Python crash course

Chapter resources

Open in Colab | Practice quiz

Note for those who are already familiar with Python: Feel free to skim through the content and zoom into only those parts which you find new and interesting.

Python is a high-level, interpreted programming language designed for "general purpose" programming. Python supports a dynamic type system and various programming paradigms, including object-oriented programming, imperative programming, functional programming, and procedural programming. The language was created in 1991 by Guido van Rossum and its name is inspired by the satirical TV series Monty Python's Flying Circus (https://en.wikipedia.org/wiki/Monty_Python%27s_Flying_Circus).

Although Python was not originally conceived as a programming language for scientific computing, its extreme versatility has contributed to the emergence of a series of libraries that make numerical computation in Python convenient and efficient. A good portion of these libraries are part of "SciPy" (https://www.scipy.org/), an open-source software ecosystem for scientific computing. In this laboratory, in addition to an introduction to Python, we will specifically cover the fundamentals of NumPy (scientific computing) and Matplotlib (2D Plotting).

Important references to consult during the course, only the following documentation:

- Python 3: https://docs.python.org/3/index.html;
- Numpy: http://www.numpy.org/;
- Matplotlib: https://matplotlib.org/.

Numbers

The numeric data types in Python are int, float, and complex. We will focus on int and float. Some examples of operations between numbers:

```
3 + 5 #sum
```

```
2 - 8 #difference
```

```
-6
```

```
3 * 5 #product
```

```
15
```

```
3 / 2 #division, note that in Python 3, integer division returns a float
```

```
1.5
```

```
3 // 2 #integer division
```

```
1
```

```
9 ** 2 #exponentiation
```

```
81
```

```
4 % 2 #modulus
```

```
0
```

```
(1 + 4) * (3 - 2) # Use of parentheses
```

```
5
```

Variables and Types

In general, numbers without a decimal point are interpreted as int, while those with a decimal point as float. Since Python is dynamically typed, we do not need to explicitly declare the type of a variable. The type will be associated with the variable as soon as we assign a value to it. We can check the type of a variable using the `type` function:

```
x = 3
y = 9
z = 1.5
h = x/y
l = x//y
type(x), type(y), type(z), type(h), type(l)
```

```
(int, int, float, float, int)
```

It is possible to cast from one type to another using the `int` and `float` functions:

```
int(2.5)
```

```
2
```

```
float(3)
```

```
3.0
```

As in C, the “in place” operations between variables are defined:

```
x = 8
y = 12
x+=y #equivalent to x=x+y
x
```

```
20
```

The notations “++” and “–” are not defined.

```
a++ #error!
```

To increment by one unit, the `+=1` notation must be used:

```
a=1
a+=1
a
```

```
2
```

> **Question 1**
>
> What is the type of the following variable?
>
> ```
> x = (3//2*2)**2+(3*0.0)
> ```

Booleans

Booleans are represented by the keywords `True` and `False` (both start with a capital letter).

```
print(True)
print(False)
```

```
True
False
```

```
type(True)
```

```
bool
```

It is possible to generate booleans using comparison operators:

```
5==5
```

```
True
```

```
7==5
```

```
False
```

```
5>4
```

```
True
```

```
9>10
```

```
False
```

```
9<=10
```

```
True
```

```
11<=10
```

```
False
```

The logical operators are `and` and `or`:`

```
print(5==5 and 3<5)
print(3>5 or 3<5)
print(3>9 or 3<2)
```

```
True
True
False
```

It is possible to perform a check on types using `type` and `==`:

```
type(2.5)==float
```

```
True
```

```
type(2)==float
```

```
False
```

Alternatively, it is possible to use the `isinstance` function:

```
isinstance(2.5,float)
```

```
True
```

```
isinstance(2.5,int)
```

```
False
```

The `isinstance` function is particularly convenient when you want to check that a variable belongs to one of a set of types. For example, if we want to check that a variable contains a number:

```
isinstance(2.5,(float,int))
```

```
True
```

```
isinstance(5,(float,int))
```

```
True
```

Print

Printing is done using the `print` function:

```
var = 2.2
print(var)
```

```
2.2
```

We can print an empty line by omitting the print parameter:

```
print(2)
print()
print(3)
```

```
2

3
```

Alternatively, we can specify to insert two “newlines” at the end of the printout by specifying the parameter `end="\n\n"` (“” is a string - we will delve into strings later):

```
print(2, end="\n\n")
print(3)
```

```
2

3
```

The same method can be used to omit the insertion of spaces between two consecutive prints:

```
print(2, end="") #"" represents an empty string
print(3)
```

```
23
```

We can print multiple elements consecutively by separating the `print` arguments with commas. Furthermore, the `print` function allows printing numbers as well as strings:

```
print(1,8/2,7,14%6,True)
```

```
1 4.0 7 2 True
```

Lists

Lists are a sequential data structure that can be used to represent sequences of values of any type. Lists can also contain elements of mixed types. A list is defined using square brackets:

```
l = [1,2,3,4,5] #this is a list (square brackets)
print(l)
```

