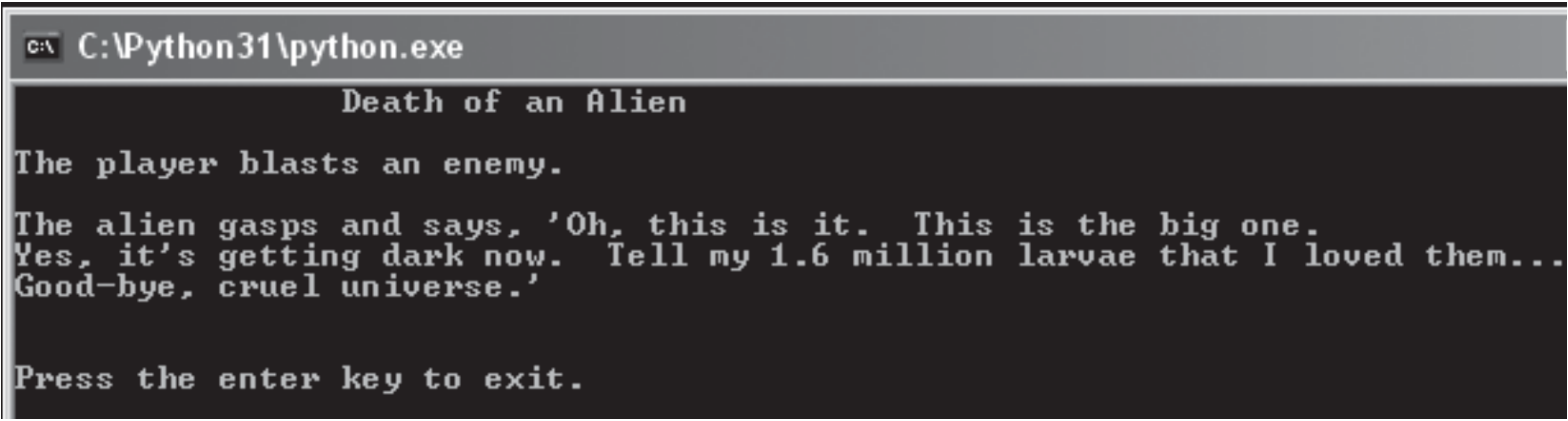
Chapter 9

Object-Oriented Programming: The Blackjack Game

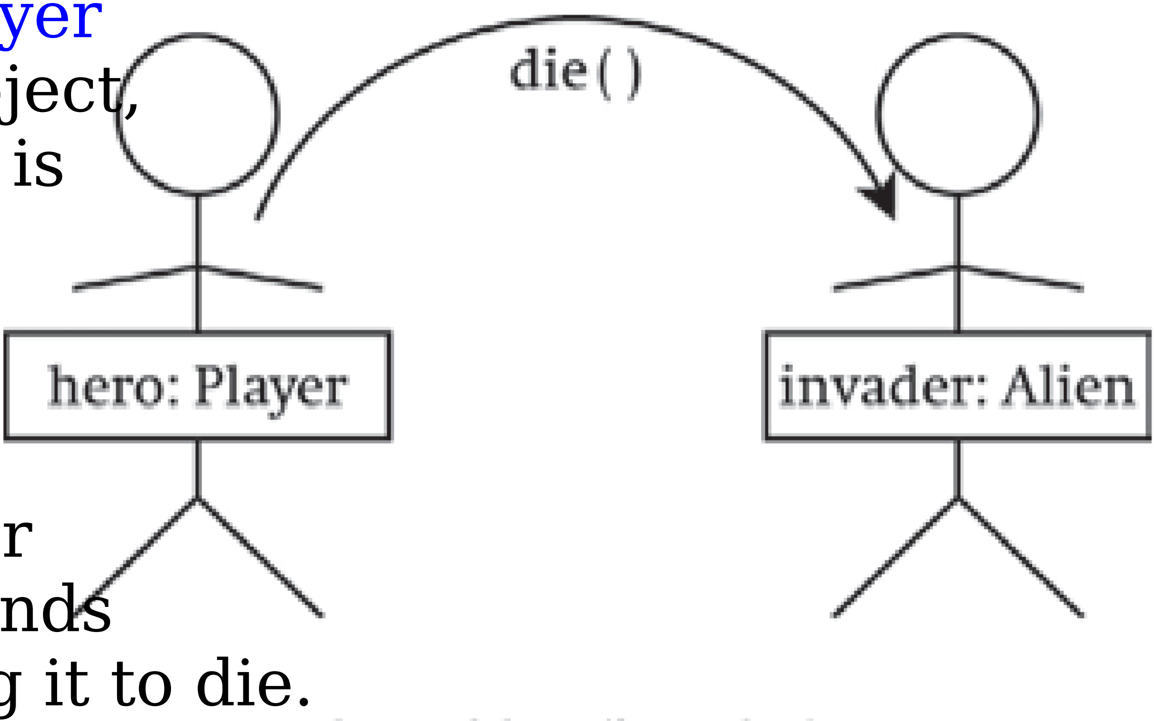
Introducing the Alien Blaster Program



```
                   Death of an Alien

The player blasts an enemy.

The alien gasps and says, 'Oh, this is it.  This is the big one.
Yes, it's getting dark now.  Tell my 1.6 million larvae that I loved them...
Good-bye, cruel universe.'


Press the enter key to exit.
```

- The code instantiates a Player object, hero, and an Alien object, invader. When hero's blast() is invoked with invader as its argument, hero invokes invader's die() method.

- It means that when a player blasts an alien, the player sends a message to the alien telling it to die.



hero.blast(invader)

alien_blaster.py

```
# Alien Blaster
# Demonstrates object interaction

class Player(object):
    """ A player in a shooter game. """
    def blast(self, enemy):
        print("The player blasts an enemy.\n")
        enemy.die()

class Alien(object):
    """ An alien in a shooter game. """
    def die(self):
        print("The alien gasps and says, 'Oh, this is it.”,
                 “This is the big one. \nYes, it's getting”,
                 “ dark now. Tell my 1.6 million larvae that”,
               “ I loved them... \nGood-bye, cruel universe.'")
```

```
# main
print("\t\tDeath of an Alien\n")

hero = Player()
invader = Alien()
hero.blast(invader)

input("\n\nPress the enter key to exit.")
```

Sending a Message

- Before you can have one object send another object a message, you need 2 objects! So, we create 2 in the main part of the program.

- Create a Player object, hero, and an Alien object, invader.

- Through hero.blast(invader), we invoke hero’s blast() and pass invader—the Alien object—as an argument.

- blast() accepts the object into its parameter enemy. So, when blast() executes, enemy refers to the Alien object.

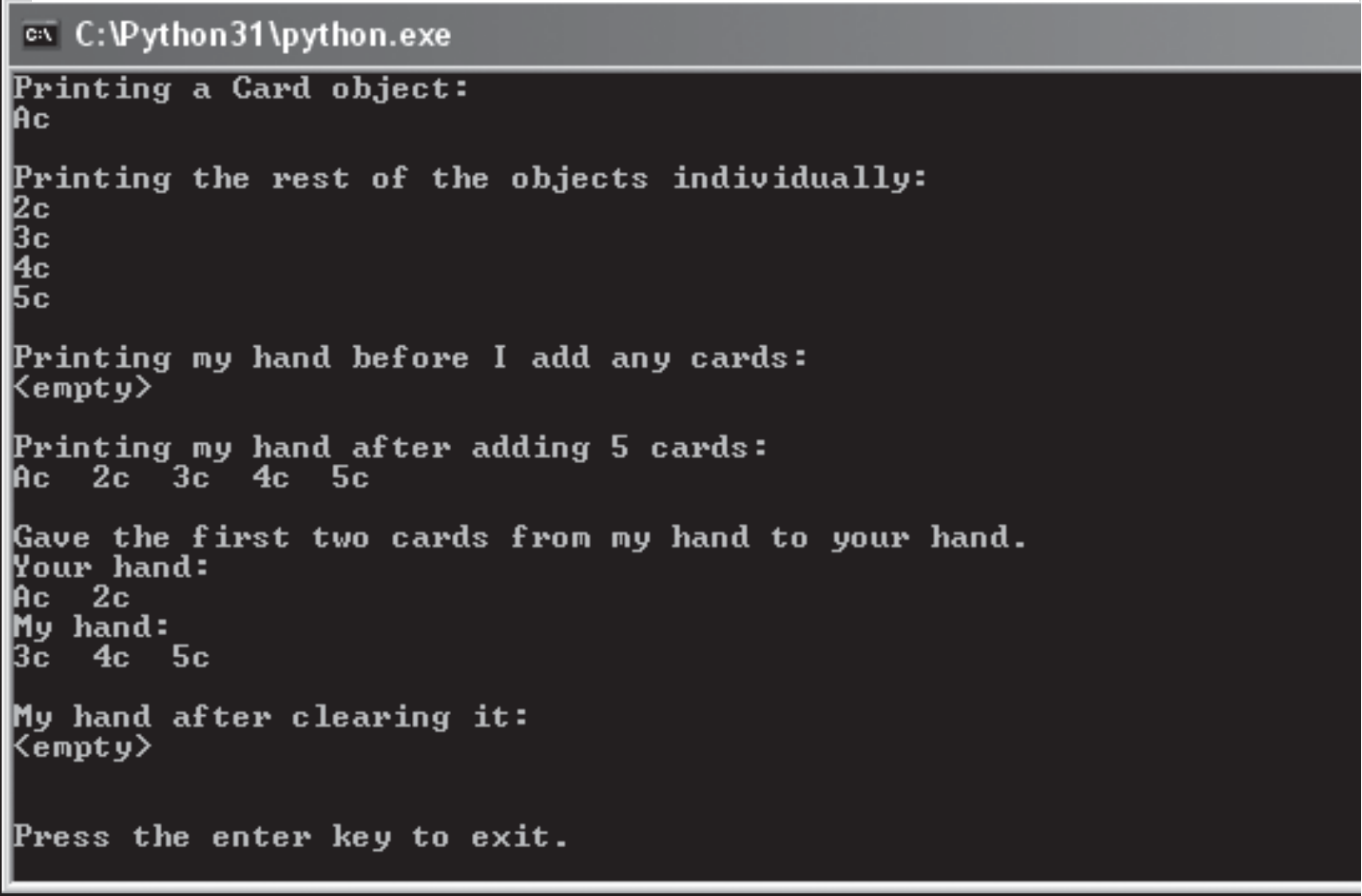
- Then blast() invokes the Alien object’s die() through enemy.die().

