Introduction to Linux and Python

Lecture notes

Contents

1 Linux

1.1 Introduction

Linux is a family of operating systems (OS) based on UNIX, an operating system which was first developed in the 1960s. Several operating systems have been developed following the Unix specification, for example "Mac OS X" and "Linux". Actually Linux is considered Unix-like, because even if it follows the Unix standard, it is not officially registered and then it cannot use the trademark UNIX.
The family Linux includes several *distributions*, for example Ubuntu, Red Hat, Debian, Fedora, Gentoo, Android (yes, the OS of your smartphone) and many others (more than 600!). These distributions can be very different, for example Ubuntu is considered more user-friendly for its easiness of installation, while Gentoo is considered more difficult to use, because it is necessary to recompile all the programs during the installation. However, at the price of a great difficulty of installation, you will get a faster and more efficient operating system. What is in common between all these distributions is the *kernel*, the Linux kernel. The kernel is the core of the operating system, the bridge between the software and the hardware.
It would be more correct to talk about GNU/Linux distributions, because the Linux kernel is only a part of the entire operating system. The other software of these operating systems (shells, compilers, libraries,...) have been developed inside the GNU project. GNU is the recursive acronym for "GNU's Not Unix!", chosen because the founder wanted to create an OS based on UNIX, but free and open to the developers.

1.2 The terminal

The UNIX operating systems are born without a graphical user interface (GUI) and the only way to interact with the computer was to use a command line interface (CLI). The program that interprets the input commands (keyboard keys) and shows the answers of the computer is the *terminal*. The program which actually processes commands and returns output is called *shell*.
Nowadays all the UNIX operating systems have a GUI (a desktop environment), but there still exist a CLI that is possible to use by a *terminal emulator* like "xterm".
To start the terminal emulator, just go in the "Applications Menu", then "Accessories" and finally click "Terminal".

1.3 The Linux file system

In Linux, the files and the folders are organized in a ordered tree-structure. The *root directory* is called simply "/" (only the slash). To go directly in the root directory, use the command:
cd /
To see the content of the directory where you are, use the command:
ls
If you are in the root directory, the list of the main directories of the OS will appear. Let's try to change directory with the command:
cd user/
Now you are in the directory that contains the personal files of each user using the system. Try to list again the folders and the files with the command **ls**, find your username and enter in your directory with command:
cd yourUsername/
This is your home directory and you can find your documents, pictures, videos and so on. By the way, when you open the terminal as described in the previous section, you will be always in this directory.

If you want to move up one directory, use the command:
cd ..
If you get lost in the *tree* of the file system, you can see where you are with the command *print working directory*:
pwd

and to go back home, just use:
cd
Finally, to create a new folder use the command **mkdir** followed by the name of the folder, for example:
mkdir first-directory
and enter in the folder with the command **cd**.

1.4 Text editors

There are many text editors in Linux, some of them have a nice graphical interface like "Gedit", others work directly in the terminal, for example "Vi".

1.4.1 Gedit

To create a text file you have just to write in the terminal emulator, the name of the editor followed by the name of the new file (this is a general rule that works with all the text editors). If the name is already used by a file (inside the directory), you will open that file. So let's try
gedit first-file.txt
A graphical interface will open and you can use the editor as you are used in other OS. You can notice that now, you are not able to use anymore the terminal, in the sense that you cannot give any other commands. Then, close the editor and add an ampersand at the end:
gedit first-file.txt &
Now you can use both the editor and the terminal. Just write something, save the file and close it. Try to give the same command, this time you will not create a new file, but you will open the previous one.

1.4.2 Vi

Sometimes, it can happen that you do not have a GUI and you need to read/write a text file from the terminal. The program Vi is a valid solution, even if it can be a bit difficult to use for a beginner. To create/open a file with Vi, you proceed like with gedit, for example:
vi second-file.txt (This time the ampersand is not necessary because the editor works inside the terminal)
Vi is divided in two modes. The *command mode* and the *insert mode*. The first one allows you to give commands, for example, to save your work and to quit the program; the second one is the mode that permits to write a text. When you are in the insert mode, you can go in the command mode by pressing Esc, when you want to move from the command to the insert move, just press I. When you finish to write the text, move in the command mode and to "save and quit" use:
:wq
If you want to quit without saving, use:
:q!
For a list of the commands, I suggest you the webpage `http://www.cs.colostate.edu/helpdocs/vi.html` .

