Sonification 101: How to convert data into music with python

Matt Russo · Follow

14 min read · May 16, 2022

Listen Share More

Do you have some data you'd like to listen to? Maybe you want to communicate news or a scientific discovery with an emotional flair? Maybe you want to make the data accessible to people who are blind? Maybe you just want to have fun making wild and unconventional music? Try **sonification.**

I'm Matt Russo, co-founder of SYSTEM Sounds and NASA sonification specialist. In this article I'll show you the basic process for converting discrete data points into musical notes with python. You can also watch this lesson in video form, download the data and code, or follow along with the interactive web-based code (no installation required).

As an example, I'll walk you through the steps we used to create this. This is a simple example of parameter mapping sonification. The approach taken here tends to work best when the data already includes a time dimension, (so you're effectively speeding up or slowing down time), but you can work with any data you like. Python is a great language to do data sonification with. It's easy to get started, has many powerful libraries, and is commonly used in data science. You can also do all of the data processing, sonification, and even synchronize visuals within a single python script or within a jupyter notebook. To follow along, I recommend using jupyter so that each block of code is its own cell. If you're new to python, check out this guide. All of the libraries used can be installed with `pip install libraryname`. Let's go!

The Data

The first step is to make sure you completely understand the data. What is it measuring? How was it collected? What does it all mean? Depending on your sonification, this stage may require some background research. In our case, the data is a list of 111 lunar impact craters. Each crater was caused by an asteroid impact within the last 1.3 billion years and more energetic impacts created bigger craters. The craters in this set are ones which are larger than 10km across and have accurate age estimates. This data comes from a paper by a friend, Sara Mazrouei. She was able to date craters by looking at how quickly the nearby debris cooled as it was plunged into the darkness of the lunar night. Bigger boulders cool slower and their presence indicates more recent impacts. The data is in the form of columns in a `.csv` file with ages measured in millions of years and diameters measured in kilometres. It also contains the lunar longitude, latititude, and the names of some of the craters. The `.csv` file is currently in the same folder in which the script is being run. Let's load the data using the pandas library and take a look at the first 5 rows:

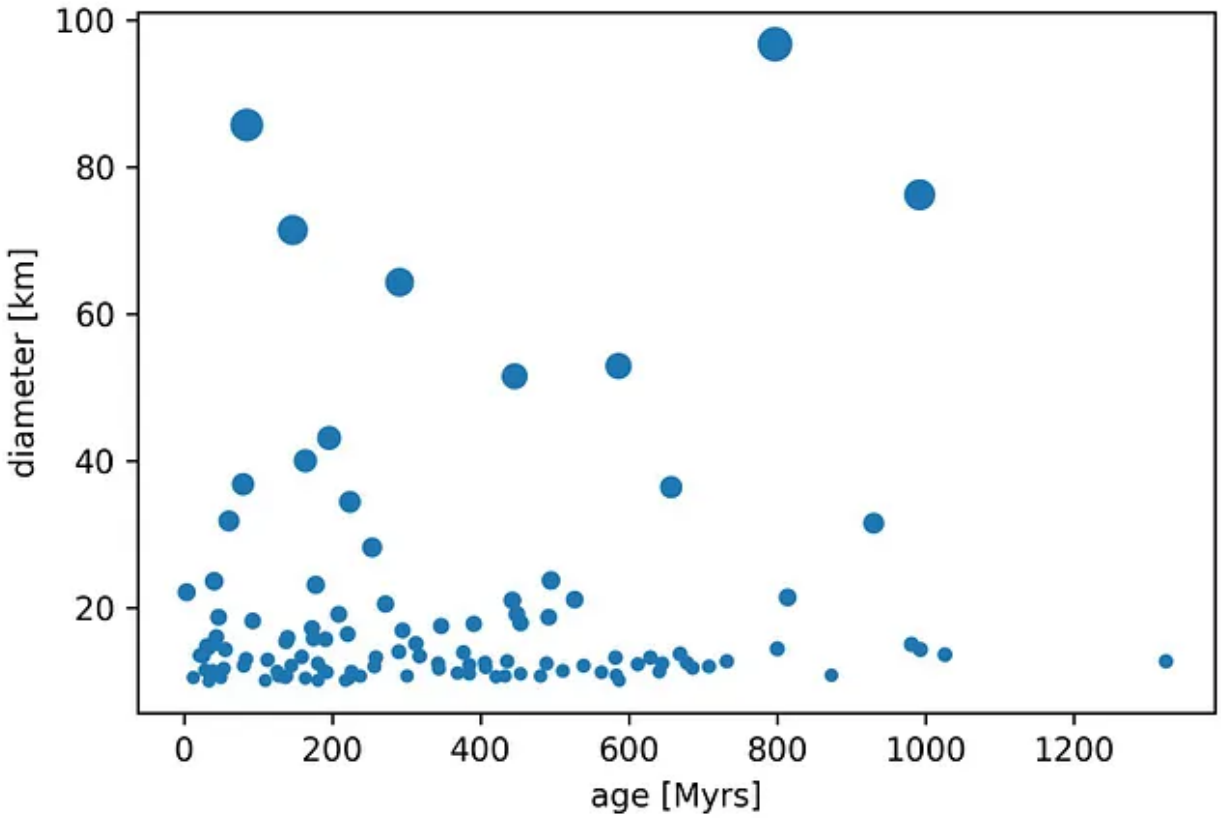
```
import pandas as pd
filename = 'lunarCraterAges'  #name of csv data file
```

```
df = pd.read_csv(filename + '.csv')  #load data as a pandas dataframe

df.head() #take a look at first 5 rows
```

	names	longitude	latitude	diameter	age
0	Copernicus	339.92969	9.63280	96.7	797
1	Tycho	348.71469	-43.25890	85.7	85
2	King	120.49220	4.93750	76.2	992
3	Jackson	196.68950	22.04000	71.4	147
4	Ohm	246.27397	18.28491	64.3	291

Then we'll extract the age and diameter values and plot them using matplotlib:

```
import matplotlib.pylab as plt

ages = df['age'].values   #get age values in an array
diameters = df['diameter'].values  #get diameter values in an array

plt.scatter(ages, diameters, s=diameters)
plt.xlabel('age [Myrs]')
plt.ylabel('diameter [km]')
plt.show()
```



