Scientific Computing

Lance Nelson

7/16/2022

Table of contents

1 Python and Jupyter Notebooks

1.1 Python

Python is a computer programming language available on all major platforms (Mac, Windows, Linux). Python is a scripting language which means that the computer interprets and runs your code at the moment you run it. In contrast, with a compiled language like C the code must first be converted into binary before it can run (called "compiling" the code). There are pros and cons to both types of languages. The on-the-fly interpretation of Python makes it quick and easy to write code and provides fast results for simple calculations. When codes become longer and more complex, on-the-fly interpretation becomes less efficient and execution time will be much slower than it would be with a compiled language. The pros and cons flip for a comiled language; writing code in a compiled language can be cumbersome and slow, but the execution time is typically much faster. Out of necessity, most programmers become proficient in both types of languages. Python(or another interpreted language) is used to "toy around" with your problem and build familiarity. As the complexity of the code increases the user is then forced to transition to a compiled language to get the needed speed. This is the famous "two language" problem and there is a new programming language designed to eliminate this problem by combining the pros from both into one language. (The name of the language is Julia)

Python is free, open source software and is maintained by the non-profit Python software foundation. This is great because it means that you will always have free access to the Python language regardless of what organization or univeristy you are affliated with. You'll never have to worry about not being able to use your Python code without paying for it. Another benefit of open source languages is that all of the codes developed by other people are available for anyone to inspect, modify, and use. This allows anyone to review another's code to ensure that it does what they say it does, or to modify it to do something else. One last benefit that comes with an open source language is the community of Python users available to answer questions and provide instruction to the beginner. Answers to most questions about python are readily available on tutorial or forum websites.

1.2 Installing Software

The first step is to install the software (if you haven't already). The most convenient way to install Python and also get many of the commonly-used libraries is to use an installer. I recommend Anaconda. When installing the software be sure to choose Python 3 since this is the current version. By default, Anaconda will install a suit of sotwares and libraries that are commonly used. If you want to install other Python libraries, open the Anaconda-Navigator (green circle icon) and select the **Environment** tab on the left. Select **Not Installed** from the pull-down to see all of the libraries that are available to be installed. To install a library, check the box next to it and click **Apply**. Anaconda will take care of the rest.

To Do:

1. Install Anaconda
2. Check to see if the library "numpy" is installed. If not, install it.

1.3 Jupyter Notebooks

A Jupyter notebook is an electronic document designed to support interactive data processing, analysis, and visualization in an easily shared format. A Jupyter notebook can contain live code, math equations, explanatory text, and the output of codes (numbers, plots, graphics, etc..). To launch a Jupyter notebook, first open Anaconda-Navigator (green circle icon) and click the **Launch** button under JupyterLab. Jupyter can also be launched from the command line by typing `jupyter-lab`. The jupyter notebook will launch in your default web browser, but it is not a website. From here you can select an already existing Jupyter notebook, denoted by the orange icons and the .ipynb extension, or create a new notebook by clicking **New** from the **File** menu.

To Do:

1. On iLearn, find the module entitled "Jupyter Notebooks" and download the file "Intro.ipynb".
2. Launch JupyterLab as explained above.
3. Open "Intro.ipynb" that you downloaded in step 1 and continue reading this book in the jupyter notebook.

1.3.1 Notebook Structure

There are two types of "cells" in a Jupyter notebook: code cells and text cells (also called Markdown cell). Code cells contain "live" Python code that can be run inside of the notebook with any output appearing directly below it. Markdown cells are designed to contain

explanatory information about what is happening inside of the code cells. They can contain text, math equations, and images. Markdown cells suport markdown, html, and Latex (for generating pretty math equations).

Both markdown and code cells can be executed by either selecting **Run Selected Cells** in the **Run** menu, by clicking the **Play** icon at the top of the notebook, or by using the **Shift-Return** shortcut when your curson is in the desired cell.

1.3.2 Navigating Jupyter Notebooks

Navigating a Jupyter notebook is fairly straightforward but there are a few handy shortcuts/hotkeys that will make navigation quicker and your workflow more efficient. When working in a Jupyter notebook, you are always operating in one of two modes: edit mode or navigate mode. In edit mode you can make modifications to the text or code in a cell and in navigate mode you can add/delete cells and modify the cell type. If you can see a blinking cursor in one of the cells you are in edit mode. Otherwise you are in navigate mode. To exit edit mode, simply press the `esc` key and you will enter navigate mode. To exit navigate mode, simply press the `enter` key and you will enter edit mode for the cell you were focused on. (You can also double click on a cell with your mouse to enter edit mode.) The `shift` + `enter` key sequence will "execute" a cell and produce the associated output. For text cells, executing just means to render the text in a nicely formatted fashion. "Executing" a code cell will actually execute the code block contained in the cell. You also enter navigate mode every time you execute a cell using the `shift` + `enter` key sequence. A summary of these shortcuts is given below:

- Up/down arrows - Navigate to different cells in the notebook.
- `Y` - turns a text cell into a code cell.
- `M` - turns a code cell into a text cell.
- `A` - inserts a new cell above the current cell.
- `B` - inserts a new cell below the current cell.
- `X` - deletes the current cell.
- `enter` - enters edit mode.
- `shift` + `enter` - execute a cell.
- `esc` - enter navigate mode.

You should take some time now to practice these shortcut keys until you become good at navigating a jupyter notebook. Consider attempting the following actions using the shortcuts above:

1. Add a cell below this one.
2. Turn the cell into a code cell (observe the distinct appearance of code cells).
3. Type the following code into the cell:
    - `print("I did it!")`

4. Execute the cell using `shift` + `enter`. Observe the output.
5. Delete the cell.
6. Enter edit mode for this text cell.
7. Add a sentence of your choice at the end of the cell.
8. "Execute" the cell and observe the new output.
9. Remove the sentence to restore the cell to it's previous state.

1.4 Tips for studying Jupyter notebooks

1.4.1 Print Statements

Jupyter notebooks in this class will be a nice mix of text cells (explanation) and code cells (examples). You will soon learn that code cells produce no output unless you explicitly tell them to using a `print` statement (similar to the one you used above). When you encounter a code cell, you should feel free to make modifications and additions to the cell until you fully understand how the code works.

1.4.2 Comments

Comments are a way to describe what each section of code does and makes it easier for you and others to understand the code. It may seem clear what each section of code does as you write it, but after a week, month or longer, it is unlikely to be obvious. Paul Wilson of the University of Wisconson at Madison is quoted as saying, "Your closest collaborator is you six months ago, but you don't reply to emails." Comment your code now so that you are not confused later.

There are several ways to add comments to your code:

1. Use `#` to start a comment. Everything on that line the follows will be ignored.
2. For longer comments that will span several lines, use triple double quotes to begin and end the comment (`"""`)

The cell below illustrates these two ways to make comments:

```
# Speed of light in a vacuum
c = 3e8

v = 300 # Speed of sound in air

"""
The variables below are the initial conditions for a cannon
```

```
launching a ball at a 30 degree angle with an initial speed of
50 m/s.  The initial height of the cannon ball is 1000 m
"""
v = 50
theta = 30
h_i = 1000
```

To Do:

1. Execute the code block below and verify that no output is produced.
2. Add print statements that help you see the result of the calculation.
3. Add simple commments next to each line explaining the code.

```
a = 2
b = 3
c = a**b
```

```
import subprocess
import sys

def install(package):
    subprocess.check_call([sys.executable, "-m", "pip", "install", package])
install("numpy")
```

Part I

Python - The Basics

2 Variables and Numbers

2.1 Variables

When performing mathematical operations, it is often desirable to store the values in *variables* for later use instead of manually typing them back in each time you need to use them. This will reduce effort because small changes to variables can automatically propagate through your calculations.

Attaching a value to a variable is called *assignment* and is performed using the equal sign (=), as demonstrated in the cell below:

```
a = 5.0
b = 3
c = a + b
```

2.1.1 Variable naming convention

There are some rules for allowed variable names in Python. They are as follows:

1. Variable names must begin with a letter or an underscore (_)
2. Variables names must only contain letters, numbers, and underscores.
3. Variable names cannot contain spaces.
4. Variables names cannot be a word reserved by Python for something else. These words are:

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

The cell below contains some allowed variable names and some that are not allowed.

To Do:

1. Determine which variable names are allowed and which are not in the cell below.
2. What does Python do if you try to define a variable using a name that is not allowed?

```
my1variable = 3
1stvariables = 2
a big constant = 3
a_big_constant = 1e8
```

It is also a good practice to make variable names meaningful. For example, in the cell below we calculate $E = mc^2$ using two choices for variable assignments. In one case, it is easy to determine what the calculation is and in the other it isn't.

```
# Good Variable Names
mass_kg = 1.6
light_speed = 3.0e8
E = mass_kg * light_speed**2


# Poor Variable Names
a = 1.6
b = 3.0e8
c = a * b**2
```

2.1.2 Numbers: Integers and Floats

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

Mathematical operations that only use integers *and* evaluate to a whole number will generate an integers (except for division). All other situations will generate a float. See the example cell below.

```
a = 24
b = 6
```

```
d = 0.3
e = a + b # Produces an integer.
f = a + d # Produces a float
g = a * b # Produces a ???
h = a / b # Produces a ???
```

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

```
int(3.0)
float(4)
```

```
4.0
```

The distinction between floats and ints is often a minor detail. Occasionally, a function will require that an argument be a float or an int but usually you won't have to worry about which one you use.

Below you will find some other common mathematical operations that can be performed on numerical variables.

```
a = 20
b = 10
c = a + b
d = a/b
r = a//b
r = a % b
e = a * b
f = c**4
```

To Do:

1. Use print statements to investigate what each operation does.

2. Can you force each operation to produce a float and an integer?
3. Add comments next to each line (Use `#` to start a comment) explaining that operation.

2.1.3 Augmented Assignment

Augmented assignment is a shortened way to make a simple modification to a variable. For example, if we want to increase the value of a variable by 10, one way to do it would be like

this.

```
a = 5
a = a + 10
```

This is certainly not difficult, but it does involve typing the variable twice which becomes cumbersome as your variable name gets longer. Alternatively, we can accomplish the same thing with the **+=** operator.

```
a = 5
a += 10
```

Augmented assignment can be used with addition, subtraction, multiplication, and division as shown in the code cell below.

To Do:

1. Predict what the final result of a will be in the code cell below.
2. Add an appropriately-place print statement to see if you were correct.
3. If you were wrong, pow-wow with your neighbor until you understand.

```
a = 7
a += 3
a -= 1
a *= 4
a /= 3
```

2.1.4 Compound Assignment

At the beginning of a program or calculation, it is often necessary to define a set of variables. Each variable may get it's own line of code, but if there are a lot of variables, this can begin to clutter your code a little. An alternative is to assign multiple variables on a single line. In the code below, we assign the atomic mass of the first three elements.

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

To Do:

1. Use print statements to verify that each variable was assigned it's own value.
2. Add assignments for the atomic masses of the next three elements on the periodic table.

2.1.5 Large numbers

Sometimes you find yourself working with large numbers in your calculation. Maybe your calculation involves the use of ten billion, which has 10 zeros in it. It can be difficult to look at all of those zeros with no commas to help break it up. In those cases, you can use an underscore (_) in place of the comma, as shown below.

```
myLargeNumber = 10000000000 # This is tough to look at.
myLargeNumber = 10_000_000_000  # This is easy to read

myLargeFloat = 5000000.6 # This is tough to read
myLargeFloat = 5_000_000.6 # This is easy to read
```

2.1.6 Very Large Numbers

If your number is very large or very small ($20-30$ zeros), you would probably rather not have to type all of the zeros at all, even if you can break it up with the underscores. For example, the Boltzman constant, which comes up in thermodynamics, has a value equal to

$$1.38 \times 10^{-23}$$

We can avoid typing all those zeros by using scientific notation when defining the variable. (see example below) This is super handy for very large and very small number. (Numbers of both variety show up frequently in physics!)

```
kB = 1.38e-23
```

2.1.7 Python functions

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

$$f(x)$$

In the cell below, you will find several useful math equations.

```
abs(-5.5)
float(2)
int(5.6)
print(1.26e-6)
round(-5.51)
str(3.2)
```

```
1.26e-06
```

```
'3.2'
```

In addition to Python's native collection of mathematical functions, there is also a `math` module with more mathematical functions. Think of a module as an add-on or tool pack for Python just like a library. The `math` module comes with every installation of python and can be *imported* (i.e. activated) using the `import math` command. After the module has been imported, any function in the module is called using `math.function()` where `function` is the name of the function. Here is a list of commonly-used function inside the `math module`:

```
import math
math.sqrt(4)
math.ceil(4.3)
math.cos(1.5)
math.sin(1.5)
math.degrees(6.28)
math.e
math.exp(5)
math.factorial(4)
math.log(200)
math.log10(1000)
math.radians(360)
math.tan(3.14)
math.pi
math.pow(2,8)
```

```
256.0
```

To Do:

1. Use print statements to figure out what each function in the code cell above does. Pay special attention to trigonometric function. Do these functions expect the argument to be in radians or degrees?

2. Add comments to remind yourself for later.

There are other ways to import functions from modules. If you only want to use a single function inside the module, you can selectively import it using `from`, as shown below.

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

```
0.06981317007977318
```

2.2 Displaying Results (The `print` Function).

You have already been using the `print` function a little and you should have noticed that Python will not display the result of a calculation/operation unless you include a print statement. In Python, you can pretty much `print()` anything and Python will just dump it to screen as it pleases. However, there are times when you may want to have a little more control over your print statements. For example, maybe you'd like to print a sentence but with some numerical values inserted occasionally:

```
age =22
gpa = 3.58342
c = 2.998e8
print("Hi, I am Joe. I am", age," years old and my gpa is", gpa, " and the speed of light
```

```
Hi, I am Joe. I am 22  years old and my gpa is 3.58342  and the speed of light is 299800000.
```

This print statement is a little awkward and the numbers appearing probably have more digits than we'd like to see. We can improve upon this print statement using something called f-string (short for formatted strings). To build an f-string, place an "f" immediately prior to the open quote and then everywhere you want a variable to appear, just enclose it in curly braces.

```
age =22
gpa = 3.5
c = 2.998e8
print(f"Hi, I am Joe. I am {age} years old and my gpa is {gpa} and the speed of light is {
```

```
Hi, I am Joe. I am 22 years old and my gpa is 3.5 and the speed of light is 299800000.0.
```

That's a clever way to insert a numerical value into a string, but the last number is still displaying too many digits. To control how the numbers are formatted, we can add a format code after the variable name:

```
age =22
gpa = 3.5
c = 2.998e8
print(f"Hi, I am Joe. I am {age:d} years old and my gpa is {gpa:5.2f} and the speed of lig
```

```
Hi, I am Joe. I am 22 years old and my gpa is  3.50 and the speed of light is 3.00e+08.
```

The structure of the stuff inside of the curly braces is as follows: `variable:formatcode` The format code indicates how you would like the variable to be formatted when it is printed. The `:d` indicates that the varible should be displayed as an integer and `:f` indicates a float. The amount of space that is allocated to display a number can be specified by placing a number in front of the `:`. The 5.2 in the float formatting indicates that the number should be displayed with at least 5 total spaces while displaying only 2 numbers after the decimal. A selection of some available format statements is given below.

format code	explanation
`{variable}`	Use the default format for the data type.
`{variable:4d}`	Display as an integer, allocating 4 spaces for it.
`{variable:.4f}`	Display as a float, with four numbers after the decimal being displayed.
`{variable:8.4f}`	Display as a float, allocating 8 total spaces and 4 numbers after the decimal place.
`{variable:8.4e}`	Display using scientific notation, allocating 8 total spaces and 4 numbers after the decimal place.

