In [4]:

```
from __future__ import print_function
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

PH2150- Scientific Computing and Employabilty

Introduction to Python (1 week) Lecture 2

Dr. Andrew Casey (a.casey@rhul.ac.uk, W054) ¶

Review of post Exam Topics

- Python Environment
- Variable Types
- Modules, Packages, Libraries
- Control Structures

The *Python* Environment

The teaching laboratory computers are all 64-bit Windows 10 machines, although *Python* is cross platform we prefer you to use these machines as (at least my expertise lies with Windows rather than Apple IoS). However feel free to use your own laptop/operating system but the support may not be so great for Mac users.

Python 2.x versus Python 3.x

There are two variants of *Python*, if you learn one you will be able to easily use the other with some small changes in syntax. This year we have chosen to use *Python* 3.6.

The environment installed in the teaching lab is called *Anaconda* and can be downloaded here:

- https://www.anaconda.com/download/ (https://www.anaconda.com/download/)

Launch Anaconda Navigator from the start menu

Within the navigator we will use *Jupyter* (for notebooks like this one) and *Spyder* (Scientific PYthon Development EnviRonment) for editing code and running code from the interpreter.

On these machines you can run *Python* 2.x or 3.x, in the Navigator (within the environments tab select *Python36* for 3.x or *Base* for 2.x) we have added a **PH2150** environment for this course which is a *Python* 3.x environment compatible with a 3D package called *mayavi*.

The Environments tab indexes (and allows you to update) the packages that are available within the selected Environment.

In addition commands can be run directly from a command line prompt using the *Anaconda Prompt* from the start menu, or by opening a terminal from the environment tab (clicking on the play symbol after the environment name).

Spyder (Scientific PYthon Developement EnviRonment)

We will open this to take a quick look at some of the features:

- If you have selected the correct environment then it should say *Python* 3.x in the interpreter
- The working directory (where you store and run the code from) can be modified using the GUI.
- There are many options that can be set about how the environment performs, for example initially the graphical output of code i.e. a matplotlib plot may be set to appear *inline* in the interpreter. It is more flexible for this to open in it's own window (allowing zooming, saving as pdf etc) to this you need to update the prefrences for how the graphics are displayed.

Select Tools ->

Preferences - opens a dialog box Within Prefrences select IPython Console (this controls the settings of the interpreter)

Select the graphics tab - Change graphics backend from *inline* to *Automatic* or *QT5*

Variables in computing.

A variable in computer programming is a storage location associated with a symbolic name that contains a known or unknown value. In Python the variable is assigned a value by using the = symbol.

= is the assignment operator

Variables come in many types, and the way that your code interacts with the variable dependends on the data type. The following are common variables types that we will encounter:

- Numbers
- Strings
- List
- Tuple
- Dictionary
- Boolean
- Object
- None (A special variable type, that means non existent, not known or empty)
- Array

Variable Names

Variable names in *Python* can contain alphanumerical characters a-z, A-Z, 0-9 and some special characters such as _. Normal variable names must start with a letter. By convention, variable names start with a lower-case letter.

Python keywords that cannot be used as variable names:

```
and, as, assert, break, class, continue, def, del, elif, else, except,
exec, finally, for, from, global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while, with, yield
```

Note: Be aware of the keyword `lambda`, which could easily be a natural choice of variable name in a scientific program. But being a keyword, it cannot be used as a variable name.

Numbers in *Python*

Python has four basic numerical types.

- Floating point numbers (floats)
- Integers (less than 32-bit), and Long Integer (greater than 32-bit)
- Complex numbers

In [1]:

```
x=4 # assigns the value '4' to the variable 'x'
print('value=',x,type(x)) # print the string "value=", x and identify varaible type
type(x)
```

```
value= 4 <class 'int'>
```

Out[1]:

```
int
```

In [3]:

```
xx=4+2j
print('value=',xx,type (xx))
print(xx.real, xx.imag)
```

```
value= (4+2j) <class 'complex'>
4.0 2.0
```

In [4]:

