Making music

September 3, 2019

```
In [1]: import random
        import matplotlib.pyplot as plt
        %matplotlib inline
        import numpy as np

        import scipy.io.wavfile as wav
        import IPython.display as ipd
```

1 Some introductory functions

1.0.1 List comprehensions

List comprehensions are a versatile syntax for mapping a function (or expression) across all elements of a list. Read the function below. Do you understand what is happening with the arguments and retun values?

```
In [2]: # The function below accepts a list L
        # and returns another list with elements of L three times as large
        def three_ize(L):
            # this is an example of a list comprehension
            LC = [3 * x for x in L]
            return LC
```

What is the return value of `three_ize(list1)`?

```
In [3]: list1 = [1,4,5,10]
```

```
In [4]: #clear
        three_ize(list1)
```

```
Out[4]: [3, 12, 15, 30]
```

1.0.2 Write the function `scale`

The function should have the following signature:

```
In [5]: def scale(L, scale_factor):
            '''
```

```
        returns a list similar to L, except that each element has been
        multiplied by scale_factor.
        '''
```

```
In [6]: #clear
        def scale(L, scale_factor):
            '''
            returns a list similar to L, except that each element has been
            multiplied by scale_factor.
            '''
            LC = [scale_factor * x for x in L]
            return LC
```

Now you can use the function `scale` with the given variable `list`

```
In [7]: #clear
        list4 = scale(list1,4)
        print(list4)
```

```
[4, 16, 20, 40]
```

1.0.3 Write the function `add_2`:

The function should have the following signature:

```
In [8]: def add_2(L, M):
            '''
            takes two lists L and M and
            returns a single list that is an element-by-element sum of the two arguments
            If the arguments are different lengths, the function add_2 should
            return a list that is as long as the shorter of the two.
            Just ignore the extra elements from the longer list.
            '''
```

```
In [9]: #clear
        def add_2(L, M):
            '''
            takes two lists L and M and
            returns a single list that is an element-by-element sum of the two arguments
            If the arguments are different lengths, the function add_2 should
            return a list that is as long as the shorter of the two.
            Just ignore the extra elements from the longer list.
            '''
            N = min(len(L),len(M))
            list_add = []
            for i in range(N):
                list_add.append(L[i]+M[i])
            return(list_add)
```

Define two lists, and use your function `add_2`

```
In [10]: #clear
         add_2(list1,list4)
```

```
Out[10]: [5, 20, 25, 50]
```

1.0.4 Write the function `add_scale_2`:

The function should have the following signature:

```
In [11]: def add_scale_2(L, M, L_scale, M_scale):
             '''
             takes two lists L and M and two floating-point numbers L_scale and M_scale.
             These stand for scale for L and scale for M, respectively.
             Returns a single list that is an element-by-element sum of the two inputs,
             each scaled by its respective floating-point value.
             If the inputs are different lengths, your add_scale_2 should return a list that is
             as long as the shorter of the two. Again, just drop any extra elements.
              '''
```

```
In [12]: #clear
         def add_scale_2(L, M, scale1, scale2):
             '''
             takes two lists L and M and two floating-point numbers L_scale and M_scale.
             These stand for scale for L and scale for M, respectively.
             Returns a single list that is an element-by-element sum of the two inputs,
             each scaled by its respective floating-point value.
             If the inputs are different lengths, your add_scale_2 should return a list that is
             as long as the shorter of the two. Again, just drop any extra elements.
              '''
             N = min(len(L),len(M))
             list_add = []
             for i in range(N):
                 list_add.append(scale1*L[i]+scale2*M[i])
             return(list_add)
```

What is the result of

```
L1 = [1,3,5,2]
L2 = [3,1,4,4]
add_scale_2(L1,L2,2,3)
```

```
In [13]: #clear
         L1 = [1,3,5,2]
         L2 = [3,1,4,4]
         add_scale_2(L1,L2,2,3)
```

```
Out[13]: [11, 9, 22, 16]
```

1.0.5 How can you obtain the same result using numpy arrays?

Define two numpy arrays and perform the same operation defined by the function `add_scale_2`

```
In [14]: #clear
         M1 = np.array(L1)
         M2 = np.array(L2)
         2*M1+3*M2

Out[14]: array([11,  9, 22, 16])
```

