SCRIPTING AND PROGRAMMING LABORATORY FOR DATA ANALYSIS

Lecture 2

Structured data types, modules, function definitions

Structured data types

Data types in Pyton

In python different data types are available:

- Basic (or simple) data types:
  - Int (long int)
  - Float
  - Complex
  - Bool
  - String
- **Structured data types (containers):**
  - **tuple ()**
  - **list []**
  - **dict {}**
  - **Set**
  - **Frozen set**

Structured data types

A nice feature of python is that, among the native data types, there are not only the basic ones common to many languages (int, float, bool, string, complex).

Python additionally features some **structured native data types.**

Python structured data types are divided into:

- Ordered
- Unordered

They can also be divided into:

- Mutable
- Immutable

List []

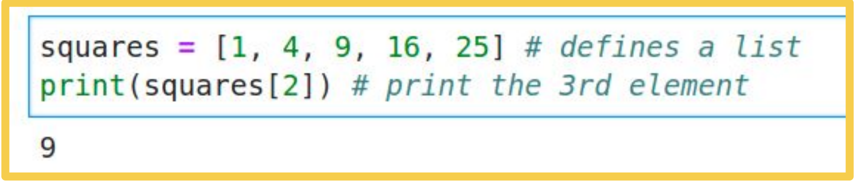
List defines possibly the *most versatile* structured data type.

It is an **ordered, mutable** (differently from strings) data type.

It can be written as a list of comma-separated values (items) between square brackets []. Lists can contain items of different types, but usually the items all have the same type.

Like strings (and all other built-in sequence types), lists can be indexed and sliced.

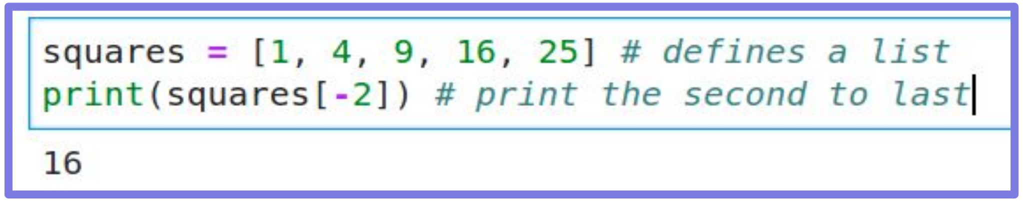
The [] operator allows you to access different elements of a list.



```
squares = [1, 4, 9, 16, 25] # defines a list
print(squares[2]) # print the 3rd element

9
```

LIST []

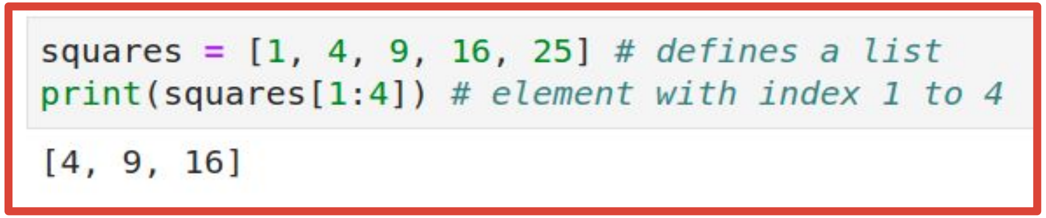
As for strings, indexing starts from 0 and goes to len(list)-1.
Negative indexing can be used to access the list from the end.



```
squares = [1, 4, 9, 16, 25] # defines a list
print(squares[-2]) # print the second to last
```

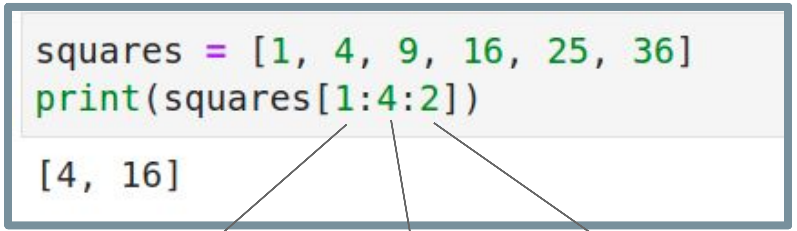
```
16
```

Note: slicing works the same as for strings!



```
squares = [1, 4, 9, 16, 25] # defines a list
print(squares[1:4]) # element with index 1 to 4
```

```
[4, 9, 16]
```



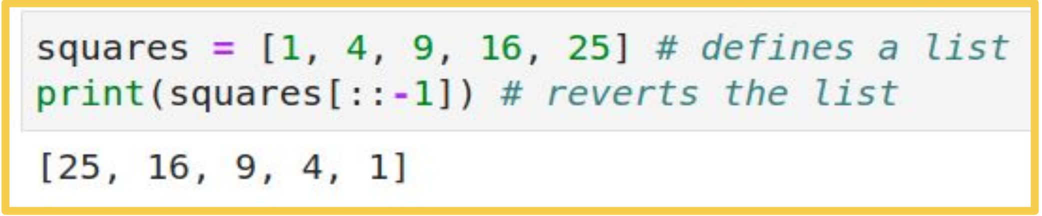
```
squares = [1, 4, 9, 16, 25, 36]
print(squares[1:4:2])
```

```
[4, 16]
```

First element to be considered

Last element to be considered

Step between elements (if not explicit, assumed to be +1)



```
squares = [1, 4, 9, 16, 25] # defines a list
print(squares[::-1]) # reverts the list
```

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

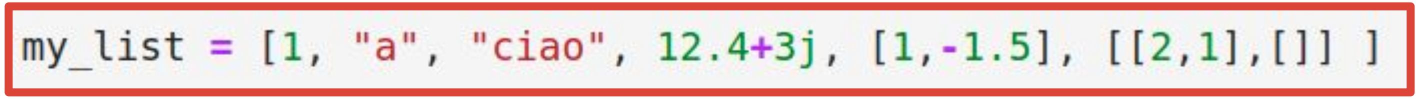
How to participate?

WEB

1. Connect to www.wooclap.com/SCRIPTPY
2. You can participate

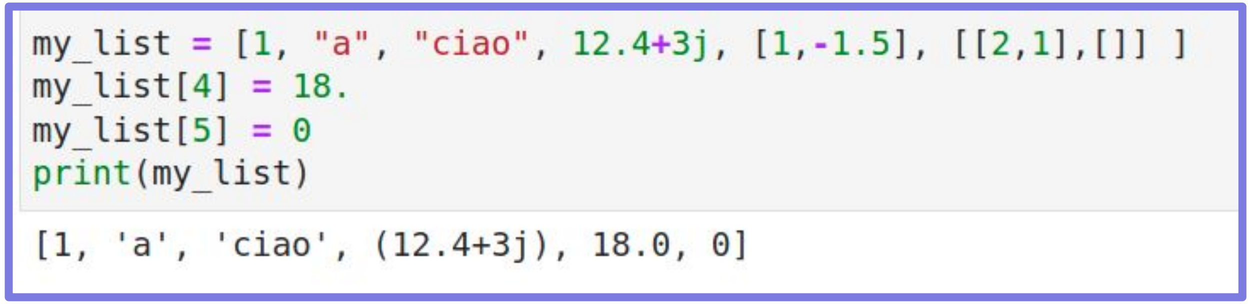
LIST []

Lists can contain **different data types**, also mixed among themselves. It is also possible to have lists of lists (of lists) and so on.



```
my_list = [1, "a", "ciao", 12.4+3j, [1,-1.5], [[2,1],[]] ]
```

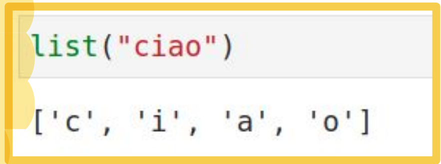
Lists are **mutable** data types (*differently from strings!*), meaning that they can be modified, elements can be added, removed and so forth



```
my_list = [1, "a", "ciao", 12.4+3j, [1,-1.5], [[2,1],[]] ]
my_list[4] = 18.
my_list[5] = 0
print(my_list)
```

```
[1, 'a', 'ciao', (12.4+3j), 18.0, 0]
```

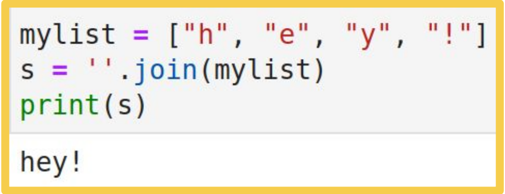
Lists are iterable. ***list()*** *converts an iterable data type into a list*



```
list("ciao")
```

```
['c', 'i', 'a', 'o']
```

"".join() *instead can be used to convert lists to strings*



```
mylist = ["h", "e", "y", "!"]
s = ''.join(mylist)
print(s)
```

```
hey!
```

LIST COMPREHENSION

It is possible to generate new lists in which each element is the result of a set of operations made on the elements of a previously existing ordered data set (e.g. another list). Or in which only some among the elements of the ordered data set are selected. **List comprehension is a feature of python that allows to define a new list in math (sets {}) style.**

MATH MODE

$S = \{x^2 : x \text{ in } \{0,1,\ldots,9\}\}$

$V = \{2^x : x \text{ in } \{1,\ldots,12\}\}$

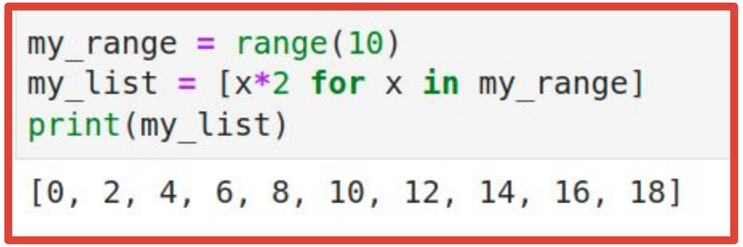
$M = \{x \mid x \text{ in } S \text{ and } x \text{ even}\}$

$\longrightarrow$

```
# PYTHONIC INSTRUCTIONS

S = [x**2 for x in range(10)]

V = [2**x for x in range(1,13)]

M = [x for x in V if x%2==0]
```

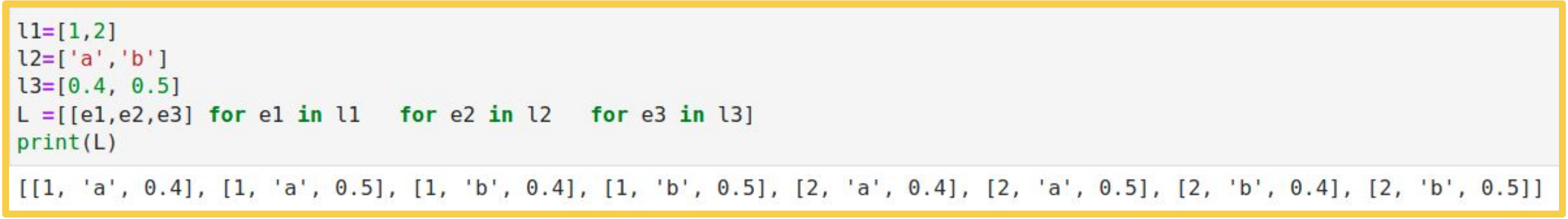
List comprehension

List comprehension works with two brackets [] **containing an expression** followed by a for and by one or more for cycles and if statements.



```
my_range = range(10)
my_list = [x*2 for x in my_range]
print(my_list)
```

