Python for Finance
Lecture Notes

Matteo Sani

Quants Staff - MPS Capital Services
matteo.sani@mpscapitalservices.it

Chapter 1

Introduction to python

`Python` is one of the most widely used programming languages, and it has been around for more than 28 years now.

First and foremost reason why `python` is much popular because it is highly productive as compared to other programming languages like `C++` and `java`. It is a much more concise and expressive language and requires less time, effort, and lines of code to perform the same operations.

This makes `python` very easy-to-learn programming language even for beginners and newbies. It is also very famous for its simple programming syntax, code readability and English-like commands that make coding in Python lot easier and efficient. With `python`, the code looks very close to how humans think. For this purpose, it must abstract the details of the computer from you. Hence, it is slower than other "lower-level language" like `C`.

There were times when computer run time was to be the main issue and the most expensive resource. But now, things have changed. Computer, servers and other hardware have become much much cheaper than ever and speed has become a less important factor. Today, development time matters more in most cases rather than execution speed. Reducing the time needed for each project saves companies tons of money.

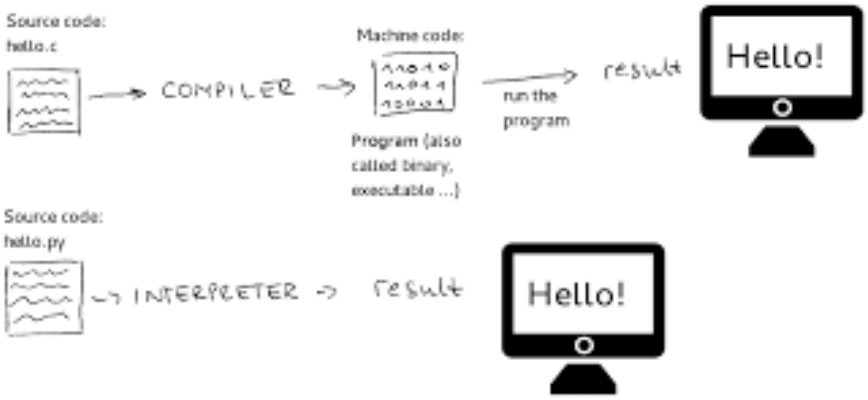
As far as the execution speed or performance of the program is concerned, we can easily manage it by horizontal scaling, means getting more servers running to get that level of speed or performance.

In short, `python` is widely used even when it is somehow slower than other languages because:

- is more productive;
- companies can optimize their most expensive resource: employees;
- rich set of libraries and frameworks;
- large community.

1.1 What is `python` ?

`Python` is a so called *interpreted language*: it takes some code (a sequence of instructions), reads and executes it. This is different from other programming languages like `C` or `C++` which *compile* code into a language that the computer can understand directly (*machine language*).



Interpreted vs compiled language

As a result, `python` is essentially an *interactive* programming language, you can program and see the results almost at the same time. This is very nice for a faster development since compilation time can be quite long (just to give an idea the compilation of our `C++` financial code takes more than one hour). However there are drawbacks in terms of performance, the *translation* to machine language has to be done in real-time resulting in slower execution times.

High-level program

```
class Triangle {
  ...
  float surface()
    return b*h/2;
  }
```

Low-level program

```
LOAD r1,b
LOAD r2,h
MUL r1,r2
DIV r1,#2
RET
```

Executable Machine code

```
0001001001000101
0010010011101100
10101101001...
```

Human readable vs machine code

In the next chapters we'll take a quick tour of `python` and see the main features and characteristics of this programming language, later on we will see how it can be useful to solve real-world finance problems.

First of all since `python`, as basically all programs, comes in different version and flavours we need to specify the particular one we are going to use. The latest version (at the time I'm wrtiting this pages) is 3.8.5, but it is continously evolving, however it is not difficult to see older versions floating around (e.g. 2.7). This is because there are some big differences between `python2.X` and `python3.X` which prevent a sizeable portion of `python2` users to stick with it (consider that moving to `python3` would require a large amount of work to adapt big projects). In conclusion we will concentrate on `python3.7`.

1.2 Python basics

Every language has *keywords*, these are reserved words that have a special meaning and tell the computer what to do. The first one we encounter is `print`: it prints to screen whatever is specified between the parenthesis.

```
print ("Hello world !")

Hello world !
```

```
print ("Welcome")
print ("to")
print ("everybody")

Welcome
to
everybody
```

Good programming practice recommends to document the code you write (you will soon see that it is surprisingly easy to forget what you wanted to do in your code). In `python` you can add comments to code starting a line with a hash character (#).

```
print ("Ciao") # this is a comment

Ciao
```

1.2.1 Variables

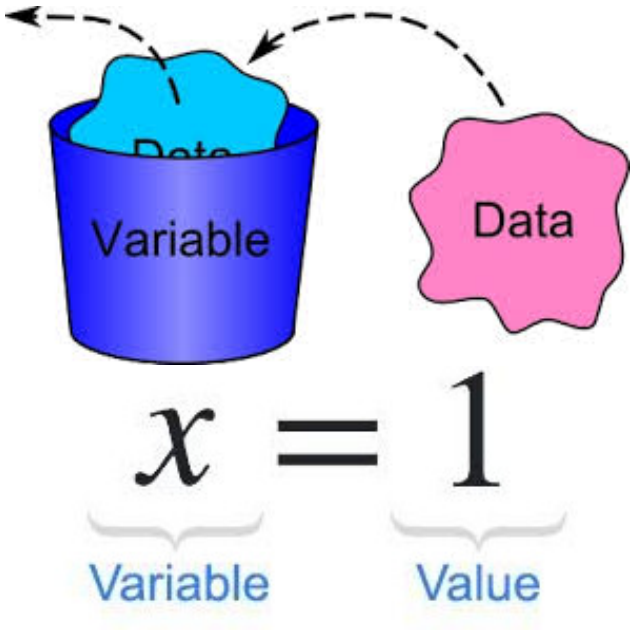
A variable is a computer memory location paired with an associated symbolic name, which contains some quantity of information referred to as a *value* (e.g. a number, a string...). Variables and hence data they contain, can be used, referenced and manipulated throughout a program. A value is assigned to a variable with the equal operator (=).

```
x = 9
print (x)

9
```

```
myphone = "Huawei P10Lite"
print (myphone)

Huawei P10Lite
```

Another very useful keyword is `type`: it tells which kind of object is stored in a variable.

```
print (type(x))
print (type(myphone))

<class 'int'>
<class 'str'>
```



Graphical representation of a variable.

After their definitions `x` and `myphone` can be used as aliases for a number and a string and their content manipulated, for example:

```
print (x+5)

14
```

There are rules that limit the variable naming possibilities, in particular they must:

- begin with a letter (`myphone`) or underscore (`_myphone`);
- other characters can be letters, numbers or more _;
- variable names are case-sensitive so myphone and myPhone are two distinct variables;

Keywords, as said, are reserved words and as such cannot be used as variable names (e.g. `print`, `type`, `for`...**).**

To use **good** variable names (and make you programs clearer and easier to read) always choose meaningful names instead of short names (i.e. `numberOfCakes` is much better than simply `n`), try to be consistent with your conventions (e.g. choose once and for all between `number_of_cakes` or `numberofcakes` or `numberOfCakes`), usually begin a variable name with underscore (_) only for a special case (will see later when this is usually done).

1.2.2 Boolean expressions

Boolean expressions evaluate to `true` or `false` only. This type of expressions usually involve logical or comparison operators like `or`, `and`, > (greater than), < (less than)... The equal boolean operator symbol is a double = (==), to not be confused with the assignement operator single = (=), with the first we compare two variables, with the second we associate a value to a variable.

Let's see some example. The following expression answer the question is 1 equal to 2:

```
1 == 2
```

```
False
```

Here another example using the not equal operator (!=):

```
True
```

```
2 < 2
```

```
False
```

```
2 <= 2  # in this case we allow the numbers to be equal too
```

```
True
```

```
print (x)
15 <= x and x <= 20 # this expression could also be written as 15 <= x <= 20
```

```
11
False
```

```
15 <= x or x <= 20
```

```
True
```

```
not (x > 20) # the not keyword negates the following expression
```

```
True
```

1.2.3 String expressions

A "string" is a sequence of characters (letters, digits, spaces, punctuation...). There are many operations that can be performed on strings, like for example concatenate (with + operator), truncate, replace characters...

```
mystring = "some text with punctuation, spaces and digits 10"
mystring.replace("s", "z")
```

```
'zome text with punctuation, zpacez and digitz 10'
```

```
"abc" + "def" # it is possible to concatenate strings with +
```

```
'abcdef'
```

```
"The number " + 4 + " is my favourite number"
# this causes an error since we are trying to concatenate a string
# with a number so two different kind of objects
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)
```

```
<ipython-input-33-b9f65c5a45f7> in <module>()
----> 1 "The number " + 4 + " is my favourite number"
      2 # this causes an error since we are trying to concatenate a string
      3 # with a number so two different kind of objects

TypeError: can only concatenate str (not "int") to str
```

To avoid this error is possible to **cast** an object to a different type which means to convert an object to a different type. In this case we can *force* the number four to be represented as a string with the `str()` function:

```
"The number " + str(4) + " is my favourite number"

'The number 4 is my favourite number'
```

```
print (type(3.4))
print (type(str(3.4)))

<class 'float'>
<class 'str'>
```

In this simple case everything worked fine but type casting is not always possible: for example a number can be converted to a string (e.g. from the integer 4 to the actual symbol "4") but the opposite is not possible (e.g. cannot convert the string "matteo" to a meaningful number). In this second case we can try to use the function `int()` to convert a string to an integer.

```
int("matteo")

---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-17-979283bb65e4> in <module>
----> 1 int("matteo")

ValueError: invalid literal for int() with base 10: 'matteo'
```

```
int("4")

4
```

Pretty string formatting: in order to get prettier strings than those obtained just concatenating with the + operator, `python` allows to format text using the following syntax "text other text".format(variable). With this notation, each is mapped to the variables listed in the `format` statement, the optional characters inside the curly brackets can determine the resulting format, for example in the following code : `.1f` means that this variable is float number and that has to be printed with 1 digit only after the decimal separator.

```
"The speed of light is about {:.1f} {}".format(299792.458, "km/s")
```

```
'The speed of light is about 299792.5 km/s'
```

In addition `format` allows for 0-padding of numbers, left or right alignement of text columns and so on.

1.2.4 Mathematical expressions

Below few examples of the basic mathematical expressions available in `python`.

```
1 + 2
```

```
3
```

```
40 - 5
```

```
35
```

```
x * 20 # remember that we set x equal to 9
```

```
180
```

```
x / 4
```

```
2.25
```

```
print (type(2.25))
```

```
<class 'float'>
```

```
x // 4 # interger division - result will be truncated to the
       # corresponding integer (no rounding)
       # 11 / 3 = 3.666666 -> 11 // 3 = 3
```

```
2
```

```
y = 3
x ** y # x to the power of y
```

```
729
```

```
3 * (x + y)
```

```
36
```

As an example of variable manipulation let's try to increment `x` by 1 and save the result again in `x`.

```
print (x)
x = x + 1
print (x)
```

```
15
16
```

Sometimes the increment of a variable plus the assignment to the same variable is written with a more compact syntax `x += 1` (this is also true for other operators e.g. `x *= 2`).

More complex mathematical functions are not directly available, let's see for example the logarithm:

```
log(3)

---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-17-ffde4d60496a> in <module>()
    ----> 1 log(3) # causes an error because the logarithm function
          2        # is not available by default


NameError: name 'log' is not defined
```

1.3 Modules

One very important feature of each language is the ability to reuse code among different programs, e.g. imagine how awful would be if you had to reimplement every time you need it a function to compute the logarithm. Usually there are mechanisms that allow to collect useful routines in *packages* (or *libraries*, or *modules*) so that later they can be called and used by any program may need them.

These collections of utilities in `python` are called *modules* and each installation of this language brings with it a standard set of them. If you need more functionality, you can download more modules from the web (there are zillions out there) or if you are not satisfied with what you found you can write your own (which is one the goal of this course in the end).

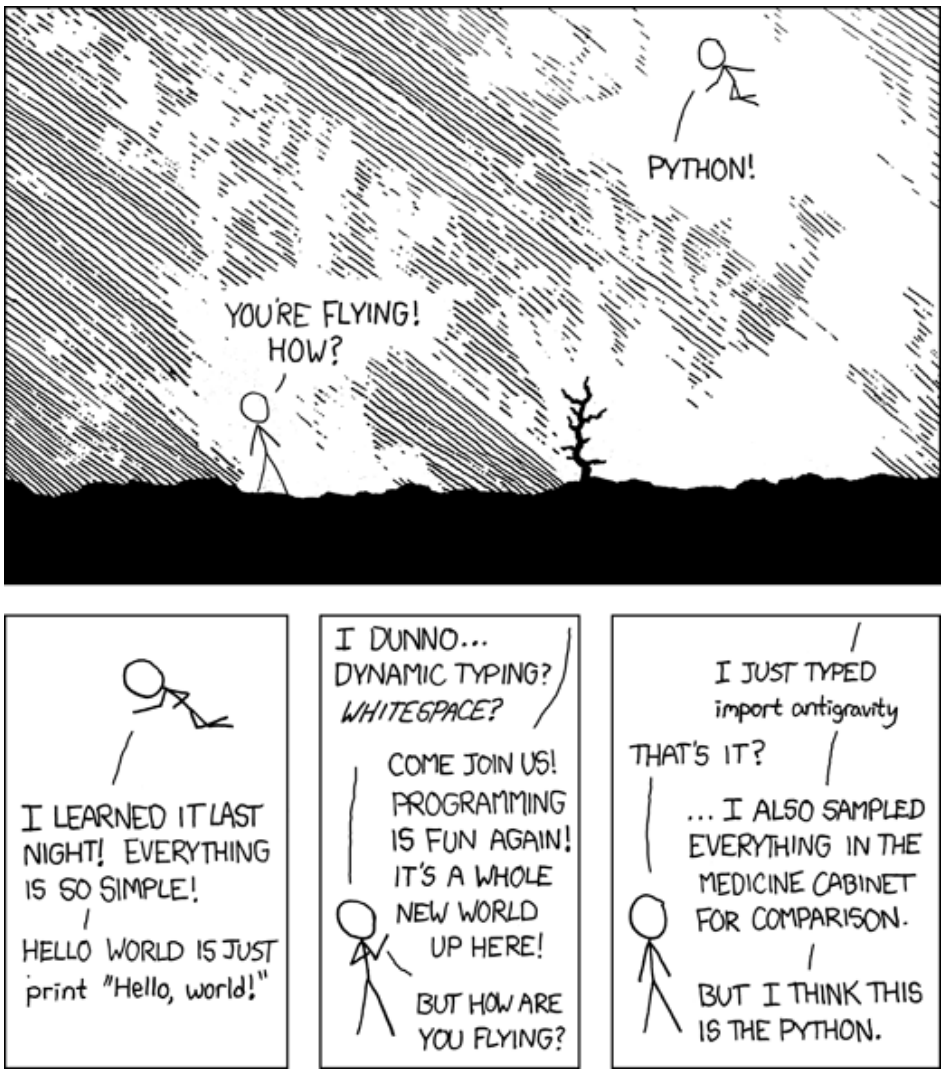
Some examples of useful modules we will use are:

- Numpy - which provides matrix algebra functionality and much more;
- Scipy - which provides a whole series of scientific computing functions;
- Pandas - which provides tools for manipulating time series or dataset in general;
- Matplotlib - for plotting graphs;
- Jupyter - for notebooks like this one.

Later we will take a closer look at three modules which are quite useful in financial analysis.

In order to load a module in a `python` program you can use the `import` keyword. To inspect a module (to understand which are its functionalities) it can be used the `help` and `dir` keywords: the first write a help message which usually describes the functionalities of a module, the latter list all the available functions of a module. **In order to access a function of a module you have to use the . (dot) operator: module-name.function-name.**

Let's see an example dealing with the`math` module which implements the most common mathematical functions.



