Personal Assistant AI-Mable

NEA PROJECT

Ákos Tibor Hadar [] Guildford County School

Contents

YouTube channel: ALevel_CompSci_NEA_GCS2

1.0 Analysis

1.1 Project overview

My goal for this project is to create a personal assistant AI. By the end, it should serve most of the core functions that other, similar AIs do today (such as Alexa, Cortana or Siri). To name a few, these would include setting a timer or a reminder, looking up articles or videos, playing music or solving arithmetic problems. In addition, these tasks would ideally be done on verbal command. The final program will mostly be written in Java and Python and will use at least one database of sample sentences and words in order to make speech recognition easier.

1.2 Previous Systems

Products such as this already exist. As I've mentioned, Alexa and Cortana are perhaps the most famous examples. However, these systems are obviously not without their flaws. One of the more recognisable 'errors' is mishearing commands. This was a fairly common mistake made by older versions of Alexa and is still around in updated models. Opening a wrong article, playing the wrong song and mishearing the activation command ("Alexa") all fall under the same category of a general miscommunication between the user and the software.

Another small thing which is not necessarily a problem but can definitely be improved on is that these systems usually run continuously. This means that they receive more unnecessary input which increases the likelihood of a command being misinterpreted or the software being activated by accident. This can be solved relatively easily by adding Operational Timetables to the system. These will essentially list the times between which the system is active or inactive. In other words, between what times it should listen for instructions. The user will obviously have authority over these timetables and other attributes of the program as a highly customisable system would be the most optimal.

Cortana does something similar: the user needs to first open the application in order for it to be able to take any input. Simply put, the software doesn't run in the background. However, the user has the option of allowing it to run while other applications are being used. In contrast, Alexa is running continuously and there are very few to no ways of specifying active times for the system.

Another factor in why these programs sometimes mishear instructions is the way they recognise and process speech. AI's understand spoken language is through Natural Language Processing algorithms (NLP algos). This type of software lies in a specific branch of computer science (more specifically, the study of Artificial Intelligence and Machine Learning) which deals with a computer's ability to understand language in a similar way we humans do. NLP algos combine computational linguistics, which is a rule-based model of language, with statistical and machine learning models. These revolutionary techniques allow AIs to process human language and to 'understand' what it means. Email filters are one of the simplest and most popular implementations of these systems.

1.3 End User(s)

This piece of software would be ideal for a wide range of different users from small businesses to regular households. It could be used for scheduling projects or meetings, to keep track of expanses or it could even be asked to sign an employee in once they arrive and sign them out when they leave.

The completed project should have multiple general functions which would make it suitable for anyone to use. However, <u>the main target here would be individuals who work on one computer: specifically, their home computer or laptop</u>. Therefore, this program would be best suited for people who mostly work from home (for example: software engineers, writers or artists).

With this software, the users would have a much comfortable digital work environment. For instance, if the user wants to check their plans for the next day while they're working, instead of stopping and pulling up their calendar, they could simply ask "What are my plans for tomorrow?" and the AI would answer.

1.4 User needs

Of course, when designing software, the most important thing to consider is the needs of the users. I've asked two potential future users of the program about what they would want an AI such as this to be capable of. Here are the tasks proposed by the two real users:

1) *It should be able to tell you the weather forecast*

2) *It should be able to call or text one of your contacts*
3) *It should be possible to ask the AI to play music or videos*
4) *It should be able to tell you the news or to recommend a few articles*
5) *It should be able to open programs on the computer when asked*
6) *It should be able to visit websites*
7) *It should be able to read articles*
8) *It should be able to set timers, reminders and alarms*

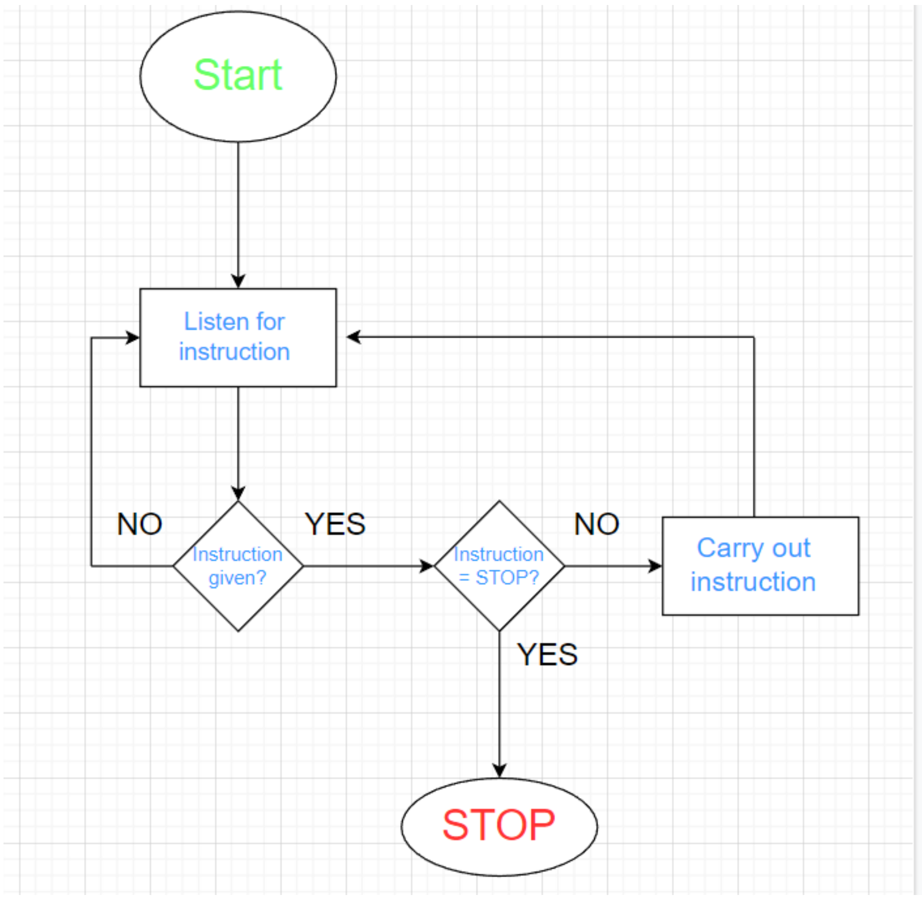
Since this will be running on a PC or laptop, the second requirement will have to be modified. This may be an easy fix: the user will need to link one of their social media accounts, for example Facebook, to the software and then they'll be able to call their contacts from that app through asking the AI to do so.

The users will also benefit from an intuitive User Interface. In it, they could customise the AI however they'd like. For example, it's voice, volume, times of activity or if it should run in the background. For the users described before, this would be the most sensible choice as this way, they could simply speak to the AI the same way they would to any actual assistant.

1.5 Objectives for the Final Version

The main objectives for this project are for the AI to understand spoken language and react to it appropriately and to carry out the built-in tasks when it has been asked to by the user. In addition to these general objectives, the final version of the software should also be able to carry out the tasks given by the two users above.

The program should follow a fairly simple underlying algorithm. It will listen for instructions until it recognises one, then it will need to identify what the instruction was and carry it out. After that, it should loop back to the listening stage and start again. This will need to persist until the instruction is a 'Stop'. Here is a simple model of how the system would function according to this algorithm:



It is clear from the Flowchart that the main loop of the system is the '***Listen - Instruction given? - Instruction = STOP? - Carry out instruction***' loop. We need to separate the 'STOP' instruction form the rest because that is the only one which can break the main loop; meaning, the only one which can shut down the software. The other loop which consists of the '***Listen - Instruction given?***' blocks is responsible for continuously listening until it recognises an instruction. When it does, the software enters the main loop.

Now, this can only be achieved, of course, if the AI first recognises an instruction. This requires it to have the ability to understand spoken language (Objective 1), or at the very least the ability to recognise key words in sentences to deduce what the user is asking. For example, there are many ways you could ask the AI to open your email: "Open up my email", "Show me my emails", "Did I get any new emails?" and the list obviously goes on. However, there is something all these sentences have in common: all of them will mention some combination of the words 'my', 'email' and 'emails'. So, the AI would need to listen for these words only and if it recognises them, it will need to open your email. Of course, there is one problem with this approach: if you say something similar to "What is email?" or "When was email created?", because the AI only listens for the key words, it will interpret both of these queries as you asking it to open your email.

One way to get around this is to establish a keyword hierarchy. This would mean that some words would overrule others. An example would be the words "what", "when", "why", "who" and "where". Since these suggest that the sentence will be a question, it is unlikely that the user is preparing to command the AI to carry out a specific task. So, when the user says "When was email created?" the word "when" will be considered before "email" meaning that even though there is a task associated with the word "email", this will be ignored and the question itself will be in the foreground most probably resulting in a Google search of "When was email created?" (The answer is October 29th, 1969).

It would also be interesting if the AI had the ability to carry a simple conversation with the user instead of just completing built-in tasks. What would be even better is if it could do this in multiple languages. This would make it more accessible and allow for a wider range of users. So, here's a table summarising the Project Objectives:

Main Objectives		**User-suggested Tasks**		**Extra Objectives**	
Understand spoken language and text input	-Tokenization: Breaking down of text into smaller semantic units (words) -Part-of-speech-tagging: Labelling words as nouns, verbs, adjectives… -Stemming and lemmatisation: Standardising words by reducing them to their root forms (e.g.: 'went' becomes 'go') -Stop word removal: Filtering out common words that no unique information (at, to, a, the…)	*It should be able to show the user the weather forecast*	-Search the weather forecast -Tell the user what the program found	*Ability to carry a simple conversation. (no more than a few sentences)*	-Respond to simple sentences such as "What's your name" and "How are you?" -Carry the conversation by responding to the user from a library of sample responses
Graphical User Interface (GUI)	-Speech button: This will activate the main loop of the system -Text area: This is where	*It should be able to call/text any of the user's contacts*	-Search for mentioned name in known contacts	*Voice Recognition*	

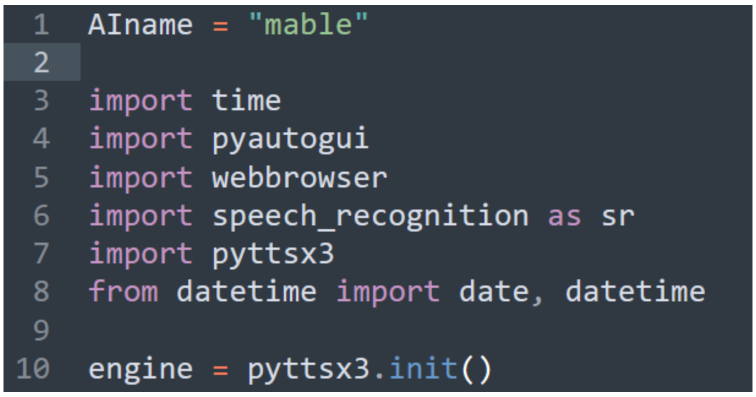
	instructions and responses will be displayed -Settings menu: volume, voice, (language?), active times settings (the times between which the program should run in the background)		-Call the desired contact when found		
Carry out specific tasks when asked by the user	-Recognize instruction -Carry out instruction	*It should be able to play music/videos*	-Open wanted app/site (Spotify, YouTube)	*Multiple languages*	-The user should be able to pick a language in settings and the AI should then switch to that language
Keyword Hierarchy	-Database with keywords linked to the system -Keyword importance (level in the hierarchy)	*It should be able to show you the news*	-Open news sites. (BBC News, CNN…) -Recommend Breaking News		
		It should be able to open programs on your computer	-Search for desired program on the computer -Run the program		
		It should be able to visit websites	-Search the website name in Google		
		It should be able to read articles	-Convert text in open article using a Text-to-Speech algorithm -Play the converted file		

		It should be able to set timers, reminders and alarms	-Open timers on the computer -Add a new timer to the app based on the duration specified by the user -Start the new timer		

1.6 'Mable' Prototype

To help gain a better understanding of how the project should look like by the end, I have prototyped it and implemented some basic tasks for the AI to carry out. For this, I decided to use Python simply because a prototype doesn't need as much detail as the final project and it's much easier than Java meaning I was able to finish faster. However, for the actual software, I plan to use Java among a mixture of other languages.

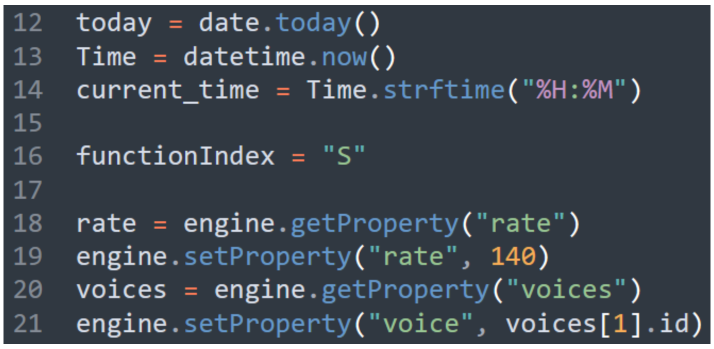
First of all, I imported a few libraries which will be needed in the future (next page):



```
AIname = "mable"

import time
import pyautogui
import webbrowser
import speech_recognition as sr
import pyttsx3
from datetime import date, datetime

engine = pyttsx3.init()
```

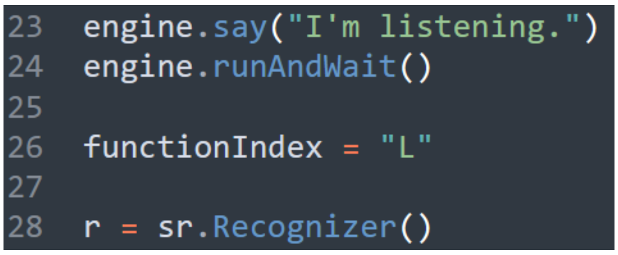
I also went ahead and named the AI and initialised the **pyttsx3** engine which is Python's newest Text-to-Speech engine. While most of the libraries' names are self-explanatory, the **pyautogui** library can't be said to be the same: this library is needed to imitate keyboard presses on the computer; this will be important later.

Next, I defined a few variables for the **datetime** library and the **pyttsx3** engine. More specifically, I set variables for today's date, the current time, the **pyttsx3** engine's speech-rate (meaning how fast the engine will talk) and its voice. I also introduced the 'functionIndex' variable and set it to 'S' (standing for 'Speech'). This variable is needed to switch between the 'Speech' state and the 'Listen' state: if the AI is speaking and listening at the same time, it will interpret its own speech as a user command; the two separate states help avoid this problem.



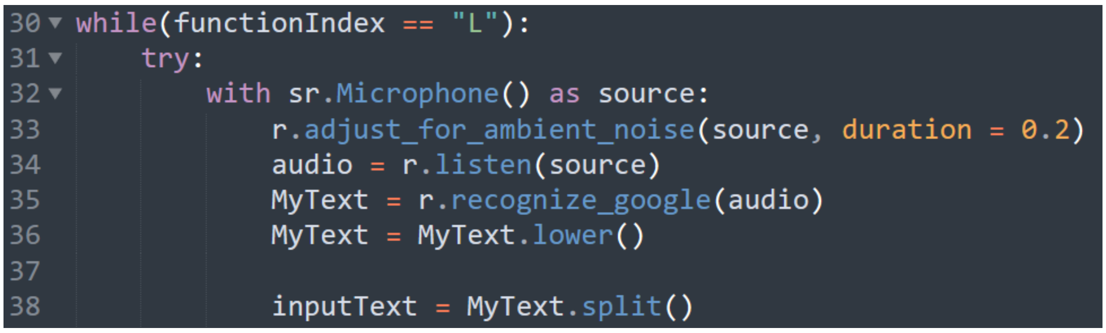
```python
today = date.today()
Time = datetime.now()
current_time = Time.strftime("%H:%M")

functionIndex = "S"

rate = engine.getProperty("rate")
engine.setProperty("rate", 140)
voices = engine.getProperty("voices")
engine.setProperty("voice", voices[1].id)
```

Now that everything is set up, we can begin coding for the main loop described in the previous flowchart. First, I added two lines of code which make the AI say 'I'm listening'. I've done this so that I know when the software restarts as each time it does, it will have to say the same thing before entering the main loop. Then, I changed the functionIndex to 'L', entering the 'Listening' state and I also set the **speech_recognition** engine's 'Recognizer' function to 'r' for simplicity. This function is responsible for recognising human language and letting the software know that it will need to be converted to text.



```python
engine.say("I'm listening.")
engine.runAndWait()

functionIndex = "L"

r = sr.Recognizer()
```

If we take a look at the flowchart of the system, we can see that the main loop starts with the '**Listen for Instruction**' block. So, we'll need to code for this first. The code below starts the '***Listen - Instruction given?***' loop which is also the beginning of the main loop:

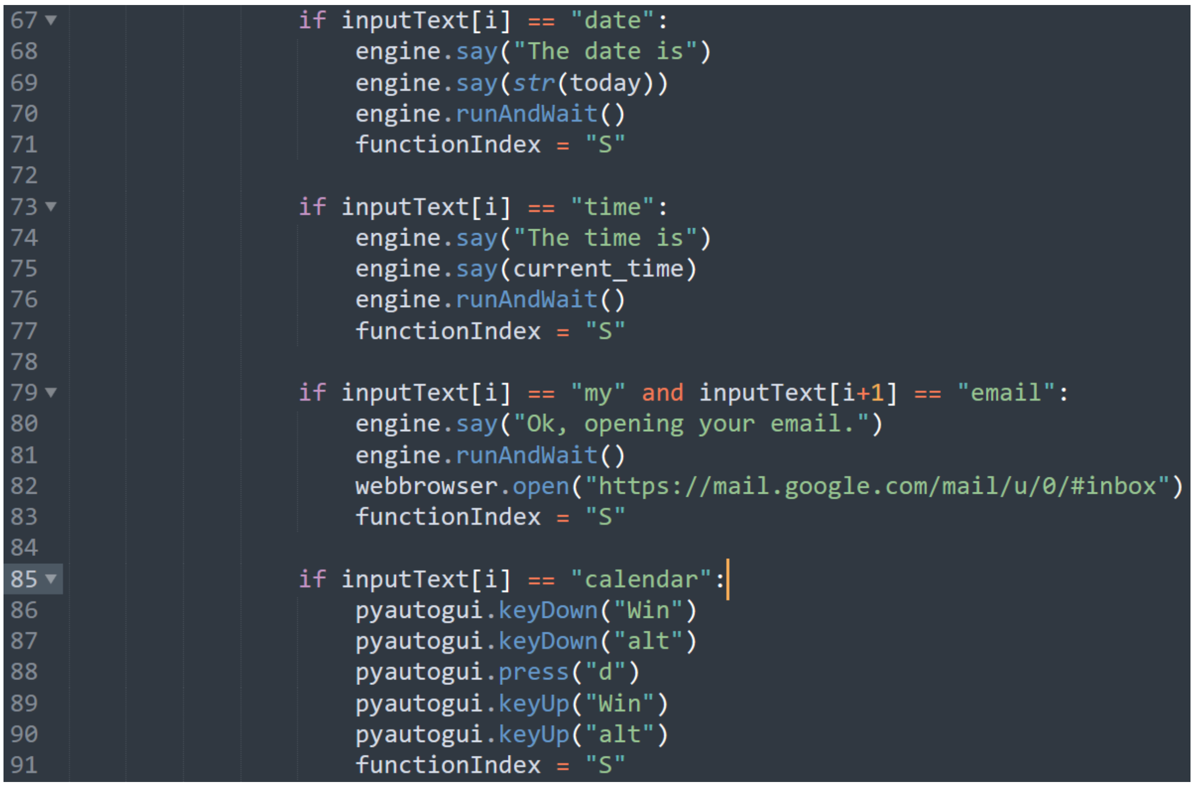


```python
30 while(functionIndex == "L"):
31     try:
32         with sr.Microphone() as source:
33             r.adjust_for_ambient_noise(source, duration = 0.2)
34             audio = r.listen(source)
35             MyText = r.recognize_google(audio)
36             MyText = MyText.lower()
37 
38             inputText = MyText.split()
```

In line 30, we start a 'while' loop meaning that the code inside the loop will keep executing while a certain criteria is met. In this case, since the software is listening, the loop will only break if the functionIndex doesn't equal 'L' (Listen). Inside the loop, the first line (line 32) sets the **speech_recognition** engine's 'Microphone' function to 'source'. Next, the AI begins adjusting to the ambient noise for 0.2 seconds so that it can separate human speech from other noises in the future (line 33). Then, it starts listening for speech in line 34 using the 'Microphone' function as the 'source'. The next line sends the audio to Google which converts it to text and sends it back to the

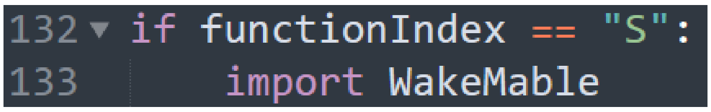
AI. The AI then converts the text into lower case letters in order to simplify the input in line 36 and creates a 1-dimensional array of the words in the received text. This array is named ‘inputText’ and will later be used to pick out keywords to understand the instructions given by the user.

Once one or multiple keywords have been recognised, the AI can jump to a variety of built-in tasks. Here are a few examples:



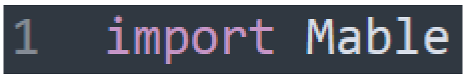
```
67                  if inputText[i] == "date":
68                      engine.say("The date is")
69                      engine.say(str(today))
70                      engine.runAndWait()
71                      functionIndex = "S"
72
73                  if inputText[i] == "time":
74                      engine.say("The time is")
75                      engine.say(current_time)
76                      engine.runAndWait()
77                      functionIndex = "S"
78
79                  if inputText[i] == "my" and inputText[i+1] == "email":
80                      engine.say("Ok, opening your email.")
81                      engine.runAndWait()
82                      webbrowser.open("https://mail.google.com/mail/u/0/#inbox")
83                      functionIndex = "S"
84
85                  if inputText[i] == "calendar":
86                      pyautogui.keyDown("Win")
87                      pyautogui.keyDown("alt")
88                      pyautogui.press("d")
89                      pyautogui.keyUp("Win")
90                      pyautogui.keyUp("alt")
91                      functionIndex = "S"
```

As you can see, keywords have specific tasks associated with them. For example, the keyword ’calendar’ (line 85 - 91) requires the use of the **pyautogui** engine to imitate the shortcut of the computer calendar while the ‘date‘ and ‘time‘ keywords use previously defined variables to give you the current time and/or date. If you look at the code closer, you’ll notice that each ’if’ statement ends with setting the functionIndex to ’S’ (Speech). This is needed to exit the ’while’ loop and restart the software so that a new instruction can be processed. The way the software restarts is by calling a different Python file called ’WakeMable’ which imports the ‘Mable‘ Python file (this is the main AI) again.



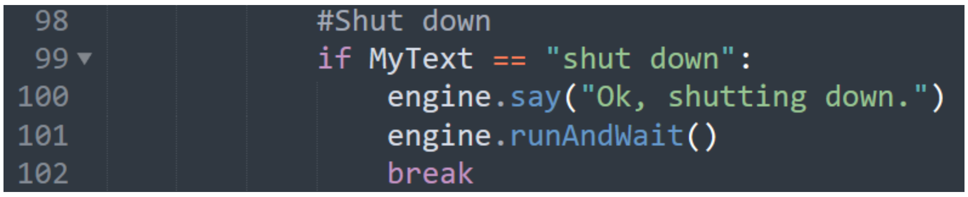
```
if functionIndex == "S":
    import WakeMable
```

This is the only line of code in the ‘WakeMable’ file:



```
import Mable
```

Finally, as shown before in case a ‘STOP’ is called on the program, we need to shut down the software. This can be done with a simple ‘break’ command:



```
 98             #Shut down
 99             if MyText == "shut down":
100                 engine.say("Ok, shutting down.")
101                 engine.runAndWait()
102                 break
```

Line 102 contains the ‘break’ command which will only execute if the user’s input is “Shut Down”. This could be improved by adding more activation commands (for example: “Close”, “Stop”, “Sleep”, “Terminate” ...) and scanning for all of them.

2.0 Design

2.1 Section Overview

In this section, we'll take a more in-depth look into how specific parts of the system will work and how they'll interlink to create the full assistant. First, we'll explore the inner workings of the main loop depicted by the flowchart from the previous section. This will be split into explaining how the AI understands spoken language, how it translates the user input into a simple command, and how the given instruction is carried out.

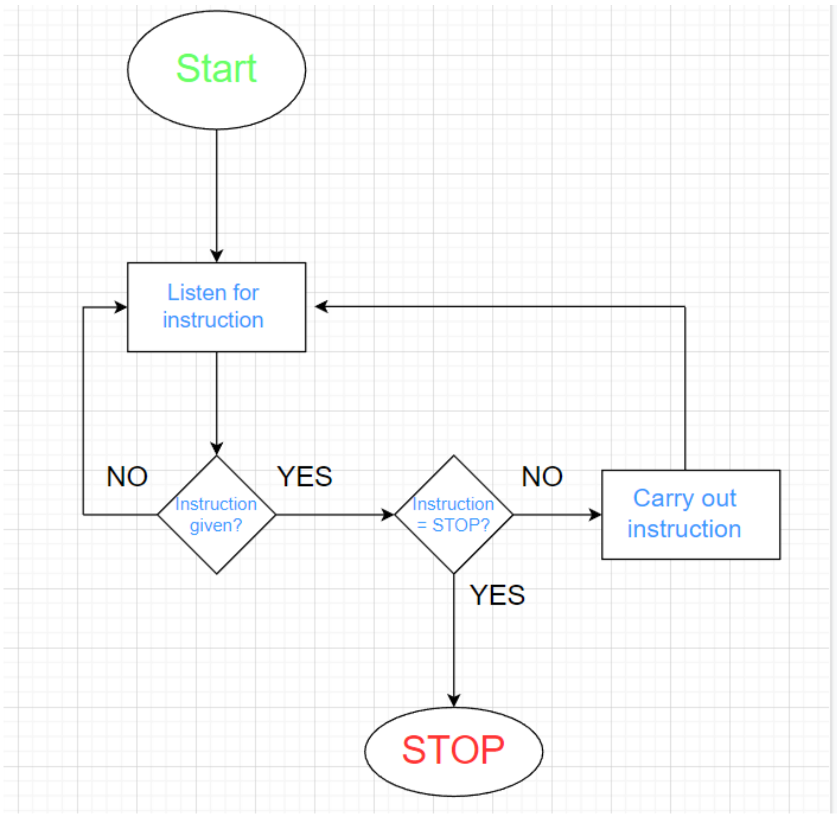
Then, we'll start looking at optimal designs for the User Interface. By the end of this stage, we should get a full, clear picture of how the UI will look like and how it will work. At the very least, we'll need a text box for user input along with an option to change the input format from text to sound so that the user can give verbal commands plus a text area to output the assistant's responses.

After that, the user-suggested tasks will be described in more detail. Specifically, how they'll be carried out step-by-step.

2.2 Understanding the Main Loop

This perhaps is the most important part of the entire project. The main loop is essentially the brain of the assistant: it's where instructions given by the user are interpreted and where they'll ultimately come to be carried out. It's crucial to understand how this part of the project will work in order to successfully build a well-functioning system.

Here, you can see the flow diagram of the entire system which was introduced in the previous section:



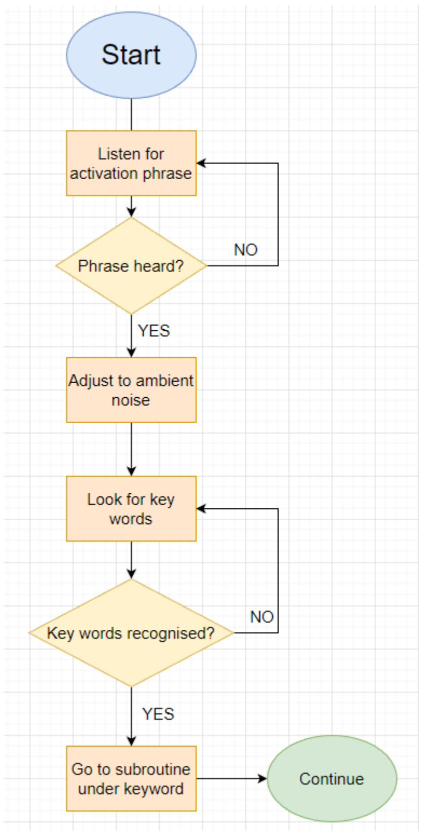
The explanation of the overall working of the system was given in the Analysis section; this section will be dedicated to explaining specific parts of this main loop in more detail. We'll go in order from the beginning of the process and work our way through the different subroutines in the system.

Firstly, the **'Listen for instruction'** block. Just like a student in a lesson, the assistant will first listen for an instruction and once it's confident it knows what it has been asked to do, it will carry out the instruction. But how does this work?

Well, we begin by giving the AI specific activation keyword or phrase which will start the listening stage. for example, for Siri this is the phrase 'Hey, Siri'. After the assistant receives the activation phrase, it will first take a set amount of time (set by us in the source code; for the 'Mable' prototype, I chose 0.2 seconds) to adjust to ambient noise. This is excess input which helps differentiate between background noise and an instruction given in English. We also give it a list of instructions/key words to listen for as described before. For example, let's consider a simpler version of our project where the AI only listens for the instructions 'Search', 'Play' and 'Stop' where 'Search' means a Google search, 'Play' means playing a song from Spotify and 'Stop' terminates the program. Then, it will start listening for the three key words we have given it, again for some set time. This is part of language understanding which will be managed by a smaller NLP system within the larger system of the AI. We'll cover the workings of an NLP system after this.

Once the program recognises a key word or instruction, it will look for the subroutine associated with that specific input which it has just received. For example, if I say, 'Search how are computers made', the AI will recognise the key word 'Search' and it will go to the subroutine defined under the 'Search' key word. If it doesn't recognise any keywords, it will give the user an error message and loop back to the start of the listen stage. The error message will just let the user know that no keywords have been recognised. (Example: "Sorry, I didn't understand")

Here's a flowchart depicting this process:



2.2.1 NLP Systems

Natural Language Processing (NLP) has existed for roughly 50 years. Its roots lay in linguistics, more specifically, linguistic analysis. Linguistic analysis is the study of the rules of language and is extremely useful for programmers who are trying build a program which needs to recognise speech and/or text. It allows the developer to build a rule-based model of language which then can be implemented in their program. Of course, language is already based on a rather extensive set of rules (grammar) which govern how we communicate. However, for a computer, we need a slightly different, abstracted version of this rule set.

An NLP system goes through four main stages of input abstraction which allow it to understand language. This process simplifies the raw input, breaking it down into the most fundamental form so that only the most basic and important information is processed as language. These four steps are **Tokenisation**, **Part-of-speech tagging**, **Stemming & Lemmatisation** and **Stop-Word removal**.

