Scripting & Computer Environments

Object-Oriented Programming (OOP): The Basics

IIIT-H

Programming Paradigms

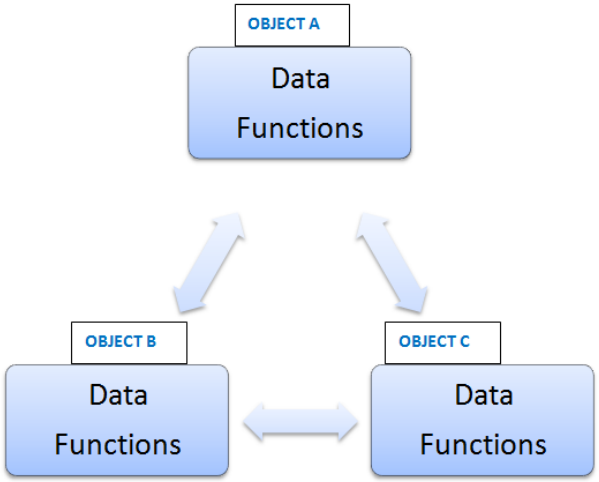
1. Procedural/Imperative Programming
2. Logical Programming
3. Functional Programming
4. Object-Oriented Programming

OOP Introduction

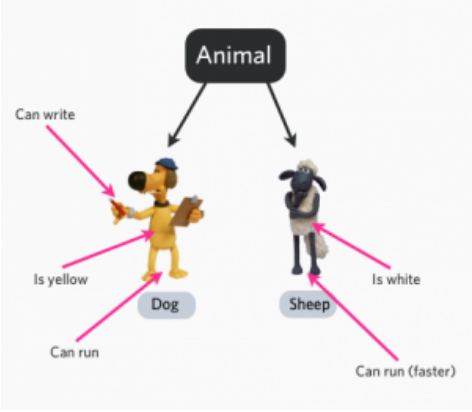
Object-Oriented Programming (OOP)

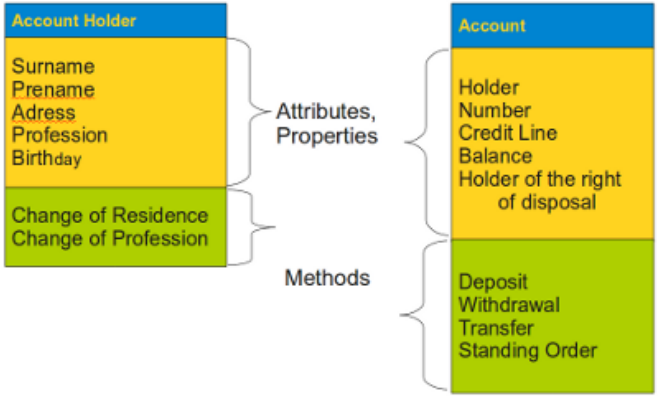
- Models a program as a set of objects with state (attributes) and behavior (methods).
- Attributes (aka *properties*) define the object.
- Methods define *actions* that can be performed on the object.
- Some languages: `Simula`, `Python`, `C++`, `Java`, `Smalltalk`, etc
- In Python, use of OOP is entirely optional, but a powerful feature.

OOP Introduction (2)



$$Object = Attributes + Methods$$

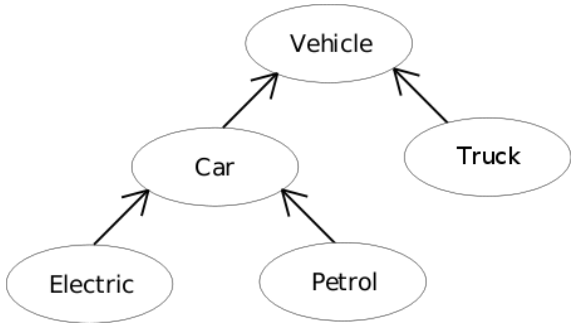




Core OOP Principles

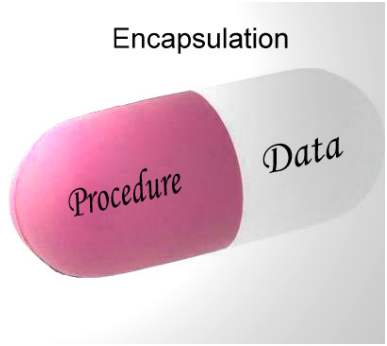
- Inheritance

  Code reuse/customization, method overriding, etc



- Encapsulation & data hiding

  The mechanism for restricting access to an object's components.



