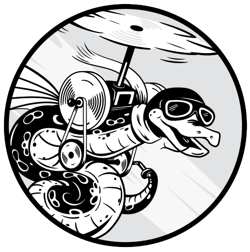
5

VARIABLES AND TYPES



Some of the most pernicious misconceptions about Python revolve around its nuances regarding variables and data types. Misunderstandings related to this *one* topic cause countless frustrating bugs, and this is unfortunate. Python's way of handling variables is at the core of its power and versatility. If you understand this, everything else falls into place.

My own understanding of this topic was cemented by "Facts and Myths About Python Names and Values," Ned Batchelder's now-legendary talk at PyCon 2015. I recommend you watch the video of the presentation at *https://youtu.be/_AEJHKGk9ns*, either now or after reading this chapter.

Variables According to Python: Names and Values

Many myths about Python variables stem from people's attempts to describe the language in terms of *other languages*. Perhaps most annoying to Python experts is the misleading aphorism, "Python has no variables," which is really just the product of someone being overly clever about the fact that the Python language uses the terms *name* and *value*, instead of *variable*.

Python developers still use the term *variable* on a regular basis, and it even appears in the documentation, as it is part of understanding the overall system. However, for the sake of clarity, I'll use the official Python terms exclusively, throughout the rest of the book.

Python uses the term *name* to refer to what would conventionally be called a variable. A name refers to a value or an object, in the same way that your name refers to you but does not contain you. There may even be multiple names for the same thing, just as you may have a given name and a nickname. A *value* is a particular instance of data in memory. The term *variable* refers to the combination of the two: a name that refers to a value. From now on, I'll only use the term *variable* in relation to this precise definition.

Assignment

Let's look at what happens under the hood when I define a variable per the above definitions like this:

```
answer = 42
```

Listing 5-1: `simple_assignment.py:1`

The name `answer` is *bound* to the value `42`, meaning the name can now be used to refer to the value in memory. This operation of binding is referred to as an *assignment*.

Look at what happens behind the scenes when I assign the variable `answer` to a new variable, `insight`:

```
insight = answer
```

Listing 5-2: `simple_assignment.py:2`

The name `insight` doesn't refer to a copy of the value `42`, but rather to the same, original value. This is illustrated in Figure 5-1.

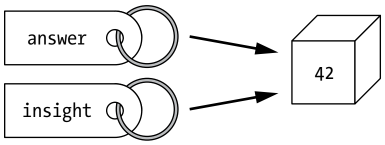


Figure 5-1: Multiple names can be bound to the same value in memory.

In memory, the name insight is bound to the value 42, which was already bound to another name: answer. Both names are still usable as variables. More importantly, insight is not bound to answer, but rather to the same value that answer was already bound to when I assigned insight. A name always points to a value.

Back in Chapter 3, I introduced the is operator, which compares *identity* —the specific location in memory that a name is bound to. This means is doesn't check whether a name points to equivalent values, but rather whether it points to the *same* value in memory.

When you make an assignment, Python makes its own decisions behind the scenes about whether to create a new value in memory or bind to an existing value. The programmer often has very little control over this decision.

Consider this example:

```
spam = 123456789
maps = spam
eggs = 123456789
```

Listing 5-3: value_and_identity.py:1

I assign identical values to spam and eggs. I also bind maps to the same value as spam. (In case you didn't catch it, "maps" is "spam" backward. No wonder GPS gets annoying.)

When I compare the names with the comparison operator (==) to check whether the values are equivalent, both expressions return True, as one would expect:

```
print(spam == maps)  # prints True
print(spam == eggs)  # prints True
```

Listing 5-4: value_and_identity.py:2

However, when I compare the identities of the names with is, something surprising happens:

```
print(spam is maps)  # prints True
print(spam is eggs)  # prints False
```

Listing 5-5: value_and_identity.py:3

The names spam and maps are both bound to the same value in memory, but eggs is bound to a different but equivalent value. Thus, spam and eggs don't share an identity. This is illustrated in Figure 5-2.

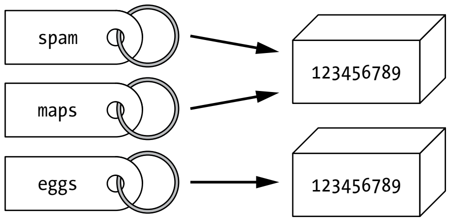


Figure 5-2: `spam` and `maps` share an identity; `eggs` is bound to an equivalent value, but it does not share identity.

It just goes to show, spam by any other name is still spam.

Python isn't guaranteed to behave exactly like this, and it may well decide to reuse an existing value. For example:

```
answer = 42
insight = 42
print(answer is insight)  # prints True
```

Listing 5-6: `assign_reuse.py`

When I assign the value `42` to `insight`, Python decides to bind that name to the existing value. Now, `answer` and `insight` happen to be bound to the same value in memory, and thus, they share an identity.

This is why the identity operator (`is`) can be sneaky. There are many situations in which `is` appears to work like the comparison operator (`==`).

GOTCHA ALERT The `is` operator checks identity. Unless you *really* know what you're doing, only use this to check if something `is None`.