Let's see what each of these stages do in detail:

Tokenisation: this stage is the breaking down of the input into smaller lexical items such as words or clauses. Breaking the input down into words is, of course, more useful. In many programming languages, this is actually a built-in function; for example, in python, this is the **'.split()'** function. The tokenisation stage is important as the AI needs to analyse every word in the sentence one-by-one in order to discern their meaning. If we give the input "What is the weather like outside?", the tokenisation function will split this into words and store them in a 1D array in the order as they appeared in the sentence: ['What', 'is', 'the', 'weather', 'like', 'outside?'].

Part-of-speech tagging: here, the program will label each word obtained from the first step as nouns, adjectives, verbs… This is done so that the AI can determine the type of instruction it has received from the user. For example, if there is an asking word in the sentence such as 'what', 'why' or ‘when’, it can assume that it needs to search the question in google (or some other search engine) and return the answer from the site. If, however, it finds a verb like ‘play’ or ‘write’, it will know to execute one of the other built-in instructions which don’t involve searching any information online. The way this could be implemented is with the use of a database labelling a library of words: one column for the words, another for the label:

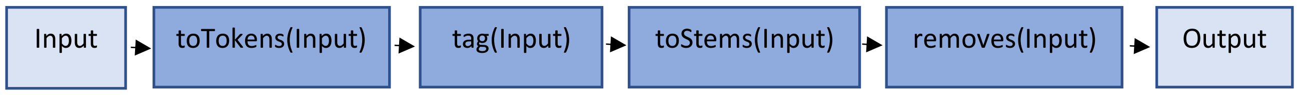
Index	Word	Label
1	house	noun
2	car	noun
3	run	verb
4	nice	adjective
5	swim	verb
…..	…..	…..

Or, to use three different databases, separating the word types from each other. Furthermore, we could use a hash table to store the words. This would provide each word with its own, unique index algorithmically generated from the words themselves. All we need to do is create a hash algo to create these indexes. In order to do this, we need to create a mathematical method (a hash function) using the individual alphabetical indexes of letters to calculate an ‘ID’, if you will, for every word. We will talk about a possible function later in this section.

Stemming & Lemmatisation: after the words have been tagged, we need to stem them. Stemming a word means simplifying it into the word stem. For example, the word ‘running’ has the ‘run’ word stem, the word ‘streams’ has the ‘stream’ word stem, the word ‘playing’ has the ‘play’ word stem and so on. This allows the AI to process much simpler inputs, speeding up the processing time of each separate instruction.

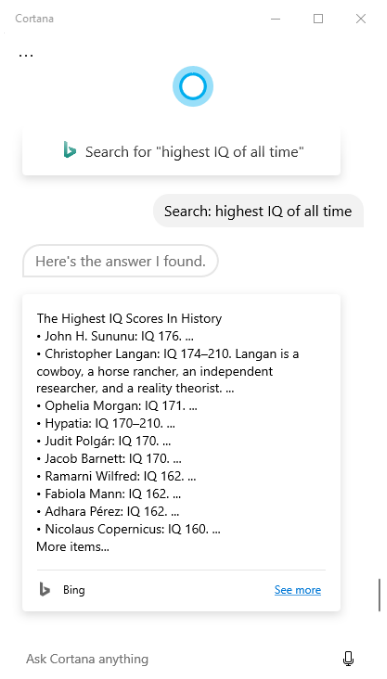
Stop-Word removal: this stage of the process is the final level of input abstraction. Here, we remove any ‘filler’ words which contribute nothing or very little to the overall meaning of the input sentence. These include ‘and’, ‘to’, ‘is’, ‘a’, ‘at’ and many more. Any connecting words such as the ones listed are essentially useless to the AI as they don’t carry any unique meaning and therefore don’t add to the instruction. Simply speaking, any word that wasn’t marked in the tagging stage would be discarded. If we look at the example sentence from before, we can easily see how this would work in practice: we have the array of words [‘What’, ’is’, ‘the’, ‘weather’, ‘like’, ‘outside?’]; from this, the words ‘is’, ‘the’ and ‘like’ would be removed, leaving the array as [‘What’, ‘weather’, ‘outside?’]. As you can see, this is much simpler to analyse and understand for a computational system since it’s now completely abstracted.

After all this is done, the program can search for key words in the input to find the appropriate instruction to carry out. Below is a simple representation of how the input data will pass from one function to the next, resulting in an abstracted output:



2.3 The UI

For the User Interface, I'm taking most of the inspiration from Cortana's UI which looks like this:

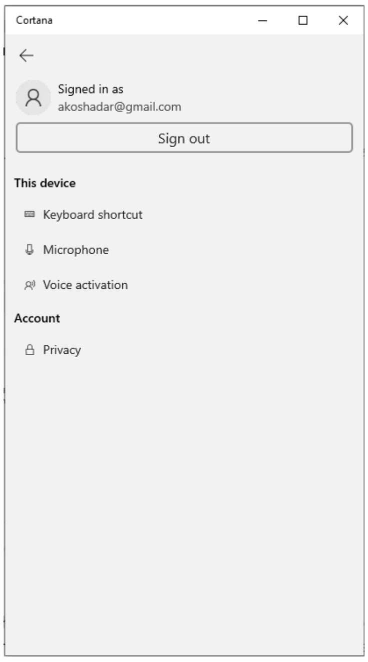


Search: highest IQ of all time

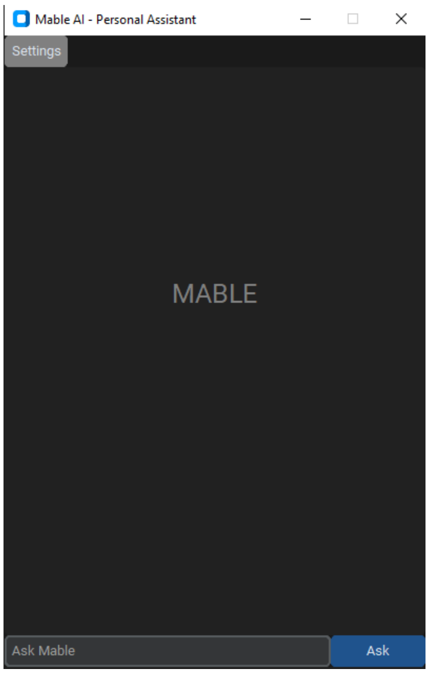
Here's the answer I found.

The Highest IQ Scores In History
• John H. Sununu: IQ 176. ...
• Christopher Langan: IQ 174–210. Langan is a cowboy, a horse rancher, an independent researcher, and a reality theorist. ...
• Ophelia Morgan: IQ 171. ...
• Hypatia: IQ 170–210. ...
• Judit Polgár: IQ 170. ...
• Jacob Barnett: IQ 170. ...
• Ramarni Wilfred: IQ 162. ...
• Fabiola Mann: IQ 162. ...
• Adhara Pérez: IQ 162. ...
• Nicolaus Copernicus: IQ 160. ...
More items...

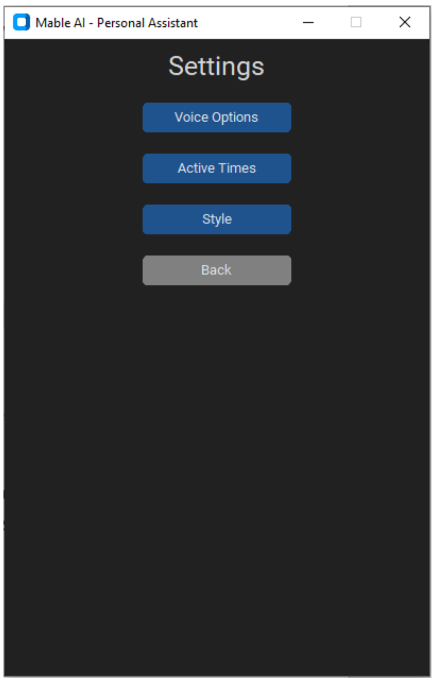
We have a text box where we can type our questions/commands for the AI. Next to the text box, we also have a microphone icon for verbal commands which is what I'm planning to implement as a secondary form of communication mainly used when the program is told to listen in the background. Above this, we have the text area where our questions and Cortana's answers are displayed just like in a regular online conversation. Above that, we have our settings icon (the three dots in the top left corner) which allows us to customise the assistant's properties (see below). So, our AI's UI will look similar to this.



On the next page is an example of how the final UI might look like.

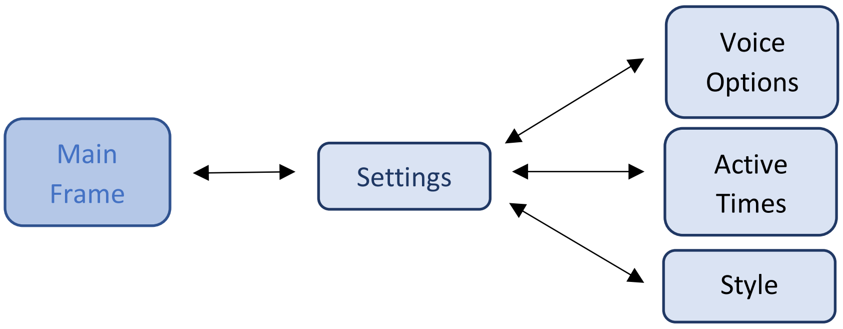


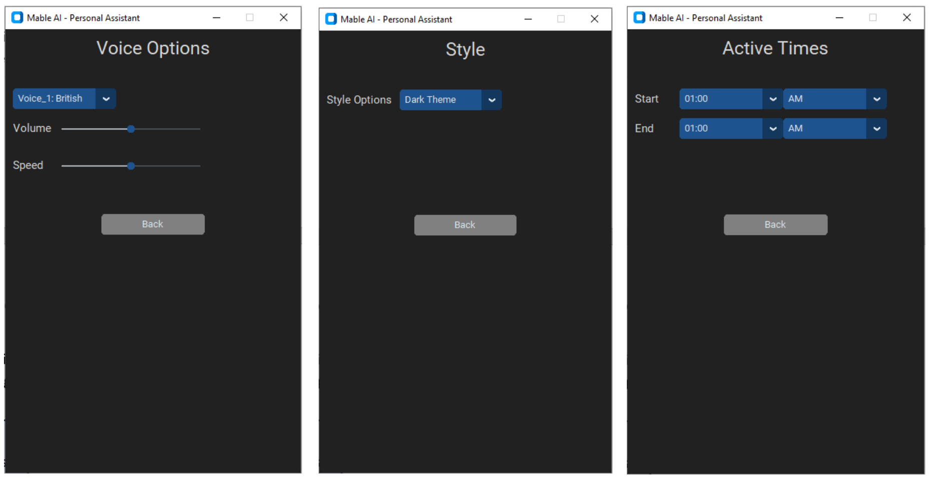
The 'Ask' button will be the voice command option. By clicking it, the user will start the analysis of their input from the text box. The Settings UI can be seen below:



The 'Voice Options' menu will provide settings such as volume, voice character (which voice the AI should use) and possibly language. In the 'Active Times' menu, the user will be able to set specific times between which the assistant should be active or off. Finally, the 'Style' menu will set the theme of the UI: Light or Dark.

We will switch between the different windows by utilising frames in the python **customtkinter** module. Each frame will replace the previous one when prompted. They will be joined together like this:





Here, you can see the three sub-windows of the Settings menu.

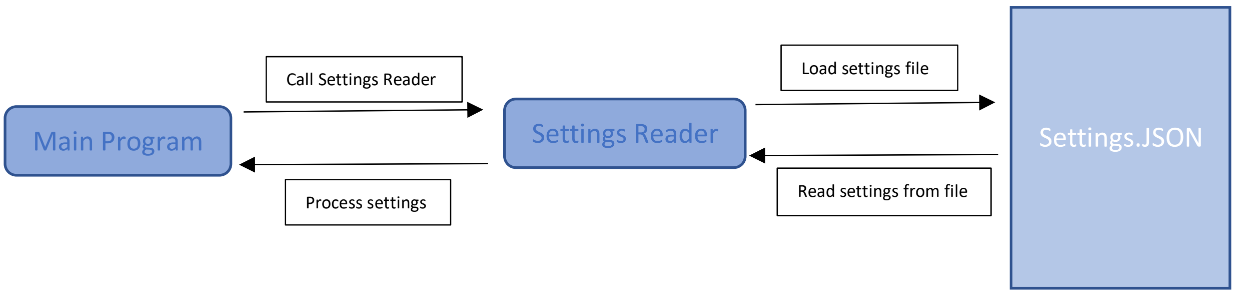
The source code for the UI will be included in the next section.

2.4 Save System

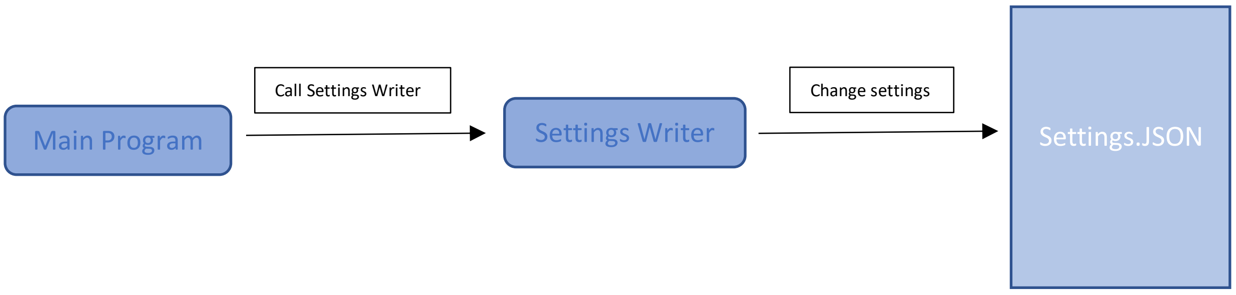
Now, we need some way to store the settings specified by the user in a file so that they can be loaded each time the program starts running. I think the easiest way to do this is to create a JSON (JavaScript Object Notation) file which stores information about the current state of the program. Every time a setting is changed by the user, we will run a dedicated Settings Writer file as a settings-writer class which will open the JSON file to save the changes made. Similarly, when starting up the program, it will first run a Settings-Reader file which will act as a settings-loader class inside the project. This will open the settings file (named Settings.JSON) and read the settings from the file, sending them to the main program. Then, this information will be loaded into the system, recovering the previously saved settings.

Take a look at the following diagram for this system:

On Start:

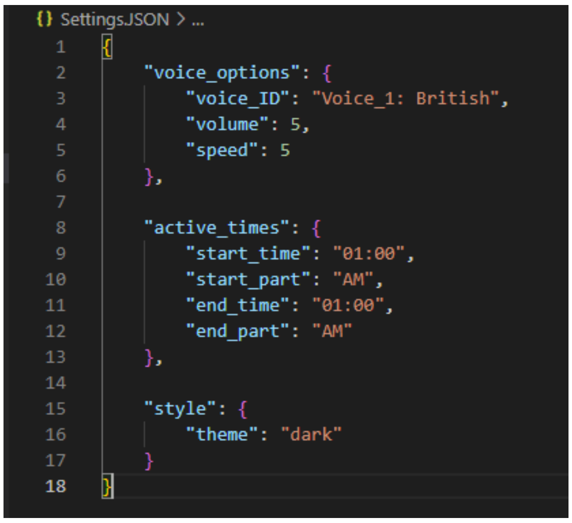


Changing Settings:



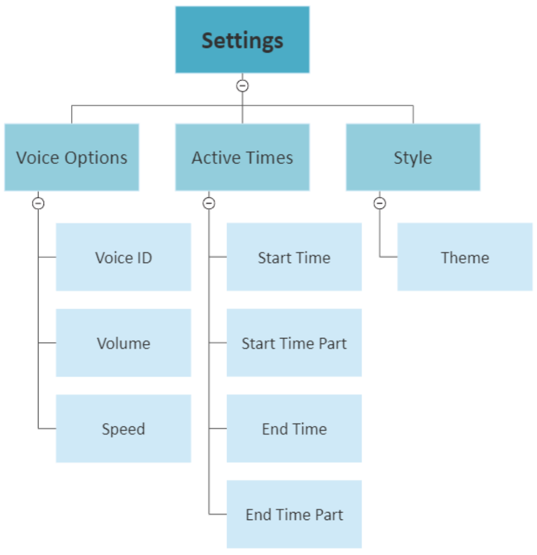
A similar way of doing this is by using a simple text file, however, a JSON file is much more practical as it can be formatted with ease.

This is a basic version of how the Settings.JSON file might look like:



```
{} Settings.JSON > ...
{
    "voice_options": {
        "voice_ID": "Voice_1: British",
        "volume": 5,
        "speed": 5
    },

    "active_times": {
        "start_time": "01:00",
        "start_part": "AM",
        "end_time": "01:00",
        "end_part": "AM"
    },

    "style": {
        "theme": "dark"
    }
}
```

You can see the how the file is much easier to understand than a simple text file as we can create a system hierarchy with the different areas of settings being neatly separated. Furthermore, we can represent this using a Hierarchy chart:



As we are treating the Settings Reader/Settings Writer (SR/SW) files as two separate classes within the project we can create class diagrams in order to better understand them.

Settings Reader	Settings Writer
voice_profile: String volume: int speed: int start: String spart: String	voice_profile: String volume: int speed: int start: String spart: String

end: String epart: String theme: String	end: String epart: String theme: String
loadSettings(): String	writeToSettings()

One thing that may not be obvious from this is the meaning of the 'spart' and 'epart' variables. They simply specify what part of the day we want to set: AM or PM.

2.5 The Hashing function

This part of the Design will be heavily maths-based. As we've said before, we need to create a function that generates a unique ID for each word we use in the AI's library. Now, creating such a function isn't that easy because of collisions. Collisions in a Hash table mean that more than one piece of data occupy the same index. This is obviously a problem since the goal is to create a completely *unique* index for each word. There are a few popular solutions to resolving collisions. One is to simply move a piece of data to the next available spot upon detecting a collision. For example, if we use a function 'H' which returns the same index (say 256) for the two words 'hello' and 'table', we can take one of the words and move it down the table to the next available spot (for example 278). Another way is to create linked lists. Linked lists are lists of data in one index. For example, the list ['hello', 'table', 'boat', 'far'] can be stored in the same index 256 if the function H gave that index as the output for each one of the items in the list. Both of these methods work great, but if we are honest, they are slightly inelegant. So, let's try and create a function that creates a completely unique index for a given piece of data.

Now, the data type that we need to work with is a String since we're creating this function for storing words. It will help to think of words as simply combinations of letters. Let's start by considering a combination with n number of letters in it. For an n letter long combination, we have 26^n possible combinations. If we imagine that word as a pad lock where each cell has 26 options (one letter of the alphabet for each), we can see how this is true: for each cell, we have 26 possibilities; so, for a choice of letter in cell 1, we have the same number of options in cell 2, cell 3, cell 4 and so on all the way to cell n. Of course, some of these combinations will result in something that isn't a word (like 'sdycw') but that's not a problem (we will see why not in a bit).

We can assign an index to each one of these combinations by considering the following method: we will start with n numbers of the letter 'a'. For now, let's say n = 3 and we can generalise later. So, we start with the string 'aaa'.

a	a	a

We can systematically go through all 3-letter combinations to 'zzz' by cycling through each letter in each cell and turning cell n-1 once when cell n has completed one full cycle (from 'a' to 'z').

Example:

a	a	a
a	a	b
a	a	c
a	a	d
…	…	…
a	b	a

a	b	b
…	…	…
z	z	z

Like this, we can get all three letter words such as ‘and’, ‘owl’, ‘bed’, ‘bad’ and so on. Not only that, because we are following a simple logical process to get to the next combo, each word will automatically have an index. However, as you may have noticed, we get some excess terms like ‘aaa’, ‘agr’, ‘tqo’ and more which are not words. But we can mathematically work out the index of each combination if we take a closer look at the mechanics of this algorithm.

First, let’s switch from letters to numbers to better understand how this works (a = 1, b = 2 …). The combinations 111 will be the first combination, 112 will be the second, 113 the third… Eventually, we get to 121. At this combination, we have completed one full cycle in cell 3. We can see this since cell 2 turned once. So, we can immediately see a very basic law that this method obeys: cell n has made k-1 full cycles where k is the index on cell n-1 i.e.: when cell 1 is at 5, cell 2 has made 4 full cycles. Also, when we have made x number of turns on a cell, we are at index x+1. So, after 15 turns on a cell, we are at index 16. This is true since we start on index 1 at turn 0. The final index of the combination, therefore, will be equal to the number of total turns that cell n (in this case cell 3) has taken plus 1 since we start at index 1:

$$I = T + 1$$

where I is the final index and T is the number of turns.

So, we need to find a formula for T. We can see that for each 26 turns in cell 3, we have one turn in cell 2. Similarly, for each 26^2 turns in cell 3, we have one turn in cell 1. So, if we generalise this, we can say that for each turn in cell n, we have 26^{n-1} turns in cell 1. So, if we multiply each of these terms by the number of turns in each single cell and take their sums, we get the total number of turns in the first cell:

$$T = \sum_{k=1}^{n} 26^{k-1} T_k$$

where T is the total number of turns in cell 1 and T_k is the total number of turns in the kth cell.

As we’ve said before, the total number of turns in a cell is always one less than its index, so, we can rewrite this equation like so:

$$T = \sum_{k=1}^{n} 26^{k-1} (I_k - 1)$$

where I_k is the index of the kth cell.

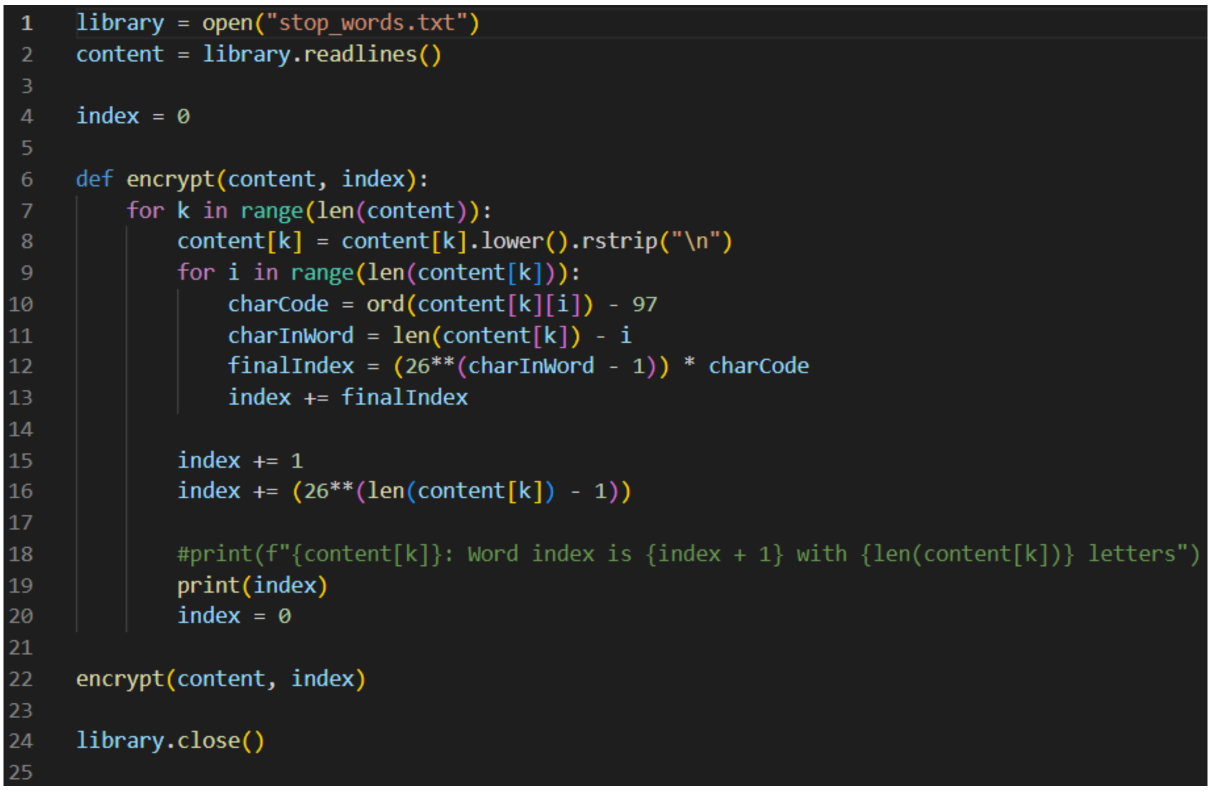
Therefore, our final equation for the index of a combination is as follows:

$$I = 1 + \sum_{k=1}^{n} 26^{k-1} (I_k - 1)$$

This gives us the index of any combination of n number of letters. But, obviously, the 100th combination of a 3 letter word is not the same the 100th of a 5 letter word. So, we need a way to distinguish between the lengths. A simple solution is to add a single digit at the end or beginning of an index and treating it as a marker for the word length. For example, if the first digit of the index is the marker, then the index 356 would indicate the 56th 3 letter word and the index 256 would indicate the 56th 2 letter word. Or, to add the index of the last word from the previous length group, offsetting the next set by the exact index number by which it follows the previous one.

We can test the function using a sample word. Let's say, 'get'. First, we need to convert it to the letter indexes: 7,5,20. Then, we get the subsequent numbers of turns from them by taking one away from each index: 6,4,19. Now, we can multiply each term by their corresponding coefficients: 6*26^2, 4*26, 19*1. This gives us the numbers 4056, 104 and 19. If we add them up, we get 4179. Finally, we add 1 since we start at index one to 4180. And thus, we have that the index of 'get' is 4180. This will be completely unique to this word, meaning that no collisions will occur.

The code for this can be seen below applied to a text file:



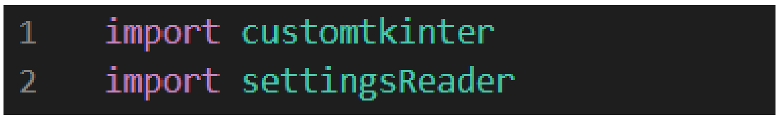
```python
library = open("stop_words.txt")
content = library.readlines()

index = 0

def encrypt(content, index):
    for k in range(len(content)):
        content[k] = content[k].lower().rstrip("\n")
        for i in range(len(content[k])):
            charCode = ord(content[k][i]) - 97
            charInWord = len(content[k]) - i
            finalIndex = (26**(charInWord - 1)) * charCode
            index += finalIndex

        index += 1
        index += (26**(len(content[k]) - 1))

        #print(f"{content[k]}: Word index is {index + 1} with {len(content[k])} letters")
        print(index)
        index = 0

encrypt(content, index)

library.close()
```

3.0 Technical Solution

3.1 Startup and the GUI

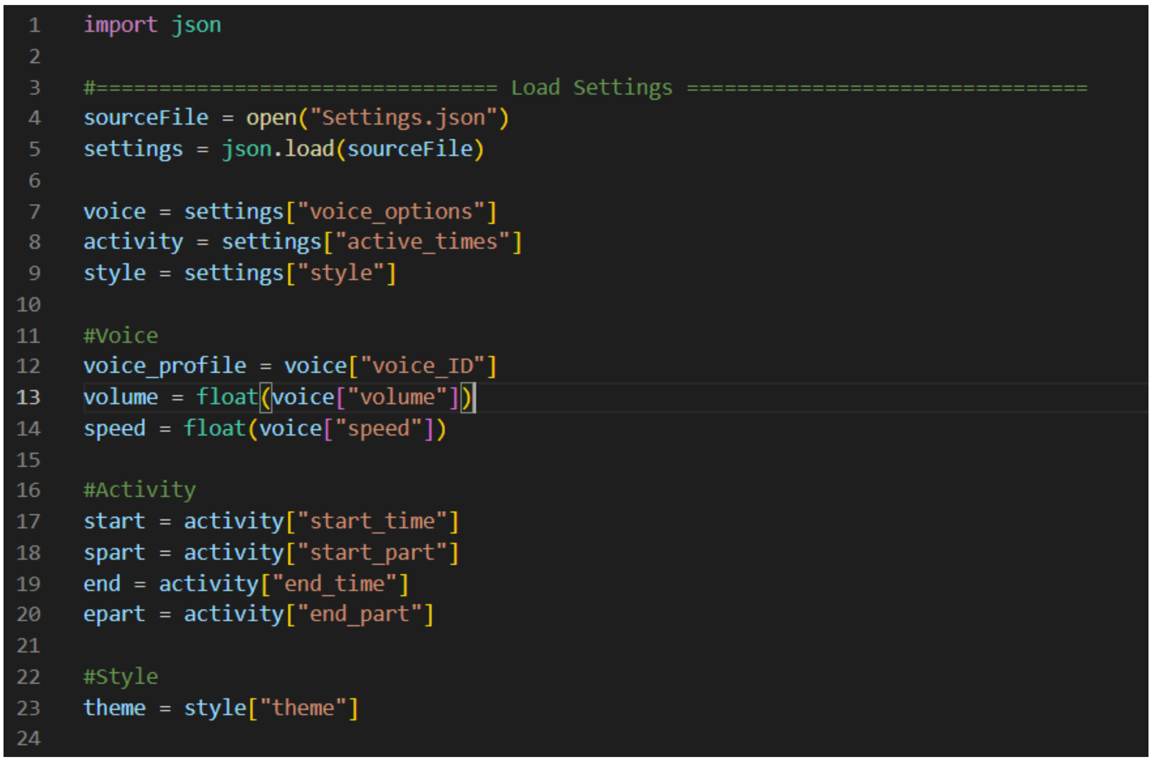
On start, the program will do two main things: load the settings and build the UI according to those settings. It will also import every built-in and every custom module.

First, it will import the costumtkinter and settingsReader modules. The first module is needed to build the GUI and the second is a custom module I made which will load the settings:



```
import customtkinter
import settingsReader
```

Here is how the settingsReader module works:

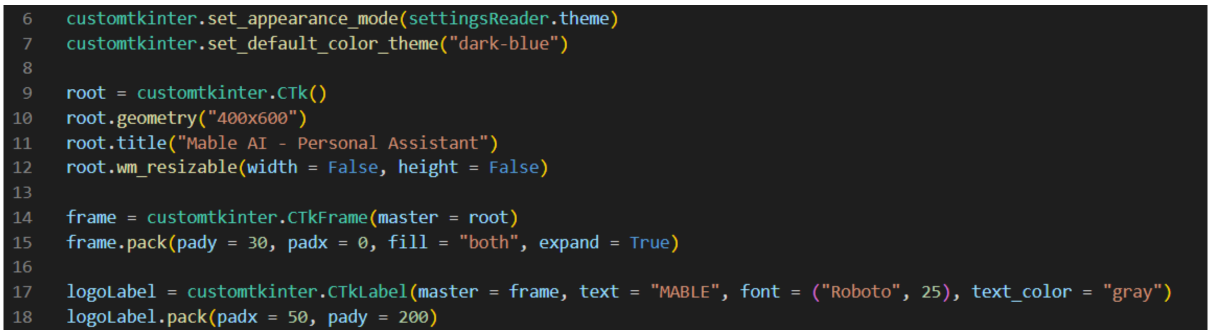


```
import json

#=============================== Load Settings ===============================
sourceFile = open("Settings.json")
settings = json.load(sourceFile)

voice = settings["voice_options"]
activity = settings["active_times"]
style = settings["style"]

#Voice
voice_profile = voice["voice_ID"]
volume = float(voice["volume"])
speed = float(voice["speed"])

#Activity
start = activity["start_time"]
spart = activity["start_part"]
end = activity["end_time"]
epart = activity["end_part"]

#Style
theme = style["theme"]

```

It first imports the json module which is needed for .json file handling and then it copies (loads) all the data from the Settings.json file which stores the previously saved settings categorically.

