Lab Exercise 3

Learning Outcomes Covered

- Usage of Python data structures such as lists and dictionaries.
- Basic object-oriented programming.

Instructions

- Unless specified otherwise,
  - you are allowed to use anything in the Python Standard Library;
  - you are NOT allowed to use any third-party library in your code;
  - you can assume all user inputs are always valid and your program for each question need not include code to validate them.
- The underlined blue texts in the Sample Runs section of some questions refer to the user inputs. It does NOT mean that your program needs to underline them.
- **Please name each of your Python script files exactly as the name in parenthesis as shown in each question heading. Marks will be deducted if violating this instruction.**

Part I: In-Class Exercise (Deadline: Feb 10th, 11:59pm)

Question 1: Stocktaking (q1.py) (20%)

Suppose that you are running an online business and a big part of your day is fulfilling orders. As the volume of orders keeps growing, you have decided to develop an inventory database and a Python program to do stocktakes more easily.

Write a function `fillable()` that takes three arguments: a dictionary `stock` representing all the merchandise you have in stock, a string *merch* representing the product your customer wants to buy, and an integer *n* representing the number of units of the product they would like to buy. The function should return `True` if you have enough stock of the merchandise to complete the sale; otherwise, it should return `False`. Assume that input arguments are always valid and n is always >= 1.

```
def fillable(stock, merch, n):
    # Your code goes here.
    return ...
```

Sample Usage:

```
stock = {
  'football': 4,
  'boardgame': 10,
  'Lego': 1,
  'doll': 5,
}
print(fillable(stock, 'football', 3))
print(fillable(stock, 'Lego', 2))
```

```
print(fillable(stock, 'action figure', 1))
```

The above code prints the following output:

```
True
False
False
```

Part II : Take-home Exercise (Deadline: 11:59pm Feb 21)

Question 2: Who is the killer? (q2.py) (20%)

Some people have been murdered on a day! You have managed to narrow the suspects down to a few. Luckily, you know every person who those suspects have seen on the day of the murders.

Write a function `find_killer(suspects_dict, victims)` as

```
def find_killer(suspects_dict, victims):
    # your code that returns the killer
```

Given a dictionary with all the names of the suspects and everyone that they have seen on that day which may look like this:

```
{
  'Jackson': ['Jacob', 'Bill', 'Lucas'],
  'Johnny': ['David', 'Kyle', 'Lucas'],
  'Peter': ['Lucy', 'Kyle']
}
```

and also a list of the names of the dead people:

```
['Lucas', 'Bill']
```

Return the name of the killer (exactly one), in this case 'Jackson', because he is the only person who saw both 'Lucas' and 'Bill'. Assume that there is only one killer in the input, so your code need not handle the cases having no killer or more than one killer.

Question 3: Did you mean ...? (q3.py) (20%)

We saw Google's "Did you mean ...?" when we searched a term but misspelled it. In this question, you are to implement a word similarity search.

Given a single word, probably misspelled, and a list of correctly spelled words. The former is the *search key* and the latter is the *dictionary* (don't confuse it with Python's dictionary). Assume that all words are lowercase strings. Your task is to find which word (*d*) from the dictionary is the most similar

to the search key (*s*). The similarity is measured by the number of different letters between *d* and *s*. The fewer the number of different letters, the higher the similarity. For example, difference between "*appl*" and "*apple*" is 1 while difference between "*notify*" and "*note*" is 3 (one difference due to i versus e, and two differences due to the additional letters "fy"). As another example, the misspelled word "*berr*" is more similar to *beer* (1-letter difference) than to *barrel* (3-letter difference).

Difference Explanation:

search_key	**word**	**difference** (= word_diff + length_diff)	word_diff	length_diff
strawbery	strawberry	2	1 (y → r)	1 (extra "y")
code	codewars	4	0	4 (extra "wars")
heaven	python	5	5	0
satisfied	satisfy	3	1 (i → y)	2 (extra "ed")

Assume that there will be no empty string in the input. And for each query, there will be **exactly one** correct (most similar) word to be found. That means, you **don't** need to handle the cases where search_key is similar to 0 or >=2 words in the dictionary.)

