Stats 21 - HW 5

 Problems have been adapted from the exercises in Think Python 2nd Ed by Allen B. Downey.

The questions have been entered into this document. You will modify the document by entering your code.

Make sure you run the cell so the requested output is visible. Download the finished document as a PDF.

You will submit:

- the rendered PDF file to Gradescope
- this ipynb file with your answers to CCLE

Reading

- Chapters 15 to 18

Please do the reading. The chapters are short.

Exercise 15.1

Write a definition for a class named Circle with attributes center and radius, where center is a Point object and radius is a number.

Instantiate a Circle object that represents a circle with its center at (150, 100) and radius 75.

Write a function named `point_in_circle` that takes a Circle and a Point and returns True if the Point lies in or on the boundary of the circle.

Write a function named `rect_in_circle` that takes a Circle and a Rectangle and returns True if the Rectangle lies entirely in or on the boundary of the circle.

Write a function named `rect_circle_overlap` that takes a Circle and a Rectangle and returns True if any of the corners of the Rectangle fall inside the circle.

In [1]:
```
# no need to modify this code
class Point:
    """Represents a point in 2-D space.
    attributes: x, y
    """

def print_point(p):
    print('(%g, %g)' % (p.x, p.y))


class Rectangle:
    """Represents a rectangle.
    attributes: width, height, corner.
    """
```

In [2]:
```
class Circle:
    """represents a circle
    attributes: center, radius"""
```

```
def point_in_circle(point, circle):
    dist = (point.x - circle.center.x)^2 + (point.y - circle.center.y)^2
    return dist < circle.radius

def rect_in_circle(rectangle, circle):
    if point_in_circle(rectangle.corner, circle):
        opp_corner = rectangle.corner
        opp_corner.x += rectangle.width
        opp_corner.y += rectangle.height
        if point_in_circle(opp_corner, circle):
            return True
    return False

def rect_circle_overlap(rectangle, circle):
    if point_in_circle(rectangle.corner, circle):
        return True
    corner2 = rectangle.corner
    corner2.x += rectangle.width
    if point_in_circle(corner2, circle):
        return True
    corner3 = rectangle.corner
    corner3.x += rectangle.width
    corner3.y += rectangle.height
    if point_in_circle(corner3, circle):
        return True
    corner4 = rectangle.corner
    corner4.y += rectangle.height
    if point_in_circle(corner4, circle):
        return True
    return False
```

Create a test case.

Create a Rectangle called box. It has a width of 100 and a height of 200. It's corner is the Point (50, 50).

Print out the vars of box.

Create a Circle. The center is located at the Ponit (150, 100). It has a radius of 75.

- Run the function to test if `box.corner` is in the `circle`.
- Run the function to test if `box` is in the `circle`.
- Run the function to test if `box` and `circle` overlap.

In [3]:
```
# your code
box = Rectangle()
box.width = 100
box.height = 200
box.corner = Point()
box.corner.x = 50
box.corner.y = 50
circle = Circle()
circle.center = Point()
circle.center.x = 150
circle.center.y = 100
circle.radius = 75
```

In [4]:
```
print(point_in_circle(box.corner, circle))
print(rect_in_circle(box, circle))
print(rect_circle_overlap(box, circle))
```

```
False
False
True
```

Exercise 16.1

Write a function called `mul_time` (multiply time) that takes a Time object and a number and returns a new Time object that contains the product of the original Time and the number.