As a final note, the built-in function `id()` returns an integer representing the identity of whatever is passed to it. These integers are the values that the `is` operator compares. If you're curious about how Python handles names and values, try playing with `id()`.

PEDANTIC NOTE In CPython, the value returned from the `id()` function is derived from the memory address for the value.

Data Types

As you've likely noticed, Python does not require you, the programmer, to declare a type for your variables. Back when I first picked up Python, I joined the `#python` channel on IRC and jumped right in.

"How do you declare the data type of a variable in Python?" I asked, in all the naivete of a first-year coder.

Within moments, I received a response that I consider to be my first true induction into the bizarre world of programming: "You're a data type."

The room regulars went on to explain that Python is a dynamically typed language, meaning I didn't have to tell the language what sort of information to put in a variable. Instead, Python would decide the type for me. I didn't even have to use a special "variable declaration" keyword. I just had to assign like this:

```
answer = 42
```

Listing 5-7: types.py:1

At that precise moment, Python became my all-time favorite language.

It's important to remember that Python is still a strongly typed language. I touched on this concept, along with dynamic typing, in Chapter 3. Ned Batchelder sums up Python's type system quite brilliantly in his aforementioned PyCon 2015 talk about names and values:

> "Names have a scope—they come and go with functions—but they have no type. Values have a type . . . but they have no scope."

Although I haven't touched on scope yet, this should already make sense. Names are bound to values, and those values exist in memory, as long as there is some *reference* to them. You can bind a name to literally any value you want, but you are limited as to what you can do with any particular value.

The type() Function

If you ever need to know a value's data type, you can use the built-in `type()` function. Recall that everything in Python is an object, so this function will really just return what class the value is an instance of:

```
type(answer)  # prints <class 'int'>
```

Listing 5-8: types.py:2

Here, you can see that the value assigned to `answer` is an integer (`int`). On rare occasions, you may want to check the data type before you do something with a value. For that, you can pair the `type()` function with the `is` operator, like this:

```
if type(answer) is int:
    print("What's the question?")
```

Listing 5-9: types.py:3a

In many cases where this sort of introspection is necessary, it may be better to use `isinstance()` instead of `type()`, as it accounts for subclasses and inheritance (see Chapter 13). The function itself returns `True` or `False`, so I can use it as the condition in an `if` statement:

```
if isinstance(answer, int):
    print("What's the question?")
```

Listing 5-10: types.py:3b

Truth be told, there is rarely a need for either. Instead, Python developers prefer a more dynamic approach.

Duck Typing

Python uses what is known (unofficially) as *duck typing*. This isn't a technical term at all; it comes from the old saying:

> If it looks like a duck, walks like a duck, and quacks like a duck,
> then it probably *is* a duck.

Python doesn't care much about what a value's data type is, but rather it cares about the *functionality* of the value's data type. For example, if an object supports all the math operators and functions, and if it accepts floats and integers as operands on the binary operators, then Python considers the object to be a numeric type.

In other words, Python doesn't care if it's actually a robotic duck or a moose in a duck costume. If it has the traits needed, the rest of the details are usually moot.

If you're familiar with object-oriented programming, particularly how quickly inheritance can get out of hand, then this whole concept of duck typing will probably be a breath of fresh air. If your class behaves as it should, it usually won't matter what it inherits from.

Scope and Garbage Collection

Scope is what defines where a variable can be accessed from. It might be available to an entire module or limited to the suite (body) of a function.

As I mentioned already, names have *scope*, whereas values do not. A name can be *global*, meaning it is defined by itself in a module, or it can be *local*, meaning it only exists within a particular function or comprehension.

Local Scope and the Reference-Counting Garbage Collector

Functions (including lambdas) and comprehensions define their own scope; they are the only structures in the language to do so. Modules and classes don't have their own scope in the strictest sense; they only have their own namespace. When a scope reaches its end, all the names defined within it are automatically deleted.

For any particular value, Python keeps a *reference count*, which is simply a count of how many references exist for that value. Every time a value is bound to a name, a reference is created (although there are other ways the language may create references). When there are no more references, the value is deleted. This is the *reference-counting garbage collector*, and it efficiently handles most garbage collection scenarios.

PEDANTIC NOTE Technically, Python's garbage collection behaviors are an implementation detail specific to CPython, the main "flavor" of Python. Other flavors of the language may (or may not) handle this differently, but it probably won't ever matter to you, unless you're doing something insanely advanced and weird.

You can see how this works with a typical function, like this:

```
def spam():
    message = "Spam"
    word = "spam"
    for _ in range(100):
        separator = ", "
        message += separator + word
    message += separator
    message += "spam!"

    return message
```

Listing 5-11: *`local_scope.py:1`*

I create a `spam()` function, inside of which I define the names `message`, `word`, and `separator`. I can access any of these names inside the function; that is their local scope. It doesn't matter that `separator` is defined within a `for` loop, as loops don't have their own scope. I can still access it outside of the loop.

