Term Test 2 Info

- Nov. 13, 11:10am -12:00pm BA1230
- Policy
  - Closed book with one yellow cheating sheet
  - Contents from Oct. 9 to next Monday
    (exclude formal proof and lambda calculus)
  - 20% policy
  - Pen for remark
- Extra office hours
  - Thursday afternoon 2:00-5:00pm
  - SF3209

Outline

- Last week
  - More functional programming
  - Misc
  - Modules
  - Beginning of classes

- Today
  - Classes (II)

Classes (II)
A Simple Class

- A class C whose parent is object:
  class C(object):
    x = 7 # class variable, not instance variable!
    y = 10 # class variable
    def m(self):
      self.y = 20 # assigns to *instance* variable

  a = C() # creates an instance of C
  b = C()
  a.x # 7
  a.y # 10 (class variable)
  a.m()
  a.y # 20 (instance variable)
  b.y # 10
  C.y = 75
  b.y # 75
  a.y # 20 (instance variable hides class variable)

A Simple Class

- Observations about class C:
  - class C(object):
    This says that object is the parent class.
    - def m(self):
      The self parameter must be the first parameter of every instance method. When m is called (a.m() in our example), self is automatically set to the instance on which m() is being called. This is like the word this in C++/Java. Note that technically you can use any word you like in place of self, but it's conventional to use self.
      - self.y = 20
      Unlike C++/Java, you have to use self in order to access instance variables.

Any assignments to self.<variable> (or, outside of a class, to <objectname>.<variable>) are always to instance variables and not to class variables. As usual in Python, instance variables are created whenever they are first assigned to.

A Simple Class

- More observations about class C:
  - a = C()
    Creating an object looks like a "call" to the class. Unlike Java, there is no new operator to create objects.
    - a.y # 10 (class variable)
      a.m()
      a.y # 20 (instance variable)
    When resolving a reference (as opposed to an assignment) to <objectname>.<variable> (or, within a class, to self.<variable>), Python first looks for an instance variable named <variable>, and if none is found, looks for a class variable (and then looks at class variables for parent classes, and so on up the inheritance tree).

A Class with a Constructor

- A class D with a constructor that takes one parameter, x:
  class D(object):
    def __init__(self, x):
      self.x = x
    def m(self):
      return self.x + 2

  a = D(5)
  a.m() # 7
  a.x # 5

  - The constructor must be called __init__(). If you don't provide a constructor, one is inherited from object (this doesn't do much).
  - Note that names that begin and end with two underscores are special methods or values.
Inheritance

- Parent and Child classes:
class Parent(object):
    x = 5
    def n(self):
        return self.x + 2
class Child(Parent):
    x = 8
    def m(self):
        return self.x + 4 + self.n()
        # self.n() is equivalent to Parent.n(self)
p = Parent()
c = Child()
p.n() # 7
c.m() # 22 (and not 19)
c.n() # 10 (and not 7)

Instance Method and Variables

- To call an instance method m() from within a class, we have to call self.m().
- To call an instance method m() of a parent class P from within a child class that has overridden m(), we call P.m(self).
- All instance methods and variables are public by default.
  – Later we’ll see that we can get around this, to a certain extent.
- There is only one instance variable of a given name for each object. So if a parent class P and a child class C both assign to self.x, they are assigning to the same variable.

Static and Class Methods

class TestStaticMethod(object):
    @staticmethod
    def foo():
        print 'calling static method foo()'

class TestClassMethod(object):
    @classmethod
    def foo(cls):
        print 'calling class method foo()'
        print 'foo() is part of class: %s', cls.__name__

>>> tsm = TestStaticMethod()
>>> TestStaticMethod.foo()
calling static method foo()

>>> tcm = TestClassMethod()
>>> TestClassMethod.foo()
calling class method foo() is part of class: TestClassMethod

>>> tcm.foo()
calling class method foo() is part of class: TestClassMethod

Inheritance

- Methods and variables are inherited, and can be overridden.
- Parent and Child classes, where Child overrides a method:
class Parent(object):
    x = 5
    def m(self):
        return self.x + 2
class Child(Parent):
    x = 8
    def m(self):
        return self.x + 4 + Parent.m(self)
p = Parent()
c = Child()
p.m() # 7
c.m() # 22 (and not 19)
**Polymorphism**

- Polymorphism is the property of "having many shapes". In object-oriented programming, it means taking on the attributes of the actual object instead of the apparent object.

```python
class P(object):
    def talk(self):
        return 'Hi: ' + self.msg()
    def msg(self):
        return 'Parent'
class C(P):
    def msg(self):
        return 'Child'
c = C()
c.talk() # 'Hi: Child'
d = P()
d.talk() # 'Hi: Parent'
```

