the above calculation using a list of concentration values?

```
concs = [0.1, 0.5, 1.0, 1.5, 2.0]
```

```
rate(1.2, concs)
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[61], line 1
----> 1 rate(1.2, concs)

Cell In[58], line 2, in rate(k, conc)
      1 def rate(k, conc):
----> 2     return k * conc

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.

```
vrate = np.vectorize(rate)
```

```
vrate(1.2, concs)
```

```
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 fed into the `np.sqrt()` function as well as an array of values.

```
np.sqrt(4)
```

```
np.float64(2.0)
```

```
np.sqrt(np.array([1, 4, 9]))
```

```
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; many 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 1 Common Methods for Generating Arrays

Method	Description
<code>np.array()</code>	Generates an array from another object
<code>np.arange()</code>	Creates an array from [start, stop) with a given step size
<code>np.linspace()</code>	Creates an array from [start, stop] with given number of steps
<code>np.empty()</code>	Creates an "empty" array (actually filled with garbage)
<code>np.zeros()</code>	Generates an array of a given dimensions filled with zeros
<code>np.ones()</code>	Generates an array of a given dimensions filled with ones
<code>np.fromfunction()</code>	Generates an array using a Python function
<code>np.genfromtxt()</code>	Loads text file data into an array
<code>np.loadtxt()</code>	Load text file data into an array (cannot handle missing data)

Table 2 Array Attribute Methods

Method	Description
<code>np.shape(array)</code>	Returns the dimensions of an array
<code>np.ndim(array)</code>	Returns the number of dimensions (e.g., a 2D array is 2)
<code>np.size(array)</code>	Returns the number of elements in an array

Table 3 Array Modification Methods

Method	Description
<code>np.flatten()</code>	Flattens an array in place
<code>np.ravel()</code>	Returns a flattened view 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 view 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>
<code>np.roll()</code>	Rolls the array along the given axis; elements that fall off one end of the array appear at the other
<code>np.floor()</code>	Returns the floor (i.e., rounds down) of all elements in an array
<code>np.round(x, decimals=y)</code>	Rounds every number in an array <code>x</code> to <code>y</code> decimal places by Banker's rounding

Table 4 Array Measurement and Statistics 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.argmax()</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 local max
<code>np.fmin()</code>	Returns the element-by-element min between two arrays of the same size
<code>np.fmax()</code>	Returns the element-by-element max between two arrays of the same size
<code>np.percentile()</code>	Returns the specified percentile
<code>np.mean()</code>	Returns the mean (commonly known as the average)
<code>np.median()</code>	Returns the median
<code>np.std()</code>	Returns the standard deviation; be sure to include <code>ddof=1</code>
<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.unique()</code>	Returns an array of unique elements in an array, set <code>return_counts=True</code> to get frequency
<code>np.unique_counts()</code>	Returns an array of unique elements in an array and a second array with the frequency of these elements

4.6 Missing Data

Real datasets frequently contain gaps or missing values, so it is important to be able to deal with missing data. When importing data into NumPy, there are two commonly employed functions, `np.genfromtxt()` and `np.loadtxt()`. Though these are largely analogous functions in terms of capabilities, there is a key difference in that `np.genfromtxt()` can handle missing data while `np.loadtxt()` cannot. This means if your dataset may contain gaps, you should use `np.genfromtxt()`.

In the event the data file contains a gap, the `np.genfromtxt()` function will place a `nan` in that location by default. The `nan` stands for "not a number" and is simply a placeholder. For example, the file `dHf_ROH.csv` contains the number of carbons in linear alcohols and the gas-phase heat of formation in kJ/mol of each alcohol. The value for 1-undecanol (eleven carbons) is missing, so `np.genfromtxt()` places a `nan` in its place.

```
np.genfromtxt('data/dHf_ROH.csv', delimiter=',')
```

```
array([[ 1., -205.],
       [ 2., -234.],
       [ 3., -256.],
       [ 4., -277.],
       [ 5., -298.],
       [ 6., -316.],
       [ 7., -340.],
       [ 8., -356.],
       [ 9., -377.],
       [ 10., -395.],
       [ 11.,   nan],
       [ 12., -437.]])
```

Some data files use placeholder values instead of no value at all. These placeholders are often `-1`, `0`, `999`, or some physically meaningless or improbable value. If you have an alternative values you want in the missing data location, you can specify this using the `filling_values=` argument. As an example below, the missing value is replaced with a `999`.

```
np.genfromtxt('data/dHf_ROH.csv', delimiter=',', filling_values=999)
```

```
array([[ 1., -205.],
       [ 2., -234.],
       [ 3., -256.],
       [ 4., -277.],
       [ 5., -298.],
       [ 6., -316.],
       [ 7., -340.],
       [ 8., -356.],
       [ 9., -377.],
       [ 10., -395.],
       [ 11.,  999.],
       [ 12., -437.]])
```

In the event you have data with missing values, the `nan` placeholders can pose an issue when running statistics on the data. Below, we use the `np.mean()` method to try to calculate the mean enthalpy of formation but get a `nan` instead because the `np.mean()` function cannot handle the placeholder.

```
dHf = np.genfromtxt('data/dHf_ROH.csv', delimiter=',')
```

```
np.mean(dHf[:,1])
```

```
np.float64(nan)
```

Alternatively, NumPy has a number of versions of functions (Table 5) that are specifically designed to handle data with missing values.

Table 5 Statistics Methods Dealing with NaNs

Function	Description
<code>np.nanstd()</code>	Standard deviation
<code>np.nanmean()</code>	Mean
<code>np.nanvar()</code>	Variance
<code>np.nanmedian()</code>	Median
<code>np.nanpercentile()</code>	Qth percentile
<code>np.nanquantile()</code>	Qth quantile

```
np.nanmean(dHf[:, 1])
```

```
np.float64(-317.3636363636364)
```

```
np.nanquantile(dHf[:, 1], 0.6)
```

```
np.float64(-298.0)
```

4.7 Random Number Generation

Stochastic simulations, addressed in [chapter 9](#), are a common tool in the sciences and rely on a series of random numbers, so 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 distribution where all values are equally likely or biased a distribution where some values are more probable than others. Below are random number functions from the NumPy `random` module useful in generating random number distributions to suit the needs of your simulations.

4.7.1 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 as 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 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.

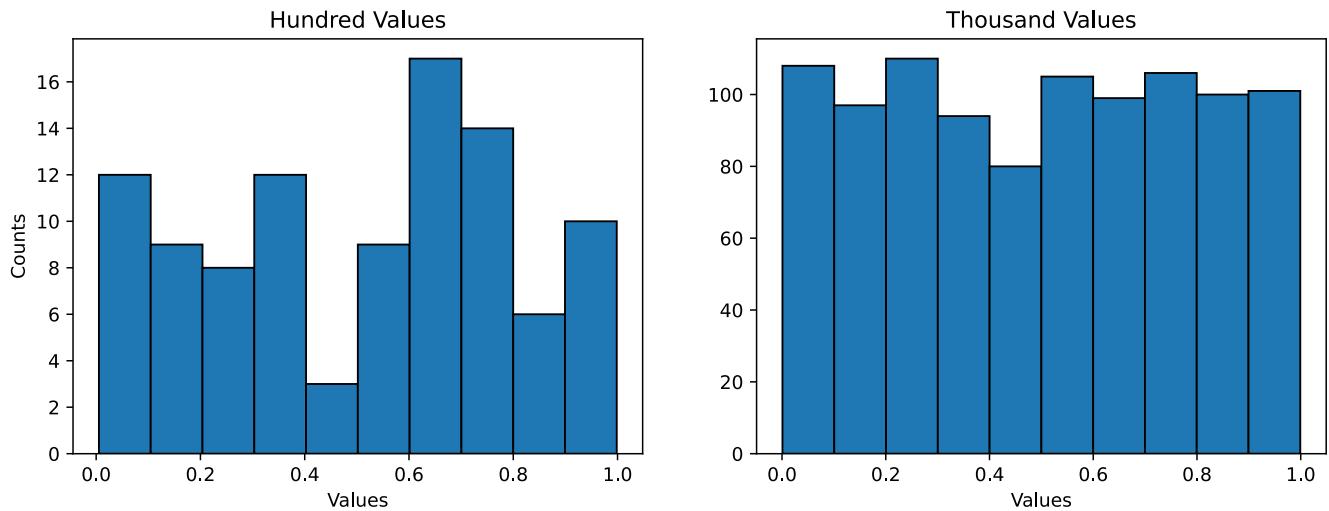


Figure 3 Histograms of a hundred (left) and a hundred thousand (right) randomly generated floats from the [0,1) range in an even distribution using the `random()` method.

Starting in version 1.18, NumPy's preferred method for producing random values is through a Generator called using the `rng = np.random.default_rng()` function. Once a generator has been created, it can be used to generate the necessary random values. NumPy has multiple methods available for generating evenly-distributed random numbers including the following two functions where `n` is the number of random values to be generated. The `rng.random(n)` function generates `n` random floats from the range [0,1). The `rng.integers(low, high=x, size=n)` function generates random integers in the range [low, high) and can generate multiple values using the size argument.

```
rng = np.random.default_rng()

rng.random(n)

rng.integers(low, high=x, size=n)
```

⚠️ Warning

Prior to version 1.18 of NumPy, random numbers were generated using function calls that look like `np.random.randint()`. While these should still work, they are considered legacy, so it is uncertain how long they will continue to be supported.

```
rng = np.random.default_rng()
```

```
rng.random(5)
```

```
array([0.32595272, 0.10690741, 0.05035009, 0.03178274, 0.38049995])
```

```
rng.integers(0, high=10, size=10)
```

```
array([3, 6, 5, 4, 5, 3, 2, 1, 4, 3])
```

4.7.2 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 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 `rng.binomial()` function call.

```
rng = np.random.default_rng()  
rng.binomial(t, p, size=n)
```

The `t` argument is the number of trials while the `size=` argument is the number of generated 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) are of a hundred randomly generated numbers in a binomial distribution with `p = 0.5` and where `t = 1` (left), `t = 2` (center), and `t = 10` (right).

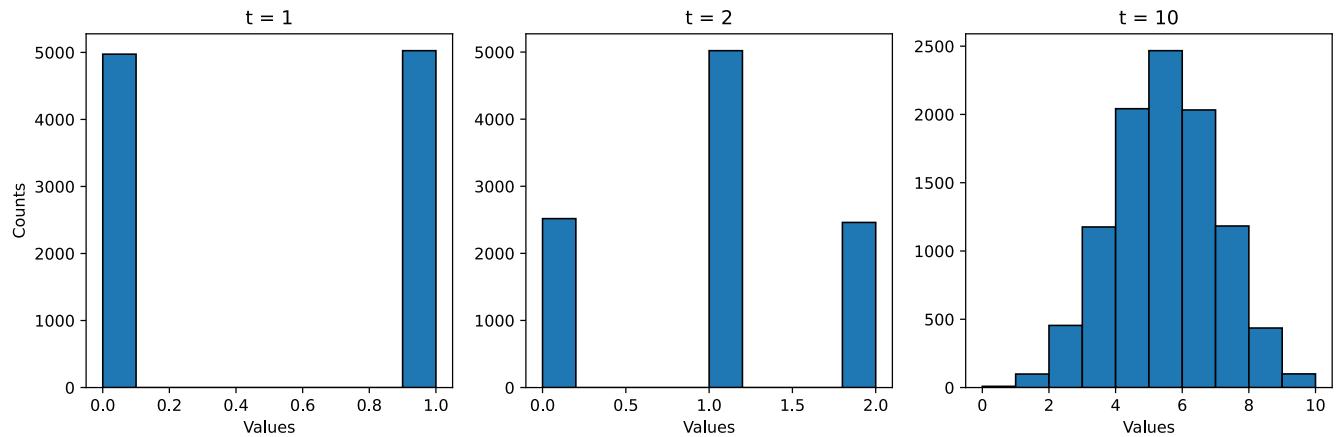


Figure 4 Histograms of a hundred randomly generated numbers in a binomial distribution with `p = 0.5` and `t = 1` (left), `t = 2` (center), and `t = 10` (right).

4.7.3 Poisson Distribution

A *Poisson distribution* is a probability distribution of how likely it is for independent events to occur in a given interval (time or space) with a known average frequency (λ). 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 `rng.poisson()` function, which accepts two arguments.

```
rng.poisson(lam=1.0, size=n)
```

The first argument, λ (`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.

```
rng.poisson(lam=3.6, size=30)
```

```
array([2, 2, 4, 2, 3, 2, 4, 3, 2, 6, 3, 3, 4, 3, 4, 4, 3, 0, 5, 5, 4, 4,
     8, 2, 2, 3, 7, 4, 4, 5])
```

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 5, left). When this simulation was repeated with thirty thousand samples (Figure 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.

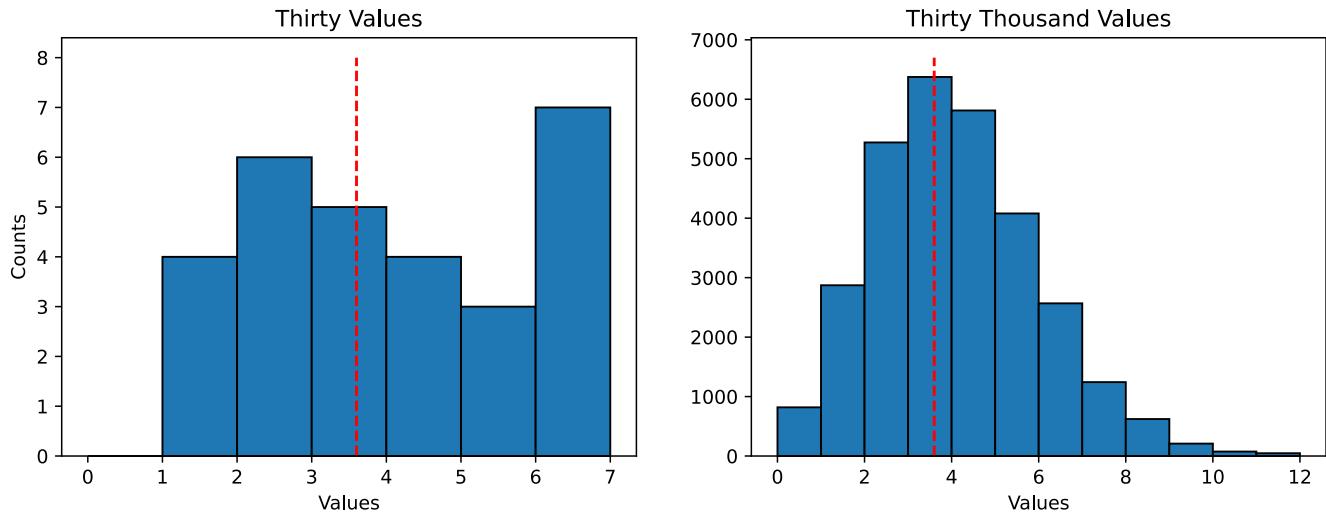


Figure 5 Histograms of thirty (left) and thirty thousand (right) randomly generated integers in a Poisson distribution with a target mean (λ) of 3.6 (dashed red line).

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 * (rng.random(10) - 0.5)
rand_float
```

```
array([-0.40626378, -0.95185875, -0.13479041,  0.9508908 ,  0.21275482,
       0.50651172,  0.13768064,  0.98436196, -0.47759103, -0.2360542 ])
```

4.7.4 Other Functions

The random module in NumPy also includes a large variety of other random number and sequence generators. This includes `rng.normal()` which generates values centered around zero in a normal distribution. The `rng.choice()` function selects a random value from a provided array of values, while the `rng.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 NumPy random functions are in Table 6.

Table 6 Summary of Common NumPy `np.random` Functions

Function	Description
<code>rng.random()</code>	Generates random floats in the range [0,1) in an even distribution
<code>rng.integers()</code>	Generates random integers from a given range in an even distribution
<code>rng.normal()</code>	Generates random floats in a normal distribution centered around zero
<code>rng.binomial()</code>	Generates random integers in a binomial distribution; takes a probability , <code>p</code> , and <code>size</code> arguments
<code>rng.poisson()</code>	Generates random floats in a Poisson distribution; takes a target mean argument (<code>lam</code>)
<code>rng.choice()</code>	Selects random values taken from a 1-D array or range
<code>rng.shuffle()</code>	Randomizes the order of an array

```
rng.random(1)
```

```
array([0.28404729])
```

```
rng.integers(0, high=100)
```

```
np.int64(29)
```

```
rng.normal(loc=0.0, scale=1.0, size=3)
```

```
array([ 1.23226045,  0.6006648 , -1.05247125])
```

```
rng.binomial(2, p=0.5, size=3)
```

```
array([1, 1, 2])
```

```
rng.poisson(lam=2.0, size=5)
```

```
array([2, 2, 1, 5, 5])
```

```
rng.choice(20, size=3)
```

```
array([ 5, 19,  2])
```

```
arr = np.array([0, 1, 2, 3, 4])
rng.shuffle(arr)
arr
```

```
array([0, 1, 2, 3, 4])
```

Further Reading

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/> (free resource)
2. NumPy User Guide. <https://numpy.org/doc/stable/user/index.html#user> (free resource)
3. VanderPlas, J. Python data Science Handbook: Essential Tools for Working with Data, 1st ed.; O'Reilly: Sebastopol, CA, 2017, chapter 2. Freely available from the author at <https://jakevdp.github.io/PythonDataScienceHandbook/> (free resource)

Exercises

Complete the following exercises in a Jupyter notebook using NumPy and NumPy arrays. Avoid using `for` loops whenever possible. Any data file(s) referred to in the problems can be found in the `data` folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

1. Generate an array containing the atomic numbers for the first 26 elements.
2. The following equation defines the relationship between energy (J) of a photo and its wavelength (m) where h is Plank's constant ($6.626 \times 10^{-34} J \cdot s$) 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 ($4.00 \times 10^{-7} m \rightarrow 8.00 \times 10^{-7} m$) 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 101.325 a hundred times.
4. The following array contains temperatures in Fahrenheit. Convert these values to $^{\circ}\text{C}$ without using a for loop.

```
F = array([0, 32, 100, 212, 451])
```

5. Generate two arrays containing the following sine functions from $x = 0 \rightarrow 10\pi$

$$y = \sin(x)$$

$$y = \sin(1.1x + 0.5)$$

- a. Plot these two sine waves on the same plot.
- b. Add the two sine functions together and plot the result.

c. Explain why the signal in part b is smaller in one area and larger in another. 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 ΔG° and K (equilibrium constant) is shown below. Plot ΔG° 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.

$$\Delta G = -RT\ln(k)$$

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 $A = 1$. Watch your energy units carefully.

$$k = Ae^{-E_a/RT}$$

8. Generate an array containing integers $0 \rightarrow 14$ (inclusive).

a. Reshape the array to be a 3×5 array.

b. Transpose the array, so now it should be a 5×3 array.

c. Make the array one-dimensional again *without* using the `reshape()` method.

9. Generating an 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.i., $1 \rightarrow 8$).

b. Generate a second array containing the energy (J) of each orbit in part A for a Bohr model of a hydrogen atom using the equation below.

$$E = -2.18 \times 10^{-18} J \frac{1}{n^2}$$

c. Combine the two arrays from parts A and B into a new 8×2 array with the first column containing the principle quantum numbers and the second containing the energies.

10. Generate a one-dimensional array with the following code and index the 5th 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.

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.

$$\begin{bmatrix} 1 & 8 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = ?$$

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.
16. Generate a random array of 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 a random array of integers from $0 \rightarrow 35$ (inclusive) and then sort it.
18. Hydrogen nuclei can have a spin of $+1/2$ and $-1/2$ and occur in approximately a 1:1 ratio. Simulate the number of $+1/2$ hydrogen nuclei in a molecule of six hydrogen atoms and plot the distribution. Hint: being that there are two possible outcomes, this can be simulated using a *binomial distribution*. See [section 4.7.2](#).
19. Using NumPy's `random` module, generate a random DNA sequence (i.e., series of 'A', 'T', 'C', and 'G' bases) 40 bases long stored in an array.

Chapter 5: Pandas

Contents

- 5.1 Basic Pandas Objects
- 5.2 Reading/Writing Data
- 5.3 Examining Data with Pandas
- 5.4 Modifying DataFrames
- Further Reading
- Exercises

While NumPy is the foundation of much of the SciPy ecosystem and provides very capable ndarray objects, it has a few missing features. 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.

```
nums = [1, 2, 3, 'four', 5, 'six', 7.0]
```

```
import numpy as np
np.array(nums)
```

```
array(['1', '2', '3', 'four', '5', 'six', '7.0'], dtype='|U32')
```

The second shortcoming is that NumPy arrays do not have strong support for labels in the data. That is, you might want to label rows and columns describing what they represent like you might see in a well constructed spreadsheet. While there is some support for this in NumPy, it is not as strong as pandas' support. 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 value of a particular column or the ability to merge two datasets with automatic alignment of related data.

To fill in these missing features, the pandas library provides a wealth of additional tools on top of NumPy for working with data, and possibly the most endearing feature, the ability to call data based on labels. That is, 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, the pandas library is imported with the `pd` alias, which is used here. This chapter assumes the following imports.

```
import pandas as pd
import matplotlib.pyplot as plt
```

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 a Series and DataFrame is that a Series is one-dimensional while a DataFrame is two-dimensional. 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.

```
mass = pd.Series([1.01, 4.00, 6.94, 9.01, 10.81])  
mass
```

```
0    1.01  
1    4.00  
2    6.94  
3    9.01  
4   10.81  
dtype: float64
```

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). Think of the numbers as the row labels you would see in a traditional spreadsheet software application.

Consistent with lists, tuples, and ndarrays, values in a Series can be accessed using indexing with square brackets as demonstrated below.

```
mass[2]
```

```
np.float64(6.94)
```

Unlike most 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 row labels can be assigned using the `index=index` argument shown below.

```
index=['H', 'He', 'Li', 'Be', 'B']  
mass2 = pd.Series([1.01, 4.00, 6.94, 9.01, 10.81], index=index)  
mass2
```

```
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](#)).

```
mass2['He']
```

```
np.float64(4.0)
```

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.

```
mass2.index
```

```
Index(['H', 'He', 'Li', 'Be', 'B'], dtype='object')
```

```
mass.index =['H', 'He', 'Li', 'Be', 'B']  
mass
```

```
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 custom index values.

```
mass2.iloc[2]
```

```
np.float64(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 with two capital letters. 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.

```
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]
```

```
columns = ['H', 'He', 'Li', 'Be', 'B']
index = ['name', 'AN', 'mass', 'IE']
elements = pd.DataFrame([name, AN, mass, IE],
                        columns=columns, index=index)
elements
```

	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 access data in a DataFrame, place the column name in square brackets.

```
elements['Li']
```

```
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](#).

```
elements.loc['IE']
```

```
H     13.6
He    24.6
Li     5.4
Be     9.3
B      8.3
Name: IE, dtype: object
```

```
atomic_number = elements.loc['AN']
```

```
atomic_number['B']
```

5

Alternatively, we can use the DataFrame directly and index it with the `loc[]` method as `[row, column]`.

```
elements.loc['IE', 'Li']
```

Numerical index values can also be used with the `iloc[]` method. This reduces indexing to how NumPy arrays are indexed.

```
elements.iloc[2:, 2]
```

```
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 1.

Table 1 Summary of Pandas Indexing

Index Method	Description
<code>s[index]</code>	Index Series with assigned index values
<code>s.iloc[index]</code>	Index Series with default numerical index values
<code>df[column]</code>	Index DataFrame with column name
<code>df.loc[row]</code>	Index DataFrame with row name
<code>df.loc[row, column]</code>	Index DataFrame with row and column names
<code>df.iloc[row, column]</code>	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](#).

Table 2 Import/Export Functions in Pandas

Function	Description
<code>read_csv()</code> and <code>to_csv()</code>	Imports/Exports data from/to a CSV file
<code>read_table()</code> and <code>to_table()</code>	General-purpose importer/exporter
<code>read_hdf5()</code> and <code>to_hdf5()</code>	Imports/Exports data from/to an HDF5 file
<code>read_clipboard()</code> and <code>to_clipboard()</code>	Transfers data to/from the clipboard to a Series or DataFrame
<code>read_excel()</code> and <code>to_excel()</code>	Reads/writes an Excel file

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). To set a space as a delimiter, use `sep=\s+`. The function also includes a series of other arguments listed below in Table 3.

Table 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 at the top of the 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 None if no header is provided in the file
<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 position in a line](#), but being that all the data columns here have spaces between them, we can use space delimitation by setting `sep=\s+`. 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.

```

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

```

```

benz = pd.read_table('data/benzene.pdb', sep=r'\s+',
                     skiprows=2, skipfooter=13, header=None,
                     engine='python')
benz

```

		0	1	2	3	4	5	6	7
	0	HETATM	1	H	UNK	1	0.0	0.000	-0.020
	1	HETATM	2	C	UNK	1	0.0	0.000	1.067
	2	HETATM	3	C	UNK	1	0.0	0.000	3.857
	3	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.

```
elements.to_csv('elements.csv')
```

If we check the directory containing the Jupyter notebook, the data folder contains 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.

```
pd.read_csv('data/elements.csv')
```

	Unnamed: 0	H	He	Li	Be	B
0	name	hydrogen	helium	lithium	beryllium	boron
1	AN	1	2	3	4	5
2	mass	1.01	4.0	6.94	9.01	10.81
3	IE	13.6	24.6	5.4	9.3	8.3

5.2.3 Excel Notebook Files

Pandas provides another useful function that imports Excel notebook files (i.e., `.xls` or `.xlsx`). Excel files are a specialized file type that requires the support of additional libraries, known as dependancies, that pandas does not install by default. A list of these dependancies is provided on the [pandas website](#). You can either install each dependency yourself, or pandas provides a shortcut (for pandas version 2.0.0 and later) of `pip install "pandas[excel]"` that is run in the Terminal window (see [section 0.2](#) for Terminal instructions). However, please check the [pandas website](#) for the full and most current instructions as things may have changed. 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.

```
pd.read_excel('data/test.xlsx')
```

	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 loads the first sheet in the file, which is the default behavior. 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 `sheet_name` argument also accepts integer index values (i.e., `0` for the first sheet and so on).

```
data = pd.read_excel('data/test.xlsx', sheet_name='Sheet2')
data
```

	a	b	Unnamed: 2	Unnamed: 3	Unnamed: 4	Unnamed: 5	Unnamed: 6	b.1
0	1	0.841471		NaN	NaN	NaN	NaN	NaN
1	2	0.909297		NaN	NaN	NaN	NaN	NaN
2	3	0.141120		NaN	NaN	NaN	NaN	NaN
3	4	-0.756802		NaN	NaN	NaN	NaN	NaN
4	5	-0.958924		NaN	NaN	NaN	NaN	NaN
5	6	-0.279415		NaN	NaN	NaN	NaN	NaN
6	7	0.656987		NaN	NaN	NaN	NaN	NaN
7	8	0.989358		NaN	NaN	NaN	NaN	NaN
8	9	0.412118		NaN	NaN	NaN	NaN	NaN

Alternatively, if you want to extract the sheet names, you can use the `sheets_names` method with the `ExcelFile` class as demonstrated below.

```
xl = pd.ExcelFile('data/test.xlsx')
xl.sheet_names
```

```
['Sheet1', 'Sheet2']
```

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.

```
data.to_excel('new_file.xlsx', sheet_name='First Sheet')
```

```
with pd.ExcelWriter('new_file.xlsx') as writer:  
    data.to_excel(writer, sheet_name='First Sheet')  
    data.to_excel(writer, sheet_name='Copy of First Sheet')
```

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. This is typically located under the Edit menu of most software applications. Alternatively, you can type Command + C on a macOS or Control + C on Windows and Linux. Finally, use the `pd.read_clipboard()` function to convert it to a pandas DataFrame.

```
pd.read_clipboard()
```

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.

```
rng = np.random.default_rng()
```

```
random = pd.DataFrame({'even': rng.random(1000),  
                      'normal': rng.normal(size=1000),  
                      'poisson': rng.poisson(lam=3.0, size=1000)})
```

```
random.head()
```

	even	normal	poisson
0	0.399306	0.366845	3
1	0.717415	0.240555	3
2	0.280823	-1.014316	4
3	0.082725	-0.249080	3
4	0.969771	0.414425	3

```
random.tail()
```

	even	normal	poisson
995	0.068204	-0.628008	3
996	0.618927	0.232807	1
997	0.103016	0.182341	5
998	0.483815	1.177764	4
999	0.750867	-0.837742	2

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.

```
random.describe()
```

	even	normal	poisson
count	1000.000000	1000.000000	1000.000000
mean	0.495366	0.009895	2.965000
std	0.293186	1.000833	1.688972
min	0.000812	-3.324801	0.000000
25%	0.230925	-0.657265	2.000000
50%	0.505785	0.003473	3.000000
75%	0.742341	0.651874	4.000000
max	0.999107	3.143804	10.000000

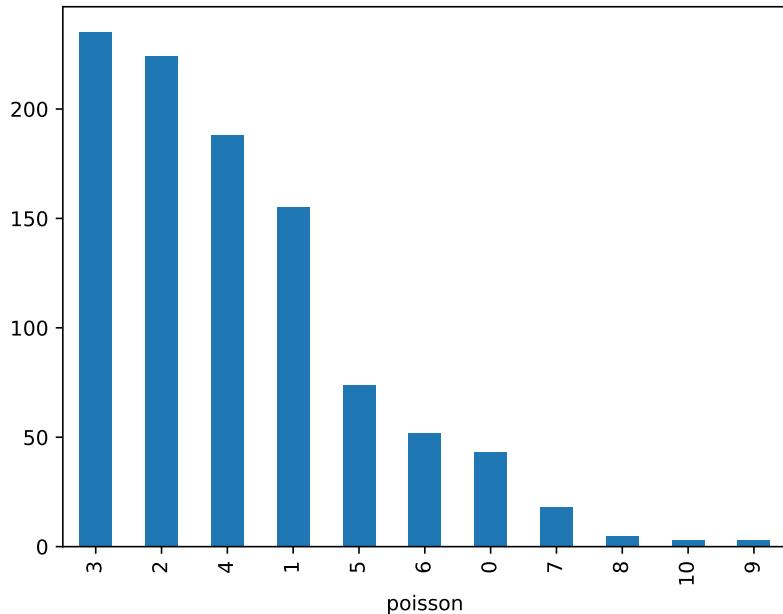
Another useful function is the `value_counts()` method which returns all unique values in a Series (or DataFrame column or row). Below, it is demonstrated on the poisson column being that the other two columns will have a relatively large number of unique values.

```
counts = random['poisson'].value_counts()  
counts
```

```
poisson  
3    235  
2    224  
4    188  
1    155  
5     74  
6     52  
0     43  
7     18  
8      5  
10     3  
9      3  
Name: count, 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.

```
counts.plot(kind='bar');
```



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 choose from such as those listed below.

Table 4 Broadcasted Pandas Methods

Function	Description
<code>abs()</code>	Absolute value
<code>count()</code>	Counts items
<code>cumsum()</code>	Cumulative sum
<code>cumprod()</code>	Cumulative product
<code>mad()</code>	Mean absolute deviation
<code>max()</code>	Maximum
<code>min()</code>	Minimum
<code>mean()</code>	Mean
<code>median()</code>	Median
<code>mode()</code>	Mode
<code>std()</code>	Standard deviation

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.

`elements`

	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

```
elements['C'] = ['carbon', 6, 12.01, 11.3]
elements
```

	H	He	Li	Be	B	C
name	hydrogen	helium	lithium	beryllium	boron	carbon
AN	1	2	3	4	5	6
mass	1.01	4.0	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.

```
nitrogen = pd.Series([7, 14.01, 'nitrogen', 14.5],
                     index=['AN', 'mass', 'name', 'IE'])
nitrogen
```

```
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.

```
elements['N'] = nitrogen
elements
```

	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	4.0	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.

```
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	4.0	6.94	9.01	10.81	12.01	14.01
IE	24.6	5.4	9.3	8.3	11.3	14.5

```
elements.drop('IE', axis=0)
```

	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	4.0	6.94	9.01	10.81	12.01	14.01

In the second example above, the hydrogen is back despite being previously dropped. This is because the `drop()` method does not by default modify the original DataFrame. To make the changes permanent, either assign the new DataFrame to a new variable or add the `inplace=True` keyword argument to the above `drop()` function.

There is a similar function `pd.dropna()` that drops columns or rows from a DataFrame that contain `nan` values. This is commonly used to remove incomplete data from a dataset. The `pd.dropna()` function behaves very similarly to the `pd.drop()` function including the `inplace=` and `axis=` arguments.

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.

```
chemdata1 = [['MW', 58.08, 32.04], ['dipole', 2.91, 1.69],  
             ['formula', 'C3H6O', 'CH3OH']]  
columns=['property', 'acetone', 'methanol']  
chmdf1 = pd.DataFrame(chemdata1, columns=columns)
```

```
chmdf1
```

	property	acetone	methanol
0	MW	58.08	32.04
1	dipole	2.91	1.69
2	formula	C3H6O	CH3OH

```
chmdata2 = [['formula', 'C6H6', 'H2O'], ['dipole', 0.0, 1.85],
            ['MW', 78.11, 18.02]]
chmfp2 = pd.DataFrame(chmdata2, columns=['property', 'benzene', 'water'])
```

chmfp2

	property	benzene	water
0	formula	C6H6	H2O
1	dipole	0.0	1.85
2	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.

```
chmfp1.merge(chmfp2)
```

	property	acetone	methanol	benzene	water
0	MW	58.08	32.04	78.11	18.02
1	dipole	2.91	1.69	0.0	1.85
2	formula	C3H6O	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.

```
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 generated 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.

```
comps1.merge(comps2, left_on='element', right_on='metal')
```

	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](#).

```
comps3 = comps1.merge(comps2, left_on='element',
                      right_on='metal')
comps3.drop('metal', axis=1, inplace=True)
comps3
```

	element	protons	IE
0	Co	27	7.88
1	Fe	26	7.90
2	Cr	24	6.79
3	Ni	28	7.64

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.

```
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]})
```

```
group1
```

	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.

```
pd.concat((group1, group2))
```

	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.

```
pd.concat((group1, group2), axis=1)
```

	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 datasets.

```
pd.merge(group1, group2, on='metal')
```

	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/> (free resource)
2. VanderPlas, J. Python data Science Handbook: Essential Tools for Working with Data, 1st ed.; O'Reilly: Sebastopol, CA, 2017, chapter 3. Freely available from the author at <https://jakevdp.github.io/PythonDataScienceHandbook/> (free resource)
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 exercises in a Jupyter notebook using the pandas library. Avoid using `for` loops unless absolutely necessary. Any data file(s) referred to in the problems can be found in the `data` folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

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.
 - a) Set the wavelengths as the index values.
 - b) Plot the absorption versus wavelength.
 - c) 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.

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.
 - a) 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.
 - b) 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

- a) 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.
 - b) 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.
 - 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

Contents

- 6.1 Feature Detection
- 6.2 Smoothing Data
- 6.3 Fourier Transforms
- 6.4 Fitting & Interpolation
- 6.5 Baseline Correction
- Further Reading
- Exercises

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 in scientific data analysis. Table 1 summarizes some of the key modules in the SciPy library.

Table 1 Common SciPy Modules

Module	Description	Examples
<code>constants()</code>	Compilation of scientific constants	
<code>fft()</code>	Fourier transform functions	Section 6.4
<code>integrate()</code>	Integration for both functions and sampled data	Sections 8.4.3 and 8.4.4
<code>interpolate()</code>	Data interpolation	Section 6.4.4
<code>io()</code>	File importers and exporters	
<code>linalg()</code>	Linear algebra functions	Section 8.3.1
<code>optimize()</code>	Optimization algorithms	Chapter 14
<code>signal()</code>	Signal processing functions	Sections 6.1.2 , 6.1.3 , and 6.2.4

Unlike NumPy, many of the functions in SciPy are stored in modules, so each module from SciPy needs to be imported individually or listed when calling the function. It is common to see specific SciPy modules imported as shown below.

```
from scipy import module
```

Alternatively, you can import a single function from a module.

```
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.

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

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 inflection points. This section covers extracting feature information from signal data.

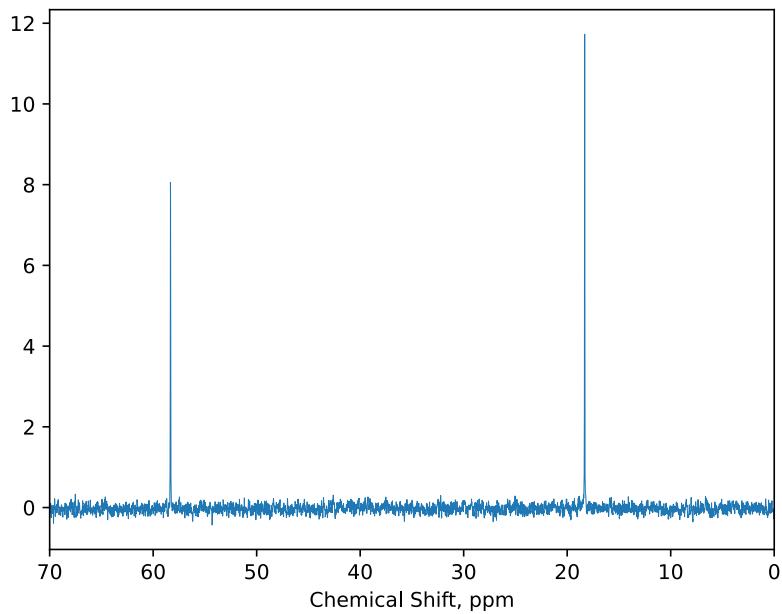
6.1.1 Global Maxima & Minima

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 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}C - ^1H Nuclear Magnetic Resonance (NMR) spectrum and an infrared (IR) spectrum. These data are imported below using NumPy.

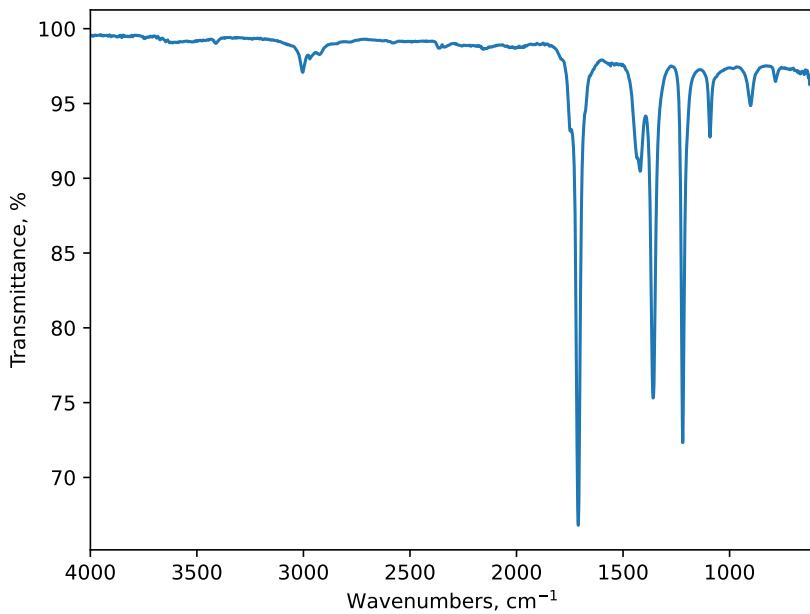
```
nmr = np.genfromtxt('data/13C_ethanol.csv', delimiter=',',
                     skip_footer=1, skip_header=1)
```

```
plt.plot(nmr[:,0], nmr[:,1], lw=0.5)
plt.xlabel('Chemical Shift, ppm')
plt.xlim(70, 0);
```



```
ir = np.genfromtxt('data/IR_acetone.csv', delimiter=',')
```

```
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.

```
np.max(nmr[:,1])
```

```
np.float64(11.7279863357544)
```

```
np.min(ir[:,1])
```

```
np.float64(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*).

```
imax = np.argmax(nmr[:,1])
imax
```

```
np.int64(5395)
```

```
imin = np.argmin(ir[:,1])
imin
```

```
np.int64(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 cm^{-1} in the IR spectrum.

```
nmr[imax, 0]
```

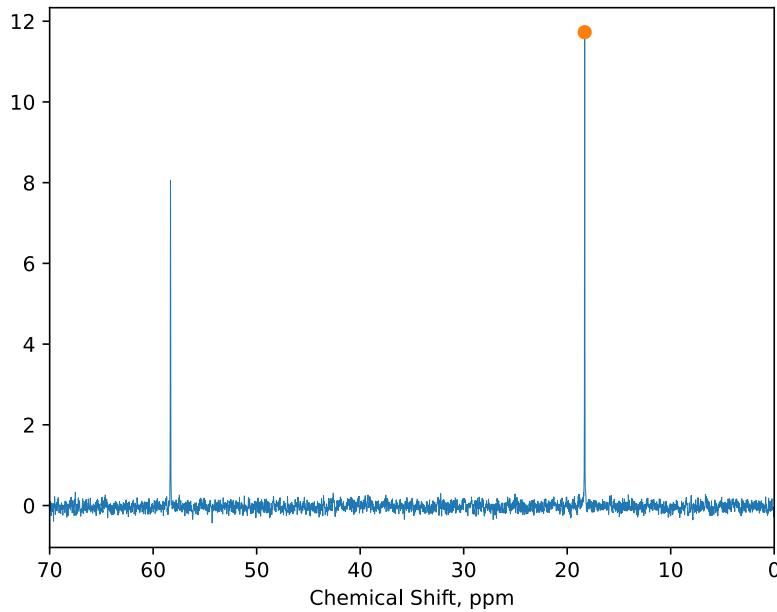
```
np.float64(18.312606267778)
```

```
ir[imin, 0]
```

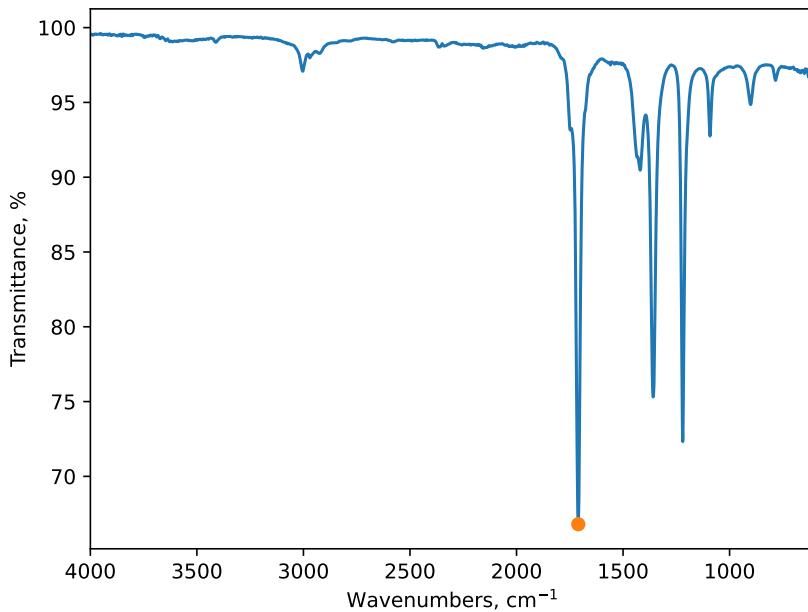
```
np.float64(1710.068)
```

Below, these values are plotted on the spectra as orange dots to validate that they are indeed the largest features in the spectra.

```
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);
```



```
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.

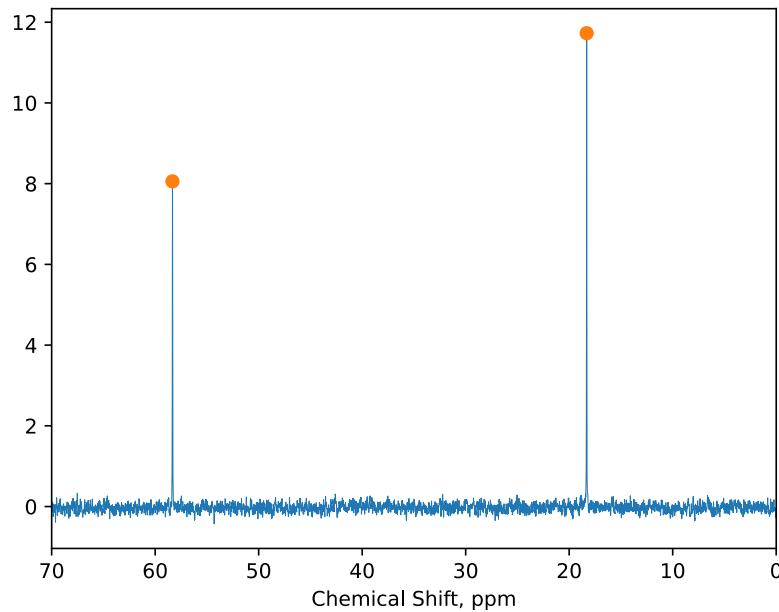
```
from scipy.signal import argrelextrema
```

```
imax = argrelextrema(nmr[:,1], order=2000)  
imax
```

```
(array([1219, 5395]),)
```

The `argrelextrema()` function returned two indices as an array wrapped in a tuple. If we plot the maxima marked with dots, we see that the function correctly identified both peaks.

```
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);
```



The `argrelextrema()` function may at times 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. Finally, the user can filter the returned values that correspond to peaks above a certain height value. Below is an example of filtering values based on a height. The window below is intentionally narrowed so that the `argrelextrema()` function returns too many values for demonstration purposes.

```
imax = argrelextrema(nmr[:,1], order=1000)[0]  
imax
```

```
array([1219, 2860, 3943, 5395, 6613])
```

Next, we will create a boolean mask (see [section 4.3.4](#)) which is a series of `True` and `False` values indicating if the data point is above a height value or not. In this example, we are using `1` as a height, but another height value may be more appropriate for different data. This is accomplished below by using the boolean `>` operator. The `nmr[imax, 1]` indexes the identified peaks from above and only returns the height values as a result of the `1`. If the `1` was not included, we would get a collection of [ppm, height] pairs.

```
mask = nmr[imax, 1] > 1  
mask
```

```
array([ True, False, False,  True, False])
```

Finally, we treat the mask of `True`/`False` values as if they are indices to get only the values for legitimate peaks.

```
imax[mask]
```

```
array([1219, 5395])
```

6.1.3 SciPy `find_peaks()` Function

The `scipy.signal` module includes a convenient `find_peaks()` function that facilitates the finding of multiple peaks in a spectrum based on parameters such as the height of the peaks or [prominence](#). This function requires a one-dimensional array as a positional argument and a number of optional, keyword arguments (Table 2).

Table 2 Select Keyword Argument for the `scipy.signal.find_peaks()` Function

Parameter	Description
<code>height=</code>	Verticle height of the peak apex.
<code>threshold=</code>	Verticle distance between a data point and the adjacent data points.
<code>distance=</code>	Horizontal distance between a peak and its nearest neighbor. If two peaks are near each other, the smaller one is discarded.
<code>priminence=</code>	Distance between a peak apex and the base of the peak.
<code>width=</code>	Peak width measured in number of data points.

Each of these parameters can take a single number treated as a minimum value. Alternatively, most of these parameters can also take two numbers in an array, list, or tuple in which case the first value is a minimum while the second value is a maximum.

```
find_peaks(data, height=min)  
find_peaks(data, height=(min, max))
```

This function only identifies positive peaks (i.e. pointing upwards). Our IR spectrum is currently represented as percent transmittance, so we can convert it to absorbance using the following equation.

$$Absorbance = 2 - \log(\% Transmittance)$$

```
absorb = 2 - np.log10(ir[:,1])
```

Now that the peaks are point upward, we feed the data into the `find_peaks()` function and decide how to best identify the peaks we are interested in. This will depend up the type of spectrum and other conditions. One straight forward method is the `height` parameter where any peaks above this level are identified at their apex.

```
from scipy.signal import find_peaks  
find_peaks(absorb)
```

```
(array([ 2,   16,   31,   46,   62,   84,  102,  116,  130,  140,  161,
       176,  199,  215,  220,  235,  249,  262,  270,  279,  291,  307,
       382,  442,  455,  471,  486,  497,  524,  625,  749,  791,  812,
       837,  864, 1020, 1114, 1285, 1573, 1698, 1731, 1858, 1879, 1908,
      1919, 1934, 1948, 1988, 2009, 2020, 2064, 2091, 2108, 2148, 2172,
      2184, 2302, 2381, 2468, 2539, 2565, 2579, 2607, 2628, 2661, 2673,
      2693, 2716, 2733, 2745, 2756, 2772, 2789, 2811, 2829, 2846, 2853,
      2865, 2886, 2895, 2909, 2927, 2950, 2958, 2971, 2985, 2996, 3008,
      3049, 3058, 3069, 3083, 3092, 3109, 3123, 3142, 3168, 3196, 3213,
      3227, 3240, 3257, 3277, 3284, 3303, 3315, 3334, 3348, 3363, 3380,
      3397, 3409, 3433, 3446, 3469, 3497, 3527, 3557, 3568, 3586, 3608,
      3651, 3662, 3711, 3731, 3758, 3774, 3794, 3805, 3826, 3838, 3852,
      3869, 3881, 3897, 3911, 3925, 3939, 3979, 3994, 4003, 4019, 4032,
      4038, 4048, 4095, 4111, 4153, 4181, 4191, 4206, 4222, 4241, 4251,
      4264, 4285, 4298, 4313, 4329, 4348, 4367, 4379, 4386, 4405, 4418,
      4437, 4454, 4472, 4488, 4523, 4534, 4549, 4568, 4590, 4603, 4618,
      4645, 4670, 4688, 4701, 4716, 4729, 4818, 4827, 4849, 4914, 4984,
      5095, 5127, 5148, 5178, 5193, 5212, 5233, 5247, 5272, 5294, 5302,
      5318, 5338, 5361, 5374, 5400, 5415, 5432, 5440, 5451, 5476, 5506,
      5526, 5540, 5554, 5567, 5578, 5598, 5618, 5636, 5656, 5666, 5673,
      5683, 5696, 5708, 5725, 5742, 5765, 5830, 5880, 5899, 5915, 5927,
      5934, 5946, 5965, 5991, 6003, 6021, 6043, 6051, 6060, 6072, 6093,
      6131, 6154, 6165, 6192, 6211, 6222, 6235, 6249, 6261, 6280, 6318,
      6368, 6375, 6404, 6432, 6451, 6462, 6478, 6502, 6519, 6532, 6547,
      6557, 6571, 6594, 6610, 6625, 6639, 6662, 6681, 6696, 6708, 6723,
      6746, 6763, 6780, 6794, 6812, 6827, 6850, 6879, 6903, 6919, 6934,
      6956, 6976, 6991, 7004, 7013, 7026]),
{}))
```

The function returns and tuple containing an array and a dictionary in this order. The array contains the indices of identified peaks while the dictionary may either be empty or include information about the identified peaks depending upon what keyword arguments are used in the function.

Below, we can plot the results of the function with the horizontal dotted line representing the chosen height and the orange dots represent identified peaks.

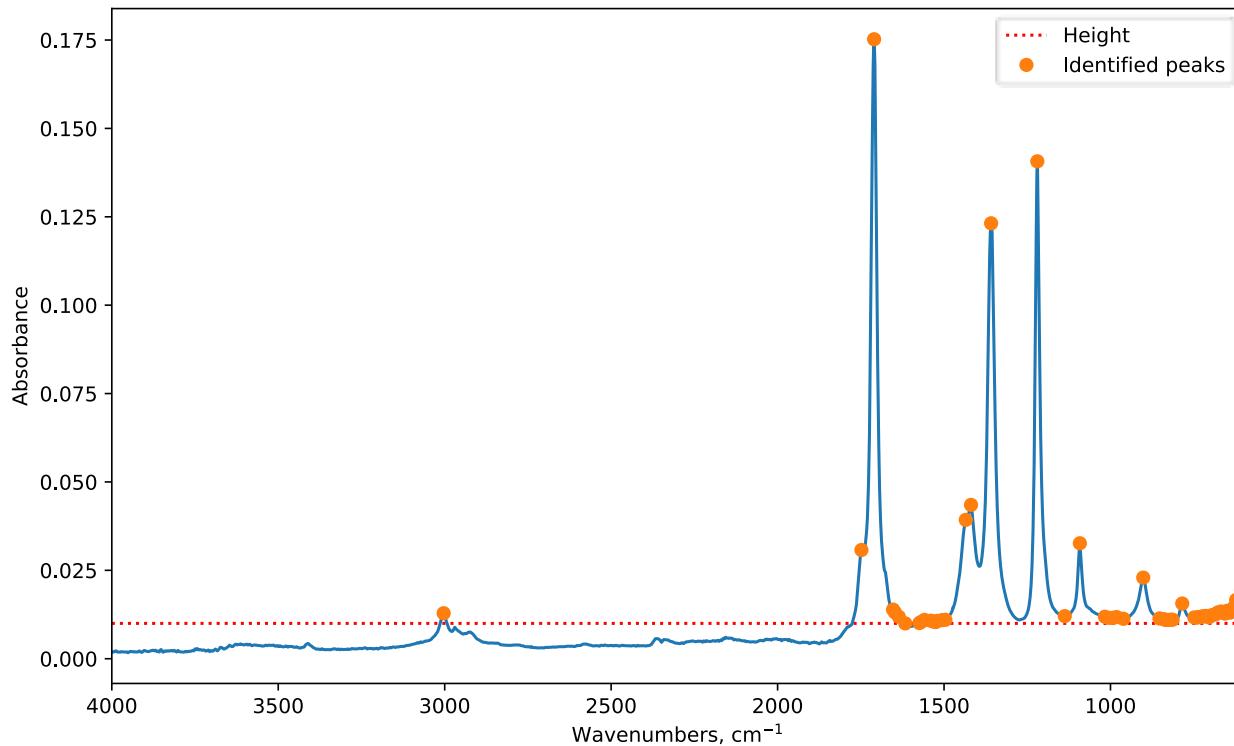
```

height = 0.01
i_peaks = find_peaks(absorb, height=height)[0]

fig = plt.figure(figsize=(10,6))
ax = fig.add_subplot(1,1,1)
ax.hlines(height, 4000, 600, 'r', linestyles='dotted', label='Height')
ax.plot(ir[:,0], absorb)
ax.plot(ir[i_peaks,0], absorb[i_peaks], 'o', label='Identified peaks')
ax.set_xlim(4000, 600)
ax.set_xlabel('Wavenumbers, cm$^{-1}$')
ax.set_ylabel('Absorbance')
ax.legend()

```

<matplotlib.legend.Legend at 0x11377a9f0>



When using keyword arguments, the `find_peaks()` function returns the values used by the keyword arguments in the dictionary. For example, because we used the `height=` argument, the heights are returned.

```
find_peaks(absorb, height=height)
```

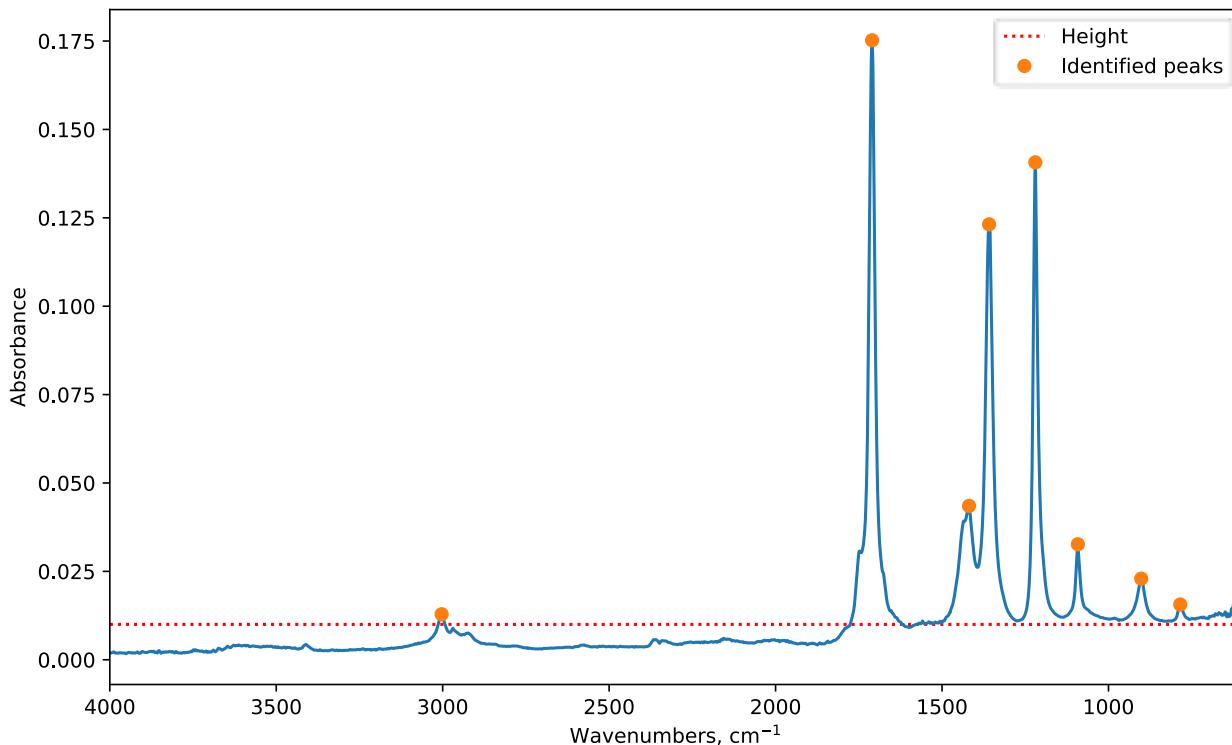
```
(array([ 2, 16, 31, 46, 62, 84, 102, 116, 130, 140, 161,
       176, 199, 215, 220, 235, 249, 262, 270, 279, 291, 307,
       382, 442, 455, 471, 486, 497, 524, 625, 749, 791, 812,
       837, 864, 1020, 1114, 1285, 1573, 1698, 1731, 1858, 1879, 1908,
       1919, 1934, 1948, 1988, 2009, 2020, 2108, 2148, 2172, 2184, 2302,
       2381, 4984]),
{'peak_heights': array([0.01713256, 0.01951099, 0.01586994, 0.0166401 , 0.01443504,
   0.01303453, 0.01354936, 0.01276529, 0.01307347, 0.01335544,
   0.01305901, 0.01252041, 0.0123151 , 0.01175591, 0.01175484,
   0.01211075, 0.0120464 , 0.01183543, 0.0118249 , 0.01181035,
   0.01161127, 0.01167738, 0.01561745, 0.01103975, 0.01103948,
   0.0109757 , 0.01097877, 0.01117627, 0.01139271, 0.0229449 ,
   0.01129687, 0.01178175, 0.01157742, 0.0115503 , 0.01188149,
   0.03266487, 0.01204908, 0.14069589, 0.123156 , 0.04351303,
   0.0392759 , 0.01102015, 0.01090951, 0.01069086, 0.01036943,
   0.01053283, 0.01075127, 0.01098612, 0.0104168 , 0.01008441,
   0.01000628, 0.01187185, 0.01309263, 0.01389863, 0.17522243,
   0.03075821, 0.01286983])})
```

This approach struggles with identifying short peaks without mislabeling non-peaks, so we need another condition to limit what is marked as a peak. The peak prominence (`prominence=`) is how far the apex of a peak is above the base of the peak. The base of the peak may or may not be the baseline of the spectrum itself. By adding this condition, now only peaks that satify both the height *and* prominence condition will be identified.

```
height = 0.01
i_peaks = find_peaks(absorb, height=height, prominence=0.002)[0]

fig = plt.figure(figsize=(10,6))
ax = fig.add_subplot(1,1,1)
ax.hlines(height, 4000, 600, 'r', linestyles='dotted', label='Height')
ax.plot(ir[:,0], absorb)
ax.plot(ir[i_peaks,0], absorb[i_peaks], 'o', label='Identified peaks')
ax.set_xlim(4000, 600)
ax.set_xlabel('Wavenumbers, cm$^{-1}$')
ax.set_ylabel('Absorbance')
ax.legend()
```

<matplotlib.legend.Legend at 0x113759430>



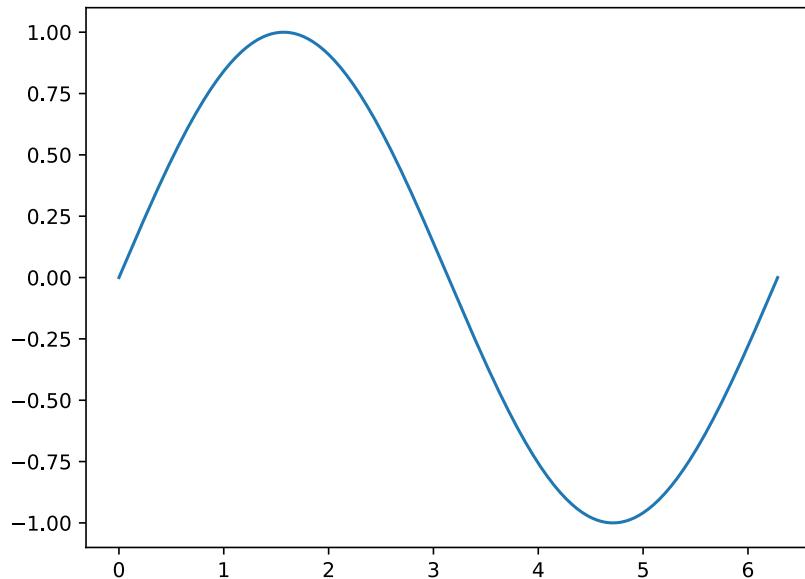
6.1.4 Slopes & Inflection Points

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. Unfortunately, noisy data can make it challenging to examine the slope as the noise causes the slope to fluctuate so much that it sometimes dwarfs the overall signal. It is sometimes recommended that the noise be first removed by signal smoothing covered in section 6.2 before trying to identify signal features. To demonstrate the challenges of noisy data, we will generate both noise-free and noisy synthetic data below and calculate the slopes for both.

```
rng = np.random.default_rng()
```

```
x = np.linspace(0, 2*np.pi, 1000)
y_smooth = np.sin(x)
y_noisy = np.sin(x) + 0.07 * rng.random(len(x))
```

```
plt.plot(x, y_smooth);
```

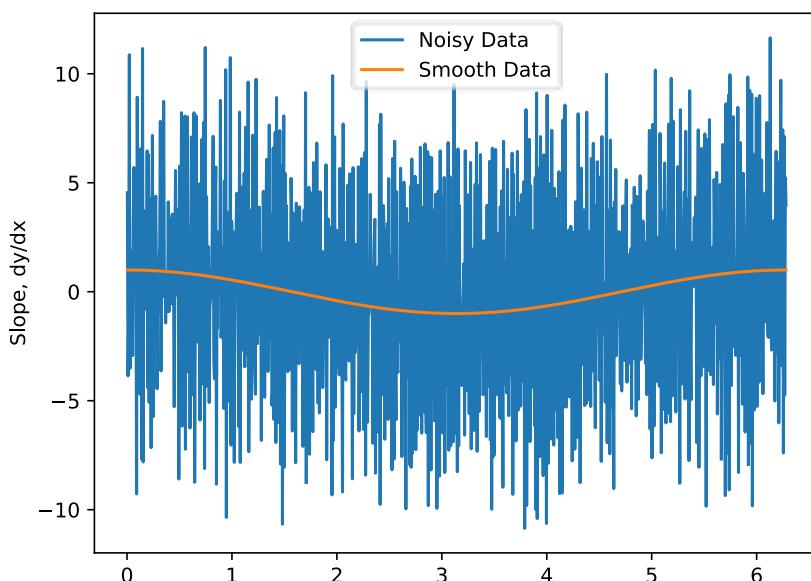


We will use NumPy to calculate the slope using the `np.diff()` function which calculates the differential of a user defined order (`n`). Because the slope is the dy/dx between every pair of adjacent points, the resulting slope data is one data point shorter than the original data. This is important when plotting the data because the length of the x and y values must be the same.

When examining the slope, it is important to use smooth data. In the example below, the slope from the noise in the noisy data dwarfs that of the main signal. Therefore, we will use the slope of the smooth data to find the inflection point below.

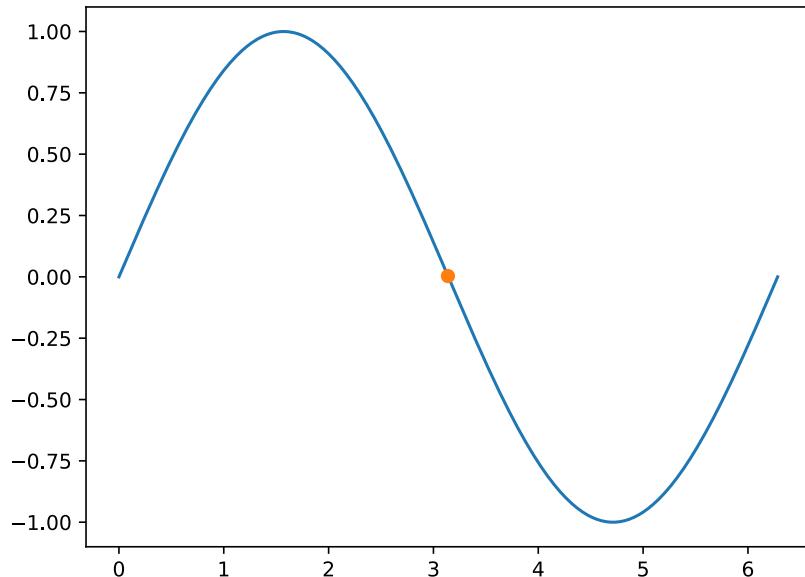
```
dx = 2*np.pi/(1000 - 1)
dy_smooth = np.diff(y_smooth, n=1)
dy_noisy = np.diff(y_noisy, n=1)
x2 = (x[:-1] + x[1:]) / 2 # x values one shorter
```

```
plt.plot(x2, dy_noisy/dx, label='Noisy Data')
plt.plot(x2, dy_smooth/dx, label='Smooth Data')
plt.ylabel('Slope, dy/dx')
plt.legend();
```



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.

```
i = np.argmin(dy_smooth) # finds min slope index
plt.plot(x, y_smooth)
plt.plot(x[i], y_smooth[i], 'o');
```



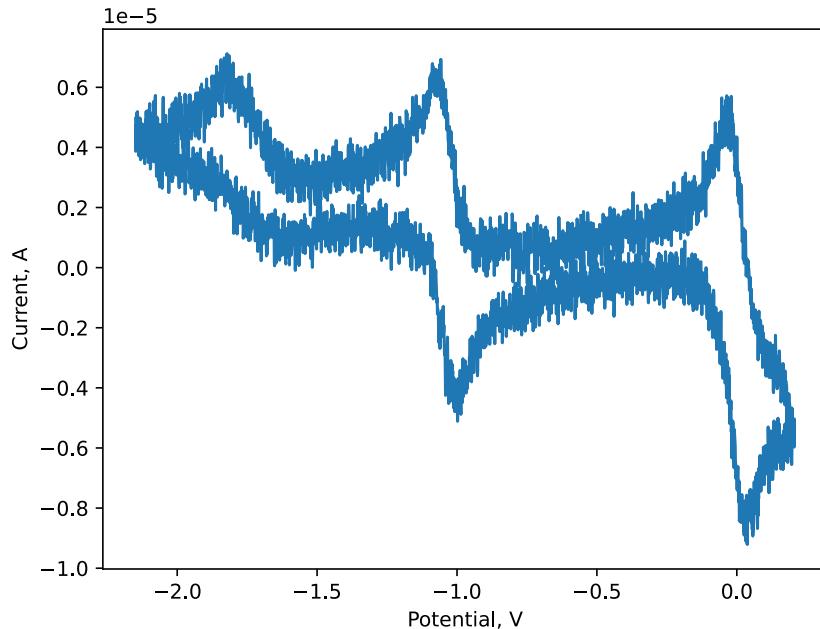
6.2 Smoothing Data

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 a noisy cyclic voltammogram (CV) stored in the file *CV_noisy.csv*.

```
CV = np.genfromtxt('data/CV_noisy.csv', delimiter=',')
potent = CV[:,0]
curr = CV[:,1]

plt.plot(potent, curr)
plt.xlabel('Potential, V')
plt.ylabel('Current, A');
```



6.2.1 Unweighted Average

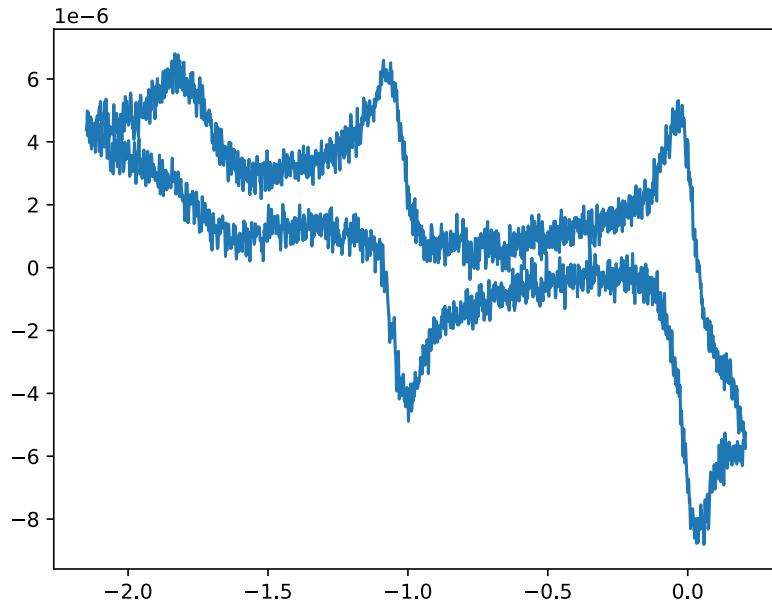
The first and simplest way to smooth data is to take the *moving average* of each data point with its immediate neighbors. This is an *unweighted sliding average smooth* or a *rectangular boxcar smooth*. From noisy data point D_j , we get smoothed data point S_j by the following equation where D_{j-1} 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.

```
sum = curr[:-2] + curr[1:-1] + curr[2:]
rect_smooth = sum / 3

plt.plot(potent[1:-1], rect_smooth);
```



The data are smoothed relative to the original data, but there is still a considerable amount of noise present.

6.2.2 Weighted Averages

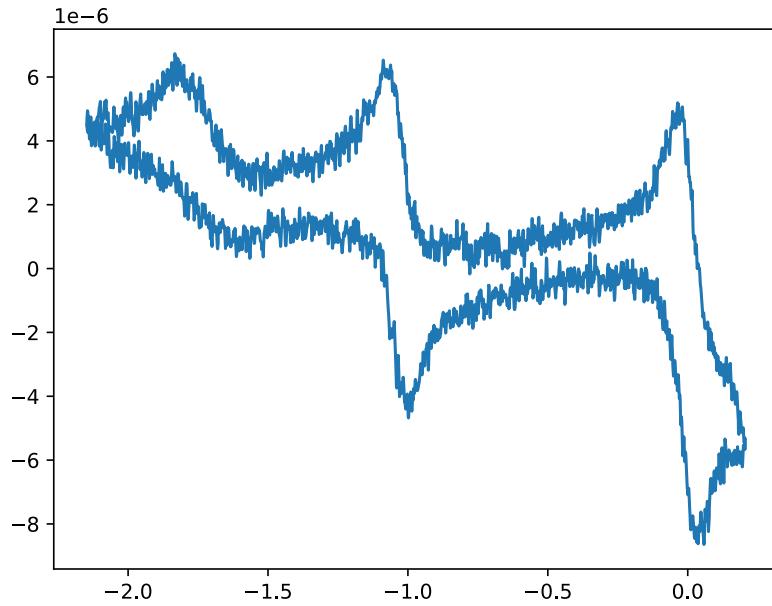
The above method treats each point equally and only takes the average with the immediately adjacent data points. The *triangular smooth* approach averages extra data points with the points closer to the original point weighted more heavily than those further away. For example, if we take the average using five data points, this is described by the following equation.

$$S_j = \frac{D_{j-2} + 2D_{j-1} + 3D_j + 2D_{j+1} + D_{j+2}}{9}$$

The resulting data is shortened by four points as the end points have insufficient neighbors to be averaged.

```
sum = curr[:-4] + 2*curr[1:-3] + 3*curr[2:-2] + 2*curr[3:-1] + curr[4:]
tri_smooth = sum / 9

plt.plot(potent[2:-2], tri_smooth);
```



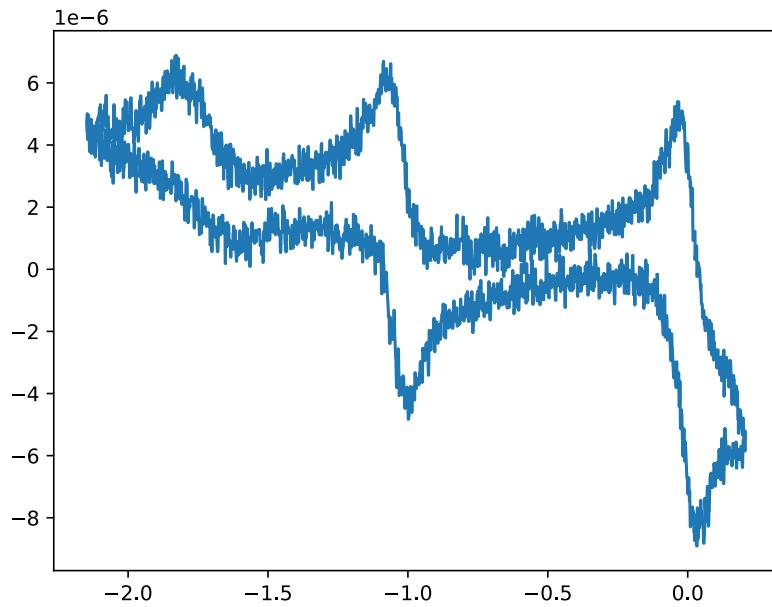
The triangular smooth results in a smoother dataset than the rectangular smooth. This is not surprising as applying the triangular smooth above is mathematically equivalent to applying the rectangular smooth twice.

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.

```
array2d = np.vstack((curr[2:], curr[1:-1], curr[:-2]))
median_smooth = np.median(array2d, axis=0)

plt.plot(potent[1:-1], median_smooth);
```



6.2.4 Savitzky–Golay

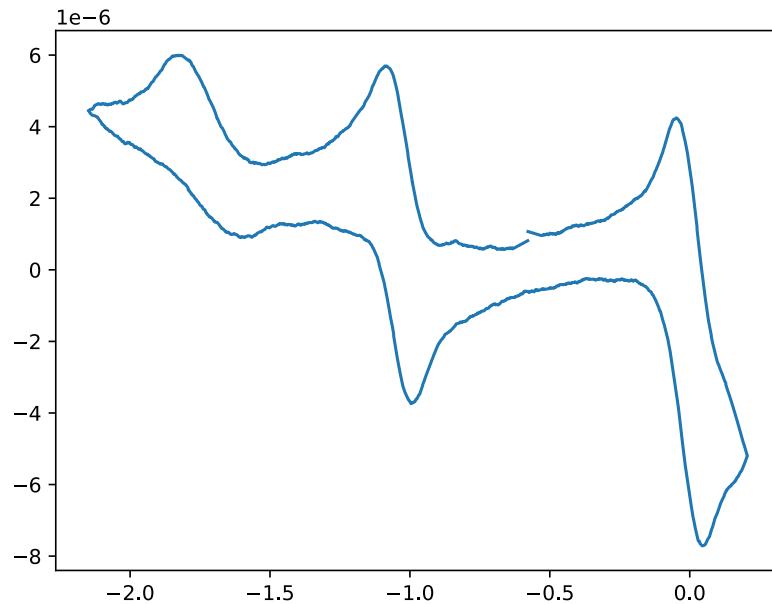
Another approach is the Savitzky–Golay filter which incrementally moves along the noisy data and fits sections (i.e., windows) of data points to a polynomial using least square minimization. While this approach had been previously described in the mathematical literature, Abraham Savitzky and M. J. E. Golay are known for applying it to spectroscopy (<https://doi.org/10.1021/ac60214a047>). Conveniently, SciPy contains a built-in function for this called `savgol_filter()` from the `scipy.signal` module shown below.

```
scipy.signal.savgol_filter(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 experiment 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.

```
from scipy.signal import savgol_filter  
sg_smooth = savgol_filter(curr, 101, 1)
```

```
plt.plot(potent, sg_smooth);
```



The Savitzky–Golay filter appears to have done a decent job removing the noise. Despite there being some remaining noise and other artifacts in the CV, the denoised CV makes it significantly easier to locate the maxima and minima in this example.

6.3 Fourier Transforms

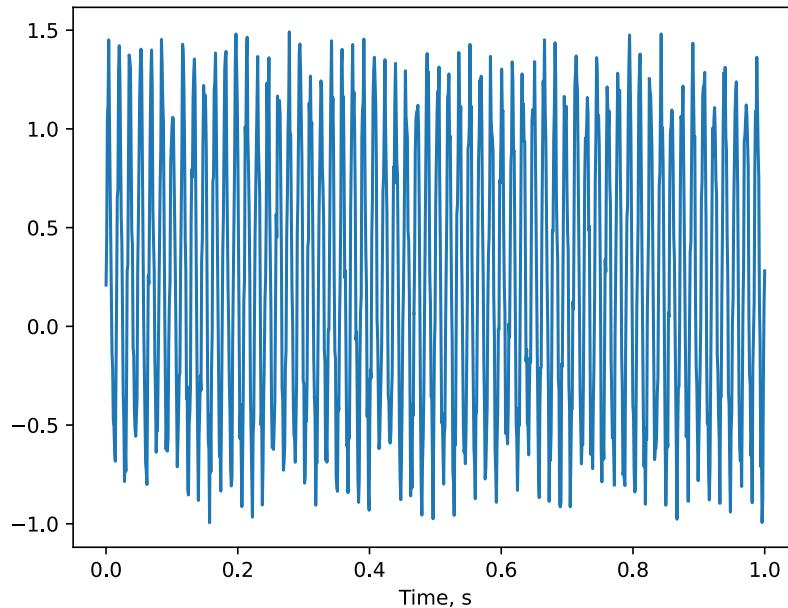
Another approach to filtering noise is to filter based on frequency. Many times, random noise in data occurs at a different frequency than the data itself, and the noise can be reduced by filtering noise frequency ranges while maintaining signal frequencies. If the noise is higher frequency than the signal, it can be filtered out with what is known as a *low-pass* filter. Alternatively, filtering out low-frequency noise is known as a *high-pass* filter, and filtering out noise both 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 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.

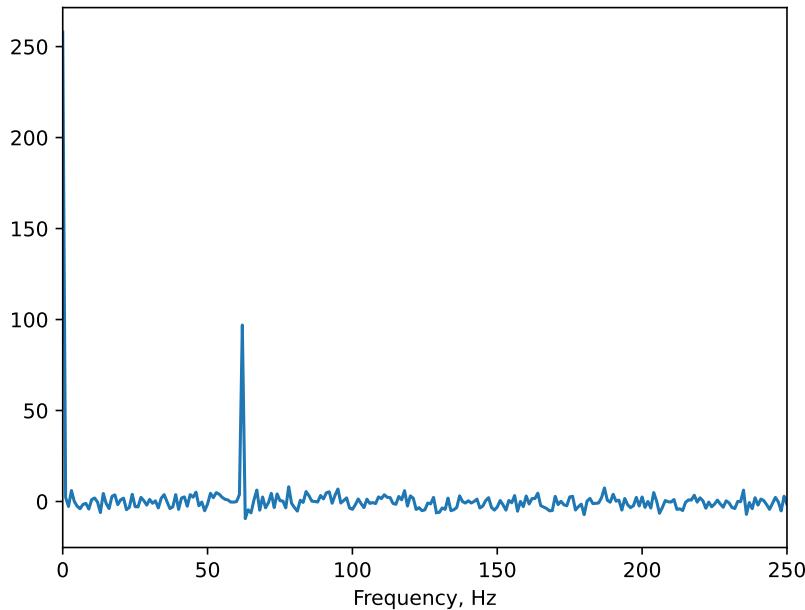
```
t = np.linspace(0,1,1000)
freq = 62.0 # Hz
signal = np.sin(freq*2*np.pi*t)
noise = rng.random(1000)
data = signal + 0.5 * noise
```

```
plt.plot(t, data)
plt.xlabel('Time, s');
```



SciPy contains an entire module called `fft` 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`.

```
from scipy.fft import fft
fdata = fft(data)
plt.plot(fdata.real)
plt.xlim(0,500/2)
plt.xlabel('Frequency, Hz');
```



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 present 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.

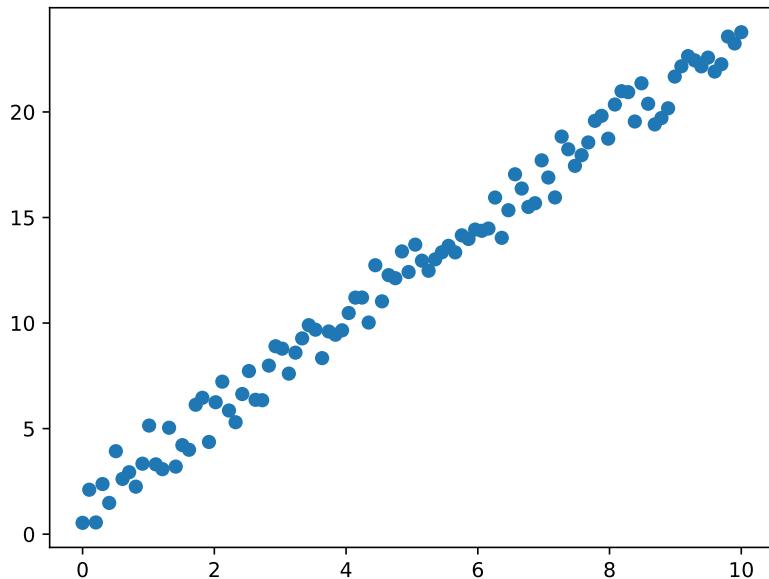
6.4 Fitting & Interpolation

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.

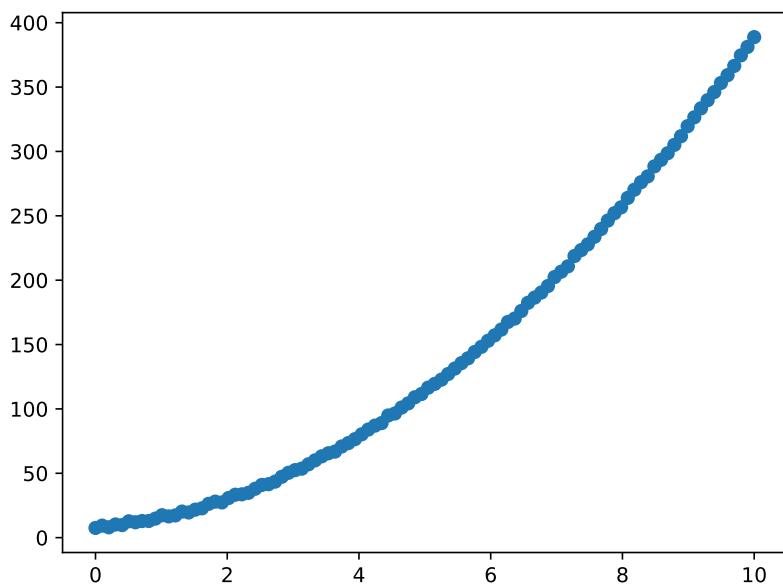
Before we can do our fitting, we need some new, noisy data to examine. A linear set of data with added noise is generated below along with a second-order curve with the noise.

```
x = np.linspace(0,10,100)
noise = rng.random(100)
y_noisy = 2.2 * x + 3 * noise
y2_noisy = 3.4 * x**2 + 4 * x + 7 + 3 * noise
```

```
plt.scatter(x, y_noisy);
```



```
plt.scatter(x, y2_noisy);
```



6.4.1 Linear Regression

Now we can fit the noisy linear data with a line using the NumPy `np.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. This is effectively a *linear regression*.

```
a, b = np.polyfit(x, y_noisy, 1)
print((a, b))
```

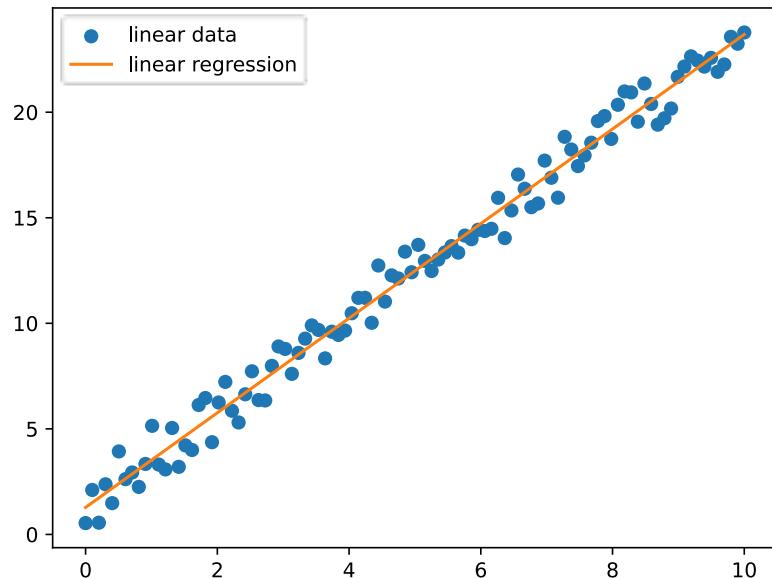
```
(np.float64(2.240683698467965), np.float64(1.2799581449656965))
```

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 $y = 2.17x + 1.66$. 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 `rng.random()` noise should be around 0.5, not zero. This could be remedied either by subtracting 0.5 from the noise or using another random number generator such as the normal distribution, such as `randn()`, which is centered around zero.

We can view our linear regression by plotting a line on top of our data points using the coefficients found above.

```
y_reg = a*x + b

plt.scatter(x, y_noisy, label='linear data')
plt.plot(x, y_reg, 'C1-', label='linear regression')
plt.legend();
```



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.

```
from scipy import stats
stats.linregress(x, y_noisy)
```

```
LinregressResult(slope=np.float64(2.2406836984679646), intercept=np.float64(1.2799581449656952), rvalue
```

Note

We are starting to see examples of functions that return multiple values which can be assigned to multiple variables using [tuple unpacking](#) like below.

$$x, y = \text{func}(z)$$

There may be times when you don't need all of the returned values from a function. In these instances, it is common to use `__` (double underscore) as a junk variable which is broadly understood to store information that will never be used in the code. You may also see a `_` (single underscore) used for this purpose, but this is discouraged as a single underscore is also used by the Python interpreter to store the last output.

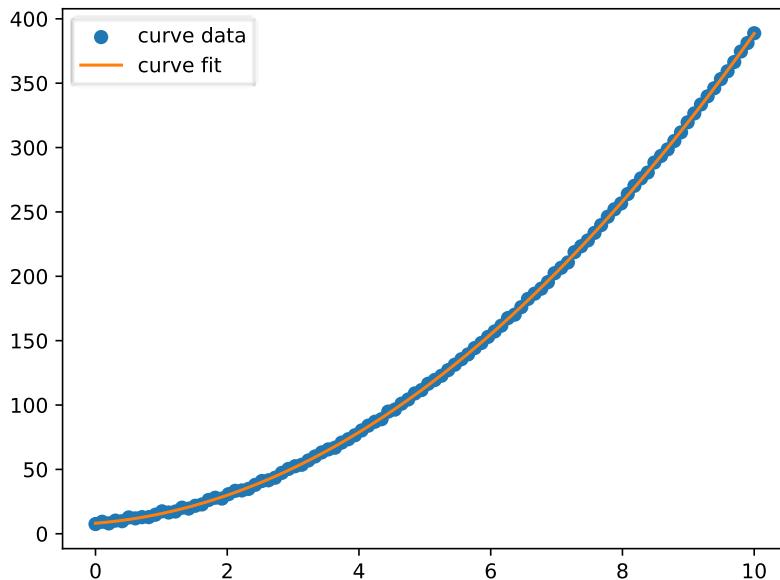
6.4.2 Polynomial Fitting

Fitting to a polynomial of a higher order works the same way except that the order is above one. You will need to already know the order of the polynomial, or you can make a guess and see how well a particular order fits the data. Below, the `np.polyfit()` function determines the second-order data can be fit by the equation $y = 3.40x^2 + 3.95x + 8.70$. We can again plot this fit equation over our data points to see how well the data agree with our equation.

```
a, b, c = np.polyfit(x, y2_noisy, 2)
print((a, b, c))
```

```
(np.float64(3.394946026145204), np.float64(4.091223437015926), np.float64(8.19657608473501))
```

```
y_fit = a*x**2 + b*x + c
plt.scatter(x, y2_noisy, label='curve data')
plt.plot(x, y_fit, 'C1-', label='curve fit')
plt.legend();
```



6.4.3 Multivariable Linear Regression

Multivariable linear regression (aka. multiple linear regression) is similar to the linear regression seen in [section 6.4.1](#) except that there are multiple independant variable. This takes the form below where y is the dependant variable, x are the independant variables (plural) with coefficients a , and b is the bias term. There are k independant variables below.

$$y = a_0x_0 + \dots + a_{k-1}x_{k-1} + b$$

The goal is to solve for the a coefficients and the value for b given a series of x -values with their corresponding y -values. Essentially, this is taking regular linear regression to three dimensions or higher. There are multiple methods available in Python to solve this type of problem including, but not limited to, the following.

1. Scikit-learn's `LinearRegressor()` demonstrated in [section 13.1.3](#)
2. Moore-Penrose psudoinvere and some matrix math demonstrated in [section 8.3.2](#)
3. NumPy's `np.linalg.lstsq()` function
4. Using optimization algorithms to fit the equation similar to what is demonstrated in [section 14.2](#)

The first option will often involve the fewest lines of code but does require knowledge of using the scikit-learn library. Being that the other options 1, 2, and 4 either require other libraries or specialized knowledge not yet addressed, we will solve a multivariable linear regression problem using the `np.linalg.lstsq()` function.

In the example below, we have an array `y` which contains our dependant variable values and array `X` which contains our independant variable values. For this approach, we want these two arrays to be related by the following equation where `a` is an array containing the our coefficents and bias term. The issue is that our array `X` has too few columns to take the dot product of, so there is nothing that multiplies by b .

$$y = a \bullet X$$

```
X = np.array([[ 2, 11, 10, 7],  
              [ 7, 13, 2, 10],  
              [ 3, 2, 8, 14],  
              [11, 11, 11, 12],  
              [ 8, 2, 12, 7],  
              [ 8, 6, 3, 13]])  
  
y = np.array([192.36, 254.1, 175.1, 284.4, 145.2, 221.3])
```

We need to add a column of ones to array `X` as is done below which simply adds ones. Now when performing the above multiplication, b is always multiplied by 1 to return b .

```
X = np.c_[X, np.ones(6)]  
  
array([[ 2., 11., 10., 7., 1.],  
       [ 7., 13., 2., 10., 1.],  
       [ 3., 2., 8., 14., 1.],  
       [11., 11., 11., 12., 1.],  
       [ 8., 2., 12., 7., 1.],  
       [ 8., 6., 3., 13., 1.]])
```

To solve for array `a`, we will use the `np.linalg.lstsq()` function which take the arrays containing independant and

dependant variables in this order.

```
a = np.linalg.lstsq(X, y)
a
```

```
(array([5.22016673, 9.07380853, 1.22219907, 8.66136646, 9.77747831]),
 array([2.02564125]),
 np.int32(5),
 array([40.69036598, 11.56475425, 9.36236285, 6.61646926, 0.34506865]))
```

The output includes the coefficients plus other information about the fit such as the sum of the squared residuals from the fit. If you just want the coefficients, use indexing like below.

```
a[0]
```

```
array([5.22016673, 9.07380853, 1.22219907, 8.66136646, 9.77747831])
```

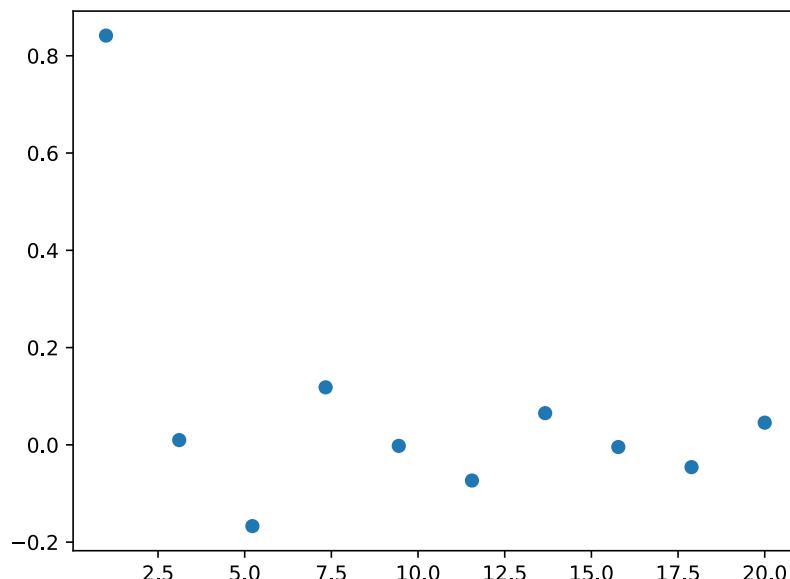
This result is interpreted as the equation $y = 5.22x_0 + 9.07x_1 + 1.22x_2 + 8.66x_3 + 9.78$.

6.4.4 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.

```
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 3.