1.0.6 Helper function `add_noise`

Take a look at the function below. What is happening to the scalar argument `x`?

```
In [15]: def add_noise(x, chance_of_replacing, noise):
             """add_noise accepts an original value, x
                and a fraction named chance_of_replacing.

                With the "chance_of_replacing" chance, it
                returns the number x + noise

                Otherwise, it should return x
             """
             r = random.uniform(0, 1)
             if r < chance_of_replacing:
                 return x + noise
             else:
                 return x
```

1.0.7 Create the function `array_add_noise`

Create a function `array_add_noise` that replace entries in a numpy array `L` using the helper function above. Entries in the array should be replaced with probability `prob`

```
In [16]: def array_add_noise(L,prob,noise):
             '''takes a 1D numpy array L,
             and modify entries of L using the helper function add_noise
             Note that this function should not return a new object,
             it should instead modify the given object
             '''

In [17]: #clear
         # Replace entries in a numpy array using the helper function randomize
         def array_add_noise(L,prob,noise):
             for i in range(len(L)):
                 L[i] = add_noise(L[i],prob,noise)
```

You are given the numpy array:

```
In [18]: L = np.array([4,2,5,6,9],dtype=float)
```

What happens to L after you call the function `array_add_noise`? Print L and `id(L)` before and after the function call.

```
In [19]: #clear
         print(L, id(L))
         array_add_noise(L,0.4,0.1*min(L))
         print(L, id(L))
```

```
[4. 2. 5. 6. 9.] 103301150080
[4.  2.2 5.  6.2 9.2] 103301150080
```

Modify the function `array_add_noise` defined above to take an optional parameter `inplace` which by default is True. When `inplace` is False, the function will create a new numpy array and return it with the modified values, but it won't replace the entries in the original array.

```
In [20]: #clear
         # Replace entries in a numpy array using the helper function randomize
         def array_add_noise(L,prob,noise,inplace=True):
             if inplace:
                 for i in range(len(L)):
                     L[i] = add_noise(L[i],prob,noise)
                 return
             else:
                 newL = np.copy(L)
                 for i in range(len(L)):
                     newL[i] = add_noise(L[i],prob,noise)
                 return newL
```

Use the updated `array_add_noise` function to: 1) modify a given numpy array inplace 2) create a new numpy array Print the numpy array before and after the function call. Print the `id`. What do you observe?

```
In [21]: #clear

         L = np.array([4,2,5,6,9],dtype=float)
         print(L,id(L))

         Lnew = array_add_noise(L,0.5,0.1,inplace=True)

         print(L,Lnew,id(L),id(Lnew))
```

```
[4. 2. 5. 6. 9.] 103301148720
[4.  2.  5.1 6.  9.1] None 103301148720 4407013448
```

2 Let's start playing with sounds

2.0.1 Create a sinusoidal sound

```
In [22]: # Define the duration of the sound we want to create:
         # duration in seconds
         duration = 5
```

```
In [23]: # Define the rate of the sound, which is the number of sample points per second
         DEFAULT_RATE = 44100
```

```
In [24]: # The total number of sample points that define your sound is:
         nsamples = int(DEFAULT_RATE*duration)
```

```
In [25]: # Then create a numpy array that define the range of the sound, i.e.,
         # nsamples points equaly spaced in the range (0,duration) [s]
         t = np.linspace(0,duration,nsamples)
```

