Introduction to Computer Programming in Python

Dr. Simon Hanna
Room: 3.44, email: s.hanna@bristol.ac.uk
Version 1.7
Last update: 16/09/2020

Contents

1 Introduction

1.1 Structure of course

1.1.1 Instruction

- A set of lectures, which will introduce the Python language, and discuss various computational methods.
- Several synchronous online sessions to demonstrate key points regarding using and running Python.
- This Python tutorial document.
- A series of 2 hour online drop-in sessions, at various times during the Autumn and Spring terms, so that students can get help with their own code.

  - Demonstrators will be available during these help sessions.
  - Please direct questions to the demonstrators during these help sessions.

1.1.2 Assessment

- There are 4 assessed exercises and 4 online tests - a precise breakdown of marks will be given in lectures and on Blackboard.
- Marks will be given for functionality but also programming style. Is the code efficient and well set out (use comments)? Is the code efficient, easy to understand and logically laid out?
- **Note that plagiarism will be checked for electronically.**

Detection of plagiarism will normally result in:

1. award of mark of zero
2. repeat exercises
3. a note on your record

Don't do it. . . .you will be caught!

1.1.3 Submiting your code

- Submit your Python scripts via Blackboard.
- Submit ONLY a single program file for each problem set.
- Only use Python version 3.x, as documented in these notes. (The current version is 3.8.3). Python 2.x is very old and should not be used for new programs. Python 3.x will be available using Anaconda on University of Bristol computers.
- The deadlines for all exercises will be given on Blackboard and in lectures.
- Standard Faculty of Science penalties for late work will be applied.

1.2 Learning Objectives

To become proficient at writing simple computer code using the Python programming language. Specifically the course will cover:

- Simple variables and objects
- Operators
- Branching
- Looping
- Input and output: console/keyboard and also via files
- String manipulation
- Functions
- Arrays as defined in the `NumPy` module
- Graph plotting

1.3 Why learn Python?

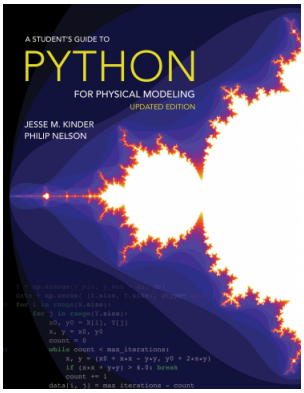
Computer programming has many uses in physics:

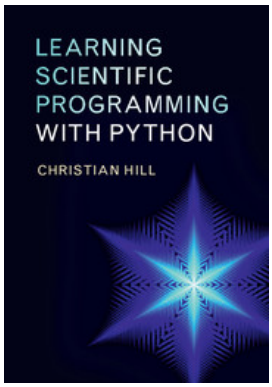
- Solving problems numerically (virtually all real world physics problems are solved numerically)
- Controlling equipment
- Processing data, generating graphs etc.

Python is an increasingly popular programming language, that combines flexibility with ease of learning and use. It is extremely well supported for scientific and mathematical applications, and contains modules for performing a wide range of numerical analyses and for generating high quality graphical output. There is a wealth of books and online tutorials available to help get you started.

1.4 Books and online resources

There are countless books available for learning Python. You should look for an up-to-date text that focuses on Python 3.x. Two recent books that teach the language in the context of physical problem solving are:





- Jesse Kinder & Philip Nelson: "A Student's Guide to Python for Physical Modelling" (Princeton University Press, Updated edition, 2018) A fairly short introduction that, however, covers all of the programming needed for the course, including the use of `NumPy` and `MatPlotLib`.
- Christian Hill: "Learning Scientific Programming with Python" (Cambridge University Press, 2016) A more comprehensive text than Kinder & Nelson which is more useful for looking things up, but takes longer to read.

Chapter references to the books by Kinder & Nelson, and Hill, will be given in the tutorial below.

Then there is the Python tutorial from the Python Software Foundation:

- https://docs.python.org/3.8/tutorial/

and various online lectures, such as those by Robert Johansson that are based on IPython (Jupyter) notebooks:

- https://github.com/jrjohansson/scientific-python-lectures

1.5 Hardware / Software

This course will focus (mainly) on standard Python 3.x and so can be run on any computer with a Python installation, i.e. a Windows PC, Apple Mac or Linux system.

An Integrated Development Environment (IDE) such as Spyder is strictly not needed. You could just use a text editor (e.g. notepad) and run Python from the command line.

BUT, an IDE does make life a lot easier.

We will be using the Spyder IDE, togther with Jupyter notebooks (or Jupyter Lab), which are all available as part of the standard Anaconda distribution, available for free download from Anaconda Inc. Anaconda, Jupyter and Spyder will be found on the PCs in Physics Rm 1.14 and across the University.

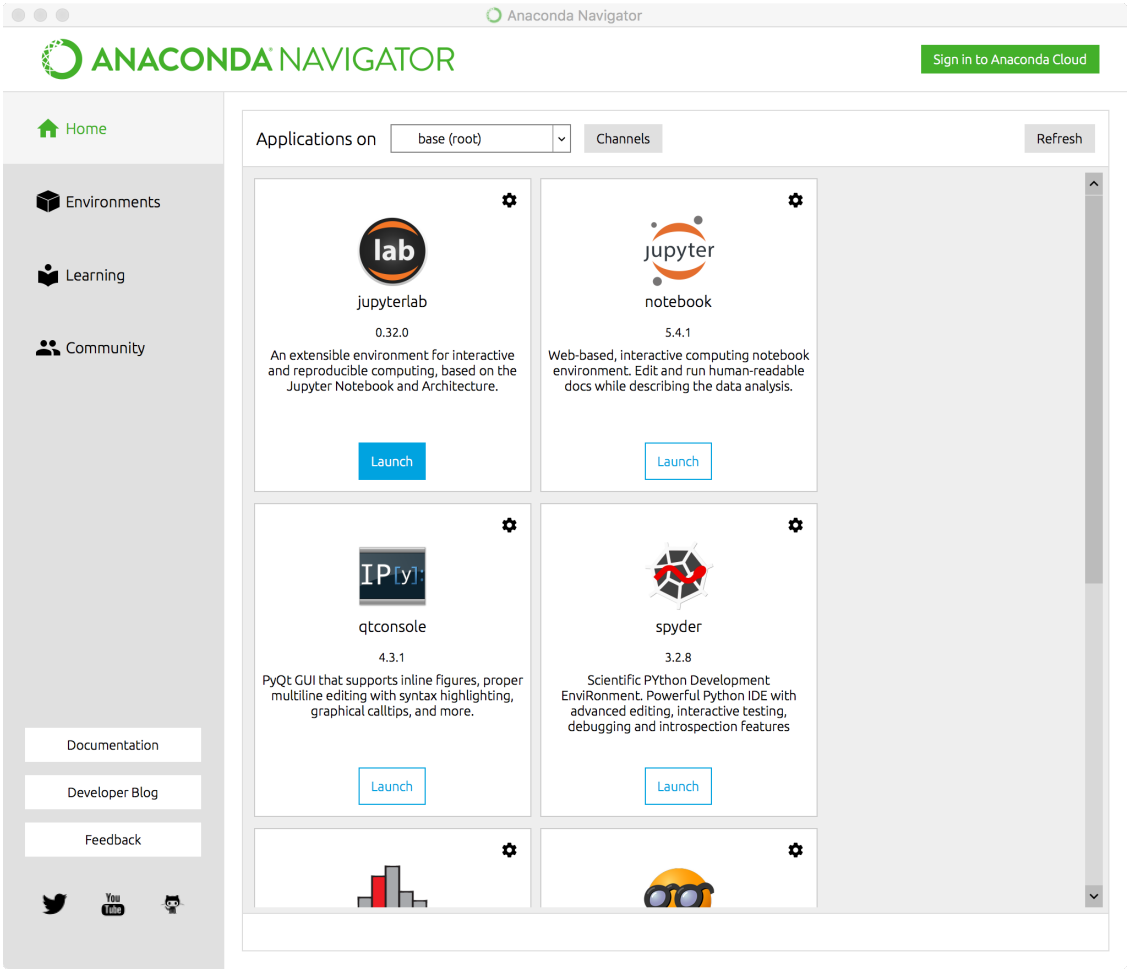
1.6 Installing Jupyter, Spyder + Python 3.8

Download Anaconda, complete with Python 3.8.3, directly from Anaconda Inc (https://www.anaconda.com/products/individual), for your particular computer hardware. The latest version is Anaconda 2020.07. HINT: You will need to scroll down to the bottom of this page to find the download links. You do NOT need to create an account. Unless you have good reason to choose otherwise, you should choose

the *Graphical Installer*. Follow the instructions given on the web-page, and accept the default installation. After the installation is complete, you will be left with a "Navigator" program which allows you to start up:

1. the Spyder IDE (scientific Python development environment);
2. the Jupyter notebook system (as used for these notes);
3. the Jupyter Qt console, another way of running interactive Python (IPython);
4. a few other utilities that you can ignore.

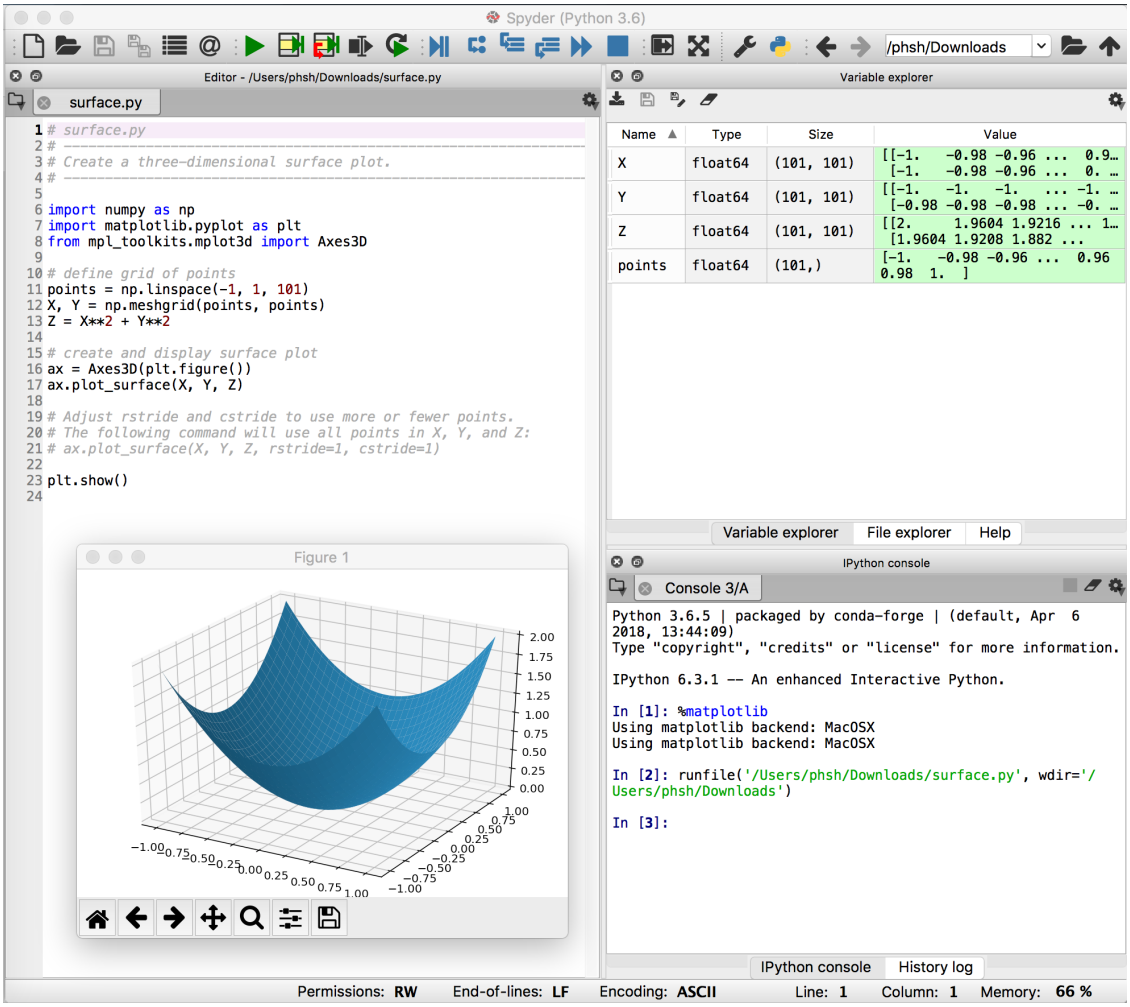
This is how the Navigator should look (the exact layout and version numbers may vary):



You now have several different ways of running Python:

- by typing Python statements in the Qt console (not recommended)
- by executing Python code within a Jupyter notebook (needed for this tutorial and the exercises during the Autumn term)
- within the Spyder IDE, typing and running scripts in the editing window (needed for the exercises in TB2)
- within the Spyder IDE, typing Python code in the IPython console window (not recommended but occasionally useful)

Here is an example of Spyder in action:



2 Starting out: Using Python

The recommended textbook by Kinder and Nelson (see above) is a very good tutorial guide - in particular reading the first four chapters and trying out the examples is a great way to get started. The tutorial guide that follows below summarises the key points that you will need for the online tests and exercises.