- Essentially, the Player object is sending the Alien object a message by invoking its die() method.

Receiving a Message

- The Alien object receives the message from the Player object in the form of its die() being invoked. The Alien object’s die() then displays a melodramatic good-bye.

Introducing the Playing Cards Program



```
Printing a Card object:
Ac

Printing the rest of the objects individually:
2c
3c
4c
5c

Printing my hand before I add any cards:
<empty>

Printing my hand after adding 5 cards:
Ac  2c  3c  4c  5c

Gave the first two cards from my hand to your hand.
Your hand:
Ac  2c
My hand:
3c  4c  5c

My hand after clearing it:
<empty>


Press the enter key to exit.
```

playing_cards.py

```
# Playing Cards
# Demonstrates combining objects

class Card(object):
    """ A playing card. """
    RANKS = ["A", "2", "3", "4", "5", "6", "7","8", "9",
             "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]

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

    def __str__(self):
        rep = self.rank + self.suit
        return rep
```

```
class Hand(object):
    """ A hand of playing cards. """
    def __init__(self):
        self.cards = []

    def __str__(self):
        if self.cards:
            rep = ""
            for card in self.cards:
                rep += str(card) + "  "
        else:
            rep = "<empty>"
        return rep

    def clear(self):
        self.cards = []

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

```python
    def give(self, card, other_hand):
        self.cards.remove(card)
        other_hand.add(card)



# main
card1 = Card(rank = "A", suit = "c")
print("Printing a Card object:")
print(card1)

card2 = Card(rank = "2", suit = "c")
card3 = Card(rank = "3", suit = "c")
card4 = Card(rank = "4", suit = "c")
card5 = Card(rank = "5", suit = "c")
print("\nPrinting the rest of the objects individually:")
print(card2)
print(card3)
print(card4)
print(card5)
```

```
my_hand = Hand()
print("\nPrinting my hand before I add any cards:")
print(my_hand)

my_hand.add(card1)
my_hand.add(card2)
my_hand.add(card3)
my_hand.add(card4)
my_hand.add(card5)
print("\nPrinting my hand after adding 5 cards:")
print(my_hand)

your_hand = Hand()
my_hand.give(card1, your_hand)
my_hand.give(card2, your_hand)
print("\nGave the 1st 2 cards to your hand.")
print("Your hand:")
print(your_hand)
print("My hand:")
print(my_hand)
```

```
my_hand.clear()
print("\nMy hand after clearing it:")
print(my_hand)

input("\n\nPress the enter key to exit.")
```

Creating the Card Class

- In the real world, interesting objects are usually made up of other, independent objects. We can do the same thing in coding. Combining objects allows you to create more complex objects from simpler ones.

- Create a Card class for objects that represent playing card:

```
class Card(object):
    RANKS = ["A", "2", "3", "4", "5", "6", "7", "8", "9",
                     "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]

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

    def __str__(self):
        rep = self.rank + self.suit
        return rep
```

- Each Card object has a rank attribute, which represents the rank of the card. The possible values are listed in the class attribute RANKS.

- Each card also has a suit attribute, which represents the suit of the card. The possible values for this attribute are listed in the class attribute SUITS.

- __str__() simply returns the concatenation of the rank and suit attributes so that an object can be printed.

Creating the Hand Class

- Create a Hand class for objects, a collection of Card objects

```
class Hand(object):
    """ A hand of playing cards. """
    def __init__(self):
        self.cards = []

    def __str__(self):
        if self.cards:
           rep = ""
           for card in self.cards:
               rep += str(card) + "  "
        else:
            rep = "<empty>"
        return rep

    def clear(self):
        self.cards = []
    def add(self, card):
        self.cards.append(card)

    def give(self, card, other_hand):
        self.cards.remove(card)
        other_hand.add(card)
```

- A new Hand object has an attribute cards that is a list of Card objects. So each single Hand object has an attribute that is a list of other objects.

- __str__() returns a string that represents the entire hand. clear() clears the list of cards by assigning an empty list to an object’s cards. add() adds an object to the cards.

- give() removes an object from the Hand object and appends it to another Hand object by using the other Hand object’s add(). Or, the 1st Hand object sends the 2nd Hand object a message to add a Card object.

Using Card Objects

- In the main part, we create and print 5 Card objects:

```
card1 = Card(rank = "A", suit = "c")
print("Printing a Card object:")
print(card1)

card2 = Card(rank = "2", suit = "c")
card3 = Card(rank = "3", suit = "c")
card4 = Card(rank = "4", suit = "c")
card5 = Card(rank = "5", suit = "c")
print("\nPrinting the rest of the objects individually:")
print(card2)
print(card3)
print(card4)
print(card5)
```

- The 1st Card object has a rank="A" and a suit="c". When we print the object, it’s displayed as Ac. The remaining objects follow the same pattern.

Combining Card Objects Using a Hand Object

- Next, we create a Hand object, my_hand, and print it:

```
my_hand = Hand()
print("\nPrinting my hand before I add any cards:")
print(my_hand)
```

- Since the object’s cards attribute is an empty list, printing the object displays the text <empty>.

- Add the 5 Card objects to my_hand and print it:

```
my_hand.add(card1)
my_hand.add(card2)
my_hand.add(card3)
my_hand.add(card4)
my_hand.add(card5)
print(my_hand)
```

- Then the text Ac 2c 3c 4c 5c is displayed.

- Create another Hand object, your_hand. Using my_hand's give() to transfer the 1st 2 cards from my_hand to your_hand:

```
your_hand = Hand()
my_hand.give(card1, your_hand)
my_hand.give(card2, your_hand)
print(your_hand)
print(my_hand)
```

- your_hand is displayed as Ac 2c while my_hand appears as 3c 4c 5c.

- Finally, invoke my_hand's clear() and print my_hand:

```
my_hand.clear()
print(my_hand)
```

- The text <empty> is displayed.

Using Inheritance to Create New Classes

- *Inheritance*, one of the key elements of OOP, allows you to base a new class on an existing one.

- By doing so, the new class automatically gets (or inherits) all of the methods and attributes of the existing class.

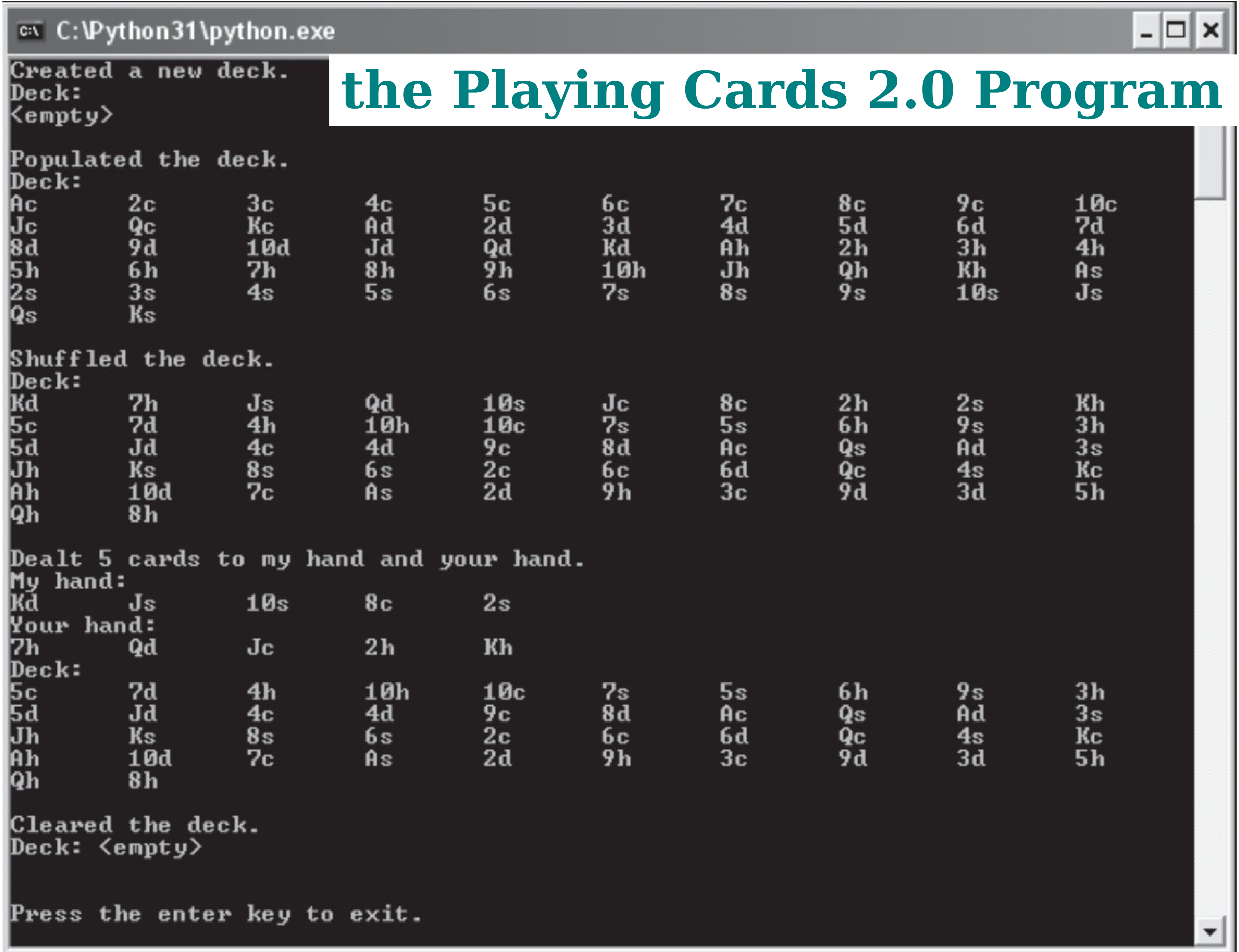
- It’s possible to create a new class that directly inherits from more than one class, ie, *multiple inheritance*.

- Inheritance is especially useful when you want to create a more specialized version of an existing class.

- By inheriting from an existing class, a new class gets all of the methods and attributes of the existing class.

- You can also add methods and attributes to the new class to extend what objects of the new class can do.

the Playing Cards 2.0 Program



```
Created a new deck.
Deck:
<empty>

Populated the deck.
Deck:
Ac      2c      3c      4c      5c      6c      7c      8c      9c      10c
Jc      Qc      Kc      Ad      2d      3d      4d      5d      6d      7d
8d      9d      10d     Jd      Qd      Kd      Ah      2h      3h      4h
5h      6h      7h      8h      9h      10h     Jh      Qh      Kh      As
2s      3s      4s      5s      6s      7s      8s      9s      10s     Js
Qs      Ks

Shuffled the deck.
Deck:
Kd      7h      Js      Qd      10s     Jc      8c      2h      2s      Kh
5c      7d      4h      10h     10c     7s      5s      6h      9s      3h
5d      Jd      4c      4d      9c      8d      Ac      Qs      Ad      3s
Jh      Ks      8s      6s      2c      6c      6d      Qc      4s      Kc
Ah      10d     7c      As      2d      9h      3c      9d      3d      5h
Qh      8h

Dealt 5 cards to my hand and your hand.
My hand:
Kd      Js      10s     8c      2s
Your hand:
7h      Qd      Jc      2h      Kh
Deck:
5c      7d      4h      10h     10c     7s      5s      6h      9s      3h
5d      Jd      4c      4d      9c      8d      Ac      Qs      Ad      3s
Jh      Ks      8s      6s      2c      6c      6d      Qc      4s      Kc
Ah      10d     7c      As      2d      9h      3c      9d      3d      5h
Qh      8h

Cleared the deck.
Deck: <empty>


Press the enter key to exit.
```

playing_cards2.py

```
# Playing Cards 2.0
# Demonstrates inheritance - class extension

class Card(object):
    """ A playing card. """
    RANKS = ["A", "2", "3", "4", "5", "6", "7",
             "8", "9", "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]

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

    def __str__(self):
        rep = self.rank + self.suit
        return rep
```

```
class Hand(object):
    """ A hand of playing cards. """
    def __init__(self):
        self.cards = []

    def __str__(self):
        if self.cards:
            rep = ""
            for card in self.cards:
                rep += str(card) + "\t"
        else:
            rep = "<empty>"
        return rep

    def clear(self):
        self.cards = []

    def add(self, card):
        self.cards.append(card)
    def give(self, card, other_hand):
        self.cards.remove(card)
        other_hand.add(card)
```

```
class Deck(Hand):
    def populate(self):
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                self.add(Card(rank, suit))
    def shuffle(self):
        import random
        random.shuffle(self.cards)

    def deal(self, hands, per_hand = 1):
        for rounds in range(per_hand):
            for hand in hands:
                if self.cards:
                    top_card = self.cards[0]
                    self.give(top_card, hand)
                else:
                    print("Out of cards!")
```

```
# main
deck1 = Deck()
print("Created a new deck.")
print("Deck:")
print(deck1)

deck1.populate()
print("\nPopulated the deck.")
print("Deck:")
print(deck1)

deck1.shuffle()
print("\nShuffled the deck.")
print("Deck:")
print(deck1)

my_hand = Hand()
your_hand = Hand()
hands = [my_hand, your_hand]
deck1.deal(hands, per_hand = 5)
print("\nDealt 5 cards to my hand and your hand.")
print("My hand:")
print(my_hand)
print("Your hand:")
print(your_hand)
print("Deck:")
print(deck1)

deck1.clear()
print("\nCleared the deck.")
print("Deck:", deck1)

input("\n\nPress the enter key to exit.")
```

Inheriting from a Base Class

- The 1st 2 classes, Card and Hand, are the same as before.
- The next thing is to create the Deck class, based on Hand:

```
class Deck(Hand):
```

- Hand is called a ***base class*** because Deck is based on it.

Deck is a ***derived class*** because it derives part of its definition from Hand.

- As a result of this relationship, Deck inherits all of Hand's methods.
- So without defining any new method, Deck objects would still have all of the methods defined in Hand:

- **__init__()** • **__str__()** • **clear()** • **add()** • **give()**

Extending a Derived Class

- You can extend a derived class by defining additional methods in it. That’s what we do in the definition of Deck.

- So, in addition to all of the methods that Deck inherits, it has the following new methods:

  - **populate()**
  - **shuffle()**
  - **deal()**

- As far as client code is concerned, any Deck method is as valid as any other—whether it’s inherited from Hand or defined in Deck.

Using the Derived Class

- In the main part we instantiate a new Deck object:

```
deck1 = Deck()
```

- We don’t have a constructor method in Deck. Deck inherits the Hand constructor, so that method is automatically invoked with the newly created Deck object.

- As a result, the new Deck object gets a cards attribute initialized to an empty list, as any Hand object would get.

- Print the new Deck object:

```
print(deck1)
```

- We didn’t define __str__() in Deck, but Deck inherits the method from Hand. Since the deck is empty, the code displays the text <empty>.

- A deck is a specialized type of hand. A deck can do anything a hand can, plus more.

- We invoke the object’s populate(), which populates the deck with the traditional 52 cards:

```
deck1.populate()
```

- Now the deck has done something a hand can’t because populate() is a new method defined in the Deck class.

- populate() loops through the 52 possible combinations of values of Card.SUITS and Card.RANKS. For each combination, the method creates a new Card object that it adds to the deck.

- Next, we print the deck again:

```
print(deck1)
```

- This time, all 52 cards are displayed, in an obvious order.

- We then shuffle the deck:

```
deck1.shuffle()
```

- In shuffle(), we imports the random module and then calls random.shuffle() with the object's cards.random.shuffle() shuffles a list's elements into a random order.

- Display the deck again to show its randomness:

```
print(deck1)
```

- Next, create 2 Hand objects and put them in a list, hands:

```
my_hand = Hand()
your_hand = Hand()
hands = [my_hand, your_hand]
```

- Deal each hand 5 cards:

```
deck1.deal(hands, per_hand = 5)
```

- deal() is a new method in Deck. It takes 2 arguments: a list of hands and the number of cards to deal each hand. The method gives a card from the deck to each hand. If the deck is out of cards, Can’t continue deal. Out of cards! is printed. The method repeats this process for the number of cards to be dealt each hand.

- To see the results of the deal, print each hand and the deck