Create a sound array corresponding to the function

$$f(t) = \sin(220(2\pi t)) + \sin(224(2\pi t))$$

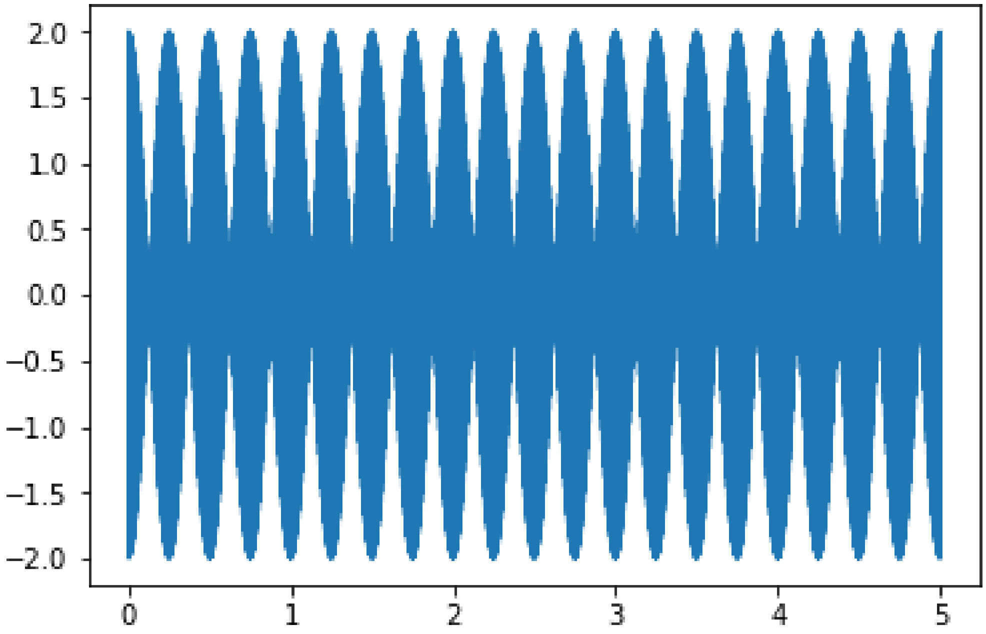
data = ...

```
In [26]: #clear
         data = np.sin(2*np.pi*220*t) + np.sin(2*np.pi*224*t)
```

Use `plt.plot(t,data)` to plot your function:

```
In [27]: #clear
         # Plot the sound
         plt.plot(t,data)
```

```
Out[27]: [<matplotlib.lines.Line2D at 0x180d3ef4a8>]
```



Check the sound you just created!

```
In [28]: #ipd.Audio(data,rate=DEFAULT_RATE)
```

You can also try different functions!

2.0.2 Create a music note

Let's make the sound of the A5 note. (https://en.wikipedia.org/wiki/Piano_key_frequencies)

```
In [29]: # We want to have the note played for 0.5 seconds
         duration = 0.5
         # Define the rate
         rate = DEFAULT_RATE
         # The number of samples needed is
         nsamples = int(rate*duration)
         # # The frequency of A5 is 880.
         freq = 880
         t = np.linspace(0, duration, nsamples)
         data = np.sin(freq*2*np.pi*t)
         ipd.Audio(data,rate=rate)
```

```
Out[29]: <IPython.lib.display.Audio object>
```

2.0.3 Write a function `make_note`

Just add the steps described above to define the function `make_note`

```
In [30]: def make_note(freq, duration=0.3, rate=DEFAULT_RATE):
             '''
             receives as arguments:
                 - frequency of the note (freq)
                 - duration of the sound (set as default equal to 0.3)
                 - rate (samples per second)
             and returns:
                 - np.array data with the beep
             '''
```

```
In [31]: #clear
         def make_note(freq, duration=0.3, rate=DEFAULT_RATE):
             '''
             receives as arguments:
                 - frequency of the note (freq)
                 - duration of the sound (set as default equal to 0.3)
                 - rate (samples per second)
             and returns:
                 - np.array data with the beep
```

```
        '''
        nsamples = int(rate * duration)
        # make a time sequence
        t = np.linspace(0, duration, nsamples)
        # make a (sine) sound wave with frequency = freq
        data = np.sin(freq*2*np.pi*t)

        return data
```

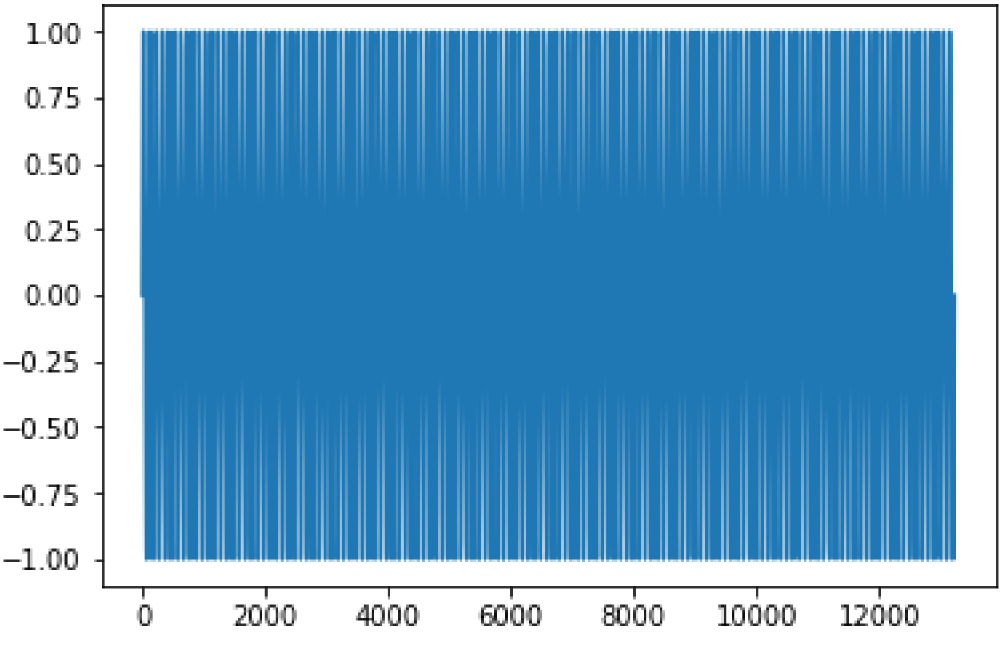
Use `note = make_note(...)` function to create the following sounds:

note	duration	freq
A4	3	440
C4	4	261.6256

Then you can plot the sound array using:
`plt.plot(note)`
And listen to the sound using:
`ipd.Audio(note,rate=DEFAULT_RATE)`

```
In [32]: #clear
         note_A5 = make_note(440)
         plt.plot(note_A5)
         ipd.Audio(note_A5,rate=DEFAULT_RATE)
```

```
Out[32]: <IPython.lib.display.Audio object>
```



2.0.4 Modify the function `make_note` so that it parabolically decays to zero over the time duration of the sound

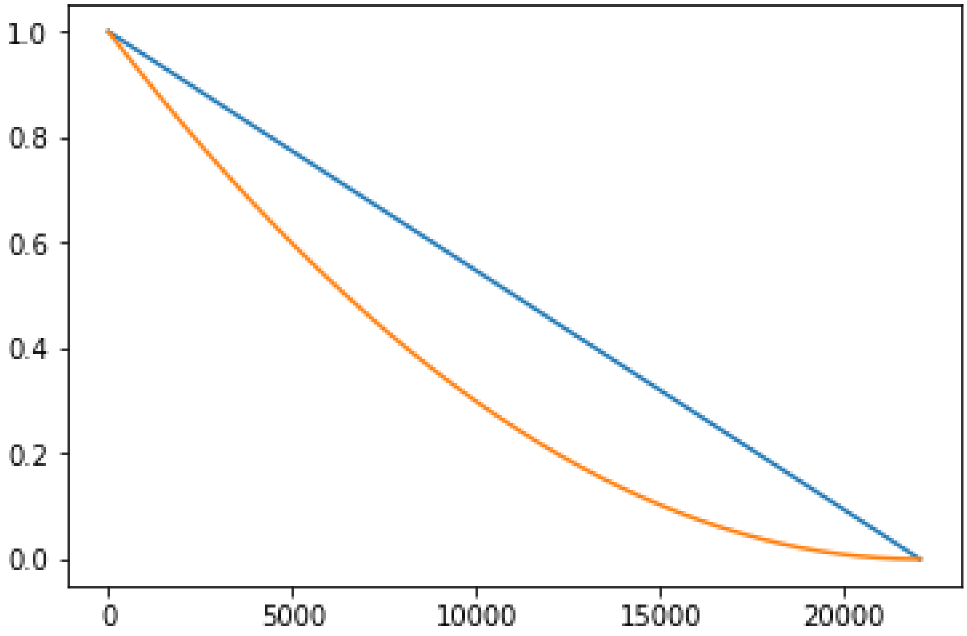
We need a ramp function, which starts with value equal to 1 and finishes with value of zero, and includes nsamples data points

```
In [33]: ramp = np.linspace(0, 1, nsamples)
```

Check the function that gives the linear decay:
`plt.plot(1-ramp)`
And also the functin that gives the parabolical decay
`plt.plot((1-ramp)**2)`

```
In [34]: #clear
         # Here is the linear decay
         plt.plot(1-ramp)
         # Here is the parabolical decay
         plt.plot((1-ramp)**2)
```

```
Out[34]: [<matplotlib.lines.Line2D at 0x1817595748>]
```



Modify the function `make_note` so that it applies the decay above to the data array

