**Functions**

miércoles, 5 de abril de 2023

12:23 p. m.

Roadmap.io

FreeCodeCamp: <https://www.freecodecamp.org/news/python-functions-define-and-call-a-function/>

W3Schools: <https://www.w3schools.com/python/python_functions.asp>

Geeksforgeeks: <https://www.geeksforgeeks.org/python-functions/>

Python Built-In functions library: <https://docs.python.org/3/library/functions.html>

Functions

Corey Schafer Tutorial: [Python Tutorial for Beginners 8: Functions](https://www.youtube.com/watch?v=9Os0o3wzS_I&list=PL-osiE80TeTskrapNbzXhwoFUiLCjGgY7&index=9)

Functions are block of code that is supposed to be reusable and more modular. In order to better understand functions, is possible to focus on what's the input and what is returning from it.

The function also take the characteristics of what's returning, for instance, let say we have the following function:

def hello\_func():

    print('Hello Function')

If executed, it will print out a sting

hello\_func()

Console: Hello Function

But if modified returning just the string, the function when called also inherit string's methods

def hello\_func():

    return 'Hello Function'

print(hello\_func().upper())

Console: HELLO FUNCTION

The 'return' reserved word

Whenever is needed to break out of a function execution, the return reserved word works as an end point.

Arguments / Parameters

This are the types of arguments that a python function can receive: Positional, Arbitrary, Keyword Arguments, Arbitrary Keyword, Default parameter value.

Positional

The most common, where the position defined and received matters

def info(name, age):

    print(f'Hi, my name is {name} and I am {age} years old')

info('Tom', 24)

Console: Hi, my name is Tom and I am 24 years old.

The function will expect to receive name and age in that order, if otherwise, it will throw an error.

def info(name, age=24):

    print(f'Hi, my name is {name} and I am {age} years old')

info(age=25, 'Tom')

Console:

File "C:\Users\USUARIO\GR\Software Development\Learning\Python\Functions\Functions.py", line 35

info(age=25, 'Tom')

^

SyntaxError: positional argument follows keyword argument

Arbitrary (\*Args)

This refers to that any number of arguments could be passed as an iterable to be processed into the function.

The arguments are treated like an iterable, so it could be loop through them

def my\_func(\*args):

    for i in args:

        print(i)

my\_func(1, 'a', (None, None), {1,3,5})

Console:

1

a

(None, None)

{1, 3, 5}

Keyword & Default Arguments (Kwargs)

There are arguments that could be stated in key, so you will have to name them directly into the function, and if there is going to be both positional arguments and keyword arguments in the same function, positional always have to go first otherwise the interpreter won't take the function.

def info(age=24, name):

    print(f'Hi, my name is {name} and I am {age} years old')

info(age=25, 'Tom')

Console: SyntaxError: non-default argument follows default argument

Keyword arguments are those that are need to be specified by name when passing it to a function:

def my\_func(greeting=str, name=str, city=str):

    print(f'{greeting} {name} from {city}!')

my\_func(greeting='Hello', name='Tom', city='Orlando')

Console: Hello Tom from Orlando!

But could also be determined as default at the definition of the function.

def my\_func(greeting='Hi', name='Jerry', city='NY'):

    print(f'{greeting} {name} from {city}!')

my\_func()

Console: Hi Jerry from NY!

And that way become optional, meaning that if no specified in a function call it will preserve the preset values.

my\_func(greeting='Hello')

Console: Hello Jerry from NY!

Arbitrary Keyword Arguments (\*\*Kwargs)

By extension, \*\*Kwargs are the same as \*Args but with Keyword arguments, meaning that by calling \*\*Kwargs into a function, the interpreter will get that you are going to pass whatever number of arguments but keyworded.

