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, in 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 and modify. 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

Basic Python

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 wan 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 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 t
```

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.

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 only 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
```

The == operator is dangerous to use because two numbers are rarely identical due to the way computers store numbers. We recommend that you not use this operator to compare two numbers. The cell below illustrates what I mean.

```
a = (25.4/10.0) * (1.0/2.54)
b = ((25.4/10.0) * 1.0)/2.54

print(a,b)

a == b
```

```
0.9999999999999999 1.0
```

```
False
```

Mathematically, these two numbers are identical, but in the computer these numbers are represented differently and therefore they are not recognized as identical. A better way to compare two numbers is to see if the absolute value of the difference of the two numbers is very small.

```
a = (25.4/10.0) * (1.0/2.54)
b = ((25.4/10.0) * 1.0)/2.54

abs(a - b) < 1e-8
```

```
True
```

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 $c = 1$.

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