23. Inheritance and Related OOP* Ideas

Topics:
The classes Card, Deck and Hand
Subclasses
Inheritance
Method Overriding

*OOP = Object Oriented Programming
Will Cover These Topics With a Single Example

It will involve operations with playing cards.

Closely follows Chapter 18 in Think Python
We Are Going to Define Three Classes

class Card:
    """" Represents a single playing card."""

class Deck:
    """" Represents a deck of cards"""

class Hand:
    """" Represents a hand of cards"""
Decks and Hands

Things to do with a deck of cards:
1. Shuffle
2. Sort*
3. Add a card
4. Remove a card

*Maybe sort in different ways

Things to do with a hand of cards:
1. Compare
2. Sort*
3. Add a card
4. Remove a card
Representing a Card

A card has a suit and a rank.

There are 4 possible suits.

There are 13 possible ranks.

Anticipate a class with two attributes
A card has a suit and a rank.
There are 4 possible suits.
There are 13 possible ranks

['Clubs', 'Diamonds', 'Hearts', 'Spades']

['Ace', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine', 'Ten', 'Jack', 'Queen', 'King']
The Class Card

class Card:
    suit_names =
    rank_names =
    def __init__(self,suit,rank):
        
def __str__(self):
        
def __cmp__(self,other):
The Class Card

class Card:

    suit_names =
    rank_names =

    def __init__(self, suit, rank):

    def __str__(self):

    def __cmp__(self, other):

Class Variable
Class Variable
Constructor

For pretty printing
For comparing one card to another
Class Variables

suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']

rank_names = [None, 'Ace', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine', 'Ten', 'Jack', 'Queen', 'King']
Class Variables

suit_names = ['Clubs', 'Diamonds',
              'Hearts', 'Spades']

rank_names = [None, 'Ace', 'Two', 'Three',
              'Four', 'Five', 'Six', 'Seven',
              'Eight', 'Nine', 'Ten', 'Jack',
              'Queen', 'King']

Putting None in the 0th entry makes for more intuitive subscripting: rank_names[7] is ‘Seven’
Suits are “Indexed”

```
suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
```

0 ↔ Clubs
1 ↔ Diamonds
2 ↔ Hearts
3 ↔ Spades

An ordering: Clubs < Diamonds < Hearts < Spades
Class Variables

suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']

rank_names = [None, 'Ace', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine', 'Ten', 'Jack', 'Queen', 'King']
class Card(object):
    suit_names =
    rank_names =

    def __init__(self, suit, rank):
        Constructor

    def __str__(self):

    def __cmp__(self, other):

Let's look at the constructor...
The Constructor: Basic Idea

def __init__(self, suit, rank):
    
    """ suit and rank are ints """
    self.suit = suit
    self.rank = rank

c = Card(2, 8)

Says:
Create a card object that represents the eight-of-hearts
The Constructor With a Convenient no-Argument Option

We'd like

c = Card()
to generate a random Card.

def __init__(self, suit=None, rank=None):
    if suit==None and rank==None:
        self.suit = randi(0,3)  # random suit
        self.rank = randi(1,13)  # random rank
    else:
        self.suit = suit
        self.rank = rank

Using the Optional Argument Idea
The Class Card

class Card(object):
    suit_names =
    rank_names =
    def __init__(self,suit,ranks):
        def __str__(self):
            def __cmp__(self,other):

Let's look at the __str__ method...
def \_\_str\_\_(self):

A special method that “pretty prints” a card when we use print

>>> c = Card(2,13)
>>> print c
    King of Hearts
def __str__(self):

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']

def __str__(self):
    i = self.suit  # suit index
    theSuit = self.suit_names[i]
    j = self.rank  # rank index
    theRank = self.rank_names[j]
    return theRank + ' ' + theSuit

Shows how to access class variables
class Card(object):
    suit_names =
    rank_names =
    def __init__(self, suit, rank):
        def __str__(self):
        def __cmp__(self, other):

    For comparing one card to another

Let's look at the __cmp__ method...
Comparing Cards

What we'd like to do:

```python
>>> C1 = Card(2,13)  # King of Hearts
>>> C2 = Card(0,5)   # Five of Clubs
>>> C1 > C2
True
```

The `__cmp__` method makes this possible
Comparing Cards

What we'd like to do if L is a list of references to Card objects:

```python
L.sort()
for c in L:
    print c
```

Sorting requires comparisons between the things that are being sorted.

The `__cmp__` method makes this possible.
How Do We Compare 2 Cards?

