When working on this quiz, recall the rules stated on the Academic Integrity Contract that you signed. You can download the **q2helper** project folder (available for Friday, on the **Weekly Schedule** link) in which to write your Regular Expressions and write/test/debug your code. Submit your completed files for **repattern1a.txt**, **repattern1b.txt**, **repattern1c.txt**, **repattern2.txt**, and your **q2solution.py** module online by **Friday**, 11:30pm. I will post my solutions to EEE reachable via the **Solutions** link on **Saturday** morning.

For parts 1a, 1b, 1c, and 2, use a text editor (I suggest using Eclipse's) to write and submit a **one line** file. The line should start with the **^** character and end (on the same line) with the **$** character. The contents of that **one line** should be exactly what you typed-in/tested in the online Regular Expression checker.

1a. (2 pts) Write a **regular expression** pattern that matches all numbers from 1-31 (interesting because it represents all the possible days in a month). It would be trivial to write a huge choice with 31 different numbers: **^(1|2|3|...|20|30|31)$** requiring 87 characters: 2 anchors (**^** and **$**), 2 parentheses, 30 **|** characters, 9 characters for single digit numbers, 20 characters for numbers 10-19, 20 characters for numbers 20-29, and 4 characters for 30 and 31. But you are required to write a shorter RE: look for commonalties. My solution is 22 characters (and I found many slightly longer variants). Put your answer in **repattern1a.txt**.

1b. (3 pts) Write a **regular expression** pattern that matches dates according to the following rules: dates are written in three parts, in the order **month**, **day**, and optional **year**, where the parts are separated by the **/** character. The **month** is a one or two digit number in the range **1-12** (with no leading **0** allowed). The **day** is a one or two digit number in the range **1-31** (with a leading **0** allowed for one digit numbers). The **year**, if included, is a two digit number (each digit can be **0**: generally, **XX** means **20XX**) or a four digit number between 1900-2099. Put your answer in **repattern1b.txt**. Here are a few legal/illegal examples. Note there is no requirement that the month and day make sense: **2/30** and **9/31** are both syntactically legal, although neither makes semantic sense.

Legal: Should Match : **2/10**, **2/10/06**, **2/10/1906**, **12/31/2015**, **12/3**, **12/03**, **2/31**, **9/4/13**
Illegal: Should Not Match: **02/10**, **13/10**, **21/13**, **12/ 13**, **5**, **5/**, **5//**

Put your answer in **repattern1b.txt**.

1c. (2 pts) Write a **regular expression** pattern that matches the same strings described in part 1b. But in addition for this pattern , ensure group **1** is the **month**; group **2** is the **day**; group **3** is the **year**. For example, if we execute **m = re.match(the-pattern, '10/13/2017')** then **m.groups()** returns **('10', '13', '2017')**. Likewise, if we execute **m = re.match(the-pattern, '10/13')** then **m.groups()** returns **('10', '13', None)**: the **None** is because the optional year part of the pattern is missing: the online tool omits showing such a group. There should be no other numbered groups. Hint **(?:...)** creates a parenthesized **regular expression** that is not numbered as a group. You can write one regular expression for both 1b and 1c, or you can write a simpler one for 1b (ignore groups) and then update it for 1c by specifying the necessary groups. Put your answer in **repattern1c.txt**.

2. (5 pts) Write a **regular expression** pattern that matches strings representing trains. A single letter stands for each kind of car in a train: **E**ngine, **C**aboose, **B**oxcar, **P**assenger car, and **D**ining car. There are four rules specifying how to form trains legally.

1. One or more **E**ngines appear only at the front; one **C**aboose only at the end. All other cars are between these.
2. **B**oxcars always come in pairs: **BB**, **BBBB**, etc.
3. There cannot be more than four **P**assenger cars in a series.
4. One **D**ining car must follow each series of **P**assenger cars (and occur no where else).

These cars cannot appear anywhere other than these locations. Here are some legal and illegal exemplars. See more legal/illegal exemplars in **bm2.txt**.

