Python Essentials 1: Module 4

Functions, Tuples, Dictionaries, and Data Processing

In this module, you will cover the following topics:

code structuring and the concept of function;
function invocation and returning a result from a function;
name scopes and variable shadowing;
tuples and their purpose, constructing and using tuples;
dictionaries and their purpose, constructing and using dictionaries.

(part 2)

Parameterized functions

The function's full power reveals itself when it can be equipped with an interface that is able to accept data provided by the invoker. Such data can modify the function's behavior, making it more flexible and adaptable to changing conditions.

A parameter is actually a variable, but there are two important factors that make parameters different and special:

parameters exist only inside functions in which they have been defined, and the only place where the parameter can be defined is a space between a pair of parentheses in the def statement;

assigning a value to the parameter is done at the time of the function's invocation, by specifying the corresponding argument.

```
def function(parameter):
  ###
```

Don't forget:

parameters live inside functions (this is their natural environment)
arguments exist outside functions, and are carriers of values passed to corresponding parameters.

There is a clear and unambiguous frontier between these two worlds.

Let's enrich the function above with just one parameter - we're going to use it to show the user the number of a value the function asks for.

We have to rebuild the def statement - this is how it looks now:

```
def message(number):
    ###
```

The definition specifies that our function operates on just one parameter named number. You can use it as an ordinary variable, but only inside the function - it isn't visible anywhere else.

Let's now improve the function's body:

```
def message(number):
    print("Enter a number:", number)
```

We've made use of the parameter. Note: we haven't assigned the parameter with any value. Is it correct?

Yes, it is.

A value for the parameter will arrive from the function's environment.

Remember: specifying one or more parameters in a function's definition is also a requirement, and you have to fulfil it during invocation. You must provide as many arguments as there are defined parameters.

Failure to do so will cause an error.

Parametrized functions: continued
Try to run the code in the editor.

```
def message(number):
    print("Enter a number:", number)

message()
```

This is what you'll see in the console:

```
TypeError: message() missing 1 required positional argument: 'number'
```

This looks better, for sure:

```
def message(number):
    print("Enter a number:", number)

message(1)
```

Moreover, it behaves better. The code will produce the following output:

```
Enter a number: 1
```

Can you see how it works? The value of the argument used during invocation (1) has been passed into the function, setting the initial value of the parameter named number.

We have to make you sensitive to one important circumstance.

It's legal, and possible, to have a variable named the same as a function's parameter.

The snippet illustrates the phenomenon:

```
def message(number):
    print("Enter a number:", number)

number = 1234
message(1)
print(number)
```

A situation like this activates a mechanism called shadowing:

parameter x shadows any variable of the same name, but...
... only inside the function defining the parameter.
The parameter named number is a completely different entity from the variable named number.

This means that the snippet above will produce the following output:

```
Enter a number: 1
1234
```

Parametrized functions: continued

A function can have as many parameters as you want, but the more parameters you have, the harder it is to memorize their roles and purposes.

Let's modify the function - it has two parameters now:

```
def message(what, number):
    print("Enter", what, "number", number)
```

This also means that invoking the function will require two arguments.

The first new parameter is intended to carry the name of the desired value.

Here it is:

```
def message(what, number):
    print("Enter", what, "number", number)

message("telephone", 11)
message("price", 5)
message("number", "number")
```

This is the output you're about to see:

```
Enter telephone number 11
Enter price number 5
Enter number number number
```

Run the code, modify it, add more parameters, and see how this affects the output.

Positional parameter passing

A technique which assigns the ith (first, second, and so on) argument to the ith (first, second, and so on) function parameter is called positional parameter passing, while arguments passed in this way are named positional arguments.

You've used it already, but Python can offer a lot more. We're going to tell you about it now.

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

my_function(1, 2, 3)
```

Note: positional parameter passing is intuitively used by people in many social occasions. For example, it may be generally accepted that when we introduce ourselves we mention our first name(s) before our last name, e.g., "My name's John Doe."

Incidentally, Hungarians do it in reverse order.

Let's implement that social custom in Python. The following function will be responsible for introducing somebody:

```
def introduction(first_name, last_name):
    print("Hello, my name is", first_name, last_name)

introduction("Luke", "Skywalker")
introduction("Jesse", "Quick")
introduction("Clark", "Kent")
```

Can you guess the output? Run the code and find out if you were right.

Now imagine that the same function is being used in Hungary. In this case, the code would look like this:

```
def introduction(first_name, last_name):
    print("Hello, my name is", first_name, last_name)

introduction("Skywalker", "Luke")
introduction("Quick", "Jesse")
introduction("Kent", "Clark")
```

The output will look different. Can you guess it?

Run the code to see if you were right here, too. Are you surprised?

Can you make the function more culture-independent?

Keyword argument passing

Python offers another convention for passing arguments, where the meaning of the argument is dictated by its name, not by its position - it's called keyword argument passing.

Take a look at the snippet:

```
def introduction(first_name, last_name):
    print("Hello, my name is", first_name, last_name)

introduction(first_name = "James", last_name = "Bond")
introduction(last_name = "Skywalker", first_name = "Luke")
```

The concept is clear - the values passed to the parameters are preceded by the target parameters' names, followed by the = sign.

The position doesn't matter here - each argument's value knows its destination on the basis of the name used.

You should be able to predict the output. Run the code to check if you were right.

Of course, you mustn't use a non-existent parameter name.

The following snippet will cause a runtime error:

```
def introduction(first_name, last_name):
    print("Hello, my name is", first_name, last_name)

introduction(surname="Skywalker", first_name="Luke")
```

This is what Python will tell you:

```
TypeError: introduction() got an unexpected keyword argument 'surname'
```

Try it out yourself.

Mixing positional and keyword arguments

You can mix both fashions if you want - there is only one unbreakable rule: you have to put positional arguments before keyword arguments.

If you think for a moment, you'll certainly guess why.

To show you how it works, we'll use the following simple three-parameter function:

```
def adding(a, b, c):
    print(a, "+", b, "+", c, "=", a + b + c)
```

Its purpose is to evaluate and present the sum of all its arguments.

The function, when invoked in the following way:

```
adding(1, 2, 3)
```

will output:

```
1 + 2 + 3 = 6
```
output

It was - as you may suspect - a pure example of positional argument passing.

Of course, you can replace such an invocation with a purely keyword variant, like this:

```
adding(c = 1, a = 2, b = 3)
```

Our program will output a line like this:

```
2 + 3 + 1 = 6
```
output

Note the order of the values.

Let's try to mix both styles now.

Look at the function invocation below:

```
adding(3, c = 1, b = 2)
```

Let's analyze it:

- the argument (3) for the a parameter is passed using the positional way;
- the arguments for c and b are specified as keyword ones.

This is what you'll see in the console:

```
3 + 2 + 1 = 6
```
output

Be careful, and beware of mistakes. If you try to pass more than one value to one argument, all you'll get is a runtime error.

Look at the invocation below - it seems that we've tried to set a twice:

```
adding(3, a = 1, b = 2)
```

Python's response:

```
TypeError: adding() got multiple values for argument 'a'
output
```

Look at the snipet below. A code like this is fully correct, but it doesn't make much sense:

```
adding(4, 3, c = 2)
```

Everything is right, but leaving in just one keyword argument looks a bit weird - what do you think?

Parametrized functions - more details

It happens at times that a particular parameter's values are in use more often than others. Such arguments may have their default (predefined) values taken into consideration when their corresponding arguments have been omitted.

They say that the most popular English last name is Smith. Let's try to take this into account.

The default parameter's value is set using clear and pictorial syntax:

```
def introduction(first_name, last_name="Smith"):
    print("Hello, my name is", first_name, last_name)
```

You only have to extend the parameter's name with the = sign, followed by the default value.
Let's invoke the function as usual:

```
introduction("James", "Doe")
```

Can you guess the output of the program? Run it and check if you were right.
And? Everything looks the same, but when you invoke the function in a way that looks a bit suspicious at first sight, like this:

```
introduction("Henry")
```

or this:

```
introduction(first_name="William")
```

there will be no error, and both invocations will succeed, while the console will show the following output:

```
Hello, my name is Henry Smith
Hello, my name is William Smith
```

output

Test it.
You can go further if it's useful. Both parameters have their default values now, look at the code below:

```
def introduction(first_name="John", last_name="Smith"):
    print("Hello, my name is", first_name, last_name)
```

This makes the following invocation absolutely valid:

```
introduction()
```

And this is the expected output:

```
Hello, my name is John Smith
```

output

If you use one keyword argument, the remaining one will take the default value:

```
introduction(last_name="Hopkins")
```

The output is:

```
Hello, my name is John Hopkins
output
```

Test it.

Congratulations - you have just learned the basic ways of communicating with functions.

Key takeaways

1. You can pass information to functions by using parameters. Your functions can have as many parameters as you need.

An example of a one-parameter function:

```
def hi(name):
    print("Hi,", name)

hi("Greg")
```

An example of a two-parameter function:

```
def hi_all(name_1, name_2):
    print("Hi,", name_2)
    print("Hi,", name_1)

hi_all("Sebastian", "Konrad")
```

An example of a three-parameter function:

```
def address(street, city, postal_code):
    print("Your address is:", street, "St.,", city, postal_code)

s = input("Street: ")
p_c = input("Postal Code: ")
c = input("City: ")

address(s, c, p_c)
```

2. You can pass arguments to a function using the following techniques:

positional argument passing in which the order of arguments passed matters (Ex. 1),
keyword (named) argument passing in which the order of arguments passed doesn't matter (Ex. 2),
a mix of positional and keyword argument passing (Ex. 3).
Ex. 1

```
def subtra(a, b):
    print(a - b)

subtra(5, 2)    # outputs: 3
subtra(2, 5)    # outputs: -3
```

Ex. 2

```
def subtra(a, b):
    print(a - b)

subtra(a=5, b=2)    # outputs: 3
subtra(b=2, a=5)    # outputs: 3
```

Ex. 3

```
def subtra(a, b):
    print(a - b)

subtra(5, b=2)    # outputs: 3
subtra(5, 2)    # outputs: 3
```

It's important to remember that positional arguments mustn't follow keyword arguments. That's why if you try to run the following snippet:

```
def subtra(a, b):
    print(a - b)

subtra(5, b=2)    # outputs: 3
subtra(a=5, 2)    # Syntax Error
```

Python will not let you do it by signalling a SyntaxError.

3. You can use the keyword argument passing technique to pre-define a value for a given argument:

```
def name(first_name, last_name="Smith"):
    print(first_name, last_name)

name("Andy")    # outputs: Andy Smith
name("Betty", "Johnson")    # outputs: Betty Johnson (the keyword argument replaced by
"Johnson")
```

Exercise 1

What is the output of the following snippet?

```
def intro(a="James Bond", b="Bond"):
    print("My name is", b + ".", a + ".")

intro()
```

Exercise 2

What is the output of the following snippet?

```
def intro(a="James Bond", b="Bond"):
    print("My name is", b + ".", a + ".")

intro(b="Sean Connery")
```

Exercise 3

What is the output of the following snippet?

```
def intro(a, b="Bond"):
    print("My name is", b + ".", a + ".")

intro("Susan")
```

Exercise 4

What is the output of the following snippet?

```
def add_numbers(a, b=2, c):
    print(a + b + c)

add_numbers(a=1, c=3)
```