Introduction to Python for Data Analysis

How to manipulate and represent data with python

Romain Madar, DU Data scientist

Laboratoire de Physique de Clermont-Ferrand (UCA, CNRS/IN2P3) – FRANCE

Contact: romain.madar@clermont.in2p3.fr

August 2, 2019

Contents

Preamble

These notes contain the material for lecture of a data scientist university degree proposed at Université Clermont-Auvergne (UCA), which is mostly based on a NumPy tutorial that I gave at the Laboratoire de Physique de Clermont in April 2019. No prerequisite knowledge is assumed but being familiar with one progamming language might be useful. It is better to know about some basic mathemtatics like simple vectorial operation or statistics. All the material of this lecture can be found in this github repository.

Python offers a rapidly evolving ecosystem to perform data analysis and it is both out of scope and hopeless to be extensive in this lecture. The main goal is therefore to make people familiar with teh basic of python and data analysis tools in order to *make them able to extend their knowledge on themself*. Object oriented programming is not presented in this lecture. Pratical exercises are also available to provide few working examples as a starting point.

This lecture is only a support to help you doing things *yourself*. As any other language, you must practice it if you want to progress. If you don't write and test code on your own, this lecture is close to be useless. I am available for any questions or general feedback on this lecture, so feel free to use my e-mail: romain.madar@clermont.in2p3.fr.

Chapter 1

Brief introduction to Python

1.1 General information

Python and jupyter-notebook can be installed using anaconda, this is probably the easiest way to follow this lecture and make your own notes. The goal of this first chapter is to give a *very quick introduction basis*, but practice is mandatory to get confortable with python objects and synthax. Practicing is possible with a web browser only using the www3school python tutorials. I recommand to follow these tutorials up to *Arrays*.

In python, there is one instruction per line. Variable assignment is done with with =, indentation is used to group instructions together under a loop or a condition block: there is no backet (like in C++) or equivalent. Comments (*i.e.* uninterpreted text) start with #. Importation of external modules or fonction can be done with three different ways: `import module`, `import module as m` or `from module import this_function`.

In the following example, the result of the command will be printed so that people can check that the computer is doing what is expected. The instruction `print(x)` will print the content of `x`. If several variables are printed, is it convenient to use `print('x={} and y={}'.format(x, y))` synthax that will print `x` and `y` in bracket fields with one command - even if they have different types.

1.2 Object types

Numbers. There are three type of numbers: int, float and complex. The usual operations (+, -, *, /) are available. In addition, there is also `a**b` (which means a^b), `a // b` and `a % b` (which are the result of integer divisions - see example below).

```
# Basic numbers and operations
a = 2
b = 3.14
print(a+b)
print(a**b)
```

```
5.140000000000001
8.815240927012887
```

```
# Complex numbers and power
a = 1j
a**2
```

```
(-1+0j)
```

```
# Integer division example (// and % operators)
a , b = 10, 4
divisor, rest = a//b, a%b
print('{} = {}x{} + {}'.format(a, divisor, b, rest))
```

```
10 = 2x4 + 2
```

Strings. String allows to manipulate words, sentence or even text with specific methods. String are also python lists and list methods work as well (see below). The common and useful string manipulations can be counting the number of letters with `len(word)` or even manipulate a collection of words using `sentence.split(' ')`. Many methods exist, which can be looked at by typing `help(str)` in a python terminal or a jupyter nootebook.

```
w1 = 'hello'
print(w1, len(w1), w1[3])
```

```
hello 5 l
```

```
# Summing two strings is possible (all other operators dont work)
blank, w2 = ' ', 'world'
sentence = w1 + blank + w2
print(sentence)
```

```
hello world
```

```
# Multiplying a string by an integer is also possible
repetition = sentence*3
print(repetition)
```

```
hello worldhello worldhello world
```

```
# Get a list of words from a sentence (cf. below for list objects)
s = 'It is rainy today'
list_words = s.split(' ')
print(list_words)
```

```
['It', 'is', 'rainy', 'today']
```

```
# Looping over the words and get the number of letters
for w in s.split(' '):
    print(w, len(w))
```

```
It 2
is 2
rainy 5
today 5
```

1.3 Object collections

There are four types of collection, which share several methods but differ from various aspects:

- list
- dictionnary
- tuple
- set

The most commonly used are the lists and dictionnary. The specificy of the set is that it is unordered, while the specificyt of the tuple is that it cannot be modified. The common methods are

- number of items: `len(x)`
- loop over items with `for element in x:`
- check if a item is in the list: `element in x`

Lists. This is one of the most used collection object in python because it is the next-to-simplest level, after individual variables. A python list is a *list of objects* with possibly different types. One can search, loop, count with list. One can also add two lists or multiply a list by an integer, which makes a *concatenation* or a *duplication* (these points will be important for numpy arrays). The indexing of elements is also a nice way to access the information of interest: one can access the i^{th} element with `my_list[i]` or get a sub-list with `my_list[i:j]`. One can also take only one element every `n` with `my_list[i:j:n]` (more precisely this takes elements of index $i + p \times n$ until j, with $p = 0, 1, 2, ...$). With this synthax, reverting the order of the list is easy: `reverted_list = my_list[::-1]`, where empty variable are default values (namely `0` and `len(my_list)`).

```
# Defining a list and access basic information
my_list1 = [1, 3, 4, 'banana']
print('Second element is {}'.format(my_list1[1]))
print('Number of elements: {}'.format(len(my_list1)))
print('Is \'banana\' in the list? {}'.format('banana' in my_list1))
```

```
Second element is 3
Number of elements: 4
Is 'banana' in the list? True
```

```
# Sum of two lists
my_list2 = ['string', 1+3j, [100, 1000]]
my_list = my_list1 + my_list2
print(my_list)
```

```
[1, 3, 4, 'banana', 'string', (1+3j), [100, 1000]]
```

```
# List multiplied by an integer
my_list = my_list*2
print(my_list)
```

```
[1, 3, 4, 'banana', 'string', (1+3j), [100, 1000], 1, 3, 4, 'banana', 'string', (1+3j), [100,
1000]]
```

```
# Looping over list element and print the type of seven first elements in the reversed order.
for element in my_list[6:0:-1]:
    print('{} is {}'.format(element, type(element)))
```

```
[100, 1000] is <class 'list'>
(1+3j) is <class 'complex'>
string is <class 'str'>
banana is <class 'str'>
4 is <class 'int'>
3 is <class 'int'>
```

sets and tuples. Tuples and sets are modified version of python list. Tuples are ordered but cannot be modified (no assignment, no addition, while sets are not ordered but can be mofided. In this context, order means indexing (so `x[i:j:n]` synthax, among others). Search or loop over elements work in the same way as for list.

```
# Tuple
t = (1, 3, 7)
print(t)

# Access the third element
print(t[2])

# Try to modify the second element - using the 'try - except' synthax
try:
    t[1] = 'hello'
except TypeError:
    print('Impossible to change the value of a tuple')
```

```
(1, 3, 7)
7
Impossible to change the value of a tuple
```

Sets can modified with methods like `s.add(x)` or `s.update([x, y])`.

```
# Set
s = {'apple', 'banana', 'orange'}
print(s)

# Add one element
s.add('pineapple')
print(s)

# Add a list
s.update(['pear', 'prune'])
print(s)
```

```
{'orange', 'banana', 'apple'}
{'pineapple', 'orange', 'banana', 'apple'}
{'orange', 'pineapple', 'apple', 'pear', 'prune', 'banana'}
```

```
# Try to access the second element - using the 'try - except' synthax
try:
    print(s[1])
except TypeError:
    print('Impossible to access element via indexing')
```

```
Impossible to access element via indexing
```

Dictionnaries. Various object types are important to manipulate and organize data. The most common one is the dictionnary which work with a pair of (`key`, `value`). The `key` must be a non-modifiable object, in practice string or integer, but cannot be a list. This is a very powerful concept to store different types of information into the same object. One can easily loop, search, modify a given key value, or even add a new key quite easily.

```
# dictionnary
person = {'name': 'Charles', 'age': 78, 'size': 173, 'gender': 'M'}
print(person)
```

```
{'name': 'Charles', 'age': 78, 'size': 173, 'gender': 'M'}
```

```
# Accessing value using the key
template = '{} ({}) is {} years old and is {} cm'
print(template.format(person['name'], person['gender'], person['age'], person['size']))
```

```
Charles (M) is 78 years old and is 173 cm
```

```
# Adding a key and its value
person['eyes'] = 'blue'
print(person)
```

```
{'name': 'Charles', 'age': 78, 'size': 173, 'gender': 'M', 'eyes': 'blue'}
```

```
# Test if a key is present
print('name' in person)
print('brand' in person)
```

```
True
False
```

1.4 Loops

Loops are at the core of programming and especially for data analysis oriented tasks. There are two way of repeating a instruction several times: the `for` loop and the `while` loop. Several instructions are common to both loops, such as `continue` (skip instruction after and switch to the next element) or `break` (stop the loop), but the use case of these two ways are different.

For loops. For data analysis, I think these are the most used ones. But as we will see in the introduction to numpy, for loops must not be used for heavy computations in python. For loops are relevant for small (~1000) data samples (and computations). We'll come back to this point in the lecture. Below, few example are given.

```
# Compute sum(i^2) for i from 0 to 9
x = 0
for i in range(0, 10):
    x += i**2
print(x)
```

```
285
```

```
# Loop over fruit via a set and print only ones with a 'p'
for fruit in s:
    if 'p' in fruit:
        print(fruit)
```

```
pineapple
apple
pear
prune
```

There are several ways to loop over dictionnary depending on how we want to access the information. Indeed, you can access information by keys, values, or both. An example of each are given below.

```
# Loop over keys for a dictionnary and access the value of each
for properties in person:
    value = person[properties]
    print('{}: {}'.format(properties, value))
```

```
name: Charles
age: 78
size: 173
gender: M
eyes: blue
```

```
# Loop over dictionnary values only
for v in person.values():
    print(v)
```

```
Charles
78
173
M
blue
```

```
# Loop over both keys and values directly
for key, value in person.items():
    print('{}: {}'.format(key, value))
```

```
name: Charles
age: 78
size: 173
gender: M
eyes: blue
```

Tip: how to sort a dictionnary? It an can be noted that dictionnary are natively *not ordered*. This means that you cannot access a item with its index, since there is no index. However, it can be convenient to sort dictionnary keys according ot their values, using the general python function `sorted(collection, key=function)` and a type of object called `OrderDict` from `collections` module, as explained below.

```
# Define a dictionnary
students_marks = {
    'Jean': 12,
    'Chloe': 17,
    'Olivier': 8,
    'Helene': 10
    }

# Print the key, values items
for name, mark in students_marks.items():
    print(name, mark)
```

```
Jean 12
Chloe 17
Olivier 8
Helene 10
```

The `sorted()` function works on any type of collection than can be looped over (called *iterable*). It needs the collection and the function which return a key on which to sort for each object of the collection. This can be for *e.g.* a letter or a number. Let's try both by defining a function getting either the mark or the first letter of the name, from a dictionnary item `(name, mark)`.

```
# Order it by increasing marks
def get_mark(item):
    return item[1]

# Or by the first letter of the name
def get_name(item):
    return item[0][0]

# Testing with an item
item_test = ('Jacques', 12)
print('{} has a mark of {} and {} as 1st letter'.format(item_test, get_mark(item_test),
↪  get_name(item_test)))
```

```
('Jacques', 12) has a mark of 12 and J as 1st letter
```

We can now apply the `sorted()` function to the collection of items of the initial directory. This will return a collection of sorted items that can be later converted into a dictionnary. This last step depends on python version (in version 2.5, one has to use `OrderDict` from `collections` module while it's not needed in python 3 - the version of python can be dynamically checked with `sys.version_info` from `sys` module).

```
# Get all items and sort them by increasing mark
all_items = students_marks.items()
items_sorted_by_marks = sorted(all_items, key=get_mark)
items_sorted_by_names = sorted(all_items, key=get_name)
```

```
# Check the version of python
import sys
version = sys.version_info[0]
isPython2 = version == 2
```

```
# Final conversion of collection of items into a dictionnary
if isPython2:
    from collections import OrderedDict
    marks_sorted_dict = OrderedDict(items_sorted_by_marks)
    names_sorted_dict = OrderedDict(items_sorted_by_names)

else:
    marks_sorted_dict = {k: v for k, v in items_sorted_by_marks}
    names_sorted_dict = {k: v for k, v in items_sorted_by_names}
```

```
# Check the mark sorted results:
for k, v in marks_sorted_dict.items():
    print(k, v)
```

```
Olivier 8
Helene 10
Jean 12
Chloe 17
```

```
# Check the name sorted results:
for k, v in names_sorted_dict.items():
    print(k, v)
```

```
Chloe 17
Helene 10
Jean 12
Olivier 8
```

While loops. They are bit less used in practive but they are quickly discribed for completness. The idea is to repeat a given instruction until a condition is reached.

```
# Cast (ie change type) the set s into a list
my_list = list(s)

# Remove item one by one until there are no items left.
while len(my_list)>0:
    my_list.pop()
    print(my_list)
```

```
['orange', 'pineapple', 'apple', 'pear', 'prune']
['orange', 'pineapple', 'apple', 'pear']
['orange', 'pineapple', 'apple']
['orange', 'pineapple']
['orange']
[]
```

1.5 Few python synthax tips

Comprehension. This is the action of building a collection with one line of code. The comprehension syntax work for all collections, with conditions, or even nested loops (loops of loops). Few examples are given below.

```
# List
list_squares = [i**2 for i in range(1, 10)]
print(list_squares)
```

```
# Dictionnary
dict_squares = {i:i**2 for i in list_squares[0:5]}
print(dict_squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
{1: 1, 4: 16, 9: 81, 16: 256, 25: 625}
```

```
# Comprehension list with a condition (e.g. keep only even numbers)
list_even = [i for i in range(0, 10) if i%2==0]
print(list_even)
```

```
[0, 2, 4, 6, 8]
```

```
# Comprehension with nested loops
sum_integers = [i*10+j for i in range(0,5) for j in range(0, 5)]
print(sum_integers)
```

```
[0, 1, 2, 3, 4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31, 32, 33, 34, 40, 41, 42, 43,
44]
```

Looping with `enumerate` and `zip`.

The keywork `enumerate` return directly a counter together with the element arising in the loop. This is useful if you need to count the number of iterations of the loop. This can be done without `enumerate` but you need to add two lines (initialisation of the counter, and incrementation).

```
# Position of each word in a sentence
sentence = 'I would like to analyse this sentence in term of word position'
words = sentence.split(' ')
for i, w in enumerate(words):
    print(w.ljust(10) + ':  ' + str(i))
```

```
I         :  0
would     :  1
like      :  2
to        :  3
analyse   :  4
this      :  5
sentence  :  6
in        :  7
term      :  8
of        :  9
word      :  10
position  :  11
```

The `zip(list1, list2)` synthax allows to form pairs using elements of each list at *the same position*. This is quite convenient to associate some objects which are stored in different collections in a very quick and

readable way. If `list1` and `list2` don't have the same size, the mimimum of the two lenght is taken. Finally, the `zip()` command can take more than two collections and return then a group of element which has the size of the number of collection.

```
# Associate fruits and colors
fruits = ['banana', 'orange', 'pineapple', 'pear', 'prune']
colors = ['yellow', 'orange', 'brown', 'green', 'purple']
for f, c in zip(fruits, colors):
    print('{} is {}'.format(f, c))
```

```
banana is yellow
orange is orange
pineapple is brown
pear is green
prune is purple
```

```
# Using zip() with three lists
l1, l2, l3 = range(0, 10), range(0, 100, 10), range(0, 1000, 100)
for i, j, k in zip(l1, l2, l3):
    print(i, j, k, i+j+k)
```

```
0 0 0 0
1 10 100 111
2 20 200 222
3 30 300 333
4 40 400 444
5 50 500 555
6 60 600 666
7 70 700 777
8 80 800 888
9 90 900 999
```

1.6 Functions

Functions are defined as a set of instruction encapsulated into one object. This is particulary convenient when one has to the same list of instructions several times. A good guideline to know when to write a function could be

> If you copy-paste the same pieace of code more than two times, then make a function

Definition. A function takes some arguments, perform some instruction an return a result. The syntax to define and call a function is showed below. In python, function must be defined *before* being used (as opposed to C++ where it can be different, as soon as the function is declared). This makes the concept of *package* quite relevant to wrapp-up several function into a python file which can be imported in the main code.

```
# Definition syntax
def function(argument):
    result = argument * 3
    return result

# Call syntax
function(2)
```

```
6
```

The type of the arguement is not fixed (since it is a general feature of python) so the same instruction will be interpreted differently depending on the type. The following example shows the different result of the function above for two argument types.

```
# Print the result for two types of arguements
print('function(10) = {}'.format(function(10)))
print('function(\'ouh\') = {}'.format(function('ouh')))
```

```
function(10) = 30
function('ouh') = ouhouhouh
```

```
def person_printer(p):
    template = '{} ({}) is {} years old and is {} cm'
    print(template.format(p['name'], p['gender'], p['age'], p['size']))
    return


def grow_old(p, n_years):

    # 1. copy the dictionnary person (otherwise p *will* be modified)
    res = p.copy()

    # 2. Compute the new age and size
    new_age = res['age'] + n_years
    new_size = res['size'] - n_years*0.13

    # 3. Assign the new age/size to the result
    res['age'] = new_age
    res['size'] = new_size

    # 4. Return the result
    return res
```

```
# Print before growing old
person_printer(person)

# Growing old
```

```
old_guy = grow_old(person, 10)

# Print after growing old
person_printer(old_guy)
```

```
Charles (M) is 78 years old and is 173 cm
Charles (M) is 88 years old and is 171.7 cm
```

Docstring.

This offers the possibility to document your code in a proper way, which is quite useful for others (and for you, when you will re-use a code after several years). This is then a good practice to do, even if it takes time. This can be accessed using the command `help(function)` or by using the keyboard shortcut `Shift+Tab` in jupyter notebook (when the cursor is after the opening parenthesis of the function). The syntax to add docstring is `'''My documentation'''` at the very begining of the function.

```
def grow_old(p, n_years):
    '''
    Take a person dictionnary and update the age and the size to make the person older.

    Parameters
    ----------
    p: dictionnary
        Person object as defined earlier in the code, with at least 'age' (year)
        and 'size' (cm) keys, to get old.
    n_years: integer
        Number of years to be added to the age of the person.

    Return
    ------
    person: dictionnary
       Person object as defined earlier in the code with age and size updated as
         age  -> age+n_years
         size -> size-n_years*0.13
    '''

    # 1. copy the dictionnary person (otherwise p will be modified, which might problematic)
    res = p.copy()

    # 2. Compute the new age and size
    new_age = res['age'] + n_years
    new_size = res['size'] - n_years*0.13

    # 3. Assign the new age/size to the result
    res['age'] = new_age
    res['size'] = new_size

    # 4. Return the result
    return res
```

```
help(grow_old)
```

```
Help on function grow_old in module __main__:

grow_old(p, n_years)
    Take a person dictionnary and update the age and the size to make the person older.

    Parameters
    -----
    p: dictionnary
        Person object as defined earlier in the code, with at least 'age' (year)
        and 'size' (cm) keys, to get old.
    n_years: integer
        Number of years to be added to the age of the person.

    Return
    ---
    person: dictionnary
       Person object as defined earlier in the code with age and size updated as
         age  -> age+n_years
         size -> size-n_years*0.13
```

There are several ways to organise the docstring and the example above is based on numpy docstring style. Note that docstring can be also added to a module (in practice, a python file) to document the content, goal and usage of this module.

Arbitrary number of arguments: `*args` **and** `**kwargs`. The example above are relatively simple and generally function takes several arguments. Sometime it is even convenient to have an unfixed number of arguments, so that the function is rather evolutive when the code grows. Python offres a nice way to define such a function thanks to the *packing* and *unpacking* notion, which is describe right below.

