Object Oriented Programming (II)

Classes have a bunch of special methods

the mirror of `__init__` is `__del__`

(it is the tear down during clean up)

```
>>>class Bear:
    def __init__(self,name):
        self.name = name
        print " made a bear called %s" % (name)
    def __del__(self):
        print "Bang! %s is no longer." % self.name
>>> y = Bear("Yogi") ; c = Bear("Cal")
 made a bear called Yogi
 made a bear called Cal
>>> del y; del c
Bang! Yogi is no longer.
Bang! Cal is no longer.
>>> y = Bear("Yogi") ; y = Bear("Cal")  ## note that I'm assigning y twice here
 made a bear called Yogi
 made a bear called Cal
Bang! Yogi is no longer.
>>> f = Bear("Fuzzy")
>>> exit()
Bang! Fuzzy is no longer.
BootCamp>
```

note: neither `__init__` or `__del__` are allowed to return anything

slightly less trivial example

file: bear.py

```
import datetime
class Bear:
    logfile_name = "bear.log"
    bear_num     = 0
    def __init__(self,name):
        self.name = name
        print " made a bear called %s" % (name)
        self.logf  = open(Bear.logfile_name,"a")
        Bear.bear_num += 1
        self.my_num = Bear.bear_num
        self.logf.write("[%s] created bear #%i named %s\n" % \
                        (datetime.datetime.now(),Bear.bear_num,self.name))
        self.logf.flush()

    def growl(self,nbeep=5):
        print "\a"*nbeep

    def __del__(self):
        print "Bang! %s is no longer." % self.name
        self.logf.write("[%s] deleted bear #%i named %s\n" % \
                        (datetime.datetime.now(),self.my_num,self.name))
        self.logf.flush()
        # decrement the number of bears in the population
        Bear.bear_num -= 1
        # dont really need to close because Python will do the garbage collection
        #  for us. but it cannot hurt to be graceful here.
        self.logf.close()
```

slightly less trivial example

```
>>> a = Bear("Yogi")
 made a bear called Yogi
>>> b = Bear("Fuzzy")
 made a bear called Fuzzy
>>> Bear.bear_num
2
>>> del a; del b
Bang! Yogi is no longer.
Bang! Fuzzy is no longer.
>>> Bear.bear_num
0
BootCamp> more bear.log
[2012-01-13 19:49:22.966413] created bear #1 named Yogi
[2012-01-13 19:49:30.820434] created bear #2 named Fuzzy
[2012-01-13 19:49:35.863398] deleted bear #1 named Yogi
[2012-01-13 19:49:35.864665] deleted bear #2 named Fuzzy
```

Classes have a bunch of special methods

`__str__` is a method that defines how a Class should represent itself as a string

it takes only `self` as an arg, must *return* a string

```
    def __str__(self):
        return " name = %s bear number = %i (population %i)" % \
               (self.name, self.my_num,Bear.bear_num)
```

```
>>> b = Bear("Fuzzy")
 made a bear called Fuzzy
>>> print b
 name = Fuzzy bear number = 1 (population 1)
>>> a = Bear("Yogi")
 made a bear called Yogi
>>> print b
 name = Fuzzy bear number = 1 (population 2)
```

this is the kind of formatting that `datetime()` is doing in it's own `__str__`

```
    def __init__(self,name):
       # other init stuff here
       self.created = datetime.datetime.now()

    def __str__(self):
        age = datetime.datetime.now() - self.created
         return " name = %s bear (age %s) number = %i (population %i)" % \
                (self.name, age, self.my_num,Bear.bear_num)
```

```
>>> a = Bear("Yogi")
 made a bear called Yogi
>>> print a
 name = Yogi bear (age 0:00:11.318241) number = 1 (population 0)
>>> print a
 name = Yogi bear (age 0:00:14.453264) number = 1 (population 0)
>>> print a
 name = Yogi bear (age 0:00:16.901385) number = 1 (population 0)
>>> print a
 name = Yogi bear (age 0:00:18.428962) number = 1 (population 0)
```

Emulating Numeric operations

you can define a whole bunch of ways that instances behave upon numerical operation

(e.g., __add__ is what gets called when you type *instance_1 + instance_2*)

__add__(*self, other*)

__sub__(*self, other*)

__mul__(*self, other*)

__div__(*self, other*)

__mod__(*self, other*)

__divmod__(*self, other*)

__pow__(*self, other*[, *modulo*

__lshift__(*self, other*)

__rshift__(*self, other*)

__and__(*self, other*)

__xor__(*self, other*)

....

