Lect 11 -- Inheritance

Rob Capra

INLS 490-172

Card Object

Every card has its own suit and rank, but there is only one copy of suit_names and rank_names.

```
class Card(object):

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5',
                  '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King']

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])

four_hearts = Card(2,4)
print four_hearts
queen_spades = Card(3,12)
print queen_spades
```

% = a way to fill-in parts of a string

__cmp__

- Allows use of <, >, == operators with objects
- Rules:
  - Take two objects
  - Return positive number if first is greater
  - Return negative number if second is greater
  - Return zero if both are equal

Q: What is the right ordering for Cards?

A: depends on the game

```
def __cmp__(self, other):
        # check the ranks
        if self.rank > other.rank: return 1
        if self.rank < other.rank: return -1
        # ranks are the same, so check the suits
        if self.suit > other.suit: return 1
        if self.suit < other.suit: return -1
        # suits and ranks are the same, so tie
        return 0
```

NOTE (from TPY): In Python 3, cmp no longer exists, and the __cmp__ method is not supported. Instead you should provide __lt__, which returns True if self is less than other

Deck Object

```
class Deck(object):
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)


mydeck = Deck()
print mydeck
```

Watch out for the double underscores!

```
def __init_(self):
```

http://docs.python.org/2/library/stdtypes.html#str.join
(also look at iterable)

Deck Object

```
import random

# in the Deck class
    def pop_card(self):
        return self.cards.pop()
    def add_card(self, card):
        self.cards.append(card)
    def shuffle(self):
        random.shuffle(self.cards)


mydeck = Deck()
print mydeck
mydeck.shuffle()
print mydeck
```

“veneer” (or “thin”) methods

Inheritance

- Inheritance allows us to define a new class that “inherits” methods and attributes from an existing object.
- We can then modify the new object.
- Example:
  - Hands of cards are similar to decks of cards
  - But have some important differences

```
class Hand(Deck):
   ''' Hand inherits from Deck. '''
```

Hand Object (inherits from Deck)

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


mydeck = Deck()
print mydeck
mydeck.shuffle()
print mydeck

myhand = Hand('new hand')
mycard = mydeck.pop_card()
myhand.add_card(mycard)
print myhand.label
print myhand
```

Deck modifies itself and Hand

```
 # in class Deck
    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

```
myhand = Hand('new hand')
mycard = mydeck.pop_card()
myhand.add_card(mycard)
print myhand.label
print myhand
mydeck.move_cards(myhand,4)
print myhand.label
print myhand
```

Inheritance Pros and Cons

- Pros
  - Can reduce amount of code / encourage code reuse
  - Sometimes reflects the real-world structure of objects
- Cons
  - Can make programs harder to read, understand,debug
  - Code is located in different places/classes
  - Often inheritance is not needed… there are other ways to structure things