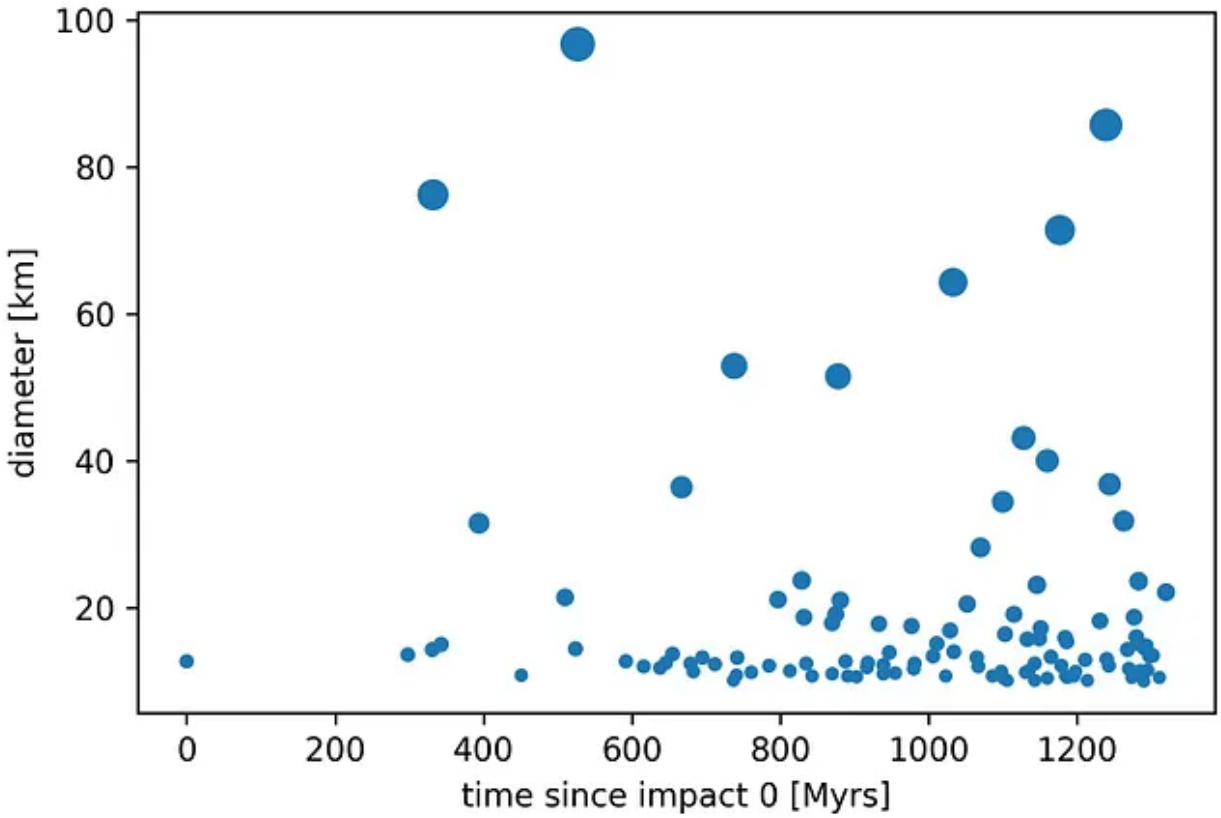
Crater diameter vs. age.

Open in app ↗

Search Medium

measurements into time values, starting at 0 for the first impact in our data:

```
times_myrs = max(ages) - ages  #measure time from 1st impact in data

plt.scatter(times_myrs, diameters, s=diameters)
plt.xlabel('time since impact 0 [Myrs]')
plt.ylabel('diameter [km]')
plt.show()
```



Crater diameters vs. time, measured forwards from the 1st impact. The size of the dots also indicates the crater diameters.

Notice that there are many, many more smaller craters than large ones (bigger asteroid impacts are much more rare). You can also see that the rate increases over time. Most of this is due to a selection effect (it's easier to date younger craters) but there is a statistically significant uptick around 290 million years ago. Interesting! This will also develop nicely as a piece of music since the tension will build over time and then resolve slightly at the very end.

The Design

Before you begin diving into the sonification itself you need to figure out what is the central, most interesting feature (or features) of the data. What aspects of the data do you want to communicate through sound? In other words, **what story are you going to tell?** The choices you make later on can either illuminate or obscure this story so it's important that you're clear on the goals of the sonification form the outset.

"I'm often asked for a library of tools that will automatically make attractive representations of any given data set. But if each set is different, the point of visualization [/sonification] is to

expose that fascinating aspect of the data and make it self-evident." — *Ben Fry [/Kelly Snook]*

In this case, the goal is pretty simple: we'd like to hear the rhythm of asteroid impacts over time. As a secondary feature, we'd also like to be able to distinguish the impacts that created different sized craters.

Now that that's settled, let's sketch out a sonification design. This stage involves making decisions about:

Mapping: Which data parameters are mapped to which auditory parameters? (Do crater sizes control the pitch of notes, or something else?)

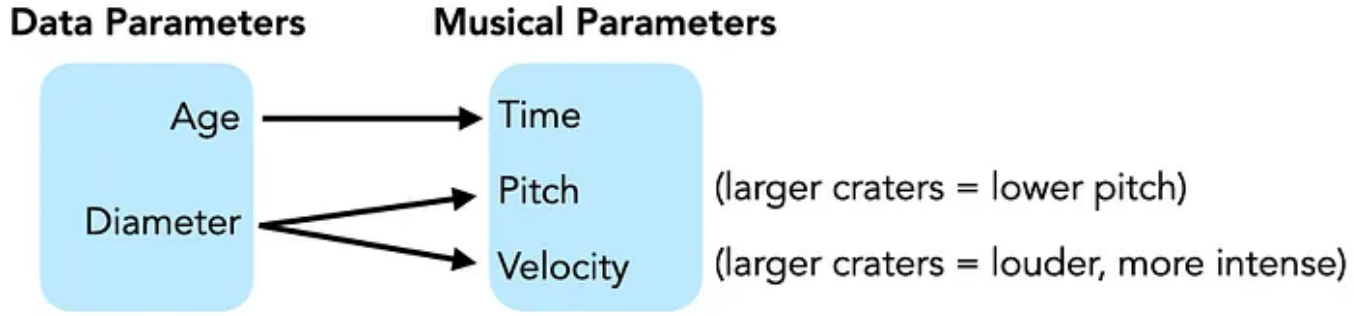
Polarity: What is the orientation of the mapping? (Do bigger craters produce higher or lower pitch notes?)

Range: What range of each musical parameter will you choose? (How long will the piece be? What is the lowest and highest note?)

Scaling (stretch/compress): Exactly how will numerical values be mapped from one range into another? (If a crater is twice as big should it be twice as loud? 4 times as loud? Something else?) Many audio parameters (such as pitch and volume) are not perceived linearly anyway so some non-linear stretching or compression is often helpful.

Quantization: Will any of the data be rounded, how? (Will crater impacts be forced to occur 'on beats'? Will pitches be rounded to notes from a certain scale?) More aggressive quantization may make it more musical but it also reduces the resolution.

Here is our basic sonification design, showing the mapping and polarity:



The mapping from data parameters to musical parameters and the associated polarity.

