Introduction to Python

Python is a high level programming language. This means that you don't have to worry much about the details of the computer when you write your code. Python takes care of that in the background, so you can write your code in a more general abstract way. This usually means that you can express concepts in python in fewer lines of code than in C or C++, and that the code is more intuitive to write and read.

Python code also isn't generally compiled like C code is. Usually python code is written as scripts which are interpreted and run on the fly by the python interpreter.

This tutorial covers:

- Introduction to Python and iPython
- Dictionaries
- Data types
- Writing functions
- Some useful Python modules:
    - csv
    - numpy
    - matplotlib
    - scipy
- Writing scripts

Notes on using this tutorial:

- This tutorial was designed for you to type the ipython commands into an ipython session, and run them line by line. Try resist the urge to cut and paste!
- This tutorial was also designed for you to play around with. Use the help to figure out how functions work, and play around with them! Try your own variables and examples, as well as the ones given in the tutorial, as you go along.

Introduction to Python and Ipython

Ipython is an interactive shell which runs python. It is a useful way to learn python, and to test and explore python code. Lets start out with some basic python commands to get used to using the ipython shell.

Start up ipython by typing in a terminal:

```
> ipython
```

Then the ipython shell starts, and now we can start typing python commands.
Lets start with something simple:

```
> print ‘Hello world!’
```

Now lets try something numerical:

```
> 6+8
> l=6
> m=8
> l+m
```

Lets make a list and explore how we can use it. Python uses zero-based indexing, so list and list indices start at 0.

```
> mylist = [6,1,3,0,5]
> mylist[0]
> mylist[3]
> mylist[-1]
> mylist[1:3]
> mylist[:2]
> mylist[:-1]
> mylist[0:4:2]
> mylist[2] = 14
> mylist
```

The first three lines above are examples of indexing. The last three lines are examples of list slicing, which is taking sections of lists. The syntax is start:end+1:step. Where the start or end+1 numbers are missing, it means “from the beginning” or “to the end”.

Before we move on lets looks at one more handy bit of functionality! Go to your iPython terminal, and press the up key. It should bring up the previous command you typed. If you press the up and down arrows you can scroll through your command history - very useful for re-running things! (The unix terminal has the same functionality. Now try:

```
> p<up arrow>
```

This should bring up the “print ‘Hello world!’” command you typed earlier. Useful if you want to repeat a command you typed a while ago - type the first few characters, then the up key, to bring up your old command!

Python includes a standard library of modules that provide all sorts of functionality. To use a module, we import it. As an example, lets look at the module math:

```
> import math
```

Now one of the great features of ipython is that is allows us to easily access documentation. Try:

```
> math?
```

That tells us what math is. Now lets see what the math module contains:
(The notation <tab> means press the tab key on your keyboard)

```
> math.<tab>
```

That gives us a list of all of the functions or methods provided by math. Lets have a closer look at a couple:

```
> math.cos?
```

So cos is a function that we can call. Lets try it.

```
> math.cos(0.5)
```

Now lets have a look at pi:

```
> math.pi?
```

So pi is just a number, meaning that we don’t call it like a function. Lets try out pi:

```
> math.pi
```

… which gives us what we would expect!

Python has a handy feature called list comprehension, which lets us create new lists easily.

```
> from math import cos, pi
> cos_list = [cos(x*pi/6) for x in range(7)]
> cos_list
```

Python also has great plotting capabilities. Lets try make some simple plots. We will use the range() function, which is one of the built-in python functions. Check what it does with:

```
> range?
```

So lets make some x-values for an example plot:

```
> x = range(-360,360)
```

Say those x values were in degrees, and we want to change them into radians. Lets use our list comprehension skills:

```
> x_rad = [2.*pi*i/360. for i in x]
> y = [cos(j) for j in x_rad]
```

Now lets plot out x versus y values!

```
> from pylab import plot, show
> plot(x_rad,y)
> show()
```

Dictionaries

Python provides a data type called a “Dictionary” which can be incredibly useful if you get used to using them. What python terms a dictionary, some other languages term associative arrays or hash tables.

