**Voice Assistant Script Tutorial**

* **Introduction**

This tutorial provides a comprehensive explanation of a Python script that implements a Voice Assistant using various libraries and APIs. The script utilizes functionalities for speech-to-text (STT) and text-to-speech (TTS) conversions, wake-up word detection, and integration with a chatbot. The tutorial will cover the objectives of the script, the libraries used, and provide a detailed explanation of each line of code.

* **Script Overview**

The Voice Assistant script is divided into several parts:

1. Importing Required Libraries
2. Class Definition: VoiceAssistant
3. Function: open\_mic
4. Function: on\_close
5. Function: wake\_check
6. Function: tts
7. Function: stt
8. Function: gui\_setup
9. Function: run
10. Main Execution

Now, let's dive into each section of the script and explain its purpose and functionality.

**1. Importing Required Libraries**

The script begins by importing the necessary libraries:



Here's a brief explanation of each imported library:

* **tkinter**: Used for creating the graphical user interface (GUI) of the voice assistant.
* **os**: Provides a way to interact with the operating system, used for environment variables and file operations.
* **pyaudio**: Allows audio recording and playback.
* **google.cloud.texttospeech**: Enables text-to-speech conversion using the Google Cloud Text-to-Speech API.
* **pygame**: Used for playing audio files.
* **glob**: Provides a way to retrieve files matching a specified pattern.
* **google.cloud.speech**: Allows speech recognition using the Google Cloud Speech-to-Text API.
* **io**: Used for working with input/output operations, particularly reading audio files.
* **sounddevice**: Enables audio recording.
* **numpy**: Provides support for working with arrays and numerical operations.
* **soundfile**: Used for writing audio data to a file.
* **concurrent.futures**: Provides a high-level interface for asynchronously executing functions using threads or processes.
* **pvporcupine**: Enables wake-up word detection.
* **time**: Used for adding delays.
* **botConnecter**: Module responsible for connecting to a chatbot that will be discussed in later docs.

**2. Class Definition: VoiceAssistant**

The script defines a class named **VoiceAssistant** that encapsulates the functionality of the voice assistant. Here's the class definition:



Let's explain the attributes of the **VoiceAssistant** class:

* **detection**: A boolean flag indicating whether a wake-up word has been detected.
* **msg**: A string to store the recognized speech message.
* **screen\_width** and **screen\_height**: Variables to hold the dimensions of the GUI window.
* **label**: A reference to a label widget in the GUI.
* **wake\_label**: A reference to a label widget displaying the wake-up word detection status.
* **generate\_button**: A reference to a button widget used to generate speech.
* **executor**: A **ThreadPoolExecutor** instance to manage concurrent execution of functions.
* **is\_closed**: A boolean flag indicating whether the application is closed.
* **root**: A reference to the main GUI window.
* **response\_message**: A string to store the response message from the chatbot.
* **speech\_label**: A reference to a label widget displaying the speech messages.

The **\_\_init\_\_** method initializes these attributes to their default values.

**3. Function: open\_mic**

The **open\_mic** function is responsible for reopening the microphone. Here's its definition:



Explanation:

* If a wake-up word has been detected (**self.detection** is **True**), it updates the speech label in the GUI to show "Opening Mic" and triggers the generation of speech by invoking the **generate\_button**'s command.
* If no wake-up word has been detected, it prints "Machine is idle" to the console.

**4. Function: on\_close**

The **on\_close** function is called when the application is closed. Here's its definition:



Explanation:

* It prints "Closing..." to the console.
* Sets **self.is\_closed** to **True** to indicate the application is closed.
* Shuts down the **executor**.
* Destroys the root GUI window.
* Prints "Application closed" to the console.
* Exits the program with a status code of **0**

**5. Function: wake\_check**

The **wake\_check** function handles wake-up word detection. Here's its definition:





Explanation:

* It sets the **keyword\_path** and **access\_key** variables for wake-up word detection.
* Initializes the **self.detection** flag as **False to insure no failure when calling wake check multiple times**.
* Defines an audio callback function (**audio\_callback**) that processes the incoming audio data.
* Creates a Porcupine keyword detection handle using the specified keyword path and access key.
* Initializes a PyAudio instance.
* Opens an audio stream for recording using the PyAudio library.
* Starts the audio stream and waits for the wake-up word detection loop to set **self.detection** to **True**.
* Stops and closes the audio stream.
* Terminates the PyAudio instance.
* Prints "All CLEAR" and the value of **self.detection** to the console.
* Initializes the Pygame mixer and plays the audio response files note that a dummy file is a 1 second of a silence audio to avoid pygame locking on a single audio.
* Waits until the audio playback is finished.
* Calls the **open\_mic** function to open the microphone.
* This function is from Porcupine-picovoice which are responsible for training and developing such code. Follow the link for their docs to acknowledge their whereabouts. <https://picovoice.ai/docs/quick-start/porcupine-python/>
* You can also train a custom wake up word here. <https://console.picovoice.ai/ppn> of course you need to setup an account for using the key. Make sure when creating a wakeup word to select the appropriate requirements for your system.

