Lecture 12: Multimodal data processing

- Working with audio, image, and video data

↓ Ask your questions in the comment section! ↓

Learning objectives

By the end of this lecture, you should:

- Understand the relevance of multimodal data in computational linguistics
- Understand how digital audio, images, and videos can be represented
- Be able to inspect and manipulate audio and image data using NumPy
- Be aware of some commonly used Python libraries for audio, image, and video processing

Why should you care about multimodal data?

Language is inherently multimodal:

- Audio: spoken language, prosody, ...
- Images: handwriting, facial expressions, pictograms, ...
- Video: sign language, body language, ...
- Cognition data: eye-tracking, EEG, MRI, ...

Written language is not the most important modality of language, it's just the easiest to work with.

Modalities other than text in natural language processing

Modality	as input	as output
Audio	speech recognition, speech translation, subtitling	text-to-speech
Image	optical character recognition, image captioning	image generation
Video	sign language recognition, sign language translation, scene captioning, audio description	sign language translation

Multimodal research at the Department of Computational Linguistics

- Language, Technology and Accessibility (Prof. Sarah Ebling): sign language, audio description, pictographs, text simplification with images
- Phonetics (Prof. Volker Dellwo): speech processing, speaker recognition
- Digital Linguistics (Prof. Lena Jäger): eye-tracking
- Computational Neuroscience of Speech & Hearing (Prof. Nathalie Giroud): EEG, MRI

In [1]:

```
import numpy as np
import scipy
import matplotlib.pyplot as plt
import seaborn as sns
from IPython.display import Audio, Image, Video
```

Audio data

In [2]:
```
Audio(filename="tongue-twister.wav")
```

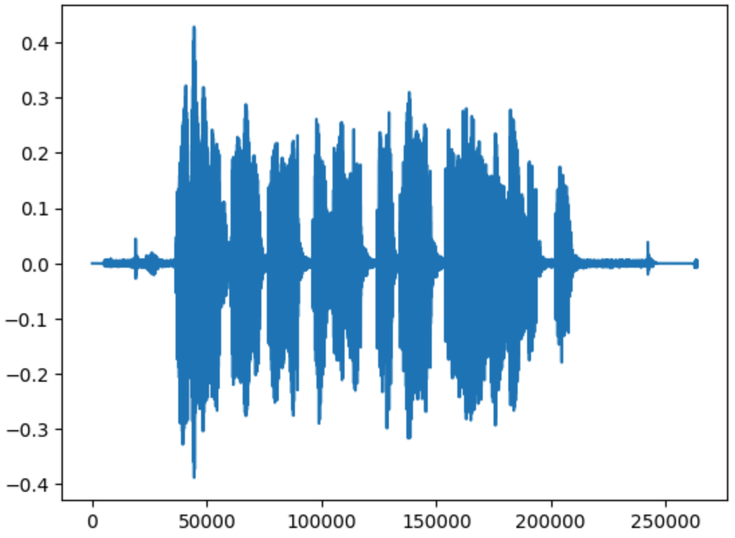
Out[2]:

In its most basic form, an audio file is a sequence of `float`s:

In [3]:
```
import soundfile as sf

audio, sampling_rate = sf.read('tongue-twister.wav')
print(f"{len(audio)} samples at {sampling_rate} samples per second")
plt.plot(audio)
```

```
263808 samples at 44100 samples per second
```

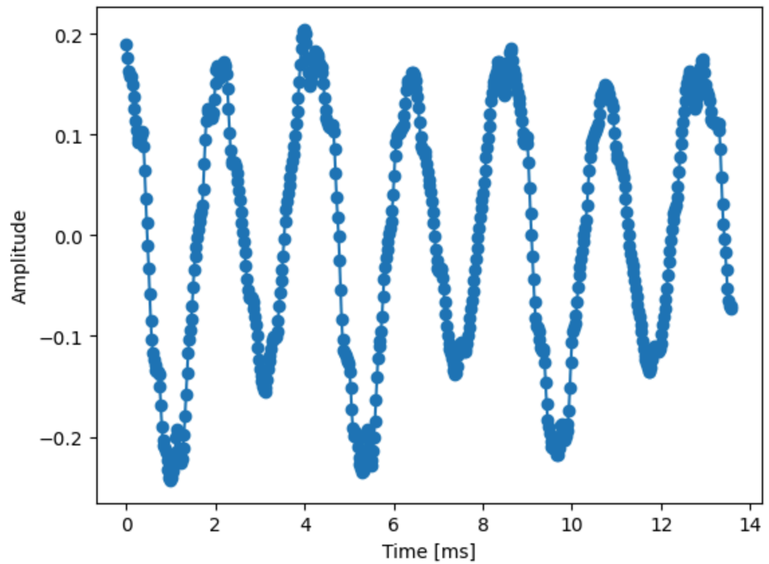
Out[3]:
```
[<matplotlib.lines.Line2D at 0x7f4b29cafa10>]
```



Waveform

In [4]:
```
waveform = audio[50000:50600]
time = np.arange(len(waveform)) / sampling_rate * 1000
plt.plot(time, waveform, marker='o')
plt.xlabel('Time [ms]')
plt.ylabel('Amplitude')
```

Out[4]:
```
Text(0, 0.5, 'Amplitude')
```



- **Sampling rate** [Hz]: number of samples per second; "resolution" of the signal
- **Frequency** [Hz]: number of oscillations per second; perceived as pitch
- **Amplitude** [no unit]: perceived as loudness

- **Sampling rate** [Hz]: number of samples per second; "resolution" of the signal
- **Frequency** [Hz]: number of oscillations per second; perceived as pitch
- **Amplitude** [no unit]: perceived as loudness

