08a-functions

September 16, 2022

1 Functions

- function is a sequence of instructions that perform a task, bundled as a unit.
- This unit can then be imported and used wherever it's needed.
- A function is a block of instructions, packaged as a whole, like a box. Functions can accept input arguments and produce output values. Both of these are optional.

Functions are among the most important concepts and constructs of any language, so let me give you a few reasons why we need them: - They **reduce code duplication** in a program. By having a specific task taken care of by a nice block of packaged code that we can import and call whenever we want, we don't need to duplicate its implementation. - They help in **splitting a complex task** or procedure into smaller blocks, each of which becomes a function. - They **hide the implementation** details from their users. - They improve **traceability** - They improve **readability**

The simpliest function would be somethink like the following (no argument, returns anything)

[1]:
```
def func():
    pass
```

Function can receive 0, 1 or multiple alguments and return 0, 1 or multiple values

[2]:
```
def my_function(name):
    """one argument function"""
    print(f'my_function: Hello {name}')

def my_function2(name, age):
    """two argument function"""
    print(f'my_function2: Hello {name}')
    return len(name), 2022 - age

my_function('Peter')

n, y = my_function2('Peter', 21)

print(f"your name has {n} letters")
print(f"Were you born in {y}?")
```

```
my_function: Hello Peter
my_function2: Hello Peter
```

```
your name has 5 letters
Were you born in 2001?
```

[3]:
```
help(my_function2)
```

```
Help on function my_function2 in module __main__:

my_function2(name, age)
    two argument function
```

1.1 Variables scopes

The scope of one variable refers to where it is visible to the code, and its value

[4]:
```
def outer():
    def inner():        # a function inside a function!? yup!
        test = 2        # inner scope
        print('inner:', test)

    test = 1            # outer scope
    inner()
    print('outer:', test)

test = 0                # global scope
outer()
print('global:', test)
inner()                 # error! function not defined in this scope
```

```
inner: 2
outer: 1
global: 0
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [4], line 13
     11 outer()
     12 print('global:', test)
---> 13 inner()

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

1.1.1 global

- The `global` statement causes the listed identifiers to refer to the global scope

[5]:
```
m = 5
m_g = 5
```

```
def local():
    global m_g
    m = 7
    m_g = 7
    print(f"m inside local(): {m}")
    print(f"m_g inside local(): {m_g}")

print(f"m before calling local(): {m}")
print(f"m_g before calling local(): {m_g}")

local()

print(f"m after calling local(): {m}")
print(f"m_g after calling local(): {m_g}")
```

```
m before calling local(): 5
m_g before calling local(): 5
m inside local(): 7
m_g inside local(): 7
m after calling local(): 5
m_g after calling local(): 7
```

In a more "deep construction" the global also works

[6]:
```
def outer():
    test = 1 # outer scope

    def inner():
        global test         # <------- global
        test = 2
        print('inner:', test)

    inner()
    print('outer:', test)

test = 0 # global scope
outer()
print('global:', test)
```

```
inner: 2
outer: 1
global: 2
```

1.1.2 nonlocal

- The `nonlocal` statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals

[7]:
```
def outer():
    test = 1 # outer scope

    def inner():
        nonlocal test          # <------- nonlocal
        test = 2
        print('inner:', test)

    inner()
    print('outer:', test)

test = 0 # global scope
outer()
print('global:', test)
```

```
inner: 2
outer: 2
global: 0
```

1.2 Parameters

- Argument passing (parameters) is nothing more than assigning an object to a local variable name
- Assigning an object to an argument name inside a function doesn't affect the caller
- Changing a mutable object argument in a function affects the caller

[8]:
```
x = 3

def func(x):
    print(x)
    x = 10          # defining a local x, not changing the global one

func(x)
print(x)
```

```
3
3
```

[9]:
```
x = [1, 2, 3]

def func2(x_in):
    print(x_in)
    x_in[1] = 42    # this affects the caller, because there is a mutable␣
 ↪object argument!

