COMP 1405B
Fall 2019 – Practice Problems #6

Objectives

- Continue to practice using the programming concepts we have covered already
- Practice writing code that uses lists
- Practice writing code that uses string operations
- Practice designing algorithms that work with the list data structure and string operations
- Practice following a precise specification

Remember to test your solutions to verify their correctness and discuss your solutions with both other students and TAs in the course. Verifying your solutions with TAs will be a good way of ensuring that you are on a path to success in the course.

Problem 1 (Merging Sorted Lists - **)

Write a function that takes two lists that are in ascending order as arguments. The function should return a new list that has all 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. Try thinking of alternate approaches to solving this problem. You should try to complete this problem without using the built-in sort functions of Python.

Problem 2 (Storing Sensor Data - **)

When working with sensor data and other time-sensitive information, the program is often required to know about the previous X measurement values, where X is an integer number specifying how long you want the memory to be, and forget about measurements outside of the last X (i.e., older measurements). Write a program that has a list variable to store values and an integer X variable to represent the desired memory length. Your program should not use the built-in list aggregation functions from Python (e.g., max, min, etc.). 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 returned by your functions.

Problem 3 (Largest Number - **)

Write a function called maxvalue that accepts a list variable as input. The input list will contain only numbers. The function must return the largest number in the list. Note that you should write the implementation yourself and not use the built-in Python max function.

Problem 4 (Largest Volume - **)

Write a function called maxvolume that accepts a list variable as input. The input list will be two dimensions. Each item in the input list will be a 3-element list representing the dimensions of a box (width, height, depth). The function must return the volume of the largest box in the list. Note: volume = width * height * depth. Test your function with some lists containing boxes with various dimensions.

Problem 5 (Top X - ****)

Write a function called topx that accepts two inputs: a list of numbers and an integer (X). The function should return a list of length X that contains the X largest values in the input list. For example, the result for topx([8, 3, 1, 2, 9, 4, 5, 7, 6], 4) would be [6, 7, 8, 9]. You can return the result list in any order, so it does not need to be sorted.

Problem 6 (Getting Rich - ***)

Assume you have a list of stock prices given to you in a list variable. Write a program that determines the largest profit you could have made by buying the stock at one price (P_{buy}) in the list and selling the stock at a different price (P_{sell}) that occurs later in the list. Assume that you buy one unit of the stock and that there are no fees, so your profit will always be $P_{sell} – P_{buy}$. Test your program with various made up lists of stock prices. Note: One way to do this is to use nested loops to iterate over each possible combination of P_{buy} and P_{sell}, where the sell index is greater than the buy index.

Problem 7 (Mastermind - ***)

For this question you will create a version of the popular game mastermind! Go here to play an online version of the game: Mastermind online (Note, the game you make will look and behave differently). The rules for your game are as follows:

- You randomly generate a password of 5 digits between [0-9]. (e.g.: [3,1,4,4,2])
- The user then has 10 turns to guess the password.
- Each turn the user makes one guess, and your program reports how many digits were correct.
    - Note: a correct guess is the right number in the right location.
    - Note: be careful with datatypes! Make sure you are comparing strings with strings, and integers with integers.
- If the user guesses all five digits correctly, the game ends with a win message.
- If the user runs out of turns, the game ends with a lose message.

You may assume that the user correctly enters 5 integers, however you must allow them to enter all 5 into a single prompt, separated by spaces.