Apparte: packing and unpacking. In short, this is the possibity to convert a list into a serie of objects (unpacking) or vis-versa (packing). This way of writing collection makes code developments very consise and fast, especially to call function with a several arguments in a nice way. This also allows to define function with an arbitrary number of arguments as already mentioned. The following dummy function is used to illustrate the concept of packing/unpacking with both a list and a dictionnary.

```
# Test function
def mean(a, b, c):
    return (a+b+c)/3.
```

It is possible to use a list of three number to specify the argument values of the `mean(a, b, c)` function, using the unpacking syntax for list: `*list`. This is demonstrated below:

```
# Packing & unpacking with a list (or a tuple): *list
my_numbers = [10, 12, 15]
mean(*my_numbers)
```

```
12.333333333333334
```

This is also sometime convenient to call the argument by their name (mostly to make the code more readable). This type of arguments are called *keyword arguments* and can be packed/unpacked into a dictionnary. Each argument name is a key of this dictionnary and the value is the values passed to the function. The unpacking is done with `**dict`.

```
# Packing & unpacking with a dictionnary: **dict
my_numbers = {'a': 10, 'b': 12, 'c': 15}
mean(**my_numbers)
```

```
12.333333333333334
```

Coming back to the initial motivation, *i.e.* having an arbitrary number of arugments. It is possible to define such a function as follow - which in that case just print number and the list of arguments:

```
# Function definition with *args
def test_function(*args):
    print('There are {} arguments: '.format(len(args)))
    for a in args:
        print('  -> {}'.format(a))
    print('')
    return
```

```
# Test with different numbers/types of arguments
test_function()
test_function('hoho')
test_function('hoho', 3)
test_function('hoho', 3, [1,'banana'], {'mood': 'happy', 'state': 'holidays'})
```

```
There are 0 arguments:

There are 1 arguments:
  -> hoho

There are 2 arguments:
  -> hoho
  -> 3

There are 4 arguments:
  -> hoho
  -> 3
  -> [1, 'banana']
  -> {'mood': 'happy', 'state': 'holidays'}
```

```
# Function definition with kwargs
def test_function(**kwargs):
    print('There are {} arguments: '.format(len(kwargs)))
    for k, v in kwargs.items():
        print(' {}={}'.format(k, v))
    print('')
    return
```

```
test_function()
test_function(x='hoho')
test_function(word='hoho', multiplicity=3)
test_function(a='hoho', N=3, shopping=[1,'banana'], feeling={'mood': 'happy', 'state':
↪  'holidays'})
```

```
There are 0 arguments:

There are 1 arguments:
 x=hoho

There are 2 arguments:
 word=hoho
 multiplicity=3

There are 4 arguments:
 a=hoho
 N=3
 shopping=[1, 'banana']
 feeling={'mood': 'happy', 'state': 'holidays'}
```

This can be used to declare argument in a very readable and concise way. This might be helpful for some cosmetic argument of plot that can be common to several plots (but not all). We'll see some concrete example later in the lecture. In the meanwhile, here is the equivalent of last call from the code above:

```
# Pack all keyword arguments in a dictionnary first
my_args = {
           'a': 'hoho',
           'N': 3,
           'shopping': [1,'banana'],
           'feeling': {'mood': 'happy', 'state': 'holidays'}
          }

# Then call the function
test_function(**my_args)
```

```
There are 4 arguments:
 a=hoho
 N=3
 shopping=[1, 'banana']
 feeling={'mood': 'happy', 'state': 'holidays'}
```

1.7 File manipulation

File handling is quite important since it enable interaction between your code and input/ouput data (called I/O). There are several features related to file handeling in python and this short section just give few basic practices.

Open/close a file. Python has native methods to open and close file. While closing a file doesn't allow for many variations, the opening can be done in different mode, depending if we want to read, write or append to the openned file. The basic syntax is:

```
# Open
f = open('my_file_name.txt', option)

# Close
f.close()
```

where option is a one letter string which can be: + `r` read (default): to just read the file + `a` append: to add content at the end of an existing file + `w` write: to write content in a file (it creates a new file) + `x` create: to create a new file

Write a file.

```
# Text to be written (can be one line string 'my text' or multiple lines string - docstring)
text = '''Gervaise avait attendu Lantier jusqu'à deux heures du matin. Puis,
toute frissonnante d'être restée en camisole à l'air vif de la fenêtre,
elle s'était assoupie, jetée en travers du lit, fiévreuse, les joues
trempées de larmes.
'''

# Open in write mode
f = open('test.txt', 'w')

# Write string
f.write(text)

# Close
f.close()
```

Read a file. The following example load the precedent file and loop over each line to analyse its content. One can note several issues: first one sentence can be on two lines, and second, each end of line containt a `\n` (which is an invisible caracter meaning "go-to-next-line"). There is a method to clean a line, called `line.strip()` and remove all spaces and invisible caracteres, unless specifed otherwise - see `help(str.strip)`.

```
# Open the file in read mode
f = open('test.txt', 'r')

# Loop over the lines
for line in f:
```

```
    # Print a header to make the ouput clearer
    print('\n\n{}'.format('='*50))

    # Print the line as it is given
    print('This line: {}'.format(line))

    # Split by '.' to isolate sentence
    sentences = line.split('.')
    print('This line has {} sentences: '.format(len(sentences)))

    # Split each sentence by ' ' to isolate words
    for i,s in enumerate(sentences):
        sclean = s.strip()
        words = sclean.split(' ')
        print(' - sentence {}: {}'.format(i, words))

f.close()
```

```
==================================================
This line: Gervaise avait attendu Lantier jusqu'à deux heures du matin. Puis,

This line has 2 sentences:
 - sentence 0: ['Gervaise', 'avait', 'attendu', 'Lantier', 'jusqu'à', 'deux', 'heures', 'du',
'matin']
 - sentence 1: ['Puis,']


==================================================
This line: toute frissonnante d'être restée en camisole à l'air vif de la fenêtre,

This line has 1 sentences:
 - sentence 0: ['toute', 'frissonnante', 'd'être', 'restée', 'en', 'camisole', 'à', 'l'air',
'vif', 'de', 'la', 'fenêtre,']


==================================================
This line: elle s'était assoupie, jetée en travers du lit, fiévreuse, les joues

This line has 1 sentences:
 - sentence 0: ['elle', 's'était', 'assoupie,', 'jetée', 'en', 'travers', 'du', 'lit,',
'fiévreuse,', 'les', 'joues']


==================================================
This line: trempées de larmes.

This line has 2 sentences:
 - sentence 0: ['trempées', 'de', 'larmes']
```

```
 - sentence 1: ['']
```

Re-write a modifed version of a file into a new file. It can be quite convenient to modify an existing file to correct a systematical mistake automatically, or simply do more complexe operation. The example below shows how to remove all the “e” from the text below and write it in a new file.

```
# Open the in/out files
f_i = open('test.txt', 'r')
f_o = open('test_without_e.txt', 'w')

# Loop over line, remove all "e" for each, and write the result in the output file
for line in f_i:
    line_without_e = line.replace('e', '') # replace "e" by nothing
    f_o.write(line_without_e)

# Close all files
f_i.close()
f_o.close()
```

```
# Open in read mode and check the result
f = open('test_without_e.txt', 'r')
print(f.read())
```

```
Grvais avait attndu Lantir jusqu'à dux hurs du matin. Puis,
tout frissonnant d'êtr rsté n camisol à l'air vif d la fnêtr,
ll s'était assoupi, jté n travrs du lit, fiévrus, ls jous
trmpés d larms.
```

Read a csv file to get data.

This use case is quite important since it allows to convert a file with data into variables accessible in the code (for some computation, plotting, etc ...). One the most basic format to store data is called *csv* (for comma-separated values) which can import/export from any spreadsheet software (like excel). This format is not necessarily appropriate for large dataset but is quite useful if a large number of situations. one must be able to manipulate it easily, as shown in the example below.

Creation of a csv file on the fly using a docstring

```
# Data taken from kaggle: https://www.kaggle.com/jolasa/waves-measuring-buoys-data-mooloolaba
data_csv_format = '''index,date,height,heightMax,period,energy,direction,temperature
1,01/01/2017 00:00,-99.9,-99.9,-99.9,-99.9,-99.9,-99.9
2,01/01/2017 00:30,0.875,1.39,4.421,4.506,-99.9,-99.9
3,01/01/2017 01:00,0.763,1.15,4.52,5.513,49,25.65
4,01/01/2017 01:30,0.77,1.41,4.582,5.647,75,25.5
5,01/01/2017 02:00,0.747,1.16,4.515,5.083,91,25.45
6,01/01/2017 02:30,0.718,1.61,4.614,6.181,68,25.45
7,01/01/2017 03:00,0.707,1.34,4.568,4.705,73,25.5
8,01/01/2017 03:30,0.729,1.21,4.786,4.484,63,25.5
```

```
9,01/01/2017 04:00,0.733,1.2,4.897,5.042,68,25.5
10,01/01/2017 04:30,0.711,1.29,5.019,8.439,66,25.5
11,01/01/2017 05:00,0.698,1.11,4.867,4.584,64,25.55
12,01/01/2017 05:30,0.686,1.14,4.755,5.211,56,25.55
13,01/01/2017 06:00,0.721,1.12,4.843,5.813,67,25.5
14,01/01/2017 06:30,0.679,1.22,4.948,4.71,81,25.45
15,01/01/2017 07:00,0.66,1.08,5.068,5.353,90,25.45
16,01/01/2017 07:30,0.662,1.18,5.263,7.436,67,25.4
17,01/01/2017 08:00,0.653,1.21,5.007,6.001,90,25.45
18,01/01/2017 08:30,0.665,1.17,4.952,6.414,90,25.55
19,01/01/2017 09:00,0.684,1.55,5.022,6.691,88,25.6
20,01/01/2017 09:30,0.679,1.09,4.926,6.804,88,25.65
21,01/01/2017 10:00,0.667,1.12,4.928,6.641,122,25.75
22,01/01/2017 10:30,0.688,1.13,4.808,5.958,91,25.7
23,01/01/2017 11:00,0.644,0.99,4.559,6.691,92,25.9
'''

# Create csv file using these data
f = open('wave_data.csv', 'w')
f.write(data_csv_format)
f.close()
```

Reading the csv file and storing values in python objects. In this example, we'll see how to store all information about the wave in a list of dictionnaries.

```
# Open the file in read mode
f = open('wave_data.csv', 'r')

# Get the first line (calling the readline() method once) to extract the feature names.
features = f.readline().strip().split(',')
print(features)

# Loop over lines and store the information
data = []
for l in f:
    values = l.strip().split(',')
    data_single_wave = {var: val for var, val in zip(features, values)}
    data.append(data_single_wave)
```

```
['index', 'date', 'height', 'heightMax', 'period', 'energy', 'direction', 'temperature']
```

```
# helper function for a nice printing
def print_wave(w):
    tmp = 'Wave {} ({}) had a heigh of {}m with a temperature of {} degree'
    print(tmp.format(w['index'], w['date'], w['height'], w['temperature']))

# Print the first 5 waves
```

```
for wave in data[:5]:
    print_wave(wave)
```

```
Wave 1 (01/01/2017 00:00) had a heigh of -99.9m with a temperature of -99.9 degree
Wave 2 (01/01/2017 00:30) had a heigh of 0.875m with a temperature of -99.9 degree
Wave 3 (01/01/2017 01:00) had a heigh of 0.763m with a temperature of 25.65 degree
Wave 4 (01/01/2017 01:30) had a heigh of 0.77m with a temperature of 25.5 degree
Wave 5 (01/01/2017 02:00) had a heigh of 0.747m with a temperature of 25.45 degree
```

At the stage, the problem is that the type of object which are stored is string and not numbers … so no computation can be made. Typically, the following code will crash because the division between string is not defined:

```
# Compute the average height
heights = [w['height'] for w in data]
average = sum(heights)/len(heights)
```

One has to cast (*i.e.* change type) the object stored into the dictionnaries. They can all casted as float but the date. The index makes more sense as integer as well. So the following can work:

```
# Manage string to time object conversion
def str_to_time(date_str):
    import datetime
    return datetime.datetime.strptime(date_str, '%d/%m/%Y %H:%M')

# Container with properly converted data
DATA = []

# Loop over waves and make the proper conversion depending on the feature name
for w in data:
    wgood = w.copy()
    for k in w:
        if k == 'index':   # Cast string into an integer
            wgood[k] = int(w[k])
        elif k == 'date':  # cast string into a datetime object
            wgood[k] = str_to_time(w[k])
        else:              # cast string into a float
            wgood[k] = float(w[k])
    DATA.append(wgood)
```

Computing the averaged heigh of wave now works but give a quite strange result:

```
# Compute the average height
heights = [w['height'] for w in DATA]
average = sum(heights)/len(heights)
print('Averaged waveheight is {:.1f} m'.format(average))
```

```
Averaged waveheight is -3.7 m
```

This is due to the first row which has all values at -99. Removing it (using indexing technics) gives a more sensible result:

```
heights = [w['height'] for w in DATA[1:]]
average = sum(heights)/len(heights)
print('Averaged waveheight is {:.1f} m'.format(average))
```

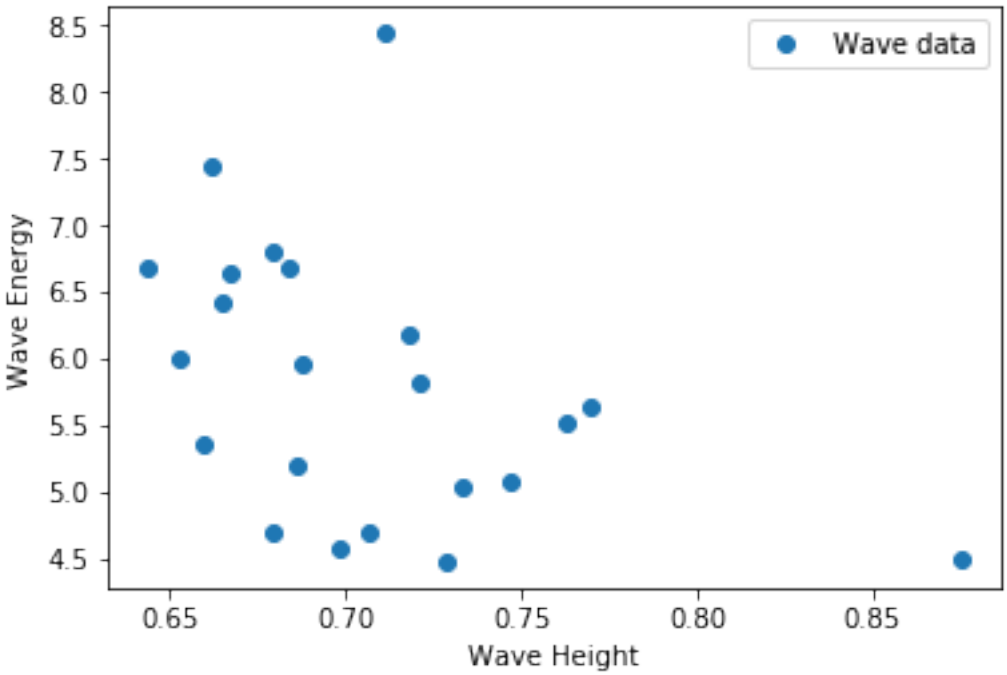
```
Averaged waveheight is 0.7 m
```

1.8 Plotting data: the very first step

Graphical representation of data is a key element of data analysis: it allows to get some intuition (and then ideas), or just perform visual checks. You might find the terminology "Exploratory Data Analysis (EDA)" in the litterature, which correspond to plot data in all possible way to *extract exploitable information* from data. The EDA is an entire field which will not cover in this lecture. We will simply gives some basic examples, which will provide a starting point to expand your knowledge. The standard library to produce plots is `matplotlib`, but it exist many other tools that we will not introduce (*e.g.* `seaborn`, `bokeh` for browser-interactive plots, `cartopy` for geographic data analysis/plots, etc.).

```
# Prepare data to plot: wave height v.s. wave energy (removing point with -99 values)
height = [w['height'] for w in DATA if w['height']>-99]
energy = [w['energy'] for w in DATA if w['energy']>-99]
```

```
# Import the key part of the package: pyplot
import matplotlib.pyplot as plt

# Display matplotlib output in the notebook
%matplotlib inline

# Call the simplest function to plot x vs y
plt.plot(height, energy, linewidth=0, marker='o', label='Wave data')

# Set x and y axis axis labels
plt.xlabel('Wave Height')
plt.ylabel('Wave Energy')

# Adding a legend based on label keyword
plt.legend();
```


Wave data
Wave Energy
Wave Height
8.5
8.0
7.5
7.0
6.5
6.0
5.5
5.0
4.5
0.65
0.70
0.75
0.80
0.85

Chapter 2

Basic introduction to NumPy

Why numpy? Numpy stands for *numerical python* and is highly optimized (and then fast) for computations in python. Numpy is one of the core package on which many others are based on, such as scipy (for *scientific python*), matplotlib or pandas (described at the end of this chapter). A lot of other scientific tools are also based on numpy and that justifies to have - at least - a basic understanding of how it works. Very well, but one could also ask why using python?

Why python? Depending on your preferences and your purposes, python can be a very good option or not (of course this is largely a matter of taste and not everyone agrees with this statement). In any case, many tools are available in python, scanning a very broad spectrum of applications, from machine learning to web design or string processing.

2.1 The core object: arrays

The core of numpy is the called numpy array. These objects allow to efficiently perform computations over large dataset in a very consise way from the language point of view, and very fast from the processing time point of view. The price to pay is to give up explicit *for* loops. This lead to somehow a counter intuitive logic - at first.

2.1.1 Main differences with usual python lists

The first point is to differenciate numpy array from python list, since they don't behave in the same way. Let's define two python lists and the two equivalent numpy arrays.

```
import numpy as np
l1, l2 = [1, 2, 3], [3, 4, 5]
a1, a2 = np.array([1, 2, 3]), np.array([3, 4, 5])
print(l1, l2)
```

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

First of all, all mathematical operations act element by element in a numpy array. For python list, the addition acts as a concatenation of the lists, and a multiplication by a scalar acts as a replication of the lists:

```
# obj1+obj2
print('python lists: {}'.format(l1+l2))
print('numpy arrays: {}'.format(a1+a2))
```

```
python lists: [1, 2, 3, 3, 4, 5]
numpy arrays: [4 6 8]
```

```
# obj*3
print('python list: {}'.format(l1*3))
print('numpy array: {}'.format(a1*3))
```

```
python list: [1, 2, 3, 1, 2, 3, 1, 2, 3]
numpy array: [3 6 9]
```

One other important difference is about the way to access element of an array, the so called slicing and indexing. Here the behaviour of python list and numpy arrays are closer expect that numpy array supports few more features, such as indexing by an array of integer (which doesn't work for python lists). Use cases of such indexing will be heavily illustrated in the next chapters.

```
# Indexing with an integer: obj[1]
print('python list: {}'.format(l1[1]))
print('numpy array: {}'.format(a1[1]))
```

```
python list: 2
numpy array: 2
```

```
# Indexing with a slicing: obj[slice(1,3)]
print('python list: {}'.format(l1[slice(1,3)]))
print('numpy array: {}'.format(l1[slice(1,3)]))
```

```
python list: [2, 3]
numpy array: [2, 3]
```

```
# Indexing with a list of integers: obj[[0,2]]
print('python list: IMPOSSIBLE')
print('numpy array: {}'.format(a1[[0,2]]))
```

```
python list: IMPOSSIBLE
numpy array: [1 3]
```

2.1.2 Main caracteristics of an array

The strenght of numpy array is to be multidimensional. This enables a description of a whole complex dataset into a single numpy array, on which one can do operations. In numpy, dimension are also called *axis*. For example, a set of 2 position in space $\vec{r}_i$ can be seen as 2D numpy array, with the first axis being the point $i = 1$ or $i = 2$, and the second axis being the coordinates (x, y, z). There are few attributes which describe multidimentional arrays:

- `a.dtype`: type of data contained in the array
- `a.shape`: number of elements along each dimension (or axis)
- `a.size`: total number of elements (product of `a.shape` elements)
- `a.ndim`: number of dimensions (or axis)

```
points = np.array([[ 0,  1, 2],
                   [ 3,  4, 5]])

print('a.dtype = {}'.format(points.dtype))
print('a.shape = {}'.format(points.shape))
print('a.size  = {}'.format(points.size))
print('a.ndim  = {}'.format(points.ndim))
```

```
a.dtype = int64
a.shape = (2, 3)
a.size  = 6
a.ndim  = 2
```

2.2 The three key features of NumPy

2.2.1 Vectorization

The *vectorization* is a way to make computations on numpy array **without explicit loops**, which are very slow in python. The idea of vectorization is to compute a given operation *element-wise* while the operation is called on the array itself. An example is given below to compute the inverse of 100000 numbers, both with explicit loop and vectorization.

```
a = np.random.randint(low=1, high=100, size=100000)

def explicit_loop_for_inverse(array):
    res = []
    for a in array:
        res.append(1./a)
    return np.array(res)
```

```
# Using explicit loop
%timeit explicit_loop_for_inverse(a)
```

```
161 ms ± 3.8 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
# Using list comprehension
%timeit [1./x for x in a]
```

```
145 ms ± 187 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
# Using vectorization
%timeit 1./a
```

```
138 µs ± 20.8 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

The suppression of explicit *for* loops is probably the most unfamiliar aspect of numpy - according to me - and deserves a bit a of practice. At the end, lines of codes becomes relatively short but ones need to properly think how to implement a given computation in a *pythonic way*.

Many standard functions are implemented in a vectorized way, they are call the *universal functions*, or `ufunc`. Few examples are given below but the full description can be found in numpy documentation.

