SCRIPTING AND PROGRAMMING LABORATORY FOR DATA ANALYSIS

Lecture 3 - part 1

Input/output and file/directory managing

Files and Directories

Python has many functionalities to manage files and directories; those that are most useful are implemented in various modules:

- **open** - built-in function to read and write files
- **os** - provides a portable way of using operating system dependent functionality
- **sys** - provides various functions and variables that are used to manipulate different parts of the Python runtime environment
- **shutil** - offers a number of high-level operations on files and collections of files (copying and removal)
- **glob** - finds all the pathnames matching a specified pattern according to the rules used by the Unix shell

Files reading and writing

Object file

The built-in function **open** creates/opens a file on the disc returning a **file object**

(see here https://docs.python.org/3/library/functions.html#open):

open(filename[, mode [, buffering]]) → file object

```
infile = open('/path_to_filename/filename.txt',mode)
```

Name of file object

path can be absolute or relative

Opening mode

Name of the file on disc

Filename is the only compulsory argument, in this case the file is opened in reading mode. If the file cannot be opened, an *OSError* is raised.

Object file

open(filename[, mode [, buffering]]) → file object

- if **mode == 'r' (default mode)**: the file is opened in read-only mode;
- if **mode == 'w'**: the file is deleted and a new file is opened in write-only mode;
- if **mode == 'a'**: the file is opened and new content is put at the end of it;
- if **mode == 'r+'**: the file is opened in read-write mode.
- if you add a **b** to above options: the file is considered a binary file (i.e. not human readable). In this case you need to manage with bytes contained in the file.

Object file - Reading

When **mode == 'r'** you can read a file with open

```
infile = open("two_col_file.dat","r") # open the file
```

The main methods we are interested to are:

- **infile.read():** the whole file is read into a single string. If a positive integer **p** is passed as argument, then an amount of **p** characters are read *if the file is an ASCII file*.

  If the file is a binary file the number passed instructs infile to read an equal amount of bytes from the file.

  When you reach the end of the file empty string is returned;

Object file - Reading

When **mode == 'r'** you can read a file with open

```
infile = open("two_col_file.dat","r") # open the file
```

The main methods we are interested to are:

- **infile.readline():** this reads the next line of the file and returns it as a string.

  If you just opened the file, **infile.readline()** reads the first line. If you call it again, it reads the second, and so on. If you reached the end of the file, it returns an empty string.

  You can move back to a given point in the file using **infile.seek(offset)** where **offset** in an integer;

Object file - Reading

When **mode == 'r'** you can read a file with open

```
infile = open("two_col_file.dat","r") # open the file
```

The main methods we are interested to are:

- **infile.readlines():** this returns a **list of strings**. Each string is a line of the file. It thus reads all the file at once;

```
[1]: infile = open("example_file.txt",'r')

[2]: infile.readlines()

[2]: ['hello\n', '75\n', 'a54\n', '42\n']
```

Object file - Reading

When **mode == 'r'** you can read a file with open

```
infile = open("two_col_file.dat","r") # open the file
```

The main methods we are interested to are:

- **infile.flush():** clears the internal buffer, e.g. deletes any content within outfile object

Object file - Writing

When **mode == 'w'** or **mode == 'a'** you can write a file on the disc

```
outfile = open("write_file.txt","w")
```

If in **'w'** mode the current position is at the beginning of file.

If in **'a'** mode the current position is **EOF** (end of file)

Object file - Writing

When **mode == 'w'** or **mode == 'a'** you can write a file on the disc

```
outfile = open("write_file.txt","w")
```

The main methods we are interested to are:

- **outfile.write(string):** write the variable **string** in the file

```
outfile = open("write_file.txt","w")

outfile.write("hello\n")
```

\n is necessary if you want a newline at next call of write

Object file - Writing

When **mode == 'w'** or **mode == 'a'** you can write a file on the disc

```python
outfile = open("write_file.txt","w")
```

The main methods we are interested to are:

- **outfile.writelines(list):** write **list** in the file, the list is collapsed into a string and it is written into the file

```python
my_list = ["ciao","5","457e"]

outfile.writelines(my_list)
```

my_list must be a list of strings!!

Object file - Writing

When **mode == 'w'** or **mode == 'a'** you can write a file on the disc

```
outfile = open("write_file.txt","w")
```

The main methods we are interested to are:

- **outfile.writelines(list):** write **list** in the file, the list is collapsed into a string and it is written into the file

```
my_list = ["ciao","5","457e"]
```

my_list must be a list of strings!!

```
GNU nano 4.8                    write_file.txt
1 ciao5457e
2
```

Object file - Writing

When **mode == 'w'** or **mode == 'a'** you can write a file on the disc

```
outfile = open("write_file.txt","w")
```

The main methods we are interested to are:

- **outfile.truncate(size):** truncates the file at **size**. If no argument it truncates the file at the current position.

Object file - Position

When **reading** or **writing** you are changing position in the file. You can navigate the file with two methods:

```
file = open("example_file.txt","r+")
```

- **file.tell():** returns an integer giving the current position in bytes
- **file.seek(offset, [whence]):** allows to move in the file. offset tells how many bytes to move, while whence specify the starting point of your movement:
  - 0(default): move by offset bytes from the beginning
  - 1: move by offset bytes from current position
  - 2: move by offset bytes from **EOF**

Object file - Closing

You can close the file by invoking the method **close()**

```
file = open("example_file.txt","r+")
```

- **file.close():** this closes the file and frees the memory. After that, any action on file raises an error

Object file - With block

To avoid explicitly taking care of file closing and memory free you can use the **with** keyword:

```
[1]: with open("example_file.txt","r") as infile:
         for i,line in enumerate(infile):

             print(i,line)

     0 hello

     1 75

     2 a54

     3 42

[2]: infile.read()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-2-9039e9e9da6f> in <module>
----> 1 infile.read()

ValueError: I/O operation on closed file.
```

As you exit the indented block of **with**, *the file gets automatically closed*

Files and directories Handling

OS MODULE

The **os** module contains several functions for manipulation of files, directories and processes. It provides a portable way of using operating system dependent functionality(see here):

- Navigate through files and directories
  - **os.getcwd( )**: returns the current working directory;
  - **os.chdir (path)**: changes the current directory to the directory defined by path;
  - **os.listdir (path)**: list all files inside the directory defined by path.