Example Game Output (text entered by the user is highlighted):
I've set my password, enter 5 digits in the range [0-9] separated by spaces (e.g. 1 3 2 4 4):
10 guesses remaining > 1 2 3 4 5
1 of 5 correct
9 guesses remaining > 2 3 4 5 1
0 of 5 correct
8 guesses remaining > 2 2 3 4 5
1 of 5 correct
7 guesses remaining > 3 3 3 4 5
1 of 5 correct
6 guesses remaining > 4 4 4 4 5
2 of 5 correct
5 guesses remaining > 5 5 4 5 5
1 of 5 correct
4 guesses remaining > 3 2 1 4 5
2 of 5 correct
3 guesses remaining > 1 1 1 4 5
3 of 5 correct
2 guesses remaining > 2 1 1 4 5
3 of 5 correct
1 guesses remaining > 3 1 3 4 5
2 of 5 correct

You'll never get my treasure because the password was ['4', '1', '1', '2', '5']

Problem 8 (Queue Implementation - ***)

A queue is a data structure used to store items. The next item removed from a queue is always the item that has been in the queue the longest (i.e., was added first - think of a line of people waiting to buy something). In this question, you will use lists to represent queues and write several functions to perform operations on queue lists. Your file will need the following components:

1. A global variable representing the maximum size of the queue (i.e., how many items can be stored in the queue at one time). This vaariable should have a **default value of 10**.
2. An **enqueue(queue, value)** function that accepts a list representing a **queue** and a **value.** The value may be any data type. If there is room in the **queue** list (i.e., the current size is less than the maximum size), **value** will be added to the end of **queue** and the function will return True. If there is no room in the queue, the function will return False.
3. A **dequeue(queue)** function. If the **queue** list has any items in it, this function will remove and return the first item (i.e., from the front of the queue). If **queue** does not have any items in it, the function should return None. In this case, None is the specific Python value representing nothing, not the string value “None”.
4. A **peek(queue)** function. If the **queue** list has any items in it, this function will return the value of the first item in the queue but leave that item in the queue (different from the dequeue function). If **queue** does not have any items in it, the function should return None. In this case, None is the specific Python value representing nothing, not the string value “None”.
5. An **isempty(queue)** function. This function should return True if the **queue** list is empty (no items) and False otherwise.
6. A **multienqueue(queue, items)** function, which takes a **queue** list and a list of elements called **items**. This function should add as many of the elements from the **items** list to the **queue** list as possible (i.e., keep adding until the queue is full or all items have been added) and return the number of items that were added successfully.
7. A **multidequeue(queue, number)** function, which takes a **queue** list and an integer **number**. This function should attempt to dequeue up to **number** items from the queue and return a new list containing all items removed (note: this may be less than **number** if the queue has become empty).

Save your queue function code into a file called **myqueue.py** that **only includes the functions and the maximum size variable** (i.e., no testing code, print statements, etc.). You can test your code by importing your myqueue.py file as a module into a separate Python code file and calling the functions with specific values. To aid in your testing, a queuetester.py file has been included in the PP6-Resources.zip file. If you run this Python file within the same directory as your myqueue.py file, it will perform a number of the queue operations. The testing program will also print out the expected values, so you can verify that your queue functions are working correctly. You should perform additional tests to further verify your functions’ correctness.

Problem 9 (Stack Implementation - ***)

In the previous problem, you implemented a queue. A stack is a similar type of data structure, but instead of using the first-in-first-out principle of a queue, a stack uses last-in-first-out. This means that the next item taken from a stack is the most recent item that was inserted. As an example, consider a stack of plates or paper in which you can only look at the top item (peek), add an item to the top of the stack (push), or remove the item from the top of the stack (pop). Create a **mystack.py** file that implements the following functions:

1) **push(stack, value)**: Adds the argument **value** onto the top of the **stack** list.
2) **pop(stack)**: If the **stack** list is empty, this function returns None. Otherwise, this function removes and returns the top value from the **stack** list.
3) **isempty(stack)**: Returns True if the **stack** list is empty (i.e., has no items), False otherwise.
4) **peak(stack)**: If the stack is not empty, returns but does not remove the top value from the **stack** list. Otherwise, returns None.

Save your stack function code into a file called **mystack.py** that **only includes the functions** (i.e., no testing code, print statements, etc.). You can use the **stacktester.py** file from cuLearn to verify the operations of your stack are working as they should be. You should test your stack further to ensure it consistently works properly. The next problem will further test your stack implementation.