2.1 Viewing this tutorial in Jupyter Notebook

As indicated above, there are several different ways of running Python code. This tutorial contains many snippets of code. You are therefore strongly encouraged to view this tutorial using Jupyter Notebook or Jupyter Lab. If you do this, you will be able to run, and re-run, the code snippets within the web-page, modifying and testing as you go.

To launch Jupyter Notebook, first launch the Anaconda Navigator and then click on the appropriate button for Jupyter. Either the Jupyter Lab or the Jupyter Notebook will work. Whichever you choose will launch a web-browser, which will present you with a view of your files on your computer. You should click through to the folder containing this tutorial, and finally select the file *PythonTutorial_v1.7.ipynb*.

You should now see an interactive version of this tutorial in your web-browser. To run the code below, just click in the code cell and hit `"Shift-Enter"` or press the “Play” button in the toolbar above:

[1]:
```
#
# My first Python program
#
print("Hello, world!")
```

```
Hello, world!
```

In the above program, "`print()`" is a built-in Python function for printing to the screen, and the lines beginning with hashes (#) are comments, and are not executed.

2.2 Using the scientific Python development environment (Spyder)

However, a more permanent approach to coding is to write code into a file (a Python script). You can do this within Spyder, and see both the code and the result of running it in the same window. Spyder has another big advantage: the `Variable Explorer` tab allows you to keep track of all variables while your program is running, which is extremely useful for fixing errors.

You can create a new Python script by choosing "`File --> New file...`" in the Spyder (python) menu. The new script should open in a tab in the left-hand window. Be sure to save the script in an appropriate folder; all python scripts should have the `.py` filename extension.

Copy the lines of code from the cell above into your file. Re-save the file and run it either by:

- clicking the play button;
- choosing "`Run --> Run`" in the spyder menu;
- hitting "`F5`" on the keyboard.

Observe the output from your program in the console window (bottom right of screen).

2.3 Runtime errors

If you make a mistake in typing your Python program, Python won't know how to execute your code. When this happens, Python will issue one or more error messages, such as the following:

[2]:
```
#
# My first (broken) Python program
#
prnt("Hello, world!")
```

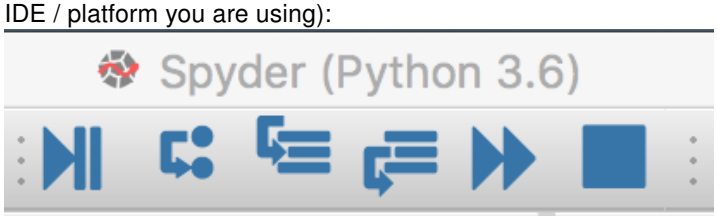
```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-2-d448f4904744> in <module>
  2 # My first (broken) Python program
  3 #
----> 4 prnt("Hello, world!")


NameError: name 'prnt' is not defined
```

In this case, the user has typed "`prnt`" instead of "`print`". Python does not recognise "`prnt`" and says so. If you correct the error in the code above, and re-run the cell, the error message will go away.

Sometimes, a simple error can cause a flurry of error messages to appear. Don't worry if this happens - just look closely at the code highlighted and see if you can identify the mistake.

2.4 Debugging

- If you are really struggling to identify the mistake in your program i.e. if it appears to run but produces the wrong answer, or does something unexpected, consider using the debugger built into Spyder (or whichever IDE / platform you are using):



- With this you can:
  - step through your code line by line checking the contents of the variables, step by step;
  - step into or out of functions;
  - set breakpoints to cause the program to halt execution at particular places;
  - examine the values of variables and watch them change as the program executes.

2.5 Initialising variables

- Note that variables initially contain random values.
- Usually this is not a problem, because you cannot use a variable until it has been defined. e.g. try the following code:

[3]:
```
x=BrandNewVariable
print(x)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-3-25b9b4f44839> in <module>
----> 1 x=BrandNewVariable
      2 print(x)


NameError: name 'BrandNewVariable' is not defined
```

- The error message above indicates that `BrandNewVariable` has not been seen before, and so cannot be used.

- To avoid the problem, always assign values to variables before you use them.
- As noted above, `list` objects can be both created and initialised using the following shorthand:

[4]:
```
MyNewList = [0.0] * 5
print(MyNewList)
```

```
[0.0, 0.0, 0.0, 0.0, 0.0]
```

- In one notable case, i.e. the NumPy array, it is possible to allocate an empty array of values without assigning actual values. This can cause unexpected problems and is best avoided.

- A simple approach to allocating NumPy arrays (see below) is to use `numpy.zeros()` which automatically assigns all array elements as zero. This will probably make more sense when you have read the section on Advanced Python, below.

3 The Python programming language: a very brief introduction

3.1 Variables and Objects

See Kinder and Nelson ch. 2.1 - 2.3; Hill ch. 2.2 - 2.4

3.1.1 Introduction

Variables represent computer memory spaces which store data. A variable has a name, which usually consists of one or more letters or numbers, beginning with a letter, and a value. Variables are created automatically, as needed. e.g.:

[5]:
```
i=1212
j=3
k=i*j
x=34523.34535
part2=1.234e21 # i.e. 1.234 times 10 to the power 21
z = 1+2j  # j is the square root of -1
name1 = 'Fred'
print (i, j, k, x, part2, z, name1)
```

```
1212 3 3636 34523.34535 1.234e+21 (1+2j) Fred
```

The variables above are called `i`, `j`, `k`, `x`, `part2`, `z` and `name1`.

- Variables in Python are also "objects".
- An object contains *attributes* including the value of the variable, and *methods* which are functions returning more information about the object.
- A list of the available methods and attributes for any object is given by the `dir(object_name)` command. For example, if you execute the following code, you will see a list of almost 50 methods and attributes, MOST OF WHICH YOU CAN IGNORE. The useful ones in the list are the last three: i.e. the method `z.conjugate()` , and the attributes `z.real` and `z.imag`.

[6]:
```
z = 1 + 2j
dir(z)
```

[6]:
```
['__abs__',
 '__add__',
 '__bool__',
 '__class__',
 '__delattr__',
 '__dir__',
 '__divmod__',
 '__doc__',
 '__eq__',
 '__float__',
 '__floordiv__',
 '__format__',
 '__ge__',
```

```
 '__getattribute__',
 '__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__',
 '__rpow__',
 '__rsub__',
 '__rtruediv__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__sub__',
 '__subclasshook__',
 '__truediv__',
 'conjugate',
 'imag',
 'real']
```

If we look again at the complex variable `z`, we can see the operation of the methods and attributes as follows.

[7]:
```
z = 1+2j  # j is the square root of -1
print (z, z.real, z.imag, z.conjugate())
```

```
(1+2j) 1.0 2.0 (1-2j)
```

Note that `z.conjugate()` is a method (and has brackets) because it needs to be calculated, whereas `z.real` and `z.imag` are attributes (no brackets) that are stored.

3.1.2 Different types of data

The basic data types:

- **integer:** (type `int`) for storing positive or negative integers. Integers are stored exactly, using as much memory as necessary to represent all of the digits.
- **floating point:** (type `float`) used to represent floating point numbers. The memory used is fixed, with typically only 15 significant figures available.

- **complex:** using the format `1+2j`, where `j` = $\sqrt{-1}$. Note the lack of any operator between the imaginary part and `j`. The `complex` object is constructed from 2 `float` objects. (Note the possibility of confusion between `j` representing $\sqrt{-1}$ and `j` representing a variable).
- **string:** (type `str`) for storing a string of characters e.g. 'Fred', 'Hello World!'.

You will find all of the above useful.

More sophisticated data types:

- **list:** a sequence of arbitrary objects, enclosed between square brackets e.g. `[1, 2.3, "Hello"]`.
- **tuple:** like a list but cannot be changed after creation (immutable), specified using round brackets e.g. `("AAPL", 3.14, 100)`.
- **set:** an unordered collection of objects, created with the `set()` function.
- **dictionary:** a type of table that associates names with values.

You will mostly just need the `list` and `tuple` types from the above.

3.1.3 Changing data types

The functions `int()`, `float()` and `str()` are used to convert between data types. For example:

[8]:
```
x = 3.14159
y = int(x)
z = str(x)
w = float(y)
print (x, y, z, w)
```

```
3.14159 3 3.14159 3.0
```

`y` is the integer part of `x`, and `w` is `y` turned back into floating point.

It is not obvious in the above that `z` is a string. However, the following sum should make it clearer:

[9]:
```
u = 'A string version of pi: ' + z
print (u)
```

```
A string version of pi: 3.14159
```

3.1.4 Object methods

This is best illustrated by example. If the variable `z` is a complex number, it is also a complex number *object*; the complex conjugate of `z` is available using the `conjugate()` *method*, as in:

[10]:
```
z = 1 + 2j
print ( z.conjugate() )
```

```
(1-2j)
```

- Note the need for brackets even though `z.conjugate()` has no arguments. They are needed because methods are functions which compute and return information relating to the object.
- Whenever an object is created, its methods will be available using the syntax: `object_name.method()`.

3.1.5 Object attributes

In addition, objects possess *attributes*, which are stored and do not need to be computed using a method. In the case of the complex number object, attributes would include the value of the variable, and the values of its real and imaginary parts. Attributes are used with a similar syntax to methods, but without the brackets i.e.

`object_name.attribute`. For example, attributes of the complex object would include the real and imaginary parts:

[11]:
```
print ( z.real, z.imag, z.conjugate().imag )
```

```
1.0 2.0 -2.0
```

3.2 Writing code

See Kinder and Nelson ch. 1 & 3.3.4; Hill ch.1.

3.2.1 Case Sensitivity

- **NOTE** that Python is case sensitive !!
- So `width` and `Width` are two different variables.
- also `Print()` is not a function in standard Python but `print()` is.

3.2.2 Comments

- Appropriate use of comments is a very important element of good programming style.
- Comments are simple English statements which explain what a particular piece of code does, or gives other information: For example – who wrote the code.
- You can also use comments to temporarily take out bits of unused code (e.g. for debugging or other testing purposes).
- There are two ways of making comments, these are illustrated below:

[12]:
```
#
# This program written 6th April 2016 by SH
#
# This is a single line comment.
# This is another.
#
# This is the definition of a function:
def myfunc(a,b):
    """This function returns the sum of the squares of a and b. This text
    can be split across several lines, and acts as an extended comment.
    However, it has another use, when used with the `help()` command (see below)."""
    c = a*a + b*b
    return c  # Comments can appear anywhere in a line

# Test the function:
print( myfunc(3.0, 4.0), myfunc(1,2))  # The function works for integers and floats
```

```
25.0 5
```

The comment between the double triple-double quotations is known as a `docstring` because it provides documentation for the function. It can be accessed as follows:

1. Using the help command, which produces a nicely formatted piece of text;
2. Printing the `__doc__` attribute of the `myfunc` object.

