# Stats 21 - HW 5

Homework copyright Miles Chen. 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 [ ]:
```
# 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 [ ]:

```
class Circle:
    """represents a circle
    attributes: center, radius"""
def __init__(self, center, radius):
    self.center = Point(center)
    self.radius = radius

def point_in_circle(p, c):
    return ((p.x - c.center.x)**2 + (p.y-c.center.y)**2) <= c.radius**2

def rect_in_circle(r, c):
    tr, bl, br = Point(), Point(), Point()
    tr.x = r.corner.x + r.width
    tr.y = r.corner.y
    bl.x = r.corner.x
    bl.y = r.corner.y - r.height
    br.x = r.corner.x + r.width
    br.y = r.corner.y - r.height
    if (point_in_circle(r.corner, c) and point_in_circle(tr, c) and
         point_in_circle(bl, c) and point_in_circle(br, c)):
        return True
    else:
        return False

def rect_circle_overlap(r, c):
    tr, bl, br = Point(), Point(), Point()
    tr.x = r.corner.x + r.width
    tr.y = r.corner.y
    bl.x = r.corner.x
    bl.y = r.corner.y - r.height
    br.x = r.corner.x + r.width
    br.y = r.corner.y - r.height
    if (point_in_circle(r.corner, c) or point_in_circle(tr, c) or
         point_in_circle(bl, c) or point_in_circle(br, c)):
        return True
    else:
        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 [ ]:
```
# your code
box = Rectangle()
box.width = 100
box.height = 200
box.corner = Point()
box.corner.x = 50
box.corner.y = 50


circle = Circle()
circle.radius = 75
circle.center = Point()
circle.center.x = 150
circle.center.y = 100
```

In [ ]:
```
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 [ ]:
```
# 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 [ ]:
```
# write your function here
def mul_time(time, num):
    time_int = time_to_int(time) * num
    new_time = int_to_time(time_int)
    return new_time
```

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

In [ ]:
```
# 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 [ ]:
```
import datetime
```

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

```
print(new_date)
```

```
2021-05-19
```

In [ ]:
```
days = {'0':'Monday', '1':'Tuesday', '2':'Wednesday', '3':'Thursday', '4':'F
```

In [ ]:
```
today = datetime.datetime.now()
print(days[str(today.weekday())])
```

```
Tuesday
```

2. 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 [ ]:
```
birthdate = "12/25/1999" # month/day/year
```

In [ ]:
```
def leap_year(year):
    return (year % 400 == 0) or ((year % 100 != 0) and (year % 4 == 0))

def time_between(date1, date2):
    months = [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    y_diff, m_diff, d_diff = date2.year - date1.year, date2.month - date1.mo
    if d_diff < 0:
        m_diff -= 1
        d_diff += months[date1.month]
        if leap_year(date1.year) and date1.month == 2:
            d_diff += 1
    if m_diff < 0:
        y_diff -= 1
        m_diff += 12
    return (y_diff, m_diff, d_diff)
```

In [ ]:
```
def birthday(birthdate):
    bday, curr = datetime.datetime.strptime(birthdate, "%m/%d/%Y"), datetime
    yr = curr.year
    age = time_between(bday, curr)
    nums = []
    birth = birthdate.split('/')
    for x in birth:
        nums.append(int(x))
    if nums[1] < curr.day or nums[0] < curr.month:
        yr += 1
    next = datetime.datetime(yr, nums[0], nums[1])
    print(age[0], "years old")
    print((next-curr), "until next birthday")
```

In [ ]:
```
birthday(birthdate)
```

```
23 years old
18 days, 8:36:23.585747 until next birthday
```

In [ ]:
```
birthdate2 = "3/26/1972"
birthday(birthdate2)
```

```
51 years old
110 days, 8:36:22.306784 until next birthday
```

3. 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 [ ]:
```
person1 = "12/25/1999"
person2 = "4/15/1970"
```

In [ ]:
```
def double_day(day1, day2):
    d1, d2 = datetime.datetime.strptime(day1, "%m/%d/%Y"), datetime.datetime
    twice = max(d1, d2) + abs(d1 - d2)
    print("Double Date: %02d/%02d/%02d" % (twice.month, twice.day, twice.yea
    print("Person 1 is: %d Years, %d Months, %d Days old" % time_between(d1,
    print("Person 2 is: %d Years, %d Months, %d Days old" % time_between(d2,
```

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

