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 5)

Some simple functions: evaluating the BMI

Let's get started on a function to evaluate the Body Mass Index (BMI).

BMI = (weight in KGs) / ((height in meters)^2)

As you can see, the formula gets two values:

weight (originally in kilograms)
height (originally in meters)

It seems that this new function will have two parameters. Its name will be bmi, but if you prefer any other name, use it instead.

Let's code the function:

```
def bmi(weight, height):
    return weight / height ** 2


print(bmi(52.5, 1.65))
```

The result produced by the sample invocation looks as follows:

```
19.283746556473833
```
output

The function fulfils our expectations, but it's a bit simple - it assumes that the values of both parameters are always meaningful. It's definitely worth checking if they're trustworthy.

Let's check them both and return None if any of them looks suspicious.

Some simple functions: evaluating BMI and converting imperial units to metric units

Look at the code in the editor. There are two things we need to pay attention to.

```
def bmi(weight, height):
    if height < 1.0 or height > 2.5 or \
    weight < 20 or weight > 200:
        return None

    return weight / height ** 2


print(bmi(352.5, 1.65))
```

First, the test invocation ensures that the protection works properly - the output is:

```
None
```
output

Second, take a look at the way the backslash (\) symbol is used. If you use it in Python code and end a line with it, it will tell Python to continue the line of code in the next line of code.

It can be particularly useful when you have to deal with long lines of code and you'd like to improve code readability.

Okay, but there's something we omitted too easily - the imperial measurements. This function is not too useful for people accustomed to pounds, feet and inches.

What can be done for them?

We can write two simple functions to convert imperial units to metric ones. Let's start with pounds.

It is a well-known fact that 1 lb = 0.45359237 kg. We'll use this in our new function.

This is our helper function, named lb_to_kg:

```
def lb_to_kg(lb):
    return lb * 0.45359237


print(lb_to_kg(1))
```

The result of the test invocation looks good:

```
0.45359237
```
output

And now it's time for feet and inches: 1 ft = 0.3048 m, and 1 in = 2.54 cm = 0.0254 m.

The function we've written is named ft_and_inch_to_m:

```
def ft_and_inch_to_m(ft, inch):
    return ft * 0.3048 + inch * 0.0254


print(ft_and_inch_to_m(1, 1))
```

The result of a quick test is:

```
0.3302
```

output

It looks as expected.

Note: we wanted to name the second parameter just in, not inch, but we couldn't. Do you know why?

in is a Python keyword - it cannot be used as a name.

Let's convert six feet into meters:

```
print(ft_and_inch_to_m(6, 0))
```

And this is the output:

```
1.8288000000000002
```

output

It's quite possible that sometimes you may want to use just feet without inches. Will Python help you? Of course it will.

We've modified the code a bit:

```
def ft_and_inch_to_m(ft, inch = 0.0):
    return ft * 0.3048 + inch * 0.0254


print(ft_and_inch_to_m(6))
```

Now the inch parameter has its default value equal to 0.0.

The code produces the following output - this is what is expected:

```
1.8288000000000002
```

output

Finally, the code is able to answer the question: what is the BMI of a person 5'7" tall and weighing 176 lbs?

This is the code we have built:

```
def ft_and_inch_to_m(ft, inch = 0.0):
    return ft * 0.3048 + inch * 0.0254


def lb_to_kg(lb):
    return lb * 0.45359237


def bmi(weight, height):
    if height < 1.0 or height > 2.5 or weight < 20 or weight > 200:
        return None

    return weight / height ** 2


print(bmi(weight = lb_to_kg(176), height = ft_and_inch_to_m(5, 7)))
```

And the answer is:

```
27.565214082533313
```
output

Run the code and test it.

Some simple functions: continued

Let's play with triangles now. We'll start with a function to check whether three sides of given lengths can build a triangle.

We know from school that the sum of two arbitrary sides has to be longer than the third side.
It won't be a hard challenge. The function will have three parameters - one for each side.
It will return True if the sides can build a triangle, and False otherwise. In this case, is_a_triangle is a good name for such a function.
Look at the code in the editor. You can find our function there. Run the program.

```
def is_a_triangle(a, b, c):
    if a + b <= c:
        return False
    if b + c <= a:
        return False
    if c + a <= b:
        return False
    return True


print(is_a_triangle(1, 1, 1))
print(is_a_triangle(1, 1, 3))
```

It seems that it works well - these are the results:

```
True
False
output
```

Can we make it more compact? It looks a bit wordy.

