Computer Music – Âm Nhạc Máy Tính

Nguyễn Quản Bá Hồng[*]

Ngày 23 tháng 4 năm 2025

Tóm tắt nội dung

This text is a part of the series *Some Topics in Advanced STEM & Beyond*:
URL: https://nqbh.github.io/advanced_STEM/.
Latest version:

- *Computer Music – Âm Nhạc Máy Tính.*
  PDF: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/computer_music/NQBH_computer_music.pdf.
  TEX: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/computer_music/NQBH_computer_music.tex.
- .
  PDF: URL: .pdf.
  TEX: URL: .tex.

Mục lục

1 Basic Computer Music

1.1 Renato Fabbri, Vilson Vieira de Silva Junior, Antônio Carlos Silvano Pessotti, Débora Cristina Corrêa, Osvaldo N. Oliveira Jr. **Musical Elements in Discrete-Time Representation of Sound**

[2 citations]

[*] A Scientist & Creative Artist Wannabe. E-mail: nguyenquanbahong@gmail.com. Bến Tre City, Việt Nam.

- Abstract. Representation of basic elements of music in terms of discrete audio signals is often used in software for musical creation & design. Nevertheless, there is no unified approach that relates these elements to discrete samples of digitized sound. In this article, each musical element is related by equations & algorithms to discrete-time samples of sounds, & each of these relations are implemented in scripts within a software toolbox, referred to as MASS (Music & Audio in Sample Sequences). Fundamental element, musical note with duration, volume, pitch, & timbre, is related quantitatively to characteristics of digital signal. Internal variations of a note, e.g. tremolos, vibratos, & spectral fluctuations, are also considered, which enables synthesis of notes inspired by real instruments & new sonorities. With this representation of notes, resources are provided for generation of higher scale musical structures, e.g. rhythmic meter, pitch intervals & cycles. This framework enables precise & trustful scientific experiments, data sonification & is useful for education & art. Efficacy of MASS is confirmed by synthesis of small musical pieces using basic notes, elaborated notes & notes in music, which reflects organization of toolbox & thus of this article. Possible to synthesize whole albums through collage of scripts & settings specified by user. With open source paradigm, toolbox can promptly scrutinized, expanded in co-authorship processes & used with freedom by musicians, engineers, & other interested parties. In fact, MASS has already been employed for diverse purposes which include music production, artistic presentations, psychoacoustic experiments & computer language diffusion where appeal of audiovisual artifacts is exploited for education.

- CCS Concepts: Applied computing → Sound & music computing. Computing methodologies → Modeling methodologies. General & reference → Surveys & overviews, Reference works.

- Additional Key Words & Phrases: music, acoustics, psychophysics, digital audio, signal processing.

- 1. Introduction. Music is usually defined as art whose medium is sound. Definition might also state: medium includes silences & temporal organization of structures, or music is also a cultural activity or product. In physics & in this document, sounds are longitudinal waves of mechanical pressure. Human auditory system perceives sounds in frequency bandwidth between 20Hz & 20kHz, with actual boundaries depending on person, climate conditions & sonic characteristics themselves. Since speed of sound ≈ 343.2 m/s, such frequency limits corresponds to wavelengths of $\frac{343.2}{20} \approx 17.6$ m & $\frac{343.2}{20000} \approx 17.16$ mm. Hearing involves stimuli in bones, stomach, ears, transfer functions of head & torso (thân mình), & processing by nervous system. Ear is a dedicated organ or appreciation of these waves, which decomposes them into their sinusoidal spectra & delivers to nervous system. Sinusoidal components are crucial to musical phenomena, as one can recognize in constitution of sounds of musical interest (e.g. harmonic sounds & noises, discussed in Sects. 2–3), & higher level musical structures (e.g. tunings, scales, & chords, Sect. 4) [55]

  Representation of sound can take many forms, from musical scores & texts in a phonetic language to electric analog signals & binary data. It includes sets of features e.g. wavelet or sinusoidal components. Although terms 'audio' & 'sound' are often used without distinction & 'audio' has many definitions which depend on context & author, audio most often means a representation of amplitude through time. In this sense, audio expresses sonic waves yield by synthesis or input by microphones, although these sources are not always neatly distinguishable e.g. as captured sounds are processed to generate new sonorities (âm thanh). Digital audio protocols often imply in quality loss (to achieve smaller files, ease storage & transfer) & are called *lossy* [47]. This is case e.g. of MP3 & Ogg Vorbis. Non-lossy representations of digital audio, called *lossless* protocols or formats, on other hand, assures perfect reconstruction of analog wave within any convenient precision. Standard paradigm of lossless audio consists of representing sound with samples equally spaced by a duration δ_s, & specifying amplitude of each sample by a fixed number of bits. This is linear Pulse Code Modulation (LPCM) representation of sound, herein referred to as PCM. A PCM audio format has 2 essential attributes: a sampling frequency $f_s = \frac{1}{\delta_s}$ (also called e.g. sampling rate or sample rate), which is number of samples used for representing a second of sound; & a bit depth, which is number of bits used for specifying amplitude of each sample. Fig. 1. Example of PCM audio: a sound wave is represented by 25 samples equally spaced in time where each sample has an amplitude specified with 4 bits. shows 25 samples of a PCM audio with a bit depth of 4, which yields $2^4 = 16$ possible values for amplitude of each sample & a total of $4 \cdot 25 = 100$ bits for representing whole sound.

  Fixed sampling frequency & bit depth yield quantization error or quantization noise. This noise diminishes as bit depth increases while greater sampling frequency allows higher frequencies to be represented. Nyquist theorem asserts: sampling frequency is twice maximum frequency: represented signal can contain [49]. Thus, for general musical purposes, suitable to use a sample rate of at least twice highest frequency heard by humans, i.e., $f_s \geq 2 \cdot 20$ kHz $= 40$ kHz. This is basic reason for adoption of sampling frequencies e.g. 44.1 kHz & 48 kHz, which are standards in Compact Disks (CD) & broadcast systems (radio & television), resp.

  Within this framework for representing sounds, musical notes can be characterized. Note often stands as 'fundamental unit' of musical structures (e.g. atoms in matter or cells in macroscopic organisms) &, in practice, it can unfold into sounds that uphold other approaches to music. This is of capital importance because science & scholastic artists widened traditional comprehension of music in 20th century to encompass discourse without explicit rhythm, melody or harmony. This is evident, e.g., in concrete, electronic, electroacoustic, & spectral musical styles. In 1990s, it became evident: popular (commercial) music had also incorporated sound amalgrams & abstract discursive arcs. [There are well known incidences of such characteristics in ethnic music, e.g. in Pygmy music, but western theory assimilated them only in last century [74].] Notes are also convenient for another reason: average listener – & a considerable part of specialists – presupposes rhythmic & pitch organization (made explicit in Sect. 4) as fundamental musical properties, & these are developed in traditional musical theory in terms of notes. Thereafter, in this article describe musical notes in PCM audio through equations & then indicate mechanisms for deriving higher level musical structures. Understand: this is not unique approach to mathematically express music in digital audio, but musical theory & practice suggest: this is a proper framework for understanding & making computer music, as should become patent in reminder of this text & is verifiable by usage of MASS toolbox. Hopefully, interested reader or programmer will be able to use this framework to synthesize music beyond traditional conceptualizations when intended.

This document provides a fundamental description of musical structures in discrete-time audio. Results include mathematical relations, usually in terms of musical characteristics & PCM samples, concise musical theory considerations, & their implementation as software routines both as very raw & straightforward algorithms & in context of rendering musical pieces. Despite general interests involved, there are only a few books & computer implementations that tackle subject directly. These mainly focus on computer implementations & way to mimic traditional instruments, with scattered mathematical formalisms for basic notations. Articles on topic appear to be lacking, to best of our knowledge, although advanced & specialized developments are often reported. A compilation of such works & their contributions is in Appendix G of [21]. Although current music software uses analytical descriptions presented here, there is no concise mathematical description of them, & far from trivial to achieve equations by analyzing available software implementations.

Accordingly, objectives of this paper:

1. Present a concise set of mathematical & algorithmic relations between basic musical elements & sequences of PCM audio samples.
2. Introduce a framework for sound & musical synthesis with control at sample level which entails potential uses in psychoacoustic experiments, data sonification & synthesis with extreme precision (recap in Sect. 5).
3. Provide a powerful theoretical framework which can be used to synthesize musical pieces & albums.
4. Provide approachability to developed framework [All analytic relations presented in this article are implemented as small scripts in public domain. They constitute MASS toolbox, available in an open source Git repository [9]. These routines are written in Python & make use of Numpy, which performs numerical routines efficiently (e.g. through LAPACK), but language & packages are by no means mandatory. Part of scripts has been ported to JavaScript (which favors their use in Web browsers e.g. Firefox & Chromium) & native Python [48, 56, 70]. These are all open technologies, published using licenses that grant permission for copying, distributing, modifying & usage in research, development, art & education. Hence, work presented here aims at being compliant with recommended practices for availability & validation & should ease co-authorship processes [43, 52].]
5. Provide a didactic (mang tính giáo huấn) presentation of content, which is highly multidisciplinary, involving signal processing, music, psychoacoustics & programming.

Reminder of this article is organized as follows: Sect. 2 characterizes basic musical note; Sect. 3 develops internal dynamics of musical notes; Sect. 4 tackles organization of musical notes into higher level musical structures [14, 41, 42, 54, 62, 72, 74, 76]. As these descriptions require knowledge on topics e.g. psychoacoustics, cultural traditions, & mathematical formalisms, text points to external complements as needed & presents methods, results, & discussions altogether. Sect. 5 is dedicated to final considerations & further work.

- 1.1. Additional material. 1 Supporting Information document [27] holds commented listings of all equations, figures, tables, & sects in this document & scripts in MASS toolbox. Another Supporting Information document [28] is a PDF version of code that implements equations & concepts in each sect [Toolbox contains a collection of Python scripts which

  - implements each of equations
  - render music & illustrate concepts
  - render each of figures used in this article.

  Documentation of toolbox consists of this article, Supporting Information documents & scripts themselves.]. Git repository [26] holds all PDF documents & Python scripts. Rendered musical pieces are referenced when convenient & linked directly through URLs, & constitute another component of framework. They are not very traditional, which facilitates understanding of specific techniques & extrapolation of note concept. There are MASS-based software packages [23, 25] & further musical pieces that are linked in Git repository.

- 1.2. Synonymy, polysemy & theoretical frames (disclaimer). Given: main topic of this article (expression of musical elements in PCM audio) is multidisciplinary & involves art, reader should be aware: much of vocabulary admits different choices of terms & defs. More specifically, often case where many words can express same concept & where 1 word can carry different meanings. This is a very deep issue which might receive a dedicated manuscript. Reader might need to read rest of this document to understand this small selection of synonymy & polysemy (đa nghĩa) in literature, but important to illustrate point before more dense sects:

  - a "note" can mean a pitch or an abstract construct with pitch & duration or a sound emitted from a musical instrument or a specific note in a score or a music.
  - Sampling rate is also called *sampling frequency* or *sample rate*.
  - A harmonic in a sound is most often a sinusoidal component which is in harmonic series of fundamental frequency. Many times, however, terms harmonic & component are not distinguished. A harmonic can also be a note performed in an instrument by preventing certain overtones (components).
  - Harmony can refer to chords or to note sets related to chords or even to "harmony" in a more general sense, as a kind of balance & consistency.

- ∗ A "tremolo" can mean different things: e.g. in a piano score, a tremolo is a fast alternation of 2 notes (pitches) while in computer music theory it is (most often) an oscillation of loudness.

  Strived to avoid nomenclature clashes & use of more terms than needed. Also, there are many theoretical standpoints for understanding musical phenomena, which is an evidence: most often there is not a single way to express or characterize musical structures. Therefore, in this article, adjectives e.g. "often", "commonly", & "frequently" are abundant & they would probably be even more numerous if wanted to be pedantically precise. Some of these issues are exposed when content is convenient, e.g. in 1st considerations of timbre.

  – Cố gắng tránh xung đột danh pháp & sử dụng nhiều thuật ngữ hơn mức cần thiết. Ngoài ra, có nhiều quan điểm lý thuyết để hiểu các hiện tượng âm nhạc, đây là bằng chứng: thường không có 1 cách duy nhất để diễn đạt hoặc mô tả các cấu trúc âm nhạc. Do đó, trong bài viết này, các tính từ như "thường xuyên", "thường xuyên", & "thường xuyên" rất nhiều & chúng có thể còn nhiều hơn nữa nếu muốn chính xác về mặt học thuật. Một số vấn đề này được nêu ra khi nội dung thuận tiện, ví dụ như trong những cân nhắc đầu tiên về âm sắc.

- 2. Characterization of musical note in discrete-time audio. In diverse artistic & theoretical contexts, music is conceived as constituted by fundamental units referred to as notes, "atoms" that constitute music itself [44, 72, 74]. In a cognitive perspective, notes are understood as discernible elements that facilitate & enrich transmission of information through music [41, 55]. Canonically, basic characteristics of a musical note are duration, loudness, pitch, & timbre (âm sắc) [41]. All relations described in this sect are implemented in file `src/sections/eqs2.1.py`. Musical pieces *5 sonic portraits & reduced-fi* are also available online to corroborate & illustrate concepts.

  - ○ 2.1. Duration. Sample frequency f_s is defined as number of samples in each sec of discrete-time signal. Let $T = \{t_i\}$ be an ordered set of real samples separated by $\delta_s = \frac{1}{f_s}$ secs ($f_s = 44.1$ kHz $\Rightarrow \delta_s = \frac{1}{44100} \approx 0.023$ ms). A musical note of duration Δ secs can be expressed as a sequence T^Δ with $\Lambda = \lfloor \Delta \cdot f_s \rfloor$ samples. I.e., integer part of multiplication is considered, & an error of $\le \delta_s$ missing secs is admitted, which is usually fine for musical purposes. Thus

  $$T^\Delta = \{t_i\}_{i=0}^{\lfloor \Delta f_s \rfloor - 1} = \{t_i\}_0^{\Lambda-1}.$$

  - ○ 2.2. Loudness. Loudness [Loudness & "volume" are often used indistinctly. In technical contexts, loudness is used for subjective perception of sound intensity while volume might be used for some measurement of loudness or to a change in intensity of signal by equipment. Accordingly, one can perceive a sound as loud or soft & change volume by turning a knob. Will use term loudness & avoid more ambiguous term volume.] is a perception of sonic intensity that depends on reverberation, spectrum, & other characteristics described in Sect. 3 [11]. One can achieve loudness variations through power of wave [11]:

  $$\mathrm{pow}(T) = \frac{\sum_{i=0}^{\Lambda-1} t_i^2}{\Lambda}.$$

  Final loudness is dependent on amplification of signal by speakers. Thus, what matters: relative power of a note in relation to the others around it, or power of a musical sect in relation to the rest. Differences in loudness are result of complex psychophysical phenomena but can often be reasoned about in terms of decibels, calculated directly from amplitudes through energy or power:

  $$V_{\mathrm{dB}} = 10 \log_{10} \frac{\mathrm{pow}(T')}{\mathrm{pow}(T)}.$$

  - ○ 2.3. Pitch.
  - ○ 2.4. Timbre.
  - ○ 2.5. Spectra of sampled sounds.
  - ○ 2.6. Basic note.
  - ○ 2.7. Spatialization: localization & reverberation.
  - ○ 2.8. Musical usages.

- 3. Variation in Basic Note.
  - ○ 3.1. Lookup table.
  - ○ 3.2. Incremental variations of frequency & intensity.
  - ○ 3.3. Application of digital filters.
  - ○ 3.4. Noise.
  - ○ 3.5. Tremolo & vibrato, AM & FM.
  - ○ 3.6. Musical usages.

- 4. Organization of notes in music.
  - ○ 4.1. Tuning, intervals, scales, & chords.
  - ○ 4.2. Atonal & tonal harmonies, harmonic expansion & modulation.

  - 4.3. Counterpoint.
  - 4.4. Rhythm.
  - 4.5. Repetition & variation: motifs & larger units.
  - 4.6. Directional structures.
  - 4.7. Cyclic structures.
  - 4.8. Serialism & post-serial techniques.
  - 4.9. Musical idiom?
  - 4.10. Musical usages.
- 5. Conclusions & Further Developments.

1.2 [HWR22]. Michael S. Horn, Melanie West, Cameron Roberts. Introduction to Digital Music with Python Programming: Learning Music with Code

[4 Amazon ratings]

- Amazon reviews. *Introduction to Digital Music with Python Programming* provides a foundation in music & code for beginner. It shows how coding empowers new forms of creative expression while simplifying & automating many of tedious aspects of production & composition.

  With help of online, interactive examples, this book covers fundamentals of rhythm, chord structure, & melodic composition alongside basics of digital production. Each new concept is anchored in a real-world musical example that will have you making beats in a matter of minutes.

  Music is also a great way to learn core programming concepts e.g. loops, variables, lists, & functions, *Introduction to Digital Music with Python Programming* is designed for beginners of all backgrounds, including high school students, undergraduates, & aspiring professionals, & requires no prev experience with music or code.

  A beginner's approach to digital music production focuses on key concepts, ensuring ease & progress in learning.

  Streamline your programming education by incorporating music, making complex core concepts easier to grasp & apply.

  Amplify your music creativity by generating unique beats with code in minutes, without needing advanced technical skills.

  A great book for learning Python programming & exploring digital music.

  This broad manual combines music theory & programming basics, providing interactive examples & real-world applications to help you compose & produce music from scratch.

  Perfect for aspiring musicians & programmers exploring music-code fusion.

- About Author. Michael S. Horn is Associate Prof of CS & Learning Sciences at Northwestern University in Evanston, Illinois, where he directs Tangible Interaction Design & Learning (TIDAL) Lab.

  Melanie West is a PhD student in Learning Sciences at Northwestern University & co-founder of Tiz Media Foundation, a nonprofit dedicated to empowering underrepresented youth through science, technology, engineering, & mathematics (STEM) programs.

  Cameron Roberts is a software developer & musician living in Chicago. He holds degrees from Northwestern University in Music Performance & CS.

- Foreword. When I was a kid growing up in Texas, I "learned" how to play viola. I put *learned* in quotes because it was really just a process of rote memorization – hours & hours of playing same songs over & over again. I learned how to read sheet music, bu only to extent that I knew note names & could translate them into grossest of physical movements. I never learned to read music as literature, to understand its deeper meaning, structure, or historical context. I never understood anything about music theory beyond being annoyed that I had to pay attention to accidentals in different keys. I never composed *anything*, not even informally scratching out a tune. I never developed habits of deep listening, of taking songs apart in my head & puzzling over how they were put together in 1st place. I never played just for fun. &, despite best intentions of parents & teachers, I never fell in love with music.

  Learning how to code was complete opposite experience for me. I was largely self-taught. Courses I took in school were electives (môn tự chọn) that I chose for myself. Teachers gave me important scaffolding at just right times, but it never felt forced. I spent hours working on games or other projects (probably when I should have been practicing viola). I drew artwork, planned out algorithms, & even synthesized my own rudimentary sound effects (hiệu ứng âm thanh thô sơ). I had no idea what I was doing, but that was liberating. No one was around to point out my mistakes or to show me ow to do things "right" way (at least, not until college). I learned how to figure things out for myself, & skills I picked up from those experiences are still relevant today. I fell in love with coding. [I was also fortunate to have grown up in a time & place where these activities were seen as socially acceptable for a person of my background & identity.]

  But I know many people whose stories are flipped 180 degrees. For them, music was so personally, socially, & culturally motivating that they couldn't get enough. They'd practice for hours & hours, not just for fun but for something much deeper.

For some it was an instrument like guitar that got them started. For others it was an app like GarageBand that gave them a playful entry point into musical ideas. To extent that they had coding experiences, those experiences ranged from uninspiring to off-putting (từ không hấp dẫn đến khó chịu). It's not that they necessarily hated coding, but it was something they saw as not being for them.

In foreword of his book, *Mindstorms: Children, Computers, & Powerful Ideas*, Seymour Papert wrote: "fell in love with gears" as a way of helping us imagine a future in which children (like me) would fall in love with computer programming, not for its own sake, but for creative worlds & powerful ideas that programming could open up. Part of what he was saying was: love & learning go hand in hand, & that computers could be an entry point into many creative & artistic domains e.g. mathematics & music. Coding can revitalize subjects that have become painfully rote in schools.

Process of developing TunePad over past several years has been a fascinating rediscovery of musical ideas for me. Code has given me a different kind of language for thinking about things like rhythm, chords, & harmony. I can experiment with composition unencumbered by my maladroit hands. Music has become something creative & alive in a way that it never was for me before. Music theory is no longer a thicket of confusing terminology & instead has become a fascinating world of mathematical beauty that structures creative process.

– Quá trình phát triển TunePad trong nhiều năm qua là 1 sự khám phá lại đầy hấp dẫn đối với tôi về các ý tưởng âm nhạc. Mã đã cho tôi 1 loại ngôn ngữ khác để suy nghĩ về những thứ như nhịp điệu, hợp âm, & sự hòa âm. Tôi có thể thử nghiệm sáng tác mà không bị cản trở bởi đôi tay vụng về của mình. Âm nhạc đã trở thành 1 thứ gì đó sáng tạo & sống động theo cách mà trước đây tôi chưa từng có. Lý thuyết âm nhạc không còn là 1 mớ thuật ngữ khó hiểu & thay vào đó đã trở thành 1 thế giới hấp dẫn của vẻ đẹp toán học cấu trúc nên quá trình sáng tạo.

Melanie, Cameron, & I hope: this book gives a similarly joyful learning experience with music & code. Hope: feel empowered to explore algorithmic & mathematical beauty of music. Hope: discover, as we have: music & code reinforce one another in surprising & powerful ways that open new creative opportunities for you. Hope, regardless of your starting point – as a coder, as a musician, as neither, as both – will discover something new about yourself & what you can become.

- 1. Why music & coding? This book is designed for people who *love* music & are interested in intersection of music & coding. Maybe you're an aspiring musician or music producer who wants to know more about coding & what it can do. Or maybe already know a little about coding, & want to expand your creative musical horizon. Or maybe a total beginner in both. Regardless of your starting point, this book is designed for you to learn about music & coding as mutually reinforcing skills. Code gives us an elegant language to think about musical ideas, & music gives us a context within which code makes sense & is immediately useful. Together they form a powerful new way to create music that will be interconnected with digital production tools of future.

  More & more code will be used to produce music, to compose music, & even to perform music for live audiences. Digital production tools e.g. Logic, Reason, Pro Tools, FL Studio, & Ableton Live are complex software applications created with *millions* of lines of code written by huge teams of software engineers. With all of these tools can write code to create custom plugins & effects. Beyond production tools, live coding is an emerging from of musical performance art in which Information Age DJs write computer code to generate music in real time for live audiences.

  In other ways, still on cusp of a radical transformation in way use code to create music. History of innovation in music has always been entwined with innovations in technology. Whether talking about Franz Liszt in 19th century, who pioneered persona of modern music virtuoso based on technological breakthroughs of piano [Fans were so infatuated with Liszt's piano "rockstar" status that they fought over his silk handkerchiefs & velvet gloves at his performances.], or DJ Kool Herc in 20th century, who pioneered hip-hop with 2 turntables & a crate full of funk records in Bronx, technologies have created new opportunities for musical expression that have challenged status quo & given birth to new genres. Don't have Franz Liszt or DJ Kool Herc of coding yet, but it's only a matter of time before coding virtuosos of tomorrow expand boundaries of what's possible in musical composition, production, & performance.

  - 1.1. What is Python? In this book learn how to create your own digital music using a computer programming language called *Python*. If not familiar with programming languages, Python is a general-purpose language 1st released in 1990s that is now 1 of most widely used languages in world. Python is designed to be easy to read & write, which makes it a popular choice for beginners. Also fully featured & powerful, making it a good choice for professionals working in fields as diverse as DS, web development, arts, & video game development. Because Python has been around for decades, it runs on every major computer OS. Examples in this book even use a version of Python that runs directly inside of your web browser without need for any special software installation.

