COMP 1005/1405
Summer 2017 – Tutorial #4

Tutorial Objectives

- Practice writing code using **lists, strings, dictionaries and functions**, as well as the other control structures we have used previously
- Depending on how fair we get in lecture before the midterm, you may have to wait to solve a few of the later problems
- Problems 1-3 involve only lists/strings
- Problems 4-6 require the use of dictionaries
- Problems 7 (lists) and 8 (dictionaries) are difficult questions that you shouldn't expect to see on an assignment or exam, but are worth trying if you want a challenge.

Problem 1

Write a function that takes two lists that are in ascending order as arguments. The function should return a new list that has all of the values from the two argument lists and is also in ascending order (e.g., [1,3,5] and [2,4,6] become [1,2,3,4,5,6]). There are multiple ways of doing this – some take less code to implement, some take less time to execute.

Problem 2

When working with sensor data, it is often a requirement to know about the previous X measurement values (X is some integer number specifying how long you want the memory to be), where measurements outside of the last X (i.e., older measurements) are no longer needed. Write a program that has variables for a list of values and a size X. Add the following functions to your program:

1. **measure(int)** – Takes a single integer value as input, which represents the most recent measurement. The function adds this new measurement to the end of the list. If the list now has more than X measurement, remove a measurement from the front of the list so that there are only the most recent X measurements.
2. **average()** – Returns the average value of the measurements in the list.
3. **min()** – Returns the minimum value of the measurements in the list.
4. **max()** – Returns the maximum value of the measurements in the list.
5. **isdanger()** – Returns true if the difference between the maximum/minimum measurement of the last X values is greater than 10. This could be used in any application where we do not want significant

change to occur over a specific time interval (i.e., the time it takes to make X measurements).

Test your functions by specifying a value for X and adding numbers to the list to verify the correct values are returns by your functions.

Problem 3

Morse code is used to represent letters with long and/or short signals of light or sound. In this problem, you will create a Morse code translator. To start, copy and paste the Morse code dictionary from the file included on the tutorial page into your Python code file. You can use this dictionary to "look up" the Morse code representation of any characters. You then need to write the rest of the program, which should do the following:

1) Use input to read in a message from the user.
2) Iterate over each character in the message and add its Morse code representation to a result string (note: only upper case characters are included in the dictionary)
3) Print out the result string once completed.

Test your program with various input strings. Some examples are included below:

Hello → .... . .-.. .-.. ---

Secret Message → ... . -.-. .-. . -   -- . ...- --. .

Hold up the train → .... --- .-.. -..   ..- .--.   - .... .   - .-. .- .. -.

(Trivia bonus marks for knowing where the second example is from! Trivia bonus marks are just like regular bonus marks, except they're worth nothing in the course…)

Problem 4

Write a function called **lettercount** that takes a single string argument. This function should return a dictionary containing the frequency of each letter within the string.

Problem 5

Searching through a long list to see if a word is present (e.g., executing the Python statement if "great" in wordlist:) can be inefficient, as you may have to search through the entire list. To increase the efficiency, we can store each word in a separate list depending on the first character of that word (i.e., apple, aardvark, anvil, are all stored in a list with other 'a' words). To do this, we will use a dictionary with single character strings as keys (e.g., 'a', 'b', etc.) and lists of words as values (e.g., [‘apple’, ‘aardvark’, ‘anvil’]). This means we have a smaller list size to search for when given a word – we only need to search the list associated with the first character, instead of the entire list of words. Download the wordsearch.py and positivewords.txt file from the tutorial page to start this question. The wordsearch.py file contains code to read in the list of positive words and store it in a list called *wordlist*. There is also a variable called *worddict* created in the file, but it is not used yet. There is a for loop (for word in wordlist:) that iterates over each word stored in *wordlist*. You need to add code inside the body of that for loop that will do the following:

1) Get the first character of the current word
2) Use the first character to add that word to the proper list within the *worddict* dictionary (note: you may have to check if the key exists in the dictionary before trying to add a value).

