6.009 Information

Website: [https://mit.edu/6.009 (https://mit.edu/6.009)](https://mit.edu/6.009)

- Location for subject information, lab distributions and submissions.
- Also sign up for Piazza forum, can ask questions there (no emails please!). Use private postings if you include code, but other please post publically so Q&A are shared.

Instructor presentations

- Lectures: Mon 11-12:30, 26-100
- Tutorials: Wed (9-11am, 1-3pm), first 30-60 mins of session

Office hours (code help, checkoff meetings using queue system)

- Wed (9-11am, 1-3pm), last 60-90 mins of session
- Fri (9-11am, 10am-12, 1-3pm, 2-4pm), 110 min sessions
- Sun (1-10pm), Mon (1-3pm, 7-10pm), and Tue-Thu (7-10pm)

Labs: the heart of 6.009

- Use python 3.6 (`python3` and `pip3`)
- Issued Friday mornings (lab0 available now -- is optional/ungraded; lab1 will issue this Friday)
- Some concept questions due before Lecture each Monday
- Rest of lab due the following Friday @ 4pm
- Checkoff meetings during office hours (necessary to get *any* credit for lab); due by Wednesday @ 10pm after lab due date
- Lateness multiplier applies to each part; 25% penalty accrues each weekday linearly from 4pm to 5pm
- Three automatic four-day extensions (for sports, music, interviews, projects, or any other reason), applied to all parts of lab

Grading

- 9 labs -- 40% of grade (labs 8 and 9 are "double" two-week labs)
- 3 quizzes -- 60% of grade. Resubmit incomplete problems for partial credit. Quizzes planned for Wednesday evenings 7:30-9:30pm. No final exam.
- See course website for details on grading policies

Python Notional Machine

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. Specifically, in the live tutorial we'll draw and manipulate environment diagrams for the following evaluations.

Variables and data types

Integers

```
In [ ]: a = 307
        b = a
        print('a:', a, '\nb:', b)
```

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

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

Lists

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

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

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

Unlike integers, lists are mutable:

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

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

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

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

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

Tuples

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

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

Unlike a list, we can't change the top most structure of a tuple. What happens if we try the following?

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

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

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

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.

Strings

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

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

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

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

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.

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

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

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

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

What happens when using the '+' operator used on lists?

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

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

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

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

Other Data Types (Next Week): sets, dictionaries

Next week, we'll refresh ourselves on sets and dictionaries. We'll find those useful in later labs, but don't require them for Lab 1 (though they might help there too!).

Functions and scoping

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

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

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.

Optional arguments and default values

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

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

In [ ]:
```
c = foo(10)
print('a:', a, '\nb:', b, '\nc:', c)
```

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

In [ ]:
```
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)
```

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

In [ ]:
```
c = foo(10)
print('a:', a, '\nb:', b, '\nc:', c)
```

So quite different... all kinds of aliasing going on. Perhaps surprisingly, the default value to an optional argument is only evaluated once, at function *definition* time. 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!

Closures

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

In [ ]:
```
add1 = add_n(1)
print(add1(7))

add2 = add_n(2)
print(add2(3))

print(add1(77))
print(add_n(8)(9))
```

Classes

A look ahead. We'll refresh ourselves on classes later, but you might want to try these on your own now.

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

print(Simple.x)
```

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

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

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

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

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

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

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

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

In [ ]:
```
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())
```

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 [ ]:
```
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))
```

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

Readings -- if you want/need more refreshers

Check out the **6.145 Notes** (https://6009.cat-soop.org/6.145_notes/):

- Assignment and aliasing: What is an environment? What is a frame? Discussed in Section 11 of Chapter 1 (https://6009.cat-soop.org/6.145_notes/chapter1).
- Functions: What happens when one is defined? What happens when one is called? Examples in Chapter 3 (https://6009.cat-soop.org/6.145_notes/chapter3).
- Closures in Chapter 4 (https://6009.cat-soop.org/6.145_notes/chapter4).
- Classes: What is a class? What is an instance? What is self? What is __init__? Discussed in Chapter 5 (https://6009.cat-soop.org/6.145_notes/chapter5)
- Inheritance in Chapter 6 (https://6009.cat-soop.org/6.145_notes/chapter6)