Worksheet: Dictionaries

CS 101 Algorithmic Problem Solving

Fall 2023

Name(s): ______________________________

HU ID *(e.g., xy01042)*: ______________________________

1. Item Frequency

You are working as a cashier at a supermarket. You have a list of items sold in a particular day and you want to count the number of times each item was sold.

Write a function *itemFrequency()* that takes a list *items* as argument and prints the number of times each item was sold in the format shown below. You must use dictionaries for this question.

Constraints

- The list *items* contains only strings and it may contain duplicates.

Interaction

The input comprises of a list containing the items purchased in a day.

The output must contain the item name and the number of times each item was purchased separated by a space on separate lines for each item.

Sample

Input	Output
[‘bin’, ‘can’, ‘bin’, ‘bin’]	bin 3
	can 1
[‘banana’, ‘pear’, ‘apple’]	banana 1
	pear 1
	apple 1

In the first case, the *items* list shows that ‘bin’ was purchased 3 times and ‘can’ was purchased once.

In the second case, the *items* list shows that each of ‘banana’, ‘pear’, and ‘apple’ were purchased only once.

Exercise

In the space provided, indicate the outputs for the given inputs.

Input	Output
[‘book’, ‘pen’]	book 1
	pen 1
[‘book’, ‘book’, ‘book’, ‘book’]	book 4
[‘bear plushy’, ‘cat plushy’, ‘fox plushy’, ‘generic plushy’, ‘cat plushy’, ‘fox plushy’, ‘fox plushy’, ‘dog plushy’]	bear plushy 1
	cat plushy 2
	fox plushy 3
	generic plushy 1
	dog plushy 1

Problem Identification
Briefly explain the underlying problem you identified in the above question that led you to your solution.

Input: `items`
Output: Frequency of each item in the list using a dictionary to store counts.

Pseudocode

```
def itemFrequency(items):
    item_counts = {}
    for item in items:
        if item in item_counts:
            item_counts[item] += 1
        else:
            item_counts[item] = 1

    for item, count in item_counts.items():
        print(f"{item} {count}")
itemFrequency(items)
```

Dry Run
Using any one of the inputs provided in the Exercise section above, dry run your pseudocode in the space below.

```
items = [’book’, ’pen’]:

1. item_counts = {}: Initialize an empty dictionary to store item frequencies.

2. Iterating through the items in [‘book’, ‘pen’]:

   a. For the first item, ‘book’:
      - ‘book’ is not in item_counts initially, so it goes to the else block.
      - ‘book’ is added to item_counts with a count of 1.

   b. For the second item, ‘pen’:
      - ‘pen’ is not in item_counts, so it goes to the else block.
      - ‘pen’ is added to item_counts with a count of 1.

3. After the loop, item_counts is now {‘book’: 1, ’pen’: 1}.
```

```
4. Iterating through the items in item_counts.items():

   a. For the first iteration, item=‘book’ and count=1, so it prints "book 1".

   b. For the second iteration, item=‘pen’ and count=1, so it prints "pen 1".

Therefore, the output for the input [‘book’, ‘pen’] will be:


book 1
pen 1
which is the expected output!
```

2. Popular Birth Month

At HU, students of the SDP program have sent out surveys to the entire student body to collect some data for a study. In this survey, students were asked if they were aware of the most popular birth month. CS students, not wanting to seem ignorant of such a basic fact, decided that they will determine the answer by querying their own department. They decided they would write a program that takes in a list of birthdays (where each birthday is stored in a dictionary) and returns the most popular birth month.

Write a function `popularBirthday()` that takes a list of dictionaries `birthdays` as argument and prints the most popular birth month.

Constraints

- The list `birthdays` contains dictionaries. Each dictionary contains 3 keys, ‘year’, ‘month’, and ‘day’ and their associated values will be integers.

Interaction

The input comprises of a list of dictionaries.

The output must be a single line containing the most popular birth month. If more than one month are most popular, then they must be separated by a space.

