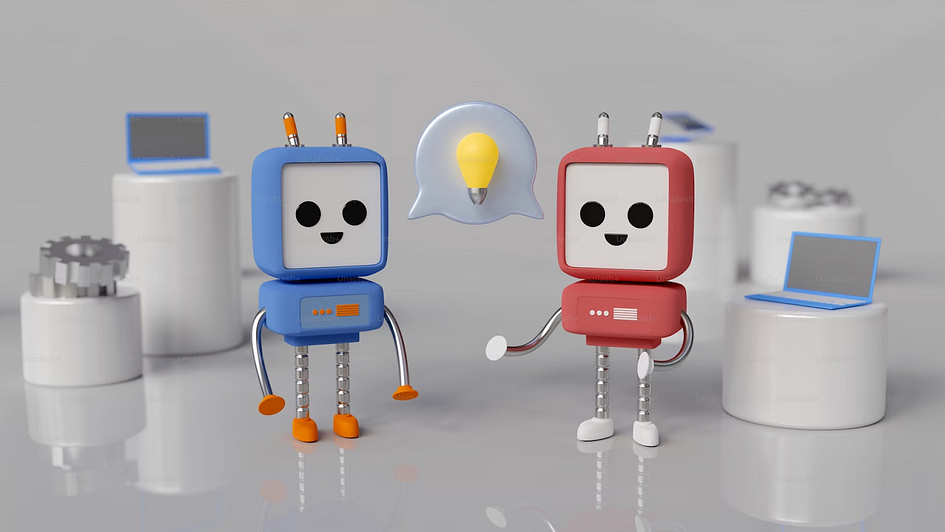
**Speech to Text to Speech with AI Using Python — a How-To Guide**

How to Create a Speech-to-Text-to-Speech Program

[[Naomi Kriger](https://naomikriger.medium.com/?source=post_page-----ee9b0b0ef082--------------------------------)](https://naomikriger.medium.com/?source=post_page-----ee9b0b0ef082--------------------------------) [Towards Data Science](https://towardsdatascience.com/?source=post_page-----ee9b0b0ef082--------------------------------) [Naomi Kriger](https://naomikriger.medium.com/?source=post_page-----ee9b0b0ef082--------------------------------)



[Image](https://unsplash.com/photos/aaujbh59zqI) by [Mariia Shalabaieva](https://unsplash.com/@maria_shalabaieva) from [unsplash](http://unsplash.com/)

It’s been exactly a decade since I started attending GeekCon (yes, a geeks’ conference 🙂) — a weekend-long hackathon-makeathon in which all projects must be useless and just-for-fun, and this year there was an exciting twist: all projects were required to incorporate some form of AI.

My group’s project was a speech-to-text-to-speech game, and here’s how it works: the user selects a character to talk to, and then verbally expresses anything they’d like to the character. This spoken input is transcribed and sent to ChatGPT, which responds as if it were the character. The response is then read aloud using text-to-speech technology.

Now that the game is up and running, bringing laughs and fun, I’ve crafted this how-to guide to help you create a similar game on your own. Throughout the article, we’ll also explore the various considerations and decisions we made during the hackathon.

Want to see the full code? [Here is the link](https://github.com/NaomiKriger/speech_to_speech_magician)!

**The Program’s Flow**

Once the server is running, the user will hear the app “talking”, prompting them to choose the figure they want to talk to and start conversing with their selected character. Each time they want to talk out loud — they should press and hold a key on the keyboard while talking. When they finish talking (and release the key), their recording will be transcribed by [Whisper](https://platform.openai.com/docs/guides/speech-to-text/quickstart) (a text-to-speech model by [OpenAI](https://platform.openai.com/docs/introduction/overview)), and the transcription will be sent to [ChatGPT](https://platform.openai.com/docs/guides/gpt/chat-completions-api) for a response. The response will be read out loud using a text-to-speech library, and the user will hear it.

**Implementation**

**Disclaimer**

Note: The project was developed on a Windows operating system and incorporates the pyttsx3 library, which lacks compatibility with M1/M2 chips. As pyttsx3 is not supported on Mac, users are advised to explore alternative text-to-speech libraries that are compatible with macOS environments.

**Openai Integration**

I utilized two OpenAI models: Whisper, for speech-to-text transcription, and the ChatGPT API for generating responses based on the user’s input to their selected figure. While doing so costs money, the pricing model is very cheap, and personally, my bill is still under $1 for all my usage. To get started, I made an initial deposit of $5, and to date, I have not exhausted this deposit, and this initial deposit won’t expire until a year from now.  
I’m not receiving any payment or benefits from OpenAI for writing this.