These pieces of data will then be fed to the main program to build the UI and restore the user's preferred settings.

Now, using the customtkinter module, we will start building the GUI. First, we create a root which is where all the frames will be 'pinned'. A root is essentially a pinboard where the UI elements are placed. This root will hold all the objects belonging to the UI:

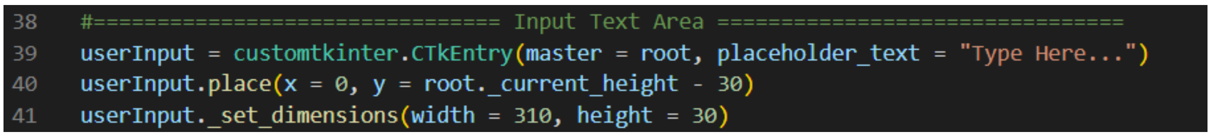


```
6   customtkinter.set_appearance_mode(settingsReader.theme)
7   customtkinter.set_default_color_theme("dark-blue")
8
9   root = customtkinter.CTk()
10  root.geometry("400x600")
11  root.title("Mable AI - Personal Assistant")
12  root.wm_resizable(width = False, height = False)
13
14  frame = customtkinter.CTkFrame(master = root)
15  frame.pack(pady = 30, padx = 0, fill = "both", expand = True)
16
17  logoLabel = customtkinter.CTkLabel(master = frame, text = "MABLE", font = ("Roboto", 25), text_color = "gray")
18  logoLabel.pack(padx = 50, pady = 200)
```

On line 9, an instance of a customtkinter window is created and named the root. On line 10, we're setting the dimensions of the root (which is also the main window) to 400 pixels by 600 pixels. Then, we are giving the window a title on the next line and setting both dimensions of the window to not be resizable on line 12. Then, we're creating a main frame called 'frame' and attaching it to the root. The frame is an object that can hold some UI elements such as text boxes or buttons. As we'll see in a bit, we can have multiple of these frames and we can switch between them. This allows us to have multiple windows in the same UI without creating other roots. Creating other roots and attaching their own frames is a lot slower and therefore inefficient so creating separate frames for the same root is faster and provides a smoother experience for the user.

On lines 17 and 18, I decided to place a logo in the 'background' of the frame which is a label with the project's name. A label is just some static text placed on a frame.

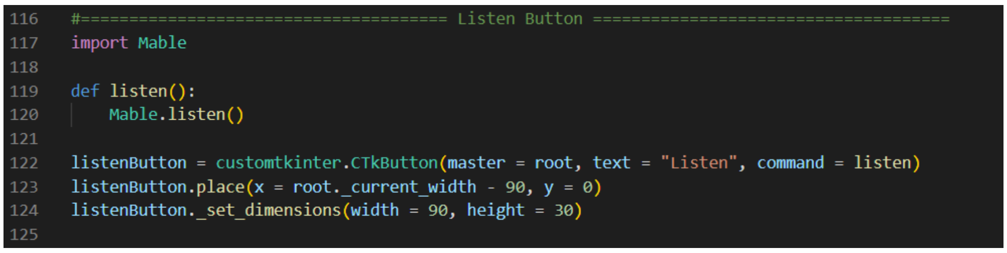
Next, we need to create the UI objects and then later code for their behaviour. We start with placing the main frame elements onto this frame. This will consist of a text area, an 'enter' button, a settings button as well as a 'listen' button which will call a different version of Mable that takes voice as input as opposed to text. Later on, we will see how this has been implemented to suit the original idea of the program listening in the background.



```
#================================= Input Text Area ================================
userInput = customtkinter.CTkEntry(master = root, placeholder_text = "Type Here...")
userInput.place(x = 0, y = root._current_height - 30)
userInput._set_dimensions(width = 310, height = 30)
```

Above is the code for the text area which is a customtkinter object of type 'CTkEntry'. It's stored as an object named 'userInput' since this is where the user will type their queries and where we'll copy them from.

The following code defines the 'listen' button:



```
#==================================== Listen Button ====================================
import Mable

def listen():
    Mable.listen()

listenButton = customtkinter.CTkButton(master = root, text = "Listen", command = listen)
listenButton.place(x = root._current_width - 90, y = 0)
listenButton._set_dimensions(width = 90, height = 30)
```

Each button has a command associated with it. The command is a function which will be called every time the button is pressed. In this case, it's the 'listen' function which calls the listen function of Mable, taking voice as input. We'll see later how this function looks like.

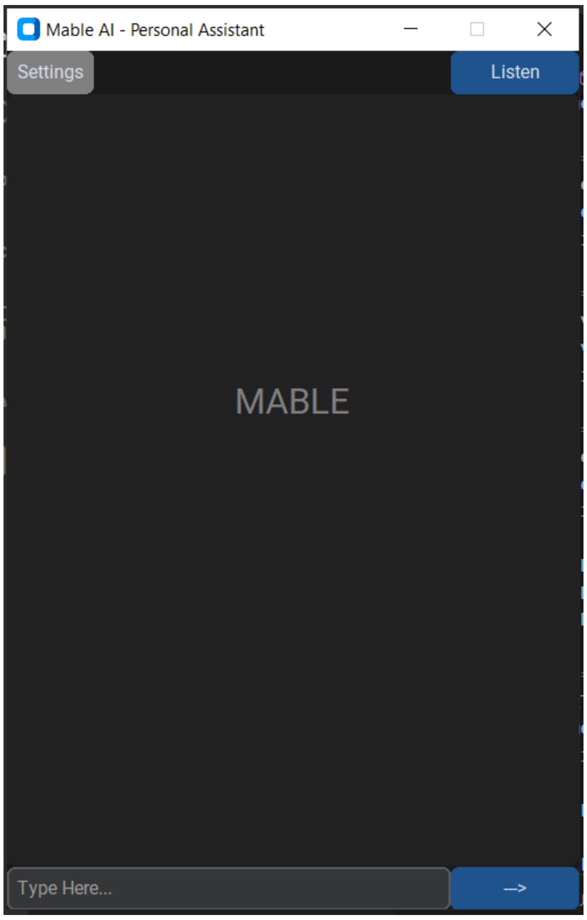
Next is the enter button or the 'ask' button. This button will call the 'ask' function which can be said to be the main or the central function in the project as it's the centre for computing most of the essential steps in analysing the user's query. Again, we will see this function and all the rest in detail later.

```
askButton = customtkinter.CTkButton(master = root, text = "--->", command = ask)
askButton.place(x = 310, y = root._current_height - 30)
askButton._set_dimensions(width = 90, height = 30)
```

Finally on the main frame, we have the settings button which takes the user to the settings window.

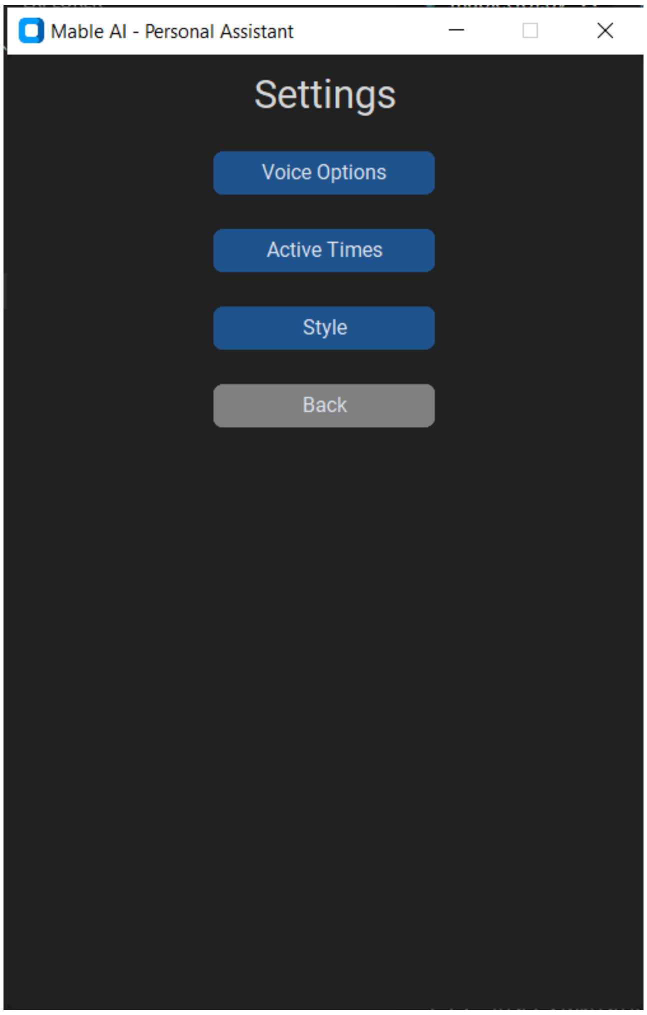
```
settingsButton = customtkinter.CTkButton(master = root, text = "Settings", command = settingsGUI, fg_color = "gray")
settingsButton.place(x = 0, y = 0)
settingsButton._set_dimensions(width = 45, height = 30)
```

The 'settingsGUI' function will swap the main frame with the settings frame where we will build the settings UI. Before we move on, here's how the main frame looks like:

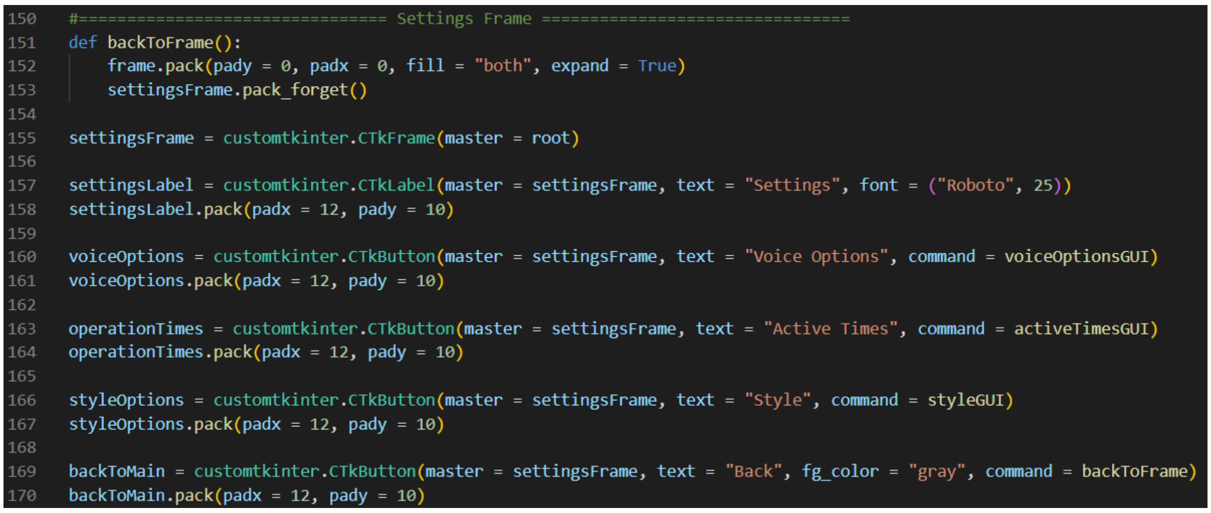


As you can see, all four UI elements described previously are placed on the main frame.

When the settings button is clicked, this window will be replaced by the settings window which can be seen on the next page.



Here, the four UI elements are all buttons which take the user to their corresponding frames. Here is the code for these buttons:



```
150  #============================== Settings Frame ==============================
151  def backToFrame():
152      frame.pack(pady = 0, padx = 0, fill = "both", expand = True)
153      settingsFrame.pack_forget()
154
155  settingsFrame = customtkinter.CTkFrame(master = root)
156
157  settingsLabel = customtkinter.CTkLabel(master = settingsFrame, text = "Settings", font = ("Roboto", 25))
158  settingsLabel.pack(padx = 12, pady = 10)
159
160  voiceOptions = customtkinter.CTkButton(master = settingsFrame, text = "Voice Options", command = voiceOptionsGUI)
161  voiceOptions.pack(padx = 12, pady = 10)
162
163  operationTimes = customtkinter.CTkButton(master = settingsFrame, text = "Active Times", command = activeTimesGUI)
164  operationTimes.pack(padx = 12, pady = 10)
165
166  styleOptions = customtkinter.CTkButton(master = settingsFrame, text = "Style", command = styleGUI)
167  styleOptions.pack(padx = 12, pady = 10)
168
169  backToMain = customtkinter.CTkButton(master = settingsFrame, text = "Back", fg_color = "gray", command = backToFrame)
170  backToMain.pack(padx = 12, pady = 10)
```

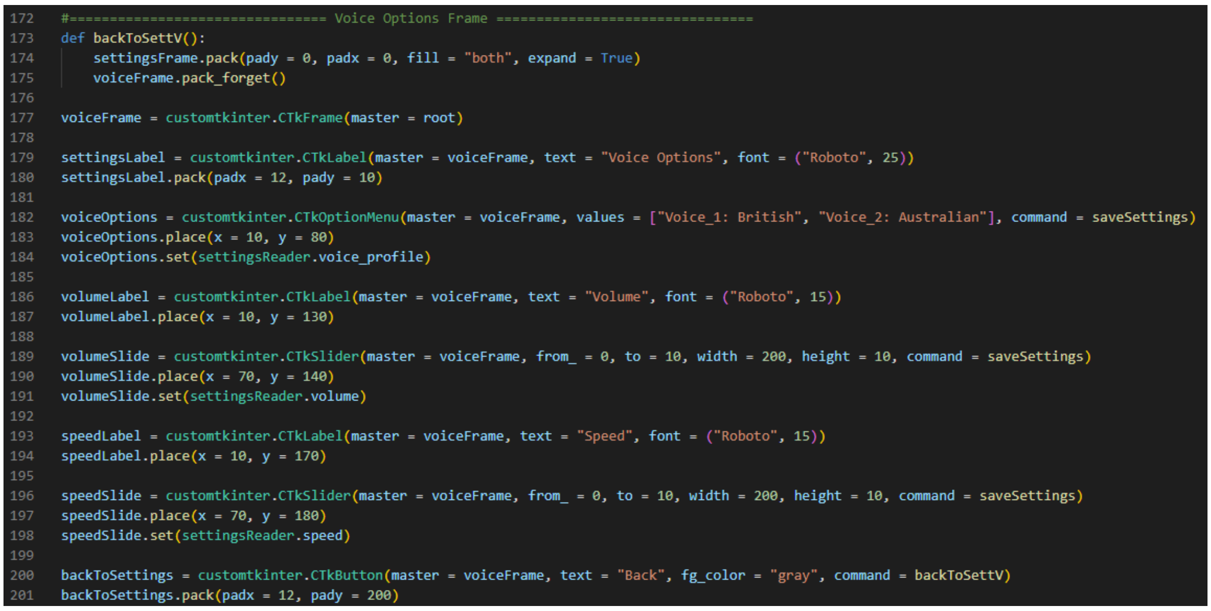
The lines from 160 to 170 code for the buttons on the settings frame. The 'backToFrame' function is the command for the back button which takes the user back to the main frame. The 'voiceOptionGUI', 'activeTimesGUI' and 'styleGUI' functions take the user to the voice option, active times and style frames. These functions look similar to the 'backToFrame' function except, instead of 'settingsFrame.pack_forget()', we replace the settingsFrame with whichever frame we are currently on.

Now, we'll see how the settings read from the Settings.JSON file will be used to customise the UI. In the 3 sub-frames shown above are the settings UI elements which the user can use to tailor the program to their tastes and needs.

3.1.1 Voice Option UI elements

First, we'll take a look at the UI elements that control the Voice options. These include the accent, volume and rate of speech.

As Mable can respond using TTS synthesis, we need the user to be able to customise Mable's voice. Here, you can see the code for the UI elements which control these settings:



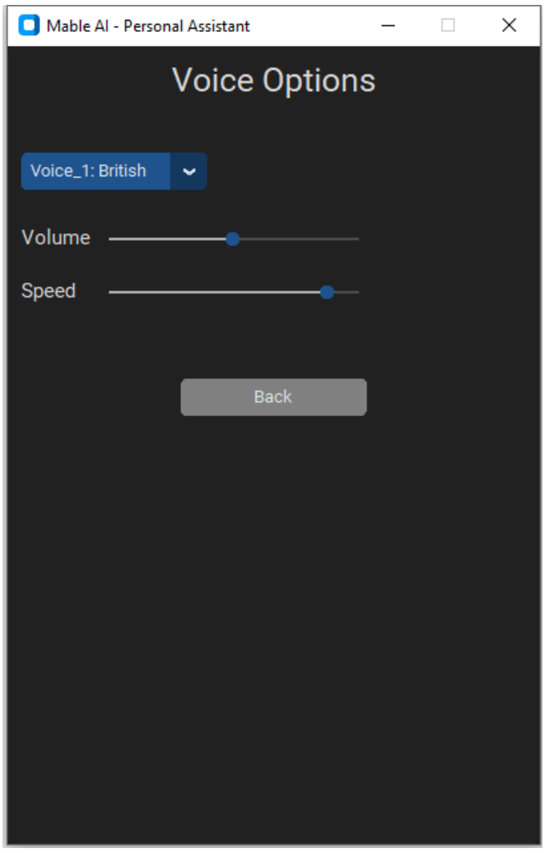
```
172  #=============================== Voice Options Frame ===============================
173  def backToSettV():
174      settingsFrame.pack(pady = 0, padx = 0, fill = "both", expand = True)
175      voiceFrame.pack_forget()
176
177  voiceFrame = customtkinter.CTkFrame(master = root)
178
179  settingsLabel = customtkinter.CTkLabel(master = voiceFrame, text = "Voice Options", font = ("Roboto", 25))
180  settingsLabel.pack(padx = 12, pady = 10)
181
182  voiceOptions = customtkinter.CTkOptionMenu(master = voiceFrame, values = ["Voice_1: British", "Voice_2: Australian"], command = saveSettings)
183  voiceOptions.place(x = 10, y = 80)
184  voiceOptions.set(settingsReader.voice_profile)
185
186  volumeLabel = customtkinter.CTkLabel(master = voiceFrame, text = "Volume", font = ("Roboto", 15))
187  volumeLabel.place(x = 10, y = 130)
188
189  volumeSlide = customtkinter.CTkSlider(master = voiceFrame, from_ = 0, to = 10, width = 200, height = 10, command = saveSettings)
190  volumeSlide.place(x = 70, y = 140)
191  volumeSlide.set(settingsReader.volume)
192
193  speedLabel = customtkinter.CTkLabel(master = voiceFrame, text = "Speed", font = ("Roboto", 15))
194  speedLabel.place(x = 10, y = 170)
195
196  speedSlide = customtkinter.CTkSlider(master = voiceFrame, from_ = 0, to = 10, width = 200, height = 10, command = saveSettings)
197  speedSlide.place(x = 70, y = 180)
198  speedSlide.set(settingsReader.speed)
199
200  backToSettings = customtkinter.CTkButton(master = voiceFrame, text = "Back", fg_color = "gray", command = backToSettV)
201  backToSettings.pack(padx = 12, pady = 200)
```

Starting from line 173, the 'backToSettV' function is defined. This function will take the user back to the setting frame from this one when the 'backToSettings' button has been pressed. Then, on lines 179 and 180, we have the code for this frame's label.

The first UI element to be defined is the 'voiceOptions' option menu. This is a dropdown menu where the user can choose between 2 voices. These to voices an be seen on line 182 in a 1D array after 'values'.

Next, we have the volume and speed slides. Both of these are Slider objects with a range from 0 to 10. The volume slider predictably controls the volume of Mable's voice while the speed slider controls the rate of speech. I also added labels next to these UI elements.

The Voice option UI can be seen on the next page.

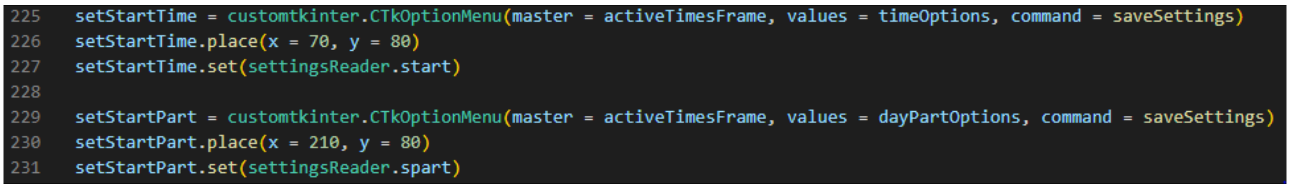


3.1.2 Active Times UI elements/The Save System

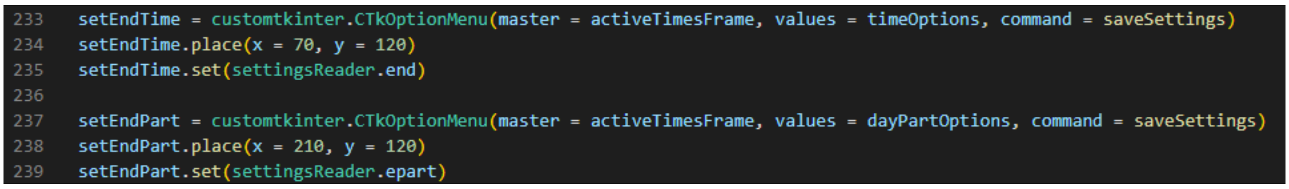
This part is really important looking at the original idea for the project. Originally, I wanted to create a system that would listen in the background for instructions from the user between set times. These times are the ‘Active times’ or the ‘Operational times’ of the voice controlled part of the system. The user is able to set the between what times they want Mable to listen fr instructions using the Active times settings UI.

They can set an hour from 01:00 to 12:00 and can also specify which half of the day they’re referring to by either selecting ‘AM’ or ‘PM’ after the time they chose. These are the ‘time’ and ‘time part’ settings where the ‘time’ refers to the actual hour selected by the user and the ‘time part’ refers to the part of the day.

Code for start/end time and start/end time part:

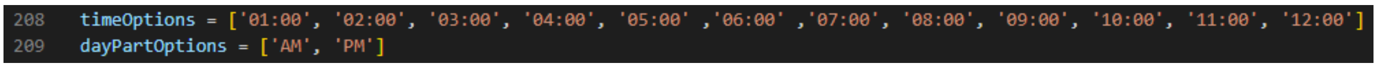


```
setStartTime = customtkinter.CTkOptionMenu(master = activeTimesFrame, values = timeOptions, command = saveSettings)
setStartTime.place(x = 70, y = 80)
setStartTime.set(settingsReader.start)

setStartPart = customtkinter.CTkOptionMenu(master = activeTimesFrame, values = dayPartOptions, command = saveSettings)
setStartPart.place(x = 210, y = 80)
setStartPart.set(settingsReader.spart)
```



```
setEndTime = customtkinter.CTkOptionMenu(master = activeTimesFrame, values = timeOptions, command = saveSettings)
setEndTime.place(x = 70, y = 120)
setEndTime.set(settingsReader.end)

setEndPart = customtkinter.CTkOptionMenu(master = activeTimesFrame, values = dayPartOptions, command = saveSettings)
setEndPart.place(x = 210, y = 120)
setEndPart.set(settingsReader.epart)
```

The values from which these dropdown menus read from are the following:

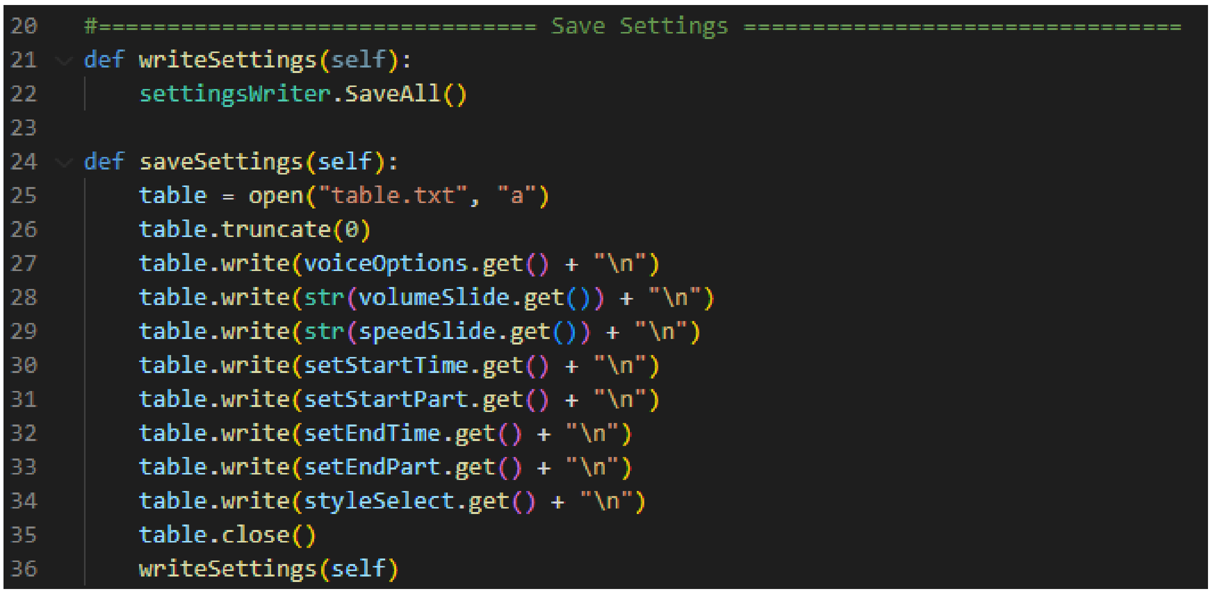


```
timeOptions = ['01:00', '02:00', '03:00', '04:00', '05:00' ,'06:00' ,'07:00', '08:00', '09:00', '10:00', '11:00', '12:00']
dayPartOptions = ['AM', 'PM']
```

One array for the hour values and one array for the day parts.

When the start time and start parts are selected, a function called ‘saveSettings’ is called to save the settings to a table from which the ‘settingsWriter’ updates the data in the Settings.JSON file. This is actually called every time the user makes a modification to the settings so that they are saved automatically.

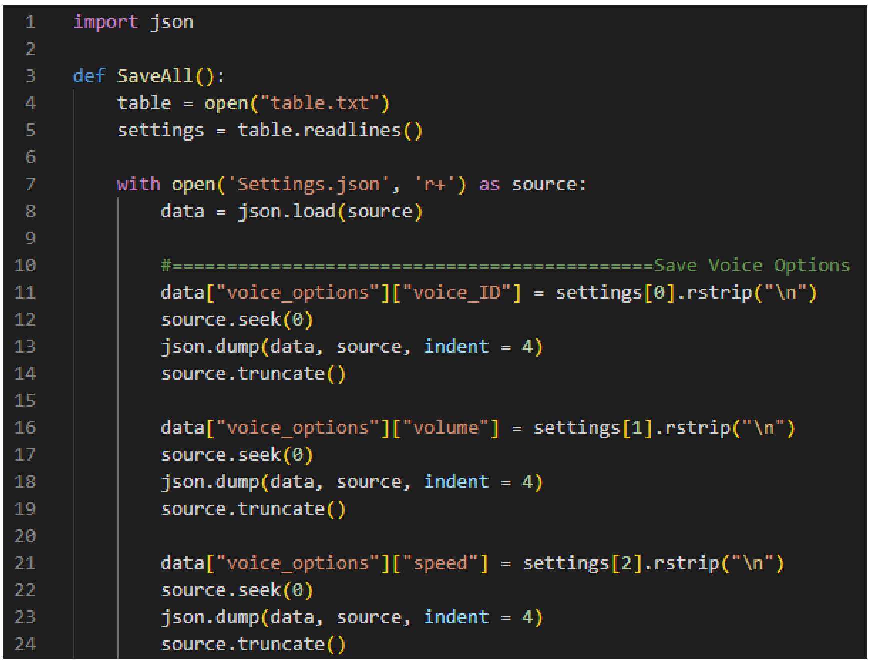
Below is the code for this function and the ‘settingsWriter’:



```
#================================ Save Settings ================================
def writeSettings(self):
    settingsWriter.SaveAll()

def saveSettings(self):
    table = open("table.txt", "a")
    table.truncate(0)
    table.write(voiceOptions.get() + "\n")
    table.write(str(volumeSlide.get()) + "\n")
    table.write(str(speedSlide.get()) + "\n")
    table.write(setStartTime.get() + "\n")
    table.write(setStartPart.get() + "\n")
    table.write(setEndTime.get() + "\n")
    table.write(setEndPart.get() + "\n")
    table.write(styleSelect.get() + "\n")
    table.close()
    writeSettings(self)
```

Here, the function is opening the table.txt file and writing the values of all the UI elements using the .get() method. Then, it’s calling the SaveAll() function from the settingsWriter custom module.

