CIS1100.py — Fall 2024 — Final Exam **Solutions**

Full Name: _____

PennID (e.g. 12345678): _____

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

_____ _____

Signature Date

Instructions are below. Not complying will lead to a 0% score on the exam.

- Do not open this exam until told by the proctor.
- You will have exactly 90 minutes to take this exam.
- Make sure your phone is turned OFF (not on vibrate!) before the exam starts.
- Food and gum are not permitted—don't be noisy or messy.
- You may not use your phone or open your bag for any reason, including to retrieve or put away pens or pencils, until you have left the exam room.
- This exam is closed-book, closed-notes, and closed computational devices.
- If you get stuck on a problem, it may be to your benefit to move on to another question and come back later.
- All code must be written in proper Python format.
- Do not separate the exam pages. Do not take any exam pages with you. The entire exam packet must be turned in as is.
- Only answers on the FRONT of pages will be graded. There are two blank pages at the end of the exam if you need extra space for any graded answers.
- Use a pencil, or blue or black pen to complete the exam.
- If you have any questions, raise your hand and a proctor will come to you.
- When you turn in your exam, you may be required to show your PennCard. If you forgot to bring your ID, talk to an exam proctor immediately.
- We wish you the best of luck!

Q1	Q2	Q3	Q4	Q5	Q6	Q7 (bonus)

Q1. Fill In The Blank

For Q1.1-1.4, write the value that gets printed.

Question 1.1

```
def mystery(name: str) -> str:
    result = ''
    for i in range(len(name)):
        result += name[i:]
    return result

print(mystery("Adi"))
```

Answer: Adidii

Question 1.2

```
def mystery_rec(l: list[int], k: int):
    if l == []:
        return []
    x = l[0]
    rest = l[1:]
    if x < k:
        return [x] + mystery_rec(rest, k)
    else:
        return mystery_rec(rest, k)

print(mystery_rec([2, 1, 8, 4], 3))
```

Answer: [2, 1]

Question 1.3

```
my_func = lambda x, y: x + (y + y)
my_vals = {3, 2, 1}
jessica_vals = {3, 4, 2, 0}
our_input = list(my_vals | jessica_vals)
print(reduce(my_func, our_input, 0))
```

Answer: 20

Question 1.4

```
faves = dict()
name = "harry"
faves[name] = "Python"
name = "travis"
faves[name] = "C++"
print(faves["harry"])
```

Answer: "Python"

Question 1.5 & 1.6

```
html_string = """
<div class="container">
  <header>
    <nav>
      <a href="/home">🏠</a>
      <p>Hi!</p>
      <a href="/about">ℹ️</a>
    </nav>
  </header>
  <table border="1">
    <tr><td>🌟</td><td>✨</td></tr>
    <tr><td>🎈</td><td>🎪</td></tr>
    <tr><td>🌸</td><td>🍀</td></tr>
    <tr><td>🌞</td><td>🌙</td></tr>
  </table>
</div>"""
soup = BeautifulSoup(html_string, 'html.parser')
print(soup.find("a")["href"])
print(len(soup.find_all("tr")))
```

First Printed Line: /home

Second Printed Line: 4

Q2. Shopping on a Budget (Debugging)

Christian has a shopping list of items to buy from Amazon, but his friend Krishna has a bunch of items that he's willing to sell. For each item on Christian's list, he'll need to find the cheapest option available by calculating each item's total price from its base price **and** its shipping cost. That option might be the one on his current shopping list, or it might be one that Krishna is offering from his inventory.
To solve this problem, Christian defines an Item dataclass:

```
@dataclass
class Item:
    kind: str
    price: float
    shipping: float
```

Help him debug his shopping function so that it returns a new Item list that always contains the cheapest Items of each kind he needs. (You can assume that Christian always needs to buy exactly one of each kind of item included in shop_list; in other words, it has no duplicates!)

