lecture_1

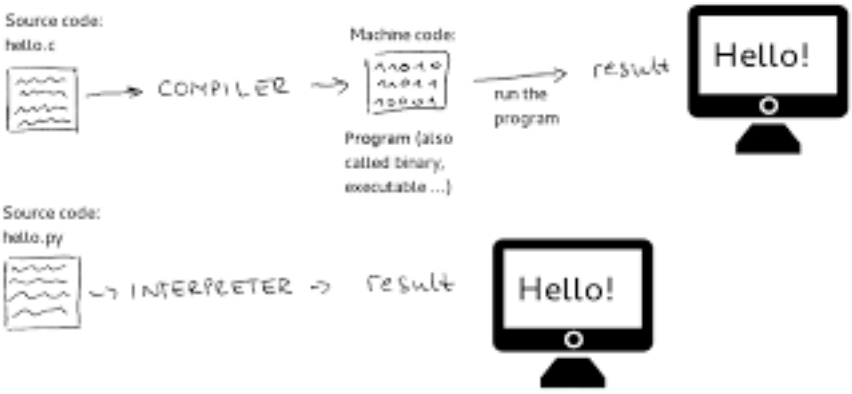
September 14, 2020

1 Introduction to `Python`

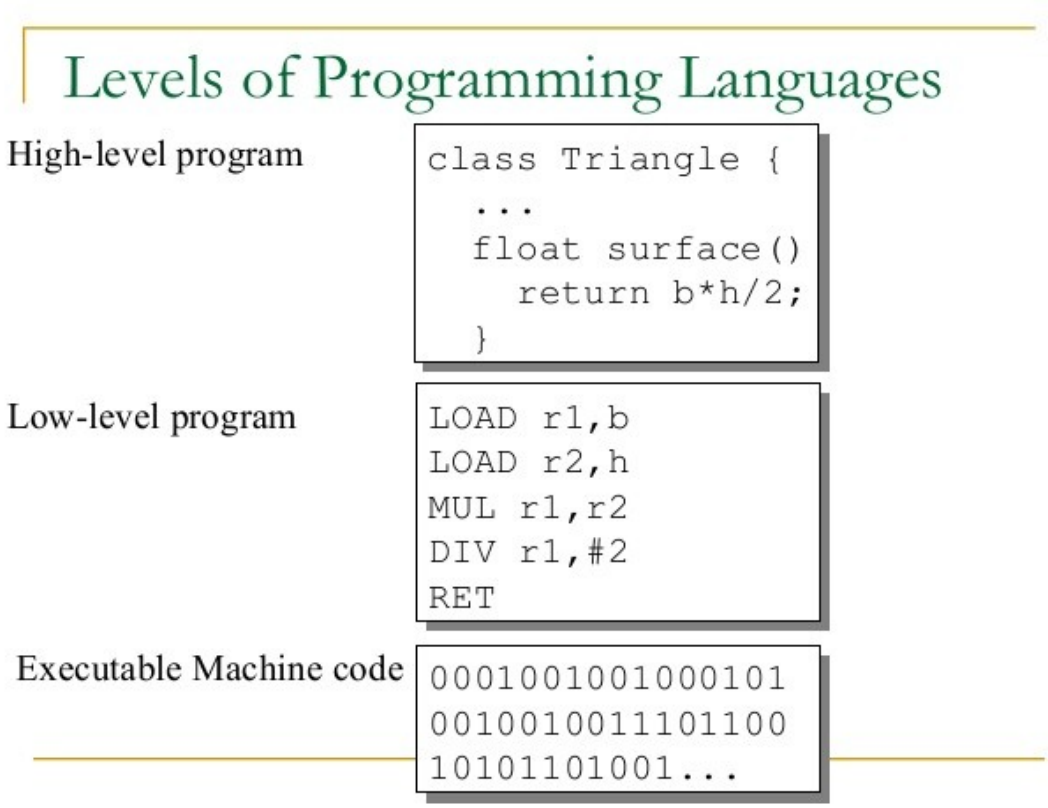
In the first two lessons of this course we'll take a quick tour of the `Python` programming language and see how to write a simple functions or classes which would actually be useful in a real-world finance environment.