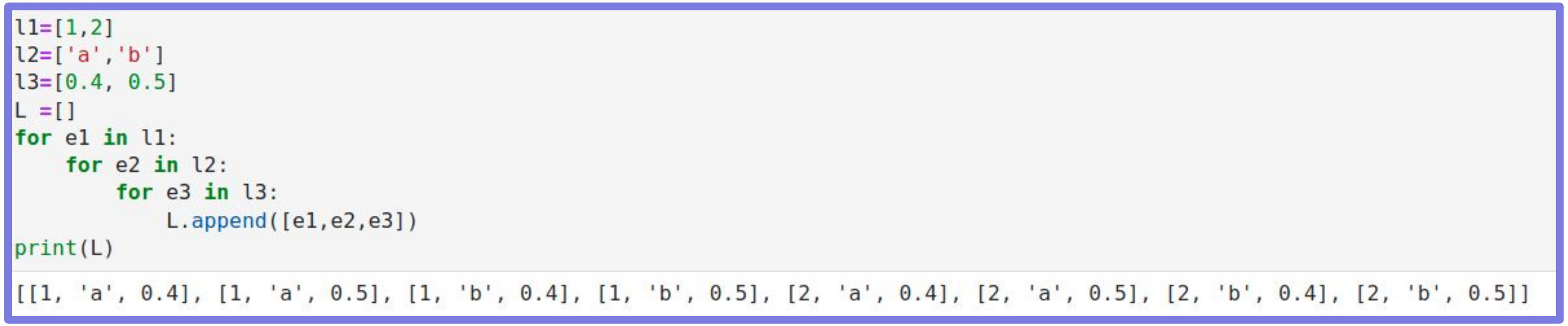
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```



```
l1=[1,2]
l2=['a','b']
l3=[0.4, 0.5]
L =[[e1,e2,e3] for e1 in l1   for e2 in l2   for e3 in l3]
print(L)
```

```
[[1, 'a', 0.4], [1, 'a', 0.5], [1, 'b', 0.4], [1, 'b', 0.5], [2, 'a', 0.4], [2, 'a', 0.5], [2, 'b', 0.4], [2, 'b', 0.5]]
```

Same as:



```
l1=[1,2]
l2=['a','b']
l3=[0.4, 0.5]
L =[]
for e1 in l1:
    for e2 in l2:
        for e3 in l3:
            L.append([e1,e2,e3])
print(L)
```

```
[[1, 'a', 0.4], [1, 'a', 0.5], [1, 'b', 0.4], [1, 'b', 0.5], [2, 'a', 0.4], [2, 'a', 0.5], [2, 'b', 0.4], [2, 'b', 0.5]]
```

List comprehension

For and if combined

```
L=[[x, y] for x in [1,2,3] for y in [3,1,4] if x != y]
```

same as

```
L = []
for x in [1,2,3]:
    for y in [3,1,4]:
        if x != y:
            L.append([x, y])
```

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

Nested list comprehension

```
matrx =  [[1,  2,   3,  4],
          [5,  6,   7,  8],
          [9,  10, 11, 12]]
matrx2 = [[row[i] for row in matrix] for i in range(4)]
print(matrx2)
```

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Methods on lists

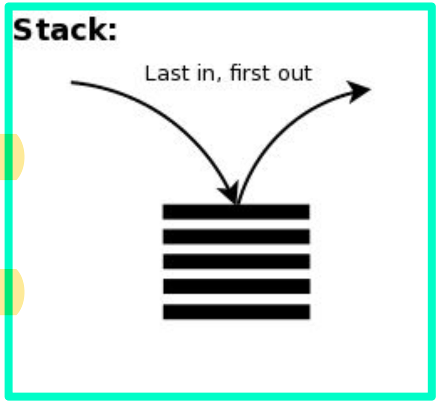
Since lists are mutable objects, *methods can be applied to change their content*. There are several **built-in methods** that can be used on lists:

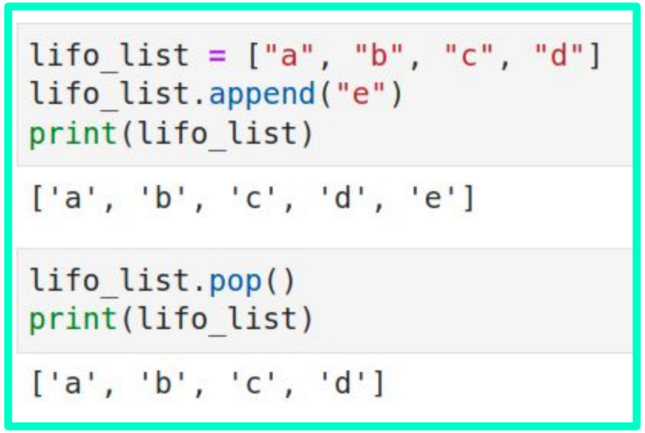
- ➔ **To insert**
  - ◆ **append(object)** #inserts an object at the end
  - ◆ **insert(index, object)** # inserts objects in position index
  - ◆ **extend(iterable)** # concatenates the list with an iterable data type
- ➔ **To search**
  - ◆ **index(value, [start, [stop]])** # searches for value and returns first **occurrence (start/stop=first/last index to be checked)**
  - ◆ **count(value)** # returns the number of occurrences of value in the list
- ➔ **To delete**
  - ◆ **remove(value)** # removes the first occurrence of value
  - ◆ **pop(index)** # removes the element in position index
- ➔ **To order**
  - ◆ **reverse()** #inverts the order of the elements
  - ◆ **sort(*, key=None, reverse=False)** #sorts the list-ascending order

Methods on lists

Methods implemented for lists are especially useful to be used in stacks or queues.

- **pop()** and **append()** can be used to implement stacks with the *last in, first out* underlying logic.
- **pop() with idex 0** and **append()** can be used to implement queues with the *first in, first out* underlying logic.



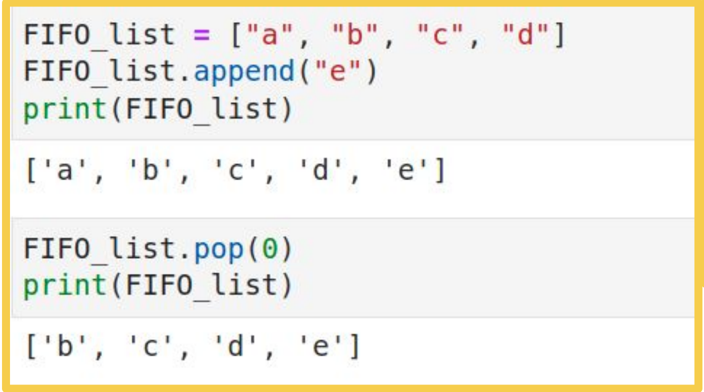


```
lifo_list = ["a", "b", "c", "d"]
lifo_list.append("e")
print(lifo_list)
```

```
['a', 'b', 'c', 'd', 'e']
```

```
lifo_list.pop()
print(lifo_list)
```

```
['a', 'b', 'c', 'd']
```

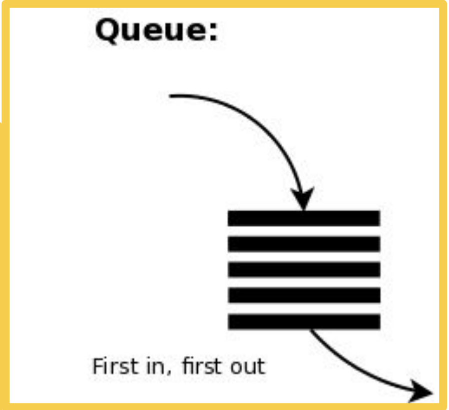


```
FIFO_list = ["a", "b", "c", "d"]
FIFO_list.append("e")
print(FIFO_list)
```

```
['a', 'b', 'c', 'd', 'e']
```

```
FIFO_list.pop(0)
print(FIFO_list)
```

```
['b', 'c', 'd', 'e']
```



More on lists

As much as for strings, lists can be *appended* with +, and a list can be *repeated* with *. **Note:** the operator + and the extend function do the same thing, but extend is much faster!

The operator += can also be used for lists.

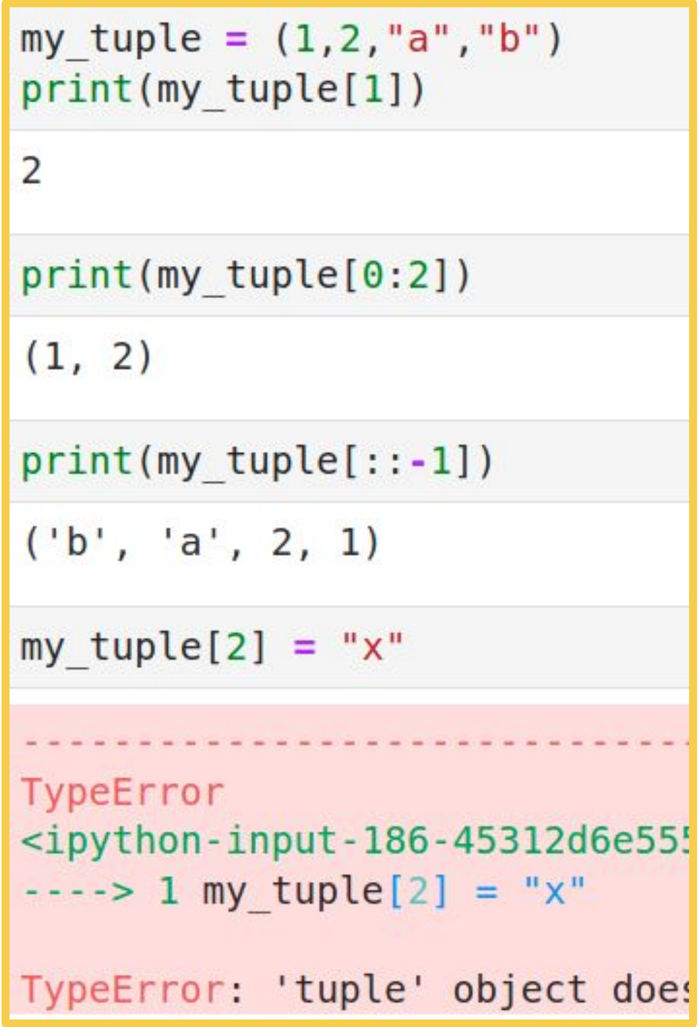
Important note: lists are a very flexible and they allow to store different data types, implying they use more memory. The module **numpy** (that will be presented in the following lectures) only allows a fixed type of data in its arrays. This implies a *reduced flexibility but a much greater performance* and more efficient memory usage. For this, in most applications of data science, numpy arrays should be preferred over lists.