1.5 Manipulating files

1.5.1 Access permissions

Now that in our folder there are a couple of files, let's try to manipulate them. First of all, list them with the command **ls**, but this time add the option **-l**:
ls -l
The answer of the terminal will be similar to the following:
-rw-r–r– 1 yourUsername zedat 253 Oct 24 11:18 first-file.txt
-rw-r–r– 1 yourUsername zedat 325 Oct 24 11:18 second-file.txt
The first character (dash - in this case) indicates the type of file: **d** for directory, **s** for special file, and - for a regular file. The following nine characters show the access permissions (**r**ead, **w**rite and e**x**ecute). The first three characters **rw-** define the owners permission. In this example, the file owner has read and write permissions only. The next three characters **r**– are the permissions for

the members of the same group of the file owner (which in this example is read only). The last three characters **r–** show the permissions for all other users and in this example it is read only. To change the permissions you can use the command "chmod", for example:
chmod "u=rwx","g=rx","o=x" first-file.txt
will change the permissions of user, group and other users.

chmod "u=rwx","g=rx" first-file.txt
will change the permissions of user and group only.

chmod "ugo=rwx" first-file.txt
will give the same permissions to owner, user and group.

The other information given by **ls -l** are the number of links to the files, the name of the owner, of the group, the size (in bytes), modified date and time, actual name of the file/directory.

1.5.2 Rename, move, copy and delete a file

Let's create a new directory "second-directory/" in your personal directory.

The command **mv** permits to do two actions, to *rename* a file:
mv first-file.txt third-file.txt

and to *move* it in another directory:
mv third-file.txt second-directory/

If you want to move the file in another "branch of the tree", you will need to write the full path starting from the root:
mv third-file.txt /home/yourUsername/Documents/

To *copy* a file in another directory, use the command:
cp third-file.txt second-directory/
cp third-file.txt /home/yourUsername/Documents/

To *copy a directory* and its content:
cp -r second-directory/ destination/

To *delete* a file, use the command:
rm third-file.txt

To *delete a directory* and its content:
rm -r second-directory/

1.6 Tar and gzip

Tar is a program used to archive multiple files in packages. **Gzip** is a compression tool used to reduce the size of a file. They can be used together to create compressed packages, like the more popular *zip* or *rar* files.

To archive and compress a folder, use the command:
tar -czvf first-tar.tar.gz first-directory/

To extract the files of a tar archive, use the command:
tar -xzvf first-tar.tar.gz

The meaning of the letters after "-" is the following:
c = to create a new file tar

x = to extract a tar file
v = verbose, to visualize in the terminal the files to compress or extract
f = to create the tar file with the name of the argument
z = to use gzip for the compression/ectraction

1.7 Working from a remote host

If you want to work with the computer of the library remotely (i.e. from a computer connected to internet), you can use the commmand **ssh**:
ssh yourUsername@pool01.chemie.fu-berlin.de
and after you have inserted the password, you can use the terminal like if you were in the library. You can also add **-X** after **ssh** if you want to use a program installed on the remote host that has a GUI, for example MATLAB or Gedit. But if the connection is not good, the program will be very slow and it could crash. In this case you could prefer to work directly with the terminal, for example with Vi.
If you want to transfer files and directories from your personal computer to the library computer and viceversa, you can with the command **scp**.
To transfer a file from a remote host to a local host:
scp yourUsername@pool01.chemie.fu-berlin.de:/remote-path/file.ext /local-path
To transfer a file from a local host to a remote host:
scp /local-path/file.ext yourUsername@pool01.chemie.fu-berlin.de:/remote-path/
If you want to move a directory (and its content), just add **-r** after **scp**.