Sample

Input	Output
[{ ’year’: 1996, ’month’: 12, ’day’: 12}, { ’year’: 1995, ’month’: 12, ’day’: 8}, { ’year’: 1999, ’month’: 4, ’day’: 30}, { ’year’: 1998, ’month’: 7, ’day’: 30}]	December
[{ ’year’: 1996, ’month’: 12, ’day’: 12}, { ’year’: 1995, ’month’: 12, ’day’: 8}, { ’year’: 1999, ’month’: 4, ’day’: 30}, { ’year’: 1998, ’month’: 4, ’day’: 30}]	April, December

In the first case, the `birthdays` shows that the 12th month was the most popular, hence December was displayed.

In the second case, the `birthdays` shows that the 4th and 12th month are the most popular, hence April and December were showed.

Exercise

In the space provided, indicate the outputs for the given inputs.

Input	Output
[{ 'year': 1996, 'month': 11, 'day': 12}, { 'year': 1995, 'month': 12, 'day': 8}, { 'year': 1999, 'month': 4, 'day': 30}, { 'year': 1998, 'month': 7, 'day': 30}]	November, December, April, July
[{ 'year': 1996, 'month': 5, 'day': 12}, { 'year': 1995, 'month': 12, 'day': 8}, { 'year': 1999, 'month': 4, 'day': 30}, { 'year': 1998, 'month': 4, 'day': 30}, { 'year': 1998, 'month': 5, 'day': 30}]	May, April
[{ 'year': 1996, 'month': 12, 'day': 12}]	

Problem Identification

Briefly explain the underlying problem you identified in the above question that led you to your solution.

Input: `birthdays`
Output: Most popular birth month by counting how many times a month occurs in list and storing the counts in a dictionary.

Pseudocode

```
def mostPopularMonth(birthdays):
    month_counts = {}
    month_list = ["January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"]

    for birthday in birthdays:
        month = birthday['month']
        if month in month_counts:
            month_counts[month] += 1
        else:
            month_counts[month] = 1

    max_count = max(month_counts.values())
    most_popular_months = [month for month, count in month_counts.items() if count == max_count]

    result_string = ""
    for month in most_popular_months:
        result_string += month_list[month-1] + ", "

    result_string2=result_string[:-2]
    print(result_string2)
mostPopularMonth(birthdays)
```

Dry Run

Using any one of the inputs provided in the Exercise section above, dry run your psuedocode in the space below.

```
1. month_counts = {}
2. month_list: List of month names.

3. Iterating through the birthdays in the provided input:

   a. For the first birthday ({'year': 1996, 'month': 5, 'day': 12}):
      - 'month' is 5, not in month_counts. Add it with a count of 1.

   b. For the second birthday ({'year': 1995, 'month': 12, 'day': 8}):
      - 'month' is 12, not in month_counts. Add it with a count of 1.

   c. For the third birthday ({'year': 1999, 'month': 4, 'day': 30}):
      - 'month' is 4, not in month_counts. Add it with a count of 1.

   d. For the fourth birthday ({'year': 1998, 'month': 4, 'day': 30}):
      - 'month' is 4. Check if 4 is in month_counts. It is, so increment its count to 2.

   e. For the fifth birthday ({'year': 1998, 'month': 5, 'day': 30}):
      - 'month' is 5. Check if 5 is in month_counts. It is, so increment its count to 2.

4. After the loop, month_counts is now {5: 2, 12: 1, 4: 2}.

5. max_count = max(month_counts.values()): Find the maximum count, which is 2.

6. most_popular_months = [month for month, count in month_counts.items()
if count == max_count]: Find the months with the maximum count, which are 5 and 4.

7. result_string = "" (empty string)

8. Iterate through most popular months:

   a. 1st iter: (month = 5), append "May, " to result_string.

   b. 2nd iter: (month = 4), append "April, " to result_string.

9. result_string2 = result_string[:-2]: Remove the trailing ", " from result_string.

10. print(result_string2): Print the final result, which is "May, April".

So, the output for the given input will be:


May, April
which is the expected output!
```

3. Price Check

At a retail shop with old-style registers, rather than scanning items and pulling the price from the database, the price of each item is typed manually. This method sometimes leads to errors.