2.3 Exercises

1. Calculate the distance from the origin to the point $(23, 81)$ usign the `math.hypot()` function and then using the following distance equation:

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

2. Solve the quadratic equation using the quadratic function below for $a = 1$, $b = 2$, and $c = 1$.

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

(Optics Application) 3. When light encounters an interface between two different object, the light bends as it proceeds into the second material. The index of refraction (n) determines how much bending happens. Bending is greater for materials with a bigger index of refraction.

3 Working with Strings

3.1 Strings

Another commonly-used type of data is a string of characters known simply as *strings*. Strings can contain a variety of characters including letters, numbers, and symbols.

3.1.1 Creating Strings

Strings are created by placing the sequence of characters in single (or double) quotes.

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

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

```
text = str(4.5)
```

One common error made when working with strings is to attempt to perform math with them. Python will not perform math with strings because it sees them as a series of characters and nothing more. In the cell below, we attempt to perform math with some strings.

```
a = "4"
b = "2"
c = 2

d = a + b
e = a * b
f = b * c
```

To Do:

1. Use print statements in the cell above to determine what happens when you add two strings together.
2. Use print statements in the cell above to determine what happens when you multiply two strings.

3. Use print statements in the cell above to determine what happens when you multiply a string and an integer.

If you want to know the length of a string, you can use the `len()` function

```
text = "some text"
len(text)
```

```
9
```

The length of the string above is 9 because a space is a valid character.

3.1.2 Displaying text and numbers together

You have been using `print()` statements quite a lot lately (hopefully) but you probably haven't printed text and numbers together. To display both text and numbers in the same message, there are several options. The first is to just put multiple variables into the `print()` function, separating them with commas.

```
g = 9.8
unit = "m/s^2"
print(g,unit)
```

```
9.8 m/s^2
```

Another option is to convert the number to a string and then "add" it to the other string. This creates a single string as an argument to the `print()` function.

```
g = 9.8
unit = "m/s^2"
print(str(g) + unit)
```

```
9.8m/s^2
```

Notice the lack of space between the number and the unit. "Adding" the two strings smashed them together exactly as they were, no spaces added. You can insert multiple numbers into a string using something called "f"-strings. (short for formatted strings). To construct an f-string, simply place an "f" in front of the string. Anytime you want to insert a number in your string, enclose it in curly braces.

```
v = 5.0
c = 3e8
print(f"The speed of light is {c} and the speed of my car is {v}")
```

```
The speed of light is 300000000.0 and the speed of my car is 5.0
```

You can specify how the number should be formatted by placing a `:` after the variable name followed by a formatting tag. Here are a few examples to explore:

```
v1 = 5.0
v2 = 8.3
c = 2.998e8
n = 2

print(f"There are {n:d} cars traveling side by side.  One car is traveling at {v1:4.2f} m/
```

```
There are 2 cars traveling side by side.  One car is traveling at 5.00 m/s and the other is
```

As you can see, the formatting tag tells the print statement how the number should be printed. For float variables, the formatting tag will have two numbers followed by an "f" (for float). The first number indicates how many total spaces should be allocated to print the number and the second number specifies the number of decimal places that should be displayed. For integer variables, use the formatting tag "g". For bigger numbers, it is often useful to print the number in scientific notation. To do this, use the ".2e" formatting tag. The number after the decimal indicates how many numbers after the decimal should be displayed.

A summary of common format tags are given in the table below.

format code	explanation
`{variable}`	Use the default format for the data type.
`{variable:4d}`	Display as an integer, allocating 4 spaces for it.
`{variable:.4f}`	Display as a float, with four numbers after the decimal being displayed.
`{variable:8.4f}`	Display as a float, allocating 8 total spaces and 4 numbers after the decimal place.
`{variable:8.4e}`	Display using scientific notation, allocating 8 total spaces and 4 numbers after the decimal place.

To Do:

1. Modify the print statement above so that the float variables are given 8 total spaces with only 1 number after the decimal being displayed.
2. Modify the print statement above so that the speed of light is displayed with 3 numbers after the decimal place.

3.1.3 Indexing and Slicing

Accessing a piece (or slice) of a string is a common task in scientific computing. Often you will import data into Python from a text file and need to extract a portion of the file for later use in calculations. Indexing allows the user to extract a single element, or character, from a string. The key detail about indexing in Python is that *indices start from zero*. That means that the first character is index zero, the second character is index 1, and so on. For example, maybe a string contains the following amino acid sequence 'MSLFKIRMPE'. For this example, the indices are as follows:

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

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

```
seq = "MSLFKIRMPI"
seq[0]
```

```
'M'
```

To access the last character in a string you could do

```
seq = "MSLFKIRMPI"
seq[len(seq) - 1]
```

```
'I'
```

But this seems overly cumbersome. An easier approach is to index backwards. The string can be reverse indexed from the last character to the first using negative indices, starting with -1 as the last charcter.

```
seq = "MSLFKIRMPE"
seq[-1]
```

```
'E'
```

To Do:

1. Access the 5th character in the peptide sequence above.
2. Access the character that is 3rd from the end in the peptide sequence above.

Indexing only provides a single character, but it is common to want a series of characters from a string. *Slicing* allows us to grab a section of a string. Slicing is performed by specifying start and stop indices separated by a colon in the square brackets. One important detail worth mentioning: the character at the starting index is included in the slice while the character located at the final index *is not* included in the slice.

```
seq = "MSLFKIRMPE"
seq[0:5]
```

```
'MSLFK'
```

Looking at the string, you notice that the character at location 5 (I) has been excluded from the slice. You can leave off the first number when slicing and the slice will start at the beginning of the string.

```
seq = "MSLFKIRMPE"
seq[:5]
```

```
'MSLFK'
```

You can also use negative indices when slicing. This is especially helpful when you want to grab the last few characters in a string.

```
file = "data.txt"
ext = file[-3:]
```

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

```
seq = "MSLFKIRMPE"
seq[0:8:2]
```

```
'MLKR'
```

You can omit the start and stop indices and Python will assume that you are slicing the entire string.

```
seq = "MSLFKIRMPE"
seq[::2]
```

```
'MLKRP'
```

3.1.4 String Methods

A *method* is a function that works only with a specif type of object. String methods only work on strings, and they don't work on other types of objects, like floats or ints. If it helps you, you can just think of a method as a function.

One example of a string method is the `capitalize()` function which returns a string with the first letter capitalized. To use a method (referred to as *calling* the method), the method name is appended to the variable you want it to operate on. For example, below is an Albert Einstein quote that needs capitalized.

```
quote = "i want to know God's thoughts. The rest are details."
quote.capitalize()
print(quote)
```

```
i want to know God's thoughts. The rest are details.
```

Notice that the original variable (quote) remains unchanged. This particular method does not change the value of the original string but rather returns a capitalized version of it. If we want to save the capitalized version, we can assign it to a new variable, or overwrite the original.

```
quote = "i want to know God's thoughts. The rest are details"
quote = quote.capitalize()
print(quote)
```

```
I want to know god's thoughts. the rest are details
```

In the cell below you will find a list of commonly-used string methods.

```
a = "spdfgssfpggg"
a.capitalize()
a.center(10)
a.count("s")
a.find("d")
a.isalnum()
a.isalpha()
a.isdigit()
a.lstrip("s")
a.rstrip("g")
a.split("s")
a.startswith("s")
a.endswith("p")
```

```
False
```

To Do:

1. Use well-placed print statements to determine what each string method does.
2. Add comments next to each method for future reference.

3.2 Exercises

1. Removing file ext

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

2. Solve the quadratic equation using the quadratic function below for $a = 1$, $b = 2$, and $c = 1$.

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

4 Boolean Variables and Conditionals

4.1 Boolean Variables

We have learned about three types of variables in Python: ints, floats, and strings. Another variable type is a boolean, which can be one of two values: `True` or `False`. You can assign a boolean variable in the same way that you assign numbers or string, using `=`

```
myBool = True
```

`True` *must be* capitalized so don't try `true` or it won't be a boolean

```
myBool = true
```

4.1.1 Boolean Logic

Often you will want to check to see if some condition is true. For example, maybe you want to know if the radius of a certain satellite's orbit is bigger or smaller than Mercury's orbit. To perform this check, there are several boolean operators that will return `True` or `False`. Take note of the boolean operators shown in the cell below along with the comments added to explain what they do.

```
r1 = 3.5e8
r2 = 2.7e6

r1 > r2 # Is r1 greater than r2
r1 < r2 # Is r1 less than r2
r1 >= r2 # Is r1 greater than or equal to r2
r1 <= r2 # Is r1 less than or equal to r2
r1 != r2 # Is r1 not equal to r2
r1 == r2 # Is r1 equal to r2
```

```
False
```

A word of caution about comparing Python floats using == is in order here. Python floats are stored as a 53-digit, base-2 binary number(that's a mouthful). If you're interested in what that means, we can talk more. If you're not that interested, just know that when you define a float in Python, the number that is stored in the computer is not *exactly* the number that you think it is. This can cause problems when you are comparing two numbers that you think should be equal but actually aren't equal in the computer.[1] The code below illustrates this problem.

```
a = 0.1
b = 3 * a
c = 0.3
print(b==c)  # Are they the same number?  You would think they would
             # be right?
print(" {:.5f} ".format(b))     # It sure looks like they are the same.
print(" {:.5f} ".format(c))     # It sure looks like they are the same.
print(" {:.45f} ".format(b))  #b--- out to 45 decimal places
print(" {:.45f} ".format(c))  #c--- out to 45 decimal places
```

```
False
 0.30000
 0.30000
 0.300000000000000044408920985006261616945266724
 0.299999999999999988897769753748434595763683319
```

The first two print statements display the value of `b` and `c` to five decimal places. The second two print statements force Python to display the value of `b` and `c` out to 45 decimal places. Notice that the true value of `b` is not *exactly* equal to 0.3. This is why the statement `print(b==c)` returns `False`. The take home message here is that comparing two floats to see if they are equal **is always a bad idea.** A better way to check to see if two floats are equal (or close enough that we can say they are equal) is to check if the absolute value of their difference is very small, like this:

```
a = 0.1
b = 3 * a
c = 0.3
print(abs(b - c) < 1e-10)
```

```
True
```

[1] There is a library called `Decimal` that will fix a lot of these problems.

4.1.2 Compound Comparisons (Logical Operators)

Comparisons like those shown above can be chained together to make compound comparisons using the `and`, `or`, and `not` operators.

Operator	Description
and	Tests for both being `True`
or	Tests for either being `True`
not	Tests for `False`

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

Truth tables are a good way to visualize the output from compound comparisons.

p	q	p and q	p or q
`True`	`True`	`True`	`True`
`True`	`False`	`False`	`True`
`False`	`True`	`False`	`True`
`False`	`False`	`False`	`False`

As a simple example, suppose you know the density (ρ) and speed of light (c) for two materials and you'd like to know if both values are bigger for material 1 or material 2.

To Do:

1. Predict the output for each compound comparison given below. Add your guess as a comment.
2. Now add appropriately-placed print statements to check your guesses.
3. Modify your guess as needed and discuss any questions with a neighbor.

```
c1 = 2.5e8
c2 = 2.48e8
 1 = 450
 2 = 580

c1 > c2 and  1 >  2
c1 < c2 and  1 <  2
c1 > c2 and  1 <  2
```

```
c1 < c2 and  1 >  2

c1 > c2 and not  1 >  2
c1 < c2 and not  1 <  2

c1 > c2 or  1 >  2
c1 < c2 or  1 <  2
```

```
True
```

4.1.3 Tests for Inclusion

You can check for inclusion using the Python `in` operator. This provides an easy way to see if a character (or word) is present in a long string. Let's say you have a long string that contains the names of Jupyter's moons (there are 79 of them!!) and you want to see if a certain moon is included in the list. The `in` statement let's us quickly test to see if it is in the list. (see example below)

```
jupytermoons = "Metis,Adrastea,Amalthea,Thebe,Io,Europa,Ganymede,Callisto,Themisto,Leda,Hi

"Cyllene" in jupytermoons
```

```
True
```

4.2 Conditions

Conditions allow the user to specify if and when certain lines or blocks of code are executed. Specifically, when a condition is true, the block of *indented* code directly below it will run.

4.2.1 `if` statement

The `if` statement is used to control when a block of code runs. It's usage is shown below ending in a colon and the block of code below indented with *four spaces*. Using the **Tab** key will also produce four spaces.

```
jupytermoons = "Metis,Adrastea,Amalthea,Thebe,Io,Europa,Ganymede,Callisto,Themisto,Leda,Hi

if "Cyllene" in jupytermoons:
```

```
    found = True
    print("Found Cyllene in the list")

if "Matis" in jupytermoons:
    found = True
    print("Found Matis in the list")
```

```
Found Cyllene in the list
```

If the boolean statement after `if` is true, the indented code below it will run. If the statement is false, Python just skips the indented lines below.

4.2.2 `else` Statment

Sometimes there will be an alternate block of code that you want to run if the `if` statement evaluates to `False`. The `else` statement is used to specify this block of code, as shown below.

```
jupytermoons = "Metis,Adrastea,Amalthea,Thebe,Io,Europa,Ganymede,Callisto,Themisto,Leda,Hi

if "Cyllene" in jupytermoons:
    found = True
    print("Found Cyllene in the string")
else:
    found = False
    print("Did not find Cyllene in the string")


if "Matis" in jupytermoons:
    found = True
    print("Found Matis in the string")
else:
    found = False
    print("Did not find Matis in the string")
```

```
Found Cyllene in the string
Did not find Matis in the string
```

Notice that the `else` statement must be followed by a colon and the block of code to be executed is indented, just as in the `if` block.

There is an additional statement called the `elif` statement, short for "else if", which is used to add extra conditions below the initial `if` statement. The block of code below the `elif` statement only runs if the `if` statement is false and the `elif` statement is true. An example is given below.

```
jupytermoons = "Metis,Adrastea,Amalthea,Thebe,Io,Europa,Ganymede,Callisto,Themisto,Leda,Hi

if "Matis" in jupytermoons:
    foundMatis = True
    print("Found Matis in the string")
elif "Cyllene" in jupytermoons:
    foundCyllene = True
    print("Found Cyllene in the string.")
else:
    foundCyl = False
    foundMatis = False
    print("Did not find Cyllene or Matis in the string")
```

```
Found Cyllene in the string.
```

It is worth noting that `else` statements are not required. If you leave the `else` statement off and the `if` statement is false, no code block will execute.

5 Exercises

1. Removing file ext

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

2. Solve the quadratic equation using the quadratic function below for $a = 1$, $b = 2$, and $c = 1$.

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

6 Lists and Tuples

Up to this point, we have only worked with single numerical values or strings (multiple characters). Often we will want to work with collections of values (perhaps the orbital period of all of the planets in our solar system.) and it will be quite inconvenient to store each value into its own variable. Instead, the values can be stored in a list or tuple. Lists and tuples are both collections of *elements* , like numbers or strings. The key difference between them is that tuples are *immutable*, which means they cannot be modified after their initial creation. On the other hand, lists are *mutable*, or able to be modified.

6.1 Creating Lists

The easiest way to create a list is by putting the list elements inside of square brackets. Below, we create a list containing the masses of all of the planets in our solar system.

```
mass = [1.8986e27,5.6846e26,10.243e25,8.6810e25,5.9736e24,4.8685e24,6.4185e23,3.3022e23]
```

Note that square brackets (`[]`) must be used when creating the list. If you accidentally use parenthesis (`()`)[1] or curly brackets (`{}`)[2] you'll end up creating something other than a list.

Lists can contain any type of data and the type of data doesn't have to be the same for all of the elements. Below, we create a list of the electron configurations for the first 10 elements on the periodic table.

```
electrons = ["1s1","1s2", "1s2-2s1","1s2-2s2","1s2-2s2-2p1","1s2-2s2-2p2","1s2-2s2-2p3","1
```

Lists can contain mixed data types. Below we construct a list of the electrical conductivities of three metals.

```
conductivity = ["Gold",4.10e7,"Copper",5.96e7,"Aluminum",3.5e7]
```

If the elements of a list are also lists, we call it a *nested* list.

