The Ultimate Guide To Speech Recognition With Python

by David Amos 87 Comments advanced data-science machine-learning

Table of Contents

Have you ever wondered how to add speech recognition to your Python project? If so, then keep reading! It's easier than you might think.

Far from a being a fad, the overwhelming success of speech-enabled products like Amazon Alexa has proven that some degree of speech support will be an essential aspect of household tech for the foreseeable future. If you think about it, the reasons why are pretty obvious. Incorporating speech recognition into your Python application offers a level of interactivity and accessibility that few technologies can match.

The accessibility improvements alone are worth considering. Speech recognition allows the elderly and the physically and visually impaired to interact with state-of-the-art products and services quickly and naturally—no GUI needed!

Best of all, including speech recognition in a Python project is really simple. In this guide, you'll find out how. You'll learn:

- How speech recognition works,
- What packages are available on PyPI; and
- How to install and use the SpeechRecognition package—a full-featured and easy-to-use Python speech recognition library.

In the end, you'll apply what you've learned to a simple "Guess the Word" game and see how it all comes together.

Free Bonus: Click here to download a Python speech recognition sample project with full source code that you can use as a basis for your own speech recognition apps.

How Speech Recognition Works – An Overview

Before we get to the nitty-gritty of doing speech recognition in Python, let's take a moment to talk about how speech recognition works. A full discussion would fill a book, so I won't bore you with all of the technical details here. In fact, this section is not pre-requisite to the rest of the tutorial. If you'd like to get straight to the point, then feel free to skip ahead.

Speech recognition has its roots in research done at Bell Labs in the early 1950s. Early systems were limited to a single speaker and had limited vocabularies of about a dozen words. Modern speech recognition systems have come a long way since their ancient counterparts. They can recognize speech from multiple speakers and have enormous vocabularies in numerous languages.

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 (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 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—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.

Picking a Python Speech Recognition Package

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

- apiai
- assemblyai
- google-cloud-speech
- pocketsphinx
- SpeechRecognition
- watson-developer-cloud
- wit

Some of these packages—such as wit and apiai—offer built-in features, like natural language processing for identifying a speaker's intent, which go beyond basic speech recognition. Others, like google-cloud-speech, focus solely on speech-to-text conversion.

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

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.

The flexibility and ease-of-use of the SpeechRecognition package make it an excellent choice for any Python project. However, support for every feature of each API it wraps is not guaranteed. You will need to spend some time researching the available options to find out if SpeechRecognition will work in your particular case.

So, now that you're convinced you should try out SpeechRecognition, the next step is getting it installed in your environment.

Installing SpeechRecognition

SpeechRecognition is compatible with Python 2.6, 2.7 and 3.3+, but requires some additional installation steps for Python 2. For this tutorial, I'll assume you are using Python 3.3+.

You can install SpeechRecognition from a terminal with pip:

Shell

```
$ pip install SpeechRecognition
```

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

```python
>>> import speech_recognition as sr
>>> sr.__version__
'3.8.1'
```

Note: The version number you get might vary. Version 3.8.1 was the latest at the time of writing.

Go ahead and keep this session open. You'll start to work with it in just a bit.

SpeechRecognition *will* work out of the box if all you need to do is work with existing audio files. Specific use cases, however, require a few dependencies. Notably, the PyAudio package is needed for capturing microphone input.

You'll see which dependencies you need as you read further. For now, let's dive in and explore the basics of the package.

The `Recognizer` Class

All of the magic in SpeechRecognition happens with 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:

```python
>>> 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
- `recognize_google()`: Google Web Speech API
- `recognize_google_cloud()`: Google Cloud Speech - requires installation of the google-cloud-speech package
- `recognize_houndify()`: Houndify by SoundHound
- `recognize_ibm()`: IBM Speech to Text
- `recognize_sphinx()`: CMU Sphinx - requires installing PocketSphinx
- `recognize_wit()`: Wit.ai

Of the seven, only `recognize_sphinx()` works offline with the CMU Sphinx engine. The other six all require an internet connection.

A full discussion of the features and benefits of each API is beyond the scope of this tutorial. Since SpeechRecognition ships with a default API key for the Google Web Speech API, you can get started with it right away. For this reason, we'll use the Web Speech API in this guide. The other six APIs all require authentication with either an API key or a username/password combination. For more information, consult the SpeechRecognition docs.

Caution: The default key provided by SpeechRecognition is for testing purposes only, and **Google may revoke it**

at any time. It is *not* a good idea to use the Google Web Speech API in production. Even with a valid API key, you'll be limited to only 50 requests per day, and there is no way to raise this quota . Fortunately, SpeechRecognition's interface is nearly identical for each API, so what you learn today will be easy to translate to a real-world project.

Each `recognize_*()` method will throw a `speech_recognition.RequestError` exception if the API is unreachable. For `recognize_sphinx()`, this could happen as the result of a missing, corrupt or incompatible Sphinx installation. For the other six methods, `RequestError` may be thrown if quota limits are met, the server is unavailable, or there is no internet connection.

Ok, enough chit-chat. Let's get our hands dirty. Go ahead and try to call `recognize_google()` in your interpreter session.

Python >>>

