PyTheory: Breaking Through Five Years of Creative Block with AI

MARCH 2026

10 min read • 2,239 words

I started PyTheory in 2019 with a simple, almost naive ambition: make music theory feel as intuitive as `requests.get()`. Model tones, scales, and chords in Python with the same "for humans" philosophy I'd brought to HTTP. The initial prototype worked. You could create a tone, build a scale, hear frequencies. It was promising.

Then I got stuck. For five years.

Not stuck in the way where you abandon a project and forget about it. Stuck in the way where you open the repo every few months, stare at the code, feel the weight of everything it should be, and close your laptop. The vision was clear—a comprehensive music theory library spanning Western harmony, Indian ragas, Arabic maqam, Japanese scales, gamelan tuning systems. The implementation path was not.

> Creative blocks aren't absence of ideas—they're presence of too many simultaneous ideas creating decision paralysis. Every session I'd think "should I fix chord identification first, or build out the Indian system, or add fingering charts?" and end up doing nothing.

The breakthrough came from an unlikely source: the same kind of human-AI collaboration I'd been writing about philosophically. Claude didn't write PyTheory for me. It did something more important—it became a thinking partner who could hold the entire problem space in working memory while I made creative decisions. It sanity-checked my music theory for embarrassing errors, caught edge cases I'd missed, and helped me write documentation that actually teaches rather than just references.

The result is a library I'm genuinely proud of. Let me show you what it can do.

Music Theory for Humans

The core philosophy is the same one that drove Requests: match the human mental model, not the machine's. When a musician thinks "give me a C major scale," the code should read like that thought:

```
from pytheory import Tone, Scale, TonedScale, Key, Chord

# Think it, type it
c = Tone.from_string("C4")
c_major = TonedScale(tonic=c)["major"]

print(c_major.note_names)
# ['C', 'D', 'E', 'F', 'G', 'A', 'B']
```

But PyTheory goes deeper than note names. Every tone knows its frequency, its MIDI number, its position in the harmonic series:

```
a4 = Tone.from_string("A4")

print(a4.frequency)   # 440.0
print(a4.midi)        # 69
print(a4.overtones(5))
# [440.0, 880.0, 1320.0, 1760.0, 2200.0]
```

You can do arithmetic with tones the way musicians think about intervals—add seven semitones to get a perfect fifth, subtract to find the distance between notes:

```
c = Tone.from_string("C4")
g = c + 7  # Up a perfect fifth

print(g.full_name)  # G4
print(c.interval_to(g))  # perfect 5th
```

This is what programming as spiritual practice looks like in the wild. The API isn't clever—it's compassionate. It meets you where your understanding already lives.

Fingering Charts: Where Theory Meets Physical Instruments

This is where I spent the most time stuck, and where Claude's help was most transformative. Modeling abstract theory is one thing. Mapping that theory onto the physical reality of frets, strings, and human fingers is another problem entirely.

> A standard guitar has 6 strings × ~20 frets = 120 possible note positions. Finding which combination of positions produces a given chord while remaining physically playable is a constraint satisfaction problem that kept me paralyzed for years.

PyTheory ships with 25 instrument presets—from standard guitar to erhu to oud to balalaika—each with authentic tuning:

```
from pytheory import Fretboard

# Standard guitar tuning (E A D G B E)
guitar = Fretboard.guitar()

# Ukulele (G C E A)
uke = Fretboard.ukulele()

# Something more exotic
sitar = Fretboard.sitar()
bouzouki = Fretboard.bouzouki_irish()
```

The fingering system generates chord charts automatically. Want to see every way to play a C major chord on guitar?

```
from pytheory import CHARTS

c_major = CHARTS["western"]["C"]
fingerings = c_major.fingerings()

# The optimal fingering
print(c_major.tab())
# e |---0---|
# B |---1---|
# G |---0---|
# D |---2---|
# A |---3---|
# E |-------|
```

You can generate complete fingering charts for any instrument. The `charts_for_fretboard()` function builds charts for all 108 Western chord variations—12 root notes across 9 qualities—mapped to whatever tuning you hand it:

```python
from pytheory import charts_for_fretboard

# Chord charts for open D tuning
open_d = Fretboard.guitar_open_d()
charts = charts_for_fretboard(open_d)

# Now you have fingerings for every chord in open D
d_major = charts["D"]
print(d_major.tab())
```

Scale diagrams show you where to find every note of a scale on your fretboard:

```python
guitar = Fretboard.guitar()
c = Tone.from_string("C4")
c_minor_pentatonic = TonedScale(tonic=c)["minor pentatonic"]

print(guitar.scale_diagram(c_minor_pentatonic, frets=12))
```

And capo support shifts everything correctly—because if you're a guitarist, you think in terms of shapes relative to the capo, and the library should respect that:

```python
guitar = Fretboard.guitar()
capoed = guitar.capo(2)

# Now all fingerings are relative to fret 2
# Open strings ring at the capoed pitch
```

This whole system—the part that blocked me for half a decade—came together in weeks once I had a collaborator who could hold the combinatorial complexity in mind while I focused on what felt right as a musician. Claude caught tuning errors ("that's not standard mandolin tuning"), flagged impossible fingerings ("a human hand can't stretch 7 frets"), and helped me think through the ranking algorithm that determines which fingering is "best."

