Python: Functions and parameters

Introduction to Computer Programming
Bachelor in Data Science

Roberto Guidi

roberto.guidi@supsi.ch

Fall 2021

SUPSI University of Applied Sciences and Arts of Southern Switzerland

Content created in collaboration with: L. Grossi, F. Landoni

Blocks of code and indentation

Multiple statements are grouped in what we call code blocks.

In Python code blocks consists of statements that present all the same indentation level.

In other languages in contrast, it is common to use curly brackets {...}, to define block start and end.

Indentation is done in the source code by using either whitespaces or tabs.

Refresher

Selection statement

```
if condition1:
  statementSequence1
elif condition2:
  statementSequence2
elif condition3:
  statementSequence3
else:
  statementSequence4
```

Refresher

Loop Statements

```
while condition:
  statementSequence
```

Executed zero or more times.

```
for i in range(5):
  statementSequence2
```

Executed a fixed number of times.

Refresher

Code reuse

What can you do if you have a piece of code in your program that you want to reuse several times?

```
x = 5
max_x = 100
rounds_x = 20
for i in range(rounds_x):
    if x < max_x:
        x += x
    else:
        break
```

Example: you want to execute the above code for three variables `x`, `y` and `z` (instead of doing it only for `x`). At the end you want to show the sum of the value contained in the three variables.

Copy/Paste solution: Effective?

```
x, max_x, rounds_x = 5, 100, 20
for i in range(rounds_x):
    if x < max_x:
        x += x
    else:
        break

y, max_y, rounds_y = 4, 80, 15
for i in range(rounds_y):
    if y < max_y:
        y += y
    else:
        break

z, max_z, rounds_z = 6, 115, 30
for i in range(rounds_z):
    if z < max_z:
        z += z
    else:
        break
```

. . . what if we must do this for hundred variables?

Solution using a subprogram

```
x = 5
max_x = 100
rounds_x = 20
x = do_calculations(x, max_x, rounds_x)

y = 4
max_y = 80
rounds_y = 15
y = do_calculations(y, max_y, rounds_y)

z = 6
max_z = 115
rounds_z = 30
z = do_calculations(z, max_z, rounds_z)

print(x + y + z)
```

Solution using a subprogram and a list

```
values = [5, 4, 6]
max_values = [100, 80, 115]
rounds = [20, 15, 30]

tot = 0
for i, val in enumerate(values):
    tot += do_calculations(val, max_values[i], rounds[i])

print(tot)
```

Solution: the Python Function

```
def do_calculations(n, max_n, rounds_n):
    for i in range(rounds_n):
        if n < max_n:
            n += n
        else:
            return n
    return n
```

Subprograms

In a python module is possible to define one or more subprograms (procedure or functions).

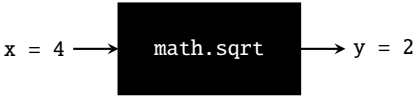
Similar to the build-in subprograms (available in the Standard Python Library), these subprograms are statements usable in the source code you are developing, but that can be customized based on your needs.

Subprograms as black boxes

A subprogram is a set of instructions that are isolated in order to:

- Ease their reuse
- Make the program more readable,
- Make the code easier to maintain

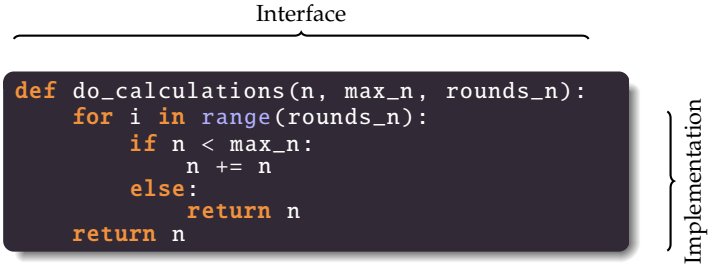
Within the code that uses it, a subprogram is seen as a black box because we are not interested in the internal implementation details, but rather in the effects it produces.



Subprograms as black boxes

Therefore, subprograms are equipped with:

- Interface for data exchange: it contains the description of the data used as input by the subprogram.
- Implementation: it contains the description of the local data and the algorithm executed by the subprogram.