In [5]:
```
# code that defines Time class and some functions needed for 16.1
# no need to modify
class Time:
    """Represents the time of day.

    attributes: hour, minute, second
    """
def print_time(t):
    """Prints a string representation of the time.

    t: Time object
    """
    print('%.2d:%.2d:%.2d' % (t.hour, t.minute, t.second))
def int_to_time(seconds):
    """Makes a new Time object.

    seconds: int seconds since midnight.
    """
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
def time_to_int(time):
    """Computes the number of seconds since midnight.

    time: Time object.
    """
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

In [6]:
```
# write your function here
def mul_time(time, multiplier):
    return int_to_time(time_to_int(time) * multiplier)
```

The following test case takes a race time and tries to calculate the running pace.

In [7]:
```
# test case:
race_time = Time()
race_time.hour = 1
race_time.minute = 34
race_time.second = 5

print('Half marathon time', end=' ')
print_time(race_time)

distance = 13.1 # miles
pace = mul_time(race_time, 1/distance)
print('Pace', end=' ')
print_time(pace)
```

```
Half marathon time 01:34:05
Pace 00:07:10
```

Exercise 16.2.

The `datetime` module provides time objects that are similar to the Time objects in this chapter, but they provide a rich set of methods and operators. Read the documentation at

https://docs.python.org/3/library/datetime.html

1. Use the datetime module and write a few lines that gets the current date and prints the day of the week.

In [8]:
```
import datetime
```

In [9]:
```
# example usage
new_date = datetime.date(2021, 5, 19)
print(new_date)
```

```
2021-05-19
```

In [10]:
```
today = datetime.date.today()
weekdays = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday")
today_of_week = weekdays[datetime.date.weekday(today)]
print(today_of_week)
```

```
Thursday
```

1. Write a function that takes a birthday as input and prints the user's age and the number of days, hours, minutes and seconds until their next birthday (the day starts at midnight).

In [11]:
```
birthdate = "12/25/1999" # month/day/year
```

In [12]:
```
# print time until birthday
def birthday_until(birthdate):
    split_birthdate = birthdate.split("/")
    birthdate = datetime.datetime(int(split_birthdate[2]), int(split_birthdate[0]), int(split_bir
    today = datetime.datetime.now()
    age = today - birthdate
    birthday = birthdate.replace(year = today.year)
    until = birthday - today
    if until.days <= 0:
        birthday = birthdate.replace(year = today.year + 1)
    until = birthday - today
    print(f"age: {age}")
    print(f"{until} until next birthday")
```

In [13]:
```
birthday_until(birthdate)
```

```
age: 8377 days, 23:09:49.723771
23 days, 0:50:10.276229 until next birthday
```

In [14]:
```
birthdate2 = "3/26/1972"
```

In [15]:
```
# print time until birthday
birthday_until(birthdate2)
```

```
age: 18512 days, 23:09:49.753771
114 days, 0:50:10.246229 until next birthday
```

1. For two people born on different days, there is a day when one is exactly twice as old as the other. That's their Double Day. Write a function that takes two birth dates and computes their Double Day. The function should also print the age of person1 in years, months, days as well as the age of person 2 in years, months, days.

In [16]:
```
person1 = "12/25/1999"
person2 = "4/15/1970"
```

In [17]:
```
def double_day(day1, day2):
    import math
    split_day1 = day1.split("/")
    split_day2 = day2.split("/")
    day1 = datetime.date(int(split_day1[2]), int(split_day1[0]), int(split_day1[1]))
    day2 = datetime.date(int(split_day2[2]), int(split_day2[0]), int(split_day2[1]))
    today = datetime.date.today()
    age1 = today - day1
    age1_days = age1.days
    age1_years, age1_days = divmod(age1_days, 365.25)
    age1_months, age1_days = divmod(age1_days, 30.4167)
    age1_years = int(age1_years)
    age1_months = int(age1_months)
    age1_days = int(age1_days)
    age2 = today - day2
    age2_days = age2.days
    age2_years, age2_days = divmod(age2_days, 365.25)
    age2_months, age2_days = divmod(age2_days, 30.4167)
    age2_years = int(age2_years)
    age2_months = int(age2_months)
    age2_days = int(age2_days)
    diff = day1 - day2
    if diff.days < 0:
        day1, day2 = day2, day1
        diff = day1 - day2
    print(f"double day: {day1 + diff}")
    print(f"person1 age: {age1_years} years, {age1_months} months, {age1_days} days old")
    print(f"person2 age: {age2_years} years, {age2_months} months, {age2_days} days old")
```

In [18]:
```
double_day(person1, person2)
```

