tutorial_1

February 7, 2018

1 Python Notional Machine -- Continued from Lecture 1

Our goal is to refresh ourselves on basics (and some subtleties) associated with Python's data and computational model. Along the way, we'll also use or refresh ourselves on the environment model as a way to think about and keep track of the effect of executing python code.

1.1 Variables and data types

1.1.1 Integers

```
In [1]: a = 7
        b = a
        print('a:', a, '\nb:', b)
```

```
a: 7
b: 7
```

```
In [2]: a = a + 10
        a += 100
        print('a:', a, '\nb:', b)
```

```
a: 117
b: 7
```

So far so good -- integers, and variables pointing to integers, are straightforward.

1.1.2 Lists

```
In [3]: x = ['baz', [1, 2], 3, 4]
        print('x:', x)
```

```
x: ['baz', [1, 2], 3, 4]
```

```
In [4]: y = x
        print('y:', y)
```

```
y: ['baz', [1, 2], 3, 4]
```

```
In [5]: x = 77
        print('x:', x, '\ny:', y)
```

```
x: 77
y: ['baz', [1, 2], 3, 4]
```

Unlike integers, lists are mutable:

```
In [6]: x = y
        x[0] = 88
        print('x:', x)
```

```
x: [88, [1, 2], 3, 4]
```

```
In [7]: print('y:', y)
```

```
y: [88, [1, 2], 3, 4]
```

As seen above, we have to be careful about sharing (also known as "aliasing") mutable data!

```
In [8]: a = [1, 2, 3]
        b = [a, a, a]
        print(b)
```

```
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

```
In [9]: b[0][0] = 4
        print(b)
```

```
[[4, 2, 3], [4, 2, 3], [4, 2, 3]]
```

1.1.3 Tuples

Tuples are a lot like lists, except that they are immutable.

```
In [10]: x = ('baz', [1, 2], 3, 4)
         y = x
         print('x:', x, '\ny:', y)
```

```
x: ('baz', [1, 2], 3, 4)
y: ('baz', [1, 2], 3, 4)
```

Unlike a list, we can't change the top most structure of a tuple; trying to change it results in an error:

```
In [11]: x[0] = 88
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-11-00de764f1e8d> in <module>()
----> 1 x[0] = 88


TypeError: 'tuple' object does not support item assignment
```

What will happen in the following (operating on x)?

```
In [12]: x[1][0] = 11
         print('x:', x, '\ny:', y)
```

```
x: ('baz', [11, 2], 3, 4)
y: ('baz', [11, 2], 3, 4)
```

So we still need to be careful! The tuple didn't change at the top level -- but it might have members that are themselves mutable.

1.1.4 Strings

Strings are also immutable. We can't change them once created.

```
In [13]: a = 'hi'
         b = a + 'gh'
         print('a:', a, '\nb:', b)
```

```
a: hi
b: high
```

```
In [14]: a[0] = 'H'
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-14-ecb470cdfa8a> in <module>()
----> 1 a[0] = 'H'
```

```
TypeError: 'str' object does not support item assignment
```

```
In [15]: c = 'hello'
         d = c
         c += ' there'
         print('c:', c, '\nd:', d)
```

```
c: hello there
d: hello
```

That's a little bit tricky. Here the '+=' operator makes a copy of c first to use as part of the new string with ' there' included at the end.

1.1.5 Back to lists: append, extend, and the '+' and '+=' operators

```
In [16]: x = [1, 2, 3]
         y = x
         x.append([4, 5])
         print('x:', x, '\ny:', y)
```

```
x: [1, 2, 3, [4, 5]]
y: [1, 2, 3, [4, 5]]
```

So again, we have to watch out for aliasing/sharing, whenever we mutate an object.

