Introduction to Sounds and Volume

Working with sounds and samples

To work with sounds in Python, we are going to use the wavfile module of the SciPy.io package. This will allow us to read and write `wav` files in Python. When the sound file is read, the data gets stored in a NumPy array. We will use the IPython.display module for displaying an audio control object on the screen, and the matplotlib.pyplot module will be used to graph the sound waves.

The following example shows the packages and modules we would need to import, along with the commands to open and read a `wav` file stored in our Google drive. The `read` function returns the sample rate of the sound and an array containing all of the sample values.

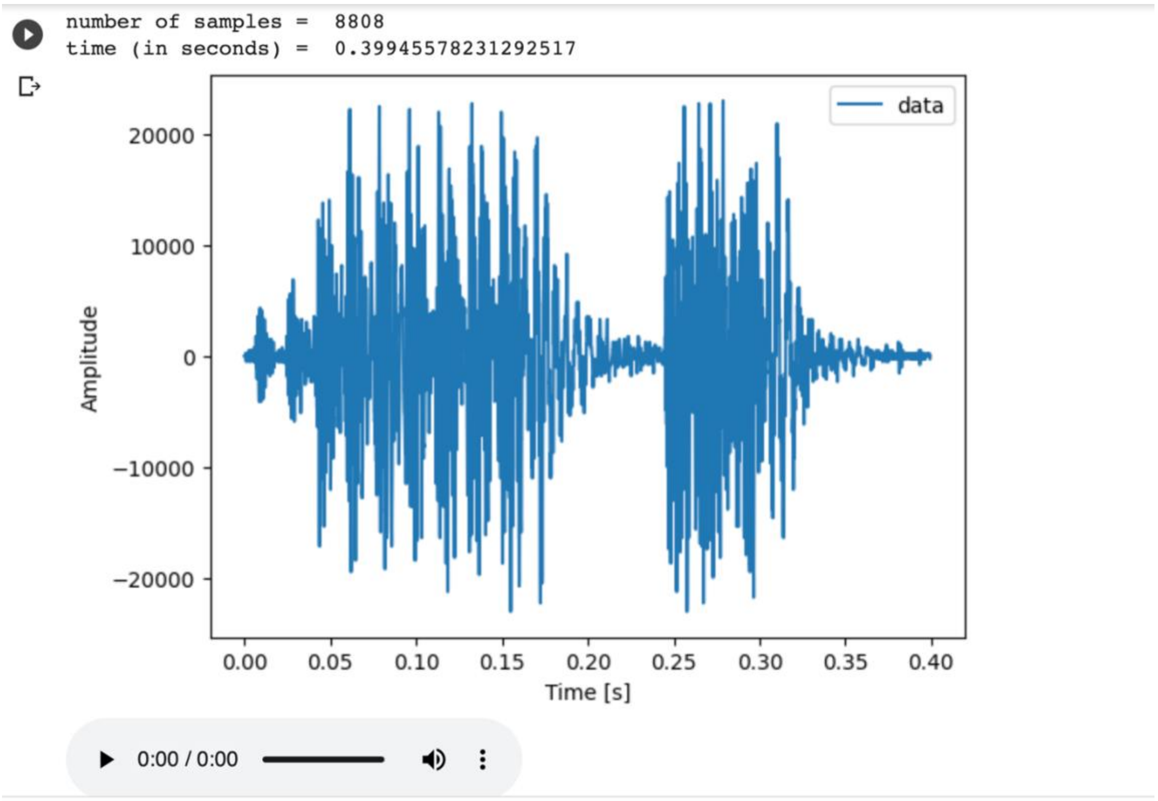
```
from IPython.display import Audio, display
from scipy.io.wavfile import read, write
import numpy as np
import matplotlib.pyplot as plt

soundFile = "/drive/MyDrive/Sounds/croak.wav"
samplerate, data = read(soundFile)
```

Once the sound file has been read, we can then determine the number of samples in the sound and the length (in seconds) of the sound. We use the Python `len` function to determine the number of elements (*i.e.,* the number of samples) in the data array. The number of seconds in the sound can be computed by dividing the number of samples by the sample rate. We can also graph the sound wave and display an audio control object to play the sound.

```
#calculate the number of samples and the length (time) of the sound
numsamples = len(data)
print("number of samples = ",numsamples)
seconds = numsamples / samplerate
print("time (in seconds) = ", seconds)

# graph the sound wave
time = np.linspace(0., seconds, data.shape[0])
plt.plot(time, data, label="data")
plt.legend()
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")
plt.show()
```

```
# display a control object to play the sound
display(Audio(soundFile, autoplay=True))
```