```
double day: 2029-09-04
person1 age: 22 years, 11 months, 6 days old
person2 age: 52 years, 7 months, 17 days old
```

In [19]:
```
# test case
person1 = "3/26/1972"
person2 = "1/20/1985"
double_day(person1, person2)
```

```
double day: 1997-11-16
person1 age: 50 years, 8 months, 6 days old
person2 age: 37 years, 10 months, 10 days old
```

In [20]:
```
# test case
person1 = "11/9/2001"
person2 = "3/23/2010"
double_day(person1, person2)
```

```
double day: 2018-08-04
person1 age: 21 years, 0 months, 21 days old
person2 age: 12 years, 8 months, 9 days old
```

Exercise 17.1.

I have included the code from chapter 17.

Change the attributes of the `Time` class to be a single integer representing seconds since midnight. Then modify the methods (and the function int_to_time) to work with the new implementation.

You should not have to modify the test code in the function `main()`. When you are done, the output should be the same as before.

In [21]:
```
# Leave this code unchanged
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)

    def print_time(self):
        print(str(self))

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def __radd__(self, other):
        return self.__add__(other)

    def add_time(self, other):
        assert self.is_valid() and other.is_valid()
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)

    def is_valid(self):
        if self.hour < 0 or self.minute < 0 or self.second < 0:
            return False
        if self.minute >= 60 or self.second >= 60:
            return False
        return True


def int_to_time(seconds):
    minutes, second = divmod(seconds, 60)
    hour, minute = divmod(minutes, 60)
    time = Time(hour, minute, second)
    return time


def main():
    start = Time(9, 45, 00)
    start.print_time()

    end = start.increment(1337)
    #end = start.increment(1337, 460)
    end.print_time()
```

```
    print('Is end after start?')
    print(end.is_after(start))

    print('Using __str__')
    print(start, end)

    start = Time(9, 45)
    duration = Time(1, 35)
    print(start + duration)
    print(start + 1337)
    print(1337 + start)

    print('Example of polymorphism')
    t1 = Time(7, 43)
    t2 = Time(7, 41)
    t3 = Time(7, 37)
    total = sum([t1, t2, t3])
    print(total)
```

In [22]:
```
# results of a few time tests. your later results should match these
main()
```

```
09:45:00
10:07:17
Is end after start?
True
Using __str__
09:45:00 10:07:17
11:20:00
10:07:17
10:07:17
Example of polymorphism
23:01:00
```

In [23]:
```
# modify this class
# you can only have one attribute: self.second
# the time is still initialized with hour, minute, second

class Time:
    def __init__(self, hour=0, minute=0, second=0):
        second = second + (minute * 60) + (hour * 360)
        self.second = second

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.second // 360, (self.second % 360) // 60, self.second % 6

    def print_time(self):
        print(str(self))

    def time_to_int(self):
        return self.seconds

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def __radd__(self, other):
        return self.__add__(other)

    def add_time(self, other):
        assert self.is_valid() and other.is_valid()
```

```
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)

    def is_valid(self):
        if self.second < 0:
            return False
        return True
```

Exercise 17.2

This exercise is a cautionary tale about one of the most common and difficult to find errors in Python.

We create a definition for a class named Kangaroo with the following methods:

1. An **init** method that initializes an attribute named pouch_contents to an empty list.
2. A method named `put_in_pouch` that takes an object of any type and adds it to pouch_contents.
3. A **str** method that returns a string representation of the Kangaroo object and the contents of the pouch.

Test your code by creating two Kangaroo objects, assigning them to variables named kanga and roo, and then adding roo to the contents of kanga's pouch.

You don't actually have to write any code for this exercise. Instead, read through the included code and answer the questions.

In [24]:
```
# `Badkangaroo.py`
class Kangaroo:
    """A Kangaroo is a marsupial."""