Implement a class `Dictionary` that holds a list of correctly spelled words and returns from the list the word which is the most similar to a given search key.

```
class Dictionary:
    def __init__(self, words):
        self.words = words

    def find_most_similar(self, search_key):
        # your code here
        return ...
```

Sample Usage:

```
fruits = Dictionary(
    ['cherry', 'pineapple', 'melon', 'strawberry', 'raspberry'])
print(fruits.find_most_similar('strawbery'))  # prints "strawberry"
print(fruits.find_most_similar('bherry'))     # prints "cherry"

things = Dictionary(['stars', 'mars', 'wars', 'codec', 'codewars'])
print(things.find_most_similar('coddwars'))   # prints "codewars"

languages = Dictionary(['javascript', 'java', 'ruby',
                        'php', 'python', 'coffeescript'])
print(languages.find_most_similar('heaven'))     # prints "python" (last letter matched)
print(languages.find_most_similar('javascript')) # prints "javascript" (same words are
```

```
                                # obviously the most similar ones)
```

The above code prints the following output:

```
strawberry
cherry
codewars
python
javascript
```

Question 4: Quarks (q4.py) (20%)

Quarks are the only elementary particles in the Standard Model of particle physics to experience all four fundamental forces. Suppose that you are modelling the interaction between a large number of quarks, write a class called `Quark` by which you can generate your own quark objects.

Your `Quark` class should allow you to create quarks of any valid *color* ("red", "blue", and "green") and any valid *flavor* ('up', 'down', 'strange', 'charm', 'top', and 'bottom').

Every quark has the same *baryon number:* 1/3. There is an *interact*() method that allows any quark to interact with another quark via the strong force. When two quarks interact, they exchange their color.

```
class Quark:
    # Your code here.
```

Sample Usage:

```
q1 = Quark("red", "up")
q2 = Quark("blue", "strange")

print("Object initialization")
print(q1.color == "red")              # prints True
print(q2.flavor == "strange")         # prints True

print("Class attributes")
print(abs(q2.baryon_number -1.0 / 3) <= 1e-5)          # prints True

print("Quarks exchange color when interacting")
```

```
q1.interact(q2)
print(q1.color == "blue")     # prints True
print(q2.color == "red")      # prints True
```

Note: in the above code, `q2.baryon_number` is a floating-point value. We cannot check equality of two floating-point values by the == operator. Such equality checks need to be done by checking if their difference is bounded by some very tiny value such as 1e-5.

Question 5: Pagination (q5.py) (20%)

In this question, you are to complete a class, called `PaginationHelper`, which is a utility class helpful for querying paging information about an array (or a Python list).

The class is designed to take in an array of values and an integer indicating how many items will be allowed per each page. The types of values contained within the collection/array are not relevant.

You should complete the PaginationHelper class as follows. The explanation of each method can be found in the comment around it.

```
class PaginationHelper:

 # The constructor takes in an array of items and an integer indicating
 # how many items fit within a single page
 def __init__(self, collection, items_per_page):

 # returns the number of items within the entire collection
 def item_count(self):

 # returns the number of pages
 def page_count(self):

 # returns the number of items on the current page. page_index is zero-based
 # this method should return -1 for page_index values that are out of range
 def page_item_count(self, page_index):

 # determines which page an item is on (zero-based indexes)
 # this method should return -1 for item_index values that are out of range
 def page_index(self, item_index):
```

Sample Usage:

The following are some examples of how this class is used:

```
helper = PaginationHelper(['a','b','c','d','e','f'], 4)
print(helper.page_count())          # prints 2
print(helper.item_count())          # prints 6
print(helper.page_item_count(0))    # prints 4
print(helper.page_item_count(1))    # last page, prints 2
print(helper.page_item_count(2))    # prints -1 since the page is invalid

# page_index() takes an item index and returns the page that it belongs to
print(helper.page_index(5))     # prints 1 (zero based index)
print(helper.page_index(2))     # prints 0
print(helper.page_index(20))    # prints -1
print(helper.page_index(-10))   # prints -1 because negative indexes are invalid
```