Problem 10 (Parentheses Validation - ***)

A string is considered to have valid parentheses if there are an equal number of opening/closing brackets and each closing bracket – e.g.,), }, or] – matches the most recent, unmatched opening bracket – e.g., (, {, or [. For example:

(({ })) is valid
{ [{ ( ) }] } is valid
(){}[] is valid
((}) is invalid

Checking the validity of parentheses is important in verifying the validity of programming code and mathematical statements. Create a new Python file called **validator.py** with a single function called **isvalid(string)**. This function will take a **single string that can consist of any characters** and use your stack implementation from the previous problem to determine if the parentheses are valid or not. The function must return True if the string has valid parentheses and False otherwise. Your function only needs to consider the three types of bracket characters mentioned above – the remaining characters, which could represent code, numbers, arithmetic operators, etc., can be ignored. You can use the validatorchecker.py file from the PP6-Resources.zip file to check whether your function is working correctly. If you are having trouble figuring out how to solve the problem, consider how you would validate each of the sets of brackets above using the stack operations available. Note that the programming for this problem is not difficult, but the verification algorithm may take some thought.

Problem 11 (Start and End Words - ***)

For the purposes of this question, we will define a word as ending a sentence if that word is immediately followed by a period. For example, in the text “This is a sentence. The previous sentence had four words.”, the ending words are ‘sentence’ and ‘words’. In a similar fashion, we will define the starting word of a sentence as any word that is preceded by the end of a sentence, as well as the first word in a string. The starting words from the previous example text would be “This” and “The”. Write a program that has:

1. A **startwords** function that takes a single string argument. This function must return a list of all sentence starting words that appear in the given string. There should be no duplicate entries in the returned list.
2. An **endwords** function that takes a single string argument. This function must return a list of all sentence ending words that appear in the given string. There should be no duplicate entries in the returned list and the periods should not be included in the ending words.

Save your function code into a file called **sentences.py** that **only includes the functions and necessary variables** (i.e., no testing code, print statements, etc.) and add it to your submission zip. You can test this program in a similar fashion to previous problems. A sentencetester.py file has been added to cuLearn to aid in testing.

Problem 12 (JSON - ***)

In this problem you will create a simple JSON parsing module called **jsonparser.py**. JSON is a human-readable file format that consists of key/value pairs. For the purposes of this tutorial, we will assume that all JSON strings follow the form below (the number of key/value pairs can vary):

{“key1”:”value1”,”key2”:”value2”,”keyn”:”valuen”}

More precisely:

1. Each JSON string will begin with a {
2. Each JSON string will end with a }
3. Each JSON string will have some number of key/value pairs, each of which is separated by a comma
4. Each matching key and value in a JSON string will be separated by a colon
5. The individual keys and values in a JSON string are enclosed within double quotation marks
6. There will be no empty spaces outside of the double-quotation marks.

JSON also supports arrays (lists of values) and nesting but we will not consider them within this problem

Write a function called **jsontolist(string)** that accepts a string argument that will have the JSON format specified above. You can assume any string your function will be

given has the proper format. This function must use string and list operations to extract each key/value pair and return them in a 2D list. For example, if the JSON string is:

{"firstname":"dave","lastname":"mckenney","position":"instructor"}

The function should return a 2D list that looks like:

[[“firstname”, “dave”], [“lastname”, “mckenney”], [“position”, “instructor”]]

Each element in the returned list is a list with two elements corresponding to a key and its matching value. Several JSON strings and their corresponding lists are included at the end of the problem so you can check your function before proceeding.

