**CONVERT SPEECH TO TEXT**

The first component of speech recognition is, of course, speech. Speech must be converted from physical sound to an electrical signal with a microphone, and then to digital data with an analog-to-digital converter. Once digitized, several models can be used to transcribe the audio to text.

Most modern speech recognition systems rely on what is known as a [Hidden Markov Model](https://en.wikipedia.org/wiki/Hidden_Markov_model) (HMM). This approach works on the assumption that a speech signal, when viewed on a short enough timescale (say, ten milliseconds), can be reasonably approximated as a stationary process-that is, a process in which statistical properties do not change over time.

In a typical HMM, the speech signal is divided into 10-millisecond fragments. The power spectrum of each fragment, which is essentially a plot of the signal’s power as a function of frequency, is mapped to a vector of real numbers known as [cepstral](https://en.wikipedia.org/wiki/Cepstrum) coefficients. The dimension of this vector is usually small-sometimes as low as 10, although more accurate systems may have dimension 32 or more. The final output of the HMM is a sequence of these vectors.

To decode the speech into text, groups of vectors are matched to one or more [phonemes](https://en.wikipedia.org/wiki/Phoneme)-a fundamental unit of speech. This calculation requires training, since the sound of a phoneme varies from speaker to speaker, and even varies from one utterance to another by the same speaker. A special algorithm is then applied to determine the most likely word (or words) that produce the given sequence of phonemes.

One can imagine that this whole process may be computationally expensive. In many modern speech recognition systems, neural networks are used to simplify the speech signal using techniques for feature transformation and dimensionality reduction before HMM recognition. Voice activity detectors (VADs) are also used to reduce an audio signal to only the portions that are likely to contain speech. This prevents the recognizer from wasting time analyzing unnecessary parts of the signal.

Fortunately, as a Python programmer, you don’t have to worry about any of this. A number of speech recognition services are available for use online through an API, and many of these services offer [Python SDKs](https://realpython.com/api-integration-in-python/).

**Picking a Python Speech Recognition Package**

A handful of packages for speech recognition exist on PyPI. A few of them include:

* [apiai](https://pypi.org/project/apiai/)
* [assemblyai](https://pypi.org/project/assemblyai/)
* [google-cloud-speech](https://pypi.org/project/google-cloud-speech/)
* [pocketsphinx](https://pypi.org/project/pocketsphinx/)
* [SpeechRecognition](https://pypi.org/project/SpeechRecognition/)
* [watson-developer-cloud](https://pypi.org/project/watson-developer-cloud/)
* [wit](https://pypi.org/project/wit/)

There is one package that stands out in terms of ease-of-use: Speech Recognition.

Recognizing speech requires audio input, and SpeechRecognition makes retrieving this input really easy. Instead of having to build scripts for accessing microphones and processing audio files from scratch, SpeechRecognition will have you up and running in just a few minutes.

The SpeechRecognition library acts as a wrapper for several popular speech APIs and is thus extremely flexible. One of these-the Google Web Speech API-supports a default API key that is hard-coded into the SpeechRecognition library. That means you can get off your feet without having to sign up for a service.

## Installing Speech Recognition

## SpeechRecognition is compatible with Python 2.6, 2.7 and 3.3+. You can install SpeechRecognition from a terminal with pip:

$ pip install SpeechRecognition

Once installed, you should verify the installation by opening an interpreter session and typing:

>>> import speech\_recognition as sr

>>> sr.\_\_version\_\_

'3.8.1' (last version)

## The Recognizer Class

The primary purpose of a Recognizer instance is, of course, to recognize speech. Each instance comes with a variety of settings and functionality for recognizing speech from an audio source.

Creating a Recognizer instance is easy. In your current interpreter session, just type:

>>> r = sr.Recognizer()

Each Recognizer instance has seven methods for recognizing speech from an audio source using various APIs. These are:

* recognize\_bing(): [Microsoft Bing Speech](https://azure.microsoft.com/en-us/services/cognitive-services/speech/)
* recognize\_google(): [Google Web Speech API](https://w3c.github.io/speech-api/speechapi.html)
* recognize\_google\_cloud(): [Google Cloud Speech](https://cloud.google.com/speech/) - requires installation of the google-cloud-speech package
* recognize\_houndify(): [Houndify](https://www.houndify.com/) by SoundHound
* recognize\_ibm(): [IBM Speech to Text](https://www.ibm.com/watson/services/speech-to-text/)
* recognize\_sphinx(): [CMU Sphinx](https://cmusphinx.github.io/) - requires installing PocketSphinx
* recognize\_wit(): [Wit.ai](https://wit.ai/)

Since SpeechRecognition ships with a default API key for the Google Web Speech API, you can get started with it right away. The other six APIs all require authentication with either an API key or a username/password combination.

## Working With Audio Files

Before you continue, you’ll need to download the “harvard.wav” file. Make sure you save it to the same directory in which your Python interpreter session is running.

SpeechRecognition makes working with audio files easy thanks to its handy AudioFile class. This class can be initialized with the path to an audio file and provides a context manager interface for reading and working with the file’s contents.

### Supported File Types

Currently, SpeechRecognition supports the following file formats:

* WAV: must be in PCM/LPCM format
* AIFF
* AIFF-C
* FLAC: must be native FLAC format; OGG-FLAC is not supported

### Using record() to Capture Data From a File

Type the following into your interpreter session to process the contents of the “harvard.wav” file:

>>> harvard = sr.AudioFile('harvard.wav')

>>> with harvard as source:

... audio = r.record(source)

The context manager opens the file and reads its contents, storing the data in an AudioFile instance called source. Then the record() method records the data from the entire file into an AudioData instance.