```
a = np.random.randint(low=1, high=100, size=3)
print('a        : {}'.format(a))
print('a^2      : {}'.format(a**2))
print('a/(1-a^a): {}'.format(a/(1-a**a)))
print('cos(a)   : {}'.format(np.cos(a)))
print('exp(a)   : {}'.format(np.exp(a)))
```

```
a        : [97 46 74]
a^2      : [9409 2116 5476]
a/(1-a^a): [-2.11167384e-17 -8.97815296e-17  7.40000000e+01]
cos(a)   : [-0.92514754 -0.43217794  0.17171734]
exp(a)   : [1.33833472e+42 9.49611942e+19 1.37338298e+32]
```

All these ufunct can work for n-dimension arrays and can be used in a very flexible way depeding on the axis you are refering too. Indeed the mathematical operation can be performed over a different axis of the array, having a totally different meaning. Let's give a simple concrete example with a 2D array of shape (5,2), *i.e.* 5 vectors of three coordinates (x, y, z) Much more examples will be discussed in the section 2.

```
# Generate 5 vectors (x,y,z)
positions = np.random.randint(low=1, high=100, size=(5, 3))

# Average of the coordinate over the 5 observations
pos_mean = np.mean(positions, axis=0)
```

```
print('mean = {}'.format(pos_mean))

# Distance to the origin sqrt(x^2 + y^2 + z^2) for the 5 observations
distances = np.sqrt(np.sum(positions**2, axis=1))
print('distances = {}'.format(distances))
```

```
mean = [37.  46.6 41.6]
distances = [ 72.70488292  70.22107946 103.85085459  92.46621004  69.26759704]
```

2.2.2 Broadcasting

The *broadcasting* is a way to compute operation between arrays of having different sizes in a implicit (and consice) manner. One concrete example could be to translate three positions $\vec{r}_i = (x, y)_i$ by a vector $\vec{d}_0$ simply by adding `points+d0` where `points.shape=(3,2)` and `d0.shape=(2,)`. Few examples are given below but more details are give in this documentation.

```
# operation between shape (3) and (1)
a = np.array([1, 2, 3])
b = np.array([5])
print('a+b = \n{}'.format(a+b))
```

```
a+b =
[6 7 8]
```

```
# operation between shape (3) and (1,2)
a = np.array([1, 2, 3])
b = np.array([
              [4],
              [5],
             ])
print('a+b = \n{}'.format(a+b))
```

```
a+b =
[[5 6 7]
 [6 7 8]]
```

```
# Translating 3 2D vectors by d0=(1,4)
points = np.random.normal(size=(3, 2))
d0 = np.array([1, 4])
print('points:\n {}\n'.format(points))
print('points+d0:\n {}'.format(points+d0))
```

```
points:
 [[ 0.55667403 -0.78854994]
 [ 2.6426123   1.5324207 ]
```

```
 [-0.73453837  0.81005631]]

points+d0:
 [[1.55667403 3.21145006]
 [3.6426123  5.5324207 ]
 [0.26546163 4.81005631]]
```

Not all shapes can be combined together and there are *broadcasting rules*, which are (quoting the numpy documentation):

> When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward. Two dimensions are compatible when
>
> 1. they are equal, or
> 2. one of them is 1

In a failing case, one can then add a new *empty axis* `np.newaxis` to an array to make their dimension equal and then the broadcasting possible. Here is a very simle example:

```
a = np.arange(10).reshape(2,5)
b = np.array([10,20])
```

```
try:
    res = a+b
    print('Possible for {} and {}:'.format(a.shape, b.shape))
    print('a+b = \n {}'.format(res))
except ValueError:
    print('Impossible for {} and {}'.format(a.shape, b.shape))
```

```
Impossible for (2, 5) and (2,)
```

```
c = b[:, np.newaxis]
try:
    res = a+c
    print('Possible for {} and {}:'.format(a.shape, c.shape))
    print('a+c = \n {}'.format(res))
except ValueError:
    print('Broadcasting for {} and {}'.format(a.shape, c.shape))
```

```
Possible for (2, 5) and (2, 1):
a+c =
 [[10 11 12 13 14]
 [25 26 27 28 29]]
```

2.2.3 Working with sub-arrays: slicing, indexing and mask (or selection)

As mentioned eariler, *slicing and indexing* are ways to access elements or sub-arrays in a smart way. Python allows slicing with `Slice()` object but `numpy` allows to push the logic much further with what is called *fancy indexing*. Few examples are given below and for more details, please have a look to this documentation page.

Rule 1: the syntax is `a[i]` to access the ith element. It is also possible to go from the last element using negative indices: `a[-1]` is the last element.

```
a = np.random.randint(low=1, high=100, size=10)
print('a = {}'.format(a))
print('a[2] = {}'.format(a[2]))
print('a[-1] = {}'.format(a[-1]))
print('a[[1, 2, 5]] = {}'.format(a[[1, 2, 5]]))
```

```
a = [98 93 61 78 48 93 47  7 43 31]
a[2] = 61
a[-1] = 31
a[[1, 2, 5]] = [93 61 93]
```

Rule 2: numpy also support array of indices. If the index array is multi-dimensional, the returned array will have the same dimension as the indices array.

```
# Small n-dimensional indices array: 3 arrays of 2 elements
indices = np.arange(6).reshape(3,2)
print('indices =\n {}'.format(indices))
print('a[indices] =\n {}'.format(a[indices]))
```

```
indices =
 [[0 1]
 [2 3]
 [4 5]]
a[indices] =
 [[98 93]
 [61 78]
 [48 93]]
```

```
# Playing with n-dimensional indices array: 2 arrays of (10, 10) arrays
indices_big = np.random.randint(low=0, high=10, size=(2, 3, 2))
print('indices_big =\n {}'.format(indices_big))
print('a[indices_big] =\n {}'.format(a[indices_big]))
```

```
indices_big =
 [[[3 5]
  [7 9]
  [1 9]]

 [[6 4]
  [7 9]
  [2 1]]]
a[indices_big] =
 [[[78 93]
  [ 7 31]
```

```
  [93 31]]

 [[47 48]
  [ 7 31]
  [61 93]]]
```

Rule 3: There is a smart way to access sub-arrays with the syntax `a[min:max:step]`. In that way, it's for example very easy to take one element over two (`step=2`), or reverse the order of an array (`step=-1`). This syntax works also for n-dimensional array, where each dimension is sperated by a comma. An example is given for a 1D array and for a 3D array of shape (5, 2, 3) - that can considered as 5 observations of 2 positions in space.

```
# 1D array
a = np.random.randint(low=1, high=100, size=10)
print('full array a              = {}'.format(a))
print('from 0 to 1: a[:2]        = {}'.format(a[:2]))
print('from 4 to end: a[4:]      = {}'.format(a[4:]))
print('reverse order: a[::-1]    = {}'.format(a[::-1]))
print('all even elements: a[::2] = {}'.format(a[::2]))
```

```
full array a              = [71 51 35  3  9 27 68 24 25 98]
from 0 to 1: a[:2]        = [71 51]
from 4 to end: a[4:]      = [ 9 27 68 24 25 98]
reverse order: a[::-1]    = [98 25 24 68 27  9  3 35 51 71]
all even elements: a[::2] = [71 35  9 68 25]
```

```
# 3D array
a = np.random.randint(low=0, high=100, size=(5, 2, 3))
print('a = \n{}'.format(a))
```

```
a =
[[[45 26 15]
  [47 86 38]]

 [[98 11 61]
  [26 81 26]]

 [[15 53 25]
  [66  2 10]]

 [[96 22 88]
  [90 36 17]]

 [[69 35 54]
  [ 2 22 90]]]
```

Let's say, one wants to take only the (x, y) coordinates for the first vector for all 5 observations. This is how each axis will be sliced: - first axis (=5 observations): `:`, *i.e.* takes all - second axis (=2 vectors): `1` *i.e.* only the 2nd element - third axis (=3 coordinates): `0:2` *i.e.* from 0 to $2 - 1 = 1$, so only (x, y)

```
# Taking only the x,y values of the first vector for all observation:
print('a[:, 0, 0:2] =\n {}'.format(a[:, 0, 0:2]))
```

```
a[:, 0, 0:2] =
 [[45 26]
 [98 11]
 [15 53]
 [96 22]
 [69 35]]
```

```
# Reverse the order of the 2 vector for each observation:
print('a[:, ::-1, :] = \n{}'.format(a[:, ::-1, :]))
```

```
a[:, ::-1, :] =
[[[47 86 38]
  [45 26 15]]

 [[26 81 26]
  [98 11 61]]

 [[66  2 10]
  [15 53 25]]

 [[90 36 17]
  [96 22 88]]

 [[ 2 22 90]
  [69 35 54]]]
```

Rule 4: The last part of of indexing is about *masking* array or in a more common language, *selecting* sub-arrays/elements. This allows to get only elements satisfying a given criteria, exploiting the indexing rules described above. Indeed, a boolean operation applied to an array such as `a>0` will directly return an array of boolean values `True` or `False` depending if the corresponding element satisfies the condition or not.

```
a = np.random.randint(low=-100, high=100, size=(5, 3))
mask = a>0
print('a = \n{}'.format(a))
print('\nmask = \n {}'.format(mask))
```

```
a =
[[  94  -73    3]
 [-100   89   47]
 [  47  -65  -53]
 [ -35 -100  -21]
 [ -65  -63  -23]]

mask =
```

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

```
print('\na[mask] = \n {}'.format(a[mask])) # always return 1D array
print('\na*mask = \n {}'.format(a*mask)) # preserves the dimension (False=0)
print('\na[~mask] = \n {}'.format(a[~mask])) # ~mask is the negation of mask
print('\na*~mask  = \n {}'.format(a*~mask)) # working for a product too.
```

```
a[mask] =
 [94  3 89 47 47]

a*mask =
 [[94  0  3]
 [ 0 89 47]
 [47  0  0]
 [ 0  0  0]
 [ 0  0  0]]

a[~mask] =
 [ -73 -100  -65  -53  -35 -100  -21  -65  -63  -23]

a*~mask  =
 [[   0  -73    0]
 [-100    0    0]
 [   0  -65  -53]
 [ -35 -100  -21]
 [ -65  -63  -23]]
```

Note the case of boolean arrays as indices has then a special treatment in numpy (since the result is always a 1D array). There is actually a dedicated numpy object called *masked array* (cf. documentation) which allows to keep the whole array but without considering some elements in the computation (*e.g.* CCD camera with dead pixel). Note however that when a boolean array is used in an mathematical operation (such as `a*mask`) then `False` is treated as `0` and `True` as `1`:

```
print('a+mask = \n{}'.format(a+mask))
```

```
a+mask =
[[  95  -73    4]
 [-100   90   48]
 [  48  -65  -53]
 [ -35 -100  -21]
 [ -65  -63  -23]]
```

This boolean arrays are also very useful to *replace a category of elements* with a given value in a very easy, consise and readable way:

```
a = np.random.randint(low=-100, high=100, size=(5, 3))
print('Before: a=\n{}'.format(a))

a[a<0] = a[a<0]**2
print('\nAfter: a=\n{}'.format(a))
```

```
Before: a=
[[-16  76  19]
 [ 23 -25  75]
 [-70 -38  35]
 [ 61 -13 -72]
 [ 60  40  93]]

After: a=
[[ 256   76   19]
 [  23  625   75]
 [4900 1444   35]
 [  61  169 5184]
 [  60   40   93]]
```

Chapter 3

Three important tools to know

3.1 `matplotlib`: graphical respresentation of data

Matplolib is an extremely rich librairy for data visualization and there is no way to cover all its features in this note. The goal of this section is just to give short and practical examples to plot data. Much more details can be obtained on the webpage. The following shows how to quickly make *histograms*, *graph*, *2D and 3D scatter plots*.

The main object of matplotlib is `matplotlib.pyplot` imported as `plt` here (and usually). The most common functions are then called on this objects, and often takes numpy arrays in argument (possibly with more than one dimension) and a lot of `kwargs` to define the plotting style.

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

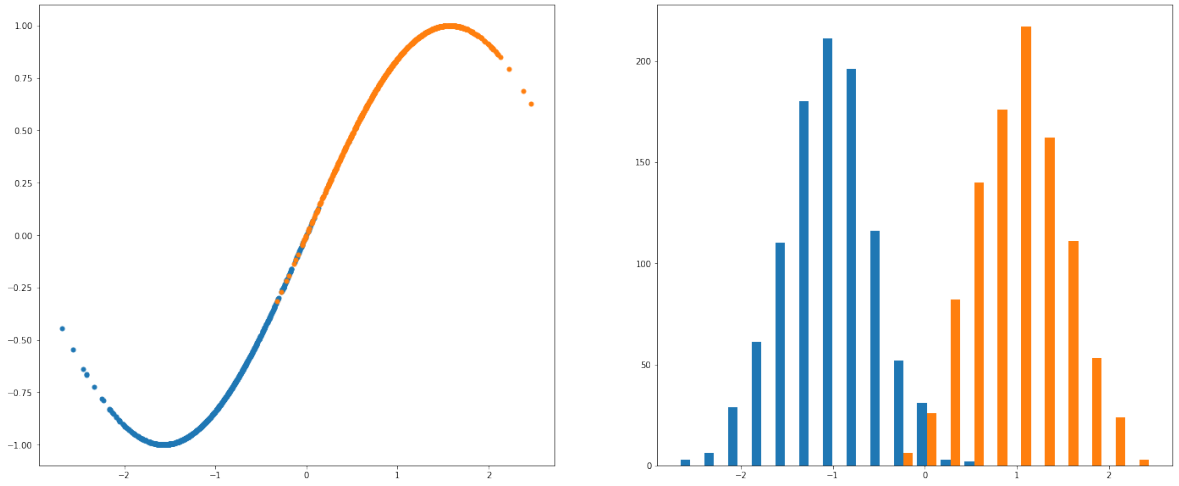
3.1.1 Example of 1D plots and histograms

To play with data, we generate 2 samples of 1000 values distributed according to a normal probability density function with $\mu = -1$ and $\mu = 1$ respectively, and $\sigma = 0.5$. These data are stored in a numpy array `x` of shape `x.shape=(1000, 2)`. We then simply compute and store the sinus of all these values into a same shape array `y`:

```
x = np.random.normal(loc=[-1, 1], scale=[0.5, 0.5], size=(1000,2))
y = np.sin(x)
```

The next step is to plot these data in two ways: first we want `y` v.s. `x`, second we want the histogram of the `x` values. We need to first create a figure, then create two *subplots* (specifying the number of line, column, and subplot index). Note that matplotlib take always the first dimension to define the numbers to plot, while higher dimensions are considered as other plots - automatically overlaid.

```
plt.figure(figsize=(24, 10))
plt.subplot(121) # 121 means 1 line, 2 column, 1st plot
plt.plot(x, y, marker='o', markersize=5, linewidth=0.0)
plt.subplot(122) # 122 means 1 line, 2 column, 2nd plot
plt.hist(x, bins=20);
```



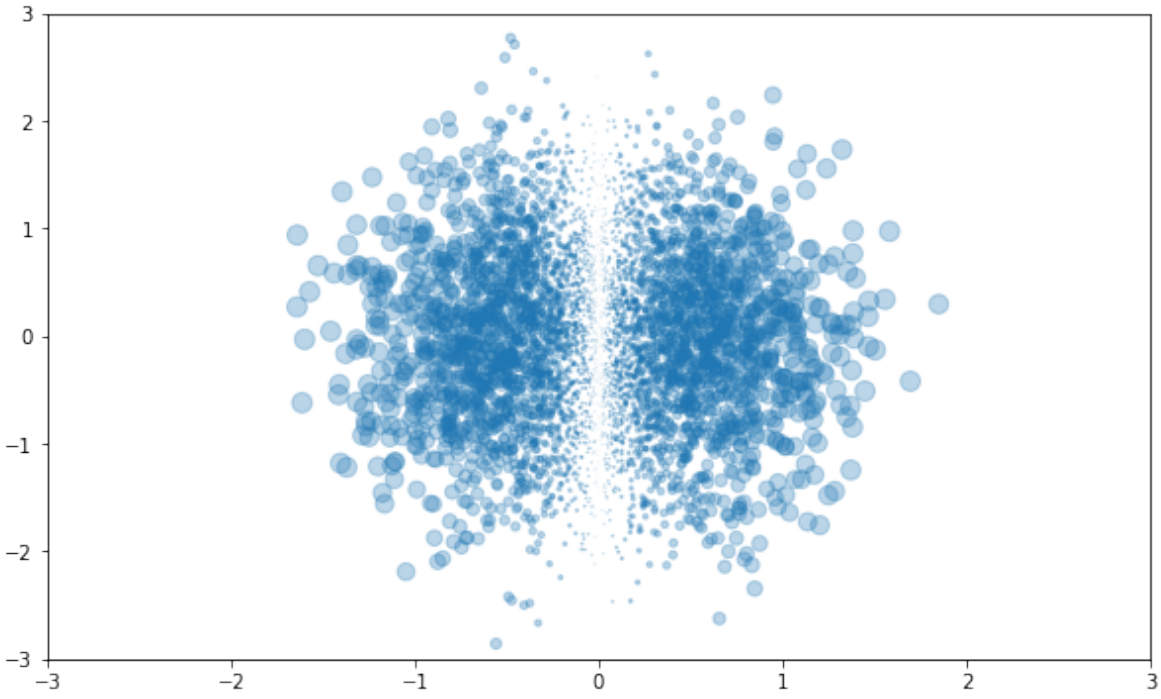
3.1.2 Example of 2D scatter plot

A scatter plot allows to draw marker in a 2D space and a thrid information is encoded into the marker size. In order to play, we generated two set of 5000 numbers distributed according to uncorrelated gaussians of (μ_0 = μ_1 = 0) and (σ_1, σ_2) = (0.5, 0.8) in a numpy array `points` of shape `points.shape=(5000,2)`. These two sets of numbers are then interepreted as (x, y) positions being loaded in two `(5000, 1)` arrays `x` and `y`:

```
points = np.random.normal(loc=[0, 0], scale=[0.5, 0.8], size=(5000,2))
x, y = points[:, 0], points[:, 1]
```

We can then plot the 5000 points in the 2D plan, and here we specify the marker size at 100 $\times$ $\sin^2(x)$ using the argument `s` of `the plt.scatter()` function (note that the array `x`, `y` and `s` must have the same shape):

```
plt.figure(figsize=(10,6))
plt.scatter(x, y, s=100*(np.sin(x))**2, marker='o', alpha=0.3)
plt.xlim(-3, 3)
plt.ylim(-3, 3);
```



3.1.3 Example of 3D plots

For 3D plots, one can generate 1000 positions in space, and operate a translation by a vector $\vec{r}_0$ using broadcasting:

```
data = np.random.normal(size=(1000, 3))
r0 = np.array([1, 4, 2])
data_trans = data + r0
```

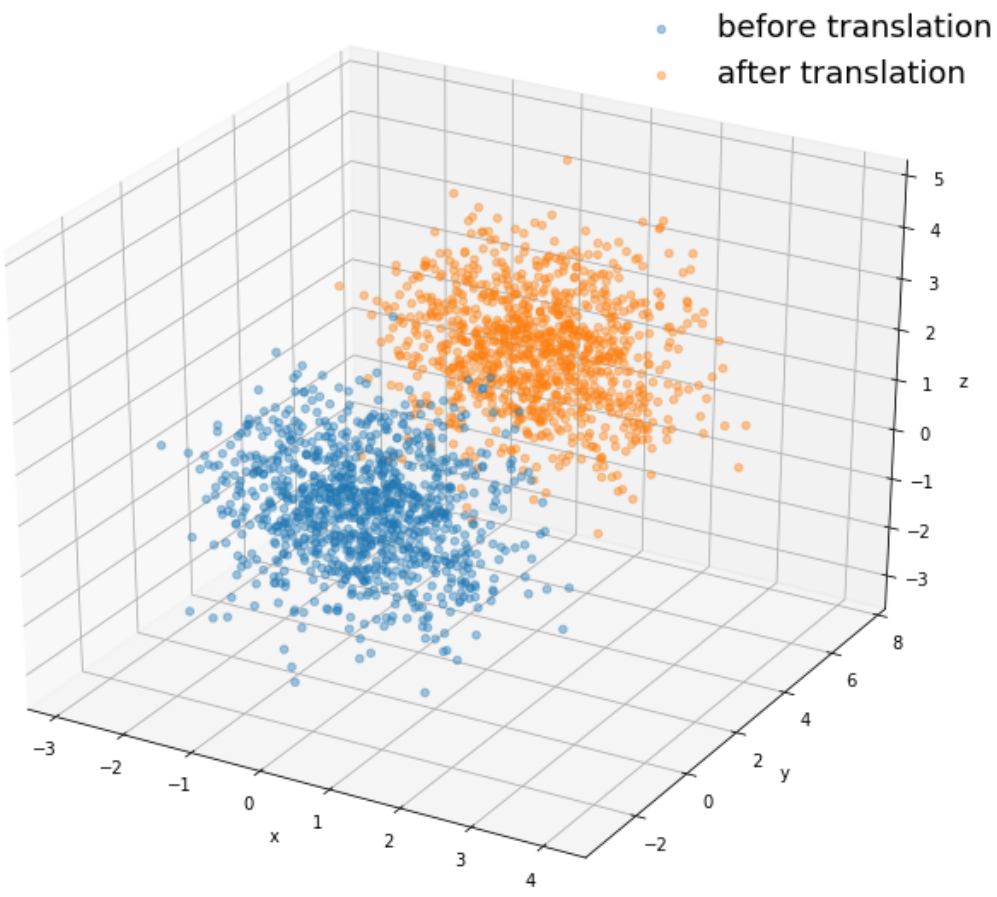
It is then easy to get back the spatial initial (*i.e.* before translation) and final (*i.e.* after translation) coordinates:

```
xi, yi, zi = data[:,0], data[:,1], data[:,2]
xf, yf, zf = data_trans[:,0], data_trans[:,1], data_trans[:,2]
```

An additional module must be imported in order to plot data in three dimensions, and the projection has to be stated. Once it's done, a simple call to `ax.scatter3D(x,y,z)` does the plot. Note that we call a function of `ax` and not `plt` as before. This is due to the `ax = plt.axes(projection='3d')` command which is needed for 3D plotting. More details are available on the matplotlib 3D tutorial.

```
from mpl_toolkits import mplot3d
plt.figure(figsize=(12,10))
ax = plt.axes(projection='3d')
ax.scatter3D(xi, yi, zi, alpha=0.4, label='before translation')
ax.scatter3D(xf, yf, zf, alpha=0.4, label='after translation')
ax.set_xlabel('x')
```

```
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.legend(frameon=False, fontsize=18);
```



3.1.4 Example of 2D function $z = f(x, y)$: notion of `meshgrid`

Another typical plot we might want to do is to represent a function of two variables (x, y) in 3D: $z = f(x, y)$. In python this implies the notion of *meshgrid* which is not trivial at first. Let's first define a 2 variable function:

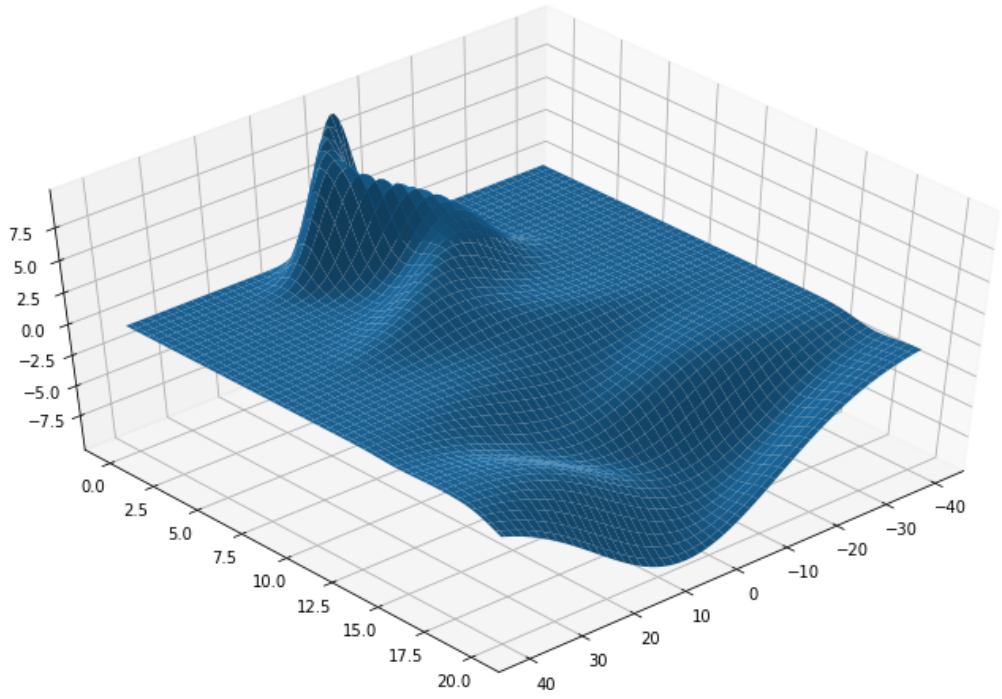
```
def my_surface(x, y):
    x0 = 5*np.sin(y)
    sigma = 5+y
    amp = (10-y)
    return amp*np.exp(-(x-x0)**2/sigma**2)
```

Let's define a (x, y) interval on which we want to describe the surface:

```
x = np.linspace(-40, 40, 100)
y = np.linspace(0, 20, 200)
```

These two numpy arrays don't have the same shape and an explicit loop would be needed to process them - which is very time consuming in python. This is where the *meshgrid* notion comes: it will provide a two arrays *with the same size* and allow then the vectorization:

```
# Meshgrid and function application
xx, yy = np.meshgrid(x, y)
Z = my_surface(xx, yy)

# Plotting
fig = plt.figure(figsize=(13,8))
ax = fig.gca(projection='3d', )
ax.plot_surface(xx, yy, Z)

# Choose the default view
ax.view_init(azim=48, elev=48)
```



3.2 pandas: import and manipulate data as numpy array

The package pandas is an very rich interface to read data from different format and produce a pandas.dataframe that can be based on numpy (but contanining a lot more features). There is no way to fully desribe this package here, the goal is simply to give functional and concrete example easily usable. More more details, please check the pandas webpage.

Many build-in functions are available to import data as pandas dataframe. One, which is particularly convenient, directly reads csv files (one can specify the columns to loads, the row to skip, and many other options ...):

```
import pandas as pd
df = pd.read_csv('../data/WaveData.csv')
print(df.head())
```

```
          Date/Time      Hs   Hmax      Tz      Tp  Peak Direction    SST
0  01/01/2017 00:00 -99.900 -99.90 -99.900 -99.900           -99.9 -99.90
1  01/01/2017 00:30   0.875   1.39   4.421   4.506           -99.9 -99.90
2  01/01/2017 01:00   0.763   1.15   4.520   5.513            49.0  25.65
3  01/01/2017 01:30   0.770   1.41   4.582   5.647            75.0  25.50
4  01/01/2017 02:00   0.747   1.16   4.515   5.083            91.0  25.45
```

```
# Rename columns names using df.rename() function
old_new_cols = {
    'Date/Time': 'date',
    'Hs': 'height',
    'Hmax': 'heightMax',
    'Tz': 'period',
    'Tp': 'energy',
    'Peak Direction': 'direction',
    'SST': 'temperature'
}

df.rename(columns=old_new_cols, inplace=True)
print(df.head())
```

```
              date  height  heightMax  period  energy  direction  temperature
0  01/01/2017 00:00 -99.900     -99.90 -99.900 -99.900      -99.9       -99.90
1  01/01/2017 00:30   0.875       1.39   4.421   4.506      -99.9       -99.90
2  01/01/2017 01:00   0.763       1.15   4.520   5.513       49.0        25.65
3  01/01/2017 01:30   0.770       1.41   4.582   5.647       75.0        25.50
4  01/01/2017 02:00   0.747       1.16   4.515   5.083       91.0        25.45
```

This is possible to clean the dataframe using some masking syntax. First, let's check how many default values are stored for each column (all but the date):

```
# Check which wave has -99 values for every variables
for c in ['height', 'heightMax', 'period', 'energy', 'direction', 'temperature']:
    n = np.count_nonzero(df[c]<=-99)
    print('{}: {} wave have <=-99'.format(c, n))
```

```
height: 85 wave have <=-99
heightMax: 85 wave have <=-99
period: 85 wave have <=-99
```

```
energy: 85 wave have <=-99
direction: 271 wave have <=-99
temperature: 262 wave have <=-99
```

```
# Simply take value above -99
print(df[df>-99].head())
```

```
                date  height  heightMax  period  energy  direction  temperature
0  01/01/2017 00:00     NaN        NaN     NaN     NaN        NaN          NaN
1  01/01/2017 00:30   0.875       1.39   4.421   4.506        NaN          NaN
2  01/01/2017 01:00   0.763       1.15   4.520   5.513       49.0        25.65
3  01/01/2017 01:30   0.770       1.41   4.582   5.647       75.0        25.50
4  01/01/2017 02:00   0.747       1.16   4.515   5.083       91.0        25.45
```

```
# Removing all entry (line) which has at least one default value
for c in ['height', 'heightMax', 'period', 'energy', 'direction', 'temperature']:
    df = df[df[c]>-99]

print(df.head())
```

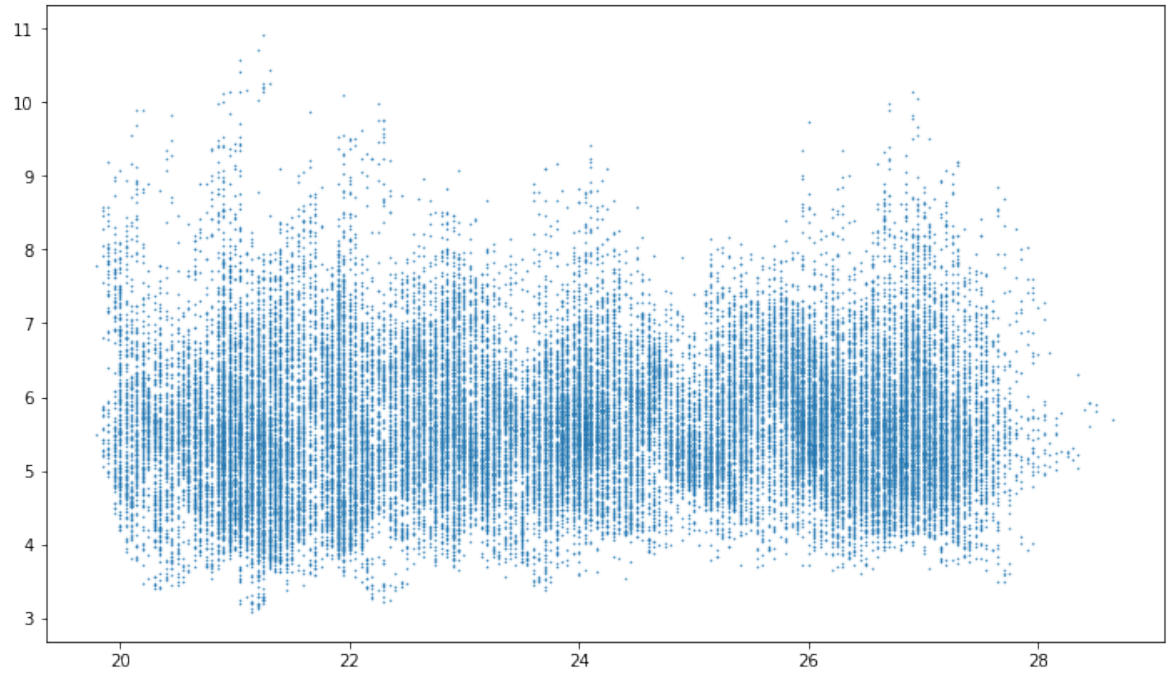
```
                date  height  heightMax  period  energy  direction  temperature
2  01/01/2017 01:00   0.763       1.15   4.520   5.513       49.0        25.65
3  01/01/2017 01:30   0.770       1.41   4.582   5.647       75.0        25.50
4  01/01/2017 02:00   0.747       1.16   4.515   5.083       91.0        25.45
5  01/01/2017 02:30   0.718       1.61   4.614   6.181       68.0        25.45
6  01/01/2017 03:00   0.707       1.34   4.568   4.705       73.0        25.50
```

We can now check how many default value we get:

```
for c in ['height', 'heightMax', 'period', 'energy', 'direction', 'temperature']:
    n = np.count_nonzero(df[c]<=-99)
    print('{}: {} wave have <=-99'.format(c, n))
```

```
height: 0 wave have <=-99
heightMax: 0 wave have <=-99
period: 0 wave have <=-99
energy: 0 wave have <=-99
direction: 0 wave have <=-99
temperature: 0 wave have <=-99
```

```
# Disable interactive matplotlib
%matplotlib inline

# Plot temperature vs period
plt.figure(figsize=(12, 7))
plt.scatter(df['temperature'], df['period'], s=0.2);
```



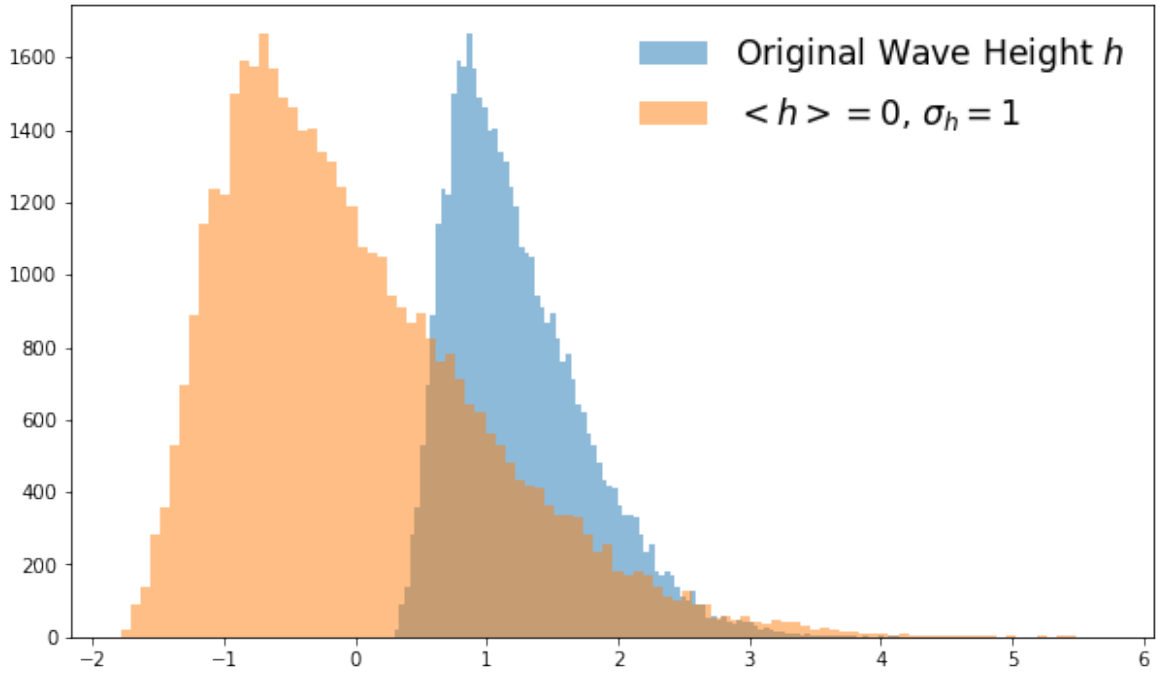
On of the nice feature of pandas is to be able to easily get numpy array, compute and store the result as a new column. For instance, it's a common practice in machine learning to *normalize* the input variables, *i.e.* transform them to have a mean of 0 and a variance of 1.0. The following example shows how to add new column which are normalized:

```
def add_normalized_variable_to_df(col):

    # Get a numpy arrays
    v = df[col].values

    # Replace NaN by 0.0
    v[np.isnan(v)] = 0

    # Compute quantities
    v_mean = np.mean(v)
    v_rms = np.sqrt(np.mean((v-v_mean)**2))

    # Add them into the pandas dataframe
    df[col+'_normalized'] = (v-v_mean)/v_rms

    return


for c in ['height', 'heightMax', 'period', 'energy', 'direction', 'temperature']:
    add_normalized_variable_to_df(c)

# Get only normalized column
normalized_cols = [c for c in df.columns.tolist() if '_normalized' in c]
print(df[normalized_cols].head())
```

