ECE 364 Lab 3 Handout

Basics of Python

September 4, 2017

Passing this lab will satisfy course objectives CO3

Instructions

- Work in your Lab03 directory.
- Copy all files from ∼ee364/labfiles/Lab03 into your Lab03 directory. You may use the following command:
  `cp -r ∼ee364/labfiles/Lab03/* ./`
- Remember to add and commit all files to SVN. **We will grade the version of the file that is in SVN!**
- 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. Also, 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.**
- Make sure you are using Python 3.4 for your project. In PyCharm, go to:

  File Menu → Settings → Project Interpreter

  And make sure that Python 3.4 (`/usr/local/bin/python3.4`) is selected.

Basics of Python

Create a Python file named `simpleTasks.py`, and do all of your work in that file. This file should only consists of function blocks, and, optionally, a conditional main block, (i.e. `if __name__ == "__main__":`). Do not include any loose Python statements, but you can, however, write any number of additional utility/helper functions that you might need.

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.

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, '*')
```

Notes:

- The files you generate must be an exact match to the ones provided, if you use the same arguments.
- The first line of the file always contain the tips of the pyramids, where each tip is only one character.
- The last line of the file always contain the bases of the pyramids, separated by a space.

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.