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 3)

Effects and results: the return instruction

All the previously presented functions have some kind of effect - they produce some text and send it to the console.

Of course, functions - like their mathematical siblings - may have results.

To get functions to return a value (but not only for this purpose) you use the return instruction.

This word gives you a full picture of its capabilities. Note: it's a Python keyword.

The return instruction has two different variants - let's consider them separately.

return without an expression
The first consists of the keyword itself, without anything following it.

When used inside a function, it causes the immediate termination of the function's execution, and an instant return (hence the name) to the point of invocation.

Note: if a function is not intended to produce a result, using the return instruction is not obligatory - it will be executed implicitly at the end of the function.

Anyway, you can use it to terminate a function's activities on demand, before the control reaches the function's last line.

Let's consider the following function:

```
def happy_new_year(wishes = True):
    print("Three...")
    print("Two...")
    print("One...")
    if not wishes:
        return

    print("Happy New Year!")
```

When invoked without any arguments:

```
happy_new_year()
```

The function causes a little noise - the output will look like this:

Three...
Two...
One...
Happy New Year!
output

Providing False as an argument:

```
happy_new_year(False)
```

will modify the function's behavior - the return instruction will cause its termination just before the wishes - this is the updated output:

```
Three...
Two...
One...
output
```

return with an expression
The second return variant is extended with an expression:

```
def function():
    return expression
```

There are two consequences of using it:

- it causes the immediate termination of the function's execution (nothing new compared to the first variant)
- moreover, the function will evaluate the expression's value and will return (hence the name once again) it as the function's result.

Yes, we already know - this example isn't really sophisticated:

```
def boring_function():
    return 123

x = boring_function()

print("The boring_function has returned its result. It's:", x)
```

The snippet writes the following text to the console:

```
The boring_function has returned its result. It's: 123
```

Let's investigate it for a while.

Analyze the figure below: Assigning the value returned by function to a variable

The return instruction, enriched with the expression (the expression is very simple here), "transports" the expression's value to the place where the function has been invoked.

The result may be freely used here, e.g., to be assigned to a variable.

It may also be completely ignored and lost without a trace.

Note, we're not being too polite here - the function returns a value, and we ignore it (we don't use it in any way):

```
def boring_function():
    print("'Boredom Mode' ON.")
    return 123
```

```
print("This lesson is interesting!")
boring_function()
print("This lesson is boring...")
```

The program produces the following output:

```
This lesson is interesting!
'Boredom Mode' ON.
This lesson is boring...
```

output

Is it punishable? Not at all.

The only disadvantage is that the result has been irretrievably lost.

Don't forget:

- you are always allowed to ignore the function's result, and be satisfied with the function's effect (if the function has any)
- if a function is intended to return a useful result, it must contain the second variant of the return instruction.

Wait a minute - does this mean that there are useless results, too? Yes - in some sense.

A few words about None

Let us introduce you to a very curious value (to be honest, a none value) named None.

Its data doesn't represent any reasonable value - actually, it's not a value at all; hence, it mustn't take part in any expressions.

For example, a snippet like this:

```
print(None + 2)
```

will cause a runtime error, described by the following diagnostic message:

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
output

Note: None is a keyword.

There are only two kinds of circumstances when None can be safely used:

- when you assign it to a variable (or return it as a function's result)
- when you compare it with a variable to diagnose its internal state.

Just like here:

```
value = None
if value is None:
    print("Sorry, you don't carry any value")
```

Don't forget this: if a function doesn't return a certain value using a return expression clause, it is assumed that it implicitly returns None.

Let's test it.

A few words about None: continued

Take a look at the code in the editor.

```
def strange_function(n):
    if(n % 2 == 0):
        return True
```

It's obvious that the strangeFunction function returns True when its argument is even.

What does it return otherwise?

We can use the following code to check it:

```
print(strange_function(2))
print(strange_function(1))
```

This is what we see in the console:

True
None
output

Don't be surprised next time you see None as a function result - it may be the symptom of a subtle mistake inside the function.

Effects and results: lists and functions

There are two additional questions that should be answered here.

The first is: may a list be sent to a function as an argument?

Of course it may! Any entity recognizable by Python can play the role of a function argument, although it has to be assured that the function is able to cope with it.

So, if you pass a list to a function, the function has to handle it like a list.