[13]:
```
print('------------------------------------------------------------')
print('---> This is what you get by typing "help (myfunc)" <---')
print('------------------------------------------------------------')
```

```
help (myfunc)

print('-----------------------------------------------------------')
print('---> This is what you get by printing "myfunc.__doc__" <---')
print('-----------------------------------------------------------')
print(myfunc.__doc__)

print('-----------------------------------------------------------')
```

```
------------------------------------------------------------
---> This is what you get by typing "help (myfunc)" <---
------------------------------------------------------------
Help on function myfunc in module __main__:

myfunc(a, b)
    This function returns the sum of the squares of a and b. This text
    can be split across several lines, and acts as an extended comment.
    However, it has another use, when used with the `help()` command (see
below).

------------------------------------------------------------
---> This is what you get by printing "myfunc.__doc__" <---
------------------------------------------------------------
This function returns the sum of the squares of a and b. This text
    can be split across several lines, and acts as an extended comment.
    However, it has another use, when used with the `help()` command (see
below).
------------------------------------------------------------
```

3.2.3 Floating Point Limits

These depend on the computer and the version of Python being used. You can find limits for your system by importing the `sys` module, and looking at appropriate attributes of `sys.float_info` as follows:

[14]:
```
import sys

print( 'Smallest fractional difference between two floats: ', sys.float_info.epsilon )
print( 'Number of digits precision: ', sys.float_info.dig )
print( 'Largest floating point number: ', sys.float_info.max )
print( 'Smallest floating point number: ', sys.float_info.min )
```

```
Smallest fractional difference between two floats:  2.220446049250313e-16
Number of digits precision:  15
Largest floating point number:  1.7976931348623157e+308
Smallest floating point number:  2.2250738585072014e-308
```

It is important to be aware of these limits when writing programs. Having only 15 digits of precision can lead to numerical errors in poorly crafted algorithms, while exceeding the largest or smallest allowed numbers can lead to a program crashing.

3.3 Input and Output

See Kinder and Nelson ch. 2.3 & 3.4; Hill ch. 2.3.

3.3.1 Writing to screen

Use the `print()` function to write output to the screen, as already demonstrated above. However, to gain more control over the operation of `print()`, you need to be able to be able to **format** the numbers being printed. There are several ways to achieve this. Below we illustrate the `format()` method that belongs to any string variable, and the % operator:

[15]:
```
pi = 3.14159265358979
print('1) Pi is approximately ',pi)
print('2) Pi is approximately ' + str(pi))
print('-----')
#
# The format() method
#
print('3) {} is roughly {:.5f}'.format('Pi',pi))  # Format ".5f" means floating point␣
 ↪with
                                                  # 5 decimal places
print('4) {} is roughly {:9.5f}'.format('Pi',pi))  # Format "9.5f" means floating␣
 ↪point with
                                                  # 5 decimal places, and 9 characters in␣
 ↪total
print('5) {1:.5f} is roughly the value of {0}'.format('Pi',pi))  # Note order of␣
 ↪arguments
print('-----')
#
# The percent operator
#
print('6) %s is around about %.4g' % ('pi',pi))
a = '7) %.4g is around about the value of %s'
b = a % (pi,'pi')
print(b)
print('-----')
#
# Different field widths
#
i = 5
print ('%1d' % (i))
print ('%2d' % (i))
print ('%3d' % (i))
print ('%4d' % (i))
print ('%5d' % (i))
```

```
1) Pi is approximately  3.14159265358979
2) Pi is approximately 3.14159265358979
-----
3) Pi is roughly 3.14159
4) Pi is roughly   3.14159
5) 3.14159 is roughly the value of Pi
-----
6) pi is around about 3.142
7) 3.142 is around about the value of pi
-----
5
 5
```

```
5
 5
  5
```

- The `format()` method belongs to all string objects. Arguments are inserted into the string in the order specified between the curly brackets e.g.`{1:.5f}` means *insert the 2nd argument (counting from zero) at this point, format it as a float with 5 d.p.* It's use is **not** restricted to the `print()` function.
- The % operator will operate on any string as well, but the arguments must be specified in the order they are needed.
- Other format specifiers are:
  - `d` for a decimal integer
  - `b` for a binary integer
  - `x` for a hexadecimal integer
  - `s` for a string
  - `g` for general format in the smallest amount of space
  - `f` for a floating point number

3.3.2 Input from keyboard

Use the `input()` function to obtain a string from the keyboard, which you must convert to a number using `float()` or `int()`. Try the following program:

[16]:
```
MyNumber = -1
while MyNumber < 0 or MyNumber > 10:
    MyInput = input('Enter a number in the range 0 to 10: ')
    MyNumber = float(MyInput)
    print('You entered the number: ',MyNumber)
print('Thank you')
```

```
Enter a number in the range 0 to 10: 3
You entered the number:  3.0
Thank you
```

Note the use of a `while` loop to ensure the user enters a number in the correct range (see later).

3.4 Doing Maths

See Kinder and Nelson ch. 1.3, 1.4 & 3.2; Hill ch. 2.2.

3.4.1 Mathematical Operators

Operator	Operation	Operator	Operation
+	Addition	//	Division (integer)
-	Subtraction	%	Modulus (remainder)
*	Multiplication	**	Exponentiation (power)
/	Division (floating point)		

Examples:

[17]:
```
i = (3 + 2) * 2      # i.e. i = 10
j = 3
print ("i%j = ",i%j) # divides i by j and returns the remainder (e.g., 10%3 = 1)
```

```
print ("i/j = ",i/j) # floating point division, result is 'float'
print ("i//j = ",i//j) # integer division, result is 'int'
print ("i**j = ",i**j) # raising to a power
```

```
i%j =  1
i/j =  3.3333333333333335
i//j =  3
i**j =  1000
```

3.4.2 Operator Shorthand

Operator	Example	Equivalent
=	a = b	a = b
+=	a += b	a = (a+b)
-=	a -= b	a = (a-b)
*=	a *= b	a = (a*b)
/=	a /= b	a = (a/b)
%=	a %= b	a = (a%b)

In each of the above statements, the right-hand-side is evaluated taking the current values of `a` and `b`, and the result is placed back in the variable `a`. For example:

[18]:
```
a = 100
b = 9
a += b
print (a)
a -= b
print (a)
a *= b
print (a)
a /= b
print (a)
a %= b
print (a)
```

```
109
100
900
100.0
1.0
```

3.4.3 Mathematical functions

Basic mathematical functions are not available in basic Python, but need to be added in by importing the math module. There are several ways to achieve this, which each have slightly different effects:

`import math`	this imports all math functions, prefixing each with 'math'. i.e. math.sin(), math.cos().
`import math as mt`	as above, but functions are prefixed with 'mt'. i.e. mt.sin(), mt.cos().
`from math import *`	imports all math functions with no prefix i.e. sin(), cos().
`from math import sin,cos`	as above but only the named functions are imported.

Here is a brief list of some of the functions available. There are many more, which can be found in the Python math module documentation.

Function	Purpose
math.sin(x), math.cos(x), math.tan(x)	trig functions
math.asin(x), math.acos(x), math.atan(x)	inverse trig functions
math.sinh(x), math.cosh(x), math.tanh(x)	hyperbolic trig functions
math.exp(x), math.log(x), math.log10(x)	$e^x,\ \ln(x),\ \log_{10}(x)$
math.sqrt(x), math.hypot(x,y), math.factorial(x)	$\sqrt{x},\ \sqrt{x^2+y^2},\ x!$

N.B. For trig functions, all arguments are in *radians*. Examples:

[19]:
```
import math
x = math.pi / 4 # math.pi = 3.14159...
print("sin(pi/4) = ",math.sin(x))
print("cos(pi/4) = ",math.cos(x))
print("tan(pi/4) = ",math.tan(x))
```

```
sin(pi/4) =  0.7071067811865475
cos(pi/4) =  0.7071067811865476
tan(pi/4) =  0.9999999999999999
```

3.4.4 Functions of Complex Numbers

These follow the same pattern as the `math` functions described above. However, they are contained in a module called `cmath` (see the Python cmath module documentation) and therefore have a different prefix

Function	Purpose
cmath.sin(z), cmath.cos(z), cmath.tan(z)	complex trig functions
cmath.asin(z), cmath.acos(z), cmath.atan(z)	complex inverse trig functions
cmath.sinh(z), cmath.cosh(z), cmath.tanh(z)	complex hyperbolic trig functions
cmath.exp(z), cmath.log(z), cmath.log10(z)	$e^z,\ \ln(z),\ \log_{10}(z)$
cmath.sqrt(z)	$\sqrt{z}$
abs(z), cmath.phase(z)	Magnitude, phase

N.B. the built-in function `abs(z)` returns the modulus of the complex number `z`. There is no `cmath` equivalent. Here is an example:

[20]:
```
import cmath

z1 = 1 + 1j # remember j is root(-1)

print("Modulus of z1: ",abs(z1))
print("Argument of z1: ",cmath.phase(z1)," radians")
print("Square root of z1: ",cmath.sqrt(z1))
```

```
Modulus of z1:  1.4142135623730951
Argument of z1:  0.7853981633974483  radians
Square root of z1:  (1.09868411346781+0.45508986056222733j)
```

3.4.5 Random numbers

These are available using a module called `random` (see the Python random module documentation).

See also Kinder & Nelson ch. 6.2 & 7; Hill ch. 6.7 for alternative approaches.

Random numbers are not really random, but follow a pseudo-random sequence generated by algorithm. Every pseudo-random sequence must be given an initial `seed` to start it off. It is common to use the system time (the time on the computer's internal clock) as the random number seed.

The random module contains functions for generating random numbers from various different distributions. The simplest are *rectangular* distributions of integers and floating point numbers. For example:

```
random.seed()       # initialises the random number seed
random.randint(a,b) # random integer between (and including) a and b
random.random()     # random floating point number between 0.0 and 1.0
random.uniform(a,b) # random floating point number between a and b
```

[21]:
```
import random

i=0
a, b = 1, 6  # shorthand for saying a = 1 and b = 6
random.seed()

while i < 10:  # repeat this 10 times
    print(i, random.randint(a,b), random.random()) # generate integers between 1 and 6␣
 ↪and
                                                   # floating point numbers between 0.
 ↪0 and 1.0.
    i += 1
```

```
0 4 0.8685290999745287
1 5 0.4128911398938969
2 1 0.7894114848229616
3 6 0.2694931721406064
4 2 0.49163680244566943
5 1 0.5941650642949519
6 5 0.41478622379529617
7 1 0.26482149700406765
8 6 0.48509933771741776
9 4 0.8228245514682924
```

3.4.6 Bytes, Bits and other storage

- Bit = 1 or 0 , this is the most basic element of number storage
- Byte = 8 bits, i.e, 00000000 to 11111111 in binary = 0..255 in decimal or 0..FF in Hexadecimal
- All numbers are stored and manipulated as bits in computers. Different number types (e.g. integers or floats) are stored differently: even if they are the same number e.g. 1 and 1.0.
- Small positive integers can be stored as a simple byte. However, Python uses as many bytes as necessary to store each integer exactly.
- Floating point numbers are typically stored in 8 bytes (64 bits) which means precision is limited to around 15 decimal digits only. The 64 bits are divided as follows (IEEE 754 standard):
  - Sign: 1 bit (represents + or -).
  - Exponent: 11 bits (allows numbers in range 10^{-308} to 10^{308}).
  - Mantissa (or fractional part of number): 52 bits (allows 15 digits of precision, see section 3.2.3 above).

3.5 Handling strings, lists, tuples and more

See Kinder & Nelson ch. 2.2, 2.3; Hill ch. 2.3, 2.4.

3.5.1 Strings

- Generally, a string is a collection of characters: e.g. ‘‘`Hello`’’.
- They are usually stored in memory as their ASCII character codes:

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

- So “Hello” is stored as: 72,101,108,108,111 in decimal.
- In Python, each character will occupy one or more bytes of memory, depending on the representation being used.