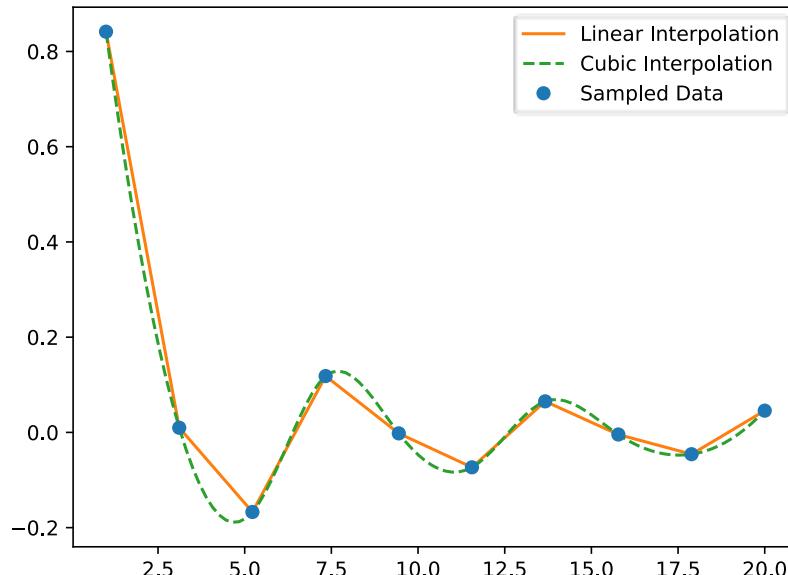
Table 3 Modes for `interp1d()` Method

Kind	Description
<code>linear()</code>	Linear interpolation between data points
<code>zero()</code>	Constant value until the next data point
<code>nearest()</code>	Predicts values equaling the closest data point
<code>quadratic()</code>	Interpolates with a second-order spline
<code>cubic()</code>	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.

```
from scipy import interpolate
f = interpolate.interp1d(x, y, kind='linear')
f2 = interpolate.interp1d(x, y, kind='cubic')

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, 'o', label='Sampled Data')
plt.legend();
```



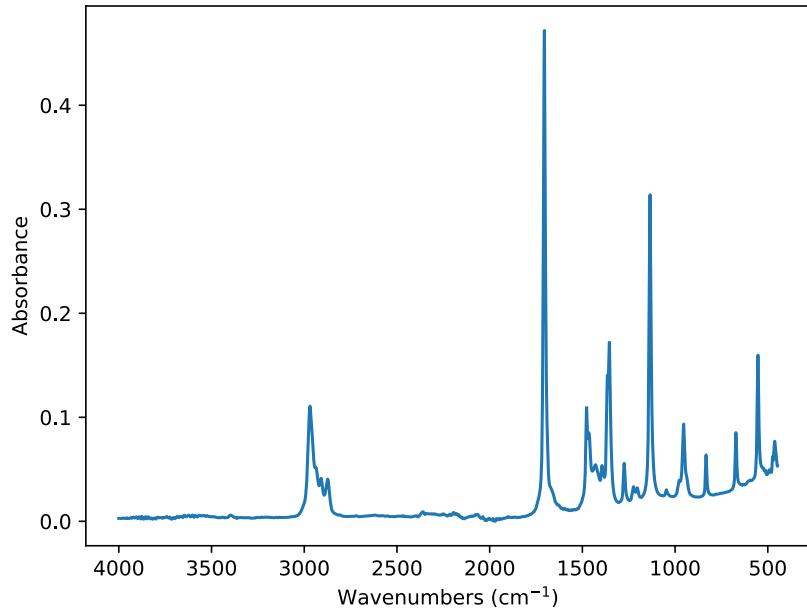
6.5 Baseline Correction

The baseline of chemical spectra may sometimes slope or undulate requiring baseline correction. This is a two step process - first, the baseline needs to be identified and then the baseline is subtracted from the original spectrum. Predicting a baseline for a spectrum is not a trivial task. Fortunately, the [pybaselines](#) library provides Python implementations of various algorithms for determining the baseline. Because pybaselines is not a standard library with Anaconda or Colab, it needs to be installed using either [pip](#) or [conda](#).

For our example below, we will correct the baseline of an IR spectrum of 2-pentanone. We can see that the baseline curves upward at frequencies below 2000 cm⁻¹.

```
IR_spec = np.genfromtxt('data/IR_2pentanone.csv', delimiter=',')
wavenums = IR_spec[:,0]
absorb = IR_spec[:,1]

plt.plot(wavenums, absorb)
plt.xlabel('Wavenumbers (cm$^{-1}$)')
plt.ylabel('Absorbance')
plt.gca().invert_xaxis();
```



To identify a baseline, we will use the [Baseline](#) module of pybaselines imported below. The first step is to create a baseline fitter object which accepts the x-axis data from your spectrum.

```
from pybaselines import Baseline
fitter = Baseline(x_data=wavenums)
```

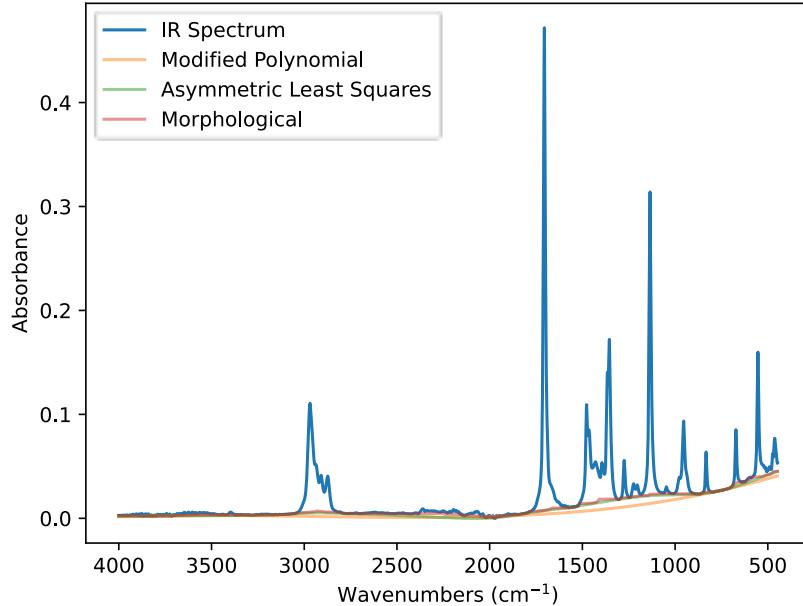
Next, the fitter object is used to predict the baseline using various baseline algorithms. There are numerous algorithms available, and many algorithms have multiple parameters that fine tune how the baseline is identified. Finding the ideal algorithm and parameters to use can come down to trial-and-error, so feel free to try a few and see what works the best. We will try the modified polynomial, asymmetric least squares, and morphological based algorithms below, but there are many others to choose from. The output of each prediction is a background and the parameters from the baseline fit.

```

bg1, params1 = fitter.modpoly(absorb, poly_order=3)
bg2, params2 = fitter.asls(absorb, lam=1e7, p=0.0002)
bg3, params3 = fitter.mor(absorb, half_window=200)

plt.plot(wavenums, absorb, label='IR Spectrum')
plt.plot(wavenums, bg1, alpha=0.5, label='Modified Polynomial')
plt.plot(wavenums, bg2, alpha=0.5, label='Asymmetric Least Squares')
plt.plot(wavenums, bg3, alpha=0.5, label='Morphological')
plt.xlabel('Wavenumbers (cm$^{-1}$)')
plt.ylabel('Absorbance')
plt.legend()
plt.gca().invert_xaxis()

```



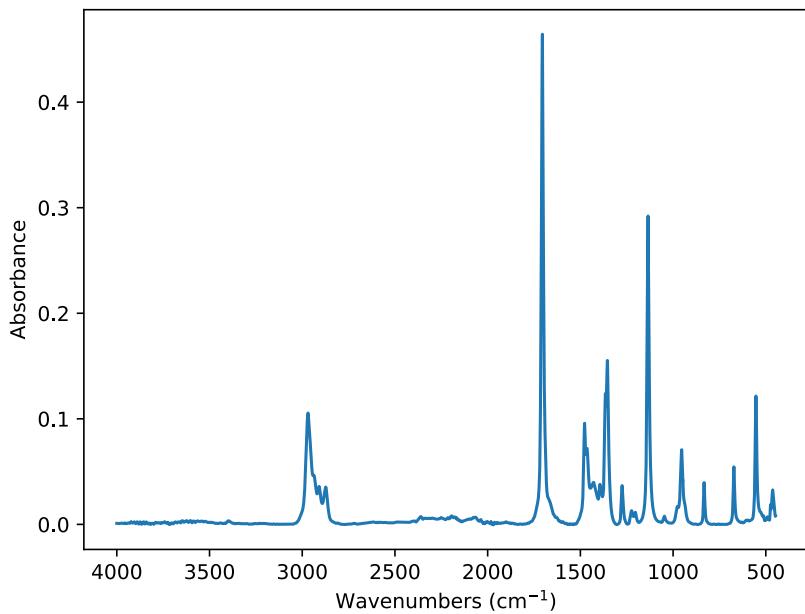
Finally, we will subtract the baseline from the original data. The good news is that the baseline fitter generated the baseline as a NumPy array with the same size as the original data, so subtraction is a matter of subtracting one array from another.

```

absorb_corrected = absorb - bg2

plt.plot(wavenums, absorb_corrected)
plt.xlabel('Wavenumbers (cm$^{-1}$)')
plt.ylabel('Absorbance')
plt.gca().invert_xaxis()

```



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/> (free resource)
2. Downey, Allen B. Think DSP, Green Tea Press, 2016. <http://greenteapress.com/wp/think-dsp/> (free resource)
3. O'Haver, T. C. An Introduction to Signal Processing in Chemical Measurement. *J. Chem. Educ.* **1991**, 68 (6), A147-A150. <https://doi.org/10.1021/ed068pA147>
4. Savitzky, A.; Golay, M.J.E. Smoothing and Differentiation of Data by Simplified Least Squares Procedures. *Anal. Chem.* **1964**, 36 (8) 1627–1639. <https://doi.org/10.1021/ac60214a047>

Exercises

Complete the following exercises in a Jupyter notebook. Any data file(s) referred to in the problems can be found in the [data](#) folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

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 points. Plot the data with a marker on both inflection points. Hint: There are two inflections points in these data with one running in the reverse direction making it have a negative slope.
3. Generate noisy synthetic data from the following code.

```

from scipy.signal import sawtooth
import numpy as np
rng = np.random.default_rng()
t = np.linspace(0, 4, 1000)
sig = sawtooth(2 * np.pi * t) + rng.random(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}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.
5. The wavelength of emitted light (λ) from hydrogen is related to the electron energy level transitions by the following equation where R_∞ 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

```

n_i = [3, 4, 5, 6, 7]
wl = [656.1, 485.2, 433.2, 409.1, 396.4]

```

Calculate a value for the Rydberg constant (R_∞) 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.

Conc A (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

```
conc = [0.10, 0.16, 0.20, 0.25, 0.41, 0.55]
rate = [0.0034, 0.0087, 0.0136, 0.0213, 0.0572, 0.103]
```

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 0.481.

Absorb. (@ 504 nm)	Red 40 (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

```
ab = [0.125, 0.940, 2.36, 2.63, 3.31, 3.77]
conc = [0.150, 1.13, 2.84, 3.16, 3.98, 4.53]
```

8. The following are points on the 2s radial wave function (Ψ) 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)	Ψ
1.0	0.21
5.0	-0.087
9.0	-0.027
13.0	-0.0058
17.0	-0.00108

```
radius = [1.0, 5.0, 9.0, 13.0, 17.0]
psi = [0.21, -0.087, -0.027, -0.0058, -0.00108]
```

9. The file **Cp2Fe_Mossbauer.txt** contains Mossbauer data for a ferrocene complex where the left data column is velocity in millimeters per second and the right column is relative transmission. Using Python, determine the velocities of the six negative peaks. Plot the spectrum with dots on lowest point of each negative peak, and be sure to label your axes complete with units.
10. Load the file **XRF_Cu.csv**, which contains X-ray fluorescence (XRF) data for elemental copper, and use Python to determine the energy in eV of the two peaks. Notice that the x-axis is not in eV yet (see row 17 of data). You are advised to load the data using a pandas function, and setting a threshold will likely be necessary.

Chapter 7: Image Processing & Analysis

Contents

- 7.1 Basic Image Structure
- 7.2 Basic Image Manipulation
- 7.3 Scikit-Image Examples
- Further Reading
- Exercises

Images are a major data format in chemistry and other sciences. 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](#), 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](#), scikit-image is designed for scientific image analysis while pillow is intended for more fundamental operations such as image rotation and cropping.

Like SciPy, scikit-image stores most of its functions in modules, so it is common to import modules individually. For example, if the user wants to import the `color` module, it is imported using the following code.

```
from skimage import color
```

Multiple modules can also be imported in a single import such as below. A list of modules and their description are shown in Table 1, and additional information can be found on the project website at <http://scikit-image.org/>.

```
from skimage import color, data, io
```

We can also import a single function from a module using the following code structure.

```
from skimage.module import function
```

Table 1 Scikit-Image Modules

Module	Description
<code>color</code>	Converts images between color spaces
<code>data</code>	Provides sample images
<code>draw</code>	Generates coordinates of geometric shapes
<code>exposure</code>	Examines and modifies image exposure levels
<code>external.tifffile</code>	Handles reading, writing, and visualizing TIFF files
<code>feature</code>	Feature detection and calculation
<code>filters</code>	Contains various image filters and functions for calculating threshold values
<code>filters.rank</code>	Returns localized measurements in the image.
<code>graph</code>	Finds optimized paths across the image
<code>io</code>	Supports reading and writing images and contains a function for displaying images
<code>measure</code>	Performs a variety of measurements and calculations on or between two images
<code>morphology</code>	Generates objects of a specified morphology
<code>novice</code>	Provides simple image functions for beginners
<code>restoration</code>	Includes image restoration tools
<code>segmentation</code>	Identifies boundaries in an image
<code>transform</code>	Performs image transformations including scaling and image rotation
<code>util</code>	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
<code>viewer</code>	Image viewer tools

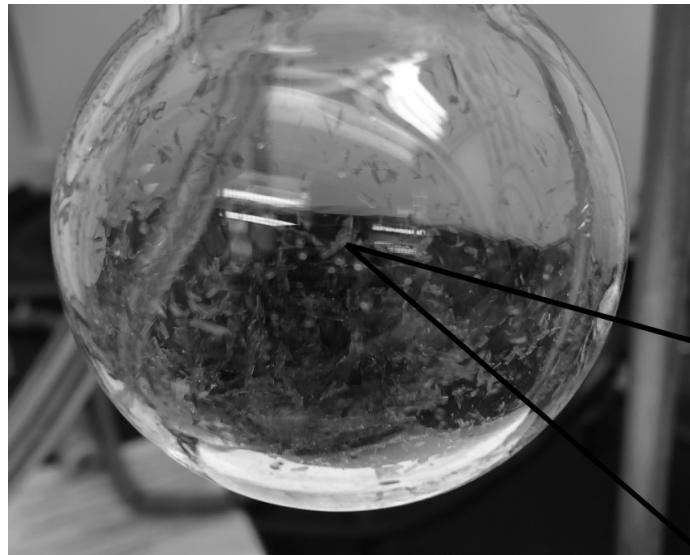
This chapter assumes the following imports. Because we will be doing some plotting, this includes the following matplotlib import and that inline plotting is enabled. In addition, there are functions inside scikit-image that are not in a module, so we also need to `import skimage` as well.

```
import matplotlib.pyplot as plt
import skimage
from skimage import data, io, color
```

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 arrays.

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 1 shows a grayscale photo of a flask containing crystals, with a 10×10 pixel excerpt showing the brightness values from the photo. While there is another major class of images known as *vector* images, we will restrict ourself to dealing with raster images in this chapter as primary scientific data tend to be raster images.



data from the image

```
75, 72, 74, 75, 73, 73, 69, 63, 59, 57  
74, 74, 75, 74, 70, 67, 62, 54, 48, 47  
73, 73, 71, 71, 73, 67, 56, 51, 45, 48  
70, 69, 67, 70, 75, 69, 55, 45, 51, 48  
67, 67, 67, 70, 73, 67, 54, 45, 51, 47  
64, 68, 70, 69, 65, 58, 53, 51, 46, 48  
64, 66, 67, 62, 54, 49, 51, 53, 45, 51  
63, 60, 58, 54, 48, 46, 48, 48, 48, 49  
58, 55, 54, 53, 50, 48, 46, 41, 46, 44  
54, 52, 55, 57, 54, 50, 46, 39, 38, 41
```

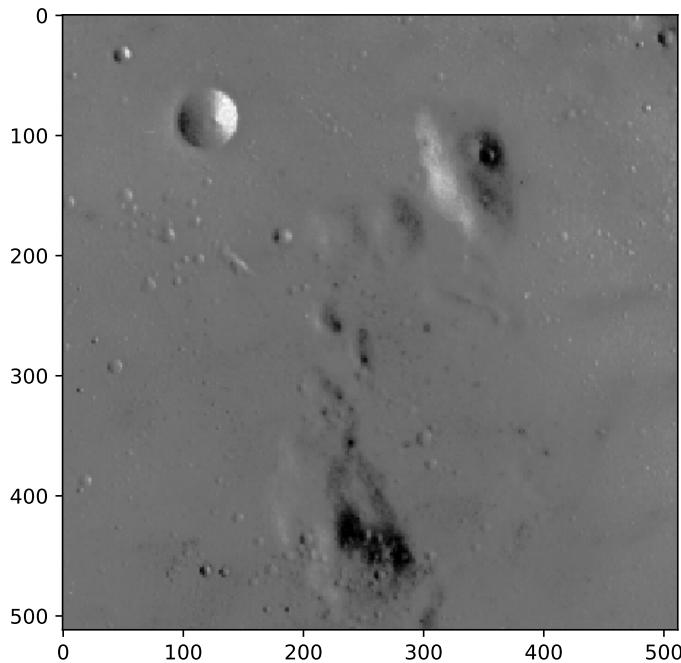
Figure 1 An excerpts of values from a grayscale image showing values representing the brightness of each pixel.

7.1.1 Loading Images

The scikit-image library includes a `data` module containing a series of images for the user to experiment with. To display images in the notebook, use the `io.imshow()` 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.

```
moon = data.moon()  
io.imshow(moon);
```

```
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4458/786239732.py:2: FutureWarning: `imshow`  
io.imshow(moon);
```



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 $0 \rightarrow 255$.

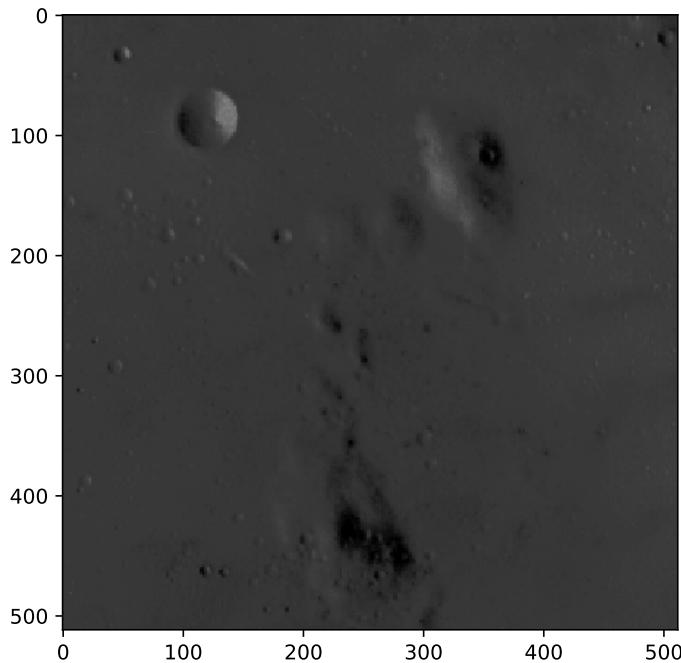
```
moon
```

```
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]], shape=(512, 512), 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.

```
moon_dark = moon // 2
io.imshow(moon_dark);
```

```
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4458/4184250868.py:2: FutureWarning: `imshow`  
io.imshow(moon_dark);
```



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 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. Their perceived color is a mixture or blend of the red, green, and blue values. 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 values from each of the three color channels.

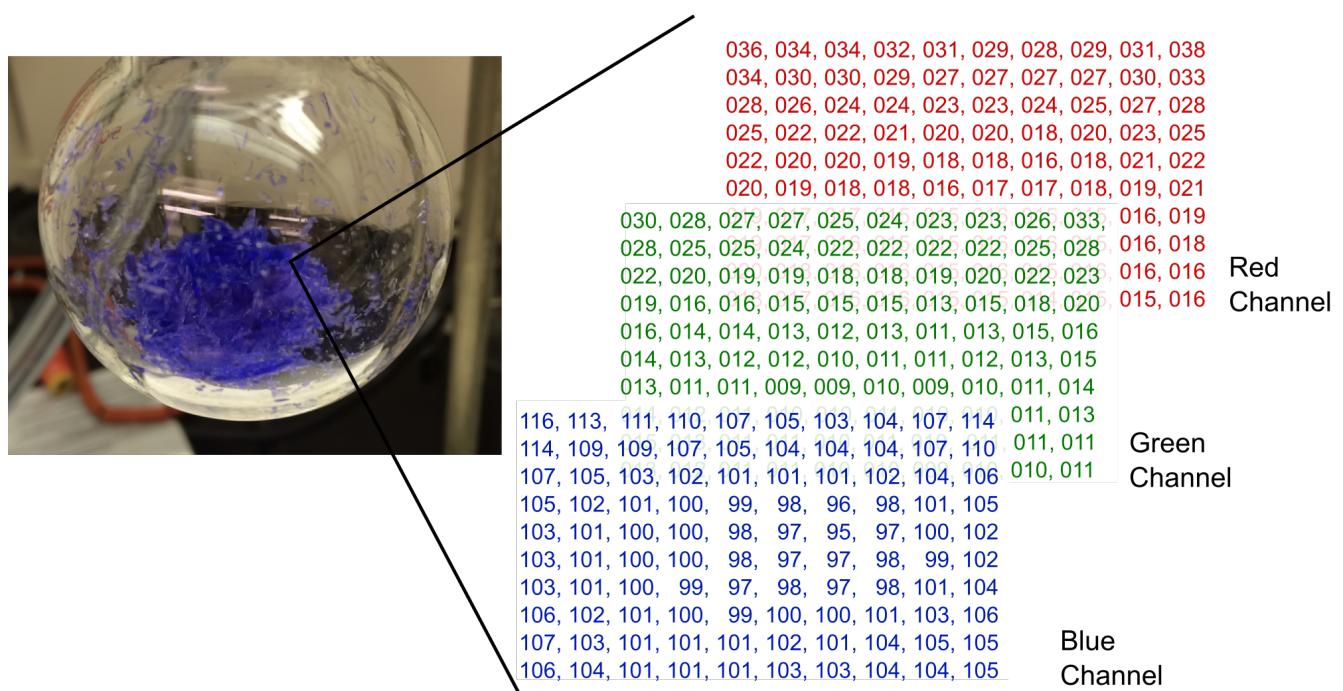


Figure 2 An excerpt of the red, green, and blue color channels for a small portion of a color image. The values in each channel represent the brightness of that color in each pixel.

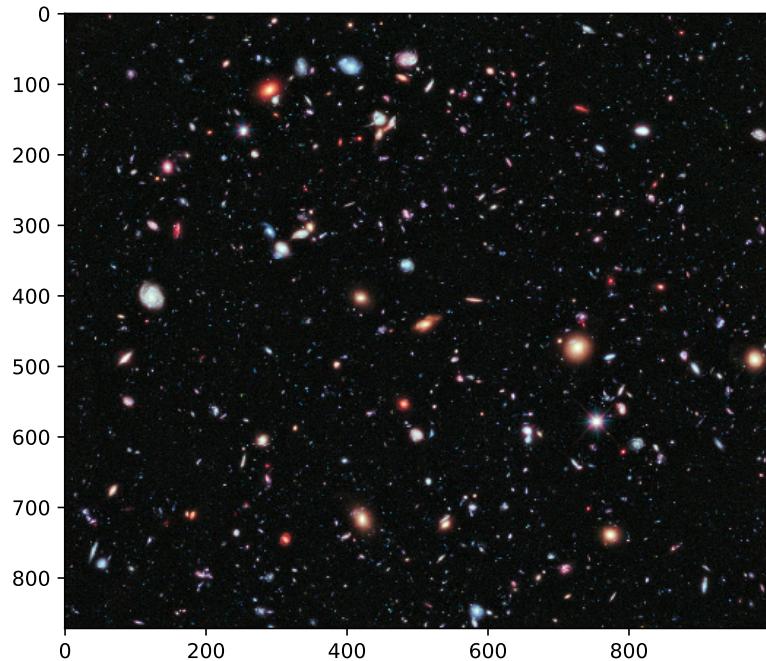
By [scikit-image convention](#), the encoding of color images in arrays is shown below, and the colors are in the order red, green, and then blue.

```
[row, column, channel]
```

We can look at an example of a color photo by loading an image from the Hubble space telescope. This image is included with the scikit-image library for users to experiment with.

```
hubble = data.hubble_deep_field()  
io.imshow(hubble);
```

```
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4458/1071723200.py:2: FutureWarning: `imshow`  
io.imshow(hubble);
```



```
hubble
```

```

array([[[15,  7,  4],
       [15,  9,  9],
       [ 9,  4,  8],
       ...,
       [18, 11,  5],
       [16, 19, 10],
       [15, 10,  6]],

      [[ 2,  7,  0],
       [ 5, 11,  7],
       [13, 19, 17],
       ...,
       [11, 10,  5],
       [13, 18, 11],
       [ 9, 11,  6]],

      [[10, 15,  9],
       [13, 18, 14],
       [18, 22, 23],
       ...,
       [ 1,  2,  0],
       [14, 15, 10],
       [ 8, 14, 10]],

      ...,

      [[19, 20, 14],
       [15, 15, 13],
       [13, 13, 13],
       ...,
       [ 2,  6,  5],
       [12, 14, 13],
       [ 7,  9,  8]],

      [[13, 10,  5],
       [ 9, 11,  8],
       [12, 18, 16],
       ...,
       [ 5,  9,  8],
       [ 6, 12, 10],
       [ 7, 13,  9]],

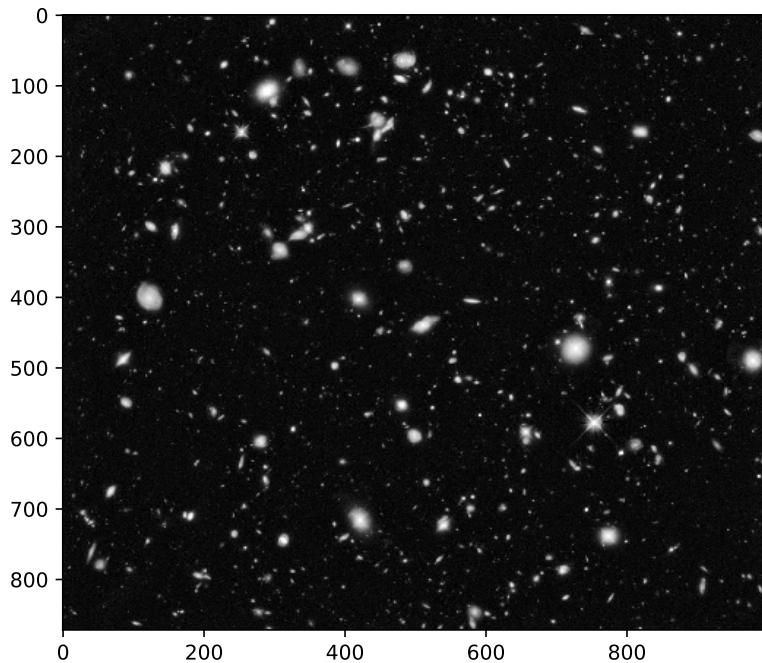
      [[21, 16, 12],
       [10, 12,  9],
       [ 9, 20, 16],
       ...,
       [11, 15, 14],
       [ 9, 18, 15],
       [ 7, 18, 12]]], shape=(872, 1000, 3), dtype=uint8)

```

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.

```
io.imshow(hubble[:, :, 0]);
```

```
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4458/3815783581.py:1: FutureWarning: `imshow`
```

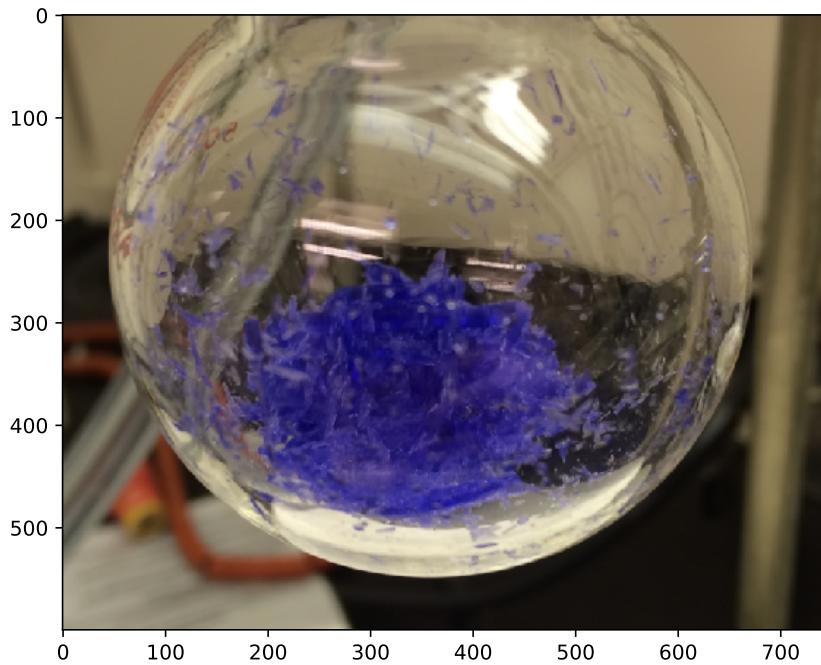


7.1.3 External Images

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, 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 $[\text{Ni}(\text{CH}_3\text{CN})_6][\text{BF}_4]_2$ crystals is read into Python.

```
flask = io.imread('data/flask.png')
io.imshow(flask);
```

```
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4458/985208856.py:2: FutureWarning: `imshow`  
io.imshow(flask);
```



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 Portable Network Graphics (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. PNG images support an alpha channel as do many image formats, but JPG/JPEG images do not support this feature.

```
flask
```

```

array([[[102, 86, 60, 255],
       [107, 90, 63, 255],
       [113, 95, 67, 255],
       ...,
       [ 88, 72, 46, 255],
       [ 90, 74, 48, 255],
       [ 92, 76, 50, 255]],

      [[103, 87, 61, 255],
       [107, 90, 63, 255],
       [112, 95, 67, 255],
       ...,
       [ 88, 72, 46, 255],
       [ 90, 74, 48, 255],
       [ 93, 77, 51, 255]],

      [[101, 85, 59, 255],
       [107, 90, 62, 255],
       [112, 95, 67, 255],
       ...,
       [ 88, 72, 46, 255],
       [ 91, 75, 49, 255],
       [ 93, 77, 51, 255]],

      ...,
      [[161, 156, 136, 255],
       [161, 156, 136, 255],
       [161, 156, 136, 255],
       ...,
       [ 18, 15, 10, 255],
       [ 19, 16, 9, 255],
       [ 20, 17, 10, 255]],

      [[160, 155, 135, 255],
       [161, 156, 136, 255],
       [161, 156, 136, 255],
       ...,
       [ 18, 15, 10, 255],
       [ 18, 15, 9, 255],
       [ 20, 17, 10, 255]],

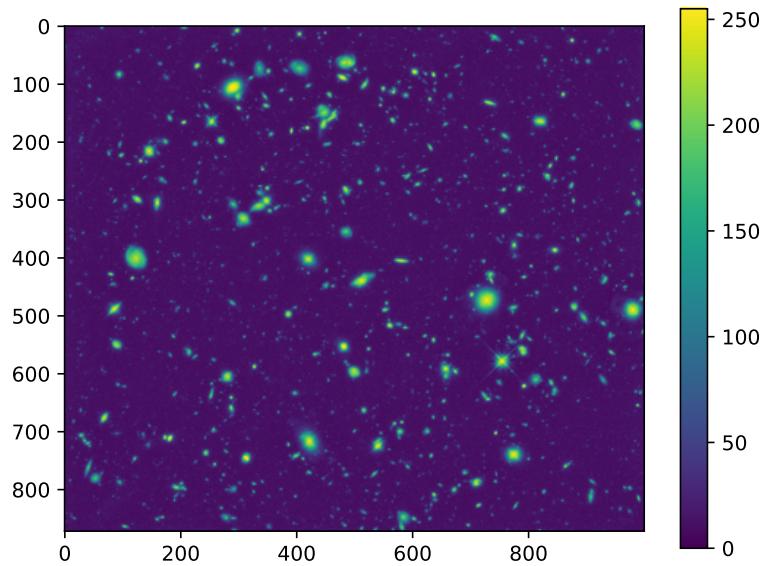
      [[160, 155, 135, 255],
       [160, 155, 135, 255],
       [161, 156, 136, 255],
       ...,
       [ 19, 16, 11, 255],
       [ 18, 15, 9, 255],
       [ 20, 17, 10, 255]]], shape=(600, 744, 4), dtype=uint8)

```

7.1.4 Colormaps

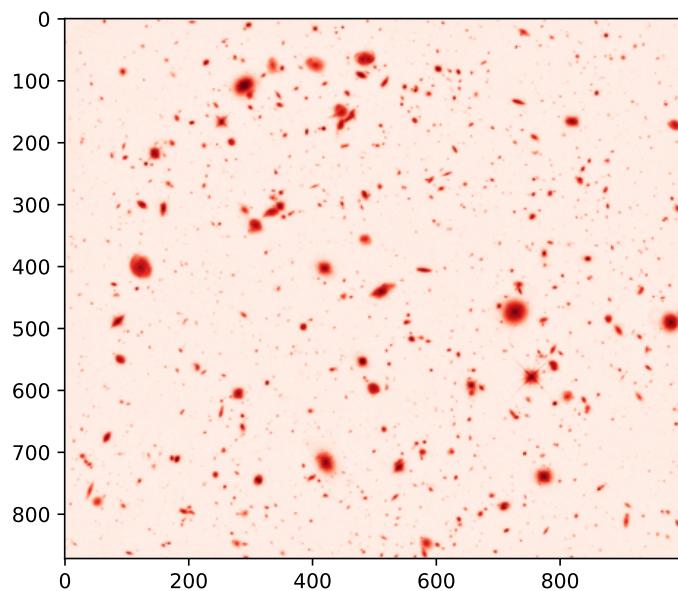
So far, we having been displaying images using the scikit-image `io.imshow()` function which is intended specifically for images. Matplotlib also provides a `plt.imshow()` function that does roughly the same thing but with some important differences. Below is the red color channel from the hubble image 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. This is known as **false color** because the colors in the image are not the real image colors. By default, the colormap *viridis* is used, but there are many other colormaps available to choose from in matplotlib.

```
import matplotlib.pyplot as plt
plt.imshow(hubble[:, :, 0])
plt.colorbar();
```



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 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.

```
plt.imshow(hubble[:, :, 0], cmap='Reds');
```



7.1.5 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 `io.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`.

```
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 2 lists some common functions. See the [scikit-image website](#) for a more complete list.

Table 2 Common Functions from the `color` Module

Function	Description
<code>color.rgb2gray()</code>	Converts from RGB to grayscale
<code>color.gray2rgb()</code>	Converts grayscale to RGB; by just replicating the gray values into three color channels
<code>color.hsv2rgb()</code>	HSV to RGB conversion
<code>color.xyz2rgb()</code>	XYZ to RGB conversion

```
hubble_gray = color.rgb2gray(hubble)
hubble_gray
```

```
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 ]], shape=(872, 1000))
```

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 $0 \rightarrow 255$. There is no rule that says lightness and darkness values need to be represented as integers. Above, they are presented as floats from $0 \rightarrow 1$. 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 $0 \rightarrow 255$ or floats from $0 \rightarrow 1$. However, there are other ranges outlined in Table 3. Though you may have 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. The approximate part is because there are equal numbers of positive and negative integers, and being that zero is a positive integer, zero is not the exact center. 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 3. These functions are not contained in a module, so you will need to just do an `import skimage` to get access to them, which was done at the start of this chapter. 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 3 Scikit-Image Functions for Converting Data Types

Functions	Description
<code>skimage.img_as_ubyte()</code>	Converts to integers from $0 \rightarrow 255$
<code>skimage.img_as_uint()</code>	Converts to integers from $0 \rightarrow 65535$
<code>skimage.img_as_int()</code>	Converts to integers from $-32768 \rightarrow 32767$
<code>skimage.img_as_bool()</code>	Converts to boolean (i.e., <code>True</code> or <code>False</code>) format
<code>skimage.img_as_float32()</code>	Converts to floats from $0 \rightarrow 1$ with 32-bit precision
<code>skimage.img_as_float64()</code> or <code>img_as_float</code>	Converts to floats from $0 \rightarrow 1$ with 64-bit precision

```
skimage.dtype_limits(hubble_gray)
```

```
(-1, 1)
```

```
hubble_gray_uint8 = skimage.img_as_ubyte(hubble_gray)
skimage.dtype_limits(hubble_gray_uint8)
```

```
(0, 255)
```

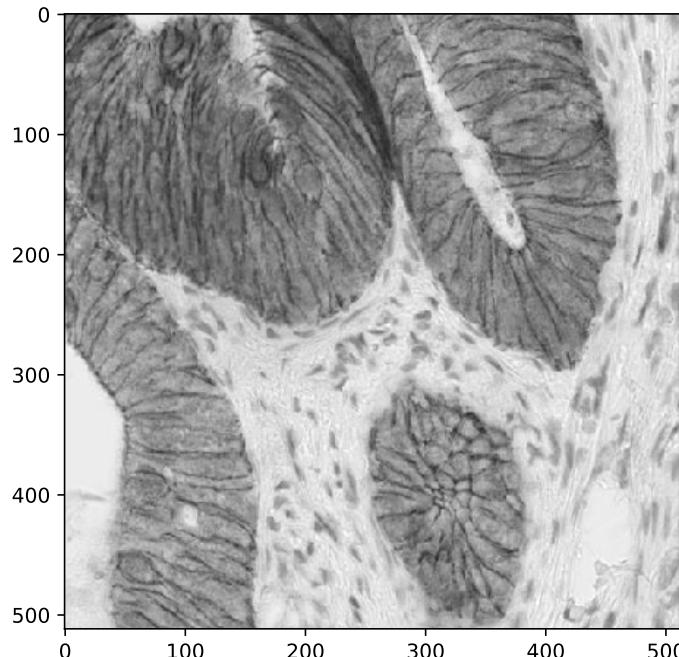
7.2.3 Image Contrast

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 (i.e., 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.

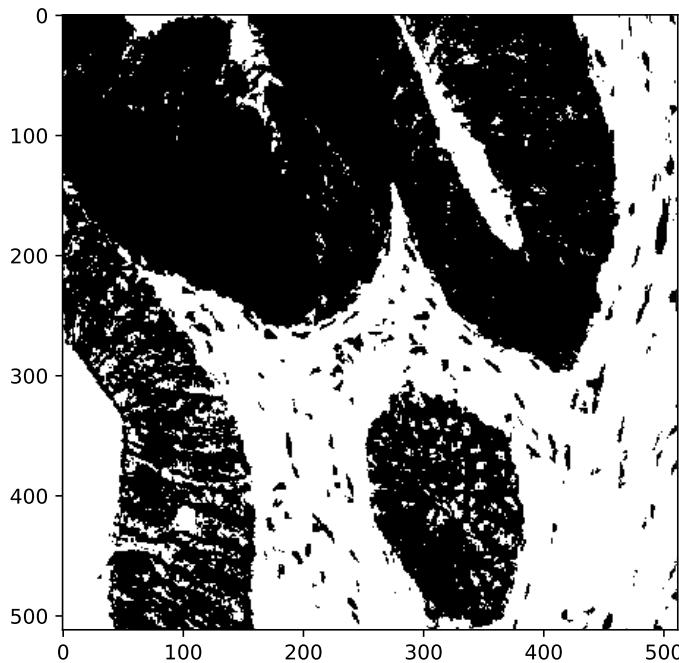
```
chem = data.immunohistochemistry()
chem_gray = color.rgb2gray(chem)
io.imshow(chem_gray);
```

```
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4458/5758897.py:3: FutureWarning: `imshow` i
io.imshow(chem_gray);
```



```
chem_bw = skimage.img_as_ubyte(chem_gray > 0.65)
# above generates a boolean encoding
io.imshow(chem_bw);
```

```
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4458/3375097781.py:3: FutureWarning: `imshow` i
io.imshow(chem_bw);
```



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, shown below in Table 4, in the filters module for automatically choosing a threshold. 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 4 Threshold Functions from the `filters` Module

Functions	Description*
<code>filters.threshold_isodata()</code>	Threshold value from ISODATA method
<code>filters.threshold_li()</code>	Threshold value from Li's minimum cross entropy method
<code>filters.threshold_local()</code>	Threshold mask (array) from local neighborhoods
<code>filters.threshold_mean()</code>	Threshold value from mean grayscale value
<code>filters.threshold_minimum()</code>	Threshold value from minimum method
<code>filters.threshold_niblack()</code>	Threshold mask (array) from the Niblack method
<code>filters.threshold_otsu()</code>	Threshold value from Otsu's method
<code>filters.threshold_sauvola()</code>	Threshold mask (array) from Sauvola method
<code>filters.threshold_triangle()</code>	Threshold value from triangle method
<code>filters.threshold_yen()</code>	Threshold value from Yen method

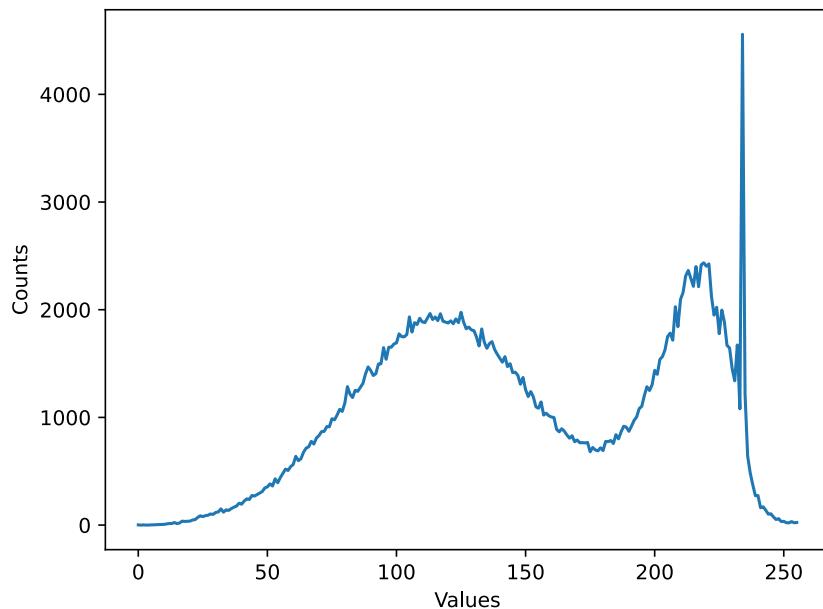
*Threshold value functions provides a single threshold value while threshold masks provides arrays of values the size of the image. They are used in the same fashion except that the latter provides a per-pixel threshold.

Below, we can see the Otsu filter being demonstrated.

```
from skimage import filters
threshold = filters.threshold_otsu(chem_gray)
chem_otsu = skimage.img_as_ubyte(chem_gray > threshold)
io.imshow(chem_otsu);
```

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.

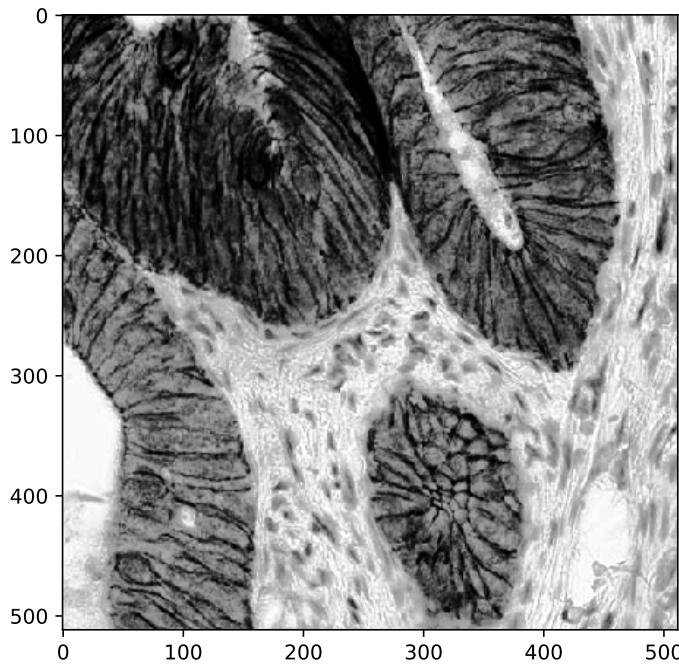
```
from skimage import exposure
hist = exposure.histogram(chem_gray)
plt.plot(hist[0])
plt.xlabel('Values')
plt.ylabel('Counts');
```



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.

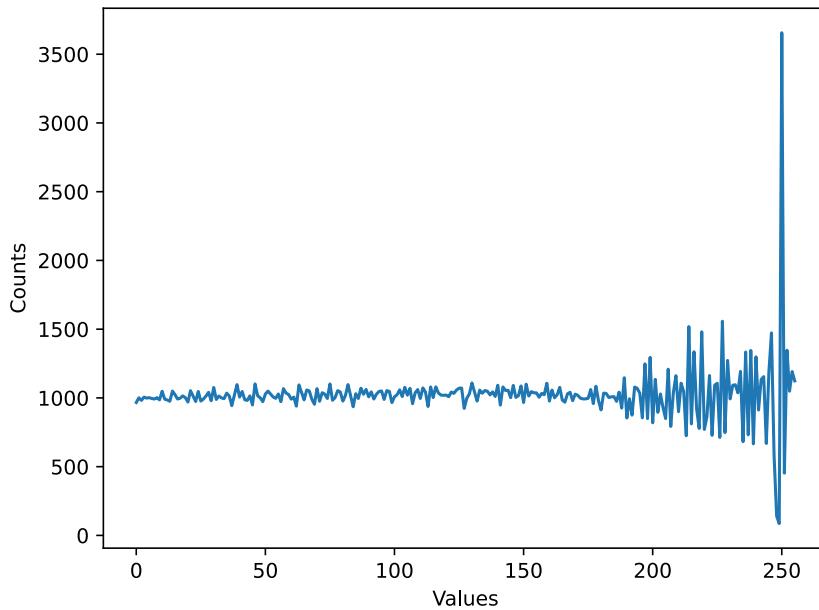
```
chem_eq = exposure.equalize_hist(chem_gray)
io.imshow(chem_eq);
```

```
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4458/1378964753.py:2: FutureWarning: `imshow`
```



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.

```
hist = exposure.histogram(chem_eq)
plt.plot(hist[0])
plt.xlabel('Values')
plt.ylabel('Counts');
```



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

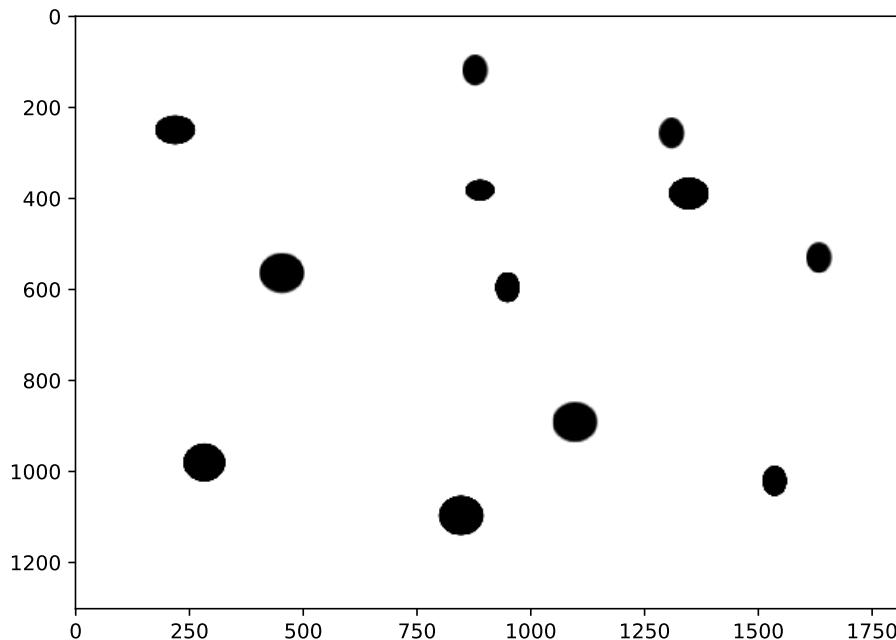
A classic problem that translates across many scientific fields is to count spots in a photograph. A biologist may need to quantify the number of bacteria colonies in a petrie dish over the course of an experiment, while an astronomer may want to count the number of stars in a large cluster. In chemistry, 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. 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=)
```

```
dots = io.imread('data/dots.png')
io.imshow(dots);
```

```
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4458/4244484902.py:2: FutureWarning: `imshow`
```



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 or using

the `color.rgb2gray()`.

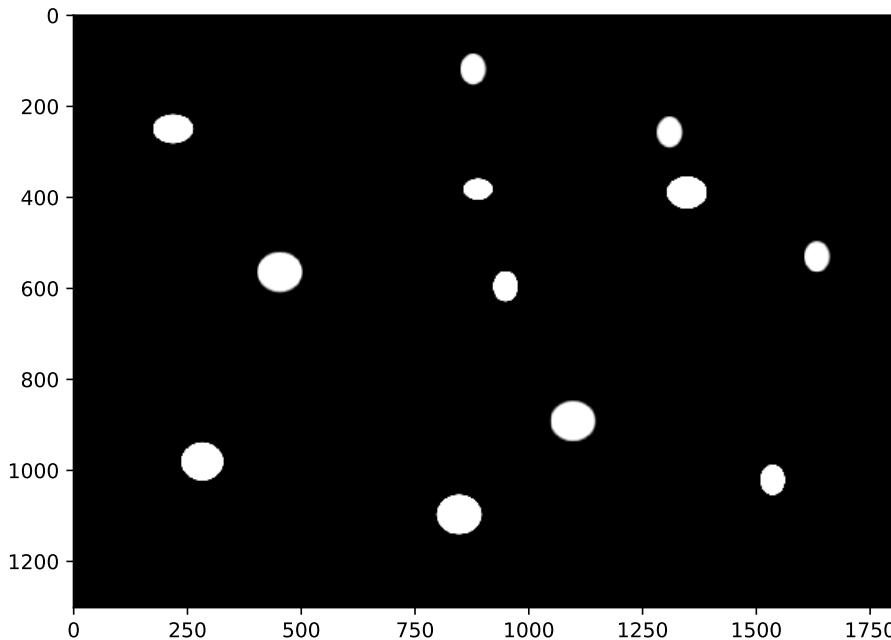
```
dots_inverted = color.rgb2gray(255 - dots)
```

or

```
dots_inverted = skimage.util.invert(dots)
```

```
dots_inverted = color.rgb2gray(255 - dots)
io.imshow(dots_inverted);
```

```
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4458/1622494045.py:2: FutureWarning: `imshow`
```



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.

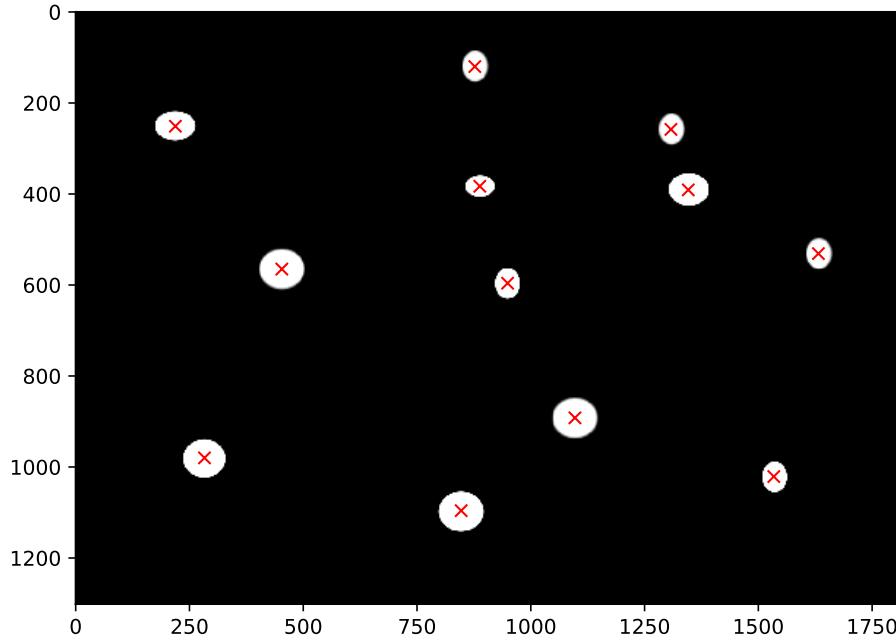
```
from skimage import feature
blobs = feature.blob_dog(dots_inverted, threshold=0.5)
blobs
```

```
array([[1096.,     847.,     26.8435456],
       [ 565.,     453.,     26.8435456],
       [ 892.,    1097.,     26.8435456],
       [ 980.,     283.,     26.8435456],
       [1021.,    1534.,    16.777216 ],
       [ 596.,     949.,    16.777216 ],
       [ 531.,    1632.,    16.777216 ],
       [ 120.,     877.,    16.777216 ],
       [ 258.,    1308.,    16.777216 ],
       [ 383.,     888.,    16.777216 ],
       [ 391.,    1346.,    26.8435456],
       [ 251.,     219.,    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 confirm that scikit-image found all the blobs, we can plot the coordinates on top of the image to see that they all line up. This is demonstrated below.

```
io.imshow(dots_inverted)
plt.plot(blobs[:,1], blobs[:,0], 'rx');
```

```
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4458/1109010740.py:1: FutureWarning: `imshow`
```



To find the number of spots, determine length of the array using the `len()` Python function or using .

```
len(blobs)
```

7.3.2 Entropy Analysis

The term *entropy* outside of the physical sciences is used to represent a quantification of disorder or irregularity. In image analysis, this disorder is the amount of pixel (brightness or color) variation within a region of the image. As you will see below, entropy is the highest near the boundaries and in noisy areas of a 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`).

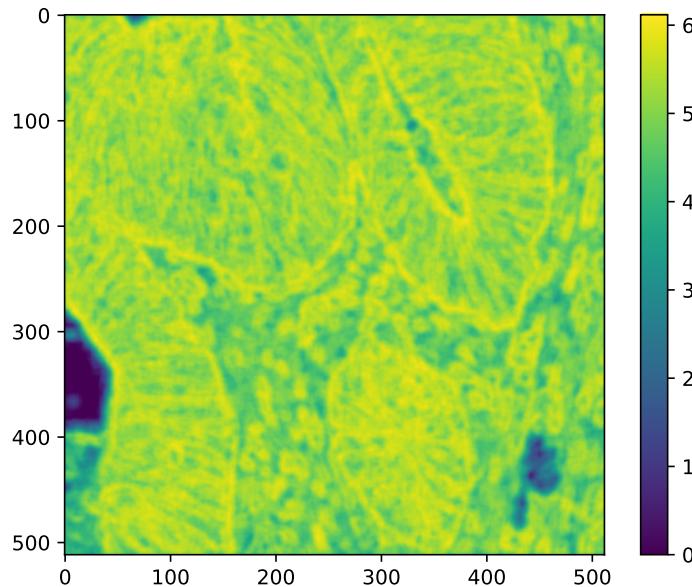
```
filters.rank.entropy(img, selem)
```

```
from skimage.morphology import disk
from skimage.filters.rank import entropy
selem = disk(5)
selem
```

```
array([[0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 1, 0],
       [0, 0, 0, 0, 0, 1, 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.

```
chem_gray_int = skimage.util.img_as_ubyte(chem_gray) #convert img to int
S = entropy(chem_gray_int, selem)
plt.imshow(S)
plt.colorbar();
```



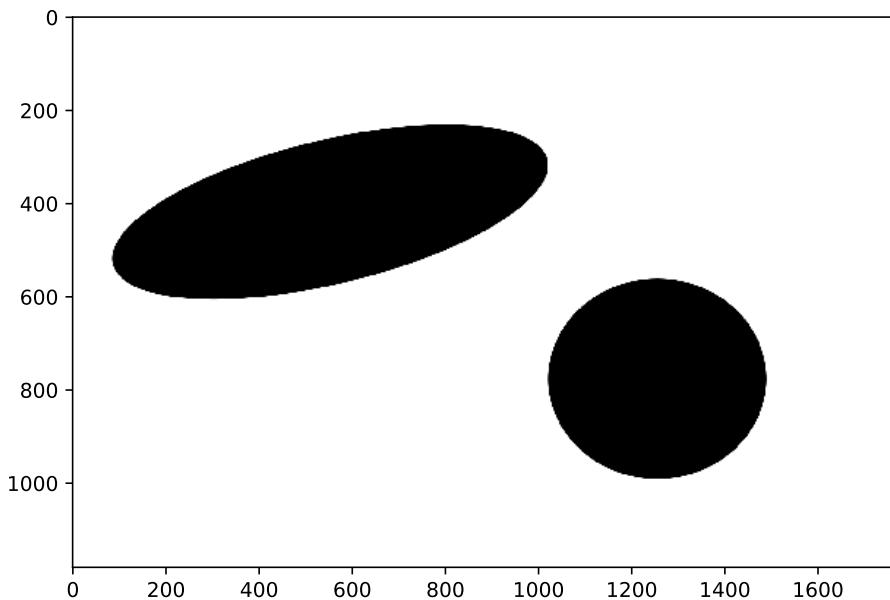
Examination of the image shows that there is an increase in entropy near the edges of the features in the image as expected. There are two regions (blue) that contain unusually low entropy. If you look back at the original image, these regions are comparatively homogeneous in color.

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.

```
ovals = io.imread('data/ovals.png')
io.imshow(ovals);
```

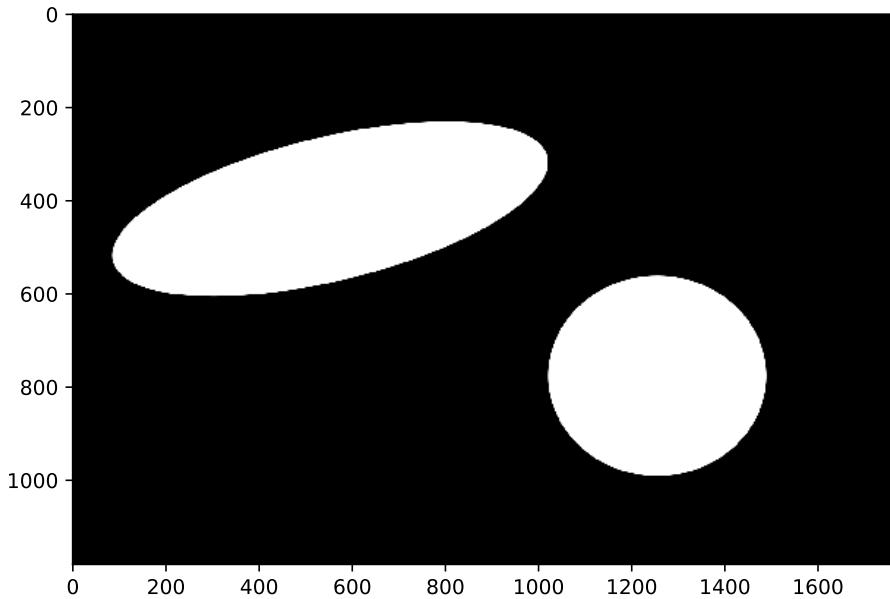
```
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4458/3848032356.py:2: FutureWarning: `imshow`
```



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. The following inverts the light and dark and also truncates the alpha channel from the RGBA image.

```
ovals_invert = color.rgb2gray(255 - ovals[:,:,:-1])
io.imshow(ovals_invert);
```

```
/var/folders/zy/7y6kpdbx6p1ffrp1vtxy3ttc0000gn/T/ipykernel_4458/2349859309.py:2: FutureWarning: `imshow`  
io.imshow(ovals_invert);
```



The `regionsprops()` 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 for the first object 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.

```
from skimage.measure import label, regionprops  
lbl = label(ovals_invert)  
props = regionprops(lbl)
```

```
props[0].eccentricity
```

```
0.9469273936534165
```

```
props[1].eccentricity
```

```
0.39666071911272044
```

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 source 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/> (free resource)
2. Pillow Documentation Page. <https://pillow.readthedocs.io/en/stable/> (free resource)
3. Tanimoto, S. L. *An Interdisciplinary Introduction to Image Processing: Pixels, Numbers, and Programs* MIT Press: Cambridge, MA, 2012.

Exercises

Complete the following exercises in a Jupyter notebook. Any data file(s) referred to in the problems can be found in the [data](#) folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

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. Generate a 100×100 pixel image containing random noise generated by a method from the [np.random](#) module such as [random\(\)](#) or [integers\(\)](#) (see [section 4.7](#)). Display the image in a Jupyter notebook along with a histogram of the pixel values. Hint: you will need to [flatten](#) the array before generating the histogram plot.

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 `util` 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?
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.
 - a) 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.
 - b) 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_img.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

Contents

- 8.1 Symbolic Mathematics
- 8.2 Algebra in SymPy
- 8.3 Matrices
- 8.4 Calculus
- 8.5 Mathematics in Python
- Further Reading
- Exercises

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 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.

```
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.

```
import math  
math.sqrt(2)
```

```
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.

```
sympy.sqrt(2)
```

$$\sqrt{2}$$

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.

```
sympy.sqrt(2).evalf()
```

$$1.4142135623731$$

One of the advantages of `evalf()` is that it also accepts a significant figures argument.

```
sympy.sqrt(2).evalf(30)
```

$$1.41421356237309504880168872421$$

```
sympy.pi.evalf(40)
```

$$3.141592653589793238462643383279502884197$$

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.

```
x, c, m = sympy.symbols('x c m')
```

```
x
```

$$x$$

There is no value attached to `x` as it is a symbol, so now it can be used to generate symbolic mathematical expressions.

```
E = m * c**2
```

```
E
```

$$c^2m$$

E**2

$$c^4m^2$$

Sympy can also be used to solve expressions for numerical values, and there are times when only certain ranges or types of numerical values make physical sense. For example, concentrations should only be nonnegative and real values. To constrain solutions of an expression to positive and nonnegative values, additional arguments known as predicates can be added to the `sympy.symbols()` function such as `nonnegative=True` or `real=True`.

```
y = sympy.symbols('y', real=True, nonnegative=True)
```

To see what constraints were placed on a variable, use the `assumptions()` function demonstrated below.

```
from sympy.core.assumptions import assumptions
assumptions(y)
```

```
{'commutative': True,
 'complex': True,
 'extended_negative': False,
 'extended_nonnegative': True,
 'extended_real': True,
 'finite': True,
 'hermitian': True,
 'imaginary': False,
 'infinite': False,
 'negative': False,
 'nonnegative': True,
 'real': True}
```

A selection of predicate arguments for the `sympy.symbols()` function are listed below in Table 1. This is not an exhaustive list, but a complete list can be found on the [SymPy website](#) under "Predicates."

Table 1 Predicates for `sympy.symbols()`

positive	negative	imaginary	real	complex
finite	infinite	nonzero	zero	integer
rational	irrational	even	prime	composite

8.1.2 Pretty Printing

Depending upon settings and version of SymPy, the output may look like Python equations which are not always the easiest to read. If so, you can turn on *pretty printing*, shown below, which will instruct SymPy to render the expressions in more traditional mathematical representations that you might see in a math textbook. More recent versions of SymPy

make this unnecessary, however, as it generates more traditional mathematical representations by default.

```
from sympy import init_printing
sympy.init_printing()
```

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.

Table 2 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>

It is important to note that any mathematical function operating on a symbol needs to be from the SymPy library. For example, using a `math.cos()` function from the `math` Python module will result in an error.

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 3 which includes polynomial expansion and factoring, expression simplification, and solving equations. The subsequent sections demonstrate each of these.

Table 3 Common Algebraic Methods

Method	Description
<code>sympy.expand()</code>	Expand polynomials
<code>sympy.factor()</code>	Factors polynomials
<code>sympy.simplify()</code>	Simplifies the expression
<code>sympy.solve()</code>	Equates the expression to zero and solves for the requested variable
<code>sympy.subs()</code>	Substitutes a variable for a value, expression, or another 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.

```
expr = (x - 1) * (3 * x + 2)
```

```
sympy.expand(expr)
```

$$3x^2 - x - 2$$

The process can be reversed by factoring the polynomial.

```
sympy.factor(3 * x**2 - x - 2)
```

$$(x - 1)(3x + 2)$$

8.2.2 Simplification

SymPy may not always return a mathematical expression into the simplest form. Below is an expression with a simpler form, and if we feed this into SymPy, it is not automatically simplified.

```
3 * x**2 - 4 * x - 15 / (x - 3)
```

$$3x^2 - 4x - \frac{15}{x - 3}$$

However, if we instruct SymPy to simplify the expression using the `simplify()` method, it will make a best attempt at finding a simpler form.

```
sympy.simplify((3 * x**2 - 4 * x - 15) / (x - 3))
```

$$3x + 5$$

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 $x^2 + 1.4x - 5.76 = 0$.

```
sympy.solve(x**2 + 1.4 * x - 5.76)
```

```
[-3.20000000000000, 1.80000000000000]
```

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.

	2 NH ₃	⇒	3 H ₂ (g)	+	N ₂ (g)
Initial	0.60 M		0.60 M		0.00 M
Change, Δ	-2x		+3x		+x
Equilibrium	0.60 - 2x		0.60 + 3x		x

$$K_c = 3.44 = \frac{[N_2][H_2]^3}{[NH_3]^2} = \frac{(x)(0.60 + 3x)^3}{(0.60 - 2x)^2}$$

To expand the right portion of the equation, we can use the `expand()` method. Notice that the variable `x` has been constrained below to real (`real=True`) and nonnegative (`nonnegative=True`) values here. This is because in this example, `x` is one of the equilibrium concentrations, so imaginary and negative values would make no physical sense. These constraints may not be appropriate for other examples.

```
x = sympy.symbols('x', real=True, nonnegative=True)
expr = (x) * (0.60 + 3 * x)**3 / (0.60 - 2 * x)**2
```

```
sympy.expand(expr)
```

$$\frac{27x^4}{4x^2 - 2.4x + 0.36} + \frac{16.2x^3}{4x^2 - 2.4x + 0.36} + \frac{3.24x^2}{4x^2 - 2.4x + 0.36} + \frac{0.216x}{4x^2 - 2.4x + 0.36}$$

This is probably not what you were expecting or hoping for. The polynomial has been expanded, but the result is still a fraction. We can instruct SymPy to simplify the results.

```
sympy.simplify(sympy.expand(expr))
```

$$\frac{x (27x^3 + 16.2x^2 + 3.24x + 0.216)}{4x^2 - 2.4x + 0.36}$$

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.

```
sympy.solve(expr - 3.44)
```

```
[0.170006841512893]
```

If the variable `x` was not constrained to real and nonnegative values, a fourth-order polynomial would return four solutions, with only one making physical sense. Because we did constrain `x`, the `solve()` function conveniently only returns 0.17.

8.2.5 Substitutions

Another common algebraic operation is the substitution of one variable in an expression for another variable, expression, or value. This is accomplished in SymPy using the `sympy.subs()` function which requires two pieces of information – the variable being replaced (`x_old`) and the new variable, expression, or value (`x_new`).

```
sympy.subs(x_old, x_new)
```

As an example, let's determine the composition of a mixture of two enantiomers based on the net optical rotation of this mixture. The net rotation of a mixture, $[\alpha]_{mix}$, of two enantiomers d and l is described below as the linear combination of rotations of each enantiomer where d and l are the mole fractions and $[\alpha]_d$ and $[\alpha]_l$ are the specific rotations of each enantiomer.

$$[\alpha]_{mix} = d[\alpha]_d + l[\alpha]_l$$

If we have a mixture where the net rotation is $+8.3^\circ$ and the d and l enantiomers have specific rotations of $+32.4^\circ$ and -32.4° , respectively, we can insert these values into the above equation to get the below result.

$$+8.3^\circ = d(+32.4^\circ) + l(-32.4^\circ)$$

We now have one equation with two unknowns, d and l . To solve this, we need a second equation which we can generate by recognizing that the sum of the fractions equals 1 just as the sum of percentages total to 100%.

$$d + l = 1$$

We rearrange the above equation to $d = 1 - l$ and now need to substitute this expression for `d` in the first equation. We can let SymPy perform this substitution.

```
d, l = sympy.symbols('d, l')
net = (d * 32.4 + l * -32.4) - 8.3
net_new = net.subs(d, 1 - l)
net_new
```

$$24.1 - 64.8l$$

We can then solve this expression for l using the `sympy.solve()` function being that it equals zero in the current form.

```
sympy.solve(net_new)
```

```
[0.371913580246914]
```

The `sympy.subs()` function can also substitute variables for numerical values. If we want to see the net rotation is $l = 0.6$ and $d = 0.4$, we can run the following.

```
net.subs([[l, 0.6], [d, 0.4]])
```

-14.78

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. 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.

```
import numpy as np  
mat = np.matrix([[1, 8], [3, 2]])
```

```
mat
```

```
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 dot product. For these operations, we have the following methods provided by NumPy's `linalg` module, Table 4, which are demonstrated in the following sections.

Table 4 Common `np.linalg` Methods

Method	Description
<code>np.linalg.dot()</code>	Calculates the dot product
<code>np.linalg.inv()</code>	Returns the inverse of an array (if it exists)
<code>np.linalg.pinv()</code>	Returns the Moore-Penrose <u>pseudoinverse</u> of an array
<code>np.linalg.det()</code>	Returns the determinant of an array
<code>np.linalg.solve()</code>	Solves a system of linear equations
<code>np.linalg.lstsq()</code>	Returns <u>approximate solution</u> to 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 .

$$6x + 10y - 5z = 21$$

$$2x + 7y + z = 13$$

$$-10x - 11y + 11z = -21$$

These equations can be rewritten in matrix or array form as follows with the left matrix holding the coefficients.

$$\begin{bmatrix} 6 & 10 & -5 \\ 2 & 7 & 1 \\ -10 & -11 & 11 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 21 \\ 13 \\ -21 \end{bmatrix}$$

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 , M^{-1} . Anything multiplied by its inverse is the identity, so $M^{-1} \cdot M$ is the identity matrix and can be ignored.

$$M^{-1} \cdot M \cdot X = M^{-1} \cdot y$$

$$X = M^{-1} \cdot y$$

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.

```
M = np.array([[6, 10, -5],  
             [-2, 7, 1],  
             [-10, -11, 11]])  
y = np.array([21, 13, -21])
```

```
np.linalg.inv(M).dot(y)
```

```
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 Beer's law is written below where A is absorbance (unitless), b is path length (cm), C is concentration (M), and ϵ is the molar absorptivity ($\text{cm}^{-1}\text{M}^{-1}$). The latter value is analyte dependent.

$$A = \epsilon b C$$

For a path length of 1.0 cm, which is quite common, the equation simplifies down to:

$$A = \epsilon C$$

When there are three analytes, x , y , and z , the absorption of light at a given wavelength equals the sum of the individual absorptions.

$$A = \epsilon_x C_x + \epsilon_y C_y + \epsilon_z C_z$$

If we measure the absorbance of a three analyte solution at three different wavelengths (λ), we get the following three equations.

$$A_{\lambda 1} = \epsilon_{x \lambda 1} C_x + \epsilon_{y \lambda 1} C_y + \epsilon_{z \lambda 1} C_z$$

$$A_{\lambda 2} = \epsilon_{x \lambda 2} C_x + \epsilon_{y \lambda 2} C_y + \epsilon_{z \lambda 2} C_z$$

$$A_{\lambda 3} = \epsilon_{x \lambda 3} C_x + \epsilon_{y \lambda 3} C_y + \epsilon_{z \lambda 3} C_z$$

As long as we know the molar absorptivity of each analyte at each wavelength collected 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.

$$\begin{bmatrix} \epsilon_{x \lambda 1} & \epsilon_{y \lambda 1} & \epsilon_{z \lambda 1} \\ \epsilon_{x \lambda 2} & \epsilon_{y \lambda 2} & \epsilon_{z \lambda 2} \\ \epsilon_{x \lambda 3} & \epsilon_{y \lambda 3} & \epsilon_{z \lambda 3} \end{bmatrix} \cdot \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix} = \begin{bmatrix} A_{\lambda 1} \\ A_{\lambda 2} \\ A_{\lambda 3} \end{bmatrix}$$

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.

```
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)
```

```
array([0.078 , 0.023 , 0.0066])
```

The concentrations are $C_x = 0.078$ M, $C_y = 0.023$ M, and $C_z = 0.0066$ M.

Alternatively, there is a `np.linalg.solve()` function that accomplishes the same calculation in a single function call. This function requires two pieces of information: the coefficient matrix and the dependant variable matrix. In our example, these are `E` and `A`, respectively.

```
np.linalg.solve(E, A)
```

```
array([0.078 , 0.023 , 0.0066])
```

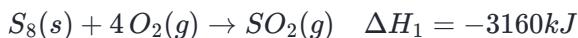
If you perform either the above calculations on other data and receive a `LinAlgError: Singular matrix` error, this means that the coefficient matrix does not have an inverse and cannot be solved by these methods. One possible reason is that the coefficient matrix is not square - a requirement for obtaining an inverse. Here are two possible solutions to working around this issue.

1. Substitute the `np.linalg.inv()` function with the Moore-Penrose pseudoinverse function `np.linalg.pinv()`. This versatile function can work with non-square matrices.
2. Substitute the `np.linalg.lstsq()` for the `np.linalg.solve()` function. The former can find approximate solutions when exact solutions do not exist or when the coefficient matrix is not square. This is not uncommon when dealing with linear fitting because not all data points may fall perfectly on the line of best fit or the number of data points does not equal the number of independent variables.

As an example inspired by [J. Chem. Educ. 2000, 77, 185-187](#), let's calculate the enthalpy of the following reaction



knowing the enthalpy of the following two subreactions.



Using Hess's law, we need to multiply the subreactions 1 and 2 by coefficients and add them together to generate the

overall net reaction. Remember that reversing a reaction results in reversing the sign of reaction enthalpy, so it's the same as multiplying by -1. We can represent this calculation using matrices shown below where r_1 and r_2 are the coefficients for each subreaction. The values in each row of the first matrix are the number of SO_2 , SO_3 , O_2 , and S_8 molecules, respectively, in the two subreactions, and the numbers in the last matrix are the numbers of the same molecules in the net reaction. The gray molecular formulas in the equation below are not part of the equation but rather are simply labels for clarity. You may notice that the coefficients in the balance equations on the reactant side are negative while products are positive. This allows us to keep track of the side they are on.

$$\begin{array}{l} \text{SO}_2 \rightarrow \begin{bmatrix} 0 & -2 \end{bmatrix} \\ \text{SO}_3 \rightarrow \begin{bmatrix} 8 & 2 \end{bmatrix} \\ \text{O}_2 \rightarrow \begin{bmatrix} -12 & -1 \end{bmatrix} \\ \text{S}_8 \rightarrow \begin{bmatrix} -1 & 0 \end{bmatrix} \end{array} \cdot \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} 8 \\ 0 \\ -8 \\ -1 \end{bmatrix}$$

If the three matrices are called A, R, and Y, respectively, we can rewrite the above calculation as follows.

$$A \cdot R = Y$$

When solving for R in the past, we simply multiplied both sides by A^{-1} like below.

$$A^{-1} \cdot A \cdot R = A^{-1} \cdot Y$$

$$R = A^{-1} \cdot Y$$

The problem we face is that matrix A is not square and thus the matrix inverse cannot be calculated. Instead, we can use the Moore-Penrose pseudoinverse in place of the regular inverse as demonstrated below.

```
A = np.array([[0, -2],
             [8, 2],
             [-12, -1],
             [-1, 0]])
Y = np.array([8, 0, -8, -1])
```

```
R = np.linalg.pinv(A).dot(Y)
R
```

```
array([ 1., -4.])
```

This means we need to multiply the first subreaction by 1 and the second subreaction by -4 (i.e., reverse it and quadruple everything).

Alternatively, we can use the `np.linalg.lstsq()` similarly to how we use the `np.linalg.solve()` function. Set the keyword argument `rcond=None` to avoid an error.

```
np.linalg.lstsq(A, Y, rcond=None)
```

```
(array([ 1., -4.]),
 array([1.85454492e-31]),
 np.int32(2),
 array([14.58924398,  2.27023349]))
```

For the final step of our calculation, we need to multiply the values r_1 and r_1 by the enthalpy values of the subreactions and add them together.

```
dH_sub = np.array([-3160, -196])
dH = R.dot(dH_sub)
dH
np.float64(-2375.999999999995)
```

This means that the enthalpy of the overall net reaction is -2376 kJ.

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 θ , and the rightmost matrix y .

$$X \cdot \theta = y$$

Ultimately, we are looking for the values of m and b , so we need to solve for matrix θ . This can be accomplished through optimization algorithms (section [14.2](#)), 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 .

$$(X^T \cdot X)^{-1} \cdot X^T \cdot y = \theta$$

As an example, below is a table of synthetic data for copper cuprizone absorbances at various concentrations at 591 nm. We can use a linear fit to create a calibration curve from this data.

Table 5 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

```
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])
```

```
y = A
X = np.vstack((C, np.ones(6))).T
X
```

```
array([[1.0e-06, 1.0e+00],
       [3.0e-06, 1.0e+00],
       [6.0e-06, 1.0e+00],
       [1.5e-05, 1.0e+00],
       [2.5e-05, 1.0e+00],
       [3.5e-05, 1.0e+00]])
```

For the sake of readability, the calculation using the normal equation has been split in half as shown below.

$$u = (X^T \cdot X)^{-1}$$

$$v = X^T \cdot y$$

$$u \cdot v = \theta$$

```
u = np.linalg.inv(X.T.dot(X))
v = X.T.dot(y)
theta = u.dot(v)
```

```
theta
```

```
array([ 1.55886203e+04, -5.45355390e-06])
```

A plot of the linear regression and the data points is shown below, 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 -5.45×10^{-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.

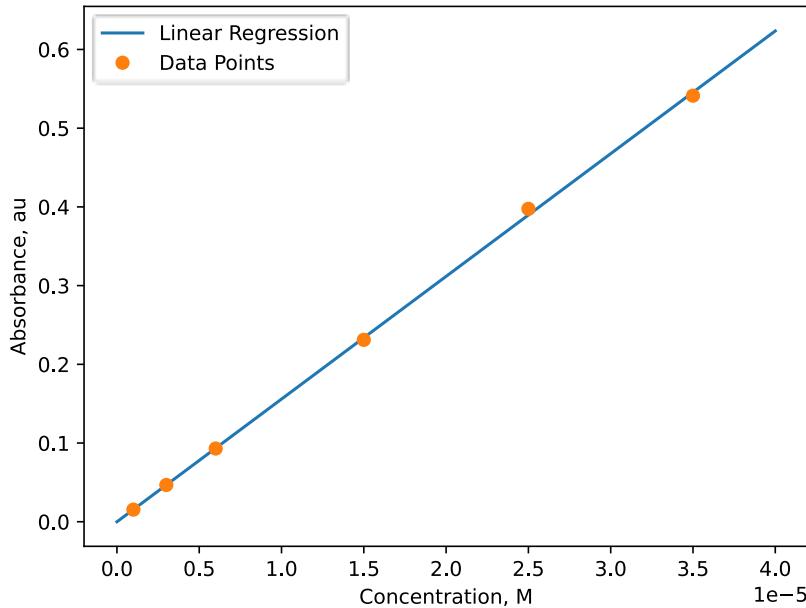
```

import matplotlib.pyplot as plt

x = np.linspace(0, 4e-5, 10)
plt.plot(x, 1.55886e4 * x - 5.45355e-6, '-', label='Linear Regression')
plt.plot(C, A, 'o', label='Data Points')

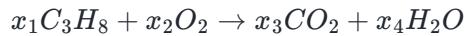
plt.legend()
plt.ticklabel_format(style='sci', axis='x', scilimits=(0, 0))
plt.xlabel('Concentration, M')
plt.ylabel('Absorbance, au');

```



8.3.4 Balancing Chemical Equations

Matrices can also be used to balance chemical equations as shown below where x_1 through x_4 are the coefficients for the balanced chemical equation.



We can then describe the number of carbon, hydrogen, and oxygen atoms in each compound using 3×1 matrices

$$\begin{bmatrix} C \\ H \\ O \end{bmatrix}$$

as shown below.

$$x_1 \begin{bmatrix} 3 \\ 8 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix} \rightarrow x_3 \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} + x_4 \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix}$$

Because the number of carbons, hydrogens, and oxygens should be the same on both sides of the balanced chemical equation, if we subtract the products from the reactants, we should get zero.

$$x_1 \begin{bmatrix} 3 \\ 8 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix} - x_3 \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} - x_4 \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

One potential issue with this set of linear equations is that making all the x variables zero is a valid solution, so to avoid this solution, we will set one of the x variables to one. Remember that a balanced chemical equation is about the appropriate ratio between the reactants and products, so setting a single coefficient to one can still generate a balanced equation. The one issue is that the coefficients generated by the software may not be integers, but this can be fixed by multiplying the fractions to get whole numbers as a final step demonstrated below.

Here we have set $x_4 = 1$.

$$x_1 \begin{bmatrix} 3 \\ 8 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix} - x_3 \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} - (1) \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Now we move the last term to the right side.

$$x_1 \begin{bmatrix} 3 \\ 8 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix} - x_3 \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix}$$

These matrices can now be merged into one larger matrix. The left matrix below will be called **M** and the right matrix below is called **b**.

$$\begin{bmatrix} 3 & 0 & -1 \\ 8 & 0 & 0 \\ 0 & 2 & -2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix}$$

We can then solve for the x values to get our coefficients using the `np.linalg.solve()` function as demonstrated below.

```
M = np.array([[3, 0, -1], [8, 0, 0], [0, 2, -2]])
b = np.array([0, 2, 1]).T

sol = np.linalg.solve(M, b)
sol
```

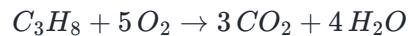
```
array([0.25, 1.25, 0.75])
```

This means that $x_1=0.25$, $x_2=1.25$, and $x_3=0.75$. We can append x_4 below and then multiply all the values by the same number to generate all integers.

```
sol = np.append(sol, 1)
sol * 4
```

```
array([1., 5., 3., 4.])
```

This means that the integer coefficients for the balanced chemical equation are $x_1=1$, $x_2=5$, $x_3=3$, and $x_4=4$.



8.3.5 Eigenvalues and Eigenvectors

This section covers using `np.linalg` to calculate *eigenvalues* and *eigenvectors* which is useful in quantum mechanics among other applications. This topic will not be utilized later in this book, so feel free to skip over this section if you have no interest in this topic.

For a square matrix A , there can exist a scalar λ and vector V that satisfy the following equation.

$$AV = \lambda V$$

The vector and scalar are known as the eigenvector and eigenvalue, respectively, and there may be more than one solution for any given matrix A .

The `np.linalg` module includes a function `np.linalg.eig()` that returns the eigenvalue(s) and eigenvector(s) for a given square matrix in this order

```
np.linalg.eig(matrix)
```

As an example, we can determine the eigenvalues and eigenvector for the following matrix.

$$A = \begin{bmatrix} 3 & 1 \\ 4 & 3 \end{bmatrix}$$

```
A = np.array([[3, 1], [4, 3]])
np.linalg.eig(A)
```

```
EigResult(eigenvalues=array([5., 1.]), eigenvectors=array([[ 0.4472136 , -0.4472136 ],
   [ 0.89442719,  0.89442719]]))
```

The first array contains the two eigenvalues while the second matrix contains the two eigenvector solutions.

Not every matrix has eigenvalues or eigenvectors. In the case of the following 90° rotation matrix, the solution generated includes j values which is Python's notation used for imaginary and complex numbers.

$$A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

```
A = np.array([[0, -1], [1, 0]])
np.linalg.eig(A)
```

```
EigResult(eigenvalues=array([0.+1.j, 0.-1.j]), eigenvectors=array([[0.70710678+0.j           , 0.70710678-0.j], [0.          -0.70710678j, 0.          +0.70710678j]]))
```

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, we will be working with the radial density functions (ψ) for hydrogen atomic orbitals. The squares of these functions (ψ^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 $Z = 1$.

```
from sympy.physics.hydrogen import R_nl
```

```
r = sympy.symbols('r')
R_21 = R_nl(2, 1, r, Z=1)
```

```
R_21
```

$$\frac{\sqrt{6}re^{-\frac{r}{2}}}{12}$$

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.

```
f = sympy.lambdify(r, R_21, modules='numpy')
```

This function is now callable by providing a value for r .

```
f(0.5)
```

```
np.float64(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.

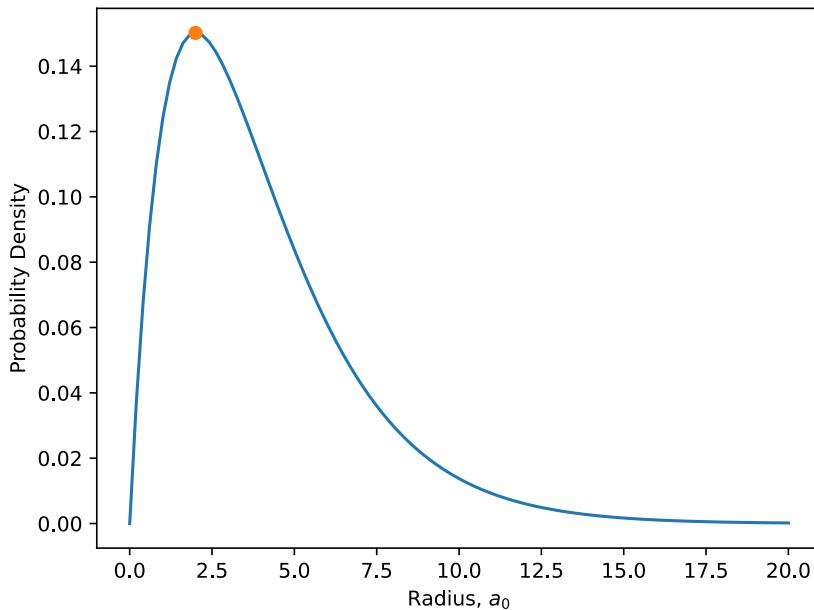
```
dR_21 = sympy.diff(R_21, r, 1)
dR_21
```

$$-\frac{\sqrt{6}re^{-\frac{r}{2}}}{24} + \frac{\sqrt{6}e^{-\frac{r}{2}}}{12}$$

```
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.

```
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');
```



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., ψ^2r^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.

```
sympy.integrate(R_21**2 * r**2, (r, 0, mx)).evalf()
```

```
0.052653017343711
```

There is a 5.27% probability finding an electron between the nucleus and the radius of maximum probability. This is probably may be a bit surprising, but examination of the radial density plot reveals that the radius of maximum probability is quite close to the nucleus with a significant amount of density beyond the maximum radius. Let's see the probability of finding an electron between 0 and 10 Bohrs from the nucleus.

```
sympy.integrate(R_21**2 * r**2, (r, 0, 10)).evalf()
```

```
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 ^1H 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*.

```
nmr = np.genfromtxt('data/Ar_NMR.csv', delimiter=',')  
nmr
```

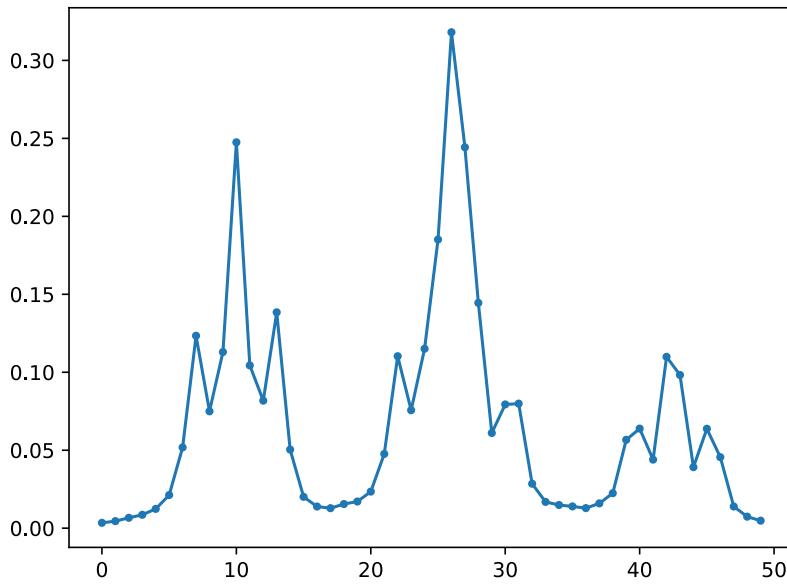
```

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],
       [1.0000000e+01, 2.47482900e-01],
       [1.1000000e+01, 1.04401566e-01],
       [1.2000000e+01, 8.17907750e-02],
       [1.3000000e+01, 1.38453960e-01],
       [1.4000000e+01, 5.04080100e-02],
       [1.5000000e+01, 2.00982630e-02],
       [1.6000000e+01, 1.38752850e-02],
       [1.7000000e+01, 1.28241135e-02],
       [1.8000000e+01, 1.54948140e-02],
       [1.9000000e+01, 1.70803180e-02],
       [2.0000000e+01, 2.34651420e-02],
       [2.1000000e+01, 4.76330930e-02],
       [2.2000000e+01, 1.10299855e-01],
       [2.3000000e+01, 7.56612400e-02],
       [2.4000000e+01, 1.15097150e-01],
       [2.5000000e+01, 1.85112450e-01],
       [2.6000000e+01, 3.18055840e-01],
       [2.7000000e+01, 2.44278220e-01],
       [2.8000000e+01, 1.44489410e-01],
       [2.9000000e+01, 6.10540140e-02],
       [3.0000000e+01, 7.93569160e-02],
       [3.1000000e+01, 7.98874400e-02],
       [3.2000000e+01, 2.85217260e-02],
       [3.3000000e+01, 1.68548040e-02],
       [3.4000000e+01, 1.48743290e-02],
       [3.5000000e+01, 1.39662160e-02],
       [3.6000000e+01, 1.28568120e-02],
       [3.7000000e+01, 1.59042850e-02],
       [3.8000000e+01, 2.24487820e-02],
       [3.9000000e+01, 5.66768200e-02],
       [4.0000000e+01, 6.38733950e-02],
       [4.1000000e+01, 4.39581830e-02],
       [4.2000000e+01, 1.09901360e-01],
       [4.3000000e+01, 9.82578100e-02],
       [4.4000000e+01, 3.91401280e-02],
       [4.5000000e+01, 6.37550060e-02],
       [4.6000000e+01, 4.55453170e-02],
       [4.7000000e+01, 1.38955860e-02],
       [4.8000000e+01, 7.40419900e-03],
       [4.9000000e+01, 4.81957300e-03]])

```

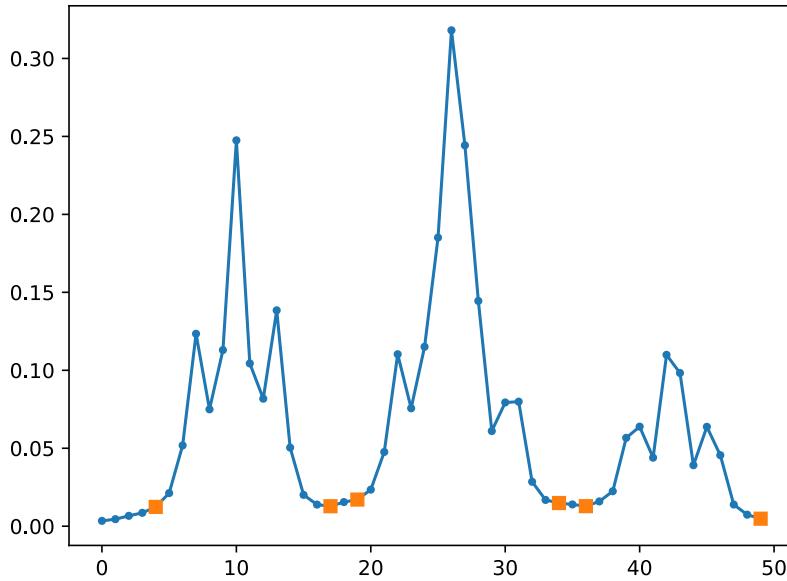
The imported data are stored in an array where the first column contains the index values and the second column contains the amplitudes.