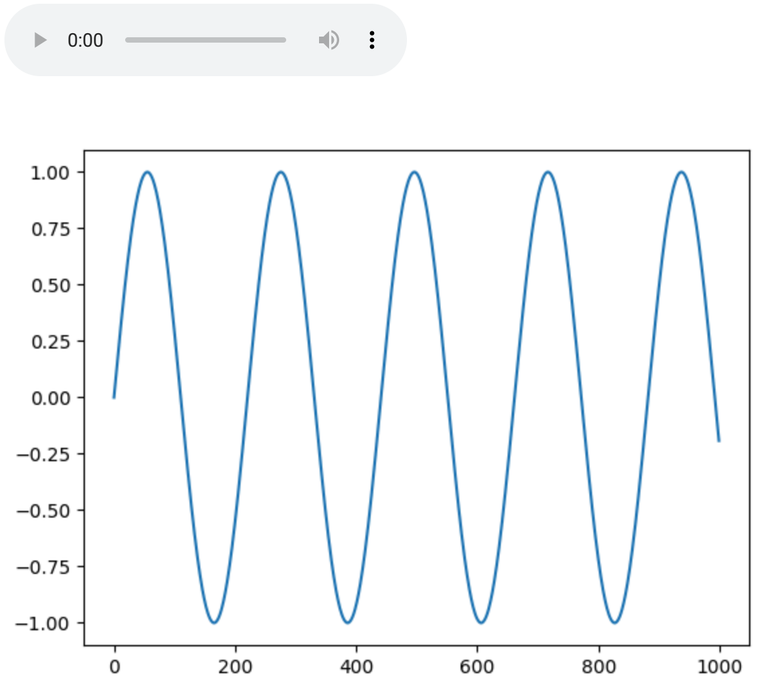
Typical fundamental frequencies of the human voice:

- Adult male: 90-155 Hz
- Adult female: 165-255 Hz

Sine waves are pure tones

In [5]:
```
def sinewave(frequency, amplitude=1, length=1, rate=44100):
    x = np.linspace(0, 2 * np.pi * frequency * length, round(length * rate))
    return np.sin(x) * amplitude

wave = sinewave(200)
plt.plot(wave[:1000])
Audio(wave, rate=44100)
```

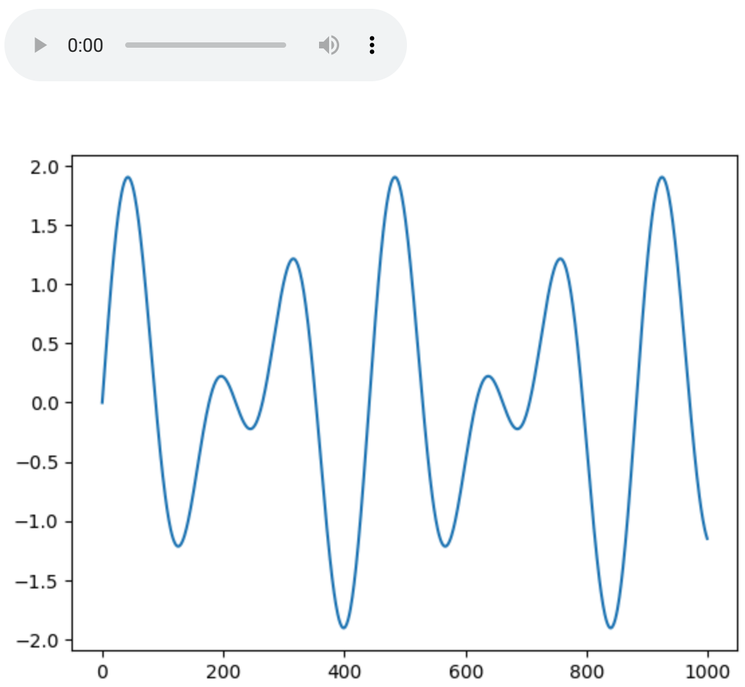
Out[5]:



In [6]:

```
wave = sinewave(200) + sinewave(300)
plt.plot(wave[:1000])
Audio(wave, rate=44100)
```

Out[6]:

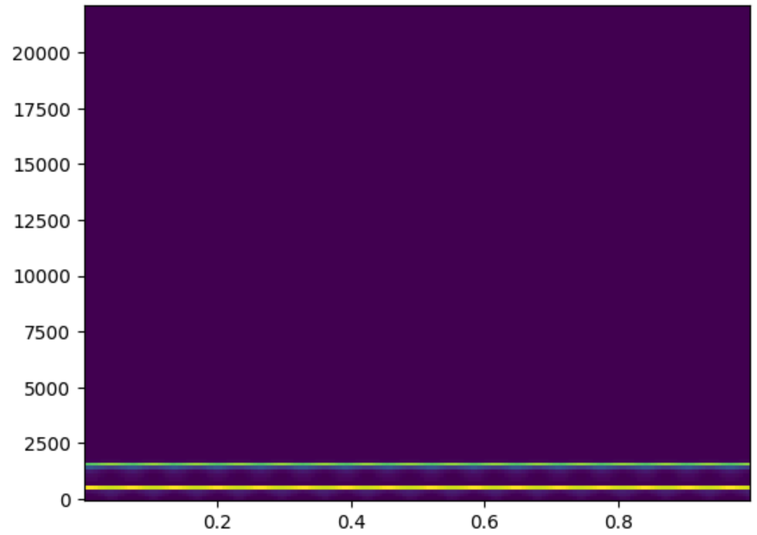


Spectrogram

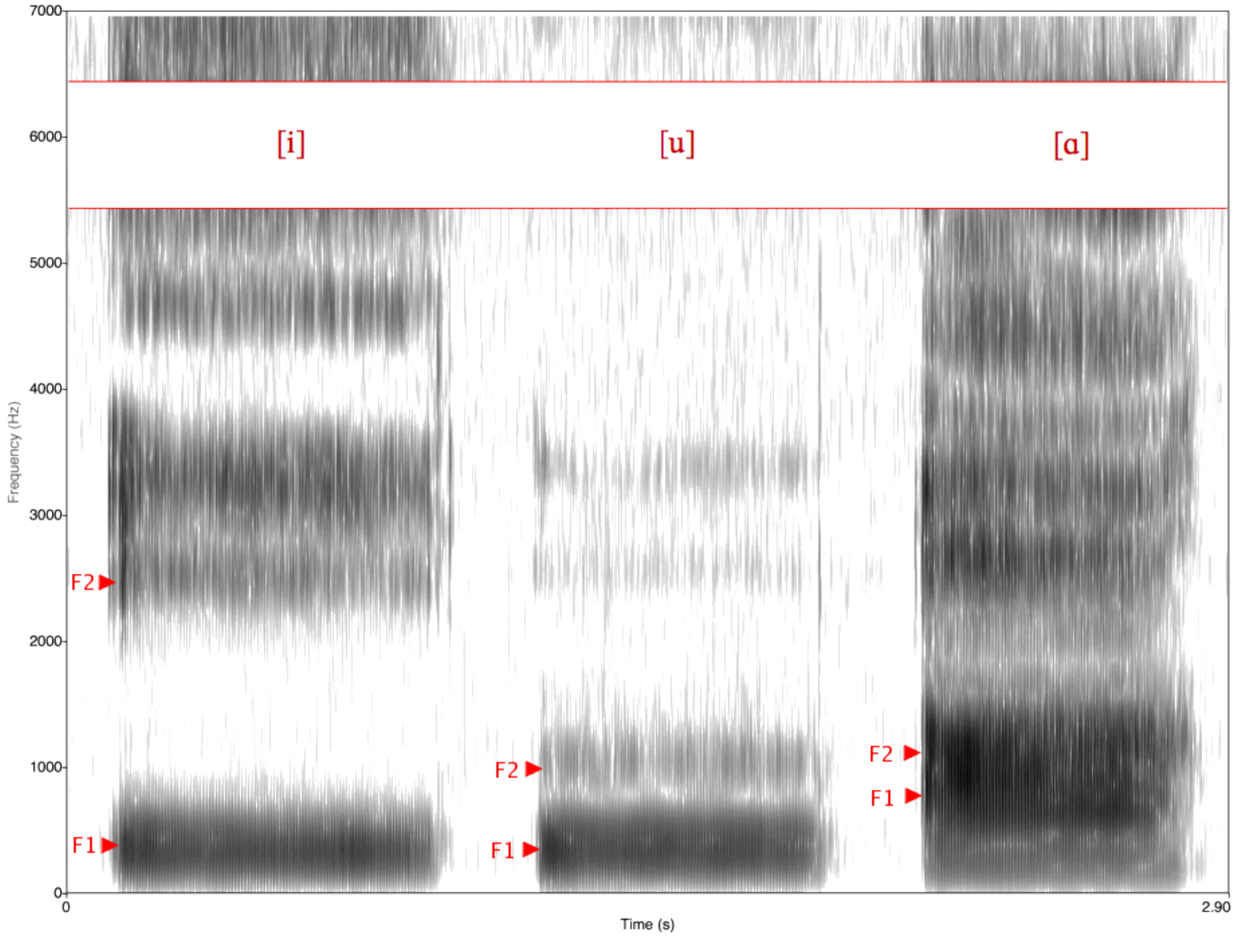
- Every periodic signal can be decomposed into a sum of sine waves.
- The combination of frequencies that make up a sound determines how it sounds.
- A **spectrogram** tells us what frequencies are present in the signal at each point in time.

In [7]:
```
wave = sinewave(500) + sinewave(1500)
f, t, Sxx = scipy.signal.spectrogram(wave, fs=44100)
plt.pcolormesh(t, f, Sxx)
Audio(wave, rate=44100)
```

Out[7]:



Spectrogram of human speech



NOTE: *We do not expect you to understand the specifics of phonetics or acoustics. For the exam, we only expect you to have a basic understanding of how audio data is represented and processed in general.*

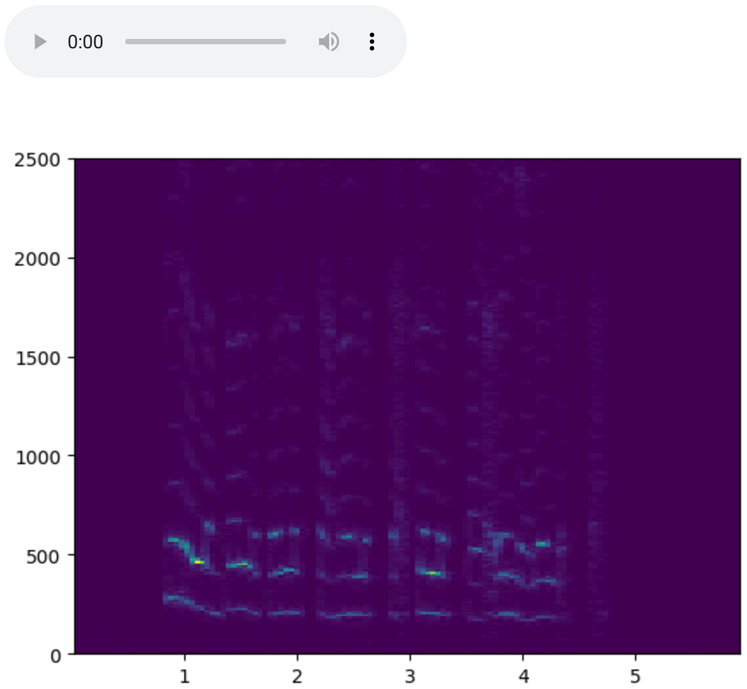
Average vowel formants for a male voice

Vowel (IPA)	Formant F1 (Hz)	Formant F2 (Hz)
i	240	2400
y	235	2100
e	390	2300
ø	370	1900
ɛ	610	1900
œ	585	1710