```python
>>> r.recognize_google()
```

What happened?

You probably got something that looks like this:

Python

```python
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: recognize_google() missing 1 required positional argument: 'audio_data'
```

You might have guessed this would happen. How could something be recognized from nothing?

All seven `recognize_*()` methods of the `Recognizer` class require an `audio_data` argument. In each case, `audio_data` must be an instance of SpeechRecognition's `AudioData` class.

There are two ways to create an `AudioData` instance: from an audio file or audio recorded by a microphone. Audio files are a little easier to get started with, so let's take a look at that first.

Working With Audio Files

Before you continue, you'll need to download an audio file. The one I used to get started, "harvard.wav," can be found here. 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

If you are working on x-86 based Linux, macOS or Windows, you should be able to work with FLAC files without a problem. On other platforms, you will need to install a FLAC encoder and ensure you have access to the `flac` command line tool. You can find more information here if this applies to you.

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

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

Python >>>

```python
>>> 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 confirm this by checking the type of `audio`:

Python >>>

```python
>>> type(audio)
<class 'speech_recognition.AudioData'>
```

You can now invoke `recognize_google()` to attempt to recognize any speech in the audio. Depending on your internet connection speed, you may have to wait several seconds before seeing the result.

Python >>>

```python
>>> r.recognize_google(audio)
'the stale smell of old beer lingers it takes heat
to bring out the odor a cold dip restores health and
zest a salt pickle taste fine with ham tacos al
Pastore are my favorite a zestful food is the hot
cross bun'
```

Congratulations! You’ve just transcribed your first audio file!

If you’re wondering where the phrases in the “harvard.wav” file come from, they are examples of Harvard Sentences. These phrases were published by the IEEE in 1965 for use in speech intelligibility testing of telephone lines. They are still used in VoIP and cellular testing today.

The Harvard Sentences are comprised of 72 lists of ten phrases. You can find freely available recordings of these phrases on the Open Speech Repository website. Recordings are available in English, Mandarin Chinese, French, and Hindi. They provide an excellent source of free material for testing your code.

Capturing Segments With `offset` and `duration`

What if you only want to capture a portion of the speech in a file? The `record()` method accepts a `duration` keyword argument that stops the recording after a specified number of seconds.

For example, the following captures any speech in the first four seconds of the file:

Python >>>

```python
>>> with harvard as source:
...     audio = r.record(source, duration=4)
...
>>> r.recognize_google(audio)
'the stale smell of old beer lingers'
```

The `record()` method, when used inside a `with` block, always moves ahead in the file stream. This means that if you record once for four seconds and then record again for four seconds, the second time returns the four seconds of audio *after* the first four seconds.

Python >>>

```python
>>> with harvard as source:
...     audio1 = r.record(source, duration=4)
...     audio2 = r.record(source, duration=4)
...
>>> r.recognize_google(audio1)
'the stale smell of old beer lingers'
>>> r.recognize_google(audio2)
'it takes heat to bring out the odor a cold dip'
```

Notice that `audio2` contains a portion of the third phrase in the file. When specifying a duration, the recording might stop mid-phrase—or even mid-word—which can hurt the accuracy of the transcription. More on this in a bit.

In addition to specifying a recording duration, the `record()` method can be given a specific starting point using the `offset` keyword argument. This value represents the number of seconds from the beginning of the file to ignore before starting to record.

To capture only the second phrase in the file, you could start with an offset of four seconds and record for, say, three seconds.