    Unlike many other common beginner programming languages, Python is "text-based", i.e., type code into an editor instead of dragging code blocks on computer screen. This makes Python a little harder to learn than other beginner languages, but it also greatly expands what you can do. By time yet through this book should feel comfortable writing short Python programs & have conceptual tools need to explore more on your own.

  - 1.2. What this book is *not*. Before get into a concrete example of what you can do with a little bit of code, just a quick note about what this book is *not*. This book is not a comprehensive guide to Python programming. There are many excellent books & tutorials designed for beginners, several of which are free. [Recommend `https://www.w3schools.com/python/`.]

    This book is also not a comprehensive guide to music theory or Western music notation. Get into core ideas behind rhythm, harmony, melody, & composition, but there are, again, many other resources available for beginners who want to go deeper. What offering is a different approach that combines learning music with learning code in equal measure.

- 1.3. What this book *is*. What will do is give an intuitive understanding of fundamental concepts behind both music & coding. Code & music are highly technical skills, full of arcane symbols & terminology, seem almost designed to intimidate beginners. In this book put core concepts to use immediately to start making music. Get to play with ideas at your own pace & get instant feedback as bring ideas to life. Skip most of technical jargon & minutiae for now – can come later. Instead, focus on developing your confidence & understanding. Importantly, skills, tools, & ways of thinking introduce in this book will be broadly applicable in many other areas as well. Working in Python code, but core structures of variables, functions, loops, conditional logical, & classes are same across many programming languages including JavaScript, Java, C, C++, & C#. After learn 1 programming language, each additional language is that much easier to pick up.
- 1.4. TunePad & EarSketch. This book uses 2 free online platforms that combine music & Python coding. 1st, called TunePad https://tunepad.com, was developed by a team of researchers at Northwestern University in Chicago. TunePad lets create short musical loops that you can layer together using a simple digital audio workstation (DAW) interface. 2nd platform, called EarSketch https://earsketch.gatech.edu, was created by researchers at Georgia Tech in Atlanta. EarSketch uses Python code to arrange samples & loops into full-length compositions. Both platforms are browser-based apps, so all need to get started is a computer (tablets or Chromebooks are fine), an internet connection, & a web browser like Chrome or Firefox. External speakers or headphones are also nice but not required. Both platforms have been around for years & have been used by many thousands of students from middle school all way up to college & beyond. TunePad & EarSketch are designed primarily as learning platforms, but there are easy ways to export your work to professional production software if want to go further.
- 1.5. A quick example. A quick example of what coding in Python looks like. This program runs in TunePad to create a simple beat pattern, variants of which have been used in literally thousands of songs e.g. *Blinding Lights* by The Weeknd & *Roses* by SAINt JHN.

  ```
  playNote(1) # play a kick drum sound
  playNote(2) # play a snare drum sound
  playNote(1)
  playNote(2)
  rewind(4)   # rewind 4 beats
  for i in range(4):
      rest(0.5)
      playNote(4, beats = 0.5) # play hat for a half beat
  ```

  These 8 lines of Python code tell TunePad to play a pattern of kick drums, snare drums, & high-hats. Most of lines are *playNote* instructions, & those instructions tell TunePad to play musical sounds indicated by numbers inside of parentheses. This example also includes sth called a *loop*. Loop is an easy way to repeat a set of actions over & over again. In this case, loop tells Python to repeat lines 7 & 8 4 times in a row. Screenshot Fig. 1.1: A TunePad program to play a simple rock beat. shows what this looks like in TunePad. Can try out example for yourself with this link: https://tunepad.com/examples/roses.
- 1.6. 5 reasons to learn code. Now seen a brief example of what can do with a few lines of Python code, here are top 5 reasons to get started with programming & music if still in doubts.
  - 1.6.1. Reason 1: Like it or note, music is already defined by code. Looking across modern musical landscape, clear: music is already defined by code. 1 of biggest common factors of almost all modern music from any popular genre: *everything* is edited, if not created entirely, with sophisticated computer software. Hard to overstate how profoundly such software has shaped sound of music in 21st century. Relatively inexpensive DAW applications & myriad ubiquitous plugins that work across platforms have had a disruptive & democratizing effect across music industry. Think about effects plugins like autotune, reverb, or ability to change pitch of a sample without changing tempo. These effects are all generated with sophisticated software. Production studios size of small offices containing hundreds of thousands of dollars' worth of equipment now fit on screen of a laptop computer available to any aspiring producer with passion, a WiFi connection, & a small budget. Reasons behind shift to digital production tools are obvious. Computers have gotten to a point where they are cheap enough, fast enough, & capacious enough to do real-time audio editing. Can convert sound waves into editable digital information with microsecond precision & then hear effects of our changes in real time. These DAWs didn't just appear out of nowhere. They were constructed by huge teams of software engineers writing code – millions of lines of it. E.g., TunePad was created with > 1.5 million lines of code written in over a dozen computer languages e.g. Python, HTML, JavaScript, CSS, & Dart. Regardless of how feel about digital nature of modern music, not going away. Learning to code will help understand a little more about how all of this works under hood. More to point, it's increasingly common for producers to write their own code to manipulate sound. E.g., in Logic, can write JavaScript code to process incoming MIDI (Musical Instrument Digital Interface) data to do things like create custom arpeggiators. Learning to code can give you more control & help expand your creative potential Fig. 1.2: Typical DAW software.
  - 1.6.2. Reason 2: Code is a powerful way to make music. Don't always think about it this way, but music is *algorithmic* in nature – it's full of mathematical relationships, logical structure, & recursive patterns. Beauty of Baroque fugue is in part a reflection of beauty of mathematical & computational ideas behind music. Call Bach a genius not just because his music is so compelling, but also because he was able to hold complex algorithms in his mind & then transcribe them to paper using representation system called Western music notation. I.e., music notation is a language for recording output of composition process, but not a language for capturing algorithmic nature of composition process itself.

Code, on other hand, is a language specifically designed to capture mathematical relationships, logical structure, & recursive patterns. E.g., take stuttered hi-hat patterns that are 1 of defining characteristics of trap music. Here are a few lines of Python code that generate randomized hi-hat stutters that can bring an otherwise conventional beat to life with sparking energy.

```
for _ in range(16):
    if randint(6) > 1: # roll die for a random number
        playNote(4, beats=0.5) # play an 8th note
    else:
        playNote(4, beats=0.25) # or play 16th notes
        playNote(4, beats=0.25)
```

Or, as another example, here's a 2-line Python program that plays a snare drum riser effect common in house, EDM, or pop music. Often hear this technique right before beat drops. This code uses a decay function so that each successive note is a little shorter resulting in a gradual acceleration effect.

```
for i in range(50): # play 50 snares
    playNote(2, beats = pow(2, -0.09 * i))
```

What's cool about these effects: they're *parametrized.* Because code describes algorithms to generate music, & not music itself, i.e., can create infinite variation by adjusting numbers involved. E.g., in trap hi-hat code, can easily play around with how frequently stuttered hats are inserted into pattern by increasing or decreasing 1 number. Can think of code as sth like a power drill; can swap out different bits to make holes of different sizes. Drill bits are like parameters that change what tool does in each specific instance. In same way, algorithms are vastly more general-purpose tools that can accomplish myriad tasks by changing input parameters.

Creating a snare drum riser with code is obviously a very different kind of thing than picking up 2 drumsticks & banging out a pattern on a real drum. &, to be clear, not advocating for code to replace learning how to perform with live musical instruments. But, code can be another tool in your musical repertoire for generating repetitive patterns, exploring mathematical ideas, or playing sequences that are too fast or intricate to play by hand.

– Tạo 1 bộ phận nâng cao trống snare bằng mã rõ ràng là 1 việc rất khác so với việc nhặt 2 dùi trống & đánh 1 mẫu trên 1 chiếc trống thật. &, nói rõ hơn, không ủng hộ việc sử dụng mã để thay thế việc học cách biểu diễn với các nhạc cụ sống. Nhưng mã có thể là 1 công cụ khác trong tiết mục âm nhạc của bạn để tạo ra các mẫu lặp lại, khám phá các ý tưởng toán học hoặc chơi các chuỗi quá nhanh hoặc phức tạp để chơi bằng tay.

∗ 1.6.3. Reason 3: Code lets you build your own musical toolkit. Becoming a professional in any field is about developing expertise with tools – acquiring equipment & knowing how to use it. Clearly, this is true in music industry, but also true in software. Professional software engineers acquire specialized equipment & software packages. Develop expertise in a range of programming languages & technical frameworks. But, they also build their own specialized tools that they use across projects. In this book, show how to build up your own library of Python functions. Can think of functions as specialized tools that you create to perform different musical tasks. In addition to examples described above, might write a function to generate a chord progression or play an arpeggio, & can use functions again & again across many musical projects.

∗ 1.6.4. Reason 4: Code is useful for a thousand & 1 other things. Python is 1 of most powerful, multi-purpose languages in world. Used to create web servers & social media platforms as much as video games, animation, & music. Used for research & DS, politics & journalism. Knowing a little Python gives access to powerful ML & AI (AI/ML) techniques that are poised to transform most aspects of human work, including in creative domains e.g. music. Python is both a scripting language & a software engineering platform – equal parts duct tape & table saw – & it's capable of everything from quick fixes to durable software applications. Learning a little Python won't make you a software engineer, just like learning a few guitar chords won't make you a performance musician. But it's a start down a path. An open door that was previously closed, & a new way of using your mind & a new way of thinking about music.

∗ 1.6.5. Reason 5: Coding makes us more human. When think about learning to code, tend to think about economic payoff. Hear arguments that learning to code is a resume builder & a path to a high-paying job. Not that this perspective is wrong, but it might be wrong reason for you to learn how to code.

Just like people who are good at music *love* music, people who are good at coding tend to *love* coding. Craft of building software can be tedious & frustrating, but it can also be rewarding. A way to express oneself creatively & to engage in craftwork. People don't learn to knit, cook, or play an instrument for lucrative (có lợi nhuận) career paths that these pursuits open up – although by all means those pursuits can lead to remarkable careers. People learn these things because they have a *passion* for them. Because they are personally fulfilling. These passions connect us to centuries of tradition; they connect us to communities of teachers, learners, & practitioners; &, in end, they make us more *human.* So when things get a little frustrating – & things always get a little frustrating when learning any worthwhile skill – remember: just like poetry, literature, or music, code is an arts as much as it is a science. &, just like woodworking, knitting, or cooking, code is a craft as much as it is an engineering discipline. Be patient & give yourself a chance to fall in love with coding.

○ 1.7. Future of music & code. Before get on with book, wanted to leave you with a brief thought about future of technology, music, & code. For as long as there have been people on this planet there has been music. &, as long as there has been music, people have created technology to expand & enhance their creative potential. A drum is a kind of technology – a piece of animal hide stretched across a hollow log & tied in place. It's a polylithic accomplishment, an assembly of parts that requires skill & craft to make. One must know how to prepare animal hide, to make rope from plant fiber, & to craft & sharpen tools. More than that, one must know how to perform with drum, to connect with an audience, to enchant them to move their bodies through an emotional & rhythmic connection to beat. Technology brings together materials & tools with knowledge. People must have knowledge both to craft an artifact & to wield it. &, over time – over generations – that knowledge is refined as it gets passed down from teacher to student. It becomes stylized & diversified. Tools, artifacts, knowledge, & practice all become sth greater. Sth we call culture.

Again & again world of music has been disrupted, democratized, & redefined by new technologies. Hip-hop was a rebellion against musical status quo fueled by low-cost technologies like recordable cassette tapes, turntables, & 808 drum machines. Early innovators shattered norms of artistic expression, redefining music, poetry, visual art, & dance in process. Inexpensive access to technology coupled with a need for new forms of authentic self-expression was a match to dry tinder of racial & economic oppression.

Hard to overstate how quickly world is still changing as a result of technological advancements. Digital artifacts & infrastructures are so ubiquitous that they have reconfigured social, economic, legal, & political systems; revolutionized scientific research; upended arts & culture; & even wormed their way into most intimate aspects of our personal & romantic lives. Already talked about transformative impact that digital tools have had on world of music in 21st century, but exhilarating (& scary) part: we're on precipice of another wave of transformation in which human creativity will be redefined by AI & ML. Imagine AI accompanists that can improvise harmonies or melodies in real time with human musicians. Or DL algorithms that can listen to millions of songs & innovate music in same genre. Or silicon poets that grasp human language well enough to compose intricate rap lyrics. Or machines with trillions of transistor synapses so complex that they begin to "dream" – inverted ML algorithms that ooze imagery unhinged enough to disturb absinthe slumber of surrealist painters. Now, imagine: this is not speculative science fiction, but reality of our world today. These things are here now & already challenging what we mean by human creativity. What are implications of a society of digital creative cyborgs?

But here's trick: we've always been cyborgs. Western music notation is as much a technology as Python code. Becoming literate in any sufficiently advanced representation system profoundly shapes how think about & perceive world around us. Classical music notation, theory, & practice shaped mind of Beethoven as much as he shaped music with it – so much so that he was still able to compose many of his most famous works while almost totally deaf. BEETHOVEN was a creative cyborg enhanced by technology of Western music notation & theory. Difference: now we've externalized many of cognitive processes into machines that think alongside us. &, increasingly, these tools are available to everyone. How that changes what it means to be a creative human being is anyone's guess.

- ○ 1.8. Book overview. Excited to have you with us on this journey through music & code. A short guide for where go from here. Chaps. 2–3 cover foundations of rhythm, pitch, & harmony. These chaps are designed to move quickly & get you coding in Python early on. Cover Python variables, loops, which both connect directly to musical concepts. Chaps. 4–6 cover foundations of chords, scales, & keys using Python lists, functions, & data structures. Chaps. 7, 8, 10 shift from music composition to music production covering topics e.g. frequency domain, modular synthesis, & other production effects. In Chap. 9, switch to EarSketch platform to talk about how various musical elements are combined to compose full-length songs. Finally, Chap. 11 provides a short overview of history of music & code along with a glimpse of what future might hold. Between each chap, provide a series of short *interludes* that are like step-by-step tutorials to introduce new music & coding concepts.

  A few notes about how to read this book. Any time include Python code, it will be shown in a programming font. Sometimes write code in a table with line numbers so that can refer to specific lines. When introduce new terms, bold word. If get confused by any of programming or music terminology, check out appendices, which contain quick overviews of all of important concepts. Often invite to follow along with online examples. Best way to learn is by doing it yourself, so strongly encourage to try coding in Python online as go through chaps.

- Interlude 1: Basic Pop Beat. In this interlude, get familiar with TunePad interface by creating a basic rock beat in style of songs like *Roses* by SAINt JHN. Can follow along online by visiting `https://tunepad.com/interlude/pop-beat`

  1. **Step 1: Deep listening.** Good to get in habit of deep listening. Deep listening is practice of trying every possible way of listening to sounds. Start by loading a favorite song in a streaming service & listening – really listening – to it. Take song apart element by element. What sounds do you hear? How are they layered together? When do different parts come into track & how do they change over time? Think about how producer balances sounds across frequency spectrum or opens up space for transitions in lyrics. Try focusing on just drums. Can start to recognize individual percussion sounds & their rhythmic patterns?
  2. **Step 2: Create a new TunePad project.** Visit `https://tunepad.com` on a laptop or Chromebook & set up an account. [Recommend using free Google Chrome browser for best overall experience.] After signing in, click on `New Project` button to create an empty project workspace. Your project will look sth like this Fig. 1.3: TunePad project workspace.
  3. **Step 3: Kick drums.** In your project window, click on `ADD CELL` button & then select `Drums` Fig. 1.4: Selecting instruments in TunePad. In TunePad can think of a "cell" as an instrument that you can program to play music. Name new instrument "Kicks" & then add this Python code.

```
# play four kick drums
playNote(1)
playNote(1)
playNote(1)
playNote(1)
```

When done, your project should look sth like Fig. 1.5: Parts of a TunePad cell.

Go ahead & press Play button at top left to hear how this sounds.

Syntax errors. Occasionally your code won't work right & get a red error message box that looks sth like Fig. 1.6: Python syntax error in TunePad. This kind of error message is called a "syntax" error. In this case, code was written as `playnote` as a lowecase "n" instead of an uppercase "N". Can fix this error by changing code to read `playNote` on line 2.

4. **Step 4: Snare drums.** In your project window, click on `ADD CELL` button again & select `Drums`. Now should have 2 drum cells one appearing above the other in your project. Name 2nd instrument "Snare Drums" & then add this Python code.

```
# play 2 snare drums on the up beats only
rest(1) # skip a beat
playNote(2) # play a snare drum sound
rest(1)
playNote(2)
```

Might start to notice text that comes after hashtag symbol # is a special part of your program. This text is called a *comment*, & it's for human coders to help organize & document their code. Anything that comes after hashtag on a line is ignored by Python. Try playing this snare drum cell to hear how it sounds. Can also play kick drum cell at same time to see how they sound together.

5. **Step 5: Hi-hats.** Click on `ADD CELL` button again to add a 3rd drum cell. Change title of this cell to be "Hats" & add code:

```
# play four hats between the kicks and snares
rest(0.5) # rest for half a beat
playNote(4, beats=0.5) # play a hat for half a beat
rest(0.5)
playNote(4, beats=0.5)
rest(0.5)
playNote(4, beats=0.5)
rest(0.5)
playNote(4, beats=0.5)
```

When play all 3 of drum cells together, should hear a basic rock beat pattern: `kick - hat - snare - hat - kick - hat - snare - hat`.

6. **Step 6: Fix your kicks.** Might notice kick drums feel a little heavy in this mix. Can make some space in pattern by resting on up beats (beats 2 & 4) when snare drums are playing. Scroll back up to your `Kick drum cell` & change code to look like this:

```
# play kicks on the down beats only
playNote(1)
rest(1)
playNote(1)
rest(1)
playNote(1)
rest(1)
playNote(1)
rest(0.5) # rest a half beat
playNote(1, beats = 0.5) # half beat pickup kick
```

7. **Step 7: Adding a bass line.** Add a new cell to your project, but this time select `Bass` instead of `Drums`. Once cell is loaded up, change voice to `Plucked Bass` Fig. 1.7: Selecting an instrument's voice in TunePad.

Entering this code to create a simplified bass line in style of *Roses* by SAINt JHN. When done, try playing everything together to get full sound.

```
playNote(5, beats=0.5) # start on low F
playNote(17, beats=0.5) # up an octave
rest(1)

playNote(10, beats=0.5) # A sharp
playNote(22, beats=0.5) # up an octave
rest(1)
```

```
playNote(8, beats=0.5) # G sharp
playNote(20, beats=0.5) # up an octave
rest(0.5)

playNote(8, beats=0.5) # G sharp - G - G
playNote(12, beats=0.5)
playNote(24, beats=0.5)

playNote(10, beats=0.75) # C sharp
playNote(22, beats=0.25) # D sharp
```

- 2. Rhythm & tempo. This chap dives into fundamentals of *rhythm* in music. Start with beat – what it is, how it's measured, & how can visualize beat to compose, edit, & play music. From there provide examples of some common rhythmic motifs from different genres of music & how to code them with Python. Main programming concepts for this chap include loops, variables, calling function, & passing parameter values. This chap covers a lot of ground, but it will give you a solid start on making music with code.
  - 2.1. Beat & tempo. *Beat* is foundation of rhythm in music. Term *beat* has a number of different meanings in music, [Term beat can also refer to main groove in a dance track ("drop the beat") or instrumental music that accompanies vocals in a hip-hop track ("she produced a beat for a new artist") in addition to other meanings.] but this chap uses it to mean a unit of time, or how long an individual note is played – e.g., "rest for 2 beats" or "play a note for half a beat". Based on beat, musical notes are combined in repeated patterns that move through time to make rhythmic sense to our ears.

    Tempo refers to speed at which rhythm moves, or how quickly 1 beat follows another in a piece of music. As a listener, can feel tempo by tapping your foot to rhythmic pulse. Standard way to measure tempo is in beats per minute (BPM or bpm), meaning total number of beats played in 1 minute's time. This is almost always a whole number like 60, 120, or 78. At a tempo of 60 bpm, your foot taps 60 times each minute (or 1 beat per sec). At 120 bpm, get 2 beats every sec; &, at 90 bpm, get 1.5 beats every sec. Later in this chap when start working with TunePad, can set tempo by clicking on bpm indicator in top bar of a project, see Fig. 2.1: TunePad project information bar. Can click on tempo, time signature, or key to change settings for your project.

    Different genres of music have their own typical tempo ranges (although every song & every artist is different). E.g., hip-hop usually falls in 60–110 bpm range, while rock is faster in 100–140 bpm range. House/techno/trance is faster still, with tempos between 120–140 bpm. [Table: Genre: Tempo Range (BPM)].

    It takes practice for musicians to perform at a steady tempo, & they sometimes use a device called a *metronome* to help keep their playing constant with pulse of music. Can create a simple metronome in TunePad using 4 lines of code in a drum cell. This works best if switch instrument to `Drums → Percussion Sounds`.

    ```
    playNote(3, velocity = 100) # louder 1st note
    playNote(3, velocity = 60)
    playNote(3, velocity = 60)
    playNote(3, velocity = 60)
    ```

    Can adjust tempo of your metronome with bpm indicator Fig. 2.1: TunePad project information bar. Can click on tempo, time signature, or key to change settings for your project. As this example illustrates, computers excel at keeping a perfectly steady tempo. This is great if want precision, but there's also a risk that resulting music will sound too rigid & machine-like. When real people play music they often speed up or slow down, either for dramatic effect or just as a result of being a human. Depending on genre, performers might add slight variations in rhythm called swing or shuffle, that's a kind of back & forth rocking of beat that you can feel almost more than you can hear. Show how to add a more human feel to computer generated music later in book.
  - 2.2. Rhythmic notations. Over centuries, musicians & composers have developed many different written systems to record & share music. With invention of digital production software, a number of other interactive representations for mixing & editing have become common as well. Here are 4 common visual representations of same rhythmic pattern. Pattern has a total duration of 4 beats & can be counted as "1 & 2, 3 & 4". 1st 2 notes are $\frac{1}{2}$ beats long followed by a note that is 1 beat long. Then pattern repeats.
    - 2.2.1. Representation 1: Standard Western music notation. 1st representation shows standard music notation (or Western notation), a system of recording notes that has been developed over many hundreds of years. 2 thick vertical lines on left side of illustration indicate: this is rhythmic notation, i.e., there is no information about musical pitch, only rhythmic timing. Dots on long horizontal lines are notes whose shapes indicate duration of each to be played. Sometimes different percussion instruments will have their notes drawn on different lines. Describe what various note symbols mean in more detail in Fig. 2.2: Standard notation example.
    - 2.2.2. Representation 2: Audio waveforms. 2nd representation shows a visualization of actual audio waveform that gets sent to speakers when play music. Waveform shows amplitude (or volume) of audio signal over time. Next chap talks more

about audio waveforms, but for now can think of a waveform as a graph that shows literal intensity of vibration of your speakers over time. When compose a beat in TunePad, can switch to waveform view by clicking on small dropdown arrow at top-left side of timeline Fig. 2.3: Waveform representation of Fig. 2.2.

∗ 2.2.3. Representation 3: Piano (MIDI) roll. 3rd representation shows a piano roll (or MIDI (Musical Instrument Digital Interface) roll). This uses solid lines to show individual notes. Length of lines represents length of individual notes, & vertical position of lines represents percussion sound being played (kick drums & snare drums in this case). This representation is increasingly common in music production software. Many tools even allow for drag & drop interaction with individual notes to compose & edit music Fig. 2.4: Piano or MIDI roll representation of Fig. 2.2.

∗ 2.2.4. Representation 4: Python code. A final representation for now shows Python code in TunePad. In this representation, duration of each note is set using `beats` parameter of `playNote` function calls.

```
playNote(2, beats = 0.5)
playNote(2, beats = 0.5)
playNote(6, beats = 1)

playNote(2, beats = 0.5)
playNote(2, beats = 0.5)
playNote(6, beats = 1)
```

Each of these representation has advantages & disadvantages; they are good for conveying some kinds of information & less good at conveying others. E.g., standard rhythm notation has been refined over centuries & is accessible to an enormous, worldwide community of musicians. On other hand, it can be confusing for people who haven't learn how to read sheet music. Timing of individual notes is communicated using tails & flags attached to notes, but there's no consistent mapping between horizontal space & timing.