```
print("My hand:")
print(my_hand)
print("Your hand:")
print(your_hand)
print("Deck:")
print(deck1)
```

- You can see that each hand has 5 cards and the deck now has only 42.

- Finally put the deck back to its initial state by clearing it:

```
deck1.clear()
```

- Then print the deck one last time to show its emptiness:

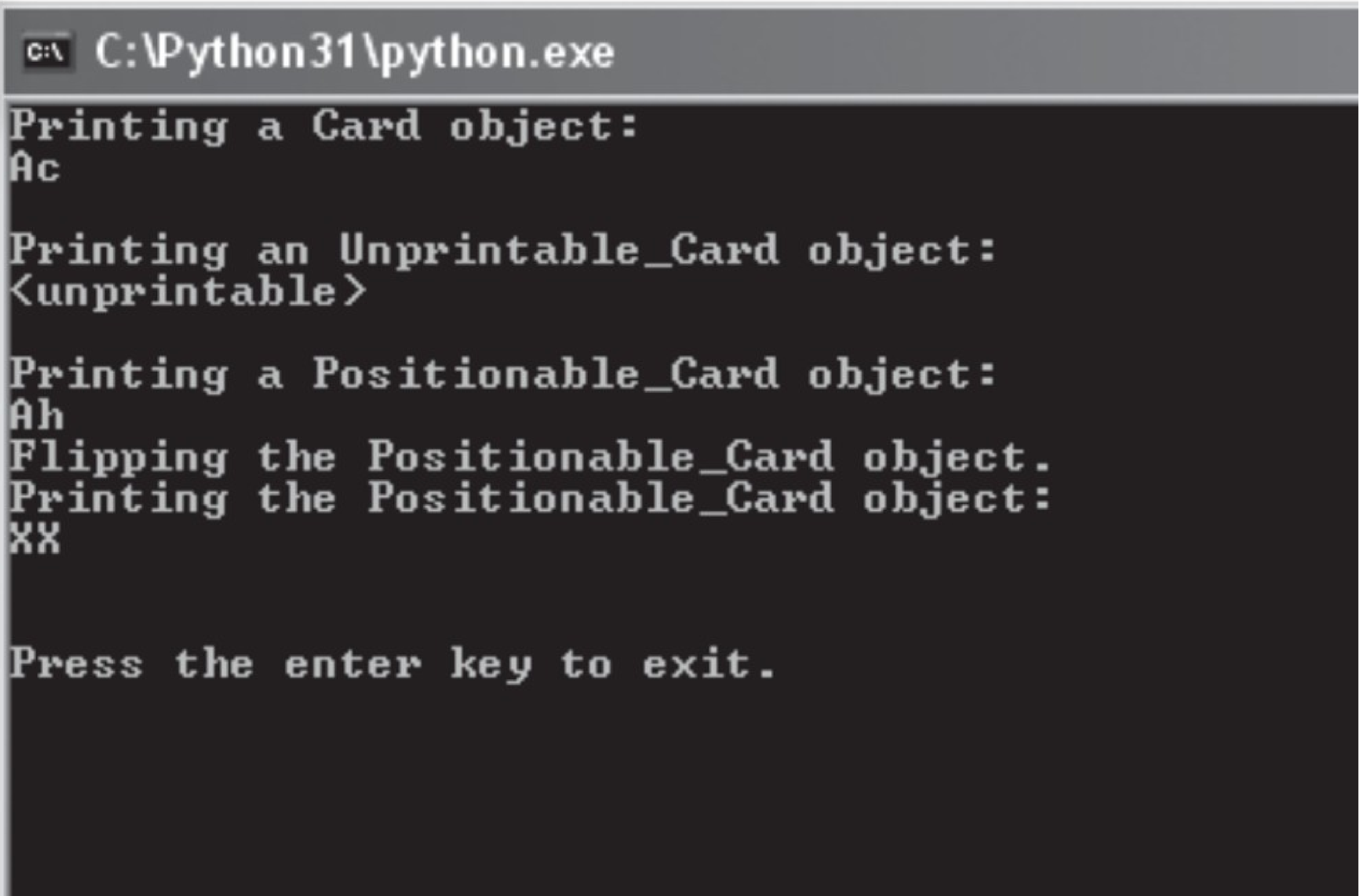
```
print("Deck:", deck1)
```

Alter the Behavior of Inherited Methods

- You can extend a class by adding new methods to a derived class. You can also redefine an inherited method of a base class in a derived class, ie, *overriding* the method.

- When you override a base class method, you can either create a method with completely new functionality, or you can incorporate the functionality of the base class method that you're overriding.

Introducing the Playing Cards 3 Program



```
Printing a Card object:
Ac

Printing an Unprintable_Card object:
<unprintable>

Printing a Positionable_Card object:
Ah
Flipping the Positionable_Card object.
Printing the Positionable_Card object:
XX


Press the enter key to exit.
```

playing_cards3.py

```
# Playing Cards 3.0
# Demonstrates inheritance - overriding methods

class Card(object):
    """ A playing card. """
    RANKS = ["A", "2", "3", "4", "5", "6", "7",
             "8", "9", "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]

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

    def __str__(self):
        rep = self.rank + self.suit
        return rep
```

```
class Unprintable_Card(Card):
    """A Card won't show its rank/suit when printed."""
    def __str__(self):
        return "<unprintable>"

class Positionable_Card(Card):
    """ A Card that can be face up or face down. """
    def __init__(self, rank, suit, face_up = True):
        super(Positionable_Card, self).__init__(rank, suit)
        self.is_face_up = face_up

    def __str__(self):
        if self.is_face_up:
            rep = super(Positionable_Card, self).__str__()
        else:
            rep = "XX"
        return rep

    def flip(self):
        self.is_face_up = not self.is_face_up
```

```
#main
card1 = Card("A", "c")
card2 = Unprintable_Card("A", "d")
card3 = Positionable_Card("A", "h")

print("Printing a Card object:")
print(card1)

print("\nPrinting an Unprintable_Card object:")
print(card2)

print("\nPrinting a Positionable_Card object:")
print(card3)
print("Flipping the Positionable_Card object.")
card3.flip()
print("Printing the Positionable_Card object:")
print(card3)

input("\n\nPress the enter key to exit.")
```

Overriding Base Class Methods

- Derive a new class for unprintable cards based on Card:

```
class Unprintable_Card(Card):
    """A Card won't show its rank/suit when printed."""
    def __str__(self):
        return "<unprintable>"
```

- Unprintable_Card inherits all of the methods of Card. But we can change an inherited method by defining it in a derived class.

- Unprintable_Card inherits __str__() from Card. But we define a new __str__() in Unprintable_Card to override (or replaces) the inherited one

- Any time you create a method in a derived class with the same name as an inherited method, you override the inherited method in the new class.

- So, when you print an Unprintable_Card object, the text <unprintable> is displayed.

- A derived class has no effect on a base class. A base class doesn’t care if you derive a new class from it, or if you override an inherited method in the new class. The base class still functions as it always has.

- So when you print a Card object, it will appear as it always does.

Invoking Base Class Methods

- Sometimes when you override the method of a base class, you want to incorporate the inherited method’s functionality.

- If we want to create a new type of playing card class based on Card. We want an object of this new class to have a new attribute to show if the card is face up.

- So we need to override the inherited constructor from Card with a new constructor that creates a face up attribute. But we still want the new constructor to create and set rank and suit attributes, like the Card constructor does.

- Instead of retyping the code from the Card constructor, we could invoke it from inside the new constructor. So it would take care of creating and initializing rank and suit attributes for an object of my new class.

- In the constructor of the new class, we could add the attribute that indicates whether the card is face up:

```
class Positionable_Card(Card):
    def __init__(self, rank, suit, face_up = True):
        super(Positionable_Card, self).__init__(rank, suit)
        self.is_face_up = face_up
```

- **super()** invoked the method of a base class (ie *superclass*). **super(**Positionable_Card, self**)**.__init__(rank, suit) invokes __init__() of Card (the superclass of Positionable_Card).

- The 1st argument to **super()**, Positionable_Card, is to invoke a method of the superclass of Positionable_Card, ie, Card. The 2nd argument, self, passes a reference to the newly instantiated Positionable_Card object so that code in the Card can add the rank and suit attributes to it.

- The next part of the statement, __init__(rank, suit), invokes the constructor of Card and passes it rank and suit.

- The next method in Positionable_Card overrides a method inherited from Card and invokes the overridden method:

```
def __str__(self):
    if self.is_face_up:
        rep = super(Positionable_Card, self).__str__()
    else:
        rep = "XX"
    return rep
```

- This __str__() first checks if an object’s face_up attribute is True. If so, the string for the card is set to the string from Card’s __str__() called with the Positionable_Card object.

- So if the card is face up, the card prints out like any object of the Card class. But if the card is not face up, the string returned is "XX".

- The last method in the class is a new one:

```
def flip(self):
    self.is_face_up = not self.is_face_up
```

- The method flips a card over by toggling the value of an object’s face_up attribute.

Using the Derived Classes

- In the main part, we create 3 objects: one from Card, one from Unprintable_Card, and the last from Positionable_Card:

```
card1 = Card("A", "c")
card2 = Unprintable_Card("A", "d")
card3 = Positionable_Card("A", "h")
```

- Print the Card object:

```
print(card1)
```

the text Ac is displayed.

- Print an Unprintable_Card object:

```
print(card2)
```

shows <unprintable> because the Unprintable_Card class overrides its inherited __str__().

- Print a Positionable_Card object:

```
print(card3)
```

- Since the object’s face_up is True, the object’s __str__() invokes Card’s __str__() and the text Ah is displayed.

- Invoke the Positionable_Card object’s flip():

```
card3.flip()
```

sets the face_up to False. Print the Positionable_Card object again:

```
print(card3)
```

- This time XX is displayed because the face_up is False.

Understanding Polymorphism

- *Polymorphism* is the quality of being able to treat different types of things the same and have those things each react in their own way.

- In OOP, polymorphism means that you can send the same message to objects of different classes related by inheritance and achieve different and appropriate results.

- Unprintable_Card is derived from Card. When you invoke __str__() of an Unprintable_Card object, you get a different result than when you invoke __str__() of a Card object.

- The result of this polymorphic behavior is that you are able to print an object even if you don’t know whether it’s an Unprintable_Card or a Card object.

- Regardless of the class of the object, when printed, its __str__() is invoked and the correct string of it is displayed.

Creating Modules

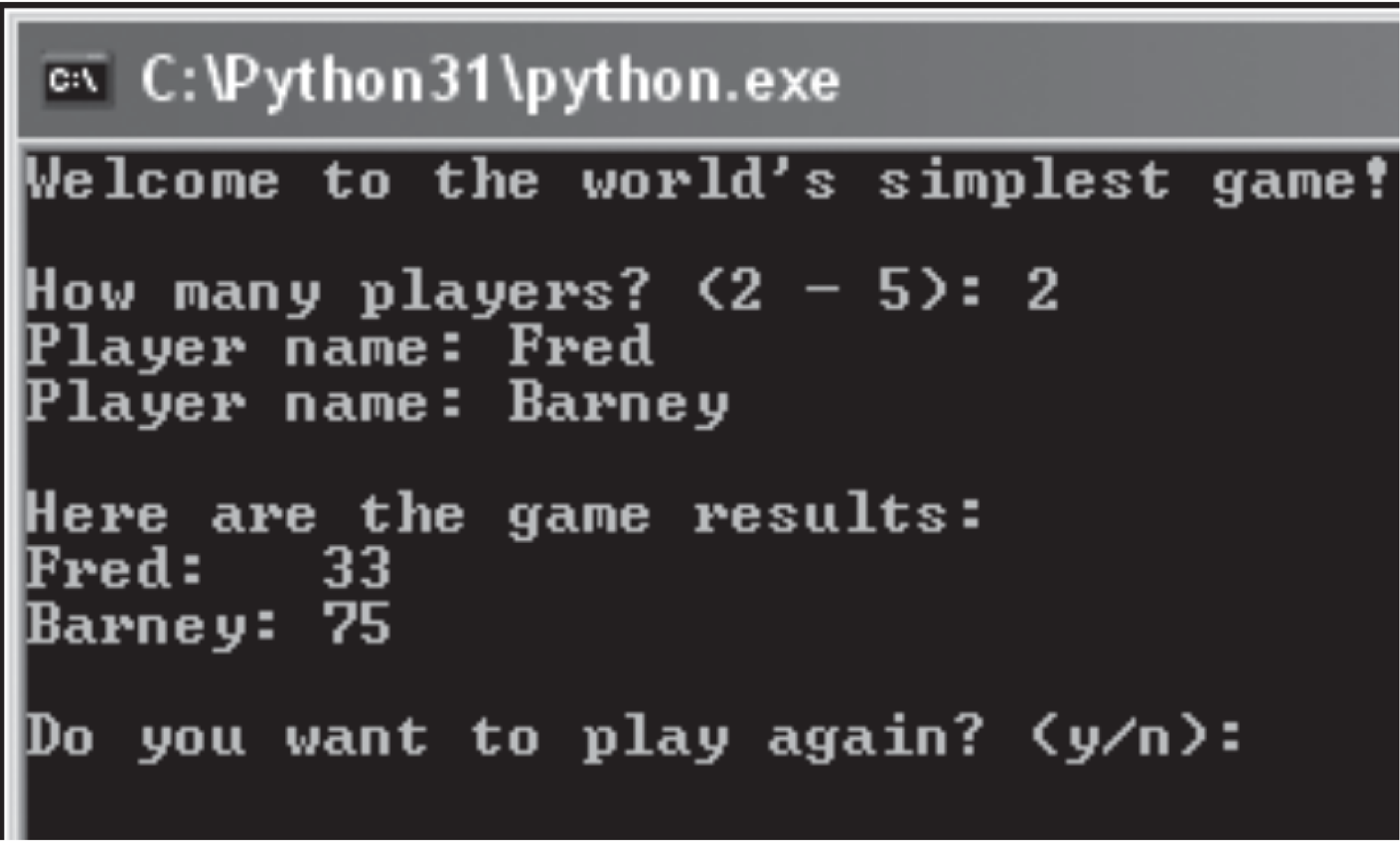
Creating your own modules provides important benefits:

* By creating your own modules, you can reuse code, which can save you time and effort.

* By breaking up a program into logical modules, large programs become easier to manage.

* By creating modules, you can share your genius.

Introducing the Simple Game Program



```
Welcome to the world's simplest game!

How many players? (2 - 5): 2
Player name: Fred
Player name: Barney

Here are the game results:
Fred:   33
Barney: 75

Do you want to play again? (y/n):
```

games.py

```
# Games
# Demonstrates module creation

class Player(object):
    """ A player for a game. """
    def __init__(self, name, score = 0):
        self.name = name
        self.score = score

    def __str__(self):
        rep = self.name + ":\t" + str(self.score)
        return rep

def ask_yes_no(question):
    """Ask a yes or no question."""
    response = None
    while response not in ("y", "n"):
        response = input(question).lower()
    return response
```

```python
def ask_number(question, low, high):
    """Ask for a number within a range."""
    response = None
    while response not in range(low, high):
        response = int(input(question))
    return response


if __name__ == "__main__":
    print("You ran this module directly.")
    input("\n\nPress the enter key to exit.")
```

Writing Modules

- Create a module the same way you write Python programs.

- When you create a module, you should build a collection of related components, such as functions and classes, and store them in a single file to be imported into a new program.

- This module is named games because we saved the file with the name games.py. Programmer-created modules are named (and imported) based on their file names.

- The next part of the program introduces a new idea related to modules. The condition of the if statement,

__name__ == "__main__"

is true if the program is run directly. It’s false if the file is imported as a module. So, if the games.py file is run directly, a message is displayed telling the user that the file is meant to be imported and not directly run.

simple_game.py

```
# Simple Game
# Demonstrates importing modules

import games, random

print("Welcome to the world's simplest game!\n")

again = None
while again != "n":
    players = []
    num = games.ask_number(question="How many ”,
    ”players? (2 - 5): ", low = 2, high = 5)
    for i in range(num):
        name = input("Player name: ")
        score = random.randrange(100) + 1
        player = games.Player(name, score)
        players.append(player)
    print("\nHere are the game results:")
    for player in players:
        print(player)

    again = games.ask_yes_no("\nPlay again? (y/n): ")

input("\n\nPress the enter key to exit.")
```

Importing Modules

- We import a programmer-created module the same way we import a built-in module, with the import statement:

```
import games, random
```

- If a programmer-created module isn’t in the same directory as the program that imports it, Python won’t be able to find the module.

- Make sure that any module you want to import is in the same directory as the programs that import it.

Using Imported Functions and Classes

- In a simple loop, we get the number of players by calling ask_number() from the games module:

```
again = None
while again != "n":
    players = []
    num = games.ask_number(question="How many”, \
    ”players? (2 - 5): ", low = 2, high = 5)
```

- Just as with other imported modules, to call a function we use dot notation, specifying first the module name, followed by the function name.

- Next, for each player, we get the player’s name and generate a random score between 1 – 100 by calling randrange() from the random module.

- Then, we create a player object using this name and score.

- Since the Player class is defined in the games module, use dot and put the module name before the class name.
- Append this new player object to a list of players:

```
for i in range(num):
    name = input("Player name: ")
    score = random.randrange(100) + 1
    player = games.Player(name, score)
    players.append(player)
```

- Print each player in the game:

```
for player in players:
    print(player)
```

- Finally, ask if the players want to play another game with ask_yes_no() from the games module:

```
again = games.ask_yes_no("\nPlay again? (y/n): ")
```

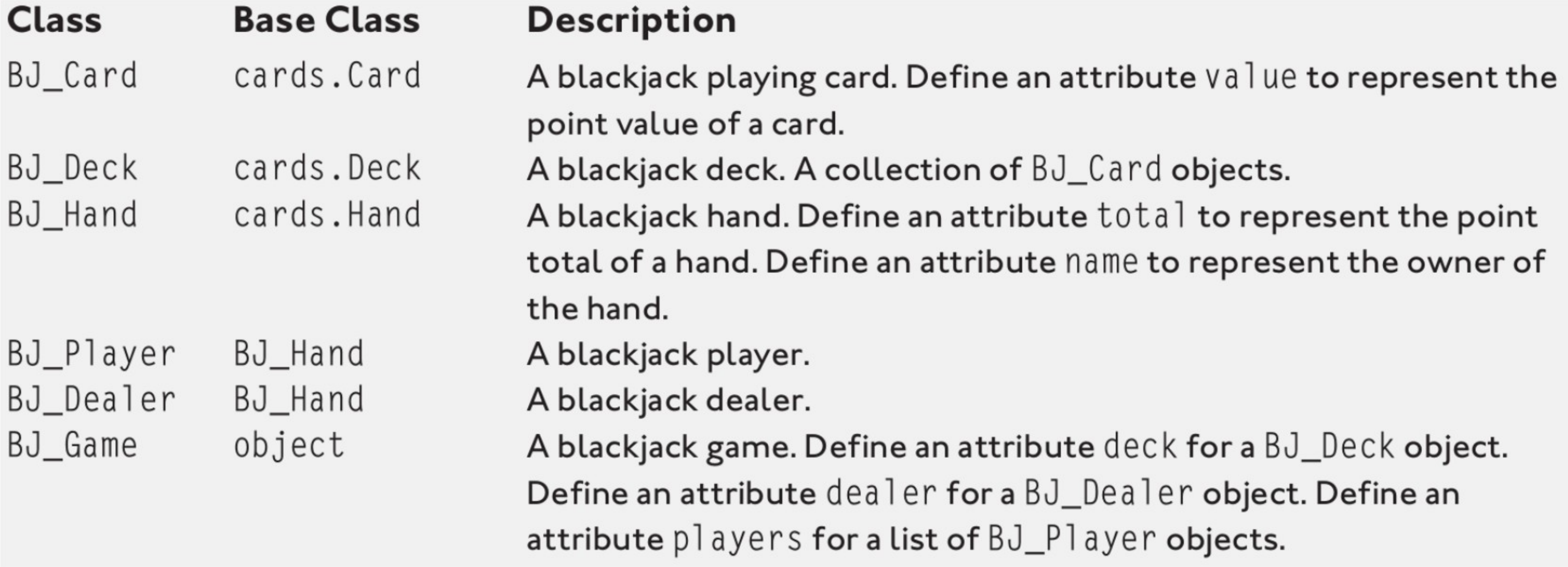
cards.py

```
# Cards Module
# Basic classes for a game with playing cards

class Card(object):
    RANKS = ["A", "2", "3", "4", "5", "6", "7",
              "8", "9", "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]
    def __init__(self, rank, suit, face_up = True):
        self.rank = rank
        self.suit = suit
        self.is_face_up = face_up

    def __str__(self):
        if self.is_face_up:
            rep = self.rank + self.suit
        else:
            rep = "XX"
        return rep
    def flip(self):
        self.is_face_up = not self.is_face_up

class Hand(object):
    """ A hand of playing cards. """
    def __init__(self):
        self.cards = []

    def __str__(self):
        if self.cards:
            rep = ""
            for card in self.cards:
                rep += str(card) + "\t"
        else:
            rep = "<empty>"
        return rep

    def clear(self):
        self.cards = []
    def add(self, card):
        self.cards.append(card)

    def give(self, card, other_hand):
        self.cards.remove(card)
        other_hand.add(card)

class Deck(Hand):
    """ A deck of playing cards. """
    def populate(self):
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                self.add(Card(rank, suit))

    def shuffle(self):
        import random
        random.shuffle(self.cards)
    def deal(self, hands, per_hand = 1):
        for rounds in range(per_hand):
            for hand in hands:
                if self.cards:
                    top_card = self.cards[0]
                    self.give(top_card, hand)
                else:
                    print("Out of cards!")

if __name__ == "__main__":
    print("This is a module for playing cards.")
    input("\n\nPress the enter key to exit.")
```

- To write the Blackjack game, this is the final cards module based on the Playing Cards programs.

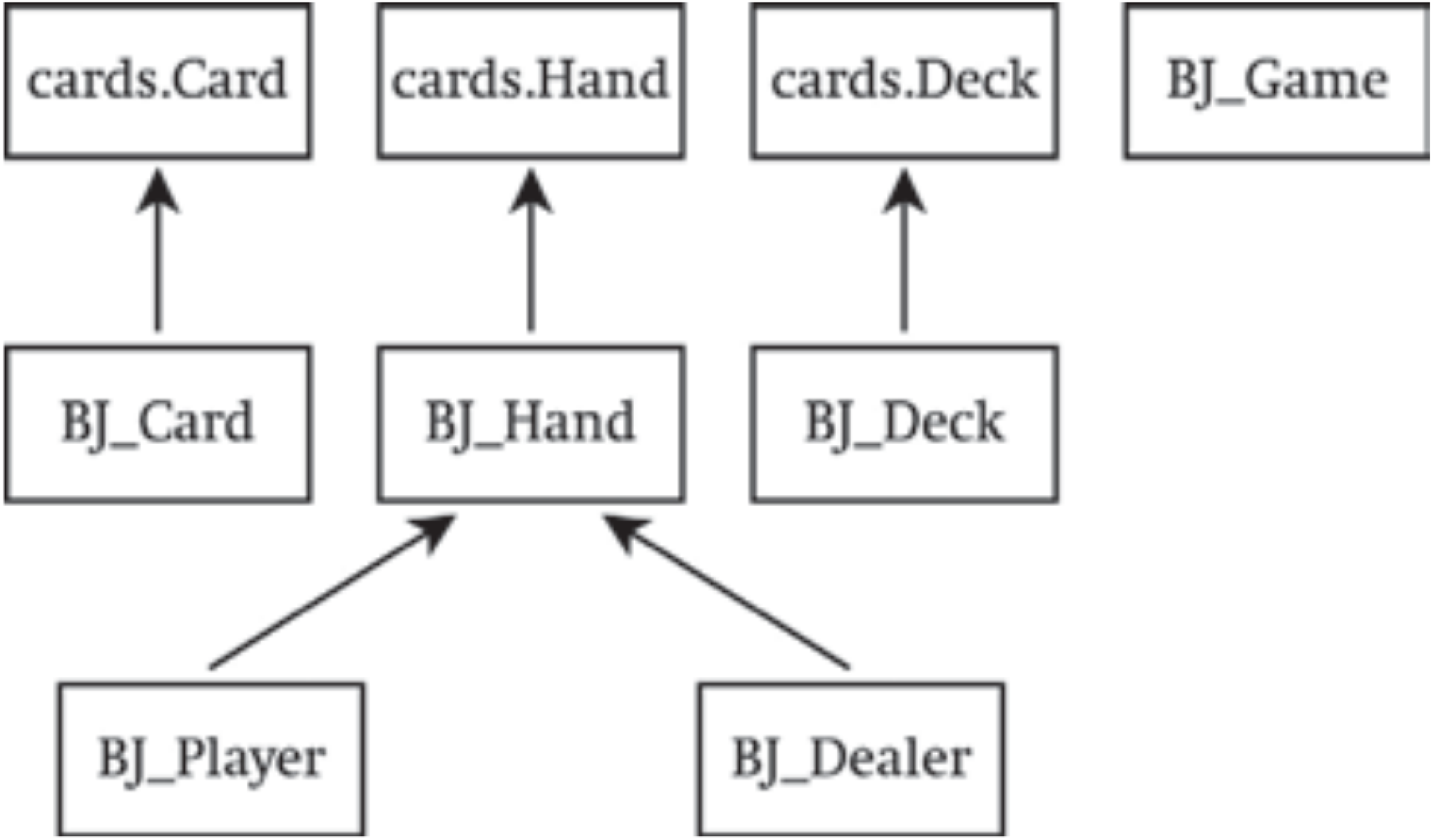
Designing the Classes

- Before you start coding a project with multiple classes, it can help to map them out on paper. You might make a list and include a brief description of each class.



Class	Base Class	Description
`BJ_Card`	`cards.Card`	A blackjack playing card. Define an attribute `value` to represent the point value of a card.
`BJ_Deck`	`cards.Deck`	A blackjack deck. A collection of `BJ_Card` objects.
`BJ_Hand`	`cards.Hand`	A blackjack hand. Define an attribute `total` to represent the point total of a hand. Define an attribute `name` to represent the owner of the hand.
`BJ_Player`	`BJ_Hand`	A blackjack player.
`BJ_Dealer`	`BJ_Hand`	A blackjack dealer.
`BJ_Game`	`object`	A blackjack game. Define an attribute `deck` for a `BJ_Deck` object. Define an attribute `dealer` for a `BJ_Dealer` object. Define an attribute `players` for a list of `BJ_Player` objects.

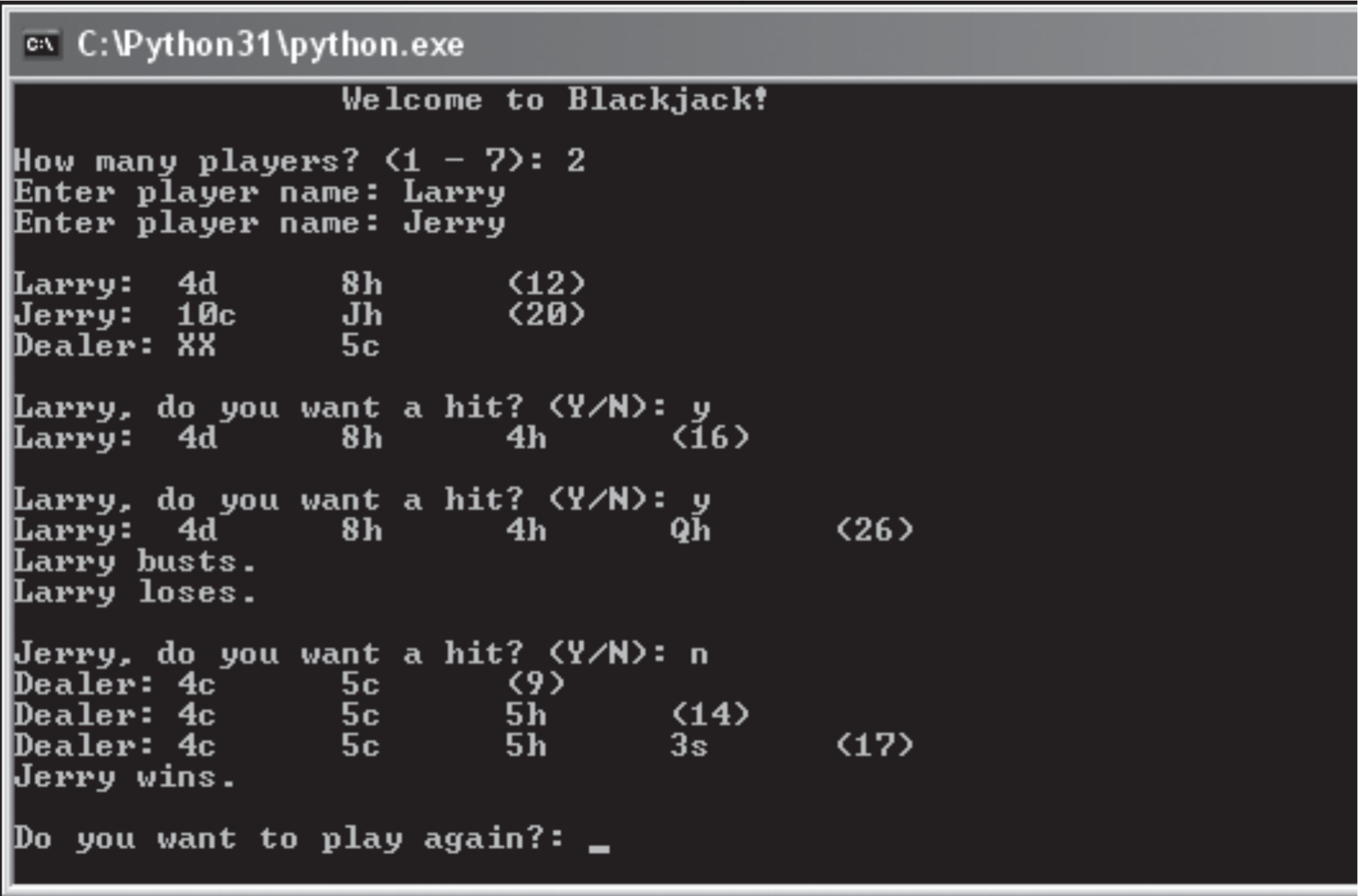
- In addition to describing your classes in words, you might want to draw a family tree of sorts to visualize how your classes are related:



Pseudocode for the Game Loop