```
Double Date: 09/04/2029
Person 1 is: 29 Years, 8 Months, 10 Days old
Person 2 is: 59 Years, 4 Months, 19 Days old
```

In [ ]:
```
# test case
person1 = "1/06/1985"
person2 = "1/05/1985"
double_day(person1, person2)
```

```
Double Date: 01/07/1985
Person 1 is: 0 Years, 0 Months, 1 Days old
Person 2 is: 0 Years, 0 Months, 2 Days old
```

In [ ]:
```
# test case
person1 = "6/10/2004"
person2 = "7/31/2004"
double_day(person1, person2)
```

```
Double Date: 09/20/2004
Person 1 is: 0 Years, 3 Months, 10 Days old
Person 2 is: 0 Years, 1 Months, 20 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 [ ]:
```
# 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 [ ]:
```
# 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 [ ]:
```
# 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):
        self.second = (((hour*60)+minute)*60) + second

    def __str__(self):
        minutes, seconds = divmod(self.second, 60)
        hours, minutes = divmod(minutes, 60)
        return '%.2d:%.2d:%.2d' % (hours, minutes, seconds)

    def print_time(self):
        print(str(self))
    def time_to_int(self):
        return self.second

    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.second + other.second
        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 == 84600:
            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 [ ]:
```
# `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 [ ]:
```
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 [ ]:
```
print(kanga)
```

```
Kanga has pouch contents:
    'wallet'
    'car keys'
    'candy'
    <__main__.Kangaroo object at 0x106588eb0>
```

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

```
Roo has pouch contents:
    'wallet'
    'car keys'
    'candy'
    <__main__.Kangaroo object at 0x106588eb0>
```

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

Your answer: The Kangaroo class is initialized with contents as an empty list, so any Kangaroo that is initialized with the default values gets a reference to the same empty list, which is why they have the same contents.

In [ ]:
```python
# `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 [ ]:
```
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 [ ]:
```
print(kanga)
```

```
Kanga has pouch contents:
    'wallet'
    'car keys'
    <__main__.Kangaroo object at 0x106589b20>
```

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

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

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

Your answer: This version, when necessary, creates a new list every time, so that every time the Kanagroo is created with default value, a different list is called.

## Exercise 18.3 (the hard one)

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 [ ]:
```
# Do not change this code block
## 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:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
    def __len__(self):
        return len(self.cards)

    def __getitem__(self, position):
        return self.cards[position]

    def __setitem__(self, key, value):
        self.cards[key] = value

    def shuffle(self):
        random.shuffle(self.cards)

    def pop_card(self):
        return self.cards.pop()

    def add_card(self, card):
        self.cards.append(card)

    def sort(self):
         self.cards.sort()

    def move_cards(self, hand, num):
        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 [ ]:

```
# Do not change this code block
## PokerHand.py : An incomplete implementation of a class that represents a
## some code that tests it. The current definition of has_flush may or may r
## for the final solution.
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 two 7-card poker hands and checks to see if any of them contains a flush. Read this code carefully before you go on.

In [ ]:
```
# 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(2):
    hand = PokerHand()
    deck.move_cards(hand, 7)
    hand.sort()
    print(hand)
    print(hand.has_flush())
    print('')
```

```
8 of Clubs
4 of Diamonds
6 of Hearts
8 of Hearts
10 of Hearts
2 of Spades
6 of Spades
False

5 of Clubs
9 of Clubs
Jack of Clubs
Queen of Diamonds
King of Hearts
Ace of Spades
9 of Spades
False
```

In [ ]:

```
# no need to change this code block
# This code chunk creates a hand,
# adds seven cards to it, 5 of which are diamonds
# it checks to see if a flush exists and returns True
hand = PokerHand()
hand.add_card(Card(1,1))
hand.add_card(Card(1,3))
hand.add_card(Card(1,13))
hand.add_card(Card(1,12))
hand.add_card(Card(1,6))
hand.add_card(Card(2,3))
hand.add_card(Card(0,7))
hand.sort()
print(hand)
print(hand.has_flush())
```