[1] Use parenthesis to create a tuple, which is just like a list but cannot be modified.

[2] Use curly brackets to create a dictionary, which is like a list but can be indexed on any data type, not just integers.

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

6.2 Indexing and Slicing Lists

Indexing is used to access individual elements of a list, and it is similar to indexing strings. The index is the position of the desired element in the list and the *index numbering starts at zero*. Accessing an element of a list is done by placing the numerical index of the element we want in square brackets behind the list name. For example, if we want the electron configuration of the first element in our list from above, we use `electrons[0]` and the electron configuration for the second element would be `electrons[1]` and so on. Just as with string, negative indices can be used to access list elements counting from the back of the list forward.

```
electrons = ["1s1","1s2", "1s2-2s1","1s2-2s2","1s2-2s2-2p1","1s2-2s2-2p2","1s2-2s2-2p3","1

electrons[1]
electrons[-2]
electrons[5]
```

To Do:

1. Predict the output for each of the indexes performed in the cell above.
2. Use print statements to check your answers.
3. Discuss any misunderstandings with a peer.

Since lists are *mutable*, we can modify the value of an element in a list using the `=` operator.

```
conductivity = ["Gold",4.10e7,"Copper",5.96e7,"Aluminum",3.5e7]

conductivity[1] = 4.15e7
conductivity
```

```
['Gold', 41500000.0, 'Copper', 59600000.0, 'Aluminum', 35000000.0]
```

To Do:

Use a print statement to verify that the `conductivity` list was indeed modified as expected.

Multiple list elements can be retrieved at once (called *slicing*) by including the start and stop indices separated by a colon: `[start:stop:step]`. A convention that occurs throughout python is that the first index is included in the slice but the second is *not*. (i.e. `[included:`

`excluded: step`]) Default values for the start location, stop location, and step sizes will be used if these values are omitted. Below we give some examples of slicing.

```
electrons = ["1s1","1s2", "1s2-2s1","1s2-2s2","1s2-2s2-2p1","1s2-2s2-2p2","1s2-2s2-2p3","1

electrons[1:]
electrons[1:3]
electrons[:3]
electrons[1:8:2]
electrons[5:2:-1]
```

To Do:

1. Predict the output for the five slices in the cell above.
2. Use print statements to check your answers.
3. Discuss any misunderstandings with a peer.

6.2.1 Tests for Inclusion

Just as with strings, the `in` operator can be used with lists to determine if a list element is present. Suppose you have a list of all the known radioactive elements on the periodic table and you'd like to know if Iridium is in the list. The `in` statement let's us quickly test to see if it is in the list. (see example below)

```
radioactiveElements = ["Technetium","Promethium","Polonium","Astatine","Radon","Francium",

"Iridium" in radioactiveElements
```

```
False
```

The `in` operator will work with numerical data as well.

```
numbers = [5,6,3,1,2]

4 in numbers
```

```
False
```

6.2.2 List Methods

Lists have a collection of methods (or functions) for accomplishing routine tasks. Some of the more common list methods are given below. All of the methods given will modify the original list (except `copy()`). As a reminder, methods only work on the object type that they were designed for (lists in this case) and they are called by appending the method name to the variable you want it to operate on. (i.e. `myList.clear()`)

Method	Description
`append(element)`	Adds a single element to the end of the list.
`clear()`	Removes all elements from a list.
`copy()`	Creates an independent copy of the list.
`count(element)`	Counts the number of occurences of `element` in the list.
`extend(elements)`	Adds multiple elements to the end of the list.
`index(element)`	Returns the index of the *first occurrence* of `element`.
`insert(index,element)`	Inserts the given `element` at the specified `index`.
`pop(index)`	Removes and returns the element given at `index`. If no index is provided, it defaults to the last element.
`remove(element)`	Removes the first occurrence of `element` in the list.
`reverse()`	Reverses the order of the entire list.
`sort()`	Sorts the list in place.[3]

```
a = [5,6,23,2,2,6]
a.sort()
```

To Do:

1. In the cell below, you will find a list of ...

6.2.3 Built-in functions for Lists

Python has several built-in functions that will work with lists. Functions are called by placing the arguments to the function in parenthesis and prepending the name of the function to the parenthesis. Here are a few common functions that are used with lists:

Method	Description
`len(list)`	Returns the number of elements in the list.

[3] It modifies the original list. In contrast, the function `sorted()` will leave the original list unchanged.

Method	Description
`max(list)`	Finds the largest element in the list.
`min(list)`	Finds the smallest element in the list.
`sum(list)`	Returns the sum of the elements in the list.[4]
`listOne + listTwo`	Joins `listOne` with `listTwo` to form one list.
`2 * list`	Repeat `list` and concenate to itself, making the list twice as long.
`zip(listOne,listTwo)`	Zips two lists together. (Creates `[a1,b1,a2,b2...]` from `[a1,a2,a3...]` and `[b1,b2,b3...]`

```
a = ["l","j","n"]
b = ["r","s","t"]
a + b
#a=[5,6,2]
#min(a)
#print("l" + "j"+"n")
```

```
['l', 'j', 'n', 'r', 's', 't']
```

6.2.4 The `range` function

Often it will be necessary to generate a (possibly very long) list of integers. Instead of constructing the list by typing these numbers one by one, Python has a built-in function called `range()` that will do it for you. The `range()` function requires at least one argument to tell it how high the range should be.

```
a = range(10)  # Generate a list of integers up to 10
a
```

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

This output probably wasn't what you expected. Instead of generating the full list of numbers, Python generates a `range` object that stands in place of it because it requires less memory. (Consider what might happen to your memory if you did `range(1000000000000)`.) To force Python to generate the list, you can convert the `range` object using the `list()` function.

[4] Only works if the list contains only ints and floats

```
a = range(10)  # Generate a list of integers up to 10
list(a)
```

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

The `range` function can be called with up to three arguments: `range(start,end,stepsize)`. Consistent with indexing, the range includes the start value and excludes the end value. Below we generate a list of integers starting at 5, ending at 100 with a stepsize of 5.

```
myList = range(5,100,5)
list(myList)
```

```
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
```

6.3 Tuples

Tuples are another object type similar to lists except that they are *immutable*... that is to say, they cannont be modified once created. They look similar to lists except that they are created using parenthesis instead of brackets. Because you can't change the elements of a tuple, they are often used so that you don't inadvertently modify (and lose) critical data. You can think of it as locking a file on your computer to avoid indavertently modifying it and losing the original content.

Below is a tuple containing... Since there will be no need to change these values, it makes sense to store them in a tuple. Indexing and slicing work exactly the same as with lists and strings, so we can still use the values inside of a tuple to perfrom simple calculations. There are only two methods associated with tuples: `count(element)` and `index(element)`. Their usage is identical to the list methods.

```
energies = (5,7,2,4,3,2,3)
energies.index(7)
len(energies)
```

```
7
```

7 Loops

Loops allow programs to rerun the same block of code multiple times. This is important because there are often

7.1 `for` loops

The `for` loop is probably the most common loop you will encounter and is a good choice when you know beforehand exactly what things you want to loop over. Here is an example of of `for` loop that is used to add up the elements of a list.

```
thesum = 0
for i in [3,2,1,9.9]:
    thesum = thesum + i
```

This would be equivalent to the following code:

```
thesum = 0

thesum = thesum + 3
thesum = thesum + 2
thesum = thesum + 1
thesum = thesum + 9.9
```

which isn't that much longer than using a loop. However, as the list gets longer and/or the mathematical operations being performed get more complex the second method would get unreasonably long. The correct language is to say that we are *iterating* over the list `[3,2,1,9.9]`. This means that the loop variable (`i` in this case but you can choose it to be whatever you want) gets assigned the values of the list elements, one by one, until it reaches the end of the list. You can use `for` loops to iterate over any multi-element object like lists or tuples. Python uses indentation to indicate where the loop ends. In this case there was only one statement inside to loop, but if you wanted more than one each line should be indented.

You can iterate over `range` objects and strings using `for` loops.

```
for i in ['Physics', 'is','so','fun']: # Iterate over a list of strings
    print(i)
```

```
Physics
is
so
fun
```

```
for i in range(5,50,3):  #Generates a list from 5 -> 50 with a step size of 3
    print(i)
```

```
5
8
11
14
17
20
23
26
29
32
35
38
41
44
47
```

These examples are so simple that you might wonder when a loop might actually be useful to you. Let's see if we can build a loop to calculate the following sum

$$\sum_{n=1}^{1000} \frac{1}{n^2} \tag{7.1}$$

```
theSum = 0
for n in range(1,1000):
    theSum = theSum + 1/n**2
print(theSum)
```

```
1.6439335666815615
```

Here, `n` is being assigned the values `1,2,3,4....1000`, one by one, until it gets all the way to 1000. Each time through the loop, `n` is different and the expression `1/n**2` evaluates to a new value. The variable `theSum` is updated each time through to be the running total of all calculations performed thus far. Here's another example of a loop used to calculate the value of 20!:

```
theProduct = 1
for n in range(1,21):
    theProduct = theProduct * n #Multiply theProduct by n
print(theProduct)
```

```
2432902008176640000
```

Remember that the range function creates a list starting at 1, going up to 21 but not including it. The math library has a function called `factorial` that does the same thing. Let's use it to check our answer:

```
from math import factorial
factorial(20)
```

```
2432902008176640000
```

7.1.1 Boolean Logic Inside Loops

Often when using loops, we only want a block of code to execute when some condition is satisfied. We can use boolean logic inside of the loop to accomplish this. For example, let's write a loop to compute the following sum:

$$\sum_{\frac{n}{5}\in \text{ Int and } \frac{n}{3}\in \text{ Int}} \frac{1}{n^2}$$

which is similar to the one we did above, but this time we only want to include terms where n is a perfect multiple of both 5 and 3. To check to see if `n` is a perfect multiple of a number we can calculate the modulo (remainder after division) using the `%` operator and check that it is equal to zero.

```
theSum = 0
for n in range(1,1000):
    if n % 5 == 0 and n % 3 == 0:
        theSum = theSum + 1/n**2
print(theSum)
```

```
0.007243985583159138
```

7.2 `while` Loops

Logic can be combined with loops using something called a `while` loop. A `while` loop is a good choice when you don't know beforehand exactly how many iterations of the loop will be executed but rather want the loop to continue to execute until some condition is met. As an example, notice that in in equation (Equation 7.1), the terms in the sum get progressively smaller as n gets bigger. It doesn't make sense to continue adding to the sum once the terms get very small. Let's compute this sum by looping until the fraction $\frac{1}{n^2}$ become smaller than 1×10^{-10}.

```
term = 1  # Load the first term in the sum
s = term  # Initialize the sum
n = 1     # Set a counter
while term > 1e-10:  # Loop while term is bigger than 1e-10
    n = n +  1        #Add 1 to n so that it will count: 2,3,4,5
    term = 1./n**2    # Calculate the next term to add
    s = s +  term     # Add 1/n^2 to the running total
```

This loop will continue to execute until `term<1e-10`. Note that unlike the `for` loop, here you have to do your own counting if you need to know how many iterations have been performed. Be careful about what value `n` starts at and when it is incremented (`n = n + 1`). Also notice that `term` must be assigned prior to the start of the loop. If it wasn't the loop's first logical test would fail and the loop wouldn't execute at all.

`while` loops should be used with caution because you can easily write a *faulty termination condition* and inadveratently write a loop that runs forever. This happens because your termination condition was never met. An example of this is given below.

Warning: Do not execute the code block below!!

```
x = 0

while x != 10:
```

```
    x = x + 3
print("Done")
```

The loop above is intended to end after a few interations when the value of x is equal to 10. However, closer inspection reveals that the value of x will never be equal to 10. After the first iteration x is equal to 3, then 6,9,12,15 and so on… but never 10. This loop will run forever because the termination condition is never met (`x != 10` never produces a `False`)!! If you choose to use a `while` loop, triple check your termination condition to make sure you haven't made a mental error. Avoiding the use of `!=` or `==` in your termination condition will help too. Use `<=` or `>=` instead.

7.3 `continue`, `break`, and `pass` Commands

The `continue`, `break`, and `pass` commands are used to control the flow of code execution in loops. Here is a description of their usage:

Operator	Description
`break`	Exits a `for`/`while` loop.
`continue`	Skips the remaining loop block and begins the next iteration.
`pass`	No action; code contiues on

The `break` statement is useful when you want to completely stop a loop early. Here is our sum loop rewritten with a `break` statement added to stop the loop after 1000 iterations.

```
term = 1  # Load the first term in the sum
s = term  # Initialize the sum
n = 1     # Set a counter
while term > 1e-10:  # Loop while term is bigger than 1e-10
    n +=  1        #Add 1 to n so that it will count: 2,3,4,5
    term = 1./n**2    # Calculate the next term to add
    s += term     # Add 1/n^2 to the running total
    if n > 1000:
        print('This is taking too long. I''m outta here...')
        break
```

```
This is taking too long. Im outta here...
```

The `continue` statement is similar to `break` except that instead of stopping the loop, it only stops the current iteration of the loop. All code below the `continue` statement will be skipped

and the next iteration will begin. For example, if you wanted to do the sum from equation ?? but only include those terms for which n is a multiple of 3, it could be done like this:

```
term = 1  # Load the first term in the sum
s = term  # Initialize the sum
n = 1     # Set a counter
while term > 1e-10:  # Loop while term is bigger than 1e-10
    n +=  1        #Add 1 to n so that it will count: 2,3,4,5
    if n % 3 != 0:
        continue
    term = 1./n**2    # Calculate the next term to add
    s += term     # Add 1/n^2 to the running total
```

Now, when the value of `n` is not a multiple of 3, the sum will not be updated and the associated terms are effectively skipped.

Finally, the `pass` statement does nothing. Seriously!! It is merely a place holder for code that has not bee written yet. Usually, you'll use the `pass` statement to run and test code without errors due to missing code.

8 Functions

We have already been using functions here and there but in this chapter we will introduce them formally and get into the details. A function encapsulates a block of code designed to perform a specific task or set of tasks. To perform the task correctly, most functions require that you provide some information (called arguments) when you call them. To call a function you type the name of the function followed by the needed arguments enclosed in parenthesis (`()`).

```
functionName(argument1, argument2, argument3...)
```

The **number and type** of arguments allowed is different for every function. As a first example, let's consider the `print` function, which is the simplest (and most familiar) function that we have used so far.

```
print("print is a function")  #Single string argument
print("print", "is", "a", "function") # Multiple string arguments
print(5) # Single integer argument
print(5,4) # Multiple integer argument
print([5,4,2,5])  # Single list argument
print() # No arguments
```

```
print is a function
print is a function
5
5 4
[5, 4, 2, 5]
```

The `print` function is pretty flexible in what it allows for arguments; the arguments can be any type of data (strings, ints, floats, booleans, and even lists) and the number of arguments can be as large as you want. Most functions are a little more strict on what they allow their arguments to be. For example, the `factorial` function only allows one argument and that argument must be an integer.[1] If you attempt to call the factorial function with a float argument or with more than one argument, the result will be an error.

[1]The factorial of a float can be calculated, but the math library is not equipped to handle it.

```
from math import factorial
a = factorial(5.54)
b = factorial(5,3)
```

To Do:

1. Run the code above and take note of the error message that results.

2. Comment out the definition of `a` so you can also see the error message for the definition of `b`.

Python functions generally fall into three groups: functions that come standard with Python (called native functions), functions that you can import into Python, and functions that you write yourself.