Write a second function called **getvalue(jsonlist, key)** that accepts two arguments: a 2D JSON list in the format returned by the **jsontolist(string)** function, and a string/key specifying the key you want to retrieve the value for. This function should iterate through the list to find the matching key and return the corresponding value. If no items within the given list match (i.e., the key does not exist), the function should return the value None, which is a special Python value representing nothing. As an example, if you saved the list from the previous function example into a variable called *mylist*, the output of the getvalue function would be:

```
getvalue(mylist, “firstname”) → “dave”
getvalue(mylist, “lastname”) → “mckenney”
getvalue(mylist, “position”) → “instructor”
getvalue(mylist, “height”) → None
```

Write a function called **setvalue(jsonlist, key, newvalue)** that accepts three arguments: a 2D JSON list in the format returned by the **jsontolist(string)** function, a string/key specifying the key you want to set the value for, and a string/value that should represents the new value for the given key. For example, if you had the same *mylist* variable from the previous function example and you executed *setvalue(mylist, “position”, “president”)*, *mylist* would then look like:

[[“firstname”, “dave”], [“lastname”, “mckenney”], [“position”, “president”]]

Write a function called **listtojson(jsonlist)** that accepts a 2D list argument in the same format that has been used previously. The function must return a JSON-formatted string that contains the key/value pairs in the 2D list. For example, if you executed *listtojson(mylist)* using the modified list from above, your function should return the following string:

{"firstname":"dave","lastname":"mckenney","position":"president"}

There are two other functions you could try adding to your module:

remove(jsonlist, key) – accepts a 2D list and a string/key. This function should find and remove the key/value pair in jsonlist that matches the argument key. If the key is not present, the function does not need to make any changes. The function should return True to indicate the key was found and removed, or False to indicate the key could not be found.
add(jsonlist, key, value) – accepts a 2d list, a string/key, and a string/value. This function should add the key/value pair to the list if the key is not already present. The function should return True if the key was added and False if it could not be added because there was already a key with the same name.

Example JSON Strings and Corresponding Lists:

string1 = '{"firstname":"dave","lastname":"mckenney","position":"instructor"}'
jsontolist(string1) → [[‘firstname’, 'dave'], [‘lastname’, 'mckenney'], ['position', 'instructor']]

string2 = '{" firstname ":"Steve","lastname":"Pearce","position":"MVP","teamname":"Boston Red Sox"}'
jsontolist(string2) → [['firstname', 'Steve'], ['lastname', 'Pearce'], ['position', 'MVP'], ['teamname', 'Boston Red Sox']]

string3 = '{"question":"What is the square root of 49?","answer":"7","points":"100"}'
jsontolist(string3) → [['question', 'What is the square root of 49?'], ['answer', '7'], ['points', '100']]

string4 = '{"symbol":"TSLA","date":"2018-11-01","open":"338.26","close":"344.28"}'
jsontolist(string4) → [['symbol', 'TSLA'], ['date', '2018-11-01'], ['open', '338.26'], ['close', '344.28']]

Problem 13 (Order/Inventory System Progressive Part 3 - ***)

In the previous two parts of this progressive problem, you developed an ordering system and a sales analysis program for an electronics store. For this problem, you will use lists to create a more detailed ordering system that will allow products to be added/removed from the stores’ catalogue, track the stock levels of products, and provide the capability to search for products.

While you are free to store the data in any way you want, so long as the functions operate correctly, the problem description will use the suggested 2D list structure included below:

```
products = [
            [product#1 name, product#1 description, product#1 price, product#1 stock],
            [product#2 name, product#2 description, product#2 price, product#2 stock],
            …etc...
        ]
```

This structure consists of a product list that contains a single 4-item list to represent each product’s information. As you will be using this list (or whatever storage mechanism you choose)

across most/all of the functions in your program, it should be defined outside of any function body.

You should use string operations to perform all the searches and name comparisons in a case-insensitive manner. This means that all of the following strings would be considered matches: “Tablet”, “tablet”, “TABLET”, “TaBLeT”, etc..

