05. Functions

September 30, 2025

1 5. Functions

A function is a group of related statements that performs a specific task.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes the code reusable.

1.1 5.1 Defining Functions

```
def function_name(parameters):
    """docstring"""
    statement(s)
    return value
```

Above shown is a function definition that consists of the following components:

- Keyword `def` marks the start of the function header.
- A function name to uniquely identify the function. When naming functions, in Python, we should use the lowercase snake_case practice.
- Optional parameters (arguments) through which we pass values to a function.
- Optional inline documentation (docstring) describing the function.
- One or more valid python statements that make up the function body
- An optional `return` statement to return a value from the function

Let's see a simple function in action:

[1]:
```
def my_function():
    """Prints a greeting message"""
    print('Hello world!')
```

We can find out more details about our function by passing it to the built-in function `help()`:

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

```
Help on function my_function in module __main__:

my_function()
    Prints a greeting message
```

1.2 Exercises 5.1

1. Create a function that prints a message of your choice.
2. Add a docstring to the function.
3. Call `help()` on the function.

1.3 5.2 Function call

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name followed by parantheses inside which we place the arguments, if the function expects them.

[4]:
```
my_function()
```

```
Hello world!
```

1.4 Exercises 5.2

1. Call the function you created previously 5 times using a loop (`for` or `while`).

1.5 5.3 Function parameters and arguments

Function parameters are the names listed in the function's definition. Function arguments are the real values passed to the function.

When defining a function, we can list its parameters in parentheses, after the function name:

[5]:
```
def greet(name):
    print(f"Hello, {name}!")
```

When calling the function, we will pass a value to it as an argument:

[6]:
```
greet("Anna")
```

```
Hello, Anna!
```

1.5.1 5.3.1 Required parameters

In the function `greet`, `name` is a required parameter. Not passing any argument in the function call will result in an error:

[7]:
```
greet()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[7], line 1
----> 1 greet()

TypeError: greet() missing 1 required positional argument: 'name'
```

All parameters are required by default. Let's see another example:

[8]:
```python
def print_sum(a, b):
    print(a + b)
```

[9]:
```python
print_sum(10, 2)  # get_sum must be called with exactly two arguments
```

```
12
```

1.5.2 5.3.2 Optional parameters

You can specify default values for parameters in a function. Parameters that have a default value are optional, meaning that if no value is passed for them in the function call, the default value will be used:

[10]:
```python
def greet(name="Anna"):
    print(f"Hello, {name}!")
```

[11]:
```python
greet()  # using the default value for parameter name
```

```
Hello, Anna!
```

[12]:
```python
greet("Jane")  # passing "Jane" as value for parameter name
```

```
Hello, Jane!
```

For functions with more parameters, optional parameters should always follow required parameters:

[13]:
```python
def welcome(name, age, is_student=False, hobbies=None):
    print(f"{name} is {age} years old.")
    if is_student:
        print(f"{name} is a student.")
    if hobbies:
        print(f"{name} has the following hobbies:")
        for hobby in hobbies:
            print(f" - {hobby}")
```

1.5.3 5.3.3 Positional and keyword arguments

When you call a function with positional arguments, arguments get mapped to formal parameters according to their **position**. All the calls we have already made used positional arguments. Let's see another example:

[14]:
```python
welcome("Anna", 20, True, ["reading", "painting"])
```

```
Anna is 20 years old.
Anna is a student.
Anna has the following hobbies:
 - reading
 - painting
```

When you call a function with keyword arguments, arguments get mapped to formal parameters according to their **name**:

[15]:
```
welcome(name="Anna", is_student=True, age=20, hobbies=["reading", "painting"])
```

```
Anna is 20 years old.
Anna is a student.
Anna has the following hobbies:
 - reading
 - painting
```

A value for any parameter, be it required or optional, can be sent as a keyword argument, for more clarity. But being able to call functions with keyword arguments is especially useful when you want to specify a value for some (not all) the default arguments:

[16]:
```
welcome("Anna", 20, hobbies=["reading", "painting"])
```

```
Anna is 20 years old.
Anna has the following hobbies:
 - reading
 - painting
```

1.6 Exercises 5.3

1. Write a function `multiply` that receives two required parameters and prints their product (using `*` operator). Call the function with the following arguments:

   - `7` and `6.2`
   - `"hello"` and `3`
   - `[0]` and `5`

   What do you notice?

2. Write a Python function called `greet` that takes an optional parameter `name`. If a `name` is provided, print a greeting message to that name; otherwise, print a generic greeting message. Call the function with and without a parameter.

