Chapter 17 Inheritance

A. Downey, *Think Python: How to Think Like a Computer Science,* 3rd ed., O'Reilly, 2024.

https://allendowney.github.io/ThinkPython/

Contents

1. Presenting Cards

- The language feature most often associated with object-oriented programming is **inheritance**. Inheritance is the ability to define a new class that is a modified version of an existing class.
- Presenting Cards
  - There are 52 playing cards in a standard deck—each of them belongs to one of four suits and one of thirteen ranks.
    - The suits are Spades, Hearts, Diamonds, and Clubs.
    - The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King.
  - Depending on which game you are playing, an Ace can be higher than King or lower than 2.

1. Presenting Cards

- To represent a playing card, it is obvious what the attributes should be: **rank** and **suit**.
  - It is less obvious what type the attributes should be.
  - One possibility is to use strings like 'Spade' for suits and 'Queen' for ranks. A problem with this implementation is that it would **not be easy to compare** cards to see which has a higher rank or suit.
  - An alternative is to use integers to **encode** the ranks and suits.
    - In this context, “encode” means that we are going to define a mapping between numbers and suits, or between numbers and ranks. This kind of encoding is not meant to be a secret (that would be “encryption”).

1. Presenting Cards

- Encode the suits:

Suit	Code
Spades	3
Hearts	2
Diamonds	1
Clubs	0

  – With this encoding, we can compare suits by comparing their codes.

- Encode the ranks:
  – We'll use the integer 2 to represent the rank 2, 3 to represent 3, and so on up to 10. The following table shows the codes for the face cards:

Rank	Code
Jack	11
Queen	12
King	13

  – And we can use either 1 or 14 to represent an Ace, depending on whether we want it to be considered lower or higher than the other ranks.

1. Presenting Cards

- Here's a definition for a class that represents a playing card, with these lists of strings as **class variables**, which are variables defined **inside a class definition**, but not inside a method:

```
class Card:
    """Represents a standard playing card."""
    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']
```

```
print(Card.suit_names)
```

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

  – Class variables are associated with the class, rather than an instance of the class, so we can access them like **Card.suit_names**, **Card.rank_names**.

2. Card Attributes

- A Card object has two attributes **suit** and **rank**.

```
class Card:
    """Represents a standard playing card."""
    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']

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

```
queen = Card(1, 12)
print(queen.suit, queen.rank)
```

```
1 12
```

- We use the instance (object) to access the attributes:
  - **queen.suit** **queen.rank**
- We use the class to access the class variable:
  - **Card.suit_names** **Card.rank_names**

- Every **Card** instance has its own **suit** and **rank** attributes, but there is only one **Card** class object, and only one copy of the class variables **suit_names** and **rank_names**.

3. Printing Cards

- The **__str__** method of the **Card** class to return a string representing a card:

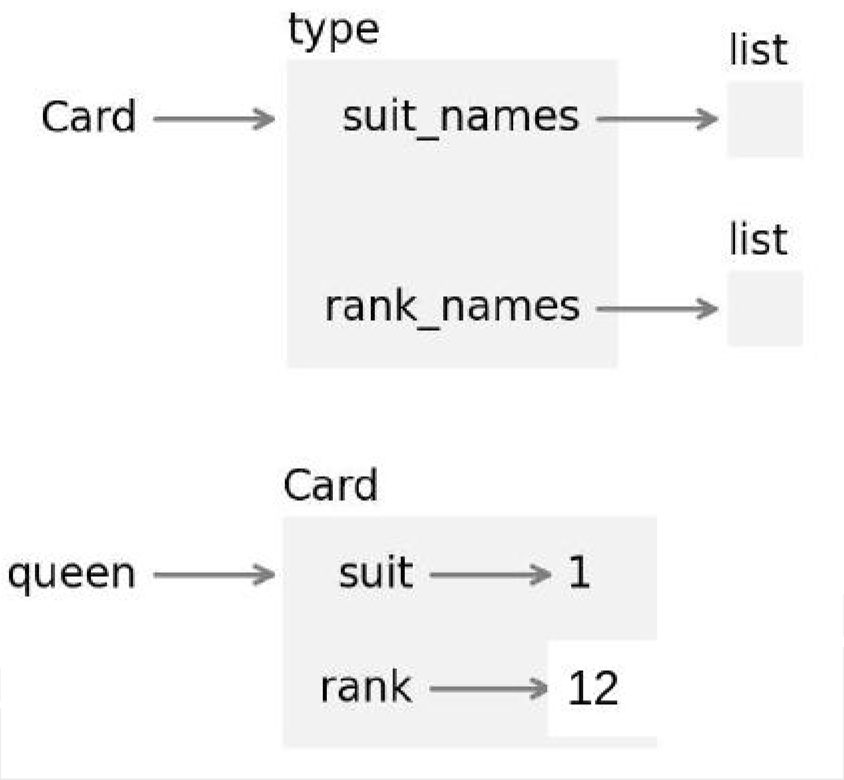
```python
class Card:
    """Represents a standard playing card."""
    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        rank_name = Card.rank_names[self.rank]
        suit_name = Card.suit_names[self.suit]
        return f'{rank_name} of {suit_name}'
```

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

