Homework 7

Intro to Programming (Term I/2021–22)

built on 2021/11/25 at 11:39:57 | ***due:** fri dec 10 @ 11:59pm*

This assignment will give you practice writing larger, more complex pieces of code. In this assignment, you will solve a number of programming puzzles and hand them in.

Be sure to read this problem set thoroughly, especially the sections related to collaboration and the hand-in procedure.

Overview:

Problem	File Name
1.	`allperm.py`
2.	`cards.py`
3.	`statspeak.py`

Problem	File Name
4.	`auction.py`
5.	`textcol.py`

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Be sure to indicate who you have worked with (refer to the hand-in instructions).

Logistics

We're using a script to grade your submission before any human being looks at it. Sadly, the script is not as forgiving as we are. *So, make sure you follow the instructions strictly.* It's a bad omen when the course staff has to manually recover your file because the script doesn't like it. Hence:

- Save your work in a file as described in the task description. This will be different for each task. **Do not save your file(s) with names other than specified.**
- Before handing anything in, you should thoroughly test everything you write.
- You will upload each file to our submission site `https://assn.cs.muzoo.io/` before the due date. Please use your SKY credentials to log into the submission site. Note that you can submit multiple times but only the latest version will be graded.
- For some task, you will be able to verify your submission online. Please do so as it checks if your solution is gradable or not. Passing verification does not mean that your solution is correct, but, at least, it passes our preliminary check.
- At the beginning of each of your solution files, write down the number of hours (roughly) you spent on that particular task, and the names of the people you collaborated with as comments. As an example, each of your files should look like this:

```
# Assignment XX, Task YY
# Name: Eye Loveprograming
# Collaborators: John Nonexistent
# Time Spent: 4:00 hrs

... your real program continues here ...
```

- The course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online on Canvas discussion.

Task 1: The Set of All Permutations (10 points)

For this task, save your work in `allperm.py`

Remember the `set` data type? Well, you can look it up in Python's documentation.

In this task, you will implement a *recursive* function `all_perm(n: int) -> Set[Tuple[int, ...]]` that takes an integer `n` > 0 and returns a set containing all the permutations of $1, 2, 3, \ldots, n$. Each permutation must be represented as a **tuple**. For example:

- `all_perm(1) == {(1,)}`
- `all_perm(2) == {(1,2), (2,1)}`
- `all_perm(3) == {(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)}`

Restrictions: Do not write a helper function. Your function must be recursive, and must work with sets and tuples directly. You are not allowed to import anything.

Hints: Consider how we can use `all_perm(2)` to create the answer for `all_perm(3)`. Perhaps the following diagram will help (pay close attention to the color coding and numbers in boldface fonts):

$$
\begin{aligned}
\texttt{all\_perm(2)} &== \{(1,2),(2,1)\}\\
\texttt{all\_perm(3)} &== \{(\mathbf{3},1,2),(1,\mathbf{3},2),(1,2,\mathbf{3}),(\mathbf{3},2,1),(2,\mathbf{3},1),(2,1,\mathbf{3})\}\\
\texttt{all\_perm(4)} &== \{(\mathbf{4},3,1,2),(3,\mathbf{4},1,2),(3,1,\mathbf{4},2),(3,1,2,\mathbf{4}),\\
&\quad (\mathbf{4},1,3,2),(1,\mathbf{4},3,2),(1,3,\mathbf{4},2),(1,3,2,\mathbf{4}),\\
&\quad (\mathbf{4},1,2,3),(1,\mathbf{4},2,3),(1,2,\mathbf{4},3),(1,2,3,\mathbf{4}),\\
&\quad (\mathbf{4},3,2,1),(3,\mathbf{4},2,1),(3,2,\mathbf{4},1),(3,2,1,\mathbf{4}),\\
&\quad (\mathbf{4},2,3,1),(2,\mathbf{4},3,1),(2,3,\mathbf{4},1),(2,3,1,\mathbf{4}),\\
&\quad (\mathbf{4},2,1,3),(2,\mathbf{4},1,3),(2,1,\mathbf{4},3),(2,1,3,\mathbf{4})\}
\end{aligned}
$$

Task 2: House of Cards (40 points)

For this task, save your work in `cards.py`

You will experiment with playing cards. The standard deck of cards has 52 playing cards, divided into 4 suits—clubs (♣), diamonds (♦), hearts (♥), and spades (♠). Each suit separately has 13 cards of ranks A (Ace), 2, 3, 4, 5, 6, 7, 8, 9, 10, J (Jack), Q (Queen), K (King).