The\*\*Kwargs are treated just like a dictionary, so the syntax goes like this

def my\_func(\*\*kwargs):

    for k, v in kwargs.items():

        print(f'"{k}" is the key and "{v}" is the value')

my\_func( shoes = 'Adidas', shirt = 'H&M', socks = 'Nike')

Console:

"shoes" is the key and "Adidas" is the value

"shirt" is the key and "H&M" is the value

"socks" is the key and "Nike" is the value

And is also possible to combine Kwarg with \*\*Kwargs

def my\_func(greeting='hi', name = "Ken", \*\*kwargs):

    for k, v in kwargs.items():

        print(f'{greeting} {name}, your {k} are from the "{v}" brand')

my\_func( shoes = 'Adidas', shirt = 'H&M', socks = 'Nike')

Console:

hi Ken, your shoes are from the "Adidas" brand

hi Ken, your shirt are from the "H&M" brand

hi Ken, your socks are from the "Nike" brand

Functions with all kinds of arguments ( Args, \*Args, Kwargs, \*\*Kwargs)

And finally, a function at it core could receive anything actually, so is just matter to clearly define how the arguments will be passed.

def my\_func(name, lastname, \*args, greeting=str, treatment = "Mr.", \*\*kwargs):

    print(f'''{greeting}, {treatment} {name} {lastname}!

    At the moment you posses {[x for x in args]} in your account

    and { {k:v for k,v in kwargs.items()} } in your stock

    ''')

account = ['$1000 COP', '$2500 USD', '$51000 RBL']

stock = {'horses' : 4, 'boats' : 10, 'cars' : 1}

my\_func('Thomas', 'Shelby', \*account , greeting='Good Evening', \*\*stock)

Console:

Good Evening, Mr. Thomas Shelby!

At the moment you posses ['$1000 COP', '$2500 USD', '$51000 RBL'] in your account

and {'horses': 4, 'boats': 10, 'cars': 1} in your stock

One last thing is that arbitrary arguments \*Args or arbitrary keyworded arguments \*\*kwargs does not necessarily need to be named that way, is just a convention it could be name anything but while have the preceding \* or \*\* the function gets its purpose

def my\_func(\*kids, \*\*courses\_grades):

    for kid in kids:

        for c, g in courses\_grades.items():

            print(f'{kid} - {c}: {g}')

my\_func('Tom', 'Susan', 'Nate', 'Sarah', spanish = 5, compsci = 9, math = 10 )

Console:

Tom - spanish: 5

Tom - compsci: 9

Tom - math: 10

Susan - spanish: 5

Susan - compsci: 9

Susan - math: 10

Nate - spanish: 5

Nate - compsci: 9

Nate - math: 10

Sarah - spanish: 5

Sarah - compsci: 9

Sarah - math: 10

Remember this!

Docstring 
The first string after the function is called the Document string or Docstrjng in short. This is used to 
describe the functionality of the function. The use of docstring in functions is optional but it is 
considered a good practice. 
The below syntax can be used to print out the docstring ot a function: 
Syntax; print (function_name. _doc_) 
Example: Adding Docstring to the function 
Python3 
A simple Python function to Check 
evenOdd(x ) ; 
•Function to check if the number is even or odd" 
print ("even") 
else: 
print ("odd") 
Driver code to call the function 
print (evenOdd _doc_) 

Variables

In terms of variables, there are three types on the functions context: Globals, Locals and Nonlocals.

Global

Variables declared outside of the function or in global scope.

This means that this variables could be accessed inside or outside of the function.

x = 'Global'

def func():

    global x

    y = 'Local'

    x = x \* 2

    print(x)

    print(y)

print('Global x =', x)

func()

print('Global x =', x)

Console:

Global x = Global

GlobalGlobal

Local

Global x = GlobalGlobal

What this code shows is that the variables, when within a function with a preceding 'global' affects the function outside of the function with whatever made within it.

Locals

Variables declared inside the function.

And even when named the same as a global variable, without the preceding reserved word, it won't affect the global variable

x = 5

def func():

   x = 10

   print('Local x =', x)

print('Global x =', x)

func()

print('Global x =', x)

Console:

Global x = 5

Local x = 10

Global x = 5

Non Locals

Variables used in nested functions whose local scope is not defined.

This means the variable could be in neither local or global scopes.

def outer():

    x = 'local'

    def inner():

        nonlocal x

        x = 'nonlocal'

        print('inner: ', x)

    inner()

    print('outer: ', x)

outer()

Console:

inner : nonlocal

outer : nonlocal

With the use of the reserved word "nonlocal" the variable out of inner is being affected during the call of the inner function. But without it, the variable never varies.

def outer():

    x = 'local'

    def inner():

        # nonlocal x

        x = 'nonlocal'

        print('inner: ', x)

    inner()

    print('outer: ', x)

outer()

Console:

inner : nonlocal

outer : local