Path handling

Within **os** the sub-module **os.path** implements some useful functions acting on pathnames (see here):

- Information about paths (bools)
  - **isdir(string)**: tells if string is a folder
  - **isfile(string)**: tells if string is a file
  - **islink(string)**: tells if string is a link
  - **exist(sting)**: tells if string exists

Path handling

Within **os** the sub-module **os.path** implements some useful functions acting on pathnames (see here):

- Manipulation of paths
  - **normcase(string)**: Normalize the case of a pathname. On Windows, convert all characters in the pathname to lowercase, and also convert forward slashes to backward slashes. On other operating systems, return the path unchanged.
  - **join(path, *paths)**: joins one or more path components intelligently (e.g. removing or adding extra slash /).
  - **split(string)**: Split the pathname path into a pair, (head, tail) where tail is the last pathname component and head is everything leading up to that. The tail part will never contain a slash.

```
[77]: os.path.split("/home/mbonetti/Dropbox/python_scripting_como/Lecture3/write_file.txt")

[77]: ('/home/mbonetti/Dropbox/python_scripting_como/Lecture3', 'write_file.txt')
```

Path handling

Within **os** the sub-module **os.path** implements some useful functions acting on pathnames (see here):

- Information about path (strings)
  - **abspath:** gives a string with the absolute path
  - **basename:** returns a string with the name of the file
  - **dirname:** returns the directory name

In the second call an empty string is returned as there is no path!

```
[67]: import os

      os.path.abspath("write_file.txt")

[67]: '/home/mbonetti/Dropbox/python_scripting_como/Lecture3/write_file.txt'

[68]: os.path.basename("/home/mbonetti/Dropbox/python_scripting_como/Lecture3/write_file.txt")

[68]: 'write_file.txt'

[69]: os.path.dirname("/home/mbonetti/Dropbox/python_scripting_como/Lecture3/write_file.txt")

[69]: '/home/mbonetti/Dropbox/python_scripting_como/Lecture3'

[70]: os.path.dirname("write_file.txt")

[70]: ''
```

OS MODULE

The **os** module contains several functions for manipulation of files, directories and processes. It provides a portable way of using operating system dependent functionality(see here):

- Create, rename or delete directories:
    - **os.mkdir (path [, mode=0777])**: creates a directory (**mode=0777** is a default and deals with folder permission in UNIX);
    - **os.makedirs(path [, mode=0777])**: create directories recursively;
    - **os.rmdir(path)**: removes the directory in path (BE CAREFUL!);
    - **os.removedirs(path)**: removes directories recursively (BE CAREFUL!);
    - **os.remove (path)**: removes a file (BE CAREFUL!);
    - **os.rename (old, new)**: renames files or directories;
    - **os.renames (old, new)**: renames files or directories recursively;
    - **os.stat(path)**: returns a tuple with info (dimension, last access,
    - last modification, privileges);
    - **os.chmod (path, mode)**: changes privileges of file or folder.

OS MODULE

Mode must be in octal base

os.chmod (path, mode): changes privileges of file or folder.

```
$ ls -l file_name
```

```
Output
-rw-r--r-- 12 linuxize users 12.0K Apr  28 10:10 file_name
|[-][-][-]-   [------] [---]
| |  |  | |      |       |
| |  |  | |      |       +-----------> 7. Group
| |  |  | |      +-------------------> 6. Owner
| |  |  | +--------------------------> 5. Alternate Access Method
| |  |  +----------------------------> 4. Others Permissions
| |  +-------------------------------> 3. Group Permissions
| +----------------------------------> 2. Owner Permissions
+------------------------------------> 1. File Type
```

The first character indicates the file type. It can be a regular file (`-`), directory (`d`), a [symbolic link](#) (`l`), or other special types of files. The following nine characters represent the file permissions, three triplets of three characters each. The first triplet shows the owner permissions, the second one group permissions, and the last triplet shows everybody else permissions.

In the example (rw-r--r--) means that the file owner has read and write permissions (rw-), the group and others have only read permissions (r--)

- `r` (read) = 4
- `w` (write) = 2
- `x` (execute) = 1
- no permissions = 0

- Owner: rwx=4+2+1=7
- Group: r-x=4+0+1=5
- Others: r-x=4+0+0=4

OS MODULE

*We create the folder **new** that contains the directory **folder***

We navigate the filesystem with the function walk. We navigate from current directory ('.')

os.walk(path):
Generate the file names in a directory tree by walking the tree either top-down or bottom-up.
For each directory in the tree rooted at directory top (including top itself), it yields a
3-tuple (dirpath, dirnames, filenames).

```
[82]: os.makedirs(os.path.join(os.getcwd(),os.path.join('new','folder')))
```

```
[93]: for root, direc, files in os.walk('.'):
          print("\nDirectories:")
          for name in direc:
              print(os.path.join(root, name))
          print("\nFiles:")
          for name in files:
              print(os.path.join(root, name))
```

```

Directories:
./.ipynb_checkpoints
./new

Files:
./write_file.txt
./Exercises_lecture_3.ipynb
./file.vtk
./test.ipynb
./example_file.txt

Directories:

Files:
./.ipynb_checkpoints/example_file-checkpoint.txt
./.ipynb_checkpoints/write_file-checkpoint.txt
./.ipynb_checkpoints/test-checkpoint.ipynb
./.ipynb_checkpoints/Exercises_lecture_3-checkpoint.ipynb

Directories:
./new/folder

Files:

Directories:

Files:
```

SHUTIL MODULE

The **shutil** module provides additional functions for high level file operation like copying and erasing (see here):

- **shutil.copy (src, dst):** copies the file src to the file or directory dst
- **shutil.copyfile (src, dst):** copies the contents of the file named src to a file named dst;
- **shutil.copymode (src, dst):** copies the permission bits from src to dst. The file contents, owner, and group are unaffected;
- **shutil.copystat (src, dst):** copies the permission bits, last access time, last modification time, and flags from src to dst;
- **shutil.copytree (src, dst [,symlinks]):** recursively copies an entire directory tree rooted at src to a directory named dst and return the destination directory;
- **shutil.rmtree(path[, ignore_errors[, onerror]]):** deletes an entire directory tree; path must point to a directory (BE CAREFUL!);
- **shutil.move (src, dst):** recursively moves a file or directory src to another location dst and return the destination.