- Polymorphism

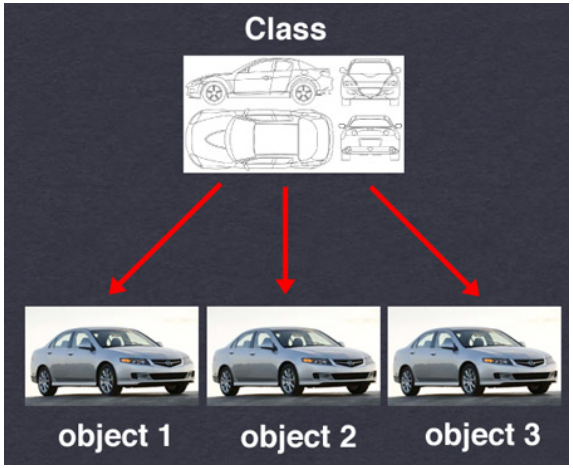
  e.g. operator overloading

Classes & Objects

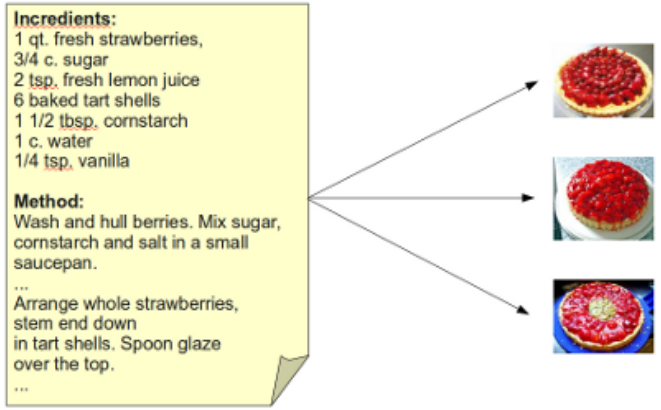
Classes

- Are at the heart of OOP.
- Are program units (like functions, modules) to package static data members (attributes) and function members (methods) together.
- Methods are just functions defined inside classes.
- Classes are simply factories/blueprints for generating objects.
- Objects are a.k.a. instances of the class.
- The process of creating new instances of a class is instantiation.

Classes & Objects: Examples



The `Car` class



The `Cake` class

- We define classes using the `class` keyword.
- Class definition creates class objects.

```
class <class_name> [( parent classes if any)]:
       "optional documentation string"
       static_member_declarations
       method_declarations
```

Example

```
class MyFirstClass:
      """The simplest class ever"""
      pass                  # do nothing
```

Example

```
class MyOwnClass:
      x = 2                 # attribute
      def f(self):          # method
          return 'hello world'
```

- We define classes using the `class` keyword.
- Class definition creates class objects.

```
class <class_name> [( parent classes if any)]:
       "optional documentation string"
       static_member_declarations
       method_declarations
```

Example

```
class MyFirstClass:
      """The simplest class ever"""
      pass              # do nothing
```

Example

```
class MyOwnClass:
      x = 2             # attribute
      def f(self):      # method
          return 'hello world'
```

- We define classes using the `class` keyword.
- Class definition creates class objects.

```
class <class_name> [( parent classes if any)]:
       "optional documentation string"
       static_member_declarations
       method_declarations
```

Example

```
class MyFirstClass:
      """The simplest class ever"""
      pass              # do nothing
```

Example

```
class MyOwnClass:
      x = 2             # attribute
      def f(self):      # method
          return 'hello world'
```

Class Methods

- Class methods vs standard functions?

Method

Defined within and belongs to a class.

A must-have extra parameter named `self` - refers to the object *itself*.

- When calling the method from inside the class, the function name be prefixed with `self` as: `self.methodname()`.

- But when calling from outside, no need to specify `self`. Python will automatically take care of it.

Example

```
class MyAdd:
    def add(self, x, y):
        return x+y

    def check(self):
        print self.add(2,4)     # call add() of same class
```

Class Methods

- Class methods vs standard functions?

Method

Defined within and belongs to a class.

A must-have extra parameter named `self` - refers to the object *itself*.

- When calling the method from inside the class, the function name be prefixed with `self` as: `self.methodname()`.

- But when calling from outside, no need to specify `self`. Python will automatically take care of it.

Example

```
class MyAdd:
    def add(self, x, y):
        return x+y

    def check(self):
        print self.add(2,4)     # call add() of same class
```

Class Instances

- `Instance` is an object of a class created at run-time.
- Each time a class is called, it creates a new instance (*class instantiation*).
- Class call uses function notation. Just use the class name as a function name as:

  `ClassName()` - returns a new instance of the class `<className>`

- We can then access the class methods through the instances as: `instance.methodname()`.