However, I cannot access any of these names outside of the function:

```
print(message)  # NameError: name 'message' is not defined
```

Listing 5-12: *`local_scope.py:2`*

Trying to access `message` outside the context of the `spam()` function where it was defined will raise a `NameError`. In this example, `message` doesn't exist in the outer scope. What's more, as soon as the function `spam()` exits, the names `message`, `word`, and `separator` are deleted. Because `word` and `separator` each referred to values with a reference count of one (meaning only one name was bound to each), the values are also deleted.

The value of `message` is not deleted when the function exits, however, because of the return statement at the end of the function (see Listing 5-11) and what I do with that value here:

```
output = spam()
print(output)
```

Listing 5-13: *`local_scope.py:3`*

I bind the value returned by `spam()` to `output` in the outer scope, meaning that value still exists in memory and can be accessed outside of the function. Assigning the value to `output` increases the reference count for that value, so even though the name `message` is deleted when `spam()` exits, the value is not.

Interpreter Shutdown

When the Python interpreter is asked to shut down, such as when a Python program terminates, it enters *interpreter shutdown*. During this phase, the interpreter goes through the process of releasing all allocated resources, calling the garbage collector multiple times, and triggering destructors in objects.

You can use the atexit module from the standard library to add functions to this interpreter shutdown process. This may be necessary in some highly technical projects, although in general, you shouldn't need to do this. Functions added via atexit.register() will be called in a last-in-first-out manner. However, be aware that it becomes difficult to work with modules, including the standard library, during interpreter shutdown. It's like trying to work in a building as it's being demolished: the janitor's closet may disappear at any time, without warning.

Global Scope

When a name is defined within a module but outside of any function, class, or comprehension, it is considered to be in *global scope*. Although it's okay to have some global scope names, having too many usually leads to the creation of code that is difficult to debug and maintain. Therefore, you should use global scope names sparingly for variables. There is often a cleaner solution, such as a class (see Chapter 7).

Properly using global scope names in the context of a more local scope, such as a function, requires you to think ahead a little. Consider what I do if I want a function that can modify a global variable storing a high score. First, I define the global variable:

```
high_score = 10
```

Listing 5-14: global_scope.py:1

I'll write this function the wrong way first:

```
def score():
    new_score = 465  # SCORING LOGIC HERE
    if new_score >❶ high_score:  # ERROR: UnboundLocalError
        print("New high score")
      ❷ high_score = new_score


score()
print(high_score)
```

Listing 5-15: global_scope.py:2

When I run this code, Python complains that I'm using a local variable before I've assigned a value to it ❶. The problem is, I'm assigning to the name high_score within the scope of the function score() ❷, and that *shadows*, or hides, the global high_score name behind the new, local high_score name. The fact that I've created a local high_score name *anywhere* *in the function* makes it impossible for the function to ever "see" the global `high_score` name.

To make this work, I need to declare that I'm going to use the global name in the local scope, instead of defining a new local name. I can do this with the `global` keyword:

```
def score():
    global high_score
    new_score = 465  # SCORING LOGIC HERE
    if new_score > high_score:
        print("New high score")
        high_score = new_score


score()
print(high_score)  # prints 465
```

Listing 5-16: `global_scope.py:3`

Before I do anything else in my function, I must specify that I'm using the global `high_score` name. This means that anywhere I assign a value to the name `high_score` in `score()`, the function will use the global name, instead of trying to create a new local name. The code now works as expected.

Every time you wish to rebind a global name from within a local scope, you must use the `global` keyword first. If you're only accessing the current value bound to a global name, you don't need to use the `global` keyword. It is vital for you to cultivate this habit, because Python won't always raise an error if you handle scope incorrectly. Consider this example:

```
current_score = 0

def score():
    new_score = 465  # SCORING LOGIC HERE
    current_score = new_score

score()
print(current_score)  # prints 0
```

Listing 5-17: `global_scope_gotcha.py:1a`

This code runs without raising any errors, but the output is wrong. A new name, `current_score`, is being created in the local scope of the function `score()`, and it is bound to the value `465`. This shadows the global name `current_score`. When the function terminates, both the `new_score` and the local `current_score` are deleted. In all of this, the global `current_score` has remained untouched. It is still bound to `0`, and that is what is printed out.

Once again, to resolve this problem, I need only use the `global` keyword:

```
current_score = 0

def score():
    global current_score
    new_score = 465  # SCORING LOGIC HERE
    current_score = new_score

score()
print(current_score)  # prints 465
```

Listing 5-18: `global_scope_gotcha.py:1b`

Because I specified that the global `current_name` is to be used in this function, the code now behaves precisely as expected, printing out the value `465`.

The Dangers of Global Scope

There is one more major gotcha to account for with global scope. Modifying any variable at a global level, as in rebinding or mutating on a name outside the context of a function, can lead to confusing behavior and surprising bugs—especially once you start dealing with multiple modules. It's acceptable for you to initially "declare" a name at a global scope, but you should do all further rebinding and mutation of that global name at the local scope level.

By the way, this does *not* apply to classes, which do not actually define their own scope. I'll return to this later in this chapter.

