

Problem Solving with Python

Peter D. Kazarinoff, PhD

Contents

| | | |
|----------|---|-----------|
| 1 | Preface | 5 |
| 1.1 | Motivation | 5 |
| 1.2 | Acknowledgements | 5 |
| 1.3 | Supporting Materials | 5 |
| 1.4 | Formatting Conventions | 6 |
| 1.5 | Errata | 8 |
| 2 | Orientation | 9 |
| 2.1 | Introduction | 9 |
| 2.2 | Why Python? | 9 |
| 2.3 | The Anaconda Distribution of Python | 11 |
| 2.4 | What is Anaconda? | 11 |
| 2.5 | Installing Anaconda on Windows | 13 |
| 2.6 | Installing Anaconda on MacOS | 17 |
| 2.7 | Installing Anaconda on Linux | 22 |
| 2.8 | Installing Python from Python.org | 22 |
| 2.9 | Summary | 23 |
| 2.10 | Review Questions | 23 |
| 3 | The Python REPL | 25 |
| 3.1 | Introduction | 25 |
| 3.2 | Python as a fancy calculator | 25 |
| 3.3 | Variables | 30 |
| 3.4 | Boolean Arithmetic | 31 |
| 3.5 | String Operations | 32 |
| 3.6 | Print Statements | 33 |
| 3.7 | Summary | 33 |

| | | |
|----------|--|-----------|
| 3.8 | Review Questions | 35 |
| 4 | Python Data Types and Variables | 37 |
| 4.1 | Introduction | 37 |
| 4.2 | Numeric Data Types | 37 |
| 4.3 | Boolean Data Type | 39 |
| 4.4 | Strings | 40 |
| 4.5 | Lists | 40 |
| 4.6 | Dictionaries and Tuples | 42 |
| 4.7 | Summary | 42 |
| 4.8 | Review Questions | 43 |
| 5 | Jupyter Notebooks | 45 |
| 5.1 | Introduction | 45 |
| 5.2 | What is a Jupyter Notebook? | 45 |
| 5.3 | Why Jupyter Notebooks? | 45 |
| 5.4 | Installing Jupyter | 46 |
| 5.5 | Opening a Jupyter Notebook | 47 |
| 5.6 | The Jupyter Notebook Interface | 47 |
| 5.7 | Magic Cells | 53 |
| 5.8 | Summary | 56 |
| 5.9 | Review Questions | 56 |
| 6 | Functions and Modules | 59 |
| 6.1 | Introduction | 59 |
| 6.2 | Why Functions? | 59 |
| 6.3 | First Function | 59 |
| 6.4 | Functions with Multiple Arguments | 61 |
| 6.5 | Functions with Default Arguments | 61 |
| 6.6 | Calling Functions from Other Files | 62 |
| 6.7 | Docstrings in Functions | 63 |
| 6.8 | Positional and Keyword Arguments | 65 |
| 6.9 | Summary | 67 |
| 6.10 | Review Questions | 68 |
| 7 | Plotting with Matplotlib | 69 |
| 7.1 | Introduction | 69 |

| | | |
|----------|----------------------------|------------|
| 7.2 | What is Matplotlib? | 69 |
| 7.3 | Installing Matplotlib | 69 |
| 7.4 | Line Plots | 70 |
| 7.5 | Multi Line Plots | 77 |
| 7.6 | Bar Plots | 80 |
| 7.7 | Error Bars | 84 |
| 7.8 | Histograms | 87 |
| 7.9 | Box Plots and Violin Plots | 88 |
| 7.10 | Scatter Plots | 90 |
| 7.11 | Plot annotations | 91 |
| 7.12 | Subplots | 93 |
| 7.13 | Plot Styles | 95 |
| 7.14 | Contour Plots | 100 |
| 7.15 | Quiver and Stream Plots | 107 |
| 7.16 | 3D Surface Plots | 122 |
| 7.17 | Summary | 127 |
| 7.18 | Review Questions | 128 |
| 8 | If Else Try Except | 129 |
| 8.1 | Introduction | 129 |
| 8.2 | Selection Statements | 129 |
| 8.3 | If statements | 129 |
| 8.4 | If-Else Statements | 131 |
| 8.5 | Try-Except Statments | 132 |
| 8.6 | Flow Charts | 132 |
| 8.7 | Summary | 133 |
| 8.8 | Review Questions | 133 |
| 9 | Loops | 135 |
| 9.1 | Introduction | 135 |
| 9.2 | While Loops | 135 |
| 9.3 | For Loops | 136 |
| 9.4 | Break and Continue | 138 |
| 9.5 | Flow Charts | 138 |
| 9.6 | Summary | 138 |

| | |
|---|------------|
| 10 Matricies and Arrays | 139 |
| 10.1 Introduction | 139 |
| 10.2 Numpy | 139 |
| 10.3 Installing Numpy | 141 |
| 10.4 Array Creation | 142 |
| 10.5 Array Indexing | 148 |
| 10.6 Array Slicing | 150 |
| 10.7 Array Operations | 151 |
| 10.8 Scalar Multiplication | 152 |
| 10.9 Systems of Linear Equations | 154 |
| 10.10 Summary | 155 |
| 10.11 Review Questions | 155 |
| 11 Symbolic Math | 157 |
| 11.1 Introduction | 157 |
| 11.2 SymPy | 157 |
| 11.3 Defining Variables | 158 |
| 11.4 Defining Equations | 161 |
| 11.5 Solving two equations for two unknowns | 161 |
| 11.6 Summary | 164 |
| 11.7 Review Questions | 164 |
| 12 Python and External Hardware | 165 |
| 12.1 Introduction | 165 |
| 12.2 Pyserial | 165 |
| 12.3 Bytes and Unicode Strings | 166 |
| 12.4 Reading a Sensor with Python | 168 |
| 12.5 Controlling an LED with Python | 173 |
| 12.6 Summary | 177 |
| 12.7 Review Questions | 177 |
| 13 MicroPython | 179 |
| 13.1 Introduction | 179 |
| 13.2 What is Micropython? | 179 |
| 13.3 Installing Micropython | 180 |
| 13.4 Micropython REPL | 190 |

| | |
|---|------------|
| 13.5 Blinking a LED | 196 |
| 13.6 Reading a Sensor | 199 |
| 13.7 Uploading code | 202 |
| 13.8 Summary | 206 |
| 13.9 Review Questions | 206 |
| 14 Appendix | 207 |
| 14.1 Contents | 207 |
| 14.2 Reserved and Key Words in Python | 207 |
| 14.3 ASCII Character Codes | 209 |
| 14.4 Virtual Environments | 212 |
| 14.5 Numpy Math Functions | 212 |
| 14.6 Git and github.com | 213 |
| 14.7 LaTeX Math | 214 |
| 14.8 Python for Undergraduate Engineers Book Construction | 214 |
| 14.9 About the Author | 215 |

Chapter 1

Preface

1.1 Motivation

The motivation for writing this book is that many undergraduate engineering students have to take a programming course based on MATLAB. MATLAB is a great piece of software, but it currently costs \$49.00 for a student license and requires a site license to be used on school computers. Subsequently, it is costly for a student to use MATLAB and it is costly for a college to support a course that uses MATLAB. In addition this site license expires eventually and student need to purchase another copy often before they finish their degree.

The Python programming language on the other hand is open source and free. To download and use Python, the cost to both the student and the college is zero (minus time spent). By moving an undergraduate engineering programming class to Python, students will save money and have greater access to the software they use in class. Further in their engineering education, students can continue to use Python for free.

1.2 Acknowledgements

The creation of this book and supporting material would not be possible without the gracious support of my wife and family. Students at Portland Community College continue to give me hope that the next generation of engineers will be a diverse group of team problem solvers.

The Python Data Science Handbook and Machine Learning in Python as well as Reiman Equations in Python served as inspiration and examples of using Jupyter Notebooks to construct a book. The bookbook repository on github provided a starting point for the tooling to convert the book from Nupyter Notebooks to the web and into LaTeX for printing.

1.3 Supporting Materials

Supporting materials for this text can be found at the textbook website:

<https://github.io/ProblemSolvingwithPython>

The textbook website contains all of the text in web format. Code examples and jupyter notebooks for the text can be found on the github repository for the book:

<https://github.com/ProfessorKazarinoff/ProblemSolvingwithPython>

Live notebooks, where the code examples found in the text can be run without installing any software, are available at:

<https://gitform.ucberkeley.edu/ProblemSolvingwithPython>

If you are an instructor and using this book in a course with students- please send me an email using your school email address. In the email, include the course your are teaching and the term, approximate enrollment, and a link to the course listing on your school website.

peter.kazarinoff@problemsolvingwithpython.com

I am happy to reply with a solution key for the end of chapter review problems as well as quiz and exam question banks.

1.4 Formatting Conventions

This book and supporting materials use the following formatting conventions:

Web Address

Web address will be shown as:

<https://github.com/professorkazarinoff/ProblemSolvingwithPython>

Import terms and vocabulary

Important terms and vocabulary is shown in *italic text*

There is a difference between *local variables* and *global variables* in Python code

File Names

File Names are shown in ***bold and italic text***

After completing the code, save the file as ***hello.py*** in the current directory.

Module and Package Names

Module and Package names will be shown in **bold text**

Numpy and **matplotlib** are two useful packages for engineers.

Inline code

Inline code including variable names and extensions are shown in monospace font

To compare a variable use `var == 'string'` and make sure to include `==`, the double equals sign

Separate code blocks

Separate code blocks will appear in their own sections in monospaced font

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

Anaconda Prompt Commands

Commands typed into the terminal or **Anaconda Prompt** are shown in separate boxes which contain the prompt symbol `>` before each line. Note the prompt `>` should not be typed. It is included to indicate Anaconda Prompt, not a character for the user to enter.

```
> conda create -n newenv python=3.6
> conda activate newenv
```

Terminal Commands

Commands typed into the terminal appear in separate boxes which contain the dollar symbol `$` before each line. Note the dollar symbol `$` should not be typed. It is included to indicate a terminal prompt, not a character for the user to enter.

```
$ pip install pint
$ cd pint_scripts
```

Python REPL Commands

Commands typed into the **Python REPL**, or Python Interpreter appear in separate code boxes, which contain the triple arrow prompt `>>>`. Note the triple arrow `>>>` sign should not be typed. Triple arrows are included to indicate the Python REPL prompt, not a character for the user to enter. Output from the Python REPL is shown on a separate line below the command, without the `>>>` prompt.

```
>>> 2 + 2
4
>>> print('Problem Solving with Python')
Problem Solving with Python
```

Jupyter Notebook cells

Commands typed into **Jupyter Notebook** cells will appear with the label `In [#] :`. Output from jupyter notebook cells are shown after the input cell. Only code in the input cells need to be typed. Output cell are be produced automatically by clicking the run button or typing `[shift]+[Enter]`

```
In [1]: A = 2
        B = 3
        C = A + B
        print(C)
```

5

1.5 Errata

Errata including any typos, code errors and formatting inconsistencies can be submitted to

errata@problemsolvingwithpython.com

Please include the chapter number and section number in your email. Thank-you in advanced for helping improve this text for future readers.

Chapter 2

Orientation

2.1 Introduction

Welcome to the world of problem solving with Python. This first Orientation chapter will help you get started by guiding you through the process of installing Python on your computer.

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

- Describe why Python is a good programming language for undergraduate engineers
- Describe applications where Python is used
- Detail advantages of Python over other programming languages
- Know the cost of Python
- Know the difference between Python and Anaconda
- Install Python on your computer
- Install Anaconda on your computer

2.2 Why Python?

You might be wondering “Why should an engineer learn Python?” There are other programming languages in the engineering world such as MATLAB, LabView, C++ and Java. What makes Python useful for engineers?

Python is a powerful programming language

Python defines the types of objects you build into your code. Unlike some other languages such as C, you do not need to declare the object type. The object type is also mutable, you can change the type of object easily and on the fly. There is a wide array of object types built into Python. Objects can change in size. Python objects can also contain mixed data types. Strings and floating point numbers can be part of the same list.

Python has an extensive standard library. A huge number of object types, functions and methods

are available for use without importing any external modules. These include math functions, list methods, calls to a computers system. There is a lot that can be done with the standard library. The first couple chapters of this book will just use the standard library. It can do a lot.

Python has over 100,000 external packages available for download and use. They are easy to install off of the python package index, commonly called PyPI (“pie pee eye”). There is a python package for just about everything. There are packages which can help you: interact with the web, make complex computations, do unit conversions, plot data, work with .csv, .xls, and .pdf files, manipulate images and video, read data from sensors and test equipment, train machine learning algorithms, design web apps, work with GIS data, work with astronomical data, and many more added every day. In this book we will use some of the more useful Python packages for engineers such as numpy, matplotlib, pandas, and scipy.

Python is easy to learn and use

Engineers solve the world’s problems in teams. One way Python helps solve these problems faster than other programming languages is that it is easy to learn and use. Python programs tends to be shorter and quicker to write than a program that does a similar function in other languages. In the rapid design, prototype, test, iterate cycle programming solutions can be spun up quickly. Python is also an easy language for fellow engineers on your team to learn. It is also quite human readable. While programmers can become preoccupied with a programs run time, it is development time that takes the longest.

Python is transportable

Python can be installed and run on each of the three major operating systems: Windows, Mac and Linux. On Mac and Linux Python comes installed out of the box. Just open up a terminal in on a Mac OSX or Linux machine and type python. That’s it, you are now using Python. On Windows I recommend downloading and using the Ananaconda distribution. The Anaconda distrobution is free and can be installed on all three major operating systems. The same programming environment can be replacated accross the three different opperating systems.

Python is free

MATLAB and LabView cost students to use and cost companies and colleges even more. Python is free to download and use. It is also open source and free to modify, contribute to, and propose improvements. All of the packages available for download on the Python Package Index, PyPI (pronouced pie-pee-eye) are free to download and install. Many more packages, scripts and utilities can be found in open source repos on github and bitbucket.

Python is growing

Python is growing in popularity. Python is particularly growing in the data sciences and in use with GIS systems, physical modeling, machine learning and computer vision. These are growing team problem solving areas for engineers.

Python has extensive standard library of modules and a vast array of external modules

The Python Standard Library

The Python Standard Library includes:

math, statistics, os, urllib, table for what they are used for. More can be found on Read-the-docs

External Modules available on PyPI

There are over 100,000 external modules available for Python on PyPI. Ones useful for engineers include

numpy, **matplotlib** and **jupyter** table for what they are used for

installing Python modules can be done on the command line or at the Anaconda Prompt using;

```
$ pip install module_name
```

where `module_name` is the name of the module you want to install.

2.3 The Anaconda Distribution of Python

I recommend problem solvers install the *Anaconda Distribution of Python*

2.4 What is Anaconda?

You might be wondering “OK, I know what Python is. It is computer programming language”
“But what is Anaconda? How is it different than Python?”

Anaconda is a Python Distribution

Anaconda is a Python Distribution. When you download Anaconda, you download a Python Interpreter plus a bunch of extra useful stuff.

How is Anaconda different from Python?

When you download Python from Python.org you get the Python Interpreter, a little text editing program called IDLE and all of the Python Standard Library modules.

When you download Anaconda from Anaconda.com, you get a Python Interpreter, Anaconda Prompt (a command line program), Spyder (a code editor) and about 200 extra Python modules that aren’t included in The Standard Library. The Anaconda distribution of Python also includes a program called Anaconda Navigator that allows you to launch Jupyter Notebooks quickly.

Why download Anaconda if I want to use is Python?

Regardless if you download Python from Python.org or if you download Anaconda (with all the extra stuff it comes with) from Anaconda.com, you will be able to write and execute Python code. However, there are a couple of advantages to using the Anaconda distribution of Python.

Anaconda includes Python plus about 200 additional Python packages

Anaconda is advantageous because it includes Python as well as about 200 additional Python packages. These other packages are all free to use. The packages that come with Anaconda includes many of the most common Python packages use to solve problems. If you download Anaconda, you get Python including The Standard Library plus 200 extra packages. If you download Python from Python.org, you just get Python and The Standard Library but no additional modules. You could install the extra modules that come with Anaconda (that don't come with plain old Python), but why not save a step (or 200 steps) and download one thing (Anaconda) instead of downloading 201 and one things (200 extra modules + 1 Python download).

Anaconda installs without administrator privileges

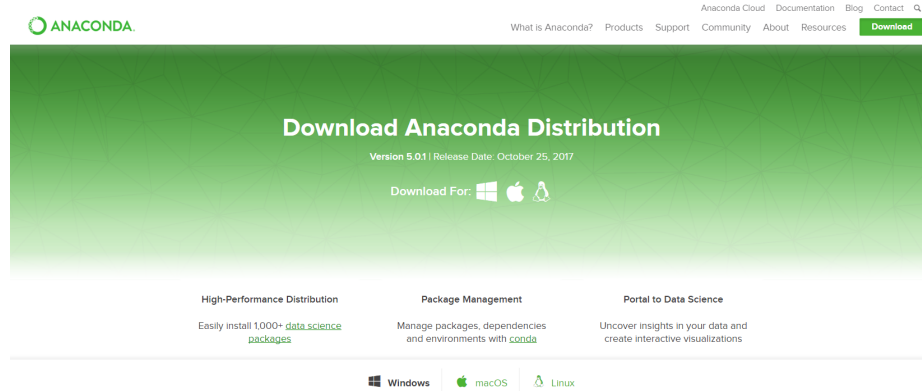
Even if you don't have the ability to install program on a computer, like a computer in a school computer lab, you can still download and use Anaconda. The Anaconda distribution of Python will also allow you to install additional modules from the Python package index ([PyPI.org](https://pypi.org)) and conda-forge, the conda package index.

Anaconda works on MacOS

If you use MacOS, you probably already have Python installed on your computer. Most MacOS installations come with Python included. The problem is that the version of Python that comes with MacOS is old (usually legacy Python, Python 2) and the version of Python that comes with MacOS is locked up behind a set of administrator privileges. Because the pre-installed version of Python can require administrator privileges, you can have trouble using the version of Python that comes on MacOS. Some things will seem to work fine, and then other things won't run at all, or you will keep getting asked for an administrator password over and over. Downloading and installing Anaconda (separate from the version of Python that came with MacOS) prevents most of these problems.

Anaconda makes package management and virtual environments easier

Another advantage of Anaconda is that package management and virtual environments are a lot easier when you have Anaconda. Virtual environments and package handling might not seem to make a huge difference right now. If you just downloaded Anaconda for the first time, you are probably not dealing with package management and virtual environments yet. (It's OK if you don't even know what those two things are yet). After you write a couple of Python programs and start downloading a couple of extra modules from PyPI or conda-forge, dealing with package management and virtual environments becomes more critical.



anaconda download page

2.5 Installing Anaconda on Windows

For undergraduate engineers, I recommend installing and using the Anaconda distribution of Python.

In this section, we will run through installing the Anaconda distribution of Python on Windows 10. I think the Anaconda distribution of Python is the the best option for undergraduate engineers who want to use Python. Anaconda is free (although the download is large which can take time) and can be installed on school or work computers where you don't have administrator access or the ability to install new programs. Anaconda comes bundled with over 100 packages pre-installed including **numpy**, **matplotlib** and **pandas**. These three packages are very useful for engineers and will be discussed in subsequent chapters.

Steps:

1. Visit [Anaconda.com/downloads](https://anaconda.com/downloads)
2. Select Windows
3. Download the .exe installer
4. Open and run the .exe installer
5. Open the Anaconda prompt and run some Python code

1. Visit the Anaconda downloads page

Go to the following link: [Anaconda.com/downloads](https://anaconda.com/downloads)

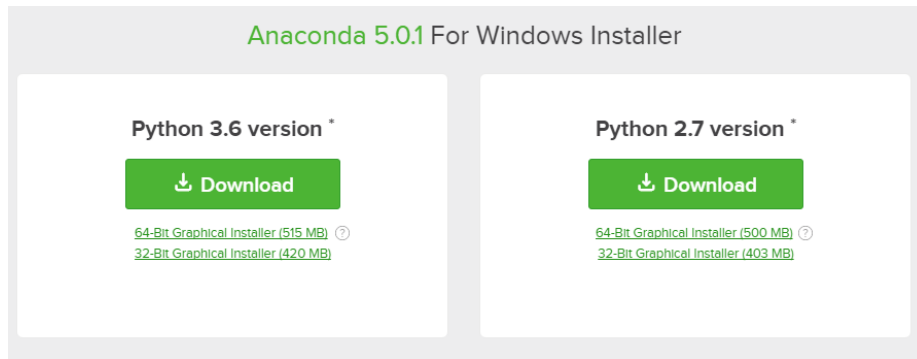
The Anaconda Downloads Page will look something like this:

2. Select Windows

Select Windows where the three operating systems are listed.



anaconda select Windows



anaconda select python 3.6

3. Download

Download the Python 3.6 distribution. Python 2.7 is legacy Python. For undergraduate engineers, select the Python 3.6 version.

You may be prompted to enter your email. You can still download Anaconda if you click **No Thanks** and don't enter your Work Email address.

The download is quite large (over 500 MB) so it may take a while for the download to complete.

4. Open and run the installer

Once the download completes, open and run the .exe installer

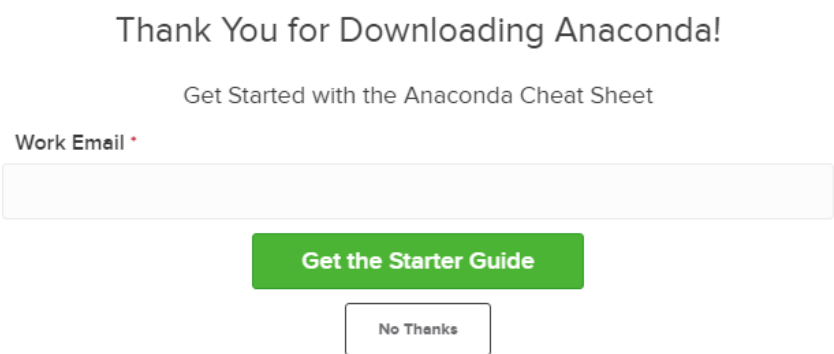
At the beginning of the install you will need to click **Next** to confirm the installation and agree to the license

At the Advanced Installation Options screen, I recommend that you **do not check** "Add Anaconda to my PATH environment variable"

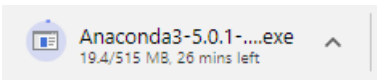
5. Open the Conda prompt from the Windows start menu

After the Anaconda install is complete, you can go to the Windows start menu and select the Anaconda Prompt

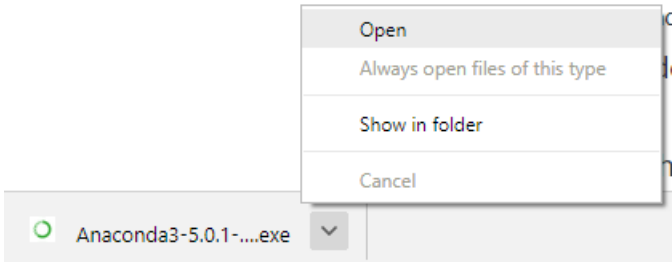
This will open up the **Anaconda Prompt**, which is often called the **conda prompt**. **Anaconda** is the Python distribution and the **Anaconda Prompt** is a command line shell (a program where you type in your commands instead of using a mouse). The black screen and text that makes up the **Anaconda Prompt** doesn't look like much, but it is really helpful for an undergraduate engineer using Python.



anaconda



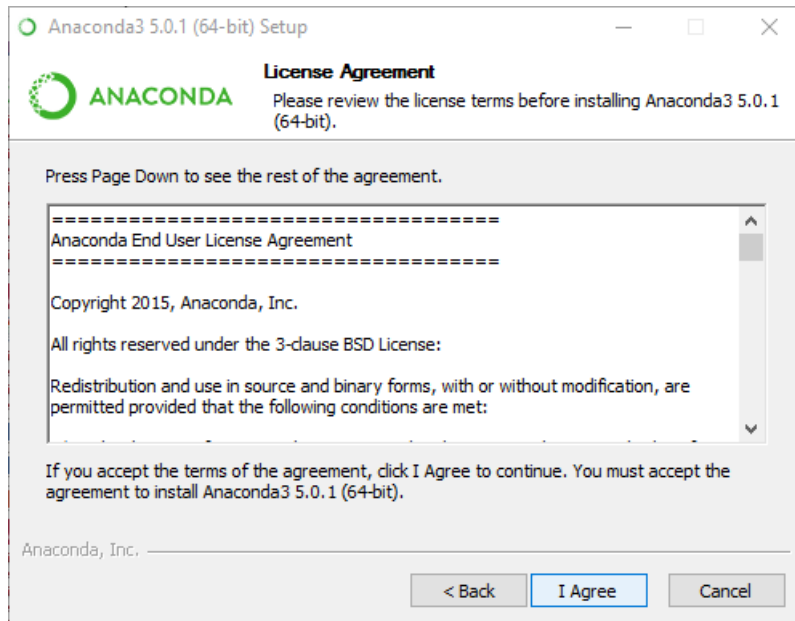
anaconda downloading



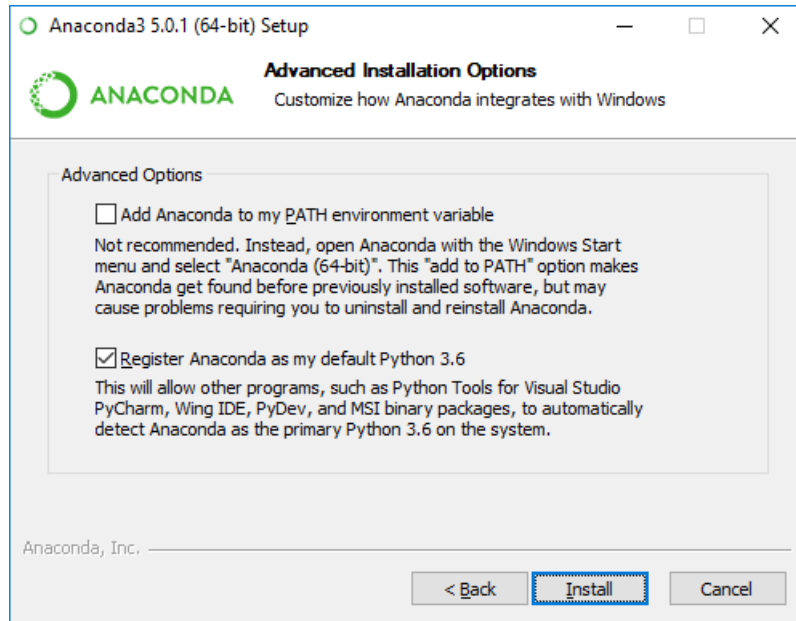
anaconda installer



anaconda installer click next



anaconda license



anaconda path variable

At the Anaconda prompt, type `python`. This will start the Python interpreter, also called the Python REPL (for Read Evaluate Print Loop).

Note the Python version. You should see something like Python 3.6.1. With the interpreter running, you will see a set of greater-than symbols `>>>` before the cursor.

Now you can type Python commands. Try typing `import this`. You should see the *Zen of Python* by Tim Peters

To close the Python interpreter, type `exit()` at the interpreter prompt `>>>`. Note the double parenthesis at the end of the `exit()` command. The `()` is needed to stop the Python interpreter and get back out to the **Anaconda Prompt**.

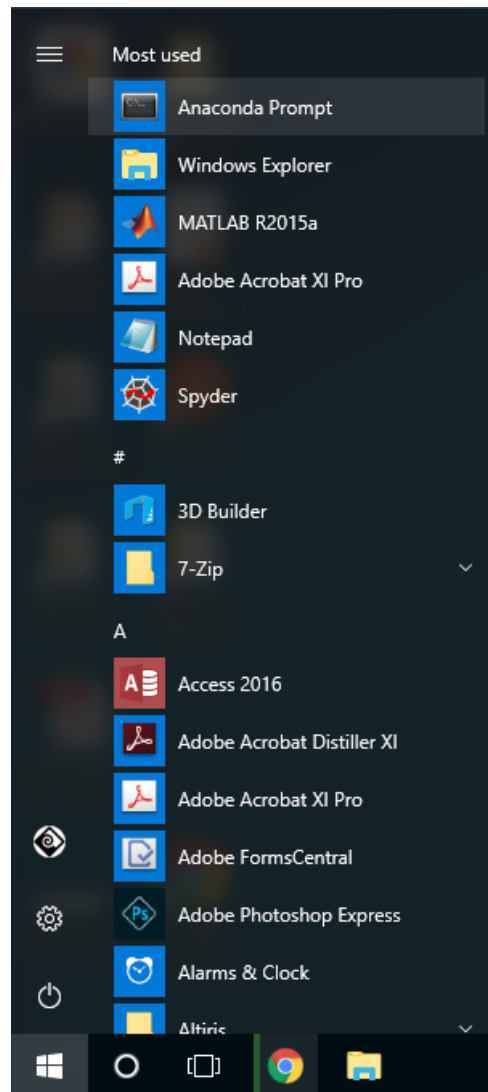
To close the **Anaconda Prompt**, you can either close the window with the mouse, or type `exit`, no parenthesis necessary.

Congratulations! You installed the Anaconda distribution on your Windows computer!

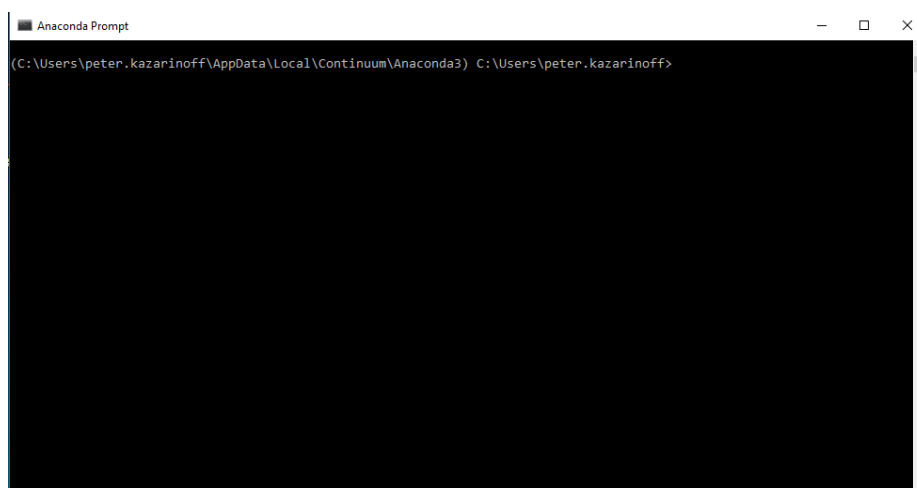
When you want to use the Python interpreter again, just click the Windows Start button and select the **Anaconda Prompt** and type `python`.

2.6 Installing Anaconda on MacOS

In this section, we will run through installing the Anaconda distribution of Python on MacOS. Most versions of MacOS come pre-installed with legacy Python (Version 2.7). You can confirm this legacy version of Python is installed by opening the MacOS **terminal**. To open the MacOS terminal use `[command]+[Space Bar]` and type `terminal` in the Spotlight Search bar.

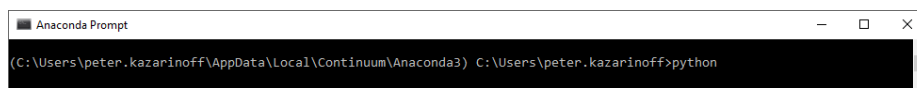


anaconda in start menu



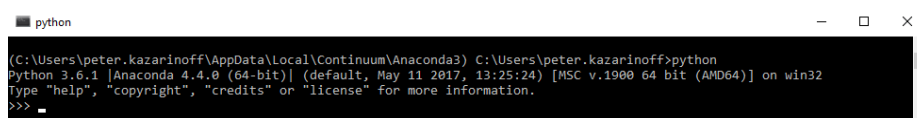
```
Anaconda Prompt
(C:\Users\peter.kazarinoff\AppData\Local\Continuum\Anaconda3) C:\Users\peter.kazarinoff>
```

anaconda prompt



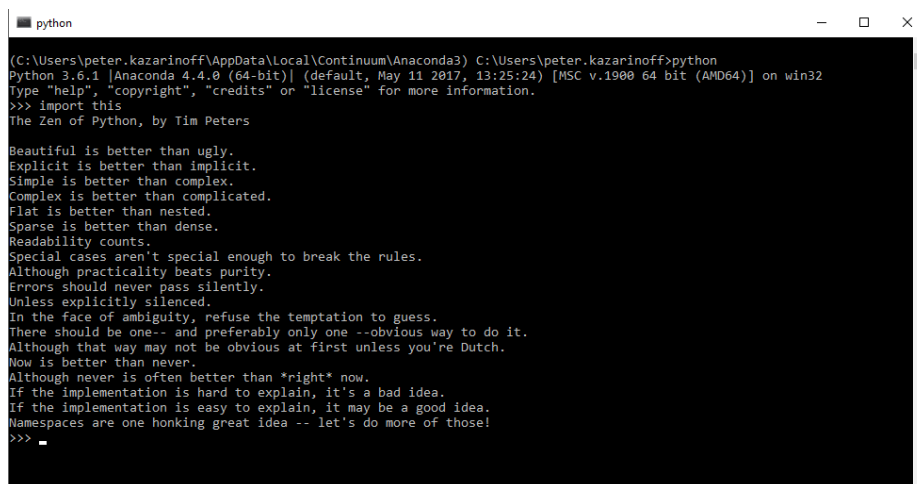
```
Anaconda Prompt
(C:\Users\peter.kazarinoff\AppData\Local\Continuum\Anaconda3) C:\Users\peter.kazarinoff>python
```

conda prompt type python



```
python
(C:\Users\peter.kazarinoff\AppData\Local\Continuum\Anaconda3) C:\Users\peter.kazarinoff>python
Python 3.6.1 [Anaconda 4.4.0 (64-bit)] (default, May 11 2017, 13:25:24) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

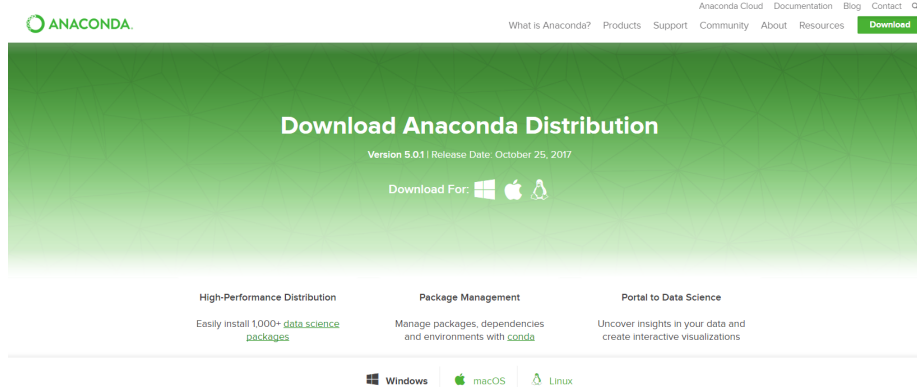
anaconda prompt



```
python
(C:\Users\peter.kazarinoff\AppData\Local\Continuum\Anaconda3) C:\Users\peter.kazarinoff>python
Python 3.6.1 [Anaconda 4.4.0 (64-bit)] (default, May 11 2017, 13:25:24) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

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

anaconda_import_this



anaconda download page

In the MacOS Terminal type (note: the dollar sign \$ is used to indicate the terminal prompt. The dollar sign \$ does not need to be typed):

```
$ python
```

You will most likely see version 2.7 is installed. An issue for MacOS users is that the installed system version of Python has a set of permissions that will not always allow Python to run and may not allow Python to install external packages. Therefore, I recommend the **Anaconda** distribution of Python is installed in addition to the system version of Python on MacOS. You will be able to run Python code using the **Anaconda** distribution of Python, and you will be able to install external packages on the **Anaconda** distribution of Python.

To install the **Anaconda** distribution of Python follow the steps below:

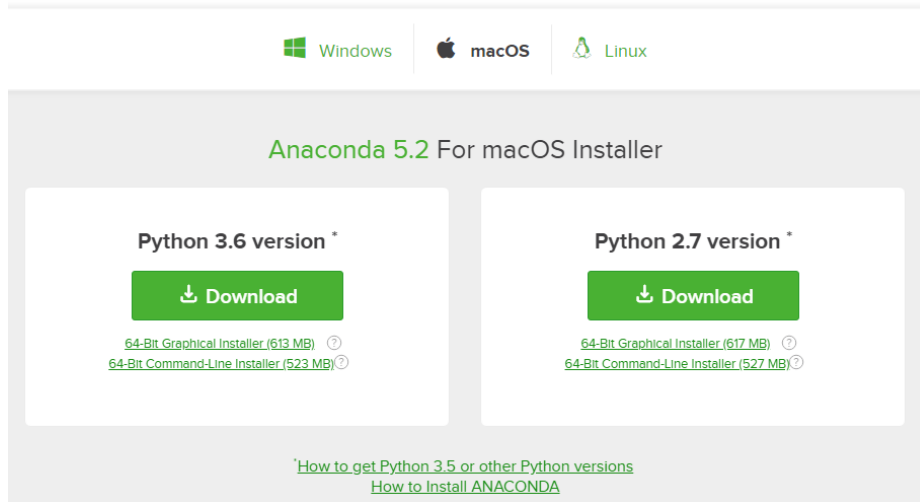
Steps:

1. Visit [Anaconda.com/downloads](https://anaconda.com/downloads)
2. Select MacOS and Download the .pkg installer
3. Open the .pkg installer
4. Follow the installation instructions
5. Source your .bash-rc file
6. Open a terminal and type python and run some code.

1. Visit the Anaconda downloads page

Go to the following link: [Anaconda.com/downloads](https://anaconda.com/downloads)

The Anaconda Downloads Page will look something like this:



Anaconda select Python 3.6

2. Select MacOS and download the .pkg installer

In the operating systems box, select [MacOS]. Then download the most recent Python 3 distribution (at the time of this writing the most recent version is Python 3.6) graphical installer by clicking the Download link. Python 2.7 is legacy Python. For undergraduate engineers, select the most recent Python 3 version.

You may be prompted to enter your email. You can still download Anaconda if you click **No Thanks** or [x] and don't enter your Work Email address.

3. Open the .pkg installer

Using the MacOS finder, navigate to the downloads folder and double click the *.pkg* installer file you just downloaded. It may be helpful to order your downloads by date to find the *.pkg* file.

4. Follow the installation instructions

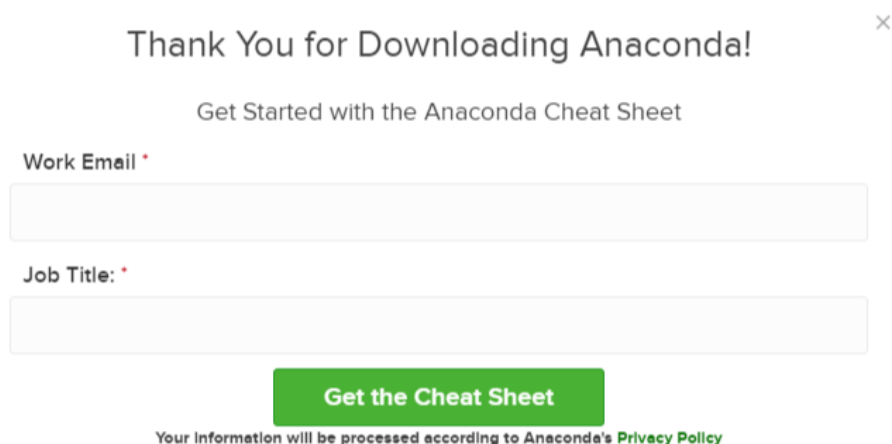
Follow the installation instructions. It is advised that you install **Anaconda** for the current user and that **Anaconda is added to your PATH**.

5. Source your .bash-rc file

Once Anaconda is installed, you need to load the changes to your PATH environment variable in the current terminal session.

Open the MacOS Terminal and type:

```
$ cd ~  
$ source .bashrc
```



Thank You for Downloading Anaconda! ×

Get Started with the Anaconda Cheat Sheet

Work Email *

Job Title: *

Get the Cheat Sheet

Your information will be processed according to Anaconda's [Privacy Policy](#)

Anaconda ask for email

6. Open a terminal and type python and run some code.

Open the MacOS Terminal and type:

```
$ python
```

You should see something like

```
Python 3.6.3 | Anaconda Inc. |
```

At the Python REPL (the Python >>> prompt) try:

```
>>> import this
```

If you see the Zen of Python, the installation was successful. Exit out of the Python REPL using `>>> exit()`. Make sure to include the double parenthesis () after the exit command.

2.7 Installing Anaconda on Linux

This section details the installation of the Anaconda distribution of Python on Linux. In particular these instructions are for Ubuntu 16.04, but the instructions should work for other Debian-based Linux distributions as well.

2.8 Installing Python from Python.org

Below are the recommended ways to install a new version of Python on the three major operating systems: Windows, Mac OSX and Linux. This book is based on Python version 3.6. Some of the problems may not work properly on legacy Python (version 2.7). I recommend using the Anaconda distribution of Python on Windows and MacOSX. The installation of Anaconda is detailed in the next section.

Installing Python on Windows

Go to [\[https://Python.org/downloads\]](https://Python.org/downloads)[\[https://Python.org/downloads\]](https://Python.org/downloads) to download the latest release. Make sure to select the box [add Python to my path] during the installation.

[image of Python.org]

Installing Python on Mac OSX

Go to <https://anaconda.com> and download the latest release.

[image of anaconda.com]

Installing Python on Linux

Go to open a terminal and enter `$ python` to see if a version of Python is already installed on the system. If the Python version is 2.7 or below, download the newest release in the apt repositories with and download the latest release.

```
$ sudo apt-get Python
```

2.9 Summary

Section summary

Vocabulary and Key Concepts

2.10 Review Questions

1. What is Python? How is the Python language different than the Python Interpreter?
2. What is the difference between the version of Python at Python.org and the version of Python at Anaconda.com?
3. Why does the text advise that undergraduate engineers use
- 4.
- 5.
- 6.
- 7.

Chapter 3

The Python REPL

3.1 Introduction

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

- Open and close the Python REPL
- Compute mathematical calculations using the Python REPL
- Use the output from Python command line as input in another problem
- Import the math and statistics module from the standard library and use their functions
- Combine True and False in logical statements

3.2 Python as a fancy calculator

Python can be used as a fancy calculator to do arithmetic like addition, subtraction, multiplication and division. It can also be used for trigonometric calculations and statistical calculations.

Arithmetic

Python can be used as a calculator to make simple or complex calculations.

We can do this easily with Python at the Python Prompt, often called the Python REPL for Read Evaluate Print Loop. The Python REPL shows three arrow symbols `>>>` after which you will see a blinking cursor. Programmers type commands at the prompt then hit [ENTER]. The is Read by the interpreter, results of running the commands are Evaluated then Printed to the command window. After the output a new `>>>` prompt appears on a new line. This process happens over and over again (in a loop). Try the following commands at the Python REPL:

Suppose the mass of a battery is 5 kg and the mass of the battery cables is 3 kg. What is the mass of the battery cable assembly?

```
>>> 5 + 3
8
```

Suppose one of the cables above is removed and it has a mass of 1.5 kg. What is the mass of the left over assembly

```
>>> 8 - 1.5
6.5
```

If the battery has a mass of 5000 g and a volume of 2500 cm^3 What is the density of the battery? The formula for density is below, where D is density, m is mass and v is volume.

$$D = \frac{m}{v}$$

In the problem above $m = 5000$ and $v = 2500$

Let's solve this with Python

```
>>> 5000 / 2500
2.0
```

What if we have 2 batteries that each weight 5 kg? How much mass is two batteries?

```
>>> 5 * 2
2.0
```

The dimension of the battery of each battery is 3 cm. What is the area of the base of the battery? To do this problems, use the the double asterisk symbol ****** to raise a number to a power. This is similar to using 3^2 .

```
>>> 3 ** 2
9
```

What is the volume of the battery if each the length, width and height of the battery are all 3 cm?

```
>>> 3 ** 3
27
```

Find the mass of the two batteries, and two cables

We can use Python to find the mass of the batteries and then use the answer, which Python saves as an underscore **_** to use in our next operation. (This is similar to **ans** in MATLAB)

```
>>> 2 * 5
10
>>> _ + 1.5 + 1
12.5
```

Section Summary

A summary of the arithmetic operations in Python is below:

| operator | name | description | example | result |
|----------|----------------|----------------------------|---------|--------|
| + | addition | adds two numbers | 2 + 3 | 5 |
| - | subtraction | subtracts two numbers | 8 - 6 | 2 |
| - | negation | negative number | -4 | -4 |
| * | multiplication | multiplies two numbers | 5 * 2 | 10 |
| / | division | divides two numbers | 6 / 3 | 2 |
| ** | exponents | raises a number to a power | 10**2 | 100 |
| _ | underscore | returns last saved value | | |

Trig: sin, cos, tan

Python as a fancy calculator is not limited to simple arithmetic. Trig functions such as sin, cos and tan are also available.

In order to use these functions, we need to learn a new concept: importing modules.

In python there are many functions built into the language when it starts. These include +, -, *, / like we saw in the previous section. However, not all functions will work right when Python starts. Say we want to find the sine of an angle. Try the following:

```
>>> sin(60)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
```

This error is returned because we have not told python to include the sin function. The sin function is part of the *standard library*. The standard library comes with every python installation and includes many functions, but not all of these functions are available to us when we start a new python window. In order to use the sin function, we first need to import it from math *module* which is part of the standard library.

Importing modules and functions is easy. We use the following syntax: from *module name* import *function name*

To import the sin() function from the math module try:

```
>>> from math import sin
>>> sin(60)
-0.3048106211022167
```

Success! Multiple modules can be imported at the same time. Say we want to use a bunch of different trig functions to solve the following problem.

Angle has a value of $\pi/6$ radians. What is the sin, cos and tan of the angle?

To solve this problem we need to import the sin(), cos(), and tan() functions. It would also be use full to have the value of π , rather than having to write 3.14.... We can import all of these functions at the same time using the syntax from <module> import <function1>, <function2>, <function3>. Note the commas in between the function names.

