CSC/CPE 202: Spring 2020

LAB-2 (Due 4/20)

Note: Always your Name and Section must be as header of your files.
If you are collaborating with anyone or any sources, please write the name of your collaborator(s) in the header of the file.

Part 1:

Review the code in location.py. Note that there is a class definition for a Location class, and an associated __init__ method. In addition, there is code to create Location objects and print information associated with those objects.

```
class Location:
    def __init__(self, name, lat, lon):
        self.name = name    # string for name of location
        self.lat = lat      # latitude in degrees (-90 to 90)
        self.lon = lon      # longitude in degrees (-180 to 180)
```

1. Write the data definition of Design Recipe for given class.
2. Without modifying the code, run location.py in whatever environment you wish (again, reference the Getting Started document if you need help in doing this)
3. Note the information that is printed out for each Location object you should see something like this:
4. Location 1: <__main__.Location object at 0x000001F6A2E0C7B8>
5. Since we have provided any specific method to provide a representation for the class, Python uses a default method. What do you notice about the information for loc1 and loc4?
6. Also note the result of the equal comparisons between the locations, in particular loc1 == loc3 and loc1 == loc4. Make sure you understand why the results are what they are.
7. Now modify the location.py code, adding in the methods (__eq__() and __repr__()). See the location_tests.py to figure out what the repr method should look like.
8. Run the location.py code with the modifications made above.
9. Now review the information printed out for each location. The __repr__ method of Location is now being used when printing the object.
10. Examine the results of the equal comparisons. How are they different from before the __eq__ method is added?
11. Why updating the latitude of loc4 effects the latitude of loc1?

Part 2: Generating permutations in lexicographic order

Goal: Write a Python program to generate all the permutations of the characters in a string. This will give you a chance to review some simple Python constructs, i.e.. Strings and Lists and solidify your understanding of recursion.

Your program **must** meet the following specification. You are to write a Python function **perm_lex** that:

- Takes a string as a single input argument. You may assume the string consists of **distinct lower case** letters (in alphabetical order). You may assume the input is a string of letters in alphabetical order.
- Returns is a **list** of strings where each string represents a permutation of the input string. The list of permutations must be in lexicographic order. (This is basically the ordering that dictionaries use. Order by the first letter (alphabetically), if tie then use the second letter, etc.
- You need to use **design recipe** for this lab assignment. I will offer you the step 4, which is template and you need to cover all other steps.
- **NOTE**: You only allow to use basic library functions of Python! If the string is empty return empty list.

Argument: abc
Returns: [abc, acb, bac, bca, cab, cba]

Step 4: Templet (Pseudocode for a recursive algorithm to generate permutations in lexicographic order.) You must follow this pseudo code.

- If the string contains a single character return a list containing that string
- Loop through all character positions of the string containing the characters to be permuted, for each character:
    - Form a simpler string by removing the character
    - Generate all permutations of the simpler string recursively
    - Add the removed character to the front of each permutation of the simpler word, and
    - add the resulting permutation to a list

- Return all these newly constructed permutations

Submit you Python function in a file called **perm_lex.py** to Polylearn and your test program in **perm_lex_testcases.py. Do not submit your files as zip file and use exactly the same file names and function name.**

Note: For a string with n characters, you program will return a list contain n! strings. Note that n! grows very quickly. For example: 15! is roughly $1.3*10^{12}$. Thus, it is probably not a good idea to test your program with long strings.

Demo your work before deadline ends.

Submissions:

1) location.py
2) location_test_cases.py
3) perm_lex.py
4) perm_lex_test_cases.py