Python for Finance
Lecture Notes

Matteo Sani

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

Contents

1 Introduction to `python` 5
- 1.1 What is `python` ? . . . 5
- 1.2 `Python` basics . . . 8
  - 1.2.1 Variables . . . 8
  - 1.2.2 Boolean expressions . . . 10
  - 1.2.3 String expressions . . . 11
  - 1.2.4 Mathematical expressions . . . 12
- 1.3 Modules . . . 14
- 1.4 Indented blocks and the `if/elif/else` statement . . . 17
- 1.5 Loops . . . 18
  - 1.5.1 `for` . . . 18
  - 1.5.2 `while` . . . 20

2 Data Containers 21
- 2.1 Lists . . . 21
- 2.2 Dictionaries . . . 24
- 2.3 Tuples . . . 27

3 Date and Time 29
- 3.1 Dates . . . 29

4 Python's Object Oriented Programming 31
- 4.1 Functions . . . 31
- 4.2 Variable scope . . . 33
- 4.3 Classes . . . 36
  - 4.3.1 Inheritance and Overriding Methods . . . 38

5 Data Manipulation and Its Representation 41
- 5.1 Getting Data . . . 41
  - 5.1.1 Manage Data . . . 43
  - 5.1.2 Unwanted observations and outliers . . . 43
  - 5.1.3 Handle Missing Data . . . 46
  - 5.1.4 Filter, Sort and Clean Data . . . 46
- 5.2 Plotting in `python` . . . 50
  - 5.2.1 Plot a graph given x and y values (scatter-plot) . . . 50
  - 5.2.2 Plotting an Histogram . . . 52

Chapter 1

Introduction to python

Python is one of the most widely used programming languages in the world, 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, meaning that more servers can be used to reach 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*).

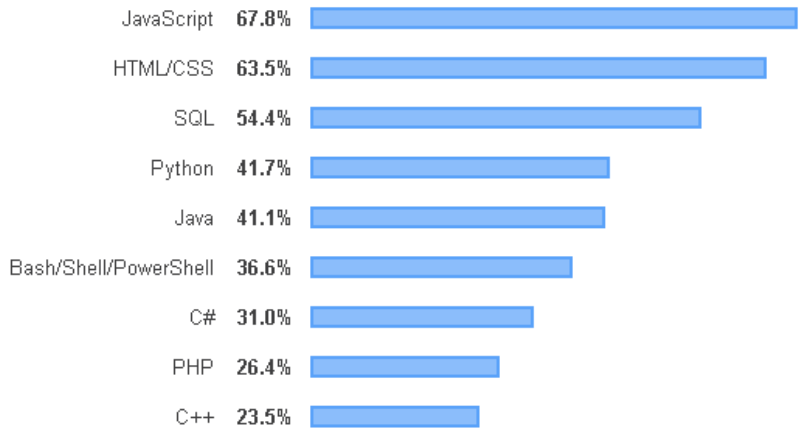


Figure 1.1: List of most used languages by developers according to Stack Overflow survey in 2019.

Figure 1.2: List of most loved languages by developers according to Stack Overflow survey in 2019.

Figure 1.3: List of most *dreaded* languages by developers according to Stack Overflow survey in 2019.

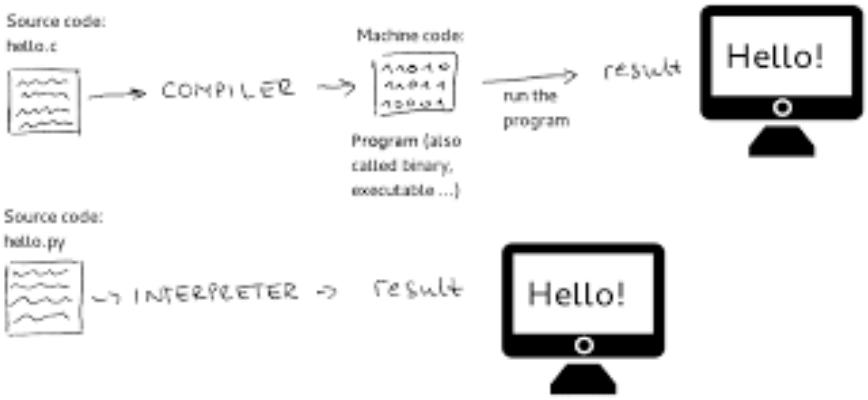


Figure 1.4: Interpreted vs compiled language

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.

