Object Oriented Programming (OOP)

1. Classes
    A. Defining Your Own
    B. Initializing
    C. Adding fields
    D. String representation
    E. Comparators
2. Static
    A. Members
    B. Methods
3. Public & Private?
4. Inheiritence

Class: Simplest Example

In [2]:
```
class SimpleClass(object):
    pass

c = SimpleClass()
print c
```

```
<__main__.SimpleClass object at 0x105868410>
```

Create an `__init__` constructor

In [6]:
```
class MyClass(object):
    def __init__(self, a, b=3):
        self.a = a
        self.b = b

# create an instance
c1 = MyClass(5, 10)
c2 = MyClass(1)  # use a default argument

# print attributes
# print "c1: a=%d, b=%d" % (c1.a, c1.b)
# print "c2: a=%d, b=%d" % (c2.a, c2.b)

c1.random = lambda x: x +1
print c1.random(2)

c1.a = 123
print c1.a
```

```
3
123
```

Make a List of Objects

In [8]:
```
# iterate through the in a list
instances = [c1, c2]

for inst in instances:
    if type(inst) == MyClass:  # test the type
        print inst
```

```
<__main__.MyClass object at 0x105868c50>
<__main__.MyClass object at 0x103fde590>
```

Adding Fields

In [10]:
```
c3 = MyClass(6, 8)
c3.special = "just for this object"  # add an attribute on fly

print c3.special
```

```
just for this object
```

String Representation

Sometimes we'd like to change how our objects appear when we print them directly. The default isn't very pretty:

```
<__main__.MyClass object at 0x10fd23a10>
```

It simply shows the module.ClassName with the address location of the object.

Let's instead define our own by overriding the base `object`'s `__str__` function, which literally controls how the object is converted into a string.

The `__str__` magic method

Just like `__init__`, `__str__` is a built-in Python method that all Python objects share. We can simply override this method:

In [9]:
```
class House(object):
    def __init__(self, npets, sqft):
        self.number_of_pets = npets
        self.square_footage = sqft

    def __str__(self):
        return "<House: %d pets, %d sqft>" % (
            self.number_of_pets, self.square_footage)

    def __repr__(self):
        return "<H: %d, %d>" % (
            self.number_of_pets, self.square_footage)

# create a house
house = House(npets=2, sqft=2000)
print house
print str(house)
print [house]
```

```
<House: 2 pets, 2000 sqft>
<House: 2 pets, 2000 sqft>
[<H: 2, 2000>]
```

`__repr__` ?

For when you're printing a list:

In [17]:
```
h1 = House(1, 1000)
h2 = House(2, 1500)
h3 = House(10, 1000)
houses = [h1, h2, h3]
print houses
```

```
[<H: 1, 1000>, <H: 2, 1500>, <H: 10, 1000>]
```

Comparing, Equality, & Sorting Objects

How can we compare two objects? How can we define custom sort functions?

Comparing Two Objects

Once again, we need to override some of `object`'s builtin methods:

In [18]:
```
class ComparableClass:

    def __lt__(self, other):
        pass   # < comparison

    def __le__(self, other):
        pass   # <= comparison

    def __eq__(self, other):
        pass   # == comparison

    def __ne__(self, other):
        pass   # != comparison

    def __gt__(self, other):
        pass   # > comparison

    def __ge__(self, other):
        pass   # >= comparison
```