```
7 of Clubs
Ace of Diamonds
3 of Diamonds
6 of Diamonds
Queen of Diamonds
King of Diamonds
3 of Hearts
True
```

3. 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).

4. Write a method named `classify` that figures out the classifications for a hand and labels it accordingly. The solution should only classify the hand that is most valuable.

For example, a 7-card hand might contain sets of three of a kind (e.g, A, A, A, J, J, J, 3). This hand has three of a kind. `has_threekind` should return True. `has_twopair` and `has_pair` will also return True. However, the most valuable poker hand that can be created with these cards is a full house (three of a kind and pair: A, A, A, J, J). The classify method will return the label "fullhouse".

Clarifications:

- The label "highcard" should only be used for hands that do not have a pair or any higher possible classification.
- Ace can count as both the low and high in a straight. e.g. A-2-3-4-5 is a straight. 10-J-Q-K-A is also a straight

Within `PokerHand`, the method `suit_hist` has been created as a helper function. You may create other helper functions as you see fit.

In [ ]:
```
# Your work:
# Modify the code in this code chunk
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 rank_hist(self):
        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, False otherwise.

        Note that this works correctly for hands with more than 5 cards.
        """
        self.suit_hist()
        for num in self.suits.values():
            if num >= 5:
                return True
        return False
    def has_straightflush(self):
        self.suit_hist()
        for i in range(0, 4):
            if (not self.suits.get(i) == None) and self.suits.get(i) >= 5:
                flush = PokerHand()
                for card in self.cards:
                    if card.suit == i:
                        flush.add_card(card)
                if flush.has_straight():
                    return True
        return False
    def has_fourkind(self):
        self.rank_hist()
        for num in self.ranks.values():
            if num == 4:
                return True
        return False
    def has_fullhouse(self):
        self.rank_hist()
        if self.ranks[max(self.ranks, key = self.ranks.get)] >= 3:
            self.ranks.pop(max(self.ranks, key = self.ranks.get))
            if self.ranks[max(self.ranks, key = self.ranks.get)] >= 2:
                return True
        return False
    def has_straight(self):
        self.rank_hist()
        if 1 in self.ranks.keys():
            self.ranks[14] = 1
        for card in self.cards:
            count = 0
            for i in range(5):
                if card.rank + i in self.ranks.keys():
                    count += 1
            if count >= 5:
                return True
        return False
    def has_threekind(self):
        self.rank_hist()
        for num in self.ranks.values():
            if num >= 3:
                return True
        return False
    def has_twopair(self):
        self.rank_hist()
        count = 0
        for num in self.ranks.values():
            if num == 2:
                count += 1
        if count >= 2:
            return True
        return False
    def has_pair(self):
        self.rank_hist()
        for num in self.ranks.values():
            if num >= 2:
                return True
        return False
    def classify(self):
        classification = 'highcard'
        if self.has_straightflush():
            classification = 'straightflush'
        elif self.has_fourkind():
            classification = 'fourkind'
        elif self.has_fullhouse():
            classification = 'fullhouse'
        elif self.has_flush():
            classification = 'flush'
        elif self.has_straight():
            classification = 'straight'
        elif self.has_threekind():
            classification = 'threekind'
        elif self.has_twopair():
            classification = 'twopair'
        elif self.has_pair():
            classification = 'pair'
        self.label = classification
        return hand
```

## Test case: full house

In [ ]:
```
# test case; do not modify
hand = PokerHand()
hand.add_card(Card(0,1))
hand.add_card(Card(1,1))
hand.add_card(Card(2,1))
hand.add_card(Card(0,11))
hand.add_card(Card(1,11))
hand.add_card(Card(2,11))
hand.add_card(Card(0,3))
hand.classify()
print(hand)
print(hand.label) # full house
```

```
Ace of Clubs
Ace of Diamonds
Ace of Hearts
Jack of Clubs
Jack of Diamonds
Jack of Hearts
3 of Clubs
fullhouse
```

## Test case: straight flush

In [ ]:
```
# test case; do not modify
hand = PokerHand()
hand.add_card(Card(0,1))
hand.add_card(Card(0,2))
hand.add_card(Card(0,3))
hand.add_card(Card(0,4))
hand.add_card(Card(0,5))
hand.add_card(Card(1,5))
hand.add_card(Card(2,5))
hand.classify()
```

```
print(hand)
print(hand.label) # straight flush
```

```
Ace of Clubs
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
5 of Diamonds
5 of Hearts
straightflush
```

In [ ]:
```
# test case; do not modify
hand = PokerHand()
hand.add_card(Card(0,1))
hand.add_card(Card(0,13))
hand.add_card(Card(0,12))
hand.add_card(Card(0,11))
hand.add_card(Card(0,10))
hand.add_card(Card(1,11))
hand.add_card(Card(2,12))
hand.classify()
print(hand)
print(hand.label) # straight flush
```