`Python` has many modules for download on the web...

```
import math
dir(math)

Out[18]: ['__doc__',
          '__loader__',
          '__name__',
          '__package__',
          '__spec__',
          'acos',
          'acosh',
          'asin',
          'asinh',
          'atan',
          'atan2',
          'atanh',
          'ceil',
          'copysign',
          'cos',
          'cosh',
...
```

```
help(math)

Help on module math:

NAME
    math

MODULE REFERENCE
    https://docs.python.org/3.6/library/math

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.
...
```

```
math.log(3)

1.0986122886681098
```

```
math.exp(3)

20.085536923187668
```

```
print (type(math.log)) # yet another type: builtin function
print (type(math.log(3)))

<class 'builtin_function_or_method'>
<class 'float'>
```

If we want to avoid to type "math." every time we compute a logarithm or an exponential, we can just import the needed functions from a module using the following syntax:

```
from math import log, exp
print (log(3))
print (exp(3))

1.0986122886681098
20.085536923187668
```

As an example let's compute the interest rate r that produces a return R of 11000 Euro when investing 10000 Euro for 2 years:

$R = Ne^{r\tau} \rightarrow r = \frac{1}{\tau}\log(\frac{R}{N})$

```
rate = (1/2)*log(11000/10000)
print (rate)

0.04765508990216247
```

1.4 Indented blocks and the `if/else` statement

Unlike other languages which uses parenthesis to isolate blocks of code `python` uses indentation. A first example of this is given by `if/then` statements. Such commands allow to dynamically run different blocks of code based on certain conditions. For example in the following we print different statements according to the value of `x`, note the that the block of code to be run according each condition is shifted (i.e. indented) with respect to the rest of the code:

```
print (x)
if x == 1:
    print ("This will not be printed")
    # the block of code that is run if the first condition is met is indented
elif x == 15:
    print ("This will not be printed either")
    # again the block of code that is run here is indented
    # to be "isolated" by the rest
```

```
else:
    print ("This *will* be printed")

16
This *will* be printed
```

If by mistake the indentation of a block is missing an error is raised:

```
if x == 1:
print ("This will not be printed")
elif x == 15:
    print ("This will not be printed either")
else:
    print ("This *will* be printed")

File "<ipython-input-38-4535a45a6419>", line 3
  print ("This will not be printed")
    ^
IndentationError: expected an indented block
```

Below another example:

```
if x != 1:
   print ("x does not equal to 1")

x does not equal to 1
```

Just for comparison this is the same code written in C++:

```
if (x == 1) {
 print ("This will not be printed");
}
else if (x == 15) {
  print ("This will not be printed either");
}
else {
print ("This *will* be printed");
}
```

N.B. Notice how indentation doesn't matter at all here since the blocks are enclosed and defined by the brackets.

1.5 Loops

Another very important feature of a language is the ability to repeatedly run the same block of code many times. These is called looping and in `python` can be done with `for` or `while` keywords.

1.5.1 for

In a `for` loop we specifiy the set (or interval) over which we want to loop and a variable will assume all the values in that set (or interval). For example let's assume we want to print all the numbers between 25 and 30 excluded (here the keyword `range` returns the list of integers between the specified limits, if the first limit is not specified 0 is assumed):

```
for i in range(25, 30):
     print (i)

25
26
27
28
29
```

At each cycle of the loop the variable `i` takes one of the values between 25 and 31. With `range` it is also possible to specify the step, so that the loop can jump every 2 units or to go in descending order:

```
for i in range (30, 25, -1):
    print (i)

30
29
28
27
26
```

If it is needed to skip values in the loop the `continue` keyword can be used; in the code below 5 is actually missing from the list in the printout since it has been skipped by the `continue`:

```
for i in range(10):
    if i == 5:
        continue
    print (i)

0
1
2
3
4
6
7
8
9
```

Instead of using `range` it is possbile to specify directly the set of looping values:

```
for i in (4, 6, 10, 20): # here we loop directly on a list of numbers
   print (i)
```

```
4
6
10
20
```

Finally looping on a string actually means to loop on each single character:

```
phrase = 'how to loop over a string'
for c in phrase:
    print (c)

h
o
w

t
o

l
o
o
p

o
v
e
r

a

s
t
r
i
n
g
```

1.5.2 while

In a `for` loop we go through all the elements of a list of objects, the `while` statement instead repeats the same block of code untill a condition is met. The following block of code is run if `x` squared is less than 50, so we first set `x=1` and at each iteration we increment it by 1 untill the condition is `True` (8 squared is 64 which is greater than 50):

```
x = 1
while x ** 2 < 50:
    print (x)
    x += 1
```

```
1
2
3
4
5
6
7
```

It is possible to exit prematurely from a `while` loop using the `break` keyword. In this case the while-condition is simply `True` so the code would run forever unless we set an exit strategy.

```
x = 1
while True:
  if (x ** 2 > 50):
      break
  print (x)
  x += 1

1
2
3
4
5
6
7
```

Chapter 2

Data Containers

In this chapter the container types available in `python` are reviewed.

2.1 Lists

A list in `python` is a container that is a *mutable*, ordered sequence of elements. Each element or value that is inside of a list is called an *item*. Each item can be accessed using square brackets notation (very important, list indexing is zero-based so the first element is the 0th). A list is considered mutable since you can add, remove or update the items in it. Ordered instead means that items are kept in the same order they have been added. Lists can be created by enclosing in square brackets the comma-separated list of the items or using the `list()` operator.

```
mylist = [21, 32, 15]
print(mylist)
print (type(mylist))

[21, 32, 15]
```

```
print(mylist[0])

21
```

If you have a list of lists (i.e. a 2-dimensional list) you can use the square brackets multiple times to access the inner elements:

```
alist = [[1,2], [3,4], [5,6]]
print (alist[1][1]) # first [1] returns [3,4], second returns 4
```

The number of elements in a list is counted using the keyword `len()`:

```
print(len(mylist))

3
```

Looping on list items can be achieved in two ways: using directly the list or by index:

```
print ("Loop using the list itself:")
for i in mylist:
```

```
    print (i)

print ("Loop by index:")
for i in range(len(mylist)): # len() returns the number of items in a list
    print (mylist[i])

Loop using the list itself:
21
32
15

Loop by index:
21
32
15
```

With the `enumerate` function is actually possible to do both at the same time since it returns two values, the index of the item and its value, so in the example below, `i` will take the item index values while `item` the item value itself:

```
for i, item in enumerate(mylist):
    print (i, item)

0 21
1 74
2 85
3 15
4 188
```

Since a list is mutable we can dynamically change its items:

```
mylist[1] = 74 # we can change list items since it's *mutable*
print (mylist)

[21, 74, 15]
```

With `append` an item is added at the end, while with `insert` an item can be added in a specified position:

```
mylist.append(188) # append add an item at the end of the list
print (mylist)

[21, 74, 15, 188]
```

To append multiple values at once to a list a loop can be used but `python` offers a single line way of doing it: `[i*2 for i in range(10)]`. This syntax is called *list comprehension*.

```
mylist.insert(2, 85) # insert an item in the desired position
                     # (2 in this example)
print (mylist)

[21, 74, 85, 15, 188]
```

Accessing items outside the list range gives an error:

```
mylist[10] # error ! it doesn't exists, the list has only 3
           # elements, so the last is item 2

---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-36-ed1e5e6c3e46> in <module>
----> 1 mylist[10] # error ! it doesn't exists, the list has only 3
      2           # elements, so the last is item 2

IndexError: list index out of range
```

There are two more nice features of python indexing: negative indices are like positive ones except that they starts from the last element, and *slicing* whcih allows to specify a range of indices.

```
print ("negative index -1 returns the last element:", mylist[-1])
print ("slice [1:3] returns items between the 1st and 2nd:", mylist[0:3])
print ("slice [:2] returns items between the 1st and 2nd:", mylist[:2])
print ("slice [2:] returns items between the 2nd and the last:", mylist[2:])

negative index -1 returns the last element: 188
slice [1:3] returns items between the 1st and 2nd: [21, 74, 85]
slice [:2] returns items between the 1st and 2nd: [21, 74]
slice [2:] returns items between the 2nd and the last: [85, 15, 188]
```

It is worth mentioning that a list doesn't have to be populated with the same kind of objects (list indices are instead always integers).

```
mixedlist = [1, 2, "b", math.sqrt]
print (mixedlist)

[1, 2, 'b', <built-in function sqrt>]
```

```
print (mixedlist['k'])

---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-72-aea4c7f9789e> in <module>()
----> 1 print (mixedlist['k'])

TypeError: list indices must be integers or slices, not str
```

A complete list of the commands available for a list can be shown with the `dir` statement:

```
dir(list)

[...
```

```
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

Their meaning is pretty clear, so for example `sort` re-order the items according to a custom criteria or `index(item)` return the index of the specified item.

2.2 Dictionaries

A we have seen lists are ordered collections of element and as such we can say that map integers (the index of each item) to values (any kind of `python` object). *Dictionaries* generalize such a concept being containers which map *keys* (**almost** any kind of `python` object) to values (any kind of `python` object). In this case since the keys are not anymore necessarily integers there is no particular ordering of the items of a dictionary.

In our previous section we had:

$$0\ (\text{0th item}) \rightarrow 21$$
$$1\ (\text{1st item}) \rightarrow 74$$
$$2\ (\text{2nd item}) \rightarrow 85$$
$$\ldots$$

With a dictionary we can have something like this:

$$\text{"}apple\text{"}(\text{key}) \rightarrow 4$$
$$\text{"}banana\text{"}(\text{key}) \rightarrow 5$$

As we will see dictionaries are very flexible and will be very usefull to represent complex data structures.

Dictionaries can be created by enclosing in curly brackets the comma-separated list of key-value pairs (key and value are separated by a :), or using the `dict()` operator. In lists we could access items by index, here we do it by key still using the square brackets. Trying to access not existing keys results in error, but we can check if a key exists with the `in` operator. As before, if a dictionary contains other dictionaries or lists, the square brackets can be applied repeatedly to access the inner items.

```
adict = {"apple": 4, "banana": 5}
print (adict["apple"])

4
```

```
adict["pear"] # error !

---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

<ipython-input-41-9d051ebd10de> in <module>
----> 1 adict["pear"] # error ! this key doesn't exists

KeyError: 'pear'
```

```
"pear" in adict # indeed

False
```

The items can be dynamically created or updated with the assignement = operator, while again `len()` returns the number of items in a dictionary.

```
adict["banana"] = 2
adict["pear"] = 10
print (len(adict))
print (adict)

3
{'apple': 4, 'banana': 2, 'pear': 10}
```

Dictionaries can be made of more complicated types than simple string and integers:

```
adict[math.log] = math.exp
```

Also dictionaries can be created with the *comprehension* syntax: `i:v for i, v in enumerate(["a", "b", "c"])`.

Looping over dictionary items can be done by key, by value or by both: `.keys()` returns a list of keys, `.values()` returns a list of values and `.items()` a list of pairs key-value.

```
print ("All keys: ", adict.keys())
for key in adict.keys():
    print (key)

print ()
print ("All values: ", adict.values())
for value in adict.values():
    print (value)

print()
print ("All key-value pairs: ", adict.items())
for key, value in adict.items():
    print (key, value)

All keys:  dict_keys(['apple', 'banana', 'pear', <built-in function log>])
apple
```

```
banana
pear
<built-in function log>

All values:  dict_values([4, 2, 10, <built-in function exp>])
4
2
10
<built-in function exp>

All key-value pairs:  dict_items([('apple', 4), ('banana', 2), ('pear', 10),
(<built-in function log>, <built-in function exp>)])
apple 4
banana 2
pear 10
<built-in function log> <built-in function exp>
```

To merge two dictionaries the function `update()` can be used, while with `del` it is possible to remove a key-value pair.

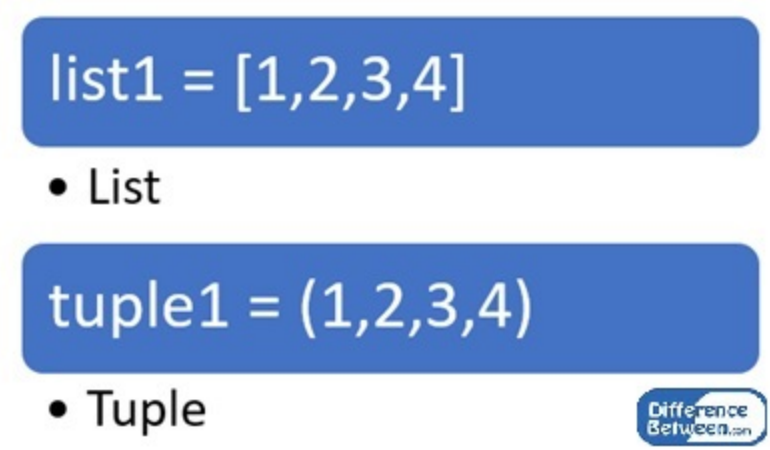
```
del adict[math.log]
seconddict = {"watermelon": 0, "strawberry": 1}
adict.update(seconddict)
print (adict)

{'apple': 4, 'banana': 2, 'pear': 10, 'watermelon': 0, 'strawberry': 1}
```

Again the complete list of dictionary functions can be shown with `dir`:

```
dir(dict)

[...
 'clear',
 'copy',
 'fromkeys',
 'get',
 'items',
 'keys',
 'pop',
 'popitem',
 'setdefault',
 'update',
 'values']
```

2.3 Tuples

Tuples create a bit of confusion for beginners because they are very similar to lists but they have some subtle conceptual differences. Nonetheless, tuples do appear when programming in `python`



At first glance list and tuples look very similar, but they are not...

so it's important to know about them.

Like lists, tuples are containers of any type of object. Unlike lists though they are *immutable* which means that once they have been created the content cannot be changed (i.e. no append, insert or delete of the elements). Furthermore since they are immutable they can be used as dictionary keys (lists cannot). To create a tuple the comma-separated list of items has to be enclosed in brackets, or the `tuple()` operator can be used. Accessing tuple items is done in exactly the same way as lists.

```
atuple = (1, 2, 3)
print ("Length: {}".format(len(atuple)))
print ("First element: {}".format(atuple[0]))
print ("Last element: {}".format(atuple[-1]))

Length: 3
First element: 1
Last element: 3
```

In the next snippet of code it is shown the so called unpacking which is another way to assign tuple values to variables.

```
x, y, z = (10, 5, 12)
print ("coord: x={} y={} z={}".format(x, y, z))

coord: x=10 y=5 z=12
```

If and ntuple has just one element don't forget the comma at the end otherwise it will be treated as a single number.

```
tuple2 = (1,)
print(type(tuple2))
tuple2 = (1)
print(type(tuple2))

<class 'tuple'>
<class 'int'>
```

Since a tuple is immutable to add new elements it is necessary to create a new object:

```
tuple1 = (1, 2, 3)
tuple2 = tuple1 + (4, 5)
print(tuple2)

(1,2,3,4,5)
```

Finally, as already said tuples can be used as dictionary keys:

```
d = {
   ('Finance', 1): 'Room 8',
    ('Finance', 2): 'Room 3',
    ('Math', 1): 'Room 6',
    ('Programming', 1): 'IT room'
    }
```

Below the full list of tuple functions:

```
dir(dict)

[...
 'count',
 'index']
```

Chapter 3

Date and Time

In this chapter we will take a little break and concentrate on a topic that it is not usually covered in this type of courses. However given its importance for financial computation the next paragraphs will be devoted to a close look up on the `datetime` module, whose utilities help in manipulating dates.

3.1 Dates

As said dates are not usually included in a standard `python` tutorial, however since they are pretty essential for finance we are going to cover this topic in some detail. In `python` the standard date class lives in the `datetime` module. We are also going to import `relativedelta` from the `dateutil` module, which allows to add/subtract days/months/years to dates, in other words to make operations on them.

In this first example the today's date is defined and with `relativedelta` two more date are created adding two months and three days to today.

```
from datetime import date, datetime
from dateutil.relativedelta import relativedelta

date1 = date.today()
print (date1)
date2 = date.today() + relativedelta(months=2)
print (date2)
date3 = date.today() - relativedelta(days=3)
print (date3)

2020-08-03
2020-10-03
2020-07-31
```

Here instead another way of computing a new date is shown: in particular a one day delta is stored in a variable and today's date is moved by three days multiplying the defined delta by three.

```
one_day = relativedelta(days=1)
date.today() - 3 * one_day

datetime.date(2020, 7, 31)
```

Next given two dates their difference is computed (and expressed in days).

