ECE 364 Prelab Assignment 01
Basics of Python

Passing this lab will satisfy course objectives CO3

Instructions

- You must meet all *base requirements* in the syllabus to receive any credit.
- Work in your Prelab01 directory, and copy all files from ~ee364/DataFolder/Prelab01 into your Prelab01 directory:

  ```
  cp -r ~ee364/DataFolder/Prelab01/* ./
  ```

- Remember to add and commit all **requird** files to SVN. **We will grade the version of the file that is in SVN!**
- Do *not* add any file that is *not* required. **You will lose points if your repository contains more files than required!**
- Make sure you file compiles. **You will not receive any credit if your file does not compile.**
- Name and spell the file, and the functions, exactly as instructed. Your scripts will be graded by an automated process. **You will lose some points, per our discretion, for any function that does not match the expected name.**
- Make sure your output from all functions match the given examples. **You will not receive points for a question whose output mismatches the expected result.**
- Unless otherwise specified, you cannot use any external library, but you can use any module in the **Python Standard Library** to solve this lab, i.e. anything under:

  ```
  https://docs.python.org/3.7/library/index.html
  ```

- Make sure you are using Python 3.7 for your Prelab.

Basics of Python

Required File `simpleTasks.py`

Create a Python file named `simpleTasks.py`, and do all of your work in that file. This is the only file you need to submit. Refer to the **Python File Structure** section below for a reminder on the requirements.

Implementation Requirements

1. **[20 pts]** Consider the string `"1547896154321687984"` containing a sequence of digits. If you search for `"154"` you can find that the string contains two instances of that number. If you search for the pattern `"1XX7"`, where `"X"` is a placeholder representing any digit, you can find the two numbers `"1547"` and `"1687"` that match that pattern, and searching for the pattern `"X8XX8X"` will result in only a single match, `"687984"`.

   Write a function called `find(pattern)` that reads the file `sequence.txt` given to you, searches for the provided input pattern string throughout, and returns a list of all number sequences that match the given pattern.

   Notes:

   - The pattern can be of arbitrary size, but it can only contain digits and placeholders.
   - There can be an arbitrary number of the placeholders anywhere in the pattern string.
   - The pattern provided will have at least one digit and one placeholder.
   - If the pattern results in a single match, return a list of one string element, and if it does not result in any matches, return an empty list.
   - The returned results must be in the order of their presence in the file, and can contain overlapping digits. For example, search for the pattern `"X38X"` in the sequence `"138389"` should return `"1383"` and `"8389"`.
   - Do *not* use Regular Expressions to solve this question, or you will receive no points.

   Hints:

   - You can solve this question anyway you want, but a *suggested* procedure is as follows:
     - Load the data sequence first into a variable.
     - Create a list of all testable subsequences.
     - Test each subsequence for condition matching.

2. **[20 pts]** Consider the string `"54789654321687984"` containing a sequence of digits. The sequence `"547"` is the only one whose digital product (the product of its individual digits) is equal to 140, while the sequences `"32168"` and `"984"` have a digital product of 288.

   Write a function called `getStreakProduct(sequence, maxSize, product)` that searches the string `sequence` for all sub-sequences, whose size is between 2 and `maxSize`, inclusively, and returns a list of all sequences whose digital product is equal to `product`.

Notes:

- If there is only one match, return a list containing one element, and if there are no matches, return an empty list.
- The returned results must be in the order of their presence in the input sequence, and can contain overlapping digits. For example, searching for sequences of max size 3, with digital product equals to 32 in the sequence `"14822"` should return `"148"`, `"48"` and `"822"`.

3. **[15 pts]** Write a function called `writePyramids(filePath, baseSize, count, char)` that saves one or more pyramid-shaped sequence of characters in file, separated by a single space at the base. The `filePath` is the full path of the file to save the pyramids to, `baseSize` is an odd integer representing the size of the pyramid's base, `count` is the number of horizontally-concatenated pyramids to create, and `char` is the character used to build the pyramids with. For example, the files `pyramid13.txt` and `pyramid15.txt` have been obtained using the commands:

```
>>> writePyramids('pyramid13.txt', 13, 6, 'X')
>>> writePyramids('pyramid15.txt', 15, 5, '*')
```

Note: The files you generate must be an *exact* match to the ones provided, if you use the same arguments.

