Information Infrastructure II

INFO I211 – Spring 2014 – Sections 18530 & 22519

Lecture 10 – 2014.02.17 & 2014.02.18

Instructor:
Mitja Hmeljak,
http://mypage.iu.edu/~mitja
mitja@indiana.edu

OOP: what have we learned so far?

Object-oriented programming is a programming language model organized around "objects"

An object is a software bundle of related *attributes* and behavior : *methods*

A class is a *blueprint* or prototype from which objects are created

```
class Player(object):
  def __init__(self, name = "Enterprise",
 fuel = 0):
    self.name = name
    self.fuel = fuel
  def status(self):
    ...
myship = Ship("Apollo")
myship.status()
```

Object encapsulation: *private* and *public* attributes and methods

Today: working with multiple classes

Use *inheritance* to build specialized classes:

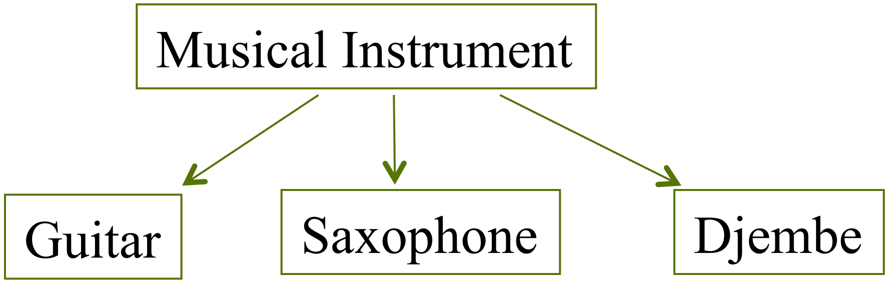
1. Derive new classes from existing ones
2. Extend the definition of existing classes
3. Override method definitions of existing classes

Create objects of different classes in the same program

Allow objects to communicate with each other

Create more complex objects by combining simpler ones

Inheritance Models "is a" Relationship

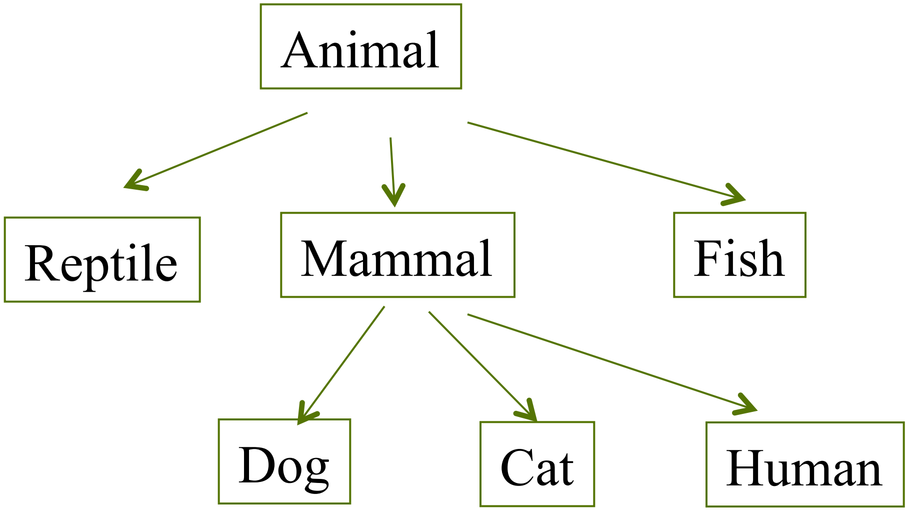


Musical Instrument

Guitar

Saxophone

Djembe



Animal

Reptile

Mammal

Fish

Dog

Cat

Human

Using Inheritance to Create New Classes

Inheritance is an OOP technique that

1. allows a new class to be based on an existing class

2. the new class automatically gets (*inherits*)
   - all the methods and
   - all the attributes
   - ... of the existing (i.e. original i.e. parent) class

Using Inheritance to Create New Classes

Children classes inherit

- all the capabilities (methods) and
- all the properties (attributes) the parent class
- (children classes are also called *derived* classes)

Why is this useful? Get code for free!