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*).



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 in terms of readability of the code since programmming is almost like instructing the computer in plain English but it has drawbacks in term of perfomance since we have the intermediate step of the "translation" (just to give an idea the compilation of our C++ financial code takes more than one hour).



1.1.1 Which `Python` should I use ?

`Python`, as basically all programs, comes in different version and flavours as you can see by the number of updates the apps in your mobile phone receives.

The latest version is `3.8.5` (but it is continously evolving), however you'll 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 since moving to `Python3` would require a lot of work to adapt the code (this process is usually called *porting*).

We will go for `Python3.7` !

1.1.2 How can I use `Python` ?

Once you have installed a `Python` distribution there are various ways of actually using it.

- the most immediate way is to just execute `python.exe` on the command line to get a `Python` console for interacting with the interpreter;
- if you are learning `Python` or do some simple data analysis, `$Jupyter notebooks $(i.e. this document) allow to see the results of your code as you write it, as well as make notes, plot graphs beside it;`
- `if you are a programmer and want to do more complex things, you'll usually want to split your code between more files to manage your project more easily. For this last case an integrated development environment (IDE) can be very useful. An IDE is a graphical user interface which makes writing complex`

code easier by providing a text editor, a file browser, a debugger (a tool that helps you to spot mistakes in your code) all in one software application. Good example is PyCharm (https://www.jetbrains.com/pycharm/).

1.1.3 Online courses

Python popularity is growing every day so it is very easy to find good (and free) online courses looking into the web. Since in this course we do not have time to cover in depth the potentiality of this language I strongly suggest you to spend some time in watching one of them. One example could be

MITx: 6.00.1x Introduction to Computer Science and Programming Using Python
https://courses.edx.org/courses/course-v1:MITx+6.00.1x+2T2017_2/course/

1.2 Python basics

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

[1]:
```
print ("Hello world !")
```

```
Hello world !
```

[2]:
```
print ("Welcome")
print ("to")
print ("everybody")
```

```
Welcome
to
everybody
```

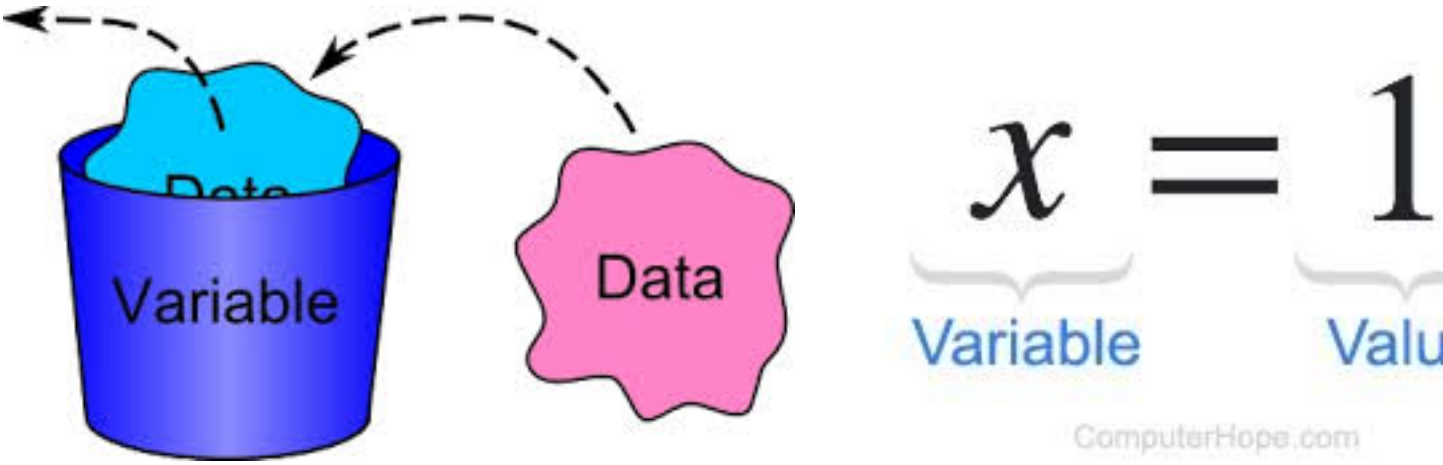
Good programming practice recommends to document the code you write (it is surprisingly easy to forget what you wanted to do in your code). In python you can add comments to the code starting your sentence with a hash character (#).