```
yy=4.0
print('value=',yy)
type(yy)
print(dir(yy))
```

```
value= 4.0
['__abs__', '__add__', '__bool__', '__class__', '__delattr__', '__dir__',
'__divmod__', '__doc__', '__eq__', '__float__', '__floordiv__', '__format_
_', '__ge__', '__getattribute__', '__getformat__', '__getnewargs__', '__gt
__', '__hash__', '__init__', '__init_subclass__', '__int__', '__le__', '__
lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__pos__', '_
_pow__', '__radd__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr
__', '__rfloordiv__', '__rmod__', '__rmul__', '__round__', '__rpow__', '__
rsub__', '__rtruediv__', '__set_format__', '__setattr__', '__sizeof__', '_
_str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', 'as_in
teger_ratio', 'conjugate', 'fromhex', 'hex', 'imag', 'is_integer', 'real']
```

Variable type *string*

A string is the variable type that stores characters, i.e. stores text.

The string is created by inserting the characters that make up the string between '*insert text here*'

In [5]:

```
mystring='Andrew' # using '' creates the string
type(mystring)
print(4*mystring) # What happens when you combine strings with operators
print(mystring[3]) # you can index individual characters from a string
print (mystring[0:3]) # you can slice elements out of a string
print(mystring.upper()) #  there are many functions that operate on strings e.g. upper
 changes all to uppercase
```

```
AndrewAndrewAndrewAndrew
r
And
ANDREW
```

Variable type *list*

A list is a flexible variable type that can store a list of variables (which can have mixed types) within in it. E.g. a list of numbers, a list of names, or a combination of numbers, strings and even other lists.

The list is created using the syntax mylist = [element1, element2,..., last element]

In [6]:

```
mylist=[1,2,3,4.0,'Andrew']
print(mylist)
print(type (mylist))
print(len(mylist)) # returns the number of elements in the list
print(mylist[0]) # index [0] returns the first element in the list
print(dir(mylist))
```

```
[1, 2, 3, 4.0, 'Andrew']
<class 'list'>
5
1
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__
dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reverse
d__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

Modules, packages

Most of the functionality in *Python* is provided by modules. The *Python* Standard Library is a large collection of modules that provides cross-platform implementations of common facilities such as access to the operating system, file I/O, string management, network communication, and much more.

A Module: A file containing *Python* definitions (functions, variables and objects) themed around a specific task. The module can be imported into your program.

To use a module in a *Python* program it first has to be imported. A module can be imported using the import statement. The convention is that all import statements are made at the beginning of your code. For example, to import the module *math*, which contains many standard mathematical functions, we can do:

In [7]:

```
import math # import the module called math
print(dir(math)) #returns all the names of functions avaialble within the math module
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'c
os', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'facto
rial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log1
0', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

In [8]:

```
#After importing these functions are now available
x=math.sqrt(7) # syntax for assigning a value of the square root (function from math) o
f 7 to the varaible x
print(x)
```

```
2.6457513110645907
```

In [9]:

```
print(help(math.sqrt))
```

```
Help on built-in function sqrt in module math:

sqrt(x, /)
    Return the square root of x.

None
```

Different ways to import modules and functions

import modulename e.g. import math

then you can use function by calling modulename.functionname()

from modulename import functionname e.g. from math import sqrt

Explicitly import a function by its name also you to call the functionname()

from modulename import*

imports all the functionnames from the module

import modulename as alias

In [10]:

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

x=np.array([1,2,3,4])
y=np.sin(x)
plt.plot(x,y)
plt.show()
```

```
<Figure size 640x480 with 1 Axes>
```

Modules, packages

A Package: Is a set of modules or a directory containing modules linked through a shared theme. Importing a package does not automatically import all the modules contained within it.

Some useful packages:

- *SciPy* (maths, science and enginering module),
- *Numpy* (Numerical Python for arrays, fourier transforms and more),
- *matplotlib* (graphical representation),
- *pandas* (datareader)
- *mayavi* (3D visualisation module in Enthought distribution)

If when running your script you get the error:

ImportError: No module named [module/package name]

Then you may need to install the package into your environment using the Anaconda Navigator Environment tab

Control Structures (*if, for, while, else, elif*)

In general, statements are executed sequentially, top to bottom. There are many instances when you want to break away from this e.g,

- Branch point: choose to go in one direction or another based on meeting some condition.
- Repeat: You may want to execute a block of code several times.

Programming languages provide various control structures that allow for more complicated execution paths. Here we will use conditionals and loops:

- if
- while
- for
- else
- elif

In addition the following functions are useful for control structures,

- range
- break
- continue

Control Structures: Boolean Variable type (True of False).

