Think Python

Exercise 18.1.

For the following program, draw a UML class diagram that shows these classes and the relationships among them.

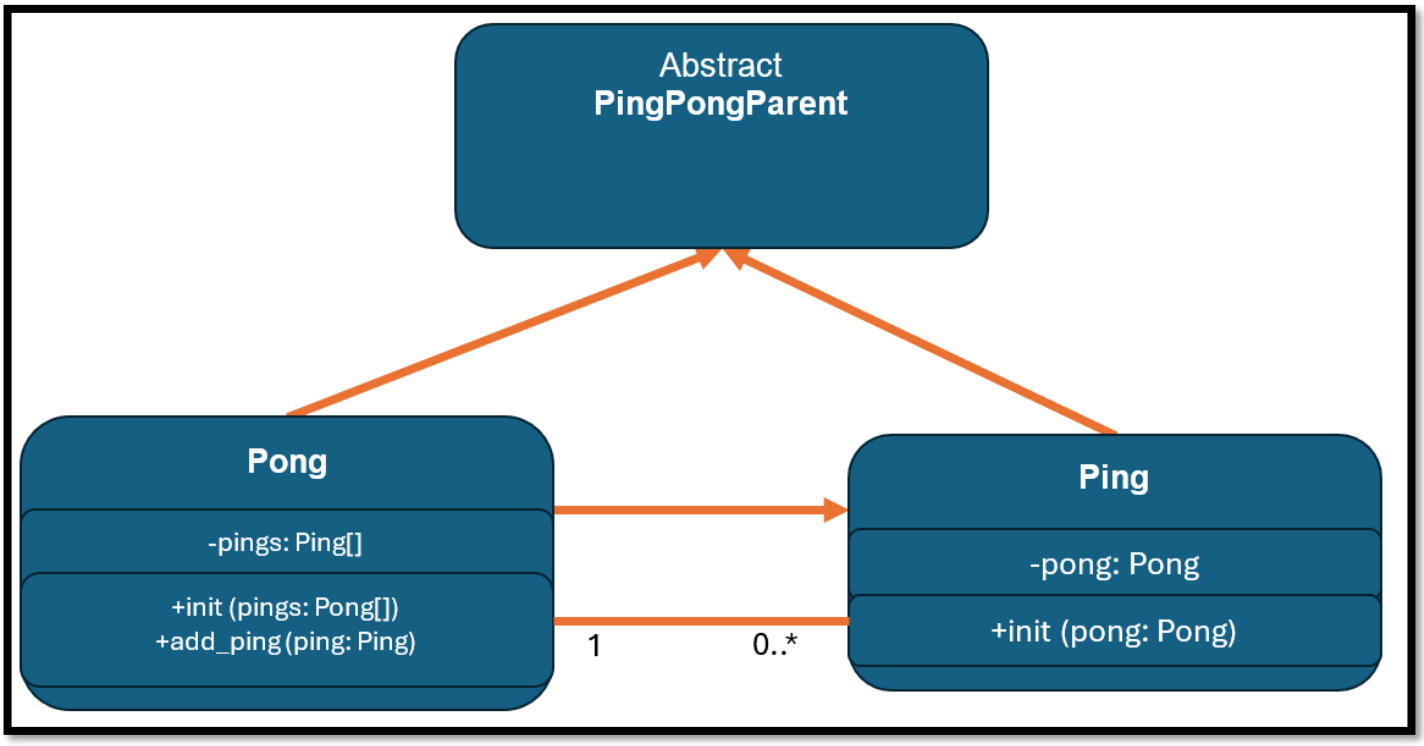
```
class PingPongParent:
    pass

class Ping(PingPongParent):
    def __init__(self, pong):
        self.pong = pong

class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
          self.pings = []
        else:
          self.pings = pings

    def add_ping(self, ping):
        self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```



Exercise 18.2

Write a Deck method called deal_hands that takes two parameters, the number of hands and the number of cards per hand. It should create the appropriate number of Hand objects, deal the appropriate number of cards per hand, and return a list of Hands.

This solution assumes a deck and a hand class.

