## 16 April 2016, 16 Nov 2013, 18 April 2015, 22 November 2013

Write a function letter_count that takes in number of words (strings) and returns a dictionary with the frequency counts of the various letters. Upper and lower-case characters are considered to be different characters.

In [1]:
```
def letter_count(*words):

    letterDict = {}

    for word in words:

        for letter in word:

            if (ord(letter) < 91 and ord(letter) > 64) or (ord(letter) < 123 and or

                if letterDict.get(letter) == None:

                    letterDict[letter] = 1

                else:

                    letterDict[letter] += 1


    return letterDict
```

In [2]:
```
letter_count("this","this")
```

Out[2]:
```
{'t': 2, 'h': 2, 'i': 2, 's': 2}
```

In [3]:
```
letter_count("baa", "baa","banana")
```

Out[3]:
```
{'b': 3, 'a': 7, 'n': 2}
```

In [4]:
```
letter_count("this","is","a","good","day")
```

Out[4]:
```
{'t': 1, 'h': 1, 'i': 2, 's': 2, 'a': 2, 'g': 1, 'o': 2, 'd': 2, 'y': 1}
```

Write a function most_frequent that takes in number of words (strings) and returns a list of the most frequently-occurring letters in the words. Upper and lower-case characters are considered to be different characters.

In [5]:
```
def most_frequent(*words):

    letterDict = {}
    maxFrequency = 0
    frequentLetterList = []

    for word in words:

        for letter in word:

            if (ord(letter) < 91 and ord(letter) > 64) or (ord(letter) < 123 and or

                if letterDict.get(letter) == None:
```

```
                    letterDict[letter] = 1

                else:

                    letterDict[letter] += 1


    for item in letterDict.items():

        if item[1] > maxFrequency:

            frequentLetterList.clear()
            maxFrequency = item[1]
            frequentLetterList.append(item[0])

        elif item[1] == maxFrequency:

            frequentLetterList.append(item[0])

    frequentLetterList.sort()

    return frequentLetterList
```

In [6]:
```
most_frequent("this","is","a","good")
```

Out[6]:
```
['i', 'o', 's']
```

In [7]:
```
most_frequent("this","is","a","good","day")
```

Out[7]:
```
['a', 'd', 'i', 'o', 's']
```

In [8]:
```
most_frequent("baa", "baa","banana")
```

Out[8]:
```
['a']
```

In this exercise, you are to write a program ngram.py to find the most frequent letter bigram in an English text and report its frequency. The text comprises English words in lower case with no punctuation. All neighbouring words are separated by exactly one space.

In [9]:
```
def frequent_bigram(sentence):

    sentenceArray = sentence.split()
    letterDict = {}
    maxFrequency = 0
    frequentLetterList = []

    for word in sentenceArray:

        wordLength = len(word)-1

        for i in range(wordLength):

            if (ord(word[i]) < 123 and ord(word[i]) > 96) and (ord(word[i+1]) < 123

                if letterDict.get(word[i]+word[i+1]) == None:

                    letterDict[word[i]+word[i+1]] = 1

                else:

                    letterDict[word[i]+word[i+1]] += 1
```

```
    for item in letterDict.items():

        if item[1] > maxFrequency:

            frequentLetterList.clear()
            maxFrequency = item[1]
            frequentLetterList.append(item[0])

        elif item[1] == maxFrequency:

            frequentLetterList.append(item[0])

    result = frequentLetterList.pop()

    return (result, maxFrequency)
```

In [10]:
```
frequent_bigram("a friend in need is a friend indeed")
```

Out[10]:
```
('nd', 3)
```

In [11]:
```
frequent_bigram("mississippi is missing")
```

Out[11]:
```
('is', 4)
```

An anagram is a type of word play, the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once; for example Torchwood can be rearranged into Doctor Who. Write a function are_anagrams that takes two strings as inputs and returns True if the string are anagrams of each other, otherwise it returns False. You may assume there are no spaces in the strings.

In [12]:
```
def are_anagrams(word1,word2):

    array1 = list(word1)
    array2 = list(word2)
    flag = True


    if len(array1) != len(array2):

        return False

    arrayLength = len(array1)

    array1.sort()
    array2.sort()

    for i in range(arrayLength):

        if array1[i] != array2[i]:

            flag = False
            break

    return flag
```

In [13]:
```
are_anagrams('dictionary', 'indicatory')
```

Out[13]:
```
True
```

In [14]:
```
are_anagrams('listen', 'silent')
```

Out[14]:
```
True
```

In [15]:
```
 are_anagrams('melon', 'watermelon')
```

Out[15]:
```
False
```

Write a function has_anagrams that takes as input a list of strings, and returns True if the list contains at least two strings which are anagrams of each other. You may wish to make use of the function are_anagrams from before.

In [16]:
```
def generate_2_Set(wordList, finalSet):

    if len(wordList) == 2 and wordList not in finalSet:
        finalSet.append(wordList)


    else:

        for i in range(len(wordList)):

            wordListCopy = wordList.copy()
            wordListCopy.pop(i)
            generate_2_Set(wordListCopy, finalSet)
```

In [17]:
```
 def has_anagrams(wordList):

        flag = False
        finalSet = []
        generate_2_Set(wordList, finalSet)

        for item in finalSet:

            if are_anagrams(item[0], item[1]) == True:
                flag = True
                break

        return flag
```

In [18]:
```
has_anagrams(['apple', 'banana', 'pear', 'reap'])
```

Out[18]:
```
True
```

In [19]:
```
has_anagrams(['apple', 'banana', 'pear', 'papaya'])
```

Out[19]:
```
False
```

Write a function find_anagrams that takes as input a list of strings, and returns a list of anagrams found in the input

