Variables and objects in Python

Trey Hunner
9 min. read • Python 3.7—3.11 • Feb. 28, 2022

Share

Tags Assignment and Mutation

In Python, variables and data structures **don't contain objects**. This fact is both commonly overlooked and tricky to internalize.

You can happily use Python for years without really understanding the below concepts, but this knowledge can certainly help alleviate *many* common Python gotchas.

Terminology

Let's start with by introducing some terminology. The last few definitions likely won't make sense until we define them in more detail later on.

Object (a.k.a. **value**): a "thing". Lists, dictionaries, strings, numbers, tuples, functions, and modules are all objects. "Object" defies definition because everything is an object in Python.

Variable (a.k.a. **name**): a name used to refer to an object.

Pointer (a.k.a. **reference**): describes where an object lives (often shown visually as an arrow)

Equality: whether two objects represent the same data

Identity: whether two pointers refer to the same object

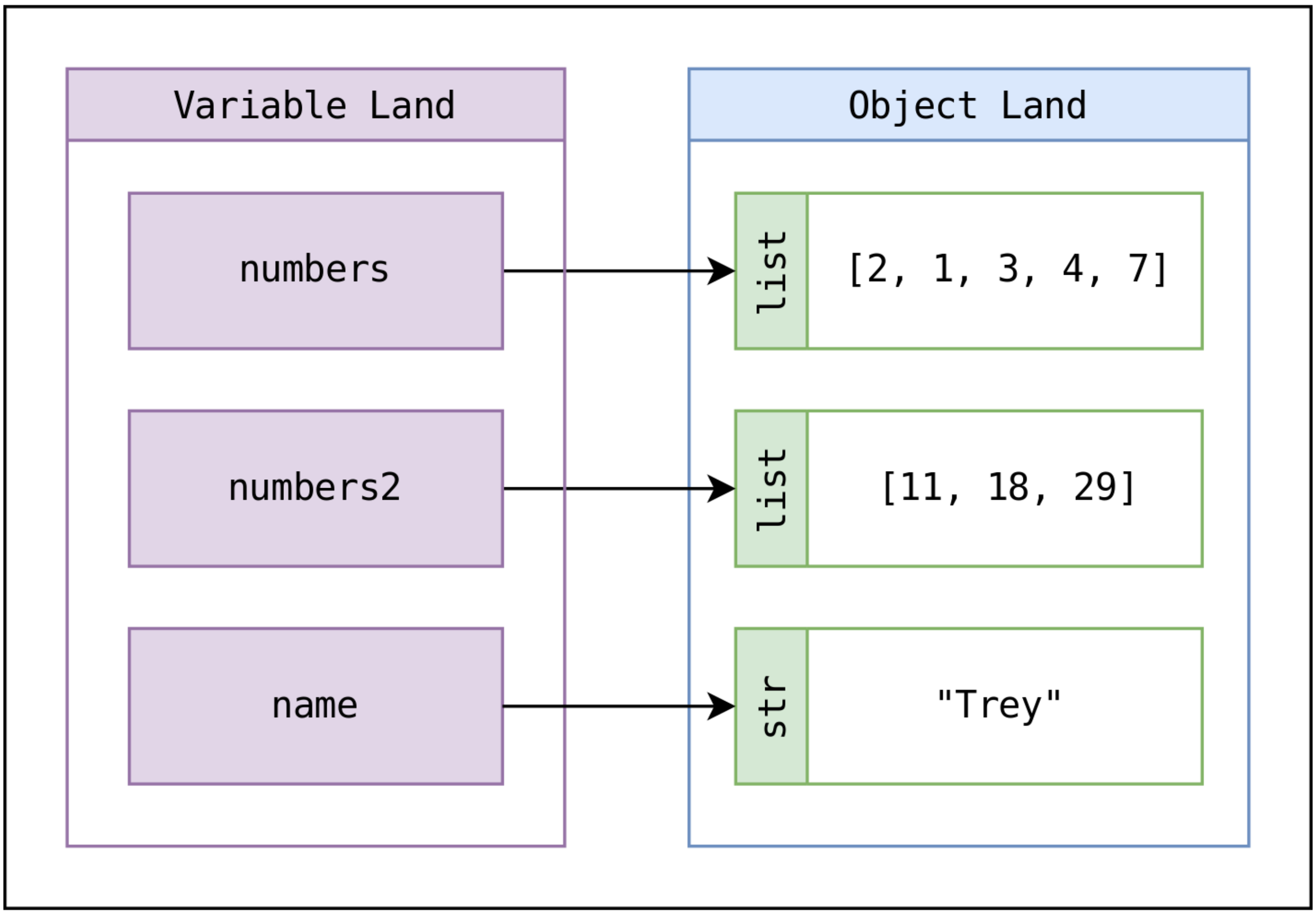
These terms are best understood by their relationships to each other and that's the primarily purpose of this article.