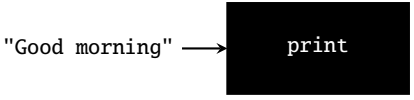
Black boxes rules

1. The interface of a black box must be simple, well defined and easy to understand.
2. To use a black box it is not necessary to know any detail about its implementation.
3. The developer of a black box must know nothing about the programs in which it could be used.

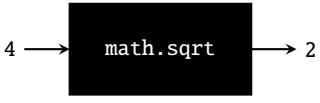
Subprograms: procedures and functions general definition

Generally, in programming, we usually divide subprograms in two categories:

- Procedure: subprogram that carry on operations using the input arguments (if any available) without returning an output value.



- Function: subprogram that carry on operations using the input arguments (if any available) returning an output value.



Subprograms: procedures and functions in Python

In Python there is no real difference between procedures and functions.

We will basically always speak about functions.

In fact, a subprogram that is not returning any value, what we would usually call procedure, returns in Python the None value by default.

```
>>> a = print("test")
test
>>> print(a)
None
```

Function declaration

A function is a block of instructions that is declared with a name and that can be called in any statements inside the code.

In Python, to define a function the following syntax is used:

```
def function_name(list_of_parameters):
    instructions
```

The instructions within the block of code are the body of the function.

The list of parameters represents the information that is exchanged with the caller code.

Function calls

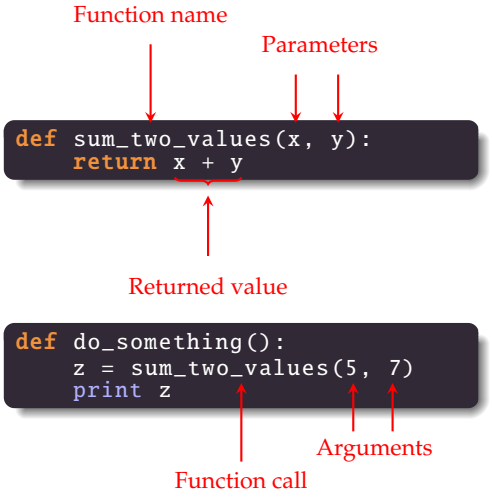
In Python, to call(invoke) a function, it is sufficient to use the name of the function, followed by the arguments of the call (list of values, comma separated, to assign to parameters).

```
function_name(list_of_parameters)
```

The value returned by a function can be used within an expression. It is for example possible to assign the value to a variable.

```
max = find_max(200, 438)
print("Max: " + str(max))
# or in another way
print("Max: " + str(find_max(200,438)))
```

Function calls example



Parameters

The parameters are used to specify the information that is exchanged between the function and the rest of the program.

The parameters define the names (identifiers) of the data that are passed to the function.

When you invoke (call) a function, you assign values to the parameters (which behave like variables). Consequently the values are transmitted and can be used within the function.

Formal parameters

The formal parameters, simply called parameters, are those used in the declaration of a function.

The list of parameters defines:

- the number
- the identifier (name)

The declaration of a formal parameters is similar to the declaration of a variable.

By default, the formal parameter is accessible, using the assigned identifier, only inside the function.

Actual parameters (arguments)

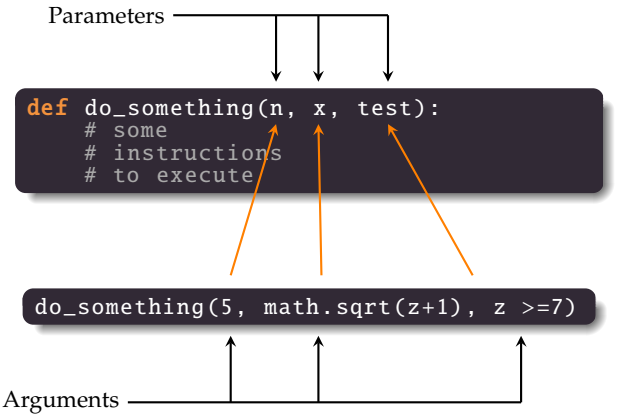
The actual parameters, also known as arguments, are those used at the moment of the call to a function.