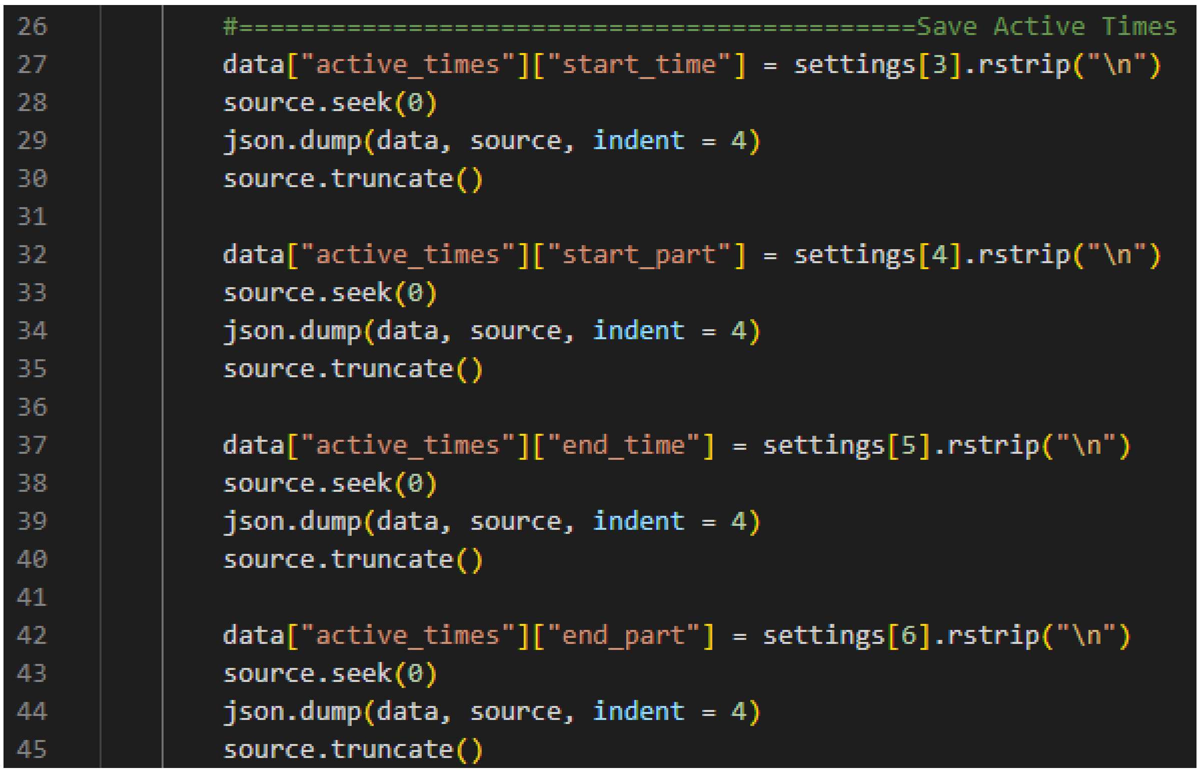
Settings writer saving voice options:



```
import json

def SaveAll():
    table = open("table.txt")
    settings = table.readlines()

    with open('Settings.json', 'r+') as source:
        data = json.load(source)

        #==========================================Save Voice Options
        data["voice_options"]["voice_ID"] = settings[0].rstrip("\n")
        source.seek(0)
        json.dump(data, source, indent = 4)
        source.truncate()

        data["voice_options"]["volume"] = settings[1].rstrip("\n")
        source.seek(0)
        json.dump(data, source, indent = 4)
        source.truncate()

        data["voice_options"]["speed"] = settings[2].rstrip("\n")
        source.seek(0)
        json.dump(data, source, indent = 4)
        source.truncate()
```

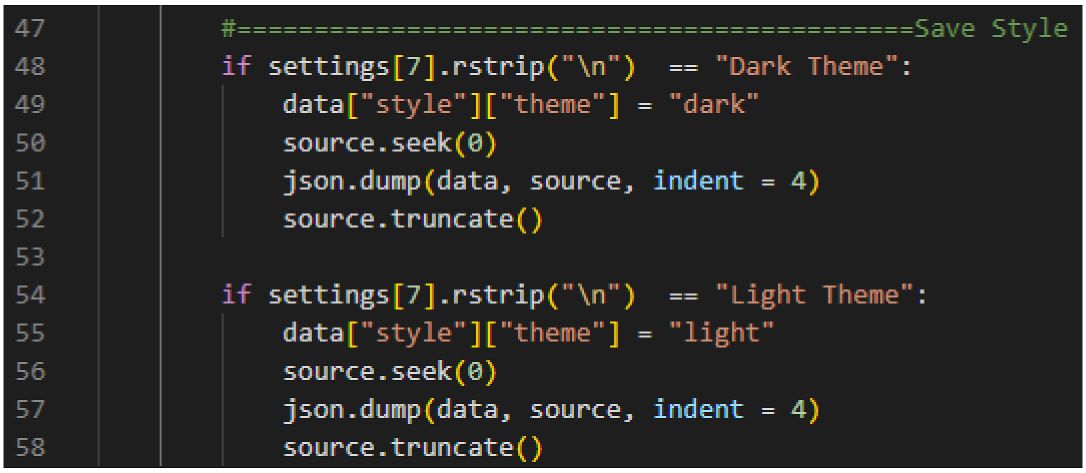
Here, the settingsWriter is reading the settings from the table which is a simple text file and writing them to the Settings.JSON file. It repeats this for each type of setting.

Active times:



```
        #============================================Save Active Times
        data["active_times"]["start_time"] = settings[3].rstrip("\n")
        source.seek(0)
        json.dump(data, source, indent = 4)
        source.truncate()

        data["active_times"]["start_part"] = settings[4].rstrip("\n")
        source.seek(0)
        json.dump(data, source, indent = 4)
        source.truncate()

        data["active_times"]["end_time"] = settings[5].rstrip("\n")
        source.seek(0)
        json.dump(data, source, indent = 4)
        source.truncate()

        data["active_times"]["end_part"] = settings[6].rstrip("\n")
        source.seek(0)
        json.dump(data, source, indent = 4)
        source.truncate()
```

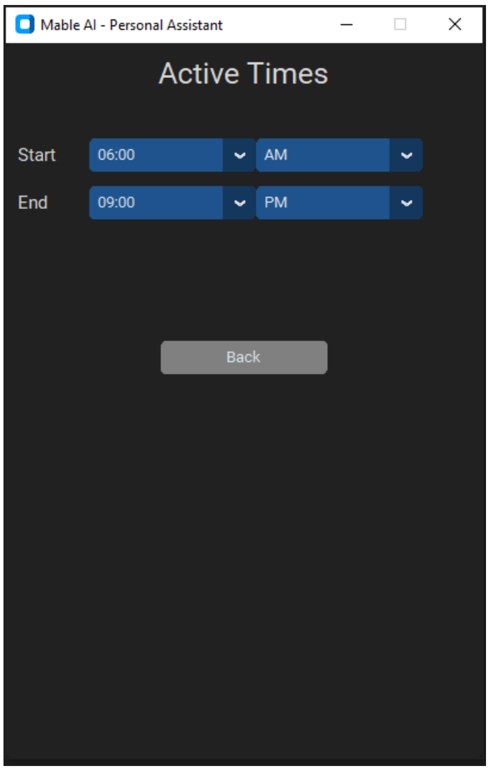
Style/Theme:



```
        #============================================Save Style
        if settings[7].rstrip("\n")  == "Dark Theme":
            data["style"]["theme"] = "dark"
            source.seek(0)
            json.dump(data, source, indent = 4)
            source.truncate()

        if settings[7].rstrip("\n")  == "Light Theme":
            data["style"]["theme"] = "light"
            source.seek(0)
            json.dump(data, source, indent = 4)
            source.truncate()
```

Specifically, what the SaveAll() function is doing is it's reading each line of the table.txt file and removing the '\n' character from them. Then, it writes each of these lines to their corresponding settings in the Settings.JSON file by selecting each element from the file and forcing it to equal that line from the table.txt file. After this, it goes back to line 1 of the Settings.JSON file and 'dumps' the data it read. This means it rewrites the whole file with the one modification made.

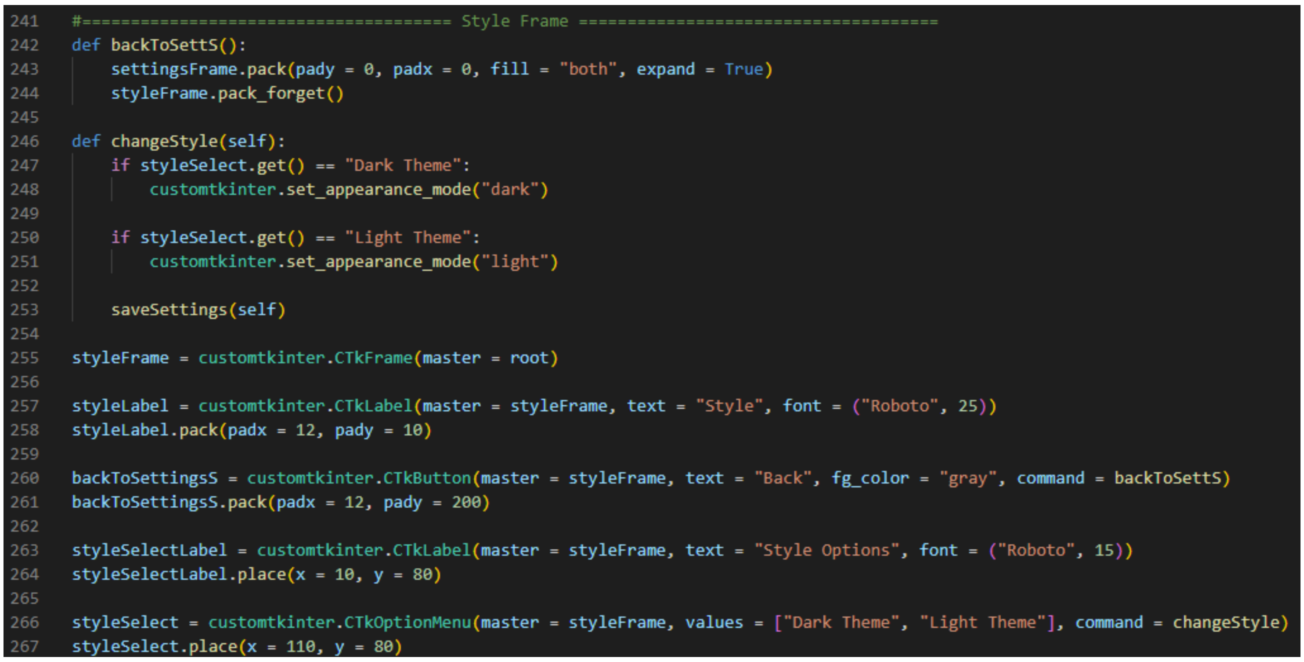
The Active times window can be seen on the next page.



3.1.3 Style UI elements

Finally, we're at the Style settings portion of the settings window. This part is the shortest and the simplest out of all the settings elements. This window only has a back button like all the others and one dropdown menu with 2 options: Dark Theme or Light Theme. The user can toggle between them as they wish and this too will be saved to the settings file as we've just seen.

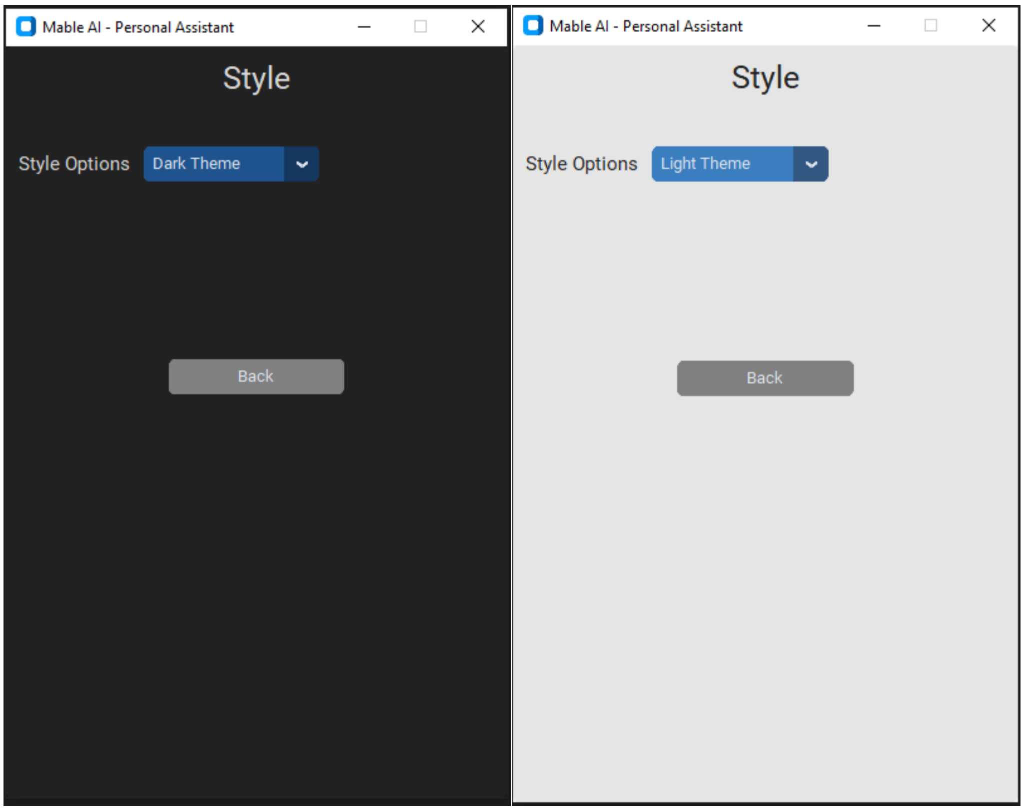
The code for this window:



```
#===================================== Style Frame ====================================
def backToSettS():
    settingsFrame.pack(pady = 0, padx = 0, fill = "both", expand = True)
    styleFrame.pack_forget()

def changeStyle(self):
    if styleSelect.get() == "Dark Theme":
        customtkinter.set_appearance_mode("dark")

    if styleSelect.get() == "Light Theme":
        customtkinter.set_appearance_mode("light")

    saveSettings(self)

styleFrame = customtkinter.CTkFrame(master = root)

styleLabel = customtkinter.CTkLabel(master = styleFrame, text = "Style", font = ("Roboto", 25))
styleLabel.pack(padx = 12, pady = 10)

backToSettingsS = customtkinter.CTkButton(master = styleFrame, text = "Back", fg_color = "gray", command = backToSettS)
backToSettingsS.pack(padx = 12, pady = 200)

styleSelectLabel = customtkinter.CTkLabel(master = styleFrame, text = "Style Options", font = ("Roboto", 15))
styleSelectLabel.place(x = 10, y = 80)

styleSelect = customtkinter.CTkOptionMenu(master = styleFrame, values = ["Dark Theme", "Light Theme"], command = changeStyle)
styleSelect.place(x = 110, y = 80)
```

The 'styleSelect' object is the option menu where the user can pick between the light and dark themes. The 'changeStyle' function sets the appearance mode based on the value of the styleSelect element and is the command function for this element.

This is the Style settings window:



3.2 Input analysis

As we've talked about in the Design section, we need a NLP (Natural Language Processing) System which abstracts the user's input. In this part of the Technical Solution section, we'll pick apart this system and carefully go through each step it takes to simplify the input. I have made a few changes to the traditionally excepted NLP System to better suit this project.

The first and perhaps biggest change is the removal of a large database of words. As I've realised while constructing the database and putting it to use, this is only needed for a complex chatbot. Meaning, an AI which is specifically designed for human-like communication. A recently developed example of such a system (although it's way more powerful than a simple chatbot like Google's Cleverbot, for example) would be ChatGPT. Since Mable is intended to be a virtual personal assistant, it's functionality as a conversation partner can be and is fairly limited and so doesn't need a majorly extensive vocabulary. However, as stated in the Analysis, I did want it to have some ability to carry a very simple conversation to make it feel closer to a 'real' PA, so I've implemented a short response library which I'll show and explain further on.

For the same reason, the second change I've made was the removal of word stemming. Again, word stemming is only really needed in larger projects and ones which are more tailored to conversation synthesis. In a smaller project such as this one, adding this step would contribute very little to the overall efficiency of input analysis if not slow it down.

Leaving these two steps out of the final project, me and a fellow classmate found that this way, Mable could carry out simple tasks when asked in a text format about 18%-20% faster than Microsoft's Cortana and only lagging behind about 3%-5% when the commands were given in speech.

Thirdly, I decided to add an 'AskWordSimplifier' function which simplifies text inputs such as 'what's' or 'what is' to simply 'what'. It gets rid of any extension of the asking word and ultimately only includes the asking word itself. This function is called at both types of input: speech and text.

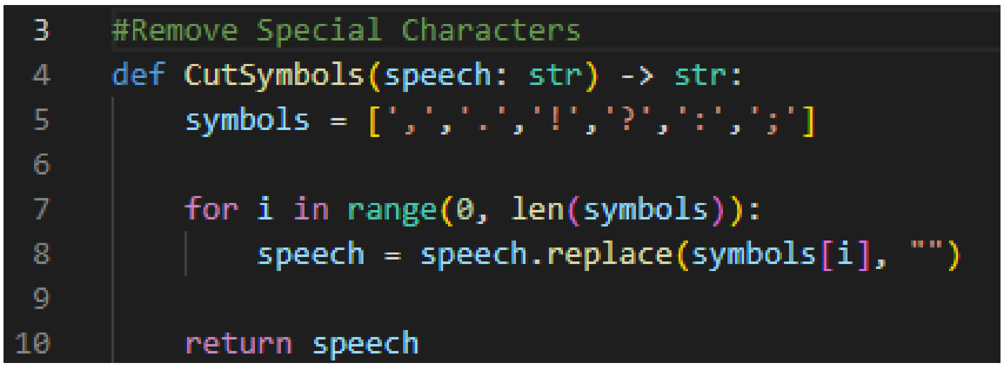
The fourth and last addition I made was a 'CutSymbols' function which is only called when the program receives a text input. As the name of the function might suggest, this function cuts any special characters from the input as they don't add any specificity to the query. These include '**@**', '**!**', '**?**', '**.**', '**,**', '**:**', and '**;**'.

So, with these changes in mind, let's look at the actual code for the NLP system.

3.2.1 NLPS Code

I've decided to create a custom module for this system as well so that the main program can call it at any time. The NLPS is comprised of four steps of abstraction. The first step is removing any special characters, the second is tokenisation, the third is simplifying asking words and the last is removing any stop words. The steps are summarised in one function called the 'LanguageAbstractor'.

Let's look at the first step: the removal of special characters.

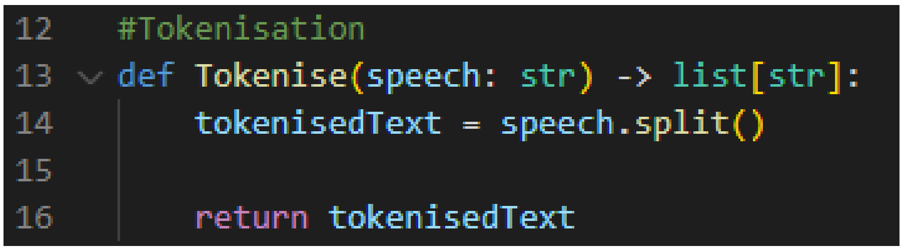


```
3   #Remove Special Characters
4   def CutSymbols(speech: str) -> str:
5       symbols = [',','.','!','?',':',';']
6   
7       for i in range(0, len(symbols)):
8           speech = speech.replace(symbols[i], "")
9   
10      return speech
```

Above is the function for special character removal. The function loops through the input string on line 7 and replaces any characters in the string which are a member of the 'symbols' array with an empty character. After this is done, it returns the 'speech' string which is the fully updated string without any special characters.

The next function is perhaps the simplest one. The tokenizer function takes the string and splits it up into words which it then places in an array in the same order as in the original string.

Here is the code for this function:

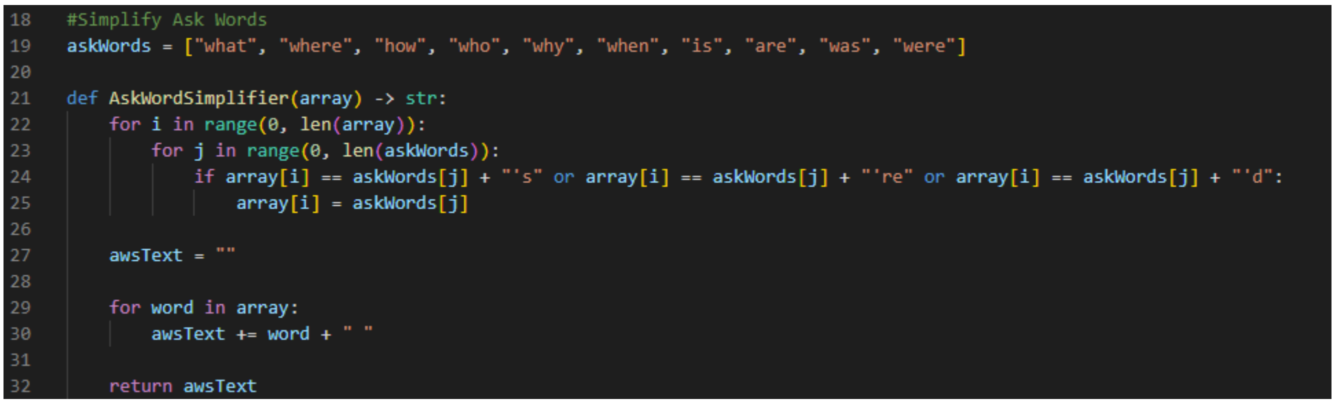


```
#Tokenisation
def Tokenise(speech: str) -> list[str]:
    tokenisedText = speech.split()

    return tokenisedText
```

As you can see, the input is of type ‘string’ and the output is of return type ‘list’. more specifically, a list of strings. This function uses a built-in Python procedure called ‘split’ which takes a string and splits it at each point where it finds a certain specified character. When a character is not specified, it defaults to looking for spaces. In a normal English sentence, there is a space between any two words, so this is perfect for tokenisation. The function then returns the ‘tokenisedText’ array which is the fully tokenised input string.

Next comes the simplification of asking words. This function takes a list of strings and finds any asking words with some extensions. It then removes these extensions.

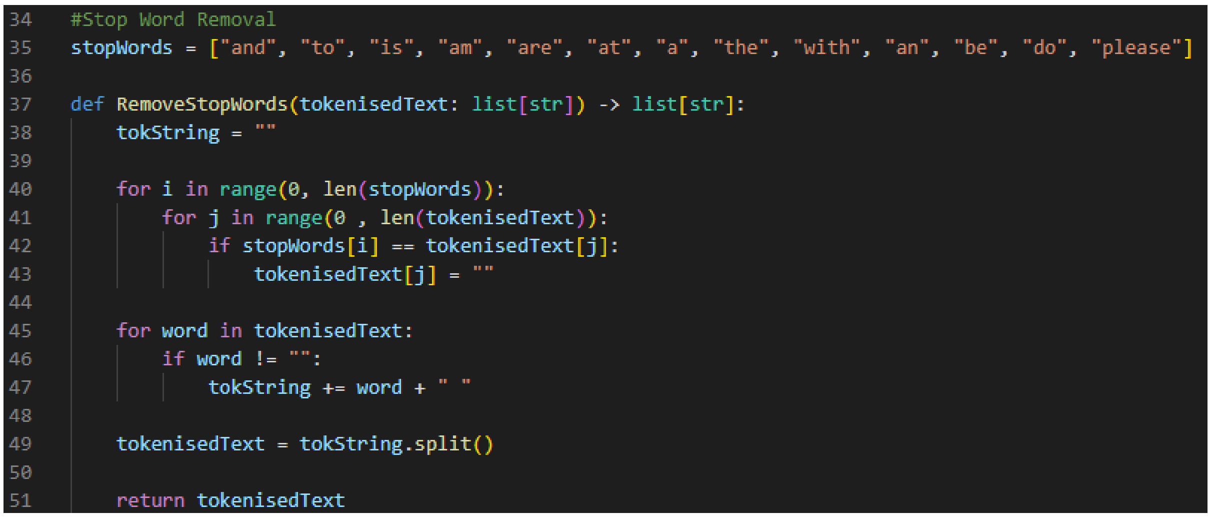
Below is the code for this:



```
18  #Simplify Ask Words
19  askWords = ["what", "where", "how", "who", "why", "when", "is", "are", "was", "were"]
20
21  def AskWordSimplifier(array) -> str:
22      for i in range(0, len(array)):
23          for j in range(0, len(askWords)):
24              if array[i] == askWords[j] + "'s" or array[i] == askWords[j] + "'re" or array[i] == askWords[j] + "'d":
25                  array[i] = askWords[j]
26
27      awsText = ""
28
29      for word in array:
30          awsText += word + " "
31
32      return awsText
```

First, we define the list of asking words on line 19. Then, we loop through the input array and for each word in the array, loop through the asking words. If any of the words in the array are the same as an asking word plus one of the extensions ‘s, ‘re or ‘d, we force that word in the array to change to the asking word with which it has been matched. This creates a new array of words without any extended asking words. Then, this array is converted back into a string.

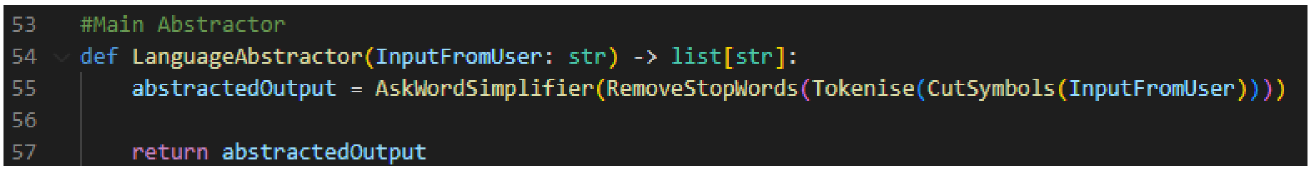
Finally, we have the stop word removal step. Here, we remove any words form the input which are classified as ‘stop words’. In case you’ve forgotten, stop words are words which have not additional information regarding the user’s query. This includes connectives such as ‘and’, ‘with’ and ‘at’ or ‘is’, ‘was’ and ‘were’.



```
34  #Stop Word Removal
35  stopWords = ["and", "to", "is", "am", "are", "at", "a", "the", "with", "an", "be", "do", "please"]
36
37  def RemoveStopWords(tokenisedText: list[str]) -> list[str]:
38      tokString = ""
39
40      for i in range(0, len(stopWords)):
41          for j in range(0 , len(tokenisedText)):
42              if stopWords[i] == tokenisedText[j]:
43                  tokenisedText[j] = ""
44
45      for word in tokenisedText:
46          if word != "":
47              tokString += word + " "
48
49      tokenisedText = tokString.split()
50
51      return tokenisedText
```

Again, we first define the array from which the function will analyse the input. In this case, the array of stop words. In the function, we loop through the stop words and for every stop word, we loop through the input array. Similarly to the previous function, if we find a pair of words that match, we replace the word in the input array with an empty character. Then, we loop through this new array and if we find a word that is not an empty character, we replace it with the word itself and a space. this is done so that when we assemble the array into a string again, there is a space between each word, forming a sentence. Finally, we take this newly assembled sentence and apply the split method to tokenise it once again and we then return this tokenised text as the output of the function.

The LanguageAbstractor takes these steps and applies them in the right order to any input from the user. (note: the order in which the functions are defined is not necessarily the order in which they are applied):



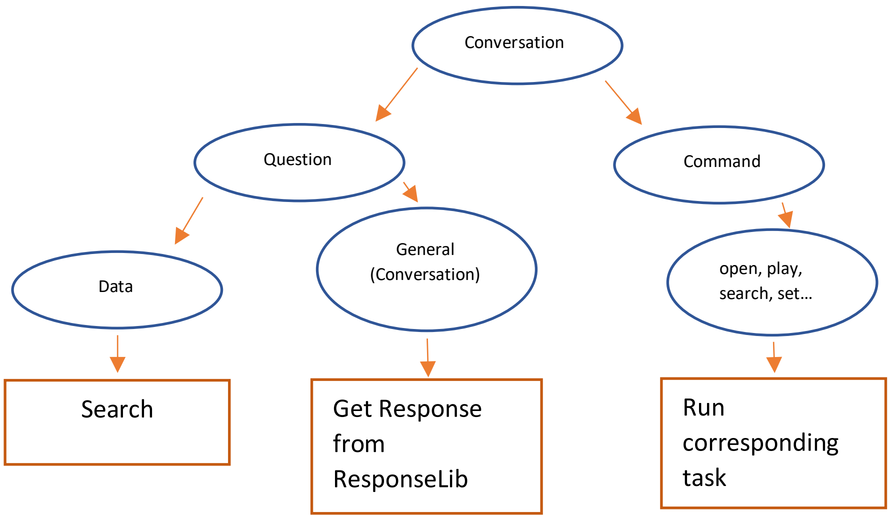
```
#Main Abstractor
def LanguageAbstractor(InputFromUser: str) -> list[str]:
    abstractedOutput = AskWordSimplifier(RemoveStopWords(Tokenise(CutSymbols(InputFromUser))))

    return abstractedOutput
```

3.2.2 Query Categorisation

In order for the program to decide what type of query it's dealing with, it needs to categorise it. This is needed since the same key words will need to trigger a different task in varying queries. For example, the query "What causes earthquakes?" is categorised as a question and has a sub-category of a 'data query' or 'data question' since the user is asking for a specific piece of data which would normally be searched for online. However, the query "What is your name?" is categorised a 'conversation query' since it's starting a conversation between the program and the user. The shared keyword between the two queries is "What". However, they technically have slightly different meanings since they correspond to different tasks and so the program needs to differentiate between the two types of the keyword.

This is done with a combination of the custom 'queryCat' and 'ConversationCheck' modules. The first module categorises the query as either a 'conversation', a 'question' or a 'command' using a 'Categorise' class. We use a class for this as we need the categories to be stored as objects since the queries are also stored as such. The 'question' and 'command' categories also have a sub-category which point to the types of question or command (what, where, how… play, open, visit…). The second module then takes the query if it has been categorised as 'conversation' type and returns a response from the response library based on the contents of the input.

For the whole categorisation process, I'm using a category tree:

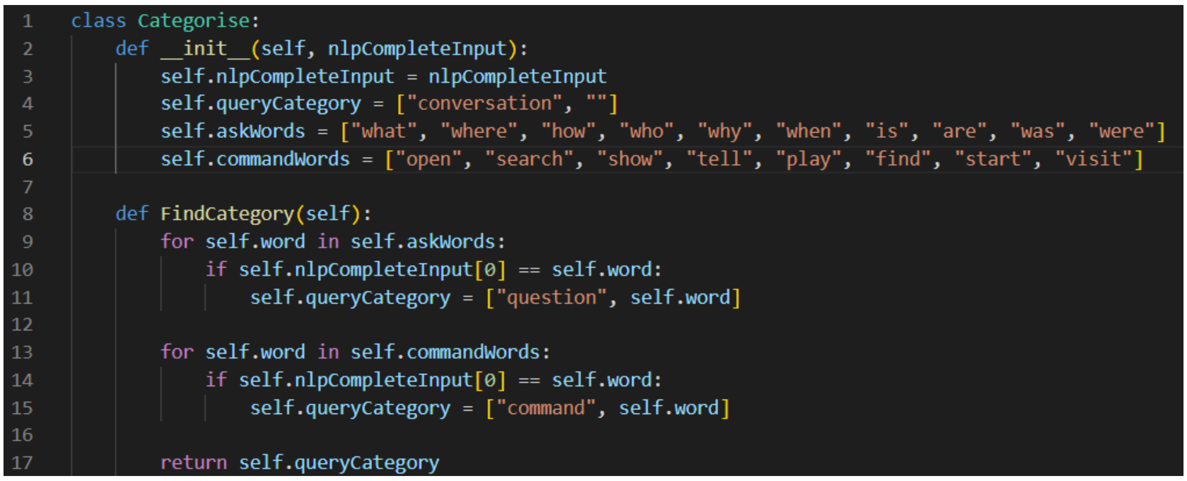


The default category is just a conversation and as the program moves through the tree, this is refined. In the end, it finds the correct task to complete.

The Categorise class's constructor has the default category and the array of ask words and command words to check the input against. The 'FindCategory' function checks if the abstracted input starts with one of the words from either of these arrays. It then returns the category and the sub-category. If it doesn't find any of the key words from the constructor, the category stays the same.

The abstracted input must start with one of the words from the instructor as all stop words have been removed from it. Alternatively, it must be a conversation which is why the default category if the conversation.