```
date1 = date(2019, 7, 2)
date2 = date(2019, 8, 16)
(date2 - date1).days

45
```

Dates can be converted to and from strings and a large variety of formats can be specified in this conversions. The format is determined by a string in which each character starting with % represent an element of the date, e.g. %Y year, %d day, %s seconds, etc...

Below dates to string convertion:

```
date1 = date(2019, 7, 2)
date1.strftime("%Y-%b-%d (%a)") # dates can formatted in many ways
                                # check the docs for more details

'2019-Jul-02 (Tue)'
```

And here, strings are converted to datetime object:

```
# a string can be convered to dates too
datetime.strptime('25 Aug 2019', "%d %b %Y").date()

datetime.date(2019, 8, 25)
```

Finally a last example showing how to get the week-day from a date:

```
date1.weekday() # 0 = monday, ..., 6 = sunday

1
```

Chapter 4

Python's Object Oriented Programming

In this chapter the main characteristics that makes `python` an *object oriented programming* language will be reviewed. Before going to OOP the concept of function and variable scope will be outlined.

4.1 Functions

A function is a block of organized, reusable code that is used to perform a single action. Functions provide better modularity for your application and high degree of code reusing. To define a function the keyword `def` is used, followed by the name of the function and by the required parameters in parenthesis.

```
# sum up all the integers between 1 and n
def my_function(n): # this function take one input only (n)
    x = 0
    for i in range(1, n+1):
        x += i
    return x # the function returns a number

my_function(5) # 5 + 4 + 3 + 2 + 1
```

Functions can return any kind of objects (numbers, strings, lists, complex objects...) but it is not mandatory to have a return value, so you can have functions **without** a `return` statement (e.g. a function that simply take a string as input and print it to screen with a particular format). In addition the syntax of the `return` is different from other languages like `Visual Basic`, the returned object doesn't have to have the same name as the function. Indeed above the variable `x` is returned and not the variable $\texttt{my}_\texttt{f}\texttt{unction}$.

```
def printing(mystring):
    print((myString).upper())
```

Functions can call other functions (once a function has been defined it can be accessed from everyone withing the same file or notebook): here $\texttt{my}_\texttt{f}\texttt{unction}_2$ calls $\texttt{my}_\texttt{f}\texttt{unction}$

```
def my_function_2(n, x):
   return "The result is : {}".format(str(my_function(n)*x))
```

```
    # returns x * result of my_function(n)
    # so function of function

my_function_2(5, 10)
```

Functions can also call themselves too (i.e *recursion*). In the next example we will see a function that computes the factorial exploiting the following relationship:

$$\begin{cases} n! = n \times (n-1)! & (\forall n > 1) \\ n! = 1 & (\forall n \leq 1) \end{cases}$$

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)

factorial(10)
```

In this example the function `factorial` is initially called with the input corresponding to the factorial we want to compute, it then call itself each time with `n` $-$ `1`, multypling together all the results.

The previous example is quite simple but recursion can be tricky sometimes so apply it with caution.

Functions input parameters also accept default values, which means that a function that works with some input values can be called with less parameters provided their default values have been specified.

In the following example the function `powers` takes three inputs: a list of numbers, an exponent (`n`) and a constant (`c`). The code loops through the provided list of numbers and process them according to the formula $item^n + c$, it puts the results in a new list which will be finally returned.

```
def powers(l, n=2, c=0):
    return [item**n+c for item in l]

print (powers([5, 11, 6], 3, 4))
print (powers([5, 11, 6]))

[129, 1335, 220]
[25, 121, 36]
```

As you can see in the example the function is called twice with two different set of parameters: in the first case we pass to it the list of numbers, the exponent and the costant, in the second only just the same list of numbers. In the latter case, being defined the default values for `n` and `c`, the function works as well, fewer inputs are provided and the missing ones will be replaced by their defaults.

When calling a function paramenters can be passed also by name for clarity, in this case of course the order doesn't matter. Compare the two results below:

```
def func(a, b, c):
    return a + b * c
```

```
print (func(c=4, b=2, a=1))
print (func(4, 2, 1))

9
6
```

In the first case the function is called by name, in the second case the parameter are implicitly assinged according to thier position.

Another nice feature of `pyhton` functions is that we can associate an help message to them so that we can easily check what a function is for by simply asking `help(functioName)`:

```
def powers(l, n=2, c=0):
    """
a shifted power function example
    """
    return [item**n+c for item in l]

help(powers)

Help on function powers in module __main__:

powers(l, n=2, c=0)
    a shifted power function example
```

Remember it is always very important to document your code !

4.2 Variable scope

Not all variables are accessible from all parts of our program, and not all variables exist for the entire lifetime of the program. The part of a program where a variable is accessible is called its *scope*.

A variable which is defined in the main body, sometimes referred to as global namesapce i.e. the code block which is not indented at all, of a file is called a *global variable*. It will be visible throughout the file, and also inside any file which imports that file. Global variables can have unintended consequences because of their wide-ranging effects, that is why we should almost never use them and they are usually represented by an uppercase name. Only objects which are intended to be used globally, like functions and classes (which will be introduced in the next section), should be put in the global namespace.

Global variables can be accessed directly inside but before doing that they have to be specified in a special statement starting with the keyword `global`. Essentially `global` tells `python` that in the following function we want to use the listed global variable.

Imagine a global variable `AGLOBALPARAM` has been defined at the beginning of a program, in the example below it is shown:

- a function that read the value of `AGLOBALPARAM` without modifying it;
- a function that read and modify `AGLOBALPARAM`;

- and a function that throws an exception (i.e. an error in technical language) because it has been badly coded.

```
AGLOBALPARAM = 10

# Here you just use AGLOBALPARAM value, but do not modify it
# param is just a local copy of AGLOBALPARAM
def multiplyParam(param):
    param = param * 10
    return (param)

# Here you actually use AGLOBALPARAM
# you modify it directly with the global command
def divideParam():
    global AGLOBALPARAM
    AGLOBALPARAM = AGLOBALPARAM / 10
    return (AGLOBALPARAM)

# Here you try to use AGLOBALPARAM but gives you an error
# AGLOBALPARAM is not defined in the function body
# and the global command has not been used neither
def sumParam():
    AGLOBALPARAM = AGLOBALPARAM + 10
    return (AGLOBALPARAM + x)

print ("AGLOBALPARAM is {} to start.".format(AGLOBALPARAM))
print ("Let's multiply it by 10.")
multiplyParam(AGLOBALPARAM)
print ("AGLOBALPARAM is still {}".format(AGLOBALPARAM))
print ("Let's divide it by 10")
divideParam()
print ("Now AGLOBALPARAM is {}".format(AGLOBALPARAM))
print ("Let's sum it to 10")
sumParam()
```

A variable which is defined in a block of code is said to be local to that block. Examples of local scopes are: functions, `for` or `while` loops, `if` blocks, ... In the case of a fucntion it means that a local variable will be accessible from the point at which it is defined until the end of the function itself (e.g. function parameters are examples of local variables).

```
# functions are not evaluated untill their are not called
def test_scope(max_val):
    for i in range(max_val):
        print (i)
    print ("max_val in 'test_scope' function is {}".format(max_val))

# the Python interpreter starts evaluating the code from here
max_val = 10
```

```
test_scope(5)
print ("max_val in global scope is {}".format(max_val))
print (i)
```

In the previous example we have defined two `max_val` variables, one which is global and it has been initially set to 10, another one which is local to the `test_scope` function. Try as much as possible to use different names for each variable you are going to use in a program to avoid possible confusion and mistakes which may lead to unexpected behaviour of your code.

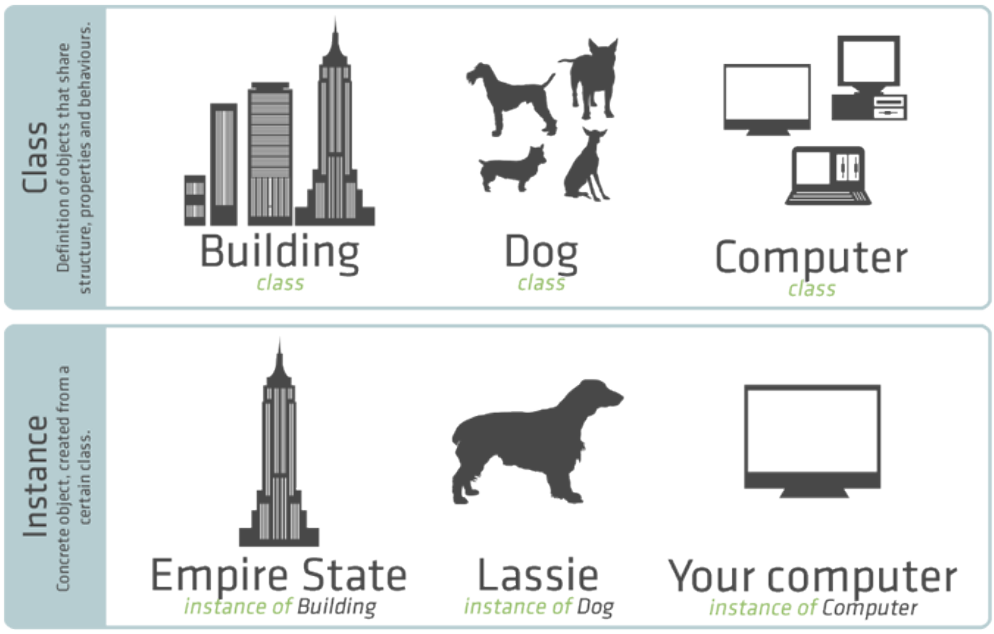
As a last example compare the following `for` loops; the first one correctly written loops with tha variables `i` and `j`, in the second one `j` has been replaced by `i`, note how this is perfectly legal but the result changes dramatically:

```
a = [["a", "b", "c"],["d", "e", "f"],["g", "h", "i"]]
for i in range(3):
    for j in range(3):
        print (a[i][j])

a
b
c
d
e
f
g
h
i
```

```
a = [["a", "b", "c"],["d", "e", "f"],["g", "h", "i"]]
for i in range(3):
    for i in range(3):
        print (a[i][i])

a
e
i
a
e
i
a
e
i
```

4.3 Classes

Classes are a key ingredient of *Object Oriented Programming* (OOP) and their concept is implemented in many languages like `python`, `Java` and `C++`. OOP is a programming model in which programs are organized around data, or objects, rather than functions and logic. **An object can be**



Graphical representation of a class instance.

thought of a dataset with unique attributes and behaviour (examples can range from physical entities, such as a human being that is described by properties like name and birthday, down to abstract concepts as a discount curve). This opposes the historical approach to programming where emphasis was placed on how the logic was written rather than how to define the data within the logic. In this framework classes are a mean for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).

Let's summarize here some terminology:

- a class is a collection of related functions, and these are called the *methods* of the class;
- methods act on *instances* of the class, which are classes initialized with some data;
- each data item has a name, and those names are called the *attributes* of the class.

In other words classes are collections of functions that operate on a dataset, and instances of that class represent individual datasets (or in other words a specialization of that class).

Examples of class are: a class representing a generic building (with number of entrances, number of floors, a flag to know if there is a garden...), a generic dog (with age, fur color...) or a generic computer (with manifacturer, RAM size, CPU type,...).

Examples of corresponding instances are: the Empire State Building (a specific building), Lassie (a very particular dog), or your computer.

To see how they can be defined let's try to code a class representing a person:

```
from datetime import date
# this is the class definition
# usually classes use camel naming convention
class Person:
```

```
    # the special method __init__ allows to instanciate a class
    # with an initial dataset (in this example a name and a birthday)
    def __init__(self, name, date_of_birth):
        # attribute of the class Person
        # name and self.name are different variable !!!
        # name will be destroyed once __init__ is processed
        # self.name lives with every particualar instance of Person
        self.name = name
        # attribute of the class Person
        self.date_of_birth = date_of_birth

    # this normal method computes the current age of the
    # "instanciated" person
    def age(self):
        today = date.today()
        age = today.year - self.date_of_birth.year
        if today.month < self.date_of_birth.month or \
            today.day < self.date_of_birth.day:
            age -= 1
        return age
```

First of all the necessary modules are imported, in this case the `datetime` module is used to managed the person age. Then the `class` keyword followed by the class name is used to start the actual class definition. In a sepate block of code all the class methods are defined like normal functions, you can see `__init__` and `age` here. These are examples of two kind of methods:

- normal methods which use or modify the instance attributes;
- special methods, which define the class behaviour: you can spot these because they start and end with two underscores (__).

`__init__` is the simplest example of special methods, it is called every time a class is instanciated (e.g. when you write me = Person(...)) and initializes the attributes of the class, in our example assign values to the `name` and `date_of_birth` attributes.

Another peculiarity of class methods with respect to standard functions is that they always take as first argument `self`. The `self` keyword is very important since allows a method to use the class attributes. Basically if you need to use the `name` attribute you have to type `self.name`.

There are lots of other things you can do with classes, but this is enough for now. So let's try to play a bit with our example:

```
# here we instanciate (create an instance of) the class
# in other words we "specialize" a generic Person with some data

me = Person("Matteo", date(1974, 10, 20))
  print (type(me))
```

Once we have create an instance of a class its methods and attributes can be accessed again with the dot notazion: `instance_name.method_name`.

```
# to access class attributes you have to use .
me.name
```

```
me.date_of_birth

# to call a class method you have to use .
# passing the parameters if needed
me.age()
```

Let's try to add more functionality to our class, adding a method that simply prints the age of the person in nice format. So complete the `Person` class with the following code:

```
    # methods in a class are just functions which can work
    # with the class attributes
    # Remember I told you functions can have no return ?
    def print_age(self):
        print ("{} is {} years old right now"\
               .format(self.name, self.age()))
```

Then try to test the new method by instanciating a new "person" and print her age.

```
her = Person("Francesca", date(1986, 1, 27))
print (her.print_age())
```

```
34
```

Chapter 5

Data Manipulation and Its Representation

In this chapter a closer look two a couple more of modules is given. These modules will result to be very useful in managing financial data and to report result of our analysis.

5.1 Getting Data

The first step of any analysis is usually the one that involves selection and manipulation of data we want to process. Data sources can be various (eg. website, figures, twitter messages, CSV or Excel files...) and partially reflect its nature which can range from *unstructured* data (whitout any inherent structure, e.g. social media data) to completely *structured* data (where the data model is defined and usually there is no error associated, e.g. stock trading data).

Our primary goal, before start processing data, is to collect and store the information in a suitable data structure. `Python` provides a very useful module, called `pandas`, which allows to collect and save data in *dataframe* objects that can be later on manipulated for analysis purposes.

Looking at `pandas` manual dataframe are defined as multi-dimensional, size-mutable, potentially heterogenous, tabular data structure with labeled axes (rows and columns), in much simpler words it is a table whose structure can be modified. It presents data in a way that is suitable for data analysis, contains multiple methods for convenient data filtering and in addition has a lot of utilities to load and save data pretty easly.

Dataframes can be created by:

- importing data from file;
- creating by hand data and then filling the dataframe.

```
import pandas as pd

# reading from file
df1 = pd.read_excel('sample.xlsx') # Excel file
df2 = pd.read_csv('sample.csv') # Comma Separeted file

df1.head(11) # show just few rows at the beginning
```

```
         Date       Price      Volume
0  2000-07-30  100.000000  191.811275
1  2000-07-31  129.216267  190.897541
2  2000-08-01  147.605516  197.476379
3  2000-08-02  107.282251  199.660061
4  2000-08-03  106.036826  200.840459
5  2000-08-04  118.872757  197.130212
6  2000-08-05  101.904544  204.552521
7  2000-08-06  106.392901  198.160030
8  2000-08-06  106.392901  191.125969
9  2000-08-06  106.392901  196.719061
10 2000-08-06  106.392901  196.759837
```

```
# creating some data in a dictionary
d = {"Nome":["Elisa", "Roberto", "Ciccio", "Topolino", "Gigi"],
     "Età":[1, 27, 25, 24, 31],
     "Punteggio":[100, 120, 95, 1300, 101]}

# filling the dataframe
df = pd.DataFrame(d)
df.head()
```

```
       Nome  Età  Punteggio
0     Elisa    1        100
1   Roberto   27        120
2    Ciccio   25         95
3  Topolino   24       1300
4      Gigi   31        101
```

Of course with pandas it is possible to perform a large number of operations on a dataframe. For example it is possible to add a column as a result of an operation on other columns. Looking back at the df1 dataframe it is possible to add a column with the daily variation of the price.

```
import numpy as np

# first let's add an empty column
df1['Variation'] = np.nan # nan stands for not a number

# loop on the Price column, compute the variation and fill the column
# len returns the number of rows of a dataframe
for i in range(1, len(df1)):
    # select the ith row and fill "Variation"
    # loc takes as inputs row and colum-name
    df1.loc[i, "Variation"] = (df1.loc[i, "Price"] - df1.loc[i-1, "Price"]) /
                        df1.loc[i-1, "Price"]

df1.head()
```

```
        Date       Price      Volume  Variation
```

```
0 2000-07-30  100.000000  191.811275        NaN
1 2000-07-31  129.216267  190.897541   0.292163
2 2000-08-01  147.605516  197.476379   0.142314
3 2000-08-02  107.282251  199.660061  -0.273183
4 2000-08-03  106.036826  200.840459  -0.011609
```

Of course the first "variation" value is NaN since there is no previous price to compare with.

