Python-Essentials

October 26, 2016

1 Python Essentials

The essentials of Python coding (for the class test) with *absolutely no explanation*.

1.1 Useful methods

Use `<TAB>` to complete variable names or see available functions. Use the `help` command or spyder's Object Inspector to see how to use specific functions.

1.2 Mathematical operators

Standard operations:

```
In [1]: print(1+1)
        print(2-1)
        print(2*3)
        print(1/2)
```

```
2
1
6
0.5
```

Powers:

```
In [2]: print(2**2)
        print(2**3)
        print(2**(1/2))
```

```
4
8
1.4142135623730951
```

1.3 Variables

Start with a letter; don't use special characters or spaces:

```
In [3]: x = 3
        y = 15
        z = x+y
        my_variable_name = x * y * z
        print(my_variable_name)
```

```
810
```

1.4 Strings

Use the same type of quotes to open and close.

```
In [4]: name = 'Ian Hawke'
        apostrophe = "can't use single quotes"
        print(name, apostrophe)
```

```
Ian Hawke can't use single quotes
```

Use triple quotes for a string spanning more than one line.

```
In [5]: multiline_string = """Hello
        world.
        This
        is
        just
        silly.
        """
        print(multiline_string)
```

```
Hello
world.
This
is
just
silly.
```

To print variables mixed in with strings either separate by commas:

```
In [6]: print("Hello", name, ", how are you?")
```

```
Hello Ian Hawke , how are you?
```

Alternatively (and more useful in general), use the `format` command (note the behaviour of `{}` in the string):

```
In [7]: print("Hello {}, how are you?".format(name))
```

```
Hello Ian Hawke, how are you?
```

1.5 Lists and slicing

Define a list with square brackets `[]`:

```
In [8]: list_variable = [1,2,3,4,5]
```

Get individual entries using square brackets after the name. Note that this **starts from `0`**:

```
In [9]: print(list_variable[0])
        print(list_variable[2])
```

```
1
3
```

Get the length of the list using `len`:

```
In [10]: print(len(list_variable))
```

```
5
```

Get multiple entries using slicing, `start:end:step`. This includes the start, does *not* include the end. `step` defaults to `1`, others to obvious:

```
In [11]: print(list_variable[0:2])
         print(list_variable[:2])
         print(list_variable[3:5])
         print(list_variable[3:])
         print(list_variable[0:5:2])
         print(list_variable[1:5:2])
```

```
[1, 2]
[1, 2]
[4, 5]
[4, 5]
[1, 3, 5]
[2, 4]
```

Use negative numbers to indicate entries from the end:

```
In [12]: print(list_variable[-1])
         print(list_variable[-2])
         print(list_variable[-1:-3:-1])
```

```
5
4
[5, 4]
```

Set entries into the list in the standard way of setting a variable, but you can do multiple entries in one go:

```
In [13]: list_variable[0] = 10
         print(list_variable)
         list_variable[0:2] = [9, 11]
         print(list_variable)
```

```
[10, 2, 3, 4, 5]
[9, 11, 3, 4, 5]
```

1.6 Loops

Do something multiple times. For each entry in a list, do something:

```
In [14]: list_variable = [1,2,3,4,5]
         for number in list_variable:
             print(number)
```

```
1
2
3
4
5
```

Each step in the loop sets the variable `number` to the next entry in the list, and executes *all code lines that are indented immediately after the colon*.

```
In [15]: for number in list_variable:
             number2 = 2*number
             number3 = number2-1
             print("number3", number3)
         print("number2", number2)
```

```
number3 1
number3 3
number3 5
number3 7
number3 9
number2 10
```

Loops can be *nested*:

```
In [16]: list1 = [1, 2, 3]
         list2 = [4, 5, 6]
         for number1 in list1:
             print("In loop 1", number1)
             for number2 in list2:
                 print("In loop 2", number2)
             print("Back in loop1", number1)
         print("Done")
```

```
In loop 1 1
In loop 2 4
In loop 2 5
In loop 2 6
Back in loop1 1
In loop 1 2
In loop 2 4
In loop 2 5
In loop 2 6
Back in loop1 2
In loop 1 3
In loop 2 4
In loop 2 5
In loop 2 6
Back in loop1 3
Done
```

We often loop over consecutive integers. Python provides the `range` function for this:

```
In [17]: for number in range(4):
             print(number)
```

```
0
1
2
3
```

