Midterm Exam - DSC 10, Spring 2024

Full Name:

PID:

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

Instructions:

- This exam consists of 10 questions, worth a total of 57 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 sheets of paper from this exam packet and read the data description.

Question 1 (8 pts)

Recall from the data description that the `"DOB"` column in `contacts` contains the date of birth of everyone in your contacts list, as a string formatted like `"MM-DD-YYYY"`.

Looking at the calendar, you see that today's date is May 3rd, 2024, which is `"05-03-2024"`.

a) (5 pts) Using today's date, fill in the blanks in the function `age_today` so that the function takes as input someone's date of birth, as a string formatted like `"MM-DD-YYYY"`, and returns that person's age, as of today.

```
def age_today(dob):
    dob = dob.split("-")
    month = ___(a)___ # the month, as an int
    day = ___(b)___ # the day, as an int
    year = ___(c)___ # the year, as an int
    if ___(d)___:
        return 2024 - year
    return 2024 - year - 1
```

(a):

(b):

(c):

(d):

b) (3 pts) Write a Python expression that evaluates to the average age of all of your contacts, as of today.

PID: _____

Question 2 (5 pts)

You wonder if any of your friends have the same birthday, for example two people both born on April 3rd. Fill in the blanks below so that the given expression evaluates to the largest number of people in `contacts` who share the same birthday.

Note: People do not need to be born in the same year to share a birthday!

```
contacts.groupby(___(a)___).___(b)___.get("Phone").___(c)___
```

(a):

(b):

(c):

Question 3 (4 pts)

Consider the histogram generated by the following code.

```
contacts.plot(kind="hist", y="Day", density=True, bins=np.arange(1, 33))
```

Which of the following questions would you be able to answer from this histogram? Select all that apply.

- ☐ How many of your contacts were born on the first day of the month?
- ☐ How many of your contacts were born on the last day of the month?
- ☐ How many of your contacts were born on a Monday?
- ☐ How many of your contacts were born on January 1st?

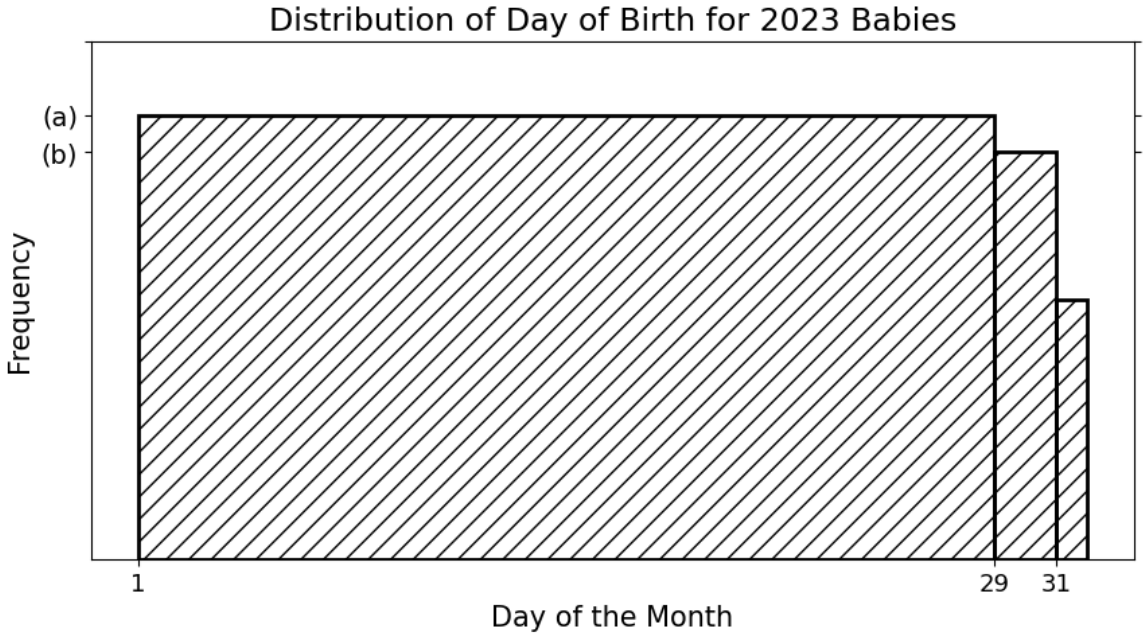
Question 4 (6 pts)

After looking at the distribution of `"Day"` for your contacts, you wonder how that distribution might look for other groups of people.