```
def deal_hands(self, num_hands, num_cards_per_hand):
    """
    Deals the specified number of hands with the given number of cards per hand.

    :param deck: The Deck object to draw cards from.
    :param num_hands: The number of hands to deal.
    :param num_cards_per_hand: The number of cards per hand.
    :return: A list of Hand objects, each representing a dealt hand.
    """

    hands = []
    for _ in range(num_hands):
       hand = Hand()
       for _ in range(num_cards_per_hand):
          hand.add_card(deck.draw_card())
       hands.append(hand)
    return hands
```

Exercise 18.3

The goal of these exercises is to estimate the probability of drawing these various hands.

1. Download the following files from https: // thinkpython. com/ code : Card.py : A complete version of the Card, Deck and Hand classes in this chapter. PokerHand.py : An incomplete implementation of a class that represents a poker hand, and some code that tests it.

2. If you run PokerHand.py, it deals seven 7-card poker hands and checks to see if any of them contains a flush. Read this code carefully before you go on.

3. Add methods to PokerHand.py named has_pair, has_twopair, etc. that return True or False according to whether or not the hand meets the relevant criteria. Your code should work correctly for “hands” that contain any number of cards (although 5 and 7 are the most common sizes).

4. Write a method named classify that figures out the highest-value classification for a hand and sets the label attribute accordingly. For example, a 7-card hand might contain a flush and a pair; it should be labelled “flush”.

5. When you are convinced that your classification methods are working, the next step is to estimate the probabilities of the various hands. Write a function in PokerHand.py that shuffles a deck of cards, divides it into hands, classifies the hands, and counts the number of times various classifications appear.

6. Print a table of the classifications and their probabilities. Run your program with larger and larger numbers of hands until the output values converge to a reasonable degree of accuracy.

Possible Hands	Poker Hand Rankings
pair: two cards with the same rank **two pair:** two pairs of cards with the same rank **three of a kind:** three cards with the same rank **straight:** five cards with ranks in sequence (aces can be high or low, so Ace-2-3-4-5 is a straight and so is 10-Jack-Queen-King-Ace, but Queen-King-Ace-2-3 is not.) **flush:** five cards with the same suit **full house:** three cards with one rank, two cards with another **four of a kind:** four cards with the same rank **straight flush:** five cards in sequence (as defined above) and with the same suit	**Straight Flush:** Five cards in sequence, all of the same suit. **Four of a Kind:** Four cards of the same rank. **Full House:** Three cards of one rank, two cards of another. **Flush:** Five cards of the same suit (not in sequence). **Straight:** Five cards in sequence (not all the same suit). **Three of a Kind:** Three cards of the same rank. **Two Pair:** Two pairs of cards with the same rank. **Pair:** Two cards of the same rank. **High Card:** None of the above, so the highest card in the hand determines the classification.

Exercise 18.3

```
from Card import Hand, Deck
import random


class PokerHand(Hand):
    """Represents a poker hand."""

    def suit_hist(self):
        """Builds a histogram of the suits that appear in the hand."""
        self.suits = {}
        for card in self.cards:
            self.suits[card.suit] = self.suits.get(card.suit, 0) + 1

    def rank_hist(self):
        """Builds a histogram of the ranks that appear in the hand."""
        self.ranks = {}
        for card in self.cards:
            self.ranks[card.rank] = self.ranks.get(card.rank, 0) + 1

    def has_flush(self):
        """Returns True if the hand has a flush."""
        self.suit_hist()
        for val in self.suits.values():
            if val >= 5:
                return True
        return False

    def has_pair(self):
        """Returns True if the hand has a pair."""
        self.rank_hist()
        for count in self.ranks.values():
            if count >= 2:
                return True
        return False

    def has_two_pair(self):
        """Returns True if the hand has two pairs."""
        self.rank_hist()
        pairs = 0
        for count in self.ranks.values():
            if count >= 2:
                pairs += 1
        return pairs >= 2

    def has_three_of_a_kind(self):
        """Returns True if the hand has three of a kind."""
        self.rank_hist()
        for count in self.ranks.values():
            if count >= 3:
                return True
        return False

    def has_straight(self):
        """Returns True if the hand has a straight."""
        self.rank_hist()
        rank_list = sorted(self.ranks.keys())
        # Account for Ace being low in Ace-2-3-4-5
        if 14 in rank_list:
            rank_list.append(1)
        for i in range(len(rank_list) - 4):
            if (rank_list[i + 4] - rank_list[i]) == 4:
                return True
        return False
```