8.1 Native Functions

There are a few functions that are always ready to go whenever you run Python. They are included with the programming language. We call these functions native functions. You have already been using some of them, like these

```
myList = [5,6,2,1]
a = len(myList)  # 'len' function is native.

b = float(5) # 'float' function is native.

c = str(67.3)  # 'str' function is native.
```

The `len`, `float` and `str` functions are all native and they all take **a single argument**. Othere native function have been mentioned in previous chapters and others will be mentiond in the future.

8.2 Imported Functions

Many times, you will need to go beyond what Python can do by itself[2]. However, that doesn't mean you have to create everything you need to do from scratch. Most likely, the function that you need has already been coded. Somebody else created the function and made it available to anyone who wants it. Groups of functions that perform similar tasks are typically bundled

[2]For example, Python does not include sin() or cos() as Native functions.

together into libraries ready to be imported so that the functions that they contain can be used.

In order to use use a function correctly, you'll need to know what information(arguments) the function expects you to give it and what information the function intends to return to you as a result. This information can be found in the library's documentation. Most libraries have great documentation with lists of the included functions, what the functions do, the expected arguments, and examples on how to use the most common ones. You can usually find the library documentation by searching the internet for the library's name plus "Python documentation".

Providing a complete list of all available libraries and function is not really the purpose of this book. Instead, we'll illustrate how to import functions and use them. As you use Python more and more you should get in the habit of searching out the appropriate library to accomplish the task at hand. When faced with a task to accomplish, your first thought should be, " I'll bet somebody has already done that. I'm going to try to find that library."

Functions are imported using the `import` statement. You've already seen how to perform very simple mathematical calculations (5/6,84, etc..), but for more complex mathematical calculations like $\sin(\frac{\pi}{2})$ or $e^{2.5}$, you'll need to import these functions from a library.

```
import math

a = math.sqrt(5.2)
b = math.sin(math.pi)
c = math.e**2.5
```

The `math.` before each function is equivalent to telling Python "Use the `sqrt()` function that you will find in the `math` book I told you to grab." If you just type

```
sqrt(5.2)
```

Python won't know where to find the `sqrt` function and an error message will result. Sometimes the name of the module can be long and typing `module.` every time you want to use one of it's functions can be cumbersome. One way around this is to rename the module to a shorter name using the `as` statement.

```
import math as mt

a = mt.sqrt(5.2)
b = mt.sin(mt.pi)
c = mt.e**2.5
```

Instead of importing an entire module, you can import only a selection of functions from that module using the `from` statement. This can make your code even more succinct by eliminating the `module.` prefix altogether. The trade-off is that it won't be as clear which function belongs to which module.

```
from math  import sqrt, sin, pi, e

a = sqrt(5.2)
b = sin(pi)
c = e**2.5
```

All of the functions belonging to a module can be imported at once using `*`.

```
from math import *

a = sqrt(5.2)
b = sin(pi)
c = e**2.5
```

8.3 User-defined Functions

After having programmed for a while, you will notice that certain tasks get repeated frequently. For example, maybe in your research project you need to calculate the force exerted on an atom due to many other nearby atoms. You could copy and paste your force-calculation code every time it was needed, but that would likely result in lots of extra code and become very cumbersome to work with. You can avoid this by creating your own function to calculate the force between any two atoms. Then, every time you need another force calculation, you simple call the function again. You only write the force-calculation part of the code once and then you execute it as many times as you need to.

To create your own function, you first need to name the function. The name should be descriptive of what it does and makes sense to you and anyone else who might use it. The first line of a function definition starts with the `def` statement (short for definition) followed by the name of the function with whatever information, called *arguments*, that needs to be fed into the function enclosed in parenthesis. **The last character in this line must be a colon.** Everything inside the function is indented four spaces and placed directly below the first line.

```
def functionName(arg1,arg2,arg3):
    # Body of Function
    # Body of Function
```

```
    # Body of Function
```

As an example, let's construct a function that calculates the distance between two atoms. The function will need to know the location of each atom, which means that there should be two arguments: the xyz coordinates of both atoms passed as a pair of lists or tuples.

```
import math

def distance(coords1,coords2):

    dx = coords1[0] - coords2[0]
    dy = coords1[1] - coords2[1]
    dz = coords1[2] - coords2[2]
    d = math.sqrt(dx**2 + dy**2 + dz**2)
    print(f"The distance is {d:5.4f}.")

distance([1,2,3],[4,5,6])

distance([4.2,9.6,4.8],[2.9,2.6,3.4])
```

```
The distance is 5.1962.
The distance is 7.2560.
```

This function works just fine with integers or floats for the coordinates.

8.3.1 The `return` statement

The `distance` function prints out the value for the distance, but what if we want to use this distance in a subsequent calculation? Maybe we want to calculate the average distance between several pairs of atoms. We can instruct the function to return the final distance using the `return` statement. If the arguments to the function are the inputs, the `return` statement specifies what the output is. Let's modify the function above to include a `return` statement.

```
import math
def distance(coords1,coords2):
    dx = coords1[0] - coords2[0]
    dy = coords1[1] - coords2[1]
    dz = coords1[2] - coords2[2]
    d = math.sqrt(dx**2 + dy**2 + dz**2)
    return d
```

```
distOne = distance([1,2,3],[4,5,6])

distTwo = distance([4.2,9.6,4.8],[2.9,2.6,3.4])

averageDistance = (distOne + distTwo)/2

print(f"The average distance is {averageDistance:5.4f}.")
```

```
The average distance is 6.2261.
```

8.3.2 Local vs Global Variable Scope

Variables created inside of a function have *local scope*. This means that they are not accessible outside of the function. In our `distance` function the variables `dx`,`dy`,`dz`, and `d` were all local variables that are used inside the function but have no value outside of it. This is convenient because we don't have to worry about overwriting a variable or using it twice. If someone sends you a function and you want to use it in your code, you don't have to worry about what variable he/she chose to use inside his function; they won't affect your code at all.

```
def distance(coords1,coords2):
    dx = coords1[0] - coords2[0]
    dy = coords1[1] - coords2[1]
    dz = coords1[2] - coords2[2]
    d = math.sqrt(dx**2 + dy**2 + dz**2)
    return d

distOne = distance([1,2,3],[4,5,6])

print(dx)  # There is no value associated with this variable outside of the function.
```

The down side to all of this is that you don't have access to function variables unless you pass them out of the function using the `return` statement.

Any variables defined outside of a function is called a *global variable*, which means that Python remembers these assignments from anywere in your code including inside of functions. Using globabl variables with the intention to use them inside of functions is usually considered bad form and confusing and is discouraged. One notable exception to this rule are physical constants like $g = 9.8$ m/s^2 (acceleration due to gravity on Earth) or $k_B = 1.38 \times 10^{-23}$ (Boltzmann's constant which is used heavily in thermodynamics) because these values will never change and may be used repeatedly. Generally speaking every variable that is used in

a function ought to be either i) passed in as an argument or ii) defined inside of the function. Below is an example of an appropriate use of a global variable.

```
def myFunction(a,b):
    c=a+g # <--- Notice the reference to 'g' here
    d = 3.0 * c
    f = 5.0 * d**4
    return f

#The variable below are global variables.
r = 10
t = 15
g = 9.8         #<--- g defined to be a global variable
result = myFunction(r,t)
```

8.3.3 Positional vs. Keyword Arguments

The function arguments we have been using so far are called *positional arguments* because they are required to be in a specific position inside the parenthesis. To see what I mean consider the example below.

```
def example(a,b):
    return a**b


resultOne = example(5,2)
resultTwo = example(2,5)

print(resultOne, resultTwo)
```

```
25 32
```

In the first call to `example` the local variable `a` gets assigned to be 5 and the local variable `b` gets assigned 2. In the second call the order of the arguments is switched and the subsequent assignments to `a` and `b` switch with it. This produces a different result from the function. Positional arguments are very common but the user must know what information goes where when calling the function.[3]

The other type of argument is the *keyword argument*. These arguments are attached to a keyword inside of the parenthesis. The advantage of a keyword argument is that the user does

[3]This is another reason why you want to choose meaningful variable names for your arguments.

not need to be concerned about the location of the argument as long as it has the proper label.

```
def example(a=1,b=2):
    return a**b


resultOne = example(a=5,b=2)
resultTwo = example(b=2,a=5)

print(resultOne, resultTwo)
```

```
25 25
```

Another advantage to using keyword arguments is that a default value can be coded into the function. This means that we can call the function with some arguments missing and default values will be used for them. In the example above, the default value of `b` is 2 and if the function is called without specifiying a value for that argument, the function will proceed as usual using the default value for `b`.

```
def example(a=1,b=2):
    return a**b


result = example(a=5)

print(result)
```

```
25
```

Part II

Python - Intermediate Concepts

9 The `numpy` module

Numpy (pronounced "num"-"pie") is a popular Python library that is heavily used in the scientific/mathematical community. So much so that numpy is typically included as part of the standard bundle of libraries that comes with your Python installation. The functions inside numpy will allow you to solve problems with less effort and will produce faster-executing code.

9.1 Numpy Arrays

You are already familiar with Python lists but may not have noticed that they are not suitable for mathematical calculations. For example, attempting to multiply a list by a scalar or evaluate a mathematical function like sin() on a list will *not* produce a mathematical result or may produce an error. For example, consider the following code.

```
myList = [4,5,7]

newList = 2 * myList
print(newList)
```

```
[4, 5, 7, 4, 5, 7]
```

You probably expected `newList` to be `[8,10,14]` but multiplying a list by a number doesn't do that. Instead it repeats the list and concatenates it to itself. To multiply each element of a list by a number you must use a `for` loop.

```
myList = [4,5,7]

newList = []
for i in myList:
    newList.append(i* 2)

print(newList)
```

```
[8, 10, 14]
```

but this seems overly cumbersome for such a simple task. Numpy *ndarrays* (short for n-dimensional arrays) or just *arrays* make this task much simpler. Arrays are similar to lists or nested lists except that mathematical operations and `numpy` functions (but not `math` functions) automatically propogate to each element instead of requiring a `for` loop to iterate over it. Because of their power and convenience, arrays are the default object type for any operation performed with NumPy.

9.2 Array Creation

9.2.1 Type Conversion from List

You can create an array from a list using numpy's `array` function. The list that is to be converted is the argument to the `array` function. Mathematical operations can then be performed on the array and that operation will propogate through to all of the elements.

```
from numpy import array

myArray = array([4,5,7])

newArray = 2 * myArray

print(newArray)
```

```
[ 8 10 14]
```

Nested lists, or lists that contain lists as their elements, can be converted to multi-dimensional arrays using the `array` function.

```
from numpy import array
myArray = array([[1,2,3],[4,5,6]])

print(myArray)
```

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

9.2.2 The `arange` and `linspace` Functions

Numpy has some sequence-generating functions that generate arrays by specifying start, stop, and stepsize values similar to the way `range` generates a list. The two most common ones are `arange` and `linspace`. The `arange` function behaves very similar to the native Python `range` function with a few notable exceptions:

1. `arange` produces an array whereas `range` produces a list.
2. The step size for `arange` does not need to be an integer.
3. `range` produces an iterator and `arange` generates a sequence of values immediately.

The arguments to `arange` are similar to `range`

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

The `linspace` function is related to the `arange` function except that instead of specifying the step size of the sequence, the sequence is generated based on the number of equally-spaced points in the given span of numbers. Additionally, `arange` excludes the stop value while `linspace` includes it. The difference between these two functions is subtle and the use of one over the other often comes down to user preference or convenience.

```
linspace(start,stop,number of points)
```

Below is an example that shows the usage of `linspace` and `arange`.

```
from numpy import linspace,arange

myArray = linspace(0,10,20)
myArray2 = arange(0,10,0.5)
print(myArray)
print(myArray2)
```

```
[ 0.          0.52631579  1.05263158  1.57894737  2.10526316  2.63157895
  3.15789474  3.68421053  4.21052632  4.73684211  5.26315789  5.78947368
  6.31578947  6.84210526  7.36842105  7.89473684  8.42105263  8.94736842
  9.47368421 10.        ]
[0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5.  5.5 6.  6.5 7.  7.5 8.  8.5
 9.  9.5]
```

When using `linspace` you may still want to know what the sequence spacing is. You can request that `linspace` provide this information by adding the optional argument `retstep = True` to the argument list. With this addition, linspace not only returns the sequence to you, but also the stepsize.

```
from numpy import linspace,arange

myArray,mydx = linspace(0,10,20,retstep= True)
print(mydx)
print(myArray)
```

```
0.5263157894736842
[ 0.          0.52631579  1.05263158  1.57894737  2.10526316  2.63157895
  3.15789474  3.68421053  4.21052632  4.73684211  5.26315789  5.78947368
  6.31578947  6.84210526  7.36842105  7.89473684  8.42105263  8.94736842
  9.47368421 10.        ]
```

Two other useful functions for generating arrays are `zeros` and `ones` which generate arrays populated with exculsively ones or zeros. The functions require shape arguments as a tuple or list to specify the shape of the array.

```
zeros((rows,columns))
```

If the array to be created is only one dimensional, the argument can be a single number instead of a tuple.

```
zeros(n)
```

```
from numpy import zeros,ones

myArray = zeros([3,4])
myArray2 = ones(5)
print(myArray)
print(myArray2)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[1. 1. 1. 1. 1.]
```

Arrays of any constant (not just one or zero) can then be easily generated by performing the needed math on the original array.

```
from numpy import zeros,ones

myArray = zeros([3,4]) + 5
myArray2 = ones(5) * 12
print(myArray)
print(myArray2)
```

```
[[5. 5. 5. 5.]
 [5. 5. 5. 5.]
 [5. 5. 5. 5.]]
[12. 12. 12. 12. 12.]
```

9.2.3 Arrays from Functions

A third approach is to generate an array from a function using the `fromfunction` function which generates an array of values using the array indices as the inputs. Ths function requires two arguments: the name of the function being used and the shape of the array being generated.

```
fromfunction(function, shape)
```

Let's make a 3 x 3 array where each element is the product of the row and column indices:

```
from numpy import fromfunction

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

myArray = fromfunction(prod,(3,3))
print(myArray)
```

```
[[0. 0. 0.]
 [0. 1. 2.]
 [0. 2. 4.]]
```

The table below gives a summary of useful functions for creating numpy arrays. The required arguments are also described.

Table 9.1: Common functions for generating arrays

Method	Description
`linspace(start,stop,n)`	Returns an array of `n` evenly-spaced points begining at `start` and ending at `stop`.
`arange(start,stop,dx)`	Returns an array begining at `start`,ending at `stop` with a step size of `dx`.
`empty(dims)`	Returns an empty array with dimensions `dim`.
`zeros(dims)`	Returns an array of zeros with dimensions `dim`.
`ones(dims)`	Returns an array of ones with dimensions `dim`.
`zeros_like(arr)`	Returns an array of zeros with dimensions that match the dimensions of `arr`.
`fromfunction(function,dims)`	Returns an array of numbers generated by evaluating `function` on the indices of an array with dimensions `dims`.
`copy(arr)`	Creates a copy of array `arr`.
`genfromtext(file)`	Reads `file` and loads the text into an array (file must only contain numbers).

9.3 Accessing and Slicing Arrays

Accessing and slicing arrays can be done in exactly the same way as is done with lists. However, there is some additional functionality for accessing and slicing arrays that do not apply to lists.

