Python

24th Feb 2022 | 6 minutes read

How to Visualize Sound in Python

Luke Hande

Python Technical

There's a lot of music and voice data out there. There are also interesting applications to go with them. We show you how to visualize sound in Python.

The analysis of audio data has become ever more relevant in recent times. Popular virtual assistant products have been released by major technology companies, and these products are becoming more common in smartphones and homes around the world. They are largely developed on top of models that analyze voice data and extract information from it.

There is a large range of applications using audio data analysis, and this is a rich topic to explore. In this article, we're going to focus on a fundamental part of the audio data analysis process – plotting the waveform and frequency spectrum of the audio file.

This article is aimed at people with a bit more background in data analysis. If you're a beginner and are looking for some material to get up to speed in data science, take a look at THIS TRACK.

Opening a WAV File

Audio files come in a variety of formats. You're probably familiar with MP3, which uses lossy compression to store data. Formats such as FLAC use lossless compression, which allows the original data to be perfectly reconstructed from the compressed data. Our audio file is in the WAV (Waveform Audio File) format, which is uncompressed. The file sizes can get large as a consequence.

The sound file we'll look at is an upbeat jingle that starts with a piano. Other sounds like bells and clapping come in throughout the jingle, with a strumming guitar part at two points in the track. It's worth mentioning these features in the audio recording because we can identify some of these later when we plot the waveform and the frequency spectrum.

To open our WAV file, we use the WAVE module in Python, which can be imported and called as follows:

Code

```
>>> import wave
>>> wav_obj = wave.open('file.wav', 'rb')
```

The ' `rb` ' mode returns a `wave_read` object. Using ' `wb` ' to open the file returns a `wave_write` object, which has different methods from the former object. You can also use a `with` statement to open the file as we demonstrate HERE. If you're interested in learning more about how to programmatically handle large numbers of files, take a look at THIS ARTICLE.

A sound wave is a continuous quantity that needs to be sampled at some time interval to digitize it. The sampling rate quantifies how many samples of the sound are taken every second. We can access this information using the following method:

Code

```
>>> sample_freq = wav_obj.getframerate()
>>> sample_freq
44100
```

The sample frequency quantifies the number of samples per second. In this case, it is 44,100 times per second, which corresponds to CD quality. The number of individual frames, or samples, is given by:

Code

```
>>> n_samples = wav_obj.getnframes()
>>> n_samples
5384326
```

We can now calculate how long our audio file is in seconds:

Code

```
>>> t_audio = n_samples/sample_freq
>>> t_audio
122.09356009070295
```

The audio file is recorded in stereo, that is, in two independent audio channels. This creates the impression of the sound coming from two different directions. We can check the number of channels as follows:

Code

```
>>> n_channels = wav_obj.getnchannels()
>>> n_channels
2
```

The next step is to get the values of the signal, that is, the amplitude of the wave at that point in time. To do this, we can use the `readframes()` method, which takes one argument, n, defining the number of frames to read:

Code

```
>>> signal_wave = wav_obj.readframes(n_samples)
```

This method returns a bytes object. Check for yourself by using the `type()` built-in function on the `signal_wave` object. To get signal values from this, we have to turn to numpy:

Code

```
>>> import numpy as np
>>> signal_array = np.frombuffer(signal_wave, dtype=np.int16)
```

This returns all data from both channels as a 1-dimensional array. If you check the shape of `signal_array`, you notice it has 10,768,652 elements, which is exactly `n_samples * n_channels`. To split the data into individual channels, we can use a clever little array slice trick:

Code

```
>>> l_channel = signal_array[0::2]
>>> r_channel = signal_array[1::2]
```

Now, our left and right channels are separated, both containing 5,384,326 integers representing the amplitude of the signal.

Next, we show some examples of how to plot the signal values. We have our data stored in arrays here, but for many data science applications, pandas is very useful. Check out THIS ARTICLE about visualizing data stored in a `DataFrame`.

Plotting the Signal Amplitude

Before we get to plotting signal values, we need to calculate the time at which each sample is taken. This is simply the total length of the track in seconds, divided by the number of samples. We can use `linspace()` from `numpy` to create an array of timestamps:

Code

```
>>> times = np.linspace(0, n_samples/sample_freq, num=n_samples)
```

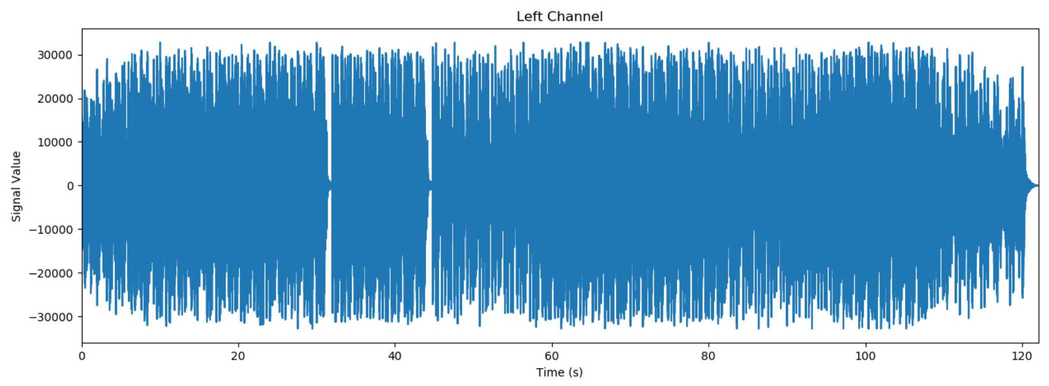
For plotting, we're going to use the `pyplot` class from `matplotlib`. If you need some background material on plotting in Python, we have some articles. Here's PART 1 and PART 2 of an introduction to `matplotlib`.

For simplicity, we only plot the signal from one channel. Let's set up the figure, and plot a time series as follows:

Code

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(15, 5))
>>> plt.plot(times, l_channel)
>>> plt.title('Left Channel')
>>> plt.ylabel('Signal Value')
>>> plt.xlabel('Time (s)')
>>> plt.xlim(0, t_audio)
>>> plt.show()
```

This opens the following figure in a new window:



We see the amplitude build up in the first 6 seconds, at which point the bells and clapping effects start. There are two brief pauses in the jingle at 31.5 and 44.5 seconds, which are evident in the signal values. After the second pause, the main instrument alternates between a guitar and a piano, which is roughly seen in the signal, where the guitar part has lower amplitudes. Then, there's a lower-amplitude outro at the end of the track.

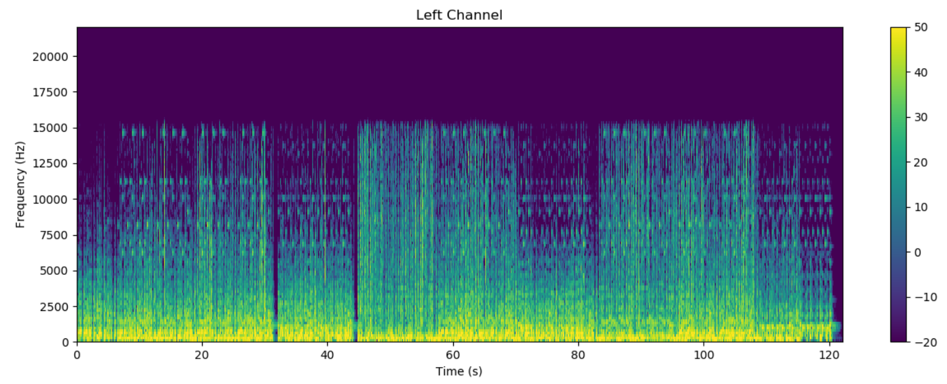
Plotting the Frequency Spectrum

Now, let’s take a look at the frequency spectrum, also known as a spectrogram. This is a visual representation of the signal strength at different frequencies, showing us which frequencies dominate the recording as a function of time:

Code

```
>>> plt.figure(figsize=(15, 5))
>>> plt.specgram(l_channel, Fs=sample_freq, vmin=-20, vmax=50)
>>> plt.title('Left Channel')
>>> plt.ylabel('Frequency (Hz)')
>>> plt.xlabel('Time (s)')
>>> plt.xlim(0, t_audio)
>>> plt.colorbar()
>>> plt.show()
```

The following plot opens in a new window:



In the plotting code above, vmin and vmax are chosen to bring out the lower frequencies that dominate this recording. Indeed, the dominant frequencies for the whole track are lower than 2.5 kHz. You see the effect of different instruments and sound effects, particularly in the frequency range of about 10 kHz to 15 kHz. Each instrument and sound effect has its own signature in the frequency spectrum.

Where to Go From Here

Plotting the waveform and frequency spectrum with Python forms a foundation for a deeper analysis of the sound data. Perhaps you can further quantify the frequencies of each part of the recording. What is the average frequency of the guitar part compared to the piano part? And here, we've only looked at one channel. Another extension of the material here is to plot both channels and see how they compare. Try plotting the difference between the channels, and you see some new and interesting features pop out of the waveform and the frequency spectrum.

Tags: Python Technical

You may also like

How to Learn Python Faster

Check out how to learn Python faster! Stop wasting time on other slow and ineffective methods.

Read more

How to Write to File in Python

Discover how to write to a file in Python using the write() and writelines() methods and the pathlib and csv modules.

Read more

Subscribe to our newsletter

Join our monthly newsletter to be notified about the latest posts.

@ Email address

Subscribe

Quick links

Pricing

Blog

Vertabelo.com

Assistance

Need assistance? Drop us a line at **contact@learnpython.com**

Write to us

Follow us

Terms of service
Privacy policy

Imprint