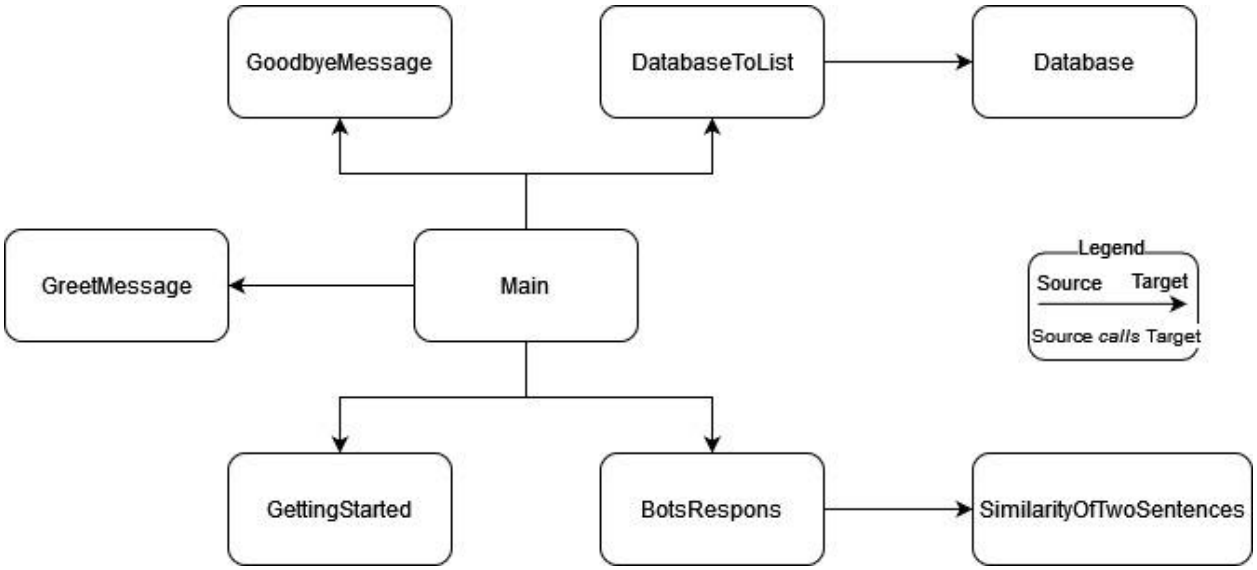
Specifications Increment 1



Stakeholder Requirements and Expectations:

As noted in the outline, this project consists of two parts: the database and software system. The purpose of this increment is to have the software system fully built and tested. Alongside the software, a skeleton database will be required in order to test the software system.

GreetMessage:

GreetMessage is a class containing a single method named greet_message(). greet_message() is a function that takes in no parameters. The function has a list with 4 pre-determined strings, greeting messages, such as "Hello", "Hi!', etc. The function randomly generates a number from 0 to 3, "num" (Use a different name, a more descriptive name). Then, the function returns the string at index "num".

Test:

Create a new class. Import the class GreetMessage, and call the function, greet_message(), 10 times. The function is working correctly if (1) the greeting messages show up correctly, and (2) randomly. Rinse and repeat 3 times. If all three runs work, then the function has passed the test.

GoodbyeMessage:

GoodbyeMessage is a class containing a single method named goodbye_message(). goodbye_message() is a function that takes in no parameters. The function has a list with 4 pre-determined strings, goodbye messages, such as "Goodbye", "See you!', etc. The function

randomly generates a number from 0 to 3, "num" (Use a different name, a more descriptive name). Then, the function returns the string at index "num".

Test:

Create a new class. Import the class GoodbyeMessage, and call the function, goodbye_message(), 10 times. The function is working correctly if (1) the goodbye messages show up correctly, and (2) randomly. Rinse and repeat 3 times. If all three runs work, then the function has passed the test.

GettingStarted:

GettingStarted is a class containing a single method named getting_started(). getting_started() is a function that takes in no parameters. The function has a list with 4 pre-determined strings, getting started messages, such as "How can I help you?", "What would you like to talk about?", etc. The function randomly generates a number from 0 to 3, "num" (Use a different name, a more descriptive name). Then, the function returns the string at index "num".

Test:

Create a new class. Import the class GettingStarted, and call the function, getting_started(), 10 times. The function is working correctly if (1) the getting started messages show up correctly, and (2) randomly. Rinse and repeat 3 times. If all three runs work, then the function has passed the test.

Database:

A .txt file named database.txt, containing paired sentences. A paired sentence is one that has a prompt and a corresponding answer. For example, "I feel lonely" (prompt) and "I am sorry to hear that. Why do you think so?" (answer). The prompt must be greater than or equal to 3 words, and a very simple, subject-predicate sentence. The answer must also be open ended and not assume things about the user. Lastly, the structure of the text file is such that the prompt and answer must be on the same line and be separated by a " @ ". For example: "I feel lonely @ I am sorry to hear that. Why do you think so?" Finally, enter at least 5 paired sentences on the topic of loneliness; with a variety of lengths. However, make sure at least two prompts are of the same length.

Test:

Create a new class. Import the class and in a loop that goes through each line, and storing, temporarily, said line into a variable, current_line. After storing the line in said current_line, use the method .split(" @ ") on current_line. Print said application, i.e., print(current_line.split(" @ ")). After running the loop, scan over the printed lines, and make sure (i) that each "prompt" and

answer are in a size 2 list, (ii) the "prompt" comes before the "answer", (iii) there is no "@", (iv) the "prompt", in the list, does not have a space at the end, (v) the "answer", in the list, does not have a space at the beginning and end (vi) the "prompt" has at least 3 words. If all the sentences have passed the six metrics, then the Database has passed the test.