Dictionaries are collections of items that have a "key" and a "value". They are like lists, except instead of having an assigned index number, you make up the index.

Dictionaries are unordered, so the order that the keys are added doesn’t necessarily reflect what order they may be reported back.

Lets try working with dictionaries in ipython. We use {} curly brackets to construct a dictionary. For each dictionary item we provide a key and a value, with a colon placed between key and value (key:value), separated by commas. Each key must be unique and each key can be in the dictionary only once.

```
> animaldict = {"animal":"dog","name":"axle","owner":"nadeem"}
```

So here we made a dictionary with three items. The keys for the items are the strings “animal”, “name” and “owner”, and their values are “dog”, “axle” and “nadeem”.

We look up the value associated with a key using square brackets. Try:

```
> animaldict[“owner”]
```

Dictionary keys can also be integers:

```
> mydictionary = {0:"first",1:"second",2:"third"}
```

In which case, indexing a dictionary and array looks pretty similar:

```
> mylist = ["first","second","third"]
> mydictionary[1]
> mylist[1]
```

One of the useful feature of a dictionary is that it can be easily looped over:

```
> for indx in animaldict:
        print "The ", indx, " is ", animaldict[indx]
```

To get all of the keys or values in a dictiorary, we can type:

```
> animaldict.keys()
> animaldict.values()
```

And we can change values in the dictionary, or add to them, using they keys:

```
> animaldict[“owner”] = “laura”
> animaldict[“breed”] = “labrador”
```

And check what our dictionary contains now:

```
> for indx in animaldict:
        print "The ", indx, " is ", animaldict[indx]
```

After the last line, just hit enter again to tell python you are finished typing.

To delete an entry:

```
> del animaldict["breed"]
> for indx in animaldict:
        print "The ", indx, " is ", animaldict[indx]
```

Dictionaries have other useful functionality. Try explore the dictionary functionality by typing

```
> animaldict.<tab>
```

And, for example, check out:

```
> animaldict.has_key?
```

Data types

Python is a dynamically typed language, which means that you don't need to specify what type a variable is (like in C, for example). Python will just set the type to match what you have put into the variable, and store the type of the object with the object. Different types of objects can have different operations performed on them.

Python comes with a number of built in data types. We'll just have a look at some of the numerical types and sequence types here.

Numerical data types:

● int	- integers
● long	- long integers
● float	- floating point numbers
● complex	- complex numbers

In ipython, try:

```
> x=8
> y=3
> x/y
```

Thats not right! What's happening there? Let do some investigation...

```
> type(x)
> type(y)
```

So x and y are both int types (integers), so when we operate on them they give an int result. We can change them into float types in a couple of ways.

```
> x=8.
> type(x)
> x=8.0
> type(x)
> x=8
> type(x)
> x=float(x)
> type(x)
```

Now try

```
> x/y
```

Should look better this time!

Each data type has methods that we can perform on it. We can check what they are using our ipython tab completion trick.

```
> x=8
> x.<tab>
> x=8.
> x.<tab>
```

See how the int and float data types have different methods available to them.

Now lets look at some of the sequence data types:

- str - string (a string is represented as a sequence of 8-bit characters)
- list - sequence of mutable objects
- tuple - sequence of immutable objects

Lets have a look at strings first. In python, strings are lists of characters.

```
> x='hello everyone'
> type(x)
```

We can index strings like lists:

```
> x[0]
> x[1:3]
```

Lets check what methods we have available for strings, and try some of them out:

```
> x.<tab>
> x.upper()
> x.islower()
> x.isupper()
```

One of the useful string methods is split. Lets check out the help:

```
> x.split?
```

We often find ourselves with strings consisting of information separated by a specific character. For example

```
> y='one, two, three, four, five, six'
> z=y.split(',')
> z[0]
```

