Scope

by Sophia

WHAT'S COVERED

In this lesson, we'll be looking at global and local scope and their implications. Specifically, this lesson covers:

1. Global and Local Scope

We briefly discussed scope when we looked at scope rules dealing with variables and functions in Unit 2. We explained that variables that are defined inside the body of a function have local scope and those defined outside of a function body have global scope.

Here was our first example with scope and using global and local variables.

EXAMPLE

```
total = 0;

def sum(arg1,arg2):
   total = arg1 + arg2;
   print ("Local total: ",total)
   return total;

sum(10,20);
print ("Global total : ", total)
```

We see the variable `total` being set to 0 before the function `sum`. Then, in that function, it takes arguments being passed in that are integers. It does some adding of those values to a variable inside the function also called `total`. Then, with the output, we see the following.

```
Local total: 30
```

```
Global total : 0
```

So, the local variable was changed with the arithmetic while the global variable was not affected.

The whole concept around scope determines how the names of objects in our code are deciphered by Python. The level of scope can determine the visibility of an object's name, so where we place an object is very important.

Scope levels in Python determine the sequence of steps that Python uses to resolve any name issues in a program. This includes variables, methods, functions, or class names that may be used in different scopes.

We can take advantage of the scopes that Python offers to create more reliable programs. By using Python scope, we can help avoid errors and minimize bugs such as name collisions. **Name collisions** occur when we have object names used in multiple scopes that mean different things.

For example, in your household the word "dog" might refer to your dog, and in a dog park "dog" could refer to any dog at the park. In conversation, to "dog someone" means to criticize someone. So the name dog can reference different things in different contexts.

The very basic meaning of scope is an area of a program where we can access a name, whether it's a variable, function or an object. That name is only visible in the code within that scope. The two general scopes that we most frequently hear about are global scope and local scope. We defined global scope as variables that can be accessed throughout the entire program. Global scope encompasses names that we define that are available to all of our code. We defined local scope as variables that can only be accessed within the block, function, method, or class that they are declared. So, local scope covers names that are defined in a specific area in the program and are only visible within that area.

DID YOU KNOW

The use of scope came about because early programming languages only had "global" names. With this global level type of naming, any part of the program could change any variable at any time. As programs got larger, debugging and maintaining the code became quite a challenge. To work with purely global names, we would need to keep in mind what the value of any given name was at any time. Therefore, Python uses the concept of scope to avoid that issue. When we use a language that implements scope, we can only access a given name* depending on where we've defined a specific name*.

*Note: We use the term "name" here as it refers to variables, functions, classes or other objects that could be assigned a name.

The names that we have in our programs will have the scope of the block of code that we define for them. When we can access the value of a given name in our code, we'll say that name is in scope. If we can't access that name, we'll say that the name is out of scope; we will get an error if we try to access that name. We will discuss an out-of-scope scenario below.

Variables in Python come into existence when we first assign them a value like the following:

```
first_name = 'sophia'
```

Our variable name is `first_name`.

Functions are available when we define them using the `def` keyword.

```
def my_function (parameter(s):
```

Our function name is `my_function()`.

Classes are available when we define them with the `class` keyword.

```
Class Employee:
```

Our class name is `Employee`.

Modules exist when we import them. All of these types of operations create or update names as all of them will assign a name to a variable, function, class or another Python object.

Python uses the location of the name assignment or definition to associate it with a particular scope. This means that where we assign or define a name in our code determines the visibility of that name. As an example, if we assign a value to a name inside of a function or method, then that name will have the local scope of that function. If we assign a value to a name outside of all functions, then that name will have a global scope.

Let's see another example of local vs. global scope. Let's do another test to see what that looks like.

↷ EXAMPLE

```
#global scope
first_name = 'Global'

def display_name():
  #local scope
  last_name = 'Local'
  return f'{first_name} is a global variable. {last_name} is a local variable.'

print(display_name())
```

In the global scope, we'll define a variable called `first_name` and set it to a string "Global". In the local scope, we'll define a variable called `last_name` and set it to string "Local". In the function `display_name`, we can access the global variable and the local variable. If we run this as is with the `print()` function, we should get the following output.

```
Global is a global variable. Local is a local variable.
```

However, if we tried to access the `last_name` variable outside of the function, we would get an error.

↷ EXAMPLE

```
#global scope
first_name = 'Global'

def display_name():
  #local scope
  last_name = 'Local'
  return f'{first_name} is a global variable. {last_name} is a local variable.'

print(display_name())
print(last_name)
```

Here is the output with the error.

```
Global is a global variable. Local is a local variable.
Traceback (most recent call last):
  File "/home/main.py", line 10, in <module>
    print(last_name)
NameError: name 'last_name' is not defined. Did you mean: 'first_name'?
```

This is because in our case, the `last_name` variable is a local variable that has the scope within the `display_name()` function. It does not exist outside of that function and it is now out of scope. It's important to note that the `last_name` variable that's defined in the function is created when the function is called and not when the function is defined. This means that we can have different local scopes when the functions are called each time.