Python's variables are pointers, not buckets

Variables in Python are not buckets containing things; they're **pointers** (they *point* to objects).

The word "pointer" may sound scary, but a lot of that scariness comes from related concepts (e.g. dereferencing) which aren't relevant in Python. In Python a pointer just represents **the connection between a variable and an objects**.

Imagine **variables** living in *variable land* and **objects** living in *object land*. A **pointer** is a little arrow that connects each variable to the object it **points to**.



This above diagram represents the state of our Python process after running this code:

```
>>> numbers = [2, 1, 3, 4, 7]
>>> numbers2 = [11, 18, 29]
>>> name = "Trey"
```

If the word **pointer** scares you, use the word **reference** instead. Whenever you see pointer-based phrases in this article, do a mental translation to a reference-based phrase:

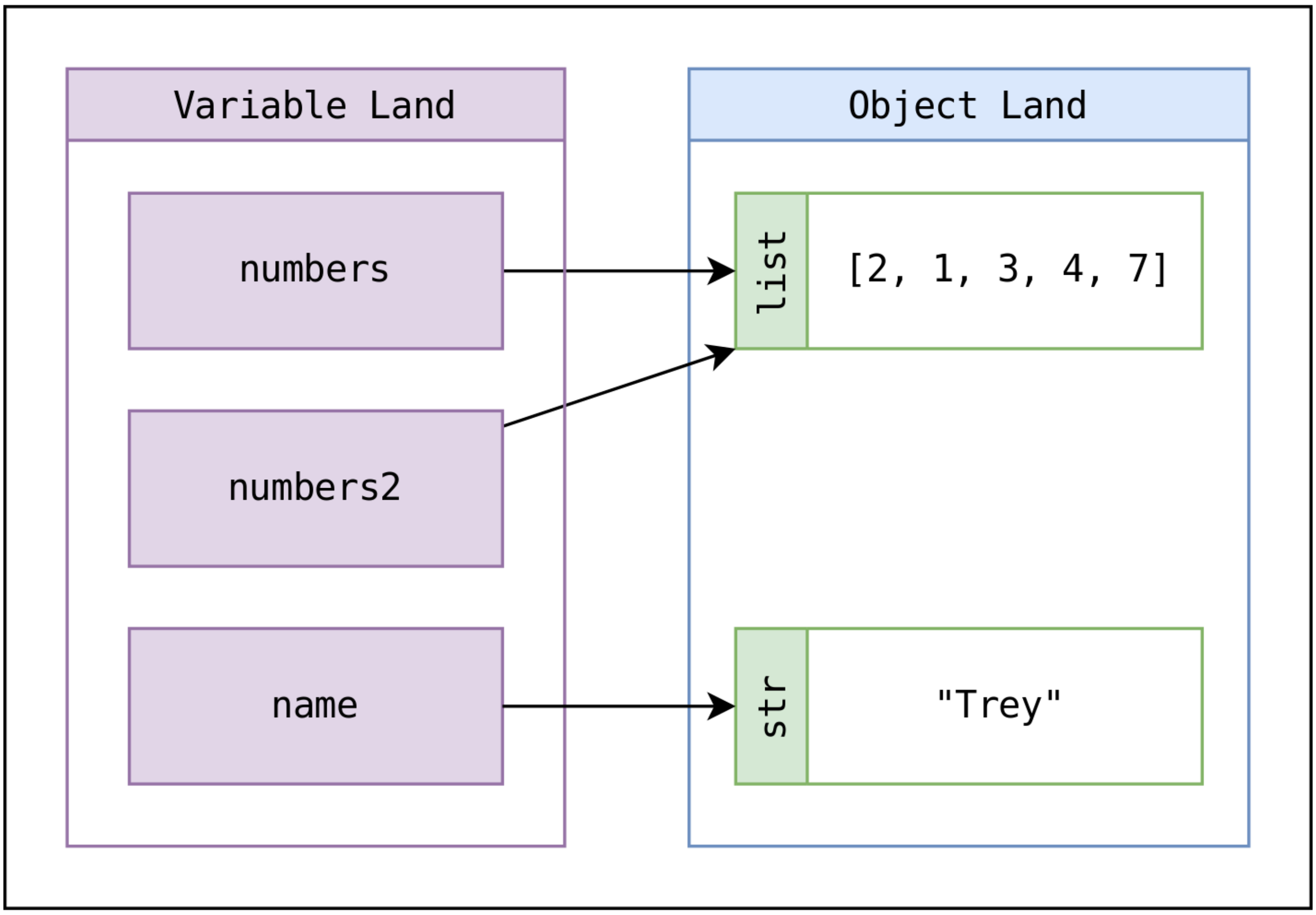
- **pointer ⇒ reference**
- **point to ⇒ refer to**
- **pointed to ⇒ referenced**
- **point X to Y ⇒ cause X to refer to Y**