The default behaviour of split is to split around spaces. So if we try

```
> z=y.split()
> z[0]
```

Now lets take a look at lists and tuples. We have seen some lists before, at the beginning of this tutorial. Lists and tuples are both sequences of objects, but lists are mutable and tuples are immutable. This means that lists can be changed after they have been created, but tuples can't be changed after they have been created. We create tuples with round brackets, rather than the square brackets we use for lists.

```
> x = (1,2,3,4,5)
> y = [1,2,3,4,5]
> type(x)
> type(y)
> y[3]
> y[3]=0
> y
> x[3]
> x[3]=0
```

The last step above should give you an error! Why?

Lets check the methods available to lists and tuples:

```
> x.<tab>
> y.<tab>
```

Lets try out one of the list methods:

```
> y.reverse?
> y.reverse()
> y
```

Writing functions, part 1

In this section we look at how you can define and use your own functions in python. Typing some of the longer functions into ipython can be a bit tricky. Indentation matters in python, so we must make sure that each line is typed the correct number of spaces before it.

The longer example function at the end of the section can be run in ipython, but if you are making lots of typos and beginning to despair, skip ahead to “Writing Python Scripts” and try putting the functions into small scripts to run.

Lets look at the format for defining a function in python, using a couple of examples:

```
> def printName(name):
        print ‘Hello, my name is ‘, name
        return
```

We start the function off with the keyword ‘def’, which stands for define. We then have the function name, followed by a list of arguments in brackets (or just empty brackets if there are no variables passed to the function), then a colon.

The body of the function is indented - this is important in python, you have to indent consistently! And the function is ended with the keyword return. This function is not actually returning anything, so we could leave out the return.

Now after defining the function above, we can use it:

```
> printName(‘Nadeem’)
> myname='Laura'
> printName(myname)
```

Now lets try a function that returns something:

```
> def myAverage(a,b,c):
myAve = (a+b+c)/3.0
return myAve
```

And lets use this function:

```
> myAverage(6.0,5.0,1.0)
> y = myAverage(1.0,2.0,3.0)
> y
```

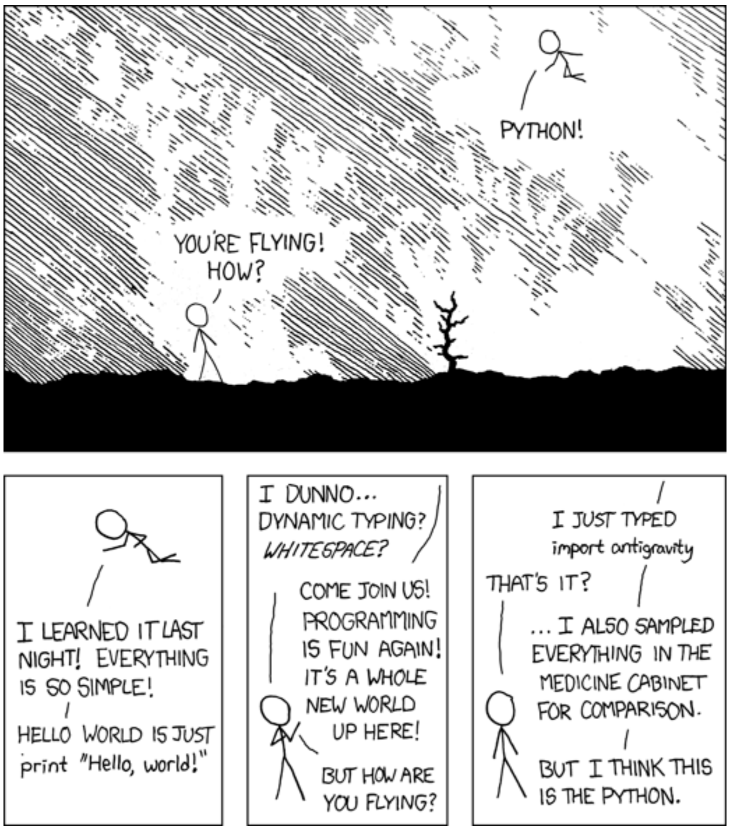
We will come back to functions later. For now, lets have a look at some useful python modules.