The argument is a value. It can thus be:

- a literal (a number, a string)
- a variable identifier
- an expression that is evaluated. Expressions that contain further calls to other functions are included.

In Python, there are many ways to pass arguments to a function. Usually, one of the most basic possibility is to pass them in the same order as the formal parameters.

Actual and formal params: positional arguments



The arguments are mapped to the parameters based on position.

Positional arguments order: example

Let's take as example the following function:

```
def show_pet_description(animal_type, pet_name):
    print(f"Animal type: {animal_type}")
    print(f"Pet name: {pet_name}")
```

What output you expect from the following function call?

```
show_pet_description("Kratos", "Cat")
```

Actual and formal params: keyword arguments

Another way to pass parameters to a function in Python is by using keyword arguments.

Using keyword arguments it is possible to bind parameters by name instead of by position.

Let's define for example a function that takes 3 parameters, `a` and `b` and prints out the three values in the same order.

```
>>> def my_function(a, b, c):
...     print(a, b, c)

>>> my_function(b=1, c=3, a=5)
5 1 3
```

By passing for example `b=1` to the function call, we specify that the value 1 must be associated with the parameter with name `b`

Keyword arguments

When using keyword arguments to call a function, the order of the parameters is no more relevant.

Parameters are matched using the name and not the position.

If we swap arguments position in the call, the result will still be the same as the previous example.

```
>>> my_function(c=3, a=5, b=1)
5 1 3
```

Mixing keywords and positional arguments

Python let's you combine the two parameter passing approaches.

It is possible to pass some of the parameters by position and the others by keyword.

```
>>> my_function(5, c=3, b=1)
5 1 3
```

The positional parameters are read from left to right, before keywords are matched.

It is not possible to put a positional argument after a keyword argument.

```
>>> my_function(a=5, 1, 3)
File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

Argument defaults

Using defaults it is possible to set a default value to some of the parameters of our function.

```
>>> def my_function(a, b=2, c=3):
...     print(a,b,c)
```

When calling the function, the parameters without a default value are required, whereas the others are optional.

```
>>> my_function(1):
1 2 3
>>> my_function(a=1):
1 2 3
```

Argument defaults: override

When using defaults it is also possible to override the default values by passing in the arguments wanted when calling the function.

```
>>> my_function(1, 5):
1 5 3
>>> my_function(1, 4, 6):
1 4 6
```

It of course possible to choose which parameters we want to override. Let's say that we want to override only parameter `c` and use the default for `b`.

```
>>> my_function(1, c=5):
1 2 5
```

Arbitrary arguments

In Python there are also ways to allow functions to accept an arbitrary number of arguments by using the *args and **kwargs syntax.

This is reported here just for the sake of completeness and to let you know that it is something possible.

We will however not talk more about this way of treating arguments in the scope of this course.

Where to put functions?

For now, all the functions developed must be put in our python source code files, outside the `__main__` block.

The functions must be declared before the `__main__` block, where we will use them.

```
# functions definitions
def test():
    print("I am the test function")

def do_something():
    print("I am the do_something function")


if __name__ == '__main__':
    # functions call
    test()
    do_something()
```

Functions completion

A function completes when:

- all the internal statements are executed
- the `return` statement is executed
- an exception is thrown (we will discuss this later)

The `return` statement

The behavior of the `return` statement is similar to the one of the `break` statement that we discussed before.

In the case of the `return` statement, the execution of the function terminates in that point.

The return statement is used to retrieve a value from the function.

The return statement is not mandatory in a Python function. If a function lacks a return statement, it will automatically return the value `None`.

The return statement: example

```
def calculate_triangle_perimeter(side1, side2, side3):
    return side1 + side2 + side3

def calculate_triangle_area(base, height):
    return base * height / 2

def calculate_triangle_longest_side(side1,  side2, side3)
    :
    if side1 < 0 or side2 < 0 or side3 < 0:
        print("Unvalid triangle!")
        return

    if side1 > side2 and side1 > side3:
        return side1

    if side2 > side1 and side2 > side3:
        return side2

    return side3
```

Variable scope between blocks

Not all variables can be accessed from anywhere inside a program.

The portion of code where the variable is accessible is called `scope`. In Python there are 4 different variables scopes that follow the so called LEGB rule:

Local -> Enclosing -> Global -> Built-In

Local scope

A variable declared within a function has a local scope.

This means that it is accessible from the point where it is declared until the end of the containing function.

After the end of the function execution, the variable is removed from the memory and no more available.

