Brandon Hoffman
Project 7.c

An assignment statement in python associates a symbolic name on the left-hand side with a value on the right hand side. This name or binding refers to the values.

```
n = 14
```

Now the name "`n`" refers to the value `14`.

In python, many names can refer to a single value so for instance:

```
n = 14
m = n
```

Now `m` and `n` are both assigned to the value `14`. In fact, they are both representing the same value in memory.

We might assume that "`m`" has made a copy of `n`, but in fact, assignment never copies data. `m` and `n` are simply referencing the same value in memory.

So what happens if we change the name or binding referring to a value shared by a different name or binding? Will they both change?

```
n = 14
m = n
n = 8
```

You might think that now `m` and `n` both had their values changed to `8`, but no, that is not the case; `m` is not making a copy of `n` it's simply referring the value `14` in memory, so when `n` is reassigned to `8`, `m`'s reference is still the same.

This is all intuitive, but things can get more complicated when we have more complicated objects, like a list:

Just like before we can assign multiple names to a value:

```
nums = [1, 2, 3]
num_list = nums
```

The behavior is as we expect, `nums` and `num_list` both print out the value `[1, 2, 3]`

Lists are what are called "mutable" which distinguishes it from the integer values we used in our example before. For us, mutable objects just simply mean that the value has methods that can change the value in-place. Immutable (which things like integers and strings fall under), have values that never change. In fact, when you are thinking you are changing them, for instance incrementing an integer by 1, you're not really change the value; you're really making a new value and assigning it to the same name.

But since lists are mutable, we can change them as they exist in memory and that will affect all the names or bindings that refer to that value.

For instance, if we try to append the value `4` to our list `nums` we can get some surprising behavior

```
nums = [1, 2, 3]
num_list = nums
nums.apppend(4)
```

Naturally, if we print the binding `nums` we will see the list `[1, 2, 3, 4]`. What we might not expect is that if we print the binding "`num_list`" we will also see the value represented as `[1, 2, 3, 4]`.

So what happened? Well, we mutated the underlying value. The append method manipulates the original object in memory that both n`ums` and `num_list` are referring to, so while you might assume "appending" `4` to the end of `nums`, only changes the binding `nums`. The change flows to all references dependent on the values associated with `nums`; in this case, `num_list`.

Naturally, this can trip up someone who isn't expecting aware of this behavior as every calculation or reference dependent upon these bindings will be impacted.