Try:

```
>>> from math import sin, cos, tan, pi
```

```
>>> pi
3.141592653589793
>>> pi/4
0.7853981633974483
>>> sin(pi/4)
0.7853981633974483
>>> sin(2*pi)
```

Section Summary

The following trig functions are part of Python's **math** module:

| trig function | name | description | example | result |
|-----------------------------|--------------------|-----------------------------------|-------------------------------|--------|
| <code>math.pi</code> | pi | mathematical constant π | <code>math.pi</code> | 3.14 |
| <code>math.sin()</code> | sine | sine of an angle in radians | <code>math.sin(4)</code> | 9.025 |
| <code>math.cos()</code> | cosine | cosine of an angle in radians | <code>cos(3.1)</code> | 400 |
| <code>math.tan()</code> | tangent | tangent of an angle in radians | <code>tan(100)</code> | 2.0 |
| <code>math.asin()</code> | arc sine | inverse sine, ouput in radians | <code>math.sin(4)</code> | 9.025 |
| <code>math.acos()</code> | arc cosine | inverse cosine, ouput in radians | <code>log(3.1)</code> | 400 |
| <code>math.atan()</code> | arc tangent | inverse tangent, ouput in radians | <code>atan(100)</code> | 2.0 |
| <code>math.radians()</code> | radians conversion | degrees to radians | <code>math.radians(90)</code> | 1.57 |
| <code>math.degrees()</code> | degree conversion | radians to degrees | <code>math.degrees(2)</code> | 114.59 |

Exponents and Logarithms

It is easy to calculate exponents and logarithms as well. Note that these need to be imported from the **math** module just like the trig functions above.

The following functions can be imported from the **math** module:

- `log`
- `log10`
- `exp`
- `e`
- `pow(x,y)`
- `sqrt`

Let's try a couple of examples

```
>>> from math import log, log10, exp, e, pow, sqrt
>>> log(3.0*e**3.4)           # note: natural log
4.4986122886681095
```

A right triangle has side lengths 3 and 4. What is the length of the hypotenuse?

```
>>> sqrt(3**2 + 4**2)
5.0
```

The power function `pow()` works like the `**` operator to raise a number to a power.


```
>>> 5**2
25

>>> pow(5,2)
25.0
```

Section Summary

The following exponent and logarithm functions are part of the **math** module:

| math module function | name | description | example | result |
|---------------------------|-------------------|----------------------------|------------------------------|--------|
| <code>math.e</code> | euler's number | mathematical constant e | <code>math.e</code> | 2.718 |
| <code>math.exp()</code> | exponent | e raised to a power | <code>math.exp(2.2)</code> | 9.025 |
| <code>math.log()</code> | natural logerithm | log base e | <code>math.log(3.1)</code> | 400 |
| <code>math.log10()</code> | base 10 logerithm | log base 10 | <code>math.log10(100)</code> | 2.0 |
| <code>math.pow()</code> | exponents | raises a number to a power | <code>math.pow(2,3)</code> | 8.0 |
| <code>math.sqrt()</code> | square root | square root of a number | <code>math.sqrt(16)</code> | 4.0 |

Staticstics

To round out this section, we will look at a couple of statistics functions. These functions are part of the **standard library** but not part of the **math** module. To access these statistics functions we need to import them from the **statistics** module using the statement `from statistics import mean, median, mode, stdev`. Then the functions `mean`, `median`, `mode` and `stdev` (standard deviation) can be used.

```
>>> from statistics import mean, median, mode, stdev

>>> test_scores = [ 60 , 83, 83, 91, 100]

>>> mean(test_scores)
83.4

>>> median(test_scores)
83

>>> mode(test_scores)
83

>>> stdev(test_scores)
14.842506526863986
```

Alternatively, we can import the whole **statistics** module using the statement `import statistics`. Then to use the functions, we need to use the names `statistics.mean`, `statistics.median`, `statistics.mode`, and `statistics.stdev`. See below:

```
>>> import statistics
```

```
>>> test_scores = [ 60 , 83, 83, 91, 100]

>>> statistics.mean(test_scores)
83.4

>>> statistics.median(test_scores)
83

>>> statistics.mode(test_scores)
83

>>> statistics.stdev(test_scores)
14.842506526863986
```

Section Summary

The following functions are part of the **statistics** module:

| statistics module function | name | description | example | result |
|----------------------------|--------------------|------------------|---------------------|--------|
| mean() | mean | mean or average | mean([1,4,5,5]) | 3.75 |
| median() | median | middle value | median([1,4,5,5]) | 4.5 |
| mode() | mode | most often | mode([1,4,5,5]) | 5 |
| stdev() | standard deviation | spread of data | stdev([1,4,5,5]) | 1.892 |
| variance() | variance | variance of data | variance([1,4,5,5]) | 3.583 |

3.3 Variables

Variables are assigned in Python using the = equals sign also called the assignment operator. The statment:

```
a = 2
```

Assigns the integer 2 to the variable a. Note this is different from the logical operator == equivalent to.

```
>>> a = 2
>>> a
2
```

```
>>> a == 2
True
```

Problem: The Arrhenius relationship states that

$$n = n_v e^{-Q_v/(RT)}$$

We can use variables to assign a value to each one of the constants in the problem.

```
>>> nv = 2.0**(-0.3)
>>> Qv = 5
>>> R = 3.18
>>> T = 293
>>> n = nv*e**(-1*Qv/(R*T))
>>> n
0.8079052775625613
```

3.4 Boolean Arithmetic

Boolean Arithmetic is the arithmetic of true and false logic. A boolean or logical value can either be True or False.

```
In [1]: A = True
        B = False
```

```
In [2]: A
```

```
Out[2]: True
```

```
In [3]: B
```

```
Out[3]: False
```

```
In [4]: A or B
```

```
Out[4]: True
```

```
In [5]: A and B
```

```
Out[5]: False
```

```
In [6]: not A
```

```
Out[6]: False
```

```
In [7]: not B
```

```
Out[7]: True
```

```
In [8]: A == B
```

```
Out[8]: False
```

```
In [9]: A != B
```

```
Out[9]: True
```

```
In [10]: C = False
```

```
In [11]: A or (C and B)
```

```
Out[11]: True
```

```
In [12]: (A and B) or C
```

```
Out[12]: False
```

```
In [1]: a = 'me'
```

3.5 String Operations

Some operations we can do on strings include indexing, concatenation, and logical comparisons

Indexing

```
In [1]: name = 'Lady Ada'
```

```
In [2]: name[1]
```

```
Out[2]: 'a'
```

```
In [3]: name[2]
```

```
Out[3]: 'd'
```

```
In [4]: name[1:4]
```

```
Out[4]: 'ady'
```

```
In [5]: name[:]
```

```
Out[5]: 'Lady Ada'
```

```
In [6]: name[-1]
```

```
Out[6]: 'a'
```

```
In [7]: name[-3:-1] #not including ending
```

```
Out[7]: 'Ad'
```

3.6 Print Statements

The `print()` function useful in Python. Below is a code example:

```
>>> name = Kendra
>>> print('Your name is: ')
Your name is
>>> print(name)
Kendra
```

3.7 Summary

This is the text summary for the chapter. Will be about half a page long

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Key Terms and Concepts

| Key Terms and Concepts |
|------------------------|
| REPL |
| Operator |
| Mathematical Operator |
| Command Line |
| Error |
| Module |
| Standard Library |
| Import |

Summary of Python Functions and Commands

Below is a summary of the functions and operators used in this chapter:

Arithmetic

| Arithmetic Operators | description |
|----------------------|----------------|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |

| Arithmetic Operators | description |
|----------------------|------------------|
| ** | Exponents |
| - | answer in memory |

Trigonometry

| Trig Function | Description |
|---------------------------------|-----------------------------|
| <code>from math import *</code> | |
| <code>sin</code> | sine of angle in radians |
| <code>cos</code> | cosine of angle in radians |
| <code>tan</code> | tangent of angle in radians |
| <code>pi</code> | π |
| <code>deg</code> | convert radians to degrees |
| <code>rad</code> | convert degrees to radians |
| <code>asin</code> | inverse sine |
| <code>acos</code> | inverse cosine |
| <code>atan</code> | inverse tangent |

Logarithms and Exponents

| Logarithms and Exponent Function | Description |
|----------------------------------|-------------------------|
| <code>from math import *</code> | |
| <code>log</code> | log base e, natural log |
| <code>log10</code> | log base 10 |
| <code>exp</code> | e^{power} |
| <code>e</code> | the math constant e |
| <code>pow(x,y)</code> | x raised to the y power |
| <code>sqrt</code> | square root |

Statistics

| Statistical Function | Description |
|---------------------------------------|------------------------------------|
| <code>from statistics import *</code> | |
| <code>mean</code> | mean (average) |
| <code>median</code> | median (middle value) |
| <code>mode</code> | (most often) |
| <code>stdev</code> | standard deviation of a sample |
| <code>pstdev</code> | standard deviation of a population |

3.8 Review Questions

2.1

2.2

2.3

2.4

2.5

2.6

2.7

2.8

2.9

2.10

Chapter 4

Python Data Types and Variables

4.1 Introduction

Python has many builtin data types. These include integers, floats, boolean, strings, and lists.

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

- Use the `type()` function to determine an object's type
- Define variables in Python with the assignment operator `=`
- Complete mathematical calculations with variables
- Complete logical evaluations with variables
- Convert variables from one data type to another

4.2 Numeric Data Types

Python has many useful built in data types. Python variables can store different types of data and can be created dynamically, without first defining a data type. It's useful for engineers to understand a couple of Python's core data types in order to write well constructed code. Below we will discuss a few different data types.

Integers

Integers are one of the Python data types. An integer is a whole number, negative, positive or zero. In Python, integer variables can be defined by simply assigning a whole number to a variable name. We can determine data type of a variable using the `type()` function.

```
>>> a = 5
>>> type(a)
<class 'int'>
>>> b = -2
```

```
>>> type(b)
<class 'int'>
>>> z = 0
>>> type(z)
<class 'int'>
```

Floating Point Numbers

Floating point numbers or *floats* are another Python data type. Floats are decimals, positive, negative and zero. Floats can also be numbers in scientific notation which contain exponents. Both a lower case *e* or an upper case *E* work to define floats in scientific notation. In Python, a float can be defined using a decimal point *.* when a variable is assigned.

```
>>> c = 6.2
>>> type(c)
<class 'float'>
>>> d = -0.03
>>> type(d)
<class 'float'>
>>> Na = 6.02e23
>>> Na
6.02e+23
>>> type(Na)
<class 'float'>
```

To make sure a variable is a float instead of an integer even if it is a whole number, a trailing decimal point *.* is used. Note the difference when a decimal point comes after the a whole number:

```
>>> g = 5
>>> type(g)
<class 'int'>
>>> g = 5.
>>> type(g)
<class 'float'>
```

Complex numbers

Another data type useful to engineers are complex numbers. A complex number is defined in Python using a real component + imaginary component *j*. The letter *j* must be used in the imaginary component. Using the letter *i* will return an error. Note how imaginary numbers can be added to integers and floats.

```
>>> comp = 4 + 2j
>>> type(comp)
<class 'complex'>
```

```
>>> comp2 = 4 + 2i
              ^
```

```
SyntaxError: invalid syntax
```

```
>>> intgr = 3
>>> type(intgr)
<class 'int'>

>>> comp_sum = comp + intgr
>>> print(comp_sum)
(7+2j)

>>> flt = 2.1
>>> comp_sum = comp + flt
>>> print(comp_sum)
(6.1+2j)
```

4.3 Boolean Data Type

The *boolean* data type is either True or False. In Python, boolean variables are defined by the True and False key words.

```
>>> a = True
>>> type(a)
<class 'bool'>

>>> b = False
>>> type(b)
<class 'bool'>
```

Note that True and False must have an Upper Case first letter. Using a lowercase true returns an error.

```
>> c = true
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'true' is not defined
d = false
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'false' is not defined
```

Ints and Floats as Bools

An int, float or complex number set to zero will always return False. An int, float or complex number set to any other number positive or negative besides zero will return True.

```
>>> zero_int = 0
>>> bool(zero_int)
False
```

```
>>> pos_int = 1
>>> bool(pos_int)
True

>>> neg_flt = -5.1
>>> bool(neg_flt)
True
```

4.4 Strings

Strings are sequences of letters, numbers, spaces and symbols. In Python, strings can be almost any length and can contain spaces. String variables are assigned in Python using quotation marks ' '. Strings can contain blank spaces. A blank space is a valid string character in Python.

```
>>> string = 'z'
>>>> type(string)
<class 'str'>

>>> string = 'Engineers'
>>> type(string)
<class 'str'>
```

Numbers as Strings

Numbers and decimals can be defined as strings too. If a decimal number is defined using quotes ' ', it will be saved as a string rather than as a float. Integers defined using quotes will become strings as well if surrounded with quotes.

```
>>> num = '5.2'
>>> type(num)
<class 'str'>

>>> num = '2'
>>> type(num)
<class 'str'>
```

4.5 Lists

Lists is a data structure in Python that can contain multiple elements of any of the other data types. A list is defined with square brackets [] and commas , between elements.

```
>>> lst = [ 1, 2, 3 ]
>>> type(lst)
list
```

```
>>> lst = [ 1, 5.3, '3rd_Element']
>>> type(lst)
list
```

Indexing Lists

You can pull individual elements out of a list using bracket [] notation. Note that Python lists start with the index zero, not the index 1. For example:

```
>>> lst = ['statics', 'strengths', 'dynamics']
>>> lst[0]
'statics'

>>> lst[1]
'strengths'

>>> lst[2]
'dynamics'
```

Remember! Python lists start indexing at [0] not at [1]. To call the elements in a list with 3 values use: lst[0], lst[1], lst[2].

Colons : are used inside the brackets to denote *all*

```
>>> lst = [2, 4, 6]
>>> lst[:]
[2, 4, 6]
```

Negative numbers can be used as indexes to call the last number of elements in the list

```
>>> lst = [2, 4, 6]
>>> lst[-1]
6
```

The colon operator can also be used to denote *all upto* and *from thru end*.

```
>>> lst = [2, 4, 6]
>>> lst[:2]
[2, 4]
```

```
lst = [2, 4, 6]
lst[2:]
[6]
```

The colon operator can also be used to denote *start : end + 1*. Note that the indexing here is not inclusive. lst[1:3] will return the 2nd element, and 3rd element but not the fourth even though the 3 is used in the index.

Remember! Python indexing is not inclusive. The last element called in an index will not be returned.

4.6 Dictionaries and Tuples

Besides lists, Python has two additional data structures that can store multiple objects. These are **dictionaries** and **tuples**.

Dictionaries

Dictionaries are made up of key: value pairs

Tuples

Tuples are immutable lists

4.7 Summary

In this chapter we reviewed a couple of different data types built-in to Python

Key Terms and Concepts

variable

integer

floating point number

boolean

dictionary

tuple

list

index, indexing

Summary of Python Functions and Commands

Built-in Data Types

| Python Object | Description |
|---------------|--|
| int | integer |
| float | floating point number |
| bool | boolean value: True or False |
| complex | complex number, real and imaginary components |
| str | string, sequence of letters, numbers and symbols |
| list | a Python list |
| dict | a Python dictionary |

| Python Object | Description |
|---------------|-------------------|
| tuple | an immutable list |

Python Functions

| Function | Description |
|----------|--|
| type() | outputs a variable or objects data type |
| len() | return the length of a string |
| str() | converts a float or int into a str (string) |
| int() | converts a float or str into an int (integer) |
| float() | converts an int or str into an float (floating point number) |

Python List Operators

| Operator | Description | Example | Result |
|----------|-------------|----------|-------------|
| [] | indexing | lst[1] | 4 |
| : | start | lst[:2] | [2, 4] |
| : | end | lst[2:] | [6, 8] |
| : | through | lst[0:3] | [2, 4, 6] |

4.8 Review Questions

- 1. Question 1
- 2. Question 2
- 3.
- 4.
- 5.
- 6.
- 7.
- 8.
- 9.

Chapter 5

Jupyter Notebooks

5.1 Introduction

In this chapter you will learn:

- What a Jupyter Notebook is
- How to open a Jupyter Notebook
- How to write code in a Jupyter Notebook
- How to run code in a Jupyter Notebook
- How to write text in a Jupyter Notebook
- How to save and close a Jupyter Notebook

5.2 What is a Jupyter Notebook?

A Jupyter notebook is sort of half way between the Python REPL and a Python module .py file.

Jupyter notebooks run in a web browser like Google Chrome where as .py files are edited with a text editor like notepad. Regular .py files only contain Python commands and comments. Jupyter notebooks contain two types of cells: code cells and markdown cells. Lines of Python code are run in code cells. Markdown cells contain comment like descriptions to describe code cells.

5.3 Why Jupyter Notebooks?

There is a vast array of editors and IDE's (Integrated Development Environments) which can be used to edit and run Python code. Why should engineers learn to use Jupyter Notebooks?

Below is a table of simple text editors and IDE's which can be used to edit and run Python code:

Text Editors

Notepad
Idle
Vim
Sublime Text
Atom

IDE's

PyCharm
Visual Studio Code
Spyder

Jupyter Notebooks provide a quick and streamlined way for engineers to prototype code and quickly share code. Jupyter notebooks also provide a way for engineers to share solutions with team members, supervisors and customers.

5.4 Installing Jupyter

To install **Jupyter Notebooks**, the simplest way is to use the **Anaconda** distribution of Python. Anaconda has **Jupyter Notebooks** pre-installed and no further steps are necessary.

Installing Jupyter on Windows

To install jupyter on Windows, open the Anaconda prompt and type:

```
> conda install jupyter
```

Type y for yes when prompted.

Installing Jupyter on Mac OSX

To install jupyter on Mac OSX, open the OSX terminal and type:

```
$ conda install jupyter
```

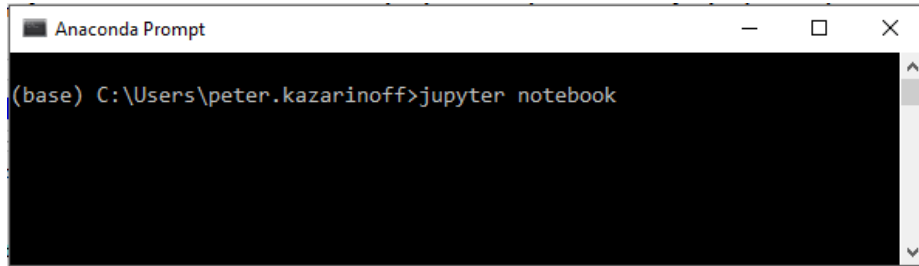
Type y for yes when prompted.

Installing Jupyter on Ubuntu Linux

To install jupyter on Ubuntu Linux, open a terminal and type:

```
$ conda install jupyter
```

Type y for yes when prompted.



Anaconda Prompt Jupyter Notebook

5.5 Opening a Jupyter Notebook

One simple way to open a jupyter notebook on Windows is to click the Windows Start Button in the lower left-hand corner, select the Anaconda Folder, and click Jupyter Notebook.

Open a Jupyter Notebook from the Anaconda prompt

To Open a Jupyter notebook from the Anaconda prompt, first open the Anaconda prompt, then type:

```
> jupyter notebook
```

This will produce output in the Anaconda Prompt Window which looks something like:

A webbrowser will open and something which looks kind of like a file browser will be shown:

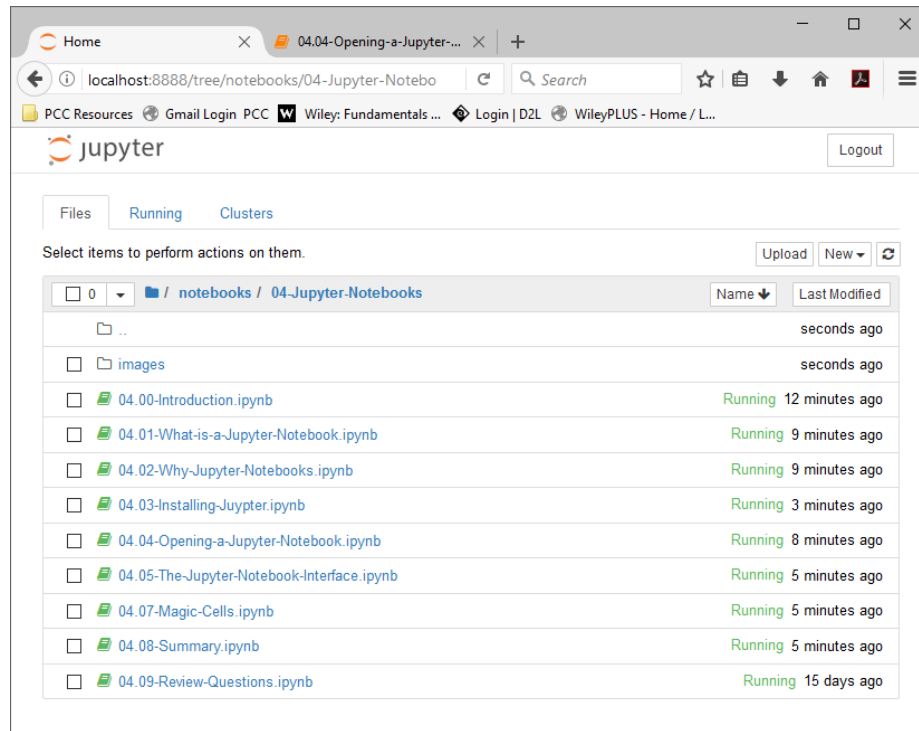
Here you can select an existing jupyter notebook file which will end with the `.ipynb` extension (which stands for IPython Notebook). You can also start a new Jupyter Notebook by clicking the [New] button in the upper right and selecting [Python 3]

5.6 The Jupyter Notebook Interface

When a new notebook opens, you will see the Jupyter Notebook interface. Across the top of the notebook will be the Jupyter icon and the Notebook name. You can click in the notebook name field and change the name of the notebook. Note that the file extension `.ipynb` is not printed in the file name field, but if you look in the Home tab, you will see that the notebook is saved with the `.ipynb` extension.

Menus and Buttons

A jupyter notebook is comprised of a bunch of cells which are arrayed one after another in boxes below the menu items and buttons. There are two main types of cells: Markdown cells and Code cells.



Jupyter File Browser

Code Cells

In code cells you can type Python Code and see the output. An example of a code cell is shown below. Note that the code cell has an the text `In[]` to the left of it.

To run the code in a code cell push the [Run] button or type [Shift]+[Enter]

Markdown Cells

In markdown cells you can type text and headings. Markdown cells are used for documentation and explaining your code. The text in a markdown cell is not executed. Markdown cells can be formatted with a few special characters

Headings

H1 Heading

H2 Heading

H3 Heading

H4 Heading

Code Blocks

For inline code blocks use the ' left quote character, the character to the left of the number 1 and above tab on most keyboards.

This is inline code: ' ' ' Inline code block ' ' ' within a paragraph

For separate code blocks use three ' left quote characters on one line, followed by the code block on separate lines. Terminate the separate code block with a line of three ' left quote characters.

```
'''
```

Separated code blocks

```
'''
```

Bold and italics

Bold and *italic font* is displayed by surrounding text with a double asterisk for **`**bold**`** and a single underscore for *`_italics_`*

`bold**`** produces **bold**

`_italics_` produces *italics*

`**_bold and italic_**` produces ***bold and italic***

Tables

Tables are displayed using the pipe | character, which is [Shift]+[\\] on most keyboards. Columns are separated by pipes and rows are separated by lines. After the header row, a row of pipes and dashes are needed to define the table.

```
| header1 | header 2 | header 3 |
| --- | --- | --- |
| col 1 | col 2 | col 3 |
| col 1 | col 2 | col 3 |
```

produces:

| header1 | header 2 | header 3 |
|---------|----------|----------|
| col 1 | col 2 | col 3 |
| col 1 | col 2 | col 3 |

Bullet Points and Lists

Bullet points are produced using the asterisk character *

```
* item 1
* item 2
* item 3
```

produces

- item 1
- item 2

Numbered lists are produced using sequential numbers followed by a dot. Indent sub-items with two spaces.

```
1. First item
2. Second item
3. Third item
  1. sub item
  2. sub item
    1. sub-sub item
    2. sub-sub item
```

produces

1. First item
 2. Second item
 3. Third item
 4. sub item
 5. sub item
1. sub-sub item
 2. sub-sub item

Horizontal Rule

A horizontal rule is specified with three asterisks on a single line.

```
***
```

produces



Links

Hyperlinks are specified using a set of square brackets [] followed by pair of parenthesis () The text inside the square brackets will be the link, the link address goes in the parenthesis

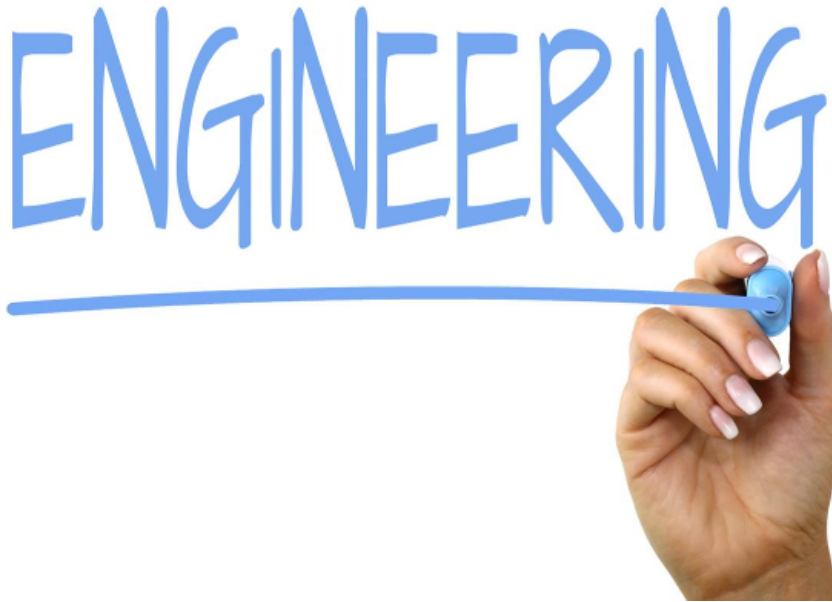
```
[link to docs](https://docs.python.org/3/)
```

produces

[link to docs](https://docs.python.org/3/)

Images

Images are embedded in Jupyter Notebook markdown using the exclamation point and square brackets ![], followed by the image file path in parenthesis (). If the image can not be displayed,



Python Logo

the text in square brackets will be shown. The image can be in the same directory as the notebook, or a relative path can be specified. In this case the image `engineering.png` is stored in the `images` directory, which is a subdirectory of the directory the notebook is saved in.

```
! [Python Logo] (images/engineering.png)
```

produces

LaTeX Math

LaTeX Math equations and symbols are rendered by markdown cells. A more extensive list of LaTeX commands can be found in the appendix.

```
$$ \int_a^b \frac{1}{x^2} dx $$
```

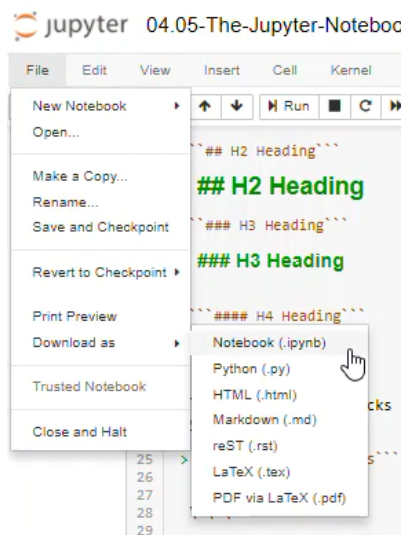
produces

$$\int_a^b \frac{1}{x^2} dx$$

html

Because jupyter notebooks are rendered by web browsers, just about any html tag can be included in the markdown portion of a notebook. An example is the `^{` `}` tags that surround super script text

```
x<sup>2</sup>
```



Jupyter Notebook Export Options

produces

x2

Text can be colored using html `` `` tags

```
<font color=red>Red Text</font>
```

produces

Red Text

warning boxes

bootstrap style warning boxes can be included in jupyter notebook markdown using `<div>` tags

```
<div class="alert alert-danger" role="alert">
  <strong>Warning!</strong> Python lists start at 0
</div>
```

produces

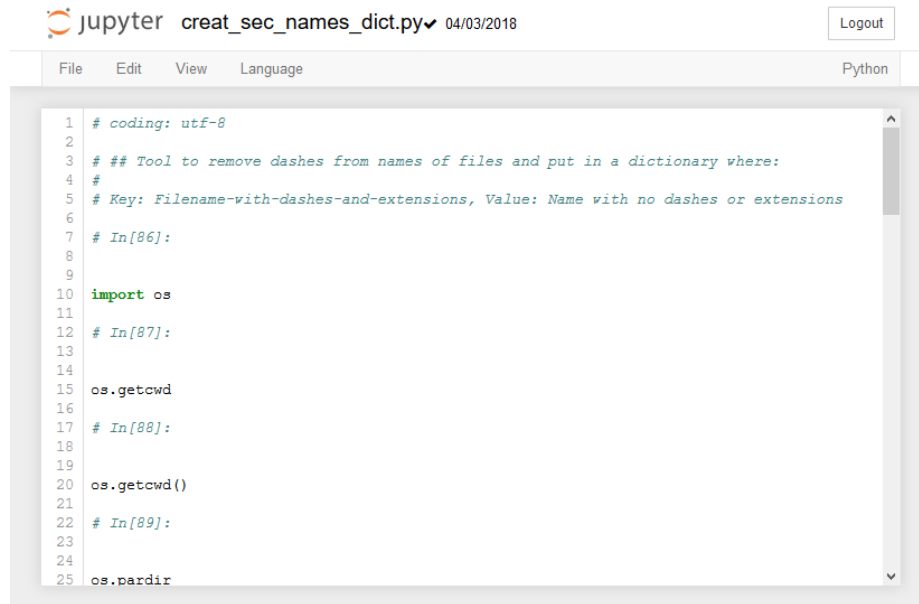
Warning! Python lists start at 0

Saving Jupyter Notebooks in Other Formats

Jupyter notebooks can be saved in other formats besides the native `.ipynb` format. These formats can be accessed on the [File] → [Download As] menu.

The available file types are:

- Notebook (.ipynb) - The native jupyter notebook format



```

1  # coding: utf-8
2
3  ## Tool to remove dashes from names of files and put in a dictionary where:
4  #
5  # Key: Filename-with-dashes-and-extensions, Value: Name with no dashes or extensions
6
7  # In[86]:
8
9
10 import os
11
12 # In[87]:
13
14
15 os.getcwd
16
17 # In[88]:
18
19
20 os.getcwd()
21
22 # In[89]:
23
24
25 os.pardir

```

Markdown Cells as Comments

- Python (.py) - The native Python code file type.
- HTML (.html) - A web page
- Markdown (.md) - Markdown format
- reST (.rst) - Restructured text format
- LaTeX (.tex) - LaTeX Article format
- PDF via LaTeX - a pdf exported from LaTeX, requires a converter

When a Notebook is saved as a .py file, any text in Markdown Cells are converted to comments, and any code cells are kept as Python code.

The .py file after this notebook is Downloaded as a Python(.py) looks like:

5.7 Magic Cells

Jupyter notebook code cells can contain special commands which are not valid Python code, but will affect the behavior of the notebook

%matplotlib inline

One of the most popular magic commands is:

```
%matplotlib inline
```

Using this command at the top of a jupyter notebook will produce matplotlib plots in cells of the notebook. Without %matplotlib inline, plots will jump out as external windows. A typical start to a jupyter notebook using **matplotlib** might start as:

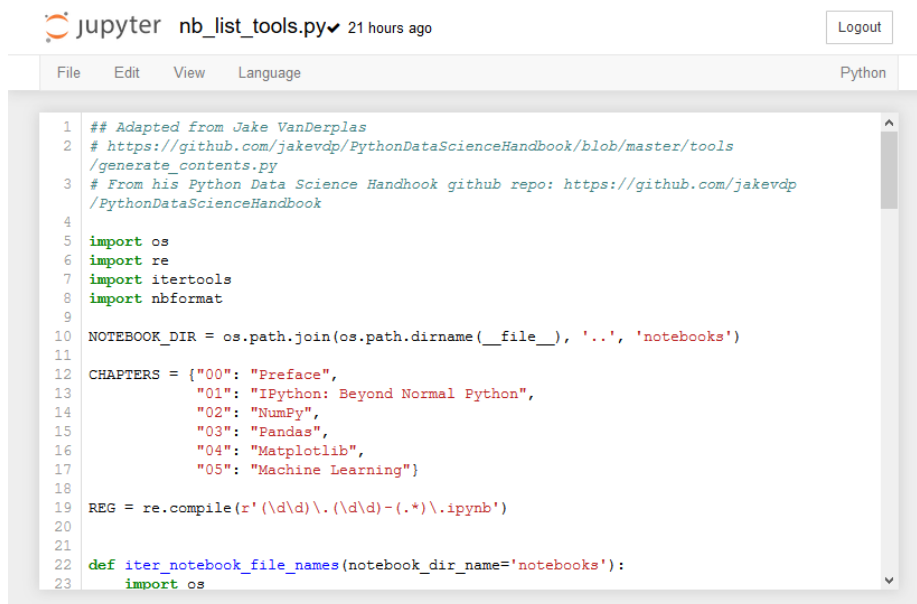


Image of Notebook

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

%run

The `%run` magic command followed by the name of a python file will run the current python file as a script. Suppose the file `hello.py` is created in the same directory as the running jupyter notebook. The directory structure will look something like this:

```
| folder
---| notebook.ipynb
---| hello.py
```

In the file `hello.py` is the code:

```
print('This code was run from a seperate Python file')
print('Hellow from the file hello.py')
```

Within our jupyter notebook, if we `%run` this file, we will get the output of or `hello.py` script in a jupyter notebook output cell.

```
In [3]: %run hello.py
```

```
This code was run from a seperate Python file
Hellow from the file hello.py
```

```
In [10]: %pwd
```

```
Out[10]: 'C:\\Users\\peter.kazarinoff\\Documents\\book\\notebooks\\04-Jupyter-Notebooks'
```

```
In [11]: %ls
```

```
Volume in drive C is Windows
```

```
Volume Serial Number is A048-4C53
```

```
Directory of C:\\Users\\peter.kazarinoff\\Documents\\book\\notebooks\\04-Jupyter-Notebooks
```

```
04/19/2018  06:01 PM    <DIR>          .
04/19/2018  06:01 PM    <DIR>          ..
04/18/2018  12:24 PM    <DIR>          .ipynb_checkpoints
04/18/2018  12:17 PM                1,164 04.00-Introduction.ipynb
04/18/2018  12:21 PM                1,125 04.01-What-is-a-Jupyter-Notebook.ipynb
04/18/2018  12:21 PM                1,477 04.02-Why-Jupyter-Notebooks.ipynb
04/18/2018  12:26 PM                1,901 04.03-Installing-Jupyter.ipynb
04/18/2018  02:37 PM                2,029 04.04-Opening-a-Jupyter-Notebook.ipynb
04/19/2018  07:23 AM                4,196 04.05-The-Jupyter-Notebook-Interface.ipynb
04/19/2018  06:01 PM            11,102 04.07-Magic-Cells.ipynb
04/18/2018  12:24 PM                1,325 04.08-Summary.ipynb
04/03/2018  04:11 PM                 970 04.09-Review-Questions.ipynb
04/19/2018  05:53 PM                 94 hello.py
04/18/2018  02:36 PM    <DIR>          images
                10 File(s)                25,383 bytes
                4 Dir(s)  133,241,610,240 bytes free
```

Other usefull magic commands

Other usefull magic commands are:

| magic command | result |
|---------------|--|
| %pwd | print the current working directory |
| %cd | change the current working directory |
| %ls | list the contents of the current directory |
| %history | the history of the In []: commands |

You can list all of the available magic commands by typing and running %lsmagic in a jupyter notebook code cell:

```
In [12]: %lsmagic
```

```
Out[12]: Available line magics:
```

```
%alias %alias_magic %autocall %automagic %autosave %bookmark %cd %clear %cls %
```

```
Available cell magics:
%%! %%HTML %%SVG %%bash %%capture %%cmd %%debug %%file %%html %%javascript %%

Automatic is ON, % prefix IS NOT needed for line magics.
```

5.8 Summary

In this chapter we learned...

Key Terms and Concepts

- Kernal
- Notebook
- Jupyter
- iPython
- Execute
- .ipynb file
- backend

Python Commands and Functions

Jupyter Notebook Magic Commands

| Command | Description |
|--------------------|---|
| %matplotlib inline | Display plots in output cells |
| %run file.py | Runs file.py and displays output |
| %pwd | Prints the working directory file path |
| %ls | List contents of the current working directory |
| %precision | sets float point precision for pretty printing |
| %whos | lists variables and types in the running kernel session |
| function? | Display help on a function |
| function?? | Display source code of a function |

5.9 Review Questions

- 1.
- 2.
- 3.

- 4.
- 5.
- 6.
- 7.
- 8.

Chapter 6

Functions and Modules

6.1 Introduction

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

- * `import functions and use them in scripts`
- * `create your own functions`
- * `run functions from other files in your scripts`
- * `create functions with default arguments`
- * `provide reusable code for other engineers to use`

6.2 Why Functions?

Functions are reusable pieces of code. Functions are an essential part of most programming languages. Functions provide a couple of benefits:

- Functions allow the same piece of code to be run multiple times
- Functions can break a long program up into smaller pieces
- Functions can be shared with other programmers

6.3 First Function

For our first function we will create a simple function that adds two to any number. Functions in Python typically contain at least two lines. The first line defines the function name and arguments.

```
def function_name(arguments):
    <code>
    return output
```

This line contains a couple parts:

```
def
```

The key word `def` needs to be the start of the line that declares the function. `Def` stands for *definition* and indicates to the Python interpreter that a function definition will follow.

```
function_name
```

Each function needs a name. The function name should start with a letter and is typically all lowercase (in Python names that start with Uppercase are usually used to define *Classes*). Function names need to start with a letter and can only contain letters, numbers and the underscore character. Just about any name will do, but it is best to avoid using any Python key words such as `def`, `class`, `if`, `else`, `for`. A complete list of reserved Python keywords is in the index.

```
(argument):
```

Function names are followed by a set of parenthesis (). Many functions have variable names, called *arguments* in between the parenthesis. The name used for the function argument should be used in the body of the function and is only local to that function. After the parenthesis comes a : colon. A colon is required to end the first line of a function.

A colon : is required at the end of the first line of every function. If the : is not present the code will not run.

```
<code>
```

The body of the function contains the code that will run when the function is called. Any variables declared by the function arguments can be used in the body of the function. Any variables used in the body of the function are *local variables*. Local variables can not be called or accessed by other scripts.

```
return
```

The `return` key word is often the last line of a function. `return` indicates that whatever expression that follows will be the output of the function. The `return` keyword is not a function or a method and parenthesis are not used after `return`, just a space

```
output
```

Whatever expression is included after `return` will be *returned* or outputted by the function. The output expression after `return` can be a single variable, value or be a more complex expression that includes multiple variables.

A simple first function, is a function that will add two to any number. The function will be called `plustwo` and have one input argument, a number. The function will return that number plus 2. So the function should operate as shown below:

```
plustwo(3)
5
```



```
In [1]: def plustwo(n):
        n = n + 2
        return n
```

```
In [2]: plustwo(3)
```

```
Out[2]: 5
```

```
In [3]: plustwo(10)
```

```
Out[3]: 12
```

6.4 Functions with Multiple Arguments

Functions can be written to accept multiple arguments. When multiple arguments are specified, they are listed within the parenthesis after the function name and separated by a comma:

```
def function_name(argument1, argument2):
    <code>
    return output
```

An example function that adds that finds the area of a triangle given the base and height would accept two arguments base and height.

```
In [4]: def triarea(base, height):
        area = 0.5 * base * height
        return area
```

```
In [3]: triarea(10,5)
```

```
Out[3]: 25.0
```

6.5 Functions with Default Arguments

Functions can be specified with default arguments. If values for these arguments are not supplied when the function is called, the default values will be used. The general format is below:

```
def function_name(argument1=default_value, argument2=default_value):
    <code>
    return output
```

An example function is one that calculates the distance an object falls based on time. The general formula for fall distance d based on fall time t can be modeled as:

$$d = \frac{1}{2}gt^2$$

Where g is the acceleration due to gravity. On earth the value of g is 9.81 m/s². But on the moon, g is about 1.625 m/s². Our function will include the default value for earth's gravity and give programmers the option of specifying a different value for g if they choose.

```
In [5]: def falldist(t, g=9.81):
        d = 0.5 * g * t**2
        return d
```

On earth, a ball that falls for three seconds, can be calculated using `falldist(3)` and leaving out a value for g .

```
In [6]: falldist(3)
```

```
Out[6]: 44.145
```

On the moon, gravity is much weaker, an acceleration of 1.625 m/s². To calculate how far a ball falls on the moon in three seconds, two arguments need to be supplied 3 and 1.625. If a second argument is given, this overrides the default value assigned in the first line of the function.

```
In [8]: falldist(3, 1.625)
```

```
Out[8]: 7.3125
```

6.6 Calling Functions from Other Files

User-defined functions can be called from other files, a different file than where the function definition was written. If a new file called *myfunctions.py* is created and contains two function definitions, `plustwo()` and `falldist()`, these functions can be used by a separate file as long as the file and function names are imported first. It is important that the file which contains the functions ends in the *.py* extension. Without a *.py* extension, the file can not be imported.

```
# myfunctions.py

def plustwo(n):
    n = n + 2
    return n

def falldist(t,g=9.81):
    d = 0.5 * g * t**2
    return d
```

This file can be imported into another script, or Jupyter Notebook running in the same directory. Using `from <filename> import <function_name>`. Note that although the file name must contain a *.py* extension, *.py* is not used as part of the file name during import.

```
In [1]: from myfunctions import plustwo
        plustwo(3)
```

```
Out[1]: 5
```

```
In [2]: from myfunctions import falldist
        falldist(3)
```

```
Out[2]: 44.145
```

6.7 Docstrings in Functions

It is good programming practice to document your code. Reusable chunks of code are particularly important to document as other programmers may use the code and you may use the code again at a different time.

Python has a couple different ways for programmers to add documentation. One way is to use simple comments. Comments are lines of code that do not get run by the Python interpreter. Comments are meant to be viewed by humans. In Python, comment lines start with the pound symbol #.

Another way to document code is to use docstrings. Docstrings are comments which are surrounded with triple quotation marks and usually contain multiple lines of explanation. A function containing a docstring takes the form:

```
def function_name(arguments):
    """
    Docstring text

    """
    <code>

    return <output>
```

Doc strings are what come up when the `help()` function is called. As an example, running the `help()` function on the builtin function `sum` brings up:

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

Help on built-in function sum in module builtins:

```
sum(iterable, start=0, /)
    Return the sum of a 'start' value (default: 0) plus an iterable of numbers

    When the iterable is empty, return the start value.
    This function is intended specifically for use with numeric values and may
    reject non-numeric types.
```

We can produce the same type of output when a user types `help()` by adding docstrings to a function.

Let's create our own function to convert g to kg. Let's call our function `g2kg`. The first thing to do is make sure that our function name is not assigned to another function or is used as a keyword by Python. We can check quick using `type()`. We know that `sum()` is a function, how about `g2kg`?

```
In [2]: print(type(sum))
        print(type(g2kg))
```

```
<class 'builtin_function_or_method'>
```

```
-----
NameError                                Traceback (most recent call last)

<ipython-input-2-487fcfc6eb43> in <module>()
      1 print(type(sum))
----> 2 print(type(g2kg))

NameError: name 'g2kg' is not defined
```

Since `g2kg` is not defined, we can use `g2kg` as the name of a user-defined function. Once we know that our Remember the parenthesis, colon and return statment.

```
In [3]: def g2kg(g):
        kg = g/1000

        return kg
```

Now let's try and use our function. How many kg's is 1300 grams. We expect the output to be 1.3 kg

```
In [4]: g2kg(1300)
```

```
Out[4]: 1.3
```

If we call `help()` on our function, nothing will be returend because our function does not contain a docstring yet.

```
In [5]: help(g2kg)
```

```
Help on function g2kg in module __main__:
```

```
g2kg(g)
```

Now let's add a docstring to the function. Common components of docstrings include a summary of the function. What the function inputs and outputs are and an example of the function running.

```
In [6]: def g2kg(g):
        """
        Function g2kg converts between g and kg

        input: a measurement in grams, int or float
        output: measurment in kg, float

        Example:

        >>> g2kg(1300)

        1.3

        """
        kg = g/1000
        return kg
```

Now let's ask for help on our function and see if the docstring is printed back.

```
In [7]: help(g2kg)
```

```
Help on function g2kg in module __main__:
```

```
g2kg(g)
  Function g2kg converts between g and kg

  input: a measurement in grams, int or float
  output: measurment in kg, float

  Example:

  >>> g2kg(1300)

  1.3
```

6.8 Positional and Keyword Arguments

Python functions can contain two types of arguments: *positional arguments* and *keyword arguments*.

Positional Arguments

An *argument* is a variable, value or object passed to a function or method. *Positional arguments* are arguments that need to be included when a function is called in the proper position or order. The first positional argument always needs to be listed first, the second positional argument needs to be listed second, the third positional argument listed third etc.

An example of positional arguments can be seen in Python's `complex()` function. This function returns a complex number with a real term and an imaginary term. The order that numbers are passed to the complex function determines which number is the real term and which term is the imaginary term.