Example

```
I=MyFirstClass()          # new instance of MyFirstClass
I1=MyOwnClass()           # new instance of MyOwnClass
I2=MyOwnClass()           # another instance
x=MyAdd()
```

Class Instances

- `Instance` is an object of a class created at run-time.
- Each time a class is called, it creates a new instance (*class instantiation*).
- Class call uses function notation. Just use the class name as a function name as:

  `ClassName()` - returns a new instance of the class `<className>`

- We can then access the class methods through the instances as: `instance.methodname()`.

Example

```
I=MyFirstClass()              # new instance of MyFirstClass
I1=MyOwnClass()               # new instance of MyOwnClass
I2=MyOwnClass()               # another instance
x=MyAdd()
```

Class Instantiation

- `Instantiation` is simply calling a class object to create an *empty* object.
- But we often want our objects to have some initial state/data.
- The `__init__()` method (`init` ≡ initialize) is a special method (constructor) that initializes our instance objects with some initial state. It is *invoked automatically*.

Example

```
class person:
      def __init__(self, who):
            self.name=who                    # class attribute
      def display(self):
            print "Hello", self.name

I1=person('Alice')
I2=person('Bob')
I1.display()                    # or person('Alice').display()
I2.display()                    # or person('Bob').display()
```

Class Instantiation

- `Instantiation` is simply calling a class object to create an *empty* object.
- But we often want our objects to have some initial state/data.
- The `__init__()` method (`init` ≡ initialize) is a special method (constructor) that initializes our instance objects with some initial state. It is *invoked automatically*.

Example

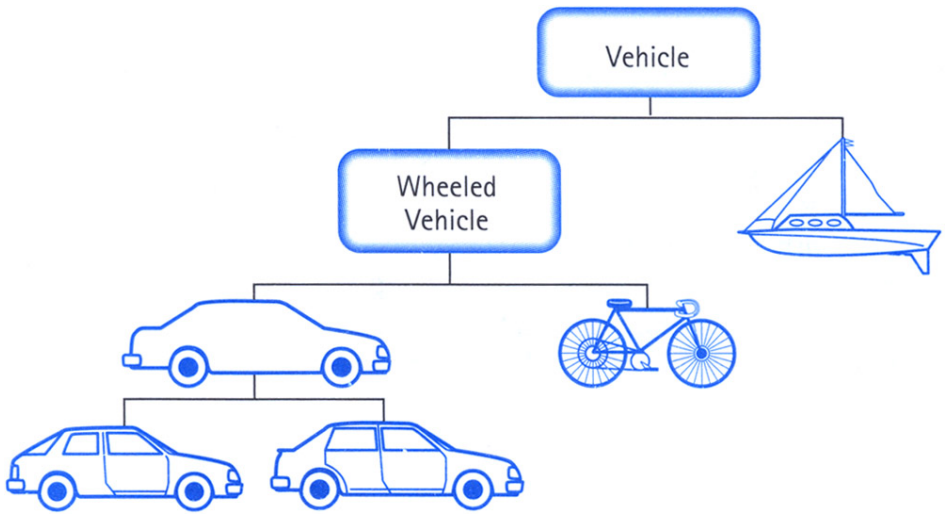
```
class person:
      def __init__(self, who):
           self.name=who                    # class attribute
      def display(self):
           print "Hello", self.name

I1=person('Alice')
I2=person('Bob')
I1.display()                    # or person('Alice').display()
I2.display()                    # or person('Bob').display()
```

OOP Principles: Inheritance

- Classes can inherit attributes and methods from other classes.
- Child class (`subclass`) vs Parent/base class (`superclass`)
- Single inheritance vs Multiple inheritance
- Inheritance models relationships of type `isa`.

  e.g. {`circle,triangle, rectangle`} isa `shape`
  {`SavingsAccount,CurrentAccount`} isa `Account`



Inheritance: Creating Subclasses

- Much like creating base classes, except that a list of parent classes to inherit from is specified.

```
class <SubclassName>(Parent1, Parent2, Parent3...):
        """optional documentation string"""
        static_member_declarations
        method_declarations
```

Example

```
class parent:
      def parentmethod(self):
          print 'Inside parent method'

class child(parent):                # subclass of 'parent'
      def childmethod(self):
          print 'Inside child method'

p=parent()              # instance of 'parent' class
c=child()               # instance of 'child' class
```