GLOB MODULE

The **glob** module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell. No tilde expansion is done, but *, ?, and character ranges expressed with [] will be correctly matched (see here):

- **glob.glob(pathname) -> list**: Return a possibly-empty list of path names that match pathname, which must be a string containing a path specification.
- **glob.iglob(pathname) -> iterator**: Return an iterator which yields the same values as **glob()** without actually storing them all simultaneously.
- **glob.fnmatch.fnmatch(filename,pattern) -> bool**: Test whether the filename string matches the pattern string, returning True or False. Both parameters are case-normalized, i.e. the function is case insensitive (use **fnmatchcase** instead)

GLOB MODULE

Check what's inside new/

Collect only files matching a pattern

```
[102… os.listdir("new")

[102… ['file1.txt', 'text.dat', 'file4.dat', 'file2.txt', 'file3.txt']

[103… import glob

      glob.glob("new/file*txt")

[103… ['new/file1.txt', 'new/file2.txt', 'new/file3.txt']

[104… for c in glob.iglob("new/file*txt"):
          print(c)

      new/file1.txt
      new/file2.txt
      new/file3.txt
```

Collect only files matching a pattern and store them into an iterator object that we use into a for cycle

GLOB MODULE

Check what's inside new/

Collect only files matching a pattern of the linux shell

```
[102… os.listdir("new")

[102… ['file1.txt', 'text.dat', 'file4.dat', 'file2.txt', 'file3.txt']

[103… import glob

      glob.glob("new/file*txt")

[106… for file in os.listdir('new'):
          if glob.fnmatch.fnmatch(file, '*.txt'):
              print(file)

      file1.txt
      file2.txt
      file3.txt
```

Here we check if the pattern match, and if it is the case we print the collected files

Command line/Keyboard input

SYS MODULE

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter (see here for a complete list). Among many functions, **sys.argv** is probably the most used one

SYS MODULE

Suppose you are running your Python script in a terminal and want to pass different input arguments directly from command line, i.e.

```
python3 test.py arg1 arg2 arg3
```

```python
import sys

print('Number of arguments:', len(sys.argv), 'arguments.')
print('Argument List:', str(sys.argv))
```

sys module gives easy access to any command-line arguments

- **sys.argv** is the list of command-line arguments.
- **len(sys.argv)** is the number of command-line arguments.

Here **sys.argv[0]** is the program, i.e. script name, while **sys.argv[i]** with **i = 1,...,len(sys.argv)-1** represent the arguments.

Remind: any **sys.argv[i]** is considered a string, so remind to cast any argument to the proper type you intended!

Input from keyboard

Any input from keyboard is managed by the built-in function **input**. Practically it reads one line from standard input and returns it as a string.

`input([prompt])`

If the prompt argument is present, it is written to standard output without a trailing newline.

```
[*]: string = input("Enter your input: ") # this will appear on the terminal and the script will wait a keyword input
     print("Received input is : ", string)

     Enter your input:
```

Here the interpreter waits for input

```
[52]: string = input("Enter your input: ") # this will appear on the terminal and the script will wait a keyword input
      print("Received input is : ", string)

      Enter your input: ciao
      Received input is :  ciao
```

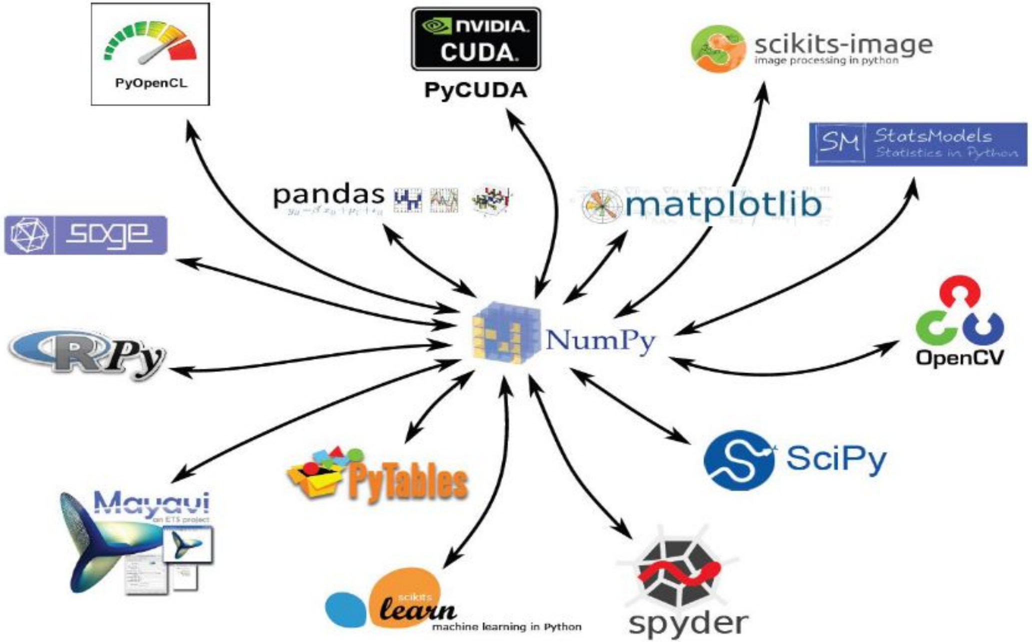
SCRIPTING AND PROGRAMMING LABORATORY FOR DATA ANALYSIS

Lecture 3 - part 2

Numpy

Numerical Python - NumPy

NumPy is an extension of Python built to optimise the management of large collections of data and primarily designed to be used in scientific applications.



Numerical Python - NumPy

In pure Python we find:

- numerical objects: mainly integer and floating point
- containers: lists, tuples, sets, dictionaries

NumPy enriches the tools at our disposal by providing:

- an intuitive multidimensional array object, the **ndarray**, together with computationally efficient methods to access and employ it;
- a set of useful mathematical multipurpose tools, e.g. linear algebra, FFT, random numbers…

Numerical Python - NumPy

The efficiency of *NumPy* leverages on some strength points:

- Functions and methods can act on entire vectors and matrices, i.e. no need of slow explicit **for loops**;
- Algorithms are specifically designed for efficiency and are combined with a more efficient memory management;
- Most part of *NumPy* is written in C and wrapped by Python, making *NumPy* naturally faster;
- Large data set can be memory-mapped, allowing for efficient reading/writing operations.

