Midterm Exam - DSC 10, Winter 2024

Full Name:

PID:

Lecture: ◯ A (9AM) ◯ B (10AM) ◯ C (11AM)

Instructions:

- This exam consists of 8 questions, worth a total of 83 points.
- Write your PID in the top right corner of each page in the space provided.
- Please write **clearly** in the provided answer boxes; we will not grade work that appears elsewhere. Completely fill in bubbles and square boxes; if we cannot tell which option(s) you selected, you may lose points.

  ◯ A bubble means that you should only **select one choice**.

  ☐ A square box means you should **select all that apply**.
- Aside from the provided Reference Sheet, you may not refer to any other resources or technology during the exam (no notes, no calculators).

By signing below, you are agreeing that you will behave honestly and fairly during and after this exam.

Signature:

Version A

Please do not open your exam until instructed to do so.

Important: Before proceeding, make sure to rip off the last two pages from this exam packet and read the page called *Clue*: The Murder Mystery Game.

Question 1 (18 pts)

Each of the following expressions evaluates to an integer. Determine the value of that integer, if possible, or circle "not enough information."

a) `(clue.get("Cardholder") == "Janine").sum()`

not enough information

b) `np.count_nonzero(clue.get("Category").str.contains("p"))`

not enough information

c)
```
clue[(clue.get("Category") == "suspect") &
      (clue.get("Cardholder") == "Janine")].shape[0]
```

not enough information

d) `len(clue.take(np.arange(5, 20, 3)).index)`

not enough information

e) `len(clue[clue.get("Category") >= "this"].index)`

not enough information

f) `clue.groupby("Cardholder").count().get("Category").sum()`

not enough information

PID: _____

Question 2 (8 pts)

Since Janine's knowledge of who holds each card will change throughout the game, the `clue` DataFrame needs to be updated by setting particular entries.

Suppose more generally that we want to write a function that changes the value of an entry in a DataFrame. The function should work for any DataFrame, not just `clue`.

What parameters would such a function require? Say what each parameter represents.

Question 3 (8 pts)

An important part of the game is knowing when you've narrowed it down to just one suspect with one weapon in one room. Then you can make your accusation and win the game!

Suppose the DataFrames `grouped` and `filtered` are defined as follows.

```
grouped = (clue.reset_index()
               .groupby(["Category", "Cardholder"])
               .count()
               .reset_index())
filtered = grouped[grouped.get("Cardholder") == "Unknown"]
```

a) (4 pts) Fill in the blank below so that `"Ready to accuse"` is printed when Janine has enough information to make an accusation and win the game.

```
if filtered.get("Card").______ == 3:
    print("Ready to accuse")
```

What goes in the blank?

◯ `count()` ◯ `sum()` ◯ `max()` ◯ `min()` ◯ `shape[0]`

b) (4 pts) Now, let's look at a different way to do the same thing. Fill in the blank below so that `"Ready to accuse"` is printed when Janine has enough information to make an accusation and win the game.

```
if filtered.get("Card").______ == 1:
    print("Ready to accuse")
```

What goes in the blank?

◯ `count()` ◯ `sum()` ◯ `max()` ◯ `min()` ◯ `shape[0]`

PID: _____

Question 4 (7 pts)

When someone is ready to make an accusation, they make a statement such as:

"It was Miss Scarlett with the dagger in the study"

While the suspect, weapon, and room may be different, an accusation will always have this form:

"It was _____ with the _____ in the _____"

Suppose the array `words` is defined as follows (note the spaces).

```
words = np.array(["It was ", " with the ", " in the "])
```

Suppose another array called `answers` has been defined. `answers` contains three elements: the name of the suspect, weapon, and room that we would like to use in our accusation, in that order. Using `words` and `answers`, complete the `for`-loop below so that `accusation` is a string, formatted as above, that represents our accusation.

```
accusation = ""
for i in ___(a)___:
    accusation = ___(b)___
```

a) (3 pts) What goes in blank (a)?

b) (4 pts) What goes in blank (b)?

Question 5 (12 pts)

Recall that the game *Clue* comes with 22 cards, one for each of the 6 suspects, 7 weapons, and 9 rooms. One suspect card, one weapon card, and one room card are chosen randomly, without being looked at, and placed aside in an envelope. The remaining 19 cards (5 suspects, 6 weapons, 8 rooms) are randomly shuffled and dealt out, splitting them as evenly as possible among the players. Suppose in a three-player game, Janine gets 6 cards, which are dealt one at a time.