> This is what I mean when I talk about AI as accessibility device. My brain could hold the musical vision or the algorithmic complexity, but not both simultaneously. Claude held one while I worked the other. That's not replacement—it's amplification of capability.

Keys, Harmony, and Progressions

Real music isn't isolated chords—it's harmonic movement through a key. PyTheory models this the way a jazz musician thinks about it:

```
key = Key("C", "major")

# Diatonic triads
print(key.chords)
# [C major, D minor, E minor, F major,
#  G major, A minor, B diminished]

# Roman numeral progressions
progression = key.progression("I", "V", "vi", "IV")
# The most famous progression in pop music

# Nashville number system
nashville = key.nashville(1, 5, 6, 4)
# Same progression, Nashville studio shorthand
```

The library includes 14 named progressions from rock to jazz to classical:

```
from pytheory import PROGRESSIONS

# 12-bar blues
blues = key.progression(*PROGRESSIONS["12-bar blues"])

# Jazz ii-V-I
jazz = key.progression(*PROGRESSIONS["ii-V-I"])

# Pachelbel's Canon progression
canon = key.progression(*PROGRESSIONS["pachelbel"])
```

And because harmony is about relationships, PyTheory can analyze what's happening inside a chord—consonance, dissonance, tension, beat frequencies:

```
chord = Chord.from_name("Cmaj7")

print(chord.harmony)     # Consonance score
print(chord.dissonance)  # Plomp-Levelt roughness
print(chord.tension)
# {'score': 0.15, 'tritones': 0,
#  'minor_seconds': 1, 'has_dominant_function': False}

# Voice leading between chords
c_major = Chord.from_name("C")
f_major = Chord.from_name("F")
movement = c_major.voice_leading(f_major)
# Finds the smoothest path between voicings
```

This is music theory made computational without losing its soul. The numbers serve the music, not the other way around.

> The dissonance model uses Plomp and Levelt's critical bandwidth research from psychoacoustics—measuring actual perceptual roughness, not just interval classification. It's the difference between "theory says this is dissonant" and "human ears actually experience this as rough."

Beyond Western: Six Musical Systems

This is where PyTheory becomes something I haven't seen in any other library. Most music theory software assumes 12-tone equal temperament and Western note names. PyTheory models six distinct musical traditions:

```
from pytheory import systems

western = systems["western"]
indian = systems["indian"]
arabic = systems["arabic"]
japanese = systems["japanese"]
blues = systems["blues"]
gamelan = systems["gamelan"]
```

Each system has its own tone names, scale structures, and musical logic. Indian classical music doesn't think in terms of C-D-E—it thinks in Sa-Re-Ga-Ma-Pa-Dha-Ni:

```
sa = Tone.from_string("C4", system=indian)
bilawal = TonedScale(tonic=sa, system=indian)["bilawal"]

print(bilawal.note_names)
# ['Sa', 'Re', 'Ga', 'Ma', 'Pa', 'Dha', 'Ni']
```

Arabic maqam operates on a different set of modal principles, with quarter-tone inflections that 12-TET can only approximate:

```
do = Tone.from_string("C4", system=arabic)
hijaz = TonedScale(tonic=do, system=arabic)["hijaz"]
rast = TonedScale(tonic=do, system=arabic)["rast"]
```

Japanese traditional scales like Hirajoshi and In have a haunting pentatonic character that's completely distinct from Western pentatonics:

```
c = Tone.from_string("C4", system=japanese)
hirajoshi = TonedScale(tonic=c, system=japanese)["hirajoshi"]
iwato = TonedScale(tonic=c, system=japanese)["iwato"]
```

And Javanese gamelan uses its own tone names entirely—ji, ro, lu, pat, mo, nem, pi—with pelog and slendro tuning systems that don't map cleanly onto any Western framework:

```
ji = Tone.from_string("ji4", system=gamelan)
pelog = TonedScale(tonic=ji, system=gamelan)["pelog"]
```

Building this required getting the music theory right across traditions I'm not expert in. This is where Claude's ability to sanity-check against established musicological sources was invaluable. Every thaat, every maqam, every Japanese scale—verified against reference material, cross-checked for accuracy. The kind of comprehensive review that would have taken me months of research, compressed into focused collaborative sessions.

> There's a real responsibility here. Modeling another culture's musical system in code means getting it right or not doing it at all. These aren't exotic curiosities—they're living traditions with their own internal logic and centuries of theoretical development. PyTheory approximates them within 12-TET constraints, which is an honest limitation worth stating clearly.

