Jesse Williams | Date: February 18th, 2026 | Version *1.1*

Artificial Intelligence (NORA)

Abstract

This project is a Python-based Large Language Model (LLM) application called NORA (Neural Organized Responsive Assistant), which listens and responds to questions and commands given by the user. It uses Ollama's Application Programming Interface (API) System for the base framework of the Model, and other data analysis to formulate unique responses. It uses Picovoice to listen and for a wake word, and uses a Whispermodel to convert Speech to Text.

Table of Contents

Introduction

(September 19th, 2025)

Since AI started becoming more relevant, I have had an interest in leading the science and technology world into an era where AI is used to help advance humanity further than ever before. I decided I wanted to either major or minor in artificial intelligence, and a friend of mine suggested that I should get to work learning how AI models work and operate now. Furthermore, I decided that I wanted to try to code my own open-source artificial intelligence to use for future projects of mine and to benefit others.

Overview

(Septembet 19th, 2025)

This project aims to learn, explore, and create a large language model (LLM) that can be used for a wide variety of applications, such as my own or other people's projects.

(October 13th, 2025)

The goal of this project is to create an AI assistant capable of assisting with a wide variety of tasks, including holding conversations and speaking out loud in response, opening and closing applications, searching the web, and scheduling tasks upon request.

Ollama LLM API

(Feburary 18th, 2026)

The Ollama LLM API is an interface that lets developers programmatically interact with a variety of models running through Ollama. By locally hosting the AI Model on my computer, I can send prompts to the running model and receive generated responses. Additionally, it allows me to control a variety of model settings, like pre-requisite information.

Research

Entry 1.0

(September 19th, 2025)

LLMs are coded in Python due to its extensive library. After I set it up, I also have to train my model, which I will do more research on later.

Entry 1.1

(September 22nd, 2025)

To make LLMs, you obviously need lots of resources for training, which requires lots of money and storage. A workaround for this is to use other open-source LLM training datasets to work around not having my own.

There are two really good open source libraries for Python, specifically for machine learning: Tensorflow—developed by Google—and PyTorch. TensorFlow provides a good platform for developing, training, and executing all types of AIs, making it a good choice for different kinds of AIs. PyTorch, however, is more widespread and commonly used for large language models, so it will most likely be the main library I use.

The ability to search online is a key factor for my AI, so I need to implement a few things to make sure it can look through the internet and also give responses, the first of which is a web crawler.

Entry 1.2

(October 13th, 2025)

I wanted to start fresh with my understanding of everything I was trying to do thus far. First, I need to become familiar with all of the dependencies I would need to accomplish everything relating to my goal.

My temporary name for the AI will be Terry, so I will refer to it as Terry from now on. I want Terry to activate or start listening whenever I say a phrase—like Siri or Jarvis. To do this, I need to install the pvporcupine library, which gives access to Porcupine—a very accurate and lightweight word engine—which I will use to detect a phrase I can say that will enable Terry to start listening.

Next, I need it to listen to my voice and transcribe what I am saying into text. I can do this by using the Faster-Whisper Python library. Faster-whisper is made for faster and memory-efficient speech-to-text transcription, making it essential for picking up out-loud speech.

For the actual LLM, I need the requests library, a library I've used in the past, so I can utilize API's, storage features, or have Terry do an image search. This is necessary because of that last part, but moreover, because I am doing this without any money.

For the large data management required for this project, I need to use another library I am familiar with, which I am actually using in my Synthetic Data Generator, Numpy. Numpy is an open-source library that is fundamental for large-scale scientific computing, but more importantly, it can handle large amounts of storage because of its arrays.

Now that I have a way to converse with Terry, I need a way to get Terry to speak back out loud. I can do this by adding a sound management library, sounddevice, which is particularly good when paired with Numpy, anyway.

That is the way to handle and manage audio, but the actual audio will come from the pyttsx3 library. This is the best option because it can work offline, which would be essential for using Terry while not connected to the internet.

Now that I have researched the core dependencies for the LLM to work, I need to research dependencies related to the extra functionalities I want to use.

BeautifulSoup4 is a Python library I have also used in the past. It is the best library for locally hosted Ollama API for the Mistral LLMng things off the internet. This will be essential if I ask Terry to search for something for me on the internet. I also need to pair it with lxml to parse through the data I scrape from websites.

If I want Terry to open applications or do tasks for me on my computer, I need to install the pyautogui library. It's a cross-platform library that allows Terry to do keyboard and mouse inputs, mimicking those of a human.

For scheduling, I need the schedule library, which is literally made for scheduling and designing periodic jobs, like scheduling functions to run at different times. For example, I could ask Terry to open my Synthetic Data Generator at 3:45 pm.

To make Terry work on Windows, Mac, and Linux, I need the platformdirs library. It is required to determine different things that would vary per platform, such as platform-specific directories.

For extra cross-platform compatibility, I'm installing the PyAudio library, which is meant for cross-play audio—in this case, it's for when Terry speaks.

Entry 1.3

(October 14th, 2025)

My friends and I were talking about my current AI project after calculus was over, and I mentioned wanting a better name for my AI. Rocco Reale said that I should call it Nora, which Dylan Ricky—my other friend—also agreed on. We spent the next 15 minutes brainstorming what Nora should stand for, and came up with: Neural, Organized, Response, Assistant.

Entry 1.4

(October 17th, 2025)

I was talking about Nora in my robotics class to some peers, and was spitballing ideas to make her better. I thought about how Tony Stark could talk to Jarvis anywhere on the go, and wondered if it was possible to do the same with Nora. At first, I didn't think there was a cost-effective way to do so, as a microbit—a mini computer—couldn't locally host the locally hosted Ollama API for the Mistral LLM, or store all of the info needed to run Nora. Then, I remembered Raspberry Pi's are also small computers, but have way more applications that include larger storage. Sure enough, a Raspberry Pi could run an ollama locally hosted Ollama API for the Mistral LLM, along with the rest of my code for Nora.

Entry 1.5

(October 18th, 2025)

To utilize different data processing and handling techniques regarding audio, I needed the librosa library. The librosa library has many tools for understanding audio signals, especially for things like automatic speech recognition, which is what I'm trying to do.

I was thinking of ways to actually get a recording sound file whenever I stumbled across the actual soundfile library. It is a powerful and user-friendly tool that is used to read and write sound files, which I would need to write my own sound files.

I started thinking about extra additions I could add to Nora whenever I upload her to a Raspberry Pi, and I came up with making her a hologram. I can 3D print the main body, with specific places to put all of the components—Raspberry Pi, mini-microphone, mini-speaker, and mini-display. I could have a 3D mesh of a head with different animations playing on the display. To make the actual hologram, I can use what's called the Pepper's Ghost effect, where you use an angled reflective surface to make the reflected image look like it's hovering. I can seal this angled reflective surface in a box to sell the hologram effect. This transparent box would go on top of the display to make whatever is on the display look like a hologram, especially in the dark

Entry 1.6

(October 19th, 2025)

While researching training data, I stumbled upon an open-source J.A.R.V.I.S AI model that had plenty of training data I could incorporate into my AI, and with some refinement and new additions, could be used to train Nora on how to act in different scenarios—link to J.A.R.V.I.S model in the References section.

Entry 1.7

(October 24th, 2025)

I told my teacher, Mr. Miller, about Nora 2.0 and asked him for any spare parts he might have. He gave me a school-issued speaker that he said I could use for my project free of charge. I will disassemble it and remove the actual speakers inside it later.

Entry 1.8

(November 4th, 2025)

After logging in to Picovoice, I discovered that they have more applications than just wake word detection, one of the most notable being speaker recognition, which I could use with Nora.

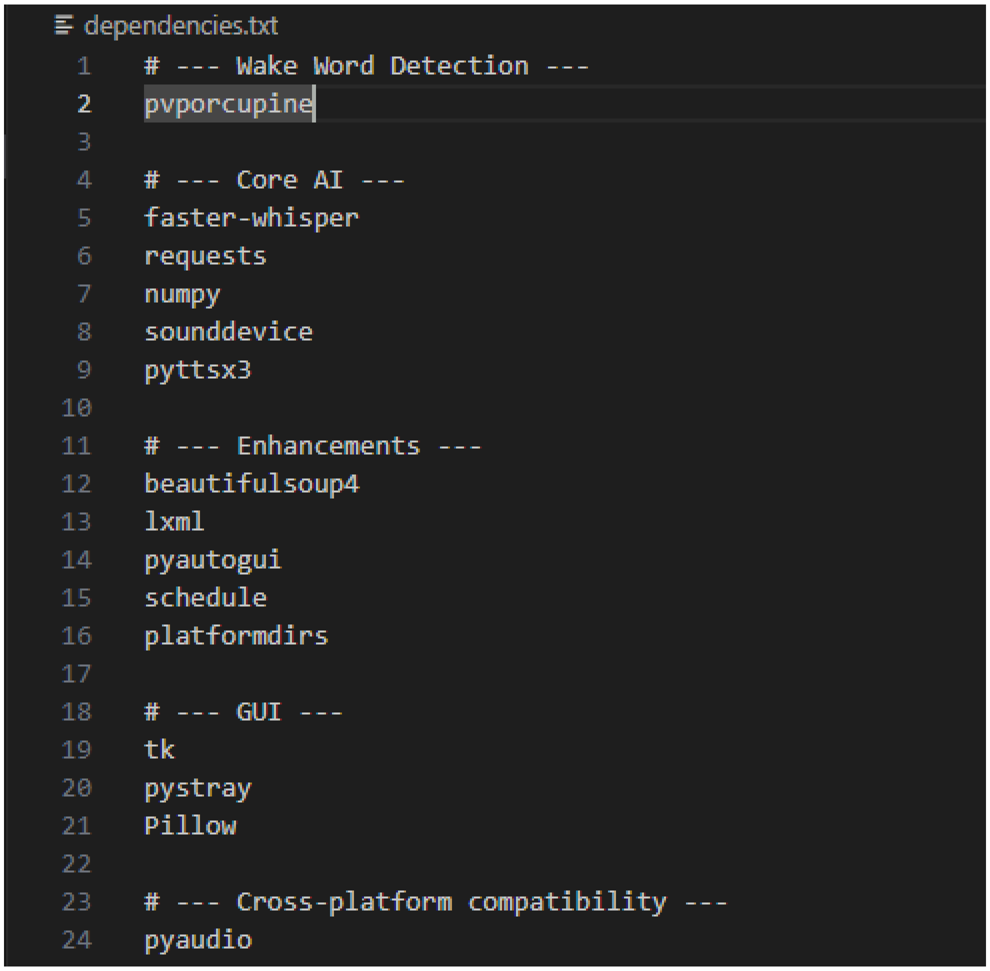
I used Picovoice to create and train an api to detect the wake word/phrase "Hey Nora" and downloaded the contents of the provided zip file. Since Picovoice only allows you to train one once a month, I have to wait til December to create one for Nora 2.0 on the Raspberry Pi.