TUPLE ()

Tuple is an **ordered, immutable** (differently from lists, and like strings) data type. Tuples *can contain items of different types* (as for lists).

A tuple can be written as a list of comma-separated values (items) between round brackets. Like lists , tuples **can be indexed and sliced** [:]; each tuple element can be accessed with the **[]** operator.

Tuples are immutable, so they do not feature methods as pop(), append(), sort() and so on!



```
my_tuple = (1,2,"a","b")
print(my_tuple[1])
```

```
2
```

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

```
(1, 2)
```

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

```
('b', 'a', 2, 1)
```

```
my_tuple[2] = "x"
```

```
---------------------------------
TypeError
<ipython-input-186-45312d6e55
----> 1 my_tuple[2] = "x"

TypeError: 'tuple' object doe
```

Tuples vs Lists

The very difference between the two is that **lists are mutable and tuples are not.**

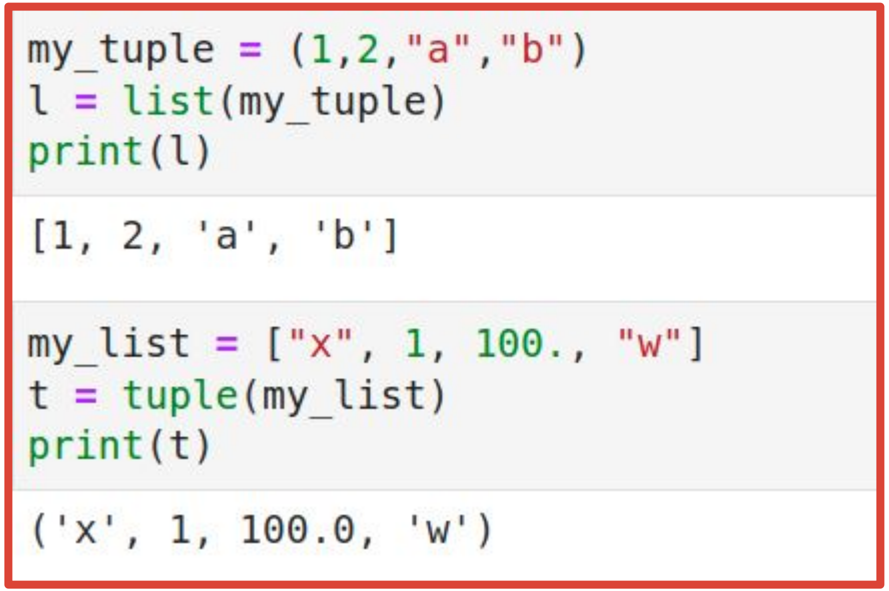
As lists are mutable, Python needs to allocate an extra memory block in case there is a need to extend the size of the list after it is created. Contrarily, as tuples are immutable and fixed size, Python allocates just the minimum memory block required for the data.

As a result, tuples are more memory efficient than the lists. Time-efficiency wise, tuples have a slight advantage over the lists especially when lookup to a value is considered.

In short: **use tuples when you want to avoid your collection of data to be changed and you need your code to be more efficient**. Use lists when you want you collection of data to be changed.

Tuples to lists, lists to tuples

Tuples can be converted to lists and vice versa via the **tuple()** and **list()** operators:



```
my_tuple = (1,2,"a","b")
l = list(my_tuple)
print(l)
```

```
[1, 2, 'a', 'b']
```

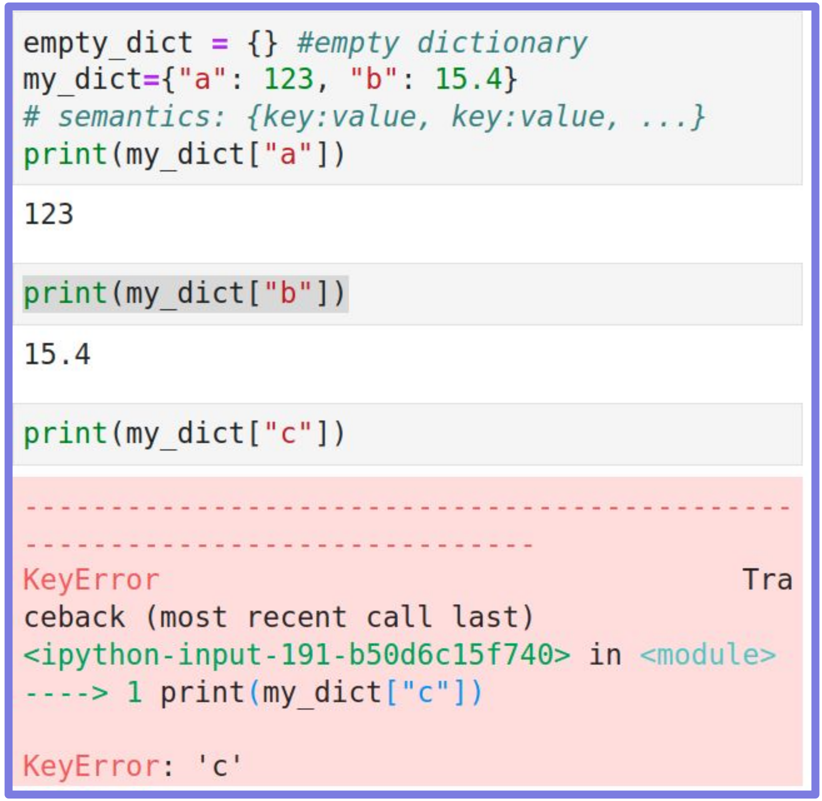
```
my_list = ["x", 1, 100., "w"]
t = tuple(my_list)
print(t)
```

```
('x', 1, 100.0, 'w')
```

DICTIONARY {}

A **dictionary** is a python built-in data type. It is an **unordered, mutable type**. All objects contained in a dictionary are identified via a key. ***Each object in a dictionary is composed by a key and a value.***

Keys are immutable types (for instance, tuples can be used as keys of a dictionary). Dictionaries are contained in curly brackets **{}**, and single elements can be accessed via the [] operator working on the keys.



```
empty_dict = {} #empty dictionary
my_dict={"a": 123, "b": 15.4}
# semantics: {key:value, key:value, ...}
print(my_dict["a"])
```

```
123
```

```
print(my_dict["b"])
```

```
15.4
```

```
print(my_dict["c"])
```

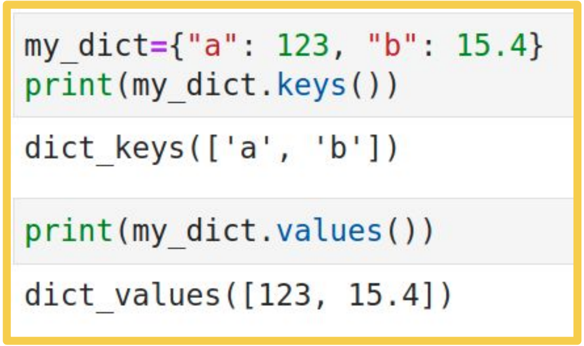
```
-----------------------------------------
-------------------------------
KeyError                               Tra
ceback (most recent call last)
<ipython-input-191-b50d6c15f740> in <module>
----> 1 print(my_dict["c"])

KeyError: 'c'
```

DICTIONARY {}

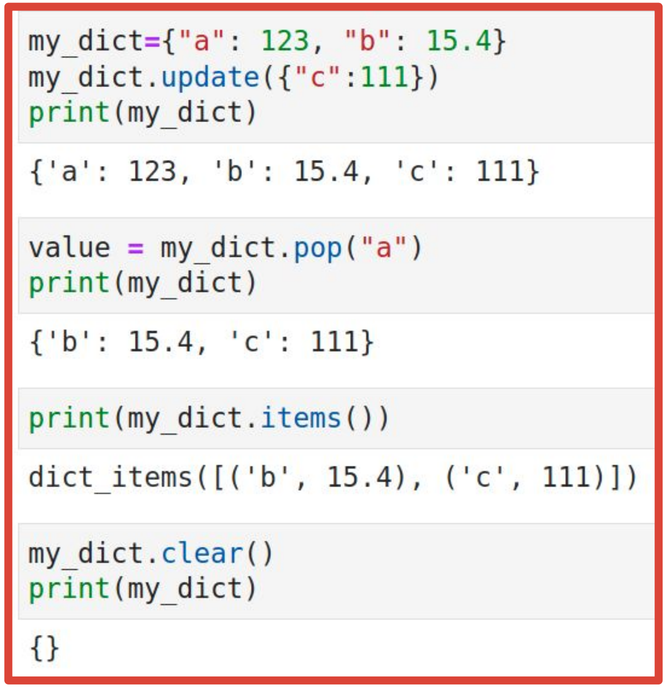
The **key()** and **value()** functions return the keys and the values in a dictionary.