In the table that follows, identify the line on which each of the bugs appears, the part of the code that is incorrect, and the code you can replace that part with to make it correct. We filled in an example bug on the first line. Note that there are no further bugs in the function signature. **There are six other bugs.**

```
1. DEF shopping(shop_list: list[Item], inventory: list[Item]) -> list[Item]:
2.     best_list = {}
3.     for curr_item in shop_list:
4.         best_item = curr_item
5.         for alt_item in range(inventory):
6.             best_cost = best_item[price] + best_item[shipping]
7.             alt_cost = alt_item.price + alt_item.shipping
8.             same_kind = best_item.kind + alt_item.kind
9.             cheaper = best_cost > alt_cost
10.            if same_kind and cheaper:
11.                 alt_item = best_item
12.        best_list.append(best_item)
13.    return best_item
```

Line Number	Incorrect Code	Replacement Code
1	`DEF`	`def`
2	`{}`	`[]`
5	`range(inventory)`	`inventory`
6	`best_item[price] + best_item[shipping]`	`best_item.price + best_item.shipping`
8	`+`	`==`
11	`alt_item = best_item`	`best_item = alt_item`
13	`best_item`	`best_list`

Q3. Complete the Program

Molly wants to search over several files to see which files contain which words. To do this she builds up a structure of type `dict[str, dict[str, int]]`. That is, she uses a dictionary that maps words to another dictionary that maps a filename to how often that word shows up in the file.

For example, if we had the document “example.txt” which had the contents “hi hi bye” and the file “bye.txt” that just had the contents “good bye”, we would have the structure to the right:

```
{
    "hi": {
        "example.txt": 2
    },
    "bye": {
        "example.txt": 1,
        "bye.txt": 1
    },
    "good": {
        "bye.txt": 1
    }
}
```

Molly's attempt to solve this problem starts on the next page. She left a few blanks in her attempt. Fill in the blanks using the table at the end of this page to help her complete the FileSearcher class!

```
class FileSearcher:
    def __init__(self):
        # self.word_locations will be dict[str, dict[str, int]]
        self.word_locations = dict()

    # Add all words from a new file into self.word_locations
    def record_file(self, fname: str):
        file = __BLANK_0__ # fname is a string, so need to open file
        contents = file.read()
        words = contents.split()
        for word in words:
            if word not in self.word_locations:
                __BLANK_1__ = dict()
            if fname __BLANK_2__:
                self.word_locations[word][fname] = 1
            else:
                __BLANK_3__
        file.close()
```

Blank #	Code
0	`open(fname, "r")`
1	`self.word_locations[word]`
2	`not in self.word_locations[word]`
3	`self.word_locations[word][fname] += 1`

Q4: Pizza Pandas-monium

Harry likes trying all the pizza in Philadelphia and recording his preferences. He notes the name of the pizzeria, the style of pizza he had there, the rating he gives that pizza, and the number of times he's tried that pizza. Some pizzerias offer multiple styles of pizza, so the same pizzeria may appear multiple times in the DataFrame. He maintains a CSV called **pizza.csv**, which he reads into a DataFrame called **pizza**. The first few rows of that DataFrame are depicted in the table below.

INDEX	place	style	rating	times
0	"Angelo's"	"NYC"	4.1	4
1	"Paulie Gee's"	"NYC"	4.0	5
2	"Paulie Gee's"	"Brooklyn"	4.4	5
3	"CJ & D's"	"Trenton"	4.5	1

Solve each of the following tasks using Pandas. You can refer to the appendix for some tools that you'll need. The later tasks can be solved using columns created in the earlier tasks, and we will grade the later tasks assuming that you completed the earlier ones correctly.

Some pizza places offer quite a number of different styles. "Dominos" is one such pizzeria. Write some pandas code that would count the number of different styles that Dominos offers by counting the number of rows where the *place* is Dominos.

```
pizza[pizza["place"] == "Dominos"]
```

Add a column to the DataFrame called *favorite*, which contains True if Harry rates a style from a pizzeria above a 4.2 and has tried it more than once. Otherwise, the value in that column for that row will be False.

```
pizza["favorite"] = (pizza["rating"] > 4.2) & (pizza["times"] > 1)
```

Write a few lines of Pandas code that ends by **printing** the fraction of rows in the DataFrame that represent a *favorite*. You can assume that the *favorite* column was created correctly.

```
faves = pizza[pizza["favorite"]].shape[0]
all = pizza.shape[0]
print(faves / all)
```

Q5. Pricey Pizzas

Here's a class that models the concept of a person's favorite pizza in terms of its size (diameter in inches) and list of toppings. The initializer for this class is defined before you. Your job will be to test and then complete the calculate_price method.

```
class Pizza:
    def __init__(self, size: int, toppings: list[str]):
        self.size = size
        self.toppings = toppings

    def calculate_price(self, ingredient_prices: dict[str, float]) -> float:
        ...
```

calculate_price takes in the prices of ingredients as set by a particular restaurant and calculates how much it will cost to purchase this pizza at those ingredient prices. The total price of a pizza is based on its size—$2 per inch, making a 10-inch pizza without toppings cost $20—as well as its toppings. If a topping is not listed in the ingredient_prices dict, then it can be assumed to cost $0.00.