```
plt.plot(nmr[:,0], nmr[:,1], '.-');
```



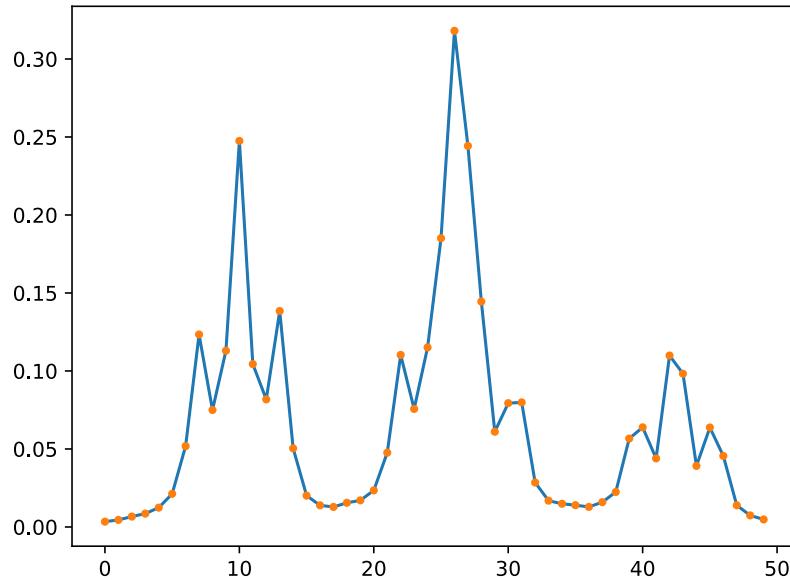
Above is a plot of the peaks with respect to the index values (not ppm). To integrate under each of the triplet peaks, first we need the index values for the edges of each peak. Below is a list, `i`, that provides reasonable boundaries, and a plot is below with these edges marked in orange squares.

```
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')
```



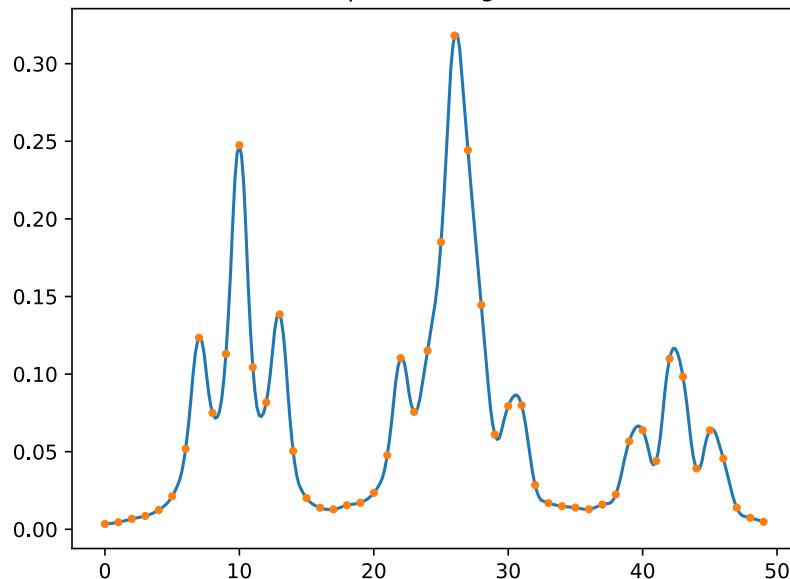
Integrations under sampled data do not include the values between data points, so these regions are estimated based on assumptions. The `trapezoid()` function assumes that any data point between known points lies directly between the known data points (i.e., linear interpolation) as shown below by the blue lines.

Trapezoidal Integration



Alternatively, the `simpson()` function uses the *Simpson's rule* which estimates the data between known points using quadratic interpolation shown below.

Simpson's Integration



Below, both the trapezoidal and Simpson's methods are demonstrated. Note that the `trapezoid(x, y)` function takes both the x and y values as required, positional arguments while `simpson(y, x=)` only requires the y data but will optionally accept the x data as a keyword argument.

```
from scipy.integrate import trapezoid, simpson

# trapezoid method
for peak in i:
    x = nmr[peak[0]:peak[1], 0]
    y = nmr[peak[0]:peak[1], 1]
    print(trapezoid(y, x))
```

```
1.0401881535  
1.529880057  
0.5834871775
```

```
# simpson method (note the different arguments)  
for peak in i:  
    x = nmr[peak[0]:peak[1], 0]  
    y = nmr[peak[0]:peak[1], 1]  
    print(simpson(y, x=x))
```

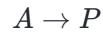
```
1.0405229256666666  
1.5661107306666666  
0.5839565783333334
```

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.



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.

```
def rate_1st(A, t):  
    return -k * A
```

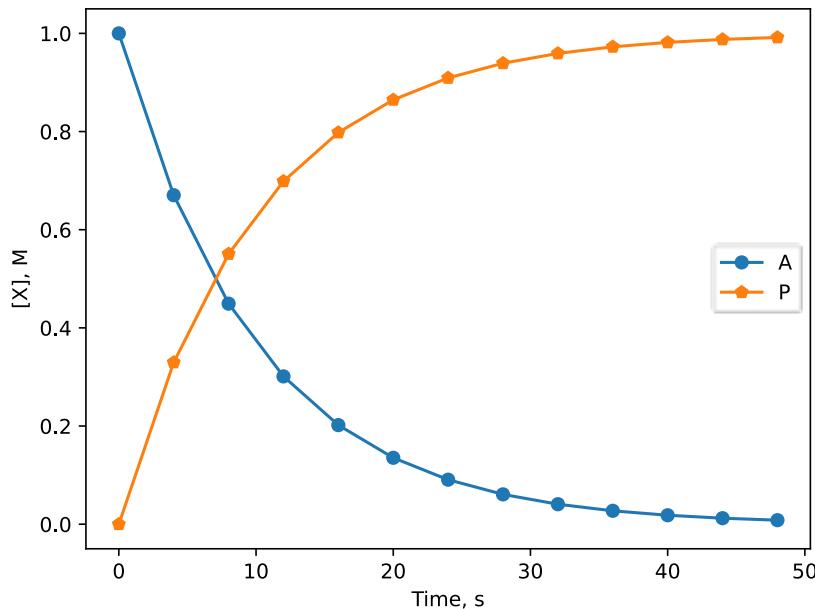
The function should take the dependent variable(s) as the first positional argument and the independent variable as the second positional argument. In this example, `A` is the dependent variable and time, `t`, is the independent variable. If there are multiple dependent variables, they need to be provided inside a composite object like a list or tuple which can

be unpacked through indexing or tuple unpacking once inside the function. 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.

```
from scipy.integrate import odeint
t = np.arange(0, 50, 4) # time(seconds)
A0 = 1 # starting concentration (molarity)
k = 0.1 # rate constant in 1/s
A_t = odeint(rate_1st, A0, t)
P_t = A0 - A_t # concentration of product
```

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.

```
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();
```



This approach to kinetic simulations can be adapted to even more complex reactions which are demonstrated in [section 9.1.4](#).

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> (free resource)
2. SciPy and NumPy Documentation Pages. <https://docs.scipy.org/doc/> (free resource)

Exercises

Complete the following exercises in a Jupyter notebook using the SymPy and SciPy libraries. Any data file(s) referred to in the problems can be found in the [data](#) folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

1. Factor the following polynomial using SymPy: $x^2 + x - 6$
2. Simplify the following mathematical expression using SymPy: $z = 3x + x^2 + 2xy$
3. Expand the following expression using SymPy: $(x-2)(x+5)(x)$
4. A 53.2 g block of lead ($C_p = 0.128 \text{ J/g}\cdot\text{C}$) at 128 °C is dropped into a 238.1 g water ($C_p = 4.18 \text{ J/g}\cdot\text{C}$) 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 $q = mCp\Delta T$.
5. The following equation relates the ΔG 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. ($R = 8.314 \text{ J/mol}\cdot\text{K}$)

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 (θ) 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 (CO_3^{2-}) measured in angstroms, rotate them 90° clockwise. Plot the initial and rotated points in different colors to show that it worked.

$$C : (2.00, 2.00) \quad O1 : (2.00, 3.28) \quad O2 : (0.27, 1.50) \quad O3 : (3.73, 1.50)$$

- 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.

a) Calculate the center of mass

b) Subtract the center of mass from all points to shift the cluster to the origin.

c) Rotate the cluster of points.

d) Add the center of mass back the cluster the shift the points back.

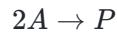
8. The following is the equation for the work performed by a reversible, isothermal (i.e., constant T) expansion of a piston by a fixed quantity of gas.

$$w = \int_{v_i}^{v_f} -nRT \frac{1}{V} dV$$

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.



10. Below are the transformation matrices for an S_4 and C_2 operation used in group theory. Show that two S_4 operations equal one C_2 operation by multiplying two S_4 operations together. That is, show that $S_4S_4 = C_2$.

$$S_4 = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad C_2 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

11. Using dot product math, write your own linear regression function that accepts the x and y coordinates of data points as separate arrays and returns the slope and intercept of a line of best fit.

Chapter 9: Simulations

Contents

- 9.1 Deterministic Simulations
- 9.2 Stochastic Simulations
- Further Reading
- Exercises

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 to 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 here for simplicity and accessibility.

The simulations in this chapter assume the following imports from NumPy, SciPy, and matplotlib.

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

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 a large number of molecules can make the randomness conform to predictable statistical behavior. 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 ^1H 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 ^1H NMR spectra deterministically by splitting all peaks into 1:1 doublets for every neighboring proton.

A recursive function 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.
    '''
    # check validity of input values
    if not isinstance(peaks, list):
        peaks = list([peaks])
    if not isinstance(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 -= 1

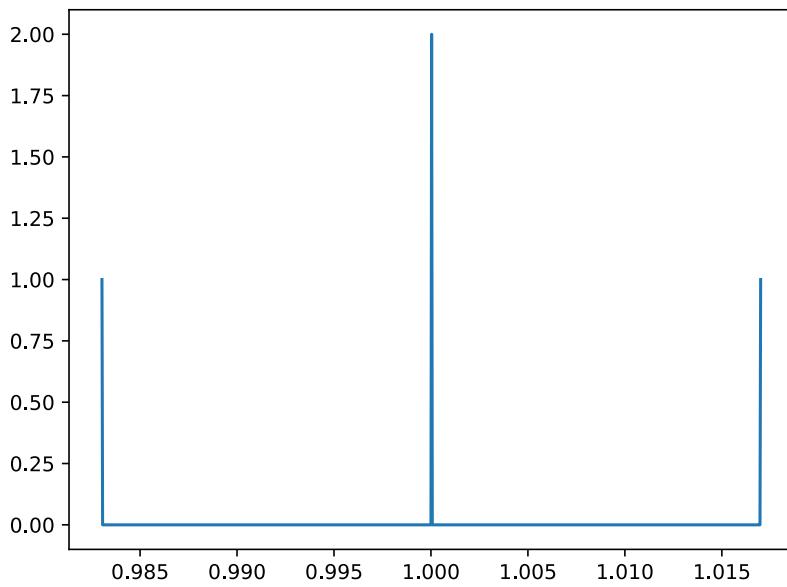
    # perform next split or return result
    if n > 0:
        return split(new_peaks, n, J, freq=freq)
    else:
        return new_peaks
```

```
split(1.00, 2, J=3.4, freq=400)
```

```
[1.008500000000002, 1.0, 1.0, 0.9915]
```

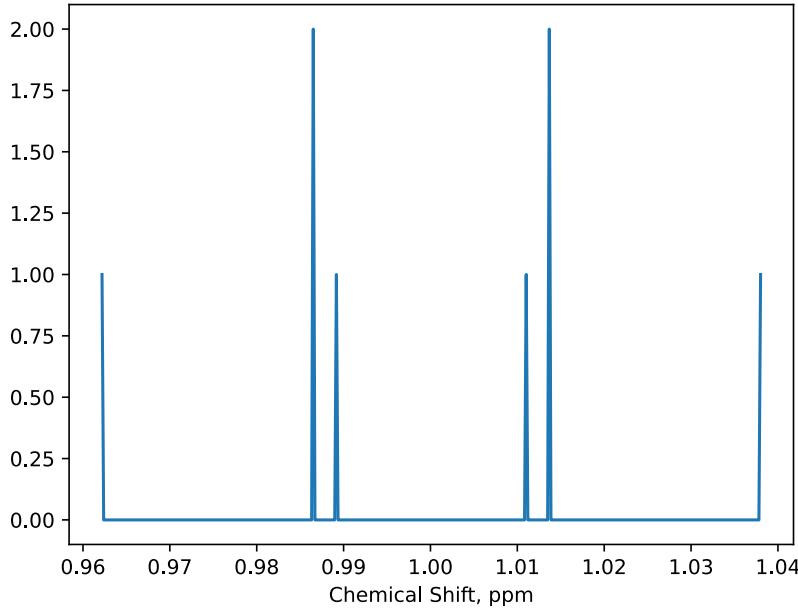
In the above example, a peak at 1.00 ppm has two neighboring protons that couple with it at 3.4 Hz, and the sample is observed at 400 MHz. There are four resulting peaks in the output list, but two peaks are at the same chemical shift of 1.00 ppm. This results in three peaks with the peak at 1.00 ppm being twice the magnitude as the other two. We can visualize this by binning the peaks and generating a line plot.

```
signal, ppm = np.histogram(split([1.00], 2, J=6.8), bins=1000)
plt.plot(ppm[1:], signal);
```



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.

```
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');
```



9.1.2 Single-Step Stepwise Chemical Kinetics

Another phenomenon that can be simulated deterministically 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 diminishing reactant concentrations. This occurs when reaction rates are dependent on the concentration of at least one reactant, 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 $k = 0.05 \text{ s}^{-1}$ 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 M in the first second giving us 0.95 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 M from [A] for the next second of reaction to get 0.903 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.

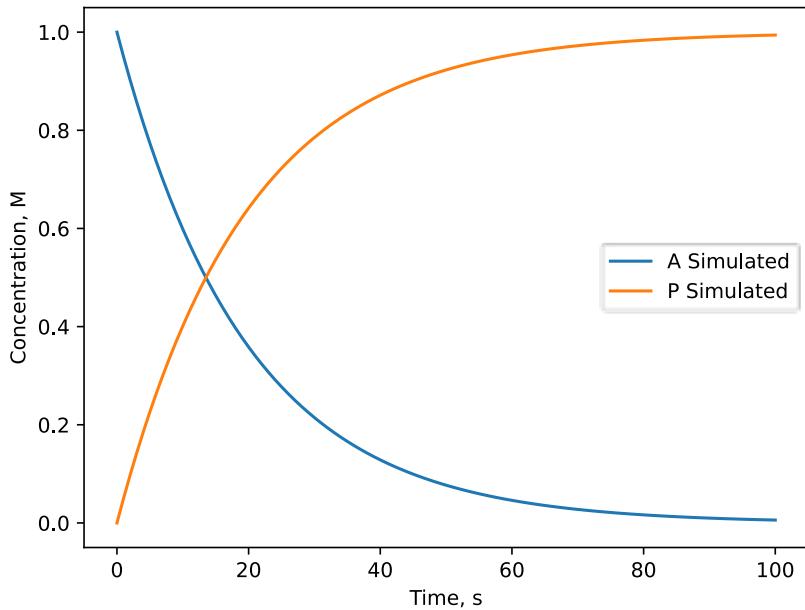
```
A, P = 1.00, 0.00 # molarity, M
k = 0.05 # 1/s for a first-order reaction
length = 100 # length of simulation in seconds
time = range(length + 1)

# 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
    # recalculate rate
    rate = k * A
    # recalculate new concentration
    A -= rate
    P += rate
```

You may be wondering why the first lines of code in the `for` loop records 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.

```
s = 5 # step size
plt.plot(time, A_conc, label='A Simulated')
plt.plot(time, P_conc, label='P Simulated')
plt.xlabel('Time, s')
plt.ylabel('Concentration, M')
plt.legend();
```



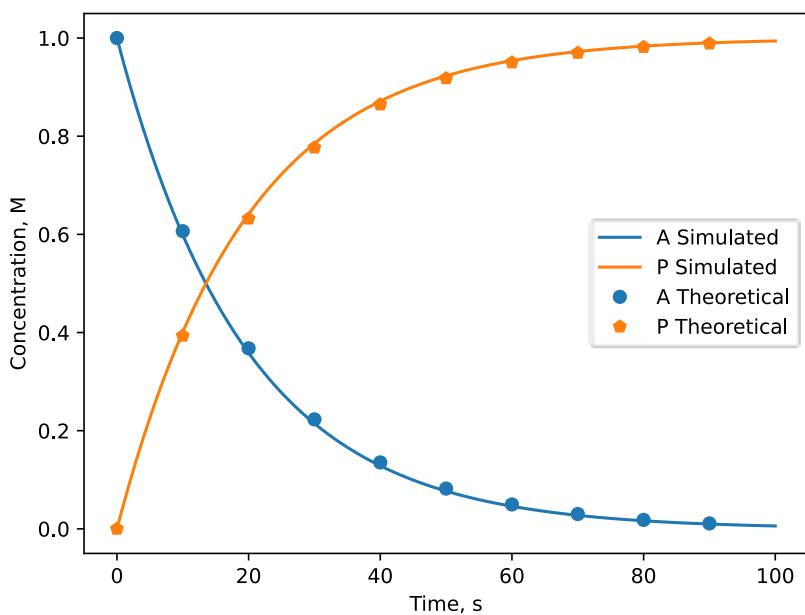
We can overlay this plot with the theoretical values using the integrated first-order rate law below.

```
t = np.arange(0,100, 10)
A_theor = 1.0 * np.exp(-k * t)
P_theor = np.ones(10) - A_theor

plt.plot(time, A_conc, '--', label='A Simulated')
plt.plot(time, P_conc, '--', label='P Simulated')
plt.xlabel('Time, s')
plt.ylabel('Concentration, M')

plt.plot(t, A_theor, 'C0o', label='A Theoretical')
plt.plot(t, P_theor, 'C1p', label='P Theoretical')

plt.legend();
```

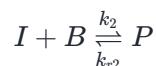


The theoretical equation and simulation results are in good agreement. A closer inspection of the two shows a slight

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. A more accurate method of performing kinetic simulations is presented in [section 9.1.4](#).

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 hand.

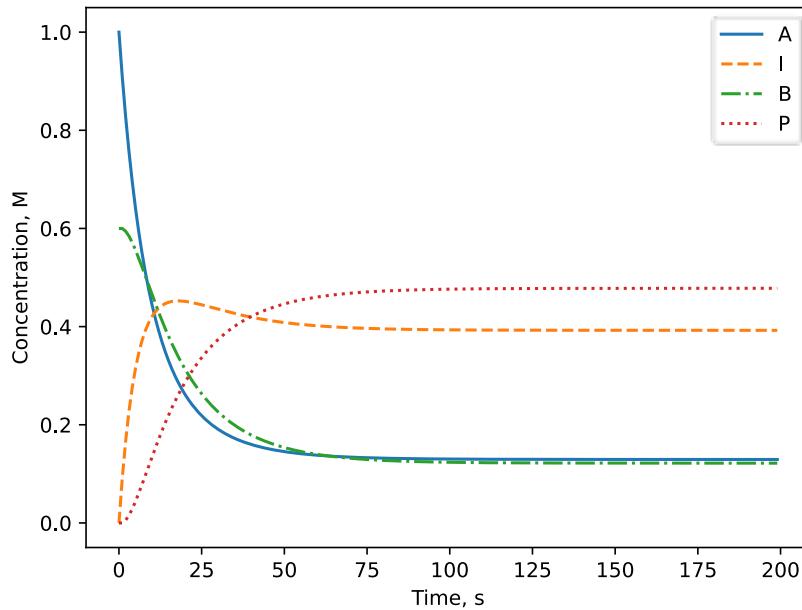


In this reaction, starting material A converts to intermediate I in the first step followed by starting material B combining with I to form the product P. Both of these steps are reversible, so there are four rate constants. The code and output of the simulation are below. Unlike the previous simulation, the simulation below appends values to lists (e.g., `A_conc`).

```
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

# the simulation
for sec in range(length):
    A_conc.append(A)
    I_conc.append(I)
    B_conc.append(B)
    P_conc.append(P)
    # recalculate rates
    rate_1 = k1 * A
    rate_r1 = kr1 * I
    rate_2 = k2 * B * I
    rate_r2 = kr2 * P
    #recalculate concentrations after next time increment
    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
```

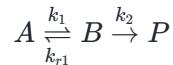
```
plt.plot(range(length), A_conc, label='A', ls='--')
plt.plot(range(length), I_conc, label='I', ls='---')
plt.plot(range(length), B_conc, label='B', ls='-.')
plt.plot(range(length), P_conc, label='P', ls=':')
plt.xlabel('Time, s')
plt.ylabel('Concentration, M')
plt.legend();
```



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. This is because the simulation fails to recalculate the rates quickly 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 where k_1 and k_{r1} are the forward and reverse rate constants, respectively, for the first step and k_2 is the rate constant for the second step.

$$\frac{d[A]}{dt} = -k_1[A] + k_{r1}[B]$$

$$\frac{d[B]}{dt} = k_1[A] - k_2[B] - k_{r1}[B]$$

$$\frac{d[P]}{dt} = k_2[B]$$

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.

```

k1, kr1, k2 = 0.2, 0.6, 0.3
A0, B0, P0 = 1.0, 0.0, 0.0
t = np.linspace(0, 50, 50)

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

```

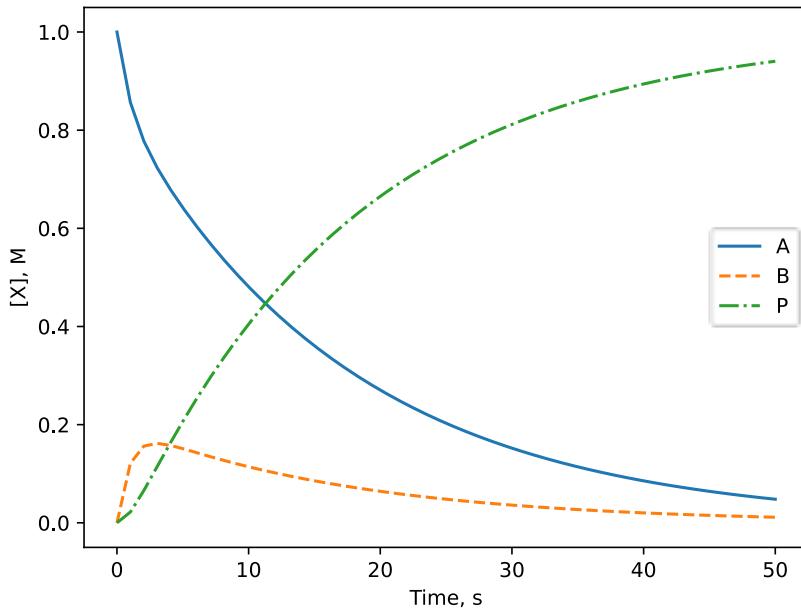
Because the `odeint()` function only takes the initial concentration (`A0`, `B0`, and `P0`) as a single argument, they need to be placed in a tuple.

```
A_t, B_t, P_t = scipy.integrate.odeint(rates, (A0, B0, P0), t).T
```

```

plt.plot(t, A_t, 'solid', label='A')
plt.plot(t, B_t, 'dashed', label='B')
plt.plot(t, P_t, 'dotted', label='P')
plt.xlabel('Time, s')
plt.ylabel('[X], M')
plt.legend();

```



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. 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 among other things. Rolling a die thousands of times is tedious, so 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. In the simulation below, 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 `binomial()` method is used here to generate a series of zeros and ones 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 the `atoms_remaining` variable.

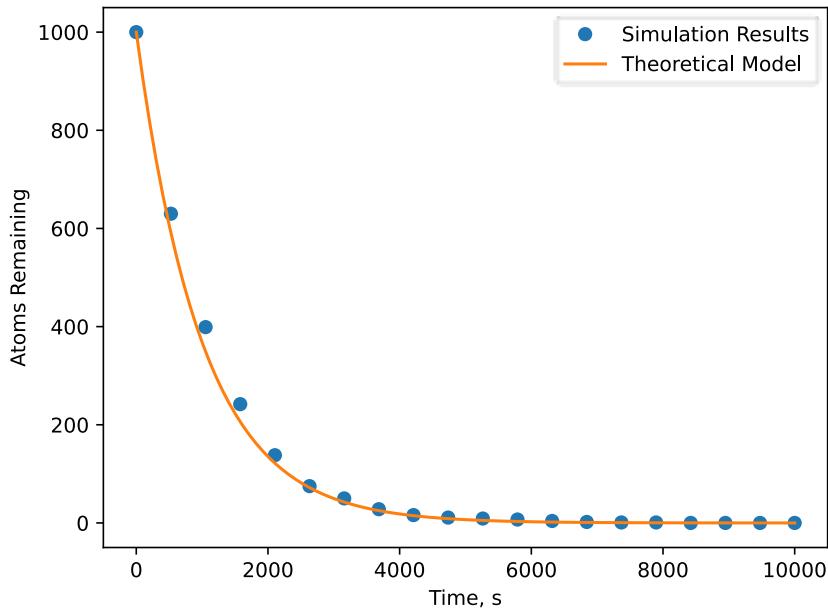
```
rng = np.random.default_rng()

starting_atoms = 1000
length = 10000 # length of simulation
num_atoms = starting_atoms
atoms_remaining = []
for x in range(length):
    atoms_remaining.append(num_atoms)
    # "rolls" dice and tallies up number of zeros
    decays = rng.binomial(1, p=0.001, size=num_atoms)
    decayed_count = np.sum(decays)
    # deduct decayed nuclei from the total
    num_atoms -= decayed_count

# convert list to array
atoms_remaining = np.array(atoms_remaining)
```

The simulation results stored in the `atoms_remaining` array 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 s^{-1} . For ease of viewing, only twenty data points from the simulation are plotting below.

```
# plot of simulation
step = np.linspace(0, length, 20)
plt.plot(step, atoms_remaining[::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 sometimes 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 dataset 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 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 $N = 19$.

$$95\%CI = \bar{x} \pm \frac{ts}{\sqrt{N}}$$

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 dataset, the 95% CI can be calculated, and we can test whether or not the true value is inside the CI. If we repeated 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.

```

trials = 100000
N = 20
t = 2.09
true = 6.2 # true value
# number of times mean inside 95% CI
in_interval = 0

for trial in range(trials):
    # create synthetic data
    error = rng.random(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 += 1

```

```
100 * in_interval / trials
```

94.85

The above simulation finds that almost 95% 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 datasets and should be closer to the theoretical value with increasing dataset size.

9.2.3 Random Flight Polymer

Polymers are long chains of repeating units called monomers. These chains can easily extend 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, 1]. NumPy does not provide a function for generating this range, so we can modify the [0,1] range from the `random()` method by subtracting 0.5 and multiplying by 2.

```

segments = 3000
coords = [[0, 0]]

for step in range(segments):
    x = coords[step][0] + 2 * (rng.random() - 0.5)
    y = coords[step][1] + 2 * (rng.random() - 0.5)
    coords.append([x, y])

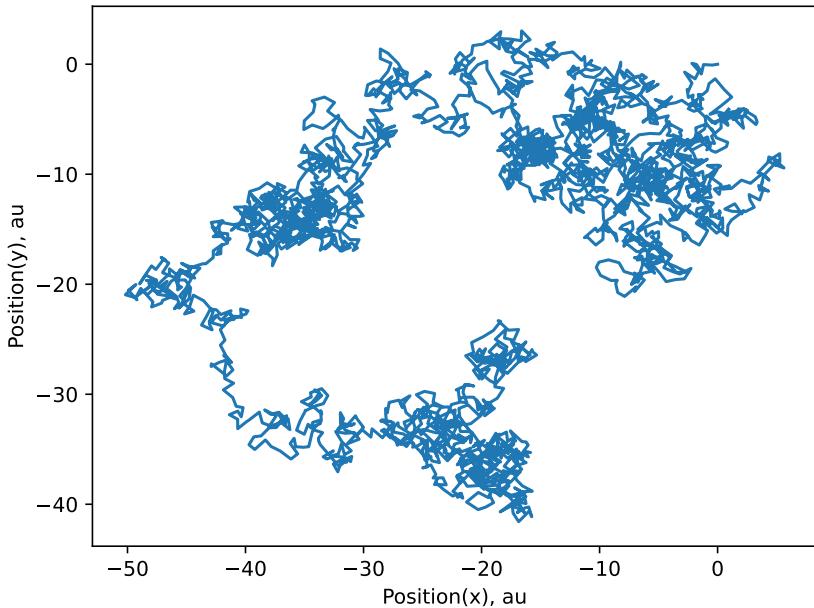
coords = np.array(coords)

```

```

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 a seemingly random fashion. If we rerun the above simulation, a different looking polymer conformation will be generated.

Further Reading

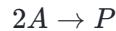
1. Downey, Allen Modeling and Simulation in Python. Book in progress. [AllenDowney/ModSimPy](#) (free resource)
2. Weiss, C. J. Introduction to Stochastic Simulations for Chemical and Physical Processes: Principles and Applications. *J. Chem. Educ.* **2017**, 94 (12), 1904–1910. <https://doi.org/10.1021/acs.jchemed.7b00395>
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. <https://doi.org/10.1021/ed085p1146>
4. Kneusel, R. T. *The Art of Randomness: Randomized Algorithms in the Real World*; No Starch Press: San Francisco, CA, 2024.

Exercises

Complete the following exercises in a Jupyter notebook. Any data file(s) referred to in the problems can be found in the [data](#) folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data

for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

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.

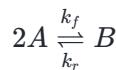


3. Plot the concentrations of A, B, C, and P with respect to time for the following three-step, non-reversible mechanism. The initial concentrations and rate constants are in the table below.



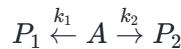
Step	Specie	[Specie] ₀ , M	Rate Constant, s ⁻¹
1	A	1.50	0.8
2	B	0.00	0.4
3	C	0.00	0.3
-	P	0.00	-

4. Simulate the following chemical equilibrium where the forward rate is described by $\text{Rate}_f = (1.3 \times 10^{-2} \text{ M}^{-1}\text{s}^{-1})[A]^2$ and the reverse rate is described by $\text{Rate}_r = (6.2 \times 10^{-3} \text{ s}^{-1})[B]$.



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 0.001 s^{-1} , 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 P₁ and P₂ and plot the resulting concentrations of both products versus time.



Use $k_1 = 0.02 \text{ M/s}$ and $k_2 = 0.04 \text{ M/s}$ 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?

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 `binomial()` method 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.

```
-A-A-A-A-A-A-B-B-B-B-B-A-A-A-A-B-B-B-A-A-A-A-
```

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 method such as `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.

```
rng = np.random.default_rng()
loc = np.cumsum(rng.integers(-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 `integers()` method and a Python dictionary or list containing the single letter amino acid codes.

11. Confidence intervals

- a) Convert the code for calculating a 95% confidence interval in [section 9.2.2](#) to a Python function that accepts the 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.
- b) 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 [section 9.1.1](#), simulate the splitting pattern for the tertiary proton in isopropyl alcohol ($(\text{CH}_3)_2\text{CHOH}$). In CDCl_3 , this proton is observed at 3.82 ppm with a coupling constant of 6 Hz. Assume no coupling with the hydroxyl proton is observed.

14. The law of large numbers indicates that as the number of trials increases, the observed average should overall converge on the statistical average. For example, when rolling a six-sided die, all numbers are equally probable to land up, so if we roll a number of dice, the average of all the numbers is expected to be around 3.5 (i.e., $(1 + 2 + 3 + 4 + 5 + 6)/6 = 3.5$). Using the `integers()` method, simulate the rolling of between two and five thousand, six-sided dice and plot the resulting average number versus the number of dice rolled. Include at least a hundred data points in your plot and label your axes.

Chapter 10: Plotting with Seaborn

Contents

- 10.1 Seaborn Plot Types
- 10.2 Regression Plots
- 10.2.2 Implot
- 10.3 Categorical Plots
- 10.4 Distribution Plots
- 10.5 Pair Plot
- 10.6 Heat Map
- 10.7 Relational Plots
- 10.8 Internal Datasets
- Further Reading
- Exercises

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 can be 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 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](#) uses it, so we will as well. All code in this chapter assumes the following import.

```
import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

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, and it is certainly not a complete list.

Table 1 Seaborn Plotting Type Categories Covered Herein

Category	Description
Regression	Draws a regression line through the data
Categorical	Plots frequency versus a category
Distribution	Plots frequency versus a continuous value
Matrix	Displays the data as a colored grid
Relational	Visualizes the relationship between two continuous variables

One distinction between some of the plotting categories above is whether they display continuous versus discrete/categorial information. When data are continuous, they can be nearly any value in a range like the density of a metal. This in contrast to discrete or categorial data that places data in a limited number of groups or bins such as the element(s) present in a metal sample.

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.

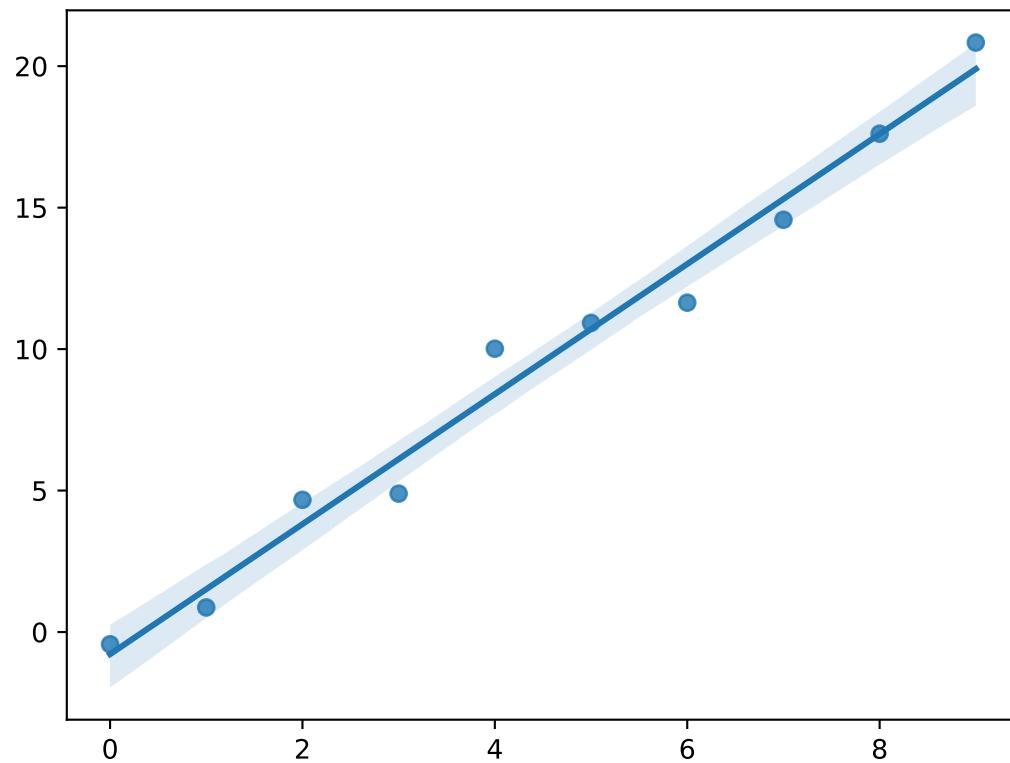
10.2.1 regplot

The `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.

```
rng = np.random.default_rng()
```

```
x = np.arange(10)
y = 2 * x + rng.normal(size=10)
```

```
sns.regplot(x=x, y=y);
```



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.

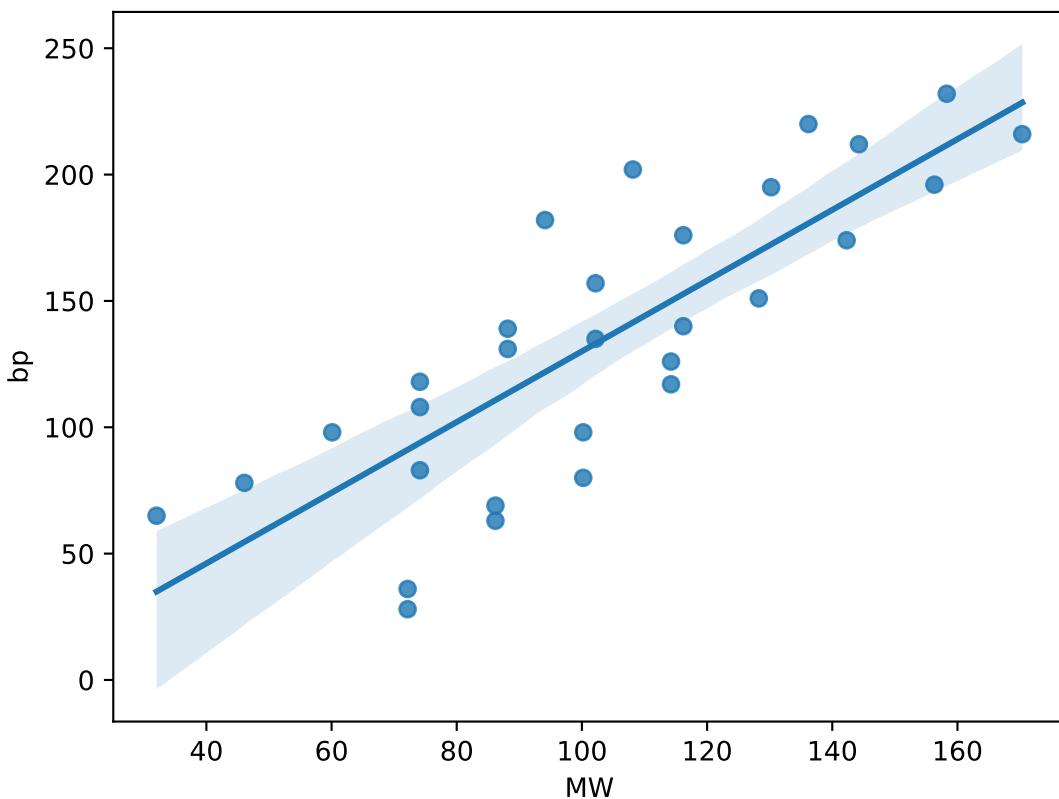
```
bp = pd.read_csv('data/org_bp.csv')
bp
```

	bp	MW	type
0	65	32.04	alcohol
1	78	46.07	alcohol
2	98	60.10	alcohol
3	118	74.12	alcohol
4	139	88.15	alcohol
5	157	102.18	alcohol
6	176	116.20	alcohol
7	195	130.23	alcohol
8	212	144.25	alcohol
9	232	158.28	alcohol
10	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

	bp	MW	type
25	135	102.18	alcohol
26	140	116.20	alcohol
27	182	94.11	alcohol
28	202	108.14	alcohol
29	220	136.19	alcohol

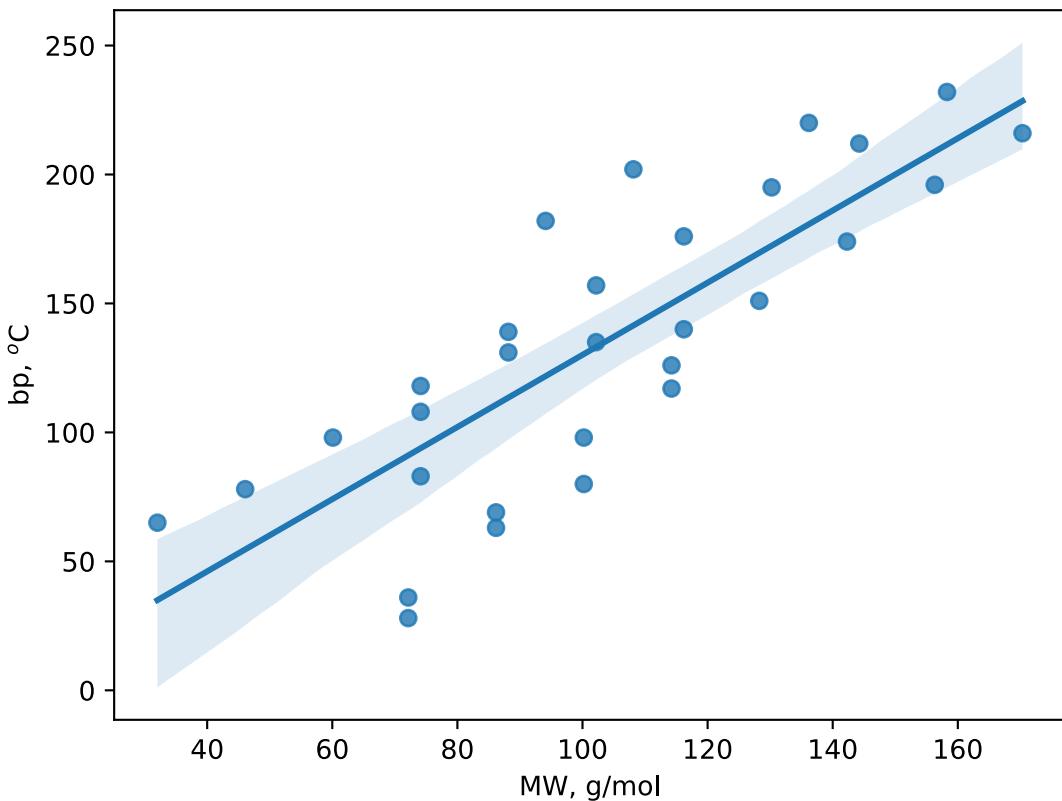
If you choose to provide column names from a pandas DataFrame, you must also provide the name of the DataFrame using the `data` keyword argument.

```
sns.regplot(x='MW', y='bp', data=bp);
```



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.

```
sns.regplot(x='MW', y='bp', data=bp)
plt.xlabel('MW, g/mol')
plt.ylabel('bp, $^\circ$C');
```



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()` function can do exactly this.

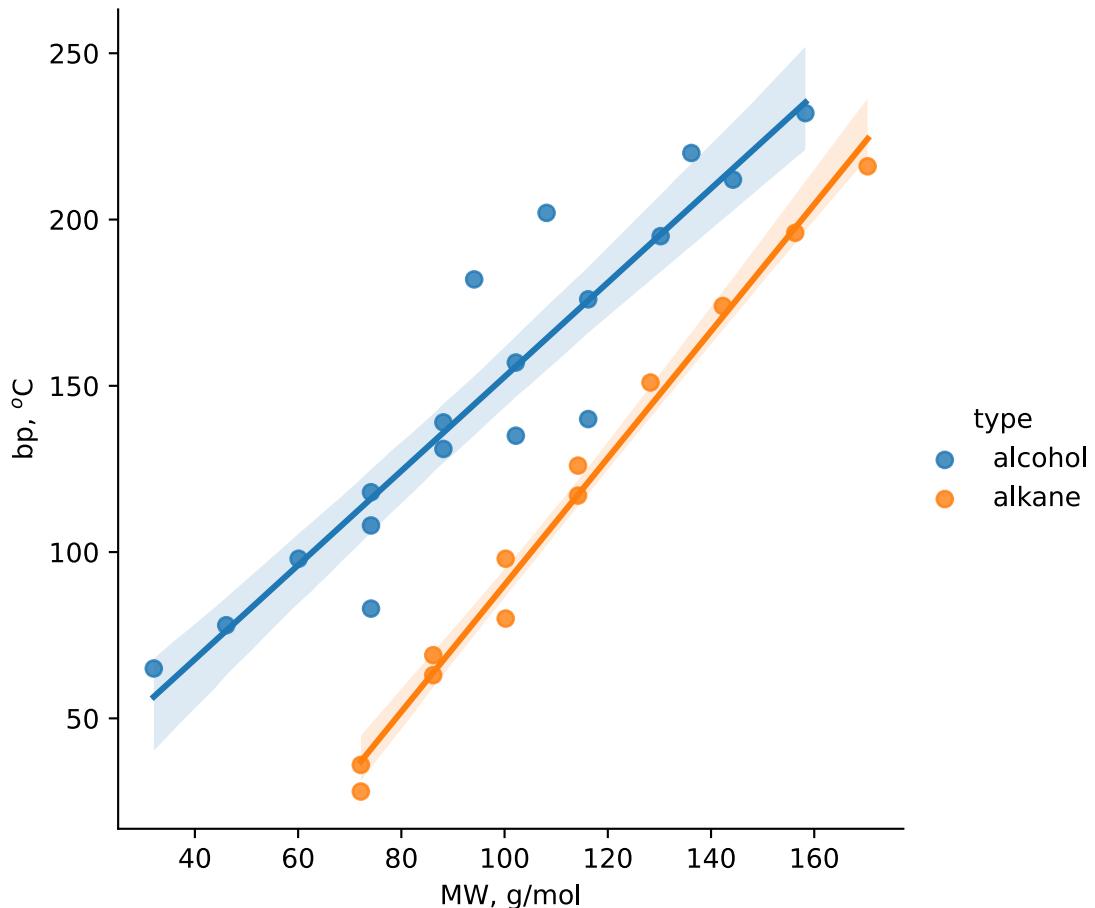
The `lmplot()` function takes the `x` and `y` variables and the DataFrame name as either positional or keyword arguments, so the function call could also be as shown below where the first three arguments are positional arguments providing the `x`-values, `y`-values, and the DataFrame name in this order.

```
sns.lmplot('MW', 'bp', bp, hue='type')
```

The `hue=` argument is the column name that dictates the color of the markers, so in this

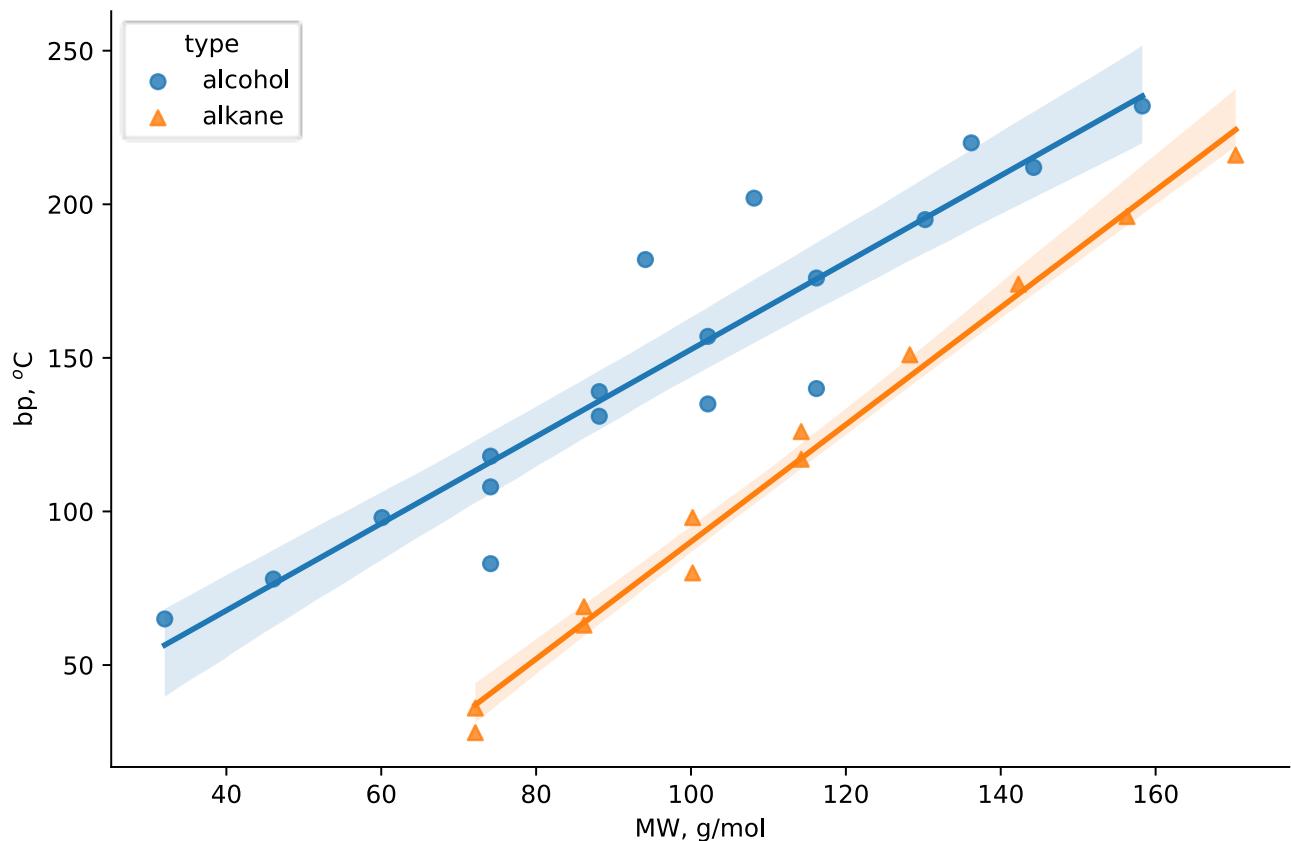
example, it will be the type of organic molecule.

```
sns.lmplot(x='MW', y='bp', data=bp, hue='type')
plt.xlabel('MW, g/mol')
plt.ylabel('bp, $^{\circ}\text{C}');
```



The `lmplot()` function also provides arguments for modifying the appearance of the plot. Below is a demonstration of a few extra adjustments to the plot. The `facet_kws` argument takes extra parameters in the form of a dictionary with key-value pairs. In this case, the `legend_out` key controls whether the legend is outside the plot's boundaries. The `aspect` argument sets the ratio of the x-axis versus the y-axis, and the marker shapes can also be modified using the `markers` argument with matplotlib conventions from [section 3.1.2](#).

```
sns.lmplot(x='MW', y='bp', hue='type', data=bp, markers=['o', '^'],
            aspect=1.5, facet_kws={'legend_out':False})
plt.xlabel('MW, g/mol')
plt.ylabel('bp, $^{\circ}\text{C}');
```



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 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. First, we need to load the values into a DataFrame.

```
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]]
```

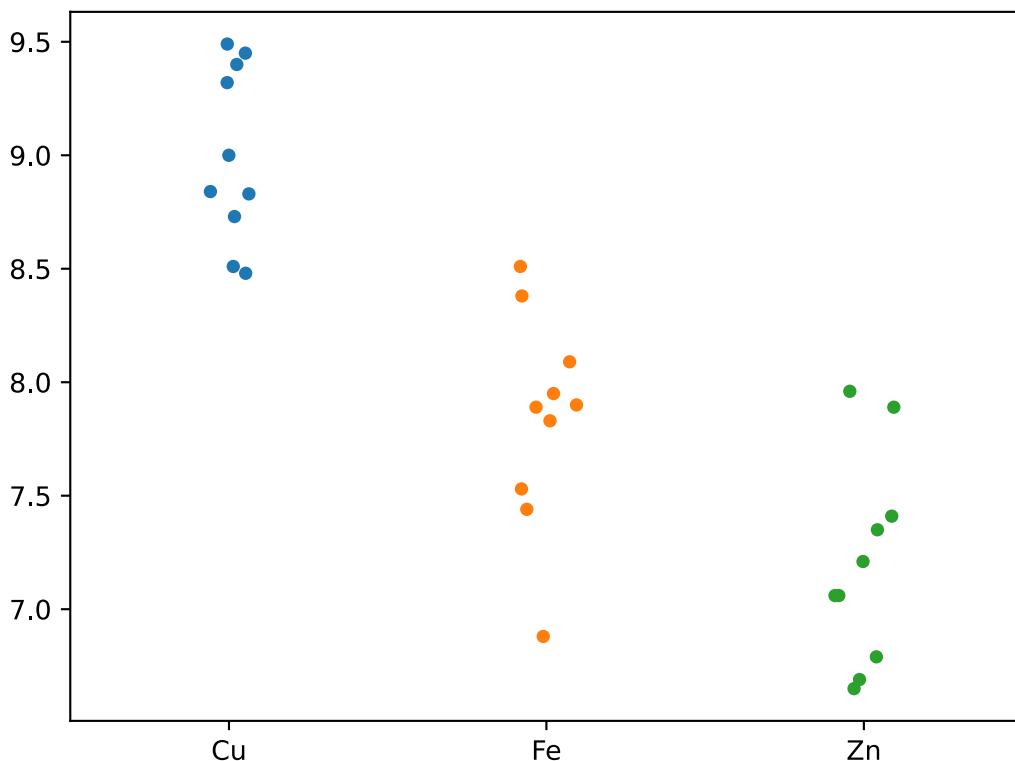
```
df = pd.DataFrame(densities, columns=labels)
df.head()
```

	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.

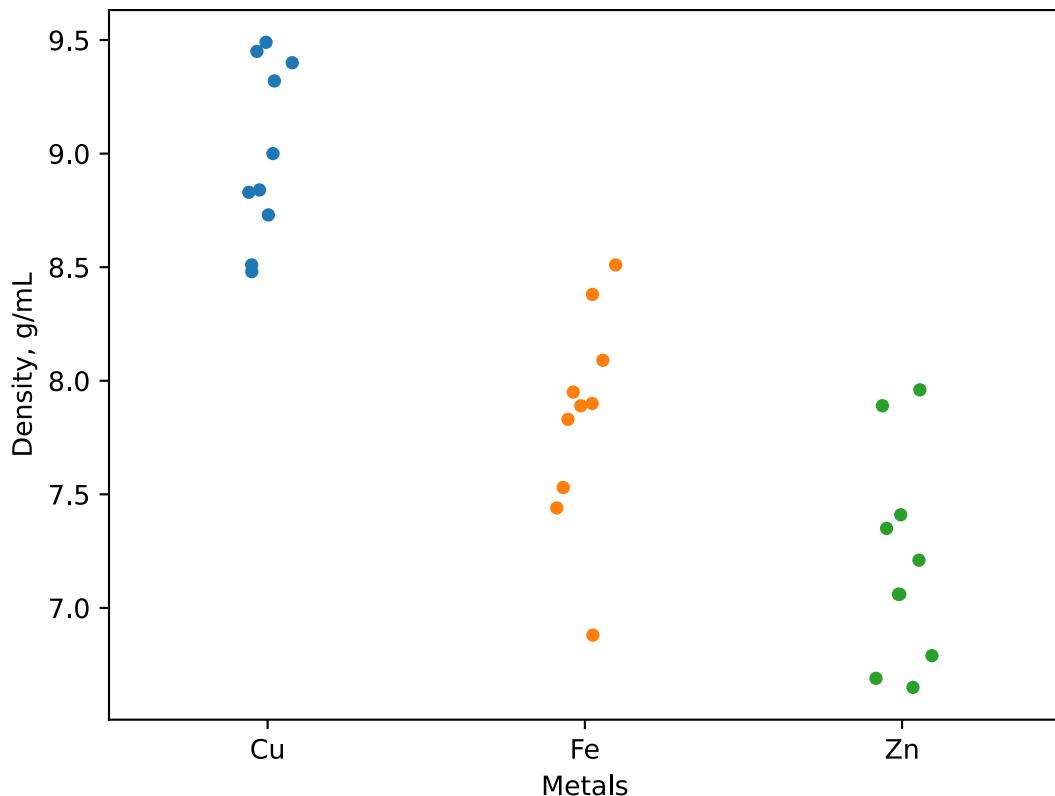
```
sns.stripplot(data=df);
```



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.

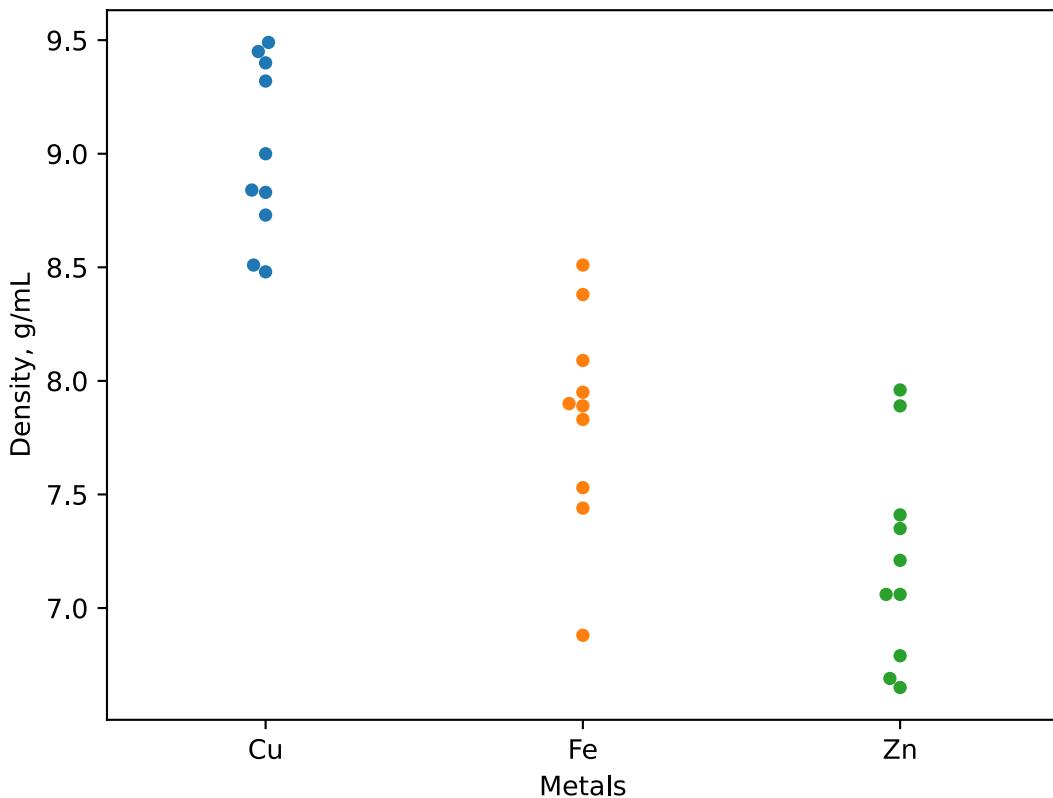
```
sns.stripplot(data=df)
plt.ylabel('Density, g/mL')
plt.xlabel('Metals');
```



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 as can happen with larger numbers of data points. 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 points are no permitted to overlap to make the quantity more apparent.

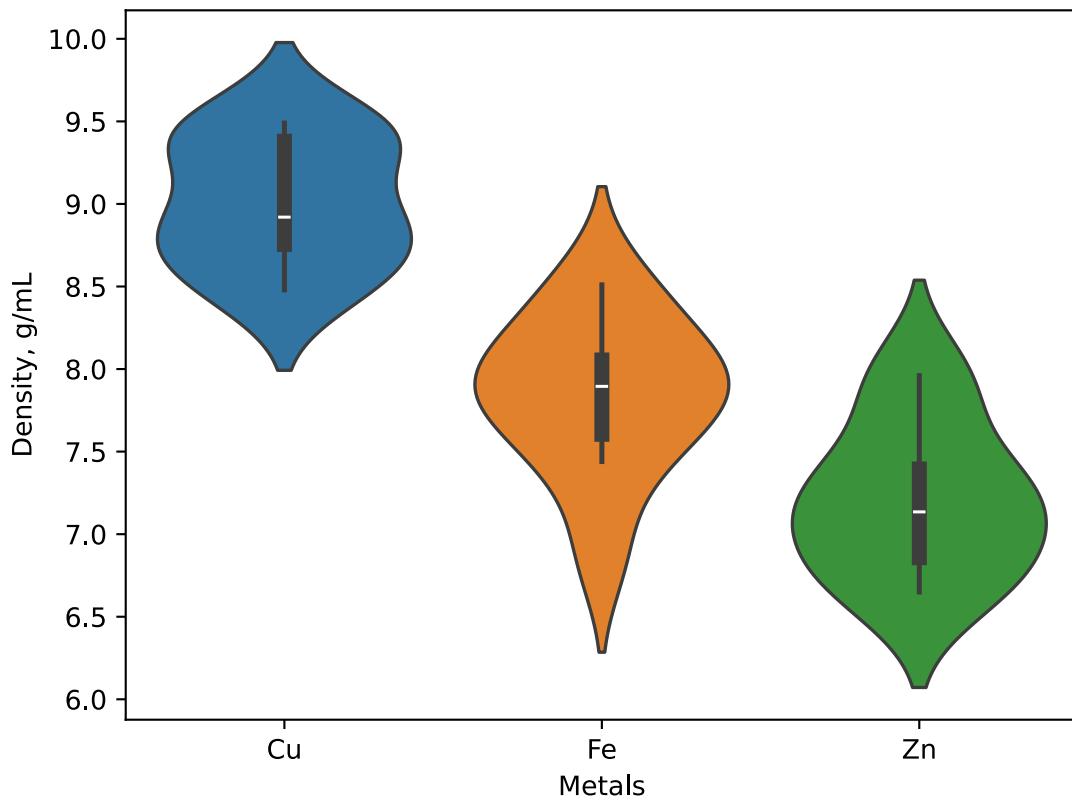
```
sns.swarmplot(data=df)
plt.ylabel('Density, g/mL')
plt.xlabel('Metals');
```



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.

```
sns.violinplot(data=df)
plt.ylabel('Density, g/mL')
plt.xlabel('Metals');
```



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 1)

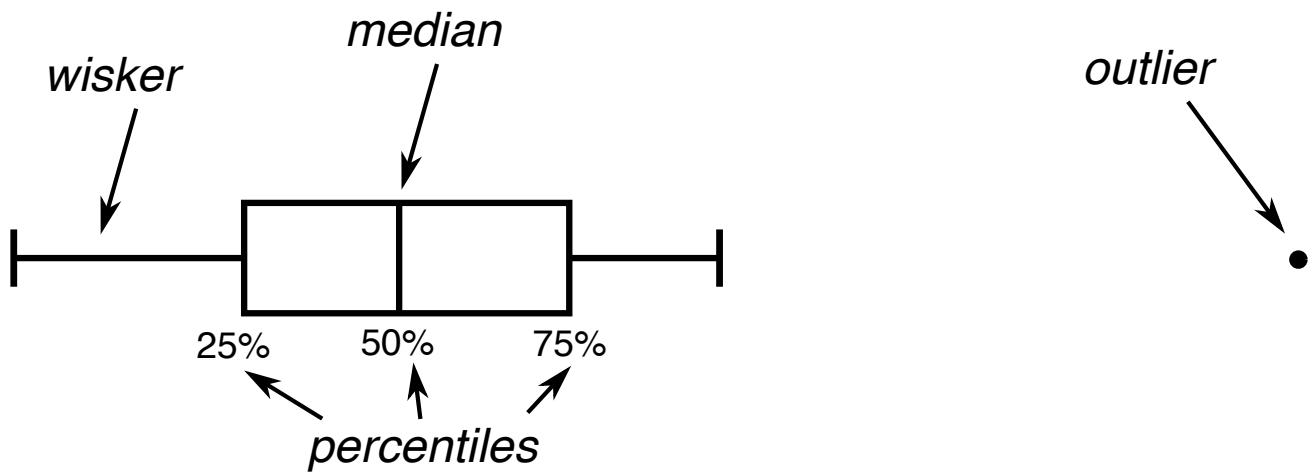
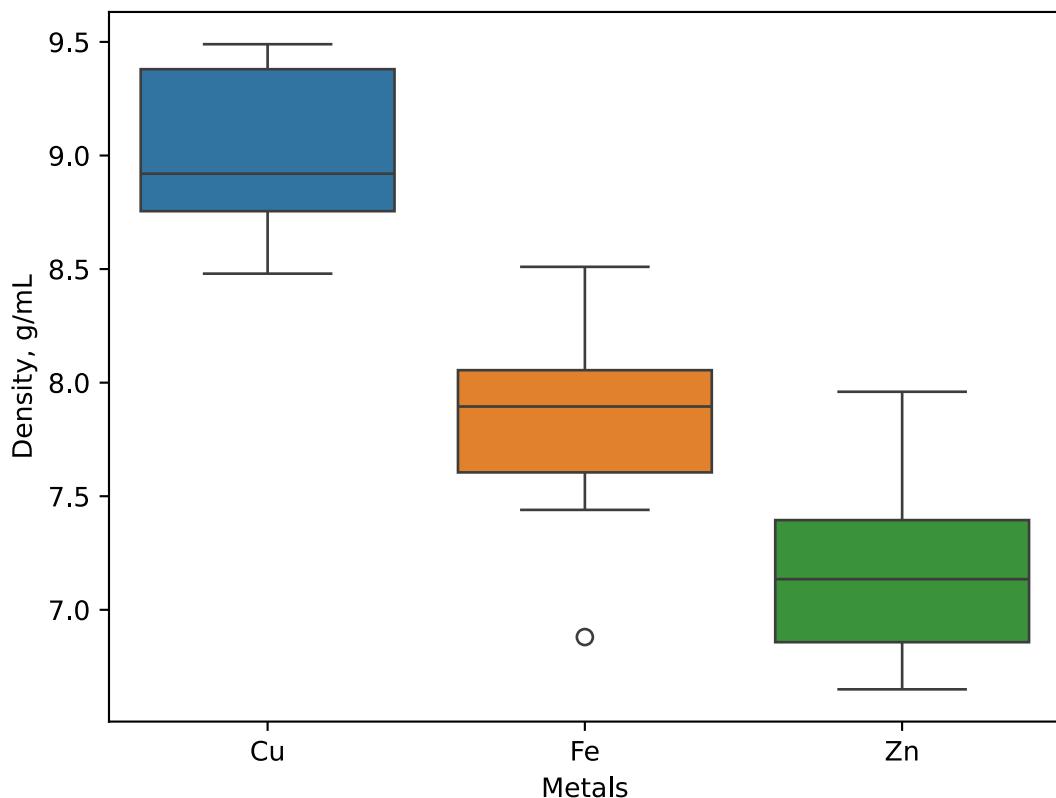


Figure 1 A box plot is composed of a box with lines at the 25th, 50th, and 75th percentiles and whiskers that extend out to the rest of the non-outlier data points. If a data point is greater than $1.5 \times$ the inner quartile range from the 25th or 75th percentiles, it is an outlier represented by a dot.

```
sns.boxplot(data=df)
plt.ylabel('Density, g/mL')
plt.xlabel('Metals');
```



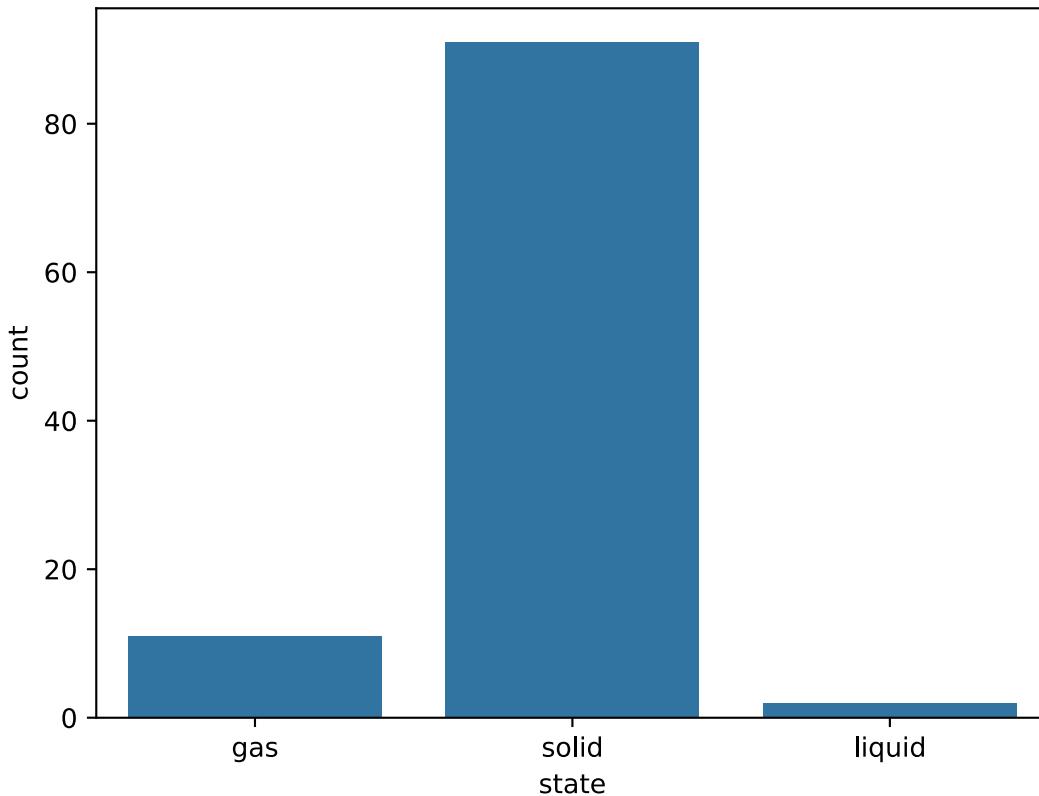
10.3.5 Count Plot

The count plot represents the frequency of values for different categories. This is 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.

```
elem = pd.read_csv('data/elements_data.csv')
elem.head()
```

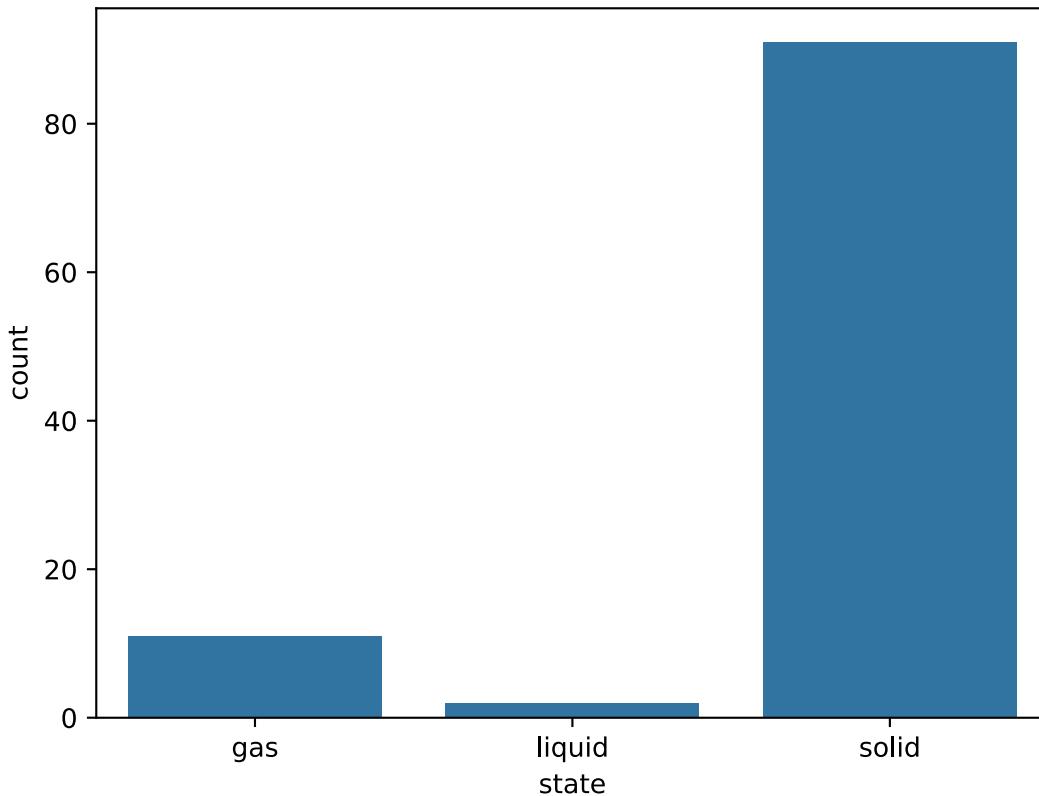
	symbol	AN	row	block	state
0	H	1	1	s	gas
1	He	2	1	s	gas
2	Li	3	2	s	solid
3	Be	4	2	s	solid
4	B	5	2	p	solid

```
sns.countplot(x='state', data=elem);
```



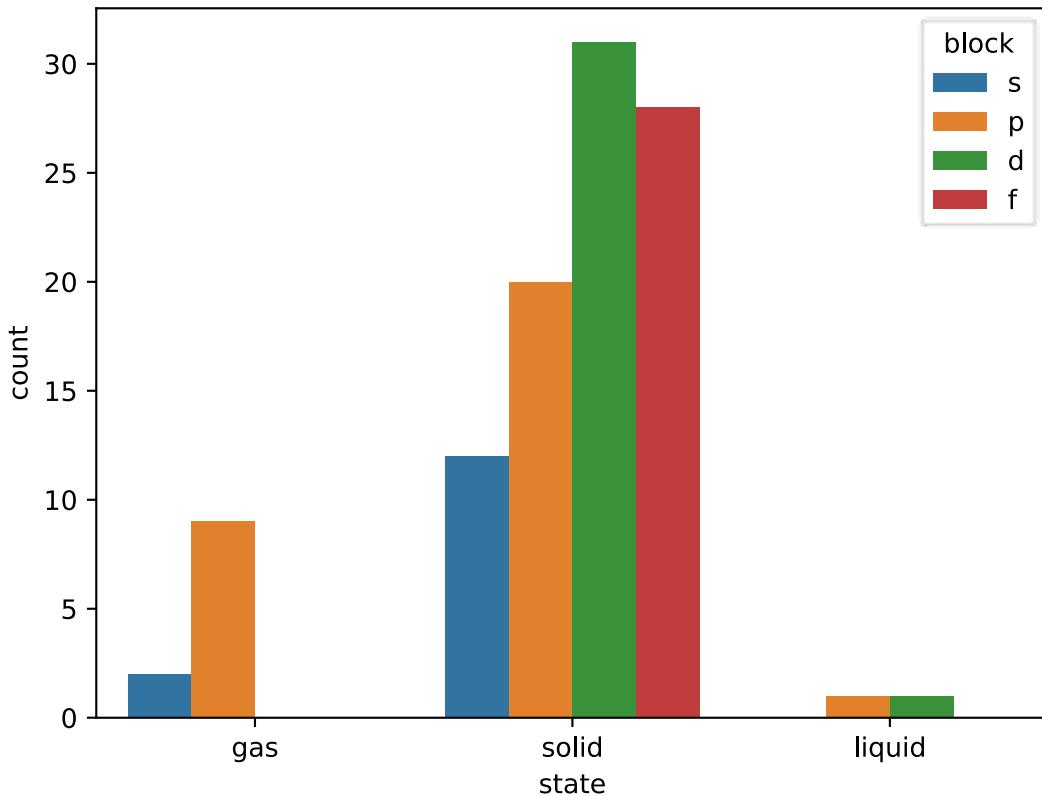
Like many plotting types in seaborn, the count plot can be further customized through keyword arguments and using other available data. One shortcoming of the above plot is that the states are listed in the order they first appear in the dataset instead of based on disorder. We can assert a different order by providing the `order` argument as a list of how the states should appear.

```
sns.countplot(x='state', data=elem, order=['gas', 'liquid', 'solid']);
```



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.

```
sns.countplot(x='state', hue='block', data=elem);
```



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.

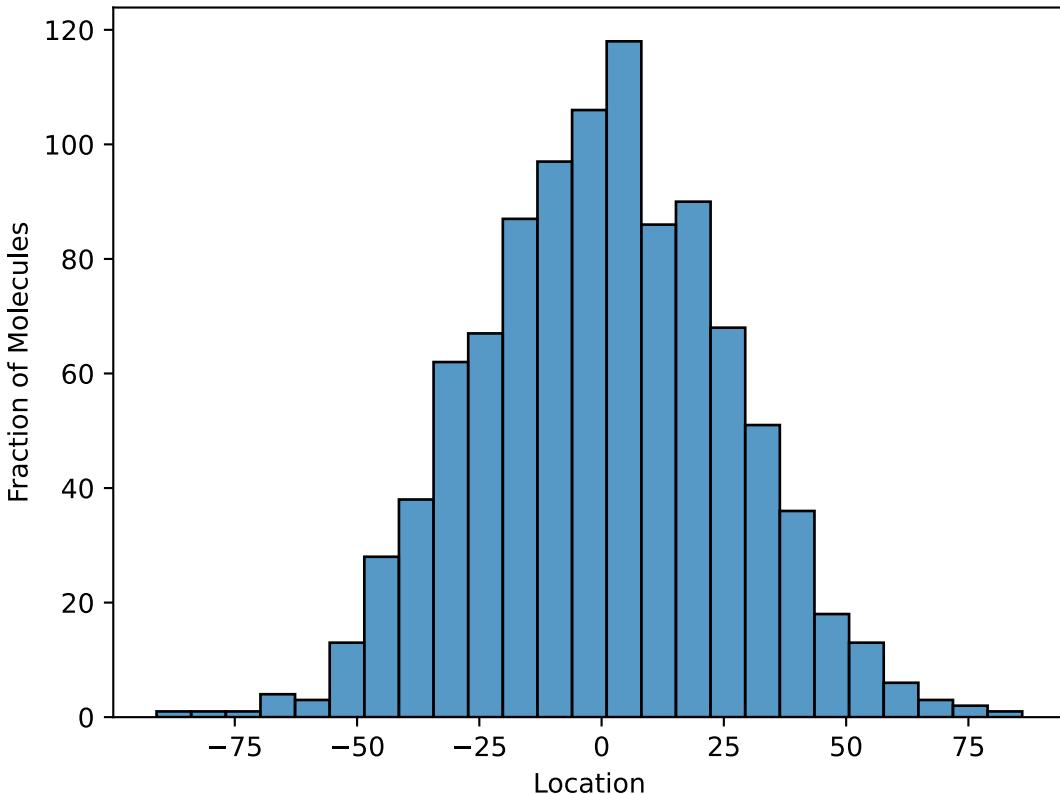
10.4.1 histplot

The `histplot()` function is one of the most basic distribution plotting functions in seaborn. This function is similar to the matplotlib `plt.hist()` function except that seaborn brings a few extra options like setting the color (`hue=`) based on a particular column of data.

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.

```
loc = np.zeros(1000) # locations of molecules
for step in range(1000):
    loc += rng.integers(-1, high=2, size=1000)
```

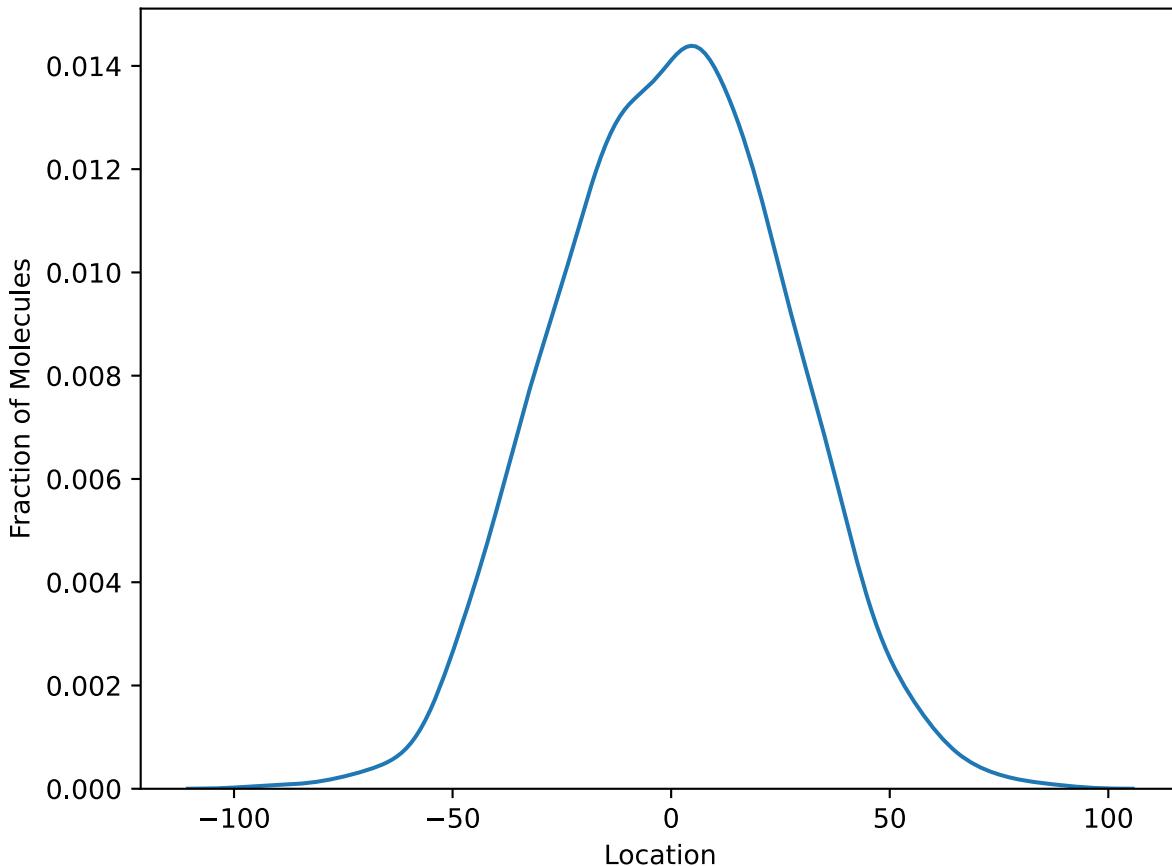
```
sns.histplot(loc)
plt.xlabel('Location')
plt.ylabel('Fraction of Molecules');
```



10.4.2 kde Plot

The `kdeplot()` function is very similar to the `histplot()` except that it 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.

```
sns.kdeplot(loc)
plt.xlabel('Location')
plt.ylabel('Fraction of Molecules')
plt.tight_layout()
```

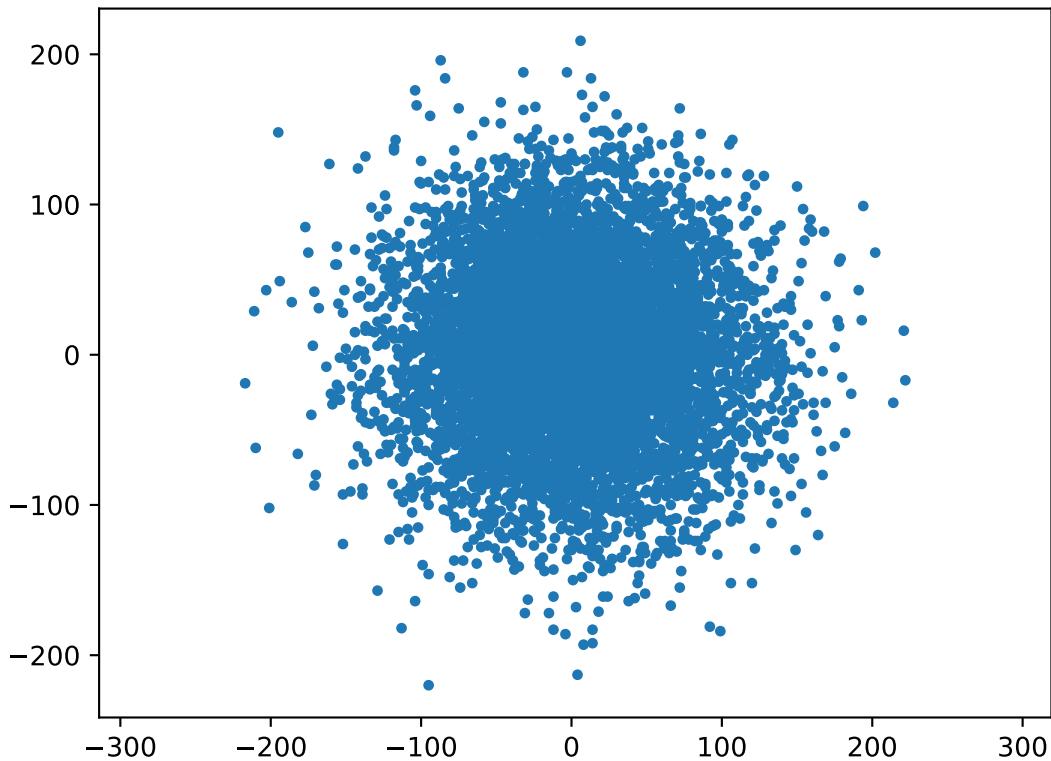


10.4.3 jointplot (diffusion simulation)

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 and the results. The principles are the same as above except applied to two dimensions.

```
x = np.sum(rng.integers(-1, high=2, size=(5000, 7000)), axis=0)
y = np.sum(rng.integers(-1, high=2, size=(5000, 7000)), axis=0)
```

```
plt.plot(x, y, '.')
plt.axis('equal');
```

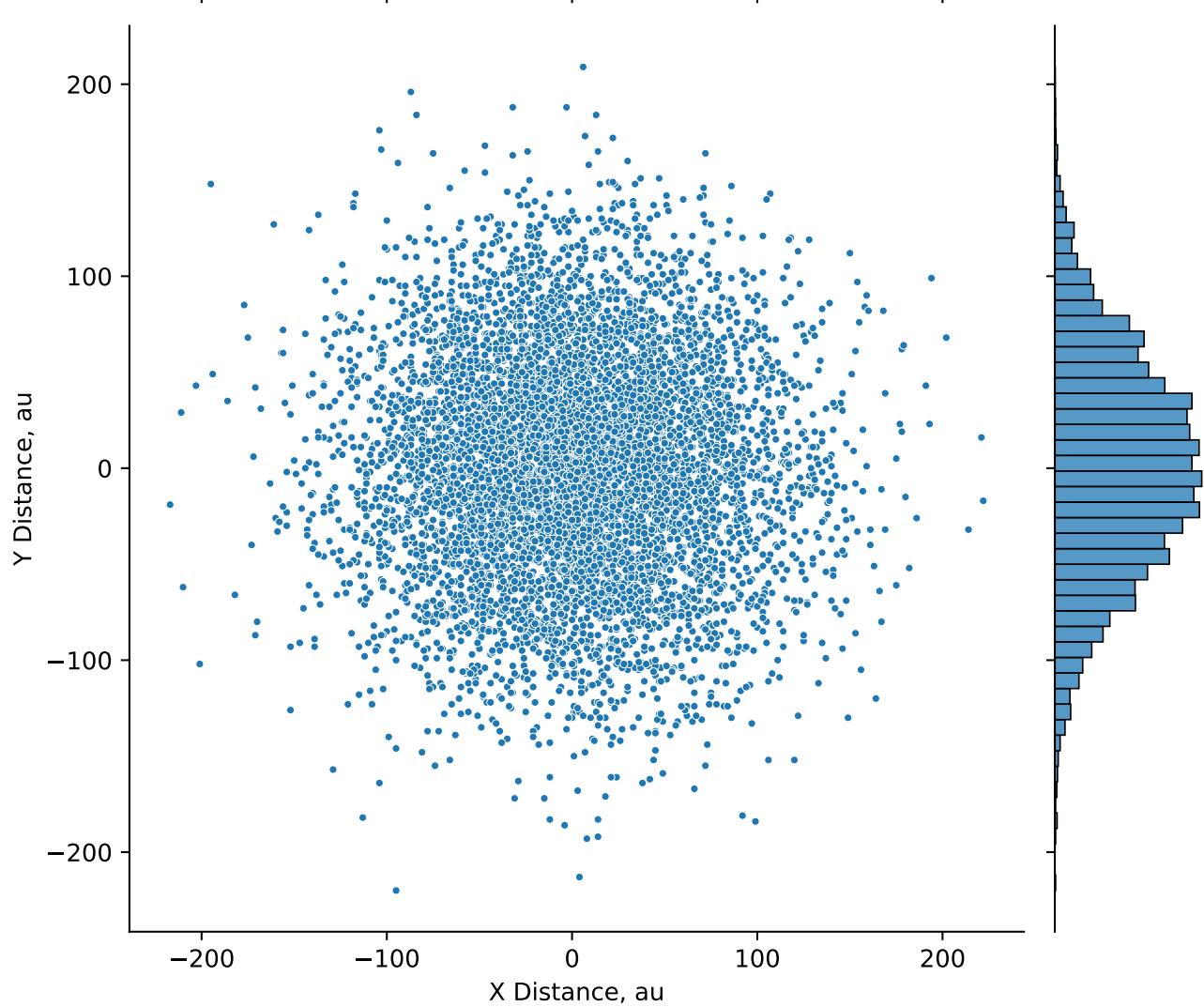


One of the issues with this plot is that there are so many data points in the plot that it is difficult to determine the distribution/density inside the blanket of solid dots. The seaborn joint plot adds histograms to the side to help the viewer recognize where most of the data points reside.

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.

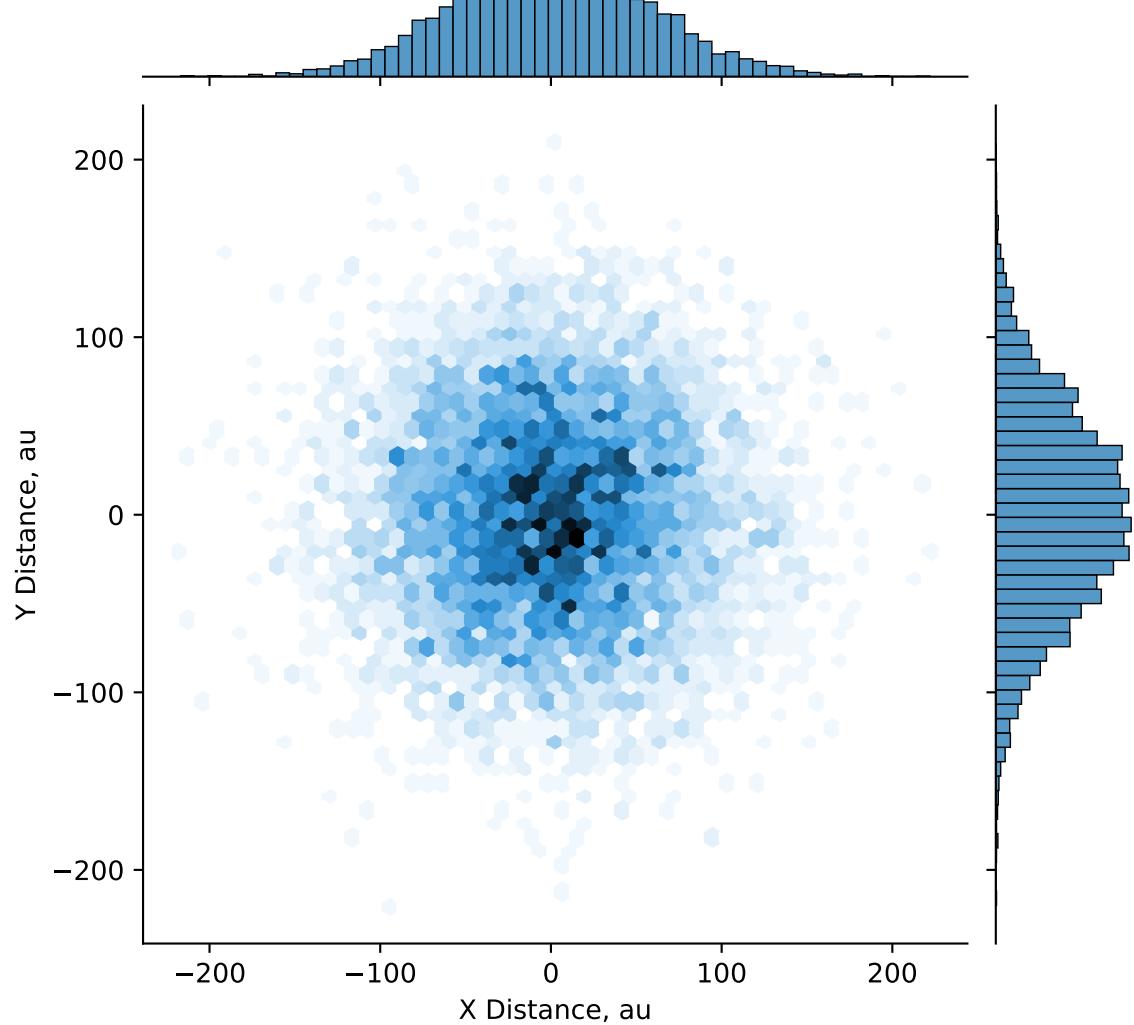
```
df = pd.DataFrame(data={'X Distance, au': x, 'Y Distance, au': y})  
sns.jointplot(x=df['X Distance, au'], y=df['Y Distance, au'],  
               height=7, color='C0', joint_kws={'s':10})
```

```
<seaborn.axisgrid.JointGrid at 0x123948050>
```

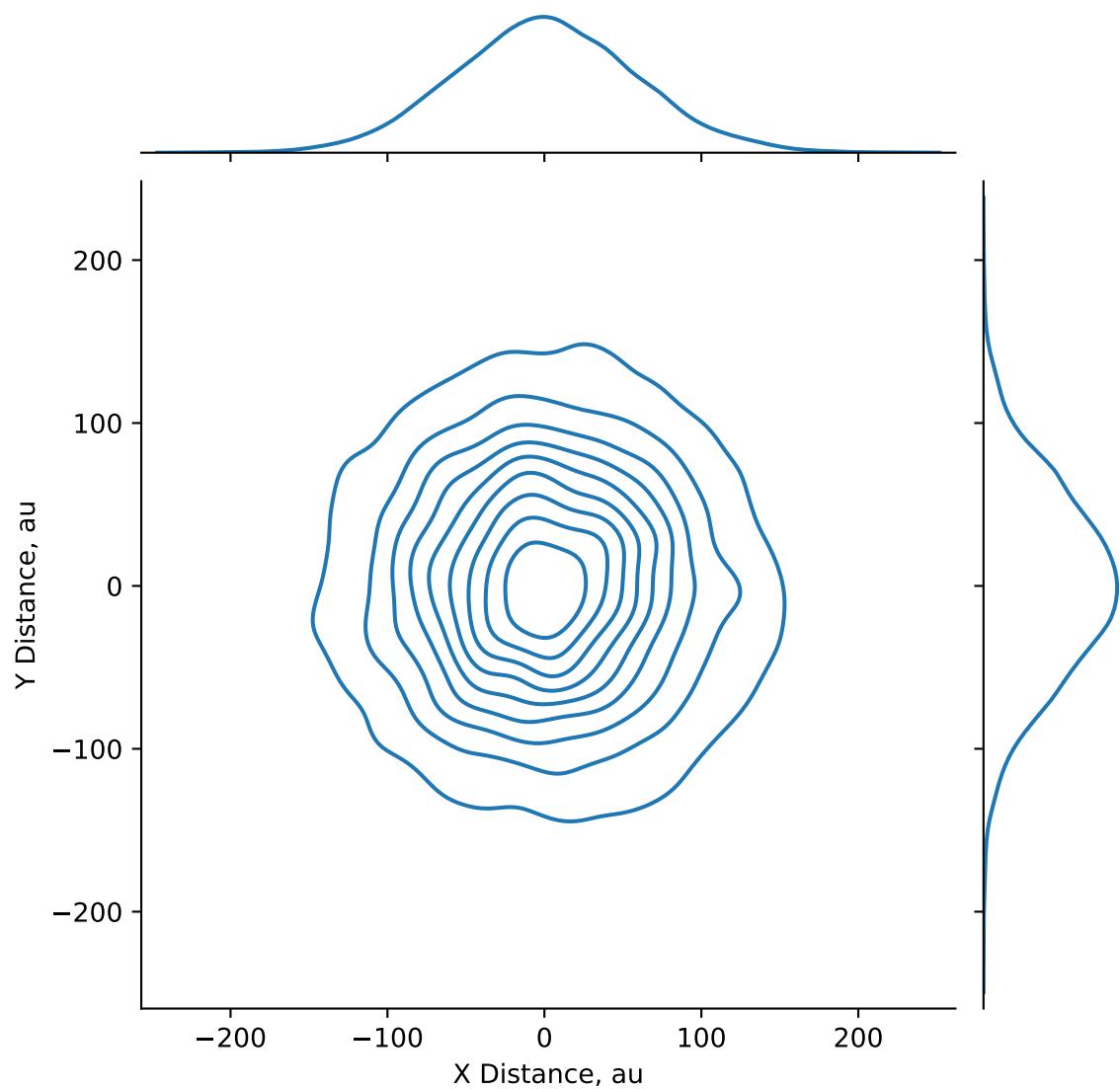


There are numerous arguments to fine tune the joint plot. For example, the joint plot does not need to be a scatter plot with histograms. The density of the data points can be represented with hexagonal patches or *kernel density estimates (kde)*. The latter represents the density of points through contours 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.

```
sns.jointplot(x=df['X Distance, au'], y=df['Y Distance, au'], kind='hex');
```



```
sns.jointplot(x=df['X Distance, au'], y=df['Y Distance, au'], kind='kde');
```



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.

