6.189 - Intro to Python
IAP 2008 - Class 6
Lead: Aseem Kishore

Lab 7: Introduction to Objects

Problem 1 – Exploring values and references

So far, we have used variables with types int, float, bool, string, list and tuple. It turns out that some of these are treated as **values**, whereas others are treated as **references**. To understand what this means and what it implies for how we program, this problem will help you explore the various types.

Start a new Python shell (or restart an existing one), and type in the following:

```
a = “python”
b = “python”
```

I told you that in this case, a and b actually refer to the same object in the heap. But how do you really know? Python provides an “id” function that takes a variable and returns the **memory address** of the object that’s actually referenced.

Try this:

```
print id(a)
print id(b)
print id(a) == id(b)
```

So it turns out the two are in fact the same. We can compare in a more concise way – Python provides an “is” keyword that tells us if two variables reference the same object or not.

Try this:

```
print a is b
```

Now try the same code above on something different. Instead of “python”, try the number 5, the number 5.0, and the boolean True. What do you find?

In all of the above cases, you can see that all those **values** (“python”, 5, 5.0, True) do in fact have memory addresses. In fact, every single value in Python has a memory address. For these basic types, however – string, int, float and bool – we will treat them as **primitives**, meaning that we won’t consider them as objects for the most part (strings are the only exception). We’ll understand why soon.

Question 1 – What happens if you say:

```
a = 5
b = 5.0
```

Are a and b still referencing the same memory address? Do they evaluate to different numerical values? To answer, try these two lines:

```
print a is b
print a == b
```

Interesting! So as a general law – *objects of different types will always reside at different memory addresses*. So, a is b will always evaluate to False if a and b are different types. This also tells you – *don't use "is" if == will do*.

Now, let's explore lists. Try this:

```
a = [1, 2, 3]
b = [1, 2, 3]
```

Are they the same object? Again, you can check with the "is" keyword, or see their actual memory addresses with the "id" function:

```
print id(a)
print id(b)
print id(a) == id(b)
print a is b
```

And again, let's check if they're treated as equal:

```
print a == b
```

So lists are treated differently from other things we've seen. They're different objects, but like math, they will be equal if they are made up of the same elements. Let's really check that they're different objects:

```
a[1] = 70
print a
print b
print a == b
```

Changes made to one list don't affect the other!

Question 2 – What happens if you say:

```
a = [1, 2, 3]
b = a
```

Are a and b referencing the same object? Are they equal? What happens if you change list a, does list b change as well?

This concept is called **aliasing**, where two variables reference the same object. With the primitive types that we saw above, Python automatically aliased our variables if they were the same value and type. With lists, we have to explicitly alias if we want the same object.

Question 3 – Are tuples aliased automatically, like primitive types, or do we have to explicitly alias them like lists?

Problem 2 – Exploring mutability

From this point on, we need to be consistent with our terms, and we need to be solid on understanding how the computer thinks and works:

- To **assign** a value to a variable means something like:

```
x = 5
y = “hello”
z = [1, 2, 3]
```

- To **modify** or **mutate** a variable means something like:

```
z.append(4)
```

In general, assignment occurs with the assignment operator (=), and modification/mutation occurs with **member functions** – we haven’t really learned about these yet, but they occur with the member operator (.) followed by a function that the object supports. In this case, append is a function that all list objects support.

However, to make things more confusing, there are statements like these:

```
z[2] = “three”
del z[1]
```

The first statement looks like an assignment, and it is – it’s an assignment to a variable inside a list (remember, lists are collections of variables – they’re useful when we have a dynamic number of variables). However, we don’t really care about that aspect. Instead, we care more that the first statement is ALSO a modification – it modifies the list z. Same with the second statement. So for now, treat list index assignment and deletion as modification.

From these definitions, we say that an object is **mutable** if it (the object itself) can be changed in any way. Similarly, we say that an object is **immutable** if it can’t be, i.e. it doesn’t support any operations or functions that mutate it.

So, back to what we were doing. We’ve found that strings are aliased automatically for us, while lists are not. And we’ve found that tuples are also automatically aliased!

Remember how we said strings and tuples are similar? This is no different. Both are arguably complex objects (you have to keep track of how many elements/letters you have, and dynamically allocate that much space for them – yes, strings and tuples are implemented on the inside with some sorts of lists!).

However, *because both are immutable types* (as in, we can’t modify any string or tuple), and because both are commonly used repeatedly, Python saves memory by aliasing automatically. But because lists are mutable, Python does not dare alias.

Just like we have objects that are mutable and immutable, we can specify it further. Since we know mutability is caused by functions or operations that change an object, we can specify whether functions will cause an object to be mutable or not.

We say a function is a **pure function** if it makes no modifications to the object it's working on. Similarly, we say a function is a **modifier function** if it does. Another way of referring to these functions is that they have **side effects**.

Why the term side effects? We've been teaching you guys to think of functions as similar to math functions – *in general*, you shouldn't take input from within a function, and you shouldn't print stuff from within a function. Instead, you should **return** a value, and the function will get evaluated to that value, just like sqrt(4) gets evaluated to 2.

Well, this style is so important that in computer science, it's natural to think of functions as these mathematical-type functions. So when they do anything other than simply return a value – anything that we can notice after the function is done executing – we consider those to be side effects. And when a function has side effects, it should be clear to know.

We know that strings and tuples are immutable, so none of the operations or functions they support should have any side effects. We'll explore some of these operations and make sure.

