QUT Astrophysics

QUT Astrophysics Research Group
An Introduction to Python for Astronomers

Dr Michael Cowley
michael.cowley@qut.edu.au

Introduction

Many of the astrophysics research projects QUT offers requires the use of Python. If you are new to Python or programming, you are encouraged to work through this guide before starting work on your research project. Before you begin, you should download and install a Python development environment on your own computer such as Anaconda, and an editor such as Jupyter Notebooks. A cloud-based alternative to both of these is Google Colab – this brief intro video should get you started. Portions of this guide are based on "Python for Astronomers" by Imad Pasha and Chris Agostino.

Python

As far as astronomers are concerned, the use of programming languages (and everything that comes with them) basically amounts to glorified calculator use. At the end of the day, we have some numbers, and we want to do things to those numbers. Different programming languages (like C, C++, Java, and Fortran) operate much like human languages in the real world – they are alternative methods of constructing statements with a certain meaning. Just like normal languages, phrasing certain things is easier in some languages than others. This guide is concerned with Python in particular. This is because Python is an easy to learn, powerful programming language, which has become the language of choice for astronomers working with data analysis and visualisation.

Hello, World!

"Hello, world!" is a simple approach to illustrate a language's basic syntax. In Python, this is dead easy. From the Jupyter Notebook or Google Colab screen, enter the following into a new cell and click run.

```
[IN]: print('Hello, world!')
```

The terminal will respond by showing your phrase in an output cell. You can think of this as "printing" the value to the screen where the 'print' command is a function. Think back to math class, where you might write sin(x). The x would be the argument of the function sin. In the example here, it may not seem useful since we told Python to print "Hello, world!" in the first place. But as we will soon find out, the beauty of coding is that you can save numeric (and other) values into little containers called variables, and no longer have to keep track of their intermediate values as they get pushed through lines of calculation. Furthermore, there will be plenty of times when either your code is failing, or you want to check the operation of your code. These are perfect places to stick a print statement, which will output values to the screen so you can manually evaluate them.

Data Types

Python, like most programming languages, divides up all the possible "things" you can play around with into what are called data types. Some of these divisions seem obvious: clearly a word like "cat" is a fundamentally different data type than a list of numbers [1,2,3,4,5]. Other divisions seem more arbitrary

at first glance: For example, Python makes the distinction between integers (the counting numbers), and floats (numbers with decimals). It does so because of the way computer processors store information in bits, but it leads to the interesting (and important) characteristic that "42" and "42." are different in Python and take up different amounts of computer memory. Some basic data types are listed and defined below, and you will learn more about them as we progress through this guide:

1. Integers: The counting numbers. e.g. `-1,0,1,2,3,4,5,` ...
2. Floats: Decimal numbers. e.g. `1., 2.345, 6.3, 999.99999,` ...
3. Strings: An iterable data type mostly used to hold words/phrases or path locations. Denoted by single or double quotes. e.g. `''cat'', ''/home/cowley'', ''1530'',` ...
4. Lists: Stored lists of any combination of data types, denoted with brackets. e.g. `[1,2,'star','fish']` or `[1, 2, [3, 4, 5], 'star']` (notice that you can have lists within lists)
5. NumPy Arrays: Like lists but can only contain one data type at a time and have different operations. Defined in NumPy, not native Python, but so ubiquitous we include them here.
6. Tuple: Also like a list, but immutable (unchangeable). Somewhat like a read-only list. These are defined with parentheses. e.g. `tuple1 = ('hi', 1, 4, 'bye')`
7. Dictionaries: A collection of pairs, where one is a "key" and the other is a "value." One can access the "value" attached to a key by indexing the dictionary by key: `dictionary_name['key']`
8. Boolean: A data type with only two possible values: True, or False. They are used in conditional statements.

Basic Math

Within Python, you can perform simple to very complex mathematical operations. Let's see how adding and subtracting works in a Jupyter Notebook. Start by entering the [IN] commands, before hitting the run command after each line:

```
[IN]: 3 + 5
[OUT]: 8
[IN]: 9 - 3
[OUT]: 6
```

We can also test multiplication and division (denoted in Python with * and /):

```
[IN]: 4 * 3
[OUT]: 12.0
[IN]: 1/2
[OUT]: 0 or 0.5
```

Whether you got 0 or 0.5 in that last step depends on if you are running Python version 2.x or Python 3.x. Python 2.x versions will tell you it's 0, while Python 3.x versions will tell you it's 0.5, which seems odd, since clearly 1/2 is never 0. It also seems odd that different Python versions would tell you different answers to the same operation. The reason we are getting 0 in Python 2.x here is that Python 2.x is performing integer division, meaning the answer must be an integer. In this sort of situation, Python simply rounds down to the nearest integer. The solution to this is to cast either the "1" or "2" (or both) as floats rather than integers. Only one is required to be a float because if one number in an operation (like addition, subtraction, multiplication, division, exponentiation, etc) is a float, it will convert all to floats and express the answer as a float. Now, 90% of the time you will need to be doing float division anyway, so the creators of Python 3.x decided to make that the default division method. We also encourage you to stick with Python 3.x moving forward. For your general knowledge, there is a function for converting integers to floats, and it looks like this:

```
[IN]: float(2)
```