Dictionaries can be updated and modified by means of built-in methods as

- **update({key: value})** #adds an element
- **clear()** # removes all items, returns {}
- **pop(K)** # removes element with key K
- **popitem()** # removes the last element, which is returned as value
- **get(key)** # returns the value of key
- **items()** # returns a set-like object providing a view on dictionary's items



```
my_dict={"a": 123, "b": 15.4}
print(my_dict.keys())
```
```
dict_keys(['a', 'b'])
```
```
print(my_dict.values())
```
```
dict_values([123, 15.4])
```



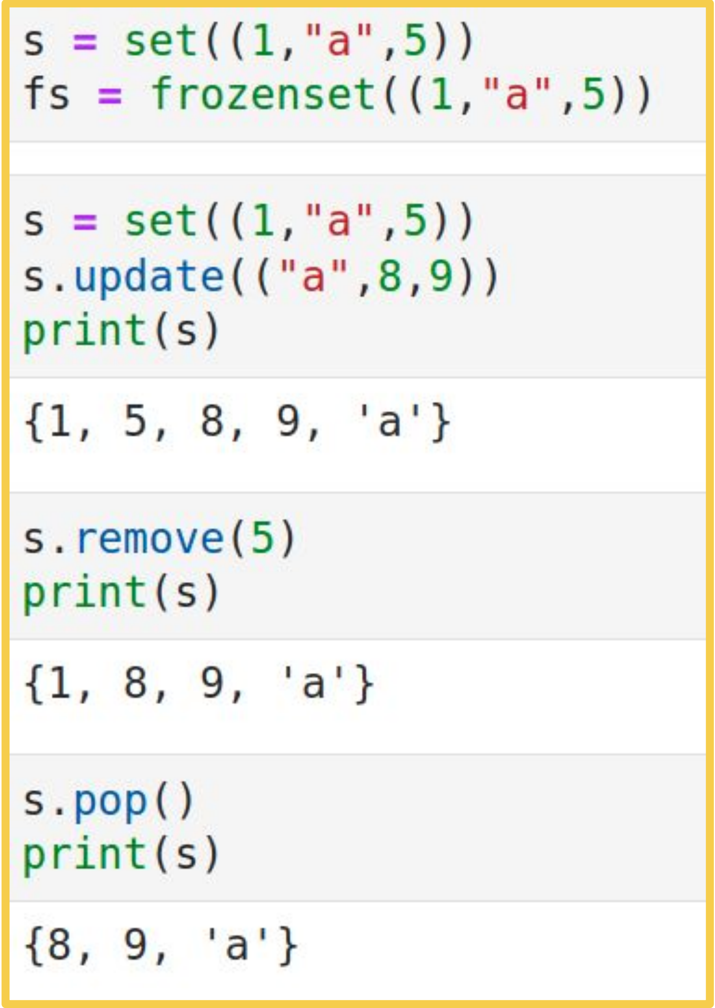
```
my_dict={"a": 123, "b": 15.4}
my_dict.update({"c":111})
print(my_dict)
```
```
{'a': 123, 'b': 15.4, 'c': 111}
```
```
value = my_dict.pop("a")
print(my_dict)
```
```
{'b': 15.4, 'c': 111}
```
```
print(my_dict.items())
```
```
dict_items([('b', 15.4), ('c', 111)])
```
```
my_dict.clear()
print(my_dict)
```
```
{}
```

SET - FROZENSET

They are **unordered collections of unique elements**. Set is **mutable**, frozenset is immutable.

Sets can be updated and modified by means of built-in methods as

- **add(obj)** #adds an element
- **update(objs)** #add objs to the set
- **discard(obj)** #removes obj from the set, if it is an element
- **remove(obj)** #removes obj from the set, obj must be an element
- **pop()** # removes an arbitrary element



```
s = set((1,"a",5))
fs = frozenset((1,"a",5))
```

```
s = set((1,"a",5))
s.update(("a",8,9))
print(s)
```

```
{1, 5, 8, 9, 'a'}
```

```
s.remove(5)
print(s)
```

```
{1, 8, 9, 'a'}
```

```
s.pop()
print(s)
```

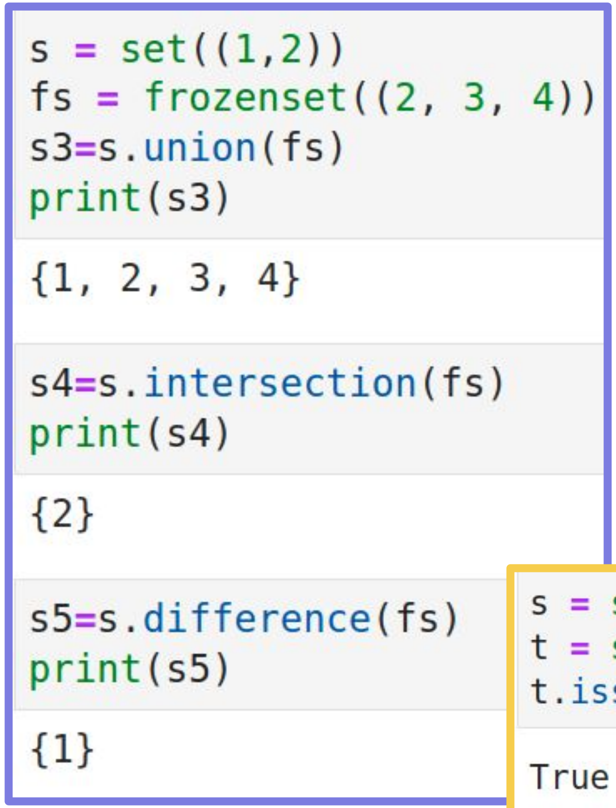
```
{8, 9, 'a'}
```

Set - Frozenset

Both data types feature methods to handle the operations between sets:

- union()
- intersection()
- difference()
- issubset()
- issuperset()

frozenset are immutable, thus they can be used as keys for dictionaries



```
s = set((1,2))
fs = frozenset((2, 3, 4))
s3=s.union(fs)
print(s3)
```

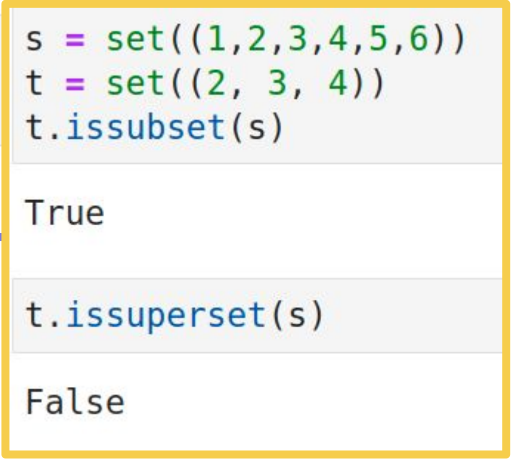
```
{1, 2, 3, 4}
```

```
s4=s.intersection(fs)
print(s4)
```

```
{2}
```

```
s5=s.difference(fs)
print(s5)
```

```
{1}
```



```
s = set((1,2,3,4,5,6))
t = set((2, 3, 4))
t.issubset(s)
```

```
True
```

```
t.issuperset(s)
```

```
False
```

Summary - structured data types

	Different data types	Mutable	Ordered
List []	✓	✓	✓
Tuple ()	✓	✗	✓
Dictionary {}	✓	✓	✗
Set	✓	✓	✗
Frozenset	✓	✗	✗

Shallow and deep copy

A **shallow copy** constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.

A **deep copy** constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

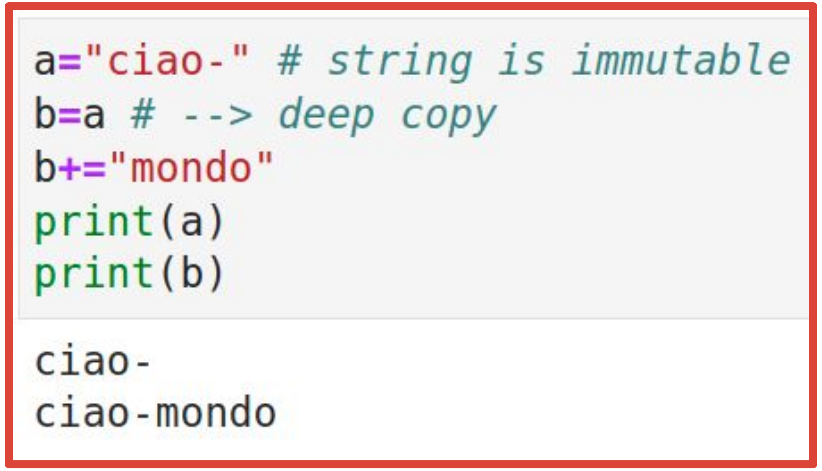
In other words:

Shallow copy is a copy by reference

Deep copy is a copy by value

By default, python uses shallow copy when copying mutable objects, and deep copy when copying immutable ones.

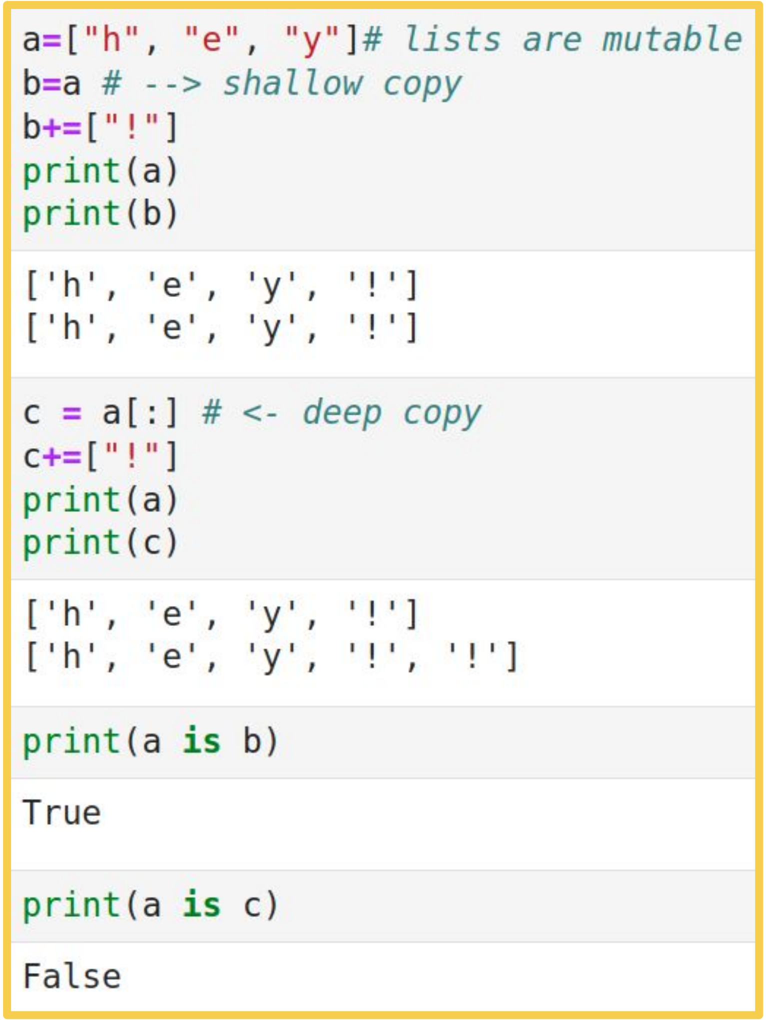
Shallow and deep copy



```
a="ciao-" # string is immutable
b=a # --> deep copy
b+="mondo"
print(a)
print(b)
```

```
ciao-
ciao-mondo
```

a is b returns True only if the two point at the same element in the memory. If they are identical but stored elsewhere in the memory, a is b returns False.



```
a=["h", "e", "y"]# lists are mutable
b=a # --> shallow copy
b+=["!"]
print(a)
print(b)
```

```
['h', 'e', 'y', '!']
['h', 'e', 'y', '!']
```

```
c = a[:] # <- deep copy
c+=["!"]
print(a)
print(c)
```

```
['h', 'e', 'y', '!']
['h', 'e', 'y', '!', '!']
```

```
print(a is b)
```

```
True
```