**6. Function: tts**

The **tts** function performs text-to-speech conversion. Here's its definition:



Explanation:

* Sets the environment variable **GOOGLE\_APPLICATION\_CREDENTIALS** to the TTS credentials file path.
* Creates a **TextToSpeechClient** instance.
* Prepares the text to be synthesized by extracting it from the **speech\_label** widget.
* Sets the voice selection parameters, language code, and SSML gender.
* Specifies the audio configuration, setting the encoding to MP3.
* Calls the **synthesize\_speech** method of the **TextToSpeechClient** to generate the synthesized speech.
* Saves the synthesized speech as an audio file.
* Initializes the Pygame mixer and loads a dummy audio file.
* Removes previous audio files to avoid cluttering the directory.
* Generates a unique audio file name and saves the synthesized speech.
* Plays the synthesized speech using Pygame mixer.
* Waits until the audio playback is finished.
* Determines the next action based on the input type. If it requires opening the microphone, it calls **open\_mic**. Otherwise, it resets **self.detection** to **False** and calls **wake\_check**.
* You can read more about it in the google docs. <https://codelabs.developers.google.com/codelabs/cloud-text-speech-python3#:~:text=1.-,Overview,formatting%2C%20and%20other%20pronunciation%20instructions.>

**7. Function: stt**

The **stt** function performs speech-to-text conversion. Here's its definition:





Explanation:

* Checks if a wake-up word has been detected (**self.detection** is **True**).
* Sets the environment variable **GOOGLE\_APPLICATION\_CREDENTIALS** to the STT credentials file path.
* Sets the sample rate, duration, and silence threshold for audio recording.
* Initializes variables to hold the recorded audio chunks, silence counter, and the final recording.
* Starts recording audio in chunks and checks for consecutive silence.
* Adds each recorded chunk to the list of chunks.
* Stops recording when a specified duration of consecutive silence is detected.
* Concatenates all recorded chunks into a single recording.
* Normalizes the recording to the 16-bit range and converts it to 16-bit data.
* Saves the recording as a WAV file.
* Creates a **SpeechClient** instance.
* Reads the audio file content.
* Defines the recognition audio and configuration.
* Performs speech recognition using the Google Cloud Speech-to-Text API.
* If no results are returned, sets **self.detection** to **False**, updates the speech label in the GUI, and calls **wake\_check**.
* Retrieves the recognized speech message from the API response and updates the speech label in the GUI.
* Calls the **main** function from **botConnecter** to obtain the response message from the chatbot.
* Updates the speech label in the GUI with the response message.
* Calls the **tts** function to convert the response message to speech.
* For a thorough and complete docs check out <https://codelabs.developers.google.com/codelabs/cloud-speech-text-python3#1>

**8. Function: gui\_setup**

The **gui\_setup** function sets up the graphical user interface (GUI) for the voice assistant. Here's its definition:



Explanation:

* Creates the root GUI window using **tk.Tk()**.
* Retrieves the screen width and height to set the window dimensions.
* Packs a frame widget within the root window and sets its width and height.
* Creates and packs labels and a button widget within the frame.
* Configures the button's height and width.
* Sets the close protocol of the root window to call the **on\_close** function.
* Starts the main event loop of the GUI.
* Note that this part of code can be altered the preference of your choice so its not a static code.

**9. Function: run**

The **run** function is responsible for starting the voice assistant. Here's its definition:



Explanation:

* Submits the **wake\_check**, **botConnecter.connectToBot**, and **gui\_setup** functions to the **executor** for concurrent execution.
* Returns the futures of the submitted functions.
* This function is very important since it can run multiple processes together at once and on a different thread. Note that you can retrieve the results of each process alone and work on the results.

**10. Main Execution**

The script's main execution section creates an instance of the **VoiceAssistant** class and calls the **run** method. Here's the main section:



Explanation:

* Creates an instance of the **VoiceAssistant** class.
* Calls the **run** method of the **assistant** instance.

**Conclusion:**

This concludes the tutorial document explaining the functionalities and objectives of the provided Python script. Each line of code has been explained along with its purpose and role in the voice assistant application.