If the complex number $3 + 5j$ needs to be created, the two positional arguments are the numbers 3 and 5. The 3 must be listed first and the 5 must be listed second.

```
In [1]: complex(3, 5)
```

```
Out[1]: (3+5j)
```

On the other hand, if the complex number $5 + 3j$ needs to be created, the 5 needs to be listed first and the 3 listed second. Writing the same arguments in a different order produces a different result.

```
In [2]: complex(5, 3)
```

```
Out[2]: (5+3j)
```

Positional Arguments specified an iterable

Positional arguments can also be passed to functions and methods using an iterable object. Examples of iterable objects include lists, tuples and sets. The general syntax to use is:

```
function(*iterable)
```

Where `function` is the name of the function and `iterable` is the name of the iterable preceded by the ampersand `*` character. An example of using a list to pass positional arguments to the `complex()` function is below:

```
In [3]: term_list = [3, 5]
        complex(*term_list)
```

```
Out[3]: (3+5j)
```

Keyword Arguments

A *keyword argument* is an argument passed to a function or a method which is preceded by a *keyword* and an equals sign the general form is:

```
function(keyword=value)
```

Where function is the function name, keyword is the keyword argument and value is the value or object passed as that keyword. Python's complex function can also accept two keyword arguments. The two keyword arguments are `real=` and `imag=`. To create the complex number $3 + 5j$ the, 3 and 5 can be passed to the function as the values assigned to the keyword arguments `real=` and `imag=`.

```
In [4]: complex(real=3, imag=5)
```

```
Out[4]: (3+5j)
```

Keyword Arguments specified by a dictionary

Keyword arguments can also be passed to functions and methods using dictionary. The dictionary used must contain the keywords as keys and the values as values. The general form is:

```
keyword_dict = {'keyword1': value1, 'keyword2': value2}
function(**keyword_dict)
```

Where function is the name of the function and keyword_dict is the name of the dictionary containing keywords and values preceded by the double ampersand `**` character. Note that the keywords when assigned as keys in a dictionary must be surrounded by quotes `' '`. An example of using a dictionary to pass keyword arguments to the `complex()` function is below:

```
In [6]: keyword_dict = {'real': 3, 'imag': 5}
        complex(**keyword_dict)
```

```
Out[6]: (3+5j)
```

```
In [14]: import typing
         typing.get_type_hints()
```

```
Out[14]: {}
```

```
In [15]: import sys
         sys.getrefcount(keyword_dict)
```

```
Out[15]: 2
```

6.9 Summary

Key Terms and Concepts

function

function definition

arguments

default arguments

positional arguments
keyword arguments
output
doc string
return
.py file
import

Python Commands

| Command | Description |
|---------|--|
| def | define a function |
| return | define the expression or value a function outputs |
| import | import a module or .py file |
| from | import a function or class from a module or .py file |
| as | name an alias for a function, method or class |
| """ | define a doc string |

6.10 Review Questions

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.

Chapter 7

Plotting with Matplotlib

7.1 Introduction

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

- Import matplotlib into a Python script or jupyter notebook
- Construct line plots with matplotlib
- Construct bar charts with matplotlib
- Add error bars to bar charts
- Plot histograms
- Plot contours
- Construct 3D mesh grid plots

7.2 What is Matplotlib?

Matplot lib is a popular python package used to plot data.

7.3 Installing Matplotlib

Before **matplotlib**'s plotting functions can be used. **Matplotlib** needs to be installed. Depending on which distribution of Python is installed on your machine, the installation methods are slightly different.

Installing Matplotlib with the Anaconda Prompt

To install **matplotlib**, open the **Anaconda Prompt** and type:

```
> conda install matplotlib
```

Type y for yes when prompted.

Installing Matplotlib with pip

To install **matplotlib** with pip, bring up a terminal window and type:

```
$ pip install matplotlib
```

This will install **matplotlib** in the current working python environment.

Verify the installation

To verify that **matplotlib** is installed try to invoke **matplotlib's** version at the Python REPL using the `.__version__` attribute common to most Python packages.

```
In [1]: >>> import matplotlib
        >>> matplotlib.__version__
```

```
Out[1]: '2.2.2'
```

7.4 Line Plots

Line plots in **matplotlib** can be created using **matplotlib's** **pyplot** library

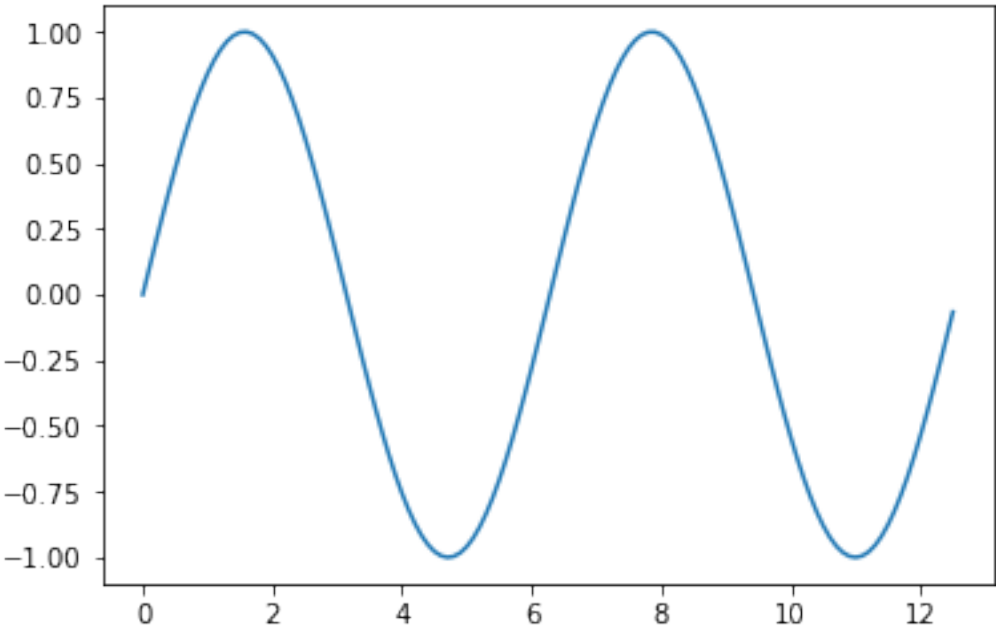
To build a line plot, first import **matplotlib**. It is common convention to import **matplotlib's** **pyplot** library as `plt`. The `plt` alias will be familiar to other Python users.

If using a jupyter notebook include the line `%matplotlib inline` command after the imports. `%matplotlib inline` is a jupyter notebook magic command which causes **matplotlib** plots to display directly inside jupyter notebook output cells.

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
        # include if using a jupyter notebook
        %matplotlib inline
```

```
In [2]: x = np.arange(0, 4 * np.pi, 0.1)
        y = np.sin(x)
```

```
In [3]: plt.plot(x, y)
        plt.show()
```



Features of a matplotlib plot

A variety of features on a **matplotlib** plot can be specified. The following is a list of commonly specified features:

Line Color, Line Width, Line Style, Line Opacity and Marker Options

The color, width and style of line in a plot can be specified. Line color, line width and line style are included as extra arguments in the `plt.plot()` function call.

```
plt.plot(<x-data>,<y-data>, linewidth=<float or int>, linestyle='<linestyle abbr>')
```

An example `plt.plot()` line including line color, line width and line style options is:

```
plt.plot(x, y, linewidth=2.0, linestyle='+', color='b', alpha=0.5, marker='o')
```

Below is a list of linewidths (many other widths are also available)

| linewidth=<float or int> | Line Width |
|--------------------------|-----------------|
| 0.5 | 0.5 pixels wide |
| 1 | 1 pixel wide |
| 1.5 | 1.5 pixels wide |
| 2 | 2 pixels wide |
| 3 | 3 pixels wide |

Below is a list of line styles

| linestyle='<style abbreviation>' | Line Style |
|----------------------------------|----------------------|
| '-' or 'solid' | solid line (default) |
| '--' or 'dashed' | dashed line |
| '-.' or 'dashdot' | dash-dot line |
| ':' or 'dotted' | dotted line |
| 'None' or ' ' or '' | no line |

Below is a list of color abbreviations. Note 'b' is used for blue and 'k' is used for black.

| color = '<color abbreviation>' | Color Name |
|--------------------------------|------------|
| 'b' | Blue |
| 'c' | Cyan |
| 'g' | Green |
| 'k' | Black |
| 'm' | magenta |
| 'r' | Red |
| 'w' | White |
| 'y' | Yellow |

Below is a list of alpha (opacity) values (any alpha value between 0.0 and 1.0 is possible)

| alpha = <float or int> | Opacity |
|------------------------|------------------|
| 0 | transparent |
| 0.5 | Half transparent |
| 1.0 | Opaque |

Colors can also be specified in hexadecimal form surrounded by quotation marks like '#FF69B4' or in RGBA (red,green,blue,opacity) color surrounded by parenthesis like (255,182,193,0.5).

| color = '<color abbreviation>' | Color Format |
|--------------------------------|--------------|
| '#FF69B4' | hexadecimal |
| (255,182,193,0.5) | RGBA |

Below is a list of maker styles

| marker='<marker abbreviation>' | Marker Style |
|--------------------------------|---------------|
| '.' | point |
| ',' | one pixel |
| 'o' | circle |
| 'v' | triangle_down |
| '^' | triangle_up |

| <code>marker='<marker abbreviation>'</code> | Marker Style |
|---|--------------|
| <code>"g"</code> | octagon |
| <code>"s"</code> | square |
| <code>"p"</code> | pentagon |
| <code>"*"</code> | star |
| <code>"h"</code> | hexagon 1 |
| <code>"H"</code> | hexagon 2 |
| <code>"+"</code> | plus |
| <code>"P"</code> | filled plus |
| <code>"x"</code> | x |
| <code>"X"</code> | filled x |
| <code>"D"</code> | diamond |
| <code>"d"</code> | thin diamond |

In addition to `marker='<marker style>'`, the color of the marker edge, the color of the marker face and the size of the marker can be specified with:

```
plt.plot( .... markeredgecolor='<color abbreviation>', markerfacecolor='<color abbreviation>', ma
```

Title

The plot title will be shown above the plot. The `title()` command accepts a string as an argument

```
plt.title('My Title')
```

x-axis label

The x-axis label will be down below the x-axis. The `xlabel()` command accepts a string as an argument.

```
plt.xlabel('My x-axis label')
```

y-axis label

The y-axis label will be shown to the left of the y-axis. The `ylabel()` command accepts a string as an argument.

```
plt.ylabel('My y-axis label')
```

Legend

The legend will appear within the plot area, in the upper right corner by default. The `legend` command accepts a list of strings and optionally accepts a `loc=` argument to position the legend in a different location

```
plt.legend(['entry1','entry2'], loc = 0)
```

The following are the location codes for legend location. These numbers need to be placed after `loc=` in the `plt.legend()` call.

| Legend Location | loc = <number> |
|-----------------|----------------|
| 'best' | 0 |
| 'upper right' | 1 |
| 'upper left' | 2 |
| 'lower left' | 3 |
| 'lower right' | 4 |
| 'right' | 5 |
| 'center left' | 6 |
| 'center right' | 7 |
| 'lower center' | 8 |
| 'upper center' | 9 |
| 'center' | 10 |

Grid

A grid can be shown on the plot using the `plt.grid()` command. By default, the grid is turned off. To turn on the grid use:

```
plt.grid(True)
```

The only valid options are `plt.grid(True)` and `plt.grid(False)`. Note that `True` and `False` are capitalized and are not enclosed in quotations.

Line Plot with specified features

Line plots with the specified features above are constructed according to this general outline:

Imports

Import `matplotlib.pyplot` as `plt`, as well as any other modules needed to work with the data. If using a jupyter notebook include `%matplotlib inline` in this section

Define Data

Your plot needs to contain something. This is defined after the imports

Plot Data including options

Use `plt.plot()` to plot the data defined above. Note the `plt.plot()` needs to be called before any details are specified. Otherwise the details have no plot to apply to. Besides the data, arguments in `plt.plot()` call can include: `* linewidth= <float or int> * linestyle='<linestyle abbreviation>' * color='<color abbreviation>' * alpha= <float or int> * marker='<marker abbreviation>' * markeredgecolor='<color abbreviation>' * markerfacecolor='<color abbreviation>' * markersize=<float or int>`

Add plot details

Add details such as a title, axis labels, legend and colors. Plot details to add include:

- `plt.title('<title string>')`
- `plt.xlabel('<x-axis label string>')`
- `plt.ylabel('<y-axis label string>')`
- `plt.legend(['list', 'of', 'strings'])`
- `plt.grid(<True or False>)`
- `plt.xticks([list of tick locations, floats or ints], [list of tick labels, strings])`
- `plt.yticks([list of tick locations, floats or ints], [list of tick labels, strings])`

Show the plot

Use the `plt.show()` command to show the plot. This will cause the plot to display in a jupyter notebook or pop out as a new window if using a separate .py script file. Note that the `plt.show()` needs to be called after all of the plot specifications.

A section of code following this outline and the resulting plot is shown below:

```
In [4]: # Imports
import numpy as np
import matplotlib.pyplot as plt
# Include if using a jupyter notebook. Remove if using a .py-file.
%matplotlib inline

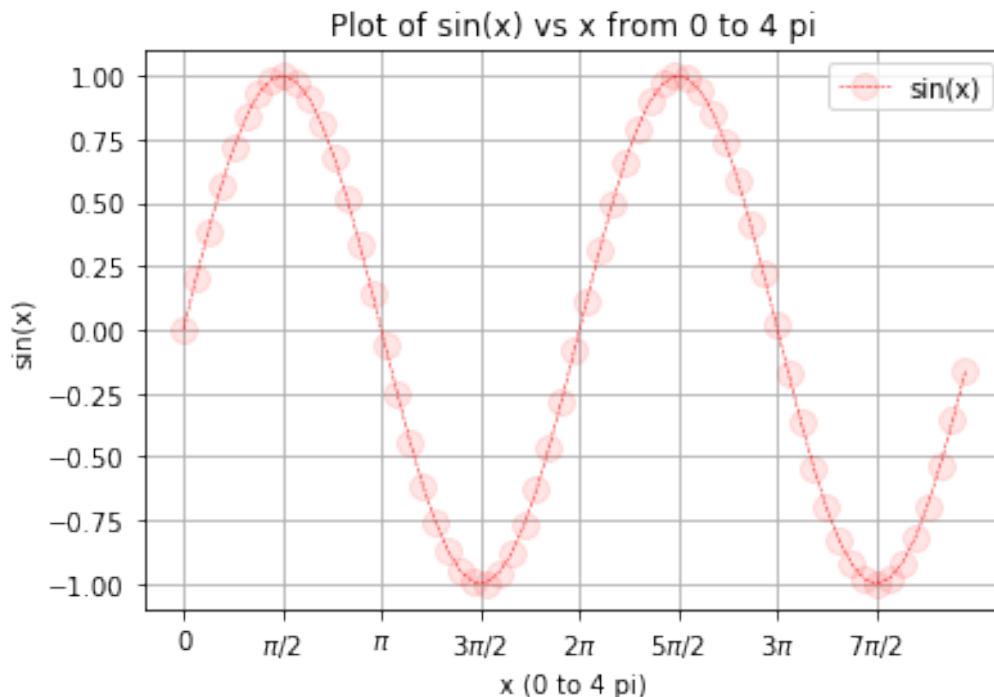
# Define Data
x = np.arange(0, 4 * np.pi, 0.2)
y = np.sin(x)

# Plot Data including options
plt.plot(x, y,
         linewidth=0.5,
         linestyle='--',
         color='r',
         marker='o',
         markersize=10,
         markerfacecolor=(1, 0, 0, 0.1))

# Add details
plt.title('Plot of sin(x) vs x from 0 to 4 pi')
plt.xlabel('x (0 to 4 pi)')
plt.ylabel('sin(x)')
plt.legend(['sin(x)'])
plt.xticks(
    np.arange(0, 4 * np.pi, np.pi / 2),
    ['0', '$\pi$/2', '$\pi$', '$3\pi/2$', '$2\pi$', '$5\pi/2$', '$3\pi$', '$7\pi/2$'])
plt.grid(True)
```



```
# Show the Plot  
plt.show()
```



7.5 Multi Line Plots

Multi-line plots in **matplotlib** can be created using **matplotlib's pyplot** library. This section builds upon the work in the previous section where a plot with one line was created. This section will also introduce **matplotlib's** object oriented approach to building a plot. The object oriented approach to building plots will be used for the rest of the chapter.

matplotlib object-oriented interface

An object-oriented plotting interface is an interface where components of the plot (like the axis, title, lines, markers, etc) are treated as programmatic *objects* that have *attributes* and *methods* associated with them. To create a new *object* is called *instantiation*. Once an object is created, or *instantiated*, the properties of that object can be modified and methods can be called on the object. The basic anatomy of a **matplotlib** plot includes a couple of layers, each of these layers is a programming *object*:

- Figure Objects: The bottom layer. Think of the figure layer as the figure window which

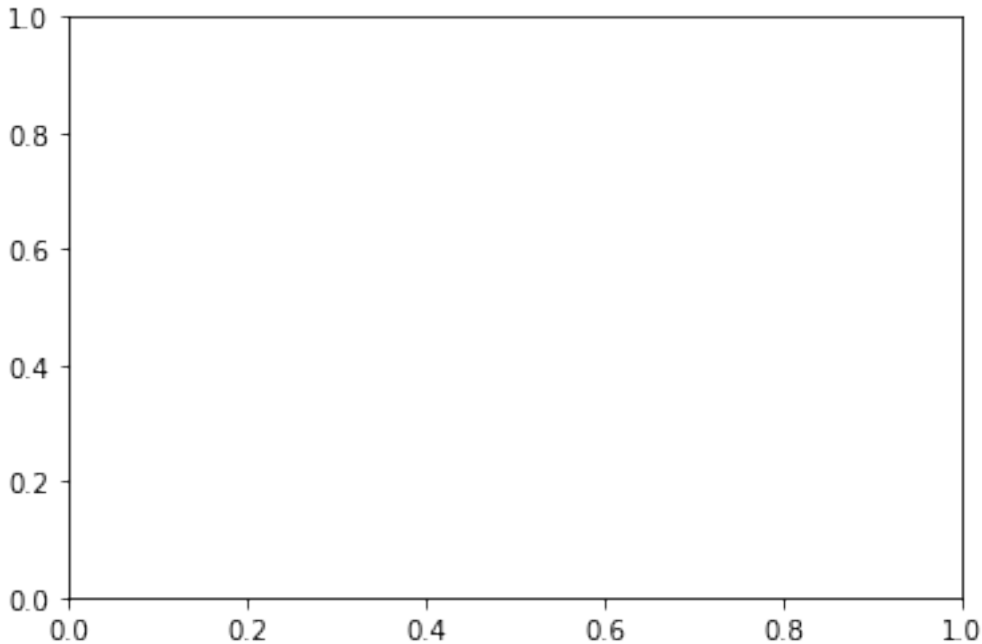
contains the minimize, maximize, close buttons. A figure can contain one plot or multiple plots

- Plot Objects: A plot builds on the Figure layer. If there are multiple plots, each plot is called a subplot.
- Axis Objects: An axis is added to a Plot layer. Axis can be thought of as sets of x and y axis that lines and bars are drawn on. An Axis contains daughter attributes like axis labels, tick labels and line thickness.
- Data Objects: data points, lines, shapes are plotted on an axis

To build a figure object, **matplotlib's** `plt.subplot()` function is used. This functions creates both a figure *object* and an axis *object*. The `plt.subplot()` function *instantiates* a figure *object* and *instantiates* an axis object. For now, we we'll leave the `subplot()` arguments blank. By default the `subplot()` function will create a single figure object and a single axis object. We'll call the figure object `fig` and the axis object `ax`. Note these two outputs of the `plt.subplots()` function are separated by a comma.

```
In [1]: import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

```
In [2]: fig, ax = plt.subplots()
```



We have a figure object and axis object, but the axis object is blank. This produces a blank plot. We can add elements to the axis object to build a plot. Let's create three **numpy** arrays to add to our axis object.

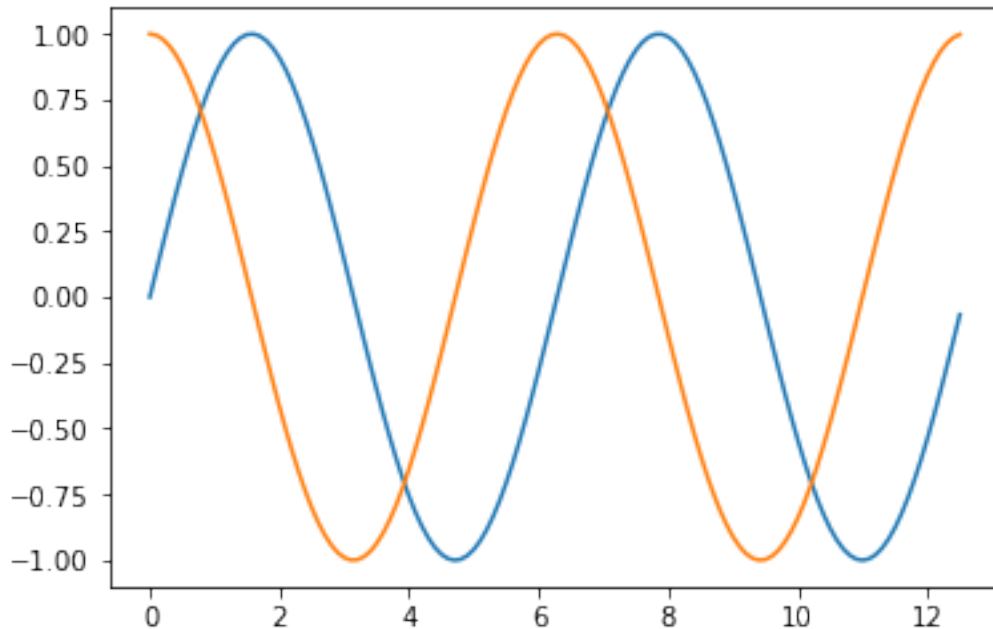
```
In [3]: x = np.arange(0,4*np.pi,0.1)
        y = np.sin(x)
        z = np.cos(x)
```

The **numpy** arrays `x`, `y`, and `z` can be added to our axis object `ax`. We add a plot attribute (a line) to our axis object `ax` using the object oriented structure `<object>.<attribute>`. In this case, `ax` is the object and `plot` is the attribute. The `plt.show()` line shows the plot on the screen.

```
In [4]: fig, ax = plt.subplots()

        ax.plot(x,y)
        ax.plot(x,z)

        plt.show()
```



The `ax` object has many methods and attributes. Two methods we can run on the `ax` object include `ax.set_title()` and `ax.legend()`. A couple daughter objects include `ax.xaxis` and `ax.yaxis`. These daughter objects in turn have methods such as `ax.xaxis.set_label_text()` and `ax.yaxis.set_label_text()`.

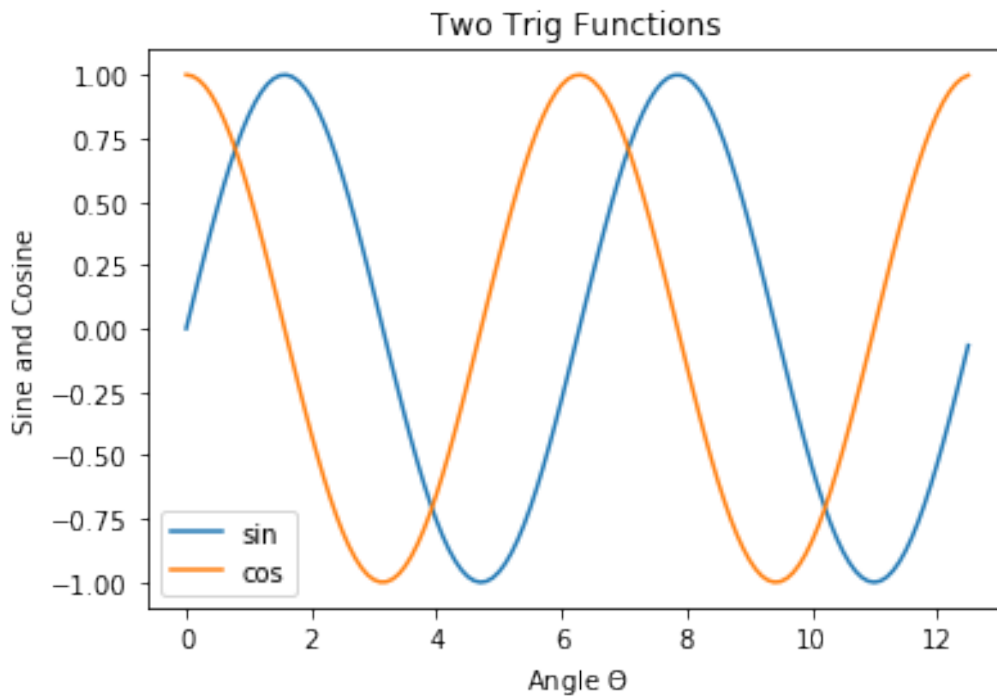
```
In [5]: fig, ax = plt.subplots()

        ax.plot(x,y)
        ax.plot(x,z)
```

```
ax.set_title('Two Trig Functions')
ax.legend(['sin', 'cos'])

ax.xaxis.set_label_text('Angle  $\Theta$ ')
ax.yaxis.set_label_text('Sine and Cosine')

plt.show()
```



7.6 Bar Plots

Bar plots in **matplotlib** can be created using **matplotlib's pyplot** library

To construct a bar plot using **matplotlib**, first import **matplotlib's pyplot** library. The common alias to use for **matplotlib.pyplot** is **plt**. If using a jupyter notebook include the line `%matplotlib inline`

```
In [7]: import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

Bar Charts

To create a bar chart, we need some data. In this case the data is from a set of **coefficient of thermal expansion** lab measurements. The **coefficient of thermal expansion** (CTE) is a material property that describes how much a material will change in length as a result of a change in temperature. Different materials have different CTE's and we can use the lab data to determine which material will expand the most if all three are heated up to the same temperature (assuming all three start at the same temperature).

First we need to input the data as **numpy** arrays:

```
In [8]: # Data
        aluminum = np.array([
            6.4e-5, 3.01e-5, 2.36e-5, 3.0e-5, 7.0e-5, 4.5e-5, 3.8e-5, 4.2e-5, 2.62e-5,
            3.6e-5
        ])
        copper = np.array([
            4.5e-5, 1.97e-5, 1.6e-5, 1.97e-5, 4.0e-5, 2.4e-5, 1.9e-5, 2.41e-5, 1.85e-5,
            3.3e-5
        ])
        steel = np.array([
            3.3e-5, 1.2e-5, 0.9e-5, 1.2e-5, 1.3e-5, 1.6e-5, 1.4e-5, 1.58e-5, 1.32e-5,
            2.1e-5
        ])
```

Next we will calculate the average or **mean** using **numpy's** `np.mean()` function.

```
In [9]: # Calculate the average
        aluminum_mean = np.mean(aluminum)
        copper_mean = np.mean(copper)
        steel_mean = np.mean(steel)
```

Then we will build a list of materials and CTE's

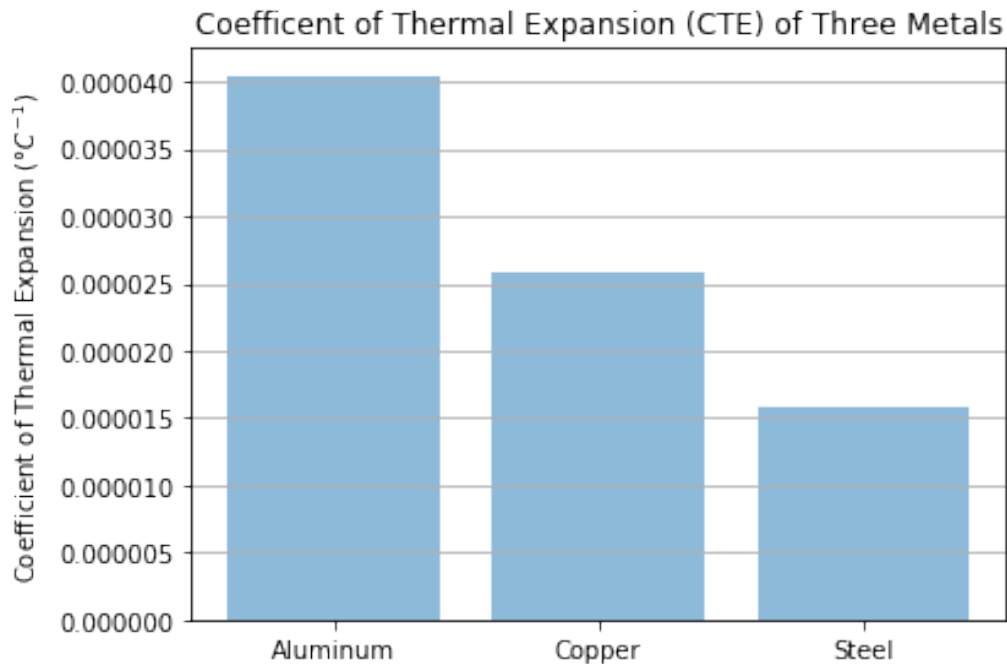
```
In [10]: # Create Arrays for the plot
        materials = ['Aluminum', 'Copper', 'Steel']
        x_pos = np.arange(len(materials))
        CTEs = [aluminum_mean, copper_mean, steel_mean]
```

With the materials, `x_pos`, and CTEs (the labels below the bars) defined, the plot can be built using the `ax.bar()` method. The `ax.bar()` method requires two positional arguments, a list of bar positions and a list of bar heights. In our case `x_pos` is the list of positions and `CTEs` is the list of bar heights.

```
In [11]: # Build the plot
        fig, ax = plt.subplots()
        ax.bar(x_pos, CTEs, align='center', alpha=0.5)
        ax.set_ylabel('Coefficient of Thermal Expansion ($\text{degree C}^{-1}$)')
        ax.set_xticks(x_pos)
```

```
ax.set_xticklabels(materials)
ax.set_title('Coefficient of Thermal Expansion (CTE) of Three Metals')
ax.yaxis.grid(True)

# Save the figure and show
plt.tight_layout()
plt.savefig('bar_plot.png')
plt.show()
```



Pie Charts

Pie charts can be constructed with **matplotlib's** `ax.pie()` method. The one required positional argument is a list of pie piece sizes. Optional keyword arguments include a list of piece labels (`label=`) and if the percentages will be auto calculated and in what format (`autopct=`).

The data we will plot is from the number of students that choose different engineering majors each year.

Each year there are approximately:

15,000 civil engineering graduates

50,000 electrical engineering graduates

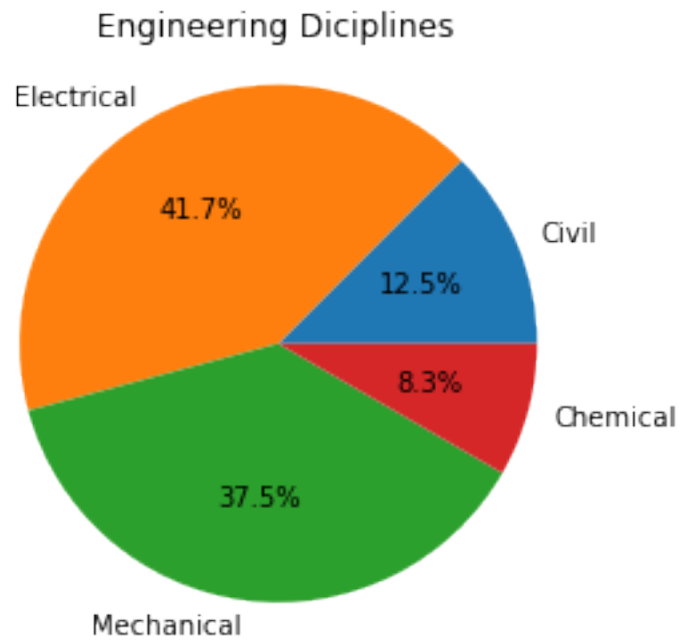
45,000 mechanical engineering graduates

10,000 chemical engineering graduates

```
In [13]: # Pie chart, where the slices will be ordered and plotted counter-clockwise:
labels = ['Civil', 'Electrical', 'Mechanical', 'Chemical']
sizes = [15, 50, 45, 10]

fig, ax = plt.subplots()
ax.pie(sizes, labels=labels, autopct='%1.1f%%')
ax.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.
ax.set_title('Engineering Diciplines')

plt.show()
```



```
In [15]: # Pie chart, where the slices will be ordered and plotted counter-clockwise
labels = ['Civil', 'Electrical', 'Mechanical', 'Chemical']
sizes = [15, 30, 45, 10]
explode = (
    0.1, 0.1, 0.1, 0.4
) # "explode, and highlight 4th entry 'Logs' by offsetting it a greater amount"

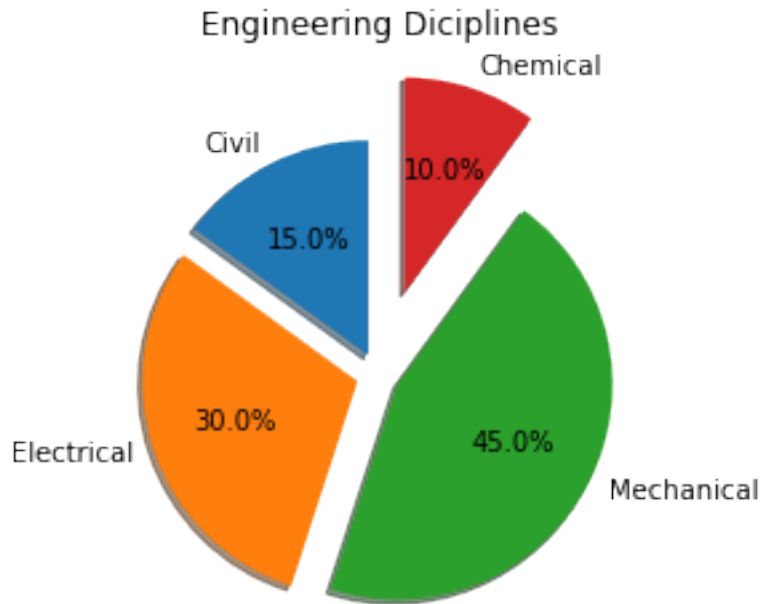
fig, ax = plt.subplots()
ax.pie(
    sizes,
    explode=explode,
    labels=labels,
    autopct='%1.1f%%',
```

```

        shadow=True,
        startangle=90)
ax.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.
ax.set_title('Engineering Diciplines')

plt.show()

```



7.7 Error Bars

Error bars can be created with **matplotlib** and applied to both line plots and bar plots.

To construct a bar plot with error bars, first import **matplotlib**. If using a jupyter notebook, include the line `%matplotlib inline`

```

In [1]: import matplotlib.pyplot as plt
        import numpy as np
        %matplotlib inline

```

```

In [2]: # Data
        aluminum = np.array([6.4e-5 , 3.01e-5 , 2.36e-5, 3.0e-5, 7.0e-5, 4.5e-5, 3.8e-5, 4.2e-5,
        copper = np.array([4.5e-5 , 1.97e-5 , 1.6e-5, 1.97e-5, 4.0e-5, 2.4e-5, 1.9e-5, 2.41e-5 ,
        steel = np.array([3.3e-5 , 1.2e-5 , 0.9e-5, 1.2e-5, 1.3e-5, 1.6e-5, 1.4e-5, 1.58e-5, 1.3

```



```

# Calculate the average
aluminum_mean = np.mean(aluminum)
copper_mean = np.mean(copper)
steel_mean = np.mean(steel)

# Calculate the standard deviation
aluminum_std = np.std(aluminum)
copper_std = np.std(copper)
steel_std = np.std(steel)

# Create Arrays for the plot
materials = ['Aluminum', 'Copper', 'Steel']
x_pos = np.arange(len(materials))
CTEs = [aluminum_mean, copper_mean, steel_mean]
error = [aluminum_std, copper_std, steel_std]

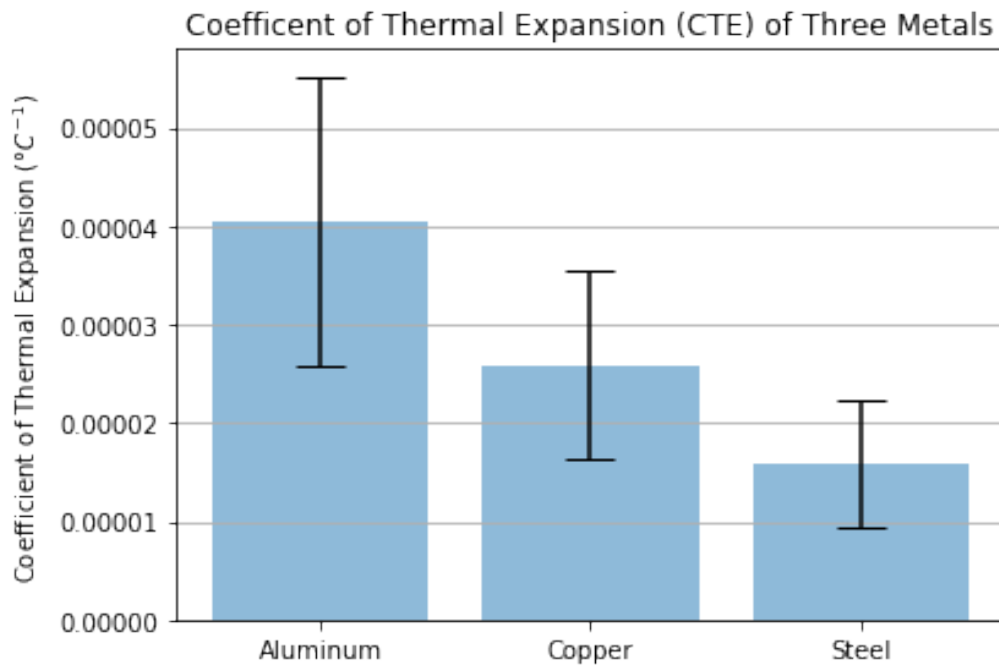
```

```

In [3]: # Build the plot
fig, ax = plt.subplots()
ax.bar(x_pos, CTEs, yerr=error, align='center', alpha=0.5, ecolor='black', capsize=10)
ax.set_ylabel('Coefficient of Thermal Expansion ( $\text{degree C}^{-1}$ )')
ax.set_xticks(x_pos)
ax.set_xticklabels(materials)
ax.set_title('Coefficient of Thermal Expansion (CTE) of Three Metals')
ax.yaxis.grid(True)

# Save the figure and show
plt.tight_layout()
plt.savefig('bar_plot_with_errorBars.png')
plt.show()

```

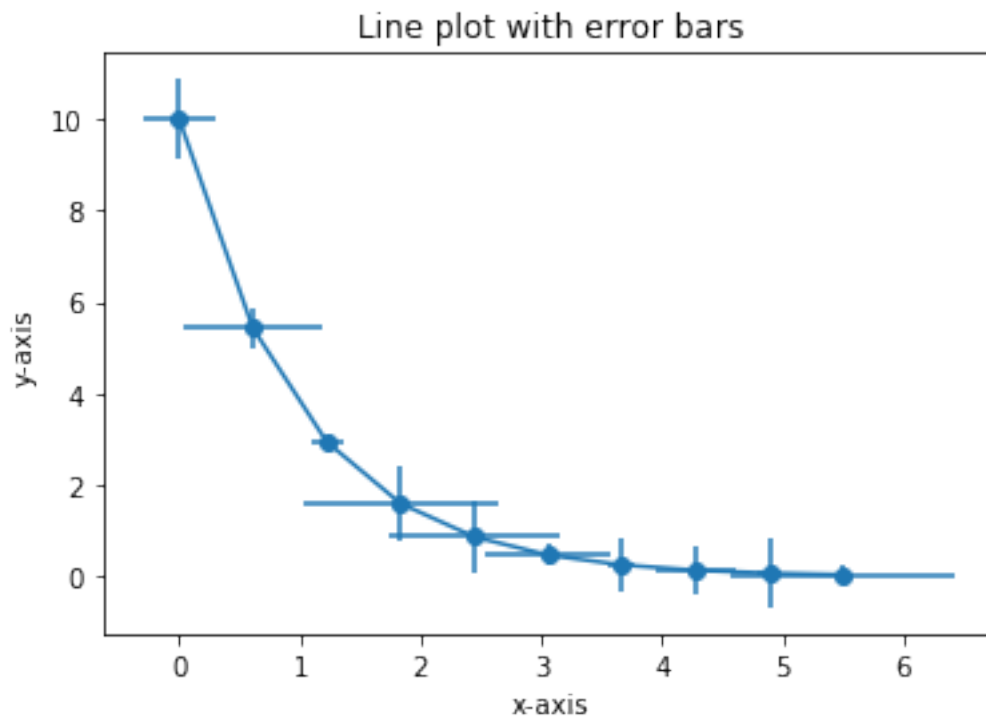


Error bars in line plots

Error bars can also be added to line plots.

