23. Inheritance

23.1. Inheritance

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.

The primary advantage of this feature is that you can add new methods to a class without modifying the existing class. It is called inheritance because the new class inherits all of the methods of the existing class. Extending this metaphor, the existing class is sometimes called the **parent** class. The new class may be called the **child** class or sometimes subclass.

Inheritance is a powerful feature. Some programs that would be complicated without inheritance can be written concisely and simply with it. Also, 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 program 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 scattered among several modules. Also, many of the things that can be done using inheritance can be done as elegantly (or more so) without it. If the natural structure of the problem does not lend itself to inheritance, this style of programming can do more harm than good.

In this chapter we will demonstrate the use of inheritance as part of a program that plays the card game Old Maid. One of our goals is to write code that could be reused to implement other card games.

23.2. A hand of cards

For almost any card game, we need to represent a hand of cards. A hand is similar to a deck, of course. Both are made up of a set of cards, and both require operations like adding and removing cards. Also, we might like the ability to shuffle both decks and hands.

A hand is also different from a deck. Depending on the game being played, we might want to perform some operations on hands that don't make sense for a deck. For example, in poker we might classify a hand (straight, flush, etc.) or compare it with another hand. In bridge, we might want to compute a score for a hand in order to make a bid.

This situation suggests the use of inheritance. If `Hand` is a subclass of `Deck`, it will have all the methods of `Deck`, and new methods can be added.

We add the code in this chapter to our `Cards.py` file from the previous chapter. In the class definition, the name of the parent class appears in parentheses:

```python
class Hand(Deck):
    pass
```

This statement indicates that the new `Hand` class inherits from the existing `Deck` class.

The `Hand` constructor initializes the attributes for the hand, which are `name` and `cards`. The string `name` identifies this hand, probably by the name of the player that holds it. The name is an optional parameter with the empty string as a default value. `cards` is the list of cards in the hand, initialized to the empty list:

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

For just about any card game, it is necessary to add and remove cards from the deck. Removing cards is already taken care of, since `Hand` inherits `remove` from `Deck`. But we have to write `add`:

```python
class Hand(Deck):
    ...
    def add(self, card):
        self.cards.append(card)
```

Again, the ellipsis indicates that we have omitted other methods. The list `append` method adds the new card to the end of the list of cards.

23.3. Dealing cards

Now that we have a `Hand` class, we want to deal cards from the `Deck` into hands. It is not immediately obvious whether this method should go in the `Hand` class or in the `Deck` class, but since it operates on a single deck and (possibly) several hands, it is more natural to put it in `Deck`.

`deal` should be fairly general, since different games will have different requirements. We may want to deal out the entire deck at once or add one card to each hand.

`deal` takes two parameters, a list (or tuple) of hands and the total number of cards to deal. If there are not enough cards in the deck, the method deals out all of the cards and stops:

```python
class Deck:
    ...
    def deal(self, hands, num_cards=999):
        num_hands = len(hands)
        for i in range(num_cards):
            if self.is_empty():
                break                    # Break if out of cards
            card = self.pop()            # Take the top card
            hand = hands[i % num_hands]  # Whose turn is next?
            hand.add(card)               # Add the card to the hand
```

The second parameter, `num_cards`, is optional; the default is a large number, which effectively means that all of the cards in the deck will get dealt.

The loop variable `i` goes from 0 to `num_cards-1`. Each time through the loop, a card is removed from the deck using the list method `pop`, which removes and returns the last item in the list.

The modulus operator (`%`) allows us to deal cards in a round robin (one card at a time to each hand). When `i` is equal to the number of hands in the list, the expression `i % num_hands` wraps around to the beginning of the list (index 0).

23.4. Printing a Hand

To print the contents of a hand, we can take advantage of the `__str__` method inherited from `Deck`. For example:

```
>>> deck = Deck()
>>> deck.shuffle()
>>> hand = Hand("frank")
>>> deck.deal([hand], 5)
>>> print(hand)
Hand frank contains
2 of Spades
 3 of Spades
  4 of Spades
   Ace of Hearts
    9 of Clubs
```

It’s not a great hand, but it has the makings of a straight flush.

Although it is convenient to inherit the existing methods, there is additional information in a `Hand` object we might want to include when we print one. To do that, we can provide a `__str__` method in the `Hand` class that overrides the one in the `Deck` class:

```python
class Hand(Deck)
    ...
    def __str__(self):
        s = "Hand " + self.name
        if self.is_empty():
            s += " is empty\n"
        else:
            s += " contains\n"
        return s + Deck.__str__(self)
```

Initially, `s` is a string that identifies the hand. If the hand is empty, the program appends the words `is empty` and returns `s`.

Otherwise, the program appends the word `contains` and the string representation of the `Deck`, computed by invoking the `__str__` method in the `Deck` class on `self`.

It may seem odd to send `self`, which refers to the current `Hand`, to a `Deck` method, until you remember that a `Hand` is a kind of `Deck`. `Hand` objects can do everything `Deck` objects can, so it is legal to send a `Hand` to a `Deck` method.

In general, it is always legal to use an instance of a subclass in place of an instance of a parent class.

