



# PyTheory Is Awesome

MARCH 2026

8 min read • 1,827 words

---

Let me show you something.

```
from pytheory import Fretboard

fb = Fretboard.guitar()

# Put your fingers on the guitar: open, first fret, open, second, third, open.
chord = fb.fingering(0, 1, 0, 2, 3, 0)

print(chord.identify())
# 'C major'
```

You give it fret positions. It tells you what chord you're playing. That's it. That's the trick. And it works with **any tuning, any number of strings, any instrument** you can model as a fretboard.

I built this library called [PyTheory \(source\)](#), and I think it might be the thing I'm most proud of building. More than Requests. More than anything with millions of downloads and corporate adoption and conference talks. This little music theory library that almost nobody uses, that almost seems purposeless, that I built entirely for myself — it's the one.

Let me tell you why.

# The Fretboard

The chord detection thing is what hooked me initially. I'm sitting with a guitar, I find some shape on the neck that sounds interesting, and I genuinely don't know what I'm playing. This happens all the time. You stumble onto something beautiful and you have no idea what to call it.

So I built a way to ask.

```
from pytheory import Fretboard

fb = Fretboard.guitar()

# That weird shape you found at 2am
mystery = fb.fingering(0, 0, 0, 2, 1, 0)
print(mystery.identify())
# 'E minor 7th'

# Oh. That's why it sounded like that.
```

But here's where it gets interesting. The fretboard system isn't hardcoded to guitar. It's a model of any set of tuned strings. So once I had that abstraction right, instruments just fell out of it:

```

from pytheory import Fretboard

# Standard guitar, obviously.
guitar = Fretboard.guitar()

# But also...
oud = Fretboard.oud()
sitar = Fretboard.sitar()
shamisen = Fretboard.shamisen()
erhu = Fretboard.erhu()
balalaika = Fretboard.balalaika()
mandolin = Fretboard.mandolin()
banjo = Fretboard.banjo()
pedal_steel = Fretboard.pedal_steel()

# Even a piano is just a very wide fretboard with one string per key.
piano = Fretboard.keyboard()
midi_controller = Fretboard.keyboard(25, "C3")

```

Twenty-five instrument presets. Guitar alone has eight tunings — standard, drop D, open G, open D, DADGAD, open C, open E, all fifths. You can apply a capo to any of them. You can define custom tunings from scratch. The whole thing is built on the simple idea that a stringed instrument is a collection of tones, and frets are just semitone offsets from those tones.

```

# Drop D for those heavy riffs.
fb_drop_d = Fretboard.guitar("drop d")

# Capo on the second fret – now your G shapes sound like A.
fb_capo = Fretboard.guitar(capo=2)

# Open G tuning, the Keith Richards special.
fb_open_g = Fretboard.guitar("open g")

```

The architecture is simple. That's the point. When the abstraction is right, everything else becomes easy.

# The World's Music

Here's where I really started having fun.

Western music theory is one system among many. It's the one most of us learned (or didn't learn — more on that in a minute), but it's not the only way humans have organized sound. So PyTheory supports six musical systems:

```
from pytheory import TonedScale

# Western – the one you probably know.
c_major = TonedScale(tonic="C4")["major"]
print(c_major.note_names)
# ['C', 'D', 'E', 'F', 'G', 'A', 'B', 'C']

# Arabic maqam – the scales that make Middle Eastern music shimmer.
hijaz = TonedScale(tonic="Do4", system="arabic")["hijaz"]
print(hijaz.note_names)
# ['Do', 'Reb', 'Mi', 'Fa', 'Sol', 'Solb', 'Sib', 'Do']

# Indian raga (Hindustani thaats) – the melodic frameworks
# that have been refined for thousands of years.
bhairav = TonedScale(tonic="Sa4", system="indian")["bhairav"]
print(bhairav.note_names)
# ['Sa', 'komal Re', 'Ga', 'Ma', 'Pa', 'komal Dha', 'Ni', 'Sa']

# Japanese pentatonic – the sound of stillness.
hirajoshi = TonedScale(tonic="C4", system="japanese")["hirajoshi"]
print(hirajoshi.note_names)
# ['C', 'D', 'D#', 'G', 'G#', 'C']

# Blues – the sound of feeling.
blues = TonedScale(tonic="C4", system="blues")["blues"]
print(blues.note_names)
# ['C', 'D#', 'F', 'F#', 'G', 'A#', 'C']

# Javanese gamelan – pelog and slendro, tuning systems
# that don't map cleanly onto Western assumptions at all.
pelog = TonedScale(tonic="C4", system="gamelan")["pelog"]
```

Each system uses its own note names, its own terminology, its own way of thinking about intervals. Indian music doesn't say "C D E" — it says "Sa Re Ga." Arabic music uses "Do Re Mi" but with different interval patterns that create those characteristic quarter-tone inflections. The Japanese scales strip everything down to five notes and find infinite expression in the space between them.

Here's the thing that made me unreasonably happy about the architecture: once the system abstraction was right, adding a new musical tradition was trivial. I added Arabic maqam scales, Indian thaats, Japanese pentatonic, and Javanese gamelan in an afternoon. The data was the hard part — figuring out the correct interval patterns, the proper note names, the cultural context. The code just worked. That's what good abstraction does. It makes the hard part about understanding, not implementation.