[3]:
```
# this is a comment and the next line prints "Ciao"
print ("Ciao") # comments like this are useful to explain what's going on in the
               # code you write
```

```
Ciao
```

1.3 Variables

Variables are essentially labels you stick to some data (e.g. a number, a string...). Variables and hence data they contain, can be used referenced and manipulated in throughout a program.



A variable can contain every kind of objects and is decleared using the = operator. To inspect the content of a variable it can be used the `print` statement.

[2]:
```
x = 9 # assign number 9 to variable named x
print (x)
```

```
9
```

[6]:
```
myphone = "Huawei P10Lite" # in this case the variable contains a string
print (myphone)
```

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

[8]:
```
print (type(x))

print (type(myphone)) # int->integer, str->string we will see later in more
                      # detail what is a string
```

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

From now on I can use `x` as an alias for a number or `myphone` as a string and manipuate their content for example I can add 5 to `x`:

[3]:
```
print (x+5)
```

```
14
```

1.3.1 Variable name rules

A `python` variable name 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 are reserved words as such you cannot use as variable names (e.g. `print`, `type`, `for`...**).**

To use GOOD variable names 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).

1.4 Mathematical expressions

[9]:
```
1 + 2
```

[9]:
```
3
```

[10]:
```
40 - 5
```

[10]:
```
35
```

[11]:
```
x * 20 # remember that we set x equal to 9
```

[11]:
```
180
```

[12]:
```
x / 4
```

[12]:
```
2.25
```

[13]:
```
print (type(2.25)) # this is a new type: floating-point value
```

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

[14]:
```
x // 4 # interger division - result will be truncated to the
       # corresponding integer (no rounding)
       # 11 / 3 = 3.666666 -> 11 // 3 = 3
```

