Object-Oriented Programming

Python for Ecologists

Tom Purucker, Chance Pascale, Tao Hong, Jon Flaishans, Marcia Snyder

Ecological Society of America Workshop
Minneapolis, MN

chancebatwalrus@gmail.com

August 2, 2013

Principles of OOP

- Classes
  - Inheritance
  - Abstraction
  - Encapsulation
  - Polymorphism
- Methods
- Decoupling

Let’s take a gamble on classes

- Fundamental data unit for card games is Card
- Collection of Cards is Deck
- Subset of the Deck is Hand
- You need a CardGame to do something with the Cards, Deck, and Hands
- OldMaidGame is an type of CardGame
- OldMaidHand is a type of Hand used in OldMaidGame

ULM a nice city in Germany, oops UML

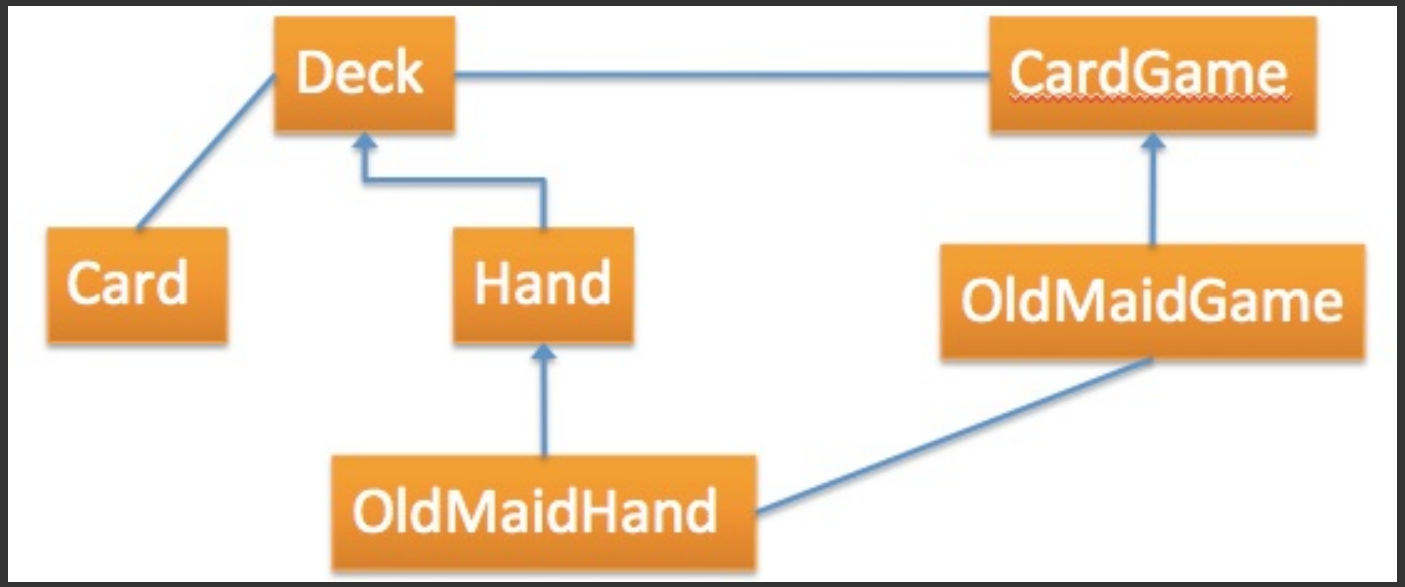

Figure: UML of OldMaidGame

Card Class Design

- Each card has a suit and a value
- Suits have no intrinsic use outside of a card so no real need to create a class for them
- Values are sometimes integers and other times strings, so represent them as string and associate true value for each game

Card Class Code

```
class Card:
  suitList=["Clubs", "Diamonds", "Hearts", "Spades"]
  rankList=['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 (self.rankList[self.rank] + "␣of␣" + self.suitList[self.suit])

  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
```

Deck class design

- A Deck is a collection of cards
- General functionality of decks are that they can be shuffled, the ’top’ card can be drawn, and many times knowing if there are any cards in the deck is necessary

Deck Class Code

```
class Deck:
  def __init__(self):
    self.cards = []
    for suit in range(4):
      for rank in range(1, 14):
        self.cards.append(Card(suit, rank))

  def printDeck(self):
    for card in self.cards:
      print card

  def __str__(self):
    s = ""
    for i in range(len(self.cards)):
      s = s + "␣" * i + str(self.cards[i]) + "\n"
    return s
```

Deck Class Code (continued)

```
def shuffle(self):
    import random
    nCards = len(self.cards)
    for i in range(nCards):
        j = random.randrange(i, nCards)
        self.cards[i],self.cards[j]=
                self.cards[j],self.cards[i]
```

Deck Class Code (continued)

```
def removeCard(self, card):
  if card in self.cards:
    self.cards.remove(card)
    return True
  else:
    return False
def popCard(self):
  return self.cards.pop()
def isEmpty(self):
  return (len(self.cards) == 0)
```

Hand Class Design

- Hand is an example of a Deck, a subset of the complete deck
- If we make Hand inherit from Deck, then we get the data structures that Deck contains
- Hand can also call the methods of Deck as if it was the superclass
- If a method of Deck class is not included in Hand code, then if that method is called on Hand object Deck method will be used.