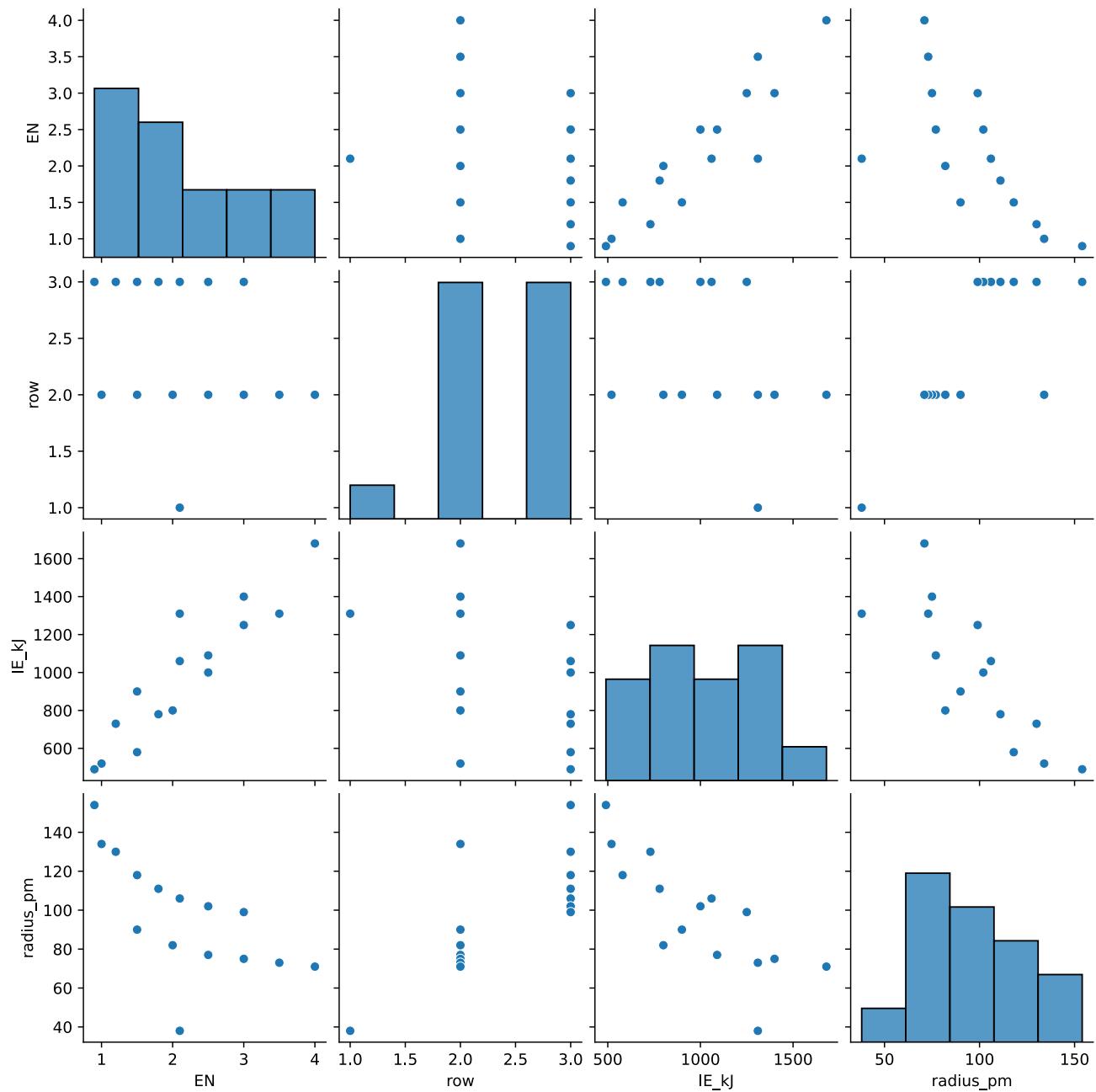
```
per = pd.read_csv('data/periodic_trends.csv')
per.head()
```

	symbol	AN	EN	row	IE_kJ	radius_pm
0	H	1	2.1	1	1310	38
1	Li	3	1.0	2	520	134
2	Be	4	1.5	2	900	90
3	B	5	2.0	2	800	82
4	C	6	2.5	2	1090	77

```
per.drop(['AN', 'symbol'], axis=1, inplace=True)
per.head()
```

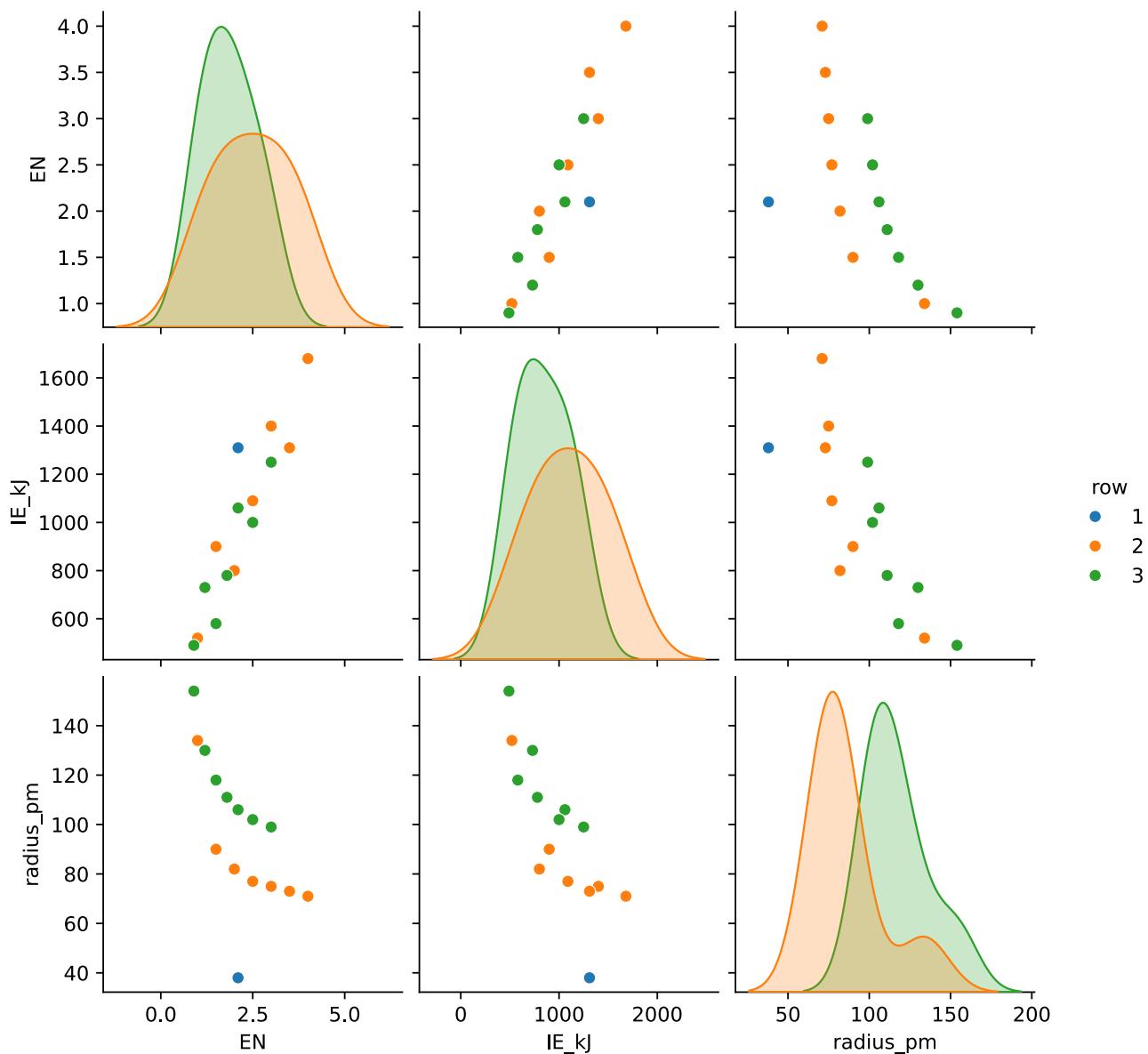
	EN	row	IE_kJ	radius_pm
0	2.1	1	1310	38
1	1.0	2	520	134
2	1.5	2	900	90
3	2.0	2	800	82
4	2.5	2	1090	77

```
sns.pairplot(per);
```



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.

```
sns.pairplot(per, hue='row', palette='tab10');
```



10.6 Heat Map

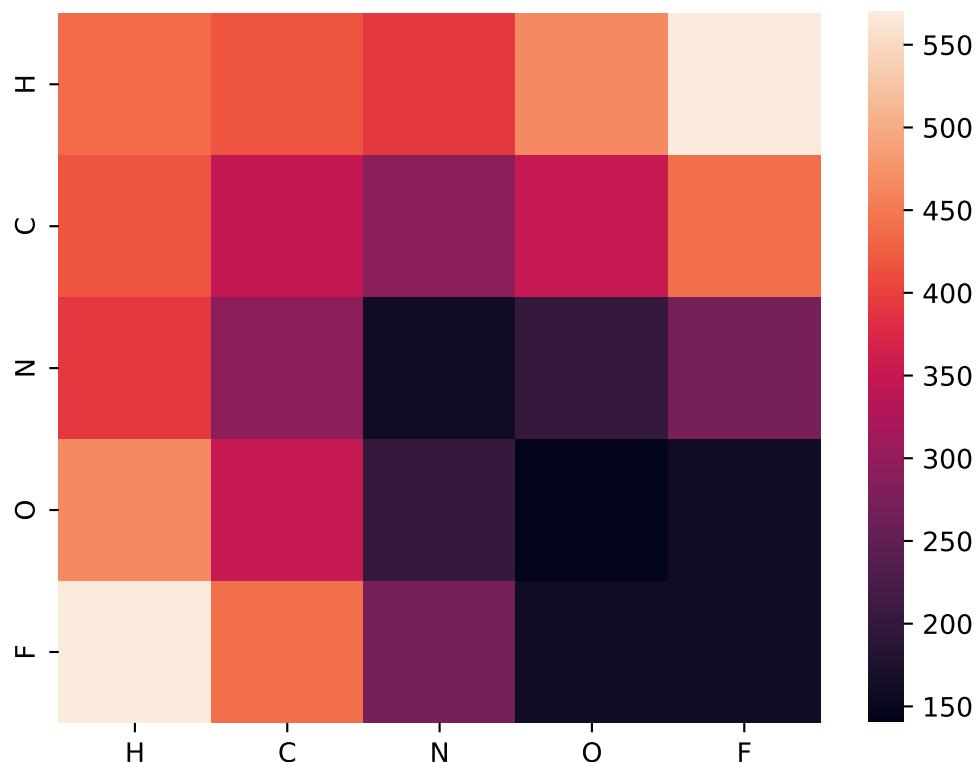
Heat maps are color representations of 2D grids of numerical data and are ideal for making large tables of values easily interpretable. As an example, we can import a table of bond dissociation energies (in kJ/mol) and visualize these data as a heat map. In the following pandas function call, the `index_col=0` tells pandas to apply the first column as column headers as well.

```
bde = pd.read_csv('data/bond_enthalpy_kJmol.csv', index_col=0)
bde
```

	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 `heatmapt()` 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.

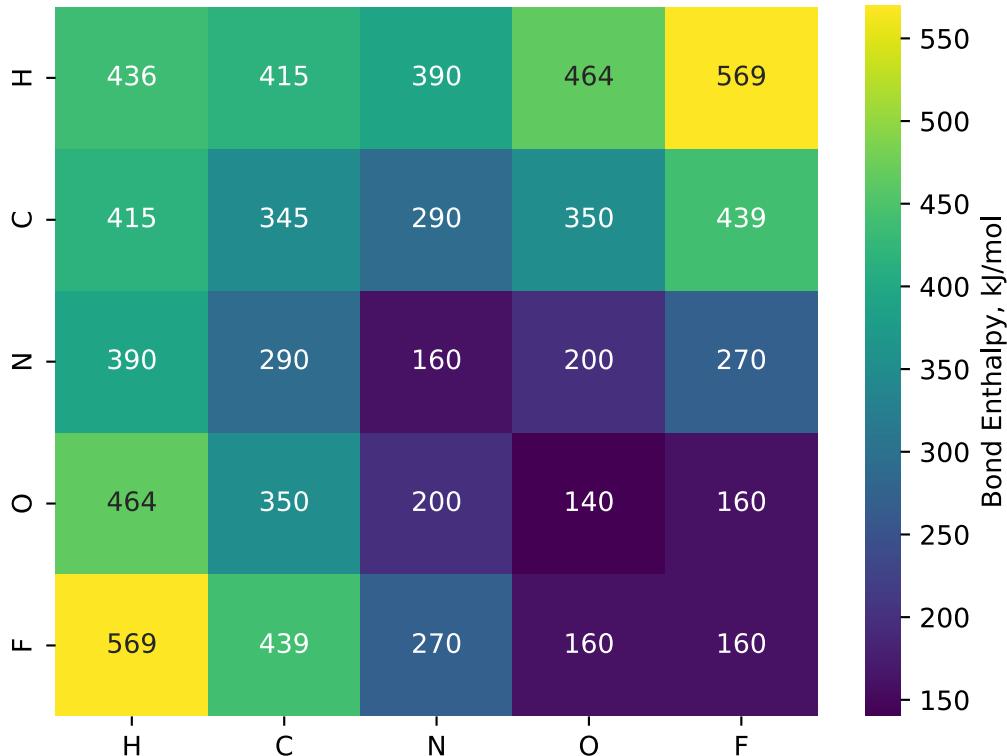
```
sns.heatmap(bde);
```



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 `heatmapt()` 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. To further customize the colorbar, use the `cbar_kws` argument that takes a dictionary of parameters found on the [matplotlib website](#). For example, to add a label, use the `label` key and the label text is the dictionary value as shown below.

```
sns.heatmap(bde, annot=True, fmt='d', cmap='viridis',
             cbar_kws={'label':'Bond Enthalpy, kJ/mol});
```



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

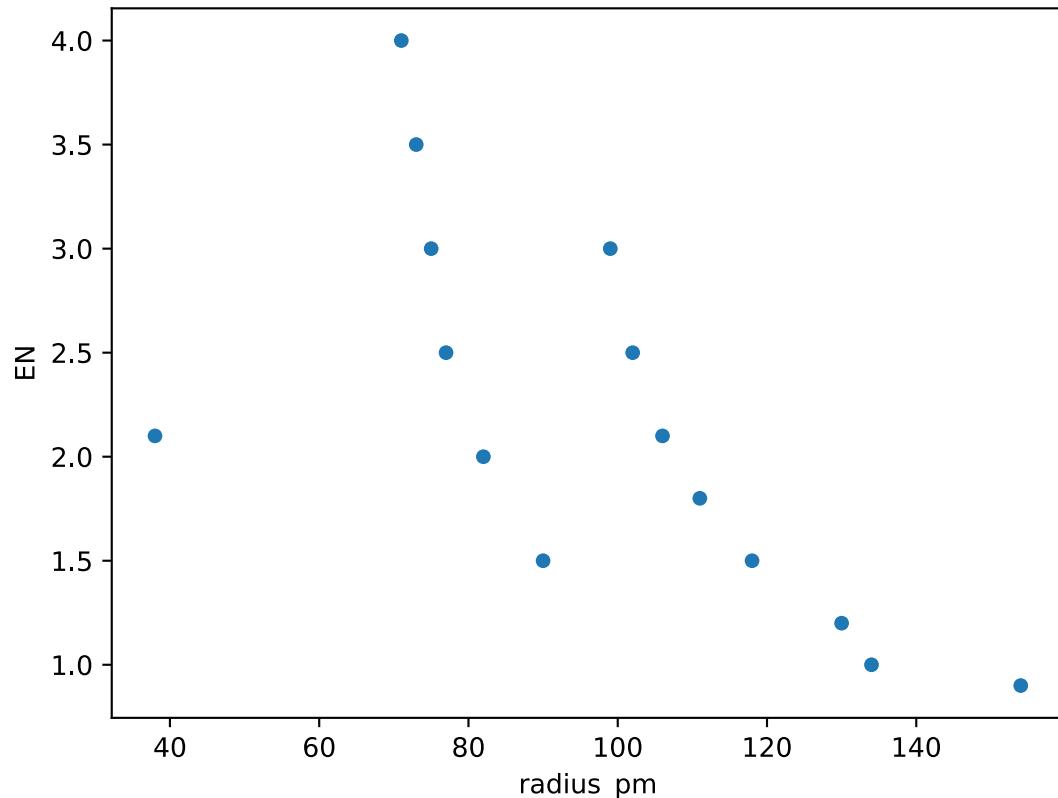
needs.

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.

```
sns.scatterplot(x='radius_pm', y='EN', data=per);
```

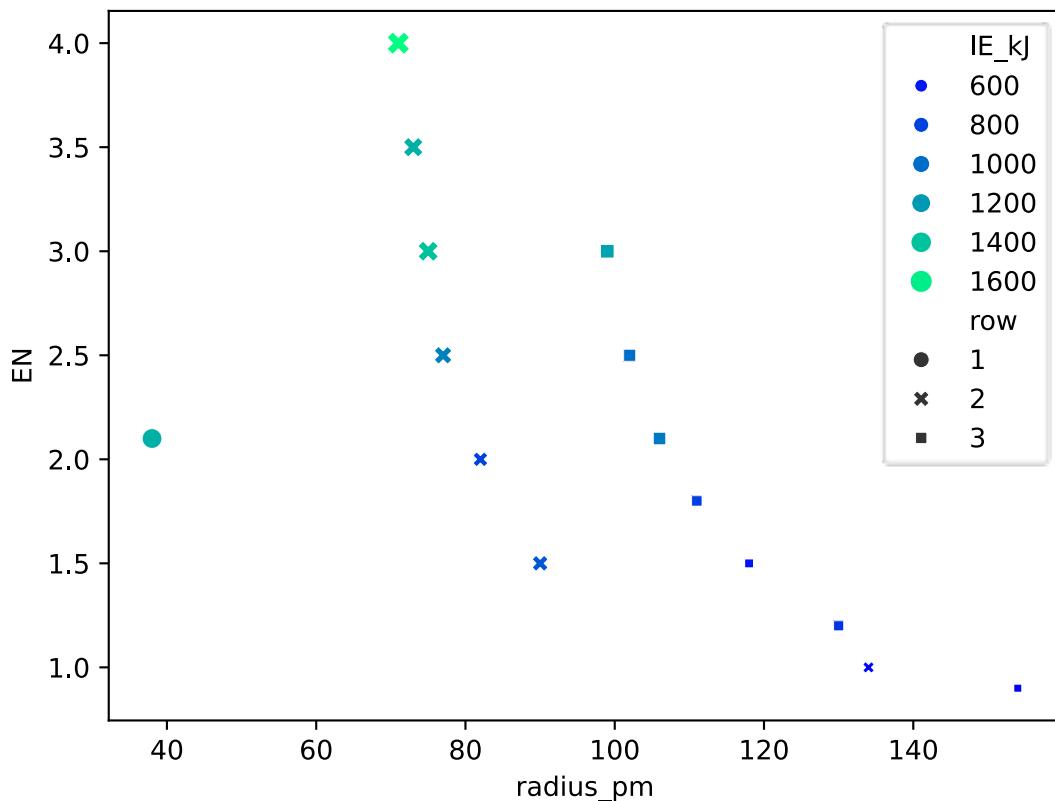


To modify the color, size, and marker style of the data points, use the `hue`, `size`, and

`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.

```
sns.scatterplot(x='radius_pm', y='EN', data=per, hue='IE_kJ',
                 size='IE_kJ', style='row', palette='winter');
```



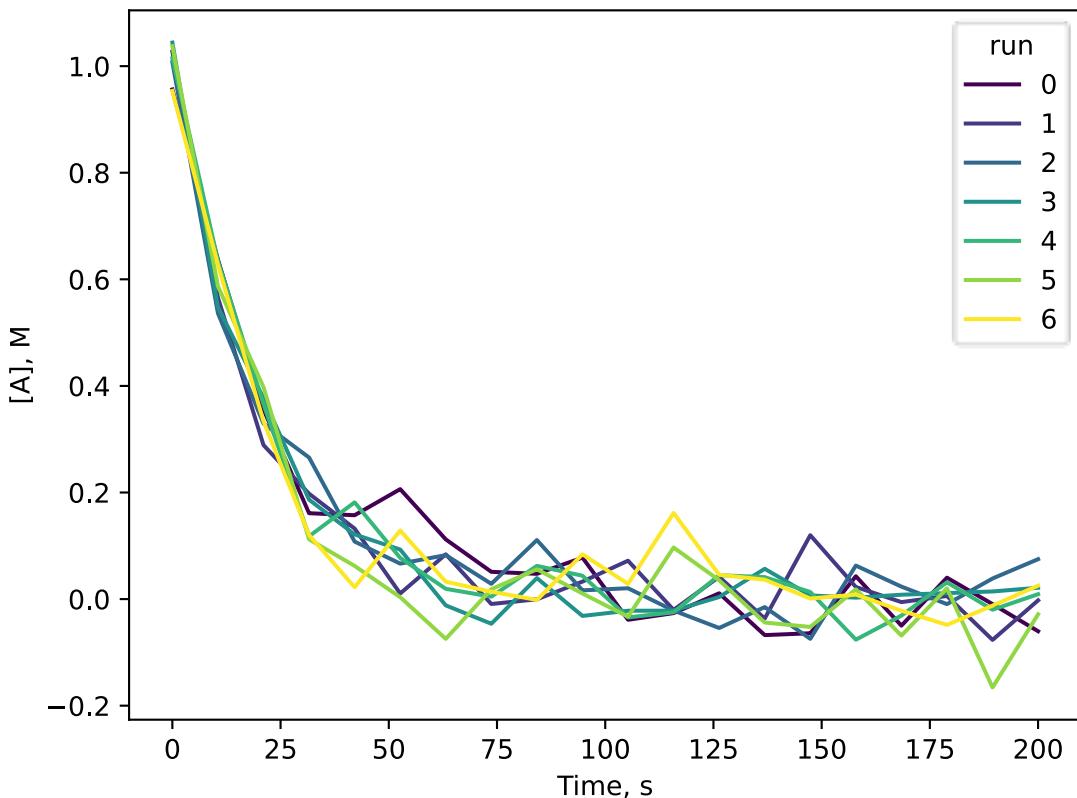
10.7.2 Line Plots

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 seven times (i.e., runs 0 → 6).

```
kinetics = pd.read_csv('data/kinetic_runs.csv')
kinetics.head()
```

	time	[A]	run	[P]
0	0.000000	0.956279	0.0	0.043721
1	10.526316	0.636978	0.0	0.363022
2	21.052632	0.355690	0.0	0.644310
3	31.578947	0.161173	0.0	0.838827
4	42.105263	0.157420	0.0	0.842580

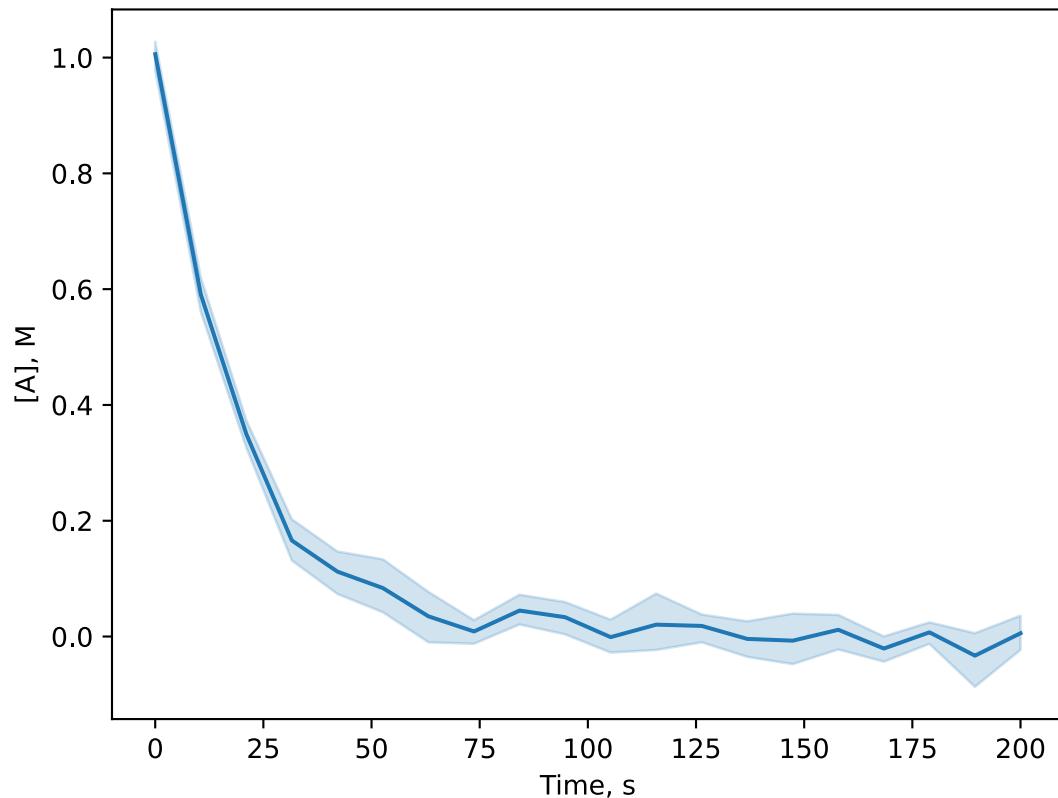
```
sns.lineplot(x='time', y='[A]', data=kinetics, hue='run', palette='viridis')
plt.xlabel('Time, s')
plt.ylabel('[A], M');
```



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 all the runs with some indication of the variation, the `lineplot()` function provides

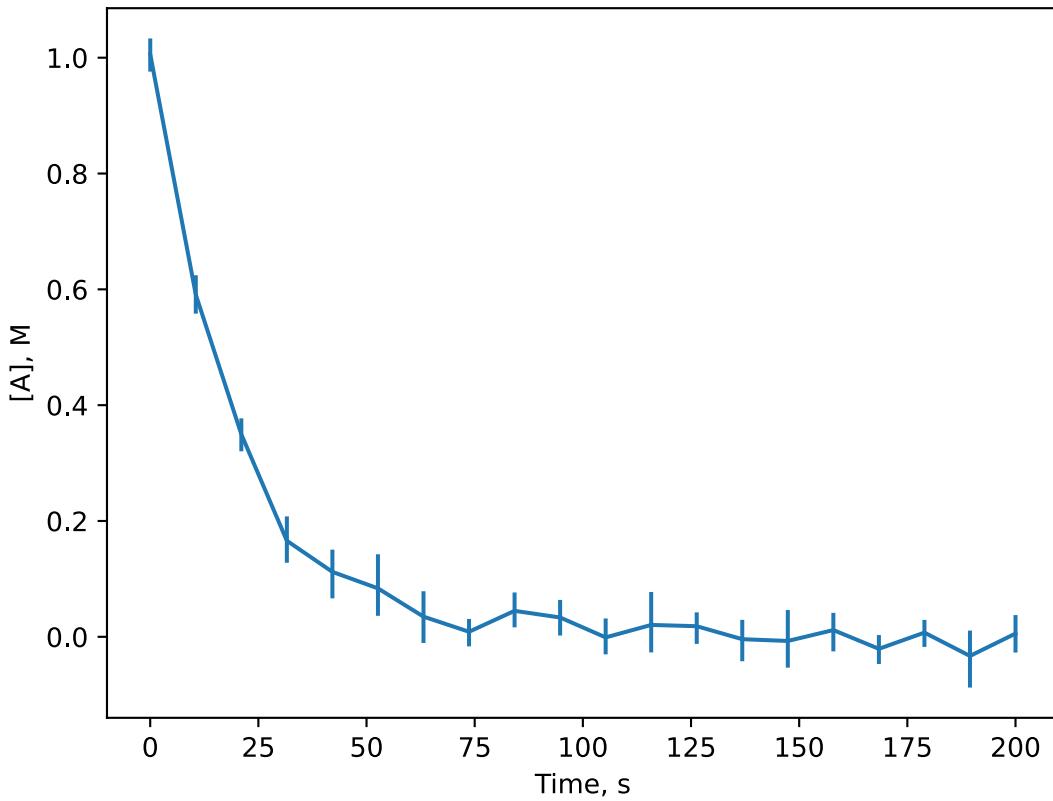
a default 95% confidence interval as is shown below.

```
sns.lineplot(x='time', y='[A]', data=kinetics)
plt.xlabel('Time, s')
plt.ylabel('[A], M');
```



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'`.

```
sns.lineplot(x='time', y='[A]', data=kinetics, err_style='bars')
plt.xlabel('Time, s')
plt.ylabel('[A], M');
```



10.8 Internal Datasets

Similar to a number of other Python libraries, seaborn brings with it datasets for users to experiment with. These are callable using the `sns.load_dataset()` function with the name of the dataset as the argument. Below is a table describing a few of the available Seaborn datasets. This list may change, so you can use the `sns.get_dataset_names()` to see the most current list.

Table 3 A Few Datasets Available in Seaborn

Name	Description
<code>anscombe</code>	Anscombe's quartet 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 used frequently in machine learning classification examples
<code>planets</code>	Information on discovered planets
<code>tips</code>	Restaurant information including bill total, tip, and information about the client
<code>titanic</code>	Titanic survivor dataset

Further Reading

1. Seaborn Website. <https://seaborn.pydata.org/> (free resource)

Exercises

Complete the following exercises in a Jupyter notebook and seaborn library. Any data file(s) referred to in the problems can be found in the [data](#) folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

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.
 - a) Separate the ketones and aldehydes values into individual Series.
 - b) Visualize the distribution of both ketone and aldehyde carbonyl stretches using a [kde plot](#).
4. Import the **elements_data.csv** file and generate a [count plot](#) 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) with respect to wavelength (λ) 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](#).
8. The following code generates the radial probability plot for hydrogen atomic orbitals for $n = 1-4$ (see [section 3.1](#)) and determines the radius of maximum probability (see [section 6.1.1](#)). These values are combined into a pandas DataFrame called [max_prob](#) where the rows are the principle quantum numbers and columns are the angular quantum numbers. Display the DataFrame using a heatmap. Your heatmap should include numerical labels on each colored block on the heatmap, and you should

select a non-default, perceptually uniform colormap for your colormap.

```
import numpy as np
import pandas as pd
import sympy
from sympy.physics.hydrogen import R_nl

R = sympy.symbols('R')
r = np.arange(0,60,0.01)

max_radii = []

for n in range(1,5):
    shell_max_radii = []
    for l in range(0, n):
        psi = R_nl(n, l, R)
        f = sympy.lambdify(R, psi, 'numpy')
        max = np.argmax(f(r)**2 * r**2)
        shell_max_radii.append(max/100)
    max_radii.append(shell_max_radii)

columns, index = (0,1,2,3), (4,3,2,1)
max_prob = pd.DataFrame(reversed(max_radii), columns=columns, index=index)
max_prob
```

Chapter 11: Plotting with Altair

Contents

- 11.1 Altair Plotting Basics
- 11.2 Panning & Zooming with `interactive()`
- 11.3 Data Types
- 11.4 Multifigure Plotting
- 11.5 Interactive Selections
- Further Reading

Note

This chapter is new and is still being reviewed. Typos may still exist and significant changes may occur.

Matplotlib can create nearly any plot you may need, but it often requires numerous lines of code to generate the desired result. Seaborn strives to remedy this by offering functions to create a series of common statistical plots types in only a few lines of code with excellent default colors and styles. Altair strives to be a middle ground by having the power of matplotlib while requiring shorter code than matplotlib. In addition, Altair includes the ability to interact with the plots such as panning, getting stats on highlighted data points, and informative dialogue boxes when hovering the cursor over a data point. While Altair has other virtues, it is the *interactive capabilities* that will be given special attention here.

If you are using your own installed version of Python, you can [install Altair using pip](#), and if you are using Colab, Altair is already installed. Altair is imported using the below command with the `alt` alias. Altair is designed to work with pandas, so [pandas](#) needs to also be imported.

Altair has a number of renderers for displaying your plots with the default behavior using a javascript frontend that requires an internet connection. If you are working offline or do not want Altair to reach out to the internet to assist in your plotting, the below command will make it work offline. There are other rendering options, but I find this works well while still maintaining the interactivity of Altair plots.

```
alt.renderers.enable('jupyter', offline=True)
```

11.1 Altair Plotting Basics

In the following example, we will visualize ligand cone angle data from [J. Am. Chem. Soc. 1975, 97, 7, 1955–1956](#) and [Chem. Rev. 1977, 77, 3, 313–348](#), so the data need to loaded into a pandas DataFrame.

```

ligands = pd.read_csv('data/cone_angles.csv', skipfooter=2, engine='python')
ligands.dropna(axis=0, inplace=True) # remove incomplete data rows
ligands.head()

```

	ligand	dH	cone_angle	CO_freq	type
2	P(OMe)3	-26.4	107	2079.5	P(OR)3
3	P(OCH2CH2Cl)3	-26.4	110	2083.2	P(OR)3
4	PMe3	-26.2	118	2064.1	PR3
5	P(OEt)3	-25.2	109	2076.3	P(OR)3
6	PMe2Ph	-25.0	127	2065.3	PR3

To generate a plot, we first need to create a `Chart` object using the `Chart()` function like below which accepts a pandas DataFrame. Most other customizations beyond this are done by concatenating a series of methods to the `Chart` object. The Chart object then needs to be instructed how to represent data points using one of the mark methods that follow the `mark_style()` pattern where the `style` is the marker type. The table below provides common options, but there are [additional options](#) as well.

Table 1 Common Altair Marker Methods

Chart Type	Description
<code>mark_point()</code>	Scatter plot
<code>mark_circle()</code>	Scatter plot using circle markers
<code>mark_line()</code>	Line plot
<code>mark_bar()</code>	Bar plot
<code>mark_rect()</code>	Heat map
<code>mark_area()</code>	Area plot
<code>mark_tick()</code>	Strip plot
<code>mark_rule()</code>	Verticle or horizontal line across entire Chart
<code>mark_arc()</code>	Pie, donut, radial, or polar bar plots
<code>mark_geoshape()</code>	Generate maps

The marks can be further customized, if desired, by providing the mark method extra keyword parameters such as those listed below.

Table 2 Select Mark Method Arguments

Marker Arguments	Description
<code>filled=</code>	Whether marker is filled or not (<code>True</code> or <code>False</code>)
<code>angle=</code>	Angle in degrees of marker
<code>opacity=</code>	Opacity ($0 \rightarrow 1$) of markers or line
<code>size=</code>	Size of markers (integer)
<code>color=</code>	Color (e.g., 'black') of line or marker
<code>shape=</code>	<u>Shape</u> of marker (e.g., 'triangle', 'square', 'circle', 'cross', 'wedge')

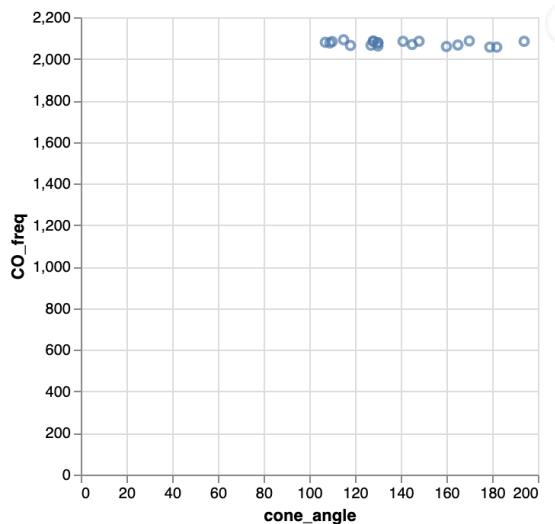
Below is a function call to make a scatter plot using the `mark_point()` method.

```
alt.Chart(ligands).mark_point()
```



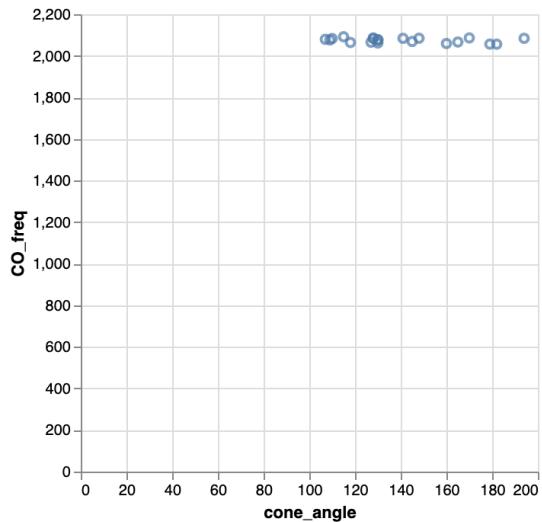
This returns only a dot because no indication has been given how to represent the information. This final piece of information is known as the **encoding** or **encoding channel** and is assigned using the `encode()` method. In the example below, the cone angle is encoded or represented by the location on the x-axis using the `x=` parameter and carbonyl (i.e., M-C≡O) stretching frequency by the position on the y-axis using the `y=` parameter. Because the `Chart` object already has the DataFrame name, the `x=` and `y=` arguments only need the DataFrame column names.

```
alt.Chart(ligands).mark_point().encode(
    x='cone_angle',
    y='C0_freq')
```



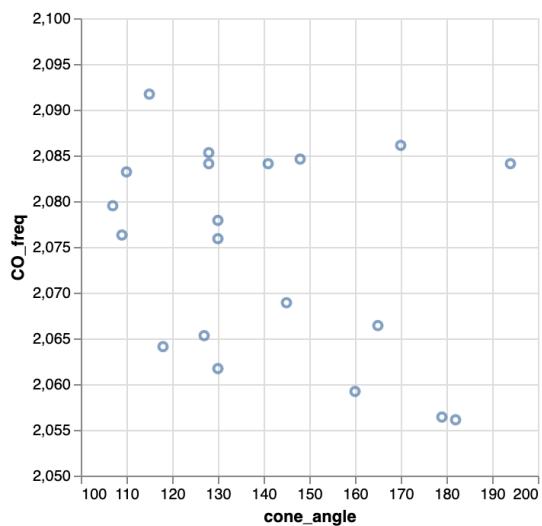
By default, Altair includes zero on the axes, so it will be necessary to adjust the ranges for both axes in this example. To accomplish this, first replace the `x=` and `y=` shorthand notation with `alt.X()` and `alt.Y()` which gives more control.

```
alt.Chart(ligands).mark_point().encode(
    alt.X('cone_angle'),
    alt.Y('CO_freq')
)
```



Then apply the `scale()` method with the `domain=` parameter to customize the plotting domains.

```
alt.Chart(ligands).mark_point().encode(
    alt.X('cone_angle').scale(domain=[100, 200]),
    alt.Y('CO_freq').scale(domain=[2050, 2100])
)
```



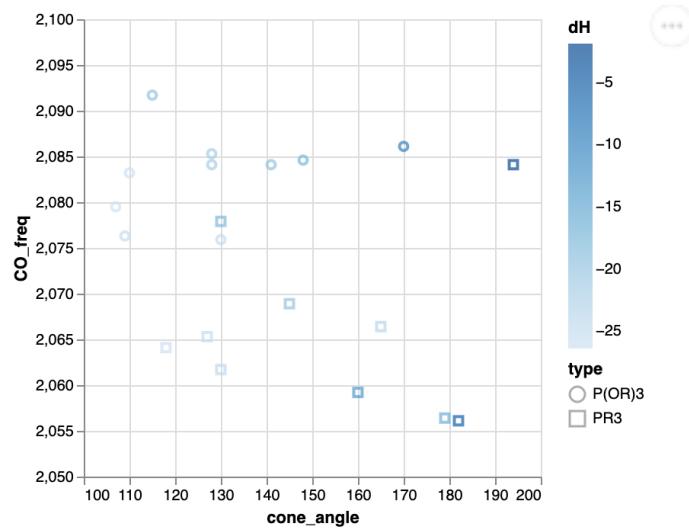
Along with the x- and y-axis positions, information can be encoded or represented using other visual indicators such as color, size, shape, etc. Below is a table of some key encodings with others listed on the [Altair website](#).

Table 3 Common Encoding Channels in Altair

Encoding	Description
<code>x</code> or <code>alt.X()</code>	Position on x-axis
<code>y</code> or <code>alt.Y()</code>	Position on y-axis
<code>color</code> or <code>alt.Color()</code>	Marker color
<code>shape</code> or <code>alt.Shape()</code>	Marker shape
<code>size</code> or <code>alt.Size()</code>	Marker size
<code>opacity</code> or <code>alt.Opacity()</code>	Opacity of the marker
<code>column</code> or <code>alt.Column()</code>	Separates plots along x-axis
<code>row</code> or <code>alt.Row()</code>	Separates plots along y-axis
<code>tooltip</code>	Dialogue box with information

For example, the chart below represents the ΔH values using the color and the type of ligand with the marker shape.

```
alt.Chart(ligands).mark_point().encode(
    alt.X('cone_angle').scale(domain=[100, 200]),
    alt.Y('CO_freq').scale(domain=[2050,2100]),
    alt.Color('dH'),
    alt.Shape('type')
)
```

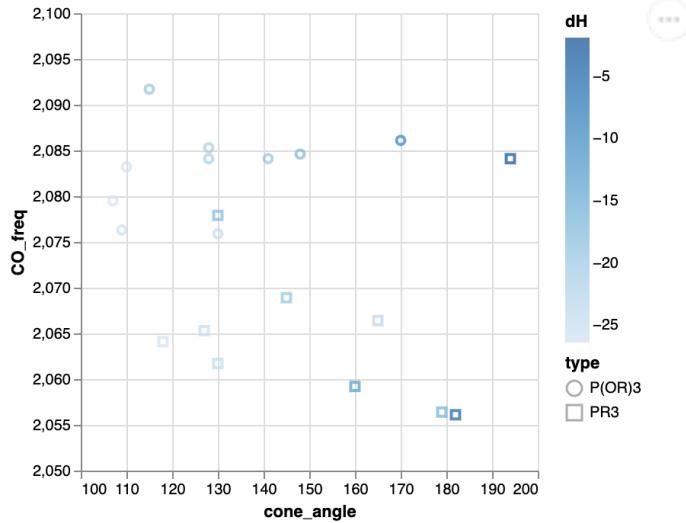


Another way to provide access to information is through a dialogue box. Just include a list or tuple with the DataFrame column information to be included in the tooltip box. Below, the user will see a small popup box with the ligand name, enthalpy, and carbonyl frequencies whenever they hover their cursor over the marker on the plot.

```

alt.Chart(ligands).mark_point().encode(
    alt.X('cone_angle').scale(domain=[100, 200]),
    alt.Y('CO_freq').scale(domain=[2050,2100]),
    alt.Color('dH'),
    alt.Shape('type'),
    tooltip=['ligand', 'dH', 'CO_freq']
)

```

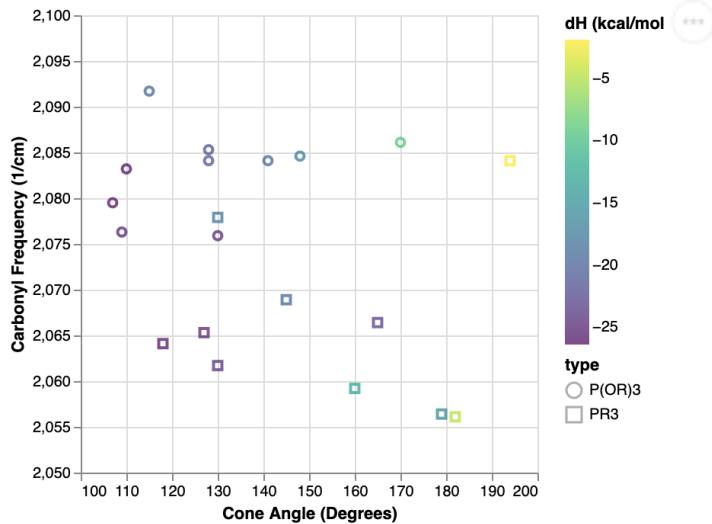


We now have a fairly reasonable plot, but further customization is often necessary. For example, better axis labels with units would be ideal which can be added using the `title()` method on each encoding channel. If you don't like the colormap, this can be set with the `scheme=` argument in the color encoding channel.

```

alt.Chart(ligands).mark_point().encode(
    alt.X('cone_angle').scale(domain=[100, 200]).title('Cone Angle (Degrees)'),
    alt.Y('CO_freq').scale(domain=[2050,2100]).title('Carbonyl Frequency (1/cm)'),
    alt.Color('dH').scale(scheme='viridis').title('dH (kcal/mol)'),
    alt.Shape('type'),
    tooltip=['ligand', 'dH', 'CO_freq']
)

```



Tip

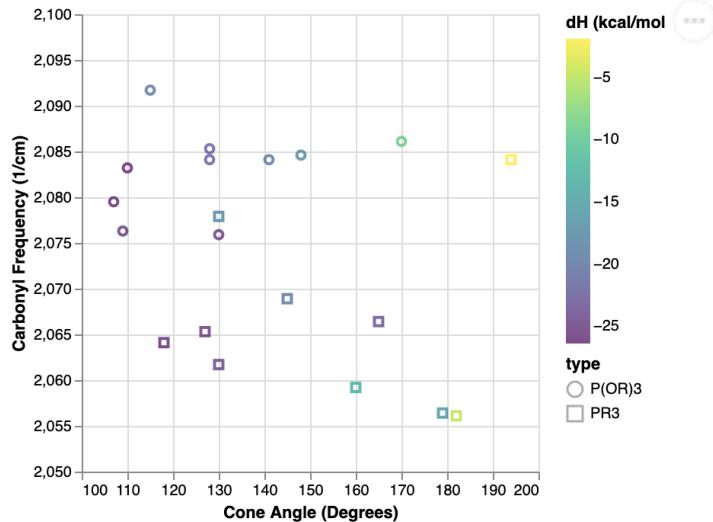
If you get an error while trying to save your plot, you may be missing an optional dependency. See [Altair website](#) for installation instructions.

As a final step for our first Altair graph, we can save it with either the (...) menu on the top right or by using the `save()` method. Like matplotlib, if no format is specified, it is deduced from the file name.

```
c = alt.Chart(ligands).mark_point().encode(
    alt.X('cone_angle').scale(domain=[100, 200]).title('Cone Angle (Degrees)'),
    alt.Y('CO_freq').scale(domain=[2050, 2100]).title('Carbonyl Frequency (Wavenumbers)'),
    alt.Color('dH').scale(scheme='viridis').title('dH (kcal/mol)'),
    alt.Shape('type')
)
c.save('first_altair_plot.pdf', format='pdf')
```

11.2 Panning & Zooming with `interactive()`

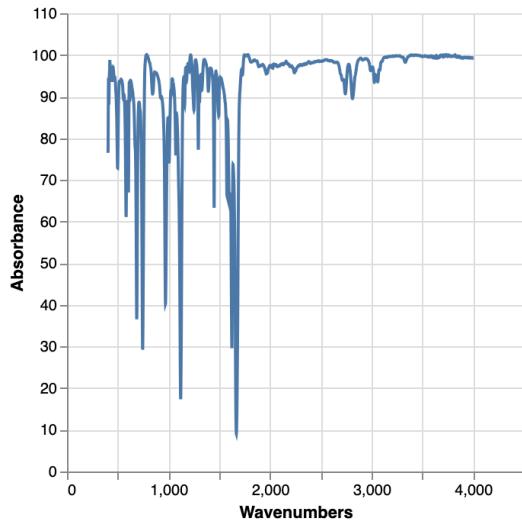
One of the major advantages of Altair over seaborn and matplotlib is the ability to interact with plots in Altair. This can take many forms, the most basic of which is the ability to pan and zoom. This can be enabled by adding the `interactive()` method to a `Chart` object. Now by dragging and scrolling, the user can pan and zoom the plot, respectively. Double click to reset the plot.



As an additional example, we will plot the IR spectrum of *trans*-cinnamaldehyde.

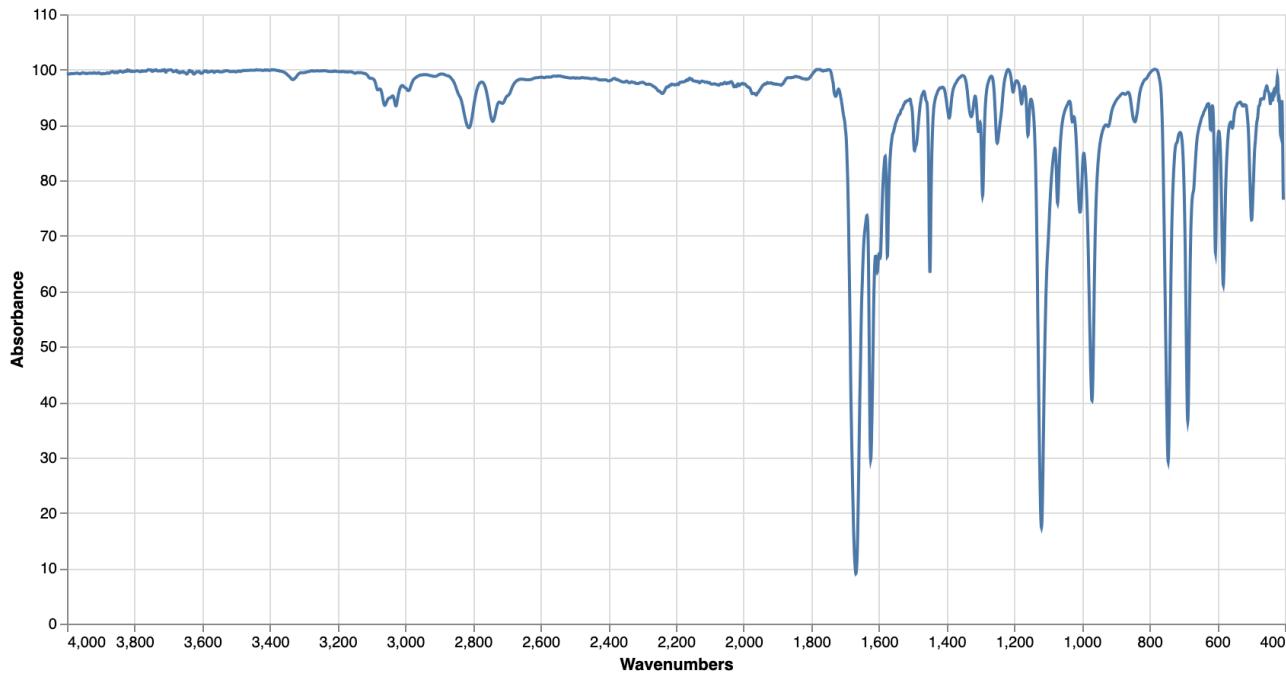
```
# load IR data of trans-cinnamaldehyde
tcinn = pd.read_csv('data/tcinn.CSV', delimiter=',', header=None)
tcinn.columns = ['Wavenumbers', 'Absorbance']
tcinn = tcinn.round({'Wavenumbers': 1, 'Absorbance': 2})
```

```
alt.data_transformers.enable("vegafusion")
alt.Chart(tcinn).mark_line().encode(
    x='Wavenumbers',
    y='Absorbance'
)
```



We now see our plot, but it's a bit tiny for an IR spectrum. The plot size can be adjusted using the `properties()` method and including the `width=` and `height=` arguments.

```
alt.Chart(tcinn).mark_line().encode(
    alt.X('Wavenumbers').scale(domain=(4000, 400)),
    y='Absorbance'
).properties(width=800, height=400)
```

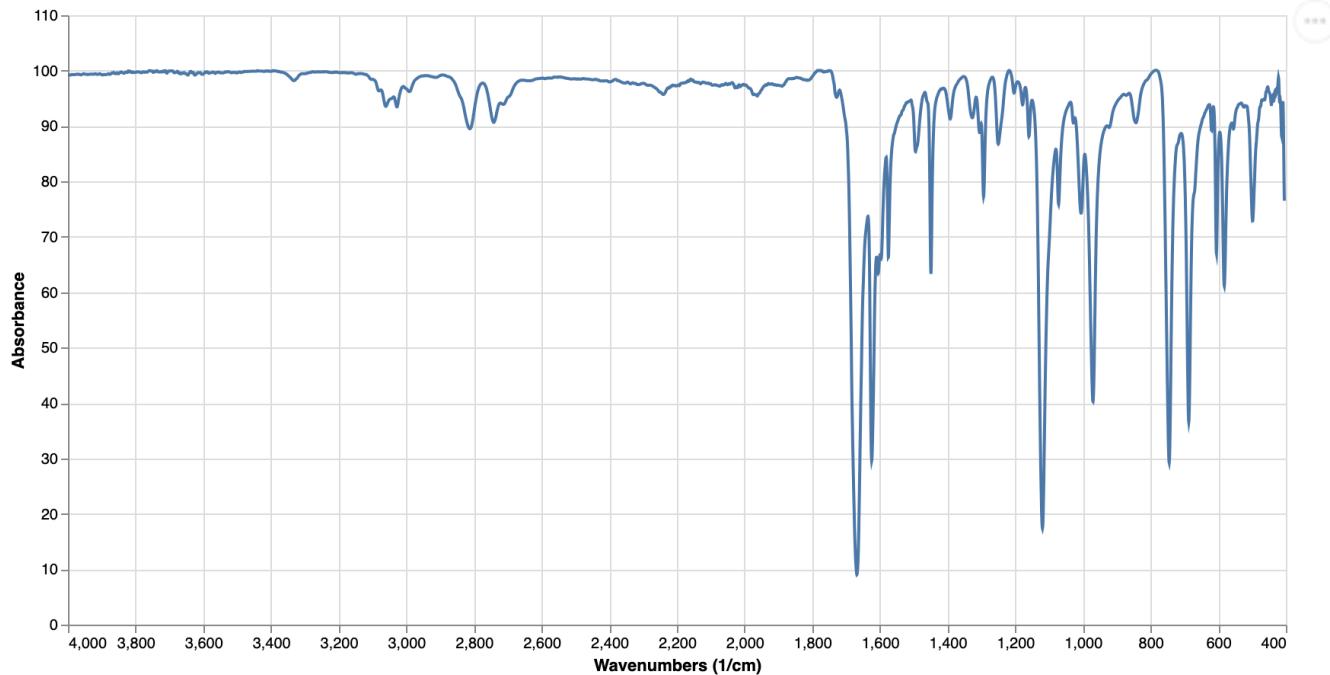


The x-axis can be reversed by either setting the `domain=(14, -1)` or by setting `reverse=True` in the `scale()` method. The plot below is also made interactive by again appending `.interactive()` and adding a `tooltip`. The nice thing about making a spectrum interactive like this is the ability to pan, zoom, and identify the frequencies of various absorbances.

```

alt.Chart(tcinn).mark_line().encode(
    alt.X('Wavenumbers').scale(domain=(4000, 400)).title('Wavenumbers (1/cm)'),
    y='Absorbance',
    tooltip=['Wavenumbers', 'Absorbance']
).properties(width=800, height=400).interactive()

```



11.3 Data Types

Altair will make a best effort to guess the data type (Table 4) and plot the data appropriately. For example, if the data are numerical values, Altair treats the values as continuous quantitative features, so it plots the data along a continuous axis with markings anywhere along the axis. Alternatively, if the data are strings, Altair treats them as nominal data which is categorical in no particular order.

Table 4 Altair Data Types

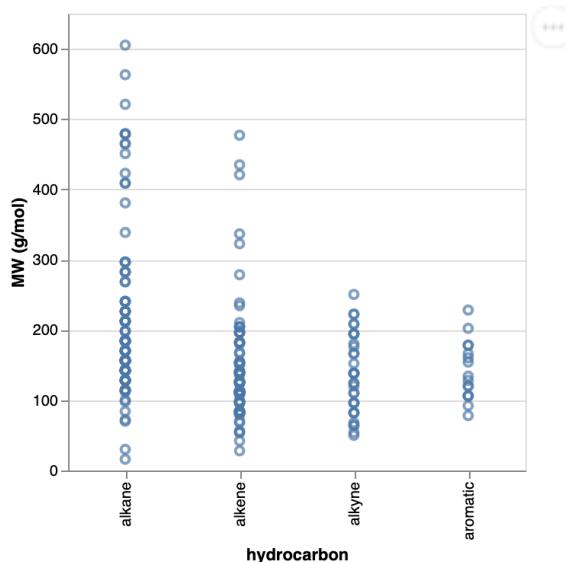
Data Type	Abbreviation	Description	Examples
Quantitative	:Q	Continuous numerical data	Densities or chemical shift
Nominal	:N	Unordered, non-continuous data	Glassware type or functional group
Ordinal	:O	Ordered, non-continuous data	Months or degree of alcohol
Time	:T	Date or time values	Date/Time data was collected
Geojson	:G	Geographical information	Location sample was collected or country of origin

For example, when plotting the molecular weight (`MW`) versus hydrocarbon type (`hydrocarbon`), Altair automatically treats the molecular weight as quantitative and the hydrocarbon type as nominal data like below.

```
HC = pd.read_csv('data/hydrocarbon.csv')
HC.head()
```

	bp	MW	EOU	hydrocarbon
0	574.0	238.46	1	alkene
1	356.0	82.15	2	alkene
2	565.0	226.45	0	alkane
3	330.0	82.15	2	alkyne
4	457.0	156.31	0	alkane

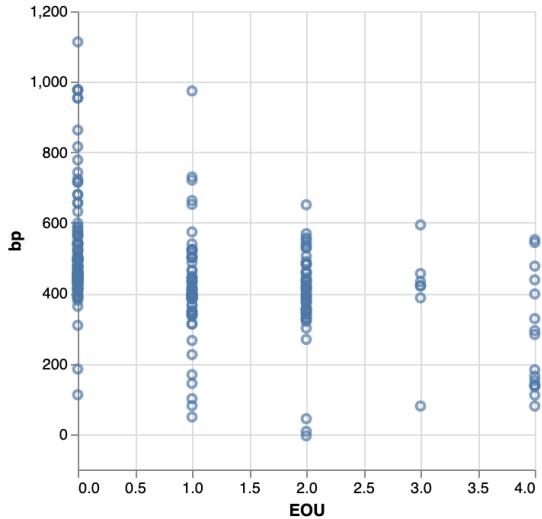
```
alt.Chart(HC).mark_point().encode(
    alt.X('hydrocarbon'),
    alt.Y('MW').title('MW (g/mol)')
).properties(width=300)
```



The user can override these default data types by either appending the data type abbreviation (Table 4) to the DataFrame header string or by setting `type=` to one of the types. One common situation is when categories are designated by numbers, which is used in machine learning datasets. For example, the hydrocarbon data includes the elements of unsaturation (`EOU`) for various hydrocarbons. Altair's default behavior is to treat the degree as continuous and quantitative which leads to the following result.

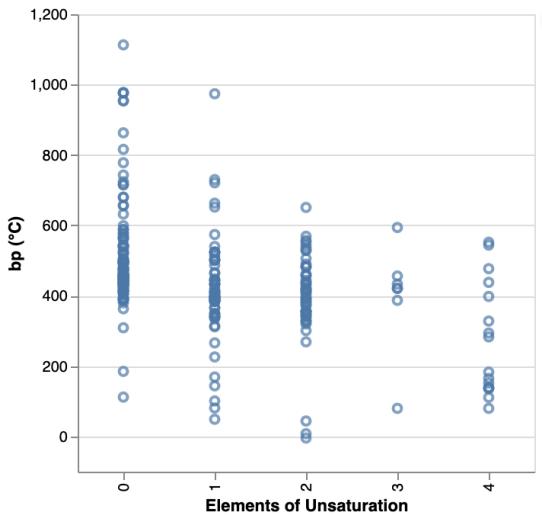
```
low_EOU = HC[HC['EOU'] <= 4]

alt.Chart(low_EOU).mark_point().encode(
    alt.X('EOU'),
    alt.Y('bp'))
```



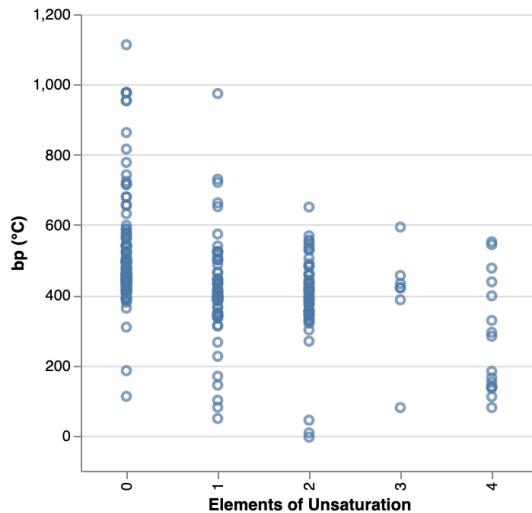
This is not really what we want because there are non-integer markings and the 4 is up against the edge of the plot. If we append `:0` to the `EOU`, this tells Altair to treat elements of unsaturation as nominal values, which are ordered but not continuous.

```
alt.Chart(low_EOU).mark_point().encode(
    alt.X('EOU:0').title('Elements of Unsaturation'),
    alt.Y('bp').title('bp (°C)')
).properties(width=300)
```



Alternatively, we can add `type='ordinal'` to get the same result.

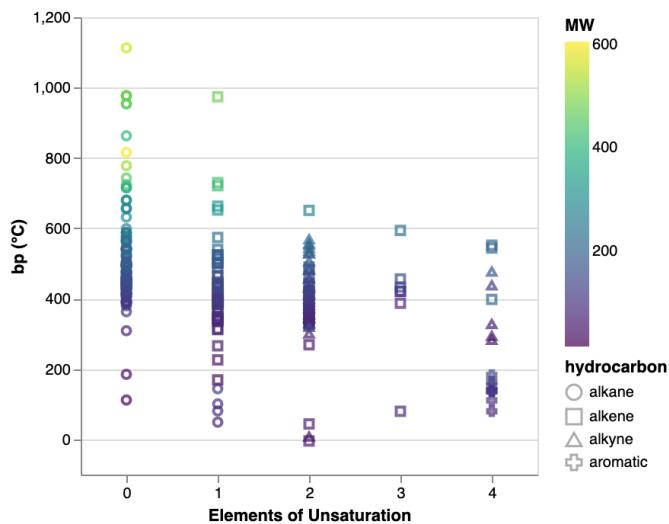
```
alt.Chart(low_EOU).mark_point().encode(
    alt.X('EOU', type='ordinal').title('Elements of Unsaturation'),
    alt.Y('bp').title('bp (°C)')
).properties(width=300)
```



Now we only get integer markings while the values are still in order.

The chart can be further customized like changing the angle of the axis labels, colors and shape or markers, and making the chart interactive.

```
alt.Chart(low_EOU).mark_point().encode(
    alt.X('EOU:0', axis=alt.Axis(labelAngle=0)).title('Elements of Unsaturation'),
    alt.Y('bp').title('bp (°C)'),
    alt.Color('MW').scale(scheme='viridis'),
    alt.Shape('hydrocarbon')
).properties(width=300).interactive()
```



11.4 Multifigure Plotting

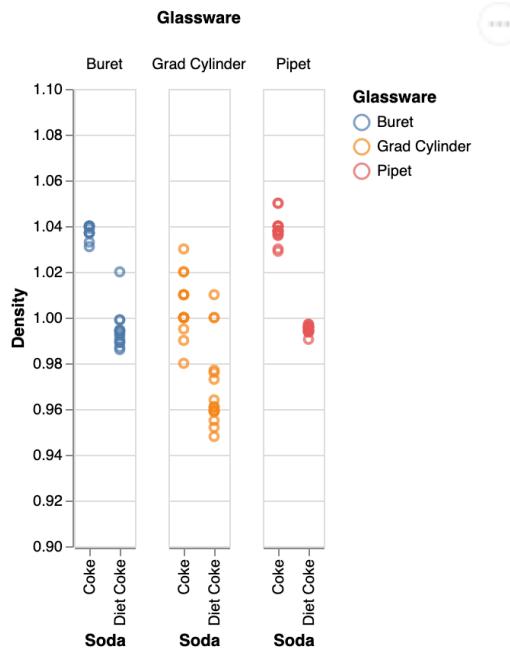
Altair supports the display of faceted figures. While the figures could be created using separate code cells, there are interactivity advantages to displaying them together (see [section 11.5](#)). In this example, we will display the density of degassed Coke and Diet Coke using different types of glassware for measuring the volume.

```
soda = pd.read_csv('data/soda.csv')
soda.head()
```

	Density	Glassware	Soda
0	1.00	Grad Cylinder	Coke
1	1.00	Grad Cylinder	Coke
2	0.99	Grad Cylinder	Coke
3	1.00	Grad Cylinder	Coke
4	0.98	Grad Cylinder	Coke

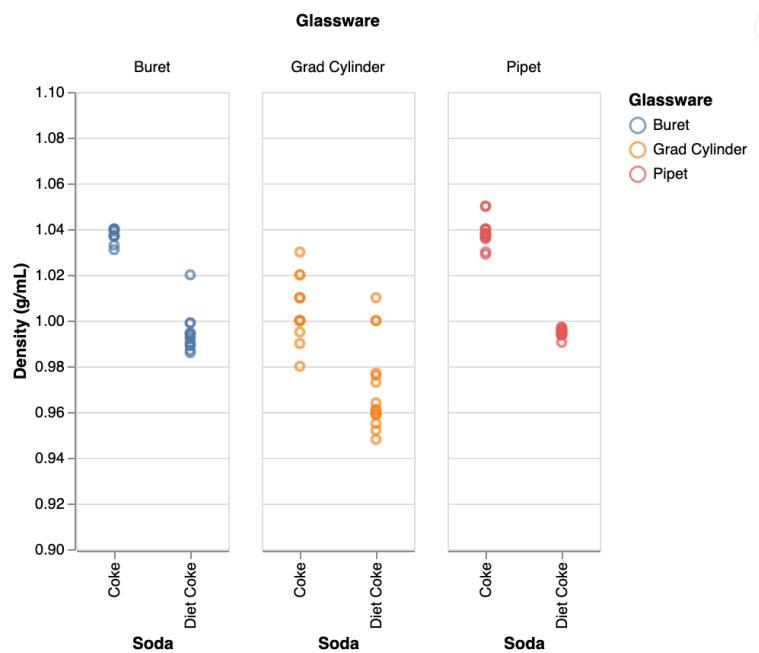
In the first example below, we use the `Column` encoding to represent the data for different glassware types. This results in what looks like three different figures that share the same y-axis label. If we were to instead use `Row` encoding, the three sections would instead be rows and share the same x-axis label.

```
alt.Chart(soda).mark_point().encode(
    alt.Y('Density').scale(domain=(0.9, 1.1)),
    alt.Column('Glassware'),
    x='Soda',
    color='Glassware')
```



This figure is a bit narrow, so we can adjust the dimensions again using `.properties(width=100)` which sets the width of each section of the graph.

```
alt.Chart(soda).mark_point().encode(
    alt.Y('Density').title('Density (g/mL)').scale(domain=(0.9, 1.1)),
    x='Soda',
    column='Glassware',
    color='Glassware').properties(width=100)
```



Another way to generate two or more figures or plots together is to concatenate or overlay them. This is accomplished by assigning two different charts to variables and using either `&`, `|`, or `+` (Table 5). Alternatively, the functions in Table 5 can be used by providing them with the Chart objects.

Table 5 Layered and Multifigured Plots

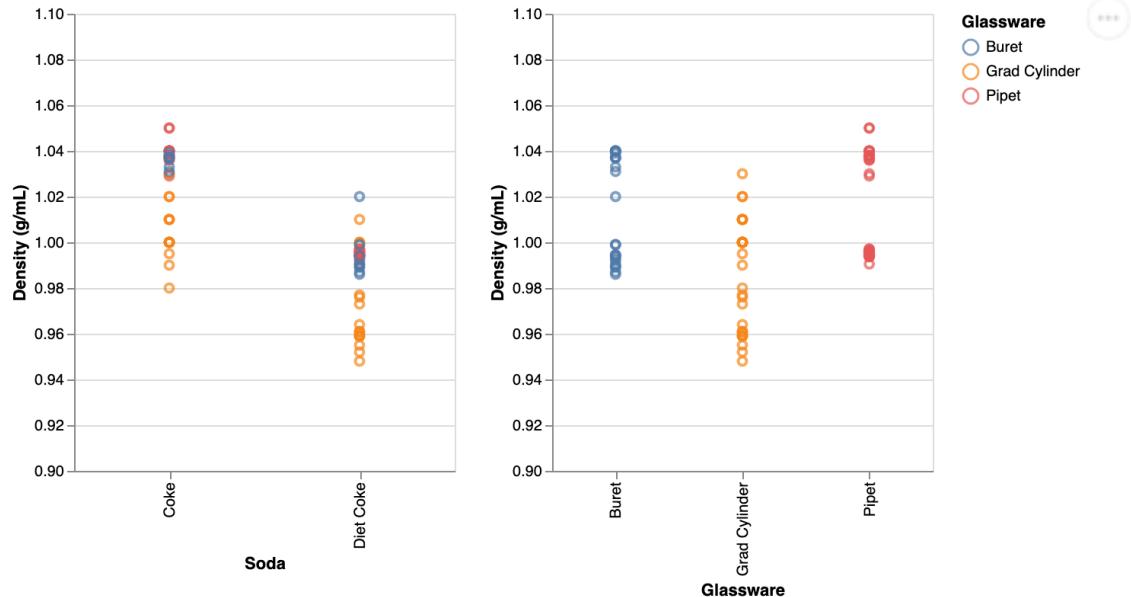
Operator	Function	Description
<code>&</code>	<code>alt.hconcat()</code>	Horizontal concatenation
<code> </code>	<code>alt.vconcat()</code>	Vertical concatenation
<code>+</code>	<code>alt.layer()</code>	Overlay two plots on top of each other

Below, two scatter plots are created with density on the y-axis and different categories on the x-axes. The figures are then horizontally concatenate.

```
chart1 = alt.Chart(soda).mark_point().encode(
    alt.Y('Density').title('Density (g/mL)').scale(domain=(0.9, 1.1)),
    x='Soda',
    color='Glassware').properties(width=250)

chart2 = alt.Chart(soda).mark_point().encode(
    alt.Y('Density').title('Density (g/mL)').scale(domain=(0.9, 1.1)),
    x='Glassware',
    color='Glassware').properties(width=250)

chart1 | chart2
```

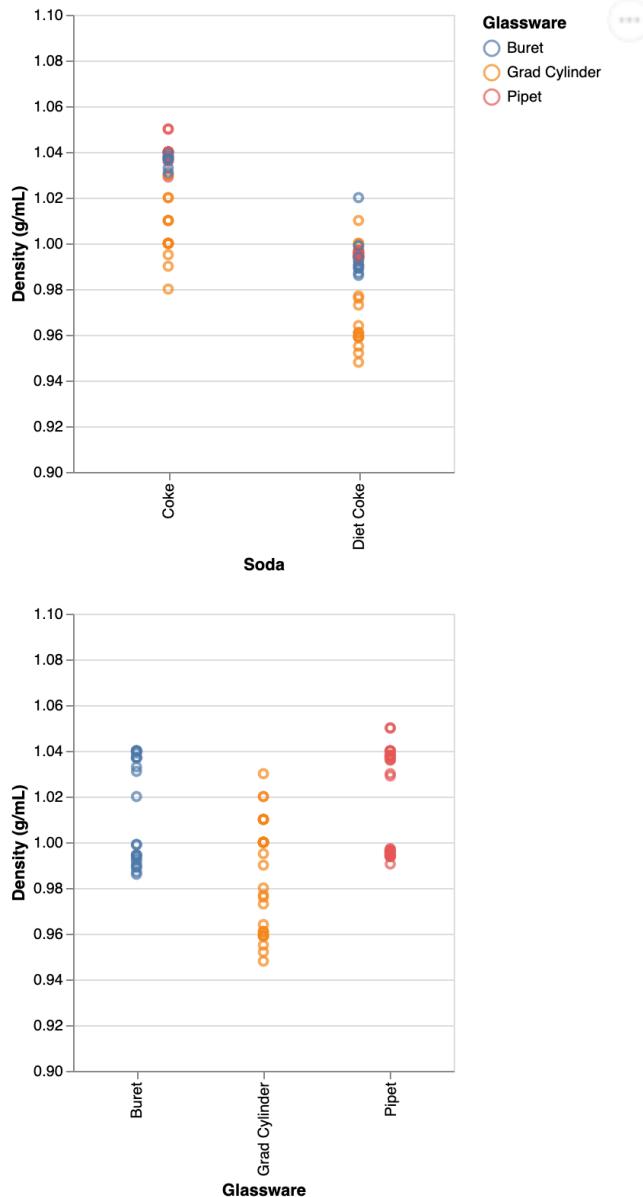


We could instead perform vertical concatenation like below. This is more useful when one plot is narrow like a small bar graph.

```
chart1 = alt.Chart(soda).mark_point().encode(
    alt.Y('Density').title('Density (g/mL)').scale(domain=(0.9, 1.1)),
    x='Soda',
    color='Glassware').properties(width=250)

chart2 = alt.Chart(soda).mark_point().encode(
    alt.Y('Density').title('Density (g/mL)').scale(domain=(0.9, 1.1)),
    x='Glassware',
    color='Glassware').properties(width=250)

chart1 & chart2
```



The overlay option () is useful for plotting more than one type of plot on the same axes like a line and scatter plot as we have often done in [chapter 3](#). An example of this is in the following section.

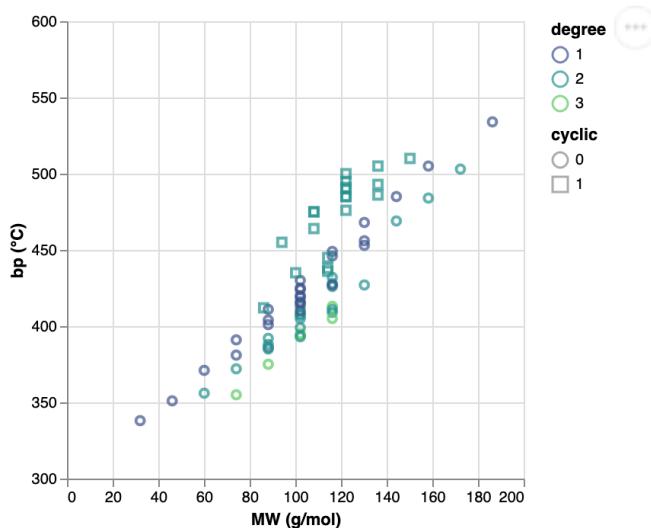
11.5 Interactive Selections

Another form of interactivity supported by Altair is to allow the user to select portions of a graph and see information about the selection such as averages, sums, and distributions. For this section, we will start by looking at a dataset with alcohol features.

	bp	MW	carbons	degree	aliphatic	avg_aryl_position	cyclic	
0	338	32.04		1	1		0.0	0
1	351	46.07		2	1		0.0	0
2	371	60.10		3	1		0.0	0
3	356	60.10		3	2		0.0	0
4	391	74.12		4	1		0.0	0

Below, the boiling point, molecular weight, degree, and whether the alcohol is cyclic (1) or non-cyclic (0) are visualized.

```
alt.Chart(ROH).mark_point().encode(
    alt.Y('bp').scale(domain=[300, 600]).title('bp (°C)'),
    alt.X('MW').scale(domain=[0, 200]).title('MW (g/mol)'),
    alt.Color('degree:0').scale(scheme='viridis'),
    alt.Shape('cyclic:N'))
```

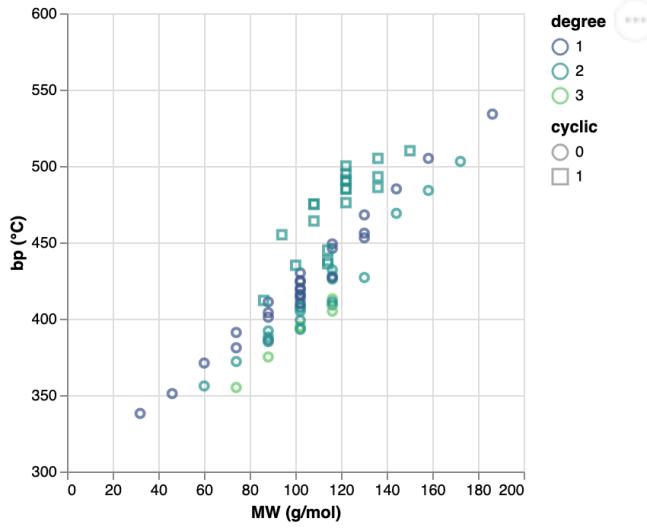


Altair allows the users to box select data points by adding an interval selection parameter using the `alt.selection_interval()` function. This selection parameter is added to the Chart through the `.add_params()` method. By default, this is a box selection which allows the user to select a rectangle anywhere in the plot. If `encodings=['x']` or `encodings=['y']` parameters are added to the `selection_interval()` function, the selection is restricted along the x- or y-axes, respectively.

```
# box selection object
selection = alt.selection_interval()

points = alt.Chart(ROH).mark_point().encode(
    alt.Y('bp').scale(domain=[300, 600]).title('bp (°C)'),
    alt.X('MW').scale(domain=[0, 200]).title('MW (g/mol)'),
    alt.Color('degree:0').scale(scheme='viridis'),
    alt.Shape('cyclic:N'))
).add_params(selection)

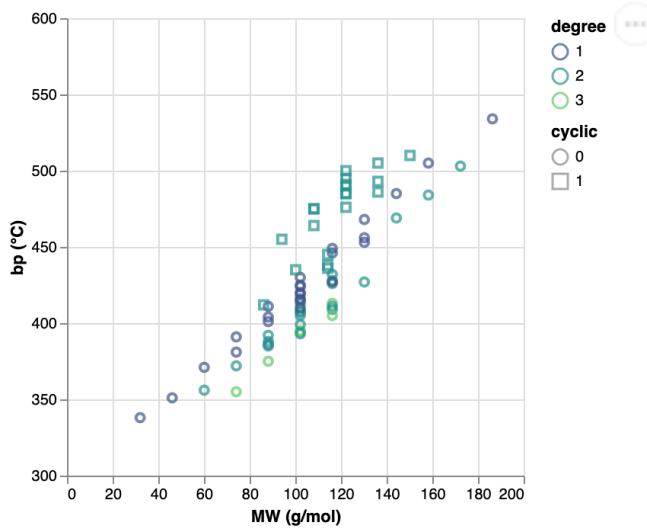
points
```



```
# X selection object
selection = alt.selection_interval(encodings=['x'])

points = alt.Chart(ROH).mark_point().encode(
    alt.Y('bp').scale(domain=[300, 600]).title('bp (°C)'),
    alt.X('MW').scale(domain=[0, 200]).title('MW (g/mol)'),
    alt.Color('degree:O').scale(scheme='viridis'),
    alt.Shape('cyclic:N')
).add_params(selection)

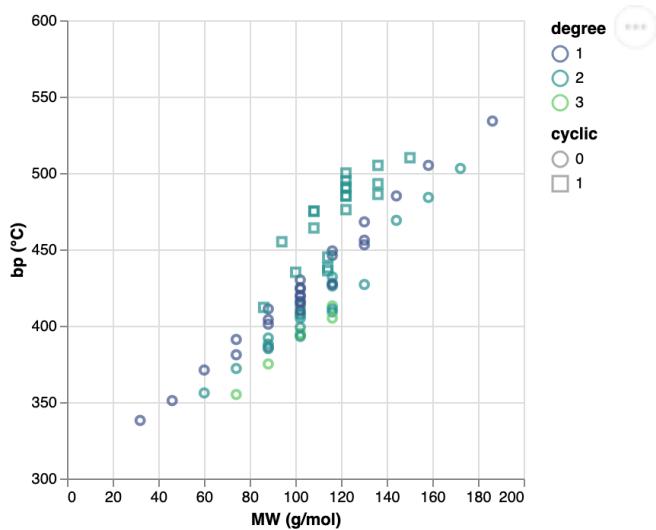
points
```



```
# Y selection object
selection = alt.selection_interval(encodings=['y'])

points = alt.Chart(ROH).mark_point().encode(
    alt.Y('bp').scale(domain=[300, 600]).title('bp (°C)'),
    alt.X('MW').scale(domain=[0, 200]).title('MW (g/mol)'),
    alt.Color('degree:O').scale(scheme='viridis'),
    alt.Shape('cyclic:N')
).add_params(selection)

points
```



The user is now able to select regions of the Chart, which is stored in the `selection` variable. This does not really do anything except make a gray box until this information is passed to another function like is done below.

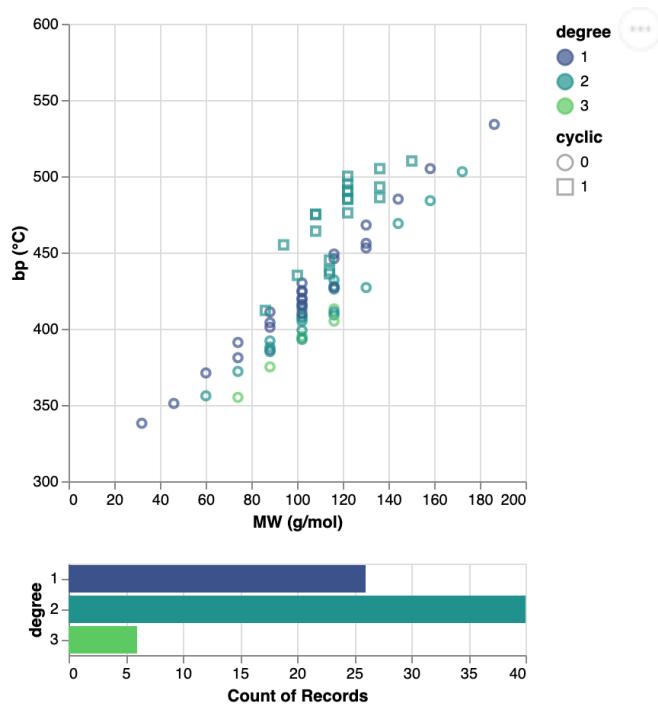
In the plot below, two Chart objects are created - one scatter plot and one bar plot. These Charts are stacked on top of each other using the `&` operator (bottom line). The selection object is added to the scatter plot using `add_params()`, like is done above while the selection object is provided to the bar plot through the `transform_filter()` function. This setup makes it so the scatter plot is where the user selects regions and the bar plot is the recipient of this selection information. Finally, notice that the bar plot x-variable contains a `count()` function instead of a DataFrame column header. This processes the selection information and uses this for the bar graph. Specifically, the bar graph here shows the total number of primary, secondary, and tertiary alcohols selected in the scatter plot.

```
# box and bar select together
selection = alt.selection_interval()

# scatter plot
scatter = alt.Chart(ROH).mark_point().encode(
    alt.Y('bp').scale(domain=[300, 600]).title('bp (°C)'),
    alt.X('MW').scale(domain=[0,200]).title('MW (g/mol)'),
    alt.Color('degree:0').scale(scheme='viridis'),
    alt.Shape('cyclic:N')
).add_params(selection)

# bar plot
bar = alt.Chart(ROH).mark_bar().encode(
    x='count()',
    y='degree:0',
    color='degree:0'
).transform_filter(selection)

# stack scatter and bar plot
scatter & bar
```



Another example below is a bar graph of the radial probability of the hydrogen 3p atomic orbital. Like above, there are two Chart objects - one bar graph and a rule or line that spans the entire Chart. Instead of stacking these Charts, they are overlaid using the `+` operator. The bar graph is provided the selection object through the `add_params()` method allowing the user to select regions in this Chart. The rule Chart accepts the selection through the `transform_filter()` method making it the recipient of the selection information. Similar to the above example, the y-axis is given a function, `mean()`, which takes the average of the selected probabilities and sets the horizontal bar to this value. The end result is a bar plot where the user can select a region and see a horizontal line marking the average probability of the selected region.

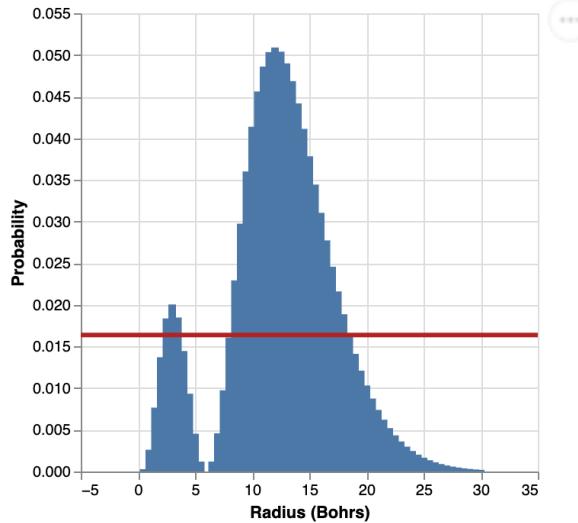
```
prob = pd.read_csv('data/prob_3p_normalized.csv')

selection = alt.selection_interval(encodings=['x'])

bar = alt.Chart(prob).mark_bar().encode(
    x=alt.X('Radius').title('Radius (Bohrs)'),
    y=alt.Y('Probability').title('Probability'),
).add_params(selection)

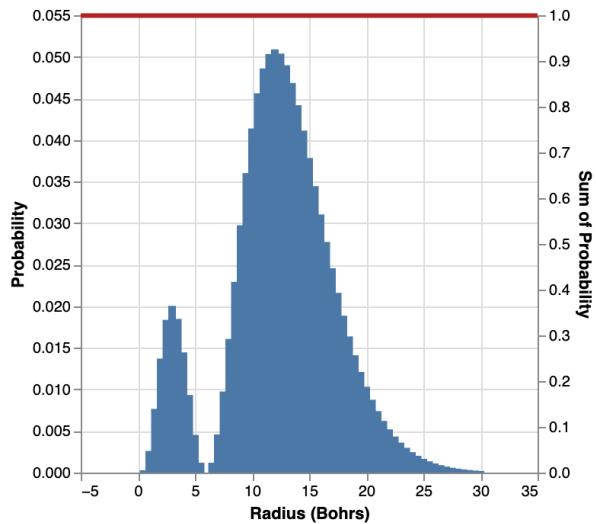
rule = alt.Chart(prob).mark_rule(color='firebrick').encode(
    y='mean(Probability)',
    size=alt.SizeValue(3)
).transform_filter(selection)

bar + rule
```



Below is a modified version of the previous graphic where instead of taking the mean of the selected region, the sum is calculated. This effectively allows the user to graphically integrate different regions of the graph. For example, by selecting the region just below the node, it can be seen that this region constitutes a little over 10% of the probability.

The two Charts are overlayed using the `alt.layer()` function instead of the `+` operator to allow more control so that a second y-axis could be added showing the sum of the selected probabilities.



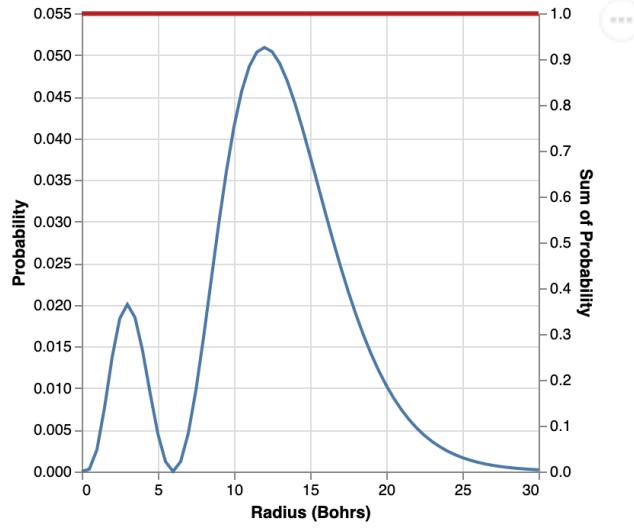
Finally, the above plot can be converted from a bar graph to a line plot by changing `mark_bar()` to `mark_line()`.

```
selection = alt.selection_interval(encodings=['x'])

bar = alt.Chart(prob).mark_line().encode(
    x=alt.X('Radius').title('Radius (Bohrs)'),
    y=alt.Y('Probability').title('Probability'),
).add_params(selection)

rule = alt.Chart(prob).mark_rule(color='firebrick').encode(
    y=alt.Y('sum(Probability)').scale(domain=(0, 1)),
    size=alt.SizeValue(3)
).transform_filter(
    selection
)

alt.layer(bar, rule, data=prob).resolve_scale(y='independent')
```



Further Reading

The best source of up-to-date information on Altair is the Altair website. Because Altair is much newer than matplotlib and seaborn, there are fewer resources currently available.

1. Altair website. <https://altair-viz.github.io/> (free resource)

Official Altair website and documentation page.

Chapter 12: Nuclear Magnetic Resonance with nmrglue and nmrsim

Contents

- 12.1 NMR Processing with nmrglue
- 12.2 Simulating NMR with nmrsim
- Further Reading
- Exercises

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 library called nmrglue to does exactly this. The library contains modules for dealing with data from each of the major NMR spectroscopy file types which includes Bruker, Pipe, Sparky, and Varian. It does not read JEOL files, but as of this writing, JEOL spectrometers support exporting data into at least one of the above file types supported by nmrglue and direct support for JEOL files is under development.

In addition, it is also sometimes helpful to be able to simulate NMR spectra to confirm spectral parameters (e.g., coupling constants), visualize hypothetical spectra of splitting patterns, or fit the line shapes or splitting patterns of experimental data. The library nmrsim provides the ability to simulate NMR spectra including dynamic NMR and is introduced in [section 12.2](#).

12.1 NMR Processing with nmrglue

Currently, nmrglue is not included with default installation of Anaconda, so you will need to install it. Instructions are included on the [nmrglue documentation page](#) or you can use `pip` to install it. If Jupyterlab is installed on your computer, you should be able to install it through the terminal using `pip install nmrglue`, and if you are using Google Colab, you should include `!pip install nmrglue` in the first code cell of the notebook (see [section 0.2](#)). nmrglue requires you have NumPy and SciPy installed, and matplotlib should also be installed for visualization.

All use of code below assumes the following imports with aliases. 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.

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

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 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 to 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 1, right). This is what is known as converting the *time domain* to the *frequency domain*.

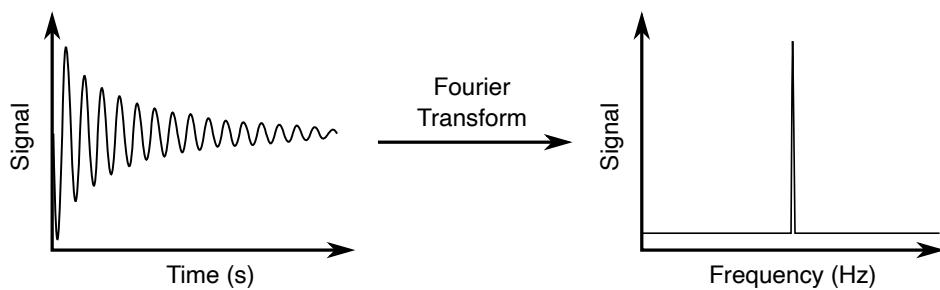


Figure 1 Raw NMR spectroscopy data is converted from the time domain (left) to the frequency domain (right) using Fourier transform.

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

12.1.1 Importing Data with nmrglue

The importing of data using nmrglue is performed by the read function from one of the submodules shown in Table 1. Additional modules can be found in the [nmrglue documentation](#). The choice of module is dictated by the data file type.