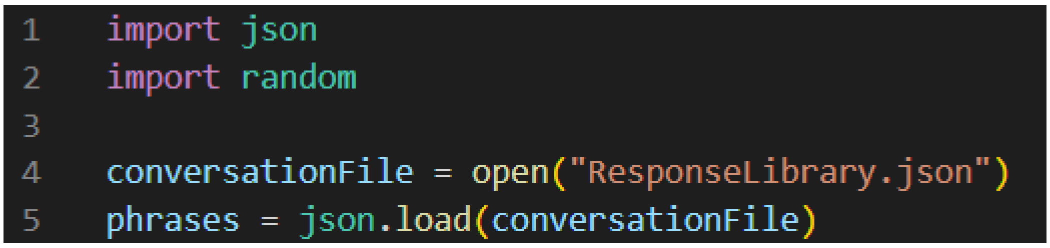
The code for the Categorise class:



```
1   class Categorise:
2       def __init__(self, nlpCompleteInput):
3           self.nlpCompleteInput = nlpCompleteInput
4           self.queryCategory = ["conversation", ""]
5           self.askWords = ["what", "where", "how", "who", "why", "when", "is", "are", "was", "were"]
6           self.commandWords = ["open", "search", "show", "tell", "play", "find", "start", "visit"]
7
8       def FindCategory(self):
9           for self.word in self.askWords:
10              if self.nlpCompleteInput[0] == self.word:
11                  self.queryCategory = ["question", self.word]
12
13          for self.word in self.commandWords:
14              if self.nlpCompleteInput[0] == self.word:
15                  self.queryCategory = ["command", self.word]
16
17          return self.queryCategory
```

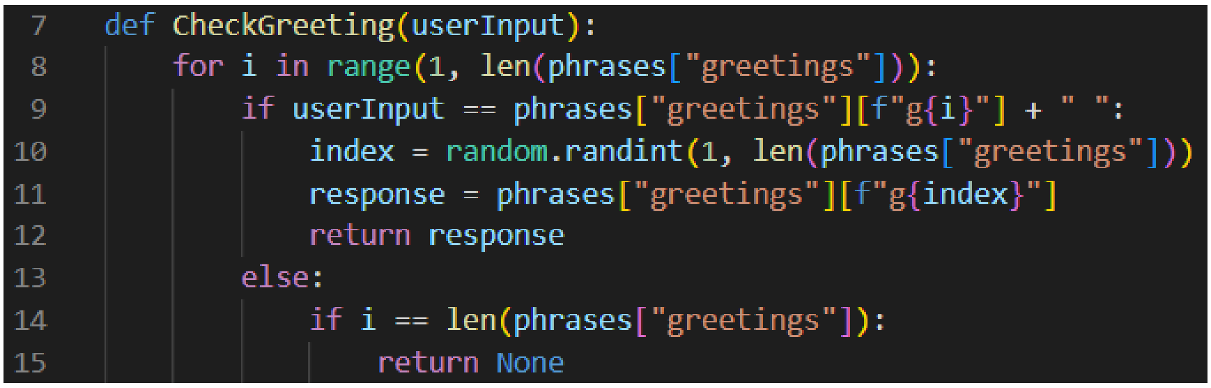
The '__init__' function is the constructor. It's the function which takes in the abstracted input and builds the Categorise object for it. Then, in the FindCategory function, we have two loops which go through the ask word and command word arrays and check if the input starts with any of the words from them. The final category is then returned on line 17.

After the query has been categorised, we call the ConversationCheck module to further analyse the text. If the query has been categorised as a conversation, then the module determines the appropriate response using the Response Library.



```
import json
import random

conversationFile = open("ResponseLibrary.json")
phrases = json.load(conversationFile)
```

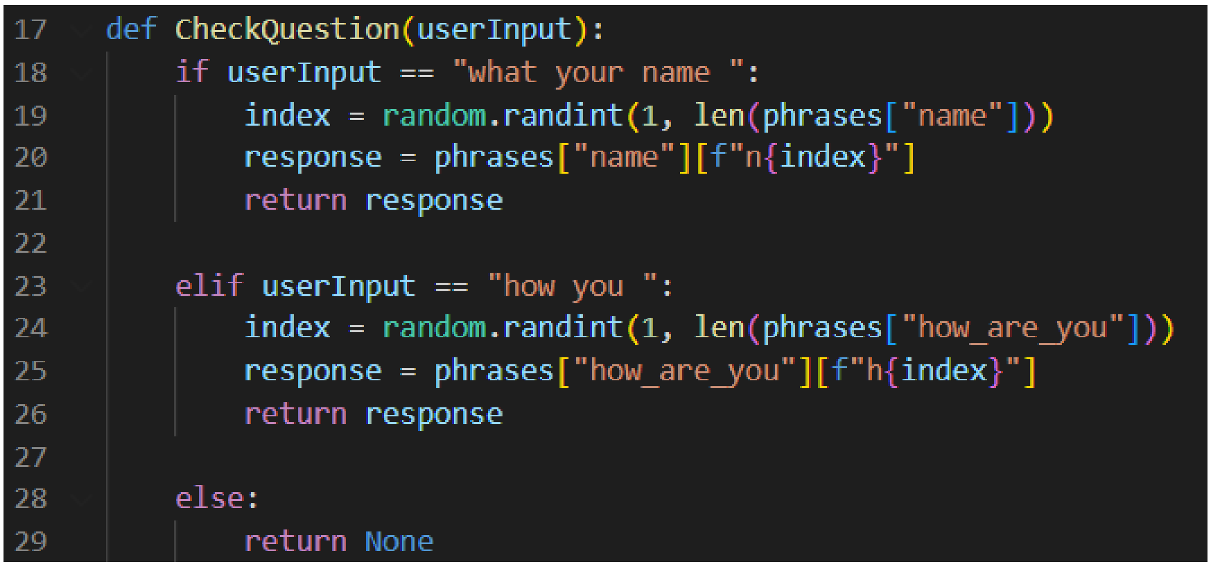
First, we open the Response Library so that we can read the possible responses from it. We also import the random module as we'll need to generate a random response from the Response Library.



```python
def CheckGreeting(userInput):
    for i in range(1, len(phrases["greetings"])):
        if userInput == phrases["greetings"][f"g{i}"] + " ":
            index = random.randint(1, len(phrases["greetings"]))
            response = phrases["greetings"][f"g{index}"]
            return response
        else:
            if i == len(phrases["greetings"]):
                return None
```

The function above checks if the user's input is a greeting. It does this by looking at the greetings in the Response Library and seeing if any of them match the user's input. If not, it returns 'None' meaning it's not a greeting. If, however, it is, then it will select a greeting from the same list at random to display on the screen.

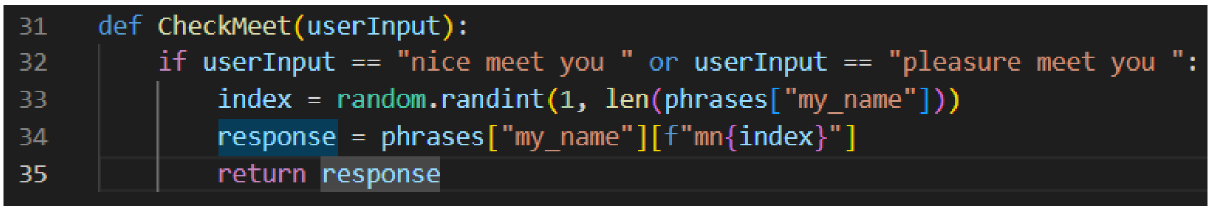
The following function checks if the user's input was a question of 'conversation' type. I've defined two such questions: "What's your name" and "How are you?". Of course, there could be many more such questions but as we've said previously, this isn't the main purpose of the project, so these two are enough. I could add more to the library later on.



```python
def CheckQuestion(userInput):
    if userInput == "what your name ":
        index = random.randint(1, len(phrases["name"]))
        response = phrases["name"][f"n{index}"]
        return response

    elif userInput == "how you ":
        index = random.randint(1, len(phrases["how_are_you"]))
        response = phrases["how_are_you"][f"h{index}"]
        return response

    else:
        return None
```

You may notice that the inputs I'm checking this against are grammatically incorrect. This is because any variation of these two questions will result in the strings above after they have been passed through the NLPS. Again, if the input doesn't fit either of these possibilities, the function returns 'None'. If it does, it displays a randomly selected response from the Response Library (RL).

The third function in the module checks if the input was something similar to a 'nice to meet you' response.



```python
def CheckMeet(userInput):
    if userInput == "nice meet you " or userInput == "pleasure meet you ":
        index = random.randint(1, len(phrases["my_name"]))
        response = phrases["my_name"][f"mn{index}"]
        return response
```

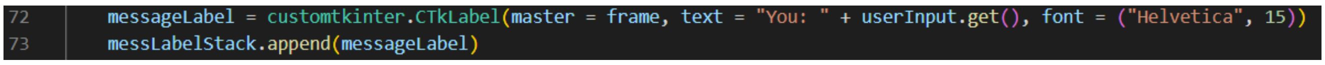
This function doesn't have an alternative because of the order in which these functions are executed. This is the last function to execute from the module and since it's only called if the input was a 'conversation' type, it has to satisfy one of the functions and so, if it didn't on the first two, it must on this last one.

3.3 Message Stack Management and Message Display

The messages which appear on the UI after the user has typed their input or after Mable responds to that input are managed by a stack. This stack contains 'Message' objects from a 'Message' class. After a message is entered or generated, a Message object is created for it which will the be placed in the Message stack. This stack is then updated after each new object with the new relative positions of the messages on the display.

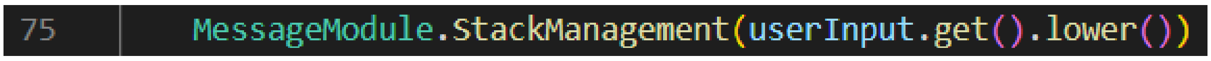
I created a Message module which handles most of the stack management. Here is the process of how a message is displayed using this stack:

First, after a message is entered from the user's side, we create a 'messageLabel' object which will be the message displayed on the screen and we add it to a stack of labels:



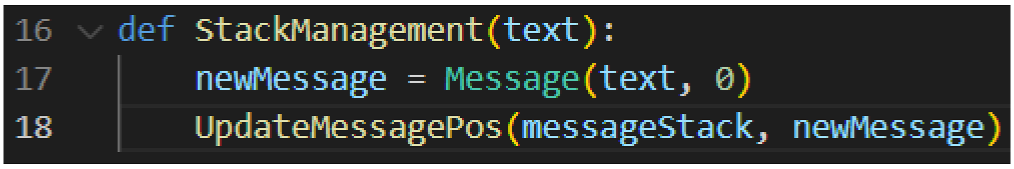
```
    messageLabel = customtkinter.CTkLabel(master = frame, text = "You: " + userInput.get(), font = ("Helvetica", 15))
    messLabelStack.append(messageLabel)
```

Then, we call the 'StackManagement' function form the custom Message module and we apply it to this new message:

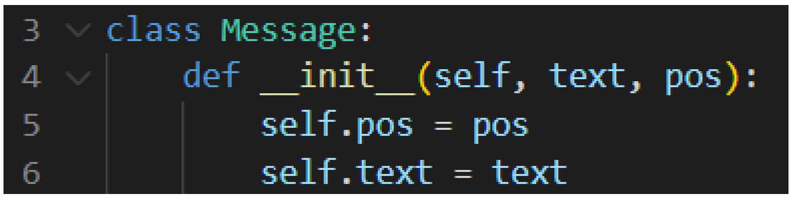


```
    MessageModule.StackManagement(userInput.get().lower())
```

This function first creates a new Message object for this label and then calls the 'UpdateMessagePos':



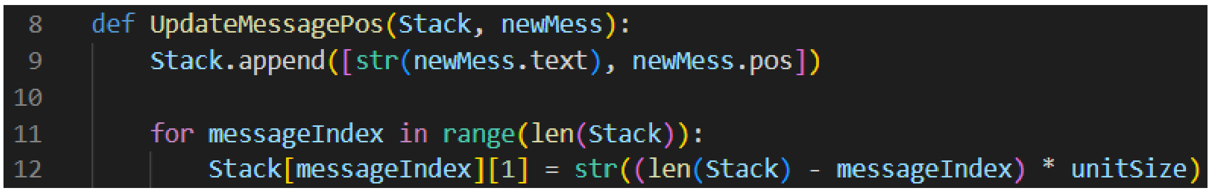
```
def StackManagement(text):
    newMessage = Message(text, 0)
    UpdateMessagePos(messageStack, newMessage)
```



```
class Message:
    def __init__(self, text, pos):
        self.pos = pos
        self.text = text
```

A Message object has a text and a position. The text is just the original message for which we are creating this object.

The UpdateMessagePos function then updates the positions of the messages in the stack by moving them up by a fixed amount called the 'unit size' which I set to 20 pixels



```
def UpdateMessagePos(Stack, newMess):
    Stack.append([str(newMess.text), newMess.pos])

    for messageIndex in range(len(Stack)):
        Stack[messageIndex][1] = str((len(Stack) - messageIndex) * unitSize)
```

It does this by looping through the object-populated Message stack and changing it to its position from the back multiplied by the unit size:

$$P_n[M] = U(L_S - I_M)$$

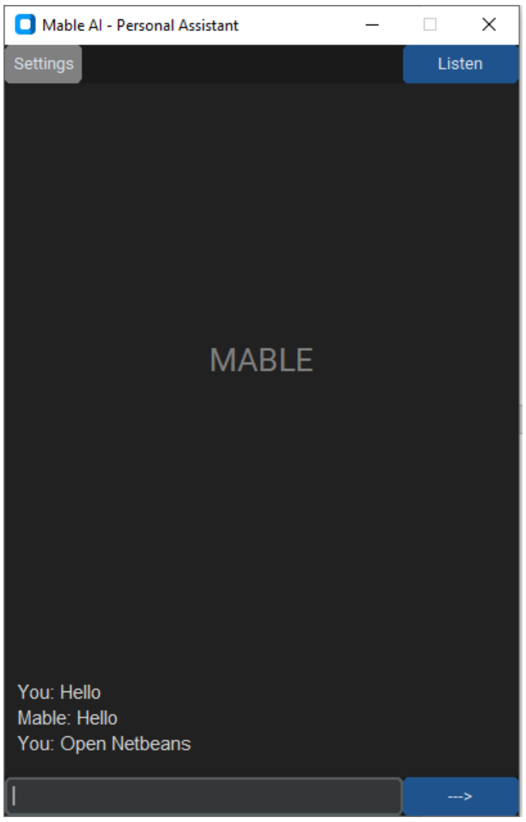
where $P_n[M]$ is the new position of the message, U is the unit size, L_S is the length of the stack and I_M is the index of the message in the stack.

The relative index of the message is $L_S - I_M$ since the message that is at the top of the stack has to be on the bottom of the screen. So, the index must be inverted.

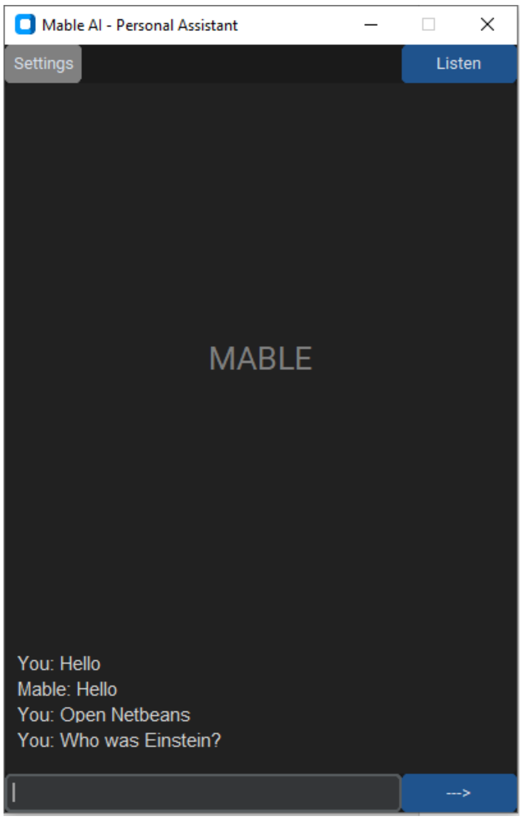
Finally, after the positions have been updated, we loop through the label stack and place each label at their updated positions:

```
    for messageIndex in range(0, len(messLabelStack)):
        messLabelStack[messageIndex].place(x = 10, y = frame._current_height - int(MessageModule.messageStack[messageIndex][1]) - 20)
```

Here is how this looks like:



You: Hello
Mable: Hello
You: Open Netbeans



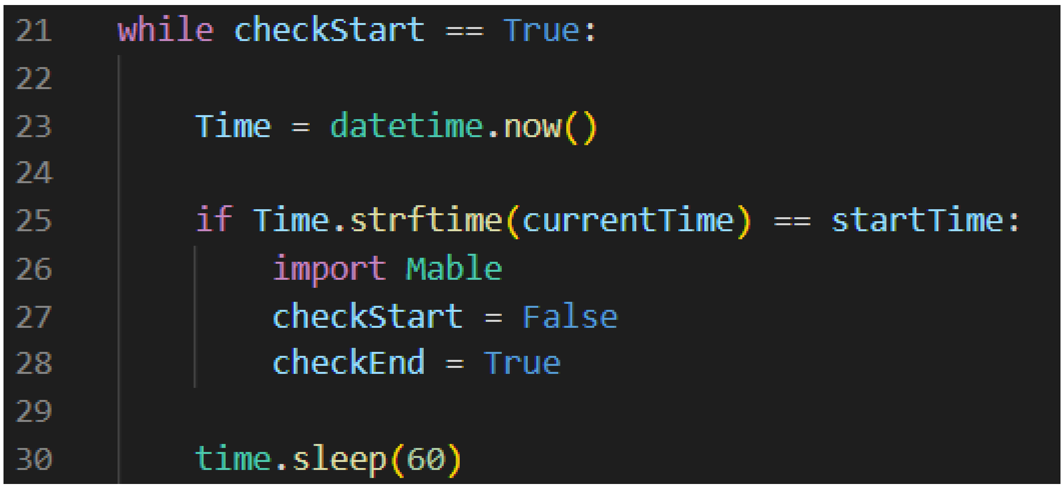
You: Hello
Mable: Hello
You: Open Netbeans
You: Who was Einstein?

As you can see, the labels from before have moved up 20 pixels to 'make space' for the new one.

3.4 Time Management System

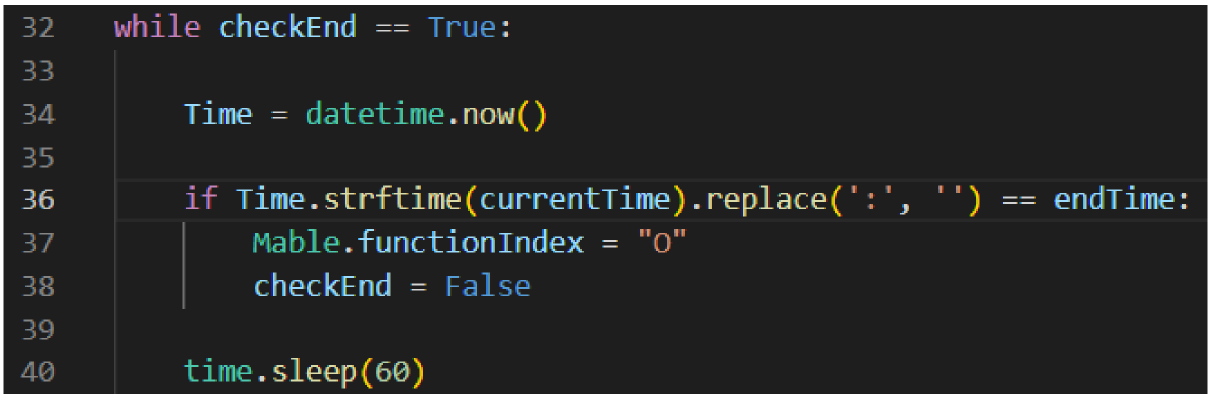
As we've said before, the program should have a mode where it listens to spoken commands between some set time intervals. These time intervals (the active times) are set by the user on the Active Times window of the Settings.

The management and monitoring of these times is the job of the Time Management System (TMS). We have two loops to check the activation and termination times. Every minute, the program takes the hour part of the current time and compares it with set start time. Once they are equal, it starts the listening part of the program and enters the other loop to check if the current time is the same is the same as the set end time. Once they are the same, it terminates the program.



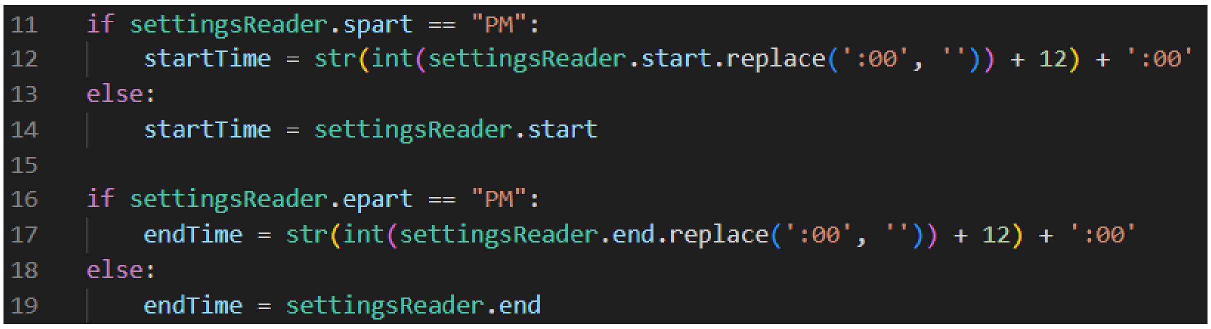
```python
while checkStart == True:

    Time = datetime.now()

    if Time.strftime(currentTime) == startTime:
        import Mable
        checkStart = False
        checkEnd = True

    time.sleep(60)
```

The code above checks the start time and the code below checks the end time.



```python
while checkEnd == True:

    Time = datetime.now()

    if Time.strftime(currentTime).replace(':', '') == endTime:
        Mable.functionIndex = "0"
        checkEnd = False

    time.sleep(60)
```

The following code takes the start and end times and converts them into a number denoting which half of the day they are in by adding 12 hours to them and then removing the ‘:00’ part from the end of the strings.



```python
if settingsReader.spart == "PM":
    startTime = str(int(settingsReader.start.replace(':00', '')) + 12) + ':00'
else:
    startTime = settingsReader.start

if settingsReader.epart == "PM":
    endTime = str(int(settingsReader.end.replace(':00', '')) + 12) + ':00'
else:
    endTime = settingsReader.end
```

These become the numbers which are compared with the current time.

3.5 Main Function and Auditory Inputs

Now we’ve arrived to the last part of the project which is also the most important part. This is the activation function of the ‘ask’ button. When the button is pressed, the input is processed and analysed in order to determine what the program should do.

Here is the main function:

```
def ask():

     messageLabel = customtkinter.CTkLabel(master = frame, text = "You: " +
userInput.get(), font = ("Helvetica", 15))
    messLabelStack.append(messageLabel)

    MessageModule.StackManagement(userInput.get().lower())

    for messageIndex in range(0, len(messLabelStack)):
        messLabelStack[messageIndex].place(x = 10, y = frame._current_height -
int(MessageModule.messageStack[messageIndex][1]) - 20)

    if userInput.get() == "cls":
        os._exit(0)

    abstractedInput = NLPS.LanguageAbstractor(userInput.get().lower())

    #Categorisation
    cat = queryCat.Categorise(abstractedInput.split())
    category = cat.FindCategory()

    if category[0] == "command":
        mableWText.instruct(abstractedInput)

    elif category[0] == "question":
        if category[1] == "what" or category[1] == "how":
            response = cc.CheckQuestion(abstractedInput)
            if response == None:
                mableWText.search(abstractedInput)
            else:
                DisplayAIResponse(response.capitalize())
        else:
            mableWText.search(abstractedInput)

    elif category[0] == "conversation":
        response = cc.CheckGreeting(abstractedInput)
        if response == None:
            response = cc.CheckMeet(abstractedInput)
            DisplayAIResponse(response.capitalize())
        else:
            DisplayAIResponse(response.capitalize())

    userInput.delete(0, len(userInput.get()))
```

First, we create a Message label and go through the stack management process as described previously. Then, we have a simple ‘if’ statement to check if the user’s input was ‘cls’ which stands for ‘close’. If so, we close the program.

After that, we call the LanguageAbstractor we talked about to carry out the steps of the NLPS. This newly processed input is then categorised using the categorisation algorithm as described before.

Next, we have a short code block to call the ConversationCheck module's steps to further categorise the input. The 'DisplayAIResponse' function goes through the Stack Management algo but for the resultant response. Finally, after the response has been determined and the correct function has been called, we delete the user's previous query from the text box.

As you can see, there is a function which we haven't talked about which is executed if the query has the 'command' category which is 'instruct'. This function is the one that determines which task to call depending on the input.

The code for this can be seen below:

```
import webbrowser
from datetime import date, datetime
import InstructionLibrary as IL

today = date.today()
Time = datetime.now()
current_time = Time.strftime("%H:%M")

#==================================== Instruction (Command)
====================================
def instruct(Text: str):
    inputText = Text.split()

    for i in range(0, len(inputText)):

        if inputText[i] == "tell" and inputText[i+1] == "me" and
inputText[i+2] == "about":
            Text = Text.replace("tell", "")
            Text = Text.replace("me", "")
            Text = Text.replace("about", "")
            webbrowser.get('C:/Program
Files/Google/Chrome/Application/chrome.exe
%s').open("https://en.wikipedia.org/wiki/"+Text)

        if inputText[i] == "videos" and inputText[i+1] == "by":
            IL.PlayVideosBy(Text.replace("play videos by", ""))

        if inputText[i] == "open":
            for k in range(0, len(inputText)):
                if inputText[k] == 'email' or inputText[k] == 'gmail':
                    IL.OpenMail()
                    break
                else:
                    if k == len(inputText) - 1:
                        IL.OpenProgram(inputText[i+1])
        if inputText[i] == "news":
            IL.GetNews()

        if inputText[i] == "weather":
            IL.GetWeather()

        if inputText[i] == "date":
            IL.GetDate()

        if inputText[i] == "time":
            IL.GetTime()

        if inputText[i] == "my" and inputText[i+1] == "email":
            IL.OpenMail()

        if inputText[i] == "calendar":
            IL.ShowCalendar()

        if inputText[i] == "visit":
            IL.VisitSite(inputText[i+1])

        if inputText[i] == "timer":
            for k in range(0, len(inputText)):
                if inputText[k] == 'minute' or inputText[k] == 'minutes':
                    IL.SetATimer(int(inputText[k - 1]))
                elif inputText[k] == 'hour' or inputText[k] == 'hours':
                    IL.SetATimer(int(inputText[k - 1]) * 60)

def search(query):
    webbrowser.get('C:/Program Files/Google/Chrome/Application/chrome.exe
%s').open("https://www.google.com/search?q="+query+"&oq="+query+"&aqs=chrome..
69i57j35i39j46i199i465i512j0i512j46i199i465i512j69i61l3.1452j0j7&sourceid=chro
me&ie=UTF-8")
```

This function simply goes through the input looking for key words such as ‘calendar’, ‘visit’ and ‘email’ and picks a task from the InstructionLibrary based on the key word. The InstructionLibrary is a module that holds a batch of procedures (the instructions) that the program can complete.

Here’s how it looks like:

```
import webbrowser
import pyautogui
from datetime import date, datetime
import subprocess as program
import os
import TimeModule
import json
```

```python
today = date.today()
Time = datetime.now()
current_time = Time.strftime("%H:%M")

browser = webbrowser.get('C:/Program
Files/Google/Chrome/Application/chrome.exe %s')

programFile = open("Programs.json")
programs = json.load(programFile)

responseFile = open("Settings.json")
responses = json.load(responseFile)

def SetATimer(duration):
    timer = TimeModule.Timer(duration)
    timer.SetTimer()

def OpenProgram(exe):
    program.Popen(programs[exe])

def PlayVideosBy(text: str):
    text = text.replace(" ", "")
    channelPage = f"https://www.youtube.com/@{text}"
    browser.open(channelPage)

def GetNews():
    browser.open("https://www.bbc.co.uk/news")

def GetWeather():
    browser.open("https://www.google.com/search?q=weather&oq=weather&aqs=chrom
e..69i57j35i39j46i199i465i512j0i512j46i199i465i512j69i61l3.1452j0j7&sourceid=c
hrome&ie=UTF-8")

def OpenMail():
    webbrowser.open("https://mail.google.com/mail/u/0/#inbox")

def ShowCalendar():
    pyautogui.keyDown("Win")
    pyautogui.keyDown("alt")
    pyautogui.press("d")
    pyautogui.keyUp("Win")
    pyautogui.keyUp("alt")

def VisitSite(domainName):
    browser.open(f"https://www.{domainName.lower()}.com/")
```

The part of Mable which is voice controlled also works similarly to the instruct function except it doesn’t read from the IL module because it needs to respond with TTS.

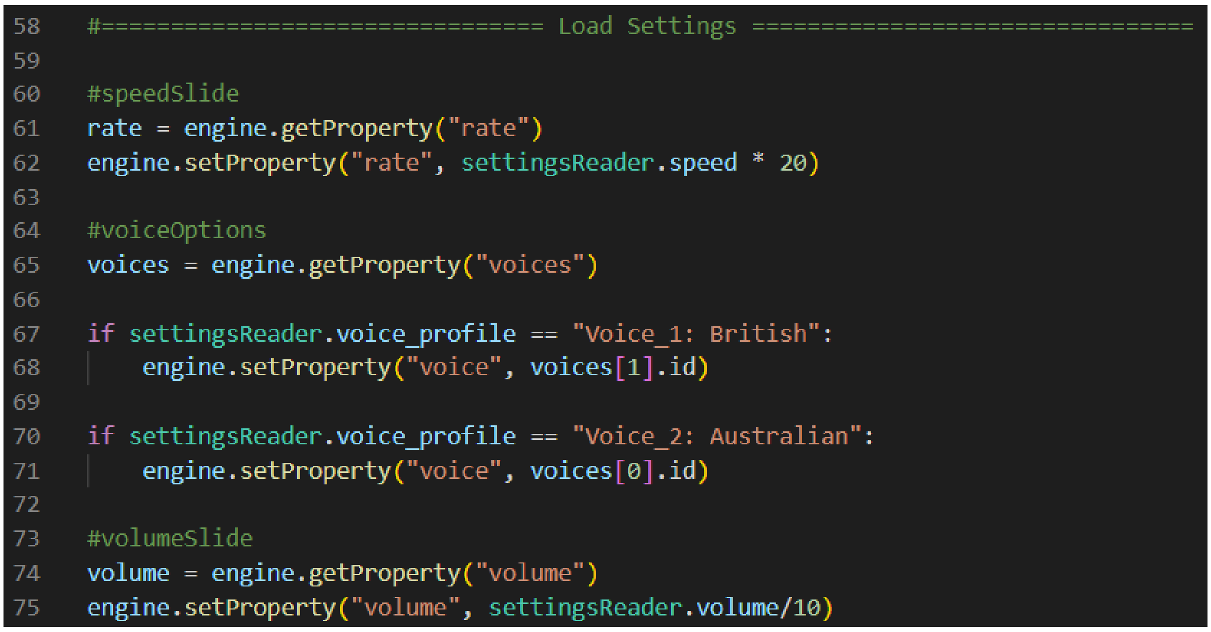
I also implemented another separate NLPS for this but in the end didn't use it since it involves the use of the database which we talked about at the beginning of this section.

```
#================================ Creating NLP System's Class (NLPSC)
================================
class NLPSystem:
    def __init__(self, speech):
        self.speech = speech

    #Tokenisation
    def Tokenise(self):
        self.tokenisedText = self.speech.split()
        print(self.tokenisedText)

    #Stemming
    def Stem(self):
        stemmer = LancasterStemmer()
        for self.word in self.tokenisedText:
            self.stemmedWord = stemmer.stem(self.word)
            print(self.stemmedWord)

    #SWR and Tagging
    def SWRaTag(self):
        self.swrText = []
        self.labels = []
        self.conn = pyodbc.connect(r'Driver={Microsoft Access Driver (*.mdb,
*.accdb)};DBQ=C:\\Users\\akosh\\OneDrive\\Asztali gép\\USB copy\\Python
Folder\\MableAI\\MableDatabase.accdb;')
        self.cursor = self.conn.cursor()

        for i in range(len(self.tokenisedText)):
            self.data = pandas.read_sql(sql = f"select Label from Words where
Word = '{self.tokenisedText[i]}'", con = self.conn)
            if (self.data['Label'] == 'stop_word').all():
                continue
            else:
                self.swrText.append(self.tokenisedText[i])
                self.labels.append(self.data['Label'])
```

The code for this can be seen above.