Example: Comparing -> Sorting

In [19]:
```
class Racecar:
    def __init__(self, color, top_speed):
        self.color = color
        self.top_speed = top_speed

    def __lt__(self, other):
        return self.top_speed < other.top_speed # < comparison

    def __le__(self, other):
        return self.top_speed <= other.top_speed  # <= comparison

    def __eq__(self, other):
        return self.top_speed == other.top_speed  # == comparison

    def __ne__(self, other):
        return self.top_speed != other.top_speed # != comparison

    def __gt__(self, other):
        return self.top_speed > other.top_speed # > comparison

    def __ge__(self, other):
        return self.top_speed >= other.top_speed # >= comparison

    def __repr__(self):
        return "<RC: %d>" % self.top_speed


# create four racecars
r1 = Racecar("red", 100)
r2 = Racecar("blue", 200)
r3 = Racecar("red", 300)
r4 = Racecar("green", 100)

# try some comparisons
print r1 <= r4 < r2 < r3
print r1 == r4
print r2 != r3

# now try sorting them
cars = [r4, r2, r3, r1]
print "Before sorting:", cars
cars.sort()
print "After sorting:", cars
```

```
True
True
True
Before sorting: [<RC: 100>, <RC: 200>, <RC: 300>, <RC: 100>]
After sorting: [<RC: 100>, <RC: 100>, <RC: 200>, <RC: 300>]
```

Sorting Objects/Tuples (Quick & Dirty way)

Don't want to override all those function? Maybe just need to sort tuples?

In [20]:
```
elements = [
    (14, "apple", True),
    (10, "banana", False),
    (11, "banana", True),
    (6, "orange", True),
    (8, "kiwi", False),
]

# tuples default to the using the entries as keys in order, cascading
# uses all entires as keys
elements.sort()
print "Default sorting scheme:\n", elements

# use the second key instead only
elements.sort(key = lambda x: x[1])
print "Using the second key:\n", elements

# ony use a pair of keys
def custom_tuplekey_ordering(tup):
    return (tup[0], tup[2])  # just use first and last entries

elements.sort(key=custom_tuplekey_ordering)
print "Using the function for 1st and last keys:\n", elements
```

```
Default sorting scheme:
[(6, 'orange', True), (8, 'kiwi', False), (10, 'banana', False), (11,
 'banana', True), (14, 'apple', True)]
Using the second key:
[(14, 'apple', True), (10, 'banana', False), (11, 'banana', True), (8,
 'kiwi', False), (6, 'orange', True)]
Using the function for 1st and last keys:
[(6, 'orange', True), (8, 'kiwi', False), (10, 'banana', False), (11,
 'banana', True), (14, 'apple', True)]
```

Sorting Objects (Quick & Dirty Way)

In [27]:
```
class SpeedBoat(object):
    def __init__(self, c, s):
        self.color = c
        self.top_speed = s
    def __repr__(self):
        return "<Boat: %s, %d>" % (self.color, self.top_speed)

# make some car objects and add to a list, unordered
r1 = SpeedBoat("red", 100)
r2 = SpeedBoat("blue", 200)
r3 = SpeedBoat("green", 300)
r4 = SpeedBoat("red", 100)
boats = [r3, r4, r2, r1]
print "Unsorted:", boats

# use the quick and dirty sorting method
boats.sort(key = lambda c: c.top_speed)
print "Sorted:", boats
```

```
Unsorted: [<Boat: green, 300>, <Boat: red, 100>, <Boat: blue, 200>, <Bo
at: red, 100>]
Sorted: [<Boat: red, 100>, <Boat: red, 100>, <Boat: blue, 200>, <Boat:
 green, 300>]
```

Static Members & Methods

Sometimes we'd like functionality attached to the class definition rather than an individual instantiation.

Static Members

In [40]:
```
class SonyStereo(object):

    MAX_DECIBELS = 0
    MIN_DECIBELS = -64

    def __init__(self):
        self.decibels = MIN_DECIBELS

    def change_volume(self, decibels):
        assert db < MAX_DECIBELS and db > MIN_DECIBELS, "dB out of rang
e!"
        self.decibels = decibels

# print class variables
print SonyStereo.MAX_DECIBELS
print SonyStereo.MIN_DECIBELS
```

```
0
-64
```

Static Methods

In [42]:
```
class SomeClass(object):
    def __init__(self):
        pass

    @staticmethod  # this is a decorator, we'll cover this later!
    def return_five():
        return 5

print SomeClass.return_five()
```

```
5
```

Public & Private?