```
In [4]: #x = np.array([0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0])
x = np.linspace(0,5.5,10)
y = 10*np.exp(-x)
xerr = np.random.random_sample((10))
yerr = np.random.random_sample((10))
```

```
In [5]: fig, ax = plt.subplots()
ax.errorbar(x, y, xerr=xerr, yerr=yerr, fmt='-o')
ax.set_xlabel('x-axis')
ax.set_ylabel('y-axis')
ax.set_title('Line plot with error bars')
plt.show()
```



7.8 Histograms

Histogram plots can be created with **matplotlib**

First import matplotlib. If using a jupyter notebook include the line `%matplotlib inline`

```
In [1]: import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

```
In [6]: import numpy as np
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
```

```
mu = 80
sigma = 7
x = np.random.normal(mu, sigma, size=200)
```

```
fig, ax = plt.subplots()
```

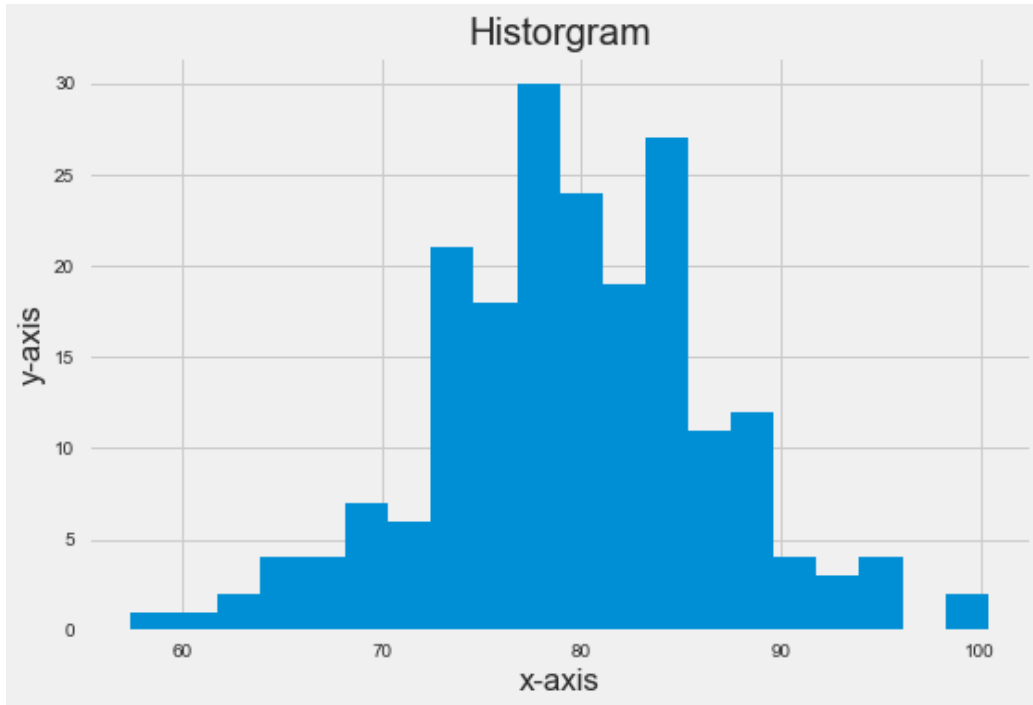
```
ax.hist(x, 20)#, density=True, histtype='bar', facecolor='g', alpha=0.8)
```

```

ax.set_title('Histogram')
ax.set_xlabel('x-axis')
ax.set_ylabel('y-axis')

fig.tight_layout()
plt.show()

```



7.9 Box Plots and Violin Plots

A couple other useful types of statistical plots are box plots and violin plots

First import matplotlib. If using a jupyter notebook include the line `%matplotlib inline`

```

In [1]: import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

```

```

In [6]: fig, ax = plt.subplots()

```

```

# generate some random data
data = [np.random.normal(0, std, 100) for std in range(6, 10)]

```

```

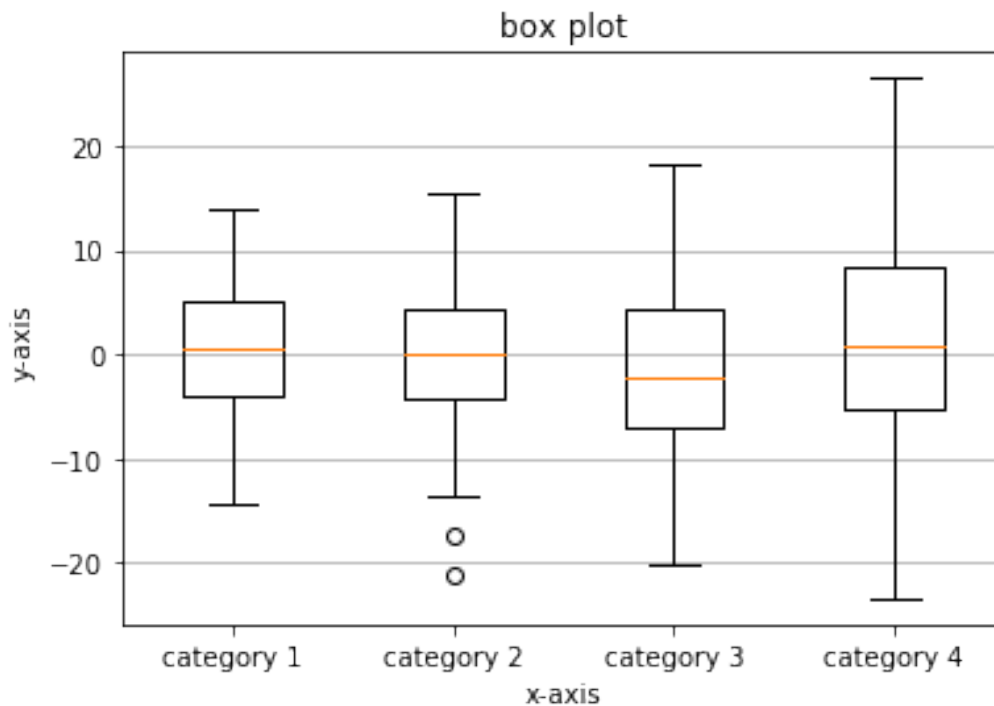
# plot box plot
ax.boxplot(data)
ax.set_title('box plot')

# adding horizontal grid lines

ax.yaxis.grid(True)
ax.set_xticks([y+1 for y in range(len(data))])
ax.set_xlabel('x-axis')
ax.set_ylabel('y-axis')

# add x-tick labels
plt.setp(ax, xticks=[y+1 for y in range(len(data))],
          xticklabels=['category 1', 'category 2', 'category 3', 'category 4'])
plt.show()

```



```
In [8]: fig, ax = plt.subplots()
```

```

# generate some random test data
all_data = [np.random.normal(0, std, 100) for std in range(6, 10)]

# plot violin plot

```

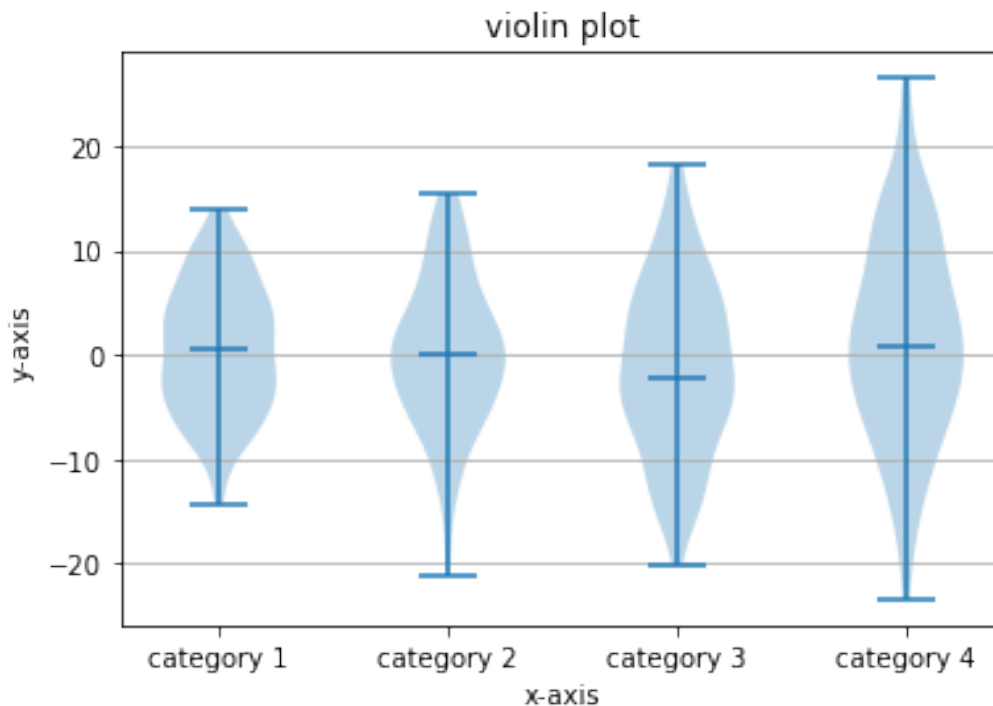
```

ax.violinplot(data, showmeans=False, showmedians=True)
ax.set_title('violin plot')

# adding horizontal grid lines
ax.yaxis.grid(True)
ax.set_xticks([y+1 for y in range(len(data))])
ax.set_xlabel('x-axis')
ax.set_ylabel('y-axis')

# add x-tick labels
plt.setp(ax, xticks=[y+1 for y in range(len(data))],
          xticklabels=['category 1', 'category 2', 'category 3', 'category 4'])
plt.show()

```



7.10 Scatter Plots

First import **matplotlib**. If using a jupyter notebook include the line `%matplotlib inline`

```

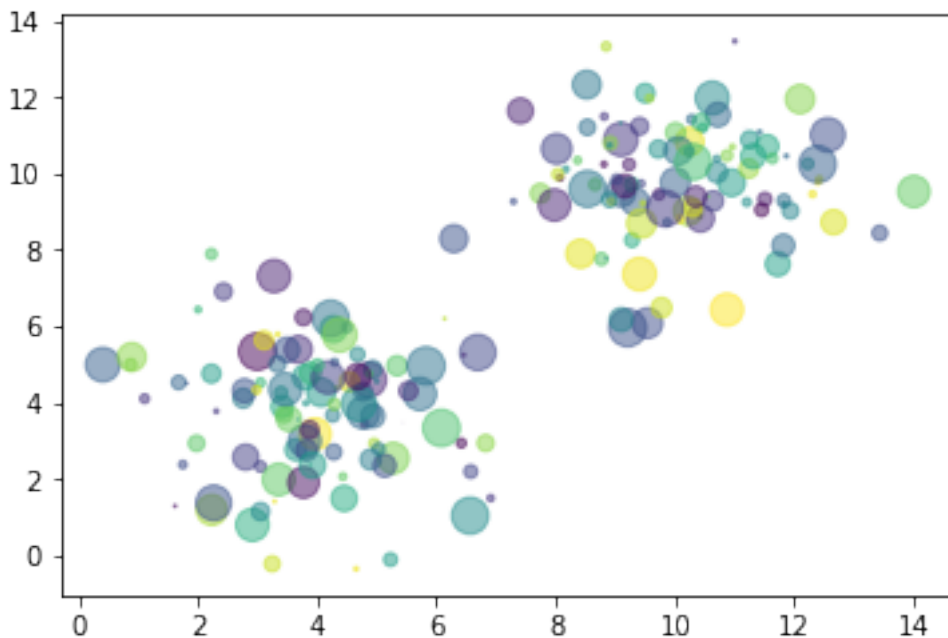
In [2]: import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

```

```
In [3]: N = 100
x1 = 1.5 * np.random.randn(N) + 10
y1 = 1.5 * np.random.randn(N) + 10
x2 = 1.5 * np.random.randn(N) + 4
y2 = 1.5 * np.random.randn(N) + 4
x = np.append(x1,x2)
y = np.append(y1,y2)
colors = np.random.rand(N*2)
area = np.pi * (8 * np.random.rand(N*2))**2 # 0 to 15 point radii

fig, ax = plt.subplots()

ax.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```



7.11 Plot annotations

Sometimes it is useful for engineers to annotate plots. Text can be included on a plot to indicate a point of interest.

First import **matplotlib**. If using a jupyter notebook include the line `%matplotlib inline`

```
In [1]: import matplotlib.pyplot as plt
```

```
import numpy as np
%matplotlib inline
```

```
In [2]: fig, ax = plt.subplots()
```

```
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
```

```
# Plot a line and add some simple annotations
```

```
line, = ax.plot(t, s)
ax.annotate('figure pixels',
            xy=(10, 10), xycoords='figure pixels')
ax.annotate('figure points',
            xy=(80, 80), xycoords='figure points')
ax.annotate('figure fraction',
            xy=(.025, .975), xycoords='figure fraction',
            horizontalalignment='left', verticalalignment='top',
            fontsize=20)
```

```
# The following examples show off how these arrows are drawn.
```

```
ax.annotate('point offset from data',
            xy=(2, 1), xycoords='data',
            xytext=(-15, 25), textcoords='offset points',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='bottom')
```

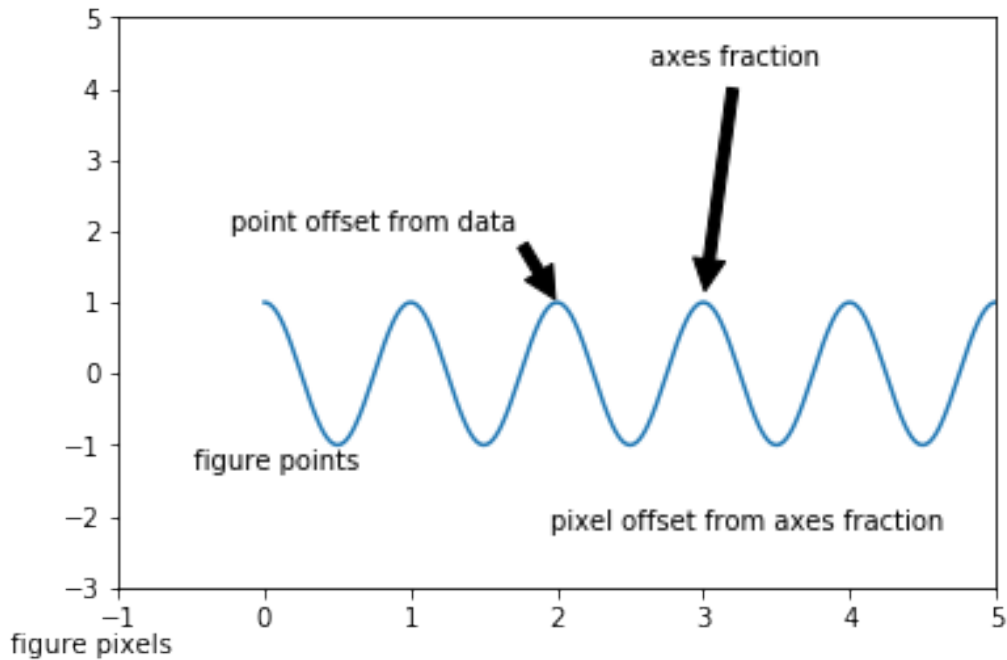
```
ax.annotate('axes fraction',
            xy=(3, 1), xycoords='data',
            xytext=(0.8, 0.95), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top')
```

```
# You may also use negative points or pixels to specify from (right, top).
# E.g., (-10, 10) is 10 points to the left of the right side of the axes and 10
# points above the bottom
```

```
ax.annotate('pixel offset from axes fraction',
            xy=(1, 0), xycoords='axes fraction',
            xytext=(-20, 20), textcoords='offset pixels',
            horizontalalignment='right',
            verticalalignment='bottom')
```

```
ax.set(xlim=(-1, 5), ylim=(-3, 5))
plt.show()
```


figure fraction



7.12 Subplots

Sometimes it is useful for engineers to include a couple plots side by side. This can be done in **matplotlib** using **subplots**

First import matplotlib. If using a jupyter notebook include the line `%matplotlib inline`

```
In [2]: import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

In [3]: # Data for plotting
t = np.arange(0.01, 20.0, 0.01)

# Create figure
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)

# linear x and y axis
ax1.plot(t, np.exp(-t / 5.0))
ax1.set(title='linear x and y')
```

```

ax1.grid()

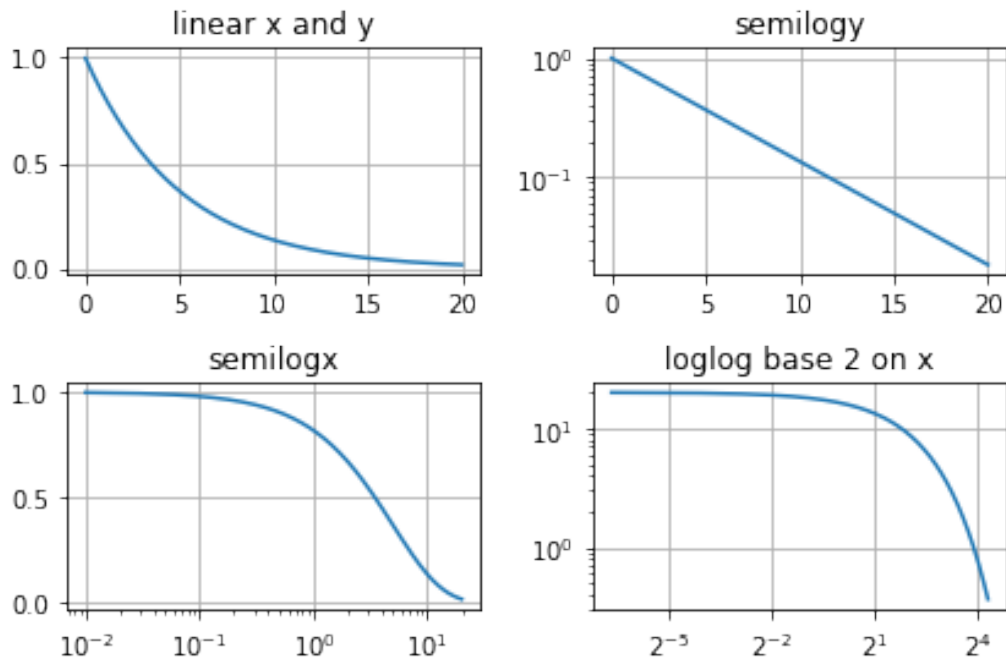
# log y axis
ax2.semilogy(t, np.exp(-t / 5.0))
ax2.set(title='semilogy')
ax2.grid()

# log x axis
ax3.semilogx(t, np.exp(-t / 5.0))
ax3.set(title='semilogx')
ax3.grid()

# log x and y axis
ax4.loglog(t, 20 * np.exp(-t / 5.0), basex=2)
ax4.set(title='loglog base 2 on x')
ax4.grid()

fig.tight_layout()
plt.show()

```



7.13 Plot Styles

Sometimes it is useful for engineers to include a couple plots side by side. This can be done in **matplotlib** using **subplots**

First import matplotlib. If using a jupyter notebook include the line `%matplotlib inline`

```
In [1]: import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

```
In [2]: plt.style.use('default')

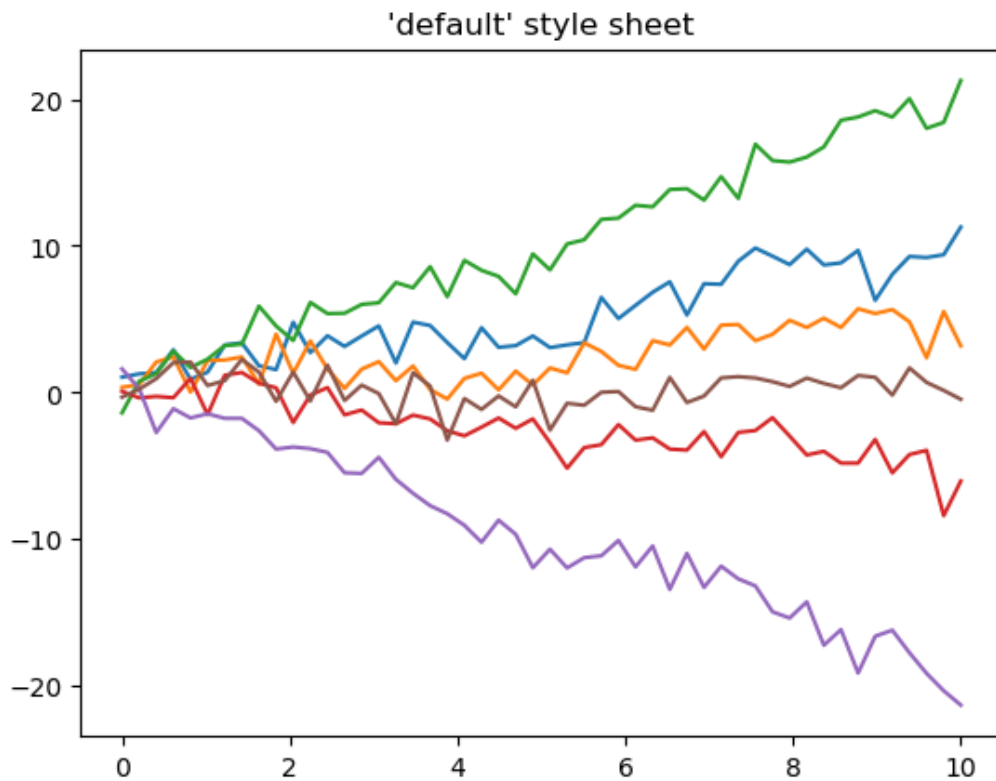
x = np.linspace(0, 10)

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, ax = plt.subplots()

ax.plot(x, np.sin(x) + x + np.random.randn(50))
ax.plot(x, np.sin(x) + 0.5 * x + np.random.randn(50))
ax.plot(x, np.sin(x) + 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 0.5 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) + np.random.randn(50))
ax.set_title("'default' style sheet")

plt.show()
```



```
In [3]: print(plt.style.available)
```

```
['bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn-rc', 'seaborn-white', 'seaborn-whitegrid', 'seaborn-darkgrid', 'seaborn-dark', 'seaborn-light', 'seaborn-bright', 'seaborn-deep', 'seaborn-pastel', 'seaborn-notebook', 'seaborn-paper', 'seaborn-beans', 'seaborn-ticks', 'seaborn-whitegrid', 'seaborn-white', 'seaborn-darkgrid', 'seaborn-dark', 'seaborn-light', 'seaborn-bright', 'seaborn-deep', 'seaborn-pastel', 'seaborn-notebook', 'seaborn-paper', 'seaborn-beans', 'seaborn-ticks']
```

```
In [8]: import matplotlib.pyplot as plt
import numpy as np
```

```
plt.style.use('fivethirtyeight')
```

```
x = np.linspace(0, 10)
```

```
# Fixing random state for reproducibility
np.random.seed(19680801)
```

```
fig, ax = plt.subplots()
```

```
ax.plot(x, np.sin(x) + x + np.random.randn(50))
```

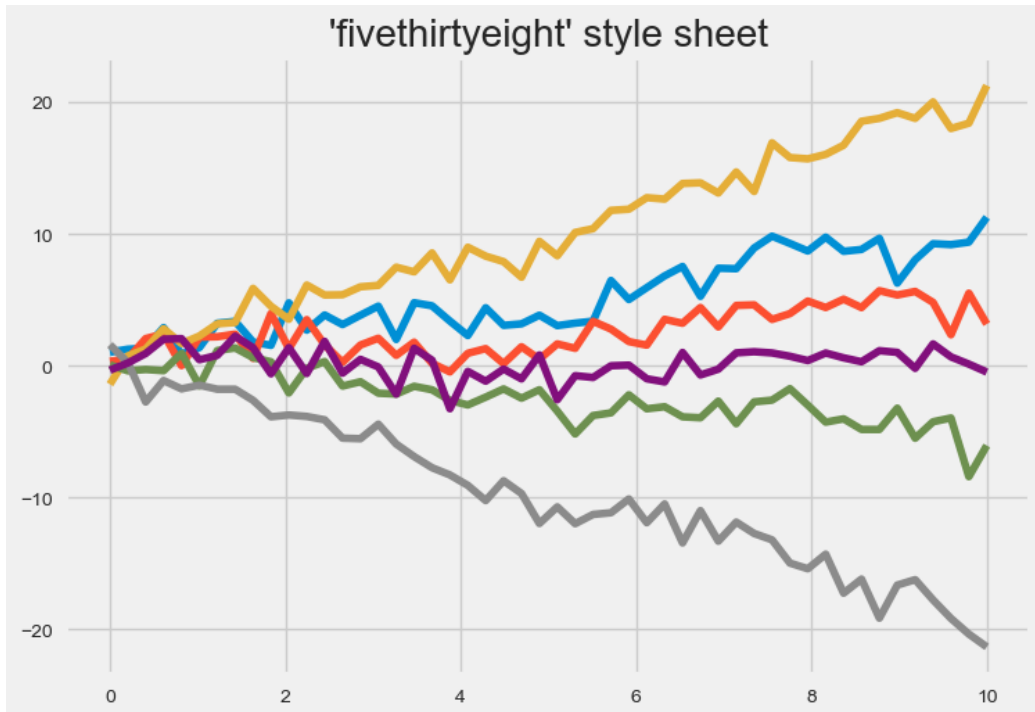
```
ax.plot(x, np.sin(x) + 0.5 * x + np.random.randn(50))
```

```

ax.plot(x, np.sin(x) + 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 0.5 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) + np.random.randn(50))
ax.set_title("'fivethirtyeight' style sheet")

plt.show()

```



```

In [9]: plt.style.use('ggplot')

x = np.linspace(0, 10)

# Fixing random state for reproducibility
np.random.seed(19680801)

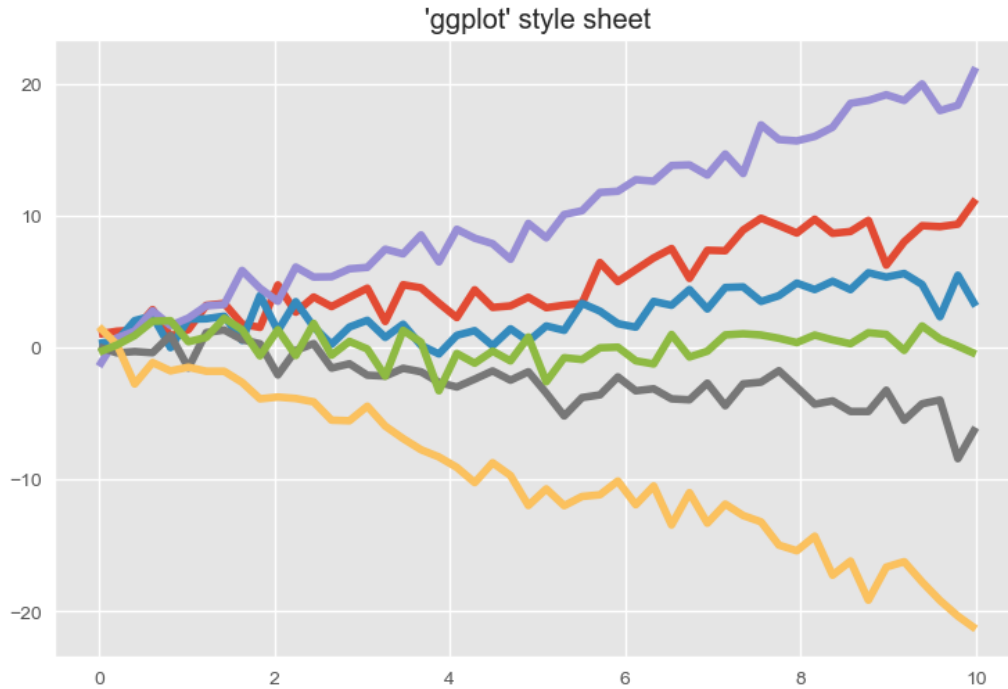
fig, ax = plt.subplots()

ax.plot(x, np.sin(x) + x + np.random.randn(50))
ax.plot(x, np.sin(x) + 0.5 * x + np.random.randn(50))
ax.plot(x, np.sin(x) + 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 0.5 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) + np.random.randn(50))

```

```
ax.set_title("'ggplot' style sheet")

plt.show()
```



```
In [6]: plt.style.use('seaborn')

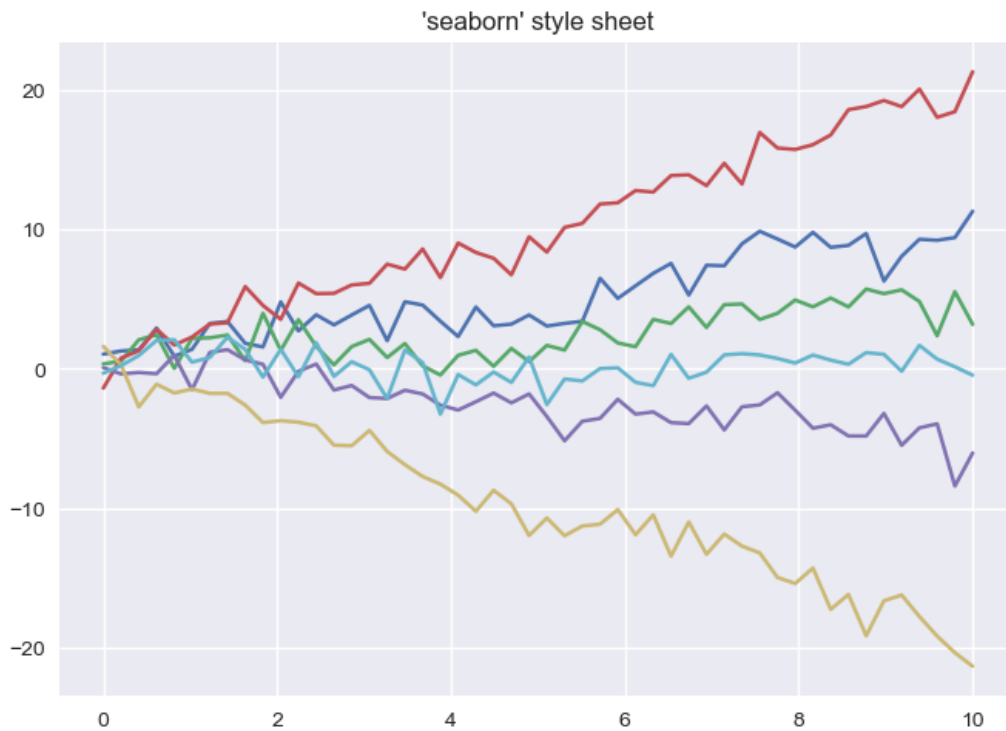
x = np.linspace(0, 10)

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, ax = plt.subplots()

ax.plot(x, np.sin(x) + x + np.random.randn(50))
ax.plot(x, np.sin(x) + 0.5 * x + np.random.randn(50))
ax.plot(x, np.sin(x) + 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 0.5 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) + np.random.randn(50))
ax.set_title("'seaborn' style sheet")

plt.show()
```



```
In [7]: plt.style.use('seaborn-colorblind')

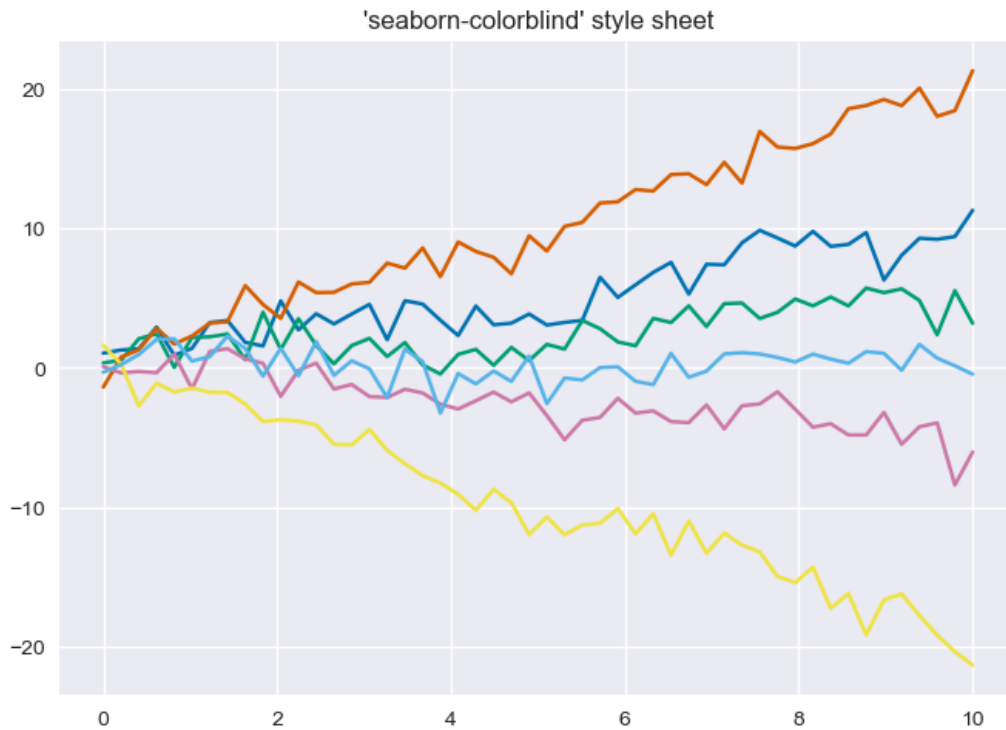
x = np.linspace(0, 10)

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, ax = plt.subplots()

ax.plot(x, np.sin(x) + x + np.random.randn(50))
ax.plot(x, np.sin(x) + 0.5 * x + np.random.randn(50))
ax.plot(x, np.sin(x) + 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 0.5 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) + np.random.randn(50))
ax.set_title("'seaborn-colorblind' style sheet")

plt.show()
```



7.14 Contour Plots

In civil engineering a contour plot could show the topology of a build sight. In mechanical engineering a contour plot could show the stress gradient across part surface.

First import matplotlib. If using a jupyter notebook include the line `%matplotlib inline`

```
In [4]: import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
```

matplotlib's `plt.contour()` method

Building contour plots using **matplotlib** entails using the `plt.contourf()` method. The basic method call is below:

```
ax.contour(X, Y, Z)
```


Where X and Y are 2D arrays of the x and y points, and Z is a 2D array of points that determines the “height” of the contour, which is represented by color in a 2D plot. The `np.meshgrid` function is useful to create two 2D arrays from two 1D arrays.

```
In [5]: x = np.arange(-3.0, 3.0, 0.1)
        y = np.arange(-3.0, 3.0, 0.1)

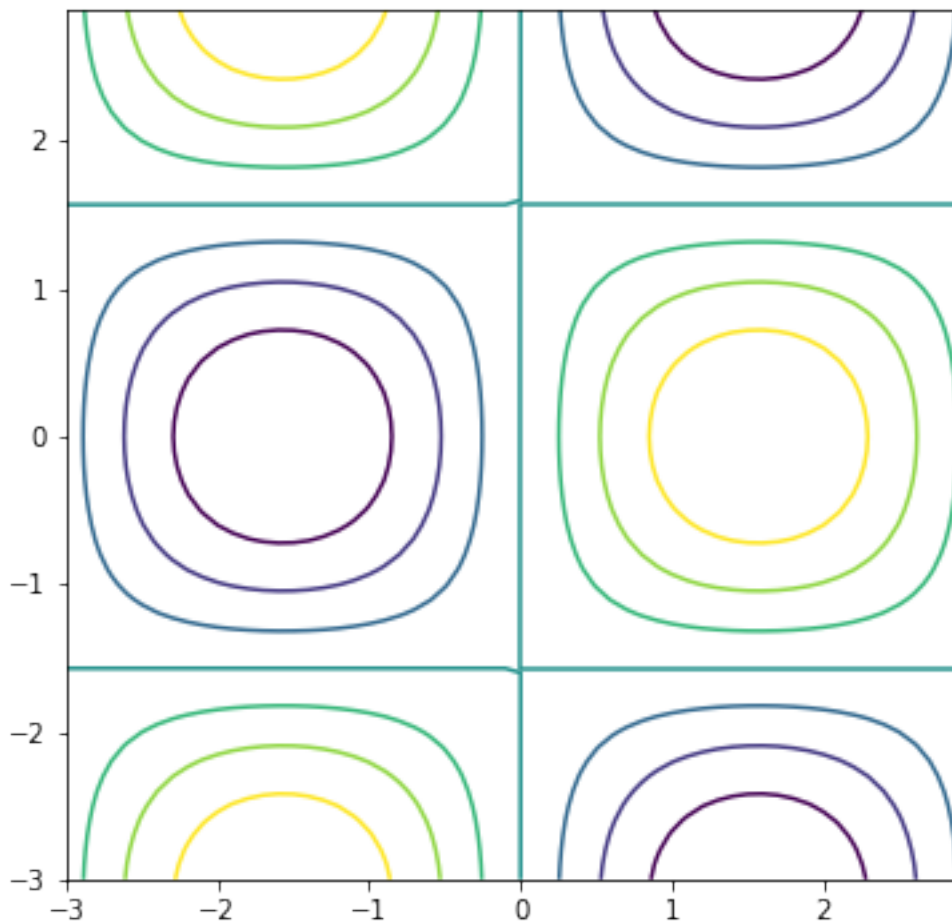
        X, Y = np.meshgrid(x, y)

        Z = np.sin(X)*np.cos(Y)

        fig, ax = plt.subplots(figsize=(6,6))

        ax.contour(X,Y,Z)

        plt.show()
```



matplotlib's `plt.contourf()` method

Matplotlib's `plt.contourf()` method is similar to `plt.contour()` except that it will produce contour plots that are “filled”. Instead of lines in a `plt.contour()` plot, shaded areas are produced by a `plt.contourf()` plot. The general method call for `plt.contourf()` is similar to `plt.contour()`.

```
ax.contourf(X, Y, Z)
```

Where `X` and `Y` are 2D arrays of the `x` and `y` points, and `Z` is a 2D array of points that determines the color of the areas on the 2D plot.

```
In [6]: import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline

x = np.arange(-3.0, 3.0, 0.1)
y = np.arange(-3.0, 3.0, 0.1)

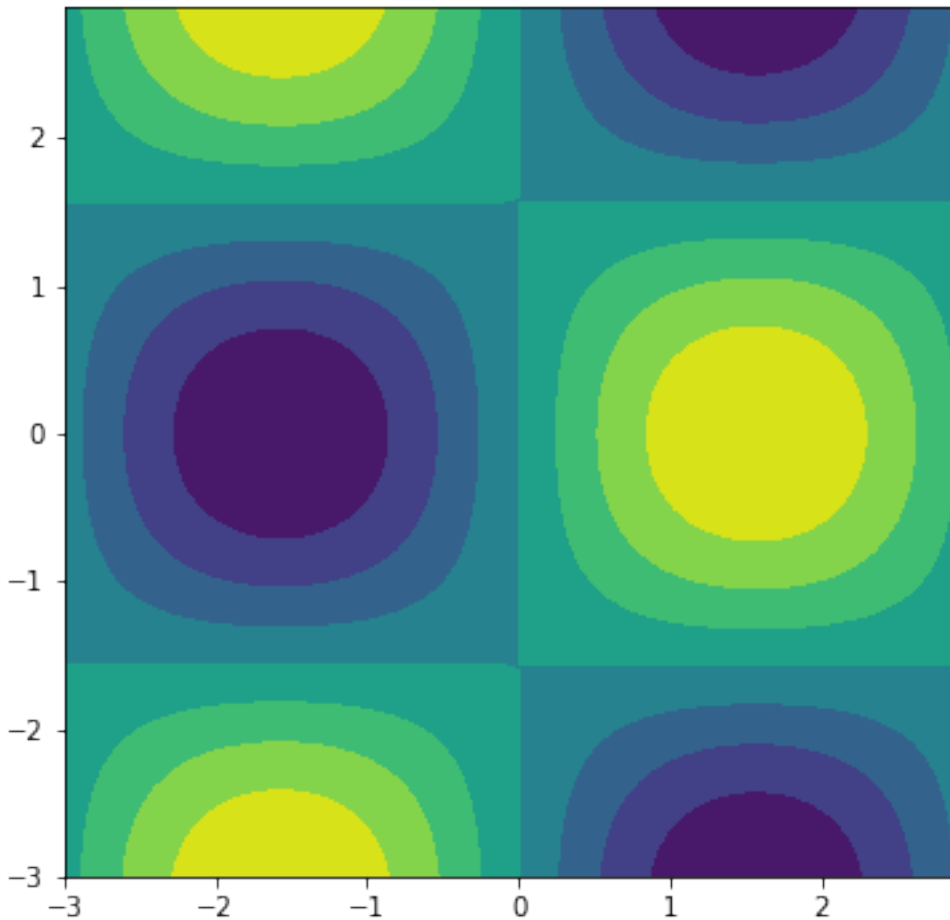
X, Y = np.meshgrid(x, y)

Z = np.sin(X)*np.cos(Y)

fig, ax = plt.subplots(figsize=(6,6))

ax.contourf(X,Y,Z)

plt.show()
```



Color bars on contour plots

Because colors represent a third dimension (like “height”) on a 2D plot, it is useful to have a scale to what each color means. A color scale is typically represented besides a plot. Color bars are added to **matplotlib** contour plots using the `fig.colorbar()` method. Since the color bar is not part of the contour plot, the color bar needs to be applied to the figure object, often called `fig`. A contour plot needs to be passed into the `fig.colorbar()` method, so when applying a color bar to a figure, a plot object needs to be available. A plot object is the output of the `ax.contour()` method. Previously the output of this method was not assigned to a variable. But to include a color bar on a contour plot, the plot object needs to be saved to a variable so that it can be passed to the `fig.colorbar()` method.

```
cf = ax1.contourf(X,Y,Z)
fig.colorbar(cf, ax=ax1)
```

Where `cf` is the plot object created by `ax1.contourf(X, Y, Z)`. The axis object that contains the contour plot, `ax1` is passed to the `fig.colorbar()` method along with the `cf` plot object.

```
In [10]: import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline

x = np.arange(-3.0, 3.0, 0.1)
y = np.arange(-3.0, 3.0, 0.1)

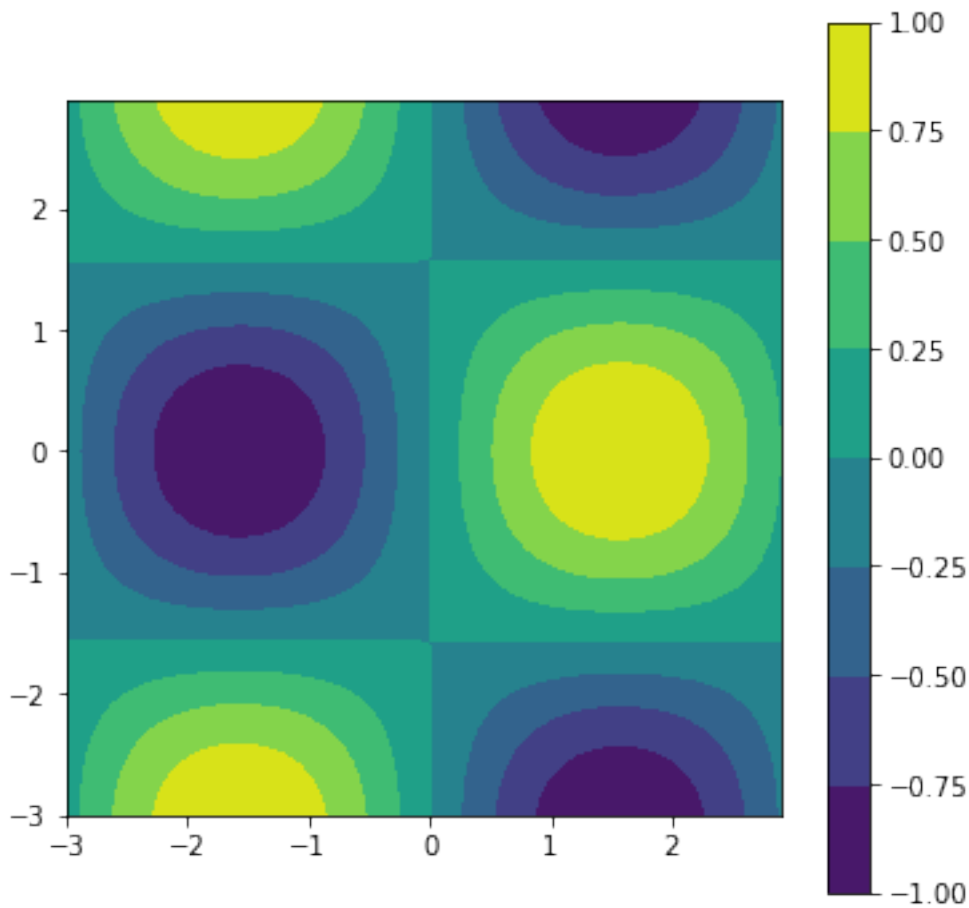
X, Y = np.meshgrid(x, y)

Z = np.sin(X)*np.cos(Y)

fig, ax1 = plt.subplots(figsize=(6,6))

ax1.set_aspect('equal')
cf = ax1.contourf(X,Y,Z)
fig.colorbar(cf, ax=ax1)

plt.show()
```



Color maps on contour plots

The default color scheme of **matplotlib** contour and filled contour plots can be modified. A general way to do this is to call **matplotlib's** `plt.get_cmap()` function which will output a color map object. There are many different colormaps available to apply to contour plots. A complete list is available in the **matplotlib** documentation. The colormap object is then passed to the `ax.contourf()` or `ax.contour` method as a keyword argument.

```
mycmap = plt.get_cmap('gist_earth')
ax.contourf(X, Y, Z, cmap=mycmap)
```

```
In [18]: import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
```

```

x = np.arange(-3.0, 3.0, 0.1)
y = np.arange(-3.0, 3.0, 0.1)

X, Y = np.meshgrid(x, y)

Z = np.sin(X)*np.cos(Y)

fig, [ax1,ax2] = plt.subplots(1,2,figsize=(12,6))

mycmap1 = plt.get_cmap('gist_earth')
ax1.set_aspect('equal')
ax1.set_title('Colormap: gist_earth')
cf1 = ax1.contourf(X,Y,Z, cmap=mycmap1)

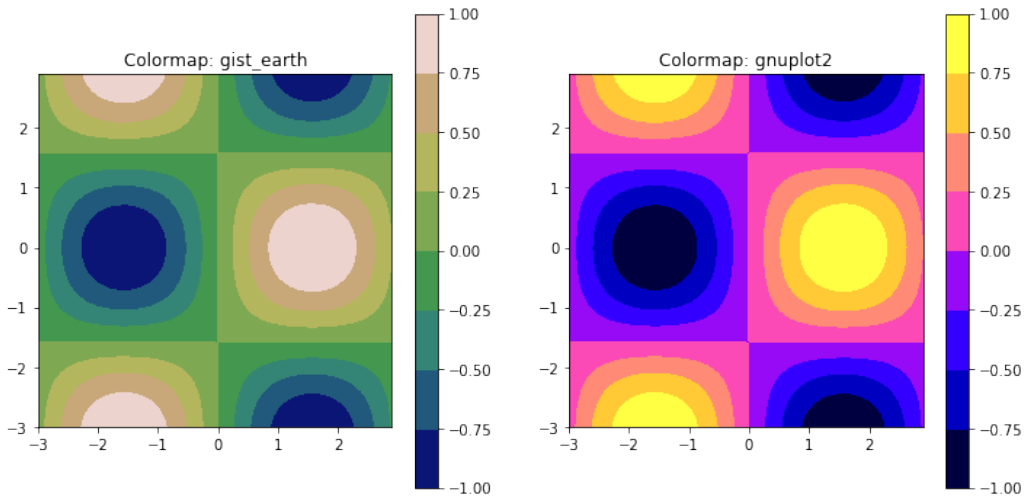
fig.colorbar(cf1, ax=ax1)

mycmap2 = plt.get_cmap('gnuplot2')
ax2.set_aspect('equal')
ax2.set_title('Colormap: gnuplot2')
cf2 = ax2.contourf(X,Y,Z, cmap=mycmap2)

fig.colorbar(cf2, ax=ax2)

plt.show()

```



7.15 Quiver and Stream Plots

In this section, you will learn how to build quiver and stream plots using **matplotlib**

Quiver Plots

A quiver plot is a type of 2D plot that shows vector lines as arrows. Quiver plots are useful in electrical engineering to visualize electrical potential and useful in mechanical engineering to show stress gradients.

To build a quiver plot, first import **matplotlib**. Again, the alias `plt` will be used instead of `matplotlib.pyplot`. If using a jupyter notebook include the line `%matplotlib inline`. For some of the quiver plots in this section, **numpy** will be needed as well.

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np

        %matplotlib inline
```

Quiver plot with one arrow

Let's build a simple quiver plot that contains one arrow to see how `matplotlib's ax.quiver()` method works. The `ax.quiver()` method takes four positional arguments:

```
ax.quiver(x_pos, y_pos, x_direct, y_direct)
```

Where `x_pos` and `y_pos` are the arrow starting positions and `x_direct`, `y_direct` are the arrow directions.

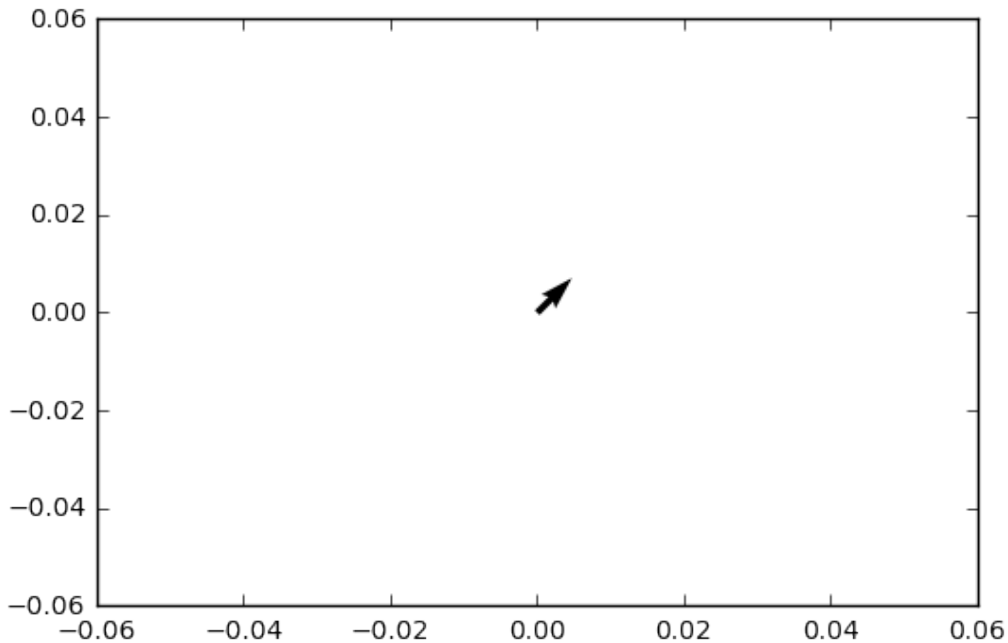
Let's build our first plot to contain one quiver arrow at the starting point `x_pos = 0`, `y_pos = 0`. We'll define this quiver arrow's direction as pointing up and to the right `x_direct = 1`, `y_direct = 1`.