Coding

Entry 1.0

(October 13th, 2025)

I wanted to restart the way I looked at things with a fresh outlook by beginning with all of the dependencies I would need. I made a text file and titled it dependencies, so I can run "pip install -r dependencies.txt" in the terminal whenever I need to install the dependencies (Fig. 1.1).

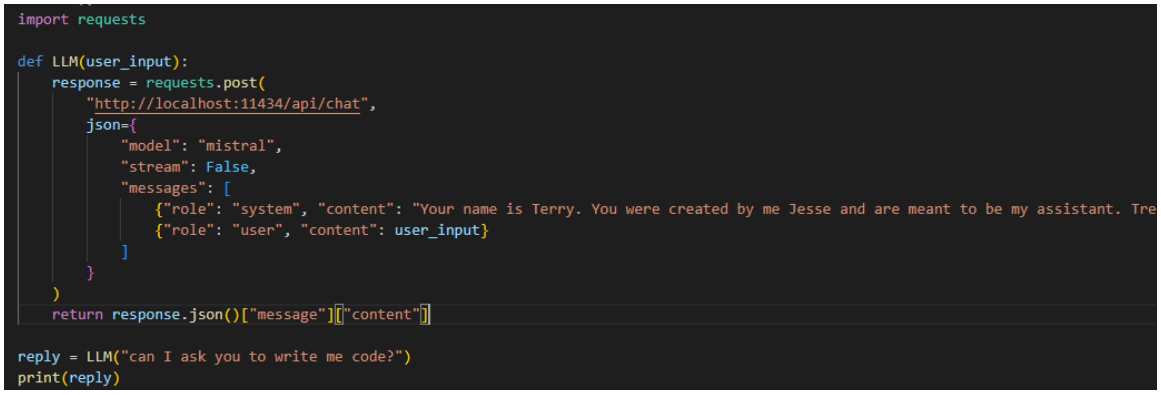


dependencies.txt

```
# --- Wake Word Detection ---
pvporcupine

# --- Core AI ---
faster-whisper
requests
numpy
sounddevice
pyttsx3

# --- Enhancements ---
beautifulsoup4
lxml
pyautogui
schedule
platformdirs

# --- GUI ---
tk
pystray
Pillow

# --- Cross-platform compatibility ---
pyaudio
```

Fig. 1.1 dependencies

Now I want to set up the actual LLM. For now, I'm going to use a locally hosted Ollama API for the Mistral LLM for my LLM, but in the future, I may make my own instead. I did this by downloading Ollama, which allows for downloading and running LLMs locally. Then I used my terminal to install minstrel AI via Ollama, and now I have a locally hosted Ollama API for the Mistral LLM I can use. I import requests and create a JSON object with related information for how the AI should behave, and the input being processed (Fig. 1.2). Then, it returns the message and the related content.

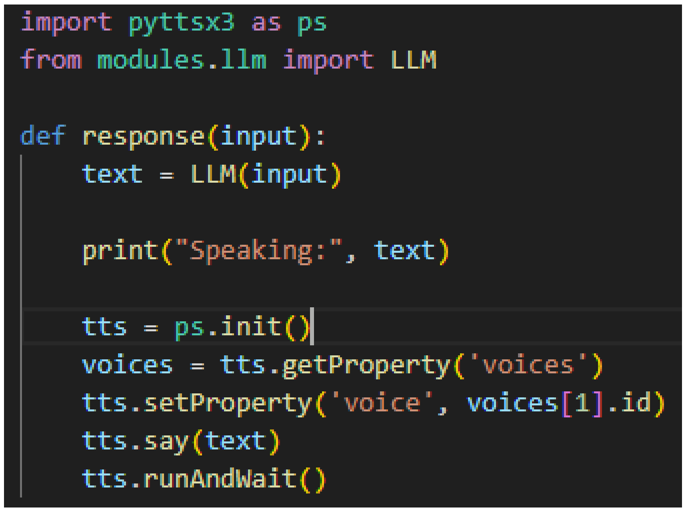


```python
import requests

def LLM(user_input):
    response = requests.post(
        "http://localhost:11434/api/chat",
        json={
            "model": "mistral",
            "stream": False,
            "messages": [
                {"role": "system", "content": "Your name is Terry. You were created by me Jesse and are meant to be my assistant. Tre
                {"role": "user", "content": user_input}
            ]
        }
    )
    return response.json()["message"]["content"]

reply = LLM("can I ask you to write me code?")
print(reply)
```

Fig. 1.2 locally hosted Ollama API for the Mistral LLM setup

I added the context string to a config file, which I will be using to adjust different things about the AI without going through every file.

Next, I wanted to do the text-to-speech system. I started by importing the pyttsx3 library and made a method that would take in an input and then call the LLM method in another file by using that input. Then I get a voice from one of the preset available ones and play the text-to-speech until it's finished (Fig. 1.3).



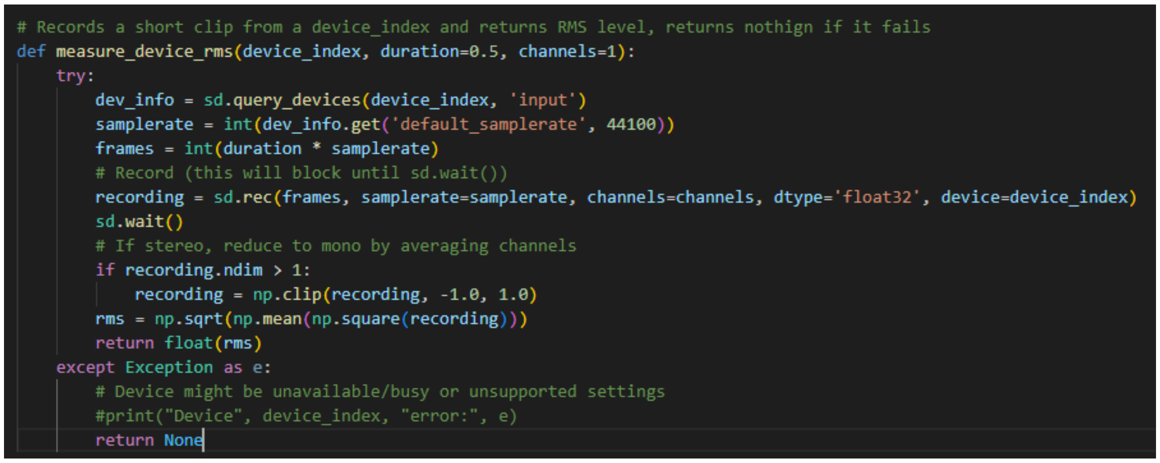
```python
import pyttsx3 as ps
from modules.llm import LLM

def response(input):
    text = LLM(input)

    print("Speaking:", text)

    tts = ps.init()
    voices = tts.getProperty('voices')
    tts.setProperty('voice', voices[1].id)
    tts.say(text)
    tts.runAndWait()
```

