CSE 160 Winter 2026 - Final Exam

Full Name: _____

Student UW **Email**: _____@uw.edu

Do not turn the page until you are instructed to do so.

Instructions:

- You have **110 minutes** to complete this exam.
- You must not begin working before time begins, and you **must stop working promptly when time is called**. Any modifications to your exam (writing or erasing) before time begins or after time is called will be reported as academic misconduct to the university.
- The exam is **closed book**, including no calculators, computers, phones, watches or other electronics.
- You are allowed a **single sheet of notes**, no larger than 8.5 x 11 inches, for yourself.
- You may also use the reference sheet (last sheet of the exam packet)
- Turn in ***all sheets*** of this exam—except the reference sheet, which you can tear off—together and in the same order when you are finished.
- You may ***only*** use parts and features of Python that have been covered in class up to this point.
- You may ask questions by raising your hand, and a TA will come over to you.

Initial here to indicate you have read and agreed to these rules: _____

Question	Points
Q1. Expressions	10
Q2. Song Selector	12
Q3. Zoologist	18
Q4. Seattle Weather	10
Q5. Hockey Rivalries	10
Q6. Movie Buff	25
Q7. Book Worm	21

Good luck!

1. Expressions (10 pts). In the table below, fill in the correct value and type (e.g., “string”) for each expression. In cells with multiple lines of code, evaluate the last line of code after the preceding lines are run. In other words, what does the expression on the last line *evaluate* to? If the code in a cell causes an error, write "Error" in the value column and leave the type column blank.

Expression	**Value**	**Type**

`x = ("cats", "dogs")`
 	
`x[0] = "bunnies"`
 	
`x`		

`vals = [10, 20, 30]`
 	
`vals.append([40, 50])`
 	
`len(vals)`		

`menu = {"burger": 15.0,`
`"soup": 7.75}`
 	
`menu["burger"] in menu`		

`[n for n in range(10) if n % 3 == 1]`		

`s = [{1: "a", 0: "b"}, {2: "c"}]`
 	
`s[0][1] + s[1][2] + s[0][0]`		

2. Song Selector (12 points). Consider the following `Playlist` class and the `p1`, `p2`, and `p3` objects:

```
class Playlist:

    def __init__(self, name):
        self.name = name
        self.songs = []

    def add_song(self, song):
        self.songs.append(song)

    def duplicate_last(self):
        if self.songs:
            self.songs.append(self.songs[-1])

    def share_song(self, other, index):
        other.songs.append(self.songs[index])


p1 = Playlist("Chill")
p1.add_song("Stateside")

p2 = p1
p2.duplicate_last()

p3 = Playlist("Study")
p1.share_song(p3, 0)

p3.songs.append("I Just Might")
p1.add_song("Afterparty")
```

After the above code is run, what is printed to the console when each of the following print statements is executed?

Print Statement	**Output**
`print(p1.name, p1.songs)`	
`print(p2.name, p2.songs)`	
`print(p3.name, p3.songs)`	

3. Zoologist (18 points) Kellen is writing code to analyze the data in `animal_data`, a dictionary where each key is a unique animal species and the corresponding value is the species' classification (e.g., reptile, mammal, bird):

```
animal_data = {'crocodile' : 'reptile', 'tortoise' : 'reptile',
               'chameleon' : 'reptile', 'eagle' : 'bird',
               'tiger' : 'mammal', 'elephant' : 'mammal'}
```

However, Kellen noticed some bugs in his implementations of the `group_data` and `analyze_data` functions below. He needs your help identifying and fixing these bugs.

3a. (8 points) The `group_data` function defined below takes a parameter `animal_dict`, a dictionary with a format that matches the `animal_data` example above. The function should return a dictionary where each key is a unique classification (e.g., reptile, mammal, bird) and the corresponding value is a list of all animals in the `animal_dict` dictionary with that classification. For example, given the `animal_data` dictionary defined above, `group_data(animal_data)` should return the following dictionary:

```
{'reptile': ['crocodile', 'tortoise', 'chameleon'],
 'bird': ['eagle'],
 'mammal': ['tiger', 'elephant']}
```