Throughout this problem, you are not allowed to `import` other modules. Please take into consideration performance expectations described at the end of this problem.

How to Represent Cards? To keep things simple, we'll represent a card using a **tuple** of length 2, indicating first the suit then its rank. The suit is always denoted as a string:

`"Club", "Diamond", "Heart", "Spade"`.

And likewise, the rank is always denoted as a string:

`"A","2", "3", ..., "J", "Q", "K"`.

For example, the card 3♦ will be represented as `("Diamond", "3")`.

Subtask I: Now that you can represent cards, you'll attempt to represent poker hands. Poker is one of the most played card games in the world. In poker, a player constructs hands of playing cards. For the purpose of this problem, a hand has exactly 5 cards. These hands can be compared using a ranking system, and the player who has the highest-ranking hand wins that deal.

Individual cards are ordered by their ranks: A (highest), K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3, 2 (lowest). The Ace card is special. While it is normally the highest-ranked card, in a hand involving 5, 4, 3, 2, A, it takes the place of the lowest card (has rank 1 in this particular case). You can learn more from Wikipedia.

Poker Hand Representation: For this problem, a poker hand is represented as a **set** of 5 cards (each card is represented as detailed previously). Hence, the type Hand is given by

```
Hand = Set[Tuple[str, str]]
```

You are to implement the following functions:

- The function `is_straight_flush(h: Hand) -> bool` returns a Boolean indicating whether the hand h is a straight flush. A straight flush is a hand that contains five cards in sequence, all of the same suit. *Caution:* The Ace can either be the highest or the lowest in the sequence. For example:

  5♠, 4♠, 3♠, 2♠, A♠ A♥, K♥, Q♥, J♥, 10♥ 10♥, 9♥, 8♥, 7♥, 6♥

- The function `is_four_of_a_kind(h: Hand) -> bool` returns a Boolean indicating whether the hand h is a four of a kind. Four of a kind is a poker hand that contains all four cards of one rank and any other card. For example, 3♣, 3♠, 3♦, 3♥, J♥.
- The function `is_full_house(h: Hand) -> bool` returns a Boolean indicating whether the hand h is a full-house hand. A full house is a hand that contains three matching cards of one rank and two matching cards of another rank. For example, 4♣, 4♠, 4♦, 7♥, 7♠.
- The function `is_two_pair(h: Hand) -> bool` returns a Boolean indicating whether the hand h is a two-pair hand. A two pair contains two cards of the same rank, plus two cards of another rank (that match each other but not the first pair), plus any card not of the ranks of the former two. For example, 9♥, 9♣, 4♠, 4♣, 10♣.

To use the type Hand discussed above, your code should like the following:

```
# near the top of the file but above all the functions
Hand = Set[Tuple[str, str]]

# write all your functions. for example,
def is_four_of_a_kind(h: Hand) -> bool:
    # ....
    # ....
```

Subtask II: Of course, different poker hands have different likelihood of occurring—some are harder than others. For this goal, we're going to write functions to enumerate all possible poker hands and classify them into different kinds of hands. Your program must generate these lists—don't just precreate and return them.

1. Implement a function `all_hands() -> List[Hand]` that takes no arguments and returns a list of all possible 5-card hands. Did we mention the five cards are distinct? (*Hint:* Your list should have size $\binom{52}{5} = 2{,}598{,}960$.) *The use of five nested for-loops is strongly frowned upon.*
2. Implement a function `all_straight_flush() -> List[Hand]` that takes no arguments and returns a list of all possible 5-card hands that are straight flush. (*Hint:* Your list should have size 40.)

3. Implement a function `all_four_of_a_kind() -> List[Hand]` that takes no arguments and returns a list of all possible 5-card hands that are four of a kind. (*Hint:* Your list should have size 624.)
4. Implement a function `all_full_house() -> List[Hand]` that takes no arguments and returns a list of all possible 5-card hands that are full house. (*Hint:* Your list should have size 3,744.)
5. Implement a function `all_two_pair() -> List[Hand]` that takes no arguments and returns a list of all possible 5-card hands that are two pair. (*Hint:* Your list should have size 123,552.)