The nonlocal Keyword

Python allows you to write functions within functions. I'll defer discussing the practicality of this until Chapter 6. Here, I mainly want to explore this functionality's impact on scope. Consider the following example:

```
spam = True


def order():
    eggs = 12

    def cook():
      ❶ nonlocal eggs

        if spam:
            print("Spam!")

        if eggs:
            eggs -= 1
            print("...and eggs.")

    cook()


order()
```

Listing 5-19: `nonlocal.py`

The function `order()` contains another function: `cook()`. Each function has its own scope.

Remember, as long as a function only accesses a global name like `spam`, you don't need to do anything special. However, trying to *assign* to a global name will actually define a new local name that shadows the global one. The same behavior is true of the inner function using names defined in the outer function, which is known as the *nested scope* or *enclosing scope*. To get around this, I specify that `eggs` is `nonlocal`, meaning it can be found in the enclosing scope, rather than in the local scope ❶. The inner function `cook()` has no trouble accessing the global name `spam`.

The `nonlocal` keyword starts looking for the indicated name in the innermost nested scope, and if it doesn't find it, it moves to the next enclosing scope above that. It repeats this until it either finds the name or determines that the name does not exist in a nonglobal enclosing scope.

Scope Resolution

Python's rule about which scopes it searches for a name, and in what order, is called the *scope resolution order*. The easiest way to remember the scope resolution order is with the acronym *LEGB*—for which my colleague Ryan gave me the handy mnemonic "Lincoln Eats Grant's Breakfast":

Local

Enclosing-function locals (that is, anything found via `nonlocal`)

Global

Built-in

Python will look in these scopes, in order, until it finds a match or reaches the end. The `nonlocal` and `global` keywords adjust the behavior of this scope resolution order.

The Curious Case of the Class

Classes have their own way of dealing with scope. Technically speaking, classes don't directly factor into the scope resolution order. Every name declared directly within a class is known as an *attribute*, and it is accessed through the dot (`.`) operator on the class (or object) name.

To demonstrate this, I'll define a class with a single attribute:

```
class Nutrimatic:
  ❶ output = "Something almost, but not quite, entirely unlike tea."

    def request(self, beverage):
        return❷ self.output


machine = Nutrimatic()
mug = machine.request("Tea")
print(mug)   # prints "Something almost, but not quite, entirely unlike tea."

print(❸ machine.output)
print(❹ Nutrimatic.output)
```

Listing 5-20: `class_attributes.py`

Those three print statements all output the same thing. Running that code gives me this:

```
Something almost, but not quite, entirely unlike tea.
Something almost, but not quite, entirely unlike tea.
Something almost, but not quite, entirely unlike tea.
```

The name `output` is a *class attribute* ❶, belonging to the `Nutrimatic` class. Even within that class, I would not be able to refer to it merely as `output`. I must access it through `self.output` ❷, as `self` refers to the class instance the function (instance method) `request()` is being called on. I can also access it via `machine.output` ❸ or `Nutrimatic.output` ❹ anywhere the object `machine` or the class `Nutrimatic` is, respectively, in scope. All of those names point to the exact same attribute: `output`. Especially in this case, there's no real difference between them.

Generational Garbage Collector

Behind the scenes, Python also has a more robust *generational garbage collector* that handles all of the odd situations a reference-counting garbage collector cannot, such as reference cycles (when two values reference one another). All of these situations, and the ways they're handled by the garbage collector, are far beyond the scope of this book.

Moving forward, the most important takeaway to remember is that the generational garbage collector incurs some performance costs. Thus, it's sometimes worthwhile to avoid reference cycles. One way to do this is with `weakref`, which creates a reference to a value without increasing that value's reference count. This feature was defined in PEP 205, and the documentation exists at *https://docs.python.org/library/weakref.*

The Immutable Truth

Values in Python can be either *immutable* or *mutable*. The difference hinges on whether the values can be *modified in place*, meaning they can be changed right where they are in memory.

Immutable types cannot be modified in place. For example, integers (`int`), floating-point numbers (`float`), strings (`str`), and tuples (`tuple`) are all immutable. If you attempt to mutate an immutable value, you'll wind up with a completely different value being created:

```
eggs = 12
carton = eggs
print(eggs is carton)  # prints True
eggs += 1
print(eggs is carton)  # prints False
print(eggs)  # prints 13
print(carton)  # prints 12
```

Listing 5-21: `immutable_types.py`

Initially, `eggs` and `carton` are both bound to the same value, and thus, they share an identity. When I modify `eggs`, it is rebound to a new value, so it no longer shares an identity with `carton`. You can see that the two names now point to different values.

Mutable types, on the other hand, can be modified in place. Lists constitute one example of a mutable type:

```
temps = [87, 76, 79]
highs = temps
print(temps is highs)  # prints True
❶ temps += [81]
print(temps is highs)  # prints True
print(highs)  # prints [87, 76, 79, 81]
print(temps)  # prints [87, 76, 79, 81]
```

Listing 5-22: `mutable_types.py`

Because the list is aliased to both `temps` and `highs`, any modifications made to the list value ❶ are visible through either name. Both names are bound to the original value, as demonstrated by the `is` comparisons. This remains the case, even after that value is mutated.

Passing by Assignment

Another frequent question from programmers new to the language is, "Does Python pass by value or by reference?"

