ECE 364 Prelab Assignment 06

Regular Expressions

Passing this lab will satisfy course objectives CO2, CO3, CO5

In order to pass CO5, you must use regular expressions **searching** or **matching** to solve this lab. Using any other method, like string functions, will *not* earn you any credit, nor will it satisfy the course objective. Please use the file checker provided to identify whether you violated this requirement or not. If you are in doubt, check with your TAs.

You can *only* use the following functions from the `re` module to perform, or help with, the text extraction: `search()`, `match()`, `compile()`, `fullmatch()`, `findall()`, `finditer()`, `escape()` and `purge()`.

Instructions

- You must meet all *base requirements* in the syllabus to receive any credit.
- Work in your Prelab06 directory, and copy all files from ~ee364/DataFolder/Prelab06 into your working directory:

  ```
  cp -r ~ee364/DataFolder/Prelab06/* ./
  ```

- Remember to add and commit all **required** 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.

Regular Expressions

Required File `regexTasks.py`

Description

Create a Python file named `regexTasks.py`, and do all of your work in that file. This is the only file you need to submit. You can write any number of additional helper functions in this file, but **DO NOT CREATE ANY MODULE VARIABLES**. Refer to the **Python File Structure** section below for a reminder on the requirements. Note that the use of type annotation is *not* required, but it is recommended for better compile-time static analysis.

Part I

- **[5 pts]** One possible format of a URL is:

  ```
  http://[BaseAddress]/[Controller]/[Action]?[QueryString]
  ```

  where `[QueryString]` contains a list of `field=value` elements, separated by the ampersand (&) symbol. For example:

  ```
  http://www.purdue.edu/Home/Calendar?Year=2016&Month=September&Semester=Fall
  ```

  Note that all elements of the URL can only contain alphanumeric characters, the underscore (_), the dash (-) or the dot (.). Write a function called `getUrlParts(url)` that takes in a URL of the above format, and returns a `(string, string, string)` tuple, where the elements of the tuple are the base address, the controller and the action. For example:

  ```
  >>> url = "http://www.purdue.edu/Home/Calendar?Year=2016&Month=September&Semester=Fall"
  >>> getUrlParts(url)
  ("www.purdue.edu", "Home", "Calendar")
  ```

- **[5 pts]** Following the information in the previous question, write a function called `getQueryParameters(url)` that takes in a URL of the above format, and returns a list of `(string, string)` tuples, where the elements of the tuple are the field and the value of each query element. For example:

  ```
  >>> url = "http://www.google.com/Math/Const?Pi=3.14&Max_Int=65536&What_Else=Not-Here"
  >>> getQueryParameters(url)
  [("Pi", "3.14"), ("Max_Int" ,"65536"), ("What_Else", "Not-Here")]
  ```

- **[5 pts]** Write a function called `getSpecial(sentence, letter)` that takes in a sentence and a single letter, and returns a list of the words that start or end with this letter, but not both, regardless of the letter case. For example:

  ```
  >>> s = "The TART program runs on Tuesdays and Thursdays, but it does not start until
  next week."
  >>> getSpecial(s, "t")
  ["The", "Tuesdays", "Thursdays", "but", "it", "not", "start", "next"]
  ```

- **[10 pts]** Write a function called `getRealMAC(sentence)` that takes in a string and searches for the presence of a MAC address anywhere in the sentence. If found, return it as a string. Otherwise, return `None`. An example of a MAC address is `58:1C:0A:6E:39:4D`.

Notes:

- The MAC address is comprised of six parts, each part consists of two hexadecimal digits.
- The parts are separated by a colon, ':', as shown above, or by a dash, '-' as in `58-1C-0A-6E-39-4D`.
- The letters in the hexadecimal digits can be present in uppercase or lowercase form.
- The MAC address can be present anywhere in the sentence.