Of course, you can use these commands if you have *Linux* on your personal computer, but they work also with *Mac OS X*, because it is a UNIX OS. If you want to work with *Windows*, you need to install before an emulator. I warmly suggest MobaXterm (the free edition is enough):
`http://mobaxterm.mobatek.net/download.html`

Other free solutions are PuTTY (with xming) and Cygwin:
`http://www.putty.org/`
`http://sourceforge.net/projects/xming/`
`https://www.cygwin.com/`

2 Python

Python is a high-level programming language characterized by a simple and readable syntax. Moreover Python is open-source and a wide community works continuously for its development. With Python we can easily create algorithms and applications for numerical computation, data analysis and models development. All these features make it very popular inside the science community.
To use Python we just need two programs, a text editor (for example Gedit) and a compiler. The text editor is necessary to write the Python files (extension ".py") that contain the list of instructions (also called source-code) of our program. The compiler is a program that "translates" the source code in computer-language. Moreover the Python compiler, executes our program immediately after the compilation. There are compilers available for Linux, Mac and Windows, but if you do not want to install it on your PC, you can also use the PC-pool version by ssh.

2.1 Let's start...

We will start this tutorial writing a "Hello, World!" program, i.e a program that outputs "Hello, World!" on the monitor. Let's start opening an empty text file with the command:
gedit helloworld.py &
Then we write in text editor the instruction:

```
print "Hello, World!"
```

Finally to compile and run the program, go back in the terminal and execute:
python helloworld.py
The answer will be:
Hello, World!

Very simple, right? Consider that with other programming languages, for example C++, to have the same result we would need to write at least 7 lines of codes. This is one of the reason why Python is so used and appreciated.
It is possible to use python also with a command line interface. Instead of creating a python script, we can open python in the terminal (just execute **python** in the terminal) and write our instructions (e.g.: **print "Hello, World!"**) directly in the CLI, getting the same result.
On the other hand, it is better to save our scripts and to use the command-line interface just to check short pieces of code.
A better alternative to the original command-line interface is **ipython**. This program introduces many useful features. For example we can use the linux commands (cd, ls, pwd), it is possible to edit a python script directly inside ipython with an editor similar to Vi (just run "edit file.py"), it is possible to compile and run python scripts inside the command line interface ("run file.py").

2.2 Variables

One of the most important component of Python (and in general of all programming languages), are the *variables*. A variable is a part of computer memory containing some data. In Python there are five different variables: numbers, strings, lists, tuples and dictionary. Moreover NumPy introduces a new variable called array. During this tutorial we will see only the Numbers and arrays, but at the end of this manual you can find information also about strings, lists and tuples.

2.2.1 Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example:

```
a = 5
```

You can also delete the reference to a number object by using the "del" statement:

```
del a
```

Python supports four different numerical types

1. int (integer numbers): **x=10**
2. float (real numbers): **x=10.8**
3. complex (complex numbers): **x=10+3J**

2.3 Operations

The algebric operations in Python are:

1. Sum

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

   c=8

2. Difference

   ```
   a = 5
   b = 3
   c = a - b
   ```

c=2

3. Multiplication

```
a = 5
b = 3
c = a * b
```

c=15

4. Exponent

```
a = 5
b = 3
c = a ** b
```

c=125

5. Division

```
a = 5
b = 3
c = a / b
```

c=1
The result is 1, because Python is doing a division between integers. To get a float result, you need either to add a "dot" after the number 5 or to use the function **float()**:

```
a = 5.
b = 3
c = a / b     # alternatively: c = float(a)/b
```

c=1.6666666666666667

6. Floor Division (division ignoring decimals)

```
a = 5.
b = 3
c = a // b
```

c=1.0

7. Modulus

```
a = 5
b = 3
c = a % b
```

c=2

2.4 NumPy

NumPy is an open source extension module for Python. It introduces a new type of variables, called "array" that we will use to define vectors and matrices (but it can define also n-dimensional objects).

To import NumPy there are three different methods:

1. `import numpy`

   We need to call any NumPy function, adding the prefix "numpy.", e.g.: numpy.array().

2. `import numpy as np`

   We can just use the prefix "np.", e.g.: np.array()

3. `from numpy import *`

   The prefix is not necessary anymore, e.g.: array().

The three methods are equivalent, but I suggest to use the second one, to remember which functions come from the NumPy module.
Another good reason to use the second method is that if we import (with the asterics) different modules that use the same name for different functions, then Python cannot distinguish them and the functions of the last module would override the functions of the other module.