DatabaseToList:

DatabaseToList is a class containing a single method named database_to_list(). database_to_list() is a function that takes in no parameters. Simply put, database_to_list() goes through each line in the database.txt via a loop, and temporarily storing said line into a variable, call it "line" (Use a different name, a more descriptive name). Afterwards, the function .split(" @ ") is applied to "line". The function returns a list of size 2. Store said list into another list: "master_list" (Use a different name, a more descriptive name). "master_list" must be declared outside the loop. After going through each line, return "master_list". Make sure that file has a relative path, and everything is done in a try-catch that handles OSError.

Test:

Create a new class. Import the DatabaseToList, and call the function, database_to_list(). Make sure you have implemented class Database first. Because it will return a list, print said list. Scan over the printed list, and make sure (i) that each "prompt" and "answer" pair are in a size 2 list, (ii) the "prompt" comes before the "answer", (iii) there is no "@", (iv) the "prompt", in the list, does not have a space at the end, (v) the "answer", in the list, does not have a space at the beginning, (vi) the "prompt" has at least 3 words, (vii) the printed list contains list(s) of size 2. If printed list has passed the seven metrics, then the DatabaseToList has passed the test.

SimilarityOfTwoSentences:

SimilarityOfTwoSentences is a class containing a single method named sentence_similarity(str_1, str_2). sentence_similarity(str_1, str_2) takes in two strings, and compares them. First, apply the method .split to the two inputted strings and store them into parsed_str_1 and parsed_str_2 respectively. Next, declare a variable of type int, call it "num", and set "num" to 0. If parsed_str_1's length is less than or equal to parsed_str_2's length, then run a loop. Said loop will go through each element of parsed_str_1, and compare said element to its corresponding element in parsed_str_2; i.e., compare parsed_str_1[0] to parsed_str_2[0], parsed_str_1[1] to parsed_str_2[1], etc. If the elements match, increase the "num" by 1. When the loop ends, a value will be calculated, it's formula is "num"/length of parsed_str_2, and returned. The names parsed_str_1, str_1, parsed_str_2, str_2, "num" should be changed to more descriptive names.

Test:

Create a new class. Import the class SimilarityOfTwoSentences, and call the function, sentence_similarity(str_1, str_2); where the four cases are tested:

(i) str_1 = "I am happy" and str_2 = "I am happy". Output expected: 1

(ii) str_1 = "I am happy" and str_2 = "I am sad". Output expected: 0.66

(iii) str_1 = "I am quite happy" and str_2 = "I am happy". Output expected: 0

(iv) str_1 = "I'm happy" and str_2 = "I am happy". Output expected: 0

If outputs come out correctly, then the class SimilarityOfTwoSentences has passed the test.

BotsRespons:

BotsRespons is a class containing a single method named bot_respons (str, list). bot_respons (str, input_list) takes in two inputs a string, str and a 2d list, input_list. input_list is assigned to database_list, and a variable, "answer", of string type is declared with the value "". Next a value of type float, named "highest_value", is declared. " highest_value" is assigned 0. Next a loop going through the outer indices of database_list is runed. Inside said loop, a string, named "list_string", is declared and assigned database_list[i][0]. Next the sentence_similarity(str_1, str_2) is called, where str_1 is str and str_2 is "list_string"; because sentence_similarity(str_1, str_2) returns a value, store said value into a variable "score". If the "score" is greater than highest_value, then assign " answer" to database_list[i][1]. Then assign "score" to " highest_value". The loops ends, and return "answer" and " highest_value".

Test:

Create a new class. Import the class BotsRespons, and call the function, bot_respons (str, input_list); where a case is tested:

str = "I don't want friends" input_list = [["I don't want friends", "I think that is quite valid"], ["I want friends", "I think that is valid"], ["Friends are quite fun to have", "I agree"]] Output expected: "I think that is quite valid", and 1

If the output comes out as expected, then BotsRespons has passed the test.

Main:

Main is a class that calls other methods to mimic human understanding. First is calls greet_message(), and the user is asked for an input. The input is considered invalid as long as it is not all letters (print an appropriate output informing the user). Next, getting_started() is called and a variable, "talk", is set to true. Next, database_to_list() is called and stored into "list". Then,

a while loop is runed as long as "talk" is true. Inside the loop, a user is asked for an input, and is considered in invalid as long as it is not all letters (print an appropriate output informing the user).. Next, if the user input, call it "user_input", is less than or equal to 2, then call goodbye_message() and set "talk" to false. Else, bot_respons is called, with the values "user_input" and "list". Because it returns two values, store said values, call them "answer_string" and "point". If "point" is less than or equal to 0, then print out "Sorry, I couldn't understand. Could you say that again? Use a simple sentence". Else print "answer_string".

Test:

First enter an input that does contain a char that is not a letter, expected output should be: an appropriate output informing the user that the input is invalid. Then, input something appropriate. Next, enter an input that does contain a char that is not a letter, expected output should be: print an appropriate output informing the user that the input is invalid. Next, input a sentence that is larger than any "prompt" in the database; the expected output should be: "Sorry, I couldn't understand. Could you say that again? Use simple sentences". Next, input a sentence that is at least 3 words long, but is smaller or equal to the largest "prompt" in the database. The expected output should be an answer that is appropriate. To determine an appropriate answer, apply the method bot_respons (str, list). After doing so, if the output is appropriate and matches the manually calculated answer, then the case is cleared. Next, enter in a sentence that is less, in terms of length, than or equal to 2; expected output is an output from the method goodbye_message() and closers of the program. If the code passed all the above tests, then the code has passed testing.