Fig. 1.3 text-to-speech setup

Entry 1.1

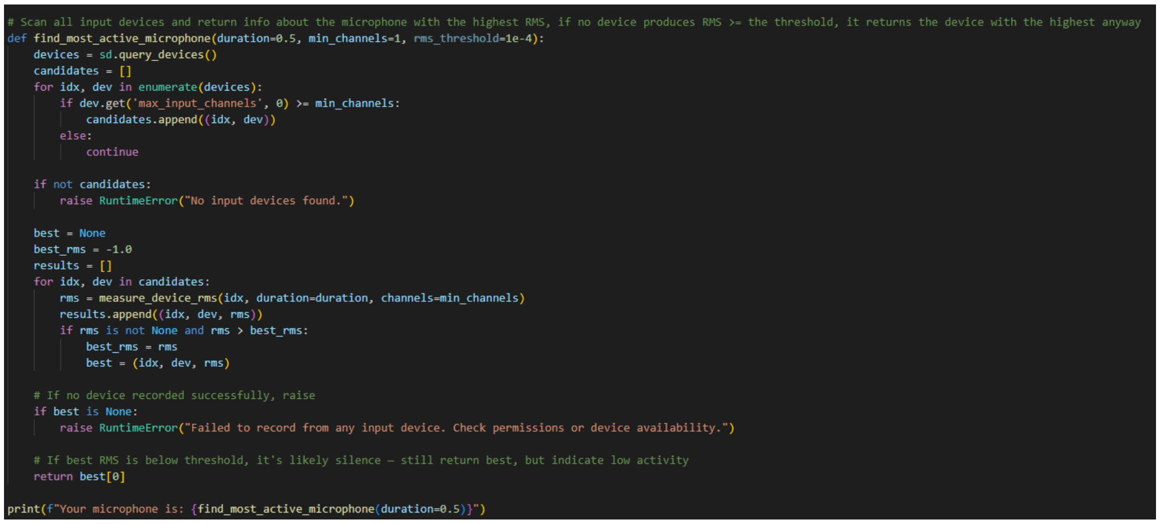
(October 13th, 2025)

Now that the locally hosted Ollama API for the Mistral LLM was fully functional, along with a TTS response, I needed to start the voice-to-speech process. I began by making a module to find the user's most active microphone and return it, so you wouldn't need to input one. The first method takes a given input audio device. It records a short clip to determine its RMS level—a mathematical measurement for the average power/loudness from a signal over time. This always returned accurate RMS values, so now I needed a method to determine the best microphone based on those results (Fig. 1.4). It starts by making a list of candidates—all of the input audio devices—and makes an array of them to use later. Then it uses that array to assign an RMS value to every input device by using the method I made previously, and storing that in a new array. While it's appending the RMS values to the microphone inputs, it actively compares them by checking if the RMS value is greater than the best RMS value, and if it is, it replaces it as the best value. Finally, it returns the best microphone and all information, and all of the corresponding information (Fig. 1.5).



```python
# Records a short clip from a device_index and returns RMS level, returns nothign if it fails
def measure_device_rms(device_index, duration=0.5, channels=1):
    try:
        dev_info = sd.query_devices(device_index, 'input')
        samplerate = int(dev_info.get('default_samplerate', 44100))
        frames = int(duration * samplerate)
        # Record (this will block until sd.wait())
        recording = sd.rec(frames, samplerate=samplerate, channels=channels, dtype='float32', device=device_index)
        sd.wait()
        # If stereo, reduce to mono by averaging channels
        if recording.ndim > 1:
            recording = np.clip(recording, -1.0, 1.0)
        rms = np.sqrt(np.mean(np.square(recording)))
        return float(rms)
    except Exception as e:
        # Device might be unavailable/busy or unsupported settings
        #print("Device", device_index, "error:", e)
        return None
```

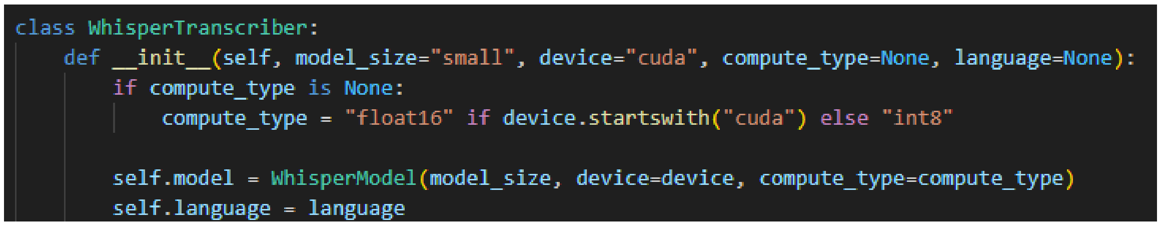
Fig. 1.4 measure_device_rms method



```python
# Scan all input devices and return info about the microphone with the highest RMS, if no device produces RMS >= the threshold, it returns the device with the highest anyway
def find_most_active_microphone(duration=0.5, min_channels=1, rms_threshold=1e-4):
    devices = sd.query_devices()
    candidates = []
    for idx, dev in enumerate(devices):
        if dev.get('max_input_channels', 0) >= min_channels:
            candidates.append((idx, dev))
        else:
            continue

    if not candidates:
        raise RuntimeError("No input devices found.")

    best = None
    best_rms = -1.0
    results = []
    for idx, dev in candidates:
        rms = measure_device_rms(idx, duration=duration, channels=min_channels)
        results.append((idx, dev, rms))
        if rms is not None and rms > best_rms:
            best_rms = rms
            best = (idx, dev, rms)

    # If no device recorded successfully, raise
    if best is None:
        raise RuntimeError("Failed to record from any input device. Check permissions or device availability.")

    # If best RMS is below threshold, it's likely silence — still return best, but indicate low activity
    return best[0]

print(f"Your microphone is: {find_most_active_microphone(duration=0.5)}")
```

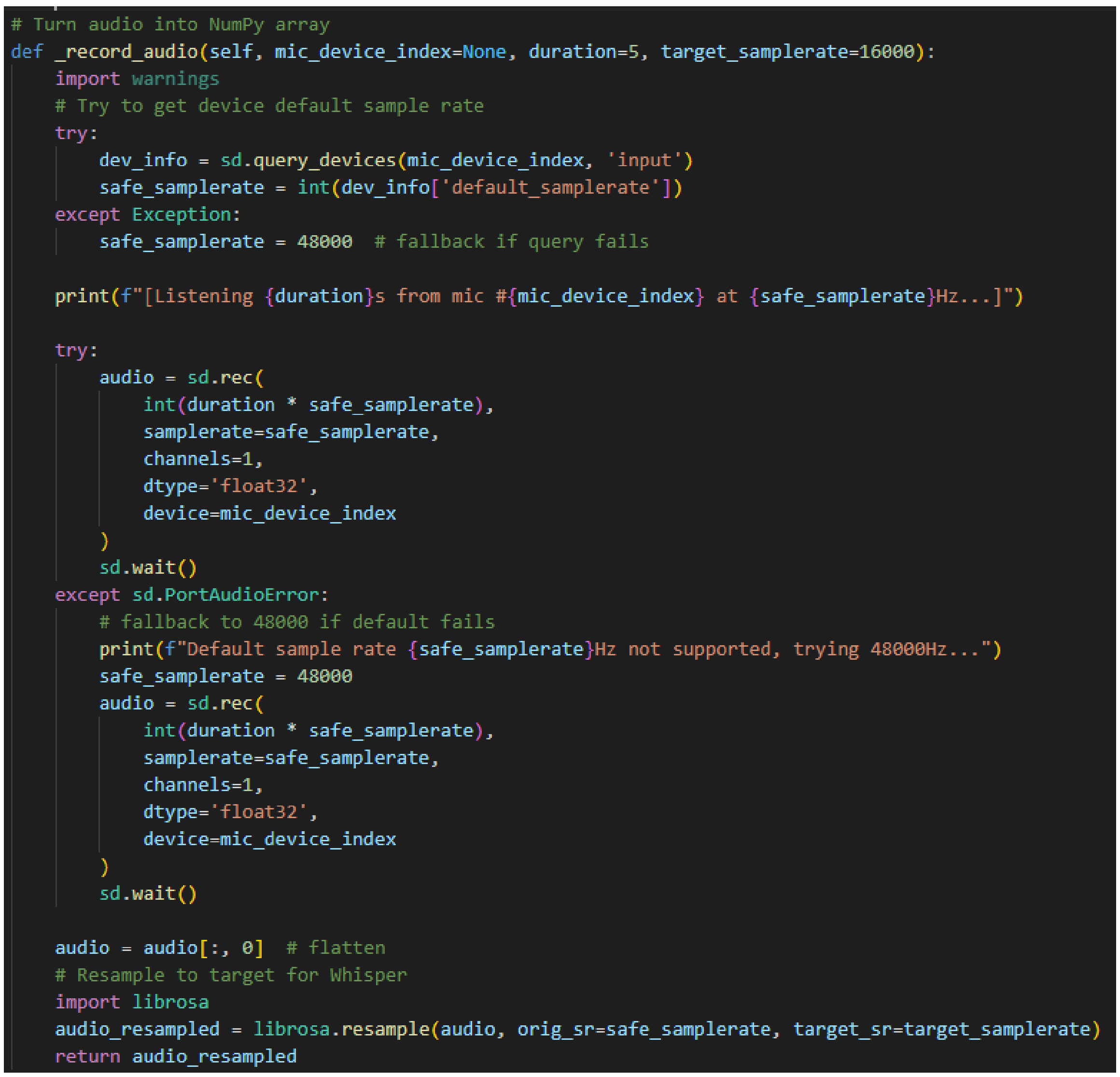
Fig. 1.5 find_most_active_microphone method, the rest of the array is only printed for debugging purposes