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

Lists can be indexed using square brackets. Indexing starts from 0 as in C:

```
print(l[0],l[2])
l[0]=8 #assignment of a new value to the first memory location
print(l)
```

```
1 3
[8, 2, 3, 4, 5]
```

It is possible to add new values to a list using the `append` function:

```
l = []
print(l)
l.append(1)
l.append(2.5)
l.append(8)
l.append(-12)
print(l)
```

```
[]
[1, 2.5, 8, -12]
```

Lists can be concatenated using the sum operator:

```
l1 = [1,5]
l2 = [4,6]
print(l1+l2)
```

```
[1, 5, 4, 6]
```

The multiplication operator can be used to repeat a list. For example:

```
l1 = [1,3]
print(l1*2) # Concatenates l1 to itself twice
```

```
[1, 3, 1, 3]
```

Using the multiplication operator, it is possible to quickly create lists with an arbitrary number of equal values. For example:

```
print([0]*5) #list of 5 zeros
print([0]*4+[1]*1) #4 zeros followed by 1 one
```

```
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 1]
```

The length of a list can be obtained using the `len` function:

```
print(l2)
print(len(l2))
```

```
[4, 6]
2
```

For lists, an ordering is defined that depends on their length: shorter lists are “less than” longer lists:

```
print([1,2,3]<[1,2,3,4])
print([1,2,3,5]⩾[1,2,3,4])
```

```
True
True
```

The `==` operator does not check if the lengths are equal, but verifies that the content of the two lists is effectively equal:

```
print([1,2,3]==[1,2,3])
print([1,2,3]==[1,3,2])
```

```
True
False
```

It is possible to check that an element belongs to the list using the keyword `in`:

```
print(7 in [1,3,4])
print(3 in [1,3,4])
```

```
False
True
```

The functions `max` and `min` can be used to calculate the maximum and minimum of a list:

```
l=[-5,2,10,6]
print(max(l))
print(min(l))
```

```
10
-5
```

It is possible to remove a value from a list using the `remove` method:

```
l=[1,2,3,4,2]
print(l)
l.remove(2)
print(l)
```

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

However, this method only removes **the first occurrence** of the passed value. If we want to remove a value identified by a specific index, we can use the `del` construct:

```
l=[1,2,3,4,2]
print(l)
del l[4]
print(l)
```

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

Additionally, to access the last element and remove the last element, we can use the `pop` method:

```
l=[1,2,3,4,5]
print(l)
print(l.pop())
print(l)
```

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

Indexing and Slicing

It is possible to extract a sublist from a list by specifying the first index (inclusive) and the last index (exclusive) separated by the `:` symbol. This notation is somewhat reminiscent of C++ string's `substr` method.

```
l = [1,2,3,4,5,6,7,8]
print("List l     →", l)
print("l[0:3]      →", l[0:3]) #from index 0 (inclusive) to index 3 (exclusive)
print("l[1:2]      →", l[1:2]) #from index 1 (inclusive) to index 2 (exclusive)
```

```
List l     → [1, 2, 3, 4, 5, 6, 7, 8]
l[0:3]      → [1, 2, 3]
l[1:2]      → [2]
```

When the first index is omitted, it is automatically replaced with “0”:

```
print("l[:2]       →", l[:2]) # from index 0 (inclusive) to index 2 (exclusive)
# equivalent to the following:
print("l[0:2]      →", l[0:2]) # from index 0 (inclusive) to index 2 (exclusive)
```

```
l[:2]       → [1, 2]
l[0:2]      → [1, 2]
```

Similarly, if we omit the second index, it is replaced with the last index of the list:

```
print("Last index of the list:",len(l))
print("l[3:]       →", l[3:]) #from index 3 (inclusive) to index 5 (exclusive)
#equivalent to the following:
print("l[3:8]      →", l[3:8]) #from index 3 (inclusive) to index 5 (exclusive)
```

```
Last index of the list: 8
l[3:]       → [4, 5, 6, 7, 8]
l[3:8]      → [4, 5, 6, 7, 8]
```

Omitting both indices:

```
print("l[:]         →", l[:]) #from index 0 (inclusive) to index 5 (exclusive)
#equivalent to:
print("l[0:8]       →", l[0:8]) #from index 0 (inclusive) to index 5 (exclusive)
```

```
l[:]         → [1, 2, 3, 4, 5, 6, 7, 8]
l[0:8]       → [1, 2, 3, 4, 5, 6, 7, 8]
```

It is also possible to specify the “step”, as a third number separated by another symbol `:`:

```
print("l[0:8:2]    →", l[0:8:2])
# from 0 (inclusive) to 8 (exclusive) with a step of 2 (every other element)
# equivalent to:
print("l[::2]      →", l[::2])
# from 0 (inclusive) to 8 (exclusive) with a step of 2 (every other element)
```

```
l[0:8:2]    → [1, 3, 5, 7]
l[::2]      → [1, 3, 5, 7]
```

To reverse the order of elements, it is also possible to specify a negative step. In this case, you must ensure that the first index is greater than the second:

```
print("l[5:2:-1]    →", l[5:2:-1])
# from 5 (inclusive) to 2 (exclusive) with a step of -1
print("l[2:5:-1]    →", l[2:5:-1])
# in this case, the first index is smaller than the second,
# so the result will be an empty list
```

```
l[5:2:-1]    → [6, 5, 4]
l[2:5:-1]    → []
```

