Lesson 34. Capstone project: card game

After reading lesson 34, you'll be able to

- Use classes to build a more complex program
- Use classes others have created to improve your program
- Allow users to play a simple version of the card game War

When you make your own object types, you can organize larger programs so that they're easier to write. The principles of modularity and abstraction introduced with functions also apply to classes. Classes are used to package a set of properties and behaviors common to a set of objects so that the objects can be used consistently in a program.

A common first program with classes is to simulate playing some sort of game with the user.

The problem

You want to simulate playing the card game War. Each round, players will take a card from one deck and compare the cards. The one with the higher card wins the round and gives their card to the other player. The winner is determined after numerous rounds, when the deck is empty. The winner is the person with fewer cards in their hand. You'll create two types of objects: a `Player` and a `CardDeck`. After defining the classes, you'll write code that simulates a game between two players. You'll first ask users for their names then create two `Player` objects. Both players will use the same card deck. Then you'll use methods defined in the `Player` and `CardDeck` classes to automatically simulate the rounds and determine the winner.

34.1. USING CLASSES THAT ALREADY EXIST

Objects built into the Python language are always there for you to use in your programs; these are objects such as `int`, `float`, `list`, and `dict`. But many other classes have already been written by other programmers and can be used to enhance the functionality of your program. Instead of typing their class definition in your code file, you can use an `import` statement to bring in the definition of another class into your file. This way, you can create objects of that type and use that class's methods in your code.

A useful class you'll want to use in your card game is the `random` class. You can bring in the random class definitions with this:

```
import random
```

Now you can create an object that can perform operations with random numbers. You use dot notation on the class name, as mentioned in lesson 31, and call the method you want to use along with any parameters it expects. For example,

```
r = random.random()
```

This gives you a random number between `0` (including) and `1` (not including) and binds it to the variable `r`. Here's another example:

```
r = random.randint(a, b)
```

This line gives you a random integer between `a` and `b` (including) and binds it to the variable `r`. Now consider this line:

It gives you a random element from a list `L` and binds it to the variable `r`.

34.2. DETAILING THE GAME RULES

The first step before beginning to code is to understand how you want your program to run, and what the specific game rules are:

- For simplicity, assume a deck contains four suits, each with cards 2 to 9. When denoting a card, use `"2H"` for the 2 of hearts, `"4D"` for the 4 of diamonds, `"7S"` for the 7 of spades, `"9C"` for the 9 of clubs, and so on.
- A player has a name (string) and a hand of cards (list).
- When the game begins, ask two players for their names and set them.
- Each round, add one card to each player's hand.
- Compare the cards just added to each player: first by the number, and then, if equal, by Spades > Hearts > Diamonds > Clubs.
- The person with the larger card removes the card from their hand, and the person with the smaller card takes the card and adds it to their hand.
- When the deck is empty, compare the number of cards the players have; the person with fewer cards wins.

You'll define two classes: one for a `Player` and one for a `CardDeck`.

34.3. DEFINING THE PLAYER CLASS

A player is defined by a name and a hand. The name is a string, and the hand is a list of strings, representing the cards. When you create a `Player` object, you give them a name as an argument and assume that they have no cards in their hand.

The first step is to define the `__init__` method to tell Python how to initialize a `Player` object. Knowing that you have two data attributes for a `Player` object, you can also write a method to return the name of the `Player`. This is shown in the following listing.

Listing 34.1. Definition for the `Player` class

```
class Player(object):
    """ a player """
    def __init__(self, name):
        """ sets the name and an empty hand """
        self.hand = []                              1
        self.name = name                            2
    def get_name(self):                             3
        """ Returns the name of the player """      3
        return self.name                            3
```

- ***1* Sets a hand to be an empty list**
- ***2* Sets the name to the string passed in when creating a Player object**
- ***3* A method to return the player's name**

Now, according to the game rules, a player can also add a card to their hand and remove a card from their hand. Notice that you check to make sure that the card added is a valid card by making sure its value is not `None`. To check the number of cards in players' hands and determine a winner, you can also add a method that tells you the number of cards in a hand. The following listing shows these three methods.