## Harmony Falls Out

Once you have scales, harmony is just a matter of stacking thirds:

```

from pytheory import TonedScale, Key

# Build a C major scale.
c_major = TonedScale(tonic="C4")["major"]

# Harmonize the whole thing – triads built on every scale degree.
chords = c_major.harmonize()
print([c.identify() for c in chords])
# ['C major', 'D minor', 'E minor', 'F major',
#  'G major', 'A minor', 'B diminished']

# The I-V-vi-IV progression that powers half of pop music.
progression = c_major.progression("I", "V", "vi", "IV")
print([c.identify() for c in progression])
# ['C major', 'G major', 'A minor', 'F major']

# Seventh chords for when you want more color.
print(c_major.seventh(4).identify())
# 'G dominant 7th'

# Detect what key a bunch of notes belong to.
print(Key.detect("C", "E", "G", "A", "D"))
# <Key C major>

```

Roman numeral analysis. Chord tension scoring. Transposition. Voice leading. It all just falls out of the basic abstractions of tones, intervals, and scales. I didn't set out to build a comprehensive music theory library. I set out to answer the question "what chord am I playing?" and the rest emerged because the foundations were right.

## For Humans

You might notice something familiar about the API design. `Fretboard.guitar()`. `TonedScale(tonic="C4")["major"]`. `chord.identify()`. It reads like English. It does what you'd expect. You don't need to understand the implementation to use it.

This is the same design philosophy behind Requests. `requests.get(url)`. You know what that does before I explain it. The API is the documentation.

Music theory is notoriously gatekept. It's wrapped in Italian terminology and arcane notation and the implicit assumption that you already spent a decade in conservatory. This is a shame, because the underlying ideas are beautiful and not actually that complicated. A major scale is just a pattern of intervals. A chord is just a stack of notes from that scale. A progression is just a sequence of chords. The math is simple. The notation makes it seem harder than it is.

PyTheory is my attempt to make music theory as accessible as HTTP. You shouldn't need a music degree to ask "what chord is this?" any more than you should need to understand TCP/IP to make a web request. The complexity is real, but it belongs behind the interface, not in front of it.

```

# You don't need to know what "diatonic harmony" means
# to use this. You just need to be curious.

from pytheory import Chord, Tone

# "I have these notes. What am I playing?"
C = Tone.from_string("C4")
E = Tone.from_string("E4")
G = Tone.from_string("G4")

chord = Chord(tones=[C, E, G])
print(chord.identify())
# 'C major'

# "What does it sound like if I lower the middle note?"
Eb = Tone.from_string("D#4")
minor_chord = Chord(tones=[C, Eb, G])
print(minor_chord.identify())
# 'C minor'

# One semitone. That's the difference between major and minor.
# Between bright and dark. Between happy and sad, if you believe
# the oversimplification. The truth is more nuanced, but the
# distance is real: one semitone.

```

## Why This Matters to Me

I've been thinking about why this library means more to me than Requests, and I think it comes down to purity of intention.

Requests was built to solve a problem. A real, practical, millions-of-developers-have-this-problem problem. And I'm glad I built it. It made Python's HTTP story dramatically better, and that mattered. But Requests also came with everything that follows from millions of users: the issues, the expectations, the pressure, the opinions about what it should become, the weight of being infrastructure that other infrastructure depends on.

PyTheory was built for no reason other than joy. Nobody asked for it. There's no market for it. It's not going to become critical infrastructure for anything. It's a library for exploring music theory in Python, and the number of people who want to do that is small and beautiful and exactly right.

When I work on PyTheory, I'm not thinking about backwards compatibility or enterprise use cases or what Hacker News will think. I'm thinking about whether the API feels right. Whether the abstractions are beautiful. Whether the code expresses something true about how music works. It's programming as a form of play, which is what got me into programming in the first place.

There's something important in that. I think we lose track of it in the professional software world — the fact that code can be an act of pure curiosity. Not everything needs to scale. Not everything needs a business model. Sometimes you build something because the domain fascinates you and you want to understand it better, and the act of modeling it in code is itself the understanding.

Music theory is a system that humans built to describe patterns in vibrating air. Programming is a system that humans built to describe patterns in logic. PyTheory is what happens when you use one to explore the other, and the recursive beauty of that — using patterns to describe patterns — is genuinely thrilling to me.

## The Quiet Library

I sometimes call PyTheory my "quiet library." It sits there in my GitHub, not trending, not viral, not changing the world. A few people find it and send me nice messages. Someone used it to build a music education tool. Someone else used it to generate chord progressions for their band. Small things. Human-scale things.

And that's perfect. Not everything needs to be big. Some of the most meaningful work happens at small scale, where the relationship between builder and built is intimate and honest. Where you can hold the whole thing in your head and know that every line of code is there because you wanted it there, not because a product manager needed a feature for Q3.

If you play an instrument, or if you're curious about music theory, or if you just want to see what happens when you model beauty in Python — give PyTheory a look. It's `pip install pytheory`. It's for humans.

And if you find a weird chord shape at 2am and you don't know what to call it, now you have a way to ask.

---

**Update:** PyTheory also grew into a mini DAW — drums, synthesis, effects chains, automation, and WAV/MIDI export, all from the REPL. I wrote about that in [A Mini DAW in the Python REPL](#).

---

Generated from [kennethreitz.org](http://kennethreitz.org) • 2026