In this case too, by omitting indices, these will be replaced with the most obvious choices. In the case of omission, however, the conditions for inclusion and exclusion of indices change. Let's see some examples:

```
print("l[:2:-1]     →", l[:2:-1])
#from the last index (inclusive) to 2 (exclusive) with a step of -1
#equivalent to:
print("l[8:2:-1]    →", l[8:2:-1])

print()
print("l[3::-1]     →", l[3::-1])
#from the third index (inclusive) to 0 (inclusive, as omitted) with a step of -1
#similar, but not equivalent to:
print("l[3:0:-1]    →", l[3:0:-1])
#from the third index (inclusive) to 0 (exclusive) with a step of -1

print()
print("l[::-1]      →", l[::-1])
#from the last index (inclusive) to the first (inclusive, as omitted) with a step of -1
#similar, but not equivalent to:
print("l[8:0:-1]    →", l[8:0:-1])
#from the last index (inclusive) to the first (exclusive) with a step of -1
```

```
l[:2:-1]     → [8, 7, 6, 5, 4]
l[8:2:-1]    → [8, 7, 6, 5, 4]

l[3::-1]     → [4, 3, 2, 1]
l[3:0:-1]    → [4, 3, 2]

l[::-1]      → [8, 7, 6, 5, 4, 3, 2, 1]
l[8:0:-1]    → [8, 7, 6, 5, 4, 3, 2]
```

The notation `::-1`, in particular, is useful for reversing lists:

```
print(l)
print(l[::-1])
```

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

Indexing and slicing can also be used to assign values to list elements. For example:

```
l = [5,7,9,-1,2,6,5,4,-6]
print(l)
l[3]=80
print(l)
```

```
[5, 7, 9, -1, 2, 6, 5, 4, -6]
[5, 7, 9, 80, 2, 6, 5, 4, -6]
```

It's also possible to assign more than one element at a time:

```
l[::2]=[0,0,0,0,0] # assign 0 to odd-positioned numbers
print(l)
```

```
[0, 7, 0, 80, 0, 6, 0, 4, 0]
```

The lists can also be nested:

```
a1 = [1,2,[4,8,[7,5]],[9],2]
print(a1)
```

```
[1, 2, [4, 8, [7, 5]], [9], 2]
```

The indexing of these nested structures occurs by concatenating the indices as follows:

```
print(a1[2][2][0]) #the first index selects the list [4,8,...]
#the second index selects the list [7,5]
#the third index selects the element 7
```

```
7
```

Question 2

Extract the list `[3, 1.2]` from the following list:

```
l = [1, 4, 5, [7, 9, -1, [0, 3, 2, 1.2], 8, []]]
```

Tuples

Tuples are similar to lists, but they are immutable. That is, they cannot be modified after their initialization. Unlike lists, tuples are defined using parentheses:

```
l = [1,2,3,4,5] # this is a list (square brackets)
t = (1,2,3,4,5) # this is a tuple (round brackets)
print(l)
print(t)
```

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

The indexing and slicing rules seen for lists also apply to tuples. However, as mentioned before, tuples cannot be modified:

```
t = (1,3,5)
t[0]=8 #returns an error because tuples are immutable
```

Initializing a tuple with a single element will produce a number. This is because parentheses are also used to group the different terms of an operation:

```
t=(1)
print(t)
```

```
1
```

To define a one-dimensional tuple, we must explicitly add a comma after the first element:

```
t=(1,)
print(t)
```

```
(1,)
```

It is also possible to omit the parentheses in the definition of tuples:

```
t1=1,3,5
t2=1,
print(t1,t2)
```

```
(1, 3, 5) (1,)
```

It is possible to convert tuples into lists and vice versa:

```
l=[1,2,3,4,5,6,7,8]
t=(4,5,6,7,4,8,2,4)
ttl = list(t)
ltt = tuple(l)

print(ttl)
print(ltt)
```

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

Tuples can also be created and “unpacked” on the fly:

```
t1=1,2,3

print(t1)

a,b,c=t1 # Unpacking the tuple
print(a,b,c)
```

```
(1, 2, 3)
1 2 3
```

This system allows to swap two variables in a single line of code:

```
var1 = "Var 1"
var2 = "Var 2"

print(var1,var2)

var1,var2=var2,var1
print(var1,var2)

#equivalent to:
var1 = "Var 1"
var2 = "Var 2"
t = (var2,var1)
var1=t[0]
var2=t[1]
print(var1,var2)
```

```
Var 1 Var 2
Var 2 Var 1
Var 2 Var 1
```

Nested tuples can be unpacked as follows:

```
t = (1,(2,3),(4,5,6))
x,(t11,t12),(t21,t22,t23) = t
print(t)
print(x, t11, t12, t21, t22, t23)
#The notation a,b,c,d,e,f = t would raise an error
```

```
(1, (2, 3), (4, 5, 6))
1 2 3 4 5 6
```

> **Question 3**
>
> What is the main difference between tuples and lists? Provide an example of a case where a tuple is a more appropriate type than a list.