5.1.1 Manage Data

Once we have created our dataframe we may want to preliminarly process data to perform very common operations like:

- remove unwanted observations or outliers;
- handle missing data;
- filter, sort and clean data.

5.1.2 Unwanted observations and outliers

Duplicates

It may happen that our data has duplicates (e.g. those can arise when combining two datasets), or the dataset contains irrelvant fields for the specific study we are carrying on. To find and remove duplicates `pandas` has convenient methods:

```
# find duplicates based on all columns
# and show just the first 15 results
#print (df1.duplicated()[:15])

# find duplicates based on'Price'
# and show just the first 15 results
print (df1.duplicated(subset=['Price'])[:15] )

0     False
1     False
2     False
3     False
4     False
5     False
6     False
7     False
8      True
9      True
10     True
11    False
12    False
13    False
14    False
dtype: bool
```

```
print ("Initial number of rows: {}".format(len(df1)))

# remove duplicates
# where the second argument can be `first`, `last`
# or `False` (consider all of the same values as duplicates).
df1 = df1.drop_duplicates(subset='Price', keep='first')

print ("Number of columns after drop: {}".format(len(df1)))

Initial number of rows: 734
Number of columns after drop: 729
```

If we would like to drop irrilevant columns for our analysis it is enough to:

```
df2 = df2.drop(columns=['Volume'])
df2.head()

         Date       Price
0  2000-07-30  100.000000
1  2000-07-31  129.216267
2  2000-08-01  147.605516
3  2000-08-02  107.282251
4  2000-08-03  106.036826
```

If instead we just want to remove few rows we can select them by index:

```
# we remove row 0th and 2nd
# axis=0 means use the index column
df2 = df2.drop([0, 2], axis=0)
df2.head()

         Date       Price
1  2000-07-31  129.216267
3  2000-08-02  107.282251
4  2000-08-03  106.036826
5  2000-08-04  118.872757
6  2000-08-05  101.904544
```

Changing the column that act as index we can select the rows also by other attributes:

```
# tell pandas to use Date as index column
df2 = df2.set_index('Date')

# select row to remove by date at this point
df2 = df2.drop(["2000-07-31"], axis=0)

df2.head()

Date        Price
2000-08-02  107.282251
2000-08-03  106.036826
```

```
2000-08-04  118.872757
2000-08-05  101.904544
2000-08-06  106.392901
```

Outliers

An outlier is an observation that lies outside the overall pattern of a distribution. Common causes can be human, measurement or experimental errors. Outliers must be handled carefully and we should remove them cautiously, *outliers are innocent until proven guilty*. We may have removed the most interesting part of our dataset !

The core statistics about a particular column can be studied by the `describe()` method which returns the following information:

- for numeric columns: the value count, mean, standard deviation, minimum, maximum and 25th, 50th and 75h quantiles for the data in a column;
- for string columns: the number of unique entries, the most frequent occurring value (*top*), and the number of times the top value occurs (*freq*).

```
df1.describe()

              Price      Volume   Variation
count    728.000000  729.000000  724.000000
mean     120.898678  200.355900    0.146330
std      490.493411    4.970745    3.637952
min        0.878873  186.430551   -0.995284
25%       14.809934  196.998603   -0.119423
50%       61.325699  200.221125   -0.005549
75%      164.021813  203.580691    0.121290
max    13000.000000  215.140868   97.756432
```

Looking at mean and std and comparing it with min and max values we could find a range outside of which we may have outliers. For example 13000.0 is several standard deviation away the mean which may indicate that it is not a good value.

Another way to spot outliers is to plot column distributions and again `pandas` comes to help us:

```
df1.hist("Variation", bins=np.arange(0, 100, 1))

---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

        <ipython-input-1-97dbdc6fcfec> in <module>
    ----> 1 df1.hist("Variation", bins=np.arange(0, 100, 1))


        NameError: name 'df1' is not defined
```

From the histograms it is clear how the value of 97.76, is far from general population. This doesn't mean they are necessarily wrong but it should make ring a bell in our head...

To remove outliers from data we can either remove the entire rows or replace the suspicious values by a default value (e.g. 0, 1, a threshold value...).

Note: missing data may be informative itself ! When filling the gap with *artificial data* (e.g. mean, median, std...) having similar properties than real observation, the added value won't be scientifically valid, no matter how sophisticated your filling method is.

```
import numpy as np

df2.replace(1300, 500)      # replace 1300 with 500
df2 = df2.replace(1300, np.nan)   # replace 1300 with NaN

df2 = df2.mask(df1 >= 600, 500)   # replace every element >=600 with 5
```

5.1.3 Handle Missing Data

Usually when importing data with `pandas` we may have some NaN values (short for *not a number* which represent the `null` value). NaN is the value that is given to missing fields in a row. Like for the outliers we can use the `replace` or `mask` methods to remove the NaNs. In case the whole row as NaN it may be wise to drop it entirely.

Additionally we can use `dropna()` which remove all the NaN at once.

```
df1 = df1.dropna()

print ("Number of rows after dropping NaN: {}".format(len(df1)))

Number of rows after dropping NaN: 724
```

5.1.4 Filter, Sort and Clean Data

Filtering

When we work with huge datasets we may reach computational limits (e.g. insufficient memory, CPU performance, too slow processing time...) and in those cases it can be helpful to filter data by attributes for example by splitting by time or some other property.

Assuming to have the following table and putting back the volume column

```
# df.iloc[row, col]
# NOTE: iloc takes row and column index (two numbers)
# loc instead takes row index and column name
print (df1.iloc[1, 2]) # returns 62 the volume associated with the row 1

print()
#df.iloc[row1:row2, col1:col2]
# this is called slicing, remember ?
print (df1.iloc[0:2, 2:3]) # returns rows 0 and 1 of column 2

197.476378531652

       Volume
1  190.897541
```

```
2  197.476379
```

```
subset = df1.iloc[:, 1]   # select column 1

subset = df1.iloc[2, :]   # select row 2

subset = df1.iloc[0:2, :] # select 2 rows

subset = df1.iloc[:2, :]  # this is equivalent to before
```

A more advanced way of filtering is the following (it apply a selection on the values). The notation is a bit awkward but very useful:

```
import datetime

# colon means all the rows
subset = df1[df1.iloc[:, 0] < datetime.datetime(2000, 8, 15)]
print (subset)

          Date       Price      Volume  Variation
1   2000-07-31  129.216267  190.897541   0.292163
2   2000-08-01  147.605516  197.476379   0.142314
3   2000-08-02  107.282251  199.660061  -0.273183
4   2000-08-03  106.036826  200.840459  -0.011609
5   2000-08-04  118.872757  197.130212   0.121052
6   2000-08-05  101.904544  204.552521  -0.142743
7   2000-08-06  106.392901  198.160030   0.044045
11  2000-08-07  107.646053  198.861429   0.011779
12  2000-08-08  106.666468  197.213497  -0.009100
13  2000-08-09  101.981029  204.425797  -0.043926
14  2000-08-10  110.100330  196.122844   0.079616
15  2000-08-11  138.656481  200.703360   0.259365
16  2000-08-12  113.180782  205.676449  -0.183732
17  2000-08-13  137.639947  203.468517   0.216107
18  2000-08-14  142.646169  198.528626   0.036372
```

Sorting

To sort our data we can use `sort_values()` method (it can be specified ascending, descending).

```
# sort by price then by date in descending order
df2.sort_values(by=['Price', "Date"], ascending=False)[:10]

      Date         Price
2000-08-20  13000.000000
2000-10-20    593.477666
2001-01-05    571.444679
2000-12-31    532.558487
2000-10-14    516.044122
2001-01-02    503.583189
```

```
2001-01-01    502.849987
2000-12-30    487.353466
2001-01-04    478.027182
2001-01-10    473.061993
```

Cleaning or Regularizing As we will see when dealing with machine learning, often we need to regularize our data to improve the stability of a training. One typical situation is when we want to *normalize* data, which means rescale the values into a range of [0, 1].

$x = [1, 43, 65, 23, 4, 57, 87, 45, 45, 23]$

$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$

$x_{new} = [0, 0.48, 0.74, 0.25, 0.03, 0.65, 1, 0.51, 0.51, 0.25]$

To apply such a transformation with `pandas` is very easy since applying the formula to a dataframe implies it is done to each row:

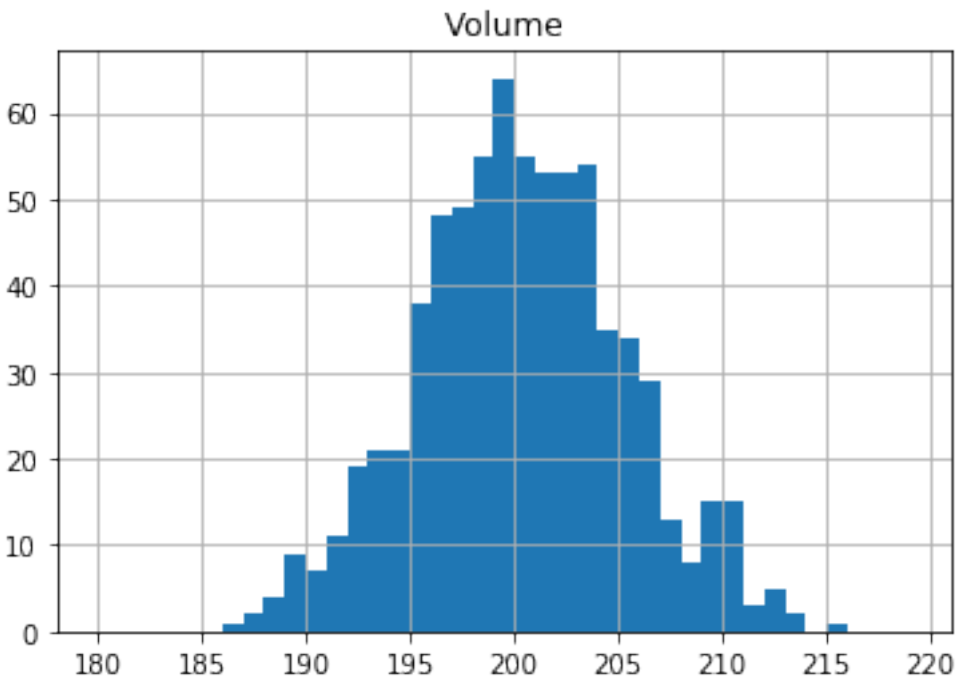
```
df1['Price'] = (df1['Price'] - df1['Price'].min()) \
    / (df1['Price'].max() - df1['Price'].min())
df1.head()

        Date     Price      Volume  Variation
1 2000-07-31  0.009873  190.897541   0.292163
2 2000-08-01  0.011287  197.476379   0.142314
3 2000-08-02  0.008185  199.660061  -0.273183
4 2000-08-03  0.008090  200.840459  -0.011609
5 2000-08-04  0.009077  197.130212   0.121052
```

Another quite common transfrmation is called *standardization*, essentially we rescale data to have 0 mean and standard deviation of 1:

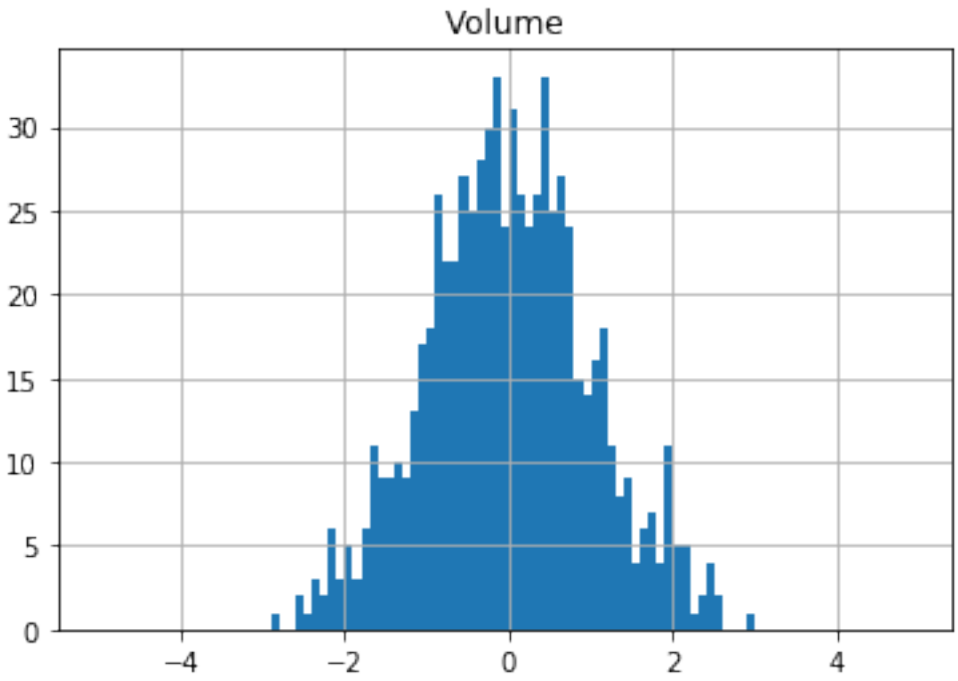
$x_{new} = \frac{x - \mu}{\sigma}$

Again it is straightforward to do it in `pandas`:

```
df1.hist('Volume', bins=np.arange(180, 220, 1))
print (df1['Volume'].mean())
print (df1['Volume'].std())

200.36750575214748
4.968224698257929
```




```
df1['Volume'] = (df1['Volume'] - df1['Volume'].mean()) / df1['Volume'].std()

df1.hist('Volume', bins=np.arange(-5, 5, 0.1))
print (df1['Volume'].mean())
print (df1['Volume'].std())

-6.148550054609154e-15
1.0
```



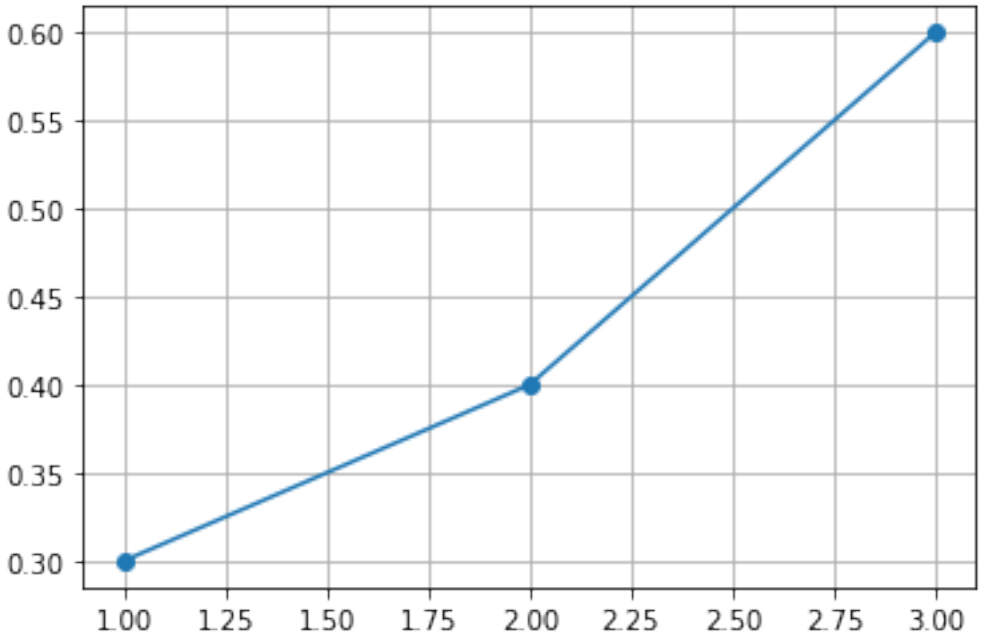
5.2 Plotting in `python`

As we have just seen `pandas` allows to quickly draw histograms of dataframe columns, but during an analysis we may want to plot distributions from `list` or objects not stored in a dataframe. Furthermore the simple and very useful provided interface doesn't grant full access to all histogram features that we need to produce nice and informative plots.

