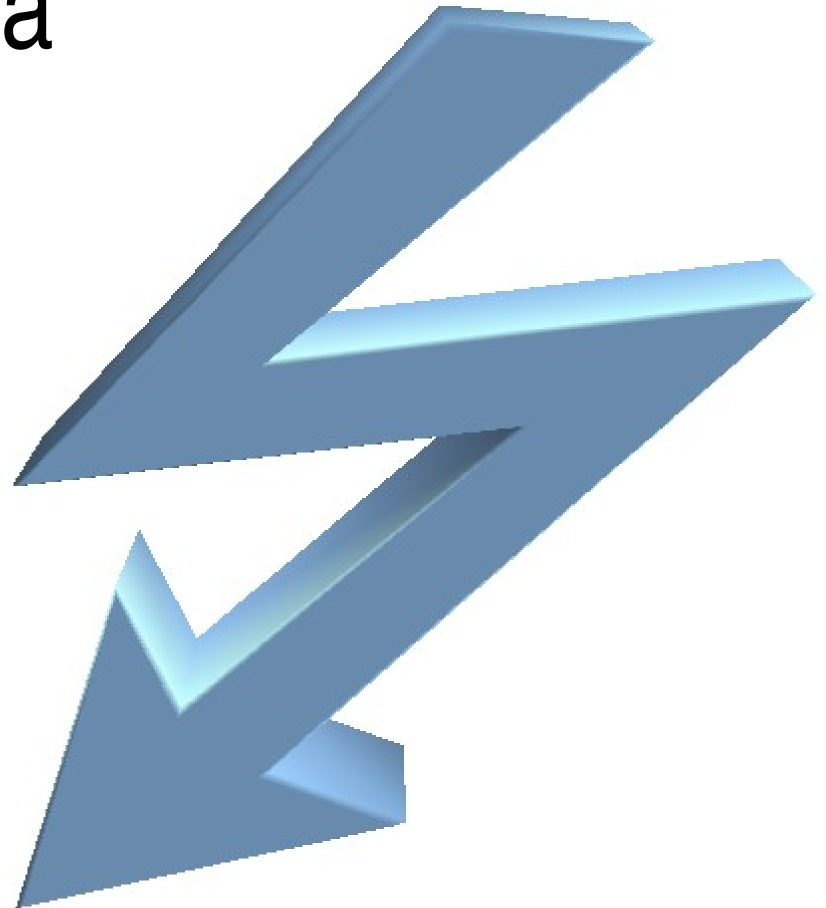
EECS 12: Lecture 7
Classes and Methods

Mark E. Phair
mphair@gmail.com
UC Irvine EECS

July 24th, 2006

Agenda

- Optional\ Keyword arguments
- Methods
- Special methods
- Polymorphism
- Python factoid of the day



Where'd you get the class, then?
We found it!
Found it? In Mercia? Classes are tropical,
and Mercia is a temperate zone.

Optional/Keyword Arguments

```
>>> def say(text = 'Moo!', times=1):
        print text*times
>>> say('hello')
hello
>>> say(times=2)
Moo!Moo!
>>> say()
Moo!
```

Optional *Objects*: A Cautionary Tale

```
>>> def atLeastFive(lst=[]):
        lst.append(5)
        return lst
>>> atLeastFive()
[5]
>>> atLeastFive()
[5, 5]
```

Optional *Objects*: A Cautionary Tale

```
>>> def atLeastFive(lst=[]):
        lst.append(5)
        return lst
>>> atLeastFive()
[5]
>>> atLeastFive()
[5, 5]
```

Let's Explore

- Create a function `dance` that takes two arguments, `step` and `speed`, where `speed` has a default value of 1
- `dance` should print out the value of `step, speed` number of times

Classes can have Methods

A *method* is a function that is part of a class

```
>>> class Cow:
        def moo(self):
            print 'Moo!'
>>> betsy = Cow()
>>> betsy.moo()
Moo!
```

`self`: the object *itself*

```
>>> class Animal:
        def setSound(self, sound):
            self.sound = sound
>>> betsy = Animal()
>>> betsy.setSound('Moo!')
>>> print betsy.sound
Moo!
```