[14]:
```
2
```

[15]:
```
y = 3
x ** y # x to the power of y
```

[15]:
```
729
```

[16]:
```
3 * (x + y)
```

[16]:
```
36
```

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

[12]:
```
print (x)
x = x + 1
```

```
print (x)
```

```
15
16
```

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

[17]:
```
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
```

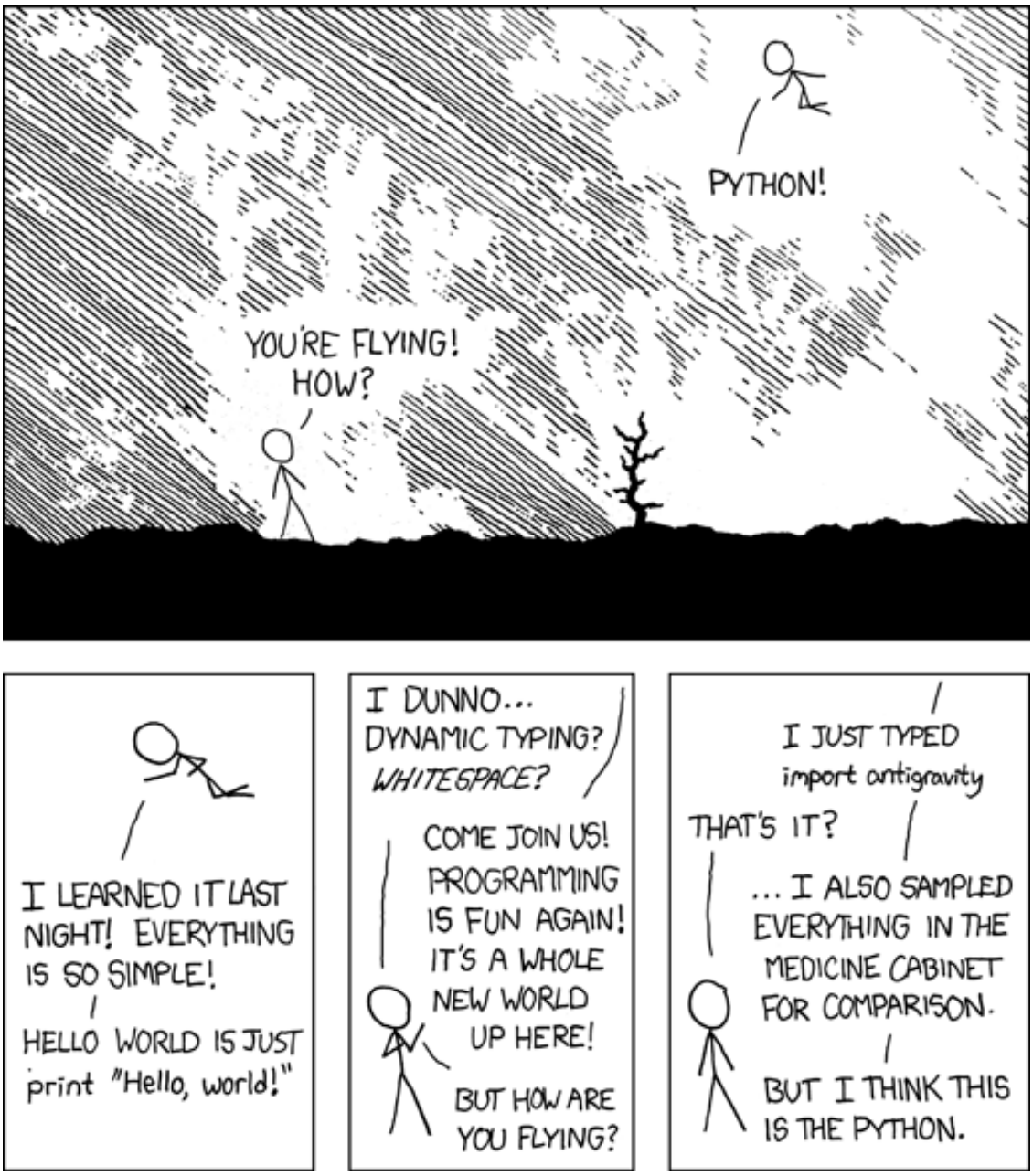
1.5 Modules

One very important feature of each language is the ability to reuse code in 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 every time you install it, it comes with a standard set of them. If you need more functionality, you can download more of them (there are zillions of packages out there) or you can of course write your own (which is 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.



In order to load a module in a `python` program you can use the `import` keyword. Information on a module can be retrieved using `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.** Let's see an example dealing with the `math` module which implements the most common mathematical functions.

[4]:
```
import math # make available the math module
dir(math) # list its content
```

```
[4]: ['__doc__',
      '__file__',
      '__loader__',
      '__name__',
      '__package__',
      '__spec__',
      'acos',
      'acosh',
      'asin',
      'asinh',
      'atan',
      'atan2',
      'atanh',
      'ceil',
      'copysign',
      'cos',
      'cosh',
      'degrees',
      'e',
      'erf',
      'erfc',
      'exp',
      'expm1',
      'fabs',
      'factorial',
      'floor',
      'fmod',
      'frexp',
      'fsum',
      'gamma',
      'gcd',
      'hypot',
      'inf',
      'isclose',
      'isfinite',
      'isinf',
      'isnan',
      'ldexp',
      'lgamma',
      'log',
      'log10',
      'log1p',
      'log2',
      'modf',
      'nan',
      'pi',
      'pow',
```

```
 'radians',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'tau',
 'trunc']
```