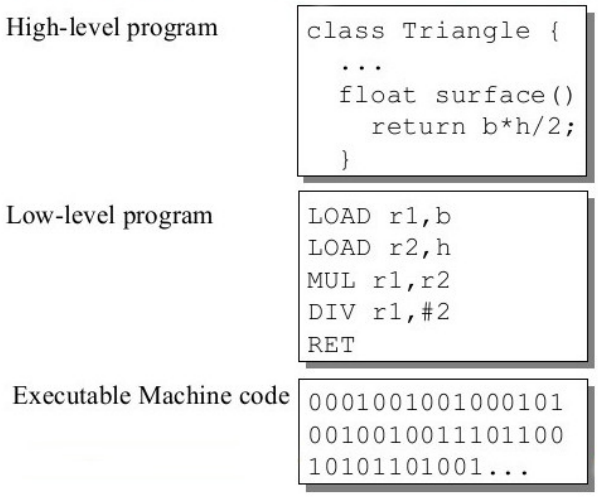


Figure 1.5: Human readable vs machine code

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 financial 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 writing this pages) is 3.8.5, but it is continuously 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 sizable 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 python 3.7.

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 a 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 (=) and printing a variable shows its content.

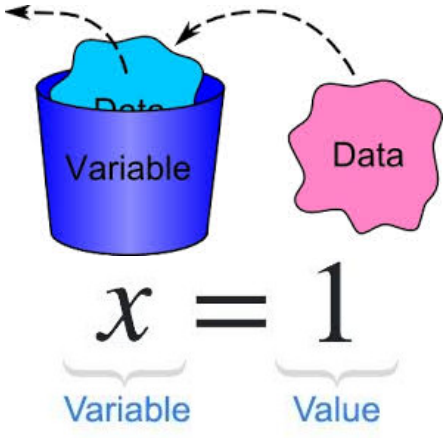


Figure 1.6: Graphical representation of a variable.

```
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'>
```

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 your 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-to Boolean operator symbol is a double = (==), to not be confused with the assignment 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 answers the question is 1 equal to 2:

```
1 == 2

False
```

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

```
1 != 2

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(var1, var2)`. 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 a float number and that has to be printed with only one 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 alignment 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 # integer 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 re-implement 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 data-set 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.

```
import math
dir(math)

Out[18]: ['__doc__',
          '__loader__',
          '__name__',
          '__package__',
          '__spec__',
          'acos',
          'acosh',
```

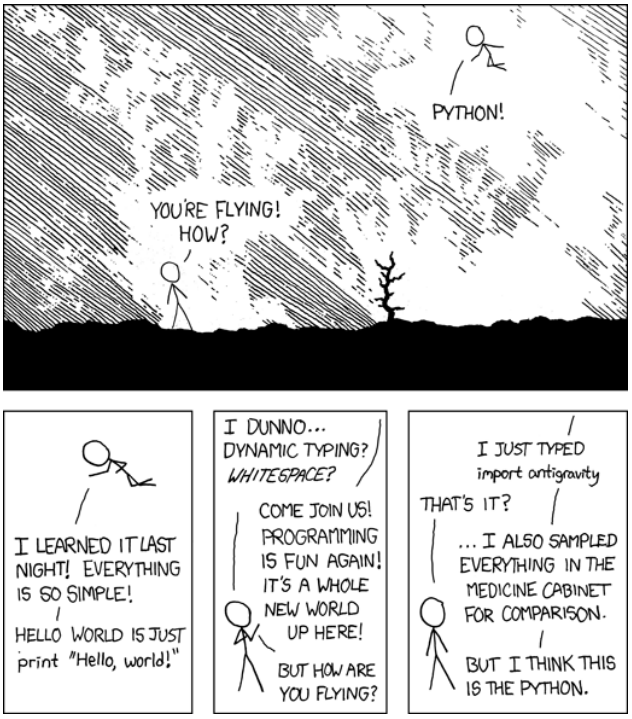


Figure 1.7: Python has many modules for download on the web...