Table 1 Examples of nmrglue Modules

Module	Description
<code>bruker</code>	Bruker data as a single file
<code>pipe</code>	Pipe data as a single file with an <code>.fid</code> extension
<code>sparky</code>	Sparky NMR file format with <code>.ucsf</code> extension
<code>varian</code>	Varian/Agilent data as a folder of data with an <code>.fid</code> extension
<code>jcampdx</code>	JCAMP-DX files with <code>.dx</code> or <code>.jdx</code> extensions

The `read()` function loads the NMR file and returns a tuple containing a dictionary of metadata and data in a NumPy array. The dictionary includes information required to complete the processing of the NMR data. Looking at the NMR data shown below, you may have 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 the data.

```
dic, data = ng.pipe.read('data/EtPh_1H_NMR_CDCl3.fid')
data
```

```
array([-0.00194889-0.00471539j, -0.00192186-0.00472489j,
       -0.00191337-0.00473085j, ..., -0.00189737+0.00591656j,
       -0.00191882+0.005872j, -0.00191135+0.00587132j],
      shape=(13107,), dtype=complex64)
```

```
# Reversed the Fourier transform for demo purposes being as this data
# was collected on a spectrometer that already Fourier transformed the data.

from scipy.fft import ifft
data = ifft(data)[::-1]
```

The dictionary, `dic`, above 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.

```
udic = ng.pipe.guess_udic(dic, data)
udic
```

```
{'ndim': 1,
 0: {'sw': 5994.65478515625,
 'complex': True,
 'obs': 399.7821960449219,
 'car': 1998.9109802246094,
 'size': 13107,
 'label': 'Proton',
 'encoding': 'direct',
 'time': False,
 'freq': True}}
```

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 2 below provides a description of each

piece of metadata contained in the universal dictionary.

Table 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.

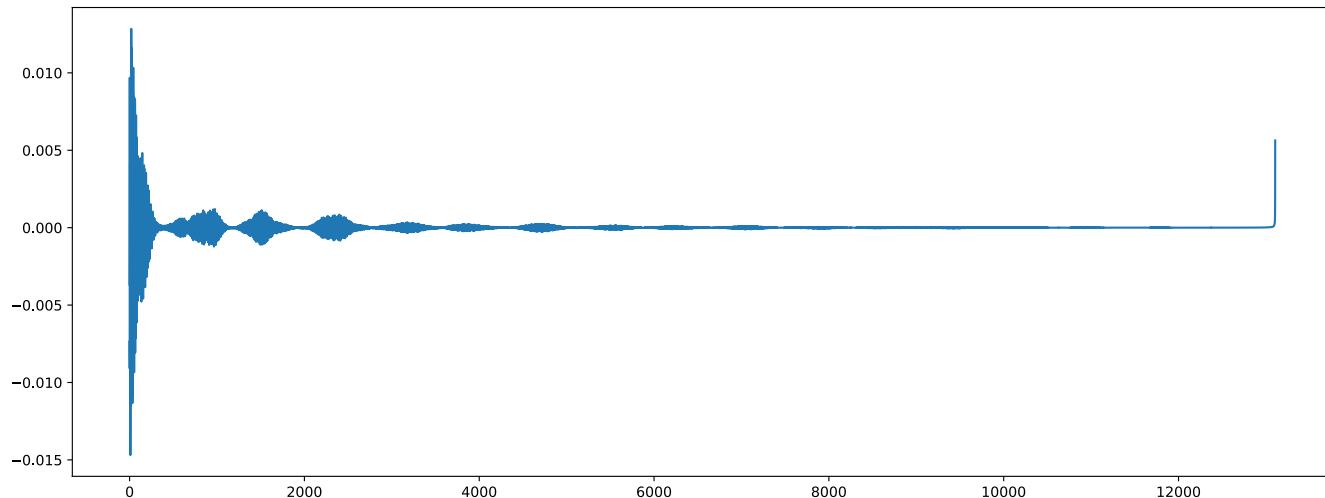
12.1.2 Fourier Transforming Data

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` like below.

```
udic[0]['time']
```

We can also view the data by plotting with matplotlib.

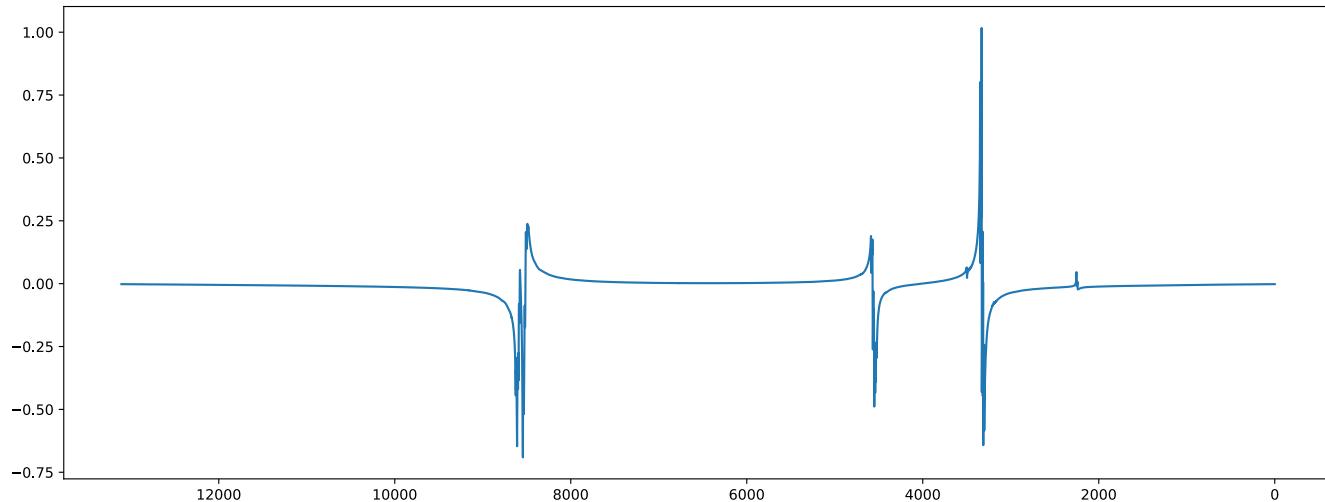
```
fig0 = plt.figure(figsize=(16, 6))
ax0 = fig0.add_subplot(1, 1, 1)
ax0.plot(data.real);
```



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()` to conform to NMR plotting conventions.

```
from scipy.fft import fft
fdata = fft(data)
```

```
fig1 = plt.figure(figsize=(16, 6))
ax1 = fig1.add_subplot(1, 1, 1)
ax1.plot(fdata.real)
plt.gca().invert_xaxis() # reverses direction of x-axis to conform to NMR plotting norms
```



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.

12.1.3 Phasing Data

Phasing is the post-processing procedure for making all peaks point upward as shown in Figure 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.

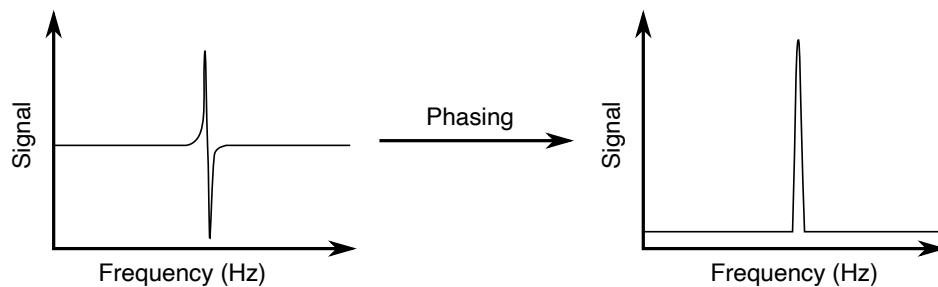


Figure 2 Phasing an NMR spectrum results in all the signals pointing in the positive direction.

12.1.3.1 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.

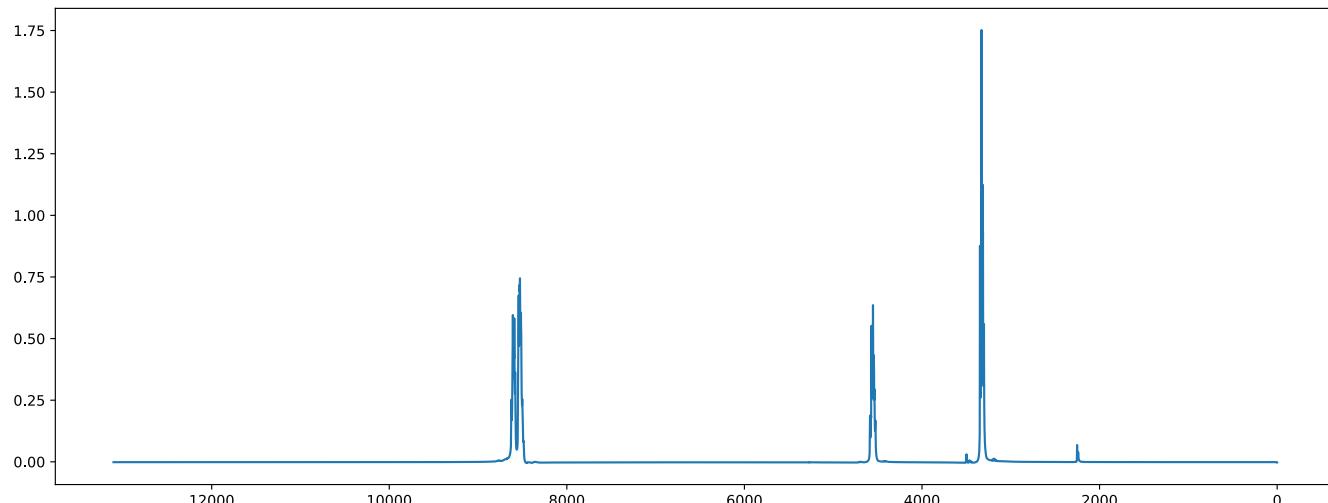
```
ng.process.proc_autophase.autops(data, algorithm)
```

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.

```
phased_data = ng.process.proc_autophase.autops(fdata, 'acme')
```

```
Optimization terminated successfully.  
Current function value: 0.001729  
Iterations: 117  
Function evaluations: 236
```

```
fig2 = plt.figure(figsize=(16, 6))  
ax2 = fig2.add_subplot(1, 1, 1)  
ax2.plot(phased_data.real)  
plt.gca().invert_xaxis()
```

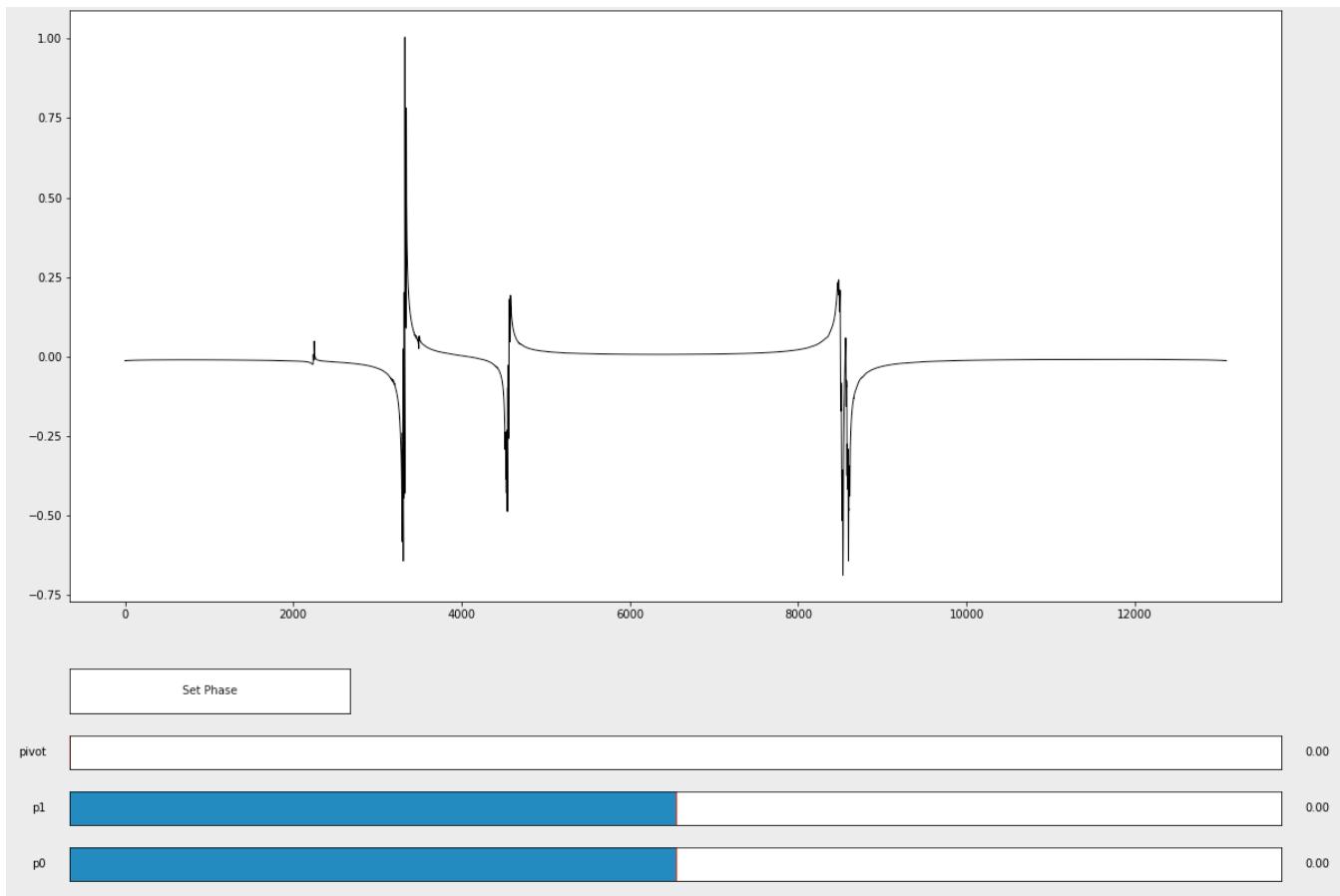


You should try both algorithms to see which works best for you. The above spectrum is the result of the `acme` autophasing algorithm which is close but still slightly off. If neither of the provided autophasing algorithms work for you, you will need to instead manually phase the NMR spectrum as discussed below.

12.1.3.2 Manual Phasing

Manually phasing the NMR spectrum is a two-step process. First, you need to call the `manual_ps()` phasing function and adjust the `p0` and `p1` sliders until the spectrum appears phased.

```
%matplotlib # exists inline plotting
p0, p1 = ng.proc_autophase.manual_ps(fdata.real)
```



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.

```
phased_data = ng.proc_base.ps(fdata, p0=p0, p1=p1)

%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.

12.1.4 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.

12.1.4.1 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 observed chemical shift (Hz) versus a standard over the spectrometer frequency (MHz) at which that particular nucleus is observed.

$$ppm = \frac{Observed\ Frequency\ (Hz)}{Spectrometer\ Frequency\ (MHz)}$$

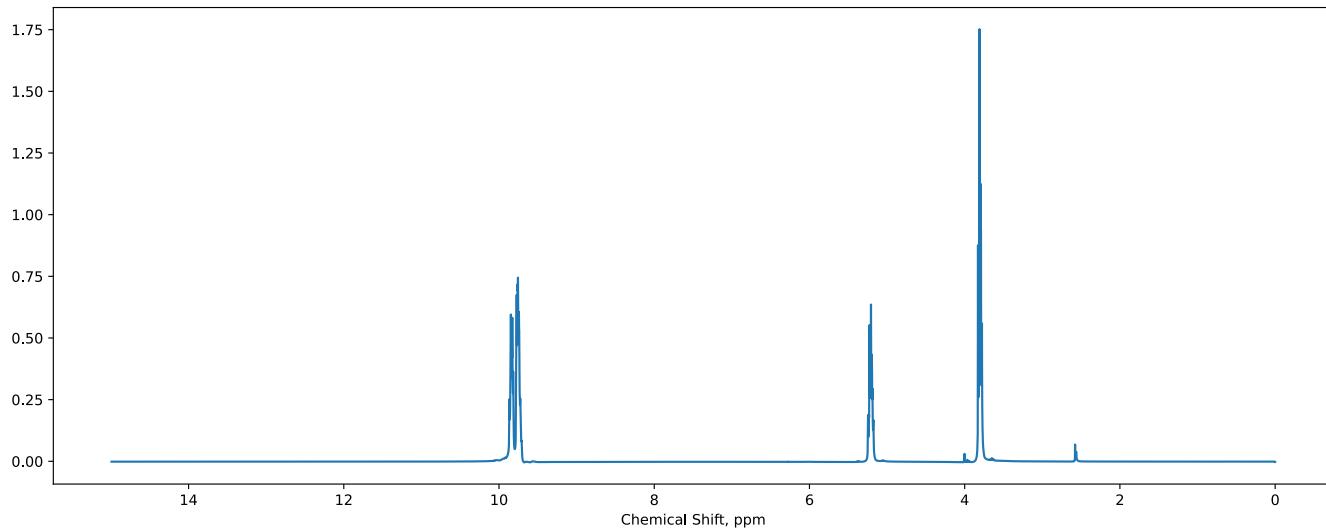
This makes the locations of the peaks consistent from spectrometer to spectrometer no matter the strength of the magnet. This is where the `udic` from [section 12.1.1](#) is important because we can obtain the observed frequency width (Hz) of the spectrum, and the resolution of the data. The latter is how many data points are in the spectrum which is important so that we avoid a plotting error (we all 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.

```
size = udic[0]['size'] # points in data
sw = udic[0]['sw']      # width in Hz
obs = udic[0]['obs']    # carrier frequency
```

```
from math import floor
hz = np.linspace(0, floor(sw), size) # x-axis in Hz
ppm = hz / obs                    # x-axis in ppm
```

Now if we plot the spectrum, we see it in a ppm scale.

```
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()
```



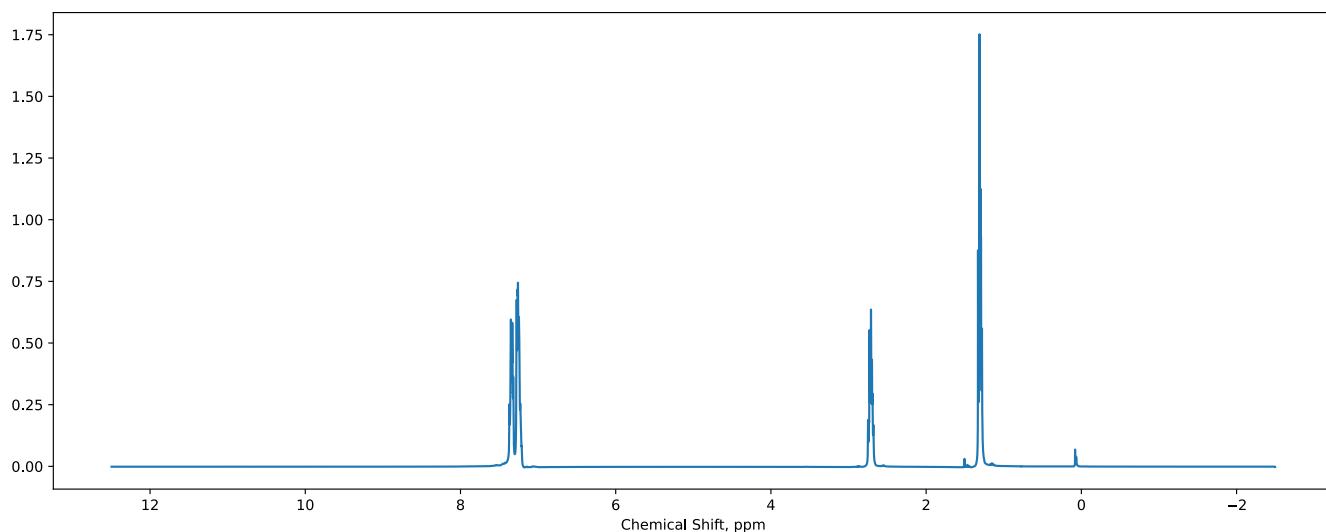
Alternatively, `nmrglue` contains an object called a *unit conversion object* that can be created and used to convert between ppm, Hz, and point index values for any position in an NMR spectrum. To create a unit conversion object, use the `make_uc()` function which takes two arguments – the dictionary, `dic`, and the original data array, `data`, generated from reading the NMR file in section 12.2.

```
unit_conv = ng.pipe.make_uc(dic, data)
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.

```
uc = ng.pipe.make_uc(dic, data)
ppm_scale = uc.ppm_scale()
```

```
phased_data_rev = phased_data.real[::-1]
fig5 = plt.figure(figsize=(16, 6))
ax5 = fig5.add_subplot(1, 1, 1)
ax5.plot(ppm_scale, phased_data_rev)
ax5.set_xlabel('Chemical Shift, ppm')
plt.gca().invert_xaxis()
```



The following example uses the ppm scale generated by the unit conversion object.

12.1.4.2 Referencing the Data

In the above spectrum, the small resonance at 0.08 ppm is internal TMS (tetramethylsilane) standard which should be located at 0.00 ppm. The temptation is to subtract 0.08 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 3).

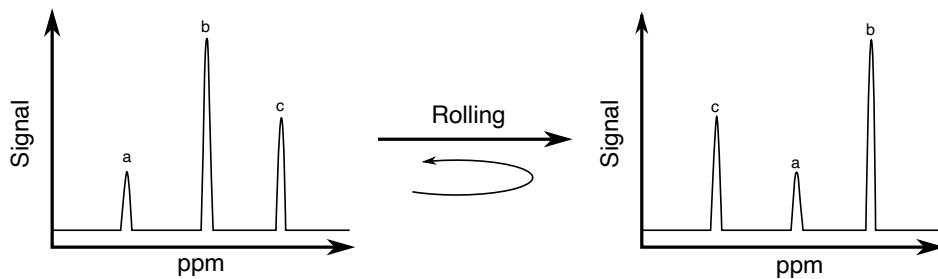


Figure 3 Referencing an NMR spectrum is performing by rolling it until the peaks reside at the correct shifts. As a signal falls off one end of the spectrum, it reappears at the other end.

Conveniently for us, NumPy has a function `np.roll()` that does exactly this to array data, and nmrglue contains its own `ng.process.proc_base.roll()` function for this task which calls the NumPy function. Feel free to use either one.

```
np.roll(array, shift)
```

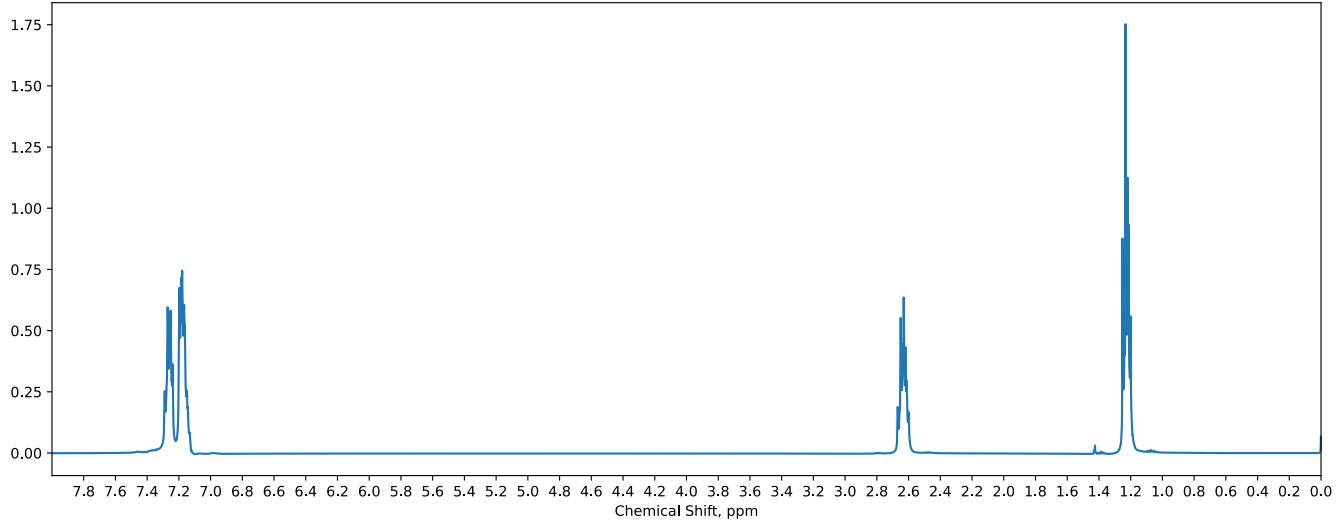
The `np.roll()` function takes two required arguments. The first is the array containing the data and the second is the amount to shift or roll the data. The shift is not in ppm but rather positions in the data array. If you know your referencing correction in ppm (Δ_{ppm}), use the following equation which describes the relationship between the correction in ppm (Δ_{ppm}) and the correction in number of data points (Δ_{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.

$$\Delta_{points} = \frac{\Delta_{ppm} \times size \times obs}{sw}$$

Alternatively, you can accomplish this same calculation using the unit conversion object by determining the data point difference between 0.00 ppm and the current position of the TMS. The example below requires the spectrum to be shifted by -0.08 ppm, and both approaches are demonstrated below.

```
ref_shift_manual = int((0.08 * size * obs) / sw) # calc shift yourself  
# OR  
ref_shift_uc = uc('0.00 ppm') - uc('0.08 ppm') # calc shift using unit conversion object  
  
data_ref = np.roll(phased_data_rev, ref_shift_uc)
```

```
fig6 = plt.figure(figsize=(16, 6))
ax6 = fig6.add_subplot(1, 1, 1)
ax6.plot(ppm_scale, data_ref.real)
ax6.set_xlabel('Chemical Shift, ppm')
plt.xlim(8, 0)
plt.xticks(np.arange(0, 8, 0.2))
plt.show()
```



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.

12.1.5 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 12.1.4.1](#)), and limits is a list or array of limits for integration.

```
ng.analysis.integration.integrate(data, conv_obj, limits)
```

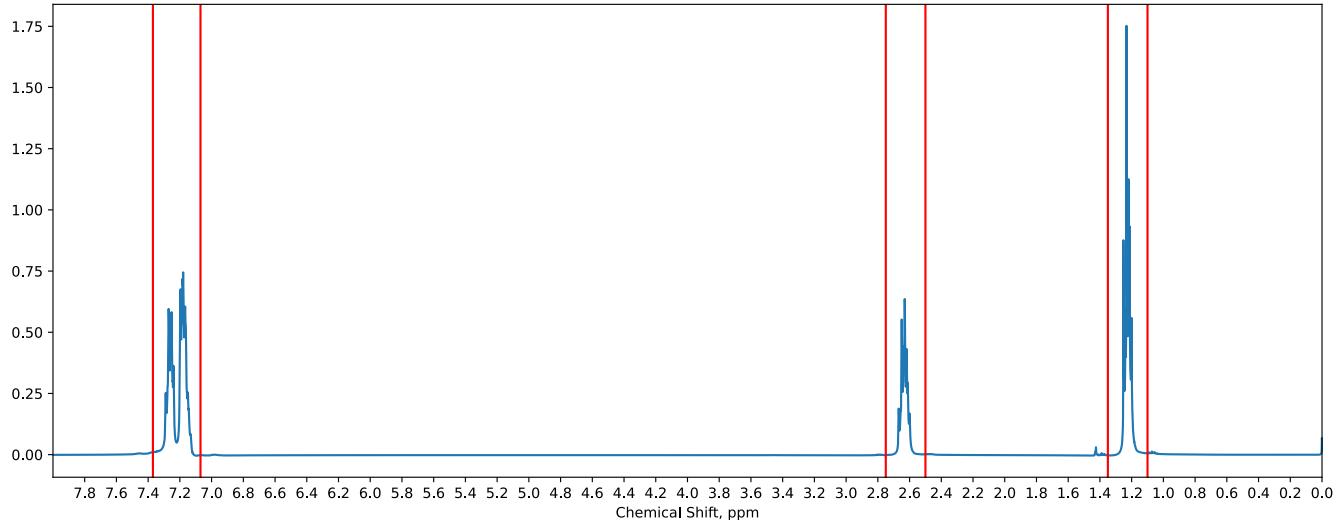
```
uc = ng.pipe.make_uc(dic, phased_data_rev)
```

```
limits = np.array([[7.07,7.37], [1.10, 1.35], [2.50,2.75]])
```

```

fig7 = plt.figure(figsize=(16, 6))
ax7 = fig7.add_subplot(1, 1, 1)
ax7.plot(ppm_scale, data_ref.real)
ax7.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 above as vertical red lines.

Now to integrate our NMR spectrum.

```

area = ng.analysis.integration.integrate(data_ref.real, uc, limits)
area

```

```
array([0.05381569, 0.03379756, 0.02178194])
```

These values are probably not what you expected, but if we divide all of them by the smallest value, it is easier to see the relative ratio of areas.

```

ratio = area / np.min(area)
ratio

```

```
array([2.47065636, 1.55163197, 1.          ])
```

The spectrum above is the ^1H NMR of ethylbenzene in 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 (CHCl_3 , 7.27 ppm) being included in the integration of the aromatic protons among other things.

```
10 / np.sum(ratio) * ratio
```

```
array([4.91938374, 3.08949203, 1.99112423])
```

12.1.6 Peak Picking

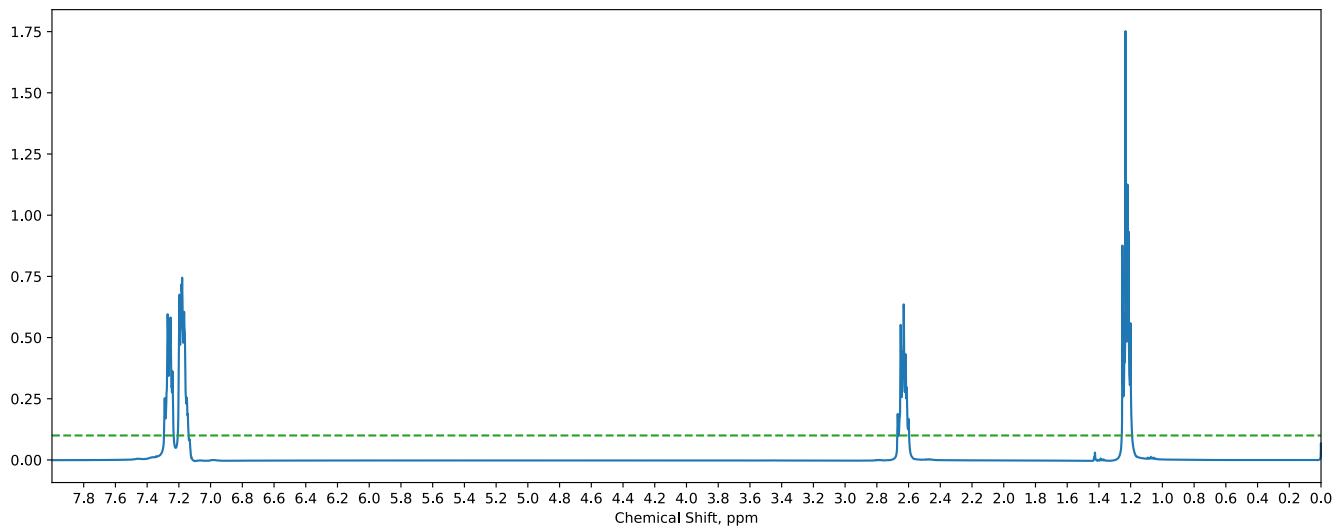
Another piece of information that is commonly extracted from NMR spectra is the chemical shift of the resonances.

Similar to integration, SciPy contains functions such as [`scipy.signal.argrelextrema\(\)`](#) or [`scipy.signal.find_peaks\(\)`](#) 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.

```
fig8 = plt.figure(figsize=(16, 6))
ax8 = fig8.add_subplot(1, 1, 1)
ax8.plot(ppm_scale, data_ref.real)
ax8.set_xlabel('Chemical Shift, ppm')
plt.xlim(8, 0)
plt.xticks(np.arange(0, 8, 0.2))
plt.axhline(0.1, c='C2', ls='--');
```



Note

The `ng.analysis.peakpick.pick()` does not work with NumPy versions 1.24 and later if you are using a version of nmrglue before 0.10. Consider upgrading your version of nmrglue if `ng.analysis.peakpick.pick()` raises an error.

```
peaks = ng.analysis.peakpick.pick(data_ref.real, pthres=0.1)
peaks
```

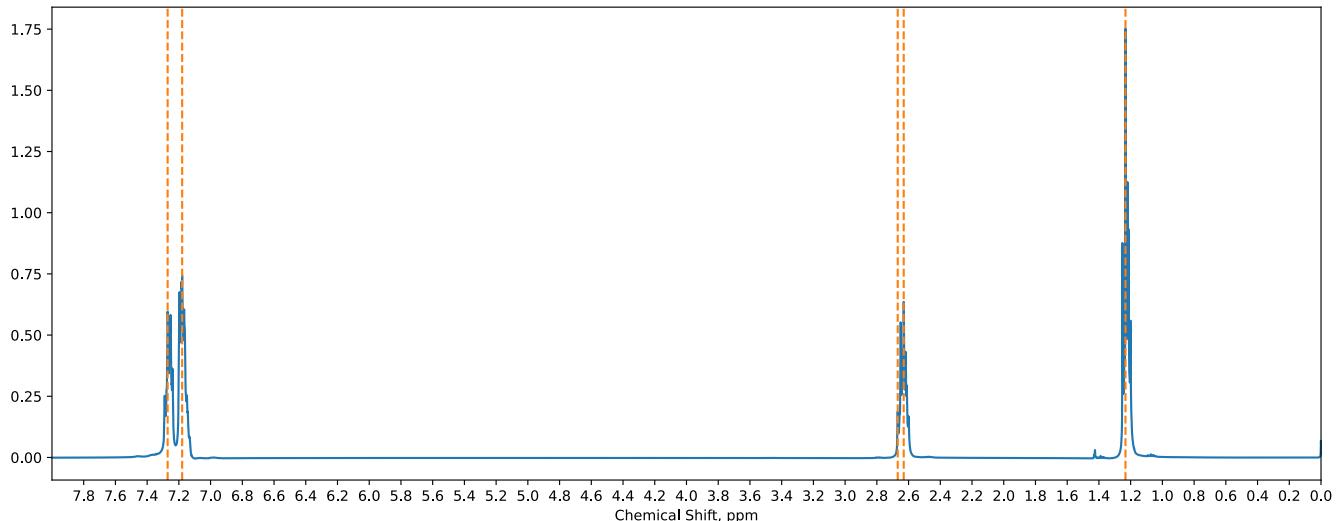
```
rec.array([(4568., 1, 32.13860124, 17.82116699),
(4648., 2, 35.96968658, 25.60328484),
(8591., 3, 5., 0.88379395),
(8624., 4, 31.36229043, 16.87084198),
(9846., 5, 34.1857674, 27.6301403)],
dtype=[('X_AXIS', '<f8'), ('cID', '<i8'), ('X_LW', '<f8'), ('VOL', '<f8')])
```

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 an estimate of the areas of each peak. We can use the index values to index the `ppm` array for the chemical shifts.

```
peak_loc = []
for x in peaks:
    peak_loc.append(ppm_scale[int(x[0])])
print(peak_loc)
```

```
[np.float64(7.270899450723067), np.float64(7.17937704723959), np.float64(2.6684665855476872), np.float64(1.1444444444444444)]
```

```
fig9 = plt.figure(figsize=(16, 6))
ax9 = fig9.add_subplot(1, 1, 1)
ax9.plot(ppm_scale, data_ref.real)
ax9.set_xlabel('Chemical Shift, ppm')
plt.xlim(8, 0)
plt.xticks(np.arange(0, 8, 0.2))
for p in peak_loc:
    plt.axvline(p, c='C1', ls='--', alpha=1)
```



We can plot the NMR spectrum with these chemical shifts marked with vertical dotted lines shown above. 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 [nmrglue documentation](#) that can be adjusted.

12.2 Simulating NMR with nmrsim

nmrsim is a Python package for simulating NMR spectra based on information such as the chemical shifts, coupling constants, and number of coupling nuclei. The package is capable of simulating individual first-order and second-order

splitting patterns or entire NMR spectra. It can also simulate dynamic NMR caused by nuclei rapidly exchanging. nmrssim is [installable using pip](#). The package has a few key functions listed below (Table 3) for simulating first-order multiplets, spin systems, and spectra. The `Multiplet()` function is used to simulate a single, first-order resonance such as a 1:2:1 triplet or a doublet-of-doublets while the `SpinSystem()` function simulates two resonance signals belonging to pairs of coupled nuclei. The `Spectrum()` function can generate entire spectra by merging the resonances generated by other functions.

Table 3 Select nmrssim Simulation Functions

Function	Description
<code>Multiplet()</code>	Simulates a single, first-order multiple
<code>SpinSystem()</code>	Simulates sets of first- or second-order multiplets generated by coupled nuclei
<code>Spectrum()</code>	Simulates first-order spectra

12.2.1 Simulating First-Order Multiplets

As an example, we can simulate the signal of methylene (i.e., -CH₂-) protons in CH₃-CH₂-CH-. Let us assume that the methyl/methylene protons have coupling constants of J = 7.8 Hz the methine/methylene protons have a coupling constant of J = 6.1 Hz. First, we need to import the `Multiplet()` function along with the `mplplot()` plotting function. The `Multiplet()` function takes the resonance frequency in Hz (`v`) as the first positional argument followed by the intensity (`I`) of the resonance signal. This can simply be the number of nuclei the signal represents and is only really important when generating entire spectra with multiple signals so that signals that represent more nuclei have a larger area. Finally, coupling constants (`J`)/number of nuclei (`n_nuc`) pairs is provided as a list of tuples, list of lists, or 2D array.

```
Multiplet(v, I, [(J, n_nuc), (J, n_nuc)])
```

The `Multiplet()` function generates a `Multiplet` object which can produce a peak list using the `peaklist()` method. The peak list is simply a list of tuples with (`v`, `I`) pairs for each peak in the multiplet.

```
from nmrssim import Multiplet
from nmrssim.plt import mplplot
```

```
mult = Multiplet(500, 2, [(7.8, 3),(6.1, 1)])
mult
```

```
<nmrssim._classes.Multiplet at 0x10e199af0>
```

```
mult_peaks = mult.peaklist()
mult_peaks
```

```
[(485.2500000000006, 0.125),  
 (491.350000000001, 0.125),  
 (493.05, 0.375),  
 (499.150000000003, 0.375),  
 (500.849999999997, 0.375),  
 (506.95, 0.375),  
 (508.649999999999, 0.125),  
 (514.749999999999, 0.125)]
```

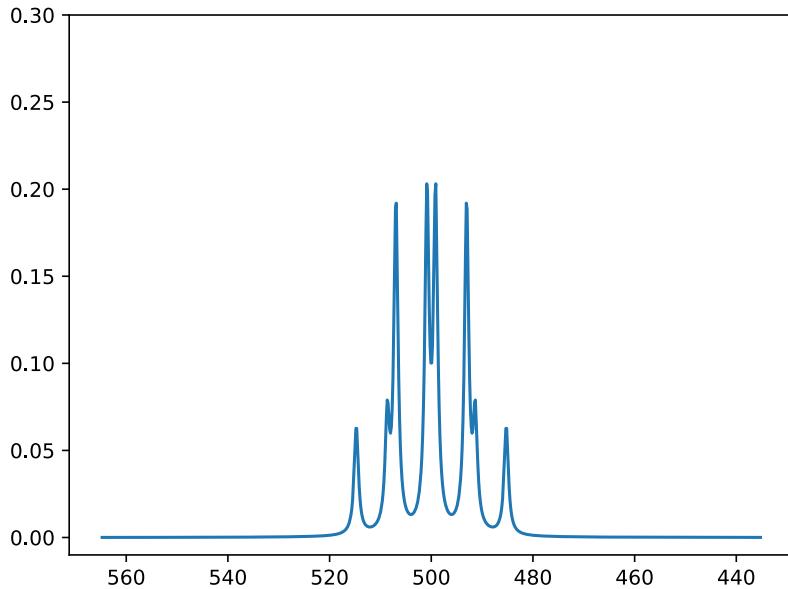
Next we need to visualize this data. For this, nmrsim provides multiple plotting functions built off of matplotlib. We will focus on the `mplplot()` function which accepts the peaklist and generates the line shapes for the actual peaks.

```
x, y = mplplot(peaklist, w=1, y_min=-0.01, y_max=1, limits=(min, max), points=800)
```

There are a number of optional, keyword arguments such as line width (`w`), *y*-axis limits (`y_min` and `y_max`), *x*-axis limits (`limits`), and number of points in the multiplet (`points`). The `mplplot()` function will return the *x*- and *y*-coordinates for the plot. To suppress this, either end the line with a `;` or give it a pair of variables to store these data.

```
freq, intens = mplplot(mult_peaks, y_max=0.3)
```

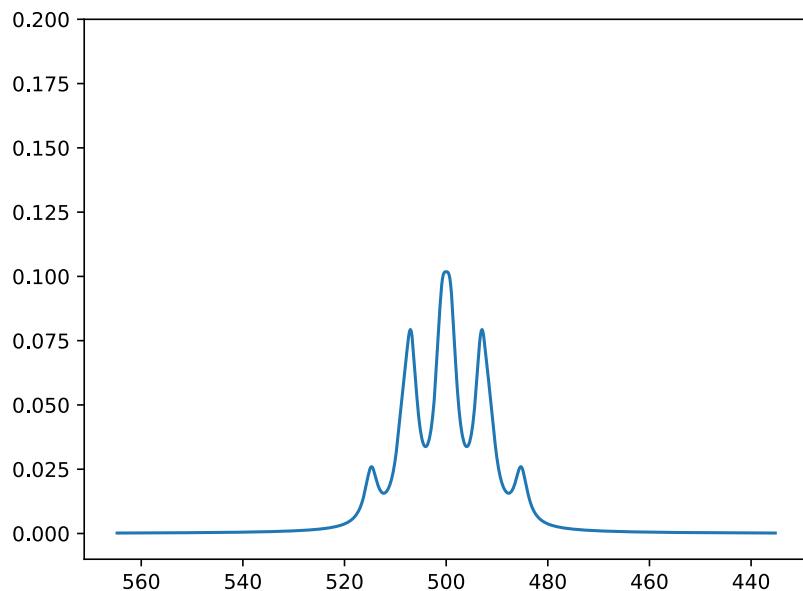
```
[<matplotlib.lines.Line2D object at 0x10dc2d100>]
```



Below is the same splitting pattern with the line width tripled.

```
mplplot(mult_peaks, y_max=0.2, w=3);
```

```
[<matplotlib.lines.Line2D object at 0x10f2fccb0>]
```

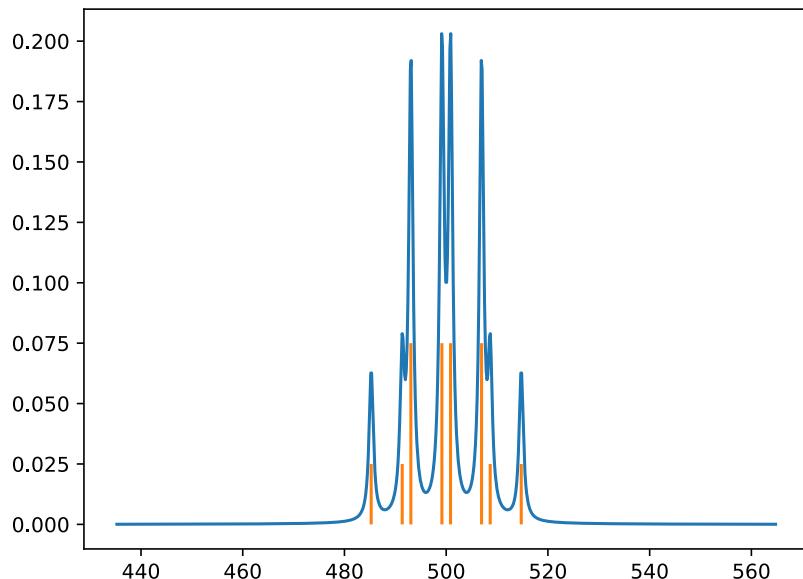


As another option, we can overlay the multiplet with lines showing the exact chemical shift and intensity ratio of each peak. This can be done either using your plotting library of choice or using the `mplplot_stick()` function in nmrsim. Below the intensity of the stem plot is reduced by a fifth to keep the lines inside the blue splitting pattern.

```
peaks = np.array(mult_peaks)

plt.plot(freq, intens)
plt.stem(peaks[:,0], peaks[:,1]/5, linefmt='C1', basefmt=' ', markerfmt=' ')
```

<StemContainer object of 3 artists>



12.2.2 Simulating Spectra

Entire NMR spectra can be simulated from the component resonance signals - either Multiplet or SpinSystem objects. Down below, we simulate the signals for the methyl, ethyl, and -OH from ethanol with a $J=7.3$ Hz. Because the -OH peak is broader due to exchange, the width of the resonance is increased by setting `w=3`. The three resonances are then

combined into a single spectrum using the `Spectrum()` function which accepts the resonances in a list and also optionally accepts minimum (`vmin=`) and maximum (`vmax`) frequency ranges for the spectrum in Hz.

💡 Tip

A spectrum can also be created by adding the resonance signals together with the `+` operator like below.

```
spec = methyl + ethyl + OH
```

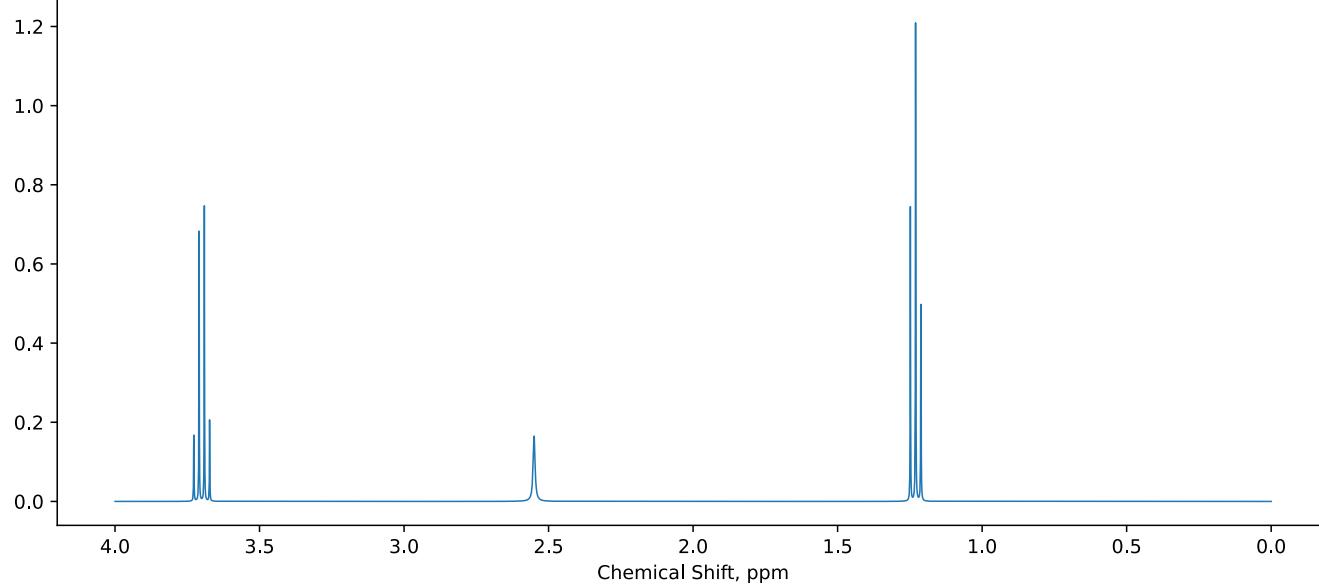
```
from nmrsim import Spectrum

# create resonances
methyl = Multiplet(492, 3, [(7.3, 2)])
ethyl = Multiplet(1480, 2, [(7.3, 3)])
OH = Multiplet(1020, 1, [], w=3)

# build spectrum
spec = Spectrum([methyl, ethyl, OH], vmin=0, vmax=1600)
v_spec, I_spec = spec.lineshape(points=4000)

# convert from Hz to ppm scale on a 400 MHz spectrometer
v_spec_ppm = v_spec / 400

plt.figure(figsize=(12, 5))
plt.plot(v_spec_ppm, I_spec, linewidth=0.8)
plt.xlabel('Chemical Shift, ppm')
plt.gca().invert_xaxis()
```



The simulation even exhibits the second-order roofing effect where coupled resonances 'lean' towards each other.

12.2.3 Simulate Second-Order Resonances

nmrsim is capable of simulating second-order splitting patterns using the following functions (Table 4). The name of each function is based on the Pople nomenclature where letters adjacent to each other in the alphabet represent resonances that are near each other in a spectrum (e.g., A and B), letters far apart in the alphabet represent resonances further part

in the spectrum (e.g., A and X), the same letter is used to represent chemically equivalent nuclei, and primes are used to differentiate chemically equivalent nuclei that are magnetically inequivalent (e.g., A and A').

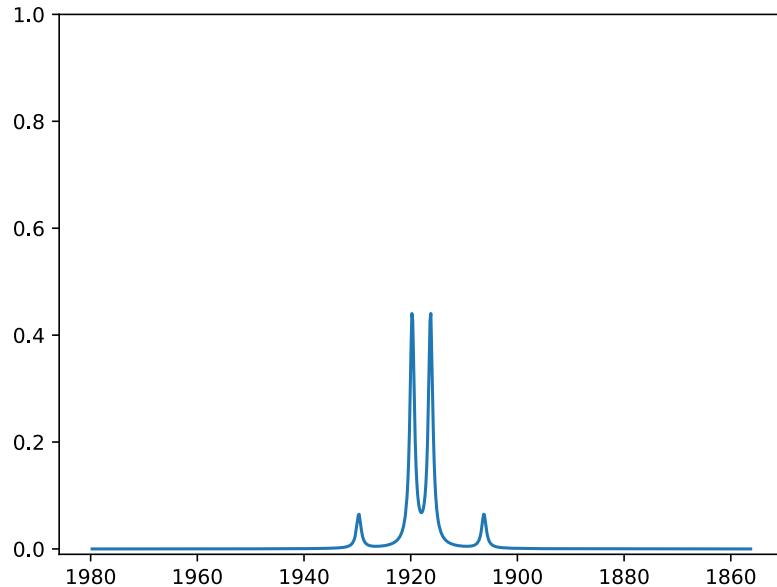
Table 4 Second-Order Simulation Functions

Function	Description
<code>AB()</code>	Simulates an AB system
<code>AB2()</code>	Simulates an AB ₂ system
<code>ABX()</code>	Simulates an ABX system
<code>ABX3()</code>	Simulates an ABX ₃ system
<code>AAXX()</code>	Simulates an AA'XX' system
<code>AAAB()</code>	Simulates an AA'BB' system

These functions typically accept the coupling constants (e.g., `Jab=`), the distance between the two nuclei (e.g., `Vab=`), and the chemical shift of the signal in Hz (`Vcentr=`). As a demonstration, below we will simulate an AB spin system where the two nuclei are coupled with $J=10.0$ Hz and separation between the two signals of 9.0 Hz.

```
from nmrsim.discrete import AB
res = AB(10, 9, 1918)
mplplot(res);
```

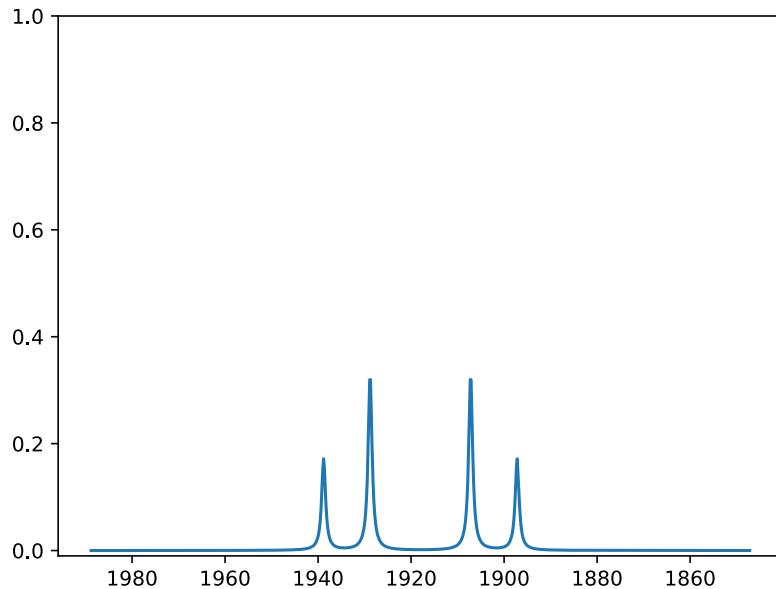
[<matplotlib.lines.Line2D object at 0x10f6909e0>]



If we increase the distance between the two nuclei to 30.0 Hz, not only do the two signals become further apart, but the second-order character, unevenness in this case, decreases. It is important to note that when measuring the distance between the two second-order signals like this, the center of a doublet with uneven heights is not the center of the doublet but rather a weighted frequency average of the two peaks based on intensities. This means the chemical shift of a doublet is closer to the larger of the two peaks in the doublet.

```
res = AB(10, 30, 1918)
mplplot(res);
```

```
[<matplotlib.lines.Line2D object at 0x10f66caa0>]
```



12.2.4 Dynamic NMR Simulations

Nuclei in some molecules can exchange with each other at observable rates. At lower temperatures, the exchange is relatively slow leading to two distinct and reasonably sharp signals representing the two environments of the exchanging nuclei. As the temperature is increased, the exchange becomes more rapid causing the two signals to broaden and become closer until they merge into a single peak and ultimately sharpen. There are two dynamic NMR functions in the `nmrsim.dnmr` module: the `dnmr_two_singlets()` function which simulates two exchanging nuclei (or groups of chemically equivalent nuclei) that are not coupling with each other while the `dnmr_AB()` function simulates two exchanging nuclei that couple with each other. Below we will simulate two non-coupled, singlet signals exchanging with either other. The required arguments are the chemical shift frequencies of the two nuclei during slow exchange (`va` and `vb`), the exchange rate constant in Hz (`k`), the half-height width of the peaks at slow exchange (`wa` and `wb`), and the fraction of the nuclei in position *a* (`pa`). Optionally, you can specify the frequency limits for the generated line shape (`limits=`) and number of data points (`points=`).

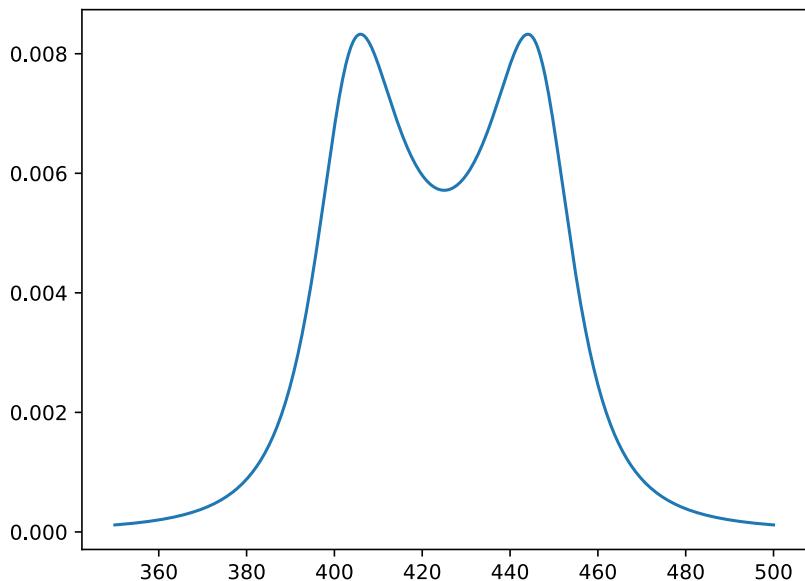
```
v, I = dnmr_two_singlets(va, vb, k, wa, wb, pa, limits=(min, max), point=800)
```

Below is a simulation with a rate constant of 70 Hz.

```
from nmrsim.dnmr import dnmr_two_singlets
```

```
v, I = dnmr_two_singlets(400, 450, 70, 2, 2, 0.5)
plt.plot(v, I)
```

```
[<matplotlib.lines.Line2D object at 0x10f6fc050>]
```



Further Reading

1. NMRglue Website. <https://www.nmrglue.com/> (free resource)
2. NMRglue Documentation Page. <http://nmrglue.readthedocs.io/en/latest/tutorial.html> (free resource)
3. J.J. Helmus, C.P. Jaroniec, NmrGlue: An open source Python package for the analysis of multidimensional NMR data, *J. Biomol. NMR* 2013, 55, 355-367, <http://dx.doi.org/10.1007/s10858-013-9718-x>. (paper on nmrglue)
4. nmrSim Documentation Page. <https://nmrsim.readthedocs.io/en/latest/introduction.html> (free resource)
5. American Chemical Society Division of Organic Chemistry, Hans Reich's NMR Spectroscopy Collection. <https://organicchemistrydata.org/hansreich/resources/nmr/?page=nmr-content%2F> (free resource)

Exercises

Complete the following exercises in a Jupyter notebook and NMRglue library. Any data file(s) referred to in the problems can be found in the [data](#) folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

1. Open the ^1H NMR spectrum of ethanol, **EtOH_1H_NMR.fid**, taken in CDCl_3 with TMS using NMRglue. Use the [pipe](#) module.
 - a) Plot the resulting spectrum and be sure to properly reference it if not done already.
 - b) Integrate the methyl ($-\text{CH}_3$) versus the methylene ($-\text{CH}_2-$) resonances and calculate the ratio.
2. Open the ^1H and ^{13}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 CDCl_3 with TMS and plot them on a ppm scale. Be sure to properly phase and reference the spectra if not done already. Use the [pipe](#) module.
3. Simulate a first-order doublet of triplets with $J=5.6$ Hz and $J=9.2$ Hz, respectively.
4. Select an article from the [Journal of Organic Chemistry](#) or some other journal and simulate an NMR spectrum with coupling (e.g., not $^{13}\text{C}[^1\text{H}]$) based on data listed in the experimental section. Note: some articles are free to access even if you do not have a subscription. Just accesss the most recent issue, and the free articles are marked "Open

Access" in ACS journals.

5. Simulate a second-order AA'BB' simulation with a $J_{AA'} = 15.0$ Hz, $J_{BB'} = 15.0$ Hz, $J_{AB} = 7.0$ Hz, $J_{AB'} = 7.0$ Hz and a separation of 27.0 Hz. Compare your simulate to what is shown on [Hans Reich's figure](#) (first set of NMR spectra on the page).

Chapter 13: Machine Learning using Scikit-Learn

Contents

- 13.1 Supervised Learning
- 13.2 Unsupervised Learning
- 13.3 Final Notes
- Further Reader
- Exercises

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. 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 13.1](#), 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, 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 alcohols we have never seen. This probably appears like a daunting task. 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.

There are multiple machine learning libraries for Python, but one of the most common, general-purpose machine learning libraries is scikit-learn. 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 effectiveness 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.

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

13.1 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 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.

13.1.1 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. 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.

```
ROH = pd.read_csv('data/ROH_data.csv', sep=',')
ROH.head()
```

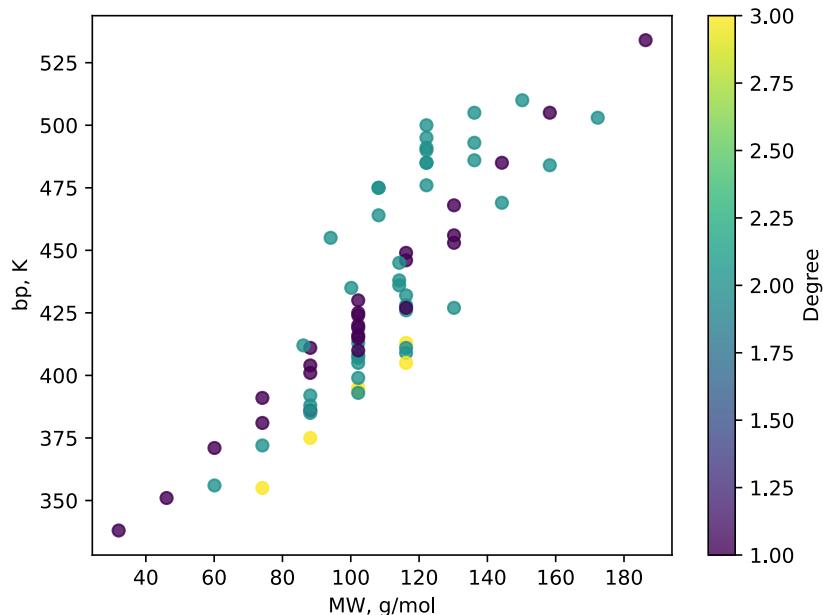
	bp	MW	carbons	degree	aliphatic	avg_aryl_position	cyclic
0	338	32.04	1	1	1	0.0	0
1	351	46.07	2	1	1	0.0	0
2	371	60.10	3	1	1	0.0	0
3	356	60.10	3	2	1	0.0	0
4	391	74.12	4	1	1	0.0	0

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 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 alcohols 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.

```
plt.scatter(ROH['MW'], ROH['bp'], alpha=0.8, c=ROH['degree'], cmap='viridis')
plt.xlabel('MW, g/mol')
plt.ylabel('bp, K')
cbar = plt.colorbar()
cbar.set_label('Degree')
```



13.1.2 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 dataset 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 dataset 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`).

```
from sklearn.model_selection import train_test_split
```

```
target = ROH['bp']
features = ROH[['MW', 'carbons', 'degree', 'aliphatic',
                'avg_aryl_position', 'cyclic']]
```

```
X_train, X_test, y_train, y_test = train_test_split(features, target,  
                                                test_size=0.25, random_state=18)
```

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.

13.1.3 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.

$$\text{target} = w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + \dots$$

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.

```
from sklearn import linear_model
```

```
reg = linear_model.LinearRegression()
```

```
reg.fit(X_train, y_train)
```

▼ `LinearRegression` ⓘ ?

► Parameters

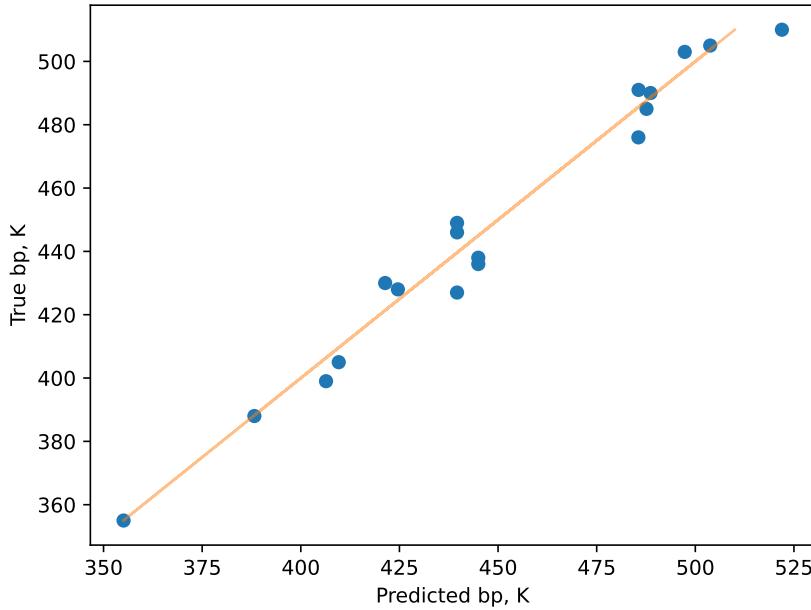
Finally, the trained model can make predictions using the `predict()` method.

```
prediction = reg.predict(X_test)  
prediction
```

```
array([521.94389573, 439.60028899, 421.38488633, 485.6143471 ,  
      355.07207513, 444.98911542, 439.60028899, 487.61879909,  
      488.64633926, 497.31838329, 388.22848073, 406.39325504,  
      424.6086577 , 444.98911542, 485.56371876, 439.60028899,  
      503.77912142, 409.61702641])
```

Remember that the algorithm has been only provided 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.

```
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.

13.1.4 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 1, the better the predictions are.

```
reg.score(X_test, y_test)
```

```
0.9738116533899365
```

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 (k being some number) 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 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.

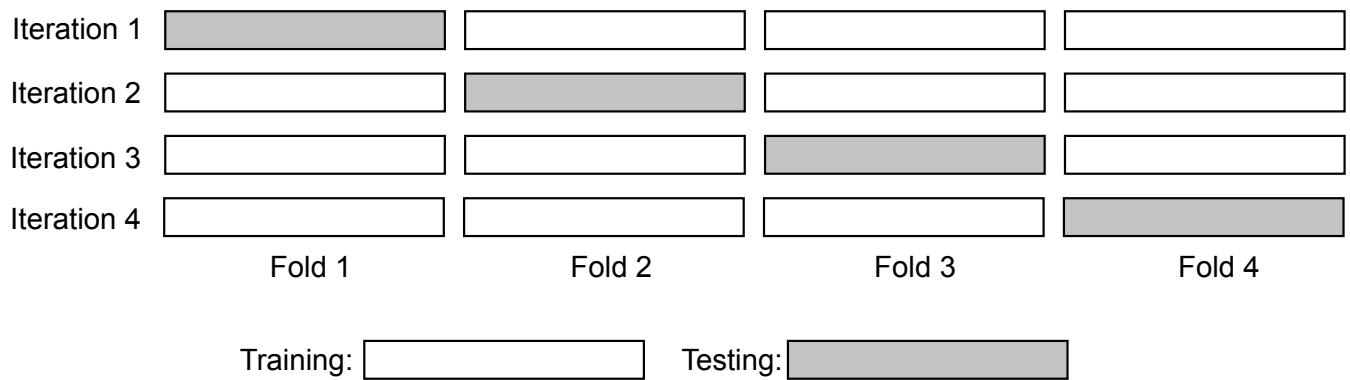


Figure 1 In each iteration of k -fold cross-validation, different folds of data are used for training and testing the algorithm.

A demonstration of k -fold cross validation is shown 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.

```

from sklearn.model_selection import cross_val_score, ShuffleSplit

splitter = ShuffleSplit(n_splits=5)

reg = linear_model.LinearRegression()

scores = cross_val_score(reg, features, target, cv=splitter)
scores

array([0.96589018, 0.97370412, 0.95866531, 0.97170325, 0.94186934])

```

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.

```

scores.mean()

np.float64(0.9623664411336886)

```

13.1.5 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.

```

reg = linear_model.LinearRegression()
reg.fit(X_train, y_train)
reg.coef_

```

```
array([ -5.06283477,  89.19634615, -14.99163129,   5.73273187,
       -2.05508033,  15.9368917 ])
```

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 ($32.04 \rightarrow 186.33 \text{ g/mol}$).

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 1, but others are detailed on the scikit-learn website.

Table 1 Preprocessing Data Scaling Functions

Scaler	Description
<code>MinMaxScaler</code>	Scales the features to a designated range; defaults to [0, 1]
<code>StandardScaler</code>	Centers the features around zero and scales them to a variance of one
<code>RobustScaler</code>	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 $0 \rightarrow 1$. 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.

```
from sklearn.preprocessing import MinMaxScaler
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.

```
X_train, X_test, y_train, y_test = train_test_split(scaled_features, target)
```

```
reg = linear_model.LinearRegression()
reg.fit(X_train, y_train)
```

▼ `LinearRegression` ⓘ ?

► `Parameters`

```
reg.coef_
```

```
array([-926.1978598, 1126.11028034, -29.41839113, 10.20167674,
       -18.28297316, 13.41985871])
```

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.

13.1.6 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 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 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 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.

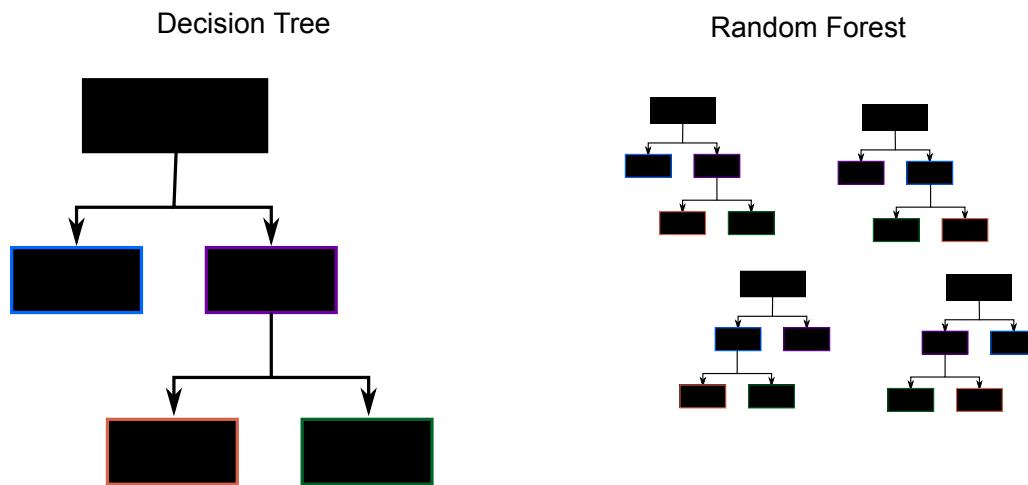


Figure 2 An illustration of a single decision tree (left) and a random forest (right) composed of numerous decision trees generated with different subsections of data.

13.1.7 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.

```
data = pd.read_csv('data/org_comp.csv')
data.head
```

```
<bound method NDFrame.head of
 0      0  455   94.11  6      1      1      MW  C  cyclic  aromatic  unsaturation
 1      0  475   108.14  7      1      1
 2      0  475   108.14  7      1      1
 3      0  464   108.14  7      1      1
 4      0  474   122.17  8      1      1
 ..
 117     2  498   135.21  9      1      1      ...
 118     2  407    99.17  6      1      0      ...
 119     2  381    85.15  5      1      0      ...
 120     2  327   113.20  7      1      0      ...
 121     2  463   127.23  8      1      0      ...
[122 rows x 7 columns]>
```

```
target = 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.

```
X_train, X_test, y_train, y_test = train_test_split(
    features, target, test_size=0.25, random_state=18)
```

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
rf.predict(X_test)
```

```
array([1, 0, 0, 0, 0, 2, 1, 0, 0, 0, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 2, 0,
       1, 2, 0, 1, 0, 2, 2, 0, 2])
```

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.

```
rf.score(X_test, y_test)
```

```
0.7419354838709677
```

13.1.8 Confusion Matrix

The above score shows that the predictions are about 74% 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.

```

from sklearn.metrics import confusion_matrix

conf_matrix = confusion_matrix(y_test, rf.predict(X_test))
conf_matrix

array([[11,  0,  1],
       [ 1,  4,  0],
       [ 6,  0,  8]])

```

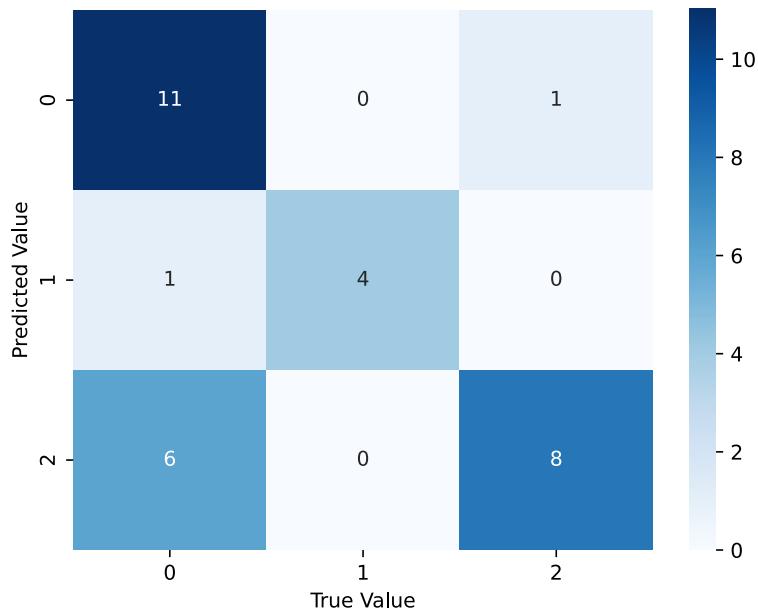
Each row is a predicted category and each column is the true category, but it is difficult to interpret the confusion matrix without labels. We can use seaborn's `heatmap()` function (see [section 10.6](#)) to produce a clearer representation.

```

import seaborn as sns

sns.heatmap(conf_matrix, annot=True, cmap='Blues')
plt.xlabel('True Value')
plt.ylabel('Predicted Value');

```



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 six 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.

13.2 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.

13.2.1 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 of high-dimensional 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...

13.2.2 Load Wine Dataset

To load the wine dataset, we first need to import the `load_wine()` function and then call the function.

```
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`.

wine.data

```
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]], shape=(178, 13))
```

wine, target

Notice again that every data point, including the category, is a number 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.

```
wine.keys()
```

```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names'])
```

We will store the features and target values in variables for use in the next section.

```
features = wine.data
target = wine.target
```

13.2.3 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 paired down to two or three.

```
wine.feature_names
```

```
['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.

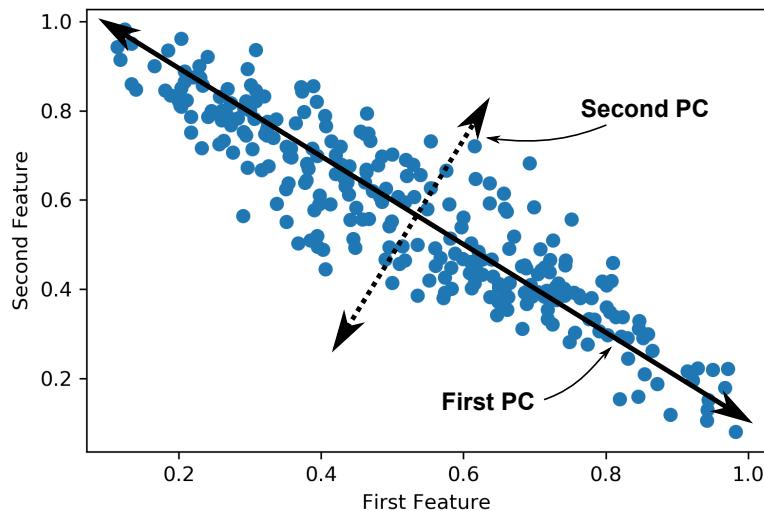


Figure 2 Principle components are axes of greatest variation of a dataset in feature space. The first principle component (solid line) is the axis of greatest variation while the second principle component (dotted line) is the axis of second greatest variation orthogonal to the first.

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 13.1.5](#).

```
from sklearn.preprocessing import StandardScaler
```

```
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.

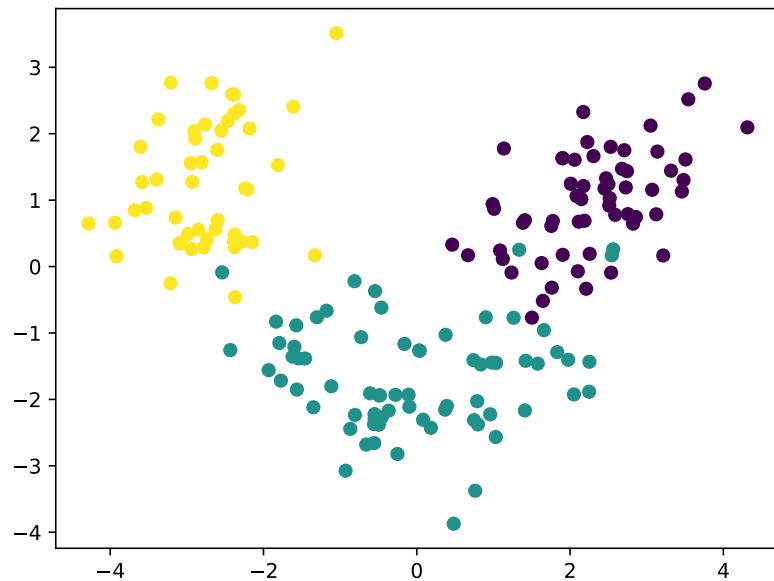
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
trans_data = pca.fit_transform(features_ss)
trans_data.shape
```

```
(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.

```
plt.scatter(trans_data[:,0], trans_data[:,1], c=target);
```



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.

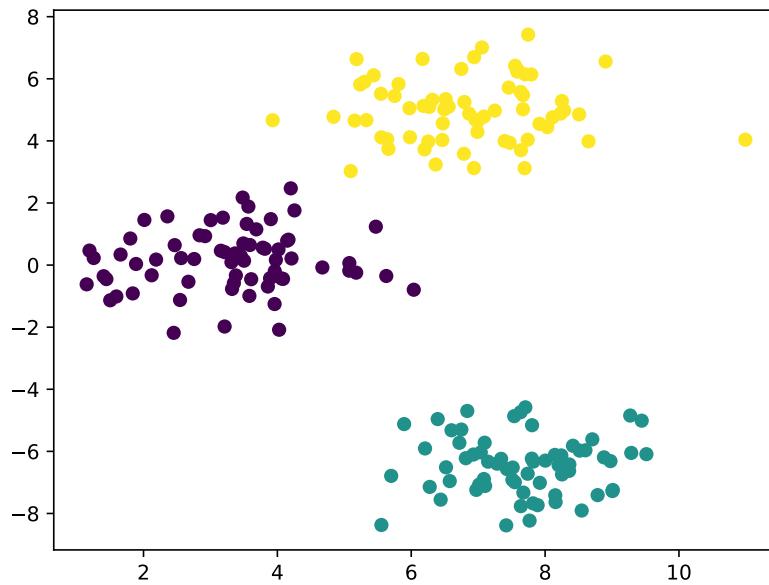
13.2.4 Clustering

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`).

```
from sklearn.datasets import make_blobs
```

```
X, y = make_blobs(n_samples=200, n_features=2, centers=3, cluster_std=1, random_state=18)
plt.scatter(X[:,0], X[:,1], c=y);
```



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. For this task, we will use the `StandardScaler()` function introduced in [section 13.2.5](#).

```
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.

```
from sklearn.cluster import DBSCAN
DB = DBSCAN(eps=0.4, min_samples=5)
DB.fit(X_ss)
```

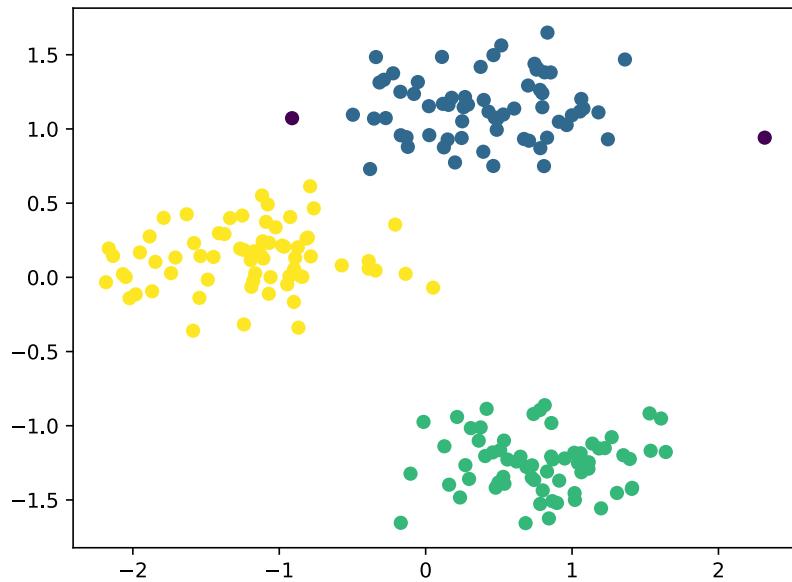
▼ DBSCAN ⓘ ?
► Parameters

```
DB.labels_
```

```
array([ 0,  0,  1,  1,  2,  0,  2,  1,  0,  0,  0,  0,  2,  2,  2,  2,  2,  1,
       0,  2,  0,  2,  0,  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,  0,  2,  0,  2,  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,  2,  2,  2,  1,  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.

```
plt.scatter(X_ss[:,0], X_ss[:,1], c=DB.labels_);
```

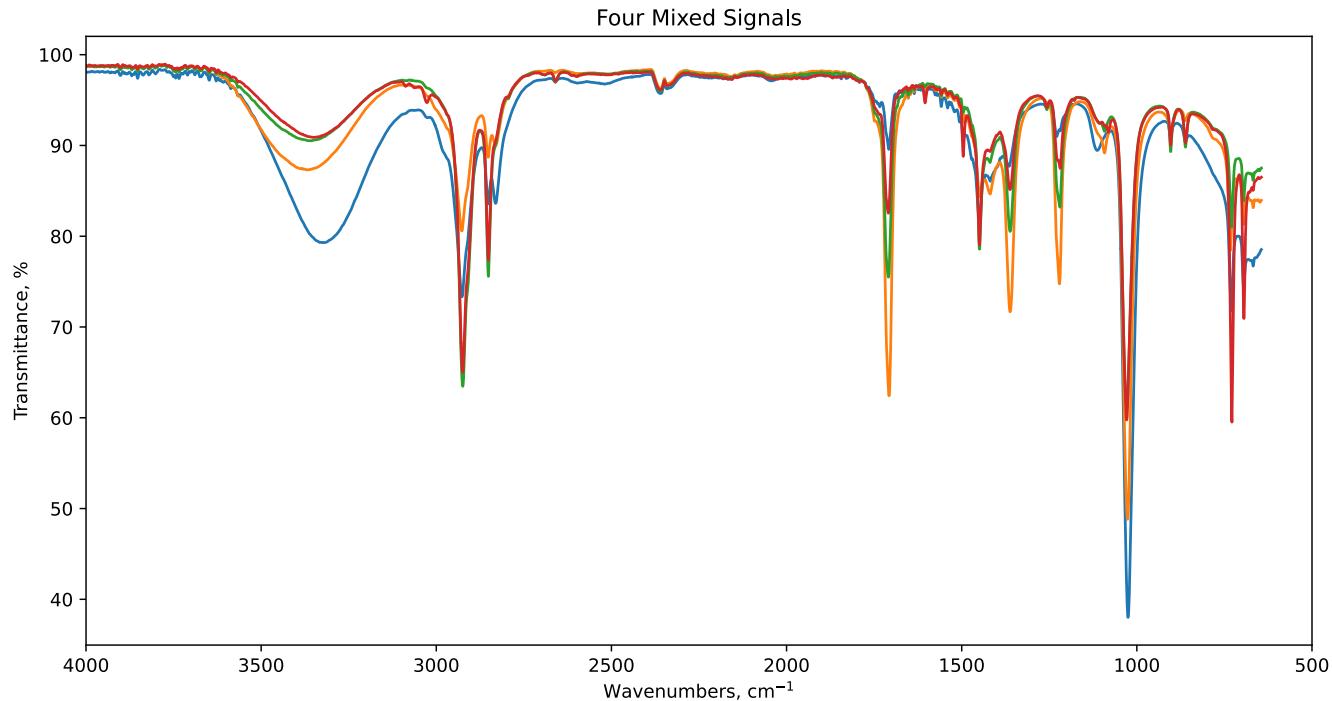


13.2.5 Blind Signal Separation

Blind signal (or source) separation (BSS) 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. 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 called `S_mix` where each column contains the data from a spectrum. For comparison purposes, we will also load IR spectra of each pure component into an array called `S_pure`. Normally we would not have spectra of pure components, hence the "blind" in blind signal separation, but this is just an example.

The code below also grabs a copy of the wavenumbers (`wn`) for plotting purposes later on. The last 300 data points of the spectra in this example are also being clipped off because they are a low signal high noise region of the spectra which reduces the effectiveness of the separation.

```
import os
data_pure = []
data_mix = []

clip = 300 # clip off noisy far end of spectrum

path = os.path.join(os.getcwd(), 'data')
os.chdir(path)

for file in os.listdir():
    if file.lower().endswith('pure.csv'):
        data_pure.append(np.genfromtxt(file, delimiter=',')[clip:,1])
        wn = np.genfromtxt(file, delimiter=',')[clip:,0]

    elif file.lower().endswith('csv') and file.lower().startswith('mix'):
        data_mix.append(np.genfromtxt(file, delimiter=',')[clip:,1])

data_array_pure = np.vstack(data_pure).T
data_array_mix = np.vstack(data_mix).T

S_pure = np.ndarray.astype(data_array_pure, float) #recast strings as floats
S_mix = np.ndarray.astype(data_array_mix, float) #recast strings as floats
```

```
os.chdir(os.path.dirname(os.getcwd()))
```

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.

```
from sklearn.decomposition import FastICA
ica = FastICA(n_components=4, random_state=42)
S_fit = ica.fit_transform(S_mix)
```

```
S_fit.shape
```

```
(6961, 4)
```

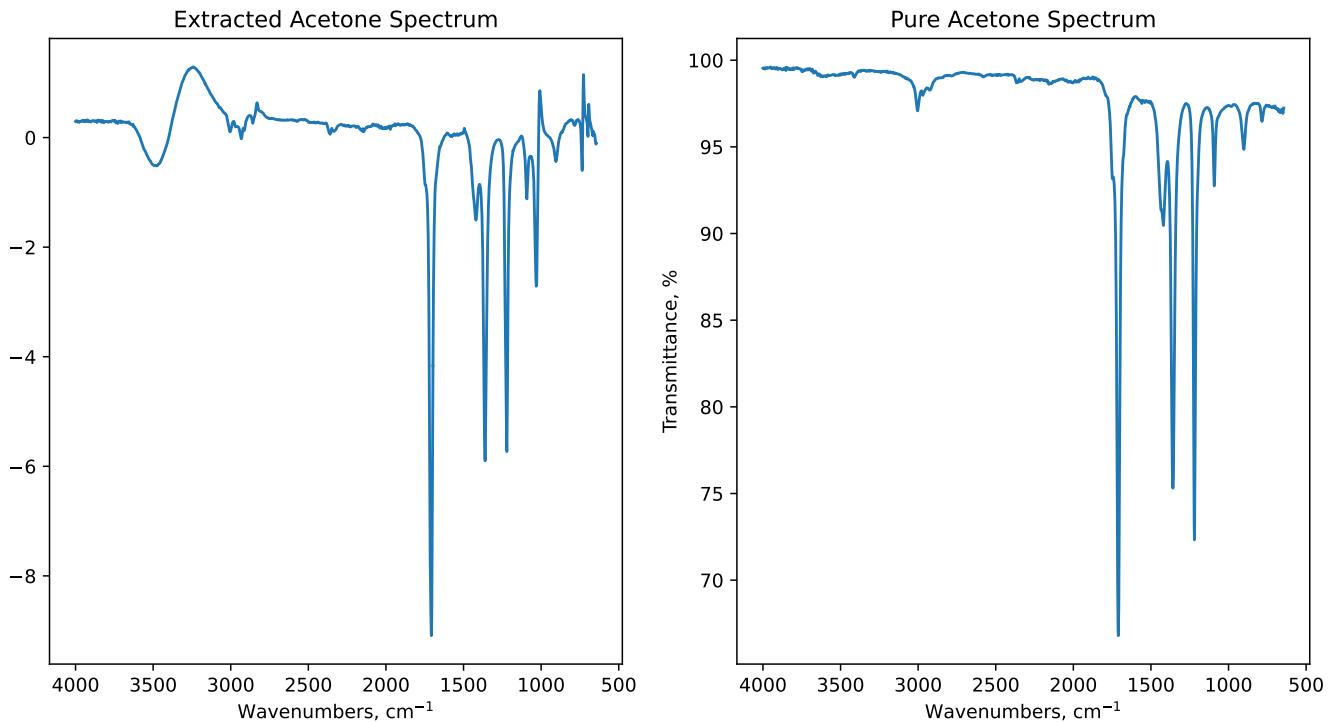
You may have noticed that instead of doing the `fit()` and `transform()` in two steps, we used a `fit_transform()` method. This method is present in many unsupervised algorithms allowing the user to perform both steps in a single function call. The resulting array `S_fit` contains the four extracted 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.

```

fig1 = plt.figure(figsize=(12,6))
ax1 = fig1.add_subplot(1,2,1)
ax1.plot(wn, S_fit[:,2])
plt.xlabel('Wavenumbers, cm$^{-1}$')
plt.title('Extracted Acetone Spectrum')
plt.gca().invert_xaxis()

ax2 = fig1.add_subplot(1,2,2)
ax2.plot(wn, S_pure[:,2])
plt.xlabel('Wavenumbers, cm$^{-1}$')
plt.ylabel('Transmittance, %')
plt.title('Pure Acetone Spectrum')
plt.gca().invert_xaxis()

```

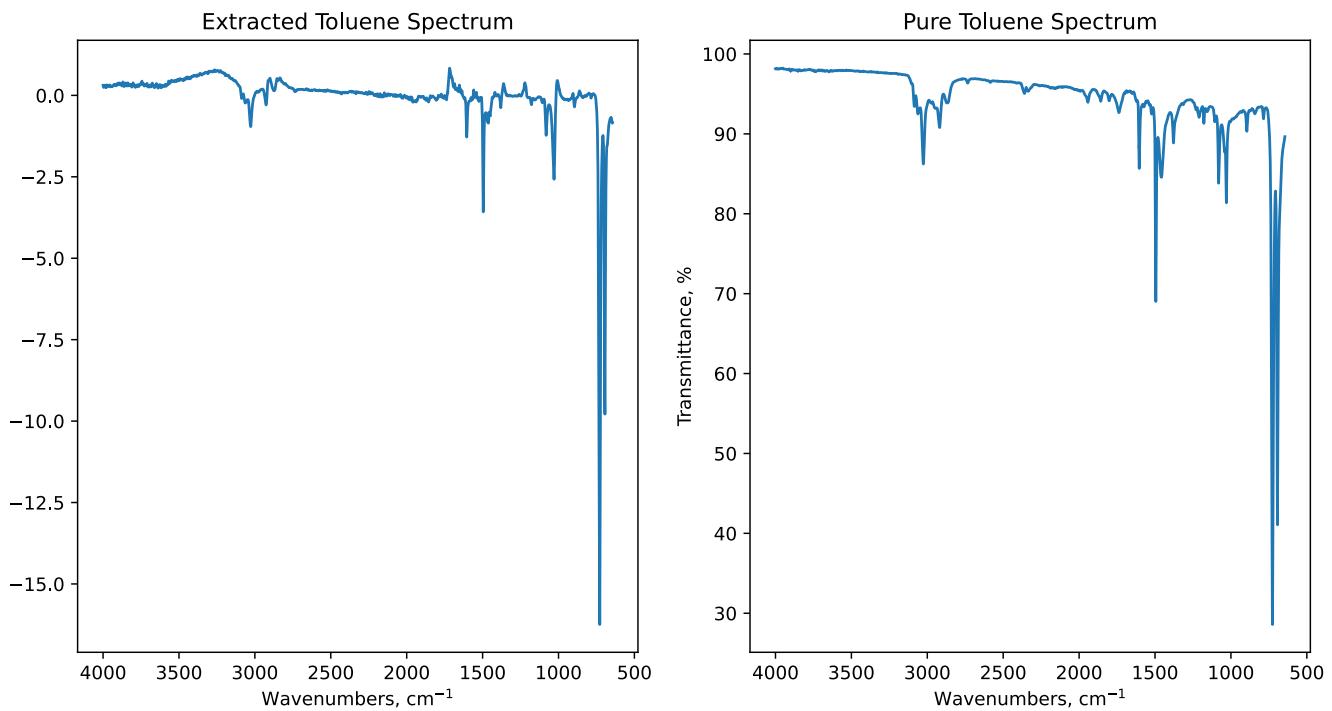


```

fig2 = plt.figure(figsize=(12,6))
ax1 = fig2.add_subplot(1,2,1)
ax1.plot(wn, S_fit[:,0])
plt.xlabel('Wavenumbers, cm$^{-1}$')
plt.title('Extracted Toluene Spectrum')
plt.gca().invert_xaxis()

ax2 = fig2.add_subplot(1,2,2)
ax2.plot(wn, S_pure[:,1])
plt.xlabel('Wavenumbers, cm$^{-1}$')
plt.ylabel('Transmittance, %')
plt.title('Pure Toluene Spectrum')
plt.gca().invert_xaxis()

```

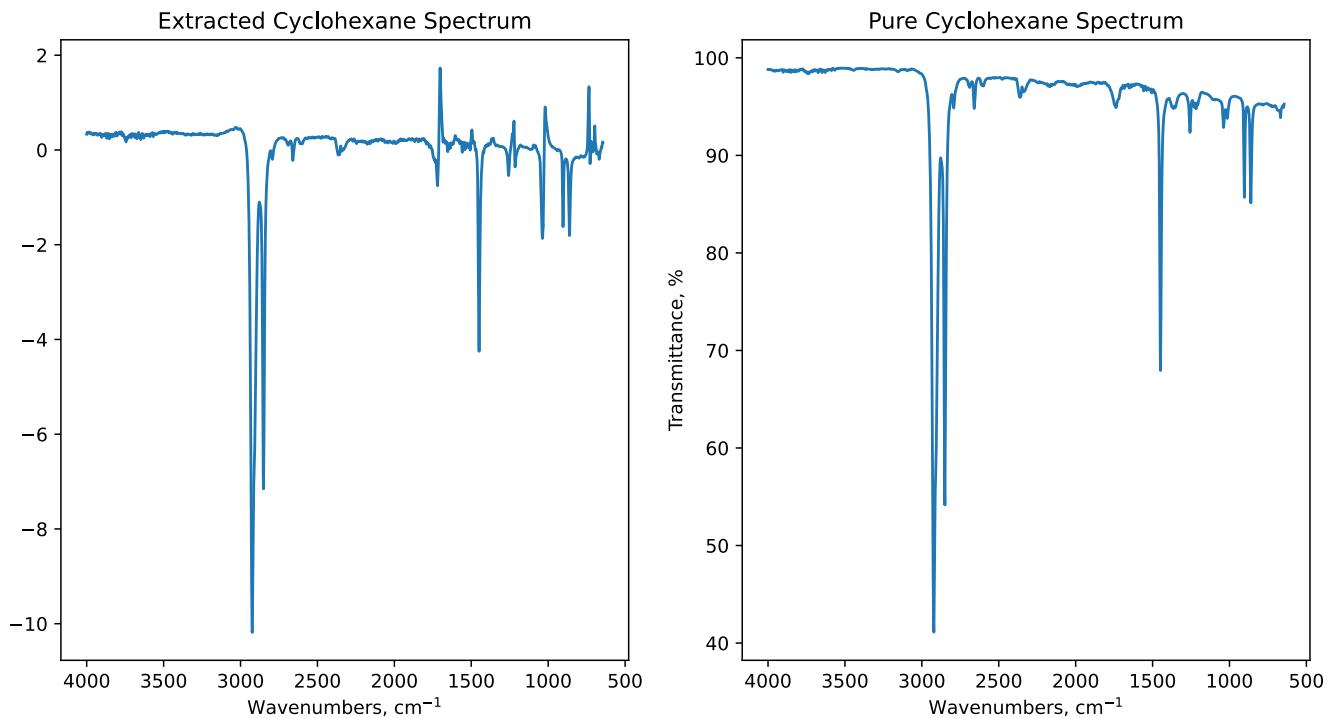


```

fig3 = plt.figure(figsize=(12,6))
ax1 = fig3.add_subplot(1,2,1)
ax1.plot(wn, S_fit[:,1])
plt.xlabel('Wavenumbers, cm$^{-1}$')
plt.title('Extracted Cyclohexane Spectrum')
plt.gca().invert_xaxis()

ax2 = fig3.add_subplot(1,2,2)
ax2.plot(wn, S_pure[:,0])
plt.xlabel('Wavenumbers, cm$^{-1}$')
plt.ylabel('Transmittance, %')
plt.title('Pure Cyclohexane Spectrum')
plt.gca().invert_xaxis()

```

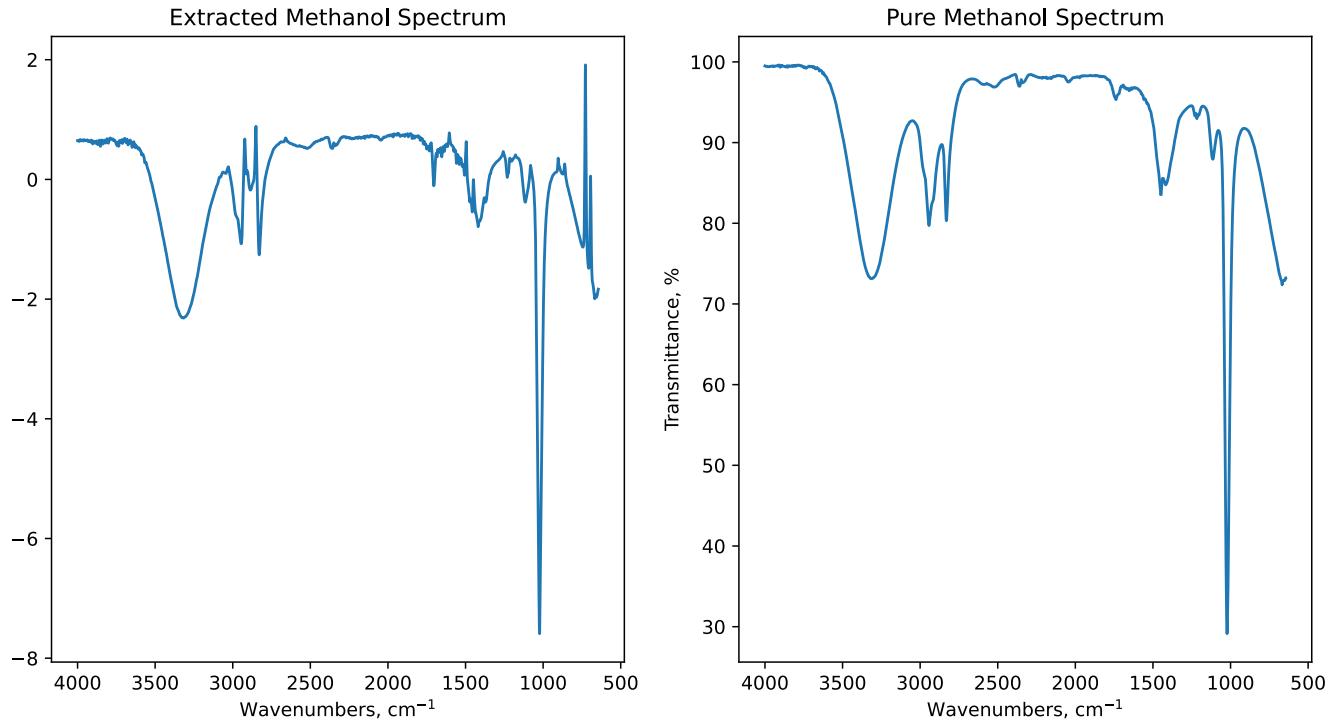


```

fig4 = plt.figure(figsize=(12,6))
ax1 = fig4.add_subplot(1,2,1)
ax1.plot(wn, S_fit[:,3])
plt.xlabel('Wavenumbers, cm$^{-1}$')
plt.title('Extracted Methanol Spectrum')
plt.gca().invert_xaxis()

ax2 = fig4.add_subplot(1,2,2)
ax2.plot(wn, S_pure[:,3])
plt.xlabel('Wavenumbers, cm$^{-1}$')
plt.ylabel('Transmittance, %')
plt.title('Pure Methanol Spectrum')
plt.gca().invert_xaxis()

```



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 requires each component to be non-negative. Finally, you may notice that there is a broad feature at around 3400 cm⁻¹ in the acetone 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 shifting the O-H band breaking down the assumption that the spectra of mixtures are purely additive.

