Introduction to Basic Python Workshop

T.M.Kinnear & Leon Schoonderwoerd

Contents

This Workshop is designed to outline the approach to the use and implementation of Python in your scientific research. It is a powerful tool for interacting with numerical simulation and data processing. It is portable and easy to adapt to specific usage and requirements. These are some of the features which have made it a successful language, highly used and widespread.

Python would be described as a high level language. We can provide code telling the computer what to do in fairly general terms, and the bits of the language and the hardware of the computer are already configured to manipulate that code into the actions necessary to provide the intended result.

It is also an 'interpreted' language. For some programming languages, all of the code must be written in advance, and then another program called a 'compiler' takes that code, and converts it into the actual instructions for the computer in one block. Interpreted languages don't do this in one go; every time a line of code is encountered, it compiles it there and then, and gives just that to the computer, then finds whether there is more code to look at. This is generally a lot slower than a code processed by a compiler, as there is time 'wasted' sending things back and forth, and doing things multiple times which need only have been done once if it knew how all of the code fits together. On the other hand, it means a dynamic interface can be provided, where you can type commands as you need them and have the computer process them as you decide what to do, seeing what results are along the way.

The most basic standard 'program' that one tends to create as a learning tool is the so-called 'hello world' program. It's exclusive purpose is to, one way or another, produce the words 'Hello world' on the screen, and then end. This is accomplished in Python with a single function call.

```
print('Hello world')
```

Python can be used both interactively, with a 'command line' for typing lines of code as you go, and in a batch mode, where a file containing many lines of code is given to the interpreter. We will be looking at both uses, starting with the interactive mode.

Note

For future reference, any lines of code presented beginning with three 'greater than' symbols, >>>, indicate typing commands into the Python prompt in interactive mode. You do *not* need to type the 'greater than' symbols.

Python 2.X Difference

Note that the above 'hello world' example would be slightly different in Python 2.X versions. `print` is exclusively a function in Python 3.X but could be used as a command word in Python 2.X, as follows:

```
print 'Hello world'
```

1 Fundamentals

1.1 Variables

A variable is much like that used in mathematics, it is a kind of placeholder representing non-specified data. In maths, it almost always represents a number. For example x within the function $y = m\,x + c$ doesn't specify what value x is, but when you choose or otherwise know a value for x the formula can make use of that value. In programming, a variable can be used to store different kinds of data, different types numbers, text, 'logical' or 'boolean' values, etc.

Task

In the Python console, try:

```
>>> a = 'hello world'
```

then

```
>>> print(a)
```

you should see that the result of the `print` command is the value '`hello world`' being displayed. The first command is an 'assignment' operation, using the assignment operator '='. The assignment operator takes the thing to its right (the `hello world` in this case) and stores it in the thing on the left (the `x` in this case). In some languages, you have to define all of your variables and their types in advance; so you would have a line of code stating the `x` will be a variable, and it will be allowed to store integer numbers, for example. PYTHON is 'dynamically typed', you don't have to specify in advance, it will create new variables and set what type of data they contain as it goes.

We can find out what type of data `x` is storing with the following function:

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

it should display something on the next line along the lines of: "`<class 'str'>`"; this indicates that the nature of the thing stored by that variable is a type of data named 'str', which stands for 'string' (i.e. a string/sequence of characters). A variable's type can be replaced as needed depending on what is to be stored.

Task

Try

```
>>> x = 2
>>> print(x, type(x))
2 <class 'int'>
>>> x = "Hello world"
>>> print(x, type(x))
Hello world <class 'str'>
```

Try to see what can be stored and how it refers to the data type necessary to store it.

Variable Types

There are several main variable types to consider; there are more than this, but these fulfil most roles we will need:

Name	Short name	Description
Integer	`int`	Any integer value, positive or negative, e.g. `3`, `8`, `-30`
Floating point	`float`	Values with a 'decimal[1]' point e.g. `-2.45`, `2.0`
Boolean	`bool`	Might also be termed a 'logical' value, it has only two 'values', true and false, notated as '`True`' and '`False`' with that capitalisation.
String	`str`	A string is a sequence of character symbols of any form, it can include numerical digits, but they are not interpreted as numbers. Defined by delimiting inverted commas or quote marks, e.g. `"I am a string 123"`, or `'I am not a string (I am really)'`.