Next, I needed to create the actual transcribing module. I started by creating a full whisper transcriber class, as I would need to define certain parameters for the whisper model I was going to be using anyway. I made the initialization method to be compatible with different parameters in case I want to tweak the version of the whisper model I use(Fig. 1.6). After setting up the model, I created the actual audio recording method, which for now will have a duration for testing purposes, but the end goal is to have it listen until the user finishes a sentence. First, it finds the audio device's samplerate, which, if it fails, will use a general safe sample rate of 48000. Then it tries to record audio and converts it into an array; however, if it fails with the current samplerate, it will fall back on the safe sample rate instead and retry. Finally, it flattens the audio into a mono one, changes the sample rate to the target one using the librosa library, and returns the final sampled audio array (Fig. 7). Now, I needed to set up the transcribe_array module. First, I created an in-memory binary WAV object for faster transcription. Then I used the soundfile library to write a sound file to that WAV using a given audio array. Finally, I use whisper to convert the sound file into text and then process the output to create a single string text that I return (Fig. 1.8). The last part is to assemble it and return the outputted string (Fig. 1.9)



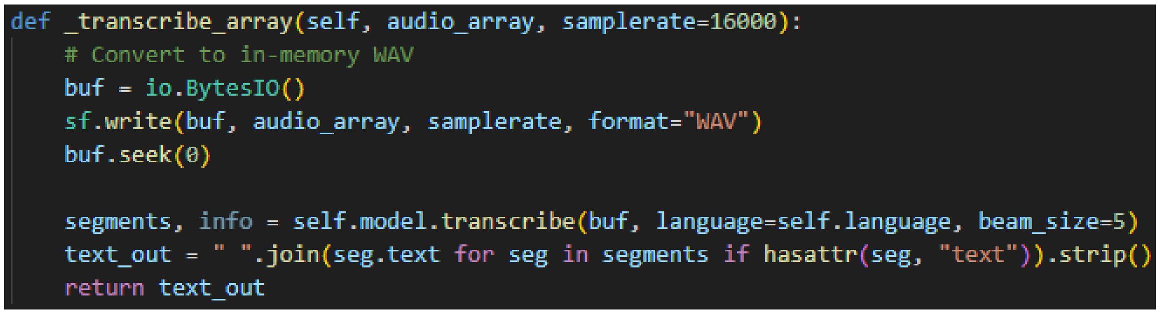
```
class WhisperTranscriber:
    def __init__(self, model_size="small", device="cuda", compute_type=None, language=None):
        if compute_type is None:
            compute_type = "float16" if device.startswith("cuda") else "int8"

        self.model = WhisperModel(model_size, device=device, compute_type=compute_type)
        self.language = language
```

Fig. 1.6 parameters for the whisper model



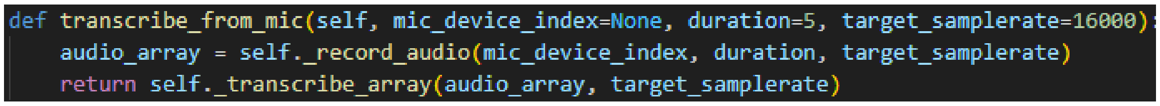
```python
# Turn audio into NumPy array
def _record_audio(self, mic_device_index=None, duration=5, target_samplerate=16000):
    import warnings
    # Try to get device default sample rate
    try:
        dev_info = sd.query_devices(mic_device_index, 'input')
        safe_samplerate = int(dev_info['default_samplerate'])
    except Exception:
        safe_samplerate = 48000  # fallback if query fails

    print(f"[Listening {duration}s from mic #{mic_device_index} at {safe_samplerate}Hz...]")

    try:
        audio = sd.rec(
            int(duration * safe_samplerate),
            samplerate=safe_samplerate,
            channels=1,
            dtype='float32',
            device=mic_device_index
        )
        sd.wait()
    except sd.PortAudioError:
        # fallback to 48000 if default fails
        print(f"Default sample rate {safe_samplerate}Hz not supported, trying 48000Hz...")
        safe_samplerate = 48000
        audio = sd.rec(
            int(duration * safe_samplerate),
            samplerate=safe_samplerate,
            channels=1,
            dtype='float32',
            device=mic_device_index
        )
        sd.wait()

    audio = audio[:, 0]  # flatten
    # Resample to target for Whisper
    import librosa
    audio_resampled = librosa.resample(audio, orig_sr=safe_samplerate, target_sr=target_samplerate)
    return audio_resampled
```

Fig. 7 audio to array method



```python
def _transcribe_array(self, audio_array, samplerate=16000):
    # Convert to in-memory WAV
    buf = io.BytesIO()
    sf.write(buf, audio_array, samplerate, format="WAV")
    buf.seek(0)

    segments, info = self.model.transcribe(buf, language=self.language, beam_size=5)
    text_out = " ".join(seg.text for seg in segments if hasattr(seg, "text")).strip()
    return text_out
```

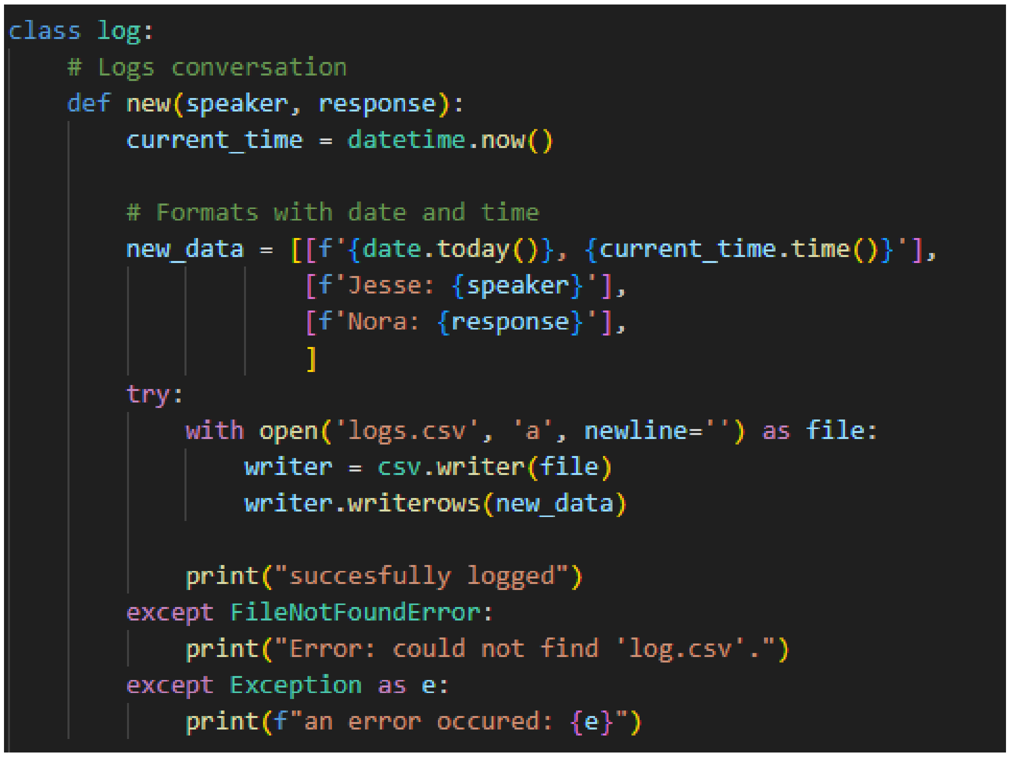
Fig. 8 transcribes the given array into text



```python
def transcribe_from_mic(self, mic_device_index=None, duration=5, target_samplerate=16000):
    audio_array = self._record_audio(mic_device_index, duration, target_samplerate)
    return self._transcribe_array(audio_array, target_samplerate)
```

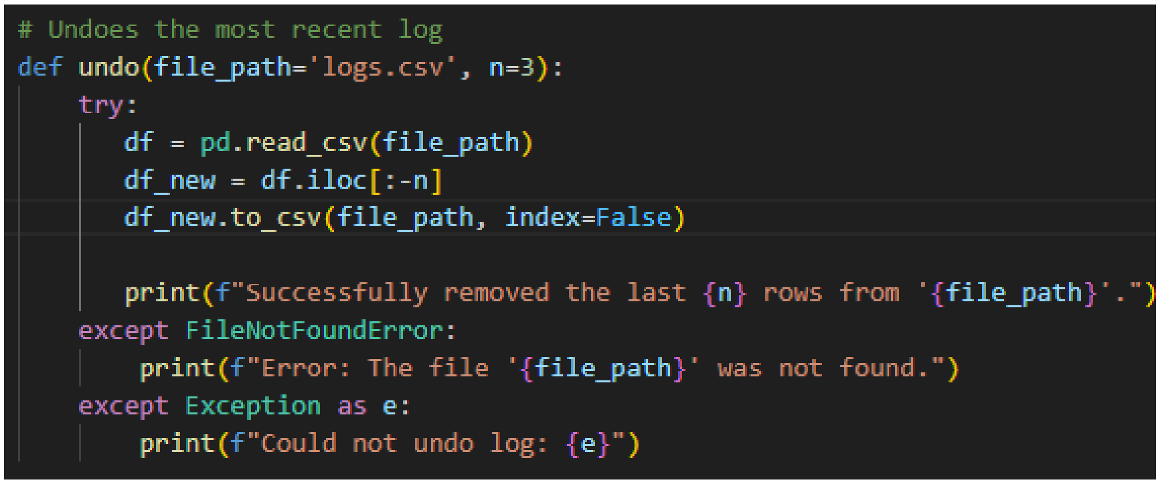
Fig. 1.9 compiles all the methods into one cohesive method that returns the given text entry

While I was away from my main computer and only had access to my school-issued MacBook, I started writing the code for the log function. I started by making a log class for all of the methods to be under. The first method logs the given sentence from the user and the response from the AI into a CSV file. It also logs the dates and times for everything (Fig. 1.10). This was already everything, but I wanted ways to remove logs, so I made an undo method, which undoes the most recent log (Fig. 1.11). Finally, I added a clear log method, which clears the entire log and starts a new one (Fig. 1.12).



