Exploring Python Functions

Estimated time needed: 10 minutes

Lab Objectives:

By the end of this reading, you should be able to:

1. Understand the concept and importance of functions in programming.
2. Learn how function takes inputs and perform tasks
3. Use built-in functions like len(), sum(), and others effectively.
4. Define and use your own functions in Python.
5. Differentiate between global and local variable scopes.
6. Learn how to use loops within the function.
7. Modify data structures using functions.

Introduction to Functions:

A function is a fundamental building block that encapsulates a specific set of actions or computations. Just as in mathematics, where functions take inputs and produce outputs, programming functions perform a similar role. They take inputs, execute a series of predefined actions or calculations, and then return an output.

Purpose of Functions:

The primary purpose of functions is to promote code modularity and reusability. Imagine you have a task that needs to be performed multiple times within a program. Instead of duplicating the same code at various places, you can define a function once and call it whenever that task is needed. This not only reduces redundancy but also makes the code easier to manage and maintain.

Benefits of Using Functions:

Modularity: Functions break down complex tasks into manageable components.
Reusability: Functions can be used multiple times without rewriting code.
Readability: Functions with meaningful names enhance code understanding.
Debugging: Isolating functions eases troubleshooting and issue fixing.
Abstraction: Functions simplify complex processes behind a user-friendly interface.
Collaboration: Team members can work on different functions concurrently.
Maintenance: Changes made in a function automatically apply wherever it's used.

How Functions Take Inputs, Perform Tasks, and Produce Outputs:

Inputs (Parameters):

Functions are designed to operate on data, and they can receive data as input. These inputs are known as parameters or arguments. Parameters provide functions with the necessary information they need to perform their tasks. Think of parameters as values that you pass to a function, allowing it to work with specific data.

Performing Tasks:

Once a function receives its input (parameters), it executes a set of predefined actions or computations. These actions can include calculations, operations on data, or even more complex tasks. The purpose of a function determines the tasks it performs. For instance, a function could calculate the sum of numbers, sort a list, format text, or fetch data from a database.

Producing Outputs:

After performing its tasks, a function can produce an output. This output is the result of the operations carried out within the function. It's the value that the function “returns” to the code that called it. Think of the output as the end product of the function's work. You can use this output in your code, assign it to variables, pass it to other functions, or even print it out for display.

Example:

Consider a function named `calculate_total` that takes two numbers as input (parameters), adds them together, performs the addition task, and then produces the sum as the output. Here's how it works:

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
```

```
1. def calculate_total(a, b):  # Parameters: a and b
```

```
    total = a + b           # Task: Addition
    return total            # Output: Sum of a and b

result = calculate_total(5, 7)  # Calling the function with inputs 5 and 7
print(result)  # Output: 12
```

Copied!

Python's Built-in Functions:

Python comes equipped with a rich set of built-in functions that provide a wide range of functionalities. These functions are readily available for you to use, and you don't need to be concerned about how they are implemented internally. Instead, you can focus on understanding what each function does and how to use it effectively.

Using Built-in Functions or Pre-defined Functions:

To use a built-in function, you simply call the function's name followed by parentheses. Any required arguments or parameters are passed within these parentheses. The function then performs its predefined task and may return an output that you can use in your code.

Here are a few examples of commonly used built-in functions:

len(): Calculates the length of a sequence or collection.

```
1
2
```

```
string_length = len("Hello, World!")  # Output: 13
list_length = len([1, 2, 3, 4, 5])   # Output: 5
```

Copied!

sum(): Adds up the elements in an iterable (list, tuple, etc.).

```
1
```

```
total = sum([10, 20, 30, 40, 50])  # Output: 150
```

Copied!

max(): Returns the maximum value in an iterable.

```
1
```

```
1. highest = max([5, 12, 8, 23, 16])  # Output: 23
```

Copied!

min(): Returns the minimum value in an iterable.

