

# Python Programming

Week Five, Class Two  
OOP in python, cont.

## Outline

- Review
  - Object-oriented programming
- Another useful object in python – sys
  - And writing and running a program
- Inheritance and polymorphism
- Mechanics of inheritance in python
  - Invoking parent class methods
  - Invoking parent class constructor
- UML

## Interview Question

- What is OOP? \_\_\_\_\_.

## Fill in the blank

- Objects \_\_\_\_\_ stuff & \_\_\_\_\_ stuff

## Interview Question:

- What is a class?
- How do you create a class in python?

## Interview Question:

- What is data hiding?

## Interview Question:

- Why is data hiding a good idea?

## Objects in Python

- What built-in object lets you determine the current working directory?
- How do you find out the values of all the currently defined environment variables?

# sys - Another useful singleton in python

- `sys` – offers access to two important streams
  - `sys.stderr` – prints to stderr
  - `sys.stdout` – prints to stdout
- `sys.path` tells you where python looks for modules – its already a list!
  - And you can change it in a program by adding to the list.
- The distinction matters when you're writing scripts to run at the command line.
  - A quick aside: how to run a python program at the command line
  - How to redirect stderr and stdout

See: <http://docs.python.org/tutorial/modules.html>

## How to write a program you can run in the Unix shell (a “script”)

1. Create a new file for your program

```
sysdemo.py
```

2. Add “she-bang” to the top

```
#!/usr/bin/env python
```

3. Make it executable

```
chmod a+x sysdemo.py
```

sysdemo.py

4. Add some code

```
#!/usr/bin/env python
import sys
if __name__ == "__main__":
    sys.stderr.write("Hi! I wrote to
sys.stderr.\n")
    sys.stdout.write("Hi! I wrote to
sys.stdout.\n")
```

## Run it like so:

```
minerva:Desktop admini$ ./sysdemo.py
Hi! I wrote to sys.stderr.
Hi! I wrote to sys.stdout.
minerva:Desktop admini$ ./sysdemo.py
> out.txt
Hi! I wrote to sys.stderr.
minerva:Desktop admini$ more out.txt
Hi! I wrote to sys.stdout.
minerva:Desktop admini$ ./sysdemo.py
> out.txt 2> err.txt
```

- Use > to redirect *stdout* and save to a file.
- Use 2> to redirect *stderr* and save to a file.
- Or..send it to the “trash” (> /dev/null)

## Object-oriented programming

- Key ideas associated with OOP:
  - *Encapsulation and data hiding (Fri last week)*
  - *Inheritance (today)*
  - *Polymorphism (today)*
  - Patterns (Friday next week)

# Key ideas in OOP (know these!)

- Encapsulation and data hiding
- *Inheritance*
- *Polymorphism*
- Patterns

## Inheritance

- Means: Defining new classes in terms of other classes.
- Terms:
  - **Parent** (or base) class
  - **Child** (or derived) class
- It provides a way to get **polymorphism** – same method name, different behaviours
  - A very useful example: Sub-classing ContentHandlers for XML parsing (XML is a self-documenting format for representing data)

# Inheritance types

- Single-inheritance
  - Inherit from one parent only
- Multiple inheritance
  - inherit from more than one parent
  - Confusing because what if two parents implement the same method?
  - Which one will the sub-class choose?

## How it works in python

- Define a base (parent) class

```
class Bird:  
    def __init__(self):  
        self._hungry = True  
  
    def eat(self):  
        if self._hungry:  
            print "Aahh..."  
            self._hungry = False  
        else:  
            print "No, thanks!"
```

Use “\_”  
notation to  
indicate it’s a  
private  
variable

## Define derived (child) class

- SongBird inherits from Bird

```
class SongBird(Bird):  
  
    def __init__(self):  
        Bird.__init__(self)  
        self._sound = "Tweet Tweet"  
  
    def sing(self):  
        print self._sound
```

- Invoke parent class constructor.
- Initializes `_hunger` instance variable.

## Bird and SongBird in the interpreter

- `b` and `sb` are instances
- `b` is a `Bird`
- `sb` is a `SongBird`
- `sb` inherits its `eat` method from `Bird`.
- Internally, it relies on an instance variable from `Bird`.

```
>>> import Animals as a  
>>> b = a.Bird()  
>>> b.eat()  
Aahh...  
>>> b.eat()  
No, thanks!  
>>> sb = a.SongBird()  
>>> sb.sing()  
Tweet Tweet  
>>> sb.eat()  
Aahh...  
>>> sb.eat()  
No, thanks!
```

## What happens if you don't invoke the parent class constructor?

- Constructors are usually where instance variables are *initialized* (given default values)
- If the child class does not invoke the `__init__` method of parent classes, their *inherited* instance variables do not get initialized.
- So...if you don't invoke the `__init__` method, inherited methods may fail if they try to access instance variables that don't exist.

