Object Oriented Programming with Python

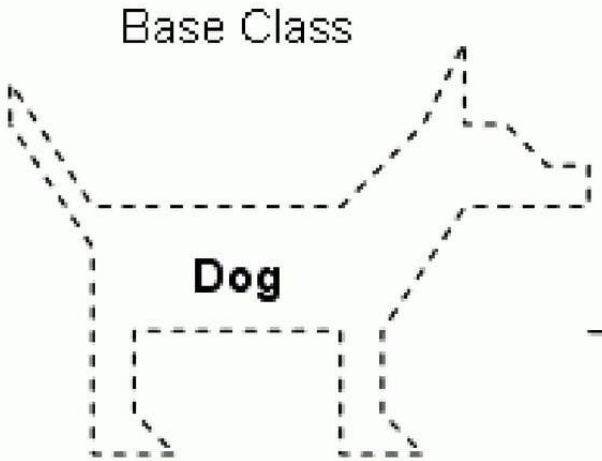
What are Objects?

- An **object** is a location in memory having a value and possibly referenced by an identifier.
- In Python, every piece of data you see or come into contact with is represented by an object.

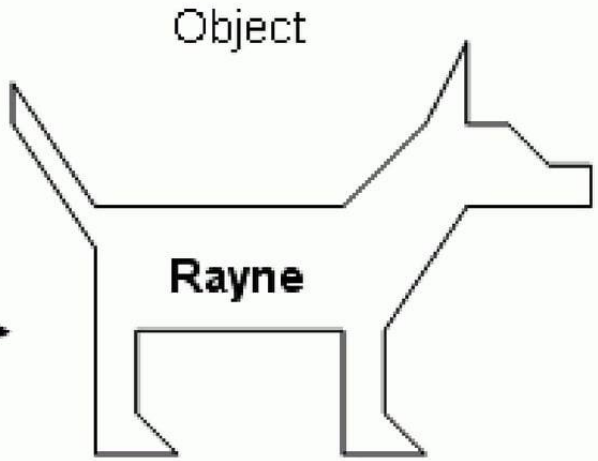
- Each of these objects has three components:
  I. Identity
  II. Type
  III. Value

```
>>>str = "This is a String"
>>>dir(str)
.....
>>>str.lower()
'this is a string'
>>>str.upper()
'THIS IS A STRING'
```

Defining a Class



Create Instance



Properties	**Methods**
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

Property values	**Methods**
Color: Gray, White, and Black	Sit
Eye Color: Blue and Brown	Lay Down
Height: 18 Inches	Shake
Length: 36 Inches	Come
Weight: 30 Pounds	

Defining a Class

- Python's class mechanism adds classes with a minimum of new syntax and semantics.
- It is a mixture of the class mechanisms found in C++ and Modula-3.
- As C++, Python class members are public and have Virtual Methods.

- A basic class consists only of the *class* keyword.
- Give a suitable name to class.
- Now You can create class members such as data members and member function.

The simplest form of class definition looks like this:

```
class className:
    <statement 1>
    <statement 2>
    .
    .
    .
    <statement N>
```

Defining a Class

- As we know how to create a Function.
- Create a function in class MyClass named func().
- Save this file with extension .py
- You can create object by invoking class name.
- Python doesn't have new keyword
- Python don't have new keyword because everything in python is an object.

```
class MyClass:
        """A simple Example
of class"""
        i=12
        def func(self):
                return "Hello World
```

```
>>>obj = MyClass(  )
>>> obj.func()
 'Hello World'
```

Defining a Class

```
1.  class Employee:
2.          'Common base class for all employees'
3.          empCount = 0

4.          def __init__(self, name, salary):
5.                  self.name = name
6.                  self.salary = salary
7.                  Employee.empCount += 1

8.          def displayCount(self):
9.                  print("Total Employee %d" % Employee.empCount)

10.         def displayEmployee(self):
11.                 print("Name : ", self.name, ", Salary: ", self.salary)
```

Defining a Class

- The first method *__init__()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- Other class methods declared as normal functions with the exception that the first argument to each method is self.
- Self: This is a Python convention. There's nothing magic about the word self.
- The first argument in __init__() and other function gets is used to refer to the instance object, and by convention, that argument is called self.

Creating instance objects

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000 )
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```

- To create instances of a class, you call the class using class name and pass in whatever arguments its *__init__* method accepts.
- During creating instance of class. Python adds the *self* argument to the list for you. You don't need to include it when you call the methods

```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

Output

```
Name : Zara ,Salary: 2000
Name : Manni ,Salary: 5000
Total Employee 2
```

Special Class Attributes in Python

- Except for self-defined class attributes in Python, class has some special attributes. They are provided by object module.

Attributes Name	Description
__dict__	Dict variable of class name space
__doc__	Document reference string of class
__name__	Class Name
__module__	Module Name consisting of class
__bases__	The tuple including all the superclasses

Form and Object for Class

- Class includes two members: form and object.
- The example in the following can reflect what is the difference between object and form for class.

