Lab 03: List Comprehension and Files

Total Points: 10 (Correctness) + 1 extra credit (optional, early submission) + 1 extra credit (optional, problem 6.2)

Due: Tuesday, January 30th, 11:59pm (SD Time)

Starter Files

Download lab03.zip. Inside the archive, you will find starter files for the questions in this lab. You can't import anything to solve the problems.

Extra Credit Opportunity

Starting from lab02, you will have the opportunity to receive a **1 point** extra credit on that lab if you submit your last attempt early (refer to this section of each lab for the early submission deadline). Note: Each lab is graded out of 10 points, and you could possibly have more than 10 points in one lab.

Early Submission Date (lab03): Friday, January 26th, 11:59pm (SD time)

Logistics

Using Python

When running a Python file, you can use the **options** on the command line to inspect your code further. Here are a few that will come in handy. If you want to learn more about other Python command-line options, take a look at the documentation. For all commands below, use `py` instead of `python3` if you are using a Windows system.

- Using no command-line options will run the code in the file you provide and return you to the command line.
  ```
  >>> python3 lab03.py
  ```
- **-i**: The `-i` option runs your Python script, then opens an interactive session. In an interactive session, you run Python code line by line and get immediate feedback instead of running an entire file all at once. To exit, type `exit()`

into the interpreter prompt. You can also use the keyboard shortcut `Ctrl-D` on Linux/Mac machines or `Ctrl-Z Enter` on Windows.
If you edit the Python file while running it interactively, you will need to **exit** and **restart** the interpreter in order for those changes to take effect.
`>>> python3 -i lab03.py`

- **-m doctest**: Runs doctests in a particular file. Doctests are surrounded by triple quotes (`"""`) within functions. Each test consists of `>>>` followed by some Python code and the expected output.
`>>> python3 -m doctest lab03.py`

After finishing each question, please run the doctests using the command above to check the correctness of the current question.

Important Requirements and Notes

1. **DO NOT IMPORT ANY PACKAGES.**
2. **There are special syntax instructions for list comprehension questions (Questions 1-5).**
3. It may be useful to add doctests when the given doctests do not cover all cases to avoid losing points from the Autograder/any hidden tests, but you are not required to add doctests in the labs.
4. Style is not required or graded on any labs, but it's recommended that you also follow the style guide for clean code. Method descriptions are provided in this lab.
5. You may assume all inputs are valid.

Special Instructions for One-Line List Comprehension

Important Note: Please read the instructions below for questions that require **one-line list comprehension**. If you failed to follow them, you will not receive credit on that question. Read Question 4.2 on the Style Guide Example for reference.

Incorrect	Correct Example(s)	Note
`lst = [...]` `return lst`	`return [...]`	

`lst_1 = [...]`
`lst_2 = [...]`
`return lst_1 + lst_2`
	`return [...] + [...]`	This is allowed for one-line requirement

`# ..... (comment)` `return [...]`	`return [...]`	Remove any comment
	`# sum of squares` `return sum([num ** 2 for num in lst])`	You can do anything on the same line to the list comprehension, as long as everything fits in the same one line.

Requirements for the rest of the questions in the lab:

Your implementation should only return an expression. Your implementation should fit in exactly one line, and break into multiple lines if space is limited.

- **No explicit loops** (for loop or while loop) are allowed.
- Do **NOT** add any inline comments.
- **There are examples of correct syntax at the top of the writeup.**

Required Questions

It's winter in Pythonia, a season for rejoicing in all things frosty: snowflakes, hot cocoa, ice skating, cozy blankets, and so much more. Today, immerse yourself in a day of winter wonder at the Pythonia Frost Festival.

Question 1:

First, you decided to explore the winter market where festive items like ornaments, evergreen trees, and seasonal treats are sold.

Question 1.1:

You started wandering around and noticed that every store had a slogan to advertise their Winter items, but these slogans did not look nicely designed. So you offered to help these stores. Write a function that takes a list of strings (`slogans`) and returns another list, where each slogan has capitalized letters.