2.4.1 Define an array

To define an array we need the function np.array().

```
a = np.array([1, 2, 3])  # Create a 1d array (vector)
b = np.array([[1,2,3],[4,5,6]])   # Create a 2d array (matrix)
```

It is also possible to specify if we want an array of int or float.

```
a = np.array([1, 2, 3],dtype=np.int64)  # Create a 1d array of int numbers
a = np.array([1, 2, 3],dtype=np.float64)# Create a 1d array of float numbers

#To declare a float array it is also possible to use the following syntax:
a = np.array([1., 2., 3.])
```

2.4.2 Other functions to create arrays

```
a = np.zeros((2,2))  # Create an array of all zeros
print a              # Prints "[[ 0.  0.]
                     #          [ 0.  0.]]"

b = np.ones((1,2))   # Create an array of all ones
print b              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7) # Create a constant array
print c               # Prints "[[ 7.  7.]
                      #          [ 7.  7.]]"

d = np.eye(2)        # Create a 2x2 identity matrix
print d              # Prints "[[ 1.  0.]
                     #          [ 0.  1.]]"

e = np.random.uniform(0,1,(2,2))         #Matrix 2x2 with random values between 0 and 1
                                         #(extracted from a uniform distribution.

f = np.random.gaussian(0,1,(2,2)) #Matrix 2x2 with random values between 0 and 1
                                  #(extracted from a Gaussian distribution
                                  #with mu=0 and sigma=1).

g = np.linspace(0,5,10) # Return evenly spaced numbers over a specified interval
print a                 # prints array([ 0.   ,  0.55,  1.11,  1.66,  2.22,
                          #              2.77,  3.33,  3.88,  4.44,  5.  ])

b = a.copy()            #Copy the content of the array a in a new array b
```

2.4.3 Array indexing

One of the most important aspects of the array (but also of lists, tuples and strings) is the indexing system. The arrays are a sequence of numbers and to each element of the array corresponds an index.
If we have a vector (1d array) of N elements, the first elements has index 0, the last element has

index $N-1$.
The following example shows how to extract elements from a vector using the indexing system.

```
a = np.array([10, 43, 53, 21, 62, 90])
print a         # Prints the complete array
print a[0]      # Prints the first element of the array
print a[2:5]    # Prints the elements starting from 3rd to 4th
print a[2:]     # Prints the element starting from 3rd element
print a[:2]     # Prints the first two elements of the array
print a[5]      # Prints the last element
print a[-1]     # Prints the last element
print a[-2]     # Prints the second last element
print a[-2:]    # Prints the last two elements
```

If we are working with matrices, we have to indicate the index of the row and of the column, with the syntax [row, column].

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])      # Create the following
                                                         #2d array with shape (3, 4)
                                                         # [[ 1  2  3  4]
                                                         #  [ 5  6  7  8]
                                                         #  [ 9 10 11 12]]

b = a[0:2, 1:3] # Pull out the subarray consisting of the first 2 rows
                # and columns 1 and 2; b is the following array of shape (2, 2):
                # [[2 3]
                #  [6 7]]


a[0, 0] = 77             #to change the first element of the matrix
a[0:2, 0] = np.array([20,20])   #to change the first two elements of the first column
a[2, :] = np.array([20, 20, 20, 20])    #to change all the elements of the last row

#It is also possible to select and edit the elements
#of an array that respect some condition
a[a>6] = 30     #to change all the elements greater than 6
a[a>10,1] = 40  #to change all the elements of the second column greater than 6
```

2.4.4 Array operations

In the following example you can find the math operations that is possible to do with the array.

```
x = np.array([[1.,2.],[3.,4.]]) #matrix 2x2
y = np.array([[5.,6.],[7.,8.]]) #matrix 2x2
v = np.array([9.,10.])  #vector 1x2
w = np.array([11., 12.])  #vector 1x2

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print x + y
print np.add(x, y)

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print x - y
print np.subtract(x, y)

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print x * y
```

```
print np.multiply(x, y)

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print x / y
print np.divide(x, y)

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print np.sqrt(x)

# Inner product of vectors; both produce 219
print v.dot(w)
print np.dot(v, w)

# Matrix / vector product; both produce the 1d array [29 67]
print x.dot(v)
print np.dot(x, v)

# Matrix / matrix product; both produce the 2d array
# [[19 22]
#  [43 50]]
print x.dot(y)
print np.dot(x, y)
```

2.4.5 Array functions

Numpy provides many useful functions for performing computations on arrays, the full list can be found here:
http://docs.scipy.org/doc/numpy/reference/routines.math.html

The following script shows some of the most important functions.