Once you get your OpenAI API key — set it as an environment variable to use upon making the API calls. Make sure not to push your key to the codebase or any public location, and not to share it unsafely.

**Speech to Text — Create Transcription**

The implementation of the speech-to-text feature was achieved using Whisper, an OpenAI model.

Below is the code snippet for the function responsible for transcription:

async def get\_transcript(audio\_file\_path: str,   
 text\_to\_draw\_while\_waiting: str) -> Optional[str]:  
 openai.api\_key = os.environ.get("OPENAI\_API\_KEY")  
 audio\_file = open(audio\_file\_path, "rb")  
 transcript = None  
  
 async def transcribe\_audio() -> None:  
 nonlocal transcript  
 try:  
 response = openai.Audio.transcribe(  
 model="whisper-1", file=audio\_file, language="en")  
 transcript = response.get("text")  
 except Exception as e:  
 print(e)  
  
 draw\_thread = Thread(target=print\_text\_while\_waiting\_for\_transcription(  
 text\_to\_draw\_while\_waiting))  
 draw\_thread.start()  
  
 transcription\_task = asyncio.create\_task(transcribe\_audio())  
 await transcription\_task  
  
 if transcript is None:  
 print("Transcription not available within the specified timeout.")  
  
 return transcript

This function is marked as asynchronous (async) since the API call may take some time to return a response, and we await it to ensure that the program doesn’t progress until the response is received.

As you can see, the get\_transcript function also invokes the print\_text\_while\_waiting\_for\_transcription function. Why? Since obtaining the transcription is a time-consuming task, we wanted to keep the user informed that the program is actively processing their request and not stuck or unresponsive. As a result, this text is gradually printed as the user awaits the next step.

**String Matching Using FuzzyWuzzy for Text Comparison**

After transcribing the speech into text, we either utilized it as is, or attempted to compare it with an existing string.

The comparison use cases were: selecting a figure from a predefined list of options, deciding whether to continue playing or not, and when opting to continue - deciding whether to choose a new figure or stick with the current one.

In such cases, we wanted to compare the user’s spoken input transcription with the options in our lists, and therefore we decided to use the FuzzyWuzzy library for string matching.

This enabled choosing the closest option from the list, as long as the matching score exceeded a predefined threshold.

Here’s a snippet of our function:

def detect\_chosen\_option\_from\_transcript(  
 transcript: str, options: List[str]) -> str:  
 best\_match\_score = 0  
 best\_match = ""  
  
 for option in options:  
 score = fuzz.token\_set\_ratio(transcript.lower(), option.lower())  
 if score > best\_match\_score:  
 best\_match\_score = score  
 best\_match = option  
  
 if best\_match\_score >= 70:  
 return best\_match  
 else:  
 return ""

If you want to learn more about the FuzzyWuzzy library and its functions — you can check out an article I wrote about it [here](https://towardsdatascience.com/string-comparison-is-easy-with-fuzzywuzzy-library-611cc1888d97).

**Get ChatGPT Response**

Once we have the transcription, we can send it over to ChatGPT to get a response.

For each ChatGPT request, we added a prompt asking for a short and funny response. We also told ChatGPT which figure to pretend to be.

So our function looked as follows:

def get\_gpt\_response(transcript: str, chosen\_figure: str) -> str:  
 system\_instructions = get\_system\_instructions(chosen\_figure)  
 try:  
 return make\_openai\_request(  
 system\_instructions=system\_instructions,   
 user\_question=transcript).choices[0].message["content"]  
 except Exception as e:  
 logging.error(f"could not get ChatGPT response. error: {str(e)}")  
 raise e

and the system instructions looked as follows:

def get\_system\_instructions(figure: str) -> str:  
 return f"You provide funny and short answers. You are: {figure}"

**Text to Speech**

For the text-to-speech part, we opted for a Python library called pyttsx3. This choice was not only straightforward to implement but also offered several additional advantages. It’s free of charge, provides two voice options — male and female — and allows you to select the speaking rate in words per minute (speech speed).

When a user starts the game, they pick a character from a predefined list of options. If we couldn’t find a match for what they said within our list, we’d randomly select a character from our “fallback figures” list. In both lists, each character was associated with a gender, so our text-to-speech function also received the voice ID corresponding to the selected gender.