REQUIREMENTS: Use one line list comprehension only

```
def updated_slogans(slogans):
    """
    >>> slogans = ["best SweaTERs", "Cheap and WARM"]
    >>> updated_slogans(slogans)
    ['BEST SWEATERS', 'CHEAP AND WARM']
    >>> slogans = ["", "SleiGH RiDe"]
    >>> updated_slogans(slogans)
    ['', 'SLEIGH RIDE']
    >>> slogans = ["!!!", "BeWaRE! DANger ahead", "Candy CANE lane"]
    >>> updated_slogans(slogans)
    ['!!!', 'BEWARE! DANGER AHEAD', 'CANDY CANE LANE']
    """
```

Question 1.2:

While you were helping different stores with their slogans, you found ornaments that you really liked and recorded their prices. Write a function that takes a list of positive integers (`prices`) and outputs the total price but you do not want to pay more than $20 for an ornament. Therefore a price greater than $20 should not be included in the total.

Example:

Input	Output	Reason
`[3, 40, 10, 2]`	15	40 is greater than 20, so we did not include it in the total. 3 + 10 + 2 = 15
`[4, 40, 20, 1]`	25	40 is greater than 20, while 4, 20, and 1 are not greater than 20.

		4+20+1 = 25

REQUIREMENTS: Use one line list comprehension only

```
def prices_total(prices):
      """
      >>> prices = [3, 40, 10, 2]
      >>> prices_total(prices)
      15
      >>> prices = [4, 40, 20, 1]
      >>> prices_total(prices)
      25
      >>> prices = []
      >>> prices_total(prices)
      0
      """
```

Question 1.3:

Store owners were so impressed with their changed slogans, that they gave you some money for your work. It means that you can actually buy some ornaments for real! Since the stores were so nice to you, you decided to buy one ornament from each store but only if it is within a budget.

Write a function that takes a list of tuples, and each tuple has three items: ornament type, price and allotted budget for this store:

`[(type, price, budget_for_store_1), (type, price, budget_for_store_2), (type, price, budget_for_store_3),...]`

Your function should return a new list where each item is either an ornament type if you can afford it, or a string 'too pricey' if you can't.

REQUIREMENTS: Use one line list comprehension only

```
def buying_ornaments(stores):
      """
```

```
    >>> stores = [("heavy", 10, 10), ("light", 5, 10), ("heavy", 100,
20)]
    >>> buying_ornaments(stores)
    ['heavy', 'light', 'too pricey']

    >>> stores = [("orange", 20, 0), ("white", 0, 0), ("green", 100,
10)]
    >>> buying_ornaments(stores)
    ['too pricey', 'white', 'too pricey']

    >>> stores = [("small", 20, 10), ("huge", 15, 14), \
("polka dot", 10, 150), ("purple", 100, 150), ("festive", 277.1, 277)]
    >>> buying_ornaments(stores)
    ['too pricey', 'too pricey', 'polka dot', 'purple', 'too pricey']
    """
```

Question 2:

Congratulations, you've selected the ornaments you are taking home. However, these ornaments of the magical kingdom of Pythonia are unique. Sometimes they get **really** heavy, so you want to figure out if you would be able to bring the ornament home by yourself, with friends' help, or have it delivered by professionals. Given a list of ornament weights (integers or floats), return the list of how you can take each ornament home using the threshold reference table below.