```
def my_function():
    x = 42
    print(x)

my_function()
# prints 42

print(x)
# Results in an error, x does not exists anymore
```

Global scope

A variable that is declared outside any function has global scope.

It is accessible in all locations inside the file where it is defined as well as inside any file that imports it (see modules).

Example:

```
x = 42 # global scope

def my_function():
    print(x) # x is 42 inside the function

my_function()
print(x) # x is 42 outside the function
```

Global scope: modify global variable from a function

Although it is accessible also in functions, it is not possible to assign a new value to the global variable.

If we try to do it, we will end up in creating a new variable in the local scope of our function

```
x = 42 # global scope

def my_function():
    x = 50
    print(x) # x is 50 inside the function

my_function()
print(x) # x is still 42 outside the function
```

Global scope: `global` keyword

In order to modify the value of a global variable from within a function, we can use the `global` keyword.

Using this keyword we can avoid the creation of a variable in the local scope, thus referring to the global one.

```
x = 42 # global scope

def my_function():
    global x
    x = 50
    print(x) # x is 50 inside the function

my_function()
print(x) # x is also 50 outside the function
```

Enclosing scope

In Python it is possible to nest function declarations one inside the other.

If a variable is assigned in an enclosing scope, it is not local to the nested function.

That means that, it is not possible to access or assign values to that variable. If we try to do it, similar to the example of global variables, we will end up in creating a new variable in the local scope.

```
# enclosing function
def my_function_1():
    x = 42
    # nested function
    def my_function_2():
        x = 0
        print(x)     # x is 0
    my_function_2()
    print(x)     # x is still 42

my_function_1()
```

Enclosing scope: `nonlocal` keyword

If we want tell the interpreter that we want to refer to the variable in the enclosing scope, without creating a new local one, we need to use the `nonlocal` keyword.

After we define that a variable in our nested function is `nonlocal`, we can then access the variable and assign new values to it.

```
# enclosing function
def my_function_1():
    x = 42
    # nested function
    def my_function_2():
        nonlocal x
        x = 0
        print(x)     # x is 0
    my_function_2()
    print(x)     # x is also 0

my_function_1()
```

Built-in scope

The build-in scope is the largest scope. It contains all the special keywords that we can call from anywhere in our code without defining them before.

Keywords are special reserved words that are available for specific purposes. They cannot be used for other purposes.

Python keywords:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

LEGB rule

In Python, when we reference a variable identifier, the LEGB rule is followed in order to search the scopes in the following order:

1. **L**ocal scope
2. **E**nclosing scope
3. **G**lobal scope
4. **B**uilt-In scope

The search is ended at the first occurence found. If nothing is found, a `NameError` is raised.

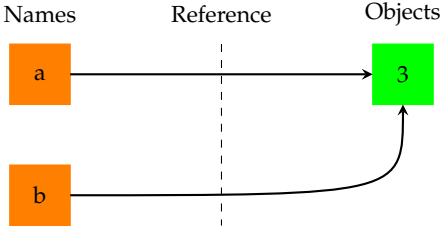
Variables memory management: shared reference

During this course, we've learned the basics about variables.

We want now to explore more in detail how they work.

Let's assume we have the following case:

```
>>> a = 3
>>> b = a
```

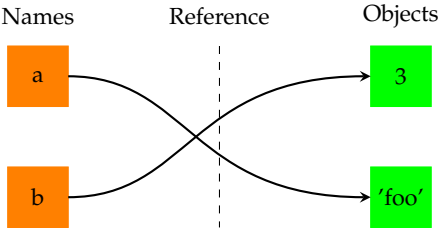


This scenario is usually called, shared reference. Multiple variable names are referencing the same object.

Variables memory management: shared reference

Let's add another line to our code:

```
>>> a = 3
>>> b = a
>>> a = 'foo'
```



With the new statement we simply created a new object 'foo' and made a reference the new object.

As we can see from the image, b is not changed still points to the object 3.

Shared reference and in In-Place Changes

The examples seen so far were based on `immutable` objects. When using `mutable` objects, as lists or dictionaries, there some other aspects to take into account.

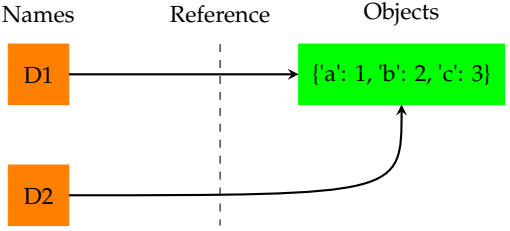
When we assign for example a value to a key of a dictionary, we are not generating a new object but we are changing the object itself in place.

This aspect is something you should always consider because it will otherwise be a source of unexpected behaviors.

Shared reference and in in-place changes

Let's see a practical example:

```
>>> D1 = {'a':1, 'b':2,'c':3}
>>> D2 = D1
```



As you already know, dictionaries and lists can be changed in place, simply by assigning a new value to a position or a key.

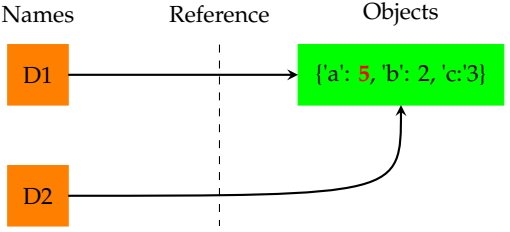
So what will happen if we try to change the value of a key of our dictionary, referenced by `D1`??

Shared reference and in in-place changes

```
>>> D1 = {'a':1, 'b':2, 'c':3}
>>> D2 = D1
>>> D1['a'] = 5

>>> D1
{'a': 5, 'b': 2, 'c': 3}

>>> D2
{'a': 5, 'b': 2, 'c': 3}
```



What we did here was not changing `D1` directly, we've changed a key of the dictionary object that is referenced by `D1`. This only impacts the content of the dictionary object.

Since `D2` is also referencing the same object as `D1`, as result, we see the new version of the dictionary also when printing the content of `D2`.

Shared reference and in in-place changes: deep copy

The same behavior applies if we try to do the same using another mutable type as a list.

This happens because by default python makes refereces when assigning a variable to another variable. If we want to avoid this, as already discussed when speaking about list copies, we should create a deep copy of our object.

To perform this task we can use the `deepcopy()` function from the `copy` module.

```
>>> import copy
>>> D1 = {'a':1, 'b':2, 'c':3}
>>> D2 = copy.deepcopy(D1)
>>> D1['a'] = 5

>>> D1
{'a': 5, 'b': 2, 'c': 3}

>>> D2
{'a': 1, 'b': 2, 'c': 3}
```

Shared references and equality

To verify equality in python we have basically 2 ways:

- using the == operator to check for `value` equality
- using the `is` operator to check for `reference` equality

```
>>> L = [1, 2, 3]
>>> M = L

>>> L == M
True
# same value

>>> L is M
True
# same object
```

```
>>> L = [1, 2, 3]
>>> M = [1, 2, 3]

>>> L == M
True
# same value

>>> L is M
False
# different object
```

Function arguments: value or reference?

Now that we know more about references, let's go back to function arguments.

In Python, arguments are passed to functions by assignment, meaning that:

- Mutable arguments are effectively passed "by reference".
- Immutable arguments are passed "by value"

Remember that: changing in place a mutable object that is passed as argument from within a function, also changes the object at the caller level.

```
a = [5, 7]
def f1(val):
    val[0] = 4

f1(a)
print(a)
# prints [4, 7]
```

Summary

- Code reuse
- Procedures and functions
- Functions declarations
- Functions parameters and arguments
- Functions usage
- The `return` statement
- Variable scopes
- Shared references
- Shared references and equality
- Functions arguments passage

Bibliography

- Learning Python 5th edition, Oreilly - Mark Lutz: Chapters 4, 16, 18
- Python Crash Course, no starch press - Eric Matthes: Chapters 8
- Python Official Documentation: https://docs.python.org/3/tutorial/
- LearnByExample: https://www.learnbyexample.org/python/