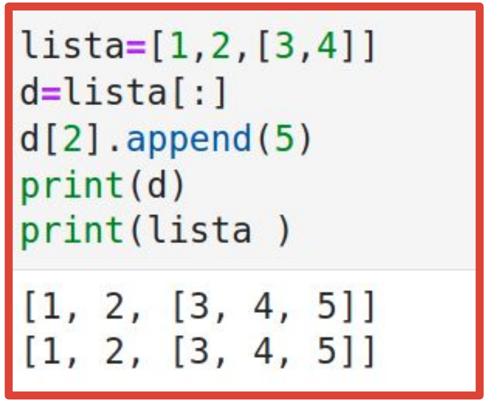
```
print(a is c)
```

```
False
```

Shallow and deep copy

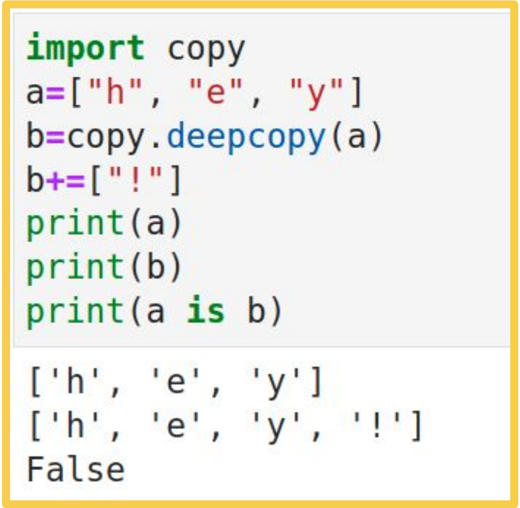
Careful! The **[:]** operator for lists only acts *at the first level*

You can import the **module copy** in order to control whether an object undergoes a shallow or a deep copy. This acts at all levels.



```
lista=[1,2,[3,4]]
d=lista[:]
d[2].append(5)
print(d)
print(lista )
```

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



```
import copy
a=["h", "e", "y"]
b=copy.deepcopy(a)
b+=["!"]
print(a)
print(b)
print(a is b)
```

```
['h', 'e', 'y']
['h', 'e', 'y', '!']
False
```

Functions

Functions in python

A **function** in python is an organized block of code that can be used to execute a given task. Functions allow for more code modularity and more usability. There are several **built in functions** in python, as *print()*, *help()*, *type()*...

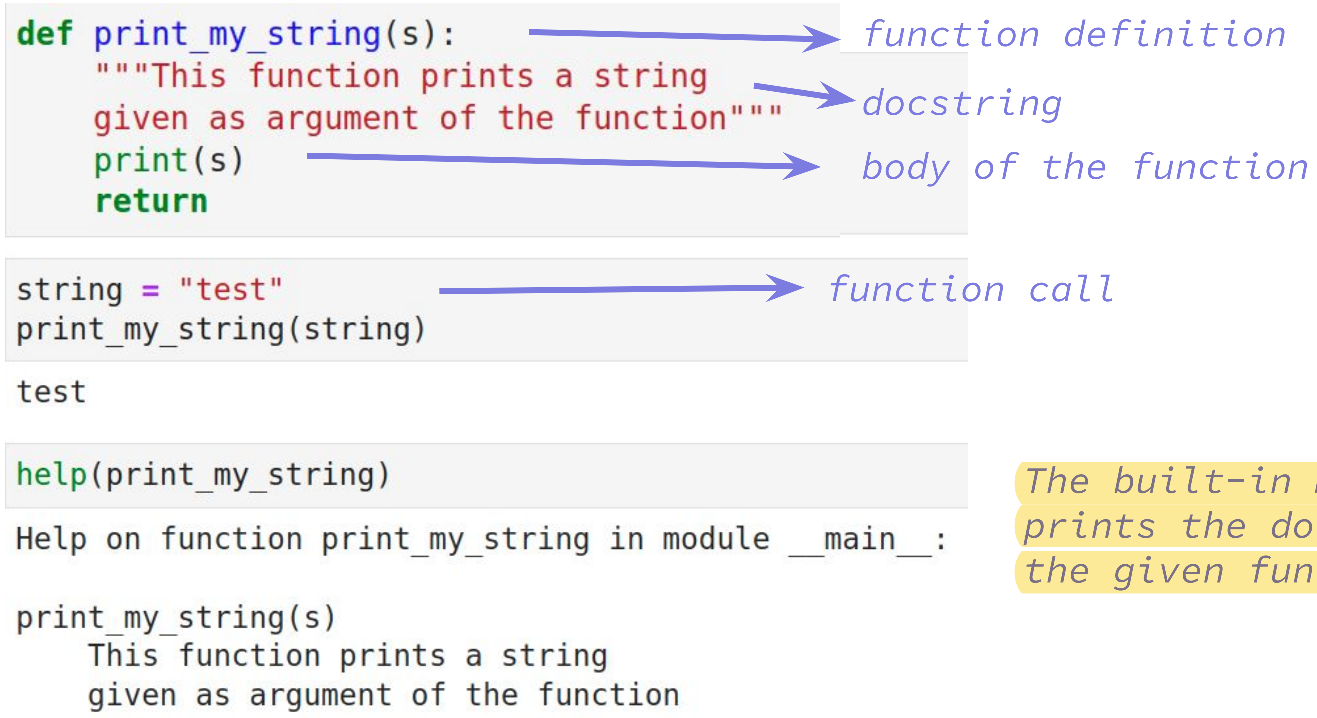
It is of course possible to define your own functions, so that you can have **user-defined functions**.

Defining a function in python can be done with the following syntax:

```
def NameFunction(arg1,arg2,…,argN):

       body of the function

       return(args)
```

Best practices: docstring

It is very advisable to use **docstrings** inside functions:



```
def print_my_string(s):
    """This function prints a string
    given as argument of the function"""
    print(s)
    return
```

```
string = "test"
print_my_string(string)
```

```
test
```

```
help(print_my_string)
```

```
Help on function print_my_string in module __main__:

print_my_string(s)
    This function prints a string
    given as argument of the function
```

The built-in method help prints the docstring of the given function.

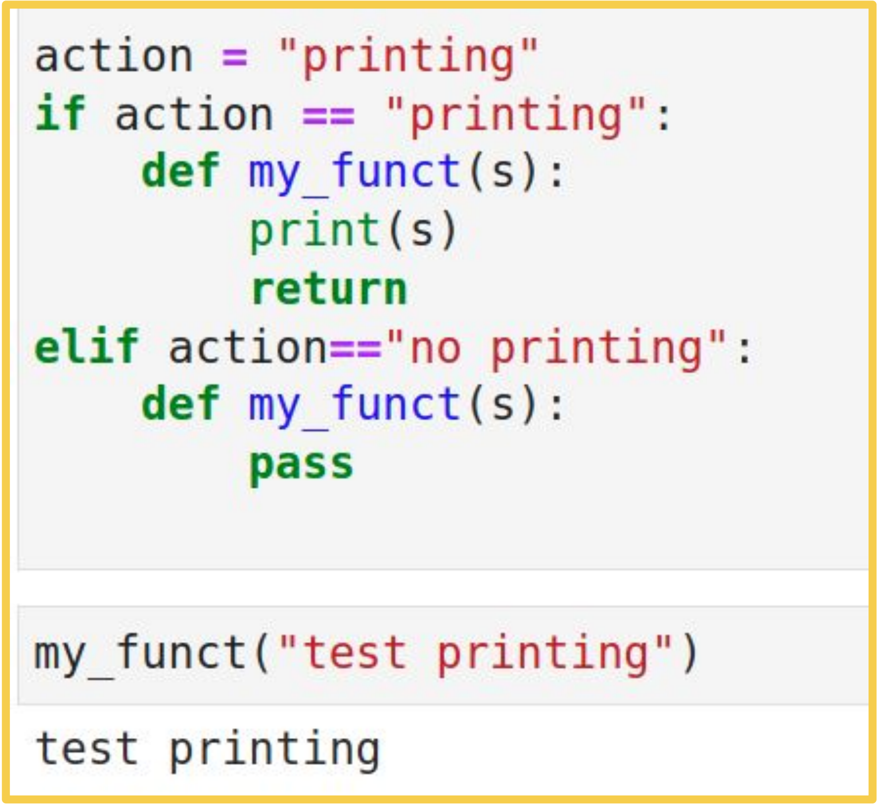
Def statement

The **def statement is an executive statement.**

When it is executed, it creates in that moment a *new object*, which is the function, and it gives it a name.

In python you can declare a function within other statements. Since every statement is evaluated at run time, the function does not exist before the def statement is found and executed.

Note: when one imports a module, all functions in the module are created at the import statement


```
action = "printing"
if action == "printing":
    def my_funct(s):
        print(s)
        return
elif action=="no printing":
    def my_funct(s):
        pass
```

```
my_funct("test printing")
```

```
test printing
```

Function arguments

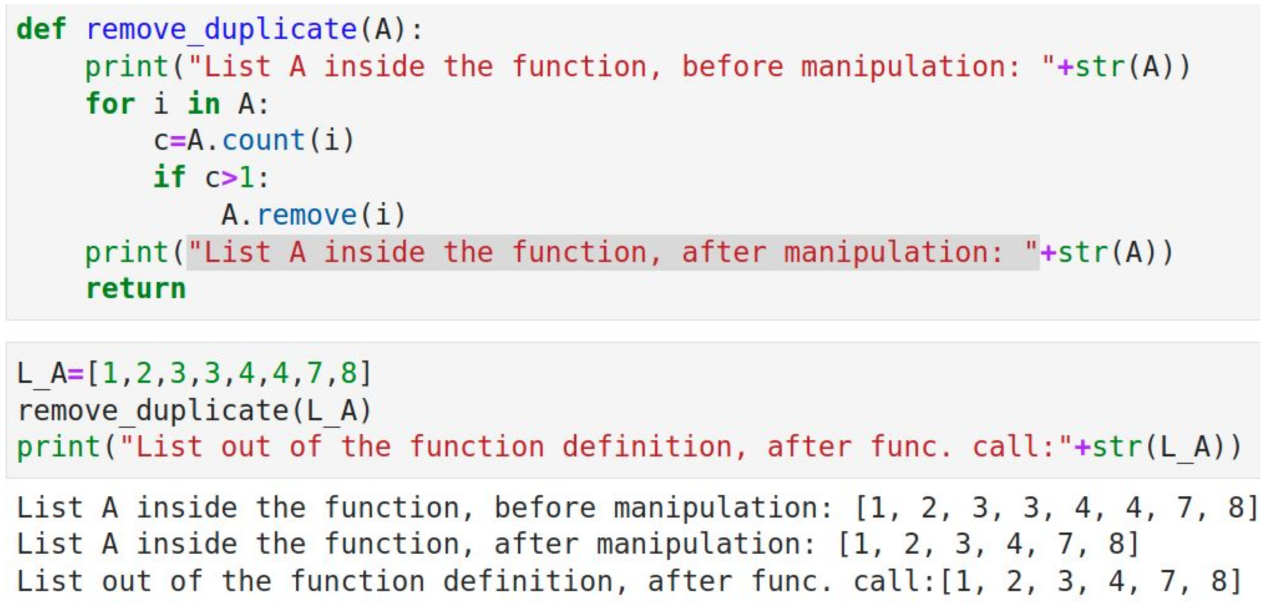
It is important to distinguish between mutable and immutable arguments when they are used as function arguments.