[5]:
```
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.

    acosh(...)
        acosh(x)

        Return the inverse hyperbolic cosine of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.

    asinh(...)
        asinh(x)
```

```
    Return the inverse hyperbolic sine of x.

atan(...)
    atan(x)

    Return the arc tangent (measured in radians) of x.

atan2(...)
    atan2(y, x)

    Return the arc tangent (measured in radians) of y/x.
    Unlike atan(y/x), the signs of both x and y are considered.

atanh(...)
    atanh(x)

    Return the inverse hyperbolic tangent of x.

ceil(...)
    ceil(x)

    Return the ceiling of x as an Integral.
    This is the smallest integer >= x.

copysign(...)
    copysign(x, y)

    Return a float with the magnitude (absolute value) of x but the sign
    of y. On platforms that support signed zeros, copysign(1.0, -0.0)
    returns -1.0.

cos(...)
    cos(x)

    Return the cosine of x (measured in radians).

cosh(...)
    cosh(x)

    Return the hyperbolic cosine of x.

degrees(...)
    degrees(x)

    Convert angle x from radians to degrees.

erf(...)
```

```
        erf(x)

        Error function at x.

    erfc(...)
        erfc(x)

        Complementary error function at x.

    exp(...)
        exp(x)

        Return e raised to the power of x.

    expm1(...)
        expm1(x)

        Return exp(x)-1.
        This function avoids the loss of precision involved in the direct
evaluation of exp(x)-1 for small x.

    fabs(...)
        fabs(x)

        Return the absolute value of the float x.

    factorial(...)
        factorial(x) -> Integral

        Find x!. Raise a ValueError if x is negative or non-integral.

    floor(...)
        floor(x)

        Return the floor of x as an Integral.
        This is the largest integer <= x.

    fmod(...)
        fmod(x, y)

        Return fmod(x, y), according to platform C.  x % y may differ.

    frexp(...)
        frexp(x)

        Return the mantissa and exponent of x, as pair (m, e).
        m is a float and e is an int, such that x = m * 2.**e.
        If x is 0, m and e are both 0.  Else 0.5 <= abs(m) < 1.0.
```

```
    fsum(...)
        fsum(iterable)

        Return an accurate floating point sum of values in the iterable.
        Assumes IEEE-754 floating point arithmetic.

    gamma(...)
        gamma(x)

        Gamma function at x.

    gcd(...)
        gcd(x, y) -> int
        greatest common divisor of x and y

    hypot(...)
        hypot(x, y)

        Return the Euclidean distance, sqrt(x*x + y*y).

    isclose(...)
        isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0) -> bool

        Determine whether two floating point numbers are close in value.

           rel_tol
               maximum difference for being considered "close", relative to the
               magnitude of the input values
            abs_tol
               maximum difference for being considered "close", regardless of
the
               magnitude of the input values

        Return True if a is close in value to b, and False otherwise.

        For the values to be considered close, the difference between them
        must be smaller than at least one of the tolerances.

        -inf, inf and NaN behave similarly to the IEEE 754 Standard.  That
        is, NaN is not close to anything, even itself.  inf and -inf are
        only close to themselves.

    isfinite(...)
        isfinite(x) -> bool

        Return True if x is neither an infinity nor a NaN, and False otherwise.
```

```
    isinf(...)
        isinf(x) -> bool

        Return True if x is a positive or negative infinity, and False
otherwise.

    isnan(...)
        isnan(x) -> bool

        Return True if x is a NaN (not a number), and False otherwise.

    ldexp(...)
        ldexp(x, i)

        Return x * (2**i).

    lgamma(...)
        lgamma(x)

        Natural logarithm of absolute value of Gamma function at x.

    log(...)
        log(x[, base])

        Return the logarithm of x to the given base.
        If the base not specified, returns the natural logarithm (base e) of x.

    log10(...)
        log10(x)

        Return the base 10 logarithm of x.

    log1p(...)
        log1p(x)

        Return the natural logarithm of 1+x (base e).
        The result is computed in a way which is accurate for x near zero.

    log2(...)
        log2(x)

        Return the base 2 logarithm of x.

    modf(...)
        modf(x)

        Return the fractional and integer parts of x.  Both results carry the
sign
```

```
        of x and are floats.

    pow(...)
        pow(x, y)

        Return x**y (x to the power of y).

    radians(...)
        radians(x)

        Convert angle x from degrees to radians.

    sin(...)
        sin(x)

        Return the sine of x (measured in radians).

    sinh(...)
        sinh(x)

        Return the hyperbolic sine of x.

    sqrt(...)
        sqrt(x)

        Return the square root of x.

    tan(...)
        tan(x)

        Return the tangent of x (measured in radians).

    tanh(...)
        tanh(x)

        Return the hyperbolic tangent of x.

    trunc(...)
        trunc(x:Real) -> Integral

        Truncates x to the nearest Integral toward 0. Uses the __trunc__ magic
method.

DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
```

```
    tau = 6.283185307179586

FILE
    /home/sani/anaconda3/envs/.python3/lib/python3.6/lib-
dynload/math.cpython-36m-x86_64-linux-gnu.so
```