First compare their suits:

- Spades > Hearts > Diamonds > Clubs

If there is a tie, then compare their ranks:

- K > Q > J > 10 > ... > 2 > Ace
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

Returning +1 means that the Card self is greater than the Card other.
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

Returning -1 means that the Card self is less than the Card other.
**How It Works**

```python
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
```

Returning 0 means that the Card `self` is the same as the Card `other`.
for k in range(7):
    YourCard = Card()
    MyCard = Card()
    if YourCard > MyCard:
        Winner = 'You'
    elif MyCard > YourCard:
        Winner = 'Me'
    else:
        Winner = 'Tie'
    print YourCard, MyCard, Winner

Two random cards
Yours is “higher”
Mine is “higher”
If we get here, the two cards are the same.
### Sample Output

<table>
<thead>
<tr>
<th>Your Card</th>
<th>My Card</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Six of Hearts</td>
<td>Six of Spades</td>
<td>Me</td>
</tr>
<tr>
<td>Eight of Spades</td>
<td>Queen of Hearts</td>
<td>You</td>
</tr>
<tr>
<td>Five of Diamonds</td>
<td>Queen of Clubs</td>
<td>You</td>
</tr>
<tr>
<td>Queen of Clubs</td>
<td>Eight of Diamonds</td>
<td>Me</td>
</tr>
<tr>
<td>Two of Clubs</td>
<td>Five of Spades</td>
<td>Me</td>
</tr>
<tr>
<td>Six of Clubs</td>
<td>Four of Spades</td>
<td>Me</td>
</tr>
<tr>
<td>Nine of Clubs</td>
<td>Seven of Spades</td>
<td>Me</td>
</tr>
</tbody>
</table>
This Completes the Discussion of the Class Card

class Card(object):
    suit_names =
    rank_names =
    def __init__(self,suit,rank):
        def __str__(self):
            def __cmp__(self,other):
                }
class Deck(object):
    def __init__(self, suit, rank):
        # Constructor

    def __str__(self):
        # Pretty Print

    def pop_card(self):
        # Remove a card from the deck

    def add_card(self, card):
        # Add a card to the deck

    def shuffle(self):
        # Shuffle the Deck

    def sort(self):
        # Sort the Deck
The Attributes

**DeckOfCards**: list of Card objects

**n**: int

*n* is the number of cards in the deck.

The "top" of the deck is

```
self.DeckOfCards[0]
```

The "bottom" of the deck is

```
self.DeckOfCards[self.n]
```
The Constructor

It will build a length-52 list of cards:

```python
def __init__(self):
    self.n = 52
    self.DeckOfCards = []
    for suit in range(4):
        for rank in range(1,14):
            card = Card(suit,rank)
            self.DeckOfCards.append(card)
```
The Constructor

def __init__(self):
    self.n = 52
    self.DeckOfCards = []
    for suit in range(4):
        for rank in range(1,14):
            card = Card(suit,rank)
            self.DeckOfCards.append(card)

Nested loops are used to cover all possible suits and ranks.
def __init__(self):
    self.n = 52
    self.DeckOfCards = []
    for suit in range(4):
        for rank in range(1,14):
            card = Card(suit,rank)
            self.DeckOfCards.append(card)

The list is built via repeated appending
The Constructor

def __init__(self):
    self.n = 52
    self.DeckOfCards = []
    for suit in range(4):
        for rank in range(1,14):
            card = Card(suit,rank)
            self.DeckOfCards.append(card)

Reminder: one constructor can call another constructor.
Create and Print a Deck

```python
D = Deck()
print D
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ace of Clubs</td>
<td>Two of Clubs</td>
<td>Three of Clubs</td>
<td>Four of Clubs</td>
<td>Five of Clubs</td>
</tr>
<tr>
<td>Six of Clubs</td>
<td>Seven of Clubs</td>
<td>Eight of Clubs</td>
<td>Nine of Clubs</td>
<td>Ten of Clubs</td>
</tr>
<tr>
<td>Jack of Clubs</td>
<td>Queen of Clubs</td>
<td>King of Clubs</td>
<td>Ace of Diamonds</td>
<td>Two of Diamonds</td>
</tr>
</tbody>
</table>

The `__str__` method is invoked and produces 52 lines of output

----------

etc
Randomly Shuffle a Card Deck

def shuffleDeck(self):
    shuffle(self.DeckOfCards)

Makes use of the list method shuffle.
The list function shuffle