Variable type – *Boolean* (True or False)

- $==$ is the equivalent to operator. Returns true if both operands are the same.
- $!=$ is not equal to operator. True if both the operands are not equal else False.
- $>$ is greater than operator. True if the first operand is greater than the second operand.
- $>=$ is greater than or equal to operator. True if the first operand is greater than or equal to the second operand.

Similarly, $<$ and $<=$ are less than and less than or equal to operators.

Combined with the logical operators *and, or, not* allows more complex conditional statements.

In [14]:

```
x=2
print(x==3) # Is x equivalent to 3?
print(x>=1 and x<3) # combination of operators
```

```
False
True
```

The *if* statement

The *if* statement is used to check a condition: if the condition is TRUE, we run a block of statements (the if-block), else (FALSE) we process another block of statements (else-block) or continue if no optional else clause.

if guess == number: # 1st condition

```
    <Block of statements> # the if block if condition is TRUE (note indented)
```

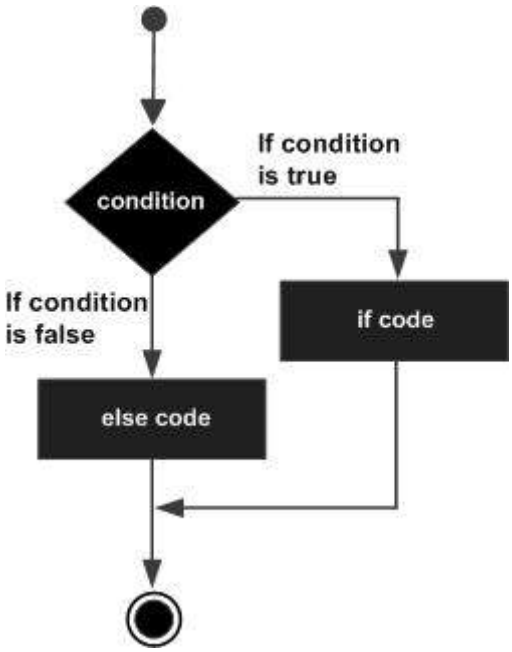
else:

```
    <Block of statements> # the else block if condition is FALSE (note indented)
```

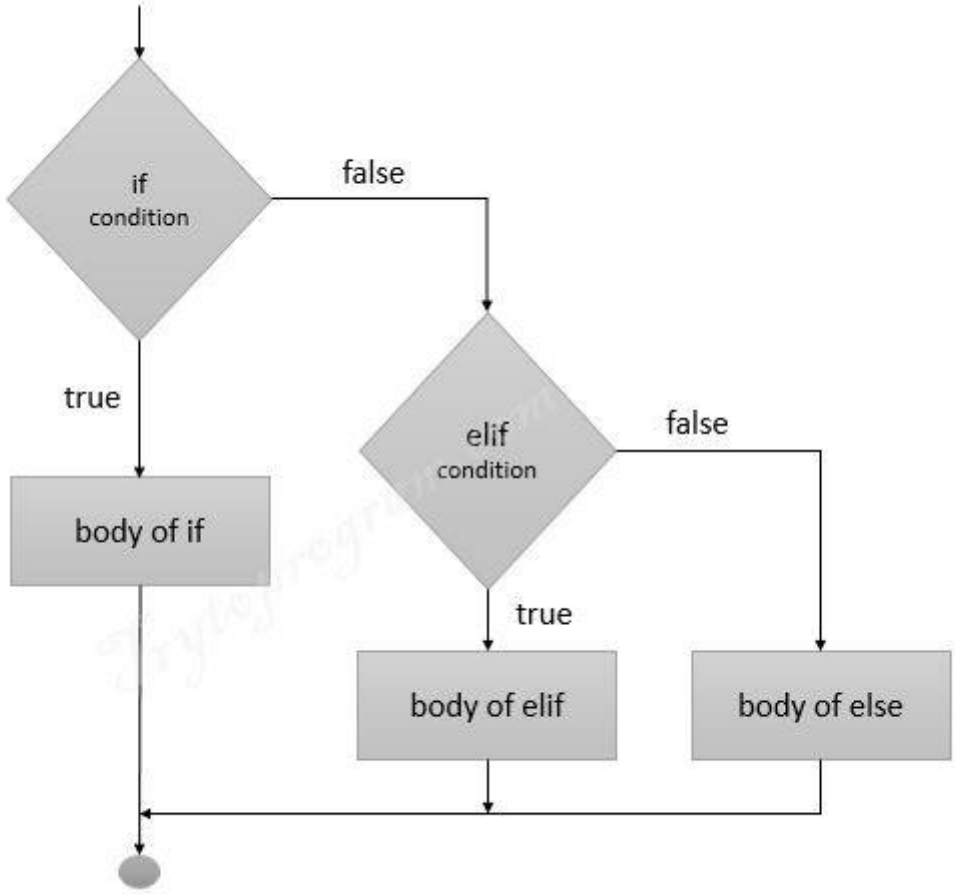
After running either block of statements you drop back into the main programme.

