Recap

• **Object Orientation**
  – merge data and functions (that operate on the data) together into classes
  – class is like a blue print of an object
  – objects are instances of a class
  – typically two kinds of members in a class
    • members that store data are *attributes*
    • members that are functions are *methods*
Example 1: Class Coordinate

Keyword to indicate declaration of a class

class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        x_diff_sq = (self.x - other.x)*(self.x - other.x)
        y_diff_sq = (self.y - other.y)*(self.y - other.y)
        return sqrt(x_diff_sq + y_diff_sq)
Example 1: Class Coordinate

Name of a class

```python
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        x_diff_sq = (self.x - other.x)**2
        y_diff_sq = (self.y - other.y)**2
        return sqrt(x_diff_sq + y_diff_sq)
```
Example 1: Class Coordinate

Parent class

class Coordinate(object):
    
def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        x_diff_sq = (self.x - other.x) * (self.x - other.x)
        y_diff_sq = (self.y - other.y) * (self.y - other.y)
        return sqrt(x_diff_sq + y_diff_sq)
**Example 1: Class Coordinate**

```python
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

def distance(self, other):
    x_diff_sq = (self.x - other.x)*(self.x - other.x)
    y_diff_sq = (self.y - other.y)*(self.y - other.y)
    return sqrt(x_diff_sq + y_diff_sq)
```
Example 1: Class Coordinate

class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        x_diff_sq = (self.x - other.x)**2
        y_diff_sq = (self.y - other.y)**2
        return sqrt(x_diff_sq + y_diff_sq)
Example 1: Class Coordinate

```python
c = Coordinate(3,4)  # new object of type Coordinate with initial attributes
z = Coordinate(0,0)

d = c.distance(z)
print(d)

d = Coordinate.distance(c,z)
print(d)
```

Equivalent
Operator Overloading

What the operator does, depends on the objects it operates on. For example:

```python
>>> a = "Hello "; b = "World"
>>> a + b # concatenation
'Hello World'
>>> c = 10; d = 20
>>> c + d # addition
30
```

This is called *operator overloading* because the operation is overloaded with more than one meaning.
Example 2: Class Fraction

class Fraction(object):
    def __init__(self, num, denom):
        self.num = num
        self.denom = denom
    def __str__(self):
        return str(self.num) + "/" + str(self.denom)
    def __add__(self, other):
        num = self.num*other.denom + self.denom*other.num
        denom = self.denom*other.denom
        return Fraction(num, denom)
    def __sub__(self, other):
        num = self.num*other.denom - self.denom*other.num
        denom = self.denom*other.denom
        return Fraction(num, denom)
    def __float__(self):
        return self.num/self.denom
Methods: set and get

A well designed class provides methods to get and set attributes.
• These methods define the *interface* to that class.
• This allows to perform error checking when values are set, and to hide the implementation of the class from the user.
class Time(object):
    def __init__(self, hour, min):
        self.setHour(hour)
        self.setMin(min)

    def setHour(self, hour):
        if 0 <= hour <= 23:
            self.hour = hour
        else:
            raise ValueError("Invalid hour value")

    def setMin(self, min):
        if 0 <= min <= 59:
            self.min = min
        else:
            raise ValueError("Invalid min value")
Methods: set and get

```python
def getHour(self):
    return self.hour

def getMin(self):
    return self.min

t = Time(15, 45)
print(t.getHour())
t.setHour(34)
t.setHour(34)
print(t.getHour())
```
Class Hierarchy

- **parent class** (superclass)
- **child class** (subclass)
  - *inherits* all data and behaviors of parent class
  - *add* more *info*
  - *add* more *behavior*
  - *override* behavior
Class Inheritance

Sometimes, we need classes that share certain (or very many, or all) attributes but are slightly different.

• Example 1: Geometry
  a point (in 2 dimensions) has an $x$ and $y$ attribute
  a circle is a point with a radius
  a cylinder is a circle with a height

• Example 2: People at universities
  A person has an address.
  A student is a person and selects modules.
  A lecturer is a person with teaching duties.

• In these cases, we define a base class and derive other classes from it.
• This is called inheritance.
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)

- everything is an object
- class object implements basic operations in Python, like binding variables, etc

INHERITANCE: SUBCLASS

```python
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return "cat:"+str(self.name)+":"+str(self.age)
```

- add new functionality with `speak()`
  - instance of type `Cat` can be called with new methods
  - instance of type `Animal` throws error if called with `Cat`'s new method
- `__init__` is not missing, uses the `Animal` version

Class Inheritance

• subclass can have **methods with same name** as superclass

• for an instance of a class, look for a method name in **current class definition**

• if not found, look for method name **up the hierarchy** (in parent, then grandparent, and so on)

• use first method up the hierarchy that you found with that method name

import math

class Point:
    """Class that represents a point """
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

class Circle(Point):
    """Class that represents a circle """
    def __init__(self, x=0, y=0, radius=0):
        Point.__init__(self, x, y)
        self.radius = radius
import math

class Point:
    # this is the base class
    """Class that represents a point """
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

class Circle(Point):
    # is derived from Point
    """Class that represents a circle """
    def __init__(self, x=0, y=0, radius=0):
        Point.__init__(self, x, y)
        self.radius = radius

Source: Computational Science and Engineering in Python by Hans Fangohr, Engineering and Environment, University of Southampton, United Kingdom
def area(self):
    return math.pi * self.radius ** 2

class Cylinder(Circle):
    # is derived from Circle

    """Class that represents a cylinder""

def __init__(self, x=0, y=0, radius=0, height=0):
    Circle.__init__(self, x, y, radius)
    self.height = height

def volume(self):
    return self.area() * self.height

Source: Computational Science and Engineering in Python by Hans Fangohr, Engineering and Environment, University of Southampton, United Kingdom