Listing 34.2. Definition for the `Player` class

```
class Player(object):
    """ a player """
    # methods from Listing 34.1
    def add_card_to_hand(self, card):
        """ card, a string
            Adds valid card to the player's hand """
        if card != None:
            self.hand.append(card)
    def remove_card_from_hand(self, card):
        """ card, a string
            Remove card from the player's hand """
        self.hand.remove(card)
    def hand_size(self):
        """ Returns the number of cards in player's h
        return len(self.hand)
```

- ***1* Adding a card to the hand adds it to the list, and adds only a card with a valid number and suit.**
- ***2* Removing a card from the hand finds the card and removes it from the list.**
- ***3* The size of the hand returns the number of elements in the list.**

A `CardDeck` class will represent a deck of cards. The deck has 32 cards, with the numbers 2 to 9 for each of the four deck types: spades, hearts, diamonds, and clubs. The following listing shows how to initialize the object type. There'll be only one data attribute, a list of all possible cards in the deck. Each card is denoted by a string; for example, of the form `"3H"` for the 3 of hearts.

Listing 34.3. Initialization for the `CardDeck` class

```
class CardDeck(object):
    """ A deck of cards 2-9 of spades, hearts, diamon
    def __init__(self):
        """ a deck of cards (strings e.g. "2C" for th
            contains all cards possible """
        hearts = "2H,3H,4H,5H,6H,7H,8H,9H"
        diamonds = "2D,3D,4D,5D,6D,7D,8D,9D"
        spades = "2S,3S,4S,5S,6S,7S,8S,9S"
        clubs = "2C,3C,4C,5C,6C,7C,8C,9C"
        self.deck = hearts.split(',')+diamonds.split(
                    spades.split(',')+clubs.split(','
```

- ***1*** **Makes a string of all possible cards in the deck**
- ***2*** **Splits the long string on the comma and adds all cards (strings) to a list for the deck**

After you decide that you'll represent a card deck with a list containing all cards in the deck, you can start to implement the methods for this class. This class will use the `random` class to pick a random card that a player will use. One method will return a random card from the deck; another method will compare two cards and tell you which one is higher.

Listing 34.4. Methods in the `CardDeck` class

```
import random

class CardDeck(object):
    """ A deck of cards 2-9 of spades, hearts, diamon
    def __init__(self):
        """ a deck of cards (strings e.g. "2C" for th
            contains all cards possible """
        hearts = "2H,3H,4H,5H,6H,7H,8H,9H"
        diamonds = "2D,3D,4D,5D,6D,7D,8D,9D"
        spades = "2S,3S,4S,5S,6S,7S,8S,9S"
        clubs = "2C,3C,4C,5C,6C,7C,8C,9C"
        self.deck = hearts.split(',')+diamonds.split(
                    + spades.split(',')+clubs.split('
    def get_card(self):
        """ Returns one random card (string) and
            returns None if there are no more cards "
        if len(self.deck) < 1:
            return None
        card = random.choice(self.deck)
        self.deck.remove(card)
        return card
    def compare_cards(self, card1, card2):
        """ returns the larger card according to
            (1) the larger of the numbers or, if equa
            (2) Spades > Hearts > Diamonds > Clubs ""
        if card1[0] > card2[0]:
            return card1
        elif card1[0] < card2[0]:
            return card2
        elif card1[1] > card2[1]:
            return card1
        else:
            return card2
```

- ***1*** **If there are no more cards in the deck, return None.**
- ***2*** **Picks a random card from the deck list**
- ***3*** **Removes the card from the deck list**
- ***4*** **Returns the value of the card (string)**
- ***5*** **Checks the card number value, returns the first card if it's higher**
- ***6*** **Checks the card number value, returns the second card if it's higher**
- ***7*** **When the card number value is equal, use the suit.**

34.5. SIMULATE THE CARD GAME

After you define object types to help you simulate a card game, you can write code that uses these types.

34.5.1. Setting up the objects

The first step is to set up the game by creating two `Player` objects and one

er object for each, and call the method to set the name. This is shown in the following listing.

Listing 34.5. Initializing game variables and objects

```
name1 = input("What's your name? Player 1: ")
player1 = Player(name1)

name2 = input("What's your name? Player 2: ")
player2 = Player(name2)

deck = CardDeck()
```

- ***1*** **Gets user input of the player 1 name**
- ***2*** **Makes a new Player object**
- ***3*** **Makes a new CardDeck object**

After initializing the object you'll use in the game, you can now simulate the game.