Source: https://en.wikipedia.org/wiki/Formant

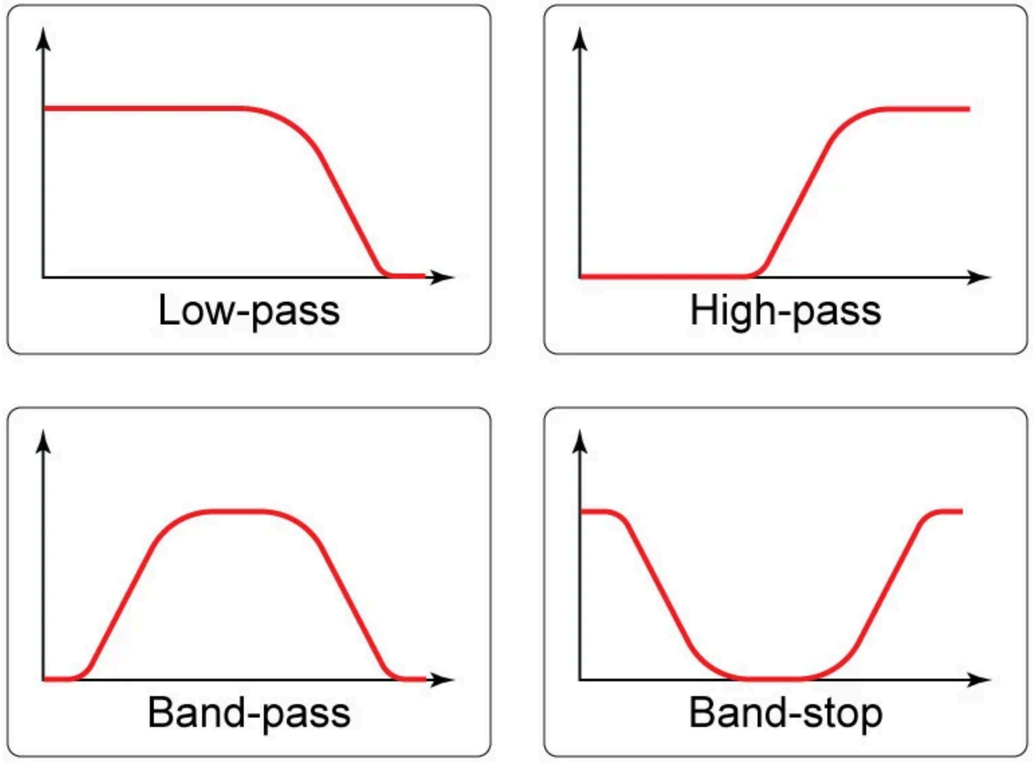
In [8]:
```
def plot_spectrogram(data, sampling_rate, ylim=None):
    f, t, Sxx = scipy.signal.spectrogram(data, fs=sampling_rate, nperseg=4096, noverlap=2048, mode=
    plt.pcolormesh(t, f, Sxx)
    if ylim is not None:
        plt.ylim(ylim)

audio, sampling_rate = sf.read('tongue-twister.wav')
plot_spectrogram(audio, sampling_rate, ylim=[0, 2500])
Audio(audio, rate=sampling_rate)
```

Out[8]:



Frequency filtering

We can use **filters** to make certain frequencies quieter or louder.



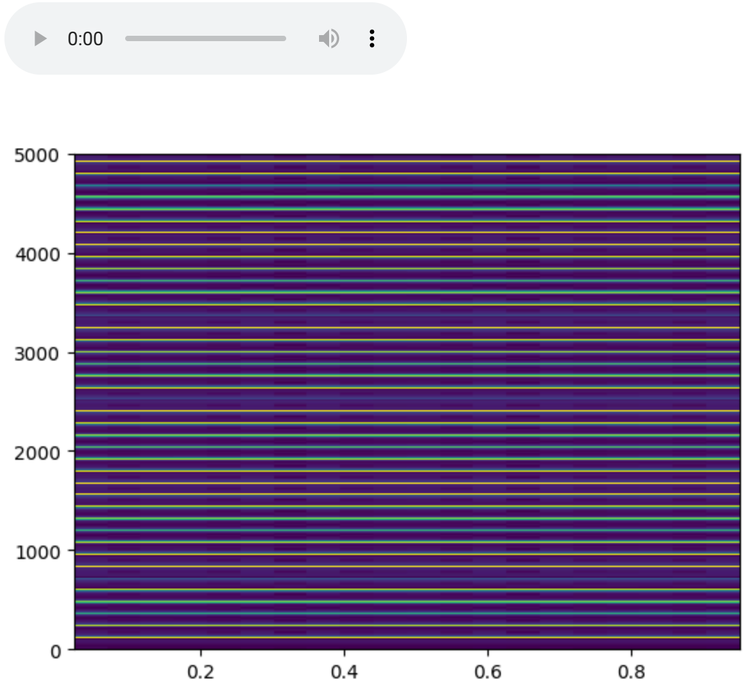
Synthesizing vowel sounds

1. Generate a source signal made up of harmonics (~ vocal cords).
2. Manipulate the source signal with formant filters (~ vocal tract).

In [9]:

```
# Generate harmonics based on fundamental frequency
f0 = 120
harmonics = sum(sinewave(f) for f in range(f0, 10000, f0))

plot_spectrogram(harmonics, 44100, ylim=[0, 5000])
Audio(harmonics, rate=44100)
```

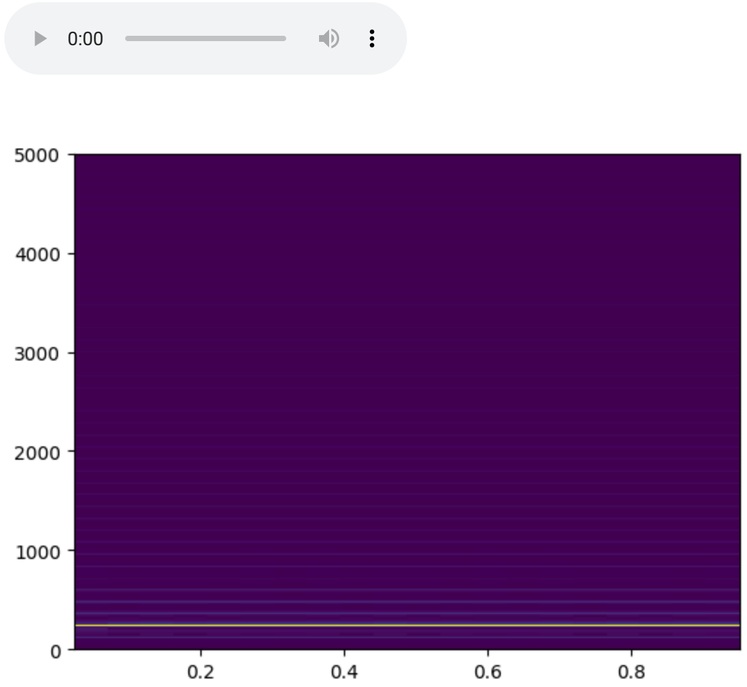
Out[9]:



In [10]:

```
def bandpass_filter(data, lowcut, highcut, sampling_rate, order=1):
    b, a = scipy.signal.butter(order, [lowcut, highcut], fs=sampling_rate, btype='band')
    y = scipy.signal.lfilter(b, a, data)
    return y

# Filter harmonics based on first formant frequency
f1 = 240
filtered_harmonics_f1 = bandpass_filter(harmonics, f1 - 20, f1 + 20, 44100)

plot_spectrogram(filtered_harmonics_f1, 44100, ylim=[0, 5000])
Audio(filtered_harmonics_f1, rate=44100)
```