[1] Not *actually* a decimal point, hence the term 'floating point', actually stored in binary.

1.2 Operators

So, now we know how to store certain types of data using variables, and display what they contain. What else can we do with them? For a start, for the numerical variables, we can do maths. Let's look at an example.

```
>>> a = 2.0
>>> b = 3.0
>>> print(a + b)
5.0
>>> print(a - b)
-1.0
>>> print(a / b)
0.666666666667
>>> print(a * b)
6.0
>>> print(a**b)
8.0
```

Note

In Python 2.X, the data types for the division operation are especially important. An integer divided by an integer will assume you wish to perform 'integer division', i.e. the number of whole times one number goes in to another (with the result of floating point division rounded down). This means that `2 / 3` can give `0`, and should generally be ensured to be performed as `2.0/3.0` or with variables of both floating point type. Python 3.X will always return a floating point value from such a division.

Arithmetic Operators

The following is a list of the key arithmetic operators in Python:

Name	Symbol	Example	Description
Addition	`+`	`a + b`	Adds the two operands.
Subtraction	`-`	`a - b`	Subtracts `b` from `a`.
Multiplication	`*`	`a * b`	Multiplies the two operands.
Division	`/`	`a / b`	Divides `a` by `b`.
Exponentation	`**`	`a**b`	Raises `a` to the power of `b`.
Integer/floor division	`//`	`a // b`	Divides `a` by `b` 'rounded down'.
Modulus	`%`	`a % b`	Returns the remainder of `a` divided by `b`.

Brackets (these may be referred to as parentheses by messages in Python), can be used to make sure that order or operation is as intended for larger calculations. Take the following example,

```
>>> x = 3.0
>>> y = 4.0
>>> mag = (x**2 + y**2)**0.5
>>> print(mag)
5.0
```

Interestingly, not just numerical values have operators; other data types and combinations of data types have their own operators. So, for instance, the `+` operator when both operands are strings is 'concatenation', it joins the strings together. The `*` operator when one operand is a string and one is a number, duplicates the string that number of times. See the following examples:

```
>>> a = "Word"
>>> b = "Another word"
>>> x = a + b
```

```
>>> y = b * 3
>>> print(x)
WordAnother word
>>> print(y)
WordWordWord
```

1.3 Comparisons

Comparison operators behave in a similar fashion to the operators we have seen so far, usually taking two operands, and returning a result. In the case of comparison operators, however, they will do what it says on the tin: compare the operands somehow. The value they return will always be a `Bool`, `True` or `False` depending on the comparison.

The ones we will look at now are those with direct mathematical analogues, so they should be fairly familiar.

Comparison Operators

Operator	Example	Description
`==`	`a == b`	Is `a` the same as `b`
`!=`	`a != b`	Is `a` different to `b` (known as 'not equals')
`<`	`a < b`	Is the value of `a` less than `b`
`>`	`a > b`	Is the value of `a` greater than `b`
`<=`	`a <= b`	Is the value of `a` less than or the same as `b`
`>=`	`a >= b`	Is the value of `a` greater than or the same as `b`

These should hopefully seem fairly straightforwards, as mentioned they are all closely related to mathematical expressions you will already have seen. Each of these result in the return of a boolean, so for example: `3 < 4` gives back `True`. If you have variables `a` and `b`, which are values `10` and `20` respectively. For an expression `c = a < b` it would substitute in the values giving `c = 10 < 20`, then the comparison operator would be evaluated, giving `c = True` and assigning that value to `c`.

Note

Remember that '==' is not the same as '='! It is a very easy mistake to make and usually trips people up!

1.4 Exercises: variables and operators

Exercise 1

Create variables with names of your choice, holding the values `1`, `2.55` and `"1"` (remembering to put the last of these in quote marks, as shown). Print out all variables and their types.
Create a new variable, which is the sum of the first two variables. Print out this new variable and its type.
Try adding the second and third variable in the same way. Does this work? Why not?[a]

[a] More information on error messages, or 'tracebacks', will be given in the section on Tracebacks.

Exercise 2

Using a variable `r` to represent the radius of a circle, have Python print out the circle's circumference and area. For this exercise, you may assume that $\pi = 3$ (or feel free to use a closer approximation). You may choose the value of `r` yourself.