Notes: If you are stuck on how to use multiple if-else conditions in a list comprehension check this file Multiple If Else Conditions

Threshold	Output
? ≤ 40	"I got it"
40 < ? ≤ 80	"Help me"
80 < ?	"Call delivery"

REQUIREMENTS: Use one line list comprehension only

```
def delivery(ornament_weights):
    """
    >>> delivery([22, 40, 55, 91])
```

```
['I got it', 'I got it', 'Help me', 'Call delivery']
>>> delivery([])
[]
>>> delivery([0, 100000])
['I got it', 'Call delivery']
"""
```

Question 3:

Luckily your friends like you, so they are willing to help for free, but the delivery people need to get paid. Write a function that calculates the cost of a delivery, which will depend on the size and weight of the ornament. Your input will be a list of lists, `ornaments`, where the first list is the weights (positive numbers) and the second list represents the sizes (positive numbers). Calculate the price of the delivery in MNT (Monty, Pythonia currency) for each ornament according to the formula below.

$$cost = (weight ** size) / (10 ** size)$$

You can assume that the input will always be a list of two lists of the same length.

Example:

Input	Output	Reason
`[[22, 40], [1, 2]]`	`[2.2, 16]`	The first value is 2.2 because (22 ** 1) / (10 ** 1) = 22/10 = 2.2 The second value is 16 because (40 ** 2) / (10 ** 2) = 1600 / 100 = 16

NOTE: Round your answer to the nearest thousandth (3 decimal places).

REQUIREMENTS: Use one line list comprehension only

```
def prices(ornaments):
    """
    >>> prices([[22, 40, 55, 91], [1, 2, 3, 3]])
    [2.2, 16, 166.375, 753.571]
```

```
    >>> prices([[], []])
    []
    >>> prices([[61], [5]])
    [8445.963]
    """
```

Question 4:

After calculating the delivery fees, you contact the delivery professional to schedule a delivery. However, due to the high demand, delivery companies charge various fixed fees for peak time service. Write a function that calculates the prices of peak delivery periods corresponding to all companies for each ornament. The function takes the following parameters:

- A list of numbers, `prices`, which you have calculated in Question 3.
- A dictionary, `company_fees`, that takes company's names as indices and their extra fixed fees as values.

Return a list of lists, where each list contains peak period prices for all ornaments corresponding to one company provided.

Example:

Input	Output	Reason

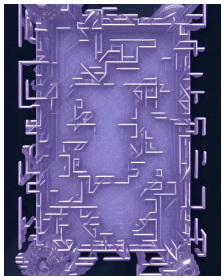
`prices = [1, 2, 2]`
`company_fees =`
`{'USPS': 3, 'FedEx': 2, 'UPS': 1.5}`
	`[[4, 5, 5], [3, 4, 4], [2.5, 3.5, 3.5]]`	The first list is the adjusted prices for USPS, so it becomes
		[1+3, 2+3, 2+3] → [4, 5, 5]
		
		The second list is the adjusted prices for FedEx, so it becomes
		[1+2, 2+2, 2+2] → [3, 4, 4]
		
		The third list is the adjusted prices for UPS, so it becomes
		[1+1.5, 2+1.5, 2+1.5] → [2.5, 3.5, 3.5]

REQUIREMENTS: Use one line list comprehension only

```
def adjusted_prices(prices, company_fees):
    """
    >>> prices = []
    >>> company_fees = {'Santa Express': 1, 'UPS': 1.5, 'USPS': 3}
    >>> adjusted_prices(prices, company_fees)
```

```
    [[], [], []]

    >>> prices = [1, 2, 2]
    >>> company_fees = {'USPS': 3, 'FedEx': 2, 'UPS': 1.5}
    >>> adjusted_prices(prices, company_fees)
    [[4, 5, 5], [3, 4, 4], [2.5, 3.5, 3.5]]

    >>> prices = [2.2, 16, 166.375, 753.571]
    >>> company_fees = {'Santa Express': 1, 'UPS': 1.5, 'USPS': 3}
    >>> adjusted_prices(prices, company_fees)
    [[3.2, 17, 167.375, 754.571], [3.7, 17.5, 167.875, 755.071], \
[5.2, 19, 169.375, 756.571]]
    """
```

Question 5:



You called the delivery professional with the lowest price and they picked up your ornaments. Now, you can immerse yourself into the Pythonia Frost Festival. You decide to play an ice maze with your DSC20 tutors.