Crater ages will be mapped to time in the obvious way (happening forwards in time). Crater diameters will be mapped to musical pitch, with larger craters being lower pitched notes. You can make the opposite choice but I feel that this choice is more intuitive since larger objects/instruments can support lower notes. We will also use the crater diameters to control the note velocity (a combination of volume and intensity), with larger craters producing louder and more intense sounds (seems to make sense!). Try to identify the range, scaling, and quantization choices that are made as we implement the sonification design.

Mapping, Mapping, Mapping...

Mapping a set of numbers from one range to another is so common in sonification we might as well write a function that does just that for us. The following function will map input values (as an integer, float, or array) taken from one range, `min_value` to `max_value`, into a new range, `min_result` to `max_result`:

```
def map_value(value, min_value, max_value, min_result, max_result):
 ‘’’maps value (or array of values) from one range to another’’’

 result = min_result + (value – min_value)/(max_value – min_value)*
(max_result – min_result)
 return result
```

Note that this will return a float (or an array of floats) and we may need to round the result to an integer depending on what we'll be doing with it. As an exercise, modify the function so that it will automatically round the result and return an integer if both `min_result` and `max_result` are integers.

Compressing Time

Option 1: Set a Conversion Factor

Our first mapping task is to compress time from over a billion years to under 1 minute. One way to do this is to set a conversion factor. Our time data is currently measured in millions of years. We'd like to compress it and express the result in beats. We're mapping to beats, rather than seconds, because the MIDI file we'll create later accepts time as being measured in beats. If we set a conversion factor of 25 million years per beat, we can just divide our `times_myrs` array by this factor. With this choice, the last

note in the piece will land on beat 52.8. If we later set the tempo to be 60 beats per minute, then 1 beat will last for 1 second and the last note in our piece will play at exactly 52.8 seconds. This approach is useful if you want to add another element later that is synchronized to a certain unit of time. For example, playing a drum on every beat would indicate that 25 million years have passed, no matter what tempo you choose.

```
myrs_per_beat = 25  #conversion factor: Myrs for each beat of music

t_data = times_myrs/myrs_per_beat #compress impact times from Myrs to
beats
```

Option 2: Set a Duration

The other option is to choose the duration in advance (measured in beats) and then use our mapping function. It maps the impact times which currently run from `0` to `max(times_myrs)`, into a new range, `0` to `duration_beats`. With both options, we are not quantizing the time measurements at all so that we can hear the true rhythm of the impacts.

```
duration_beats = 52.8 #desired duration in beats (actually, onset of
last note)

t_data = map_value(times_myrs, 0, max(times_myrs), 0,duration_beats)
```

Next we'll set a variable for the tempo, `bpm=60`, and print the resulting duration in seconds:

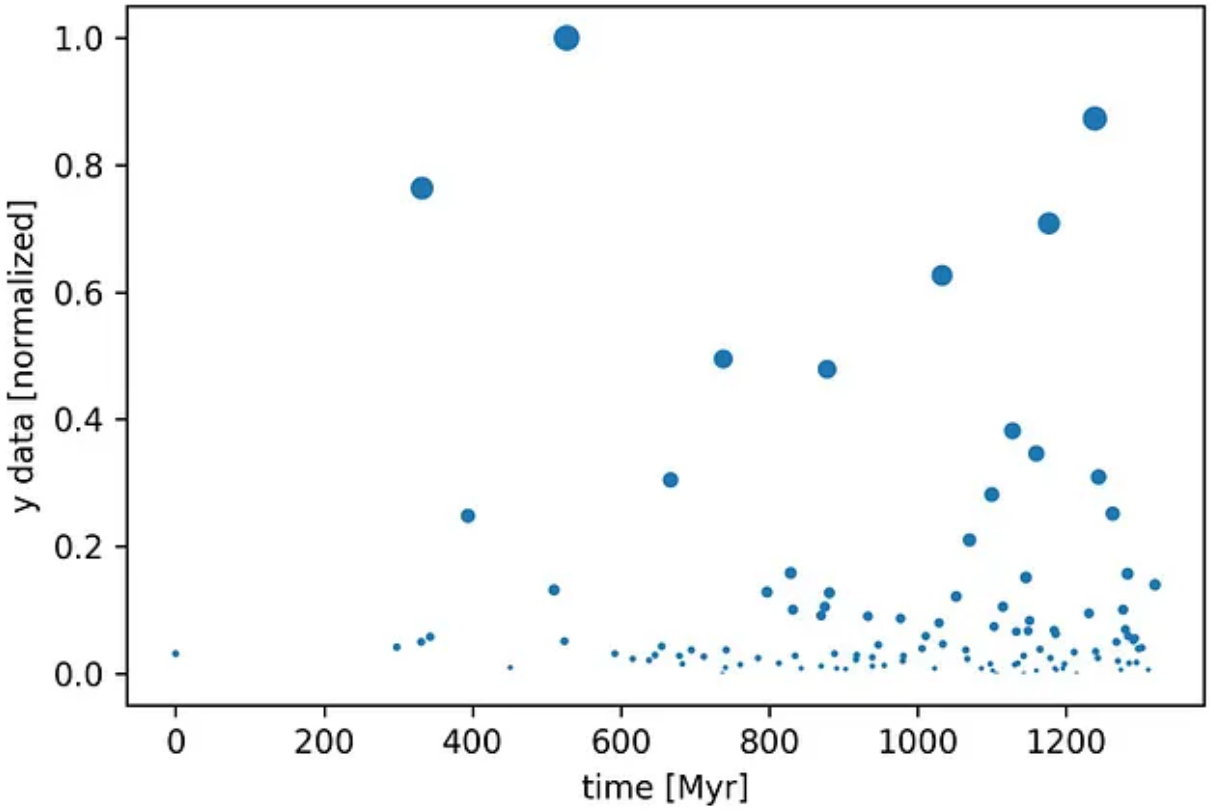
```
bpm = 60  #beats per minute, if bpm = 60, 1 beat = 1 sec

duration_sec = duration_beats*60/bpm #duration in seconds

print('Duration:', duration_sec, 'seconds')
>> Duration: 52.8 seconds
```

If you change the tempo, the piece will still be 52.8 beats but will be shorter or longer than 52.8 seconds. You'll be able to change the tempo later if you hear the result and think it's too fast or too slow.