NumPy

Everything starts with an import statement:

```
import numpy as np # everything inside numpy now can be invoked as np.something
```

The package is organised in few sub-packages

Core contains the **ndarray**

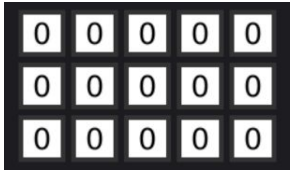
Sub-Packages	Purpose	Comments
core	basic objects	all names exported to numpy
lib	Addintional utilities	all names exported to numpy
linalg	Basic linear algebra	LinearAlgebra derived from Numeric
fft	Discrete Fourier transforms	FFT derived from Numeric
random	Random number generators	RandomArray derived from Numeric
distutils	Enhanced build and distribution	improvements built on standard distutils
testing	unit-testing	utility functions useful for testing
f2py	Automatic wrapping of Fortran code	a useful utility needed by SciPy

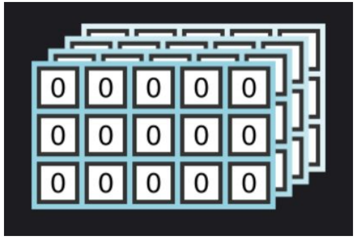
NumPy- ndarray

Ndarray is a multidimensional fixed type structure, i.e. differently from lists, numpy array **must** contain variables of the **same type**!

When referring to **ndarray** it is important to know:

- **rank** -> this is simply the number of axes (or dimensions)
  - A 1-dimensional array has rank 1 (a list)
  - A 2-dimensional array (a matrix) has rank 2
  - A 3-dimensional array has rank 3, e.g. a stack of matrices
  - A N-dimensional array has rank N





NumPy- ndarray

Ndarray is a multidimensional fixed type structure, i.e. differently from lists, numpy array **must** contain variables of the **same type**!

When referring to **ndarray** it is important to know:

- **shape** -> The shape of an array specifies the length of the array in each dimension. It is usually represented as a **tuple**. For a given array, the number of elements in the shape tuple will be equal to the **rank** of the array.

Array creation from lists

```
a = np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11]])
```

```
a.shape
```

```
(3, 4)
```

Tupla with the dimensions

Rank is 2

NumPy- ndarray

Ndarray is a multidimensional fixed type structure, i.e. differently from lists, numpy array **must** contain variables of the same **type**!

When referring to **ndarray** it is important to know:

- **size** -> The size of an array is simply the total number of elements. It is found by multiplying together all the elements of the **shape**.

NumPy- ndarray

Ndarray is a multidimensional fixed type structure, i.e. differently from lists, numpy array **must** contain variables of the **same type**!**

When referring to **ndarray** it is important to know:

- **dtype** -> this is the type of elements stored into a ndarray, typically ints or floats. For example, data type numpy.int32 is a 32 bit integer that occupies exactly 4 byte (32 bits) of memory.

** see next for a better definition

NumPy- ndarray

Ndarray is a multidimensional fixed type structure, i.e. differently from lists, numpy array **must** contain variables of the same **type**!

When referring to **ndarray** it is important to know:

- **itemsize** -> the dimension in memory of a single element. So for example an int32 array will have an itemsize of 4 (32 bits divided by 8 gives 4 bytes).

NumPy- ndarray

Ndarray summary:

```
[110…  a2 = np.zeros((3, 5))

       print(a2.ndim)      # 2
       print(a2.shape)     # (3, 5)
       print(a2.size)      # 5
       print(a2.dtype)     # float64
       print(a2.itemsize)  # 8 (float64 is an 8 byte quantity)
       print(a2.data)      # <memory at XXXX>

       2
       (3, 5)
       15
       float64
       8
       <memory at 0x7f7c7041e110>
```

rank

shape

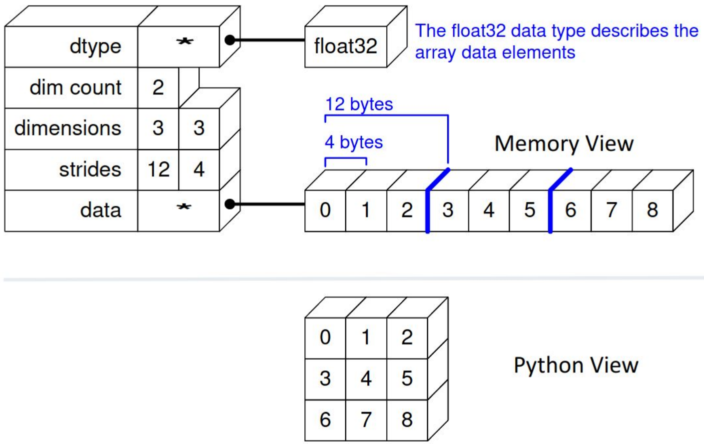
size

data type

Dimension of each element

memory address

```
[111…  a2.data?

       Type:        memoryview
       String form: <memory at 0x7f7c7041e110>
       Length:      3
       Docstring:   Create a new memoryview object which references the given object.
```



NumPy- ndarray creation

We can create a **ndarray** is several ways, the most straightforward is to use the array function:

array(object, dtype=None, copy=1, order=None) -> array

- Where **object** is actually the collection of data you want to store in the array.

This is a rank 2 array!

```
lista=[1,2,3,4]
tupla=(5,6,7,8)

a=np.array(lista) #from a list
b=np.array(tupla) #from a tupla
c=np.array([lista,tupla]) #from a list and from a tupla

print(a)
print(b)
print(c)
```

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

NumPy- ndarray creation

We can create a **ndarray** is several ways, the most straightforward is to use the array function:

```
array(object, dtype=None, copy=1, order=None) -> array
```

- With **dtype** you can specify the date type of the elements. We have more data type with respect to pure python: like float or double of C. See here for a list of available data types.

NumPy- ndarray creation

We can create a **ndarray** is several ways, the most straightforward is to use the array function:

array(object, dtype=None, copy=1, order=None) -> array

- You can have a user-defined data type: **dt=np.dtype([('Name','S3'),(‘years', np.int64)])**, then you can create an array with such type!

```
dt=np.dtype([('Name','S3'),('Years', np.int64)])
a=np.array([('Chiara',3),('Marco',4)],dtype=dt)

print(a)
```

```
[(b'Chi', 3) (b'Mar', 4)]
```

Custom type: each element is formed by a string of 3 char and an integer

Note that any method and function refers to ***np***

NumPy- ndarray creation

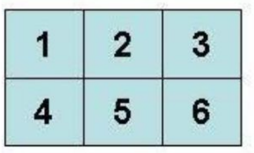
We can create a **ndarray** is several ways, the most straightforward is to use the array function:

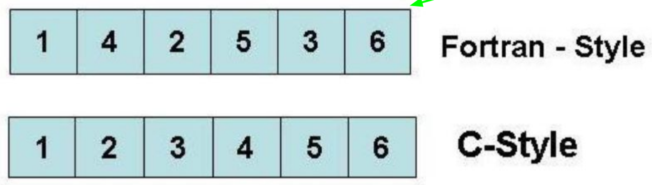
`array(object, dtype=None, copy=1, order=None) -> array`

- With **order** you can specify how data are stored in memory

Column-major order

This is a rank 2 array!





