Python 101

Functions in Python

Contents

1. An introduction to the functions
2. Various types of arguments in functions
3. Builtin functions
4. DocStrings
5. The Scope of variables
6. Lambda functions
7. Generator functions
8. Interview Questions

An introduction to functions

In a mathematical sense function is something takes a value and produces corresponding value.

The same is true for programming as well. But in mathematics where we use some mathematical expressions to define a function in Programming we use code ie. set of instructions to define what a Programming function does.

Based on the second point we can also say that functions in programming are a way to bind together a bunch of instructions. So that when we call that function all of those bunch of instructions will get executed.

An introduction to functions contd..

In Python we declare that some identifier is a function by suffixing a keyword **def** before it. Like following-

```
def FUNCTION_NAME(FUNCTION_ARGUMENTS):

    STATEMENTS_TO_EXECUTE (ie. a code_block)
```

Try out the following code on the interpreter

```
>>>  def function_name(function_argument):
. . . .       print('I got executed')
. . . .       print('whoever called it passed this value to the function = ', function_argument)
>>> function_name('abcd')
I got executed
whoever called it passed this value to the function = abcd
>>> function_name(123)
I got executed
whoever called it passed this value to the function = 123
```

###

Mind that irrespective of what Type of value we will pass to function we don't need to do anything in function argument as python is dynamically typed.

Various Types of Arguments in Functions

We saw in last example the argument was mandatory to be given.

But what if user didn't want to pass value to the function. In that case we use a default value for the argument if user didn't pass any. Which makes them **optional arguments.**

```
def function_name(argument=default_value):
```

Something to remember is that the optional arguments will always come after the mandatory arguments.

```
def function_name(mandatory_args, optional_args=default_values):
```

Try out the following code on the interpreter

```
>>>  def greet_user(user, do_greet=True):
. . . .       if do_greet:
. . . .             print('Hello', user)
. . . .       else:
. . . .             print("Bye", user)
>>> greet_user("Guido") ← mind that we didnt need passing value of do_greet
Hello Guido
>>> greet_user("Guido", False)
Bye Guido
```

WARNING (PSF tutorial) - if the optional arguments value is mutable eg. List, then as the default values are evaluated only once. It will start carrying values on changes to the optional argument.

```
>>> def warning(i, L=[]):
. . . .   L.append(i)
. . . .   print(L)
>>> warning(1)
[1]
>>> warning(2)
[1, 2] ← we would have expected it to print [2] but it carried the last appended value as L=[] didnt get initialized on second call.
```

Various Types of Arguments in Functions

contd..

When we call a function we usually call it like this, In this method we are considering the “**positional arguments**” as we are mapping values to the position of arguments in the function signature.

```
function(value1, value2, value3)
```

But what if there are too many arguments? Then keeping track of position becomes cumbersome. Thus we use the corresponding argument’s name. ie. kwarg (keyword arguments)

```
function(arg1=value1, arg2=value2, arg3=value3)
```

Try out the following code on the interpreter

```
>>>  def greet_user(arg1, arg2, arg3):
. . . .       print(arg1, arg2, arg3)
>>> greet_user(1,2,3)
1 2 3
>>> greet_user(arg2=2, arg3=3, arg1=1)
1 2 3

# We can call it using a mix of positional and keyword arguments as well.

>>> greet_user(1, arg3=3, arg2=2)
1 2 3

Mind that we might pass a single value twice in this case as well.
>>> greet_user(1, arg1=1, arg2=2)

TypeError: greet_user got multiple values for argument 'arg1'


# This will raise error that multiple values have been passed for 1 argument that is arg1
```

Various Types of Arguments in Functions

contd..

All examples till yet had knowledge about the number of arguments we want from user. But what if we can not determine the number of arguments the user will give?

In this case we use **Arbitrary Args and Kwargs.**

These are based on **unpacking and packing** principles.

Try out the following code on the interpreter

```
>>>  def arbitrary_test(arg1, arg2='abc', *objects): ← *objects converts whatever is there into a list
. . . .        print("objects has values", objects)
. . . .        print("arg1 value is", arg1)
. . . .        print("arg2 vaue is", arg2)

>>> arbitrary_test(1,2,3,4,5,6)
(3,4,5,6)

# Here it is a positional argument again, because the variable name is not passed. If we want arbitrary number of
named arguments we can do the following

>>> def arbitrary_test(arg1, arg2.='abc', **objects):
. . . .        print(objects)
>>> arbitrary_test(1,2, arg1=1, arg2=2, arg3=3)
{'arg1':  1, 'arg2': 2, 'arg3': 3}

# * essentially unpacks the tuple which we saw in last lecture without the need of having
# ** essentially unpacks the dictionaries (we will study about dictionaries in Data Structure in Python (Part-2))
```

Builtin Functions

Built in functions are those functions that we do not need to import from anywhere in order to use in Python.