9.3.1 One-dimensional Arrays

Elements from a one-dimensional array can be extracted using square brackets (`[]`) just like we have done with lists.

```
from numpy import array

myArray = array([3,4,7,8])
print(myArray[2])
```

7

9.3.2 Multi-dimensional Arrays

Multi-dimensional array can be indexed in a similar fashion to nested lists, but because we often encounter multi-dimensional arrays there is a shortcut that makes the syntax simpler and more convenient. Let's consider a two-dimensional array as an example. To access the entire second row of the array, provide the row index in square brackets just as with one-dimesional arrays.

```
from numpy import array
myArray = array([[1,2,3],[4,5,6], [7,8,9]])

print(myArray[1])
```

```
[4 5 6]
```

To access the second element *in the second row*, we can add another set of square brackets with the appropriate index inside, just as we did with nested lists. However, for convenience the second set of square brackets can be omitted and the row and column indices can be placed next to each other and separted by a comma.

```
array_name[row,column]
```

```
from numpy import array
myArray = array([[1,2,3],[4,5,6], [7,8,9]])

print(myArray[1][1])  # This works, but is a bit hard on the eyes
print(myArray[1,1])  # This works also and is easier to look at.
```

```
5
5
```

9.3.3 Accessing Multiple Elements

Multiple elements of an array can be accessed using a list for the index instead of a single number.

```
from numpy import array

myArray = array([1,2,3,4,5,6,7,8,9,10])
```

```
print(myArray[2])  # Extract element 2
print(myArray[ [3,6,9] ])  # Extract elements 3, 6, and 9.
```

```
3
[ 4  7 10]
```

This can even be done with multi-dimensional arrays. If the index is a single list, the corresponding rows will be extract. If the corresponding list of columns is added to the index list, individual elements will be extracted.

```
array_name[[rows]]  # Access set of rows
```

```
array_name[[rows], [columns]]  # Access set of elements
```

```
from numpy import array
myArray = array([[1,2,3],[4,5,6], [7,8,9]])

print(myArray[[0,1]]) # Extract rows 0 and 1.

print(myArray[[1,2,0],[0,2,2]])   # Extract elements (1,0), (2,2), and (0,2)
```

```
[[1 2 3]
 [4 5 6]]
[4 9 3]
```

9.4 Slicing Arrays

9.4.1 Multi-dimensional Arrays

We've already shown you how to slice a list using the : operator. The same can be done with arrays. However, for 2D (and higher) arrays the slicing is more powerful (intuitive). It can be helpful to visualize an array as a matrix, even if it is not being treated that way Mathematically. For example, let's say that you define the following array:

```
from numpy import array
myArray = array([[1,2,3],[4,5,6], [7,8,9]])
```

which can be visualized as the following matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

To slice out the following 2 x 2 sub matrix:

$$\begin{pmatrix} 5 & 6 \\ 8 & 9 \end{pmatrix}$$

we could do

```
from numpy import array
myArray = array([[1,2,3],[4,5,6], [7,8,9]])

print(myArray[1:3,1:3])
```

```
[[5 6]
 [8 9]]
```

To include all of the elements in a given dimension, use the : alone with no numbers surrounding it.

```
from numpy import array
myArray = array([[1,2,3],[4,5,6], [7,8,9]])

print(myArray[:,1:3])  # Extract all rows with columns 1 and 2
```

```
[[2 3]
 [5 6]
 [8 9]]
```

9.4.2 Boolean Slicing

Boolean operations can be evaluated on arrays to produce corrsponding arrays of booleans. The boolean array can then be used to index the original array and extract elements that meet some criteria.

```
from numpy import array

a = array([1,2,3,4,5,6])

boolArray = a > 2

print(boolArray)

print(a[boolArray])
```

```
[False False  True  True  True  True]
[3 4 5 6]
```

This also works on multi-dimensional arrays although the result is always one-dimensional regardless of the shape of the original array.

```
from numpy import array
myArray = array([[1,2,3],[4,5,6], [7,8,9]])

print(myArray[myArray>2]) # Extract elements that are greater than 2.
```

```
[3 4 5 6 7 8 9]
```

9.5 Vectorization and Broadcasting

A major advantage of numpy arrays over lists is that operations *vectorize* across the arrays. This means that mathematical operations propogate through the array instead of requiring a `for` loop. This speeds up the calculation and makes code easier to write and read. Simple mathematical operations like adding, subtracting, etc can be performed on arrays as you would expect and the operation propogates through to all elements.

```
from numpy import array

a = array([1,2,3])
b = array([4,5,6])

c = a + b
d = a**2
e = 2 * b
```

```
f = 2/b
g = a * b

print(c,d,e,f,g)
```

```
[5 7 9] [1 4 9] [ 8 10 12] [0.5        0.4        0.33333333] [ 4 10 18]
```

All of the common mathematical operations that you learned for numbers now apply to arrays. Cool!

9.5.1 Numpy Functions

The numpy library has a massive collection of vectorized mathematical functions and these functions should be used instead of similar functions from other libraries that are not vectorized (like `math`).

```
from numpy import array
from numpy import sqrt as nsqrt
from math import sqrt as mathsqrt

squares = array([1,4,9,16,25])

print(nsqrt(squares))
#print(mathsqrt(squares))  #This will fail because it wasn't a numpy function.
```

```
[1. 2. 3. 4. 5.]
```

9.5.2 Arrays of same Dimensions

If a mathematical operation is performed between two arrays of the same dimensions, the mathematical operation is performed between corresponding elements in the two arrays. For example, if two 2 x 2 arrays are added together, element (0,0) of the first array gets added to the corresponding element in the second and so forth for all elements:

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

```
from numpy import array

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

c = a + b  # Add the elements of the arrays together.
d = a * b  # Multiply the elements of the arrays together.

print(c)
print(d)
```

```
[[ 6  8]
 [10 12]]
[[ 5 12]
 [21 32]]
```

9.5.3 Arrays of Different Dimensions

When a mathematical operation between two arrays of different dimensions is attempted, Python has to figure out how to make them have the same shape before performing the operation. *Broadcasting* refers to the set of rules used for operations like this. To handled arrays with different dimensions, NumPy pads or clones the array with fewer dimensions to make it have the same dimensions as the larger array. For example, what would happen if you attempted this operation:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 2 & 2 \end{pmatrix}$$

One array is 2 x 2 and the other is 1 x 2. Before the addition can take place, NumPy clones the smaller array and repeats it until it has the same size as the bigger array.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$$

```
from numpy import array

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

```
print(c)
```

```
[[3 4]
 [5 6]]
```

There are some cases where NumPy simply cannot figure out how to broadcast one of the arrays appropriately and an error results. When broadcasting, NumPy must verify that all dimensions are compatible with each other. Two dimensions are compatible when i) they are equal or ii) one of the dimensions is 1. For example, if we tried to perform the following mathematical operation, broadcasting would fail because the first dimension of the first array is 2 and the first dimension of the second array is 3.

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \end{pmatrix} + \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}$$

```
from numpy import array

a = array([[1,2,3],[4,5,6]])
b = array([[1,1,1],[2,2,2],[3,3,3]])
c = a + b
```

but if we attempted

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix} + \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}$$

the operation would succeed because the first dimension of the first array is 1 and the second dimension of both arrays are 3.

9.5.4 Vectorizing user-defined functions

Standard Python functions are often designed to perform a single calculation rather than iterate over a list to perform many calculations. For example, here is a function to calculate the average acceleration of an object given its final velocity and time of travel.

```
def accel(velocity, time):
    return velocity / time
```

```
print(accel(52.6,5.6))
```

```
9.392857142857144
```

Now what if I have a list of many times that I'd like to feed into this function and get an acceleration value for each one.

```
def accel(velocity, time):
    return velocity / time


times = [4.6,7.9,3.2,8.5,9.2,4.7]
print(accel(52.6,times))  # Produces an error because you can't divide by a list
```

An error results here because Python does not know how to divide by a list. We can NumPy-ify this function using a function called `vectorize`. The resulting function will behave just like the other functions from the NumPy library, vectorizing across the list of times.

```
from numpy import vectorize
def accel(velocity, time):
    return velocity / time


times = [4.6,7.9,3.2,8.5,9.2,4.7]
vaccel = vectorize(accel)  # Vectorize the function!
print(vaccel(52.6,times))  # Succeeds because NumPy knows how to vectorize.
```

```
[11.43478261  6.65822785 16.4375      6.18823529  5.7173913  11.19148936]
```

Of course, we also could have just converted our times list into an array and used the original function.

```
from numpy import array
def accel(velocity, time):
    return velocity / time


times = array([4.6,7.9,3.2,8.5,9.2,4.7])
print(accel(52.6,times))  # Succeeds because times is an array not a list.
```

```
[11.43478261  6.65822785 16.4375      6.18823529  5.7173913  11.19148936]
```

9.6 Manipulating and Modifying Arrays

A wealth of functions exist to perform routine manipulation tasks on arrays once they are created. Often these tasks will involve changing the number of rows or columns or merging two arrays into one. The *size* and *shape* of an array are the number of elements and dimensions, respectively. These can be determined using the `shape` and `size` methods.

```
from numpy import array

a = array([[1,2,3],[4,5,6]])
print(a.size)
print(a.shape)
```

```
6
(2, 3)
```

9.6.1 Reshaping Arrays

The dimensions of an array can be modified using the `reshape` function. This methods maintains the number of elements and the order of elements but repacks them into a different number of rows and columns. Because the number of elements is maintained, the size of the new array has to be the same as the original. Let's see an example.

```
from numpy import array, reshape

a = array([[1,2,3],[4,5,6]])

b = reshape(a,[3,2])

print(a)
print(b)
```

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

The original array (`a`) was a 2 x 3 and had 6 elements and the reshaped array also has 6 elements but is a 3 x 2. You can start with a one-dimensional array and reshape it to a higher dimensional array.

```
from numpy import linspace, reshape

a = linspace(0,10,12)

b = reshape(a,[3,4])

print(a)
print(b)
```

```
[ 0.          0.90909091  1.81818182  2.72727273  3.63636364  4.54545455
  5.45454545  6.36363636  7.27272727  8.18181818  9.09090909 10.        ]
[[ 0.          0.90909091  1.81818182  2.72727273]
 [ 3.63636364  4.54545455  5.45454545  6.36363636]
 [ 7.27272727  8.18181818  9.09090909 10.        ]]
```

9.6.2 Flattening Arrays

Flattening an array takes a higher-dimensional array and squishes it into a one-dimensional array. You can "flatten" an array with the `flatten` method, but note that `flatten` doesn't actually modify the original array.

```
from numpy import array

a = array([[1,2,3],[4,5,6]])
a.flatten()

print(a)  # 'a' remains unchanged

a = a.flatten() # If you want to change the definition of a, redifine it.
print(a)
```

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

9.6.3 Transposing Arrays

Transposing an array rotates it across the diagonal and can be accomplished with the `transpose` function. There is also a shortcut method for this of `array.T` to accomplish the

same thing but just as with `flatten` it does not modify the original array. (neither does `transpose`)

```
from numpy import array,transpose

a = array([[1,2,3],[4,5,6]])
transpose(a)

print(a)  # 'a' remains unchanged

a = a.T # If you want to change the definition of a, redefine it.
print(a)
```

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

9.6.4 Merging Arrays

Several function exist for combining multiple arrays into a single array. We'll give examples for a couple and mention the others in reference tables. The most commonly used functions for this task are `vstack` (vertical stacking) and `hstack` (horizontal stacking). `vstack` will stack the original arrays vertically to create the new array and `hstack` will stack them horizontally. Here are some examples.

```
from numpy import linspace, hstack, vstack

a = linspace(0,10,10)
b = linspace(0,5,10)

c = vstack((a,b))
print(c)  # 'a' remains unchanged

d = hstack((a,b))
print(a.T)
print(d)
```

```
[[ 0.          1.11111111  2.22222222  3.33333333  4.44444444  5.55555556
```

```
   6.66666667  7.77777778  8.88888889 10.        ]
 [ 0.          0.55555556  1.11111111  1.66666667  2.22222222  2.77777778
   3.33333333  3.88888889  4.44444444  5.        ]]
[ 0.          1.11111111  2.22222222  3.33333333  4.44444444  5.55555556
  6.66666667  7.77777778  8.88888889 10.        ]
[ 0.          1.11111111  2.22222222  3.33333333  4.44444444  5.55555556
  6.66666667  7.77777778  8.88888889 10.          0.          0.55555556
  1.11111111  1.66666667  2.22222222  2.77777778  3.33333333  3.88888889
  4.44444444  5.        ]
```

9.7 Commonly-used Array Methods and Functions

NumPy contains an extensive listing of array methods and functions and it would be impractical to list them all here. However, below you will find some tables of some of the commonly used ones that can serve as a reference.

Table 9.2: Array Attribute Methods and Functions

Method	Description
`shape(array)` or `array.shape`	Returns the dimensions of the array.
`ndim(array)` or `array.ndim`	Returns the number of dimensions (i.e. a 2D array is 2).
`size(array)` or `array.size`	Returns the number of elements in an array.

Table 9.3: Array Modification Methods and Functions

Method	Description
`array.flatten()`	Returns a flattened view of `array`.
`reshape(array,dims)` or `array.reshape(dims)`	Returns a view of `array` reshaped into an array with dimensions given by `dims`.
`array.resize(dims)`	**Modifies** `array` to be a resized array with dimensions `dims`.
`transpose(array)` or `array.T`	Returns a view of the transpose of `array`.
`sort(array)`	Returns a view of a sorted version of `array`.
`array.sort()`	**Modifies** `array` to be sorted.
`argsort(array)`	Returns index values that will sort `array`.
`array.fill(x)`	**Modifies** `array` so that all elements are equal to `x`.
`vstack(a,b)`	Vertically stack arrays `a` and `b` to form the new array.
`hstack(a,b)`	Horizontally stack arrays `a` and `b` to form the new array.

Method	Description
`vsplit(array,n)`	Splits `array` veritcally into `n` equal parts.
`hsplit(array,n)`	Splits `array` horizontally into `n` equal parts.
`append(array,x)`	Returns a view of `array` with `x` added to the end of the array.
`insert(array,n,x)`	Returns a view of `array` with `x` inserted at location `n`.
`delete(array,n)`	Returns a view of `array` with element at location `n` removed.
`unique(array)`	Returns a view of the unique elements of `array`.

Table 9.4: Array Measurement Methods and Functions

Method	Description
`min(array)` or `array.min()`	Returns the minimum value in an array.
`max(array)` or `array.max()`	Returns the maximum value in an array.
`argmin(array)` or `array.argmin()`	Returns the location of the minimum value in an array.
`argmax(array)` or `array.argmax()`	Returns the location of the maximum value in an array.
`fmin(array1,array2)`	Returns the minimum between two arrays of the same size. (Arrays are compared elementwise.)
`fmax(array1,array2)`	Returns the maximum between two arrays of the same size. (Arrays are compared elementwise.)
`mean(array)` or `array.mean()`	Returns the mean of `array`.
`median(array)` or `array.median()`	Returns the median of `array`.
`std(array)` or `array.std()`	Returns the standard deviation of `array`.
`cumprod(array)` or `array.cumprod()`	Returns the cumulative product of `array`.
`cumsum(array)` or `array.cumsum()`	Returns the cumulative sum of `array`.
`sum(array)` or `array.sum()`	Returns the sum of all elements in `array`.
`prod(array)` or `array.prod()`	Returns the product of all elements in `array`.
`floor(array)`	Returns the floor (i.e., rounds down) of all elements in `array`.
`ceil(array)`	Returns the ceiling (i.e., rounds up) of all elements in `array`.

10 File I/O (Input/Output)

Up to now, we have been working with computer-generated or manually typed data sets. Often in a scientific setting your data will be stored in a file and you will need to read the contents of the file into Python so you can perform an analysis. Most data files are *text* files, but there is a large variety of these that differ mostly in the way the information is formatted. The file extension (i.e., the 3 or 4 letters after the period at the end of a file name) specifies the formatting of the file. For example, a *.csv* file (short for comma-seperated values) has commas to separate the information.

10.1 Reading Lines

The first way to read a file is using a `for` loop to iterate over the file line by line. Admittedly, this is not the most elegant or efficient way to read a file but we present it first because it always works. First, the file is opened using the `open` command. The file should be attached to a variable for later use. Next, the data is read one line at a time using the `readlines()` method. We should use a `for` loop for this. Finally, it is a good idea to close the file when you're finished.