23.5. The `CardGame` class

The `CardGame` class takes care of some basic chores common to all games, such as creating the deck and shuffling it:

```python
class CardGame:
    def __init__(self):
        self.deck = Deck()
        self.deck.shuffle()
```

This is the first case we have seen where the initialization method performs a significant computation, beyond initializing attributes.

To implement specific games, we can inherit from `CardGame` and add features for the new game. As an example, we'll write a simulation of Old Maid.

The object of Old Maid is to get rid of cards in your hand. You do this by matching cards by rank and color. For example, the 4 of Clubs matches the 4 of Spades since both suits are black. The Jack of Hearts matches the Jack of Diamonds since both are red.

To begin the game, the Queen of Clubs is removed from the deck so that the Queen of Spades has no match. The fifty-one remaining cards are dealt to the players in a round robin. After the deal, all players match and discard as many cards as possible.

When no more matches can be made, play begins. In turn, each player picks a card (without looking) from the closest neighbor to the left who still has cards. If the chosen card matches a card in the player's hand, the pair is removed. Otherwise, the card is added to the player's hand. Eventually all possible matches are made, leaving only the Queen of Spades in the loser's hand.

In our computer simulation of the game, the computer plays all hands. Unfortunately, some nuances of the real game are lost. In a real game, the player with the Old Maid goes to some effort to get their neighbor to pick that card, by displaying it a little more prominently, or perhaps failing to display it more prominently, or even failing to fail to display that card more prominently. The computer simply picks a neighbor's card at random.

23.6. `OldMaidHand` class

A hand for playing Old Maid requires some abilities beyond the general abilities of a `Hand`. We will define a new class, `OldMaidHand`, that inherits from `Hand` and provides an additional method called `remove_matches`:

```python
class OldMaidHand(Hand):
    def remove_matches(self):
        count = 0
        original_cards = self.cards[:]
        for card in original_cards:
            match = Card(3 - card.suit, card.rank)
            if match in self.cards:
                self.cards.remove(card)
                self.cards.remove(match)
                print("Hand {0}: {1} matches {2}"
                        .format(self.name, card, match))
                count += 1
        return count
```

We start by making a copy of the list of cards, so that we can traverse the copy while removing cards from the original. Since `self.cards` is modified in the loop, we don't want to use it to control the traversal. Python can get quite confused if it is traversing a list that is changing!

For each card in the hand, we figure out what the matching card is and go looking for it. The match card has the same rank and the other suit of the same color. The expression `3 - card.suit` turns a Club (suit 0) into a Spade (suit 3) and a Diamond (suit 1) into a Heart (suit 2). You should satisfy yourself that the opposite operations also work. If the match card is also in the hand, both cards are removed.

The following example demonstrates how to use `remove_matches`:

```
>>> game = CardGame()
>>> hand = OldMaidHand("frank")
>>> game.deck.deal([hand], 13)
>>> print(hand)
Hand frank contains
Ace of Spades
 2 of Diamonds
  7 of Spades
   8 of Clubs
    6 of Hearts
     8 of Spades
      7 of Clubs
       Queen of Clubs
        7 of Diamonds
         5 of Clubs
          Jack of Diamonds
           10 of Diamonds
            10 of Hearts
>>> hand.remove_matches()
Hand frank: 7 of Spades matches 7 of Clubs
Hand frank: 8 of Spades matches 8 of Clubs
Hand frank: 10 of Diamonds matches 10 of Hearts
>>> print(hand)
Hand frank contains
Ace of Spades
 2 of Diamonds
  6 of Hearts
   Queen of Clubs
    7 of Diamonds
     5 of Clubs
      Jack of Diamonds
```

Notice that there is no `__init__` method for the `OldMaidHand` class. We inherit it from `Hand`.

23.7. `OldMaidGame` class

Now we can turn our attention to the game itself. `OldMaidGame` is a subclass of `CardGame` with a new method called `play` that takes a list of players as a parameter.

Since `__init__` is inherited from `CardGame`, a new `OldMaidGame` object contains a new shuffled deck:

```python
class OldMaidGame(CardGame):
    def play(self, names):
        # Remove Queen of Clubs
        self.deck.remove(Card(0,12))

        # Make a hand for each player
        self.hands = []
        for name in names:
            self.hands.append(OldMaidHand(name))

        # Deal the cards
        self.deck.deal(self.hands)
        print("---------- Cards have been dealt")
        self.print_hands()

        # Remove initial matches
        matches = self.remove_all_matches()
        print("---------- Matches discarded, play begins")
        self.print_hands()

        # Play until all 50 cards are matched
        turn = 0
        num_hands = len(self.hands)
        while matches < 25:
            matches += self.play_one_turn(turn)
            turn = (turn + 1) % num_hands

        print("---------- Game is Over")
        self.print_hands()
```

The writing of `print_hands` has been left as an exercise.

Some of the steps of the game have been separated into methods. `remove_all_matches` traverses the list of hands and invokes `remove_matches` on each:

```python
class OldMaidGame(CardGame):
    ...
    def remove_all_matches(self):
        count = 0
        for hand in self.hands:
            count += hand.remove_matches()
        return count
```

`count` is an accumulator that adds up the number of matches in each hand. When we've gone through every hand, the total is returned (`count`).

