CS313E: Elements of Software Design

Classes

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: October 4, 2012 at 09:46

Object Orientation

Recall that the basic idea of object oriented programming (OOP) and also of abstract data types is to view the world as a collection of objects.

Each object has:

- some *attributes* (i.e., data) that it maintains characterizing its current state;
- a set of actions (*methods*) that it can perform.

The programmer interacts with these objects by invoking their methods, which may:

- update the state of the object,
- query the object about its current state,
- compute some function of the state and externally provided values,
- some combination of these.

Class vs. Instance Data and Methods

When you define a class, you can include attributes (data) and methods that belong to the *class itself*, not to an instance of the class.

For example, when defining a class `PlayingCard`, the list of possible suits (Hearts, Clubs, Spades, Diamonds) and ranks (A, 2, ..., Q, K) are data relevant to the class itself, not to particular cards.

Keeping track of how many Cirles have been created is relevant to the Circle class, not to any particular Circle object.

Most classes don't have class data or methods, just instance data or methods.

Classes

In Python, an object type is defined as a class.

```
class ClassName (parentClass):
   """The docstring for the class."""

   # Data associated with the class, not with instances.
   CLASSDATA = ....   # there may not be any

   # Methods associated with the class, not with instances.
   # Don't use the self parameter.
   def methodname( params ):
      ...

   def __init__ (self, otherParams):
      """The constructor for the class."""
      ...

   <other methods>
```

A Simple Class: Circle

```
import math
class Circle:
    """Define a class of circles. Circles have an associated
    radius."""

    # This is a class attribute, not an instance attribute.
    _circlesCount = 0

    def printCount():
        # Note: this is a class method, not an instance method.
        print("Created " + str( Circle._circlesCount ) + " circles.")

    def __init__(self, radius):
        self._radius = radius
        Circle._circlesCount += 1

    def __str__(self):
        return "Circle with radius " + str(self._radius)

    def circumference(self):
        return 2 * math.pi * self._radius

    def area(self):
        return math.pi * (self._radius ** 2)
```

What is Self?

When you create a new object of a class (e.g., a new Circle or Rectangle), you create a new "container" to put the data in, and then fill it with the "attributes" (data) of the object. `self` is a pointer to that container, and used to refer to it within the class definition.

```
class Rectangle:

   def __init__(w, h):
      self._width = w
      self._height = h

   def getWidth(self):
      return self._width
   ...
```

In creating a new Rectangle object, first create the "container (pointed to by `self`), and then put into it the width and height fields. Why don't you need self outside the class definition?

A Simple Class: Circle

```
>>> from Circle import *
>>> Circle.printCount()
Created 0 circles.
>>> c1 = Circle(1)
>>> print (c1)
Circle with radius 1
>>> c1.circumference()
6.283185307179586
>>> c1.area()
3.141592653589793
>>> Circle.printCount()
Created 1 circles.
>>> c2 = Circle( 1 / (2 * math.pi))
>>> Circle.printCount()
Created 2 circles.
>>> c2.area()
0.07957747154594767
>>> c2.circumference()
1.0
```

Accessors and Mutators

Most classes have methods that allow obtaining and modifying the values of instance variables. These are called *accessors* and *mutators* (or "getters" and "setters."), respectively.

```
    def getRadius(self):
        """Return the Circle's current radius."""
        return self._radius

    def resize(self, newradius):
        """Change the Circle's radius value."""
        self._radius = newradius
```

```
>>> from Circle import *
>>> c1 = Circle(1)
>>> c1.area()
3.141592653589793
>>> c1.resize(2)
>>> c1.area()
12.566370614359172
```

Did You Get It?

Try the following on your own:

1. Extend the Circle class to add positional information, i.e., the x and y coordinates of the center of the circle.
2. Define a function to compute the distance between any two points (x_1, y_1) and (x_2, y_2). This can be outside the Circle class or a class method (rather than an instance method). Below is the formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

3. Add a method to test whether a point (x, y) is inside the circle (is within radius of the center), and another to see if it's on the circle.
4. Define a Rectangle class analogous to the Circle class. Do coordinates make sense here?
5. If you want to get really fancy, add some Turtle graphics functionality to display the Circles and Rectangles you define.