Answer the probability questions that follow. Leave your answers **unsimplified**.

a) (4 pts) Cards are dealt one at a time. What is the probability that the first card Janine is dealt is a weapon card?

b) (4 pts) What is the probability that all 6 of Janine's cards are weapon cards?

c) (4 pts) Determine the probability that exactly one of the first two cards Janine is dealt is a weapon card. This probability can be expressed in the form

$$\frac{k \cdot (k+1)}{m \cdot (m+1)}$$

where k and m are **integers**. What are the values of k and m?

Hint: There is no need for any sort of calculation that you can't do easily in your head, such as long division or multiplication.

$k =$ $\qquad$ $m =$

PID: _____

Question 6 (4 pts)

Which of the following probabilities could most easily be approximated by writing a simulation in Python? Select the best answer.

◯ The probability that Janine wins the game.

◯ The probability that a three-player game takes less than 30 minutes to play.

◯ The probability that Janine has three or more suspect cards.

◯ The probability that Janine visits the kitchen at some point in the game.

Question 7 (18 pts)

Part of the gameplay of *Clue* involves moving around the gameboard. The gameboard has 9 rooms, arranged on a grid, and players roll dice to determine how many spaces they can move.

The DataFrame `dist` contains a row and a column for each of the 9 rooms. The entry in row r and column c represents the shortest distance between rooms r and c on the *Clue* gameboard, or the smallest dice roll that would be required to move between rooms r and c. Since you don't need to move at all to get from a room to the same room, the entries on the diagonal are all 0.

`dist` is indexed by `"Room"`, and the room names appear exactly as they appear in the index of the `clue` DataFrame. These same values are also the column labels in `dist`.

a) (4 pts) Two of the following expressions are equivalent, meaning they evaluate to the same value without erroring. Select these **two expressions**.

- ☐ `dist.get("kitchen").loc["library"]`
- ☐ `dist.get("kitchen").iloc["library"]`
- ☐ `dist.get("library").loc["kitchen"]`
- ☐ `dist.get("library").iloc["kitchen"]`

Explain in **one sentence** why these two expressions are the same.

b) (4 pts) On the *Clue* gameboard, there are two "secret passages." Each secret passage connects two rooms. Players can immediately move through secret passages without rolling, so in `dist` we record the distance as 0 between two rooms that are connected with a secret passage.

Suppose we run the following code.

```
nonzero = 0
for col in dist.columns:
    nonzero = nonzero + np.count_nonzero(dist.get(col))
```

Determine the value of `nonzero` after the above code is run.

`nonzero =` ☐

c) (6 pts) Fill in blanks so that the expression below evaluates to a DataFrame with all the same information as `dist`, plus **one extra column** called `"Cardholder"` containing Janine's knowledge of who holds each room card.

```
dist.merge(___(a)___, ___(b)___, ___(c)___)
```

(i) What goes in blank (a)?

(ii) What goes in blank (b)?

(iii) What goes in blank (c)?

d) (4 pts) Suppose we generate a scatter plot as follows.

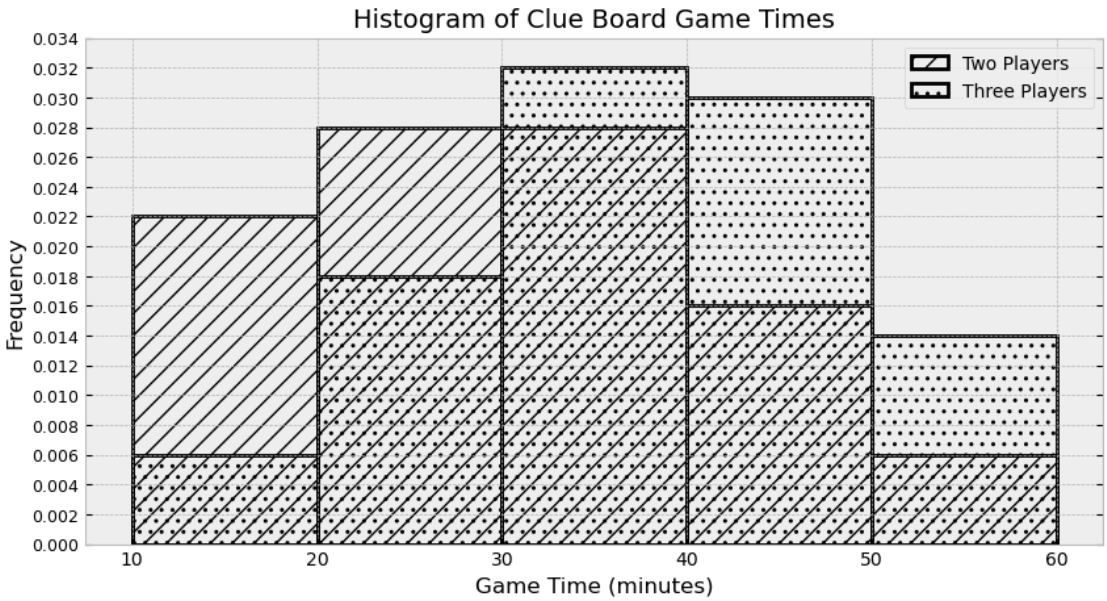
```
dist.plot(kind="scatter", x="kitchen", y="study");
```