Write a function `priceCheck()` that takes four lists as arguments - `products` containing the list of products, `productPrices` containing the prices of the product at the same index in `products`, `productsSold` containing the list of products sold, `soldPrice` containing the price at which the product at the same index in `productsSold` was sold. The function verifies the sold price of each product with the actual price and prints the number of erroneous sales. You must use dictionaries for this question.

Constraints

- The lists `products` and `productsSold` contain only strings and `productsSold` may contain duplicates.
- The lists `productPrices` and `soldPrice` contain only floats.

Interaction

The input comprises of four lists, corresponding to `products`, `productPrices`, `productsSold` , and `soldPrice`.

The output must be a single integer denoting the number of erroneous sales.

Sample

Input	Output
['eggs', 'milk', 'cheese']	2
[2.89, 3.29, 5.79]	
['eggs', 'eggs', 'cheese', 'milk']	
[2.89, 2.99, 5.97, 3.29]	
['eggs', 'milk', 'cheese']	0
[2.89, 3.29, 5.79]	
['eggs', 'eggs', 'cheese', 'milk']	
[2.89, 2.89, 5.79, 3.29]	

In the first case, the second sale of eggs (sold at 2.99, actual price 2.89) and the sale of cheese (sold at 5.97, actual price 5.79) was erroneous.

In the second case, the there were no erroneous sales.

Exercise

In the space provided, indicate the outputs for the given inputs.

Input	Output
[‘eggs’, ‘milk’, ‘cheese’]	2
[2.89, 3.29, 5.79]	
[‘eggs’, ‘cheese’, ‘milk’]	
[5.79, 2.89, 3.29]	
[‘eggs’, ‘milk’, ‘cheese’, ‘butter’]	1
[2.89, 3.29, 5.79, 4.24]	
[‘eggs’, ‘eggs’, ‘cheese’, ‘milk’]	
[2.89, 2.89, 5.79, 4.24]	
['book', 'pen', 'pencil', 'eraser']	0
[2.00, 1.00, 0.5, 0.2]	
['book', 'pen', 'pencil', 'eraser']	
[2.00, 1.00, 0.5, 0.2]	

Problem Identification

Briefly explain the underlying problem you identified in the above question that led you to your solution.

Input:`products`, `productsSold`, `productPrices` and `soldPrice`
Output: A count of how many times sold price of a product does not match its acutal price.

Pseudocode

```
def priceCheck(products,productPrices,productsSold,soldPrice):
    real_prices_dict = {products[i]: productPrices[i] for i in range(len(
                                products))}
    count = 0
    for i in range(len(productsSold)):
        if soldPrice[i] != real_prices_dict[productsSold[i]]:
            count += 1
    return count
print(priceCheck(products,productPrices,productsSold,soldPrice))
```

Dry Run

Using any one of the inputs provided in the Exercise section above, dry run your psuedocode in the space below.

```
    Input:-
    products = [‘eggs’, ‘milk’, ‘cheese’, ‘butter’]
    productPrices = [2.89, 3.29, 5.79, 4.24]
    productsSold = [‘eggs’, ‘eggs’, ‘cheese’, ‘milk’]
    soldPrice = [2.89, 2.89, 5.79, 4.24]

1. real_prices_dict: {‘eggs’: 2.89, ‘milk’: 3.29, ‘cheese’: 5.79, ‘butter’: 4.24}.
2. count = 0:
3. Iterating through the indices of productsSold:
   a. (i = 0):
      - productsSold[0] is ‘eggs’, and soldPrice[0] is 2.89.
      - The real price for ‘eggs’ is real_prices_dict[’eggs’] which is also 2.89.
      They match, so no increment to the count.

   b. (i = 1):
      - productsSold[1] is ‘eggs’, and soldPrice[1] is 2.89.
      - The real price for ‘eggs’ is again real_prices_dict[’eggs’] which is 2.89.
      They match, so no increment to the count.

   c. (i = 2):
      - productsSold[2] is ‘cheese’, and soldPrice[2] is 5.79.
      - The real price for ‘cheese’ is real_prices_dict[’cheese’] which is 5.79.
      They match, so no increment to the count.

   d. (i = 3):
      - productsSold[3] is ‘milk’, and soldPrice[3] is 4.24.
      - The real price for ‘milk’ is real_prices_dict[’milk’] which is 3.29.
      They don’t match, so increment the count to 1.

4. After the loop, count is 1.

5. return count: Return the final count.

So, the output for the given inputs is:
1
which is the expected output!
```

