Introduction to functions with Python.

Mon 10 May 2021

Some of the predefined Python functions can be found here:

- https://docs.python.org/3/library/functions.html
- https://docs.python.org/3/library/math.html#module-math

What's happening if you want:

- Your system to do a specific task?
- reuse a set of instructions?
- modify the behaviour of a an existing function?

You define your own function.

Function definition:

```
# a function that does nothing
def foo(): # <--- signature
    pass   # <--- body
```

```
# a function that prints 4 times
def print4():
    msg = "Bonjour le monde"
    for _ in range(4):
        print(msg)
```

```
# a function that does not return anything, it prints "hello world" message
def bar():
    print("hello world")
```

```
# a function that returns the integer 42
def bar():
    return 42
```

Parameters:

```
# a function that takes 3 positional parameters or required parameters
# and returns a sum of those parameters
def baz(a, b, c):
    s = a + b + c
    return s
```

```
# a function that takes 3 positional parameters: a, b, c
# and 2 optional parameters: d, e
def qux(a, b, c, d="foo", e=42):
    print(d, e)
    s = a + c + b
    return s
```

```
# keep in mind, positional parameters
# always come before optional parameters
>>> def qux( d="foo", e=42, a, b, c):
...     print(d, e)
...     s = a + c + b
...     return s
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
>>>
```

First class citizen:

In Python, functions are said to be **first class citizens**. Because, functions can be:

- stored inside variables, lists, dictionaries, tuples, sets.
- passed as arguments to others functions.
- defined within others functions.
- returned from others functions.

Examples:

```
>>> # storing function inside a variable or a list
>>>
>>> def triple(x):
...     return x*3
...
>>>
>>> foo = triple
>>> foo(2)
4
>>>
>>> operations = [triple, print, type]
>>> operations[0](2)
4
>>> operations[1](2)
2
>>> operations[2](2)
<class 'int'>
>>>
```

```
>>> # passing a function as argument to another function
>>>
>>> lst = list(range(10))
>>>
>>> lst2 = list(map(triple, lst))
>>> print(lst2)
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

```
>>> # a function defined inside another function and returned from that function
>>>
>>> def power_generator(num):
...     # inner function definition
...     def power_n(power):
...         return num ** power
...
...     return power_n
...
>>> power_two = power_generator(2)
>>> print(power_two(8))
256
```

Anonymous functions:

Lambda functions are single-expression functions that are not bound to a name. The return statement is implicit. Examples:

```
>>> (lambda x: x + 1)(2)
>>>
>>> # ou encore
>>>
>>> foo = lambda x, y: x*y # stockage d'une fonction dans une variable
>>> foo(2,2)
4
```

```
>>> full_name = lambda first, last, age: f'Hello, my name is {first.title()} {last.title()} and I am
>>> full_name("Patrick", "Nsukami", 42)
'Hello, my name is Patrick Nsukami and I am 42 years old.'
>>>
>>>
>>> def full_name(first, last, age):
...     return f'Hello, my name is {first.title()} {last.title()} and I am {age} years old.'
...
>>> full_name("Patrick", "Nsukami", 42)
'Hello, my name is Patrick Nsukami and I am 42 years old.'
>>>
```

Use cases for lambda functions:

Most of the time, you'll define and use Lambda functions with the following functions: filter, map, reduce, sort, sorted, min, max.

Passing a varying number of arguments to a function:

To pass an unspecified number of arguments to your functions, you can use 2 special symbols: *args and **kwargs.