In this problem, we will consider the distribution of day of birth (just the day of the month, not the month or year) for all babies born in the year 2023, under the assumption that all babies born in 2023 were equally likely to be born on each of the 365 days of the year.

The density histogram below shows this distribution.

Note: 2023 was not a leap year, so February had 28 days.



a) (3 pts) What is height `(a)`?

◯ $\frac{1}{12}$ ◯ $\frac{12}{365}$ ◯ 1 ◯ $\frac{1}{28}$ ◯ $\frac{28}{365}$ ◯ $\frac{12 \cdot 28}{365}$

b) (3 pts) Express height `(b)` in terms of height `(a)`.

◯ `(b)` = $\frac{7}{12}$·`(a)` ◯ `(b)` = $\frac{27}{28}$·`(a)` ◯ `(b)` = $\frac{11}{12}$·`(a)`

◯ `(b)` = $\frac{28}{29}$·`(a)` ◯ `(b)` = $\frac{7}{11}$·`(a)` ◯ `(b)` = $\frac{12 \cdot 28}{365}$·`(a)`

PID: _____

Question 5 (5 pts)

Fill in the blanks in the function `sum_phone` below. This function should take as input a string representing a phone number, given in the form `"xxx-xxx-xxxx"`, and return the sum of the digits of that phone number, as an `int`.

For example, `sum_phone("501-800-3002")` should evaluate to `19`.

```
def sum_phone(phone_number):
    total = 0
    for digit in phone_number:
        if ___(a)___:
            ___(b)___
    return total
```

(a): [ ]

(b): [ ]

Question 6 (6 pts)

The first contact in `contacts` is your friend Calvin, who has an interesting phone number, with all the digits in descending order: 987-654-3210. Fill in the blanks below so that each expression evaluates to the sum of the digits in Calvin's phone number.

a) (2 pts)

```
contacts.get("Phone").apply(sum_phone).loc[___(a)____]
```

(a): [ ]

b) (2 pts)

```
sum_phone(contacts.get("Phone").iloc[___(b)____])
```

(b): [ ]

c) (2 pts)

```
np.arange(__(c)__,__(d)__,__(e)__).sum()
```

(c): [ ], (d): [ ], (e): [ ]

Question 7 (3 pts)

Write a Python expression that evaluates to a DataFrame of only your contacts whose phone numbers end in `"1234"`.

Note: Do not use slicing, if you know what that is. You must use methods from this course.

Question 8 (6 pts)

a) (3 pts) Oh no! A monkey has grabbed your phone and is dialing a phone number by randomly pressing buttons on the keypad, such that each button pressed is equally likely to be any of ten digits 0 through 9.

The monkey managed to dial a ten-digit number and call that number. What is the probability that the monkey calls one of your contacts? Give your answer as a Python expression, using the DataFrame `contacts`.

b) (3 pts) Now, your cat is stepping carefully across the keypad of your phone, pressing 10 buttons. Each button is sampled randomly **without replacement** from the digits 0 through 9.

You catch your cat in the act of dialing, when the cat has already dialed 987-654. Based on this information, what is the probability that the cat dials your friend Calvin's number, 987-654-3210? Give your answer as an unsimplified mathematical expression.

PID: _____

Question 9 (8 pts)

Arya's phone number has an interesting property: after the area code (the first three digits), the remaining seven numbers of his phone number consist of only two distinct digits.

Recall from the previous question that when the monkey dials a phone number, each digit it selects is equally likely to be any of the digits 0 through 9. Further, when the cat is dialing a phone number, it makes sure to only use each digit once.