A function like this one here:

```
def list_sum(lst):
    s = 0

    for elem in lst:
        s += elem

    return s
```

and invoked like this:

```
print(list_sum([5, 4, 3]))
```

will return 12 as a result, but you should expect problems if you invoke it in this risky way:

```
print(list_sum(5))
```

Python's response will be unequivocal:

TypeError: 'int' object is not iterable
output

This is caused by the fact that a single integer value mustn't be iterated through by the for loop.

Effects and results: lists and functions - continued

The second question is: may a list be a function result?

Yes, of course! Any entity recognizable by Python can be a function result.

Look at the code in the editor.

```
def strange_list_fun(n):
    strange_list = []

    for i in range(0, n):
        strange_list.insert(0, i)

    return strange_list

print(strange_list_fun(5))
```

The program's output will be like this:

```
[4, 3, 2, 1, 0]
output
```

Now you can write functions with and without results.

Let's dive a little deeper into the issues connected with variables in functions. This is essential for creating effective and safe functions.

LAB

Estimated time
10-15 minutes

Level of difficulty
Easy

Objectives
Familiarize the student with:

projecting and writing parameterized functions;
utilizing the return statement;
testing the functions.

Scenario
Your task is to write and test a function which takes one argument (a year) and returns True if the year is a leap year, or False otherwise.

The seed of the function is already sown in the skeleton code in the editor.

Note: we've also prepared a short testing code, which you can use to test your function.

The code uses two lists - one with the test data, and the other containing the expected results. The code will tell you if any of your results are invalid.

```
def is_year_leap(year):
#
# put your code here
#

test_data = [1900, 2000, 2016, 1987]
test_results = [False, True, True, False]
for i in range(len(test_data)):
      yr = test_data[i]
      print(yr,"->",end="")
      result = is_year_leap(yr)
      if result == test_results[i]:
            print("OK")
      else:
            print("Failed")
```

LAB

Estimated time
15-20 minutes

Level of difficulty
Medium

Prerequisites
LAB 4.3.1.6

Objectives
Familiarize the student with:

projecting and writing parameterized functions;
utilizing the return statement;
utilizing the student's own functions.

Scenario
Your task is to write and test a function which takes two arguments (a year and a month) and returns the number of days for the given month/year pair (while only February is sensitive to the year value, your function should be universal).

The initial part of the function is ready. Now, convince the function to return None if its arguments don't make sense.

Of course, you can (and should) use the previously written and tested function (LAB 4.3.1.6). It may be very helpful. We encourage you to use a list filled with the months' lengths. You can create it inside the function - this trick will significantly shorten the code.

We've prepared a testing code. Expand it to include more test cases.

```
def is_year_leap(year):
#
# Your code from LAB 4.3.1.6.
#

def days_in_month(year, month):
#
# Write your new code here.
#

test_years = [1900, 2000, 2016, 1987]
test_months = [2, 2, 1, 11]
test_results = [28, 29, 31, 30]
for i in range(len(test_years)):
      yr = test_years[i]
      mo = test_months[i]
      print(yr, mo, "->", end="")
      result = days_in_month(yr, mo)
      if result == test_results[i]:
            print("OK")
      else:
            print("Failed")
```

LAB

Estimated time
20-30 minutes

Level of difficulty
Medium

Prerequisites
LAB 4.3.1.6
LAB 4.3.1.7

Objectives
Familiarize the student with:

projecting and writing parameterized functions;
utilizing the return statement;
building a set of utility functions;
utilizing the student's own functions.

Scenario
Your task is to write and test a function which takes three arguments (a year, a month, and a day of the month) and returns the corresponding day of the year, or returns None if any of the arguments is invalid.

Use the previously written and tested functions. Add some test cases to the code. This test is only a beginning.

```
def is_year_leap(year):
#
# Your code from LAB 4.3.1.6.
#

def days_in_month(year, month):
#
# Your code from LAB 4.3.1.7.
#

def day_of_year(year, month, day):
#
# Write your new code here.
#

print(day_of_year(2000, 12, 31))
```

LAB

Estimated time
15-20 minutes

Level of difficulty
Medium

Objectives
familiarizing the student with classic notions and algorithms;
improving the student's skills in defining and using functions.
Scenario
A natural number is prime if it is greater than 1 and has no divisors other than 1 and itself.