Exercise 3

Try writing a series of expressions in a code which will calculate the solutions to a quadratic equation of the form,

$$0 = a\,x^2 + b\,x + c \tag{1}$$

by evaluating the quadratic formula,

$$x = \frac{-b \pm \sqrt{b^2 - 4\,a\,c}}{2\,a} \tag{2}$$

2 Lists

One important variable type in Python that we have not seen yet is the 'list'. We have seen so far how we are able to store a piece of data of a variety of types in a variable. What if we have a collection of values we wish to store? Data points from an experiment, for instance. We wouldn't wish to create a separate variable for every single one; that would be awkward and inefficient.

So, what is a list? If a variable can be considered as a labelled box for containing a value; a list can be considered as a row of boxes. The row has one label referring to it, but each box within that row can be accessed independently by reference to how far along the row it is. The following is an example of the creation and use of a list.

```
>>> mylist = [1, 2, "word", 2.4]
>>> print(mylist)
[1, 2, 'word', 2.3999999999999999]
>>> print(mylist[1])
2
>>> print(mylist[-2])
word
```

The next concepts become trickier. Individual values from the list can be referred to using their 'index', which is what position along the list the value sits at. A value from within the list can be indexed by square brackets after the name of the list, with a index number in. Importantly, this numbering starts from zero, so `mylist[0]` is the first number in the list[2].

Just to further confuse matters, we are allowed to use *negative* indices. These refer to positions within the list counting backwards from the end of the list. So `mylist[-2]` refers to the value which is second to last in the list, however long the list may be.

Python contains some built-in functions that are useful for dealing with lists, such as `min()`, `max()` and `sum()`.[3] These work as you would expect:

```
>>> numbers = [3, 7, 5, 5, 2, 0]
>>> print(sum(numbers))
22
>>> print(min(numbers), max(numbers))
0, 7
```

Another commonly used function is `len()`, which returns the length of a list.[4] A full overview of python 3.7's built-in functions can be found at
https://docs.python.org/3/library/functions.html.

[2] This originates from the index referring to offsets from the beginning of arrays; the first value has no offset from the start, so its index is zero.

[3] We will be treating functions in more detail later on in these notes. For now, you can consider them to be 'black boxes' performing a specific task, such as computing the sum of a list of values.

[4] Some of the functions describe here do not work exclusively on lists. Try using `len()`, `min()` and `max()` on a string variable, and try to make sense of the output.

```
>>> print(len(numbers))
6
```

2.1 Exercises: lists

Exercise 4

Create a list (with a name of your choice) containing the strings "Python", "is", "a", "great", "programming" and "language". Print the first and second item from either end of the list (i.e., the first, second, second-to-last and last item).

Exercise 5

Using Python's list functions, calculate the mean and median values of a list of integers.

3 Tracebacks

PROGRAMMING is a task which does not always go precisely as planned first time. Very frequently you will encounter errors of one form or another, and it is important to be able to take the most possible information from them when it comes to going to to fixing the problem. The most usual form of 'error message' you will see in PYTHON is a 'traceback'. This is the name for the sequence of messages in which PYTHON tries to describe the most recent series of events/instructions which resulted in the current problem from which it cannot continue. Examples of these can be produced very quickly, and it is useful to see what some of the more common messages report, and understand the structure of tracebacks to better help us take advantage of the diagnostic information they provide.

The following code can be run in the PYTHON console, and will fail, providing the traceback seen immediately after the instruction.

```
>>> 'a' / 2
Traceback (most recent call last):

  File "<ipython-input-1-0993d71af7b7>", line 1, in <module>
    'a' / 2

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

The instruction was `'a' / 2`. The traceback for the command begins with the statement '`Traceback (most recent call last):`', this indicates that something has gone wrong and it will try to follow the commands back to just after things stopped working as planned. The next line describes the location of where a particular part of the trace occurred: `File "<ipython-input-1-0993d71af7b7>", line 1, in <module>`. In this case the 'file' is not an actual file, as the command was run in a PYTHON console, it mentions that it occurred on line 1 in '`<module>`' and then lists the line of code itself `'a' / 2`. The next part is the actual instruction on which the execution failed and a specific reason why: '`TypeError: unsupported operand type(s) for /: 'str' and 'int'`'. This particular message is pretty easily decipherable. It can be read as: 'For the symbol '/' on this line, the operands were types string and int; that combination isn't supported', or something to that effect. This tells us that, whilst for the symbol * there is an operator defined for operands string and integer ('repeat'), the symbol / has no such allowed combination of operand types.

When working with saved script files, you may get something which appears more complicated. The following code:

```
a = '2'
```

```
if (a < 3):
    print('This')
else:
    print('Not this')
```

was saved as `testread.py` in a directory on the author's computer. It was then run in Spyder, and causes the following traceback:

```
Traceback (most recent call last):

  File "<ipython-input-16-bbd0b2a49127>", line 1, in <module>
    runfile('C:/Users/tmk5/Documents/Workshops/GradNet/testread.py',
↪   wdir='C:/Users/tmk5/Documents/Workshops/GradNet')

  File "C:\ProgramData\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py",
↪   line 866, in runfile
    execfile(filename, namespace)

  File "C:\ProgramData\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py",
↪   line 102, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

  File "C:/Users/tmk5/Documents/Workshops/GradNet/testread.py", line 3, in <module>
    if (a < 3):

TypeError: '<' not supported between instances of 'str' and 'int'
```

This appears more confusing, but takes the same form as before. A traceback attempts to follow the entire path up to what went wrong, and in this case, technically, there are several steps. Spyder happens to run saved codes via a command `runfile()` which is the first juncture described in the traceback. This runs another routine (`execfile()`) which happens to be located in a file named `sitecustomize.py` within Spyder's path. This routine called a further routine named `exec()` also located in the same file (line numbers are given for each). Finally, the actual line of code for the real issue occurs. A similar error to before, that the 'less than' operator < does not support operations between strings and integers. This is because the variable `a` stored the *symbol* '3' as a string, not an actual numerical type. The final error also indicates the file and its path, as well as line number.

Usually, it is best to start at the end of a trace, and work back only as far as needed to solve the problem. In this case those earlier steps occurred to get to the point that the script failed, but were unrelated to the cause of the failure.

3.1 Exercises: tracebacks

Exercise 6

During any of the other exercises (and any coding you do in the future), it is almost guaranteed you will make some mistakes. This exercise is a reminder that bugged PYTHON code tends to tell you what is wrong. Try debugging your mistakes with the use of PYTHON's traceback messages.

Exercise 7

Among the files you have been given, there is a file titled `traceback_to_debug.py`. This file contains a script with some errors. Run the script, and use the traceback errors to find and remove the errors.
Bonus exercise: can you find out what the script is intended to do? Does it work as expected?

4 Loops

One important programming structure is known as a loop. Loops permit tasks to be performed multiple times, using different values, or various other forms of repeating any block of code. Languages tend to differ on how loops are handled; many have loops which are generally explicitly over ranges of values (C or Fortran for instance). Python is slightly different. Its main looping structure operates via 'iterators'. We can think of them in this context as operating over lists.

```
thing = [1, 3, 6.0, 8, "word"]
for i in thing:
    print(i)
```

the output of this code will be,

```
1
3
6.0
8
word
```

The structure of what is inside a loop or after/outside of it is defined by indented blocks. The `print` statement in the above example must be indented (the Python specification calls for four spaces). The first command which is not so indented is then after/outside the loop. As much code as desired can be included in an indented block, including other structures like loops, which would result in a further level of indentation for their contents.

Loop Structure

The structure of `for` statements is the following:

```
for value in  list:
     code to repeat
```

For each item in 'list', 'value' will be assigned to be that item for the purpose of any operations contained in 'code to repeat'.

4.1 Exercises: loops

Exercise 8

Using a loop, print out the squares of all integers between 1 and 10.

Exercise 9

Complete a code which runs your quadratic equation code from exercise 3, so that it will automatically solve the equations for `a = 2`, `b = 3` and the following values of `c` automatically: `-15, -10, 0, 5, 10, 15`.

5 Conditionals

A central concept of general purpose computing requires that codes can 'branch', take different routes, and perform different tasks, based on results as it goes. The structure for this which we are going to examine is an '`if`' statement. To make a decision, it requires a boolean value, a `True` means that it will perform the code nested inside of it, a `False` means it will skip it. This is shown in the following example. The code will take the first path which evaluates to `True`, be it the initial `if`, or subsequent 'else ifs' (`elif`).

