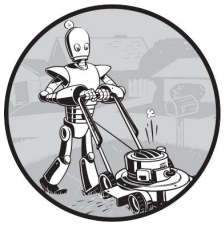
24

TEXT-TO-SPEECH AND SPEECH RECOGNITION ENGINES



This chapter covers a text-to-speech package, pyttsx3, and a speech recognition package, Whisper. Text-to-speech packages can convert text strings into spoken words, then send them to your computer's speakers or save them to an audio file. By adding this new dimension to your programs, you can free the user from having to read text off a screen. For example, a cooking recipe application could read the ingredients list aloud as you move through the kitchen, and your daily script could scrape news articles (or your emails) and then prepare an MP3 to play during your morning commute.

On the other end, speech recognition technologies can convert audio files of spoken words into text string values. You can use this capability to add voice commands to your program or automate the transcription of podcasts. And, unlike with humans, you can always mute the volume on a computer that talks too much.

Both pyttsx3 and Whisper are free to use and don't require an internet connection. The text-to-speech and speech recognition engines featured in this chapter aren't limited to English, and work with most widely spoken human languages.

Text-to-Speech Engine

To produce spoken audio, the pyttsx3 third-party package uses your operating system's built-in text-to-speech engine: Microsoft Speech API (SAPI5) on Windows, NSSpeechSynthesizer on macOS, and eSpeak on Linux. On Linux, you may need to install the engine by running `sudo`

`apt install espeak` from a terminal window. You can install pyttsx3 by running `pip install pyttsx3` from a terminal. Appendix A has full instructions for installing third-party packages.

The name of the package is based on *py* for Python, *tts* for text-to-speech, *x* because it's extended from the original `pytts` package, and *3* because it's for Python 3.

Generating Speech

Producing a computerized voice is a complex topic in computer science. Fortunately, the operating system's text-to-speech engine does the hard work for us, and interacting with this engine is straightforward. Open a new file editor, enter the following code, and save it as *hello_tts.py*:

```
import pyttsx3
engine = pyttsx3.init()
engine.say('Hello. How are you doing?')
engine.runAndWait()  # The computer speaks.
feeling = input('>')
engine.say('Yes. I am feeling ' + feeling + '
as well.')
engine.runAndWait()  # The computer speaks
again.
```

After importing the `pyttsx3` module, we call the `pyttsx3.init()` function to initialize the speech engine. This function returns an `Engine` object. We can pass a string of text to its `say()` method to tell the engine what to speak, but the actual speaking won't begin until we call the `runAndWait()` method. This method blocks (that is, will not return) until the computer has finished speaking the entire string.

The program doesn't produce any text output, because it never calls the `print()` function. Instead, you should hear your computer speak, "Hello. How are you doing?" (Make sure the volume isn't muted.) The user can enter a response from the keyboard, to which the computer should verbally reply, "Yes. I am feeling <*your response*> as well."

The `pyttsx3` module allows you to make some changes to the computer voice. You can pass the strings `'rate'`, `'volume'`, and `'voices'` to the `getProperty()` method of the `Engine` object to view its current settings. Enter the following into the interactive shell:

```
>>> import pyttsx3
>>> engine = pyttsx3.init()
>>> engine.getProperty('volume')
1.0
>>> engine.getProperty('rate')
200
>>> engine.getProperty('voices')
[<pyttsx3.voice.Voice object at
0x0000029DA7FB4B10>,
<pyttsx3.voice.Voice object at
0x0000029DAA3DAAD0>]
```

Note that the output may differ on your computer. The volume setting is a float, where `1.0` indicates 100 percent. The computer voice speaks at a rate of 200 words per minute. Continue this example with the following code:

```
>>> for voice in
engine.getProperty('voices'):  # List all the
available voices.
...     print(voice.name, voice.gender,
voice.age, voice.languages)
...
Microsoft David Desktop - English (United
States) None None []
Microsoft Zira Desktop - English (United
States) None None []
```

On my Windows laptop with the *English (United States)* language, `getProperty('voices')` returns two `Voice` objects. (Note that this string is the plural `'voices'` and not the singular `'voice'`.) These `Voice` objects have `name`, `gender`, and `age` attributes, though `gender` and `age` are set to `None` when the operating system doesn't store that information. The `languages` attribute is a list of strings of languages the voice supports, which is a blank list if that information is unknown.