```
    def has_full_house(self):
        """Returns True if the hand has a full house."""
        self.rank_hist()
        has_three = False
        has_pair = False
        for count in self.ranks.values():
            if count >= 3:
                has_three = True
            elif count >= 2:
                has_pair = True
        return has_three and has_pair

    def has_four_of_a_kind(self):
        """Returns True if the hand has four of a kind."""
        self.rank_hist()
        for count in self.ranks.values():
            if count >= 4:
                return True
        return False

    def has_straight_flush(self):
        """Returns True if the hand has a straight flush."""
        if not self.has_flush():
            return False

        # Group cards by suit
        self.suit_hist()
        suit_cards = {suit: [] for suit in self.suits}
        for card in self.cards:
            suit_cards[card.suit].append(card.rank)

        # Check for a straight in each suit
        for ranks in suit_cards.values():
            ranks = sorted(set(ranks))
            if 14 in ranks:
                ranks.append(1)  # Account for Ace being low
            for i in range(len(ranks) - 4):
                if (ranks[i + 4] - ranks[i]) == 4:
                    return True
        return False

    def classify(self):
        """Classifies the hand and sets the label to the highest-value
classification."""
        if self.has_straight_flush():
            self.label = 'Straight Flush'
        elif self.has_four_of_a_kind():
            self.label = 'Four of a Kind'
        elif self.has_full_house():
            self.label = 'Full House'
        elif self.has_flush():
            self.label = 'Flush'
        elif self.has_straight():
            self.label = 'Straight'
        elif self.has_three_of_a_kind():
            self.label = 'Three of a Kind'
        elif self.has_two_pair():
            self.label = 'Two Pair'
        elif self.has_pair():
            self.label = 'Pair'
        else:
            self.label = 'High Card'


def estimate_hand_probabilities(num_simulations=10000):
  """Estimates the probabilities of various poker hands."""
  hand_counts = {
```

```
        'Straight Flush': 0,
        'Four of a Kind': 0,
        'Full House': 0,
        'Flush': 0,
        'Straight': 0,
        'Three of a Kind': 0,
        'Two Pair': 0,
        'Pair': 0,
        'High Card': 0
    }

    # Simulate the poker hands
    for _ in range(num_simulations):
        deck = Deck()  # Reinitialize the deck in each simulation
        deck.shuffle()

        # Deal a 7-card hand (can be changed to 5 cards for different simulations)
        hand = PokerHand()
        deck.move_cards(hand, 7)  # Move 7 cards to the hand
        hand.sort()

        # Classify the hand and update the count
        hand.classify()
        hand_counts[hand.label] += 1

    # Calculate probabilities
    probabilities = {label: count / num_simulations for label, count in
hand_counts.items()}

    return probabilities



if __name__ == '__main__':
    # Estimate hand probabilities with 10000 simulations
    probabilities = estimate_hand_probabilities(num_simulations=10000)

    # Print the results in a readable format
    print("Poker Hand Probabilities (based on 10,000 simulations):")
    for hand, prob in probabilities.items():
        print(f"{hand}: {prob:.4f}")
```

Poker Hand Probabilities (based on 10,000 simulations):

Straight Flush: 0.0003

Four of a Kind: 0.0015

Full House: 0.0235

Flush: 0.0322

Straight: 0.0448

Three of a Kind: 0.0492

Two Pair: 0.2310

Pair: 0.4461

High Card: 0.1714