CSE101 – Fall 2019
Programming Assignment #4

Due November 21, 2019 by 11:59pm, KST. The assignment is worth 25 points.

Instructions

For each of the following problems, create an error-free Python program.

- Each program should be submitted in a separate Python file that follows a particular naming convention: Submit the answer for problem 1 as "Assign4Answer1.py" and for problem 2 as "Assign4Answer2.py" and so on.
- These programs should execute properly in PyCharm using the setup we created in lab.
- At the top of every file add your name and Stony Brook email address in a comment.
- Please provide at least 2 test cases for each problem, except for Problem 3 that only has one test case. This means calling the function you have created with an example input, so that when I run the entire program, I see the output of your test cases in the console.
- Please continue to use the naming conventions in Python and programming style that was mentioned in Assignment 1.
- Download the Assignment 4 Supplemental Files and unzip and place these 4 files in your project directory. These will be used for problem 3 and 4.

Problems

Problem 1: Recursive functions for numbers (5 points)

For this problem you must write the functions in a recursive manner (i.e. the function must call itself) – it is not acceptable to submit an iterative solution to these problems.

A. Complete the following function that uses recursion to concatenate the numbers in forward order to form a string.

```
# If n = 9, it returns '0123456789'
# If n = 13, it returns '012345678910111213'
# Pre-condition: n >= 0
#
def concat_to(n):
    return None  # Replace this with your implementation
```

B. Complete the following function that uses recursion to concatenates the numbers in reverse order to form a string.

```
# If n = 9, it returns '9876543210'
# If n = 13, it returns '1312111098765432l0'
# Pre-condition: n >= 0
#
```

```
def concat_reverse_to(n):
    return None  # Replace this with your implementation
```

C. Complete the recursive function `gcd(m, n)` that calculate the greatest common denominator of two numbers with the following rules:

```
# If m = n, it returns n
# If m < n, it returns gcd(m, n-m)
# If m > n, it returns gcd(m-n, n)
#
def gcd(m,n):
    return None  # Replace this with your implementation
```

Problem 2: Recursive functions for lists (6 points)

For this problem you must write the functions in a recursive manner (i.e. the function must call itself) – it is not acceptable to submit an iterative solution to these problems.

A. Complete the following function that uses recursion to find and return the even elements in the list u.

```
# find_evens([1, 2, 3, 4] returns [2, 4]
# find_evens([1, 2, 3, 4, 5, 6, 7, 8, 9, 10] returns [2, 4, 6, 8, 10]
#
def find_evens(u):
    return None  # Replace this with your implementation
```

B. Complete the following recursive function that returns the zip of two lists u and v of the same length. Zipping the lists should place the first element from each into a new array, followed by the second elements, and so on (see example output).

```
# zip([1, 2, 3], [4, 5, 6]) returns [1, 4, 2, 5, 3, 6]
#
def zip(u, v):
    return None  # Replace this with your implementation
```

C. Complete the following recursive function that removes all occurrences of the number `x` from the list `nums`.

```
# remove_number(5, [1, 2, 3, 4, 5, 6, 5, 2, 1]) returns [1, 2, 3, 4, 6, 2, 1]
#
def remove_number(x, nums):
    return None  # Replace this with your implementation
```

Problem 3: Iris dataset (6 points)

Refer to the file irisdata.txt. This data set is commonly used in machine learning experiments and consists of 50 samples from each of three species of Iris flower (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters.

Write a Python program to read the data from the text file and print the averages of the sepal length, sepal width, petal length, and pedal width for all three types of Iris flowers.