Let's continue the interactive shell example by calling the `setProperty()` method to change these settings:

```
>>> engine.setProperty('rate', 300)
>>> engine.setProperty('volume', 0.5)
>>> voices = engine.getProperty('voices')
>>> engine.setProperty('voice', voices[1].id)
>>> engine.say('The quick brown fox jumps
over the yellow lazy dog.')
>>> engine.runAndWait()
```

In this example, we've changed the speaking rate to 300 words per minute and set the volume to 50 percent by passing `0.5` for the `'volume'` rate. We then changed the voice to the female "Zira" voice that Windows provides by passing the `id` attribute of the `Voice` object at index `1` of the list that `getProperty('voices')` returned. Also note that to set the voice, we use the singular `'voice'` string and not the plural `'voices'` string.

Saving Speech Audio to WAV Files

The `pyttsx3` module's `save_to_file()` method can save the generated speech to a WAV file (with the *.wav* file extension). Enter the following into the interactive shell:

```
>>> import pyttsx3
>>> engine = pyttsx3.init()
>>> engine.save_to_file('Hello. How are you
doing?', 'hello.wav')
>>> engine.runAndWait()  # The computer
creates hello.wav.
```

The first argument to `save_to_file()` is a string of the speech to generate, while the second string argument is the filename of the *.wav* file. The text string could be a short sentence, as in the interactive shell example, or it could be pages of text. On my computer, `pyttsx3` was able to turn a string of 1,800 words into a 10-minute-long audio file in about two seconds. It's important to note that calling

`save_to_file()` alone isn't enough. You must also call the `runAndWait()` method before Python will create the *.wav* file.

The `pyttsx3` module can save *.wav* files only, not *.mp3* files or any other audio format.

Speech Recognition

Whisper is a speech recognition system that can recognize multiple languages. Given an audio or video file, Whisper can return the speech as text in a Python string. It also returns the start and end times for groups of words, which you can use to generate subtitle files.

Install Whisper by running **`pip install openai-whisper`** from the terminal. (Note that the name of the speech recognition package is `openai-whisper`; the `whisper` package on the PyPI website refers to something else.) This is a large download and may take several minutes to install. Also, the first time you call the `load_model()` function, your computer will download the speech recognition model, which can be hundreds of megabytes or more in size.

Let's say you have an audio file named *hello.wav* in the current working directory. (Whisper can also handle *.mp3* and several other audio formats.) You could enter the following into the interactive shell:

```
>>> import whisper
>>> model = whisper.load_model('base')
>>> result = model.transcribe('hello.wav')
>>> print(result['text'])
Hello. How are you doing?
```

After importing the `whisper` module, you must load the speech recognition model to use by calling the `whisper.load_model()` function, passing it the string of the trained machine learning model you want to use: `'tiny'`, `'base'`, `'small'`, `'medium'`, or `'large-v3'`. (New models will continue to be released as well.) The smaller of these models can transcribe audio more quickly, but the larger models will do so more accurately, even when the audio has ambient noise in the background.

The first time you load a model, your computer must be connected to the internet so that the `whisper` module can download it from OpenAI's servers. Table 24-1 lists the model names as strings you could pass to the `whisper.load_model()` function, along with their file size, memory usage, and the results of some runtime tests on my laptop.

Table 24-1: Properties of Whisper Speech Recognition Models

Model name	Model file size	Estimated required memory	Runtime for a 10-word, 3-second audio sample	Runtime for an 1,800-word, 15-minute audio sample
`'tiny'`	74MB	1GB	1.9s	1m 20s
`'base'`	142MB	1GB	3.0s	2m 34s
`'small'`	472MB	2GB	9.6s	6m 37s
`'medium'`	1.5GB	5GB	28.6s	20m 17s
`'large-v3'`	3GB	10GB	51.9s	32m 58s

My personal opinion is that for 99 percent of purposes, the `'base'` model should be suitable and the `'medium'` model is good enough when more accuracy is needed. All models produce errors, so the output should always undergo human review. You can experiment with larger models if you find many transcription errors in the text, or smaller models if Whisper is taking too long to transcribe the audio. As you can see in Table 24-1, however, there is a substantial difference in the two minutes and 34 seconds that the `'base'` model takes to transcribe 15 minutes of audio and the nearly 33 minutes that the `'large-v3'` model takes.

