Lecture 7-3

Classes and Inheritance

Week 7 Friday

Miles Chen, PhD

Taken directly from Chapter 18 of Think Python by Allen B Downey

Inheritance

Inheritance is the ability to define a new class that is a modified version of an existing class.

In this chapter, we will demonstrate inheritance using classes to represent playing cards, decks of cards, and poker hands.

https://en.wikipedia.org/wiki/List_of_poker_hands

Card objects

There are 52 cards in a deck.

There are 4 suits: Spades, Hearts, Diamonds, and Clubs (descending order in bridge).

Each suit has 13 ranks: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King.

If we define a new object to represent a playing card, the attributes will be `rank` and `suit`.

How we should store the attributes is not obvious. If we use strings, it will not be easy to compare cards to see which has a higher rank or suit.

Another option is to use integers to *encode* the ranks and suits. For example, we use the following for the suits:

- spades: 3
- hearts: 2
- diamonds: 1
- clubs: 0

For the ranks, we'll use the numeric value, with Jacks: 11, Queens: 12, Kings: 13

In [1]:

```
class Card:
    """Represents a standard playing card."""

    def __init__(self, suit = 0, rank = 2):
        self.suit = suit
        self.rank = rank
```

In [1]:

```
class Card:
    """Represents a standard playing card."""

    def __init__(self, suit = 0, rank = 2):
        self.suit = suit
        self.rank = rank
```

The default card would be a two of Clubs.

In [1]:

```
class Card:
    """Represents a standard playing card."""

    def __init__(self, suit = 0, rank = 2):
        self.suit = suit
        self.rank = rank
```

The default card would be a two of Clubs.

In [2]:

```
queen_of_diamonds = Card(1, 12)
```

We also want the card objects to be read easily by humans.

So we need a way to go from the integer codes back to suits and ranks.

We'll do this by creating a list of names and then defining the `__str__` method to represent the card.

In [3]:

```
class Card:
    def __init__(self, suit = 0, rank = 2):
        self.suit = suit
        self.rank = rank

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return "{:s} of {:s}".format(Card.rank_names[self.rank],
                                     Card.suit_names[self.suit])
```

In [3]:

```python
class Card:
    def __init__(self, suit = 0, rank = 2):
        self.suit = suit
        self.rank = rank

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return "{:s} of {:s}".format(Card.rank_names[self.rank],
                                     Card.suit_names[self.suit])
```

Variables like `suit_names` and `rank_names`, which are defined inside a class but outside of any method, are called **class attributes** because they are associated with the class object `Card`. Note that in their definition, `suit_names` and `rank_names` are not preceeded by `self`.

This term distinguishes them from variables like `suit` and `rank`, which are called **instance attributes** because they are associated with a particular instance. These attributes are preceeded by `self`

If we create multiple cards, every card has its own `suit` and `rank` but there is only copy of `suit_names` and `rank_names`.

The first value (index zero) in `rank_names` is `None` because there is no card with a rank zero.

In [4]:

```
card1 = Card(2, 11)
print(card1)
```

```
Jack of Hearts
```

In [4]:

```
card1 = Card(2, 11)
print(card1)
```

```
Jack of Hearts
```

In [5]:

```
card1 = Card(1, 1)
print(card1)
```

```
Ace of Diamonds
```

In [4]:

```
card1 = Card(2, 11)
print(card1)
```

```
Jack of Hearts
```

In [5]:

```
card1 = Card(1, 1)
print(card1)
```

```
Ace of Diamonds
```

In [6]:

```
card1 = Card(3, 6)
print(card1)
```

```
6 of Spades
```

Comparing Cards

For built-in types, we can use relational operators like >, <, == that compare values and determine when one is greater than, less than, or equal to another.

For our own defined classes, we can use a special method `__lt__` which stands for 'less than'. Similarly, there is another dunder method `__eq__` that can be used to test equality.

We'll arbitrarily choose to rank suits as more important, so all of the Spades will outrank all of the Diamonds.