```
Queen of Diamonds
```



4. Comparing Cards

- **__eq__ (==)** takes two **Card** objects as parameters and returns **True** if they have the same **suit** and **rank**.
  - If we use the **!=** operator, Python invokes a special method called **__ne__**, if it exists. Otherwise, it invokes **__eq__** and inverts the result—so if **__eq__** returns **True**, the result of the **!=** operator is **False**.
- To compare two cards to see which is bigger, we can define a special method called **__lt__ (<)**, which is short for "less than."
  - For the sake of this example, let's assume that **suit is more important than rank**—so all Spades outrank all Hearts, which outrank all Diamonds, and so on. If two cards have the same suit, the one with the higher rank wins.

4. Comparing Cards

```python
class Card:
    """Represents a standard playing card."""
    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        rank_name = Card.rank_names[self.rank]
        suit_name = Card.suit_names[self.suit]
        return f'{rank_name} of {suit_name}'

    def __eq__(self, other):
        return self.suit == other.suit and self.rank == other.rank

    def to_tuple(self):
        return (self.suit, self.rank)

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

```python
queen = Card(1, 12)
six = Card(1, 6)

six < queen
```

```
True
```

- **Tuple comparison compares the first elements from each tuple, which represent the suits. If they are the same, it compares the second elements, which represent the ranks.**

4. Comparing Cards

- If we use the **>** operator, it invokes a special method called **__gt__**, if it exists. Otherwise it invokes **__lt__** with the arguments in the opposite order.

```python
class Card:
    """Represents a standard playing card."""
    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        rank_name = Card.rank_names[self.rank]
        suit_name = Card.suit_names[self.suit]
        return f'{rank_name} of {suit_name}'

    def __eq__(self, other):
        return self.suit == other.suit and self.rank == other.rank

    def to_tuple(self):
        return (self.suit, self.rank)

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

```python
queen = Card(1, 12)
six = Card(1, 6)

queen > six
```

4. Comparing Cards

- If we use the **<=** operator, it invokes a special method called **__le__**.
- If we use the **>=** operator, it uses **__ge__** if it exists. Otherwise, it invokes **__le__** with the arguments in the opposite order.

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

    def to_tuple(self):
        return (self.suit, self.rank)

    def __lt__(self, other):
        return self.to_tuple() < other.to_tuple()

    def __le__(self, other):
        return self.to_tuple() <= other.to_tuple()
```

```
queen = Card(1, 12)
six = Card(1, 6)

six <= queen
```

```
True
```

```
queen >= six
```

```
True
```

5. Decks

- We have objects that represent cards, let's define objects that represent **decks**.

```python
class Deck:
    def __init__(self, cards):
        self.cards = cards

    def make_cards():
        cards = []
        for suit in range(4):
            for rank in range(2, 15):
                card = Card(suit, rank)
                cards.append(card)
        return cards
```

```python
cards = Deck.make_cards()
deck = Deck(cards)

len(deck.cards)
```

```
52
```

  - **make_cards** is a static method.

6. Printing the Deck

- Define a **__str__** method for **Deck**:

```python
class Deck:
    def __init__(self, cards):
        self.cards = cards

    def make_cards():
        cards = []
        for suit in range(4):
            for rank in range(2, 15):
                card = Card(suit, rank)
                cards.append(card)
        return cards

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