Normalizing and Scaling The Data

Mapping the crater diameters to musical pitches takes a few steps. Let's start by normalizing the diameter data so that it runs from 0 to 1 and plotting the result:

```
y_data = map_value(diameters, min(diameters), max(diameters), 0, 1)

plt.scatter(times_myrs, y_data, s=50*y_data)
plt.xlabel('time [Myr]')
plt.ylabel('y data [normalized]')
plt.show()
```

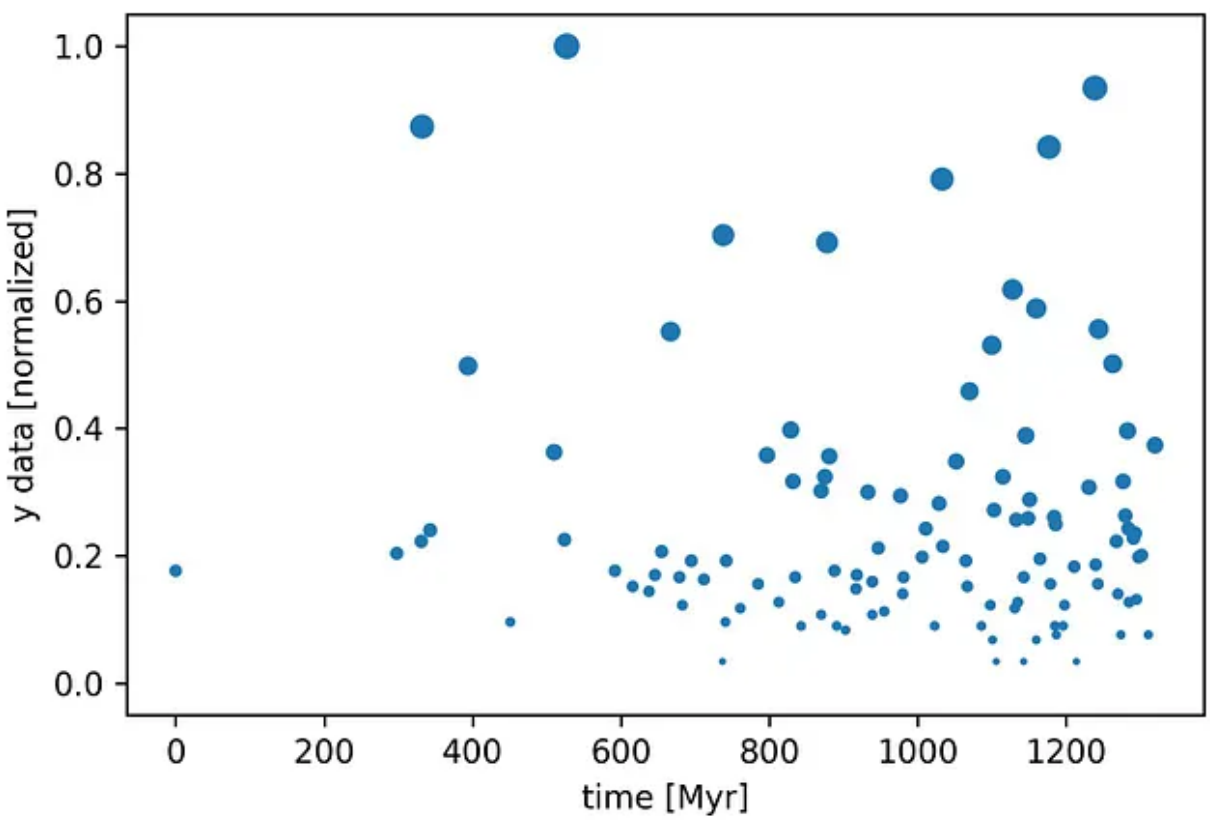


Normalized diameter data vs. time. The size of the dots indicates the normalized diameters.

We can see that the many small craters are bunched up at the bottom. If we map this data to a limited set of musical pitches, many of those will be mapped to the same note.

Maybe you want that (to communicate that most are of similar, small size) but we're going to do something different. We want to be able to hear the difference in crater size even for the small craters so we'll use an extra degree of freedom and use a non-linear scaling. Raising all normalized data values to the power of 0.5 (a square root) helps spread out the smaller craters, making it easier to hear their different sizes.

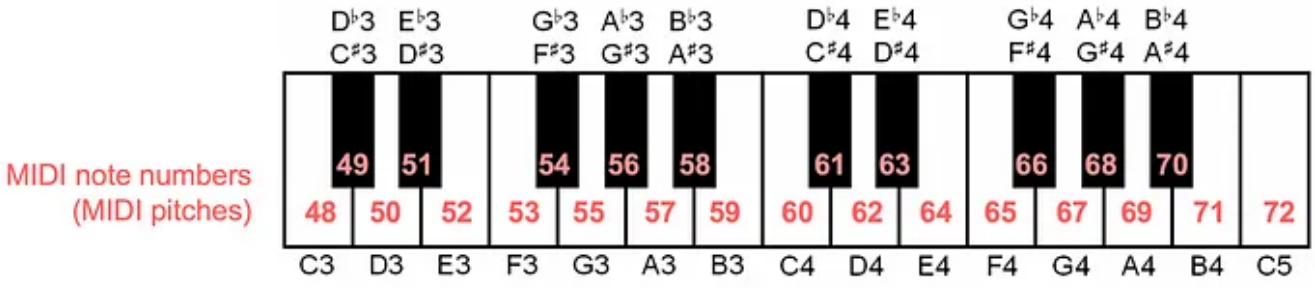
```
y_scale = 0.5  #lower than 1 to spread out more evenly

y_data = y_data**y_scale

plt.scatter(times_myrs, y_data, s=50*y_data)
plt.xlabel('time [Myr]')
plt.ylabel('y data [normalized]')
plt.show()
```



Normalized and scaled diameter data vs. time. The size of the dots indicates the normalized and scaled diameters.

Mapping Data to Notes

Now the creative part really begins. Since there's no inherent musical harmony in the data (as opposed to say, planetary orbits), we might as well choose any set of notes we want. We can think in terms of note names but we'll need to convert these to MIDI note numbers before saving the data. Each musical note (all of the notes on a piano plus more) corresponds to a certain MIDI note number which is an integer from 0–127. The typical range for a piano is 21–108 but other instruments have different ranges.



MIDI note numbers and piano note names, from Müller, FMP, Springer 2015.

The audiolazy library has some really convenient functions for converting between note names, midi note numbers, and also note frequencies if you need that too. The number after a note letter name indicates the octave.

```
from audiolazy import str2midi, midi2str

print(str2midi('C3'))
>> 48

print(midi2str(63))
>> 'D#4'
```

It's common to map data to a few octaves of a certain musical scale. Written out, 3 octaves of a C major scale would be:

```
note_names = ['C2','D2','E2','F2','G2','A2','B2',
              'C3','D3','E3','F3','G3','A3','B3',
              'C4','D4','E4','F4','G4','A4','B4']
```

In some sonifications, you might want to play it safer and use a pentatonic scale. Here's 3 octaves of C major pentatonic:

```
note_names = ['C2','D2','E2','G2','A2',
              'C3','D3','E3','G3','A3',
              'C4','D4','E4','G4','A4']
```

This scale is more consonant (less dissonant) than a typical major scale since there are no pairs of notes that differ by only a semi-tone (like B-C and E-F). It's very important that you pick a scale (and a range of notes) that helps convey the mood you're going for.