To perform the comparison, we'll use tuple comparison

In [7]:

```
class Card:
    def __init__(self, suit = 0, rank = 2):
        self.suit = suit
        self.rank = rank

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return "{:s} of {:s}".format(Card.rank_names[self.rank],
                                     Card.suit_names[self.suit])

    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2

    def __eq__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 == t2
```

In [8]:

```
card1 = Card(1, 12)
print(card1)
```

```
Queen of Diamonds
```

In [8]:

```
card1 = Card(1, 12)
print(card1)
```

```
Queen of Diamonds
```

In [9]:

```
card2 = Card(2, 11)
print(card2)
```

```
Jack of Hearts
```

In [8]:

```python
card1 = Card(1, 12)
print(card1)
```

```
Queen of Diamonds
```

In [9]:

```python
card2 = Card(2, 11)
print(card2)
```

```
Jack of Hearts
```

In [10]:

```python
# diamonds are ranked lower than Hearts
card1 < card2
```

Out[10]:

```
True
```

In [8]:

```
card1 = Card(1, 12)
print(card1)
```

```
Queen of Diamonds
```

In [9]:

```
card2 = Card(2, 11)
print(card2)
```

```
Jack of Hearts
```

In [10]:

```
# diamonds are ranked lower than Hearts
card1 < card2
```

Out[10]:

```
True
```

In [11]:

```
card3 = Card(2, 12)
print(card3)
```

```
Queen of Hearts
```

In [8]:
```
card1 = Card(1, 12)
print(card1)
```

```
Queen of Diamonds
```

In [9]:
```
card2 = Card(2, 11)
print(card2)
```

```
Jack of Hearts
```

In [10]:
```
# diamonds are ranked lower than Hearts
card1 < card2
```

Out[10]:
```
True
```

In [11]:
```
card3 = Card(2, 12)
print(card3)
```

```
Queen of Hearts
```

In [12]:
```
card3 < card2
```

Out[12]:
```
False
```

In [13]:

```
card4 = Card(1, 12)
```

In [13]:

```
card4 = Card(1, 12)
```

In [14]:

```
print(card1)
```

```
Queen of Diamonds
```

In [13]:

```
card4 = Card(1, 12)
```

In [14]:

```
print(card1)
```

```
Queen of Diamonds
```

In [15]:

```
print(card4)
```

```
Queen of Diamonds
```

In [13]:
```
card4 = Card(1, 12)
```

In [14]:
```
print(card1)
```

```
Queen of Diamonds
```

In [15]:
```
print(card4)
```

```
Queen of Diamonds
```

In [16]:
```
card1 == card4
```

Out[16]:
```
True
```

In [13]:
```
card4 = Card(1, 12)
```

In [14]:
```
print(card1)
```

```
Queen of Diamonds
```

In [15]:
```
print(card4)
```

```
Queen of Diamonds
```

In [16]:
```
card1 == card4
```

Out[16]:
```
True
```

In [17]:
```
print(card3)
```

```
Queen of Hearts
```

In [13]:
```python
card4 = Card(1, 12)
```

In [14]:
```python
print(card1)
```

```
Queen of Diamonds
```

In [15]:
```python
print(card4)
```

```
Queen of Diamonds
```

In [16]:
```python
card1 == card4
```

Out[16]:
```
True
```

In [17]:
```python
print(card3)
```

```
Queen of Hearts
```

In [18]:
```python
card1 == card3
```

Out[18]:
```
False
```

Building a Deck

Now that we have Cards, the next step is to define Decks. Since a deck is made up of cards, it is natural for each Deck to contain a list of cards as an attribute.

The following is a class definition for Deck. The `__init__` method creates the attribute cards and generates the standard set of fifty-two cards.