Performance Expectations: For Subtask I, each function should finish instantaneously (i.e., using < 0.1 seconds). For Task II, running all these functions from start to finish should take < 30 seconds.

Task 3: Statistically Speaking (15 points)

For this task, save your work in `statspeak.py`

As we have seen in lecture, a class provides a means to make code and data live (conceptually) in the same place. For this task, you'll implement a class `DataFrame` that maintains a collection of data items inside `self.items`, which you keep as a list of floats, and provides the following operations:

1. Upon creation (remember that `__init__` function?), the collection is empty.
2. A class method `def add(self, x)` will add either `x` or the members of `x` to the list `self.items`. Here `x` is one of the following: (1) a number, (2) a list, or (3) a tuple. If `x` is a number, it is added to the list directly. If `x` is a list or a tuple, its members are individually added to the list.
3. A class method `mean(self)` will compute and return the mean of the data maintained in the collection.
4. A class method `percentile(self, r)` will compute and return the value at the r-th percentile of the data in the collection, where r is an integer in $[0, 100)$ (i.e., excluding 100 but including 0). For this problem, this is the value at the position $\lfloor(\frac{r}{100})\,\ell\rfloor$ (counting from 0) in the sorted list of values, where ℓ is the size of the collection. As an example, for the data collection $[4, 2, 8, 7, 3, 1, 5]$, the 98-th percentile value is 8 because the collection once sorted is $[1, 2, 3, 4, 5, 7, 8]$ and $\lfloor(\frac{98}{100})\,\ell\rfloor = \lfloor 0.98 \times 7\rfloor = 6$. You may find the built-in function `sorted` useful.
5. A class method `mode(self)` will compute and return the mode of the data. The mode of the data is the item that appears the most often in the dataset (i.e., has the highest frequency). If many items have the same highest frequency, your function can return any one of them.
6. A class method `sd(self)` will compute and return the standard deviation of the data. If the data items are $x_0, x_1, \ldots, x_{N-1}$, use the formula
$$\sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(x_i - \overline{x})^2},$$
where N is the number of data items in your collection and $\overline{x}$ is the mean of the data.

When these methods are called, it is possible that they are interleaved. For example, call `add`, followed by another `add`, then `mean`, then a couple more `add`'s. It is, however, guaranteed that the collection will *not* be empty when your code is asked to report a statistic.

Task 4: Auction House (30 points)

For this task, save your work in `auction.py`

This problem contains several subtasks. Your goal for this problem is to develop some classes to model operations at an electronic auction house and use them to analyze auction data.

At an auction house, several auctions can take place at the same time. For each auction, there are bidders who bid in that auction. Every bid raises the price of the auction to a higher value and the highest bidder—i.e., the bidder who gives the highest bid value—wins that auction.

In this problem, you will capture this concept in two classes: `Bid` and `Auction`.

Subtask I: You will implement two classes `Bid` and `Auction`. The details are as follows:

To keep things simple, each *bid* contains 3 attributes:

- `bid_id`: the identifer for this bid;
- `bidder_id`: the identifer of this bidder; and
- `auction`: the identifer of this auction.

Notice that `bid_id` is a unique identifier (think of a ticket) used to identify a bid whereas `bidder_id` identifies the person who made this made.

Details about your `Bid` class: Your `Bid` class

- can be instantiated via `Bid(bid_id, bidder_id, auction)`, for example,

  ```
  bidtest = Bid(1, '8dac2b', 'ewmzr')
  ```

- provides the following access to data: If `b` is a `Bid` instance, then `b.bid_id`, `b.bidder_id`, and `b.auction` store this bid's `bid_id`, `bidder_id`, and `auction`, respectively.
- has appropriate `__str__` and `__repr__` methods that show this bid's information. We do not specify the format.
- supports comparison using $<, >, \leq, \geq, ==$: two bids are compared exclusively by their `bid_id`. As an example, a bid with `bid_id = 3` is (strictly) smaller than a bid with `bid_id= 11`.

