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.

Why do we need functions?

You've come across functions many times so far, but the view on their merits that we have given you has been rather one-sided. You've only invoked the functions by using them as tools to make life easier, and to simplify time-consuming and tedious tasks.

When you want some data to be printed on the console, you use print(). When you want to read the value of a variable, you use input(), coupled with either int() or float().

You've also made use of some methods, which are in fact functions, but declared in a very specific way.

Now you'll learn how to write your own functions, and how to use them. We'll write several functions together, from the very simple to the rather complex, which will require your focus and attention.

It often happens that a particular piece of code is repeated many times in your program. It's repeated either literally, or with only a few minor modifications, consisting of the use of other variables in the same algorithm. It also happens that a programmer cannot resist simplifying the work, and begins to clone such pieces of code using the clipboard and copy-paste operations.

It could end up as greatly frustrating when suddenly it turns out that there was an error in the cloned code. The programmer will have a lot of drudgery to find all the places that need corrections. There's also a high risk of the corrections causing errors.

We can now define the first condition which can help you decide when to start writing your own functions: if a particular fragment of the code begins to appear in more than one place, consider the possibility of isolating it in the form of a function invoked from the points where the original code was placed before.

It may happen that the algorithm you're going to implement is so complex that your code begins to grow in an uncontrolled manner, and suddenly you notice that you're not able to navigate through it so easily anymore.

You can try to cope with the issue by commenting the code extensively, but soon you find that this dramatically worsens your situation - too many comments make the code larger and harder to read. Some say that a well-written function should be viewed entirely in one glance.

A good and attentive developer divides the code (or more accurately: the problem) into well-isolated pieces, and encodes each of them in the form of a function.

This considerably simplifies the work of the program, because each piece of code can be encoded separately, and tested separately. The process described here is often called decomposition.

We can now state the second condition: if a piece of code becomes so large that reading and understating it may cause a problem, consider dividing it into separate, smaller problems, and implement each of them in the form of a separate function.

This decomposition continues until you get a set of short functions, easy to understand and test.

Decomposition

It often happens that the problem is so large and complex that it cannot be assigned to a single developer, and a team of developers have to work on it. The problem must be split between several developers in a way that ensures their efficient and seamless cooperation.

It seems inconceivable that more than one programmer should write the same piece of code at the same time, so the job has to be dispersed among all the team members.

This kind of decomposition has a different purpose to the one described previously - it's not only about sharing the work, but also about sharing the responsibility among many developers.

Each of them writes a clearly defined and described set of functions, which when combined into the module (we'll tell you about this a bit later) will give the final product.

This leads us directly to the third condition: if you're going to divide the work among multiple programmers, decompose the problem to allow the product to be implemented as a set of separately written functions packed together in different modules.

Where do the functions come from?
In general, functions come from at least three places:

from Python itself - numerous functions (like print()) are an integral part of Python, and are always available without any additional effort on behalf of the programmer; we call these functions built-in functions;

from Python's preinstalled modules - a lot of functions, very useful ones, but used significantly less often than built-in ones, are available in a number of modules installed together with Python; the use of these functions requires some additional steps from the programmer in order to make them fully accessible (we'll tell you about this in a while);

directly from your code - you can write your own functions, place them inside your code, and use them freely;

there is one other possibility, but it's connected with classes, so we'll omit it for now.

Your first function

Take a look at the snippet in the editor.

```
print("Enter a value: ")
a = int(input())

print("Enter a value: ")
b = int(input())

print("Enter a value: ")
c = int(input())
```

It's rather simple, but we only want it to be an example of transforming a repeating part of a code into a function.

The messages sent to the console by the print() function are always the same. Of course, there's nothing really bad in such a code, but try to imagine what you would have to do if your boss asked you to change the message to make it more polite, e.g., to start it with the phrase "Please,".

It seems that you'd have to spend some time changing all the occurrences of the message (you'd use a clipboard, of course, but it wouldn't make your life much easier). It's obvious that you'd probably make some mistakes during the amendment process, and you (and your boss) would get a bit frustrated.

Is it possible to separate such a repeatable part of the code, name it and make it reusable? It would mean that a change made once in one place would be propagated to all the places where it's used.

Of course, such a code should work only when it's explicitly launched.

Yes, it's possible. This is exactly what functions are for.

Your first function

How do you make such a function?

You need to define it. The word define is significant here.

This is what the simplest function definition looks like:

```
def function_name():
    function_body
```

It always starts with the keyword def (for define)
next after def goes the name of the function (the rules for naming functions are exactly the same as for naming variables)
after the function name, there's a place for a pair of parentheses (they contain nothing here, but that will change soon)
the line has to be ended with a colon;
the line directly after def begins the function body - a couple (at least one) of necessarily nested instructions, which will be executed every time the function is invoked; note: the function ends where the nesting ends, so you have to be careful.