```
In [17]: x = [1, 2, 3]
         y = x
         x.extend([4, 5])
         print('x:', x, '\ny:', y)
```

```
x: [1, 2, 3, 4, 5]
y: [1, 2, 3, 4, 5]
```

Here's an interesting case, to check understanding of the '+' operator used on lists:

```
In [18]: x = [1, 2, 3]
         y = x
         x = x + [4, 5]
         print('x:', x)
```

```
x: [1, 2, 3, 4, 5]
```

So the '+' operator on a list looks sort of like extend. But has it changed x in place, or made a copy of x first for use in the longer list?

And what happens to y in the above?

```
In [19]: print('y:', y)

y: [1, 2, 3]
```

So that clarifies things -- the "+" operator on a list makes a (shallow) copy of the left argument first, then uses that copy in the new larger list.

Another case, this time using the "+=" operator with a list. Note: in the case of integers, a = a + and a += gave exactly the same result. How about in the case of lists?

```
In [20]: x = [1, 2, 3]
         y = x
         x += [4, 5]
         print('x:', x, '\ny:', y)

x: [1, 2, 3, 4, 5]
y: [1, 2, 3, 4, 5]
```

So x += is NOT the same thing as x = x + if x is a list! Here it actually DOES mutate or change x in place, if that is allowed (i.e., if x is a mutable object).

Contrast this with the same thing, but for x in the case where x was a string. Since strings are immutable, python does not change x in place. Rather, the += operator is overloaded to do a top-level copy of the target, make that copy part of the new larger object, and assign that new object to the variable.

Let's check your understanding. What will happen in the following, that looks just like the code above for lists, but instead using tuples. What will x and y be after executing this?

```
In [21]: x = (1, 2, 3)
         y = x
         x += (4, 5)
         print('x:', x, '\ny:', y)

x: (1, 2, 3, 4, 5)
y: (1, 2, 3)
```

1.1.6 Other data types for you to refresh yourself on (later): sets, dictionaries

If we have enough time, we can come back to and refresh ourselves on sets and dictionaries. We'll find those useful in later labs, but don't need them for Lab 1.

1.2 Functions and scoping

```
In [22]: x = 100
         def foo(y):
             return x + y
         z = foo(7)
         print('x:', x, '\nfoo:', foo, '\nz:', z)
```

```
x: 100
foo: <function foo at 0x000002BDCD8BD7B8>
z: 107
```

```
In [23]: def bar(x):
             x = 1000
             return foo(7)
         w = bar('hi')
         print('x:', x, '\nw:', w)
```

```
x: 100
w: 107
```

Importantly, foo "remembers" that it was created in the global environment, so looks in the global environment to find a value for 'x'. It does NOT look back in its "call chain"; rather, it looks back in its parent environment.

1.2.1 Optional arguments and default values

```
In [24]: def foo(x, y = []):
             y = y + [x]
             return y

         a = foo(7)
         b = foo(88, [1, 2, 3])
         print('a:', a, '\nb:', b)
```

```
a: [7]
b: [1, 2, 3, 88]
```

```
In [25]: c = foo(7)
         print('a:', a, '\nb:', b, '\nc:', c)
```

```
a: [7]
b: [1, 2, 3, 88]
c: [7]
```

Let's try something that looks close to the same thing... but with an important difference!

```
In [26]: def foo(x, y = []):
             y.append(x)   # different here
             return y

         a = foo(7)
         b = foo(88, [1, 2, 3])
         print('a:', a, '\nb:', b)
```

```
a: [7]
b: [1, 2, 3, 88]
```

Okay, so far it looks the same as with the earlier foo.