This is a more compact version:

```
def is_a_triangle(a, b, c):
    if a + b <= c or b + c <= a or c + a <= b:
        return False
    return True


print(is_a_triangle(1, 1, 1))
print(is_a_triangle(1, 1, 3))
```

Can we compact it even more?

Yes, we can - look:

```
def is_a_triangle(a, b, c):
    return a + b > c and b + c > a and c + a > b

print(is_a_triangle(1, 1, 1))
print(is_a_triangle(1, 1, 3))
```

We've negated the condition (reversed the relational operators and replaced ors with ands, receiving a universal expression for testing triangles).

Let's install the function in a larger program. It'll ask the user for three values and make use of the function.

Some simple functions: triangles and the Pythagorean theorem

Look at the code in the editor. It asks the user for three values. Then it makes use of the is_a_triangle function. The code is ready to run.

```
def is_a_triangle(a, b, c):
    return a + b > c and b + c > a and c + a > b


a = float(input('Enter the first side\'s length: '))
b = float(input('Enter the second side\'s length: '))
c = float(input('Enter the third side\'s length: '))

if is_a_triangle(a, b, c):
    print('Yes, it can be a triangle.')
else:
    print('No, it can\'t be a triangle.')
```

In the second step, we'll try to ensure that a certain triangle is a right-angle triangle.

We will need to make use of the Pythagorean theorem:

```
c^2 = a^2 + b^2
```

How do we recognize which of the three sides is the hypotenuse?

The hypotenuse is the longest side.

Here is the code:

```
def is_a_triangle(a, b, c):
    return a + b > c and b + c > a and c + a > b


def is_a_right_triangle(a, b, c):
    if not is_a_triangle(a, b, c):
        return False
    if c > a and c > b:
        return c ** 2 == a ** 2 + b ** 2
    if a > b and a > c:
        return a ** 2 == b ** 2 + c ** 2


print(is_a_right_triangle(5, 3, 4))
print(is_a_right_triangle(1, 3, 4))
```

Look at how we test the relationship between the hypotenuse and the remaining sides - we choose the longest side, and apply the Pythagorean theorem to check if everything is right. This requires three checks in total.

Some simple functions: evaluating a triangle's area

We can also evaluate a triangle's area. Heron's formula will be handy here:

```
s = (a + b + c) / 2
A = (s * (s - a) * (s - b) * (s - c)) ^ 0.5
```

We're going use the exponentiation operator to find the square root - it may seem strange, but it works

This is the resulting code:

```
def is_a_triangle(a, b, c):
    return a + b > c and b + c > a and c + a > b


def heron(a, b, c):
    p = (a + b + c) / 2
    return (p * (p - a) * (p - b) * (p - c)) ** 0.5


def area_of_triangle(a, b, c):
    if not is_a_triangle(a, b, c):
        return None
    return heron(a, b, c)


print(area_of_triangle(1., 1., 2. ** .5))
```

We try it with a right-angle triangle as a half of a square with one side equal to 1. This means that its area should be equal to 0.5.

It's odd - the code produces the following output:

```
0.49999999999999983
```
output

It's very close to 0.5, but it isn't exactly 0.5. What does it mean? Is it an error?

No, it isn't. This is the specifics of floating-point calculations. We'll tell you more about it soon.

Some simple functions: factorials

Another function we're about to write is factorials. Do you remember how a factorial is defined?

```
0! = 1 (yes! it's true)
1! = 1
2! = 1 * 2
3! = 1 * 2 * 3
4! = 1 * 2 * 3 * 4
:
:
n! = 1 * 2 ** 3 * 4 * ... * n-1 * n
```

It's marked with an exclamation mark, and is equal to the product of all natural numbers from one up to its argument.

Let's write our code. We'll create a function and call it factorial_function. Here is the code:

```
def factorial_function(n):
    if n < 0:
        return None
    if n < 2:
        return 1

    product = 1
    for i in range(2, n + 1):
        product *= i
    return product


for n in range(1, 6):  # testing
    print(n, factorial_function(n))
```

Notice how we mirror step by step the mathematical definition, and how we use the for loop to find the product.

We add a simple testing code, and these are the results we get:

```
1 1
2 2
3 6
4 24
5 120
```

Some simple functions: Fibonacci numbers

Are you familiar with Fibonacci numbers?