The `group_data` function implementation has **2 bugs**. Your task is to **annotate** (write on) the code below to fix the bugs so that the program behaves as expected. You may add, delete, or move code.

```
1  def group_data(animal_dict):
2
3
4
5      grouped_data = {}
6
7
8
9
10     for animal in animal_dict.values():
11
12
13
14
15         classification = animal_data[animal]
16
17
18
19
20         grouped_data[classification].append(animal)
21
22
23
24
25     return grouped_data
26
```

3b. (10 points) After the `group_data` function is debugged, assume the result of calling `group_data(animal_data)` is assigned to the variable `animals_by_classification`:

```
animals_by_classification = group_data(animal_data)
```

The `analyze_data` function should take a dictionary like `animals_by_classification` as a parameter and return the proportion of warm-blooded animals in the provided data. The animal classifications that are warm-blooded are provided in the `warm_blooded` list.

For example, the proportion of warm-blooded animals in the `animal_data` dictionary on the previous page is 0.5, since there are three cold-blooded animals (reptiles) and three warm-blooded animals (mammals and birds). So `analyze_data(group_data(animal_data))` should return `0.5`.

Kellen noticed **3 bugs** in his implementation of the `analyze_data` function. **Annotate** (write on) the code below to fix the bugs so that the program behaves as expected. You may add, delete, or move code.

```
1  def analyze_data(grouped_data):
2
3
4      warm_blooded = ['mammal', 'bird']
5
6      cold_count = 0
7
8      warm_count = 0
9
10
11
12     for classification in grouped_data:
13
14
15
16         animal_count = grouped_data[classification]
17
18
19
20         if classification == warm_blooded:
21
22
23             warm_count += animal_count
24
25
26         else:
27
28
29             cold_count += animal_count
30
31
32
33     return warm_count / warm_count + cold_count
34
```

4. Seattle Weather (10 points) Write a function called `average_temperature`, which takes two parameters:

- `temp_measurements`, a nested list of integers where each row (inner list) represents the temperature measurements from a particular weather station in Seattle from Monday to Sunday (in that order)
- `day_of_week`, a string (e.g. `"Tuesday"`)

The function should return a float representing the average temperature across all weather stations for the given `day_of_week`. For example, if the following temperature measurements are recorded:

```
weather_station_measurements = [[50, 55, 60, 52, 48, 45, 50],
                                [52, 58, 62, 55, 50, 47, 51],
                                [49, 54, 61, 49, 47, 45, 48]]
```

Then `average_temperature(weather_station_measurements, "Wednesday")` should return `61.0`, since the average temperature on Wednesday (the third column) was `(60 + 62 + 61) / 3 = 61.0`.

Your function must calculate the average across all temperature measurements for the given day, *regardless* of how many weather stations are included in the data. You may assume a valid `day_of_week` is passed in. While not required, **the `lst.index()` method might be helpful here.**

```
def average_temperature(temp_measurements, day_of_week):

    days = ["Monday", "Tuesday", "Wednesday", "Thursday",
            "Friday", "Saturday", "Sunday"]

    # Write your code here
```

5. Hockey Rivalries (10 points) The `pwhl_games` dictionary contains the home game opponents for three Professional Women's Hockey League (PWHL) teams. The key is the name of the home team, and the value is a dictionary mapping each opponent name to the number of times the home team will play them.

```
pwhl_games = {"Frost": {"Torrent": 1, "Goldeneyes": 1, "Victoire": 3},
              "Fleet": {"Sceptres": 2, "Sirens": 1},
              "Torrent": {"Goldeneyes": 2, "Charge": 1}}
```

For example, the Frost will play 1 game against the Torrent, 1 against the Goldeneyes, and 3 against the Victoire in the Frost's home arena.

In order to display all upcoming games, write a function called `upcoming_games` that takes in the nested dictionary and returns a list of tuples. Each tuple represents an upcoming game (as indicated by the nested dictionary), where the first item is the home team name, and the second item is the visiting team name:

```
(home, away)
```

Given the `pwhl_games` nested dictionary above, `upcoming_games(pwhl_games)` should return:

```
[('Frost', 'Torrent'), ('Frost', 'Goldeneyes'), ('Frost', 'Victoire'),
('Frost', 'Victoire'), ('Frost', 'Victoire'), ('Fleet', 'Sceptres'),
('Fleet', 'Sceptres'), ('Fleet', 'Sirens'), ('Torrent', 'Goldeneyes'),
('Torrent', 'Goldeneyes'), ('Torrent', 'Charge')]
```

```
def upcoming_games(home_games):
    # Write your code here
```