Python >>>

```python
>>> with harvard as source:
...     audio = r.record(source, offset=4, duration=3)
...
>>> recognizer.recognize_google(audio)
'it takes heat to bring out the odor'
```

The `offset` and `duration` keyword arguments are useful for segmenting an audio file *if* you have prior knowledge of the structure of the speech in the file. However, using them hastily can result in poor transcriptions. To see this effect, try the following in your interpreter:

Python >>>

```python
>>> with harvard as source:
...     audio = r.record(source, offset=4.7, duration=2.8)
...
>>> recognizer.recognize_google(audio)
'Mesquite to bring out the odor Aiko'
```

By starting the recording at 4.7 seconds, you miss the “it t” portion a the beginning of the phrase “it takes heat to bring out the odor,” so the API only got “akes heat,” which it matched to “Mesquite.”

Similarly, at the end of the recording, you captured “a co,” which is the beginning of the third phrase “a cold dip restores health and zest.” This was matched to “Aiko” by the API.

There is another reason you may get inaccurate transcriptions. Noise! The above examples worked well because the audio file is reasonably clean. In the real world, unless you have the opportunity to process audio files beforehand, you can not expect the audio to be noise-free.

The Effect of Noise on Speech Recognition

Noise is a fact of life. *All* audio 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 here. 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?

Python >>>

```python
>>> 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.

Python >>>

```python
>>> 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.

Python >>>

```python
>>> 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.

If you find yourself running up against these issues frequently, you may have to resort to some pre-processing of the audio. This can be done with audio editing software or a Python package (such as SciPy) that can apply filters to the files. A detailed discussion of this is beyond the scope of this tutorial—check out Allen Downey's Think DSP book if you are interested. For now, just be aware that ambient noise in an audio file can cause problems and must be addressed in order to maximize the accuracy of speech recognition.

When working with noisy files, it can be helpful to see the actual API response. Most APIs return a JSON string 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`.

Python >>>

```python
>>> 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.

By now, you have a pretty good idea of the basics of the SpeechRecognition package. You've seen how to create an `AudioFile` instance from an audio file and use the `record()` method to capture data from the file. You learned how record segments of a file using the `offset` and `duration` keyword arguments of `record()`, and you experienced the detrimental effect noise can have on transcription accuracy.

Now for the fun part. Let's transition from transcribing static audio files to making your project interactive by accepting input from a microphone.

Working With Microphones

To access your microphone with SpeechRecognizer, you'll have to install the PyAudio package. Go ahead and close your current interpreter session, and let's do that.

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`:

Shell

```shell
$ 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`:

Shell

```
$ brew install portaudio
$ pip install pyaudio
```

Windows

On Windows, you can install PyAudio with `pip`:

Shell

```
$ pip install pyaudio
```

Testing the Installation

Once you've got PyAudio installed, you can test the installation from the console.

Shell

```
$ python -m speech_recognition
```

Make sure your default microphone is on and unmuted. If the installation worked, you should see something like this:

Shell

```
A moment of silence, please...
Set minimum energy threshold to 600.4452854381937
Say something!
```

Go ahead and play around with it a little bit by speaking into your microphone and seeing how well SpeechRecognition transcribes your speech.

Note: If you are on Ubuntu and get some funky output like 'ALSA lib … Unknown PCM', refer to this page for tips on suppressing these messages. This output comes from the ALSA package installed with Ubuntu—not SpeechRecognition or PyAudio. In all reality, these messages may indicate a problem with your ALSA configuration, but in my experience, they do not impact the functionality of your code. They are mostly a nuisance.

The `Microphone` Class

Open up another interpreter session and create an instance of the recognizer class.

Python >>>

```
>>> import speech_recognition as sr
>>> r = sr.Recognizer()
```

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.

Python >>>

```
>>> mic = sr.Microphone()
```

If your system has no default microphone (such as on a RaspberryPi), or you want to use a microphone other than the default, you will need to specify which one to use by supplying a device index. You can get a list of microphone names by calling the `list_microphone_names()` static method of the `Microphone` class.

Python >>>

```
>>> sr.Microphone.list_microphone_names()
['HDA Intel PCH: ALC272 Analog (hw:0,0)',
 'HDA Intel PCH: HDMI 0 (hw:0,3)',
 'sysdefault',
 'front',
 'surround40',
 'surround51',
 'surround71',
 'hdmi',
 'pulse',
 'dmix',
 'default']
```

Note that your output may differ from the above example.

The device index of the microphone is the index of its name in the list returned by `list_microphone_names()`. For example, given the above output, if you want to use the microphone called "front," which has index 3 in the list, you would create a microphone instance like this:

Python >>>