34.5.2. Simulating rounds in the game

A game consists of many rounds and continues until the deck is empty. It's possible to calculate the number of rounds players will play; if each player takes a card every round and there are 32 cards in the deck, there'll be 16 rounds. You could use a `for` loop to count the rounds, but a `while` loop is also an acceptable way of implementing the rounds.

In each round, each player gets a card, so call the `get_card` method on the deck twice, once for each player. Each player object then calls `add_card_to_hand`, which adds the random card returned from the deck to their own hand.

Then, both players will have at least a card, and there are two cases to consider:

- The game is over because the deck is empty.
- The deck still contains cards, and players must compare and decide who gives the other a card.

When the game is over, you check the sizes of the hands by calling `hand_size` on each player object. The player with the larger hand loses, and you exit from the loop.

If the game isn't over, you need to decide which player has the higher card by calling the `compare_cards` method on the deck with the two players' cards. The returned value is the higher card, and if the number values are equal, the suit decides which card weighs more. If the higher card is the same as `player1`'s card, `player1` needs to give the card to `player2`. In code, this translates to `player1` calling `remove_card_from_hand` and `player2` calling `add_card_to_hand`. A similar situation happens when the larger card is the same as `player2`'s card. See the following listing.

Listing 34.6. Loop to simulate rounds in the game

```
name1 = input("What's your name? Player 1: ")
player1 = Player(name1)
name2 = input("What's your name? Player 2: ")
player2 = Player(name2)
deck = CardDeck()

while True:
    player1_card = deck.get_card()
    player2_card = deck.get_card()
    player1.add_card_to_hand(player1_card)
    player2.add_card_to_hand(player2_card)

    if player1_card == None or player2_card == None:
        print("Game Over. No more cards in deck.")
        print(name1, " has ", player1.hand_size())
        print(name2, " has ", player2.hand_size())
        print("Who won?")
        if player1.hand_size() > player2.hand_size():
            print(name2, " wins!")
        elif player1.hand_size() < player2.hand_size(
            print(name1, " wins!")
        else:
            print("A Tie!")
        break

    else:
        print(name1, ": ", player1_card)
        print(name2, ": ", player2_card)
        if deck.compare_cards(player1_card,player2_ca
            player2.add_card_to_hand(player1_card)
            player1.remove_card_from_hand(player1_car
        else:
```

- **1 Game over because at least one player has no more cards**
- **2 Checks the sizes of the hands, and player2 wins because they have fewer cards**
- **3 Checks the sizes of the hands, and player1 wins because they have fewer cards**
- **4 Players have the same number of cards, so a tie**
- **5 The break exits the while loop when one player wins or there's a tie.**
- **6 Game can continue because there are still cards to compare between hands**
- **7 Compares cards between players, and the returned card is the higher card**
- **8 Higher card belongs to player1, so add player1's card to player2's hand**
- **9 Higher card belongs to player1, so remove player1's card from their hand.**

34.6. MODULARITY AND ABSTRACTION WITH CLASSES

Implementing this game is a large undertaking. Without breaking the problem into smaller subtasks, coding the game would quickly become messy.

Using objects and object-oriented programming, you've also managed to modularize your program even further. You separated your code into different objects and gave each object a set of data attributes and a set of methods.

Using object-oriented programming also allowed you to separate two main ideas: creating classes that organize your code, and using the classes to implement code that plays the game. While simulating the gameplay, you were able to use the objects of the same type consistently, leading to neat and easy-to-read code. This abstracted the details of how the object types and their methods were implemented, and you were able to use the docstrings of methods to decide which methods were appropriate during the simulation.

SUMMARY

In this lesson, my objective was to teach you how to write a larger program that uses classes others have created to improve your program, and how to create your own classes and use them to play a game.

The code for the class definitions needs to be written only once. It dictates the overall properties of your objects and operations you can do with these objects. This code doesn't manipulate any specific objects. The code for the gameplay itself (the code not including the class definitions) is straightforward, because you're creating objects and calling methods on the appropriate objects.

This structure separates code that tells others what an object is and what it can do, from code that uses these objects to achieve various tasks. In this way, you're hiding some of the unnecessary coding details that you don't need to know in order to implement the gameplay.