```
          '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 only 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/elif/else` statement

Unlike other languages which uses parenthesis to isolate blocks of code `python` uses *indentation*. A first example of this peculiarity is given by `if/elif/else` 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 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 specify 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 possible 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 until 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 until 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 has index 0 actually). 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 = list([21, 32, 15])
mylist = [21, 32, 15]
print(mylist)
print (type(mylist))

[21, 32, 15]
<class 'list'>
```

```
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]

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

[21, 74, 85, 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*.

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
```

Read carefully the error messages usually they are very explicative and can help a lot in *debugging* (i.e. finding mistakes) in your programs.

There are two more nice features of `python` indexing:

- negative indices are like positive ones except that they starts from the last element;
- *slicing* which allows to specify a range of indices to select more items at once (if the first or last limits are missing slicing will start from the first or end with last index respectively).

```
print ("negative index -1 returns the last element:", mylist[-1])
print ("slice [1:3] returns items 1st and 2nd:", mylist[0:3])
print ("slice [:2] returns items 0th and 1st:", 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 1st and 2nd: [21, 74, 85]
slice [:2] returns items 0th and 1st: [21, 74]
slice [2:] returns items between the 2nd and the last: [85, 15, 188]
```

Needless to say that slicing with `[:]` returns the entire list.

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 elements 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 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 useful 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 assignment = 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 ("All values: ", adict.values())
for value in adict.values():
    print (value)
```

```
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` so it's important to know about them.

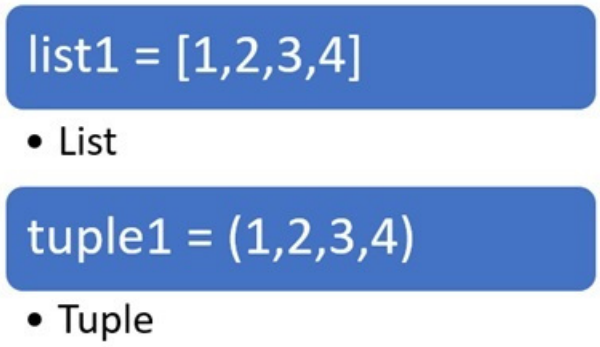


Figure 2.1: At first glance list and tuples look very similar, but they are not...

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 an 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 usage will help in manipulating dates.

3.1 Dates

As said dates are not usually included in a standard `python` tutorials, however since they are pretty essential for finance we are going to cover this topic in some detail. In `python` the date utilities mainly lives in the `datetime` module. We are also going to show `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 dates are created adding two months and three days to the first one.

```
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 conversion:

```
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, a string is converted to `datetime` object:

```
# a string can be converted 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 however the concepts 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. Functions are called by name passing the necessary parameters if any.

```
# 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 `my_function`. Below an example of function not returning anything.

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

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

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

```
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`, multiplying together all the results. The previous example is quite simple but recursion can be tricky sometimes so apply it with caution.

Functions input parameters can have 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 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 constant, in the second just the same list of numbers. In the latter case, being defined the default values for `n` and `c`, the function works perfectly well, fewer inputs are provided and the missing ones are replaced by their defaults.

When calling a function parameters 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 assigned according to their position.

Another nice feature of `python` 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 namespace 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 cannot be accessed directly inside functions but before using them they have to be named 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 few cases are outlined:

- 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 function 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 until 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 choose different names for each variable you are going to use in a program to avoid confusions 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 the 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 basically in every modern programming language like `python`, `Java` and `C++`. OOP is a programming model in which programs are organized around data, or objects, rather than functions and logic. **Any object can be 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 its state (member variables or attributes), and implementations of behavior (member functions or methods). Let's summarize here some terminology:

CLASS →	collection of functions that operate on some dataset
DATA ITEMS →	are called the **attributes** of the class
CLASS INSTANCE →	a specific collection of data belonging to the particular object we are representing
CLASS FUNCTIONS →	are called **methods** of the class, and act on **instances**

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 manufacturer, RAM size, CPU type,...). While examples of corresponding instances are: the Empire State Building (a specific building), Lassie (a very particular dog), or your computer, see Fig. 4.1.