    def __init__(self, name, contents=[]):
        """Initialize the pouch contents.

        name: string
        contents: initial pouch contents.
        """
        self.name = name
        self.pouch_contents = contents

    def __str__(self):
        """Return a string representaion of this Kangaroo.
        """
        t = [ self.name + ' has pouch contents:' ]
        for obj in self.pouch_contents:
            s = '    ' + object.__str__(obj)
            t.append(s)
        return '\n'.join(t)

    def put_in_pouch(self, item):
        """Adds a new item to the pouch contents.

        item: object to be added
        """
        self.pouch_contents.append(item)
```

In [25]:
```
kanga = Kangaroo('Kanga')
roo = Kangaroo('Roo')
kanga.put_in_pouch('wallet')
kanga.put_in_pouch('car keys')
```

```
roo.put_in_pouch('candy')
kanga.put_in_pouch(roo)
```

In [26]:
```
print(kanga)
```

```
Kanga has pouch contents:
    'wallet'
    'car keys'
    'candy'
    <__main__.Kangaroo object at 0x0000026A2321EA00>
```

In [27]:
```
print(roo)
```

```
Roo has pouch contents:
    'wallet'
    'car keys'
    'candy'
    <__main__.Kangaroo object at 0x0000026A2321EA00>
```

Question: Why does roo and kanga have the same contents?

Your answer: When using the function `put_in_pouch()`, the new item is appended to the existing list of items. Since the default is a pointer to an empty list, the new items are appended to that specific list. Every instance of Kangaroo has the same pointer to the same default list, so putting an item into one Kangaroo's pouch puts it in every Kangaroo's pouch.

In [28]:
```
# `GoodKangaroo.py`
class Kangaroo:
    """A Kangaroo is a marsupial."""

    def __init__(self, name, contents=[]):
        """Initialize the pouch contents.

        name: string
        contents: initial pouch contents.
        """
        # The problem is the default value for contents.
        # Default values get evaluated ONCE, when the function
        # is defined; they don't get evaluated again when the
        # function is called.

        # In this case that means that when __init__ is defined,
        # [] gets evaluated and contents gets a reference to
        # an empty list.

        # After that, every Kangaroo that gets the default
        # value gets a reference to THE SAME list.  If any
        # Kangaroo modifies this shared list, they all see
        # the change.

        # The next version of __init__ shows an idiomatic way
        # to avoid this problem.
        self.name = name
        self.pouch_contents = contents

    def __init__(self, name, contents=None):
        """Initialize the pouch contents.

        name: string
        contents: initial pouch contents.
        """
        # In this version, the default value is None.  When
        # __init__ runs, it checks the value of contents and,
        # if necessary, creates a new empty list.  That way,
        # every Kangaroo that gets the default value gets a
```

```
        # reference to a different list.

        # As a general rule, you should avoid using a mutable
        # object as a default value, unless you really know
        # what you are doing.
        self.name = name
        if contents == None:
            contents = []
        self.pouch_contents = contents

    def __str__(self):
        """Return a string representaion of this Kangaroo.
        """
        t = [ self.name + ' has pouch contents:' ]
        for obj in self.pouch_contents:
            s = '    ' + object.__str__(obj)
            t.append(s)
        return '\n'.join(t)

    def put_in_pouch(self, item):
        """Adds a new item to the pouch contents.

        item: object to be added
        """
        self.pouch_contents.append(item)
```

In [29]:
```
kanga = Kangaroo('Kanga')
roo = Kangaroo('Roo')
kanga.put_in_pouch('wallet')
kanga.put_in_pouch('car keys')
roo.put_in_pouch('candy')
kanga.put_in_pouch(roo)
```

In [30]:
```
print(kanga)
```

```
Kanga has pouch contents:
    'wallet'
    'car keys'
    <__main__.Kangaroo object at 0x0000026A232109D0>
```

In [31]:
```
print(roo)
```