The `__str__` method builds a list of the string representation of cards and uses the string method `join`

In [19]:

```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

In [19]:

```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

In [20]:

```
deck = Deck()
```

In [21]:

```python
print(deck)
```

```
Ace of Clubs
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Clubs
7 of Clubs
8 of Clubs
9 of Clubs
10 of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Diamonds
2 of Diamonds
3 of Diamonds
4 of Diamonds
5 of Diamonds
6 of Diamonds
7 of Diamonds
8 of Diamonds
9 of Diamonds
10 of Diamonds
Jack of Diamonds
Queen of Diamonds
King of Diamonds
Ace of Hearts
2 of Hearts
3 of Hearts
4 of Hearts
```

5 of Hearts
6 of Hearts
7 of Hearts
8 of Hearts
9 of Hearts
10 of Hearts
Jack of Hearts
Queen of Hearts
King of Hearts
Ace of Spades
2 of Spades
3 of Spades
4 of Spades
5 of Spades
6 of Spades
7 of Spades
8 of Spades
9 of Spades
10 of Spades
Jack of Spades
Queen of Spades
King of Spades

Add, remove, shuffle, and sort

To deal cards, we can create a method that removes a card from the deck and returns it. We can use define the following method inside the class:

```
def pop_card(self):
    return self.cards.pop()
```

To add a card, we can use the list method `append`

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

We can also add a `shuffle` method to mix the cards

```
def shuffle(self):
    random.shuffle(self.cards)
```

Because we have defined the method `__lt__` for the cards, we can perform `sort` operations to sort the cards

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

In [22]:

```
import random

class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)

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

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

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

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

In [23]:

```
deck = Deck()
```

In [24]:

```
len(deck.cards)
```

Out[24]:

```
52
```

In [23]:
```
deck = Deck()
```

In [24]:
```
len(deck.cards)
```

Out[24]:
```
52
```

In [25]:
```
print(deck)
```

```
Ace of Clubs
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Clubs
7 of Clubs
8 of Clubs
9 of Clubs
10 of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Diamonds
2 of Diamonds
3 of Diamonds
4 of Diamonds
5 of Diamonds
6 of Diamonds
7 of Diamonds
8 of Diamonds
9 of Diamonds
```

```
10 of Diamonds
Jack of Diamonds
Queen of Diamonds
King of Diamonds
Ace of Hearts
2 of Hearts
3 of Hearts
4 of Hearts
5 of Hearts
6 of Hearts
7 of Hearts
8 of Hearts
9 of Hearts
10 of Hearts
Jack of Hearts
Queen of Hearts
King of Hearts
Ace of Spades
2 of Spades
3 of Spades
4 of Spades
5 of Spades
6 of Spades
7 of Spades
8 of Spades
9 of Spades
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

In [26]:

```
deck.shuffle()
```

In [26]:

```
deck.shuffle()
```

In [27]:

```
print(str(deck)[:100])
print("...")
print(str(deck)[-100:])
```

```
King of Spades
8 of Diamonds
Queen of Spades
2 of Hearts
2 of Diamonds
6 of Hearts
King of Clubs
7 o
...
 of Spades
Ace of Spades
5 of Diamonds
10 of Clubs
9 of Spades
2 of Spades
10 of Spades
Ace of Clubs
```

In [28]:

```
print(deck.pop_card())
```

```
Ace of Clubs
```

In [28]:

```
print(deck.pop_card())
```

```
Ace of Clubs
```

In [29]:

```
print(deck.pop_card())
```

```
10 of Spades
```

In [28]:

```
print(deck.pop_card())
```

```
Ace of Clubs
```

In [29]:

```
print(deck.pop_card())
```

```
10 of Spades
```

In [30]:

```
len(deck.cards)
```

Out[30]:

```
50
```

In [28]:

```
print(deck.pop_card())
```

```
Ace of Clubs
```

In [29]:

```
print(deck.pop_card())
```

```
10 of Spades
```

In [30]:

```
len(deck.cards)
```

Out[30]:

```
50
```

In [31]:

```
print(str(deck)[:100])
print("...")
print(str(deck)[-100:])
```

```
King of Spades
8 of Diamonds
Queen of Spades
2 of Hearts
2 of Diamonds
6 of Hearts
King of Clubs
7 o
...
of Clubs
Ace of Diamonds
4 of Spades
Ace of Spades
```

```
5 of Diamonds
10 of Clubs
9 of Spades
2 of Spades
```

Inheritance

Inheritance is the ability to define a new class that is a modified version of an existing class.

For example, let's say we want a new class to represent a "hand", that is, the cards held by one player.