```
x = np.array([[1,2],[3,4]]) #define a matrix

#sum()
np.sum(x)  # Compute sum of all elements; prints "10"
np.sum(x, axis=0)  # Compute sum of each column; prints "[4 6]"
np.sum(x, axis=1)  # Compute sum of each row; prints "[3 7]"

#diagonal()
d = x.diagonal()  #d is a 1d array containing the elements of the diagonal

#transpose
y = x.T          #y is the transpose of x

#inverse
z = np.linalg.inv(x)

#trace
trace = np.trace(x)      #trace of x

#eigenvalues
eigenvalues = np.linalg.eig(x)[0]        #eigenvalues

#eigenvectors
eigenvectors = np.linalg.eig(x)[1]       #eigenvectors (in columns)
```

```
#floor
a = np.array([-1.2, 0.2, 1.8, 9.9])
b = np.floor(a)          #floor() approximates the array toward the
                         #first smallest integer
print b #[-2.  0.  1.  9.]

#determinant
np.linalg.det(x)
```

2.4.6 Math functions

The following are typical math functions that operate on numbers and arrays.

```
import numpy as np

x = np.array([0, 1, 2, 3])

#sine
np.sin(x)

#cosine
np.cos(x)

#tangent
np.tan(x)

#arcsine
np.arcsin(x)

#arccos
np.arccos(x)

#arctan
np.arctan(x)

#logarithm (base e)
np.log(x)

#exponential
np.exp(x)

#absolute value
np.abs(x)

#real part
np.real(x)

#imaginary part
np.imag(x)
```

2.5 Matplotlib

Matplotlib is another useful library that collects several functions to draw graphs. A complete list of the functions can be found here:
`http://matplotlib.org/gallery.html`

The following is an example to plot the function sin(x) and cos(x).

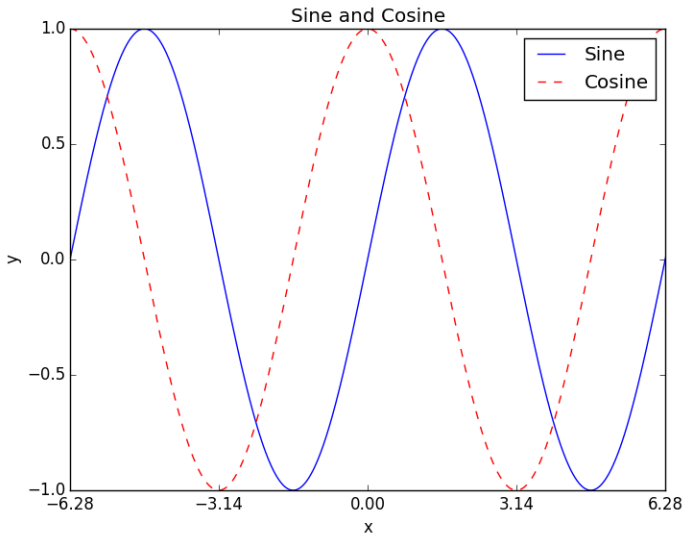
```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.linspace(-10, 10, 1000)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin) #plot a blue line
plt.plot(x, y_cos, 'r--') #plot a red dashed line
plt.xlim(-2*np.pi,2*np.pi) #set the x range
plt.xticks(np.linspace(-2*np.pi,2*np.pi,5)) #set the position of the ticks
plt.xlabel('x')
plt.ylabel('y')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

It is also possible to not open the graph and to save it directly in the working directory, replacing the last line with:

```
plt.savefig('name_figure.eps', format='eps', dpi=1000)
```



2.6 Control Statements: if, for, while loop

The control statements are blocks of instructions executed by Python (or in general by a programming language) only if particular conditions are satisfied or that can be repeated automatically many times. It is important to understand from the beginning that this kind of instructions slow the execution of a program, then, when it is possible, it is important to look for other solutions. In Python+NumPy for example, there is the possibility to *vectorize* an algorithm, but we will talk about this argument later.