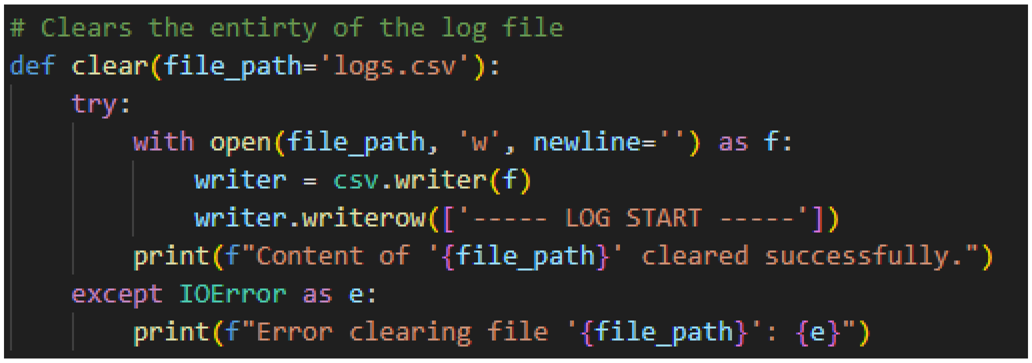
```python
class log:
    # Logs conversation
    def new(speaker, response):
        current_time = datetime.now()

        # Formats with date and time
        new_data = [[f'{date.today()}, {current_time.time()}'],
                    [f'Jesse: {speaker}'],
                    [f'Nora: {response}'],
                    ]
        try:
            with open('logs.csv', 'a', newline='') as file:
                writer = csv.writer(file)
                writer.writerows(new_data)

            print("succesfully logged")
        except FileNotFoundError:
            print("Error: could not find 'log.csv'.")
        except Exception as e:
            print(f"an error occured: {e}")
```

Fig. 1.10 logs the most recent conversation



```python
# Undoes the most recent log
def undo(file_path='logs.csv', n=3):
    try:
        df = pd.read_csv(file_path)
        df_new = df.iloc[:-n]
        df_new.to_csv(file_path, index=False)

        print(f"Successfully removed the last {n} rows from '{file_path}'.")
    except FileNotFoundError:
         print(f"Error: The file '{file_path}' was not found.")
    except Exception as e:
         print(f"Could not undo log: {e}")
```

Fig. 1.11 undoes the previous log



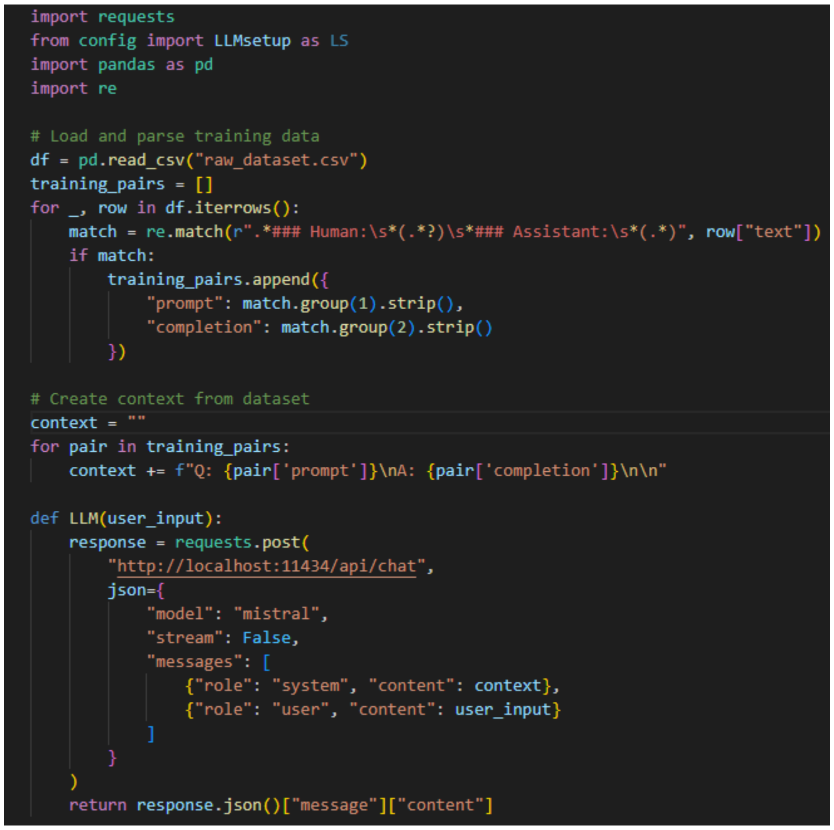
```python
# Clears the entirty of the log file
def clear(file_path='logs.csv'):
    try:
        with open(file_path, 'w', newline='') as f:
            writer = csv.writer(f)
            writer.writerow(['----- LOG START -----'])
        print(f"Content of '{file_path}' cleared successfully.")
    except IOError as e:
        print(f"Error clearing file '{file_path}': {e}")
```

Fig. 1.12 clears the entire log

Entry 1.2

(October 13th, 2025)

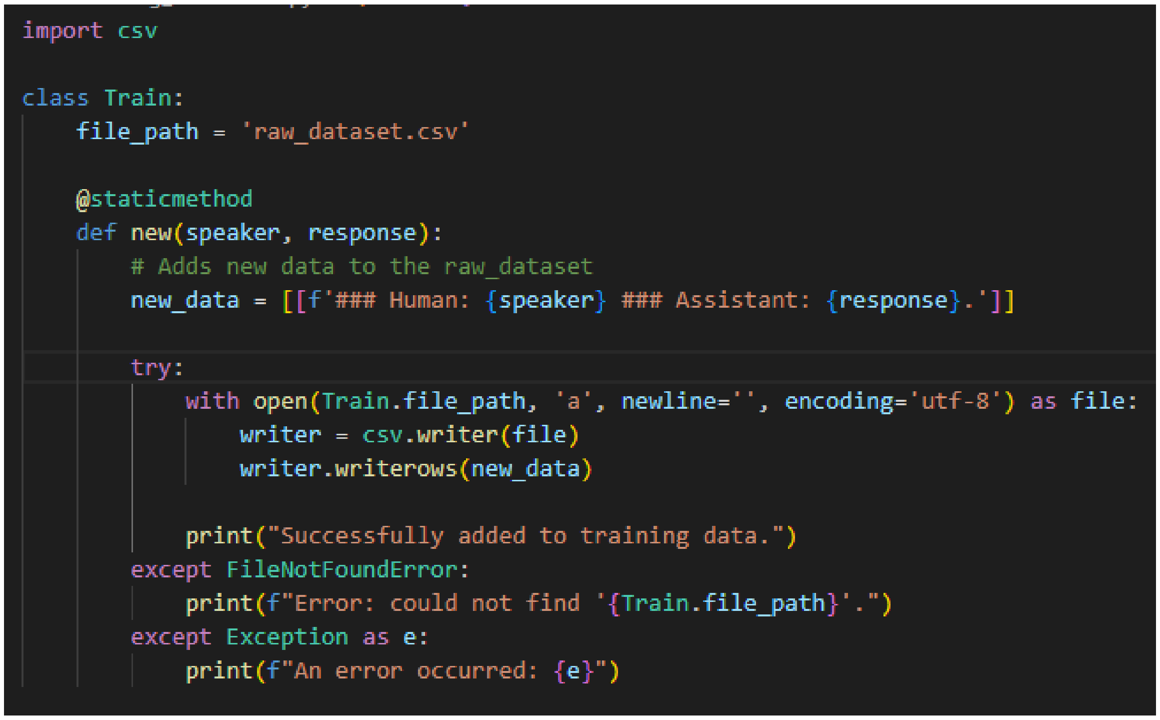
I wanted to start on another important Nora function: self-training. I want to be able to say train.new to Nora, to add successful conversations into Nora's training data. I started by getting a training data format from an open-source J.A.R.V.I.S model and configuring it to match Nora specifically. Then I updated the API module to read and separate the questions and responses from the CSV format, and into a new formatted string that is sent to the api with every request as extra context (Fig. 1.13).



```python
import requests
from config import LLMsetup as LS
import pandas as pd
import re

# Load and parse training data
df = pd.read_csv("raw_dataset.csv")
training_pairs = []
for _, row in df.iterrows():
    match = re.match(r".*### Human:\s*(.*?)\s*### Assistant:\s*(.*)", row["text"])
    if match:
        training_pairs.append({
            "prompt": match.group(1).strip(),
            "completion": match.group(2).strip()
        })

# Create context from dataset
context = ""
for pair in training_pairs:
    context += f"Q: {pair['prompt']}\nA: {pair['completion']}\n\n"

def LLM(user_input):
    response = requests.post(
        "http://localhost:11434/api/chat",
        json={
            "model": "mistral",
            "stream": False,
            "messages": [
                {"role": "system", "content": context},
                {"role": "user", "content": user_input}
            ]
        }
    )
    return response.json()["message"]["content"]
```

Fig. 1.13 the new part is above the LLM method

Next, I made a new module called Train in which I created another class, just in case I wanted to add future methods to it later, but the main method I am adding is new. The purpose will be to log successful conversations into the training data to have Nora train herself on successful interactions—I may make this automated later. It is a static method that works similarly to the log.new function I coded, except it formats the conversation into the same formatting as the training data, then adds it to the CSV file (Fig. 1.14).