```
a = 2
b = 4
if (a == b):
    print('a is the same as b')
elif (a < b):
    print('a is smaller than b')
else:
    print('a is bigger than b')
```

for which the output would be,

```
a is smaller than b
```

Conditionals Structure

The structure of `if` statements is the following:

```
if statement a:
     code a
elif  statement b:
     code b
elif   statement c:
     code c

...

else:
     code d
```

The `if` itself is always followed by a statement which must evaluate to a boolean (`True`/`False`), then a colon. The next line(s) must be indented and represent any code to be executed if statement a (in the above example) were true. `if` can be followed by any number of `elif`s, which each need their own boolean expression and a colon, then any code indented by one level. These can be followed by an `else`, which specifies any code which should be run if none of the preceding statement for `if` and `elif`s are true. `elif`s and `else`s cannot be used without an `if`. `if` must always be first, and `else`, if used, must always be last.

5.1 Exercises: conditionals

Exercise 10

Write a script that, from a list of integers between 1 and 100, computes the sum of all values between 20 and 40.
Write another script that computes the sums of all values in 'bins' of width 25 (i.e., 1–25, 26–50, 51–75 and 76–100) within a single loop.

Exercise 11: approximating π

Write a script that approximates π by the following algorithm:

1. Generate N pairs of random numbers $(a, b) \in [0, 1] \times [0, 1]$.
2. For each pair, calculate the distance of this point to the origin $(0, 0)$.
3. Approximate π as $\pi \approx 4\frac{N(r<1)}{N}$, where $N(r < 1)$ is the number of points withhin a distance 1 from the origin.

To generate random numbers, write

```
import random
```

at the start of your script, and use

```
x = random.random()
```

to assign a random number in the interval $[0, 1)$ to `x`. You can read the documentation at https://docs.python.org/3/library/random.html for more information about Python's built in random number functionality.
FYI: this algorithm is called a *Monte Carlo* algorithm.

6 Files

Python deals with interactions with files via a concept called a 'file object'; a file object behaves a bit like variables which we have defined in the past (and to many extents is treated exactly the same, though for reasons which may not yet be entirely clear). A file object has several 'methods' attached to it; these are the available ways in which the particular file referred to by the object may be interacted with. Let's look at an example to start us off. First, assume the existence of a plain text file named `example.csv` in the same directory as our script, it has the contents:

```
1, 4
1, 7
-5, 6
2, 2
3, 7
7, 2
```

This content could be read in to Python using a sequence of commands such as,

```
with open('example.csv','r') as f:
    fdata = f.read()

print(fdata)
print(repr(fdata))
```

The second of the print lines would result in:

```
'1,4\n1,7\n-5,6\n2,2\n3,7\n7,2'
```

Files are stored as single continuous streams of binary data; to represent where line breaks are meant to go, this has to be encoded like a character/symbol as any other. It's a special character, though; when it's encountered by a file reader, they are usually configured to realise that it's a special character, and then do something differently to how simple symbols are displayed on screen. One such character like this already mentioned is the 'tab' character. Another similar character is the one (or ones) used to indicate that a line break should occur. Now, this gets a little complicated, as different systems have slightly different conventions for what indicates a line break. Linux systems, and some Windows systems, use just character number 10 in the ascii symbol table 'Line Feed', represented in Python notation (and many others) as `'\n'`. Mac systems use character number 13 'Carriage Return', represented as `'\r'`. Windows systems generally use both in a sequence starting with Carriage Return `'\r\n'`.

Assuming that the line break symbol is `'\n'`, files can be broken up into lists at those points, making a list element per line.

```
with open('example.csv','r') as f:
    fdata = f.read().split('\n')

print(fdata)
```

For which the result would be,

```
['1, 4', '1, 7', '-5, 6', '2, 2', '3, 7', '7, 2']
```

Combining this new list with a loop can let us do the following:

```
with open('example.csv','r') as fin:
    fdata = fin.read().split('\n')

for line in fdata:
    print(line.split(',')[0])
```

For which the result would be the following, having extracted the first column of data,

```
1
1
-5
2
3
7
```

6.1 Exercises: Files

Exercise 12

There is a file named `read_in_example.csv` which is a comma-separated form file. Try reading this file in and finding the mean, minimum and maximum of each column. Determine which column, has the higher mean and have the code report this to the screen.

7 Functions

Functions can be considered as ways to (repeatedly) refer to batches of code, without writing all of the code necessary every time. They usually perform some kind of commonly (or sometimes uncommonly) used task, to make the overall construction of codes easier. We have already encountered several of Python's built in functions (such as `print()` and `type()`), but, as we shall see, a powerful feature of Python (and almost any programming language) is the ability to define custom functions with your own code.

Functions usually have 'arguments', things provided to the function for it to perform its task.

The `input()` function allows you to request that the person running the program (the 'user') types something in. The argument is the prompt text which will be displayed to the user.