Code-reuse: inheritance allows a new class to re-use code which already existed in another class (the parent class)

Derived Classes are *New* Classes

Derived classes can provide specializations of existing classes by adding new attributes and methods.

It's also called class ***subtyping***.

The new class (and the objects instantiated from that class) has data or behavior aspects that are not part of the inherited class.

Example: **Overriding** the "+" operator

"+" is a method that can have different meanings:

- the "+" method is used to add two *numbers*
- the "+" method is concatenate two *strings*, etc.

← *the same method does something different*

Inheritance Example: Animal Class

```
class Animal(object):
    def __init__(self, name):    # Constructor
        self.name = name
    def get_name(self):
        return self.name

class Cat(Animal):
    def talk(self):
        return 'Meow!'

class Dog(Animal):
    def talk(self):
        return 'Woof! Woof!'

animals = [Cat('Jack'), Cat('Jo'), Dog('Jill')]

for animal in animals:
    print animal.talk() + ' I am ' + animal.get_name()
```

Base class: A class upon which another is based; it is inherited from by a derived class
Derived class: A class that is based upon another class; it inherits from a base class

keeping Python code organized in Classes: one file per (externally used) class

e.g. the three classes from the previous page :

```
class Animal(object)...
class Cat(Animal)...
class Dog(Animal)...
```

are more clearly placed in separate files:

```
animal.py
cat.py
dog.py
```

then use `from ... import` and

```
if __name__ == '__main__':
```

Altering the Behavior of Inherited Methods: Overriding

Override: To redefine how inherited method of base class works in derived class

Two choices when overriding methods:

a. provide completely new functionality in overridden method, or:
b. incorporate functionality of overridden method, and add new functionality

Overriding to Create a New Version

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

    def talk(self):
        return 'Hello!'

class Cat(Animal):
    def talk(self):
        return 'Meow!'
```

Overriding to Add More

One can incorporate the inherited method's functionality in the overridden method, then add more – i.e. write additional behavior in the *child* class:

```
Class Card(object):
...
class Positionable_Card(Card):
  def __init__(self, rank, suit, face_up = True):

    super(Positionable_Card, self).__init__(rank, suit)
    #invoke parent's __init__ method by calling super()

    self.is_face_up = face_up
    # additional behavior: save the "face_up" parameter
```

in this example, Card is the *superclass* of Positionable_Card (superclass == base class)

Invoking Base Class Methods

Incorporate inherited method's functionality by calling super()

In the previous example, the constructor for Positionable_Card invokes the Card constructor, and then goes on to assign a new attribute

- super() invokes the constructor method of the superclass (i.e. Card) thus:
  - The first argument is the *current* class name: Positionable_Card
  - The second argument is a reference to object itself: self
  - Then call the superclass method, with parameters: sent, __init__(rank, suit)

```
Class Card(object):
...
class Positionable_Card(Card):
  def __init__(self, rank, suit, face_up = True):

    super(Positionable_Card, self).__init__(rank, suit)
    #invoke parent's method by calling super()
```

Understanding Polymorphism

Polymorphism: Aspect of object-oriented programming that allows you to send the **same message to objects of different classes** related by inheritance, and achieve **different** but appropriate **results** for each object

When you invoke talk() method of Cat object, you get different result than when you invoke the same method of a Animal (or Dog) object

Summary: What can be Done Through Inheritance

the new class gets *all* methods and attributes of existing class

the new class can also define *additional* methods and attributes, to create more specialized version of existing class

the new class can *override* the old methods

when overriding a method, the new definition can:

a. have completely different functionality than the original definition, or

b. the new definition can incorporate the functionality of the original

The super() function allows you to invoke the method of a superclass