6.1 Variable Reassignment, Revisited

So far, we have largely treated values and variables in Python as being constant over time. Once a variable is initialized, its value does not change as the program runs. This principle has made it easier to reason about our code: once we assign a variable to a value, we can easily look up the variable's value at any later point in the program.[1]

[1] Indeed, this is a fact that we take for granted in mathematics: if we say "let $x = 10$" in a calculation or proof, we expect x to keep that same value from start to finish!

However, in programs it is sometimes useful to have variables change value over time. We saw one example of this last week when we studied for loops, in which both the loop variable and accumulator are reassigned to different values as the loop iterates. In this chapter, we'll go deeper and investigate two related but distinct ways that a variable can change over time in a program: *variable reassignment* and *object mutation*. Let's begin by revisiting variable reassignment.

Variable reassignment

Recall that a Python statement of the form `___ = ___` is called an *assignment statement*, which takes a variable name on the left-hand side and an expression on the right-hand side, and assigns the value of the expression to the variable.

A **variable reassignment** is a Python action that assigns a value to a variable when that variable already refers to a value. The most common kind of variable reassignment is with an assignment statement:

```
>>> x = 1
>>> x      # Here, x refers to value 1
1
>>> x = 5  # The variable x is reassigned with this statem
>>> x      # Now, x refers to the value 5
5
```

The for loops that we studied in Chapter 5 all used variable reassignment to update the *accumulator variable* inside the loop. For example, here is our `my_sum` function from 5.4 Repeated Execution: For Loops.

```
def my_sum(numbers: list[int]) -> int:
    """Return the sum of the given numbers."""
    sum_so_far = 0
    for number in numbers:
        sum_so_far = sum_so_far + number
    return sum_so_far
```

At each loop iteration, the statement `sum_so_far = sum_so_far + number` did two things:

1. Evaluate the right-hand side (`sum_so_far + number`) using the *current* value of `sum_so_far`, obtaining a new value.
2. Reassign `sum_so_far` to refer to that new value.

This is the Python mechanism that causes `sum_so_far` to refer to the total sum at the end of the loop, which of course was the whole point of the loop! Indeed, updating loop accumulators is one of the most natural uses of variable reassignment.

This for loop actually illustrates another common form of variable reassignment: reassigning the *loop variable* to a different value at each loop iteration of a for loop. For example, when we call `my_sum([10, 20, 30])`, the loop variable `number` gets assigned to the value `10`, then reassigned to the value `20`, and then reassigned to the value `30`.

Variable reassignment never affects other variables

Consider the following Python code:

```
x = 1
y = x + 2
x = 7
```

Here, the variable `x` is reassigned to `7` on line 3. But what happens to `y`? Does it now also get "reassigned" to `9` (which is `7 + 2`), or does it stay at its original value `3`?

We can state Python's behaviour here with one simple rule: **variable reassignment only changes the immediate variable being reassigned, and does not change any other variables or values, even ones that were defined using the variable being reassigned**. And so in the above example, `y` still refers to the value `3`, even after `x` is reassigned to `7`.

This rule might seem a bit strange at first, but is actually the simplest way that Python could execute variable reassignment: it allows programmers to reason about these assignment statements in a top-down order, without worrying that future assignment statements could affect previous ones. If we're tracing through our code carefully and read `y = x + 2`, I can safely predict the value of `y` based on the *current* value of `x`, without worrying about how `x` might be reassigned later in the program.[2]

[2] While this reasoning is correct for variable reassignment, the story will get more complicated when we discuss the other form of "value change" in the next section.

Augmented assignment statements

Our `my_sum` code from earlier this section uses a very common case of variable reassignment: updating a numeric variable by adding a given value.

```
sum_so_far = sum_so_far + numbers
```

Because this type of reassignment is so common, Python (and many other programming languages) defines a special syntax to represent this operation called the **augmented assignment statement** (often shortened to **augmented assignment**). There are a few different forms of augmented assignment, supporting different "update" operations. Here is the syntax for an addition-based augmented assignment:

```
<variable> += <expression>
```

When the Python interpreter executes the statement *for numeric data*, it does the following:

1. Check that `<variable>` already exists (i.e., is assigned a value). If the variable doesn't exist, a `NameError` is raised.[3]
2. Evaluate `<expression>`.
3. Add the value of `<variable>` and the value of `<expression>` from Step 2.
4. Reassign `<variable>` to the new sum value produced in Step 3.

[3] Try it in the Python console!

So for example, we could replace the line of code `sum_so_far = sum_so_far + number` with the augmented assignment statement to achieve the same effect:

```
def my_sum_v2(numbers: list[int]) -> int:
    """Return the sum of the given numbers."""
    sum_so_far = 0
    for number in numbers:
        sum_so_far += number  # NEW: augmented assignment
    return sum_so_far
```

Pretty cool! The Python programming language defines other forms of augmented assignment for different arithmetic operations: `-=`, `*=`, `//=`, `%=`, `/=`, and `**=`, although of these `-=` and `*=` are most common. We encourage you to try each of these out in the Python console, and keep them in mind when you are performing variable reassignment with numeric values.

Looking ahead: augmented assignment for other data types

You might have wondered about two subtleties in our introduction of augmented assignment. Why do we call this type of statement "augmented assignment" and not "augmented *re*assignment"? And why did we restrict our attention to only numeric data, when we know Python operators like `+` support other kinds of data types, like strings and lists?

It turns out that these augmented assignment operators like `+=` behave differently for different data types, and in some cases do not actually reassign a variable. This might sound confusing at first, but leads into the next section, where we'll discuss the other way of "changing a variable's value": *object mutation*. Let's keep going!