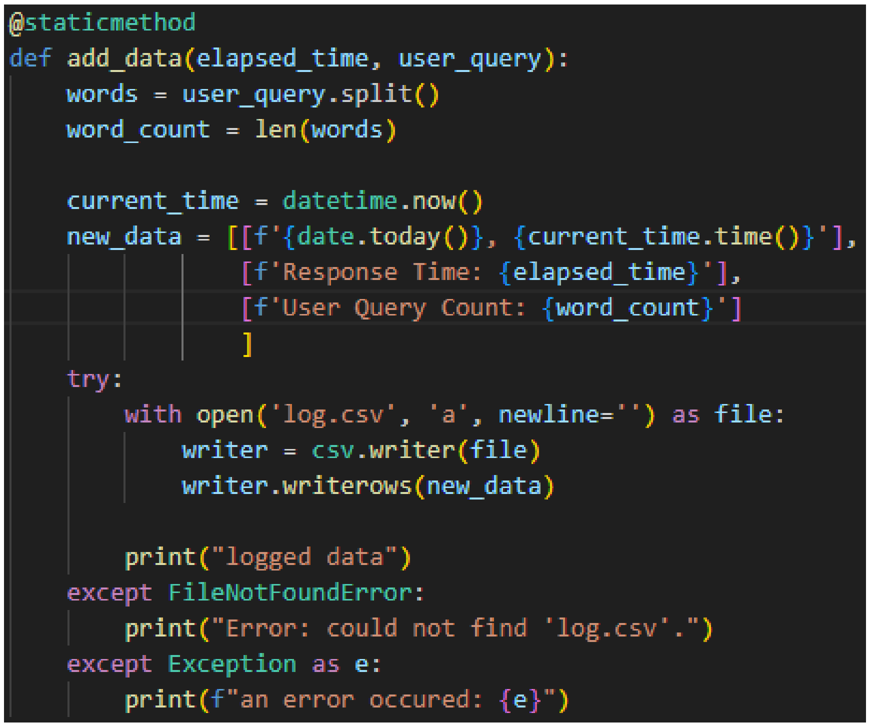


```python
import csv

class Train:
    file_path = 'raw_dataset.csv'

    @staticmethod
    def new(speaker, response):
        # Adds new data to the raw_dataset
        new_data = [[f'### Human: {speaker} ### Assistant: {response}.']]

        try:
            with open(Train.file_path, 'a', newline='', encoding='utf-8') as file:
                writer = csv.writer(file)
                writer.writerows(new_data)

            print("Successfully added to training data.")
        except FileNotFoundError:
            print(f"Error: could not find '{Train.file_path}'.")
        except Exception as e:
            print(f"An error occurred: {e}")
```

Fig. 1.14 Tain.new adds previous conversations into the training data CSV

I changed all the log methods to be static methods.

I added another method to the log module. After every interaction, it logs the query time and the user query count in the log CSV. This is useful data to help me determine the averages of both to see if I can improve those times in the future. It also logs the dates and times of every log (Fig. 1.15)



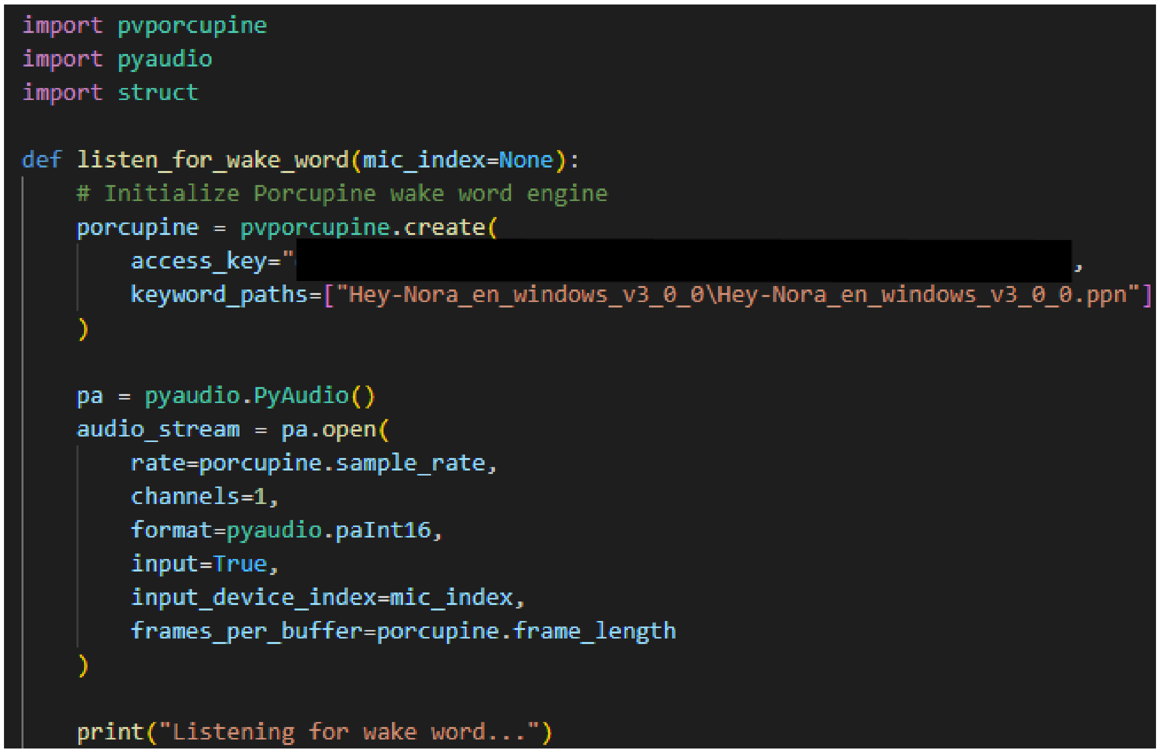
```
@staticmethod
def add_data(elapsed_time, user_query):
    words = user_query.split()
    word_count = len(words)

    current_time = datetime.now()
    new_data = [[f'{date.today()}, {current_time.time()}'],
                [f'Response Time: {elapsed_time}'],
                [f'User Query Count: {word_count}']
                ]
    try:
        with open('log.csv', 'a', newline='') as file:
            writer = csv.writer(file)
            writer.writerows(new_data)

        print("logged data")
    except FileNotFoundError:
        print("Error: could not find 'log.csv'.")
    except Exception as e:
        print(f"an error occured: {e}")
```

Fig. 1.15 new addition to the log module

Entry 1.3

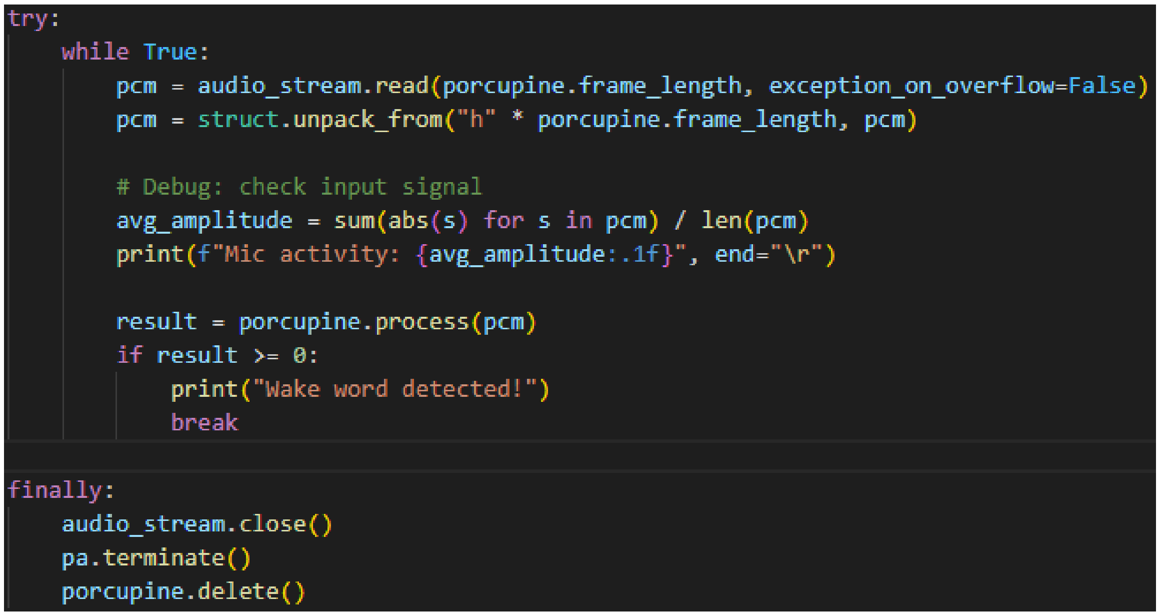
(November 4th, 2025)