Assignment statements point a variable to an object. That's it.

If we run this code:

```
>>> numbers = [2, 1, 3, 4, 7]
>>> numbers2 = numbers
>>> name = "Trey"
```

The state of our variables and objects would look like this:



Note that `numbers` and `numbers2` **point to the same object**. If we *change* that object, both variables will seem to "see" that change:

```
>>> numbers.pop()
7
>>> numbers
[2, 1, 3, 4]
>>> numbers2
[2, 1, 3, 4]
```

That strangeness was all due to this assignment statement:

```
>>> numbers2 = numbers
```

Assignment statements don't copy anything: they just point a variable to an object. So assigning one variable to another variable just **points two variables to the same object**.

The 2 types of "change" in Python

Python has 2 distinct types of "change":

1. **Assignment** changes a variable (it changes *which* object it points to)
2. **Mutation** changes an object (which any number of variables might point to)

The word "change" is often ambiguous. The phrase "we changed `x`" could mean "we re-assigned `x`" or it might mean "we mutated the object `x` points to".

Mutations change objects, not variables. But variables *point to* objects. So if another variable points to an object that *we've just mutated*, that other variable will reflect the same change; not because the variable changed but because **the object it points to** changed.

Equality compares objects and identity compares pointers

Python's `==` operator checks that two objects **represent the same data** (a.k.a. **equality**):

```
>>> my_numbers = [2, 1, 3, 4]
>>> your_numbers = [2, 1, 3, 4]
>>> my_numbers == your_numbers
True
```

Python's `is` operator checks whether two objects **are the same object** (a.k.a. **identity**):

```
>>> my_numbers is your_numbers
False
```

The variables `my_numbers` and `your_numbers` point to **objects representing the same data**, but the objects they point to **are not the same object**

```
>>> my_numbers[0] = 7
>>> my_numbers == your_numbers
False
```

If two variables point to the same object:

```
>>> my_numbers_again = my_numbers
>>> my_numbers is my_numbers_again
True
```

Changing the object one variable points also changes the object the other points to because they both point to the same object:

```
>>> my_numbers_again.append(7)
>>> my_numbers_again
[2, 1, 3, 4, 7]
>>> my_numbers
[2, 1, 3, 4, 7]
```

The `==` operator checks for **equality** and the `is` operator checks for **identity**. This distinction between identity and equality exists because variables **don't contain objects**, they **point to objects**.

In Python equality checks are very common and identity checks are very rare.

There's no exception for immutable objects

But wait, modifying a number *doesn't* change other variables pointing to the same number, right?

```
>>> n = 3
>>> m = n  # n and m point to the same number
>>> n += 2
>>> n  # n has changed
5
>>> m  # but m hasn't changed!
3
```

Well, **modifying a number is not possible** in Python. Numbers and strings are both **immutable**, meaning you can't mutate them. You **cannot change** an immutable object.

So what about that `+=` operator above? Didn't that mutate a number? (It didn't.)

With immutable objects, these two statements are equivalent:

```
>>> n += 2
>>> n = n + 2
```

For immutable objects, augmented assignments (`+=`, `*=`, `%=`, etc.) perform an operation (which returns a new object) and then do an assignment (to that new object).

Any operation you might *think* changes a string or a number instead returns a new object. Any operation on an immutable object always **returns a new object** instead of modifying the original.

Data structures contain pointers

Like variables, data structures **don't contain objects**, they **contain pointers to objects**.