The answer is, "Effectively, neither." More accurately, as Ned Batchelder describes it, Python *passes by assignment*.

Neither the values nor the names bound to them are moved. Instead, each value is bound to the parameter via assignment. Consider a simple function:

```
def greet(person):
    print(f"Hello, {person}.")


my_name = "Jason"
greet(my_name)
```

Here, there is one copy of the string value `"Jason"` in memory, and that is bound to the name `my_name`. When I pass `my_name` to the `greet()` function—specifically, to the `person` parameter—it's the same as if I had said (`person = my_name`).

Again, assignment never makes a copy of a value. The name `person` is now bound to the value `"Jason"`.

This concept of passing by assignment gets tricky when you start working with mutable values, such as lists. To demonstrate this often-unexpected behavior, I've written a function that finds the lowest temperature in a list passed to it:

```
def find_lowest(temperatures):
    temperatures.sort()
    print(temperatures[0])
```

Listing 5-23: `lowest_temp.py:1a`

At first glance, you may assume that passing a list to the `temperatures` parameter will make a copy, so it shouldn't matter if you modify the value bound to the parameter. However, lists are mutable, meaning *the value itself* can be modified:

```
temps = [85, 76, 79, 72, 81]
find_lowest(temps)
print(temps)
```

Listing 5-24: `lowest_temp.py:2`

When I passed `temps` to the function's `temperatures` parameter, I only *aliased* the list, so any changes made on `temperatures` are visible from all the other names bound to that same list value—namely, from `temps`.

You can see this in action when I run this code and get the following output:

```
72
[72, 76, 79, 81, 85]
```

When `find_lowest()` sorted the list passed to `temperatures`, it actually sorted the one mutable list that both `temps` and `temperatures` aliased. This is a clear case of a function having *side effects*, which are changes to values that existed before the function call.

An awe-inspiring number of bugs originate from this one type of misunderstanding. In general, functions should not have side effects, meaning that any values passed to the function as arguments should *not* be directly mutated. To avoid mutating the original value, I have to explicitly make a copy of it. Here's how I'd do that in the `find_lowest()` function:

```
def find_lowest(temperatures):
    sorted_temps =❶ sorted(temperatures)  # sorted returns a new list
    print(sorted_temps[0])
```

Listing 5-25: `lowest_temp.py:1b`

The `sorted()` function has no side effects; it creates a new list using the items in the list passed to it ❶. It then sorts this new list and returns it. I bind this new list to `sorted_temps`. Thus, the original list (bound to `temps` and `temperatures`) is untouched.

If you're coming from C and C++, it may be helpful to remember the potential hang-ups related to pass-by-pointer or pass-by-reference. Although

Python's assignment is scarcely similar from a technical standpoint, the risks of side effects and unintended mutations are the same.

Collections and References

All collections, including lists, employ a clever little semantic detail that can become a royal pain if you don't know to expect it: ***Individual items are references.*** Just as a name is bound to a value, so also are items in collections bound to values, in the same manner. This binding is called a *reference*.

A simple example involves trying to create a tic-tac-toe board. This first version won't work quite how you'd expect.

I'll start by creating the game board:

```
board = [["-"]❶ * 3] * 3  # Create a board
```

Listing 5-26: `tic_tac_toe.py:1a`

I'm trying to create a two-dimensional board. You can fill a collection, like a list, with several items, all with the same repeating value, using the multiplication operator ❶, as I've done here. I enclose the repeating value in square brackets and multiply it by the number of repetitions I want. A single row of my board is defined with `["-"] * 3`, which makes a list of three `"-"` strings.

Unfortunately, this won't work the way you'd expect. The problem begins when I attempt to define the second dimension of the array—three copies of the `[["-"] * 3]` list—using multiplication. You can see the problem manifest when I try to make a move:

```
❷ board[1][0] = "X"  # Make a move

# Print board to screen
for row in board:
    print(f"{row[0]} {row[1]} {row[2]}")
```

Listing 5-27: `tic_tac_toe.py:2`

When I mark a move on the board ❷, I want to see that change in only one spot on the board, like this:

```
- - -
X - -
- - -
```

Instead, I get this nasty surprise:

```
X - -
X - -
X - -
```

Cue the weeping and gnashing of teeth. Somehow, that one change has propagated to *all three rows*. Why?

Initially, I created a list with three "-" values as items ❶. Since strings are immutable and thus cannot be modified in place, this works as expected. Rebinding the first item in the list to "X" does not affect the other two items.

The outer dimension of the list is composed of three list items. Because I defined *one* list and used it *three* times, I now have three *aliases* for one mutable value! By changing that list through one reference (the second row), I'm mutating that one shared value ❷, so all three references see the change.