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

```
from datetime import date

class Person:
```

First of all, if needed, we have to import the necessary modules, 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.

The Constructor Method

After declaring the class name, the constructor method must be defined. In `python`, this is denoted by `__init__()` regardless the class name. The `__init__` function, as every other method, takes `self` as the first argument, and then any number of arguments as desired by the programmer. The *constructor* allows to specify the initial state of a class by setting its attribute values. For this example that describes a Person, the programmer wants to know the name, the birthday and a job (this last one won't be initialize by the constructor).

The `self` parameter is used to create class attributes. Variables whose name starts with `self` have *class scope*, which means are available within each class method. To use the parameters and

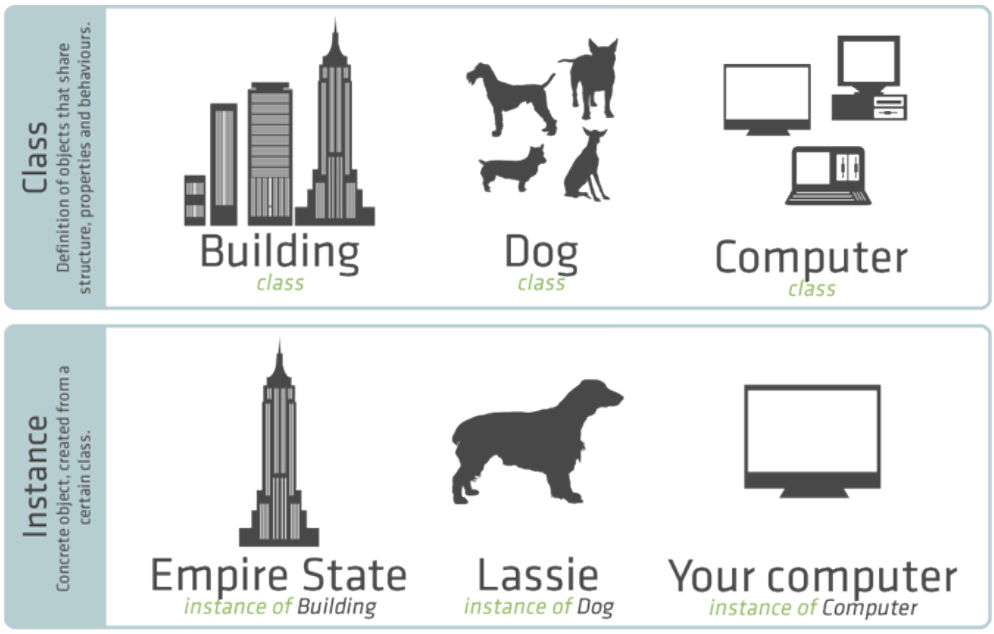


Figure 4.1: Graphical representation of a class instance.

associate them with a particular instance of the class, within the `__init__` method, create variables for each argument like this: `self.variableName = param`.

```
from datetime import date

# this is the class definition
# usually classes use camel naming convention
class Person:
    # the special method __init__ allows to instantiate a class
    # with an initial dataset
    def __init__(self, name, birthday):
        self.name = name
        self.birthday = birthday
        self.occupation = None # this attribute not set at instantiation
```

Now we that have a class definition that represent a generic person we can specialized it to some real person.

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

<class '__main__.Person'>
```

Essentially when we instantiate a class `python` first calls the `__init__` method and initializes the class attributes with the parameter we are passing.

Class Methods

We haven't yet defined any "person behaviour", so let's add a couple of methods to our class, one computing the person's age and the other setting its primary occupation.

```
class Person:
    def __init__(self, name, birthday):
        self.name = name
        self.birthday = birthday
        self.employment = None

    # this is a normal method and will work on some class attribute
    def age(self, d=date.today()):
        age = (d - self.birthday).days/365
        print ("{} is {:.0f} years old".format(self.name, age))

    def mainOccupation(self, occupation):
        self.employment = occupation
        print ("{}'s main occupation is: {}".format(self.name, self.employment))
```

To access class attributes and methods the . (dot) operator has to be used.