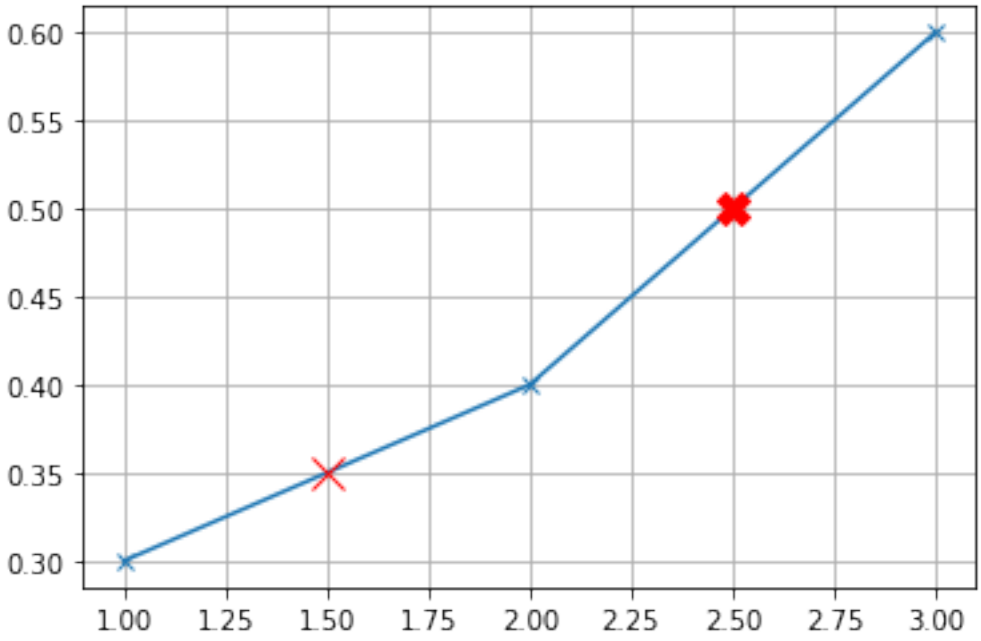
In order to do so we can use the `matplotlib` module which is specifically dedicated to plotting (pandas interface is based on the same module indeed). Let's look briefly to its capability by examples.

5.2.1 Plot a graph given x and y values (scatter-plot)

```
from matplotlib import pyplot as plt

x = [1, 2, 3]
y = [0.3, 0.4, 0.6]

plt.plot(x, y, marker='o') # we are using circle markers
plt.grid(True)             # this line activate grid drawing
plt.show()
```

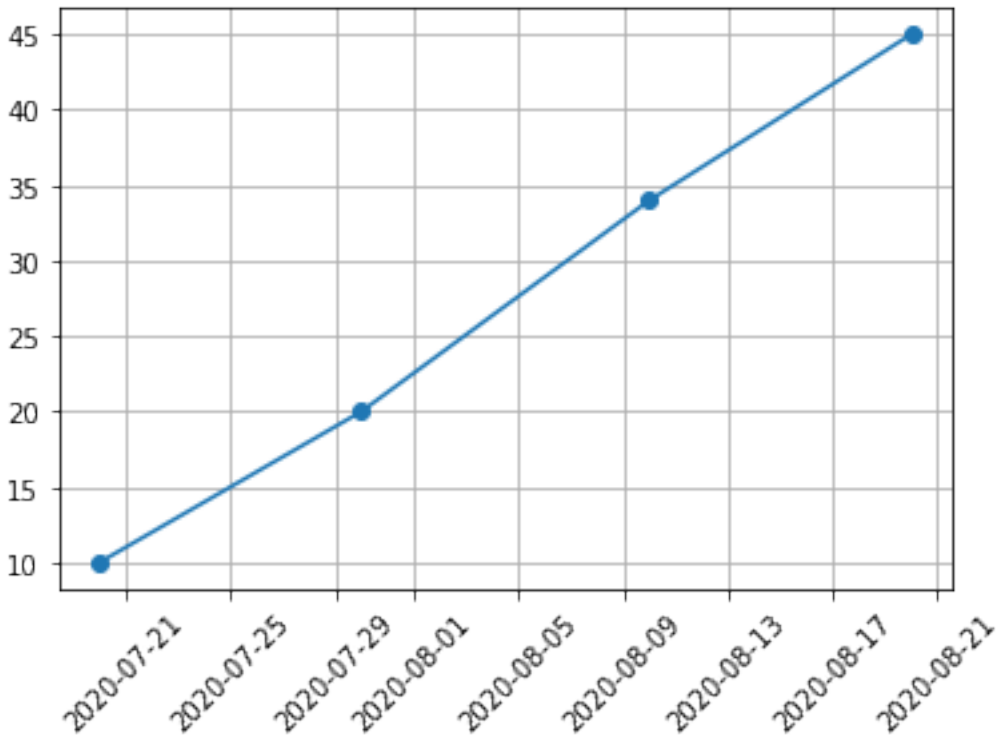



```
# if we want to plot specific points too

x = [1, 2, 3]
y = [0.3, 0.4, 0.6]

plt.plot(x, y, marker='x')
plt.plot(2.5, 0.5, marker='X', ms=12, color='red')
plt.plot(1.5, 0.35, marker='x', ms=12, color='red')
plt.grid(True)
plt.show()
```



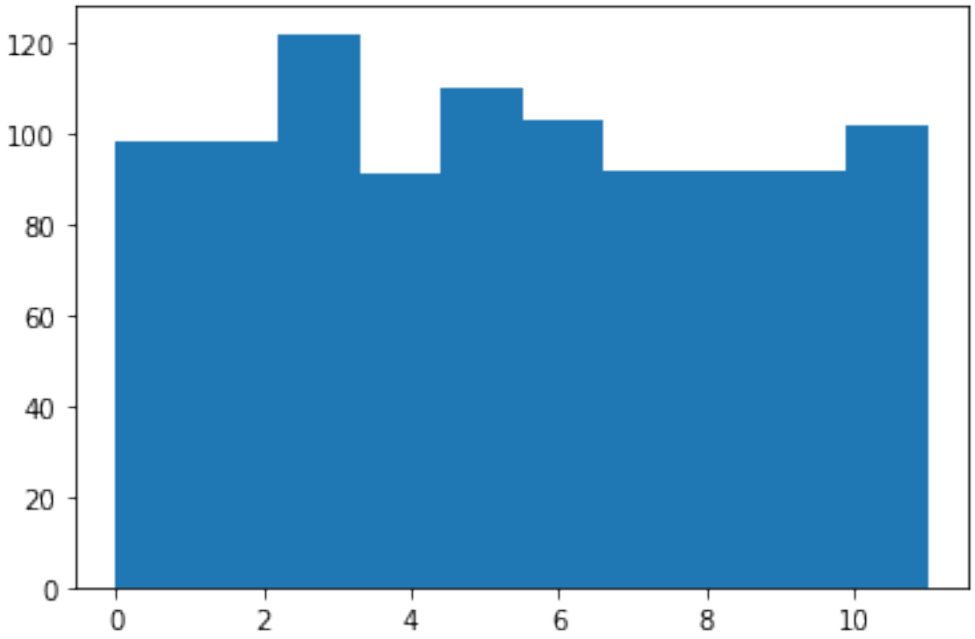
What if x values are dates ?

```
import datetime
from matplotlib import pyplot as plt
import matplotlib.dates as mdates

x = [datetime.date(2020, 7, 20),
     datetime.date(2020, 7, 30),
     datetime.date(2020, 8, 10),
     datetime.date(2020, 8, 20)]
y = [10, 20, 34, 45]
plt.plot(x, y, marker='o')
# this line tells matplotlib we have dates on x axis
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
# this one instead rotate labels to avoid superimposition
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```



5.2.2 Plotting an Histogram

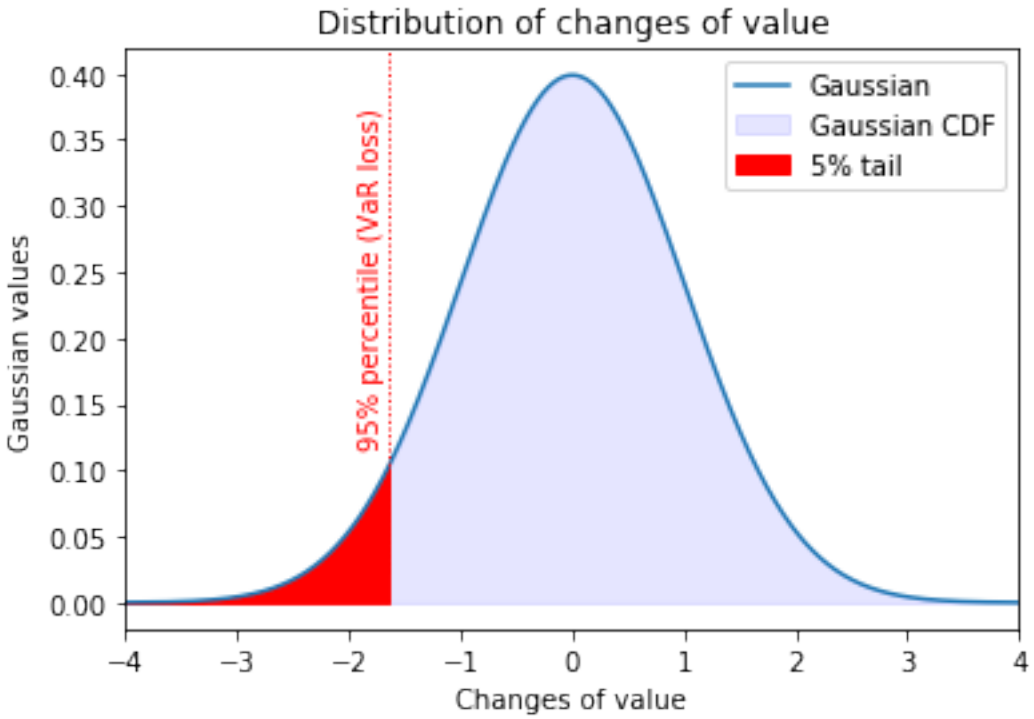
```
import random
numbers = []
for _ in range(1000):
  numbers.append(random.randint(1, 10))

from matplotlib import pyplot as plt

# Here we define the binning
# 6 is the number of bins, going from 0 to 10
plt.hist(numbers, 10, range=[0, 11])
plt.show()
```



Plotting a Function In this case let's try to make the plot prettier adding labels, legend... All the commands apply also to the previous examples.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# define the functions to plot
# a gaussian with mean=0  and sigma=1
# in scipy module this is called norm
mu=0
sigma = 1
x = np.arange(-10, -1.645, 0.001)
x_all = np.arange(-4, 4, 0.001)
y = norm.pdf(x, 0, 1)
y_all = norm.pdf(x_all, 0, 1)

# draw the gaussian
plt.plot(x_all, y_all, label='Gaussian')

# fill with different alpha using x_all and y_all as limits
# alpha set the transparency level: 0 trasparent, 1 solid
plt.fill_between(x_all, y_all, 0, alpha=0.1, color='blue', label="Gaussian CDF")

# fill with color red using x and y as limits
# label associate text to the object for the legend
```

```
plt.fill_between(x, y, 0, alpha=1, color='red', label="5% tail")

# set x axis limits
plt.xlim([-4, 4])

# add a label for X axis
plt.xlabel("Changes of value")

# add a label to y axis
plt.ylabel("Gaussian values")

# add histogram title
plt.title("Distribution of changes of value")

# draw a vertical line at x=-1.645
# y limits are in percent w.r.t. to y axis length
plt.axvline(x=-1.645, ymin=0.1, ymax=1, linestyle=':', linewidth=1, color = 'red')

# write some text to explain the line
plt.text(-1.9, .12, '95% percentile (VaR loss)',fontsize=10, rotation=90, color='red')

plt.legend()
plt.show()
```



If you are particularly satisfied by your work you can save the graph to a file:

```
plt.savefig('normal_curve.png')

<Figure size 432x288 with 0 Axes>
```

Chapter 6

Discount Factors and Forward Rate

In this chapter we will start to see the first applications of `python` to financial calculations. In particular we will consider discount curves and forward rates, implementing the first utilities that will fill our financial module. In addition we will briefly review a widely used mathematical tool: *interpolation*.

6.1 Linear interpolation

Interpolation is a method of constructing new points within the range of a discrete set of known data points.

Consider to have few data points, obtained by sampling or experimenting. These points represent the values of a function $f(x)$, where x is an independent variable (e.g. in recording a trip: distances at certain times, $d = f(t)$). It is often necessary to estimate the value of that function for intermediate values of the independent variable (e.g. in our previous example what is the distance d at a new time t for which there isn't a sample ?).

Example 1

Assume you are going on holidays by car and that luckily there isn't much traffic so that you can drive at constant speed (which gives a linear relation between travelled space and time i.e. $s = v \cdot t$, which means that if you plot the distances s as a function of the time t you get a line with slope v).

Given two samples of the car travelled distance s_1 and s_2 taken at two different times t_1 and t_2 you can linearly interpolate to find your position at different times using the following relations:

$$w = \frac{t - t_1}{t_2 - t_1}$$

where t is a generic time at which we want to know the distance s)

$$s = (1 - w) \cdot s_1 + w \cdot s_2$$

where s is the desired travelled distance.

Derivation: the equation of a line for two points (t_1, s_1) and (t_2, s_2) can be written as:

$$\frac{t - t_1}{t_2 - t_1} = \frac{s - s_1}{s_2 - s_1}$$

Setting $w = \frac{t-t_1}{t_2-t_1}$ and solving for s we find the desired solution:

$$w = \frac{t - t_1}{t_2 - t_1} \Rightarrow (s_2 - s_1) \cdot w = s - s_1 \Rightarrow ...$$

Back to our example, if $s_1 = 25.75$ km $(@t_1 = 15$ min$)$ and $s_2 = 171.7$ km $(@t_2 = 100$ min$)$ let's find distance travelled in 1 hour (interpolation):

```
s_1 = 25.75 # distance in km
t_1 = 15    # elapsed time in minutes
s_2 = 171.7
t_2 = 100

t = 60

w = (t - t_1)/(t_2 - t_1)
s = (1 - w)*s_1 + w*s_2

print ("{:.1f} km".format(s))

103.0 km
```

Always interpret results with critically to understand if it makes sense or is wrong. In the previous example we certainly expected something between 25.75 and 171.7 km (our range ends) furthermore since we are looking for the distance at a time which is almost halfway the interval, the result will be somehow in the middle or around 98.6 km. This is indeed more or less what we have got. This simple reasoning should be applied everytime you have a result to quickly judge it.

If we believe the relation between our variable stays the same, we can use the same formula to *extrapolate* values *outside* our initial sample. For example if we keep the same constant velocity in our trip we could check the distance travelled after 3 hours:

```
s_1 = 25.75 # distance in km
t_1 = 15    # elapsed time in minutes
s_2 = 171.7
t_2 = 100

t = 180

w = (t - t_1)/(t_2 - t_1)
s = (1 - w)*s_1 + w*s_2

print ("{:.1f} km".format(s))

309.1 km
```

6.1.1 Log-linear interpolation

When the variable we would like to interpolate has an exponential relation with the unknown we can fall back to the previous case by simply applying the logarithm on both sides of the relation.