There are a few ways to fix this, but all of them work by ensuring each row references a separate value, like so:

```
board = [["-"] * 3 for _ in range(3)]
```

Listing 5-28: tic_tac_toe.py:1b

I only needed to change how I defined the game board initially. I now use a *list comprehension* to create the rows. In short, this list comprehension will define a separate list value from ["-"] * 3 three different times. (List comprehensions get complicated; they'll be explained in depth in Chapter 10.) Running the code now results in the expected behavior:

```
- - -
X - -
- - -
```

Long story short, whenever you're working with a collection, remember that an item is no different from any other name. Here is one more example to drive this point home:

```
scores_team_1 = [100, 95, 120]
scores_team_2 = [45, 30, 10]
scores_team_3 = [200, 35, 190]

scores = (scores_team_1, scores_team_2, scores_team_3)
```

Listing 5-29: team_scores.py:1

I create three lists, assigning each to a name. Then, I pack all three into the tuple scores. You may remember from earlier that tuples cannot be modified directly, because they're immutable. That same rule does not necessarily apply to a tuple's items. You can't change the tuple itself, but you can (indirectly) modify the values its items refer to:

```
scores_team_1[0] = 300
print(scores[0])  # prints [300, 95, 120]
```

Listing 5-30: team_scores.py:2

When I mutate the list scores_team_1, that change appears in the first item of the tuple, because that item only aliased a mutable value.

I could also directly mutate a mutable list in the `scores` tuple through two-dimensional subscription, like this:

```
scores[0][0] = 400
print(scores[0])  # prints [400, 95, 120]
```

Listing 5-31: `team_scores.py:3`

Tuples don't give you any sort of security about things being modified. Immutability exists mainly for efficiency, not for any sort of security. Mutable values are *always* going to be mutable, no matter where they live or how they're referred to.

The problems in the two examples above may seem relatively easy to spot, but things start getting troublesome when the related code is spread out across a large file or multiple files. Mutating on a name in one module may unexpectedly modify an item of a collection in a completely different module, and you might never have expected it.

Shallow Copy

There are many ways to ensure you are binding a name to a *copy* of a mutable value, instead of aliasing the original; the most explicit of these ways is with the `copy()` function. This is sometimes also known as a *shallow copy*, in contrast to the *deep copy* I'll cover later.

To demonstrate this, I'll create a `Taco` class (see Chapter 7) that allows you to define the class with various toppings and then add a sauce afterward. This first version has a bug:

```
class Taco:

    def __init__(self, toppings):
        self.ingredients = toppings

    def add_sauce(self, sauce):
        self.ingredients.append(sauce)
```

Listing 5-32: `mutable_ tacos.py:1a`

In the `Taco` class, the initializer `__init__()` accepts a list of toppings, which it stores as the `ingredients` list. The `add_sauce()` method will add the specified `sauce` string to the `ingredients` list.

(Can you anticipate the problem?)

I use the class as follows:

```
default_toppings = ["Lettuce", "Tomato", "Beef"]
hot_taco = Taco(default_toppings)
mild_taco = Taco(default_toppings)
hot_taco.add_sauce("Salsa")
```

Listing 5-33: `mutable_ tacos.py:2a`

I define a list of toppings I want on all my tacos, and then I define two tacos: `hot_taco` and `mild_taco`. I pass the `default_toppings` list to the initializer for each taco. Then I add `"Salsa"` to the list of toppings to `hot_taco`, but I don't want any `"Salsa"` on `mild_taco`.

To make sure this is working, I print out the list of `ingredients` for the two tacos, as well as the `default_toppings` list I started with:

```
print(f"Hot: {hot_taco.ingredients}")
print(f"Mild: {mild_taco.ingredients}")
print(f"Default: {default_toppings}")
```

Listing 5-34: `mutable_ tacos.py:3`

That outputs the following:

```
Hot: ['Lettuce', 'Tomato', 'Beef', 'Salsa']
Mild: ['Lettuce', 'Tomato', 'Beef', 'Salsa']
Default: ['Lettuce', 'Tomato', 'Beef', 'Salsa']
```

Waiter, there's a bug in my taco!

The trouble is, when I created my `hot_taco` and `mild_taco` object by passing `default_toppings` to the `Taco` initializer, I bound both `hot_taco.ingredients` and `mild_taco.ingredients` to the same list value as `default_toppings`. These are now all aliases of the same value in memory. Then, when I call the function `hot_taco.add_sauce()`, I mutate that list value. The addition of `"Salsa"` is visible not only in `hot_taco.ingredients`, but also (unexpectedly) in `mild_taco.ingredients` and in the `default_toppings` list. This is definitely not the desired behavior; adding `"Salsa"` to one taco should only affect that one taco.

One way to resolve this is to ensure I'm assigning a copy of the mutable value. In the case of my `Taco` class, I will rewrite the initializer so it assigns a copy of the specified list to `self.ingredients`, instead of aliasing:

```
import copy


class Taco:

    def __init__(self, toppings):
        self.ingredients =❶ copy.copy(toppings)

    def add_sauce(self, sauce):
        self.ingredients.append(sauce)
```

Listing 5-35: `mutable_ tacos.py:1b`

I make a copy with the `copy.copy()` function ❶, which is imported from `copy`.

I make a copy of the list passed to `toppings` within `Taco.__init__()`, assigning that copy to `self.ingredients`. Any changes made to `self.ingredients` don't affect the others; adding "Salsa" to hot_taco does not change mild_taco .ingredients, nor does it change default_toppings:

```
Hot: ['Lettuce', 'Tomato', 'Beef', 'Salsa']
Mild: ['Lettuce', 'Tomato', 'Beef']
Default: ['Lettuce', 'Tomato', 'Beef']
```

Deep Copy

A shallow copy is all well and good for lists of immutable values, but as previously mentioned, when a mutable value contains other mutable values, changes to those values can appear to replicate in weird ways.