The Constructor

Using the turtle graphics module, when you say:

```
ttl = Turtle()
```

What you're really doing is calling the `__init__` function of the Turtle module, which might be defined something like:

```
class Turtle:

    def __init__():
        self._position = (0, 0)
        self._direction = 0
        self._isdown = False
        self._width = 2
        self._color = (0, 0, 0)
```

There is probably no `__str__` function in the Turtle module, because it's not very useful to print a turtle. *But you could have one. What might it do?*

OO Thinking

Suppose you want to write a Python program to play Poker. What is the *object oriented* way of thinking about this problem?

First question: What are the *objects* involved in a game of Poker?

OO Thinking

Suppose you want to write a Python program to play Poker. What is the *object oriented* way of thinking about this problem?

First question: What are the *objects* involved in a game of Poker?

- Card (rank and suit)
- Deck of Cards (an ordered collection of cards)
- Hand (a collection of 5 cards dealt from a Deck)
- Player (an entity that makes decisions about its hand)
- Driver (something to orchestrate the play of the game)

There are probably other ways to conceptualize this problem. It's good practice to put each class into its own file.

Designing a Class

Let's start at the bottom. Suppose we want to design a representation in Python of a playing Card.

- What data is associated with a Card?
- What actions are associated with a Card?

Designing a Class

Let's start at the bottom. Suppose we want to design a representation in Python of a playing Card.

- What data is associated with a Card?
- What actions are associated with a Card?

Data: rank and suit

Methods:

- Tell me your rank.
- Tell me your suit.
- How would you like to be printed?

Designing a Class

We'll define a `Card` class with those attributes and methods.

Notice that there are:

- a *class* definition (defines the type of an arbitrary playing card),
- *instances* of that class (particular cards).

Poker: Card Class

```
class Card:
    """A card object with a suit and rank."""

    # These are class attributes, not instance attributes
    RANKS = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)
    SUITS = ('Spades', 'Diamonds', 'Hearts', 'Clubs')

    # This is called as Card(rank, suit).
    def __init__(self, rank, suit):
        """Create a Card object with the given rank and suit."""
        if (not rank in Card.RANKS or not suit in Card.SUITS ):
            print ("Not a legal card specification.")
            return
        self._rank = rank
        self._suit = suit

    def getRank(self):
        """Return my rank."""
        return self._rank

    def getSuit(self):
        """Return my suit."""
        return self._suit
```

Poker: Card Class

```
# This is the continuation of the Card class.

def __str__(self):
    """Return a string that is the print representation
    of this Card's value."""
```

What might this look like?

Poker: Card Class

```
# This is the continuation of the Card class.

def __str__(self):
    """Return a string that is the print representation
    of this Card's value."""
```

What might this look like?

```
    # Create a dictionary for the special cases.
    translate = { 1:'Ace', 11:'Jack', 12:'Queen', 13:'King' }
    r = self._rank
    # See if r is a special case (printwise).
    if r in [1, 11, 12, 13]:
        myrank = translate[r]
    else:
        myrank = str( r )
    return myrank + ' of ' + self._suit
```

Poker: Card Class

```
>>> from Card import *
>>> print (Card.RANKS)
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)
>>> print (Card.SUITS)
('Spades', 'Diamonds', 'Hearts', 'Clubs')
>>> c1 = Card(2, "Spades")
>>> c1.getRank()
2
>>> c1.getSuit()
'Spades'
>>> c1
<Card.Card object at 0xb763d4ec>
>>> print(c1)
2 of Spades
>>> c2 = Card(12, 'Hearts')
>>> print(c2)
Queen of Hearts
>>> (c1 < c2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Card() < Card()
```

Something Cool

Suppose we add the following function to our `Card` class.

```
def __lt__(self, other):
    return ( self._rank < other.getRank() )
```

This assumes that `other` is another Card object; if we're being very careful, we could check that in our code.