But we don't even have to limit ourselves to regular scales. In this example, it's not important that the listener can tell exactly how much bigger one crater is than another, just that they are different sizes. Given this freedom, we're going to use an irregular note set, a rich voicing of a Cmaj13#11 chord:

```
note_names = ['C1','C2','G2',
              'C3','E3','G3','A3','B3',
              'D4','E4','G4','A4','B4',
              'D5','E5','G5','A5','B5',
              'D6','E6','F#6','G6','A6']
```

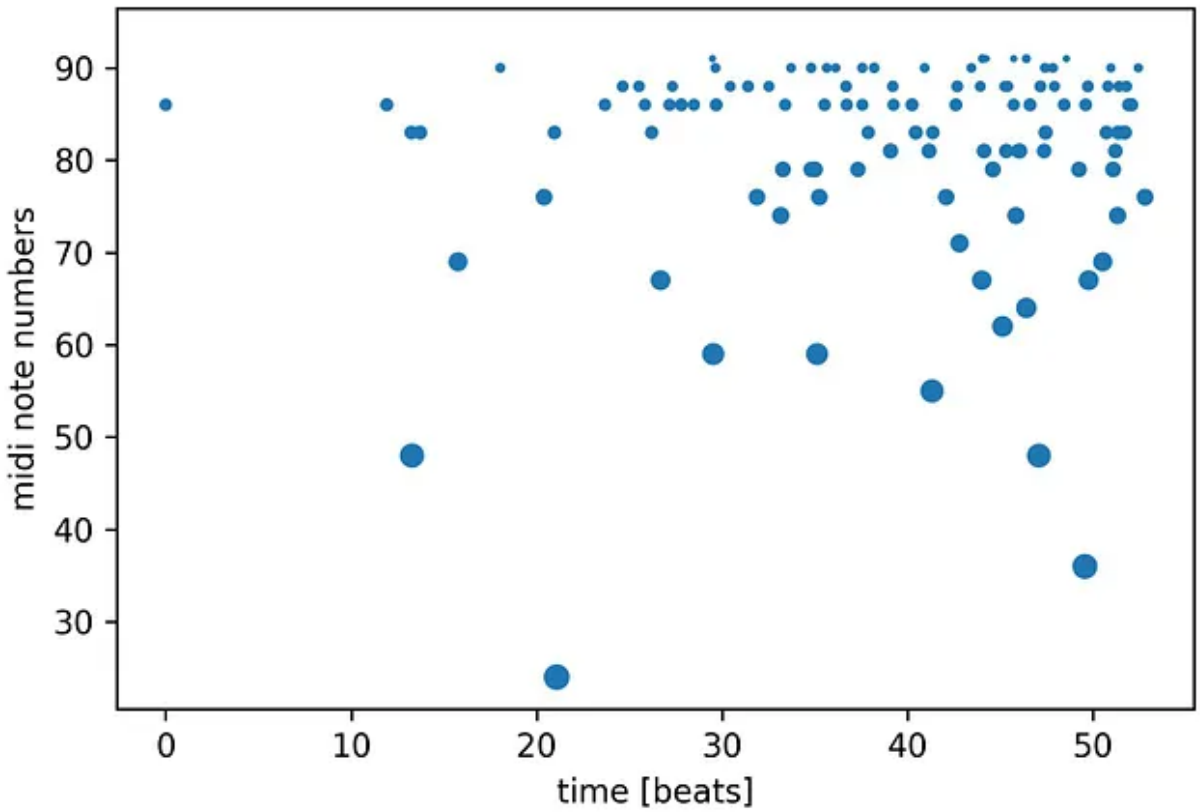
It's made up of notes from the C lydian scale, chosen for aesthetic and artistic reasons. There will only be one melodic musical part so a large range will not conflict with other instruments with overlapping ranges. Once we've chosen our note set, we make a list of the corresponding midi note numbers. We will also define a variable `n_notes` to store the number of notes we will map data to.

```
note_midis = [str2midi(n) for n in note_names]
n_notes = len(note_midis)
```

We can then use our `map_value` function to map our data, which currently runs from `0` to `1`, into a new range, `n_notes-1` to `0`. The result will be a list index. It will tell us which element of the `note_midis` each data point is mapped to. The smallest crater (`y_data=0`) is mapped to `22`, which is the index of the last element of the list (the highest note). The largest is mapped to `0`, the first element of the list (lowest note). Note that we need to round the result because list indices have to be integers. Then, we

use that index to access the midi note number and append it to a new list, `midi_data`. Plotting the result will show the higher notes toward the top.

```
midi_data = []
for i in range(len(y_data)):
    note_index = round(map_value(y_data[i], 0, 1, n_notes-1, 0))
    midi_data.append(note_midis[note_index])

plt.scatter(t_data, midi_data, s=50*y_data)
plt.xlabel('time [beats]')
plt.ylabel('midi note numbers')
plt.show()
```



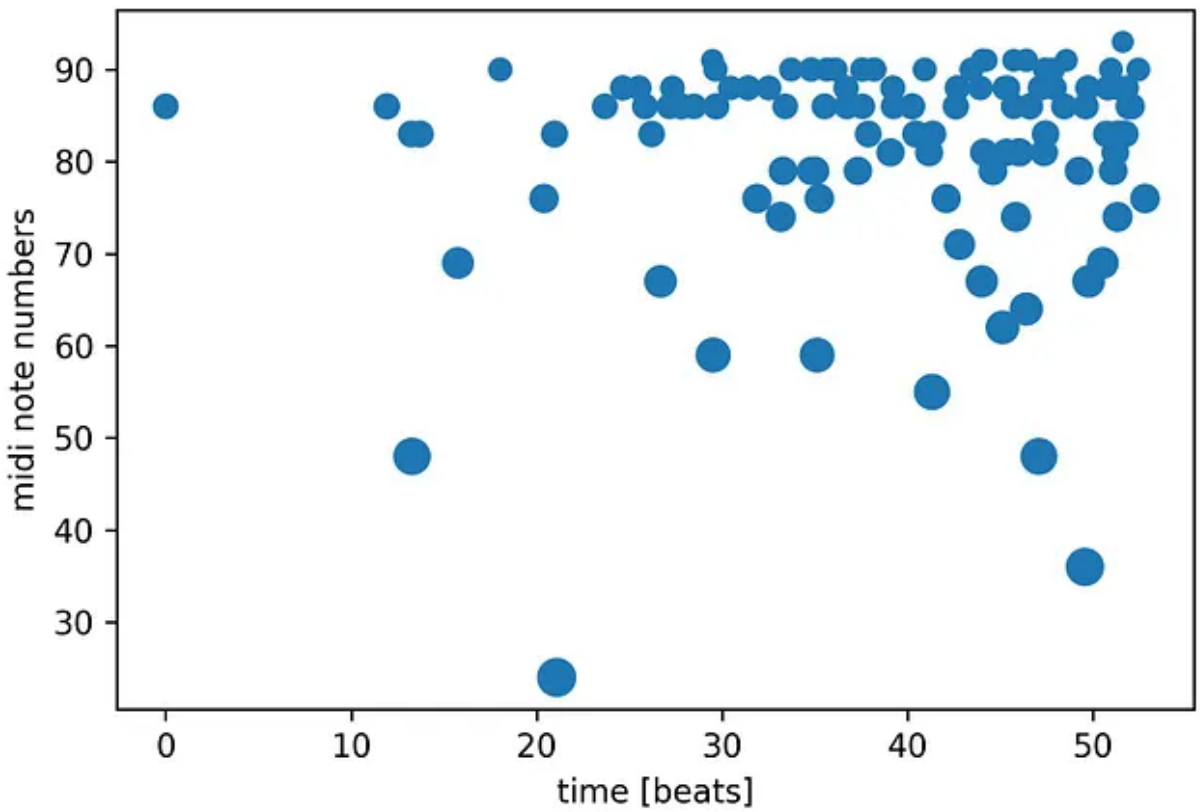
MIDI note numbers vs. time in beats. The size of the dots indicates the normalized and scaled diameters.

