Classes

Introduction: Classes

Classes allow us to create a *custom type of object* -- that is, an object with its own *behaviors* and its own ways of storing *data*.

Consider that each of the objects we've worked with previously has its own behavior, and stores data in its own way: dicts store pairs, sets store unique values, lists store sequential values, etc.

An object's *behaviors* can be seen in its methods, as well as how it responds to operations like subscript, operators, etc.

An object's *data* is simply the data contained in the object or that the object represents: a string's characters, a list's object sequence, etc.

Objectives for this Unit: Classes

- Understand what classes, objects and attributes are and why they are useful
- Create our own classes -- our own object types
- Set *attributes* in objects and read attributes from objects
- Define *methods* in classes that can be used by objects
- Define object *initializers* with **__init__()**
- Use *getter* and *setter* methods to enforce *encapsulation*
- Understand class *inheritance*
- Understand *polymorphism*

Class Example: the *date* and *timedelta* object types

First let's look at object types that demonstrate the convenience and range of behaviors of objects.

A *date* object can be set to any date and knows how to calculate dates into the future or past.
To change the date, we use a *timedelta* object, which can be set to an "interval" of days to be added to or subtracted from a date object.

```
from datetime import date, timedelta

dt = date(1926, 12, 30)         # create a new date object set to 12/30/1
td = timedelta(days=3)          # create a new timedelta object:  3 day i

dt = dt + timedelta(days=3)     # add the interval to the date object:  p

print dt                        # '1927-01-02' (3 days after the original


dt2 = date.today()              # as of this writing:  set to 2016-08-01
dt2 = dt2 + timedelta(days=1)   # add 1 day to today's date

print dt2                       # '2016-08-02'

print type(dt)                  # <type 'datetime.datetime'>
print type(td)                  # <type 'datetime.timedelta'>
```

Class Example: the proposed *server* object type

Now let's imagine a useful object -- this proposed class will allow you to interact with a server programmatically. Each server object represents a server that you can ping, restart, copy files to and from, etc.

```
import time
from sysadmin import Server


s1 = Server('blaikieserv')

if s1.ping():
    print '{} is alive '.format(s1.hostname)

s1.restart()                        # restarts the server

s1.copyfile_up('myfile.txt')        # copies a file to the server
s1.copyfile_down('yourfile.txt')    # copies a file from the server

print s1.uptime()                   # blaikieserv has been alive for 2 sec
```

A *class* block defines an object "factory" which produces *objects* (*instances*) of the class.

Method calls on the object refer to functions defined in the class.

```
class Greeting(object):
    """ greets the user """

    def greet(self):
        print 'hello, user!'


c = Greeting()

c.greet()                   # hello, user!

print type(c)               # <class '__main__.Greeting'>
```

Each class *object* or *instance* is of a type named after the class. In this way, *class* and *type* are almost synonymous.

Each object holds an *attribute dictionary*

Data is stored in each object through its attributes, which can be written and read just like dictionary keys and values.

```
class Something(object):
    """ just makes 'Something' objects """

obj1 = Something()
obj2 = Something()

obj1.var = 5             # set attribute 'var' to int 5
obj1.var2 = 'hello'      # set attribute 'var2' to str 'hello'

obj2.var = 1000          # set attribute 'var' to int 1000
obj2.var2 = [1, 2, 3, 4] # set attribute 'var2' to list [1, 2, 3, 4]


print obj1.var           # 5
print obj1.var2          # hello

print obj2.var           # 1000
print obj2.var2          # [1, 2, 3, 4]

obj2.var2.append(5)      # appending to the list stored to attribute var2

print obj2.var2          # [1, 2, 3, 4, 5]
```

In fact the attribute dictionary is a real dict, stored within a "magic" attribute of the object:

```
print obj1.__dict__      # {'var': 5, 'var2': 'hello'}

print obj2.__dict__      # {'var': 1000, 'var2': [1, 2, 3, 4, 5]}
```

The class also holds an attribute dictionary

Data can also be stored in a class through class attributes or through variables defined in the class.

```
class MyClass():
    """ The MyClass class holds some data """

    var = 10              # set a variable in the class (a class variable


MyClass.var2 = 'hello'    # set an attribute directly in the class object

print MyClass.var         # 10      (attribute was set as variable in cla
print MyClass.var2        # 'hello' (attribute was set as attribute in cl

print MyClass.__dict__    # {'var': 10,
                          #  '__module__': '__main__',
                          #  '__doc__': ' The MyClass class holds some da
                          #  'var2': 'hello'}
```

The additional __**module**__ and __**doc**__ attributes are automatically added -- __**module**__ indicates the active module (here, that the class is defined in the script being run); __**doc**__ is a special string reserved for documentation on the class).