13.3 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 Reader

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 (free resource)

2. VanderPlas, J. Python data Science Handbook: Essential Tools for Working with Data, 1st ed.; O'Reilly: Sebastopol, CA, 2017, chapter 5. Freely available from the author at <https://jakevdp.github.io/PythonDataScienceHandbook/> (free resource)
3. Müller, A. C.; Guido, S. *Introduction to Machine Learning with Python: A Guide for Data Scientists*, O'Reilly: Sebastopol, CA, 2016. -

This book is general introduction to machine learning using scikit-learn and discusses many of the algorithms.

4. Géron, A. Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, 1st ed.; O'Reilly: Sebastopol, CA 2017.

This book provides a deeper discussion into the algorithms behind machine learning and provides an introduction into both scikit-learn and TensorFlow. A newer addition is also available that also provides an introduction to the keras machine learning library. The math is relatively approachable for someone without a strong math background, and the math can be glossed over if need be.

5. 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 article that applies scikit-learn to a chemical problem using both supervised and unsupervised techniques. <https://doi.org/10.1021/acssensors.5b00029>

6. Chen, J.; Wang, X. Z. A New Approach to Near-Infrared Spectral Data Analysis Using Independent Component Analysis. *J. Chem. Inf. Comput. Sci.* **2001**, 41, 992–1001.

This article provides extra background on how [principle component analysis \(PCA\)](#) and [independant component analysis \(ICA\)](#) work, among other topics, and apply ICA to analyzing chemical mixtures using near-infrared spectroscopy. <https://doi.org/10.1021/ci0004053>

Exercises

Complete the following exercises in a Jupyter notebook and scikit-learn library. Any data file(s) referred to in the problems can be found in the [data](#) folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

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.
2. Open the file titled **NMR_mixed_problem.csv** which contains three ^1H NMR spectra. Each spectrum (columns) is a mixture of three chemical compounds in different ratios (artificially generated). Use fastICA to separate out three pure ^1H NMR spectra of each component. Compare your separated spectra to the pure NMR spectra in **NMR_pure_problem.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. You will need to provide this algorithm the number of clusters you feel is most appropriate.
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, and use `plt.cm.get_cmap('turbo', 10)` to generate a colormap with ten colors. You will need to import `PCA` from `sklearn.decomposition`.
 - b) Train the Gaussian Naive Bayes algorithm to classify the digits. Be sure to evaluate the effectiveness using a testing dataset. Import `GaussianNB` from `sklearn.naive_bayes`.

Chapter 14: Optimization and Root Finding

Contents

- 14.1 Minimization
- 14.2 Fitting Equations to Data
- 14.3 Root Finding
- Further Reading
- Exercises

Optimization is the process of improving something to the extent that it cannot be reasonably improved any further. This often involves maximizing desirable attributes and/or minimizing those that are undesirable, so finding the maximum and minimum are common optimization goals. While you may or may not have previously worked *directly* with optimization, you almost certainly have used it as part of a larger application or task such as energy minimization of a molecule, regression analysis, or a number of machine learning algorithms.

In optimization tasks, we often find ourselves searching for the maximum or minimum of a given mathematical function. If we, for example, seek to minimize a function $f(a, b)$, our goal is to find values for input variables a and b to generate the smallest possible output from the function f . One approach is to manually try different input values until you get the smallest possible output, but this kind of tedious and time-consuming task is best left to computers. The

`scipy.optimize` module contains a number of tools for performing optimizations of mathematical functions. The goal of this chapter is to introduce the `scipy.optimize` module and apply it to chemical applications. This chapter does not go into the deeper theory behind optimization, such as specific algorithms. For those interested in some of the deeper theory of optimization, see the [Further Reading](#) section.

Before we begin, we first need to address how we measure what is "best"? For this, we use a *cost function*, also known as an *objective function* or *criterion*, which is a mathematical function that takes in features and returns a value that is a measure of "goodness." If we were a company that is trying to maximize our profits, the objective function would likely be some mathematical equation that calculates our net profit. Optimization of a molecule's conformation involves minimizing the energy, so the objective function here is the function that calculates the energy of the molecule based on the attributes like bond angles and lengths. In the examples below, each of the `scipy.optimize` functions takes as its first argument an objective function in the form of a Python function.

```
scipy.optimize.func(obj_func)
```

The examples in this chapter assume the following imports from NumPy, SciPy, pandas, and matplotlib.

```
import numpy as np
import pandas as pd
from scipy import optimize
import matplotlib.pyplot as plt
```

14.1 Minimization

The first task we will look at is minimization, and for this, `scipy.optimize` has two related functions `scipy.optimize.minimize()` and `scipy.optimize.minimize_scalar()`. Both functions minimize the provided function, but the difference is in the number of independent variables that the objective function takes. A function with only one independent variable, $f(a)$, is known as *univariate* while a function that takes multiple independent variables, $f(a, b, \dots)$, is known as *multipariable*. The `minimize()` function can minimize either multipariable and univariate functions while `minimize_scalar()` can only accept univariate objective functions.

14.1.1 Univariate Minimization

If we are trying to minimize a function with a single independent variable, the `scipy.optimize.minimize_scalar()` is likely a good choice. As a simple example, we will find the radius of minimal energy for two xenon atoms using the Lennard-Jones equation below which describes the potential energy with respect to the distance, r , between the two atoms. In this example, $\sigma = 4.10$ angstroms and $\epsilon = 1.77$ kJ/mol.

$$PE = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right]$$

Being that energy described by the Lennard-Jones energy equation is what we are trying to minimize, this is our objective function. We first need to define this equation as a Python function.

```
def PE_LJ(r):
    epsilon = 4.10 #kJ/mol
    sigma = 1.77 #angstroms
    PE = 4 * epsilon * ( (sigma/r)**12 - (sigma/r)**6)
    return PE
```

Next, we will feed our objective function into the `scipy.optimize.minimize_scalar()` function along with some constraints. This is known as *constrained optimization* and is accomplished by setting the `method='bounded'` and setting the `bounds=` to the range of values the function will operate in. In this case, we are constraining the values of r to a specific range.

```
scipy.optimize.minimize_scalar(func, bounds=(start, stop), method=)
```

Creating bounds is typically optional, but if you know roughly where the minimum will be or where it cannot be, this is helpful information. In this example, it is important to provide constraints on r to ensure the `minimize_scalar()` function does not try $r = 0$ and generate a `ZeroDivisionError`.

```
opt = optimize.minimize_scalar(PE_LJ, bounds=(0.1,100),
                               method='bounded')
opt
```

```
message: Solution found.
success: True
status: 0
fun: -4.099999999992542
x: 1.986757378942203
nit: 21
nfev: 21
```

Alternatively, we can use the `bracket=(a, b)` argument where $f(b) < f(a)$. This argument is different from the `bounds=` argument in that instead of telling the function a region to search, it tells the `minimize_scalar()` function the *direction* to search for the minimum. The minimum does not need to be between a and b , but it simply tells the function that if it moves in the direction of $a \rightarrow b$, it will be moving *toward* the minimum.

```
opt = optimize.minimize_scalar(PE_LJ, bracket=(0.1,100))
opt
```

```
message:
    Optimization terminated successfully;
    The returned value satisfies the termination criteria
    (using xtol = 1.48e-08 )
success: True
    fun: -4.099999999999997
    x: 1.9867578344041286
    nit: 23
    nfev: 26
```

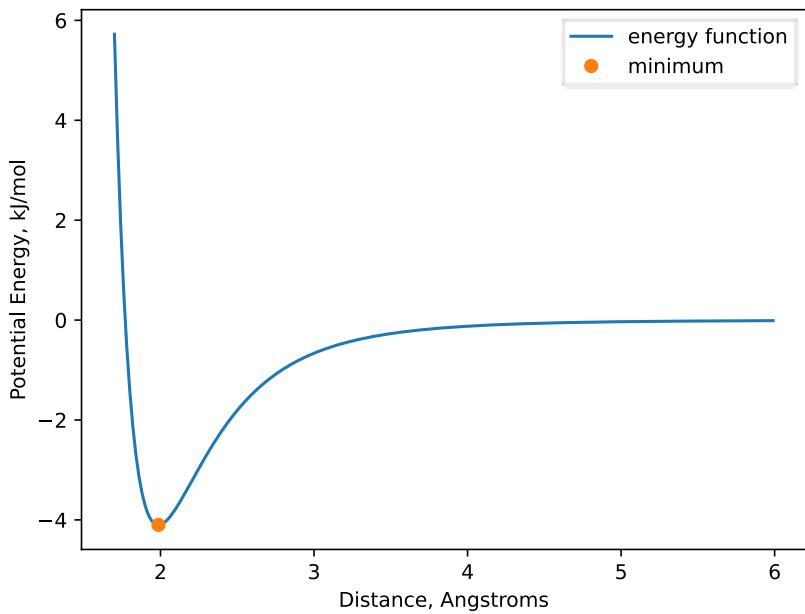
After running our optimization function, an `OptimizeResult` object is returned. This object has a series of attributes listed above, but the two most important are `success` and `x`. The `success` attribute tells us if the optimization function was successful at converging on a solution while the `x` attribute is the optimized solution. We can access the solution using `opt.x` to learn that the minimized distance according to the Lennard-Jones energy equation is 1.99 angstroms.

```
opt.x
```

```
np.float64(1.9867578344041286)
```

Being that our energy function is only univariant, we can easily visualize the function and our minimized solution (orange dot) as done below.

```
r = np.arange(1.7, 6, 0.01)
PE = PE_LJ(r)
plt.plot(r, PE, label='energy function')
plt.plot(opt.x, PE_LJ(opt.x), 'o', label='minimum')
plt.xlabel('Distance, Angstroms')
plt.ylabel('Potential Energy, kJ/mol')
plt.legend();
```



👉 How it works...

The goal of optimization is to minimize the objective function which can be accomplished through a number of algorithms. Knowledge of these algorithms is not required to use optimization, but if you are curious, here is the view from 10,000 feet. Despite the wide variety of algorithms available, they generally operate by an almost trial-and-error approach. They start with initial input values for the objective function and then try slightly different input values. If the new input values decrease the objective function, they are accepted, and if they increase the objective function, they are rejected. This continues on for a number of iterations finding values that progressively decrease the objective function until the algorithm can no longer minimize the objective function any further. The final input values are then returned by the optimization function as the optimized values. Optimization algorithms can differ by, for example, how they decide which input values to try next or how different the subsequent input values to try should be. See [Further Reading](#) for more information on optimization algorithms.

14.1.2 Minimization for Maximization

The SciPy library does not contain any maximization functions, but maximization functions are not really necessary as minimizing the negative of a function provides the maximum. For example, below we have the radial probability function for the hydrogen 3s orbital. For convenience, the SymPy library's `sympy.physics` module is used to generate the 3s radial function (ψ , `psi`) as a Python function. For this maximization example, let's find the radius of maximum probability for the electron. The normalized probability can be calculated by $\psi^2 r^2$ where r is the distance from the nucleus.

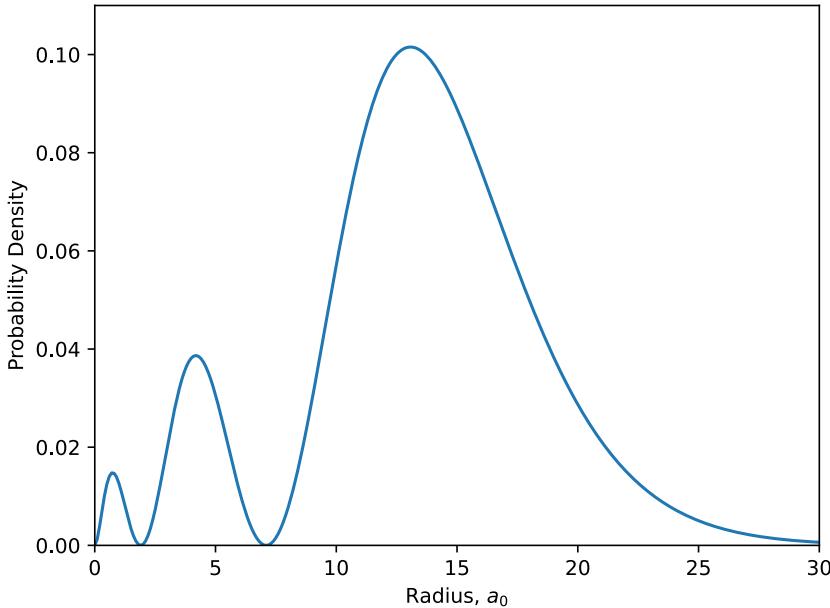
```

import sympy
from sympy.physics.hydrogen import R_nl
R = sympy.symbols('R')

psi_expr = R_nl(3, 0, R) # generate wave function using SymPy
psi = sympy.lambdify(R, psi_expr, 'numpy') # convert to a Python function

r = np.arange(0,40,0.1)
plt.plot(r, psi(r)**2 * r**2) # r is in bohrs (~0.529 angstroms)
plt.xlim(0,30)
plt.ylim(0,0.11)
plt.margins(x=0, y=0)
plt.xlabel('Radius, $a_0$')
plt.ylabel('Probability Density');

```



There are multiple ways to make the function negative like including a negative sign in the Python function definition. Our Python function has already been created, so below we will make the radial probability density negative using a [lambda](#) function (see [section 2.1.4](#) for review on lambda functions).

```

mx = optimize.minimize_scalar(lambda x: -psi(x)**2 * x**2)
mx

```

```

message:
    Optimization terminated successfully;
    The returned value satisfies the termination criteria
    (using xtol = 1.48e-08 )
success: True
fun: -0.014833612579485785
x: 0.7400370693225894
nit: 13
nfev: 16

```

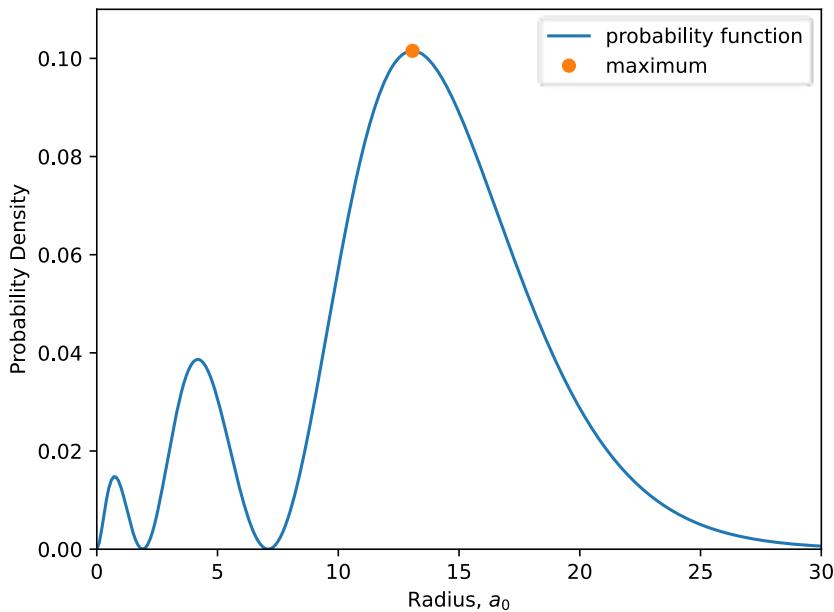
The value returned is the first local maximum but not the global maximum we are seeking. To ensure we get the global maximum, we need to add a constraint for the range of radii used by the optimization function.

```
mx = optimize.minimize_scalar(lambda x: -psi(x)**2 * x**2,
                                bounds=(10,20), method='bounded')
mx
```

```
message: Solution found.
success: True
status: 0
fun: -0.10153431119853075
x: 13.074031887574048
nit: 11
nfev: 11
```

The global maximum is plotted as an orange dot below.

```
plt.plot(r, psi(r)**2 * r**2, label='probability function')
plt.plot(mx.x, psi(mx.x)**2 * mx.x**2, 'o', label='maximum')
plt.xlim(0,30)
plt.ylim(0,0.11)
plt.xlabel('Radius, $a_0$')
plt.ylabel('Probability Density')
plt.margins(x=0, y=0)
plt.legend();
```



14.1.3 Multivariant Minimization

One of the key minimization functions in the `scipy.optimize` module is the `minimize()` function which is capable of minimizing multiple variables simultaneously. This function requires at least two arguments, the objective function and initial guesses for each value as a list or tuple.

```
scipy.optimize.minimize(obj_func, (guess))
```

As an example, we will calculate the equilibrium concentrations for a tandem equilibrium shown below between three different isomers assuming we place an initial 122 mmol of the isomer A into solution and allow it to equilibrate at 25 °C. The two equilibrium constants for this equilibrium are $K_1=5.0$ and $K_2=0.80$.



To solve this problem, we need to adjust the three isomer concentrations, our variables, such that they get as close as possible to the equilibrium ratios set by the equilibrium constants.

The first step is to write an objective function as a Python function, `obj_func()`, that quantifies how poor the solution is. It is the value from this function that we are minimizing to generate the optimal solution to our problem. Being that our goal is to bring the isomer quantities as close to the equilibrium ratios as possible, a reasonable objective function will calculate how far our isomer ratios are from equilibrium. The quality of our solution will be calculated from the squares of the difference between a proposed solution and the target equilibrium constants so that the further the proposed solution is from the target equilibrium values, the exponentially worse the quality of the solution will be evaluated as.

```
K1, K2 = 5.0, 0.80

def obj_func(guess):
    A, B, C = guess

    Q1 = B/A # reaction quotient
    Q2 = C/B # reaction quotient

    quality = (Q1 - K1)**2 + (Q2 - K2)**2

    return quality
```

Next, we provide the `minimize()` function both our objective function and an initial guess for the quantities A, B, and C. The initial guess needs to be a single collection such as a tuple, array, or list. The output of the `minimize()` function is again an `OptimizeResult` object with the `x` attribute accessing the minimized quantities for A, B, and C, respectively.

```
guess = (0.5, 0.25, 0.25)
equ = optimize.minimize(obj_func, guess)
equ
```

```
message: Optimization terminated successfully.
success: True
status: 0
fun: 9.425980662206073e-14
      x: [ 1.917e-01  9.583e-01  7.667e-01]
      nit: 8
      jac: [-5.538e-06  3.657e-06 -1.149e-07]
      hess_inv: [[ 3.200e-03  1.199e-02  5.102e-03]
                  [ 1.199e-02  5.824e-02  2.450e-02]
                  [ 5.102e-03  2.450e-02  4.582e-01]]
      nfev: 60
      njev: 15
```

To access the minimized values, use `equ.x` in this example. We can then verify the results by calculating the equilibrium values based on the calculated equilibrium quantities.

```
equ.x[1]/equ.x[0]
```

```
np.float64(5.00000300516061)
```

```
equ.x[2]/equ.x[1]
```

```
np.float64(0.799999371517403)
```

Both values are in excellent agreement with K_1 and K_2 listed above. One step still remains to solve our problem. In the above problem, it is stated that we started with 122 mmol of isomer A, so if we take the sum of the quantities of A, B, and C, they need to equal 122.

```
np.sum(equ.x)
```

```
np.float64(1.9166790953545365)
```

They do not total to 122 mmol, so we need to scale the quantities up to a total of 122 mmol. Keep in mind that scaling up our values for A, B, and C will not change the ratios.

```
scale_factor = 122 / np.sum(equ.x)
scale_factor * equ.x
```

```
array([12.1999972, 61.00000228, 48.79999799])
```

The final equilibrium quantities for A, B, and C are 12.2, 61.0, and 48.8 mmol, respectively.

⚠ Warning

It is important to recognize that just because an optimization function generates an answer does not mean that it is indeed the correct answer for your problem. The generated answer is the optimization algorithm's best effort in producing the optimal result which may be, for example, a local minimum instead of the global minimum. If there is a way to verify the answer such as is done in the equilibrium example above, this is a prudent last step before using this information.

14.2 Fitting Equations to Data

An common application of optimization is fitting an equation to a series of data points such as a linear regression. While linear regression also happens to have an analytical solution demonstrated in [section 8.3.3](#), we will solve it here using optimization. In the figure below, a regression line (solid orange) runs through the data points. The *residuals* are the difference between the regression line and the data points (green vertical dotted lines). The goal of linear regression is to generate a regression line that minimizes these residuals.

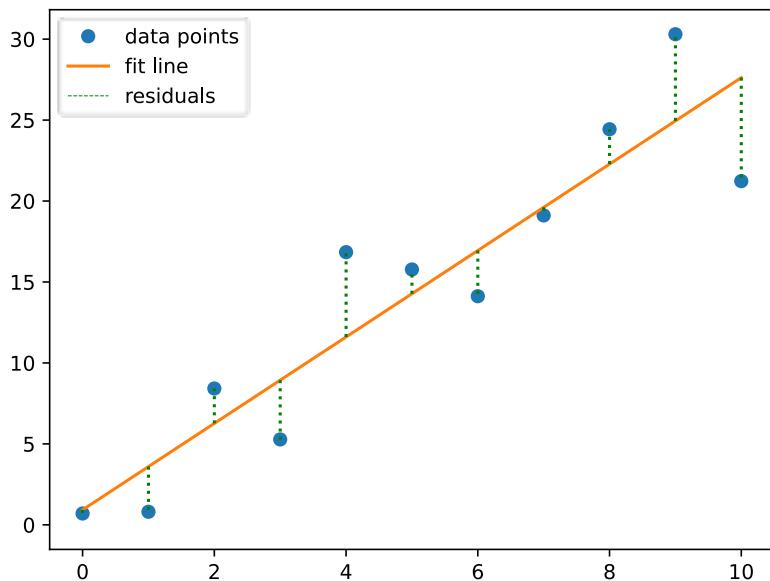


Figure 1 An example of a line of best fit (solid orange) running through data points (blue) with residuals (green dashed) shown as the difference on the y -axis between the data point and linear regression.

One of the major questions in regression is how do we measure the quality of the fit. We could in principle use the total absolute sum of the residuals, known as the *least absolute deviation* cost or objective function, but the commonly accepted objective function for fitting equations to data is *mean square error (MSE)* function. This is the average of the square of the difference between the equation's predictions and the actual data points, or another way of wording this is MSE is the average square residual of the fit line. The MSE equation is shown below where f_i is the y -value from the regression line, y_i is the data point y -value, and N is the number of data points.

$$MSE = \frac{1}{N} \sum_{i=1}^N (f_i - y_i)^2$$

There are two general types of regression: linear regression and nonlinear regression. The key difference is that the former fits data to a linear equation (or plane or hyperplane for higher dimensions) while the latter fits data to nonlinear equations.

14.2.1 Linear Equations

There are numerous examples of linear equations in chemistry, and often when equations are nonlinear, they can be rearranged into a linear form. One classic example of a linear trend is the absorption of light being passed through a solution of colored analyte (i.e., material being quantified) with respect to the concentration of the analyte. This is related by Beer's law shown below where A is absorption, ϵ is the molar absorptivity constant for a particular analyte, b is path length of the sample, and C is the concentration of analyte.

$$A = \epsilon b C$$

Being that the path length for our instrument is 1 cm, which is quite common, this equation simplifies to the following.

$$A = \epsilon C$$

By measuring the absorbance of multiple samples of analyte at known concentrations, the absorbance can be plotted with respect to concentration, and the slope of the linear trend is the molar absorptivity, ϵ .

As our sample data, let's again use the copper cuprizone data we saw in chapter 8.

Table 1 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

```
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])
```

The function we will use to fit this data is the `optimize.curve_fit()` function which performs a least-square minimization that fits an equation to the data provided. Despite this function being often described for fitting an equation to nonlinear data, this function is highly versatile and can fit both linear and nonlinear data. This function requires the theoretical equation, `func`, in the form of a Python function, the independent variable, `xdata`, and the dependent variable, `ydata`. The `curve_fit()` function also allows the user to optionally provide an initial guess for the equation variables/constants, `p0`. This can help speed up the process for more challenging problems and helps ensure the algorithm converges on a reasonable solution.

```
optimize.curve_fit(func, xdata, ydata, p0=())
```

Below we have defined a Python function describing our equation that will be used to fit the data. The Python function used with `optimize.curve_fit()` requires that the first argument of the Python function must be the independent variable(s), and all the rest of the arguments are the parameters used to fit the equation to the data. In this case, these are the slope, m , and the y -intercept, b .

```
def lin_func(x, m, b):
    return m*x + b
```

The objective function is then provided to the `optimize.curve_fit()` function along with the data to fit. The `curve_fit()` function returns two arrays: the optimized parameters and the estimated covariance of the optimized parameters. We are only concerned with the optimized parameters right now, so we use the `_` junk variable to hold the covariance array.

```
const, __ = optimize.curve_fit(lin_func, C, A)
const
```

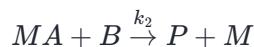
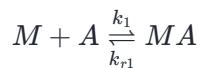
```
array([ 1.55886228e+04, -5.51832054e-06])
```

According to the `curve_fit()` function, the slope is $1.55 \times 10^4 \text{ cm}^{-1}\text{M}^{-1}$ while the y -intercept is -5.45×10^{-6} .

14.2.2 Nonlinear Regression

Optimization can also be used to find the best fit for nonlinear data based off of a theoretical equation. One application of nonlinear fitting is to fit data to a theoretical rate law as a means of determining one or more rate constants in the equation. For this, we will again use the `curve_fit()` function from the `scipy.optimize` module.

To demonstrate this process, let's consider the two-step reaction of $\text{A} + \text{B} \rightarrow \text{P}$ catalyzed by a metal catalyst M.



The theoretical rate law for this two-step reaction is shown below.

$$\text{Rate} = \frac{k_2 k_1 [\text{M}][\text{A}][\text{B}]}{k_{r1} + k_2 [\text{B}]}$$

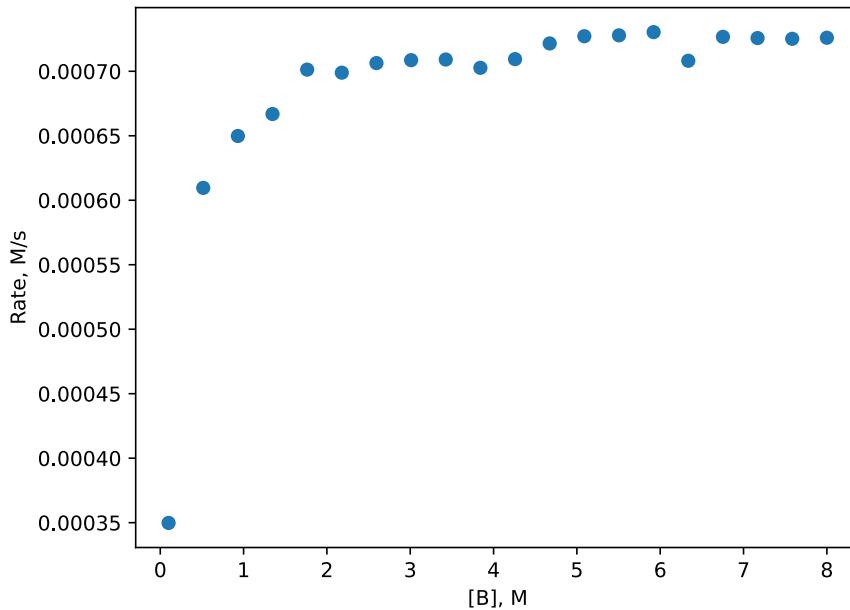
We need to again define the theoretical equation in the form of a Python function. Our function calculates the rate of the chemical reaction versus the concentration of B, but it would also work using data for rate versus the concentration of A depending upon what data you happen to have.

```
def frate(B, k1, kr1, k2):
    rate = (k2 * k1 * M * A * B)/(kr1 + k2 * B)
    return rate
```

For our example, we will generate some simulated data with random noise mixed in it. The values of our rate constants will be $k_1=1.2$, $k_{r2}=0.48$, $k_2=4.29$, and we will set $[\text{A}] = 0.50 \text{ M}$ and $[\text{M}] = 1.2 \times 10^{-3} \text{ M}$. The concentrations of $[\text{A}]$ and $[\text{M}]$ are unchanged during the course of the rate measurement (e.g., using the method of initial rates).

```
M, A = 1.2e-3, 0.50
k1, kr1, k2 = 1.2, 0.48, 4.29
points = 20
conc = np.linspace(0.1, 8, points)
rng = np.random.default_rng(seed=18)
rate = frate(conc, k1, kr1, k2) + rng.random(points)/40000
```

```
plt.plot(conc, rate, 'o')
plt.xlabel('[B], M')
plt.ylabel('Rate, M/s');
```



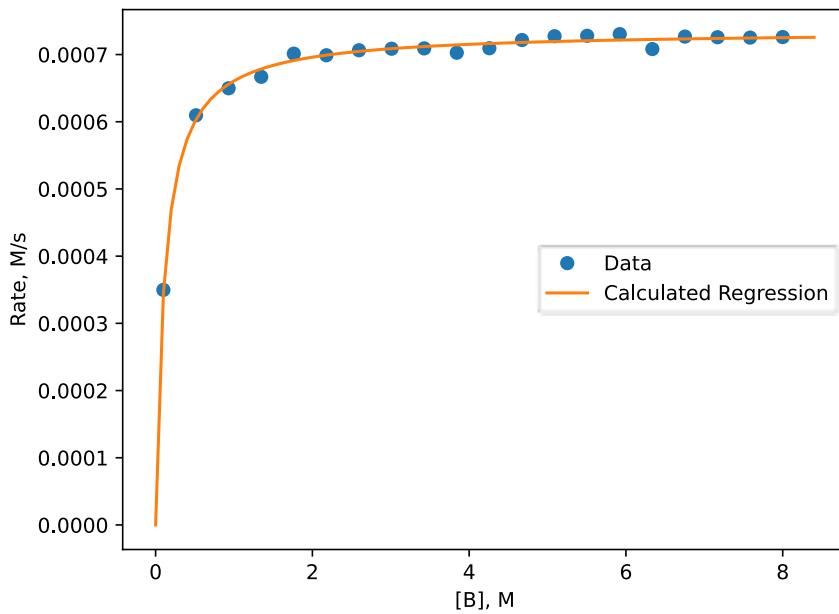
Now that we have our data, we can fit it to the theoretical equation to extract the rate constants.

```
const, __ = optimize.curve_fit(frate, conc, rate, bounds=(0, 5))
const
```

```
array([1.22558323, 0.50705095, 4.49294906])
```

These rate constants are in good agreement with those used to generate the data. We can also plot the simulated data versus the rate equation generated by our curve fitting below.

```
plt.plot(conc, rate, 'o', label='Data')
x = np.arange(0, 8.5, 0.1)
plt.plot(x, frate(x, const[0], const[1], const[2]),
        '--', label='Calculated Regression')
plt.xlabel('[B], M')
plt.ylabel('Rate, M/s')
plt.legend(loc=7);
```



Note

If you are optimizing a function with multiple parameters, bounds are formatted with two lists or tuples. The first contains the lower bounds while the second contains the upper bounds as demonstrated below.

```
bounds = ((a_low, b_low, c_low), (a_high, b_high, c_high))
optimize.curve_fit(func, xdata, ydata, bounds=bounds)
```

Another feature of the `optimize.curve_fit()` function is that it also accepts the uncertainty or errors in each data point. All regression examples seen so far in this book assume that each data point has the same level of uncertainty, but it is not uncommon for data to have different uncertainties. If your uncertainty varies, you can provide the `curve_fit()` function with the uncertainties as standard deviations to the `sigma=` argument as an array-like object (e.g., list, set, or NumPy array). When uncertainties are provided, data points with more uncertainty have less influence on the resulting regression than data points with less uncertainty. See the `scipy.optimize.curve_fit()` documentation for more information and options.

In the example below, we will again fit concentration versus kinetic rate data from the above two-step chemical reaction. This time, we also have an array, `uncertainty`, that provides degrees of uncertainty for the rates.

```
uncertainty = [0.10e-6, 0.12e-6, 0.15e-6, 0.18e-6, 2.0e-6,
               2.1e-6, 2.3e-6, 2.6e-6, 2.9e-6, 3.0e-6,
               3.0e-6, 3.1e-6, 2.9e-6, 3.5e-6, 3.9e-6,
               4.0e-6, 4.1e-6, 4.4e-6, 5.7e-6, 5.3e-6] # M/s

const, __ = optimize.curve_fit(frate, conc, rate,
                               sigma = uncertainty, bounds=(0, 5))
const
```

```
array([1.21309944, 0.48076176, 4.48808724])
```

Comparing these constants to those calculated with the assumption of constant uncertainty, the values are similar but have a noticeable difference. The general rule is that the greater the *variation* in the uncertainties, the more the constants

will differ from those derived with the assumption of constant uncertainty.

Note

Fitting data to a mathematical function can also be accomplished using the `optimize.least_squares()` function. The key difference between using `curve_fit()` and `least_squares()` is that the former accepts the theoretical equation and data directly while the latter requires a Python function that calculates the `residuals`. Interestingly, the source code for the `curve_fit()` function calls the `least_squares()` function. We use the `curve_fit()` function here as it is more intuitive and convenient.

There is another related function, `optimize.leastsq()`, that performs a similar operation but only uses the Levenberg-Marquardt algorithm and is described as legacy on the scipy.org website. The `optimize.least_squares()` function is more versatile and is likely the better choice of the two.

14.2.3 Mixed Analyte Example

Below is an additional example where we use optimization to determine the concentrations of three different dyes mixed together and analyzed by UV-Vis spectroscopy. This example was inspired by a *Journal of Chemical Education* article by [Jesse Maccione, Joseph Welch, and Emily C. Heider](#). By Beer's law, the absorbance (A) of an analyte is the product of the molar absorptivity constant (ϵ) for that analyte, the path length in cm (b), and concentration (C).

$$A = \epsilon b C$$

If there are multiple analytes in solution, the total absorbance (A_{tot}) is equal to the sum of the absorbances for the individual analytes. In our example, we will be dealing with a mixture of red, blue, and yellow dyes.

$$A_{tot} = A_{red} + A_{blue} + A_{yellow}$$

We ultimately want concentrations of the dyes, so we can substitute in Beer's law for the three dye absorbances.

$$A_{tot} = \epsilon_{red} b C_{red} + \epsilon_{blue} b C_{blue} + \epsilon_{yellow} b C_{yellow}$$

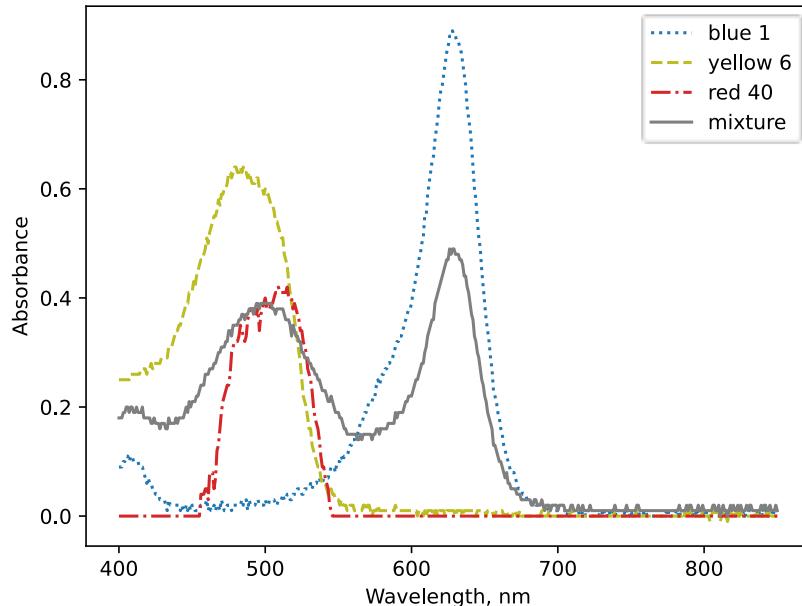
The path length is a constant that depends upon the instrument, and the molar absorptivity constants (ϵ) are constants that depend upon the analytes and the wavelength we are measuring absorbances at. This means that for a particular set of dyes and instrument, the total absorbance (A_{tot}) depends upon the unknown concentrations of individual dyes. Because we have three unknowns, we need three equations to solve for the unknowns. This can be accomplished by measuring the absorbance and molar absorptivity at a minimum of three different wavelengths as demonstrated in [section 8.3.2](#). In this chapter, we will instead measure absorbances at every nanometer from 400 nm to 850 nm and allow the optimization function to fit the total absorbances by adjusting the individual dye concentrations.

First, we will import the absorbance data from the `food_coloring.csv` file using pandas and plot it to see what the data look like. In the CSV file, there are UV-Vis spectra for pure red, pure blue, pure yellow, and a mixture of the three.

```
data = pd.read_csv('data/food_coloring.csv')
data.index = data['nm']
data.drop('nm', axis=1, inplace=True)
```

```
A_red = data['red_40']
A_yellow = data['yellow_6']
A_blue = data['blue_1']
A_mix = data['mix_1']
```

```
plt.plot(data.index, A_blue, c='C0', linestyle=':')
plt.plot(data.index, A_yellow, c='C8', linestyle='--')
plt.plot(data.index, A_red, c='C3', linestyle='-.')
plt.plot(data.index, A_mix, c='C7')
plt.xlabel('Wavelength, nm')
plt.ylabel('Absorbance')
plt.legend(['blue 1', 'yellow 6', 'red 40', 'mixture']);
```



Next, we will use the absorbances for each pure dye sample to find the molar absorptivities using Beer's law. The path length, b , in this instrument is 1 cm, and the molarities are known from the experimental setup. That is, below we are solving for molar absorptivity (ϵ) by the following.

$$\epsilon = \frac{A}{C}$$

```
eps_red = A_red / 4.09e-5
eps_blue = A_blue / 5.00e-6
eps_yellow = A_yellow / 2.92e-5
```

Finally, we will write a Python function that calculates the total absorbance from the individual concentrations and molar absorptivities, and we will provide this function to the `optimize.curve_fit()` function. The fitting parameters are the calculated concentrations of the individual dyes.

```
def absorb(spec, C_red, C_blue, C_yellow):
    return eps_red * C_red + eps_blue * C_blue + eps_yellow * C_yellow
```

```
fit, __ = optimize.curve_fit(absorb, data.index, A_mix)
fit
```

```
array([1.44922873e-05, 2.84592011e-06, 1.26645031e-05])
```

The end result is that the red, blue, and yellow dyes have concentrations of 1.45×10^{-5} M, 2.85×10^{-6} M, and 1.27×10^{-5} M.

Below is a quick demonstration on how to also solve this problem using the `optimize.least_squares()` function. As mentioned earlier, both the `curve_fit()` and `least_squares()` functions can be used to solve the same problems. The `least_squares()` function requires a Python function that calculates the residuals (i.e., the difference between the calculated and measured absorbances) instead of the theoretical equation. This function also requires an initial guess for the fit parameters. Even if you don't know the concentrations, just give some reasonable value. In this case, we guessed 1×10^{-3} M for each dye.

```
def residuals(X):
    C_red, C_blue, C_yellow = X
    A_calc = C_red * eps_red + C_blue * eps_blue + C_yellow * eps_yellow
    return A_mix - A_calc

lstsq = optimize.least_squares(residuals, (1e-3, 1e-3, 1e-3))
lstsq.x
```

```
array([1.44922873e-05, 2.84592011e-06, 1.26645031e-05])
```

The resulting concentrations for the three dyes appears identical (or nearly so) to those calculated by the `curve_fit()` function.

14.3 Root Finding

Root finding is the process of determining where a function equals zero, $f(a, b, \dots) = 0$. Being that any equation can be rearranged to equal zero, this is a versatile way of solving an equation. If the function is univariate, $f(a) = 0$, this task may sometimes seem trivial even without optimization algorithms, but as the complexity of the equation or number of variables increases, using optimization algorithms can be beneficial.

Like the minimization functions above, there are two related versions of the root finding functions:

`scipy.optimize.root()` and `scipy.optimize.root_scalar()`. The key difference is that the `root()` function can solve for both univariate and multivariate functions while `root_scalar()` can only solve for univariate functions. Both functions require a function, `func`, to find the root of, and `root()` function also requires an initial guess, `x0`. The `root_scalar()` function also allows for an optional range of values that bracket the root, `bracket=` to be provided by the user.

```
scipy.optimize.root(func, x0)
scipy.optimize.root_scalar(func, bracket=(start, stop))
```

As a root finding example, we can locate the nodes in a radial wave function for the hydrogen 3s orbital. Because there is

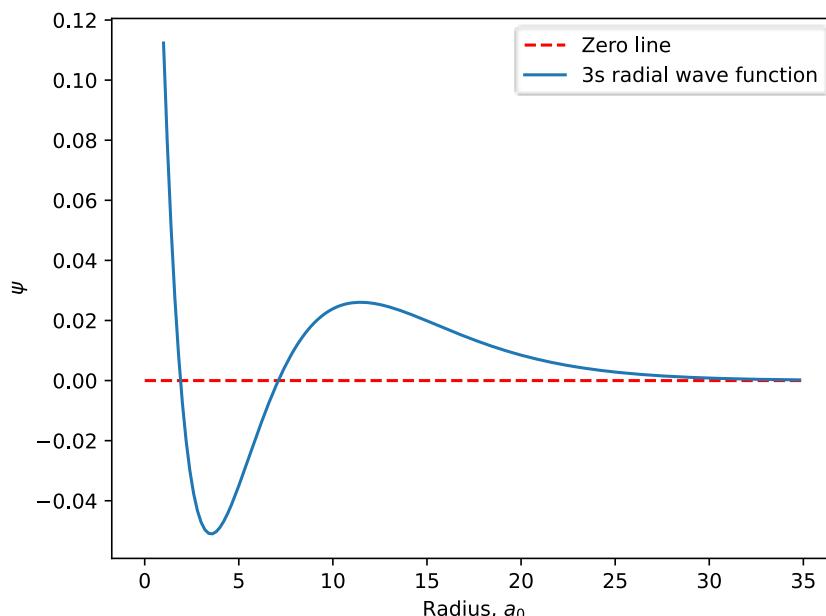
only one variable, r , we can use the `scipy.optimize.root_scalar()` function. Below, we first define our radial wave function as a Python function, `orbital_3s`.

```
def orbital_3s(r):
    wf = (2/27)*np.sqrt(3)*(2*r**2/9 - 2*r + 3)* np.exp(-r/3)
    return wf
```

Before we find the roots, let's visualize the function to see what we are dealing with. The horizontal dotted line at $y = 0$ is provided as a visual guide. The roots are located where the solid line of the wave function intersects with the dotted line.

```
r = np.arange(1, 35, 0.2)
psi_3s = [orbital_3s(num) for num in r]

plt.hlines(0, 0, 35, 'r', linestyles='--', label='Zero line')
plt.plot(r, psi_3s, '-', label='3s radial wave function')
plt.legend()
plt.xlabel('Radius, $a_0$')
plt.ylabel('$\psi$');
```



The function has two nodes, so our `bracket=` values will determine which we will end up solving for.

```
node1 = optimize.root_scalar(orbital_3s, bracket=[0, 3])
node1
```

```
      converged: True
      flag: converged
function_calls: 11
iterations: 10
      root: 1.901923788646684
     method: brentq
```

```
node2 = optimize.root_scalar(orbital_3s, bracket=[5, 10])
node2
```

```

converged: True
    flag: converged
function_calls: 9
iterations: 8
    root: 7.098076211353316
method: brentq

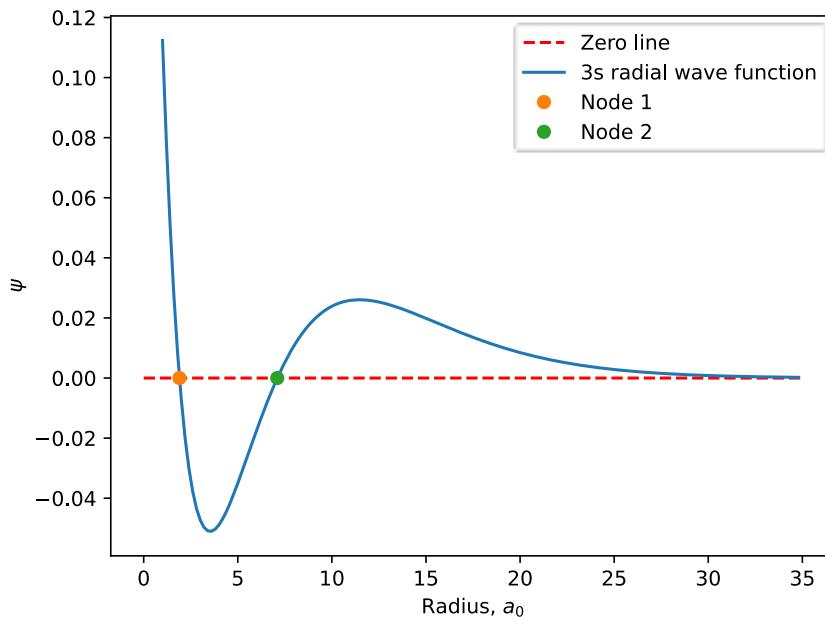
```

```

r = np.arange(1, 35, 0.2)
psi_3s = [orbital_3s(num) for num in r]

plt.hlines(0, 0, 35, 'r', linestyles='--', label='Zero line')
plt.plot(r, psi_3s, '-', label='3s radial wave function')
plt.plot(node1.root, orbital_3s(node1.root), 'o', label='Node 1')
plt.plot(node2.root, orbital_3s(node2.root), 'o', label='Node 2')
plt.xlabel('Radius, $a_0$')
plt.ylabel('$\psi$')
plt.legend();

```



The two dots above show the location of the two roots for this function which clearly are located on the nodes of the wave function.

Further Reading

1. The `scipy.optimize` module user guide. <https://docs.scipy.org/doc/scipy/tutorial/optimize.html> (free resource)
2. Watt, J.; Borhani, R.; Katsaggelos, A. K. Machine Learning Refined: Foundations, Algorithms, and Applications; 2nd ed.; Cambridge University Press, 2020, pp 21-124. These chapters are a good introduction to optimization algorithms.

Exercises

Solve the following problems using Python in a Jupyter notebook and functions from the `scipy.optimize` module. Any data file(s) referred to in the problems can be found in the `data` folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate

chapter file and then clicking the **Download** button.

1. A warm or hot object emits radiation in a range of wavelengths described by Plank's law shown below where B is radiance, λ is frequency of radiation, c is the speed of light, h is Plank's constant, k is Boltzmann's constant, and T is temperature of the object in K.

$$B(\lambda) = \frac{2hc^2}{\lambda^5} \frac{1}{e^{\frac{hc}{\lambda kT}} - 1}$$

Determine the wavelength of greatest radiance for an object at 5000 K using a minimization function. Hint: be sure to include an extra negative sign in the Python function that you define, and you will want to use either bounds or brackets to prevent the minimization function from trying zero and generating a `ZeroDivisionError`.

2. The three isomers of ethyltoluene (i.e., *ortho*-, *meta*-, and *para*-) interchange under Friedel-Crafts conditions facilitated by aluminum chloride. An investigation into this isomer equilibrium by [Allen, R. H. et al.](#) experimentally determined the rate constants for the interconversion of these isomers. Using the rate constant data, the following equilibrium constants were calculated: $K_{om}=7.2$, $K_{pm}=2.47$, and $K_{op}=2.9$ where each equilibrium constant is defined below.

$$K_{om} = \frac{[meta]}{[ortho]}, \quad K_{pm} = \frac{[meta]}{[para]}, \quad K_{op} = \frac{[para]}{[ortho]}$$

Using this information, calculate the percentages of each isomer at equilibrium. Compare your percentage to those provided in the above paper (in the abstract).

3. A sealed piston contains 0.32 moles of helium gas at 298 K. Determine the value of R by performing a nonlinear fit on the data below with the `optimize.curve_fit()` function and the ideal gas law.

$$P = \frac{nRT}{V}$$

Volume (L)	Pressure (atm)
0.401	21.8
0.701	11.3
1.22	5.17
1.80	5.49
2.39	3.86
2.83	4.34
3.09	2.72

4. Below is the theoretical kinetic rate law for a chemical reaction of $A \rightarrow P$ catalyzed by 0.001 M of a metal catalyst C. The table includes kinetic data for the rate, concentration of A, and the uncertainty in rate. Use the `optimize.curve_fit()` function to determine values for k_1 and K_{eq} . Plot the data below with an overlay of calculated values using the constants that you determined to show that they are reasonable values.

$$Rate = \frac{k_1 K_{eq}[A][C]}{1 + K_{eq}[A]}$$

Rate, M/s	[A], M	Rate Uncertainty, M/s
2.18e-06	0.01	0.11e-6
1.72e-05	0.71	0.12e-6
2.75e-05	1.43	0.25e-6
4.36e-05	2.14	0.40e-6
5.23e-05	2.86	0.50e-6
5.23e-05	3.57	1.0e-6
6.71e-05	4.29	1.5e-6
6.26e-05	5.00	1.8e-6

5. One method of solving acid-base equilibrium concentrations is through polynomials as demonstrated by [F. Bamdad](#). Below is a third-degree polynomial from the equilibria resulting from placing hydrocyanic acid (HCN) in water where x is the concentration of hydronium, K_a is the acid equilibrium constant, K_w is equilibrium constant for the autoionization of water, and $[HCN]_0$ is the initial concentration of hydrocyanic acid. Solve for the concentration of hydronium using a root finding algorithm in the `scipy.optimize` module assuming $[HCN]_0 = 6.8 \times 10^{-6}$ M and $K_a = 6.2 \times 10^{-10}$.

$$x^3 + K_a x^2 + (K_w + [HCN]_0 K_a)x - K_w K_a = 0$$

6. The van der Waals equation is a modified form of the ideal gas law but includes two correction factors that account for intermolecular forces and the volume of gas molecules. These correction factors include constants a and b which are gas-dependent, and the values of a and b can be calculated by fitting the van der Waals equation to pressure versus volume data.

$$\left(P + a \frac{n^2}{V^2} \right) (V - nb) = nRT$$

Load the file `PV_C0.csv` containing pressure and volume data for one mole of carbon monoxide at 298 K acquired from the [NIST Chemistry WebBook](#). Fit the van der Waals equation to this dataset to determine a and b values for carbon monoxide.

Chapter 15: Cheminformatics with RDKit

Contents

- 15.1 Loading Molecular Representations into RDKit
- 15.2 Visualizing Chemical Structures
- 15.3 Stereochemistry
- 15.4 `Chem.Descriptor` Module
- 15.5 Searching Molecules for Structural Patterns
- 15.6 Atoms and Bonds
- Further Reading
- Exercises

Cheminformatics can be thought of as the intersection of data science, computer science, and chemistry as a means of better understanding and solving chemical problems. This chapter introduces a popular and versatile Python cheminformatics library known as RDKit which is useful for tasks such as:

- Visualizing molecules
- Reading SMILES or InChI molecular representations
- Quantifying structural features in molecules such as number of rings or hydrogen bond donors
- Generating all possible stereoisomers of a molecular structure
- Filtering molecules based on structural features

This is a popular library for those in chemical computing research with examples of its use being relatively easy to find in the chemical literature. As of this writing, RDKit can be installed with either conda or pip (see [section 0.2.1](#) and link below). If you are using Google Colab, you will need to install RDKit at the top of your notebook (see [section 0.2.2](#)) as it is not installed by default in Colab.

[Installing RDKit](#)

This chapter assumes the following imports from RDKit.

```
from rdkit import Chem
from rdkit.Chem import AllChem, Descriptors, PandasTools
from rdkit.Chem.Draw import SimilarityMaps
from rdkit.Chem import rdFingerprintGenerator

from rdkit.Chem.Draw import IPythonConsole
IPythonConsole.ipython_useSVG = True

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

RDKit is composed of a number of modules including, but not limited to, the following.

Table 1 Key Modules and Submodules in the RDKit Library

Module/ Submodule	Description
<code>Chem</code>	General purpose tools for chemistry. The RDKit website describes it as "A module for molecules and stuff".
<code>Chem.AllChem</code>	Submodule containing more specialized or less often used features; needs to be imported separately from <code>Chem</code>
<code>Chem.Descriptors</code>	Submodule for quantifying molecular features
<code>Chem.Draw</code>	Submodule for visualizing molecules
<code>ML</code>	Machine learning tools

The `Chem` and `ML` modules are the major modules in RDKit, but for this chapter, we will only be focusing on the `Chem` module which has already been imported above.

15.1 Loading Molecular Representations into RDKit

There are many ways to depict molecular structures on paper such as Lewis structures, line-angle structural formulas, and condensed notation. When representing molecules for a computer, machine-readable methods such as Simplified Molecular-Input Line-Entry System (SMILES), the International Chemical Identifier (InChI), or mol files are preferred. For example, the SMILES and InChI representations for benzene are listed below.

```
SMILES: c1ccccc1
```

```
InChI: 1S/C6H6/c1-2-4-6-5-3-1/h1-6H
```

These are not the most human-readable formats, but computer software such as RDKit is quite good at dealing with them. We will not get into the structure and rules for interpreting these representations here because it is not really necessary; reading and writing them is RDKit's job. You can obtain these representations of a molecular structure from a variety of sources such as generating them from chemical drawing software (e.g., ChemDraw or ChemDoodle), searching [NIST Chemical Webbook](#) or [NIH PubChem](#), and many other sources. In this chapter, we will mainly focus on SMILES representations, but working with the InChI and MOL file formats is analogous and may be used from time-to-time herein.

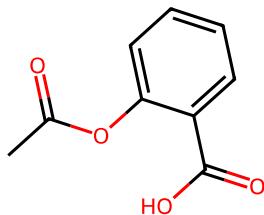
The functions below can read and write molecular structures from a variety of formats including SMILES, InChI, and MOL files. When reading these molecular structures, a Molecule object (RDKit-specific class of object) is generated.

Table 2 Functions for Loading Molecular Structures

Function	Description
<code>Chem.MolFromSmiles()</code>	Generates a Molecule object from SMILES representation
<code>Chem.MolToSmiles()</code>	Generates SMILES representation from a Molecule object
<code>Chem.inchi.MolFromInchi()</code>	Generates a Molecule object from InChI representation
<code>Chem.inchi.MolToInchi()</code>	Generates InChI representation from a Molecule object
<code>MolFromMolFile()</code>	Generates a Molecule object from a MOL file

As an example, we will load the structure of aspirin (acetylsalicylic acid) using the `Chem.MolFromSmiles()` function from the `Chem` module.

```
aspirin = Chem.MolFromSmiles('O=C(C)Oc1ccccc1C(=O)O')
```



If we check the object type, we find that it is a Molecule (`rdchem.Mol`) RDKit object.

```
type(aspirin)
```

```
rdkit.Chem.rdchem.Mol
```

RDKit can generate other molecular representations such as InChI from the Molecule object as demonstrated below.

```
Chem.inchi.MolToInchi(aspirin)
```

```
'InChI=1S/C9H8O4/c1-6(10)13-8-5-3-2-4-7(8)9(11)12/h2-5H,1H3,(H,11,12)'
```

15.2 Visualizing Chemical Structures

In the above examples, RDKit provided an image of the molecule simply by Jupyter running the Molecule object. By default, this generates a rather small and low resolution image. To generate a sharper image, like above, running the following code at the top of a notebook changes the settings to produce SVG (Scalable Vector Graphic) images which are a vector graphic format.

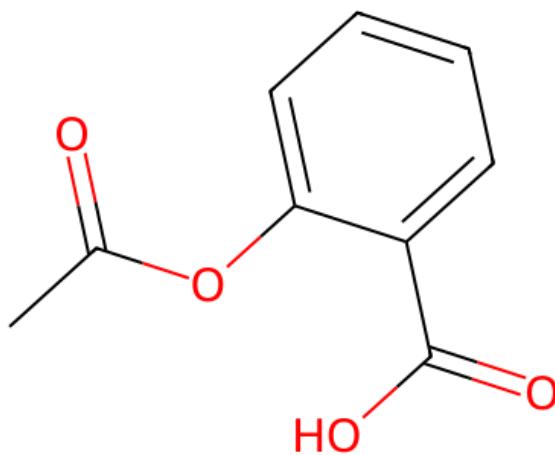
```
from rdkit.Chem.Draw import IPythonConsole
IPythonConsole.ipython_useSVG = True
```

15.2.1 Single Chemical Structures

However, simply running the Molecule object does not provide easy control over the image. In this section, we will generate images that can be saved along with visualizing grids of molecules and other visual representations.

To view the molecule, we can use the `Chem.Draw.MolToImage(Mol)` function which takes one required positional arguments of the Molecule object (`Mol`). Optional keyword arguments can be used to set other parameters such as the image size (`size=`) in pixels.

```
Chem.Draw.MolToImage(aspirin, size=(400,400))
```



If we want to save the image to a file, this is accomplished using the `Chem.Draw.MolToFile()` function which requires two pieces of information - the Molecule object and the name of the new file as a string.

```
Chem.Draw.MolToFile(mol_object, 'file_name.png', size=(width, height), imageType='png')
```

Other optional parameters include the `size=` which is a tuple that takes the width and height, respectively, in pixels, and the `imageType=` accepts a string to designate the file format ('png' or 'svg').

```
Chem.Draw.MolToFile(aspirin, 'aspirin.svg',
                     size=(500,500),
                     imageType='svg')
```

Molecules can also be displayed in plots created by matplotlib. Below is an example of the *trans*-cinnamic acid structure being displayed on top of the IR spectrum of the compound.

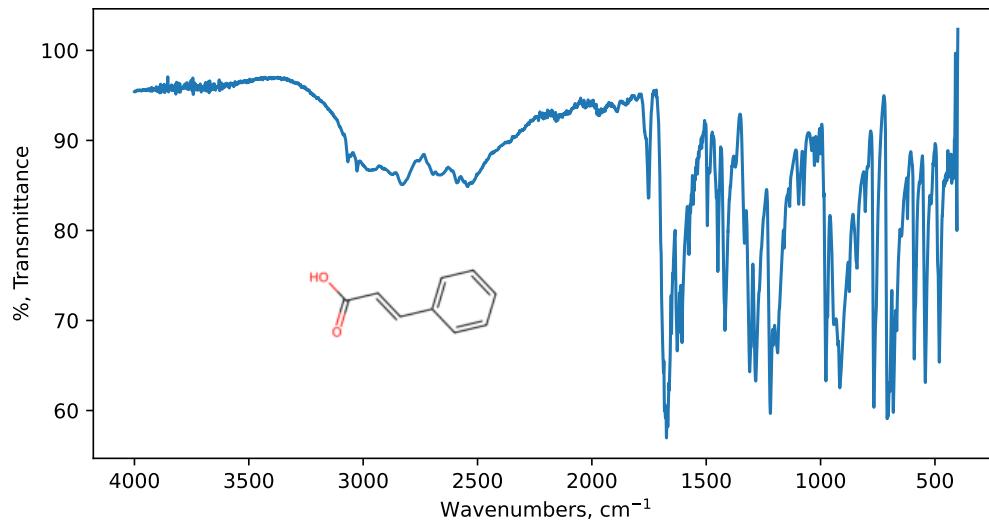
```
cinn_acid = Chem.MolFromSmiles('O=C(0)\C=C\c1ccccc1')
image = Chem.Draw.MolToImage(cinn_acid)
```

```

IR = np.genfromtxt('data/cinnamic_acid.CSV', delimiter=',')
plt.figure(figsize=(8,4))
plt.plot(IR[:-1,0], IR[:-1,1])
plt.gca().invert_xaxis()
plt.xlabel('Wavenumbers, cm$^{-1}$')
plt.ylabel('%, Transmittance')

ax = plt.axes([0.15, 0.2, 0.48, 0.35], frameon=False)
ax.axis('off')
ax.imshow(image);

```



15.2.2 Grids of Chemical Structures

Whenever we are dealing with collections of molecules, it may be helpful to generate an image that includes multiple molecular structures known as a grid. As an example, we will load the SMILES strings of the twenty common amino acids from a text file using pandas and then load the Molecule objects for each structure into a single list called `AminoAcids`.

```

df = pd.read_csv('data/amino_acid_SMILES.txt', skiprows=2)
df

```

	name	SMILES
0	alanine	C[C@@H](C(=O)[O-])[NH3+]
1	arginine	[NH3+][C@@H](CCCNC(=NH2+)N)C(=O)[O-]
2	asparagine	O=C(N)C[C@H]([NH3+])C(=O)[O-]
3	aspartate	C([C@@H](C(=O)[O-])[NH3+])C(=O)[O-]
4	cysteine	C([C@@H](C(=O)[O-])[NH3+])S
5	glutamine	[NH3+][C@@H](CCC(=O)N)C([O-])=O
6	glutamate	C(CC(=O)[O-])[C@@H](C(=O)[O-])[NH3+]
7	glycine	C(C(=O)[O-])[NH3+]
8	histidine	O=C([C@H](CC1=CNC=N1)[NH3+])[O-]
9	isoleucine	CC[C@H](C)[C@@H](C(=O)[O-])[NH3+]
10	leucine	CC(C)C[C@@H](C(=O)[O-])[NH3+]
11	lysine	C(CC[NH3+])C[C@@H](C(=O)[O-])[NH3+]
12	methionine	CSCC[C@H]([NH3+])C(=O)[O-]
13	phenylalanine	[NH3+][C@@H](CC1=CC=CC=C1)C([O-])=O
14	proline	[O-]C(=O)[C@H](CCC2)[NH2+]2
15	serine	C([C@@H](C(=O)[O-])[NH3+])O
16	threonine	C[C@H]([C@@H](C(=O)[O-])[NH3+])O
17	tryptophan	c1[nH]c2cccc2c1C[C@H]([NH3+])C(=O)[O-]
18	tyrosine	[NH3+][C@@H](Cc1ccc(O)cc1)C([O-])=O
19	valine	CC(C)[C@@H](C(=O)[O-])[NH3+]

```
AminoAcids = [Chem.MolFromSmiles(SMILES) for SMILES in df['SMILES']]
```

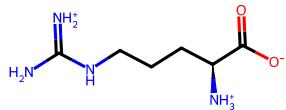
```
[<rdkit.Chem.rdchem.Mol at 0x114cf6490>,
<rdkit.Chem.rdchem.Mol at 0x114cf5310>,
<rdkit.Chem.rdchem.Mol at 0x114cf6ff0>,
<rdkit.Chem.rdchem.Mol at 0x114cf5540>,
<rdkit.Chem.rdchem.Mol at 0x114cf5380>,
<rdkit.Chem.rdchem.Mol at 0x114cf5230>,
<rdkit.Chem.rdchem.Mol at 0x114cf5150>,
<rdkit.Chem.rdchem.Mol at 0x114cf5070>,
<rdkit.Chem.rdchem.Mol at 0x114cf4f90>,
<rdkit.Chem.rdchem.Mol at 0x114cf4eb0>,
<rdkit.Chem.rdchem.Mol at 0x114cf4dd0>,
<rdkit.Chem.rdchem.Mol at 0x114cf4c80>,
<rdkit.Chem.rdchem.Mol at 0x114cf4d60>,
<rdkit.Chem.rdchem.Mol at 0x114cf4ac0>,
<rdkit.Chem.rdchem.Mol at 0x114cf4ba0>,
<rdkit.Chem.rdchem.Mol at 0x114cf49e0>,
<rdkit.Chem.rdchem.Mol at 0x114cf4900>,
<rdkit.Chem.rdchem.Mol at 0x114cf6730>,
<rdkit.Chem.rdchem.Mol at 0x114cf67a0>,
<rdkit.Chem.rdchem.Mol at 0x114cf6810>]
```

To generate the grid, we will use the `MolsToGridImage()` function from the `Chem.Draw` submodule. This function requires one positional argument of an array-like object (e.g., list, tuple, ndarray, etc.) containing the Molecule objects. Other optional keyword arguments include the number of molecules per row (`molsPerRow=`), the pixel dimensions of each molecule (`subImgSize=`), labels below each molecule (`legends=`), and the ability to make images SVG format (`useSVG=`). The image dimensions only matter if using a raster image format and requires a tuple with the width and height in that order. The `legends=` argument requires an array-like object with the labels in the same order as the object containing the Molecule objects.

```
Chem.Draw.MolsToGridImage(AminoAcids,  
                           molsPerRow=4,  
                           subImgSize=(200,200),  
                           legends=list(df['name']),  
                           useSVG=True)
```



alanine



arginine



asparagine



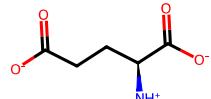
aspartate



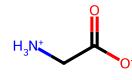
cysteine



glutamine



glutamate



glycine



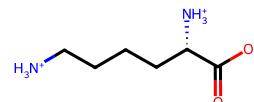
histidine



isoleucine



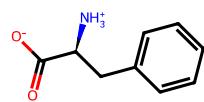
leucine



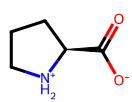
lysine



methionine



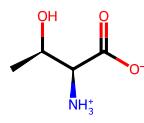
phenylalanine



proline



serine



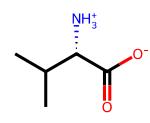
threonine



tryptophan



tyrosine



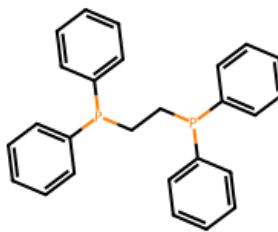
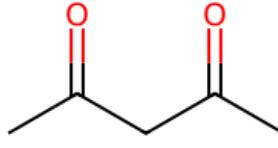
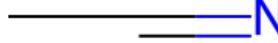
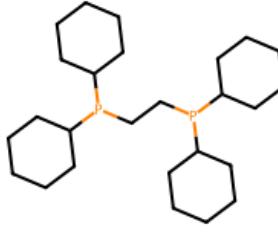
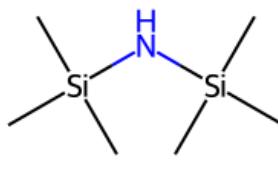
valine

15.2.3 Molecules in Pandas DataFrames

RDKit also supports visualizing molecules inside pandas DataFrames using the `AddMoleculeColumnToFrame()` function from the `PandasTools` submodule (`rdkit.Chem.PandasTools`). This function accepts a DataFrame with a column of SMILES (`smilesCol=`) and adds a new column of Molecule objects. The `molCol=` parameter will be the header for the new column.

```
ligands = pd.read_csv('data/ligands.csv')  
ligands
```

	ligand	smiles
0	dppe	c1ccc(P(CCP(c2cccccc2)c2cccccc2)c2cccccc2)cc1
1	acac	CC(=O)CC(C)=O
2	acetonitrile	CC#N
3	dcpe	C1CCC(P(CCP(C2CCCCC2)C2CCCCC2)C2CCCCC2)CC1
4	HMDS	C[Si](C)(C)N[Si](C)(C)C
5	PPh3	c1ccc(P(c2cccccc2)c2cccccc2)cc1

	ligand	smiles	molecules
0	dppe	c1ccc(P(CCP(c2ccccc2)c2ccccc2)c2ccccc2)cc1	
1	acac	CC(=O)CC(C)=O	
2	acetonitrile	CC#N	
3	dcpe	C1CCC(P(CCP(C2CCCCC2)C2CCCCC2)C2CCCCC2)CC1	
4	HMDS	C[Si](C)(C)N[Si](C)(C)C	

	ligand	smiles	molecules
5	PPh3	c1ccc(P(c2ccccc2)c2ccccc2)cc1	

15.3 Stereochemistry

RDKit can assign the stereochemistry of stereocenters, including chiral centers (*R* vs. *S*) and alkene stereocenters (*E* vs. *Z*), determine the number of isomers possible, and even generate all possible isomers. Whether or not any stereochemistry is designated in the SMILES representation or Molecule object is an important detail in carrying out the above tasks. Even though a molecule may contain a chiral center or an alkene carbon, the stereochemistry around that atom may be ambiguous.

The SMILES representation shows stereochemistry around a tetrahedral carbon with either @ or @@ and around an alkene with \ and / symbols. If the SMILES representation does not include these symbols, the stereochemistry is not indicated.

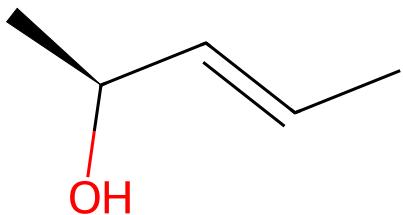
Table 3 SMILES Stereochemical Designations

Designation	Alkene	Chiral sp^3 Atom
No isomer designation	C(=CC)C	CCC(C)O
First isomer	C/C=C\ C	CC[C@H](C)O
Second isomer	C/C=C/C	CC[C@@H](C)O

15.3.1 Assigning Stereochemistry

The first task is to assign the absolute stereochemistry of a molecule. As an example, below we have a single isomer of pent-3-en-2-ol which has a single chiral center and an alkene that could potentially be either *E* or *Z*. Let's have RDKit tell us the absolute configuration (i.e., *R* or *S*) of the tetrahedral chiral center and if the alkene is *E* or *Z*. First, we will load the SMILES representation of this compound, `'O[C@H](C)/C=C/C'`, which contains both @ and / symbols, so we know the stereochemistry is assigned in this representation. When we visualize it below, we can see a wedge for the methyl on the chiral center instead of a regular line, for example.

```
pentenol = Chem.MolFromSmiles('O[C@H](C)/C=C/C')
pentenol
```



To obtain the absolute configuration (i.e., R or S), we can use the `Chem.FindMolChiralCenters()` function which returns the absolute configuration and an index indication which atom has that configuration.

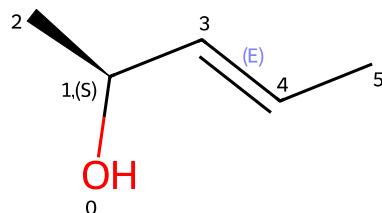
```
Chem.FindMolChiralCenters(pentenol)
```

```
[(1, 'S')]
```

Our pent-3-en-2-ol isomer above has an S stereocenter. Being that pent-3-en-2-ol has only one chiral center, it is not difficult to determine which atom has the stereochemistry, but if there are multiple chiral centers, it can get confusing. To see the atom indices and stereochemistry labels on the molecule, this can be enabled (or disabled using `False`) by the following code.

```
IPythonConsole.drawOptions.addAtomIndices = True
IPythonConsole.drawOptions.addStereoAnnotation = True
```

```
pentenol
```



To obtain the stereochemistry of double bonds, we can iterate through the bonds and obtain the stereochemistry using the `GetStereo()` bond method as shown below. There are three possible outputs listed below.

Table 4 Bond Stereochemical Designations in RDKit

Ouput	Description
STEREONONE	No stereochemistry (often not a double bond)
STEREOE	E stereochemistry
STEREOZ	Z stereochemistry

```
for bond in pentenol.GetBonds():
    print(bond.GetStereo())
```

STEREONONE
STEREONONE
STEREONONE
STEREOE
STEREONONE

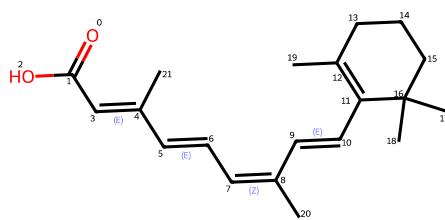
In the above example, there are four bonds with no stereochemistry due to being single bonds, and there is one *E* bond corresponding to the alkene. If there are multiple double bonds, it can be difficult to determine which bond has which stereochemistry. In this case, either use the image like shown above or use additional bond methods (see [section 15.6](#)) to obtain more information about the bonds.

As another example, below we will look at the bonds in 9-cis-retinoic acid where we can see examples of all three possible bond stereochemical assignments.

```
retinoic = Chem.MolFromSmiles(r'0=C(0)\C=C(\C=C\C=C(/C=C/C1=C(/CCCC1(C)C)C)C)')

for bond in retinoic.GetBonds():
    print(bond.GetStereo())
```

retinoic



15.3.2 Counting and Generating Isomers

Another interesting feature of RDKit is the ability to determine the number of stereoisomers possible for a given structure and to generate the different isomers. In both these applications, RDKit treats any explicitly assigned stereocenter as

fixed and will not allow it to be changed. For example, below we will again look at (2S, 3E)-pent-3-en-2-ol. Because the structure already designates this as the (2S, 3E) isomer, the stereochemistry of the chiral center and alkene cannot be changed. As a result, when using the `GetStereoisomerCount()` method from the `EnumerateStereoisomers` module, it returns a 1 indicating that there is only one stereoisomer possible with these constraints.

```
Chem.EnumerateStereoisomers.GetStereoisomerCount(pentenol)
```

```
1
```

In contrast, if we provide the `GetStereoisomerCount()` function hexan-2-ol without any stereochemistry designated (see [above](#)), it returns 2 as the number of stereoisomers. This is because (S)-hexan-2-ol and (R)-hexan-2-ol are both possible isomers.

```
hexanol = Chem.MolFromSmiles('OC(C)CCCC')
Chem.EnumerateStereoisomers.GetStereoisomerCount(hexanol)
```

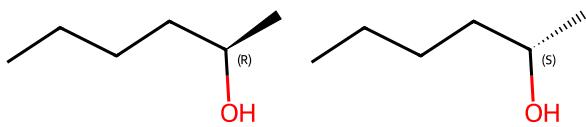
```
2
```

The `EnumerateStereoisomers` module can also generate the different possible isomers, and again, it will only generate isomers by changing stereochemical features that do not already have assigned configurations. If we again look at hexan-2-ol, it generates two Molecule objects which are the two isomers.

```
isomers = list(Chem.EnumerateStereoisomers.EnumerateStereoisomers(hexanol))
isomers
```

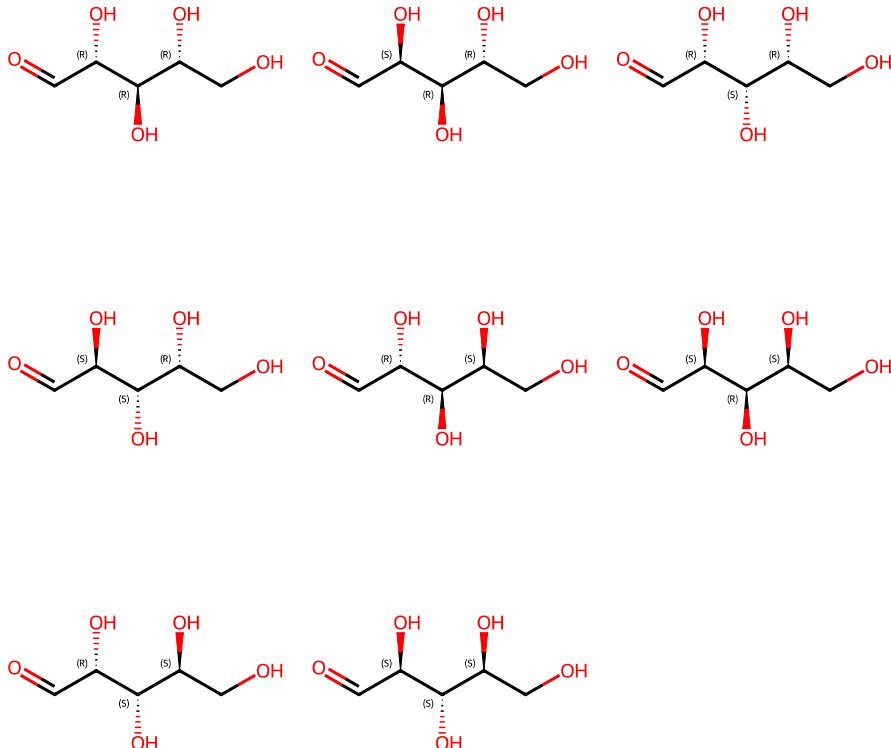
```
[<rdkit.Chem.rdchem.Mol at 0x114f4cd60>,
 <rdkit.Chem.rdchem.Mol at 0x114f4cfe0>]
```

```
IPythonConsole.drawOptions.addAtomIndices = False
IPythonConsole.drawOptions.addStereoAnnotation = True
Chem.Draw.MolsToGridImage(isomers)
```



As a more challenging example, arabinos has three chiral centers allowing for up to eight possible stereoisomers. Because there is a lack of symmetry between the top and bottom (i.e., -CHO and -CH₂OH are different), no meso compound can exist, so it will have the full eight stereoisomers. The real challenge lies in drawing out all eight... unless we make RDKit do the work for us like below.

```
arabinos = Chem.MolFromSmiles('O=CC(0)C(0)C(0)CO')
isomers = list(Chem.EnumerateStereoisomers.EnumerateStereoisomers(arabinos))
Chem.Draw.MolsToGridImage(isomers, useSVG=True)
```



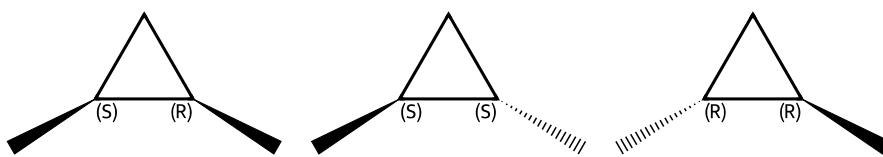
While the examples above mainly focus on stereoisomers from tetrahedral chiral centers, this also works with *E/Z* stereoisomers. One limitation with RDKit is that it currently struggles to recognize non-alkene *cis/trans* stereoisomers when there are stereocenters that are not chiral centers involved such as rings (see [GitHub issue 5597](#)). For example, with 1,2,3-trimethylcyclopropane, it only believes there are eight stereoisomers when in fact there are two.

```
TriCProp = Chem.MolFromSmiles('CC1C(C1C)C ')
Chem.EnumerateStereoisomers.GetStereoisomerCount(TriCProp)
```

8

In contrast, it has no difficulty identifying the three isomers for 1,2-dimethylcyclopropane because both methylated carbons are chiral centers.

```
DiCProp = Chem.MolFromSmiles('CC1CC1C')
CPropisomers = list(Chem.EnumerateStereoisomers.EnumerateStereoisomers(DiCProp))
Chem.Draw.MolsToGridImage(CPropisomers)
```



15.4 Chem.Descriptor Module

RDKit can be used to determine a number of key physical properties of molecules known as *descriptors* using the `Chem.Descriptor` module. These can be useful for generating features for a large number of molecules for machine learning or understanding structural trends in a body of chemical compounds.

15.4.1 Molecular Features

There are numerous descriptor functions available which are callable using `Chem.Descriptors.method()` where `method()` is the name of a descriptor function that accepts an RDKit Molecule object and returns a numerical value. Below are a few examples of descriptor functions, with a more complete list is available on the [RDKit website](#).

Table 5 Examples of Molecular Descriptors

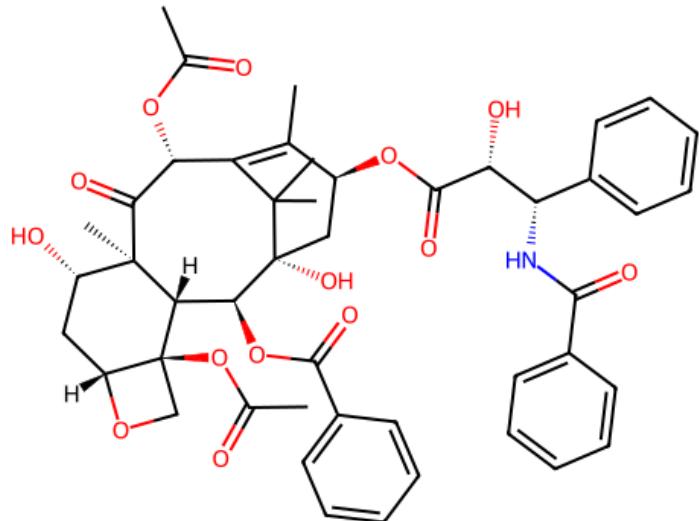
Function	Description
<code>MolWt</code>	Molecular weight, assumes natural isotopic distribution
<code>HeavyAtomCount()</code>	Number of non-hydrogen atoms
<code>NOCount()</code>	Number of N and O atoms
<code>NumAliphaticRings()</code>	Number of aliphatic rings
<code>NumAromaticRings()</code>	Number of aromatic rings
<code>NumSaturatedRings()</code>	Number of saturated rings
<code>NumHAcceptors()</code>	Number of hydrogen bond acceptors
<code>NumHDonors()</code>	Number of hydrogen bond donors
<code>NumRadicalElectrons()</code>	Number of radical electrons
<code>NumValenceElectrons()</code>	Number of valence electrons
<code>NumRotatableBonds()</code>	Number of rotatable bonds
<code>RingCount()</code>	Number of rings

Below we will look at a few of these descriptor functions demonstrated on the compound paclitaxel. Specifically, we will generate the molecular weight, number of rings, number of aromatic rings, number of valence electrons, and number of

rotatable bonds.

```
ptx = Chem.MolFromSmiles('CC1=C2[C@@]( [C@]( [C@H]( [C@@H]3[C@]4([C@H](OC4)C[C@@H]1'\
    '([C@]3(C(=O)[C@H]2OC(=O)C)OC(=O)OC(=O)c5cccc5)'\
    '(C[C@H]1OC(=O)[C@H](O)[C@@H](NC(=O)c6cccc6)c7cccc7)O)(C)C')'

AllChem.Compute2DCoords(ptx) # makes molecule display more clearly
Chem.Draw.MolToImage(ptx, size=(500,500))
```



```
# molecular weight
Chem.Descriptors.MolWt(ptx)
```

853.9180000000003

```
# number of rings
Chem.Descriptors.RingCount(ptx)
```

7

```
# number of aromatic rings
Chem.Descriptors.NumAromaticRings(ptx)
```

3

```
# number of valence electrons
Chem.Descriptors.NumValenceElectrons(ptx)
```

```
# number of rotatable bonds
Chem.Descriptors.NumRotatableBonds(ptx)
```

10

15.4.2 Quantifying Functional Groups

Among the descriptor methods is a long list of functions that look like `fr_group()` where `group` is the name or abbreviation of a chemical functional group. These functions return an integer quantification of that functional group present in the molecule. A table with a few examples is provided below, but there are over 80 of these functions available in RDKit.

Table 6 Examples of Methods to Quantify Functional Groups

Function	Functional Group
<code>fr_Al_OH()</code>	Aliphatic alcohols
<code>fr_aldehyde()</code>	Aldehydes
<code>fr_amide()</code>	Amide
<code>fr_C_C()</code>	Carbonyl oxygens
<code>fr_guanido()</code>	Guanidine
<code>fr_NH0()</code>	Amines with 0 H's (i.e., tertiary)
<code>fr_phenol()</code>	Phenol
<code>fr_phos_ester()</code>	Phosphoric ester
<code>fr_SH()</code>	Thiol

💡 Tip

To see a complete list of functional groups, type `Chem.Descriptors.fr_` into a code cell, press **Tab** for autocomplete, and see the long list of options. If the functional group is not obvious from the name, place the computer cursor inside the function's parentheses and press **Shift + Tab** to see the Docstring description of what functional group it quantifies.

We will again look at paclitaxel to see how many benzene rings, aliphatic alcohols, aromatic carboxyls, and esters are present in the structure.

```
# number of benzene rings  
Chem.Descriptors.fr_benzene(ptx)
```

3

```
# number of aliphatic alcohols  
Chem.Descriptors.fr_Al_OH(ptx)
```

3

```
# number of aromatic carboxyls  
Chem.Descriptors.fr_Ar_COO(ptx)
```

0

```
# number of esters  
Chem.Descriptors.fr_estер(ptx)
```

4

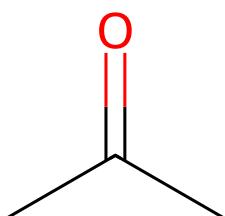
15.5 Searching Molecules for Structural Patterns

Molecules can be searched for key structural features using the `HasSubstructMatch()` method which returns `True` or `False` depending if a structural pattern exists in a molecule or not. This function requires two RDKit Molecule objects - one Molecule object (`molecule`) is checked for the presence of the other Molecule object structure (`substructure`) as shown below. There are optional keyword parameters such as `useChirality=` which allows for chirality to be factored into whether there is a match or not. The default setting is `useChirality=False`.

```
molecule.HasSubstructMatch(substructure, useChirality=False)
```

As an example, we will look for the presence of a carbonyl (i.e., C=O bond) in acetone and pent-3-en-2-ol below, so the substructure that we will search for is a `C=O`.

```
acetone = Chem.MolFromSmiles('CC(=O)C')  
acetone
```



```
substructure = Chem.MolFromSmiles('C=O')
acetone.HasSubstructMatch(substructure)
```

True

```
pentenol.HasSubstructMatch(substructure)
```

False

Not very surprisingly, the `HasSubstructMatch()` function returns `True` for acetone and `False` for the alcohol because the latter has a single CO bond, not a double. If we change our substructure to `C0`, we are now searching for a carbon-oxygen single bond (see Table 7), so acetone returns `False` while pent-3-en-2-ol returns `True`.

Table 7 SMILES Bond Order Notation

SMILES Bond	Bond Type
- (or nothing)	Single
=	Double
#	Triple
:	Aromatic

```
substructure = Chem.MolFromSmiles('CO')
substructure
```



```
acetone.HasSubstructMatch(substructure)
```

False

```
pentenol.HasSubstructMatch(substructure)
```

True

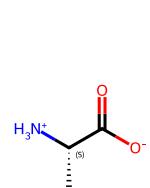
For a more interesting set of examples, we can search our collection of 20 common amino acids (see [section 15.2.2](#)) for key substructures. We will start by using glycine, the simplest of the common amino acids, as the substructure which should return all 20 amino acids. As an extra step below, we will also orient all the amino acids in the same way with

respect to the substructure. That is, the substructural element that we are searching for in each amino acid will be oriented the same way for all 20 amino acids.

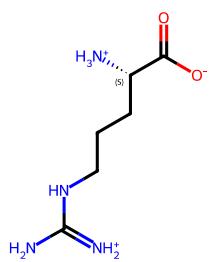
```
# searches for substructure
substructure= Chem.MolFromSmiles('C(C(=O)[O-])[NH3+]')
matching_amino_acids = [AA for AA in AminoAcids if AA.HasSubstructMatch(substructure)]

# orients common substructures the same way
AllChem.Compute2DCoords(substructure)
for amino_acid in matching_amino_acids:
    _ = AllChem.GenerateDepictionMatching2DStructure(amino_acid, substructure)

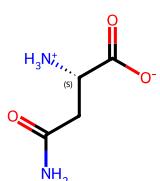
# generates grid of matching molecules
Chem.Draw.MolsToGridImage(matching_amino_acids,
                           molsPerRow=4,
                           subImgSize=(200,200),
                           legends=list(df['name']))
```



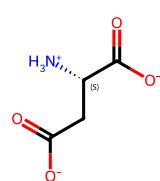
alanine



arginine



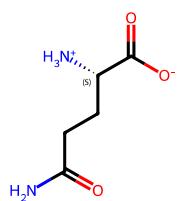
asparagine



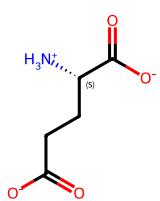
aspartate



cysteine



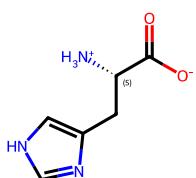
glutamine



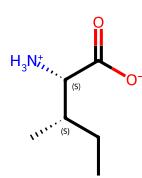
glutamate



glycine



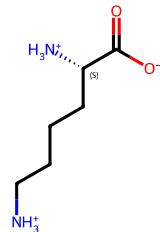
histidine



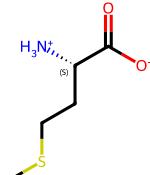
isoleucine



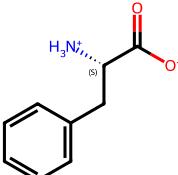
leucine



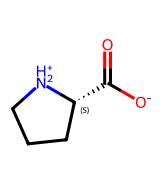
lysine



methionine



phenylalanine



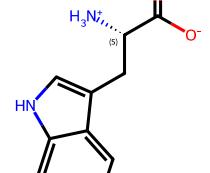
proline



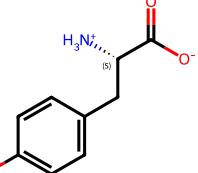
serine



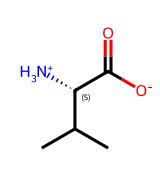
threonine



tryptophan



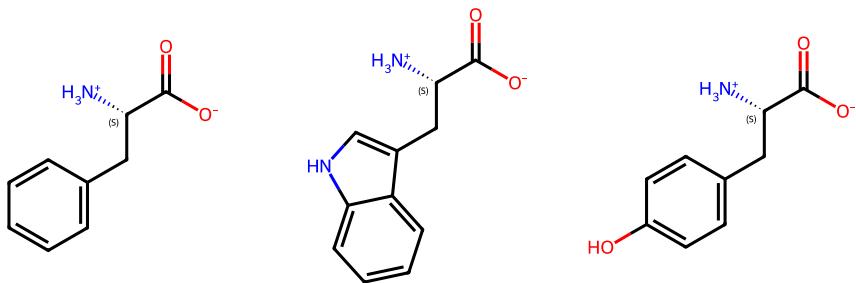
tyrosine



valine

Indeed, it did return all 20 amino acids, and notice how the core structure of all amino acids are oriented the same direction. Now let us try something a little more interesting by search for all amino acids with a benzene ring in them. The substructural bonding pattern in this case is benzene itself, and the three aromatic amino acids are returned.