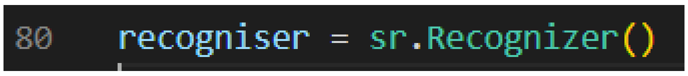
For this part of the code, I used the pyttsx3 and speech_recognition modules. Since this wasn't the primary focus of the project and is only a convenient addition, the code for this isn't too extensive. When this version runs, a 'listen' procedure is triggered which records the audio given by the user and uses the speech_recognition module to convert that data into a string which is then stored. This string is then treated the same as the text input from the text box on the main screen of the UI.

We first load the settings from the Settings.JSON file:



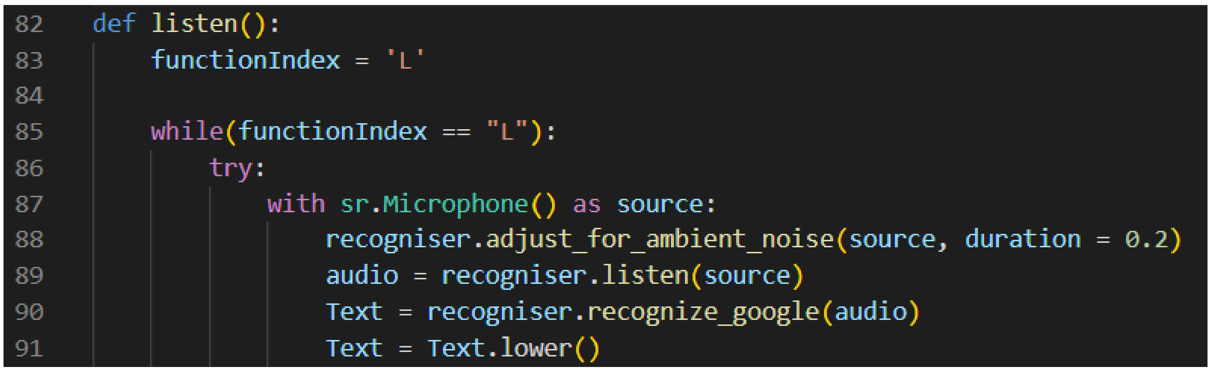
```python
#================================ Load Settings ================================

#speedSlide
rate = engine.getProperty("rate")
engine.setProperty("rate", settingsReader.speed * 20)

#voiceOptions
voices = engine.getProperty("voices")

if settingsReader.voice_profile == "Voice_1: British":
    engine.setProperty("voice", voices[1].id)

if settingsReader.voice_profile == "Voice_2: Australian":
    engine.setProperty("voice", voices[0].id)

#volumeSlide
volume = engine.getProperty("volume")
engine.setProperty("volume", settingsReader.volume/10)
```

Then, we create the speech recogniser object:



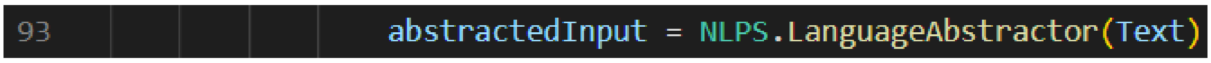
```python
recogniser = sr.Recognizer()
```

Next, the program starts to listen to the audio input using this object when the ‘listen’ function is called:



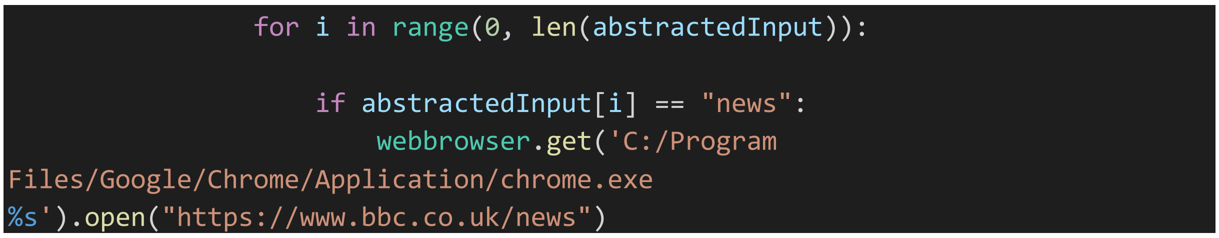
```python
def listen():
    functionIndex = 'L'

    while(functionIndex == "L"):
        try:
            with sr.Microphone() as source:
                recogniser.adjust_for_ambient_noise(source, duration = 0.2)
                audio = recogniser.listen(source)
                Text = recogniser.recognize_google(audio)
                Text = Text.lower()
```

The resulting string is stored as the ‘Text’ variable. Once we get the string back, we call the LanguageAbstractor from the NLPS module:



```python
                abstractedInput = NLPS.LanguageAbstractor(Text)
```

Finally, we loop through the abstracted input, looking for keywords so that we can call the correct function for that word:



```python
                for i in range(0, len(abstractedInput)):

                    if abstractedInput[i] == "news":
                        webbrowser.get('C:/Program 
Files/Google/Chrome/Application/chrome.exe 
%s').open("https://www.bbc.co.uk/news")
```

```
                    if abstractedInput[i] == "weather":
                        webbrowser.get('C:/Program
Files/Google/Chrome/Application/chrome.exe
%s').open("https://www.google.com/search?q=weather&oq=weather&aqs=chrome..69i5
7j35i39j46i199i465i512j0i512j46i199i465i512j69i61l3.1452j0j7&sourceid=chrome&i
e=UTF-8")

                    if abstractedInput[i] == "my" and abstractedInput[i+1] ==
"name" and abstractedInput[i+2] == "is":
                        engine.say("Nice to meet you," + abstractedInput[i+3])
                        engine.runAndWait()

                    if abstractedInput[i] == "your" and abstractedInput[i+1]
== "name":
                        engine.say("My name is Mable")
                        engine.say("What's yours?")
                        engine.runAndWait()
                        functionIndex = "S"

                    if abstractedInput[i] == "date":
                        engine.say("The date is")
                        engine.say(str(today))
                        engine.runAndWait()
                        functionIndex = "S"

                    if abstractedInput[i] == "time":
                        engine.say("The time is")
                        engine.say(current_time)
                        engine.runAndWait()
                        functionIndex = "S"

                    if abstractedInput[i] == "my" and abstractedInput[i+1] ==
"email":
                        engine.say("Ok, opening your email.")
                        engine.runAndWait()
                        webbrowser.open("https://mail.google.com/mail/u/0/#inb
ox")

                        functionIndex = "S"

                    if abstractedInput[i] == "calendar":
                        pyautogui.keyDown("Win")
                        pyautogui.keyDown("alt")
                        pyautogui.press("d")
                        pyautogui.keyUp("Win")
                        pyautogui.keyUp("alt")
                        functionIndex = "S"
                    if abstractedInput[i] == "mable":
                        engine.say("Yes?")
                        engine.runAndWait()
                        functionIndex = "S"

                #Shut down
                if Text == "shut down":
                    engine.say("Ok, shutting down.")
                    engine.runAndWait()
                    functionIndex = "O"

                #Search
                if abstractedInput[0] == "search":
                    Text = Text.replace("search", "")
                    engine.say("Ok, here's what I found.")
                    engine.runAndWait()
                    webbrowser.get('C:/Program
Files/Google/Chrome/Application/chrome.exe
%s').open("https://www.google.com/search?q="+Text+"&oq="+Text+"&aqs=chrome..69
i57j35i39j46i199i465i512j0i512j46i199i465i512j69i61l3.1452j0j7&sourceid=chrome
&ie=UTF-8")
                    functionIndex = "S"

                if abstractedInput[0] == "tell" and abstractedInput[1] == "me"
and abstractedInput[2] == "about":
                    Text = Text.replace("tell", "")
                    Text = Text.replace("me", "")
                    Text = Text.replace("about", "")
                    engine.say("Okay, here's what I found on " + Text)
                    engine.runAndWait()
                    webbrowser.get('C:/Program
Files/Google/Chrome/Application/chrome.exe
%s').open("https://en.wikipedia.org/wiki/"+Text)

        except sr.RequestError as e:
            print("Could not request results; {0}".format(e))
            #engine.say("Request Error.")
            #engine.runAndWait()
            #import WakeMable
            functionIndex = 'O'

        except sr.UnknownValueError:
            print("unknown error occured")
            #engine.say("Unknown Error.")
            #engine.runAndWait()
            #import WakeMable
            functionIndex = 'O'
```

```
    if functionIndex != "L":
        pass
```

We need two function indexes: one for listening, and one for speaking, since if the program responded while it was listening, it would interpret its own speech as a new input.

4.0 Testing

Main Objectives		**User-suggested Tasks**		**Extra Objectives**	
Understand spoken language and text input	-Tokenization: Breaking down of text into smaller semantic units (words) -Part-of-speech-tagging: Labelling words as nouns, verbs, adjectives… -Stemming and lemmatisation: Standardising words by reducing them to their root forms (e.g.: 'went' becomes 'go') -Stop word removal: Filtering out common words that no unique	*It should be able to show the user the weather forecast*	-Search the weather forecast -Tell the user what the program found	*Ability to carry a simple conversation. (no more than a few sentences)*	-Respond to simple sentences such as "What's your name" and "How are you?" -Carry the conversation by responding to the user from a library of sample responses

	information (at, to, a, the…)				
Graphical User Interface (GUI)	-Speech button: This will activate the main loop of the system -Text area: This is where instructions and responses will be displayed -Settings menu: volume, voice, (language?), active times settings (the times between which the program should run in the background)	*It should be able to call/text any of the user's contacts*	-Search for mentioned name in known contacts -Call the desired contact when found	*Voice Recognition*	
Carry out specific tasks when asked by the user	-Recognize instruction -Carry out instruction	*It should be able to play music/videos*	-Open wanted app/site (Spotify, YouTube)	*Multiple languages*	-The user should be able to pick a language in settings and the AI should then switch to that language
Keyword Hierarchy	-Database with keywords linked to the system -Keyword importance (level in the hierarchy)	*It should be able to show you the news*	-Open news sites. (BBC News, CNN…) -Recommend Breaking News		
		It should be able to open programs on your computer	-Search for desired program on the computer -Run the program		
		It should be able to visit websites	-Search the website name in Google		

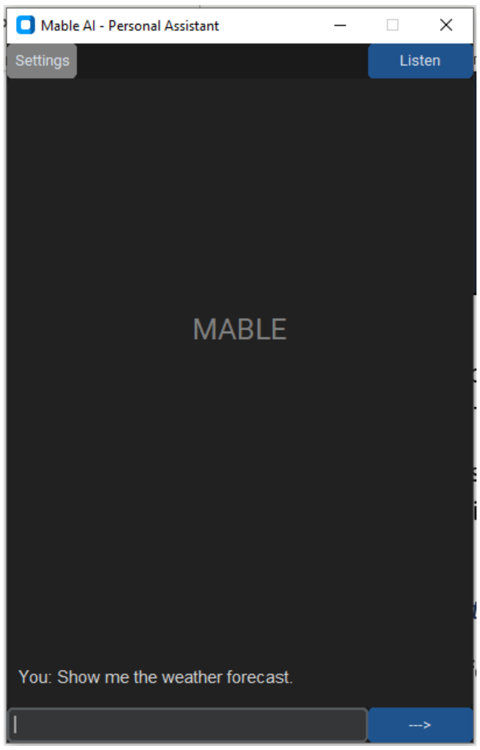
		It should be able to read articles	-Convert text in open article using a Text-to-Speech algorithm -Play the converted file		
		It should be able to set timers, reminders and alarms	-Open timers on the computer -Add a new timer to the app based on the duration specified by the user -Start the new timer		

Above is the table of the project's final objectives from the Analysis section. Here, we will go through them and test each one of these objectives to see if they have been successfully implemented.

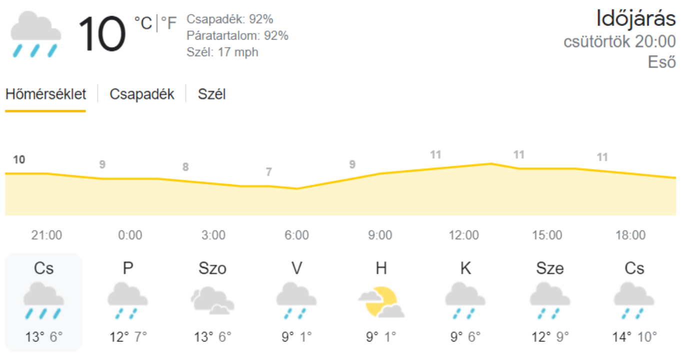
We'll test the User-suggested task since they also test the GUI and the program's ability to understand text input which are two of the main objectives.

Note - The understanding of spoken language can best be demonstrated in the video accompanying this document so we'll not discuss it here.

Looking up the weather forecast:

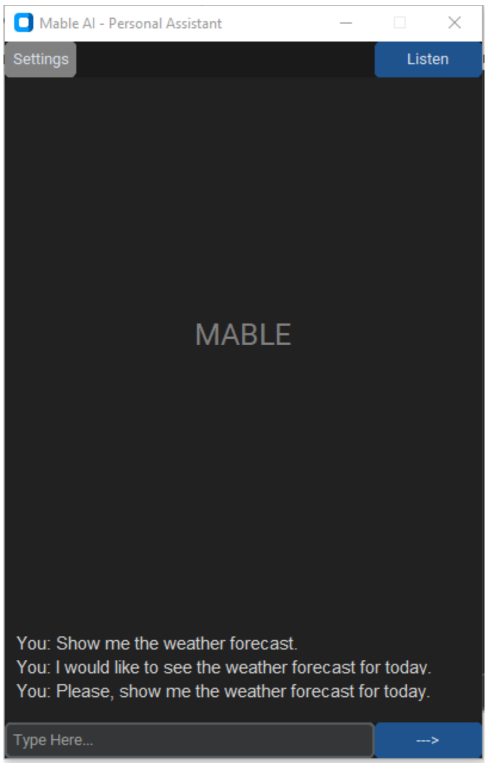


You: Show me the weather forecast.



On the left hand side, you can see my input above the text box and on the right, you can see the program's response. This task has therefore been implemented correctly.

Here are two further queries I tested which received the same response from the program:

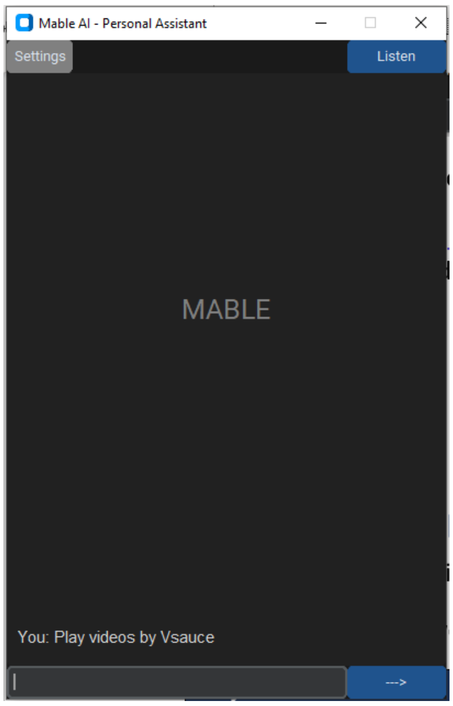


You: Show me the weather forecast.
You: I would like to see the weather forecast for today.
You: Please, show me the weather forecast for today.

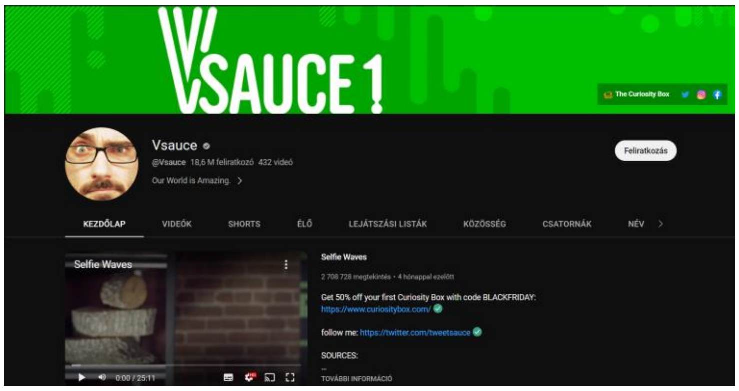
Calling/texting user's contacts:

Unfortunately, I wasn't able to implement this feature due to the strict time limit on the project. However, it could be added relatively easily using an opensource API for a service such as Whatsapp.

Playing videos/music:



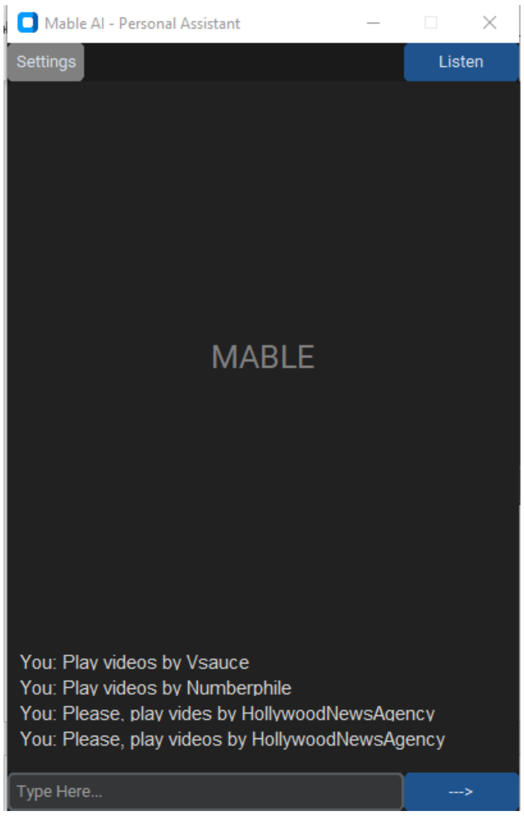
You: Play videos by Vsauce



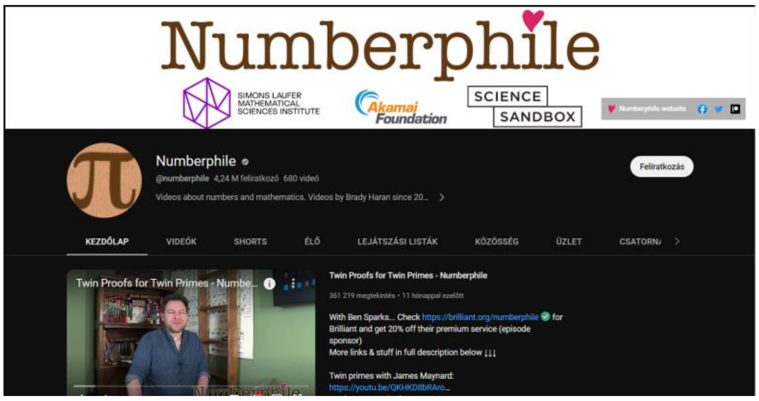
Selfie Waves
2 708 728 megtekintés • 4 hónappal ezelőtt
Get 50% off your first Curiosity Box with code BLACKFRIDAY:
https://www.curiositybox.com/
follow me: https://twitter.com/tweetsauce
SOURCES:
TOVÁBBI INFORMÁCIÓ

Again, on the right you can see my input and on the left, Mable's response. It directs the user to the home page of whichever channel they specify. In this case, Vsauce.

Below are two further queries I made and the program's rsponses to them:

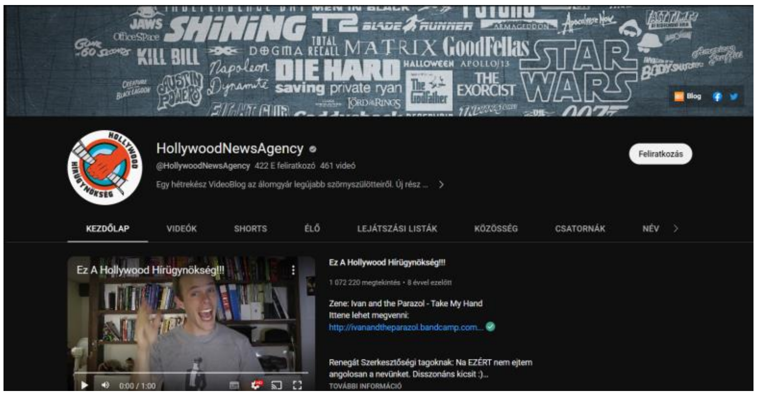


You: Play videos by Vsauce
You: Play videos by Numberphile
You: Please, play vides by HollywoodNewsAgency
You: Please, play videos by HollywoodNewsAgency



Numberphile
@numberphile 4,24 M feliratkozó 680 videó
Videos about numbers and mathematics. Videos by Brady Haran since 20...

Twin Proofs for Twin Primes - Numberphile
361 219 megtekintés • 11 hónappal ezelőtt
With Ben Sparks... Check https://brilliant.org/numberphile for Brilliant and get 20% off their premium service (episode sponsor)
More links & stuff in full description below ↓↓↓
Twin primes with James Maynard:
https://youtu.be/QKHKD8bRAro...



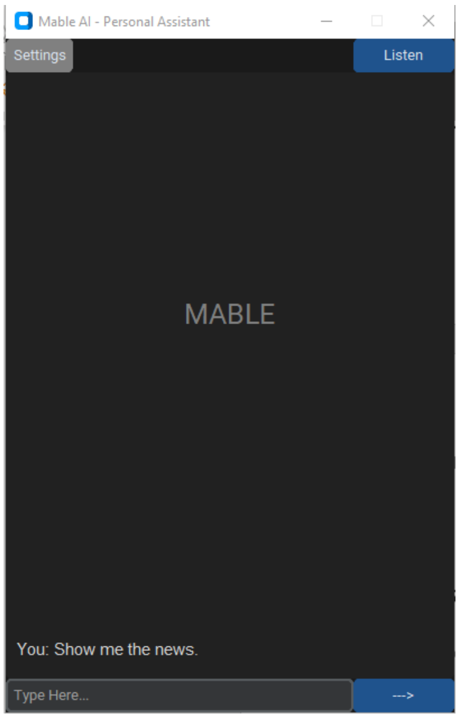
HollywoodNewsAgency
@HollywoodNewsAgency 422 E feliratkozó 461 videó
Egy hétrekész VideoBlog az álomgyár legújabb szörnyszülötteiről. Új rész ...

Ez A Hollywood Hírügynökség!!!
1 072 220 megtekintés • 8 évvel ezelőtt
Zene: Ivan and the Parazol - Take My Hand
Ittene lehet megvenni:
http://ivanandtheparazol.bandcamp.com...

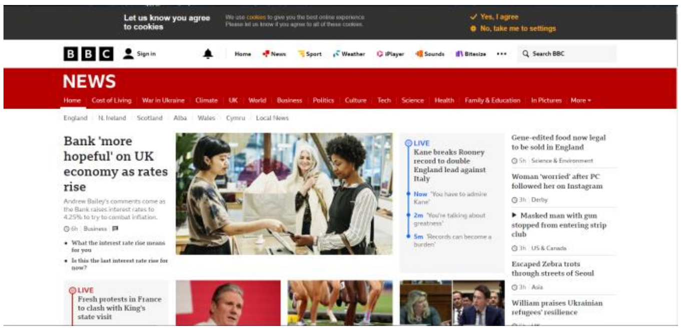
Renegát Szerkesztőségi tagoknak: Na EZÉRT nem ejtem angolosan a nevünket. Disszonáns kicsit :)...
TOVÁBBI INFORMÁCIÓ

As you can see, typos are not autocorrected and so my first request for the HollywoodNewsAgency channel was not processed.

Showing the latest news:



You: Show me the news.



Let us know you agree to cookies
We use cookies to give you the best online experience. Please let us know if you agree to all of these cookies.

Bank 'more hopeful' on UK economy as rates rise
Andrew Bailey's comments come as the Bank raises interest rates to 4.25% to try to combat inflation.
6h Business
- What the interest rate rise means for you
- Is this the last interest rate rise for now?

LIVE
Kane breaks Rooney record to double England lead against Italy
Now 'You have to admire Kane'
2m 'You're talking about greatness'
5m 'Records can become a burden'

Gene-edited food now legal to be sold in England
5h Science & Environment

Woman 'worried' after PC followed her on Instagram
3h Derby

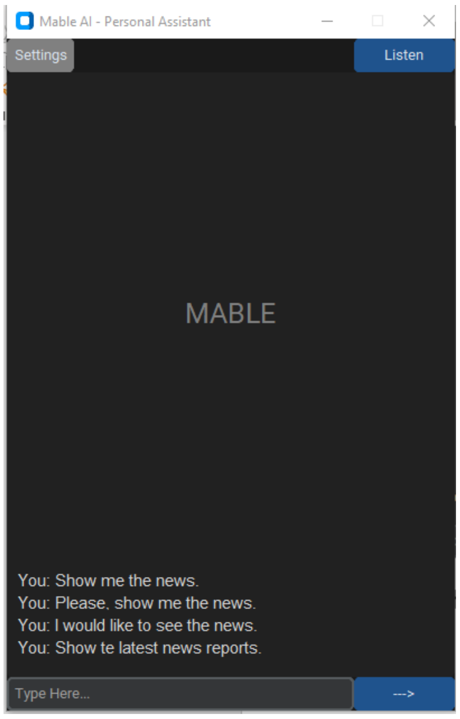
Masked man with gun stopped from entering strip club
3h US & Canada

Escaped Zebra trots through streets of Seoul
3h Asia

William praises Ukrainian refugees' resilience

LIVE
Fresh protests in France to clash with King's state visit

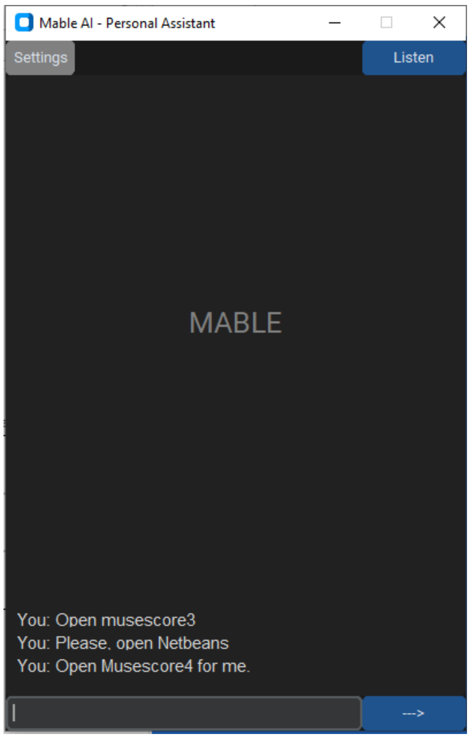
When prompted, the program directs the user to BBC news. Again, this works for multiple variations of the input:



```
You: Show me the news.
You: Please, show me the news.
You: I would like to see the news.
You: Show te latest news reports.
```

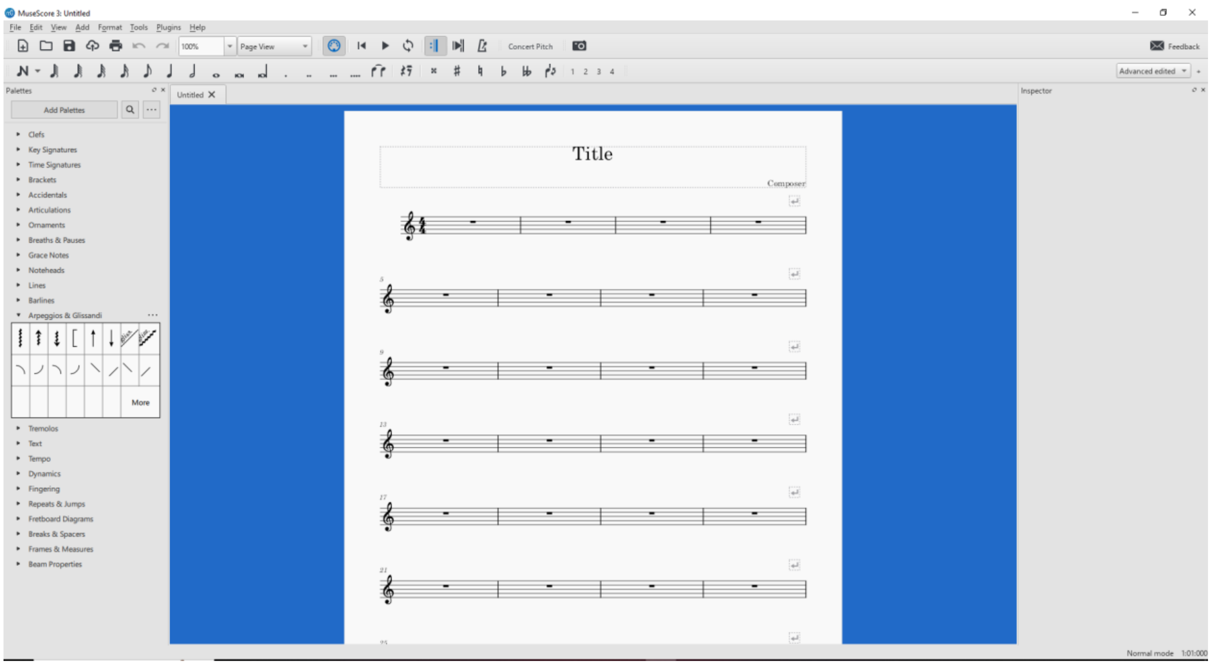
For all of the above inputs, I was directed to the BBC news home page. You may again notice typo on my last query. It's disregarded as the word I mistyped was 'the' which is a stop word which are removed from the input when it's abstracted.