Strings

In Python we can define strings in three ways:

```
s1 = 'Single quotes'
s2 = "Double quotes, can also contain single quotes '' "
s3 = """Triple
double quotes
can be defined on multiple lines"""
type(s1), type(s2), type(s3)
```

```
(str, str, str)
```

Printing takes place using the “print” function:

```
print(s1)
print(s2)
print(s3)
```

```
Single quotes
Double quotes, can also contain single quotes ''
Triple
double quotes
can be defined on multiple lines
```

Strings also have a set of predefined methods:

```
print("hello".upper()) #make everything uppercase
print("HELLO".lower()) #all lowercase
print("hello how are you".capitalize()) #first letter uppercase
print("hello how are you".split()) #splits a string and returns a list
print("hello, how are you".split(','))# splits when it finds the comma
print("-".join(["one","two","three"])) #builds a string by concatenating the elements
#of the list and separating them by the delimiter
```

```
HELLO
hello
Hello how are you
['hello', 'how', 'are', 'you']
['hello', ' how are you']
one-two-three
```

Strings can be indexed similarly to arrays to obtain substrings:

```
s = "Hello World"
print(s[:4]) #first 4 characters
print(s[4:]) #from the fourth character to the end
print(s[4:7]) #from the fourth to the sixth character
print(s[::-1]) #string reversal
```

```
Hell
o World
o W
dlroW olleH
```

The `split` method in particular can be used for tokenization or to easily extract substrings. For example, let's say we want to extract the number `2017` from the string `A-2017-B2`:

```
print("A-2017-B2".split('-')[1])
```

```
2017
```

The `==` operator checks that two strings are equal:

```
print("hello"=="hello")
print("hello"=="hello2")
```

```
True
False
```

The other operators reflect the lexicographical order between strings:

```
print("abc"<"def")
print("Abc">"def")
```

```
True
False
```

> **Question 4**
>
> Which code allows manipulating the string `azyp-kk9-382` to obtain the string `Kk9`?

String Formatting

We can build formatted strings following a syntax similar to printf:

```
# To build the formatted string, I follow the string with the "%" symbol and then insert
# a tuple containing the arguments
s1 = "This %s is formatted. I can insert numbers, for example %0.2f" %
        ("string",3.00002)
print(s1)
```

```
This string is formatted. I can insert numbers, for example 3.00
```

An alternative and more recent way to format strings is to use the “format” method:

```
s2 = "This {} is formatted. I can insert numbers, for example {}"\
    .format("string",3.000002)# the "\" character allows to break the line
print(s2)
```

```
This string is formatted. I can insert numbers, for example 3.000002
```

It is possible to specify the type of each argument using the colon:

```
print("This {:s} is formatted. I can insert numbers, for example {:0.2f}"\
      .format("string",3.00002)) #Positional parameters, without specifying indices
```

```
This string is formatted. I can insert numbers, for example 3.00
```

It’s also possible to assign names to arguments so you can call them out of order:

```
print("This {str:s} is formatted. I can insert numbers, for example {num:0.2f}"\
    .format(num=3.00002, str="string"))
```

```
This string is formatted. I can insert numbers, for example 3.00
```

> **Question 5**
>
> Given the variables:
>
> `python a = "hello" b = "world" c = 2.0`
>
> Use string formatting to print the string `hello 2 times world`.

Dictionaries

Dictionaries are similar to lists, but they are indexed by “hashable” objects, for example strings:

```
d = {"val1":1, "val2":2}
print(d)

print(d["val1"])
```

```
{'val1': 1, 'val2': 2}
1
```

It is possible to obtain the list of keys and values as follows:

```
print(d.keys()) #keys and values are in random order
print(d.values())
```

```
dict_keys(['val1', 'val2'])
dict_values([1, 2])
```

It is possible to index dictionaries with tuples (which are “hashable”)

```
d = {(2,3):5, (4,6):11}

print(d[(2,3)])
```

```
5
```

Dictionaries can also be extended dynamically:

```
d = dict() #empty dictionary
d["key"]="value"
print(d)
```

```
{'key': 'value'}
```

We can check that an element is among the keys of a dictionary as follows:

```
d = {1:'hello', '5': 5, 8: -1}
print(5 in d)
print('5' in d)
```

```
False
True
```

It is possible to check that an element is among the values of a dictionary as follows:

```
print(-1 in d.values())
```

```
True
```

Set

Sets are data structures that can contain only one instance of a given element:

```
s = {1,2,3,3}
print(s) # only one "3" can be contained
```

```
{1, 2, 3}
```

We can add an element to a set using the “add” method:

```
print(s)
s.add(5)
print(s)
s.add(1) # has no effect. 1 is already present
print(s)
```

```
{1, 2, 3}
{1, 2, 3, 5}
{1, 2, 3, 5}
```

Also in this case, we can check whether an element belongs to a set using the keyword `in` :

```
s={1,5,-1}
print(-1 in s)
print(8 in s)
```

```
True
False
```

It is also possible to create sets from lists:

```
set([1,3,3,2,5,1])
```

```
{1, 2, 3, 5}
```

If/elif/else constructs

Conditional constructs work similarly to C-based languages. Unlike those languages, however, Python replaces parentheses with mandatory indentation. The following C++ code:

```
int var1 = 5;
int var2 = 10;
if(var1<var2) {
    int var3 = var1+var2;
    cout << "Hello World "<<var3;
}
cout << "End";
```

is instead written as follows:

```
var1 = 5
var2 = 10
if var1<var2:
    var3 = var1+var2
    print("Hello World",var3)
print("End")
```

```
Hello World 15
End
```

In practice:

- the condition to be checked is not enclosed in parentheses;
- the colon indicates the beginning of the if body;
- indentation determines what belongs to the if body and what does not belong to the if body.