If you have a vector that you want to loop over whilst keeping track of the index, use the `enumerate` function:

```
In [18]: list_enum = [3.4, 7.9, -6.4, 0.1]
         for i, number in enumerate(list_enum):
             print("Index is", i, "number is", number)
```

```
Index is 0 number is 3.4
Index is 1 number is 7.9
```

```
Index is 2 number is -6.4
Index is 3 number is 0.1
```

Finally, if you don't know how often you want to do the loop, you can use a `while` loop:

```
In [19]: number = 15
         while number > 0:
             number = number / 2 - 1
             print("number is now", number)
```

```
number is now 6.5
number is now 2.25
number is now 0.125
number is now -0.9375
```

1.7 Control flow

To make the code execute certain statements *only if* certain conditions are met, the `if/elif/else` commands are needed:

```
In [20]: if 1<0:
             print("1<0")
         elif 2>1:
             print("2>1")
         else:
             print("Neither 1<0 nor 2>1")
```

```
2>1
```

```
In [21]: if 1>0:
             print("1>0")
         elif 2>1:
             print("2>1")
         else:
             print("Neither 1>0 nor 2>1")
```

```
1>0
```

```
In [22]: if 1<0:
             print("1<0")
         elif 2<1:
             print("2<1")
         else:
             print("Neither 1<0 nor 2<1")
```

```
Neither 1<0 nor 2<1
```

As with loops, the code to be executed is indented after the colon.

1.8 Functions

Saves blocks of code that can be re-used:

```
In [23]: def divide(a, b):
             """
             Divide a by b.
             """
             result = a/b
             return result
```

The name of the function appears immediately after the `def` keyword. Here it is `divide`.
The arguments are the variables in the brackets. Here they are `a` and `b`.
The code executed is everything that is indented after the colon (as with the loops above).
The value you get from calling the function is that after the `return`: this could be a number, or a list, or something more complex.

```
In [24]: print(divide(1,2))
```

```
0.5
```

The string (called the *docstring*) appears when you ask for help on the function:

```
In [25]: help(divide)
```

```
Help on function divide in module __main__:

divide(a, b)
    Divide a by b.
```

1.9 Packages

To make commands from a package available you have to `import` them. Commands or variables are then referred to by the name with which you imported followed by a dot.

```
In [26]: import numpy
         print(numpy.pi)
```

```
3.141592653589793
```

```
In [27]: import scipy.constants
         print(scipy.constants.m_e)
```

```
9.10938356e-31
```

```
In [28]: from scipy import constants
         print(constants.m_e)
```

```
9.10938356e-31
```

If you create a file and save it with the name `file1.py`, then you can `import` the content of that file from another file in the *same directory* using the command `import file1`.

1.10 Linear algebra and numpy

We use `numpy` for most linear algebra.

Create a vector, matrix, or array.

```
In [29]: x = numpy.array([1.0, 2.0])
         b = numpy.array([3.0, 4.0])
         A = numpy.array([ [1.0, 2.0], [3.0, 4.0] ])
         C = numpy.array([ [1.0, 2.0, 3.0], [4.0, 5.0, 6.0] ])
         print(x)
         print(A)
         print(C)
```

```
[ 1.  2.]
[[ 1.  2.]
 [ 3.  4.]]
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
```

Get their lengths and shapes.

```
In [30]: print(len(x))
         print(x.shape)
         print(len(A))
         print(A.shape)
```

```
2
(2,)
2
(2, 2)
```

Create arrays creating just zeros or ones, either of specific sizes (eg `zeros`), or of the same size as an existing array (eg `zeros_like`):

```
In [31]: v = numpy.ones(3)
         w = numpy.zeros((4,4))
         print(v)
         print(w)
         D = numpy.ones_like(A)
         print(D)
```

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

We'll often need to create *linearly spaced* arrays over an interval, like $[0, 1]$:

```
In [32]: z = numpy.linspace(0,1,6)
         print(z)
```

```
[ 0.   0.2  0.4  0.6  0.8  1. ]
```

Mathematical operations apply to all components at once:

```
In [33]: print(x+1)
         print(2*A)
         print(C**2)
```

```
[ 2.  3.]
[[ 2.  4.]
 [ 6.  8.]]
[[  1.   4.   9.]
 [ 16.  25.  36.]]
```

Specific `numpy` functions apply to all components at once:

```
In [34]: print(numpy.sin(x))
```

```
[ 0.84147098  0.90929743]
```

Functions can be applied to entire arrays, for example to sum all entries:

```
In [35]: print(numpy.sum(x))
         print(numpy.sum(A))
```

```
3.0
10.0
```