func2(x)
print(x)
```

```
[1, 2, 3]
```

```
[1, 42, 3]
```

Assigning an object to an argument name within a function doesn't affect the caller

[10]:
```
x = [1, 2, 3]

def func3(x):
    x[1] = 42                # this changes the caller!
    x = 'something else'     # this points x to a new string object

func3(x)
print(x)                     # still prints: [1, 42, 3]
```

```
[1, 42, 3]
```

1.2.1 Positional argument

- **Positional arguments** are read from left to right and they are the most common type of arguments
- **Keyword arguments** are assigned by keyword using the `name=value` syntax
- The counterpart of keyword arguments, on the definition side, is **default values**. The syntax is the same, `name=value`, and allows us to not have to provide an argument if we are happy with the given default. You cannot specify a default argument on the left of a positional one

[11]:
```
def func(a, b, c):
    print(a, b, c)
```

[12]:
```
func(1, 2, 3)
```

```
1 2 3
```

[13]:
```
func(a=1, c=2, b=3)
```

```
1 3 2
```

[14]:
```
func(1, c=2, b=3)
```

```
1 3 2
```

with default values for the arguments

[15]:
```
def func2(a, b=4, c=88):
    print(a, b, c)

func2(1)                  # prints: 1 4 88
func2(b=5, a=7, c=9)      # prints: 7 5 9
func2(42, c=9)            # prints: 42 4 9
```

```
1 4 88
7 5 9
42 4 9
```

[16]:
```
func2(b=1, c=2, 42)      # SyntaxError: non-keyword arg after keyword arg
```

```
  Cell In [16], line 1
    func2(b=1, c=2, 42)      # SyntaxError: non-keyword arg after keyword arg
                       ^
SyntaxError: positional argument follows keyword argument
```

1.2.2 Variable positional arguments (optional)

Sometimes you may want to pass a non fixed number of positional arguments to a function and Python provides you with the ability to do it

[17]:
```
def minimum(*n):
    print(n)             # n is a tuple
    if n:                # n <> None
        mn = n[0]
        for value in n[1:]:
            if value < mn:
                mn = value
        print(mn)
```

[18]:
```
minimum(1, 3, -7, 9, 10)
```

```
(1, 3, -7, 9, 10)
-7
```

[19]:
```
minimum()
```

```
()
```

1.2.3 unpacking

Considering the simple function

[20]:
```
values = (1, 3, -7, 9)

def func(*args):
    print(args)
```

The following call is equivalent to: `func((1, 3, -7, 9))`

[21]:
```
func(values)
```

```
((1, 3, -7, 9),)
```

but, the following is equivalent to: func(1, 3, -7, 9)

[22]:
```
func(*values)
```

```
(1, 3, -7, 9)
```

In the first one, we call `func` with one argument, which in the case is a four elements tuple. In the second example, by using the `*` syntax, we're doing something called **unpacking**, which means that the four elements tuple is unpacked, and the function is called with four arguments: 1, 3, -7, 9.

1.2.4 Variable keyword arguments

Variable keyword arguments are very similar to variable positional arguments. The only difference is the syntax (`**` instead of `*`) and that they are collected in a dictionary.

[23]:
```
def func(**kwargs):
    print(kwargs)
```

The following calls are equivalent

[24]:
```
func(a=1, b=42)
```

```
{'a': 1, 'b': 42}
```

[25]:
```
func(**{'a': 1, 'b': 42})
```

```
{'a': 1, 'b': 42}
```

[26]:
```
func(**dict(a=1, b=42))
```

```
{'a': 1, 'b': 42}
```

Mixing it all, we can combine input parameters, as long as you follow these ordering rules:

- when defining a function, normal positional arguments come first (`name`), then any default arguments (`name=value`), then the variable positional arguments (`*name`, or simply `*`), then any keyword-only arguments (either `name` or `name=value` form is good), then any variable keyword arguments (`**name`).
- On the other hand, when calling a function, arguments must be given in the following order: positional arguments first (`value`), then any combination of keyword arguments (`name=value`), variable positional arguments (`*name`), then variable keyword arguments (`**name`)

[27]:
```
def func(a, b, c=7, *args, **kwargs):
    print(10 * '-')
    print('a, b, c:', a, b, c)
    print('args:', args)
    print('kwargs:', kwargs)

func(1, 2, 3, *(5, 7, 9), **{'A': 'a', 'B': 'b'})
```

```
----------
a, b, c: 1 2 3
args: (5, 7, 9)
kwargs: {'A': 'a', 'B': 'b'}
```

[28]:
```
func(1, 2, 3, 5, 7, 9, A='a', B='b')  # same as previous one
```