Now we can compare two cards using a convenient notation:

```
>>> from Card import *
>>> c1 = Card(2, "Spades")
>>> c2 = Card(5, "Diamonds")
>>> c1 < c2
True
>>> c2 < c1
False
>>> c1 > c2
False
```

Notice that we're comparing cards only according to rank, and Ace is less than 2. Think how you'd define a more robust test.

Comparing Class Instances

You can use all of the standard relational operators assuming you have defined `__lt__` and `__le__` so Python can figure out what you mean. You can always do equality comparison $X == Y$, but it will be an "is" test unless you define `__eq__`.

You can also define `__gt__` and `__ge__` but be careful that your definitions form a consistent collection.

You *shouldn't* define all of those functions, just enough to get it to work. That is, if you have `__lt__`, you don't need `__ge__` because that's just the negation.

Aside: Equality Comparisons

`(X == Y)` tests for structural equivalence of values. `(X is Y)` tests whether two objects are in fact the same object. Sometimes those are not the same thing

```
>>> x = [1, 2, 3]
>>> y = x
>>> z = [1, 2, 3]
>>> x == y
True
>>> x is y
True
>>> x == z
True
>>> x is z
False
```

Abstraction

Notice that we defined the Card class abstractly. There's nothing about it that indicates we're going to be playing Poker. *That's why it's good to start at the bottom!*

It would work as well for blackjack or canasta. It wouldn't work for Uno or another game using a specialty deck. What would you do for such cases?

Now the *interface* to the Card class is the methods: getSuit(), getRank(), print, and the relational comparisons. *Any other way of manipulating a Card object "violates the abstraction."*

Aside: Those Funny Names

In general, any method name in Python of the form `__xyz__` is probably not intended to be called directly. These are called "magic methods" and have associated functional syntax ("syntactic sugar"):

`__init__`	`ClassName()`
`__len__`	`len()`
`__str__`	`str()`
`__lt__`	`<`
`__eq__`	`==`
`__add__`	`+`

However, you often can call them directly if you want.

```
>> "abc".__add__("def")
'abcdef'
>>> >>> l = [1, 2, 3, 4, 5]
>>> len(l)
5
>>> l.__len__()
5
```

Defining the Deck

Now that we have the Card class, we can build on top of it. Which should we define next, Hand or Deck? Why?

Defining the Deck

Now that we have the Card class, we can build on top of it. Which should we define next, Hand or Deck? Why?

We define Deck class next, because we depend on having a deck to deal a hand. What is the interface for a Deck? What do we want to do with a deck of cards?

Defining the Deck

Now that we have the Card class, we can build on top of it. Which should we define next, Hand or Deck? Why?

We define Deck class next, because we depend on having a deck to deal a hand. What is the interface for a Deck? What do we want to do with a deck of cards?

- Create a new Deck object
- Shuffle
- Show its print string
- Say how many cards are left
- Deal a card

Poker: Deck Class

```
import random
from Card import *

class Deck:
    """Definition of the Deck class.  Each Deck is just a list of
    cards.  It is initialized to contain the full deck of 52 cards."""

    def __init__(self):
        """Return a new deck of cards."""
        self._cards = []
        # For each suit and each rank, generate
        # a card and add it to the deck.
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                c = Card(rank, suit)
                self._cards.append(c)

    def shuffle(self):
        """Shuffle the cards in place."""
        # Note that random.shuffle is a method on Lists,
        # not on Decks.  If we want to shuffle a Deck, we
        # have to define it ourselves.
        random.shuffle(self._cards)
```

Aside:Mixing Levels of Abstraction

The main attribute of a Deck object is a list of Cards. *But don't think you can call List methods on a Deck object.* This is a very common error.