```
>>> # a function that takes an infinite number of arguments
>>> # and returns the sum of all the passed arguments
>>> def plus(*args):
...     print(args)
...     return sum(args)
...
>>>
>>> plus(1, 2, 3)
(1, 2, 3)
6
>>> plus(1, 2, 3, 6, 3, 1, 90)
(1, 2, 3, 6, 3, 1, 90)
106
>>>
```

```
>>> # a function that takes an infinite number of keyword arguments
>>> # and returns the sum of all the passed keyword arguments
>>> def foo(**kwargs):
...     print(kwargs) # notice, kwargs is a dictionary
...     return sum(v for v in kwargs.values())
...
>>>
>>> foo(a=4, b=5)
{'a': 4, 'b': 5}
9
>>> foo(a=4, b=5, c=6, e=10)
{'a': 4, 'b': 5, 'c': 6, 'e': 10}
25
>>>
```

Order is important:

You can't write the following, *args always comes before **kwargs:

```
>>> def foo(**kwargs, *args):
  File "<stdin>", line 1
    def foo(**kwargs, *args):
                      ^
SyntaxError: invalid syntax
>>>
```

You can't write the following, positional arguments comes before *args:

```
>>> def foo(*args, a, b, c):  # positional args always before *args
...     print(args)
...     return a+b+c
...
>>> foo(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() missing 3 required keyword-only arguments: 'a', 'b', and 'c'
>>>
>>> foo(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() missing 3 required keyword-only arguments: 'a', 'b', and 'c'
>>>
```

Recursive functions:

In Python, a function can call other functions. It is also possible for a function to call itself. A function is said to be **recursive** when that function calls itself during its execution. A recursive function will continue to call itself until some condition is met to return a result.

Example of recursion in real life:

- place two parallel mirrors facing each other. Any object in between them would be reflected recursively.
- paper sizes and formats: A4, A3, A2, ...
- the Romanesco broccoli
- the Mandelbrot set

Recursion is a way for you to find the solution to a complex problem, by using/combining solutions to smaller/simpler problems. Recursion is useful when you know a "a trivial case/solution" to the initial problem.

Exemple gcd:

```
def gcd_recursive(a,b):
    """ gcd_recursive(a,b): return greatest common divisor between 2 integers a and b. """

    if b==0: # base condition that will stop the recursion
        return a # trivial solution

    else:
        return gcd_recursive(b,a%b)
```

Be careful:

When you write a function in its recursive form, you must think about the base condition that stops the recursion. You don't want the function calls itself infinitely. To make sure, infinite recursions are avoided, the Python interpreter limits the depths of recursion. Example:

```
>>> def rec():
...     rec()
...
>>> rec()
>>> rec()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in rec
  File "<stdin>", line 2, in rec
  File "<stdin>", line 2, in rec
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
>>>
```

NB:

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time*.
3. Recursive functions are harder to debug.
4. A recursive function can be written in an iterative form.

Iterative functions:

You'll recognize a function in its iterative form when:

- there is a loop
- we know exactly how many iterations we will do
- within the loop, there are instructions that help find the final result
- a variable is used to store a result after each iteration

Example, gcd, in its iterative form:

```
def gcd_iterative(a, b):
    """ gcd_iterative(a,b): return greatest common divisor between 2 integers a and b. """

    if a > b:
        smallest = b
    else:
        smallest = a

    for i in range(1, smallest+1):  # the loop / we know the when to start, when to stop, and the st
        if((a % i == 0) and (b % i == 0)):  # instructions to find the final result
            gcd = i

    return gcd
```

Docstrings:

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. It is strongly recommended to put docstrings inside all your functions. The convention asks us to always use """triple double quotes""" around docstrings. Examples:

```
>>> # a function without docstring
>>> def avg(*args):
...     return sum(args) / len(args)
...
>>>
>>> help(avg) # no help/documentation available for that function
Help on function avg in module __main__:

avg(*args)
(END)
```

2 types of docstrings, one line & multi lines:

One-line docstring are for really obvious cases. They should really fit on one line. For example:

```
>>> # One line docstring
>>>
>>> def avg(*args):
...     """Calculate and return average of passed arguments."""
...     return sum(arg)/len(args)
...
>>> help(avg)
Help on function avg in module __main__:

avg(*args)
    Calculate and return average of passed arguments.
(END)
```

Multi-line docstrings consist of a summary line, followed by a blank line, followed by a more elaborate description. To see an example of a multiline docstring, please, go check the following link: from line 102, to line 251.