```
   height_normalized  heightMax_normalized  period_normalized  \
2          -0.898215             -1.047342          -1.184338
3          -0.884973             -0.757690          -1.117565
4          -0.928484             -1.036201          -1.189723
5          -0.983346             -0.534881          -1.083102
6          -1.004155             -0.835673          -1.132643

   energy_normalized  direction_normalized  temperature_normalized
2          -1.463956             -2.044360                0.762152
3          -1.407891             -0.973294                0.694918
4          -1.643866             -0.314176                0.672506
5          -1.184467             -1.261658                0.672506
6          -1.802020             -1.055683                0.694918
```

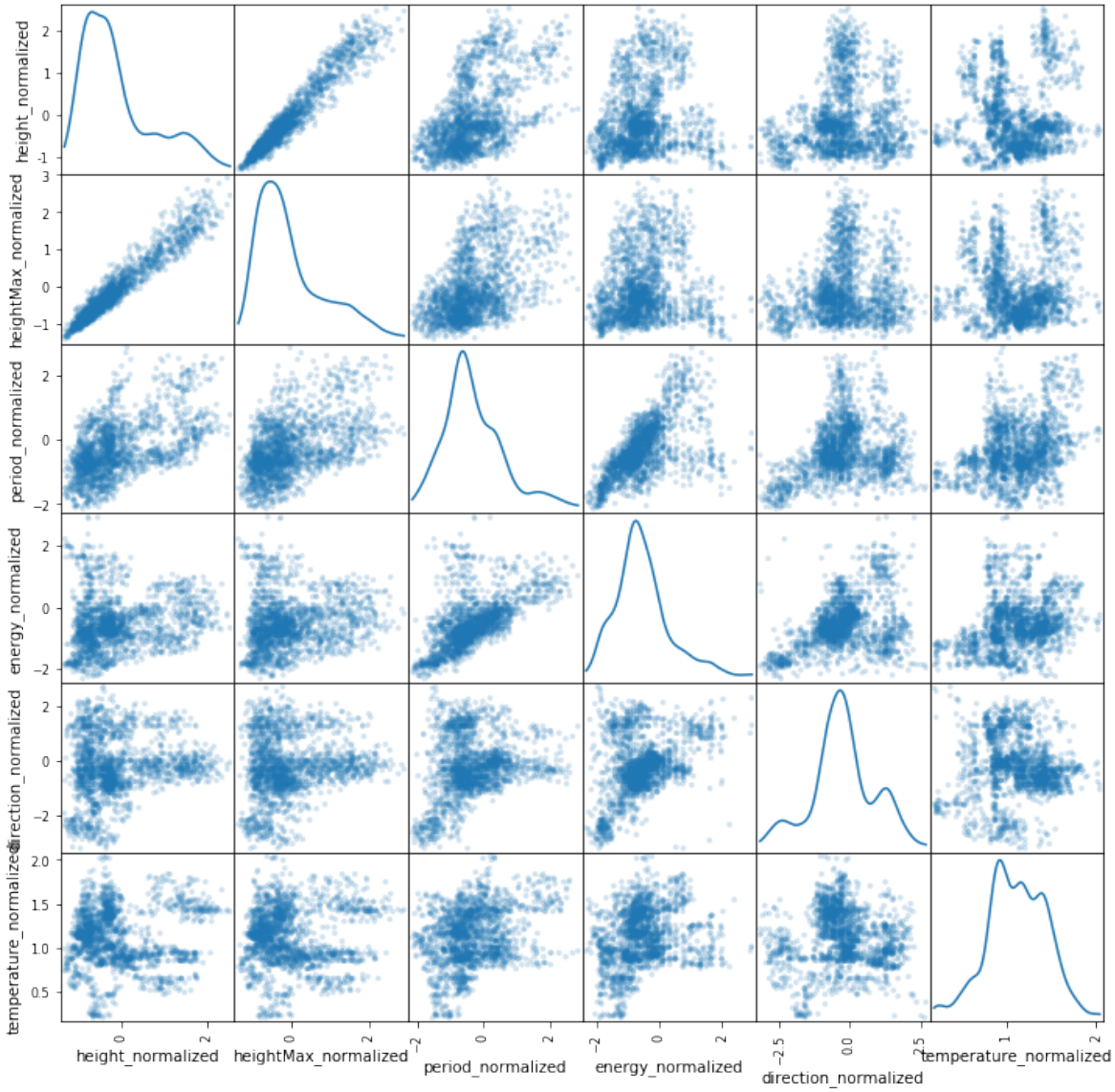
One can simply plot the content of a pandas dataframe using the name of the column. For instance, one can compare the evolution of H_T after each transformation (which is trivial in this illustrative case):

```
plt.figure(figsize=(10, 6))
plt.hist(df['height'], bins=100, alpha=0.5, label='Original Wave Height $h$')
plt.hist(df['height_normalized'], bins=100, alpha=0.5, label='$<h> = 0$, $\sigma_{h}=1$')
plt.legend(frameon=False, fontsize='xx-large');
```



There are also many plotting function already included into the pandas library. To show only one example (all functions are decribed in the pandas visualization tutorial), here is the *scatter matrix* between variables (defined as a subset of the ones stored the dataframe) obtained in a single line of code:

```
from pandas.plotting import scatter_matrix
scatter_matrix(df[normalized_cols][:2000], figsize=(12, 12), alpha=0.2, s=50,
↪  diagonal='kde');
```



3.3 `scipy`: mathematics, physics and engineering

The scipy project is python-based ecosystem of open-source software for mathematics, science, and engineering. In particular, the following core package are part of it: NumPy, matplotlib, pandas, scipy library (very quickly introduced here) and SymPy (symbolic calculations with mathematical expressions *a la* mathematica).

Obviously, there is no way to extensively present the scipy library in this short introduction, but one can quickly summarize few features and illustrate one with a concrete and useful example: fitting data points with a function. Among the main features, the SciPy library contains:

- Integration (`scipy.integrate`): integrals, differential equations, etc ...
- Optimization (`scipy.optimize`): minimization, fits, etc ...
- Interpolation (`scipy.interpolate`): smoothing methods, etc ...
- Fourier Transforms (`scipy.fftpack`): spectral analysis, etc ...

- Signal Processing (`scipy.signal`): transfer functions, filtering, etc ...
- Linear Algebra (`scipy.linalg`): matrix operation, diagonalisation, determinant, etc ...
- Statistics (`scipy.stats`): random number, probability density function, cumulative distribution, etc ...

```
from scipy import optimize
from scipy import stats
```

Let's now show how to perform a fit of data with error bar using one particular function of `scipy.optimize`. First, we need to generate some data where we choose 20 measurements, with some noise of ~30% and an combined uncertainty of an absolute 0.1 uncertainty and 10% relative uncertainy:

```
Npoints, Nsampling = 20, 1000
xcont = np.linspace(-5.0, 3.5, Nsampling)
x = np.linspace(-5, 3.0, Npoints)
y = 2*(np.sin(x/2)**2 + np.random.random(Npoints)*0.3)
dy = np.sqrt(0.10**2 + (0.10*y)**2)
```

Then we need to define functions with which we want to fit our data, for example a degree 1 polynoms. The syntax has to be `func(x, *pars)`:

```
def pol1(x, p0, p1):
    return p0 + x*p1
```

The following lines actually perform the fit and return both the optimal parameters and the covariances for the dgree 1 polynom:

```
p, cov = optimize.curve_fit(pol1, x, y, sigma=dy)
```

One can then generalize the procedure by plotting the result of the fit for polynoms of several degrees, after having plotted the data. This is a good way to compare different models for the same data. First, we define an arbitrary degree polynom `plo_func()` and we *vectorize* it using `np.vectorize` so that it can accept NumPy arrays:

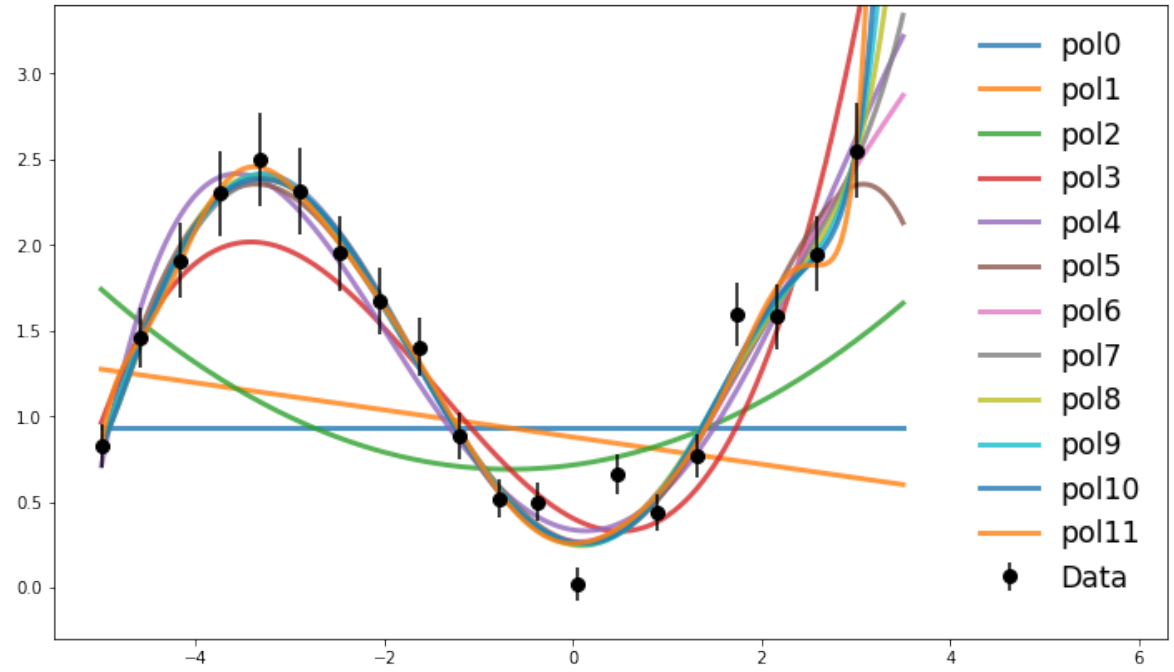
```
def pol_func(x, *coeff):
    '''Arbitrary degree polynom: f(x) = a0 + a1*x + a2*x^2 + ... aN*x^N'''
    a = np.array([coeff[i]*x**i for i in range(len(coeff))])
    return np.sum(a)

pol_func = np.vectorize(pol_func)
```

In the previous call for `optimize.curve_fit()`, we didn't use additional arguement. For this example, we need to specify at least the starting point of the parameters `p0` *because the number of paramter will be assessed using `len(p0)` (it's not known a priori since it is dynamically allocated)*. Other options can be specified, such as the miminum and maximum allowed values of parameters. Here is a wrapp-up function performing the fit for an arbitrary polynom degree:

```
def fit_polynom(degree):
    nPars = degree+1
    p0, pmin, pmax = [1.0]*nPars, [-10]*nPars, [10]*nPars
    fit_options = {'p0': p0, 'bounds': (pmin, pmax), 'check_finite': True}
    par, cov = optimize.curve_fit(pol_func, x, y, sigma=dy, **fit_options)
    return par, cov

degree_max = 12
```

The following code try every polynomial functions up to a degree `degree_max=`, perform the fit and overlay the the result for each together with the experimental data on the same figure:

```
# Figure for the result
fig = plt.figure(figsize=(12,7))

# Fitting & plotting
for d in np.arange(0, degree_max):
    par, cov = fit_polynom(d)
    plt.plot(xcont, pol_func(xcont, *par), label='pol{}'.format(d),
             linewidth=3, alpha=0.8)

# Plotting data
style = {'marker': 'o', 'color': 'black', 'markersize': 8,
         'linestyle': '', 'zorder': 10, 'label': 'Data'}
plt.errorbar(x, y, yerr=dy, **style)

# Plot cosmetics
plt.xlim(-5.5, 6.3)
plt.ylim(-0.3, 3.4)
plt.legend(frameon=False, fontsize='xx-large');
```



It is possible to quantify how well a given model explain the observations, computing what we call the *goodness of fit*. In a frequentist approach, this can be assessed by the fraction of pseudo-data coming from - in principle - repeating the exact same experiment, with to a worst agreement for a given model. The agreement can be quantified using $\chi^2 = \sum_{i=1}^{n} \frac{(y_i - f(x_i))^2}{\sigma_i^2}$ and its probability density function (PDF) directly gives access to the fraction of "worst pseudo-data" (by integrating the PDF from χ^2 to ∞). More precisely, one can use the cumulative distribution function (CDF) of χ^2 computed with n degrees of freedom, for instance `Npoins`, *i.e.* `len(x)`. More details can be found, for examble, in the statistics review of the Particle Data Group. The following two functions allow to compute the goodness of fit:

```
def get_chi2_nDOF(y, dy, yfit):
    r = (y-yfit)/dy
    return  np.sum(r**2), len(y)

def get_pvalue(chi2, nDOF):
    return 1-stats.chi2.cdf(chi2, df=nDOF)
```

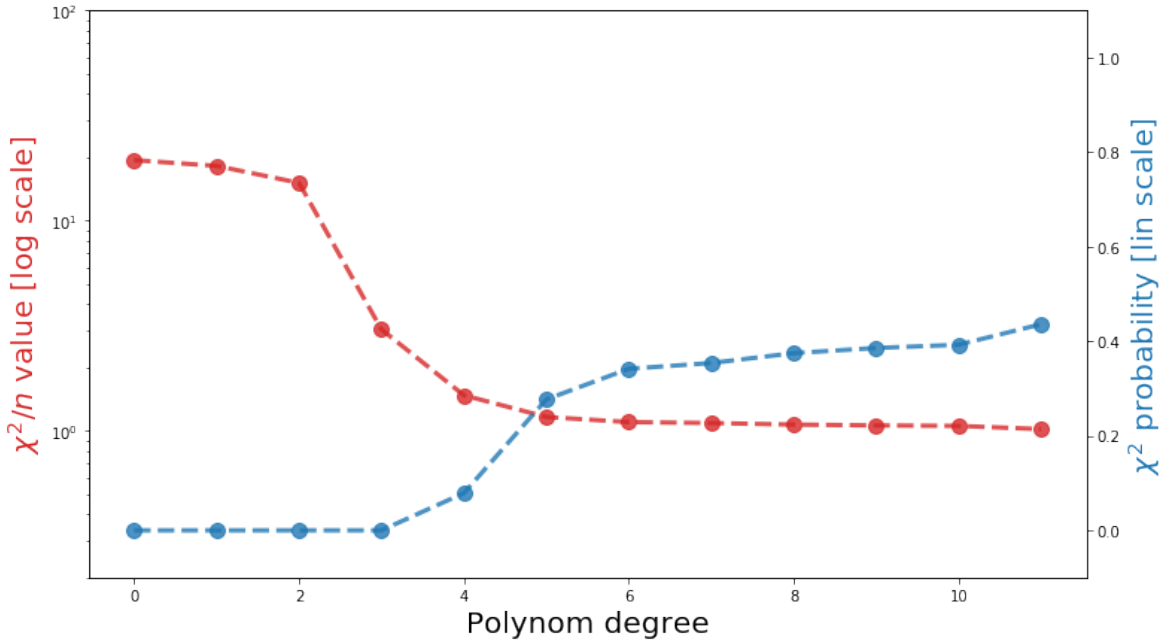
We can now perform all these fits and extract the goodness of fit (χ^2 and p-value) for each model:

```
# Fitting and getting p-value
degree, chiSquare, pvalue = [], [], []
for d in np.arange(degree_max):
    par, cov = fit_polynom(d)
    c2, n = get_chi2_nDOF(y, dy, pol_func(x, *par))
    degree.append(d), chiSquare.append(c2), pvalue.append(get_pvalue(c2, n))
```

The following piece of code plot both the χ^2 and the p-value versus the degree of the polynom using two different y-axis. This gives another way to use matplolib by defining explicit object such as `ax` and `fig` and

call methods on those (called *stateless appraoch*), instead of using function on `plt` (called *stateful approach*). For more details on these different approaches, see this RealPython post.

```
# Plotting the result with 2 different axis
fig, ax1 = plt.subplots(figsize=(12,7))
ax1.set_xlabel('Polynom degree', fontsize=20)
style = {'marker': 'o', 'markersize': 10, 'alpha': 0.8,
         'linestyle': '--', 'linewidth': 3}

# Plot chi2/n
ax1.semilogy(degree, np.array(chiSquare)/Npoints, color='tab:red', **style)
ax1.set_ylim(0.2, 100)
ax1.set_ylabel('$\chi^2/n$ value [log scale]', color='tab:red', fontsize=20)

# Plot p-values
ax2 = ax1.twinx()
ax2.plot(degree, pvalue, color='tab:blue', **style)
ax2.set_ylim(-0.1, 1.1)
ax2.set_ylabel('$\chi^2$ probability [lin scale]', color='tab:blue', fontsize=20);
```



This is also possible to know whether two models give a similar description or if one model is better than the other. This is called the *F-test*, based on the residual sum of square $RSS \equiv \sum(y_i - f(x_i))^2$ (RSS). For a fit of n data points with a model M1 defined by p_1 parameters with RSS_1 and a model M2 defined by p_2 parameters with RSS_2 *with* $p_2 > p_1$, the F-test is defined by:

$$F(M1, M2) = \frac{\left(\frac{RSS_1 - RSS_2}{p_2 - p_1}\right)}{\left(\frac{RSS_2}{n - p_2}\right)}$$

If the p-value of F is close to 1.0, it means that the two model are equally compatible with data and there

is no indication that a choice should be made. In that case, one would favour the simplest model with less paramters. If the p-value of F is close to 0, then the two compared model are actually different and one has to select one.

Let's first define the RSS function and compute it over all the polynoms:

```
def get_RSS(y, yfit):
    return np.sum((y-yfit)**2)

RSS = []
for d in np.arange(degree_max):
    par, cov = fit_polynom(d)
    RSS.append(get_RSS(y=y, yfit=pol_func(x, *par)))
```

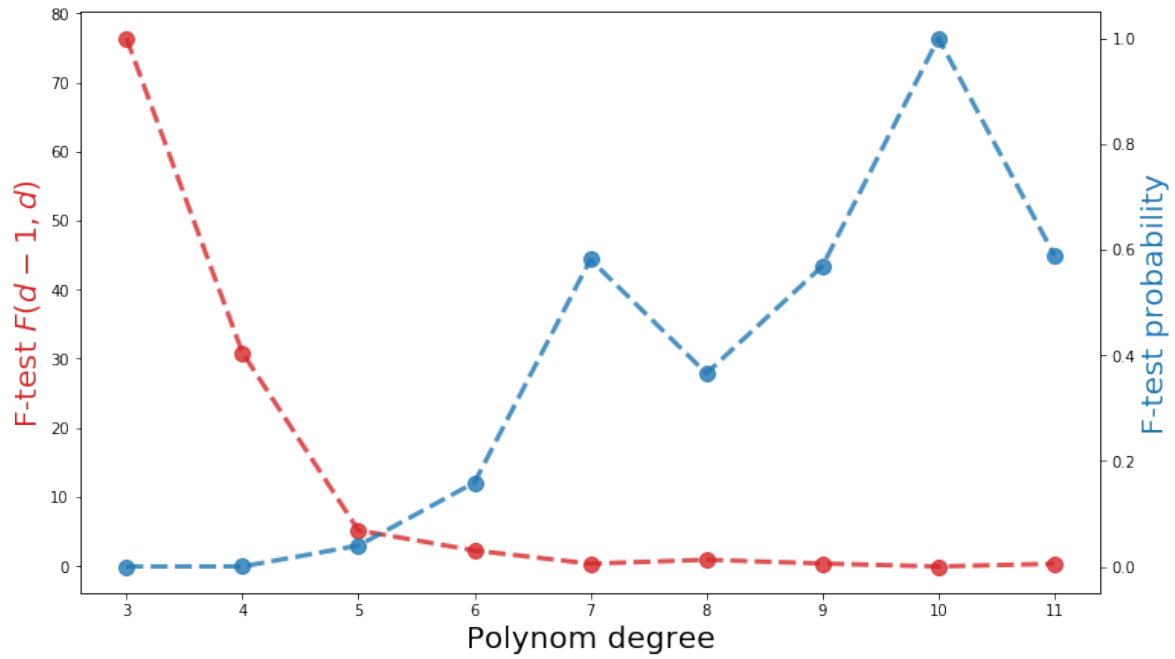
Then, one can define a function to actually compare two degrees where d_2 must be larger than d_1:

```
def compare_pol_pq(d1, d2):

    # Sanity checks
    if d1 not in degree or d2 not in degree:
        raise NameError('Degree no supported')
    if d2<d1:
        d2, d1 = d1, d2

    # Extract M1 and M2 numbers
    RSS1, p1 = RSS[d1], d1+1
    RSS2, p2 = RSS[d2], d2+1

    # F-value & p-value
    Fval = (RSS1-RSS2)/(p2-p1) * (Npoints-p2)/RSS2
    pval = 1-stats.f.cdf(x=Fval, dfn=p2-p1, dfd=Npoints-p2)

    return Fval, pval
```

Plot the F-test values and its p-values for the comparison between $d-1$ and d starting from $d=3$. This plot shows that from $d=6$, there is a faire compatibility between the models (at worst 15%):

```
# Computing Ftest comparisons starting from degree 4
dmin, compare = 3, compare_pol_pq
dinf, dsup = np.arange(dmin-1, degree_max), np.arange(dmin, degree_max)
result = np.array([compare(d1, d2) for d1, d2 in zip(dinf, dsup)])
Ftest, pval = result[:, 0], result[:, 1]

