Python Namespace and Scope

To simply put it, a namespace is a collection of names.

In Python, we can imagine a namespace as a mapping of every name we have defined to corresponding objects.

It is used to store the values of variables and other objects in the program, and to associate them with a specific name.

This allows us to use the same name for different variables or objects in different parts of your code, without causing any conflicts or confusion.

Types of Python namespace

A namespace containing all the **built-in names** is created when we start the Python interpreter and exists as long as the interpreter runs.

This is the reason that built-in functions like id(), print() etc. are always available to us from any part of the program. Each module creates its own **global namespace.**

These different namespaces are isolated. Hence, the same name that may exist in different modules does not collide.

Modules can have various functions and classes. A **local namespace** is created when a function is called, which has all the names defined in it.

Similar is the case with class. The following diagram may help to clarify this concept.

Python Namespaces

Python Variable Scope

Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program. The concept of scope comes into play.

A scope is the portion of a program from where a namespace can be accessed directly without any prefix.

At any given moment, there are at least three nested scopes.

1. Scope of the current function which has local names
2. Scope of the module which has global names
3. Outermost scope which has built-in names

When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace.

If there is a function inside another function, a new scope is nested inside the local scope.

Example 1: Scope and Namespace in Python

```
# global_var is in the global namespace
global_var = 10
```

```
def outer_function():
    #  outer_var is in the local namespace
    outer_var = 20

    def inner_function():
        #  inner_var is in the nested local namespace
        inner_var = 30

        print(inner_var)

    print(outer_var)

    inner_function()

# print the value of the global variable
print(global_var)

# call the outer function and print local and nested local variables
outer_function()
```

Output

```
10
20
30
```

In the above example, there are three separate namespaces: the global namespace, the local namespace within the outer function, and the local namespace within the inner function.

Here,

- *global_var* - is in the global namespace with value **10**
- *outer_val* - is in the local namespace of `outer_function()` with value **20**
- *inner_val* - is in the nested local namespace of `inner_function()` with value **30**

When the code is executed, the *global_var* global variable is printed first, followed by the local variable: *outer_var* and *inner_var* when the outer and inner functions are called.

Example 2: Use of global Keyword in Python

```
# define global variable
global_var = 10

def my_function():
    # define local variable
    local_var = 20

    # modify global variable value
    global global_var
    global_var = 30

# print global variable value
print(global_var)

# call the function and modify the global variable
my_function()

# print the modified value of the global variable
print(global_var)
```

Output

```
10
30
```

Here, when the function is called, the global keyword is used to indicate that *global_var* is a global variable, and its value is modified to **30**.

So, when the code is executed, *global_var* is printed first with a value of **10**, then the function is called and the global variable is modified to **30** from the inside of the function.

And finally the modified value of *global_var* is printed again.

Also Read:

- Python Variable Scope

Table of Contents