The *elif* statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE..

The *if* statement



The *elif* statement



if example

In [16]:

```
x=[1,2,3,4,5]
if len(x)<10: # if this condition is true run the idented block of statements
    print(x) # block of statements
    print(x[3])
    print('length of x =',len(x)) # when you reach the end of the block the if statemen
t is finsished
```

```
[1, 2, 3, 4, 5]
4
length of x = 5
```

The *for* loop

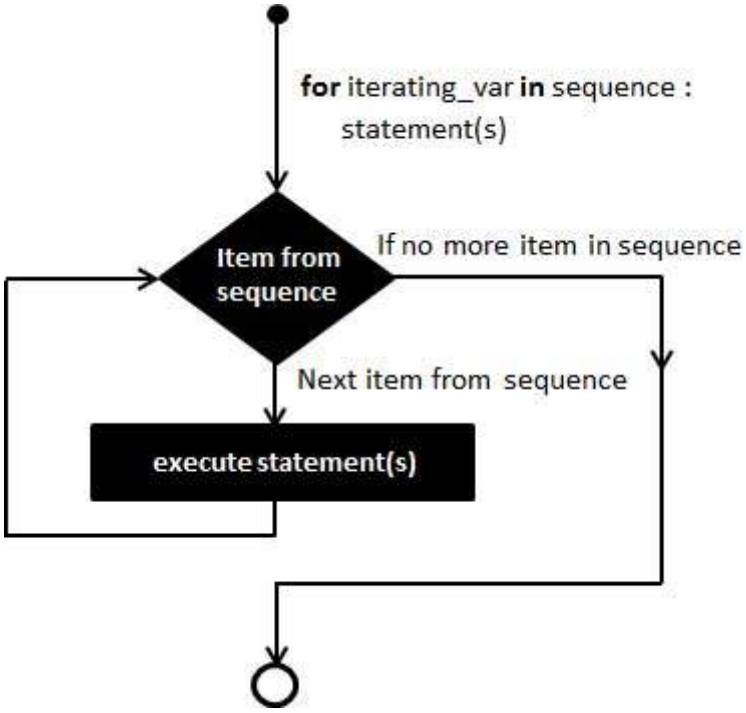
Loops: *for* loops are traditionally used when you have a piece of code which you want to repeat a number of times. As an alternative, there is the *while* Loop, the *while* loop is used when a condition is to be met, or if you want a piece of code to repeat until intrupted by another event.

In [17]:

```
for letter in 'Python':     # loops for each character in string
    print( 'Current Letter :', letter)

fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # loops for each element in list
    print ('Current fruit :', fruit)

for index in range(len(fruits)): # loops for each element in list
   print ('Current fruit :', fruits[index])

FruitsA=[x for x in fruits if x[0]=='a'] # returns [‘apple’], this is known as list com
prehension.

print(FruitsA)
```

