Chapter 12
Scope and Lifetime of Variables

12.1 Introduction

We have already defined several variables in the examples we have being working with in this book. In practice, most of these variables have been what are known as *global* variables. That is there are (potentially) accessible anywhere (or globally) in our programs.

In this chapter we will look at *local* variables as defined within a function, at global variables and how they can be referenced within a function and finally we will consider *nonlocal* variables.

12.2 Local Variables

In practice developers usually try to limit the number of global variables in their programs as global variables can be accessed anywhere and can be modified anywhere and this can result in unexpected behaviours (and has been the cause of many, many bugs in all sorts of programs over the years).

However, not all variables are global. When we define a function, we can create variables which are scoped only to that function and are not accessible or visible outside of the function. These variables are referred to as local variables (as they are local to the function).

This is a great help in developing more modular code which has been proven to be easier to maintain and in fact develop and test.

In the following function local variable called `a_variable` has been created and initialised to hold the value 100.

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_12

```
def my_function():
    a_variable = 100
    print(a_variable)

my_function()
```

When this function is called `a_variable` will be initialised to 100 and will then be printed out to the console:

```
100
```

Thus when we ran the `my_function()` it successfully printed out the value `100` which was held in the local (to the function) variable `a_variable`.

However if we attempt to access `a_variable` outside the function, then it will not be defined and we will generate an error, for example:

```
my_function()
print(a_variable)
```

When we run this code, we get the number 100 printed out from the call the `my_function()`. However, an error is then reported by Python:

```
100
Traceback (most recent call last):
  File "localvars.py", line 7, in <module>
    print(a_variable)
NameError: name 'a_variable' is not defined
```

This indicates that `a_variable` is *undefined* at the top level (which is the global scope). Thus, we can say that `a_variable` is not globally defined.

This is because `a_variable` only exists and only has meaning inside `my_function`; outside of that function it cannot be seen.

In fact, each time the function is called, `a_variable` comes back into existence as a *new* variable, so the value in `a_variable` is not even seen from one invocation of the function to another.

This raises the question what happens if a *global* variable called `a_variable` is defined? For example, if we have the following:

```
a_variable = 25
my_function()
print(a_variable)
```

Actually, this is fine and is supported by Python. There are now two versions of `a_variable` in the program; one of which is defined *globally* and one of which is defined within the *context* of the function.

Python does not get confused between these and treats then as completely separately. This is just like having two people called *John* in the same class in school. If they were only called John this might cause some confusion, but if they have different surnames then it is easy to distinguish between them via their full names such as *John Jones* and *John Smith*.

In this case we have *global* a_variable and my_function a_variable. Thus if we run the above code we get

```
100
25
```

The value 100 does not overwrite the value 25 as they are completely different variables.

12.3 The Global Keyword

But what happens if what you want is to reference the global variable within a function.

As long as Python does not think you have defined a local variable then all will be fine. For example

```
max = 100
def print_max():
    print(max)
print_max()
```

This prints out the value 100.

However, things go a bit astray if you try to modify the global variable inside the function. At this point Python thinks you are creating a local variable. If as part of the assignment you try to reference the current value of that (now) local variable you will get an error indicating that it currently does not have a value. For example, if we write:

```
def print_max():
    max = max + 1
    print(max)
print_max()
```

And then run this example, we will get

```
Traceback (most recent call last):
  File "localvars.py", line 17, in <module>
    print_max()
  File "localvars.py", line 14, in print_max
    max = max + 1
UnboundLocalError: local variable 'max' referenced before
assignment
```

Indicating that we have referenced max before it was assigned a value—even though it was assigned a value *globally* before the function was called!

Why does it do this? To protect us from ourselves—Python is really saying 'Do you really want to modify a global variable here?'. Instead it is treating max as a *local* variable and as such it is being referenced before a value has been assigned to it.

To tell Python that we know what we are doing and that we want to reference the global variable at this point we need to use the keyword global with the name of the variable. For example:

```
max = 100

def print_max():
    global max
    max = max + 1
    print(max)

print_max()
print(max)
```

Now when we try to update the variable max inside the function print_max(), Python knows we mean the *global* version of the variable and uses that one. The result is that we now print out the value 101 and max is updated to 101 for everyone everywhere!

12.4 Nonlocal Variables

It is possible to define functions inside other functions, and this can be very useful when we are working with collections of data and operations such as map() (which maps a function to all the elements of a collection of data).

However, local variables are local to a specific function; even functions defined within another function cannot modify the outer functions local variables (as the inner function is a separate function). They can reference it, just as we could reference the global variable earlier; the issue is again modification.

The global keyword is no help here as the outer function's variables are not global, they are local to a function.

For example, if we define a nested function (inner) inside the parent outer function (outer) and want the inner function to modify the local field we have a problem:

```
def outer():
    title = 'original title'

    def inner():
        title = 'another title'
        print('inner:', title)

    inner()
    print('outer:', title)

outer()
```

In this example both `outer()` and `inner()` functions modify the `title` variable. However, they are not the same `title` variable and as long as this is what we need then that is fine; both functions have their own version of a `title` local variable.

This can be seen in the output where the outer function maintains its own value for `title`:

```
inner: another title
outer: original title
```

However, if what we want is for the `inner()` function to modify the `outer()` function's `title` variable then we have a problem.

This problem can be solved using the `nonlocal` keyword. This indicates that a variable is not global but is also not local to the current function and Python should look within the scope in which the function is defined to fund a local variable with the same name:

If we now declare `title` as `nonlocal` in the `inner()` function, then it will use the `outer()` functions version of `title` (it will be shared between them) and thus when the `inner()` function changes the `title` it will change the it for both functions:

```
def outer():
    title = 'original title'
    def inner():
        nonlocal title
        title = 'another title'
        print('inner:', title)
    inner()
    print('outer:', title)

outer()
```

The result of running this is

```
inner: another title
outer: another title
```

12.5 Hints

Points to note about the scope and lifetime of variables

1. The scope of a variable is the part of a program where the variable is known. Parameters and variables defined inside a function are not visible from outside. Hence, they have a local scope.
2. The lifetime of a variable is the period throughout which the variable exits in the memory of your Python program. The lifetime of variables inside a function is as long as the function executes. These local variables are destroyed as soon as the function returns or terminates. This means that the function does not store the values in a variable from one invocation to another.

12.6 Online Resources

See the Python Standard Library documentation for:

- https://docs.python.org/3/faq/programming.html#what-are-the-rules-for-local-and-global-variables-in-python which provides further information on the Python rules for local and global variables.

12.7 Exercise

Return to the number guess game—did you have to make any compromises with the variables to overcome the global variable issue? If so can you resolve them now with the use of the `global`?