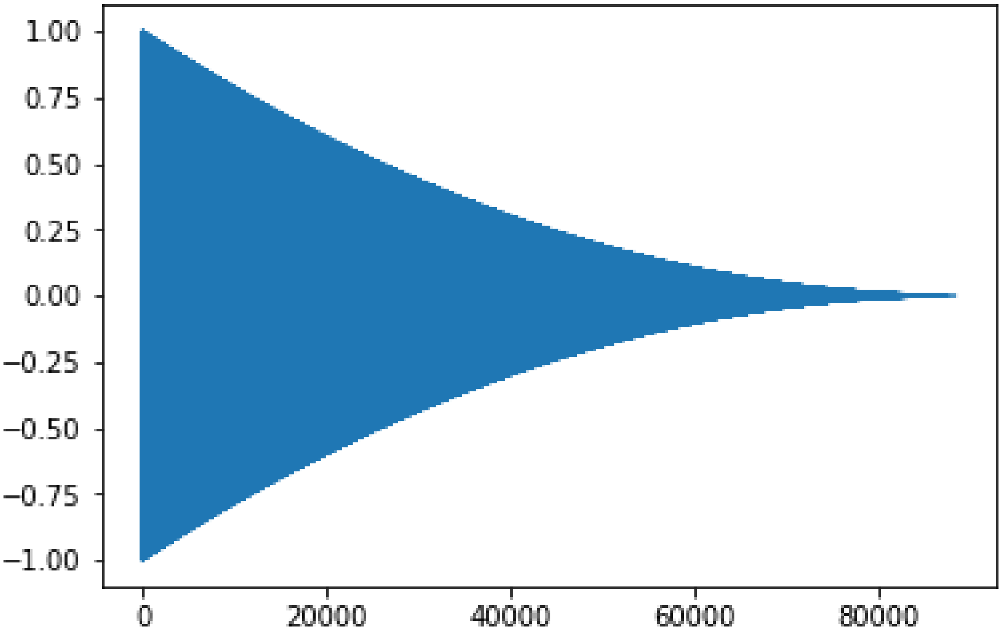
```
In [35]: #clear
         def make_note(freq, duration=0.3, rate=DEFAULT_RATE):
             '''
             receives as arguments:
```

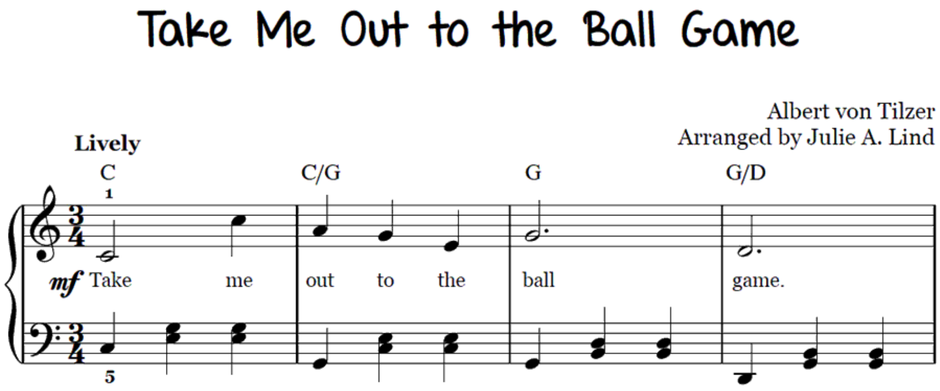
```
        - frequency of the note (freq)
        - duration of the sound (set as default equal to 0.3)
        - rate (samples per second)
    and returns:
        - np.array data with the beep
    '''
    nsamples = int(rate * duration)
    # make a time sequence
    t = np.linspace(0, duration, nsamples)
    # make a (sine) sound wave with frequency = freq
    data = np.sin(freq*2*np.pi*t)
    ramp = np.linspace(0, 1, nsamples)

    return data*(1-ramp)**2
```

