2.3 Local Variables and Function Scope

One of the key purposes of functions is to separate different computations in a program, so that we don't have to worry about them all at once. When we write our code in separate functions, we can focus on working with just a single function, and ignore the rest of the code in other functions.

One way in which Python supports this way of designing programs is through separating the variables in each functions so that *a function call can only access its own variables, but not variables defined within other functions*. In this section, we'll explore how this works, learning more about how Python keep track of function calls and variables.

Example 1: introducing local variable scope

Consider the `square` example from Section 2.2:

```
def square(x: float) -> float:
    """Return x squared.

    >>> square(3.0)
    9.0
    >>> square(2.5)
    6.25
    """
    return x ** 2
```

The parameter `x` is a *variable* that is assigned a value based on when the function was called. Because this variable is only useful inside the function body, Python does not allow it to be accessed from outside the body. We say that `x` is a **local variable** of `square` because it is limited to the function body. More formally, we define the **scope** of a variable to be the places in the program code where that variable can be accessed. A *local variable* of a function is a variable whose scope is the body of that function.

Let's illustrate by first creating a variable in the Python console, and then calling `square`.

```
>>> n = 10.0
>>> result = square(n + 3.5)
```

We know that when `square` is called, its argument expression `n + 3.5` is evaluated first, producing the value `13.5`, which is then assigned to the parameter `x`. Now let's consider what our value-based memory model looks like when the `return` statement inside the body of `square` is executed. A naive memory model diagram would simply show the two variables `n` and `x` and their corresponding values:[1]

Variable	*Value*
`n`	`10.0`
`x`	`13.5`

[1] We do not show `result` because it hasn't been assigned a value yet; this only happens *after* `square` returns.

But this is very misleading! In our memory model diagrams, we group the variables together based on whether they are introduced in the Python console or inside a function:

`__main__` (Python console)

Variable	*Value*
`n`	`10.0`

`square`

Variable	*Value*
`x`	`13.5`

We use the name `__main__` to label the table for variables defined in the Python console.[2] Inside the body of `square`, the *only* variable that can be used is `x`, and outside, in the Python console, the *only* variable that can be used is `n`. This may seem a little tricky at first, but these memory model diagrams are a good way to visualize what's going on. At the point that the body of square is evaluated, only the "`square`" table in the memory model is active:

`__main__`

Variable	*Value*
`n`	`10.0`

`square`

Variable	*Value*
`x`	`13.5`

[2] This is a special name in Python—more on this later.

But after `square` returns and we're back to the Python console, the "`square`" table is no longer accessible, and only the `__main__` table is active:

`__main__`

Variable	*Value*
`n`	`10.0`
`result`	`182.25`

`square`

Variable	*Value*
`x`	`13.5`

Trying to access variable `x` from the Python console results in an error:

```
>>> n = 10.0
>>> square(n + 3.5)
182.25
>>> x
Traceback (most recent call last):
  ...
NameError: name 'x' is not defined
```

Example 2: duplicate variable names

The principle of "separate tables" in our memory model applies even when we use the same variable name in two different places. Suppose we modify our example above to use `x` instead of `n` in the Python console:

```
>>> x = 10.0
>>> result = square(x + 3.5)
```

Following the same reasoning as above, the argument expression `x + 3.5` is evaluated to produce `13.5`, which is then assigned to the parameter `x`. Does this modify the `x` variable in the Python console? No! They are different variables even though they share the same name.