Vector dot products and matrix multiplications use the `dot` function:

```
In [36]: print(numpy.dot(x,x))
         print(numpy.dot(A,x))
```

```
5.0
[  5.  11.]
```

More specific linear algebra functions can be found in the `linalg` subpackage.
Solve the linear system:

```
In [37]: print(numpy.linalg.solve(A, b))
```

```
[-2.   2.5]
```

Matrix determinant:

```
In [38]: print(numpy.linalg.det(A))
```

```
-2.0
```

Matrix eigenvalues and eigenvectors:

```
In [39]: print(numpy.linalg.eig(A))
```

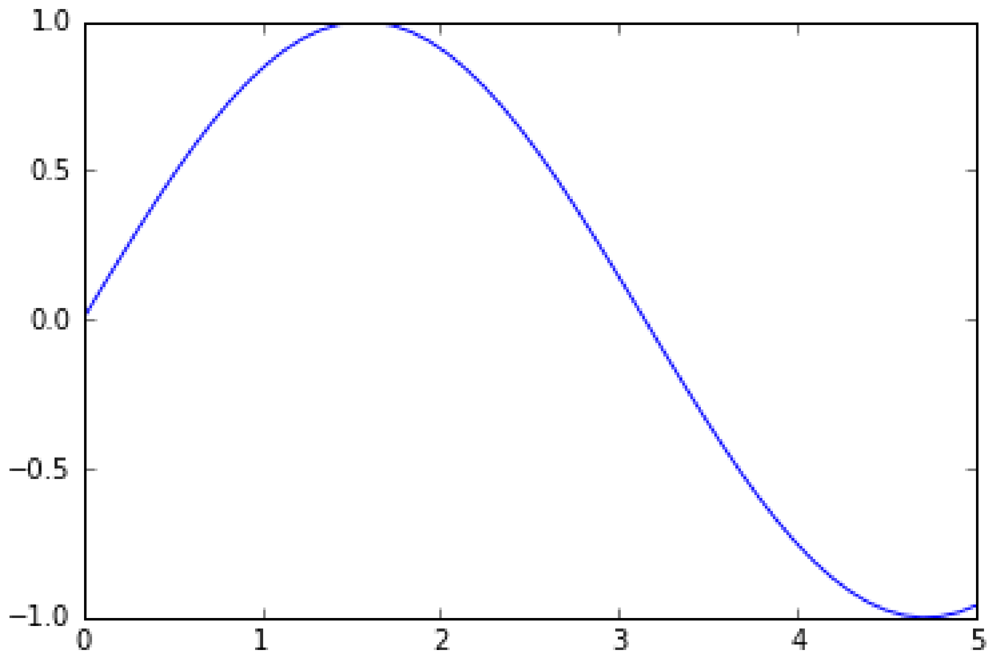
```
(array([-0.37228132,  5.37228132]), array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]]))
```

1.11 Plotting

The `matplotlib` library is usually used. The `pyplot` interface is the most straightforward.

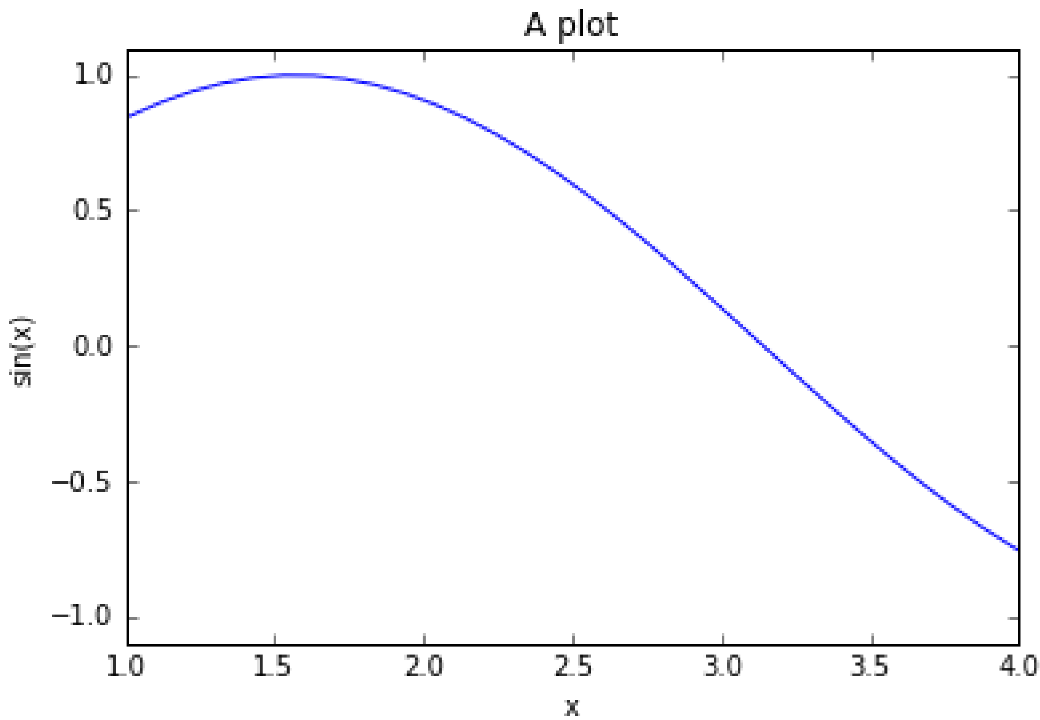
```
In [40]: from matplotlib import pyplot
```

```
In [42]: x = numpy.linspace(0,5)
         y = numpy.sin(x)
         pyplot.plot(x, y)
         pyplot.show()
```