Row-major order

Different part of the array are contiguous in memory, but the ordering is different!

NumPy- ndarray creation

A **ndarray** can be created using some built-in functions

- Creation of an array of dimension **shape** filled with zeros

  `zeros( shape, dtype=float, order ='C' )`

- Creation of an array of dimension **shape** filled with ones

  `ones( shape, dtype=None, order ='C' )`

- Creation of a diagonal **NxN** matrix

  `diagonal( N, dtype=None, order ='C' )`

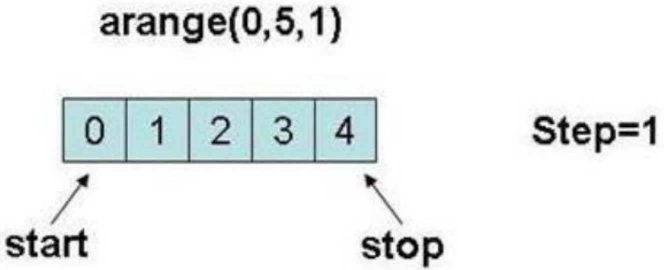
- Creation of a **NxM** matrix, with **k**-diagonal filled with ones

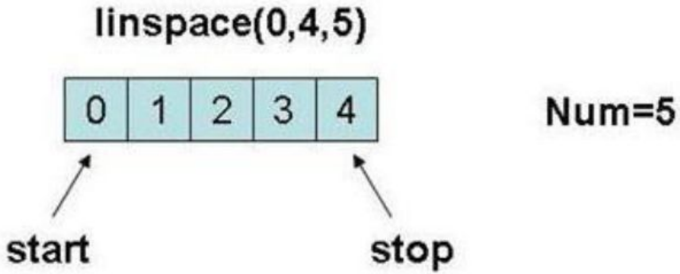
  `eye( N, M=None, k=0, dtype=float )`

NumPy- ndarray creation

A **ndarray** can be created using some built-in functions

- Creation of a uniformly spaced sequence
  - **arange([start,] stop[, step,], dtype=None):** this goes from start to stop (excluded) with a given step;
  - **linspace(start, stop, num=50, endpoint=True):** this goes from start to stop (included) using num points;





NumPy- ndarray creation

A **ndarray** can be created using some built-in functions

- Directly from strings using the function **fromstring**

```
[128… np.fromstring("1.2 2 3 4",sep=' ')
[128… array([1.2, 2. , 3. , 4. ])
```

Specify the separator!

NumPy- Reshaping & Resizing

Reshape and Resize are two methods through which you can modify the shape and dimension of a numpy array.

- **reshape(shape, order='C'):** you get a **new** structure where data are redistributed according the new **shape**. The dimensions of the initial and final array must be the same, i.e. N = MxK

```
a = np.arange(0,20,1)
print(a)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

```
b = a.reshape(5,4)
print(a)
print("\n")
print(b)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]


[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]
```

NumPy- Reshaping & Resizing

Reshape and Resize are two methods through which you can modify the shape and dimension of a numpy array.

- **resize(new_shape, order='C'):** the array is changed in-place such that data are redistributed according the the new **shape**. The dimensions of the initial and final array can be different

```
c = np.arange(0,20,1)
print(c, c.size)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19] 20
```

```
c.resize((5,6))
print(c,c.size)
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19  0  0  0  0]
 [ 0  0  0  0  0  0]] 30
```

Missing elements are filled by zeros

NumPy- Reshaping & Resizing

Reshape and Resize are two methods through which you can modify the shape and dimension of a numpy array.

- **resize(new_shape, order='C')**: the array is changed in-place such that data are redistributed according the the new **shape**. The dimensions of the initial and final array can be different

Note that resize works only if the array is not reference or it is references by another array!!

NumPy- Reshaping & Resizing

Reshape and Resize are two methods through which you can modify the shape and dimension of a numpy array.

```
d = c
c.resize((5,6))
print(c,c.size)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_570902/1956066370.py in <module>
      1 d = c
----> 2 c.resize((5,6))
      3 print(c,c.size)

ValueError: cannot resize an array that references or is referenced
by another array in this way.
Use the np.resize function or refcheck=False
```

NumPy- Indexing, Slicing

Like for **lists**, you can access elements of **ndarray** through **square brackets []**. **ndarray** also support the slicing operator [:].

- 1-d array employs the very same notation of lists

```
a = np.arange(0,20,1)
print(a)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

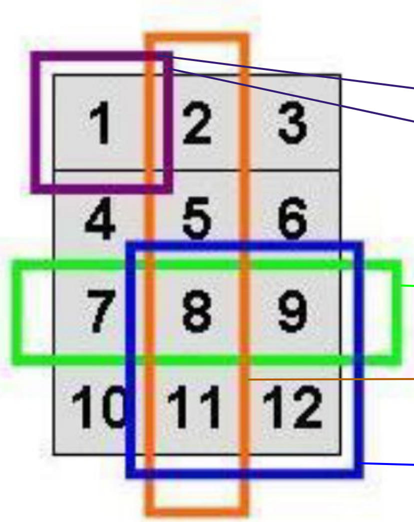
```
print(a[0],a[1:5])
```

```
0 [1 2 3 4]
```

NumPy- Indexing, Slicing

Like for **lists**, you can access elements of **ndarray** through **square brackets** []. **ndarray** also support the slicing operator [:].

- N-d array are more powerful




```
a = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
```

```
print(a[0][0],"\n")
print(a[0,0],"\n")
print(a[2],"\n")
print(a[:,1],"\n")
print(a[2:,1:3],"\n")
```

```
1

1

[7 8 9]

[ 2  5  8 11]

[[ 8  9]
 [11 12]]
```

NumPy- Indexing, Slicing

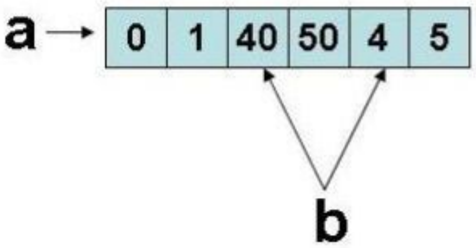
Copying a **ndarray** requires a little more care: a slicing operation creates a view onto the original array, if you modify the view, you are changing the original array!