**Python Script Tutorial: Direct Line API Helper**

**Objective:**

The objective of this Python script is to provide a tutorial on how to use the Direct Line API Helper to connect to a bot, send messages, and receive responses. The script utilizes the DirectLineAPI class from the direct\_line\_api\_helper module.

**Script Overview:**

**Th**e script consists of several functions and global variables:

1. **connectToBot():** This function establishes a connection to the bot using the Direct Line API.
2. **main(x):** This function sends a message x to the bot, receives a response, and processes the response accordingly.
3. **InputType():** This function returns the current state of the input detection.

**Script Walkthrough:**

1. **Importing the necessary module:**

from direct\_line\_api\_helper import DirectLineAPI

**Explanation:**

The script begins by importing the DirectLineAPI class from the direct\_line\_api\_helper module. This class provides the necessary functionalities for connecting to the bot and sending/receiving messages.

1. **Global Variable Initialization:**

expecting\_input\_detection = None

**Explanation:**

The global variable expecting\_input\_detection is initialized as None. Its purpose will be explained later in the tutorial.

1. **Connecting to the Bot:**

def connectToBot():

    global api

    api=DirectLineAPI("Ha6KfOZ08lE.nwDOAXzn8UhRuIsuX0YBjw37RAKPOLmEm3WWBTpb5uM")

    api.set\_headers()

    api.start\_conversation()

**Explanation:**

The connectToBot() function establishes a connection to the bot using the Direct Line API. It initializes an instance of the DirectLineAPI class with a secret token. The set\_headers() method sets the necessary headers for the API requests, and the start\_conversation() method starts the conversation with the bot.

1. **Sending and Receiving Messages:**

def main(x):

    global expecting\_input\_detection

    print("THE MESSAGE SENT IS",x)

    api.send\_message(x)

    print("recieving the message")

    response1 = api.get\_messages()

    print(response1)

    if response1['inputHint'] == 'acceptingInput':

        print("ACCEPTING INPUT: THE MESSAGE IS:" + response1['text'])

        intent = response1.get('intent')

        entities = response1.get('entities')

        if intent and entities:

            full\_entity = ""

            for key, value in entities.items():

                full\_entity += key + " " + str(value) + " "

            print(intent, " " + full\_entity)

        expecting\_input\_detection = False

        return response1['text']

    else:

        expecting\_input\_detection = True

        print("EXPECTING INPUT: THE MESSAGE IS:" + response1['text'])

        return response1['text']

**Explanation:**

The main(x) function is responsible for sending a message x to the bot, receiving the response, and processing it accordingly. It follows the steps below:

* Prints the message being sent (x) for reference.
* Calls the send\_message(x) method of the api instance to send the message to the bot.
* Prints a message indicating the reception of the response.
* Retrieves the response using the get\_messages() method of the api instance.
* Checks if the input hint of the response is set to 'acceptingInput'. If true, it means the bot expects user input.
  + Prints the received message and extracts the intent and entities (if available) from the response.
  + Sets the expecting\_input\_detection variable to False.
  + Returns the received text from the response.
* If the input hint is not set to 'acceptingInput', it means the bot is not expecting user input.
  + Sets the expecting\_input\_detection variable to True.
  + Prints the received message.
  + Returns the received text from the response.

1. **Input Detection State:**

def InputType():

    global expecting\_input\_detection

    return expecting\_input\_detection

**Explanation**:

The InputType() function returns the current state of the input detection by returning the value of the expecting\_input\_detection variable.

**Conclusion:**

This Python script provides a basic framework for connecting to a bot, sending messages, and receiving responses using the Direct Line API. The tutorial document explains the script's objectives, provides an overview of the script structure, and offers a step-by-step explanation of each function and line of code.

**Python Script Tutorial: DirectLineAPI**

**Overview**

The **DirectLineAPI** class is a Python script that provides a simplified interface for interacting with the Direct Line API of the Bot Framework. It allows you to start a conversation with a bot, send messages to the bot, and retrieve the bot's responses. This tutorial will guide you through the functionalities and objectives of each function and explain the purpose of each line of code in the script.

The Link for the full Microsoft documentation: <https://learn.microsoft.com/en-us/azure/bot-service/rest-api/bot-framework-rest-direct-line-3-0-concepts?view=azure-bot-service-4.0>

**Objectives**

The objectives of this script are as follows:

1. Establish a connection with the Direct Line API using a secret key.
2. Start a conversation with a bot and obtain a conversation ID.
3. Send messages to the bot and receive responses.
4. Retrieve the bot's response and extract relevant information.