Mapping Data To Velocity

For each MIDI note we can also specify the note's velocity as an integer from 0–127. 0 will be faint and soft and 127 will be loud and intense. To keep the smallest craters still clearly audible we'll limit the velocity to the range 35–127. Then, we can use the

`map_value` function as long as we remember to round the result. The plot is the same as the previous plot except that the note velocities now control the dot size.

```
vel_min,vel_max = 35,127   #minimum and maximum note velocity

vel_data = []
for i in range(len(y_data)):
    note_velocity = round(map_value(y_data[i],0,1,vel_min, vel_max))
    vel_data.append(note_velocity)

plt.scatter(t_data, midi_data, s=vel_data)
plt.xlabel('time [beats]')
plt.ylabel('midi note numbers')
plt.show()
```



Same as the previous graph except that the size of the dot indicates the note velocity.

Writing a MIDI File

All that's left to do here is save the data into a MIDI file. The midiutil library makes this easy. You need to create a new MIDIFile object, set the tempo, and then add the notes

one at a time. For each note you can specify the track, channel, time (measured in beats), pitch (midi note number), volume (velocity), and duration (measured in beats). We'll just set all of the note durations to 2 beats but you could use data to control this too. For more complex projects you can also write notes on different tracks or midi channels. Then you need to create and write the data to a MIDI file which will have the extension `.mid`. It will appear in the same folder in which you're running the python script or notebook.

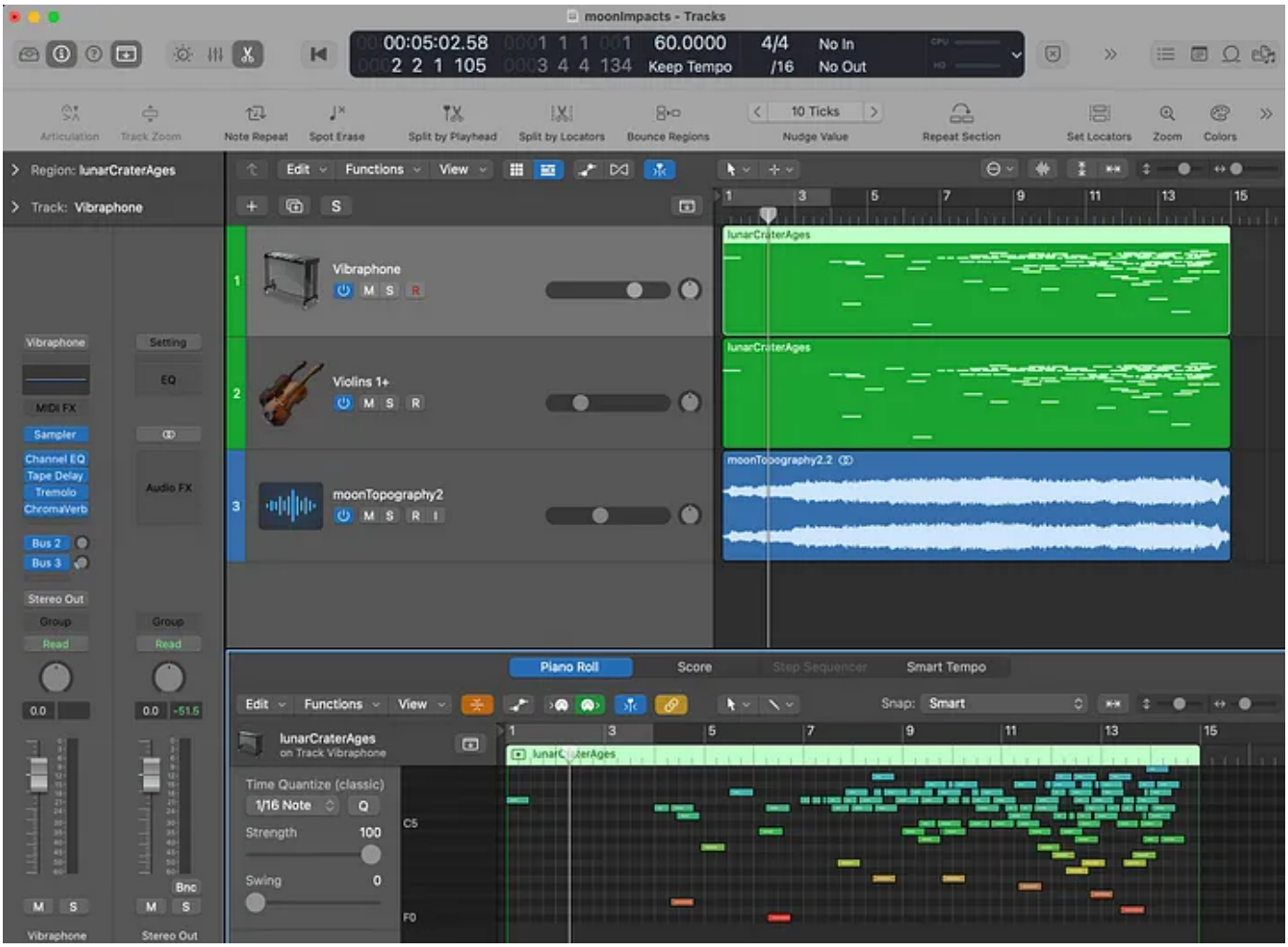
```
from midiutil import MIDIFile

#create midi file object, add tempo
my_midi_file = MIDIFile(1) #one track
my_midi_file.addTempo(track=0, time=0, tempo=bpm)

#add midi notes
for i in range(len(t_data)):
    my_midi_file.addNote(track=0, channel=0, time=t_data[I],
pitch=midi_data[i], volume=vel_data[I], duration=2)

#create and save the midi file itself
with open(filename + '.mid', "wb") as f:
    my_midi_file.writeFile(f)
```