A hand is similar to a deck: both are made up of a collection of cards, and both require operations like adding and removing cards.

A hand is also different from a deck; there are operations we want for hands that don’t make sense for a deck. For example, in poker we might compare two hands to see which one wins. In bridge, we might compute a score for a hand in order to make a bid.

This relationship between classes—similar, but different—lends itself to inheritance.

To define a new class that inherits from an existing class, you put the name of the existing class in parentheses.

Inheritance

Inheritance is the ability to define a new class that is a modified version of an existing class.

For example, let's say we want a new class to represent a "hand", that is, the cards held by one player.

A hand is similar to a deck: both are made up of a collection of cards, and both require operations like adding and removing cards.

A hand is also different from a deck; there are operations we want for hands that don’t make sense for a deck. For example, in poker we might compare two hands to see which one wins. In bridge, we might compute a score for a hand in order to make a bid.

This relationship between classes—similar, but different—lends itself to inheritance.

To define a new class that inherits from an existing class, you put the name of the existing class in parentheses.

In [32]:

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

This definition indicates that Hand inherits from Deck; that means we can use methods like `pop_card` and `add_card` for Hands as well as Decks.

When a new class inherits from an existing one, the existing one is called the **parent** and the new class is called the **child**.

If we have nothing else defined, then `Hand` inherits `__init__` from `Deck`, which is not what we want.

If we provide an `init` method in the `Hand` class, it overrides the one from `Deck`.

This definition indicates that Hand inherits from Deck; that means we can use methods like `pop_card` and `add_card` for Hands as well as Decks.

When a new class inherits from an existing one, the existing one is called the **parent** and the new class is called the **child**.

If we have nothing else defined, then `Hand` inherits `__init__` from `Deck`, which is not what we want.

If we provide an `init` method in the `Hand` class, it overrides the one from `Deck`.

In [33]:

```
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

When you create a Hand, Python invokes this init method, not the one in Deck.

When you create a Hand, Python invokes this init method, not the one in Deck.

In [34]:

```
hand = Hand('new hand')
hand.cards
```

Out[34]:

```
[]
```

When you create a Hand, Python invokes this init method, not the one in Deck.

In [34]:

```
hand = Hand('new hand')
hand.cards
```

Out[34]:

```
[]
```

In [35]:

```
hand.label
```

Out[35]:

```
'new hand'
```

When you create a Hand, Python invokes this init method, not the one in Deck.

In [34]:
```
hand = Hand('new hand')
hand.cards
```

Out[34]:
```
[]
```

In [35]:
```
hand.label
```

Out[35]:
```
'new hand'
```

In [36]:
```
deck = Deck()
card = deck.pop_card()
hand.add_card(card)
```

When you create a Hand, Python invokes this init method, not the one in Deck.

In [34]:
```
hand = Hand('new hand')
hand.cards
```

Out[34]:
```
[]
```

In [35]:
```
hand.label
```

Out[35]:
```
'new hand'
```

In [36]:
```
deck = Deck()
card = deck.pop_card()
hand.add_card(card)
```

In [37]:
```
print(hand)
```

```
King of Spades
```

We can add these steps into a method called `move_cards` into the `Deck` class.

```
def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

`move_cards` takes two arguments, a Hand object and the number of cards to deal. It modifies both `self` and `hand`, and returns `None`.

In some games, cards are moved from one hand to another, or from a hand back to the deck. You can use `move_cards` for any of these operations: `self` can be either a Deck or a Hand, and `hand`, despite the name, can also be a Deck.

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it. Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them.

In some cases, the inheritance structure reflects the natural structure of the problem, which makes the design easier to understand.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be spread across several modules.