object.attribute lookup tries to read from object, then from class

If an attribute can't be found in an object, it is searched for in the class.

```
class MyClass(object):
  classval = 10         # class attribute

a = MyClass()
b = MyClass()

b.classval = 99         # instance attribute of same name

print a.classval        # 10 - still class attribute
print b.classval        # 99 - instance attribute

del b.classval          # delete instance attribute

print b.classval        # 10 -- now back to class attribute
```

Method calls pass the object as first (implicit) argument, called *self*

Object methods or *instance methods* allow us to work with the object's data.

```
class Do(object):
    def printme(self):
        print self      # <__main__.Do object at 0x1006de910>

x = Do()

print x                 # <__main__.Do object at 0x1006de910>
x.printme()
```

Note that **x** and **self** have the same hex code. This indicates that they are the very same object.

Instance methods / object methods and object attributes: changing object "state"

Since instance methods pass the object, and we can store values in object attributes, we can combine these to have a method modify an object's values.

```
class Sum(object):
    def add(self, val):
        if not hasattr(self, 'x'):
            self.x = 0
        self.x = self.x + val

myobj = Sum()
myobj.add(5)
myobj.add(10)

print myobj.x      # 15
```

Objects are often modified using *getter* and *setter* methods

These methods are used to read and write object attributes in a controlled way.

```
class Counter(object):
    def setval(self, val):     # arguments are:  the instance, and the va
        if not isinstance(val, int):
            raise TypeError('arg must be a string')

        self.value = val        # set the value in the instance's attribu

    def getval(self):          # only one argument:  the instance
        return self.value       # return the instance attribute value

    def increment(self):
        self.value = self.value + 1

a = Counter()
b = Counter()

a.setval(10)       # although we pass one argument, the implied first arg

a.increment()
a.increment()

print a.getval()   # 12


b.setval('hello')  # TypeError
```

__init__() is *automagically* called when a new instance is created

The *initializer* of an object allows us to set the initial attribute values of the object.

```
class MyCounter(object):
  def __init__(self, initval):   # self is implied 1st argument (the inst
    try:
      initval = int(initval)     # test initval to be an int,
    except ValueError:           # set to 0 if incorrect
      initval = 0
    self.value = initval         # initval was passed to the constructor

  def increment_val(self):
    self.value = self.value + 1

  def get_val(self):
    return self.value

a = MyCounter(0)
b = MyCounter(100)

a.increment_val()
a.increment_val()
a.increment_val()

b.increment_val()
b.increment_val()

print a.get_val()    # 3
print b.get_val()    # 102
```

Classes can be organized into an an *inheritance tree*

When a class inherits from another class, attribute lookups can pass to the *parent* class when accessed from the *child*.

```
class Animal(object):
  def __init__(self, name):
    self.name = name
  def eat(self, food):
    print '%s eats %s' % (self.name, food)

class Dog(Animal):
  def fetch(self, thing):
    print '%s goes after the %s!' % (self.name, thing)

class Cat(Animal):
  def swatstring(self):
    print '%s shreds the string!' % (self.name)
  def eat(self, food):
    if food in ['cat food', 'fish', 'chicken']:
      print '%s eats the %s' % (self.name, food)
    else:
      print '%s:  snif - snif - snif - nah...' % self.name

d = Dog('Rover')
c = Cat('Atilla')

d.eat('wood')                 # Rover eats wood.
c.eat('dog food')             # Atilla:  snif - snif - snif - nah...
```

Conceptually similar methods can be unified through *polymorphism*

Same-named methods in two different classes can share a conceptual similarity.

```
class Animal(object):
  def __init__(self, name):
    self.name = name
  def eat(self, food):
    print '%s eats %s' % (self.name, food)

class Dog(Animal):
  def fetch(self, thing):
    print '%s goes after the %s!' % (self.name, thing)
  def speak(self):
    print '%s:  Bark!  Bark!' % (self.name)

class Cat(Animal):
  def swatstring(self):
    print '%s shreds the string!' % (self.name)
  def eat(self, food):
    if food in ['cat food', 'fish', 'chicken']:
      print '%s eats the %s' % (self.name, food)
    else:
      print '%s:  snif - snif - snif - nah...' % self.name
  def speak(self):
    print '%s:  Meow!' % (self.name)

for a in (Dog('Rover'), Dog('Fido'), Cat('Fluffy'), Cat('Precious'), Dog(
  a.speak()

                  # Rover:  Bark!  Bark!
                  # Fido:  Bark!  Bark!
                  # Fluffy:  Meow!
                  # Precious:  Meow!
                  # Rex:  Bark!  Bark!
                  # Kittypie:  Meow!
```