For example, consider what happens when I try to make a copy of a Taco object before changing one of the two tacos. My first attempt results in some undesired behavior. Building on the same Taco class as before (see Listing 5-35), I'll use the copy of one taco to define another:

```
default_toppings = ["Lettuce", "Tomato", "Cheese", "Beef"]
mild_taco = Taco(default_toppings)
hot_taco = ❶copy.copy(mild_taco)
hot_taco.add_sauce("Salsa")
```

Listing 5-36: mutable_ tacos.py:2b

I want to create a new taco (hot_taco) that is initially identical to mild_taco, but with added "Salsa". I'm attempting this by binding a copy of mild_taco ❶ to hot_taco.

Running the revised code (including Listing 5-34) produces the following:

```
Hot: ["Lettuce", "Tomato", "Cheese", "Beef", "Salsa"]
Mild: ["Lettuce", "Tomato", "Cheese", "Beef", "Salsa"]
Default: ["Lettuce", "Tomato", "Cheese", "Beef"]
```

I might not expect any changes made to hot_taco to reflect in mild_taco, but unexpected changes have clearly happened.

The issue is that, when I make a copy of the Taco object value itself, I am not making a copy of the self.ingredients list *within* the object. Both Taco objects contain references to the same list value.

To fix this problem, I can use *deep copy* to ensure that any mutable values inside the object are copied as well. In this case, a deep copy of a Taco object will create a copy of the Taco value, as well as a copy of the any mutable values that Taco contains references to—namely, the list self.ingredients. Listing 5-37 shows that same program, using deep copy:

```
default_toppings = ["Lettuce", "Tomato", "Beef"]
mild_taco = Taco(default_toppings)
hot_taco =❶ copy.deepcopy(mild_taco)
hot_taco.add_sauce("Salsa")
```

Listing 5-37: mutable_ tacos.py:2c

The only change is that I'm using `copy.deepcopy()`, instead of `copy.copy()` ❶. Now when I mutate the list inside `hot_taco`, it doesn't affect `mild_taco`:

```
Hot: ["Lettuce", "Tomato", "Cheese", "Beef", "Salsa"]
Mild: ["Lettuce", "Tomato", "Cheese", "Beef"]
Default: ["Lettuce", "Tomato", "Cheese", "Beef"]
```

I don't know about you, but I'm getting hungry for tacos.

Copying is the most generic way to solve the problem of passing around mutable objects. However, depending on what you're doing, there may be an approach better suited to the particular collection you're using. For example, many collections, like lists, have functions that return a copy of the collection with some specific modification. When you're solving these sorts of issues with mutability, you can start by employing copy and deep copy. Then, you can exchange that for a more domain-specific solution later.

Coercion and Conversion

Names do not have types. Therefore, Python has no need of type casting, at least in the typical sense of the term.

Allowing Python to figure out the conversions by itself, such as when adding together an integer (`int`) and a `float`, is called *coercion*. Here are a few examples:

```
print(42.5)  # coerces to a string
x = 5 + 1.5  # coerces to a float (6.5)
y = 5 + True  # coerces to an int (6)...and is also considered a bad idea
```

Listing 5-38: `coercion.py`

Even so, there are potential situations in which you may need to use one value to create a value of a different type, such as when you are creating a string from an integer. *Conversion* is the process of explicitly casting a value of one type to another type.

Every type in Python is an instance of a class. Therefore, the class of the type you want to create only needs to have an initializer that can handle the data type of the value you're converting from. (This is usually done through duck typing.)

One of the more common scenarios is to convert a string containing a number into a numeric type, such as a `float`:

```
life_universe_everything = "42"

answer = float(life_universe_everything)
```

Listing 5-39: `conversion.py:1`

Here, I start with a piece of information as a string value, which is bound to the name `life_universe_everything`. Imagine I want to do some complex mathematical analysis on this data; to do this, I must first convert the data into a floating-point number. The desired type would be an instance of the class `float`. That particular class has an initializer (`__init__()`) that accepts a string as an argument, which is something I know from the documentation.

I initialize a `float()` object, pass `life_universe_everything` to the initializer, and bind the resulting object to the name `answer`.

I'll print out the `type` and value of `answer`:

```
print(type(answer))
print(answer)
```

Listing 5-40: `conversion.py:2`

That outputs the following:

```
<class 'float'>
42.0
```

Since there were no errors, you can see that the result is a `float` with value `42.0`, bound to `answer`.

Every class defines its own initializers. In the case of `float()`, if the string passed to it cannot be interpreted as a floating-point number, a `ValueError` will be raised. Always consult the documentation for the object you're initializing.

A Note About Systems Hungarian Notation

If you're coming from a statically typed language like C++ or Java, you're probably used to working with data types. Thus, when picking up a dynamically typed language such as Python, it might be tempting to employ some means of "remembering" what type of value every name is bound to. ***Don't do this!*** You will find the most success using Python if you learn to take full advantage of dynamic typing, weak binding, and duck typing.