```
1. 1
```

```
1. lowest = min([5, 12, 8, 23, 16])  # Output: 5
```

Copied!

Python's built-in functions offer a wide array of functionalities, from basic operations like len() and sum() to more specialized tasks.

Defining Your Own Functions:

Defining functions is like creating your own mini-programs:

1. Use `def` followed by the function name and parentheses.

Here is the syntax of function:

```
1. 1
2. 2
```

```
1. def function_name():
2.     pass
```

Copied!

A "`pass`" statement in a programming function is a placeholder or a no-op (no operation) statement. It is used when you want to define a function or a code block syntactically but do not want to specify any functionality or implementation at that moment.

- **Placeholder:** "`pass`" acts as a temporary placeholder for future code that you intend to write within a function or a code block.
- **Syntax Requirement:** In many programming languages like Python, using "`pass`" is necessary when you define a function or a conditional block. It ensures that the code remains syntactically correct, even if it doesn't do anything yet.
- **No Operation:** "`pass`" itself doesn't perform any meaningful action. When the interpreter encounters “pass”, it simply moves on to the next statement without executing any code.

Function Parameters:

- Parameters are like inputs for functions.
- They go inside parentheses when defining the function.
- Functions can have multiple parameters.

Example:

```
1
2
3
4
5
```

```
def greet(name):
    print("Hello, " + name)

result = greet("Alice")
print(result)  # Output: Hello, Alice
```

Copied!

Docstrings (Documentation Strings):

- Docstrings explain what a function does.
- Placed inside triple quotes under the function definition.
- Helps other developers understand your function.

Example:

```
1
2
3
4
5
6
7
8
```

```
def multiply(a, b):
    """
    This function multiplies two numbers.
    Input: a (number), b (number)
    Output: Product of a and b
    """
    print(a * b)
```

```
8. multiply(2,6)
```

Copied!

Return Statement:

- Return gives back a value from a function.
- Ends the function's execution and sends the result.
- A function can return various types of data.

Example:

```
1. 1
2. 2
3. 3
4. 4
```

```
1. def add(a, b):
2.     return a + b
3.
4. sum_result = add(3, 5)  # sum_result gets the value 8
```

Copied!

Understanding Scopes and Variables:

Scope is where a variable can be seen and used:

- **Global Scope:** Variables defined outside functions; accessible everywhere.
- **Local Scope:** Variables inside functions; only usable within that function.

Example:

Part 1: Global Variable Declaration

```
1. 1
```

```
1. global_variable = "I'm global"
```

Copied!

This line initializes a global variable called `global_variable` and assigns it the value “I’m global”.

Global variables are accessible throughout the entire program, both inside and outside functions.

Part 2: Function Definition

```
1
2
3
4
```

```
def example_function():
    local_variable = "I'm local"
    print(global_variable)  # Accessing global variable
    print(local_variable)   # Accessing local variable
```

Copied!

Here, we define a function called `example_function()`.

Within this function:

- A local variable named local_variable is declared and initialized with the string value “I'm local.” This variable is local to the function and can only be accessed within the function's scope.
- The function then prints the values of both the **global variable (global_variable) and the local variable (local_variable)**. It demonstrates that you can access both global and local variables from within a function.

Part 3: Function Call

```
1
```

```
example_function()
```

Copied!

In this part, we call the `example_function()` by invoking it. This results in the function's code being executed.
As a result of this function call, it will print the values of the global and local variables within the function.

Part 4: Accessing Global Variable Outside the Function

```
1
```

```
print(global_variable)  # Accessible outside the function
```

Copied!

After calling the function, we print the value of the global variable `global_variable` outside of the function. **This demonstrates that global variables are accessible both inside and outside of functions.**

Part 5: Attempting to Access Local Variable Outside the Function

```
1
```

```
# print(local_variable)  # Error, local variable not visible here
```

Copied!

In this part, we are attempting to print the value of the local variable `local_variable` outside of the function. However, this line would result in an error.

> Local variables are only visible and accessible within the scope of the function where they are defined.

Attempting to access them outside of that scope would raise a `"NameError"`.

Using Functions with Loops:

Functions and Loops Together:

1. Functions can contain code with loops.
2. This makes complex tasks more organized.
3. The loop code becomes a repeatable function.

Example:

```
1
2
3
4
5
```

```
def print_numbers(limit):
    for i in range(1, limit+1):
        print(i)

print_numbers(5)  # Output: 1 2 3 4 5
```

Copied!

Enhancing Code Organization and Reusability:

1. Functions group similar actions for easy understanding.
2. Looping within functions keeps code clean.
3. You can reuse a function to repeat actions.

Example

```
1
2
3
4
5
```

```
def greet(name):
    return "Hello, " + name

for _ in range(3):
    print(greet("Alice"))
```

Copied!

Modifying data structure using functions

We'll use Python and a list as the data structure for this illustration. In this example, we will create functions to add and remove elements from a list.

Part 1: Initialize an Empty List

```
1
2
```

```
# Define an empty list as the initial data structure
my_list = []
```

Copied!

In this part, we start by creating an empty list named `my_list`. This empty list serves as the data structure that we will modify throughout the code.

Part 2: Define a Function to Add Elements

```
1
2
```

```
3. 3
```

```
1. # Function to add an element to the list
2. def add_element(data_structure, element):
3.     data_structure.append(element)
```

Copied!

Here, we define a function called `add_element`. This function takes two parameters:

- `data_structure`: This parameter represents the list to which we want to add an element.
- `element`: This parameter represents the element we want to add to the list.

Inside the function, we use the `append` method to add the provided element to the data_structure, which is assumed to be a list.

Part 3: Define a Function to Remove Elements

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
```

```
1. # Function to remove an element from the list
2. def remove_element(data_structure, element):
3.     if element in data_structure:
4.         data_structure.remove(element)
5.     else:
6.         print(f"{element} not found in the list.")
```

Copied!

In this part, we define another function called `remove_element`. It also takes two parameters:

- `data_structure`: The list from which we want to remove an element.
- `element`: The element we want to remove from the list.

Inside the function, we use conditional statements to check if element is present in the data_structure. If it is, we use the `remove` method to remove the first occurrence of element. If it's not found, we print a message indicating that the element was not found in the list.

Part 4: Add Elements to the List

```
1. 1
2. 2
3. 3
4. 4
```

```
1. # Add elements to the list using the add_element function
2. add_element(my_list, 42)
3. add_element(my_list, 17)
4. add_element(my_list, 99)
```

Copied!

Here, we use the `add_element` function to add three elements (42, 17, and 99) to the `my_list`. These are added one at a time using function calls.

Part 5: Print the Current List

```
1. 1
2. 2
```

```
1. # Print the current list
2. print("Current list:", my_list)
```

Copied!

This part simply prints the current state of the `my_list` to the console, allowing us to see the elements that have been added so far.

Part 6: Remove Elements from the List

```
1. 1
2. 2
3. 3
```

```
1. # Remove an element from the list using the remove_element function
2. remove_element(my_list, 17)
3. remove_element(my_list, 55)  # This will print a message since 55 is not in the list
```

Copied!

In this part, we use the `remove_element` function to remove elements from the my_list. First, we attempt to remove 17 (which is in the list), and then we try to remove 55 (which is not in the list). **The second call to remove_element will print a message indicating that 55 was not found.**

Part 7: Print the Updated List

```
1. 1
2. 2

1. # Print the updated list
2. print("Updated list:", my_list)
```

Copied!

Finally, we print the updated `my_list` to the console. This allows us to observe the modifications made to the list by adding and removing elements using the defined functions.

Conclusion:

Congratulations! You've completed the Reading Instruction Lab on Python functions. You've gained a solid understanding of functions, their significance, and how to create and use them effectively. These skills will empower you to write more organized, modular, and powerful code in your Python projects.

Author

Akansha Yadav

Changelog

Date	Version	Changed by	Change Description
2023-08-24	1.0	Akansha Yadav	Created reading