```python
>>> a = [1,2,3,4,5,6,7,8,9,10]
>>> shuffle(a)
>>> a
[10, 1, 3, 9, 2, 5, 7, 4, 8, 6]
>>> shuffle(a)
>>> a
[4, 9, 1, 3, 7, 10, 5, 6, 8, 2]
```

This function can be applied to any list. A random permutation. NOT THE PERFECT SHUFFLE
Create, Shuffle, and Print a Deck

D = Deck()
D.shuffle()
print D

Jack of Spades
Four of Hearts
Seven of Diamonds
Three of Spades
Eight of Diamonds
Seven of Clubs
Ace of Hearts
Six of Spades
Ace of Diamonds
Five of Diamonds
Eight of Clubs
Eight of Hearts
Queen of Diamonds
Six of Diamonds
Six of Hearts

etc
Remove a Card

def pop_card(self, Where):
    return self.cards.pop()

Recall how to pop an entry in a list:

```python
>>> x = [10,20,30,40]
>>> x.pop(2)
30
>>> x
[10, 20, 40]
```
Remove a Card

```python
def pop_card(self, Where):
    if Where == 'Top':
        c = self.DeckOfCards.pop(0)
    elif Where == 'Bot':
        c = self.DeckOfCards.pop()
    elif Where == None:
        k = randi(0, self.n - 1)
        c = self.DeckOfCards.pop(k)
    self.n -= 1
    return c
```

Three alternatives. The selected card can come off the top or bottom of the deck or it can be selected randomly.
Add a Card to a Deck

```python
def add_card(self, card):
    self.DeckOfCards.append(card)
```

self.DeckOfCards is a list of cards
Sort a Deck

def sort(self):
    self.DeckOfCards.sort()

This is possible because we defined a

    __cmp__

method in the Card class.
Create and shuffle a deck. Then repeatedly select a card off the top of the Deck, display it, and put it back in the deck at the bottom.
This Completes the Discussion of the Deck Class

class Deck(object):
    def __init__(self, suit, rank):

        def __str__(self):

            def pop_card(self):

                def add_card(self, card):

                    def shuffle(self):

                        def sort(self):
Next Up: The Hand Class

class Hand(Deck):
    def __init__(self, suit, rank):

    def __str__(self):

    def sort(self):
The Hand Class

class Hand(Deck):
    def __init__(self, suit, rank):
        def __str__(self):
            def sort(self):

The Hand Class inherits all the methods from the Deck class.

What Does this Mean?
The Hand Class

class Hand(Deck):
    def __init__(self, suit, rank):
    def __str__(self):
    def sort(self):

Hand Class methods override the methods from the Deck class that have the same name. The Deck class also has methods called __str__ and sort. What does “overriding” mean?
Create a Deck. Shuffle It. Extract 10 Cards. Make a Hand. Print it.

D = Deck()
D.shuffle()
H = Hand('CVL')
for k in range(10):
    c = D.pop_card()
    H.add_card(c)
print H

CVL:
  Ace of Hearts
  Three of Clubs
  Four of Spades
  Four of Diamonds
  Five of Hearts
  Six of Hearts
  Seven of Spades
  Eight of Spades
  Queen of Clubs
  Queen of Spades
Create a Deck. Shuffle It.
Extract 10 Cards. Make a Hand. Print it.

D = Deck()
D.shuffle()
H = Hand('CVL')
for k in range(10):
    c = D.pop_card()
    H.add_card(c)
print H

CVL:
    Queen of Clubs
    Three of Clubs
    Eight of Spades
    Six of Hearts
    Queen of Spades
    Ace of Hearts
    Five of Hearts
    Four of Spades
    Seven of Spades
    Four of Diamonds

The `add_card` method is inherited from the Deck class
Create a Deck. Shuffle It. Extract 10 Cards. Make a Hand. Print it.

```
D = Deck()
D.shuffle()
H = Hand('CVL')
for k in range(10):
    C = D.pop_card()
    H.add_card(C)
print H
```