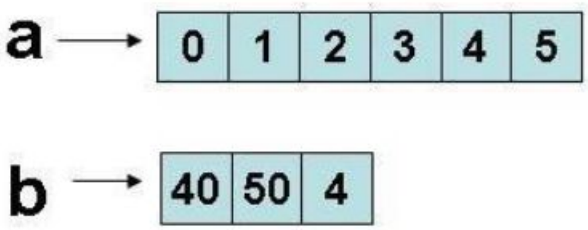
```python
a = np.arange(6) # this is a numpy array
b = a[2:5]
b[0] = 40
b[1] = 50

print(a)
```

```
[ 0  1 40 50  4  5]
```

```python
a = list(range(6)) # this is a list
b = a[2:5]
b[0] = 40
b[1] = 50

print(a)
```

```
[0, 1, 2, 3, 4, 5]
```





NumPy- Indexing, Slicing

Copying a **ndarray** requires a little more care: a slicing operation creates a view onto the original array, <u>if you modify the view, you are changing the original array</u>!

In order to obtain a <u>copy by value</u> (instead a <u>copy by reference</u>), you need to use the function **copy**

```python
a = np.arange(6) # this is a numpy array
c = a.copy()
c[1] = 50

print(a)
print(c)
```

```
[0 1 2 3 4 5]
[ 0 50  2  3  4  5]
```

NumPy- Fancy Indexing

Fancy indexing is a more elaborate way to access multiple array elements at once.

We can achieve this through:

- **An array of indices** (a NumPy array, a Python list, or a sequence of integers)
- **A boolean mask**

NumPy- Fancy Indexing

Fancy indexing is a more elaborate way to access multiple array elements at once.

We can achieve this through:

- **An array of indices**

A new array y is created from x and containing those values corresponding to indices into the array inside [].

The new array has the shape of the index array and the values of the starting array x

```
x = np.arange(10,20,1)
y = x[np.array([2,2,4,6])]
z = x[np.array([[2,3,5],[7,8,9]])]


print(x,"\n")
print(y,"\n")
print(z,"\n")
```

```
[10 11 12 13 14 15 16 17 18 19]

[12 12 14 16]

[[12 13 15]
 [17 18 19]]
```

NumPy- Fancy Indexing

Fancy indexing is a more elaborate way to access multiple array elements at once.

We can achieve this through:

- **A boolean mask:** this is even more powerful as you can select elements satisfying a boolean condition

This selects all elements satisfying the condition

We can directly change elements that satisfy the condition into the array

```python
w = np.array([
[ 0, 1, 2, 3, 4, 5],
[ 6, 7, 8, 9, 10, 11],
[12, 13, 14, 15, 16, 17]])

k = w[w>10]

w[w>10] = 1

print(k,"\n")
print(w,"\n")
```

```
[11 12 13 14 15 16 17]

[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10  1]
 [ 1  1  1  1  1  1]]
```

NumPy- Fancy Indexing

Fancy indexing is a more elaborate way to access multiple array elements at once.

We can achieve this through:

- **A boolean mask:** this is even more powerful as you can select elements satisfying a boolean condition

This operation creates a boolean mask

```
w = np.array([
[ 0, 1, 2, 3, 4, 5],
[ 6, 7, 8, 9, 10, 11],
[12, 13, 14, 15, 16, 17]])

bb = w>10

print(bb,"\n")
print(w[bb])
```

```
[[False False False False False False]
 [False False False False False  True]
 [ True  True  True  True  True  True]]

[11 12 13 14 15 16 17]
```

This selects all elements satisfying the condition

We can directly change elements that satisfy the condition into the array

NumPy- Fancy Indexing

Fancy indexing is a more elaborate way to access multiple array elements at once.

We can achieve this through:

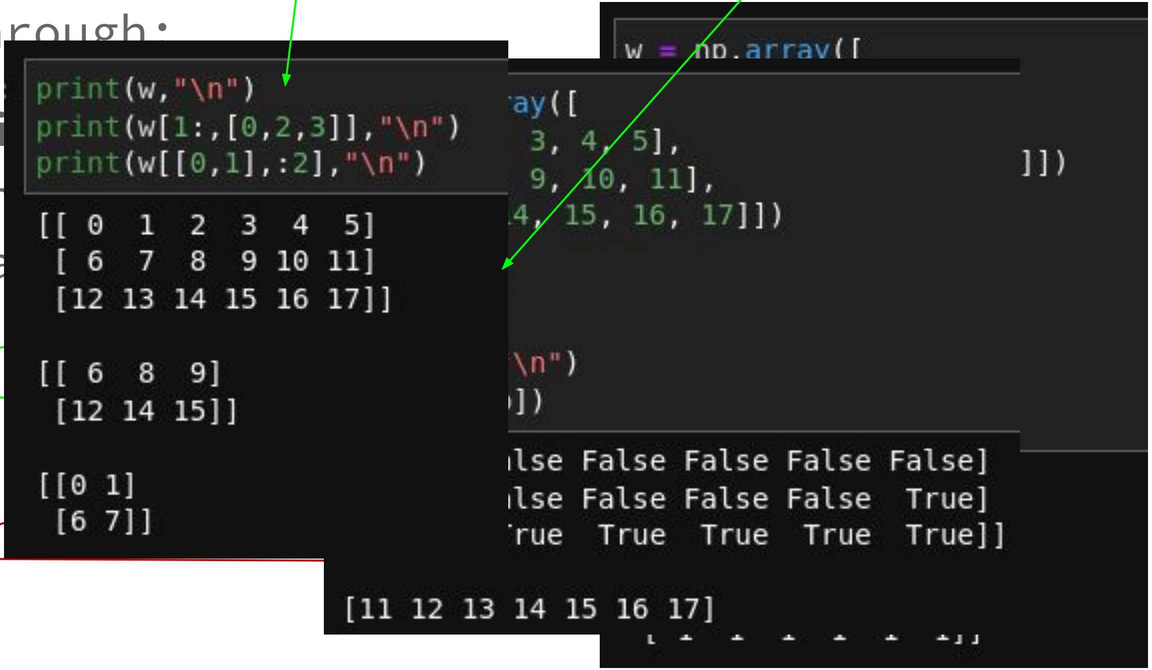
- **A boolean mask:** thi
  powerful as you can
  satisfying a boolea

This selects all elements satisfying the condition

We can directly change elements that satisfy the condition into the array

Combination with slicing

This operation creates a boolean mask



NumPy- Array Math

Basic mathematical operation are implemented elementwise on arrays, i.e. operate on each element separately (**element-wise operation**):