Inheritance: Creating Subclasses

- Much like creating base classes, except that a list of parent classes to inherit from is specified.

```
class <SubclassName>(Parent1, Parent2, Parent3...):
        """optional documentation string"""
        static_member_declarations
        method_declarations
```

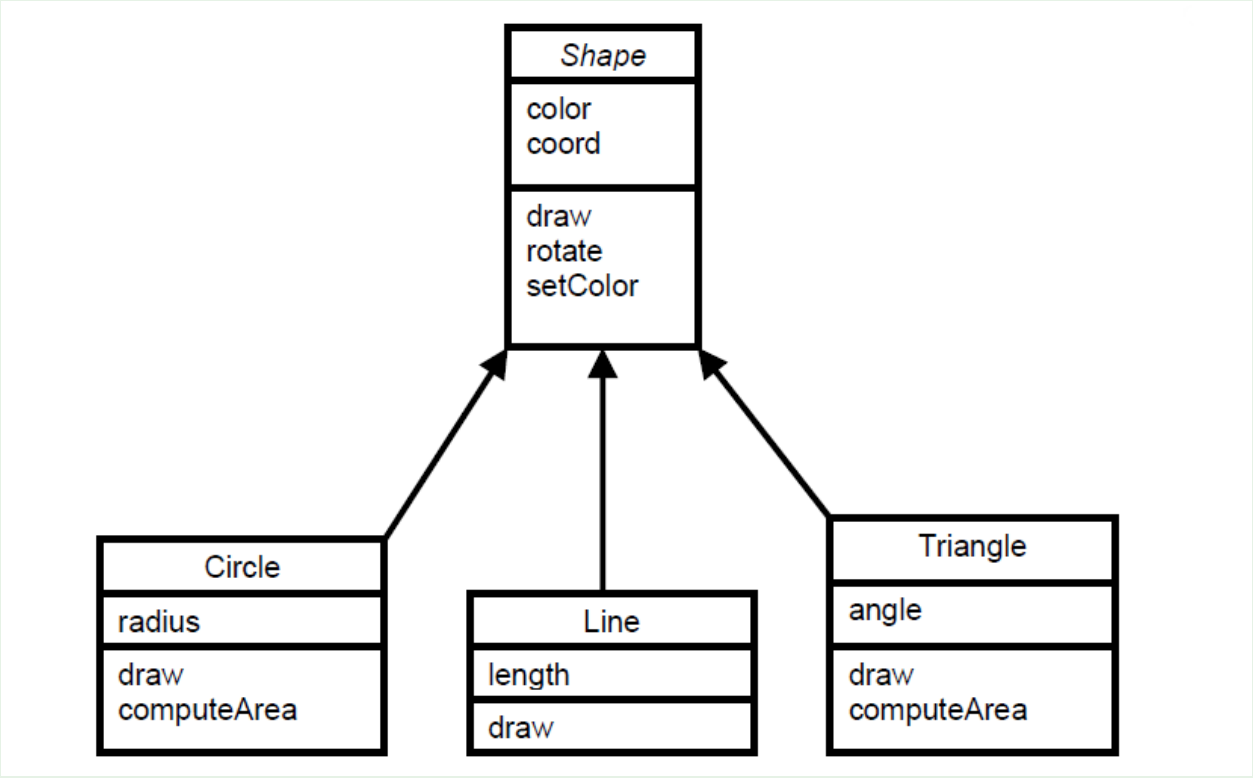
Example

```
class parent:
      def parentmethod(self):
          print 'Inside parent method'

class child(parent):                # subclass of 'parent'
      def childmethod(self):
          print 'Inside child method'

p=parent()                  # instance of 'parent' class
c=child()                   # instance of 'child' class
```

Example


Shape
color
coord
draw
rotate
setColor
Circle
radius
draw
computeArea
Line
length
draw
Triangle
angle
draw
computeArea

Inheritance: Method Overriding

- A subclass may define its own version of a method already defined in the parent class.
- This is called `method overriding`.

Example

```
class parent:
      def func(self):
          print "This is parent's func() "

class child(parent):
      def func(self):                          # overriden
          print "This is child's func() "

p=parent()
c=child()
p.func()
c.func()
```

OOP Principles: Encapsulation

- By default, attributes in Python are `public` i.e. they can be accessed from anywhere (both within module and outside).
- We may want some attributes to be not accessible (this is called `Data Hiding`)
- Data can be protected by making members `private` or `protected`.
- Python's way of hiding data:

Usage	Is	Meaning
`var`	`public`	Can be accessed from inside and outside
`_var`	`protected`	Like `public`, but not directly accessed from outside.
`__var`	`private`	Can not be accessed from outside