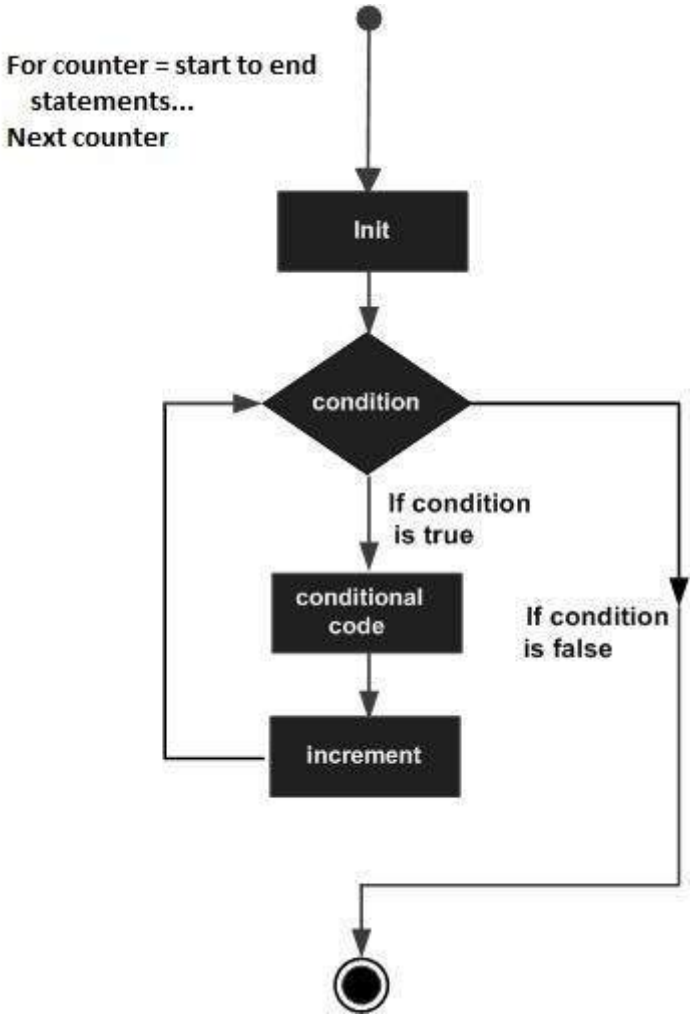
```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Current fruit : banana
Current fruit : apple
Current fruit : mango
['apple']
```

for flowchart



Iterating over all the values in a sequence

for flowchart



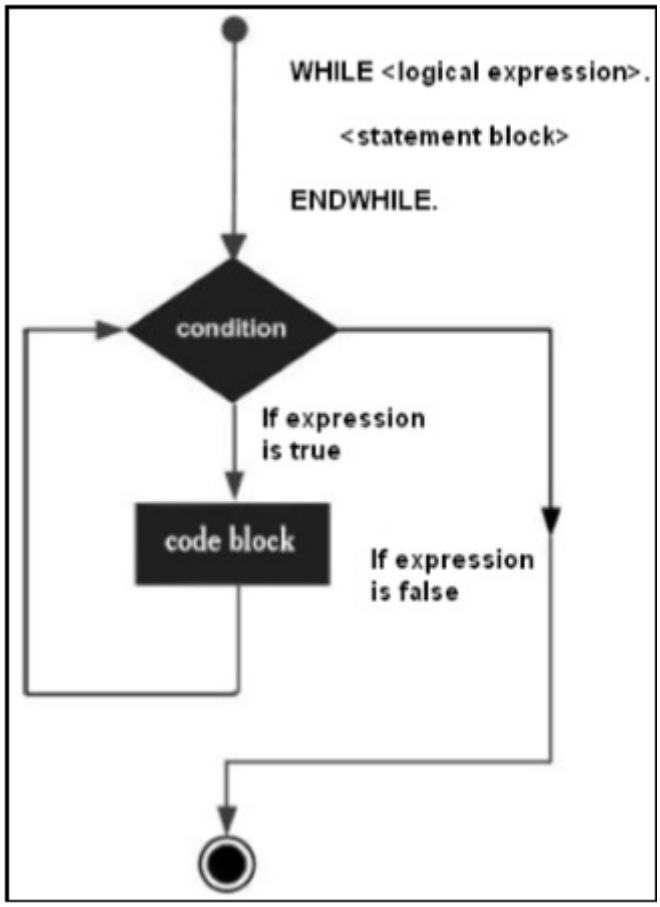
Incrementing a for loop from start to end.

In [18]:

```
#while expression: #while TRUE execute block of statements
#<Block of statements>

var = 1
while var == 1 :  # This constructs an infinite loop
    num = input("Enter a number  :")
    print( "You entered: ", num)

    if num=='Q': # This gives a route out of loop
        print ('Goodbye')
        break # break will terminate the loop
```

```
Enter a number  :4
You entered:  4
Enter a number  :5
You entered:  5
Enter a number  :Q
You entered:  Q
Goodbye
```

while flowchart



Continue until condition is FALSE.

In [ ]:

```

```