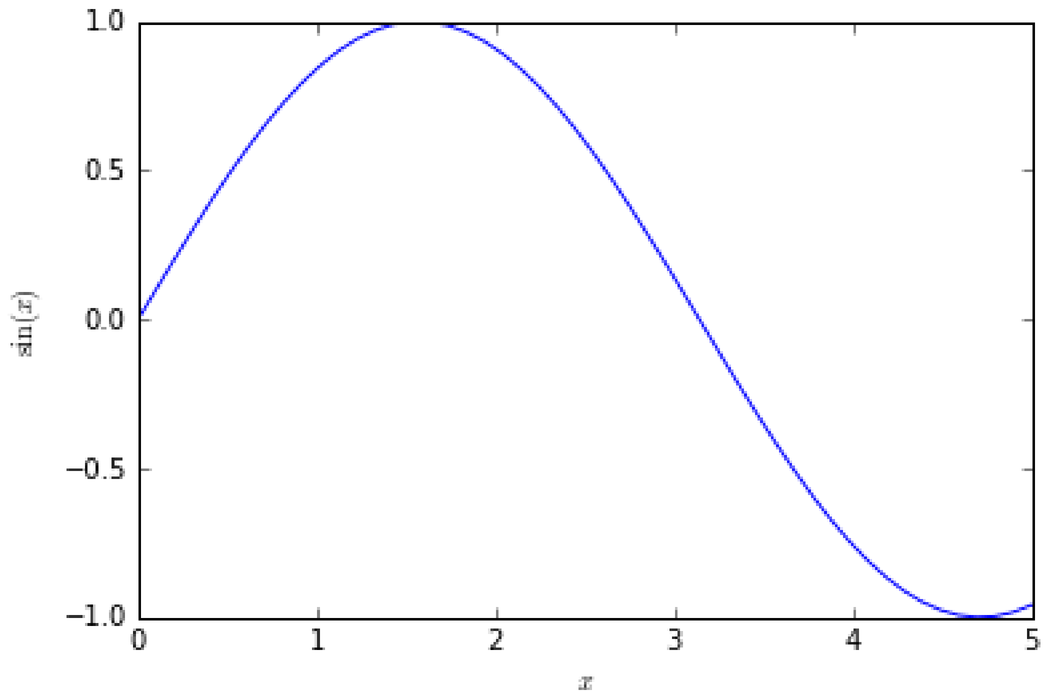
We can set the axis labels, the limits of the plots, the title:

```
In [43]: x = numpy.linspace(0,5)
         y = numpy.sin(x)
         pyplot.plot(x, y)
         pyplot.xlabel("x")
         pyplot.ylabel("sin(x)")
         pyplot.xlim(1, 4)
         pyplot.ylim(-1.1, 1.1)
         pyplot.title("A plot")
         pyplot.show()
```



For prettier mathematics we can use LaTeX by surrounding the text with `$$`. However, we should ensure the string is *raw* by putting `r` before it:

```
In [44]: x = numpy.linspace(0,5)
         y = numpy.sin(x)
         pyplot.plot(x, y)
         pyplot.xlabel(r"$x$")
         pyplot.ylabel(r"$\sin(x)$")
         pyplot.show()
```



For surface plots we need to construct matrices, or arrays, for the coordinates and the thing to plot.