Note that you do not need to complete the functions in the order they are specified. If I were to recommend an order, I would implement the skeleton of the menu function (part 8) first and then add and test each of the additional functions in the following order: load inventory (part 1), list products (part 7), add product (part 3), save inventory (part 2), add product stock (part 5), remove product (part 4), sell product (part 6).

Part 1 – Load Inventory Function

Create a function called load_inventory(filename). The filename argument in this case specifies the name of a file that contains all the inventory/product information for the store, including product names, descriptions, prices, and stock levels. This function should clear any information already in the product list (i.e., a fresh start) and then re-initialize the product list using the file specified by the filename argument. You can structure your file any way you like, but a suggested line-by-line structure similar to the one used previously is given below:

Product #1 Name
Product #1 Description
Product #1 Price
Product #1 Stock
Product #2 Name
Product #2 Description
Product #2 Price
Product #2 Stock
…etc…

Part 2 – Save Inventory Function

Add another function to your file called save_inventory(filename). This function should save all the current product list information into the file with the specified name. It should also overwrite any data that is already in the file. Once you have defined both the save and load inventory functions, you will be able to load some stored information, perform some changes using the other functions, then save those changes back to the file so they are remembered when the program restarts and the file is loaded again.

Part 3 – Add Product Function

Add another function to your file called add_new_product(name, desc, price, stock). This function accepts four arguments representing the information for a new product. It should add that new product to the store’s product list only if no other product in the store’s product list has the same name (case should be ignored, so “Tablet” and “tablet” would be considered the same name). This will ensure the names of products in the

store are unique, which will be important to remove ambiguity in other functions (e.g., one that removes products by name). This function must return True to indicate the product was successfully added or False to indicate the product was not added.

Part 4 – Remove Product Function

Add another function to the file called remove_product(name). This function must go through the store's product list and remove a product from the list if that product's name matches the given name argument (again, this should not be case sensitive). The function should return True to indicate a product was removed and False if a product was not removed.

Part 5 – Add Product Stock Function

Add another function to the file called add_product_stock(name, units). This function must search through the store's product list to find a product with the matching name. If a matching product is found, the stock of that product in the store should be increased by the given number of units. The function should return True to indicate the stock level of a product was updated and False to indicate no updates were made (e.g., if a product was not found with that name).

Part 6 – Sell Product Function

Add another function to the file called sell_product(name, units). This function must search through the store's product list to find a product with the matching name. If a matching product is found and the stock level of that product is at least the number of units specified (i.e., the store has that many units available to sell), then the stock level of that product should be decreased by the specified number of units. This function must return True to indicate that a product was found and the stock was updated. If a product with that name is not found or there are not enough units in stock to complete the sale, the function must return False.

Part 7 – List Products Function

Add another function to the file called list_products(keyword), which accepts a single string argument. This function will search through the list of products and print out those that match the given keyword argument. If the keyword is "*", then all products should be considered a match (essentially, this prints the entire store inventory). Otherwise, a product should match if the keyword is contained either in the product name or the product description (not case sensitive, "tablet" should match "Tablet", and "TABLET", etc.). The printout should format the product information in an easy to read fashion that includes the name, description, price and number of units in stock.

Part 8 – System Menu

Add a final function to the file called main(). This function must print out a menu with the 9 options detailed in the table below and allow the user to repeatedly enter choices until they chose to quit. An example of what the output might look like is included in the menu-demo.txt file within the PP6-Resources.zip file.