Use your function to create the note A8 (freq=7040) with duration of 2 seconds. Then plot the sound array using `plt.plot(note)` and listen to the sound using `ipd.Audio(note,rate=DEFAULT_RATE)`

```
In [36]: #clear
         data_A1 = make_note(7040,duration=2)
         plt.plot(data_A1)
         ipd.Audio(data_A1,rate=DEFAULT_RATE)
```

```
Out[36]: <IPython.lib.display.Audio object>
```





title

2.0.5 Make music

You can use numpy.hstack to combine notes to make music. Try to make a music by using the frequencies in `freq_example` consecutively, using the same duration for all notes. Store the combined array in the variable `music`.

```
In [37]: freq_example = [261.6256,293.6648,329.6276,349.2282,391.9954,440.0000,493.8833,523.2511
```

```
In [38]: music = ...
```

```
In [39]: #clear
         music = np.array([])
         for fr in freq_example:
             music = np.hstack( (music, make_note(fr,duration=0.5)) )
```

Listen to the music you created using `ipd.Audio(music,rate=DEFAULT_RATE)`

```
In [40]: #clear
         ipd.Audio(music,rate=DEFAULT_RATE)
```

```
Out[40]: <IPython.lib.display.Audio object>
```

What did you get?

2.0.6 We can make "real" music :-)

Here is how we could write the song above:

note	duration	freq
C	2	261.626
C	1	523.251
A	1	440.0
G	1	391.995
E	1	329.628
G	3	391.995
D	3	293.665

We enter the above information as a list of lists:

```
In [41]: notes = [
             [2,261.626]
```

```
    # make a time sequence
    t = np.linspace(0, duration, nsamples)
    # make a (sine) sound wave with frequency = freq
    data = np.sin(freq*2*np.pi*t)
    ramp = np.linspace(0, 1, nsamples)

    return data*(1-ramp)**2
```

```
In [43]: music = np.array([])

         for note in notes:
             music = np.hstack( (music, make_note(note[1],duration=0.5*note[0])) )
         ipd.Audio(music,rate=DEFAULT_RATE)

Out[43]: <IPython.lib.display.Audio object>

In [44]: ipd.Audio(music,rate=DEFAULT_RATE)

Out[44]: <IPython.lib.display.Audio object>
```

where `notes[i]` gives the list [duration,freq] for the note `i`. You can again use hstack (or any other method you want) to combine the notes to make music.

Create the numpy array `music` using the list `notes` above, and play the music using `ipd.Audio(music,rate=DEFAULT_RATE)`

2.0.7 Name the music!

I will now give you different notes, and you will tell me the name of the music.

```
In [45]: #clear
         # Frozen
         notes = [
          [0.5,392],
          [0.5,392],
          [0.5,392],
          [0.5,293.66],
          [0.5,392],
          [0.5,493.88],
          [1,440],
          [3,493.88],
          [1,0],
          [0.5,392],
          [0.5,392],
          [0.5,293.66],
          [0.5,392],
          [0.5,493.88],
          [2,440]]

         # Star is Born
```

```
notes = [[0.5,659.2551],
         [0.5,659.2551],
        [0.5,659.2551],
        [1,587.3295],
        [4.5,493.8833],
         [1,0],
         [0.5,523.2511],
         [0.5,523.2511],
         [0.5,523.2511],
         [0.5,523.2511],
         [0.5,523.2511],
         [0.5,523.2511],
         [0.5,523.2511],
        [1,493.8833],
         [2.5,440.0000]
]

# Star Wars
notes = [[2,261.6256],
         [1,391.9954],
        [0.5,349.2282],
         [0.5, 329.6276],
        [0.5,293.6648],
        [2, 523.2511],
        [1,391.9954],
         [0.5,349.2282],
         [0.5, 329.6276],
        [0.5,293.6648],
        [2, 523.2511],
        [1,391.9954],
         [0.5,349.2282],
         [0.5, 329.6276],
        [0.5,349.2282],
        [3,293.6648]
]

# The greatest show man
notes = [[0.6,293.6648],
        [0.6,440.0000],
        [1.2, 369.9944 ],
        [0.3, 0 ],
        [0.6,293.6648],
        [0.6,440.0000],
        [1.2, 369.9944 ],
        [0.3, 0 ],
        [0.6,293.6648],
        [0.6,440.0000],
        [1.2, 369.9944 ],
```

```
               [0.4, 369.9944 ],
               [0.8, 369.9944 ],
               [0.4, 329.6276],
               [1.2, 329.6276] ,
               [0.6, 293.6648],
               [0.6, 293.6648],
               [1.6, 293.6648]]
```

```
In [46]: notes = [[2,261.6256],
                  [1,391.9954],
                 [0.5,349.2282],
                  [0.5, 329.6276],
                 [0.5,293.6648],
                 [2, 523.2511],
                 [1,391.9954],
                  [0.5,349.2282],
                  [0.5, 329.6276],
                 [0.5,293.6648],
                 [2, 523.2511],
                 [1,391.9954],
                  [0.5,349.2282],
                  [0.5, 329.6276],
                 [0.5,349.2282],
                 [3,293.6648]
         ]
```

```
In [47]: #clear
         music = np.array([])

         for note in notes:
             music = np.hstack( (music, make_note(note[1],duration=0.5*note[0])) )
         ipd.Audio(music,rate=DEFAULT_RATE)
```

```
Out[47]: <IPython.lib.display.Audio object>
```