You can now invoke recognize\_google() to attempt to recognize any speech in the audio.

>>> r.recognize\_google(audio)

### Capturing Segments With offset and duration

You can use duration and offset, if you only want to capture a portion of the speech in a file. With offset you can start anywhere, and with duration you can stop anywhere.

>>> with harvard as source:

... audio = r.record(source, offset=4, duration=3)

>>> recognizer.recognize\_google(audio)

### The Effect of Noise on Speech Recognition

 Allaudio recordings have some degree of noise in them, and un-handled noise can wreck the accuracy of speech recognition apps.

To get a feel for how noise can affect speech recognition, download the “jackhammer.wav” file. As always, make sure you save this to your interpreter session’s working directory.

This file has the phrase “the stale smell of old beer lingers” spoken with a loud jackhammer in the background.

What happens when you try to transcribe this file?

>>> jackhammer = sr.AudioFile('jackhammer.wav')

>>> with jackhammer as source:

... audio = r.record(source)

...

>>> r.recognize\_google(audio)

'the snail smell of old gear vendors'

Way off!

So how do you deal with this? One thing you can try is using the adjust\_for\_ambient\_noise() method of the Recognizer class.

>>> with jackhammer as source:

... r.adjust\_for\_ambient\_noise(source)

... audio = r.record(source)

...

>>> r.recognize\_google(audio)

'still smell of old beer vendors'

That got you a little closer to the actual phrase, but it still isn’t perfect. Also, “the” is missing from the beginning of the phrase. Why is that?

The adjust\_for\_ambient\_noise() method reads the first second of the file stream and calibrates the recognizer to the noise level of the audio. Hence, that portion of the stream is consumed before you call record() to capture the data.

You can adjust the time-frame that adjust\_for\_ambient\_noise() uses for analysis with the duration keyword argument. This argument takes a numerical value in seconds and is set to 1 by default. Try lowering this value to 0.5.

>>> with jackhammer as source:

... r.adjust\_for\_ambient\_noise(source, duration=0.5)

... audio = r.record(source)

...

>>> r.recognize\_google(audio)

'the snail smell like old Beer Mongers'

Well, that got you “the” at the beginning of the phrase, but now you have some new issues! Sometimes it isn’t possible to remove the effect of the noise—the signal is just too noisy to be dealt with successfully. That’s the case with this file.

When working with noisy files, it can be helpful to see the actual API response. Most APIs return a [JSON string](https://realpython.com/python-json/) containing many possible transcriptions. The recognize\_google() method will always return the *most likely* transcription unless you force it to give you the full response.

You can do this by setting the show\_all keyword argument of the recognize\_google() method to True.

>>> r.recognize\_google(audio, show\_all=True)

{'alternative': [

{'transcript': 'the snail smell like old Beer Mongers'},

{'transcript': 'the still smell of old beer vendors'},

{'transcript': 'the snail smell like old beer vendors'},

{'transcript': 'the stale smell of old beer vendors'},

{'transcript': 'the snail smell like old beermongers'},

{'transcript': 'destihl smell of old beer vendors'},

{'transcript': 'the still smell like old beer vendors'},

{'transcript': 'bastille smell of old beer vendors'},

{'transcript': 'the still smell like old beermongers'},

{'transcript': 'the still smell of old beer venders'},

{'transcript': 'the still smelling old beer vendors'},

{'transcript': 'musty smell of old beer vendors'},

{'transcript': 'the still smell of old beer vendor'}

], 'final': True}

As you can see, recognize\_google() returns a dictionary with the key 'alternative' that points to a list of possible transcripts. The structure of this response may vary from API to API and is mainly useful for debugging.

## Working With Microphones

To access your microphone with SpeechRecognizer, you’ll have to install the PyAudio package.

### Installing PyAudio

The process for installing PyAudio will vary depending on your operating system.

#### Debian Linux

If you’re on Debian-based Linux (like Ubuntu) you can install PyAudio with apt:

$ sudo apt-get install python-pyaudio python3-pyaudio

Once installed, you may still need to run pip install pyaudio, especially if you are working in a virtual environment.

#### macOS

For macOS, first you will need to install PortAudio with Homebrew, and then install PyAudio with pip:

$ brew install portaudio

$ pip install pyaudio

#### Windows

On Windows, you can install PyAudio with pip:

$ pip install pyaudio

### The Microphone Class

Now, instead of using an audio file as the source, you will use the default system microphone. You can access this by creating an instance of the Microphone class.

>>> mic = sr.Microphone()

### Using listen() to Capture Microphone Input

Just like the AudioFile class, Microphone is a context manager. You can capture input from the microphone using the listen() method of the Recognizer class inside of the with block. This method takes an audio source as its first argument and records input from the source until silence is detected.

>>> with mic as source:

... audio = r.listen(source)

>>> r.recognize\_google(audio)

## Recognizing Speech in Languages Other Than English

To recognize speech in a different language, set the language keyword argument of the recognize\_\*() method to a string corresponding to the desired language. For example, the following recognizes Turkish speech in an audio file:

r.recognize\_google(audio, language='tr-TR')