```
In [2]: fig, ax = plt.subplots()

        x_pos = 0
        y_pos = 0
        x_direct = 1
        y_direct = 1

        ax.quiver(x_pos,y_pos,x_direct,y_direct)

        plt.show()
```



We can see one arrow pointing up and to the right.

Quiver plot with two arrows

Now let's add a second arrow to the quiver plot by passing in two starting points and two arrow directions.

We'll keep our original arrow- starting position at the origin 0,0 and pointing up and to the right, direction 1,1. We'll define a second arrow with a starting position of -0.5,0.5 which points straight down (in the 0,-1 direction).

An additional keyword argument to add to the `ax.quiver()` method is `scale=5`. This will scale the arrow lengths so the arrows show up better on the quiver plot.

To see the start and end of both arrows, we'll set the axis limits between -1.5 and 1.5 using the `ax.axis()` method and passing in a list of axis limits in the form `[xmin, xmax, ymin, ymax]`.

We can see two arrows. One arrow points to the upper right and the other arrow points straight down.

Quiver plot using a meshgrid

Two arrows is great, but to create a whole 2D surface worth of arrows, we'll utilize **numpy's** `meshgrid()` function.

We need to build a set of arrays that denote the x and y starting positions of each quiver arrow on the plot. We will call our quiver arrow starting position arrays `X` and `Y`.

We can use the x, y arrow starting *positions* to define the x and y components of each quiver arrow *direction*. We will call the quiver arrow direction arrays u and v . On this quiver plot, we will define the quiver arrow direction based upon the quiver arrow starting point using:

$$x_{\text{direction}} = \cos(x_{\text{starting point}})$$

$$y_{\text{direction}} = \sin(y_{\text{starting point}})$$

```
In [3]: x = np.arange(0,2.2,0.2)
        y = np.arange(0,2.2,0.2)

        X, Y = np.meshgrid(x, y)
        u = np.cos(X)*Y
        v = np.sin(y)*Y
```

Now we can build the quiver plot using **matplotlib's** `ax.quiver()` method. Again, the method call takes four positional arguments:

```
ax.quiver(x_pos, y_pos, x_direct, y_direct)
```

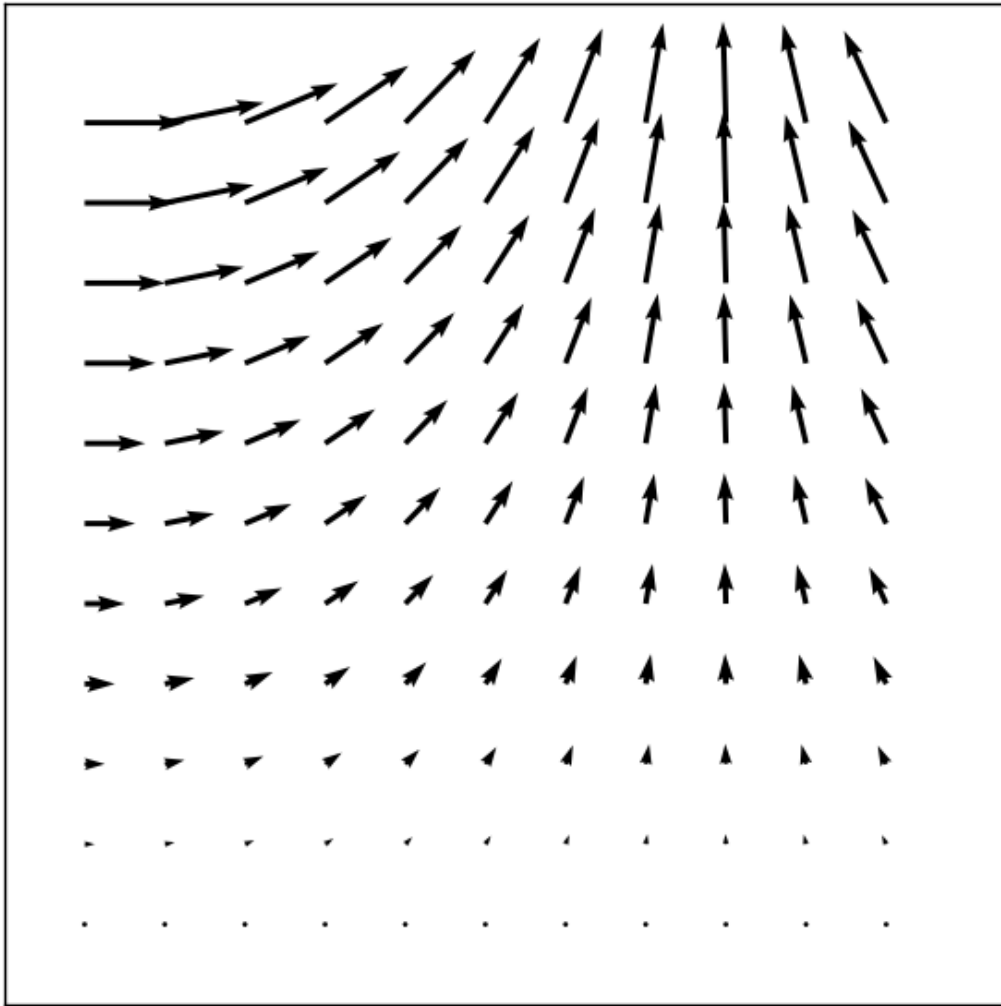
This time `x_pos` and `y_pos` are 2D arrays which contain the starting positions of the arrows and `x_direct, y_direct` are 2D arrays which contain the arrow directions.

The commands `ax.xaxis.set_ticks([])` and `ax.yaxis.set_ticks([])` removes the tick marks from the axis and `ax.set_aspect('equal')` sets the aspect ratio of the plot to 1:1.

```
In [4]: fig, ax = plt.subplots(figsize=(7,7))
        ax.quiver(X,Y,u,v)

        ax.xaxis.set_ticks([])
        ax.yaxis.set_ticks([])
        ax.axis([-0.2, 2.3, -0.2, 2.3])
        ax.set_aspect('equal')

        plt.show()
```



Now let's build another quiver plot with the \hat{i} and \hat{j} components of the arrows, \vec{F} are dependant upon the arrow starting point x, y according to the function:

$$\vec{F} = \frac{x}{5} \hat{i} - \frac{y}{5} \hat{j}$$

Again we can use **numpy's** `np.meshgrid()` function to build the arrow starting position arrays, then apply our function \vec{F} to the x and y arrow starting point arrays.

```
In [11]: x = np.arange(-1,1,0.1)
         y = np.arange(-1,1,0.1)

         X, Y = np.meshgrid(x, y)
         u = np.cos(X)*Y
```

```
v = np.sin(y)*Y

X,Y = np.meshgrid(x,y)

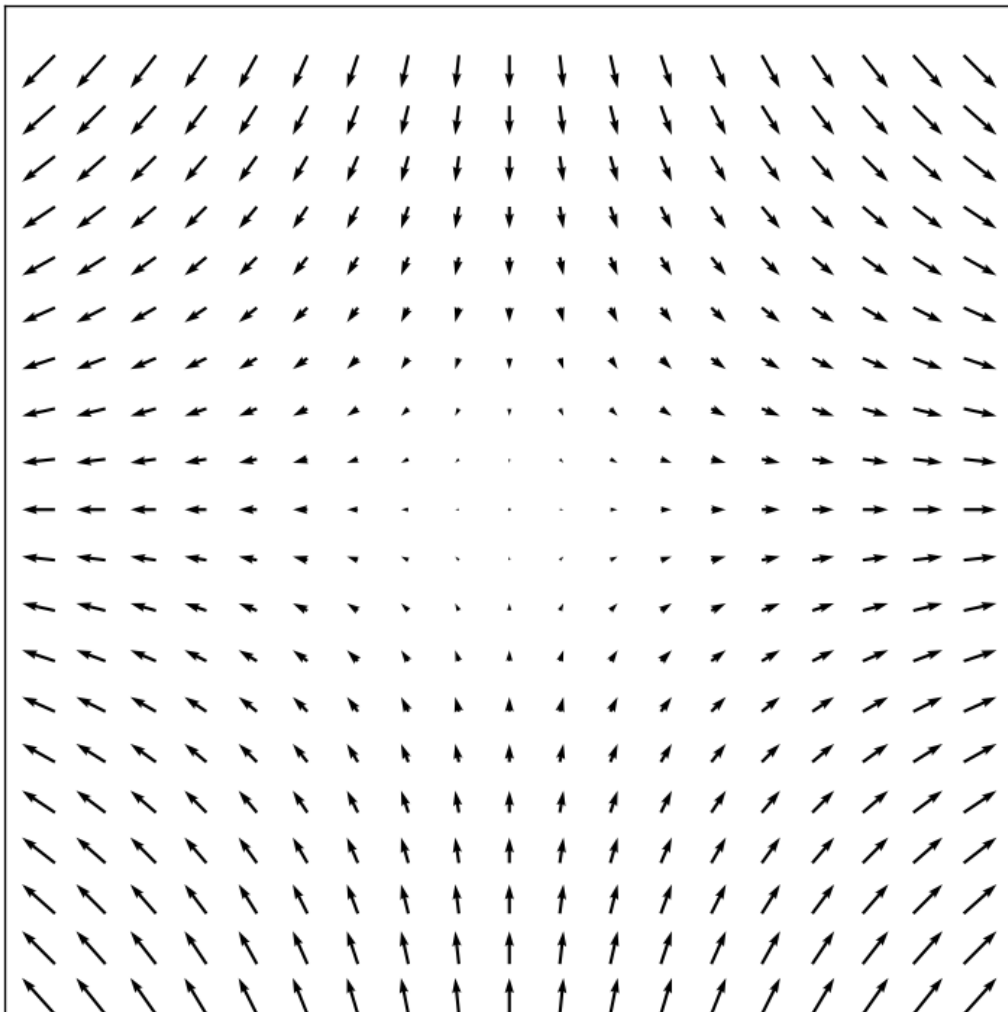
u = X/5
v = -Y/5

fig, ax = plt.subplots(figsize=(9,9))

ax.quiver(X,Y,u,v)

ax.xaxis.set_ticks([])
ax.yaxis.set_ticks([])
ax.set_aspect('equal')

plt.show()
```



Quiver plot containing a gradient

Next let's build another quiver plot using the gradient function. The gradient function will have the form:

$$z = xe^{-x^2-y^2}$$

We can use **numpy's** **np.gradient()** function to apply the gradient function to every arrow's x,y starting position.

```
In [6]: x = np.arange(-2,2.2,0.2)
        y = np.arange(-2,2.2,0.2)

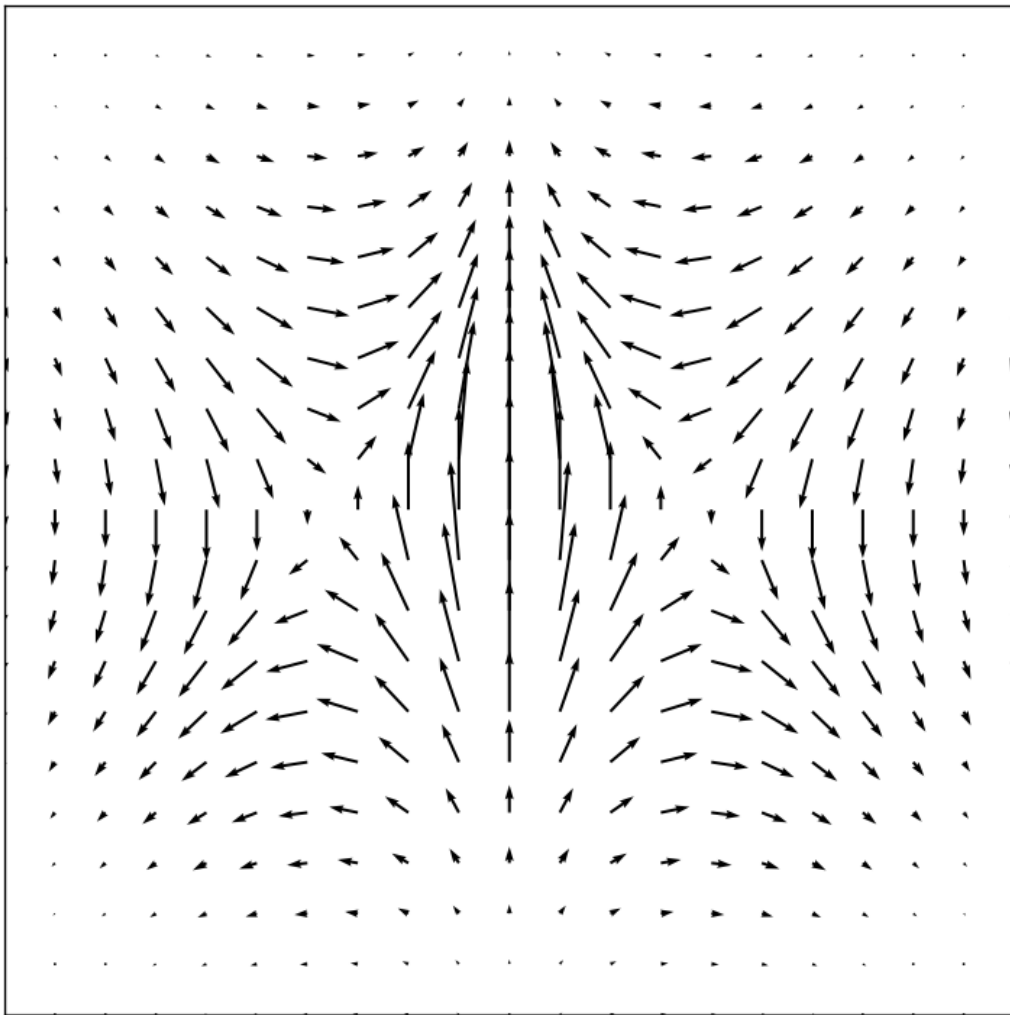
        X, Y = np.meshgrid(x, y)
        z = X*np.exp(-X**2 -Y**2)
        dx, dy = np.gradient(z)

        fig, ax = plt.subplots(figsize=(9,9))

        ax.quiver(X,Y,dx,dy)

        ax.xaxis.set_ticks([])
        ax.yaxis.set_ticks([])
        ax.set_aspect('equal')

        plt.show()
```



Quiver plot with four vortices

Now let's build a quiver plot containing four vortices. The function \vec{F} which describes the 2D field is:

$$\vec{F} = \sin(x)\cos(y) \hat{i} - \cos(x)\sin(y) \hat{j}$$

Again we can build these arrays using **numpy** and plot them with **matplotlib**.

```
In [8]: x = np.arange(0, 2*np.pi+2*np.pi/20, 2*np.pi/20)
        y = np.arange(0, 2*np.pi+2*np.pi/20, 2*np.pi/20)
```

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

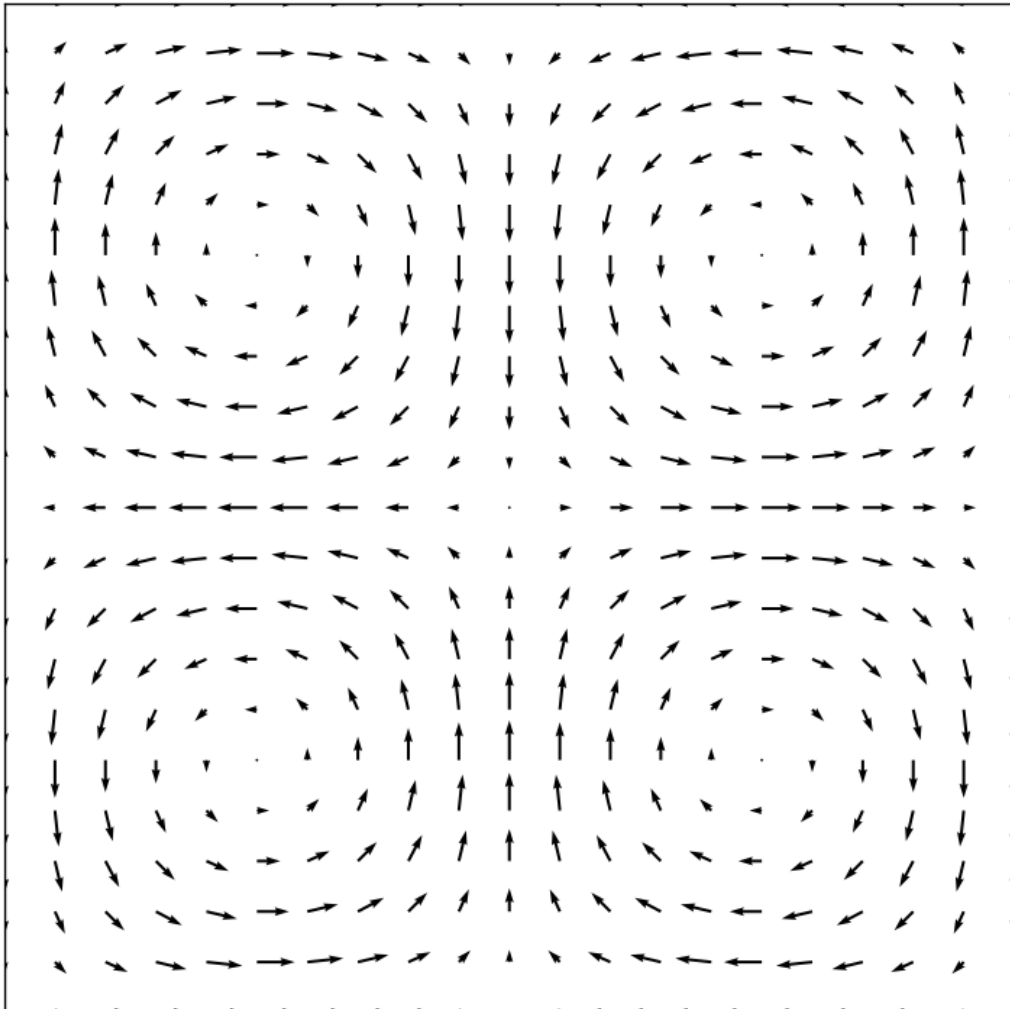
u = np.sin(X)*np.cos(Y)
v = -np.cos(X)*np.sin(Y)

fig, ax = plt.subplots(figsize=(9,9))

ax.quiver(X,Y,u,v)

ax.xaxis.set_ticks([])
ax.yaxis.set_ticks([])
ax.axis([0,2*np.pi,0,2*np.pi])
ax.set_aspect('equal')

plt.show()
```



Quiver plots with color

Now let's add some color to the quiver plots. The `ax.quiver()` method has an optional fifth positional argument that specifies the quiver arrow color. The quiver arrow color argument needs to have the same dimensions as the position and direction arrays.

Using **matplotlib** subplots, we can build a figure which contains 3 quiver plots each in color

```
In [12]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fig, [ax1,ax2,ax3] = plt.subplots(1,3)

x = np.arange(0,2.2,0.2)
y = np.arange(0,2.2,0.2)

X, Y = np.meshgrid(x, y)
u = np.cos(X)*Y
v = np.sin(y)*Y

n = -2
R = np.sqrt(((v-n)/2)**2 + ((u-n)/2)**2)

ax1.quiver(X,Y,u,v,R, alpha=0.8)

ax1.xaxis.set_ticks([])
ax1.yaxis.set_ticks([])
ax1.axis([-0.2, 2.3, -0.2, 2.3])
ax1.set_aspect('equal')

x = np.arange(-2,2.2,0.2)
y = np.arange(-2,2.2,0.2)

X, Y = np.meshgrid(x, y)
z = X*np.exp(-X**2 -Y**2)
dx, dy = np.gradient(z)

n = -2
R = np.sqrt(((dx-n)/2)**2 + ((dy-n)/2)**2)

ax2.quiver(X,Y,dx,dy,R)

ax2.xaxis.set_ticks([])
ax2.yaxis.set_ticks([])
```

```

ax2.set_aspect('equal')

x = np.arange(0,2*np.pi+2*np.pi/20,2*np.pi/20)
y = np.arange(0,2*np.pi+2*np.pi/20,2*np.pi/20)

X,Y = np.meshgrid(x,y)

u = np.sin(X)*np.cos(Y)
v = -np.cos(X)*np.sin(Y)

n = -1
R = np.sqrt(((dx-n)/2)**2 + ((dy-n)/2)**2)

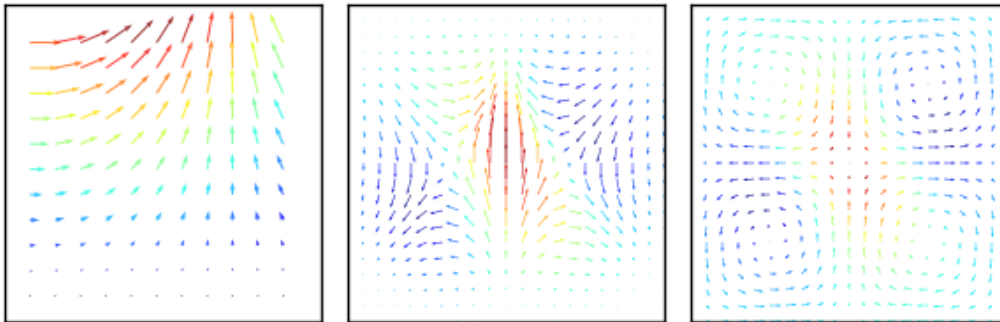
ax3.quiver(X,Y,u,v,R)

ax3.xaxis.set_ticks([])
ax3.yaxis.set_ticks([])
ax3.axis([0,2*np.pi,0,2*np.pi])
ax3.set_aspect('equal')

plt.tight_layout()
fig.savefig('3_quiver_plots.png', dpi=300, bbox_inches='tight')

plt.show()

```



Stream Plots

A stream plot is a type of plot used to show fluid flow and 2D field gradients.

The basic method to build a stream plot in **matplotlib** is:

```
ax.streamplot(x_grid,y_grid,x_vec,y_vec, density=spacing)
```


Where `x_grid` and `y_grid` are arrays of `x,y` points. The arrays `x_vec` and `y_vec` denote the stream velocity at each point on the grid. The keyword argument `density=spacing` specifies how close together to draw the stream lines.

A simple stream plot

Let's build a simple plot of stream lines on a 10 x 10 grid where all the stream lines are parallel and point to the right.

```
In [1]: import matplotlib.pyplot as plt
import numpy as np

#include if using a jupyter notebook. If using a .py script, comment out
%matplotlib inline

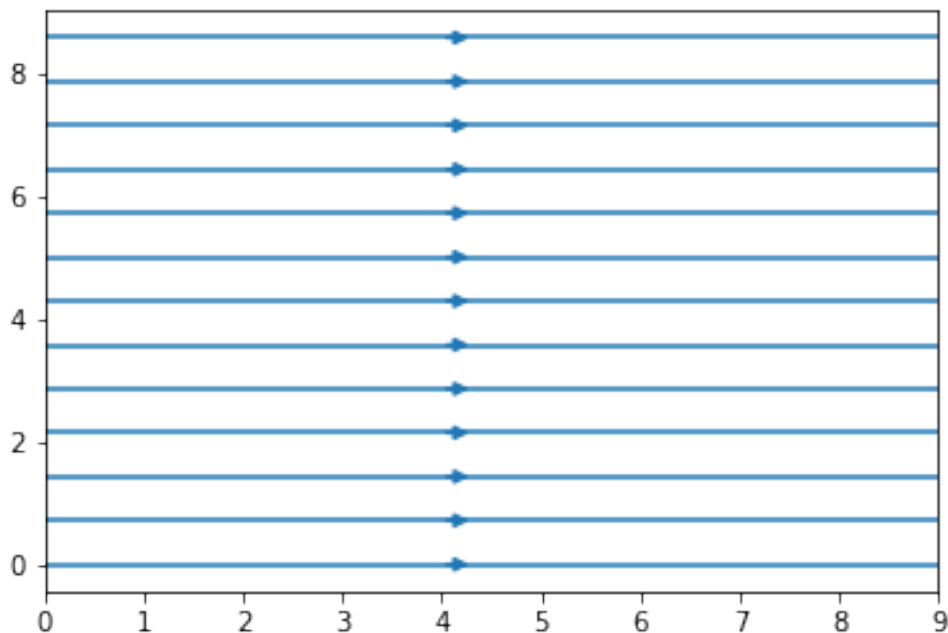
In [2]: x = np.arange(0,10)
y = np.arange(0,10)

X, Y = np.meshgrid(x,y)
u = np.ones((10,10))
v = np.zeros((10,10))

fig, ax = plt.subplots()

ax.streamplot(X,Y,u,v, density = 0.5)

plt.show()
```



The plot contains parallel streamlines all pointing to the right.

Stream plot of a field

We can build a stream plot which shows field lines based on a defined 2D vector field.

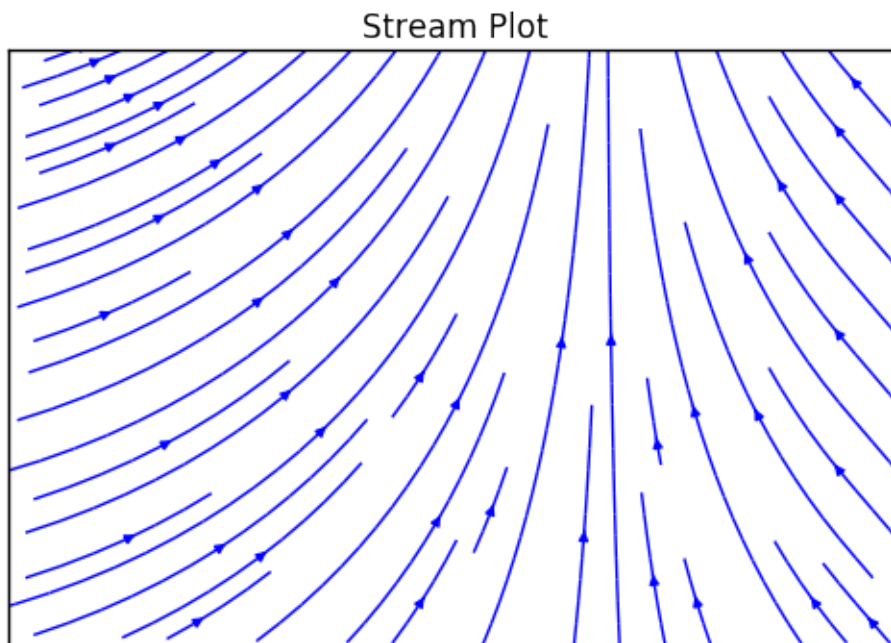
```
In [25]: x = np.arange(0,2.2,0.1)
        y = np.arange(0,2.2,0.1)

        X, Y = np.meshgrid(x, y)
        u = np.cos(X)*Y
        v = np.sin(y)*Y

        fig, ax = plt.subplots()

        ax.streamplot(X,Y,u,v, density = 1)
        ax.axis([0.5,2.1,0,2])
        ax.xaxis.set_ticks([])
        ax.yaxis.set_ticks([])
        ax.set_title('Stream Plot')

        plt.show()
```



Stream plot of two point charges

We can build a stream plot showing the electric field due to two point charges. The electric field at any point is dependant upon the position and distance relative to the point charges.

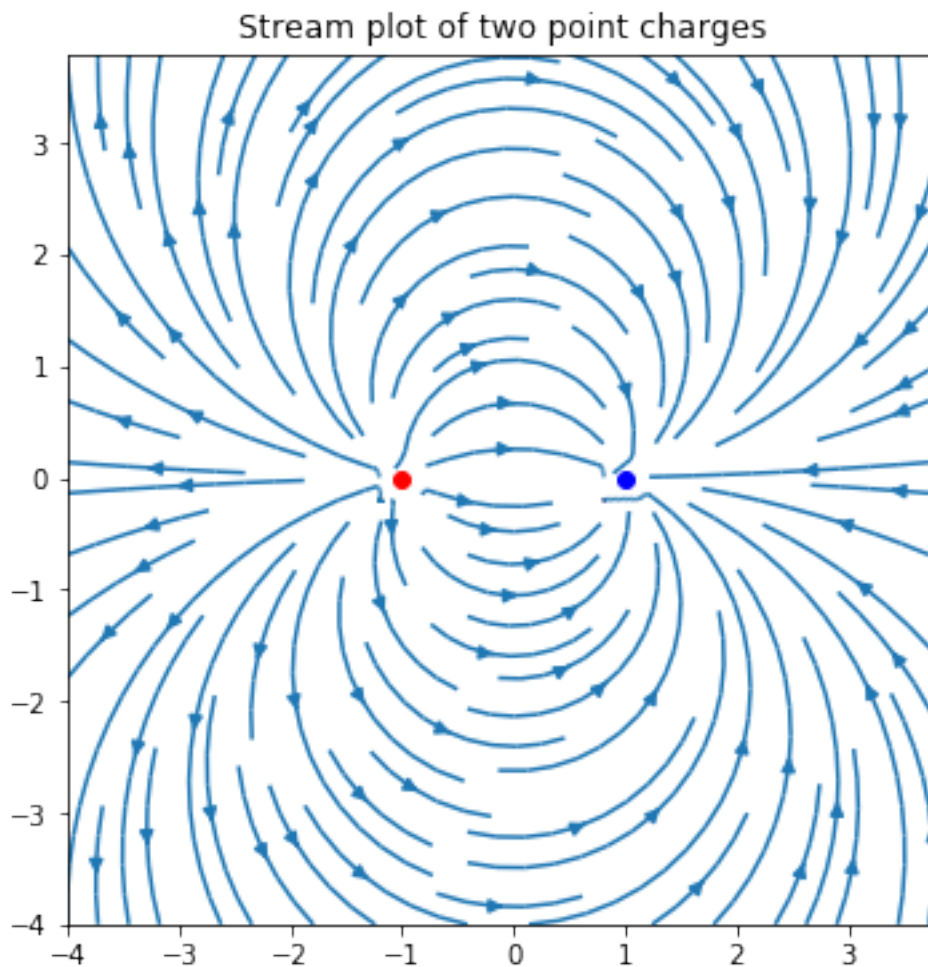
```
In [14]: import numpy as np
import matplotlib.pyplot as plt

x = np.arange(-4,4,0.2)
y = np.arange(-4,4,0.2)

X,Y = np.meshgrid(x,y)
Ex = (X + 1)/((X+1)**2 + Y**2) - (X - 1)/((X-1)**2 + Y**2)
Ey = Y/((X+1)**2 + Y**2) - Y/((X-1)**2 + Y**2)

fig, ax = plt.subplots(figsize=(6,6))
ax.streamplot(X,Y,Ex,Ey)
ax.set_aspect('equal')
ax.plot(-1,0,'-or')
ax.plot(1,0,'-ob')

ax.set_title('Stream plot of two point charges')
plt.show()
```



Stream plot showing fluid flow around an object

Stream plots can also be used to show how fluid flows around a stationary object. In this example, we will consider a 2D circle as the stationary object and model the fluid flow around the circle.

In [5]: `#from: https://tonysyu.github.io/plotting-streamlines-with-matplotlib-and-sympy.html#.Wz`

```
import matplotlib.pyplot as plt
import numpy as np
import sympy
from sympy.abc import x, y

%matplotlib inline
```

```

def cylinder_stream_function(U=1, R=1):
    r = sympy.sqrt(x**2 + y**2)
    theta = sympy.atan2(y, x)
    return U * (r - R**2 / r) * sympy.sin(theta)

def velocity_field(psi):
    u = sympy.lambdify((x, y), psi.diff(y), 'numpy')
    v = sympy.lambdify((x, y), -psi.diff(x), 'numpy')
    return u, v

import numpy as np

def plot_streamlines(ax, u, v, xlim=(-1, 1), ylim=(-1, 1)):
    x0, x1 = xlim
    y0, y1 = ylim
    Y, X = np.ogrid[y0:y1:100j, x0:x1:100j]
    ax.streamplot(X, Y, u(X, Y), v(X, Y), color='cornflowerblue')

def format_axes(ax):
    ax.set_aspect('equal')
    ax.figure.subplots_adjust(bottom=0, top=1, left=0, right=1)
    ax.xaxis.set_ticks([])
    ax.yaxis.set_ticks([])
    ax.set_aspect('equal')

import matplotlib.pyplot as plt

psi = cylinder_stream_function()
u, v = velocity_field(psi)

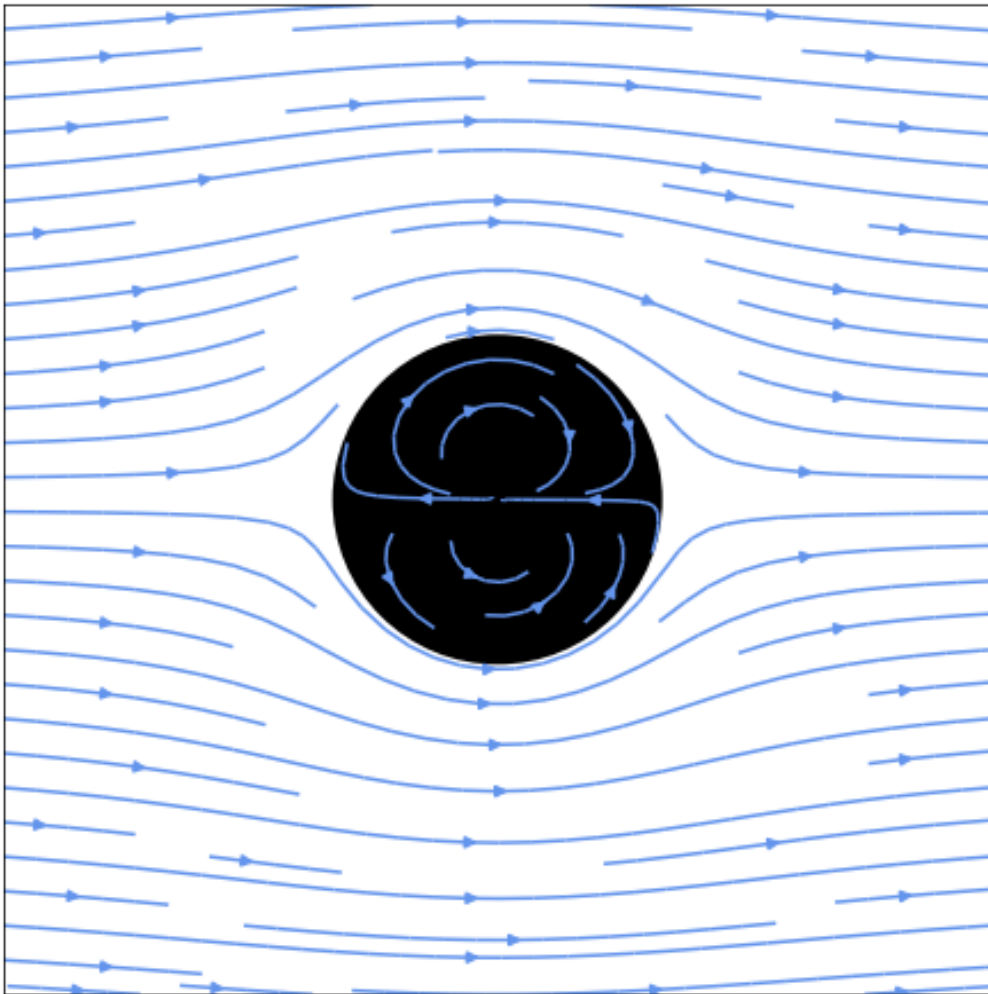
xlim = ylim = (-3, 3)
fig, ax = plt.subplots(figsize=(6, 6))
plot_streamlines(ax, u, v, xlim, ylim)
#ax.streamplot(u, v, xlim, ylim)

c = plt.Circle((0, 0), radius=1, facecolor='k')
ax.add_patch(c)

format_axes(ax)

plt.show()

```



7.16 3D Surface Plots

3D surface plots are useful for engineers.

The `mpl_toolkits.mplot3d` import `axes3d` submodule included with **matplotlib** provides the methods necessary to create 3D surface plots.

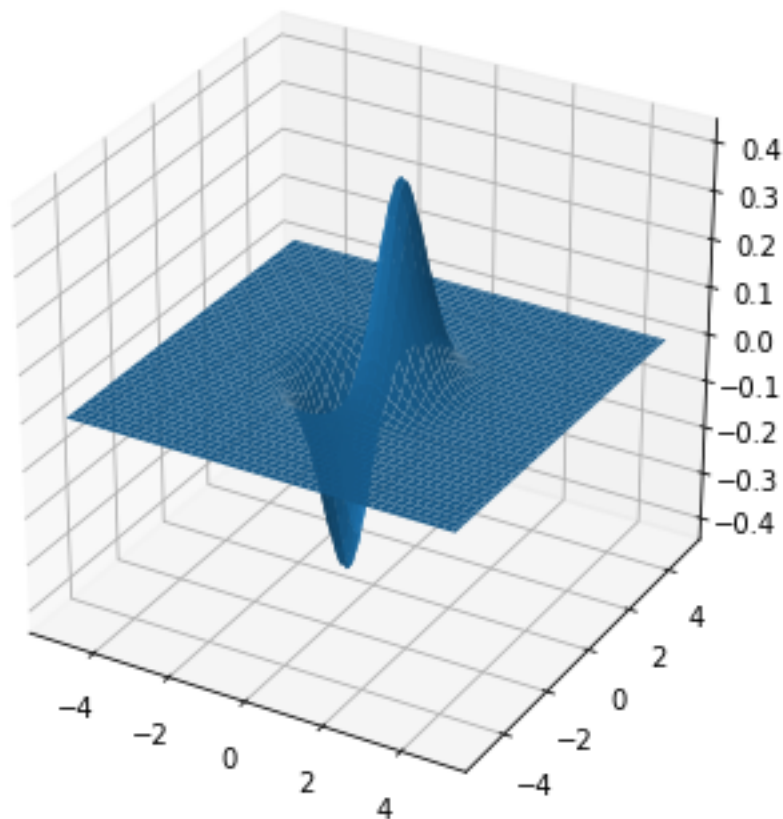
First import **matplotlib**. If using a jupyter notebook include the line `%matplotlib inline`

Surface Plots

```
In [1]: from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

In [2]: x = np.arange(-5,5,0.1)
y = np.arange(-5,5,0.1)
X,Y = np.meshgrid(x,y)
Z = X*np.exp(-X**2 - Y**2)

In [3]: fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z)
plt.show()
```



Wire Frame Plots

Wire frame 3D surface plots can be constructed using **matplotlib's** `ax.plot_wireframe()` method. The general method call is:

```
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)
```

Where `X` and `Y` are 2D array of `x` and `y` points and `Z` is a 2D array of heights. The keyword arguments `rstride=` and `cstride=` are the row step size and the column step size. These keyword arguments control how close together the “wires” in the wireplot are drawn.

```
In [4]: from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(121, projection='3d')
ax2 = fig.add_subplot(122, projection='3d')

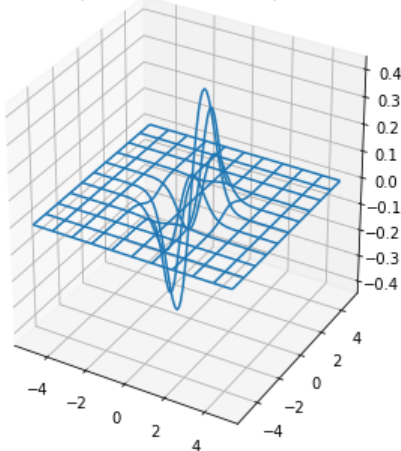
x = np.arange(-5,5,0.1)
y = np.arange(-5,5,0.1)
X,Y = np.meshgrid(x,y)
Z = X*np.exp(-X**2 - Y**2)

# Plot a basic wireframe.
ax1.plot_wireframe(X, Y, Z, rstride=10, cstride=10)
ax1.set_title('row step size 10, column step size 10')

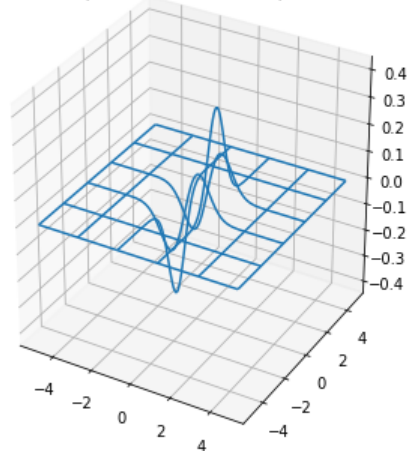
ax2.plot_wireframe(X, Y, Z, rstride=20, cstride=20)
ax2.set_title('row step size 20, column step size 20')

plt.show()
```

row step size 10, column step size 10



row step size 20, column step size 20



Gradient Surface Plots

Gradient surface plots combine a 3D surface plot with a 2D contour plot. Along with the Z height included on the surface, a color is also included. The general method call is:

```
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, linewidth=0, antialiased=False)
```

Where the keyword argument `cmap=` assigns the colors to the surface.

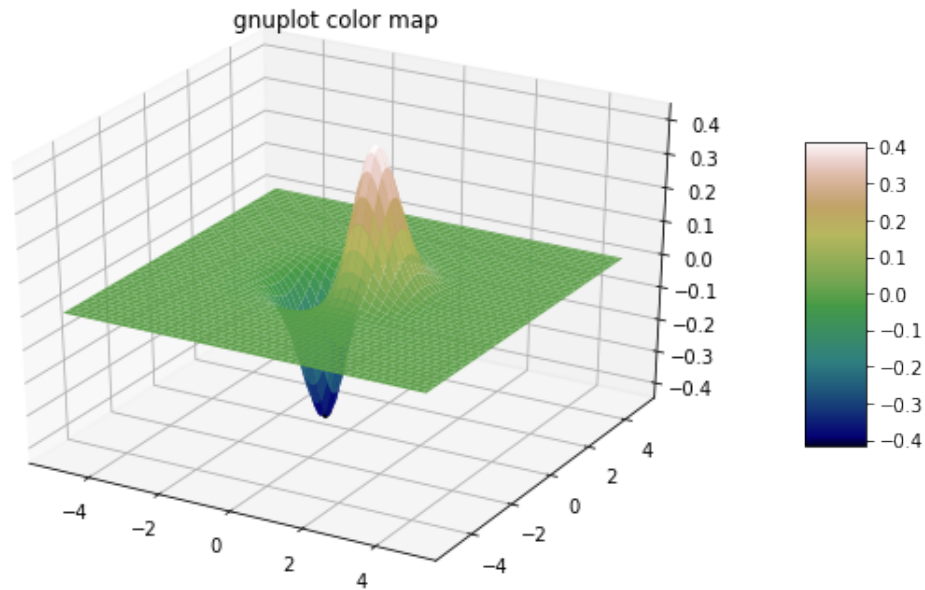
```
In [5]: from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(10,6))
ax1 = fig.add_subplot(111, projection='3d')

x = np.arange(-5,5,0.1)
y = np.arange(-5,5,0.1)
X,Y = np.meshgrid(x,y)
Z = X*np.exp(-X**2 - Y**2)

mycmap = plt.get_cmap('gist_earth')
ax1.set_title('gnuplot color map')
surf1 = ax1.plot_surface(X, Y, Z, cmap=mycmap)
fig.colorbar(surf1, ax=ax1, shrink=0.5, aspect=5)

plt.show()
```



3D Surface Plots with 2D contour plot projections

3D Surface Plots can be projected onto 2D surfaces. Below is sample code:

```
In [6]: from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure(figsize=(12,6))
ax = fig.add_subplot(111, projection='3d')
#ax = fig.gca(projection='3d')

x = np.arange(-5,5,0.1)
y = np.arange(-5,5,0.1)
X,Y = np.meshgrid(x,y)
Z = X*np.exp(-X**2 - Y**2)

mycmap = plt.get_cmap('gist_earth')
surf = ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.8, cmap=mycmap)
cset = ax.contourf(X, Y, Z, zdir='z', offset=np.min(Z), cmap=mycmap)
cset = ax.contourf(X, Y, Z, zdir='x', offset=-5, cmap=mycmap)
cset = ax.contourf(X, Y, Z, zdir='y', offset=5, cmap=mycmap)

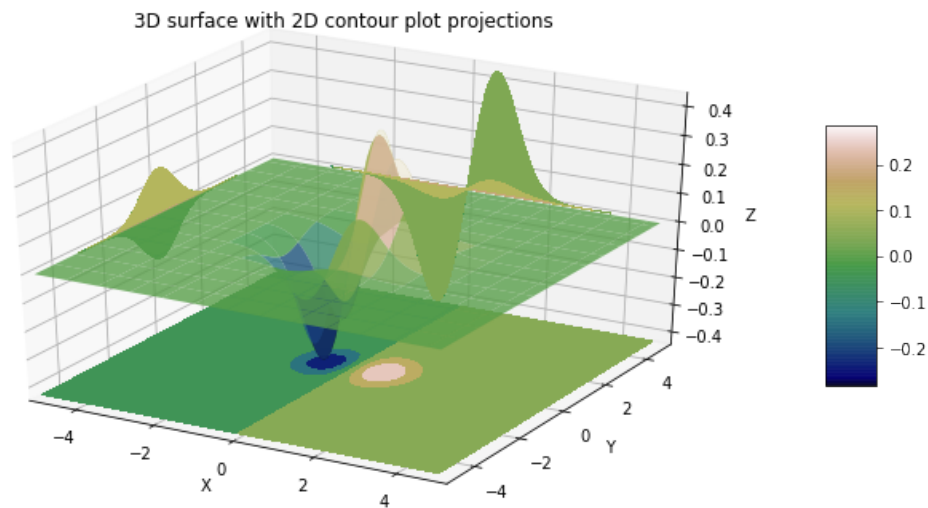
fig.colorbar(surf, ax=ax, shrink=0.5, aspect=5)
```

```

ax.set_xlabel('X')
ax.set_xlim(-5, 5)
ax.set_ylabel('Y')
ax.set_ylim(-5, 5)
ax.set_zlabel('Z')
ax.set_zlim(np.min(Z), np.max(Z))
ax.set_title('3D surface with 2D contour plot projections')

plt.show()

```



7.17 Summary

In this chapter you learned how to create plots using Python and **matplotlib**.

Types of charts: * line graphs * bar graphs * pie charts * bar and line graphs with error bars * scatter plots * histograms * box plots and violin plots * quiver plots * heat maps

Key Terms and Concepts

plot

object

attribute

object oriented programming

method

Additional Resources

Matplotlib official documentation: <https://matplotlib.org/contents.html>

Matplotlib summary notebook on Kaggle: <https://www.kaggle.com/grroverpr/matplotlib-plotting-guide/notebook>

Python Plotting With Matplotlib (Guide) on Real Python: <https://realpython.com/python-matplotlib-guide/#why-can-matplotlib-be-confusing>

Python For Data Science: Matplotlib Cheat Sheet from DataCamp: https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Python_Matplotlib_Cheat_Sheet.pdf

7.18 Review Questions

6.1

6.2

6.3

6.4

6.5

6.6

6.7

6.8

6.9

Chapter 8

If Else Try Except

8.1 Introduction

After completing the following chapter, you will be able to:

- Use if, else if and else selection structures
- Use the pass and try and except key words
- Construct flow charts that describe how a script operates

8.2 Selection Statements

Selection statements are used in programming to decide or select particular blocks of code to run based on a defined logical condition. The primary selection structures in Python are:

- if
- else
- elif
- try
- except

8.3 If statements

The *if* statement is one of the basic selection structures in Python. The syntax for a section of code that uses an if statement is below:

```
if <logical_condition>:  
    <code to run>
```

The keyword *if* begins the selection statement. Following *if* a logical condition needs to be included. A logical condition is an expression that can be evaluated as True or False. An example

of logical condition is `a<5`. This logical condition will return `True` if `a` is less than 5. Otherwise if `a` is 5 or greater it will return `False`. Following the logical condition, a colon `:` is needed. After the if statement, a chunk of code to run when the condition is `True` is included. The section of code must be indented and indented the same amount. By convention this is 4 spaces. Most Python code editors, including Jupyter Notebooks will indent code after an if statement automatically.