# Plotting figure and syle the results
fig, ax1 = plt.subplots(figsize=(12,7))
ax1.set_xlabel('Polynom degree', fontsize=20)
style = {'marker': 'o', 'markersize': 10, 'alpha': 0.8,
         'linestyle': '--', 'linewidth': 3}
```

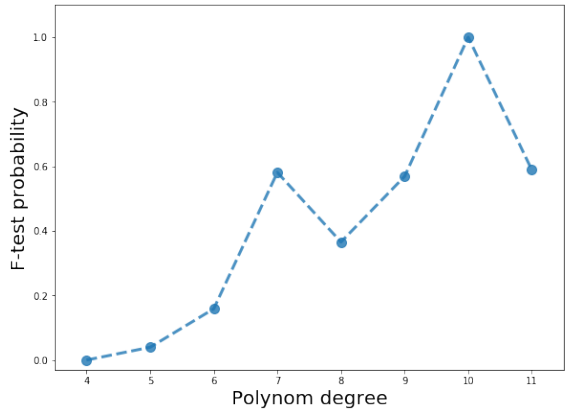
```
# Plot of F-test values
ax1.plot(dsup, Ftest, color='tab:red', **style)
ax1.set_ylabel('F-test $F(d-1,d)$', color='tab:red', fontsize=20)

# Plot of p-values
ax2 = ax1.twinx()
ax2.plot(dsup, pval, color='tab:blue', **style)
ax2.set_ylabel('F-test probability', color='tab:blue', fontsize=20);
```

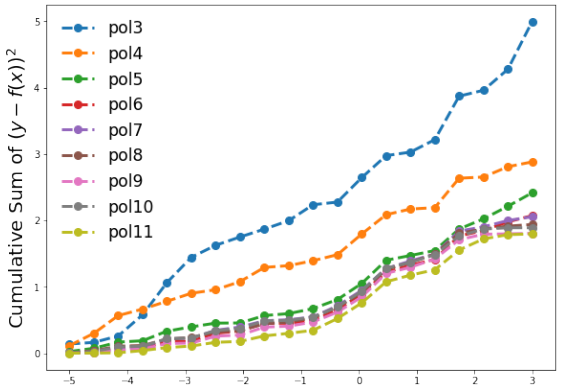


In order to effectively check by eye that the F-test give a sensible information, one can plot the cumulative sum of fit residus $(y - f(x))^2$ and the F-test probability side-by-side:

```
# Plotting figure and syle the results
plt.figure(figsize=(21,7))
style = {'marker': 'o', 'markersize': 10, 'alpha': 0.8,
         'linestyle': '--', 'linewidth': 3}

# Plot of F-test values
plt.subplot('121')
plt.plot(dsup[1:], pval[1:], **style)
plt.ylabel('F-test probability' , fontsize=20);
plt.xlabel('Polynom degree', fontsize=20)
plt.xlim(3.5, 11.5)
plt.ylim(-0.03, 1.1)

# Putting the cumsum of fit residus aside
plt.subplot('122')
style = {'marker': 'o', 'markersize': 8,
         'linestyle': '--', 'linewidth': 3}
```

```
for d in np.arange(3, degree_max):
    par, cov = fit_polynom(d)
    RSScum = np.cumsum(np.abs(y-pol_func(x, *par)))
    plt.plot(x, RSScum, label='pol{}'.format(d), **style)
plt.legend(frameon=False, fontsize='xx-large');
plt.ylabel('Cumulative Sum of $(y-f(x))^2$', fontsize=20);
```





We can then see on these 2 plots that indeed, comparing low degrees polynoms (like $d = 4$ and $d = 3$) leads to q large difference (and thus a low F-test probability), while comparing high degrees polynoms (*e.g.* $d = 6$ and $d = 5$) leads to very similar prediction meaning a sizable F-test probability. On sees that the probability that $d = 10$ is similar to $d = 10$ is quite low, which can be probably understood by a quite different RSS (last point on the right plot above for `pol11` and `pol10`).

Chapter 4

High dimensional data manipulation

The present chapter makes use of the concept previously introduced to perform computation that one would do with high dimensional data. Typically, if a given dataset consist of several 3D positions for each observation, one has to deal with many numbers. It is possible that grouping these vectors by pairs is relevant to understand the problem. Or maybe other operation within these various 3D vector is useful. Since we want to use the full power of numpy, all these computations cannot be done with an explicit loop over observations and/or over vectors.

This chapter consider few of these typical use cases and their implementation using numpy, using a simple toy dataset made by hand. Most likely, you will never face such a situation for machine learning algorithm, but it is good to go trough these examples to show some of the limitation of not being able to loop overs observations.

Let's first perform the usual imports:

```
import itertools
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
%matplotlib inline
```

Then, one can setup he default matplolib style for all following plots (more details on available option can be found on how to customize matplotlib):

```
mpl.rcParams['legend.frameon'] = False
mpl.rcParams['legend.fontsize'] = 'xx-large'
mpl.rcParams['xtick.labelsize'] = 16
mpl.rcParams['ytick.labelsize'] = 16
mpl.rcParams['axes.titlesize'] = 18
mpl.rcParams['axes.labelsize'] = 18
mpl.rcParams['lines.linewidth'] = 2.5
mpl.rcParams['figure.figsize'] = (10, 7)
```

4.1 Data model and goals

We consider 1 millions observations, each defined by ten 3D vectors $(r_0, ..., r_9)$ where $r_i = (x, y, z)$ (arrow for vector will be omitted from now on). These pseudo-data can represent position in space or RGB colors for an image. This is just an example to play with and apply numpy concepts for both simple computations (element-by-element functions, statistics calculations) and more complex computation exploiting the multi-dimensional structure of the data. For example, one might want to compute the distance between all pairs (r_i, r_j), which has to be done without loop.

Using the `np.random` module, it is possible to generate n-dimensional arrays easily. In our case, we want to generate an array containing our observations with have 3 dimensions (or *axis* in numpy language), and the size along each of these axis will have the following value and meaning:

- `axis=0`: over 1 million events
- `axis=1`: over 10 vectors
- `axis=2`: over 3 coordinates

```
r = np.random.random_sample((1000000, 10, 3))
```

It is possible to print the first two observations as follow:

```
print(r[0:2])
```

```
[[[3.12646048e-01 4.55257576e-01 7.62120920e-01]
  [7.57904883e-01 5.75783716e-01 9.85730373e-01]
  [8.58351019e-01 9.97112982e-01 6.94945548e-02]
  [3.97010641e-01 3.30452282e-01 4.76705513e-01]
  [1.90192515e-01 8.46642981e-01 9.44922049e-01]
  [2.28626376e-01 5.32713270e-01 5.86119632e-02]
  [8.78240385e-01 4.84309389e-01 5.41506300e-01]
  [9.04149582e-01 4.92954799e-01 2.21837932e-01]
  [7.92243462e-01 9.92160857e-01 5.22952886e-01]
  [5.90601463e-01 8.57334963e-01 5.76432781e-01]]

 [[4.89732288e-01 2.45947658e-01 5.24605965e-01]
  [2.79684077e-01 1.75887137e-01 5.96777979e-01]
  [6.36118572e-01 8.08656904e-01 5.67401037e-01]
  [5.01315803e-01 3.79415584e-01 9.73566504e-05]
  [8.04499847e-01 4.01132808e-01 2.73607136e-01]
  [4.62946717e-01 8.46061510e-01 8.82090460e-01]
  [2.30961828e-02 6.79642827e-01 4.79125888e-01]
  [7.51716668e-01 9.00985276e-01 6.87922769e-01]
  [9.47722015e-01 3.27310436e-01 1.49407680e-02]
  [3.36817391e-01 2.63926051e-01 6.90325842e-01]]]
```

4.2 Mean over the differents axis

4.2.1 Mean over observations (axis=0)

This mean will average all observations *i.e.* over the first dimension, returning an array of dimension `(10, 3)` corresponding to the average $r_i = (x_i, y_i, z_i)$ over the observations.

```
m0 = np.mean(r, axis=0)
print(m0.shape)
```

```
(10, 3)
```

Note the computation time of 30ms for 30 averages over a million number:

```
%timeit np.mean(r, axis=0)
```

```
29.9 ms ± 165 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

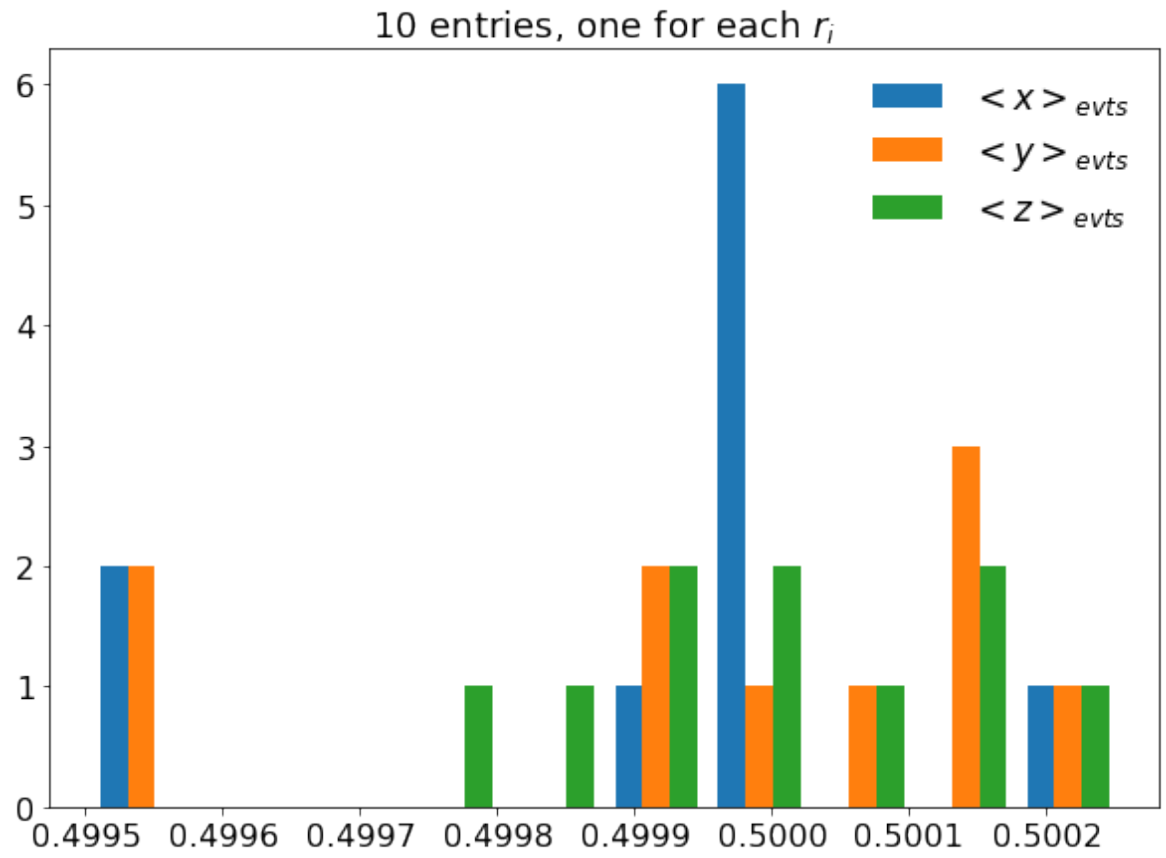
While it takes 10 times longer for a *single mean* over a million number with an explicit loop, so **the gain of vectorization is a factor 300**:

```
def explicit_loop(array):
    res=0
    for a in array:
        res += a/len(array)

%timeit explicit_loop(np.random.random_sample(size=1000000))
```

```
286 ms ± 1.32 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The distributions of `m0` obtained with `plt.hist()` results into three separate histograms (one for each x, y, x) each having 10 entries (one per r_i):

```
plt.hist(m0, label=['$<x>_{evts}$', '$<y>_{evts}$', '$<z>_{evts}$'])
plt.title('10 entries, one for each $r_i$')
plt.legend();
```



4.2.2 Mean over the 10 vectors (axis=1)

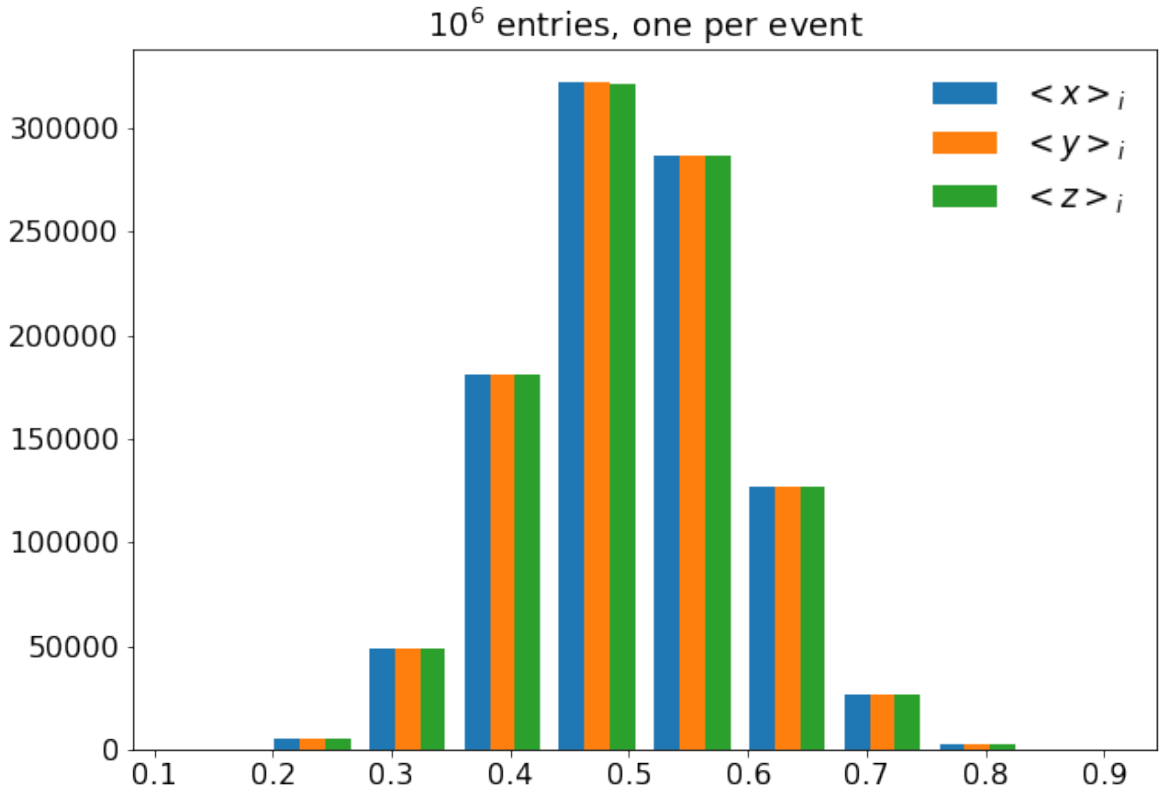
This one will compute the average over the 10 vectors, for each observations, reducing into a `(1000000, 3)` shape array, as seen below. This is 3D barycenter of each observation.

```
m1 = np.mean(r, axis=1)
print(m1.shape)
```

```
(1000000, 3)
```

One can plot the obtained array `m1` using `plt.hist()`, which results into 3 histograms of a million entry each:

```
plt.hist(m1, label=['$<x>_{i}$', '$<y>_{i}$', '$<z>_{i}$'])
plt.title('$10^6$ entries, one per event')
plt.legend();
```



4.2.3 Mean over the coordintates (axis=2)

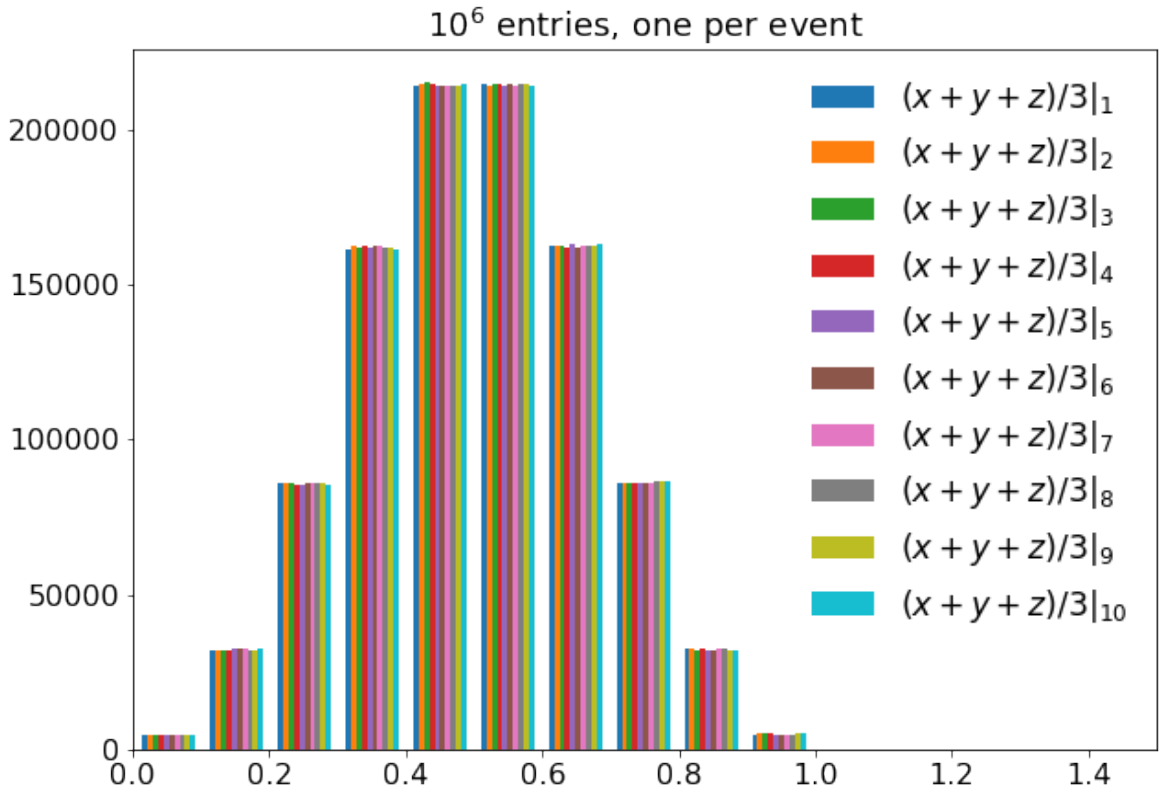
This directly computes the average over the three coordinates $(x + y + z)/3$ for each vector of each event, resulting in 10 values per event:

```
m2 = np.mean(r, axis=2)
print(m2.shape)
```

```
(1000000, 10)
```

The `plt.hist()` of the resulting array `m2` corresponds then to 10 histograms of a million entries each:

```
names = ['$(x+y+z)/3|_{'+'{}'.format(i)+'}$' for i in range(1, 11)]
plt.hist(m2, label=names)
plt.title('$10^6$ entries, one per event')
plt.xlim(0, 1.5)
plt.legend();
```



4.3 Distance computation

Computing particular distances inside a given event is relevant for many applications (distances here can be seen as any type of metric). For example, these computation are crucial in learning algorithms based on nearest neighbor approach. In collider physics, it's always useful to compute angle between two objects (tracks, deposit, particles, ...) in order to compute invariant masses, or isolation in a given cone, etc ...

4.3.1 Distance to a reference r_0

We can start simple by defining a new origin `r0`