Try these:

```
a = “hello”
print a.index(“e”)      ← prints the index of the first “e”
print a
print a.upper()
print a
```

The first function, index, makes sense – we don't expect it to change the string since we're just finding an element, not changing it. But what about upper? As you can see, upper doesn't change the string a. Instead, it returns a new string.

This is a design pattern you'll see over and over – when objects are immutable, then they will often have these types of **factory functions** (or **creators**) that *return new objects* rather than modify the original.

Now let's explore lists. We know that lists are mutable, so we expect that some functions will have side effects.

Try these:

```
b = [‘x’, ‘y’, ‘z’]
print b.count(‘y’)      ← prints how many occurrences of ‘y’ there are
print b
print b.reverse()
print b
```

Again, we didn't expect count to change the list. But what about reverse? This function actually modified b, so we say that a side effect of reverse is that the list b is reversed. But what did that line print?

Try this:

```
c = None
```

```
print c
print type(c)
print c == False
print c == 0
```

None is a special type in Python that literally represents the idea that there is nothing here. In other languages, the same idea is often given the name “null”. It is equal to nothing else, and the only way to refer to it is by its value, None.

Question 4 – Is None aliased automatically by Python?

Getting back to mutability, we saw that the line printed None. Try this:

```
print b.reverse() == None
```

Remember, we call functions by putting parentheses after the function name (optionally with arguments inside; in this case, reverse doesn’t take any arguments). And when we call a function, if it returns anything, the function gets evaluated with the return value.

So does this mean that reverse returns None? Maybe, we don’t know. It turns out that if a function *doesn’t* specify a return value (i.e. by explicitly declaring return ___ before the function is finished executing), then Python evaluates the function to be None.

This leads to another common design pattern – when functions have side effects, they will often return None to make their purpose explicit.

Question 5 – Try this:

```
x = [1, 2, 3]
x = x.append(4)
print x
```

What gets printed? Why? How do we fix it if we want x == [1, 2, 3, 4]?

Question 6 – Now try this:

```
y = [1, 2, 3, 4]
print y.pop()
print y
```

Has y changed? Or did pop return a value? Why? If you’re confused, try this:

```
help(y.pop)
```

With the last example, you can see an example of an exception to the regular design pattern. But that’s fine – as long as the side effect is made explicit and clear, having a useful return value is not a problem. But the opposite is not fine – don’t have any side effects if the user does not expect them, and instead expects a return value.

Problem 3 – Exploring scope

Now that we understand references and aliasing and mutability, let’s add one more element and then put it all together.

Make sure you are comfortable with the concept of local variables. Draw stack diagrams if they are useful. Now, we’ll apply the same idea of local variables inside functions, but we’ll observe whether the local variables are references or values.

Try this:

```
def foo(x):
  print “point 2:”, id(x)
  x = [1, 2, 3]
  print “point 3:”, id(x)

L = [1, 2, 3]
print “point 1:”, id(L)
foo(L)
print “point 4:”, id(L)
```

We see that when L is passed to the function foo, it is passed by its reference. Just like when we think about scope normally, inside the function foo is a local variable x. Initially, x becomes whatever L evaluates to – which is the reference to the actual list object. However, when we re-assign x, we’re only changing the value of the local variable x. And again, since lists aren’t aliased, x now points to a different object. And, as we expect from scope, when we exit the function, the original variable L didn’t change its value.

Question 7 – Try this subtle variation instead:

```
def foo(L):
  print “point 2:”, id(L)
  L = [1, 2, 3]
  print “point 3:”, id(L)

L = [1, 2, 3]
print “point 1:”, id(L)
foo(L)
print “point 4:”, id(L)
```

What happens? Is it the same as above? Why or why not?

Question 8 – Try another subtle variation instead:

```
def foo():
  print “point 2:”, id(L)
  L = [1, 2, 3]
  print “point 3:”, id(L)

L = [1, 2, 3]
print “point 1:”, id(L)
```

```
foo()
print “point 4:”, id(L)
```

What happens? Is it the same as above? Why or why not?

So far, this has had nothing to do with mutability, but it does confirm that scope works with values just as it works with references – *assignments to local variables inside a function will not change the value of variables declared outside the function*.

And another important point – *a variable will first be looked up in the local frame/stack, and if it’s not found, then it will be looked up in an outside stack*. The last part is a little trickier than we think, so won’t worry about the details. Just remember that Python will always first look for the variable in the local stack.

Now let’s explore mutability with scope. Remember that modifying an object is different from assigning a variable.

Question 9 – Try this:

```
def foo(x):
  print “point 2:”, id(x)
  x.append(4)
  print “point 3:”, id(x)

L = [1, 2, 3]
print “point 1:”, id(L)
foo(L)
print “point 4:”, id(L)
```

What happens? Are the lists the same, or different? Why or why not?

Question 10 – Try this:

```
def foo(x):
  print “point 2:”, x
  x.append(4)
  print “point 3:”, x

L = [1, 2, 3]
print “point 1:”, L
foo(L)
print “point 4:”, L
```

Has L changed after the call of foo? Why or why not?

This example shows us that when we pass objects to a function, any modifications to those objects will persist beyond that function. Why? Because we’re passing the reference to that object, so the local variable x inside the function will refer to the same object.

Great – now you understand how to work with objects! Since objects are really useful for abstraction, you’re now ready to tackle some really interesting problems.