Short Python and Numpy Introduction

1 Python Introduction

Python is an object oriented general purpose programming language. It is widely used for various kinds of applications and runs on most devices. With the interactive Python shell it is also possible to execute pieces of code on the fly.

1.1 Hello World

```
print('Hello World')
```

1.2 Script Files

Python commands can be simply typed into the command window where they are immediately executed. For longer programs you can write a script file which is executed sequentially. For that use the editor of your choice and save the file, e.g. `myScript.py`. To execute it there are two common ways. One is by typing

```
python myScript.py
```

from the shell. The other is by executing it from the interactive Python shell in Python 2.x:

```
execfile('myScript.py')
```

and in Python 3.x:

```
exec(open('myScript.py').read()).
```

1.3 Variables and Arrays

You can assign values to variables via the assignment operator '=':

```
a = 5
b = a + 3
```

This will assign the value 5 to `a` and value 8 to `b`. Variables are case sensitive, which means that `myVar` is a different variable then `myvar`. They can have almost any name with alphanumeric characters and the underscore but must start with a letter. Names which are Python keywords, such as `abs`, are allowed, but should be avoided. You can check if a keyword exists by typing into the Python console the name and pressing twice the tab-key (auto completion).

1.4 Arithmetic

In Python the arithmetic operators are the usual, e.g. `+, -, *, /`, etc. For exponents use two asterisks **:

```
3**2
```

Be careful when dividing integers since Python 2.x assumes you want an integer division. The operation

```
9/4
```

returns 2 instead of 2.25, while in Python 3.x it returns correctly 2.25. To get that result in Python 2.x you need to write at least one of the numbers as a real number:

```
9./4
```

1.5 Functions

Functions are subprograms which take arguments and can return values, e.g.

```
a = abs(-5)
```

Most common mathematical functions are already built into Python or are part of the Numpy library. There are ways to define own function (see section **??**).

Functions can take various kinds of input and perform different tasks. To check what kind of input arguments are required and to get the help of the function you need to type the function name followed by a question mark:

```
>>> abs?
Type:       builtin_function_or_method
String Form:<built-in function abs>
Namespace:  Python builtin
Docstring:
abs(number) -> number

Return the absolute value of the argument.
```

1.6 Output

Explicit output can be achieved via `print` which allows some formatting. It can take variables and text:

```
a = 5
b = -1.3
print('a = {0}, b = {1}'.format(a, b))
```

The output will be:

```
a = 5, b = -1.3
```

Everything between the single quotations will be printed on the screen. Here we have to use { and } to specify the output of a variable.

1.7 Conditionals

Conditional statements can be achieved with the usual `if`, `else if` and `else` statements:

```
if (a > 0):
    print('a > 0')
else if (a < 0):
    print('a < 0')
else:
    print('a = 0')
```

Depending on which of the conditions is true, the code after the first true conditional is executed. If none of them is true, execute what is coming after `else`. The conditional operators for (un)equality and larger/smaller are: `==`, `!=`, `>`, `<`, `>=`, `<=`. Note that the equality operator is `==` and not `=` which would assign a new value to the variable.

With logical operators one can combine multiple conditional, e.g.

```
if (a == 5) and (b < 3):
    c = 7
```

1.8 Indentations

Each block of code needs to be indented by 4 spaces. This makes the code more readable and eliminates the need to use an end statement.

```
i = 0
while (i < 3):
    print('i = {0}'.format(i))
    i = i + 1
print('loop terminated')
```

1.9 Loops

Loops are used for repeated execution of parts of the code:

```
for i in range(10):
    y = i**2
    print('y = {0}'.format(y))
```

Here `i` is the incremental variable and is increased by one every time the loop is repeated. `i` runs from 0 to 9 in this example.

1.10 Importing Libraries

To extend Python's functionalities there exist libraries for specific tasks. The Numpy library, for instance, contains functions for numerical computation. In order to use such libraries they need to be imported and should be assigned a name:

```
import numpy as np
```

Here we import the Numpy library and assign it the name `np`. All its functions can then be accessed using that name followed by a dot:

```
x = 3.1
y = np.sin(x)
```

2 User Defined Functions

For convenience we can define our own functions which perform any kind of operations, accept user defined arguments and return various results (or none):

```
def myFun():
    print ('This is myFun.')
```

Note that we need the parentheses and the colon. The four spaces are needed, since we have a separate code block. The function finishes before the first line which is not indented. To call this function simply type:

```
>>> myFun()
This is myFun.
```

Functions can accept arguments and return values:

```
def addition(a, b):
    y = a + b
    return y
```

Here the arguments are called `a` and `b` which are variables only seen by the function `addition` and have no reference to the outside. This function can be called:

```
>>> x = 5
>>> addition(x, 3)
8
```

3 Basic Numpy

Numpy offers are wealth of useful functionality for numerical computation. It contains basic mathematical functions like `sin` and `exp` and the useful Numpy arrays. For using this library it needs to be imported:

```
import numpy as np
```

Arrays are arguably the most used feature of Numpy. They make computation fast and easy. An array can be created by specifying its elements:

```
a = np.array([0, 1, 2, 2.1, 2.2, 2.3, 10])
```

or by using some of Numpy's functions like

```
b = np.zeros(10)
c = np.ones(100)
```

An array's element can be accessed using the square brackets:

```
>>> a[0]
0.0
```

Note that the first index is `0` instead of `1`. One can access array elements by counting backwards from the end:

```
>>> a[-1]
10.0
>>> a[-2]
2.3
```

Several elements can be accessed by using the colon `:`, e. g.:

```
x = a[3:6]
y = a[0:-1:2]
```

Here the first number specifies the start and the second the end index. The third and optional number specifies the increment.

Arrays of multiple dimensions can be easily created the same way:

```
u = np.array([[0, 1, 2], [10, 20, 30]])
```

Note the additional brackets. One can also use Numpy's functions:

```
v = np.zeros((3, 3))
```

Their elements are then accessed by e. g.

```
u[0,1]
```

A common pitfall when using Numpy arrays is when the user intends to copy one array by executing

```
a = np.array([0, 1, 2, 3])
b = a
```

Here `b` is not a new array. It is merely a reference to `a` and changind `b` also changes `a`:

```
>>> b[0] = 5
>>> a[0]
5
```

In order to properly copy Numpy arrays one needs to use the `copy` command:

```
b = a.copy()
```

Other useful Numpy functions:
`linspace`, `meshgrid`, `save`, `load`, `where`, `shape`, `ndim`, `size`, `dtype`, `arange`, `max`, `min`, `all`, `any`, `sum`

4 Basic Plotting

For plotting we use the library Matplotlib which is part of the Pylab library. Import it:

```
import pylab as plt
```

A simple plot can be achieved with the `plot` function:

```
x = np.array([0, 1, 2, 3, 4.5])
y = x**2
plt.plot(x, y)
```

For two-dimensional plots one can use `imshow`:

```
x = np.array([0, 1, 2, 3, 4.5])
y = np.array([0, 1, 2, 3, 4.5])
u = np.meshgrid(x,y)
z = u[0]**2 + u[1]**2
plt.imshow(z)
```

Note that `imshow` plots from top to down and uses inverted axis. To remedy this you would rather use:

```
plt.imshow(zip(*z), origin = 'lower')
```

Other useful Matplotlib functions:
`semilogy`, `loglog`, `figure`, `errorbar`, `xlabel`, `ylabel`, `legend`, `xlim`, `ylim`

5 Further Reading

https://docs.python.org/3.5/tutorial/introduction.html
http://wiki.scipy.org/Tentative_NumPy_Tutorial
http://matplotlib.org/gallery.html