However, there is a much faster way to create floats when you are entering a number manually, which is simply to add a decimal (period) to any number. Try it yourself: demonstrate that 1./2 and 1/2. both output the proper answer. The place when the float() command comes in handy is when you have a variable (say, called "x") in your code, and you don't necessarily know what its value is, perhaps it is the sum of many calculations, but is just an intermediary holding value. If before the next stage of calculations you require it to be a certain data type, you can use this hard casting, like:

```
[IN]: x = float(x)
```

or

```
[IN]: x = int(x)
```

Which will convert it to an integer if it is not already. The change from Python 2.x to 3.x has been painful for many reasons, but one of them has been the fact that any old code that made use of integer divisions as a default now must be changed. Be mindful of this if using old code.

The other basic math operation in Python is exponentiation. In Python this is denoted with a double asterisk ('**'). For example:

```
[IN]: 2**3
[OUT]: 8
```

To perform more "complicated" math like sin, cos, sqrt, etc., requires the use of some additional packages, which we'll discuss soon.

Variables

While using Python as a calculator can be fun, the real power of programming comes in being able to store things (numbers, lists, etc) as variables and access them later. variable is a user-defined, symbolic name which points to a spot in a computer's memory where a value has been stored. The variable's name can then be used to retrieve the value, and the value can be changed at will. Declaring variables in Python is easy; you simply type a desired variable name, an equal sign, and what you want it to be. For example:

```
[IN]: x = 5.0
[IN]: y = 'cat'
[IN]: university = 'no life' + 'no sleep'
```

The above would set the variable x to the floating-point number 5, set y to the string "cat", and set university to the concatenated string "no life no sleep" (more on string concatenation in a bit). Notice that Python doesn't output anything when you declare a variable as it did when you entered a math operation. But rest assured, those values are stored in the computer. If you type:

```
[IN]: print(x)
[OUT]: 5.0
```

As a sidenote, in any Jupyter Notebook terminal, simply entering a variable and hitting <Enter> will print the value:

```
[IN]: y
[OUT]: 'cat'
```

Variables in Python are mutable—that is, you can change them, within certain bounds. Most simply if you consecutively typed:

```
[IN]: x = 5
[IN]: x = 3
```

then printed "x" you would find it is equal to 3. You can also use variables to change themselves:

```
[IN]: x = 5
[IN]: x = 2 * x + 3
```

In this case, the new value for x at the end of the line would be 2 times the value of x going in, plus 3. (in this case, 13). You can also add, subtract, and multiply variables if they are of the right data type:

```
[IN]: x = 5
[IN]: y = 6.
[IN]: z = x + y
[IN]: x = 2 * z
[IN]: y = x / z
```

That is probably a bit confusing to follow and illustrates why typically we avoid such oft redefining of variables, and instead come up with new variable names to store the various sums and products. We can see that when dealing with floats as our data type, the math operations we are used to have the typical "mathematical" results. When dealing with other data types, the behaviour of these operations is unique to that data type. For example, adding two strings 'a' and 'b' produces the single string 'ab'— and something like 4 * 'a' will return 'aaaa'. But the power raising operation 'a'**2 is meaningless and returns an error. We will be spending time learning which operations can be used to modify each data type, and what their various effects are, over the course of this text.

There is definitely subtly involved in determining which data types can be operated together, and in which situation casting is valid (for example, the int() function we discussed can never convert "cat" to an integer, and will throw an error). While some will be touched on here, much of it is common sense and experimentation.

Storing and Manipulating Data in Python

So far, we have primarily focused on data types that are responsible for storing a single piece of information – like a single float, or a string. However, remember that our primary goal for using Python as astronomers is as a tool with which to explore and manipulate data. That data could be in a whole multitude of different forms, including observational images from a telescope, catalogues of measurements from a scientific instrument, the output files of a supercomputer simulation, etc. We use Python because while we could sometimes easily perform a calculation on a star's flux to obtain its luminosity, we might have a collection of 10,000, or even over a million stars. This is where Python comes in — and primarily, where the array, list, and dictionary data types become not only useful but essential.

Arrays vs Lists

Lists and NumPy arrays are the common data type with which to store data, because they allow us to dump all our individual measurements, etc., into a single variable. One of the first questions that usually emerges from students at this stage is "what is the difference between lists and arrays and when do I use

one or the other?" It's a good question, and one worth taking a section to explore. We'll begin with lists, as these are the native data type within Python. Let's define a list to play with:

```
[IN]: my_list = [1,5,2,7,3,7,8]
```

This list of numbers could be, say, the distance in parsecs (evidently rounded) of several nearby stars. What happens if I want to multiply all the distances by 2? Let's try:

```
[IN]: my_list*2
[OUT]: [1,5,2,7,3,7,8,1,5,2,7,3,7,8]
```