4. Word Morphism

Many words share a common pattern of letter arrangement. For example, both the words `apple` and `gooey` have letters in the pattern 1-2-2-3-4. Carefully mapping each letter in one word (`apple`) to a suitable letter (`a` to `g`, `p` to `o`, `l` to `e`, `e` to `y`) will give us the second word (`gooey`). You are given a mapping and a string, and must morph the string by applying the mapping once. Each mapping is a nested list containing a pair of numbers. Each number is an ordinal for a letter. The first number is the ordinal of the letter to morph from. The second number is the ordinal of the letter to morph to.

Write a function *wordMorph()* that taking in a 2D list *mapping* and a string *s* and returns the string obtained by applying the mapping once on *s*.

Constraints

- The list *mapping* contains lists of length 2. Each of these lists contain integers.
- The string *s* contains only lowercase letters.

Interaction

The input comprises of two lines. The first line is a 2D list denoting *mapping*. The second line is the string *s*.

The output must be a single line containing the string obtained by applying the mapping once on *s*.

Hint The *chr*() function transforms an ordinal number into the corresponding Unicode character, returning it as a string.

Sample

Input	Output
[[97, 103], [112, 111], [108, 101], [101, 121]]	gooey
apple	
[[102, 112], [97, 108], [99, 97], [101, 121], [98, 114], [111, 111], [107, 109]]	playroom
facebook	

In the first case, ‘a’ i.e. ord(97) is mapped to ord(103) i.e. ‘g’; similarly, ‘p’ is mapped to ‘o’, ‘l’ is mapped to ‘e’, and ‘e’ is mapped to ‘y’.

In the second case, ‘f’ is mapped to ‘p’, ‘a’ is mapped to ‘l’, ‘c’ is mapped to ‘a’, ‘e’ is mapped to ‘y’, ‘b’ is mapped to ‘r’, ‘o’ is mapped to ‘o’, ‘k’ is mapped to ‘m’.

Exercise

In the space provided, indicate the outputs for the given inputs.

Input	Output
[[97, 103], [112, 111], [108, 101], [101, 121]]	
eapplee	goeeygg
[[104, 97], [105, 105], [106, 115], [97, 104], [98, 97]]	
hijab	aisha
[[105, 115], [98, 97], [97, 97], [100,100]]	
ibad	saad

Problem Identification
Briefly explain the underlying problem you identified in the above question that led you to your solution.

Input: Two dimensional list denoting `mapping`, and string `s`
Output: String obtained by applying mapping on `s`

Pseudocode

```
def wordMorph(mapping,s):
    string = {}
    count = 0
    for i in range(len(s)):
        if s[i] not in string.keys():
            string[s[i]] = mapping[count][1]
            count+=1

    for i in s:
        print(chr(string[i]), end= '')
    print()

wordMorph(mapping,s)
```

Dry Run
Using any of the inputs provided in the Exercise section above, dry run your psuedocode in the space below.

Input= [104, 97], [105, 105], [106, 115], [97, 104], [98,97]], 'hijab'

1. **Initialization:** $string = \{\}$, $count = 0$.
2. **First Loop** ($i = 0$)**:**
   - Character 'h' is not in *string.keys*(), so add it to *string* with the corresponding value from the first pair in *mapping* (which is 'a').
   - Updated *string*: $\{'h' : 97\}$, *count* increments to 1.
3. **Second Loop** ($i = 1$)**:**
   - Character 'i' is not in *string.keys*(), so add it to *string* with the corresponding value from the second pair in *mapping* (which is 'i').
   - Updated *string*: $\{'h' : 97, 'i' : 105\}$, *count* increments to 2.
4. **Third Loop** ($i = 2$)**:**
   - Character 'j' is not in *string.keys*(), so add it to *string* with the corresponding value from the third pair in *mapping* (which is 's').