```
CVL:
Queen of Clubs
Three of Clubs
Eight of Spades
Six of Hearts
Queen of Spades
Ace of Hearts
Five of Hearts
Four of Spades
Seven of Spades
Four of Diamonds
```

The print function from the Hand class overrides the print function from the Deck Class.
Inheritance Chit Chat

The existing class **Deck** is the **parent**

The new class **Hand** is the **child**

**Hand** is a **subclass** of **Deck**

Inheritance is a very important mechanism when it comes to maintaining and updating software.
Decks and Hands

Things to do with a deck of cards:
1. Shuffle
2. Sort*
3. Add a card
4. Remove a card

Things to do with a hand of cards:
1. Compare
2. Sort*
3. Add a card
4. Remove a card

*Maybe sort in different ways
Another Example of Overriding

As written, when a Deck is sorted, it is sorted by suit first and then by rank.

To be different, when a Hand is sorted, let’s sort by rank first and then by suit.

Seven of Clubs  
Ten of Diamonds  
Six of Hearts  
Eight of Hearts  
Ace of Spades  

VS

Ace of Spades  
Six of Hearts  
Seven of Clubs  
Eight of Hearts  
Ten of Diamonds
The sort Method in the Hand Class

def sort(self):
    self.cards.sort(MyCompare)

def MyCompare(H1, H2):
    if H1.rank > H2.rank:
        return 1
    if H1.rank < H2.rank:
        return -1
    if H1.suit > H2.suit:
        return 1
    if H1.suit < H2.suit:
        return -1
    return 0

Sorts by rank first, then suit.

This sort Method overrides the sort method in Deck, which sorts by suit first, then rank.
A random 10-card deck $D$

Three of Hearts  
Four of Spades  
Seven of Diamonds  
Five of Spades  
Queen of Diamonds  
Four of Hearts  
Ten of Diamonds  
Queen of Hearts  
Two of Spades  
Ace of Clubs

Since $D$ is a Deck object, Python invokes the sort method defined in the Deck class.

$D$.sort()

Ace of Clubs  
Seven of Diamonds  
Ten of Diamonds  
Queen of Diamonds  
Three of Hearts  
Four of Hearts  
Queen of Hearts  
Two of Spades  
Four of Spades  
Five of Spades

Sorts by suit first, then rank.
A random 10-card Hand H

Three of Hearts
Four of Spades
Seven of Diamonds
Five of Spades
Queen of Diamonds
Four of Hearts
Ten of Diamonds
Queen of Hearts
Two of Spades
Ace of Clubs

H.sort()

Ace of Clubs
Two of Spades
Three of Hearts
Four of Hearts
Four of Spades
Five of Spades
Seven of Diamonds
Ten of Diamonds
Queen of Diamonds
Queen of Hearts

Sorts by rank first, then suit.

Since H is a Hand object, Python invokes the sort method defined in the Hand class.
A Couple of Examples
Dealing 4 Bridge Hands

```python
D = Deck(); D.shuffle()
L = []
for k in range(4):
    L.append(Hand(str(k)))
for k in range(52):
    L[k%4].add_card(D.pop_card())
for k in range(4):
    print L[k].sort()
```

Set up and shuffle the deck
Dealing 4 Bridge Hands

```python
D = Deck(); D.shuffle()
L = []
for k in range(4):
    L.append(Hand(str(k))
for k in range(52):
    L[k%4].add_card(D.pop_card())
for k in range(4):
    print L[k].sort()
```

Set Up a length-4 list of Hands
Dealing 4 Bridge Hands

```python
D = Deck(); D.shuffle()
L = []
for k in range(4):
    L.append(Hand(str(k))
for k in range(52):
    L[k%4].add_card( D.pop_card() )
for k in range(4):
    print L[k].sort()
```
Dealing 4 Bridge Hands

D = Deck(); D.shuffle()
L = []
for k in range(4):
    L.append(Hand(str(k))
for k in range(52):
    L[k%4].add_card(D.pop_card_card())
for k in range(4):
    print L[k].sort()

Add to every 4th hand
Dealing 4 Bridge Hands

```python
D = Deck(); D.shuffle()
L = []
for k in range(4):
    L.append(Hand(str(k))
for k in range(52):
    L[k%4].add_card(D.pop_card())
for k in range(4):
    print L[k].sort()
```

Sort and print each Hand
Next Example from Poker
Probability of a Full House

Core Problem: When does a 5-card hand consist of two of one rank and three of another?

- Seven of Spades
- Seven of Diamonds
- Ten of Clubs
- Ten of Spades
- Ten of Diamonds

- Four of Spades
- Four of Diamonds
- Jack of Hearts
- Jack of Clubs
- Jack of Spades
H.sort()

r = []

for c in H.cards:
    r.append(c.rank)


if B1 or B2:
    print 'Full House'
Is a Hand $H$ a Full House?

```python
H.sort()

r = []
for c in H.cards:
    r.append(c.rank)


if B1 or B2:
    print 'Full House'
```

Sort the Hand by rank
# Three Hands

### Yes:

- Seven of Spades
- Seven of Diamonds
- Seven of Clubs
- Ten of Spades
- Ten of Diamonds

### Yes:

- Four of Spades
- Four of Diamonds
- Jack of Hearts
- Jack of Clubs
- Jack of Spades

### No:

- Four of Spades
- Four of Diamonds
- Five of Hearts
- Jack of Clubs
- Jack of Spades
Is a Hand H a Full House?

H.sort()

r = []

for c in H.cards:
    r.append(c.rank)


If B1 or B2:
    print 'Full House'

Form a list of the ranks
Is a Hand H a Full House?

```python
H.sort()
r = []
for c in H.cards:
    r.append(c.rank)
if B1 or B2:
    print 'Full House'
```

Boolean Business