ООР

Все мы равны, все мы объекты

Что это такое

ООП - методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования

Строгое определение ООП

1) объектно-ориентированное программирование использует в качестве основных логических конструктивных элементов объекты, а не алгоритмы

2) каждый объект является экземпляром определенного класса

3) классы образуют иерархии

Программа считается объектно-ориентированной, только если выполнены все три указанных требования.

В частности, программирование, не использующее наследование, называется не объектно-ориентированным, а программированием с помощью абстрактных типов данных

Понятия ООП

- Абстракция
- Инкапсуляция
- Наследование
- Полиморфизм

- Класс
- Объект

Создание класса

```
class ClassName:
    'Optional class documentation string'
    class_suite
```

Пример класса

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

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

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

Создание объектов

"This would create first object of Employee class"

emp1 = Employee("Zara", 2000)

"This would create second object of Employee class"

emp2 = Employee("Manni", 5000)

Доступ к полям класса и объекта

```
emp1.displayEmployee()

emp2.displayEmployee()

print "Total Employee %d" % Employee.empCount
```

Модификации полей объекта

emp1.age = 7

emp1.age = 8

del emp1.age

Манипуляции с полями объектов

hasattr(emp1, 'age')

getattr(emp1, 'age')

setattr(emp1, 'age', 8)

delattr(empl, 'age')

Встроенные поля о которых стоит знать

__dict__ - словарь с неймспейсом класса
__doc__ - документация класса
__name__ - имя класса
__module__ - модуль класса (название)
__main__ - для запуска класса
__bases__ - тупл с базовыми классами

Наследование классов

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

Пример наследования

```
class Parent:
   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):
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()
c.childMethod()
c.parentMethod()
c.setAttr(200)
c.getAttr()
```

Еще немножко наследования

class A:

.....

class B:

.....

class C(A, B):

.....

Функции для проверки

- issubclass(sub, sup)
- isinstance(obj, Class)

Переопределение методов

```
class Parent:
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent):
   def myMethod(self):
      print 'Calling child method'

c = Child()
c.myMethod()
```

То что надо переопределять

__init__ (self [,args...]) - конструктор, например obj = className(args)

__del__(self) - деструктор, например del obj

__str__(self) - строчное представление, например str(obj)

__cmp__ (self, x) - компаратор для объектов, например cmp(obj, x)

Переопределение операторов

```
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
```

Инкапсуляция полей

```
class JustCounter:
   __secretCount = 0

   def count(self):
      self.__secretCount += 1
      print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

Пример полиморфизма

```
class Animal:
    def __init__(self, name):
        self.name = name
    def talk(self):
        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()
```

Композиция, агрегация и ассоциация

- Ассоциация - у кажлого свой цикл, нет владельцев (Учитель - Студент), живут отдельно друг от друга
- Агрегация - свои циклы, но есть владельцы и определенные элементы пренадлежат только определенным владельцам (Учитель - Кафедра), учитель пренадлежит кафедре, но если удалить кафедру - он остается
- Композиция - сильный вид связи (иногда называется “смертельная связь”), пример Дом - Комнаты, нету жизненных циклов для элементов и при удалении владельца - удаляются и элементы

Принципы SOLID

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Другие принципы

- DRY
- KISS
- YAGNI
- ...