Some useful python modules

One of the great things about python is that it has so many useful modules, which add all sorts of functionality. And you get all this great functionality by simply importing the module, as you saw in the introductory section.

Some useful python modules:
csv
math
numpy
matplotlib
pylab
scipy
astropy
pyfits
h5py

Lets have a look at a few of these.


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

Module: csv

```
> import csv
> csv?
> csv.<tab>
```

Lets try import a csv file into python using this module. Lyts just check the help first:

```
> csv.reader?
```

Okay, so now lets make a little file to test things with. Save the following text as a file “example.csv”

```
"laura", 1, 7.8
"tomm", 3, 22.2
"tomb", 15, 0.7
"ludwig", 88, 3.1
"mattieu", 2, 6.1
"simon", 8, 155.5
```

(You can create the file using an editor like gedit, or any other editor you prefer. Save it in the directory you are runnning ipython in).

Then open the file:

```
> eg_file = file("example.csv")
> csvreader = csv.reader(eg_file)
> for row in csvreader:
        print row
```

See how the csv reader automatically separates the data out, using the commas to separate the columns?
Rather than just printing the data, we can perform operations on the data too

```
> eg_file = file("example.csv")
> csvreader = csv.reader(eg_file)
> for row in csvreader:
        print row[0], ' ', float(row[1])/float(row[2])
```

Notice how we had to change the type of the numerical data values before we divided them. The csv reader just reads in all of the values as strings, and we can't perform numerical operations on strings. So we change them into floats.

Module: numpy

Numpy is Python's scientific computing package. It is too large to cover properly here, so we will just introduce some of the numpy functionality.

```
> import numpy
> numpy.<tab>
```

One of the main features of numpy is the numpy array data type. Lets use a numpy function called arange to make an array. We can import the arange function from numpy in a few ways, and each of them has pros and cons. (The text behind the left arrows << is describing the line for your benefit, don't type it into ipython!)

```
> import numpy              << import numpy
> numpy.arange?             << then we can use arange out of the numpy we have imported

> import numpy as np        << import numpy and give it the alias np
> np.arange?                << now we use arange out of numpy, using the shorter alias

> from numpy import arange  << here we just import arange from numpy
> arange?                   << we have imported arange explicitly, so we don't need to call it out of
                               numpy

> from numpy import *       << here we import everything out of numpy
> arange?                   << we have imported arange explicitly (along with everything else in
                               numpy!) so we can use it directly, as above
```

Lets use the aliasing method.

```
> import numpy as np
> np.arange?
> x=np.arange(1,15,2)
> x
> type(x)
```

Notice that x is a numpy ndarray. We can also turn lists into numpy arrays:

```
> y=[4,3,2,5,6,7]
> y
> type(y)
> z=np.array(y)
> z
> type(z)
```

Numpy has its own set of types, and you can tell what type a numpy array holds using the method dtype. We can also explicitly tell numpy what type to use, when we create an array using a numpy function.

```
> z.dtype
> x.dtype
> h=np.arange(1,15,2,dtype=np.int32)
> h
> h.dtype
> k=np.arange(6,dtype=np.complex)
> k
```

We have a couple of ways to check characteristics of our array. For example, what if we are checking for array values greater than 3:

```
> h[np.where(h>3)]
> h > 3
```

Numpy is a very powerful package, with way too many functions to go into here. You will discover more amazing features of numpy as you use it!

Module: matplotlib

Matplotlib provides all sorts of plotting functionality. We will just try some basic examples here, but matplotlib can do very advanced plotting. So you can do more in-depth exploring of matplotlib when you need to make more complex figures.