```
file = open("squares.csv")
for line in file.readlines():
    print(line)

file.close()
```

```
1,1

2,4

3,9

4,16

5,25
```

```
6,36

7,49

8,64

9,81

10,100
```

You can see how each line gets read separately and printed off. But this isn't super useful yet because we'd probably like to have the numbers stored in lists for our forthcoming analysis. We can fix this by creating some empty lists and appending the appropriate values as they are read in.

```
numbers = []
squares = []

file = open("squares.csv")
for line in file.readlines():
    numbers.append( int(line.split(',')[0]) )
    squares.append( int(line.split(',')[1]) )

file.close()

print(numbers)
print(squares)
```

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

Now the data from the file is saved in Python lists and our analysis can proceed.

One final note: if all you want to do is read your file in as a list of strings (one string for each line), that can be done without a `for` loop using the function `readlines` (rather than `readline`).

```
file = open("squares.csv")
data = file.readlines()
file.close()

print(data)
```

```
['1,1\n', '2,4\n', '3,9\n', '4,16\n', '5,25\n', '6,36\n', '7,49\n', '8,64\n', '9,81\n', '10,
```

10.2 Using NumPy's `genfromtxt` function

If the data file is highly structured (every line looks the same, separater character is consistent across the file, etc) then NumPy's `genfromtxt` function can read the data very efficiently into an array. The `genfromtxt` function requires only one argument (the file name) with another optional argument (`delimiter`) that is typically included to specify the character used to separate the data. Below is an example for using this function to read in the .csv data we have been working with.

```
from numpy import genfromtxt

data = genfromtxt('squares.csv',delimiter = ",")

print(data)
```

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

Notice that the data was read into a NumPy array (2D in this case because there were two columns of data), which means that we can easily slice off the individual columns if needed.

```
from numpy import genfromtxt

data = genfromtxt('squares.csv',delimiter = ",")

numbers = data[:,0]
squares = data[:,1]
print(numbers)
print(squares)
```

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

11 Basic Plotting

Creating plots is an important task in science and engineering. The old addage "A picture is worth a thousand words!" is wrong.... it's worth way more than that if you do it right. When making plots on a computer it is important to remember that computers don't plot functions, rather they plot individual points. Only when you connect those points does the image look like the function you are so used to seeing. In this chapter we will use a library called `matplotlib` for plotting. More specifically, we will import the `pyplot` function inside of `matplotlib`. It is customary to use `plt` as an alias for `pyplot`.

```
from matplotlib import pyplot as plt
%matplotlib inline
```

The `%matplotlib inline` statement is a Jupyter notebook command. It tells Jupyter to display any plots generated directly in the notebook instead of in a separate window. If you use matplotlib in another environment, you should remove this line and instead place `plt.show()` af the plot commands.

11.1 Plotting Functions of a Single Variable

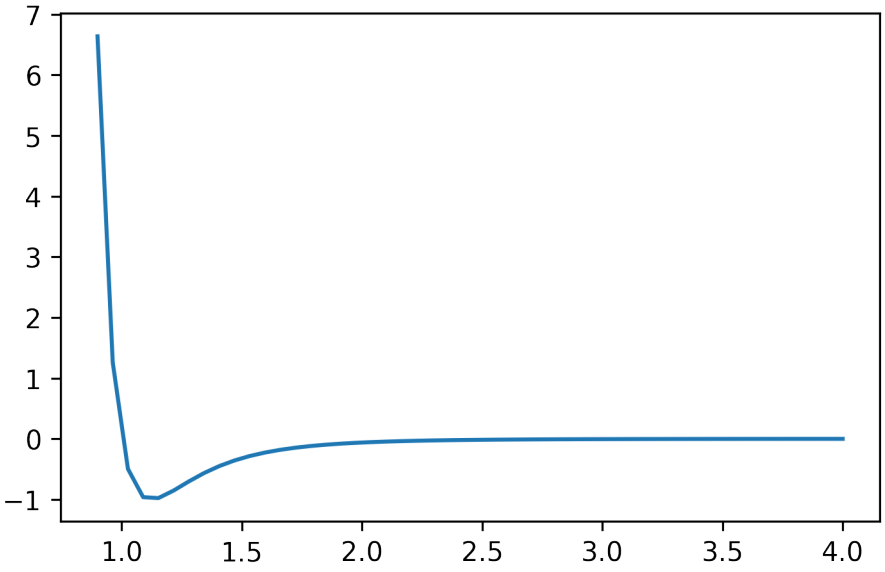
In order to make a plot, matplotlib needs lists of the x and y coordinates for the points that are going to be plotted. A good choice for generating the x-coordinates is either `linspace` or `arange` from NumPy. The following equation is called the Lennard-Jones equation and it gives the energy of two atoms interacting as a function of separation distance

$$E = 4\sigma\left[\left(\frac{\epsilon}{r}\right)^{12} - \left(\frac{\epsilon}{r}\right)^{6}\right]$$

Let's plot this function from 0.9 to 4.0.

```
from matplotlib import pyplot as plt
%matplotlib inline
from numpy import linspace,sqrt

sigma = 1
```

```
epsilon = 1
r = linspace(0.9,4,50)
energy = 4 * sigma* ((epsilon/r)**12 - (epsilon/r)**6)

plt.figure()
plt.plot(r,energy)
```



By default, matplotlib connects the points to form a line (albeit a bit jagged because we didn't plot that many points).

11.1.1 Linestyles, Markers, and Colors

Three optional arguments can help you control the look of the plot: `marker`, `linestyle`, and `color`. All of these arguments take strings. The `linestyle` argument determines if the line is solid or dashed and what type of dashing to use. The `marker` argument specifies the shape of the plot marker to be used. Below are tables listing possible options for these arguments.

Table 11.1: Common Marker Styles

Argument	Description
`o`	circle
`*`	star
`p`	pentagon

Argument	Description
`^`	triangle
`s`	square

Table 11.2: Common Line Styles

Argument	Description
`-`	solid
`--`	dashed
`-.`	dash-dot
`:`	dotted

Table 11.3: Common Colors

Argument	Description
`b`	blue
`r`	red
`k`	black
`g`	green
`m`	magenta
`c`	cyan
`y`	yellow

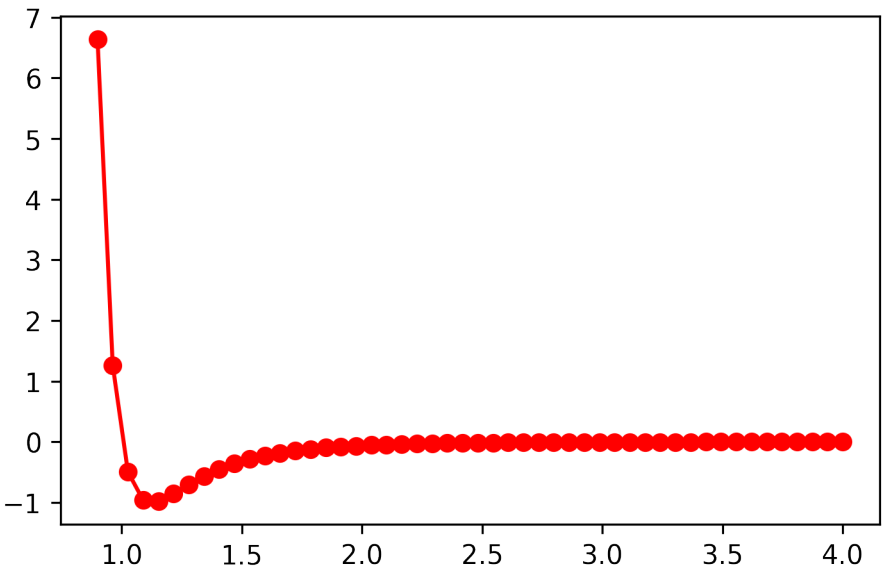
Several other keyword arguments exist for helping you customize the look of your plots. They are summarized in the table below.

Table 11.4: A Few Common `plot` keyword arguments

Argument	Description
`linestyle` or `ls`	line style
`marker`	marker shape
`linewidth` or `lw`	line width
`color` or `c`	line color
`markersize` or `ms`	marker size

Here is the plot from above with some of these keyword arguments added.

```
from matplotlib import pyplot as plt
%matplotlib inline
from numpy import linspace,sqrt

r = linspace(0.9,4,50)
energy = 4 * ((1/r)**12 - (1/r)**6)

plt.figure()
plt.plot(r,energy,linestyle = '-', marker = 'o', color = 'r')
```



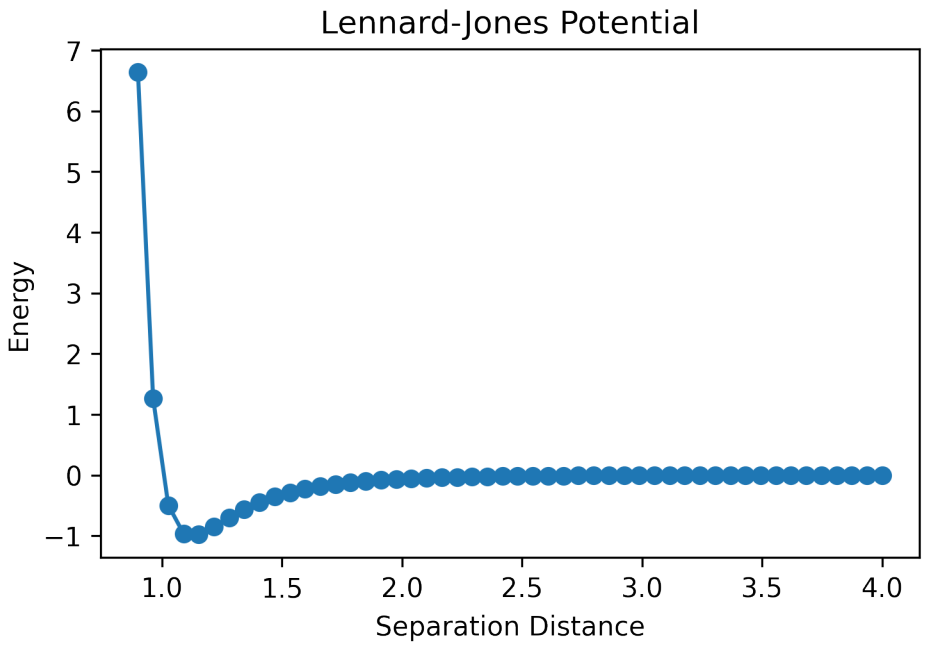
11.1.2 Labeling Plots

All good plots have axes labels and a title and you can add them to a matplotlib plot using the `xlabel()`, `ylabel()`, and `title()` functions which are placed on their own line after the `plot` command.

```
from matplotlib import pyplot as plt
%matplotlib inline
from numpy import linspace,sqrt

r = linspace(0.9,4,50)
energy = 4 * ((1/r)**12 - (1/r)**6)
```

```
plt.figure()
plt.plot(r,energy,marker = 'o')
plt.xlabel("Separation Distance")
plt.ylabel("Energy")
plt.title("Lennard-Jones Potential")
```

```
Text(0.5, 1.0, 'Lennard-Jones Potential')
```



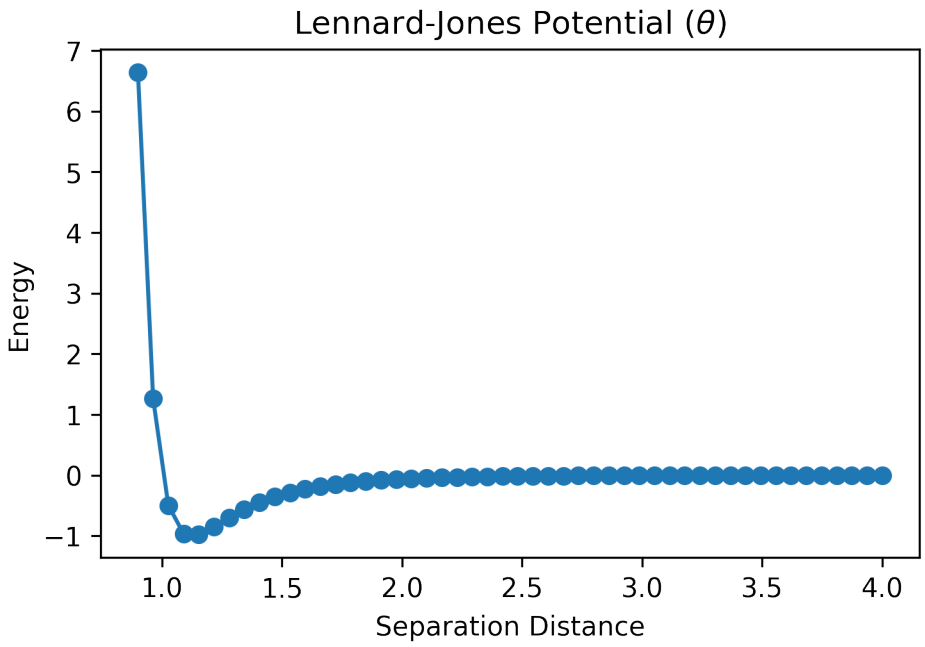
11.1.3 Greek Letters

In physics, Greek letters get used all the time and you may find yourself wanting to use one in a plot title or axes label. This can be accomplished by placing an `r` in front of the title string and then placing the name of the greek variable inside of `$` with a backslash in front of it. This is better illustrated with an example.

```
from matplotlib import pyplot as plt
%matplotlib inline
from numpy import linspace,sqrt

r = linspace(0.9,4,50)
```

```
energy = 4 * ((1/r)**12 - (1/r)**6)

plt.figure()
plt.plot(r,energy,marker = 'o')
plt.xlabel("Separation Distance")
plt.ylabel("Energy")
plt.title(r"Lennard-Jones Potential ($\theta$)")
```

```
Text(0.5, 1.0, 'Lennard-Jones Potential ($\\theta$)')
```



You can subcript any character using the `_` character followed by the subscript. θ_1 can be written as `\theta_1`. If the subscript is more than one character, you'll need to enclose it in curly braces. θ_{12} is written as `\theta_{12}`. Superscripts work the same way only using the `^` chararcter instead of the underscore.

Table 11.5: Lowercase greek letters

Argument	Description
α	`\alpha`
β	`\beta`
γ	`\gamma`

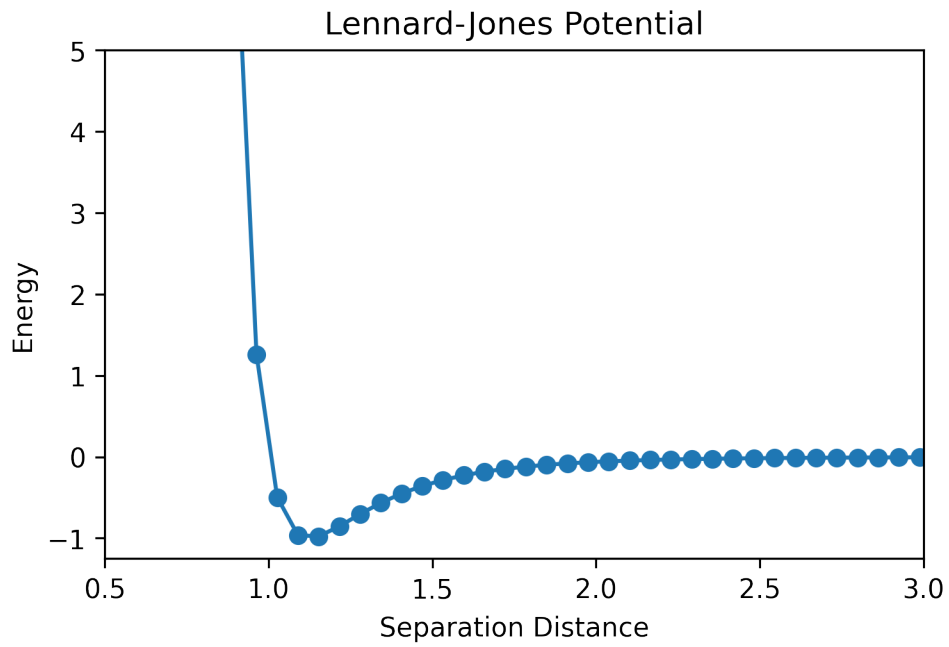
Argument	Description
δ	\delta
ϵ	\epsilon
ϕ	\phi
θ	\theta
κ	\kappa
λ	\lambda
μ	\mu
ν	\nu
π	\pi
ρ	\rho
σ	\sigma
τ	\tau
ξ	\xi
ζ	\zeta

11.1.4 Controlling the Axes

By default, matplotlib will size the plotwindow to include all of the points. If you want to zoom in our out, you can do so with the `xlim` and `ylim` functions. These functions should be placed after the plot command on their own line just like the label commands.