Doing so the previous formulas apply again except that at the end we have to exponentiate to get back the original variable. Assume the followign is the relationshipo between p and h, two generic variables:

$$p = \exp(c \cdot h)$$

Applying the logarithm to both sides of the equation gives:

$$s = \log(p) = \log(\exp(c \cdot h)) = c \cdot h$$

which is a linear relation between the new variable s and h. At this point we can use the results of the previous section to interpolate for values of the quantity s, just remeber to exponentiate the result to get the correct p. In formulas:

$$w = \frac{h - h_1}{h_2 - h_1}$$

$$s = (1 - w) \cdot s_1 + w \cdot s_2 \quad (\text{remember now } s = \log(p))$$

$$p = \exp(s)$$

Example 2

Atmospheric pressure decreases with the altitude (i.e. the highest you flight the lower is the pressure) following an exponential law:

$$p = p_0 \cdot e^{-\alpha h}$$

where

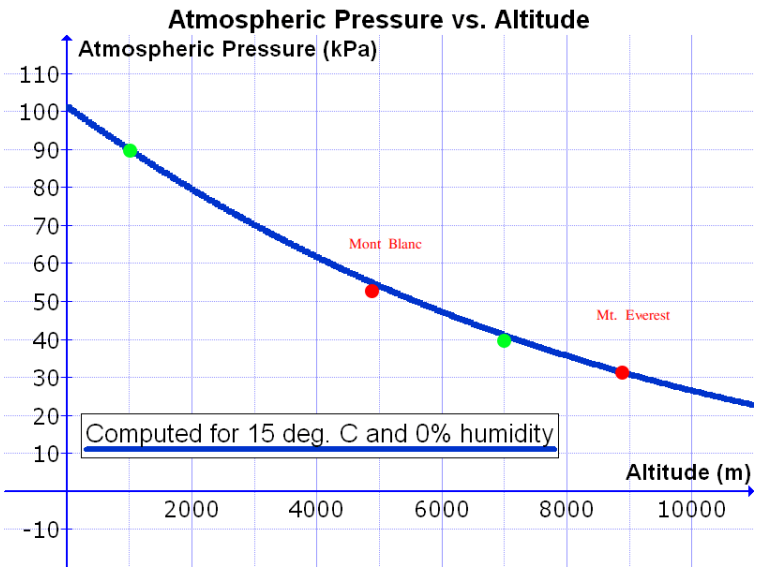
- h is the altitude
- p_0 is the pressure at sea level
- α is a constant

Taking the logarithm of each side of the equation I get a linear relation which can be interpolated as done before:

$$\tilde{s} = \log(p) = \log(p_0 \cdot e^{-\alpha h}) \propto -\alpha \cdot h$$

Now assume that we have measured $p_1 = 90$ kPa ($h_1 = 1000$ m) and $p_2 = 40$ kPa ($h_1 = 7000$ m) what will be the atmospheric pressure on top of the Mont Blanc (4812 m) ? and on top of Mount Everest (8848 m) ?

```
# pressure on top of the Mont Blanc (interpolation)
from math import log, exp

# first we take the logarithm of our measurements to use the linear
# relation to interpolate
h_1 = 1000 # height in meters
s_1 = log(90) # logarithm of the pressure at heigth h1
h_2 = 7000 # height in meters
s_2 = log(40) # logarithm of the pressure at heigth h2
```



Atmospheric pressure versus altitude (wikipedia). Green points represent our measurements, red points represent interpolation/extrapolation.

```
h = 4812

w = (h - h_1)/(h_2 - h_1)
s = (1 - w)*s_1 + w*s_2

print ("{:.1f} kPa".format(exp(s)))

53.8 kPa
```

```
# pressure on top of the Mount Everest (extrapolation)
from math import log, exp

# first we take the logarithm of our measurements to use the linear
# relation to interpolate
h_1 = 1000 # height in meters
s_1 = log(90) # logarithm of the pressure at heigth h1
h_2 = 7000 # height in meters
s_2 = log(40) # logarithm of the pressure at heigth h2

h = 8848

w = (h - h_1)/(h_2 - h_1)
s = (1 - w)*s_1 + w*s_2

print ("{:.1f} kPa".format(exp(s)))

31.2 kPa
```

6.2 Discount curve interpolation

Now we can come back to finance and using what we have just learnt try to write a function which interpolates some given discount factors.

Needed data:

- a list of pillars dates specifying the value dates of the given discount factors, $t_0, ..., t_{n-1}$;
- a list of given discount factors, $D(t_0), ..., D(t_{n-1})$;
- a pricing date ('today' date) which corresponds to $t = 0$.

The input argument to the function will be the value date at which we want to interpolate the discount factor. Since the discount factor can be expressed as $D = e^{-r(T-t)}$ the function will use a log-linear interpolation to return the value at a date not included in the given pillars.

$$D(t) = \exp\Big((1-w)\cdot\ln(D(t_i)) + w\cdot\ln(D(t_{i+1}))\Big); \quad w = \frac{t-t_i}{t_{i+1}-t_i}$$

where i is such that $t_i \leq t \leq t_{i+1}$. More technically we can say that we are doing a linear interpolation over time in the log space:

$$d(t_i) := \ln(D(t_i))$$

$$d(t) = (1-w)d(t_i) + wd(t_{i+1}); \quad w = \frac{t-t_i}{t_{i+1}-t_i}$$

$$D(t) = \exp(d(t))$$

where again i is such that $t_i \leq t \leq t_{i+1}$

Instead of reinventing the wheel and perform the interpolation with our own code, we'll use the function `interp` provided by the `python` module `numpy`; this function linearly interpolates the points to estimate the value of f at some x. Say we want to interpolate the points at $x = 2.5$ given the following values:

```
import numpy as np

xp = [0, 1, 5]
fp = [0, 2, 4]
np.interp(2.5, xp, fp)

2.75
```

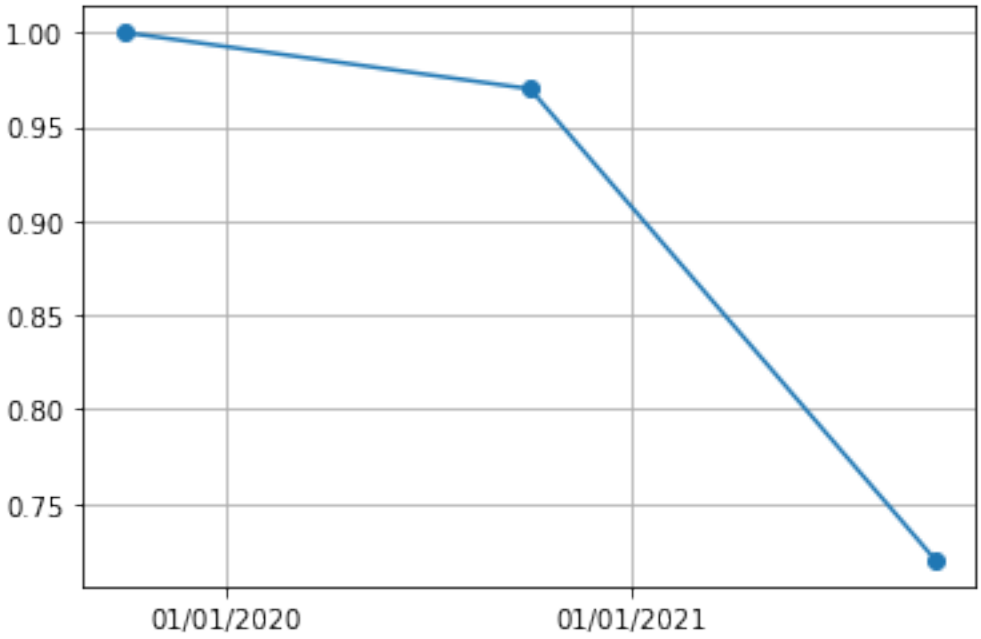
Assume we have three discount factors instead:

```
# import modules and objects that we need
from datetime import date
import numpy, math
from matplotlib import pyplot as plt
import matplotlib.dates as mdates
# with this notation we tell python to use mdates as an alias
# for matplotlib.dates
```

```
# define the input data
today_date = date(2019, 10, 1)

pillar_dates = [date(2019, 10, 1), date(2020, 10, 1), date(2021, 10, 1)]
discount_factors = [1.0, 0.97, 0.72]
```

Let's see what this fake discount curve looks like when plotted on a graph:

```
plt.plot(pillar_dates, discount_factors, marker='o')
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%m/%d/%Y'))
plt.gca().xaxis.set_major_locator(mdates.YearLocator())
plt.grid(True)
plt.show()
```



Since it is a computation that from now on we need to perform quite often it is convenient to write a function that compute the discount factor at an arbitrary date.

```
# define the df function
def df(d):
    # first thing we need to do is to apply the logarithm function
    # to the discount factors since we are doing log-linear and
    # not just linear interpolation
    log_discount_factors = []
    for discount_factor in discount_factors:
        log_discount_factors.append(math.log(discount_factor))

    # perform the linear interpolation of the log discount factors
    interpolated_log_discount_factor = \
```

```
        numpy.interp(d, pillar_dates, log_discount_factors)

    # return the interpolated discount factor
    return math.exp(interpolated_log_discount_factor)
```

This is almost OK, **but it won't work** because `numpy.interp` only accepts numbers/lists of numbers as arguments i.e. it doesn't automatically convert or interpret dates as numbers and doesn't know how to interpolate them. So we need to do the conversion ourselves before passing the dates into the interpolation function. The following updated version of our function converts the pillar dates into "pillar days" i.e. each date is replaced by the number of days today (t_0):

```
def df(d):
    # first thing we need to do is to apply the logarithm function
    # to the discount factors since we are doing log-linear and
    # not just linear interpolation
    log_discount_factors = []
    for discount_factor in discount_factors:
        log_discount_factors.append(math.log(discount_factor))

    # convert the pillar dates to pillar 'days'
    # i.e. number of days from today
    # to write shorter code we can use this NEW notation
    # which condenses for and list creation in one line
    pillar_days = \
        [(pillar_date - today_date).days for pillar_date in pillar_dates]

    # obviously we need to do the same to the value date
    # argument of the df function
    d_days = (d - today_date).days

    # perform the linear interpolation of the log discount factors
    interpolated_log_discount_factor = \
        numpy.interp(d_days, pillar_days, log_discount_factors)

    # return the interpolated discount factor
    return math.exp(interpolated_log_discount_factor)
```

Now we can use the `df` function to get discount factors on value dates between the given pillar dates:

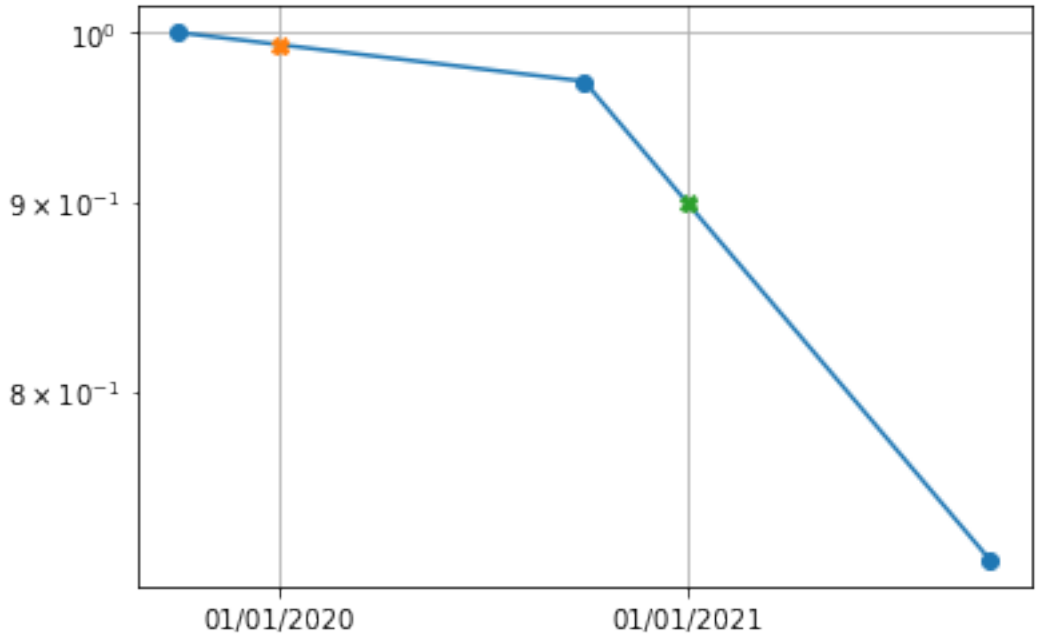
```
d0 = date(2020, 1, 1)
df0 = df(d0)
print (df0)

0.9923728228571693
```

```
d1 = date(2021, 1, 1)
df1 = df(d1)
print (df1)
```

```
0.8997999273630835
```

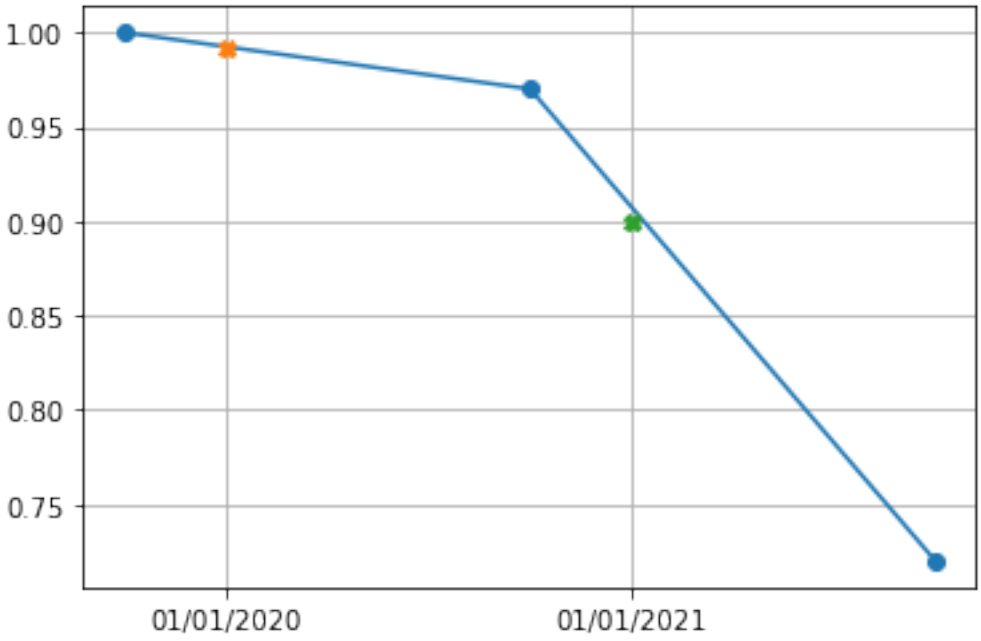
Another very useful way to check the correctness of a result is by plotting data, so let's see what these look like when plotted on a semi-log graph and if they make sense:

```
from matplotlib import pyplot as plt
import matplotlib.dates as mdates

plt.semilogy(pillar_dates, discount_factors, marker='o')
plt.semilogy(d0,df0 , marker='X')
plt.semilogy(d1,df1 , marker='X')
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%m/%d/%Y'))
plt.gca().xaxis.set_major_locator(mdates.YearLocator())
plt.grid(True)
plt.show()
```



Let's see what these look like when plotted on a linear graph instead:

```
from matplotlib import pyplot as plt
import matplotlib.dates as mdates
plt.plot(pillar_dates, discount_factors, marker='o')
plt.plot(d0,df0 , marker='X')
plt.plot(d1,df1 , marker='X')
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%m/%d/%Y'))
plt.gca().xaxis.set_major_locator(mdates.YearLocator())
plt.grid(True)
plt.show()
```



6.3 Calculating Forward Rates

A forward rate is an interest rate applicable to a financial transaction that will take place in the future. Forward rates are calculated from the spot rate and by exploiting no arbitrage condition i.e investing at rate r_1 for the period $(0, T_1)$ and then *reinvesting* at rate $r_{1,2}$ for the time period (T_1, T_2) is equivalent to invest at rate r_2 for the full time period $(0, T_2)$ (two investors shouldn't be able to earn money from arbitraging between different interest periods). That said:

$$(1 + r_1 T_1)(1 + r_{1,2}(T_2 - T_1)) = 1 + r_2 T_2$$

Solving for $r_{1,2}$ leads to

$$F(T_1, T_2) = r_{1,2} = \frac{1}{T_2 - T_1}\left(\frac{D(T_1)}{D(T_2)} - 1\right) \quad (\text{where } D(T_i) = \frac{1}{1 + r_i T_i})$$

```
from datetime import date
import numpy, math

today_date = date (2019, 1, 1)

pillar_dates = [date(2019 , 1 ,1),
                date(2020, 1, 1),
                date(2021, 10 ,1)]
discount_factors = [1.0, 0.97, 0.72]
```

```
def forward_rate(t1, t2):
    return 365.0/(t2-t1).days * (df(t1) / df(t2) - 1)

forward_rate(date(2019, 2, 1), date(2019, 8, 1))
```

6.3.1 2008 Financial Crisis

Looking at the historical series of the Euribor (6M) rate versus the Eonia Overnight Indexed Swap (OIS-6M) rate over the time interval 2006-2011 it becomes apparent how before August 2007 the two rates display strictly overlapping trends differing of no more than 6 bps.