file: bear1.py

```
class Bear:
    """
    class to show off addition (and multiplication)
    """
    bear_num = 0
    def __init__(self,name):
        self.name = name
        print " made a bear called %s" % (name)
        Bear.bear_num += 1
        self.my_num = Bear.bear_num

    def __add__(self,other):
        ## spawn a little tike
        cub = Bear("progeny_of_%s_and_%s" % (self.name,other.name))
        cub.parents = (self,other)
        return cub

    def __mul__(self,other):
        ## multiply (as in "go forth and multiply") is really the
        ## same as adding
        self.__add__(other)
```

```
>>> y = Bear("Yogi") ; c = Bear("Fuzzy")
 made a bear called Yogi
 made a bear called Fuzzy
>>> our_kid = y + c
 made a bear called progeny_of_Yogi_and_Fuzzy
>>> our_kid.
our_kid.__add__     our_kid.__doc__     our_kid.__module__  our_kid.bear_num    our_kid.name
our_kid.__class__   our_kid.__init__    our_kid.__mul__     our_kid.my_num
our_kid.parents
>>> our_kid.parents
(<__main__.Bear instance at 0x10308a560>,
 <__main__.Bear instance at 0x10308a488>)
>>> our_kid.parents[0].name
'Yogi'
>>> our_kid.parents[1].name
'Fuzzy'
>>> our_kid1 = y * c
 made a bear called progeny_of_Yogi_and_Fuzzy
```

Other Useful Specials

- **__dict__ :** Dictionary containing the class's namespace.
- **__doc__ :** Class documentation string, or None if undefined.
- **__name__:** Class name.
- **__module__:** Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__ :** A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

```
>>> print Bear.__doc__
    simple class to show off usage of special methods like __del__ and __str__
>>> print Bear.__name__
Bear
>>> print Bear.__module__
__main__
>>> print Bear.__bases__
()
>>> print Bear.__dict__
{'__module__': '__main__', '__del__': <function __del__ at 0x1030507d0>, '__str__':
<function __str__ at 0x103050848>, 'growl': <function growl at 0x1030508c0>,
'logfile_name': 'bear.log', 'bear_num': 0, '__doc__': '\n    simple class to show
off usage of special methods like __del__ and __str__\n    ', '__init__': <function
__init__ at 0x103050758>}
```

"Hiding" class data attributes

```
>>>class JustCounter:
    __secretCount = 0
      def count(self):
        self.__secretCount += 1
        print self.__secretCount

>>> counter = JustCounter()
>>> counter.count() ; counter.count()
1
2
>>> print counter.__secretCount
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print counter.__secretCount
AttributeError: JustCounter instance has no attribute '__secretCount'
>>>
```

double underscore attributes are exposed as

object._className__attrName

above: `counter._JustCounter__secretCount`

no attribute is ever precisely private

A note on referencing...

```
>>> a = Bear("Yogi")
>>> a
<__main__.Bear instance at 0x101619e60>
>>> b = a
>>> b  # reference to the same mem loc
<__main__.Bear instance at 0x101619e60>
>>> a.name = "Fuzzy"
>>> b.name
'Fuzzy'
>>> Bear.pop
1
>>> import copy
>>> c = copy.copy(a)
>>> c  # new memory location
<__main__.Bear instance at 0x101634710>
>>> Bear.pop
1
>>> c.name
'Fuzzy'
>>> c.name = "Smelly"
>>> a.name
'Fuzzy'
```

```
>>> a.list = [1,2,3]
>>> c.list
------------------------------------------------
---------
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
AttributeError: Bear instance has no attribute
'list'

>>> d = copy.copy(a)
>>> d.list
[1, 2, 3]
>>> d.name
'Fuzzy'
>>> d.name = "Yogi"
>>> a.name
'Fuzzy'
>>> a.list[0] = -1
>>> d.list
[-1, 2, 3]
>>> e = copy.deepcopy(a)
>>> a.list[0] = "a"
>>> e.list
[-1, 2, 3]
```

deepcopy: copies all attributes pointed to internally

Subclassing & Inheritance

```
class classname(baseclass):
```

For example,

```
class Flower(Plant):
```

Here we say that
"the class *Flower* is a subclass of the base class *Plant*."
Plant may itself be a subclass of *LivingThing*

