CSC326 Object Oriented Programming

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.0	2011-09		JZ

Contents

1 Agenda

- Classes and Objects
- Classes and Attributes
- Classes and Methods
- Operator Overloading
- Inheritance

2 Classes and Objects

- We have seen built-in types
- User defined types: class
- Example: point in 2-D space

```
class Point( object ) :
    """reprsents a point in 2-D space"""
```

- header: indicates a new class with name "Point"
- header: indicates the class is a kind of "object", a built-in type
- body: a docstring explaining what the class is for

```
>>> print Point
<class '__main__.Point'>
```

- Class is a factory of objects
  - Call Point() to create instance as if it were a function
  - returns a "reference" to a Point object

```
>>> blank = Point()
>>> print blank
<__main__.Point instance at 0xb7e9d2ac>
```

- Note Point instance and Point class is **NOT** the same thing

```
>>> print type(blank)
<class '__main__.Point'>
```

- Attributes
  - Each object has named elementes, called attributes (or fields) In
  - python, attribuites are introduced by use (with dot notation), not by declaration (C++)

```
>>> blank.x = 3.0
>>> blank.y = 4.0
>>> print blank.x
3.0
>>> distance = math.sqrt( blank.x**2 + blank.y**2)
>>> print distance
5.0
```

- Rectangle

```
class Rectangle( object ) :
      """ represent a rectangle.
          attributes: width, height, corner
      """
```

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

```
def find_center( box ) :
    p = Point()
    p.x = box.corner.x + box.width/2.0
    p.y = box.corner.y + box.height/2.0
    return p
```

- Objects are mutable

def grow_rectangle(rect, dwidth, dheight) : rect.width = `dwidth rect.height` = dheight

- Copying Objects
  - Alias (copying references) is not always wanted
  - Copying content of object to another object is sometimes wanted
  - Shallow Copy

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0
>>> import copy
>>> p2 = copy.copy(p1)
>>> print p2.x, p2.y
3.0 4.0
>>> p1 is p2
False
```

```
>>> box2 = copy.copy( box )
>>> box2 is box
>>> False
>>> box2.corner is box.corner
True
```

Note
Shallow copy copies the object and any references it contains, but not the embedded objects.

- Deep Copy

```
>>> box3 = copy.deepcopy( box )
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

3 Classes and Functions

- Pure function

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

- Function that does not modify objects

```
def add_time( t1, t2 )
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

- protype and patch!

```
def add_time( t1, t2 )
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    if sum.second >= 60 :
        sum.second -= 60
        sum.minute += 1
    if sum.minute >= 60 :
        sum.minute -= 60
        sum.hour += 1
    return sum
```

- modifiers

def increment(time, seconds) time.second = `seconds if time.second >= 60 :  if time.second >= 60 : time.second -= 60 time.minute` = 1 if time.minute >= 60 : time.minute -= 60 time.hour += 1

Note
Is it correct? Write the correct version without using loops

- Planning

```
def time_to_int( time ) :
    minutes = time.hour * 60 + time.minute
    seconds = minute * 60 + time.second
    return seconds
def int_to_time( seconds ) :
    time = Time()
    minutes, time.second = divmod( seconds, 60 )
    time.hour, time.minute = divmod( minutes, 60 )
    return time
def add_time( t1, t2 ) :
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

4 Classes and Methods

- What is object orientation?
- So far
  - Encapsulation: we have seen it with class attributes
  - Actions (the verb phrase) are captured in ordinary function
  - Function call on objects (verb-centric)
- Change of perspective (noun-centric)
  - Object is given a function (method) to act on
  - Leads to change of syntax:
  - method(o, . . .) → o.method(. . .)

```
class Time( object ) :
   """ .. """
def print_time( time ) :
   print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)
```

```
>>> Time.print_time( start )
```

```
class Time( object ) :
   """ .. """
   def print_time( time ) :
       print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)
```

```
>>> start.print_time()
```

- By convention, first parameter is named self

```
class Time( object ) :
   """ .. """
   def print_time( self ) :
       print '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

```
# inside class Time
   def increment( self, seconds ) :
       seconds += self.time_to_int()
       return int_to_time( seconds )
```

- Contract-based programming
  - Contract between class developer and user
  - A **limited** set of functions (methods) are defined for a class
  - Users use and only uses method to modify attributes
  - In reality it is often violated (e.g., C++ friend class)
- Constructor
  - invoked when an object is instantiated (when Time() is called)

```
# inside class TimeXS
   def __init__( self, hour=0, minute=0,second=0) :
       self.hour = hour
       self.minute = minute
       self.second = second
```

- Dumper
  - invoked when print the object

```
# inside class TimeXS
   def __str__( self ) :
       return '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)
```

- get rid of print_time, and allow it to be printed the same way as all other types

5 Operator Overloading

- operators are nothing but methods

```
# inside class Time
   def __add__( self, other ) :
       seconds = self.time_to_int() + other.time_to_int()
       return int_to_time( seconds )
```

```
>>> start = Time( 9, 45 )
>>> duration = Time( 1, 35 )
>>> print start + duration
11:20:00
```

- type-based dispatch

```
# inside class Time
   def __add__( self, other ) :
       if isinstance( other, Time ) :
           return self.add_time( other )
       else :
           return self.increment( other )
```

```
>>> start = Time( 9, 45 )
>>> duration = Time( 1, 35 )
>>> print start + duration
11:20:00
>>> print start + 1337
10:07:17
```

- What if you do

```
>>> print 1337 + start
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

- Rescue

```
# inside class Time
   def __radd__( self, other ) :
       return self.__add__( other )
```

- Polymorphic Programming again!

6 Inheritance

- Classes are used to model real world objects
- Real world objects have relationships among each other
- has-a relation: often an object is a container of other objects

```
class Card( object ) :
   def __init__( self, suit=0, rank=2 ) :
       self.suit = suit
       self.rank = rank
   def __cmp__( self, other ) :
       if self.suit > other.suit : return 1
       if self.suit < other.suit : return -1
       if self.rank > other.rank : return 1
       if self.rank < other.rank : return -1
       return 0
```

```
class Card( object ) :
   def __init__( self, suit=0, rank=2 ) :
       self.suit = suit
       self.rank = rank
   def __cmp__( self, other ) :
       return cmp( (self.suit, self.rank), (other.suit, other.rank) )
```

```
class Deck( object ) :
   def __init__( self, suit=0, rank=2 ) :
       self.cards = []
       for suit in range( 4 ) :
         for rank in range( 1, 14 ) :
             card = Card( suit, rank )
             self.cards.append( card )
   def pop_cards( self ) ...
   def add_cards( self ) ...
   def shuffle_cards( self ) ...
```

- is-a relation: often objects belong to the same catogries
  - Share similarities → can be abstracted by base class
  - Have own "personality → define inherited class

```
class Hand( Deck ) :
   def __init__( self, label = '' ) :
       self.cards = []
       self.label = label
```

- Note that *init* method is **overridden**
- In the mean time, all other attributes/methods for Deck can be used
  - class diagram

[NOTE] Do not be too carried away with inheritance. It is a pitfall to create too many level of inheritance because each level adds one level of indirection and one **barrier** of understanding. You may end up spending too much time designing the class rather than the actual work, and create code too hard to evolve.

7 Recap

- Change of perspective from procedural programming to object oriented programming
- Attributes and Methods
- Polymorphic programming: operator overloading
- Class relations: is-a / has-a
- Mindful of abuse