After setting up the PPN file from Picovoice's wake word feature, I added a method and made one of the parameters the mic index, so it uses the correct one from earlier. Then I initialize the wake word engine using my Pvporcupin key and the file path to the PPN file. Then I used Pyaudio to listen to my mic(Fig. 1.16). Then I made a loop that constantly listens to my mic, converts the raw bytes into integers representing different audio amplitudes, creates an average of the amplitudes—a rough measure of the volume, constantly displays the volume for debugging purposes, and finally, it sends the audio frame to the wake word detection. Then it checks if it's the correct wake word. If it is, then the loop will end with a break, and it cleans up everything(Fig. 1.17).



```
import pvporcupine
import pyaudio
import struct

def listen_for_wake_word(mic_index=None):
    # Initialize Porcupine wake word engine
    porcupine = pvporcupine.create(
        access_key="[redacted],
        keyword_paths=["Hey-Nora_en_windows_v3_0_0\Hey-Nora_en_windows_v3_0_0.ppn"]
    )

    pa = pyaudio.PyAudio()
    audio_stream = pa.open(
        rate=porcupine.sample_rate,
        channels=1,
        format=pyaudio.paInt16,
        input=True,
        input_device_index=mic_index,
        frames_per_buffer=porcupine.frame_length
    )

    print("Listening for wake word...")
```

Fig. 1.16 access key is blacked out for obvious reasons



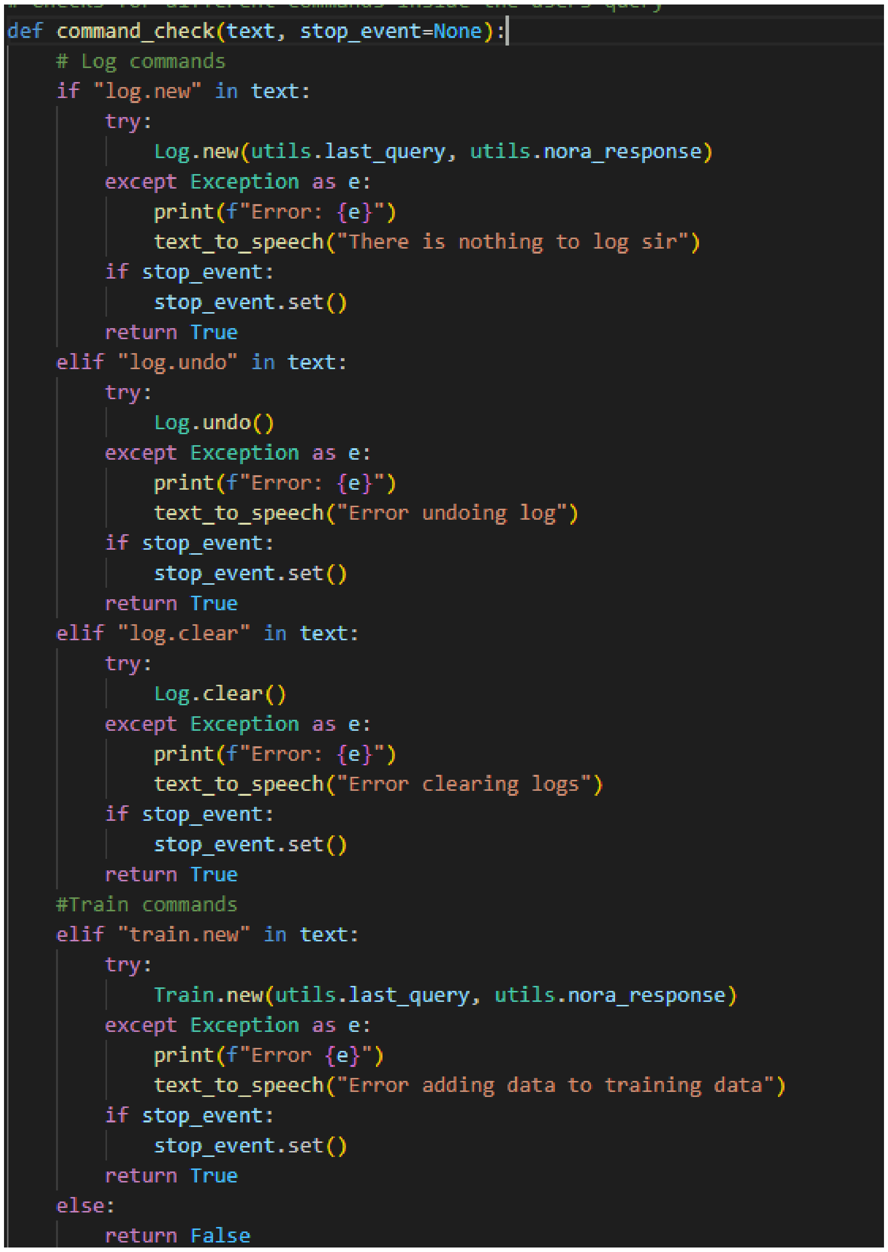
```
try:
    while True:
        pcm = audio_stream.read(porcupine.frame_length, exception_on_overflow=False)
        pcm = struct.unpack_from("h" * porcupine.frame_length, pcm)

        # Debug: check input signal
        avg_amplitude = sum(abs(s) for s in pcm) / len(pcm)
        print(f"Mic activity: {avg_amplitude:.1f}", end="\r")

        result = porcupine.process(pcm)
        if result >= 0:
            print("Wake word detected!")
            break

finally:
    audio_stream.close()
    pa.terminate()
    porcupine.delete()
```

Fig. 1.17 It runs this method before any other processing, so you have to say the wake word first

Entry 1.4

(November 16th, 2025)

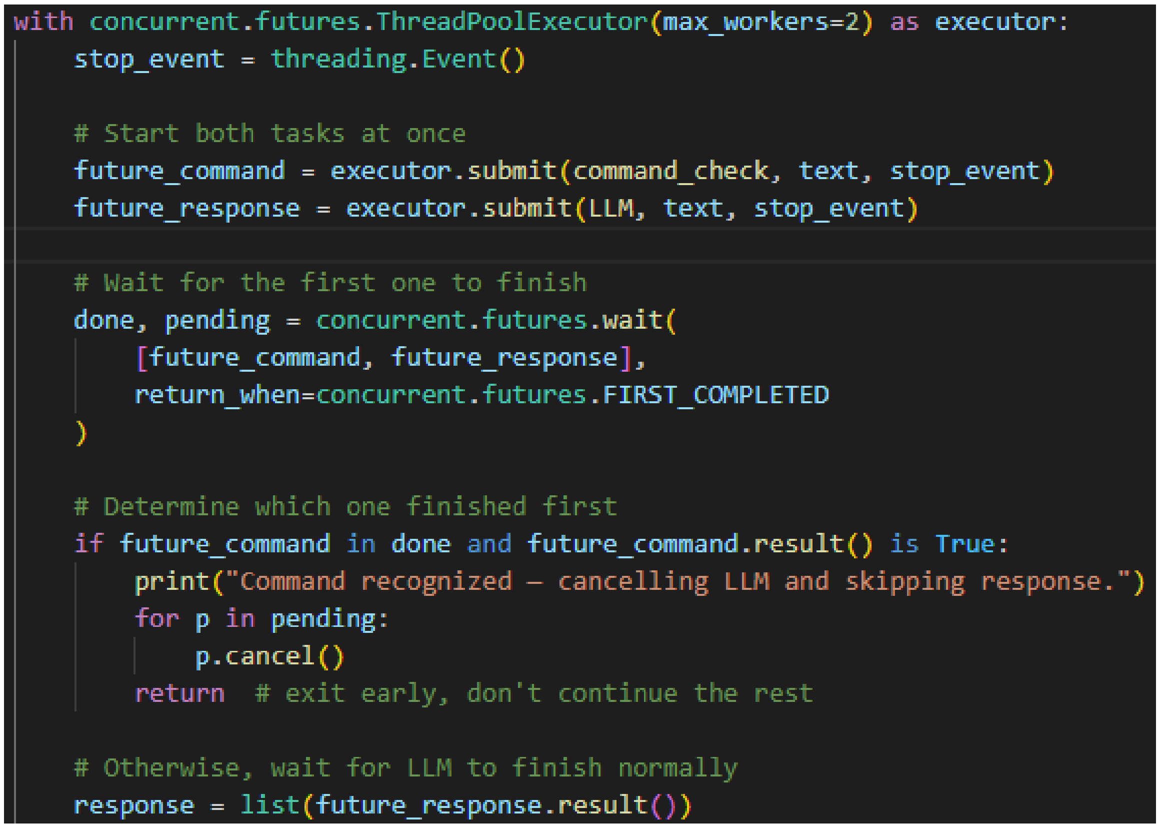
Now that the wake word was functioning, I needed to make it so she could differentiate between normal conversations and commands I want her to do. The concept was simple; all I had to do was retrieve the text string I specified and review it along with the list of commands to see if I had included one. I did this by making a new method in the execute_command module that takes the text as a parameter and looks for any commands, then executes them (Fig. 1.18).



```
def command_check(text, stop_event=None):
    # Log commands
    if "log.new" in text:
        try:
            Log.new(utils.last_query, utils.nora_response)
        except Exception as e:
            print(f"Error: {e}")
            text_to_speech("There is nothing to log sir")
        if stop_event:
            stop_event.set()
        return True
    elif "log.undo" in text:
        try:
            Log.undo()
        except Exception as e:
            print(f"Error: {e}")
            text_to_speech("Error undoing log")
        if stop_event:
            stop_event.set()
        return True
    elif "log.clear" in text:
        try:
            Log.clear()
        except Exception as e:
            print(f"Error: {e}")
            text_to_speech("Error clearing logs")
        if stop_event:
            stop_event.set()
        return True
    #Train commands
    elif "train.new" in text:
        try:
            Train.new(utils.last_query, utils.nora_response)
        except Exception as e:
            print(f"Error {e}")
            text_to_speech("Error adding data to training data")
        if stop_event:
            stop_event.set()
        return True
    else:
        return False
```