```
from matplotlib import pyplot as plt
%matplotlib inline
from numpy import linspace,sqrt

r = linspace(0.9,4,50)
energy = 4 * ((1/r)**12 - (1/r)**6)

plt.figure()
plt.plot(r,energy,marker = 'o')
plt.xlim(0.5,3)
plt.ylim(-1.25,5)
plt.xlabel("Separation Distance")
plt.ylabel("Energy")
plt.title("Lennard-Jones Potential")
```

```
Text(0.5, 1.0, 'Lennard-Jones Potential')
```



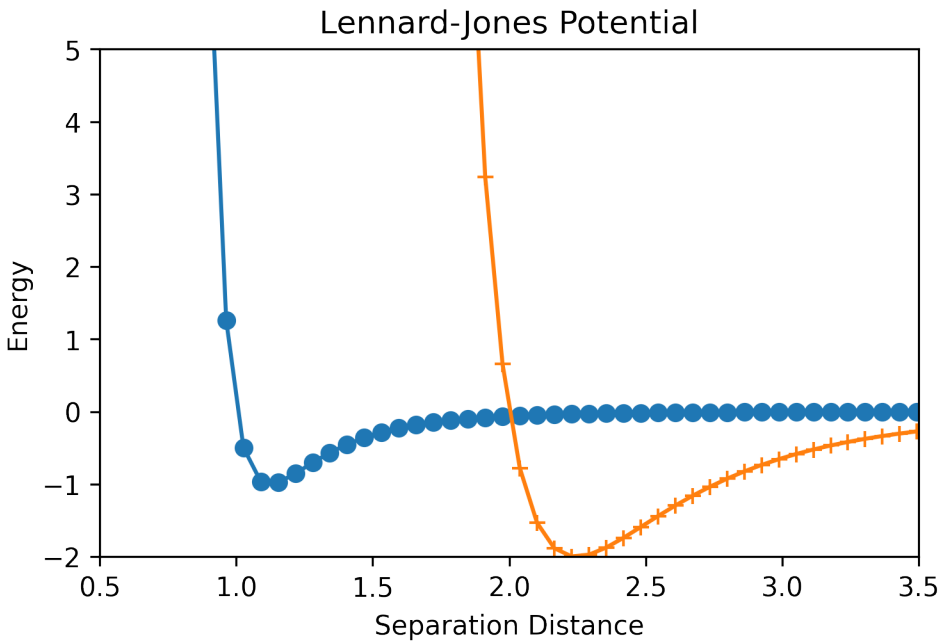
11.1.5 Overlaying Plots

Often you will want to plot more than one set of data on the same set of axes. This can be accomplished two ways. The first way is to call the `plot` function twice in the same Jupyter notebook cell. Matplotlib will automatically place the plots on the same figure and scale it appropriately. Below you will find a plot of the Lennard-Jones potential for two choices of the parameters σ and ϵ.

```
from matplotlib import pyplot as plt
%matplotlib inline
from numpy import linspace,sqrt

r = linspace(0.9,4,50)

sigmaOne = 1
epsilonOne = 1
sigmaTwo = 2
epsilonTwo = 2
energyOne = 4 * sigmaOne* ((epsilonOne/r)**12 - (epsilonOne/r)**6)
energyTwo = 4 * sigmaTwo * ((epsilonTwo/r)**12 - (epsilonTwo/r)**6)
```

```
plt.figure()
plt.plot(r,energyOne,marker = 'o')
plt.plot(r,energyTwo,marker = '+')
plt.xlim(0.5,3.5)
plt.ylim(-2,5)
plt.xlabel("Separation Distance")
plt.ylabel("Energy")
plt.title("Lennard-Jones Potential")
```

```
Text(0.5, 1.0, 'Lennard-Jones Potential')
```

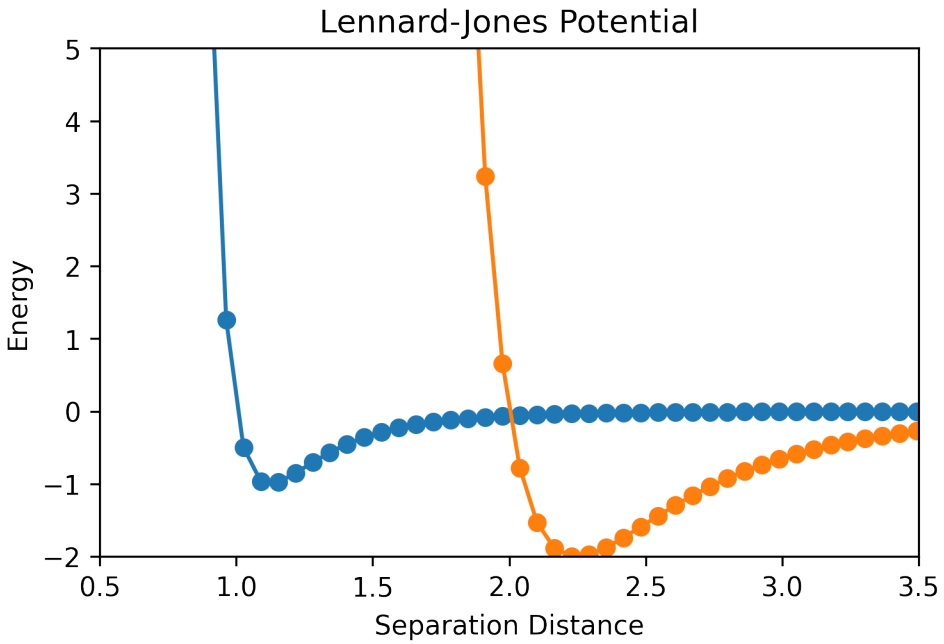


The other way to overlay plots is to include both sets of data into a single `plot` command.

```
from matplotlib import pyplot as plt
%matplotlib inline
from numpy import linspace,sqrt

r = linspace(0.9,4,50)

sigmaOne = 1
epsilonOne = 1
```

```
sigmaTwo = 2
epsilonTwo = 2
energyOne = 4 * sigmaOne* ((epsilonOne/r)**12 - (epsilonOne/r)**6)
energyTwo = 4 * sigmaTwo * ((epsilonTwo/r)**12 - (epsilonTwo/r)**6)

plt.figure()
plt.plot(r,energyOne,r,energyTwo,marker = 'o')
plt.xlim(0.5,3.5)
plt.ylim(-2,5)
plt.xlabel("Separation Distance")
plt.ylabel("Energy")
plt.title("Lennard-Jones Potential")
```

```
Text(0.5, 1.0, 'Lennard-Jones Potential')
```



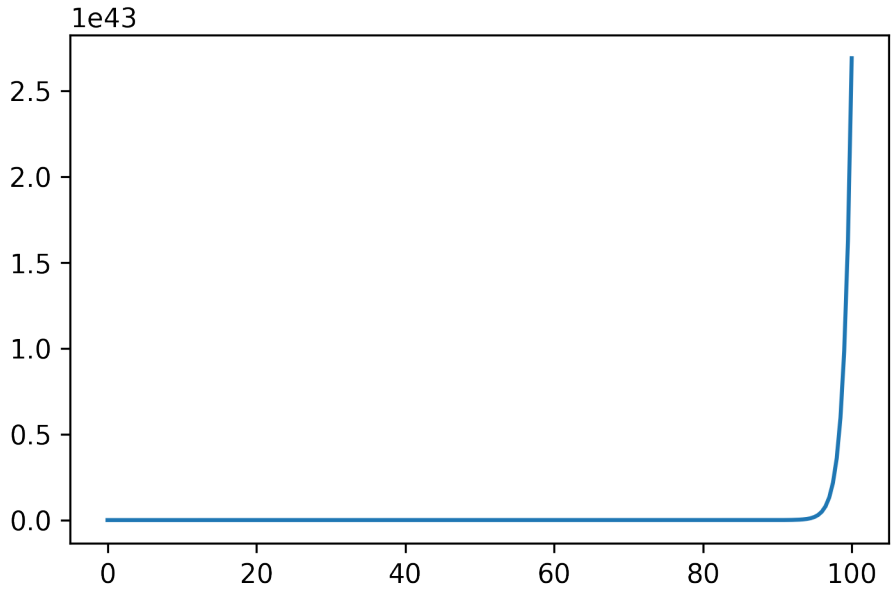
11.2 Other Plot Types

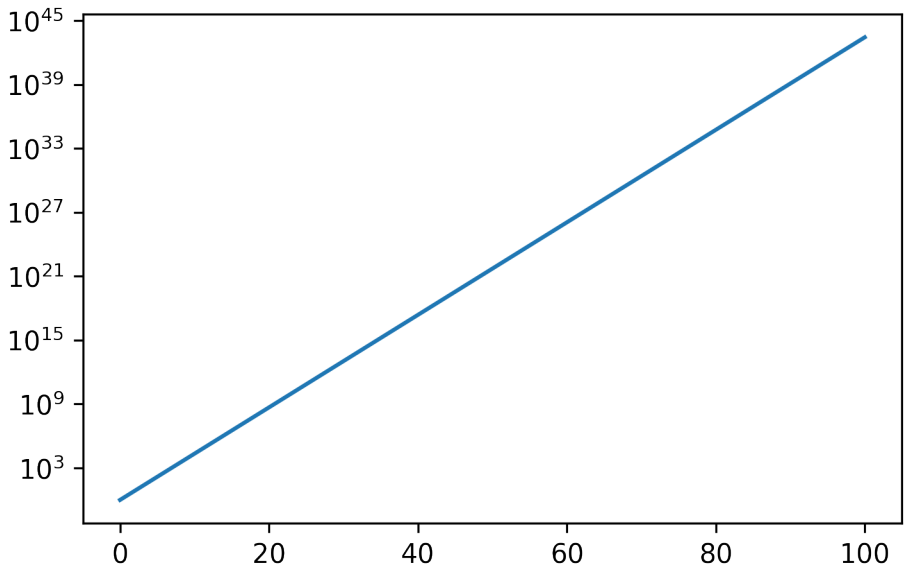
Beyond the line plot that we learned about in the previous section, matplotlib can generate many other types of plots that are very useful in a scientific setting. We'll explore some of them here.

11.2.1 Logarithmic Plots

Sometimes the function being plotted increases or decreases by many orders of magnitude and a normal linear plot would not be particularly useful. Logarithmic plots can be made with the use of the `semilogx`, `semilogy`, or `loglog` functions depending on which axes you want to be on a logarithmic scale. Consider the first plot produced below. Notice that it rises from 0 at $x = 0$ to 10^{45} at $x = 100$. Plotting this function with the y-axis scaled logarithmically will smooth out the plot and make it easier to analyze.

```
from matplotlib import pyplot as plt
%matplotlib inline
from numpy import linspace,exp

x = linspace(0,100,200)
y = exp(x)

plt.figure()
plt.plot(x,y)
plt.figure()
plt.semilogy(x,y)
```



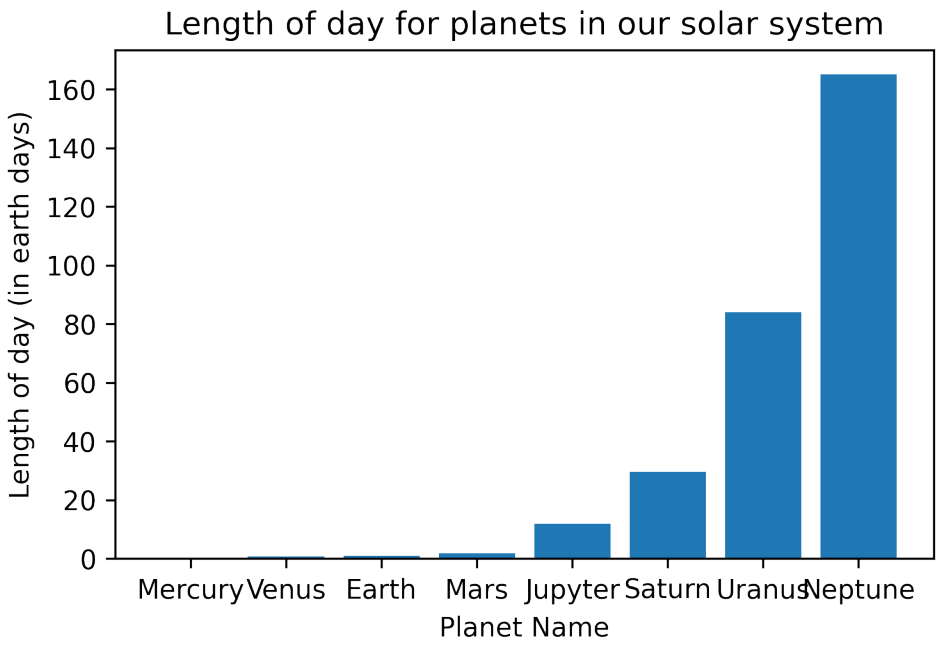


11.2.2 Bar Plots

A bar plot and a scatter plot are quite similar except that instead of a plot marker indicating the associated value, the height of the bar represents the value. You can make a bar plot using the `bar` function inside of `pyplot`.

```
from matplotlib import pyplot as plt
%matplotlib inline
from numpy import linspace,sqrt

#d = [5.8e10,1.9e11,1.5e11,2.3e11,7.8e11,1.4e12,2.9e12,4.5e12]
d = [1,2,3,4,5,6,7,8]
T = [0.241,0.615,1,1.88,11.9,29.5,84,165]

plt.figure()
plt.bar(d,T,tick_label = ["Mercury","Venus","Earth","Mars","Jupyter","Saturn","Uranus","Ne
plt.xlabel("Planet Name")
plt.ylabel("Length of day (in earth days)")
plt.title("Length of day for planets in our solar system")
```

```
Text(0.5, 1.0, 'Length of day for planets in our solar system')
```



Notice the optional argument `tick_labels` used to add labels to the bars. If that were left off, the bars would be labeled using the numbers supplied. Other optional arguments that are available for the `bar` command are given below.

Table 11.6: A Few Common `bar` keyword arguments

Argument	Description
`width`	bar width
`color`	bar color
`xerr`	X error bar
`yerr`	Y error bar
`capsize`	size of caps on error bars

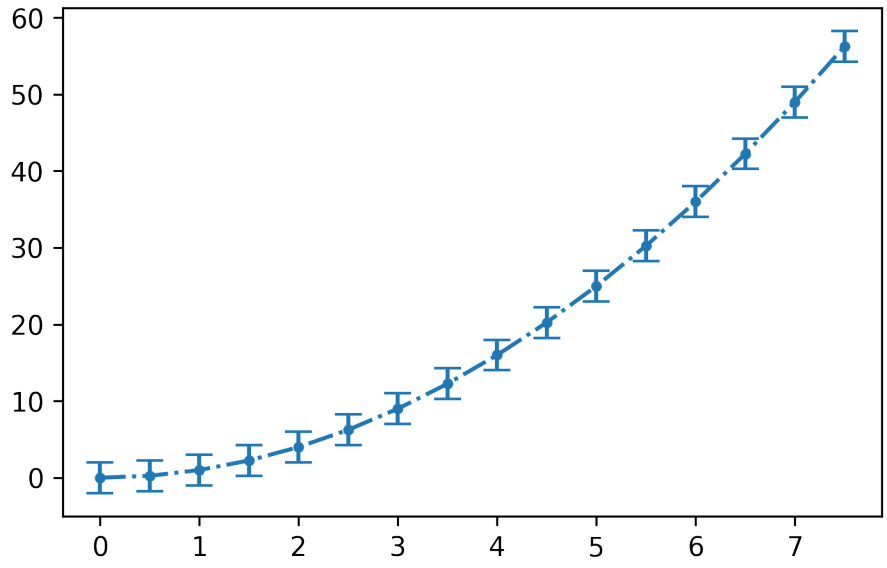
11.2.3 Errorbar Plots

To make plots with error bars use matplotlilb's `errorbar` function. You can choose to add error bars on the x or y axis using the keyword arguments `xerr` and `yerr`.