```
>>> # This is just an example; do not run
>>> mic = sr.Microphone(device_index=3)
```

For most projects, though, you'll probably want to use the default system microphone.

Using `listen()` to Capture Microphone Input

Now that you've got a `Microphone` instance ready to go, it's time to capture some 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.

Python >>>

```
>>> with mic as source:
...     audio = r.listen(source)
...
```

Once you execute the `with` block, try speaking "hello" into your microphone. Wait a moment for the interpreter prompt to display again. Once the ">>>" prompt returns, you're ready to recognize the speech.

Python >>>

```
>>> r.recognize_google(audio)
'hello'
```

If the prompt never returns, your microphone is most likely picking up too much ambient noise. You can interrupt the process with +ctrl+c++ to get your prompt back.

To handle ambient noise, you'll need to use the `adjust_for_ambient_noise()` method of the `Recognizer` class, just like you did when trying to make sense of the noisy audio file. Since input from a microphone is far less predictable than input from an audio file, it is a good idea to do this anytime you listen for microphone input.

Python >>>

```python
>>> with mic as source:
...     r.adjust_for_ambient_noise(source)
...     audio = r.listen(source)
...
```

After running the above code, wait a second for `adjust_for_ambient_noise()` to do its thing, then try speaking "hello" into the microphone. Again, you will have to wait a moment for the interpreter prompt to return before trying to recognize the speech.

Recall that `adjust_for_ambient_noise()` analyzes the audio source for one second. If this seems too long to you, feel free to adjust this with the `duration` keyword argument.

The SpeechRecognition documentation recommends using a duration no less than 0.5 seconds. In some cases, you may find that durations longer than the default of one second generate better results. The minimum value you need depends on the microphone's ambient environment. Unfortunately, this information is typically unknown during development. In my experience, the default duration of one second is adequate for most applications.

Handling Unrecognizable Speech

Try typing the previous code example in to the interpeter and making some unintelligible noises into the microphone. You should get something like this in response:

Python

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/david/real_python/speech_recognition_primer/venv/lib/python3.5/site-packages/speech_recognit
    if not isinstance(actual_result, dict) or len(actual_result.get("alternative", [])) == 0: raise Unknown
speech_recognition.UnknownValueError
```

Audio that cannot be matched to text by the API raises an `UnknownValueError` exception. You should always wrap calls to the API with `try` and `except` blocks to handle this exception.

Note: You may have to try harder than you expect to get the exception thrown. The API works very hard to transcribe any vocal sounds. Even short grunts were transcribed as words like "how" for me. Coughing, hand claps, and tongue clicks would consistently raise the exception.

Putting It All Together: A "Guess the Word" Game

Now that you've seen the basics of recognizing speech with the SpeechRecognition package let's put your newfound knowledge to use and write a small game that picks a random word from a list and gives the user three attempts to guess the word.

Here is the full script:

Python

```python
import random
import time

import speech_recognition as sr


def recognize_speech_from_mic(recognizer, microphone):
    """Transcribe speech from recorded from `microphone`.

    Returns a dictionary with three keys:
    "success": a boolean indicating whether or not the API request was
               successful
    "error":   `None` if no error occured, otherwise a string containing
               an error message if the API could not be reached or
               speech was unrecognizable
    "transcription": `None` if speech could not be transcribed,
               otherwise a string containing the transcribed text
    """
    # check that recognizer and microphone arguments are appropriate type
    if not isinstance(recognizer, sr.Recognizer):
        raise TypeError("`recognizer` must be `Recognizer` instance")

    if not isinstance(microphone, sr.Microphone):
        raise TypeError("`microphone` must be `Microphone` instance")

    # adjust the recognizer sensitivity to ambient noise and record audio
    # from the microphone
    with microphone as source:
        recognizer.adjust_for_ambient_noise(source)
        audio = recognizer.listen(source)

    # set up the response object
    response = {
        "success": True,
        "error": None,
        "transcription": None
    }

    # try recognizing the speech in the recording
    # if a RequestError or UnknownValueError exception is caught,
    #     update the response object accordingly
    try:
        response["transcription"] = recognizer.recognize_google(audio)
    except sr.RequestError:
        # API was unreachable or unresponsive
        response["success"] = False
        response["error"] = "API unavailable"
    except sr.UnknownValueError:
        # speech was unintelligible
        response["error"] = "Unable to recognize speech"

    return response


if __name__ == "__main__":
    # set the list of words, maxnumber of guesses, and prompt limit
    WORDS = ["apple", "banana", "grape", "orange", "mango", "lemon"]
    NUM_GUESSES = 3
    PROMPT_LIMIT = 5

    # create recognizer and mic instances
    recognizer = sr.Recognizer()
    microphone = sr.Microphone()

    # get a random word from the list
    word = random.choice(WORDS)

    # format the instructions string
    instructions = (
        "I'm thinking of one of these words:\n"
        "{words}\n"
        "You have {n} tries to guess which one.\n"
    ).format(words=', '.join(WORDS), n=NUM_GUESSES)

    # show instructions and wait 3 seconds before starting the game
    print(instructions)
    time.sleep(3)

    for i in range(NUM_GUESSES):
        # get the guess from the user
        # if a transcription is returned, break out of the loop and
        #     continue
        # if no transcription returned and API request failed, break
        #     loop and continue
        # if API request succeeded but no transcription was returned,
        #     re-prompt the user to say their guess again. Do this up
        #     to PROMPT_LIMIT times
        for j in range(PROMPT_LIMIT):
            print('Guess {}. Speak!'.format(i+1))
            guess = recognize_speech_from_mic(recognizer, microphone)
            if guess["transcription"]:
                break
            if not guess["success"]:
                break
            print("I didn't catch that. What did you say?\n")

        # if there was an error, stop the game
        if guess["error"]:
            print("ERROR: {}".format(guess["error"]))
            break

        # show the user the transcription
        print("You said: {}".format(guess["transcription"]))

        # determine if guess is correct and if any attempts remain
        guess_is_correct = guess["transcription"].lower() == word.lower()
        user_has_more_attempts = i < NUM_GUESSES - 1

        # determine if the user has won the game
        # if not, repeat the loop if user has more attempts
        # if no attempts left, the user loses the game
        if guess_is_correct:
            print("Correct! You win!".format(word))
            break
        elif user_has_more_attempts:
            print("Incorrect. Try again.\n")
        else:
            print("Sorry, you lose!\nI was thinking of '{}'.".format(word))
            break
```

Let’s break that down a little bit.

The `recognize_speech_from_mic()` function takes a `Recognizer` and `Microphone` instance as arguments and returns a dictionary with three keys. The first key, `"success"`, is a boolean that indicates whether or not the API request was successful. The second key, `"error"`, is either `None` or an error message indicating that the API is unavailable or the speech was unintelligible. Finally, the `"transcription"` key contains the transcription of the audio recorded by the microphone.

The function first checks that the `recognizer` and `microphone` arguments are of the correct type, and raises a `TypeError` if either is invalid:

```python
if not isinstance(recognizer, sr.Recognizer):
    raise TypeError('`recognizer` must be `Recognizer` instance')

if not isinstance(microphone, sr.Microphone):
    raise TypeError('`microphone` must be a `Microphone` instance')
```

The `listen()` method is then used to record microphone input:

```python
with microphone as source:
    recognizer.adjust_for_ambient_noise(source)
    audio = recognizer.listen(source)
```

The `adjust_for_ambient_noise()` method is used to calibrate the recognizer for changing noise conditions each time the `recognize_speech_from_mic()` function is called.

Next, `recognize_google()` is called to transcribe any speech in the recording. A `try...except` block is used to catch the `RequestError` and `UnknownValueError` exceptions and handle them accordingly. The success of the API request, any error messages, and the transcribed speech are stored in the `success`, `error` and `transcription` keys of the `response` dictionary, which is returned by the `recognize_speech_from_mic()` function.

```python
response = {
    "success": True,
    "error": None,
    "transcription": None
}

try:
    response["transcription"] = recognizer.recognize_google(audio)
except sr.RequestError:
    # API was unreachable or unresponsive
    response["success"] = False
    response["error"] = "API unavailable"
except sr.UnknownValueError:
    # speech was unintelligible
    response["error"] = "Unable to recognize speech"

return response
```