```
----------
a, b, c: 1 2 3
args: (5, 7, 9)
kwargs: {'A': 'a', 'B': 'b'}
```

and another sometimes useful example

[29]:
```
def connect(**options):
    conn_params = {
        'host': options.get('host', '127.0.0.1'),
        'port': options.get('port', 5432),
        'user': options.get('user', ''),
        'pwd': options.get('pwd', ''),
    }
    print(conn_params)
    # we then connect to the db (commented out)
    # db.connect(**conn_params)
    # ...

connect()
connect(host='127.0.0.42', port=5433)
connect(port=5431, user='fab', pwd='gandalf')
```

```
{'host': '127.0.0.1', 'port': 5432, 'user': '', 'pwd': ''}
{'host': '127.0.0.42', 'port': 5433, 'user': '', 'pwd': ''}
{'host': '127.0.0.1', 'port': 5431, 'user': 'fab', 'pwd': 'gandalf'}
```

1.3 Return of values.

In Python, you can return a tuple, and this implies that you can return whatever you want

[30]:
```
def factorial(n):
    if n == 0:
        return 1

    fact = 1
    while n > 1:
        fact *= n
        n = n - 1
    return fact


factorial(5)
```

[30]: 120

Return multiple values

[31]:
```
def moddiv(a, b):
    return a // b, a % b

print(moddiv(20, 7)) # prints (2, 6)
```

```
(2, 6)
```

which can be received in two variables

[32]:
```
d, r = moddiv(20, 7)
```

[33]:
```
d
```

[33]: 2

[34]:
```
r
```

[34]: 6

or in a tuple

[35]:
```
t = moddiv(20, 7)
t
```

[35]: (2, 6)

1.4 Anonymous function - lambda (optional)

- Anonymous functions are called lambdas in Python, and are usually used when a fully-fledged function with its own name would be overkill, and all we want is a quick, simple one-liner that does the job
- Defining a lambda is very easy and follows this form: `func_name = lambda [parameter_list]: expression}`.
- A function object is returned, which is equivalent to this: `def func_name([parameter_list]): return expression`

[36]:
```
def is_multiple_of_five(n):
    return not n % 5

def get_multiples_of_five(n):
    return list(filter(is_multiple_of_five, range(n)))

print(get_multiples_of_five(50))
```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

or, using a lambda function

[37]:
```
def get_multiples_of_five(n):
    return list(filter(lambda k : k % 5 == 0, range(n)))

print(get_multiples_of_five(50))
```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

And 2 more examples

[38]:
```
f = lambda x: x**2

f(5)
```

[38]:
```
25
```

[39]:
```
odd = lambda x: bool(x % 2)

odd(3)
```

[39]:
```
True
```

1.5 Conclusion

In short,
- **Functions should do one thing**: Functions that do one thing are easy to describe in one short sentence. Functions which do multiple things can be split into smaller functions which do one thing. These smaller functions are usually easier to read and understand.

- **Functions should be small**: The smaller they are, the easier it is to test them and to write them so that they do one thing.
- **The fewer input parameters, the better**: Functions which take a lot of arguments quickly become harder to manage (among other issues).
- **Functions should be consistent in their return values**: Returning `False` or `None` is not the same thing, even if within a `Boolean` context they both evaluate to `False`.`False` means that we have information (False), while `None` means that there is no information. Try writing functions which return in a consistent way, no matter what happens in their body.
- **Functions shouldn't have side effects**: In other words, functions should not affect the values you call them with. This is probably the hardest statement to understand at this point, so remember the example above with lists.

1.6 Exercises

Go here...

[ ]:
```

```