You may want to print out the *worddict* dictionary variable once you are completed to verify that words are being added correctly (you can ignore the search time that is printed out for now).

Once you have done this, add code to the isindict(word, searchdict) function which returns True if the *word* is stored in *searchdict* and False otherwise. The testing code that is included below the loop where you initialized the dictionary values should now work. This code executes the following algorithm for the list-based search and dictionary-based search:

1) Record the start time
2) Repeat *repeats* times:
    a. Randomly pick a word
    b. Search using function being tested (list-based or dictionary-based)
3) Record the end time

Once the process is finished, the start time is subtracted from the end time to calculate the total time the program took to execute all searches. Once the code is working, you can increase the value of the *repeats* variable. The difference in time should be printed out and, if implemented correctly, the dictionary-based searching should execute significantly faster than the list-based search.

If you have extra time, consider adding a second dictionary-based search that uses the first 2 letters of a word as the key. Does this increase or decrease the search performance relative to the other methods?

Problem 6

An interesting application of dictionaries is their use in caching. In Computer Science, a cache stores data so future requests for that data are performed faster (i.e., they are optimized). Factorial calculations are one thing that can benefit from caching. Note the definition of a factorial is:

$$n! = n * (n-1) * (n-2) * \ldots * 3 * 2 * 1$$

When you calculate the factorial in the order specified in this equation (multiplying from highest to lowest), you can stop early if you know the factorial value of a number between n and 1. For example, if we know the value of 5! is 120, then we can calculate the value of 7! two possible ways:

$$7! = 7 * 6 * 5 * 4 * 3 * 2 * 1$$
$$7! = 7 * 6 * 5! = 7 * 6 * 120$$

Download the factorial.py file from the tutorial page as a starting point, or quickly write your own factorial method if you haven't done so already. Create a new function called cachedfactorial(int, dictionary), which takes the number to calculate a factorial and a dictionary that stores the cached values. The cachedfactorial function should calculate the factorial, but should update the cache whenever a new factorial value is calculated and use the cache to stop early if possible.

If you have time, evaluate the performance of each function as was done in the previous question by repeatedly calculating factorial values. The code for this is included in the file, you just have to remove the ``` comments. Try calculating the factorial values from 1-5000 (does it make a difference if you go from 1-5000 or from 5000-1? Why?). Try using the random integer function to calculate factorial values of numbers between 1-500 many times.

Problem 7

Assume you are given a 2-dimensional list that represents a Tic-Tac-Toe board (see below for board example, Google Tic-Tac-Toe if you are unfamiliar with the game) where "X" is used to represent one player, "O" is used to represent another player, and "" represents an empty space. Write the following functions:

1. **checkwinner(list)** – Takes the 2D game board list as input. This should return "X" if the X player has won the game, "O" if the O player has won the game, "T" if the game board is full and the game is a tie, or "" if they game neither player has won but the game is not over.
2. **getwinningmove(list, str)** – Inputs are the 2D game board list and either "X" or "O" to represent the player making the move. This function returns the row and the column (the first and second index to reference the location in the 2D list) that will allow the current player to win the game. If there is no single move that will let the player win the game, this function should return -1, -1.
3. **makemove(list, str)** – Inputs are the 2D game board list and either "X" or "O" to represent the player making the move. If there is a winning move possible, this function should place the symbol ("X" or "O") at the winning move location in the game board list. If there is not a winning move possible, this function should place the symbol at any available location (you can chose randomly or just find one).

Test your functions with various game board lists. If you are looking for a further challenge, you can finish implementing the Tic-Tac-Toe game by repeatedly making moves for each player. You can further modify the game to get one of the players' moves from the user, so the user can play the game against the simple artificial intelligence you have created. You can also improve the artificial intelligence by adding in more rules for the **makemove** function.

Problem 8

This is an interesting problem involving dictionaries (I think it is, at least) that I was originally going to include in an assignment, but I decided it was overly complex and difficult to explain. If you feel like trying it out though, here it is:

In this problem, you will develop a program that will store a large number of valid words and must be capable of checking if a given string is a valid word (e.g., like a spell-check program). An easy way to do this is to store all of the words in a simple list and search through this list whenever you want to verify a given string is valid. This approach, however, will take up a large amount of memory and will take a long time to determine whether a word is present or not, since you have to search through a long list.