**The Object Construction Process**

- Every class has a static method called `__new__`.
  - `o = C()` calls `C.__new__(C)`, which returns a new instance of C.
- If additional arguments are given to C(), these are also passed on to `__new__`.
  - Next, `C.__init__()` is called to initialize the new instance.
    - Again, any additional arguments are passed on to `__init__`.
    - If C doesn't have an `__init__()` method, the parent class' `__init__()` is called.
  - If C has a parent class P, and C has an `__init__()` method, then P's `__init__()` method is not called unless C's `__init__()` calls it explicitly:
    ```python
class C(P):
    def __init__(self):
        P.__init__(self)
        # additional stuff specific to C would go here.
```

**Polymorphism**

- Observations about c.talk():
  - The call `self.msg()` in P's `talk()` gets the `msg()` in C, because the actual object is a C.
    - Well, the actual object is also a P, since every C is a P, but the "lowest" version of `msg()` is the one that gets used.
    - On the other hand, when we call d.talk(), the actual object is an instance of P and not of C.

**Multiple Inheritance**

- Like C++ but unlike Java, Python allows multiple inheritance.
  - This means that a class can have multiple parent classes.
```python
class A(object): ...
class B(object): ...
class C(A, B): ...
```

- Issues to consider:
  - Suppose A and B each define a method `m()`, and C does not define such a method. Which `m()` gets called in the following situation?
    ```python
c = C()
c.m()
```
  - Things get even more interesting with diamond-shaped inheritance. In the current example, object is an ancestor of C two different ways (through A and through B).
  - How do we make sure that each ancestor class' constructor gets called exactly once?
Multiple Inheritance

- An example:
  - Suppose we want an object that can store a bunch of data items and draw a picture on the screen.
  - Suppose we have a List class that can store a bunch of data items.
  - Suppose we have a Window class that can draw a picture on the screen.
  - Then we can define an LW class that has List and Window as parents.

\[
\begin{align*}
\text{object} & \quad \text{List} \quad \text{Window} \\
& \quad \text{LW}
\end{align*}
\]

Multiple Inheritance

- To make things more interesting, suppose List and Window are both children of Store.

\[
\begin{array}{c}
\text{Object} \\
\downarrow & \\
\text{Store} & \quad \text{List} \quad \text{Window} \\
\downarrow & \quad \downarrow \\
\text{LW} & \quad \text{Window}
\end{array}
\]

- Suppose Store, List and Window define __getitem__() and m(), but LW does not define its own __getitem__() or m(). Which __getitem__() does it inherit? Which m() does it inherit?

- Answer: Python defines a method resolution order.
  - When looking for a method, it checks classes in the order specified by the method resolution order.

Multiple Inheritance

- Another issue: suppose Store, List, and Window each define an __init__() method. When writing LW's __init__() method, how can we make sure each of its ancestor's __init__() methods is called?

  - One solution: Each class' __init__() should call the __init__() of each of its parents.

  - Problem: If we did this, Store's __init__() will get called twice when we're constructing an LW object (once as a result of calling List's __init__() and once as a result of calling Window's __init__()).

  - Better solution: Call the ancestor's __init__() methods in the order specified by the method resolution order.

Multiple Inheritance

- The function super() can be used to determine what comes next in an object's method resolution order.
  - super(C, o) is the same object as o but looks for methods starting after class C in o's method resolution order.
  - For example, if object o is an instance of LW, then super(List, o).__getitem__() calls Window's __getitem__() method since Window follows List in LW.__mro__.

  - To make sure each ancestor's __init__() gets called exactly once, add the line super(C, self).__init__() to the __init__() method of each class C.

  - Note that the C inside the super() call should match the name of the class within which this call is being made.

  - For now, we're glossing over the issue of passing arguments to __init__().
Multiple Inheritance

- Example:
  ```python
class Store(object):
    def __init__(self):
        super(Store, self).__init__()
        # other stuff goes here

class List(Store):
    def __init__(self):
        super(List, self).__init__()

class Window(Store):
    def __init__(self):
        super(Window, self).__init__()

class LW(List, Window):
    def __init__(self):
        super(LW, self).__init__()
```

- This ensures that when an LW instance is constructed, the __init__() methods are called in the order LW, List, Window, Store.

Exercises

- Write classes A, B, and C, where C’s parent is B, and B’s parent is A. A’s parent is object. Experiment with inheritance and polymorphism.

- Continuing on with our Fibonacci number example:
  - Recall that the first two numbers in the Fibonacci sequence are both 1. But it makes sense to consider a sequence defined the same way where the first two numbers are something else. (e.g. 2, 7, 9, 16, 25, 41,..., is such a sequence where we’ve chosen 2 and 7 as the first two numbers.)
  - So write a class NewFibonacci whose constructor takes two numbers; the class uses these two numbers as the first two numbers in the sequence.
  - The class should have a method calculate(n) that returns the n-th number in the sequence.
  - Also add a method next(). The first call to next() returns the first number in the sequence, the second call returns the second number, and so on. You’ll obviously need instance variables to save state between calls.
  - Finally, add a method writeToFile(n, filename), that writes the first n numbers in the sequence to the file named filename, one number per line.