Suppose the scatterplot has a point at (4, 6). What can we conclude about the *Clue* gameboard?

◯ The kitchen is 4 spaces away from the study.
◯ The kitchen is 6 spaces away from the study.
◯ Another room besides the kitchen is 4 spaces away from the study.
◯ Another room besides the kitchen is 6 spaces away from the study.

Question 8 (8 pts)

The histogram below shows the distribution of game times in minutes for both two-player and three-player games of *Clue*, with each distribution representing 1000 games played.



a) (4 pts) How many **more** three-player games than two-player games took at least 50 minutes to play? Give your answer as an **integer, rounded to the nearest multiple of 10**.

b) (4 pts) Calculate the approximate area of overlap of the two histograms. Give your answer as a **proportion between 0 and 1, rounded to two decimal places**.

🔍 *Clue*: The Murder Mystery Game 🔍

Clue is a murder mystery game where players use the process of elimination to figure out the details of a crime. The premise is that a murder was committed inside a large home, by one of **6 suspects**, with one of **7 weapons**, and in one of **9 rooms**.

The game comes with **22 cards**, one for each of the 6 suspects, 7 weapons, and 9 rooms. To set up the game, one suspect card, one weapon card, and one room card are chosen randomly, without being looked at, and placed aside in an envelope. The cards in the envelope represent the details of the murder: who did it, with what weapon, and in what room.

The remaining 19 cards are randomly shuffled and dealt out to the players (as equally as possible). Players then look at the cards they were dealt and can conclude that any cards they see were **not** involved in the murder. In the gameplay, players take turns moving around to different rooms of the house on the gameboard, which gives them opportunities to see cards in other players' hands and further eliminate suspects, weapons, and rooms. The first player to narrow it down to one suspect, with one weapon, and in one room can make an accusation and win the game!

Suppose Janine, Henry, and Paige are playing a game of Clue. Janine and Paige are each dealt 6 cards, and Henry is dealt 7. The DataFrame `clue` has 22 rows, one for each card in the game. `clue` represents **Janine's knowledge** of who is holding each card. `clue` is indexed by `"Card"`, which contains the name of each suspect, weapon, and room in the game. The `"Category"` column contains `"suspect"`, `"weapon"`, or `"room"`. The `"Cardholder"` column contains `"Janine"`, `"Henry"`, `"Paige"`, or `"Unknown"`.

Since Janine's knowledge is changing throughout the game, the `"Cardholder"` column needs to be updated frequently. At the beginning of the game, the `"Cardholder"` column contains only `"Janine"` and `"Unknown"` values. We'll assume throughout this exam that `clue` contains Janine's current knowledge at an arbitrary point in time, not necessarily at the beginning of the game. For example, `clue` **may look like** the DataFrame at right.

Card	Category	Cardholder
Col. Mustard	suspect	Unknown
Dr. Orchid	suspect	Henry
Miss Scarlett	suspect	Henry
Mr. Green	suspect	Paige
Mrs. Peacock	suspect	Unknown
...	...	...
hall	room	Janine
kitchen	room	Janine
library	room	Unknown
lounge	room	Janine
study	room	Unknown

22 rows × 2 columns

Note: Throughout the exam, assume we have already run `import babypandas as bpd` and `import numpy as np`.

DSC 10 Reference Sheet

Below, `df` is a DataFrame, `ser` is a Series, `arr` is an array, `babypandas` has been imported as `bpd`, and `numpy` has been imported as `np`.

Building and Organizing DataFrames

Each function/method below creates a new DataFrame. Remember to save it!

`bpd.DataFrame()`
: Creates an empty DataFrame.

`bpd.read_csv(path_to_file)`
: Creates a DataFrame by reading from a CSV file.