2.0.8 Let's listen to movie sound clips

```
In [48]: # Name the movie!
         ipd.Audio("swnotry.wav")
```

```
Out[48]: <IPython.lib.display.Audio object>
```

```
In [49]: # Name the movie!
         ipd.Audio("honest.wav")
```

```
Out[49]: <IPython.lib.display.Audio object>
```

2.0.9 Inspect the type of the data in `music_data`

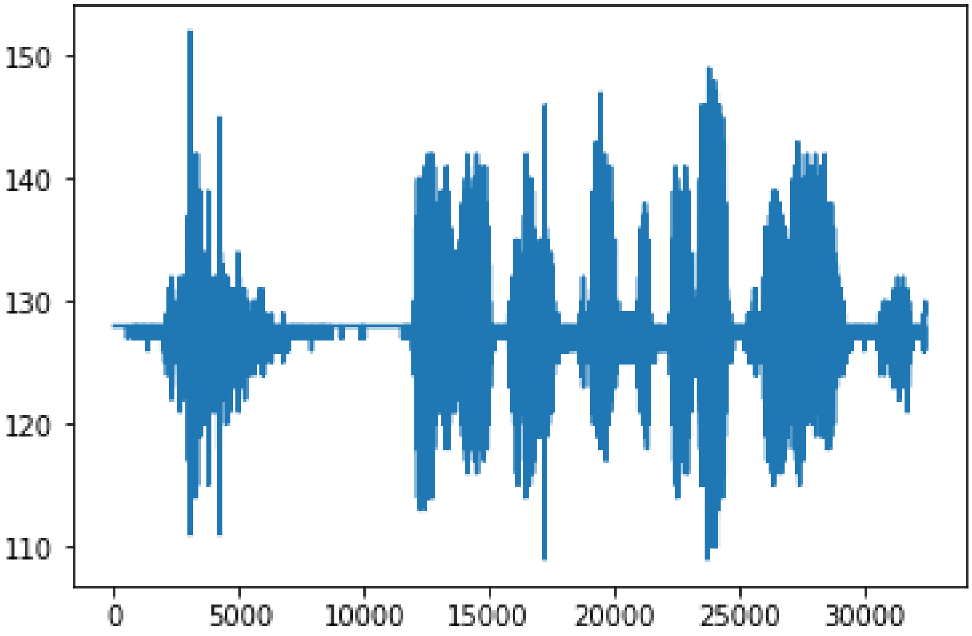
```
In [50]: # scipy.io.wavfile.read: return the sample rate (in samples/sec) and data from a WAV fi
         filename = "honest.wav"
         rate, music_data = wav.read(filename)
         print("The sound has rate (in samples per second) = ", rate)
         print("The data has", len(music_data), "sample points")
         print(type(music_data))
```

```
The sound has rate (in samples per second) =  11025
The data has 32470 sample points
<class 'numpy.ndarray'>
```

We can also play the `music_data` array obtained using `wav.read`:
sound = np.array(music_data,dtype=float)
plt.plot(sound)
And we use the same `rate` to play the audio
ipd.Audio(sound, rate=rate)

```
In [51]: #clear
         sound = np.array(music_data,dtype=float)
         plt.plot(sound)
         ipd.Audio(sound, rate=rate)
```

```
Out[51]: <IPython.lib.display.Audio object>
```



2.0.10 Change the speed of the sound

Make it twice as fast

```
In [52]: #clear
         ipd.Audio(sound, rate=2*rate)
```

```
Out[52]: <IPython.lib.display.Audio object>
```

Make it twice as slow

```
In [53]: #clear
         ipd.Audio(sound, rate=0.5*rate)
```

```
Out[53]: <IPython.lib.display.Audio object>
```