Ok ... that didn't work! It seems like the default "list" behaviour associated with multiplication is to create a new list with the original list repeated N times. If you think about it, this makes sense – recall from our original definition of lists that they can contain any data type, and indeed any combination of data types. If our list contained a mix including strings or any other non-numerical data type, this operation would fail if defined this way. Does that mean we can't multiply every number in a list by 2? No, but it does mean that we will have to utilise what's known as "iterating" to go through the list one by one and replace each value with 2 times itself. We'll get to this later, but for fun, here's a concise way to do it:

```
[IN]: [i*2 for i in my_list]
[OUT]: [2,10,4,14,6,14,16]
```

As it turns out, there's a shorter (and computationally faster) way to apply mathematical operations to every element of a collection. NumPy arrays (defined in the importable package NumPy, which we will talk about soon) obey the following:

```
[IN]: my_array = np.array([1,5,2,7,3,7,8])
[IN]: my_array*2
[OUT]: array([2,10,4,14,6,14,16])
```

Great, so now we've shown that we can perform mathematical operations on entire arrays all at once. In a slightly more subtle point, this is also faster than the previous method shown with strings, because arrays are being treated computationally as matrices, which computers are very good at solving and operating on. So NumPy's libraries are doing linear algebra to apply your mathematical operation to every element of the array, rather than going through and multiplying each element one by one manually.

In astronomy, almost all your data you work with will be in arrays, rather than lists. There are times, when working in your code, that it is more convenient to throw some values into a list. But particularly when dealing with large datasets, you almost certainly will be working primarily with arrays.

Now that we have all these values stored in an array container, how do we get them out?

Array, String, and List Indexing

The process to extract subsets of data from a larger array/list is known as slicing, or indexing. Given a list, array, or string (all 1-dimensional here for simplicity), each entry is assigned an index. By convention (that you may find annoying), this index starts with 0, rather than one (this is true for most programming languages). Below we have a sample list, with the indices for each entry listed below:

List Entry	1	2	4	'cat'	5
Index	0	1	2	3	4

Here, '1' is the 0th entry (or element) in the list, and 5 is the 4th. Let's say then that you wanted to extract the 0th entry from the list, to use for some other coding purpose. The way to slice a variable (of the proper data type) is by typing the variable name, attached on the right with closed brackets and an index number. For example, to extract the 0th element and set a variable for it:

```
[IN]: list_1 = [1, 2, 4, 'cat', 5]
[IN]: x = list_1[0]
[IN]: print(x)
[OUT]: 1
```

If I wanted to select the third entry in the list (keeping in mind you start counting at 0), I would type:

```
[IN]: print(list_1[3])
[OUT]: 'cat'
```

Arrays can be sliced in precisely the same way as lists. Interestingly, strings can also be sliced. So, if we had set:

```
[IN]: var = list_1[3]
[IN]: print(var[1])
[OUT]: 'a'
```

We get the 'a' from 'cat'. Unfortunately, if you have a long integer like x = 123456789, you can't slice through x the way you can with lists, arrays, and strings. What you can do is turn (or cast) x as a string:

```
[IN]: x = 123456789
[IN]: x = str(x)
```

Now that x is a string, you can happily index it:

```
[IN]: print(x[0])
[OUT]: '1'
```

Normally if you try to convert a string like 'cat' to a float or int, Python will spit an error. However, if you attempt to convert a string that only contains numbers, Python can successfully make the conversion. Therefore, we can get the integer number of the 0th element of 123456789 like so:

```
[IN]: x = 123456789
[IN]: x = str(x)
[IN]: int(x[0]) # or float(x[0]) for the float
[OUT]: 1
```

Sometimes we want more than a single value from a list/array/string. There is also a way to slice through multiple indices at once. The format is as follows. Take the previous example of the string '123456789'. Say we want the 0th, 1st, 2nd, and 3rd elements to be pulled, turned back into an integer, and set as the value of the variable H:

```
[IN]: H = int(x[0:4])
```

So instead of a single index in the brackets, we have a start index, a colon, and an end index. Also note, Python will go up to, but not include, the end index given. As a shortcut, if you are starting from the beginning, or slicing from some midpoint to the end, you can omit the 0 before the colon, or the final index after, i.e:

```
[IN]: print(x[0:4])
```

will return the same as:

```
[IN]: print(x[:4])
```

and if you don't know how long an array is but want to index it from its nth element to the end, simply:

```
[IN]: print(x[n:])
```

You can also slice through an array backwards using what are known as negative indices, that is, and index of "-1" refers to the last element in an array/list/string, and "-2" the second to last, etc. An example of indexing from the last to 5th from last element might be:

```
[IN]: print(x[-1:-6])
```

Two-Dimensional Slicing

Astronomical data is often stored in 2D arrays — essentially a large grid or matrix of numbers described by two indices, a row, and a column. (If it helps, you can think of the arrays above as matrices with row length 5 and column height 1, so you only needed to index the column of interest.) Let's say "A" is a 2D array I created with NumPy that looks like this:

```
[IN]: import numpy as np
[IN]: A=np.array([[1,3,4,5,6],[4,5,9,3,7],[9,4,6,7,1]])
[IN]: print(A)
[OUT]: [[1, 3, 4, 5, 6]
       [ 4, 5, 9, 3, 7]
       [ 9, 4, 6, 7, 1]]
```