6. Movie Buff (25 points) The `movies.csv` file contains the movie name, genre, rating, and release year for various movies:

```
Inception,Sci-Fi,8.8,2010
The Dark Knight,Action,9.0,2008
Frozen,Animation,7.4,2013
Toy Story,Animation,8.3,1995
Interstellar,Sci-Fi,8.6,2014
The Matrix,Sci-Fi,8.7,1999
La La Land,Romance,8.0,2016
```

6a. (15 points) Write a function called `import_movies` that accepts the name of a CSV file as a string and returns a dictionary where each movie (name) in the file is a key. The value corresponding to a given key should be a list of the genre, rating, and release year (in that order) for that movie. For example, `import_movies('movies.csv')` should return:

```
{'Inception': ['Sci-Fi', 8.8, 2010], 'The Dark Knight': ['Action', 9.0, 2008],
 'Frozen': ['Animation', 7.4, 2013], 'Toy Story': ['Animation', 8.3, 1995],
 'Interstellar': ['Sci-Fi', 8.6, 2014], 'The Matrix': ['Sci-Fi', 8.7, 1999],
 'La La Land': ['Romance', 8.0, 2016]}
```

```
# Write your code here
```

6b. (10 points) Lisa wants to filter this movie data based on rating and year. She started writing the function `filter_movies`, which takes three parameters:

- `filename`, the name of the CSV file containing the movie data
- `min_rating`, a float representing the minimum rating for a movie to be included in the filtered list
- `max_year`, an integer representing the latest release year that can still be included in the filtered list

Lisa also assigned the result of calling `import_movies(filename)` to the variable `movies_dict`. For this part, you should assume that the `import_movies` function has been implemented correctly (according to the specifications from the previous page) and you **must** use the `movies_dict` variable in your solution.

Complete the implementation of `filter_movies` so that the function **returns a list of movie names** (in any order) from `filename` that satisfy the following criteria:

- A rating of `min_rating` or *higher*
- A release year of `max_year` or *earlier*

For example the `filter_movies('movies.csv', 8.6, 2013)` should return:

`['Inception', 'The Dark Knight', 'The Matrix']` (where the movies appear in any order)

If no movies satisfy both criteria, then an empty list should be returned.

```
def filter_movies(filename, min_rating, max_year):

    movies_dict = import_movies(filename)

    # Write your code here
```

7. Book Worm (21 points) Consider the following class, which represents a book:

```
class Book:

    def __init__(self, title, author, pages):
        '''
        Initializes a book object with the title, author, and page count.

        Parameters:
        title (str): The title of the book
        author (str): The author of the book
        pages (int): The total number of pages in the book
        '''

        self.title = title
        self.author = author
        self.pages = pages
```

7a. (6 points) In the space below, add a `get_description` method to the `Book` class, which returns a string describing the book in the format `"[TITLE] by [AUTHOR] ([PAGES] pages)"`.

Given the object:

```
book1 = Book("The Hobbit", "J.R.R. Tolkien", 310)
```

The method call `book1.get_description()` should return the string:

```
"The Hobbit by J.R.R. Tolkien (310 pages)"
```

```
# Write your code here
```

7b. (15 points) To keep track of all of her books, Emma needs you to program a `Library` class that can contain multiple instances of the `Book` class. The following code should produce the given outputs (where `>>>` indicates console output resulting from print statements):

```
local_library = Library("Husky Public Library")
book3 = Book("The Martian", "Andy Weir", 369)
local_library.add_book(book3)
local_library.print_catalog()

>>> Husky Public Library Catalog:
>>> The Martian by Andy Weir (369 pages)

local_library.add_book(Book("Night", "Elie Wiesel", 120))
local_library.print_catalog()

>>> Husky Public Library Catalog:
>>> The Martian by Andy Weir (369 pages)
>>> Night by Elie Wiesel (120 pages)
```

Define the `Library` class to behave as indicated in the example above. First, start the class definition and define the appropriate `__init__` method in the space below. A `Library` object should have two instance variables:

- `self.name`, the name of the library
- `self.catalog`, a list containing instances of the `Book` class

```
# Write your code here
```

In addition to an `__init__` method, the class should also have the following two methods:

- `add_book`, which accepts an instance of the `Book` class as a parameter and adds the book to `self.catalog`.
- `print_catalog`, which should print `"[LIBRARY NAME] Catalog:"` followed by the description (returned by the `get_description` method) of each book in `self.catalog`. `Books` should be printed in the order they have been added to the `Library`.

