Handout 3 – Objects

[…]

Associate these words together:

- variable
- value
- lookup table
- assignment (the = operator)

i.e. "assigning a variable changes the value of the variable in the lookup table."

And associate these words together:

- object
- memory address / id
- heap
- modification (through member functions mostly, and some operators, e.g. L[0] = 2)

i.e. "modifying an object changes the actual object in the heap."

The lookup table basically has two columns: one for the variable name, and one for the variable value. Variables can have **two types of values**:

- **primitive types:** these are numbers (ints, floats) and booleans. We will not consider these as objects.
- **reference types:** these are **pointers** to objects (lists, tuples, strings and more) in the heap. (precisely, they are the memory addresses of the objects they point to.)

When we say x = 5, the value of the variable x is a primitive (the number 5). When we say y = "hello", the value of the variable y is a reference, to an object in the heap (the string "hello").

Aliasing occurs when multiple variables point to the same object:

```
a = [1, 2, 3]  <-- a is a pointer to a list object
b = a  <-- b is a pointer to the same list object that a points to
c = a  <-- same with c
d = b  <-- same with d
```

In the above example, we aliased manually/explicitly. So if we modified a (e.g. by calling append), we would see the change in b, c and d.

Aliasing can also occur automatically, in the case of strings:

```
a = "hello"  <-- all four of these have the same id
b = "hello"
c = "hello"
```

d = "hello"

So even though we wrote "hello" four different times, Python automatically aliased all of them.

As pointed out in class, I was wrong in the notes for Lab 7 -- tuples do NOT automatically alias. If you did the above example with (1, 2, 3) instead of "hello", you would indeed have four different objects. My mistake.

Scope refers to the frame you're looking at in the stack diagram. When inside a function, you're in a **local scope**, meaning any variables you create will only live inside the function -- they're local to the function.

However, remember that variables can be references/pointers to objects in the heap, so if a local variable is pointing to the same object as a variable outside of the function, the variable outside of the function will still see any modifications to the object.

Make sure to review problem 3 of lab 7, all the subtle changes to the functions and understand why they happen. Draw stack diagrams to help you. Keep considering whether a statement is an assignment (that changes a variable) or a modification (that changes an object).

[…]