```
from matplotlib import pyplot as plt
```

```
from numpy import arange

x = arange(0,8,0.5)
y = x**2

x_err = 0.05 # Same error for all points
y_error = 2 # Different error for each point

plt.errorbar(x,y,linestyle = '-.', marker = 'o',markersize = 3,yerr=y_error,capsize = 5)
```

```
<ErrorbarContainer object of 3 artists>
```



The `linestyle`, `marker`, and `markersize` arguments work with this plot type also.

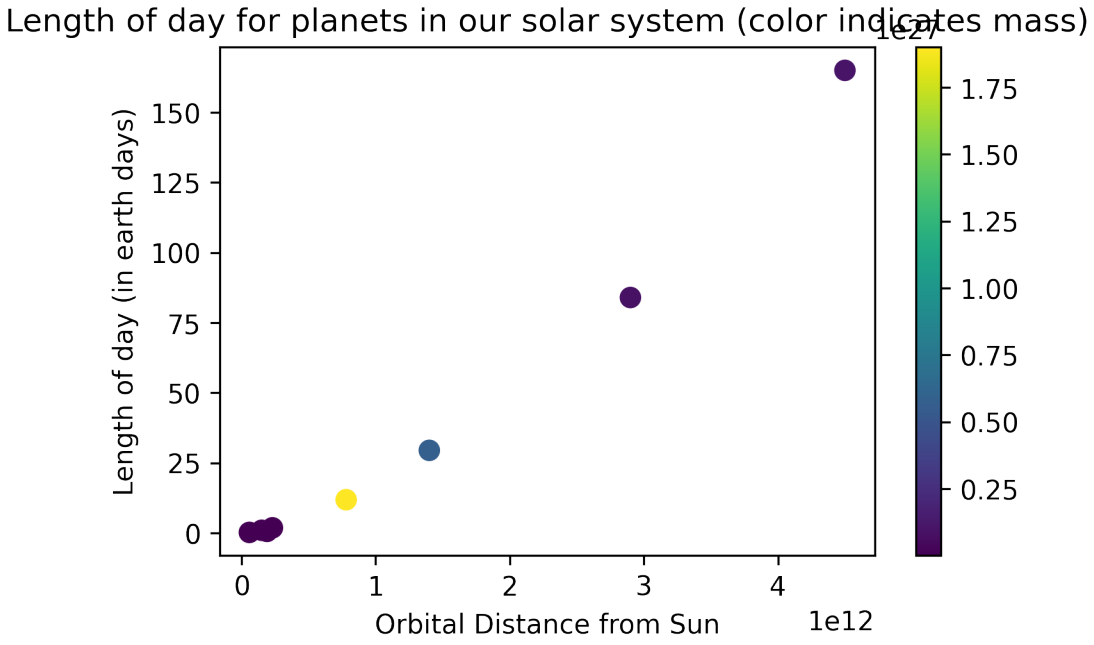
11.2.4 Scatter Plots

We have already generated scatter plots using the `plot` function but you can also use the `scatter` command to do the same thing. The only other additional functionality with `scatter` is the ability to specify the color, shape, and size of each plot marker individually.

```
from matplotlib import pyplot as plt
%matplotlib inline
```

```
from numpy import linspace,log

d = [5.8e10,1.9e11,1.5e11,2.3e11,7.8e11,1.4e12,2.9e12,4.5e12]
#d = [1,2,3,4,5,6,7,8]
T = [0.241,0.615,1,1.88,11.9,29.5,84,165]
mass = [3.2e23,4.9e24,6e24,6.4e23,1.9e27,5.7e26,8.7e25,1e26]

plt.figure()
plt.scatter(d,T,c = mass,s= 50)
plt.xlabel("Orbital Distance from Sun")
plt.ylabel("Length of day (in earth days)")
plt.title("Length of day for planets in our solar system (color indicates mass)")
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x116951b50>
```



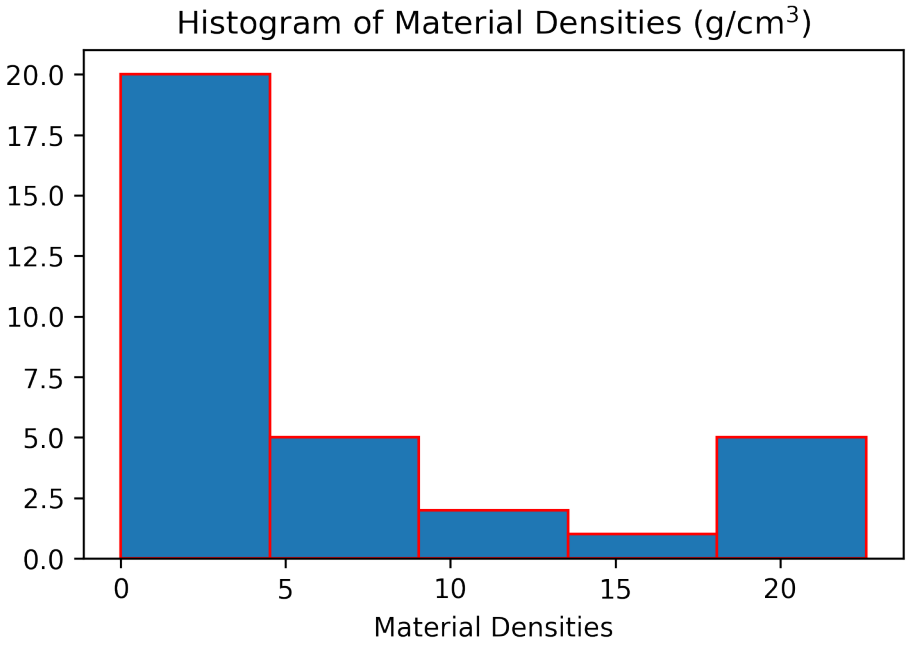
11.2.5 Histograms

Histograms display bars representing the frequency of values in a given data set. Unlike bar plots, the width of the bar is meaningful since the each bar represents the number of x-values

that fall within a range given by the width of the bar. A histogram can be constructed using the `hist` function. There is only one required argument, which is the data set. Some commonly used keyword arguments are `bins` which specifies how many equally-spaced groups (called *bins*) to generate and `edgecolor` which can be used to specify the color of the barsedges.

```
from matplotlib import pyplot as plt
%matplotlib inline
from numpy import linspace,log

densities = [0.00009,0.000178,0.00125,0.001251,0.001293,0.001977,0.534,0.810,0.900,0.920,0

plt.figure()
plt.hist(densities,bins = 5,edgecolor = 'r')
plt.xlabel("Material Densities")
plt.title(r"Histogram of Material Densities (g/cm$^3$)")
```

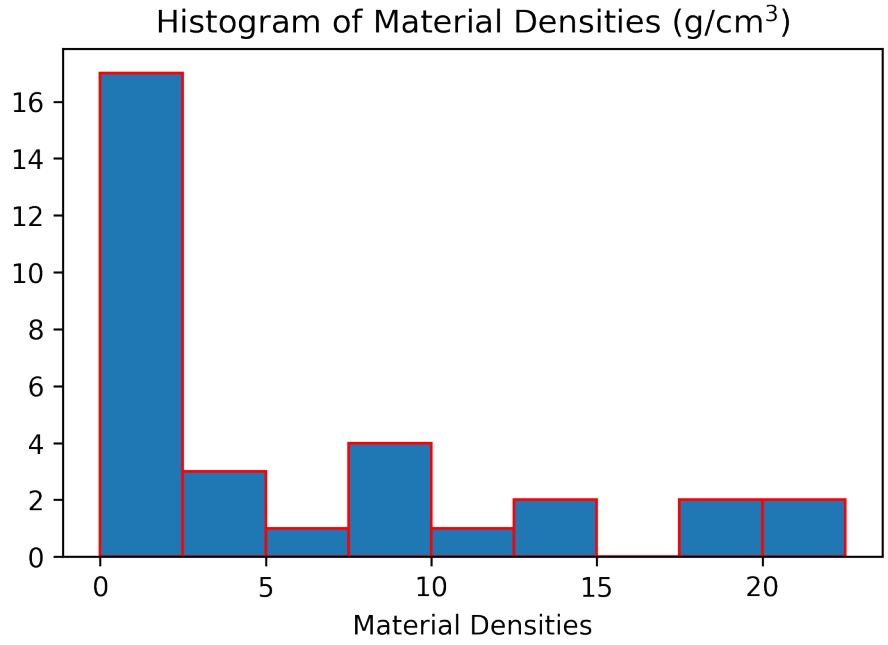
```
Text(0.5, 1.0, 'Histogram of Material Densities (g/cm$^3$)')
```



Here we have used `bins = 5` which will produce a histogram with 5 equal-size bins. Alternatively, we can specify the exact locations of all the bin edges by placing them in a list.

```
from matplotlib import pyplot as plt
%matplotlib inline
from numpy import linspace,log

densities = [0.00009,0.000178,0.00125,0.001251,0.001293,0.001977,0.534,0.810,0.900,0.920,0

plt.figure()
plt.hist(densities,bins = [0,2.5,5,7.5,10,12.5,15,17.5,20,22.5],edgecolor = 'r')
plt.xlabel("Material Densities")
plt.title(r"Histogram of Material Densities (g/cm$^3$)")
```

```
Text(0.5, 1.0, 'Histogram of Material Densities (g/cm$^3$)')
```



11.3 Multifigure Plots

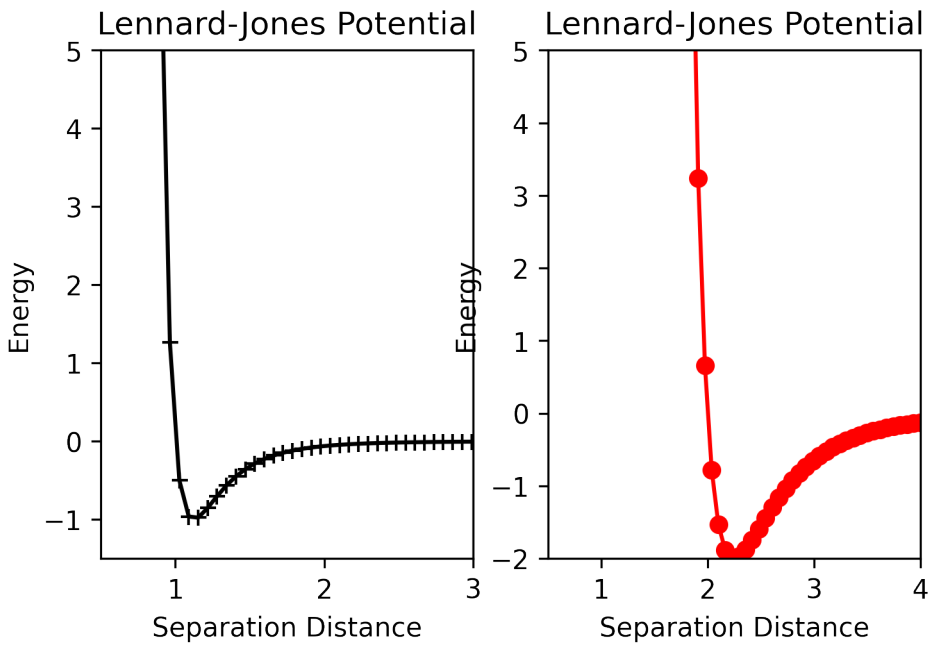
To generate multiple, independent plots in the same figure a few more lines of code are necessary to specify how you want the plots arranged. There are two general methods for doing that and we will demonstrate both of them. In the first method, you'll first use the `figure()` command to generate a figure and the `subplot()` function to generate the individual plots in the figure. The `subplot` function take three arguments.

```
subplot(rows,columns,plot_number)
```

As an example, to generate a figure of two plots side by side.

```
from matplotlib import pyplot as plt
%matplotlib inline
from numpy import linspace,sqrt

r = linspace(0.9,4,50)

sigmaOne = 1
epsilonOne = 1
sigmaTwo = 2
epsilonTwo = 2
energyOne = 4 * sigmaOne* ((epsilonOne/r)**12 - (epsilonOne/r)**6)
energyTwo = 4 * sigmaTwo * ((epsilonTwo/r)**12 - (epsilonTwo/r)**6)

plt.figure()
plt.subplot(1,2,1)
plt.plot(r,energyOne,marker = '+',color = 'k')
plt.xlim(0.5,3.0)
plt.ylim(-1.5,5)
plt.xlabel("Separation Distance")
plt.ylabel("Energy")
plt.title("Lennard-Jones Potential")
plt.subplot(1,2,2)
plt.plot(r,energyTwo,marker = 'o',color = 'r')
plt.xlim(0.5,4.0)
plt.ylim(-2,5)
plt.xlabel("Separation Distance")
plt.ylabel("Energy")
plt.title("Lennard-Jones Potential")
```

```
Text(0.5, 1.0, 'Lennard-Jones Potential')
```



Pay close attention to the `subplot` function. `subplot(1,2,1)` will generate a 1 x 2 grid of plots and place the next plot generated at the 1st location. Below is an example of a more advanced array of plots. Pay close attention to the `subplot` functions until you understand.

```
from matplotlib import pyplot as plt
%matplotlib inline
from numpy import linspace,sqrt

r = linspace(0.9,4,50)

sigmaOne = 1
epsilonOne = 1
sigmaTwo = 2
epsilonTwo = 2
energyOne = 4 * sigmaOne* ((epsilonOne/r)**12 - (epsilonOne/r)**6)
energyTwo = 4 * sigmaTwo * ((epsilonTwo/r)**12 - (epsilonTwo/r)**6)

plt.figure()
plt.subplot(2,1,1)
plt.plot(r,energyOne,marker = '+',color = 'k')
plt.xlim(0.5,3.0)
plt.ylim(-1.5,5)
```

```
plt.xlabel("Separation Distance")
plt.ylabel("Energy")
plt.title("Lennard-Jones Potential")
plt.subplot(2,2,3)
plt.plot(r,energyTwo,marker = 'o',color = 'r')
plt.xlim(0.5,4.0)
plt.ylim(-2,5)
plt.xlabel("Separation Distance")
plt.ylabel("Energy")
plt.title("Lennard-Jones Potential")
plt.subplot(2,2,4)
plt.plot(r,energyTwo,marker = 'o',color = 'r')
plt.xlim(0.5,4.0)
plt.ylim(-2,5)
plt.xlabel("Separation Distance")
plt.ylabel("Energy")
plt.title("Lennard-Jones Potential")
plt.tight_layout()
```