```
Roo has pouch contents:
    'candy'
```

Question: How does the `goodkangaroo` version fix the issue?

Your answer: `goodkangaroo` does not use one single pointer for the default value. Setting the default value to `None` means that every time a new Kangaroo object is created, if it defaults to an empty list, it will point to a new empty list rather than the same list that may have been modified.

Exercise 18.3

The following are the possible hands in poker, in increasing order of value and decreasing order of probability:

- 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: fove cards in sequence (as defined above) and with the same suit

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

In [32]:
```
# no need to change this code block
## Card.py : A complete version of the Card, Deck and Hand classes
## in chapter 18.

import random

class Card:
    """Represents a standard playing card.

    Attributes:
      suit: integer 0-3
      rank: integer 1-13
    """

    suit_names = ["Clubs", "Diamonds", "Hearts", "Spades"]
    rank_names = [None, "Ace", "2", "3", "4", "5", "6", "7",
              "8", "9", "10", "Jack", "Queen", "King"]

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        """Returns a human-readable string representation."""
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])

    def __eq__(self, other):
        """Checks whether self and other have the same rank and suit.

        returns: boolean
        """
        return self.suit == other.suit and self.rank == other.rank

    def __lt__(self, other):
        """Compares this card to other, first by suit, then rank.

        returns: boolean
        """
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2


class Deck:
    """Represents a deck of cards.

    Attributes:
      cards: list of Card objects.
    """

    def __init__(self):
        """Initializes the Deck with 52 cards.
        """
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
```

```
                card = Card(suit, rank)
                self.cards.append(card)

    def __str__(self):
        """Returns a string representation of the deck.
        """
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)

    def add_card(self, card):
        """Adds a card to the deck.

        card: Card
        """
        self.cards.append(card)

    def remove_card(self, card):
        """Removes a card from the deck or raises exception if it is not there.

        card: Card
        """
        self.cards.remove(card)

    def pop_card(self, i=-1):
        """Removes and returns a card from the deck.

        i: index of the card to pop; by default, pops the last card.
        """
        return self.cards.pop(i)

    def shuffle(self):
        """Shuffles the cards in this deck."""
        random.shuffle(self.cards)

    def sort(self):
        """Sorts the cards in ascending order."""
        self.cards.sort()

    def move_cards(self, hand, num):
        """Moves the given number of cards from the deck into the Hand.

        hand: destination Hand object
        num: integer number of cards to move
        """
        for i in range(num):
            hand.add_card(self.pop_card())


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

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


def find_defining_class(obj, method_name):
    """Finds and returns the class object that will provide
    the definition of method_name (as a string) if it is
    invoked on obj.

    obj: any python object
    method_name: string method name
    """
```

```
    for ty in type(obj).mro():
        if method_name in ty.__dict__:
            return ty
    return None
```

In [33]:
```
# no need to change this code block
## PokerHand.py : An incomplete implementation of a class that represents a poker hand, and
## some code that tests it.
class PokerHand(Hand):
    """Represents a poker hand."""

    # all_labels is a list of all the labels in order from highest rank
    # to lowest rank
    all_labels = ['straightflush', 'fourkind', 'fullhouse', 'flush',
                  'straight', 'threekind', 'twopair', 'pair', 'highcard']

    def suit_hist(self):
        """Builds a histogram of the suits that appear in the hand.

        Stores the result in attribute suits.
        """
        self.suits = {}
        for card in self.cards:
            self.suits[card.suit] = self.suits.get(card.suit, 0) + 1

    def has_flush(self):
        """Returns True if the hand has a flush, False otherwise.