When the total number of matches reaches twenty-five, fifty cards have been removed from the hands, which means that only one card is left and the game is over.

The variable `turn` keeps track of which player's turn it is. It starts at 0 and increases by one each time; when it reaches `num_hands`, the modulus operator wraps it back around to 0.

The method `play_one_turn` takes a parameter that indicates whose turn it is. The return value is the number of matches made during this turn:

```python
class OldMaidGame(CardGame):
    ...
    def play_one_turn(self, i):
        if self.hands[i].is_empty():
            return 0
        neighbor = self.find_neighbor(i)
        picked_card = self.hands[neighbor].pop()
        self.hands[i].add(picked_card)
        print("Hand", self.hands[i].name, "picked", picked_card)
        count = self.hands[i].remove_matches()
        self.hands[i].shuffle()
        return count
```

If a player's hand is empty, that player is out of the game, so he or she does nothing and returns 0.

Otherwise, a turn consists of finding the first player on the left that has cards, taking one card from the neighbor, and checking for matches. Before returning, the cards in the hand are shuffled so that the next player's choice is random.

The method `find_neighbor` starts with the player to the immediate left and continues around the circle until it finds a player that still has cards:

```
class OldMaidGame(CardGame):
    ...
    def find_neighbor(self, i):
        num_hands = len(self.hands)
        for next in range(1,num_hands):
            neighbor = (i + next) % num_hands
            if not self.hands[neighbor].is_empty():
                return neighbor
```

If `find_neighbor` ever went all the way around the circle without finding cards, it would return `None` and cause an error elsewhere in the program. Fortunately, we can prove that that will never happen (as long as the end of the game is detected correctly).

We have omitted the `print_hands` method. You can write that one yourself.

The following output is from a truncated form of the game where only the top fifteen cards (tens and higher) were dealt to three players. With this small deck, play stops after seven matches instead of twenty-five.

```
>>> import cards
>>> game = cards.OldMaidGame()
>>> game.play(["Allen","Jeff","Chris"])
---------- Cards have been dealt
Hand Allen contains
King of Hearts
 Jack of Clubs
  Queen of Spades
   King of Spades
    10 of Diamonds

Hand Jeff contains
Queen of Hearts
 Jack of Spades
  Jack of Hearts
   King of Diamonds
    Queen of Diamonds

Hand Chris contains
Jack of Diamonds
 King of Clubs
  10 of Spades
   10 of Hearts
    10 of Clubs

Hand Jeff: Queen of Hearts matches Queen of Diamonds
Hand Chris: 10 of Spades matches 10 of Clubs
---------- Matches discarded, play begins
Hand Allen contains
King of Hearts
 Jack of Clubs
  Queen of Spades
   King of Spades
    10 of Diamonds

Hand Jeff contains
Jack of Spades
 Jack of Hearts
  King of Diamonds

Hand Chris contains
Jack of Diamonds
 King of Clubs
  10 of Hearts

Hand Allen picked King of Diamonds
Hand Allen: King of Hearts matches King of Diamonds
Hand Jeff picked 10 of Hearts
Hand Chris picked Jack of Clubs
Hand Allen picked Jack of Hearts
Hand Jeff picked Jack of Diamonds
Hand Chris picked Queen of Spades
Hand Allen picked Jack of Diamonds
Hand Allen: Jack of Hearts matches Jack of Diamonds
Hand Jeff picked King of Clubs
Hand Chris picked King of Spades
Hand Allen picked 10 of Hearts
Hand Allen: 10 of Diamonds matches 10 of Hearts
Hand Jeff picked Queen of Spades
Hand Chris picked Jack of Spades
Hand Chris: Jack of Clubs matches Jack of Spades
Hand Jeff picked King of Spades
Hand Jeff: King of Clubs matches King of Spades
---------- Game is Over
Hand Allen is empty

Hand Jeff contains
Queen of Spades

Hand Chris is empty
```

So Jeff loses.

23.8. Glossary

inheritance	The ability to define a new class that is a modified version of a previously defined class.
parent class	The class from which a child class inherits.
child class	A new class created by inheriting from an existing class; also called a subclass.

23.9. Exercises

1. Add a method, `print_hands`, to the `OldMaidGame` class which traverses `self.hands` and prints each hand.
2. Define a new kind of Turtle, `TurtleGTX`, that comes with some extra features: it can jump forward a given distance, and it has an odometer that keeps track of how far the turtle has travelled since it came off the production line. (The parent class has a number of synonyms like `fd`, `forward`, `back`, `backward`, and `bk`: for this exercise, just focus on putting this functionality into the `forward` method.) Think carefully about how to count the distance if the turtle is asked to move forward by a negative amount. (We would not want to buy a second-hand turtle whose odometer reading was faked because its previous owner drove it backwards around the block too often. Try this in a car near you, and see if the car's odometer counts up or down when you reverse.)
3. After travelling some random distance, your turtle should break down with a flat tyre. After this happens, raise an exception whenever `forward` is called. Also provide a `change_tyre` method that can fix the flat.