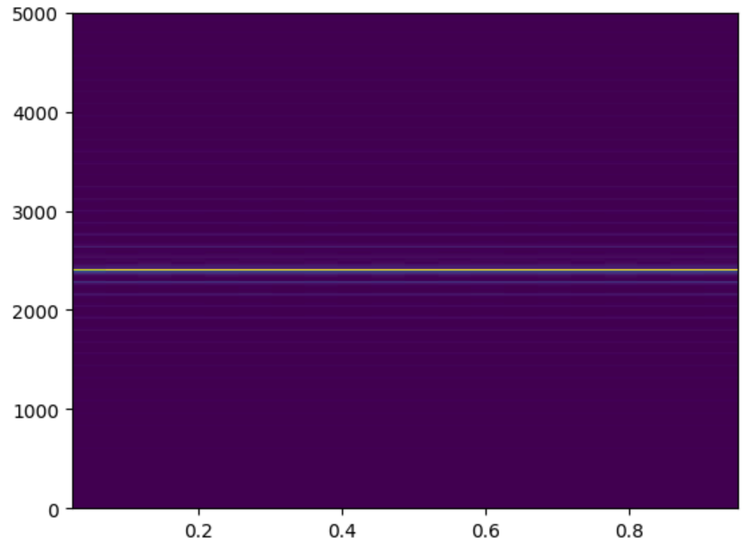
Out[10]:



In [11]:

```
# Filter harmonics based on second formant frequency
f2 = 2400
filtered_harmonics_f2 = bandpass_filter(harmonics, f2 - 20, f2 + 20, 44100)

plot_spectrogram(filtered_harmonics_f2, 44100, ylim=[0, 5000])
Audio(filtered_harmonics_f2, rate=44100)
```

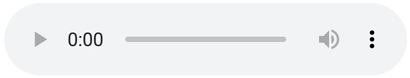
Out[11]:

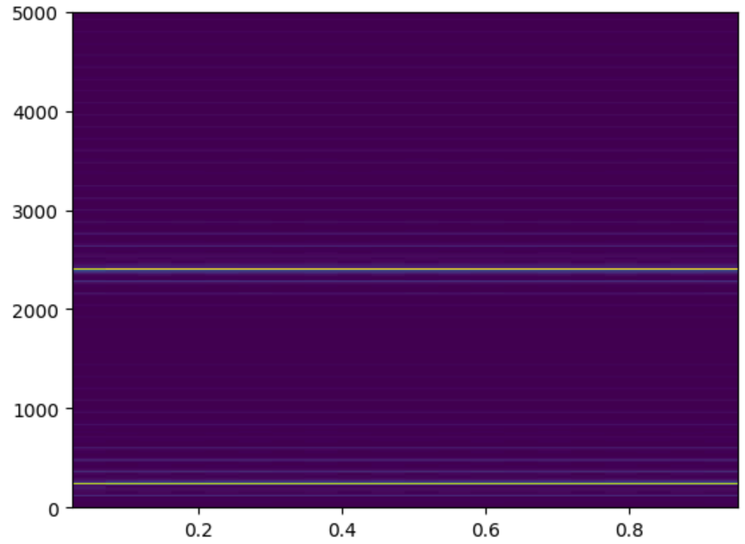


In [12]:

```
# Add the two filtered signals together
vowel = filtered_harmonics_f1 + filtered_harmonics_f2

plot_spectrogram(vowel, 44100, ylim=[0, 5000])
Audio(vowel, rate=44100)
```

Out[12]:





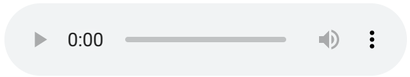
In [13]:

```
# Putting it all in a function

def generate_vowel(f0, f1, f2, sampling_rate=44100):
    harmonics = sum(sinewave(f) for f in range(f0, 10000, f0))
    filtered_harmonics_f1 = bandpass_filter(harmonics, f1 - 20, f1 + 20, sampling_rate)
    filtered_harmonics_f2 = bandpass_filter(harmonics, f2 - 20, f2 + 20, sampling_rate)
    return filtered_harmonics_f1 + filtered_harmonics_f2

i = generate_vowel(120, 240, 2400)
u = generate_vowel(120, 250, 595)
a = generate_vowel(120, 850, 1610)
```

In [14]:

```
Audio(a, rate=44100)
```

Out[14]:

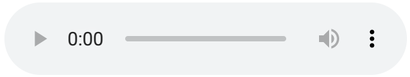


Remembering the sampling rate is essential

In [15]:
```
audio, sampling_rate = sf.read('tongue-twister.wav')
print(audio.shape)
print(sampling_rate)
Audio(audio, rate=22050)
```

```
(263808,)
44100
```

Out[15]:



Multiple audio channels

- So far, we've only talked about **mono** audio (1 channel).
- **Stereo** audio has 2 channels: left and right.
- **Surround sound** (common in cinemas) can have 8 or even more channels.

If we have n channels, we essentially have n separate audio signals.

Multiple audio channels

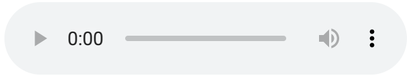
- So far, we've only talked about **mono** audio (1 channel).
- **Stereo** audio has 2 channels: left and right.
- **Surround sound** (common in cinemas) can have 8 or even more channels.

If we have n channels, we essentially have n separate audio signals.

In [16]:
```
audio, sampling_rate = sf.read('tongue-twister-stereo.wav')
print(audio.shape)
print(audio)
Audio(audio.T, rate=44100)
```

```
(263808, 2)
[[ 0.           0.        ]
 [ 0.           0.        ]
 [ 0.           0.        ]
 ...
 [-0.0020752  -0.0020752 ]
 [-0.00048828 -0.00048828]
 [ 0.00082397  0.00082397]]
```

Out[16]:



Converting stereo to mono:

In [17]:

```
audio_mono = audio.mean(axis=1)
print(audio_mono.shape)
print(audio_mono)
Audio(audio_mono, rate=44100)
```

```
(263808,)
[ 0.          0.          0.         ... -0.0020752  -0.00048828
  0.00082397]
```

Out[17]:

Packages for audio processing

- soundfile: basic reading and writing
- numpy: low-level inspection/editing (cropping, concatenation, channel selection, ...)
- scipy: low-level signal processing (filters, spectrograms, ...)
- librosa: high-level speech/music processing (beat tracking, feature extraction, effects, ...)