Lets start by making some data for a simple plot:

```
> x = np.arange(15,30,0.1)
> m = 2.5
> c = -28.0
> y = m*x + c
> y
```

Our x variable is just a numpy ndarray, because we created it with the numpy function arange. Notice how when we performed mathematical operations on it it performs the operation on each element of the array, to form a new array. This is one of the great features of numpy.

As an aside, before we get to plotting, lets just use this opportunity to see how this would differ for a list:

```
> z = [15., 15.1, 15.2, 15.3, 15.4, 15.5, 15.6, 5.7]
> 2.5*z
> 2*z
```

So numpy is giving us some great functionality that lists don't give us. Yayi numpy!

Now lets get back to plotting.

```
> import matplotlib.pylab as plt
> plt.plot(x,y)
> plt.show()
```

There is our plot! Notice the buttons below the plot – you can zoom in, pan around, go back to the original size, and save the figure.

Now close that figure, and lets create another figure with some additions, like axis labels.

```
> plt.plot(x,y)
> plt.xlabel('x axis')
> plt.ylabel('y axis')
> plt.xlim([15,25])
> plt.show()
```

How about plotting two curves on the same plot? Lets try a second order polynomial.

```
> z = 0.5*x**2 - 25.0*x + 315.0
> plt.plot(x,y)
> plt.plot(x,z)
> plt.show()
```

So here we are plotting y versus x, and z versus x. Notice how python automatically makes the two curves different colours. We can control the colours and the line styles when we create the plots. Lets check the options available to us:

```
> plt.plot?
```

```
> plt.plot(x,y,'g+')
> plt.plot(x,z,color='red', linestyle='dashed', marker='o',linewidth=3)
> plt.xlim([18,20])
> plt.show()
```

Notice that the green x-y plot now only has the data points plotted as crosses, while the red x-z plot has the data points plotted as red circles, with a dashed red line.

We can create a legend to label the different lines on our plot.

```
> plt.legend?
```

The help tells us that there are a few different ways to label our lines for a legend. Lets try one way:

```
> plt.plot(x,y,'g+',label='first order polynomial')
> plt.plot(x,z,color='red', linestyle='dashed', marker='o',linewidth=3,label='second order polynomial')
> plt.legend()
> plt.show()
```

You can experiment with the different ways of creating plots and adding labels and legends and grid lines and so on to the plots, and controlling their parameters such as positions, sizes and fonts. Matplotlib can also be used in more sophisticated ways to make more complex plots. The matplotlib webpage is a great resource for all of the matplotlib functionality and examples of how to use it.

Matplotlib challenge:

Reproduce Figure 2 from the paper:
Li, H. Z.; Xie, G. Z.; Yi, T. F.; Chen, L. E.; Dai, H., The Spectral Energy Distributions of the Fermi Blazars and Connection Among Low-Energy Peaked BL Lacertae, High-Energy Peaked BL Lacertae, and Flat Spectrum Radio Quasars, ApJ, 709, 1407

The data is available from Vizier, but for convenience it is saved in the files:
li_fsrq.txt, li_hbl.txt, li_lbl.txt

Tips:

- Use numpy.loadtxt to import the data.
- The first column of a two-dimensional array can be extracted using the index notation myarray[:,0], the second column with myarray[:,0], etc.