Instead, you will implement a much more efficient solution. Consider the following words:

apple, application, apply, appetizer, apples, applies, applications

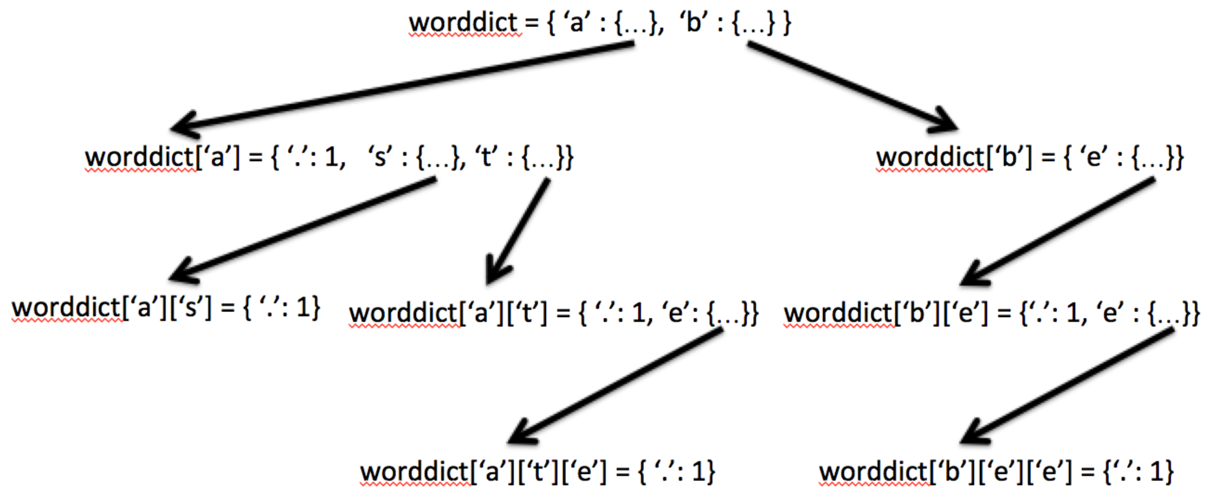
These words all share several of the same starting characters, which would be duplicated in a list-based implementation of the program. You will remove this unnecessary reproduction of information by implementing a dictionary-based solution. Within your solution, there will be a single base dictionary, which will have keys representing characters of the alphabet (these will represent the first character of a word) and values that are also dictionaries. Note that the values of this base dictionary, which are also dictionaries, can be used to store characters representing the second character of the word as keys with values that are dictionaries, and so on. iIf you want an idea of where to start, check out the wordstorage-three.py file, which has a solution that works only for words of length 3. That file would need to be modified to use loops, so words of any size can be handled.

Write a program that has a single base dictionary variable to store words and the following three functions:
1) addword(string): This function will add a word to your word storage dictionary. You will have to iterate over each character of the word and move deeper into the word storage dictionary (i.e., into the internal dictionaries associated with each character index), ensuring that the necessary key/values exist or are created. Note: to signify a word has ended, add an entry in the dictionary value associated with the final character of the word that has a key of '.' and a value of 1.