Open programs on the user's device:

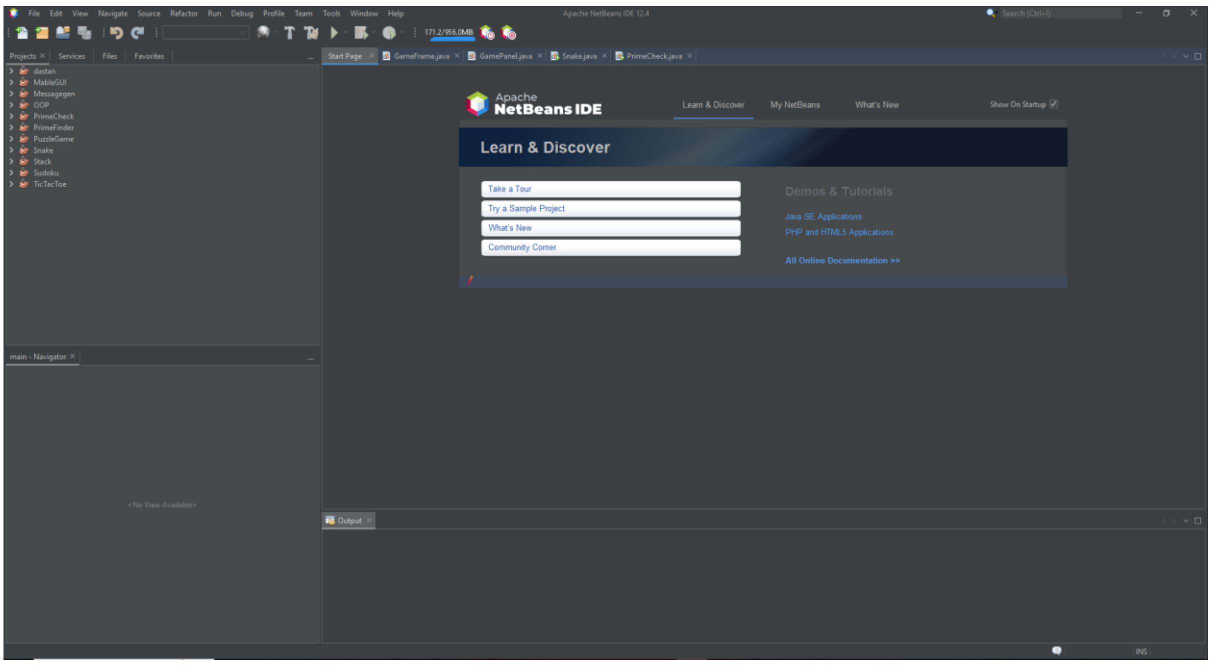


```
You: Open musescore3
You: Please, open Netbeans
You: Open Musescore4 for me.
```

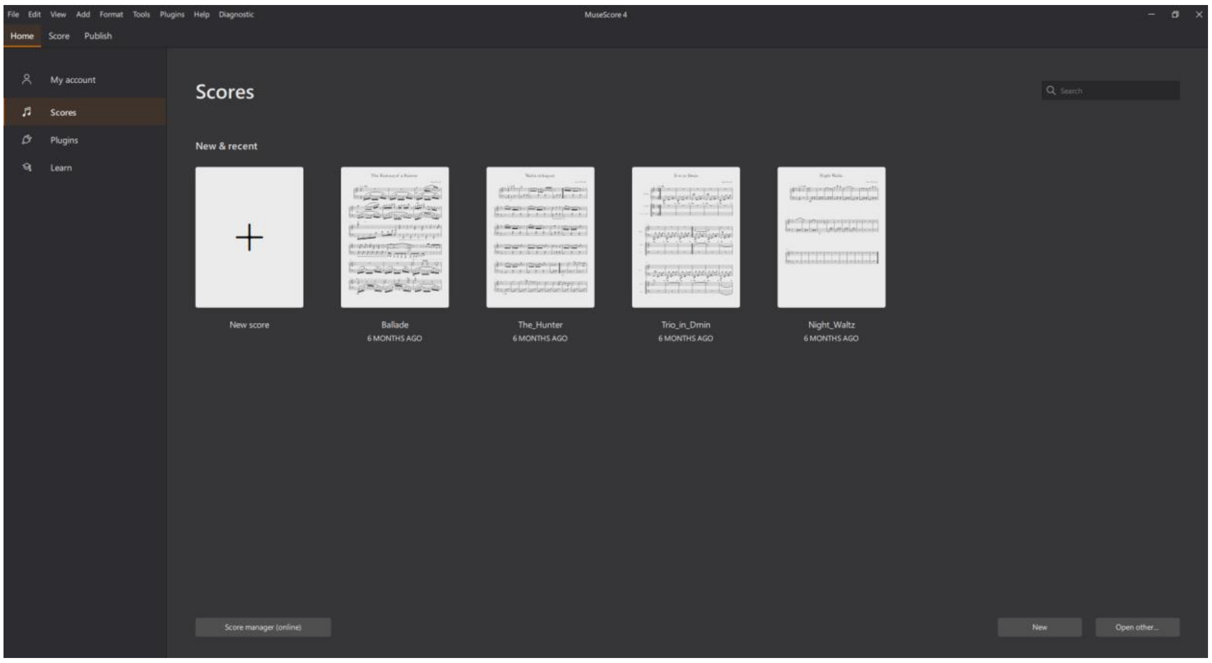
Musescore3:



Netbeans:

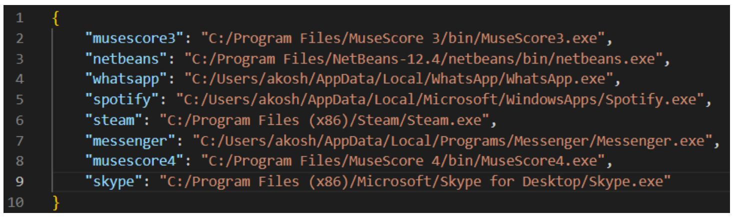

File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Apache NetBeans IDE 12.4
Projects Services Files Favorites
dastan
MableGUI
Messagegen
OOP
PrimeCheck
PrimeFinder
PuzzleGame
Snake
Stack
Sudoku
TicTacToe
Start Page GameFrame.java GamePanel.java Snake.java PrimeCheck.java
Apache NetBeans IDE
Learn & Discover My NetBeans What's New Show On Startup
Learn & Discover
Take a Tour
Try a Sample Project
What's New
Community Corner
Demos & Tutorials
Java SE Applications
PHP and HTML5 Applications
All Online Documentation >>
main - Navigator
<No View Available>
Output

Musescore4:



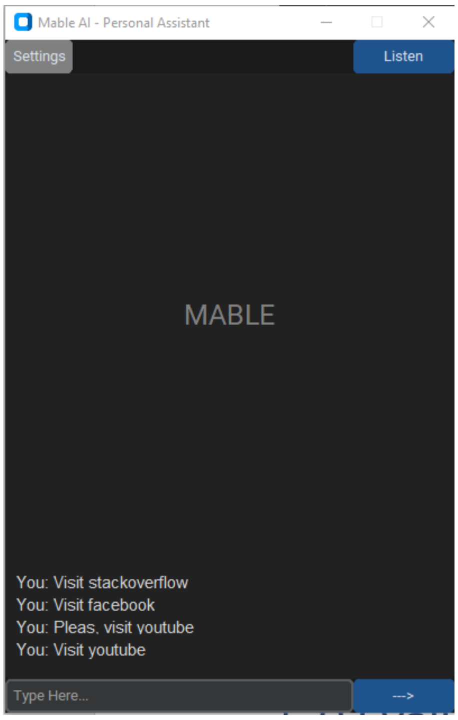
The program successfully understood three variations of the same command for different programs and launched them.

Note – the program can only launch programs which are added to the Programs.JSON file:

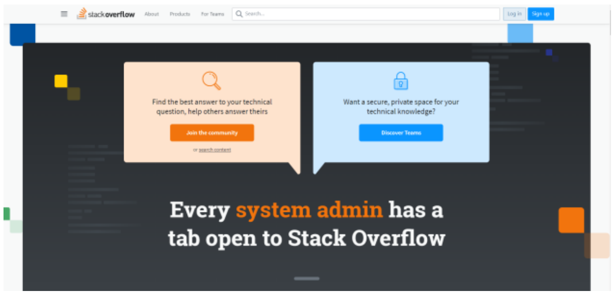

```
{
    "musescore3": "C:/Program Files/MuseScore 3/bin/MuseScore3.exe",
    "netbeans": "C:/Program Files/NetBeans-12.4/netbeans/bin/netbeans.exe",
    "whatsapp": "C:/Users/akosh/AppData/Local/WhatsApp/WhatsApp.exe",
    "spotify": "C:/Users/akosh/AppData/Local/Microsoft/WindowsApps/Spotify.exe",
    "steam": "C:/Program Files (x86)/Steam/Steam.exe",
    "messenger": "C:/Users/akosh/AppData/Local/Programs/Messenger/Messenger.exe",
    "musescore4": "C:/Program Files/MuseScore 4/bin/MuseScore4.exe",
    "skype": "C:/Program Files (x86)/Microsoft/Skype for Desktop/Skype.exe"
}
```

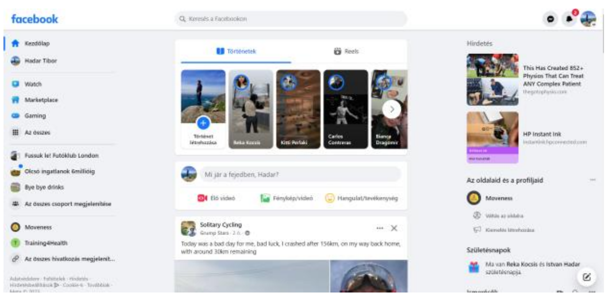
This is a library of programs and their paths on the system.

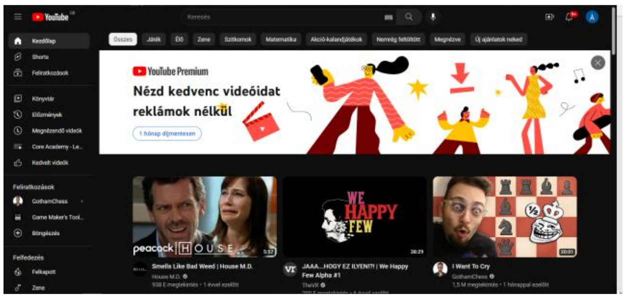
Visiting websites:



You: Visit stackoverflow
You: Visit facebook
You: Pleas, visit youtube
You: Visit youtube


stackoverflow
Find the best answer to your technical question, help others answer theirs
Join the community
Want a secure, private space for your technical knowledge?
Discover Teams
Every system admin has a tab open to Stack Overflow





The code successfully recognises that the ser wants to visit a specific website and directs them to the URL of the main page of that website. This again works irrespective of the style in which the user chooses to enter their query.

Reading articles:

Unfortunately, this feature is not included in this version of the program.

Setting Timers/Reminders/Alarms:

This feature is included to some extent. Only timers can be started by the program.

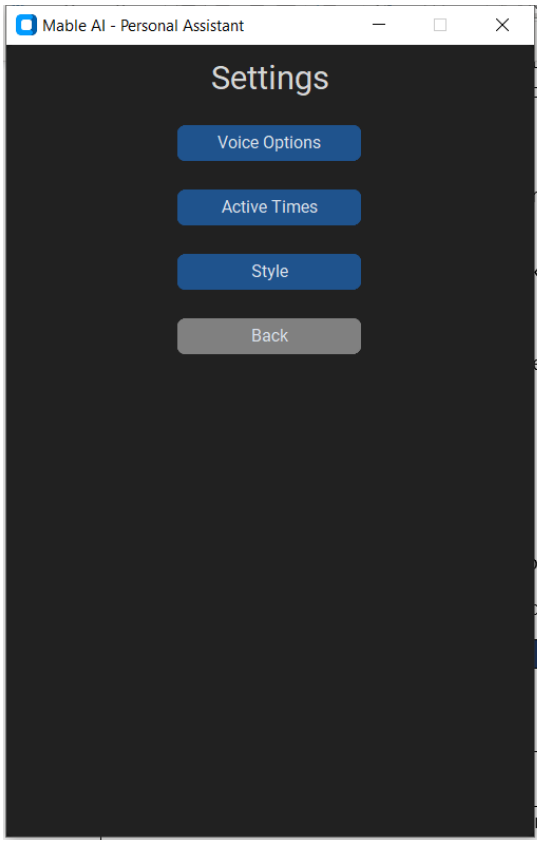
When the user enters a duration to set a timer to, it will scan through the input to get if the time was given in seconds or in minutes. It will then create a Timer object which will start the timer for the specified duration.

However, there is a bug with this part. The user can't do anything on the program while the timer is going. This is because the program is waiting for the set amount of time, so no other function can be triggered during this time.

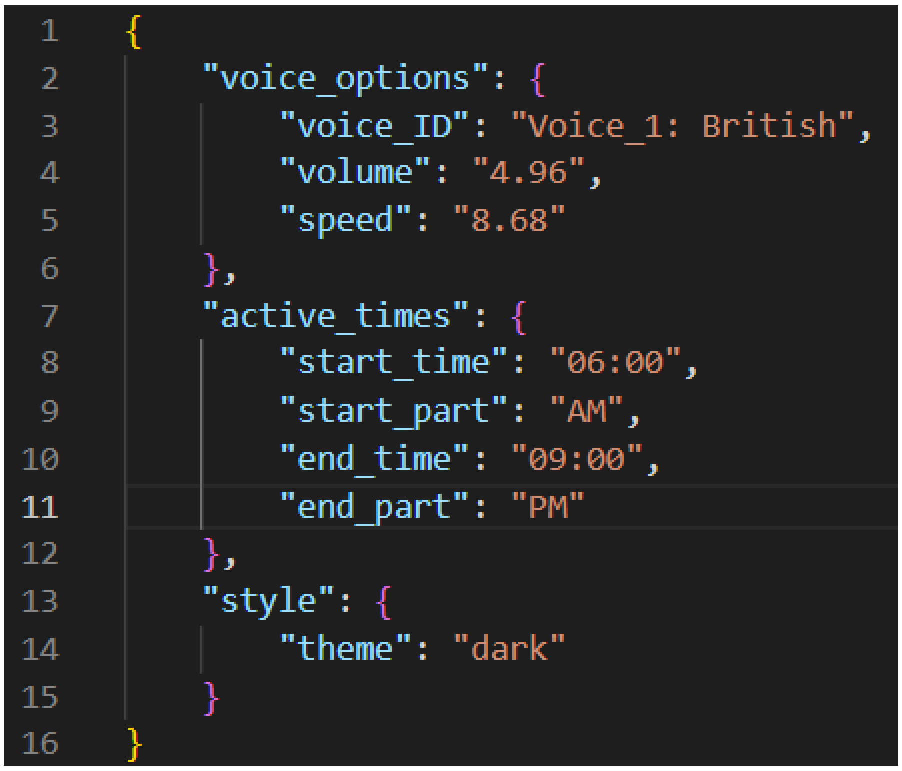
Once the timer is finished, the user is given back control over the GUI.

GUI:

The user can change the settings using the GUI elements.

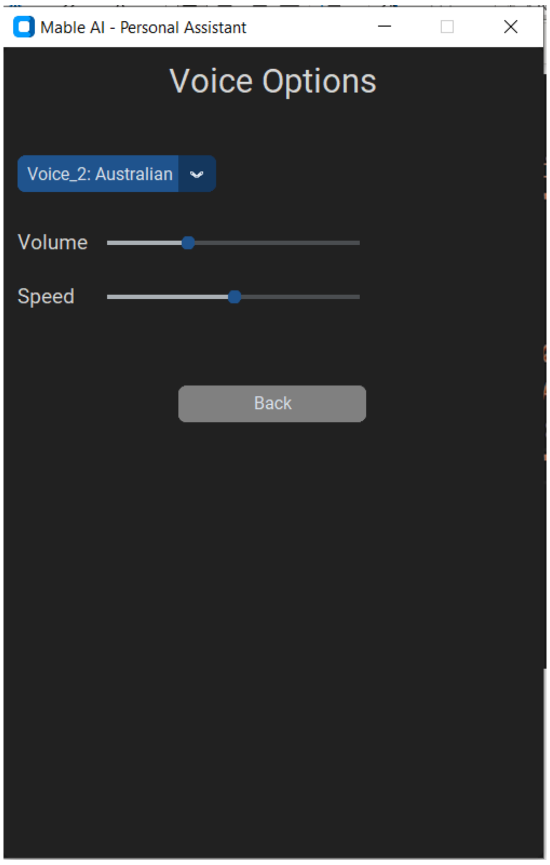


When any of these buttons are clicked, the user is taken to the window of that setting page. Inside those windows, the user can then customise the settings.



```json
{
    "voice_options": {
        "voice_ID": "Voice_1: British",
        "volume": "4.96",
        "speed": "8.68"
    },
    "active_times": {
        "start_time": "06:00",
        "start_part": "AM",
        "end_time": "09:00",
        "end_part": "PM"
    },
    "style": {
        "theme": "dark"
    }
}
```

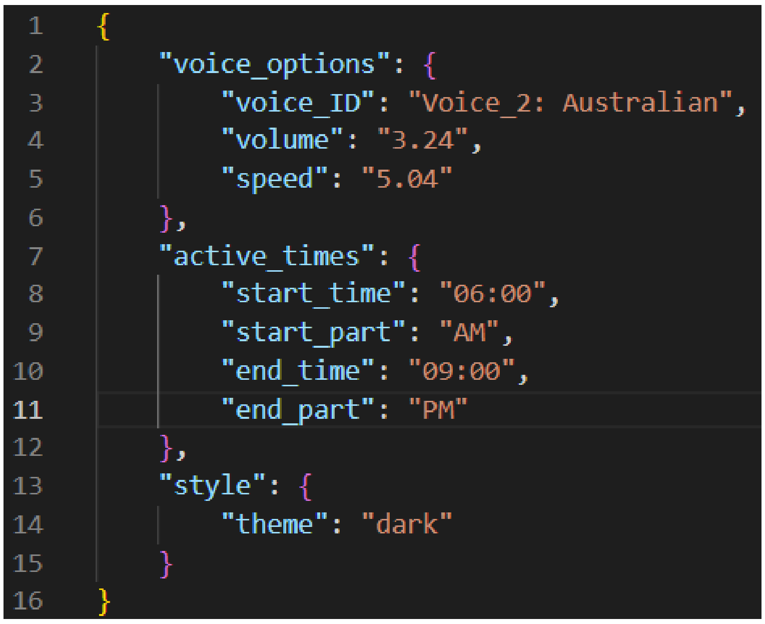
Above are the initial settings. We're going to go through all the settings windows and change them to test if they work as intended.



Voice_2: Australian

Volume

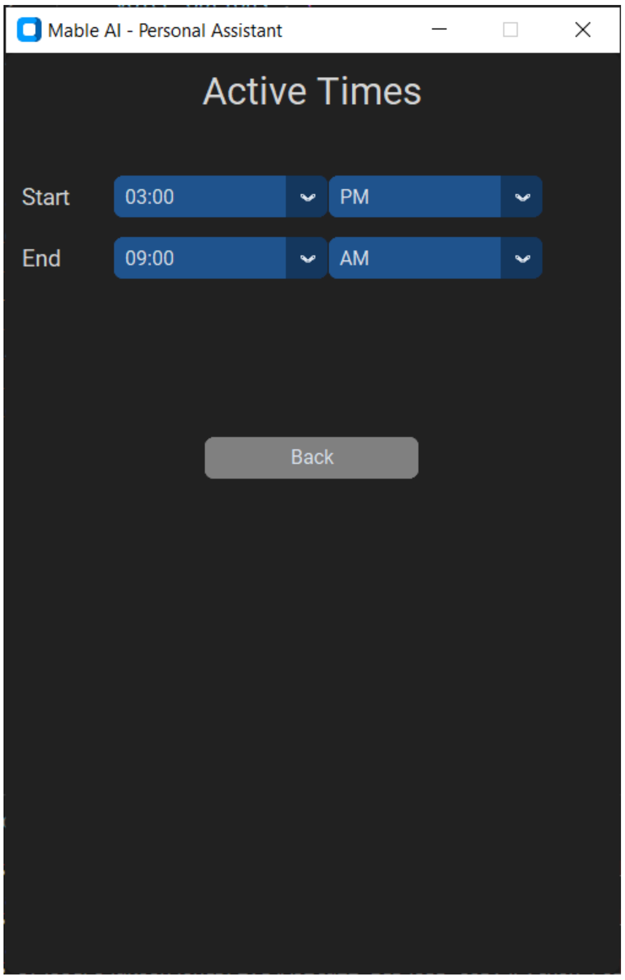
Speed



```json
{
    "voice_options": {
        "voice_ID": "Voice_2: Australian",
        "volume": "3.24",
        "speed": "5.04"
    },
    "active_times": {
        "start_time": "06:00",
        "start_part": "AM",
        "end_time": "09:00",
        "end_part": "PM"
    },
    "style": {
        "theme": "dark"
    }
}
```

Above are the changes made to the Voice settings and the resultant changes made.

Next is the test of the Active times menu.



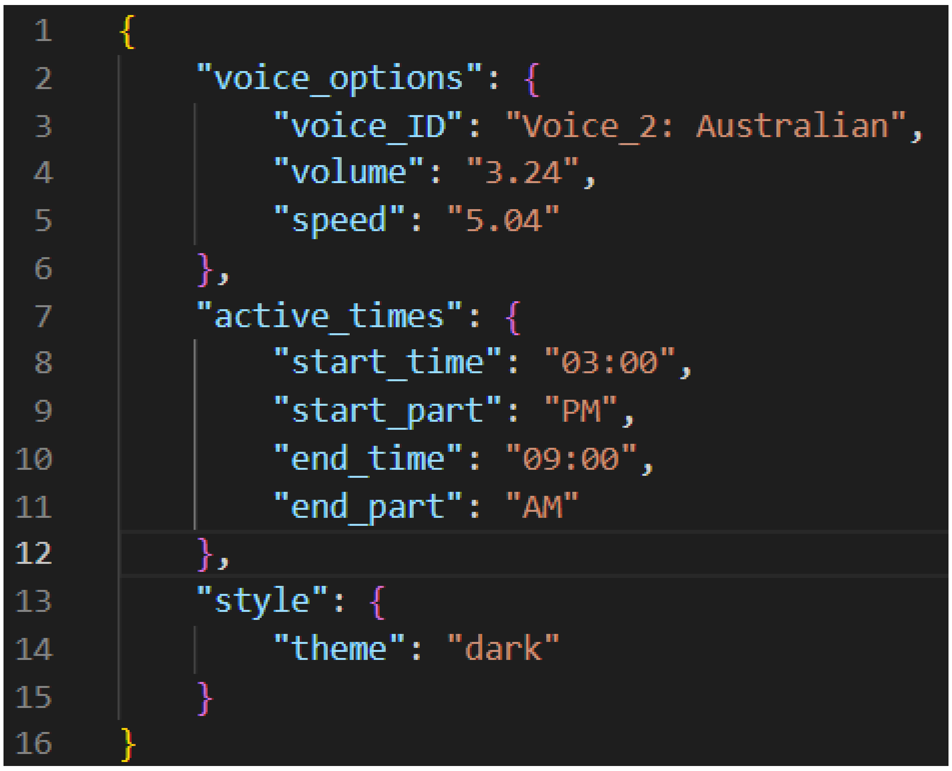
Mable AI - Personal Assistant

Active Times

Start 03:00 PM

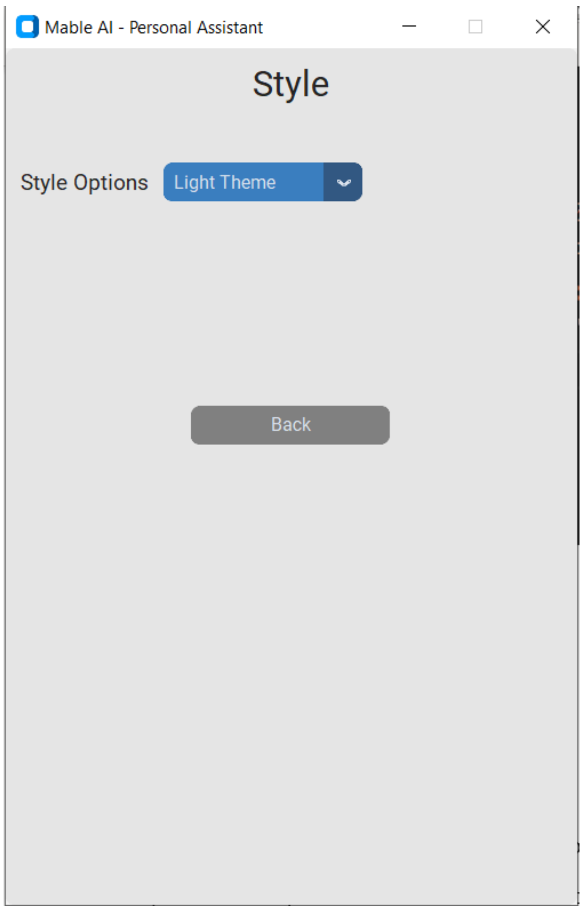
End 09:00 AM

Back



```
{
    "voice_options": {
        "voice_ID": "Voice_2: Australian",
        "volume": "3.24",
        "speed": "5.04"
    },
    "active_times": {
        "start_time": "03:00",
        "start_part": "PM",
        "end_time": "09:00",
        "end_part": "AM"
    },
    "style": {
        "theme": "dark"
    }
}
```

Finally, the Style menu.

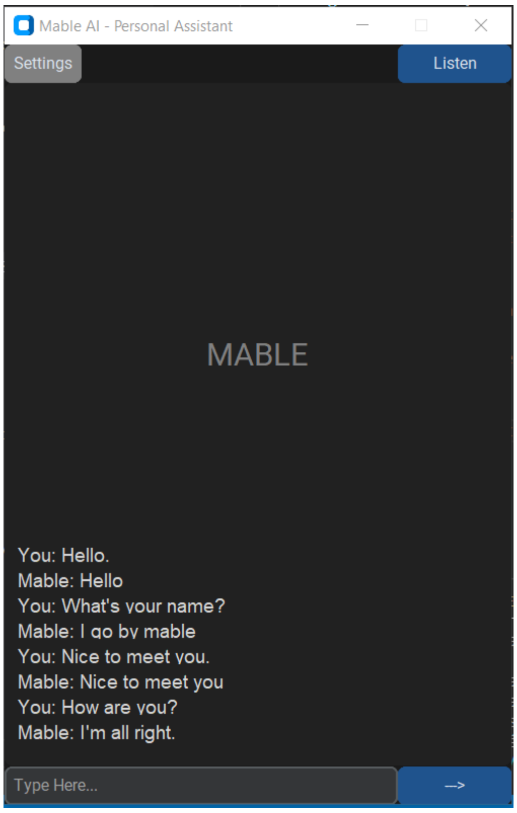


As you can see, all the settings windows changed the data in the Settings.JSON file as intended and therefore, the preferred settings of the user.

Simple Conversation Test:

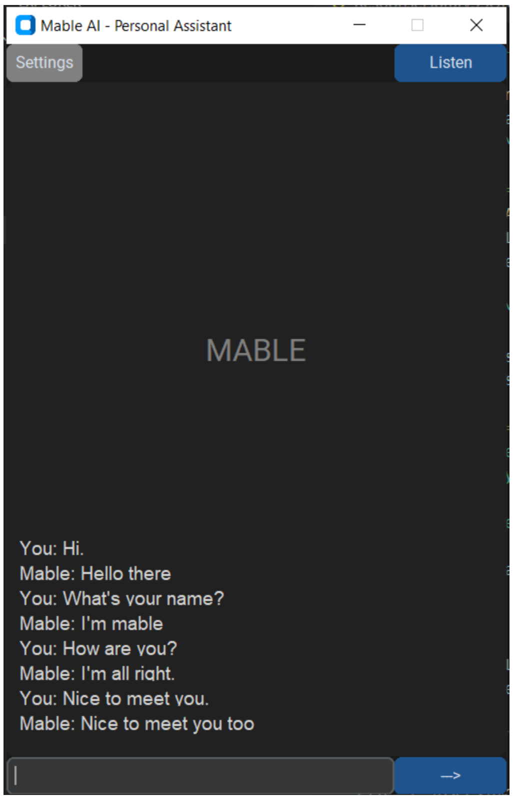
Now, we need to test if Mable can carry a simple Conversation.

First conversation:



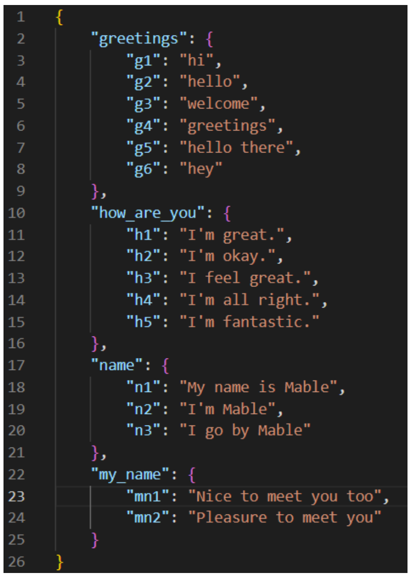
```
You: Hello.
Mable: Hello
You: What's your name?
Mable: I go by mable
You: Nice to meet you.
Mable: Nice to meet you
You: How are you?
Mable: I'm all right.
```

Second conversation:



```
You: Hi.
Mable: Hello there
You: What's your name?
Mable: I'm mable
You: How are you?
Mable: I'm all right.
You: Nice to meet you.
Mable: Nice to meet you too
```

The program will choose a random response from a predefined list of appropriate responses in the Response Library.



```
{
    "greetings": {
        "g1": "hi",
        "g2": "hello",
        "g3": "welcome",
        "g4": "greetings",
        "g5": "hello there",
        "g6": "hey"
    },
    "how_are_you": {
        "h1": "I'm great.",
        "h2": "I'm okay.",
        "h3": "I feel great.",
        "h4": "I'm all right.",
        "h5": "I'm fantastic."
    },
    "name": {
        "n1": "My name is Mable",
        "n2": "I'm Mable",
        "n3": "I go by Mable"
    },
    "my_name": {
        "mn1": "Nice to meet you too",
        "mn2": "Pleasure to meet you"
    }
}
```

5.0 Evaluation

Evidence for the following conclusions will be found in my video demonstrating the final project.

First, the Main Objectives and their level of success (LoS):

Objective	LoS
Understand Spoken Language and Text Input	This objective has been fully met since the program responds to both text and auditory inputs from the user.
GUI	This objective has been fully met since the project's front end is an easy-to-navigate GUI.
Carry out specific tasks when asked by the user	This objective has been fully met since the program carries out the task it has been asked to by the user.
Keyword Hierarchy	This objective was abandoned since it's only needed for a complex chatbot. (See beginning of Technical Solution)

Most of the Main objectives have been successfully implemented in the final project. As we've talked about before, the Keyword Hierarchy and the database are only needed for a project which has the main focus of simulating a real-life conversation.