Expected Output:

```
Class Iris-setosa:
Average sepallength: 5.006
Average sepalwidth: 3.418
Average petallength: 1.464
Average petalwidth: 0.244

Class Iris-versicolor:
Average sepallength: 5.936
Average sepalwidth: 2.77
Average petallength: 4.26
Average petalwidth: 1.326

Class Iris-virginica:
Average sepallength: 6.588
Average sepalwidth: 2.974
Average petallength: 5.552
Average petalwidth: 2.026
```

Problem 4: Text Analysis (8 points)

Automated authorship detection is the process of using a computer program to analyze a large collection of texts, one of which has an unknown author, and making guesses about the author of that text. The basic idea is to use different statistics from the text – called ”features” in the machine learning community – to form a linguistic ”signature” for each text. For example, one basic linguistic feature utilizes the number of unique words. Once a signature is obtained, comparisons can be made to known authors and the text of their works.

In this problem, you will write a program to analyze some text files to determine three base-level linguistic features:

1. the average word length,

2. the Type-Token Ratio which is the number of different words divided by the total number of words (this is an indicator of how repetitive the text is), and

3. the Hapax Legomena Ratio which is the number of unique words (words only used once) divided by the total number of words.

We also want to see the most frequent words for the text to see how they differ by text. However, if we don’t do any additional text processing, this list will mainly be commonly used words across all texts such as “the”, “to”, “a”, and “I”. One standard approach to this issue is to keep a list of the common words that you want to exclude and then ignore them.

We will take a simpler approach and just look at the 10 most frequent words that are at least 5 characters long. This will exclude a lot of the common connecting words and be easier to implement.

Write a function called `text_features` that takes a file name as its parameter and output those base-level linguistic features and 10 most frequent words (and the count for how many times each word was used) as follows. With ’`file.txt`’ as an input file your program should produce the following output. (Actual numbers

may vary slightly depending on how you handled your data – I reduced all the strings to lowercase and removed punctuation from any words, similar to the spam filtering problem.)

```
Total Words 35
Average Word Length 3.91
Number of Different Words 26
Number of Words Used Once 20
Type-Token Ratio 0.7429
Hapax Legomena Ratio 0.5714

Most Frequent Words (word, count):
confidence 2, cicero 1, tullius 1, marcus 1, started 1, before 1,
defeated 1, twice 1
```

Assume that words in an input file are separated by the usual white space characters.

You will want to write some helper functions that will be useful to write `text_features`. Some possible helper functions would be:

- `get_average_word_length` which returns the average length of the words contained in the file as a real number (float).
- `get_number_of_different_words` which returns the number of different words (where duplicate words are counted as one word).
- `get_number_of_unique_words` which returns the number of words that appear exactly once (duplicate words are not counted).
- `get_most_frequent_words` which returns the most frequent words

Note that I did not specify what these functions take as their input parameter(s). It depends on how you design your program. There are several possible designs that you could consider. You already learned how to build a dictionary with the lab exercises and spam filter example. So, one possibility would be to create a dictionary and pass it to these functions. Another possibility would be to create a list and use it rather than a dictionary. With a list it might be a little easier to write the first function (`get_average_word_length`), it will be a lot easier to write the other functions with a dictionary.

You are provided with a test file, file.txt, and two ”mystery” files, mystery1.txt and mystery2.txt to analyze. The mystery files were obtained from Project Gutenberg, a website with text files of free public-domain books. Below is sample output from my solution for the two mystery files.

mystery1.txt Output:

```
Total Words 164100
Average Word Length 4.04
Number of Different Words 10897
Number of Words Used Once 5597
Type-Token Ratio 0.0664
Hapax Legomena Ratio 0.0341

Most Frequent Words (word, count):
there 767, which 656, could 493, would 428, shall 427, helsing 299,
before 277, again 246, seemed 243, about 239
```

mystery2.txt Output:

```
Total Words 78038
Average Word Length 4.38
Number of Different Words 7694
Number of Words Used Once 3972
Type-Token Ratio 0.0986
Hapax Legomena Ratio 0.0509

Most Frequent Words (word, count):
tuppence 579, tommy 533, there 326, julius 295, would 236, about 213,
don't 189, james 155, think 154, right 142
```