I will confess: the first year I used Python, I used *Systems Hungarian notation*—the convention of appending a prefix denoting data type to every variable name—to try to "defeat" the language's dynamic typing system. My code was littered with such debris as `intScore`, `floatAverage`, and `boolGameOver`. I picked up the habit from my time using Visual BASIC.NET, and I thought I was brilliant. In fact, I was depriving myself of many opportunities to refactor.

Systems Hungarian notation will quickly render code obtuse. For example:

```
def calculate_age(intBirthYear, intCurrentYear):
    intAge = intCurrentYear - intBirthYear
    return intAge
```

```
def calculate_third_age_year(intCurrentAge, intCurrentYear):
    floatThirdAge = intCurrentAge / 3
    floatCurrentYear = float(intCurrentYear)
    floatThirdAgeYear = floatCurrentYear - floatThirdAge
    intThirdAgeYear = int(floatThirdAgeYear)
    return intThirdAgeYear


strBirthYear = "1985"  # get from user, assume data validation
intBirthYear = int(strBirthYear)

strCurrentYear = "2010"  # get from system
intCurrentYear = int(strCurrentYear)

intCurrentAge = calculate_age(intBirthYear, intCurrentYear)
intThirdAgeYear = calculate_third_age_year(intCurrentAge, intCurrentYear)
print(intThirdAgeYear)
```

Listing 5-41: `evils_of_systems_hungarian.py`

Needless to say, this code is quite painful to read. On the other hand, if you make full use of Python's typing system (and resist the urge to store every intermediate step), the code will be decidedly more compact:

```
def calculate_age(birth_year, current_year):
    return (current_year - birth_year)


def calculate_third_age_year(current_age, current_year):
    return int(current_year - (current_age / 3))


birth_year = "1985"  # get from user, assume data validation
birth_year = int(birth_year)

current_year = "2010"  # get from system
current_year = int(current_year)

current_age = calculate_age(birth_year, current_year)
third_age_year = calculate_third_age_year(current_age, current_year)
print(third_age_year)
```

Listing 5-42: `duck_typing_feels_better.py`

My code became far cleaner once I stopped treating Python like a statically typed language. Python's typing system is a big part of what makes it such a readable and compact language.

Terminology Review

I've introduced a lot of important new words in this section. Since I'll be using this vocabulary frequently throughout the rest of the book, doing a quick recap here is prudent.

alias (v.) To bind a mutable value to more than one name. Mutations performed on a value bound to one name will be visible on all names bound to that mutable value.

assignment (n.) The act of binding a value to a name. Assignment never copies data.

bind (v.) To create a reference between a name and a value.

coercion (n.) The act of implicitly casting a value from one type to another.

conversion (n.) The act of explicitly casting a value from one type to another.

copy (v.) To create a new value in memory from the same data as another value.

data (n.) Information stored in a value. You may have copies of any given data stored in other values.

deep copy (v.) To both copy an object to a new value *and* copy all the data from values referenced within that object to new values.

identity (n.) The specific location in memory that a name is bound to. When two names share an identity, they are bound to the same value in memory.

immutable (adj.) Of or relating to a value that *cannot* be modified in place.

mutable (adj.) Of or relating to a value that *can* be modified in place.

mutate (v.) To change a value in place.

name (n.) A reference to a value in memory, commonly thought of as a "variable" in Python. A name must always be bound to a value. ***Names have scope, but not type.***

rebind (v.) To bind an existing name to a different value.

reference (n.) The association between a name and a value.

scope (n.) A property that defines what section of the code a name is accessible from, such as from within a function or within a module.

shallow copy (v.) To copy an object to a new value but *not* copy the data from values referenced within that object to new values.

type (n.) A property that defines how a raw value is interpreted, for example, as an integer or a boolean.

value (n.) A unique copy of data in memory. There must be a reference to a value, or else the value is deleted. ***Values have type, but not scope.***

variable (n.) A combination of a name and the value the name refers to.

weakref (n.) A reference that does not increase the reference count on the value.

To help keep us grounded in these concepts, we usually use the term *name* instead of *variable.* Instead of *changing* something, we *(re)bind a name* or *mutate a value.* Assignment never copies—it literally always binds a name to a value. Passing to a function is just assignment.

By the way, if you ever have trouble wrapping your head around these concepts and how they play out in your code, try the visualizer at *http://pythontutor.com/*.

Wrapping Up

It's easy to take something like variables for granted, but by understanding Python's unique approach, you can better avail yourself of the power that is available through dynamic typing. I must admit, Python has somewhat spoiled me. When I work in statically typed languages, I find myself pining for the expressiveness of duck typing.

Still, working with Python-style dynamic typing can take getting used to if you have a background in other languages. It's like learning how to speak a new human language: only with time and practice will you begin to think in the new tongue.

If all this is making your head swim, let me reiterate the single most important principles. Names have scope, but no type. Values have type, but no scope. A name can be bound to any value, and a value can be bound to any number of names. It really is that dead simple! If you remember that much, you'll go a long way.