Python Copies

November 2, 2015

Reference: Think, 89-100

1 Identifiers

All objects (variables) in Python have a unique numeric id. The id never changes during the object's lifetime. Python will report an object's unique ID as type `int` via the `id()` function. Some examples:

```
In [2]: a = 9
        print id(a)
```

```
5745776
```

```
In [3]: b = 'sss'
        print id(b)
```

```
51745256
```

```
In [1]: print id( [7.26, 9, 'a'] )
```

```
50015008
```

```
In [6]: print type( id(17.3) )
```

```
<type 'int'>
```

But what is the id actually refering to?

In the standard C-based version of Python `id(object)` returns the object's address in the computer's memory. Every object is stored somewhere in memory, and that place has a numerical address like a house on a street. The address is the `id`. In the same way that we talk about the residents of a house, we can talk about the value stored at a given id. However, the memory address is relative to the program. When you run the code above you will (most likely) get a different response, because the available memory in your computer's RAM will be different than on mine.

The term "reference" is often used to mean an object's address or its id. For example, we say that a list is passed to a function by reference. This means that that when the functon is called the actual values are not passed into the function, instead the function receives a reference that points to the location in memory where the values are stored. This is important for memory management. If you are passing large objects back and forth throughout the execution of a program then you will be using much more memory than if you simply pass the reference to large objects.

A list of tuples, containing ecological data for every ecozone in North America and the points needed to represent their polygons this information could be hundreds of megabytes in size. If we didn't use references, or pointers, the functions calling the polygons (clipping, selecting, manipulating) could rapidly create gigabytes worth of objects in the RAM that would grind your program to a halt.

Some languages require that programmers have a good understanding of how references are used, and often a programmer will manipulate addresses directly, such as directly assigning or manipulating values at specific addresses. Python is similar to Java and C# in that it does not allow direct manipulation of memory addresses. However, certain particulars of memory allocation in Python do require some background since copying objects have some very specific implications for object management.

2 Aliases and Clones

Consider the following:

```
In [3]: a = 9
        print id(a)

        b = a
        print id(b)
```

```
29797488
29797488
```

In this case `b` is an alias of `a` because they refer to the same object (the integer 9). This object is considered "aliased" because it has more than one reference: 9 has the alias `a` and `b`. Both `a` and `b` point to the same physical memory location. For small objects this doesn't have much of an effect of memory management, but if the objects were very long lists, dictionaries or tuples then this means that Python only needs to manage one instance of the object since any copy simply points to that physical location in memory.

If we then set `b` to some other value, its id will change, but not the id of `a`:

```
In [4]: b = 10
        print 'a doesn\'t change its location:', id(a), ', but b does:', id(b)
```

```
a doesn't change its location: 29797488 , but b does: 29797476
```

We've been doing this for individual integer values, but this also work with lists (as it should for reasons outlied above):

```
In [5]: a = [1,2,3]
        b = a
        print id(a),id(b)
```

```
50013808 50013808
```

But here's the really critical issue: `b` is an alias of `a`, both point to the same address, so changing an element of `b` actually changes `a`. BEWARE! These are not copies, they are aliases.

```
In [6]: b[1] = 200
        print a
```

```
[1, 200, 3]
```

Our two lists `a` and `b` are the same. Suppose we want two separate lists with the same elements? *i.e.*, suppose we want a copy, not an alias? Maybe we want to start from the same base point but then apply different transformations to the datasets to compare them at some point in the analysis.

The `list()` function can help resolve this issue somewhat, but not completely:

2.1 Example 9.1:

```
In [1]: a = [ [97,'a'], [98,'b'] ]
        c = a
        b = list(a)

        print '   address of a:',id(a)
        print '   address of b:',id(b),'\n'

        print 'address of a[0]:',id(a[0])
```

```
        print 'address of b[0]:',id(b[0]),'\n'

        print 'address of a[1]:',id(a[1])
        print 'address of b[1]:',id(b[1])

address of a: 49719216
   address of b: 51673128

address of a[0]: 50295312
address of b[0]: 50295312

address of a[1]: 50235872
address of b[1]: 50235872
```

So this is a bit confusing. The lists are distinct, but not the individual elements within the list. So `a` is distinct from `b`, but `a[0]` and `b[0]` both point to the same address. If we change `a[0]`, `b[0]` will also change. Technically `b` is a **clone** of `a`. It is not a true alias, but the alias issue is there.

The alias problem exists for dictionaries, even when using the `dict()` function. It exists for tuples as well. If you change a mutable element of a tuple, the aliased object will also change.

2.2 Exercise 9.1:

Create an example proving the same issue exists for dictionaries.

Go to the Exercise 9.1. PDF on Learn@UW

3 Shallow vs. Deep Copies

We are seeing that we are making shallow copies, we are copying a value that points to the original data. The `id()`'s of complex objects are copied, but not the actual contents.

The `copy` module has a function that will handle this issue. The function is called `deepcopy`. This function does what it sounds like. It creates a new copy of the object in memeory and a new copy for each element of that object.

3.1 Example 9.2:

```
In [7]: import copy

        a = [ [97,'a'], [98,'b'] ]
        b = copy.deepcopy(a)

        print '   address of a:',id(a)
        print '   address of b:',id(b),'\n'

        print 'address of a[0]:',id(a[0])
        print 'address of b[0]:',id(b[0]),'\n'

        print 'address of a[1]:',id(a[1])
        print 'address of b[1]:',id(b[1])

address of a: 50015648
   address of b: 50015288

address of a[0]: 50066296
address of b[0]: 50015848
```

```
address of a[1]: 50015488
address of b[1]: 50015768
```

Now we've created lists that contain the same values, but that are distinct with respect to memory management. The lists are different, and so are their items.

Bottom Line: use `deepcopy()` if you want truly distinct dictionaries or lists.

Note: `copy.deepcopy()` can be used with (almost) any object, not just lists, dictionaries and tuples. The `copy` module also contains a function called `copy`. This makes shallow copies, not deep copies.

Recap: A shallow `copy` constructs a new object and fills the new object with references of objects found in the original. A `deepcopy` constructs a new object and fills the new object with copies of objects found in the original. It does this recursively. [This is a slight simplification of what happens behind the scenes]

3.2 Exercise 9.2:

Write a function `showIDs(seq)` that accepts a sequence `seq` as an argument. The function will print the address of the sequence and the address of every member of the sequence.

Embed your function in a program containing these lines:

```
In [ ]: import copy

        x = 2*[ 4*[1.5] ]
        y = x
        z = copy.deepcopy(x)

        print x
        showIDs(x)
        showIDs(y)
        showIDs(z)
```

Explain the output. In particular: a) What does the list x contain? b) Why are the ids of the elements of x the same? c) Why are the ids of the elements of x the same as those of y? d) Why are the ids of z different than those of y and x? e) Why are the ids of z the same?

Go to the Exercise 9.2. PDF on Learn@UW