They are a sequence of integer numbers built using a very simple rule:

the first element of the sequence is equal to one (Fib1 = 1)
the second is also equal to one (Fib2 = 1)
every subsequent number is the the_sum of the two preceding numbers:
(Fibi = Fibi-1 + Fibi-2)
Here are some of the first Fibonacci numbers:

```
fib(1) = 1
fib(2) = 1
fib(3) = 1 + 1 = 2
fib(4) = 1 + 2 = 3
fib(5) = 2 + 3 = 5
fib(6) = 3 + 5 = 8
fib(7) = 5 + 8 = 13
```

What do you think about implementing this as a function?

Let's create our fib function and test it. Here it is:

```
def fib(n):
    if n < 1:
        return None
    if n < 3:
        return 1

    elem_1 = elem_2 = 1
    the_sum = 0
    for i in range(3, n + 1):
        the_sum = elem_1 + elem_2
        elem_1, elem_2 = elem_2, the_sum
    return the_sum


for n in range(1, 10):  # testing
    print(n, "->", fib(n))
```

Analyze the for loop body carefully, and find out how we move the elem_1 and elem_2 variables through the subsequent Fibonacci numbers.

The test part of the code produces the following output:

```
1 -> 1
2 -> 1
3 -> 2
4 -> 3
5 -> 5
6 -> 8
7 -> 13
8 -> 21
9 -> 34
```

Some simple functions: recursion

There's one more thing we want to show you to make everything complete - it's recursion.
This term may describe many different concepts, but one of them is especially interesting - the one referring to computer programming.
In this field, recursion is a technique where a function invokes itself.
These two cases seem to be the best to illustrate the phenomenon - factorials and Fibonacci numbers. Especially the latter.
The Fibonacci numbers definition is a clear example of recursion. We already told you that:

```
Fib(i) = Fib(i-1) + Fib(i-2)
```

The definition of the ith number refers to the i-1 number, and so on, till you reach the first two.
Can it be used in the code? Yes, it can. It can also make the code shorter and clearer.
The second version of our fib() function makes direct use of this definition:

```
def fib(n):
    if n < 1:
        return None
    if n < 3:
        return 1
    return fib(n - 1) + fib(n - 2)
```

The code is much clearer now.
But is it really safe? Does it entail any risk?
Yes, there is a little risk indeed. If you forget to consider the conditions which can stop the chain of recursive invocations, the program may enter an infinite loop. You have to be careful.
The factorial has a second, recursive side too. Look:

```
n! = 1 × 2 × 3 × ... × n-1 × n
```

It's obvious that:

```
1 × 2 × 3 × ... × n-1 = (n-1)!
```

So, finally, the result is:

```
n! = (n-1)! × n
```

This is in fact a ready recipe for our new solution.
Here it is:

```
def factorial_function(n):
    if n < 0:
        return None
    if n < 2:
        return 1
    return n * factorial_function(n - 1)
```

Does it work? Yes, it does. Try it for yourself.
Our short functional journey is almost over. The next section will take care of two curious Python data types: tuples and dictionaries.

Key takeaways

1. A function can call other functions or even itself. When a function calls itself, this situation is known as recursion, and the function which calls itself and contains a specified termination condition (i.e., the base case - a condition which doesn't tell the function to make any further calls to that function) is called a recursive function.

2. You can use recursive functions in Python to write clean, elegant code, and divide it into smaller, organized chunks. On the other hand, you need to be very careful as it might be easy to make a mistake and create a function which never terminates. You also need to remember that recursive calls consume a lot of memory, and therefore may sometimes be inefficient.

When using recursion, you need to take all its advantages and disadvantages into consideration.

The factorial function is a classic example of how the concept of recursion can be put in practice:

```
# Recursive implementation of the factorial function.

def factorial(n):
    if n == 1:    # The base case (termination condition.)
        return 1
    else:
        return n * factorial(n - 1)


print(factorial(4)) # 4 * 3 * 2 * 1 = 24
```

Exercise 1

What will happen when you attempt to run the following snippet and why?

```
def factorial(n):
    return n * factorial(n - 1)


print(factorial(4))
```

Exercise 2

What is the output of the following snippet?

```
def fun(a):
    if a > 30:
        return 3
    else:
        return a + fun(a + 3)


print(fun(25))
```