```
substructure = Chem.MolFromSmiles('c1ccccc1')
AA_with_pattern = [AA for AA in AminoAcids if AA.HasSubstructMatch(substructure)]
Chem.Draw.MolsToGridImage(AA_with_pattern)
```



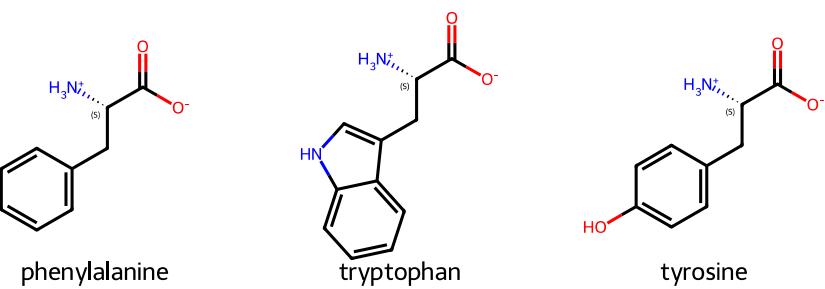
It might be nice to still have the name labels for our three matches, so the above search is repeated but instead on a zip object comprised of the names of the amino acids and the Molecule objects.

```
AA_zipped = list(zip(df['name'], AminoAcids))

substructure = Chem.MolFromSmiles('c1ccccc1')
with_pattern = [AA for AA in AA_zipped if AA[1].HasSubstructMatch(substructure)]

name = [AA[0] for AA in with_pattern]
mol_obj = [AA[1] for AA in with_pattern]

Chem.Draw.MolsToGridImage(mol_obj, legends=name)
```



15.6 Atoms and Bonds

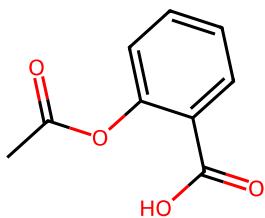
RDKit allows access to information on specific atoms and bonds through the `GetAtoms()` and `GetBonds()` methods, respectively. These functions return a sequence type of object that can be iterated through using a `for` loop to access individual atoms or bonds. Using the following methods, the user can access or even modify various pieces of information about the atoms or bonds. Below Table 9 and Table 10 contain some key functions for working with atoms and bonds.

Table 9 Select Atom Methods

Function	Description
<code>GetDegree()</code>	Returns number of atoms bonded directly to it, includes hydrogens only if they are explicitly defined
<code>GetAtomicNum()</code>	Returns atomic number
<code>GetChiralTag()</code>	Determines if the atom is a chiral center and CW or CCW designation
<code>GetFormalCharge()</code>	Returns formal charge of atom
<code>GetHybridization()</code>	Returns hybridization of atom
<code>GetIsAromatic()</code>	Returns bool as to whether atom is aromatic
<code>GetIsotope()</code>	Returns isotope number if designated, otherwise returns <code>0</code>
<code>GetNeighbors()</code>	Returns tuple of directly bonded atoms
<code>GetSymbol()</code>	Returns atomic symbols as a string
<code>GetTotalNumHs()</code>	Returns number of hydrogens bonded to the atom
<code>IsInRing()</code>	Returns bool designating if the atom is in a ring
<code>SetAtomicNum()</code>	Sets the atomic number to user defined value
<code>SetFormalCharge()</code>	Sets formal charge to user defined value
<code>SetIsotope()</code>	Sets isotope to user defined integer value

As an example, let's look at the atoms in aspirin.

aspirin



If we generate a list populated with the degrees of atoms (i.e., number of other atoms bonded directly to it), you may notice that there are no 4 values even though the methyl (i.e., $-\text{CH}_3$) carbon should have four atoms attached to it. This is because the hydrogens are not explicitly designated in the structure (i.e., they are implicit), so they are not counted.

```
[atom.GetDegree() for atom in aspirin.GetAtoms()]
```

```
[1, 3, 1, 2, 3, 2, 2, 2, 2, 3, 3, 1, 1]
```

We can count the number of implicit hydrogens using the `GetNumImplicitHs()` method, and the third value is a `3` making it the methyl carbon.

```
[atom.GetNumImplicitHs() for atom in aspirin.GetAtoms()]
```

```
[0, 0, 3, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1]
```

We can also use these atom methods to change values and attributes of various atoms. For example, we can set the isotopes of the carbonyl carbons (i.e., C=O) to ^{13}C . This is accomplished with the following code that iterates through all the atoms and finds the carbonyl carbons by testing for atoms that have an atomic number of 6, are not aromatic, and have no hydrogens and then setting the isotope value to 13. The molecular weight is calculated before and after the isotopes are changed for comparison.

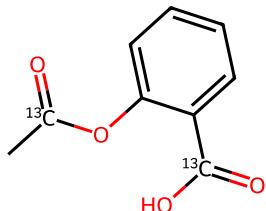
```
Chem.Descriptors.MolWt(aspirin)
```

```
180.1589999999996
```

```
for atom in aspirin.GetAtoms():
    if atom.GetAtomicNum() == 6 and \
       not atom.GetIsAromatic() and \
       atom.GetTotalNumHs() == 0:
        atom.SetIsotope(13)
```

```
print(Chem.Descriptors.MolWt(aspirin))
aspirin
```

```
182.14370968
```



The molar mass has increased due to two of the carbon atoms being isotopically labeled, and we can see in the image which of the two carbons were isotopically labeled. It is worth noting that the molecular weight before isotopically labeling assumes a natural distribution of isotopes which for carbon is 98.9% ^{12}C and 1.1% ^{13}C . In the isotopically labeled structure, the two carbonyl carbons are 100% ^{13}C .

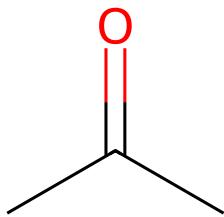
Using bond methods, we can perform analogous types of operations except that bonds have different attributes than atoms. A table of selected bond methods is provided below.

Table 10 Select Bond Methods

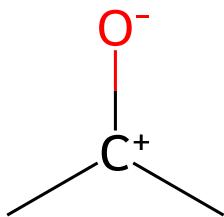
Function	Description
<code>GetBeginAtom()</code>	Returns first atom in bond
<code>GetEndAtom()</code>	Returns second atom in bond
<code>GetBondType()</code>	Returns type of bond (e.g., SINGLE, DOUBLE, AROMATIC)
<code>GetIsAromatic()</code>	Returns bool as to whether bond is aromatic
<code>GetIsConjugated()</code>	Returns bool as to whether bond is conjugated
<code>IsInRing()</code>	Returns bool as to whether bond is in ring
<code>SetBondType()</code>	Sets bond type
<code>SetIsAromatic()</code>	Sets bool designating if a bond is aromatic

As a demonstration, we will examine the bonds in the structure of acetone and change the carbonyl double bond to a single bond. This is done by searching for a double bond, setting it to a single bond, and then changing the formal charges of the atoms attached to that bond.

acetone



```
for bond in acetone.GetBonds():
    if bond.GetBondType() == Chem.BondType.DOUBLE:
        bond.SetBondType(Chem.BondType.SINGLE)
        end = bond.GetEndAtom().SetFormalCharge(-1)
        begin = bond.GetBeginAtom().SetFormalCharge(+1)
acetone
```



Further Reading

1. RDKit: Open-Source Cheminformatics Software. <https://www.rdkit.org/> (free resource)
2. The RDKit Book (collection of examples). https://www.rdkit.org/docs/RDKit_Book.html (free resource)

Exercises

Complete the following exercises in a Jupyter notebook using RDKit. You are encouraged to also to use data libraries such as NumPy or pandas to support your solutions. Any data file(s) referred to in the problems can be found in the [data](#) folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

1. Load the structure for **morphine** into RDKit using either a SMILES or InChI representation. You will need to either generate one of these representations using chemical drawing software or find one online from a free resource.
 - a) Visualize the structure of morphine and save it as an SVG image file.
 - b) Use RDKit to determine the number of chiral centers in the structure. Your code should output an integer value, not just a list of chiral centers.
 - c) Use RDKit to determine the number of hydrogen bond acceptors in the structure.
 - d) Use RDKit to determine the number of rings in the structure.
2. Load the **amino_acid_SMILES.txt** file and use RDKit for the following.
 - a) Determine the absolute configuration (i.e., R vs. S) of the α -carbon for all the chiral amino acids. Most are the same, but one is an exception. Which is it?
 - b) How many amino acids have two chiral centers?
3. Load the **organic_molecules.txt** dataset containing SMILES representations of a range of organic molecules.
 - a) Using descriptors, generate a list containing the SMILES representations of only primary and secondary aliphatic alcohols.
 - b) Using pattern matching, generate an image containing only primary alcohols from the file. To help you along, here is the SMARTS representation of a primary alcohol for the pattern matching. `Chem.MolFromSmarts(' [CH2] [OH] ')`
 - c) Calculate the percentage of heavy-element (i.e., not with hydrogens) bonds that are C-O bonds.
 - d) Calculate the percentage of carbon atoms in a ring.
4. Use RDKit to generate an image showing all isomers of 1,2-dimethylcyclohexane. You will need to look up the SMILES or other representation first.

Chapter 16: Bioinformatics with Biopython and Nglview

Contents

- 16.1 Working with Sequences
- 16.2 Structural Information
- 16.3 Visualization of Molecules
- Further Reading

Bioinformatics is the field of working with biological or biochemical data using computing resources, and while the underlying techniques for working with biological data is fundamentally the same as what has been seen so far, this field is large and significant enough to warrant its own chapter. More importantly, bioinformatics contains a multitude of specialized file formats making this a significant hurdle in working with these data. The good news is that biological/biochemical file formats are usually text files like seen in the previous chapters, and there are Python libraries available to facilitate the parsing and working with these file formats and data. This chapter focuses on a few common file formats, parsing them with both our own Python code and using the Biopython library to perform the heavy lifting.

The Biopython library is among the well-known bioinformatics Python libraries handy for working with biological and biochemical data. It will need to be installed in Jupyter or Google Colab because it is not a default library. As of this writing, Biopython can be installed [using pip](#) by `pip install biopython`, and a Conda option is also available. Once installed, it is imported as `Bio`. This chapter assumes the following imports.

```
import Bio
from Bio import PDB, SeqIO, SeqUtils, Align

# Turns off warning (about data in PDB files)
import warnings
from Bio import BiopythonWarning
warnings.simplefilter('ignore', BiopythonWarning)

import matplotlib.pyplot as plt
import seaborn as sns
import os
```

16.1 Working with Sequences

Among the most fundamental data in bioinformatics are sequences which simply provide the order of monomers in a sequence of nucleotides or amino acids. For protein sequences, these monomers are mainly the 20 common amino acids with other less frequent amino acids and other species possible, and for nucleic acid sequences, the monomers are nucleotides. In this section, we will work with sequences inside Biopython to performing various operations such as sequence alignment and translating mRNA sequences into peptide sequences.

Inside Biopython, sequences are often stored as a Sequence object which looks like a string inside a list wrapped in

`Seq()` such as below. This object contains many of the same methods as a Python string plus some extra, so you can still iterate through Sequence objects with a `for` loop along with index, slice, reverse them, and alter the case like a string.

```
Seq('GCCGGCAGTCACACGCACAGGC')
```

16.1.1 Reading FASTA Files with Biopython

There are numerous file formats that can store sequence data, but for the examples in this section, we will focus on the FASTA file format which only holds the sequence data and a small amount of metadata (i.e., data about the data). FASTA files are text files that look like the following when opened in a text editor. A FASTA file can contain a single or multiple sequence entries with the first line of each entry beginning with a `>`. The rest of this line includes helpful information about the sequence such as the organism and what specific molecule it relates to. The rest of the text block is sequence information. There is no strict rule on how many letters can be contained in each line, but 70 is a common length.

```
>\>7AIZ_1|Chains A, D|Nitrogenase vanadium-iron protein alpha chain|Azotobacter vinelandii (354)
MPMVILLECDKDIPERQKHIYLKAPNEDTREFLPIANAATIPGTLSERGCAFCGAKLVIGGVLKDTIQMIH
MPMVILLECDKDIPERQKHIYLKAPNEDTREFLPIANAATIPGTLSERGCAFCGAKLVIGGVLKDTIQMIH
GPLGCAYDTWHTKRYPTDNGHFMKYVWSTDMKESHVVFGGEKRLEKSMHEAFDEMPDIKRMIVYTTCP
ALIGDDIKAVAKVMKDRPDVDFVTECPFGSGVSQSKGHVHLNVNIGWINEKVETMEKEITSEYTMNFIGD
FNIQGDTQLLQTYWDRLGIQVVAHFTGNGTYYDDLRCMHQAQLNVNCARSSGYIANELKKRYGIPRLID
SWGFNYMAEGIRKICAFFGIEEKGEELIAEYAKWPKLDWYKERLQGKKMAIWTGGPRLWHWTKSVEDD
LGVQVVAMSSKFGHEDEFKVIARGKEGTYYIDDNELEFFEIIDLVKPDVIFTGPRVGEVKKLHIPYV
NGHGYHNGPYMGFEGFVNLRDMYNAVHNPLRHLAAVDIRDKSQTPVIVRGAA
```

In the following example, we will load a FASTA file containing Norway rat RNA using the `SeqIO.read()` and `SeqIO.parse()` functions which are similar except that `SeqIO.read()` can only load FASTA files with a single entry while `SeqIO.parse()` can open files with single or multiple entries. Both will be demonstrated below, and both require two positional arguments - the file or file path as a string and the file type as a string.

```
SeqIO.read('file_name', 'file_type')
SeqIO.parse('file_name', 'file_type')
```

```
rat = SeqIO.read('data/rcsb_pdb_430D.fasta', 'fasta')
rat
```

```
SeqRecord(seq=Seq('GGGUGUCAGUACGAGAGGAACCGCACCC'), id='430D_1|Chain', name='430D_1|Chain', description='Norway rat 430D RNA')
```

The `SeqIO.read()` function returns a Sequence Record object which has a few attributes shown in the table below. The most important attribute is the sequence itself which is stored as a Sequence object.

Table 1 Sequence Record Attributes

Attribute	Description
<code>id</code>	Returns the sequence ID from the file's first line
<code>description</code>	Returns a description from the file's first line
<code>seq</code>	Returns the sequence as a Sequence object
<code>name</code>	Returns the sequence name from the file's first line (may be same as ID)

```
rat.seq
```

```
Seq('GGGUGUCAGUACGAGAGGAACCGCACCC')
```

In the event we have a file containing multiple entries, the `SeqIO.parse()` function is required. The function works the same way as the `SeqIO.read()` version except that a one-time use iterator object is returned that contains each entry from the FASTA file. To extract this information, we need to iterate over it using a `for` loop. Data from each entry can be accessed using the same methods as the `SeqIO.read()` function. This is demonstrated below using a FASTA file for a protein structure of [Norwegian rat hemoglobin](#).

```
fasta_data = SeqIO.parse('data/rcsb_pdb_3DHT.fasta', 'fasta')
seq_list = []
for entry in fasta_data:
    seq_list.append(entry.seq)
seq_list
```

```
[Seq('VLSADDKTNIKNCWKGKIGGHGGEYGEEARQMFQAFPTTKTYFSHIDVSPGSAQ...KYR'),
 Seq('VHLTDAEKAAVNGLWGKVNPDDVGGEALGRLLVVYPWTQRYFDSFGDLSSASAI...KYH')]
```

Because the iterator is a one-time use object, attempting to iterate over it again, like below, fails to return any data, so be sure to attach any data to a variable or append it to a list.

```
for entry in fasta_data:
    print(entry.seq)
```

16.1.2 GC Content of Nucleotide Sequence

One piece of information we can extract from a nucleotide sequence is the GC content. In DNA for example, there are two complementary strands hydrogen bonded together which contain the base pairs adenine(A)/thymine(T) and guanine(G)/cytosine(C), so the number of adenines equals the number of thymines and the number of guanines equals the number of cytosines. However, the number of A/T pairs does not necessarily equal the number of G/C pairs. The *GC content* of DNA is the fraction of total bases that are G/C which can be calculated using the number (*n*) of G and C bases divided by the total number of all bases in the sequence.

$$GC \text{ content} = \frac{GC \text{ bases}}{sequence \text{ length}} = \frac{n_G + n_C}{n_G + n_C + n_A + n_T}$$

Below, we will calculate the GC content of a DNA sequence in a FASTA file using Biopython's `gc_fraction(seq)` function which accepts a Biopython sequence and returns the GC content in fraction form.

```
DNA = SeqIO.parse('data/DNA_sequence_drago.fasta', 'fasta')
rat_seq = [x.seq for x in DNA]

SeqUtils.gc_fraction(*rat_seq)
```

0.5296912114014252

Sometimes there are characters in a DNA sequence other than A, T, C, and G due to ambiguities among other reasons. An N means that the base is unidentifiable while S means it is either C or G and W means it is either A or T. The `gc_fraction()` function provides an `ambiguous=` parameter that can be used to decide how to deal with ambiguous characters. Below are the three string options for the `ambiguous=` parameter where `remove` is the default setting.

Table 2 Settings for `gc_fraction()` `ambiguous=` Parameter

Options	Description
<code>'remove'</code>	Default setting; only uses 'ATCGSW' characters and ignores the rest
<code>'ignore'</code>	Uses 'GCS' characters for GC count and rest of characters for sequence length
<code>'weighted'</code>	Applies weights to various characters effectively forming a weighted average

Our sequence contains some `N` characters, so if we set it to `ignore`, the GC content value is expected to decrease due to a larger denominator in the equation above versus the default `remove` option.

```
SeqUtils.gc_fraction(*rat_seq, ambiguous='ignore')
```

0.5247058823529411

16.1.3 Nucleic Acids - Transcription, Translation, and Replication

In protein synthesis, the coding (or informational) strand of DNA is transcribed to mRNA which is then translated to a protein sequence. DNA can also replicate by unwinding and using additional complementary nucleotides to bond the coding and template strands. Biopython makes performing digital analogue of these operations relatively simple using the following functions.

Table 3 Methods for Performing Transcription, Translation, and Replication

Function	Description
<code>transcribe()</code>	Transcribes coding DNA strand to mRNA (maintains 5' → 3' direction)
<code>translate()</code>	Translates mRNA sequence (5' → 3') to a peptide sequence (N → C)
<code>complement()</code>	Converts 5' → 3' nucleotide sequence to the 3' → 5' complementary sequence
<code>reverse_complement()</code>	Converts 5' → 3' DNA strand to 5' → 3' complementary sequence
<code>reverse_complement_rna()</code>	Converts 5' → 3' RNA strand to 5' → 3' complementary sequence
<code>complement_rna()</code>	Converts 5' → 3' RNA strand to 3' → 5' complementary sequence
<code>replace(old, new)</code>	Replaces <code>old</code> items in sequence with <code>new</code> (can also be used to replace spaces)

While some functions in Biopython accept strings or Sequence objects, the functions above work exclusively with Sequence objects. The good news is that if you have a string, it is easy to convert to a Sequence object using the `Seq()` function like below.

```
coding_DNA = Bio.Seq.Seq('GGAGAGTGACGCCGGCAGTCACACGACAGGCTGCAGCAACGAAAGAT')
coding_DNA
```

```
Seq('GGAGAGTGACGCCGGCAGTCACACGACAGGCTGCAGCAACGAAAGAT')
```

We can perform transcription using the `transcribe()` method which operates on a DNA strand and assumes that the DNA strand is the coding (or informational) strand. It also assumes that the sequence is in the 5' → 3' direction and returns the mRNA sequence also in the 5' → 3' direction.

```
mRNA = coding_DNA.transcribe()
mRNA
```

```
Seq('GGAGAGUGACGCCGGCAGUCACACGACAGGCUGCAGCAACGAAAGAU')
```

If you find yourself with the template strand, this can be converted to the coding strand using the `reverse_complement()` function like below which takes a DNA strand in the 5' → 3' direction and returns the complementary strand also in the 5' → 3' direction. This coding strand can then be transcribed to mRNA.

```
template_DNA = Bio.Seq.Seq('ATCTTCGTTGCTGCAGCCTGTGCGTGTGACTGCCGGCGTCACTCTCC')
coding_DNA = template_DNA.reverse_complement()
coding_DNA.transcribe()
```

```
Seq('GGAGAGUGACGCCGGCAGUCACACGACAGGCUGCAGCAACGAAAGAU')
```

Once we have our mRNA sequence, we can translate it to a peptide sequence using the `translate()` method which is performed using the standard codon table.

```
mRNA.translate()
```

```
Seq('GE*RRQSHAQAAATKD')
```

By default, this function will translate the entire mRNA sequence disregarding any stop codons. To heed the stop codons, set the `to_stop` parameter to `True`.

```
mRNA.translate(to_stop=True)
```

```
Seq('GE')
```

16.1.4 Sequence Alignment

Biopython can perform both global and local pairwise alignments of sequences including nucleic acids and proteins. The difference between these types of alignments is that *global pairwise alignment* attempts to align the entirety of two sequences of at least somewhat similar length while *local pairwise alignment* attempts to align subsequences of the two sequences. Local alignment essentially attempts to find common regions between multiple sequences. The alignment processes generates a score based on user-defined rules and attempts to maximize this score to generate the "best" alignment. For example, aligned bases in two DNA sequences might be awarded a +1 while misaligned bases are penalized a -1.

Pairwise sequence alignment in Biopython starts with creating a `PairwiseAligner` object which requires the type of alignment (`'global'` or `'local'`). Optionally, you can set the scoring parameters which dictate how a match, mismatch, starting a gap, extending a gap, and ending a gap affect the score. By default, +1 is awarded for every match and mismatches and gaps are all 0. Below, the `PairwiseAligner` is set to `'global'` and scoring parameters are adjusted as shown.

Once we have created the `PairwiseAligner` object, we can use the `align()` method to return the optimal alignment between the two sequences based on the scoring parameters. It is important to note that there can be multiple optimal sequence alignments (i.e., tied for best score) based on our scoring parameters, so the `align()` method can return multiple alignments.

Below, the aligned sequences are stored in the variable `alignment`. When we check the length of this object, we find it contains 15 alignments which can be viewed by indexing or iteration.

```
seq1 = 'GGAGAGTGACGCCGGCAGTCACACGCAACGGCTGCAGCAACGAAAAGTT'
seq2 = 'GGAGAGTGACGCCGGCAGTCACACGCTCAGGCTGCAGCAACGAAAAAGTT'

alignments = aligner.align(seq1, seq2)
len(alignments)
```

```
print(alignment[0])
```

target	0 GGAGAGTGACGCC-GGCAGTCACACGCACAGGCTGCAGCAACG-AAAAGTT- 49
query	0 - . - - 52
	0 GGAGAGTGACGCCGGGAGTCACACGCTCAGGCTGCAGCAACGAAAAAGTTA 52

The score from the optimal alignments can be viewed using the `score` method. Keep in mind the score is affected by not only the quality of the alignment based on the alignment parameters but also sequence length, so it is not necessarily useful for comparing alignments between different pairs of sequences.

```
score = aligner.score(seq1, seq2)  
score
```

```
44.0
```

16.2 Structural Information

In this section, we will work with two common file formats for storing biochemical data: PDB and mmCIF. Both of these file formats are text files, so information can always be extracted using pure Python code you wrote yourself. However, there are also preexisting tools that can make this process substantially easier such as Biopython or scikit-bio (see [Further Reading](#)). Below you will see demonstrations of both pure Python and Biopython approaches with an emphasis on using Biopython.

Protein Database (PDB) and Macromolecular Crystallographic Information File (mmCIF) files are designed to hold protein sequence and structural information while the [FASTA file format](#) only holds sequence data for proteins and nucleic acids. The FASTA file format is simpler than the PDB and mmCIF file formats, but there is significant amounts of structural data, addressed below, contained in the latter formats that go beyond the sequence.

16.2.1 Reading PDB Files with Python

The PDB file format is a classic file format for holding protein sequence and structural information including the information listed below. While the PDB is being slowly replaced by the mmCIF (see [section 16.2.2](#)), the PDB file format is still quite common and worth looking at.

- Amino acid sequence of each strand
- Location and identity of non-amino acid species
- xyz coordinates of atoms in the crystal structure including trapped solvents
- Connectivity information
- Metadata about the protein (e.g., source organism, resolution, etc.)
- Secondary structural information

First, we need a PDB file of a protein structure which can be downloaded for free from the [RCSB Protein Data Bank](#). The **Download Files** menu on the top right provides a number of file format options including **PDB Format**. In the example below, we will look at the [Vanadium nitrogenase VFe protein structure](#) in the **7aiz.pdb** file.

The PDB file is organized where each line holds a different type of information, and a label in all caps on the far left of each line indicates what type of information is stored in that line. Below are some key labels (i.e., record type), but this is far from a comprehensive list. Data within a line is identifiable based on the character *position* in a line. This is in contrast to many other file types where data in a single line are distinguished by separators such as commas or spaces. For more information on the PDB file format, see the [Protein Data Bank website](#). If you are using JupyterLab, you can double click the PDB file to open it and view the contents.

Table 4 Selected PDB File Record Types

Record Type	Description
HEADER	Name of protein and date
TITLE	Name of molecule
COMPND	Information about the compound
SOURCE	Information about the source of the protein (e.g., source organism)
SEQRES	Amino acid sequence and strand identity
HET, HETNAM	Information about non-amino acids in protein structure
HELIX	Information about helicies including type, start and end amino acids, etc.
SHEET	Information about sheets including start and end amino acids and sense
ATOM	Information about atoms in structure inclidng xyz coordinates, identity, amino acid, etc.
SSBOND	Identifies cysteins involved in each disulfide bond

Before we rely on Biopython to extract information from data files, we will use pure Python. As a short demonstration, the code below opens the PDB file and appends each line to a list called `data`. We can examine a few of the lines using slicing to see information about the structure of the protein. The lines shown below provide information about the helicies and sheets in the protein structure.

```
file = 'data/7aiz.pdb'

data = []
with open(file, 'r') as f:
    for line in f:
        data.append(line)
```

```
data[1190:1200]
```

```

['HELIX 109 AM1 ARG F 24 THR F 44 1 21 \n',
 'HELIX 110 AM2 THR F 52 PHE F 73 1 22 \n',
 'HELIX 111 AM3 PRO F 74 GLN F 78 5 5 \n',
 'HELIX 112 AM4 ASN F 80 ILE F 100 1 21 \n',
 'SHEET 1 AA1 6 ILE A 19 LEU A 21 0 \n',
 'SHEET 2 AA1 6 TYR A 380 ASP A 383 -1 0 TYR A 381 N TYR A 20 \n',
 'SHEET 3 AA1 6 GLN A 354 SER A 360 1 N MET A 358 0 TYR A 380 \n',
 'SHEET 4 AA1 6 LYS A 330 THR A 335 1 N MET A 331 0 GLN A 354 \n',
 'SHEET 5 AA1 6 VAL A 401 THR A 404 1 0 PHE A 403 N ALA A 332 \n',
 'SHEET 6 AA1 6 TYR A 419 ASN A 421 1 0 VAL A 420 N ILE A 402 \n']

```

As an exercise, we can extract information about the β -sheets in the protein. Specifically, we will look at the relative directions (sense) of adjacent strands which can run in the same direction (parallel) or in the opposite directions (antiparallel) as the previous strand. This is indicated by the integer in positions 39-40 of a SHEET line and can be either 0 for the first strand of a β -sheet, 1 for a strand parallel with the previous strand, and -1 for a strand antiparallel with the previous strand. The function below extracts this information by opening the PDB `file`, moving through each line of the file, and if the line begins with `SHEET`, it appends the relative direction to a list and returns the populated list.

```

def get_sheet_direction(file):
    '''Accepts a PDB files path (string) and returns a list
    of values indicating if a strand starts a beta sheet (0),
    strand is parallel to the previous strand (1), or is
    antiparallel to the previous strand (-1).

    >>> ('1abc.pdb') -> [0, 1, 1, 1, -1]
    ...

    structure_list = []

    with open(file, 'r') as f:
        for line in f:
            if line.startswith('SHEET'):
                sense = int(line[38:40].strip())
                structure_list.append(sense)

    return structure_list

```

```

sheet_sense = get_sheet_direction('data/7aiz.pdb')
print(sheet_sense)

```

```
[0, -1, 1, 1, 1, 1, 0, 1, 1, -1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, -1, 1, 1,
```

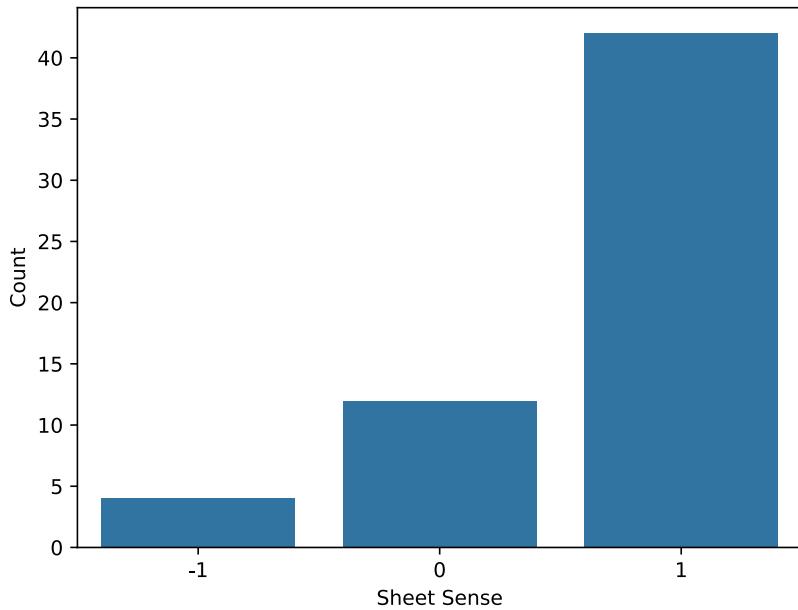
```

sns.countplot(x=sheet_sense, order=[-1, 0, 1])

plt.xlabel('Sheet Sense')
plt.ylabel('Count')

```

```
Text(0, 0.5, 'Count')
```



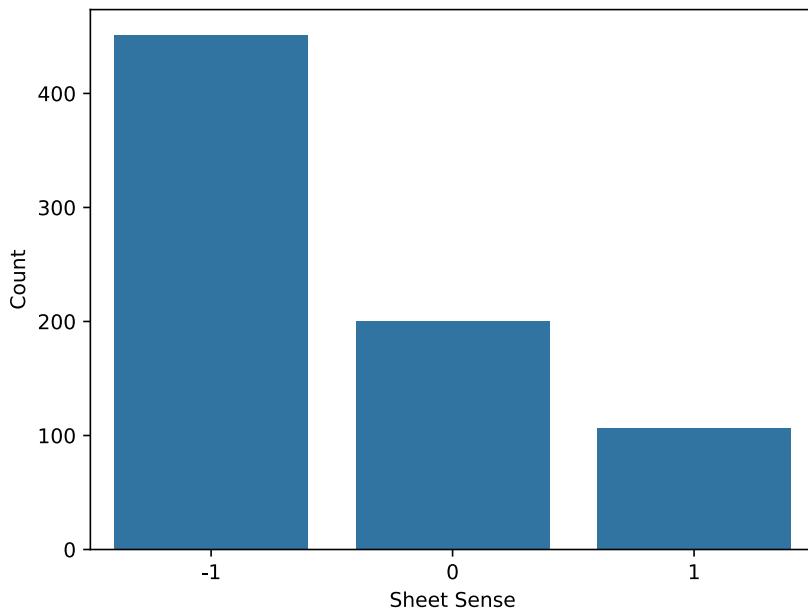
According to the graph above, parallel β -sheet strands are significantly more prevalent in this protein structure than antiparallel strands. This might be different for other proteins, so we will expand this analysis to a folder full of protein structures.

```
current_directory = os.getcwd()
data_folder = os.path.join(current_directory, 'data/proteins')

sheet_sense = []
for file in os.listdir('data/proteins'):
    if file.endswith('pdb'):
        sheet_sense.extend(get_sheet_direction(os.path.join(data_folder, file)))
```

```
sns.countplot(x=sheet_sense, order=[-1, 0, 1])
plt.xlabel('Sheet Sense')
plt.ylabel('Count')
```

```
Text(0, 0.5, 'Count')
```



The trend over a larger sample of proteins is that antiparallel is significantly more common than parallel, so it seems that the 7aiz protein structure is an exception to the typical trend. However, this is only little over a dozen structures, so it would require a much larger dataset to be certain of this trend.

16.2.2 Reading Structural Files with Biopython

Next, we will use the Biopython library to read data from PDB and other structural files. One of the appeals of using Biopython is that the user does not need to understand the structure of the file format; Biopython parses the files allowing you to focus on higher-level concerns.

First, we need to import the `PDB` module of the Biopython library with the `import Bio.PDB` command if you have not done so already (see [start of this chapter](#)). Biopython, like SciPy, requires that individual modules be imported one at a time instead of the entire library (i.e., `import Bio` is not enough). You are welcome to import functions individually (e.g., `from Bio.PDB import PDBParser()`), but herein we will only import the module using `import Bio.PDB` so that the code more clearly shows the source of every function. The `PDB` module provides tools for dealing with the 3D structural data of macromolecules such as proteins and DNA. To parse a PDB file, we first create a parser object using the `Bio.PDB.PDBParser()` function.

```
parser = PDB.PDBParser()
```

We will then use the `get_structure()` function to read in data from a file. This function requires two positional arguments - a name for the structure and the name of the file. Both arguments are strings, and the structure name can be anything you like.

```
structure = parser.get_structure('7aiz', 'data/7aiz.pdb')
```

Despite the name, the `PDB` module contains tools for dealing with the other file formats such as mmCIF, PQR, and MMTF. The mmCIF file format is the successor to the PDB format making it an increasingly common file format. The good news is that parsing different structural files is almost identical as Biopython deals with most of the file format details behind the scenes. The only difference in dealing with mmCIF files versus PDB in Biopython is that we use the

`PDB.MMCIFParser()` function to read the mmCIF file instead of `PDB.PDBParser()`, so mmCIF code would look like the following.

```
parser = PDB.MMCIFParser()
structure = parser.get_structure('7aiz', 'data/7aiz.cif')
```

A list of various file parsers is provided in Table 5.

Table 5 Selected File Parser Functions from `Bio.PDB`

File Type	Parser Function
PDB	<code>PDB.PDBParser()</code>
mmCIF	<code>PDB.MMCIFParser()</code>
PQR	<code>PDB.PDBParser(is_pqr=True)</code>
MMTF	<code>Bio.PDB.mmtf.MMTFParser()</code>

16.2.3 Writing Files with Biopython

Biopython is also capable of writing structures to new PDB or mmCIF files, but by default it will not include much of the metadata (e.g., resolution, name of structure, authors, etc.) and information about secondary structures in the new files.

The general methodology is to first create a writing object using either `PDB.PDBIO()` or `PDB.MMCIFIO()` for creating a new PDB or mmCIF file, respectively. Next, use the `set_structure()` method on the writing object to load the data from an individual structure. Finally, write the file using the `save()` function and providing it with the name of the new file as a string.

```
# write a new PDB
io = PDB.PDBIO()
io.set_structure(structure[0])
io.save('new_protein.pdb')
```

```
# write a new mmCIF
io = PDB.MMCIFIO()
io.set_structure(structure[0])
io.save('new_protein.cif')
```

16.2.4 Accessing Strands, Residues, and Atoms

The structural data extracted from the PDB or mmCIF by Biopython is organized in the hierachal order of structure → model → chain → residue → atom. This means that models are contained within the structure, chains are contained within each model, residues are contained within each chain, and atoms are contained within each residue. The structure is the protein, the model is a particular 3D model of the protein, the chain is a single peptide chain in the protein, the residue is single amino acid residue in the chain, and the atom is each atom within a given amino acid residue (Table 6).

Table 6 Levels of Structure from PDB Data

Level	Description
Structure	Protein structure; may contain multiple models
Model	Particular 3D model of the protein (usually only one)
Chain	Peptide chain
Residue	Amino acid residue in a given chain
Atom	Atoms in a particular amino acid residue

While PDB files can contain multiple models of a protein, most only contain one. Even though there is only one model in our data, we will need to access the first (and only) model using indexing. For the first protein model, use `structure[0]`, and if there were a second, it would be `structure[1]`.

```
protein_model = structure[0]
```

Because of the hierarchical structure, each level of structure can be accessed by iterating through the level above it. For example, the following code will append all atoms in every residue in every chain in the protein model to a list called `atoms`.

```
atoms = []
for chain in protein_model:
    for residue in chain:
        for atom in residue:
            atoms.append(atom)

atoms[:10]
```

```
[<Atom N>,
<Atom CA>,
<Atom C>,
<Atom O>,
<Atom CB>,
<Atom CG>,
<Atom CD>,
<Atom N>,
<Atom CA>,
<Atom C>]
```

This can add up to a large number of `for` loops in your code. Alternatively, you can get more direct access to the different levels of structure using the following methods that yield a generator.

Table 7 Functions for Accessing Different Levels of Structure

Function	Object	Description
<code>get_chains()</code>	Model	Accesses peptide chains
<code>get_residues()</code>	Model, Chain	Accesses amino acid residues
<code>get_atoms()</code>	Model, Chain, Residue	Accesses individual atoms
<code>get_parent()</code>	Atom	Returns parent residue of atom

For example, the following appends all residues in the protein model to a list and displays the first ten residues.

```
res_list = []
for residue in protein_model.get_residues():
    res_list.append(residue)

res_list[:10]
```

```
[<Residue PRO het=  resseq=2 icode= >,
<Residue MET het=  resseq=3 icode= >,
<Residue VAL het=  resseq=4 icode= >,
<Residue LEU het=  resseq=5 icode= >,
<Residue LEU het=  resseq=6 icode= >,
<Residue GLU het=  resseq=7 icode= >,
<Residue CYS het=  resseq=8 icode= >,
<Residue ASP het=  resseq=9 icode= >,
<Residue LYS het=  resseq=10 icode= >,
<Residue ASP het=  resseq=11 icode= >]
```

Parts of the protein structure can also be accessed using keys (i.e., the ID's) of the various levels of structure. This does require more knowledge of the structure beforehand, though. To first get access to the ID's, you can iterate through a structure and use the `get_id()` method to see all of the substructure ID's. Alternatively, you can use the `get_unpacked_list()` function to get a list of all substructures of an object with ID's. For example, below we iterate through the protein model to get the strand ID's. The same can be done with iterating through strands to obtain the residue ID's or through residues to obtain the atom ID's. The strand and atom ID's will be letters (strings) while residue ID's are integers.

```
for strand in protein_model:
    print(strand.get_id())
```

```
A  
B  
C  
D  
E  
F
```

```
strand_A = protein_model['A']
strand_A
```

```
<Chain id=A>
```

```
residue_10 = strand_A[10]
residue_10
```

```
<Residue LYS het= resseq=10 icode= >
```

As a demonstration of both the `get_id()` and `get_unpacked_list()` approaches, below we can see the atoms present in a lysine residue.

```
residue_10.get_unpacked_list()
```

```
[<Atom N>,
 <Atom CA>,
 <Atom C>,
 <Atom O>,
 <Atom CB>,
 <Atom CG>,
 <Atom CD>,
 <Atom CE>,
 <Atom NZ>]
```

```
for atom in residue_10:
    print(atom.get_id())
```

```
N
CA
C
O
CB
CG
CD
CE
NZ
```

```
residue_10['CA']
```

```
<Atom CA>
```

16.2.5 Attributes of Atoms, Residues and Strands

Once we can access the atoms, residues, and strand, information can be extracted such as the identity, 3D coordinates, bond angles, and more. For example, below is a table of interesting atom attributes/functions.

Table 8 Selected Atom Attributes/Functions

Attribute/Function	Description
<code>get_name()</code>	Returns the name of the atom as a string
<code>get_coord()</code>	Returns the xyz coordinates of the atom as an array
<code>get_vector()</code>	Returns the xyz coordinates of the atom as a vector object
<code>transform()</code>	Rotates or translates the atomic coordinates along the xyz axes

The following code is used to obtain the 3D coordinates as arrays for all atoms in the protein model.

```
atom_coords = []
for atom in protein_model.get_atoms():
    atom_coords.append(atom.get_coord())

atom_coords[:5]
```

```
[array([ 89.966, -16.871,  91.86 ], dtype=float32),
 array([ 89.302, -16.084,  90.821], dtype=float32),
 array([ 89.475, -14.614,  91.157], dtype=float32),
 array([ 89.936, -14.284,  92.28 ], dtype=float32),
 array([ 87.831, -16.524,  90.863], dtype=float32)]
```

We can likewise access information about residues such as the following

Table 9 Selected Residue Attributes/Functions

Attribute/Function	Description
<code>get_resname()</code>	Returns the name of the residue as a three-letter code string
<code>get_segid()</code>	Returns the segment ID if available
<code>get_atoms()</code>	Returns the atoms in the residue at a generator
<code>get_unpacked_list()</code>	Returns atoms in the residue as a list

```
res_list = []
for residue in protein_model.get_residues():
    res_list.append(residue.get_resname())

res_list[:5]
```

```
['PRO', 'MET', 'VAL', 'LEU', 'LEU']
```

There are a lot of interesting data obtainable from the strands, but getting access to these data is a little more involved. We need to first initiate (i.e., creating) a polypeptide builder object using `PDB.PPBuilder()` and then build the Polyptetides object using the `build_peptides()` method. The `build_peptides()` function accepts the structure as the one required argument and by default only returns standard amino acids in the peptide chains unless the `aa_only=False` argument is included. The peptetide information in the example below is stored in the variable `peptides` which shows the six peptide chains in this particular protein structure along with sequence identifier integers

that indicate the position of the amino acid along the peptide chain.

```
ppb = PDB.PPBuilder()
peptides = ppb.build_peptides(structure[0])

peptides
```

```
[<Polypeptide start=2 end=474>,
 <Polypeptide start=12 end=475>,
 <Polypeptide start=2 end=113>,
 <Polypeptide start=2 end=474>,
 <Polypeptide start=11 end=475>,
 <Polypeptide start=3 end=113>]
```

We can iterate through the PolyPeptide object (`peptides`) to get the individual peptide chains. With the peptide chains, we can obtain information about the peptide chain, such as the names of amino acids, phi (ϕ) and psi (ψ) angles, etc., using the various methods tabulated below.

Table 10 Selected `strand` Attributes/Functions

Attribute/Function	Description
<code>get_sequence()</code>	Returns the sequence of each strand using single-letter amino acid codes
<code>get_phi_psi_list()</code>	Returns a list of phi and psi dihedral angles in radians
<code>get_ca_list()</code>	Returns list of alpha carbons
<code>get_theta_list()</code>	Returns a list of theta angles in radians
<code>get_tau_list()</code>	Returns list of tau torsional angles in radians

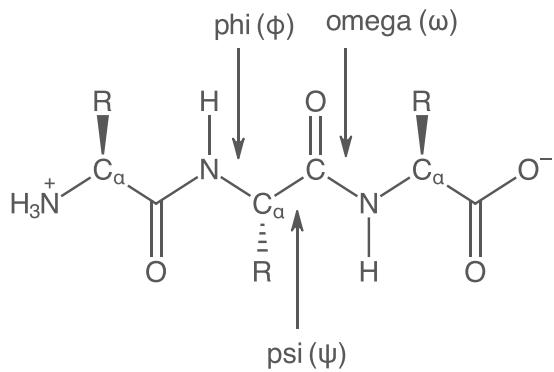
In the example below, we iterate through the peptide strands in `peptides` and print the theta angles in radians.

```
for strand in peptides:
    C_a = strand.get_theta_list()
    print(C_a[:5])
```

```
[np.float64(1.9516790193468274), np.float64(2.287601877880969), np.float64(1.7982004900815982), np.float64(2.3268945513738237), np.float64(1.848500563201942), np.float64(2.2282952590602303), np.float64(1.511341690238054), np.float64(1.5829981072097719), np.float64(1.5844900414925847), np.float64(1.9673252796978447), np.float64(2.3178730899065365), np.float64(1.7942727121398567), np.float64(1.971514259887586), np.float64(2.2624600169176046), np.float64(1.7954830025275168), np.float64(2.216191919496574), np.float64(1.508875006315003), np.float64(1.5822328296905053), np.float64(1.5822328296905053)]
```

16.2.6 Ramachandran Plots

As an example application, we can generate a Ramachandran plot which visualizes the trends of the psi (ψ) versus phi (ϕ) dihedral angles along peptide chains. While the omega (ω) dihedral angles tend to be flat, the psi (ψ) versus phi (ϕ) dihedral angles tend to exist in distinct ranges.



The general methodology below is:

1. Parse PDB files in the **data/proteins** folder using a PDB parser
2. Build a PolyPeptide object using a PDB builder
3. Iterate over the peptides and store the $\text{psi} (\psi)$ and $\text{phi} (\phi)$ dihedral angles
4. Plot the results as $\text{psi} (\psi)$ versus $\text{phi} (\phi)$

```
phi, psi = [], []
current_directory = os.getcwd()
data_folder = os.path.join(current_directory, 'data/proteins')

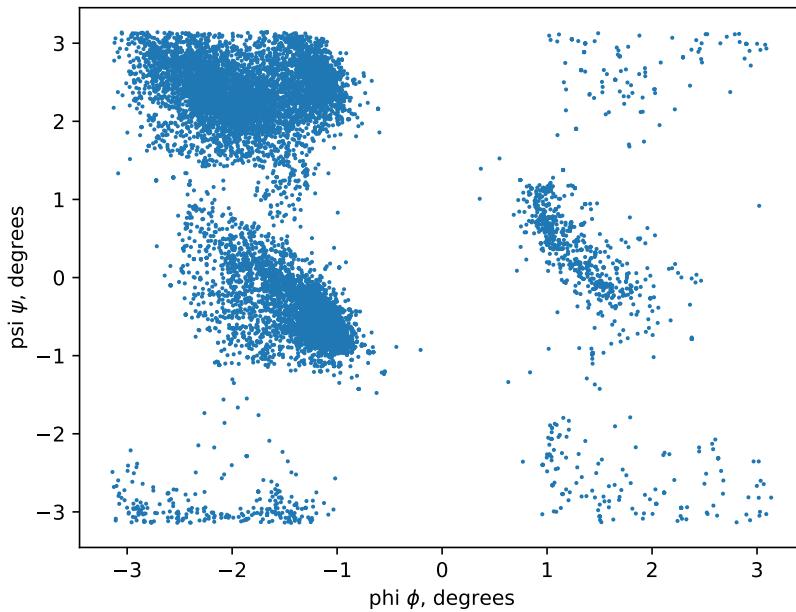
parser = PDB.PDBParser()
ppb = PDB.PPBuilder()

for file in os.listdir('data/proteins'):
    if file.endswith('pdb'):
        structure = parser.get_structure('file', os.path.join(data_folder, file))
        peptides = ppb.build_peptides(structure[0])
        for strand in peptides:
            phi.extend(x[0] for x in strand.get_phi_psi_list()[1:-1])
            psi.extend(x[1] for x in strand.get_phi_psi_list()[1:-1])
```

phi[:10]

```
[np.float64(-1.3150748393961473),
 np.float64(-2.7159390905663523),
 np.float64(-2.909570150157909),
 np.float64(-1.9350566725748244),
 np.float64(-2.3853630088972273),
 np.float64(1.3306807975618),
 np.float64(-1.6592559311514123),
 np.float64(-1.788129930665399),
 np.float64(-1.3609620204292667),
 np.float64(-1.014135279691033)]
```

```
plt.scatter(phi, psi, s=1)
plt.xlabel('phi $\backslash\phi$', degrees')
plt.ylabel('psi $\backslash\psi$', degrees');
```

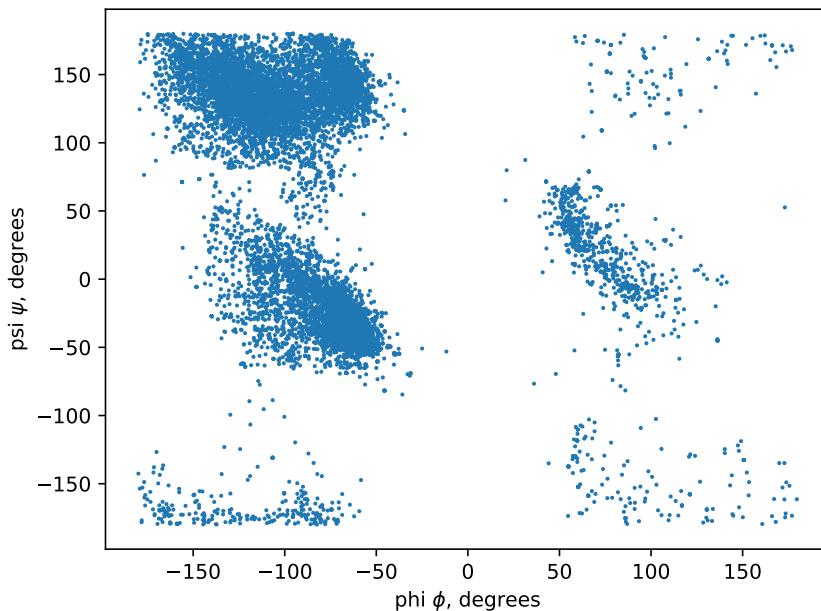


You may notice that the first and last dihedral angles were sliced off the list of phi and psi angles (last two lines of code). This is because there are no phi (ϕ) values for first amino acid and no psi (ψ) values in the last amino acid of a strand. Dihedral angle measurements require four atoms, and the terminal amino acids are missing one of the required four atoms. For example, phi (ϕ) dihedral angles are measured along the N-C _{α} bond of a C(O)-N-C _{α} -C(O) chain of atoms, but the first amino acid only has N-C _{α} -C(O).

The Ramachandran plot above is in radians which can be converted to degree (1 radian = $180/\pi$) as is done below.

```
import math
psi_deg = [rad * (180 / math.pi) for rad in psi]
phi_deg = [rad * (180 / math.pi) for rad in phi]

plt.scatter(phi_deg, psi_deg, s=1)
plt.xlabel('phi $\backslash\phi$, degrees')
plt.ylabel('psi $\backslash\psi$, degrees');
```



Other representations of Ramachandran plots using different plotting types or color coding the markers based on

secondary protein structure can be seen in the [Notebook 3 of the Visualization of Top8000 Protein Dataset](#) mini tutorial.

16.3 Visualization of Molecules

There are many pieces of software for viewing molecular structures directly from your desktop, but there are currently few for viewing structures within a Jupyter notebook. This section provides a brief introduction to nglview for interactively viewing molecular structures. Additional information on nglview can be found on the [nglview documentation page](#).

💡 Tip

Nglview often requires a restart after installation before working. As of this writing, I am having good luck with the most recent version, 3.1.2, working in JupyterLab for my students and me.

Nglview is not a standard library for Anaconda or Colab, so it needs to be installed, and as of this writing, nglview can be installed using either [pip](#) or [conda](#). Below, it will be imported with the `nv` alias. A restart may be required after installation.

```
import nglview as nv
```

16.3.1 Loading Structures in Nglview

Molecular structures can be loaded using a number of different sources including directly from files, from RDKit Molecule objects, from Biopython structure objects, and from psi4 molecules among others. Below is table of some key functions for loading molecular structures.

Table 11 A Selection of Nglview Functions for Loading Structural Data

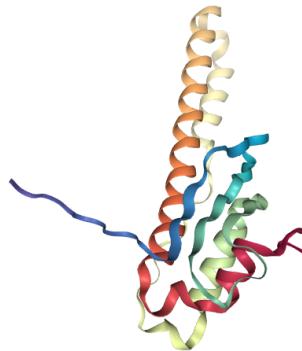
Function	Description
<code>nv.show_file()</code>	Loads from a file (e.g., PDB or mmCIF) on your computer
<code>nv.show_pdbid()</code>	Fetches data from RCSB database when provided a PDB ID (e.g., '7aiz')
<code>nv.show_rdkit()</code>	Loads structure from a 3D RDKit Molecule object
<code>nv.show_biopython()</code>	Loads data from a Biopython structure object

As our first example, we will load a file using the `show_file()` function which accepts a protein data file such as PDB. The structure is displayed in an interactive window where clicking and dragging rotates the molecule and scrolling zooms in and out. The size of this window can be expanded or contracted using the little gray arrow control(s) on the right corners of the display window.

💡 Note

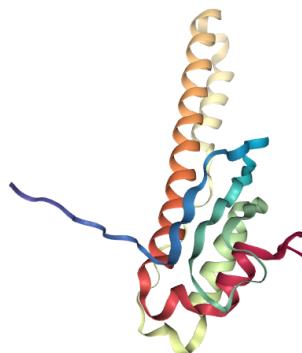
The following examples are no longer interactive. If you run this code in your own notebook, you will be able to interact with the structures.

```
prot = nv.show_file('data/3hpb.pdb')
prot
```



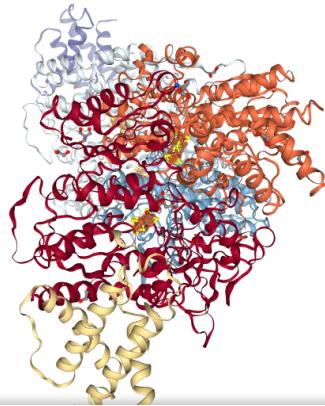
The next example accepts the four-letter ID for a protein crystal structure and fetches the data from an online database.

```
prot = nv.show_pdbid('3hpb')
prot
```



We can also view a molecule loaded from a Biopython structure object (see [section 16.2.2](#)) using the `show_biopython()` function.

```
structure = parser.get_structure('7aiz', 'data/7aiz.pdb')
prot_struct = nv.show_biopython(structure)
prot_struct
```

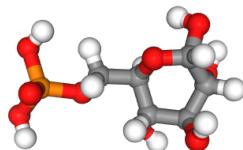


RDKit Molecule objects can also be viewed in nglview using the `show_rdkit()` function, but first a 3D representation of the molecule needs to be generated using the `AllChem.EmbedMolecule(mol_object)` function. Many SMILES representations do not include all of the hydrogens, so implicit hydrogens need to be added in using the `Chem.AddHs()` method. The visualization of glucose 6-phosphate from a SMILES representation is shown below.

```
from rdkit import Chem
from rdkit.Chem import AllChem

mol = Chem.MolFromSmiles('O[C@H]1[C@H](O)[C@@H](COP(=O)(O)=O)OC(O)[C@H]1O')
mol = Chem.AddHs(mol) # add H's
AllChem.EmbedMolecule(mol) # generate 3D structure

G6P = nv.show_rdkit(mol)
G6P
```



16.3.2 Nglview Representations

The way molecules are represented by nglview can be modified using `add_representation(rep)` which takes a variety of string parameters indicating the representation. A few examples of representations are listed below with a more complete list provided on the [nglview documentation](#) page.

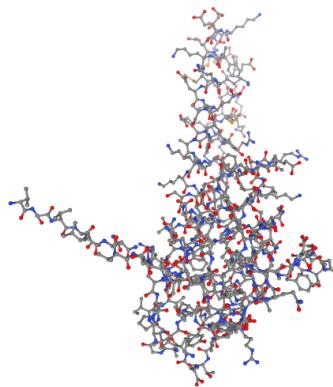
The default representation is `cartoon` which shows the peptide backbone as strands and ribbons (for secondary structures). It is important to clear the default representation using the `clear_representation()` method before adding a new representation. Otherwise, you will have both representations showing up on top of each other, unless this is what

you want.

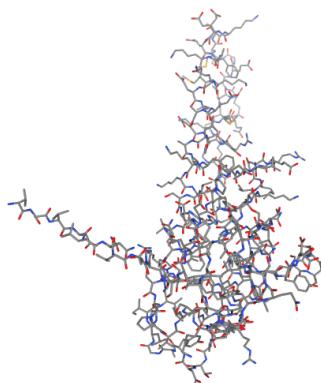
Table 12 Selected Molecular Representations

Representation	Description
cartoon	Cartoon with strands and ribbons; sidechains not shown
ball+stick	Atomic spheres and stick bonds; sidechains shown
licorice	Balls and sticks where atoms and bonds have the same radii; sidechains shown
rope	Backbone is shown as a tube; sidechains not shown
spacefill	Spacefilling model with atoms showing atomic size; sidechains shown
surface	Shows the surface of the molecule; other surface parameters available

```
prot_3hpb = nv.show_file('data/3hpb.pdb')
prot_3hpb.clear_representations()
prot_3hpb.add_representation('ball+stick', selection='protein')
prot_3hpb
```



```
prot_3hpb = nv.show_file('data/3hpb.pdb')
prot_3hpb.clear_representations()
prot_3hpb.add_representation('licorice', selection='protein')
prot_3hpb
```



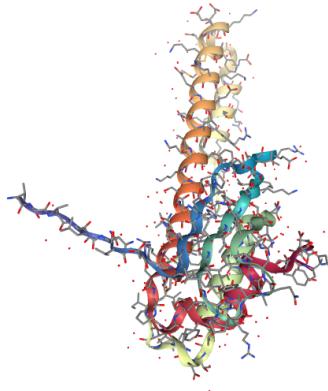
Different sections of a protein can be represented differently using the `selection=` parameter in the `add_representation()` function. This includes using residue numbers from the structure file or using a variety of string arguments that select different types of structures. A short list of options is included below with a more complete list on the [nglview documentation](#) page.

Table 13 Selected Options for the `selection=` Parameter

Selection Option	Surace Surrounded Region(s)
<code>All</code>	Everything (default)
<code>protein</code>	Peptide chains
<code>dna</code>	DNA regions
<code>water</code>	Waters
<code>helix</code>	Helicies
<code>sheet</code>	Sheets
<code>hydrophobic</code>	Hydrophobic amino acids
<code>hydrophilic</code>	Hydrophilic amino acids
<code>acidic</code>	Acidic amino acids
<code>basic</code>	Basic amino acids
<code>polar</code>	Polar amino acids
<code>nonpolar</code>	Nonpolar amino acids

As an example, we can show a protein structure with the backbone as the default cartoon and the sidechains using a licorice structure as shown below.

```
prot_1rpy = nv.show_file('data/3hpB.pdb')
prot_1rpy.add_representation('licorice', selection='sidechains')
prot_1rpy
```



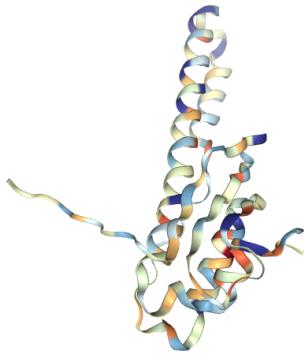
The colors can be customized using the `color=` parameter. This can accept either a color name as a string (e.g., `'blue'`) or color code the molecule based on other features such as hydrophobicity or chain.

Table 14 Selected Options for the `color=` Parameter

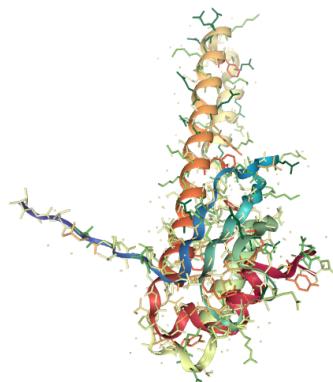
Option	Description
<code>chainid</code>	Each chain is colored differently
<code>chainname</code>	Each chain is colored differently
<code>element</code>	Uses standard element color coding (for licorice or ball+stick representations)
<code>hydrophobicity</code>	Sections colored by peptide hydrophobicity
<code>moleculetype</code>	Each molecule colored by type (e.g., peptide chain versus sulfate)
<code>residueindex</code>	Color changes gradually down the peptide chain
<code>resname</code>	Each peptide side chain is assigned a color
<code>sstruc</code>	Colors based on secondary structure

Not all of the above options work for every representation, and some only work on the peptide side chains.

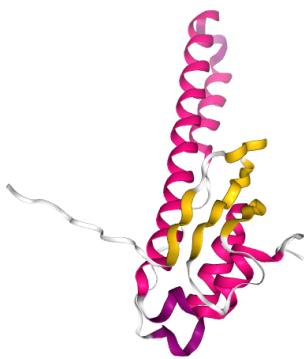
```
prot_1rpy = nv.show_file('data/3hpB.pdb')
prot_1rpy.clear_representations()
prot_1rpy.add_representation('cartoon', color='hydrophobicity')
prot_1rpy
```



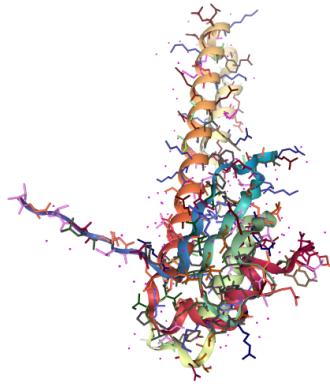
```
prot_1rpy = nv.show_file('data/3hpб.pdb')
prot_1rpy.add_representation('licorice', color='hydrophobicity')
prot_1rpy
```



```
prot_1rpy = nv.show_file('data/3hpб.pdb')
prot_1rpy.clear_representations()
prot_1rpy.add_representation('cartoon', color='sstruc')
prot_1rpy
```



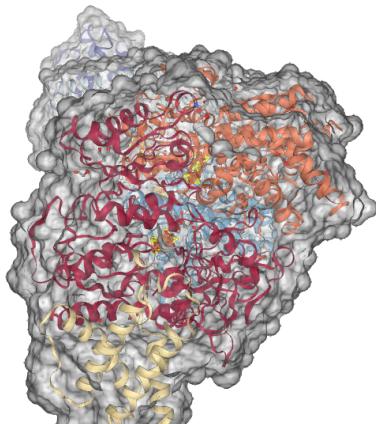
```
prot_1rpy = nv.show_file('data/3hpB.pdb')
prot_1rpy.add_representation('licorice', color='resname')
prot_1rpy
```



16.3.3 Showing Surfaces

To view the molecule with a surface, use the `add_surface()` method which takes a number of optional parameters. Possibly the most important is `opacity=` which accepts a float from $0 \rightarrow 1$ indicating how opaque the surface is with 1 exhibiting no translucency and 0 being completely transparent.

```
full_surface = nv.show_biopython(structure)
full_surface.add_surface(opacity=0.3)
full_surface
```

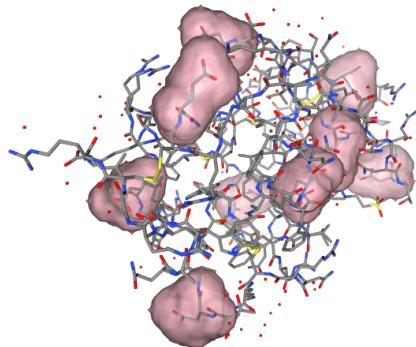


Another useful parameter is the `selection=` parameter that operates like described in [section 16.3.2](#) where only the selected components have a surface around them. In the example below, only acidic amino acids are wrapped in a surface.

```

acidic = nv.show_file('data/8pfy.cif')
acidic.clear_representations()
acidic.add_representation('licorice')
acidic.add_surface(selection='acidic',
                   opacity=0.4,
                   color='pink')
acidic

```

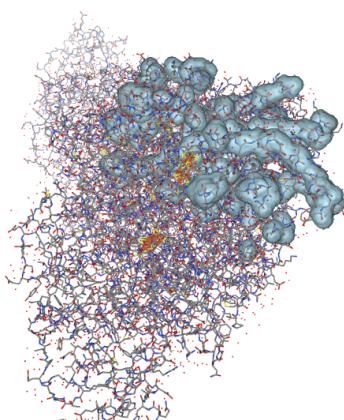


We can also use `and` and `or` to produce more complex selections such as below where we only wrap backbones of residues that are acidic or basic and must only be in strand B.

```

acidbase = nv.show_biopython(structure)
acidbase.clear_representations()
acidbase.add_representation('licorice')
acidbase.add_surface(selection=':B and backbone and (basic or acidic)',
                     opacity=0.3,
                     color='lightblue')
acidbase

```

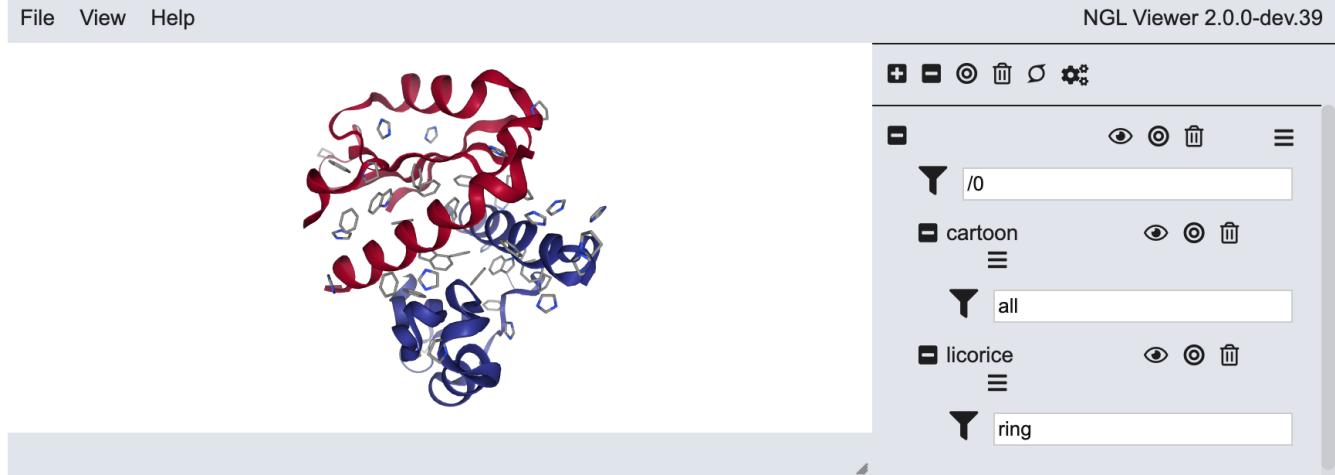


16.3.4 Interactive GUI

Nglview also supports an interactive graphical user interface (GUI) within Jupyter notebooks. From the panel on the right, the user can add representations and change selections using the same selection keywords as above (see Table 12). From the **File** menu on the top left, new files can be opened or proteins can be fetched using the protein ID.

```
prot = nv.show_pdbid('1rpy')
prot.clear_representations()
prot.add_representation('cartoon')
prot.add_representation('licorice', selection='ring')

prot.gui_style = 'ngl'
prot
```



Further Reading

1. PDB format documentation. <https://www.wwpdb.org/documentation/file-format> (free resource)
2. Introduction to PDB File Format. <https://www.cgl.ucsf.edu/chimera/docs/UsersGuide/tutorials/pdbintro.html> (free resource)
3. Biopython Website. <https://biopython.org> (free resource)
4. Nglview Manual. <https://nglviewer.org/ngl/api/manual/index.html> (free resource)
5. Scikit-bio Website. <http://scikit-bio.org> (free resource). This is another related library with many similar features that may be of interest to Biopython users.
6. Biopython Publication. Cock, P. J. A.; Antao, T.; Chang, J. T.; Chapman, B. A.; Cox, C. J.; Dalke, A.; Friedberg, I.; Hamelryck, T.; Kauff, F.; Wilczynski, B.; De Hoon, M. J. L. Biopython: Freely Available Python Tools for Computational Molecular Biology and Bioinformatics. *Bioinformatics* **2009**, 25, 1422– 1423,

Chapter 17: Command Line & Spyder

Contents

- 17.1 Navigating the Terminal
- 17.2 Running Scripts
- 17.3 Additional Inputs
- 17.4 Running .py Files in Jupyter
- 17.5 Spyder
- Further Reading
- Exercises

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 or Jupyter's terminal. 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 terminal to run your Python script which is discussed below.

17.1 Navigating the Terminal

The terminal is the command line interface used in macOS and unix-like systems such as the Linux and BSD families and allows users to perform a wide array of tasks from installing and running software to file management. If you are using Linux or Mac, launch the terminal from the applications, and if you are on Windows, you will likely first need to activate the Bash command line before proceeding. Alternatively, if you are using the JupyterLab version of Jupyter, you can launch a terminal window from the Launcher menu

(see [section 0.2](#), Figure 2). In section 17.2, you will learn to run Python scripts from the terminal, 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 brief primer on navigating the file system through the Terminal.

17.1.1 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 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 view the contents of the directory, we can list its contents 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 such as icons, and it is often best not to alter or delete these invisible files.

17.1.2 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 ..` (space with two periods) is used.

```
$ cd ..
```

There is certainly much more that can be done in the terminal, but this is enough of a foundation for you to find and run scripts as we will do below.

17.2 Running Scripts

Now that you know the basics of the terminal command line, we can now run our first script. Open a text editor of your choice. Be careful if you write Python code in a regular word processor (e.g., Word, LibreOffice, Pages, etc...) as it may save extra formatting in any text file generated. A better option is to either use Spyder introduced in [section 17.5](#) or (easiest) select Python File from the JupyterLab launcher. Write some Python code in a new file and save it 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. For this demonstration, I'll include the following code in my text file.

```
import random
rng = np.random.default_rng()
rdn = rng.integers(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` followed by a space and then drag-and-drop the file into the terminal window. This will result in the file path and name being automatically pasted into the terminal window.

```
$ python /Users/Me/Desktop/first_script.py
```

65

17.3 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.

17.3.1 User Inputs

In the event you want the 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
```

17.3.2 `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.

17.4 Running .py Files in Jupyter

As a way to combine Python scripts in external .py files and Jupyter notebooks, it is possible to run these Python scripts from the Jupyter notebook using the `%run` magic command. As an example, let's say we have the following code in a file called *dist.py*.

```
pt1 = (1,5,9)
pt2 = (9, 0, 3)

def distance(coord1, coord2):
    x1, y1, z1 = coord1
    x2, y2, z2 = coord2

    return ((x1 - x2)**2 + (y1 - y2)**2 + (z1 - z2)**2)**0.5
```

We can run this code from a Jupyter notebook using the following command. Like we've seen previously, Jupyter assumes the referenced file is in the same directory as the Jupyter notebook unless otherwise indicated.

```
%run dist.py
```

```
pt1
```

```
(1, 5, 9)
```

```
distance(pt1, pt2)
```

```
11.180339887498949
```

Now that the *dist.py* file has been executed, the variables and function are available in the Jupyter notebook as if this code had been run in a Jupyter code cell.

17.5 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 1). The latter method is often slower because it requires that Navigator be first launched.

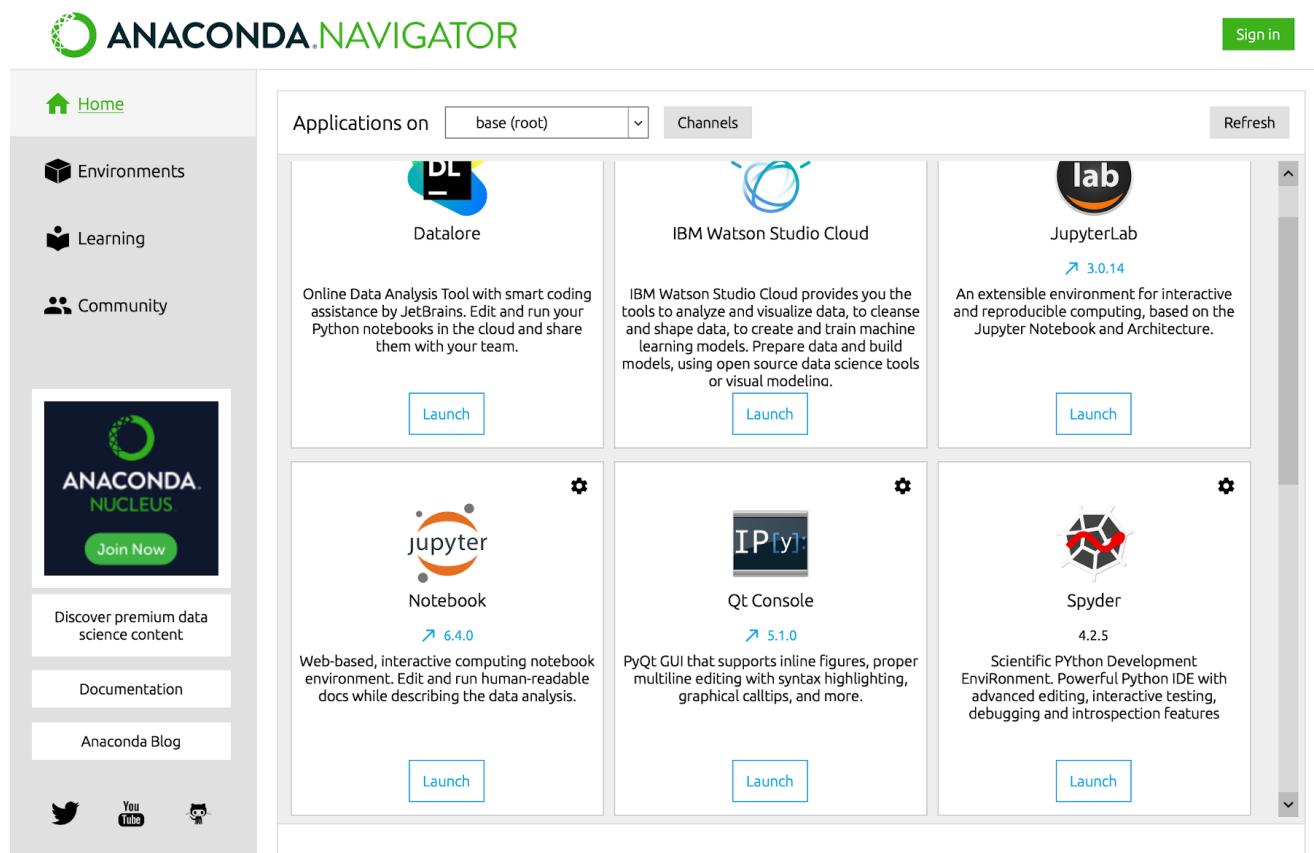


Figure 1 A screenshot of the Anaconda Navigator application launcher window.

Once Spyder has launched, you will be greeted by an interface divided into three windows (Figure 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 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.

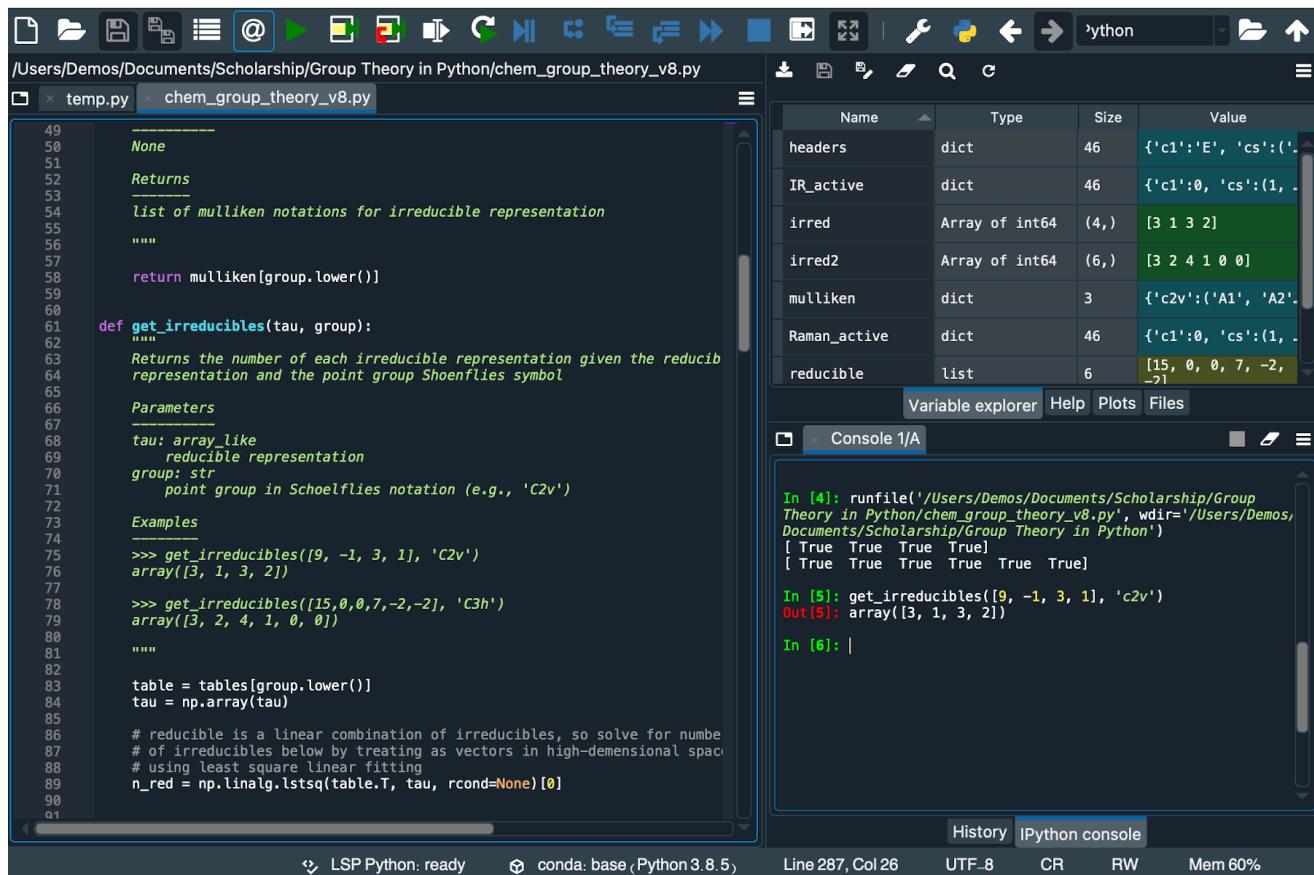


Figure 2 The Spyder interface with the text editor (left), variable explorer (top right), and interpreter (bottom right).

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/> (free resource)

Exercises

Complete the following problems by writing a Python scripts either in a text editor or Spyder and run them from the terminal. JupyterLab, 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! Any data file(s) referred to in the problems can be found in the [data](#) folder in the same directory as this chapter's Jupyter notebook. Alternatively, you can download a zip file of the data for this chapter from [here](#) by selecting the appropriate chapter file and then clicking the **Download** button.

1. When an electron in a hydrogen atom relaxes from a higher to a lower energy orbital, a photon is released with the wavelength in nm 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 prints the wavelength (λ) of light emitted with units.

$$\frac{1}{\lambda_{nm}} = 1.097 \times 10^7 nm^{-1} \left(\frac{1}{n_i^2} - \frac{1}{n_f^2} \right)$$

2. In the folder titled 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 datasets in the folder. This script will accept the folder name instead of the file name. Remember to use the

 module described in [section 2.4.1](#).

Appendix 0: Ipython Widgets

Contents

- Basic Widgets
- Generating Widgets using Decorators
- Customized Widgets
- Slow Functions
- Simulating NMR Splitting Patterns

You can create widgets such as sliders or check boxes in Jupyter notebooks to make it easier to rapidly modify input values in your code. This can be useful for rapid experimentation with different parameters in your code or as part of a demo. For this, we will use [Jupyter Widgets](#). In the following examples, we will simulate an NMR free induction decay (fid) signal and NMR splitting pattern to see how changing various parameters affects the end result. This section assumes knowledge of chapters 0-4, but you probably can (mostly) follow along if you are through chapter 1.

Note

While the widgets in this appendix are movable, the graphs do *not* change because this is a static book with no kernel running in the back. If you download this notebook and run it yourself, the values and graphs will automatically update as you interact with the widgets.

This notebook requires that you have [ipywidgets installed either using pip or conda](#). There is a good chance you already have it installed, though. The last example also assumes you have nmrsim installed from [section 12.2](#). This appendix assumes the following imports.

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

from nmrsim import Multiplet
from nmrsim.plt import mplplot

from ipywidgets import interact, interact_manual, FloatSlider, FloatRangeSlider, RadioButtons, fixed
```

Basic Widgets

To create a widget that affects your code, you must first package the code in a single Python function. Below we will simulate an NMR free induction decay (fid) by the following equation where t is time in seconds, ν is frequency in Hz, and T_2 is the relaxation constant.

$$\text{signal}(t) = \cos(2\pi\nu t)e^{-t/T_2}$$

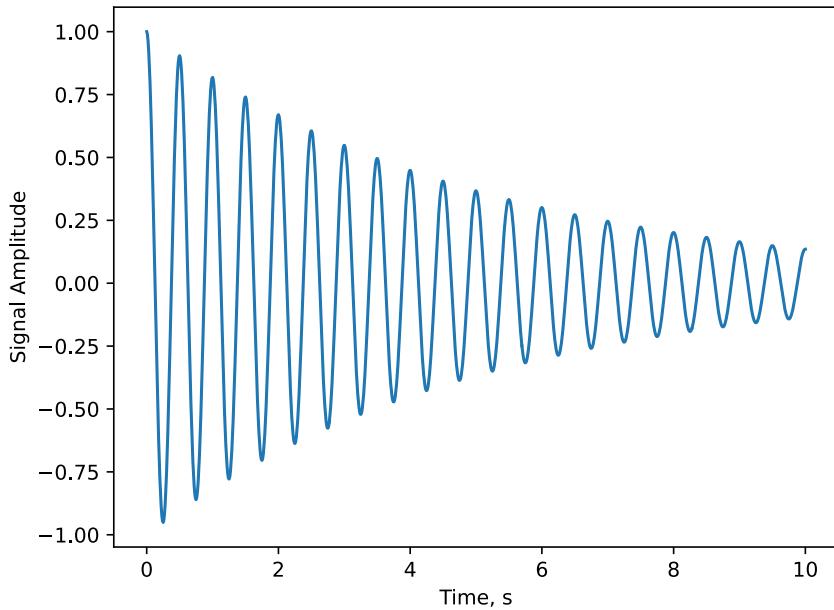
We will see how the frequency (ν) and T_2 affect the appearance of the fid. To do this, we will write a function,

`plot_fid(nu, T2)`, that accepts these two parameters as arguments and generates a plot of signal versus time.

```
def plot_fid(nu, T2):
    t = np.linspace(0,10,1000)
    wave = np.cos(2*np.pi*nu*t)
    decay_func = np.exp(-t/T2)

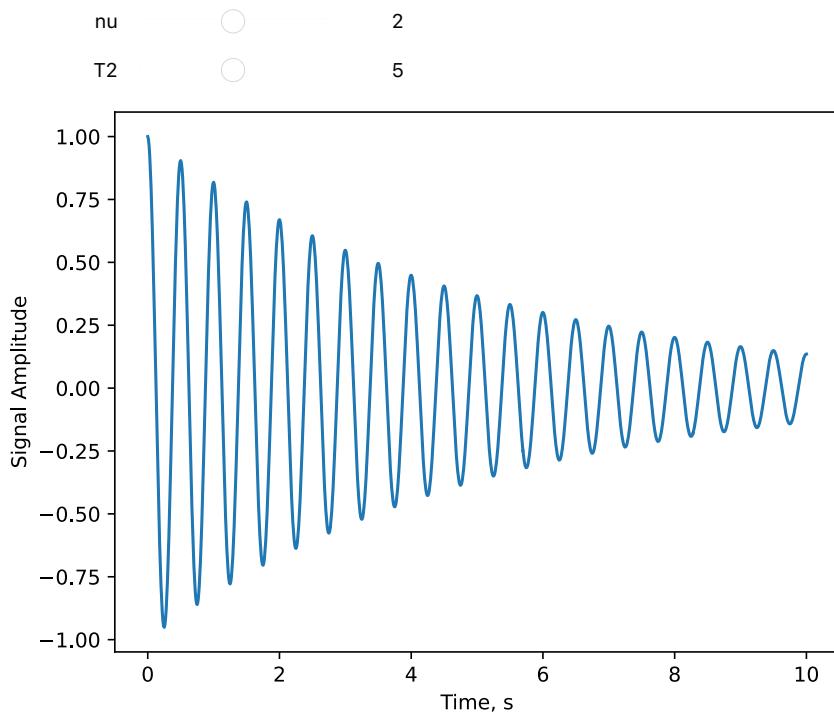
    plt.plot(t, wave*decay_func)
    plt.xlabel('Time, s')
    plt.ylabel('Signal Amplitude')

plot_fid(2, 5)
```



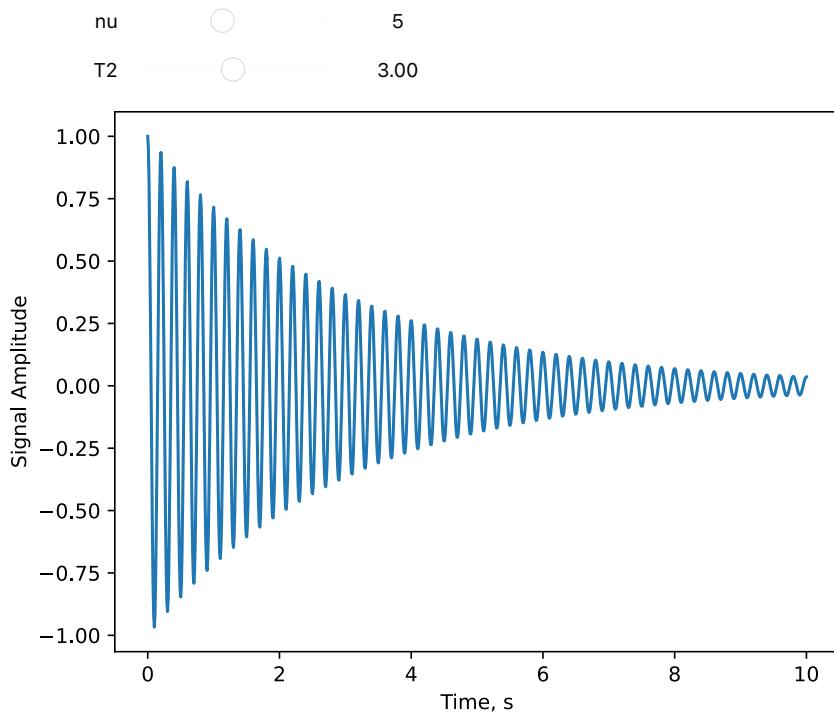
To make this function interactive, we will use the `interact()` function from ipywidgets which takes our function above as a required, positional argument. We also need to provide initial values for our two parameters as keyword arguments as demonstrated below. When we run our code, two sliders appear above our graph. As noted above, the sliders do not affect the plot *in this static book* but would automatically change the graph if you run the code in your own Jupyter notebook.

```
interact(plot_fid, nu=2, T2=5);
```



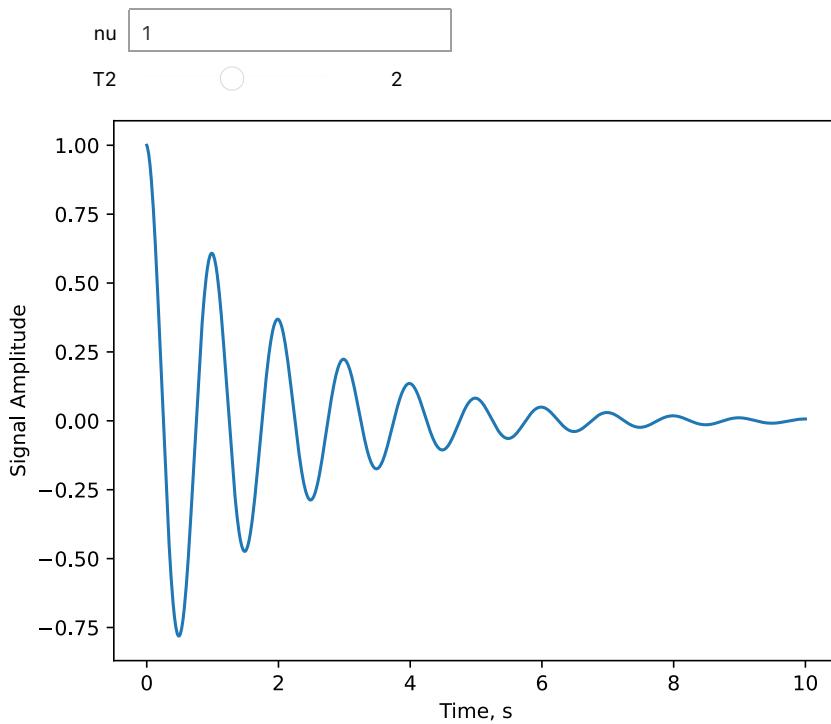
The `interact()` function makes a guess at the ranges of values you might need for your parameters, but you can also explicitly define these by providing a tuple with minimum, maximum, and step size values in this order (`(min, max, step)`).

```
interact(plot_fid, nu=(1,10,1), T2=(1,5, 0.5));
```



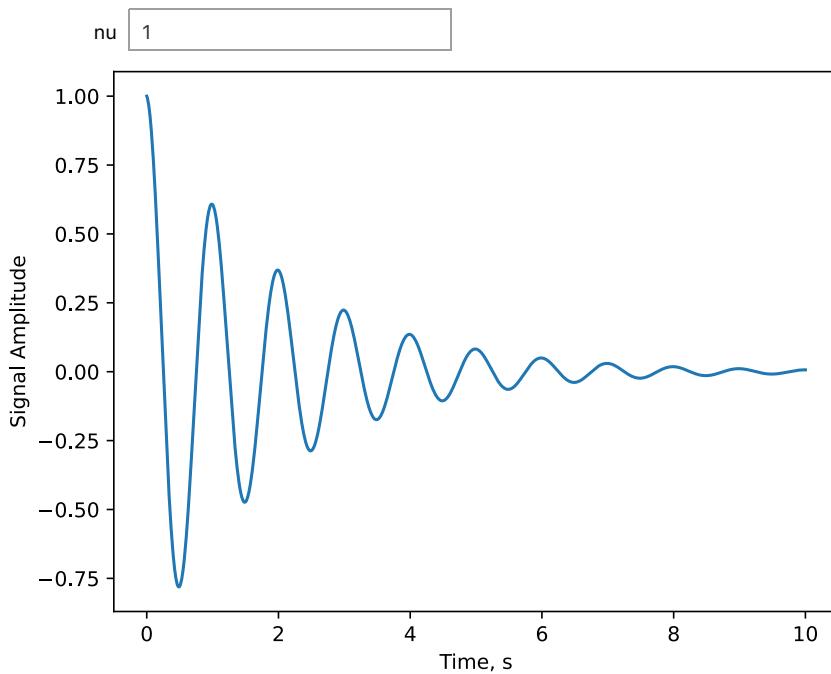
At this point, you may be wondering why you get slides versus any other type of widget. The `interact()` function automatically generates sliders for function arguments with numerical values. If the argument in `interact()` contains a list, a dropdown menu appears, a bool generates a check box, and a text argument produces a text box.

```
interact(plot_fid, nu=[1,2,3,4,5,6], T2=2);
```



If you want a value to be unchangeable by the widgets, wrap the desired value in the `fixed()` function as demonstrated below.

```
interact(plot_fid, nu=[1,2,3,4,5,6], T2=fixed(2));
```

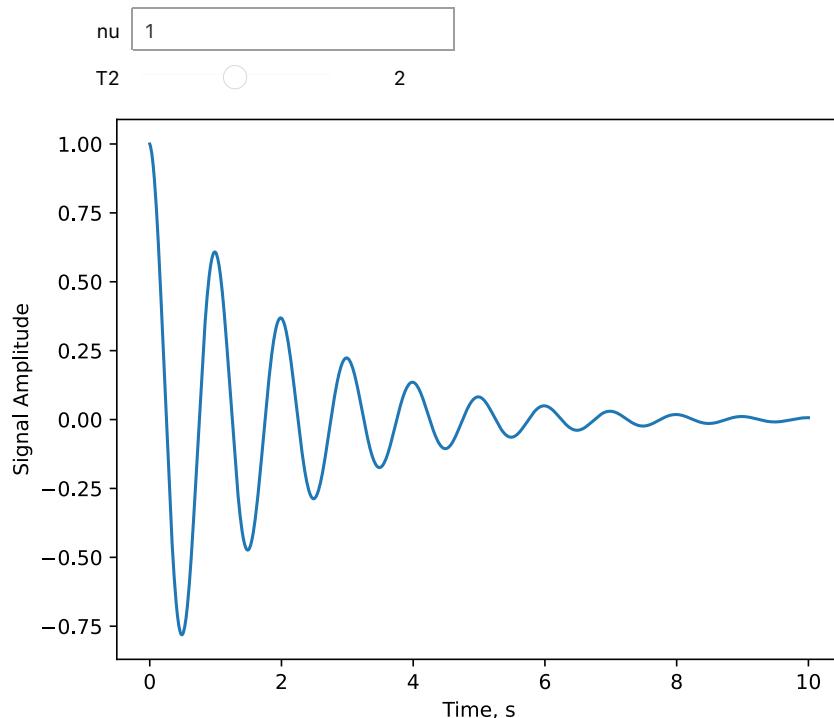


Generating Widgets using Decorators

Another way to create ipython widgets is to employ the `interact()` function as a decorator for your function. Instead of calling the `interact()` function after you define your function, you place `@interact()` just above your own function definition and skip feeding your function into the `interact()` function. The code below generates an equivalent outcome as we saw just above.

```
@interact(nu=[1,2,3,4,5,6], T2=2)
def plot_fid(nu, T2):
    t = np.linspace(0,10,1000)
    wave = np.cos(2*np.pi*nu*t)
    decay_func = np.exp(-t/T2)

    plt.plot(t, wave*decay_func)
    plt.xlabel('Time, s')
    plt.ylabel('Signal Amplitude')
```



Customized Widgets

You can customize your widgets with more widget types listed in the [ipywidgets documentation page](#). For example, if we want our frequency to be controlled by buttons, we can create a button widget with ipywidget's `RadioButtons()` function and assign that to the frequency variable in the `interact()` function. Each customized widget can have different arguments, so it is a good idea to view the documentation on the [ipywidgets documentation page](#).

```
button_widget = RadioButtons(options=[1,2,3,4,5,6])
interact(plot_fid, nu=button_widget, T2=(1,5,0.5));
```

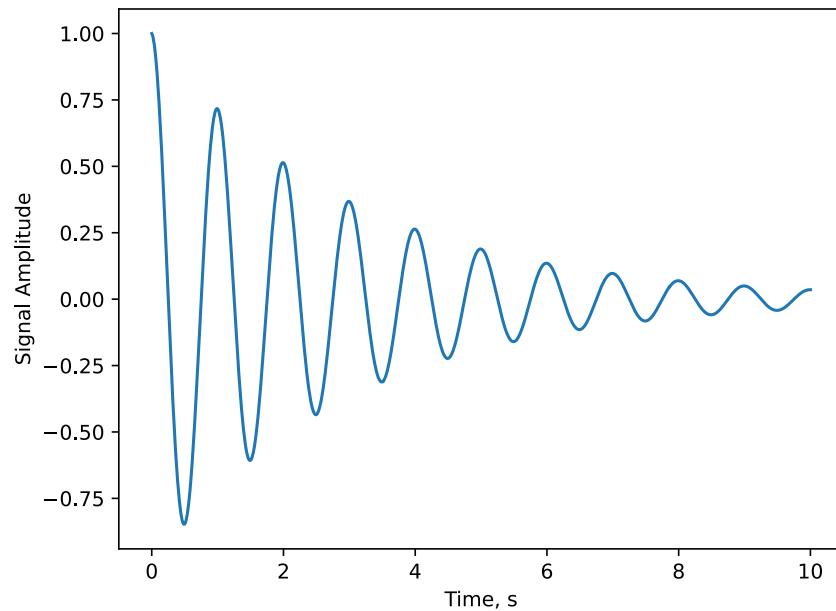
nu

- 1
- 2
- 3
- 4
- 5
- 6

T2



3.00



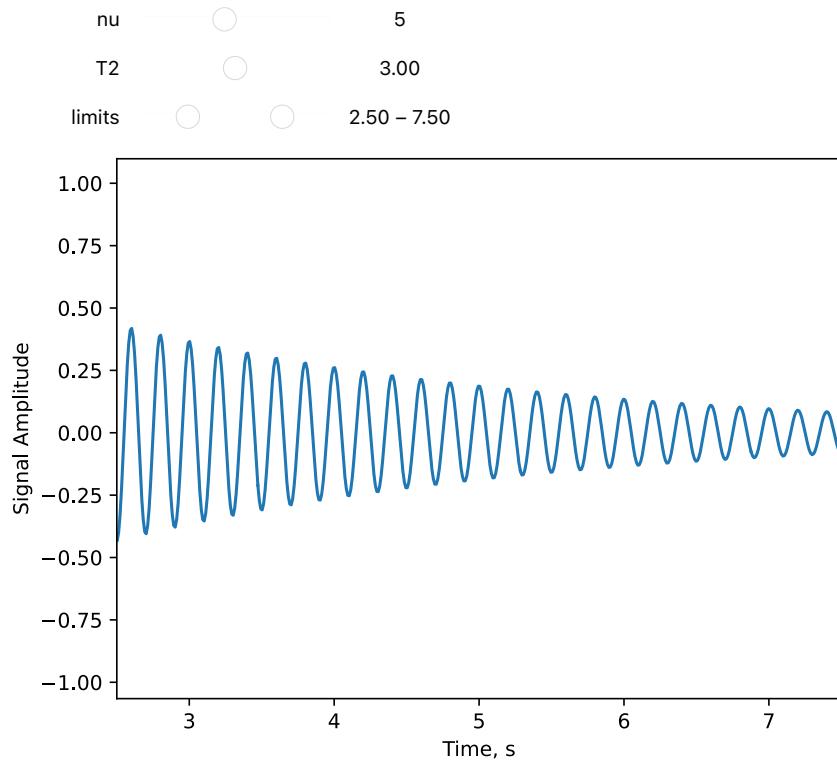
As a second example of a custom widget, we will create a slider with upper and lower limits using either

`FloatRangeSlider()` or `IntRangeSlider()`. As you might guess, one is for float values and the other is for integers. It is important to note that these two widgets return two values in a tuple, so your function must be written to accept a two-valued tuple as an argument.

```
def plot_fid_limits(nu, T2, limits):
    t = np.linspace(0,10,1000)
    wave = np.cos(2*np.pi*nu*t)
    decay_func = np.exp(-t/T2)

    plt.plot(t, wave*decay_func)
    plt.xlabel('Time, s')
    plt.ylabel('Signal Amplitude')
    plt.xlim(limits)

frs = FloatRangeSlider(min=0, max=10, step=0.5)
interact(plot_fid_limits, nu=(1,10,1), T2=(1,5, 0.5), limits=frs);
```



Slow Functions

If your function is slow to run, you may not want it to execute every time a slider moves. There are two solutions to this. The first is to use the `interact_manual()` function which is a cousin of the `interact()` function except that your function only runs when you click the Run Interact button.

