25. 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

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

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

Representing a Card
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):

Class Variables

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

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 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'
The Class Card

class Card(object):
    suit_names =
    rank_names =
    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank
    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

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

The Class Card

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

Let's look at the __str__ method...

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
The Class Card

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:

>>> 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:

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

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

How Do We Compare 2 Cards?

How It Works

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.

How It Works

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.

Example

```python
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.

This Completes the Discussion of the Class Card

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

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>

Next Up: The Class Deck

```python
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)
```

Nested loops are used to cover all possible suits and ranks.

The Constructor

The list is built via repeated appending.

Reminder: one constructor can call another constructor.

Create and Print a Deck

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

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

Randomly Shuffle a Card Deck

```python
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]
```

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

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

Recall how to pop an entry in a list:

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

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

```python
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.

```python
D = Deck()
D.shuffle()
for k in range(5):
c = D.pop_card('Top')
    print c
D.add_card(c)
```

This Completes the Discussion of the Deck Class

```python
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

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

The Hand Class

```python
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?

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.

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

<table>
<thead>
<tr>
<th>CVL:</th>
<th>Ace of Hearts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Three of Clubs</td>
</tr>
<tr>
<td></td>
<td>Four of Spades</td>
</tr>
<tr>
<td></td>
<td>Five of Hearts</td>
</tr>
<tr>
<td></td>
<td>Six of Hearts</td>
</tr>
<tr>
<td></td>
<td>Seven of Spades</td>
</tr>
<tr>
<td></td>
<td>Eight of Spades</td>
</tr>
<tr>
<td></td>
<td>Queen of Clubs</td>
</tr>
<tr>
<td></td>
<td>Queen of Spades</td>
</tr>
</tbody>
</table>

What Does this Mean?
Create a Deck. Shuffle It. 
Extract 10 Cards. Make a Hand. 
Print it.

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

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.

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

vs
```

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

The sort Method in the Hand Class

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

```python
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
```

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

D.sort()

Sorts by suit first, then rank.

A random 10-card Hand H

H.sort()

Sorts by rank first, then suit.

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())
for k in range(4):
    print L[k].sort()

Set up and shuffle the deck

Set Up a length-4 list of Hands

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())
for k in range(4):
    print L[k].sort()

Get a card from the Deck
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())
for k in range(4):
    print L[k].sort()
```

Add to every 4th hand

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())
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?

Four of Spades
Four of Diamonds
Jack 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'
```

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'
```

Sort the Hand by rank
Three Hands

Yes:
- Seven of Spades
- Seven of Diamonds
- Ten of Spades
- Ten of Diamonds

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'

Boolean Business