Notice how Python is handling the list structure. There are three one dimensional lists stacked within an extra set of brackets (like a list of lists). We can slice it with two indices, row, then column. Also, be careful as row then column translates into (y,x), which is the opposite of how we are usually taught to determine ordered pairs of coordinates. For example, to pull the 3 in the second row, we type:

```
[IN]: print(A[1][3])
```

Alternatively, you can use the comma syntax to equal effect. For example, to pull the 6 in the first row:

```
[IN]: print(A[0,4])
```

Try this yourself. What would be the way of slicing to pull the 4 in the last row? Using the same colon notation from above, how would you pull a whole row?
Given a 2D array, you may want to take a chunk of it, either end to end, or somewhere in the middle. The syntax for doing so is a combination of commas and colons. Remember that colons either separate a start and end index or refer to a whole column if no start/end are specified. Let's say you have an image with

1000x1000 pixels, which you are viewing as a 2D array of 1000x1000 values. The following is a list of example slices, from which you can infer how to slice any section you'd like:

```
[IN]: array[350:370,:]
```

takes the full rows 350-370 in the image (fig. 1; left)

```
[IN]: array[:,350:360]
```

takes the full columns 350-360 in the image (fig. 1; centre)

```
[IN]: array[350:370, 350:360]
```

takes the box in the region between/including rows 350-370 and cols 350-360 (fig. 1; right)

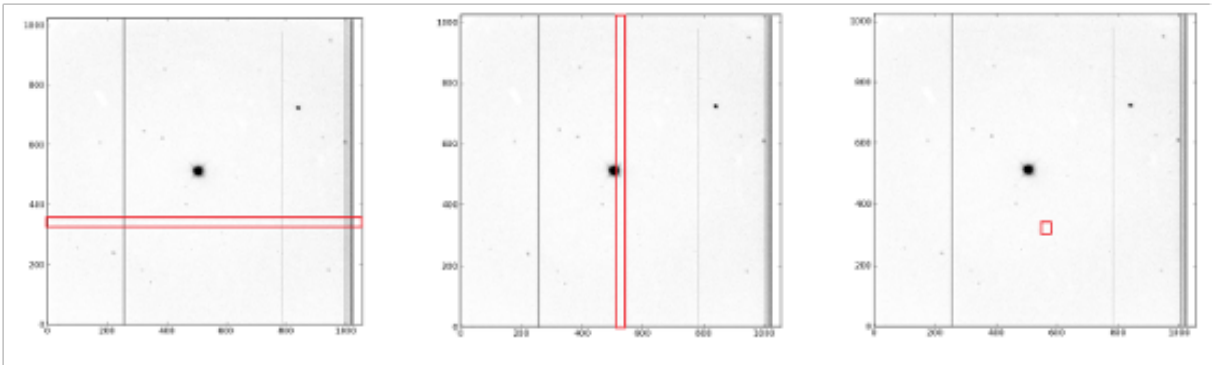

Figure 1: Rows 350 to 370 pulled (left), columns 350 to 360 pulled (Centre), box of rows 350 to 370, columns 350 to 360 (right)

Modifying Lists and Arrays

While we have shown how you can create a list of elements and how to extract and see specific values within them, we are yet to talk about adding and removing, or changing elements of lists and arrays. Say we have a list of integers as follows: [1, 2, 3, 4, 5, 6, 7]. The simplest way to change a value within the list is to set a new value equal to the slice of that list. For example:

```
[IN]: list1 = [1,2,3,4,5,6,7]
[IN]: list1[2] = 'hi'
```

When we print the list one now, we will see that the third element of the list (formerly the integer 3) will have been replaced:

```
[IN]: print(list1)
[OUT]: [1,2,'hi',4,5,6,7]
```

Of course, now that there is a string in our list, we can't do things like sum(list1) and expect to get a proper value. Now let us see how to delete values out of a list. This will involve use of the 'del' command. To demonstrate deleting an element, let's continue using our list1 from above. The following:

```
[IN]: del list1[-1]
```

This will delete the last entry in the list. (Note, a -1 index means the last element in a list or array, and -2 references the second to last, etc). We could also have used forward indexing just fine.
Be careful with this command. Remember that once you delete an entry, the indexes corresponding to all the remaining values get shifted. So you could run del list[0] 3 times and it would continue deleting the new 0th entry in the list. While the principles of what we've used apply equally well to arrays, the syntax of how everything is done will be somewhat different, due to the way numpy.array was created (we will discuss working with NumPy arrays more later). Now, if we want to add a new value to the end of a list (an extremely common task), we simply type:

```
[IN]: list_name.append(new_value)
```

This is possible because 'append' is a method (built-in function) of the list object in Python. Before moving on, let's look at two very basic image (2D array) manipulation commands that might come in handy. Let's go back to our 1000x1000 entry 2D array. There are simple commands for if you want to flip the image vertically and horizontally. For a vertical flip (about the horizontal centreline):

```
[IN]: flip_vert_array = array[::-1]
```