An example set of code below shows how an if statement might work:

```
In [7]: a = 2
        if a<5:
            print('less than five')
```

```
less than five
```

In the first line of the code example above, the variable `a` is assigned the value 2. The second line of code is the if statement. The if-statement starts with the keyword `if` and is followed by the logical condition `a<5` and a colon `:`. The logical condition `a<5` will return either `True` or `False` depending on the value of `a`. Since `a=2`, the logical condition `a<5` evaluates as `True`. The line `print('less than five')` is indented after the if-statement. This is the line of code that will run if the if-statement is `True`. Since the if-statement is `True` the indented line `print('less than five')` runs and the user sees the text `less than five`.

Multiple if statements

If statements can be chained together one after another to create a programmatic flow. For example, the following code block utilizes three different if-statements, each followed by an indented code block.

```
In [3]: a = 2
        if a<0:
            print('is negative')
        if a == 0:
            print('is zero')
        if a>0:
            print('is positive')
```

```
is positive
```

Note how each if-statement is followed by a logical condition and a colon `:`. Also note how the code below each if statement is indented. With the code left-justified (not indented), all three code lines will run and the output will be different. The `pass` keyword is used as code will not run unless at least one line of code is indented after the if-statement. `pass` is a line of code that does nothing.

```
In [6]: a = 2
        if a<0:
            pass
```

```
print('is negative')
if a == 0:
    pass
print('is zero')
if a>0:
    pass
print('is positive')
```

```
is negative
is zero
is positive
```

8.4 If-Else Statements

If statements can include *else* clauses. An *else* clause is a section of code that run if the if statement is False. The general form is:

```
if <true_condition>:
    <code block 1>
else:
    <code block 2>
```

The *else* key word needs to be on it's own line and be at the same indentation level as the *if* keyword that it corresponds to. *else* needs to be followed by a colon : and any code that is to run as part of the *else* statement must be indented the same amount. A sample if/else code section is below:

```
In [4]: a = 5
        if a>10:
            print('a is greater than 10')
        else:
            print('a is less than 10')
```

```
a is less than 10
```

Since *a=5* assigns a value to *a* that is less than 10, the code under the *if* statement does not run. Therefore the code under the *else* statement does run and *a is less than 10* is printed. If the value of *a* is modified so that it is greater than 10, the code under the *if* statement will run, and the code under the *else* keyword will not.

```
In [5]: a = 20
        if a>10:
            print('a is greater than 10')
        else:
            print('a is less than 10')
```

```
a is greater than 10
```

elif

The *else if* statement can be added to an if statement to run different sections of code depending on which one of many conditions are True. The basic syntax of an else if section of code is:

```
if <true_condition>:
    <code block 1>
elif <true_condition>:
    <code block 2>
else:
    <code block 3>
```

The keyword `elif` must be followed by a logical statement that evaluates to True or False followed by a colon `:`. The `<code block>` will run if the else if condition is True and will be skipped if the else if condition is False. An example section of code is below:

```
In [7]: color = 'green'
        if color == 'red':
            print('The color is red')
        elif color == 'green':
            print('The color is green')
        else:
            print('The color is not red or green')
```

The color is green

If we modify the code to assign the string 'orange' to the variable `color`, the code under the `if` will not run and the code under the `elif` will not run either. Only the code under the `else` will be executed.

```
In [8]: color = 'orange'
        if color == 'red':
            print('The color is red')
        elif color == 'green':
            print('The color is green')
        else:
            print('The color is not red or green')
```

The color is not red or green

8.5 Try-Except Statements

8.6 Flow Charts

Flow charts are used to graphically represent the flow of a program. There are for basic shapes used in a flow chart, each with a specific use:

- oval: start and stop
- diamond: selection structures and loops
- parallelogram: input and output
- rectangle: calculations

Below is a sample program:

```
In [2]: # start
        num = input('Enter a number: ')
        num = float(num)
        if num>0:
            print('num greater than zero')
        if num<0:
            print('num less than zero')
        print('Done')
        # end
```

```
Enter a number: 8
num greater than zero
Done
```

8.7 Summary

Key Terms and Concepts

selection structures

if

else

else-if

exceptions

try

flow chart

decision tree

8.8 Review Questions

In []: 1.

2.

3.

4.

5.

6.

Chapter 9

Loops

9.1 Introduction

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

- use a while loop
- use a for loop
- use the break statement
- use the continue statement
- construct flow charts that describe a program which contains a loop

9.2 While Loops

A *While Loop* is a type of loop that will continue running as long as a condition is True. When the condition becomes False the loop will stop running. The general form of a while loop is below:

```
while <true_statement>:  
    <code>
```

The keyword `while` must be included, as well as a `<true_statement>` that can be evaluated as True or False. The `<code>` after the while statement must be indented and each line of code that will run as part of the loop needs to be indented the same amount. An example is below:

```
In [1]: i = 0  
        while i<4:  
            print(i)  
            i = i+1
```

```
0  
1
```

2
3

The first line `i=0` created a variable `i` and assigned it the value 0. The next line declared what condition was needed to keep the loop running. The statement `i<4` will be `True` or `False` depending on the variable `i`. Since `i=0`, the statement `i<4` is `True` and the `for` loop starts to run. The code that run inside the loop prints the value of `i` then increases `i` by 1. When `i=4`, the statement `i<4` is `False` and the `while` loop ends.

Using a while loop to validate user input

While loops can be utilized to validate user input. Say we want to insist that a user inputs positive number. We can code this into a `while` loop that keeps on repeating until the user enters valid input. The code below will continue to ask a user for a positive number until one is entered.

```
In [1]: num_input = -1
        while num_input < 0:
            str_input = input('Enter a positive number: ')
            num_input = float(str_input)
```

Enter a positive number: 4

In the section of code above, it is important to initialize the variable `num_input` with a value that causes the `while` statement `num_input < 0` to evaluate as `True`. `num_input = -1` causes the statement `num_input < 0` to evaluate as `True` and any other negative number would have worked as well. If the `while` statement can't be evaluated as `True` or `False`, Python will throw an error. It is therefore necessary to convert the user's input from a string to a float. The statement `'5' < 0` does not evaluate to `true` or `false`, because the string `'5'` can't be compared to the number 0.

9.3 For Loops

For Loops are a component of many programming languages. A *For Loop* is a programming structure where a section of code is run a specified number of times.

Say we want to print out the statement:

```
Engineers solve problems in teams
Engineers solve problems in teams
Engineers solve problems in teams
```

We want to see the statement printed three times. One way to do accomplish this is with three print statements in a row:

```
In [1]: print('Engineers solve problems in teams')
        print('Engineers solve problems in teams')
        print('Engineers solve problems in teams')
```

```
Engineers solve problems in teams
Engineers solve problems in teams
Engineers solve problems in teams
```

For Loops with range()

Another way to accomplish the same thing is to use a for loop. The basic structure of a for loop is below:

```
for <var> in range(<num>):
    <statements>
```

Where <var> can be any variable name, range(<num>) is the number of times the for loop will run and <statements> are the lines of code that execute each time the loop runs. Note the for loop starts with the keyword for and includes a colon :. Both for and the colon : are required. Also note that <statements> was indented. Each statement that will run in the for loop need to be indented the same amount. The standard indentation is 4 spaces. Let's rewrite our example above using a for loop:

```
In [2]: for i in range(3):
        print('Engineers solve problems in teams')
```

```
Engineers solve problems in teams
Engineers solve problems in teams
Engineers solve problems in teams
```

The Python range() function will return a iterable list of values starting from zero and ending at n-1. For instance, when range(3) is called, the values 0, 1, 2 are returned. Note that 3 is not part of the iterable, even though the function input was range(3). This can be confirmed with a for loop:

```
In [3]: for i in range(3):
        print(f'This is range number: {i}')
```

```
This is range number: 0
This is range number: 1
This is range number: 2
```

For loops with lists

For loops can also be run using lists. If a list is used, the loop will run as many times as there are items in the list. The general structure is:

```
for <var> in <list>:
    <statements>
```

Where <var> is a variable name assigned to the item in the list, <list> is the list object and <statements> are the programming statements that run for each item in the list. An example is below:

```
In [5]: my_list = ['electrical', 'mechanical', 'civil']
        for item in my_list:
            print(item)
```

```
electrical
mechanical
civil
```

Note how the loop ran three times, because there were three items in the list. Each time through the loop the variable `item` was set to one particular item in the list. The first time through the loop, `item='electrical'`. The second time through the loop `item='mechanical'` and the third time through the loop `item='civil'`.

9.4 Break and Continue

Break and continue are two ways to modify the behavior of a loop

9.5 Flow Charts

Flow charts show the flow of a program graphically.

9.6 Summary

The summary from this chapter

Key Terms and Concepts

- loop
- while loop
- for loop
- break statement
- continue statement
- infinite loop
- flow chart

Chapter 10

Matricies and Arrays

10.1 Introduction

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

- Define a **numpy** array
- Modify a **numpy** array
- Index a **numpy** array
- Run mathematical operations on **numpy** arrays
- Solve a system of linear equations using matrices

10.2 Numpy

Numpy is a Python package used for numerical calculations, working with arrays of a homogeneous data type and scientific computing.

In previous chapters, **numpy** was used for the different functions and methods it provides. In addition to **numpy** math functions such as `np.sin()` **numpy** can also be used to construct homogeneous arrays and preform mathematical operations on arrays. A **numpy** array is different from a Python list. The data types stored in a Python list can all be different:

```
python_list = [ 1, -0.038, 'gear', True]
```

The list above contains four different data types 1 is an integer, -0.038 is a float, 'gear' is a string, and 'True' is a boolean.

```
In [1]: python_list = [1, -0.038, 'gear', True]
        for item in python_list:
            print(type(item))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

If the same four elements are stored in a **numpy** array, **numpy** will force all of the four items to conform to the same data type. In this case, all four items are converted to type '`<U32`', which is a string data type in **numpy**. **Numpy** arrays can also be two-dimensional, three-dimensional, or up to n-dimensional.

```
In [2]: import numpy as np
        np.array([1, -0.038, 'gear', True])
```

```
Out[2]: array(['1', '-0.038', 'gear', 'True'], dtype='<U32')
```

Numpy arrays are useful because mathematical operations can be run on the entire array simultaneously. If a list of numbers is stored in a regular Python list, when the list is multiplied by a scalar the list extends and repeats instead of multiplying each number in the list by the scalar.

```
In [3]: lst = [1, 2, 3, 4]
        lst*2
```

```
Out[3]: [1, 2, 3, 4, 1, 2, 3, 4]
```

To multiply each element of a Python list by the scalar number 2, a loop can be used:

```
In [4]: lst = [1, 2, 3, 4]
        for i, item in enumerate(lst):
            lst[i] = lst[i]*2
        lst
```

```
Out[4]: [2, 4, 6, 8]
```

The method above is fairly cumbersome and is also quite *computationally expensive*. An operation that is computationally expensive is an operation that takes a lot of processing time and/or storage resources like RAM. Another way of completing this same action is to use a **numpy** array. The **numpy** array can be multiplied by a scalar and this will produce an array with each element multiplied by the scalar.

```
In [5]: nparray= np.array([1,2,3,4])
        2*nparray
```

```
Out[5]: array([2, 4, 6, 8])
```

If we have a very long list, we can compare the amount of time it takes for each operation. Jupyter Notebooks have a nice built-in way to time how long it takes a line of code to execute. In a Jupyter

Notebook, when a line start with `%timeit` followed by code, the notebook will run the line of code multiple times and output an average of the time spent to complete the line of code. We can use `%timeit` to compare an mathematical operation on a Python list using a for loop to the same mathematical operation on a **numpy** array.

```
In [6]: lst = list(range(10000))
        %timeit for i, item in enumerate(lst): lst[i] = lst[i]*2
```

1.96 ms ± 97 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

```
In [7]: nparray= np.arange(0,10000,1)
        %timeit 2*nparray
```

10.6 µs ± 8.35 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

With 10,000 integers, the Python list and for-loop takes an average of single milliseconds, while the **numpy** array completing the same operation takes tens of microseconds. This is a speed increase of over 100x by using the **numpy** array. For larger lists and **numpy** arrays the speed increase using **numpy** is considerable.

10.3 Installing Numpy

Before **numpy**'s functions and methods can be used, **numpy** must be installed. Depending on which distribution of Python is being used, the installation methods are slightly different.

Install numpy on Anaconda

If the Anaconda distribution of Python is installed, **numpy** comes pre-packaged and no further installation steps are necessary.

If using a version of Python from python.org or a version of Python that came with the operating system, the **Anaconda Prompt** and **conda** or **pip** can be used to install **numpy**.

Install numpy with the Anaconda Prompt

To install **numpy**, open the **Anaconda Prompt** and type:

```
> conda install numpy
```

Type y for yes when prompted.

Install numpy with pip

To install **numpy** with **pip**, bring up a terminal window and type:

```
$ pip install numpy
```

This will install **numpy** in the current working Python environment.

Verify numpy installation

To verify **numpy** is installed, try to invoke **numpy** version at the Python REPL by calling the `.__version__` attribute common to most Python packages.

```
In [3]: import numpy as np
        np.__version__
```

```
Out[3]: '1.14.2'
```

10.4 Array Creation

Numpy arrays are created with the `np.array()` function. The arguments provided to `np.array()` needs to be a list or iterable. An example is below. Note how the list `[1,2,3]` is passed into the function with square brackets at either end.

```
In [3]: import numpy as np
        np.array([1,2,3])
```

```
Out[3]: array([1, 2, 3])
```

The data type can be passed into the `np.array()` function as a second optional positional arguments. Available data types include `'int64'`, `'float'`, `complex` and `>U32` (string data type).

```
In [5]: import numpy as np
        np.array([1,2,3],dtype='float')
```

```
Out[5]: array([1., 2., 3.])
```

The data type store in a **numpy** array can be determined using the `.dtype` method. For instance, an array of floats will return `float64`.

```
In [12]: import numpy as np
         my_array = np.array([1,2,3],dtype='float')
         my_array.dtype
```

```
Out[12]: dtype('float64')
```

Arrays of Regularly Spaced Numbers

`np.arange()`

Numpy's `np.arange()` function will create a **numpy** array according the arguments `start`, `stop+1`, `step`.

```
my_array = np.arange(start, stop+1, step)
```

This function is useful for creating an array of regularly spaced numbers. Consider creating a numpy array of even numbers between 0 and 10. Note that just like counting in Python, counting in numpy starts at zero and ends at `n+1`.

```
In [19]: np.arange(0,10+2,2)
```

```
Out[19]: array([ 0,  2,  4,  6,  8, 10])
```

`np.linspace()`

Numpy's `np.linspace()` function will create a **numpy** array according the arguments `start`, `stop`, `number of elements`.

```
my_array = np.linspace(start, stop, number of elements)
```

This function is useful for creating an array of regularly spaced numbers where the spacing is not known but the number of values is. Consider creating a **numpy** array of 10 numbers between 0 and 2π .

```
In [20]: np.linspace(0,2*np.pi,10)
```

```
Out[20]: array([0.          , 0.6981317 , 1.3962634 , 2.0943951 , 2.7925268 ,
               3.4906585 , 4.1887902 , 4.88692191, 5.58505361, 6.28318531])
```

`np.logspace()`

Numpy's `np.logspace()` function will create a **numpy** array according the arguments `start`, `stop`, `number of elements`, but unlike `np.linspace()`, `np.logspace()` will produce a logarithmically spaced array.

```
my_array = np.logspace(start, stop, number of elements)
```

This function is useful for creating an array of logarithmically spaced numbers where the spacing interval is not known but the number of values is. Consider creating a **numpy** array of 10 logarithmically spaced numbers between 0.1 and 1.

```
In [26]: np.logspace(0.1, 1, 10)
```

```
Out[26]: array([ 1.25892541,  1.58489319,  1.99526231,  2.51188643,  3.16227766,
               3.98107171,  5.01187234,  6.30957344,  7.94328235, 10.          ])
```

np.zeros()

Numpy's `np.zeros()` function will create a **numpy** array containing zeros of a specific size. This is often useful when the size of an array is known, but the values that will go into it have not been created yet.

```
my_array = np.zeros((rows,cols))
```

```
In [53]: np.zeros((5,5))
```

```
Out[53]: array([[0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.]])
```

np.ones()

Numpy's `np.ones()` function will create a **numpy** array containing all 1's of a specific size. This is often useful when the size of an array is known, but the values that will go into it have not been created yet.

```
my_array = np.ones((rows,cols))
```

```
In [54]: np.ones((3,5))
```

```
Out[54]: array([[1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.]])
```

Arrays of Random Numbers

Numpy has functions to create many different types of random number arrays in the `np.random` module. A few of the common random number types are below.

Array of Random Integers

Arrays of random integers can be created with **numpy's** `np.random.randint()` function. The general syntax is:

```
np.random.randint(lower limit, upper limit, number of values)
```

To create an array of 5 different random numbers, each random number between 1 and 10:

```
In [30]: np.random.randint(0,10,5)
```

```
Out[30]: array([9, 4, 9, 1, 9])
```

A multi-dimensional size can be provided as the third argument. A 5 x 5 array of random numbers between 1 and 10:

```
In [31]: np.random.randint(0,10,[5,5])
```

```
Out[31]: array([[8, 1, 6, 7, 7],
                [9, 9, 0, 7, 7],
                [6, 2, 1, 0, 2],
                [8, 3, 4, 2, 1],
                [4, 5, 1, 0, 3]])
```

Array of Random Floats

Arrays of random floating point numbers can be created with **numpy's** `np.random.rand()` function. The general syntax is:

```
np.random.rand(number of values)
```

To create an array of 5 different random numbers, each random number between 0 and 1:

```
In [36]: np.random.rand(5)
```

```
Out[36]: array([0.02421777, 0.06312956, 0.45358935, 0.72194406, 0.65034115])
```

Random Array Choice from a List

```
np.random.choice(list of choices, number of choices)
```

To choose three numbers at random from a list of [1,5,9,11] use:

```
In [37]: lst = [1,5,9,11]
         np.random.choice(lst,3)
```

```
Out[37]: array([1, 1, 9])
```

Random Array with a Normal Distribution

`np.random.randn()` will return a random array of numbers with a normal distribution, assuming a mean of 0 and variance of 1.

```
np.random.randn(number of values)
```

```
In [38]: np.random.randn(10)
```

```
Out[38]: array([-1.88792019,  0.5247404 , -1.16660705,  0.68567961, -0.57930791,
                -0.54634836,  0.1534303 , -1.64928217, -0.01712827,  0.9343703 ])
```

To specify a mean `mu` and a standard deviation `sigma`, the function can be wrapped with:

```
In [39]: mu = 70
         sigma = 6.6

         sigma * np.random.randn(10) + mu

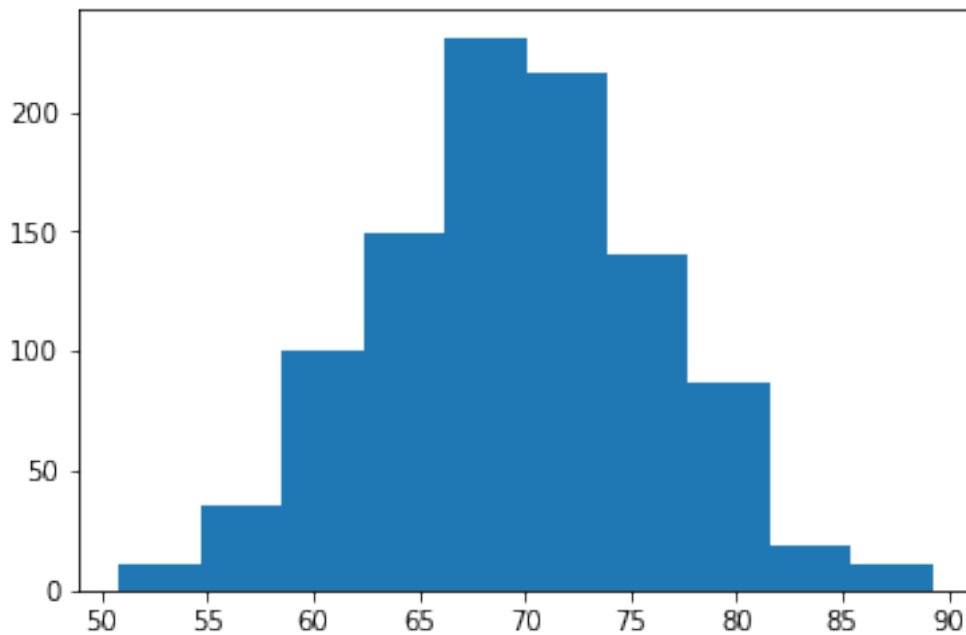
Out[39]: array([65.56436887, 70.7579809 , 78.70706918, 76.5538997 , 72.83272356,
                62.17107227, 57.15193348, 66.07326289, 75.47326815, 76.13660167])
```

To quickly plot a normal distribution, **matplotlib** `plt.hist()` function can be used.

```
In [43]: import matplotlib.pyplot as plt
         import numpy as np
         %matplotlib inline

         mu = 70
         sigma = 6.6

         sample = sigma * np.random.randn(1000) + mu
         plt.hist(sample)
         plt.show()
```



2-D Arrays

np.meshgrid

```
In [47]: x = np.arange(0,6)
         y = np.arange(0,11,2)
         X, Y = np.meshgrid(x,y)
         print(X)
         print(Y)
```

```
[[0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [0 1 2 3 4 5]]
[[ 0  0  0  0  0  0]
 [ 2  2  2  2  2  2]
 [ 4  4  4  4  4  4]
 [ 6  6  6  6  6  6]
 [ 8  8  8  8  8  8]
 [10 10 10 10 10 10]]
```

np.mgrid

```
In [52]: X, Y = np.mgrid[0:5,0:11:2]
         print(X)
         print(Y)
```

```
[[0 0 0 0 0 0]
 [1 1 1 1 1 1]
 [2 2 2 2 2 2]
 [3 3 3 3 3 3]
 [4 4 4 4 4 4]]
[[ 0  2  4  6  8 10]
 [ 0  2  4  6  8 10]
 [ 0  2  4  6  8 10]
 [ 0  2  4  6  8 10]
 [ 0  2  4  6  8 10]]
```

Section Summary

Below is a list of numpy functions used in this section

np.array np.arange np.linspace np.logspace np.zeros np.ones np.random.rand np.random.randint
np.random.randn np.meshgrid np.mgrid

10.5 Array Indexing

Elements in a numpy array can be accessed using *indexing*. Indexing is an operation that pulls out a select set of values from an array. The *index* of a value in an array is that value's location. There is a difference between the value that is stored and the index or location of that value within an array. Take the following example. Remember that counting in Python starts at zero.

```
In [2]: import numpy as np

a = np.array([2,4,6])
```

In the array above, there are three values stored: 2, 4 and 6. Each of these values has a different index. The value 2 has an index of 0. 2 is in the 0 location in the array. The value 4 has an index of 1 and the value 6 has an index of 2. A table summarizes this below:

| Index or Location | Value |
|-------------------|-------|
| 0 | 2 |
| 1 | 4 |
| 2 | 6 |

Individual values stored in array can be accessed using indexing. The general form is:

```
<value> = <array>[index]
```

Where <value> is the value store in the array, <array> is the array object name and [index] specifies the index or location of that value. Applied to our example above, the value 6 is stored at index 2.

```
In [3]: import numpy as np

a = np.array([2,4,6])
value = a[2]
print(value)
```

6

Multi-dimensional Array Indexing

Multidimensional arrays are indexed as well. A simple 2-D array is defined by a list of lists.

```
In [5]: import numpy as np

a = np.array([[2,3,4],[6,7,8]])
print(a)
```



```
[[2 3 4]
 [6 7 8]]
```

Values in a 2-D array can be accessed using the general notation below:

```
<value> = <array>[row,col]
```

Where <value> is the value pulled out of the 2-D array named <array>. [row,col] specifies the row and column index of the value. Remember that Python counting starts at 0, so the first row is row zero and the first column is column zero.

We can access the value 8 in the array above by calling the row and column index [1,2] for the 2nd row (remember row 0 is the first row) and the 3rd column (remember column 0 is the first column).

```
In [6]: import numpy as np

        a = np.array([[2,3,4],[6,7,8]])
        value = a[1,2]
        print(value)
```

```
8
```

Assigning Values with Indexing

Array indexing can be used to access values in an array and they can also be used for *assigning* values of an array. In order to assign a value to a particular index or location in an array, the following general form is used:

```
<array>[index] = <value>
```

Where <array> is the array object that value will be assigned to, [index] is the index or location the value will be put in and <value> is the value assigned to that location.

If we want to put the value 10 into the third index or location of an array, it can be coded as follows:

```
In [8]: import numpy as np
        a = np.array([2,4,6])
        a[2] = 10
        print(a)
```

```
[ 2  4 10]
```

Values can also be assigned into 2-D arrays using the form

```
<array>[row,col] = <value>
```

An example is below:

```
In [10]: import numpy as np

a = np.array([[2,3,4],[6,7,8]])
print(a)

a[1,2]=20
print(a)
```

```
[[2 3 4]
 [6 7 8]]
[[ 2  3  4]
 [ 6  7 20]]
```

10.6 Array Slicing

Multiple values stored within an array can be accessed simultaneously with array *slicing*. To pull out a section or slice of an array, the colon operator `:` is used when calling the index. The general form is:

```
<slice> = <array>[start:end+1]
```

Where `<slice>` is the slice or section of the array object `<array>`. The index of the slice is specified in `[start:end+1]`. Note that Python counting starts at zero and ends at `n+1`. To pull the first 2 values out of an array, the indexing operation is `[0:2]`. An example is below:

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

a = np.array([2, 4, 6])
b = a[0:2]
print(b)
```

```
[2 4]
```

A blank stands in for the last index. A slicing operation `[1:]` will pull out the 2nd through last value of an array

```
In [3]: import numpy as np

a = np.array([2, 4, 6, 8])
b = a[1:]
print(b)
```

```
[4 6 8]
```

A blank also stands in for the first index. The slicing operation `[:3]` will pull out the 1st through third values of an array

```
In [4]: import numpy as np

        a = np.array([2, 4, 6, 8])
        b = a[:3]
        print(b)
```

```
[2 4 6]
```

Slicing 2D Arrays

2D arrays can be sliced with the general form

```
<slice> = <array>[start_row:end_row+1, start_col:end_col-1]
```

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

        a = np.array([[2, 4, 6, 8], [10, 20, 30, 40]])
        b = a[0:2, 0:3]
        print(b)
```

```
[[ 2  4  6]
 [10 20 30]]
```

Again, a blank represents the first index or the last index. The colon operator also represents “all”. To slice out the first two rows and all columns that code is:

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

        a = np.array([[2, 4, 6, 8], [10, 20, 30, 40]])
        b = a[:2, :] #[first two rows, all columns]
        print(b)
```

```
[[ 2  4  6  8]
 [10 20 30 40]]
```

10.7 Array Operations

Mathematical operations can be completed using numpy arrays.

Scalar Addition

Scalars can be added and subtracted from arrays and arrays can be added and subtracted from each other:

```
In [2]: import numpy as np

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

```
[3 4 5]
```

```
In [3]: import numpy as np

        a = np.array([1, 2, 3])
        b = np.array([2, 4, 6])
        c = a + b
        print(c)
```

```
[3 6 9]
```

10.8 Scalar Multiplication

Numpy arrays can be multiplied and divided by scalars:

```
In [4]: import numpy as np

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

```
[3 6 9]
```

```
In [5]: import numpy as np

        a = np.array([10,20,30])
        b = a/2
        print(b)
```

```
[ 5. 10. 15.]
```

Matrix Multiplication

Numpy array can be multiplied by each other using matrix multiplication. These matrix multiplication methods include element-wise multiplication, the dot product and the cross product

Element-wise multiplication

```
In [14]: import numpy as np
```

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

```
Out[14]: array([ 4, 10, 18])
```

Dot Product

```
In [13]: import numpy as np
```

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

```
Out[13]: 32
```

Cross Product

```
In [12]: import numpy as np
```

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

```
Out[12]: array([-3,  6, -3])
```

Exponents and Logarithms

```
np.exp()
np.log() - natural log
np.log2() base-2 log
np.log10() base 10 log
np.e
```

Trigonometry

```
sin()
cos()
tan()
arcsin()
arccos()
arctan()
hypot() Given the sides of a right triangle, returns the hypotenuse.
```

```
degrees()
radians()
deg2rad()
rad2deg()
```

10.9 Systems of Linear Equations

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

A system of linear equations is shown below

$$8x + 3y - 2z = 9$$

$$-4x + 7y + 5z = 15$$

$$3x + 4y - 12z = 35$$

The `np.linalg.solve()` function can be used to solve this system for the variables x , y and z . Set a numpy array `A` as a 3 by 3 array of the coefficients. Set a numpy array `b` as the right-hand side of the equations. Then solve for the values of x , y and z using `np.linalg.solve(A, b)`. The resulting array will have three entries. One entry for each variable.

```
In [2]: A = np.array([[8, 3, -2], [-4, 7, 5], [3, 4, -12]])
        b = np.array([9, 15, 35])
        x = np.linalg.solve(A, b)
        x
```

```
Out[2]: array([-0.58226371,  3.22870478, -1.98599767])
```

```
In [16]: x[0]
```

```
Out[16]: -0.5822637106184365
```

We can plug the valuse of x , y and z back into one of the equations to check the answer.

$$8x + 3y - 2z = 9$$

Where x is the first entry in the array, y is the second entry in the array and z is the third entry in the array.

```
x = x[0]
```

```
y = x[1]
```

```
z = x[2]
```

When these values are plugged into the equation above, the answer should be 9.0.

```
In [4]: 8 * x[0] + 3 * x[1] - 2 * x[2]
```

```
Out[4]: 9.0
```

Another example from engineering statics is the following:
three forces operate on a point. The forces are defined as:

F1 =

F2 =

F3 =

What is the tension in AC and BC?

10.10 Summary

Key Terms and Concepts

numpy

array

matrix

computationally expensive

slice

index

homogenous data type

Numpy Functions and Methods

`np.array()`

10.11 Review Questions

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.
- 7.

8.

9.

Chapter 11

Symbolic Math

11.1 Introduction

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

- Define Python variables as symbolic math variables
- Define mathematical equations using symbolic math variables
- Solve for symbolic math variables in terms of other symbolic math variables
- Derive numerical solutions using symbolic math variable substitution
- Simplify equations using symbolic math variables

11.2 SymPy

SymPy <http://www.sympy.org> is a Python library for **symbolic math**.

Symbolic Math is using symbols to represent mathematical expressions. An example symbolic math expression is below:

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

In the expression, we have the variables x , y and z .

If we define a second symbolic math expression as:

$$x = a + b$$

then we can substitute in $a + b$ for x .

This would result in the expression:

$$(a + b)^2 + y^2 = z$$

$$a^2 + 2ab + b^2 + y^2 = z$$

Now to solve for y in terms of a, b and z , would result in:

$$y = \sqrt{z - a^2 - 2ab - b^2}$$

If we have numerical values for z, a and b , we can use Python to calculate the value of y . But if we don't have numerical values for z, a and b Python can also be used to rearrange terms and solve for one variable in terms of the other. Working with mathematical symbols in a programatic way instead of working with numerical values in a programatic way is call symbolic math.

To work with symbolic math, the **SymPy** library needs to be installed. Using the **Anaconda Prompt** this can be accomplished with the command:

```
$ conda install sympy
```

Sympy is included with the Anaconda distribution of Python, so if you have the full Anaconda distrobution, you will be notified that the **Sympy** library is already installed.

11.3 Defining Variables

To define variables with **SymPy**, first import the `symbols()` function from the **SymPy** module:

```
In [1]: from sympy import symbols
```

Symbolic math symbols are declared using **SymPy's** `symbols()` function. Note that in the arguments of the `symbols()` function, symbol names are separated by a space (no comma) and surrounded by quotes. The output of the `symbols()` function are **SymPy** symbol objects. These output objects need to be separated by comas with no quotation marks.

```
In [2]: x, y = symbols('x y')
```

Substitution

Now that the symbols x and y are instantiated, a symbolic math expression using x and y can be created. A symbolic math expression is a combination of symbolic math variables with numbers and mathematical opporators (such as $+, -, /$ and $*$). The standard Python rules for calculating numbers apply in **SymPy** symbolic math expressions.

```
In [3]: expr = 2*x + y
```

Use the `.subs()` method to insert a numerical value into a symbolic math expression, `.`. The first argument of the `.subs()` method is the variable and the second argument is the numerical value. In the expression above:

$$2x + y$$

If we substitute

$$x = 2$$

The resulting expression should be

$$2(2) + y$$

$$4 + y$$

```
In [4]: expr.subs(x, 2)
```

```
Out[4]: y + 4
```

The `.subs()` method does not replace variables in place, it only completes a one-time substitution. If `expr` is called after the `.subs()` method is applied, the original `expr` expression is returned

```
In [5]: expr
```

```
Out[5]: 2*x + y
```

In order to make the substitution permanent, a new expression object needs to be instantiated as the output of the `.subs()` method.

```
In [6]: expr = 2*x + y
        expr2 = expr.subs(x, 2)
        expr2
```

```
Out[6]: y + 4
```

SymPy variables can also be substituted into **SymPy** expressions

```
In [7]: x, y, z = symbols('x y z')
        expr = 2*x + y
        expr2 = expr.subs(x, z)
        expr2
```

```
Out[7]: y + 2*z
```

More complex substitutions can also be used. Consider the following:

$$2x + y$$

substitute in

$$y = 2x^2 + z^{-3}$$

results in

$$2x + 2x^2 + z^{-3}$$

```
In [8]: x, y, z = symbols('x y z')
        expr = 2*x + y
        expr2 = expr.subs(y, 2*x**2 + z**(-3))
        expr2
```

```
Out[8]: 2*x**2 + 2*x + z**(-3)
```

A more practical example could involve a large equation and several variable substitutions

$$n_0 e^{-Q_v/RT}$$

$$n_0 = 3.48 \times 10^{-6}$$

$$Q_v = 12,700$$

$$R = 8.31$$

$$T = 1000 + 273$$

```
In [9]: from sympy import symbols, exp
        n0, Qv, R, T = symbols('n0 Qv R T')
        expr = n0*exp(-Qv/(R*T))
```

Multiply **SymPy** `subs()` methods can be chained together to substitute multiple variables in one line of code

```
In [10]: expr.subs(n0, 3.48e-6).subs(Qv,12700).subs(R, 8031).subs(T, 1000+273)
```

```
Out[10]: 3.48e-6*exp(-12700/10223463)
```

To evaluate an expression as a floating point number, use the `.evalf()` method

```
In [16]: expr2 = expr.subs(n0, 3.48e-6).subs(Qv,12700).subs(R, 8031).subs(T, 1000+273)
```

```
In [17]: expr2.evalf()
```

```
Out[17]: 3.47567968697765e-6
```

11.4 Defining Equations

Using symbolic math variables we can define equations using **SymPy**. *Equations* in **SymPy** are different than *expressions*. An expression does not have equality. An equation has equality.

```
In [28]: from sympy import symbols, Eq, solveset, solve
         from sympy.solvers.solveset import linsolve
```

```
In [29]: x, y = symbols('x y')
```

```
In [30]: eq1 = Eq(4*x+2*y - 3)
```

```
In [33]: eq2 = Eq(-2*x-6*y + 10)
```

```
In [37]: sol = solve((eq1, eq2),(x, y))
         sol
```

```
Out[37]: {x: -1/10, y: 17/10}
```

The **SymPy** solution object is a dictionary. The keys are the **SymPy** variable objects and the values are the numerical values these variables correspond to.

```
In [40]: print(f'The solution is x = {sol[x]}, y = {sol[y]}')
```

```
The solution is x = -1/10, y = 17/10
```

11.5 Solving two equations for two unknowns

SymPy can be used to solve two equations for two unknowns. Consider the following engineering statics problem:

GIVEN:

A mass of 22 lbs is hung from a ring. The ring is supported by two cords, cord CE is 30 degrees above the horizontal to the right and cord BD is 45 degrees to the left above the horizontal.

$m = 20 \text{ lb}$

TCE @ +30 degrees CCW relative to +x-axis

TBD @ +45 degrees CW relative to -x-axis

FIND:

magnitude of TCE and TBD

SOLUTION:

To solve for the magnitude of TCE and TBD, we will need to solve two equations for two unknowns. To accomplish this with **SymPy**, first we need to import **numpy** and **sympy**. The **SymPy** functions `symbols`, `Eq` and `solve` will all be needed.

```
In [4]: import numpy as np
        from sympy import symbols, Eq, solve
```

Next define the symbolic math variables (that will be used in the equations) as **SymPy** symbols objects. Multiple symbolic math variables can be defined at the same time. Note the argument names (on the right-hand side of the assignment operator =) need to be enclosed in quotes ' ' and separated by spaces, no commas. The object names (on the left-hand side of the assignment operator =) are separated with commas, no quotes.

```
In [5]: Tce, Tbd = symbols('Tce Tbd')
```

Next the two equations need to be defined. Assuming the ring is in static equilibrium:

$$\Sigma \vec{F} = 0$$

$$\Sigma F_x = 0$$

$$\Sigma F_y = 0$$

The three forces operating on the ring are defined as:

$$T_{ce} = \text{tension in cable CE}$$

$$\vec{T}_{ce} = T_{ce} \cos(30) \hat{i} + T_{ce} \sin(30) \hat{j}$$

$$T_{bd} = \text{tension in cable BD}$$

$$\vec{T}_{bd} = -T_{bd} \cos(45) \hat{i} + T_{bd} \sin(45) \hat{j}$$

$$\vec{m} = 0 \hat{i} - 22 \hat{j}$$

Taking $\Sigma F_x = 0$ (sum of the \hat{i} terms):

$$T_{ce} \cos(30) - T_{bd} \cos(45) = 0$$

Taking $\Sigma F_y = 0$ (sum of the \hat{j} terms):

$$T_{ce} \sin(30) + T_{bd} \sin(45) - 22 = 0$$

Our first equation, based on the sum of the forces in the x-direction (the \hat{i} terms) is:

$$T_{ce} \cos(30) - T_{bd} \cos(45) = 0$$

This equation can be represented as a **Sympy** equation object. Note the right-hand side of the equation is 0. **Sympy** equation objects are instantiated with expressions equal to zero. If the expression was not equal to zero, we would simply subtract both sides by the term on the right-hand side of the = equals sign and use the resulting expression (equal to zero) to create the equation object.

```
In [6]: eq1=Eq(Tce*np.cos(np.radians(30)) - Tbd*np.cos(np.radians(45)))
        print(eq1)
```

```
Eq(-0.707106781186548*Tbd + 0.866025403784439*Tce, 0)
```

Our second equation, based on the sum of the forces in the y-direction is:

$$T_{ce}\sin(30) + T_{bd}\sin(45) - 22 = 0$$

We can define this equation as a **Sympy** equation object as well:

```
In [7]: eq2=Eq(Tce*np.sin(np.radians(30)) + Tbd*np.sin(np.radians(45))-22)
        print(eq2)
```

```
Eq(0.707106781186547*Tbd + 0.5*Tce - 22, 0)
```

Now to solve the two equations for T_{ce} and T_{bd} we use sympy's solve method. The first argument is a tuple of the equations we want to solve, the second argument is the variables we want to solve for.

```
In [8]: solve((eq1,eq2),(Tce, Tbd))
```

```
Out[8]: {Tce: 16.1051177665153, Tbd: 19.7246603876972}
```

We end up with a dictionary, the keys are the variable names and the values are the solved numerical solution

Now let's solve the same problem, but keep m as a variable.

```
In [9]: m, Tce, Tbd = symbols('m, Tab, Tac')
```

```
In [10]: eq1=Eq(Tce*np.cos(np.radians(30)) - Tbd*np.cos(np.radians(45)))
        eq2=Eq(Tce*np.sin(np.radians(30)) + Tbd*np.sin(np.radians(45))-m)
```

```
In [11]: solve((eq1,eq2),(Tce,Tbd))
```

```
Out[11]: {Tab: 0.732050807568878*m, Tac: 0.896575472168054*m}
```

Now our solution is in terms of the variable m .

11.6 Summary

Key Terms and Concepts

symbolic math

symbolic variable

numerical calculation

systems of equations

expression

equation

11.7 Review Questions

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.
- 7.

Chapter 12

Python and External Hardware

12.1 Introduction

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

- Connect external hardware to a computer running Python
- Install the **PySerial** library
- Read data over a serial connection using Python
- Save data coming in over a serial line to Python variables and external files
- Write data to a serial line using Python
- Read data from a sensor using Python
- Control and LED using Python

12.2 Pyserial

Pyserial is a **Python** package that is used to assist in communication between external hardware and a computer running **Python**. **Pyserial** is a useful package for engineers because it allows engineers to exchange data between a computer and pieces of hardware such as voltmeters, oscilloscopes, strain gauges, flow meters etc.

Pyserial provides an interface to communicate over the *serial* communication protocol. *Serial* communication is one of the oldest computer communication protocols and predates the kind of communication now typical used between computers and other pieces of hardware such as USB and HDMI. USB stands for Universal Serial Bus and is built upon and extends the original serial communication interface.

Installing Pyserial

To use **pyserial** with **Python**, **pyserial** needs to be installed over the command line using **pip** or installed using the **Anaconda Prompt**:

```
> conda install pyserial
```

or

```
$ pip install pyserial
```

After installing **pyserial**, the successful installation can be confirmed at the Python REPL:

```
In [1]: >>> import serial
        >>> print(serial.__version__)
```

3.4

Note: even though **pyserial** is installed with `conda install pyserial`, the module is imported with the line `import serial`.

12.3 Bytes and Unicode Strings

Before using **pyserial** and communicating with external hardware, it is import to understand the difference between *bytes* and *unicode strings* in Python. The distinction between the two is important because strings in Python are *unicode* by default. However, external hardware like Arduino's, oscilloscopes and voltmeters transmit characters as *bytes*.

Unicode Strings

In Python when a new string is defined, the syntax is:

```
In [1]: ustring = 'A unicode string'
```

We can see what data type `ustring` is with the `type` function:

```
In [2]: print(type(ustring))
```

```
<class 'str'>
```

When Python shows the variable `ustring` is of `<class 'str'>`, it indicates `ustring` is a *unicode string*. **In Python 3 all strings are unicode strings by default.** *Unicode Strings* are useful because there are many letter and letter-like characters that are not part of the the set of letters, numbers and symbols on a regular computer keyboard. For instance, English is not the most widely spoken language on the planet. In Spanish, the accent character is used over certain vowels. Letters with accents can't be represented by the letters on a standard keyboard. However those accent letters are part of a set of letters, numbers and symbols in *unicode strings*.

Byte Strings

Another way that characters such as letters, numbers and punctuation can be stored is as *bytes*. A *byte* is a unit of computer information that has a fixed width (one byte long). Because of this fixed width, one *byte* only has a small number of unique combinations. This limits *byte strings* to basically only the letter, numbers and punctuation marks on a computer keyboard. This limited set of characters are called ASCII (pronounced *ask-ee two*) characters. A table of ASCII character codes is in the appendix. For instance, ASCII code 49 corresponds to the number one 1. **Python does not use *byte strings* by default.** But external hardware such as Arduinos, oscilloscopes, and voltmeters speak *byte strings* by default. In fact almost all machines speak *byte strings* by default including the servers that bring Netflix to your laptop.

To define a *byte string* in Python, a letter b is placed before the quotation marks when a string is created.

```
In [3]: bstring = b'bstring'
```

We can view the type of our bstring variable using the type function

```
In [4]: print(type(bstring))
```

```
<class 'bytes'>
```

Convert between unicode and bytes

In order for Python to communicate with external hardware and external hardware to communicate with Python, we need to be able to convert between *unicode strings* and *byte strings*. This conversion is done with the `.encode()` and `.decode()` methods.

The `.encode()` method “encodes” a unicode string to a byte string.

```
<byte string> = <unicode string>.encode()
```

The `.decode()` method “decodes” a byte string into a unicode string.

```
<unicode string> = <byte string>.decode()
```

Remember machines speak bytes, Python strings are unicode. We need to decode what machines transmit to Python before further processing. Python defaults unicode (and machines do not) so within our Python code we need to encode our unicode strings so machines can understand it.

```
In [5]: new_bstring = ustring.encode()  
        type(new_bstring)
```

```
Out[5]: bytes
```

```
In [6]: new_ustring = bstring.decode()  
        type(new_ustring)
```

```
Out[6]: str
```

If a command from Python (a unicode string) is sent to a piece of external hardware (that reads bytes), the `.encode()` method should be applied to the unicode string before being sent from Python to the piece of hardware.

If a chunk of data is coming in from a piece of external hardware (a byte string) and read by Python (which speaks *unicode*), the `.decode()` method should be applied to the byte string (to convert it to a unicode string) before it is processed further by Python.

12.4 Reading a Sensor with Python

In this section you will learn how to read a sensor connected to an external piece of hardware (an Arduino) using Python. To accomplish this the following hardware is required:

- A computer running Python
- An Arduino
- A potentiometer (the sensor)
- Wires, a resistor and a breadboard to connect the sensor to the Arduino
- A USB cable to connect the Arduino to the computer

Set up the sensor

Connect the sensor to the Arduino using a resistor, wires and a breadboard. Note the long lead of the LED is connected to PIN13 and the short lead of the LED is connected through a resistor to ground. If the LED is wired backwards, the LED will not turn on. The blue square with an arrow on it is a potentiometer. It is a good sensor to use because potentiometers can be controlled by a user and the user knows when the sensor signal is changed.

