CSE 163: Intermediate Data Programming

Practice Exam 1 – Summer 2026

Full Name: _____ **Student ID:** _____

Quiz Section: _____ **UW NetID:** _____

Instructions & Policies

- You have **60 minutes** to complete this exam.
- This exam is **closed book** and **closed notes**, except for one double-sided handwritten note sheet and the provided syntax reference sheet.
- **No electronic devices** (phones, laptops, calculators, smart watches) are permitted.
- Write your answers in the space provided. There is sufficient room provided to answer all question parts completely.
- Read every question carefully. Partial credit may be awarded for work that is shown.
- You may assume that all necessary libraries have already been imported for any programming questions.
- There are **5** questions on this exam, worth a total of **100** points.
- If you have a question, raise your hand and wait for a TA to come to you. Any clarifications that need to be made will be posted on the screen.
- Do not open or turn the page until instructed to do so.

Academic Honesty Statement

By signing below, I affirm that the work on this exam is entirely my own and that I have neither given nor received any unauthorized assistance. I have read the instructions above and agree to abide by the course and university academic honesty policies. I understand that if I am found in violation of these policies, I will receive a score of 0 on the final exam.

Signature: _____ **Date:** _____

Do not write in the box below.

Question:	1	2	3	4	5	Total
Points:	20	20	20	20	20	100
Score:						

Question 1 (20 points)

Complete the function `count_words` below, filling in the input and output type annotations, a clear docstring, and the function body. It takes as input the path to a CSV file and returns a dictionary that counts the number of times each word appears in the file. Each line of the file contains one or more comma-separated words. You do not need to worry about other punctuation marks or spaces in this problem. You may not use advanced libraries or functions not introduced in this course, such as `collections.Counter`, for this task.

For example, suppose the file `words.csv` contains:

```
Buffalo,buffalo,Buffalo,buffalo
buffalo,buffalo,Buffalo,buffalo
```

Then calling your function on it should give:

```
print(count_words("words.csv"))
# {'Buffalo': 3, 'buffalo': 5}
```

Complete the following function:

```
def count_words(file_path: __________) -> __________:
    # your code here
```

Possible solution:

```
def count_words(file_path: str) -> dict[str, int]:
    """
    Returns a dictionary mapping each unique word in the given
    CSV file to the number of times it appears. Every comma-
    separated value in the file is treated as a single word.
    """
    word_count = {}
    with open(file_path) as f:
        for line in f:
            for word in line.strip().split(','):
                if word not in word_count:
                    word_count[word] = 0
                word_count[word] += 1
    return word_count
```

Question 2 (20 points)

Here are the first few rows of `villagers`, a DataFrame of Animal Crossing villagers.

`villagers =`

id	Name	Species	Personality	DaysOnIsland
0	Jitters	Bird	Jock	749
1	Apollo	Eagle	Cranky	242
2	Daisy	Dog	Normal	10

(a) (6 points) Write Pandas code that will evaluate to only the `Name`s of every villager with personality `Jock` and species `Bird`.

Possible solution:

```
villagers.loc[(villagers["Species"] == "Bird") & (villagers["
    Personality"] == "Jock"), "Name"]
```

(b) (8 points) Here are the first few rows of a related DataFrame, `catchphrases`.

`catchphrases =`

	Resident	Catchphrase
0	Apollo	pah
1	Bertha	bloop
2	Daisy	bow WOW

Merge `villagers` and `catchphrases` on the `Name` column from `villagers` and the `Resident` column from `catchphrases`, while preserving all information from both datasets in the resulting DataFrame.

Possible solution:

```
villagers.merge(catchphrases, left_on="Name", right_on="Resident",
   how="outer")
```

(c) (6 points) Write a line of code to find the index of the maximum `DaysOnIsland` of villagers with the `Cranky` personality.

Possible solution:

```
villagers.loc[villagers["Personality"] == "Cranky", "DaysOnIsland"].
   idxmax()
```

Question 3 (20 points)

A DataFrame named `housing` contains information about home prices in different US cities. Here are the first few rows:

`housing =`

	state	city	year	median_price	median_income
0	WA	Seattle	2020	800000	95000
1	WA	Seattle	2021	850000	97000
2	OR	Portland	2020	550000	78000
3	OR	Portland	2021	590000	80000
4	ME	Portland	2020	320000	61000

A student runs the following line of code:

```
housing = housing.set_index(["state", "city"])
```

The resulting DataFrame has a MultiIndex with `state` as the first level and `city` as the second level.

(a) (6 points) Write a single line of Pandas code to calculate the average `median_price` for each (`state`, `city`) combination.

