Chat-Bot README

First Time Use Instructions

Our project relies on certain Python libraries for API calls and other functionality. We have included all the required libraries as a “requirements.txt” file. To use this you must run “pip install -r requirements.txt”. You will also need Python installed to run it.

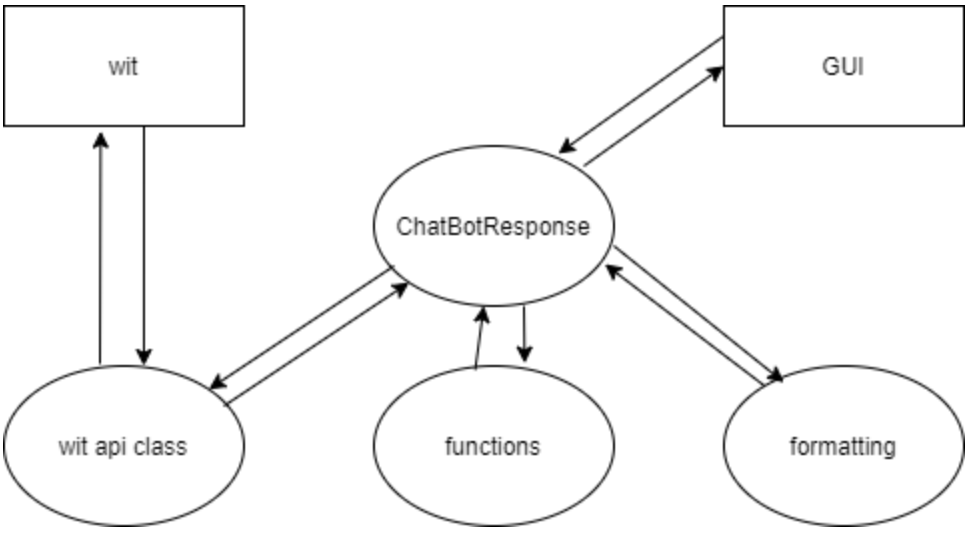
To run the chatbot, download the zip file off of Github and extract the file. Open a console of any sort capable of running pip commands and navigate to the directory “chat-bot”, then run the command “pip install -r requirements.txt”. Once dependencies are downloaded, navigate to chat-bot, and run “main.py”. This will open a GUI where you can talk to the bot.

If you plan on using the new features from the individual assignment, you will need an API key. I have included my API key in the project report that is not getting uploaded to GitHub for security reasons. To use the key, simply open up directionsapi.py, staticmapsapi.py, and streetviewapi.py and replace the line ‘API_KEY = apikey.getKey()’ with ‘API_KEY = “<api_from_project_report>”’.

Project Description

Our team is developing an interactive conversational weather and navigation agent that will utilize various methods and APIs to respond to relevant user questions. The user can ask the bot questions regarding the current and future weather predictions and get brief POI information.The role of this bot would be to answer the frequent questions a user would need to know before leaving their house.The structure of our project is to make use of a third party software (wit.ai) to train and develop an existing AI that will help identify specific traits, entities and intents within the user question; then to use those parameters to query data via weather and geographic data APIs and provide output via a GUI.

Class Organization



Chat-Bot README

Class Responsibility/Description

location.py
Responsible for all location and distance functions.

getLocation(place = None)
Input: Place or location as a string, can be any format (city, country, address, etc)
Output: Tuple containing lat and long coords as float values of the location. If no location is given, it will default to using the user's location. Return type is a tuple of floats.

distanceByLatLong(entities)
Input: List of entities containing 1-2 locations
Output: Distance between the 2 locations in KM, if only 1 location is given, the second location is the location of the user. Return type is float.

timechatbot.py
Responsible for all timezone and time functions.

getTimezone(entities, place = None)
Input: List of entities containing locations, or a location as a string.
Output: If a string is given for 'place', outputs the timezone of that location. Otherwise outputs the timezone of the first location entity in the entity list. Return type is string.

getLocalTime(entities, place = None)
Input: List of entities containing locations, or a location as a string.
Output: If a string is given for 'place', outputs the local time of that location. Otherwise outputs the local time of the first location entity in the entity list. Return type is datetime.

getTimeDifference(entities)
Input: List of entities containing 1-2 locations.
Output: If 2 locations are input, get the difference in the 2 local times. If 1 location is given, it assumes the second location is the user's location. Return type is datetime.

witapi.py
Responsible for making API requests to wit.ai.