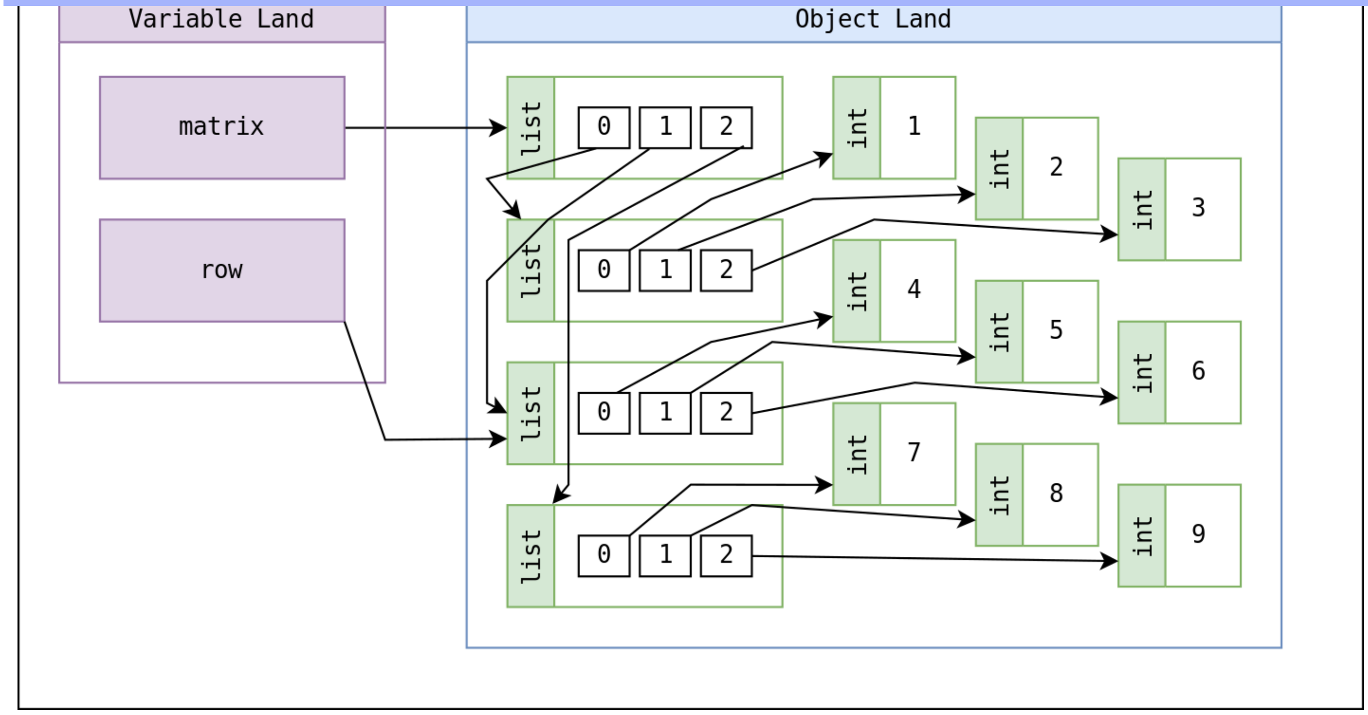
Let's say we make a list-of-lists:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

And then we make a variable pointing to the second list in our list-of-lists:

```
>>> row = matrix[1]
>>> row
[4, 5, 6]
```

The state of our variables and objects now looks like this:



Our `row` variable **points to the same object** as index `1` in our `matrix` list:

```
>>> row is matrix[1]
True
```

So if we mutate the list that `row` points to:

```
>>> row[0] = 1000
```

We'll see that change in both places:

```
>>> row
[1000, 5, 6]
>>> matrix
[[1, 2, 3], [1000, 5, 6], [7, 8, 9]]
```

It's common to speak of data structures "containing" objects, but they actually only contain pointers to objects.

Function arguments act like assignment statements

Function calls also perform assignments.