Audio waveform is good at showing what sound *actually* looks like – how long each note rings out ("release") & how sharp its onset is ("attack"). Helpful for music production, mixing, & mastering. On other hand, waveforms don't really tell you much about pitch of a note or its intended timing as recorded by composer.

Python code is easier for computers to read than humans – it's definitely not sth you would hand to a musician to sight read. On other hand, it has advantage that it can be incorporated into computer *algorithms* & manipulated & transformed in endless ways.

There are many, many other notation systems designed to transcribe a musical performance – what hear at a live performance – onto a sheet of paper or a computer screen. Each of these representations was invented for a specific purpose &/or genre of music. Might pick a representation based on context & whether you're in role of a musician (& what kind of instrument you play), a singer, a composer, a sound engineer, or a producer. Music notation systems are as rich & varied as cultures & musical traditions that invented them. 1 nice thing about working with software: easy to switch between multiple representations of music depending on task trying to accomplish.

○ 2.3. Standard rhythmic notation. This sect will review a standard musical notation system that has roots in European musical traditions. This system is versatile & has been refined & adapted over a long period of time across many countries & continents to work with an increasingly diverse range of instruments & musical genres. Start with percussive rhythmic note values in this chap, & move on to working with pitched instruments in Chap. 3.

Fig. 2.5: Common note symbols starting with a whole note (4 beats) on top down to 16th notes on bottom. Notes on each new row are half length of row above. shows most common symbols used in rhythmic music notation. Notes are represented with oval-shaped dots that are either open or closed. All notes except for whole note have tails attached to them that can point either up or down. It doesn't matter which direction (up or down) tail points. Notes that are faster than a quarter note also have horizontal flags or beams connected to tails. Each additional flag or beam indicates note is twice as fast.

Symbol: Name: Beats: TunePad code:

1. Whole Note: Larger open circle with no tail & no flag: 4: `playNote(1, beats = 4)`
2. Half Note: Open circle with a tail & no flag: 2: `playNote(1, beats = 2)`
3. Quarter Note: Solid circle with a tail & no flag: 1: `playNote(1, beats = 1)`
4. 8th Note: Solid circle with a tail & 1 flag or bar: 0.5 or $\frac{1}{2}$: `playNote(1, beats = 0.5)`
5. 16th Note: Solid circle with a tail & 2 flags or bars: 0.25 or $\frac{1}{4}$: `playNote(1, beats = 0.25)`
6. Dotted Half Note: Open circle with a tail. Dot adds an extra beat to half note: 3: `playNote(1, beats = 3)`
7. Dotted Quarter Note: Solid circle with a tail. Dot adds an extra half-beat: 1.5: `playNote(1, beats = 1.5)`
8. Dotted 8th Note: Solid circle with tail & 1 flag. Dot adds an extra quarter beat: 0.75. `playNote(1, beats = 0.75)`

Standard notation also includes *dotted notes*, where a small dot follows note symbol. With a dotted note, take original note's duration & add half of its value to it. So, a dotted quarter note is 1.5 beats long, a dotted half note is 3 beats long, etc.

There are also symbols representing different durations of silence or "rests".

Symbol: Name: Beats: TunePad code

1. Whole Rest: 4: `rest(beats = 4)`
2. Half Rest: 2: `rest(beats = 2)`

3. Quarter Rest: 1: `rest(beats = 1)`
4. 8th Rest: 0.5 or $\frac{1}{2}$: `rest(beats = 0.5)`
5. 16th Rest: 0.25 or $\frac{1}{4}$: `rest(beats = 0.25)`

- 2.4. Time signatures. In standard notation, notes are grouped into segments called *measures* (or bars). Each measure contains a fixed number of beats, & duration of all notes in a measure should add up to that amount. Relationship between measures & beats is represented by a fraction called a *time signature*. Numerator (or top number) indicates number of beats in measure, & denominator (bottom number) indicates beat duration.

  – Trong ký hiệu chuẩn, các nốt nhạc được nhóm thành các đoạn gọi là *nhịp* (hoặc ô nhịp). Mỗi ô nhịp chứa 1 số phách cố định, & thời lượng của tất cả các nốt nhạc trong 1 ô nhịp phải cộng lại bằng số lượng đó. Mối quan hệ giữa các ô nhịp & nhịp được biểu diễn bằng 1 phân số gọi là *nhịp điệu*. Tử số (hoặc số trên cùng) biểu thị số phách trong ô nhịp, & mẫu số (số dưới cùng) biểu thị thời lượng của phách.

  1. $\frac{4}{4}$: 4-4 Time or "Common tTime": There are 4 beats in each measure, & each beat is a quarter note. This time signature is sometimes indicated using a special symbol
  2. $\frac{2}{2}$: 2-2 Time or "Cut Time": There are 2 beats in each measure, & beat value is a half note. Cut time is sometimes indicated with a 'C' with a line through it.
  3. $\frac{2}{4}$: 2-4 Time: There are 2 beats in each measure, & quarter note gets beat.
  4. $\frac{3}{4}$: 3-4 Time: There are 3 beats in each measure, & quarter note gets beat.
  5. $\frac{3}{8}$: 3-8 Time: There are 3 beats in each measure, & 8th note gets beat.

  Most common time signature is $\frac{4}{4}$. So common, in fact, referred to as *common time*. Often denoted by a C symbol shown in table. In common time, there are 4 beats to each measure, & quarter note "gets beat" meaning: 1 beat is same as 1 quarter note.

  Vertical lines separate measures in standard notation. In example, there are 2 measures in 4/4 time (4 beats in each measure, & each beat is a quarter note).

  If have a time signature of 3/4, then there are 3 beats per measure, & each beat's duration is a quarter note. Some examples of songs is 3/4 time are *My Favorite Things* from *The Sound of Music, My 1st Song* by Jay Z, *Manic Depression* by Jimi Hendrix, & *Kiss from a Rose* by Seal.

  If those notes were 8th notes, it would look like Fig.

  Other common time signatures include 2/4 time (with 2 quarter note beats per measure) & 2/2 time (with 2 *half note* beats in each measure). With 2/2 there are actually 4 quarter notes in each measure because 1 half note has same duration as 2 quarter notes. For this reason, 2/2 time is performed similarly to common time, but is generally faster. It is referred to as *cut time* & is denoted by a C symbol with a line through it.

  Can adjust time signature of your TunePad project by clicking on time indicator in top bar (see Fig. 2.1).

- 2.5. Percussion sounds & instruments. Working with rhythm, come across lots of terminology for different percussion instruments & sounds. A quick rundown on some of most common drum sounds that you'll work with in digital music (Fig. 2.6: Drums in a typical drum kit.)

  Drum names: Description: TunePad note number

  1. Kick or bass drum: Kick drum (or bass drum) makes a loud, low thumping sound. Kicks are commonly placed on beats 1 & 3 in rock, pop, house, & electronic dance music. In other genres like hip-hop & funk, kick drums are very prominent, but their placement is more varied: 0 & 1
  2. Snare: Snare drums make a recognizable sharp staccato sound that cuts across frequency spectrum. They are built with special wires called snares that give drums its unique snapping sound. Snare drums are commonly used on beats 2 & 4: 2 & 3
  3. Hi-hat: Hi-hat is a combination of 2 cymbals sandwiched together on a metal rod. A foot pedal opens or closes cymbals together. In closed position hi-hat makes a bright tapping sound. In open position cymbal is allowed to ring out. Hi-hats have become an integral part of rhythm across almost all genres of popular music.: 4 (closed), 5 (open)
  4. Low, mid, high tom: Tom drums (tom-toms) are cylindrical drums that have a less snappy sound than snare drum. Drum kits typically have multiple tom drums with slightly different pitches (e.g. low, mid, & high).: 6, 7, 8
  5. Crash cymbal: A large cymbal that makes a loud crash sound, often used as a percussion accent: 9
  6. Claps & shakers: Different TunePad drum kits include a range of other percussion sounds common in popular music including various claps, shakers, & other sounds.: 10 & 11

  * 2.5.1. 808 Drum kit. Released in early 1980s, Roland 808 drum machine was a hugely influential sound in early hip-hop music (& other genres as well). 8.8 used electronic synthesis techniques to create synthesis replicas of drum sounds like kicks, snares, hats, toms, cowbells, & rim shots. Tinkerers would also open up 808s & hack circuits to create entirely new sounds. Today 808s usually refers to low, booming bass lines that were 1st generated using tweaked versions of 808s' kick drums. TunePad's default drum kit uses samples that sound like original electronically synthesized 808s (Fig. 2.7: Roland 808 drum sequencer.).
  * 2.5.2. Selecting TunePad instruments. When coding in Tunepad, sound that your code makes will depend on instrument you have selected. If coding a rhythm, can choose from several different drum kits including an 808 & rock kits. Can change instrument by clicking on selector shown below Fig. 2.8: Changing an instrument's voice in TunePad.

- 2.6. Coding rhythm in Python.
  - 2.6.1. Syntax errors. Python is a text-based language, i.e., you're going to be typing code that has to follow strict grammatical rules. When speak a natural language like English, grammar is important, but can usually bend or break rules & still get your message across. When say something ambiguous it can be ironic, humorous, or poetic. This isn't case in Python. Python has no sense of humor & no appreciation for poetry. If make a grammatical mistake in coding, Python gives a message called a *syntax error*. These messages can be confusing, but they're there to help you fix your code in same way that a spell checker helps you fix typos. Here's what a syntax error looks like in TunePad (Fig. 2.9: Example of a Python syntax error in TunePad. This line of code was missing a parenthesis symbol.)

    This line of code was missing a parenthesis symbol, which generated error message "bad input on line 5". Notice Python is giving hints about where problems are & how to fix them, but those hints aren't always that helpful & can be frustrating for beginners.
  - 2.6.2. Flow of control. A Python program is made up of a list of statements. For most part, each statement goes on its own line in your program. Python will read & perform each line of code from top to bottom in order that you write them. In programming this is called *flow of control*. This is similar to way you would read words in a book or notes on a line of sheet music. Difference: programming languages also have special rules that let you change flow of control. Those rules include *loops* (which repeat some part of your code multiple times), *conditional logic* (which runs some part of your code only if some condition is met), & *user-defined functions* (which lets you create your own functions that can be called). Talk about these special "control structures" later in book.
- 2.7. Calling functions. Almost everything you do in Python involves *calling* functions. A function (sometimes called a command or an instruction) tells Python to do something or to compute a value. E.g., `playNote` function tells TunePad to make a sound. There are 3 things you have to do to call a function:

  1st, have to write name of function. Functions have 1-word names (with no spaces) that can consist of letters, numbers, & underscore _ character. Multi-word functions will either use underscore character between words as in `my_multi_word_function()` or each new word will be capitalized as in `playNote()`.

  2nd, after type name of function, have to include parentheses. Parentheses tell Python that you're calling a function.

  3rd, include any *parameters* that you want to *pass* to function in between left & right parentheses. A parameter provides extra information or tells function how to behave. E.g., `playNote` statement needs at least 1 parameter to tell it which note or sound to play. Sometimes functions accept multiple parameters (some of which can be optional). `playNote` function accepts several optional parameters described in next sect. Each additional parameter is separated with a comma (Fig. 2.10: Calling `playNote` function in TunePad with 2 parameters inside parentheses.)
- 2.8. playNote functions. `playNote` function tells TunePad to play a percussion sound or a musical note. `playNote` function accepts up to 4 parameters contained within parentheses.

  ```
  playNote(1, beats = 1, velocity = 100, sustain = 0)
  ```

  Name: Description

  - `note`: This is a *required* parameter that says which note or percussion sound to play. Kind of sound depends on which instrument you have selected in TunePad for this code. Can play more than 1 note at same time by enclosing notes in square brackets.
  - `beats`: An *optional* parameter that says how long to play note. TunePad *playhead* will be moved forward by duration given. This parameter can be a whole number (like 1 or 2), a decimal number (like 1.5 or 0.25), or a fraction (like 1/2).
  - `velocity`: An *optional* parameter that says how loud to play note or sound. A value of 100 is full volume, & a value of 0 is no volume (muted). Velocity is a technical term in digital music that means how fast or how hard you hit instrument. You might imagine it as how loud a drum sounds based on how hard it gets hit.
  - `sustain`: An *optional* parameter that allows a note to ring out for an additional number of beats without advancing playhead.
  - 2.8.1. Optional parameters. Sometimes parameters are *optional*, i.e., they have a value that gets provided by default if you don't specify one. For `playNote`, only note parameter is required. If don't pass other parameters, it provides values for you by default. Can also include *names* of parameters in a function call. E.g., all 4 of lines below do exactly same thing; they play a note for 1 beat. 1st 2 use parameters without their names. 2nd 2 include names of parameter, followed by equals sign =, followed by parameter value.

    ```
    playNote(60) # the beats parameter is optional
    playNote(60, 1) # with the beats parameter set to 1
    playNote(60, beats = 1) # with a parameter name for beats
    playNote(note = 60, beats = 1) # with a parameter name for note and beats
    ```

  - 2.8.2. Comments. In code above, some of text appears after hashtag # symbols on each line. This text is called a *comment*. A comment is a freedom note that programmers add to make their code easier to understand. Comment text is ignored by Python, so you can write anything you want after hashtag symbol on a line. Can also use a hashtag at beginning of a line to temporarily disable code. This is called "commenting out" code.

- 2.9. `rest` function. Silence is an important element of music. `rest` function generates silence, or a break in sound. It only takes 1 parameter, which is length of time the rest is held. So `rest(beats = 2)` will trigger a rest for a length of 2 beats. If don't provide a parameter, `rest` uses a value of 1.0 by default.

  ```
  rest() # rest for one beat
  rest(1.0) # rest for one beat
  rest(0.25) # rest for one quarter beat
  rest(beats = 0.25) # rest for one quarter beat
  ```

- 2.10. Examples of `playNote, rest`. Try a few examples of `playNote, rest` to get warmed up. This rhythm plays 2 8 notes (beats = 0.5) followed by a quarter note (beats = 1). Pattern then repeats a 2nd time. Here's how would code this in TunePad with a kick drum & snare:

  ```
  playNote(1, beats = 0.5) # play a kick drum (1) for half a beat
  playNote(1, beats = 0.5)
  playNote(2, beats = 1) # play snare (2) for one beat
  playNote(1, beats = 0.5) # play kick (1) for half a beat
  playNote(1, beats = 0.5)
  playNote(2, beats = 1) # play snare (2) for one beat
  ```

  Here's another example that plays a quarter note followed by a rest of 0.5 beats followed by an 8 note (beats = 0.5). Pattern is repeated 2 times in a row:

  ```
  playNote(2, beats = 1) # play a snare drum (2) for one beat
  rest(beats = 0.5) # rest for half a beat
  playNote(1, beats = 0.5) # play a kick drum (1) for half a beat
  playNote(2, beats = 1) # play a snare drum (2) for one beat
  rest(beats = 0.5) # rest for half a beat
  playNote(1, beats = 0.5)
  ```

  A 3rd example that plays 8 notes in a row, each an 8 note (beats = 0.5).

- 2.11. Loops. All of examples in prev sect included repeated elements. &, if listen closely, can hear repeated elements at all levels of music. There are repeated rhythmic patterns, recurring melodic motifs, & storylines defined by song sects that get repeated & elaborated. It turns out: there are many circumstances in both music & computer programming where we want to repeat something over & over again.

  To show how can take advantage of some of capabilities of Python, start with last example from prev sec where we wanted to tap out a run of 8th notes (0.5 beats) on hi-hat. 1 way to program that rhythm would be to just type in 8 `playNotes` in a row.

  ```
  for i in range(8):
      playNote(4, beats = 0.5)
      print(i)
  ```

  This will get job done, but there are a few problems with this style of code. 1 problem: it violates 1 of most important character traits of a computer programmer – laziness! A lazy programmer is someone who works smart, not hard. A lazy programmer avoids doing repetitive, error-prone work. A lazy programmer knows that there are some things that computer can do better than a human can.

  In Python (& just about any other programming language), when want to do something multiple times, can use a loop. Python has a number of different kinds of loops, but, in this case, our best option is sth called a `for` loop. Version of code on right repeats 8 times in a row. For each iteration of loop, TunePad `playNote` function gets called.

  With original code on left, had to do a lot of tying (or, more likely, copying & pasting) to enter our program – a warning sign that we're not being lazy enough. We generated a lot of repetitive code, which makes program harder to read (not as legible), error prone, & not as elegant as it could be. Right-side code accomplishes same thing with just 3 lines of code instead of 8.

  Finally, code on left is harder to change & reuse. What if wanted to use a different drum sound (like a snare instead of a hat)? Or, what if wanted to tap out a run of 16 16th notes instead of 8 8th notes? Would have to go through code line by line making same change over & over again. This is a slow, error-prone process that is definitely not lazy or elegant.

  To see why this is better, try changing code on right so that it plays 16 16th notes instead of 8 8th notes. Or try changing drum sound from a hat to sth else. `print` statement on line 3 is just there to help you see what's going on with your code. If click on `Show Python Output` option, can see how variable called `i` (that gets created on line 1) counts up from 0 to 7 Fig. 2.11: How to show print output of your code in a TunePad cell. More detail about anatomy of a `for` loop (Fig. 2.12: Anatomy of a `for` loop in Python.) A `for` loop with range function:

  - begins with `for` keyword

- ∗ includes a loop *variable* name; this can be anything you want (above it is `i`). Each time loop goes around, loop variable is incremented by 1.
- ∗ includes `in` keyword
- ∗ includes `range` function that says how many times to repeat
- ∗ includes a colon :
- ∗ includes a block of code indented by 4 spaces

Python uses *indentation* to determine what's *inside* loop, meaning it's sect of code that gets repeated multiple times. Intended block of code is repeated total number of times specified by `range`. Try adding a few extras to prev example. In version below, add a run of 16th notes for last beat.

```
for i in range(6):
    playNote(4, beats = 0.5)
for i in range(4):
    playNote(4, beats = 0.25)
```

But there are lots of other things we could do as well. If wanted to play an even faster run, could use code like:

```
for i in range(8):
    playNote(4, beats = 0.125)
```

Or, if wanted to play a triplet that divides a half-beat into 3 equal parts, could do sth like this:

```
for i in range(3):
playNote(4, beats = 0.25 / 3) # divide into 3 parts
```

If open this example in TunePad, can experiment with different combinations of numbers to get different effects: https://tunepad.com/examples/loops-and-hats

- ○ 2.12. Variables. A *variable* is a name you give to some piece of information in a Python program. Can think of a variable as a kind of nickname or alias. Similar to loops, variables help make your code more elegant, easier to read, & easier to change in future. E.g., code on left plays a drum pattern without variables, & code on right plays same thing with variables. Notice how variables help make code easier to understand because they give us descriptive names for various drum sounds instead of just numbers.

```
playNote(0)
playNote(4)
playNote(2)
playNote(4)

kick = 0
hat = 4
snare = 2

playNote(kick)
playNote(hat)
playNote(snare)
playNote(hat)
```

In version on right defined a variable called `kick` on line 1, a variable called `hat` on line 2, & a variable called `snare` on line 3. Each variable is *initialized* to a different number for corresponding drum sound. Also possible to change value of a variable later in program by assigning it a different number.

```
kick = 0
playNote(kick) # plays sound 0
kick = 1       # set kick to a different value
playNote(kick) # plays sound 1
```

Variable names can be anything you want as long as they're 1 word long (no spaces) & consist only of letters, numbers, & underscore character `_`. Variable names cannot start with a number, & they can't be same as any existing Python keyword.

As begin to get comfortable with code & to exercise your creativity, find yourself wanting to experiment with sounds. Might want to try different sounds for same rhythmic pattern, maybe change a high-hat sound to a shaker to get a more organic feel. Using variables makes it easy to experiment by changing values around.

Another example with a hi-hat pattern. Imagine really like this pattern, but wondering how it would sound with a different percussion instrument. Maybe you want to change 4 sound to a shaker sound (like 11). Nice thing about variables: can give them just about any name you want as long as Python is not already using that name for something else. This way you can make name meaningful to you. So, for our shaker example could create a variable with a meaningful name like `shake` & set it equal to 11. When use variable `shake` you are inserting whatever number is currently assigned to it.

```
for i in range(8):
    playNote(4,
    beats = 0.5)
for i in range(8):
    playNote(4,
    beats = 0.25)
for i in range(4):
    playNote(4,
    beats = 0.5)

shake = 11
for i in range(8):
    playNote(shake,
    beats = 0.5)
for i in range(8):
    playNote(shake,
    beats = 0.25)
for i in range(4):
    playNote(shake,
    beats = 0.5)
```

As progress with coding, find that loops & variables help create a smoother workflow that gives you more flexibility, freedom, & creative power. Try out using variables with exercise https://tunepad.com/examples/variables.

- 2.13. More on syntax errors. Python code is like a language with strict grammatical rules called syntax. When make a mistake in coding – & everyone makes coding mistakes all time – Python will give feedback about what error is & approximately what line it's on. E.g., if been trying to code exercises in this chap, may have seen a message like Fig. 2.13: Example of a Python syntax error. Command 'ployNote' should instead say 'playNote'.

  This is telling us: there is an error on line 6 that can be fixed by changing text, "ployNote". When using a variable or function in your code, Python is expecting you to type it *exactly* the same as it was defined. A simple typo can stop your program from running, but it's also easily fixed. Here just need to update line to say `playNote` instead of `ployNote`.

  Other syntax errors are trickier. Message in Fig. 2.14: Example of a Python syntax error. Here problem is actually on line 1, not line 2. Message in Fig. 2.14 is confusing because problem is actually on line 1 even though syntax error says line 2. Problem is a missing right parenthesis on line 1.

  1 technique coders use to find source of errors like this: comment out lines of code before & after an error. E.g., to comment out1st line of code above, could change it to look like this:

```
# playNote(60
rest(1)
```

  Adding hashtag at beginning of 1st line means Python ignores it, in this case fixing syntax error & giving us another clue about source of problem.

  Another surprisingly helpful trick: just paste your error message verbatim into your favorite search engine. There are huge communities of Python coders out there who have figured out how to solve almost every problem with code imaginable. You can often find a quick fix to your problem just by browsing through a few of top search results.

  If want practice fixing syntax errors in your code, can try 1 of mystery-melody challenges on TunePad: https://tunepad.com/examples/mystery-melody.

- 2.14. `playhead`. Timing of notes in TunePad is determined by position of an object called `playhead`. In early days of music production, recordings were made using analog tape. Sound wave signals coming from a microphone or some other source were physically stored on magnetic tape using a mechanism called a *record head*. As tape moved by, record head would inscribe patterns of magnetic material inside of tape, thus creating a recording of music. To play recording back, a `playhead` would pick up fluctuations in tape's magnetic material & convert it back into sound waves for listeners to hear. Fast forward to digital realm. No longer have playheads or record heads, but maintain that metaphor when referring to notion of sound moving in time. Concept of a playhead is common across audio production software as point in time where audio is playing.

  – Thời gian của các nốt nhạc trong TunePad được xác định bởi vị trí của 1 đối tượng được gọi là `playhead`. Vào những ngày đầu của quá trình sản xuất âm nhạc, các bản ghi âm được thực hiện bằng băng analog. Tín hiệu sóng âm phát ra từ micrô hoặc 1 số nguồn khác được lưu trữ vật lý trên băng từ bằng 1 cơ chế được gọi là *record head*. Khi băng di chuyển qua, đầu ghi sẽ khắc các mẫu vật liệu từ tính bên trong băng, do đó tạo ra bản ghi âm nhạc. Để phát lại bản ghi âm, `playhead` sẽ thu các dao động trong vật liệu từ tính của băng & chuyển đổi nó trở lại thành sóng âm để người nghe có thể nghe. Chuyển nhanh đến thế giới kỹ thuật số. Không còn đầu phát hoặc đầu ghi nữa, nhưng vẫn duy trì phép ẩn dụ đó khi đề cập đến khái niệm âm thanh chuyển động theo thời gian. Khái niệm về đầu phát phổ biến trong các phần mềm sản xuất âm thanh như 1 điểm thời gian mà âm thanh đang phát.