Assignment:

[22]:
```
string1 = "Hello" # can use double quotes
string2 = 'World' # or single quotes
```

```
string3 = "-o-"*4
string4 = string3 + string1 + ' ' + string2 + string3
empty = "" # an empty string of zero length
print(string4)
```

```
-o--o--o--o-Hello World-o--o--o--o-
```

Converting numbers to strings:

- Use the `str()` function. e.g.

[23]:
```
s1 = str(100)
s2 = str(100.0)
s3 = str(1e20)
s4 = s1 + ' ' + s2 + ' ' + s3
print(s4)
```

```
100 100.0 1e+20
```

`s4` now equals "100 100.0 1e+20"

Special "escape" sequences for extra format control

- To include a quotation mark in a string, use the opposite quotes to define the string e.g.
  - s1 = "Fred's World"
  - s2 = ' "Hello", he said'
- Other non-printing characters may be included by using the appropriate "escape" sequence, usually indicated by a backslash, \. Here are some common ones:

Escape Sequence	Definition
\'	single quote
\"	double quote
\n	new line
\t	tab
\\	the backslash itself

For example:

[24]:
```
s1 = 'Fred\'s\nWorld'
print(s1)
```

```
Fred's
World
```

Slicing and dicing strings

- Can be useful for reading in data from files (more on this later).
- Individual characters in a string can be indexed (referred to) using square brackets, and counting from zero.
- For example:

[25]:
```
s1 = "Captain Marvel" # (c) Marvel Comics.
print(s1[0], s1[4], s1[-1])
```

```
C a l
```

- Note how the negative index counts back from the far end of the string.
- A section of the string is obtained using `s1[i:j]`, which will include the `i`th character but exclude the `j`th. For example:

[26]:
```
print(s1[3:6])
print(s1[:7])
print(s1[7:])
```

```
tai
Captain
 Marvel
```

- Optionally, a third index may be used to specify the `step` used:

[27]:
```
print(s1[::2]) # gives every other character of the whole string
```

```
CpanMre
```

- To find whether a string includes a particular sub-string use the `in` operator:

[28]:
```
print('Captain' in s1)
print('Sergeant' in s1)
```

```
True
False
```

String Methods

- Strings are *immutable* which means they cannot be redefined once created (although they can be completely replaced).
- To find the length of a string (in characters) use the built-in `len()` method:

[29]:
```
print("Length of s1 is ",len(s1)," characters.")
```

```
Length of s1 is  14  characters.
```

`len()` also works with lists and other multi-element objects.

- There are many *methods* available for operating on strings e.g. we have already met the `format()` method when printing strings using `print()`.
- Here is a selection (see textbooks for more):

String Method	Definition
index (*sub_string*)	returns the lowest index of the string where *sub_string* occurs
split ([*separator*])	returns a `list` of sub_strings which were separated in the original string by *separator*. If *separator* is not given, the sub_strings are demarcated by spaces. Lists will be covered in the following section.
isalpha ()	returns `True` if the string is formed entirely of alphabetic characters
isdigit ()	returns `True` if the string is formed entirely of numeric characters

Other functions include `upper()`, `lower()` and `title()` for changing the case of the string and `replace(old,new)` for swapping occurences of `old` with `new` within a string.

In addition, we have already met the following functions for converting strings to integers or floats:

- `int(string)` converts a string to an integer
- `float(string)` converts a string to a floating point numbers
- Here are some examples. In each case, the original string is unchanged:

[30]:
```
s2 = "Guardians of the Galaxy" # (c) Marvel Comics
print(s2.index("Galaxy"))
```

```
17
```

[31]:
```
s3 = s2.split(" ")
print("There are ",len(s3)," substrings")
print("  substring 1: ",s3[0])
print("  substring 2: ",s3[1])
print("  substring 3: ",s3[2])
print("  substring 4: ",s3[3])
```

```
There are  4  substrings
  substring 1:  Guardians
  substring 2:  of
  substring 3:  the
  substring 4:  Galaxy
```

[32]:
```
# Example use of string methods
#
# 'for', 'range' and 'if, elif, else' will be covered in later sections

string1 = "10 Green Bottles"

for i in range(len(string1)):
    if string1[i].isalpha():
        print("Character %2d is \"%s\" and is alphabetic" % (i,string1[i]))
    elif string1[i].isdigit():
        print("Character %2d is \"%s\" and is numeric" % (i,string1[i]))
    else:
        print("Character %2d is \"%s\" and is not alphanumeric" % (i,string1[i]))
```

```
Character  0 is "1" and is numeric
Character  1 is "0" and is numeric
Character  2 is " " and is not alphanumeric
Character  3 is "G" and is alphabetic
Character  4 is "r" and is alphabetic
Character  5 is "e" and is alphabetic
Character  6 is "e" and is alphabetic
Character  7 is "n" and is alphabetic
Character  8 is " " and is not alphanumeric
Character  9 is "B" and is alphabetic
Character 10 is "o" and is alphabetic
Character 11 is "t" and is alphabetic
Character 12 is "t" and is alphabetic
Character 13 is "l" and is alphabetic
```

```
Character 14 is "e" and is alphabetic
Character 15 is "s" and is alphabetic
```

[33]:
```
# Another string example
#
#Could be used when reading data from a file where numbers and text are mixed

string1 = "Width=34.5565"  # load values into string1

numbit = string1[string1.index('=')+1:]  # find the position of the = sign
                                         # then copy the substring into numbit
width = float(numbit)  # do the conversion, if numbit is not numeric then the␣
 ↪conversion will fail

print("The number was %f" % (width))  # display the result of the conversion
```

```
The number was 34.556500
```

3.5.2 Lists

- A *list* is an ordered array of objects, separated by commas within square brackets. For example:

[34]:
```
MyList = [1, 3.1415, "Tuesday", 1+1j]
print(MyList)
```

```
[1, 3.1415, 'Tuesday', (1+1j)]
```

- Elements in lists are referenced using square brackets, numbering from zero:

[35]:
```
print(MyList[0], MyList[3])
```

```
1 (1+1j)
```

- Lists can contain other lists:

[36]:
```
MyNewList = [3.162, MyList, 1.414]
print(MyNewList)
```

```
[3.162, [1, 3.1415, 'Tuesday', (1+1j)], 1.414]
```

- Nested lists and strings can be accessed using repeated square brackets. e.g. for the 2nd item in `MyNewList[1]` use:

[37]:
```
MyNewList[1][1]
```

[37]:
```
3.1415
```

- and for the 4th letter of the 3rd item in MyNewList[1] use:

[38]:
```
MyNewList[1][2][3]
```

[38]:
```
's'
```

We met lists previously, when handling strings. Recall:

[39]:
```
s2 = "Guardians Of The Galaxy"
s3 = s2.split(" ") # s3 is a list of strings (words)
print(s3)
```

```
['Guardians', 'Of', 'The', 'Galaxy']
```

Lists are *mutable* which means they can be changed, lengthened and shortened after creation (unlike strings which are immutable). For example:

[40]:
```
s3[0] = "Stars"
print(s3)
```

```
['Stars', 'Of', 'The', 'Galaxy']
```

Here are some examples of `list` methods (see textbooks for other examples):

[41]:
```
s3.insert(3,"local")  # insert a new element at a specified location
print(s3)
```

```
['Stars', 'Of', 'The', 'local', 'Galaxy']
```

[42]:
```
s3.append("Masters")  # add an element to the end of the list
print(s3)
```

```
['Stars', 'Of', 'The', 'local', 'Galaxy', 'Masters']
```

[43]:
```
s3.remove("Stars")  # remove the first occurrence of a particular element
s3.remove("local")  # remove the first occurrence of a particular element
print(s3)
```

```
['Of', 'The', 'Galaxy', 'Masters']
```

[44]:
```
s3[2] = 'Universe'
s3.sort()  # sort in the list in place (this is alphabetical...)
print(s3)
```

```
['Masters', 'Of', 'The', 'Universe']
```

Lists of numbers can be useful for storing data, for example, for graphs. However, in general lists are quite inefficient both in their use of memory and the time taken to access individual elements. For serious numerical work, therefore, it is recommended that you use NumPy arrays instead of lists (see later, section 6.1).

3.5.3 Tuples, sets and dictionaries

Tuples A `tuple` is similar to a list but cannot be altered after creation i.e. it is immutable. Tuples are indicated through the use of round brackets in their definitions, and may be indexed in the same way as strings and lists (using square brackets). For example:

[45]:
```
My_Tuple = (1.23, 4, 'five', [6,7,8])
print(My_Tuple,'\n',My_Tuple[2],'\n',My_Tuple[3])
```

```
(1.23, 4, 'five', [6, 7, 8])
 five
 [6, 7, 8]
```

We will come across occasional uses for tuples. One example is in initialising variables:

```
x, y, z = 1.0, 2.0, 3.0
```

makes the assignment x=1.0, y=2.0, z=3.0, while:

```
u, v = v, u
```

swaps the values of the variables u and v.

Dictionaries and Sets

- A `Dictionary` is an object that consists of associated *key-value* pairs. For example:

[46]:
```
densitySI = {'water': 1000.0, 'gold': 19320.0, 'mercury': 13546.0, 'air': 1.2,␣
 ↪'helium': 0.179}
print("The density of water is:",densitySI['water'],"kg/m3")
print("The density of gold is:", densitySI['gold'],"kg/m3")
```

```
The density of water is: 1000.0 kg/m3
The density of gold is: 19320.0 kg/m3
```

- Note the use of curly brackets in the definition of `densitySI`.
- A `set` is a *mutable* object that consists of an unordered collection of unique items. For example the set defined by `{1, 2, 3, 3, 2, 3, 4, 5, 4, 2, 1}` contains 5 unique entries:

[47]:
```
MySet = {1, 2, 3, 5, 2, 3, 4, 5, 4, 2, 1}
print(MySet)
```

```
{1, 2, 3, 4, 5}
```

- Note the use of curly brackets in the definition of `densitySI`.
- Dictionaries and Sets are covered in more detail in the recommended textbooks and will only be of limited use in the Computational Physics assignments.

4 Program Control

4.1 Loops

See Kinder & Nelson ch. 3.1; Hill ch. 2.4.3, 2.5.

The big advantage of computer programs over manual calculation is the ability to repeat operations many times, very quickly. To achieve this, some sort of a program loop is required. There are two types of loop structure in common use in Python: the `for` loop and the `while` loop. Both will be explained below.

4.1.1 "for" loops

The `for` loop requires the idea of an `iterable` object, so we begin with that. An iterable object is essentially an object that contains many elements which may be cycled through in sequence. `tuples` and `lists` are examples of iterable objects. The basic structure is as follows:

```
for *item* in *iterable_object*:
    # do some stuff
```

- `*item*` is an element from the iterable object, taken in sequence.

- `*iterable_object*` could be a list of numbers, `[1, 2, 3, 4, 5]`, or other items, `['England', 'Northern Ireland', 'Scotland', 'Wales']`.
- The colon `":"` indicates the start of the loop.
- All the lines to be executed within the loop must be indented, and by an identical amount (e.g. one tab, 4 spaces etc). Tabs and spaces cannot be mixed. The end of the loop will be the end of the indented section.
- Incorrect use of indents can be a cause of great anguish to the novice Python programmer.

Here are some examples.

[48]:
```
for Country in ['England', 'Northern Ireland', 'Scotland', 'Wales']:
    print(Country)
```

```
England
Northern Ireland
Scotland
Wales
```

In the above, a variable called `Country` is created and assigned, in turn, to each of the names in the list.

