C4M: The Python Memory Model

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

We have a memory table and a variables table.

Initially, the memory table looks something like this (the memory addresses are made up).

Address	**Value**
1000	
1010	
1020	2
1030	3
1040	4
1050	

Some integers are already stored at some of the addresses. 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 an address is assigned.

For example, take this code:

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

This causes the variable table to store the addresses of the values of 2 and 4, like so

Variable name	**Corresponding address**
a	1020
b	1040

In this hypothetical example, we’d expect the following output:

```
>> id(a)
1020

>> id(2)
1020

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

The built-in function `id(object)` returns the memory address of `object.`

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

Here's what happens with lists.

When we define a list, it also goes into the memory table.

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

Python finds an unoccupied space in memory, and places our list there. It also updates the variables table (we use the notation @1020 to mean that we are referring to the address 1020 rather than the integer 1020):

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

Memory table

Variable name	**Corresponding address**
a	1020
b	1040
list0	1000

Global variables

We can also call id() using arguments that are lists, and we can set variables to be lists.

```
>> id(list0)
1000

>> list1 = list0
>> id(list1)
1000

>> list0[0] = 5
>> print list0
[5, 4]
```

```
>> print list1
[5, 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*.

When we can change an object, we say that it is *mutable* (i.e., it can change). Lists are *mutable*.

Why didn't we encounter this problem before? Because everything we've seen up to now —strings, ints, floats—wasn't mutable (we say that, for example, ints are *immutable*). We can't change the value of 2, or 3.14, or "hello" (you might imagine that we can change the value of "hello", but in Python, we can't).

(Note: we *sort of* can, see here: http://codegolf.stackexchange.com/questions/28786/write-a-program-that-makes-2-2-5/28851#28851)

Note that we can still assign completely new lists or other values to existing variables:

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

Here, `list1` used to be an alias for `list0`, but then 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.

What about functions? Passing arguments to function is the same as creating aliases. Consider the following example:

```
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]

g()
```

The only complication here is that now we have different sets of local variables. Every time we call a function, a local variable table is created. The table is discarded after every call ends (so that if we call the same function twice, the table is created twice).

The 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, here's the state of the memory:

Global variables		**Locals [change_list]**	**Locals [g]**	
change_list()	20000	Doesn't exist	L	1000
g()	20002			
__name__	20004			
….				

Address	**Value**
1000	[@1010, @1020, @1030]
1010	1
1020	2
1030	3
1040	4
1050	

Memory table

Now, we call change_list with the parameter L, which at this point stores the address of the list, which is 1000.

In `g()`, the *local variable* `L` *of function* `change_list()` is set to the address `1000`. We have to variables called `L`: the local variable in function `g()`, and the local variable in function `change_list()`. They happen to be equal for now.

Now, we execute `L[0] = 1000` in the function `change_list().` That means that the contents of the list `L` change, though the list itself is still there.

Global variables		**Locals [change_list]**		**Locals [g]**	
change_list()	20000	L	1000	L	1000
g()	20002				
__name__	20004				
….					

Address	**Value**
1000	[1040, 1020,1030]
1010	1
1020	2
1030	3
1040	4
1050	

Memory table

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 `L`:

Global variables		**Locals [change_list]**		**Locals [g]**	
change_list()	20000	L	1050	L	1000
g()	20002				
__name__	20004				
….					

Address	**Value**
1000	[@1040, @1020, @1030]
1010	1
1020	2
1030	3
1040	4
1050	[@1030, @1020, @1010]

Memory table

Now we're returning from change_list(), which means the local variable L in change_list() is discarded (though the list whose address it stores might persist for a while).

<table>
<tr><th>Global variables</th><th>Locals [change_list]</th><th>Locals [g]</th></tr>
<tr>
<td>
<table>
<tr><td>change_list()</td><td>20000</td></tr>
<tr><td>g()</td><td>20002</td></tr>
<tr><td>__name__</td><td>20004</td></tr>
<tr><td>….</td><td></td></tr>
</table>
</td>
<td>Doesn't exist</td>
<td>
<table>
<tr><td>L</td><td>1000</td></tr>
</table>
</td>
</tr>
</table>

Address	**Value**
1000	[@1040, @1020, @1030]
1010	1
1020	2
1030	3
1040	4
1050	[@1030, @1020, @1010]

Memory table

We're now back in `g()`, so that what gets printed is the list at address `1000`.that point `[4, 2, 3].`