Making Music

A MIDI file is like a score. It specifies which notes to play when, and with which velocity. To actually make sounds we need something that can perform the MIDI file. If you'd just like to hear the music quickly, you can use the pygame library. If you're writing python in a jupyter notebook you might want to have the `.stop()` command ready to go in the next cell. Otherwise the full piece will play.

```
import pygame

pygame.init()
pygame.mixer.music.load(filename + '.mid')
pygame.mixer.music.play()

#pygame.mixer.music.stop() #run this to stop, it's the only way!
```

This uses a default piano sound which isn't very exciting and it doesn't allow you to skip to certain parts. To really bring it to life you need to open the MIDI file in a digital audio workstation (DAW). I use Logic Pro but there are several to choose from and some are free and available online. Once you drag the MIDI file onto a new track in a DAW, you can choose an instrument sound from the DAW's library. You can also change the tempo, add effects, or add completely new layers to complete your musical world.



A screenshot of the Logic session with the sonified data being played on a vibraphone and with violins. The lower window shows each MIDI note, color-coded by note velocity.

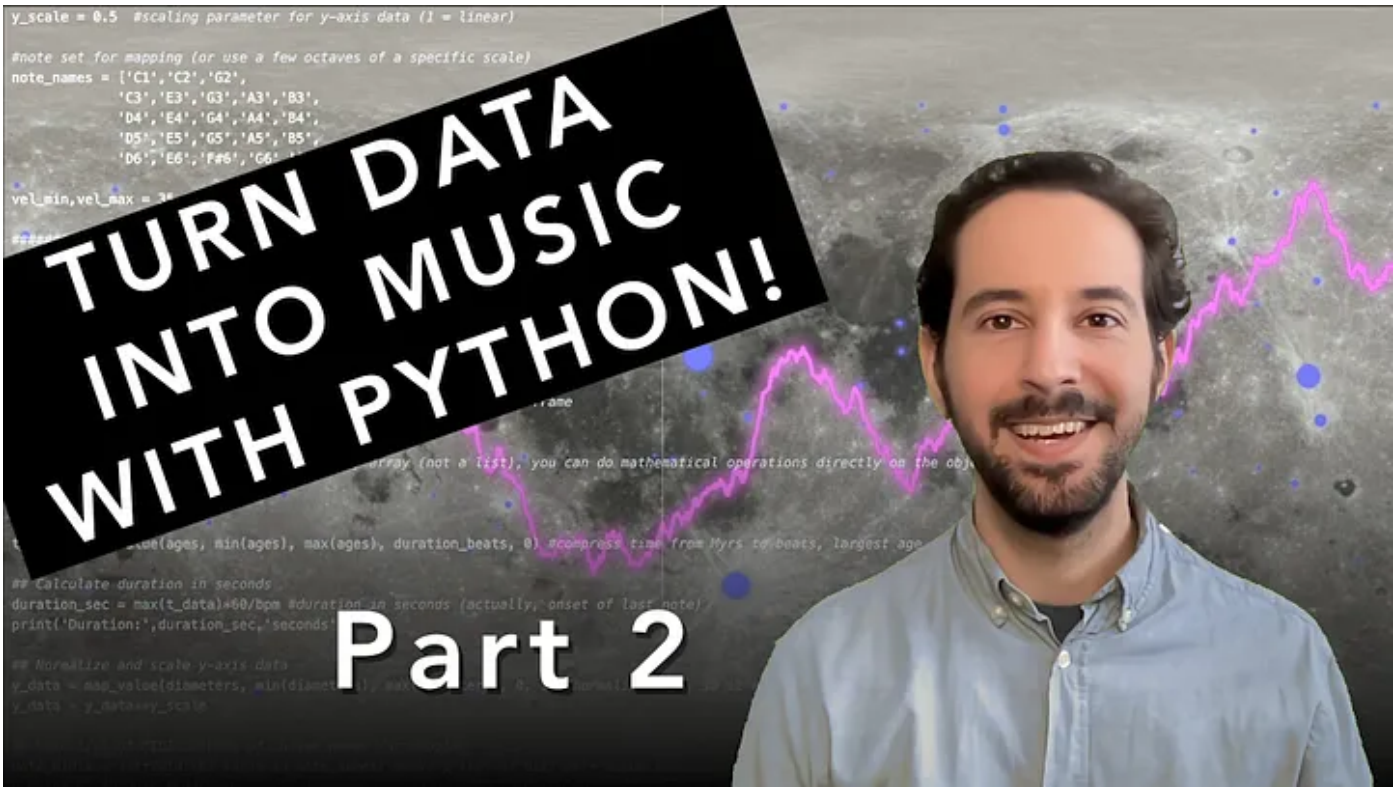
In this case, we'll play the MIDI data with vibraphone and violins, but the violins will be turned way down and drenched in reverb. The vibraphone has a strong attack (which seems fitting for asteroid impacts) and the verby violins will sustain a little more after the initial attack and decay. The lowest notes are outside of the normal range of a vibraphone and sound strange. In the published version we added a sine wave bass tone to help suppport these notes. To ground the piece and to keep it

evolving during the long gaps between early impacts, we also added a low drone sound. The drone was created by converting a topographic map of the Moon into an audio waveform. This is similar to spinning the Moon under a giant record needle as the needle glides from the North to South poles. This produced a harsh sound that morphed slowly due to the changing terrain at different latitudes on the Moon. I'll cover working with continuous data like this in a future post.

Conclusion

And that's it! Now that you've got the hang of it, try sonifying other data and see where it takes you.

You can do a lot by controlling the timing, pitch, velocity, and duration of notes but this is really just scratching the surface. In part 2 of the video series I show you how to control almost any musical parameter with data. Part 2 is only available on Gumroad.



Resources

Data Sonification Archive: browse a large and well-organized collection of sonification projects.

The Sonification Handbook: Download the definitive guide on sonification.

ICAD: The International Community for Auditory Display

SYSTEM Sounds: Explore more of my work.