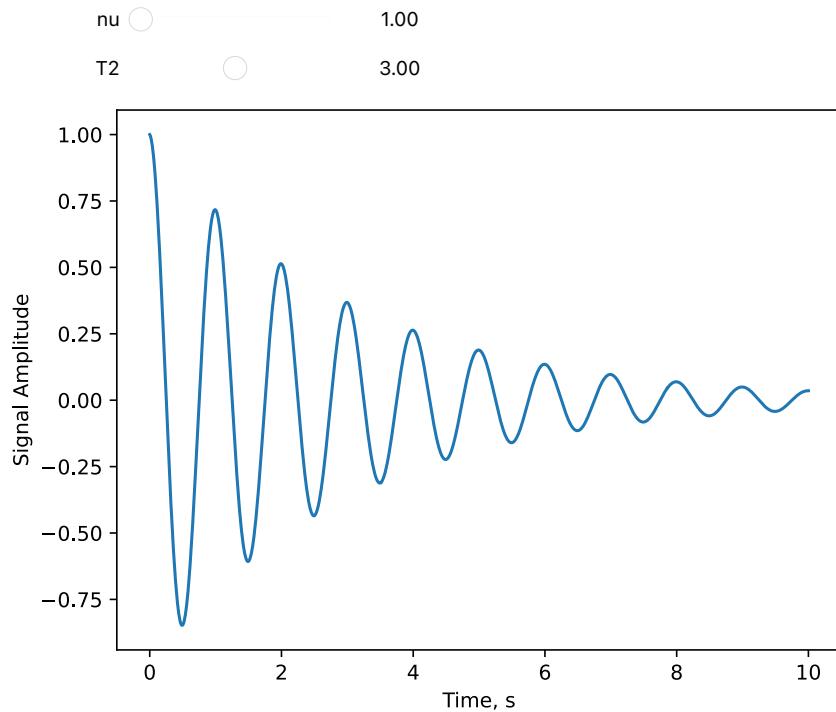
```
interact_manual(plot_fid, nu=(1,10,1), T2=(1,5,0.5));
```

nu 5
T2 3.00

Run Interact

The second option is to create a custom slider widget and set the parameter `continuous_update=False`. This will result in your function only running once you let go of the slider with your mouse. A basic float slider can be created with the `FloatSlider()` function like is done below.

```
fs = FloatSlider(min=1, max=10, step=1, continuous_update=False)
interact(plot_fid, nu=fs, T2=(1,5, 0.5));
```



Simulating NMR Splitting Patterns

As an additional example, we will simulate NMR splitting patterns below using the `nmrsim` library introduced in [section 12.2](#). For this, we will use the `Multiplet()` function which takes the resonance frequency in Hz (`v`) as the first positional argument followed by the intensity (`I`) of the resonance signal. The parameters that we are most interested in here is the number of each type of neighbors and the coupling constants with these neighbors which are provided as coupling constants(`J`) and number of nuclei (`n_nuc`) pairs in a list of tuples.

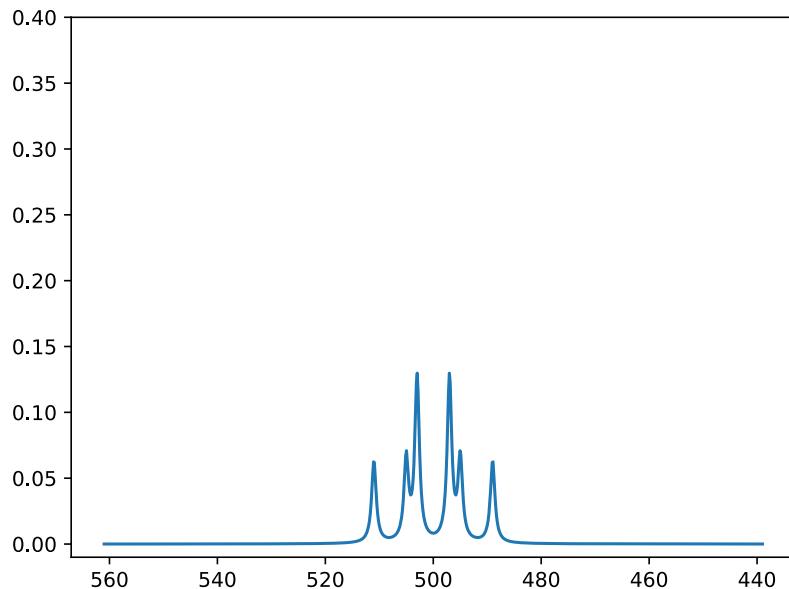
```
Multiple(v, I, [(J1, n_nuc1), (J2, n_nuc1)])
```

The function below assumes our signal is being split by two types of neighboring nuclei - `n_nuc1` of the first type of neighbors with a `J1` coupling constant and `n_nuc2` of the second type of neighbors with a `J2` coupling constant. This resonance will be visualized using the `mplplot()` function from `nmrsim`.

```
def plot_nmr(J1=8.0, J2=6.0, n1=2, n2=1, y_max=0.4):
    res = Multiplet(500, 1, [(J1, n1), (J2, n2)])
    mplplot(res.peaklist(), y_max=y_max)

plot_nmr();
```

```
[<matplotlib.lines.Line2D object at 0x10b375df0>]
```

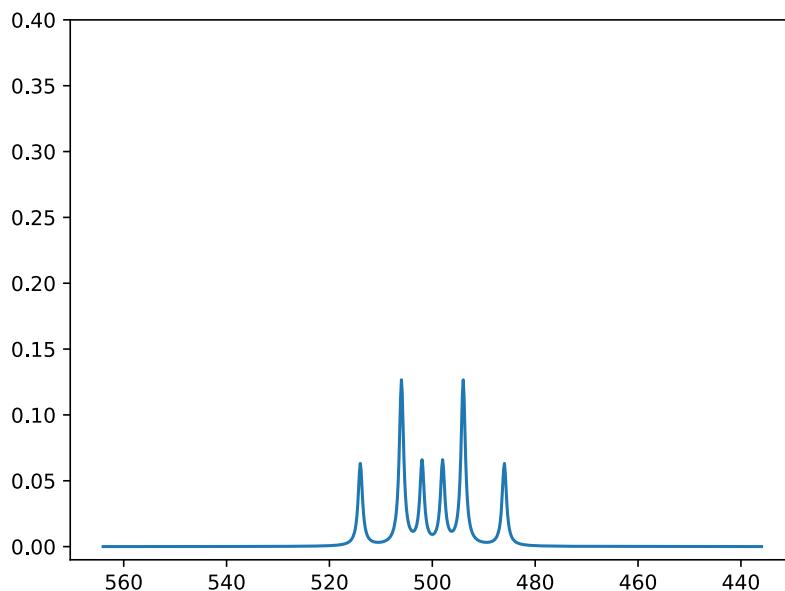


We can again feed our function into `interact()` which produces sliders because our parameters are all numbers.

```
interact(plot_nmr, J1=8.0, J2=12.0);
```

J1	<input type="range"/>	8.00
J2	<input type="range"/>	12.00
n1	<input type="range"/>	2
n2	<input type="range"/>	1
y_max	<input type="range"/>	0.40

```
[<matplotlib.lines.Line2D object at 0x10b4b7440>]
```

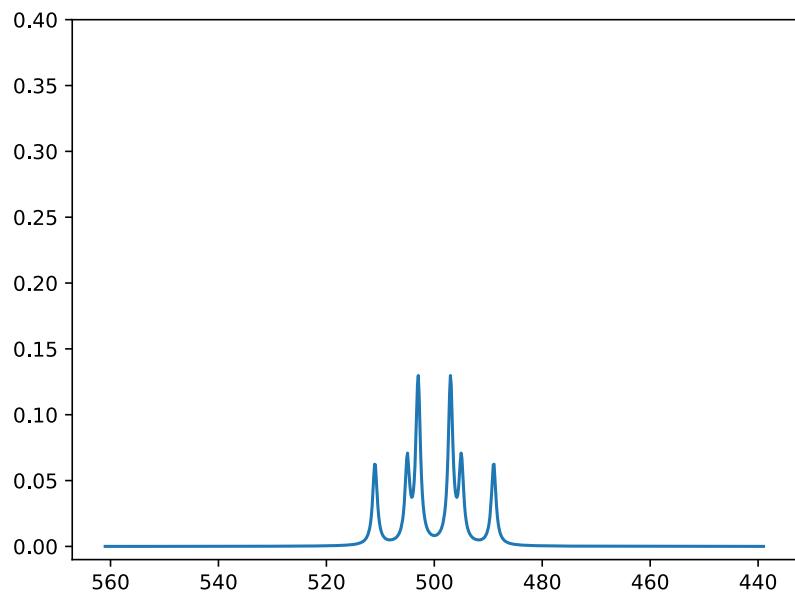


We can change the widget type to pull-down menus like below.

```
interact(plot_nmr, n1=[1,2,3], n2=[1,2,3], J1=(0, 16), J2=(0, 16));
```

J1 8
J2 6
n1 2
n2 1
y_max 0.40

```
[<matplotlib.lines.Line2D object at 0x10b4eb1d0>]
```



Appendix 1: Remote Requests

There are a number of freely available online chemical databases that can be used to build datasets such as the Chemical Abstract Services (CAS), ChEMBL, ChemSpider, RCSB Protein Data Bank, PubChem, and PubMed among others. While some databases principally support access through a web browser such as Spectral Database for Organic Compounds (SDBS), many databases support programmatically accessing the data that enables the user to automate the downloading or searching of data from databases.

This requires the database to have what is known as an Application Programming Interface (API) that allows Python to communicate with the database software. The APIs often have idiosyncratic formatting rules that must be carefully followed to ensure no errors arise. It is also important to follow the database usages rules such as how much data may be downloaded, what the data may be used for, or if users are required to register with the database. The latter is often free for academic or nonprofit use. In this example, you will learn to access the PubChem databases and build a small dataset of organic chemicals with the chemical features to describe them. PubChem does not require any registration to use it, but there is a rate limit to accessing the data which will be addressed below.

To access the database, we will use the Python [requests](#) library which allows the user to use Python to access data from remote web servers. This package is installed by default with Anaconda or can be installed using pip. It is also prudent to keep this library updated just as you would with a web browser because it makes remote requests.

PubChem requests uses a URL like your web browser with the following five components:

- prolog_URL - `https://pubchem.ncbi.nlm.nih.gov/rest/pug`
- data_input - `compound/smiles`
- identifier - `OC(C=1C=CN=C2C=CC(OC)=CC21)C3N4CCC(C3)C(C=C)C4`
- operation - `property/Volume3D`
- output - `txt`

The prolong is the base URL which allows requests to find the remote database server, the data_input indicates what information will be provided to look up a chemical compound, the identifier is the chemical identifier, the operation is what information you want out, and the output is the format of the returned information. The latter will be text in our case, but you can have PubChem return other formats such as PNG or CSV if desired. The five above pieces are concatenated with `/` separating them using the `join()` string method and are provided as an overall URL to the requests library. You could also concatenate the above strings using the `+` operator as long as you ensure there are `/` separating each component.

```
full_url = '/'.join([prolog_URL, data_input, identifier, operation, ouput])
```

Once the result is concatenated, it will look something like below.

```
https://pubchem.ncbi.nlm.nih.gov/rest/pug/compound/smiles/OC(C=1C=CN=C2C=CC(OC)=CC21)C3N4CCC(C3)C(C=C)C4
```

This URL is then fed into the `requests.get()` function like below which makes the request to the remote serve to fetch the information.

```
requests.get(full_url)
```

```
import requests
```

```
prolog_URL = "https://pubchem.ncbi.nlm.nih.gov/rest/pug"
data_input = "compound/smiles"
identifier = 'OC(C=1C=CN=C2C=CC(OC)=CC21)C3N4CCC(C3)C(C=C)C4'
operation = "property/Volume3D"
output = "txt"

full_url = '/'.join([prolog_URL, data_input, identifier, operation, output])

res = requests.get(full_url)
res
```

```
<Response [200]>
```

Once you have the result, use the `.text` method to get the regular text, and you will need to remove the last two characters.

```
res.text
```

```
'252.200000000\n'
```

```
res.text[:-1]
```

```
'252.200000000'
```

If you want to access a larger number of molecules, you will need to use a `for` loop with a list of molecular identifiers that can be swapped out in each request. It is important to note that PubChem limits requests to **no more than 5 per second**, so you will need to limit your request rate. This is relatively easy to accomplish using the `time.sleep(n)` function from the native Python `time` module where `n` is the number of seconds to pause your code. For example, every time `time.sleep(1)` is run, the function waits 1 second before the next line of code is executed. By placing this in our `for` loop, it ensures a maximum rate of requests will not be exceeded.

As an example, below we request the volume of four alcohols from PubChem and store them in a list.

```
import time

ROH_smiles = ['CC(0)C', 'C1CCCC10', 'CC(C)(C)O', 'O[C@H]1[C@H](C(C)C)CC[C@H](C)C1']

volumes = []
for ROH in ROH_smiles:
    full_url = '/'.join([prolog_URL, data_input, ROH, operation, output])
    res = requests.get(full_url)
    volumes.append(res.text[:-1])
    time.sleep(1) # pauses for 1 second

volumes
```

```
[ '54.3000000000', '84.6000000000', '66.7000000000', '134.3000000000' ]
```

Appendix 2: Visualizing Atomic Orbitals

Contents

- Radial Wavefunctions
- Angular Wavefunctions
- Complete Wavefunction

Note

This appendix assumes a future version of SymPy for the `Z_lm()` function. This function has been temporarily defined in a code cell below to provide this feature until the next SymPy release.

The visualization of atomic orbitals and orbital information is important enough topics in chemistry to warrant specific attention. This appendix focuses on different methods of visualizing various aspects of atomic orbitals and tools to assist in this task. This content is not included in the chapter on plotting with matplotlib because this appendix heavily utilizes various libraries such as SymPy, interact, and NumPy not yet introduced before Chapter 03. While this appendix is written to be standalone as much as possible, knowledge of [matplotlib](#), including [surface plots](#), will be helpful along with [NumPy](#) and [SymPy](#) basics.

Atomic orbitals are described by a wavefunction, $\Psi(n, l, m)$, which is the product of the radial wavefunction, $R(n, l)$, and the angular wavefunction, $Y(l, m)$. Each atomic orbital has a different wavefunction Ψ , but they sometimes share common radial wavefunctions.

$$\Psi(n, l, m) = R(n, l)Y(l, m)$$

The **radial wavefunction** depends upon the principal (n) and angular (l) quantum numbers and provides information about the wavefunction or electron probability at various distances from the nucleus. The radial wavefunction is independent of the direction. The **angular wavefunction** describes the *direction* of the orbital with respect to the spherical coordinate angles and depends upon the angular (l) and magnetic (m or m_l) quantum numbers. We will first visualize the radial and angular components individually before combining them into a more complete picture of atomic orbitals.

We will use NumPy and matplotlib heavily in this chapter, and we will make heavy use of the SymPy library for convenient functions in its [hydrogen module](#). These are all imported below.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

import sympy
from sympy.physics.hydrogen import R_nl, Psi_nl, Z_lm
```

```
# delete this cell and replace with actual Z_lm after next SymPy release
from sympy.functions.special.spherical_harmonics import Znm
def Z_lm(l, m, phi, theta):
    return Znm(l, m, theta, phi).expand(func=True)
```

Radial Wavefunctions

Because the radial wavefunctions are independent of direction, they can be represented effectively on a simple 2D plot. The toughest part is coding the equations for every combination of n and l . The good news is that the SymPy library includes a function, `R_nl()`, in the [Hydrogen Wavefunction](#) (`sympy.physics.hydrogen`) module that provides this functionality. This function takes the principle quantum number (n), angular quantum number (l), radius in Bohrs (r), and atomic number (Z). A Bohr equals about 52.9 pm.

```
R_nl(n, l, m, r, Z=1)
```

We can evaluate the function for any hydrogen-like atomic orbitals such as the 3p orbitals ($n = 3$ and $l = 1$) at 4.0 Bohrs.

```
R_nl(3, 1, 4.0, Z=1)
```

$$0.0173561901639985\sqrt{6}$$

SymPy prefers to return results in exact form, so it includes $\sqrt{6}$ in this particular result. To get a float answer, use the `evalf()` method.

```
R_nl(3, 1, 4.0, Z=1).evalf()
```

$$0.0425138097805085$$

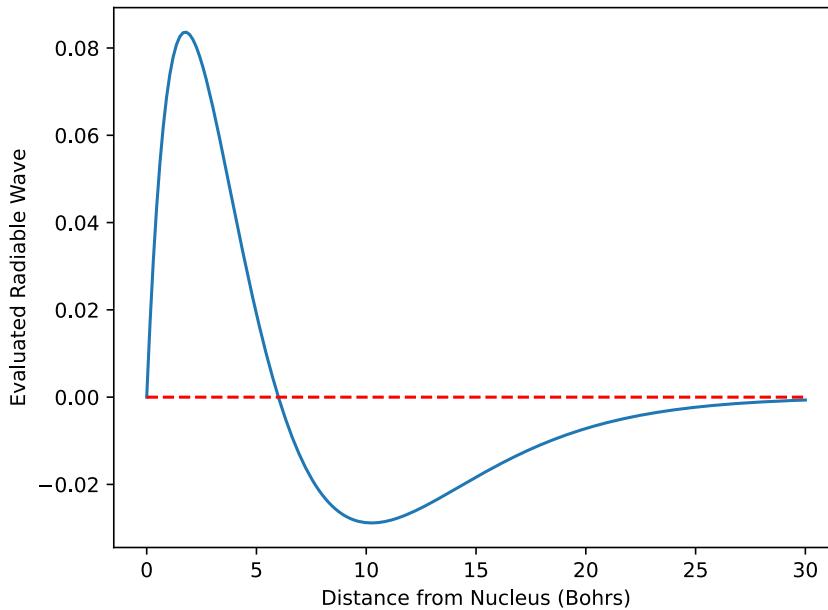
It might now be interesting to evaluate this radial function at a range of distances and plot them. This function does not support taking multiple radii, so you have two options below.

1. Iterate through a list or array of radii and evaluate this function one radius at a time.
2. Convert the `R_nl()` function to a function that can accept an array using the `lambdify()` method.

Both approaches are demonstrated below.

```
# first approach - iterate through iterable
radii = np.linspace(0, 30, 200)
R_eval = [R_nl(3, 1, r, Z=1) for r in radii]

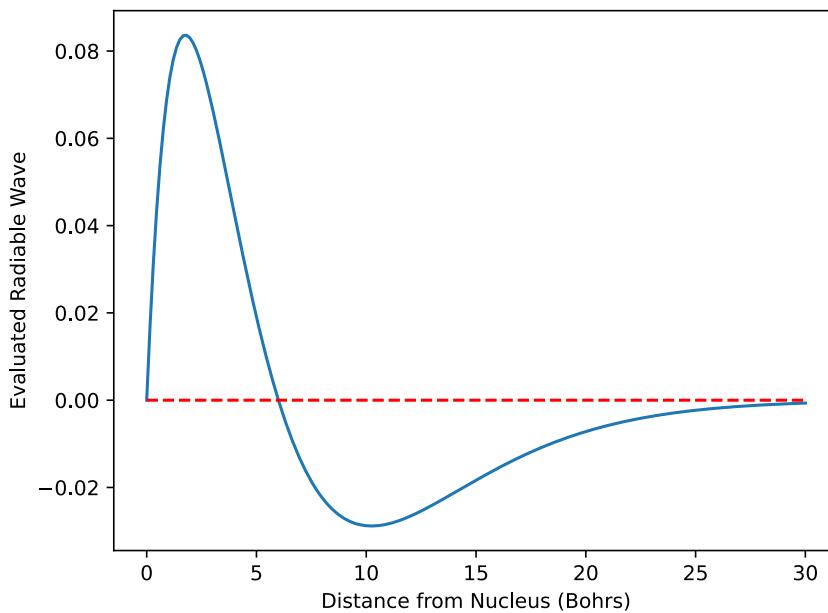
plt.plot(radii, R_eval)
plt.hlines(0, 0, 30, colors='r', linestyles='dashed')
plt.xlabel('Distance from Nucleus (Bohrs)')
plt.ylabel('Evaluated Radial Wave');
```



```
# second approach - lambdify
r = sympy.symbols('r') # create SymPy symbol

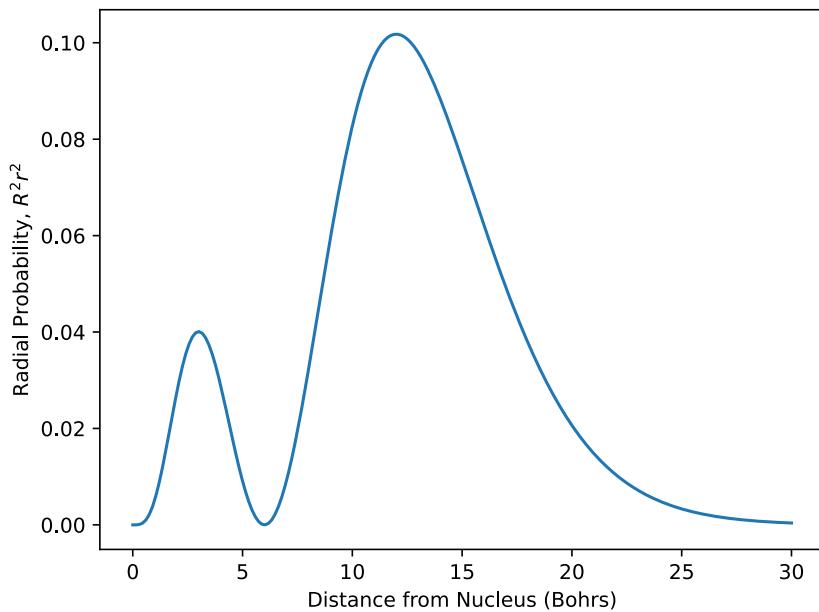
# create a numpy compatible function using lambdify
R_3p = sympy.lambdify(r, R_nl(3, 1, r, Z=1), modules='numpy')
radii = np.linspace(0, 30, 200)

plt.plot(radii, R_3p(radii))
plt.hlines(0, 0, 30, colors='r', linestyles='dashed')
plt.xlabel('Distance from Nucleus (Bohrs)')
plt.ylabel('Evaluated Radiable Wave');
```



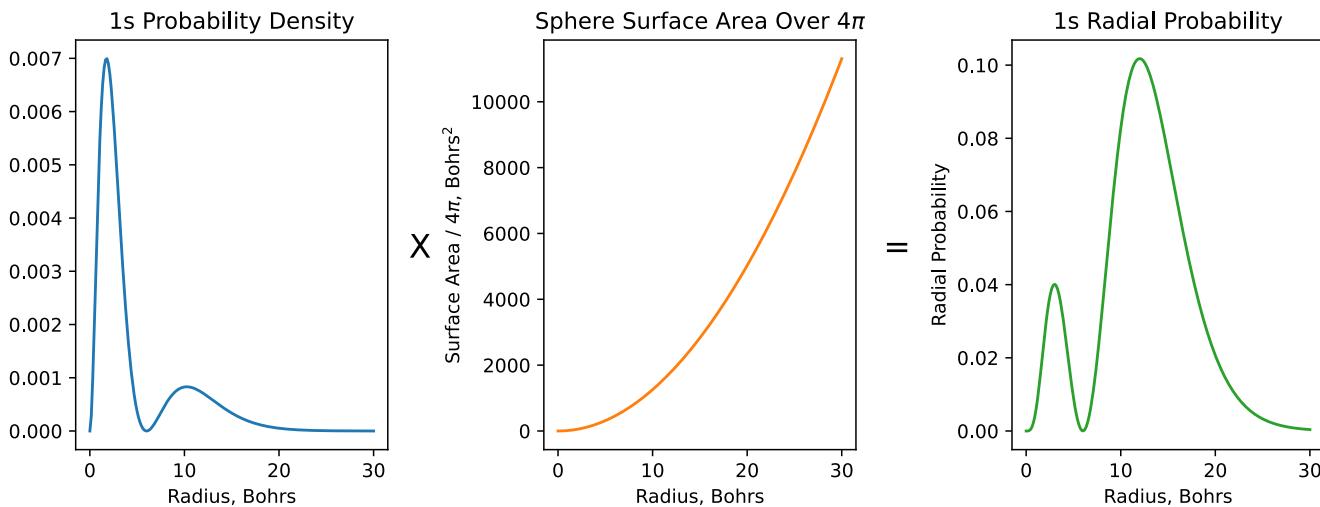
The electron **probability density** can be found by calculating R^2 where R is the radial wavefunction, and the **radial probability** is R^2r^2 where r is the distance from the nucleus.

```
plt.plot(radii, R_3p(radii)**2 * radii**2)
plt.xlabel('Distance from Nucleus (Bohrs)')
plt.ylabel('Radial Probability, $R^2r^2$');
```



The reason we multiply the probability density by the square of the radial wavefunction, r^2 , is to account for the greater surface area of a sphere ($A_{sphere} = 4\pi r^2$) the larger the radius. We are effectively carrying out the calculation depicted below. We divide the sphere surface area by 4π to normalize the integration making the probability over all space total to one.

[▶ Show code cell source](#)



One of the uses of these radial plots is to compare the radial probability of multiple different orbitals on the same axes like below for the fourth row of the periodic table. This can be used, for example, to discuss the valence electron configurations of Cr and Cu.

```

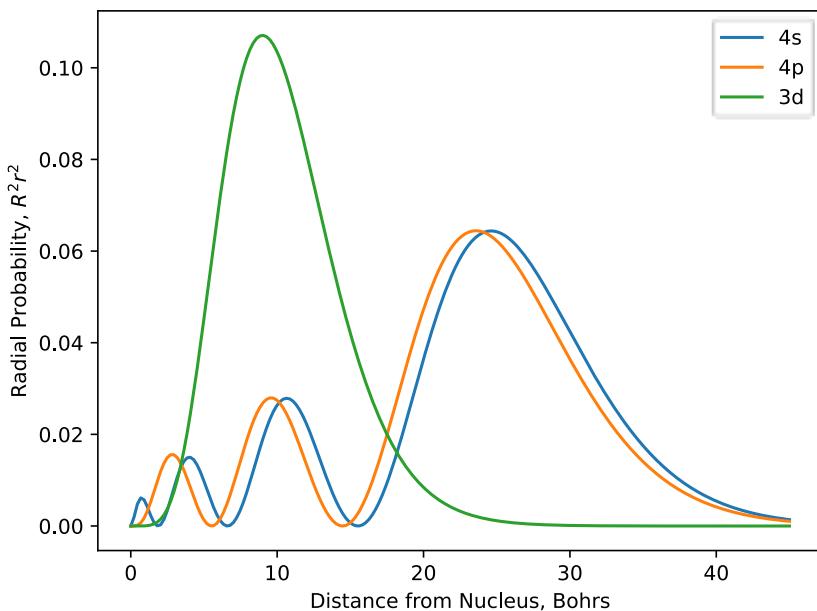
r = sympy.symbols('r') # create SymPy symbol

# create a numpy compatible function using lambdify
R_3s = sympy.lambdify(r, R_nl(4, 0, r, Z=1), modules='numpy')
R_3p = sympy.lambdify(r, R_nl(4, 1, r, Z=1), modules='numpy')
R_3d = sympy.lambdify(r, R_nl(3, 2, r, Z=1), modules='numpy')

radii = np.linspace(0, 45, 200)

plt.plot(radii, R_3s(radii)**2 * radii**2, label='4s')
plt.plot(radii, R_3p(radii)**2 * radii**2, label='4p')
plt.plot(radii, R_3d(radii)**2 * radii**2, label='3d')
plt.xlabel('Distance from Nucleus, Bohrs')
plt.ylabel('Radial Probability, $R^2r^2$');
#plt.ylim(0, 0.2)
plt.legend();

```



The probability R^2r^2 can be integrated using the `sympy.integrate()` function which accepts the function or mathematical expression to be integrated and a tuple that contains the variable, the min, and the max values.

```
sympy.integrate(f(x), (x, min, max))
```

For example, we can integrate the `R_nl()` for the 2s orbital from 0 to 3.0 Bohrs like below.

0.473330547984585

Let's test that the radial probability is normalized by integrating from zero to infinity.

```
sympy.integrate(R_nl(2, 0, r, Z=1)**2 * r**2, (r, 0, sympy.oo)).evalf()
```

1.0

Angular Wavefunctions

The other component of Ψ is the angular wavefunctions which provides directional information about an orbital. The angular equations can be coded by hand or we can also use the `Y_lm()` or `Z_lm()` spherical harmonics wavefunctions from `sympy.physics.hydrogen` to assist us. The difference between these two functions is that `Y_lm()` may return a complex expression whereas `Z_lm()` will return the real-valued angular wavefunction. Because our goal is to visualize the wavefunctions, we will restrict ourselves to the latter here. The angular wavefunction provides information in all directions, so we will plot this information in 3D.

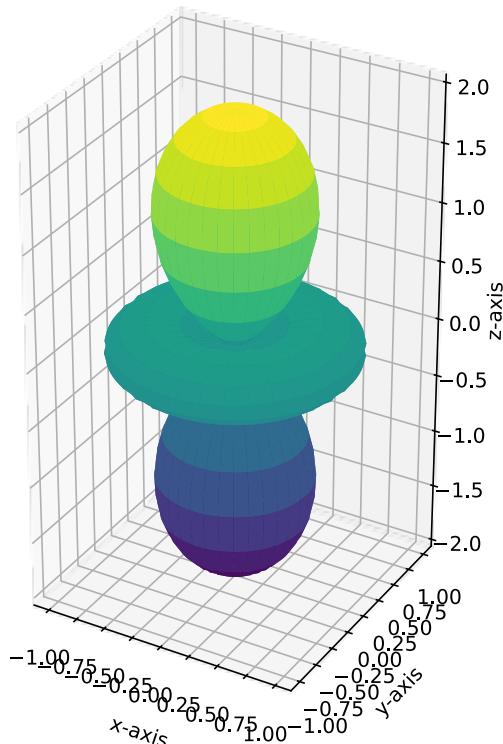
There are multiple conventions for spherical coordinates. We will use the SciPy/Sympy convention of using theta (θ) for the azimuthal (i.e., direction on xy-plane) and phi (ϕ) as the polar angle (i.e., angle from the positive z-axis) for plotting the angular wavefunctions. Below, we plot the d_{z^2} orbital by coding the [angular wavefunction expression](#) by hand.

```
# generate mesh grid of theta and phi values
theta, phi = np.meshgrid(np.linspace(0, np.pi, 51),
                        np.linspace(0, 2 * np.pi, 101))

# convert angles to xyz values of a sphere, r = 1
x = np.sin(theta) * np.sin(phi)
y = np.sin(theta) * np.cos(phi)
z = np.cos(theta)

# multiply xyz values by angular wavefunction
dz2 = np.sqrt((5 / 16) * np.pi) * (3 * np.cos(theta)**2 - 1)
X, Y, Z = x * dz2, y * dz2, z * dz2

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')
ax.set_aspect('equal') # sets aspect ratio to equal
```



Alternatively, we can use the `Z_lm()` function from `sympy.physics.hydrogen` to generate the angular wavefunction based on the angular and magnetic quantum numbers.

```
Z_lm(l, m, phi, theta)
```

SymPy functions cannot calculate wavefunctions for an array of angles like NumPy functions can, but fortunately SymPy functions can be converted to NumPy function using the `lambdify()` method. Just provide the `lambdify()` method with a collection of argument variables for the wavefunction as SymPy symbols, the wavefunction, and `modules='numpy'`, and it returns a new function.

```
# from sympy.physics import Z_lm

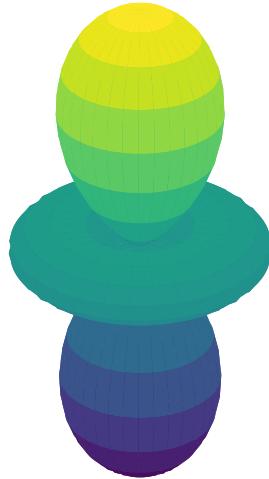
theta, phi = np.meshgrid(np.linspace(0, np.pi, 51),
                         np.linspace(0, 2*np.pi, 101))

x = np.sin(theta) * np.sin(phi)
y = np.sin(theta) * np.cos(phi)
z = np.cos(theta)

# create a numpy function
p, t = sympy.symbols('p t')
f = sympy.lambdify((p, t), Z_lm(2, 0, p, t), modules='numpy')

# multiply xyz values by wave angular wavefunction
f_pt = f(phi, theta)
X, Y, Z = x * f_pt, y * f_pt, z * f_pt

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_aspect('equal') # sets aspect ratio to equal
ax.set_axis_off() # turns off axes and background
```



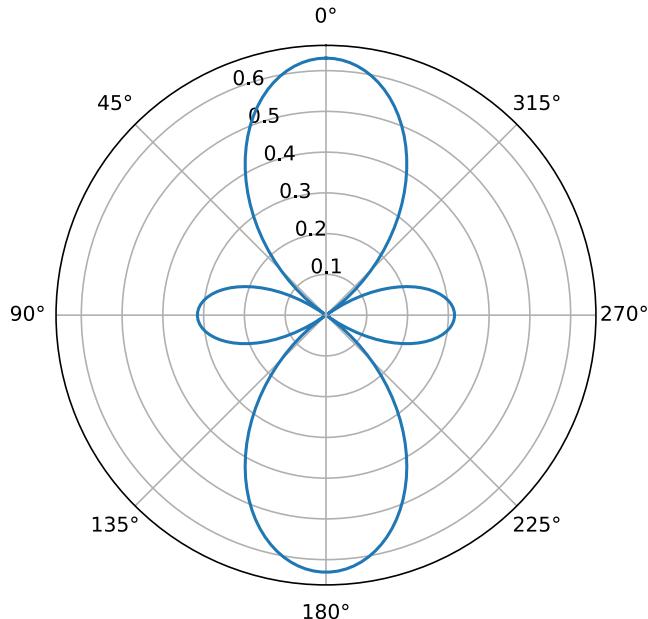
We can also visualize the angular component of wavefunctions in 2D using a [polar plot](#), but we can only visualize one angle at a time. Below we will visualize theta and leave phi fixed. Because we are only visualizing in 2D and not sweeping around the phi angles, we need to make theta go from $0 \rightarrow 2\pi$.

```

l, m = 2, 0
azimuth, polar = sympy.symbols('azimuth polar')
f = sympy.lambdify((polar, azimuth), Z_lm(l, m, polar, azimuth), modules='numpy')

th = np.linspace(0, 2 * np.pi, 200)
fig = plt.figure()
ax = fig.add_subplot(111, polar=True)
ax.plot(th, np.abs(f(0, th)))
# orient 0 degrees to up/north
ax.set_theta_zero_location('N');

```

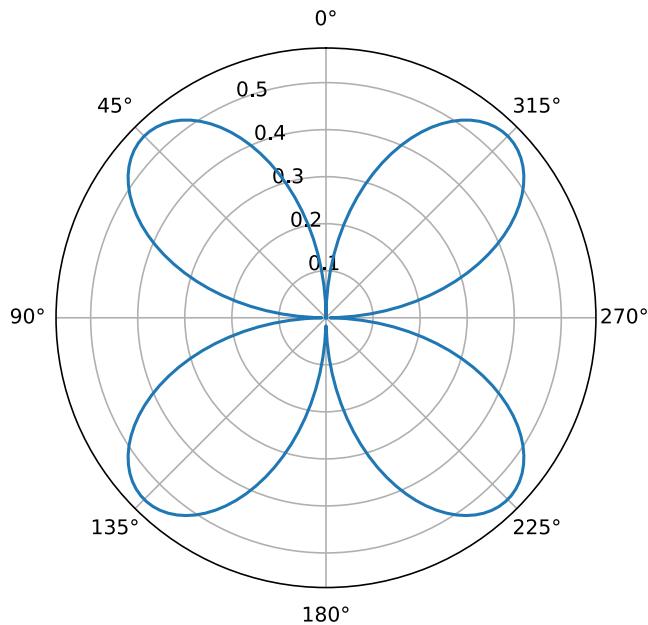


```

l, m = 2, 1
azimuth, polar = sympy.symbols('azimuth polar')
f = sympy.lambdify((polar, azimuth), Z_lm(l, m, polar, azimuth), modules='numpy')

th = np.linspace(0, 2 * np.pi, 200)
fig = plt.figure()
ax = fig.add_subplot(111, polar=True)
ax.plot(th, np.abs(f(0, th)))
# orient 0 degrees to up/north
ax.set_theta_zero_location('N');

```

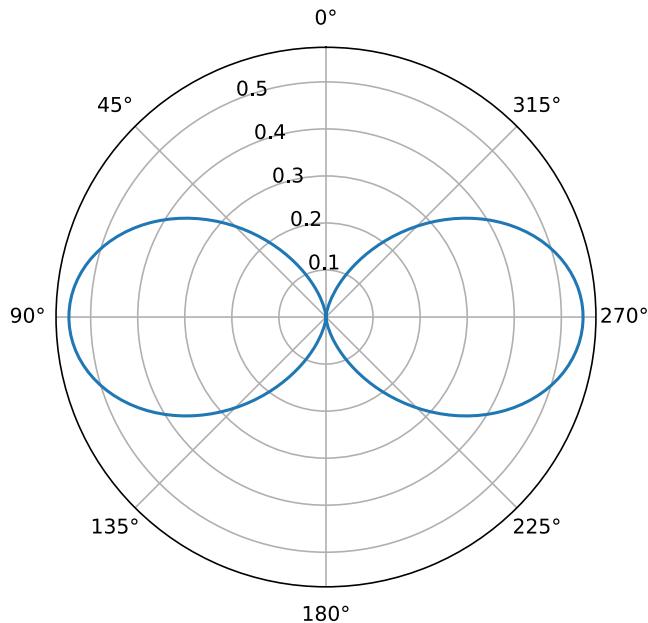


```

l, m = 2, 2
azimuth, polar = sympy.symbols('azimuth polar')
f = sympy.lambdify((polar, azimuth), Z_lm(l, m, polar, azimuth), modules='numpy')

th = np.linspace(0, 2 * np.pi, 200)
fig = plt.figure()
ax = fig.add_subplot(111, polar=True)
ax.plot(th, np.abs(f(0, th)))
# orient 0 degrees to up/north
ax.set_theta_zero_location('N');

```



The last orbital image is a d -orbital viewed from the side.

Complete Wavefunction

Now we will visualize both angular and radial components together (Ψ) which is again the product of the radial, $R(n, l)$

and angular, $Y(l, m)$ wavefunctions.

$$\Psi(n, l, m) = R(n, l)Y(l, m)$$

To obtain the entire wavefunction, Ψ , we can either multiply the radial and angular wavefunctions from the previous sections or use the SymPy `Psi_nlm()` function which makes this task a little more convenient. Orbitals have no edge, so there are multiple ways of representing orbitals including contour plots, isosurfaces, 90% surface plots, scatter plots, and translucent 3D plots. The scatter and contour plot methods are demonstrated below. We will need the probability density, P , of the atomic orbital which is proportional to the product of a wavefunction, Ψ , and its complex conjugate, Ψ^* or the square of the absolute value of a wavefunction.

$$P = \Psi^*\Psi = |\Psi|^2$$

First, let's take a look at the `Psi_nlm()` function which operates similarly to the other SymPy wavefunctions above. Below, we integrate it over all space returning 1 which tells us that this function is normalized when we include $r^2\sin(\theta)$.

```
azimuth, polar, r = sympy.symbols('azimuth polar r')
wf = Psi_nlm(3, 1, 0, r, azimuth, polar)

# integrate normalized wavefunction over all area
sympy.integrate(wf**2 * r**2 * sympy.sin(polar),
                 (r, 0, sympy.oo),
                 (azimuth, 0, 2 * sympy.pi),
                 (polar, 0, sympy.pi))
```

1

Now let's visualize an orbital using a scatter plot. We will use a strategy previously reported in [J. Chem. Educ., 1990, 67, 42-44](#) which includes the following steps.

1. Use a random number generator to produce a series of r , θ , and ϕ values or just r and θ values depending upon dimensions
2. Use the values above to calculate the xyz or yz values
3. Use the above radius and angles to calculate probabilities using the wavefunction
4. Normalize the probabilities by dividing by the maximum probability value across all the data points
5. If each normalized probability is above a random value from $0 \rightarrow 1$, it gets included in the scatter plot

```

# 2p orbital - 3D simulation

# create wavefunction as python function
r, azimuth, polar = sympy.symbols('r azimuth polar')
wf_sym = Psi_nlm(2, 1, 0, r, azimuth, polar)
wf = sympy.lambdify((r, azimuth, polar), wf_sym, modules='numpy')

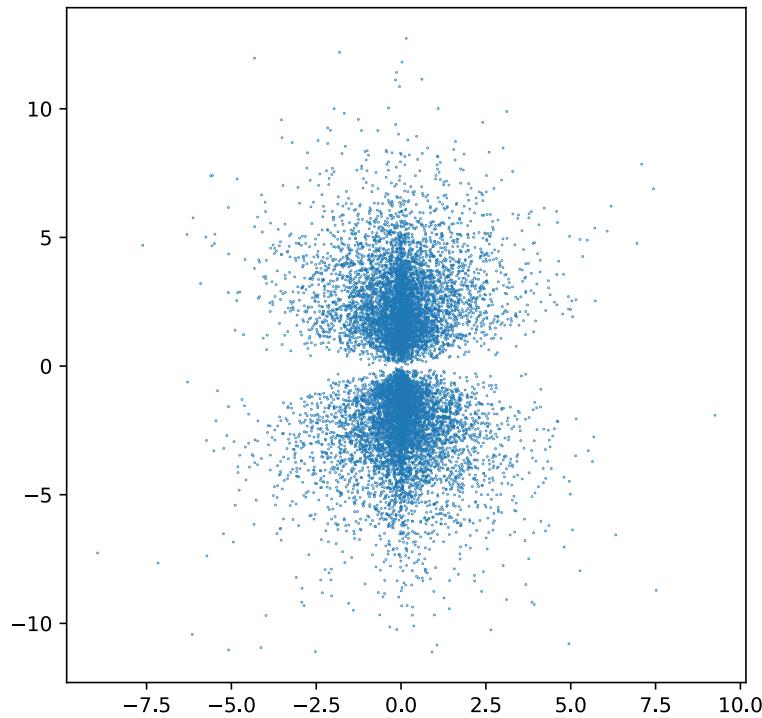
# generate random coordinates
rng = np.random.default_rng(seed=21)
n_points = 100000
r = 15 * rng.random(size=(n_points))
polar = np.pi * rng.random(size=(n_points))
azimuth = 2 * np.pi * rng.random(size=(n_points))

x = r * np.sin(polar) * np.sin(azimuth)
y = r * np.sin(polar) * np.cos(azimuth)
z = r * np.cos(polar)

# normalize and create mask
prob_dens = np.abs(wf(r, azimuth, polar))**2
norm_prob = prob_dens / prob_dens.max()
mask = norm_prob > rng.random(n_points)

#plt.plot(y[mask], z[mask], ',');
fig = plt.figure(figsize = (6,6))
ax = fig.add_subplot(1, 1, 1)
ax.scatter(y[mask], z[mask], s=0.1);

```



The plot above makes the shape of the orbital look like the orbital lobes are almost conical, which is not what we typically see in accurate orbital shapes. This is an effect of there being more data points visualized along the vertical axis due to the orbital being thicker there. If we instead reduce the simulation to 2D (i.e., only yz plane), like below, the orbital lobes appear rounder because we are visualizing a slice through the middle of the orbital.

```

# 2p orbital - 2D simulation

# create wavefunction as python function
r, polar = sympy.symbols('r polar')
wf_sym = Psi_nlm(2, 1, 0, r, 0, polar)
wf = sympy.lambdify((r, polar), wf_sym, modules='numpy')

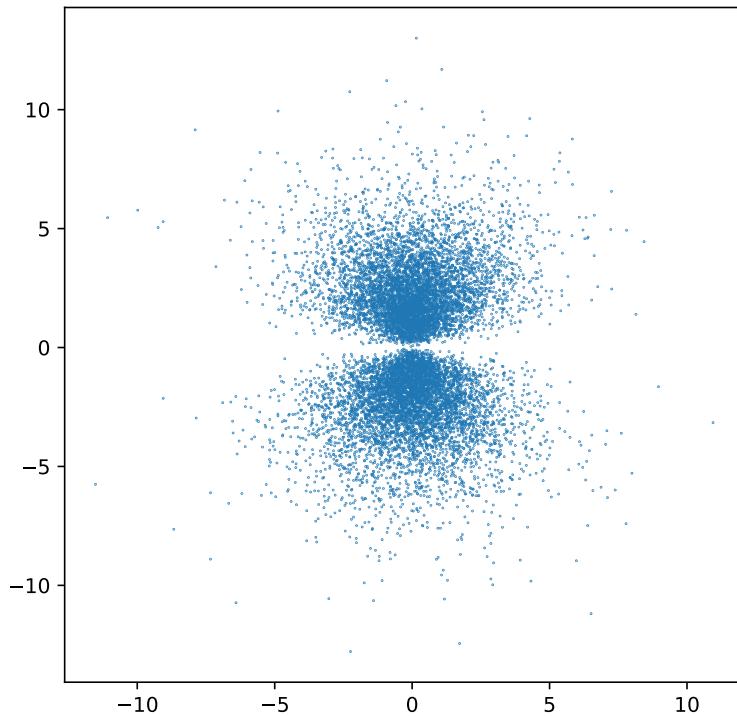
# generate random coordinates
rng = np.random.default_rng(seed=21)
n_points = 100000
r = 15 * rng.random(size=(n_points))
polar = 2 * np.pi * rng.random(size=(n_points))

x = r * np.sin(polar) * np.sin(0)
y = r * np.sin(polar) * np.cos(0)
z = r * np.cos(polar)

# normalize and create mask
prob_dens = np.abs(wf(r, polar))**2
norm_prob = prob_dens / prob_dens.max()
mask = norm_prob > rng.random(n_points)

fig = plt.figure(figsize = (6, 6))
ax = fig.add_subplot(1, 1, 1)
ax.scatter(y[mask], z[mask], s=0.1);

```



We can visualize larger orbitals to see more nodes such as in the 3p and 3s orbitals below. We can also color the points based on the sign of the wavefunction before calculating the probability. In the examples below, the color only represents the sign of the wavefunction and not the magnitude of the value.

```

# 3p orbital

# create wavefunction as python function
r, polar = sympy.symbols('r, polar')
wf_sym = Psi_nlm(3, 1, 0, r, 0, polar)
wf = sympy.lambdify((r, polar), wf_sym, modules='numpy')

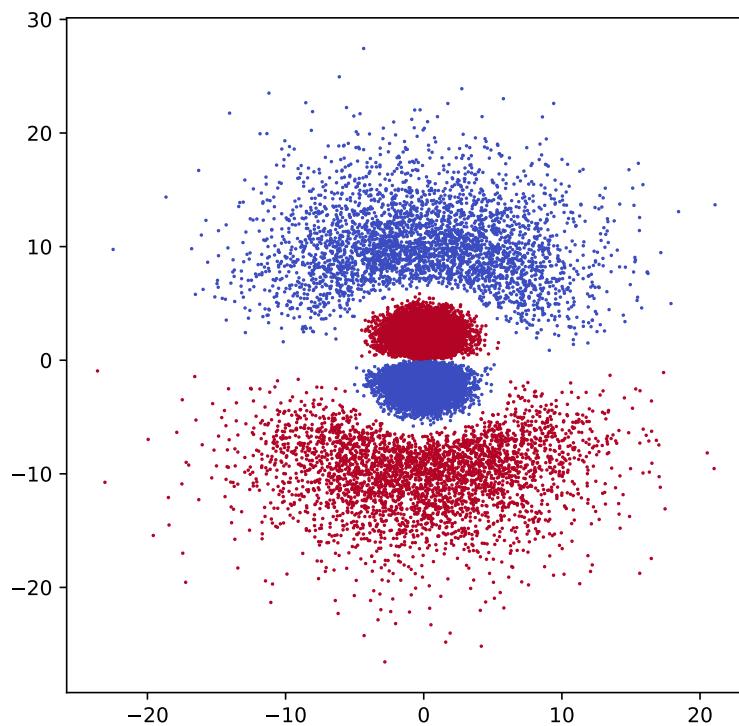
# generate random coordinates
rng = np.random.default_rng(seed=21)
n_points = 500000
r = 30 * rng.random(size=(n_points))
polar = 2 * np.pi * rng.random(size=(n_points))

x = r * np.sin(polar) * np.sin(0)
y = r * np.sin(polar) * np.cos(0)
z = r * np.cos(polar)

# normalize and create mask
prob_dens = np.abs(wf(r, polar))**2
norm_prob = prob_dens / prob_dens.max()
mask = norm_prob > rng.random(n_points)

#plt.plot(x[mask], y[mask],
fig = plt.figure(figsize = (6, 6))
ax = fig.add_subplot(1, 1, 1)
is_pos = wf(r, polar)[mask] > 0 # test if wavefunc is positive
ax.scatter(y[mask], z[mask], s=0.5, c=is_pos, cmap='coolwarm');

```



```

# 3s orbital

# create wavefunction as python function
r, polar = sympy.symbols('r, polar')
wf_sym = Psi_nlm(3, 0, 0, r, 0, polar)
wf = sympy.lambdify((r, polar), wf_sym, modules='numpy')

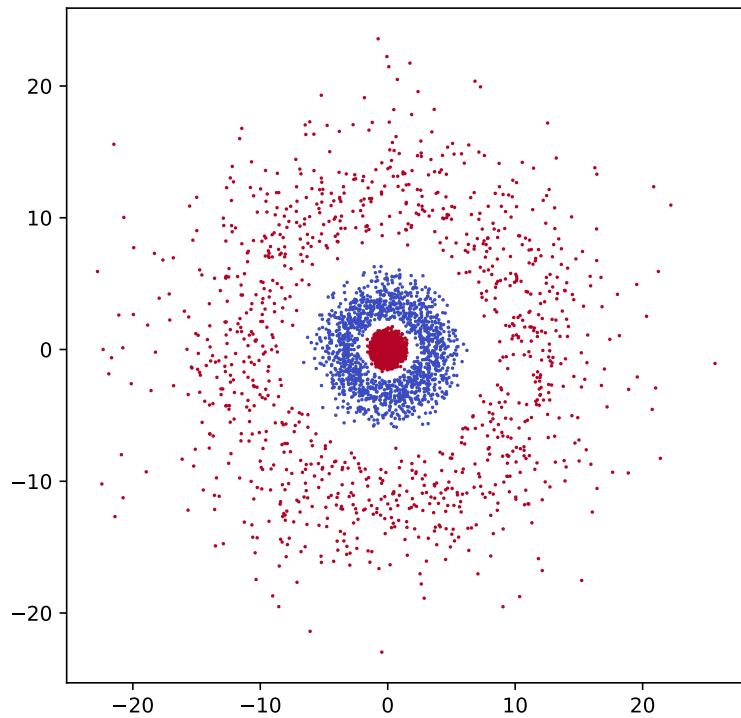
# generate random coordinates
rng = np.random.default_rng(seed=21)
n_points = 1000000
r = 30 * rng.random(size=(n_points))
polar = 2 * np.pi * rng.random(size=(n_points))

x = r * np.sin(polar) * np.sin(0)
y = r * np.sin(polar) * np.cos(0)
z = r * np.cos(polar)

# normalize and create mask
prob_dens = np.abs(wf(r, polar))**2
norm_prob = prob_dens / prob_dens.max()
mask = norm_prob > rng.random(n_points)

fig = plt.figure(figsize = (6, 6))
ax = fig.add_subplot(1, 1, 1)
is_pos = wf(r, polar)[mask] > 0 # test if wavefunc is positive
ax.scatter(y[mask], z[mask], s=0.5, c=is_pos, cmap='coolwarm');

```



A second way to visualize orbitals is through a contour plot. Here we calculate the probability in a mesh of locations and provide the `plt.contour()` function with the locations and probabilities.

```

Y, Z = np.meshgrid(np.linspace(-20, 20, 200),
                   np.linspace(-20, 20, 200))

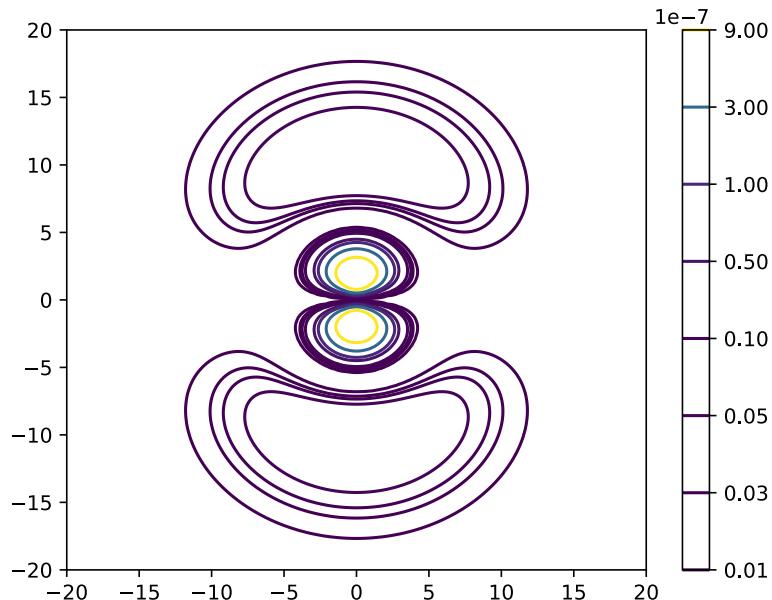
# create wavefunction as python function
r, polar = sympy.symbols('r polar')
wf = Psi_nlm(3, 1, 0, r, 0, polar)
f = sympy.lambdify((r, polar), wf * sympy.conjugate(wf), modules='numpy')
polar = np.arctan(Z / Y)
r = np.sqrt(Y**2 + Z**2)

# calculate probability
prob = np.abs(f(r, polar))**2

plt.contour(Z, Y, prob, levels=[1e-9, 3e-9, 5e-9, 1e-8, 5e-8, 1e-7, 3e-7, 9e-7])
plt.colorbar()

```

<matplotlib.colorbar.Colorbar at 0x10e10b5f0>



```

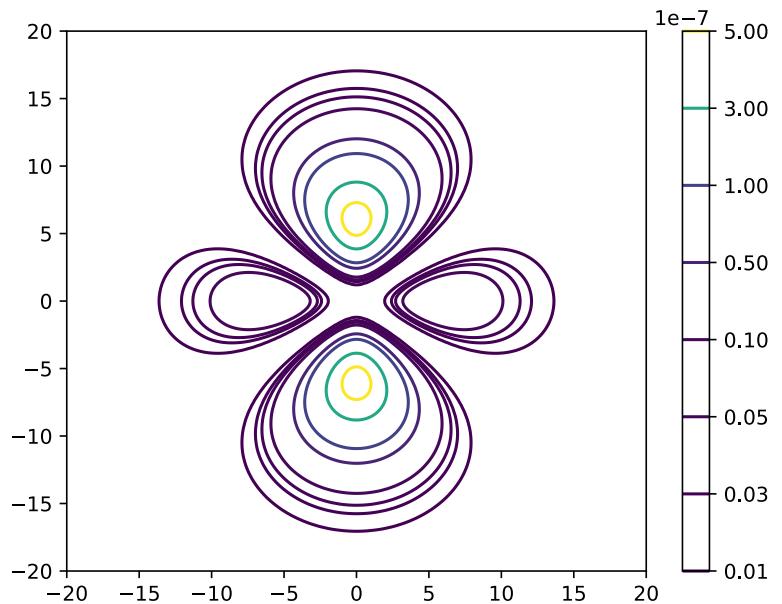
Y, Z = np.meshgrid(np.linspace(-20, 20, 200),
                   np.linspace(-20, 20, 200))

# create wavefunction as python function
r, polar = sympy.symbols('r polar')
wf = Psi_nlm(3, 2, 0, r, 0, polar)
f = sympy.lambdify((r, polar), wf * sympy.conjugate(wf), modules='numpy')
polar = np.arctan(Z / Y)
r = np.sqrt(Y**2 + Z**2)

# calculate probability
prob = np.abs(f(r, polar))**2

plt.contour(Z, Y, prob, levels=[1e-9, 3e-9, 5e-9, 1e-8, 5e-8, 1e-7, 3e-7, 5e-7])
plt.colorbar();

```



Appendix 3: Uncertainty Propagation

Contents

- **Uncertainties** Package
- Simulating Uncertainties
- Further Reading

Uncertainty occurs in any scientific measurement and is often represented as the standard deviations, σ , of measurements or the 95% confidence interval, 95% CI. When performing calculations containing values with uncertainty, the uncertainty needs to be propagated through the calculations, which is a tedious and error-prone task when done by hand. This appendix demonstrates how to use the Python `uncertainties` package to remove most of the pain from uncertainty propagation along with simulating uncertainty using a random number generator.

Uncertainties Package

As of this writing, the `uncertainties` package can be [installed using pip](#). We will then import a couple key functions, `ufloat()` and `ufloat_fromstr()`, along with the `umath` module which brings a range of math functions (e.g., log and sin). We will also import NumPy and matplotlib to use in the simulation section.

```
from uncertainties import ufloat, ufloat_fromstr
from uncertainties import umath

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

Uncertainties Variable

Basic mathematical operations with the `uncertainties` package center around the `uncertainties` variable object. This is created using the `ufloat()` function which accepts two important values - the first is the nominal value and the second is the standard deviation.

```
ufloat(nominal_value, std_dev)
```

For example, let's say we have a value of 18.66 with a standard deviation of 0.03.

```
val = ufloat(18.32, 0.03)
```

We can access the nominal value or the standard deviation by themselves using the `nominal_value` or `std_dev` methods, respectively.

```
val.nominal_value
```

```
18.32
```

```
val.std_dev
```

```
0.03
```

Values from Strings

If you are calculating uncertainties taken from a text problem, the `uncertainties` package provides a convenience function `ufloat_fromstr()` that allows you to copy-and-paste in values and their uncertainties all together. Below are acceptable formats.

```
ufloat_fromstr('0.011 ± 0.002')
```

```
0.011+-0.002
```

```
ufloat_fromstr('0.172807(0.000008)')
```

```
0.172807+-8e-06
```

```
ufloat_fromstr('0.172807 +/- 0.000008')
```

```
0.172807+-8e-06
```

```
ufloat_fromstr('0.172')
```

```
0.172+-0.001
```

The last one did not include an uncertainty, so the uncertainty was interpreted to be ± 1 of the least significant decimal place.

Simple Calculations

Beyond this, we just need to carry out our mathematical operations. For example, let's say we want to calculate the molar absorptivity constant using Beer's law, $A = \epsilon b C$, where A is absorbance, ϵ is the molar absorptivity constant, b is the path length in cm, and C is concentration in molarity. If the $A = 0.3822 \pm 0.0003$, $b = 1.00 \pm 0.01$ cm, and $C = 0.0017 \pm 0.0001$ M, we can calculate the molar absorptivity constant like below.

```
A = ufloat(0.3822, 0.0003)
b = ufloat(1.00, 0.01)
C = ufloat(0.0017, 0.0001)

E = A / (b * C)
E
```

```
224.8235294117647+/-13.415813085736838
```

This results in $225 \pm 13 \text{ cm}^{-1}\text{M}^{-1}$.

If we multiply an uncertainties variable object by a regular `int` or `float()`, the `int` or `float()` is treated as having no uncertainty, so the uncertainty of the other value scales linearly with the nominal value. In the example below, both values triple.

```
3 * b
```

```
3.0+/-0.03
```

The `umath` module provides special mathematical functions like square root or sine. For example, if we want to calculate the pH of a solution with an $[\text{H}_3\text{O}^+] = 6.33 \times 10^{-6} \pm 3 \times 10^{-7} \text{ M}$, or $(6.33 \pm 0.3) \times 10^{-6} \text{ M}$, we can carry out this calculation below which gives us a $\text{pH} = 5.199 \pm 0.021$.

```
H3O = ufloat(6.33e-6, 3e-7)
-umath.log10(H3O)
```

```
5.198596289982645+/-0.02058267686745269
```

Correlated Values

The above calculations assume that all the values in the calculation have no correlation with each other, which is not always the case. When correlation occurs, this adds an extra layer of complexity to the error propagation calculations. The `uncertainties` package recognizes some correlation automatically and handles it for you such as below when subtracting a value by itself.

```
b - b
```

```
0.0+/-0
```

If a new value is calculated using `uncertainties`, the package automatically recognizes and factors in the correlation into future calculations. We can get a sense of the correlation using the `covariance_matrix()` or `correlation_matrix()` functions. For example, we can input variables from the above Beer's law problem to see the covariance and correlation matrices.

```
from uncertainties import covariance_matrix, correlation_matrix
```

```
covariance_matrix([b, C, E])
```

```
[[0.0001, 0.0, -0.022482352941176467],  
 [0.0, 1e-08, -0.0013224913494809688],  
 [-0.022482352941176467, -0.0013224913494809688, 179.98404075142776]]
```

```
correlation_matrix([b, C, E])
```

```
array([[ 1.          ,  0.          , -0.16758099],  
       [ 0.          ,  1.          , -0.98577055],  
       [-0.16758099, -0.98577055,  1.          ]])
```

When correlated values are derived outside of `uncertainties` such as in linear regressions, the user needs to provide correlation information when creating uncertainties variable objects. This is done with the `correlated_values()` function which requires the nominal values and a covariance matrix as the two required positional arguments.

Alternatively, you can use the related `correlated_values_norm()` function which instead accepts the nominal values and the correlation matrix.

```
correlated_values(nominal_values, covariance_matrix)  
correlated_values_norm(nominal_values, correlation_matrix)
```

The good news is that NumPy and SciPy functions can also return the covariance matrix along with the best fit parameters. For example, `scipy.optimize.curve_fit()` automatically returns `pcov` which is the "estimated approximate" covariance matrix and `numpy.polyfit(cov=True)` returns the scaled covariance matrix as a second returned item when `cov=True`.

Below, we will demonstrate this using a calibration curve for absorbance and concentration data using the `np.polyfit()` function introduced in [section 6.4.1](#).

```
A_data = np.array([0.104, 0.197, 0.361, 0.706, 0.970])  
C_data = np.array([1.0e-06, 2.0e-06, 4.0e-06, 8.0e-06, 1.2e-05])  
  
fit, cov = np.polyfit(C_data, A_data, deg=1, cov=True)  
fit
```

```
array([7.93846154e+04, 3.89230769e-02])
```

```
cov
```

```
array([[ 6.86575444e+06, -3.70750740e+01],  
       [-3.70750740e+01,  3.14451553e-04]])
```

The fit returns the slope and y -intercept values along with the covariance matrix. We can then create our uncertainties variable in `uncertainties` by providing both to the `correlated_values()` function.

```
from uncertainties import correlated_values  
  
m, b = correlated_values(fit, cov)  
m
```

```
79384.61538461538+/-2620.258467760363
```

```
b
```

```
0.03892307692307681+/-0.017732781881431937
```

If we then decide to calculate the concentration for an absorbance of 0.501, for example, `uncertainties` will factor in uncertainty and correlation automatically like below.

```
(0.501 - b) / m
```

```
5.8207364341085285e-06+/-1.3535749157410873e-07
```

If we were to carry about the above calculation *without* factoring in correlation, it would look like below. While the value itself does not change, the uncertainty is overestimated.

```
m_uncorr = ufloat_fromstr('79384.6153846154+/-2620.258467760346')  
b_uncorr = ufloat_fromstr('0.038923076923077046+/-0.01773278188143182')  
  
(0.501 - b_uncorr) / m_uncorr
```

```
5.820736434108524e-06+/-2.9463551943621185e-07
```

Simulating Uncertainties

We can also simulate uncertainties using Monte Carlo simulations as demonstrated below. Let's say we want to calculate the molar absorptivity constant using the same nominal and standard deviation values as above. Using a random number generator, we can generate values for A, l, and C with the given standard deviations using the `normal(nominal_value, std_dev)` function from the `numpy.random module`. We then carry out the calculation with all of these values. The molar absorptivity is the average of these values with an uncertainty calculated from the standard deviation of these calculated values.

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

```

A_nom, A_sig = 0.3822, 0.0003
l_nom, l_sig = 1.00, 0.01
C_nom, C_sig = 0.0017, 0.0001

N = int(1e7)
rng = np.random.default_rng(seed=21)
A = rng.normal(loc=A_nom, scale=A_sig, size=N)
l = rng.normal(loc=l_nom, scale=l_sig, size=N)
C = rng.normal(loc=C_nom, scale=C_sig, size=N)

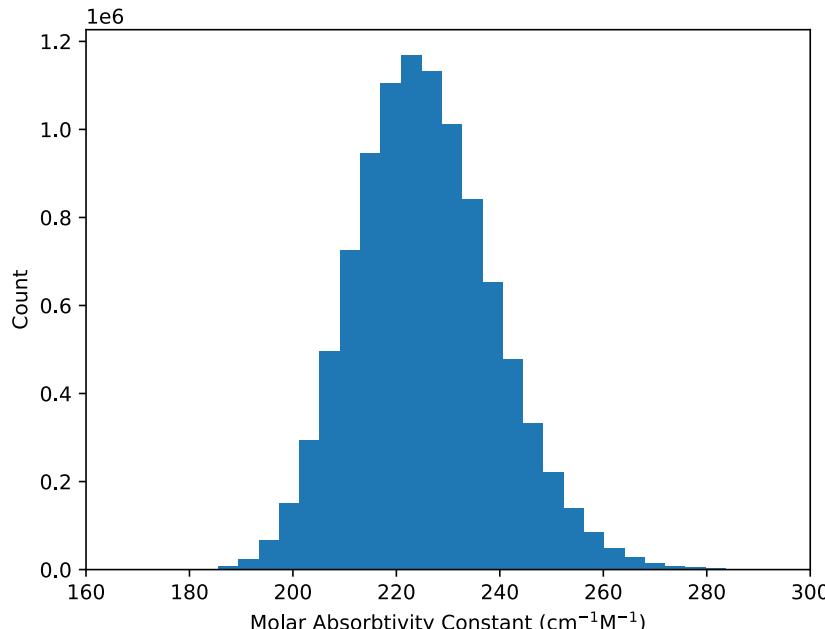
E = A / (l * C)

```

```

plt.hist(E, bins=40)
plt.xlim(160, 300)
plt.xlabel('Molar Absorbtivity Constant (cm$^{-1}$M$^{-1}$)')
plt.ylabel('Count');

```



```

print(np.mean(E))
print(np.std(E, ddof=1))

```

```

225.63035520507208
13.600103576801123

```

This results in a value of $226 \pm 14 \text{ cm}^{-1}\text{M}^{-1}$, which is close to what we calculated using the `uncertainties` library.

Further Reading

1. Documentation for `uncertainties` package. <https://uncertainties.readthedocs.io/en/latest/> (free resource)
2. NumPy `polyfit()` Documentation. <https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html> (free resource)
3. SciPy `curve_fit()` Documentation. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html (free resource)

4. Salter, C. Error Analysis Using the Variance-Covariance Matrix. *J. Chem. Educ.* **2000**, 77 (9), 1239. <https://doi.org/10.1021/ed077p1239>.

Provides background and an example of using a variance-covariance matrix.

Appendix 4: Regular Expressions

Contents

- Regular Expression Basics
- Finding CAS Numbers
- Parse NMR Data
- Further Reading

There is a saying that synthetic chemists spend 10% of their time running reactions and 90% of their time purifying compounds. A similar saying could be said that working with chemical data is 10% performing the intended calculations or analyses on the data and 90% of the time cleaning and organizing the data. While these are both hyperbole, they underline the large amount of effort required to clean materials. This chapter is dedicated to a powerful methods known as **regular expressions**, or **regex** for short, for cleaning and filtering text data, especially in situations requiring complex pattern matching. Python [string methods](#) and [indexing](#) offer basic search and filtering functionality, but they tend to only allow for identifying simple and consistent patterns. For example, if you want a file name without the file extension (e.g., *titration* instead of *titration.png*), this can be solved using indexing and the string `split()` method because file extensions always follow the last period in the full file name. Likewise, parsing data from a [PDB file](#) can be parsed with only a string search and slicing because PDB files follow very strict formatting rules based on labels and positions in rows. The reason these two examples are not terribly complex to parse is because they are consistent and were designed to be machine readable. However, not all data follow well-defined formatting rules or there could be more variation that needs to be accounted for. Regular expressions is not strictly a Python feature but rather is a syntax supported by Python using the `re` module imported below. This module is a built-in Python module, so it comes with every installation of Python.

```
import re
```

Below we will first cover some key functions from the `re` modules followed by generating more complex patterns, and finally ending with a couple chemical databases and literature examples.

Regular Expression Basics

`re` Functions

The `re` module provides a series of functions including those listed in **Table 1** that allow the user to search for, split on, or substitute for patterns within a string. Additional functions can be found on the [Python regular expressions page](#).

Table 1 Select `re` Functions

Regex Function	Description
<code>re.findall(pattern, str)</code>	Returns a list of strings that match the pattern
<code>re.finditer(pattern, str)</code>	Returns iterable of Match objects
<code>re.search(pattern, str)</code>	Returns the <i>first</i> pattern match as Match object
<code>re.split(pattern, str)</code>	Splits string at pattern matches
<code>re.sub(pattern, replacement, str)</code>	Replaces all occurrences with new string

The way these functions work is that the user provides a pattern to search for, which in the most basic scenarios can be a simple string, along with a string in which the function will search for the pattern. In the example below, we search a string of amine names for an aniline derivative by using `'aniline'` as a pattern.

```
amines = '2-methylcyclohexylamine N-methylaniline 3-methylbutylamine N-methyl-3-pentanamine o-methylaniline'
pattern = 'aniline'

re.findall(pattern, amines)
```

```
['aniline', 'aniline']
```

This is not terribly informative being that all it tells us is that `'aniline'` appears twice. The `re.finditer()` function can be used instead to return an iterator providing the user with the location of each match using either a `for` loop or `list()` function. We can see below that there are three matches along with the indices of those matches and the string that matches the pattern.

```
for x in re.finditer('aniline', amines):
    print(x)
```

```
<re.Match object; span=(32, 39), match='aniline'>
<re.Match object; span=(90, 97), match='aniline'>
```

```
list(re.finditer('aniline', amines))
```

```
[<re.Match object; span=(32, 39), match='aniline'>,
 <re.Match object; span=(90, 97), match='aniline'>]
```

To access the matched strings, use the `group()` method on the `Match` objections like below.

```
for x in re.finditer('aniline', amines):
    print(x.group())
```

```
aniline
aniline
```

The `re` module can also be used to find and replace patterns such as replacing `'aniline'` with `'anilinium'` like

below.

```
re.sub('aniline', 'anilinium', amines)
```

```
'2-methylcyclohexylamine N-methylanilinium 3-methylbutylamine N-methyl-3-pantanamine o-methylanilinium'
```

We could still probably have done the above tasks with string methods and indexing. The real power of regular expressions is its ability to generate more complex and flexible patterns, which is what we address below.

Symbols & Characters

Let's try something a little more complicated by searching for any instance of a methyl *not* located on a nitrogen. This means that the name should have a `'methyl'` string with a hyphenated number before it. The `re` module provides syntax, **Table 2**, for indicating specific types of characters and delimiters. For example, `\d` indicates a digit. Many of these character designators also have a negative version using the capital letter, so `\D`, for example, signifies any character except a number.

Table 2 Regex Character Designators

Character Type	Present	Not Present	Description/Examples
Any character	<code>.</code>		Any character except new line (i.e., <code>\n</code>)
Digits	<code>\d</code>	<code>\D</code>	Digits from 0-9
Letters/Word characters	<code>\w</code>	<code>\W</code>	abcABC
Space	<code>\s</code>	<code>\S</code>	White space, tabs, and end-of-lines
Boundary between words	<code>\b</code>	<code>\B</code>	Space, start of line, or non-alphanumeric characters
Character at start of string	<code>^</code>		<code>^2</code> finds a 2 at the start of a string
Character at end of string	<code>\$</code>		<code>^1</code> finds a 1 at the end of a string

Being that we need any number before the methyl, the pattern is `\d-methyl`. Now that we have patterns that use a backslash, you may see a `SyntaxError` because the backslash is also a Python escape character. To avoid this error, either precede the backslash with another backslash, `\d-methyl`, or make your string a raw string by preceding it with an `r` like is done below.

```
for x in re.finditer(r'\d-methyl', amines):
    print(x)
```

```
<re.Match object; span=(0, 8), match='2-methyl'>
<re.Match object; span=(40, 48), match='3-methyl'>
```

The `\D` could be used as a means of locating methyls that are *not* on an aliphatic carbon chain because they do not have numbers before them (at least in this example) like is done below. Now that our patterns are more broad, the listing of matches like below are more informative because we can see that both `N-methyl` and `o-methyl` fit our pattern.

```
for x in re.findall(r'\D-methyl', amines):
    print(x)
```

```
<re.Match object; span=(24, 32), match='N-methyl'>
<re.Match object; span=(59, 67), match='N-methyl'>
<re.Match object; span=(82, 90), match='o-methyl'>
```

As another example, below is a string that lists chemical identifiers including chemical names, CAS numbers, and a PubChem CID. The first thing we might want to do is split this up into a list where each item represents a different chemical.

```
chemicals = ('2-methylphenol    methanol N,N-diethylamine pentanol 281-23-2 '
             'ethyl benzoate glycerol 93-89-0 5793 ethanoic acid acetic anhydride')
```

Using a string method to split based on spaces demonstrated below will not work well because some chemicals (ethyl benzoate, ethanoic acid, and acetic anhydride) have a space in the name. There is also a complication where there are multiple spaces after `'2-methylphenol'`. This problem will be solved below using additional tools from regular expressions.

```
re.split(r'\s', chemicals)
```

```
['2-methylphenol',
 '',
 '',
 'methanol',
 'N,N-diethylamine',
 'pentanol',
 '281-23-2',
 'ethyl',
 'benzoate',
 'glycerol',
 '93-89-0',
 '5793',
 'ethanoic',
 'acid',
 'acetic',
 'anhydride']
```

Quantifiers

Let's first deal with the multiple spaces using quantifiers in **Table 3**. These quantifiers allow the user to specify how many of something will be in the pattern. For example, the `a+` will search for one or more a's while `\s{1,3}` looks for 1-3 spaces.

Table 3 Regex Quantifiers

Flag	Description	Example
*	Search for 0 or more	\w* for 0 or more letters
?	0 or 1	\s? for a space that may or may not be present
+	Search for 1 or more	\d+ for one or more digits
{}	Number of preceding character to search for	\d{3} for three digits, \d{3, 7} for 3-7 digit

Below, we use \s+ to split our string of chemicals based on one or more spaces.

```
re.split(r'\s+', chemicals)
```

```
['2-methylphenol',
'methanol',
'N,N-diethylamine',
'pentanol',
'281-23-2',
'ethyl',
'benzoate',
'glycerol',
'93-89-0',
'5793',
'ethanoic',
'acid',
'acetic',
'anhydride']
```

Lookahead and Lookbehind

Now let's address the issue of spaces inside the name. IUPAC nomenclature for esters follow a pattern where the first word always ends in -yl, and carboxylic acids and anhydrides have -ic at the end of the first word (i.e., the carboxyl part). These trends can be used to identify spaces where the string should *not* be split, and we will carry this out using something known as a **lookahead** or **lookbehind** shown in **Table 4**. These look for the presence or absence of something before or after our main pattern. We specifically want spaces that do *not* have a yl or ic preceding them. We will add these one at a time. Below (?<!yl) is added in front of \s+ to avoid splitting on yl patterns.

Table 4 Lookahead and Lookbehind Syntax

	Lookahead (\rightarrow)	Lookbehind (\leftarrow)
Present	pattern1(?=pattern2)	(?<=pattern2)pattern1
Absent	pattern1(?!=pattern2)	(?<!pattern2)pattern1

```
pattern = r'(?<!yl)\s+'
re.split(pattern, chemicals)
```

```
['2-methylphenol',
'methanol',
'N,N-diethylamine',
'pentanol',
'281-23-2',
'ethyl benzoate',
'glycerol',
'93-89-0',
'5793',
'ethanoic',
'acid',
'acetic',
'anhydride']
```

A lookbehind for the `ic` can also be added like below.

```
pattern = r'(?<!yl)(?<!ic)\s+'
re.split(pattern, chemicals)
```

```
['2-methylphenol',
'methanol',
'N,N-diethylamine',
'pentanol',
'281-23-2',
'ethyl benzoate',
'glycerol',
'93-89-0',
'5793',
'ethanoic acid',
'acetic anhydride']
```

Character Sets

What happens if there are multiple symbols that need to be matched? By placing the symbols or characters to be matched in square brackets, `[]`, anything in the brackets is searched for. For example, it is not uncommon to see numbers separated by either a period or dash (e.g., phone numbers), so `[.-.]` can be used to indicate that either symbol is a fit. Regular expressions also allows for ranges of letters and numbers such as `[a-e]` for any of the first five lowercase letters. It is a good idea to place the dash first to ensure that it does not get interpreted as a range.

Below there is a string of toluene derivatives. If we want to filter for only *para*-substituted toluene derivatives, the name (at least in this example) should start with either *p*- or *4*-. Both symbols can be enclosed in the square brackets like `[4p]`. The next challenge is figuring out how to deal with the rest of the symbols. We could try `.+` to indicated any number of more symbols, but this includes white spaces and returns the rest of the string.

```
toluene = '3-chlorotoluene 4-methyltoluene p-bromotoluene o-methoxytoluene'
re.findall(r'[4p]-.+', toluene)
```

```
['4-methyltoluene p-bromotoluene o-methoxytoluene']
```

To solve this, we can again use character sets to include any letter, number, or dash like below. By including the `+` behind the square brackets, this means one or more of these symbols.

```
re.findall(r'[4p] [‐\d\w]+', toluene)
```

```
['4-methyltoluene', 'p-bromotoluene']
```

Groups

Regular expressions in Python also support the extraction of information from specific segments in a string. In section [section 1.3.4](#), string formatting is introduced where the user can create a template string and insert different strings in various locations. Below are examples where the compound and molecular weight can be swapped out using either the `format()` method or *f*-string formatting.

```
compound = 'ammonia'  
MW = 17.03  
  
'The molar mass of {} is {} g/mol.'.format(compound, MW)
```

```
'The molar mass of ammonia is 17.03 g/mol.'
```

```
compound = 'urea'  
MW = 60.06  
  
f'The molar mass of {compound} is {MW} g/mol.'
```

```
'The molar mass of urea is 60.06 g/mol.'
```

Groups in regular expressions are essentially the opposite of above where information from the string is instead *extracted*. Groups are helpful for extracting data from a larger pattern. Below are a couple beginnings of NMR data listings that would appear in chemical literature. If we are interested in the carrier frequency, we simply write out the regular expression as normal but then wrap the part we want to extract in parentheses.

```
1H NMR (CDCl3, 400 MHz):  
13C NMR (C6D6, 100 MHz):
```

```
H_NMR = '1H NMR (CDCl3, 400 MHz):'  
C_NMR = '13C NMR (C6D6, 100 MHz):'  
  
carrier = r'1\d? [HC] NMR \(([^\d\w]+, (\d+) MHz)\):'
```

```
re.findall(carrier, H_NMR)
```

```
['400']
```

```
re.findall(carrier, C_NMR)
```

```
['100']
```

Multiple groupings can be extracted by wrapping multiple sections in parentheses. Below extracts both the solvent and the carrier frequency.

```
carrier = r'1\d? [HC] NMR ((([\d\w]+), (\d+) MHz):'
re.findall(carrier, H_NMR)
```

```
[('CDCl3', '400')]
```

Finding CAS Numbers

Let's now do some extra examples. When downloading data files from [PubChem](#), the CAS number is mixed in with other names and numerical identifiers. There are two challenges here. The first is that CAS numbers vary in length. They are always three segments of numbers separated by hyphens, such as 58-08-2 or 2501-94-2, where the second segment is always two digits and the thirds is always a single digit. However the first segment varies from 2-7 digits. The second major issue is that the CAS numbers are mixed in with other chemical identifiers such as CID numbers, common names, and IUPAC names. These other identifiers can include hyphens and numbers, so indexing and string searches cannot easily filter for CAS numbers without a long series of boolean conditions.

This is a relatively simple task for regular expressions. We indicate digits with the `\d` and use curly brackets to indicate the number of digits as demonstrated below.

```
re.findall(r'\d{2,7}-\d{2}-\d', chemicals)
```

```
['281-23-2', '93-89-0']
```

As a demonstration, [PubChem](#) allows for the free download of datasets which include a *Synonym* column. This column includes identifiers such as common and IUPAC names, CAS numbers, and PubChem CID numbers. The following code extracts the CAS numbers from one of these files. Two additional challenges arise from multiple CAS numbers being listed for a given compound or no CAS number being listed at all. When there are multiple CAS numbers, the most common one is stored, and if no CAS number is present, a `NaN` is stored in its place.

```

# get CAS number from Synonyms column
import pandas as pd
import numpy as np

solv = pd.read_csv('data/solvents.csv')
names = solv['Synonyms']

cas_pattern = r'\d{2,7}-\d{2}-\d'
cas = []
for row in names:
    cas_in_row = re.findall(cas_pattern, row)
    try:
        # get more common CAS number
        most_common_cas = max(set(cas_in_row), key=cas_in_row.count)
        cas.append(most_common_cas)
    except ValueError:
        # append NaN if no CAS number found
        cas.append(np.nan)

cas[:10]

```

```

['107-06-2',
 '120-82-1',
 '67-64-1',
 '71-43-2',
 '71-36-3',
 '111-65-9',
 '67-68-5',
 '64-17-5',
 '75-12-7',
 '67-56-1']

```

Parse NMR Data

When data on an NMR spectrum is reported in the literature, it follows relatively strict formatting rules, but these rules are designed to be ready by humans, not machines. To make things more complicated, there are numerous commas and spaces in the data making it difficult to use these as delimiters, so regular expressions are ideal for parsing this kind of data. Below is the ^1H NMR data for butanamide in $\text{DMSO}-d_6$ at 22°C following [American Chemical Society guidelines](#).

^1H NMR ($(\text{CD})_3\text{SO}$, 400 MHz): δ 7.23 (br, 1H), 6.70 (br, 1H), 2.00 (t, 2H, $J = 7.3$ Hz), 1.48 (tq, 2H, $J = 7.3, 7.3$ Hz), 0.84 (t, 3H, $J = 7.3$ Hz).

As an example, we will extract the entries for each signal in the NMR spectrum. Each entry looks like `7.23 (br, 1H)` or `0.84 (t, 3H, J = 7.3 Hz)` where the decimal is the chemical shift and additional information on the signal is provided in the parentheses behind the chemical shift.

```

proton = ('1H NMR ((CD)3SO, 400 MHz): 6 7.23 (br, 1H), 6.70 (br, 1H), '
          '2.00 (t, 2H, J = 7.3 Hz), 1.48 (tq, 2H, J = 7.3, 7.3 Hz), '
          '0.84 (t, 3H, J = 7.3 Hz).')

```

Each signal starts with a number to two decimal places, but there may be one or two digits before the decimal place. Even though our example always has one digit before the decimal, we want our code to be robust and versatile. The regular expression for this number is `'\d{1,2}.\d{2}'`.

```
nmr_pattern = r'\d{1,2}.\d{2}'  
re.findall(nmr_pattern, proton)
```

```
['7.23', '6.70', '2.00', '1.48', '0.84']
```

Next, the information about the signal is stored in parentheses separated from the chemical shift by a space. We will use `'\s+'` just in case someone accidentally used multiple spaces. Because parentheses are a regular expression character, we need to precede it with a backslash to indicate that we actually mean just a parentheses character.

```
nmr_pattern = r'\d+.\d{2}\s+\('  
re.findall(nmr_pattern, proton)
```

```
['7.23 (' , '6.70 (' , '2.00 (' , '1.48 (' , '0.84 (' ]
```

Inside the parentheses is the

- Splitting pattern as one or more letters, `'\w+'`
- Integration as an integer with an `H`, so `'\d+H'`
- Coupling information as starting with `J =` followed by a number to two decimal places, so `'J\s+=\s+\d+.\d+\s+Hz'`.

```
nmr_pattern = r'\d+.\d{2}\s+\(\w+, \s+\d+H, \s+J\s+=\s+\d+.\d\s+Hz\)'  
re.findall(nmr_pattern, proton)
```

```
['2.00 (t, 2H, J = 7.3 Hz)', '0.84 (t, 3H, J = 7.3 Hz)']
```

The current pattern misses the signals that do not include the coupling information or have multiple coupling constants. This is where quantifier are helpful. By placing the regular expression that pattern matches `, J = 7.3` in square brackets followed by an asterisk like below, it indicates that there could be zero or more of these.

```
[, \s+J\s?= \s?\d+.\d]*
```

```
nmr_pattern = r'\d+.\d{2}\s+\(\w+, \s+\d+H[, \s+J\s?= \s?\d+.\d]*\sHz\)'  
re.findall(nmr_pattern, proton)
```

```
['2.00 (t, 2H, J = 7.3 Hz)',  
'1.48 (tq, 2H, J = 7.3, 7.3 Hz)',  
'0.84 (t, 3H, J = 7.3 Hz)']
```

Now the regular expression finds all signals that have coupling constants but is still missing the two without coupling constants. This is because the pattern still requires a `' Hz'`. Because there should be either zero or one of these, the regular expression that searches for this should also be enclosed in square brackets and followed by an `*` like below.

```
[\sHz]*
```

```
nmr_pattern = r'\d+\.\d{2}\s\((\w+, \s+\d+H[, \s+J\s?= \s?\d+\.\d]*[\sHz]*\))'  
re.findall(nmr_pattern, proton)
```

```
['7.23 (br, 1H)',  
 '6.70 (br, 1H)',  
 '2.00 (t, 2H, J = 7.3 Hz)',  
 '1.48 (tq, 2H, J = 7.3, 7.3 Hz)',  
 '0.84 (t, 3H, J = 7.3 Hz)']
```

It looks like the code finds all the signals. One more addition that would be helpful in making the code more robust is to add the possibility of a negative chemical shift. While proton chemical shift are typically positive, negative values do show up in situations such as silanes with Si-H bonds or metal hydrides. To allow for this possibility, a `-?` is placed in the front indicated that the negative may or may not be there. To test this, an extra negative resonance was added just for testing purposes.

```
proton = ('1H NMR ((CD)3SO, 400 MHz): 6 7.23 (br, 1H), 6.70 (br, 1H),  
 2.00 (t, 2H, J = 7.3 Hz), 1.48 (tq, 2H, J = 7.3, 7.3 Hz),  
 0.84 (t, 3H, J = 7.3 Hz), -0.54 (s, 1H).')
```

```
nmr_pattern = r'-?\d+\.\d{2}\s\((\w+, \s+\d+H[, \s+J\s?= \s?\d+\.\d]*[\sHz]*\))'  
re.findall(nmr_pattern, proton)
```

```
['7.23 (br, 1H)',  
 '6.70 (br, 1H)',  
 '2.00 (t, 2H, J = 7.3 Hz)',  
 '1.48 (tq, 2H, J = 7.3, 7.3 Hz)',  
 '0.84 (t, 3H, J = 7.3 Hz)',  
 '-0.54 (s, 1H)']
```

If someone wanted to extract values from the NMR signals, additional regular expressions could be written to iterate through the list and extract the desired information.

Further Reading

1. Documentation for `re` package. <https://docs.python.org/3/library/re.html> (free resource)

Python `re` module documentation. Provides a good list of flags.

2. Regular Expressions HOWTO. <https://docs.python.org/3/howto/regex.html> (free resource)

An official Python documentation page that provides an additional tutorial on regular expressions in Python.

3. Datacamp Regular Expressions Cheat Sheet. <https://www.datacamp.com/cheat-sheet/regular-expression> (free resource)

A one-page summary of key regular expression pattern characters good for hanging above a desk.