Option Number	Option Text	Description
1	Load Inventory	Asks the user for a filename and then uses the load_inventory function to perform the load operation.
2	Save Inventory	Asks the user for a filename and then uses the save_inventory function to perform the save operation.
3	Add New Product	Ask the user to enter the four pieces of product information and then use the add_new_product function to add the product to the catalogue. Should print out a message indicating whether the product was added successfully or not (use the return value of the function to decide).
4	Add Product Stock	Ask the user to enter the product name and a number of units, then use the add_product_stock function to update the stock levels of the product. Should print out a message indicating whether the update was successful or not (use the return value of the function to decide).
5	List Products	Use the list_products function to print all the products in the store catalogue.
6	Search Products	Ask the user to enter a keyword to search for. Use the list_products function to print any matching products.
7	Sell Products	Ask the user for a product name and a number of units to sell. Use the sell_product function to perform the sell operation. Should print out a message indicating whether the sale was successful or not (use the return value of the function to decide).
8	Remove Products	Ask the user for a product name and then use the remove_product function to take the product out of the store inventory. Should print out a message indicating whether the removal was successful or not (use the return value of the function to decide).
9	Quit	Exits the loop and the program stops.

Problem 14 (Search Engine Progressive Problem 2 - ****)

In Part 1 of the Search Engine progressive problem, you created a search engine that used local files. There was a pages.txt file that included the list of pages/files you had access to, along with each of the files containing webpage content. Currently though, the only way to add more pages is to manually create new files and modify the pages.txt file. In this part, we will create a web crawler program. This program will be capable of exploring the web, finding new pages, and saving their contents to your hard drive. Implementing a web crawler will give us an automatic way of generating many files to search. In the next part, we will look at ways to improve the storage and searching

functionality. Note: this web crawler will only work for well-defined web pages and likely will not work well if let loose on the Internet in general (there are modules to handle pages that are not formatted exactly as we hope).

To start this problem, you will need the webdev.py module, which can be found in the PP6-Resources.zip file on cuLearn. This module has three functions for performing various important tasks:

readurl(url): Tries to read and return the content of the web page at the given url (a string). Returns an empty string if there is an error while reading.

getlinks(text): Returns a list of all URLs (pages) that are linked to in the given text (a string).

striphtml(text): Removes all HTML tags from the given text (a string) and returns the resulting string. The text returned represents the words on the page.

You can import the module into your own program and use all three functions. It may be beneficial to work out the getlinks(text) and striphtml(text) functions on your own once we have covered string operations in class.

The web crawler will use two lists of URLs (URLs are strings): candidates and crawled. The candidates list will contain URLs of pages that the web crawler has discovered and could read/save. The crawled list will contain URLs that the web crawler has already saved, which can be used to avoid reading/saving the same page multiple times. You may also want to create a maxpages variable that will represent the maximum number of pages you want to crawl/save. This will prevent your program running for a long time while you are still testing. A maximum of 5 pages is a good starting point. You can increase it once you are confident the web crawler is working. The following paragraph will explain how the web crawler should work. There is pseudocode included at the end of this problem to provide extra direction if you are struggling.

The web crawler should start with a single URL added to the candidates list. This is the only manual part of the program. Start with candidates = ["https://sikaman.dyndns.org/courses/4601/resources/N-0.html"]. As long as there are candidate pages remaining, the web crawler should take one URL out of the candidates list and read that page's contents. Once the page's contents have been read, the web crawler should extract all of the linked URLs within the page and add them to the candidates list. Don't add URLs that have already been crawled/saved or that are already in the candidates list, or you could end up going in circles. After this, the HTML should be removed from the page contents, leaving just the text content. Write the remaining page contents to a file and add the file name to the list of pages (pages.txt from the last part). Note that you can't save files that have special characters in the name (e.g., : and /). For now, you can extract the last part of the URL, which represents

the page name (e.g., N-0.html) and use that for the file name. Once you are finished crawling and saving the current page, add the URL to the crawled list so you won't revisit the same page again. This process should repeat until there are no candidate pages remaining or your list of crawled pages has at least maxpages URLs.

