Here's what happens (approximately) with variables in Python. This is one of the hardest concepts in ESC180, so don't worry if you don't get it right away: just keep working on it.

1 The memory table and variable table

We represent the computer's memory using a *memory table*. The memory table tells us what data is stored at what address in the computer. In Python, we refer to data using named variables. At any point in time, each variable name is associated with an address. In order to figure out what data the variable refers to, we need to look up which address the variable refers to, and then figure out what's stored at that address. The variable tables are used to figure out what address each variable refers to. Note that the same variable name could refer to different addresses at different points in time.

We distinguish the integer 4200 from the address 4200 by using @4200 to denote the address.

Consider the following example:

Memory table

Address	Value
2000	42
2010	43
4000	`"hello"`

Variable table(globals)

Variable	Address
`n`	@2000
`greet`	@4000

The variable `greet` refers to address `4000`. Ad address `4000`, the string `"hello"` is stored. That means that we expect the following:

```
>> greet
hello
```

We can find out which address a variable refers to using Python's built-in `id()` function. `id(object)` returns the memory address of object[1]. In our example, we'd expect

```
>> id(n)
2000
```

2 Changes in the memory and variable tables

When Python starts up, the memory table looks something like this (the memory addresses are made up).

Memory table

Address	Value
1000	
1010	
1020	2
1030	3
1040	4
1050	

[1] technically, id() isn't always *quite* the address in all implementations of Python, but it's close enough for our purposes, and it is in CPython, which is the implementation of Python that we are using.

Some integers are already stored at some of the addresses[2]. For example, the integer 2 is stored in address (i.e., cell) 1020. When we assign a value to a variable, what really happens is that the variable name becomes associated with an address. For example, consider the following code:

```
a = 2
b = 4
```

Executing the code changes the variable table so that the addresses of the integers 2 and 4 are associated with the variable names `a` and `b`.

Variable table (globals)

Variable name	Address
a	@1020
b	@1040

We can associated new variable names with values that already exist in the memory table.

```
>> id(a)
1020
>> id(2)
1020
>> c = a
>> id(c)
1020
```

Basically, anything that refers to 2 (`a` and the literal `2` initially, and then `c` is well) has the same `id` (NOTE: technically, `id` isn't always *quite* the address in all implementations of Python, but it's close enough for our purposes, and it is in CPython, which is the implementation of Python that we are using.)

The variable table after executing the code above will be:

Variable table (globals)

Variable name	Address
a	@1020
b	@1040
c	@1020

3 Lists

Here's what happens with lists. When we define a list, it also goes into the memory table. Python finds an unoccupied space in memory, and places our list there. The contents of the list – the addresses of each of the elements – are all stored in memory.

Here is what happens when the following list is defined:

```
>> list0 = [2, 4]
```

[2] As we saw in lecture, CPython preloads integers in the range -4...256

Memory table

Address	Value
1000	[@1020, @1040]
1010	
1020	2
1030	3
1040	4
1050	

Variable table (globals)

Variable	Address
`a`	@1020
`b`	@1040
`c`	@1020
`list0`	@1000

3.1 Aliasing

Consider the following code:

```
>> id(list0)
1000
>> list1 = list0
>> id(list1)
1000
>> list0[0] = 3
>> list0
[3, 4]
>> list1
[3, 4]
```

What happened? `list0` and `list1` refer to the same address (1000), so any change in `list0` changes `list1` and vice versa, since they're the exact same list. We say that `list0` and `list1` are *aliases*.

Here are the memory and variable tables after the change was made:

Memory table

Address	Value
1000	[@1030, @1040]
1010	
1020	2
1030	3
1040	4
1050	

Variable table (globals)

Variable	Address
`a`	@1020
`b`	@1040
`c`	@1020
`list0`	@1000
`list1`	@1000

There are two important things to understand here.

First, when we go `list1 = list0`, the memory table does not change – only the variable table does. What happens is that `list1` starts referring to the address to which `list0` was referring.

Second, `list0[0]` means the first element of the list at address 1000. That's exactly the same as what `list1[0]` means. For that reason, after setting `list1 = list0`, changing `list0[0]` is exactly the same as changing `list1[0]`.

When we can change an object, we say that it is mutable (i.e., it can change). Lists are *mutable*. On the other hand, integers are *immutable* – while it's possible to make a variable that used to refer to the address containing 2 to refer to the address containing 3, that would be a change in the variable table, not a change in the memory table.

That's why we haven't encountered the issue of aliasing before. Everything we've seen up to now – strings, ints, floats – wasn't mutable. We can't change the value of 2, or 3.14, or `"hello"`[3].

