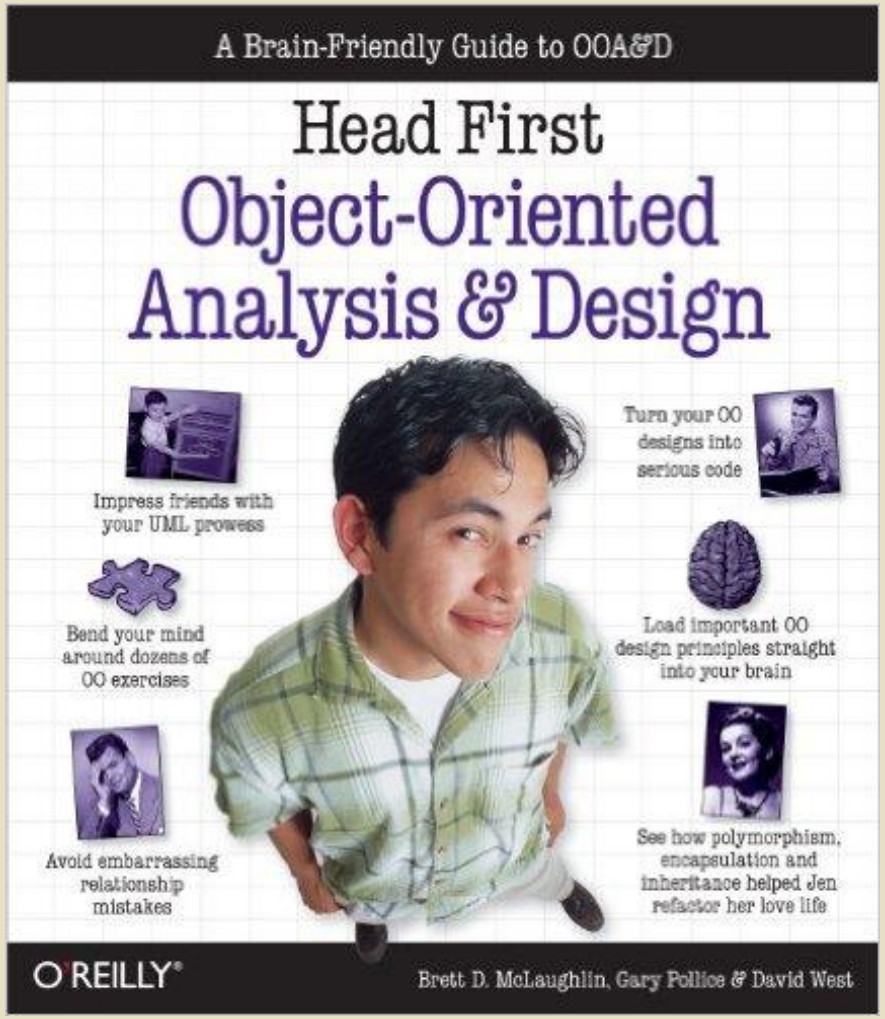
OBJECT ORIENTED PROGRAMMING WITH PYTHON

CSCI 5458: Object Oriented Analysis and Design - Presentation

Gaurav Bishnoi

Assumption:

Audience have the knowledge of Object Oriented Analysis and Design.



Summary:

- Introduction of Python's methods to realize:
  - Class Definition
  - Inheritance
  - Multiple Inheritance
  - Accessibility
  - Polymorphism
  - Encapsulation
- Comparison of Python's OOP methods with other programming languages.

Advantages of Object-Oriented Programing

- OOP provides a clear modular structure for programs which makes it good for defining abstract datatypes where implementation details are hidden and the unit has a clearly defined interface.
- OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
- OOP provides a good framework for code libraries where supplied software components can be easily adapted and modified by the programmer. This is particularly useful for developing graphical user interfaces.

Python?

- Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.
- It is high-level built in data structures.
- It has a large and comprehensive standard library.
- It has simple and easy to learn syntax that emphasizes readability.
- It features a fully dynamic type system and automatic memory management.

Advantages of Python

- Simple and easy to study.
- Free and Open Source.
- Portability and Expansibility.
- Large and Comprehensive Standard Libraries.
- Canonical Code.
- High Level Programing Language.

Python Class Example

```
class Person:                                          #Class Definition

    def __init__(self,name):                           #Constructor Function
        self.name = name

    def Intro(self):                                   #Method
        print 'Hello, I am ', self.name

    def __del__(self):                                 #Destructor Function
        print '%s says goodbye!' % self.name

gaurav = Person('Gaurav Bishnoi')                      #Object Definition
```

Class and Object Syntax

- Class Definition Syntax:

  class subclass(superclass):

  {Attributes and Methods}

- Object Instantiation Syntax:

  object = class()

- Syntax for Method and Attribute Invoke:

  object.attribute

  Object.method()

Constructor: ___init__()

- The __init__ method runs as soon as an object of a class is instantiated. Its aim is to initialize the object.
- Class declaration with constructor:

```
class Person:                                   #Class Definition
    def __init__(self,name):                    #Constructor Function
        self.name = name
```

- Object instantiation:

```
gaurav = Person('Gaurav Bishnoi')                #Object Definition
```

- Now object ‘gaurav’ has name(gaurav.name) = ‘Gaurav Bishnoi’

self == this

- ‘self’ is equal to ‘this’ in C++ or Java.
- It is used in a class by its functions to access the data of class.

```
class Person:                                   #Class Definition
    def __init__(self,name):                    #Constructor Function
        self.name = name
```

Inheritance:

- Subclass can invoke attributes and methods in superclass.
- Class 'Man' inherits from class 'Person':

```
class Person:                          #Superclass Definition
    def Walk(self):                    #Method declared in superclass
        print 'I am walking'

class Man(Person):                     #Subclass Definition
    def Run(self):                     #Method declrared in subclass
        print 'I am running'

hulk = Man()                           #Instantiating object of class 'Man'
hulk.Walk()                             #Invoking superclass method from subclass object
hulk.Run ()                             #Invoking subclass method from subclass object
```

```
sh-4.3$ python main.py
I am walking
I am running
```

Multiple Inheritance:

- Python supports multiple inheritance but in limited form.
- A subclass inheriting from multiple superclasses:

```
class subclass(superclass1, superclass2, superclass3):
{Attributes and Methods}
```

- If an attribute is not found in subclass, first, it is searched in superclass1, then recursively in the classes of superclass1, and only if it is not found there, it is searched in superclass2, and so on.

Multiple Inheritance: Example

- Class ‘Runner’ inherits from ‘Person’ and ‘Athlete’.

```
class Person:                          #Superclass1 Definition
    def A(self):                       #Method same as in Athlete class
        print 'I am a Person'

class Athlete():                       #Superclass2 Definition
    def A(self):                       #Same method as in Person class
        print 'I am an athletic Person'
    def B(self):                       #Different method from Person class
        print 'I am an Athlete'

class Runner(Person, Athlete):         #Subclass Definition
    def C(self):                       #Subclass method
        print 'I am a runner'

hulk = Runner()                        #Instantiating object of class 'Runner'
hulk.A()                               #Method from class Person will run
hulk.B()                               #Method from class Athlete will run
hulk.C()
```

- Method ‘A’ is in both superclasses. But since class ‘Person’ in is left in declaration of subclass ‘Runner’, method ‘A’ of class ‘Person’ will be invoked from object.

Encapsulation: Accessibility

- There are no special keywords in Python for encapsulation like:
  - private
  - protected
  - public
- By default all attributes are public.
- Procedure in Python to define ‘private’:
  - Add “__” in front of the variable and function name can hide them when accessing them from out of class.

private: Example

```
class course:                          #class definition
    def __init__(self):
        self.show = 'CSCI'             #'public' attribute
        self.__hide = '5448'           #'private' attribute

    def showDetails:                   #Function to access both attributes
        print self.show
        print self.hide

OOAD = course()                        #Object instantiation for course class
```

- We can access ‘show’ attribute by ‘OOAD.show’ but ‘OOAD.__hide’ will fail
- To access ‘hide’ attribute, we have to call ‘OOAD.showDetails’.

Reality of Accessibility in Python

- The Accessibility Method doesn't hide the 'private' attribute completely from outside of containing class.
- The Accessibility Method changes private name like __variable or __function() to _ClassName__variable or _ClassName__function(). So we can't access them because of wrong names.
- We even can use the special syntax to access the private data or methods. The syntax is actually its changed name.

Accessing ‘private’: Example

```
class course:                          #class definition
    def __init__(self):
        self.show = 'CSCI'             #'public' attribute
        self.__hide = '5448'           #'private' attribute

    def showDetails:                   #Function to access both attributes
        print self.show
        print self.hide

OOAD = course()                        #Object instantiation for course class
```

- Here we can access the ‘private’ ‘hide’ attribute using ‘OOAD._course__hide’.
- So, the changed name of ‘hide’ is ‘_course__hide’.

Accessibility: Python vs Java

- Java is a static and strong type definition language while Python is a dynamic and weak type definition language.
- Java has special keywords for encapsulation such as ‘private’, ‘public’ and ‘protected’ while Python has all the attributes and methods as public in default.
- Python uses accessibility methods to realize encapsulation virtually.

Polymorphism

- We can realize polymorphism in Python just like in Java or many other Object-Oriented Languages.
- Because Python is a dynamic programming language, it means Python is strongly typed as the interpreter keeps track of all variables types. It reflects the polymorphism character in Python.
- In Python many operators have the property of polymorphism, such as '+' operator. Integers, strings, constants, variables, etc realize their relative 'add' operation using '+' operator.

Polymorphism: Example 1

```
In [7]: 5 + 5
Out[7]: 10

In [8]: a = 5

In [9]: b = 5

In [10]: a + b
Out[10]: 10

In [11]: 'Hello' + ' ' + 'World'
Out[11]: 'Hello World'
```

Traditional Polymorphism

- Traditionally we consider polymorphism in terms of classes and objects that we define or instantiate ourselves.
- Polymorphic behavior allows you to specify common methods in an "abstract" level, and implement them in particular instances
- Python supports this traditional polymorphism.

Traditional Polymorphism: Example

```
class Person(object):
    def pay_bill():
        raise NotImplementedError

class Millionare(Person):
    def pay_bill():
        print "Here you go! Keep the change!"

class GradStudent(Person):
    def pay_bill():
        print "Can I owe you ten bucks or do the dishes?"
```

- Here, ‘Millionare’ and ‘Gradstudent’ are both ‘Person’ but when it comes to paying a bill(‘pay_bill()’), their specific "pay the bill" action is different.

Conclusion

- Python has some special advantages as an Object-Oriented Programing language but also has some disadvantages.
- Python can support operator overloading and multiple inheritance that some OOP languages such as Java don't.
- The advantages for Python to use design pattern is that it supports dynamic type binding. In other words, an object is rarely only one instance of a class, it can be dynamically changed at runtime.
- For Encapsulation, it doesn't have the keywords of 'private', 'public' and 'protected'. But it manages to somewhat provide it by some Accessibility Methods.