Auction Rules. A bidder (a person as determined by a `bidder_id`) can participate in many auctions. The bids are made in the order of `bid_id`—that is, the bid with `bid_id=0` is the first bid made, then the bid with `bid_id=1` is next, and so on. We'll assume that every auction starts at \$1 and each bid after that raises the price of that particular auction by \$1.5. This means, the first bidder for an auction ends up placing a bid of $1 + 1.5 = \$2.5$. Note that the bidders don't get to call how much they're bidding—the auction value just goes up automatically by \$1.5.

The *winner* of an auction is the person who participated in that auction and placed the highest bid.

We'll model an auction as a class called `Auction`. Each auction is identified by an auction identifier. The class has the following features:

- It can be created via `Auction(auction)`, for example,

  ```
  bidtest = Auction('ewmzr')
  ```

  creates an instance of `Auction` with auction identifer `'ewmzr'`.
- The method `placeBid(bidder_id)` reflects the action of the bidder with `bidder_id` placing a bid on this auction. That is to say, if `a` is an `Auction` instance, the call to `a.placeBid(bidder_id)` places a bid from bidder with `bidder_id`.
- If `a` is an `Auction` instance, then `a.price` is the current price of this auction, and `a.winner` is the current winner of this auction. Before anyone places a bid, `a.winner` is, by convention, `None`.

You have now completed the class creation/modeling part. Our next step is to analyze a dataset.

Subtask II: An auction dataset is stored in a file, kept in the CSV format (CSV stands for comma-separated values; read more about CSV online). The *first line* of such a file contains the names of the fields. Subsequent lines are bids from an auction house. Each bid (a row in the file) contains the fields described earlier, among others. For example, consider the following excerpt:

```
bid_id,bidder_id,auction,merchandise,device,time,country,ip,url
0,8dac2b,ewmzr,jewelry,phone0,9759243157894736,us,69.166.231.58,vasstdc27m7nks3
1,668d39,aeqok,furniture,phone1,9759243157894736,in,50.201.125.84,jmqlhflrzwuay9c
```

The 1st line is the "header" line, which lists all the fields. The 2nd line shows a bid with `bid_id = 0`. The person who made this bid has `bidder_id = '8dac2b'`, and the bid is for the auction `'ewmzr'`.

For this subtask, you will implement a function `CSV2List(csvFilename: str) -> List[Bid]` which

- takes as input a file name; and
- returns a list of `Bid` instances ordered by their `bid_id`, from small to large.

Keep in mind that the input file may not list the bids in the right `bid_id` order. You will need to reorder them.

Hint #1: How does `sorted` or `list.sort` interact with your custom-made comparison above?

Hint #2: `import csv`. It's not fast but it understands CSV.

Subtask III: In the final subtask, you will implement the following two functions to analyze the bids:

- a function `mostPopularAuction(bidList: List[Bid]) -> Set[Auction]` takes in a list of `Bid` instances (for example, as what you would get from `CSV2List` above) and returns a **set** of identifiers (each a string) of the most popular auction(s). The most popular auction is defined as the auction that has the most distinct number of bidders. There may be multiple auctions with the same number of bidders.
- a function `auctionWinners(bidList: List[Bid]) -> Dict[str, Auction]` takes in a list of `Bid` instances (same as above) and returns a dictionary with the following property:

  > If d is the resulting dictionary and a is an auction identifier, then `d[a]` is an `Auction` instance that reflects the state of this auction (the auction with identifier a) after going through all the bids.

Other Things:

- We're testing your program with datasets that contain up to 1 million rows (and 20MB in size).
- On such datasets, we expect each of your functions to run within 15 seconds.
- There will be sample input/output files on the course website.

Task 5: EXTRA: Text-based Newspaper Columns (0 points)

For this task, save your work in `textcol.py`

READ THIS BEFORE YOU ATTEMPT IT: This is an extra problem for those who want a more challenging task. It is worth 0 points on the assignment; however, you'll earn

- bragging rights;
- a place on the course's wall of fame;
- brownie points that may be exchanged for real points if needed at the end; and
- above all, an opportunity to practice programming.