```
me.name

'Matteo'
```

```
me.age(date.today())

Matteo is 46 years old
```

4.3.1 Inheritance and Overriding Methods

Inheritance is basically the idea that different classes can have similar components, and in order to avoid repeating code, inheritance is used to link parent classes to descendant classes.

For example, in a fantasy story, there are heroes and monsters but both the heroes and the monsters are characters. And both dragons and orcs are monsters. Though dragons and orcs are different monsters, they share some qualities: they both have a color, they both have a size, they both have enemies. Orcs might have characteristics that dragons do not; for example, what kind of weapon does the orc carry? (Fig. 4.2). Inheritance allows the classes to share information relevant to multiple parts of the code.

Inheritance allows code to be reused and reduces the complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors). To see how it works in more detail two derived class will be implemented from `Person`: `Adult` and `Child`.

```
  class Adult(Person):
    def __init__(self, name, birthday, drv_license_id):
        Person.__init__(self, name, birthday) # this is a special syntax
        self.drv_license_id = drv_license_id
```

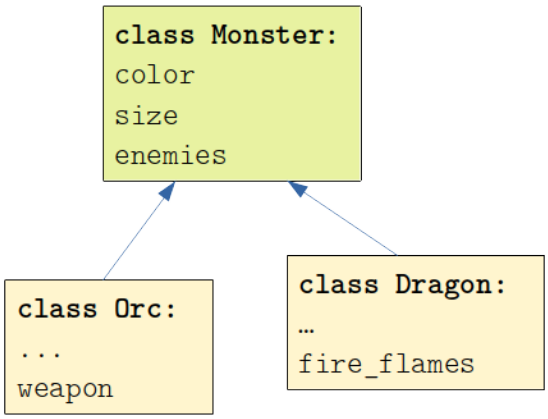


Figure 4.2: Example of inheritance: both `Dragon` and `Orc` inherits from `Monster`, but both add additional attributes that qualify better each "monster features".

```
class Child(Person):
    def mainOccupation(self):
        self.employment = "schoolchild"
        print ("{} is a {}".format(self.name, self.employment))
```

Inheritance is specified in parenthesis after the class name. `Child` still inherits from `Person` and modifies it by setting the only occupation allowed for a child, overriding the `mainOccupation` method. `Adult` class inherits from `Person` and extends it adding a new attribute (driving license id).

```
pippo = Adult("Goofy", date(1936, 1, 1), "A1234")

qui = Child("Huey", date(2014, 10, 9))
```

Behind the scenes, when instantiating a `Child` class, the constructor of `Person` is called and the attributes, `name` and `birthday` are initialised.

For the `Adult` class things are a little bit different because a new attribute has been added, so we need to code its own constructor where we first call explicitly the `Person`'s constructor (`Person.__init__(self,...)` and then we initialise the new attribute.

```
pippo.age()
pippo.mainOccupation("Comic's character")

Goofy is 85 years old
Goofy's main occupation is: Comic's character
```

```
qui.age()
qui.mainOccupation()

Huey is 6 years old
Huey is a schoolchild
```

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 (e.g. website, figures, twitter messages, CSV or Excel files...) and partially reflect its nature which can range from *unstructured* data (without 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 heterogeneous, 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 easily.

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 Separated 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 preliminary 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 irrelevant 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 irrelevant 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 re-scale 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:

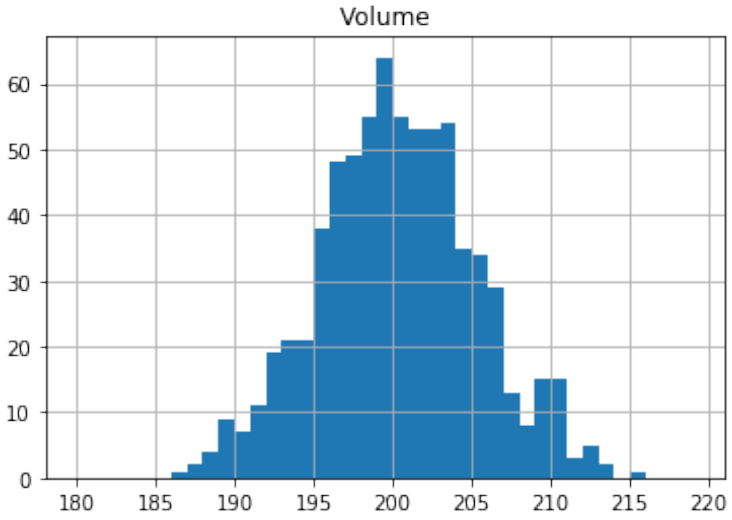
```
df1['Price'] = (df1['Price'] - df1['Price'].min()) \
    / (df1['Price'].max() - df1['Price'].min())
df1.head()