You may be tempted to try to apply List methods to Decks; they're different types. *The only methods available on Decks are the ones you define in the Deck interface.*

```
>>> from Deck import *
>>> d = Deck()
>>> c = Card(2, 'Spades')
>>> d.append(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Deck' object has no attribute 'append'
```

Poker: Deck Class

```
# Continues Deck class.

def deal(self):
    """Remove and return the top card, or None
    if the deck is empty."""
    # The following only works because we've
    # defined __len__ below.
    if len(self) == 0:
        return None
    else:
        # Removes the top card from the deck and
        # returns it.
        return self._cards.pop(0)

def __len__(self):
    """Returns the number of cards left in the deck."""
    return len(self._cards)

def __str__(self):
    result = ""
    for c in self._cards:
        result = result + str(c) + "\n"
    return result
```

Poker: Deck Class

```
>>> from Deck import *
>>> deck = Deck()
>>> len(deck)
52
>>> print (deck)
Ace of Spades
2 of Spades
3 of Spades
...
King of Clubs

>>> deck.shuffle()
>>> print (deck)
Ace of Clubs
5 of Clubs
10 of Spades
...
6 of Diamonds
```

Poker: Deck Class

```
>>> c = deck.deal()
>>> print (c)
Ace of Clubs
>>> c = deck.deal()
>>> print (c)
5 of Clubs

>>> deck.__len__()
50
>>> len(deck)
50
```

Poker: Hand Class

At this point I have Card and Deck defined in their own files, Card.py and Deck.py.

```
import Card
from Deck import *

class Hand:
   """A hand is simply a list of 5 cards, dealt from the deck."""

    # Note that we have to pass in a Deck because we need
    # a place from which to draw the cards.
    def __init__(self, deck):
        """A hand is simply the first five cards in the deck, if there are
        five cards available.  If not, return None."""
        ...

    def __str__(self):
        """The string representation of the Hand, which is just the
        five cards."""
        ...
```

Poker: Hand Class

```
# Continuation of the Hand Class
# Numerous helper functions missing here

def hasPair(self):
    """Return a boolean indicating whether
    the current hand contains a pair."""
    ...

def evaluateHand(self):
    if self.hasRoyalFlush():
        return "Royal Flush"
    elif self.hasStraightFlush():
        return "Straight Flush"
    elif self.hasFourOfAKind():
        return "Four of a kind"
    ...
    elif self.hasPair():
        return "Pair"
    else:
        return "Nothing"
```

The Poker Program

Assume we've already defined the Card, Deck, and Hand classes. Let's look at the top-level driver routine. The idea is to deal *n* hands and evaluate them.

```
"""This is the driver routine for my Poker playing program.
   It is interactive with the user supplying a positive number of
   hands to 'play.'  These are generated and evaluated, with the result
   printed for each hand.  We generate a new deck every time we run
   short on cards in the current deck.
"""

from Deck import *
from Hand import *

def dealSomeHands():
    # Generate the initial deck and shuffle it.
    d = Deck()
    d.shuffle()

    # continues on next slide
```

The Poker Program (2)

```
# Find out from the user how many hands to generate.
while True:
    # Run this while loop until we get a legal (positive
    # integer) answer from the user.
    nStr = input("How many hands should I deal? ")
    if not nStr.isdigit():
        print (" Positive number required. Try again!")
    else:
        n = int( nStr ); break
# Generate n hands.
for i in range( n ):
    # If there are not enough cards left in the deck
    # to deal a hand, generate and shuffle a new deck.
    if ( len(d) < 5 ):
        print("\nDealing a new deck.")
        d = Deck()
        d.shuffle()
    # Generate a new hand, print it, and evaluate it.
    h = Hand(d)
    print("\nHand drawn (", i + 1, "): ", sep="")
    print(h)
    print( h.evaluateHand() )
```

The Poker Program (3)

```
# This is a trick to allow executing this if we're at the
# "top level" and not otherwise.  This keeps this code from
# being run when I just import this class.

if __name__ == '__main__':
    dealSomeHands()
```

Running the Poker Program

```
python Poker.py
How many hands should I deal? 12

Hand drawn (1):
Queen of Clubs
Jack of Hearts
9 of Diamonds
Jack of Clubs
3 of Spades

Pair

...
```

Running the Poker Program (2)

```
...

Hand drawn (10):
5 of Spades
King of Spades
6 of Hearts
7 of Spades
Queen of Spades

Nothing

Dealing a new deck.

Hand drawn (11):
6 of Clubs
9 of Diamonds
2 of Spades
8 of Hearts
Queen of Spades

...
```