Learning Python

Session 15 - OOP (Object Oriented Programming)

Know BIG DATA

www.KnowBigData.com

Object Oriented Programming

- **Encapsulation**
    - Hiding internal details
- **Inheritance**
    - Objects can inherit the behavior from others
- **Polymorphism**
    - Showing multiple kind of behaviors
    - Same function but different arguments for e.g.

Bank Account Example

```
balance = 0

def deposit(amount):
    global balance
    balance += amount
    return balance

def withdraw(amount):
    global balance
    balance -= amount
    return balance
```

```
deposit(10);
withdraw(10);
```

Problems?

Bank Account Example

```
balance = 0

def deposit(amount):
    global balance
    balance += amount
    return balance

def withdraw(amount):
    global balance
    balance -= amount
    return balance
```

```
deposit(10);
withdraw(10);
```

Problems?

You can not use it for multiple accounts.

Bank Account Example

Another Attempt

```
def make_account():
    return {'balance': 0}

def deposit(account, amount):
    account['balance'] += amount
    return account['balance']

def withdraw(account, amount):
    account['balance'] -= amount
    return account['balance']
```

```
>>> a = make_account()
>>> b = make_account()
>>> deposit(a, 100)
100
>>> deposit(b, 50)
50
>>> withdraw(b, 10)
40
>>> withdraw(a, 10)
90
```

Problems?

Bank Account Example

Another Attempt

```
def make_account():
    return {'balance': 0}

def deposit(account, amount):
    account['balance'] += amount
    return account['balance']

def withdraw(account, amount):
    account['balance'] -= amount
    return account['balance']
```

Problems?
Too repetitive.

Using Class

```
class BankAccount:
    def __init__(self):
        self.balance = 0

    def withdraw(self, amount):
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance
```

```
>>> a = BankAccount()
>>> b = BankAccount()
>>> a.deposit(100)
100
>>> b.deposit(50)
50
>>> b.withdraw(10)
40
>>> a.withdraw(10)
90
```

Class

```
class BankAccount:
    def __init__(self):
        self.balance = 0

    def withdraw(self, amount):
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance
```

1. Is a keyword
2. It is an encapsulation
3. Self defines the scope of variables
4. Class can have functions

Class - Functions

```
class BankAccount:
    def __init__(self):
        self.balance = 0

    def withdraw(self, amount):
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance
```

- *a = BankAccount()*
- *a.deposit(100)*

- *While defining we are made available “self”*
- *But we don’t pass*
- *So, every function of class should have 1 argument*

Class - Constructors

```
class BankAccount:
    def __init__(self):
        self.balance = 0

    def withdraw(self, amount):
        self.balance -= amount
        return self.balance
```

- Special Function with the name __init__
- init gets called when we create object
  - a = BankAccount()
- Very useful in setting up initial variables.

Inheritance

```
class MinimumBalanceAccount(BankAccount):
    def __init__(self, minimum_balance):
        BankAccount.__init__(self)
        self.minimum_balance = minimum_balance

    def withdraw(self, amount):
        if self.balance - amount < self.minimum_balance:
            print 'Sorry, min. balance must be maintained.'
        else:
            BankAccount.withdraw(self, amount)
```

- Extend functionality
- More specialized

```
class A:
    def f(self):
        return self.g()
    def g(self):
        return 'A'

class B(A):
    def g(self):
        return 'B'

a = A()
b = B()

print a.f(), b.f()
print a.g(), b.g()
```

Output?

```
class A:
    def f(self):
        return self.g()
    def g(self):
        return 'A'

class B(A):
    def g(self):
        return 'B'

a = A()
b = B()

print a.f(), b.f()
print a.g(), b.g()
```

Output?

```
A, B
A, B
```

Example: Drawing Shapes - Canvas

```
class Canvas:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.data = [[' '] * width for i in range(height)]

    def setpixel(self, row, col):
        self.data[row][col] = '*'

    def getpixel(self, row, col):
        return self.data[row][col]

    def display(self):
        print "\n".join(["".join(row) for row in self.data])
```

- A Canvas is a big string with spaces
- We can set a particular index as *
- Get a particular index
- And then render

Example: Drawing Shapes - Shape

```
class Shape:
    def paint(self, canvas): pass

class Square(Rectangle):
    def __init__(self, x, y, size):
        Rectangle.__init__(self, x, y, size, size)

class CompoundShape(Shape):
    def __init__(self, shapes):
        self.shapes = shapes
    def paint(self, canvas):
        for s in self.shapes:
            s.paint(canvas)
```

- pass is a null operation -- when it is executed, nothing happens.
- Any Shape class that is supposed to have paint method.
- Square is a shape
- CompountShape is shape that has other shapes

Example: Drawing Shapes - Shape

```
class Rectangle(Shape):
    def __init__(self, x, y, w, h):
        self.x = x
        self.y = y
        self.w = w
        self.h = h
    def hline(self, x, y, w):
        pass
    def vline(self, x, y, h):
        pass
    def paint(self, canvas):
        hline(self.x, self.y, self.w)
        hline(self.x, self.y + self.h, self.w)
        vline(self.x, self.y, self.h)
        vline(self.x + self.w, self.y, self.h)
```

- Rectangle is a shape

Example: Drawing

```
c = Canvas()
r = Rectangle(20,20,0,0)
r.paint(c)
```

- Create a canvas
- Create a rectangle
- Paint a rectangle on canvas

OOP - Operator Overloading
Special Class Methods

```
>>> a, b = 1, 2
>>> a + b
3
>>> a.__add__(b)
3
```

__add__	+
__sub__	-
__mul__	*
__div__	/

OOP - Operator Overloading
Special Class Methods

```
class MyClass:
   def __init__(self):
      self.x = 0;
   def __add__(self,b):
      self.x += b;
```

```
m = MyClass()
m + 1
m.x
m + 1
m.x
```

Exceptions

- You can handle errors using try..except
- You can catch an exception using
    - try:
    -     ...
    - except IOError, e:
    -     print e
    -     ...
- You can raise exception using raise.
    - raise Exception("error message")
- All exception are classes that inherit Exception()

```
class MyClass:
    def __init__(self):
        self.x = 0;
    def __add__(self,b):
        if self.x > 5:
            raise Exception(" > 5");
        self.x += b;
```

```
>>> m = MyClass()
>>> m + 6
>>> m + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in __add__
Exception:  > 5
```

What will be the output?

```
try:
    print "a"
    raise Exception("doom")
except:
    print "b"
else:
    print "c"
finally:
    print "d"
```

?

What will be the output?

```
try:
    print "a"
    raise Exception("doom")
except:
    print "b"
else:
    print "c"
finally:
    print "d"
```

```
a
b
d
```

Summary

- Why Classes
- Functions
- Constructors
- Operator Overloading
- Raising Exceptions