```
>>> a = input('Please type something in: ')
Please type something in: I typed these words when prompted
>>> print(a)
I typed these words when prompted
```

When we require numbers, the input from `input()` must be converted into a numerical data type with `int()` for integers or `float()` for floating point (decimal) values. Such as follows:

```
>>> a = int(input('Please provide a number: '))
Please provide a number: 6
>>> print(a)
6
>>> print(type(a))
<class 'int'>
```

The `range` function generates a sequence of numbers; the argument is the number of numbers to generate (automatically starting at 0). This avoids needing to manually generate trivial numerical sequences.

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
b = range(10)
print(a)
print(b)
```

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

You can also define your own functions using the '`def`' instruction.

```
def my_own_function(a,b):
    x = 2.0 * a + 3.0 * b
    return x

def my_own_function_2(string):
    return '!!' + string + '!!'

print(my_own_function(1,2))
print(my_own_function(6,3))
print(my_own_function_2('Some text'))
```

for which the output would be,

```
8.0
21.0
!!Some text!!
```

7.1 Exercises: functions

Exercise 13

Write a function `print_if_string()` with one argument. The function should print the value of the argument if the argument is a `string`. If the argument is not a `string`, you can either ignore it or print out a message of your own choice.
Test your function on a number of different variables with different types to check if it works correctly.

Exercise 14

Adapt your quadratic equation script from questions 3 and 9 to make the solution algorithm itself into a function with variables a, b and c.
Optionally, have this function use the `input()` function such that the user specifies a, b and c each time they run the script.

8 Advanced

This last section contains some pointers to two of the most useful Python modules, NumPy and MatPlotLib. There are no further exercises, but you should have been given the materials for the 'Introduction to Scientific Python' workshop, which contains plenty of further practice.

8.1 NumPy

The module named NumPy provides access to a wide variety of mathematical routines and input/output processes to make it easier to write complex codes faster. It is recommended to have a look through the general capabilities which it offers, these and other details can be found on its website, http://www.numpy.org/. In addition, it has thorough definitions of all of the objects and methods which it provides, these can help in working out how to accomplish certain tasks, or better understand the wide variety of options some of the functionality provides.

Basic loading of a module, and some of the interaction with the NumPy module in particular are found in the following example:

```
>>> import numpy #load module
>>> print(numpy.__version__) #get version
'1.11.1'
>>> print(numpy.random.rand()) #make random number
0.062399692509412241
>>> print(numpy.sqrt(16.0)) #takes square root of 16
4.0
```

We can also store structures called 'arrays' using NumPy; these are very similar to lists, but are better to use for maths. This is due to the 'vectorisation' of maths applied to lists, which let us apply a process across all of the elements in the array in one command. See the following example.

```
>>> import numpy
>>> a = [1, 3, 5, 7, 20] #list
>>> b = numpy.array(a) #array
>>> print(a, type(a), a[1], type(a[1]))
[1, 3, 5, 7, 20] <type 'list'> 3 <type 'int'>
>>> print(b, type(b), b[1], type(b[1]))
[ 1  3  5  7 20] <type 'numpy.ndarray'> 3 <type 'numpy.int32'>
>>> #print(a**2) #would not work!
>>> print(b**2) #vectorised operation
[  1   9  25  49 400]
```

Another advantage of the NumPy module is its support for data processing. The file `example.csv` contains several lines, each with a pair of comma separated values. We will now look at an alternative method for reading in those data, making use of the NumPy module.

```
import numpy
data = numpy.genfromtxt('example.csv',delimiter=',')
print(numpy.shape(data))
print(data)
```

The result should be,

```
(6, 2)
[[ 1.  4.]
 [ 1.  7.]
 [-5.  6.]
 [ 2.  2.]
 [ 3.  7.]
 [ 7.  2.]]
```

In this case, the result is a 2D array (think of it as rows and columns) with all of the values from the file. The `numpy.shape()` function shows the dimensions of the 2D array, in other words, the first axis is 6 long, the second is 2 long.