attributes of the baseclass are **inherited** by the subclass

```
class Plant:
    num_known = 0
    def __init__(self,common_name,latin_name=None):
        self.latin_name = latin_name
        self.common_name = common_name
        Plant.num_known += 1

class Flower(Plant):
    has_pedals = True
```

```
>>> p = Plant("poison ivy")
>>> e = Flower("poppy")
>>> e.has_pedals
True
>>> Plant.num_known
2
>>> Flower.__bases__[0].__name__
"Plant"
```

instantiation of a Flower reuses the `__init__` from the Plant class. It also sets `has_pedals = True`

```
class Plant:
    num_known = 0
    def __init__(self,common_name,latin_name=None):
        self.latin_name = latin_name
        self.common_name = common_name
        Plant.num_known += 1

    def __str__(self):
        return "I am a plant (%s)!" % self.common_name

class Flower(Plant):
    has_pedals = True
    def __str__(self):
        return "I am a flower (%s)!" % self.common_name
```

now the `__str__` method of Flower takes precedence over the `__str__` method of the parent class

```
>>> f = Flower("rose") ; print f
I am a flower (rose)!
>>> p = Plant("oak") ; print p
I am a plant (oak)!
```

```
class Flower(Plant):
    has_pedals = True

    def __init__(self,common_name,npedals=5,pedal_color="red",latin_name=None):
        ## call the __init__ of the
        Plant.__init__(self,common_name,latin_name=latin_name)
        self.npedals=5
        self.pedal_color = pedal_color
        
```

we can still use the parent class' `__init__`

```
>>> f = Flower("rose") ; print f
I am a flower (rose)!
>>> f.npedals
5
```

Multiple Inheritances

```
class Flower1(Plant,EdibleFood,SmellyStuff)
```

when executing a method the namespace of:

Flower1 is searched first

Plant second (and it's baseclasses...and their baseclasses)

EdibleFood second (and it's baseclasses...and their baseclasses)

SmellyStuff second (and it's baseclasses...and their baseclasses)

Errors (& Handling)

From before...

```
try:
   tmp = raw_input("Enter a number “ + \
                   and I'll square it: ")
   print float(tmp)**2
except:
   print "dude. I asked you for a number and " + \
         "%s is not a number." % tmp
finally:
   print "thanks for playing!"
```

There are many different kinds of exceptions that can be raised by an error and each of them can be handled differently...

```
BaseException
Exception
StandardError
ArithmeticError
LookupError
EnvironmentError
AssertionError
AttributeError
EOFError
GeneratorExit
IOError
ImportError
IndexError
KeyError
KeyboardInterrupt
MemoryError
NameError
OverflowError
ReferenceError
RuntimeError
StopIteration
SyntaxError
SystemError
SystemExit
TypeError
ValueError
VMSError
WindowsError
ZeroDivisionError
Warning
UserWarning
DeprecationWarning
PendingDeprecationWarning
SyntaxWarning
RuntimeWarning
FutureWarning
ImportWarning
```

```
>>> 3.1415/0
------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

we can handle the error by explicating catching it and doing something with it

```
>>> def this_fails():
        x = 3.1415/0
>>> try:
        this_fails()
    except ZeroDivisionError as detail:
        print 'Handling run-time error:', detail

Handling run-time error: integer division or modulo by zero
```

`import exceptions`

http://docs.python.org/library/exceptions.html

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", result
...     finally:
...         print "executing finally clause"
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Catch Multiple Error Types

file: catcherr.py

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as (errno, strerror):
    print "I/O error(%i): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

`exc_info(...)`

`exc_info()` -> (type, value, traceback)

Return information about the most recent exception caught by an except clause in the current stack frame or in an older stack frame.

raising errors

we can raise errors in our codes (which themselves might be caught upstream)

```
>>> a = "cat food"
>>> if a != "spam":
...     raise NameError("anything that isn't spam breaks my code")
...
------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython console>", line 2, in <module>
NameError: anything that isn't spam breaks my code
>>>
```

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
```

Errors are a family of classes (and subclasses)!

We can create our own exception classes by subclassing others (e.g., `Exception`)

```
>>> import datetime
>>>
>>> class MyError(Exception):
     def __init__(self,value=None):
             ## call the baseclass Exception __init__
         Exception.__init__(self)
         self.value = value
         print "exception with %s at time %s" % (self.value,datetime.datetime.now())
     def __str__(self):
             return "you said %s" % self.value

>>> raise MyError("darnit")
exception with darnit at time 2012-01-13 23:00:07.863117
---------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
MyError: you said darnit
```