We're ready to define our prompting function. We'll name it message - here it is:

```
def message():
    print("Enter a value: ")
```

The function is extremely simple, but fully usable. We've named it message, but you can label it according to your taste. Let's use it.

Our code contains the function definition now:

```
def message():
    print("Enter a value: ")

print("We start here.")
print("We end here.")
```

Note: we don't use the function at all - there's no invocation of it inside the code.

When you run it, you see the following output:

```
We start here.
We end here.
```
output

This means that Python reads the function's definitions and remembers them, but won't launch any of them without your permission.

We've modified the code now - we've inserted the function's invocation between the start and end messages:

```
def message():
    print("Enter a value: ")

print("We start here.")
message()
print("We end here.")
```

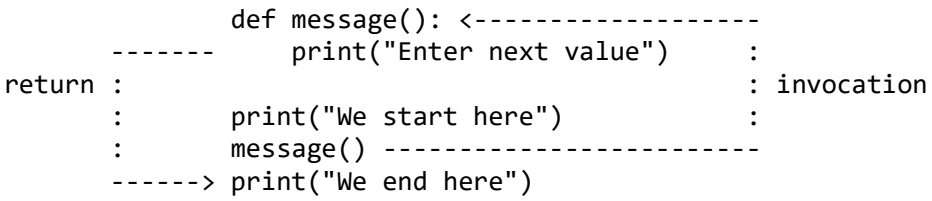
The output looks different now:

We start here.
Enter a value:
We end here.
output

Test the code, modify it, experiment with it.

How functions work

Look at the picture below:



It tries to show you the whole process:

when you invoke a function, Python remembers the place where it happened and jumps into the invoked function;
the body of the function is then executed;
reaching the end of the function forces Python to return to the place directly after the point of invocation.

There are two, very important, catches. Here's the first of them:

You mustn't invoke a function which is not known at the moment of invocation.

Remember - Python reads your code from top to bottom. It's not going to look ahead in order to find a function you forgot to put in the right place ("right" means "before invocation".)

We've inserted an error into this code - can you see the difference?

```
print("We start here.")
message()
print("We end here.")


def message():
    print("Enter a value: ")
```

We've moved the function to the end of the code. Is Python able to find it when the execution reaches the invocation?

No, it isn't. The error message will read:

NameError: name 'message' is not defined
output

Don't try to force Python to look for functions you didn't deliver at the right time.

The second catch sounds a little simpler:

You mustn't have a function and a variable of the same name.

The following snippet is erroneous:

```
def message():
    print("Enter a value: ")

message = 1
```

Assigning a value to the name message causes Python to forget its previous role. The function named message becomes unavailable.

Fortunately, you're free to mix your code with functions - you're not obliged to put all your functions at the top of your source file.

Look at the snippet:

```
print("We start here.")


def message():
    print("Enter a value: ")

message()

print("We end here.")
```

It may look strange, but it's completely correct, and works as intended.

Let's return to our primary example, and employ the function for the right job, like here:

```
def message():
    print("Enter a value: ")

message()
a = int(input())
message()
b = int(input())
message()
c = int(input())
```

Modifying the prompting message is now easy and clear - you can do it by changing the code in just one place - inside the function's body.

Open the sandbox, and try to do it yourself.

Key takeaways

1. A function is a block of code that performs a specific task when the function is called (invoked). You can use functions to make your code reusable, better organized, and more readable. Functions can have parameters and return values.

2. There are at least four basic types of functions in Python:
built-in functions which are an integral part of Python (such as the print() function). You can see a complete list of Python built-in functions at https://docs.python.org/3/library/functions.html.
the ones that come from pre-installed modules (you'll learn about them in the Python Essentials 2 course)
user-defined functions which are written by users for users - you can write your own functions and use them freely in your code,
the lambda functions (you'll learn about them in the Python Essentials 2 course.)

3. You can define your own function using the def keyword and the following syntax:

```
def your_function(optional parameters):
    # the body of the function
```

You can define a function which doesn't take any arguments, e.g.:

```
def message():    # defining a function
    print("Hello")    # body of the function

message()    # calling the function
```

You can define a function which takes arguments, too, just like the one-parameter function below:

```
def hello(name):    # defining a function
    print("Hello,", name)    # body of the function

name = input("Enter your name: ")
hello(name)    # calling the function
```

We'll tell you more about parametrized functions in the next section. Don't worry.

Exercise 1

The input() function is an example of a:
a) user-defined function
b) built-in function

Exercise 2
What happens when you try to invoke a function before you define it? Example:

```
hi()
def hi():
    print("hi!")
```

Exercise 3
What will happen when you run the code below?

```
def hi():
    print("hi")
hi(5)
```