Object Oriented Programming with Python

Slides courtesy:Chapter Eight (Part I)

Object Oriented Programming; Classes, constructors, attributes, and methods

Objectives

- Create classes to define objects
- Write methods and create attributes for objects
- Instantiate objects from classes
- Restrict access to an object’s attributes
- Work with both new-style and old-style classes

Python Is Object-Oriented

- **Object-oriented programming (OOP):** Methodology that defines problems in terms of objects that send messages to each other
  - In a game, a **Missile** object could send a **Ship** object a message to **Explode**

Understanding Object-Oriented Basics

- OOP allows representation of real-life objects as software objects (e.g., a dictionary as an object)
- **Object: A single software unit that combines attributes and methods**
- **Attribute:** A "characteristic" of an object; like a variable associated with a kind of object
- **Method:** A "behavior" of an object; like a function associated with a kind of object
- **Class:** Code that defines the **attributes** and **methods** of a kind of object **(**A class is a collection of variables and functions working with these variables)

Fundamental Concepts of OOP

- Information hiding
- Data Abstraction
- Data Encapsulation
- Modularity
- Polymorphism
- Inheritance

OO Paradigm - Review

- Three Characteristics of OO Languages
  - Inheritance
    - It isn't necessary to build every class from scratch – attributes can be derived from other classes
  - Polymorphism
    - The meaning of a method attribute depends on the object's class/subclass
  - Encapsulation
    - Object behavior and object data are combined in a single entity. Object interface defines interaction with the object; no need to know/understand the implementation.

Constructors

```
def __init__(self):
    self.value = 0
```

or

```
def __init__(self, thing):
    self.value = thing
```

Usage: **`x = MyClass( )`** sets **`x.value to 0`**

`y = MyClass(5)` sets **`y.value to 5`**

(default constructor and parameterized constructor)

Class with Constructor

```
>>> class Complex:
        def __init__(self, realp,
  imagp):
      self.r = realp
      self.i = imagp

>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Attributes (Data Members)

- Attributes are defined by an assignment statement, just as variables are defined (as opposed to being declared).

```
def set(self, value):
   self.value = value
```

- Can be defined in classes or instances of classes.
- Attributes attached to classes belong to all subclasses and instance objects, but attributes attached to instances only belong to that instance.

Python Class Objects

- There is no `new` operator; to instantiate an object for a class, use functional notation:

```
x = MyClass()
y = MyClass()
```

- Each time a class is called, a new instance object is created. If the class has a constructor, you can use it here.
- Instance objects have data attributes and methods as defined in the class.

Python Class Data Members

- Variables in Python are created when they are assigned to.
  - New data members (attributes) can be created the same way; e.g.,
    `x.counter = 1`
    creates a new data attribute for the object **`x`** – but not for **`y`**.
- Beware – data attributes override method attributes with the same name!

Information Hiding

- There is no foolproof way to enforce information hiding in Python, so there is no way to define a true abstract data type
- **Everything is public by default; it is possible to partially circumvent the methods for defining private attributes.**

Inheritance

```
class DerivedClassName(BaseClassName):
  <statement-1>
  . . .
  <statement-N>
```

- If a requested attribute is not found in the derived class, it is searched for in the base class. This rule is applied recursively if the base class itself is derived from some other class.

Multiple Inheritance

- Python supports multiple inheritance:

```
class DerivedClassName(B1, B2, B3):
    <statement-1>
    . . .
    <statement-N>
```

Multiple Inheritance

- Resolving conflicts: (Given the expression *object.attribute,* find definition of *attribute)*
  - Depth-first, left-to-right search of the base classes.
  - “Depth-first”: start at a leaf node in the inheritance hierarchy (the object); search the object first, its class next, superclasses in L-R order as listed in the definition.

Polymorphism

- In computer science, **polymorphism** is a programming language feature that allows values of different data types to be handled using a uniform interface.

  http://en.wikipedia.org/wiki/Type_polymorphism; 3/29/2010

Types of Polymorphism

- Polymorphism with virtual functions is sometimes called subtype polymorphism, inclusion polymorphism or pure polymorphism
  - This is the intended meaning when we say OO programming implements polymorphism.
- Parametric polymorphism comes from templates or generic functions
- Overloading is a kind of ad-hoc polymorphism

Polymorphism

- Polymorphism is a product of inheritance:
  - A method's definition is determined by the class of the object that invokes it.
- By re-defining a method in a subclass (giving it a new implementation), it is possible for a derived class to override the parent class definition.

Polymorphism

- Virtual functions are those that can be overridden
  - C++: defined with key word `virtual`
  - Java & Python: every method is virtual by default
- Difference between abstract and virtual functions:
  - Abstract methods aren’t defined

Object Orientation in Python

- In Python, everything is an object – integers, strings, dictionaries, …
- Class objects are instantiated from user-defined classes, other objects are from language defined types.

Python Classes

- Can be defined anywhere in the program
- **All methods and instance variables are public, by default**
  - The language provides “limited support” for private identifiers .

Example

```
class MyClass:
  def set(self, value):
       self.value = value
 def  display(self):
        print(self.value)
```

`MyClass` has two methods: `set` and `display`; and one attribute: `value`.

The class definition is terminated by a blank line.

Example

The first parameter in each method refers to the object itself. Self is the traditional name of the parameter, but it can be anything.