## Crow – subclass of SongBird

- Invoke the parent class constructor to set instance variables defined in the parent class

python

```
>>> c = a.Crow()
>>> c.sing()
Caw Caw
>>> c.eat()
Aahh...
>>> c.eat()
No, thanks!
```

emacs

```
class Crow(SongBird):
    def __init__(self):
        SongBird.__init__(self)
        self._sound = "Caw Caw"
```

## Try removing the call to the parent class constructor

- Use the comment character (#) to temporarily remove the statement:

```
class Crow(SongBird):  
  
    def __init__(self):  
        # SongBird.__init__(self)  
        self._sound = "Caw Caw"
```

## Re-import and repeat

- The `eat` method fails because we didn't set the `_hungry` instance variable
- Why does `sing` work?

```
>>> reload(a)  
<module 'Animals' from 'Animals.py'>  
>>> c = a.Crow()  
>>> c.sing()  
Caw Caw  
>>> c.eat()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "Animals.py", line 11, in eat  
    if self._hungry:  
AttributeError: Crow instance has no attribute '_hungry'
```

## Multiple Inheritance – simple example

- Cat inherits from Feline and HousePet

```
class Feline:
    def purr(self):
        print "purr purr"

class HousePet:
    def demandFood(self):
        print "Feed me now"

class Cat(Feline,HousePet):
    def demandFood(self):
        print "Meow! Meow!"
```

## Multiple inheritance – simple example


- The cat instance can do what both its parents can do

```
>>> c = a.Cat()
>>> c.purr()
purr purr
>>> c.demandFood()
Meow! Meow!
>>> 
```

What happens if the same method is available from both parents?

- Which one does python choose?
- Specify the parent you want

```
class Horse:
    def makeNoise(self):
        print "whinny"
class Donkey:
    def makeNoise(self):
        print "bray"
class Mule(Horse,Donkey):
    def makeNoise(self):
        Donkey.makeNoise(self)
```



## How it works

- Instance of Mule uses the Donkey's version

```
>>> h = a.Horse()
>>> d = a.Donkey()
>>> m = a.Mule()
>>> h.makeNoise()
whinny
>>> d.makeNoise()
bray
>>> m.makeNoise()
bray
```

## One more class:

```
class Flock:

    def __init__(self):
        self._birds = []

    def addBird(self, bird):
        self._birds.append(bird)

    def countBirds(self):
        return len(self._birds)
```

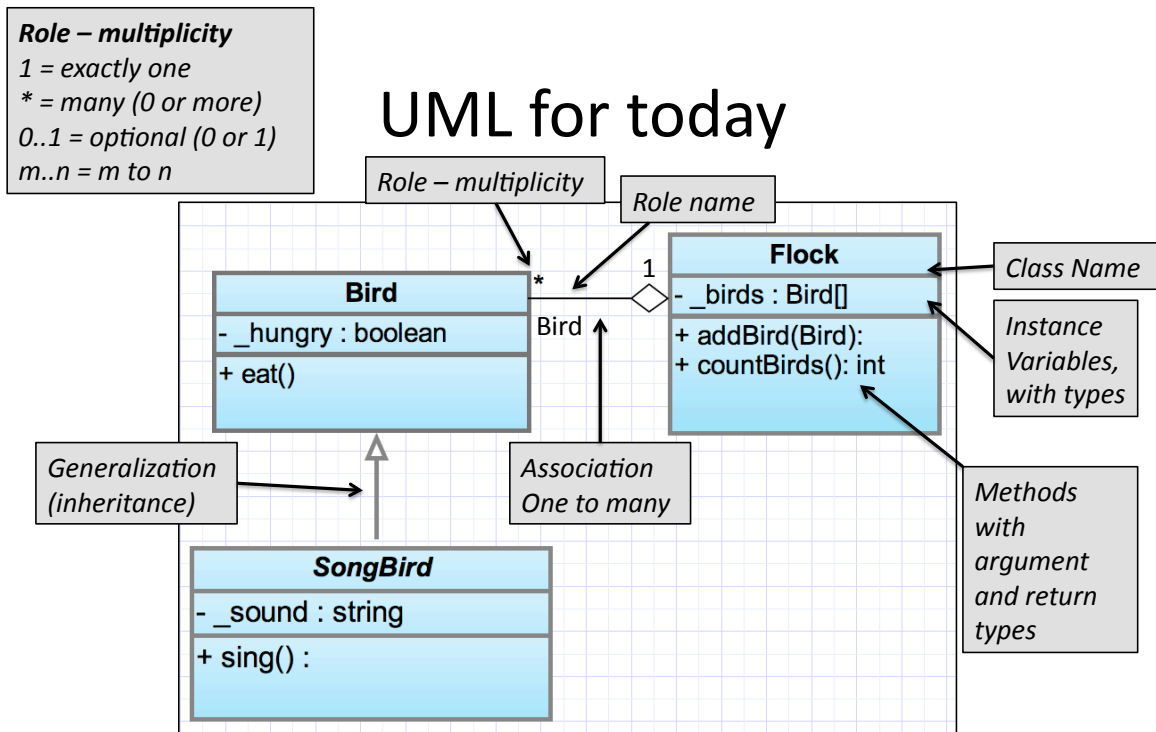
Note how the Flock has many instances of Bird. It contains Birds. (Containment relationship)

## UML

- Unified Modeling language
- Graphical notation for modeling software systems
- Classes have attributes and methods (they have stuff and can do stuff)
- UML **class diagrams** illustrate these relationships
  - Other types of diagrams in UML, like: state and interaction diagrams showing the way objects change and interact in a program

# UML class diagram

- Will include Bird, SongBird, and Flock
- For each class, we show
  - Class Name
  - Attributes
  - Operations (Methods)
- We draw them together to show their relationships
  - Associations (such as containment)
  - Generalizations (via inheritance)



**Note:** If you include role name, you can omit its corresponding instance variable in the Class “box” – use your judgement – your goals are clarity and communication!!

# Project One

- You'll define a class hierarchy with containment relationships
- AND draw a UML diagram
- Will be assigned next week!
  - Uses your Range class from HW7