In the following example, the variable `i` is set to each of the list of integers shown:

[49]:
```
for i in [1,3,2,6,5]:
    print("i:",i,"i^2:",i*i,"i^3:",i*i*i)
print("End of loop")
```

```
i: 1 i^2: 1 i^3: 1
i: 3 i^2: 9 i^3: 27
i: 2 i^2: 4 i^3: 8
i: 6 i^2: 36 i^3: 216
i: 5 i^2: 25 i^3: 125
End of loop
```

Note, in the following, the use of the iterable `range()` object. `range()` has three forms, as indicated. `range(n)` produces a set of integers from `0` to `n-1`:

[50]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

`range(m,n)` produces integers from `m` to `n-1`:

[51]:
```
for i in range(2,7):
    print(i)
```

```
2
3
4
5
6
```

and `range(m,n,s)` produces integers from `m` in steps of `s`, ending one value before `n`:

[52]:
```
for i in range(10,0,-2):
    print(i)
```

```
10
8
6
4
2
```

4.1.2 "while" loops

We have already met the `while` loop. The basic structure is:

```
while *condition_is_true*:
    # do some stuff
```

As with the `for` loop, the `while` loop will repeat only the indented part of the code. All indents must be an identical number of tabs or spaces. For example:

[53]:
```
i=0  # Initialise the variable
while (i<10):
    print("i=",i,"i^2=",i*i)
    i+=2 # increment the variable
```

```
i= 0 i^2= 0
i= 2 i^2= 4
i= 4 i^2= 16
i= 6 i^2= 36
i= 8 i^2= 64
```

Another example: a very poor method of numerical integration. This short program will perform the integral:

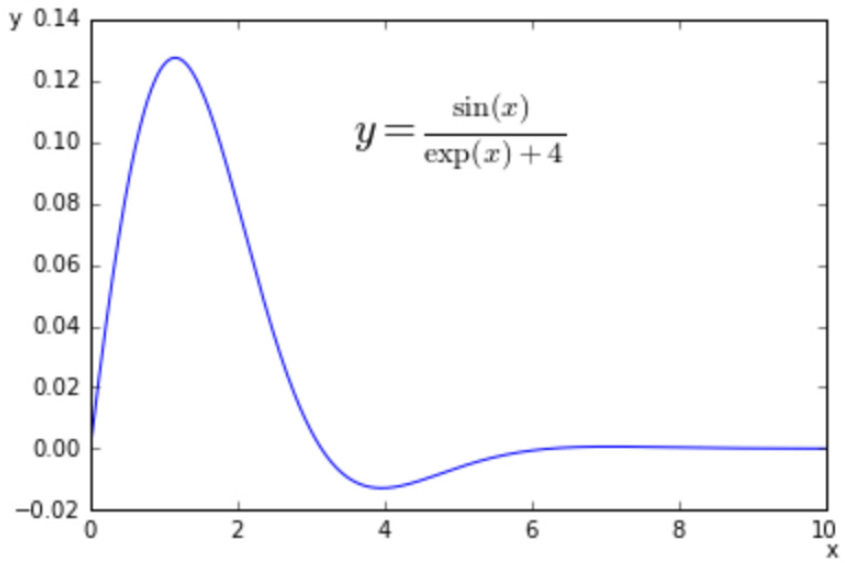
$$\int_0^\infty \frac{\sin(x)}{\exp(x)+4}\mathrm{d}x,$$

where infinity is approximated by 10.

[54]:
```
#
# A method to integrate a function by summing lots of rectangles
#

import math

xmin = 0.0         # minimum x value for integral
xinfinity = 10.0   # maximum x value for integral
numsteps = 10000   # number of rectangles to approximate integral
xstep = xinfinity / numsteps  # width of a rectangle, i.e. dx
x = xmin           # current x value
i = 0              # loop counter
sum = 0.0          # the current value of the integral

while (x<xinfinity):
    x = xmin + i*xstep
    sum += math.sin(x)/(math.exp(x)+4)*xstep
    i += 1

print("Value of integral is:",sum)
```

```
Value of integral is: 0.2090887803611044
```

For information, the function appears as follows. We will cover plotting later on.



4.1.3 Nested loops

Loops can be nested, by using multiple indentation. For example, the following short program prints products: $i \times j$ where $i = 1 \ldots 3$ and $j = 1 \ldots 3$:

[55]:
```
i = 0
while (i<3):
    i += 1
    j = 0
    print("-------------")
    while (j<3):
        j += 1
        print("|",i,"x", j,"=", i*j,"|")
print("-------------")
```

```
-------------
| 1 x 1 = 1 |
| 1 x 2 = 2 |
| 1 x 3 = 3 |
-------------
| 2 x 1 = 2 |
| 2 x 2 = 4 |
| 2 x 3 = 6 |
-------------
| 3 x 1 = 3 |
| 3 x 2 = 6 |
| 3 x 3 = 9 |
-------------
```

4.1.4 Breaking out of a loop

- You can break out of a loop using the `break` statement. Here is the previous example with a break when the product gets too big:

[56]:
```
i = 0
while (i<3):
    i += 1
    j = 0
    print("-------------")
    while (j<3):
        j += 1
        if i*j > 4:
            print("| Too big!! |")
            break
        print("|",i,"x", j,"=", i*j,"|")
print("-------------")
```

```
-------------
| 1 x 1 = 1 |
| 1 x 2 = 2 |
| 1 x 3 = 3 |
-------------
| 2 x 1 = 2 |
| 2 x 2 = 4 |
| Too big!! |
-------------
| 3 x 1 = 3 |
| Too big!! |
-------------
```

4.2 Decision Making and Branching

See Kinder & Nelson ch. 3.4; Hill ch. 2.5.

The other control structure essential to any programming language is the conditional statement, which allows decisions to be made based on the values of variables. Conditional statements are usually referred to as `if` statements.

4.2.1 The `if` statement

- Program branching is controlled using conditional statements.
- These are also known as `if` statements.
- The basic structure is:

```
if ... elif ... else
```

i.e.:

```
if *expression_is_true*:
    {do this}
elif *other_expression_is_true*:
    {do that}
else:
    {do the other}
```

- Note the use of indentation again to control which lines are executed.
- Example:

[57]:
```
for i in range(10):
    if (i < 3):
        print(i,"is less than 3")
    elif (i > 6):
        print(i,"is greater than 6")
    else:
        print(i,"is in the middle")
```

```
0 is less than 3
1 is less than 3
2 is less than 3
3 is in the middle
4 is in the middle
5 is in the middle
6 is in the middle
7 is greater than 6
8 is greater than 6
9 is greater than 6
```

- Complex `if` statements may be built up using the following operators

Operator	Function
==	Equality (note use of double = sign)
!=	Not equals (inequality)
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
and	logical AND
or	logical OR
not	logical NOT

- Examples:
  - `if (j==3) or (j==5):` True if j = 3 or 5 (note double == signs)
  - `if j>3 and j<6:` True if j > 3 but less than 6
  - `if j != 3:` True if j not equal to 3
  - `if ((j>3)and(j!=6)) or (j==1):` True if j is greater than 3 and not equal to 6, OR if j equals 1

4.2.2 Equality tests on floats

- Consider the following:

[61]:
```
MyInput = input('Enter the square of 0.1: ')
MyNumber = float(MyInput)
```

```
if MyNumber == 0.1*0.1:
    print("You were correct")
else:
    print(MyNumber,"is not the square of 0.1")
```

```
Enter the square of 0.1: 0.01
0.01 is not the square of 0.1
```

- You might think that if you enter 0.01 at the prompt then the `if` statement would be true.
- However, it depends on the version of Python and the system precision.
- On my system, 0.1*0.1 = 0.010000000000000002; hence the test fails.
- So, it is best to **always avoid equality tests on floating point numbers**; instead use something like this:

[62]:
```
import sys
MyInput = input('Enter the square of 0.1: ')
MyNumber = float(MyInput)

if abs(MyNumber - 0.1*0.1) < sys.float_info.epsilon:
    print("You were correct")
else:
    print(MyNumber,"is not the square of 0.1")
```

```
Enter the square of 0.1: 0.01
You were correct
```

- The function `abs()` takes the absolute value (without sign) of a number.
- sys.float_info.epsilon is the smallest fractional difference between two floats that can be distinguished. It is usually around 10^{-15} depending on your computer.
- Instead of `sys.float_info.epsilon` you can use a larger *tolerance* of your own, depending on your application.

5 Functions

See Kinder & Nelson ch. 6.1; Hill ch. 2.7.

- Functions provide a useful way of reducing the amount of code you need to create, by allowing re-use of certain key parts.
- Functions should always be defined at the top of your script, **before** the main part of the program. This way you can keep them separate and simplify the overall structure of your program.

5.1 Basic Functions

- Use **functions** to simplify code structure and also to maximise reuse of code.
- Example:

[63]:
```
def volume (height, width, length ):
    """
    This is called a function definition.  You should place it before the
    main program. This block of text is intended to describe the
    operation of the function, and will be accessed using the
    'help(volume)' command.
    """
    return height*width*length
```

```
#
# Here is the main program
#
dim1 = 23.5
dim2 = 789.232
dim3 = 1.099
print("The volume is",volume(dim1,dim2,dim3))
#
# Print the 'help' text...
#
help(volume)
```

```
The volume is 20383.100247999995
Help on function volume in module __main__:

volume(height, width, length)
    This is called a function definition.  You should place it before the
    main program. This block of text is intended to describe the
    operation of the function, and will be accessed using the
    'help(volume)' command.
```

- Note the following features of the function definition:
  - Function declaration using the `def` command. Give your functions memorable names so it is obvious what they do.
  - Argument list: within the function, the arguments have names that are local to the function **i.e. they are not known outside the function**. This can be both useful and confusing. It is useful because it means you can re-use variable names within more than one function without them interfering with each other, but it is also confusing for much the same reason.
  - However, variables from the main program **are** available within the function, which can be very confusing. Therefore it is best practice to pass in **all** the required variables through the argument list, and avoid re-use of variables names.
  - The *body* of the function is indicated by indentation, as with loops and conditional statements.
  - The `return` statement is used to return the **result** of performing the function. In the present case that would be the *volume*.

5.2 Function `return` values

- Generally speaking, every function should return a value through the `return` statement.
- Sometimes you want to return several values e.g. the components of a vector. This is readily achieved, as you can pass back any object, including `tuples` and `lists`.
- If you don't want the function to return anything, Python will return an object called `None`.
- Consider the following examples:

[64]:
```
def printdoublex(x):
    """
    This rather un-intelligent function prints 2 times its argument
    and returns nothing.
    """
    print(2*x)
#
# Main program
#
height = 3.54
```

```
width = 7.42
length = 1.0e3
printdoublex(height)
printdoublex(length)
printdoublex(width)
```

```
7.08
2000.0
14.84
```

[65]:
```
def vector_add (vector1, vector2):
    """
    This function performs vector addition. The arguments, vector1 and
    vector2, should be list objects of length 3, and the value returned
    will be a similar list object of the same length.
    """
    vector3 = [0.0] * 3
    for i in range(3):
        vector3[i] = vector1[i] + vector2[i]

    return vector3
#
# Main program
#
a = (0, 0, 0)
b = (1, 2, 3)
c = vector_add (a, b)
print ("The vector sum of",a,"and",b,"is:",c)
```

```
The vector sum of (0, 0, 0) and (1, 2, 3) is: [1, 2, 3]
```

- Note that the inputs in this case were `tuples` while the output was a `list`.
- The code also works if the arguments are `lists`.
- Note the need to initialise `vector3` before assigning values to it.