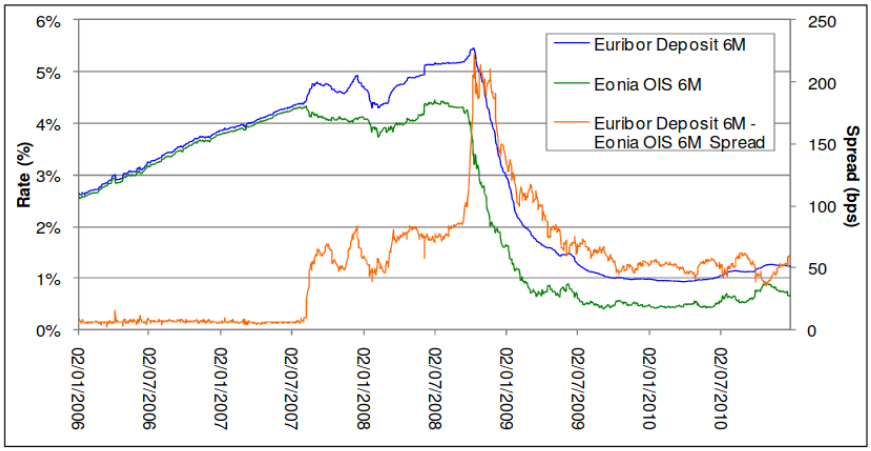


Figure 1: historical series of Euribor Deposit 6M rate versus Eonia OIS 6M rate. The corresponding spread is shown on the right axis (Jan. 06 – Dec. 10 window, source: Bloomberg).

In August 2007 however we observe a sudden increase of the Euribor rate and a simultaneous decrease of the OIS rate that leads to the explosion of the corresponding basis spread, touching the peak of 222 bps in October 2008, when Lehman Brothers filed for bankruptcy. Successively the basis has sensibly reduced and stabilized between 40 bps and 60 bps (notice that the pre-crisis level has never been recovered). The same effect is observed for other similar couples of series, e.g. Euribor 3M vs OIS 3M.

The reason of the abrupt divergence between the Euribor and OIS rates can be explained by considering both the monetary policy decisions adopted by international authorities in response to the financial turmoil, and the impact of the credit crunch on the credit and liquidity risk perception of the market, coupled with the different financial meaning and dynamics of these rates.

- The Euribor rate is the reference rate for over-the-counter (OTC) transactions in the Euro area. It is defined as the rate at which Euro interbank Deposits are being offered within the EMU zone by one prime bank to an
  The rate fixings for a strip of 15 maturities (from one day to one year) are constructed as the average of the rates submitted (excluding the highest and lowest 15% tails) by a panel of 42 banks, selected among the EU banks with the highest volume of business in the Euro zone money markets, plus some large international bank from non-EU countries with important euro zone operations. **Thus, Euribor rates reflect the average cost of funding of banks in the interbank market at each given maturity. During the crisis the solvency and solidity of the whole financial sector was brought into question and the credit and liquidity risk and premia associated to interbank counterparties sharply increased.** The Euribor rates

immediately reflected these dynamics and raise to their highest values over more than 10 years. As seen in the plot above, the Euribor 6M rate suddenly increased on August 2007 and reached 5.49% on 10th October 2008.

- The Eonia rate is the reference rate for overnight OTC transactions in the Euro area. It is constructed as the average rate of the overnight transactions (one day maturity deposits) executed during a given busines by a panel of banks on the interbank money market, weighted with the corresponding transaction volumes. **The Eonia Contribution Panel coincides with the Euribor Contribution Panel, thus Eonia rate includes information on the short term (overnight) liquidity expectations of banks in the Euro money market. It is also used by the European Central Bank (ECB) as a method of effecting and observing the transmission of its monetary policy actions. During the crisis the central banks were mainly concerned about restabilising the level of liquidity in the market, thus they reduced the level of the official rates.** Furthermore, the daily tenor of the Eonia rate makes negligible the credit and liquidity risks reflected on it: for this reason the OIS rates are considered the best proxies available in the market for the risk-free rate.

As a practial result of the divergence of the two indices, after the 2008 financial crisis, it is not possible anymore to use a single discount curve to correctly price forward rates of all tenors. For example, if we want to calculate the net present value of a forward 6-month libor coupon, we need to simultaneously use two different discount curves:

- the 6-month libor curve for determining the forward rate;
- the EONIA curve for discounting the expected cash flow.

We are now going to explore how to implement the following calculation:

$$\text{NPV} = D_{\text{EONIA}}(T_1) \times \frac{1}{T_2 - T_1}\left(\frac{D_{\text{LIBOR}}(T_1)}{D_{\text{LIBOR}}(T_2)} - 1\right)$$

Every object oriented language allows for a design pattern, that uses containers (in `python` dictionaries) to group together data, then having functions operate on those containers (or dictionaries), perhaps with a few additional parameters. This is easily doable with **classes**.

Now we can finally write a `DiscountCurve` class which contains the pillar dates and pillar discount factors as attributes and which has methods for calculating the discount factor and forward rate at arbitrary dates.

```
import math
import numpy
from datetime import date

class DiscountCurve:

    def __init__(self, today, pillar_dates, discount_factors):
        self.today = today
        self.pillar_dates = pillar_dates
        self.discount_factors = discount_factors

    def df(self, d):
        log_discount_factors = \
```

```
          [math.log(discount_factor)
           for discount_factor in self.discount_factors]
        pillar_days = [(pillar_date - self.today).days
                       for pillar_date in self.pillar_dates]
        d_days = (d - self.today).days
        interpolated_log_discount_factor = \
            numpy.interp(d_days, pillar_days, log_discount_factors)
        return math.exp(interpolated_log_discount_factor)

    def forward_rate(self, d1, d2):
        return (self.df(d1) / self.df(d2) - 1.0) * \
                (365.0 / ((d2 - d1).days))
```

Now that we have this useful class let's test it by creating EONIA and LIBOR curve object, and then compute discount factors and forward rates. Note that in the following we use the `parameter=argument` syntax just to make it really clear what we are doing, it is not necessary but it's just for clarity.

```
eonia_curve = DiscountCurve(today=date(2019, 10, 1),
                            pillar_dates=[date(2019, 10, 1),
                                          date(2020, 10, 1),
                                          date(2021, 10, 1)],
                            discount_factors=[1.0, 0.95, 0.8])

# build the Libor curve object
libor_curve = DiscountCurve(today=date(2019, 10, 1),
                            pillar_dates=[date(2019, 10, 1),
                                          date(2020, 4, 1),
                                          date(2020, 10, 1)],
                            discount_factors=[1.0, 0.98, 0.82])

# Let's compute the discount factor of the two curves
# on the 2020-5-1
print (eonia_curve.df(date(2020, 5, 1)))
print (libor_curve.df(date(2020, 5, 1)))

0.9705901255781632
0.9517777485424973
```

```
# Let's compute now the 6m forward rate at 1-4-2020
        print (eonia_curve.forward_rate(date(2019, 10, 1),
                                        date(2020, 10, 1)))

        print (libor_curve.forward_rate(date(2020, 4, 1),
                                        date(2020, 10, 1)))

0.05248777681909687
0.3891776622684259
```

Chapter 7

Swaps and Bootstrapping

In this chapter the Overnight Index Swap contract is reviewed and new functionalities to compute its net present value (NPV) will be added to our financial module. Beside financial arguments the another very important mathematical technique is introduced: *bootstrapping*.

7.1 Overnight Index Swap

Overnight Index Swaps (OIS) are products which pay a floating coupon, determined by overnight rate fixings over the reference periods, against a fixed coupon. Interest rate swaps are usually used to mitigate the risks of fluctuations of varying interest rates, or to benefit from lower interest rates. We will always look at these products from the point of view of the **receiver of the floating leg**. By definition an OIS is defined by:

- a notional amount N;
- a start date d_0;
- a sequence of payment dates $d_1, ..., d_n$;
- a fixed rate K.

In the following and for simplicity we are assuming that the fixed and floating legs of our OIS have the same notional and payment dates, although this is not necessarily always the case in practice.

To evaluate the net present value of such products the cash flows of each leg have to be calculated; today's NPV then is the sum of all the discounted cash flows.

Floating leg: at each payment date, the floating leg pays a cash flow determined as follows:

$$f_{\text{float},\, i} = N\left\{\prod_{d=d_{i-1}}^{d=d_i-1}\left(1 + r_{o/n}(d)\cdot\frac{1}{360}\right) - 1\right\}$$

This formula is valid for an EONIA swap, i.e. for OIS swaps in EUR, other currencies might have different conventions. The $\frac{1}{360}$ fraction appears because EONIA rates are quoted using the ACT/360 daycount convention and here we're making the simplifying assumption of ignoring weekends and holidays, so we assume that each overnight rate is valid for only one day. The sum of the discounted expected values of these cash flows is

$$\text{NPV}_{\text{float}} = \sum_{i=1}^{n} D(d_i)\mathbb{E}[f_{\text{float},\,i}]$$

where $D(d)$ is the discount factor with expiry d. On the other hand, by definition (remember practical lesson 4 with forward rates), we also have the following relationship

$$\mathbb{E}[f_{\text{float},\,i}] = N \cdot \left(\frac{D_{ois}(d_{i-1})}{D_{ois}(d_i)} - 1\right)$$

hence

$$\text{NPV}_{\text{float}} = N \cdot \sum_{i=1}^{n} D(d_i)\left(\frac{D_{ois}(d_{i-1})}{D_{ois}(d_i)} - 1\right)$$

where $D_{ois}(d)$ is the discount factor implied by OIS prices (we will see it better later).

The correct curve to use for discounting the flows of a collateralized contract, like OIS, is the one associated with the collateral. Since OIS contracts are collateralized with cash, and cash accrues daily interes at the overnight rate, the OIS curve is itself the correct curve with which to discount the flows of an OIS contract !

In summary, $D = D_{ois}$ so the NPV simplifies to

$$\begin{aligned}\text{NPV}_{\text{float}} &= N \cdot \sum_{i=1}^{n}[D(d_{i-1}) - D(d_i)] = \\ &= N \cdot [(D(d_0) - D(d_1)) + (D(d_1) - D(d_2)) + ... + (D(d_{n-1}) - D(d_n))] \\ &= N \cdot [D(d_0) - D(d_n)]\end{aligned} \tag{7.1}$$

Fixed leg: calculation for the fixed leg is simpler; each cash flow is equal to

$$f_{\text{fix},\,i} = N \cdot K \cdot \frac{d_i - d_{i-1}}{360}$$

so the NPV of the fixed leg is

$$\text{NPV}_{\text{fix}} = N \cdot K \cdot \sum_{i=1}^{n} D(d_i)\frac{d_i - d_{i-1}}{360}$$

Ultimately the aim will be to take a series of OIS quotations, and determine the discount factors implied by their prices. To do this we will build a pricing `class`, with a method which takes discount curve as input and produces the net present value of the OIS as the output. Then we will put this function inside a numerical optimizer to *invert* the process to determine the implied discount factors from their prices (market quotes).

```
class OvernightIndexSwap:
    # this method is called to build the instance,
    # n.b.: payment_dates should be a list of dates,
    # including the start date as the first element
    def __init__(self, notional, payment_dates, fixed_rate):
        self.notional = notional
        self.payment_dates = payment_dates
        self.fixed_rate = fixed_rate
```

```
    # this method takes a discount curve and calculates
    # the NPV of the floating leg using that curve
    def npv_floating_leg(self, discount_curve):
        # self.payment_date s[0] is the start date of the swap
        # self.payment_date s[-1] is the last payment date of the swap
        return self.notional * (discount_curve.df(self.payment_dates[0]) -
                                discount_curve.df(self.payment_dates[-1]))

    # this method takes a discount curve and calculates the NPV
    # of the fixed leg using that curve
    def npv_fixed_leg(self, discount_curve):
        npv = 0
        # we loop from i=1 up to but not including the length of the date list
        for i in range(1, len(self.payment_dates)):
            # we can do i-1, because the loop starts with i=1
            start_date = self.payment_dates[i-1]
            end_date = self.payment_dates[i]
            tau = (end_date - start_date).days / 360
            df = discount_curve.df(end_date)
            npv = npv + df * tau
        return self.notional * self.fixed_rate * npv

    # this method calculates the NPV of the OIS swap
    # n.b.: inside this method we call the other two
    # methods of the class on the same instance 'self',
    # using self.npv_XXX_leg(...), and we pass the
    # discount_curve we received as an argument
    def npv(self, discount_curve):
        float_npv = self.npv_floating_leg(discount_curve)
        fixed_npv = self.npv_fixed_leg(discount_curve)
        return float_npv - fixed_npv

from datetime import date

ois = OvernightIndexSwap(
    # the notional, one million
    1e6,
    # the list of product dates,
    # i.e. the start date then the payment dates
    [date(2020, 1, 1),
     date(2020, 4, 1),
     date(2020, 7, 1),
     date(2020, 10, 1),
     date(2021, 1, 1)],
    # the fixed rate, 2.5%
    0.025
)
```

To test our new class we have need a discount curve to use as input of the `npv` method. In the following example a fake curve is defined, then it is used with an OIS product.

```
from datetime import date
from finmarket import DiscountCurve

curve = DiscountCurve(date(2020, 1, 1),
                      [date(2020, 1, 1),
                       date(2021, 6, 1),
                       date(2022, 1, 1)],
                      [1.0, 0.98, 0.82])
curve.df(date(2020, 7, 1))

0.9929132520645648

ois.npv(curve)

-10990.364227052869
```

7.2 Bootstrapping

Now we are going to look at how extract a discount curve from OIS market data, via a process called *bootstrapping*. This is the ABC of financial mathematics, since you almost always need a discount curve to price any contract, especially if you are interested in its NPV. We are going to concentrate on EONIA swaps in order to build an EUR discount curve.

7.2.1 Building OIS instances

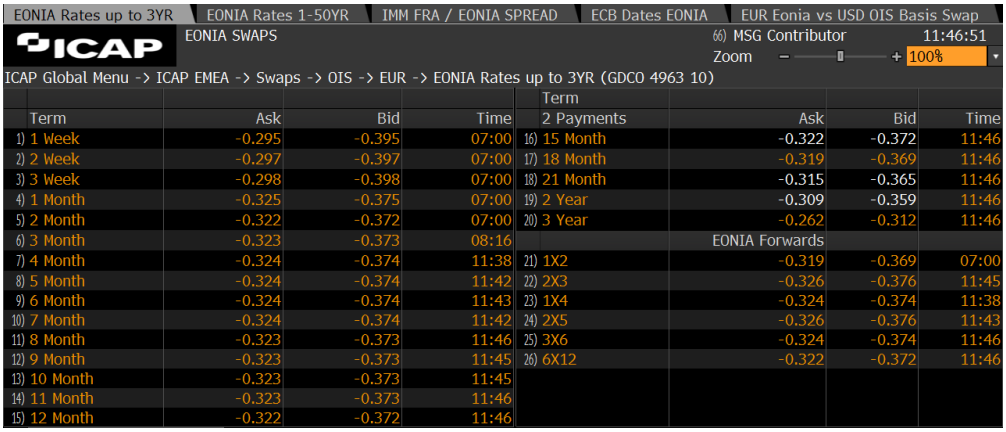
The first problem is actually getting data, the swap market quotes, from somewhere, and this is not actually as simple as it sounds.

The issue is that the EONIA swap market is over the counter (OTC) and it's not straightforward to access it. Unlike (some) listed futures, where anyone with a retail brokerage account can view and apply realtime prices, to trade in the EONIA swap market you have to be a financial institution or at least a large company and have an agreement with a broker which operates in the market. One of the main brokers in the OIS market is ICAP.

Though there exist some electronic platform in which market participants post bids and offers and other participants can apply them, in practice a lot of trading is still done over "voice", i.e. by phone or more commonly over chat. For convenience, however, Bloomberg provides a service which displays indicative realtime rates as provided by a selection of relevant brokers. (*n.b. interest rate swap quotes vary from standard price quotes of commonly traded instruments, they can appear puzzling because the quotes are effectively interest rates*)

As part of the quantitative analyst duties there is the set up an Excel spreadsheet which acquires this data from Bloomberg in realtime. From this spreadsheet, it is easy to export the data into other formats.