By default, parameters, and names we assign inside a function only exist inside the function or local scope associated with the function call. When the function returns, the local scope is destroyed, and the names are all lost. Therefore, we could not access the `last_name` variable outside of the function as we have shown above.

The global scope exists when we first start any Python program. We can access the value of any global name from any place in the code. Internally in Python, it turns our program's main code into a module called `__main__` to hold the program's execution. The **`__main__` function** is the name of the top-level program in Python. The top-level is the first user-specified module that runs when a program is being executed. So, the top-level program is the entry point to the executed code and it imports all other modules that the program needs. This is automatically created by Python and is not directly defined unless your program has multiple files in which we want to define where the entry point of the program is. As an example, a house can have multiple entryways (doors) and if you wanted to direct traffic through a single door, you would have something indicating what door to use. We will make use of this function in Unit 4.

Global scope only executes once per program execution. This global scope stays in existence until the program terminates and all of the names are forgotten. Otherwise, the next time we run the program, the names would remember their values from the previous time it was executed.

Let's look at the following code again.

EXAMPLE

```
#global scope
first_name = 'Global'

def display_name():
  #local scope
  last_name = 'Local'
  return f'{first_name} is a global variable. {last_name} is a local variable.'

print(display_name())
print(first_name)
```

Here is that output.

```
Global is a global variable. Local is a local variable.
Global
```

The variable `first_name` is accessible within the function as well as outside of the function when we print it. Let's see what happens if we change the `first_name` inside of the function.

EXAMPLE

```
#global scope
first_name = 'Global'

def display_name():
  #local scope
  last_name = 'Local'

  #changing global name
  first_name = 'ChangeMe'
  return f'{first_name} is a global variable. {last_name} is a local variable.'

print(display_name())
print(first_name)
```

Now, with this output:

```
ChangeMe is a global variable. Local is a local variable.
Global
```

Oops, notice that the `first_name` variable returned `ChangeMe` but afterwards, in the global scope, `first_name` still had the value of "Global". This is because inside of the `display_name()` function, a new local scope of `first_name` was created that's separate from the one used globally. If we wanted the function

to make use and update the global scope name instead, we need to define it using the `global` keyword with the name. The reserved keyword **global** allows the modification of a variable outside of the local scope.

EXAMPLE

```
#global scope
first_name = 'Global'

def display_name():
  #local scope
  last_name = 'Local'

  #changing global name
  global first_name
  first_name = 'ChangeMe'
  return f'{first_name} is a global variable. {last_name} is a local variable.'

print(display_name())
print(first_name)
```

Notice with this scenario, the `first_name` at the global level was used and updated.

```
ChangeMe is a global variable. Local is a local variable.
ChangeMe
```

By doing this, the `display_name()` function didn't create a new name in the local scope, but rather updated the value in the global scope.

When we declare global names, these names stay in memory for the entire program execution. If we only use the global name in one or two places in the program, it may simply be wasted space. As programs get larger, the use of global names, in general, is a bad practice because those global names are kept in memory. For example, when you are in a classroom, you may remember the names of your classmates. In comparison, with global names, you know you have to remember your classmates' names for every single classroom you have been in.

TRY IT

Directions: Feel free to try any of the global vs. local scope examples above in the IDE.

TERMS TO KNOW

Name Collisions

Name collisions occur when similar named objects are used in multiple scopes that mean different things.

__main__

The `__main__` function is the name of the top-level program in Python. The top level is the first user specified module that runs when a program is being executed.

global

The reserved keyword `global` allows the modification of a variable outside of the current scope.

SUMMARY

In this lesson, we learned that the scope of a name defines where that name is accessible throughout our code, as the term “name” refers to variables, functions, classes or other objects that could be assigned a name. In Python, scope is implemented as local or global. If a name is not found in the local scope or the global scope, we’ll get an error. We also looked at how to define and update variables at both a **local and global scope**.

Best of luck in your learning!

Source: THIS CONTENT AND SUPPLEMENTAL MATERIAL HAS BEEN ADAPTED FROM “PYTHON FOR EVERYBODY” BY DR. CHARLES R. SEVERANCE ACCESS FOR FREE AT www.py4e.com/html3/ LICENSE: CREATIVE COMMONS ATTRIBUTION 3.0 UNPORTED.

TERMS TO KNOW

Name Collisions

Name collisions occur when similar named objects are used in multiple scopes that mean different things.

__main__

The `__main__` function is the name of the top-level program in Python. The top level is the first user specified module that runs when a program is being executed.

global

The reserved keyword `global` allows the modification of a variable outside of the current scope.