The output of this code would look like the following:



```
number of samples =  8808
time (in seconds) =  0.39945578231292517
```

Working with sample values

The data array contains the amplitude values for each sample. The sound in the example above has 8808 samples, and they can each be accessed by a unique value, called an *index*. The indices start at 0 and go up to, but not including the number of samples. In the example above, the samples would be indexed from 0 to 8807. To get the value of the first sample, we would use `data[0]`. To get the value of the last sample, we would use `data[8807]`. If we would like to change the value of the sample, we can use an assignment statement, assigning the sample a value between -32768 and 32767, as in `data[105] = 12672`. This would give the sample at index 105 the value 12672.

Volume

As you may recall from science class, the amplitude of a sound is the main factor in the volume. If the amplitude of a sound increases or decreases, the volume of a sound increases and decreases accordingly. To change the volume of a sound, we need to change the values of the amplitude that are stored in each sample.

For example, suppose we want to increase the volume in a sound. This means the amplitude of the sound will need to be increased. We can do this in code with the following function:

Example: Increase Volume

```
def increaseVolume(soundData):
  # Make an array of all zeros, same length as soundData
  new_data = np.zeros(len(soundData), dtype=np.int16)

  # loop through sound data and double the sample values
  for i in range(len(soundData)):
    # set the values in the new_data array
    new_data[i] = soundData[i] * 2

  # return the data for the new sound
  return new_data
```

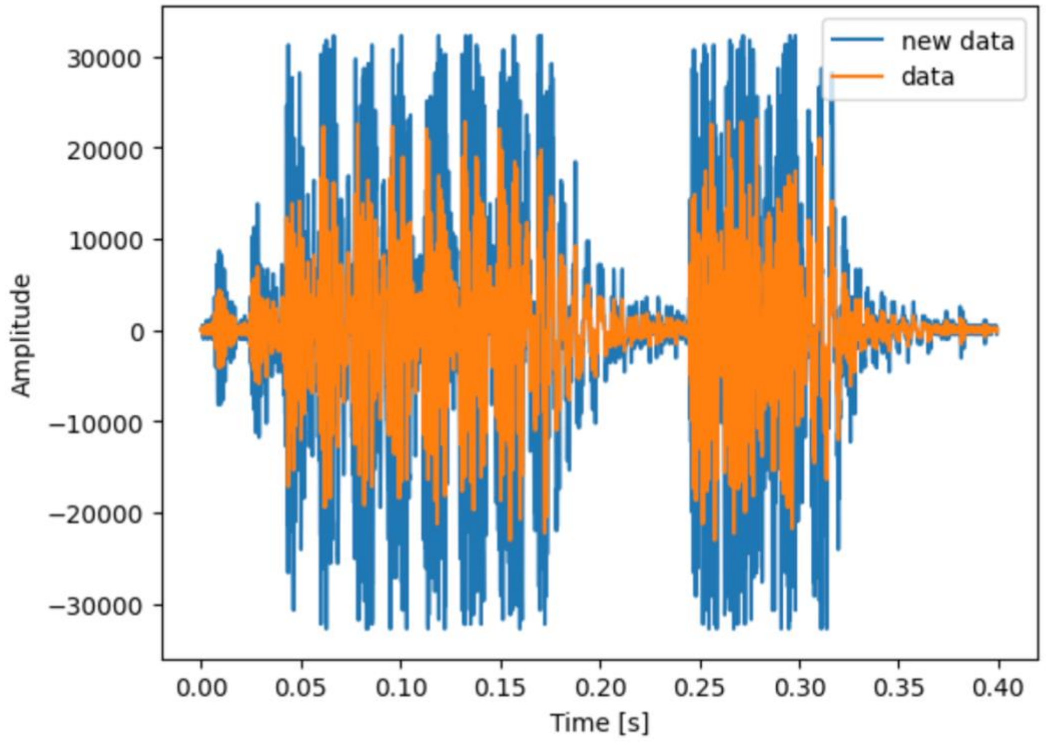
Let's look at what is going on in this function. We pass an array of sample values from a sound into the function, and then the very first thing we do is create a new array of zeros to hold the sample values for the new sound we will be making. Doing this ensures that we do not modify the original sound. We then loop through all of the samples in the new sound, getting the sample value of each, and then setting the corresponding value of the new array to be twice as large.