This is what our text-to-speech function looked like:

def text\_to\_speech(text: str, gender: str = Gender.FEMALE.value) -> None:  
 engine = pyttsx3.init()  
  
 engine.setProperty("rate", WORDS\_PER\_MINUTE\_RATE)  
 voices = engine.getProperty("voices")  
 voice\_id = voices[0].id if gender == "male" else voices[1].id  
 engine.setProperty("voice", voice\_id)  
  
 engine.say(text)  
 engine.runAndWait()

**The Main Flow**

Now that we’ve more or less got all the pieces of our app in place, it’s time to dive into the gameplay! The main flow is outlined below. You might notice some functions we haven’t delved into (e.g. choose\_figure, play\_round), but you can explore the full code by [checking out the repo](https://github.com/NaomiKriger/speech_to_speech_magician). Eventually, most of these higher-level functions tie into the internal functions we’ve covered above.

Here’s a snippet of the main game flow:

import asyncio  
  
from src.handle\_transcript import text\_to\_speech  
from src.main\_flow\_helpers import choose\_figure, start, play\_round, \  
 is\_another\_round  
  
  
def farewell() -> None:  
 farewell\_message = "It was great having you here, " \  
 "hope to see you again soon!"  
 print(f"\n{farewell\_message}")  
 text\_to\_speech(farewell\_message)  
  
  
async def get\_round\_settings(figure: str) -> dict:  
 new\_round\_choice = await is\_another\_round()  
 if new\_round\_choice == "new figure":  
 return {"figure": "", "another\_round": True}  
 elif new\_round\_choice == "no":  
 return {"figure": "", "another\_round": False}  
 elif new\_round\_choice == "yes":  
 return {"figure": figure, "another\_round": True}  
  
  
async def main():  
 start()  
 another\_round = True  
 figure = ""  
  
 while True:  
 if not figure:  
 figure = await choose\_figure()  
  
 while another\_round:  
 await play\_round(chosen\_figure=figure)  
 user\_choices = await get\_round\_settings(figure)  
 figure, another\_round = \  
 user\_choices.get("figure"), user\_choices.get("another\_round")  
 if not figure:  
 break  
  
 if another\_round is False:  
 farewell()  
 break  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 asyncio.run(main())

**The Roads Not Taken**

We had several ideas in mind that we didn’t get to implement during the hackathon. This was either because we did not find an API we were satisfied with during that weekend, or due to the time constraints preventing us from developing certain features. These are the paths we didn’t take for this project:

**Matching the Response Voice with the Chosen Figure’s “Actual” Voice**

Imagine if the user chose to talk to Shrek, Trump, or Oprah Winfrey. We wanted our text-to-speech library or API to articulate responses using voices that matched the chosen figure. However, we couldn’t find a library or API during the hackathon that offered this feature at a reasonable cost. We’re still open to suggestions if you have any =)

**Let the Users Talk to “Themselves”**

Another intriguing idea was to prompt users to provide a vocal sample of themselves speaking. We would then train a model using this sample and have all the responses generated by ChatGPT read aloud in the user’s own voice. In this scenario, the user could choose the tone of the responses (affirmative and supportive, sarcastic, angry, etc.), but the voice would closely resemble that of the user. However, we couldn’t find an API that supported this within the constraints of the hackathon.

**Adding a Frontend to Our Application**

Our initial plan was to include a frontend component in our application. However, due to a last-minute change in the number of participants in our group, we decided to prioritize the backend development. As a result, the application currently runs on the command line interface (CLI) and doesn’t have frontend side.

**Additional Improvements We Have In Mind**

Latency is what bothers me most at the moment.

There are several components in the flow with a relatively high latency that in my opinion slightly harm the user experience. For example: the time it takes from finishing providing the audio input and receiving a transcription, and the time it takes since the user presses a button until the system actually starts recording the audio. So if the user starts talking right after pressing the key — there will be at least one second of audio that won’t be recorded due to this lag.

**Link to the Repo & Credits**

Want to see the whole project? [It’s right here](https://github.com/NaomiKriger/speech_to_speech_magician)!

Also, warm credit goes to [Lior Yardeni](https://www.linkedin.com/in/lioryardeni), my hackathon partner with whom I created this game.

**Summing Up**

In this article, we learned how to create a speech-to-text-to-speech game using Python, and intertwined it with AI. We’ve used the Whisper model by OpenAI for speech recognition, played around with the FuzzyWuzzy library for text matching, tapped into ChatGPT’s conversational magic via their developer API, and brought it all to life with pyttsx3 for text-to-speech. While OpenAI’s services (Whisper and ChatGPT for developers) do come with a modest cost, it’s budget-friendly.

We hope you’ve found this guide enlightening and that it’s motivating you to embark on your projects.

Cheers to coding and fun! 🚀