[66]:
```
def vector_add2 (vector1, vector2):
    """
    This function performs vector addition. The arguments, vector1 and
    vector2, should be list objects of the same length, and the value
    returned will be a similar list object of the same length.
    """
    n = len(vector1)

    if n != len(vector2):
        print ("Error: input vectors have different lengths")
        return

    vector3 = [0.0] * n
    for i in range(n):
        vector3[i] = vector1[i] + vector2[i]

    return vector3
#
# Main program
```

```
#
a = (0, 0, 0, 0)
b = (1, 2, 3, 4)
c = (5, 6, 7)
print ("The vector sum of",a,"and",b,"is:",vector_add2(a,b))
print ("The vector sum of",b,"and",c,"is:",vector_add2(b,c))
```

```
The vector sum of (0, 0, 0, 0) and (1, 2, 3, 4) is: [1, 2, 3, 4]
Error: input vectors have different lengths
The vector sum of (1, 2, 3, 4) and (5, 6, 7) is: None
```

5.3 key-value pairs in function argument lists

- The arguments to a function must be specified in one of two ways:
  1. in the same order as the function definition
  2. in a different order but paired with a keyword
- The keyword is just the name of the argument in the function definition.
- This example also demonstrates the use of *default* argument values:

[67]:
```
def pythagoras (a=3.0, b=4.0):
    """
    Demonstration of the use of key-value pairs in function argument lists.
    This function also has default values set, so that some arguments may be
    omitted.
    """
    return math.sqrt(a*a + b*b)
#
# Main program
#
print (pythagoras())
print (pythagoras(b=5.0, a=12.0))
print (pythagoras(b=3.0))
```

```
5.0
13.0
4.242640687119285
```

6 More Advanced Python

In this section we will cover:

- Arrays and the NumPy module
- Vectorisation
- Writing to and from files
- Graph plotting

6.1 Arrays and the NumPy module

See Kinder & Nelson ch. 2.2 & 3.2; Hill ch. 2.4, 3.1.2 & 6.1.

6.1.1 Arrays using `list` objects

- Often, in writing programs with Physics applications in mind, the idea of an **array** is useful.
- An **array** is a list of numbers that can be referred to by an *index*.

- The `list` object provides one way of creating an array.
- Arrays can be used to store experimental data points for analysis, for storing values of function evaluated at a set of points, or for a range of other things.
- Arrays are generally very useful.
- For example, the following program lists a set of experimental data points:

[68]:
```
#
# Arrays : single dimension
#
# A five element list array will have indices from 0 to 4
#
data = [0.0] * 5

data[0] = 3.1
data[1] = 12.2
data[2] = 74.4
data[3] = 67.33
data[4] = 343.2

print("Element number 3 = ",data[3])

sum = 0.0
print("\nAll the elements:\n")
for i in range(5):
    print("Element %d %6.2f" % (i,data[i]))
    sum += data[i]

print('\nThe average data value is %6.2f' % (sum/5))
```

```
Element number 3 =  67.33

All the elements:

Element 0   3.10
Element 1  12.20
Element 2  74.40
Element 3  67.33
Element 4 343.20

The average data value is 100.05
```

- What happens if we try to use array element `data[5]` which is not defined?

6.1.2 Multi-dimensional arrays using lists

- Multi-dimensional arrays are possible, though rather cumbersome to set up.
- Here is an example using two dimensions:

[69]:
```
#
# Arrays : multiple dimension
#
# Declare an array of zeros with 5 rows and 2 columns.
# Start with a null list and 'append' each row:
#
myarray = []
```

```
for i in range(5):
    myarray.append([0.0]*2)
#
# assign new values to all array elements
#
for j in range(2):
    for i in range(5):
        myarray[i][j] = i*j

print("Array element 3,1 is",myarray[3][1])
print("Array row 3 is",myarray[3])
print("The whole array is",myarray)
```

```
Array element 3,1 is 3
Array row 3 is [0, 3]
The whole array is [[0, 0], [0, 1], [0, 2], [0, 3], [0, 4]]
```

6.1.3 Better arrays, using NumPy

- NumPy is a module with **lots** of useful functions for scientific work, including:
  1. Array handling
  2. File reading / writing
  3. Faster mathematical functions (than the built-in functions)
  4. Statistics
  5. Linear algebra
- We will use it mainly for the first and third applications in the list i.e. as a convenient way of creating arrays, and for it's range of fast mathematical functions.
- Apart from ease of creation, NumPy arrays have two other big advantages: they are **fast** and they are **vectorisable**. The consequences of this will become clear later on.

6.1.4 Creating arrays with NumPy

- NumPy arrays can be regarded as a multidimensional table of values indexed by a tuple of integers.
- All elements in a NumPy array must be of the same type (unlike Python lists).
- This, however, is entirely appropriate for scientific work.
- A NumPy array is physically stored as a contiguous block of memory, which makes it very efficient for passing between functions and for performing general algebraic manipulations on it.
- It is generally a good idea to import NumPy as `np` (this is the common approach in the recommended text books):

[70]:
```
import numpy as np
```

- NumPy arrays can be initialised with zeros or ones:

[71]:
```
a = np.ones(4)
print(a)
b = np.zeros( (3,3) ) # Note use of tuple for index list
print(b)
```

```
[1. 1. 1. 1.]
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

- NumPy arrays can be initialised as a unit matrix:

[72]:
```
c = np.eye(4)
print(c)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

- A more general form of initialisation is as follows:

[73]:
```
row_vector = np.array( [2.0, 2.4, 5.6])
column_vector = np.array( [[2.0], [2.4], [5.6]] )
print(row_vector)
print(column_vector)
```

```
[2.  2.4 5.6]
[[2. ]
 [2.4]
 [5.6]]
```

[74]:
```
matrix = np.array( [ [1.0, 1.4, 4.3], [3.2, 4.1, 2.3] ] )
print(matrix)
```

```
[[1.  1.4 4.3]
 [3.2 4.1 2.3]]
```

- To create an array of equally spaced values, e.g. for graph plotting, use the `np.arange` and `np.linspace` functions.
- For `np.arange(<start>, end, <increment>)`
  - start and increment are optional
  - the array of values will end *before* the end point
  - `np.arange()` is like a more flexible form of the built-in `range()` function

[75]:
```
xvals = np.arange(10) # Generates integers from 0 to < 10
print(xvals)
```

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

[76]:
```
xvals = np.arange(2,10) # Generates integers from 2 to < 10
print(xvals)
```

```
[2 3 4 5 6 7 8 9]
```

[77]:
```
xvals = np.arange(2,10,2) # Generates even integers from 2 to < 10
print(xvals)
```

```
[2 4 6 8]
```

[78]:
```
xvals = np.arange(1.0,5.0,0.5) # Generates floats every 0.5 from 1.0 to 4.5
for i in range(len(xvals)):  # len gives length of array xvals
    print("i: %d xvals[%d]: %4.2f" % (i,i,xvals[i]))
```

```
i: 0 xvals[0]: 1.00
i: 1 xvals[1]: 1.50
```

```
i: 2 xvals[2]: 2.00
i: 3 xvals[3]: 2.50
i: 4 xvals[4]: 3.00
i: 5 xvals[5]: 3.50
i: 6 xvals[6]: 4.00
i: 7 xvals[7]: 4.50
```

- For `np.linspace(<start>, end, <number_steps>)`
  - the array of values will end *exactly* at the end point
  - the array will have exactly number_steps entries
  - This is very useful for graph plotting.
  - For example:

[79]:
```
xvals = np.linspace(1.0,5.0,4)
for i in range(len(xvals)):  # len gives length of array xvals
    print("i: %d xvals[%d]: %4.2f" % (i,i,xvals[i]))
```

```
i: 0 xvals[0]: 1.00
i: 1 xvals[1]: 2.33
i: 2 xvals[2]: 3.67
i: 3 xvals[3]: 5.00
```

6.1.5 Vectorization: a really useful (and potentially confusing) feature of NumPy arrays

- Let's say we have a 1-dimensional array of x values, and we want to compute a function f(x) for each of them. We could:
  1. Write a loop and evaluate each value of f(x) in turn (slow)
  2. Take advantage of the NumPy vectorisation to achieve it in one line (fast)

[80]:
```
import numpy as np
x = np.linspace(0,1,11)
y = x**2 + x + 1  # This line is an example of vectorisation
                  # y is created as a numpy array and all its
                  # entries are assigned in one go.

for i in range(len(x)):
    print(x[i],y[i])
```

```
0.0 1.0
0.1 1.11
0.2 1.24
0.30000000000000004 1.3900000000000001
0.4 1.56
0.5 1.75
0.6000000000000001 1.9600000000000002
0.7000000000000001 2.1900000000000004
0.8 2.4400000000000004
0.9 2.71
1.0 3.0
```

- Vectorisation **only** works with NumPy arrays, not with Python lists.
- Vectorisation **only** works with the NumPy mathematical functions, not the `math` ones i.e. this works:

[81]:
```
x = np.linspace(0,np.pi/2,11)
y = np.sin(x)  # Must be the NumPy sin() function
```

```
for i in range(len(x)):
    print("%6.4f %6.4f" % (x[i],y[i]))
```

```
0.0000 0.0000
0.1571 0.1564
0.3142 0.3090
0.4712 0.4540
0.6283 0.5878
0.7854 0.7071
0.9425 0.8090
1.0996 0.8910
1.2566 0.9511
1.4137 0.9877
1.5708 1.0000
```

- But, **this does not work:**

[82]:
```
x = np.linspace(0,np.pi/2,11)
y = math.sin(x)  # Must be the NumPy sin() function

for i in range(len(x)):
    print("%6.4f %6.4f" % (x[i],y[i]))
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-82-535142dcb689> in <module>
      1 x = np.linspace(0,np.pi/2,11)
----> 2 y = math.sin(x)  # Must be the NumPy sin() function
      3
      4 for i in range(len(x)):
      5     print("%6.4f %6.4f" % (x[i],y[i]))


TypeError: only size-1 arrays can be converted to Python scalars
```

6.2 Reading and writing to files

See Kinder & Nelson ch. 4.1 & 4.2; Hill ch. 2.6.

The operation of reading or writing to a file consists of 3 stages:

1. Open the file: this requires specifying the filename and associating this with a special file-type variable;
2. Perform the reads or writes, using the appropriate file-type variable to specify the location;
3. Close the file: this tells the operating system we are done, and ensures any final writes are completed.

Files can be either text-based or raw binary. While binary files are more compact, text files have the advantage that they can be read by both computers and humans and are, therefore, the recommended type to use.

6.2.1 Opening and closing a file

- To `open` a file i.e. prepare it for reading and writing, use the `open()` function.

- To `close` the file again i.e. finish any reading and writing and leave the file ready for use in another application, use the `close()` method of the file object.
- Different modes (reading, writing etc) are available as follows. The options 'r', 'w' and 'a' will be the most useful:

File mode	Operation
r	open an existing text file to read
w	open a text file to write. If the file exists already it contents will be overwritten. If the file does not already exist it will be created.
a	Appends to a text file. Opens an existing file for writing and appends the newly written information to the end of the file.
r+	open a text file for reading and writing

- A further set of options, 'rb', 'wb', 'ab' and 'rb+' are available for reading and writing binary files. These are best avoided as debugging binary read/write operations can be very tricky.

[83]:
```
MyFileObject = open('anyfile.txt','w') # creates an empty file called `anyfile.txt'
                                       # and prepares it for writing, 'w'.
MyFileObject.close()  # closes the file again
```

6.2.2 Writing to files

- To write to a file, use a modified version of the `print()` function.
- Note that `print()` has two optional arguments, one to specify the string separator, and the other the file object.
- The following code generates a file with two columns separated by the string `', '`.
- To separate the columns with tabs, use the separation string `'\t'`.

[84]:
```
MyFileObject = open('anyfile.txt','w')

x = np.linspace(0,np.pi/2,101)
y = np.sin(x)

for i in range(len(x)):
    print("%6.4f" % x[i], "%6.4f" % y[i], sep=', ', file=MyFileObject)

MyFileObject.close()  # closes the file again
```

- Alternatively, could use: print("%6.4f, %6.4f" % (x[i],y[i]), file=MyFileObject)

6.2.3 Reading data from a file

- For a given file object, the `read(n)` method reads n bytes. This is not especially useful for text files, where a given number could be represented by a varying number of bytes. If n is omitted, the entire file is read in a single operation.
- The `readline()` method is more useful for reading text files.

[85]:
```
MyFileObject = open('anyfile.txt','r')

for i in range(5):
    print(i, MyFileObject.readline(), end='')
```

```
MyFileObject.close()
```

```
0 0.0000, 0.0000
1 0.0157, 0.0157
2 0.0314, 0.0314
3 0.0471, 0.0471
4 0.0628, 0.0628
```

- Each use of `MyFileObject.readline()` reads one line from the file.
- The `end=''` argument is needed to prevent double spacing of the output (it gets rid of an extra linefeed).
- Alternatively, file objects are iterable, so `MyFileObject` can be used to control a loop, reading the file, one line at a time.