Since indentation has syntactic value, it becomes mandatory. Furthermore, it is not possible to indent parts of code where it is not significant. For example, the following code returns an error:

```
print("Hello")
    print("World") #indentation error
```

The indentation rules just seen also apply to loops and other constructs where it is necessary to delimit blocks of code. The if construct also allows specifying an `else` branch and an `elif` branch for cascaded controls. Let's look at some examples:

```
true_condition = True
false_condition = False

if true_condition: #the colon ":" is mandatory
    word="cool!"
    print(word) #indentation is mandatory

if false_condition:
    print("not cool :(")

if not false_condition: #we negate the condition with "not"
    word="cool"
    print(word,"again :)")

if false_condition:
    word="this"
    print(word+" is "+"false")
else: #colon + indentation
    print("true")

if false_condition:
    print("false")
elif 5>4: #implements an "else if"
    print("5>4")
else:
    print("4<5??")
```

```
cool!
cool again :)
true
5>4
```

It is also possible to check if a value belongs to a list. This is very useful for checking if a parameter is allowed.

```
allowed_parameters = ["single", "double", 5, -2]

parameter = "double"

if parameter in allowed_parameters:
    print(parameter,"is ok")
else:
    print(parameter,"not in",allowed_parameters)

x=8
if x in allowed_parameters:
    print(x,"is not ok")
else:
    print(x,"not in",allowed_parameters)
```

```
double is ok
8 not in ['single', 'double', 5, -2]
```

An “inline” variant of the if construct can be used for conditional assignments:

```
var1 = 5
var2 = 3
m = var1 if var1>var2 else var2 # Calculate the maximum
print(m)
```

```
5
```

In Python there is no switch construct; to implement it, a series of cascaded elif statements is used:

```
s = "hello"
if s=="help":
    print("help")
elif s=="world":
    print("hello",s)
elif s=="hello":
    print(s,"world")
else:
    print("Default")
```

```
hello world
```

Question 6

Consider the following code:

```
x=2
if x>0:
    y=12
    x=x+y
    print y
else:
        z=28
        h=12
        print z+h
```

Highlight any syntactic errors. Rewrite the code in C++ or Java and compare the two versions of the code.

While and For Loops

While loops are defined as follows:

```
i=0
while i<5: # Colon ":"
    print(i) # Indentation
    i+=1
```

```
0
1
2
3
4
```

The syntax of for loops is a bit different from the standard C syntax. For loops in Python are more like **foreach** and require an "iterable" (for example, a list or a tuple) to be executed:

```
l=[1,7,2,5]
for v in l:
    print(v)
```

```
1
7
2
5
```

To write something equivalent to the following C code:

```
for (int i=0; i<5; i++) {...}
```

we can use the **range** function which generates sequential numbers on the fly:

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Range does not directly generate a list, but a "range-type" "generator", i.e., an object capable of generating numbers. If we want to convert it into a list, we must do so explicitly:

```
print(range(5)) #range type object, generates numbers from 0 to 5 (exclusive)
print(list(range(5))) #range just generates consecutive numbers
#converting range to a list, we can verify which numbers are generated
```

```
range(0, 5)
[0, 1, 2, 3, 4]
```

If we wanted to simultaneously iterate through indices and values of an array, we could write:

```
array=[1,7,2,4,5]
for i in range(len(array)):
    print(i,"→",array[i])
```

```
0 → 1
1 → 7
2 → 2
3 → 4
4 → 5
```

In Python, however, it is possible to use the `enumerate` function to obtain the same result in a more compact way:

```
for index,value in enumerate(array): #get both index and value
    print(index,"→",value)
```

```
0 → 1
1 → 7
2 → 2
3 → 4
4 → 5
```

Now let's assume we want to simultaneously iterate through all the i-th elements of multiple lists. For example:

```
a1=[1,6,2,5]
a2=[1,8,2,7]
a3=[9,2,5,2]

for i in range(len(a1)):
    print(a1[i],a2[i],a3[i])
```

```
1 1 9
6 8 2
2 2 5
5 7 2
```

A very useful function when working with loops is `zip`, which allows to group the corresponding elements from different lists:

```
l1 = [1,6,5,2]
l2 = [3,8,9,2]
zipped=list(zip(l1,l2))
print(zipped)
```

```
[(1, 3), (6, 8), (5, 9), (2, 2)]
```