You can test the `recognize_speech_from_mic()` function by saving the above script to a file called "guessing_game.py" and running the following in an interpreter session:

```python
>>> import speech_recognition as sr
>>> from guessing_game import recognize_speech_from_mic
>>> r = sr.Recognizer()
>>> m = sr.Microphone()
>>> recognize_speech_from_mic(r, m)
{'success': True, 'error': None, 'transcription': 'hello'}
>>> # Your output will vary depending on what you say
```

The game itself is pretty simple. First, a list of words, a maximum number of allowed guesses and a prompt limit are declared:

Python

```python
WORDS = ['apple', 'banana', 'grape', 'orange', 'mango', 'lemon']
NUM_GUESSES = 3
PROMPT_LIMIT = 5
```

Next, a `Recognizer` and `Microphone` instance is created and a random word is chosen from `WORDS`:

Python

```python
recognizer = sr.Recognizer()
microphone = sr.Microphone()
word = random.choice(WORDS)
```

After printing some instructions and waiting for 3 three seconds, a `for` loop is used to manage each user attempt at guessing the chosen word. The first thing inside the `for` loop is another `for` loop that prompts the user at most `PROMPT_LIMIT` times for a guess, attempting to recognize the input each time with the `recognize_speech_from_mic()` function and storing the dictionary returned to the local variable `guess`.

If the `"transcription"` key of `guess` is not `None`, then the user's speech was transcribed and the inner loop is terminated with `break`. If the speech was not transcribed and the `"success"` key is set to `False`, then an API error occurred and the loop is again terminated with `break`. Otherwise, the API request was successful but the speech was unrecognizable. The user is warned and the `for` loop repeats, giving the user another chance at the current attempt.

Python

```python
for j in range(PROMPT_LIMIT):
    print('Guess {}. Speak!'.format(i+1))
    guess = recognize_speech_from_mic(recognizer, microphone)
    if guess["transcription"]:
        break
    if not guess["success"]:
        break
    print("I didn't catch that. What did you say?\n")
```

Once the inner `for` loop terminates, the `guess` dictionary is checked for errors. If any occurred, the error message is displayed and the outer `for` loop is terminated with `break`, which will end the program execution.

Python

```python
if guess['error']:
    print("ERROR: {}".format(guess["error"]))
    break
```

If there weren't any errors, the transcription is compared to the randomly selected word. The `lower()` method for string objects is used to ensure better matching of the guess to the chosen word. The API may return speech matched to the word "apple" as "Apple" *or* "apple," and either response should count as a correct answer.

If the guess was correct, the user wins and the game is terminated. If the user was incorrect and has any remaining attempts, the outer `for` loop repeats and a new guess is retrieved. Otherwise, the user loses the game.

Python

```
guess_is_correct = guess["transcription"].lower() == word.lower()
user_has_more_attempts = i < NUM_GUESSES - 1

if guess_is_correct:
    print('Correct! You win!'.format(word))
    break
elif user_has_more_attempts:
    print('Incorrect. Try again.\n')
else:
    print("Sorry, you lose!\nI was thinking of '{}'.".format(word))
    break
```

When run, the output will look something like this:

Shell

```
I'm thinking of one of these words:
apple, banana, grape, orange, mango, lemon
You have 3 tries to guess which one.

Guess 1. Speak!
You said: banana
Incorrect. Try again.

Guess 2. Speak!
You said: lemon
Incorrect. Try again.

Guess 3. Speak!
You said: Orange
Correct! You win!
```

Recap and Additional Resources

In this tutorial, you've seen how to install the SpeechRecognition package and use its `Recognizer` class to easily recognize speech from both a file—using `record()`—and microphone input—using `listen()`. You also saw how to process segments of an audio file using the `offset` and `duration` keyword arguments of the `record()` method.

You've seen the effect noise can have on the accuracy of transcriptions, and have learned how to adjust a `Recognizer` instance's sensitivity to ambient noise with `adjust_for_ambient_noise()`. You have also learned which exceptions a `Recognizer` instance may throw—`RequestError` for bad API requests and `UnkownValueError` for unintelligible speech—and how to handle these with `try...except` blocks.

Speech recognition is a deep subject, and what you have learned here barely scratches the surface. If you're interested in learning more, here are some additional resources.

Free Bonus: Click here to download a Python speech recognition sample project with full source code that you can use as a basis for your own speech recognition apps.