In [38]:

```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)

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

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

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

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

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

In [39]:

```
deck = Deck()
hand = Hand('new hand')
```

In [39]:

```
deck = Deck()
hand = Hand('new hand')
```

In [40]:

```
deck.move_cards(hand, 5)
```

In [39]:

```
deck = Deck()
hand = Hand('new hand')
```

In [40]:

```
deck.move_cards(hand, 5)
```

In [41]:

```
print(hand)
```

```
King of Spades
Queen of Spades
Jack of Spades
10 of Spades
9 of Spades
```

In [42]:

```
deck = Deck()
hand = Hand('new hand')
```

In [42]:

```
deck = Deck()
hand = Hand('new hand')
```

In [43]:

```
deck.shuffle()
print(str(deck)[-100:])
deck.move_cards(hand, 5)
```

```
arts
10 of Hearts
Jack of Clubs
10 of Diamonds
Queen of Spades
4 of Spades
3 of Diamonds
9 of Spades
```

In [42]:

```python
deck = Deck()
hand = Hand('new hand')
```

In [43]:

```python
deck.shuffle()
print(str(deck)[-100:])
deck.move_cards(hand, 5)
```

```
arts
10 of Hearts
Jack of Clubs
10 of Diamonds
Queen of Spades
4 of Spades
3 of Diamonds
9 of Spades
```

In [44]:

```python
print(hand)
```

```
9 of Spades
3 of Diamonds
4 of Spades
Queen of Spades
10 of Diamonds
```

In [42]:

```
deck = Deck()
hand = Hand('new hand')
```

In [43]:

```
deck.shuffle()
print(str(deck)[-100:])
deck.move_cards(hand, 5)
```

```
arts
10 of Hearts
Jack of Clubs
10 of Diamonds
Queen of Spades
4 of Spades
3 of Diamonds
9 of Spades
```

In [44]:

```
print(hand)
```

```
9 of Spades
3 of Diamonds
4 of Spades
Queen of Spades
10 of Diamonds
```

In [45]:

```
print(str(deck)[-30:])
```

```
rts
10 of Hearts
Jack of Clubs
```

Class relationships

Class diagrams are useful to represent the relationships that exist between classes.

There are different kids of relationships between classes:

- Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called **HAS-A**, as in "a Rechtangle has a point"
- One class might inherit from another. This relationship is called **IS-A**, as in "a Hand is a kind of a Deck"
- One class might depend on another in the sense that objects in one class take objects in the second class as parameters, or use objects in the second class as part of a computation. This kind of relationship is called a **dependency**.

Class relationships

Class diagrams are useful to represent the relationships that exist between classes.

There are different kids of relationships between classes:

- Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called **HAS-A**, as in "a Rechtangle has a point"
- One class might inherit from another. This relationship is called **IS-A**, as in "a Hand is a kind of a Deck"
- One class might depend on another in the sense that objects in one class take objects in the second class as parameters, or use objects in the second class as part of a computation. This kind of relationship is called a **dependency**.

In [46]:

```python
vars(hand)
```

Out[46]:

```
{'cards': [<__main__.Card at 0x1f385426ac8>,
  <__main__.Card at 0x1f3853f0f08>,
  <__main__.Card at 0x1f3854264c8>,
  <__main__.Card at 0x1f385426cc8>,
  <__main__.Card at 0x1f385405fc8>],
 'label': 'new hand'}
```

Class relationships

Class diagrams are useful to represent the relationships that exist between classes.

There are different kids of relationships between classes:

- Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called **HAS-A**, as in "a Rechtangle has a point"
- One class might inherit from another. This relationship is called **IS-A**, as in "a Hand is a kind of a Deck"
- One class might depend on another in the sense that objects in one class take objects in the second class as parameters, or use objects in the second class as part of a computation. This kind of relationship is called a **dependency**.

In [46]:

```
vars(hand)
```

Out[46]:

```
{'cards': [<__main__.Card at 0x1f385426ac8>,
  <__main__.Card at 0x1f3853f0f08>,
  <__main__.Card at 0x1f3854264c8>,
  <__main__.Card at 0x1f385426cc8>,
  <__main__.Card at 0x1f385405fc8>],
 'label': 'new hand'}
```

Here's a design suggestion: when you override a method, the interface of the new method should be the same as the old. It should take the same parameters, return the same type, and obey the same preconditions and postconditions. If you follow this rule, you will find that any function designed to work with an instance of a parent class, like a Deck, will also work with instances of child classes like a Hand and PokerHand.