Next, the User-Suggested Tasks:

Objective	LoS
Show weather forecast	This objective has been fully met.
Call/text contacts	This objective has not been met since the system is not linked to any services such as Whatsapp.
Play music/video	This objective has been fully met.
Show news	This objective has been fully met.
Run programs installed on the user's device	This objective has been fully met.
Visit websites	This objective has been fully met.
Read articles	This objective has not been met.
Set timers/reminders/alarms	This objective has been partially met since the program can only start timers.

Again, the majority of the objectives for this section have been met either fully or partially. In future versions of the program, the features which I have not managed to include in now could be added.

For example, the ability to call or text contacts could be achieved by using an opensource API for an app such as Whatsapp or Messenger.

Finally, the Extra Objectives:

Objective	LoS
Carry a simple conversation	This objective has been fully met.
Voice recognition	This objective has been fully met.
Multiple Languages	This objective has not been met.

In total, **11/15** of the overall objectives and **24/30** of the sub-objectives have been met. This means that **80%** of the initial objectives have been successfully implemented in the final project.

Table of grouped programming concepts used:

Group	Concept	Page
A	Use of a Stack	from 39
A	Hashing Algorithm	from 21
A	JSON files	from 19 and 29
A	Use of a Tree	from 36
A	Mathematical Model	from 21
B	Text files	from 29
C	Single-dimensional arrays	from 29

Appendix

The code:

```
--ConversationCheck.py
import json
import random

conversationFile = open("ResponseLibrary.json")
phrases = json.load(conversationFile)

def CheckGreeting(userInput):
    for i in range(1, len(phrases["greetings"])):
        if userInput == phrases["greetings"][f"g{i}"] + " ":
            index = random.randint(1, len(phrases["greetings"]))
            response = phrases["greetings"][f"g{index}"]
            return response
        else:
            if i == len(phrases["greetings"]):
                return None

def CheckQuestion(userInput):
    if userInput == "what your name ":
        index = random.randint(1, len(phrases["name"]))
        response = phrases["name"][f"n{index}"]
        return response

    elif userInput == "how you ":
        index = random.randint(1, len(phrases["how_are_you"]))
        response = phrases["how_are_you"][f"h{index}"]
        return response
```

```
  else:
      return None


def CheckMeet(userInput):
  if userInput == "nice meet you " or userInput == "pleasure meet you ":
      index = random.randint(1, len(phrases["my_name"]))
      response = phrases["my_name"][f"mn{index}"]
      return response


--InstructionLibrary.py
import webbrowser
import pyautogui
from datetime import date, datetime
import subprocess as program
import os
import TimeModule
import json


today = date.today()
Time = datetime.now()
current_time = Time.strftime("%H:%M")


browser = webbrowser.get('C:/Program Files/Google/Chrome/Application/chrome.exe %s')


programFile = open("Programs.json")
programs = json.load(programFile)


responseFile = open("Settings.json")
responses = json.load(responseFile)
```

```
def SetATimer(duration):
    timer = TimeModule.Timer(duration)
    timer.SetTimer()


def OpenProgram(exe):
    program.Popen(programs[exe])


def PlayVideosBy(text: str):
    text = text.replace(" ", "")
    channelPage = f"https://www.youtube.com/@{text}"
    browser.open(channelPage)


def GetNews():
    browser.open("https://www.bbc.co.uk/news")


def GetWeather():

browser.open("https://www.google.com/search?q=weather&oq=weather&aqs=chrome..69i57j35i3
9j46i199i465i512j0i512j46i199i465i512j69i61l3.1452j0j7&sourceid=chrome&ie=UTF-8")


def GetDate():
    print(f"The date is {str(today)}")


def GetTime():
    print(f"The time is {current_time}")


def OpenMail():
    webbrowser.open("https://mail.google.com/mail/u/0/#inbox")


def ShowCalendar():
    pyautogui.keyDown("Win")
    pyautogui.keyDown("alt")
    pyautogui.press("d")
    pyautogui.keyUp("Win")
    pyautogui.keyUp("alt")

def VisitSite(domainName):
    browser.open(f"https://www.{domainName.lower()}.com/")

--Mable.py
import os
import time
import pyautogui
import webbrowser
import speech_recognition as sr
import pyttsx3
from datetime import date, datetime
import settingsReader
import pyodbc
import pandas
from nltk import LancasterStemmer
import NLPS
import queryCat

#================================ Creating NLP System's Class (NLPSC)
================================
class NLPSystem:
        def __init__(self, speech):
                self.speech = speech

        #Tokenisation
        def Tokenise(self):
                self.tokenisedText = self.speech.split()
                print(self.tokenisedText)


        #Stemming

        def Stem(self):

                stemmer = LancasterStemmer()

                for self.word in self.tokenisedText:

                        self.stemmedWord = stemmer.stem(self.word)

                        print(self.stemmedWord)


        #SWR and Tagging

        def SWRaTag(self):

                self.swrText = []

                self.labels = []

                self.conn = pyodbc.connect(r'Driver={Microsoft Access Driver (*.mdb,
*.accdb)};DBQ=C:\\Users\\akosh\\OneDrive\\Asztali gép\\USB copy\\Python
Folder\\MableAI\\MableDatabase.accdb;')

                self.cursor = self.conn.cursor()


                for i in range(len(self.tokenisedText)):

                        self.data = pandas.read_sql(sql = f"select Label from Words where Word =
'{self.tokenisedText[i]}'", con = self.conn)

                        if (self.data['Label'] == 'stop_word').all():

                                continue

                        else:

                                self.swrText.append(self.tokenisedText[i])

                                self.labels.append(self.data['Label'])


                print(self.swrText)

                print(self.labels)


engine = pyttsx3.init()
```

```
today = date.today()
Time = datetime.now()
current_time = Time.strftime("%H:%M")

functionIndex = "S"

#=============================== Load Settings ================================

#speedSlide
rate = engine.getProperty("rate")
engine.setProperty("rate", settingsReader.speed * 20)

#voiceOptions
voices = engine.getProperty("voices")

if settingsReader.voice_profile == "Voice_1: British":
        engine.setProperty("voice", voices[1].id)

if settingsReader.voice_profile == "Voice_2: Australian":
        engine.setProperty("voice", voices[0].id)

#volumeSlide
volume = engine.getProperty("volume")
engine.setProperty("volume", settingsReader.volume/10)

#engine.say("I'm listening.")
#engine.runAndWait()

recogniser = sr.Recognizer()

def listen():
    functionIndex = 'L'


    while(functionIndex == "L"):
        try:
            with sr.Microphone() as source:
                recogniser.adjust_for_ambient_noise(source, duration = 0.2)
                audio = recogniser.listen(source)
                Text = recogniser.recognize_google(audio)
                Text = Text.lower()


                abstractedInput = NLPS.LanguageAbstractor(Text)


                cat = queryCat.Categorise(Text)
                cat.FindCategory()


                for i in range(0, len(abstractedInput)):


                    if abstractedInput[i] == "news":
                        webbrowser.get('C:/Program
Files/Google/Chrome/Application/chrome.exe %s').open("https://www.bbc.co.uk/news")


                    if abstractedInput[i] == "weather":
                        webbrowser.get('C:/Program
Files/Google/Chrome/Application/chrome.exe
%s').open("https://www.google.com/search?q=weather&oq=weather&aqs=chrome..69i57j35i39j46i
199i465i512j0i512j46i199i465i512j69i61l3.1452j0j7&sourceid=chrome&ie=UTF-8")


                    if abstractedInput[i] == "my" and abstractedInput[i+1] ==
"name" and abstractedInput[i+2] == "is":
                        engine.say("Nice to meet you," +
abstractedInput[i+3])
                        engine.runAndWait()
```

```
                                        if abstractedInput[i] == "your" and abstractedInput[i+1] ==
"name":
                                              engine.say("My name is Mable")
                                              engine.say("What's yours?")
                                              engine.runAndWait()
                                              functionIndex = "S"

                                        if abstractedInput[i] == "date":
                                              engine.say("The date is")
                                              engine.say(str(today))
                                              engine.runAndWait()
                                              functionIndex = "S"

                                        if abstractedInput[i] == "time":
                                              engine.say("The time is")
                                              engine.say(current_time)
                                              engine.runAndWait()
                                              functionIndex = "S"

                                        if abstractedInput[i] == "my" and abstractedInput[i+1] ==
"email":
                                              engine.say("Ok, opening your email.")
                                              engine.runAndWait()

      webbrowser.open("https://mail.google.com/mail/u/0/#inbox")
                                              functionIndex = "S"

                                        if abstractedInput[i] == "calendar":
                                              pyautogui.keyDown("Win")
                                              pyautogui.keyDown("alt")
                                              pyautogui.press("d")
                                              pyautogui.keyUp("Win")
```

```
                        pyautogui.keyUp("alt")
                        functionIndex = "S"

                    if abstractedInput[i] == "mable":
                        engine.say("Yes?")
                        engine.runAndWait()
                        functionIndex = "S"

                #Shut down
                if Text == "shut down":
                    engine.say("Ok, shutting down.")
                    engine.runAndWait()
                    functionIndex = "O"

                #Search
                if abstractedInput[0] == "search":
                    Text = Text.replace("search", "")
                    engine.say("Ok, here's what I found.")
                    engine.runAndWait()
                    webbrowser.get('C:/Program
Files/Google/Chrome/Application/chrome.exe
%s').open("https://www.google.com/search?q="+Text+"&oq="+Text+"&aqs=chrome..69i57j35i39j46
i199i465i512j0i512j46i199i465i512j69i61l3.1452j0j7&sourceid=chrome&ie=UTF-8")
                    functionIndex = "S"

                if abstractedInput[0] == "tell" and abstractedInput[1] == "me" and
abstractedInput[2] == "about":
                    Text = Text.replace("tell", "")
                    Text = Text.replace("me", "")
                    Text = Text.replace("about", "")
                    engine.say("Okay, here's what I found on " + Text)
                    engine.runAndWait()
```

```
                                        webbrowser.get('C:/Program
Files/Google/Chrome/Application/chrome.exe %s').open("https://en.wikipedia.org/wiki/"+Text)


            except sr.RequestError as e:
                print("Could not request results; {0}".format(e))
                #engine.say("Request Error.")
                #engine.runAndWait()
                #import WakeMable
                functionIndex = 'O'


            except sr.UnknownValueError:
                print("unknown error occured")
                #engine.say("Unknown Error.")
                #engine.runAndWait()
                #import WakeMable
                functionIndex = 'O'


        if functionIndex != "L":
            pass


--MableGUI.py
import customtkinter
import settingsReader
import settingsWriter
import os


customtkinter.set_appearance_mode(settingsReader.theme)
customtkinter.set_default_color_theme("dark-blue")


root = customtkinter.CTk()
root.geometry("400x600")
root.title("Mable AI - Personal Assistant")

root.wm_resizable(width = False, height = False)


frame = customtkinter.CTkFrame(master = root)

frame.pack(pady = 30, padx = 0, fill = "both", expand = True)


logoLabel = customtkinter.CTkLabel(master = frame, text = "MABLE", font = ("Roboto", 25),
text_color = "gray")

logoLabel.pack(padx = 50, pady = 200)


#=============================== Save Settings ================================

def writeSettings(self):

   settingsWriter.SaveAll()


def saveSettings(self):

   table = open("table.txt", "a")

   table.truncate(0)

   table.write(voiceOptions.get() + "\n")

   table.write(str(volumeSlide.get()) + "\n")

   table.write(str(speedSlide.get()) + "\n")

   table.write(setStartTime.get() + "\n")

   table.write(setStartPart.get() + "\n")

   table.write(setEndTime.get() + "\n")

   table.write(setEndPart.get() + "\n")

   table.write(styleSelect.get() + "\n")

   table.close()

   writeSettings(self)


#=============================== Input Text Area ================================

userInput = customtkinter.CTkEntry(master = root, placeholder_text = "Type Here...")

userInput.place(x = 0, y = root._current_height - 30)
userInput._set_dimensions(width = 310, height = 30)


#================================= Stack Management Function
=================================

messageStack = []


import MessageModule


def StackManagement():

    newMessage = MessageModule.Message(userInput.get().lower(), 0)

    MessageModule.UpdateMessagePos(messageStack, newMessage)


#================================= Displaying Mable's Responses
=================================

def DisplayAIResponse(response):

    messageLabel = customtkinter.CTkLabel(master = frame, text = "Mable: " + response, font =
("Helvetica", 15))

    messLabelStack.append(messageLabel)


    MessageModule.StackManagement(response.lower())


    for messageIndex in range(0, len(messLabelStack)):

        messLabelStack[messageIndex].place(x = 10, y = frame._current_height -
int(MessageModule.messageStack[messageIndex][1]) - 20)


#================================= Ask Function =================================

import mableWText

import queryCat

import NLPS

import ConversationCheck as cc


messLabelStack = []
```

```
def ask():

    messageLabel = customtkinter.CTkLabel(master = frame, text = "You: " + userInput.get(), font =
("Helvetica", 15))
    messLabelStack.append(messageLabel)

    MessageModule.StackManagement(userInput.get().lower())

    for messageIndex in range(0, len(messLabelStack)):
        messLabelStack[messageIndex].place(x = 10, y = frame._current_height -
int(MessageModule.messageStack[messageIndex][1]) - 20)

    if userInput.get() == "cls":
        os._exit(0)

    abstractedInput = NLPS.LanguageAbstractor(userInput.get().lower())

    #Categorisation
    cat = queryCat.Categorise(abstractedInput.split())
    category = cat.FindCategory()

    if category[0] == "command":
        mableWText.instruct(abstractedInput)

    elif category[0] == "question":
        if category[1] == "what" or category[1] == "how":
            response = cc.CheckQuestion(abstractedInput)
            if response == None:
                mableWText.search(abstractedInput)
            else:
                DisplayAIResponse(response.capitalize())
        else:
            mableWText.search(abstractedInput)

    elif category[0] == "conversation":
        response = cc.CheckGreeting(abstractedInput)
        if response == None:
            response = cc.CheckMeet(abstractedInput)
            DisplayAIResponse(response.capitalize())
        else:
            DisplayAIResponse(response.capitalize())

    userInput.delete(0, len(userInput.get()))

askButton = customtkinter.CTkButton(master = root, text = "--->", command = ask)
askButton.place(x = 310, y = root._current_height - 30)
askButton._set_dimensions(width = 90, height = 30)

#==================================== Listen Button
====================================
import Mable

def listen():
    Mable.listen()

listenButton = customtkinter.CTkButton(master = root, text = "Listen", command = listen)
listenButton.place(x = root._current_width - 90, y = 0)
listenButton._set_dimensions(width = 90, height = 30)

#============================== Settings Window Def
==============================
def settingsGUI():
    settingsFrame.pack(pady = 0, padx = 0, fill = "both", expand = True)
    frame.pack_forget()


#================================= Voice Options Def ================================
def voiceOptionsGUI():
    voiceFrame.pack(pady = 0, padx = 0, fill = "both", expand = True)
    settingsFrame.pack_forget()


#================================= Active Times Def ================================
def activeTimesGUI():
    activeTimesFrame.pack(pady = 0, padx = 0, fill = "both", expand = True)
    settingsFrame.pack_forget()


#================================= Style Def ================================
def styleGUI():
    styleFrame.pack(pady = 0, padx = 0, fill = "both", expand = True)
    settingsFrame.pack_forget()


settingsButton = customtkinter.CTkButton(master = root, text = "Settings", command = settingsGUI,
fg_color = "gray")
settingsButton.place(x = 0, y = 0)
settingsButton._set_dimensions(width = 45, height = 30)


#================================ Settings Frame ================================
def backToFrame():
    frame.pack(pady = 0, padx = 0, fill = "both", expand = True)
    settingsFrame.pack_forget()


settingsFrame = customtkinter.CTkFrame(master = root)


settingsLabel = customtkinter.CTkLabel(master = settingsFrame, text = "Settings", font = ("Roboto",
25))
settingsLabel.pack(padx = 12, pady = 10)
```

```
voiceOptions = customtkinter.CTkButton(master = settingsFrame, text = "Voice Options", command
= voiceOptionsGUI)
voiceOptions.pack(padx = 12, pady = 10)

operationTimes = customtkinter.CTkButton(master = settingsFrame, text = "Active Times", command
= activeTimesGUI)
operationTimes.pack(padx = 12, pady = 10)

styleOptions = customtkinter.CTkButton(master = settingsFrame, text = "Style", command =
styleGUI)
styleOptions.pack(padx = 12, pady = 10)

backToMain = customtkinter.CTkButton(master = settingsFrame, text = "Back", fg_color = "gray",
command = backToFrame)
backToMain.pack(padx = 12, pady = 10)

#=============================== Voice Options Frame
===============================
def backToSettV():
   settingsFrame.pack(pady = 0, padx = 0, fill = "both", expand = True)
   voiceFrame.pack_forget()

voiceFrame = customtkinter.CTkFrame(master = root)

settingsLabel = customtkinter.CTkLabel(master = voiceFrame, text = "Voice Options", font =
("Roboto", 25))
settingsLabel.pack(padx = 12, pady = 10)

voiceOptions = customtkinter.CTkOptionMenu(master = voiceFrame, values = ["Voice_1: British",
"Voice_2: Australian"], command = saveSettings)
voiceOptions.place(x = 10, y = 80)
voiceOptions.set(settingsReader.voice_profile)
```

```
volumeLabel = customtkinter.CTkLabel(master = voiceFrame, text = "Volume", font = ("Roboto", 15))

volumeLabel.place(x = 10, y = 130)


volumeSlide = customtkinter.CTkSlider(master = voiceFrame, from_ = 0, to = 10, width = 200, height
= 10, command = saveSettings)

volumeSlide.place(x = 70, y = 140)

volumeSlide.set(settingsReader.volume)


speedLabel = customtkinter.CTkLabel(master = voiceFrame, text = "Speed", font = ("Roboto", 15))

speedLabel.place(x = 10, y = 170)


speedSlide = customtkinter.CTkSlider(master = voiceFrame, from_ = 0, to = 10, width = 200, height =
10, command = saveSettings)

speedSlide.place(x = 70, y = 180)

speedSlide.set(settingsReader.speed)


backToSettings = customtkinter.CTkButton(master = voiceFrame, text = "Back", fg_color = "gray",
command = backToSettV)

backToSettings.pack(padx = 12, pady = 200)


#=============================== Active Times Frame ===============================

def backToSettAT():

    settingsFrame.pack(pady = 0, padx = 0, fill = "both", expand = True)

    activeTimesFrame.pack_forget()


timeOptions = ['01:00', '02:00', '03:00', '04:00', '05:00' ,'06:00' ,'07:00', '08:00', '09:00', '10:00',
'11:00', '12:00']

dayPartOptions = ['AM', 'PM']


activeTimesFrame = customtkinter.CTkFrame(master = root)
```

```
activeTimesLabel = customtkinter.CTkLabel(master = activeTimesFrame, text = "Active Times", font = ("Roboto", 25))

activeTimesLabel.pack(padx = 12, pady = 10)


backToSettingsAT = customtkinter.CTkButton(master = activeTimesFrame, text = "Back", fg_color = "gray", command = backToSettAT)

backToSettingsAT.pack(padx = 12, pady = 200)


startTimeLabel = customtkinter.CTkLabel(master = activeTimesFrame, text = "Start", font = ("Roboto", 15))

startTimeLabel.place(x = 10, y = 80)


endTimeLabel = customtkinter.CTkLabel(master = activeTimesFrame, text = "End", font = ("Roboto", 15))

endTimeLabel.place(x = 10, y = 120)


setStartTime = customtkinter.CTkOptionMenu(master = activeTimesFrame, values = timeOptions, command = saveSettings)

setStartTime.place(x = 70, y = 80)

setStartTime.set(settingsReader.start)


setStartPart = customtkinter.CTkOptionMenu(master = activeTimesFrame, values = dayPartOptions, command = saveSettings)

setStartPart.place(x = 210, y = 80)

setStartPart.set(settingsReader.spart)


setEndTime = customtkinter.CTkOptionMenu(master = activeTimesFrame, values = timeOptions, command = saveSettings)

setEndTime.place(x = 70, y = 120)

setEndTime.set(settingsReader.end)


setEndPart = customtkinter.CTkOptionMenu(master = activeTimesFrame, values = dayPartOptions, command = saveSettings)

setEndPart.place(x = 210, y = 120)
setEndPart.set(settingsReader.epart)


#==================================== Style Frame
====================================

def backToSettS():
    settingsFrame.pack(pady = 0, padx = 0, fill = "both", expand = True)
    styleFrame.pack_forget()


def changeStyle(self):
    if styleSelect.get() == "Dark Theme":
        customtkinter.set_appearance_mode("dark")


    if styleSelect.get() == "Light Theme":
        customtkinter.set_appearance_mode("light")


    saveSettings(self)


styleFrame = customtkinter.CTkFrame(master = root)


styleLabel = customtkinter.CTkLabel(master = styleFrame, text = "Style", font = ("Roboto", 25))
styleLabel.pack(padx = 12, pady = 10)


backToSettingsS = customtkinter.CTkButton(master = styleFrame, text = "Back", fg_color = "gray",
command = backToSettS)
backToSettingsS.pack(padx = 12, pady = 200)


styleSelectLabel = customtkinter.CTkLabel(master = styleFrame, text = "Style Options", font =
("Roboto", 15))
styleSelectLabel.place(x = 10, y = 80)


styleSelect = customtkinter.CTkOptionMenu(master = styleFrame, values = ["Dark Theme", "Light
Theme"], command = changeStyle)
```

```
styleSelect.place(x = 110, y = 80)


if settingsReader.theme == "dark":
    styleSelect.set("Dark Theme")


if settingsReader.theme == "light":
    styleSelect.set("Light Theme")


root.mainloop()


--MableWText.py
import webbrowser
from datetime import date, datetime
import InstructionLibrary as IL


today = date.today()
Time = datetime.now()
current_time = Time.strftime("%H:%M")


#================================== Instruction (Command)
==================================
def instruct(Text: str):
    inputText = Text.split()


    for i in range(0, len(inputText)):


        if inputText[i] == "tell" and inputText[i+1] == "me" and inputText[i+2] == "about":
            Text = Text.replace("tell", "")
            Text = Text.replace("me", "")
            Text = Text.replace("about", "")
            print("Okay, here's what I found on " + Text)
            webbrowser.get('C:/Program Files/Google/Chrome/Application/chrome.exe
%s').open("https://en.wikipedia.org/wiki/"+Text)

        if inputText[i] == "videos" and inputText[i+1] == "by":
            IL.PlayVideosBy(Text.replace("play videos by", ""))

        if inputText[i] == "open":
            for k in range(0, len(inputText)):
                if inputText[k] == 'email' or inputText[k] == 'gmail':
                    IL.OpenMail()
                    break
                else:
                    if k == len(inputText) - 1:
                        IL.OpenProgram(inputText[i+1])

        if inputText[i] == "news":
            IL.GetNews()

        if inputText[i] == "weather":
            IL.GetWeather()

        if inputText[i] == "my" and inputText[i+1] == "name" and inputText[i+2] == "is":
            print("Nice to meet you, " + inputText[i+3])

        if inputText[i] == "your" and inputText[i+1] == "name":
            IL.SayName()

        if inputText[i] == "date":
            IL.GetDate()

        if inputText[i] == "time":
            IL.GetTime()

        if inputText[i] == "my" and inputText[i+1] == "email":
            IL.OpenMail()

        if inputText[i] == "calendar":
            IL.ShowCalendar()

        if inputText[i] == "visit":
            IL.VisitSite(inputText[i+1])

        if inputText[i] == "timer":
            for k in range(0, len(inputText)):
                if inputText[k] == 'minute' or inputText[k] == 'minutes':
                    IL.SetATimer(int(inputText[k - 1]))
                elif inputText[k] == 'hour' or inputText[k] == 'hours':
                    IL.SetATimer(int(inputText[k - 1]) * 60)

def search(query):
    webbrowser.get('C:/Program Files/Google/Chrome/Application/chrome.exe
%s').open("https://www.google.com/search?q="+query+"&oq="+query+"&aqs=chrome..69i57j35i39
j46i199i465i512j0i512j46i199i465i512j69i61l3.1452j0j7&sourceid=chrome&ie=UTF-8")

--MessageModule.py
unitSize = 20

class Message:
    def __init__(self, text, pos):
        self.pos = pos
        self.text = text

def UpdateMessagePos(Stack, newMess):
    Stack.append([str(newMess.text), newMess.pos])

    for messageIndex in range(len(Stack)):
        Stack[messageIndex][1] = str((len(Stack) - messageIndex) * unitSize)

messageStack = []

def StackManagement(text):
    newMessage = Message(text, 0)
    UpdateMessagePos(messageStack, newMessage)

--NLPS.py
#=================================== NLP System
===================================

#Remove Special Characters
def CutSymbols(speech: str) -> str:
    symbols = [',','.','!','?',':',';']

    for i in range(0, len(symbols)):
        speech = speech.replace(symbols[i], "")

    return speech

#Tokenisation
def Tokenise(speech: str) -> list[str]:
    tokenisedText = speech.split()

    return tokenisedText

#Simplify Ask Words
```

```
askWords = ["what", "where", "how", "who", "why", "when", "is", "are", "was", "were"]


def AskWordSimplifier(array) -> str:
    for i in range(0, len(array)):
        for j in range(0, len(askWords)):
            if array[i] == askWords[j] + "'s" or array[i] == askWords[j] + "'re" or array[i] == askWords[j] +
"'d":
                array[i] = askWords[j]


    awsText = ""


    for word in array:
        awsText += word + " "


    return awsText


#Stop Word Removal
stopWords = ["and", "to", "is", "am", "are", "at", "a", "the", "with", "an", "be", "do", "please"]


def RemoveStopWords(tokenisedText: list[str]) -> list[str]:
    tokString = ""


    for i in range(0, len(stopWords)):
        for j in range(0 , len(tokenisedText)):
            if stopWords[i] == tokenisedText[j]:
                tokenisedText[j] = ""


    for word in tokenisedText:
        if word != "":
            tokString += word + " "
    tokenisedText = tokString.split()


    return tokenisedText


#Main Abstractor
def LanguageAbstractor(InputFromUser: str) -> list[str]:
    abstractedOutput =
AskWordSimplifier(RemoveStopWords(Tokenise(CutSymbols(InputFromUser))))


    return abstractedOutput


--QueryCat.py
lass Categorise:
    def __init__(self, nlpCompleteInput):
        self.nlpCompleteInput = nlpCompleteInput
        self.queryCategory = ["conversation", ""]
        self.askWords = ["what", "where", "how", "who", "why", "when", "is", "are", "was", "were"]
        self.commandWords = ["open", "search", "show", "tell", "play", "find", "start", "visit"]


    def FindCategory(self):
        for self.word in self.askWords:
            if self.nlpCompleteInput[0] == self.word:
                self.queryCategory = ["question", self.word]


        for self.word in self.commandWords:
            if self.nlpCompleteInput[0] == self.word:
                self.queryCategory = ["command", self.word]


        return self.queryCategory
```

```
--settingsReader.py
import json

#=============================== Load Settings ================================
sourceFile = open("Settings.json")
settings = json.load(sourceFile)

voice = settings["voice_options"]
activity = settings["active_times"]
style = settings["style"]

#Voice
voice_profile = voice["voice_ID"]
volume = float(voice["volume"])
speed = float(voice["speed"])

#Activity
start = activity["start_time"]
spart = activity["start_part"]
end = activity["end_time"]
epart = activity["end_part"]

#Style
theme = style["theme"]

--settingsWriter.py
import json

def SaveAll():
   table = open("table.txt")
   settings = table.readlines()
```

```
with open('Settings.json', 'r+') as source:
    data = json.load(source)

    #==========================================Save Voice Options
    data["voice_options"]["voice_ID"] = settings[0].rstrip("\n")
    source.seek(0)
    json.dump(data, source, indent = 4)
    source.truncate()

    data["voice_options"]["volume"] = settings[1].rstrip("\n")
    source.seek(0)
    json.dump(data, source, indent = 4)
    source.truncate()

    data["voice_options"]["speed"] = settings[2].rstrip("\n")
    source.seek(0)
    json.dump(data, source, indent = 4)
    source.truncate()

    #==========================================Save Active Times
    data["active_times"]["start_time"] = settings[3].rstrip("\n")
    source.seek(0)
    json.dump(data, source, indent = 4)
    source.truncate()

    data["active_times"]["start_part"] = settings[4].rstrip("\n")
    source.seek(0)
    json.dump(data, source, indent = 4)
    source.truncate()
    data["active_times"]["end_time"] = settings[5].rstrip("\n")
    source.seek(0)
    json.dump(data, source, indent = 4)
    source.truncate()

    data["active_times"]["end_part"] = settings[6].rstrip("\n")
    source.seek(0)
    json.dump(data, source, indent = 4)
    source.truncate()

    #==========================================Save Style
    if settings[7].rstrip("\n")  == "Dark Theme":
        data["style"]["theme"] = "dark"
        source.seek(0)
        json.dump(data, source, indent = 4)
        source.truncate()

    if settings[7].rstrip("\n")  == "Light Theme":
        data["style"]["theme"] = "light"
        source.seek(0)
        json.dump(data, source, indent = 4)
        source.truncate()
```

--TimeManagementSystem.py

```
import time
from datetime import datetime
import settingsReader
import os

checkStart = True
checkEnd = False
```

```
currentTime = f"%H:{'00'}"

if settingsReader.spart == "PM":
    startTime = str(int(settingsReader.start.replace(':00', '')) + 12) + ':00'
else:
    startTime = settingsReader.start

if settingsReader.epart == "PM":
    endTime = str(int(settingsReader.end.replace(':00', '')) + 12) + ':00'
else:
    endTime = settingsReader.end

while checkStart == True:

    Time = datetime.now()

    if Time.strftime(currentTime) == startTime:
        import Mable
        checkStart = False
        checkEnd = True

    time.sleep(60)

while checkEnd == True:

    Time = datetime.now()

    if Time.strftime(currentTime).replace(':', '') == endTime:
        Mable.functionIndex = "O"
        checkEnd = False
    time.sleep(60)


--TimeModule.py
import time
import pyttsx3
import settingsReader


engine = pyttsx3.init()


#speedSlide
rate = engine.getProperty("rate")
engine.setProperty("rate", settingsReader.speed * 20)


#voiceOptions
voices = engine.getProperty("voices")


if settingsReader.voice_profile == "Voice_1: British":
        engine.setProperty("voice", voices[1].id)


if settingsReader.voice_profile == "Voice_2: Australian":
        engine.setProperty("voice", voices[0].id)


#volumeSlide
volume = engine.getProperty("volume")
engine.setProperty("volume", settingsReader.volume/10)


class Timer:
    def __init__(self, waitTime):
        self.waitTime = waitTime
```

```
def SetTimer(self):
    engine.say(f"Okay. Timer set for {self.waitTime} minutes.")
    engine.runAndWait()


    time.sleep(60 * self.waitTime)


    engine.say("Time! Your timer has finished.")
    engine.runAndWait()
```