        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 transformation is called *standardization*, essentially we re-scale data to have 0 mean and standard deviation of 1:

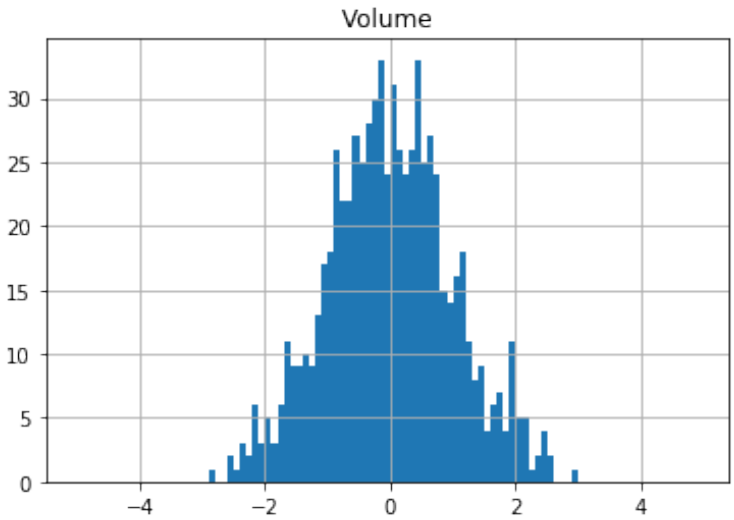
$$x_{new} = \frac{x - \mu}{\sigma}$$

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
```



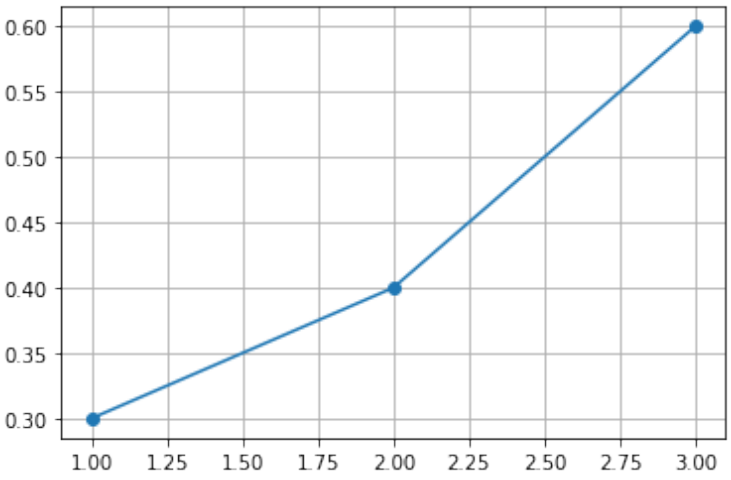
5.2 Plotting in `python`

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.

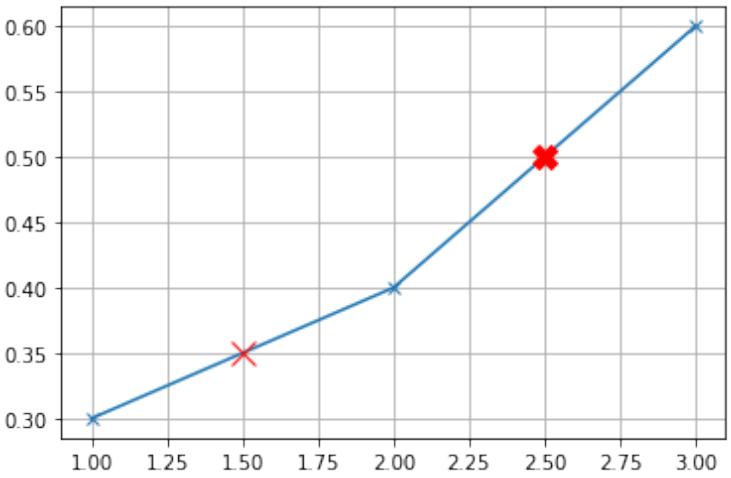
In order to do so we can use the `matplotlib` module which is specifically dedicated to plotting (pandas interface is based on the same module indeed). Let's look briefly to its capability by examples.

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()
```