For a horizontal flip (about the vertical centreline):

```
[IN]: flip_hor_array = array[:,::-1]
```

This technique works for 1D arrays as well. Simply use the following to reverse the elements in the list:

```
[IN]: some_list = original[::-1]
```

Finally for this section, let's look at a process known as concatenation. Concatenation is the process of joining two things together end-to-end. We've already seen how to do this for lists, and the same method works for strings, e.g:

```
[IN]: string1 = 'hello'
[IN]: string2 = ' world'
[IN]: finalstring = string1 + string2
```

Note, we could've accounted for the space at the end of string1 instead or made it a separate string. Concatenating arrays takes a little more work, so I'm going to skip that topic until we've covered NumPy in more depth.

Dictionaries

A dictionary is a Python container, like a list or array, but instead uses 'keys' instead of indices to specify elements within the container. That is, the order of elements (values) in a dictionary is irrelevant, and values are retrieved by indexing for the appropriate key. Dictionaries in Python are created using curly brackets, inside which go key-value pairs (colon separated), which themselves are separated by commas, e.g:

```
[IN]: simple_dict = {'key1':value1,'key2',val2}
```

To pull value1 from the dictionary, I would index it as:

```
[IN]: pulled_value = simple_dict['key1']
```

We can also change values in a dictionary, or add new key-value pairs, using this index notation. For example, if we wanted to change val2, we would use:

```
[IN]: simple_dict['key2'] = new_val
```

and to add a new key-value pair, I would simply type:

```
[IN]: simple_dict['new_key'] = new_value
```

While I have chosen my keys to be words within strings, this is not required. I could have chosen keys that were numeric (i.e. 1, 2, 3), which would have worked fine. But the strength of dictionaries is generally that keys hold meaning and are easy to remember because they relate to what I'm placing in the values, which can be any data type. For example, say I wanted to tally how many of different kinds of fruits I have. I might set up a dictionary:

```
[IN]: fruits = {'bananas':5, 'apples':3, 'pears':17}
```

I can now easily query for how many pears I have, as opposed to creating a list [5,3,17] and having to keep track of the fact that pears were the 3rd entry. If I went and bought some mangoes, I could easily add them in via:

```
[IN]: fruits['mangoes'] = 4
```

Dictionaries can be nested inside dictionaries, as can lists, hinting at the rich data structures one can create to house complicated sets of data. That said, keep in mind that the more you nest lists/arrays/dictionaries within each other, the more complicated and irregular the indexing process becomes.

Libraries and Basic Script Writing

You've now seen how you can use a Jupyter Notebook to do basic math, and that there are various data types that come "preinstalled" within Python (e.g., lists, strings, integers, etc). However, once a code requires more sophisticated analytical tools (especially for astronomical processes), it becomes apparent that the vanilla Jupyter Notebook functions are not sufficient. Thankfully, there are hundreds of functions that have been written to accomplish these tasks, most of which are organised into what are called libraries. A library is a maintained collection of functions which can be installed and imported into a Python code to be used. NumPy and SciPy are examples of libraries. Most Python distributions come with a lot of these libraries included, and installing new libraries is generally straightforward.

Four libraries we will review here include NumPy, Matplotlib, SciPy and AstroPy. NumPy is an extremely versatile library of functions to do the things Python can't. For example, while you can create a polynomial yourself (x**2 + 3*x + 1), Python provides no way to make sine and cosine functions. That's where NumPy comes in. Matplotlib, meanwhile, is a library with functions dedicated to plotting data and making graphs. AstroPy is a library with functions specifically for astronomical applications: we will be using it to import fits images (images taken by telescopes), among other things. SciPy is a library that contains special use functions that are often used in science. Since there are thousands of these functions, instead of memorising them all, the best way to learn is to Google or query Stack Exchange for the type of function you are looking for, and you'll find the SciPy or NumPy function you need. The ones you use most often will then become second nature.

Installing Libraries

As mentioned above, most scientific distributions of Python (like Anaconda) come with important packages preinstalled. However, for most smaller packages, like AstroPy, or PyFits, or those for programs you are

using written by other scientists, you will likely have to install them yourself. The easiest way, when available, is to use the conda command. If a library is available via this approach, then installing is as simple as typing:

```
[IN]: conda install NumPy
```

The package will then install if available. If it is already installed, you will likely see something as below:

```
[OUT]: # All requested packages already installed.
```

You should install the following packages before moving on: NumPy, Matplotlib, SciPy and AstroPy. To check if they are pre-installed, type the following:

```
[IN]: conda list
```

Importing Libraries

Given these libraries are not automatically loaded when Python runs, we must import them. As shown below, this can be done in the Python interpreter or as the first few lines of a script.

```
[IN]: import numpy
[IN]: import matplotlib.pyplot
[IN]: import astropy.io.fits
```