Part II

You are given a data file called `` `Employees.dat' `` containing employee information for some company, where each line represents an employee entry. The information required for every employee are: "name", "ID", "phone number" and "State," which is the US State name in which the employee resides. Unfortunately, the information in the file is not properly maintained, and hence not all the required employee information is present in the file. Moreover, for the same piece of information, the format is not consistent. The description of the data contained in the file is as follows:

- The employee name is always present, but it is sometimes formatted as `"<First> <Last>"`, and other times as `"<Last>, <First>"`. Both the first and last names only contain uppercase and lowercase letters.
- The employee ID is a UUID[1], a 32 hexadecimal character string. The canonical form of a UUID is formatted as 8-4-4-4-12, e.g. `4f43b41b-c200-4c09-9d45-8892c5b70cea`. However, the ID in the file can be present in lowercase or uppercase form. It also may or may not be hyphenated, and it may be surrounded by curly brackets. Here are some examples of how the given ID might be present:

  ```
  4f43b41b-c200-4c09-9d45-8892c5b70cea
  {4f43b41b-c200-4c09-9d45-8892c5b70cea}
  4F43B41B-C200-4C09-9D45-8892C5B70CEA
  4F43B41BC2004C099D458892C5B70CEA
  {4F43B41B-C200-4C09-9D45-8892C5B70CEA}
  ```

  Finally, note that the ID may be missing from the employee entry.
- The employee phone number can be formatted as one of the following: `"(XXX) XXX-XXXX"`, `"XXX-XXX-XXXX"` or `"XXXXXXXXXX"`. The phone number may be missing from the entry.
- The State contains only letters, but it can be one word, like "Indiana" or two words, like "New York." The State name may be missing from the entry.
- For every entry, the fields are separated by a varying number of semicolons (;), commas (,) and spaces.
- The order of the fields is always the same: `name, ID, phone, state,` i.e. if the ID and the State are present, the ID always shows up before the State.
- If an entry has all of the fields, i.e. `name, ID, phone, state,` present, this entry is considered complete.
- If an entry has only the employee name present, but no other fields, that entry is rejected.
- If an entry has the employee name and one or two additional fields, that entry is considered partially complete.

Your task is to create a set of functions to parse the file and return the required data in a normalized format.

[1]UUID stands for **"Universally Unique Identifier"**. This is also sometimes referred to as a GUID, which stands for **"Globally Unique Identifier"**.

Requirements

- In all of the functions below, when returning an employee name, it must be formatted as `"<First> <Last>"`.
- If any function returns a phone number, it must be formatted as `"(XXX) XXX-XXXX"`.
- If any function returns an ID, it must be formatted in the canonical form of 8-4-4-4-12, where all the characters are in lowercase. This can be achieved by using the function `UUID()` from the `uuid` module. You will need to use the string representation of the UUID, by using the `str()` to get the desired canonical form. For example:

```
>>> from uuid import UUID
>>> i = "{4F43B41BC2004C099D458892C5B70CEA}"
>>> str(UUID(i))
"4f43b41b-c200-4c09-9d45-8892c5b70cea"
```

- [**10 pts**] Write a function called `getRejectedEntries()` that returns a sorted `list` of the rejected employee names, i.e. the employees whose names are present, but with no other information.
- [**10 pts**] Write a function called `getEmployeesWithIDs()` that returns a `{string: string}` dictionary, of all employees with IDs, where the key is the employee name, and the value is the ID. The dictionary must contain the employees whose IDs are present in the file, regardless whether any other information is present or not.
- [**5 pts**] Write a function called `getEmployeesWithoutIDs()` that returns a sorted `list` of employee names of the "non-rejected" employees whose IDs are *not* present. (Note that these employees must have phone and/or State information.)
- [**10 pts**] Write a function called `getEmployeesWithPhones()` that returns a `{string: string}` dictionary, of all employees with phone numbers, where the key is the employee name, and the value is the phone number. The dictionary must contain the employees whose phone numbers are present in the file, regardless whether any other information is present or not.
- [**10 pts**] Write a function called `getEmployeesWithStates()` that returns a `{string: string}` dictionary, of all employees whose State of residence is present, where the key is the employee name, and the value is the State name. The dictionary must contain the employees whose State of residence is present in the file, regardless whether any other information is present or not.
- [**10 pts**] Write a function called `getCompleteEntries()` that returns a dictionary, where the key is the employee name, and the value is the three-element tuples `(string, string, string)`, where the first element is the ID, the second is the phone number, the third is the State of residence. This dictionary must contain the employees whose information is complete in the file, i.e. their IDs, phone numbers and States are all present.

Final Check!

You are given a file that checks for the exact spelling of the required functions, along with the potential use of string functions. Make sure you run this file **before** your final commit. Remember that:

- **You will receive zero if you do not check your file for string functions, and end up using one or more in your submission.**
- **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)
```