In TunePad, when place a note with `playNote` function, it advances playhead forward in time by duration of note specified by `beats` parameter. There are several functions available to get information about position of playhead & move it forward or backward in time.

Function: Description

- `getPlayhead()`: Returns current position of playhead in beats. Note: `getPlayhead` returns elapsed number of beats, i.e. if playhead is at beginning of track function will return 0. If 1.5 beats have elapsed, `getPlayhead` will return 1.5. If 40 beats have elapsed, it will return 40, & so on.
- `getMeasure()`: Returns current measure as an integer value. Note: `getMeasure` returns an elapsed number of measures. So, if playhead is at beginning of track or anywhere before end of 1st measure, function will return 0. If playhead ≥ start of 2nd measure, `getMeasure` will return 1, & so on.
- `getBeat()`: Returns an elapsed number of beats *within current measure* as a decimal number. E.g., if playhead has advanced by a quarter beat within a measure, `getBeat` will return 0.25. Value returned by `getBeat` will always < total number of beats in a measure.
- `fastForward(beats)`: Advance playhead forward by given number of beats relative to current position. Note: this is identical to `rest` function. Negative beat values move playhead backward.
- `rewind(beats)`: Move playhead back in time by given number of beats. This can be a useful way to play multiple notes at same time. Beats parameter specifies number of *beats* to move playhead. Negative values of beats move playhead forward.
- `rest(beats)`: Advance playhead forward by given number of beats without playing a sound. This is identical to `fastForward` function.
- `moveTo(beats)`: Move playhead to an arbitrary position. `beats` parameter specifies point that playhead will be placed as an elapsed number of beats. E.g., `moveTo(0)` will move playhead to beginning of a track (zero elapsed beats). `moveTo(1)` will place playhead at end of 1st beat & right before start of 2nd beat.

Can control where playhead is relative to music we make by using `moveTo, fastForward, rewind` commands. `rewind` & `fastForward` functions move playhead backward or forward relative to current point in time. `moveTo` function takes playhead & moves it to an arbitrary point in time. In TunePad, playhead represents an *elapsed* number of beats. So, to move to beginning of a track, would use `moveTo(0)`, i.e. 0 elapsed beats. To move to beginning of 2nd beat, would use `moveTo(1)`, i.e. 1 elapsed beat. These commands are useful for adding multiple overlapping rhythms to a single TunePad cell. See more on how these commands can be used in Chap. 8.

- 2.15. Basic drum patterns. Code some foundational drum patterns. There is also a link to code in TunePad so that you can play around with beat & make it your own.
  - 2.15.1. 4-on-the-floor. 4-on-the-floor pattern is a staple of House, EDM, disco, & pop music. It has a driving dance beat defined by 4 kick drum hits on each beat (thus 4 beats on floor). This beat is simple but versatile. Can spice it up by moving hi-hats around & adding kicks, snares, & other drums in unexpected places. Can make basic pattern with just 3 instruments: kick, snare, & hats.
    Kick drums are lowest drum sound in a drum kit. Start by laying down kick drums on each beat of measure. These low sounds give this pattern a driving rhythmic structure that sounds great at higher tempos. Then add snare hits on even beats (2 & 4). Snare adds energy & texture to beat. Finally, add hi-hats. These are highest pitch instruments in most drum beats, & they help outline groove to emphasize beat. Can find this example at https://tunepad.com/examples/four-on-the-foor.

```
# define instrument variables
kick = 0
snare = 2
hat = 4

# lay down four kicks (on the floor)
playNote(kick)
playNote(kick)
playNote(kick)
playNote(kick)

moveTo(0) # reset playhead to the beginning

# add snares on the even beats
rest(1)
playNote(snare)
rest(1)
playNote(snare)
```

```
moveTo(0) # reset playhead to the beginning

# hi-hat pattern with a loop!
for i in range(8):
    playNote(hat, 0.5)
```

∗ 2.15.2. Blues. Blues is a genre of music that evolved from African American experience, starting as field songs, evolving into spirituals, & eventually became Blues. This beat is in 3/4 time, i.e. there are 3 beats in each measure. Can find this example at https://tunepad.com/examples/blues-beat.

```
# define instrument variables
kick = 0
snare = 2
hat = 4

# lay down kick and snare pattern
playNote(kick, beats = 0.5)
playNote(kick, beats = 0.25)
playNote(snare, beats = 0.25)
rest(.25)
playNote(kick, beats = 0.25)
playNote(kick, beats = 0.25)
rest(.25)
playNote(kick, beats = 0.25)
playNote(snare, beats = 0.25)
playNote(kick, beats = 0.25)
playNote(kick, beats = 0.25)

moveTo(0) # reset playhead to beginning

# add hi-hats
playNote (4, beats = 0.25)
for i in range(3):
    rest(0.25)
    playNote (hat, beats = 0.25)
    playNote (hat, beats = 0.25)
rest(0.25)
playNote(hat, beats = 0.25)
```

∗ 2.15.3. Latin. Latin beats are known for their syncopated rhythms that emphasize so-called "weak beats" in a measure. This drum pattern is 2 measures long. Our kick pattern sounds much like a heartbeat & solidly grounds our entire beat. Our snare is playing a *clave* pattern, which is common in many forms of Afro-Cuban music e.g. salsa, mambo, reggae, reggaeton, & dancehall. In 2nd measure, have hits on 1st beat & 2nd half of 2nd beat (counts 1 & 2.5). Finally, add a hi-hat on every 8th note. Can find this example at https://tunepad.com/examples/latin-beat.

```
# define instrument variables
kick = 0
snare = 3
hat = 4

# lay down kicks for the heartbeat
for i in range(2):
    playNote(kick, beats = 1.5)
    playNote(kick, beats = 0.5)
    playNote(kick, beats = 1.5)
    playNote(kick, beats = 0.5)

moveTo(0) # reset playhead

# add snare
rest(1.0)
playNote(snare)
playNote(snare)
rest(1.0)
playNote(snare, beats = 1.5)
```

```
    playNote(snare, beats = 1.5)
    playNote(snare)

    moveTo(0) # reset playhead

    # lay down hi-hats
    for i in range(16):
        playNote(hat, beats = 0.5)
```

- ∗ 2.15.4. Reggae. A common reggae beat is *1-drop beat*, which gets its name due to fact there's no hit on 1st beat. Rather, accent is on 3rd beat which contributes to strong backbeat & laid back feel in reggae. Using swung 8th notes for our hi-hats, & adding an open hi-hat hit on very last note to add texture. 1st hit is held for $\frac{2}{3}$ of beat & 2nd for $\frac{1}{3}$. Both our kick & snare hit on 3rd beat. Can find this example at https://tunepad.com/examples/reggae-beat.

```
    # define instrument variables
    kick = 0
    snare = 2
    hat = 4
    open_hat = 5

    # lay down kick and snare together
    rest(2)
    playNote([kick, snare])

    moveTo(0) # reset playhead

    # lay down swung hi-hat pattern
    for i in range(3):
        playNote(hat, 2.0 / 3) # two-thirds
        playNote(hat, 1.0 / 3) # one-third

    playNote(hat, 2.0 / 3)
    playNote(open_hat, 1.0 / 3)
```

- ∗ 2.15.5. Other common patterns. Here are a few other drum patterns in different genres that you can try coding for yourself. Hip-hop (late-1990s) 90 bpm, Hip-hop (mid-2000s) 85 bpm, basic pop/rock 130 bpm, Trap (mid-2010s) 130 bpm (double dots mean stuttered hi-hats), pop/hip-hop 70 bpm, West coast beat (late 2010s) 100 bpm, Dance/EDM/Hip-hop (circa 1982) 130 bpm, Hip-hop (mid-1990s) 85 bpm.

○ 2.16. Drum sequencers. A drum sequencer is a tool for creating drum patterns. Early sequencers like Roland 808 were physical pieces of hardward. Now most people use software-based sequencers, although basic principles are the same: Sequencers look like a grid with rows for different drum sounds & columns for short time slices (usually 16th notes or 32nd notes).

TunePad includes a drum sequencer (Fig. 2.15: TunePad composer interface provides drum & bass sequencers.) that can be helpful for playing around with different rhythmic ideas tunepad.com/composer. Can add drum sounds at different time slices by clicking on gray squares of grid, & once have a pattern you like you can convert it into Python code. When converting a drum sequencer pattern to code, it can be helpful to code column by column instead row by row. What that means: we work left to right across drum pattern. For each column, look at all sounds that hit at that time slice. Can then cue up each of those sounds using a simple `playNote` statement. A quick example pattern.

Look at 1st column & see there's a single kick drum.

```
playNote(0, beats = 0.25)
```

Look at 2nd column & see that it's empty, so rest:

```
rest(0.25)
```

3rd column includes both a hat (note 4) & a kick (note 0). To play these together, can use a single `playNote` command with both sounds enclosed in square brackets like this:

```
playNote([ 0, 4 ], beats = 0.25)
```

A special Python structure: list – a convenient way to play more than 1 sound at same time. If keep going with this column-by-column strategy, complete code:

```
playNote(0, beats = 0.25)
```

```
rest(0.25)
playNote([ 0, 4 ], beats = 0.25)
rest(0.25) # kick + hat
playNote(2, beats = 0.25)
rest(0.25)
playNote(4, beats = 0.25)
playNote(10, beats = 0.25)
rest(0.25)
playNote(10, beats = 0.25)
playNote([ 0, 4 ], beats = 0.25) # kick + hat
playNote(0, beats = 0.25)
playNote(2, beats = 0.25)
rest(0.25)
playNote(4, beats = 0.25)
rest(0.25)
```

Coding column by column can be a little quicker & produces more compact code.

Note 1. *Term beat can also refer to main groove in a dance track ("drop beat") or instrumental music that accompanies vocals in a hip-hop track ("she produced a beat for a new artist") in addition to other meanings.*

- Interlude 2: Custom Trap Beat. In this interlude, run with skills you picked up in preceding chap to create a custom Trap beat. This beat will use kick drum, snare, & hi-hats. Can follow along online by visiting `https://tunepad.com/interlude/trap-beat`.

  1. **Step 1: Defining variables.** Start by logging into TunePad & creating a new project called "Custom Trap Beat". Add a new *Drums* instrument to your project. In this call, declare *variables* for your drum sounds.

     ```
     # variables for drums
     kick = 1
     snare = 2
     hat = 4
     ```

  2. **Step 2: Basic drum pattern.** Code for a basic drum pattern. Add this code to your Drum call after variables:

     ```
     # kick and snare
     playNote(kick, beats = 0.75)
     playNote(kick, beats = 0.25)
     playNote(snare, beats = 1.5)
     playNote(kick, beats = 0.75)
     playNote(kick, beats = 0.75)
     playNote(kick, beats = 2)
     playNote(snare, beats = 1)
     ```

     Break down each of these lines 1 by 1 [Table] When done, pattern should look sth like Fig. 2.16: Basic drum pattern.

  3. **Step 3: Add hi-hat rolls & stutters.** Now add a new `Drum Cell` to your project for hi-hat rolls & stutters. To add our hi-hat runs, 1st review `for loops` in Python Fig. 2.17: Declaring a for loop for hi-hat runs in Python.

     Indented block of code is run total number of times specified by *range* of loop. Try this example pattern in your project:

     ```
     for i in range(4):
         playNote(hat, beats = 0.25)

     for i in range(4):
         playNote(hat, beats = 0.25 / 2)

     for i in range(8):
         playNote(hat, beats = 0.25)

     for i in range(5):
         playNote(hat, beats = 0.25 / 5)

     playNote(hat, beats = 0.25)
     ```

     Your cell should now have a pattern like Fig. 2.18: Hi-hat stutter patterns.

4. **Step 4: Customize.** After trying example in Step 3, make up your own stutter pattern to go with your kick & snare drums. Can use any combination of beats, but make sure it adds up to a multiple of 4 beats so that your beat loops correctly! Here are a few for loops that play stutters at different speeds:

   ```
   # couplet
   for i in range(2):
       playNote(hat, beats = 0.25 / 2) # divide in half

   # triplet
   for i in range(3):
       playNote(hat, beats = 0.25 / 3) # divide into 3 parts

   # quad
   for i in range(4):
       playNote(hat, beats = 0.25 / 4) # divide into 4 parts

   # fifthlet?
   for i in range(5):
       playNote(hat, beats = 0.25 / 5) # divide into 5 parts
   ```

   Try out different instrument sounds by changing values of variables & switching to a different drum kit. Can also experiment with changing tempo. For more inspiration, this TunePad project has several popular hip-hop beat patterns that you can experiment with https://tunepad.com/interlude/drum-examples.

- 3. Pitch, harmony, & dissonance. Chap. 2 introduced basics of rhythm & how to use Python programming language to code beats with percussion sounds. In this chap, explore topics of pitch, harmony, & dissonance – or what happens when you bring tonal instruments & human voice into music. Start with physical properties of sound (including frequency, amplitude, & wavelength) & why different musical notes sound harmonious or dissonant when played together. Also talk about different ways to represent pitch, including frequency value, musical note names, & MIDI (Musical Instrument Digital Interface) note numbers that we can use with Python code & TunePad.

  – Chương 2 giới thiệu những điều cơ bản về nhịp điệu & cách sử dụng ngôn ngữ lập trình Python để mã hóa nhịp điệu với âm thanh bộ gõ. Trong chương này, hãy khám phá các chủ đề về cao độ, sự hòa hợp, & sự bất hòa – hoặc điều gì xảy ra khi bạn đưa nhạc cụ có âm & giọng nói của con người vào âm nhạc. Bắt đầu với các đặc tính vật lý của âm thanh (bao gồm tần số, biên độ, & bước sóng) & lý do tại sao các nốt nhạc khác nhau nghe có vẻ hòa hợp hay bất hòa khi chơi cùng nhau. Ngoài ra, hãy nói về các cách khác nhau để biểu diễn cao độ, bao gồm giá trị tần số, tên nốt nhạc, & số nốt MIDI (Giao diện kỹ thuật số nhạc cụ) mà chúng ta có thể sử dụng với mã Python & TunePad.

  ◦ 3.1. Sound Waves. All sound, no matter how simple or complex, is made up of waves of energy that travel through air, water, or some other physical medium. If could see a sound wave, it might look sth like ripples of water from a pebble dropped in a still pound. Pebble is like source of sound, & ripples are sound waves that expand outward in all directions. Any source of sound (car horns, cell phone rings, chirping birds, or a plucked guitar string) sends vibrating waves of air pressure out at around 343 meters per sec (speed of sound) from source. It's not that air molecules themselves travel from source of sound to our ears; it's that small localized movements in molecules create fluctuations in air pressure that propagate outward over long distances.

    – Mọi âm thanh, dù đơn giản hay phức tạp, đều được tạo thành từ các sóng năng lượng truyền qua không khí, nước hoặc 1 số môi trường vật lý khác. Nếu có thể nhìn thấy sóng âm, nó có thể trông giống như gợn sóng nước từ 1 viên sỏi thả vào 1 pound đứng yên. Viên sỏi giống như nguồn âm thanh, & gợn sóng là sóng âm lan ra ngoài theo mọi hướng. Bất kỳ nguồn âm thanh nào (tiếng còi xe, chuông điện thoại di động, tiếng chim hót hoặc dây đàn guitar gảy) đều phát ra sóng rung động của áp suất không khí với tốc độ khoảng 343 mét 1 giây (tốc độ âm thanh) từ nguồn. Không phải bản thân các phân tử không khí di chuyển từ nguồn âm thanh đến tai chúng ta; mà là các chuyển động cục bộ nhỏ trong các phân tử tạo ra sự dao động trong áp suất không khí lan truyền ra ngoài trên những khoảng cách xa.

    Once those waves reach human ear, they are captured by outer ear & funneled to a seashell-shaped muscle in inner ear called *cochlea*. This muscles has tiny hairs that resonate at different frequencies causing messages to get sent to brain that we interpret as sound.

    Remark 1 (Protect your hearing)**.** *As musicians or music producers, your sense of hearing is 1 of your most precious assets. Always wear ear protection when you're exposed to loud sustained sounds! Loud sounds can damage your inner ear permanently, meaning you can start to close your ability to hear.*

    All sound waves have following properties: *frequency, wavelength, & amplitude.*

  ◦ 3.2. Frequency. Frequency refers to number of times a complete waveform passes through a single point over a period of time or how fast wave is vibrating. It is measured by cycles per sec in a unit called *hertz* (Hz). 1 cycle per sec is equivalent to 1 Hz, & 1000 cycles are equivalent to 1000 Hz, or 1 kHz (prorounecd kilohertz). Higher frequency, higher pitch of sound.

    Fig. 3.1: Sound is made up of compression waves of air molecules that expand outward at a speed of around 343 m/s. Frequency of a sound wave refers to how fast it vibrates: amplitude refers to intensity of sound; & wavelength refers to length of 1 complete cycle of waveform.

- 3.3. Wavelength. Wavelength refers to length of 1 complete cycle of a wave in physical space. This is distance from 1 peak or zero crossing to next. Can't actually see sound waves, but wavelength can be calculated by dividing speed of sound (≈ 343 m/s) by its frequency. So, for pitch of a *Concert A* note (440 Hz), length of waveform would be ≈ 78 cm or 2.6 ft.

  $$\frac{343\text{ m/s}}{440\text{ Hz}} = 0.78\text{ m} = 78\text{ cm} = 2.56\text{ ft}.$$

  On most pianos, wavelength of lowest bass note is almost 40 feet long! In contrast, wavelength of highest note is only around 3 inches. $\boxed{\text{Longer wavelength, lower the note.}}$

  Lower frequency sound also tends to travel longer distances. Think of a car playing loud music. As it approaches, you can hear fat sound of a bass guitar or a kick drum long before you can hear other instruments. Using this property, people in West Africa were able to transmit detailed messages over long distances using a language of deep drum sounds. A drummer called a "carrier" would drum out a rhythmic pattern on a huge log drum that carried messages like "all people should gather at market place tomorrow morning". All those within hearing range, which under ideal conditions could be as far as 7 miles, would receive message.

  – Âm thanh tần số thấp hơn cũng có xu hướng truyền đi xa hơn. Hãy nghĩ đến 1 chiếc ô tô đang phát nhạc lớn. Khi nó đến gần, bạn có thể nghe thấy âm thanh to của 1 cây đàn ghi-ta bass hoặc trống đá rất lâu trước khi bạn có thể nghe thấy các nhạc cụ khác. Sử dụng đặc tính này, người dân ở Tây Phi có thể truyền tải các thông điệp chi tiết trên những khoảng cách xa bằng ngôn ngữ của âm thanh trống sâu. Một tay trống được gọi là "người mang" sẽ đánh 1 mẫu nhịp điệu trên 1 chiếc trống gỗ lớn mang theo các thông điệp như "tất cả mọi người nên tập trung tại chợ vào sáng mai". Tất cả những người trong phạm vi nghe được, trong điều kiện lý tưởng có thể cách xa tới 7 dặm, sẽ nhận được thông điệp.

- 3.4. Amplitude. Amplitude is related to volume of a sound, or how high peaks of waveform are Fig. 3.1. You can think of this as how much energy passes through a fixed amount of space over a fixed amount of time. Human ear perceives a vast range of sound levels, from sounds that are softer than a whisper to sounds that are louder than a pain-inducing jackhammer. In order to communicate volume of sound in a manageable way, music producers & engineers use a unit of loudness called *decibels* (dB). Whispered voice level might be 30 dB, while jackhammer sound would be about 110 dB. Loud noises > 120 dB can cause immediate harm to ears.

  – Biên độ liên quan đến âm lượng của âm thanh hoặc độ cao của các đỉnh sóng Hình 3.1. Bạn có thể nghĩ về điều này như lượng năng lượng đi qua 1 lượng không gian cố định trong 1 khoảng thời gian cố định. Tai người cảm nhận được 1 phạm vi rộng lớn các mức âm thanh, từ âm thanh nhẹ hơn tiếng thì thầm đến âm thanh to hơn tiếng búa khoan gây đau đớn. Để truyền đạt âm lượng âm thanh theo cách dễ quản lý, các nhà sản xuất âm nhạc & kỹ sư sử dụng 1 đơn vị độ lớn gọi là *decibel* (dB). Mức giọng nói thì thầm có thể là 30 dB, trong khi âm thanh của búa khoan sẽ là khoảng 110 dB. Tiếng ồn lớn > 120 dB có thể gây hại ngay lập tức cho tai.

- 3.5. Dynamics. Variation of amplitude levels from low to high within a musical composition is referred to as dynamics. Difference between softest sound to loudest sound is called *dynamic range* of music. You can look at *waveform* of an audio signal to get a quick sense for its dynamic range. In general, lower heights mean lower amplitude & higher heights mean higher amplitude. Loudness of a sound is also dependent on frequency. So, looking at a waveform alone won't tell you how loud sth will sound to listeners (Fig. 3.3: A waveform with varying amplitude).

  – Động lực học. Sự thay đổi mức biên độ từ thấp đến cao trong 1 bản nhạc được gọi là động lực học. Sự khác biệt giữa âm thanh nhỏ nhất đến âm thanh to nhất được gọi là *dynamic range* của âm nhạc. Bạn có thể xem *waveform* của tín hiệu âm thanh để có cảm nhận nhanh về dải động của nó. Nhìn chung, độ cao thấp hơn có nghĩa là biên độ thấp hơn & độ cao cao hơn có nghĩa là biên độ cao hơn. Độ to của âm thanh cũng phụ thuộc vào tần số. Vì vậy, chỉ xem dạng sóng sẽ không cho bạn biết âm thanh nào đó sẽ to như thế nào đối với người nghe (Hình 3.3: Dạng sóng có biên độ thay đổi).

- 3.6. Bandwidth. Bandwidth refers to range of frequencies present in audio. As in case of dynamic range, can think of this as difference between highest & lowest frequencies. Humans with good hearing can distinguish sounds between 20 Hz & 20000 Hz. Most audio formats designed for music support frequencies up to 22 kHz (pronounced 22 kilohertz or 22000 hertz) so that they can capture full range of human hearing.

  – Băng thông đề cập đến phạm vi tần số có trong âm thanh. Giống như trường hợp của phạm vi động, có thể coi đây là sự khác biệt giữa tần số cao nhất & thấp nhất. Con người có thính giác tốt có thể phân biệt âm thanh giữa 20 Hz & 20000 Hz. Hầu hết các định dạng âm thanh được thiết kế cho âm nhạc đều hỗ trợ tần số lên đến 22 kHz (phát âm là 22 kilohertz hoặc 220000 hertz) để chúng có thể thu được toàn bộ phạm vi thính giác của con người.

  Musical instruments naturally fall within range of human hearing at different places on frequency spectrum; this is referred to as instrument's bandwidth. *Instrument bandwidth* is important to music producers as they arrange a musical composition. In addition to quality of sound of instruments, those in different bandwidths can complement each other. Like a cello & a flute, or a bass & a saxophone. Music producers are keenly aware of influence of low- & high-frequency instruments on their listeners. Musical instruments in bass register are often foundation of composition, holding everything together.

  – Nhạc cụ tự nhiên nằm trong phạm vi nghe của con người ở các vị trí khác nhau trên phổ tần số; điều này được gọi là băng thông của nhạc cụ. *Băng thông nhạc cụ* rất quan trọng đối với các nhà sản xuất âm nhạc khi họ sắp xếp 1 bản nhạc. Ngoài chất lượng âm thanh của các nhạc cụ, những nhạc cụ có băng thông khác nhau có thể bổ sung cho nhau. Giống như đàn cello & sáo, hoặc đàn bass & saxophone. Các nhà sản xuất âm nhạc nhận thức sâu sắc về ảnh hưởng của các nhạc cụ có tần số thấp & cao đến người nghe của họ. Nhạc cụ có âm trầm thường là nền tảng của bản nhạc, giữ mọi thứ lại với nhau.