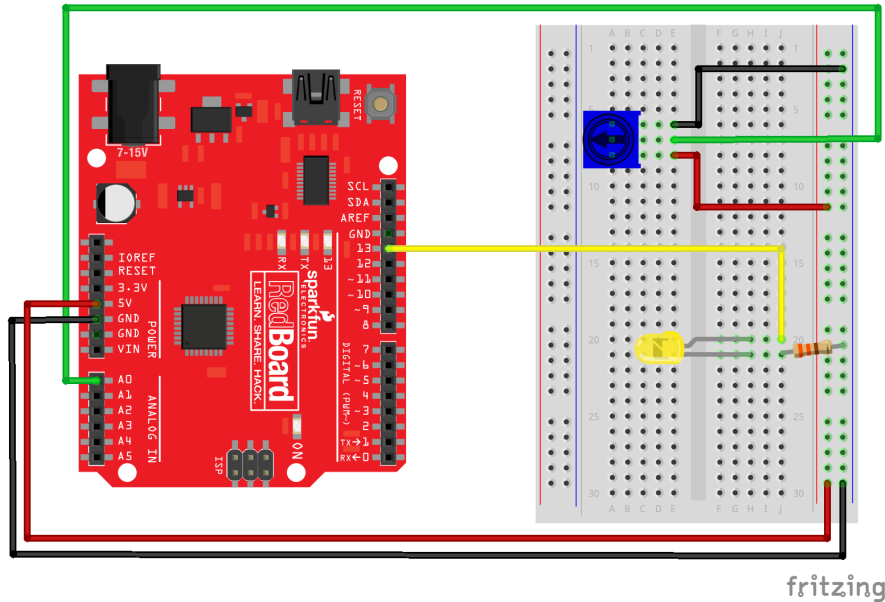
The hardware schematic below describes how to connect the Arduino, LED and potentiometer.

Upload code to the Arduino

Once the LED and potentiometer is hooked up to the Arduino, upload the following code to the Arduino using the Arduino IDE. Note Arduinos use a programming language that is not Python. This Arduino sketch (a sketch is what an Arduino program is called) does a couple things. First the Arduino reads the sensor value and stores the sensor value in a variable. Then the Arduino sends the sensor value over the serial line (as a byte string). Next the sensor value is compared to 500. If the sensor value is less than 500 the LED stays off. If the sensor value is greater than 500, the LED turns on. This process repeats in a loop.

```
// potentiometer_read.ino
// reads a potentiometer and sends value over serial
int sensorPin = A0;    // The potentiometer is connected to analog pin 0
int ledPin = 13;       // The LED is connected to digital pin 13
int sensorValue;       // an integer variable to store the potentiometer reading

void setup() // this function runs once when the sketch starts
{
    // make the LED pin (pin 13) an output pin
```



Arduino with potentiometer

```
pinMode(ledPin, OUTPUT);

// initialize serial communication:
Serial.begin(9600);
}

void loop() // this function runs repeatedly after setup() finishes
{
    sensorValue = analogRead(sensorPin); // read the voltage at pin A0
    Serial.println(sensorValue);          // Output sensor value to Serial Monitor

    if (sensorValue < 500) {               // if sensor output is less than 500,
        digitalWrite(ledPin, LOW); }       // Turn the LED off

    else {                                 // if sensor output is greater than 500
        digitalWrite(ledPin, HIGH); }      // Keep the LED on

    delay(100);                            // Pause 100 milliseconds before next reading
}
```

Connect the Arduino to the computer and Upload the Sketch

Connect the Arduino to the computer using a USB cable. Upload the sketch from the computer to the Arduino. In the Arduino IDE, click the check mark to verify and the arrow to upload. If the sketch does not upload, check which COM port is selected in Tools → Ports.



Arduino IDE Check to Verify



Arduino IDE Arrow to Upload

Check the Sensor Signal

To verify the Arduino sketch is working correctly, the sensor signal can be checked in three ways:

- * The LED will turn on and off as the potentiometer dial is turned
- * In the Arduino Serial Monitor, numbers change as the potentiometer dial is turned
- * In the Arduino Serial Plotter, the line moves as the potentiometer dial is turned.

LED turns ON and OFF

The LED should turn on and off as the potentiometer is turned. If the LED does not turn on and off when the potentiometer is turned, make sure the potentiometer is turned back and forth through its full range of rotation. Ensure the USB cable is plugged in to both the Arduino and the computer.

Arduino Serial Monitor

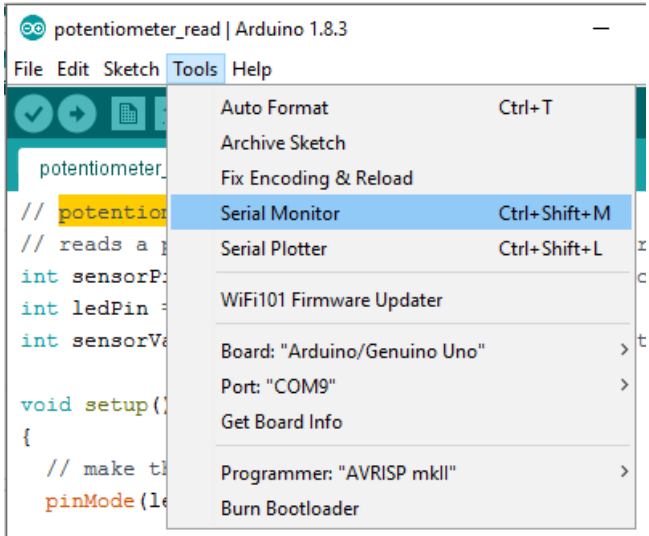
A running list of numbers should be seen in the Arduino Serial Monitor if the sketch is working. To access the Arduino Serial Monitor use Tools → Serial Monitor.

The output in the Serial Monitor should be a running list of numbers between zero and 1024. When the potentiometer is dialed back and forth the numbers streaming down the Serial Monitor should change. If output is not shown in the Serial Monitor, ensure both Auto Scroll, NL & CR, 9600 baud are selected. Also make sure the Port is set correctly in Tools → Port.

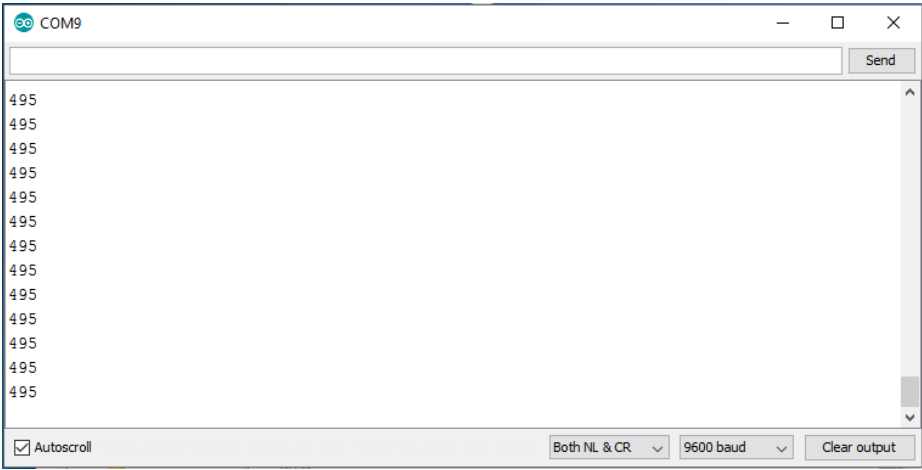
Arduino Serial Plotter

A moving line should be seen in the Arduino Serial Plotter if the sketch is working correctly. To access the Arduino Serial Plotter use Tools → Serial Monitor. Note the Serial Monitor needs to be closed before the Serial Plotter can be opened.

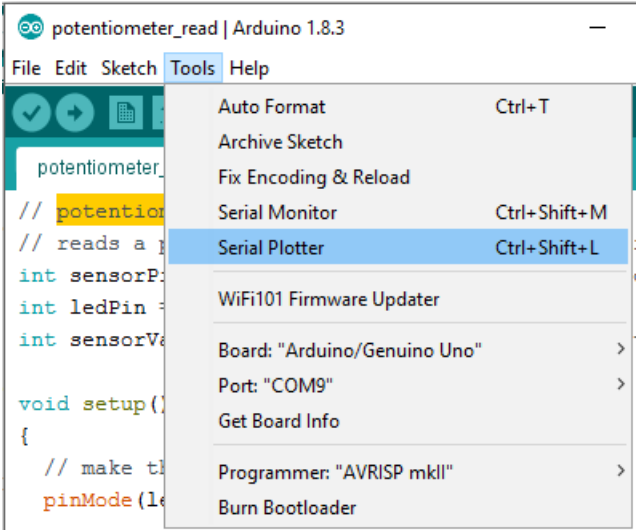
The output of the Serial Plotter should be a running line graph. The height of the line on the graph should change as the potentiometer is dialed back and forth. If the Serial Plotter is blank, make sure 9600 baud is selected in the lower right corner of the Serial Plotter. Also make sure the Port has been set correctly in Tools → Port.



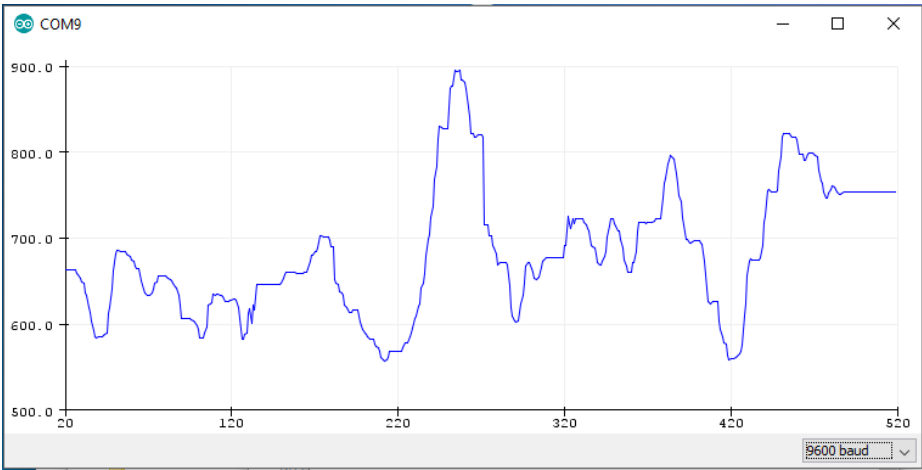
Arudion IDE Tools Serial Monitor



Arduino Serial Monitor Ouput



Arduino Serial Plotter Menu



Arudino Seral Plotter

Write a Python Script to read the sensor

Once the hardware is connected and the Arduino sketch is working correctly, a Python script can be constructed to read the sensor value. This can be accomplished using the **PySerial** package. Make sure **PySerial** is installed before the script is run. At the top of the script, import the **PySerial**. Note although the package is called **PySerial** to import the package use `import serial`.

```
import serial
import time
```

Next create a loop that will run for about 5 seconds while data is collected from the sensor. If it seems like the loop is stuck, press [Ctrl] + [c].

```
data = [] #create an empty list to store the data
for i in range(50):
    with serial.Serial('COM4', 9800, timeout=1) as ser:
        line = ser.readline() # read a '\n' terminated line
        string = line.decode() # convert byte string to unicode string
        data.append(float(string)) # convert unicode string to float and add to data
        time.sleep(0.1) # wait 0.1 seconds before reading the next line
```

Now after the data has been collected, it can be displayed with the `print()` function and a `for` loop. The output should look like the numbers in the Arduino Serial Monitor.

```
for line in data:
    print(line)
```

The data can also be plotted with **matplotlib**. The plot should look like the line plot in the Arduino Serial Plotter.

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot(data)
plt.show()
```

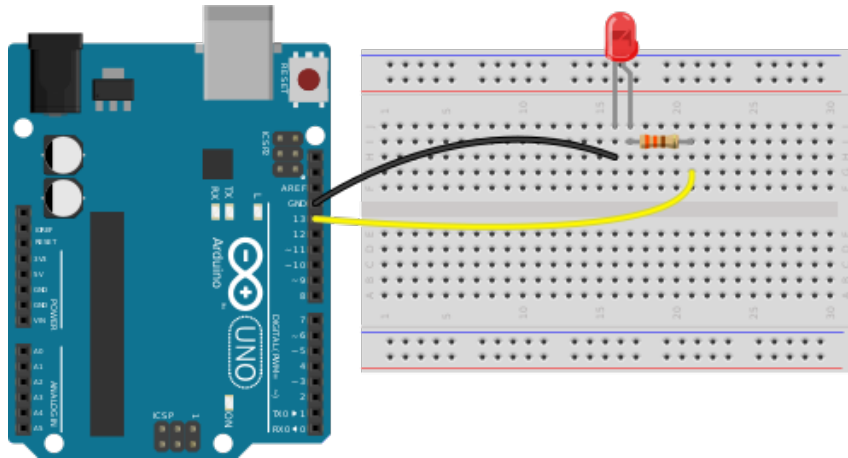
12.5 Controlling an LED with Python

In this section we are going to control an LED connected to an external piece of hardware (in this case an Arduino) using Python. To accomplish this the following hardware is required:

- A computer running Python
- An Arduino
- An LED
- Wires, a resistor and a breadboard to connect the LED to the Arduino
- A USB cable to connect the Arduino to the computer

Set up the LED

Connect the LED to the Arduino using a resistor, wires and a breadboard. Note the short leg of the resistor should be connected to ground and the long leg of the resistor should be connected to a



Arduino with LED

resistor and then PIN 13. A resistor is needed to prevent too much current from flowing through the LED. This type of resistor is called a *pull up resistor*.

Upload code to the Arduino

Upload the following code to the Arduino using the Arduino IDE. This is the example sketch called Physical Pixel. The sketch can be found in the Arduino IDE by going to File -> Examples -> 04.Communication -> PhysicalPixel

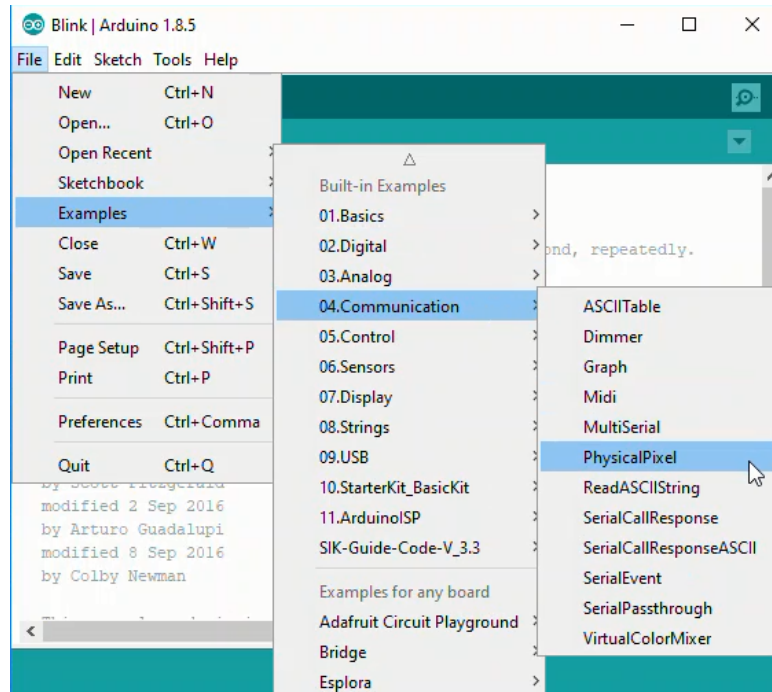
The code for the Physical Pixel Sketch is also shown below. Note that Arduinos use a programming language that is not Python.

```
// Found in the Arduino IDE: File --> Examples --> 04.Communication --> PhysicalPixel

const int ledPin = 13; // the pin that the LED is attached to
int incomingByte;      // a variable to read incoming serial data into

void setup() {
  // initialize serial communication:
  Serial.begin(9600);
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // see if there's incoming serial data:
  if (Serial.available() > 0) {
    // read the oldest byte in the serial buffer:
    incomingByte = Serial.read();
    // if it's a capital H (ASCII 72), turn on the LED:
    if (incomingByte == 'H') {
      digitalWrite(ledPin, HIGH);
    }
  }
}
```



Physical Pixel Example Sketch in the Arduino IDE



Arduino IDE Check to Verify

```

}
// if it's an L (ASCII 76) turn off the LED:
if (incomingByte == 'L') {
    digitalWrite(ledPin, LOW);
}
}
}

```

Connect the Arduino to the Computer

Connect the Arduino to the computer using a USB cable. Make sure a port is selected in the Arduino IDE: Tools → Port → COM4 (for example).

In the Arduino IDE, click the checkmark to verify and the arrow to upload. If the sketch does not upload, check which COM port is selected in Tools → Ports.



Arduino IDE Arrow to Upload

Turn the LED on and off with the Arduino Serial Monitor

Bring up the Arduino Serial Monitor and type L and H and see the LED turn on and off.

Write a Python Script to turn the LED on and off

After the LED is connected and the LED turns on and off based on input at the Arduino Serial Monitor, it is time to write a Python script to turn the LED on and off. This can be accomplished using the **PySerial** package. Make sure **PySerial** is installed before script is run.

At the top of the script, import the **PySerial**. Note that even though the package is called **PySerial** the line `import serial` is used.

```
import serial
import time
```

In the next part of the Python scripts, create a loop that will run for about 5 seconds while the LED is blinked on and off. Note the byte string H need to be sent to the Arduino, not the unicode string H. The unicode string is pre-pended with the letter b as in `ser.writelines(b'H')`.

```
data = [] #create an empty list to store the data
for i in range(10):
    with serial.Serial('COM4', 9800, timeout=1) as ser:
        ser.writelines(b'H') # send a byte
        time.sleep(0.5) # wait 0.5 seconds before reading the next line
        ser.writelines(b'L') # send a byte
```

Write a Python script to allow a user to turn the LED on and off

Once the LED blinks on and off, it is time to write a Python script to allow a user to turn the LED on and off. At the top of the Python script import the PySerial Module.

```
import serial
import time
```

Next give the user instructions. If the user types H, the LED will turn on. If the user types L the LED will turn off. If the user types q, the program will quit.

```
print('This is a program that allows a user to turn an LED on and off')
print('H to turn the LED on')
print('L to turn the LED off')
print('q to quit')
```

Finally, the script needs a loop to ask the user for an H or L character. Once the user enters the letter, the letter needs to be converted to a byte string. The byte string can then be sent over the serial line to the Arduino. A delay is added so that the Arduino can process the previous command before dealing with the next one.

```
user_input = input('H = on, L = off, q = quit' : )
while user_input != 'q':
    with serial.Serial('COM4', 9800, timeout=1) as ser:
        byte_command = encode(user_input)
        ser.writelines(byte_command)    # send a byte
        time.sleep(0.1) # wait 0.5 seconds before reading the next line

print('q entered. Exiting the program')
```

12.6 Summary

Key Terms and Concepts

External Hardware

Sensor

LED

Resistor

breadboard

Arduino

Serial Communication

USB

Unicode

Unicode String

UTF-8

Byte code

ASCII

ASCII Character

12.7 Review Questions

- 1.
- 2.
- 3.
- 4.

- 5.
- 6.
- 7.
- 8.
- 9.
- 10.

Chapter 13

MicroPython

13.1 Introduction

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

- Install MicroPython on a microcontroller
- Run Python commands on a microcontroller using the MicroPython REPL
- Save module files and run Python scripts on a microcontroller
- Use MicroPython to read sensor data using a microcontroller
- Use MicroPython to activate a relay using a microcontroller

13.2 What is Micropython?

What is Micropython?

[Micropython](#) is a port, or version of Python designed to run on small, inexpensive, low-power microcontrollers. Examples of microcontrollers that Micropython can run on include the [pyboard](#), the [WiPy](#) and ESP8266 boards like the [Adafruit Feather Huzzah](#). Normally, Python is run on a desktop or laptop computer (also on big servers at server farms). Compared to a desktop or laptop, microcontrollers are much smaller, cheaper and less powerful. A “regular” version of Python can’t run on small, cheap microcontrollers because Python is too resource intensive. Regular Python takes up too much hard disk space, runs on too much RAM and requires a more powerful processor than microcontrollers have.

It is pretty amazing that a version of Python (Micropython) runs on these small, cheap microcontrollers like the ESP8266. To get Micropython to run at all on these small boards, Micropython only contains a subset of all the standard library modules included with “regular” Python. Some of the libraries that are included with Micropython don’t have the full set of functions and classes that come with the full version of Python. This allows Micropython to be compact (around 600 kB for

the ESP8266 port) and only use a small amount of RAM (down to 16k according to the [Micropython main page](#))

You can try using Micropython online with this neat [Micropython online emulator](#). The emulator allows you to run commands at a Micropyton Prompt and see the result on a virtual pyboard.

What is Micropyton used for?

Micropython is installed on small, cheap microcontrollers like the [ESP8266](#). Anything these small microcontrollers can do, Micropython can do. This includes using the microcontroller as a remote sensor to measure things like temperature, humidity and light level. Micropython can also be used to blink LED's, control arrays of LED's, or run small displays. Micropython can control servo motors, stepper motors and solenoids. Civil Engineers could use Micropython to monitor water levels. Mechanical Engineers could use Micropython to drive robots. Electrical Engineers could use micropython to measure voltage levels in embedded systems. In the later posts in this series, we will use Micropython, running on a small cheap ESP8266 board, to create a remote internet-connected weather station. The last posts in the series will use Micropyton, running on a **really cheap** (around \$2) ESP-01 module to turn on and off an LED from any computer connected to the internet anywhere in the world.

Why should undergraduate Engineers learn Mircopython?

Using Python to solve engineering problems such as calculations, statistics, modeling and visualization is really useful for undergraduate Engineers. But Python on it's own is fairly limited in controlling devices outside the computer it's running on. You don't want to leave a laptop in a remote estuary to measure water temperature, but you could leave a little microcontroller and low-cost temperature sensor. A small robot can't carry around a heavy laptop, but a small, light, low-power board could run a simple robot. You don't want to use a laptop for every small electrical measurement or embedded system control, but a \$2 WiFi module would work.

In addition, learning how to use Micropython on small, cheap microcontrller can help undergraduates Engineers understand how programming works. It is a different kind of feedback and excitement seeing a motor whirl around compared to seeing a picture of a motor with the speed displayed as text. There is a different kind of wonder seeing an array of LED's light up compared to a 2-D plot on a computer screen. Plus Micropython is just fun! It's as easy to program Micropython as it is to program regular Python. The little projects you can do with Micropython running on a small, low-cost board are almost unlimited. We could send Micropython to space in a micro-satellite, or bury Micropython underground in a small boring machine, or launch Micorpython into the sky on a weather balloon.

13.3 Installing Micropython

Micropython is a port of the Python programming language that runs on small, inexpensive microcontrollers. In this section, we will install Micropython on an Adafruit Feather Huzzah ESP8266 board using Python and a package called **esptool**. In subsequent sections we will build our Feather Huzzah microcontroller into a WiFi-enabled weather station.

To install Micropython on a microcontroller, like the Adafruit Feather Huzzah ESP8266, we need

the following hardware:

| Hardware | Purpose |
|---------------------------------|--|
| Windows 10 | install Micropython on the microcontroller |
| Adafruit Feather Huzzah ESP8266 | microcontroller running Microphythonn |
| microUSB cable | connect microcontroller to computer |

| Hardware | Purpose |
|---------------------------------|--|
| Windows 10 laptop | install Micropython on the microcontroller |
| Adafruit Feather Huzzah ESP8266 | microcontroller running Microphython |
| microUSB cable | connect microcontroller to computer |

To install Micropython we will use the following software and tools:

| Software | Purpose |
|---------------------------------|--|
| Windows 10 | download Micropython |
| Anaconda distribution of Python | run esptool that installs Micropython |
| Anaconda Prompt | Install the esptool package using pip |
| esptool | A pip installable package used to install Micropython |
| firmware .bin file | Version of Micropython run on the microcontroller |

Summary of Steps:

1. Install the [Anaconda distribution](#) of Python
2. Create a new conda environment and `pip install esptool`
3. Download the [latest Micropython .bin firmware file](#)
4. Install the [SiLabs driver](#) for the Adafruit Feather Huzzah ESP8266
5. Connect the Adafruit Feather Huzzah ESP8266 board to the laptop using a microUSB cable
6. Determine which serial port the Feather Huzzah is connected to
7. Run the esptool to upload the .bin firmware file to the Feather Huzzah
8. Download and install [Putty](#), a serial monitor
9. Use Putty to connect to the Feather Huzzah and run commands in the Micropython REPL

Install the Anaconda distribution of Python

If you don't have Anaconda installed already, go to [Anaconda.com/download](https://anaconda.com/download) and install the latest version. The Anaconda distribution of Python is the Python distribution I recommend for undergraduate engineers. You want to download and install the Python 3.6 Version (the Python 2.7 Version is legacy Python). Most laptops and desktops run a 64-bit version of Windows 10. If in doubt, you can check your Windows installation, or just go with the 64-bit version.

Create a new conda environment and install esptool.py

It's best practice when using Python to work in virtual environments. We'll create a new virtual environment with conda to use with our Micropython projects. Open the Anaconda prompt and create a new virtual environment named `micropython`. Activate the environment with the `conda activate` command. After activating the virtual environment you should see `(micropython)` before the Anaconda Prompt. Once inside the virtual environment, use `pip` to install `esptool`. The `esptool` will be used to upload the Micropython `.bin` firmware file onto the Adafruit Feather Huzzah board. Confirm that `esptool` is installed in the `(micropython)` virtual environment with `conda list`. I also created a new directory in the **Documents** folder called **micropython** to store all the project files.

```
> conda create -n micropython python=3.6
> conda activate micropython
(micropython) > pip install esptool
(micropython) > conda list
(micropython) > cd Documents
(micropython) > mkdir micropthon
(micropython) > cd micropython
```

Download the latest micropython firmware .bin file

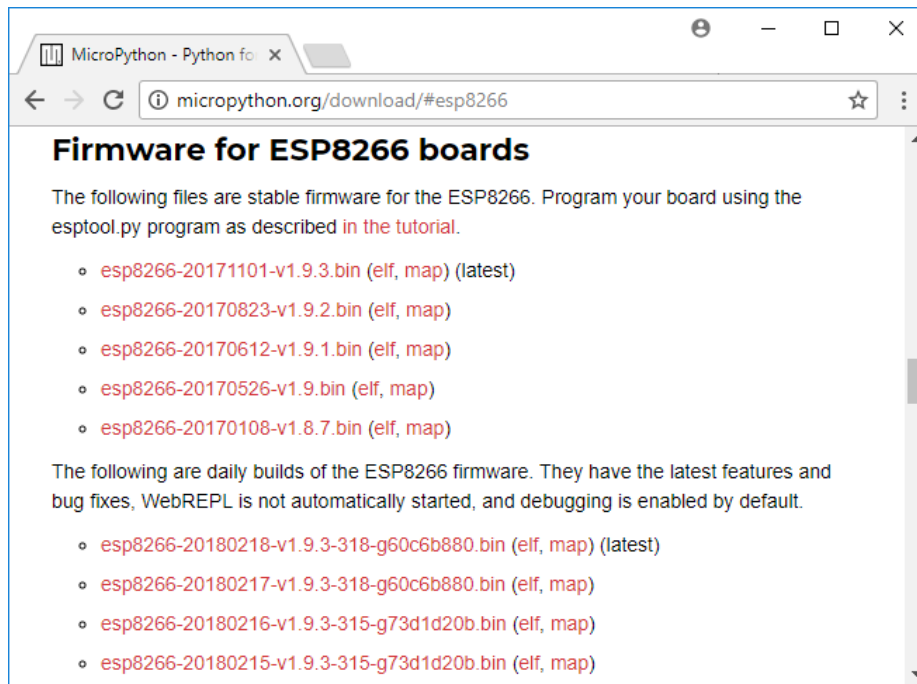
Go to [github.com](https://github.com/micropython/micropython.org/download#esp8266) and [download the latest .bin firmware](#) file at `micropython.org/download#esp8266`. Move the `.bin` firmware file to a new **micropython** directory. The `.bin` firmware file is the version of Micropython that will run on the Adafruit Feather Huzzah ESP8266. Straight from Adafruit, the Feather Huzzah microcontroller does not have Micropython installed, so we need to install Micropython ourselves. After installing the Micropython `.bin` firmware file onto the board, we will be able to bring up the Micropython REPL prompt, type commands into the Micropython REPL and run Micropython `.py` scripts on the board.

Install the SiLabs driver for the Adafruit Feather Huzzah ESP8266

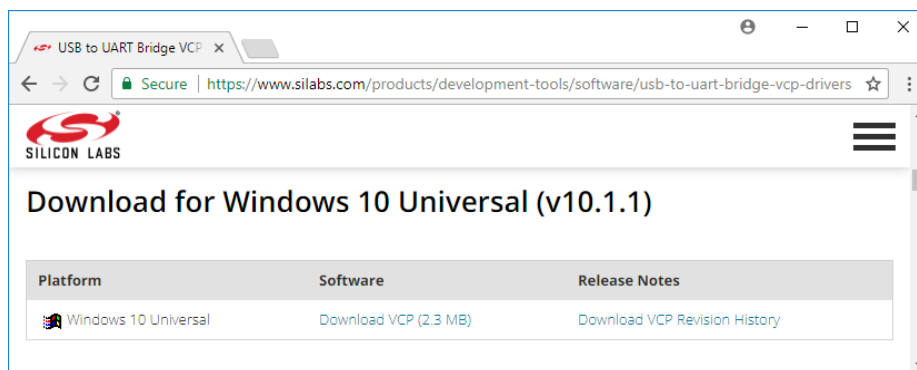
Before we can connect the Adafruit Feather Huzzah to the computer, we need a specific driver installed. For my Windows 10 laptop to see the Adafruit Feather Huzzah board, the [CP210x USB to UART Bridge VCP driver](#) needs to be downloaded from SiLabs and installed. This is quick and easy, but does require admin privileges.

Connect the Adafruit Feather Huzzah ESP8266 board to the laptop

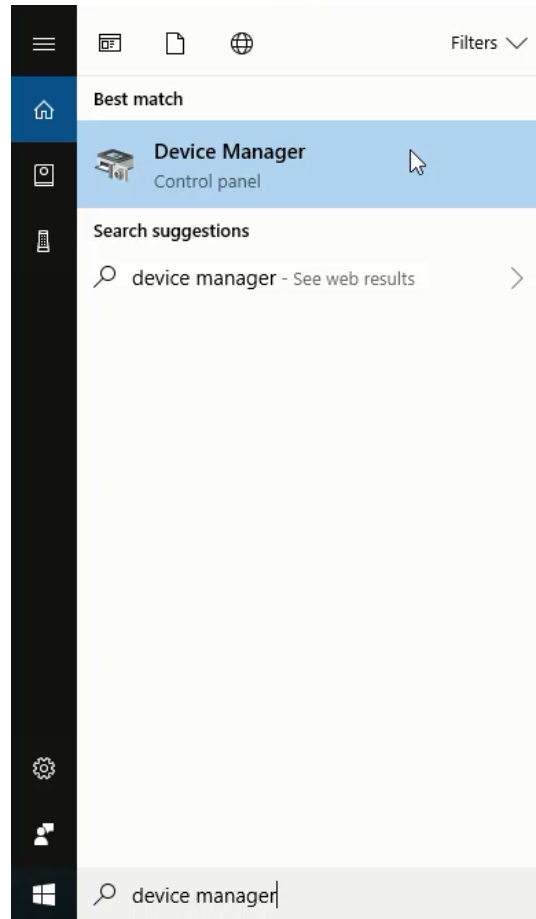
Use a microUSB cable (the same kind of cable that charges many mobile phones) to connect the Feather Huzzah to the computer. Make sure that the microUSB cable is a full USB **data cable** and not just a simple power cable. I had trouble getting the Feather Huzzah to work, and it turned out the reason was the microUSB cable was only a charging cable and could not transfer data.



.bin firmware on github



SiLabs Driver

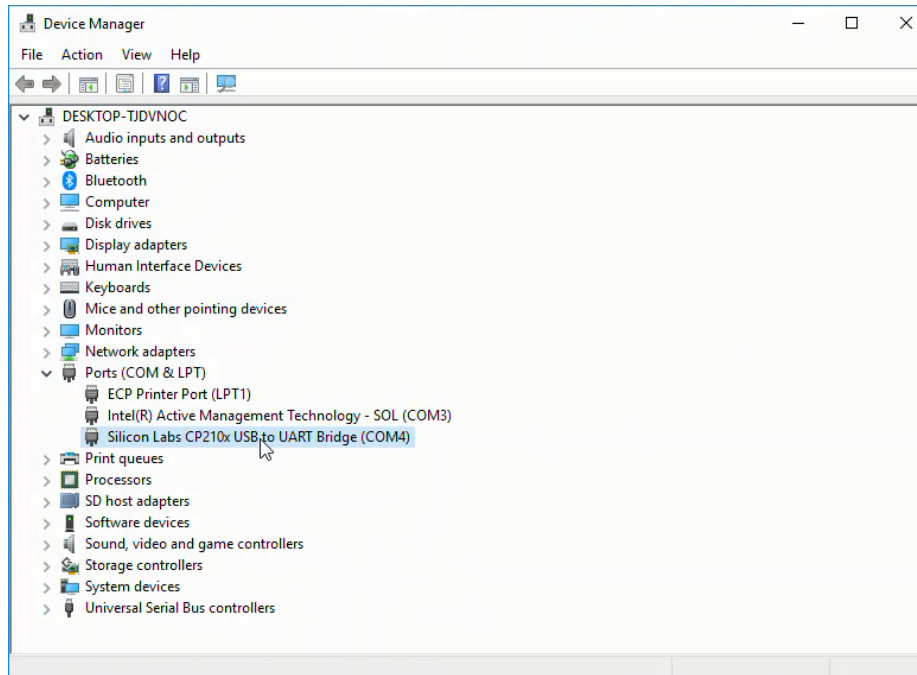


Find Device Manager

Determine which serial port the Feather Huzzah is connected to

Use Windows Device Manager to determine which serial port the Feather Huzzah board is connected to. We will need the serial port as one of the parameters when we upload the .bin firmware file on the board. Look for something like **Silicon Labs CP210x USB to UART Bridge (COM4)** in the **Ports (COM & LPT)** menu. The USB to UART bridge is actually the Feather Huzzah board. CP210x refers to the chip that handles serial communication on the Feather Huzzah, not the esp8266 chip itself. Make note of the number after **(COM)**. It often comes up as **(COM4)** but it may be different on your computer.

The first time I plugged the board into my laptop, Windows couldn't see the board. I looked through the Device Manager under the Ports menu and the Feather board just didn't show up. Turns out the first USB cable I used was just a charging only cable. When I switched this out for a microUSB data cable, the board came right up under **Ports (COM & LPT)**.



Device Manager Menu

Run esptool to upload the .bin file to the Feather Huzzah

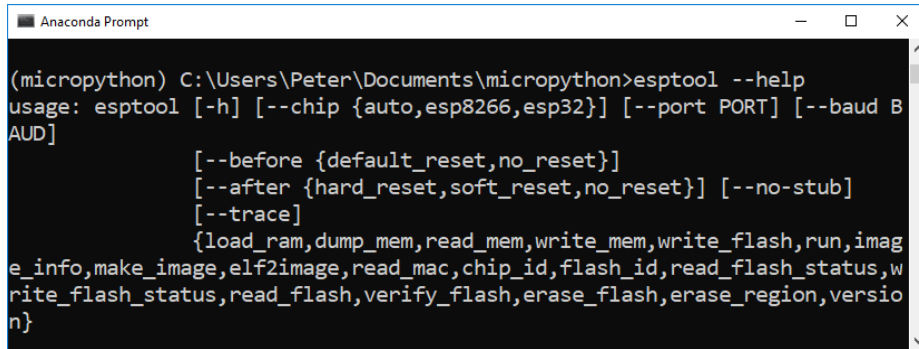
Open the Anaconda Prompt and `cd` into the **micropython** directory with the .bin file. You can use the `dir` command to see the directory contents. Make sure the .bin firmware file is in the directory. It will be called something like `esp8266-20171101-v1.9.3.bin`. Activate the micropython environment with `conda activate micropython`. Run `esptool --help` to ensure esptool is installed properly. Note there is no .py extension after `esptool`. On my Windows laptop, the command `esptool` worked, but the command `esptool.py` did not (this is different than the commands shown on the [Micropython docs](#)). If you try to run `esptool` and you are not in the (micropython) virtual environment, you will get an error.

```
> cd Documents
> cd micropython
> pwd
Documents/micropython
> dir
> conda activate micropython
(micropython) > esptool --help
```

Before we write the .bin firmware file to the board, we should first erase the flash memory on the Feather Huzzah using the `esptool erase_flash` command. Make sure to specify the `--port`. This is the COM port you found in the Windows Device Manager. In my case the port was COM4.

```
(micropython) esptool --port COM4 erase_flash
```

Now it's time to write the .bin firmware file to the flash memory on the board using the `esptool write_flash` command. Make sure to use the exact .bin firmware file name you see sitting

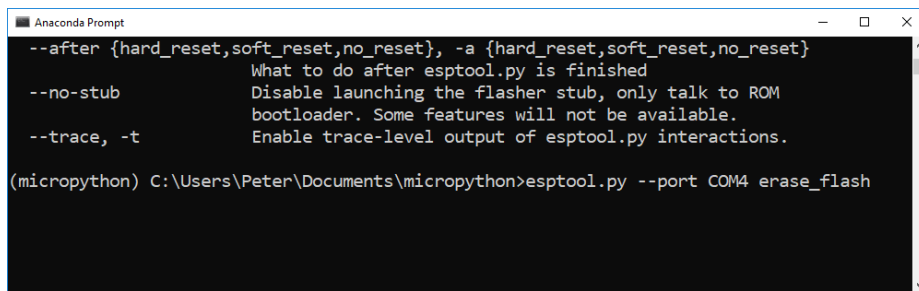


```

(micropython) C:\Users\Peter\Documents\micropython>esptool --help
usage: esptool [-h] [--chip {auto,esp8266,esp32}] [--port PORT] [--baud B
AUD]
               [--before {default_reset,no_reset}]
               [--after {hard_reset,soft_reset,no_reset}] [--no-stub]
               [--trace]
               {load_ram,dump_mem,read_mem,write_mem,write_flash,run,imag
e_info,make_image,elf2image,read_mac,chip_id,flash_id,read_flash_status,w
rite_flash_status,read_flash,verify_flash,erase_flash,erase_region,versio
n}

```

esptool help



```

--after {hard_reset,soft_reset,no_reset}, -a {hard_reset,soft_reset,no_reset}
What to do after esptool.py is finished
--no-stub      Disable launching the flasher stub, only talk to ROM
bootloader. Some features will not be available.
--trace, -t    Enable trace-level output of esptool.py interactions.

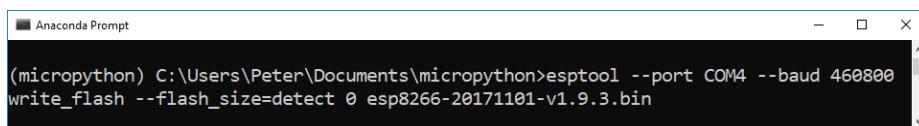
(micropython) C:\Users\Peter\Documents\micropython>esptool.py --port COM4 erase_flash

```

esptool erase flash

in the **micropython** directory. The port has to be set as the port you found in the Windows Device Manager. `---baud` is the baud rate, or upload speed. I found that `--baud 460800` worked, but you could also specify `--baud 115200` which is slower. The upload time was a matter of seconds with either baud rate. The 0 after `--flash_size=detect` means we want the firmware to be written at the start of the flash memory (the 0th position) on the board. Again, make sure the .bin firmware file name is correct. It is easy to mistype. Another issue I ran into was that I tried to use the command `esptool.py` instead of `esptool` as shown on the [Micropython docs](#). The documentation for [Micropython on the ESP8266](#) specifies the command `esptool.py` (including the .py file extension). This did work on my Windows 10 machine. Omitting the .py file extension, and running `esptool` worked instead.