Immutable data types (strings, tuples, numbers as int, float, complex) are passed by value. It means that any manipulation of the passed argument within the function does not have an effect outside of the function body.

Mutable data types (lists, dictionaries), instead, are passed by reference. It means that manipulations of those parameters within the function result in those objects to be also varied in the part of the code that is calling the function.

Note: variables defined within the scope of a function only existin within the function.

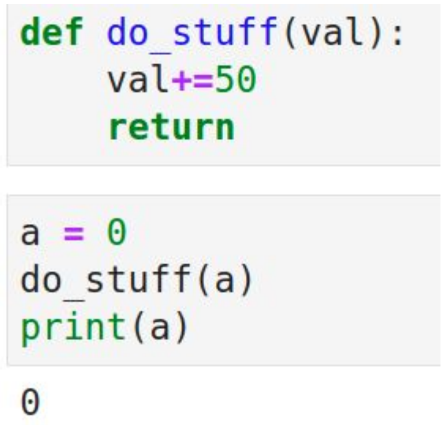
FUNCTION ARGUMENTS



```python
def remove_duplicate(A):
    print("List A inside the function, before manipulation: "+str(A))
    for i in A:
        c=A.count(i)
        if c>1:
            A.remove(i)
    print("List A inside the function, after manipulation: "+str(A))
    return
```

```python
L_A=[1,2,3,3,4,4,7,8]
remove_duplicate(L_A)
print("List out of the function definition, after func. call:"+str(L_A))
```

```
List A inside the function, before manipulation: [1, 2, 3, 3, 4, 4, 7, 8]
List A inside the function, after manipulation: [1, 2, 3, 4, 7, 8]
List out of the function definition, after func. call:[1, 2, 3, 4, 7, 8]
```



```python
def do_stuff(val):
    val+=50
    return
```

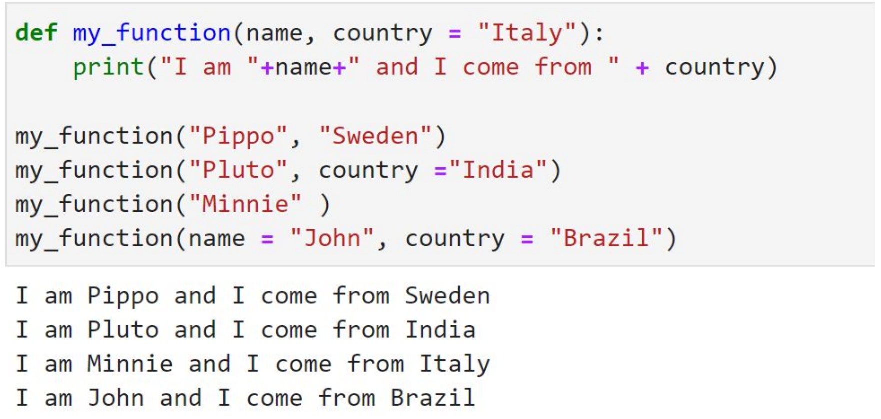
```python
a = 0
do_stuff(a)
print(a)
```

```
0
```

FUNCTION ARGUMENTS

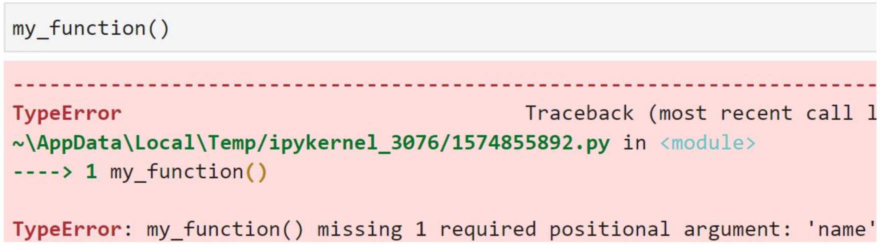
The number of required arguments that are passed into a function must comply with the function definition. Note that they can be passed either in the right order or with a *key=value* syntax.

It is also possible to define some optional arguments whose values are set to a default. Optional arguments must follow the mandatory arguments both in the function definition and in the function call.



```
def my_function(name, country = "Italy"):
    print("I am "+name+" and I come from " + country)

my_function("Pippo", "Sweden")
my_function("Pluto", country ="India")
my_function("Minnie" )
my_function(name = "John", country = "Brazil")
```

```
I am Pippo and I come from Sweden
I am Pluto and I come from India
I am Minnie and I come from Italy
I am John and I come from Brazil
```



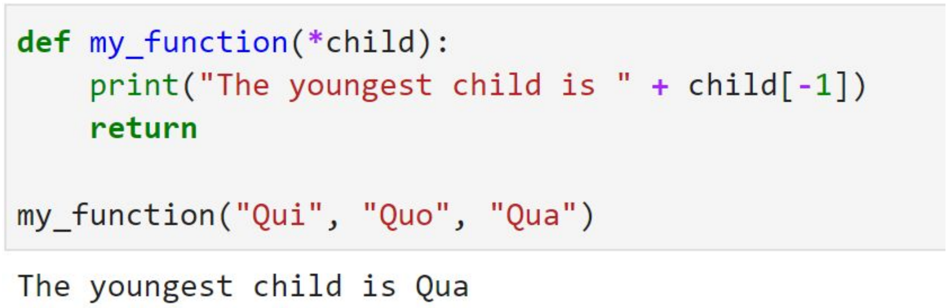
```
my_function()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call l
~\AppData\Local\Temp/ipykernel_3076/1574855892.py in <module>
----> 1 my_function()

TypeError: my_function() missing 1 required positional argument: 'name'
```

ARBITRARY FUNCTION ARGUMENTS *ARG

Python allows to use as an argument an object containing an unknown number of arguments - this is useful when you don't know the number of arguments you want to give to your function. You can put the symbol * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:



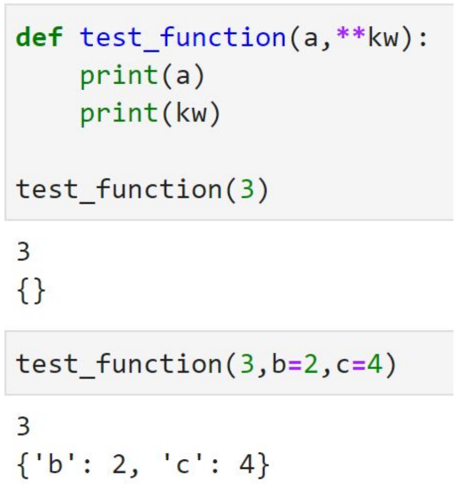
```
def my_function(*child):
    print("The youngest child is " + child[-1])
    return

my_function("Qui", "Quo", "Qua")
```

```
The youngest child is Qua
```

Arbitrary function arguments **arg

If you do not know how many keyword arguments will be passed into your function, add two asterisks ** before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly.

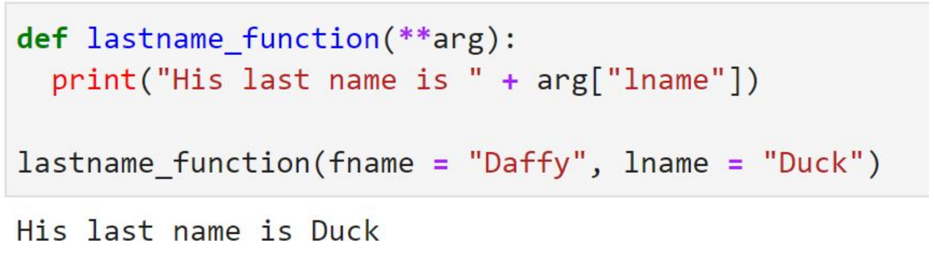


```
def test_function(a,**kw):
    print(a)
    print(kw)

test_function(3)
```

```
3
{}
```

```
test_function(3,b=2,c=4)
```

```
3
{'b': 2, 'c': 4}
```

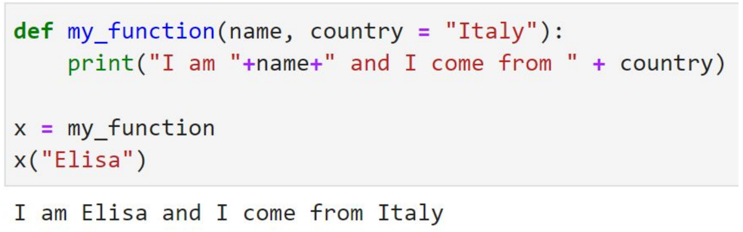


```
def lastname_function(**arg):
  print("His last name is " + arg["lname"])

lastname_function(fname = "Daffy", lname = "Duck")
```

```
His last name is Duck
```

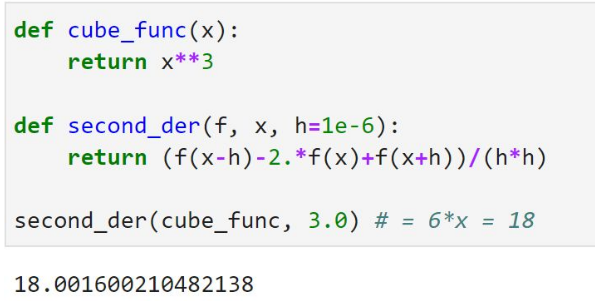
FUNCTIONS ARE OBJECTS

Python functions are objects, meaning that they are extremely flexible. This allows for functions to be used as arguments of other functions, to be returned by functions, to be associated with a variable and so forth!