```
Ace of Clubs
King of Clubs
Queen of Clubs
Jack of Clubs
10 of Clubs
Jack of Diamonds
Queen of Hearts
straightflush
```

## Test case: straight

In [ ]:
```
# test case; do not modify
hand = PokerHand()
hand.add_card(Card(0,2))
hand.add_card(Card(0,3))
hand.add_card(Card(1,4))
hand.add_card(Card(2,5))
hand.add_card(Card(1,2))
hand.add_card(Card(3,6))
hand.add_card(Card(2,6))
hand.classify()
print(hand)
print(hand.label) # straight
```

```
2 of Clubs
3 of Clubs
4 of Diamonds
5 of Hearts
2 of Diamonds
6 of Spades
6 of Hearts
straight
```

## Test case: straight

In [ ]:
```
# test case; do not modify
hand = PokerHand()
hand.add_card(Card(0,2))
hand.add_card(Card(0,3))
hand.add_card(Card(2,5))
hand.add_card(Card(0,10))
hand.add_card(Card(1,10))
hand.add_card(Card(1,4))
hand.add_card(Card(0,6))
hand.classify()
print(hand)
print(hand.label) # straight
```

```
2 of Clubs
3 of Clubs
5 of Hearts
10 of Clubs
10 of Diamonds
4 of Diamonds
6 of Clubs
straight
```

## Test case: flush (contains a straight and a flush, but is not straight flush)

In [ ]:
```
# test case; do not modify
hand = PokerHand()
hand.add_card(Card(0,2))
hand.add_card(Card(0,3))
hand.add_card(Card(0,4))
hand.add_card(Card(0,5))
hand.add_card(Card(1,6))
hand.add_card(Card(1,7))
hand.add_card(Card(0,8))
hand.classify()
print(hand)
print(hand.label) # flush
```

```
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Diamonds
7 of Diamonds
8 of Clubs
flush
```

## Test case: two pair

In [ ]:

```
# test case; do not modify
hand = PokerHand()
hand.add_card(Card(0,2))
hand.add_card(Card(1,2))
hand.add_card(Card(0,4))
hand.add_card(Card(1,4))
hand.add_card(Card(0,5))
hand.add_card(Card(1,5))
hand.add_card(Card(0,6))
hand.classify()
print(hand)
print(hand.label) # two pair
```

```
2 of Clubs
2 of Diamonds
4 of Clubs
4 of Diamonds
5 of Clubs
5 of Diamonds
6 of Clubs
twopair
```

5. 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 [ ]:

```
# 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=7, num_hands=7):
        """Deals hands from the deck and returns Hands. The hands are classi
        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 [ ]:
```
# 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 [ ]:
```
# test code. no need to modify
def main():
    # the label histogram: map from label to number of occurances
    labelhist = Hist()

    # loop n times, dealing 5 hands per iteration, 7 cards each
    n = 20000
    for i in range(n):
        if i % 1000 == 0:
            print(i)

        deck = PokerDeck()
        deck.shuffle()

        hands = deck.deal_hands(7, 5)
        for hand in hands:
            labelhist.count(hand.label)

    # print the results
    total = 5.0 * n
    print(total, 'hands dealt:')

    for label in PokerHand.all_labels:
        freq = labelhist.get(label, 0)
        p = 100 * freq / total
        if freq == 0:
            odds = float('inf')
        else:
            odds = (total - freq) / freq
        print('{:} happens with probability {:.3f}%; odds against: {:.2f} :
```

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

```
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
15000
16000
17000
18000
19000
100000.0 hands dealt:
straightflush happens with probability 0.024%; odds against: 4165.67 : 1
fourkind happens with probability 0.171%; odds against: 583.80 : 1
fullhouse happens with probability 2.607%; odds against: 37.36 : 1
flush happens with probability 3.059%; odds against: 31.69 : 1
straight happens with probability 4.711%; odds against: 20.23 : 1
threekind happens with probability 4.862%; odds against: 19.57 : 1
twopair happens with probability 23.679%; odds against: 3.22 : 1
pair happens with probability 43.588%; odds against: 1.29 : 1
highcard happens with probability 17.299%; odds against: 4.78 : 1
```

The following list of probabilities can serve as a guide to see if you are on track. https://en.wikipedia.org/wiki/Poker_probability#7-card_poker_hands