Question 5.1

Verify your understanding of the above by completing the following two unit tests. In both cases, you should write a test that:

1. calls the method on the provided inputs, &
2. verifies that the output of the method matches what should be expected given the specification above.

```
def testOne(self):
        harrys_fave = Pizza(10, ["sausage", "onion", "pepper flakes"])
        angelos_prices = {"sausage": 2, "onion": 0.5, "pepperoni": 2}
        # TODO: Complete me!

        actual = harrys_fave.calculate_price(angelos_prices)
        expected = 22.5
        self.assertEqual(expected, actual)
```

```
def testTwo(self):
        travis_fave = Pizza(14, ["anchovies", "garlic", "olives"])
        dominos_prices = {"anchovies": 1, "onion": 0.25, "olives": 3}
        # TODO: Complete me!

        actual = travis_fave.calculate_price(dominos_prices)
        expected = 32
        self.assertEqual(expected, actual)
```

Question 5.2

Implement calculate_price below based on the specification from the previous page.

```
def calculate_price(self, ingredient_prices: dict[str, float]) -> float:
        total_price = 2.0 * self.size
        for topping in self.toppings:
            if topping in ingredient_prices:
                total_price += ingredient_prices[topping]
        return total_price
```

Q6. Coding

Travis teaches a course that has a group project. Groups can be of various sizes, so he represents each group as a tuple of names and then keeps those tuples in a list.

For example:

```
[ ("Harry", "Jessica", "Michael"),
  ("Adi", "Sukya", "Cedric", "Molly", Jared") ]
```

The above list has two groups, one with three people in it, the other with five people in it.

Question 6.1

Travis wants groups to have exactly four people in them. To help make sure that groups were formed correctly, he wants to write the function `groups_need_rebalancing`, which takes in the previously mentioned list of tuples and returns `True` if there is at least one non-empty group with less than four members in it **and** there is at least one group with more than four members in it. The function returns `False` otherwise.

An example implementation of this function would be:

```
def groups_need_rebalancing(groups: list[tuple[str]]) -> bool:
    has_small_group = False
    for group in groups:
        has_small_group = has_small_group or (4 > len(group) > 0)

    has_big_group = False
    for group in groups:
        has_big_group = has_big_group or (len(group) > 4)

    return has_small_group and has_big_group
```

Your job is to complete this function **without using any loops.** Recursion, list/set/dict comprehensions, lists, sets, dicts, higher order functions, lambdas and python built-in functions are all OK.

```
def groups_need_rebalancing(groups: list[tuple[str]]) -> bool:
    small_groups = filter(lambda t: 4 > len(t) > 0, groups)
    big_groups = filter(lambda t: len(t) > 4, groups)
    return len(small_groups) > 0 and len(big_groups) > 0)
```

Question 6.2

Travis wants to write a function `find_largest_group()` that is given a list like the one above and it returns the group (tuple) that has the most people in it.

You can assume that there will be at least one non-empty group in the input list. If there is a tie for which group has the most people, you can choose one of the groups that have the largest length

An example implementation of this function would be:

```
def find_largest_group(groups: list[tuple[str]]) -> tuple[str]:
    largest_group = groups[0]
    for group in groups[1:]:
        if len(group) > len(largest_group):
            largset_group = group
    return largest_group
```

Your job is to complete this function **recursively**. You are free to define any recursive helper function you wish. You are not permitted to use any loops, comprehensions, higher order functions, or python built-in functions (other than `len()`)

```
def find_largest_group(groups: list[tuple[str]]) -> tuple[str]:
    if (len(groups) == 1):
        return groups[0]
    curr = groups[0]
    rest = find_largest_group(groups[1:])
    if (len(curr) > len(rest)):
        return curr
    else:
        return rest
```

Q7. Bonus

Share something that you are proud of accomplishing from this year. Or, if you're not up for reflection, pictures are always fun. In any case: all exams will get full credit for this question, even if the question is left blank.

Extra Answers Page (This page is intentionally blank)

You may use this page for additional space for answers; keep it attached to this exam. Clearly note on the original question page that your answer is on this extra page, and clearly note on this page what question you are answering.