Hand Class Code

```
class Hand(Deck):
    pass
    def __init__(self, name = ""):
        self.cards = []
        self.name = name

    def addCard(self, card):
        self.cards.append(card)

    def deal(self, hands, nCards = 999):
        nHands = len(hands)
            for i in range(nCards):
                if self.isEmpty():
                    break # break if out of cards
                card = self.popCard() # take the top card
                hand = hands[i % nHands] # whose turn is next?
                hand.addCard(card) # add the card to the hand
```

Hand Class Code (continued)

```
    # if next function not in code
    # Deck __str__ would be called
      def __str__(self):
        s = "Hand " + self.name
        if self.isEmpty():
          return s + " is empty\n"
        else:
          return s + " contains\n" + Deck.__str__(self)
```

CardGame Class Design/Code

- Card games contain a single deck, which should be shuffled at the beginning of each game

```
class CardGame:

  def __init__(self):
    self.deck = Deck()
    self.deck.shuffle()
```

OldMaidHand Design/Code

- Pretty much a normal card hand but needs method to find and remove all matches and return a count of matches

```
class OldMaidHand(Hand):
  def removeMatches(self):
    count = 0
    originalCards = self.cards[:]
    for card in originalCards:
      match = Card(3 - card.suit, card.rank)
      if match in self.cards:
        self.cards.remove(card)
        self.cards.remove(match)
        print "Hand %s: %s matches %s" %
              (self.name, card, match)
        count = count + 1
      return count
```

OldMaidGame Design

- Like all classes that "extend" CardGame, a deck is needed and it should be shuffled, oh wait CardGame already does this
- Playing the game performs the following steps:
  1. Take the Queen of hearts out of the deck
  2. Deal OldMaidHands
  3. Remove and count number of matches
  4. Each turn for a player is the same(25 turns):
     1. Check if hand is empty, if so do nothing
     2. Take neighbor playerâĂŹs âĂIJtopâĂİ card
     3. Remove and count number of matches
     4. Shuffle your hand
  5. Top score after all turns is winner

OldMaidGame Code

```
class OldMaidGame(CardGame):
    def play(self, names):
        self.deck.removeCard(Card(0, 12)) # remove Queen of Clubs
        self.hands = [] # make a hand for each player
        for name in names :
            self.hands.append(OldMaidHand(name))
            # deal the cards
            self.deck.deal(self.hands)
            print "----------_Cards_have_been_dealt"
            self.printHands()
            # remove initial matches
            matches = self.removeAllMatches()
            print "----------_Matches_discarded,_play_begins"
            self.printHands()
    turn = 0  # play until all 50 cards are matched
        numHands = len(self.hands)
        while matches < 25:
            matches = matches + self.playOneTurn(turn)
            turn = (turn + 1) % numHands
            print "----------_Game_is_Over"
            self.printHands()
```

OldMaidGame Code (continued)

```
def removeAllMatches(self):
        count = 0
        for hand in self.hands:
            count = count + hand.removeMatches()
        return count

def playOneTurn(self, i):
        if self.hands[i].isEmpty():
            return 0
        neighbor = self.findNeighbor(i)
        pickedCard = self.hands[neighbor].popCard()
        self.hands[i].addCard(pickedCard)
        print "Hand", self.hands[i].name, "picked", pickedCard
        count = self.hands[i].removeMatches()
        self.hands[i].shuffle()
        return count
```

OldMaidGame Code (continued)

```
def findNeighbor(self, i):
        numHands = len(self.hands)
        for next in range(1, numHands):
            neighbor = (i + next) % numHands
            if not self.hands[neighbor].isEmpty():
                return neighbor
```

What is __init__.py

- Files named __init__.py are used to mark directories on disk as a Python package directories. If you have the files
- mydir/spam/__init__.py mydir/spam/module.py and mydir is on your path, you can import the code in module.py as:
- import spam.module or
- from spam import module If you remove the __init__.py file, Python will no longer look for submodules inside that directory, so attempts to import the module will fail.
- The __init__.py file is usually empty, but can be used to export selected portions of the package under more convenient names, hold convenience functions, etc. Given the example above, the contents of the __init__ module can be accessed as
- import spam