If everything went well, you should end up with maxpages files saved on your computer, as well as a list of the files in the pages.txt file. Some example output that you might expect (especially if you get the next candidate by removing the first element from the candidates list) is included in web-crawler-example-output.txt. If you removed/added pages in a different order, your results may be slightly different. You now have a way to find many pages and save their contents easily. Your previous search example operated under the assumption that each word would be on a separate line. You can either modify your crawler so that it splits the page contents at each word (string operations) and writes those to each line, or change your search example from the previous part to use string operations to break apart the text into a list of words. You could also consider saving everything in lower-case to allow you to search without considering case. You could also modify your file-based approach to load all the file information into lists. This will provide significantly faster searching functionality when compared to reading through all of the files for each search operation. We will improve further on the search efficiency in the final part of this problem.

If you get stuck or things aren't working, remember to break the problem down and tackle it piece-by-piece. You can also add print statements within your code to see what values different variables have as the program runs. This can be a useful way to find errors in your code and verify intermediate steps are producing the correct values.

Web Crawler Pseudocode

1. Create *candidates* list and *crawled* list
2. Add "https://sikaman.dyndns.org/courses/4601/resources/N-0.html" to the *candidates* list
3. Open the pages.txt file for writing
4. Set *maxpages* to 5 (or whatever number you want)
5. While *candidates* isn't empty and *crawled* does not have maxpages:
    a. Remove first URL from *candidates* list and save in variable called *current*
    b. Set *content* to be the page contents of the *current* URL (webdev module)
    c. Set *links* to be the set of links contained in *content* (webdev module)
    d. For each URL in *links*:
        i. If the URL is not in *candidates* and is not in *crawled*:
            1. Add the URL to *candidates*

    e. Remove HTML from *content*, which leaves just the text content (webdev module)
    f. Let *pagename* be the last part of *current* URL (anything after the last / in *current*)
    g. Write *pagename* to the pages.txt file (don't forget "\n")
    h. Open *pagename* file, write *contents* to the file, close the file
    i. Add *current* to *crawled* list
6. Close pages.txt file

Problem 15 (Tic-Tac-Toe Progressive Problem 2 - ***)

In Part 1 of the Tic-Tac-Toe problems, you created a representation of a tic-tac-toe board using 9 variables and implemented the logic to check if a tic-tac-toe game was finished based on the value of those variables. We now have additional tools at our disposal, including loops, functions, and lists, which will allow us to implement a complete human-playable tic-tac-toe game.

To start this part, create a new game board representation that uses a 2D list variable (3x3) to represent the tic-tac-toe board, instead of 9 individual variables. Functionally, this is equivalent to the 9 variable solution from part 1, except now board[0][0] represents the top-left space, board [1][1] represents the middle space, board[0][2] represents the top-right space, etc.. You can think of the first index as representing the 'row' of the board and the second index representing the 'column'.

Create a function called isgameover() that uses the list variable and returns True if the game is over and False otherwise. You can simply replace your references to the 9 variables with the appropriate list indices, or reduce the amount of code by using loops (e.g., you can use a for loop to check all the rows using the same code). Test this function before moving on.

Create a second function called printboard() that prints out the board in a nicely formatted way. You can modify the code that printed out the board from Part 1. Create a third function called resetboard(). This should reset the board to be all empty spaces.

Finally, you can create the logic to run an entire game of tic-tac-toe. The pseudocode for a tic-tac-toe game looks something like this:

1. Select a starting player randomly
2. Print the board
3. While the game is not over:
    a. Ask the current player to pick a place to play
    b. If the place is not valid (i.e., it is filled), go back to 3a

    c. Place the current player's symbol in the place they selected
    d. Print the board
    e. Swap the current player
4. Determine the winner and display
5. Reset the board and go back to step 1

There are various ways you can implement this logic. One way is to remember the current player by using a variable that contains their symbol ("X" or "O"). You can then swap the value after each turn and include it in the prompt so the players know who should be picking a location. To get a location on the board, you can ask the player to specify a row and column value. Alternatively, you can label the locations on the board from 1-9 and have them select a specific number (you will have to find a way to convert 1-9 to the 2 list index values). Once you have all of this, you should have a 2-player tic-tac-toe game that uses only human players.