[86]:
```
MyFileObject = open('anyfile.txt','r')

xvals, yvals = [], [] # Create two lists to hold the data

for line in MyFileObject:
    fields = line.split(',')       # split each line at the commas
    xvals.append(float(fields[0])) # append the new x value to the list
    yvals.append(float(fields[1])) # append the new y value to the list

print(yvals)

MyFileObject.close()
```

```
[0.0, 0.0157, 0.0314, 0.0471, 0.0628, 0.0785, 0.0941, 0.1097, 0.1253, 0.1409,
0.1564, 0.1719, 0.1874, 0.2028, 0.2181, 0.2334, 0.2487, 0.2639, 0.279, 0.294,
0.309, 0.3239, 0.3387, 0.3535, 0.3681, 0.3827, 0.3971, 0.4115, 0.4258, 0.4399,
0.454, 0.4679, 0.4818, 0.4955, 0.509, 0.5225, 0.5358, 0.549, 0.5621, 0.575,
0.5878, 0.6004, 0.6129, 0.6252, 0.6374, 0.6494, 0.6613, 0.673, 0.6845, 0.6959,
0.7071, 0.7181, 0.729, 0.7396, 0.7501, 0.7604, 0.7705, 0.7804, 0.7902, 0.7997,
0.809, 0.8181, 0.8271, 0.8358, 0.8443, 0.8526, 0.8607, 0.8686, 0.8763, 0.8838,
0.891, 0.898, 0.9048, 0.9114, 0.9178, 0.9239, 0.9298, 0.9354, 0.9409, 0.9461,
0.9511, 0.9558, 0.9603, 0.9646, 0.9686, 0.9724, 0.9759, 0.9792, 0.9823, 0.9851,
0.9877, 0.99, 0.9921, 0.994, 0.9956, 0.9969, 0.998, 0.9989, 0.9995, 0.9999, 1.0]
```

6.2.4 NumPy functions for file input/output

- You can also use the NumPy functions `np.loadtxt()` and `np.savetxt()` for reading and writing text files of data.
- Information on these may be found in the recommended textbooks, and on the NumPy website: http://www.numpy.org.

6.3 Plotting graphs

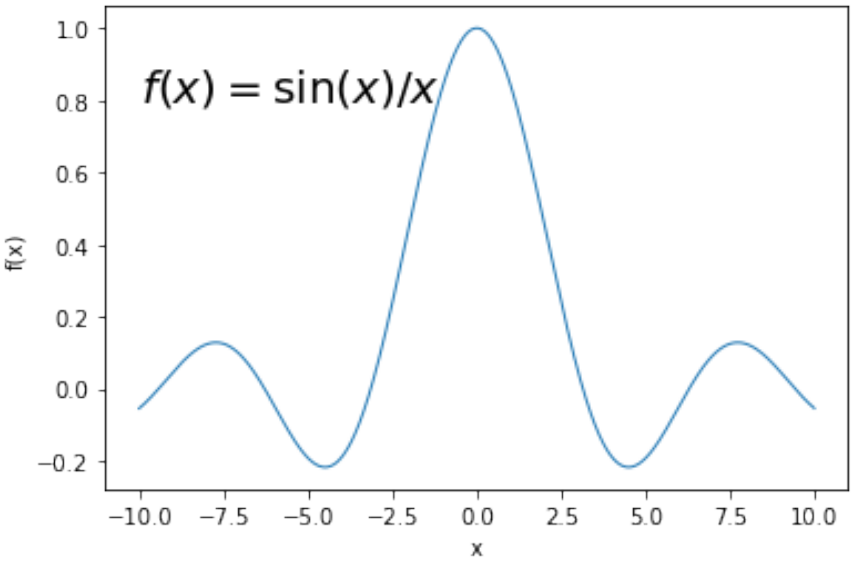
See Kinder & Nelson ch. 4.3; Hill ch. 7.

6.3.1 Line plots using Matplotlib and pylab

- Matplotlib is a sophisticated graphing module, with capabilities for producing publication quality graphs, for journal articles, books etc.

- pylab is a simple plotting interface to Matplotlib. We will focus on use of pylab, through the following examples. Much more is possible; see textbooks and the Matplotlib website (http://www.matplotlib.org) for more information.
- Here is a simple line plot:

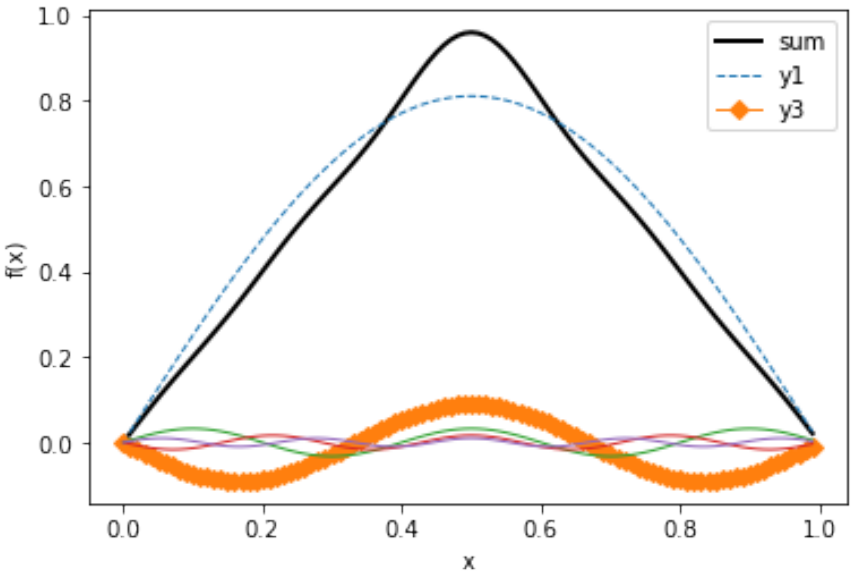
[87]:
```
#
# A plot of sin(x)/x
#
# following line needed to get graphs to appear in Jupyter worksheet
#
%matplotlib inline
import pylab
import numpy as np
#
# Generate the x and y values to plot
#
x = np.arange(-10, 10, 0.01)
y = np.sin(x)/x

pylab.plot(x, y, linewidth=1)
pylab.text(-5.5, 0.8,
     r"$f(x) = \sin(x)/x$", horizontalalignment='center',
     fontsize=20) # Note the use of a LaTeX label
pylab.xlabel("x")
pylab.ylabel("f(x)")
pylab.show() # this command not always needed
```



The following puts several lines together on one plot, using various plot styles:

[88]:
```
#
# several graphs together on the same plot
#
%matplotlib inline
```

```
import pylab
import numpy as np
#
# Generate the x and y values to plot using Vectorisation
#
A = 8/(np.pi**2)
x = np.arange(0, 1, 0.01)
y1 = np.sin(np.pi*x)*A
y3 = -np.sin(3*np.pi*x)/9*A
y5 = np.sin(5*np.pi*x)/25*A
y7 = -np.sin(7*np.pi*x)/49*A
y9 = np.sin(9*np.pi*x)/81*A
y = y1+y3+y5+y7+y9

pylab.plot(x, y, linewidth=2, color='k', label='sum')
pylab.plot(x, y1, linewidth=1, linestyle='--', label='y1')
pylab.plot(x, y3, linewidth=1, marker='D', label='y3')
pylab.plot(x, y5, linewidth=1)
pylab.plot(x, y7, linewidth=1)
pylab.plot(x, y9, linewidth=1)

pylab.xlabel("x")
pylab.ylabel("f(x)")
pylab.legend()
pylab.show() # this command not always needed
```

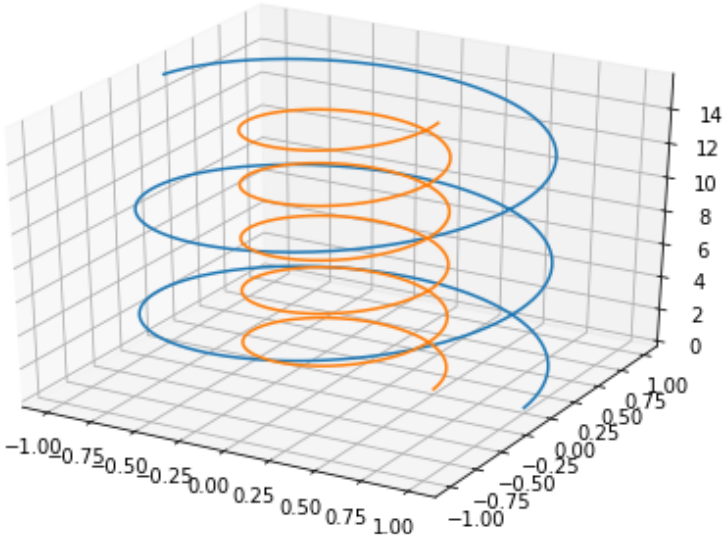


6.3.2 3-dimensional plots

- Various types of 3-d plots are possible, but you first need to import the Axes3D module from Matplotlib to create the appropriate set of axes:

[89]:
```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
#
# Create a figure with 3D axes
#
MyFig = plt.figure()
ax = Axes3D(MyFig)
#
# Generate a plot, using the plot method of the axes
#
theta = np.linspace(0, 5*np.pi, 501)
ax.plot (np.cos(theta), np.sin(theta), theta)
ax.plot (0.5*np.cos(2*theta), 0.5*np.sin(2*theta), theta)
```

[89]:
```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f83b894ad10>]
```



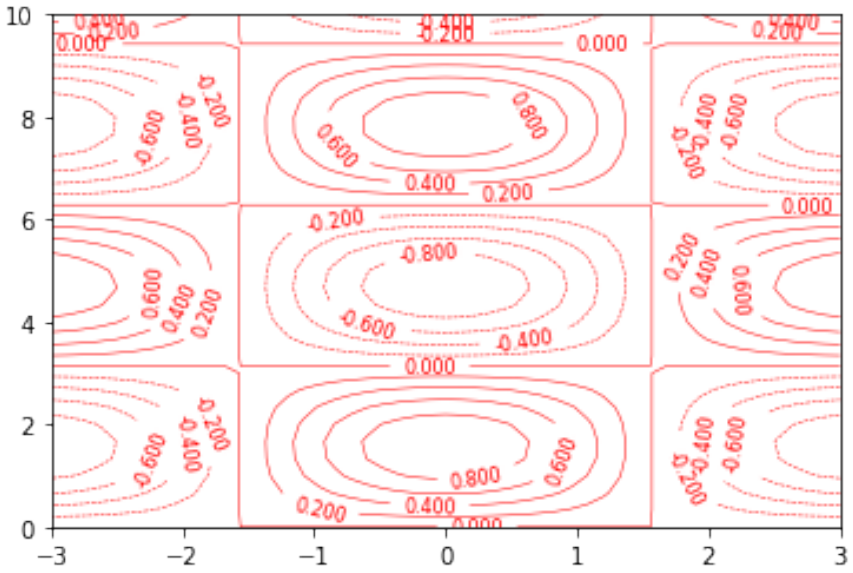
If you use Axes3D from within Spyder, the graph should appear in a separate window and you will be able to change the view by dragging with the mouse.