Module: scipy

Like numpy, scipy is a powerful module with many useful functions. Scipy provides scientific tools to supplement Numpy. We will just look at the curve fitting function and a smoothing function.

```
> import scipy.optimize as opt
> opt.curve_fit?
```

So for this example, lets first make some data to fit. Lets just fit a straight line, to start off with something simple. So lets make a straight_line function:

```
> def straight_line(x_values,a,b):
return a*x_values + b
```

Now lets make some fake data. To do this we will make some straight line data, then add noise using the numpy function randn, which gives random floats from a Normal distribution of mean 0 and variance 1

```
> x = np.arange(15,30,0.1)
> m = -4.5
> c = 110.5
> y = straight_line(x,m,c)
```

Lets just plot this quick to see all is well:

```
> plt.plot(x,y)
> plt.show()
```

Now lets add the noise:

```
> np.random.randn?
```

So lets say we want to add random numbers with a variance of 5 to the perfect line we already have. Lets make an array of the same length as our x array, and fill it with these random numbers, then add it to the y array (we could do this in other ways too – you can try some different ways!)

```
> x_len = len(x)
> y_noise = 5.0*(np.random.randn(x_len))
> y_noise
> y_sim = y + y_noise
```

Lets plot the new noisy data and the original y data just to check:

```
> plt.plot(x,y,'b-')
> plt.plot(x,y_sim,'r*')
> plt.show()
```

Now lets do the fit and see if we can recover parameters close to the m = -4.5 and c = 110.5 we used to create the noisy data! The help for curve_fit says the outputs it gives is popt, pcov. The popt output will be the fitted variables – in our case, m and c. So lets see what we get

```
> popt, pcov = opt.curve_fit(straight_line, x, y_sim)
> popt
```

Looks good!

Now lets try a smoothing function. Lets make some fake data to start with again, and add some spikes to it. Lets use the same line we had above as our starting point.

```
> x = np.arange(15,30,0.1)
> m = -4.5
> c = 110.5
> y = straight_line(x,m,c)
> len(x)
```

We see that x has a lengths of 150. So lets just randomly, manually, add a few spikes over the length of y.

```
> y[30] = 40.0
> y[50] = -25.0
> y[60] = -20.0
> y[100] = 45.0
> y[135] = 6.0
```

Now lets smooth this data.

```
> from scipy.ndimage.filters import gaussian_filter
> gaussian_filter?
```

Lets try a standard deviation of 3 for the Gaussian smoothing kernel.

```
> z = gaussian_filter(y,3)
```

And lets plot it.

```
> plt.plot(x,y)
> plt.plot(x,y)
> plt.show()
```

You can see that the spikey data we put into the filter has been smoothed.

Writing Python Scripts

iPython is a great tool to explore and test, but often we will want to write longer chunks of python code that we would like to run and re-run, maybe using different data or different parameters. For this we would write a script - a piece of Python code that we run as a unit.

Lets make an example script from some of the code we have used in this tutorial. Save the following code in a file named first_script.py:
(You can create the file using an editor like gedit, or any other editor you prefer.

```
import matplotlib.pylab as plt
import numpy as np

def straight_line(x_values,a,b):
    return a*x_values + b

x = np.arange(15,30,0.1)
m = 2.5
c = -28.0
# using the straight_line function
y = straight_line(x,m,c)
z = 0.5*x**2 - 25.0*x + 315.0        # making a second order polynomial curve

plt.plot(x,y,'g+',label='first order polynomial')
plt.plot(x,z,color='red', linestyle='dashed', marker='o',linewidth=3,label='second order
polynomial')

plt.xlabel('x axis')
plt.ylabel('y axis')

plt.legend()
plt.show()
```

There are a few ways to run this script. From inside ipython, we can type:

```
> run first_script.py
```

From the unix command line we can type

```
> python first_script.py
```

Or we can tell the computer that we want to run the script with python, inside the script, by adding this line as the first line of the script:

```
#!/usr/bin/python

import matplotlib.pylab as plt
import numpy as np

etc.
```

Then to run this script from the unix command line we need to make it executable:

```
> chmod a+x first_script.py
```

And then run it:

```
> ./first_script.py
```

A few general things to note:

- We usually use the extension .py for python scripts
- Just as a matter of convention, we usually import all of the modules at the beginning
- Python cares about consistent indentation. So if you start a for statement or if statement or something else that requires indentation, stick with the same level of indentation throughout.
- We can add comments to python code using # characters – you can see a couple of examples of this in the script above.
- Notice how we used a function to create the line in our script above. Functions are especially useful when you using the same set of commands many times. Try changing the script above to have a few different straight lines on the plot, with different slopes and intercepts, by re-using the straight_line function. You can also try adding other functions to make other curves – how about a cos_curve function?