```
r0 = np.array([1, 2, 1])
```

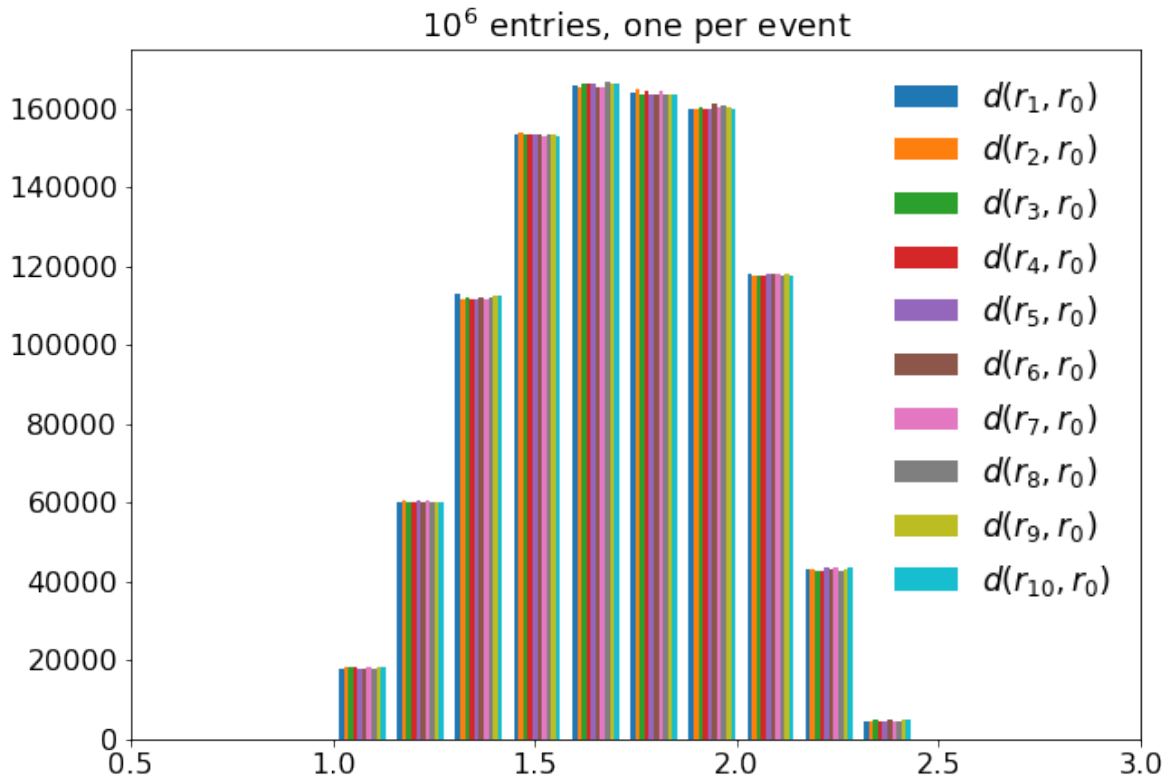
and compute the distance to this new origin for all points, using `**2` to square all numbers, perform the sum over the coordinate (`axis=2`) and square-root everything with `**0.5`:

```
d = np.sum((r-r0)**2, axis=2)**0.5
print(d.shape)
```

```
(1000000, 10)
```

As expected the result is 10 numbers for each of the events, which can be easily plotted:

```
names = ['$d(r_{'+'{}'.format(i)+'},r_0)$' for i in range(1, 11)]
plt.hist(d, label=names)
plt.title('$10^6$ entries, one per event')
plt.xlim(0.5, 3)
plt.legend();
```



4.3.2 Distance between r_i and $< r >_i$ for each event

Another calculation is to compute the averaged position for each event and see how distant each vector is from this position. To perform such a calculation, we will use numpy array broadcasting. Let's first compute the average position for every events:

```
r_mean = np.mean(r, axis=1)
```

Now, let's broadcast this array of shape `(1e6, 3)` with the full dataset, *i.e.* an array of shape `(1e6, 10, 3)`, by computing the distance for each point:

```
try:
    d_to_mean = np.sum((r-r_mean)**2, axis=2)**0.5
except ValueError:
    print('Impossible for {} and {}'.format(r.shape, r_mean.shape))
```

```
Impossible for (1000000, 10, 3) and (1000000, 3)
```

There is one missing dimension, describing the 10 positions, which has to be created so that the array can be copied 10 times over along this dimension:

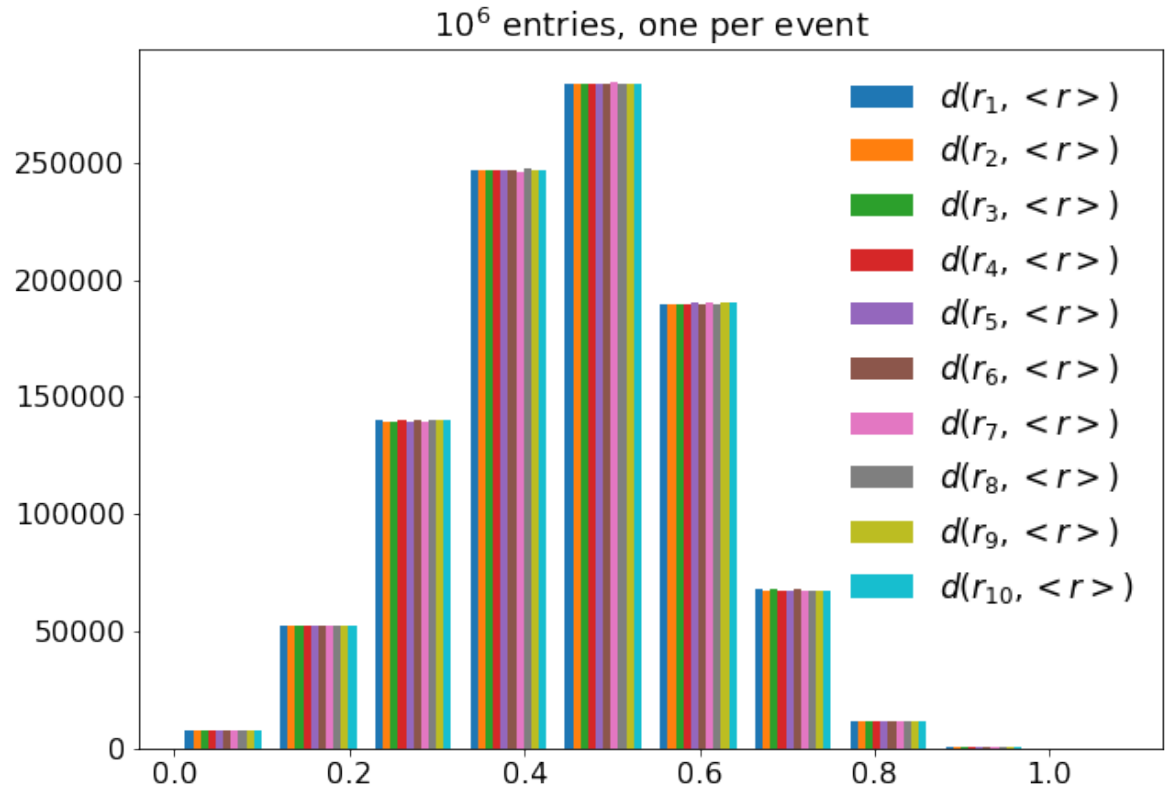
```
r_mean_3d = r_mean[:, np.newaxis, :]
```

We can now retry the operation:

```
try:
    d_to_mean = np.sum((r-r_mean_3d)**2, axis=2)**0.5
    print('Possible for {} and {}'.format(r.shape, r_mean_3d.shape))
except ValueError:
    print('Impossible for {} and {}'.format(r.shape, r_mean_3d.shape))
```

```
Possible for (1000000, 10, 3) and (1000000, 1, 3)
```

```
names = ['$d(r_{'+'{}'.format(i)+'},<r>)$' for i in range(1, 11)]
plt.hist(d_to_mean, label=names)
plt.title('$10^6$ entries, one per event')
plt.legend();
```



4.4 Pairing 3D vectors for each observation, without a loop

Being able to pair objects is obviously important for many type of calculations. This allows to probe correlations at the first order, to identify sub-systems, etc … In a traditional way, a pairing would involve a *for* loop in which the combinatorics can be done for each event. Working with numpy, one has to perform the combinatorics in a vectorized way and return a new numpy array containing all the pairs. Once done, one can perform many types of computations on this new array.

4.4.1 Finding all possible (r_i, r_j) pairs for all events

One solution to perform such a task without *for* loop was found on stackoverflow. The idea is to simply work on indices to build the pairs (since it doesn't really matter what are the nature of the objects), and use numpy *fancy* indexing. Let proceed step by step with a smallest array to understand the procedure (namely 2 observations of 5 positions):

```
a = r[0:2,0:5]
print(a)
```

```
[[[3.12646048e-01 4.55257576e-01 7.62120920e-01]
  [7.57904883e-01 5.75783716e-01 9.85730373e-01]
  [8.58351019e-01 9.97112982e-01 6.94945548e-02]
  [3.97010641e-01 3.30452282e-01 4.76705513e-01]
  [1.90192515e-01 8.46642981e-01 9.44922049e-01]]

 [[4.89732288e-01 2.45947658e-01 5.24605965e-01]
  [2.79684077e-01 1.75887137e-01 5.96777979e-01]
  [6.36118572e-01 8.08656904e-01 5.67401037e-01]
  [5.01315803e-01 3.79415584e-01 9.73566504e-05]
  [8.04499847e-01 4.01132808e-01 2.73607136e-01]]]
```

Since we want to work with the indicies of the 5 vectors, we create a numpy array of integer going from 0 to 4 (`a.shape[1]` is the number of elements along the second dimension, *i.e.* 5):

```
array_indices = np.arange(a.shape[1])
print(array_indices)
```

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

Then, we use the package `itertools` to deal with the combinatorics. This will return an *iterator* that can be turned into a numpy array using `np.fromiter()`. But this function requires to specify the data type `dt`, which is done using a structured array synthax here (*i.e.* `[(varName1,type1), (varName2,type2)]`). For more details on data type, check this documentation page.

```
dt = np.dtype([('index1', np.intp), ('index2', np.intp)])
print(dt)
```

```
[('index1', '<i8'), ('index2', '<i8')]
```

```
array_indice_comb = np.fromiter(itertools.combinations(array_indices, 2), dt)
print(array_indice_comb)
```

```
[(0, 1) (0, 2) (0, 3) (0, 4) (1, 2) (1, 3) (1, 4) (2, 3) (2, 4) (3, 4)]
```

The next step is to format these numbers in a indices array with the proper dimension, so that when we do `a[indices]`, we get all the pairs. For instance, we need to have all 10 pairs, each with two elements corresponding to a shape `indices.shape=(10,2)`. We can achieved this in two steps:

1. `array_indice_comb.view(np.intp)` return the exact same data of `array_indice_comb` as a 1D array of positive integer.
2. we reshape the resulting array with `reshape(-1, 2)`, where -1 means "compute the size of the first dimension to have 2 objects (we wants pair!) in the second dimension.

```
indices = array_indice_comb.view(np.intp).reshape(-1, 2)
print(indices)
```

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

The final steps is exploit fancy indexing along `axis=1` *i.e.* the 5 spatial positions. In practice, for each observation `iobs`, we want to have `a[iobs, indices]`. There are two ways to do this: (a) `a[:, indices]` or (b) using the numpy function `np.take(a, indices, axis)` which makes the code more independant from the structure of `a`:

```
a_pairs = np.take(a, indices, axis=1)
print(a_pairs.shape)
```

```
(2, 10, 2, 3)
```

```
a_pairs = a[:,indices]
print(a_pairs.shape)
```

```
(2, 10, 2, 3)
```

We have now 2 events, each having 10 pairs, each having 2 objects (still a pair!), each having 3 coordinates (spatial positions). We can print all the 10 pairs for the first observation:

```
print(a_pairs[0])
```

```
[[[0.31264605 0.45525758 0.76212092]
  [0.75790488 0.57578372 0.98573037]]

 [[0.31264605 0.45525758 0.76212092]
  [0.85835102 0.99711298 0.06949455]]

 [[0.31264605 0.45525758 0.76212092]
  [0.39701064 0.33045228 0.47670551]]

 [[0.31264605 0.45525758 0.76212092]
  [0.19019251 0.84664298 0.94492205]]

 [[0.75790488 0.57578372 0.98573037]
  [0.85835102 0.99711298 0.06949455]]

 [[0.75790488 0.57578372 0.98573037]
  [0.39701064 0.33045228 0.47670551]]

 [[0.75790488 0.57578372 0.98573037]
  [0.19019251 0.84664298 0.94492205]]

 [[0.85835102 0.99711298 0.06949455]
  [0.39701064 0.33045228 0.47670551]]

 [[0.85835102 0.99711298 0.06949455]
  [0.19019251 0.84664298 0.94492205]]

 [[0.39701064 0.33045228 0.47670551]
  [0.19019251 0.84664298 0.94492205]]]
```

Once understood, we can wrapp-up this code into a function where we generalize the number of objects we want to group `n` and the axis along which we want to group `axis`:

```
def combs_nd(a, n, axis=0):
    i = np.arange(a.shape[axis])
    dt = np.dtype([('', np.intp)]*n)
    i = np.fromiter(itertools.combinations(i, n), dt)
    i = i.view(np.intp).reshape(-1, n)
    return np.take(a, i, axis=axis)
```

As a sanity check, we can re-compute `a_pair` and compare with the previous results:

```
a_pairs = combs_nd(a=r[0:2,0:5], n=2, axis=1)
print(a_pairs[0])
```

```
[[[0.31264605 0.45525758 0.76212092]
  [0.75790488 0.57578372 0.98573037]]

 [[0.31264605 0.45525758 0.76212092]
  [0.85835102 0.99711298 0.06949455]]

 [[0.31264605 0.45525758 0.76212092]
  [0.39701064 0.33045228 0.47670551]]

 [[0.31264605 0.45525758 0.76212092]
  [0.19019251 0.84664298 0.94492205]]

 [[0.75790488 0.57578372 0.98573037]
  [0.85835102 0.99711298 0.06949455]]

 [[0.75790488 0.57578372 0.98573037]
  [0.39701064 0.33045228 0.47670551]]

 [[0.75790488 0.57578372 0.98573037]
  [0.19019251 0.84664298 0.94492205]]

 [[0.85835102 0.99711298 0.06949455]
  [0.39701064 0.33045228 0.47670551]]

 [[0.85835102 0.99711298 0.06949455]
  [0.19019251 0.84664298 0.94492205]]

 [[0.39701064 0.33045228 0.47670551]
  [0.19019251 0.84664298 0.94492205]]]
```

It can be intersting to see that this operation takes less than a second for a million observations of 10 vectors, meaning 45 pairs:

```
%timeit combs_nd(a=r, n=2, axis=1)
```

```
966 ms ± 33.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

4.4.2 Computing (minimum) distances on these pairs

Once we have these pairs, we can for example computes all the distances and find which pair has the closest objects. Starting with the pairs:

```
pairs = combs_nd(a=r, n=2, axis=1)
```

We can then define the vectorial difference between the two position of a pair, and compute the euclidiean distance:

```
dp = pairs[:, :, 0, :]-pairs[:, :, 1, :]
distances = (np.sum(dp**2, axis=2))**0.5
```

And get the minimum distance for each event:

```
smallest_distance = np.min(distances, axis=1)
print(smallest_distance.shape)
```

```
(1000000,)
```

All these instructions can be put into a function which can be timed:

```
def compute_dr_min(a):
    pairs = combs_nd(a, 2, axis=1)
    i1 = tuple([None, None, 0, None])
    i2 = tuple([None, None, 1, None])
    return np.min(np.sum((pairs[i1]-pairs[i2])**2, axis=2)**0.5, axis=1)
```

```
%timeit compute_dr_min(r)
```

```
980 ms ± 122 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Note that doing all operations in the less possible amount of lines can significantly speed up the process. Let's define another function where the difference between the pair elements is done separately:

```
def compute_dr_min_more_steps(a):
    pairs = combs_nd(a, 2, axis=1)
    dp = pairs[:, :, 0, :]-pairs[:, :, 1, :]
    return np.min(np.sum(dp**2, axis=2)**0.5, axis=1)
```

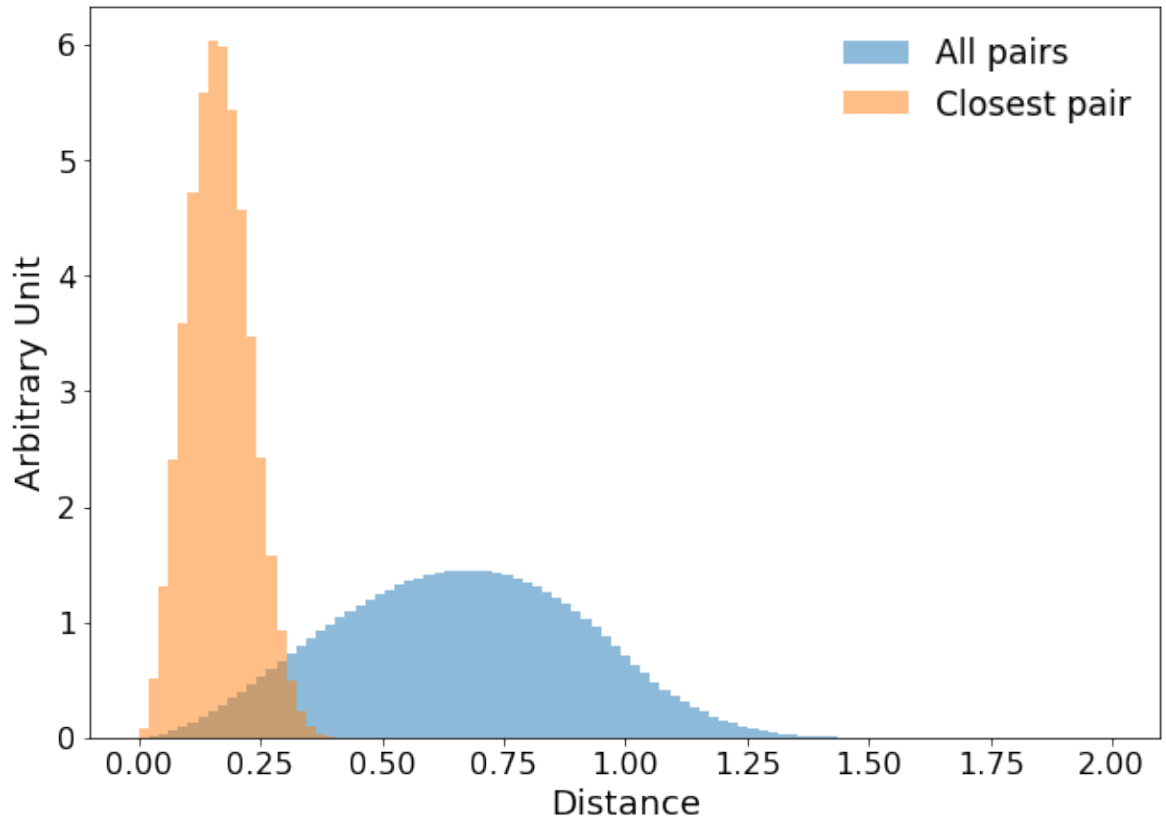
And let's compare the performance on 0.2 million observations:

```
%timeit compute_dr_min(a=r[:200000])
%timeit compute_dr_min_more_steps(a=r[:200000])
```

```
194 ms ± 1.77 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
606 ms ± 5.13 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Let's now plot the distributions of all distances for all the pairs (using `flatten()` function which returns a 1D array), and only the pair having the smallest distances:

```
plot_style = {
    'bins': np.linspace(0, 2, 100),
    'alpha': 0.5,
    'density': True,
}

plt.hist(distances.flatten(), label='All pairs' ,**plot_style)
plt.hist(smallest_distance, label='Closest pair', **plot_style)
plt.xlabel('Distance')
plt.ylabel('Arbitrary Unit')
plt.legend();
```



4.5 Selecting a subset of r_i based on (x, y, z) values, without loop

The next step in our exploration "loop-less calculations" is to be able to perform the same kind of computation described above but only on a subset of positions, selected according to a given criteria. For example, we might want to keep particles only if there have positive charge. Many obvious application can be found in other physics field and/or machine learning. Let's start with accessing the three arrays of coordinates in order to select points based on some easy criteria.

```
x, y, z = r[:, :, 0], r[:, :, 1], r[:, :, 2]
```