You're interested in estimating the probability that a phone number dialed by the monkey or the cat has exactly two distinct digits after the area code, like Arya's phone number. You write the following code, which you plan to use for both the monkey and cat scenarios.

```
digits = np.arange(10)
property_count = 0
num_trials = 10000
for i in np.arange(num_trials):
    after_area_code = __(x)__
    num_distinct = len(np.unique(after_area_code))
    if __(y)__:
        property_count = property_count + 1
probability_estimate = property_count / num_trials
```

a) (3 pts) First, you want to estimate the probability that the **monkey** randomly generates a number with only 2 distinct digits after the area code. What code should be used to fill in blank `(x)`?

`(x)`: _____

b) (1 pt) Next, you want to estimate the probability that the **cat** randomly generates a number with only 2 distinct digits after the area code. What code should be used to fill in blank `(x)`?

`(x)`: _____

c) (2 pts) In either case, whether you're simulating the monkey or the cat, what should be used to fill in blank `(y)`?

`(y)`: _____

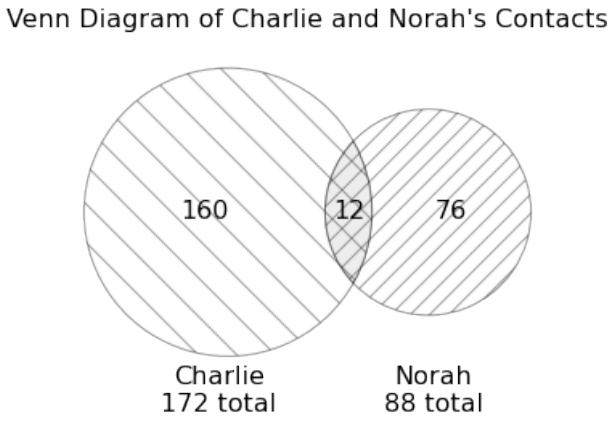
d) (2 pts) When you are simulating the **cat**, what will the value of `probability_estimate` be after the code executes?

`probability_estimate`: _____

Question 10 (6 pts)

Suppose Charlie and Norah each have separate DataFrames for their contacts, called `charlie` and `norah`, respectively. These DataFrames have the same column names and format as your DataFrame, `contacts`.

As illustrated in the diagram below, Charlie has 172 contacts in total, whereas Norah has 88 contacts. 12 of these contacts are shared, meaning they appear in both `charlie` and `norah`.



a) (3 pts) What does the following expression evaluate to?

```
charlie.merge(norah, left_index=True, right_index=True).shape[0]
```

b) (3 pts) One day, when updating her phone's operating system, Norah accidentally duplicates the 12 contacts she has in common with Charlie. Now, the `norah` DataFrame has 100 rows.

What does the following expression evaluate to?

```
norah.merge(norah, left_index=True, right_index=True).shape[0]
```

Contacts

In this exam, you'll work with a data set representing your phone's contact list. `contacts` is indexed by `"Person"`. You store each person in your contacts under a unique name.

The columns of `contacts` are as follows:

- `"Phone"` (`str`): Your contact's phone number, formatted like `"xxx-xxx-xxxx"`, where each `x` represents a digit 0 through 9. All of your contacts have distinct phone numbers.
- `"Day"` (`int`): The day of the month your contact was born. Values are 1 through 31.
- `"Month"` (`str`): The month your contact was born.
- `"Year"` (`int`): The year your contact was born.
- `"DOB"` (`str`): Your contact's date of birth, in the form `"MM-DD-YYYY"`.

The first few rows of `contacts` are shown below, though `contacts` has many more rows than pictured.

Person	Phone	Day	Month	Year	DOB
Calvin	987-654-3210	30	October	1987	10-30-1987
Minchan	510-219-5128	29	July	1999	07-29-1999
Kate	725-489-1945	2	August	2004	08-02-2004
Pallavi	827-593-3209	14	February	1950	02-14-1950

Throughout this exam, we will refer to `contacts` repeatedly.

Assume that 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.