```
(micropython) > esptool --port COM4 --baud 460800 write_flash --flash_size=detect
```

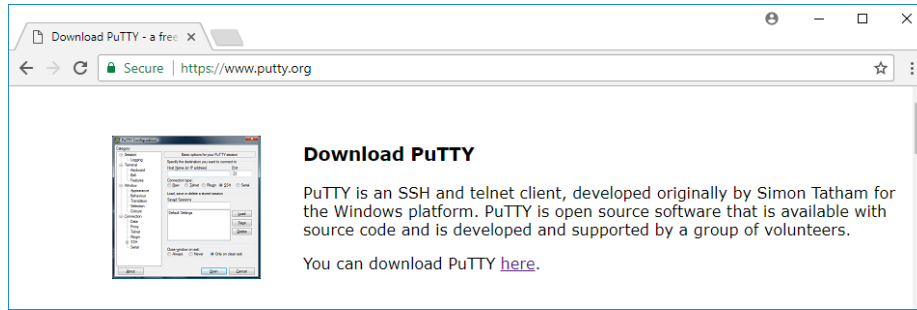


```

(micropython) C:\Users\Peter\Documents\micropython>esptool --port COM4 --baud 460800
write_flash --flash_size=detect 0 esp8266-20171101-v1.9.3.bin

```

esptool write flash



Download Putty

Download and install Putty, a serial monitor

Now that MicroPython is installed on the board, we need to talk to our board over a serial connection. Windows 10 doesn't have a built-in serial monitor (like screen on OSX and Linux). So we need to download and install **PuTTY**. PuTTY is a lightweight SSH and serial client for Windows. PuTTY will allow us to communicate with the Adafruit Feather Huzzah board. [PuTTY can be downloaded here](https://www.putty.org). PuTTY is pretty small and the download and install should be pretty quick.

Use Putty to connect to the Feather Huzzah

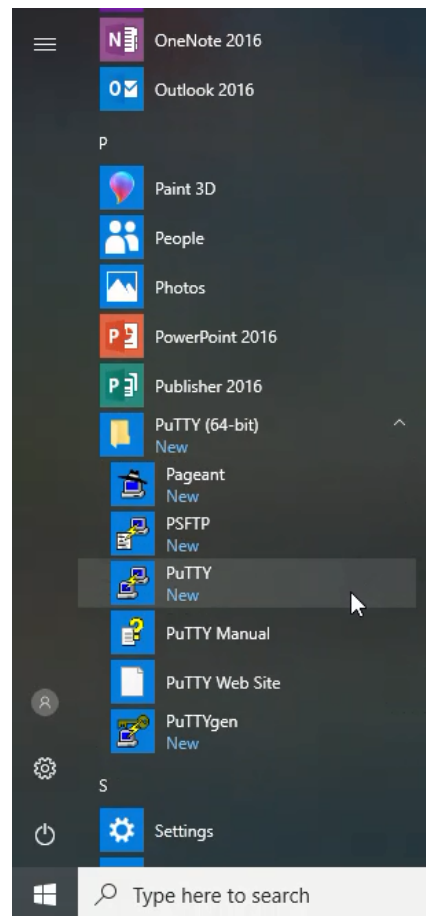
Ensure the Feather board is connected to the computer with a USB cable and ensure you can see the board in the Windows Device Manager. Then use PuTTY to connect to the board over serial. Make sure you specify the correct serial port in the **Serial line** box and **115200** baud in the Speed box. **MicroPython is set to run at 115200 baud**, other baud rates will lead to junk characters in the serial monitor. I had trouble finding the serial connection option in PuTTY. When I opened PuTTY, the default was an SSH connection. We can't connect to the Feather Huzzah over SSH. You need to select the **Serial** radio button below the header **Connection type**: near the top of the PuTTY window.

If you see `>>>` the MicroPython REPL (the MicroPython prompt) is running and the Adafruit Feather Huzzah ESP8266 is working! This version of Python isn't running on your computer, it's MicroPython running on the little microcontroller! Sometimes I had to type [Enter] or Ctrl-D to get the `>>>` REPL prompt to show up. A few times I needed to close PuTTY, unplug then replug the board and try PuTTY again. The Feather Huzzah also has a tiny little black RESET button that can be pressed to restart the board.

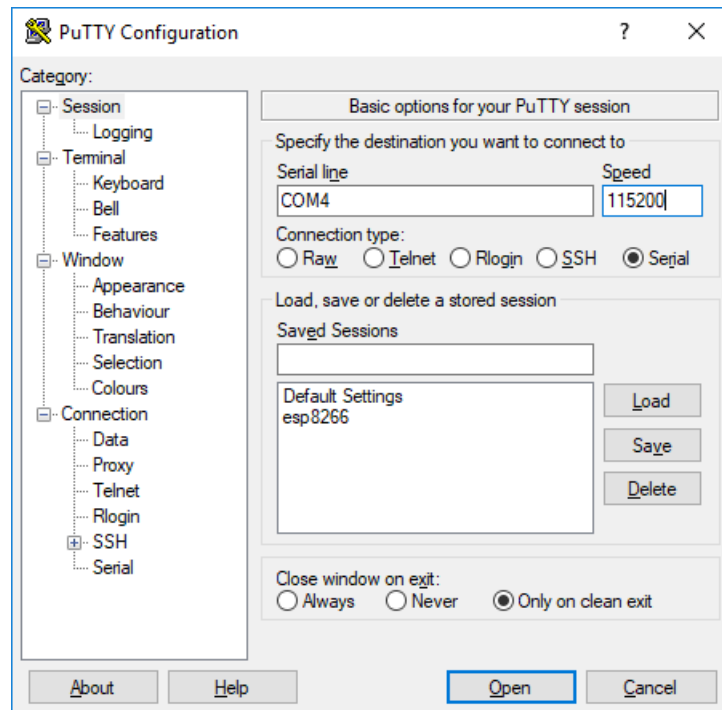
At the `>>>` MicroPython REPL prompt try the following commands:

```
>>> print('Micropython for Engineers!')
Micropython for Engineers

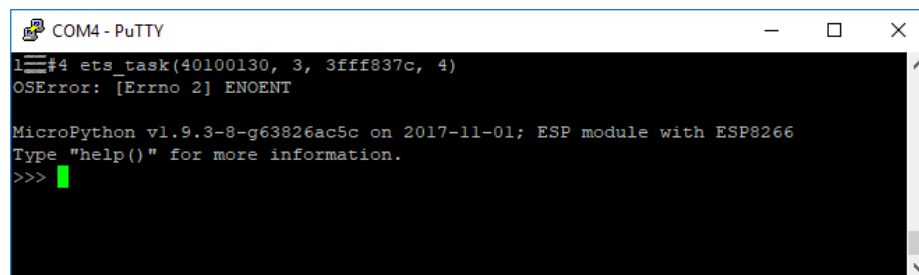
>>> import sys
>>> sys.platform
'esp8266'
```



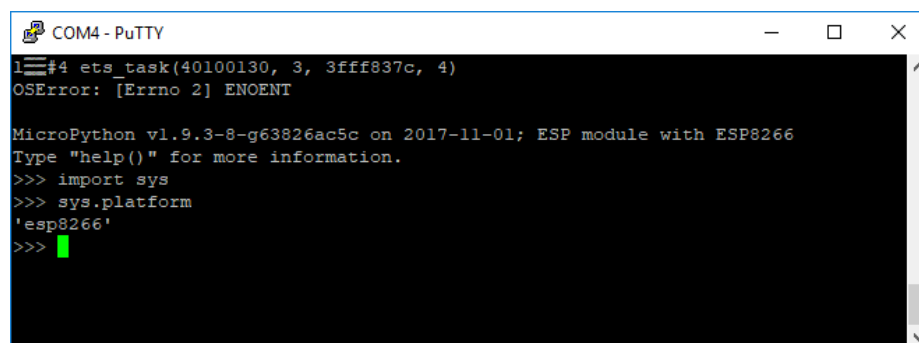
Putty in start menu



Putty config



REPL Prompt



sys_dot_platform

13.4 Micropython REPL

The last section detailed the installation of Micropython on an Adafruit Feather Huzzah ESP8266 microcontroller using Python and a package called **esptool**. In this section, we are going to write commands to the Micropython REPL (the Micropython prompt) to turn on and off an LED connected to the Feather Huzzah board. The posts in this series:

Before you can use the Micropython REPL (the Micropython prompt) running on the Adafruit Feather Huzzah ESP8266, Micropython needs to be installed on the board and Putty needs to be installed to communicate with the board over serial. See the previous section for information on how to install Micropython on the board and install [PuTTY](#) on a Windows 10 machine.

Summary of Steps

1. Connect the Adafruit Feather Huzzah ESP8266 using a USB cable
2. Determine which COM port the board is connected to using the Windows Device Manager
3. Open Putty and connect to the board at 115200 baud
4. Run commands at the prompt to turn the built-in LED on the Adafruit Feather Huzzah ESP8266 on and off

Connect the Adafruit Feather Huzzah ESP8266 board to the laptop

Use a microUSB cable to connect the Feather Huzzah to the computer. Make sure that the microUSB cable is a full USB data cable and not just a simple power cable. Cables that are just used to charge phones may only be power cables and may not be capable of transmitting data.

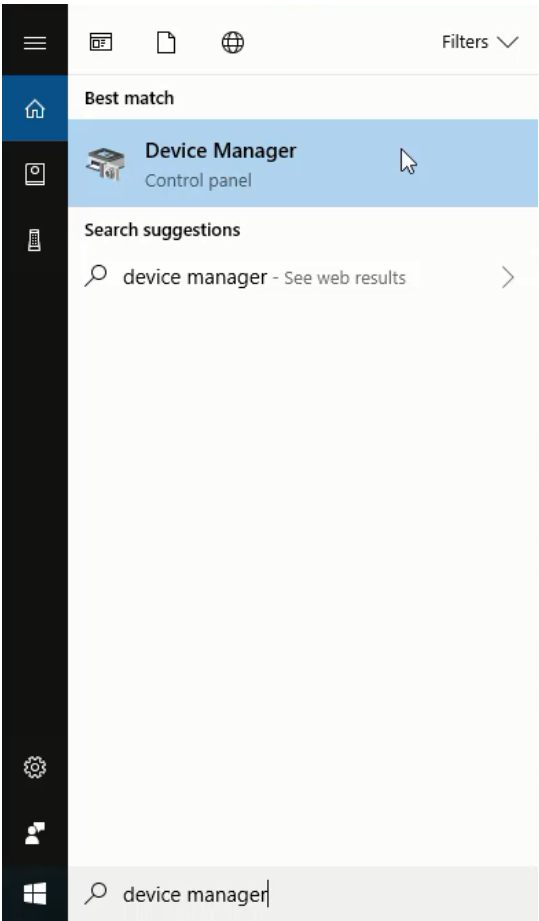
Determine which serial port the Feather Huzzah is connected to

Use Windows Device Manager to determine which serial port the Feather Huzzah is connected to. On my Windows 10 laptop, it usually comes up as COM4. You can find the serial port by looking in the Ports (COM & LPT) category of the Windows Device Manager. Look for something like **Silicon Labs CP210x USB to UART Bridge (COM4)** in the **Ports (COM & LPT)** menu. It is the **COM#** that you are looking for.

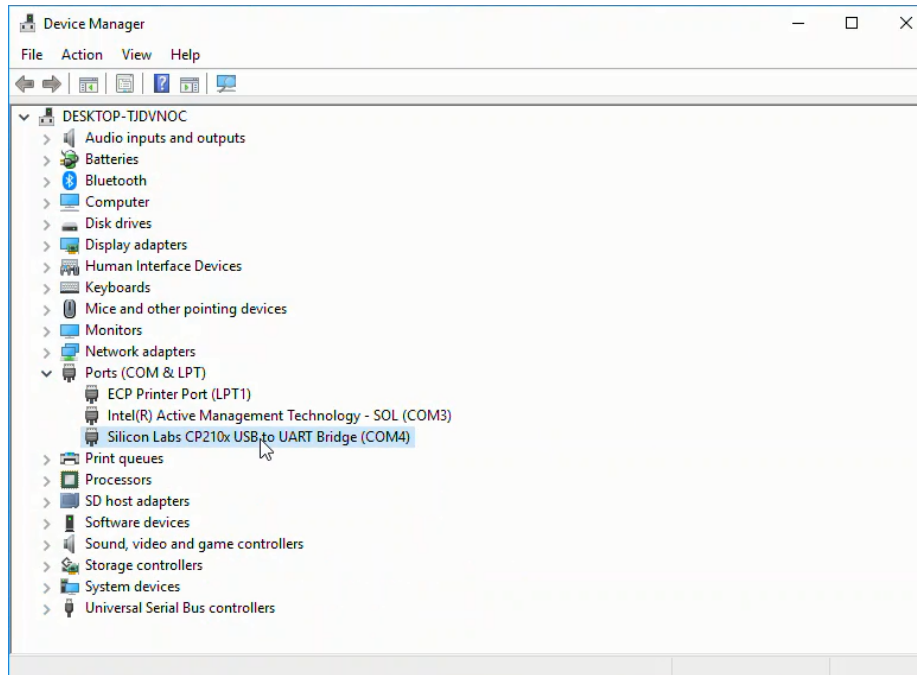
Use Putty to connect to the Feather Huzzah

Ensure the Feather Huzzah board is connected with a USB cable, then connect to it with Putty using the proper serial port (COM#) and 115200 baud. Remember to use the **Serial** radio button under **Connection Type**: to select serial communication or you will be trying to communicate with the Feather Huzzah over SSH which won't work.

This should bring up the Micropython REPL prompt `>>>`. If you can't see the `>>>` prompt, try typing [Enter], Ctrl-D, pushing the RESET button on the Feather Huzzah or unplugging then replugging the USB cable.



Find Device Manager



Device Manager Menu

Run commands at the prompt to turn the built-in LED on the Adafruit Feather Huzzah ESP8266 on and off

At the micropython REPL (the Micropython command prompt >>>) try the following commands:

```
>>> print('Micropython for Engineers!')
Micropython for Engineers
```

If we import the sys module, we can see the Micropython implementation and platform.

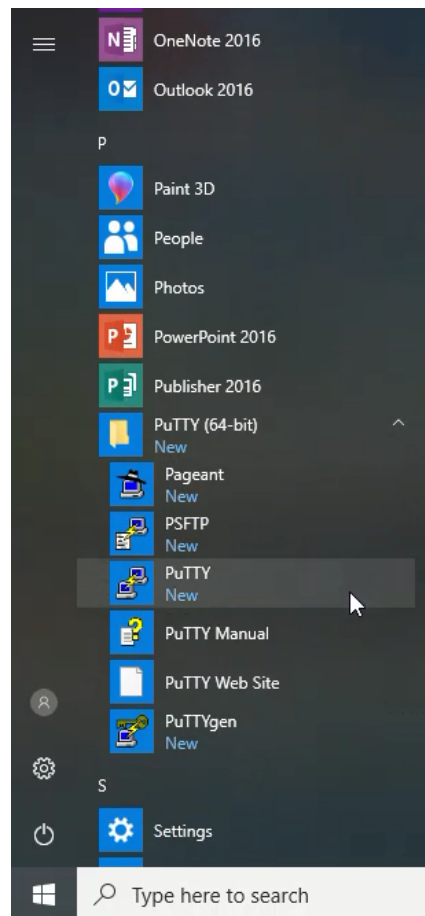
```
>>> import sys
>>> sys.implementation
(name='micropython', version=(1, 9, 3))
>>> sys.platform
'esp8266'
```

If you see similar output, that means Micropython is working on the Feather Huzzah. We can also view the flash memory size of our Feather Huzzah and the size of the Micropython firmware we installed. Try this at the Micropython prompt:

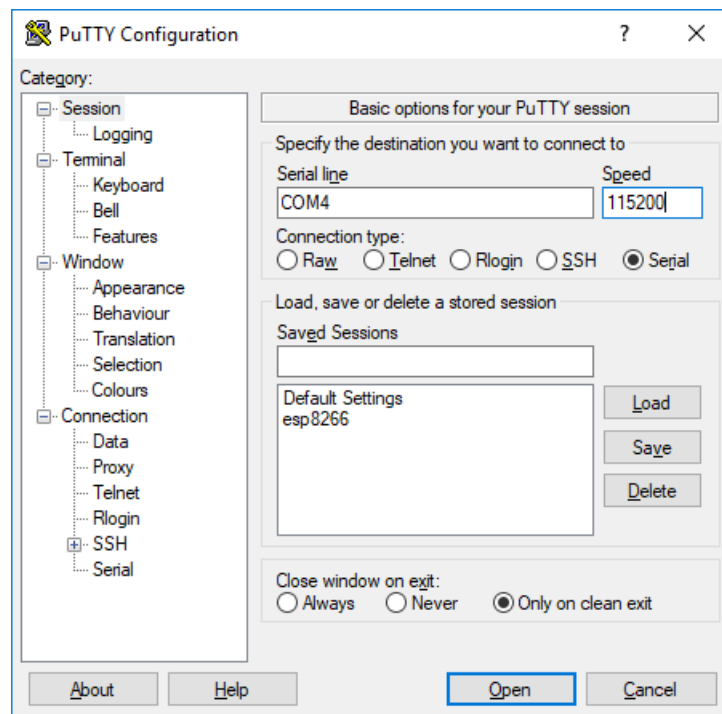
```
>>> import port_diag
```

We can see the flash memory size is 4 MB. Below the label `Firmware checksum:` we can see a line for `size: 600872`. This means the size of our Micropython installation is about 600 KB or 0.6 MB. Just over half a megabyte and we are running a working version of Python!

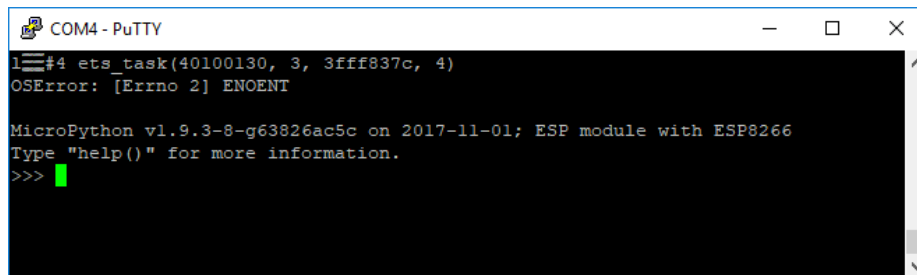
Now let's turn the Feather Huzzah's built-in LED on and off. The Feather Huzzah has a built-in red LED connected to Pin 0. We can access this LED with Micropython's machine module. First



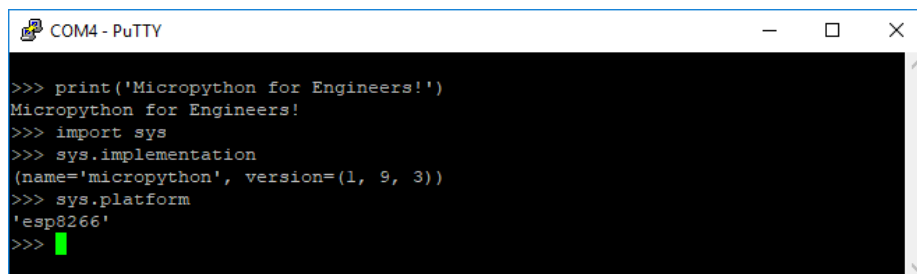
Putty in start menu



Putty config



REPL prompt



REPL prompt

```

COM8 - PuTTY
>>> import port_diag
FlashROM:
Flash ID: 1640ef (Vendor: ef Device: 4016)
Flash bootloader data:
Byte @2: 00
Byte @3: 40 (Flash size: 4MB Flash freq: 40MHZ)
Firmware checksum:
size: 600872
md5: 35e285a80516e70242ebf7d780d6c70f
True

Networking:
STA ifconfig: ('10.0.0.23', '255.255.255.0', '10.0.0.1', '75.75.75.75')
AP ifconfig: ('192.168.4.1', '255.255.255.0', '192.168.4.1', '75.75.75.75')
Free WiFi driver buffers of type:
0: 8 (1,2 TX)
1: 0 (4 Mngmt TX (len: 0x41-0x100))
2: 8 (5 Mngmt TX (len: 0-0x40))
3: 4 (7)
4: 7 (8 RX)
lwIP PCBs:
Active PCB states:
Listen PCB states:
Local port 8266, foreign port 9530 snd_nxt 0 rcv_nxt 1073716632 State: LISTEN
TIME-WAIT PCB states:
>>>

```

REPL prompt

we use the machine module to create a Pin object. The first argument when we instantiate the Pin object is the pin number on the board (in this case 0). Pin zero on the Feather Huzzah is connected to the built-in red LED. The second argument is the pin type. We want Pin 0 to act as an output pin (machine.Pin.OUT). We are going to assign our pin the attribute .on() or .off(). This will cause the Feather board to output a positive voltage or no voltage to Pin 0 to turn the built-in red LED on and off. You can also connect Pin 0 to an LED through a resistor (then to ground) and have this LED turn on and off.

```

>>> import machine
>>> pin = machine.Pin(0, machine.Pin.OUT)

```

Note that Pin 0 on the Adafruit Feather Huzzah is kind of wired “backwards”. We call pin.off() and the built-in LED turns **on** and call pin.on() and the built-in LED turns **off**.

```

>>> pin.on()
>>> pin.off()
>>> pin.on()
>>> pin.off()

```

Now let’s see if we can make the LED blink. We’ll do this with a simple for loop. At the micropython REPL, initiating a loop will automatically indent the next line, so a tab is not needed before the pin.on() statement. To run the loop, we type backspace on an empty line (to backspace from an indented line) and hit return.

```

>>> import time
>>> for i in range(10):
...     pin.on()
...     time.sleep(1)

```

```
...     pin.off()
...     time.sleep(1)
...
```

This will blink the LED on and off for a total of 20 seconds.

13.5 Blinking a LED

This is the four section of a chapter on Micropython. In this section, you will learn how to blink the built in LED on to an Adafruit Feather Huzzah ESP8266 using the Micropython REPL.

Before the LED on the Adafruit Feather Huzzah ESP8266 can be blinked, Micropython needs to be installed on the Feather Huzzah board and Putty needs to be installed (if using Windows 10) to facilitate communication between the Feather Huzzah board and the computer. Alternatively, the MacOS or Linux terminal can be used for serial communication.

The Feather Huzzah as a built-in red LED mounted on the board close to the USB cable input. Micropython can be used to blink this LED on and off.

Connect the Adafruit Feather Huzzah to the computer with a USB cable and bring up the Micropython REPL using Putty.

Connect the Adafruit Feather Huzzah ESP8266 to the computer with a microUSB cable. Ensure this is a USB data cable, not just a charging cable. Open Putty and connect to the Feather Huzzah using the proper serial port (COM#) and 115200 baud. (Remember to use the **Serial** radio button under **Connection Type**.)

This will bring up the Micropython REPL prompt >>>. If you can't see the >>> prompt, try typing [Enter] or Ctrl-D or push the RESET button on the Feather Huzzah. If none of these methods work, try closing Putty and unplugging then replugging in the USB cable.

Run code at the Micropython REPL to turn the LED on and off

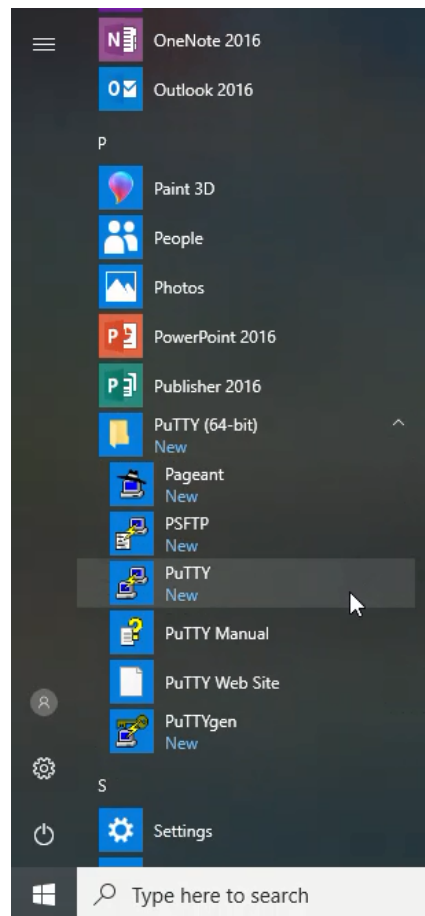
In the Putty Serial Window, test to see if the Micropython REPL is functioning with a basic *Hello World* program.

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

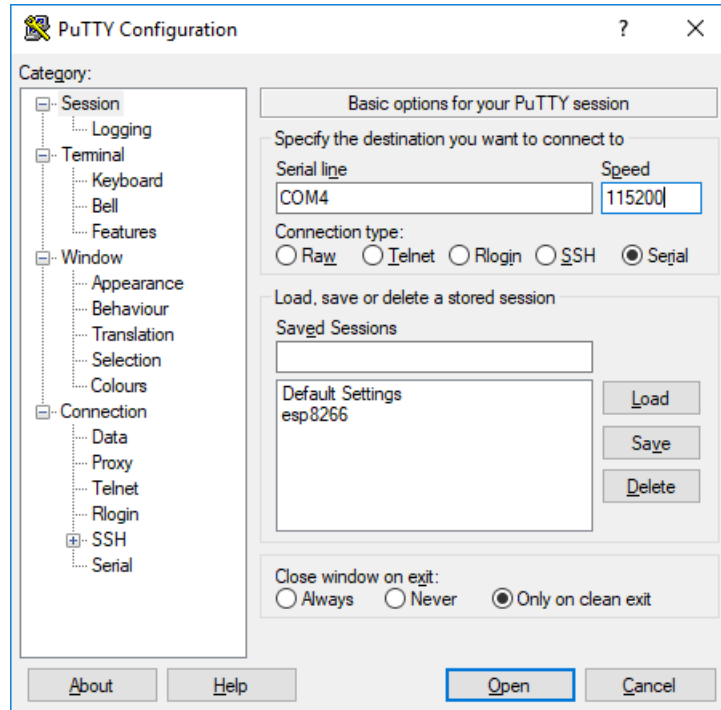
Next, we will blink the Feather Huzzah's builtin LED. The Feather Huzzah has a built-in LED connected to Pin 0. If we control the current going to Pin 0, we control the built-in LED. To control a Pin using Micropython, first the **machine** module needs to be imported. Next a Pin object needs to be created. The integer passed into `machine.Pin()` determines the pin number assigned to the Pin object.

```
>>> import machine
>>> pin = machine.Pin(0)
```

The value (on or off) of Pin 0 can be returned using



Putty in start menu



Putty config

```
>>> pin.value
1
```

To assign a value to Pin 0, the Pin object must be created as an *output* pin. An output pin is a pin where a program or user determines the pin output. An input pin is a pin set up to read input, like the input from a sensor. In this case we want to assign Pin 0 as an output pin.

```
>>> pin = machine.Pin(0, machine.Pin.OUT)
```

```
# turn the LED on
```

```
>>> pin.value(0)
```

```
# turn the LED off
```

```
>>> pin.value(1)
```

Run code at the Micropython REPL to blink the LED

Now we can write a for loop at the Micropython REPL to blink the LED on and off. In order to do this, we need to import the **machine** module and the **time** module.

```
>>> import machine
>>> import time
>>> pin = machine.Pin(0, machine.Pin.OUT)
>>> for i in range(10):
```

```

...     pin.value(1)
...     time.sleep(0.5)
...     pin.value(0)
...     time.sleep(0.5)
# backspace to exit loop and execute look.
....

```

13.6 Reading a Sensor

This is the fifth section of a chapter on Micropython. In this section, you will learn how to connect a temperature sensor to an Adafruit Feather Huzzah ESP8266 and use the Micropython REPL to read the temperature.

Before we can use the [MCP9808 temperature sensor](#) running on the Adafruit Feather Huzzah ESP8266, Micropython needs to be installed on the board and Putty needs to be installed in Windows 10 to communicate with the board over serial. Alternatively, the MacOS or Linux terminal can be used for communication.

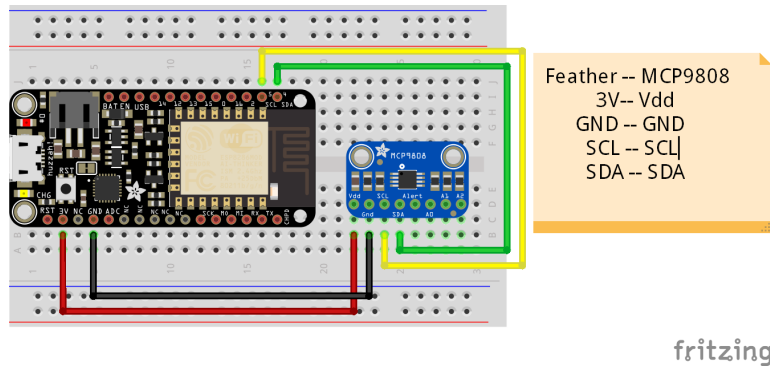
Summary of Steps

1. Connect the temperature sensor to the Adafruit Feather Huzzah ESP8266
2. Connect the Adafruit Feather Huzzah ESP8266 to the computer with a USB cable and bring up the Micropython REPL using Putty.
3. Run code at the Micropython REPL to read the temperature

Connect the MCP9808 temperature sensor to the Adafruit Feather Huzzah board

Connect the [MCP9808 temperature sensor](#) breakout board to the Feather Huzzah board with jumper wires. There are four connections: A 3V power line from the Feather Huzzah to the MCP9808 Vdd pin, GND connected between both boards, and the I2C data and clock lines connected between the two boards. On the Feather Huzzah ESP8266, the I2C data line is SDA (pin 4) and the I2C clock line is SCL (pin 5). These connect with the MPC9808 I2C data line SDA and the MPC9808 I2C clock line SCL. Unlike Serial communication where RX connects to TX, in I2C communication SDA connects to SDA and SCL connects to SCL.

| Feather Huzzah | wire | MCP9808 |
|----------------|--------|---------|
| 3V | red | Vdd |
| GND | black | GND |
| SDA (pin 4) | green | SDA |
| SCL (pin 5) | yellow | SCL |



Fritzing Image

Connect the Adafruit Feather Huzzah to the computer with a USB cable and bring up the Micropython REPL using Putty.

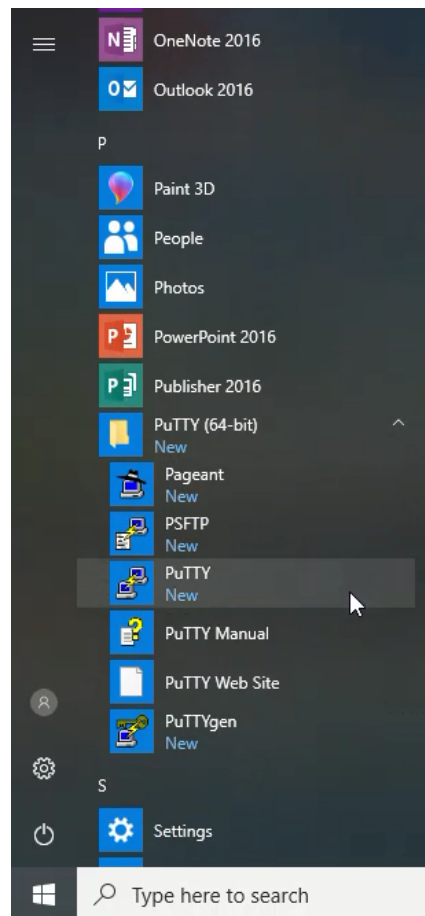
Connect the Adafruit Feather Huzzah ESP8266 to the computer with a microUSB cable. Ensure this is a USB data cable, not just a charging cable. Open Putty and connect to the Feather Huzzah using the proper serial port (COM#) and 115200 baud. (Remember to use the **Serial** radio button under **Connection Type**.)

This will bring up the Micropython REPL prompt `>>>`. If you can't see the `>>>` prompt, try typing [Enter] or Ctrl-D or push the RESET button on the Feather Huzzah. If none of these methods work, try closing Putty and unplugging then replugging in the USB cable.

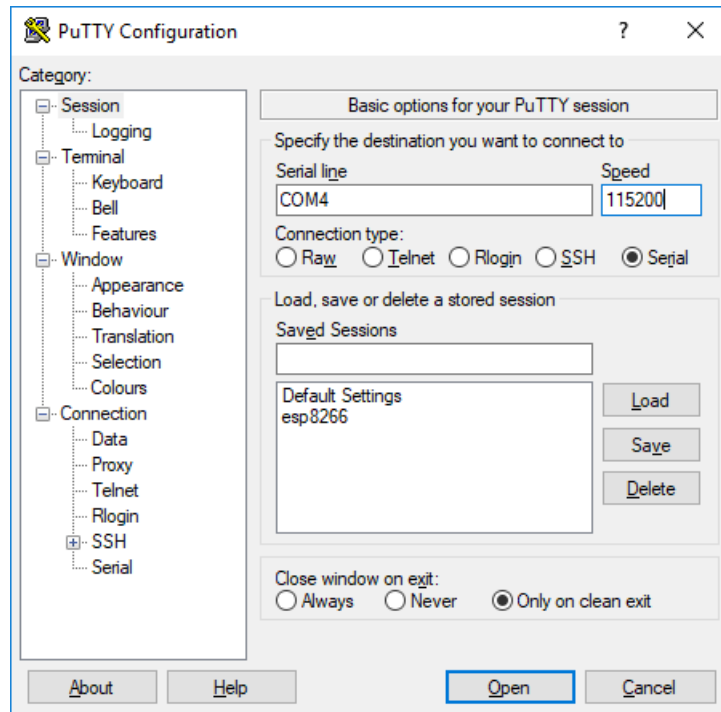
Run code at the Micropython REPL to read the temperature

In the Putty Serial Window, import the machine module and then create an instance of the machine.I2C class with the `scl` and `sda` parameters set as `scl=machine.Pin(5)` and `sda=machine.Pin(4)`. Then create an empty bytearray which will store the data coming in from the MCP9808 temperature sensor. As strings in Micropython are UTF-8 encoded by default (like in Python 3), a *bytearray* needs to be used to read the raw output from the MCP9808 chip registers. The command `i2c.readfrom_mem_into()` method brings in the data from the sensor and saves it to the `byte_data` variable. The arguments inside the `i2c.readfrom_mem_into()` method 24 and 5 correspond to the I2C memory address and registry address of the temperature data stored in the MCP9808 temperature sensor.

```
>>> import machine
>>> i2c = machine.I2C(scl=machine.Pin(5), sda=machine.Pin(4))
>>> byte_data = bytearray(2)
>>> i2c.readfrom_mem_into(24, 5, byte_data)
>>> value = byte_data[0] << 8 | byte_data[1]
>>> temp = (value & 0xFFF) / 16.0
>>> if value & 0x1000:
...     temp -= 256.0
>>> print(temp)
```



Putty in start menu



Putty config

13.7 Uploading code

In this section you will learn how to create a working WiFi weather station built using an Adafruit Feather Huzzah ESP9266, and a temperature sensor. The working WiFi weather station will post the temperature to [ThingSpeak.com](https://thingspeak.com)

Before the Micropython REPL (the Python prompt) running on the Adafruit Feather Huzzah ESP8266 can be used, Micropython needs to be installed on the board. Putty also needs to be installed on a computer in order for the computer to communicate with Feather Huzzah over serial. See the previous section on how to install Micropython on an Adafruit Feather Huzzah ESP9266 and how to install Putty on a Windows computer.

Summary of Steps

1. Install **ampy** with **pip**
2. Write python code
3. Put the code on the board with **ampy**
4. Run functions from the Micropyton REPL
5. Run a program

Install ampy with pip

Ampy is a Python package developed by Adafruit, the company that makes the Feather Huzzah board. **Ampy** is used to push code stored on a computer to the Feather Huzzah board. **Ampy** can be installed using the **Anaconda Prompt**. Alternatively, **pip** can be used to install **ampy**. If using a virtual environment, active the virtual environment first before proceeding with the **ampy** package installation.

```
> conda activate micropython
(micropython) > pip install ampy-adafruit
(micropython) > ampy --help
```

Write Python Code

Now write the Python code that will be put on the board. The Feather Huzzah board has two main Python files: **boot.py** and **main.py**. Additional files can also be added to the board. **boot.py** is the file that runs first when the board is powered up. After **boot.py** runs, then **main.py** will run. Another **.py** file can be added to the board to provide **main.py** with some functions and classes to work with.

Two general things need to be accomplished on the Feather Huzzah board to turn it into a WiFi weather station: read the temperature and post the temperature to ThingSpeak.com. Two different **.py** files will be constructed, one **.py** file for each of these general functionalities (reading the temperature and posting the temperature).

The first **MCP9808.py** file will simplify reading temperature data off of the Adafruit MCP9808 temperature breakout. A function that parses out the temperature data from the I2C bus and return it as the output for the **readtemp** function will be created. The function needs to import the **machine** module to use the I2C bus. The **machine** module provides the class to create a new **i2c** object. When the **i2c** object is instantiated, the **scl** and **sda** pins that the sensor is connected to need to be specified. **scl** is the **i2c** clock line and **sda** is the **i2c** data line. These are pins 5 and 4 on the Adafruit Feather Huzzah. Then a new byte array variable needs to be created, so that later data from the sensor can be saved into it. Next read the sensor data using the **i2c.readfrom_mem_into()** function. The first argument is the I2C bus address for the sensor. In this case the sensor is at I2C bus address 24. The line **>>> i2c.scan()** typed into the micropython REPL will show the I2C bus address. The next function argument is the register on the MCP9808 temperature sensor where the temperature value is stored, which happens to be register 5. If register 5 is accessed on the MCP, the temperature can be recorded. The third argument is the variable to store the temperature data into. The **i2c.readfrom_mem_into()** method changes the variable that is a method argument, rather than changing a variable which is the method output as most methods do. This is why the **byte_data** variable needs to be created before calling the **i2c.readfrom_mem_into()** method. Next some post processing is needed to convert the byte array into a temperature in degrees C.

```
# MCP9808.py

# Functions for the MCP9808 temperature sensor
# https://learn.adafruit.com/micropython-hardware-i2c-devices/i2c-master

def readtemp():
    import machine
    i2c = machine.I2C(scl=machine.Pin(5), sda=machine.Pin(4))
```

```

byte_data = bytearray(2)
i2c.readfrom_mem_into(24, 5, byte_data)
value = byte_data[0] << 8 | byte_data[1]
temp = (value & 0xFFF) / 16.0
if value & 0x1000:
    temp -= 256.0
return temp

```

Now build a Python file for the set of WiFi functions.

```
#wifitools.py
```

```

# Wifi connection and post functions for an ESP8266 board running micropython
#https://docs.micropython.org/en/v1.8.6/esp8266/esp8266/tutorial/network_basics.html

```

```

def connect(SSID,password):
    import network
    sta_if = network.WLAN(network.STA_IF)
    if not sta_if.isconnected():
        print('connecting to network...')
        sta_if.active(True)
        sta_if.connect(SSID, password)
        while not sta_if.isconnected():
            pass
    print('network config:', sta_if.ifconfig())

```

```
#https://docs.micropython.org/en/v1.8.6/esp8266/esp8266/tutorial/network_tcp.html
```

```

def http_get(url):
    import socket
    _, _, host, path = url.split('/', 3)
    addr = socket.getaddrinfo(host, 80)[0][-1]
    s = socket.socket()
    s.connect(addr)
    s.send(bytes('GET %s HTTP/1.0\r\nHost: %s\r\n\r\n' % (path, host), 'utf8'))
    while True:
        data = s.recv(100)
        if data:
            print(str(data, 'utf8'), end='')
        else:
            break

```

```

def thingspeak_post(API_key,data):
    if not isinstance(data, str):
        data = str(data)
    #base_url = 'https://api.thingspeak.com/update?api_key=THECLASSAPIKEY&field1='
    base_url = 'https://api.thingspeak.com/update?api_key='
    mid_url = '&field1='
    url = base_url + API_key + mid_url + data
    http_get(url)

```


Now we will construct a Micropython script called *main.py* which will use the functions stored in *MCP9808.py*. The *main.py* script will import the *MCP9808.py* and *wifitools* functions then use the `wifitools.connect()` function to connect to the WiFi network. A `time.sleep(5)` line allows time for the Feather Huzzah board to connect to the WiFi network. Next we'll run a loop for a total of 8 hours at 60 minutes each hour. Inside the loop, we'll read in the temperature from the MCP9808 using the `MCP9808.readtemp()` function and post it to ThingSpeak.com using the `wifitools.thingspeak_post()` function. To take the temperature once a minute, we need to `time.sleep(60)` (wait 60 seconds) between each measurement.

```
# main.py
# ESP8266 Feather Huzzah Weather Station

import wifitools
import MCP9808
import time
import config

api_key = config.API_KEY
ssid = config.SSID
password = config.WIFI_PASSWORD

wifitools.connect(ssid,password)
time.sleep(5)

for i in range(8*60):
    data = MCP9808.readtemp()
    wifitools.thingspeak_post(api_key,data)
    time.sleep(60)
```

Upload Python Code to the Feather Huzzah with ampy

Once all the *.py* files are created, ensure the Feather Huzzah board is connected with a USB cable, and be aware of which serial port the Feather Huzzah is connected to. Upload the code files to the board using **ampy**. Make sure you are in the directory with the *.py* files and that you are working in a virtual environment that has **ampy** installed in it. In the example code below, the (micropython) virtual environment is active.

```
(micropython) > ampy --port COM4 put MCP9808.py
(micropython) > ampy --port COM4 put wifitools.py
(micropython) > ampy --port COM4 put main.py
(micropython) > ampy --port COM4 ls
boot.py
wifitools.py
MCP9808.py
config.py
main.py
```

Unplug and power up the Feather Huzzah and watch the data on ThingSpeak.com

The Feather Huzzah needs to be restarted to run the code uploaded with **ampy**. To restart the board, unplug and then replug in the board's power (the USB cable). Once power is restored, the board will run through the *boot.py* script then start the *main.py* script. When it runs the *main.py* script, the board will connect to the WiFi network, read the temperature then upload the temperature to ThingSpeak.com. When ThingSpeak.com is viewed, you will see the temperature plotted on the ThingSpeak.com Channel's page.

13.8 Summary

Key Terms and Concepts

Micropython

Microcontroller

Baud Rate

13.9 Review Questions

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.
- 7.
- 8.
- 9.

Chapter 14

Appendix

14.1 Contents

The following will be detailed in the appendix:

- Reserved and Key Words in Python
- Python **math** and **statistics** module fuctions
- Answers to selected problems
- Glossary
- About the Author
- About the contrirubuoers

14.2 Reserved and Key Words in Python

The following are reserved and key words in Python. These words should not be used as the names for user-defined functions, classes, methods or modules. The key words can be accessed with the following code:

```
In [5]: import keyword
        print('There are {} key words'.format(len(keyword.kwlist)))
        for keywrđ in keyword.kwlist:
            print(keywrđ)
```

```
There are 33 key words
False
None
True
and
```

as
assert
break
class
continue
def
del
elif
else
except
finally
for
from
global
if
import
in
is
lambda
nonlocal
not
or
pass
raise
return
try
while
with
yield

Logical Key Words

True
False
not
and
or
is
None
in

Control Flow Key Words

if

else
elif
for
while
break
continue
pass
try
except
finally
raise
return
yield

Definition Key Words

def
global
nonlocal
class
lambda
with
assert
del

Module Key Words

import
from
as
with

14.3 ASCII Character Codes

The following is a list of ASCII character codes. These character codes can also be accessed using the following code:

```
In [2]: for ASCIIcode in range(32,127):
        print('ASCII code: {}      Character: {}'.format(ASCIIcode,chr(ASCIIcode)))

ASCII code: 32      Character:
ASCII code: 33      Character: !
ASCII code: 34      Character: "
ASCII code: 35      Character: #
ASCII code: 36      Character: $
ASCII code: 37      Character: %
ASCII code: 38      Character: &
ASCII code: 39      Character: '
ASCII code: 40      Character: (
ASCII code: 41      Character: )
ASCII code: 42      Character: *
ASCII code: 43      Character: +
ASCII code: 44      Character: ,
ASCII code: 45      Character: -
ASCII code: 46      Character: .
ASCII code: 47      Character: /
ASCII code: 48      Character: 0
ASCII code: 49      Character: 1
ASCII code: 50      Character: 2
ASCII code: 51      Character: 3
ASCII code: 52      Character: 4
ASCII code: 53      Character: 5
ASCII code: 54      Character: 6
ASCII code: 55      Character: 7
ASCII code: 56      Character: 8
ASCII code: 57      Character: 9
ASCII code: 58      Character: :
ASCII code: 59      Character: ;
ASCII code: 60      Character: <
ASCII code: 61      Character: =
ASCII code: 62      Character: >
ASCII code: 63      Character: ?
ASCII code: 64      Character: @
ASCII code: 65      Character: A
ASCII code: 66      Character: B
ASCII code: 67      Character: C
ASCII code: 68      Character: D
ASCII code: 69      Character: E
ASCII code: 70      Character: F
ASCII code: 71      Character: G
ASCII code: 72      Character: H
ASCII code: 73      Character: I
ASCII code: 74      Character: J
ASCII code: 75      Character: K
ASCII code: 76      Character: L
ASCII code: 77      Character: M
ASCII code: 78      Character: N
```

| | |
|-----------------|--------------|
| ASCII code: 79 | Character: O |
| ASCII code: 80 | Character: P |
| ASCII code: 81 | Character: Q |
| ASCII code: 82 | Character: R |
| ASCII code: 83 | Character: S |
| ASCII code: 84 | Character: T |
| ASCII code: 85 | Character: U |
| ASCII code: 86 | Character: V |
| ASCII code: 87 | Character: W |
| ASCII code: 88 | Character: X |
| ASCII code: 89 | Character: Y |
| ASCII code: 90 | Character: Z |
| ASCII code: 91 | Character: [|
| ASCII code: 92 | Character: \ |
| ASCII code: 93 | Character:] |
| ASCII code: 94 | Character: ^ |
| ASCII code: 95 | Character: _ |
| ASCII code: 96 | Character: ` |
| ASCII code: 97 | Character: a |
| ASCII code: 98 | Character: b |
| ASCII code: 99 | Character: c |
| ASCII code: 100 | Character: d |
| ASCII code: 101 | Character: e |
| ASCII code: 102 | Character: f |
| ASCII code: 103 | Character: g |
| ASCII code: 104 | Character: h |
| ASCII code: 105 | Character: i |
| ASCII code: 106 | Character: j |
| ASCII code: 107 | Character: k |
| ASCII code: 108 | Character: l |
| ASCII code: 109 | Character: m |
| ASCII code: 110 | Character: n |
| ASCII code: 111 | Character: o |
| ASCII code: 112 | Character: p |
| ASCII code: 113 | Character: q |
| ASCII code: 114 | Character: r |
| ASCII code: 115 | Character: s |
| ASCII code: 116 | Character: t |
| ASCII code: 117 | Character: u |
| ASCII code: 118 | Character: v |
| ASCII code: 119 | Character: w |
| ASCII code: 120 | Character: x |
| ASCII code: 121 | Character: y |
| ASCII code: 122 | Character: z |
| ASCII code: 123 | Character: { |
| ASCII code: 124 | Character: |
| ASCII code: 125 | Character: } |
| ASCII code: 126 | Character: ~ |

14.4 Virtual Environments

Using **virtual environments** is a good standard of practice in Python

14.5 Numpy Math Functions

The code below will print out all of the numpy functions and methods:

```
In [7]: #import numpy as np
        #for func in dir(np):
        #    print(func)
```

Numpy Statistics Functions and Methods

```
np.mean
np.median
np.std
np.var
np.erf
```

Numpy Trigonometric Functions and Methods

```
np.pi
np.sin
np.cos
np.tan
np.arcsin
np.arccos
np.arctan
np.arcsinh
np.arccosh
np.arctanh
np.arctan2
np.radians
np.rad2deg
np.deg2rad
np.radians
```

```
np.sinc
np.sinh
np.tanh
```

```
np.angle
```

Numpy Exponential and Logarithmic Functions and Methods


```

np.log
np.log10
np.log1p
np.log2
np.logaddexp
np.logaddexp2
np.exp
np.exp2
np.sqrt
np.power
np.e

```

Numpy Matrix Creation and Manipulation Functions and Methods

```

np.linspace
np.zeros
np.ones
np.ndarray
np.matrix
np.transpose
np.size
np.shape
np.reshape
np.meshgrid
np.dot
np.asmatrix
np.asarray
np.arange
np.array

```

14.6 Git and github.com

Git is a common version control tool used by computer program developers to save code and work on code as a team. **Git** is a program run from the command line or **Anaconda Prompt**.

github.com is a website and service used by open source projects to share code and allow other users to make changes to existing code.

Both **git** and **github.com** are useful of engineers working in teams.

To use git and github the understanding of a few terms is import:

- **git** - a command line program used to track file changes and colaborate on code with others
- **repo** - short name for “repository”. A repo is a directory that contains files and other subfolders with files
- **local repo** - a directory that contains files and subfolders on your computer that git knows about
- **remote repo** - a set of files and subfolders stored in the cloud that git knows about

Useful git commands are summarized below:

```

git init initialize a new repository
git remote add origin https://github.com/user/repo.git links a local git repo with a remote git repo on github.com
git add . adds all the files and changes to the local git repo
git commit -m "commit message" commits the changes in the local repo
git push origin master pushes the changes up to the remote repo on github.com
git pull origin master pulls the version in the remote repo down to the local repo
git clone https://github.com/user/repo.git copies a remote repo on github.com to a local directory

```

14.7 LaTeX Math

Latex Math can be included in Jupyter Notebook markdown cells and parts of matplotlib plots. A list of usable LaTeX commands are below

```

\frac{}{}
^{ }
_{ }

\pi
\delta
\epsilon
\sigma

```

14.8 Python for Undergraduate Engineers Book Construction

Jupyter Notebooks

This book was constructed using **jupyter notebooks** saved on github.com. The github repo for the books can be found at:

<https://github.com/ProfessorKazarinoff/PythonForUndergraduateEngineers>

The directory structure of the github repo contains the **jupyter notebooks** that were use the write the book, a set of conversion tools and the output of these conversion tools, the book website and pdf's that make the hard copy of the book

```

PythonForUndergraduateEngineers/
|-- conversion_tools/
|-- notebooks/
|-- LICENSE
|-- notebooks/
|-- pdf/
|-- README.md
|-- website/

```

The notebooks directory contains a folder for each chapter of the book:

```
Notebooks/
|-- 00-Preface/
|-- 01-Orientation/
|-- 02-The-Python-REPL/
|-- 03-Data-Types-and-Variables/
|-- 04-Jupyter-Notebooks/
|-- 05-Functions-and-Modules/
|-- 06-Plotting-with-Matplotlib/
|-- 07-If-Else-Try-Except/
|-- 08-Loops/
|-- 09-Linear-Algebra/
|-- 10-Symbolic-Math/
|-- 11-Python-and-External-Hardware/
|-- 12-MicroPython/
|-- 99-Appendix/
|-- figures/
`-- TOC.ipynb
```

Within each chapter directory there is a **jupyter notebook** for each section and an images directory for any images used in the markdown sections of the notebooks

```
01-Orientation/
|-- 01.00-Welcome.ipynb
|-- 01.01-Why-Python.ipynb
|-- 01.02-Installing-Python.ipynb
|-- 01.03-Installing-Anaconda.ipynb
|-- 01.04-Installing-Anaconda-on-OSX.ipynb
|-- 01.05-Summary.ipynb
|-- 01.06-Review-Questions.ipynb
`-- images/
```

Website

The website for the book was constructed using `mkdocs`. Jupyter notebooks were exported to html with the markdown cells unformatted.

Hardcopy

The hard copy of the book was constructed using **LateX** and **nbconvert**. A conversion script combined all of the notebooks into one BIG notebook and converted the one BIG notebook to **LaTeX** using a custom template.

14.9 About the Author

Peter D. Kazarinoff, PhD is a full-time faculty member in Engineering and Engineering Technology at Portland Community College in Portland, OR. Peter earned a PhD in Engineering from the

University of Washington and a BA from Cornell University.

Peter currently lives in Portland OR with his wife and two kids.