Chat-Bot README

sendRequest(question)
Input: String to be sent to the chatbot.
Output: JSON containing entities, intents, and traits from wit.ai. Return type is JSON.

responsechatbot.py
Responsible for handling all function calling based on the response from wit.ai to a question.

getResponse(question)
Input: String to be sent to the chatbot.
Output: String that is the chatbot's response. Return type is string.

responseformat.py
Every function in responseformat.py takes in the return data from their respective function from other classes, as well as all the inputs that were sent to them. They all return a formatted string to make the bot's talking sound better.

Example:
getDistanceFormat(distance, entities)
Input: Distance in KM retrieved from distanceByLatLong in location.py, as well as the list of entities sent to distanceByLatLong in location.py.
Output: String containing a properly worded response from the bot. Return type is string.

geoInfo.py
Responsible for all weather and point of interest data.

get_temperature(entities, lat=None, long=None, wtype='celsius')
Input: List of entities containing locations.
Output: The temperature of all locations in the entity list in celsius. Return type is an array of floats.

get_weather(entities, lat = 0, long = 0)
Input: List of entities containing locations, a json type, or a location as a string.
Output: the weather of that location. Return type is string.

get_point_of_interest(entities, limit=5, latitude = 0, longitude = 0)
Input: List of entities containing locations, a json type, or a location as a string.
Output: the POI of that location. Return type is string.

main.py
Acts as a GUI for the user to enter questions and to view the chatbot's responses.

Chat-Bot README

Assignment 3 Features

For assignment 3, we added synonym recognition, POS tagging, spell checking, and setup sockets to converse with another group's chatbot. Synonym replacement and spell checking is all done in the background, meaning the user will not see their sentence changed. The chatbot will, however, get the version of the question that has corrected spelling and common versions of words for synonyms.

synonym.py
Responsible for recognizing synonyms of commonly used functionalities.

replaceSynonyms(question)
Input: The question being sent to the chatbot as a string.
Output: String containing the same question but with synonyms for the main intents being replaced by their more common version. An example of this would be replacing the word 'conditions' with 'weather', as our chatbot better recognizes a weather request with the word 'weather' being used rather than the synonyms.

posTag_spellCheck.py
Responsible for coming up with POS tags for questions the user enters, as well as interpreting spelling errors for the chatbot. We had a tough time finding a way to implement the POS tags into the conversation flow of the chatbot, but for spell checking the bot now is able to give proper responses even with spelling mistakes.

pos_tagging(sentence)
Input: The question being sent to the chatbot as a string.
Output: Dictionary containing all the words in the sentence, as well as the relevant POS tag for each.

check_spelling_update(sentence)
Input: The question being sent to the chatbot as a string.
Output: The same sentence returned with all spelling errors corrected as best as possible as a string.

outofscoperesponse.py
Responsible for generating responses when the question is out of the chatbot's scope of conversation.

Chat-Bot README

getResponse()
Input: None
Output: String that is the chatbot's response to out of scope questions. Randomly picks between 5.

socketserver.py
Hosts a server that another chatbot can connect to, then initiates a conversation with that bot that lasts 15 turns each. There are no functions to cover here, but it is worth noting that the other chatbot must connect as a client, rather than a host.

Individual Project APIs

For the individual project, I added 3 APIs. The first is Google Directions API, which I incorporated to give directions between 2 specified locations. The second is Google Static Maps API, which I incorporated to allow the user to request a map for a location and receive an image. The third is Google Street View API, which I incorporated to allow the user to request a street view at a given address and receive an image.

directionsapi.py
sendRequest(entities)
Input: List of entities including 1-2 location entities.
Output: Dictionary including all information including directions from point A to point B.

staticmapsapi.py
sendRequest(entities)
Input: List of entities including a location entity.
Output: PNG image of the requested map.

streetviewapi.py
sendRequest(entities)
Input: List of entities including a location entity.
Output: PNG image of the requested street view.