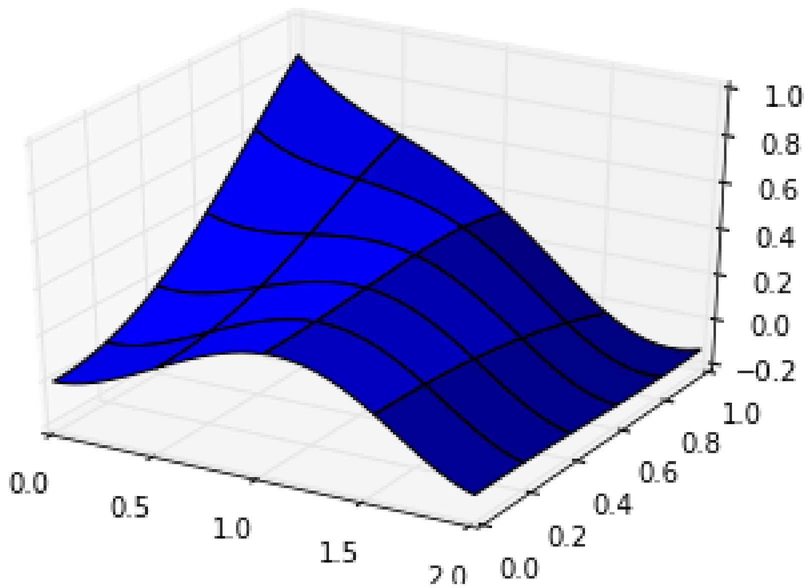
```
In [45]: x = numpy.linspace(0, 2, 40)
         y = numpy.linspace(0, 1, 50)
         X, Y = numpy.meshgrid(x, y)
         Z = numpy.exp(-X) * numpy.sin(X**2 + Y**2)
```

Then we need to construct a 3D *axis* on which we can plot:

```
In [46]: from mpl_toolkits.mplot3d.axes3d import Axes3D

In [47]: fig = pyplot.figure()
         axis = fig.add_subplot(1,1,1,projection='3d')

         axis.plot_surface(X, Y, Z)
         pyplot.show()
```

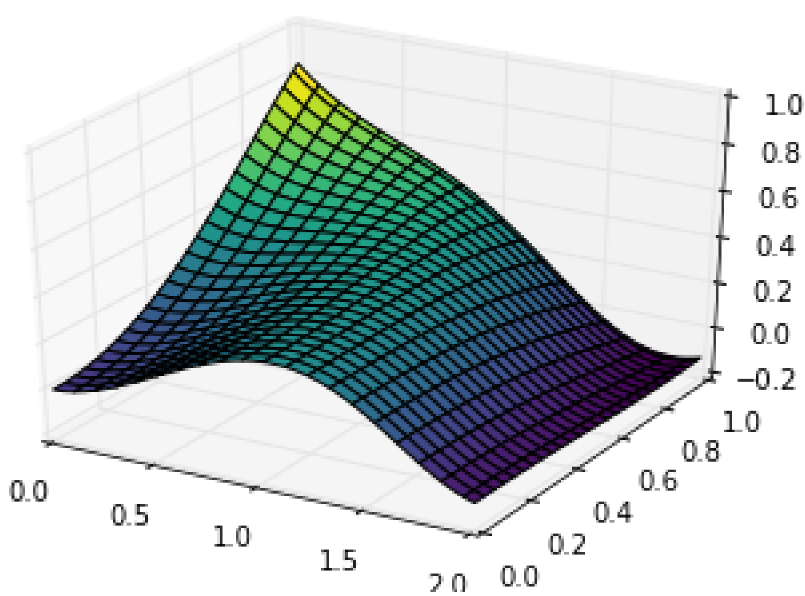


We can plot every row and column line by changing arguments, and change the colourmap:

```
In [48]: from matplotlib import cm

         fig = pyplot.figure()
         axis = fig.add_subplot(1,1,1,projection='3d')

         axis.plot_surface(X, Y, Z, rstride=2, cstride=2, cmap=cm.viridis)
         pyplot.show()
```



1.12 Black box solvers

The `scipy` package will solve many problems for you.

The most important parts for our purposes are the `optimize` package, for root finding:

```
In [49]: from scipy import optimize

         def f(x):
             return numpy.exp(-x) - x + 1

         root = optimize.brentq(f, 0, 2)
         print("Root is", root)
```

```
Root is 1.278464542761074
```

And the `integrate` package, for quadratures and solving differential equations:

```
In [50]: from scipy import integrate

         quadrature, error = integrate.quad(f, 0, 1)
         print("Integral is", quadrature)
```

```
Integral is 1.1321205588285577
```