In practice, `zip` groups the i-th elements of the lists into tuples. The i-th tuple of `zipped` contains the i-th elements of the two lists. By combining `zip` with a for loop, we can obtain the following result:

```
for v1,v2,v3 in zip(a1,a2,a3):
    print(v1,v2,v3)
```

```
1 1 9
6 8 2
2 2 5
5 7 2
```

which is equivalent to the code seen previously. It is also possible to combine zip and enumerate as follows:

```
for i,(v1,v2,v3) in enumerate(zip(a1,a2,a3)):
    print(i,"→",v1,v2,v3)
```

```
0 → 1 1 9
1 → 6 8 2
2 → 2 2 5
3 → 5 7 2
```

Attention, also `zip`, like `range`, produces a generator. Therefore it is necessary to convert it explicitly into a list to print its values:

```
print(zip(l1,l2))
print(list(zip(l1,l2)))
```

```
<zip object at 0x105674cc0>
[(1, 3), (6, 8), (5, 9), (2, 2)]
```

Understanding Lists and Dictionaries

List comprehension is a syntactic tool that allows you to define lists on the fly from other lists, in an iterative manner. For example, it is possible to multiply all elements of a list by a value, as follows:

```
a = list(range(8))

b = [x*3 for x in a] # List "a" is iterated. Variable "x" will contain the values of "a"
        in each iteration.

print(a)
print(b)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 3, 6, 9, 12, 15, 18, 21]
```

It's also possible to include only some elements selectively:

```
print([x for x in a if x%2==0]) # Includes only the even numbers
```

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

```
a = [1,3,8,2,9]
b = [4,9,2,1,4]

c = [x+y for x,y in zip(a,b)]

print(a)
print(b)
print(c) # its elements are the sums of the elements from a and b
```

```
[1, 3, 8, 2, 9]
[4, 9, 2, 1, 4]
[5, 12, 10, 3, 13]
```

The mechanisms of understanding can also be used in the case of dictionaries:

```
a = ["one","two","three","four","five","six","seven"]
b = range(1,8)
d = {i:s for i,s in zip(a,b)}
print(d)
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5, 'six': 6, 'seven': 7}
```

> **Question 7**
>
> Can all operations that can be performed using list and dictionary comprehensions be performed using a for loop? What are the main advantages of these techniques compared to using for loops?

Definition of Functions

Given what has already been said about indentation, the definition of a function is very natural:

```
def fun(x, y):
    return x**y

print(fun(3,2)) # the default value "2" is used
```

```
9
```

Furthermore, similarly to what happens with the C language, it is possible to define default values for parameters:

```
def fun(x, y=2):
    return x**y

print(fun(3)) #the default value "2" is used
```

```
9
```

The parameters of a function can be specified in a different order than the one in which they were defined by calling them by name:

```
print(fun(y=3,x=2))
```

```
8
```

It is possible to define a function that returns more than one element using tuples:

```
def soMuchFun(x,y):
    return x**y, y**x

print(soMuchFun(2,3))

a,b=soMuchFun(2,3) # I can "unpack" the returned tuple
print(a,b)
```

```
(8, 9)
8 9
```

It is also possible to define anonymous functions as follows:

```
myfun = lambda x: x**2 #one input and one output
print(myfun(2))

myfun1 = lambda x,y: x+y #two inputs and one output
print(myfun1(2,3))

myfun2 = lambda x,y: (x**2,y**2) #two inputs and two outputs
print(myfun2(2,3))
```

```
4
5
(4, 9)
```

> **Question 8**
>
> What is the advantage of defining a function using **lambda**? What are its limitations?

Map and Filter

The functions **map** and **filter** allow operations to be performed on the elements of a list. In particular, **map** applies a function to all elements of a list:

```
def pow2(x):
    return(x**2)

l1 = list(range(6))
l2 = list(map(pow2,l1)) # Applies pow2 to all elements of the list
print(l1)
print(l2)
```

```
[0, 1, 2, 3, 4, 5]
[0, 1, 4, 9, 16, 25]
```

When using **map** and **filter**, anonymous functions become particularly useful, allowing us to write more compactly. For example, we can rewrite what was seen above as follows:

```
l2 = list(map(lambda x: x**2, l1))
print(l2)
```

```
[0, 1, 4, 9, 16, 25]
```

Filter allows selecting a subset of the elements of a list based on a condition. The condition is specified by passing a function that takes the list element as input and returns a boolean:

```
print(list(filter(lambda x: x%2==0,l1))) # Filter only even numbers
```

```
[0, 2, 4]
```

Object-Oriented Programming - Definition of Classes

Object-oriented programming in Python is intuitive. The main differences compared to the most common object-oriented programming languages are the following:

- visibility modifiers do not exist (**private**, **protected**, and **public**). By convention, all private symbols are preceded by "__";
- every method is a function with a default argument (**self**) that represents the object's state;
- the constructor is called "__init__".

```
class Class(object): # inherit from the standard "object" class
    def __init__(self, x): # constructor
        self.x=x # insert the value x into the object's state

    def print_value(self):
        print(self.x)

    def power(self,y=2):
        self.x =  self.x**y

c = Class(3)
c.print_value()
c.power(3)
c.print_value()
c.power()
c.print_value()
```

```
3
27
729
```