Question 5.1:

At the start of the maze, you are given an ordered sequence of encoded messages on the way out of the maze and a lookup table to decode messages, where the ordering of messages corresponds to the order of crossroads along the way. Write a function that takes in a list of 8 bit bit-strings (8 characters that are either 1 or 0), `encoded_messages`, and a dictionary, `decode_scheme`, as a lookup table, where:

- Each bit-string in `encoded_messages` is a sequence of zeros and ones, represented as a string of length 8.
- `decode_scheme` has integers from 0 to 5 as keys and direction strings as values, where '>' means go right, '<' means go left, and '+' means go straight.

The function would follow a decoding scheme and decode each message in the following steps:

1. Compute the number of 1's in the bit-string.
2. Compute modulo 6 of the number of 1's in each bit-string.
3. Use the result from step 2 to look up the decoded direction in `decode_scheme`.

Your function returns a list of directions that corresponds to each encoded message. The order stays the same.

Notes:

- It is possible for `encoded_messages` to be empty – some ice mazes may not have a single crossroad! In this case, your function returns an empty list.
- To generalize, the input `encoded_messages` does not need to have a fixed length, though a given ice maze has a fixed number of crossroads.
- You may find the function `count()` helpful in finding the number of 1's in each bitstring.

Example:

Input	Output	Reason
`encoded_messages = ['11111111', '10111111']` `decode_scheme = {0: '+', 1: '>', 2: '+', 3: '<', 4: '<', 5: '>'}`	`['+', '>']`	The first value is "+" because the first string has 8 ones, 8 % 6 = 2, the key 2 in the decode_scheme corresponds to the "+" symbol The second value is ">" because the second string has 7 ones, 7 % 6 = 1, the key 1 in the decode_scheme corresponds to the ">" symbol.

REQUIREMENTS: Use one line list comprehension only

```
def decode_directions(encoded_messages, decode_scheme):
    """
    >>> encoded_messages = []
    >>> decode_scheme = {0: '+', 1: '>', 2: '+', 3: '<', 4: '<', 5:
'>'}
    >>> decode_directions(encoded_messages, decode_scheme)
    []

    >>> encoded_messages = ['11111111', '10111111', '01000011',
'10101111']
    >>> decode_directions(encoded_messages, decode_scheme)
```

```
    ['+', '>', '<', '+']

    >>> encoded_messages = ['10010000', '00100111', '01100101',
'11001001']
    >>> decode_scheme = {0: '+', 1: '+', 2: '+', 3: '+', 4: '+', 5:
'+'}
    >>> decode_directions(encoded_messages, decode_scheme)
    ['+', '+', '+', '+']
    """
```

Question 5.2:

You've repeated the process for every encoded message. Now you have a list of directions that you can rely on when you are stuck. After exiting the ice maze, you feel you chose "go straight" more than the other two directions when facing a crossroad. Therefore, you want to check the proportion of times you go straight when facing a crossroad – in order to find the exit of the ice maze.

Write a function that takes in a list of `directions` you have decoded. Return the proportion of times you go straight (represented by `'+'` symbol) when facing a crossroad, rounded to 2 decimal places if there are more decimal places – remember how you did that in HW1! If the input list, `directions`, is empty, return 0.0.

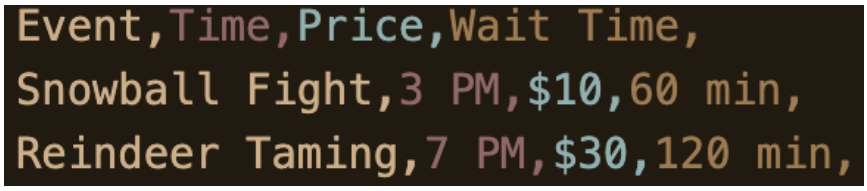
REQUIREMENTS: You must use list comprehensions here (not explicit loops) but you can use multiple list comprehensions if needed.

```
def prop_straight(directions):
    """
    >>> directions = []
    >>> prop_straight(directions)
    0.0

    >>> directions = ['+', '<', '+']
    >>> prop_straight(directions)
    66.67

    >>> directions = ['+', '>', '<', '+']
    >>> prop_straight(directions)
    50.0
```

```
    >>> directions = ['+', '+', '+', '+']
    >>> prop_straight(directions)
    100.0
    """
```