- 3.7. Pitch. Within spectrum of human hearing, specific frequencies, ranges of frequencies, & combinations of frequencies are essential for creating music. This sect covers some combinations of musical tones common in Western music culture. Then work in TunePad to try out different combinations & explore those relationships through well-known musical compositions.
  – Trong phổ thính giác của con người, các tần số cụ thể, phạm vi tần số, & sự kết hợp của các tần số là điều cần thiết để tạo ra âm nhạc. Giáo phái này bao gồm 1 số sự kết hợp của các giai điệu âm nhạc phổ biến trong văn hóa âm nhạc phương Tây. Sau đó, hãy làm việc trong TunePad để thử các sự kết hợp khác nhau & khám phá các mối quan hệ đó thông qua các tác phẩm âm nhạc nổi tiếng.

  While music producers & engineers often think in terms of frequencies (hertz), musicians use pitch & intervals to describe musical tones & relationships between them. Pitches are individual notes like F, G, A, B, C, D, E as seen on piano keyboard. Interval between each adjacent note on a traditional keyboard is called a half step or a semitone. These base pitches can also have *accidentals*. Accidentals are like modifiers to notes that raise or lower base pitch. A note with a sharp # applied has its pitch raised by a semitone, which a note with a flat ♭ applied is lowered by a semitone. Black notes on a piano are notes with accidentals. E.g., moving a C# (black key) is a half step. Moving directly from a C to a D (both white keys) is called a whole step. Moving from a B to a C or an E to an F is also a half step because there's no black key in between (Fig. 3.4: A half step is distance between 2 adjacent piano keys, measured in semitones.)

  – Trong khi các nhà sản xuất âm nhạc & kỹ sư thường nghĩ theo tần số (hertz), thì các nhạc sĩ sử dụng cao độ & khoảng cách để mô tả các cung bậc âm nhạc & mối quan hệ giữa chúng. Cao độ là các nốt riêng lẻ như F, G, A, B, C, D, E như thấy trên bàn phím piano. Khoảng cách giữa mỗi nốt liền kề trên bàn phím truyền thống được gọi là nửa cung hoặc nửa cung. Các cao độ cơ bản này cũng có thể có *dấu hóa ngẫu nhiên*. Dấu hóa ngẫu nhiên giống như các dấu hiệu bổ nghĩa cho các nốt làm tăng hoặc giảm cao độ cơ bản. Một nốt có dấu thăng # được áp dụng có cao độ được tăng lên 1 nửa cung, trong khi 1 nốt có dấu giáng ♭ được áp dụng sẽ hạ xuống 1 nửa cung. Các nốt đen trên đàn piano là các nốt có dấu hóa ngẫu nhiên. Ví dụ, di chuyển 1 C# (phím đen) là nửa cung. Di chuyển trực tiếp từ C sang D (cả hai đều là phím trắng) được gọi là 1 cung trọn vẹn. Di chuyển từ B sang C hoặc từ E sang F cũng là nửa cung vì không có phím đen nào ở giữa (Hình 3.4: Nửa cung là khoảng cách giữa 2 phím đàn piano liền kề, được đo bằng nửa cung.)

- 3.8. Musical Instrument Digital Interface. 1 takeaway from prev sect: note names are confusing. There are multiple names for same pitch (G# is same as A♭), & note names are repeated every octave. To help make things less ambiguous, computers & digital musical instruments use a standardized format called *MIDI*, which stands for Musical Instrument Digital Interface. MIDI is a protocol, or set of rules, for how digital musical instruments communicate. Digital musical instruments send message to your computer or to other musical instruments. Typical MIDI controllers look like piano keyboards or drum pads but can take many other forms as well. When play a MIDI instrument, it sends information about a note's pitch, timing, & volume along with other messages about vibrato, pitch bend, pressure, panning, & clock signals. This table show 2 octaves of notes with their typical frequency values [Table]. Appendix contains a complete table with note names, frequency values, & MIDI numbers.

  TunePad uses MIDI numbers to designate pitch. To play a C0, lowest pitch on TunePad keyboard, use code `playNote(12)`. To play a C4, a middle C in center of an 880key piano, use code `playNote(60)`. MIDI notes go all way up to note G9 with note value 127.

  Now experiment with pitch in TunePad. Try creating a new piano instrument in TunePad & adding this code:

  ```
  # code for first piano cell
  playNote(48)
  playNote(55)
  playNote(60)
  playNote(55)
  ```

  This program plays 4 notes: 48 is a C, 55 is a G, & 60 is a middle C. Now add a 2nd piano instrument to same project so that you have 2 cells. Add this code to 2nd cell:

  ```
  # code for second piano cell
  playNote(72, beats = 4) # C5
  playNote(79, beats = 4) # G5
  playNote(76, beats = 4) # E5
  playNote(79, beats = 4) # G5
  ```

  This Python program looks similar to 1st one, but we've changed length of each note using *beats* parameter. In this case, asking TunePad to play 4 notes, each 4 beats long. Try playing both piano parts at same time. Can also make our notes shorter instead of longer. Add a 3rd piano instrument with notes that are each 1 half beat long. Try playing all 3 pianos together.

  ```
  # code for third piano cell
  playNote(36, beats = 0.5)
  playNote(36, beats = 0.5)
  playNote(43, beats = 0.5)
  playNote(43, beats = 0.5)
  ```

```
playNote(48, beats = 0.5)
playNote(48, beats = 0.5)
playNote(43, beats = 0.5)
playNote(43, beats = 0.5)
```

- 3.9. Harmony. *Harmony* in music can be defined as a combination of notes that, when played together, have a pleasing sound. Although opinions about what sounds good in music are highly subjective, certain combinations of notes played together can elicit predictable psychological responses – some combinations of notes sound *harmonious* while others some *discordant*. Musicians use this phenomenon to create an emotional tone for their compositions.

  – *Harmony* trong âm nhạc có thể được định nghĩa là sự kết hợp các nốt nhạc, khi chơi cùng nhau, tạo ra âm thanh dễ chịu. Mặc dù ý kiến về những gì nghe hay trong âm nhạc là rất chủ quan, nhưng 1 số sự kết hợp các nốt nhạc chơi cùng nhau có thể gợi ra những phản ứng tâm lý có thể dự đoán được – 1 số sự kết hợp các nốt nhạc nghe *hài hòa* trong khi 1 số khác lại *không hài hòa*. Các nhạc sĩ sử dụng hiện tượng này để tạo ra giai điệu cảm xúc cho các sáng tác của họ.

  In Western music, much of our conception of pitch is built on different mathematical ratios. Consider string of an instrument like a guitar or violin. Plucking open A (2nd lowest) string plays an A, which has a frequency of 110 Hz. Now if touch string at its midpoint, dividing it in half, still hear an A an octave above previous one – twice frequency of 1st note, or 220 Hz. If touch string $\frac{1}{3}$ of way down & pluck it, result is an E above higher A. This E is exactly 3 times our original frequency, or 330 Hz. Likewise, dividing string into 4ths multiplies original frequency by 4. Can continue this division on string as follows Fig. 3.5: harmonic series.

  – Trong âm nhạc phương Tây, phần lớn quan niệm của chúng ta về cao độ được xây dựng dựa trên các tỷ lệ toán học khác nhau. Hãy xem xét dây của 1 nhạc cụ như đàn ghi-ta hoặc đàn violin. Gảy mở dây A (dây thấp thứ 2) sẽ tạo ra nốt A, có tần số 110 Hz. Bây giờ nếu chạm vào dây ở điểm giữa của nó, chia nó thành hai nửa, vẫn nghe thấy nốt A cao hơn 1 quãng tám so với nốt trước đó – gấp đôi tần số của nốt đầu tiên, hoặc 220 Hz. Nếu chạm vào dây $\frac{1}{3}$ xuống 1 khoảng & gảy nó, kết quả là nốt E cao hơn nốt A cao hơn. Nốt E này chính xác gấp 3 lần tần số ban đầu của chúng ta, hoặc 330 Hz. Tương tự như vậy, chia dây thành 4 quãng sẽ nhân tần số ban đầu với 4. Có thể tiếp tục phép chia này trên dây như sau Hình 3.5: chuỗi điều hòa.

  Resulting sequence of ascending pitches this produces is known as *harmonic series*. If 2 notes have a harmonic relationship, i.e., 2 frequencies s.t. result is a whole number. Harmonic series is simply set of frequencies that have a harmonic relationship to a *fundamental pitch* (initial note). Our initial experiment with string illustrates this relationship.

  To find frequencies that make up harmonic series for a given pitch, multiply its frequency by set of whole numbers. For A1, which is 55 Hz, 1st 8 harmonics would be [Table].

  In table, MIDI value is given for each harmonic of A1. Notice these values are given in decimal format. In TunePad, `playNote` accepts both whole & decimal values. Whole numbers are a data type referred to as *integer* values, or just as *ints*. Decimals are a separate data type referred to as *floating point* values, or just *floats*.

  Listen to an example https://tunepad.com/examples/harmonic-series. Notes with frequencies that form simple ratios, e.g. 2:1, 3:2, 4:3, or 5:4, tend to sound good together. E.g., can take note A4 (440 Hz) & add a frequency that is 1.5 times its value, giving us an E4 (660 Hz). This results in a ratio of 3:2 & a pleasant sound. However, if add a frequency that is 1.3 times value of 440 Hz, end up with 572 Hz, which creates a not-so-pleasant combination of tones. It's not an accident that there is no corresponding musical note to 572 Hz on piano keyboard.

- 3.10. Intervals. In music, an *interval* is distance between 2 notes. These notes can either be played simultaneously or not. If they are played simultaneously, pitches are called a *dyad* or a *chord*. Otherwise, they are a *melodic* interval. An interval is always measured from lowest note. Intervals have 2 different components: *generic interval* & quality. Generic interval is distance from 1 note of a scale to another; this can also be described as number of letter names between 2 notes, including both notes in question. E.g., generic interval of C4 & E4 has C4, D4, & E4 in between. That's 3 notes, so we have a 3rd. Generic interval between F#3 & G3 is a second. Generic interval between G2 & G3 in an 8th – also known as an octave. Quality can be 1 of 5 options: Perfect, Major, Minor, Augmented, or Diminished. Each quality has a distinct sound & can generate different emotional responses. Some common intervals in music along with their frequency ratios & half steps. [Table]

  These intervals are based on harmonic series, but this isn't exactly how most instruments are tuned. Talk more about this below. Also, naming of ratios (5th, 4th, Major 3rd, etc.) will make more sense in Chap. 5 where talk about scales & keys. Notice that there's 1 particularly nasty-looking ratio called *Tritone interval* (45:32). This interval has historically been referred to as *Devil in Music* & was frequently avoided in music composition for its dissonant qualities.

  – Các khoảng này dựa trên chuỗi hài hòa, nhưng đây không phải là cách chính xác mà hầu hết các nhạc cụ được lên dây. Hãy nói thêm về điều này bên dưới. Ngoài ra, việc đặt tên cho các tỷ lệ (5, 4, 3 trưởng, v.v.) sẽ hợp lý hơn trong Chương 5, nơi nói về các thang âm & cung. Lưu ý rằng có 1 tỷ lệ trông đặc biệt khó chịu được gọi là *Quãng ba cung* (45:32). Khoảng này trước đây được gọi là *Devil in Music* & thường bị tránh trong sáng tác nhạc vì tính chất bất hòa của nó.

  1 of simplest & most common intervals is octave, which has a frequency ratio of 2 to 1 (2:1) – i.e., higher pitch completes 2 cycles in same amount of time that lower pitch completes 1 full cycle. Notes that are octave intervals from 1 another have same letter name & are grouped together on a piano keyboard. Notice repeating patterns where C is highlighted note, C3, C4, C5 (Fig. 3.4: A half step is distance between 2 adjacent piano keys, measured in semitones.). To illustrate, can begin with *middle C* (C4), which is ≈ 262 Hz, & then move to a C5, which is an octave above it at 524 Hz. Can see C5 is double C4

frequency forming octave ratio, 2:1. Waveforms representing 2 notes forming this octave are plotted in Fig. 3.6: 2 waves at an interval of an octave. Can see that for every single complete cycle of 262 Hz wave, C4, there are 2 full cycles of waveform for octave above it, 524 Hz C5. Easier to count cycles if look at 0-crossings.

What does an octave look like in code? As have seen, each note on piano keyboard is a half step, & there are 12 half steps between octaves. Try counting notes between C4 & C5. Remember, black keys count!

TunePad tracks notes on keyboard by half steps, so can easily play any octave interval without having to figure out exact note number. E.g., this code plays a middle C (60) & a C 1 octave higher.

```
note = 60
playNote(note)
playNote(note + 12)
```

This code assigned number 60 to variable `note` on 1st line. 3rd line played a note 1 octave higher by adding 12 to original `note` variable (not 72 is played). Expanding on this, can substitute any number you want for `note` & generate an octave above it by adding 12.

Octaves sound good together in music & are used in many popular songs. E.g., in song *Over the Rainbow* composed by Harold Arlen from *Wizard of Oz*, beginning 2 notes are an octave apart.

```
# First two bars of "Over the Rainbow"
# Composed by Henry Arlen
playNote(60, beats = 2) # note C4
playNote(60 + 12, beats = 2) # note C5
playNote(71, beats = 1)
playNote(67, beats = 0.5)
playNote(69, beats = 0.5)
playNote(71, beats = 0.5)
rest(0.5)
playNote(72, beats = 1)
```

Try this example at https://tunepad.com/examples/rainbow.

Another interval relationship important to Western music is ratio of 3:2, also known as perfect 5th, which has 7 half steps between notes. With this interval ratio, there are 3 complete cycles of higher frequency for every 2 periods of lower frequency (Fig. 3.7: Ratio between note C 262 Hz & note G 393 Hz is considered a perfect 5th.)

Henry Mancini uses a perfect 5th (G3 392 Hz & D4 587 Hz) in 1st 2 notes in melody for song *Moon River*. Code 1st few bars of *Moon River* using a variable called `root_note` to set starting note. This allows us flexibility to easily play song beginning from any note on keyboard & relationship between notes stays same no matter which note you start with. Try changing value of variable `root_note` to another MIDI note. This can come in handy when you are composing for a singer who would rather have song in another key or octave.

```
# First bars of "Moon River"
# Composed by Henry Mancini
root note = 55
playNote(root note, beats = 3)
playNote(root note + 7, beats = 1)
playNote(root note + 5, beats = 2)
playNote(root note + 4, beats = 1.5)
playNote(root note + 2, beats = 0.5)
playNote(root note, beats = 0.5)
playNote(root note - 2, beats = 0.5)
playNote(root note, beats = 2)
```

See this example at https://tunepad.com/examples/moon.

- 3.11. Dissonance. Dissonance refers to combinations of notes which when combined have an unpleasant sound that creates tension. Interval of a minor second (or 1 half step) is a complex frequency ratio of about 9.5:1. This combination gives you a sense of suspense. Can hear effect of dissonance used in composition by John Willimas for movie *Jaws*. Use `for` loop for this example, as 2 notes are repeated.

```
# bass line for the theme from Jaws
# composed by John Williams
for i in range(8):
    playNote(40, beats = 0.5) # E2
    playNote(41, beats = 0.5) # F2
```

Intervals that are dissonant are unstable, leaving listener with impression that notes *want* to move elsewhere to resolve to more stable or *consonant* intervals.

Can try this example in TunePad to hear how notes that are 1 half step apart crunch when played together.

```
# half step - the notes are just 1 number away
playNote(41, beats=1, sustain=3)
playNote(42, beats=1, sustain=2)
rest(2)
playNote([41, 42], beats=4)
rest(2)

# whole step - these notes are 2 numbers away
playNote(41, beats=1, sustain=3)
playNote(43, beats=1, sustain=2)
rest(2)
playNote([41, 43], beats=4)
rest(2)
```

Listen to this example https://tunepad.com/examples/dissonance.

Another example of use of dissonant intervals comes from horror movie *Halloween* (1978). Theme song by John Carpenter creates a sense of suspense & deep unease with use of dissonant intervals e.g. Tritone (ratio 45:32).

- 3.12. Temperaments & Tuning. Follow along at https://tunepad.com/examples/temperaments .

Intervals in prev sects were based on ratios called perfect or pure intervals. Waves of so-called perfect intervals align at a simple integer ratio. If 2 tones form a perfect interval, it will result in a louder sound, as amplitudes are added. If 1 of tones is out of tune, then there will be interference between 2 waves. This interference manifests as an audible rhythmic swelling or "wah-wah" between waves, which we call *beating*. Farther 2 tones are from being perfect, faster beating. If tones are apart far enough, might even hear this beating as a 3rd tone, called a *combination tone*. Pure & impure intervals are not a value judgment but a description of natural phenomena.

Using notes based on these simple ratios seems to make a lot of sense – it's based on simple mathematical relationships that we know sound good to human ear. But, it turns out: quickly run into problems using this system when start trying to tune an instrument like a piano. E.g., say trying to tune an A4 against a fixed lower tone on a keyboard using pure ratios. If tuning this A4 against an F4 at ≈ 349 Hz, our intervals form a major 3rd at a 5:4 ratio of frequencies. This results in our A4 being ≈ 436.26 Hz. But, if tune our A4 against an F#4 at 370 Hz, this produces a minor 3rd, which is at a 6:5 ratio of frequencies. Now our A4 is 444 Hz instead of 436.25 Hz! How can it be that same note maps to different frequencies?

Question of how to map frequency – of which there are endless possible values – to a fine set of notes means that we have to both arbitrarily choose a starting point & also decide at what intervals to increment. This is basis for what are called *temperaments*. Temperaments are systems that define sizes of different intervals – how tones relate to 1 another. In choosing tones in an octave, must compromise between our melodic intervals & our harmony. Ideally, want a system with as consistent melodic intervals & that is as close to perfect harmonic intervals as possible. In a system based on perfect ratios – also referred to *Just Intonation* – divisions, or semitones, of an octave are not evenly distributed. I.e., there are unique sets of tunings for every note we choose as base note of our octave. Just Intonation also does not form a closed loop of an octave. This is getting into weeds a bit, but if derive each note's frequency by tuning ratio of a perfect 5th (3:2) from prev note, do not end up at same place 1 octave higher. In fact, tuning ratio of 3:2 12 times brings us back to our exact starting note only after 7 octaves. Because Just Intonation has too many mathematical snares to be represented by 12 notes of keyboard, it's not a stable tuning system & not a temperament. By definition, a temperament is a calculated deviation from Just Intonation that maps each note to exactly 1 frequency while still getting as close as possible to pure intervals.

Most contemporary music, including TunePad, is based on a system called Equal Temperament. Octave at a pure 2:1 ratio serves as foundation, which is then divided into 12 equal half steps. Most often, Western harmony is built primarily from 3rds, 5ths, & octaves. Every octave (& unison) is a pure interval in Equal Temperament. Perfect 4ths & 5ths are *nearly* pure intervals. Major & minor 3rds are quite far from perfect, but because we have grown so accustomed to hearing these intervals, they do not sound off to our ears. Because Equal Temperament is, well, equal, every chord will have same sound in every key. Each semitone is equally sized, & every note maps to exactly 1 frequency. Furthermore, each semitone is divided into 100 cents, which we can use to further specify intonation. With our intervals decided, now only have to choose a starting pitch from which to tune others. Most of time in North America, system is aligned to A440, i.e. A4 is equal to exactly 440 Hz.

Keyboard instruments have fixed pitch, while singers & instruments e.g. violin or flute have flexible tuning. In acoustic performance, pitch can vary due to many factors. No instrument is perfectly in tune. Tuning can be affected by factors e.g. air pressure & temperature. Even a performer's physiology can affect tuning. Often, performers will tune harmonies using Just Intonation s.t. a chord uses pure intervals & is more pleasing. Many musicians will do this without even being aware that they are doing it – Just Intonation just *feels* in tune.

Important to remember that decision to tune to A440 & to divide octave into 12 equal semitones is only 1 possibility in response to debates about musical tuning that date back thousands of years, & it's only 1 of a myriad of ways that music can be tuned. There are many alternate tuning systems, both historical & contemporary from both Western & non-Western cultures, which are still in use today.

– Điều quan trọng cần nhớ là quyết định lên dây A440 & chia quãng tám thành 12 nửa cung bằng nhau chỉ là 1 khả năng để đáp lại các cuộc tranh luận về cách lên dây nhạc có từ hàng ngàn năm trước, & đó chỉ là 1 trong vô số cách để lên dây nhạc. Có nhiều hệ thống lên dây thay thế, cả lịch sử & đương đại từ cả nền văn hóa phương Tây & không phải phương Tây, vẫn được sử dụng cho đến ngày nay.

- Interlude 3: Melodies & Lists. For this interlude, code a short sect of a remix of BEETHOVEN's *Für Elise* created by artist & YouTuber KYLE EXUM (Bassthoven, 2020). Because this song has a more intricate melody, learn how to play sequences of notes written out as Python *lists*. Talk more about lists in next chap, but for now, can think of them as a way to hold > 1 note in a single variable.

1. **Step 1: Variables.** Create a new Keyboard instrument & add some variables for our different note names.

```
A = 69 # set variable A equal to 69
B = 71
C = 72
D = 74
E = 76
Eb = 75 # E flat
Gs = 68 # G sharp
_ = None
```

   Last line is a little strange. It defines a variable called `_`
   - In Python, underscore `_` character is a valid variable name.
   - Set this variable to have a special value called `None`.
   - Calling `playNote` with this value is same thing as a rest. It plays nothing.

2. **Step 2: Phrases.**
   - For this song, going to define 4 musical phrases that get repeated to make melody.
   - Each phrase gets its own variable.
   - Each variable will hold lists of notes in order they should be played.
   - You create a Python list by enclosing variables inside of square brackets.
   - Use underscore character `_` mean play nothing.
   - Sometimes subtract 12 from a note, i.e., to play note on octave lower.

```
# four basic phrases that repeat throughout
p1 = [ E, Eb, E, Eb, E, B, D, C, A, _, _, _ ]
p2 = [ A, C - 12, E - 12, A, B, _, _, _ ]
p3 = [ B, E - 12, Gs, B, C, _, _, _, C, _, _, _ ]
p4 = [ B, E - 12, C, B, A, _, _, _, A, _, _, _ ]
```

3. **Step 3: Playing Phrases.** Now have defined our variables, can start to play melody. 1 way to do this: use a Python *for loop* to iterate through every note. 1 cool thing about Python: can join lists together using plus sign +. Here's what everything looks like together.

```
_ = None
A = 69
B = 71
C = 72
D = 74
E = 76
Eb = E - 1 # E flat
Gs = A - 1 # G sharp

# 4 basic phrases that repeat throughout
p1 = [ E, Eb, E, Eb, E, B, D, C, A, _, _, _ ]
p2 = [ A, C - 12, E - 12, A, B, _, _, _ ]
p3 = [ B, E - 12, Gs, B, C, _, _, _, C, _, _, _ ]
p4 = [ B, E - 12, C, B, A, _, _, _, A, _, _, _ ]
p5 = [ A,_,_,_,A,_,_,_,A,_, _, _, A, _, _, _ ]

for note in p1 + p5 + p2 + p3 + p1 + p2 + p4:
    playNote(note, beats = 0.5)

for note in p1 + p2 + p3 + p1 + p2 + p4:
    playNote(note, beats = 0.5)
```

4. **Step 4: Bass!** Add a Bass to your project & change voice to `808 Bass`. Can copy code below for bass pattern Fig. 3.8: Select 808 Bass voice.
5. **Step 5: Drums.** To finish up, layer in a simple drum pattern that works well with melody. Create a new `Drum instrument` & add this code.

```
rest(12)
for i in range(4):
    playNote(21)
    rest(2)
    playNote(21)
    rest(1)
    playNote(16, beats = 0.5)
    rest(1)
    playNote(16, beats = 0.5)
    rest(1)
    playNote(21)
    rest(2)
    playNote(21)
    rest(1)
    playNote(28, beats = 0.5)
    rest(1)
    playNote

for i in range(16):
    playNote(0)
    playNote(2, beats = 0.5)
    playNote(2, beats = 0.5)
    playNote(10)
    playNote(0)
```

Can try this project online https://tunepad.com/interlude/bassthoven.