With the `model.Whisper` object that `whisper.load_model()` returns, you can call the `transcribe()` method to perform the actual transcription. Pass the method a string of the audio file's name. This method will take anywhere from a few seconds to a few hours to run, depending on the model and length of the audio file. Whisper can accept any audio or video file, and converts it to the format it requires automatically.

Whisper can automatically detect the language of the audio, but you can specify the language by passing a language keyword argument to transcribe, such as `model.transcribe('hello.wav', language='English')`. To find the languages that Whisper supports, you can run `whisper --help` from the terminal. Whisper is pretty good (but never perfect) at guessing where it should insert punctuation and at capitalizing proper names. However, you should always review the output to clean up any mistakes.

The dictionary that `model.transcribe()` returns has several key-value pairs, but the `'text'` key contains the string of the transcription.

By default, Whisper uses your CPU to transcribe text, but if your computer has a 3D graphics card, you can greatly speed up

transcriptions by setting it up to use the graphics processing unit (GPU). You'll find these setup instructions in the online documentation at *https://github.com/openai/whisper*. If your computer has an NVIDIA graphics card, you can follow the instructions in Appendix A to install packages for faster speech recognition. To use the GPU, replace the `whisper.load_model('base')` code in this chapter with `whisper.load_model('base', device='cuda')`.

You can find several additional options for Whisper in its online documentation.

Creating Subtitle Files

In addition to the transcribed audio, Whisper's results dictionary contains timing information that identifies the text's location in the audio file. You can use this text and timing data to generate subtitle files that other software can ingest. The two most common subtitle file formats are SRT SubRip Subtitle (with the *.srt* extension) and VTT Web Video Text Tracks (with the *.vtt* file extension). SRT is an older and more widespread standard, while modern video websites generally use VTT. The formats are similar. For example, here is the first part of an SRT file:

```
1
00:00:00,000 --> 00:00:05,640
Dinosaurs are a diverse group of reptiles of
the clade dinosauria. They first

2
00:00:05,640 --> 00:00:14,960
appeared during the triassic period. Between
245 and 233.23 million years ago.
--snip--
```

Compare it with the first part of a VTT file for the same subtitles:

```
WEBVTT

00:00.000 --> 00:05.640
Dinosaurs are a diverse group of reptiles of
```

```
the clade dinosauria. They first

00:05.640 --> 00:14.960
appeared during the triassic period. Between
245 and 233.23 million years ago.
--snip--
```

These files both indicate that the words “Dinosaurs are a diverse group ...” appear between the start of the transcribed audio file (at 0 seconds) and the 5.640-second mark.

Whisper can also output its results as TSV data (with the *.tsv* extension) or JSON data (with the *.json* extension). TSV isn’t an official subtitles format, but it may be useful if you need to export the text and timing data to, say, another Python program that can read it using the `csv` module covered in Chapter 18. TSV-formatted subtitles look like the following:

```
start  end    text
0      5640   Dinosaurs are a diverse group
of reptiles of the clade dinosauria. They
5640   14960  appeared during the triassic
period. Between 245 and 233.23 million years
ago.
--snip--
```

To create these subtitle files, add two extra lines of code after calling `model.transcribe()`:

```
>>> import whisper
>>> model = whisper.load_model('base')
>>> result = model.transcribe('hello.wav')
❶ >>> write_function =
whisper.utils.get_writer('srt', '.')
❷ >>> write_function(result, 'audio')
```

The `whisper.utils.get_writer()` function ❶ accepts the subtitle file format as a string (`'srt'`, `'vtt'`, `'txt'`, `'tsv'`, or `'json'`) and the folder in which to save the file (with the `'.'` string meaning the current working directory). The `get_writer()` function returns a function to pass the transcription results. (This is a rather odd way to create the transcript files, but it's the way the `whisper` module is designed.) We store it in a variable named `write_function`, which we can then treat as a function and call, passing the `result` dictionary and the filename for the subtitle file ❷. These two lines of code produce an SRT-formatted file named *audio.srt* in the current working directory, using the text and timing information in the `result` dictionary.