To extend a class, one does as follows:

```
class Class2(Class): # derive the class "Class" and inherit methods and properties
    def __init__(self,x):
        super(Class2, self).__init__(x) # call the constructor of the parent class
    def prt(self): # redefine the prt method
        print("yeah",self.x)

c2 = Class2(2)
c2.power()
c2.prt()
```

```
yeah 4
```

Duck Typing

To identify data types, Python follows the principle of duck typing. According to this principle, types are defined using the *duck test*:

> If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

This means that types are defined in relation to the operations that can be performed on them. Let's look at an example (taken from Wikipedia).

```
class Sparrow(object):
    def fly(self):
        print("Sparrow flying")

class Airplane(object):
    def fly(self):
        print("Airplane flying")

class Whale(object):
    def swim(self):
        print("Whale swimming")

def lift_off(entity):
    entity.fly()

sparrow = Sparrow()
airplane = Airplane()
whale = Whale()

try:
    lift_off(sparrow)
    lift_off(airplane)
    lift_off(whale) #Error, the "type" of this object does not allow executing the "fly"
        method.
                    #According to the duck test, this type is incompatible
except AttributeError as e:
    print("Error:",e)
```

```
Sparrow flying
Airplane flying
Error: 'Whale' object has no attribute 'fly'
```

Exceptions

Similar to many modern languages, Python supports the use of exceptions. It is possible to catch an exception with the `try - except` construct:

```
try:
    5/0
except:
    print("Houston, we have a problem!")
```

```
Houston, we have a problem!
```

In Python, exceptions are typed. We can decide which types of exceptions to catch as follows:

```
try:
    5/0
except ZeroDivisionError:
    print("Houston, we have a problem!")
```

```
Houston, we have a problem!
```

We can access the triggered exception (e.g., to get more information) as follows:

```
try:
    5/0
except ZeroDivisionError as e:
    print("Houston, we have a problem!")
    print(e)
```

```
Houston, we have a problem!
division by zero
```

It is possible to raise an exception using `raise`:

```
def div(x,y):
    if y==0:
        raise ZeroDivisionError() #raises an exception
    else:
        return x/y
div(5,2)
div(5,0)
```

We can define new exceptions by extending the `Exception` class:

```
class MyException(Exception):
    def __init__(self, message):
        self.message = message

raise MyException("Exception!")
```

A fast and convenient way to raise an exception (an AssertionError specifically) when something goes wrong, is to use `assert`, which takes a boolean as input and, optionally, an error message. The boolean should be set to `False` if something went wrong. Let's see an example:

```
def div(x,y):
    assert y≠0, "Cannot divide by zero!"
    return x/y

div(5,2)
div(5,0)
```

Definition of Modules

When building complex programs, it can be useful to group function and class definitions into modules. The simplest way to define a module in Python is to place the definitions inside a dedicated file `modulo.py`. The definitions can then be imported using the `from modulo import funzione` syntax, provided that the file calling the functions and the one defining the module are in the same folder. Let's see an example:

```
#file modulo.py
def mysum(a,b):
    return a+b

def myprod(a,b):
    return a*b
```

```
#file main.py (same folder as modulo.py)
from modulo import mysum, myprod
print(mysum(2,3)) #5
print(myprod(2,3)) #6
```

Further information on more advanced uses of modules and packages can be found here: https://docs.python.org/3/tutorial/modules.html.

```
import numpy as np #The "as" notation allows us to reference the numpy namespace simply
        with np in the future
```

```
l = [[1,2,3],[4,5,2],[1,8,3]] #a list containing three lists
print("List of lists:",l) #it is displayed as defined
a = np.array(l) #Build a numpy array from the list of lists
print("Numpy array:\n",a) #each inner list is identified as a row of a two-dimensional
        matrix
print("Numpy array from tuple:\n",np.array(((1,2,3),(4,5,6)))) #Can also create numpy
        arrays from tuples
```

```
List of lists: [[1, 2, 3], [4, 5, 2], [1, 8, 3]]
Numpy array:
 [[1 2 3]
 [4 5 2]
 [1 8 3]]
Numpy array from tuple:
 [[1 2 3]
 [4 5 6]]
```

Every numpy array has a *shape* property that allows us to determine the number of dimensions of the structure:

```
print(a.shape) # This is a 3 x 3 matrix
```

```
(3, 3)
```

Let's look at a few more examples

```
array = np.array([1,2,3,4])
matrix = np.array([[1,2,3,4],[5,4,2,3],[7,5,3,2],[0,2,3,1]])
tensor = np.array([[[1,2,3,4],['a','b','c','d']],[[5,4,2,3],['a','b','c','d']],
        [[7,5,3,2],['a','b','c','d']],[[0,2,3,1],['a','b','c','d']]])
print('Array:',array, array.shape) #one-dimensional array, it will have only one
        dimension
print('Matrix:\n',matrix, matrix.shape)
print('matrix:\n',tensor, tensor.shape) #tensor, it will have two dimensions
```

```
Array: [1 2 3 4] (4,)
Matrix:
 [[1 2 3 4]
 [5 4 2 3]
 [7 5 3 2]
 [0 2 3 1]] (4, 4)
matrix:
 [[['1' '2' '3' '4']
  ['a' 'b' 'c' 'd']]

 [['5' '4' '2' '3']
  ['a' 'b' 'c' 'd']]

 [['7' '5' '3' '2']
  ['a' 'b' 'c' 'd']]

 [['0' '2' '3' '1']
  ['a' 'b' 'c' 'd']]] (4, 2, 4)
```