        Note that this works correctly for hands with more than 5 cards.
        """
        self.suit_hist()
        for val in self.suits.values():
            if val >= 5:
                return True
        return False
```

If you run the following cell, 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.

In [34]:
```
# no need to change this code block
# make a deck
deck = Deck()
deck.shuffle()

# deal the cards and classify the hands
for i in range(7):
    hand = PokerHand()
    deck.move_cards(hand, 7)
    hand.sort()
    print(hand)
    print(hand.has_flush())
    print('')
```

9 of Clubs
4 of Diamonds
5 of Diamonds
8 of Diamonds
Jack of Diamonds
King of Hearts
4 of Spades
False

6 of Clubs
8 of Clubs
5 of Hearts
Ace of Spades
8 of Spades
10 of Spades
Queen of Spades
False

Ace of Diamonds
3 of Diamonds
King of Diamonds
8 of Hearts
Jack of Hearts
3 of Spades
7 of Spades
False

3 of Clubs
4 of Clubs
7 of Clubs
9 of Diamonds
3 of Hearts
9 of Hearts
6 of Spades
False

Jack of Clubs
Queen of Clubs
6 of Diamonds
Queen of Diamonds
2 of Hearts
4 of Hearts
10 of Hearts
False

Ace of Clubs
King of Clubs
10 of Diamonds
Ace of Hearts
6 of Hearts
7 of Hearts
2 of Spades
False

2 of Clubs
2 of Diamonds
7 of Diamonds
Queen of Hearts
5 of Spades
9 of Spades
King of Spades
False

1. Add methods to `class PokerHand` 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).

2. Write a method named `classify` that figures out the classifications for a hand and creates a list of labels accordingly. For example, a 7-card hand might contain a flush and a pair. It will create an attribute `labels` which is a list `["flush", "pair"]

In [35]:
```
# fix this code here
class PokerHand(Hand):
    """Represents a poker hand."""

    # all_labels is a list of all the labels in order from highest rank
    # to lowest rank
    all_labels = ['straightflush', 'fourkind', 'fullhouse', 'flush',
                  'straight', 'threekind', 'twopair', 'pair', 'highcard']

    def suit_hist(self):
        """Builds a histogram of the suits that appear in the hand.

        Stores the result in attribute suits.
        """
        self.suits = {}
        for card in self.cards:
            self.suits[card.suit] = self.suits.get(card.suit, 0) + 1

    def has_flush(self):
        """Returns True if the hand has a flush, False otherwise.

        Note that this works correctly for hands with more than 5 cards.
        """
        self.suit_hist()
        for val in self.suits.values():
            if val >= 5:
                return True
        return False

    def has_highcard(self):
        return True

    def has_pair(self):
        ranks = {}
        for card in self.cards:
            if card.rank in ranks:
                return True
            ranks[card.rank] = 0
        return False

    def has_twopair(self):
        ranks = {}
        for card in self.cards:
            ranks[card.rank] = ranks.get(card.rank, 0) + 1
        for key in ranks:
            if ranks[key] >= 4:
                return True
            if ranks[key] >= 2:
                newranks = ranks.copy()
                newranks.pop(key)
                for newkey in newranks:
                    if newranks[newkey] >= 2:
                        return True
        return False
```

```
    def has_threekind(self):
        ranks = {}
        for card in self.cards:
            ranks[card.rank] = ranks.get(card.rank, 0) + 1
        for key in ranks:
            if ranks[key] >= 3:
                return True
        return False

    def has_fourkind(self):
        ranks = {}
        for card in self.cards:
            ranks[card.rank] = ranks.get(card.rank, 0) + 1
        for key in ranks:
            if ranks[key] >= 4:
                return True
        return False

    def has_fullhouse(self):
        ranks = {}
        for card in self.cards:
            ranks[card.rank] = ranks.get(card.rank, 0) + 1
        for key in ranks:
            if ranks[key] >= 3:
                newranks = ranks.copy()
                newranks.pop(key)
                for newkey in newranks:
                    if newranks[newkey] >= 2:
                        return True
        return False

    def has_straight(self):
        ranks = []
        ordered = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 1]
        for card in self.cards:
            if card.rank not in ranks:
                ranks.append(card.rank)
            sorted = ranks.sort()
        if len(ranks) >= 5:
            n = len(ranks) - 4
            for i in range(n):
                straight = True
                first = ranks[i]
                check = ordered[first:first + 5]
                for j in check:
                    if j not in ranks:
                        straight = False
                if straight:
                    return True
        return False