Python **doesn't strictly enforce** notions of public, private, or protected methods or any sort. It does however, have conventions:

Conventions:

- Private methods should start with an underscore _
- Built-in or methods descending from `object` start and end with a double underscore __
- *Magic* built-in object methods are of the form `__xxxx__`

Method Naming Conventions

In [47]:
```
def _private_module_method():
    pass

def public_module_method():
    pass

class Boat(object):
    def __init__(self):
        self._private_field = True
        self.public_field    = None

    def _private_instance_method(self):
        pass

    @staticmethod
    def _private_class_method(self):
        pass

    def public_method(self):
        pass
```

Inheiritence

It is one of the cornerstones of OOP software design, allowing for rich hierarchies of object typing and functionalities.

(Insert image of animal taxonomy here)

Example: Hierarchy of Typing with `isinstance()`

In [66]:
```
class Shape(object):
    pass

class Rectangle(Shape):
    pass

s = Shape()
r = Rectangle()

print "s is of type():", type(s)
print "r is of type():", type(r), "\n"

print "Rectangle is a Rectangle?", isinstance(r, Rectangle)
print "Rectangle is a Shape?", isinstance(r, Shape)
print "Shape is a Rectangle?", isinstance(s, Rectangle)
```

```
s is of type(): <class '__main__.Shape'>
r is of type(): <class '__main__.Rectangle'>

Rectangle is a Rectangle? True
Rectangle is a Shape? True
Shape is a Rectangle? False
```

Example: Overriding Methods

In [49]:
```
class Person(object):
    def __init__(self, health, power, **kwargs):
        self.health = health
        self.power = power

    def eat(self):
        return "Omnomnomnom!"

    def talk(self):
        return "Just a peasant."

    def __repr__(self):
        return "<Person>"

class Warrior(Person):

    def __init__(self, health, power, weapon):
        self.weapon = weapon
        Person.__init__(self, health, power)

    def talk(self):
        return "Smash, fight, arggghhh"

    def __repr__(self):
        return "<Warrior>"

class King(Warrior):

    def __init__(self, health, power, weapon, crown):
        self.crown = crown
        Warrior.__init__(self, health, power, weapon)

    def talk(self):
        return "Bow before me!"

    def __repr__(self):
        return "<King>"

# create some characters
person = Person(health=100, power=10)
warrior = Warrior(health=120, power=50, weapon='sword')
king = King(health=110, power=20, weapon='saber', crown='gold')

characters = [person, warrior, king]
for character in characters:
    print "Type: %s" % character
    print "Talking: %s" % character.talk()
    print "Eating: %s" % character.eat()
    print "Is this person a King?", isinstance(character, King)
    print "Is this person a Warrior?", isinstance(character, Warrior)
    print "Is this person a Person?", isinstance(character, Person)
    print
```

```
Type: <Person>
Talking: Just a peasant.
Eating: Omnomnomnom!
Is this person a King? False
Is this person a Warrior? False
Is this person a Person? True

Type: <Warrior>
Talking: Smash, fight, arggghhh
Eating: Omnomnomnom!
Is this person a King? False
Is this person a Warrior? True
Is this person a Person? True

Type: <King>
Talking: Bow before me!
Eating: Omnomnomnom!
Is this person a King? True
Is this person a Warrior? True
Is this person a Person? True
```

Here we see that:

- All Kings are Warriors and Persons
- The base `eat()` method is kept throughout classes
- The `talk()` methods are successfully overriden
- We can pass parameters through to children initializers and specify which arguments are used

and we have successfully learned how to use inheiritence in Python!

Lab: Calculator Class

Fill in the method definitions in the file `excercises/classes.py`.

Make sure you can pass tests with:

```
$ py.test tests/test_classes.py::ClassesExcercises
```

Wrap-Up

1. Classes
    A. Defining Your Own
    B. Initializing
    C. Adding fields
    D. String representation
    E. Comparators
2. Static
    A. Members
    B. Methods
3. Public & Private?
4. Inheiritence