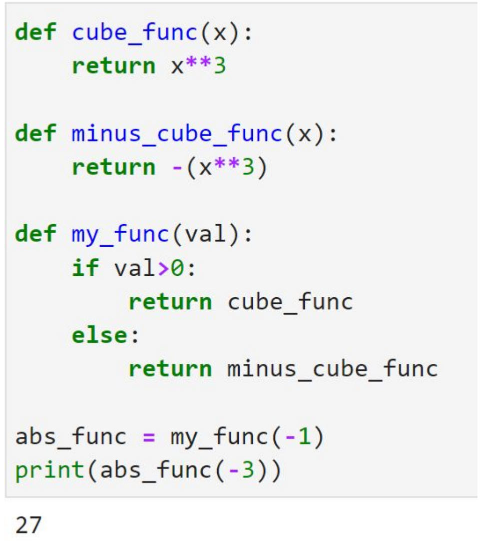
```
def my_function(name, country = "Italy"):
    print("I am "+name+" and I come from " + country)

x = my_function
x("Elisa")
```

```
I am Elisa and I come from Italy
```



```
def cube_func(x):
    return x**3

def second_der(f, x, h=1e-6):
    return (f(x-h)-2.*f(x)+f(x+h))/(h*h)

second_der(cube_func, 3.0) # = 6*x = 18
```

```
18.001600210482138
```

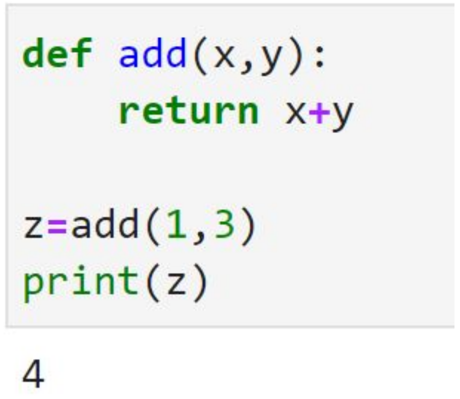


```
def cube_func(x):
    return x**3

def minus_cube_func(x):
    return -(x**3)

def my_func(val):
    if val>0:
        return cube_func
    else:
        return minus_cube_func

abs_func = my_func(-1)
print(abs_func(-3))
```

```
27
```

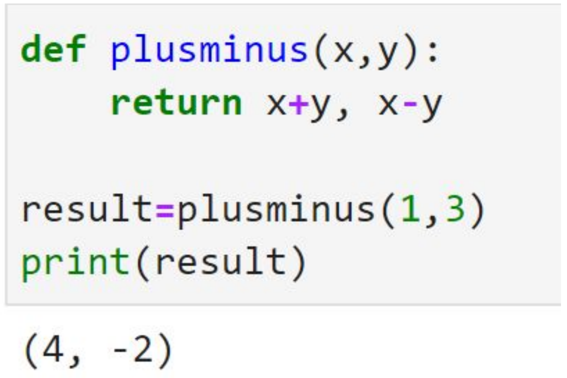
Return value

The return key allows to specify which value(s) should be returned by a function. In python there is a way to specify more than one return value.



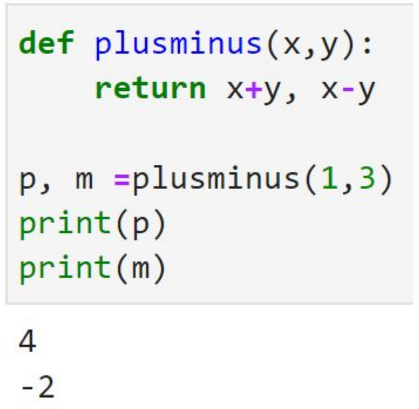
```
def add(x,y):
    return x+y

z=add(1,3)
print(z)
```

```
4
```



```
def plusminus(x,y):
    return x+y, x-y

result=plusminus(1,3)
print(result)
```

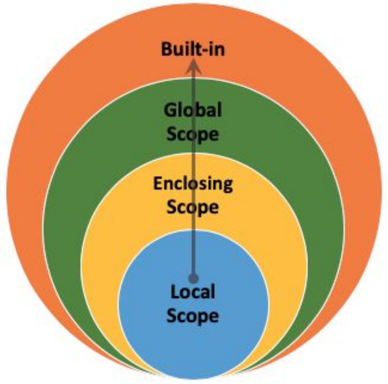
```
(4, -2)
```



```
def plusminus(x,y):
    return x+y, x-y

p, m =plusminus(1,3)
print(p)
print(m)
```

```
4
-2
```

Variables scoping



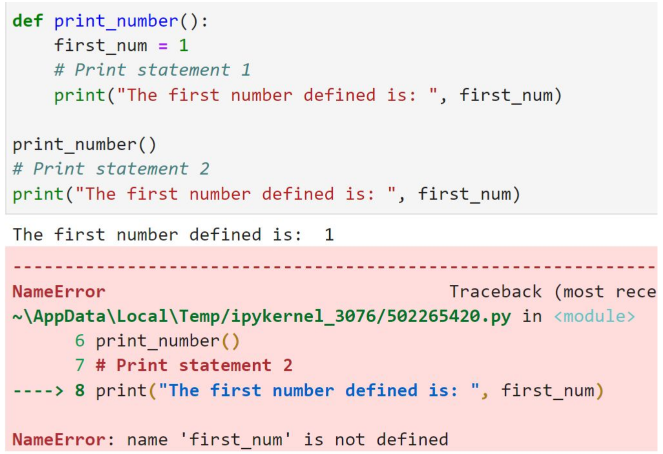
The scope of a variable in python is that part of the code where it is visible.

Not all variables can be accessed from anywhere in a program. The part of a program where a variable is accessible is called its scope. There are four major types of variable scope and is the basis for the LEGB rule. LEGB stands for

Local -> Enclosing -> Global -> Built-in.

Local Scope

Whenever you define a variable within a function, its scope lies **only** within the function. It is accessible from the point at which it is defined until the end of the function and exists for as long as the function is executing



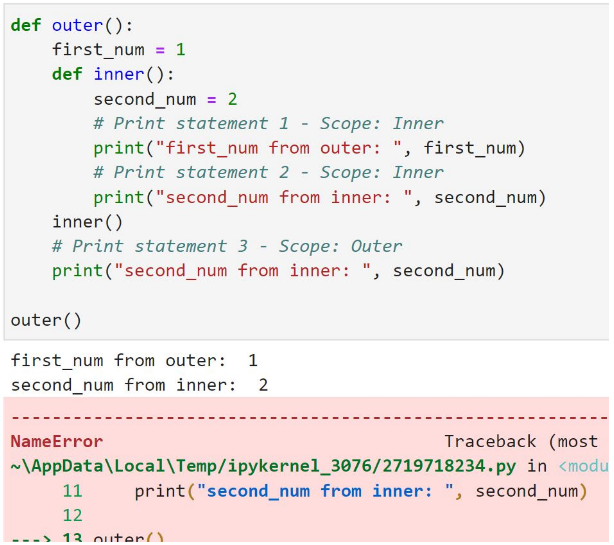
```
def print_number():
    first_num = 1
    # Print statement 1
    print("The first number defined is: ", first_num)

print_number()
# Print statement 2
print("The first number defined is: ", first_num)
```

```
The first number defined is:  1
---------------------------------------------------------------------
NameError                                 Traceback (most rece
~\AppData\Local\Temp/ipykernel_3076/502265420.py in <module>
      6 print_number()
      7 # Print statement 2
----> 8 print("The first number defined is: ", first_num)

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

Variables scoping

Enclosing Scope

What if we have a nested function (function defined inside another function)? This function is enclosing another one:

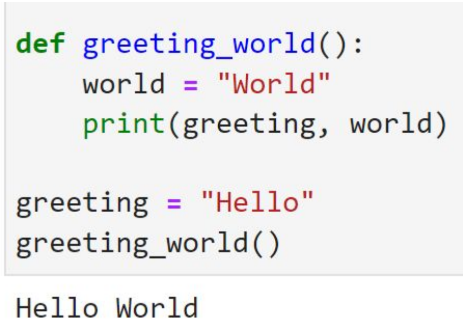


```
def outer():
    first_num = 1
    def inner():
        second_num = 2
        # Print statement 1 - Scope: Inner
        print("first_num from outer: ", first_num)
        # Print statement 2 - Scope: Inner
        print("second_num from inner: ", second_num)
    inner()
    # Print statement 3 - Scope: Outer
    print("second_num from inner: ", second_num)

outer()
```

```
first_num from outer:  1
second_num from inner:  2
---------------------------------------------------------
NameError                                 Traceback (most
~\AppData\Local\Temp/ipykernel_3076/2719718234.py in <modu
     11     print("second_num from inner: ", second_num)
     12
---> 13 outer()
```

Global Scope

Whenever a variable is defined outside any function in your code, it becomes a global variable, and its scope is anywhere within the program, which means it can be used by any function as long as it is defined before the function is called.



```
def greeting_world():
    world = "World"
    print(greeting, world)

greeting = "Hello"
greeting_world()
```

```
Hello World
```

Built-in Scope

All the special reserved keywords fall under this scope. We can call the keywords anywhere within our program without having to define them before use.

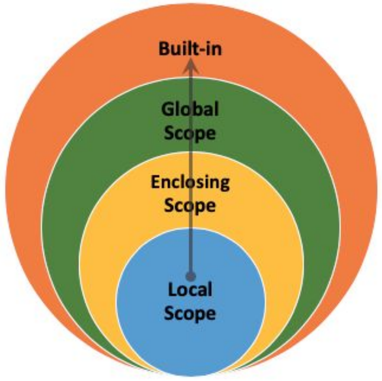
For instance:

False, True, None, if, else…

Variables scoping

Local variables can overwrite local ones.

In this case, the global statement can be useful



```
def square(x):
    x=x*x
    print("Inside  function: x = ", x)

x=5
square(x)
print("Outside function: x = ", x)
```

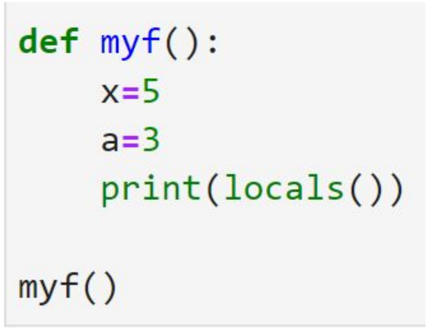
```
Inside  function: x =  25
Outside function: x =  5
```

```
x=5

def square(w):
    global x
    x=w
    x=x*x
    print("Inside  function: x = ", x)

square(x)
print("Outside function: x = ", x)
```

```
Inside  function: x =  25
Outside function: x =  25
```

Note: the locals() and globals() functions return a **dictionary** containing respectively the local and global variables in the given scope.

VARIABLES SCOPING