In [20]:
```
def remove_node(word, possibleSet, anagramFamily):

    if len(possibleSet) == 0:

        return
```

```
    anagramFamily.append(word)

    for item in possibleSet:

        if word == item[0] and are_anagrams(item[0],item[1]):
            item = item.copy()
            possibleSet.remove(item)
            remove_node(item[1], possibleSet, anagramFamily)

        elif word == item[1] and are_anagrams(item[0],item[1]):
            item = item.copy()
            possibleSet.remove(item)
            remove_node(item[0], possibleSet, anagramFamily)

        elif are_anagrams(item[0],item[1]) == 0:

            possibleSet.remove(item)
```

In [21]:
```
def find_anagrams(wordList):

    possibleSet = []
    finalSet = []
    generate_2_Set(wordList, possibleSet)

    while len(possibleSet) > 0:
        anagramFamily = []
        word = possibleSet[0][0]
        remove_node(word, possibleSet, anagramFamily)
        if len(anagramFamily) > 1:
            finalSet.append(anagramFamily)

    return finalSet
```

In [22]:
```
find_anagrams(['actress', 'grudge', 'recasts', 'rugged',
'casters', 'apple', 'pear', 'reap'])
```

Out[22]:
```
[['pear', 'reap'], ['rugged', 'grudge'], ['recasts', 'casters', 'actress']]
```

MasterMind Game : After making a guess, the codemaker will inform the codebreaker how many pegs of the guess are correct in both colour and position, i.e., the number of exact matches. Implement a function correct_pegs(code, guess) which takes as inputs two strings, the code and the guess, and returns the number of characters in the guess that match the code in both colour and position.

In [23]:
```
def correct_pegs(code, guess):

    count = 0


    for i in range(len(code)):

        if code[i] == guess[i]:
            count += 1
```

```
    return count
```

In [24]:
```
correct_pegs("rrbk", "rkbb")
```

Out[24]: 2

In addition to the number of correct pegs, the codemaker will also inform the number of correct colour code peg placed in the wrong position. We call this the number of partial matches. If there are duplicated colours in the guess, the number of partial matches should correspond to the number of pegs that are in the wrong position without double counting.

For example, if the code is "wwbbb" and the guess is "bbwww", the number of partial matches is 4. The first two ‘b’s in the guess are in the wrong position, and so are the next two ‘w’s. The last ‘w’ does not match any in the code because the first two ‘w’s in the code is already matched in the guess.

Implement a function check_guess(code, guess) that takes as inputs two strings, the code and the guess, and returns a tuple of 2 elements, the first being the number of exact matches and the second element the number of partial matches.

In [25]:
```
def check_guess(code, guess):

    correct_letters = correct_pegs(code, guess)

    codeDict = {}
    guessDict = {}

    commonCount = 0

    for letter in code:

        if codeDict.get(letter) == None:

            codeDict[letter] = 1

        else:

            codeDict[letter] += 1


    for letter in guess:

        if guessDict.get(letter) == None:

            guessDict[letter] = 1

        else:

            guessDict[letter] += 1


    for key in codeDict.keys():

        if guessDict.get(key) != None:

            commonCount += min(guessDict[key], codeDict[key])

    return (correct_letters, commonCount-correct_letters)
```

In [26]:
```
check_guess("wwbbb", "bbwww")
```

Out[26]:
```
(0, 4)
```

In [27]:
```
check_guess("rrbbyy", "rcbyrg")
```

Out[27]:
```
(2, 2)
```

In [28]:
```
check_guess("rbyg", "bbbb")
```

Out[28]:
```
(1, 0)
```

Write a function odd position digits that takes in a positive integer and returns an integer consisting of only the odd-position (counting from the front) digits

In [29]:
```
def odd_position_digits(num):

    num_string = str(num)

    num_array = []

    final = 0
    for i in range(len(num_string)):
        if i%2 == 0:
            num_array.append(num_string[i])


    length = len(num_array)
    for i in range(length):
        val = int(num_array[i])
        final = final + val*(10**(length-i-1))
    return final
```

In [30]:
```
odd_position_digits(123456789)
```

Out[30]:
```
13579
```

In [31]:
```
odd_position_digits(1234567890)
```

Out[31]:
```
13579
```

In [32]:
```
odd_position_digits(1122334455)
```

Out[32]:
```
12345
```

In [33]:
```
odd_position_digits(183654729)
```

Out[33]:
```
13579
```

In [34]:
```
odd_position_digits(987654321)
```

Out[34]:
```
97531
```

In [35]:
```
 odd_position_digits(123)
```

Out[35]:
```
13
```

Write a function reverse even that takes in a positive integer and returns an integer with the digits at the even positions (counting from the front) reversed.

In [36]:
```
def swap(i , j, array):

    temp = array[i]
    array[i] = array[j]
    array[j] = temp
```

In [37]:
```
def reverse_even(num):

    num_string = str(num)
    num_array = []
    final = 0

    for i in range(len(num_string)):

            num_array.append(num_string[i])

    length = len(num_array)
    last = length + 1

    if length%2 == 1:

        last = last - 1

    for i in range(int(length/2)):

        if i%2 == 1:

            swap(i , last-1-i, num_array)

    for i in range(length):

        val = int(num_array[i])

        final = final + val*(10**(length-i-1))

    return final
```

In [38]:
```
 reverse_even(123456789)
```

Out[38]:
```
183654729
```

In [39]:
```
reverse_even(1234567890)
```

Out[39]:
```
1038567492
```

In [40]:
```
reverse_even(1122334455)
```

Out[40]: 1524334251

In [41]:
```
reverse_even(183654729)
```

Out[41]: 123456789

In [42]:
```
reverse_even(987654321)
```

Out[42]: 927456381

In [43]:
```
reverse_even(123)
```

Out[43]: 123