`self`: the object *itself* (continued)

```
>>> class Animal:
        def setSound(self, sound):
            self.sound = sound
        def makeSound(self):
            print self.sound
>>> betsy = Animal()
>>> betsy.setSound('Moo!')
>>> betsy.makeSound()
Moo!
```

__init__: initialize

```
>>> class Animal:
        def __init__(self, sound):
            self.sound = sound
>>> betsy = Animal('Moo!')
>>> print betsy.sound
Moo!
```

`__init__` is one of many special methods

- underscore underscore init underscore underscore
- These special methods exist even if we do not define them, but we define them to get special behavior
- In the case of `__init__`, we want special initialization behavior
- Writing a method when there is already one there is called *overloading*

Equality

```
class Interval:
    def __init__(self, start, end):
        self.start = start
        self.end = end
>>> a, b = Interval(0,1), Interval(0,1)
>>> a == b
False
```

Overloading __eq__

```
class Interval:
    def __init__(self, start, end):
        self.start = start
        self.end = end
    def __eq__(self, other):
        return self.start == other.start \
            and self.end == other.end
>>> a, b = Interval(0,1), Interval(0,1)
>>> a == b
True
```

Overloading __add__

```
class Interval:
    def __init__(self, start, end):
        self.start,self.end = start,end
    def __add__(self, other):
        return Interval(\
            min(self.start,other.start), \
            max(self.end, other.end))
>>> a, b = Interval(0,1), Interval(1,2)
>>> a + b
<__main__.Interval instance at 0xb7d6ccec>
```

Overloading `__str__`

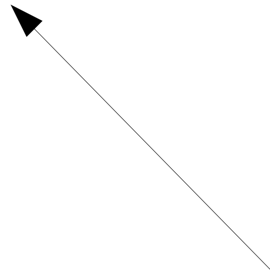
```
class Interval:
   def __init__(self, start, end):...
   def __add__(self, other):...
   def __str__(self):
      return 'Interval('+ \
        str(self.start) + ', ' + \
        str(self.end) + ')'
>>> Interval(0,1) + Interval(1,2)
Interval(0,2)
```

Polymorphism: Many shapes

```
>>> Interval(0,1) + Interval(1,2)
Interval(0,2)
>>> Interval('a','b') + Interval('b','e')
Interval(a,e)
```

__repr__ versus __str__

```
>>> Interval(0,1) + Interval(1,2)
Interval(0,2)
>>> Interval('a','b') + Interval('b','e')
Interval(a,e)
```



__str__: No quotes!

Overloading `__repr__`

```
class Interval:
    def __init__(self, start, end):...
    def __add__(self, other):...
    def __repr__(self):
        return 'Interval('+ \
          repr(self.start) + ', ' + \
          repr(self.end) + ')'
>>> Interval('a','b') + Interval('b','e')
Interval('a','e')
```

Overloading `__mul__`

```
class Interval:
   def __init__(self, start, end):...
   def __mul__(self, rhs):
      return Interval(self.start * rhs,\
                      self.end * rhs)
>>> Interval(1,3) * 3
Interval(3, 9)
```

Overloading `__rmul__`, `__radd__`

```
class Interval:
   def __init__(self, start, end):...
   def __rmul__(self, lhs):
      return Interval(lhs * self.start,\
                      lhs * self.end)
>>> 5 * Interval(1,3)
Interval(5, 15)
```

Overloading __cmp__

```
class Interval:
   def __init__(self, start, end):...
   def __cmp__(self, rhs):
      if self.end == rhs.end:
         if self.start == rhs.start:
            return 0
         else:
            return cmp(self.start, rhs.start)
      else:
         return cmp(self.end, rhs.end)
```

Overloading __cmp__: shorter version

```
class Interval:
    def __init__(self, start, end):...
    def __cmp__(self, rhs):
        if self.end == rhs.end:
            return cmp(self.start, rhs.start)
        else:
            return cmp(self.end, rhs.end)
>>> Interval(5,6) > Interval(4,5)
True
```

Python Factoid of the day: Other Assignment operators

- Addition

```
a = a + 1
a += 1
```

- Multiplication

```
c = c * 5
c *= 5
```

- Subtraction

```
b = b - 1
b -= 1
```

- Division

```
d = d / 5
d /= 5
```