```python
queen = Card(1, 12)
six = Card(1, 6)

small_deck = Deck([queen, six])
print(small_deck)
```

```
Queen of Diamonds
6 of Diamonds
```

7. Add, Remove, Shuffle, and Sort

- To deal cards, we would like a method that removes a card from the deck and returns it. The list method **pop** provides a convenient way to do that:

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

- To add a card, we can use the list method **append**:

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

- To shuffle the deck, we can use the **shuffle** function from the **random** module:

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

7. Add, Remove, Shuffle, and Sort

- To sort the cards, we can use the list method **sort**, which sorts the elements “in place”—that is, it modifies the list rather than creating a new list:
  - When we invoke sort, it uses the **__lt__** method to compare cards:

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

8. Parents and Children

- Inheritance is the ability to define a new class that is a modified version of an existing class. As an example, let’s say we want a 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—where one is a specialized version of another—lends itself to **inheritance**.
- To define a new class that inherits from an existing class, we put the name of the existing class in parentheses.

8. Parents and Children

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

- This definition indicates that **Hand** inherits from **Deck**, which means that **Hand** objects can access methods defined in **Deck**, like **take_card** and **put_card**.

- **Hand** also inherits **__init__** from **Deck**, but if we define **__init__** in the **Hand** class, it overrides the one in the **Deck** class:

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

    def __init__(self, label=''):
        self.label = label
        self.cards = []
```

```python
hand = Hand('player 1')

hand.label
```

```
'player 1'
```

- This version of **__init__** takes an optional string as a parameter, and always starts with an empty list of cards.
- When we create a **Hand** , Python invokes this method, not the one in **Deck**.

8. Parents and Children

- To deal a card, we can use **take_card** to remove a card from a **Deck**, and **put_card** to add the card to a **Hand**:

```
deck = Deck(cards)
card = deck.take_card()
hand.put_card(card)
print(hand)
```

```
Ace of Spades
```

- Let's encapsulate this code in a **Deck** method called **move_cards**:

```
    def move_cards(self, other, num):
        for i in range(num):
            card = self.take_card()
            other.put_card(card)
```

- This method is **polymorphic**—that is, it works with more than one type: **self** and other can be either a **Hand** or a **Deck**. So we can use this method to deal a card from **Deck** to a **Hand**, from one **Hand** to another, or from a **Hand** back to a **Deck** .

8. Parents and Children

- When a new class inherits from an existing one, the existing one is called the **parent** and the new class is called the **child**. In general:
  - Instances of the child class should have all of the attributes of the parent class, but they can have additional attributes.
  - The child class should have all of the methods of the parent class, but it can have additional methods.
  - If a child class overrides a method from the parent class, the new method should take the same parameters and return a compatible result.
- Liskov substitution principle (Barbara Liskov)
  - Any function or method designed to work with an instance of a parent class, like a **Deck**, will also work with instances of a child class, like **Hand**.

9. Specialization

- Let’s make a class called **BridgeHand** that represents a hand in bridge—a widely played card game.
  - We’ll inherit from **Hand** and add a new method called **high_card_point_count** that evaluates a hand using a “high card point” method, which adds up points for the high cards in the hand.
  - Mapping from card names to their point values:
    - ‘Ace’ → 4
    - ‘King’ → 3
    - ‘Queen’ → 2
    - ‘Jack’ → 1

9. Specialization

- Here's the **BridgeHand** class definition that contains as a class variable a dictionary that maps from card names to their point values:

```python
class BridgeHand(Hand):
    """Represents a bridge hand."""

    hcp_dict = {'Ace': 4, 'King': 3, 'Queen': 2, 'Jack': 1, }
```

```python
rank = 12
rank_name = Card.rank_names[rank]
score = BridgeHand.hcp_dict.get(rank_name, 0)
rank_name, score
```

```
('Queen', 2)
```

9. Specialization

- The following method loops through the cards in a **BridgeHand** and adds up their scores:

```
class BridgeHand(Hand):
    """Represents a bridge hand."""

    hcp_dict = {'Ace': 4, 'King': 3, 'Queen': 2, 'Jack': 1, }

    def high_card_point_count(self):
        count = 0
        for card in self.cards:
            rank_name = Card.rank_names[card.rank]
            count += BridgeHand.hcp_dict.get(rank_name, 0)
        return count
```

```
cards = Deck.make_cards()
deck = Deck(cards)

hand = BridgeHand('player 2')

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

print(hand)
hand.high_card_point_count()
```

```
Jack of Diamonds
10 of Hearts
8 of Spades
6 of Clubs
King of Diamonds

4
```

- **BridgeHand** inherits the variables and methods of **Hand** and adds a class variable and a method that are specific to bridge. This way of using inheritance is called **specialization** because it defines a new class that is specialized for a particular use, like playing bridge.

- BridgeHand inherits the variables and methods of Hand and adds a class variable and a method that are specific to bridge. This way of using inheritance is called **specialization** because it defines a new class that is specialized for a particular use, like playing bridge.