3.2 Note: creating new objects in memory

We have already seen this happen with lists, but it's worth noting that integers (and strings, etc.) will not in generall have already been loaded in memory. For example, the following will find an empty slot in memory, put 42000 there, and then create an entry in the variable table making `n` refer to the memory slot where 42000 was put:

```
n = 42000
```

This is also applicable to lists:

```
>> list0 = [1, 2, 3]
>> list1 = [4, 5, 6]
>> list1 = list0
>> list0 = [10, 11, 12]
>> list1
[1, 2, 3]
```

Here, `list1` used to be an alias for `list0`. When we went `list0 = [10, 11, 12]`, it wasn't the case that the contents of `list0` changed; rather, `list0` started referring to a new list entirely. `list1`, meanwhile, kept referring to `list0`'s old address.

4 Functions and Local Variables

Consider the following example.

```
def change_list(L1):
    L1[0] = 4
    L1 = [3,2,1]
def g():
    L = [1,2,3]
    change_list(L)
    print(L) #[4, 2, 3]

if __name__ == '__main__':
    g()
```

What happens when we call `change_list(L)` is that implicitly, we assign `L1 = L`. That is, the local variable `L1` and `L` are aliases of each other.

Every time a function is called, a local variable table for that particular call is created. That local variable table is discarded after the call ends. If we call the same function twice, two separate copies of the table are created[4]

[3] Actually, we *sort of* can, see here:
`http://codegolf.stackexchange.com/questions/28786/write-a-program-that-makes-2-2-5/28851#28851`

[4] This is almost completely true. A complication arises when default parameters are used, but we will not get into this right now.

In this case, e global variables are just the functions `g`, `change_list`, the string `__name__`, etc. All the action is in `g()` and `change_list()`.

After `L = [1,2,3]` is executed inside of `g()`, here's the state of the memory (again, with the addresses made up). In this example, we take care to list more global variables than usual – for example, we list the global variable `__name__`.

Variable table (globals)

Variable	Address
`change_list()`	@20000
`g()`	@20100
`__name__`	@20200

Variable table (locals – g())

Variable	Address
`L`	@1000

Memory table

Address	Value
1000	[@1010, @1020, @1030]
1010	1
1020	2
1030	3
1040	4
1050	
...	
20000	`<change_list()>`
20100	`<g()>`
20200	`"__main__"`

When we call `change_list()` with the argument `L`, the parameter `L1` (a local variable) gets assigned the address @1000. Now, we execute `L1[0] = 4` inside the function `change_list()`. Since the variable `L1` inside `change_list()` refers to @1000, the effect is to change the first element of the list stored at address 1000. Note that there is just one list there.

Variable table (globals)

Variable	Address
`change_list()`	@20000
`g()`	@20100
`__name__`	@20200

Variable table (locals – `g()`)

Variable	Address
`L`	@1000

(locals – `change_list()`)

Variable	Address
`L1`	@1000

Memory table

Address	Value
1000	[@1040, @1020, @1030]
1010	1
1020	2
1030	3
1040	4
1050	
...	
20000	`<change_list()>`
20100	`<g()>`
20200	`"__main__"`

Now, we execute `L = [3,2,1]`. Several things happen: a new list is created, with the contents `[3,2,1]`. Then the address of that new list is stored in the local-to-`change_list()` variable `L1`:

Variable table (globals)

Variable	Address
`change_list()`	@20000
`g()`	@20100
`__name__`	@20200

Variable table (locals – `g()`)

Variable	Address
`L`	@1000

(locals – `change_list()`)

Variable	Address
`L1`	@1050

Memory table

Address	Value
1000	[@1040, @1020, @1030]
1010	1
1020	2
1030	3
1040	4
1050	[@1030, @1020, @1010]
...	
20000	`<change_list()>`
20100	`<g()>`
20200	`"__main__"`

Note that the local variable `L` in `g()` is unaffected. When wee returning from `change_list()`, the local variable `L` in `change_list()` will be discarded (though the list whose address it stores might persist for a while).

Variable table (globals)

Variable	Address
`change_list()`	@20000
`g()`	@20100
`__name__`	@20200

Variable table (locals – `g()`)

Variable	Address
`L`	@1000

(locals – `change_list()`)

Doesn't exist

Memory table

Address	Value
1000	[@1040, @1020, @1030]
1010	1
1020	2
1030	3
1040	4
1050	[@1030, @1020, @1010]
...	
20000	`<change_list()>`
20100	`<g()>`
20200	`"__main__"`

We're now back in `g()`, so that what gets printed is the list at address 1000.

Note that the story would be exactly the same (if slightly more confusing), if the code were:

```
def change_list(L):
    L[0] = 4
    L = [3,2,1]
def g():
    L = [1,2,3]
    change_list(L)
    print(L) #[4, 2, 3]

if __name__ == '__main__':
    g()
```

Everything would be exactly the same, except there would be two local variables called `L` – one in `change_list()`, and one in `g()`. The `L` in `change_list()` would behave exactly the same as the `L1` in `change_list()`.