27. Important Object-Oriented Programming Ideas

Topics:
- Class Variables
- Inheritance
- Method Overriding
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
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 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']

An ordering: Clubs < Diamonds < Hearts < Spade

0 ➔ Clubs
1 ➔ Diamonds
2 ➔ Hearts
3 ➔ 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']

We will shortly see how this data can be accessed.
class Card:
    suit_names =
    rank_names =
    def __init__(self, suit, rank):
        Constructor

    def __str__(self):

    def __cmp__(self, other):
The Constructor: Basic Idea

```python
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:
        self.suit = randi(0,3)  # random suit
        self.rank = randi(1,13) # random rank
    else:
        self.suit = suit
        self.rank = rank
class Card:
    suit_names =
    rank_names =
    def __init__(self,suit,rank):
        def __str__(self):
            def __cmp__(self,other):

For pretty printing
def __str__(self)

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

```python
>>> c = Card(2,13)
>>> print c
    King of Hearts
```
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 a class variable
class Card:
    suit_names =
    rank_names =
    def __init__(self, suit, rank):
        
    def __str__(self):
        
    def __cmp__(self, other):
        For comparing one card to another
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:

```
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, 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

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

The Card self is not greater than 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

The Card self is the same as Card other
class Card:
    suit_names =
    rank_names =
    def __init__(self, suit, rank):

    def __str__(self):

    def __cmp__(self, other):
class Deck:
    def __init__(self, suit, rank):
    def __str__(self):
    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 Constructor

It will build a length-52 list of cards:

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

It will build a length-52 list of cards:

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

Fact 1. This class has one attribute: a list of cards
The Constructor

It will build a length-52 list of cards:

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

Fact 2. Nested loops are used to cover all possible suits and ranks.
The Constructor

It will build a length-52 list of cards:

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

Fact 3. The list is built via repeated appending
The Constructor

It will build a length-52 list of cards:

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

Fact 4. Our first example of a constructor that calls another constructor
Create and Print a Deck

D = Deck()
print D

The \_\_str\_\_ method is invoked and produces
52 lines of output ---------------

Ace of Clubs
Two of Clubs
Three of Clubs
Four of Clubs
Five of Clubs
Six of Clubs
Seven of Clubs
Eight of Clubs
Nine of Clubs
Ten of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Diamonds
Two of Diamonds

etc
def shuffle(self):
    shuffle(self.cards)
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

The `__str__` method is invoked and produces 52 lines of output
def pop_card(self):
    return self.cards.pop()

Recall how to pop the last value in a list:

```python
>>> x = [10, 20, 30, 40]
>>> x.pop()
40
>>> x
[10, 20, 30]
```
Create and Shuffle a Deck. Then remove 47 cards and Print

D = Deck()
D.shuffle()
for k in range(47):
    D.pop_card()
print D

Nine of Hearts
Ten of Spades
King of Diamonds
Queen of Diamonds
Two of Spades
Add a Card to a Deck

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

`self.cards` is a list of cards
Sort a Deck

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

This is possible because we defined a __cmp__ method in the Card class.
Combine a Pair of Card Decks, Sort the Result, and Print

D1 = Deck()
D2 = Deck()
for k in range(52):
    C = D1.pop_card()
    D2.add_card(C)
D2.sort()
print D2

Pop a card off of one deck and add it to the other.
This Completes the Discussion of the Deck Class

class Deck:
    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):

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?
class Hand(Deck):

def __init__(self, suit, rank):

def __str__(self):

def sort(self):

The Hand Class methods override the methods from the Deck class that have the same name.

What Does this 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

The add_card method is inherited from the Deck class

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
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

The print function from the Hand class overrides the Deck print function.

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 existing class Deck is the parent

The new class Hand is the child

A Hand is a kind of Deck

A crucial 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
   
*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
A Better 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

It sorts on the rank attribute, not the suit attribute as in the Deck class

```python
def sort(self):
    self.cards.sort(key=Card.get_rank)
```
A Couple of Examples

More in Lab 11
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())
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())

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

Form a list of the ranks

```python
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'
```

Boolean Business