Writing functions, part 2

Lets take a deeper look at writing functions.

We are going to write some longer functions in this section. All of this can be done in ipython. But if you are making lots of typos and beginning to despair, you can put the functions into small scripts to run.

First lets go back to our name printing function:

```
> def printName(name):
        print 'Hello, my name is ', name
        return
```

> We've seen what happens if we run the function with a variable

```
> printName('Maria')
```

What if we call the function with no variable?

```
> printName()
```

It tells us that we need to give it an argument. In some situations, we will want to have a default argument set:

```
> def printName(name='Mutsa'):
        print 'Hello, my name is ', name
        return
```

Now try using our new function with the default:

```
> printName('Minoshni')
> printName()
```

So if we use it without any argument, it uses the default, otherwise it uses the argument we give to it.

The function we have written so far all return nothing. We can also create functions that return variables. For example:

```
> def addList(a,b):
        out = []
        for i in range(len(a)):
         out.append(a[i]+b[i])
        return out
```

Now try it:

```
> k=[1,2,3]
> l=[10,11,12]
> addList(k,l)
> d = addList(k,l)
> d
```

Try changing the indentation of the return statement:

```
> def addList(a,b):
        out = []
        for i in range(len(a)):
         out.append(a[i]+b[i])
         return out
```

```
> addList(k,l)
```

What is happening here?
Can you think of ways to make this function better? (What if we are adding two lists with different sizes, for example?)

We can also define variables that have a variable number of arguments, using *arguments and **keyword arguments, which are usually abbreviated *args and **kwargs. The *args variable takes a list of any extra variables we pass to a function. For example:

```
> def printShopping(shop,*args):
    print 'Please go to', shop, 'and buy:'
    for item in args:
        print item
    return

> printShopping('Pick and Pay','milk','bread','toothpicks')
> printShopping('Mr Price','shoes')
```

The variable args is a list of the variables that come after shop. For our first function call above, args = ['milk','bread','toothpicks']. So we can use it as a list.

The **kwargs argument, on the other hand, takes keyword-variable pairs and puts them into a dictionary.

```
> def printShopping(shop,*args,**kwargs):
    print 'Please go to', shop, 'and buy:'
    for item in args:
        print item
    print 'And then please go pay the bills:’
    for item in kwargs:
        print item, 'for’, kwargs[item]
    return

> printShopping('Mr Price','shoes',electricity='$100',cellphone='$20')
```

So kwargs is a dictionary, and kwargs = {'electricity': '$100', 'cellphone': '$20'} for the function call above. We can use normal dictionary operations on it.

Helpful References

This link provides very useful basics in Python:
http://www.astro.ufl.edu/~warner/prog/python.html

The python website offers its own python tutorial:
http://docs.python.org/2/tutorial

There is a useful introductory section of the tutorial here:
http://docs.python.org/2/tutorial/introduction.html

We also recommend participants to follow the online python tutorial at:
http://www.codecademy.com/tracks/python

Tips for good coding practice:
http://queue.acm.org/detail.cfm?id=2063168

Exercises

Exercise 1
a) Make an array of 10000 gaussian random variables (Hint: numpy.random.randn)
b) Plot the probability distribution of the numbers (Hint: numpy.histogram)
c) Fit a Gaussian to the probability distribution (Hint: scipy.optimize.curvefit)

Exercise 2
a) Download an image from the SUMMS catalogue, in fits format
b) Import the image into python (Hint: pyfits.open)
c) Smooth the image with a kernel width of 4 (Hint: scipy.ndimage.filters.gaussian_filter)