```python
def myf():
    x=5
    a=3
    print(locals())

myf()
```

```
{'x': 5, 'a': 3}
```

```python
globals()
```

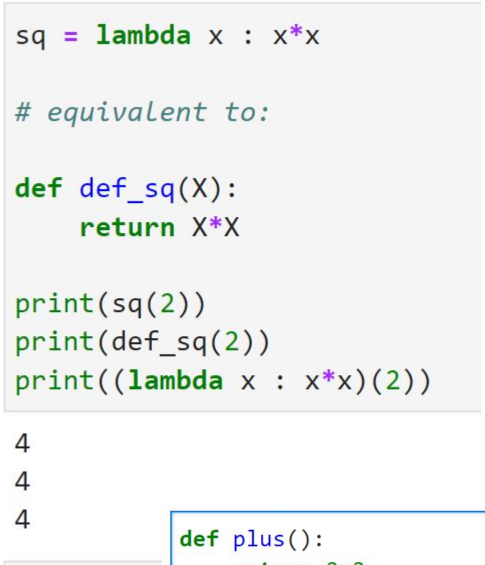
```
{'__name__': '__main__',
 '__doc__': 'Automatically created module for IPython interactive environment',
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
 '__builtins__': <module 'builtins' (built-in)>,
 '_ih': ['',
  'x=5\n\ndef square(w):\n    global x\n    x=w\n    x=x*x\n    print("Inside
function: x = ", x)\n    \nsquare(x)\nprint("Outside function: x = ", x)',
  'locals()'],
 '_oh': {},
 '_dh': ['C:\\Users\\Elisa\\Dropbox\\Postdoc'],
 'In': ['',
  'x=5\n\ndef square(w):\n    global x\n    x=w\n    x=x*x\n    print("Inside
function: x = ", x)\n    \nsquare(x)\nprint("Outside function: x = ", x)',
  'locals()'],
 'Out': {},
 'get_ipython': <bound method InteractiveShell.get_ipython of
<ipykernel.zmqshell.ZMQInteractiveShell object at 0x00000162DB2D5340>>,
 'exit': <IPython.core.autocall.ZMQExitAutocall at 0x162db3396d0>,
 'quit': <IPython.core.autocall.ZMQExitAutocall at 0x162db3396d0>,
 '_': '',
 '__': '',
 '___': '',
 '_i': '\n\nx=5\n\ndef square(w):\n    global x\n    x=w\n    x=x*x\n
print("Inside  function: x = ", x)\n    \nsquare(x)\nprint("Outside function: x = ",
x)',
 '_ii': '',
 '_iii': '',
 '_i1': '\n\nx=5\n\ndef square(w):\n    global x\n    x=w\n    x=x*x\n
print("Inside  function: x = ", x)\n    \nsquare(x)\nprint("Outside function: x = ",
x)',
 'x': 25,
 'square': <function __main__.square(w)>,
 '_i2': 'locals()'}
```

Lambda functions

Python allows for inline functions without a name to be defined the lambda keyword. The syntax is:

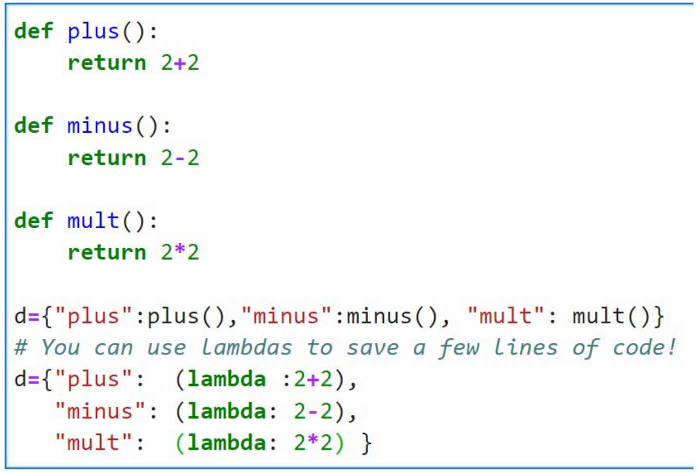
lambda <args>: <expression>

Which is equivalent to a function with arguments <args> and return value <expression>.



```
sq = lambda x : x*x

# equivalent to:

def def_sq(X):
    return X*X

print(sq(2))
print(def_sq(2))
print((lambda x : x*x)(2))
```

```
4
4
4
```



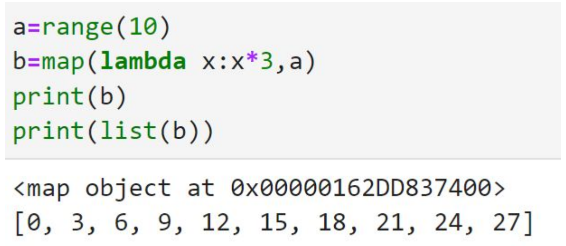
```
def plus():
    return 2+2

def minus():
    return 2-2

def mult():
    return 2*2

d={"plus":plus(),"minus":minus(), "mult": mult()}
# You can use lambdas to save a few lines of code!
d={"plus":  (lambda :2+2),
   "minus": (lambda: 2-2),
   "mult":  (lambda: 2*2) }
```

LAMBDA FUNCTIONS

Lambda functions combined with some built-in functions that allow for specific operations. The built-in functions are:

- filter(function or None, sequence)
- map(function, sequence,[sequence,…])

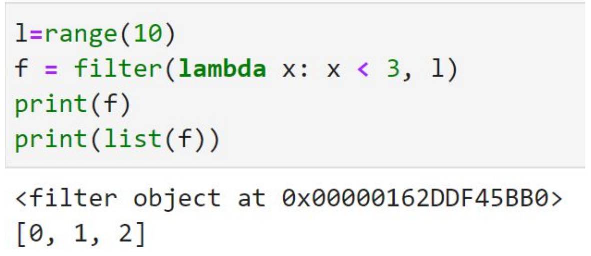
map:
Applies a function to all elements in a given sequence and returns the modified elements



```
a=range(10)
b=map(lambda x:x*3,a)
print(b)
print(list(b))
```

```
<map object at 0x00000162DD837400>
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

filter:
Applies a filter to a sequence of data and returns all the elements for which the given function is True

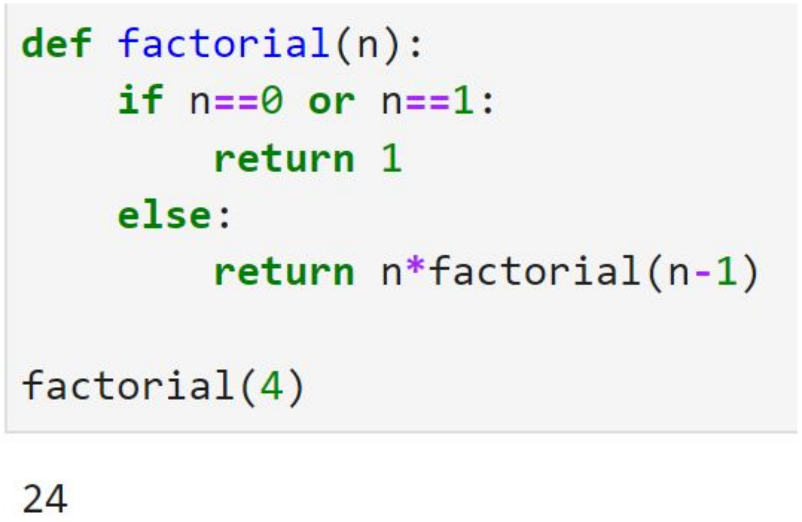


```
l=range(10)
f = filter(lambda x: x < 3, l)
print(f)
print(list(f))
```

```
<filter object at 0x00000162DDF45BB0>
[0, 1, 2]
```

Recursive functions

A recursive function contains
a call to itself in its body



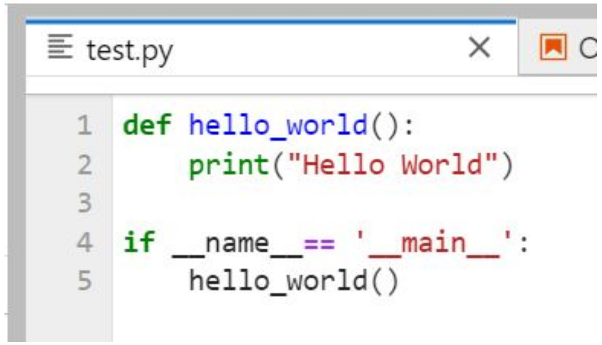
```
def factorial(n):
    if n==0 or n==1:
        return 1
    else:
        return n*factorial(n-1)

factorial(4)
```

```
24
```

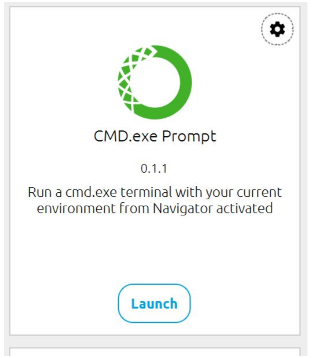
Modules & introspection

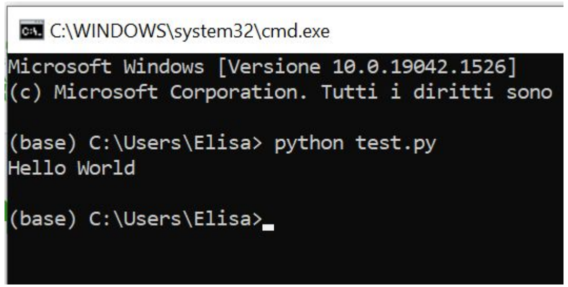
Modules in Python

A file in the computer ending in .py is seen as a module by python. A module can contain any python code.


test.py

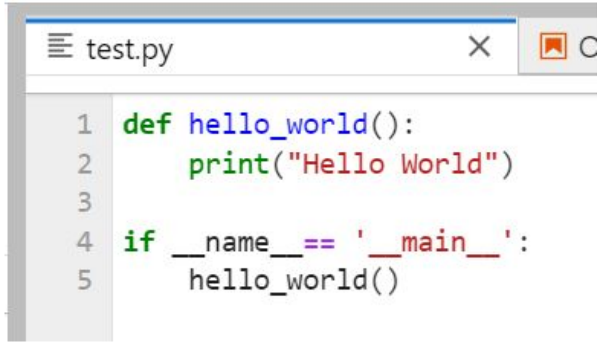
```python
def hello_world():
    print("Hello World")

if __name__== '__main__':
    hello_world()
```


CMD.exe Prompt
0.1.1
Run a cmd.exe terminal with your current environment from Navigator activated
Launch


C:\WINDOWS\system32\cmd.exe

```
Microsoft Windows [Versione 10.0.19042.1526]
(c) Microsoft Corporation. Tutti i diritti sono

(base) C:\Users\Elisa> python test.py
Hello World

(base) C:\Users\Elisa>_
```

Import and use a module


test.py

```python
def hello_world():
    print("Hello World")

if __name__== '__main__':
    hello_world()
```

```python
import test
```

```python
dir(test) # returns the names in the current scope
```

```
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'hello_world']
```

```python
print(test.__file__) # returns the file path
```

```
C:\Users\Elisa\test.py
```

```python
test.hello_world()
```

```
Hello World
```

Import and use a Module

Python is constituted by many modules that are either provided when python itself is installed, or can be added and installed at a later time (e.g. numpy, scipy, matplotlib…)

Modules can be imported with from/import commands:

➔ from os import *
➔ from os import
➔ from os import path as PP
➔ import os
➔ import os as O