[19]:
```
math.log(3) # accessing the logarithm function
```

[19]:
```
1.0986122886681098
```

[20]:
```
math.exp(3) # accessing the exponential function
```

[20]:
```
20.085536923187668
```

[21]:
```
print (type(math.log)) # yet another type: builtin function
print (type(math.log(3)))
```

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

Since we are lazy and we don't want 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:

[7]:
```
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 about 11000 Euro when investing 10000 Euro for 2 years:

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

[11]:
```
rate = (1/2)*log(11000/10000)
print (rate)
```

```
0.04765508990216247
```

1.6 Boolean expressions

The expressions we have seen so far evaluate to a number. 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)... Let's see some example. The following expression answer the question is 1 equal to 2:

[24]:
```
1 == 2
# single = assigns a value to a variable like in x = 9
# double == checks the equality of two objects
```

[24]: False

[25]:
```
1 != 2 # != is the "not equal to" operator
```

[25]: True

[26]:
```
2 < 2
```

[26]: False

[27]:
```
2 <= 2  # in this case we allow the numbers to be equal too
```

[27]: True

[28]:
```
print (x)
15 <= x and x <= 20
```

```
11
```

[28]: False

[29]:
```
15 <= x or x <= 20
```

[29]: True

[30]:
```
not (x > 20) # the not keyword negates the following expression
```

[30]: True

1.7 String expressions

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

[14]:
```
mystring = "some text with punctuation, spaces and digits 10"
```

[16]:
```
mystring.replace("s", "z")
```

[16]: 'zome text with punctuation, zpacez and digitz 10'

[15]:
```
"abc" + "def"
```

[15]:
```
'abcdef'
```

[33]:
```
"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, `python` will try then to convert it to the desired type. In this case we can *force* the number four to be represented as a string with the `str()` function:

[34]:
```
"The number " + str(4) + " is my favourite number"
```

[34]:
```
'The number 4 is my favourite number'
```

[35]:
```
print (type(3.4))
print (type(str(3.4)))
```

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

Type casting is not always possible though: 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). Here we use the function `int()` to try to convert a string to an integer.

[17]:
```
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'
```

[19]:
```
int("4")
```

[19]:
```
4
```

In order to get prettier strings than just concatenating with +, python allows to format text using the following syntax (which for example allows for float rounding):

[20]:
```
"The speed of light is about {:.1f} {}".format(299792.458, "km/s")
# each {} is mapped to the variables listed later in the "format"
```

[20]:
```
'The speed of light is about 299792.5 km/s'
```

1.8 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 the `if/then` statements. Such statements 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:

[21]:
```
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
```

[38]:
```
# if by mistake I forget to indent some block I get an error
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
```

As an example, in C++ the previous code would have been:

```
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.

1.9 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.

1.9.1 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 function `range` returns the list of integers between the specified limits, if the first limit is not specified 0 is assumed):

[41]:
```
for i in range(25, 30):
    print (i)
```

```
25
26
27
28
29
```

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

[42]:
```
for i in range (30, 25, -1):
    print (i)
```

```
30
29
```

```
28
27
26
```

If we want to skip values in the loop we have to use the `continue` keyword, below 5 is actually missing from the list in the printout:

[43]:
```
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:

[44]:
```
for i in (4, 6, 10, 20):
    print (i)
```

```
4
6
10
20
```

Looping on a string actually means to loop on each single character:

[45]:
```
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
```

1.9.2 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, we first set `x=1` and at each iteration we increment it by 1 untill the condition is `True`, (indeed 8 squared is 64 which is greater than 50):

[46]:
```
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 condition is simply `True` so the code would run forever unless we set an exit strategy.

[47]:
```
x = 1
while True:
    if (x ** 2 > 50):
        break
    print (x)
    x += 1
```