After we have covered recursion, we will implement a well-known AI algorithm that will be impossible to beat (called minimax). If you want to experiment and/or get started, you can modify your code to support more naive AI players. Write two new functions – gethumanmove() and getaimove(). The gethumanmove() function can use the same logic you already had: print out a prompt and have the user enter a row/column. You can write your own logic in the getaimove() to use the state of the game (i.e., the values in the board list) to decide on a move. A basic AI will just pick a random spot on the board to play, but this won't win very often. A slightly more sophisticated AI will check all spots on the board and see if any of them represent a winning move (you could write an additional function for this). You will also need to modify your game so that it is aware of which players are human and which are AI (a Boolean variable for each player?). Your turn logic will then involve determining if the current player is human, in which case gethumanmove() should be used, or an AI player, in which case, getaimove() is used.

Problem 16 (Social Network Progressive Problem 2 - ****)

In the social network set of problems, we will develop a model of a social network and implement some algorithms to work on that model (friend recommendations, targeted advertising, etc.). For the purposes of these problems, we will assume a single person's 'profile' in the social network can be modelled using a list with the following structure:

[name, [locationx, locationy], [friends list], [interest list]]

We can use a 2D list to store all the profiles. We could also expand the model by adding more elements to each person's profile list. As we will be using lists, we will be working with many index values (e.g., 0 is name, 1 is location, etc.). Instead of using these numbers everywhere in your code, it is good practice to define variables at the start of your program to represent these index values with words and use these variable names

in the rest of your code. For example, nameindex = 0, friendsindex = 2, etc.. This will also make extending your profile model easier – if you want to change where something is stored (e.g., because you added a new piece of information), you will only need to change the value of the variable that represents that index.

To get started, download the socialnetwork.py file from the tutorial page. This file has two 2D lists defined: *smallnetwork* and *largenetwork*. The smaller network has only a few profiles and will be easier to work with while you figure out some of your implementations. You can change the list that you assign to the *network* variable to easily select which model you want to work with. You can also cut/paste the larger network into an alternate text file to save for later, if you don't like the amount of space it takes up in your code.

To start, we will write some functions to find out some information about the people in the network. First, write a function called getnames(), which should return a list of all of the names of the people in the network. Print it out to verify. The social-network-small-results.txt and social-network-large-results.txt files on cuLearn include the correct answers so you can verify your solutions. Write two more functions called getmostpopular() and getleastpopular() which return the name of the person who has the most friends and the least friends respectively (or one of the people, in the case of ties). For an added challenge, modify these functions to return the list of the people that are most popular and least popular (i.e., include the ties).

One way that social networks can suggest new friends is based on the number of common friends two people have. The more friends two people have in common, the more likely they are to become friends. Write a new function called recommendfriendfor(name) that takes the name of one person in the network. This function should return the name of the person (or one of, in case of a tie) in the network that has the most friends in common with the person with the input name. Note that it only makes sense to consider others who are not already friends with the person. You can either assume that the name input is valid (i.e., there is a person in the network that has that name), or you can return None to indicate that a person with that name could not be found in the network. If you are struggling with this problem, remember to think of similar problems that we have worked on. We are interested in finding the maximum of something, as we have done many times. How we calculate the 'values' to compare in this case, involves more steps than just reading numbers from a user/file. It might be worth writing a separate function that takes two profiles and returns the number of common friends.

You may also want to write a second function that recommends friends based on the largest number of shared interests. Before you do, can you think of a way to generalize the function(s) you already have so that they can work for common friends or common interests? Depending on how you implemented the previous function, you may be able to add an additional input argument and only slightly modify your function's code.

Finally, write one more function called targetadvertisement(interest), which takes a single string that is one of the interests that people have in the network. This function should return the name of the most popular person (i.e., has most friends) that is also interested in the given interest. Next week, we will improve on this function by considering more complex properties of the network structure.