6.3.3 Contour plots

- First you need to create a 2-d mesh of points
- The contour plot is then straightforward:

[90]:
```
%matplotlib inline
#
# Adapted from Kinder and Nelson, 2015
#
import numpy as np
```

```
import matplotlib.pyplot as plt
#
# Create a grid of x and y values,
# using the linspace and meshgrid functions
#
x = np.linspace(-3, 3, 51)
y = np.linspace(0, 10, 31)
X, Y = np.meshgrid(x, y)
#
# Use vectorisation to generate function values
#
MyFunc = np.cos(X) * np.sin(Y)
#
# Plot and label the contours
#
plt.figure()
MyContours = plt.contour(X, Y, MyFunc, 10, linewidths=0.5, colors='r')
plt.clabel(MyContours,fontsize=8)
plt.show()
```

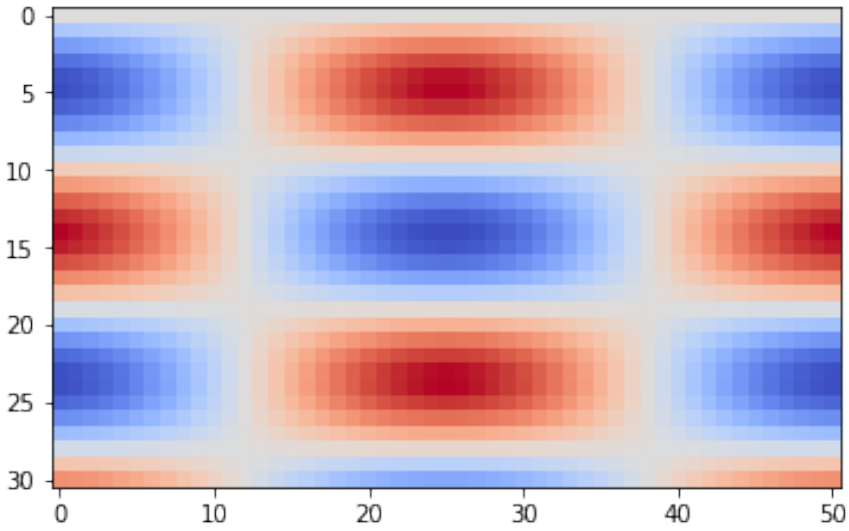


- Sometimes it is more appropriate to view the data as an intensity plot. This can be achieved as follows:

[91]:
```
%matplotlib inline
#
# Adapted from Kinder and Nelson, 2015
#
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
#
# Create a grid of x and y values,
# using the linspace and meshgrid functions
```

```
#
x = np.linspace(-3, 3, 51)
y = np.linspace(0, 10, 31)
X, Y = np.meshgrid(x, y)
#
# Use vectorisation to generate function values
#
MyFunc = np.cos(X) * np.sin(Y)
#
# Plot the function as an intensity map
#
plt.imshow(MyFunc, cmap=cm.coolwarm)
```

[91]: <matplotlib.image.AxesImage at 0x7f8398adf510>

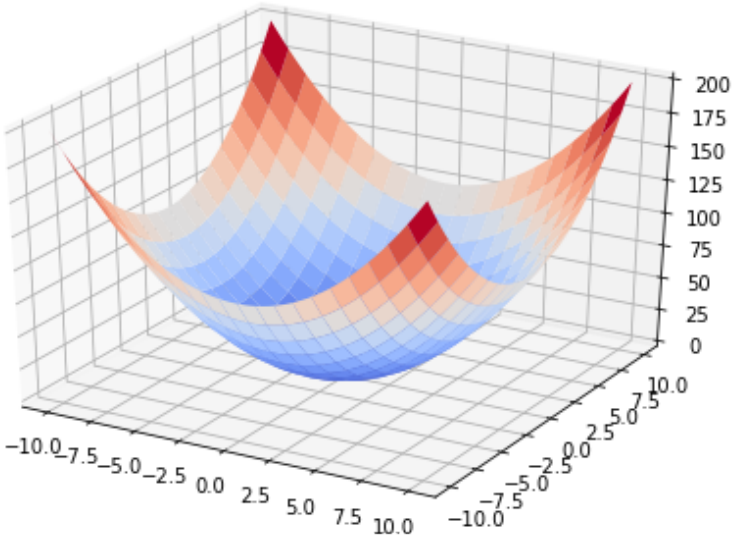


6.3.4 Surface Plots

- As previously, the Axes3D module is needed

[92]:
```
%matplotlib inline
#
# Adapted from Kinder and Nelson, 2015
#
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
#
# Create a grid of x and y values,
# using the linspace and meshgrid functions
#
points = np.linspace(-10, 10, 101)
X, Y = np.meshgrid(points, points)
```

```
Z = X**2 + Y**2
#
# create and display surface plot
#
ax = Axes3D(plt.figure())
ax.plot_surface(X, Y, Z, rstride=5, cstride=5, cmap=cm.coolwarm, linewidth=0.1)
#
plt.show()
```



Appendix A (Optional section - read at your own risk) Arduino programming from Python

This section was last updated in May 2018 - some of the version numbers for pyFirmata and pyMata may be out of date.

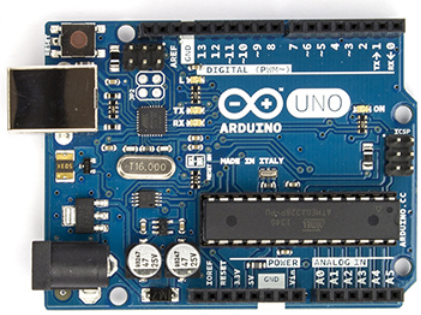
A.1 Introduction

If you have never heard of the Arduino system for interfacing computers to experiments and other "real-world" devices, then you probably should read no further. However, you may have come across the Arduino as a hobbyist, or you may be planning a project (undergraduate or otherwise) that involves using an Arduino to control some piece of apparatus. In that case, the following notes are intended to give a flavour of what is possible, and to point you in the right direction to get started. There are many different approaches to using the Arduino from Python. The following is just one possibility that satisfies the most basic requirements.

A.2 The Arduino programming model

The Arduino is a self-contained microcontroller board, that "talks" to a host computer via some sort of interface. Typically, the interface will be a USB cable, but there are other possibilities including ethernet, Bluetooth and Wifi. The Arduino consists of a very basic microprocessor, some memory and a number of input/output (I/O) pins, which are used to connect to the real world. The Arduino can be used to send and receive signals via these pins.

This is the typical appearance of an entry level Arduino UNO:



The normal mode of operation for the Arduino is to upload a program (known as a Sketch) to the device, and allow it to run in a loop in an autonomous fashion. The Sketch is written in the C programming language, and is uploaded to the Arduino using the Arduino IDE. Interaction with the Arduino will only be possible if the Sketch includes the provision to send and receive data over the interface; a separate program running on the computer is required to monitor the interface, either logging data or sending control signals as appropriate. It is this separate control program which may be written in Python.

Unless you have some very specific requirements for your interface, it is simplest to use a pre-existing standalone Arduino Sketch, and focus your efforts on the Python control program. We give some basic examples of this below.

A.3 Preparing the Arduino board

These instructions relate to the Arduino UNO board; other boards will follow a similar pattern.

First it is necessary to install the Arduino IDE, available from https://www.arduino.cc/en/Main/Software . Once installed, the Arduino board should be connected to a USB port of the computer, and the Arduino IDE started. From the IDE menu, you should select:

- From `"Tools --> Board..."` choose the appropriate board type.
- From `"Tools --> Port..."` choose the port your Arduino is connected to. This will vary depending on the computer type, and will typically be of the form:
  - `"/dev/cu.usbmodemxxx"` for a Mac
  - `"/dev/ttyACMxx"` for Linux
  - `"COMx"` for a PC

Finally, the standalone Sketch is required. We will use the Firmata protocol for communicating with the Arduino; an appropriate Sketch for this is available within the IDE at `"File --> Examples --> Firmata --> StandardFirmata"`. Load this Sketch, compile it using `"Sketch --> Verify/Compile"` and upload it to the Arduino using "'Sketch –> Upload".

A.4 pySerial and pyFirmata

Communication between the host computer and the Arduino will be handled using the Firmata protocol. This is available as a module, pyFirmata, which is available within Python. pyFirmata is, in turn, dependent on a serial communications module, known as pySerial. Neither of these modules are available within the standard Anaconda-based installation, so a little effort is required to install them.

In fact, pySerial is known to Anaconda, and a single command is sufficient to install it i.e.: * conda install pyserial

at the command prompt. Once pySerial is installed, verify that Python knows about it by typing the following command at the IPython prompt:

[1]:
```
import serial
```

There should be no error messages. pyFirmata is available from https://pypi.python.org/pypi/pyFirmata/1.0.3 . Download the package and follow the installation instructions. Once pyFirmata is installed, verify that Python knows about it by typing the following command at the IPython prompt:

[2]:
```
import pyfirmata
```

If there are no error messages, your Python installation is ready to address the Arduino.

A.5 Simple Arduino test

At the start of each Python program addressing the Arduino, you will need to setup the port and board type as follows:

[3]:
```
port = '/dev/cu.usbmodem1411'  # substitute your port name here
board = pyfirmata.Arduino(port)  # this identifies the Arduino attached to your port
pin = 13  # choose pin number 13
```

The Arduino UNO has a test LED permanently attached to output pin 13 of the board. The following lines can be used to toggle the LED on:

[7]:
```
board.digital[pin].write(1)  # setting pin 13 to 'HIGH', LED is on
```

and off:

[9]:
```
board.digital[pin].write(0)  # setting pin 13 to 'LOW', LED is off
```

The state of pin 13 may be read at any time using the command:

[10]:
```
board.digital[pin].read()  # read the digital value of pin 13
```

[10]: 0

A.6 A program to control an LED

The following program uses pulse-width modulation (PWM) to control the intensity of an LED attached to one of the digital outputs of the Arduino. To see the effect of this program you will need to connect an LED between pin 11 of the Arduino and Ground, via a small resistor (say 220Ω). The program causes the LED to pulse smoothly five times.

[16]:
```
from pyfirmata import Arduino, PWM
from time import sleep
from math import sin, pi

port = '/dev/cu.usbmodem1411' # Change this for your particular installation
board = Arduino(port)

sleep(1)  # Allow time for the Arduino to initialise

pin = 11
board.digital[pin].mode = PWM  # Setup pin 11 to use PWM

period = 50  # Sets the duration of each pulse
```

```
number = 5  # Sets the number of pulses
for i in range(0, number*period+1):
    board.digital[pin].write(sin(i*pi/period)**2) # Sinusoidal intensity variation
    sleep(0.05)
    if int(i/period)*period == i:
        print(i*100/(number*period),"% done") # Give progress statement

board.digital[pin].write(0)  # Finally set output to zero
board.exit()  # Tidy closure of Arduino
```

```
0.0 % done
20.0 % done
40.0 % done
60.0 % done
80.0 % done
100.0 % done
```

A.7 Logging data to file and screen

This program illustrates reading a voltage from one of the analog input pins of the Arduino. This might be, for example, the output from some sort of sensor.

[18]:
```
from pyfirmata import Arduino, util
from time import sleep

port = '/dev/cu.usbmodem1411'  # Change this for your particular installation
board = Arduino(port)

sleep(1)  # Allow time for the Arduino to initialise

it = util.Iterator(board) # These two lines prevent the serial buffer from overflowing
it.start()

sleep(1)  # Allow time for the Arduino to catch up

pin = 0  # Choose one of the analog pins as input
samples = 10 # Set number of samples to take

board.analog[pin].enable_reporting()  # Enable reading from the chosen pin

for i in range(0, samples):
    sleep(1)
    print("Sample: ",i,board.analog[pin].read()) # Read a value and print it

board.analog[pin].disable_reporting()  # Disable reading from the chosen pin
board.exit()  # Tidy closure of Arduino
```

```
Sample:  0 0.7107
Sample:  1 0.6647
Sample:  2 0.5904
Sample:  3 0.4936
Sample:  4 0.3812
Sample:  5 0.2776
Sample:  6 0.39
```

```
Sample:  7 0.6628
Sample:  8 0.7752
Sample:  9 0.7771
```

A.8 Final points

- If the above programs fail to execute from within the Jupyter notebook, the simplest solution is often to copy the code into a separate script file and execute it from the command line.
- The Firmata protocol is only one of many that perform similar operations on the Arduino. More are available from the Arduino IDE menu.
- An alternative python interface to work with the Firmata protocol is provided by the PyMata module, available from https://pypi.python.org/pypi/PyMata/2.17 .