```
1
2
3
4
5
```

```
6
7
```

1.10 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 (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 the list. Ordered instead means that items are kept in the same order they have been added to the list.

[22]:
```
mylist = [21, 32, 15]
print(mylist)
print (type(mylist))
```

```
[21, 32, 15]
```

[24]:
```
mylist[0]
```

[24]:
```
21
```

The number of elements in a list are counted using `len()`:

[25]:
```
len(mylist)
```

[25]:
```
3
```

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

[26]:
```
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, 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:

[39]:
```
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:

[27]:
```
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 of list, with `insert` an item can be added in a desired position of the list:

[34]:
```
mylist.append(188) # append add an item at the end of the list
mylist
```

[34]:
```
[21, 74, 15, 188]
```

[35]:
```
mylist.insert(2, 85) # insert an item in the desired position
                     # (2 in this example)
mylist
```

[35]:
```
[21, 74, 85, 15, 188]
```

Accessing items outside the list range gives an error:

[36]:
```
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.

[38]:
```
print ("negative index -1 returns the last element:", mylist[-1])
print ("slicing [1:3] returns the elements between the 1st and 2nd:", mylist[0:
↪3])
print ("slicing [:2] returns the elements between the 1st and 2nd:", mylist[:2])
print ("slicing [2:] returns the elements between the 2nd and the last:",␣
↪mylist[2:])
```

```
negative index -1 returns the last element: 188
slicing [1:3] returns the elements between the 1st and 2nd: [21, 74, 85]
slicing [:2] returns the elements between the 1st and 2nd: [21, 74]
slicing [2:] returns the elements 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).

[71]:
```
mixedlist = [1, 2, "b", math.sqrt]
print (mixedlist)
```

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

[72]:
```
print (mixedlist[0])
print (mixedlist['k'])
```

```
1
```

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

TypeError                                 Traceback (most recent call last)

<ipython-input-72-aea4c7f9789e> in <module>()
      1 print (mixedlist[0])
----> 2 print (mixedlist['k'])


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

1.11 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 objects 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 example of the list 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.

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. We can check if a key exists with the `in` operator.

[40]:
```
adict = {"apple": 4, "banana": 5}
print (adict["apple"])
```

```
4
```

[41]:
```
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'
```

[75]:
```
"pear" in adict # indeed
```

[75]:
```
False
```

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

[43]:
```
adict["banana"] = 2
adict["pear"] = 10
print (len(adict))
print (adict)
```

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

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

[44]:
```
adict[math.log] = math.exp
```

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

[45]:
```
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.

[46]:
```
del adict[math.log]
seconddict = {"watermelon": 0, "strawberry": 1}
adict.update(seconddict)
```

```
print (adict)
```

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

1.12 Exercises

1.12.1 Exercise 1.1

- What is the built-in function that `python` uses to iterate over a number sequence ? Wrtie an example that uses it.
- What is a string in `python` ? Declare one string variable and try to manipulate it.
- What does the continue do in `python` ? Show an example of its usage.
- When should you use the break in `python` ? Show an example of its usage.
- What is a dictionary in `python` programming ? Create a dictionary, modify it and then print all its items.
- Which `python` function will you use to convert a number to a string ? Show an example.

2 Advanced hints

If you would like to know much **more** about python during the course without the EDX site you can look these videos during the course in your spare time:

Some more information about different kind of languages: https://www.youtube.com/watch?v=9oYFH4OmYDY

Basic data types: https://www.youtube.com/watch?v=XIjrEt2lz1U

Variables: https://www.youtube.com/watch?v=z2NLjdfxEyQ

Branching: https://www.youtube.com/watch?v=8vr3nyg5QcM

Tuples: https://www.youtube.com/watch?v=CwZyWaap5Z8

Lists: https://www.youtube.com/watch?v=eMyWO0tcxKg

List Operations: https://www.youtube.com/watch?v=rQBho4-bI3o

Mutation, Aliasing, Cloning: https://www.youtube.com/watch?v=2SRXg8Or-Pc

Functions as Objects: https://www.youtube.com/watch?v=pheM3rVmGMU

Dictionaries: https://www.youtube.com/watch?v=elSt5hke-Rs