4.5.1 Counting number of points amont the 10 with $x_i > y_i$ in each event

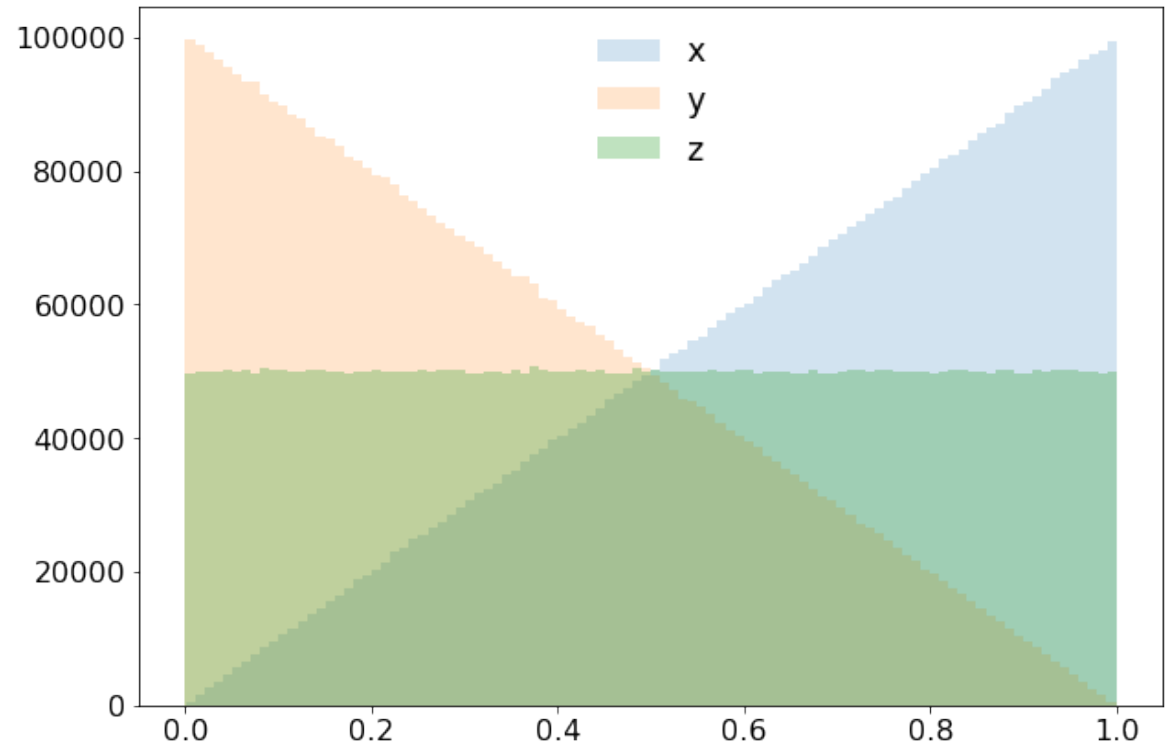
We will use the numpy masking feature described in the first chapter, defining a index of boolean based on `x` and `y` arrays:

```
idx = x > y
print(idx.shape)
```

```
(1000000, 10)
```

We can quickly check the distribution for the selected coordinates: x and y are anty correlated - as expected - while z is flat - as expected.

```
plt.hist(x[idx], bins=100, alpha=0.2, label='x')
plt.hist(y[idx], bins=100, alpha=0.2, label='y')
plt.hist(z[idx], bins=100, alpha=0.3, label='z')
plt.legend();
```



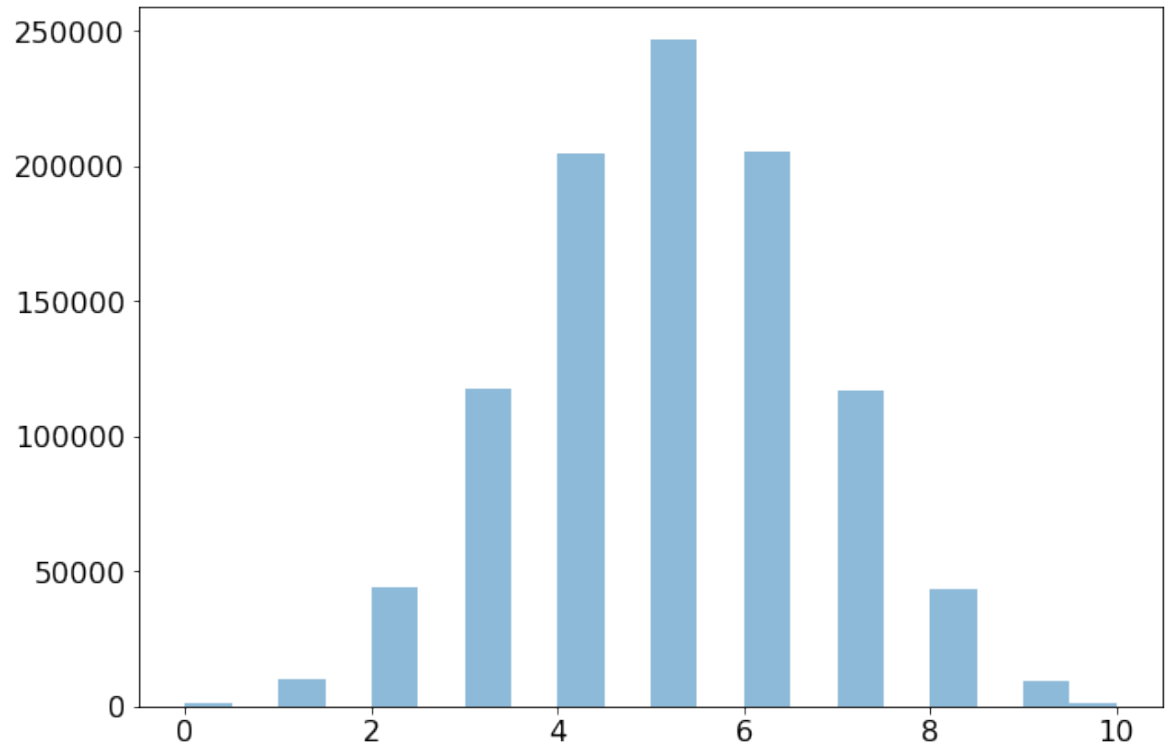
If we want to better understand how this selection affect our data, one might want to count the number of points per event satisfying this selection, using `np.count_nonzero()` on the boolean array along the axis representing the 10 vectors `axis=1`:

```
c = np.count_nonzero(idx, axis=1)
print(c.shape)
```

```
(1000000,)
```

We can then plot the distribution of this number over all the events:

```
plt.hist(c, bins=20, alpha=0.5);
```



4.5.2 Plotting z for the two types of population ($x > y$ and $x < y$)

This is obviously useful to inspect the different populations - something we want to do very often. For the plotting purpose, let's consider only the 500 first observations that we dump into `sx`, `sy`, `sz` (`s` for small):

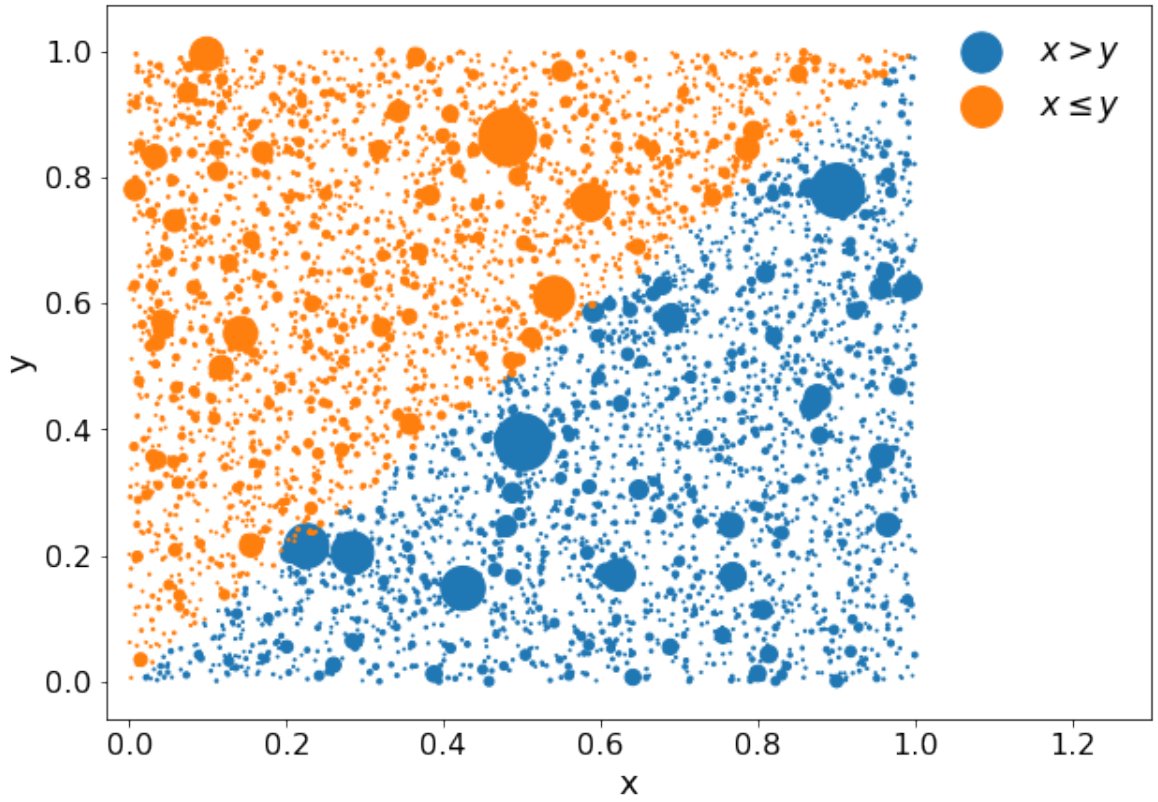
```
sx, sy, sz = x[0:500, ...], y[0:500, ...], z[0:500, ...]
```

We define the mask computed on these small arrays `smask`:

```
smask = sx>sy
```

And we can plot the result in the 2D plane (x, y) with the z coordinate as marker size, for instance $1/(z + 10^{-3})$. The two populations are defined using both `smask` and `~smask` to make sure the union of the two is the original dataset:

```
plt.scatter(sx[smask], sy[smask], s=(sz[smask]+1e-3)**-1, label='$x>y$')
plt.scatter(sx[~smask], sy[~smask], s=(sz[~smask]+1e-3)**-1, label='$x\leq y$')
plt.xlabel('x')
plt.ylabel('y')
plt.xlim(-0.03, 1.3)
plt.legend();
```



4.5.3 Computation of $x_i + y_i + z_i$ sum over a subset of the 10 positions

Once we are able to isolate a subset of points, we might to computes new numbers only based on those. This is what is proposed here with the sum of the three coordintates. Let's first compute and the sum, called `ht`, over all the 10 points:

```
ht1 = np.sum(x+y+z, axis=1)
print(ht1.shape)
```

```
(1000000,)
```

Apply now a selection, which multiply the value by 0 (*i.e.* `False`) if the condition is not satistifed:

```
selection = x>y
ht2 = np.sum((x+y+z)*selection, axis=1)
```

Of course, this works only for computation which is not affected by a 0: if we want to compute the product of coordinate, this approach will obvioulsy not work.

```
prod = np.product((x+y+z)*selection, axis=1)
eff = np.count_nonzero(prod>0)/len(prod)
print('Efficiency of prod>0: {:.5f}'.format(eff))
```

```
Efficiency of prod>0: 0.00093
```

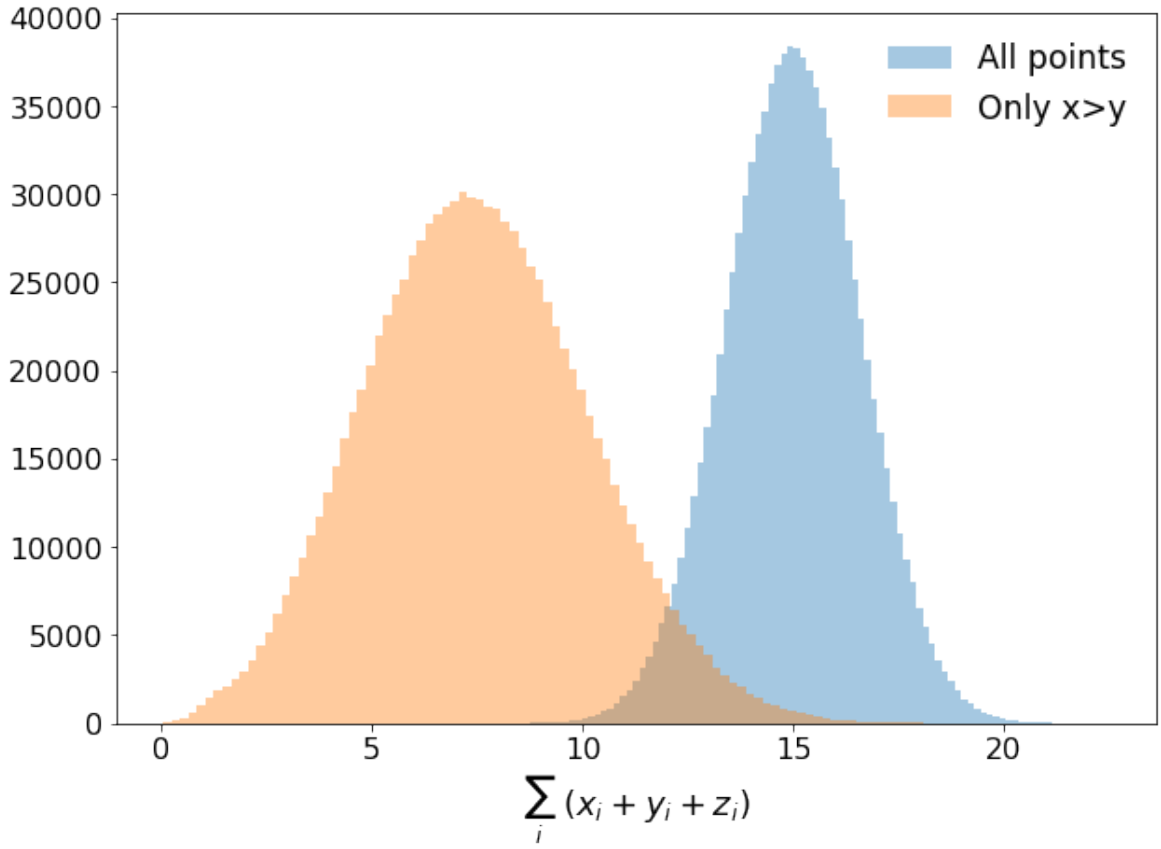
In a more general manner, we should use *masked arrays* which completely remove the masked elements from any computations:

```
mx = np.ma.array(x, mask=selection)
my = np.ma.array(y, mask=selection)
mz = np.ma.array(z, mask=selection)
prod = np.product((mx+my+mz), axis=1)
eff = np.count_nonzero(prod>0)/len(prod)
print('Efficiency of prod>0: {:.5f}'.format(eff))
```

```
Efficiency of prod>0: 1.00000
```

Finally one can plot the result, removing the observation with `ht2==0` (case where all the 10 points have $x \leq y$):

```
plt.hist(ht1, bins=100, alpha=0.4, label='All points')
plt.hist(ht2[ht2>0], bins=100, alpha=0.4, label='Only x>y')
plt.xlabel('$\sum_{i} \; (x_i+y_i+z_i)$')
plt.legend();
```



4.5.4 Pairing with a subset of r_i verifying $x_i > y_i$ only

Another computation would be to redo the pairing on the subset of selected position. In order to do so, we follow the same logic, expect that we will directly replace removed values by `nan` in order to be easily identifiable in after the pairing. It's *very important to copy the orignal data with the module* `copy`, otherwise, the orignal data will be modifed in the following piece of code:

```
import copy
selection = x>y
selected_r = copy.copy(r)
selected_r[selection] = np.nan
print(selected_r[0])
```

```
[[0.31264605 0.45525758 0.76212092]
 [        nan         nan         nan]
 [0.85835102 0.99711298 0.06949455]
 [        nan         nan         nan]
 [0.19019251 0.84664298 0.94492205]
 [0.22862638 0.53271327 0.05861196]
 [        nan         nan         nan]
 [        nan         nan         nan]
 [0.79224346 0.99216086 0.52295289]
 [0.59060146 0.85733496 0.57643278]]
```

On can now calling the paring function on the filtered dataset:

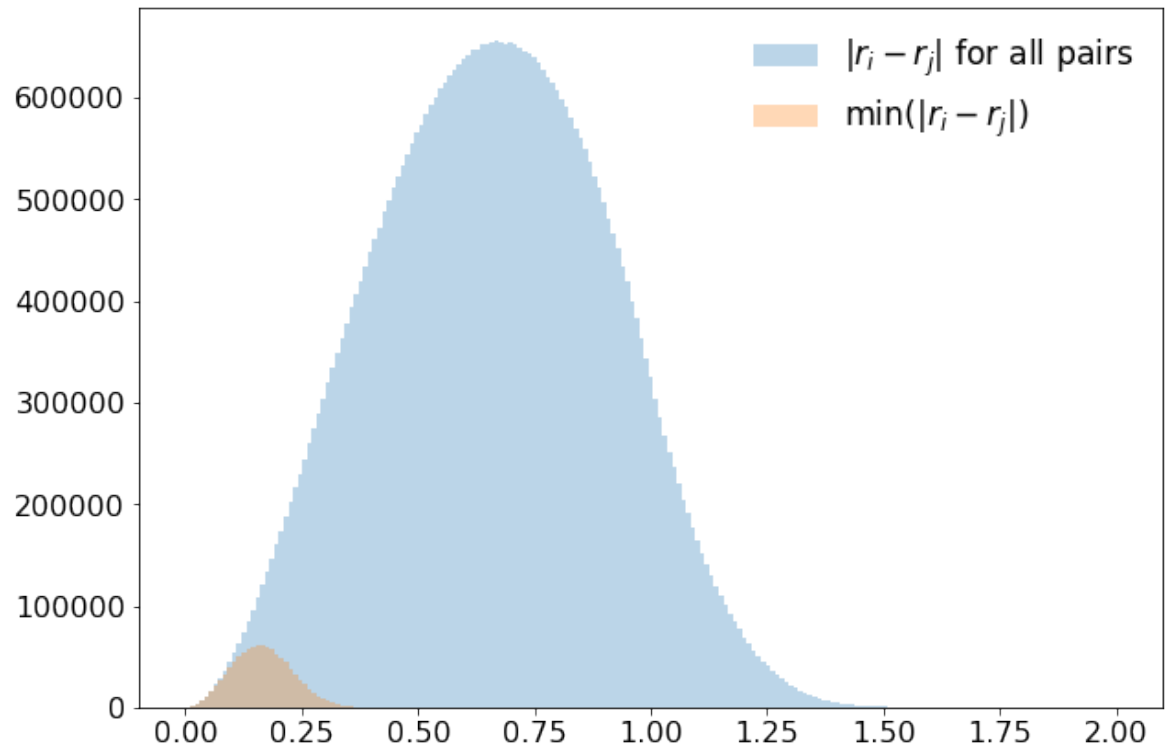
```
selected_pairs = combs_nd(selected_r, n=2, axis=1)
```

And compute the distances, but replacing back the `np.nan` by a default values that will not be seen on a plot.

```
p1, p2 = pairs[:, :, 0, :], pairs[:, :, 1, :]
dp = np.sum((p1-p2)**2, axis=2)**0.5
dp[np.isnan(dp)] = 999
```

And plotting the distributions of both all distances and minimum distances for pairs made out of points verifying $x > y$:

```
plot_style = {'bins': np.linspace(0, 2, 200), 'alpha':0.3}
plt.hist(dp.flatten(), label='$|r_i-r_j|$ for all pairs', **plot_style)
plt.hist(np.min(dp, axis=1), label='min$(|r_i-r_j|)$', **plot_style)
plt.legend();
```



4.6 Some comments

Manipulating numpy array is quite powerful and fast for both computation and plotting, at the condition that we use numpy optimization, namely vectorization, indexing and broadcasting. This is ofter possible when this has also some limitations as we saw above. Namely, we add to play a bit with “patchwork approaches” to

achieve what we want without loops in the last two sections. Typically, what will work for one computation will not work for another (replacing rejected values by 0 works for an addition and not for a product). For the pairing as well, we had to replace all rejected values by `np.nan` in order to filter them later on. This kind of practice makes things less readable when complexity increases - according to me. Or maybe there are smarter ways to do things.