2.0.11 Add noise to the sound

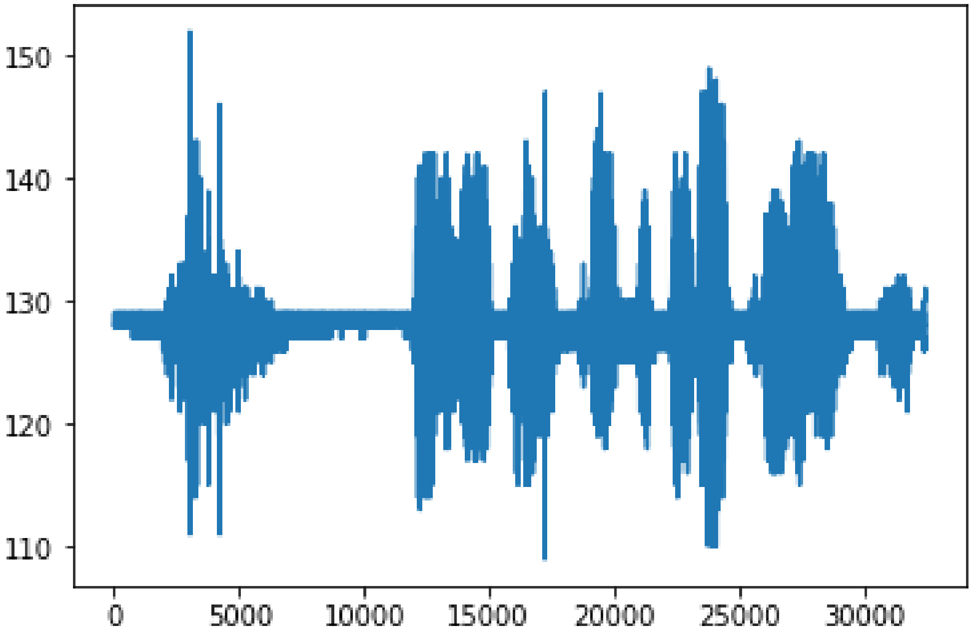
Let's modify at random some of the elements of the numpy array.
Use the function `array_add_noise` to create the variable `noisy_sound`
`noisy_sound = array_add_noise(sound, ...)`
Choose the probability and the noise level

```
In [54]: #clear
         noise_level = 0.01*min(sound)
         prob = 0.4
         noisy_sound = array_add_noise(sound,prob,noise_level,inplace=False)
```

Then you can plot and play:
plt.plot(noisy_sound)
ipd.Audio(noisy_sound, rate=rate)

```
In [55]: #clear
         plt.plot(noisy_sound)
         ipd.Audio(noisy_sound, rate=rate)
```

```
Out[55]: <IPython.lib.display.Audio object>
```



2.0.12 Scramble the sound!

Check this one out! You can play with the number of splits.

```
In [56]: split_sound = np.array_split(sound, 8)
         np.random.shuffle(split_sound)
         flat_list = [item for sublist in split_sound for item in sublist]
```

```
In [57]: ipd.Audio(np.array(flat_list), rate=rate)
```

```
Out[57]: <IPython.lib.display.Audio object>
```

2.0.13 Combine two different sounds

Let's have fun with these audio clips!

```
In [58]: filename1 = "odds.wav"
         sr1, data1 = wav.read(filename1)
         sound1 = np.array(data1,dtype=float)
```

```
In [59]: ipd.Audio(data1,rate=sr1)
```

```
Out[59]: <IPython.lib.display.Audio object>
```

```
In [60]: filename2 = "pass.wav"
         sr2, data2 = wav.read(filename2)
         sound2 = np.array(data2,dtype=float)
```

```
In [61]: ipd.Audio(data2,rate=sr2)

Out[61]: <IPython.lib.display.Audio object>
```

```
In [62]: data_combined = np.hstack((0.01*sound1,sound2))
         print(data1.shape, data2.shape,data_combined.shape)
         # plt.plot(data_combined)

(245376,) (65160,) (310536,)
```

```
In [63]: # ipd.Audio(data_combined[100000:],rate=(sr1+sr2)/2 )
```

We had to cheat and modify the magnitude of the first sound, so that we could hear both with similar volume. We also had to modify the rate, here using the average. Can you think of better ways to manipulate these two arrays to get something interesting?