This problem is inspired by and adapted from a reddit challenge (credit to Challenge #225). In the old days, graphical interface wasn't even available and text interface was the norm. But programmers were able to accomplish many higher art forms using just text. We're going to explore newspaper columns rendered completely in text. Now column-style writing often puts images and features to the left or right of the body of text, for example (quotation marks only shown to indicate where the lines begin and end):

```
"This is an example piece of text. This is an exam-"
"ple piece of text. This is an example piece of"
"text. This is an example"
"piece of text. This is a +-----------------------+"
"sample for a challenge.  |                       |"
"Lorum ipsum dolor sit a- |       top class       |"
"met and other words. The |        feature        |"
"proper word for a layout |                       |"
"like this would be type- +-----------------------+"
"setting, or so I would"
"imagine, but for now let's carry on calling it an"
"example piece of text. Hold up - the end of the"
"                 paragraph is approaching - notice"
"+--------------+ the double line break for a para-"
"|              | graph."
"|              |"
"|   feature    | And so begins the start of the"
"|   bonanza    | second paragraph but as you can"
"|              | see it's only marginally better"
"|              | than the other one so you've not"
"+--------------+ really gained much - sorry. I am"
"                 certainly not a budding author"
"as you can see from this example input. Perhaps I"
"need to work on my writing skills."
```

Some words overfill the column and end up getting hyphenated. For this task, you will assume that any hyphens at the end of a line join a single unhyphenated word together (for example, the `exam-` and `ple` in the above input form the word `example` and not `exam-ple`). However, hyphenated words that do not span multiple lines should retain their hyphens.

Additionally, side features will only appear at the far left or right of the input, and will always be bordered by the `+---+` style shown above. They will also never have "holes" in them, so the situation on the right will never happen.

```
+--------------------+
|                    |
| Inside the feature |
|                    |
| +----------------+ |
| |                | |
| |     Outside    | |
| |                | |
| +----------------+ |
|                    |
+--------------------+
```

Paragraphs in the input are separated by double line breaks.

Your Task: You'll implement a class `TextColumn` that has the following properties:

- An instance of `TextColumn` can be created with `TextColumn(lines)`, where `lines` a list of strings, each string representing a line. Internal bookkeeping is up to you as long as you provide the functionality stated.
- If `c` is an instance of `TextColumn`, then `c.paragraphs()` will return a list of strings, each representing a paragraph in the given string. The text of a paragraph will flow smoothly—words that were split across lines through hyphenation will be joined back. Importantly, the paragraphs have to be listed in the order that they appear (from top to bottom).

  For instance, the text in the above example would result in the following list (we artificially added ↪ to indicate line breaks due to typesetting limitations):

```
["This␣is␣an␣example␣piece␣of␣text.␣This␣is␣an␣example␣piece␣of␣text.␣This␣is␣an␣
    ↪example␣piece␣of␣text.␣This␣is␣an␣example␣piece␣of␣text.␣This␣is␣a␣sample␣
    ↪for␣a␣challenge.␣Lorum␣ipsum␣dolor␣sit␣amet␣and␣other␣words.␣The␣proper␣
    ↪word␣for␣a␣layout␣like␣this␣would␣be␣typesetting,␣or␣so␣I␣would␣imagine,␣
    ↪but␣for␣now␣let's␣carry␣on␣calling␣it␣an␣example␣piece␣of␣text.␣Hold␣up␣-␣
    ↪the␣end␣of␣the␣paragraph␣is␣approaching␣-␣notice␣the␣double␣line␣break␣for␣
    ↪a␣paragraph.",
"And␣so␣begins␣the␣start␣of␣the␣second␣paragraph␣but␣as␣you␣can␣see␣it's␣only␣
    ↪marginally␣better␣than␣the␣other␣one␣so␣you've␣not␣really␣gained␣much␣-␣
    ↪sorry.␣I␣am␣certainly␣not␣a␣budding␣author␣as␣you␣can␣see␣from␣this␣example␣
    ↪input.␣Perhaps␣I␣need␣to␣work␣on␣my␣writing␣skills."]
```

- If `c` is an instance of `TextColumn`, then `c.features()` will return a list of strings, each representing a feature text in the given input. Features are side features described above. The features can follow any order of your liking as long as the list contains all the text inside the features (with trailing white spaces removed). For the above example, the list will be

```
["top class feature", "feature bonanza"]
```