```
In [27]: c = foo(7)
         print('a:', a, '\nb:', b, '\nc:', c)
```

```
a: [7, 7]
b: [1, 2, 3, 88]
c: [7, 7]
```

So quite different... all kinds of aliasing going on. The moral here is to be VERY careful (and indeed it may be best to simply avoid) having optional/default arguments that are mutable structures like lists... it's hard to remember or debug such aliasing!

1.3 Closures

```
In [28]: def add_n(n):
             def inner(x):
                 return x + n
             return inner
```

```
In [29]: add1 = add_n(1)
         add2 = add_n(2)

         print(add2(3))
         print(add1(7))
         print(add_n(8)(9))
```

```
5
8
17
```

1.4 Brain Teasers

What happens when this program is run? 0. It prints 12, then 13, then ..., then 16 1. It prints 13, then 14, then ..., then 17 2. It prints 16, then 15, then ..., then 12 3. It prints 17, then 16, then ..., then 13 4. A Python error occurs 5. Something else

```
In [30]: functions = []
         for i in range(5):
             def func(x):
                 return x + i
             functions.append(func)

         for f in functions:
             print(f(12))
```

```
16
16
16
16
16
```

Compare with the following:

```
In [31]: functions = []
         for i in range(5):
             def makefunc(n):
                 def func(x):
                     return x + n
                 return func
             functions.append(makefunc(i))

         for f in functions:
             print(f(12))
```

```
12
13
14
15
16
```

Another approach:

```
In [32]: functions = []
         for i in range(5):
             def func(x, i=i):
                 return x + i
             functions.append(func)

         for f in functions:
             print(f(12))
```

```
12
13
```

```
14
15
16
```

1.5 Classes

```
In [33]: x = 'global var'
         class Simple:
             x = 'class var'

         print(Simple.x)
```

```
class var
```

```
In [34]: x = 'global var'
         class Simple:
             x = 'class var'

         s = Simple()
         print(s.x)
```

```
class var
```

```
In [35]: x = 'global var'
         class Simple:
             x = 'class var'

         s = Simple()
         s.x = 'instance var'
         print(s.x)
         print(Simple.x)
```

```
instance var
class var
```

```
In [36]: x = 'global var'
         class Simple:
             x = 'class var'
             def __init__(self):
                 x = 'local var'

         s = Simple()
         print(s.x)
```

```
class var
```

```
In [37]: x = 'global var'
         class Simple:
             x = 'class var'
             def __init__(self):
                 x = 'local var'
                 self.x = 'instance var'

         s = Simple()
         print(s.x)
```

```
instance var
```

```
In [38]: x = 'global var'
         class Simple:
             x = 'class var'
             def __init__(self):
                 x = 'local var'
                 self.x = 'instance var'
             def which_x(self):
                 return x

         s = Simple()
         print(s.which_x())
```

```
global var
```

1.6 Reference Counting

This is an advanced feature you don't need to know about, but you might be curious about. Python knows to throw away an object when its "reference counter" reaches zero. You can inspect the current value of an object's reference counter with `sys.getrefcount`.

```
In [39]: import sys
         L1 = [1, 2, 3]
         print(sys.getrefcount(L1))
         L2 = L1
         print(sys.getrefcount(L1))
         L3 = [L1, L1, L1]
         print(sys.getrefcount(L1))
         L3.pop()
         print(sys.getrefcount(L1))
         L3 = 7
         print(sys.getrefcount(L1))
```

```
2
3
6
```

```
5
3
```

```
In [40]: abc = 0
         print(sys.getrefcount(123))
         abc = 123
         print(sys.getrefcount(123))
```

```
15
16
```

1.7 Readings -- if you want/need more refreshers

Check out the 6.s080 readings:

Assignment and aliasing: What is an environment? What is a frame? Discussed in https://py.mit.edu/6.s080/assignment0.0/readings

Functions: What happens when one is defined? What happens when one is called? Simple example in sections 4-5 of https://py.mit.edu/6.s080/assignment1.0/readings

Closures in 2.3-2.4 of https://py.mit.edu/6.s080/assignment1.1/readings

Classes: What is a class? What is an instance? What is self? What is *_init_*? Discussed in https://py.mit.edu6.s080/assignment2.0/readings

Inheritance in sections 1-3 of https://py.mit.edu/6.s080/assignment2.1/readings