CSE 163: Intermediate Data Programming

Practice Exam 2 – 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 `highest_scores` 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 in which each line has the form `name,score` (a student's name and one integer score), and returns a dictionary mapping each student's name to their single highest score. A name may appear on more than one line. You may not use advanced libraries or functions not introduced in this course for this task.

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

```
Alice,85
Bob,90
Alice,92
Bob,88
```

Then calling your function on it should give:

```
print(highest_scores("scores.csv"))
# {'Alice': 92, 'Bob': 90}
```

Complete the following function:

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

Possible solution:

```
def highest_scores(file_path: str) -> dict[str, int]:
    """
    Returns a dictionary mapping each student's name to the
    highest score they appear with in the given CSV file.
    Each line of the file has the form: name,score
    """
    highest = {}
    with open(file_path) as f:
        for line in f:
            name, score = line.strip().split(",")
            score = int(score)
            if name not in highest or score > highest[name]:
                highest[name] = score
    return highest
```

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) We want to merge `villagers` with another DataFrame. Among `"inner"`, `"outer"`, `"left"`, and `"right"`, which value of the `how` parameter should we use if we do not want to lose any information from *either* DataFrame? Explain why.

Possible solution: `"outer"`. An outer join keeps every row from both DataFrames (filling unmatched values with `NaN`), so no information from either side is dropped.

(b) (7 points) Write code that returns a new DataFrame containing only the villagers whose `Personality` is `Cranky`, with each of their `Name`s converted to uppercase.

Possible solution:

```
cranky = villagers[villagers["Personality"] == "Cranky"]
cranky["Name"] = cranky["Name"].str.upper()
```

(c) (7 points) We give `villagers` a MultiIndex with the line below.

```
villagers_mi = villagers.set_index(["Name", "DaysOnIsland"]).sort_index()
```

Write a line of code that filters `villagers_mi` to only the rows where `DaysOnIsland` is at least 100.

Possible solution: `DaysOnIsland` is the second index level, so slice the first level with `slice(None)` and the second with `slice(100, None)`:

```
villagers_mi.loc[(slice(None), slice(100, None)), ]
```

Question 3 (20 points)

Suppose the following DataFrame `gamers` contains information about different players' game characters.

`gamers =`

name	element	class	attack_score
Sheamin	fire	mage	8
Arpan	water	warrior	9
Katie	water	healer	4
Melissa	fire	warrior	7
Patrick	earth	bard	7
Kevin	fire	bard	6
Hannah	earth	mage	5
Adrian	air	warrior	5

(a) (7 points) We want the average attack score for each class, so we use the line below. Explain why this does not give us what we want, then write a corrected line of code.

```
gamers.groupby["attack_score"].mean()
```

Possible solution: The `groupby` is written with square brackets, so it never actually groups by a column; and we want to group by `class` (not `attack_score`) and then select the `attack_score` column:

```
gamers.groupby("class")["attack_score"].mean()
```

(b) (6 points) What is the output of the following line on the same DataFrame? Draw the resulting table.

```
gamers.groupby("class")["element"].count()
```

Possible solution:

class	element
bard	2
healer	1
mage	2
warrior	3

(c) (7 points) A student wants the sum of `attack_score` for every (`class`, `element`) combination. Their desired output is:

class	element	attack_score
bard	earth	7
	fire	6
healer	water	4
mage	earth	5
	fire	8
warrior	air	5
	fire	7
	water	9

They write the line below. Is it correct? Why or why not, and what would you change?

```
gamers.groupby(["element", "class"])["attack_score"].sum()
```

Possible solution: No. The order of the grouping columns determines the order of the index levels in the result. This groups by `element` first, but the desired output is indexed by `class` first. Switch the order of the columns:

```
gamers.groupby(["class", "element"])["attack_score"].sum()
```

Question 4 (20 points)

Below are a GeoDataFrame `counties` (one polygon per county) and a regular DataFrame `census` (population counts, with no geometry). Here are the first few rows of each.

`counties =`

	county	geometry
0	King	`POLYGON ((...))`
1	Snohomish	`POLYGON ((...))`
2	Pierce	`POLYGON ((...))`

`census =`

	county	population	area
0	King	2340000	2307
1	Snohomish	864000	3722
2	Pierce	946000	1807

(a) (6 points) Write a line of code that combines each county's `population` and `area` values to its geometry by joining `counties` and `census` on the `county` column. Save the result as `merged`.

Possible solution:

```
merged = counties.merge(census, on="county")
```

(b) (7 points) Add a `density` column to `merged` that contains each county's `population` divided by its geographic `area`.

Possible solution:

```
merged["density"] = merged["people"] / merged["area"]
```

(c) (7 points) Draw a choropleth of only the counties whose `density` is greater than the median density, shaded by `density`, with a legend.

Possible solution:

```
dense = merged[merged["density"] > merged["density"].median()]
dense.plot(column="density", legend=True)
```

Question 5 (20 points)

(a) (10 points) Your friend wants to predict housing prices with a linear regression model, and has selected the features and label with the starter code below. Complete a full linear-regression pipeline: a train/test split with `test_size = 0.35`, training, and collecting predictions on the test set.

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

Possible solution:

```
features = data.loc[:, data.columns != 'price']
labels = data['price']
features_train, features_test, labels_train, labels_test =
    train_test_split(features, labels, test_size=0.35)
model = LinearRegression()
model.fit(features_train, labels_train)
predictions = model.predict(features_test)
```

(b) (5 points) The dataset contains many noisy, irrelevant features. The model fits the training data almost perfectly but predicts poorly on the test set. What is this called, and briefly explain how the noisy features contribute to it.

Possible solution: Overfitting. With many irrelevant features, the model fits noise in the training data rather than the true underlying relationship, so it does not generalize to new (test) data.

(c) (5 points) A LASSO analysis identifies the four most useful features: `'location'`, `'square_footage'`, `'room_count'`, and `'amenities'`. Write the single line of code that redefines `features` to use only these columns.

Possible solution:

```
features = data[['location', 'square_footage', 'room_count', 'amenities']]
```