If you mutate an object that was passed-in to your function, you've mutated the original object:

```
>>> def smallest_n(items, n):
...     items.sort()  # This mutates the list (it sorts in-place)
...     return items[:n]
...
>>> numbers = [29, 7, 1, 4, 11, 18, 2]
>>> smallest_n(numbers, 4)
[1, 2, 4, 7]
>>> numbers
[1, 2, 4, 7, 11, 18, 29]
```

But if you reassign a variable to a different object, the original object will not change:

```
>>> def smallest_n(items, n):
...     items = sorted(items)  # this makes a new list (original is unchanged)
...     return items[:n]
...
>>> numbers = [29, 7, 1, 4, 11, 18, 2]
>>> smallest_n(numbers, 4)
[1, 2, 4, 7]
>>> numbers
[29, 7, 1, 4, 11, 18, 2]
```

We're reassigning the `items` variable here. That reassignment changes *which* object the `items` variable points to, but it doesn't change the original object.

We **changed an object** in the first case and we **changed a variable** in the second case.

Here's another example you'll sometimes see:

```
class Widget:
    def __init__(self, attrs=(), choices=()):
        self.attrs = list(attrs)
        self.choices = list(choices)
```

Class initializer methods often copy iterables given to them by making a new list out of their items. This allows the class to accept any iterable (not just lists) and decouples the original iterable from the class (modifying these lists won't upset the original caller). The above example was borrowed from Django.

Don't mutate the objects passed-in to your function unless the function caller expects you to.

Need to copy a list in Python?

```
>>> numbers = [2000, 1000, 3000]
```

You could call the `copy` method (if you're certain your iterable is a list):

```
>>> my_numbers = numbers.copy()
```

Or you could pass it to the `list` constructor (this works on **any iterable**):

```
>>> my_numbers = list(numbers)
```

Both of these techniques make a new list which **points to the same objects** as the original list.

The two lists are distinct, but the objects within them are the same:

```
>>> numbers is my_numbers
False
>>> numbers[0] is my_numbers[0]
True
```

Since integers (and all numbers) are immutable in Python we don't really care that each list contains the same objects because we can't mutate those objects anyway.

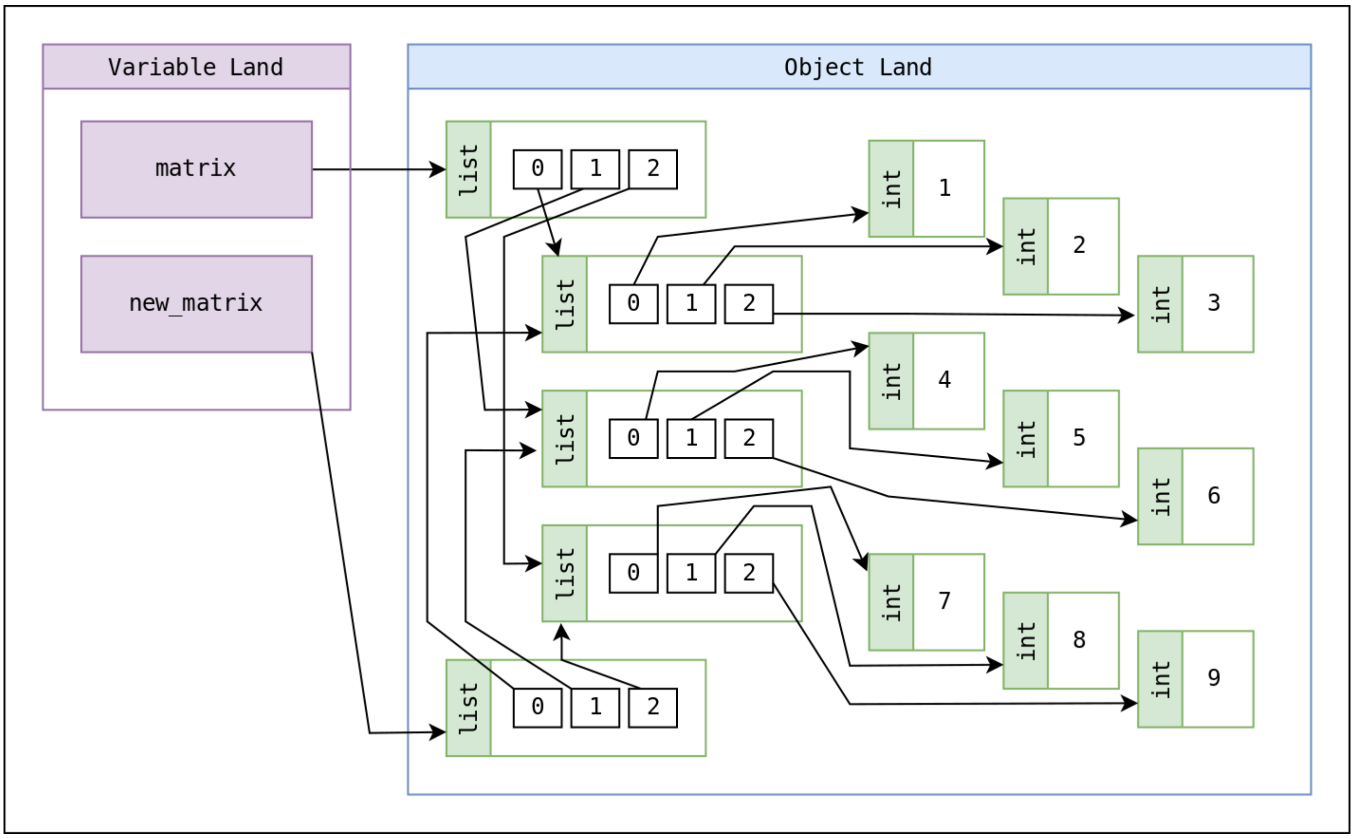
With mutable objects, this distinction matters. This makes two list-of-lists which each contain pointers to the same three lists:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> new_matrix = list(matrix)
```

These two lists aren't the same, but each item within them is the same:

```
>>> matrix is new_matrix
False
>>> matrix[0] is new_matrix[0]
True
```

Here's a rather complex visual representation of these two objects and the pointers they contain:



So if we mutate the first item in one list, it'll mutate the same item within the other list:

```
>>> matrix[0].append(100)
>>> matrix
[[1, 2, 3, 100], [4, 5, 6], [7, 8, 9]]
>>> new_matrix
[[1, 2, 3, 100], [4, 5, 6], [7, 8, 9]]
```

When you copy an object in Python, if that object **points to other objects**, you'll copy pointers to those other objects instead of copying the objects themselves.

New Python programmers respond to this behavior by sprinkling `copy.deepcopy` into their code. The `deepcopy` function attempts to recursively copy an object along with all objects it points to.

Sometimes new Python programmers will use `deepcopy` to recursively copy data structures:

```
tweet_data = [{"date": "Feb 04 2014", "text": "Hi Twitter"}, {"date": "Apr 16 2014", "text": "At #pycon2014"}]

# Parse date strings into datetime objects
processed_data = deepcopy(tweet_data)
for tweet in processed_data:
    tweet["date"] = datetime.strptime(tweet["date"], "%b %d %Y")
```

But in Python, we often prefer to make new objects instead of mutating existing objects. So we could entirely remove that `deepcopy` usage above by making a new list of new dictionaries instead of deep-copying our old list-of-dictionaries.

```
# Parse date strings into datetime objects
processed_data = [
    {**tweet, "date": datetime.strptime(tweet["date"], "%b %d %Y")}
    for tweet in tweet_data
]
```

We tend to prefer shallow copies in Python. If you **don't mutate objects that don't belong to you** you usually won't have any need for `deepcopy`.

The `deepcopy` function certainly has its uses, but it's often unnecessary. "How to avoid using `deepcopy`" warrants a separate discussion in a future article.

Summary

Variables in Python are not buckets containing things; they're **pointers** (they *point* to objects).

Python's model of variables and objects boils down to two primary rules:

1. **Mutation** changes an object
2. **Assignment** points a variable to an object

As well as these corollary rules:

1. **Reassigning** a variable points it to **a different object**, leaving the original object unchanged
2. **Assignments don't copy** anything, so it's up to you to copy objects as needed

Furthermore, data structures work the same way: lists and dictionaries container **pointers to objects** rather than the objects themselves. And attributes work the same way: **attributes point to objects** (just like any variable points to an object). So **objects cannot contain objects in Python** (they can only *point to* objects).

And note that while **mutations change objects** (not variables), multiple variables *can* point to the same object. If two variables point to the same object changes to that object will be seen when accessing either variable (because they both point to *the same* object).

For more on this topic see:

- My screencast series on Assignments and Mutation in Python
- Ned Batchelder's Python Names and Values talk
- Brandon Rhodes' Names, Objects, and Plummeting From The Cliff talk

This mental model of Python is tricky to internalize so it's okay if it still feels confusing! Python's features and best practices *often* nudge us toward "doing the right thing" automatically. But if your code is acting strangely, it might be due to changing an object you didn't mean to change.