Downloading Videos from Websites

While downloading audio files to transcribe with Whisper's speech recognition is often straightforward, video websites such as YouTube often don't make it easy to download their content. The `yt-dlp` module allows Python scripts to download videos from YouTube and hundreds of other video websites so that you can watch them offline. Appendix A has instructions for installing `yt-dlp`. Once it's installed, the following code will download the video at the given URL:

```
>>> import yt_dlp
>>> video_url = 'https://www.youtube.com/
watch?v=kSrnLbioN6w'
>>> with yt_dlp.YoutubeDL() as ydl:
...     ydl.download([video_url])
...
```

Note that the `ydl.download()` function expects a list of video URLs, which is why we put the `video_url` string inside a list before passing it to the function call. The video's filename is based on the title on the video website, and could have a *.mp4*, *.mkv*, or other video format file extension. You'll see a lot of debugging output as the video downloads.

The video website could refuse the download due to age or login requirements, geographic restrictions, or anti-web scraping measures. If you're encountering errors, the first step you should try is installing the latest version of `yt-dlp`, which updates to stay compatible as video websites change their layout.

You can read about `yt-dlp`'s many configuration options in the online documentation at *https://pypi.org/project/yt-dlp/*. For example, you can extract the audio from a YouTube video by passing a dictionary of configuration settings to the `yt_dlp.YoutubeDL()` function:

```
>>> import yt_dlp
>>> video_url = 'https://www.youtube.com/
watch?v=kSrnLbioN6w'
>>> options = {
...     ❶ 'quiet': True,  # Suppress the
output.
...       'no_warnings': True,  # Suppress
warnings.
...     ❷ 'outtmpl': 'downloaded_content.%
(ext)s',
...       'format': 'm4a/bestaudio/best',
...       'postprocessors': [{
# Extract audio using ffmpeg.
...           'key': 'FFmpegExtractAudio',
...           'preferredcodec': 'm4a',
...       }]
...}
...
>>> with yt_dlp.YoutubeDL(options) as ydl:
...     ydl.download([video_url])
...
```

The `'quiet': True` and `'no_warnings': True` key-value pairs ❶ prevent the verbose debugging output. The `options` dictionary passed to `yt_dlp.Youtube()` tells it to download the video and then extract the audio to a file named *downloaded_content.m4a* ❷. (The file extension may differ if the video has a different audio format, though the *.m4a* format is the most popular one.) If we hadn't set the `'outtmpl': 'downloaded_content.%(ext)s'` key-value pair in the `options` dictionary, the downloaded

filename would be based on the video's title (excluding characters not allowed in filenames, such as question marks and colons).

To get the exact filename, we can use glob patterns, discussed in Chapter 10. We know the main part of the file is `'downloaded_content'`, but the file extension could be any audio format. The following code uses `Path` objects to find the exact downloaded filename:

```
>>> from pathlib import Path
>>> matching_filenames =
list(Path().glob('downloaded_content.*'))
>>> downloaded_filename =
str(matching_filenames[0])
>>> downloaded_filename
'downloaded_content.m4a'
```

Setting the filename makes it easier for the code to use this file later, such as by running it through Whisper speech recognition. The `'base'` and `'medium'` models create much higher-quality subtitles than YouTube's current autogenerated subtitles.

If you just want to download the information about a given video, you can tell `yt-dlp` to skip the file and download only its metadata with the following code:

```
>>> import yt_dlp, json
>>> video_url = 'https://www.youtube.com/
watch?v=kSrnLbioN6w'
>>> options = {
...     'quiet': True,  # Suppress the
output.
...     'no_warnings': True,  # Suppress
warnings.
...   ❶ 'skip_download': True,  # Do not
download the video.
...}
...
>>> with yt_dlp.YoutubeDL(options) as ydl:
```

```
...   ❷ info = ydl.extract_info(video_url)
...   ❸ json_info = ydl.sanitize_info(info)
...     print('TITLE:', json_info['title'])
# Print the video title.
...     print('KEYS:', json_info.keys())
...     with open('metadata.json', 'w',
encoding='utf-8') as json_file:
...       ❹ json_file.write(json.dumps(json_i
nfo))
...
TITLE: Beyond the Basic Stuff with Python -
Al Sweigart - Part 1
KEYS: dict_keys(['id', 'title', 'formats',
'thumbnails', 'thumbnail',
'description', 'channel_id', 'channel_url',
'duration', 'view_count',
'average_rating', 'age_limit', 'webpage_url',
--snip--
```