- We can perform operations with scalars, they act on each array element

```
a = np.array([1, 2, 3, 4])

a + 1 # addition of a scalar

array([2, 3, 4, 5])

a - 2 # subtraction of a scalar

array([-1,  0,  1,  2])

3*a # multiplication by a scalar

array([ 3,  6,  9, 12])

2**a # array as power

array([ 2,  4,  8, 16])

a/2 # division by a scalar

array([0.5, 1. , 1.5, 2. ])
```

NumPy- Array Math

Basic mathematical operation are implemented elementwise on arrays, i.e. operate on each element separately (**element-wise operation**):

- We can also use arithmetic operations with arrays, they act element-wise on the elements. **BEWARE:** arrays must have the **compatible** shapes (see next)

```
a = np.array([1, 2, 3, 4])
b = np.ones(4) + 1

a-b # subtraction

array([-1.,  0.,  1.,  2.])

a*b # multiplication

array([2., 4., 6., 8.])

a += 1 # self-increment
print(a)

[2 3 4 5]

2**(a+1) - a

array([ 6, 13, 28, 59])
```

NumPy- Array Math

Basic mathematical operation are implemented elementwise on arrays, i.e. operate on each element separately (**element-wise operation**):

- And comparison operator as well

Boolean array, the = operator acts element-wise

Some functions operate array-wise, returning a single result!

```
a = np.array([1, 2, 3, 4])
b = np.array([4, 2, 2, 4])

a == b

array([False,  True, False,  True])

a > b

array([False, False,  True, False])

c = np.array([1, 2, 3, 4])

np.array_equal(a,b)

False

np.array_equal(a,c)

True
```

NumPy- Ufunc

Further to **ndarray**, *Numpy* defines a collection of universal functions known as **ufunc** that allow to operate element by element without any explicit loop. Usually such **ufunc** are wrapper to very efficient C/Fortran functions.

Some of them just overload the arithmetic operators, e.g.

np.multiply(x,y) <--------> x*y

Other implements trigonometric, exponential etc. functions

np.sin(x), np.exp(x)

or comparison functions (all belonging to **np** namespace)

greater, less, equal, logical_and/_or/_xor/_nor, maximum, minimum

NumPy- Ufunc

Further to **ndarray**, *Numpy* defines a collection of universal functions known as **ufunc** that allow to operate element by element without any explicit loop. Usually such **ufunc** are wrapper to very efficient C/Fortran functions.

Some of them just overload the arithmetic operators, e.g.

np.multiply(x,y) <--------> x*y

Other implements trigonometric, exponential etc. functions

np.sin(x), np.ex

or comparison functions (all belong

greater, less, equal, logical_and/
minimum

```
a = np.array([1, 2, 3, 4])
b = np.array([4, 2, 2, 4])
np.maximum(a,b)
```

```
array([4, 2, 3, 4])
```

NumPy- Broadcasting

Since the various operators act element-wise, arrays must have the same **shape**!

```
a = np.arange(4)
a + np.array([1,2])

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-71-cf0a6e60a4f5> in <module>
      1 a = np.arange(4)
----> 2 a + np.array([1,2])

ValueError: operands could not be broadcast together with shapes (4,) (2,)
```

NumPy- Broadcasting

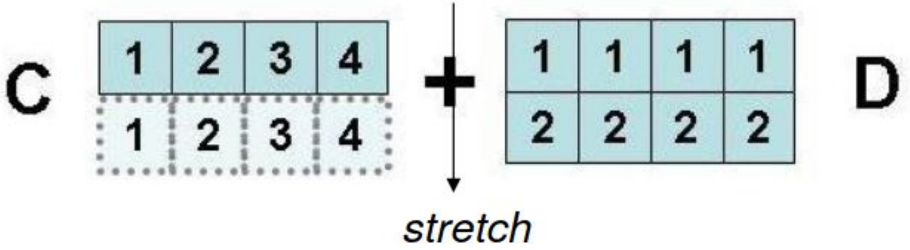
Since the various operators act element-wise, arrays must have the same **shape**!

Nevertheless, it is possible to work with arrays of different **dimensions**:

```
c=np.arange(1,5)
d=np.array([[1,1,1,1],[2,2,2,2]])

print(c)
print()
print(d)
print("\n",d+c)
```

```
[1 2 3 4]

[[1 1 1 1]
 [2 2 2 2]]

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

One component of shape must be in common!
Here **d.shape=(2,4)**, *while* **c.shape=(4,)**



NumPy- Broadcasting

Since the various operators act element-wise, arrays must have the same **shape**!

Nevertheless, it is possible to work with arrays of different **dimensions**. As a general rule:

- If the arrays don't have the same rank then prepend the shape of the lower rank array with 1s until both shapes have the same length.
- The two arrays are compatible in a dimension if they have the same size in the dimension or if one of the arrays has size 1 in that dimension.
- The arrays can be broadcast together iff they are compatible with all dimensions.
- After broadcasting, each array behaves as if it had shape equal to the element-wise maximum of shapes of the two input arrays.
- In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension.

NumPy- Broadcasting

Since the various operators act element-wise, arrays must have the same **shape**!

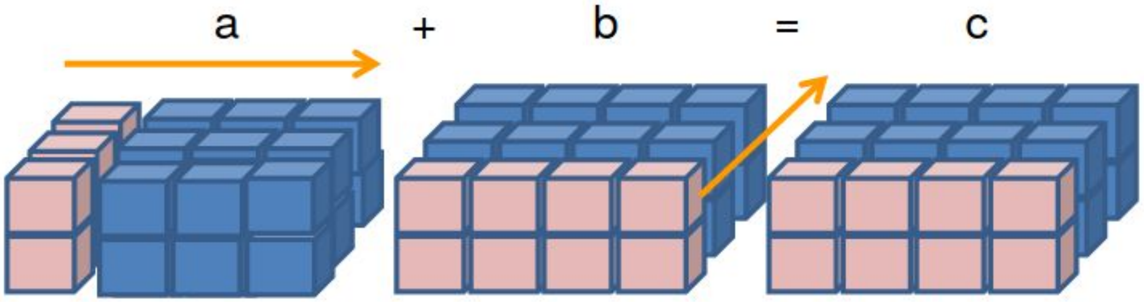
Note that one of the dimension is 1!

```python
a=np.arange(6)
a=a.reshape((2,1,3))
print(a)
print()

b=np.arange(8)
b=b.reshape((2,4,1))
print(b)
```

```
[[[0 1 2]]

 [[3 4 5]]]

[[[0]
  [1]
  [2]
  [3]]

 [[4]
  [5]
  [6]
  [7]]]
```

```python
c=a+b
print(c,c.shape)
```

```
[[[ 0  1  2]
  [ 1  2  3]
  [ 2  3  4]
  [ 3  4  5]]

 [[ 7  8  9]
  [ 8  9 10]
  [ 9 10 11]
  [10 11 12]]] (2, 4, 3)
```



NumPy- Array performance

One of the great advantage of *Numpy* concerns the efficiency of computation!

By avoiding explicit loops, your code will run much faster!