2.6.1 For - loop

The "for statements" are a sequence of instructions repeated N times. To realize a for loop, we need a *counter* (i in the example), that increases its value at each iteration. The following example prints out all the element of a list:

```
x = np.array([10, 20, 30, 40, 50])
N = len(x)
for i in range(0,N):     #the counter i takes all the values of the list
 print x[i]              #range(0,N) at each new iteration
```

the function range generates the indeces that we want to explore **range(N) = [0, 1, 2, 3, 4]**. As already explained, the indexes of an array go from 0 to $N-1$.
Another remark about the previous script, is that it is very important to leave a white space before **print x[i]**, otherwise you will get an IndentationError. In fact the identation tells to Python which lines own to the block. Let's try a different code:

```
x = np.array([10, 20, 30, 40, 50])
N = len(x)
for i in range(N):
 print x[i]
print "ok"
```

The result is:
10
20
30
40
50
ok

Let's now add a white space before **print "ok"**:

```
import numpy as np
x = np.array([10, 20, 30, 40, 50])
N = len(x)
for i in range(N):
 print x[i]
 print "ok"
```

The result is:
10
ok
20
ok
30
ok
40
ok
50
ok

This time the instruction **print "ok"** was part of the block "for", and so Python printed out "ok" at each iteration. Let's add another white space before **print "ok"**:

```
import numpy as np
x = np.array([10, 20, 30, 40, 50])
N = len(x)
for i in range(N):
 print x[i]
  print "ok"
```

This time, we will get an error:

```
print "ok"
^
IndentationError: unexpected indent
```

The reason is that, adding another white space, we are saying to Python that the instruction **print "ok"** owns to another block that actaully does not exist in this code.
A final remark, it is not important how many white spaces we use to define a block (I suggest to use at least three spaces or better a TAB space), but it is important that we use always the same number of white spaces inside the same block.

2.6.2 if - else statements

The "if statements" are a sequence of instructions executed only if a particular condition is satisfied. For example:

```
import numpy as np
x = 1
if x==1 :
        print "x is 1"
```

Also in this case, it is important to be careful with the identation.
The other comparison operators are:

- equal ==
- not equal !=
- greater than >
- greater than or equal to >=
- smaller than <
- smaller than or equal to <=

We can also realize complex conditions using the logic operators & (AND) and | (OR):

```
x = 1
y = 3
z = 0
if ( x==1 & y<2 ) | z!=1 :
        print "true"
```

We are asking to Python: "If x is equal to 1 AND y is smaller than 2, OR z is not equal to 1" then print "true".
We can also add several options using "elif" (else if) and "else":

```
x = 21
if x == 1:
        print "1"
elif x == 2:
        print "2"
else:
        print "another number"
```

2.6.3 While loop

"While" is something similar to "for" and we will use it when we are asking to python to repeat some instructions, until a certain condition is reached. For example:

```
x = 0
while (x < 9):
        print "The count is:", x
        x = x + 1
```

In this case we are asking to Python to increase the variable x until it is equal to 9.

2.7 Functions

A function is a block of code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. Some functions are already implemented in Python, for example **sum(x)** returns the sum of the elements of the list **x**. Other useful functions are **min(x)** (minimum element of a list), **max(x)** (maximum element of a list), **abs(x)** (absolute value of a number).
If your are working with a CLI, the function **help()** gives a description of a function directly in the terminal (e.g.: help(sum).
In the official Python documentation you can find a complete list of functions already implemented: `https://docs.python.org/2/library/functions.html`

As we have already saw, NumPy (and many other libraries available online) introduces new functions. Furthermore it is also possible to write our own functions, directly in the main program or in a separate file (better option).
In the following example, we have two files. The first one defines the functions mean() $\frac{1}{N}\sum_{i=0}^{N} x_i$ and standard_deviation() $\sqrt{\frac{1}{N}\sum_{i=0}^{N} x_i - \langle x \rangle}$.
The second file is the main program and computes the mean and the standard deviation of an array, calling the functions defined in the file userfunctions.py.

