

## CS313E: Elements of Software Design

## Classes

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

Last updated: October 9, 2011 at 15:33

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 *data* that it maintains characterizing it's 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.

## 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

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

    <other methods>
```

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

    _circlesCount = 0

    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)

    def printCount():
        print("Created " + str( Circle._circlesCount ) + " circles.")
```

```

>>> 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

```

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).

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

- 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.
- 5 If you want to get really fancy, add some Turtle graphics functionality to display the Circles and Rectangles you define.

## 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

```

## The Constructor

Using the turtle graphics module, when you say:

```
t1 = 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?*

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

First: 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)
- Driver (something to orchestrate the play of the game)

Suppose we want to design a representation in Python of a playing Card.

- What data is associated with a Card? (rank and suit)
- What actions are associated with a Card?
  - What's your rank?
  - What's your suit?
  - How would you like to be printed?

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

*Notice that there are:*

- a *class* definition,
- *instances* of that class.

## 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."""
        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 my value."""
    r = self._rank
    if ( r == 1 ):
        myrank = "Ace"
    elif ( r == 11 ):
        myrank = "Jack"
    elif ( r == 12 ):
        myrank = "Queen"
    elif ( r == 13 ):
        myrank = "King"
    else:
        myrank = str( r )
    return myrank + ' of ' + self._suit
```

```
>>> 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()
```

Suppose we add the following function to our Card class.

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

Now we can compare two cards using a convenient notation:

```
>>> from Card import *
>>> c2 = Card(5, "Diamonds")
>>> c1 = Card(2, "Spades")
>>> 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 about using a more robust test.

## Aside: Those Funny Names

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

|                       |                          |
|-----------------------|--------------------------|
| <code>__init__</code> | <code>ClassName()</code> |
| <code>__len__</code>  | <code>len()</code>       |
| <code>__str__</code>  | <code>str()</code>       |
| <code>__lt__</code>   | <code>&lt;</code>        |
| <code>__eq__</code>   | <code>==</code>          |

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

```
>>> l = [1, 2, 3, 4, 5]
>>> len(l)
5
>>> l.__len__()
5
```

## 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 suit in Card.SUITS:
            for rank in Card.RANKS:
                c = Card(rank, suit)
                self._cards.append(c)

    def shuffle(self):
        """Shuffle the cards."""
        random.shuffle(self._cards)
```

```
# 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:
        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
```

```
>>> 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
```

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

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

```
import Card
from Deck import *

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

    def __init__(self, deck):
        """A hand is simply the first five cards in the deck, if there are
        five cards available."""
        ...

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

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

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"

# Some code here to generate the deck, shuffle it,
# generate k hands, evaluate each one, and print the
# value.
```

```
felix:~/cs313e/python/Poker> python Hand.py
```

```
Hand drawn:
6 of Clubs
Jack of Spades
Ace of Diamonds
2 of Spades
10 of Hearts
Nothing
```

```
Hand drawn:
8 of Hearts
8 of Spades
9 of Clubs
9 of Diamonds
10 of Spades
Two pair
```

```
Hand drawn:
Jack of Clubs
Queen of Hearts
6 of Spades
10 of Diamonds
4 of Hearts
Nothing
```