- 4. Chords – Hợp âm. Chords are an essential building block of musical compositions. Skillful use of chords can set foundation of a song & create a sense of emotional movement. However, even though basic ideas behind chords are easy to understand, there's an overwhelming amount of terminology & technical detail that can take years to learn. Using code helps us cut through layers of complicated terminology to reveal elegant structures beneath. With code, chords are nothing more than lists of numbers that follow consistent patterns. Work through chords using Python *lists & functions*. Learn some of traditional music terminology & what it means, but also build your own toolkit of computer code to use for new compositions.

  - 4.1. Chords. Can follow along with interactive online examples at https://tunepad.com/examples/chord-basics. In Chap. 3, introduced idea of harmony & dissonance. ≥ 2 notes have a harmonic relationship if their frequencies have integer ratios. E.g., when 2 notes are 1 octave apart, higher note vibrates exactly 2 complete cycles for every 1 cycle of lower note (a 2:1 ratio). When 2 notes are a 5th apart, their frequencies have a 3:2 ratio. Higher note vibrates exactly 3 times for every 2 complete cycles of lower note.

    Building on this idea of harmonic relationships between notes, a *chord* is more than 1 note played together at same time. In Python can think of a chord as a *list* of numbers representing MIDI (Musical Instrument Digital Interface) note values. E.g., this code plays a C major chord in TunePad.

    ```
    Cmaj = [ 48, 52, 55 ] # notes C, E, G
    playNote(Cmaj)
    ```

    Here `Cmaj` is a variable. Instead of assigning that variable to a single number, we're assigning it a *list* of numbers. In Python, a list is a set of values enclosed in square brackets & separated by commas. Then on 2nd line we play all 3 notes together using `Cmaj` variable Fig. 4.1: C major chord with MIDI note numbers.

    Can also play same chord using just a single line of code where pass list of numbers directly to `playNote` function.

    ```
    playNote([ 48, 52, 55 ])
    ```

    But, using variable is nice because it helps make our code easier to read & understand. Here are a few other chord examples:

```
Cmaj = [ 48, 52, 55 ] # C major chord
Fmaj = [ 53, 57, 60 ] # F major chord
Gmaj = [ 55, 59, 62 ] # G major chord
```

A chord's name comes from 2 parts. 1st part is *root* note of chord – usually 1st note in list. E.g., a F major chord starts with note 53 (or an F on piano keyboard). & G major chord starts with note 55, a G on keyboard.

2nd part of a chord's name is its type or *quality*. In our examples, `Cmaj, Fmaj, Gmaj` are all *major* chords. Later in this chap review several common chord types & how to create them in code. Each chord type has a consistent pattern. E.g., all major chords follow exact same pattern: take root note, add 4 to get 2nd note, & then add 7 to get last note. Can build a major chord up from any base note you want as long as it follows this same pattern Fig. 4.2: Creating chords as lists of numbers in Python. Each major chord follows same pattern.

Another way to write this in code: define a single root note variable & then create chord based on root:

```
root = 48
playNote([ root, root + 4, root + 7 ]) # C major
root = 52
playNote([ root, root + 4, root + 7 ]) # F major
```

1 thing to notice about this code: 1st 2 lines & last 2 lines are almost identical. Just changing root note value. In fact, all we need to make any chord we want is its root note & pattern that defines its quality. Once we know patterns, rest is easy. But, it would be tedious & error prone to write out `[root, root + 4, root + 7]` every time we wanted to use a major chord. Fortunately, Python gives us a powerful tool for exactly this kind of situation: *user-defined functions*.

- 4.2. User-defined functions. In 1st few chaps of this book, using functions to play music: `playNote, rest, moveTo` are all functions provided to TunePad. When want to use a function, just type its name & list parameters to send it inside parentheses. With Python, can also create our own functions to build up a musical toolkit. Creating functions also helps make code shorter & easier to understand because we'll be able to use same segments of code over & over again without having to copy & paste. A quick example that creates a major chord based on a root note.

```
def majorChord(root):
    chord = [ root, root + 4, root + 7 ]
    return chord
```

Now that have defined function, can use it as a shortcut in TunePad.

```
Cmaj = majorChord(48)
Fmaj = majorChord(53)
playNote(Cmaj, beats = 2)
playNote(Fmaj, beats = 2)
```

There's a lot going on with these few lines of code. Break it down line by line.

- Line 1 starts with `def` keyword. This is short for "define", & it tells Python that we're about to define a function.
- Next is name of function we're defining. In this case, calling it `majorChord`, but we could use any name want as long as it follows Python's naming rules.
- After function name, need to list out all of function's *parameters* enclosed inside parentheses. In this case, there's only 1 parameter called *root*. If need > 1 parameter, separate each parameter with commas. Can think of parameters as special kinds of variables that are only usable inside of a function.
- After parameter list, need colon character :. This tells Python: *body* of function is coming next Fig. 4.3: How to declare a user-defined function in Python.
- Line 2 starts body of function. Here just creating a variable called *chord* & assigning it to a list of numbers that define a major chord. Numbers are root note, root note +4, & root note +7. An important thing to notice: this line of code is intended by 4 spaces. Just like with loops, indentation tells Python: these lines are part of function body. They're considered to be inside function, not outside.
- Line 3 is also indented by 4 spaces because it's also part of function body. This line uses special `return` keyword to say what value function produces. In this case, returning `chord` variable, list of 3 numbers that make up our major chord. When Python gets to `return` keyword, it immediately exits function & returns value given on that line.
- Also possible to define a function with no return value. In this case, Python provides a special return value called `None`.

That's it for our function def. Now can see how it's used on lines 4–7. Notice can call our new function multiple times to generate different chords, letting us reuse code to create more readable & elegant programs. In rest of chap, use this same template to begin to build up a library of chord types, each with its own function.

Can define variables inside a function just like you might otherwise. A variable's *scope* refers to where we can use this variable or function. Can think of this as level of indentation for a function. Variables defined within def of a function can only be used in that function; their scope is within that function. Following code will result in an error on line 4.

```
def majorChord(root):
    chord = [ root, root + 4, root + 7 ]
majorChord(60)
playNote(chord) # ERROR
```

`chord` variable lives only in our `majorChord` function. Can refer to this as a local variable. Alternatively, there are global variables which can be used anywhere after they've been defined. In code below, `root, chord` are global variables, & therefore can be freely used in body of any functions:

```
root = 60
chord = [ root, root + 4, root + 7 ]
def songPar1():
    playNote(chord)
    playNote(root)
songPart1()
```

Before move on, let's define 1 more function that shows how we can use our new functions inside of other functions:

```
def playMajorChord(root, duration):
    chord = majorChord(root)
    playNote(chord, beats = duration)
```

This function uses our `majorChord` function to both build a chord from a root note & then calls `playNote` to play chord. This new `playMajorChord` function takes 2 parameters, `root` note & a `duration` that says how long to play note.

Note 2. *By convention, Python function & variable names use lowercase letters, with different words separated by underscore character, e.g.,* `play_major_chord`*. This style is referred to as "*`snake_case`*". However, this book uses a different style called "camelCase" where names start with a lowercase letter & uppercase letters are used to start new words (as in* `playMajorChord`*). Use camelCase in this book to be consistent with other programming language conventions e.g. JavaScript, but should feel free to use* `snake_case` *for your own Python programs if want to.*

- 4.3. Common chord types. This sect reviews some of most commonly used chord types in modern musical genres. For each chord, provide pattern of numbers that defines its quality, an example of a chord of that type on piano keyboard, & a TunePad function that generates chords from a root note. Also describe chord in terms of musical intervals. Names for intervals can be a little confusing, especially when combined with note names or MIDI note values.

  Common chord types/qualities.

  - Major triad: Major 7th: Suspended 2
  - Minor: Minor 7th: Suspended 4
  - Diminished: Dominant 7th: Augmented

  To hear these chords in action, go to https://tunepad.com/examples/chord-functions.

  - 4.3.1. Major triad. Major chords are commonly described as cheerful & happy. They consist of a root note, root + 5, & root + 7. In music theory, 2nd note of a major triad is called a *major 3rd*, & 3rd note is called a *perfect 5th*. This can be referred to as a *triad* due to fact there there are 3 distinct notes Fig. 4.4: C major chord.
    - Pattern: [0, 4, 7]
    - Intervals: Major 3rd, Perfect 5th
    - Notation: C major, CMaj, CM, C
    - Python Function:

      ```
      def majorChord(root):
          return [ root, root + 4, root + 7 ]
      ```

  - 4.3.2. Minor triad. Minor chords convey more of a somber tone. They're very similar to major chords except that 2nd note adds 3 to root note instead of 4. In music theory, 2nd note is called a *minor 3rd* (instead of a major 3rd). But, even with this small change, difference in mood is dramatic Fig. 4.5: D minor chord.
    - Pattern: [0, 3, 7]
    - Intervals: Minor 3rd, Perfect 5th
    - Notation: D minor, Dmin, Dm C
    - Python Function:

      ```
      def minorChord(root):
          return [ root, root + 3, root + 7 ]
      ```

∗ 4.3.3. Diminished triad. Diminished chords instill tension & instability in music. They're often used as a way to transition between chords in a progression. There are a few different types of diminished chords, but simplest, 3-note variety, is almost identical to minor triad except last note is decreased (diminished) by 1. This is called a *diminished 5th interval* Fig. 4.6: B diminished chord.

· Pattern: [0, 3, 6]

· Intervals: Minor 3rd, Diminished 5th

· Notation: B dim, B°

· Python Function:

```
def diminishedTriad(root):
    return [ root, root + 3, root + 6 ]
```

∗ 4.3.4. Major 7th. Major 7th chord starts with a major triad & then adds a 4th note to end of list (root + 11). This addition is called a *major 7th interval.* Extra note creates a more sophisticated & contemplative feeling Fig. 4.7: C major 7th chord.

· Pattern: [0, 4, 7, 11]

· Intervals: Major 3rd, Perfect 5th, Major 7th

· Notation: $\mathrm{Cmaj}^7, \mathrm{CM}^7, \mathrm{CMa}^7$

· Python Function:

```
def major7th(root):
    return [ root, root + 4, root + 7, root + 11 ]
```

∗ 4.3.5. Minor 7th. A minor 7th starts with a minor triad & adds a minor 7th (root + 10). This chord is a bit more moody than major 7th in feel Fig. 4.8: D minor 7th chord.

· Pattern: [0, 3, 7, 10]

· Intervals: Minor 3rd, Perfect 5th, Minor 7th

· Notation: $\mathrm{Dmin}^7, \mathrm{Dm}^7$

· Python Function:

```
def minor7th(root):
    return [ root, root + 3, root + 7, root + 10 ]
```

∗ 4.3.6. Dominant 7th. Just like major & minor 7ths, can create a dominant 7th by combining some of our earlier building blocks. Dominant 7th starts with a major triad & adds a minor 7th to get pattern $[0, 4, 7, 10]$. Combination of major & minor intervals can create a feeling of restlessness Fig. 4.9: G dominant 7th chord.

· Pattern: [0, 4, 7, 10]

· Intervals: Minor 3rd, Perfect 5th, Minor 7th

· Notation: G^7

· Python Function:

```
def dominant7th(root):
    return [ root, root + 4, root + 7, root + 10 ]
```

∗ 4.3.7. Suspended 2 & suspended 4. Suspended triad chords start with a major triad, but shift middle note up or down. A sus2 chord combines a root note, a major 2nd (+2), & a perfect 5th Fig. 4.10: Csus2 chord.

· Pattern: [0, 2, 7]

· Intervals: Major 2nd, Perfect 5th

· Notation: Csus2, $\mathrm{C}^{\mathrm{sus2}}$

· Python Function:

```
def sus2(root):
    return [ root, root + 2, root + 7 ]
```

A sus4 chord shifts middle note in other direction to a major 4th (+5) Fig. 4.11: Csus4 chord.

· Pattern: [0, 5, 7]

· Intervals: Major 4th, Perfect 5th

· Notation: Csus4, $\mathrm{C}^{\mathrm{sus4}}$

· Python Function:

```
def sus4(root):
return [ root, root + 5, root + 7 ]
```

Can try both versions of suspended chord in interactive tutorial.

- ∗ 4.3.8. Augmented triad. Last type of chord we'll cover here is called an *augmented triad*. This is just a major triad with a "sharpened 5th": last note is raised from a perfect 5th (+7) to an augmented 5th (+8). This chord can add a feeling of suspense or anxiety Fig. 4.12: C augmented chord.
  - · Pattern: [0, 4, 8]
  - · Intervals: Minor 3rd, Perfect 5th
  - · Notation: Caug, C+
  - · Python Function:

```
def augmentedTriad(root):
    return [ root, root + 4, root + 8 ]
```

  There are many, many other kinds of chords we can explore that add extra notes or use notes in different patterns. Extended chords bring in intervals like 9ths, 11ths, & 13ths, & inverted chords shift position of root note. With 88 keys on a piano keyboard & dozens of chord qualities to choose from, there are hundreds & hundreds of possible chords we can use in a given song. How do we reduce this complexity to generate music that sounds good? There are 3 answers to this question:

  - · 1st, *musical keys* are like templates that give us a collection of chords & notes that will sound good together. Once we know what key we're in, set of possible chords becomes much more manageable. Next chap will cover main ideas behind keys & scales.
  - · 2nd, there are standard *chord progressions* that are used consistently in different genres of music. A chord progression is a sequence of chords that set up a compositional structure for a piece of music. In Chap. 6, show how to generate progression of chords from common templates.
  - · 3rd answer is simply hard-won experience. As develop your musical ear, will become more & more familiar with chord types & progressions & how they're used in different genres. This experience will help you begin to innovate & improvise.

  Common interval names: [Table: Semitones: Name]

Note 3. *Function names have to start with a letter (lowercase or uppercase). Names can include letters, numbers, & underscore _ character. Unicode characters are also allowed.*

- Interlude 4: Playing Chords. In this interlude we're going to explore a few options for playing chords using TunePad & Python. When composing harmony of your song, have more to consider than just what chords to choose. Also have to consider how to play these chords. Subtle variation in timbre, harmonics, & timing can make a huge difference in sound that you ultimately produce. In Chap. 4, saw how to play chords using a list & a single `playNote` statement like this:

```
playNote([48, 53, 55], beats = 4.0)
```

This is your most basic & most mechanical sounding option. Here are a few other ideas & techniques to experiment with. Can follow along online with this TunePad project https://tunepad.com/interlude/play-chords.

- ○ Option 1: Block chords. With block chords you play every note in a chord at exact same time & for exact same duration. This approach is simple & can add a strong rhythmic feel to your music. But in some situations using block chords can sound harsh & overly mechanical. Here's a simple function that takes a chord (a list of numbers) & plays each note at same time for an equal duration:

```
def block(chord, beats):
    playNote(chord, beats)
```

- ○ Option 2: Rolled chords. Sometimes when humans play chords, they introduce subtle variations in timing between note onsets. This variation can be intentional & exaggerated or simply a natural result of playing or strumming chords by hand. This style is called a *rolled* chord. Might roll a chord if want to bring out a change in harmony or if you want to emulate a strummed instrument. Artists like Dr. Dre have used rolled chords on piano to create iconic sounds. Can also combine rolled chords & block chords. If your chord progression is changing chords, can draw attention to this by rolling chord that changes. This function rolls a chord by adding a short, fixed delay between each note onset.

```
def rolled(chord, duration):
    delay = 0.1 # how far to space out note start times
    offset = 0 # accumulated delay
    for note in chord:
        playNote(note, beats = delay, sustain = duration - offset)
        offset += delay # keep track of accumulated delay
    fastForward(duration - offset)
```

Function uses a couple of "bookkeeping" variables called `delay, offset`. `delay` variable just says how long to pause before each successive note in chord is played. `offset` variable keeps track of total amount of delay that we've introduced in for loop. Use `offset` variable in 2 places. 1st, on line 6, adjust `sustain` parameters so that all of notes in chord are released at exact same time (sustain parameter lets not ring out longer than what gets passed in beats parameter). 2nd, on line 9, adjust playhead position after loop finishes. This makes it so that calling rolled function advances playhead forward by *exact amount* specified in `duration` parameter. 1 quick note: line 7 uses plus-equal operator $+=$. This is a short hand way of saying `offset = offset + delay`.

- Option 3: Random rolled chords. Can take this technique a step further by *randomly* varying note onset times. To do this, use 1 of Python's utilities from random module: `uniform` function. This function takes an input of 2 numbers & generates a random decimal number between those 2 numbers. Will use this to generate an offset between each successive note in chord. As with previous example, use `sustain` parameter of `playNote` to hold out each note for remaining duration of original inputted beats, subtracting total cumulative amount of offset each time.

```
from random import uniform
def rolled(chord, duration):
    max_delay = 0.15
    offset = 0
    for note in chord:
        next_delay = uniform(0, max_delay)
        playNote(note, beats = next_delay, sustain = duration - offset)
        offset += next_delay
    fastForward(duration - offset)
```

This code is a little more complicated than previous example, talk through it 1 line at a time. On 1st line, importing `uniform` function. On line 2, defining our rolled function with 2 parameters: a list of notes in a chord & number of total beats. On line 3, defining a constant value that defines maximum value that our offset between 2 notes can take. Higher values will create a more spaced-out sound, & lower values will create more closed, tighter-sounding chords. On line 4, initialize a variable to track total offset at each step, which starts at 0. Starting on line 5, iterate through each note of chord. At each step, calculate offset to next note & then call `playNote`. Finally, on line 9, move playhead remaining number of beats.

If this code seems confusing, that's okay. Dive more into understanding this kind of code in later chaps. Can treat this function as another tool in your coding toolkit.

- Option 4: Arpeggios. Another way of playing chords: play 1 note at a time. This method is called an *arpeggio*. Order you play notes doesn't matter. Can start at any note & play notes of chord in any order to get sound that's right for your track. In code below, starting with lowest note & working up in increasing order of pitch.

```
def arpeggio(chord, total_beats):
    duration = total_beats / len(chord)
    for note in chord:
        playNote(note, beats = duration)
```

On 1st line, set up our function def, which takes 2 parameters: a list of notes in a chord & total number of beats to play chord. On 2nd line, calculate duration of each individual note by evenly dividing total number fo beats by number of notes in chord. On lines 3 & 4, use a for loop to iterate through list of notes in chord, playing each note for same number of beats.

- Option 5: Patterned arpeggios. This arpeggio function is ok, but try to make sth a little more interesting. Remember, don't have to play each note evenly & can switch up order of chord. Define a function that takes a 7th chord, which has exactly 4 notes, & play it over duration of a measure. Going to use concept of *indexing*, which read more about in next chap. For now, just know that indexing is how we access specific elements of a list. To index a list in Python, use square brackets around position of element want to use. These positions start at 0, so if have a variable `chord` & want to access 1st element of list, root, would type `chord[0]`. With this in mind, define a function. Have 4 beats to work with & 4 notes – that's indices 0–3. A quick example of what's possible, but you can experiment with different note variations & note orders to get different effects.

```
def my_pattern(chord):
    playNote(chord[0], 0.75)
    playNote(chord[1], 0.25)
    playNote(chord[2], 0.5)
    playNote(chord[1], 0.5)
    playNote(chord[3], 0.5)
    playNote(chord[2], 0.5)
    rest(1)
```

In this function, we aren't doing anything fancy to iterate through chord, & don't need to calculate our beats each time. Our beat values for lines 2–8 add up to exactly 4.0 beats. If following along, can try tweaking these to be different values that still up to 4.0 beats. Can also try tweaking indices of notes to change which chord tone plays.

- 5. Scales, keys, & melody. *Scales* are patterns of notes played 1 at a time in ascending or descending order of pitch. Most scales span 1 octave using some subset of 12 possible notes on piano keyboard. When scale completes octave, pattern starts over. *Keys* are similar to scales except ordering of notes doesn't matter, & they contain all of notes in scale regardless of octave that you start on. Keys are like templates that help us select notes & chords that we know will sound good together. Keys give harmonic & melodic structure to music.
  - 5.1. Chromatic scale. Building blocks of scales are half steps & whole steps. Half steps are smallest interval commonly used in music & distance between 2 notes that are next to each other in pitch & on piano keyboard. Whole steps are made up of 2 half steps.

    Most basic scale is chromatic scale. In this scale, every note is exactly 1 half step up from previous note. This scale can start on any note & spans an octave in 12 notes. Starting with a C on piano keyboard, would have following notes: C C# D D# E F F# G G# A A# B. Or, using MIDI (Musical Instrument Digital Interface) note numbers we could also write: 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59.

    Playing a chromatic scale in TunePad is easy using a loop:

    ```
    # loop from 48 up to (but not including) 60
    for note in range(48, 60):
        playNote(note)
    ```

    If wanted to play a chromatic scale starting on a different root note, could just change numbers in `range` function above.
  - 5.2. Major & minor scales. Perhaps most important scales in Western music are major & minor scales. These scales each use 7 out of 12 possible notes in an octave. There are 12 major scales & 12 minor scales, 1 for each possible starting pitch. After 7th note, next note would be 1st note – or *tonic* – an octave up. Scales are named by their tonic & quality in same way that chords are named. A major scale starting on note D would be called *D Major*.

    Major scales are commonly described as cheerful & happy (like major chords). Major scale is made up of following intervals: *whole step, whole step, half step, whole step, whole step, whole step, half step*. Major scale starting on C would have notes shown in Fig. 5.1: Whole step & half step intervals of C major scale.

    In MIDI version, can see whole steps skip up by 2 notes, which half steps skip up by 1 step. [Table: Note names: MIDI numbers: Intervals].

    Minor scales also use 7 notes out of each octave, but in a different order than major scales. This difference in intervals contributes to different emotional connotation of scale. Minor scales are commonly described as sad, melancholy, & distant. A minor scale starting on C would have following notes: [Table: Note names: MIDI numbers: Intervals].

    Major & minor scales are both examples of *modes*. Modes are simply different ways of ordering intervals of a scale.
  - 5.3. Pentatonic scales. Pentatonic scales are a subset of notes of major & minor scales. There are 5 notes in a pentatonic scale. These scales have no half step intervals, which results in less dissonance between notes. Many common melodies are based on pentatonic scales, especially in folk & pop music. Melody of *Amazing Grace* is pentatonic, as is Ed Sheeran's *Shape of You*.

    There are both major & minor pentatonic scales. Major pentatonic is created by omitting 4th & 7th notes of major scale. Minor pentatonic omits 2nd & 6th notse of minor scale. Can experiment with sound of pentatonic scale by playing only black keys of piano keyboard, which forms either an F# major pentatonic scale or a D# minor pentatonic scale Fig. 5.2: C Major Pentatonic Scale & F# Major Pentatonic Scale. F# Major pentatonic scale uses only black keys of keyboard. D# Minor pentatonic scale starts with D# & uses only black keys as well.
  - 5.4. Building scales in TunePad. Building scales in TunePad is similar to building chords. Because scales are just patterns of intervals (spaces between notes), can create short functions to generate scales. Every major scale has an identical pattern of intervals, & same is true for minor scales as well. Only thing that changes is starting note. To generate scales in TunePad, all we need to do is decide what note to start on & then apply pattern to this starting note.

    1 of advantages of thinking about music in terms of computer code: we don't have to memorize endless scales & combinations of notes & chords that make up different keys. Professional musicians train for years to learn how to play different scales without having to think about it so that they can fluidly switch from 1 key to another. This is part of what makes improvisational musicians so impressive. Playing a solo means knowing exactly which notes & chords can be played & how those notes & chords relate to a genre or theme of a piece being performed.