File userfunctions.py:

```
import numpy as np

def my_mean(x):
        N = len(x)
        s = np.sum(x)

        return s/N

def my_standard_deviation(x):
        N = len(x)
        m = my_mean(x)
        s = 0

        for i in range(0,N):
                s = s + (x[i]-m)**2

        return np.sqrt(1./N*s)
```

File main.py:

```
import numpy as np
from userfunctions import *

x = np.random.normal(5,3,10000) #gaussian distribution with mu=5 and sigma=3
m = my_mean(x)
s = my_standard_deviation(x)
print m
```

```
print s
```

You can also check the results with the NumPy functions.

```
m = np.mean(x)
s = np.(x)
print m
print s
```

2.8 SciPy

Numpy provides a high-performance multidimensional array and basic tools to compute with and manipulate these arrays. SciPy builds on this, and provides a large number of functions that operate on numpy arrays and are useful for different types of scientific and engineering applications. The best way to get familiar with SciPy is to browse the documentation:
`http://docs.scipy.org/doc/scipy/reference/index.html`

The following example shows how to interpolate a set of points with the spline method imported from SciPy:

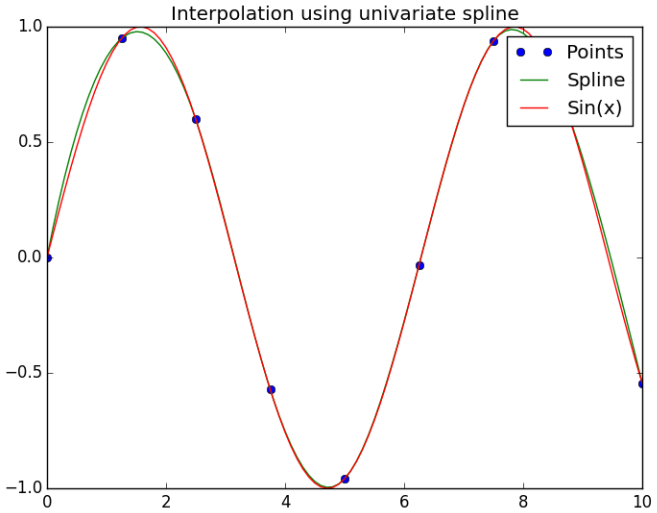
```
import numpy as np
from scipy.interpolate import InterpolatedUnivariateSpline
import matplotlib.pyplot as plt

# setup data
x = np.linspace(0, 10, 9)
y = np.sin(x)
xi = np.linspace(0, 10, 101)

# use fitpack2 method
ius = InterpolatedUnivariateSpline(x, y)
yi = ius(xi)


plt.plot(x, y, 'bo')
plt.plot(xi, yi, 'g')
plt.plot(xi, np.sin(xi), 'r')
plt.title('Interpolation using univariate spline')
plt.legend(['Points','Spline','Sin(x)'])

plt.show()
```



2.9 Other Python variables

2.9.1 Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks:

```
str = 'Hello World!'
```

It is possible to print out the content of a string using the "print" statement:

```
print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str[:2]      # Prints the first two elements of the string
print str[-1]      # Prints the last element
print str[-2]      # Prints the second last element
print str[-2:]     # Prints the last two elements
```

2.9.2 Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets.

```
lis1 = [1, 2, 3, 10, 50, 1000]
lis2 = ['abcd', 786 , 2.23, 'john', 70.2]
lis3 = [[1, 2, 3],[4,5,6],[7,8,9]]      #list of lists
```

We can also define a list as a sequence of numbers using the function range():

```
x = range(1,6)      # defines a list of numbers from 1 to 5
```

Another useful function is len(), to get the length of a list (or of a string):

```
len(x)
```

```
5
```

2.9.3 Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.
The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as read-only lists.

```
lis1 = (1, 2, 3, 10, 50, 1000)
lis2 = ('abcd', 786 , 2.23, 'john', 70.2)
```

It is also possible to convert a list/tuple in a NumPy array.

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
b = np.asarray(a)
```