Image data

In [18]:
```
import PIL.Image

image = PIL.Image.open("hedgehog.jpg")
image
```

Out[18]:

Images as 3-dimensional arrays

In [19]:
```
image_array = np.asarray(image)
image_array.shape
```

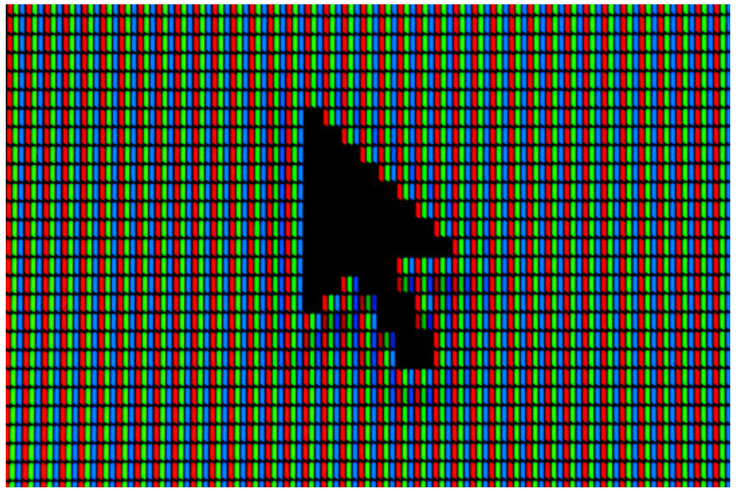
Out[19]:
```
(467, 700, 3)
```

In [20]:
```
image_array
```

Out[20]:
```
array([[[ 25,  52,   1],
        [ 27,  54,   1],
        [ 28,  57,   3],
        ...,
        [ 64, 106,   6],
        [ 61, 106,   5],
        [ 59, 104,   3]],

       [[ 25,  52,   1],
        [ 26,  55,   1],
        [ 28,  57,   1],
        ...,
        [ 65, 107,   7],
        [ 62, 106,   8],
        [ 60, 105,   4]],

       [[ 23,  52,   0],
        [ 25,  54,   0],
        [ 27,  56,   0],
        ...,
        [ 66, 108,  10],
```

RGB

Computer screens display colors as combinations of red, green, and blue light:



Usually, RGB images use 1 byte per color:

R	G	B	Color
0	0	0	
255	255	255	
128	128	128	
255	0	0	
0	255	0	
0	0	255	
177	127	4	

Colors are often represented in hexadecimal notation by concatenating the R, G, and B values:

RGB	Color
#FF0000	
#000088	
#A100FF	

Color channels

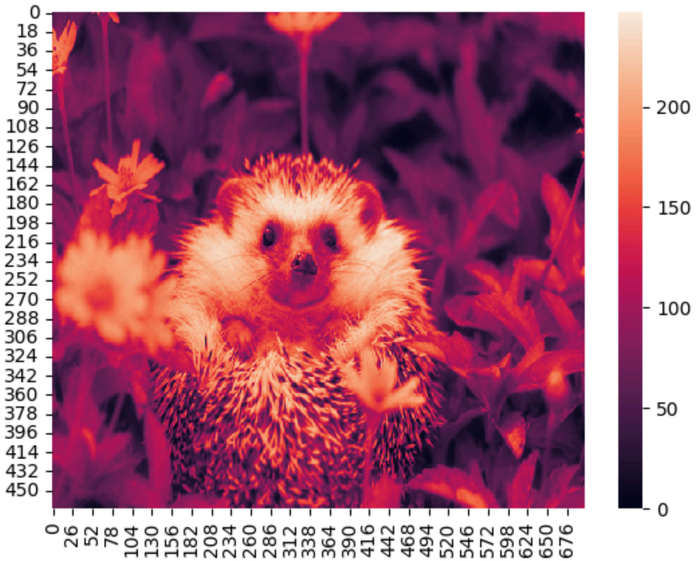
The R, G, and B channels of an image can be separated:

In [21]:
```
r = image_array[:, :, 0]
g = image_array[:, :, 1]
b = image_array[:, :, 2]

r.shape
```

Out[21]:
```
(467, 700)
```

In [22]:
```
sns.heatmap(g)
```

Out[22]:
```
<Axes: >
```



A simple way to convert an RGB image to black-and-white:

In [23]:
```
# Average the RGB channels
average = (r / 3 + g / 3 + b / 3).astype(np.uint8)
average.shape
```

Out[23]:
```
(467, 700)
```

A simple way to convert an RGB image to black-and-white:

In [23]:
```
# Average the RGB channels
average = (r / 3 + g / 3 + b / 3).astype(np.uint8)
average.shape
```

Out[23]:
```
(467, 700)
```

In [24]:
```
# Use the averaged array for all three channels
grayscale = np.stack([average] * 3, axis=2)
grayscale.shape
```

Out[24]:
```
(467, 700, 3)
```

In [25]:
```
# Convert the result back to a PIL image
PIL.Image.fromarray(grayscale)
```

Out[25]:

Adding an alpha channel: RGBA

- The alpha channel determines the transparency of each pixel.
- 0: fully transparent, 255: fully opaque.
- JPEG does not support alpha channels, but PNG does.

In [26]:
```
image = PIL.Image.open("python-logo.png")
image
```

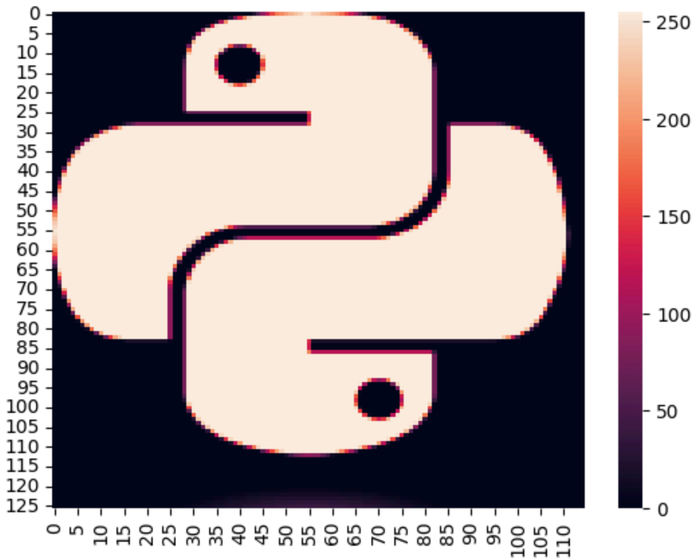
Out[26]:

In [27]:
```
image_array = np.asarray(image)
image_array.shape
```

Out[27]:
```
(126, 115, 4)
```

In [28]:
```
alpha = image_array[:, :, 3]
sns.heatmap(alpha)
```

Out[28]:
```
<Axes: >
```



Cropping, resizing, and rotating images with PIL

In [29]:
```
image = PIL.Image.open("hedgehog.jpg")
image.crop((200, 200, 400, 300))  # (left, top, right, bottom)
```

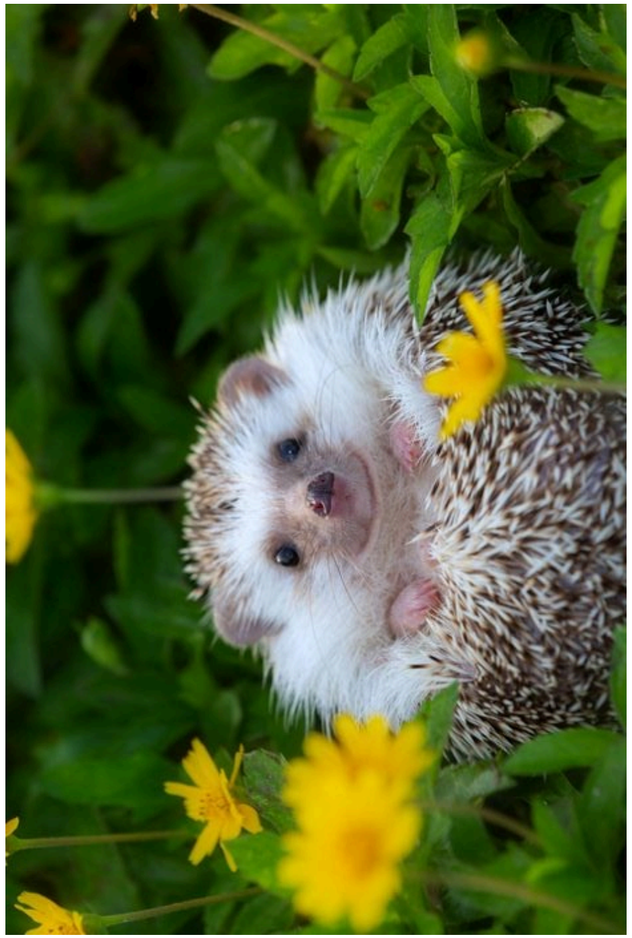
Out[29]:

In [30]:
```
image.resize((200, 200))
```

Out[30]:

In [31]:
```
image.rotate(90, expand=True)
```

Out[31]:



Lossless vs. lossy compression

3 KB

PNG

3 KB

JPEG

Quality 20

24 KB

JPEG

Quality 100

Format	Compression	Transparency	Use case
PNG	lossless	yes	Logos, icons, screenshots
JPEG	lossy	no	Photographs

Packages for image processing

- numpy: low-level inspection/editing
- PIL/Pillow: basic editing (cropping, resizing, rotating, ...)
- scikit-image: more advanced manipulation/analysis (filters, segmentation, ...)
- opencv-python: computer vision (face detection, object tracking, ...)

Video data

What we colloquially call a "video" usually consists of:

- A sequence of images (**frames**)
- An audio signal (one or more channels)

Video data

What we colloquially call a "video" usually consists of:

- A sequence of images (**frames**)
- An audio signal (one or more channels)

- **Frame rate** [fps]: number of frames per second
- **Resolution** [pixels]: width x height of each frame

The frame rate is comparable to the sampling rate in audio, but it is usually much lower (24-60 fps vs. 44100 Hz).

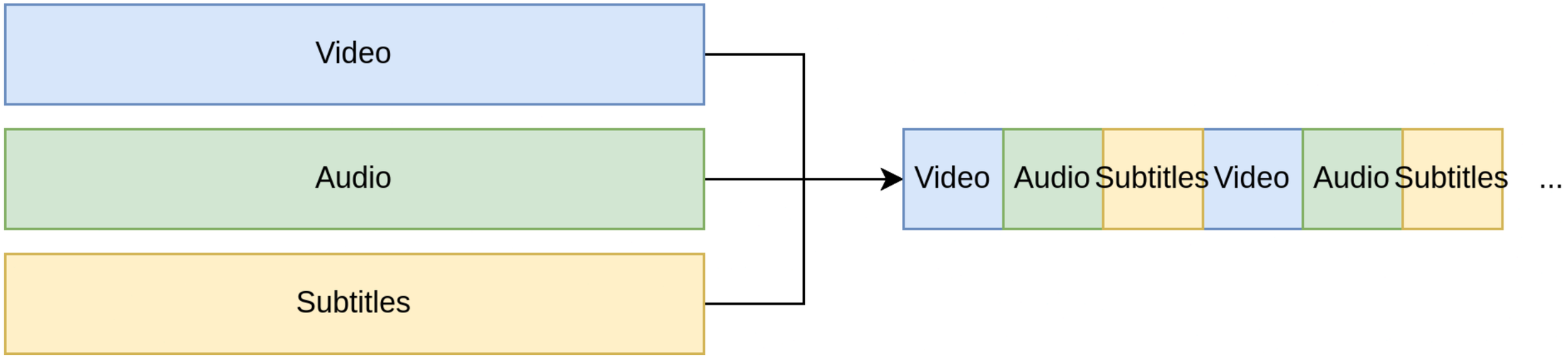
MPEG-4 / MP4 file format

MP4 is a commonly used **container format** that can store:

- Video: encoded as a sequence of frames
- Audio: encoded as a sequence of samples
- Subtitles: encoded as text
- Other data

MP4 is optimized for **streaming**:

- Multiple parallel streams of data (video, audio, subtitles) are interleaved/multiplexed into a single stream.
- This allows us to start playing the video before the entire file has been downloaded or read into memory.



FFmpeg

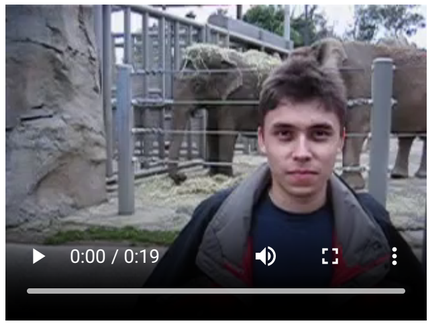
- An open-source library and CLI for working with audio and video files (not only MPEG-based ones).
- Many software products (media players, editors, etc.) use FFmpeg internally.
- Python libraries for working with video usually wrap FFmpeg.