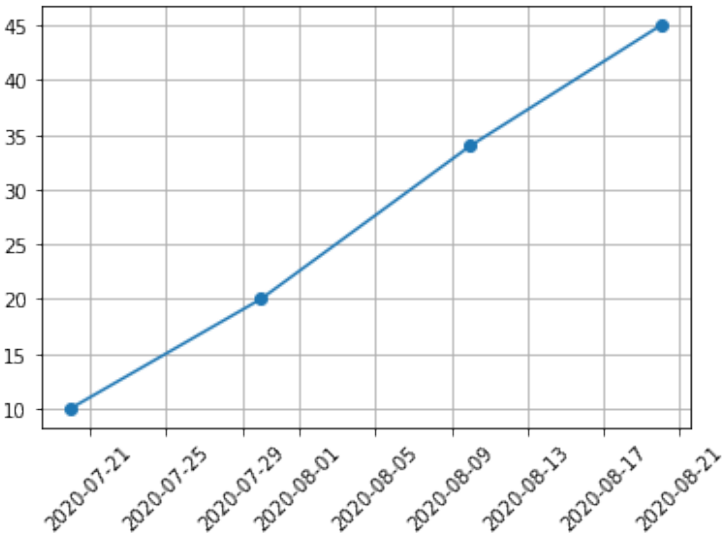



```
# if we want to plot specific points too

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()
```



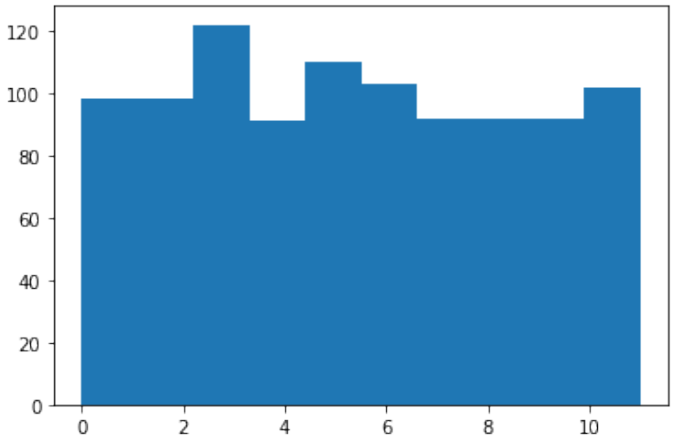
What if x values are dates ?

```
import datetime
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()
```

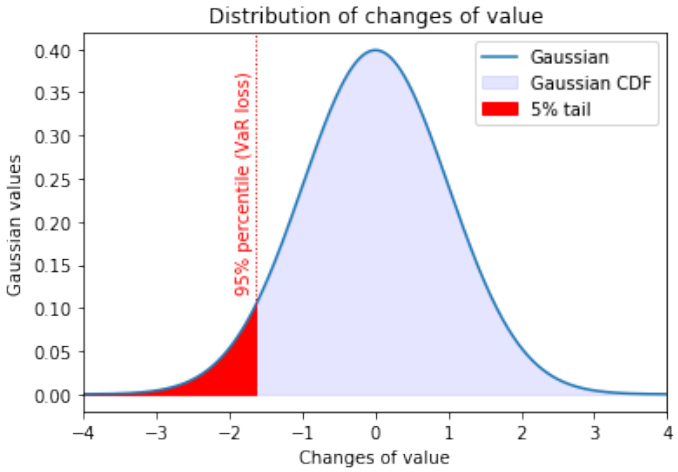


Plotting a Function

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>
```