Hints:

- You must use a "Diff" tool to compare your result with the given samples. Visual eyeballing comparison is *not* sufficient, and can result in zero credit for missing one or more spaces. Note that PyCharm includes file comparison capability.
- You can solve this question anyway you want, but a *suggested* procedure is as follows:
  - Generate line n for a single pyramid.
  - Generate a single pyramid, then use it to generate the rest.
  - Combine the pyramids in a list.
  - Write the list to the target file.

4. **[10 pts]** Consider the string `"AAASSSSSSAPPPSSPPBBCCCSSS"` containing a sequence of uppercase letters. Write a function called `getStreaks(sequence, letters)` that takes in a string similar to the example shown, and returns a list of the streaks, in the order of their appearance in the sequence, of the letters provided. For example:

```
>>> sequence = "AAASSSSSSAPPPSSPPBBCCCSSS"
>>> getStreaks(sequence, "SAQT")
['AAA', 'SSSSSS', 'A', 'SS', 'SSS']
>>> getStreaks(sequence, "PAZ")
['AAA', 'A', 'PPP', 'PP']
```

Notes:

- Return an empty list if no matches were found of any of the letters passed.
- As shown in the example, the order of letters in the input argument `letters` is arbitrary, and not all letters may be present.

5. **[5 pts]** Write a function called `findNames(nameList, part, name)` that takes in a list of strings `nameList`, where each string contains the first and last name of a person. The function should search for the `name` passed, and return a list of matches, based on the constraint string `part`, where `part` can be `"F"`, for searching in first names only, `"L"` for searching in last names only, or `"FL"` for searching in both. For example:

```
>>> names = ["George Smith", "Mark Johnson", "Cordell Theodore", "Maria Satterfield",
             "Johnson Cadence"]
>>> findNames(names, "L", "johnson")
["Mark Johnson"]
>>> findNames(names, "F", "JOHNSON")
["Johnson Cadence"]
>>> findNames(names, "FL", "Johnson")
["Mark Johnson", "Johnson Cadence"]
```

Notes:

- Return an empty list if no matches were found, or if the `part` string contains anything other than the options mentioned above. You may assume you will always receive uppercase letters.
- As shown in the example, the name to search for can have different letter casing from the ones in list.

6. **[5 pts]** The number 9 can be represented in binary as `1001`, and if we convert it to a list of Booleans we will have `[True, False, False, True]`. Write a function called `convertToBoolean(num, size)` that takes in a positive integer and returns a list of Booleans representing the number `num`, where the length of the list is a *minimum* of `size`. For example:

```
>>> convertToBoolean(135, 12)
[False, False, False, False, True, False, False, False, False, True, True, True]

>>> convertToBoolean(9, 3)
[True, False, False, True]
```

Notes:

- If the requested size is not sufficient, you need to expand the list to satisfy the needs of the input number as shown in the second example.
- Verify that the input parameters are integers, and return an empty list if they are not.

7. **[5 pts]** Write a function called `convertToInteger(boolList)` that does the opposite of the previous function, i.e. it takes in a list of Booleans, and returns their equivalent integer. For example:

```
>>> bList = [True, False, False, False, False, True, True, True]
>>> convertToInteger(bList)
135

>>> bList = [False, False, True, False, False, True]
>>> convertToInteger(bList)
9
```

Notes:

- Verify that the input parameter is list, and return `None` if it is not.
- Verify that all elements are Booleans, and return `None` otherwise.
- Verify that the list is *not* empty, and return `None` if it is.

Final Check!

You are given a file that checks for the exact spelling of the required functions. Make sure you run this file before your final commit. Remember that **you will lose some points for any function that does not match the expected name.**

Python File Structure

The following is the expected file structure that you need to conform to:

```
#########################################################
#   Author:     <Your Full Name>
#   email:      <Your Email>
#   ID:         <Your course ID, e.g. ee364j20>
#   Date:       <Start Date>
#########################################################

import os     # List of module import statements
import sys    # Each one on a line


#########################################################
# No Module-Level Variables or Statements!
# ONLY FUNCTIONS BEYOND THIS POINT!
#########################################################

def functionName1(a: float, b: float) -> float:
    return 0.


def functionName2(c: str, d: str) -> int:
    return 1


# This block is optional and can be used for testing.
# We will NOT look into its content.
#########################################################
if __name__ == "__main__":
    # Write anything here to test your code.
    z = functionName1(1, 2)
    print(z)
```