Define the `add_book` method here:

```
# Write your code here
```

Define the `print_catalog` here:

```
# Write your code here
```

[page intentionally left blank]

OPTIONAL (0 points). Draw us a picture :)

CSE 160 26wi Final Exam Reference Sheet

```
# if / elif / else syntax
if condition1:
    # statements
elif condition2:
    # other statements
else:
    # more statements
```

```
# for loop syntax
for i in sequence:
    # statements

# function definition syntax
def function_name(param1, param2, …):
    # statements
```

Sequences and Lists

range([***start***,] ***stop*** [, ***step***])	Returns a sequence of numbers from ***start*** (inclusive) to ***stop*** (exclusive) incrementing by ***step***
len(lst)	Returns the number of elements in ***lst***
lst = []	Creates an empty list
lst[***idx***]	Returns the element in ***lst*** at index ***idx***
lst[***start***:***end***:***step***]	Returns a slice of ***lst*** from index ***start*** (optional) to index ***end*** (exclusive), incrementing by ***step*** (optional)
lst.append(***elmt***)	Adds the ***elmt*** to the end of ***lst***, returns **None**
lst.extend(***sequence***)	Adds each of the elements in ***sequence*** to the end of ***lst***, returns **None**
lst.index(***elmt***)	Returns the index of the first occurrence of ***elmt*** in ***lst***, errors if ***elmt*** is not in ***lst***
lst.count(***elmt***)	Returns the number of times ***elmt*** occurs in ***lst***
lst.remove(***elmt***)	Removes the first occurrence of ***elmt*** from ***lst***, errors if ***elmt*** is not in ***lst***, returns **None**
lst.pop(***idx***) lst.pop()	Removes and returns the element at index ***idx*** in ***lst*** With no parameter, removes the last element in ***lst***
lst.insert(idx, elmt)	Inserts ***elmt*** into ***lst*** at index ***idx***, returns **None**

File I/O

```
file = open(filepath, "r")

# read entire file at once
file.read()
file.close()
```

```
file = open(filepath, "r")

# read line by line
for line in file:
    # process line
```

```
file = open(filepath, "w")

# write entire file at once
file.write(string_to_write)
file.close()
```

Dictionaries

`my_dict = {}` `my_dict = dict()`	Creates a new, empty dictionary
***my_dict*[*key*]**	Returns the value associated with the given ***key*** in ***my_dict***
`del` ***my_dict*[*key*]**	Remove ***key*** (and its associated value) from ***my_dict***
`list(`***my_dict***`.keys())`	Returns a list of keys in ***my_dict***
`list(`***my_dict***`.values())`	Returns a list of values in ***my_dict***
`list(`***my_dict***`.items())`	Returns a list of tuples of the form (***key***, ***value***)
`sorted(`***my_dict***`)`	Returns a sorted list of the keys in ***my_dict***

Sorting

`sorted(`***collection*** `[, `***key***`, `***reverse***`])`	Returns a sorted copy of ***collection***, based on the optional sort key (***key***) and optional order preference (***reverse***)
`lst.sort([key, reverse])`	Sorts the given list (***lst***), based on the optional sort key (***key***) and optional order preference (***reverse***), and returns **None**
key	A function to determine how to compare two values

Classes

`class` ***Name***`:`
`    # class methods`
`    def method(self [, args]):`
`        # method body`
	Defines a new class named ***Name*** with the subsequently defined methods (and attributes)

`def __init__(self [, args]):` `    # method body`	Function that is called during the creation of an instance of the class (e.g. ***Name()***)
self	Required parameter for all methods of a class. Refers to the specific instance of the class. Can hold any number of arbitrary variables (attributes), as in ***self*.my_attribute**
n `= Name()`	Instantiates (creates) a new instance of the ***Name*** class and assigns a reference to it to the variable ***n***
***n*.*method*([*args*])**	Calls ***method*** on the instance of the class (***n***), optionally passing in any required arguments. Inside the function, **self** is assigned to ***n***