    A quick example of generating a scale with Python code in TunePad:

    ```
    def majorScale(tonic):
        intervals = [ 0, 2, 4, 5, 7, 9, 11 ]
        return [ i + tonic for i in intervals ]
    ```

    Example above uses a new Python concept called a *list comprehension*. A list comprehension is a shorthand way to create a list in Python. Line 2 uses a list comprehension to create a new list consisting each element of intervals list added to tonic value: `[ i + tonic for i in intervals]` This is equivalent to writing:

```
result = [ ]
for i in intervals:
    result.append(i + tonic)
```

2nd version is a little more cumbersome to write than 1st version using list comprehension, although either version is fine to use.

∗ 5.4.1. Major scale.

- Intervals: [0, 2, 4, 5, 7, 9, 11]
- Notation: C major, CMaj, CM, C
- Python Function with List Comprehension:

```
def majorScale(tonic):
    intervals = [ 0, 2, 4, 5, 7, 9, 11 ]
    return [ i + tonic for i in intervals ]
```

An alternative Python function with a loop instead of a list comprehension:

```
def majorScale(tonic):
    intervals = [ 0, 2, 4, 5, 7, 9, 11 ]
    scale = [ ]
    for i in intervals:
        scale.append(i + tonic)
    return scale
```

A 3rd variation with no loop & no list comprehension:

```
def majorScale(tonic):
    return [ tonic, tonic + 2, tonic + 4, tonic + 5, tonic + 7, tonic + 9, tonic + 11 ]
```

∗ 5.4.2. Minor scale.

- Intervals: [0, 2, 3, 5, 7, 8, 10]
- Notation: C minor, Cmin, Cm
- Python Function

```
def minorScale(tonic):
    intervals = [ 0, 2, 3, 5, 7, 8, 10 ]
    return [ i + tonic for i in intervals ]
```

∗ 5.4.3. Major pentatonic scale.

- Intervals: [0, 2, 4, 7, 9]
- Python Function:

```
def majorPentScale(tonic):
    intervals = [ 0, 2, 4, 7, 9 ]
    return [ i + tonic for i in intervals ]
```

∗ 5.4.4. Minor pentatonic scale.

- Intervals: [0, 3, 5, 7, 10]
- Python Function:

```
def minorPentScale(tonic):
    intervals = [ 0, 3, 5, 7, 10 ]
    return [ i + tonic for i in intervals ]
```

∗ 5.4.5. Chromatic scale.

- Intervals:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
- Python Function:

```
def minorPentScale(tonic):
    intervals = [ 0, 3, 5, 7, 10 ]
    return [ i + tonic for i in intervals ]
```

Try these functions at https://tunepad.com/examples/build-scales.

○ 5.5. Playing scales in TunePad. Now have seen how to build a scale, can use functions from prev sect to play music. Unlike with chords, notes of a scale aren't usually played all at once. Most basic way to play a scale: play 1 note at a time, in ascending order. Somehow we have to access each element of list individually & pass that to `playNote` Fig. 5.3: A representation of a list with values & indices.

Each element can be accessed using its position in list – called an *index*. In coding, 1st element of a list is at index 0; 2nd element of list is at index 1, 3rd at index 2; & so on. In Python can also access last element of a list at index −1. Accessing individual elements of a list is referred to as *indexing*. In code, do this by using square brackets & index number. Can also use this technique to change value of individual elements in a list.

```
notes = [ 60, 62, 64 ]
notes[2] = 66 # replace the value 64 with 66
playNote( notes[0] )
playNote( notes[1] )
playNote( notes[2] )
```

In line 1, define a new list called *notes* with 3 values. In line 2, replace value at index 2 with new value of 66. In lines 3–5, play each note of list, 1 at a time. *1 of most confusing parts about computer programming for beginners: lists start at index 0 & end at an index 1 less than length of list.* However, with a little practice this become less & less confusing.

Note 4. *If try to index into a list with an index that doesn't exist, Python will stop running & complain with an* `IndexError`*. Because indices start at 0, valid indices are 0 all way up to length of list minus 1.*

Another way to iterate through a list is by using a *for loop*. Previously, when have seen loops, have used them to do exact same operation multiple times in a row. Recall syntax of a for loop:

```
for var in range(start, stop):
```

Can replace `range(start, stop)` part of a for loop with a list instead. This will execute body of loop once for every element in list. There's a special variable here called "loop variable" that gets set to value of each consecutive element in list every time the loop repeats. In code above, `var` is loop variable, but can use any valid Python variable name. E.g., a loop that plays all notes of a major scale starting on note 60.

```
for note in majorScale(60):
    playNote(note)
```

In this example, `note` is our loop variable. For every iteration of loop it gets set to next note in scale. Give this a try at https://tunepad.com/examples/play-scales.

- 5.6. Other kinds of scales. There are many other types of scales, but variants of major & minor scale are most common in popular music. Other scales that we don't cover here include set of church modes, whole tone scales, diminished scales, & modes of limited transposition.

  – Các loại thang âm khác. Có nhiều loại thang âm khác, nhưng các biến thể của thang âm trưởng & thứ là phổ biến nhất trong âm nhạc đại chúng. Các thang âm khác mà chúng tôi không đề cập ở đây bao gồm bộ các chế độ nhà thờ, thang âm toàn cung, thang âm giảm, & chế độ chuyển cung hạn chế.

  Above have discussed scales common to Western music, but concept of collections of notes is cross-cultural. Arabic maqam is system of melodic modes in traditional Arabic music used in both compositions & improvisations. In Indian classical music Rage are collections of melodic modes & motifs, each connoting a distinct personality or emotion. Gamelan music in Indonesia is organized by Pathet, which is a system of hierarchies of notes in which different notes have prominence. Composers from West have often borrowed – or in some cases, stolen – these scales for their own music. This raises many issues of appropriation & exploitation within music industry. Music industry has a long history of marginalizing groups while also profiting off of cultural traditions without properly compensating or recognizing musical provenance.

  – Ở trên đã thảo luận về các thang âm phổ biến trong âm nhạc phương Tây, nhưng khái niệm về tập hợp các nốt nhạc là liên văn hóa. Maqam tiếng Ả Rập là hệ thống các chế độ giai điệu trong âm nhạc Ả Rập truyền thống được sử dụng trong cả sáng tác & ngẫu hứng. Trong âm nhạc cổ điển Ấn Độ, Rage là tập hợp các chế độ giai điệu & họa tiết, mỗi chế độ biểu thị 1 tính cách hoặc cảm xúc riêng biệt. Âm nhạc Gamelan ở Indonesia được tổ chức theo Pathet, đây là 1 hệ thống phân cấp các nốt nhạc trong đó các nốt nhạc khác nhau có sự nổi bật. Các nhà soạn nhạc phương Tây thường mượn – hoặc trong 1 số trường hợp, đánh cắp – các thang âm này cho âm nhạc của riêng họ. Điều này làm nảy sinh nhiều vấn đề về chiếm đoạt & khai thác trong ngành công nghiệp âm nhạc. Ngành công nghiệp âm nhạc có lịch sử lâu dài về việc gạt ra ngoài lề các nhóm trong khi cũng kiếm lợi từ các truyền thống văn hóa mà không đền bù hoặc công nhận nguồn gốc âm nhạc 1 cách thỏa đáng.

- 5.7. Keys. When writing music, there are seemingly endless notes to choose from. Keys are 1 way to narrow down question of what note to choose. Keys are underlying organizational framework of most music & encode both melodic & harmonic structures & rules. Knowing these rules (& how to break them) helps us to write music that listeners can easily comprehend & appreciate.

  Concept of keys is closely related to that of scales. Keys are composed of all of notes in all of octaves that make up scale with same name. E.g., notes in key C major are same as notes in C major scale across all octaves. But, while scales are usually played in increasing or decreasing order of pitch, ordering of notes in a key doesn't matter. Notes that are part of a given key are called *diatonic*, & remaining notes that are not part of that key are called *chromatic*.

Hundreds of years ago, different keys used to be associated with different emotions, so composers would choose specific keys that reinforce mood of their composition. This is because intervals in each key were slightly different due to system of tuning; different keys were actually aurally distinct from 1 another. In modern times, each key is made up of exact same intervals.

- 5.8. Circle of 5ths. Keys are organized according to *Circle of 5ths*. Circle of 5ths is essentially a pattern of intervals. Moving clockwise around circle is moving tonic note up by a 5th from previous key. This adds 1 raised – or sharp – note as 7th note of scale. Alternatively, moving counterclockwise raises tonic by a 4th & is often referred to as Circle of 4ths. This adds 1 flat note as 4th note of scale Fig. 5.4: Circle of 5ths arranges musical keys.

  Major & minor keys that share all of same notes are considered relative keys. For a minor scale, relative major scale starts on 3rd note; for a major scale, relative minor starts on 6th note. Relative minor key of C major is A minor, & relative major of A minor is C major.

  Keys that are adjacent on Circle of 4ths or 5ths – e.g., D major & G major – share nearly all of same notes & are considered *closely related*. Relative major or minor key for a given key is also considered closely related. Generally, when a song changes keys – also known as *modulating* – it goes to 1 of closely related keys. Because closely related keys share most of same notes, modulating to 1 of these keys is less jarring to listener.

- 5.9. Melody. *Melody* is central component of much of music we listen to. It's part of a song that gets stuck in your head. Much of what goes into great melody writing is intuition & practice, but knowing a bit of theory can help you get started. Melodies have 2 essential parameters: pitch & rhythm. These elements are of equal importance, but in this sect, mostly be looking at pitch.

  When it comes to writing melodies, understanding that you are working within confines of keys, scales, & a harmonic chord progression is a great place to start. Can use our song's harmony to provide a structural scaffold. Often, melodies place chord tones from harmony on strong beats of measure (beats 1 & 3). These tones are consonant with harmony, i.e. they sound pleasing. Simplest melody might stick solely to these chord tones. In example below, only play chord tones of C major & D minor.

```
# over C major
playNote(55, 0.75)
playNote(55, 0.25)
playNote(52, 1)
playNote(48, 0.5)
rest(1.5)

# over D minor
playNote(57, 0.75)
playNote(57, 0.25)
playNote(53, 1)
playNote(50, 0.5)
```

  Follow along with these examples at https://tunepad.com/examples/simple-melody.

  Dissonance is also a powerful tool in melody writing. This can add interest & variation & sometimes have an intense emotional impact on listeners. A melody with no dissonance, that only plays chord tones, becomes boring. 1 way to utilize dissonance: add notes in between our chord tones to fill in our melody. Can choose notes that correspond with scale based on our song's key. In example below, now filling in space between last 2 notes of each measure:

```
# over C major
playNote(55, 0.75)
playNote(55, 0.25)
playNote(52, 0.5)
playNote(50, 0.5) # passing tone
playNote(48, 0.5)
rest(1.5)

# over D minor
playNote(57, 0.75)
playNote(57, 0.25)
playNote(53, 0.5)
playNote(52, 0.5) # passing tone
playNote(50, 0.5)
```

  Sth to consider when writing a melody is *contour*. Contour describes shape melody takes: natural rises & falls in pitch. A melody can either move stepwise to adjacent notes or leap to more distant notes. This motion can either decrease or increase in pitch. I.e., have 4 types of motion a melody might take, each with different connotations. E.g., might hear a melody that opens with a large leap as more emotional. Can expect most melodies to be within range of about an octave to an octave & a half. In example below, combine idea of leaps to chord tones & passing tones:

```
# over C major
playNote(55, 1)
playNote(64, 1.5) # large leap
playNote(60, 1.5)

# over D minor
playNote(57, 1)
playNote(65, 1.5) # large leap
playNote(67, 1) # passing tone
playNote(69, 0.5)
```

If consider contour & pitch content as a vertical phenomenon, can think of melodic form as a horizontal structure. Can break melodies into parts called *phrases*. If melodies are paragraphs, then phrases are like musical sentences. They are complete thoughts that are punctuated & combined to form more complete & cohesive ideas. Phrases are often 2, 4, or 8 bars in duration. These phrases are combined to form larger structures, which become overall song form. Explore this more in Chap. 9.

Principles of repetition & variation work in opposition to 1 another. In writing melodies, there generally needs to be enough repetition so that a listener has sth to latch onto. But with too much repetition, a melody becomes boring. A catchy melody is result of striking a balance between these 2 forces.

1 way to build intuition about melody writing: analyze melodies from artists you like & want to emulate. Critical listening skills that you develop from analyzing existing melodies is directly applicable to writing your own melodies. Experimentation & improvisation are other great ways to build up this intuition. Can try tapping out rhythms to serve basis of a melody, or play around on a piano or another instrument. Try playing around with our automatic melody generator at https://tunepad.com/examples/melody-gen.

- Interlude 5: Lean On Me. BILL WITHERS. Practice using chords by recreating a small part of piano harmony from song *Lean on Me* by BILL WITHERS (1972), Columbia Records. A simplified version of chord structure that you can try entering into a TunePad project.

```
# Chord Variables
Cmaj = [ 48, 52, 55 ]
Dmin = [ 50, 53, 57 ]
Emin = [ 52, 55, 59 ]
Fmaj = [ 53, 57, 60 ]
Bdim = [ 47, 50, 53 ]
```

[Table: Chord: Beats: Python]. Can see full code here: https://tunepad.com/interlude/chord-progressions. 1 thing to notice is how chord progression mirrors emotion of song as a whole. WITHERS mixes upbeat ("I'll be your friend. I'll help you carry on") with harsh reality of life ("We all have pain. We all have sorrow"). Harmony starts on a major chord (C major) but then passes through a succession of minor chords (D minor, E minor) before eventually landing on more encouraging major chords for prolonged notes (F major). It's as if harmony is also saying: we're going to go through some hard times, but it'll work out in end.

Version above is slightly modified from original in that we're using simplified chords & a diminished B chord at end that slides into a C major. As listen to it, notice how B diminished feels unstable as if it needs to resolve into C major to bring harmony full circle to signal a transition in song.

More elegant code. 1 temping way to code this up would be to just type all of `playNote` functions, 1 after another. This works, but it's not necessarily most elegant way to express music. When you're coding, there's always > 1 way to solve a problem, so it's good to get into habit of asking if there are other, easier ways to accomplish things. E.g., what if wanted to change velocity of all of chords? We'd have to edit 1 line at a time or use find & replace. As an alternative, what if we put all of chords into a list & then iterated through that list with a for loop?

```
chords = [ Cmaj, Cmaj, Dmin, Emin, Fmaj, Fmaj, Emin, Dmin, Cmaj ]
for chord in chords:
    playNote(chord)
```

This would be an improvement. If nothing else, would have reduced number of lines needed to play harmony. Obvious problem: it won't work because notes are different lengths. Some are long (4 beats) & others are short (1 beat). But this code plays all chords with equal duration.

If there were an easy way to iterate through 2 lists at same time, could make 1 list with chords & another with durations. Python includes exactly this kind of feature with sth called `zip` function. Think of it like a zipper that merges 2 Python lists together instead of 2 pieces of fabric. It walks through lists, element by element, & merges them together into pairs of values. Result looks sth like this:

```
chords = [ Cmaj, Cmaj, Dmin, Emin, Fmaj, Fmaj, Emin, Dmin, Cmaj ]
durations = [ 4, 1, 1, 1, 4, 1, 1, 1, 4 ]
for chord, duration in zip(chords, durations):
    playNote(chord, beats = duration)
```

Complete example or you can try it online: https://tunepad.com/interlude/lean-on-me.

```
CM = [ 48, 52, 55, 55 + 12 ]
Dm = [ 50, 53, 57, 57 + 12 ]
Em = [ 52, 55, 59, 59 + 12 ]
FM = [ 53, 57, 60, 60 + 12 ]
Bd = [ 47, 50, 53, 53 + 12 ]

chords = [ CM, CM, Dm, Em, FM, FM, Em, Dm, CM, CM, Dm, Em ]
durations = [ 4, 1, 1, 1, 4, 1, 1, 1, 4, 1, 1, 1, 3, 4 ]

for chord, duration in zip(chords + [ Em, Dm ], durations):
    playNote(chord, beats = duration)

for chord, duration in zip(chords + [ Bd, CM ], durations):
    playNote(chord, beats = duration)
```

- 6. Diatonic chords & chord progressions. Now that have some familiarity with chords, question is how to use them. How can we reduce hundreds of chords & thousands of combinations of chords down to a manageable set of options? How can we explore creative musical space that chords provide without feeling overwhelmed?

  – Hợp âm diatonic & tiến trình hợp âm. Bây giờ đã quen với hợp âm, câu hỏi đặt ra là làm thế nào để sử dụng chúng. Làm thế nào chúng ta có thể giảm hàng trăm hợp âm & hàng nghìn tổ hợp hợp âm xuống một tập hợp các tùy chọn có thể quản lý được? Làm thế nào chúng ta có thể khám phá không gian âm nhạc sáng tạo mà hợp âm cung cấp mà không cảm thấy choáng ngợp?

  1 answer to these questions: use keys to select subsets of chords that we know will sound good together. From there we can follow guidelines for arranging chords into sequential patterns called *progressions* that will support various harmonic elements that come together in a piece of music.

  This chap introduces traditional *Roman numeral* system for referring to chords that fit with a particular key along with methods for choosing chord progressions. Along way we'll code functions for creating chord progressions in any key using concept of Python *dictionaries*.

  - 6.1. Diatonic chords. A *diatonic* chord is any chord that can be played using only 7 notes of current key. E.g., if working in key of C major, diatonic chords consist of all of chords you can play with only white keys on piano keyboard: C D E F G A B. Main diatonic chords we can make with just these 7 notes Fig. 6.1: 7 diatonic chords of C major. Hear these chords at https://tunepad.com/examples/diatonic-chords.

    If have a piano keyboard handy, try playing with these 7 chords to get a feel for how they sound. What emotions do you feel as chords ring out? What patterns of chords sound good together? No matter what key we're in, there will always be 7 diatonic chords, 1 for each note in scale. To build a diatonic chord, just pick any note from scale as root of chord. Then go up 2 notes for "3rd" of chord, & up 2 notes again for "5th" of chord. Can keep moving up by 2 notes of scale to get "7th" & "9th" & so on.

    Can refer to these chords by their scale degree (1st, 2nd, 3rd, 4th, 5th, 6th, & 7th). A chord built on 5th note of a scale would be "5" chord for that key. If we're in any *major key*, 1st, 4th, & 5th diatonic chords will have a major quality (regardless of starting note of key). Further, 2nd, 3rd, & 6th chords will always have a minor quality, & 7th chord is diminished. E.g., in C major, would have following diatonic chords: CMaj Dmin Emin FMaj GMaj Amin Bdim. This pattern of chord qualities is same for any major key because all of major keys have same pattern of note intervals. Same idea is true for minor keys. Because all minor keys have same pattern of intervals, quality of chords stays consistent. 1st, 4th, & 5th diatonic chords are minor quality. 3rd, 6th, & 7th chords are major quality. 2nd chord is diminished. In C minor, e.g., would have following diatonic chords: Cmin Ddim E♭Maj Fmin Gmin A♭Maj B♭Maj. In popular music, chord progressions are made up almost entirely of diatonic chords. Recall from prev chap: melody of a song is also built on this harmonic scaffold. Melodies often use primarily notes from underlying chord progression, because these notes are more consonant & pleasing. Notes from outside harmony are typically used in passing as decoration or as a neighboring tone.

- 6.2. Roman numerals. Each of 12 major keys & 12 minor keys have 7 diatonic chords, giving us an overwhelming total of 168 diatonic chords. To reduce this complexity, musicians, producers, & composers use a system of *Roman numerals* to refer to different diatonic chords by their scale degree rather than their specific name. If chord has a major quality, it gets an uppercase Roman numeral. If it has a minor quality, it gets a lowercase Roman numeral. Each key also has 1 diminished chord, which is both lowercase & has an accompanied ° symbol.

  Note 5. *Roman numerals are a system of numbering which originated in ancient Rome. In this system, numbers are composed of combinations of letters. In music, only numerals corresponding to numbers one through 7 are used: I, II, III, IV, V, VI, & VII.*

  That gives us following Roman numerals ∀ of diatonic chords of any key. [Table: Scale degree (Major keys, Minor keys): 1st: 2nd: 3rd: 4th: 5th: 6th: 7th]. With this system there are just these 7 symbols to focus on instead of 168. Roman numerals take some getting used to, but they give us a language for thinking about chord progressions without having tp refer to name of each specific chord. In coding or mathematics, this kind of generalization is called an *abstraction.* Abstraction means removing specific details of individual situations & focusing instead on bigger picture patterns. In coding, use language constructs like variables, functions, & parameters to create abstractions that reduce complexity of our code & problems we're trying to solve.

- 6.3. Tendency tones & harmonic functions. Now know how to refer to diatonic chords by their names, how do order them into pleasing chord progressions? Ordering of chords within a progression isn't random. Notes that make up chords have different tendencies relative to 1 another – i.e. listeners hears them as wanting to resolve in expected ways when played with other notes in a chord progression. Strongest of these tendencies is pull of 7th note of a scale back to root note of scale (tonic). In most cases, hear this note as wanting to resolve upward by a half step back to tonic. If this note does not resolve, often hear progression as incomplete. 2nd & 5th notes of a scale also have a strong pull back to tonic.

  Tendencies of individual notes give each chord a characteristic or *function.* Can think of a chord's function as its desire – how it relates to prev chords & how it *wants* to move music forward. Chords are described as having 3 primary functions: *tonic, predominant, & dominant.* Most chord progressions typically progress from tonic to predominant to dominant back to tonic. [Table: Tonic chords: Predominant chords: Dominant chords].

  Chords that have same function also share many of same notes. E.g., iii (3) chords & vi (6) chords share 2 of same notes with tonic chords (I), so they're grouped together. Predominant ii & IV chords also share 2 notes in common, as do dominant V & vii° chords Fig. 6.2: Dominant V & vii° chords share 2 notes in common. This is easy to see when you line piano diagrams up vertically.

  Chord functions might best be described as rough guidelines that help inform our decisions about composition. There are also many variations to basic chords. Chords can be inverted (i.e., root is no longer lowest note), extended with additional notes to add color, or combined with "chromatic" chords that include notes from outside of main key. Secondary dominant chords are dominant chords borrowed from other related keys. Knowing which chord to use in any given context comes down to musical experience, taste, & conventions of different genres.

- 6.4. Chord progressions. Many common chord progressions follow scheme of *tonic → predominant → dominant.* Tonic brings stability & grounding. Predominant is a departure from this stability that builds tension. Predominant pulls toward dominant, which eventually resolves back to tonic. After a chord progression finishes, it starts over. Different genres of music have different harmonic rules & standard chord progressions, but here are flow charts that help visualize common chord progression patterns Fig. 6.3: Flowcharts for generating chord progressions in major & minor keys.

  An example of how to use these charts. If start with tonic I chord on top, might move down to dominant V chord, & then slide up to tonic prolongation vi chord – a subtle tease with resolution. Could then go to predominant IV chord before progression resolves back to tonic I chord & repeats. This progression would be I → V → vi → IV (1, 5, 6, 4), which is an extremely common pattern in popular music Fig. 6.4: Example of using flowchart to generate a chord progression.

  Contemporary pop music uses many of same progressions as early rock & Roll & Blues music – much of early rock music came out of Blues. As a result, many early rock songs are built on Blues progressions, most notably I-IV-V. 1 of most ubiquitous progressions – especially in early rock – is "doo wop" progression I-vi-IV-V. Same chords can be reordered to form our I-V-vi-IV example. [Table: Common major progressions: Common minor progressions]

  Hip-hop songs center more around rhythm & vocals & tend to use shorter progressions, often in a minor key with only 1 or 2 chords e.g. i-V, i-VI, & i-ii°. Hip-hop developed alongside advances in recording technology that allowed early artists to remix samples from other songs, &, as a result, genre also borrows progressions from pop & rock music.

  When writing chord progressions, 1 tactic: borrow from existing songs to help you develop your own ear & begin to think critically about harmony. Can also experiment on your own. Use harmonic conventions to narrow down some of options, but also try breaking rules as you become more confident.

- 6.5. Chord inversions. An *inverted* chord is just like an ordinary chord except that root note is no longer lowest pitch. Take C major as an example. When root note C is also lowest note of chord, say chord is in root position Fig. 6.5: C major chord in root position, 1st inversion & 2nd inversion.