```
def for_array(a):
    for i in range(a.shape[0]):
        for j in range(a.shape[1]):
            a[i,j]=3*a[i,j]+1

    return a

def no_loop(a):
    a=a*3+1
    return a
```

```
a = np.ones((1000,1000))
```

```
%%time
for_array(a)
```

```
CPU times: user 680 ms, sys: 1.98 ms, total: 682 ms
Wall time: 681 ms

array([[13., 13., 13., ..., 13., 13., 13.],
       [13., 13., 13., ..., 13., 13., 13.],
       [13., 13., 13., ..., 13., 13., 13.],
       ...,
       [13., 13., 13., ..., 13., 13., 13.],
       [13., 13., 13., ..., 13., 13., 13.],
       [13., 13., 13., ..., 13., 13., 13.]])
```

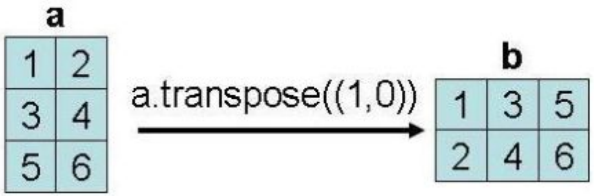
```
%%time
no_loop(a)
```

```
CPU times: user 2.96 ms, sys: 288 µs, total: 3.25 ms
Wall time: 2.22 ms

array([[13., 13., 13., ..., 13., 13., 13.],
       [13., 13., 13., ..., 13., 13., 13.],
       [13., 13., 13., ..., 13., 13., 13.],
       ...,
       [13., 13., 13., ..., 13., 13., 13.],
       [13., 13., 13., ..., 13., 13., 13.],
       [13., 13., 13., ..., 13., 13., 13.]])
```

NumPy- Additional functions

You can find a large number of *Numpy* function acting on arrays, the best way to understand them is to look at the documentation (https://numpy.org/doc/stable/index.html).

- **fill(value)** -> an array is filled with value;
- **sort(axis=-1, kind='quicksort', order=None)** -> the array is sorted in ascending order; default is to sort according to last axis!;
- **transpose(*axis)** -> implements transposition specified by axis.
- **dot(a,b)** -> scalar product between **a** and **b**, could be arrays or matrix



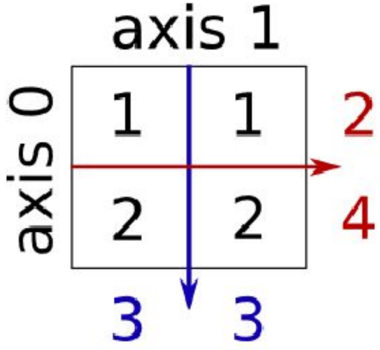
NumPy- Additional functions

You can find a large number of *Numpy* function acting on arrays, the best way to understand them is to look at the documentation (https://numpy.org/doc/stable/index.html).

- **sum(axis=None)** -> computes the sum of array elements;

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

print(x.sum(axis=0)) # sum by column

print(x.sum(axis=1)) # sum by row

print(x.sum()) # sum all elements
```

```
[3 3]
[2 4]
6
```



NumPy- Additional functions

You can find a large number of *Numpy* function acting on arrays, the best way to understand them is to look at the documentation (https://numpy.org/doc/stable/index.html).

- **sum(axis=None)** -> computes the sum of array elements;
- Statistics: **ndarray.mean(), ndarray.std(), ndarray.median()**, etc.;
- Extrema: **ndarray.max(), ndarray.min(), ndarray.argmax(), ndarray.armin()** (index of max or min):
- Logical: **ndarray.all(), ndarray.any()** if there are True in the array;

NumPy- Grid evaluation

Since *Numpy* operates element-wise, it could happen that you get a result that you do not expect. X and Y are multiplied element by element. This is not a grid evaluation though!

```
x = np.arange(4)
y = np.arange(4)
def f(x,y):
    return x**2+y
```

```
f(x,y)
```

```
array([ 0,  2,  6, 12])
```

NumPy- Grid evaluation

Since *Numpy* operates element-wise, it could happen that you get a result that you do not expect. X and Y are multiplied element by element. <u>This is not a grid evaluation though</u>!

```
x = np.arange(4)
y = np.arange(4)
def f(x,y):
    return x**2+y
```

```
f(x,y)
```

```
array([ 0,  2,  6, 12])
```

The function meshgrid does the job for you and evaluates the function on the grid!

```
x = np.arange(4)
y = np.arange(4)
xx, yy = np.meshgrid(x,y)

f(xx,yy)
```

```
array([[ 0,  1,  4,  9],
       [ 1,  2,  5, 10],
       [ 2,  3,  6, 11],
       [ 3,  4,  7, 12]])
```

NumPy- Input/Output

Numpy provides simple functions to read and write files, specifically **loadtxt** and **savetxt**. This functions are alternative to the built-in function **open**.

- **numpy.loadtxt(**fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0, encoding='bytes', max_rows=None, *, like=None**)**
- **numpy.savetxt(**fname, X, fmt='%.18e', delimiter=' ', newline='\n', header='', footer='', comments='# ', encoding=None**)**

NumPy- Input/Output

Numpy provides simple functions to read and write files, specifically **loadtxt** and **savetxt**. This functions are alternative to the built-in function **open**. The convenience of *Numpy* functions is that they automatically deal with arrays.

Name of the file, usually a string

The dtype of data in the file

- **numpy.loadtxt(**fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0, encoding='bytes', max_rows=None, *, like=None**)**

Specify if a line can be skipped if marked

Skip lines

Read only selected columns

If True, the returned array is transposed, so that arguments may be unpacked using x, y, z = loadtxt(...)
i.e. each column is stored into an array

NumPy- Input/Output

Numpy provides simple functions to read and write files, specifically **loadtxt** and **savetxt**. This functions are alternative to the built-in function **open**. The convenience of *Numpy* functions is that they automatically deal with arrays.

Name of the file, usually a string, but can be a file object

Data to be saved, 1D or 2D array_like

- **numpy.savetxt(**fname, X, fmt='%.18e', delimiter=' ', newline='\n', header='', footer='', comments='# ', encoding=None)

Data format

Delimiter between data