Question 6:

You have the data of all of the activities you would like to do stored in a `csv` (comma separated values) file that looks like this:



Question 6.1:

You became curious about the most expensive activity during the Frost Festival. Write a function that takes the name of the file and returns the most expensive activity listed. If there is a tie, return the last one.

Note:

- `readline()` method might be handy here.
- EXPLICIT FOR LOOPS ALLOWED

```
def most_expensive(file1):
    """
    >>> most_expensive("Files/frost_festival1.csv")
    'Sleigh Ride'

    >>> most_expensive("Files/frost_festival2.csv")
    'Skate with the Elves'

    >>> most_expensive("Files/frost_festival3.csv")
    'Ice Maze'
    """
```

Question 6.2: (optional, extra credit)

On your way to the most expensive activity, you've met a tutor who gave you his list of activities (in exactly the same format as yours). So you decided to combine them into one file and continue the fun for the rest of the night. Write a function that takes two file names and combines them into one. Your function should stack the contents of the second file, `tutor_file`, (without the header) underneath the contents of the first file, `my_file`, so that the values separated by commas correspond to the header.

Notes:

- EXPLICIT FOR LOOPS ALLOWED
- Both files have the same first line, but you need to keep only one.
    - `readline()` method might be handy here.
- You can modify the first file.
    - You can use append, "a", mode to add more content to the file without overwriting old content.
- You can expect at least one activity per file.
- **WARNING**: Be careful, after running the doctest your file1 will be modified, so you need to change its content back to its original state (how it was when you downloaded it) if you want to run the doctest again successfully.

```
def combine(my_file, tutor_file):
    """
    >>> combine("Files/my_list.csv", "Files/tutor_list.csv")
    >>> with open("Files/my_list.csv") as f:
    ...     print(f.read())
    ...
    Event,Time,Price,Wait Time,
    Yeti Hunting,10 AM,$10,0 min,
    Skate with the Elves,12 PM,$10,120 min,
    Photobooth,3 PM,$5,30 min,
    Ice Maze,5 PM,$12,150 min,
    Sleigh Ride,7 PM,$15,30 min,

    >>> combine("Files/my_list2.csv", "Files/tutor_list2.csv")
    >>> with open("Files/my_list2.csv") as f:
    ...     print(f.read())
    ...
    Event,Time,Price,Wait Time,
    Yeti Hunting,11 AM,$10,120 min,
    Skate with the Elves,2 PM,$8,120 min,
```

```
Photobooth,3 PM,$5,30 min,
Ice Maze,4 PM,$17,90 min,
Skate with the Elves,7 PM,$8,120 min,
Photobooth,9 PM,$5,30 min,
"""
```

Submission

When submitting to Gradescope, you can include the Files folder by compressing it with your lab03.py file, creating a .zip file you can submit that zip file to Gradescope. If you have your lab03.py and files folder inside of another folder, make sure that you select lab03.py and the Files folder directly to compress it, rather than compressing the encompassing folder, as it will cause issues with your submission.

You can do this by going into File Explorer or Finder and selecting **both** the Files folder and the lab03.py file. Right-click and then select either

Windows: Send to -> Compressed (zipped) folder

Mac: Compress 2 items

Your .zip file should have the following files if you open it:

Files/

--- frost_festival1.txt

--- frost_festival2.txt

--- ... (other files)

lab03.py

Please submit the homework via Gradescope. You may submit more than once before the deadline, and only the final submission will be graded. Please refer to the Gradescope FAQ if you have any issues with your submission.