Connect With Me

Personal Website — www.astromattrusso.com

Twitter — @astromattrusso

Gumroad Profile — https://astromattrusso.gumroad.com/

Sonification | Data Sonification | Python | Midi | Tutorial

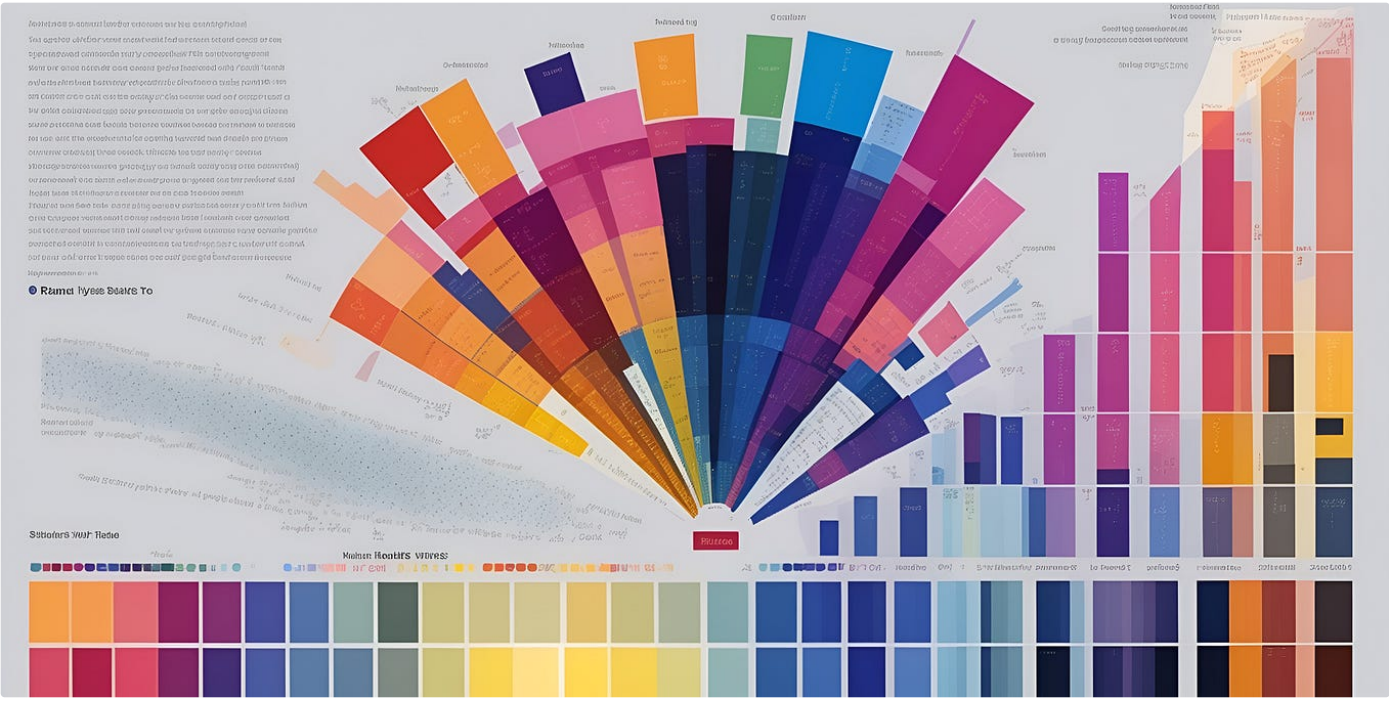
Follow

Written by Matt Russo

17 Followers

Astrophysicist, musician, and sonification specialist for NASA. Co-founder of www.system-sounds.com. Teaches physics at the University of Toronto.

Recommended from Medium



Bex T. in Towards AI

3 Best (Often Better) Alternatives To Histograms

Avoid the most dangerous pitfall of histograms

· 10 min read · 5 days ago

576 3

Mike Todasco

Deep Fakes For All: The Proliferation of AI Voice Cloning

Last week we hit a major milestone in AI voice cloning. Using Play.HT's new 2.0 model anyone can create a voice clone with just 30 seconds...

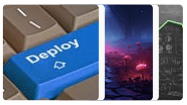
3 min read · Aug 18

14

Lists

Coding & Development

11 stories · 171 saves



Predictive Modeling w/ Python

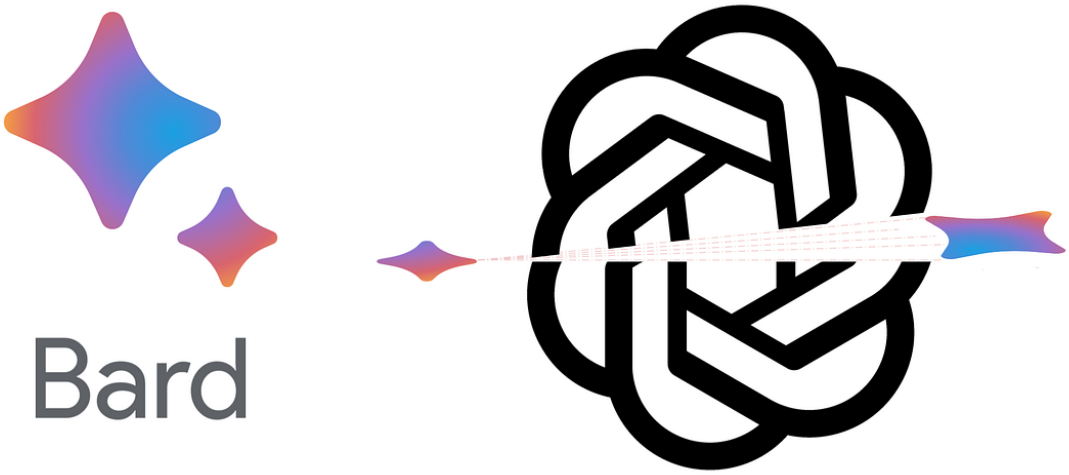
20 stories · 389 saves

Practical Guides to Machine Learning

10 stories · 430 saves

New_Reading_List

174 stories · 106 saves



AL Anany

The ChatGPT Hype Is Over—Now Watch How Google Will Kill ChatGPT.

It never happens instantly. The business game is longer than you know.

· 6 min read · Sep 1

6.9K 225

Unbecoming

10 Seconds That Ended My 20 Year Marriage

It's August in Northern Virginia, hot and humid. I still haven't showered from my morning trail run. I'm wearing my stay-at-home mom...

· 4 min read · Feb 16, 2022

63K 938

John MicMico in Artificial Intelligence in Plain English

Breaking Down the Mix: Using Python and Neural Networks to Separate Audio Tracks

Have you ever wondered how producers and DJs create those amazing remixes, where they isolate specific instruments from a song and turn...

6 min read · Mar 19

31

Daniel in d*classified

Exploring the Potential of AI for Music Emotion Recognition and Generation

Through analyzing instrumental audio content, Lim Xi Chen Terry has explored various models and techniques to classify audio clips based on...

8 min read · May 24

56

See more recommendations