Complicated? Not at all. For example, 8 isn't a prime number, as you can divide it by 2 and 4 (we can't use divisors equal to 1 and 8, as the definition prohibits this).

On the other hand, 7 is a prime number, as we can't find any legal divisors for it.

Your task is to write a function checking whether a number is prime or not.

The function:

is called is_prime;
takes one argument (the value to check)
returns True if the argument is a prime number, and False otherwise.
Hint: try to divide the argument by all subsequent values (starting from 2) and check the remainder - if it's zero, your number cannot be a prime; think carefully about when you should stop the process.

If you need to know the square root of any value, you can utilize the ** operator. Remember: the square root of x is the same as x0.5

Complete the code in the editor.

Run your code and check whether your output is the same as ours.

Expected output
2 3 5 7 11 13 17 19

```
def is_prime(num):
#
# Write your code here.
#

for i in range(1, 20):
      if is_prime(i + 1):
                  print(i + 1, end=" ")
print()
```

LAB

Estimated time
10-15 minutes

Level of difficulty
Easy

Objectives
improving the student's skills in defining, using and testing functions.
Scenario
A car's fuel consumption may be expressed in many different ways. For example, in Europe, it is shown as the amount of fuel consumed per 100 kilometers.

In the USA, it is shown as the number of miles traveled by a car using one gallon of fuel.

Your task is to write a pair of functions converting l/100km into mpg, and vice versa.

The functions:

are named liters_100km_to_miles_gallon and miles_gallon_to_liters_100km respectively;
take one argument (the value corresponding to their names)
Complete the code in the editor.

Run your code and check whether your output is the same as ours.

Here is some information to help you:

1 American mile = 1609.344 metres;
1 American gallon = 3.785411784 litres.
Expected output
60.31143162393162
31.36194444444444
23.52145833333333
3.9007393587617467
7.490910297239916
10.009131205673757

```
def liters_100km_to_miles_gallon(liters):
#
# Write your code here.
#

def miles_gallon_to_liters_100km(miles):
#
# Write your code here
#

print(liters_100km_to_miles_gallon(3.9))
print(liters_100km_to_miles_gallon(7.5))
print(liters_100km_to_miles_gallon(10.))
print(miles_gallon_to_liters_100km(60.3))
print(miles_gallon_to_liters_100km(31.4))
print(miles_gallon_to_liters_100km(23.5))
```

Key takeaways

1. You can use the return keyword to tell a function to return some value. The return statement exits the function, e.g.:

```
def multiply(a, b):
    return a * b

print(multiply(3, 4))    # outputs: 12


def multiply(a, b):
    return

print(multiply(3, 4))    # outputs: None
```

2. The result of a function can be easily assigned to a variable, e.g.:

```
def wishes():
    return "Happy Birthday!"

w = wishes()

print(w)    # outputs: Happy Birthday!
```

Look at the difference in output in the following two examples:

```
# Example 1
def wishes():
    print("My Wishes")
    return "Happy Birthday"

wishes()    # outputs: My Wishes


# Example 2
def wishes():
    print("My Wishes")
    return "Happy Birthday"

print(wishes())

# outputs: My Wishes
#          Happy Birthday
```

3. You can use a list as a function's argument, e.g.:

```
def hi_everybody(my_list):
    for name in my_list:
        print("Hi,", name)

hi_everybody(["Adam", "John", "Lucy"])
```

4. A list can be a function result, too, e.g.:

```
def create_list(n):
    my_list = []
    for i in range(n):
        my_list.append(i)
    return my_list

print(create_list(5))
```

Exercise 1
What is the output of the following snippet?

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

Exercise 2
What is the output of the following snippet?

```
def is_int(data):
    if type(data) == int:
        return True
    elif type(data) == float:
        return False
print(is_int(5))
print(is_int(5.0))
print(is_int("5"))
```

Exercise 3
What is the output of the following snippet?

```
def even_num_lst(ran):
    lst = []
    for num in range(ran):
        if num % 2 == 0:
            lst.append(num)
    return lst
print(even_num_lst(11))
```

Exercise 4
What is the output of the following snippet?

```
def list_updater(lst):
    upd_list = []
    for elem in lst:
        elem **= 2
        upd_list.append(elem)
    return upd_list
foo = [1, 2, 3, 4, 5]
print(list_updater(foo))
```