`df.assign(name_of_column=column_data)`
: Adds/replaces a column. `name_of_column` should not have quotes or spaces.

`df.drop(columns=column_name or [col_1_name, ..., col_k_name])`
: Drops a single column, or every column in a list of column names.

`df.set_index(column_name)`
: Moves a column to the index.

`df.reset_index()`
: Moves the index to a column.

`df.sort_values(by=column_name, ascending=True)`
: Sorts the entire DataFrame in ascending order by the values in a column. `ascending` can be omitted, as its default value is `True`.

`left.merge(right, left_on=left_column, right_on=right_column)`
: Merges the DataFrames `left` and `right` by the specified columns. Can use `on` instead of `left_on` and `right_on` if the column names are the same.

`left.merge(right, left_index=True, right_on=right_column)`
: Merges using `left`'s index instead of a column. Can also be done with `right_index=True`.

Arrays and NumPy

`arr[index]`
: The element at position `index` in the array `arr`. The first element is `arr[0]`.

`np.append(arr, value)`
: A copy of `arr` with `value` appended to the end. This does not change `arr` unless you store the result in `arr`.

`np.count_nonzero(arr)`
: The number of non-zero entries in `arr`. `True` counts as 1, `False` counts as 0.

`np.arange(start, stop, step)`
: An array of numbers starting with `start`, increasing/decreasing in increments of `step`, and stopping before (excluding) `stop`. If `start` or `step` are omitted, the default values are 0 and 1, respectively.

`np.percentile(arr, p)`
: The pth percentile of the numbers in `arr`.

These also work if `arr` is a Series, list, or some other type of sequence.

Plotting

`df.plot(kind=kind, x=col_x, y=col_y)`
: Draws a plot. `kind` may be `'scatter'`, `'line'`, `'bar'`, or `'barh'`. If `x` is omitted, the index is used. For most kinds of plots, if `y` is omitted, all columns are plotted on shared axes.

`df.plot(kind='hist', y=data_col, bins=the_bins, density=True)`
: Plots a density histogram of the numerical data in `data_col`. `the_bins` can be a number of bins, or a sequence specifying bin endpoints. Scaled so that the total area is 1.

Accessing Data

`df.shape[0]` and `df.shape[1]`
: The number of rows and the number of columns, respectively.

`df.get(column_name)`
: One column, as a Series.

`df.get([col_1_name, ..., col_k_name])`
: Several columns, as a DataFrame.

`ser.loc[label]`
: An element of `ser`, accessed by its row label. Often `ser` comes from `df.get(column_name)`.

`ser.iloc[position]`
: An element of `ser`, accessed by its integer position. Positions start at 0. Often `ser` comes from `df.get(column_name)`.

`df.index[position]`
: An element in the index, accessed by its integer position. Positions start at 0.

`df.take([position_1, ..., position_k])`
: A DataFrame of specific rows, accessed by integer position. Often used with `np.arange`.

`df[bool_arr]`
: A DataFrame of specific rows, usually where some condition is satisfied. See: Querying.

Series Methods

Series have the following methods:

`.count()`, `.max()`, `.min()`, `.sum()`, `.mean()`, `.median()`, `.unique()`

Querying

Querying (also called filtering or Boolean indexing) selects a subset of a DataFrame's rows. We need a sequence of Boolean values, `condition`, with length equal to the number of rows of the DataFrame. The expression `df[condition]` results in a DataFrame containing only those rows whose corresponding element in `condition` is `True`.

Boolean sequences are easily constructed by comparing an array, index, or Series to a value using the comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`.

`(bool_arr_1) & (bool_arr_2)`
: The "and" of two Boolean arrays. Contains `True` where both Boolean arrays contain `True`, otherwise contains `False`.

`(bool_arr_1) | (bool_arr_2)`
: The "or" of two Boolean arrays. Contains `True` where at least one of the Boolean arrays contains `True`, otherwise contains `False`.

`df[df.get(column_name) > 42]`
: Retrieves all rows for which the given column is bigger than 42.

`df[(df.get(column_name) > 42) & (df.get(column_name) < 100)]`
: Retrieves all rows for which the given column is between 42 and 100. Parentheses are important!

`df[df.get(column_name).str.contains(pattern)]`
: Retrieves all rows for which the given column contains the string `pattern`.

`df[df.index > 2]`
: Retrieves all rows for which the index is greater than 2.

Grouping

Use `df.groupby(column_name)`, followed by one of these aggregation methods:

`.mean()`, `.median()`, `.count()`, `.max()`, `.min()`, `.sum()`

This results in a DataFrame indexed by the group names. Only those columns whose data type permits the selected aggregation method are kept – for instance, `.sum()` will drop columns containing strings.