```
def set(self, value):
    self.value = value
```

When the method is called, the self parameter is omitted

Example

Declare and assign value to a class variable:

```
>>> y = MyClass()
>>> y.set(4)
>>> y.display()
4
>>> y.value
4
```

Constructor - Example

Constructors are defined with the `__init__` method.

Instead of

```
  def set(self, value):
```

use

```
  def __init__(self, value):
```

Constructors

```
def __init__(self):
    self.value = 0
```

or

```
def __init__(self, thing):
    self.value = thing
```

Usage: **`x = MyClass( )`** sets **`x.value to 0`**

`y = MyClass(5)` sets **`y.value to 5`**

(default constructor and parameterized constructor)

Class with Constructor

```
>>> class Complex:
        def __init__(self, realp,
  imagp):
      self.r = realp
      self.i = imagp

>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Attributes (Data Members)

- Attributes are defined by an assignment statement, just as variables are defined (as opposed to being declared).

```
def set(self, value):
   self.value = value
```

- Can be defined in classes or instances of classes.
- Attributes attached to classes belong to all subclasses and instance objects, but attributes attached to instances only belong to that instance.

Attributes (Data Members)

Define class:

Class name, begin with capital letter, by convention

object: class based on (Python built-in type)

Define a method

Like defining a function

Must **have a special first parameter**, `self,` which provides way for a method to refer to object itself

Python Class Objects

- There is no `new` operator; to instantiate an object for a class, use functional notation:

```
x = MyClass()
y = MyClass()
```

- Each time a class is called, a new instance object is created. If the class has a constructor, you can use it here.
- Instance objects have data attributes and methods as defined in the class.

Python Class Data Members

- Variables in Python are created when they are assigned to.
  - New data members (attributes) can be created the same way; e.g.,
    `x.counter = 1`
    creates a new data attribute for the object **`x`** – but not for **`y`**.
- Beware – data attributes override method attributes with the same name!

Information Hiding

- There is no foolproof way to enforce information hiding in Python, so there is no way to define a true abstract data type
- **Everything is public by default; it is possible to partially circumvent the methods for defining private attributes.**

Information Hiding

```
#define the Vehicle class
class Vehicle:
    name = ""
    kind = "car"
    color = ""
    value = 100.00
    def description(self):
        desc_str = "%s is a %s %s worth $%.2f." % (self.name, self.color,
self.kind, self.value)
        return desc_str


# your code goes here
car1 = Vehicle()
car1.name = "Fer"
car1.color = "red"
car1.kind = "convertible"
car1.value = 60000.00
```

Information Hiding

```
car2 = Vehicle() #creating an object
car2.name = "Jump"
car2.color = "blue"
car2.kind = "van"
car2.value = 10000.00
```

```
# test code
print(car1.description())
print(car2.description())
```

Information Hiding

```
class Cup:
    def __init__(self, color):
        self._color = color    # protected variable
        self.__content = None  # private variable


def fill(self, tea):
        self.__content = tea
def empty(self):
        self.__content = None


redCup = Cup("red")
redCup.color = "red"
redCup.content = "tea"
redCup.empty()
redCup.fill("coffee")
print redCup.color
```

Information Hiding

```
class Person(object):
def __init__(self, name=None, job=None, quote=None, hash={}) :
      self.name = name
      self.job = job
      self.quote = quote
      self.hash = hash
#create an empty list
personList = []

#create two class instances
personList.append(Person("Payne N. Diaz", "coach", "Without exception, there is no rule!"))
personList.append(Person("Mia Serts", "bicyclist", "If the world didn't suck, we'd all fall off!"))

# assign a single entry to the dictionary of each class instance

personList[0].hash['person0'] = 0
personList[1].hash['person1'] = 1

# print dictionary of first class instance
print personList[0].hash{'person0': 0, 'person1': 1}
# print dictionary of second class instance
print personList[1].hash{'person0': 0, 'person1': 1}
```

Two More Special Methods

```
class Puppy(object):
    def __init__(self):
        self.name = []
        self.color = []

    def __setitem__(self, name, color):
        self.name.append(name)
        self.color.append(color)

    def __getitem__(self, name):
        if name in self.name:
            return self.color[self.name.index(name)]
        else:
            return None

dog = Puppy()
dog['Max'] = 'brown'
dog['Ruby'] = 'yellow'
print "Max is", dog['Max']
```

Summary

- Object-oriented Programming (OOP) is a methodology of programming where new types of objects are defined
- An object is a single software unit that combines attributes and methods
- An attribute is a “characteristic” of an object; it’s a variable associated with an object (“instance variable”)
- A method is a “behavior” of an object; it’s a function associated with an object
- A class defines the attributes and methods of a kind of object