```
class A:
    i = 123
    def __init__(self):
        self.i = 12345

print A.i
print A().i

>>>
123
12345
```

Invoke form: just invoke data or method in the class, so i=123

Invoke object: instantialize object Firstly, and then invoke data or Methods.
Here it experienced __init__(), i=12345

Class Scope

- Another important aspect of Python classes is scope.
- The scope of a variable is the context in which it's visible to the program.
- Variables that are available everywhere (**Global Variables**)
- Variables that are only available to members of a certain class (**Member variables**).
- Variables that are only available to particular instances of a class (**Instance variables**).

Destroying Objects (Garbage Collection):

- Python deletes unneeded objects (built-in types or class instances) automatically to free memory space.
- An object's reference count increases when it's assigned a new name or placed in a container (list, tuple or dictionary).
- The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope.
- You normally won't notice when the garbage collector destroys an orphaned instance and reclaims its space
- But a class can implement the special method *__del__()*, called a destructor
- This method might be used to clean up any non-memory resources used by an instance.

Destroying Objects (Garbage Collection):

```
a = 40 # Create object <40>
b = a # Increase ref. count of <40>
c = [b] # Increase ref. count of <40>

del a # Decrease ref. count of <40>
b = 100 # Decrease ref. count of <40>
c[0] = -1 # Decrease ref. count of <40>
```

Class Inheritance:

- Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.
- Like Java subclass can invoke Attributes and methods in superclass.

Syntax

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
    'Optional class documentation string'
    class_suite
```

- Python support multiple inheritance but we will only concern single parent inheritance.

```
class Parent: # define parent class
        parentAttr = 100
        def __init__(self):
                print "Calling parent constructor“
        def parentMethod(self):
                print 'Calling parent method‘
        def setAttr(self, attr):
                Parent.parentAttr = attr
        def getAttr(self):
                print "Parent attribute :" ,
Parent.parentAttr
class Child(Parent): # define child class
         def __init__(self):
                print "Calling child constructor"
         def childMethod(self):
                print 'Calling child method'

c = Child() # instance of child
c.childMethod() # child calls its method
c.parentMethod() # calls parent's method
c.setAttr(200) # again call parent's method
c.getAttr() # again call parent's method
```

Static Variable

Output

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Overriding Methods:

- You can always override your parent class methods.
- One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

```
class Parent: # define parent class
        def myMethod(self):
                print 'Calling parent method'
class Child(Parent): # define child class
        def myMethod(self):
                print 'Calling child method'

c = Child() # instance of child
c.myMethod() # child calls overridden method
```

Overloading Operators:

- In python we also overload operators as there we overload ‘+’ operator.

```
class Vector:
        def __init__(self, a, b):
                self.a = a
                self.b = b
        def __str__(self):
                return 'Vector (%d, %d)' % (self.a, self.b)
        def __add__(self,other):
                return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
Print(v1 + v2)
```

Method Overloading

- In python method overloading is not acceptable.
- This will be against the spirit of python to worry a lot about what types are passed into methods.
- Although you can set default value in python which will be a better way.
- Or you can do something with tuples or lists like this…

```
def print_names(names):
        """Takes a space-delimited string or an iterable"""
        try:
                for name in names.split(): # string case
                        print name
        except AttributeError:
                for name in names:
                        print name
```

Polymorphism:

- Polymorphism is an important definition in OOP. Absolutely, we can realize polymorphism in Python just like in JAVA. I call it “traditional polymorphism”
- In the next slide, there is an example of polymorphism in Python.
- But in Python,

Only traditional polymorphism exist?

No!

Polymorphism:

```
class Animal:
        def __init__(self, name): # Constructor of the class
                self.name = name
        def talk(self): # Abstract method, defined by convention only
                raise NotImplementedError("Subclass must implement
                                                abstract method")
class Cat(Animal):
        def talk(self):
                return 'Meow!'
class Dog(Animal):
        def talk(self):
                return 'Woof! Woof!'

animals = [Cat('Missy'), Cat('Mr. Mistoffelees'), Dog('Lassie')]
for animal in animals:
Print(animal.name + ': ' + animal.talk())
```

Everywhere is polymorphism in Python

- So, in Python, many operators have the property of polymorphism. Like the following example:

```
>>> 1+2
3
>>> 'key'+'board'
'keyboard'
>>> [1,2,3]+[4,5,6,7]
[1, 2, 3, 4, 5, 6, 7]
>>> (1,2,3)+(4,5,6)
(1, 2, 3, 4, 5, 6)
>>> {A:a, B:b}+{C:c, D:d}
```

- Looks stupid, but the key is that variables can support any objects which support 'add' operation. Not only integer but also string, list, tuple and dictionary can realize their relative 'add' operation.