Fig. 1.18 I still need to make the search and calendar commands

This works as is, but I wanted to optimize it so it will still work efficiently, which meant I needed to check for the commands while it was generating a response with Nora. I did this by using a thread pool in the main that runs both methods at the same time, and waits and sees if the command one returns false or not. If it does, it will say the Nora-generated response; if it is true, it will run the command (Fig. 1.19).



```
with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
    stop_event = threading.Event()

    # Start both tasks at once
    future_command = executor.submit(command_check, text, stop_event)
    future_response = executor.submit(LLM, text, stop_event)

    # Wait for the first one to finish
    done, pending = concurrent.futures.wait(
        [future_command, future_response],
        return_when=concurrent.futures.FIRST_COMPLETED
    )

    # Determine which one finished first
    if future_command in done and future_command.result() is True:
        print("Command recognized — cancelling LLM and skipping response.")
        for p in pending:
            p.cancel()
        return  # exit early, don't continue the rest

    # Otherwise, wait for LLM to finish normally
    response = list(future_response.result())
```

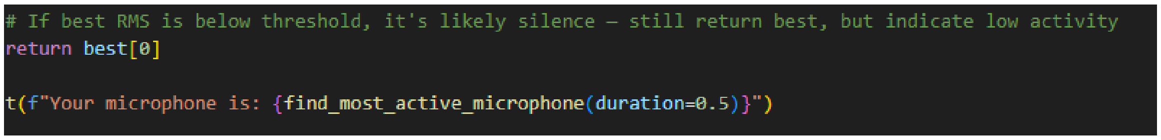
Fig. 1.19 This is probably the most useful thing I have learned in a while

Bugs

Entry 1.0

(October 18th, 2025)

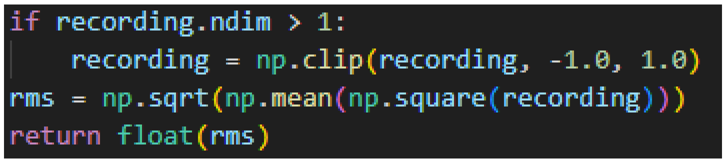
While coding the speech-to-text module, I ran into several issues. First, I kept getting an error message that my microphone object had no attributes. I realized that my find_most_active_microphone method was returning the beginning of a string instead of the correct value, and even after I removed the string, it still wasn't returning the correct value. I fixed this by just not returning a list value and only returning the index (Fig. 2.1).



```
# If best RMS is below threshold, it's likely silence — still return best, but indicate low activity
return best[0]

t(f"Your microphone is: {find_most_active_microphone(duration=0.5)}")
```

Fig. 2.1 returns the microphone with the best RMS value's index

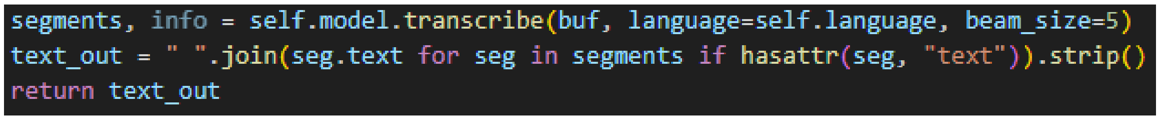
For a while, I kept getting a runtime error that would cause several subsequent errors. It still worked, but there were still errors nonetheless. I figured out that it was being caused because my mic data wasn't normalized. I fixed it by replacing the "recording.mean" statement, which was originally to calculate the average of the recording value to reduce stereo recording channels to mono, with a clip method from NumPy to prevent overflow by better limiting the values (Fig. 2.2). This worked because originally, the values being averaged were still too large, so cutting them down solved that problem.



```
if recording.ndim > 1:
    recording = np.clip(recording, -1.0, 1.0)
rms = np.sqrt(np.mean(np.square(recording)))
return float(rms)
```

Fig 2.2 better method of limiting values

I kept receiving traceback errors and determined the cause to be my speech-to-text module. The error was saying that there was a problem with my transcribe_array method, and on closer inspection, I realized that it wasn't my method, but more so the model of fast-whisper I was using, which only returns segment objects, which is important because right now my method, when adding the transcribed text to a string, was using .txt and .get, when you can't use .get on segment objects. So all I needed to do was remove the .get from the statement (Fig. 2.3).



```
segments, info = self.model.transcribe(buf, language=self.language, beam_size=5)
text_out = " ".join(seg.text for seg in segments if hasattr(seg, "text")).strip()
return text_out
```

Fig. 2.3 removed the .get statement for the standalone .text one

I finished coding the first wake word, but I kept getting a random error saying 'missing access key'. I looked it up, and it turns out that PVPorcupine already has specific wake words, and you can only use theirs unless you have your own ppn file. To do this, I need to create an account on the official PicVoice AI website at https://console.picovoice.ai/. Unfortunately, you can only create an account for personal use if you're 18 years or older, so I need to consider an alternative solution.

Entry 1.1

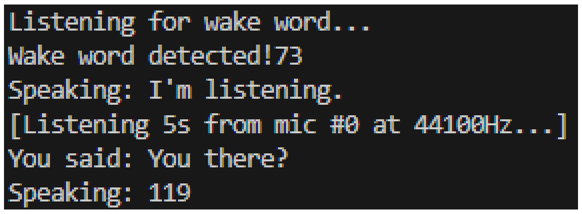
(October 28th, 2025)

To solve the last problem I was having, instead of making a convoluted solution, I asked my mom to create an account so I could still use it, which is perfectly within the website's rules.

Entry 1.2

(November 8th, 2025)

When trying to add a log feature that logs the response time, user query count, and the current date and time. I was doing this to track efficiency and optimisation. However, the method in which I'm doing it is causing problems. I have it keep track of the elapsed time in the LLM module, and return it along with the LLM response in a list. Then, in the main method, I have it set the response time and response to two different variables. The problem is that whenever Nora speaks, she just speaks the response time sometimes, and other times she just says her statement. The only thing I can think of that would be causing it is me ordering the list incorrectly, but that isn't the case because it wouldn't sometimes be right and sometimes be wrong, then (Fig. 2.4). This has been a problem for multiple days; I have just neglected to tackle it until now.



```
Listening for wake word...
Wake word detected!73
Speaking: I'm listening.
[Listening 5s from mic #0 at 44100Hz...]
You said: You there?
Speaking: 119
```

Fig. 2.4 The "speaking" is Nora

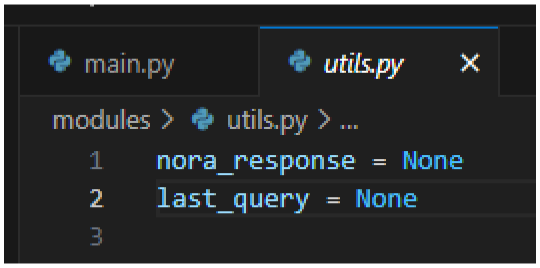
The issue was that I was using curly brackets in my return instead of something like parentheses. This was incorrect because curly brackets are used in Python to create unordered sets, not organized lists, so it would constantly flip-flop between the time being the first item and the response.

Entry 1.3

(November 16th, 2025)

I was having trouble getting the last response and user query from the main to the execute_command module. I tried making an accessor method, setting up the parameters in both methods, making the variables global, etc.

Eventually, I realized I could use the utils module to store the variable states in there, and have both main and execute_command access those specific variables from utils and change their state that way, so it always live updates (Fig. 2.5).



```
nora_response = None
last_query = None
```

Fig. 2.5 Variable states stored in utils.py

6.0 Future Work

1. Demonstrate Optimization (show memory CPU profiling results [especially on Raspberry Pi], maybe graph the performance impact of whisper models)
2. Include a system diagram (an Image showing the project flow)
3. Demo Video
4. Optimize response time
5. Finish the search function
6. 3D model a head and animate idle and talking
7. Add a GUI for Nora 1.0
8. Configure Nora 1.0 to make Nora 2.0, which works on Raspberry Pis
9. Buy other components to assemble Nora
10. 3D model a case to hold everything
11. Create a hologram utilizing the Pepper's Ghost effect

7.0 References

- Ollama (https://ollama.ai) – for locally hosted LLMs and Mistral model integration
- Faster Whisper (https://github.com/guillaumekln/faster-whisper) – speech-to-text engine
- Pyttsx3 (https://pypi.org/project/pyttsx3/) – text-to-speech library
- Librosa (https://librosa.org/) – audio processing library
- BeautifulSoup4 (https://www.crummy.com/software/BeautifulSoup/) – web scraping framework
- J.A.R.V.I.S (https://github.com/codewithbro95/J.A.R.V.I.S) - training data template