**Script Breakdown**

* 1. **Importing Libraries**

import json

import  requests

The script begins by importing the necessary libraries. The **json** library is used for working with JSON data, and the **requests** library is used to send HTTP requests.

* 1. **DirectLineAPI Class**

class DirectLineAPI(object):

    def \_\_init\_\_(self, direct\_line\_secret):

        self.direct\_line\_secret = direct\_line\_secret

        self.base\_url = 'https://directline.botframework.com/v3/directline'

        self.headers = self.set\_headers()

        self.last\_message\_id = None

        self.response = None

**Explanation:**

The DirectLineAPI class is defined, which encapsulates the functionalities related to the Direct Line API. The \_\_init\_\_ method initializes the class with the provided direct\_line\_secret, sets the base URL for the API, and initializes variables for headers, last message ID, and response.

* 1. **Setting Headers**

    def set\_headers(self):

        headers = {'Content-Type': 'application/json'}

        value = ' '.join(['Bearer', self.direct\_line\_secret])

        headers.update({'Authorization': value})

        return headers

**Explanation:**

The **set\_headers** method is responsible for setting the required headers for the API requests. It creates a dictionary **headers** with the 'Content-Type' key and the value 'application/json'. It then appends the authorization token to the headers by joining the 'Bearer' keyword with the **direct\_line\_secret**. The updated headers dictionary is returned.

* 1. **Starting a Conversation**

    def start\_conversation(self):

        url = '/'.join([self.base\_url, 'conversations'])

        bot\_response = requests.post(url, headers=self.headers)

        json\_response = bot\_response.json()

        if 'error' in json\_response:

            print("Conversation ID not available [request failed]")

            return None

        else:

            print("succed")

            self.conversationid = json\_response['conversationId']

            return self.conversationid

**Explanation:**

The start\_conversation method initiates a conversation with the bot. It constructs the URL by joining the base\_url and 'conversations' using the / separator. The requests.post method is then used to send a POST request to the URL with the specified headers. The response is parsed as JSON.

If the JSON response contains an 'error' key, it indicates that the conversation ID is not available, and an appropriate message is printed, returning None. Otherwise, the conversation ID is extracted from the response, stored in the conversationid variable, and returned.

* 1. **Sending a Message**

    def send\_message(self, text):

        print("Entered send message")

        print(self.conversationid)

        url = '/'.join([self.base\_url, 'conversations', self.conversationid, 'activities'])

        json\_payload = {

            'locale': 'en-EN',

            'type': 'message',

            'from': {'id': 'user1'},

            'text': text

        }

        print(url,json\_payload)

        bot\_response = requests.post(url, headers=self.headers, json=json\_payload)

        print(bot\_response)

        if bot\_response.status\_code == 200:

            print("success send message")

            return "message sent"

        return "error contacting bot"

**Explanation:**

The send\_message method is responsible for sending a message to the bot. It constructs the URL by joining the base\_url, 'conversations', conversationid, and 'activities'. It creates a JSON payload containing the message details, such as the locale, type, sender's ID, and the text of the message.

The requests.post method is used to send a POST request to the URL with the specified headers and JSON payload. If the response status code is 200 (indicating a successful request), the method returns "Message sent". Otherwise, it returns "Error contacting bot".

* 1. **Retrieving Messages**

    def get\_messages(self):

        url = '/'.join([self.base\_url, 'conversations', self.conversationid, 'activities'])

        bot\_response = requests.get(url, headers=self.headers)

        if bot\_response.status\_code == 200:

            json\_response = bot\_response.json()

            self.response = json\_response['activities'][-1]['text']

            data = json.loads(self.response)

            data['inputHint'] = json\_response['activities'][-1]['inputHint']

            return data

        else:

            print("Error contacting bot for response")

**Explanation:**

The get\_messages method retrieves the messages from the bot. It constructs the URL by joining the base\_url, 'conversations', conversationid, and 'activities'. It sends a GET request to the URL with the specified headers.

If the response status code is 200, indicating a successful request, the JSON response is parsed. The last received activity's text is extracted and stored in the response variable. The text is then loaded into a Python dictionary using json.loads, and the 'inputHint' field is added to the dictionary. The resulting dictionary is returned.

If the response status code is not 200, an appropriate error message is printed.

**Conclusion**

This tutorial provided an explanation of the functionalities and objectives of each function in the DirectLineAPI script. Each line of code was explained to give you a clear understanding of what it does. By following this tutorial, you should now have a solid understanding of how to use the DirectLineAPI class to interact with the Direct Line API of the Bot Framework in Python.