`df.groupby([col_1_name, ..., col_k_name])` creates subgroups, first grouping by `col_1_name`, then, within each group, grouping by `col_2_name`, and so on. The resulting DataFrame has one row for every combination of values in the given columns. Typically, grouping with subgroups is followed by `.reset_index()`.

Writing Functions

```
def function_name(argument_1, ..., argument_k):
  <function body>
```

For example, this function squares a number:

```
def square_a_number(number):
  return number**2
```

Applying Functions

`df.get(column_name).apply(function_name)`
: Applies a function of one parameter to every entry in the column. Returns a Series of the same size containing the results.

`if`-statements

```
if <condition>:
  <if body>
elif <second_condition>:
  <elif body>
elif <third_condition>:
  <elif body>
...
else:
  <else body>
```

Conditionally execute code. The `elif` and `else` blocks are optional.

Statistics and Hypothesis Testing

A **sample** is a subset of a **population**. A **statistic** is a number computed using the sample. The field of statistics is about using a sample to say something about the population, which is called **inference**.

An **experiment** is a process whose outcome is random; for example, flipping 100 coins. An **observed statistic** is a statistic computed from the outcome of an experiment; for example, the number of heads observed. A **model** is a set of assumptions about how the data was generated. For example: the result of a coin flip is equally likely to be heads or tails. **Hypothesis testing** is the process of testing the validity of a model based on simulating data with the model and comparing it to observed data.

To perform a **hypothesis test**, we first establish a **null hypothesis**: this is a precise assumption about how the data was generated. For instance: the coin is fair. Then we state an **alternative hypothesis**, such as: the coin is not fair.

To **test** the hypothesis, we compute the probability of seeing an outcome at least as extreme as the observed statistic under the assumptions of the null hypothesis; this is called the **p-value**. In practice, we do this by simulating many of outcomes using the null hypothesis and counting how many times the outcome is equal to or more extreme than what was originally observed.

`for`-loops

```
for <loop variable> in <sequence>:
  <loop body>
```

Performs the loop body for every element of the sequence. For example, to print the squares of the numbers 0 through 9:

```
for i in np.arange(10):
  print(i**2)
```

Random Sampling

`np.random.choice(arr, size, replace=True, p=[p_0, p_1, ...])`
: An array of elements chosen from `arr` at random, with replacement, such that `array[i]` is selected with probability `p_i`. `replace` can be omitted, as its default value is `True`. If `size` is omitted, the result is a single randomly chosen element instead of an array of length `size`.

`np.random.multinomial(n, [p_0, p_1, p_2, ...])`
: An array where each element contains the number of occurrences of an event, where events have probabilities `p_0`, `p_1`, `p_2`, ... For instance, if each M&M is red with probability 0.2, green with probability 0.5, and brown with probability 0.3, then a random selection of 100 M&Ms is given by:

`np.random.multinomial(100, [0.2, 0.5, 0.3]).`

The result might be `[22, 45, 33]`, representing the number of red, green, and brown M&Ms, respectively.

`np.random.permutation(arr)`
: A random shuffling/reordering of the input.

`df.sample(n, replace=False)`
: A random sample of n rows from `df`, selected without replacement. `replace` can be omitted, as its default value is `False`.

Bootstrapping and Confidence Intervals

When we compute a statistic from a sample, such as the median salary of San Diego city employees, since our sample is random, our statistic could have been different if we'd had a different sample. Bootstrapping allows us to answer: "how different could it have been?" by giving us an approximation of the distribution of the sample statistic.
Suppose `salaries` contains a column called "Salary" containing the salary of each employee in a sample. The observed median salary is:

```
observed = salaries.get('Salary').median()
```

To make a 95% confidence interval for the median salary, we bootstrap by repeatedly resampling the data in our sample, with replacement:

```
boot_medians = np.array([])
for i in np.arange(10000):
  # 1. Resample the data.
  resample = salaries.sample(salaries.shape[0], replace=True)

  # 2. Compute the statistic on the bootstrap resample.
  boot_median = resample.get('Salary').median()

  # 3. Save the result.
  boot_medians = np.append(boot_medians, boot_median)
```

The endpoints of a 95% bootstrapped confidence interval are:

```
left = np.percentile(boot_medians, 2.5)
right = np.percentile(boot_medians, 97.5)
```

This interval gives a range of reasonable values where we'd guess the population median would fall.