```
Deal each player and dealer initial 2 cards
For each player
    While the player asks for a hit and is not busted
        Deal the player an additional card
If there are no players still playing
    Show the dealer’s 2 cards
Otherwise
    While the dealer must hit and the dealer is not busted
        Deal the dealer an additional card
    If the dealer is busted
        For each player who is still playing
            The player wins
    Otherwise
        For each player who is still playing
            If the player’s total > the dealer’s total
                The player wins
            Otherwise, if the player’s total< the dealer’s total
                The player loses
            Otherwise
                The player pushes
```

Introducing the Blackjack Game



```
                Welcome to Blackjack!

How many players? (1 - 7): 2
Enter player name: Larry
Enter player name: Jerry

Larry:  4d      8h      (12)
Jerry:  10c     Jh      (20)
Dealer: XX      5c

Larry, do you want a hit? (Y/N): y
Larry:  4d      8h      4h      (16)

Larry, do you want a hit? (Y/N): y
Larry:  4d      8h      4h      Qh      (26)
Larry busts.
Larry loses.

Jerry, do you want a hit? (Y/N): n
Dealer: 4c      5c      (9)
Dealer: 4c      5c      5h      (14)
Dealer: 4c      5c      5h      3s      (17)
Jerry wins.

Do you want to play again?: _
```

blackjack.py

```python
# Blackjack
# From 1 to 7 players compete against a dealer

import cards, games

class BJ_Card(cards.Card):
    """ A Blackjack Card. """
    ACE_VALUE = 1

    @property
    def value(self):
        if self.is_face_up:
            v = BJ_Card.RANKS.index(self.rank) + 1
            if v > 10:
                v = 10
        else:
            v = None
        return v
```

```
class BJ_Deck(cards.Deck):
    """ A Blackjack Deck. """
    def populate(self):
        for suit in BJ_Card.SUITS:
            for rank in BJ_Card.RANKS:
                self.cards.append(BJ_Card(rank, suit))

class BJ_Hand(cards.Hand):
    """ A Blackjack Hand. """
    def __init__(self, name):
        super(BJ_Hand, self).__init__()
        self.name = name

    def __str__(self):
        rep=self.name+":\t"+super(BJ_Hand,self).__str__()
        if self.total:
            rep += "(" + str(self.total) + ")"
        return rep
```

```
@property
def total(self):
    # if a card in the hand = None, then total = None
    for card in self.cards:
        if not card.value:
            return None
    t = 0     # add up card values, treat each Ace as 1
    for card in self.cards:
          t += card.value

    # determine if hand contains an Ace
    contains_ace = False
    for card in self.cards:
        if card.value == BJ_Card.ACE_VALUE:
            contains_ace = True

    # if  total is low, treat Ace = 11
    if contains_ace and t <= 11:
        t += 10   # add only 10 since we add 1 to Ace
    return t
    def is_busted(self):
        return self.total > 21

class BJ_Player(BJ_Hand):
    def is_hitting(self):
        response = games.ask_yes_no("\n" + self.name +
                                    ", do you want a hit? (Y/N): ")
        return response == "y"

    def bust(self):
        print(self.name, "busts.")
        self.lose()

    def lose(self):
        print(self.name, "loses.")

    def win(self):
        print(self.name, "wins.")

    def push(self):
        print(self.name, "pushes.")
```

```
class BJ_Dealer(BJ_Hand):
    """ A Blackjack Dealer. """
    def is_hitting(self):
        return self.total < 17

    def bust(self):
        print(self.name, "busts.")

    def flip_first_card(self):
        first_card = self.cards[0]
        first_card.flip()

class BJ_Game(object):
    """ A Blackjack Game. """
    def __init__(self, names):
        self.players = []
        for name in names:
            player = BJ_Player(name)
            self.players.append(player)
        self.dealer = BJ_Dealer("Dealer")

        self.deck = BJ_Deck()
        self.deck.populate()
        self.deck.shuffle()

    @property
    def still_playing(self):
        sp = []
        for player in self.players:
            if not player.is_busted():
                sp.append(player)
        return sp

    def __additional_cards(self, player):
        while not player.is_busted() and player.is_hitting():
            self.deck.deal([player])
            print(player)
            if player.is_busted():
                player.bust()
    def play(self):
        # deal initial 2 cards to everyone
        self.deck.deal(self.players + [self.dealer],
                       per_hand=2)
        self.dealer.flip_first_card()    # hide dealer 1st card
        for player in self.players:
            print(player)
        print(self.dealer)

        # deal additional cards to players
        for player in self.players:
            self.__additional_cards(player)

        self.dealer.flip_first_card()     # reveal dealer's first

        if not self.still_playing:
            # All players have busted, show the dealer's
            print(self.dealer)
        else:
            print(self.dealer)      # deal extra cards to dealer
            self.__additional_cards(self.dealer)
    if self.dealer.is_busted():
        # everyone still playing wins
        for player in self.still_playing:
            player.win()
    else:
        # compare the player still playing to dealer
        for player in self.still_playing:
            if player.total > self.dealer.total:
                player.win()
            elif player.total < self.dealer.total:
                player.lose()
            else:
                player.push()