Let's see some operations between NumPy arrays:

```
a1 = np.array([1,2,3,4])
a2 = np.array([4,3,8,1])
print("Sum:",a1+a2) #somma tra vettori
print("Elementwise multiplication:",a1*a2) #moltiplicazione tra elementi corrispondenti
print("Power of two:",a1**2) #quadrato degli elementi
print("Elementwise power:",a1**a2) #elevamento a potenza elemento per elemento
print("Scalar product:",a1.dot(a2)) #prodotto scalare tra vettori
print("Minimum:",a1.min()) #minimo dell'array
print("Maximum:",a1.max()) #massimo dell'array
print("Sum:",a2.sum()) #somma di tutti i valori dell'array
print("Product:",a2.prod()) #prodotto di tutti i valori dell'array
print("Mean:",a1.mean()) #media di tutti i valori dell'array
```

```
Sum: [ 5  5 11  5]
Elementwise multiplication: [ 4  6 24  4]
Power of two: [ 1  4  9 16]
Elementwise power: [   1    8 6561    4]
Scalar product: 38
Minimum: 1
Maximum: 4
Sum: 16
Product: 96
Mean: 2.5
```

Operations on matrices:

```
m1 = np.array([[1,2,3,4],[5,4,2,3],[7,5,3,2],[0,2,3,1]])
m2 = np.array([[8,2,1,4],[0,4,6,1],[4,4,2,0],[0,1,8,6]])

print("Sum:",m1+m2) # Matrix sum
print("Elementwise product:\n",m1*m2) # Element-wise product
print("Power of two:\n",m1**2) # Square of the elements
print("Elementwise power:\n",m1**m2) # Element-wise power
print("Matrix multiplication:\n",m1.dot(m2)) # Matrix multiplication
print("Minimum:",m1.min()) # Minimum
print("Maximum:",m1.max()) # Maximum
print("Minimum along columns:",m1.min(0)) # Minimum along columns
print("Minimum along rows:",m1.min(1)) # Minimum along rows
print("Sum:",m1.sum()) # Sum of values
print("Mean:",m1.mean()) # Mean value
print("Diagonal:",m1.diagonal()) # Main diagonal of the matrix
print("Transposed:\n",m1.T) # Transposed matrix
```

```
Sum: [[ 9  4  4  8]
 [ 5  8  8  4]
 [11  9  5  2]
 [ 0  3 11  7]]
Elementwise product:
 [[ 8  4  3 16]
 [ 0 16 12  3]
 [28 20  6  0]
 [ 0  2 24  6]]
Power of two:
 [[ 1  4  9 16]
 [25 16  4  9]
 [49 25  9  4]
 [ 0  4  9  1]]
Elementwise power:
 [[   1    4    3  256]
 [   1  256   64    3]
 [2401  625    9    1]
 [   1    2 6561    1]]
Matrix multiplication:
 [[20 26 51 30]
 [48 37 57 42]
 [68 48 59 45]
 [12 21 26  8]]
Minimum: 0
Maximum: 7
Minimum along columns: [0 2 2 1]
Minimum along rows: [1 2 2 0]
Sum: 47
Mean: 2.9375
Diagonal: [1 4 3 1]
Transposed:
 [[1 5 7 0]
 [2 4 5 2]
 [3 2 3 3]
 [4 3 2 1]]
```

Exercises

Exercise 1

Define the list `[1,8,2,6,15,21,76,22,0,111,23,12,24]`, then:

- Print the first number of the list;
- Print the last number of the list;
- Print the sum of numbers with odd indices (e.g., 1,3,5,...) in the list;
- Print the list sorted in reverse order;

- Print the average of the numbers contained in the list.

Exercise 2

Define a dictionary `mesi` that maps the names of the months to their numerical equivalents. For example, the result of:

```
print mesi['Gennaio']
```

should be

```
1
```

Exercise 3

Consider the following lists:

```
l1 = [1,2,3]
l2 = [4,5,6]
l3 = [5,2,6]
```

Combine a for loop, **zip** and **enumerate** to obtain the following output:

```
0 → 10
1 → 9
2 → 15
```

Exercise 4

Repeat exercise **2** using dictionary comprehension. To do this, first define the list `['January','February','March','April','May','June', 'July','August','September','October','November','December']`.

Then build the desired dictionary using dictionary comprehension and the `enum` function

Exercise 5

Given the variables:

```
obj='triangle'
area=21.167822
```

print the strings:

- The area of the triangle is 21.16
- 21.1678 is the area of the triangle

Use string formatting to obtain the result.

Exercise 6

Write a function that extracts the domain from an email address. For example, if the address is “furnari@dmi.unict.it”, the function must extract “dmi.unict.it”.