EC	Legal: the smallest train
EEEPPDBBPDBBBBC	Legal : simple train showing all kinds of cars
EEBB	Illegal: no caboose (everything else OK)
EBBBC	Illegal: three (odd number of) boxcars in a row

EEPPPPPDBBC	Illegal: more than four passenger cars in a row
EEPPBBC	Illegal: no dining car after passenger cars
EEBBDC	Illegal: dining car after box car

Put your answer in **`repattern2.txt`**.

3. (7 pts) EBNF allows us to name rules and then build complex descriptions whose right-hand sides use these names. But Regular Expression (RE) patterns are not named, so they cannot contain the names of other patterns. It would be useful to have named REs and use their names in other REs. In this problem, we will represent named RE patterns by using a **`dict`** (whose **keys** are the names and whose **associated values** are RE patterns that can contain names), and then repeatedly replace the names by their RE patterns, to produce complicated RE patterns that contains no names.

Define a function named **`expand_re`** that takes one **`dict`** as an argument, representing various names and their associated RE patterns; **`expand_re`** returns **`None`**, but mutates the **`dict`** by repeatedly replacing each name by its pattern, in all the other patterns. The names in patterns will always appear between **`#`**s. For example, if **`p`** is the **`dict {'digit': r'[0-9]', 'integer': r'[+-]?#digit##digit#*'}`** then after calling **`expand_re(p)`**, **`p`** is now the **`dict {'integer': '[+-]?(?:[0-9])(?:[0-9])*', 'digit': '[0-9]'}`**. Notice that **`digit`** remains the same, but each **`#digit#`** in **`integer`** has been replaced by its associated pattern and put **inside a pair of parentheses prefaced by ?:**. Hint: For every rule in the dictionary, substitute (see the **`sub`** function in **`re`**) all occurrences of its **key** (as a pattern, in the form **#key#**) by its associated value (always putting the value inside parentheses), in every rule in the dictionary. The order in which names are replaced by patterns is not important. Hint: I used **`re.compile`** for the **#key#** pattern (here no **`^`** or **`$`** anchors!), and my function was 4 lines long (this number is not a requirement).

The **`q2solution.py`** module contains the example above and two more complicated ones (and in comments, the **`dict`**s that result when all the RE patterns are substituted for their names). These examples are tested in the **`bscq2S21.txt`** file as well.

4. (6 pts) Write a function named **`multi_search`**, that takes two open files as arguments and returns a **`list`** of 3-**`tuples`** as a result (based on the information in both files). The **`multi_search`** function reads the first file, interpreting each line in it as a regular expression pattern. Then it reads the second file, and for each line in the file, it produces a 3-**`tuple`** in the returned **`list`**, but only if the line matches (using **`re.search`**: meaning the match is not required to begin at the start of the line) one or more of the regular expression patterns read from the first file. Each 3-**`tuple`** contains the line number of the text matched in the second file (starting at line 1), the line of text itself from the second file, and a **`list`** of all the pattern line numbers (starting at line 1) from the first file that matched it: the **`list`** should show these numbers in ascending order (which can be done without sorting). Close both files right before the function finishes.

For example if the files **`pats1.txt`** and **`texts1.txt`** store the information shown at the bottom of this page, then calling the function as **`multi_search(open(pats1.txt'),open(texts1.txt'))`** returns the following list: `[(1, 'See Spot.', [1, 2]), (2, 'See Snoopy', [1]), (3, '  run.', [2]), (5, 'Run dogs run.', [1, 2, 3]), (6, 'Dogs are great.', [1, 2, 3])]`.

Hint: Call **`rstrip`** for each line read from each file; I used comprehensions, calls to **`enumerate`**, and the **`re.compile`** function (for efficiency, since the same patterns never change).

pats1.txt

```
^[A-Z]
\.$
[a-z]\w*(es|s)\W
```

texts1.txt

```
See Spot.
See Snoopy
  run.
(pause)
Run dogs run.
Dogs are great.
```