We may not want a 2D array, we may prefer two 1D arrays to work with, each representing a column. NumPy permits us to do this, too.

```
import numpy
xs, ys = numpy.genfromtxt('example.csv',delimiter=',',unpack=True,dtype=int)
print(xs)
print(ys)
```

The result should be,

```
[ 1  1 -5  2  3  7]
[4 7 6 2 7 2]
```

The `unpack=True` argument tells it to break the 2D array into columns, as long as sufficient variable names are listed before the function (the `xs,ys`, for two columns). In addition, we have set an argument `dtype=int`, which forces the data type of the values to be integers rather than the default of floating point values.

NumPy's abilities do not end there! It also contains various statistical functions which are of great use. The following example illustrates some of the most commonly used.

```
import numpy
xs, ys = numpy.loadtxt('example.csv',delimiter=',',dtype=int,unpack=True)
print(numpy.mean(ys), numpy.min(ys), numpy.max(ys), numpy.std(ys))
```

The result should be,

```
4.66666666667 2 7 2.13437474581
```

These produce the mean, minimum, maximum and the standard deviation of the 'ys' column, respectively.

8.2 MatPlotLib

Other than statistics, another method of reducing raw data is through plotting it. Once more assuming the same `example.csv` file as previously, we can begin making use of a different module to NumPy, named MatPlotLib. MatPlotLib is a plotting library designed to have similar commands to MatLab. We will plot the data from that file as a simple line graph.

```
import numpy as np
import matplotlib.pyplot as plt
xs, ys = np.loadtxt('example.csv',delimiter=',',dtype=int,unpack=True) #load in the data
ys2 = ys**2 #make `ys2' the square of `ys'
plt.plot(xs,ys,'ro') #plot ys against xs, with red circle markers (ro)
plt.plot(xs,ys2,'go') #plot ys2 against xs, with green circle markers (go)
plt.xlabel('my x axis')
plt.ylabel('my y axis')
plt.savefig('myplot.pdf') #save the finished figure as a .pdf
plt.show() #also display it on screen
plt.clf() #clear the figure afterwards for any new plots which follow
```

This will produce a plot with two sets of data points. One is the raw data from the file, the other is each of the y values squared.

The result is the plot shown in figure 1.

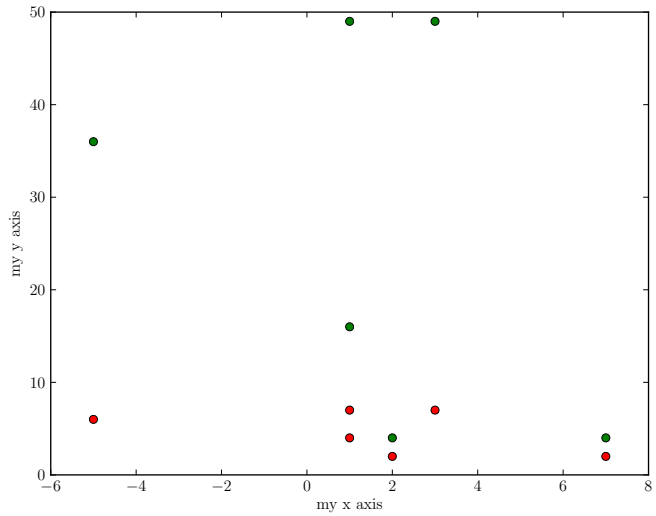

Figure 1: Figure produced from the execution of the first example code in Section 8.2

One other aspect of plotting which is extremely useful is adding a legend. When using the plotting commands, the optional argument '`label='some text'`' can be used (with 'some text' being the name of the data series). With this added, we can have the command '`plt.legend()`' on a subsequent line, which will display a legend for any data series with labels. Try this and see what it looks like! The example above could be modified to have the plotting lines read:

```
plt.plot(xs,ys,'ro',label='First series')
plt.plot(xs,ys2,'go',label='Second series')
```

and then

```
plt.legend()
```

to get the legend itself to display.

9 Finale

You should now have access to a variety of tools which permit useful tasks to be performed with Python. Python has immense capacity for processing data, analysing data, generating simulations, coding interactive GUIs and much more. Far more, in fact, than can be included in such a workshop as this.

A wide variety of further resources are available, both physically and on the Internet. I would encourage you to consult as many of these as possible and gain knowledge through exploring applications of coding and how to approach data analysis computationally.