To test this function, we would have code like the following:

```
soundFile = "/drive/MyDrive/Sounds/croak.wav"
samplerate, data = read(soundFile)
seconds = len(data)/samplerate

# call the function
newSoundData = increaseVolume(data)

# save the new data to a sound file
write("/drive/MyDrive/Sounds/louder_croak.wav", samplerate,
newSoundData)
```

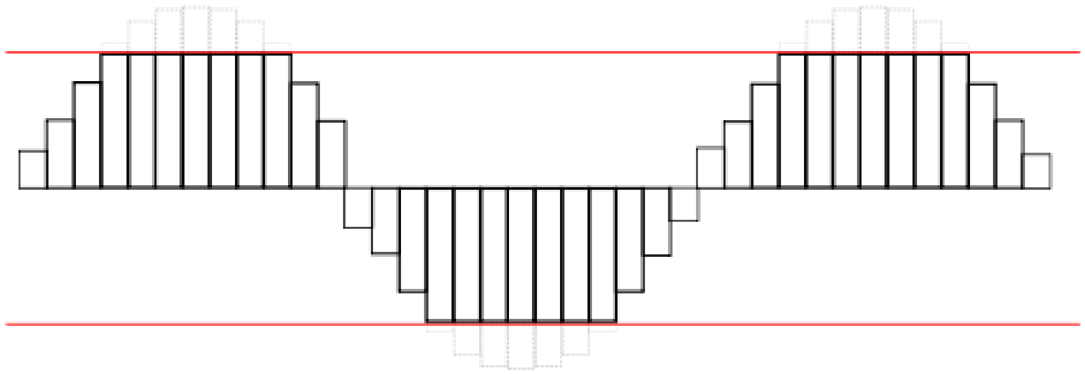
```
# graph the sound waves
time = np.linspace(0., seconds, data.shape[0])
plt.plot(time, newSoundData, label="new data")
plt.plot(time, data, label="data")
plt.legend()
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")
plt.show()

# output two audio controls, one for each sound
display(Audio("/drive/MyDrive/Sounds/croak.wav", autoplay=True))
display(Audio("/drive/MyDrive/Sounds/louder_croak.wav",
autoplay=True))
```

How do we know this function worked? When we play the original sound, and then the new sound, we can probably hear that the new sound is louder. We also plot both sounds and look at corresponding values on the graphs. We should find that the values in the new sound are double those in the original. (See graph below.)



What happened with the sample value at index 6145? The original sample value was 19456, but when this was multiplied by 2, it should have been 38912.

Remember, because we are using 16-bit two's complement representation for the storage of sample values, the values must range between -32768 and 32767. So this new sample value has been capped at 32767. This is what's known as *clipping*. The amplitude of the sound gets stopped at the maximum capacity. When this happened, the sine wave that represents the sound looks squared-off at the peaks, similar to:



You can often hear when a sound is clipped. It will sound like the audio is starting to 'break up', which is light distortion. It may be an unpleasant sound to your ears.

Now suppose we want to make the volume of a sound to be as loud as possible. This is called *normalizing* the sound. In order to do this, we need to find out what the largest (absolute) value of a sample in the sound is. Once we know the largest value, we figure out what factor to multiply our samples by. We know the largest value a sample can have is 32767. If we divide this by our largest sample value, we get the factor we should use:

(largest sample value) * factor = 32767.

To find the largest absolute value of the samples, we will define a variable, say, `largest`, and assign it to be 0. (Using absolute values, every value will be greater than or equal to 0). Then we will check all the sample values. If we find a sample value with absolute value greater than `largest`, we will replace `largest` with that new value. We will keep checking the sample values, comparing to the new value of `largest`, until we have compared all sample values, and the very largest value is stored in the variable `largest`. Our function to normalize a sound looks like the following:

```
# This function makes a sound as loud as possible
# without clipping
def normalize(soundData):
  # find the largest sample value
  largest = 0
  for index in range(len(soundData)):
    if abs(soundData[index]) > largest:
      largest = abs(soundData[index])

  # compute the multiplication factor
  factor = 32767.0 / largest

  new_data = changeVolume(soundData, factor)
  return new_data
```

We should now take some time to experiment with changing volume, by working through the next Activity.

Activity: Experimenting with Volume