Chapter 3 : Functions

Aviral Janveja

1 Introduction

Till now, we have covered the basic language mechanisms of Python. We are now able to write different code files for different computations, where each file contains some program and each program is a sequence of instructions. Although, this approach works fine for small-scale programs, things would get really messy for larger programs.

This is where functions come in. In Python, a function is a block of reusable code that performs a specific task. For instance, a function for finding if a number is even or not. Functions provide a way to keep our code organized into reusable and modular components as we scale our programs. Functions also provide the mechanism to achieve **decomposition** and **abstraction** in Python, while significantly simplifying the **debugging** process as you just need to debug once and then run anywhere.

1.1 Abstraction

In Python, functions can be considered a form of abstraction. A well-designed function abstracts away the implementation details, providing a clear interface for the user. Users of the function don't need to understand how the function achieves its task; they only need to know how to use it. For example, you don't need to know how a laptop really works at the micro-electronic level in order to use it.

1.2 Decomposition

Decomposition is the process of breaking down a complex problem into smaller, more manageable sub-problems or tasks. Each sub-problem can be solved independently, making it easier to understand and implement.

Functions play a crucial role in decomposition. By breaking down a problem into smaller functions, each responsible for a specific task, you can tackle individual components without dealing with the entire complexity at once. This leads to more organized, maintainable and reusable code.

2 Function Definition & Call

Here is the basic function definition and how to call one :

```
def function_name(parameters):
    # Function body
    # Code to perform a specific task
    return result  # Optional

# Calling the above function.
function_name(arguments)
```

Let us break down the above syntax:

- **def** keyword is used to define a function.
- **function_name** is the name of the function. Choose a descriptive and meaningful name that reflects the purpose of the function.
- **Parameters** are the variables listed in the function definition, acting as placeholders for the values that will be passed in. **Arguments** are the actual values that are passed to the function when it is called.
- **Function body** contains the code to be executed when the function is called. The body is indented, typically by four spaces.
- **return** keyword is used to specify the value that the function returns. A function may not have a return statement, in which case it returns the type **None** by default.

Checkout the following example to see how to define and call a function in Python : function_example.py

3 Scope of a Function

In Python, the scope of a function refers to the frame of the program where the variables and parameters defined within that function can be accessed. It determines the visibility and lifetime of the variables within the function.

3.1 Local Scope

Variables defined inside a function have local scope. They can only be accessed within the function in which they are defined. They are created when the function is called. Once the function execution completes, local variables are destroyed and their memory is released.

You can define a local variable by assigning a value to it inside a function.

```
def my_function():
    x = 10  # Local variable
    print(x)  # Accessible only within the function

my_function()
print(x)  # Error: NameError: name 'x' is not defined
```

3.2 Global Scope

Global variables are defined outside of any function, usually at the top level of the script or module. They have a global scope, which means they can be accessed from anywhere within the script or module, including inside functions.

You can define a global variable by simply assigning a value to it outside of any function.

```
# Global variable
z = 30

def my_function():
    print(z)  # Accessible within the function

my_function()
```

When you reference a variable within a function, Python first checks if it's a local variable. If not found, it looks for it in the global scope. You can check the following example that demonstrates the same : function_scope.py

However, you need to use the **global** keyword to modify a global variable from inside a function. Trying to modify a global variable without using the global keyword leads to an error. For example :

```
x = 5

def test_func(y):
    x += 1

test_func(x)
print(x)
```

This leads to an **UnboundLocalError**. This error occurs because Python treats the variable being modified within the function as a local variable unless explicitly declared as global. As it was not declared and bound locally to a value before trying to modify it, hence "Unbound Local" error.

Further, modifying global variables directly from within functions is generally discouraged due to readability, maintainability and debugging concerns.

3.3 Functions As Arguments

In Python, it is also possible to pass functions as arguments to other functions, just like any other data type such as integers or strings. This example demonstrates how functions can be passed as arguments to other functions, allowing for dynamic behavior and code re-usability : function_as_argument.py

4 Nested Functions

Nested functions are functions defined inside other functions. They have access to variables and parameters from the enclosing function's scope and they can also

access global variables. Nested functions are useful for encapsulating functionality that is only needed within the context of the outer function.

Here's a simple example of nested functions : nested_functions.py

This example illustrates how nested functions can be used to encapsulate functionality and share variables from the outer function's scope.