Python Variable Scopes

Summary: in this tutorial, you'll learn how Python variable scopes work. After the tutorial, you'll have a good understanding of built-in, local, and global scopes.

Introduction to Python variable scopes

When you assign an object to a variable (https://www.pythontutorial.net/python-basics/python-variables/) , the variable will reference (https://www.pythontutorial.net/advanced-python/python-references/) that object in the memory. And it's saying that the variable is bound to the object.

After the assignment, you can access the object using the variable name in various parts of your code. However, you cannot access the variable everywhere in the code.

The variable name and its binding (name and object) only exist in specific parts of your code.

The part of the code where you define the name/binding is called the **lexical scope** of the variables.

Python stores these bindings in something called namespaces. Every scope has its own namespace.

And you can think that a namespace is a table which contains the label and the reference that the label is bound to.

Global scopes

The global scope is basically the module (https://www.pythontutorial.net/python-basics/python-module/) scope. The global scope spans a single Python source code file only.

Python doesn't have a truly global scope that spans across all modules except for the built-in scope.

The built-in scope is a special scope that provides globally available objects such as `print` , `len` , `None` , `True` , and `False` .

Basically, the built-in and global variables exist everywhere inside a module.

Internally, global scopes are nested inside the built-in scope:

If you access a variable from a scope and Python doesn't find it in the namespace of that scope, it'll search in the enclosing scope's namespace.

Suppose that you have the following statement in a module called `app.py` :

```
print('Hello')
```

In this `app.py` module, Python looks for the `print` function in the module scope (`app.py`).

Since Python doesn't find the definition of the `print` function in the `app.py` module scope, Python goes up to the enclosing scope, which is the built-in scope, and looks for the `print` function there. In this case, it can find the `print` function in the built-in scope.

If you change the statement to the following, you'll get an runtime error:

```
print(counter)
```

In this example, Python doesn't find the `counter` in the current global scope. Therefore, Python looks for it in the enclosing scope, which is the built-in scope.

However, the variable `counter` doesn't exist in the built-in scope. Therefore, Python issues a `NameError` exception:

```
NameError: name 'counter' is not defined
```

Local scopes

When creating a function (https://www.pythontutorial.net/python-basics/python-functions/) , you can define parameters and variables for the function. For example:

```
def increment(counter, by=1):
    result = counter + by
    return result
```

When you execute the code, Python carries two phases: compilation and execution.

When Python compiles the file, it adds the `increment` function to the global scope. In addition, Python determines that the `counter` , `by` , and `result` variables inside the `increment()` function will be local to the `increment()` function. And Python won't create the `counter` , `by` and `result` variables until the function is executed.

Every time you call a function, Python creates a new scope. Python also assigns the variables defined inside the function to that scope. And this scope is called a function local scope or local scope.

In our example, when you call the `increment()` function:

```
increment(10,2)
```

... Python creates a local scope for the `increment()` function call.

Also, Python creates local variables `counter` , `by` , and `result` in the local namespace and binds them to values `10` , `2` , and `12` .

When the function completes, Python will delete the local scope. And all the local variables such as `counter` , `by` , and `result` variables are out of scope. If you attempt to access these variables from outside the `increment()` function, you'll get an error.

And if you call the `increment()` function again:

```
increment(100,3)
```

… Python creates a new local scope and variables including `counter` , `by` and `result` , and binds them to values `100` , `3` , and `103` .

Variable lookups

In Python, scopes are nested. For example, local scopes are nested inside a module scope. And module scopes are nested inside the built-scope:

When you access an object bound to a variable, Python tries to find the object:

- in the current local scope first.
- and goes up the chain of enclosing scopes if Python doesn’t find the object in the current scope.

The global keyword

When you retrieve the value of a global variable from inside a function, Python automatically searches the local scope’s namespace and up the chain of all enclosing scope namespaces. For example:

```
counter = 10


def current():
    print(counter)


current()
```

In this example, when the `current()` function is running, Python looks for the `counter` variable in the local scope.

Since Python doesn't find it, it searches for the variable in the global scope. And Python can find the `counter` variable in the global scope in this case.

However, if you assign a value to a global variable from inside a function, Python will place that variable into the local namespace instead. For example:

```
counter = 10


def reset():
    counter = 0
    print(counter)


reset()
print(counter)
```

Output:

```
0
10
```

At the compile time, Python interprets the `counter` as a local variable.

When `reset()` function is running, Python finds the `counter` in the local scope. The `print(counter)` statement inside the `reset()` function shows the value of the `counter` , which is zero.

When we print `counter` after the `reset()` function completes, it shows 10 instead.

In this example, the local `counter` variable masks the global `counter` variable.

If you want to access a global variable from inside a function, you can use the `global` keyword. For example:

```
counter = 10


def reset():
    global counter
    counter = 0
    print(counter) # 0


reset()

print(counter) # 0
```

Output:

```
0
0
```

In this example, the following statement:

```
global counter
```

…instructs Python that the `counter` variable is bound to the global scope, not the local scope.

Note that it's not a good practice to access the global variable inside a function.

Summary

- The scopes of variables are the parts of the code where you can access the variables.
- The built-in scope is accessible from everywhere.
- The global scope (or module scope) can be accessible from every part of the module.
- The local scope is accessible from inside a function.
- Python stores the objects and their bindings in the namespace of the scope.
- Python looks up an object in the current scope first and goes up to the enclosing scope if Python doesn't find it.
- Python scopes are nested.
- Use the `global` keyword if you want to access a global variable from inside a function.