  - Updated *string*: $\{'h': 97, 'i': 105, 'j': 115\}$, *count* increments to 3.

5. **Fourth Loop ($i = 3$):**
   - Character 'a' is not in $string.keys()$, so add it to *string* with the corresponding value from the fourth pair in *mapping* (which is 'h').
   - Updated *string*: $\{'h': 97, 'i': 105, 'j': 115, 'a': 104\}$, *count* increments to 4.
6. **Fifth Loop ($i = 4$):**
   - Character 'b' is not in $string.keys()$, so add it to *string* with the corresponding value from the fifth pair in *mapping* (which is 'a').
   - Updated *string*: $\{'h': 97, 'i': 105, 'j': 115, 'a': 104, 'b': 97\}$, *count* increments to 5.
7. **Print Result:**
   - Convert the mapped values in *string* to characters and print the resulting string: $s =$ 'hijab' becomes 'aisha', which is the expected output.

LET'S LEARN TO DEBUG

5. Does a mapping exist?

You've intercepted an enemy message, and you know that it's been encoded using a mapping. You also know that the message is in English and that the mapping is a one-to-one mapping between letters. You want to decode the message, but you don't know the mapping. You decide to write a program that will determine if a mapping exists between two strings.

Write a function *mappingExist()* that takes two strings a and b as argument and returns *True* if a one-to-one mapping exists such that a can be morphed into b, otherwise returns *False*.

Constraints

- The strings a and b contain only lowercase letters.
- $len(a) = len(b)$

Interaction

The input comprises a single line containing 2 space-separated strings denoting a and b.

The output must be a single line containing *True* if a one-to-one mapping exists such that a can be morphed into b, otherwise *False*.

Sample

Input	Output
big tug	True
legends trailer	False

In the first case, mapping 'b' to 't', 'i' to 'u', and 'g' to 'g' serves as a valid mapping for converting a to b.

In the second case, no such mapping exists such that a can be morphed into b ('e' is being morphed into 'r' as well as into 'i', which is not valid for a one-to-one mapping).

Proposed Solution

```
1  def mappingExist(a,b):
2    d = { }
3    for i in range(len(a)):
4      if a[i] in d:
5          if d[a[i]] == b[i]:
6              return True
7      else:
8          d[a[i]] = b[i]
9    return False
```

Dry Run
Using the inputs provided in the Sample section above, dry run the proposed code solution below.

Input = big tug

1. **Initialization:** $d = \{\}$.
2. **First Iteration** ($i = 0$)**:**
   - Character 'b' is not in d, so add it to d with the corresponding value from "tug" (which is 't').
   - Updated d: $\{'b' :' t'\}$.
3. **Second Iteration** ($i = 1$)**:**
   - Character 'i' is not in d, so add it to d with the corresponding value from "tug" (which is 'u').
   - Updated d: $\{'b' :' t',' i' :' u'\}$.
4. **Third Iteration** ($i = 2$)**:**
   - Character 'g' is not in d, so add it to d with the corresponding value from "tug" (which is 'g').
   - Updated d: $\{'b' :' t',' i' :' u',' g' :' g'\}$.
5. **After the Loop:**
   - The loop completes without returning True.
   - Although a one-on-one mapping exists between all characters, the function does not return True because of the conditions in lines 4 and 5. If there are no duplicate letters, the function returns False, which is not our expected output.

Error Identification
Briefly explain the errors you identified in the proposed code solution. Mention the line number and the errors in each line.

- **Lines 5-6:** The code checks if the character mapping exists in the dictionary on line 5. If found, it prematurely returns `True` on line 6.
- **Line 9:** In case of no duplicate characters, all characters are successfully mapped and added to the dictionary. The function then concludes by returning `False` on line 9, instead of returning `True` to indicate that a one-on-one character mapping does exist.

Correct Solution
Rewrite the lines of code you mentioned above with their errors corrected.

```
def mappingExist(a,b):
    d = {}
    for i in range(len(a)):
        if a[i] in d:
            if d[a[i]] != b[i]:
                return False
        else:
            d[a[i]] = b[i]
    return True
print(mappingExist(a,b))
```

Rough Work