`__main__`

Variable	*Value*
`x`	`10.0`

`square`

Variable	*Value*
`x`	`13.5`

We can confirm this after the function call is evaluated by checking the value of the original `x`.

```
>>> x = 10.0
>>> result = square(x + 3.5)
>>> result
182.25
>>> x
10.0
```

Here is what our memory model looks like after `square` has returned:

`__main__`

Variable	*Value*
`x`	`10.0`
`result`	`182.25`

`square` *(destroyed)*

Variable	*Value*
`x`	`13.5`

Example 3: (not) accessing another function's variables

Our last example in this section involves two functions, one of which calls the other:

```
def square(x: float) -> float:
    """Return x squared.

    >>> square(3.0)
    9.0
    >>> square(2.5)
    6.25
    """
    return x ** 2


def square_of_sum(numbers: list) -> float:
    """Return the square of the sum of the given numbers."""
    total = sum(numbers)
    return square(total)
```

Let's first call our new function `square_of_sum` in the Python console:

```
>>> nums = [1.5, 2.5]
>>> result = square_of_sum(nums)
>>> result
16.0
```

We can trace what happens at three points when we call `square_of_sum`:

Right before `square_of_sum` is called (from console)

`__main__`

Variable	*Value*
`nums`	`[1.5, 2.5]`

Right before `square` is called (from `square_of_sum`)

`__main__`

Variable	*Value*
`nums`	`[1.5, 2.5]`

`square_of_sum`

Variable	*Value*
`numbers`	`[1.5, 2.5]`
`total`	`4.0`

Right before `square` returns

`__main__`

Variable	*Value*
`nums`	`[1.5, 2.5]`

`square_of_sum`

Variable	*Value*
`numbers`	`[1.5, 2.5]`
`total`	`4.0`

`square`

Variable	*Value*
`x`	`4.0`

From these diagrams, we see how the list `[1.5, 2.5]` is passed from the console to `square_of_sum`, and how the number `4.0` is passed from `square_of_sum` to `square`.

Now suppose we wanted to do something a bit silly: have `square` access `total` instead of `x`. We know from our memory model that these variables should be assigned the same value, so the program's behaviour shouldn't change, right?

```
def square(x: float) -> float:
    """Return x squared.

    >>> square(3.0)
    9.0
    >>> square(2.5)
    6.25
    """
    return total ** 2  # Now we're using total instead of x


def square_of_sum(numbers: list) -> float:
    """Return the square of the sum of the given numbers."""
    total = sum(numbers)
    return square(total)
```

Let's see what happens when we try to call `square_of_sum` in the Python console now:

```
>>> nums = [1.5, 2.5]
>>> square_of_sum(nums)
Traceback (most recent call last):
  ...
  File "<input>", line 1, in <module>
  File "<input>", line 15, in square_of_sum
  File "<input>", line 9, in square
NameError: name 'total' is not defined
```

An error occurs! Let's take a look at the state of memory when `square` is called (this is the same as above):

`__main__`

Variable	*Value*
`nums`	`[1.5, 2.5]`

`square_of_sum`

Variable	*Value*
`numbers`	`[1.5, 2.5]`
`total`	`4.0`

`square`

Variable	*Value*
`x`	`4.0`

Well, there is indeed both a `total` variable and an `x` variable with the same value, `4.0`. So why are we getting this error? The answer is Python's rule for local scope: *a local variable can only be accessed in the function body it is defined*. Here, the statement `return total ** 2` is in the body of `square`, but attempts to access the local variable of a different function (`square_of_sum`). When the Python interpreter attempts to retrieve the value of `total`, it looks only in the scope of `square`, and doesn't find `total`, resulting in a `NameError`.

The somewhat non-intuitive point about this behaviour is that this happens *even when `square_of_sum` is still active*. In our example, `square` is called from within `square_of_sum`, and so the variable `total` *does* exist in Python's memory—it just isn't accessible. While this might seem like a limitation of the language, it's actually a good thing: this prevents you from accidentally using a variable from a completely different function when working on a function.

Summary

In this section, we learned about how Python handles *local variables*, by making them accessible only from within the function that they are defined. Though we hope this makes intuitive sense, some of the details and diagrams we presented here were fairly technical. We recommend coming back to this section in a few days and reviewing this material, perhaps by explaining in your own words what's happening in each example. You can also practice drawing this style of memory model diagram for future code that you write.