3. Call the function above on each of the items in the following list:

   ```
   names = ["Anna", "Jane", "Mike"]
   ```

1.7 5.4 Function output (`return` statement)

By now, we have written functions that only display different messages. However, functions are commonly used to return different values which can be further used in the program.

Let's take a look at built-in function `len()`. If you simply write `len(some_object)` in your program, nothing happens:

[17]:
```
my_list = [10, 12, 14]
len(my_list)
for item in my_list:
    print(item)
```

```
10
12
14
```

That is because `len` doesn't print anything, it rather returns that value:

[18]:
```python
list_length = len(my_list)
print(list_length)
```

```
3
```

This behavior enables it to be used in expressions, like:

[19]:
```python
if len(my_list) > 2:
    print("List has more than two elements.")
```

```
List has more than two elements.
```

We can also create functions that return values. The `return` statement is used to exit a function and go back to the place from where it was called. This statement can contain an expression that gets evaluated and the value is returned.

Let's compare the following two functions:

[20]:
```python
def print_sum(a, b):
    print(a + b)
```

[21]:
```python
def get_sum(a, b):
    return a + b
```

[22]:
```python
sum_printed = print_sum(10, 15)
```

```
25
```

[23]:
```python
sum_returned = get_sum(10, 15)
```

If there is no expression in the `return` statement or the `return` statement itself is not present inside a function, then the function will return the `None` object. That is why, `sum_printed` is `None`:

[24]:
```python
print(sum_printed)
```

```
None
```

But `sum_returned` contains the actual result:

[25]:
```python
print(sum_returned)
```

```
25
```

Which can be used in other computations:

[26]:
```python
sum_returned ** 2
```

[26]:
```
625
```

A function can have multiple `return` statements, but a single one will be executed: the first one the interpreter encounters.

[27]:
```
def classify_number(num):
    if num > 0:
        return "Positive"
    elif num < 0:
        return "Negative"
    return "Zero"
```

[28]:
```
classify_number(10)
```

[28]:
```
'Positive'
```

[29]:
```
classify_number(0)
```

[29]:
```
'Zero'
```

A function cannot return multiple objects. The closest thing to that is returning a tuple, and then unpacking its result into different variables:

[30]:
```
def do_math(a, b):
    return a + b, a - b, a * b, a // b
```

[31]:
```
total, diff, prod, quotient = do_math(10, 2)
```

[32]:
```
print(total, diff, prod, quotient)
```

```
12 8 20 5
```

1.8 Exercises 5.4

1. Write a function that takes a number as a parameter and **prints** its square.
2. Write another function that takes a number as a parameter and **returns** its square.
3. Are the results of the two functions above different? Which of the two functions can be used to compute x2 + y2 ?
4. Write a function `get_rectangle_properties` that takes the width and height of a rectangle as parameters and returns the area and perimeter of the rectangle. Call the function and assign the result to two variables.
5. Write a function `build_html` which receives two parameters `tag` and `content`, and returns the following string: > <tag>content</tag>

   Call it so that you build the following output: > <div><h1>Functions</h1><p>Functions are fun!</p></div>

1.9 5.5 Scope and lifetime of variables

Variable scope and lifetime refer to where and when variables can be accessed within a Python program.

1.9.1 5.5.1 Variable Scope

Variable scope refers to the region of code where a variable is accessible.

In Python, variables can have different scopes, primarily: - **Global Scope**: Variables defined outside of any function or class have global scope. They can be accessed from anywhere within the program. - **Local Scope**: Variables defined within a function have local scope. They can only be accessed within the function where they are defined.

[33]:
```python
x = 10  # Global variable

def func():
    print(x)  # Accessing global variable

func()  # Output: 10
```

```
10
```

[36]:
```python
def func():
    y = 20  # Local variable
    print(y)

func()  # Output: 20
# print(y)  # Error: NameError: name 'y' is not defined
```

```
20
```

1.9.2 5.5.2 Variable lifetime

The lifetime of a variable is the period throughout which the variable exists in the memory.

In Python, variable lifetime is determined by its scope: - Global variables exist for the entire duration of the program. - Local variables are created when the function is called and destroyed when the function exits. Hence, a function does not remember the value of a variable from its previous calls.

## Exercises 5.5

1. Define a global variable `vowels` as a string containing all vowels (lowercase and uppercase). Write a function `remove_vowels` that receives a string as a parameter and returns the string with vowels removed. You will have to use the global variable inside the function. Call the function on a multiline string defined outside the function.