    def in_a_row(self, ranks, n=5):
        unique_ranks = []
        ordered = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 1]
        for card in self.cards:
            if card.rank not in unique_ranks:
                unique_ranks.append(card.rank)
            sorted = unique_ranks.sort()
        if len(unique_ranks) >= n:
            for i in range(len(ranks)):
                in_row = True
                first = ranks[i]
                check = ordered[first:first + n]
                for j in check:
                    if j not in ranks:
```

```
                    in_row = False
            if in_row:
                return True
        return False

    def has_straightflush(self):
        if not self.has_flush():
            return False
        possible = {}
        self.suit_hist()
        for key in self.suits:
            if self.suits[key] >= 5:
                possible[key] = self.suits[key]
        ranks = {}
        for card in self.cards:
            if card.suit in possible:
                ranks[card.suit] = [ranks.get(card.suit), card.rank]
        for i in ranks:
            straight = True
            suiterank = ranks[i]
            n = len(suiterank) - 4
            for i in range(n):
                straight = True
                first = suiterank[i]
                check = ordered[first:first + 5]
                for j in check:
                    if j not in ranks:
                        straight = False
                if straight:
                    return True
        return False

    def classify(self):
        self.labels = ['highcard']
        if self.has_flush():
            self.labels.append('flush')
        if self.has_pair():
            self.labels.append('self')
        if self.has_twopair():
            self.labels.append('two pair')
        if self.has_threekind():
            self.labels.append('three of a kind')
        if self.has_fourkind():
            self.labels.append('four of a kind')
        if self.has_fullhouse():
            self.labels.append('full house')
        if self.has_straight():
            self.labels.append('straight')
        if self.has_straightflush():
            self.labels.append('straight flush')
```

1. When you are convinced that your classification methods are working, the next step is to estimate the probabilities of the various hands.

Use the following functions that will shuffle a deck of cards, divides it into hands, classifies the hands, and counts the number of times various classifications appear.

In [36]:
```
# no need to change this code block
class PokerDeck(Deck):
    """Represents a deck of cards that can deal poker hands."""

    def deal_hands(self, num_cards=5, num_hands=10):
        """Deals hands from the deck and returns Hands.
```

```
        num_cards: cards per hand
        num_hands: number of hands

        returns: list of Hands
        """
        hands = []
        for i in range(num_hands):
            hand = PokerHand()
            self.move_cards(hand, num_cards)
            hand.classify()
            hands.append(hand)
        return hands
```

In [37]:
```
# no need to change this code block
class Hist(dict):
    """A map from each item (x) to its frequency."""

    def __init__(self, seq=[]):
        "Creates a new histogram starting with the items in seq."
        for x in seq:
            self.count(x)

    def count(self, x, f=1):
        "Increments (or decrements) the counter associated with item x."
        self[x] = self.get(x, 0) + f
        if self[x] == 0:
            del self[x]
```

In [38]:
```
# test code. no need to modify
def main():
    # the label histogram: map from label to number of occurances
    lhist = Hist()

    # loop n times, dealing 7 hands per iteration, 7 cards each
    n = 10000
    for i in range(n):
        if i % 1000 == 0:
            print(i)

        deck = PokerDeck()
        deck.shuffle()

        hands = deck.deal_hands(7, 7)
        for hand in hands:
            for label in hand.labels:
                lhist.count(label)

    # print the results
    total = 7.0 * n
    print(total, 'hands dealt:')

    for label in PokerHand.all_labels:
        freq = lhist.get(label, 0)
        if freq == 0:
            continue
        p = total / freq
        print('%s happens one time in %.2f' % (label, p))
```

In [39]:
```
# test code
main()
```

```
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
70000.0 hands dealt:
flush happens one time in 32.27
straight happens one time in 22.95
highcard happens one time in 1.00
```