In the following we use a similarly created dataset (`ois_data.xlsx`) to derive our discount curve; with the help of the `pandas` module the dataset can be inspected:




```
import pandas, datetime

observation_date = datetime.date.today()

df = pandas.read_excel('ois_data.xlsx')
df.head()

   months  quote
0       1 -0.350
1       2 -0.347
2       3 -0.348
3       4 -0.350
4       5 -0.350
```

Then we can move this data to dictionary for later usage:

```
market_quotes = {}
for i in range(len(df)):
    key = df.loc[i, 'months']
    value = df.loc[i, 'quote']
    market_quotes[key] = value

print (market_quotes)

{1: -0.35, 2: -0.347, 3: -0.348, 4: -0.35, 5: -0.35, 6: -0.351, 7: -0.351, 8:
-0.351, 9: -0.351, 10: -0.351, 11: -0.35, 12: -0.35, 15: -0.35, 18: -0.348, 21:
-0.345, 24: -0.34, 36: -0.296, 48: -0.228, 60: -0.139, 72: -0.031, 84: 0.087,
96: 0.205, 108: 0.318, 120: 0.424, 132: 0.519, 144: 0.603, 180: 0.794, 240:
0.959, 300: 1.02, 360: 1.048, 480: 1.061, 600: 1.022, 720: 0.997}
```

Let's say we want to build a 15 months swap instance using data contained in `ois_data` file (be careful when doing this operation and doublecheck the units of rates, quotes, etc... in this case for example they are expressed in % so you need to multiply the quote by 0.01):

```
ois = OvernightIndexSwap(1e6,
                         [date(2019, 10, 23),
```

```
                        date(2020, 10, 23),
                        date(2020, 1, 23)],
                       market_quotes[12]*0.01
                      )
# print the last payment date (15 months after obs date)
ois.payment_dates[-1]
```

Clearly to use the `npv` method to calculate the OIS' NPV we need a discount curve with which to evaluate it and here comes to hand the bootstrapping technique !

7.2.2 Bootstrapping Technique

In finance, bootstrapping is a method for constructing a (zero-coupon) fixed-income yield curve from the prices of a set of coupon-bearing products, e.g. bonds and swaps. The term structure of spot returns is recovered from the bond yields by solving for them recursively, by forward substitution: this iterative process is called the *bootstrap method*. The usefulness of bootstrapping is that using only a few carefully selected zero-coupon products, it becomes possible to derive par swap rates (forward and spot) for all maturities given the solved curve.

To illustrate bootstrapping let's consider the following example which can be solved analytically: we have some coupon paying bond with maturities ranging from 1 to 5 years, each having a value of €100 and traded at par. To determine the zero-coupon yield curve preceed as follows:

1. at the end of the first year this 1^{st} bond will pay a coupon of €4 (= €100 * 4%) plus the principal amount (= €100) which sums up to €104 while the bond is trading at €100. Therefore, the 1-year spot rate S_{1y} can be calculated as, $€100 = €104/(1 + S_{1y})$;

2. at the end of second year the sum of the cash flows of the 2^{nd} bond can be compared to its trading price to compute the 2-year spot rate S_{2y} as $€100 = €5/(1 + S_{1y}) + €105/(1 + S_{2y})^2$, using the previously derived value of S_{1y};

3. at the end of third year the sum of the cash flows of the 3^{rd} bond can be compared to its trading price to calculate th 3-year spot rate S_{3y} as $€100 = €6/(1 + S_{1y}) + €6/(1 + S_{2y})^2 + €106/(1 + S_{3y})^3$, using S_{1y} and S_{2y} computed before;

4. repeat the same reasoning fo the other bonds.

Putting all together we can construct a system of equations (now omitting the currency symbol for simplicity):

$$
\begin{cases}
100 = \dfrac{104}{(1+S_{1y})} \\
100 = \dfrac{5}{(1+S_{1y})} + \dfrac{105}{(1+S_{2y})^2} \\
100 = \dfrac{6}{(1+S_{1y})} + \dfrac{6}{(1+S_{2y})^2} + \dfrac{106}{(1+S_{3y})^3} \\
100 = \dfrac{7}{(1+S_{1y})} + \dfrac{7}{(1+S_{2y})^2} + \dfrac{7}{(1+S_{3y})^3} + \dfrac{107}{(1+S_{4y})^4} \\
100 = \dfrac{8}{(1+S_{1y})} + \dfrac{8}{(1+S_{2y})^2} + \dfrac{8}{(1+S_{3y})^3} + \dfrac{7}{(1+S_{4y})^4} + \dfrac{108}{(1+S_{5y})^5}
\end{cases}
$$

This system can be solved quite easily: from the first equation S_{1y} can be derived, from the second S_{2y}, from the third S_{3y} and so on. So

$$100 = 104/(1 + S_{1y}) \quad \rightarrow \quad S_{1y} = 104/100 - 1 = 4\%$$

Moving to the second equation:

$$100 = 5/(1 + 0.04) + 105/(1 + S_{2y})^2 \quad \rightarrow \quad S_{2y}^2 + 2S_{2y} - 0.103030 = 0$$
$$S_{2y} = -1 \pm \sqrt{1 + 0.103030} = \begin{cases} \cancel{-2.05023} \\ 0.0503 \end{cases}$$

where the first solution has been discarded because negative. From the third one on it is not as simple to solve them analytically since involve third order (or more) equations. Anyway it is possible to solve them numerically and the results are:

default

years	coupon rate	bond price	spot rate
1	1.00 %	€100	4.00%
2	2.00 %	€100	5.03%
3	3.00 %	€100	6.08%
4	4.00 %	€100	7.19%
5	5.00 %	€100	8.36%

The last column of the table provide us with the terms to fill the zero-coupon yield curve. The very same mechanism can be generalized and extended to more maturities to get a more detailed yield curve. In general terms the previous system can be written as:

$$\begin{cases} f_1(S_1, p_1) = 0 \\ f_2(S_1, S_2, p_2) = 0 \\ f_3(S_1, S_2, S_3, p_3) = 0 \\ f_4(S_1, S_2, S_3, S_4, p_4) = 0 \\ \dots \end{cases}$$

where S_i are the unknown spot rate and p_i the prices of the considered products. The iterative procedure we have applied before exploits the first equation to find $S_1 = f_1^{-1}(p_1)$, the second to find $S_2 = f_2^{-1}(S_1, p_2)$ and so on and so forth; this algorithm works since each equation will determine exactly one *free* spot rate which is not already determined by the others.

7.2.3 Bootstrap as Minimization Problem

We can now describe the bootstrapping algorithm in general terms as follows:

1. define the set of yielding products , these will generally be coupon-bearing bonds;
2. derive discount factors for the corresponding terms;

3. *bootstrap* the zero-coupon curve, successively calibrating this curve such that it returns the prices of these inputs.

Instead of iteratively finding the solution of each equation as before, equivalently we could define a vector (list) of spot rates $\vec{S} = (S_1, S_2, S_3, \ldots)$ seeking for a particular $\vec{S_0}$ which solves the following equation:

$$F = f_1^2(S_1) + f_2^2(S_1, S_2) + f_3^2(S_1, S_2, S_3) + f_4^2(S_1, S_2, S_3, S_4) + \ldots = 0$$

Under this terms the bootstrapping technique can be considered as a minimization problem indeed we need to find $\vec{S_0}$ which *minimize* F, makes it as close as possible to 0.

Back to our Overnight Index Swap, the general idea here is to get the discount curve such that it prices correctly each OIS by minimizing the sum of the squared OIS NPVs:

$$\min_{curve}\Big\{\sum_{i=1}^{n}\text{NPV}(\text{ois}_i, \text{curve})^2\Big\}$$

A discount curve is characterized by pillar dates and the corresponding discount factors. The description of the problem we have given above does not, in theory, specifies any constraint on the pillar dates of the discount curve. However, the pillar dates determine the number of unknown variables (i.e. the dimensionality n of the optimization problem). A curve with n pillar dates has n discount factors (note that the first discount factor with value date equal to the today date, is constrained to 1). **In practice, therefore, it makes sense to choose the pillar dates in such a way that there are exactly the right number of degrees of freedom in the optimization to match data.** So the natural choice is to choose the pillar dates of the discount curve equal to the set of expiry dates of the swaps.

Therefore, once we've fixed $\vec{d}$ to be a vector of pillar dates equal to the expiry dates of the OIS swaps, and we use the notation $\vec{x}$ to represent the vector of pillar discount factors, then the problem becomes:

$$\min_{\vec{x}}\Big\{\sum_{i=1}^{n}\text{NPV}(\text{ois}_i, \text{curve}(\tilde{\text{d}}, \tilde{\text{x}}))^2\Big\}$$

again this is an optmization problem (**to find the minimum of the above expression as a function of** $\vec{x}$) which can sovled using one of the available numerical optimization routines in `python`.

So let's start by defining a set of OIS objects to cover all the maturities defined by the market data we have collected in `ois_data`.

```
from finmarkets import DiscountCurve, generate_swap_dates
import ois_data

pillar_dates = [ois_data.observation_date]

swaps = [] # container of the OIS objects

for quote in ois_data.quotes:
    swap = OvernightIndexSwap(
        # notional - doesn't really matter what we put here
        1e6,
```

```
            # payment dates
            generate_swap_dates(
                ois_data.observation_date,
                quote['months']
            ),

            # the fixed rate (in the file is expressed in percent)
            0.01 * quote['rate']
        )
        swaps.append(swap)
        pillar_dates.append(swap.payment_dates[-1])

pillar_dates = sorted(pillar_dates)
n_df_vector = len(pillar_dates)

type(pillar_dates), len(pillar_dates), pillar_dates[0], pillar_dates[-1]

(list, 34, datetime.date(2016, 11, 23), datetime.date(2076, 11, 23))
```

Now we can implement the minimization algorithm.

7.2.4 How Does the Minimization Algorithm Work ?

- Define an *objective function* i.e. the function that is actually minimized to reach our goal. In our case we want to find the discount curve which minimize the sum of the squared NPVs (swap quotes are considered their fair-values);
- set the intial value of the unknown parameters and their range of variability. We will set all the discount factors to 1 with a range of $[0.01, 100]$ (of course the first element of the list, today's discount factor will be set constant to 1);
- the *minimizer* will compute the objective function value;
- then will move the parameter values in such a way to find a smaller value of the objective function (e.g. *following* the derivative w.r.t. each parameter);
- the last two steps will be repeated until further variations of the $\vec{x}$ values won't change significantly the objective function (i.e. we have found a minimum of the function so the minimization process is completed !).

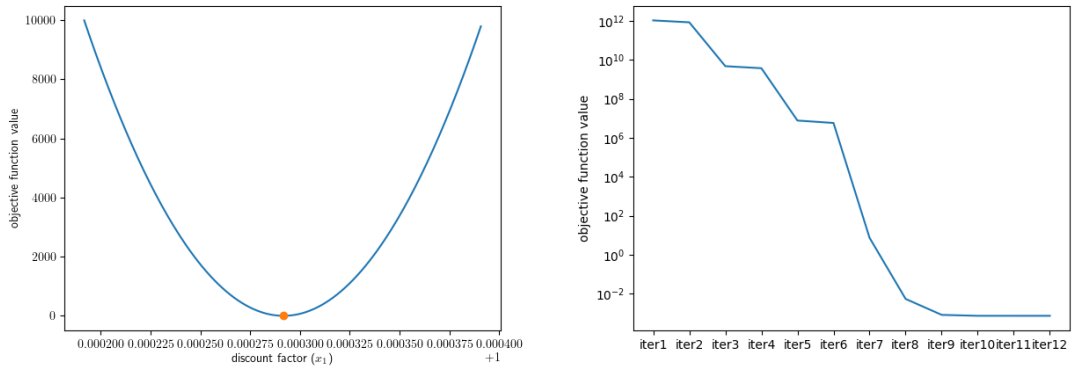
```
def objective_function(x):
    curve = DiscountCurve(
        ois_data.observation_date,
        pillar_dates,
        x
    )

    sum_sq = 0.0

    for swap in swaps:
```

```
        sum_sq += swap.npv(curve) ** 2

    return sum_sq
```

Of course we don't need to write our minimization algorithm since we can use the one provided by `python` which is defined in `scipy.optimize`, function `minimize`. The following code exactly implements the points desribed above:

```
from scipy.optimize import minimize
# initialize to 1 the x vector (random choice)
x0 = [1.0 for i in range(n_df_vector)]

# set wide constraints on the discount factors
# in the minimization problem the value of each x_i
# will be bound between these limits
bounds = [(0.01, 100.0) for i in range(n_df_vector)]

# in addition we have an additional constraint:
# we want the first pillar to be 1 (fixed)
# (because it has pillar date = today)
bounds[0] = (1.0, 1.0)

# finally we run the minimization
result = minimize(objective_function, x0, bounds=bounds)
```

Let's look at some diagnostic plots to check if the minimization was successful:



On the left the objective function value as a function of discount factor (x_1), on the right the value of objective function at each iteration.

```
# print the diagnostic of the minimization problem
result
```

```
fun: 0.0007370890117814888
hess_inv: <34x34 LbfgsInvHessProduct with dtype=float64>
```

```
jac: array([ 6.40061365e+05, -4.14161280e+01, -1.93984741e+01,  5.37030079e+00,
       3.02530223e+01,  5.95243457e+01,  9.05547884e+01,  1.24363295e+02,
       1.59125629e+02,  1.97071574e+02,  2.36973096e+02,  2.76028231e+02,
      -9.72901535e+02, -3.95831915e+02, -3.67814737e+02, -3.29982597e+02,
      -3.11447882e+02,  1.31701062e+02,  5.96919251e+02,  9.85290168e+02,
       1.18797301e+03,  1.09656582e+03,  7.03858626e+02,  7.86777868e+01,
      -6.55082634e+02, -1.36397212e+03, -1.89121870e+02,  1.93487828e+03,
       5.40226051e+02, -1.65947564e+02, -5.36852034e+02, -2.47750882e+03,
      -1.84414691e+02,  1.90090790e+03])
message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
   nfev: 875
    nit: 12
 status: 0
success: True
      x: array([1.        , 1.00029175, 1.00058831, 1.00089012, 1.00116802,
      1.00147021, 1.00176786, 1.00207128, 1.00236508, 1.00266885,
      1.00297281, 1.0032578 , 1.00356124, 1.00445968, 1.00529983,
      1.00614269, 1.00693061, 1.00906201, 1.0093198 , 1.00710112,
      1.0018986 , 0.99379504, 0.9833297 , 0.97101001, 0.95723164,
      0.9426886 , 0.92772535, 0.88314869, 0.8178113 , 0.76554845,
      0.71988664, 0.64350636, 0.59281978, 0.54547324])
```

```
# objective function value with starting point parameters
objective_function(x0)
```

```
1055841619695.9585
```

```
# objective function value with final values
objective_function(result.x)
```

```
0.0007370890117814888
```

The objective function at the end of the minimization is not exactly 0 (and rarely it will be) but its value is small enough for us to be satisfied, we started with 10^{13} and now it is 10^{-3} so 16 orders of magnitude smaller. This means that with the derived discount curve the NPV's of our OIS won't be identically 0 but so small that we can consider them as they were.

```
# define the discount curve object using the
# resulting discount factors (result.x)
curve = DiscountCurve(ois_data.observation_date, pillar_dates, result.x)
```

```
from datetime import date
curve.df(date(2059, 11, 23))
```

```
0.6278698804291626
```

```
# 50 years rate
import math
```

```
-math.log(curve.df(date(2059, 11, 23))) / 50
```

```
0.009308446615020075
```

```
list(result.x)
```

```
[1.0,
 1.0002917467402102,
 1.000588313127234,
 1.0008901199538132,
 1.0011680243827574,
 1.0014702089411056,
 1.0017678648937525,
 1.0020712764009663,
 1.0023650754871605,
 1.0026688489207223,
 1.0029728065322203,
 1.0032577961650246,
 1.003561243417754,
 1.004459676763001,
 1.0052998330195508,
 1.0061426892577996,
 1.006930607427503,
 1.009062012124004,
 1.0093198010277291,
 1.0071011202466504,
 1.00189860229147,
 0.9937950392360159,
 0.9833296977458316,
 0.9710100057905164,
 0.9572316430523317,
 0.9426886049809743,
 0.9277253536938298,
 0.8831486876894581,
 0.817811304643707,
 0.7655484543652669,
 0.7198866407284839,
 0.643506361425598,
 0.5928197767210946,
 0.5454732370629979]
```