# remove everyone's cards
for player in self.players:
    player.clear()
self.dealer.clear()
```

```
def main():
    print("\t\tWelcome to Blackjack!\n")

    names = []
    number = games.ask_number("How many players?,\
             “ (1 - 7): ", low = 1, high = 8)
    for i in range(number):
        name = input("Enter player name: ")
        names.append(name)
    print()

    game = BJ_Game(names)

    again = None
    while again != "n":
        game.play()
        again=games.ask_yes_no("\nWant to play again?:")

main()
input("\n\nPress the enter key to exit.")
```

Importing the cards and games Modules

- In the first part of the Blackjack (BJ) program, we import the 2 modules, cards and games:

```
import cards, games
```

- We created the games module in the Simple Game program, earlier in this chapter.

The BJ_Card Class

- The BJ_Card class extends the definition of what a card is by inheriting from cards.Card. In BJ_Card, we create a new property, value, for the point value of a card:

```
class BJ_Card(cards.Card):
    """ A Blackjack Card. """
    ACE_VALUE = 1

    @property
    def value(self):
        if self.is_face_up:
            v = BJ_Card.RANKS.index(self.rank) + 1
            if v > 10:
                v = 10
        else:
            v = None
        return v
```

- The method returns a number between 1 and 10, which represents the value of a blackjack card.

- The 1st part of the calculation is computed through

```
v = BJ_Card.RANKS.index(self.rank) + 1
```

- This expression takes rank of an object (say "6") and finds its corresponding index number in BJ_Card.RANKS through the list method index() (for "6" this would be 5).

- 1 is added to the result since the code starts counting at 0.

- since rank attributes of "J", "Q", and "K" result in numbers larger than 10, any value greater than 10 is set to 10.

- If an object's face_up attribute is False, this whole process is skipped and a value of None is returned.

The BJ_Deck Class

- The BJ_Deck class creates a deck of BJ cards. The class is almost exactly the same as its base class, cards.Deck

- The only difference is that we override cards.Deck‘s populate() so that a new BJ_Deck object gets populated with BJ_Card objects:

```
class BJ_Deck(cards.Deck):
    """ A Blackjack Deck. """
    def populate(self):
        for suit in BJ_Card.SUITS:
            for rank in BJ_Card.RANKS:
                self.cards.append(BJ_Card(rank, suit))
```

The BJ_Hand Class

- The BJ_Hand class, based on cards.Hand, is used for BJ hands. We override the cards.Hand constructor and add a name attribute to represent the the hand owner:

```
class BJ_Hand(cards.Hand):
    def __init__(self, name):
        super(BJ_Hand, self).__init__()
        self.name = name
```

- Override the inherited __str__() to display the total point value of the hand:

```
    def __str__(self):
        rep=self.name+":\t"+super(BJ_Hand,self).__str__()
        if self.total:
            rep += "(" + str(self.total) + ")"
        return rep
```

- We concatenate the object’s name with the string returned from cards.Hand.__str__() for the object.

- If the object’s total property isn’t None, we concatenate the string representation of the value of total.

- We then create a property called total, which represents the total point value of a BJ hand.

- If a BJ hand has a face-down card in it, then its total property is None.

- Otherwise, the value is calculated by adding the point values of all the cards in the hand:

```
@property
def total(self):
    # if a card in the hand = None, then total = None
    for card in self.cards:
        if not card.value:
            return None
    t = 0     # add up card values, treat each Ace as 1
    for card in self.cards:
          t += card.value

    # determine if hand contains an Ace
    contains_ace = False
    for card in self.cards:
        if card.value == BJ_Card.ACE_VALUE:
            contains_ace = True

    # if total is low, treat Ace = 11
    if contains_ace and t <= 11:
        t += 10   # add only 10 since we add 1 to Ace
    return t
```

- The 1st part of this method checks if any card in the BJ hand has a value equal to None (which means that the card is face-down). If so, the method returns None.

- The next part sums the point values of all the cards in the hand. The next part determines if the hand contains an ace. If so, the last part of the method determines if the card's point value should be 11 or 1.

- The last method in BJ_Hand is is_busted(). It returns True if the object's total > 21. Otherwise, it returns False:

```
def is_busted(self):
    return self.total > 21
```

- This kind of method, which returns either True or False, is used to represent a condition of an object with 2 possibilities , such as "on" or "off." It results in a more elegant method.

The BJ_Player Class

- The BJ_Player class, derived from BJ_Hand, is for BJ player:

```
class BJ_Player(BJ_Hand):
    def is_hitting(self):
        response = games.ask_yes_no("\n" + self.name +\
                            ", do you want a hit? (Y/N): ")
        return response == "y"

    def bust(self):
        print(self.name, "busts.")
        self.lose()

    def lose(self):
        print(self.name, "loses.")

    def win(self):
        print(self.name, "wins.")

    def push(self):
        print(self.name, "pushes.")
```

- is_hitting() returns True if the player wants another hit and returns False if the player doesn’t.

- bust() announces that a player busts and invokes the object’s lose(). lose() announces that a player loses.

- win() announces that a player wins. And push() announces that a player pushes.

- These simple methods form a great skeleton structure to handle the more complex issues that arise when players are allowed to bet.

The BJ_Dealer Class

- The BJ_Dealer class, derived from BJ_Hand, is used for the game’s BJ dealer:

```
class BJ_Dealer(BJ_Hand):
    def is_hitting(self):
        return self.total < 17

    def bust(self):
        print(self.name, "busts.")

    def flip_first_card(self):
        first_card = self.cards[0]
        first_card.flip()
```

- is_hitting() checks whether the dealer takes additional cards. Since a dealer must hit on any hand totaling 17 or less, the method returns True if the object’s total property is less than 17; otherwise, it returns False.

The BJ_Game Class

- The BJ_Game class is used to create a single object that represents a blackjack game.

- The mechanics of the game are complex enough that we create a few elements outside the method, including an __additional_cards() method that takes care of dealing additional cards to a player and a still_playing property that returns a list of all players still playing in the round.

The __init__() Method

The constructor receives a list of names and creates a player for each name, and also a dealer and a deck:

```
class BJ_Game(object):
    """ A Blackjack Game. """
    def __init__(self, names):
        self.players = []
        for name in names:
            player = BJ_Player(name)
            self.players.append(player)

        self.dealer = BJ_Dealer("Dealer")

        self.deck = BJ_Deck()
        self.deck.populate()
        self.deck.shuffle()
```

The still_playing Property

still_playing returns a list of all the players that are still playing (those that haven’t busted this round):

```
    @property
    def still_playing(self):
        sp = []
        for player in self.players:
            if not player.is_busted():
                sp.append(player)
        return sp
```

The __additional_cards() Method

- __additional_cards() deals additional cards to either a player or the dealer.

- The method receives an object into its player parameter, which can be either BJ_Player or BJ_Dealer. The method continues while the object’s is_busted() returns False and its is_hitting() returns True. If the object’s is_busted() returns True, then the object’s bust() is invoked:

```
def __additional_cards(self, player):
    while not player.is_busted() and player.is_hitting():
        self.deck.deal([player])
        print(player)
        if player.is_busted():
            player.bust()
```

- Polymorphism is at work here in 2 method calls. player.is_hitting() works equally well whether player refers to a BJ_Player object or a BJ_Dealer object.

- __additional_cards() never has to know which type of object it’s working with. The same is true in the line player.bust(), since BJ_Player and BJ_Dealer each defines its own bust().

The play() Method

- play() is where the game loop is defined and bears a resemblance to the earlier pseudocode (see the code).

- Each player and dealer is dealt the initial 2 cards. The dealer's 1st card is flipped to hide its value. Next, all of the hands are displayed. Then, each player is given cards as long as the player requests additional cards and hasn't busted.

- If all players have busted, the dealer's 1st card is flipped and the dealer's hand is printed. Otherwise, play continues.

- The dealer gets cards as long as the dealer hand total < 17. If the dealer busts, all remaining players win. Otherwise, each remaining player's hand is compared with the dealer's.

- If the player's total > the dealer's, the player wins. If the player's total is less, the player loses. If the two totals are equal, the player pushes.

The main() Function

main() gets the names of all the players, puts them in a list, and creates a BJ_Game object, using the list as an argument. Next, the function invokes the object’s play() and will continue to do so until the players no longer want to play:

```
def main():
    names = []
    number = games.ask_number("How many players?”,\
             “ (1 - 7): ", low = 1, high = 8)
    for i in range(number):
        name = input("Enter player name: ")
        names.append(name)
    print()
    game = BJ_Game(names)
    again = None
    while again != "n":
        game.play()
        again=games.ask_yes_no("\nWant to play again?:")
```