For more information on the SpeechRecognition package:

- Library reference
- Examples
- Troubleshooting page

A few interesting internet resources:

- Behind the Mic: The Science of Talking with Computers . A short film about speech processing by Google.
- A Historical Perspective of Speech Recognition by Huang, Baker and Reddy. Communications of the ACM (2014). This article provides an in-depth and scholarly look at the evolution of speech recognition technology.
- The Past, Present and Future of Speech Recognition Technology by Clark Boyd at The Startup. This blog post presents an overview of speech recognition technology, with some thoughts about the future.

Some good books about speech recognition:

- The Voice in the Machine: Building Computers That Understand Speech , Pieraccini, MIT Press (2012). An accessible general-audience book covering the history of, as well as modern advances in, speech processing.
- Fundamentals of Speech Recognition , Rabiner and Juang, Prentice Hall (1993). Rabiner, a researcher at Bell Labs, was instrumental in designing some of the first commercially viable speech recognizers. This book is now over 20 years old, but a lot of the fundamentals remain the same.
- Automatic Speech Recognition: A Deep Learning Approach , Yu and Deng, Springer (2014). Yu and Deng are researchers at Microsoft and both very active in the field of speech processing. This book covers a lot of modern approaches and cutting-edge research but is not for the mathematically faint-of-heart.

Appendix: Recognizing Speech in Languages Other Than English

Throughout this tutorial, we've been recognizing speech in English, which is the default language for each `recognize_*()` method of the SpeechRecognition package. However, it is absolutely possible to recognize speech in other languages, and is quite simple to accomplish.

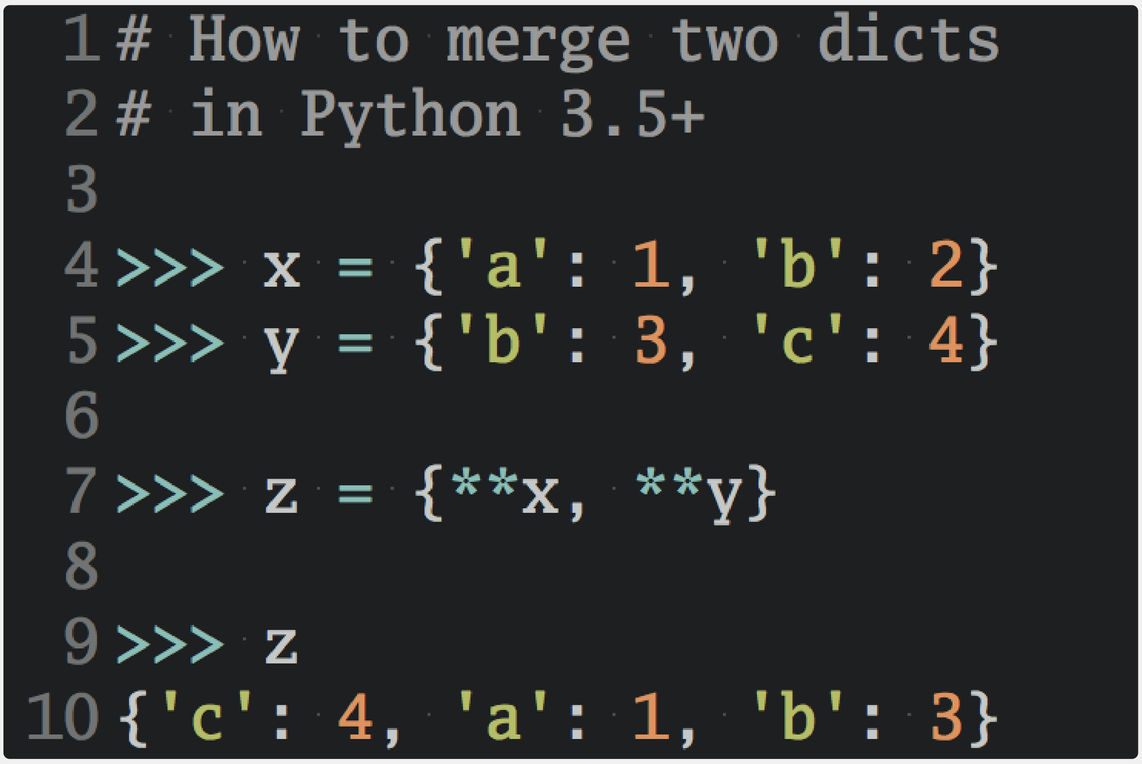
To recognize speech in a different language, set the `language` keyword argument of the `recognize_*()` method to a string corresponding to the desired language. Most of the methods accept a BCP-47 language tag, such as `'en-US'` for American English, or `'fr-FR'` for French. For example, the following recognizes French speech in an audio file:

Python
```python
import speech_recognition as sr

r = sr.Recognizer()

with sr.AudioFile('path/to/audiofile.wav') as source:
    audio = r.record(source)

r.recognize_google(audio, language='fr-FR')
```

Only the following methods accept a `language` keyword argument:

- `recognize_bing()`
- `recognize_google()`
- `recognize_google_cloud()`
- `recognize_ibm()`
- `recognize_sphinx()`

To find out which language tags are supported by the API you are using, you'll have to consult the corresponding documentation. A list of tags accepted by `recognize_google()` can be found in this Stack Overflow answer.



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

About **David Amos**

David is a mathematician by training, a data scientist/Python developer by profession, and a coffee junkie by choice.

» More about David

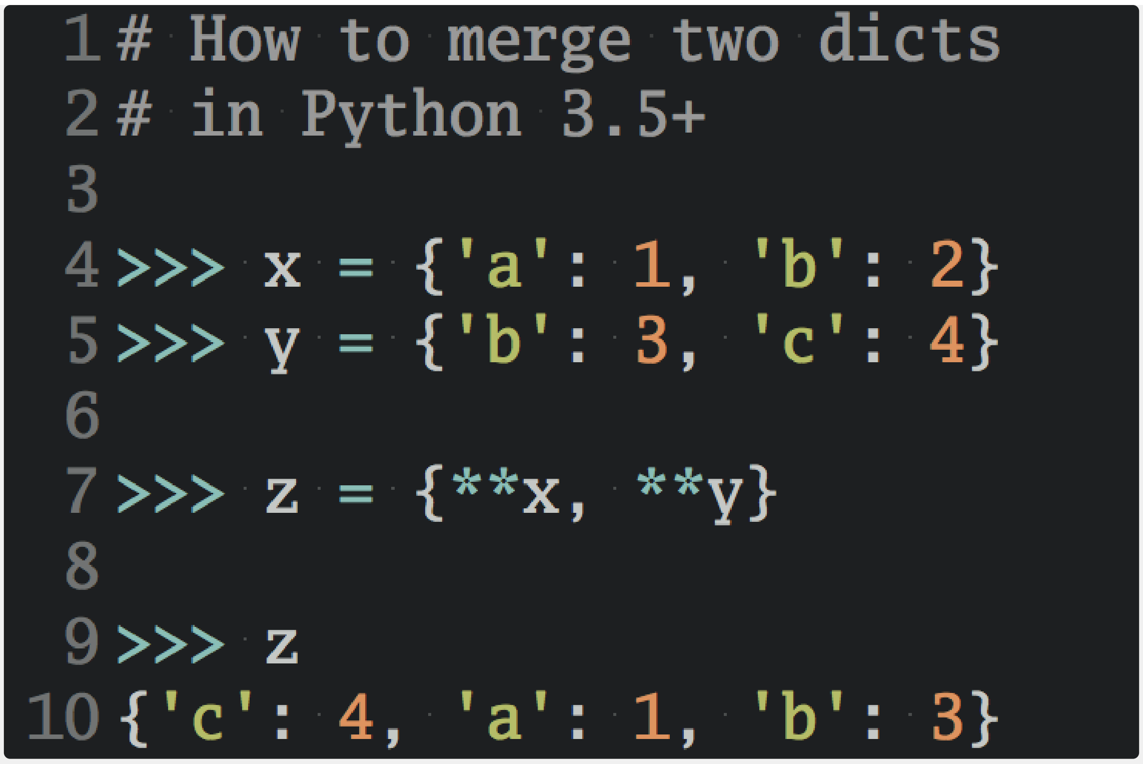
Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

What Do You Think?

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won’t make the cut here.

Keep Learning

Related Tutorial Categories: advanced data-science machine-learning



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email…

Get Python Tricks »

No spam. Unsubscribe any time.

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end intermediate machine-learning python testing tools web-dev web-scraping

Table of Contents