We can see a list of all the builtins using

```
>>> dir(__builtins__)
```

DocString

Docstring is a string given right underneath the function header to give information to the user who will use our function using “help”.

PEP 257 Gives its standards.

It can be used for easy Documentation of our program if we declare docstrings properly. Thus it is called Documentation - String = DocString.

Try out the following code on the interpreter

```
>>>  def docstring_test(arg1, arg2='abc', *objects):
. . . .      " " " The work that function does eg. this function does nothing
. . . .
. . . .      Keyword Arguments:
. . . .
. . . .      arg1 - - This is the first argument
. . . .      arg2 - - This is the second argument
             " " "
. . . .      pass
>>> help(docstring_test)
```

The Scope of Variables

There are only two scopes generally speaking - 1. local 2. global .

If at runtime we want to see which variables are there in the local scope we can use function **locals()** similarly in order to see the global variables we can use function **globals()**

The scope of a variable is always in the codeblock and in child-codeblocks wherever the variable is defined.

Try out the following code on the interpreter

```
>>> x = 1
>>> locals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins'
(built-in)>, 'x': 1}
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins'
(built-in)>, 'x': 1}
>>>  def scope_test():
            y = 2
. . . .     print(locals())
. . . .     print(globals())
>>> scope_test()
{'y': 2}
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins'
(built-in)>, 'x': 1, 'scope_test': <function scope_test at 0x7f2f057aab70>}
```

Lambda Functions

These are single expression functions which are almost completely analogous to mathematical functions.

eg. f(x,y) = x+y can be written in lambda form as-

lambda x,y : x+y ← mind that as it is in one line there is no code block needed.

Lambda functions are always assigned to a variable and then are called via that variable.

Try out the following code on the interpreter

```
>>> lambda_function = lambda x : x**2
>>> lambda_function(2)
4
>>> lambda_function(3)
9
```

Generator functions and problem with normal iterables

Let's assume this case. We need to create a function that returns the user a list that contains numbers from 0 to n-1 if user passes n as the function argument.

Our approach will be creating a list obviously and then appending 0 to n-1 in the list and returning the list to whoever called the function. But think for very very large or indefinite values of n. The space required for it will be very large because we are returning all of the values at once. This is a basic approach using a normal iterable like a list. But what if we could do something like this-

In order to save space and computation power we can let the caller call the function each time it needs next value keeping track of what last value was returned to the caller so that we can return the next value in sequence.

Generator functions and problem with normal iterables contd..

The basic idea here is that we are NOT **returning** the value from the function instead we are **yielding** new results each time the function is called.

It makes sense for some cases as well where we need large evaluated sequences. Because in those sequences only one value is needed at one instant.

Lets understand using an example of a fibonacci series.

Try out the following code on the interpreter

```
>>>  def fibonacci():
. . . .      prev, curr = 0, 1
. . . .      while True:
. . . .           yield curr
. . . .           prev, curr = curr, prev + curr
>>> fibonacci()
<generator object fibonacci at 0x7fd25c1eaca8> ← It didn't return any value upon calling the function. It just returned some generator. Now we know that we can call next on this generator to get the next value in sequence.

>>> fibonacci_generator = fibonacci()
>>> next(finboacci_generator)
1
>>> next(finboacci_generator)
1
>>> next(finboacci_generator)
2
```

Generator functions and problem with normal iterables contd..

How could the generator possibly work?

It’s obvious that it is keeping some sort of state inside it. Where state means the last evaluation that it has performed so that it can perform the next evaluation.

Generators are essentially a simplified form of iterators. We will discuss iterators after we have finished OOP in Python. But iterators are not used anymore because Generators provide same functionality that iterators do in a simple way.

Interview Questions

What are types of arguments in Python functions?

From declaration perspective-

1. Default arguments
2. Optional arguments

From calling perspective-

1. Keyword arguments
2. Positional arguments

How many times is optional arguments assigned it’s default value?

Only the first time a function is called in life time of a program.

Why do we use Generators?

In order to avoid memory exhaustion by very large return values from program.

What are lambda functions?

Single line functions, with single expression body. These are mostly used as an argument to other full functions.

How can you classify Python’s function call? Call by value or Call by reference?

Neither.

Python uses call by object reference.

https://github.com/ash2shukla/PythonFunctionCalling/blob/master/How%20Function%20Call%20works%20in%20python.ipynb

Can you show disassembly of the code of a function in python?

Yes it is possible to do so using **dis** module.

```
>>> def some_function():
. . . .    pass
>>> import dis
>>> dis.disassemble(some_function.__code__)
```

← Here __code__ is the attribute of every function that contains information about the body of the function.

Name a few builtin functions you remember.

range, sum, len, print, input etc.

Is it essential to return a value from a function?

No its not essential for us to program our function such that it will return a value.

But implicitly it always returns a value. When we do not return anything from a function then **None** is returned. ← Here is **None** is the **literal** which is equivalent to NULL in C/C++.