CSE 160 Winter 2026 - Midterm Exam **KEY**

Full Name: _____

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

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

Instructions:

- You have **the entire class period (50 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. Magic Powers	10
Q3. Grocery Shopping	10
Q4. Friends	25
Q5. Missing Assignments	20

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

`a = 4`
`b = 2`
`a / b + 1`
	3.0	**float**

`str(7 % 2) + '8'`	**'18'**	**string**

`c = '12.5'`
`d = 13`
`if c < 15:`
`    d = 22`
`d`
	Error	

`list(range(20, 0, -5))`	**[20, 15, 10, 5]**	**list**
`[1, 2, 3, 4, 5].index(2)`	**1**	**int**

2. Magic Powers (10 points). Consider the following functions:

```
def healing(health):
      health += 3
      print("Healing activated")

def strength(x):
      boost = x * 2
      return boost + 1

def levitation(x, power):
      x = strength(x)
      print(power)
      return x < len(power)

def next_spell(health, mana):
      if health < 10:
            healing(health)
            print("Health:", health)
      elif mana % 2 == 0:
            return levitation(mana // 2, "+++")
      return strength(mana)
```

What is printed to the console when each of the following statements is executed? Place each new line of output on a separate line.

Print Statement	**Output**
`print(healing(16))`	**Healing activated** **None**
`print(strength(6))`	**13**
`print(levitation(1, "++\n++"))`	**++** **++** **True**
`print(next_spell(4, 9))`	**Healing activated** **Health: 4** **19**
`print(next_spell(12, 6))`	**+++** **False**

3. Grocery Shopping (10 points). James is trying to find deals for the next time he goes grocery shopping. He decides to write a function named `search_grocery`. This function should accept two arguments:

1. **`filename`**, the name of a CSV file containing information about grocery store items.
2. **`category`**, the name of a particular grocery store section (e.g. “Produce”).

Each CSV file James uses with this function will have the following information on each line, separated by commas:

```
item_name,category,weight,price
```

For example, the `grocery.csv` file looks like this:

```
Tangerines,Produce,1,2.99
Frozen Shrimp,Seafood,1,12.99
Blueberries,Produce,0.5,5.99
Shampoo,Hygiene,0.75,7.49
```

The `search_grocery` function should search the CSV file (indicated `filename`) for items in the corresponding `category` (e.g. `"Produce"`, `"Seafood"`, `"Hygiene"`). **The function should print information about each product from that category in the following format**:

```
<name> are $<price> for <weight> lb
```

If there aren’t any products in `category`, the function should print:

```
There aren’t any products of that category in the store!
```

Finally, the `search_grocery` function should **return the number of products in the `category`** in the CSV file.

Below are two examples of the intended functionality of `search_grocery`, using the `grocery.csv` file:

Function Call	**Print Output**	**Return Value**
`search_grocery("grocery.csv", "Produce")`	`Tangerines are $2.99 for 1 lb` `Blueberries are $5.99 for 0.5 lb`	2
`search_grocery("grocery.csv", "Hygiene")`	`Shampoo are $7.49 for 0.75 lb`	1
`search_grocery("grocery.csv", "Meat")`	`There aren’t any products of that category in the store!`	0

Lacking sufficient coffee, James’ implementation of the `search_grocery` function (on the next page) **contains 4 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.

```
def search_grocery(filename, category):

    file = open(“filename“)

    count = 0

    for line in file:

        tokens = line.strip().split(",")

        if tokens[1] == category:

            print(tokens[0] + " are $" + tokens[3] + " for " + tokens[2] + "
lb")

            count += 1

    if count == 0: # un-indent one level (including body)

        print("There aren’t any products of that category in the store!")

    file.close()

    return count

    file.close()
```

4. Friends (25 points). You are working with the nested friends list structure discussed in section. This nested list consists of lists containing each person's friends, with their own name at index 0. For example:

```
friends_lists = [["Brianna", "Daniella", "Emma", "Varun"],
                 ["Riya", "Katie", "Arpan", "Shengyi"],
                 ["Sara", "Kellen", "Suhas", "Lisa", "Arpan"]]
```

The first nested list contains Brianna's friends, the second contains Riya's friends, and so on.

4a. (10 points) First, write a function named `lookup` that takes a single list of strings and a string called `target`, which is the name of a particular person. The function should return the name of the person whose friends list is passed in (i.e., the first name in the list) ***only*** if `target` is one of their friends. Otherwise, the function should return `None`. You can assume that the list contains at least two elements.

For example:

`lookup(friends_lists[1], "Arpan")` should return `"Riya"` since Arpan ***is*** on Riya's friends list.

`lookup(["James", "Ruth"], "Adrian")` should return `None` since Adrian *is* ***not*** on James' friends list.

`lookup(friends_lists[0], "Brianna")` should return `None` since Brianna cannot be friends with herself.

```
# Write your code here
def lookup(friend_list, target):
    if target in friend_list[1:]:
        return friend_list[0]


# ALTERNATE VERSION:
def lookup(friend_list, target):
    for friend in friend_list[1:]:
        if friend == target:
            return friend_list[0]
```

4b. (15 points) Now, write a function named `find_friends` that takes a nested list (like `friends_lists`) and a string called `target`. The function should **return a list containing the friends whose list `target` appears in**. If `target` does not appear as anyone's friend, the function should return an empty list.

For example:

`find_friends(friends_lists, "Kellen")` should return `["Sara"]`, since Kellen only appears in Sara's friends list.

`find_friends(friends_lists, "Arpan")` should return `["Riya", "Sara"]`, since Arpan appears in both Riya and Sara's friends lists.

`find_friends(friends_lists, "James")` should return `[]`, since James does not appear in any friends list.

`find_friends(friends_lists, "Brianna")` should return `[]`, since Brianna doesn't appear in anyone else's friends list (and Brianna can't be friends with herself).

You *can* call the `lookup` function (which you implemented on the previous page) in your implementation of the `find_friends` function, although this is not *required*.

```
# Write your code here
def find_friends(lists_of_friends, target):
    friends = []
    for friend_list in lists_of_friends:
        friend = lookup(friend_list, target)
        if friend != None:
            friends.append(friend)
    return friends
```

5. Missing Assignments (20 points). Write a function `missing_work` that takes in one parameter `students`, which is a nested list, and returns a list containing the names of students with scores equal to 0. Each element of `students` is a list containing a student's name, major, and assignment score, in that order.

For example:

```
students = [["Marcello", "HCDE", 78],
            ["Sabrina", "CSE", 0],
            ["John", "INFO", 0],
            ["Olivia", "ECON", 85]]
```

The function should **return a list of the names of all students whose assignment score is equal to 0.** (maybe they forgot to submit the assignment!)

For example:

`missing_work(students)` should return `["Sabrina", "John"]`

`missing_work([])` should return `[]`

You must use `range()` to iterate through *any* list in order to receive full credit.

```
# Write your solution here
def missing_work(students):
    missing_students = []
    for i in range(len(students)):
        student_name = students[i][0]
        student_score = students[i][2]
        if student_score == 0:
            missing_students.append(student_name)
    return missing_students
```

[page intentionally left blank]

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