Notice that there is a dot notation within some of the imports. This is associated with classes. These libraries are huge and loading all the functions in them is unnecessary if you know what you want. Since pretty much everything you need to plot is within the pyplot sub-library of Matplotlib, we can just import that sub-library. Now that the functions are loaded, you can use them in your code. However, the syntax for using them is slightly different than that of normal Python functions. Because Python needs to know where the function you are calling is coming from, you must first write the library, then the function, using the same dot notation as above. For example, a sin function might be:

```
[IN]: import numpy
[IN]: import x = numpy.arange(100)
[IN]: y = numpy.sin(x)
```

Clearly, writing out numpy all over your code would take forever. Luckily, Python allows us to import the libraries and name them whatever we want for the purposes of our code. Two common examples are shown below:

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

We are already discovering that as the tasks we are trying to handle become more complicated and involve importing libraries, performing said tasks within the Jupyter Notebook terminal environment is unwieldy and inefficient. Therefore, we shall write a program instead.

Writing a Program

So far, we have been working entirely in the Jupyter Notebook interpreter. While this is a quick and easy way to practice with Python, it may not always be suitable for the application. Therefore, we often write

what are known as scripts, or programs. A program is a self-contained list of commands that are stored in a file that can be read by Python. Essentially, it is a text file, with each line being the exact syntax you would have typed into the terminal. Python then opens your program and runs it through the interpreter, line by line. For example, if this is what you did in interpreter before:

```
[IN]: import numpy as np
[IN]: import matplotlib.pyplot as plt
[IN]: x = np.arange(100)
[IN]: y = x**2 + np.sin(3*x)
[IN]: plt.plot(x,y)
[IN]: plt.show()
```

You would type this up in any plaintext text editor (popular examples include Vim, Emacs, Sublime Text, Nova, PyCharm, Atom, etc), and save it as something like 'program.py'. Then to run it, simply open the interpreter (in the same directory as the file) and type:

```
[IN]: run program.py
```

Your plot will be output. Note that if your code doesn't involve any interactive elements, you can also run it from the regular terminal via:

```
~$ Python program.py
```

There are innumerable advantages to writing scripts rather than working directly in the interpreter, most of which are hopefully self-evident. Your code will then be transportable (between computers, people, etc.). Also, you can adjust a single element (or fix a mistake) and rerun your script without having to retype every line (which would be required in the terminal).
How do we know how a function works or what its inputs and outputs are? For now, you've been taking my word for it. But no need! Besides searching online for a package function readme file, we can do this straight within the interpreter. Simply type:

```
[IN]: help(np.genfromtxt)
```

Python will then give you a helpful rundown of how the function works.

Working with Arrays

Earlier, we discussed how you can initialise a list, add to it, replace values in it, etc. We will now repeat using the relevant syntax for NumPy arrays. Here are some ways to initialise a basic NumPy array:

```
[IN]: np.zeros(len_desired) # array of zeros
[IN]: np.ones(len_desired) # array of 1's
[IN]: np.ones(len_desired)*2 # array of 2's
[IN]: np.arange(start,stop,step) # array of integers from start to stop in step jumps
[IN]: np.linspace(start,stop,num) # array of floats with num equally spread values
[IN]: np.logspace(start,stop,num) # array of floats with num log spread values
```

Recall for lists, the syntax for appending was:

```
[IN]: listname.append(newvalue)
```

For arrays, we call the specific function:

```
[IN]: arrayname = np.append(arrayname, new value)
```

If you need to change a value in an array, the syntax is identical to before. Simply set the following to change:

```
[IN]: arrayname[index] = new value
```

To delete values from an array, you can use:

```
[IN]: arrayname = np.delete(arrayname, indices)
```

where indices can be a single index or a range. To insert values into an array, call:

```
[IN]: arrayname = np.insert(arrayname, index, value)
```

and your value will be inserted before the index specified.

If you want to append one array onto the end of another (i.e., concatenate them), you can't use the '+' syntax used for strings and lists, because you'll end up making a new array, the same size as the originals, with each new value being the sum of the two values in corresponding positions in the original arrays. Instead, we need to call:

```
[IN]: np.concatenate((arr1, arr2, ...))
```

to join them together. Alternatively, if you have an array you need to split up, you can use:

```
[IN]: np.split(arr, indices)
```

If you specify a single number, like 3, it will attempt to divide your array into 3 equal length arrays. If you provide a range of indices in order, it will know to split your array at those spots. There are far more fiddlier things you can do with arrays, particularly once you start working with 2 and 3 dimensional arrays. We will touch on that below, but primarily the SciPy documentation and the web are good resources for learning about NumPy array functions.

Conditionals and Loops

We saw earlier how to create programs and run them in Python. That powerful structure allows us to save text files containing coherent sets of Python commands which Python can run for us all at once. As of now, understanding how Python interprets our simple programs is easy: it takes each line and enters it into the terminal. The real power of programming, however, lies in our ability to write programs that don't just contain a list of sequential commands. We can write code that repeats itself automatically, jumps around to different sections of the document, runs different functions depending on various inputs, and more.