If we just want the metadata for the video and not the video itself, we can include the `'skip_download': True` key-value pair ❶ in the `options` dictionary passed to `yt_dlp.YoutubeDL()`. The `ydl.extract_info()` method call returns a dictionary of information about the video ❷. Some of this data might not be properly formatted as JSON (discussed in Chapter 18), but we can get it in a JSON-compatible form by calling `ydl.sanitize()` ❸. The dictionary that the `sanitize()` method returns has several keys, including `'title'` for the name of the video, `'duration'` for the video's length in seconds, and so on. Our code here additionally writes this JSON data to a file named *metadata.json* ❹.

Summary

One of Python's great strengths is its vast ecosystem of third-party packages for tasks such as text-to-speech and speech recognition. These packages take some of the hardest problems in computer science and make them available to your programs with just a few lines of code.

The pyttsx3 package does text-to-speech using your computer's speech engine to create audio that you can either play from the speakers or save to a *.wav* file. The Whisper speech recognition system uses several underlying models to transcribe the words of an audio file. These models have different sizes; the smaller models transcribe faster with less accuracy, while larger models are slower but more accurate. They work for many human languages, not just English. Whisper runs on your computer and doesn't connect to online servers except to download the model on first use.

The speech engines that these Python packages use have seen a large leap in quality that wasn't available before the 2020s. Python is an excellent "glue" language that allows your scripts to connect with this software so that you can add these speech features to your own programs with just a few lines of code. If you want to learn more about text-to-speech and speech recognition, you can find many fun example projects in *Make Python Talk* by Mark Liu (No Starch Press, 2021).

Practice Questions

1. How can you make pyttsx3's voice speak faster?
2. What audio format does `pyttsx3` save to?
3. Do pyttsx3 and Whisper rely on online services?
4. Do pyttsx3 and Whisper support other languages besides English?
5. What is the name of Whisper's default machine learning model for speech recognition?
6. What are two common subtitle text file formats?
7. Can `yt-dlp` download videos from websites besides YouTube?

Practice Programs

For practice, write programs to do the following tasks.

Adding Voice to Guess the Number

Revisit the guess the number game from Chapter 3 and add a voice feature to it. Replace all of the function calls to `print()` with calls to a function named `speak()`. Next, define the `speak()` function to accept a string argument (just like `print()` did), but have it both print the string to the screen and say it out loud. For example, you'll replace this line of code

```
print('I am thinking of a number between 1
and 20.')
```

with this line of code:

```
speak('I am thinking of a number between 1
and 20.')
```

To make full use of the speech-generation feature, let's change the `'Your guess is too low.'` and `'Your guess is too high.'` text to say the player's guess. For example, the computer should say, "Your guess, 42, is too low." You can also add a voice feature to other projects in this book, such as the rock, paper, scissors game.

Singing "99 Bottles of Beer"

Cumulative songs are songs whose verses repeat with additions or slight changes. The songs "99 Bottles of Beer" and "The 12 Days of Christmas" are examples of cumulative songs. Write a program that sings (or at least speaks) the lyrics in "99 Bottles of Beer":

```
99 bottles of beer on the wall,
99 bottles of beer,
Take one down, pass it around,
98 bottles of beer on the wall.
```

These lyrics repeat, with one fewer bottle each time. The song continues until it reaches zero bottles, at which point the last line is "No more bottles of beer on the wall." (You may wish to have the program start at 2 or 3 instead of 99 to make testing easier.)

YouTube Transcriber

Write a program that glues together the features of `yt-dlp` and Whisper to automatically download YouTube videos and produce subtitle files in the *.srt* format. The input can be a list of URLs to download and transcribe. You can also add options to produce different subtitle formats. Python is an excellent "glue language" for combining the capabilities of different modules.