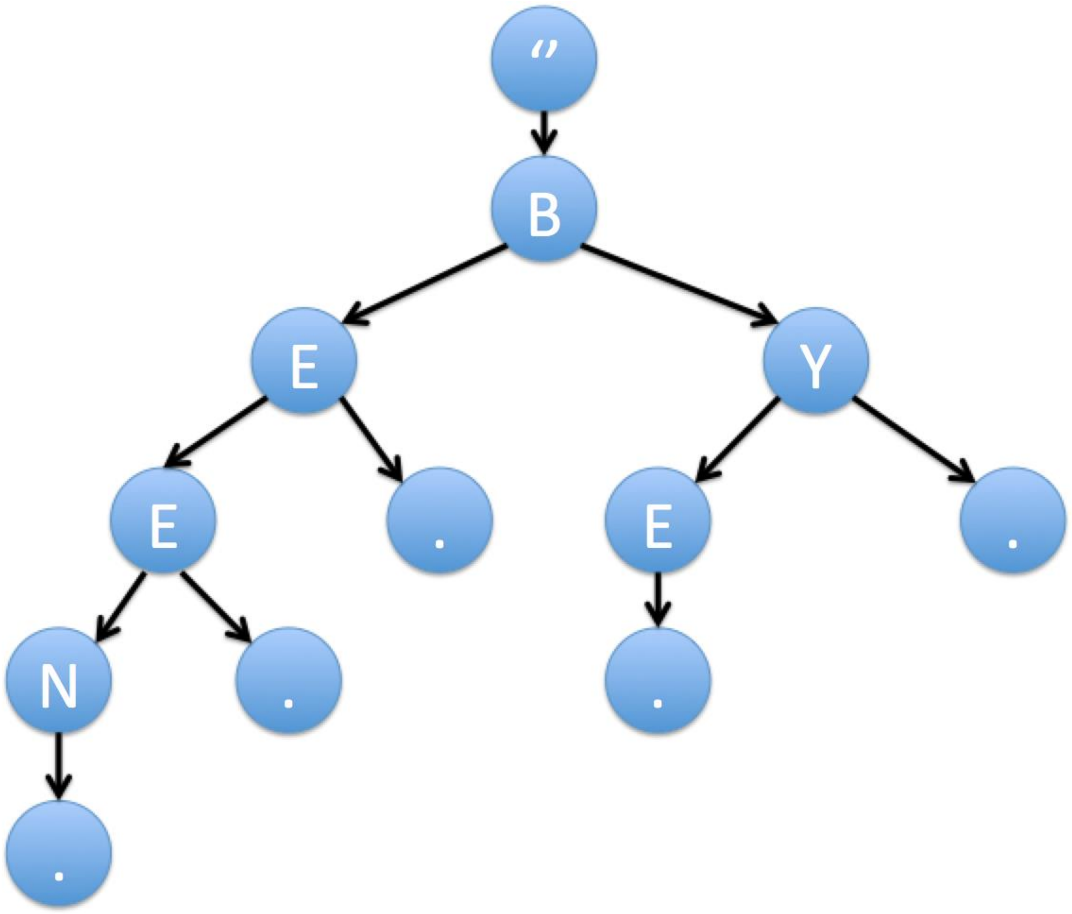
2) checkword(string): Returns true if the word is contained in the word storage dictionary. You can use a similar process to search for a word as you do when adding a word. If the necessary key does not exist for a certain character in the word, you know that the word has not been added (otherwise, you would have added that key during the add process). In addition, if you can find keys for each character in the word, if the key '.' exists in the final character's dictionary, you know this is a valid word (otherwise, it is not a valid word).
3) initialize(string): Takes a string representing a filename. Each line in the specified file will contain a single word. This function will open the specified file and read each word, adding it to the dictionary using the addword function.

An example of the resulting dictionary structure after the words be, bee, been, by, and bye have been inserted, is shown below:
{'b': {'y': {'.': 1, 'e': {'.': 1}}, 'e': {'.': 1, 'e': {'.': 1, 'n': {'.': 1}}}}}

Two (poorly drawn) visual examples that may also help:



worddict = { 'a' : {…}, 'b' : {…} }
worddict['a'] = { '.': 1, 's' : {…}, 't' : {…}}
worddict['b'] = { 'e' : {…}}
worddict['a']['s'] = { '.': 1}
worddict['a']['t'] = { '.': 1, 'e': {…}}
worddict['b']['e'] = {'.': 1, 'e' : {…}}
worddict['a']['t']['e'] = { '.': 1}
worddict['b']['e']['e'] = {'.': 1}


''
B
E
Y
E
.
E
.
N
.
.
.