Three Temperaments

Equal temperament isn't the only way to tune. PyTheory supports three historically significant tuning systems:

```
from pytheory import Tone

a = Tone.from_string("A4")

# Equal temperament (modern standard)
a.pitch(temperament="equal")      # 440.0 Hz

# Pythagorean (pure fifths, ancient Greek)
a.pitch(temperament="pythagorean")

# Quarter-comma meantone (pure thirds, Renaissance)
a.pitch(temperament="meantone")
```

You can even hear the difference:

```
from pytheory import play, Synth

c = Tone.from_string("C4")

# Listen in equal temperament
play(c, temperament="equal", synth=Synth.SINE)

# Now in Pythagorean—hear how the fifth rings differently
play(c, temperament="pythagorean", synth=Synth.SAW)
```

The CLI

Because sometimes you just want a quick answer without writing a script:

```
# Play a C major chord
$ pytheory play C4 E4 G4
  Playing: C4 E4 G4
  Synth:   sine
  Duration: 1000 ms

# Play a chord by name
$ pytheory play Cmaj7
  Playing: C major 7th (C4 E4 G4 B4)
  Synth:   sine
  Duration: 1000 ms

# What's in the key of D major?
$ pytheory key D major
  Key: D major
  Notes: D E F# G A B C#
  Chords: D  Em  F#m  G  A  Bm  C#dim

# What chord am I playing?
$ pytheory chord C E G
  Chord:     C major
  Tones:     C4 E4 G4
  Intervals: [4, 3]
  Harmony:   0.2381
  Dissonance: 2.7786
  Tension:   0.00 (tritones=0)

$ pytheory chord C E G Bb
  C dominant 7th (C7)

# Show me the fingering
$ pytheory fingering Am7
  e |---0---|
  B |---1---|
  G |---0---|
  D |---2---|
  A |---0---|
  E |-------|
# Detect the key from a set of notes
$ pytheory detect C D E F G A B
  C major (100% match)
```

What Breaking Through Actually Felt Like

I want to be honest about what happened here, because I think it matters for anyone sitting on their own stalled projects.

The creative block wasn't about not knowing music theory. It wasn't about not knowing Python. It was about the gap between vision and execution being so wide that every attempt to cross it felt overwhelming. PyTheory needed to be correct—music theory has right answers, and a library full of wrong enharmonic spellings or misidentified intervals would be worse than no library at all. That correctness requirement made every decision feel high-stakes, which made progress feel impossible.

> Perfectionism isn't about high standards. It's about the inability to tolerate the intermediate state where things are partially done and partially wrong. It's a debugging problem—your quality-control process is running before there's anything to control.

Claude changed the dynamic in three specific ways:

Sanity checking. I could write a chord identification algorithm and immediately ask "does this correctly identify a half-diminished seventh chord in third inversion?" Getting instant, knowledgeable feedback on music theory correctness removed the fear of shipping errors. Not because Claude is infallible, but because having a second perspective meant I wasn't solely responsible for catching every mistake.

Holding complexity. The fingering chart system requires thinking about music theory, instrument physics, combinatorial optimization, and user experience simultaneously. I can hold two of those in my head at once. Claude held the rest. We'd work through problems together—me providing the musical intuition about what feels right under the fingers, Claude tracking the algorithmic implications.

Documentation as thinking. Writing the docs with Claude forced me to articulate why each design decision made sense. "Why does `Tone + 7` return a tone but `Tone - Tone` return an integer?" Because addition models transposition (moving a note up) while subtraction models interval measurement (the distance between notes). These conversations crystallized design decisions I'd been waffling on for years.

The test suite went from sparse to 476 tests at 97% coverage. Not because Claude wrote tests I wouldn't have—but because having a collaborator made the tedious work of comprehensive testing feel like conversation rather than homework.

The Recursive Loop, Again

There's a deeper pattern here that connects to everything I've been writing about. PyTheory is a library that models how humans understand music. Building it required understanding how humans understand music. The tool and the understanding co-evolved—each design decision deepening my own grasp of theory, each theoretical insight suggesting better API design.

This is the recursive loop in miniature. The code shaped my thinking about music. My thinking about music shaped the code. And if PyTheory helps someone else understand music theory—a student, a hobbyist, a curious programmer—then my thinking, embedded in code, shapes their consciousness too.

That's the responsibility. That's why getting it right matters. And that's why having a thinking partner who could help me meet that responsibility was the difference between another five years of staring at the repo and actually shipping something.

```
# The smallest useful program in PyTheory
from pytheory import Key

key = Key("C", "major")
print(key.progression("I", "V", "vi", "IV"))

# Four chords. The foundation of a thousand songs.
# Theory made tangible. Complexity made approachable.
# For humans.
```

PyTheory is open source at github.com/kennethreitz/pytheory. Full documentation at pytheory.kennethreitz.org. Install with `uv pip install pytheory`. Contributions welcome, especially from musicians who know the traditions I've only approximated.

Related essays: Programming as Spiritual Practice · The Recursive Loop · The Mirror