We can create programs like this by implementing various conditional statements and loops. A conditional statement begins a defined, separated block of code which only runs if the conditional statement is evaluated by the interpreter to be "true". Essentially, you are telling the computer "only run this block of code IF some condition is true." The condition itself is determined by the programmer. Let us start with some examples of conditional statements. The primary conditional you will use is "if". The syntax for creating an if-statement is as follows:

```
x = 5
y = 7
if 2*x**2 > y**2:
   print('I can math')
```

We start the line with the word "if", which is a special word in Python that tells the interpreter to evaluate the "truth-ness" of the rest of the line, up to the colon (the colon is important, don't forget it). In the case above, the if-statement would print "I can math", because 2 * (52) = 50 > 49. In this case, because x and y were simply defined to be numbers, the condition would always be true, and the print statement would always occur. But most of the time in your code, you have variables which are arrays, or parts of arrays, and the values have been changed in various steps of the code that you can't keep track of. Also note, like for functions, all lines to be considered part of the conditional must be indented one tab.

To create a conditional with an "equals" condition, you must use the strange syntax of the "==" double equals, in the spot where you otherwise had > or <. The reason for the double-equals notation is that in Python, a single "=" sign is reserved for setting the values of variables. As we will see later, the "+=" notation means "set x = x+1" . Some other conditional combinations are "not equal," given by "!=", greater than or equal to, ">=", and less than or equal to "<=".

Conditional	Symbol	Conditional	Symbol	Conditional	Symbol
Equals	==	Greater than	>	Less than	<
Not equals	!=	Greater than/equal to	>=	Less than/equal to	<=

We are not limited to one conditional per statement; we can combine as many as we need (within reason).

```
x = raw_input('Enter a number:')
x = float(x)
y = 15
z = 20
if (x >y) and (x !=z):
  print('Nice')
if (z > x) or (x != y):
  z = x + y + z
```

So here we have 2 if-statements, with the two possible combinations of conditionals, 'or' and 'and'. These statements can be combined indefinitely (for example, if ((a and b and c) and (d and f)) or (g +1>y) demonstrates how you can combine 'and' and 'or's' to suit your needs).

From now on, we will begin dropping new Python commands and code into our examples, and will explain them either in comments in the code, or after the example. In this example, the command raw_input('text') prints 'text' to the screen and waits for the user to enter something. Whatever is entered is stored as a string in the variable x. (So above, if you said, "enter a number" and a user entered a letter, the code wouldn't work). So, using the if-statement we have been able to set off blocks of code to be run only if some combination of conditionals is true. What happens otherwise? Typically, we include an "else" statement following the if block, to determine all other cases.

```
x = raw_input('Enter a number: ')
if int(x) == 5:
  print('Wow, this was an unlikely coincidence.')
else:
  print('Well, that is interesting.')
```

If your 'else' statement contains an if statement as well, you can use the "elif" command, which stands for else if. This saves you the trouble of an extra indent.

```
if x < 0:
  print('Negative')
else:
if x == 0:
  print('Zero')
else:
  print('Positive')
```

Can be condensed to:

```
if x < 0:
  print('Negative')
elif x == 0:
  print('Zero')
else:
  print('Positive')
```

So now we know how to set up a "fork" in our code, to allow it to go in different directions based on various conditions. There is another type of block which instead continues to run the block over and over as long as some condition is met (to be clear, we refer to block as the indented section of code within various loops, conditions, functions, etc). This is known as a while-loop.

The two primary loops in Python are the while and for loops. A while-loop is a set off block of code that will continue to run sequentially, over and over, so long as a certain condition is met. A for-loop is a set off block of code that contains a temporary variable known as an iterator and runs the block of code over and over for different specified values of that iterator.

While-Loops

Let's begin with a simple example of a while-loop:

```
x = 100
while x > 5:
  print(x)
x = x -1
```

What's going on here? We initialise x to be some value. The next line of code read by the interpreter (remember it goes line by line) tells it that if x is greater than five, keep running the indented code over and over. The indented code in question prints x, then sets x = x-1. Eventually, after 95 times through the loop (and 95 prints), x would become 6-1 = 5, which would no longer satisfy the while statement. The interpreter would then move on to the next line of code in the document. This brings up a very important point: you can see that if we had not included the "x = x-1" part of the code, x would never end up being 5 or less. Thus, your code would hang in this loop for all eternity. Luckily, if you find yourself in this situation, there is hope besides frantically shutting off the computer. Python interpreters have built in keyboard shortcuts to interrupt and stop your code from running. When using while loops, be sure you have included something within the loop that will eventually cause it to end. As a precaution, most programs that are more involved have special if statements within the while loop that will automatically break out of the while loop if, say, a certain threshold of time has passed. The rules for the conditionals themselves (the x>5 above) are the same as for if.