Possible solution:

```
housing.groupby(["state", "city"])["median_price"].mean()
```

(b) (7 points) Your coworker instead writes the line below. How does its output compare to your answer from part (a), and why does their code behave this way?

```
housing.groupby("city")["median_price"].mean()
```

Possible solution: Their code groups by `city` only, so it produces one average per city rather than one per (`state`, `city`) combination. Its output has fewer rows and a single-level index, whereas the part (a) result keeps both index levels.

(c) (7 points) Your coworker wants a line plot of how `median_price` changes over time, with a separate line for each state. They write the code below. What is missing and why? Rewrite the line of code with the necessary additions.

```
sns.relplot(data=housing, x="year", y="median_price")
```

Possible solution: It is missing `kind="line"` (the default is a scatter plot) and `hue="state"` (to draw a separate line per state):

```
sns.relplot(data=housing, x="year", y="median_price", hue="state",
    kind="line")
```

Question 4 (20 points)

The two GeoDataFrames below, `parks` and `blocks`, share the same coordinate reference system (CRS), and contain only the columns specified below. Here are the first few rows of each.

`parks =`

	name	geometry
0	Lincoln Park	`POINT (0.5 0.5)`
1	Cedar Park	`POINT (1.5 0.5)`
2	Elm Park	`POINT (0.5 1.5)`

`blocks =`

	neighborhood	population	geometry
0	Downtown	1200	`POLYGON ((0 0, 1 0, 1 1, 0 1, 0 0))`
1	Downtown	900	`POLYGON ((1 0, 2 0, 2 1, 1 1, 1 0))`
2	Riverside	1500	`POLYGON ((0 1, 1 1, 1 2, 0 2, 0 1))`

(a) (6 points) Using a spatial join, combine each park with the `neighborhood` and `population` of the block it falls within (keep only the matches). Save the result as `parks_joined`.

Possible solution:

```
parks_joined = gpd.sjoin(parks, blocks, how="inner")
```

(b) (6 points) Create a GeoDataFrame `neighborhoods` that is grouped by `neighborhood` (i.e., each row represents a single neighborhood), whose geometry is the neighborhood's blocks combined and whose `population` is their total.

Possible solution:

```
neighborhoods = blocks.dissolve(by="neighborhood", aggfunc="sum")
```

(c) (8 points) Using `parks_joined` and `neighborhoods`, write code that (i) counts the number of parks in each neighborhood, (ii) adds those counts to `neighborhoods` as a `park_count` column (neighborhoods with no parks should be 0), and (iii) draws a choropleth map shaded by `park_count`, with a legend.

Possible solution:

```
park_counts = parks_joined.groupby("neighborhood")["name"].count()
neighborhoods["park_count"] = park_counts
neighborhoods["park_count"] = neighborhoods["park_count"].fillna(0)
neighborhoods.plot(column="park_count", legend=True)
```

Question 5 (20 points)

(a) (10 points) You want to train a decision tree to classify tumors as benign or malignant based on patient symptoms. To start, you set a very large max depth so the tree can fully fit the training data. Given the starter code below, write a full decision-tree pipeline: a train/test split with `test_size = 0.35`, training, and collecting predictions on both the training and test sets.

```
features = data.loc[:, data.columns != 'tumor_class']
labels = data['tumor_class']
```

Possible solution:

```
features = data.loc[:, data.columns != 'tumor_class']
labels = data['tumor_class']
features_train, features_test, labels_train, labels_test =
   train_test_split(features, labels, test_size=0.35)
model = DecisionTreeClassifier(max_depth=100)
model.fit(features_train, labels_train)
train_predictions = model.predict(features_train)
test_predictions = model.predict(features_test)
```

(b) (5 points) You add the code below to your pipeline; it reports a **training accuracy of 1.00** and a **test accuracy of 0.68**.

```
train_accuracy = accuracy_score(labels_train, train_predictions)
test_accuracy = accuracy_score(labels_test, test_predictions)
```

Based on these values, is the model underfitting or overfitting? Explain your reasoning.

Possible solution: Overfitting. The model classifies the training data almost perfectly but does much worse on the test data, which means it has memorized the training set rather than learning patterns that generalize to new data.

(c) (5 points) Based on your answer to part (b), name one change you could make to the model to help it generalize better, and write the single line of code (modifying the model from part (a)) that implements it.

Possible solution: The model is overfitting, so reduce the tree's capacity by lowering `max_depth` to a smaller value (any reasonable smaller depth is fine):

```
model = DecisionTreeClassifier(max_depth=3)
```