→ **If you're interested in working with video data, learning FFmpeg is a good idea.**

Me at the zoo

In [32]:
```
Video("me-at-the-zoo.mp4")
```

Out[32]:



Inspecting the multiplexed data stream

In [33]:

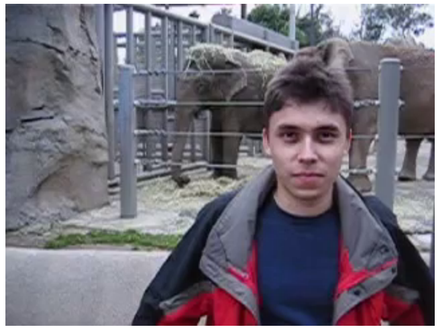
```
import av

with av.open("me-at-the-zoo.mp4") as container:
    stream = container.decode()
    for data in stream:
        print(data)
```

```
<av.VideoFrame #0, pts=0 yuv420p 320x240 at 0x7f4b293817e0>
<av.VideoFrame #1, pts=1024 yuv420p 320x240 at 0x7f4b29381850>
<av.VideoFrame #2, pts=2048 yuv420p 320x240 at 0x7f4b293819a0>
<av.VideoFrame #3, pts=3072 yuv420p 320x240 at 0x7f4b29381a10>
<av.VideoFrame #4, pts=4096 yuv420p 320x240 at 0x7f4b29381a80>
<av.VideoFrame #5, pts=5120 yuv420p 320x240 at 0x7f4b29381af0>
<av.VideoFrame #6, pts=6144 yuv420p 320x240 at 0x7f4b29381b60>
<av.AudioFrame 0 pts=0, 1024 samples at 44100Hz, mono, fltp at 0x7f4b29381c40
<av.AudioFrame 1 pts=1024, 1024 samples at 44100Hz, mono, fltp at 0x7f4b29381cb0
<av.AudioFrame 2 pts=2048, 1024 samples at 44100Hz, mono, fltp at 0x7f4b29381b60
<av.AudioFrame 3 pts=3072, 1024 samples at 44100Hz, mono, fltp at 0x7f4b29381c40
<av.AudioFrame 4 pts=4096, 1024 samples at 44100Hz, mono, fltp at 0x7f4b29381cb0
<av.AudioFrame 5 pts=5120, 1024 samples at 44100Hz, mono, fltp at 0x7f4b29381b60
<av.AudioFrame 6 pts=6144, 1024 samples at 44100Hz, mono, fltp at 0x7f4b29381c40
<av.AudioFrame 7 pts=7168, 1024 samples at 44100Hz, mono, fltp at 0x7f4b29381cb0
<av.AudioFrame 8 pts=8192, 1024 samples at 44100Hz, mono, fltp at 0x7f4b29381b60
<av.AudioFrame 9 pts=9216, 1024 samples at 44100Hz, mono, fltp at 0x7f4b29381c40
```

Extracting video frames

In [34]:
```
with av.open("me-at-the-zoo.mp4") as container:
    video_stream = container.decode(video=0)
    frame = next(video_stream)
    image = frame.to_image()
image
```

Out[34]:



Extracting audio

In [35]:
```
with av.open("me-at-the-zoo.mp4") as container:
    audio_stream = container.decode(audio=0)
    frame = next(audio_stream)
    audio = frame.to_ndarray()

audio.shape
```

Out[35]:
```
(1, 1024)
```

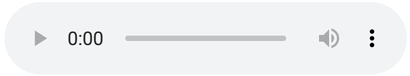
In [36]:
```
audio
```

Out[36]:
```
array([[0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

In [37]:
```
with av.open("me-at-the-zoo.mp4") as container:
    full_audio = np.zeros((1, 0))
    for frame in container.decode(audio=0):
        audio = frame.to_ndarray()
        full_audio = np.concatenate([full_audio, audio], axis=1)

print(full_audio.shape)
Audio(full_audio[0], rate=frame.sample_rate)
```

```
(1, 840704)
```

Out[37]:



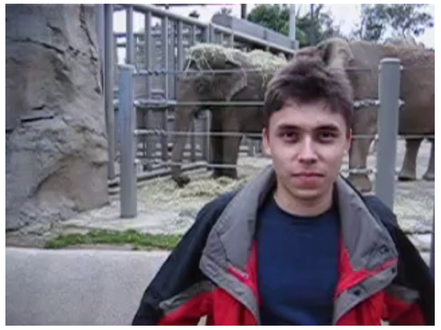
MoviePy

NOTE: *The latest version of MoviePy is not compatible with the latest version of NumPy. Install* `numpy==1.24.4` *to avoid issues.*

In [38]:
```
from moviepy.editor import VideoFileClip

clip = VideoFileClip("me-at-the-zoo.mp4")
frame = clip.get_frame(0)

print(frame.shape)
PIL.Image.fromarray(frame)
```

```
(240, 320, 3)
```

Out[38]:

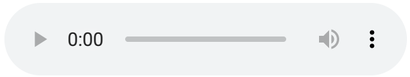


In [39]:

```
audio = clip.audio
audio_array = audio.to_soundarray()

print(audio_array.shape)
Audio(audio_array.T, rate=audio.fps)
```

```
(840546, 2)
```

Out[39]:



Conclusion

- **Audio** signal = sequence of float-valued samples
    - **Sample rate**: number of samples played per second
    - **Frequency**: number of oscillations per second in a periodic signal
    - **Channels**: parallel audio signals

      mono = 1 channel, **stereo** = 2 channels
- **Image** = array of RGB(A) values
    - **Resolution**: width/height in pixels
    - **Color channels**: R = red, G = green, B = blue, A = alpha
    - **Compression**: lossy (e.g., JPEG) vs. lossless (e.g., PNG)
- **Video** = multimodal data streams (video, audio, subtitles)
    - **Frame rate**: number of video frames (images) played per second

Tools / libraries:

- NumPy for sample-level/pixel-level processing
- **Audio**: FFmpeg, `soundfile`, `librosa`
- **Image**: `PIL`/Pillow, `scikit-image`, `opencv-python`
- **Video**: FFmpeg, `av`, `moviepy`

Quiz: Test your understanding