For-Loops

For-loops are one of the most powerful tools in Python. What they allow us to do is write a block of code that's like a template. It has the code we want to run, but without defining exactly "on what" the code

acts. We then initialise a for-loop, picking a range of values, variables, etc., to plug into those designated spots in our block of code. The simple for-loop:

```
arr = [1,2,3,4,5,6,7,8,9,10]
for i in arr:
  if i %2 ==0:
print i
```

would print 2,4,6,8,10 (the even numbers). The % sign means "modulo," and the conditional would read "if i divided by two has a remainder of 0:". The letter i in this loop is a generalised iterator- when you type "for i in arr" you are telling the computer to run the block of code, replacing i in the block with the first second, third, etc. element in the array. (you could use any character/combination of characters for i, but i is standard practice (followed by j, and k if necessary).

The point of for-loops is that they are as generalisable as possible. In the above example, the array "arr" could be replaced with any variable that is an iterable data type. You could say, "for i in range(15)" to have it plug the numbers 0 through 14 into your block of code, wherever a variable 'i' appeared. you could even iterate over a string, and it would plug in the elements of the string (as single character strings) into your block of code.

One common iteration practice is to iterate over an ascending list of numbers equal to the length of a certain array. In this situation you could use "for i in range(len(array)-1)", where "array" is your array and len( ) is the command for returning the number of elements in an array, list, or string. The minus one is needed because the nth element of an array, list, or string is has an index of n-1.

Nested For-Loops

Just briefly, I'd like to mention that you can in fact nest multiple for-loops together, if you need to iterate over more than one value in your code. This often happens when dealing with two-dimensional arrays.

```
for i in range(len(x)-1):
  for j in range(len(y)-1):
    if arr[i,j]<1500.:
      arr[i,j]=0
```

In the above example, x would be a variable representing the x coordinates in the array, with a similar deal for y. This particular block of code would run through every combination of i, j to hit every spot on the 2D array, and if the value at any given point was below the 1500 threshold, it would just set that element to be 0.

This might be a good spot to point out that the above code isn't the most efficient way to accomplish its task. For example, the following would be more efficient:

```
array_name[np.where(array_name < 1500)[0]] = 0
```

This is due to the fact that loops always involve performing a task over and over many times, while some NumPy functions leverage linear algebra to act upon entire arrays simultaneously. A lot of the time spent after learning the basic building blocks of programming is focused on determining the most efficient ways of completing a programming task, minimising either run time, memory usage, or both.

Opening and Writing Files

Now that you know how to concatenate strings and generate for-loops, we can cover the file opening/writing process. There are several ways of opening data files in python. Python itself has a built in mechanism

for opening/writing files, and NumPy also has support for file handling. To open a file in python's interface, we type `file1 = open('filename.txt','w')` where 'w' indicates we plan to write to the file. (We could instead use 'r' for read only, or 'a' for appending to a file that already contains data. Below is an example you can try:

```
[IN]: file1 = open('file.txt','w')
[IN]: file1.write('this is a file')
[IN]: file1.write(' this is not a drill')
[IN]: file1.close()
```

The close statement above tells python to close and save the file to the hard disk (Note: in Google Colab, click on the folder icon on the side of the screen to access your list of files, which you can upload/download).

NumPy also has a file opening/writing framework that is often useful to use. The two we will discuss here are `np.loadtxt` and `np.genfromtxt`. These are useful tools because they have many specifiable options, and load your data straight into NumPy arrays.

Let's say we load the following file:

```
[IN]: data = np.loadtxt('filename.txt')
```

and assume the file has three columns: 'times', 'positions', and 'velocities'. These would all be stored in data, and could be singled out as such:

```
[IN]: data = np.transpose(data)
[IN]: times = data[0]
[IN]: positions = data[1]
[IN]: velocities = data[2]
```

Because of the way columns/rows work in Python, data in multiple columns are read in as individual data pairs. On the other hand, simply running an `np.transpose` on them sorts them to be 3 long separate arrays with all the times, all the positions, and all the velocities split.

Oftentimes data files have headers and footers – text that tells you what data is stored in the file. Of course, we don't want to try to read these into python as our data. For example, to skip a 5 line header and 3 line footer text, you can use:

```
[IN]: data = np.genfromtxt('file.txt', skip_header=5, skip_footer=3)
```

This function is pretty versatile, and also has options for skipping columns, specifying data types, etc. A highly beneficial option is the ability to automatically assign arrays a name that's the same as the header. For example, if our file contained data with a single header labelled 'times', 'positions', and 'velocities', we could use:

```
[IN]: data = np.genfromtxt('file.txt', names=True)
```

I could return the 'positions' array by simply typing:

```
[IN]: data['positions']
```

Believe it or not, that's all there is to basic programming. By cleverly combining for loops, while loops, and conditional statements, we can do a lot of powerful analysis. While there is a lot more to Python (for example, you can introduce classes and object-orientation), this is all you need to do the majority of

scientific coding. What is missing in the above descriptions is the multitude of Python and NumPy functions you will need to use along the way. Details of these will be provided in individual projects as required.

In the meantime, you are encouraged to go complete Parts 1 and 2 of my "Introduction to Python for Astronomers Notebook" activities. Download the files provided and run the notebooks from either QUT's Jupyter Notebook Hub (jupyter.qutanalytics.io/), Google Colab, or from your own machine (IDE required).