  When 3rd of chord is lowest note, chord is in its 1st inversion. In case of C major, i.e., E is now lowest note. When 5th of chord is lowest, it's 2nd inversion, & so on. Each inversion has exactly same notes as root chord, but ordering of notes by their pitch is different.

- 6.6. Voice leading. *Voice leading* deals with relationship between notes in consecutive chords in a progression. Principle behind voice leading: treat each note of a chord as an individual melodic voice. Imagine 3 human vocalists each singing 1 individual note of a chord. Because considering each voice independently, idea: minimize leaps 1 person's voice has to make between chords so that progression is smoother & easier to sing. By considering different possible inversions of each chord we can create more of a dovetailing effect with subtle shifts between successive chords. Not only will this improve sound of your progressions, but it will also improve potential playability of music on instruments like guitar, piano, or vocal harmony. 2 figures show same progression with & without voice leading Fig. 6.6: Chord progression I-V-vi-IV without voice leading & with voice leading. Chords V, vi, & IV are inverted to reduce pitch range & to minimize movement of individual voices "singing" notes of chords. Hear these examples at https://tunepad.com/examples/voice-leading.

- 6.7. Python dictionaries. In Python, *dictionaries* or *maps* are unordered sets of data consisting of values referenced by *keys*. These keys aren't same as musical keys. They're more like kind of keys that open locked doors. Each different key open its own door.

  Dictionaries are extremely useful in programming because they provide an easy way to store multiple data elements by name. E.g., if wanted to store information for a music streaming service, might need to save song name, artist, release date, genre, record label, song length, & album artwork. A dictionary gives you an easy method for storing all of these elements in a single data object. (like *.bib files)

```
track_info = {
    "artist" : "Herbie Hancock",
    "album" : "Head Hunters",
    "label" : "Columbia Records",
    "genre" : "Jazz-Funk",
    "year" : 1973,
    "track" : "Chameleon",
    "length" : 15.75 }
```

  Dictionaries are defined using curly braces with keys & values separated by a colon. Different entries are separated by commas. After defining a dictionary, can change existing values or add new values using associated keys. Similar to way we access values in a list with an index, use square brackets & a key to access elements in a dictionary.

```
track_info["artwork"] = "https://images.ssl-images-amz.com/
images/81KRhL.jpg"
```

  In this line, because key "artwork" hasn't been used in dictionary yet, it creates a new key-value pair. If "artwork" had been added already, it would change existing value. 1 thing to notice: values in a dictionary can be any type including strings, numbers, lists, or even other dictionaries. Dictionary keys can also be strings or numerical values, but they must be unique for each value stored.

- 6.8. Programming with diatonic chords. With Python code, there are many different ways to determine diatonic chords for a given key. Here are `majorChord, minorChord, diminishedChord` functions from Chap. 4 again.

```
def majorChord(root):
    return [root, root + 4, root + 7]

def minorChord(root):
    return [root, root + 3, root + 7]

def dimChord(root):
    return [root, root + 3, root + 6]
```

  Can use these functions to define variables for each diatonic chord in C major:

```
I = majorChord(48)
ii = minorChord(50)
iii = minorChord(52)
IV = majorChord(53)
V = majorChord(55)
vi = minorChord(57)
vii0 = diminishedChord(59)
```

  This code is clear & readable, but not as reusable as it could be. What if want to play in a different key? Or in a different octave? Would have to change *each* line of code. As an alternative, could write a function that takes tonic as an input & returns a dictionary that maps Roman numerals to individual diatonic chords.

```
def buildChords(tonic):
    numerals_lookup = { "I" : majorChord(tonic),
                        "ii" : minorChord(tonic+2),
                        "iii" : minorChord(tonic+4),
                        "IV" : majorChord(tonic+5),
                        "V" : majorChord(tonic+7),
                        "vi" : minorChord(tonic+9),
                        "vii0" : diminishedChord(tonic+11)}
          return numerals_lookup
```

Method above works for major keys, but what if wanted it to work with minor keys as well? Can add another parameter & an `if-else` statement to handle this as well.

```
def buildChords(tonic, mode):
    if mode == "major":
        numerals_lookup = {"I" : majorChord(tonic),
                           "ii" : minorChord(tonic+2),
                           "iii" : minorChord(tonic+4),
                           "IV" : majorChord(tonic+5),
                           "V" : majorChord(tonic+7),
                           "vi" : minorChord(tonic+9),
                           "vii0" : diminishedChord(tonic+11)}
    else:
        numerals_lookup = {"i" : minorChord(tonic),
                           "ii0" : diminishedChord(tonic+2),
                           "III" : majorChord(tonic+3),
                           "iv" : minorChord(tonic+5),
                           "v" : minorChord(tonic+7),
                            "VI" : majorChord(tonic+8),
                           "VII" : majorChord(tonic+10)}
    return numerals_lookup
```

These are far from only solution for creating diatonic chords for different keys. In general, there are almost an endless number of ways to solve complex problems in programming. Figuring out which approach is best for a given circumstance takes practice & experience, but your goal is usually to write code that is as simple & easy to understand as possible. Try this code at `https://tunepad.com/examples/chord-dictionary`.

- Interlude 6: Random Chord Progressions. A short Python example that generates & then plays random chord progressions using charts in Fig. 6.3. Can start with a table that maps each chord to a simplified set of possible transition chords. Table below on left uses Roman numerals, & table on right uses Arabic numbers to show same thing. Note these tables don't include all of possibilities from flow charts above, but most of possible transitions are included. Now can turn this transition table into a computer algorithm using Python.

  ○ **Step 1: Random chord algorithm.** Create a new piano cell in a TunePad project & add this code.

```
from random import choice # import the choice function

progression = [ 1 ] # create a list with just one chord
chord = choice([3, 4, 5, 6]) # choose a random next chord

while chord != 1: # repeat while chord is not equal to 1
    progression.append(chord) # add the next chord to the list
    if chord == 2: # if the current chord is 2
        chord = choice([1, 5]) # then choose a random next chord
    elif chord == 3: # else if the current chord is 3
        chord = 4 # ...
    elif chord == 4:
        chord = choice([1, 2, 5])
    elif chord == 5:
        chord = choice([1, 6])
    else: # the chord is 6
        chord = choice([ 2, 3, 4 ])
print(progression)
```

Break it down line by line. Line 1 imports a function called *choice* from Python's `random` module. `choice` function selects 1 element from a list at random. Can think of it as picking a random card from a deck. Line 3 creates a variable called `progression` that consists of a list with only 1 element in it. This list will hold our finished chord progression, & start it with tonic chord 1. Line 4 picks next chord at random. Use our transition table to select from 3–6 as possible next chords in sequence. Save random choice in a variable called `chord`. Line 6 is a new kind of Python loop called a `while loop`. This loop repeats indefinitely until a certain condition is met. In our case, going to repeat loop until our `chord` variable = 1. Line 7 is part of while loop. It adds our new `chord` to end of `progression` list using `append` function. 1st time through loop, `progression` list will have 2 elements, 1 & whatever random chord was selected on line 4. Each additional time through loop, line 7 will add another chord to list. Line 8 asks *if* our random chord = 2. If so, it selects a random next chord based on values in our transition table (on line 9). Line 10 only gets used if line 8 is False. This line says: otherwise, if value of `chord` is 3, then set next chord to 4. Lines 12, 14, & 16 handle next set of options for value of `chord`, following transition table.

Once loop completes, line 18 print out result. A sample output might be `[1, 5, 6, 3, 4, 2]`, but since this uses random selection, output is likely to be different each time code runs.

- Step 2: Play chords. So how do our random chord progressions sound? Can add a few more lines of code at end to play our progression in TunePad. Start by defining our diatonic chords in a dictionary. Instead of using Roman numerals as keys, we're going to use chord numbers.

```
tonic = 48
chords = {
    1 : majorChord(tonic),
    2 : minorChord(tonic + 2),
    3 : minorChord(tonic + 4),
    4 : majorChord(tonic + 5),
    5 : majorChord(tonic + 7),
    6 : minorChord(tonic + 9),
    7 : diminishedChord(tonic + 11) }
```

Next can iterate over our `progression` list, playing each chord in turn.

```
for chord in progression:
    playNote(chords[chord], beats = 2)
```

Can try this code on TunePad https://tunepad.com/interlude/random-chords.

- 7. Frequency, fourier, & filters. Chap. 3 introduced idea that different instruments & voices naturally fall into different ranges of frequency spectrum, from low sounds like a bass to high sounds like hi-hats. In this chap, further explore frequency spectrum, this time with an emphasis on techniques for mixing multiple layers of a musical composition into a cohesive whole. Show how sound can be decomposed into its component frequencies & how can use filters & other tools to shape sonic parameters e.g. frequency, loudness, & stereo balance. Also show how to apply these standard filters & effects in TunePad using Python code.

  - 7.1. Timbre. (Âm sắc) All sound is made up of waves of air pressure that travel outward from a sound's source until they eventually reach our inner ears. Sound waves that vibrate regularly, or periodically, are special kinds of audio signals which human brain interprets as musical pitch. Rate of vibration (or frequency) determines how high or low pitch sounds. 1 surprising things about musical notes: they are almost never composed of just 1 frequency of sound. In fact, what people hear as 1 musical note is actually a whole range of frequencies stacked on top of 1 another. E.g., Fig. 7.1: Sound energy generated by a flute playing a single note. Sound contains a series of spikes at regular "harmonic" frequency intervals. shows sound energy generated by a flute playing a single note. Figure shows energy level at different frequencies across whole range of human hearing (from about 20 Hz–20000 Hz). Frequency level is shown on horizontal axis & energy level is shown on vertical axis. Spikes in graph show sounds generated by flute at different frequency levels. So even though we only hear a single note, there are actually a whole range of frequencies present in sound, 1 for each spike.

    Frequency combinations like this allow people to distinguish different kinds of instruments from 1 another. It's how your brain can tell difference between a trombone & cello, even when they're playing exact same note. Many frequencies of a single sound are called its *frequency spectrum*, e.g., Fig. 7.1, & they create what is called *timbre* (pronounced "TAM-ber") – often called *tone color* or *tone quality*. Timbre is like fingerprint of a sound.

    – Các tổ hợp tần số như thế này cho phép mọi người phân biệt các loại nhạc cụ khác nhau. Đó là cách não của bạn có thể phân biệt được sự khác biệt giữa kèn trombone & cello, ngay cả khi chúng chơi cùng một nốt nhạc. Nhiều tần số của một âm thanh duy nhất được gọi là *phổ tần số* của nó, ví dụ, Hình 7.1, & chúng tạo ra cái gọi là *âm sắc* (phát âm là "TAM-ber") – thường được gọi là *màu sắc âm thanh* hoặc *chất lượng âm thanh*. Âm sắc giống như dấu vân tay của âm thanh.

    - **Timbre.** Unique fingerprint of a sound that results from how we perceive multiple frequencies combining together.
    - **Fundamental frequency.** Lowest (& usually) loudest frequency that we perceive as pitch of a note.
    - **Partials or overtones.** Other frequencies beyond fundamental frequency that are also present when we hear a note.
    - **Harmonic frequency.** Any frequency that is close to an integer multiple of fundamental frequency.
    - **Inharmonic frequency.** Any partial that is not an integer multiple of fundamental frequency.

What we perceive as pitch of a note is usually lowest of frequencies present. This is known as *fundamental frequency*, or often simply *fundamental*. Also usually loudest of frequencies. Remaining frequencies are referred to as *partials* or *overtones*. If frequency of partial is close to an integer multiple of fundamental, when partial is considered to be a *harmonic* of fundamental. Otherwise, partial is considered *inharmonic*. Most pitched or melodic instruments – e.g. saxophones, flutes, & guitars – have very harmonic spectrums Fig. 7.1. Non-pitched or percussive instruments often have very inharmonic spectrums, i.e., you don't really perceive pitch of these instruments. You can hear how this sounds here: https://tunepad.com/examples/spectrums.

To help make this more clear, consider a sound consisting of following frequencies: 200 Hz, 400 Hz, & 500 Hz Fig. 7.2: Frequency combinations: fundamentals, partials, harmonic,& inharmonic. Fundamental of sound would be 200 Hz, because it's lowest (& loudest) frequency. 400 Hz frequency would be 1st partial & would be considered a harmonic because it's an integer multiple of fundamental 400 Hz/200 Hz = 2. 500 Hz frequency would be 2nd partial, but it's not a harmonic because 500 Hz/200 Hz = 2.5. Add 100 Hz as new fundamental frequency. In this case, 200 Hz, 400 Hz, & 500 Hz would all be considered harmonics of 100 Hz because they are all simple integer ratios of 100 Hz (2, 4, & 5). Listen to an example https://tunepad.com/examples/timbre.

Almost all sounds consist of a complex combination of frequencies. 1 exception to this rule is sine wave (Fig. 7.2 shows combinations of sine waves). Since waves are made up of just 1 frequency with no other partials, & they are often described as sounding clear or pure because of this. Sine waves are easy to generate using electronics or a computer, but they rarely occur in nature, i.e., they can also sound artificial & harsh.

It turns out: any periodic sound can be described as a combination of a (possibly infinite) number of sine waves forming partial frequencies. Sounds that have very few partials, like a whistle, are often very close to sine waves. Sounds that have many partials, like a saxophone, have much richer & more complex waveforms. 1 way to imagine this: sine waves are like primary colors of paint that you can mix together to form every other color. Fig. 7.3: A square wave (or any other audio signal) can be described as a series of sine waves making up partial frequencies. shows how multiple sine waves at different harmonic frequencies can combine to approximate a more complex signal like a square wave.

○ 7.2. Envelopes. There are other complex properties of sound waves that contribute to an instrument's timbre. 1 of most important of these is how volume of a sound evolves & changes over duration of a note. This is called sound's *envelope*. A simplified envelope is commonly described using 4 stages: Attack, Decay, Sustain, & Release or ADSR for short Fig. 7.4: ADSR envelope.

ADSR envelope has both time components & amplitude (loudness) components. When play a note on piano or another instrument, *attack* is time from when key is 1st pressed to when note reaches its maximum volume. *Decay* is time it takes note to reach a lower secondary volume. *Sustain* is loudness of this 2nd volume. Finally, *release* is how long it takes for note to completely fade out. So, attack, decay, & release are all measures of time, while sustain is a measure of loudness.

– Phong bì ADSR có cả thành phần thời gian & thành phần biên độ (độ lớn). Khi chơi một nốt nhạc trên đàn piano hoặc một nhạc cụ khác, *attack* là thời gian từ khi phím được nhấn lần đầu tiên đến khi nốt nhạc đạt đến âm lượng tối đa. *Decay* là thời gian nốt nhạc đạt đến âm lượng thứ cấp thấp hơn. *Sustain* là độ lớn của âm lượng thứ 2 này. Cuối cùng, *release* là thời gian nốt nhạc mất bao lâu để mờ dần hoàn toàn. Vì vậy, attack, decay, & release đều là các phép đo thời gian, trong khi sustain là phép đo độ lớn.

A sound like a snare drum has a sharp attack & a quick release, while sounds like cymbals or chimes have fast attacks but slower releases that ring out over longer periods of time. Other sounds like violins have both slower attacks & releases. Attack, decay, & release sects of an envelope can also be curved instead of straight lines, which sometimes better approximates sound of real musical instruments. But important to remember: ADSR envelopes are always simplifications of reality. E.g., sustain of a piano note actually gradually decreases in volume over time until note is finally released. Revisit idea of ADSR envelopes in Chap. 10 to see how this can be applied when creating synthesized musical instruments.

– Âm thanh như trống snare có một cú đánh sắc nét & một cú nhả nhanh, trong khi âm thanh như chũm chọe hoặc chuông có các cú đánh nhanh nhưng các cú nhả chậm hơn, vang lên trong thời gian dài hơn. Các âm thanh khác như đàn violin có cả các cú đánh chậm hơn & các cú nhả. Các phần tấn công, suy giảm, & giải phóng của một lớp vỏ cũng có thể cong thay vì các đường thẳng, đôi khi gần đúng hơn với âm thanh của các nhạc cụ thực. Nhưng điều quan trọng cần nhớ: Các lớp vỏ ADSR luôn là sự đơn giản hóa của thực tế. Ví dụ, độ duy trì của một nốt nhạc piano thực sự giảm dần về âm lượng theo thời gian cho đến khi nốt nhạc cuối cùng được nhả ra. Xem lại ý tưởng về các lớp vỏ ADSR trong Chương 10 để xem cách áp dụng điều này khi tạo ra các nhạc cụ tổng hợp.

○ 7.3. Fourier. JEAN-BAPTISTE JOSEPH FOURIER was a French mathematician & physicist whose work in 19th century led to what we now call *Fourier Analysis*, a process through which we can decompose a complex sound signal into its constituent individual frequencies. Idea: can take any complex sound & determine all of frequencies that contribute to energy in signal – basically finding a set of sine waves that can be combined to represent a more complex waveform. Composition of sound signal by its frequency components is called signal's *spectrum*, & it can be generated through a mathematical operation called *Fourier transformation* – an essential part of all modern music production. For any given slice of time, spectrum might look like Fig. 7.1. But can also spread this information out over many time slices to visualize frequency & amplitude of a signal as it changes over longer periods of time. This visualization is called a *spectrogram* Fig. 7.5: A spectrogram shows intensity of frequencies in an audio signal over time. Heatmap colors correspond to intensity or energy at different frequencies. Time is represented on horizontal axis & frequency in kilohertz on vertical axis. A spectrogram typically shows time on horizontal axis, frequency on vertical axis, & intensity of different frequencies using heatmap colors. Warmer colors indicate more energy, while cooler colors intricate less energy.

– Jean-Baptiste Joseph Fourier là một nhà toán học & vật lý người Pháp, công trình của ông vào thế kỷ 19 đã dẫn đến cái mà ngày nay chúng ta gọi là *Phân tích Fourier*, một quá trình mà qua đó chúng ta có thể phân tích một tín hiệu âm thanh phức tạp thành các tần số riêng lẻ cấu thành nên nó. Ý tưởng: có thể lấy bất kỳ âm thanh phức tạp nào & xác định tất cả các tần số góp phần tạo nên năng lượng trong tín hiệu – về cơ bản là tìm một tập hợp các sóng sin có thể kết hợp để biểu diễn một dạng sóng phức tạp hơn. Thành phần của tín hiệu âm thanh theo các thành phần tần số của nó được gọi là *phổ* của tín hiệu, & nó có thể được tạo ra thông qua một phép toán gọi là *Biến đổi Fourier* – một phần thiết yếu của mọi hoạt động sản xuất âm nhạc hiện đại. Đối với bất kỳ lát cắt thời gian nào, phổ có thể trông giống như Hình 7.1. Nhưng cũng có thể phân tán thông tin này thành nhiều lát cắt thời gian để trực quan hóa tần số & biên độ của tín hiệu khi nó thay đổi trong khoảng thời gian dài hơn. Trực quan hóa này được gọi là *phổ đồ* Hình 7.5: Phổ đồ hiển thị cường độ tần số trong tín hiệu âm thanh theo thời gian. Màu sắc bản đồ nhiệt tương ứng với cường độ hoặc năng lượng ở các tần số khác nhau. Thời gian được biểu diễn trên trục ngang & tần số tính bằng kilohertz trên trục dọc. Một quang phổ thường hiển thị thời gian trên trục ngang, tần số trên trục dọc, & cường độ của các tần số khác nhau bằng cách sử dụng màu sắc bản đồ nhiệt. Màu ấm hơn biểu thị nhiều năng lượng hơn, trong khi màu lạnh hơn biểu thị ít năng lượng hơn.

This representation helps producers see & understand properties of sounds e.g. timbre & loudness. A spectrogram might show unwanted noise in background, or point out: audio is heavy on lower frequencies & sounds tiny. Through years of training, music producers can interpret spectrograms to visually understand how various frequency bands contribute to a mix.

– Biểu diễn này giúp nhà sản xuất thấy & hiểu các đặc tính của âm thanh, ví dụ như âm sắc & độ to. Một phổ đồ có thể hiển thị tiếng ồn không mong muốn trong nền hoặc chỉ ra: âm thanh nặng ở tần số thấp & nghe rất nhỏ. Qua nhiều năm đào tạo, nhà sản xuất âm nhạc có thể diễn giải phổ đồ để hiểu trực quan cách các dải tần số khác nhau đóng góp vào bản phối.

○ 7.4. Mixing & mastering. Recording all parts of a song is only 1 part of process of creating a finished piece of music that's ready to be shared with world. A music producer still has task of making all of various sonic layers work together as a cohesive whole. How does bass line complement rhythm? Does it interfere with percussion sounds? Are vocals getting drowned out by instrumentals? Are instruments competing with 1 another? Is overall mix too muddy or harsh or boomy? Process of *mixing* is about overall compositional structure of a song & finding balance between individual musical elements that have been recorded, sampled, or generated. Of course, *rough* mixes get put together throughout creation process as different parts of a song are recorded. E.g., a recording studio would need a rough mix of drums, bass, & keyboards before overdubbing vocals. But final mix is when all of elements are balanced, placed in space, & blended together to make an artistic statement. Mixing can be a complex process that involves planning, deep listening, & a lot of patience to get it right.
– Trộn & master. Thu âm tất cả các phần của một bài hát chỉ là 1 phần của quá trình tạo ra một bản nhạc hoàn chỉnh, sẵn sàng chia sẻ với thế giới. Nhà sản xuất âm nhạc vẫn có nhiệm vụ làm cho tất cả các lớp âm thanh khác nhau hoạt động cùng nhau như một tổng thể gắn kết. Dòng âm trầm bổ sung cho nhịp điệu như thế nào? Nó có can thiệp vào âm thanh của bộ gõ không? Giọng hát có bị lấn át bởi nhạc cụ không? Các nhạc cụ có cạnh tranh với nhau không? Bản phối tổng thể có quá đục, quá gắt hay quá ầm ầm không? Quá trình *mix* là về cấu trúc sáng tác tổng thể của một bài hát & tìm kiếm sự cân bằng giữa các yếu tố âm nhạc riêng lẻ đã được thu âm, lấy mẫu hoặc tạo ra. Tất nhiên, các bản phối *rough* được kết hợp lại trong suốt quá trình sáng tác khi các phần khác nhau của một bài hát được thu âm. Ví dụ: một phòng thu âm sẽ cần bản phối thô của trống, bass, & keyboard trước khi thu âm giọng hát. Nhưng bản phối cuối cùng là khi tất cả các yếu tố được cân bằng, đặt trong không gian, & hòa trộn với nhau để tạo nên một tuyên bố nghệ thuật. Việc phối nhạc có thể là một quá trình phức tạp đòi hỏi phải lập kế hoạch, lắng nghe sâu sắc và rất nhiều kiên nhẫn để làm đúng. *Deep listening* is process of paying close attention to relationship between musical elements in a song. This involves simultaneously being aware of compositional structure & frequency bandwidth of individual sounds. With deep listening, you are paying attention to different components & how they relate to each other. Is there a call-&-response between guitar & trumpet? Are they playing together? Should separate them by adjusting equalization to bring 1 out in foreground or will a simple change in volume do trick? Perhaps putting them in separate spaces within stereo spectrum will work, or using a 100 ms delay effect to place it in lateral space away from listener. There is a *lot* to experiment with, & it takes time to perfect art of mixing. It takes practice to develop this listening skill, but with practice you will become keenly aware of nuances. E.g., hear when instruments with same frequency bandwidth overlap. With digital tools that allow real-time frequency spectrum analysis, can actually see where they overlap. Don't worry about getting it right 1st time; mixing is a process that can require several iterations before you reach that "sweet" spot.

– *Nghe sâu* là quá trình chú ý kỹ đến mối quan hệ giữa các yếu tố âm nhạc trong một bài hát. Điều này bao gồm việc đồng thời nhận thức được cấu trúc sáng tác & băng thông tần số của từng âm thanh. Với việc nghe sâu, bạn sẽ chú ý đến các thành phần khác nhau & cách chúng liên quan đến nhau. Có sự tương tác qua lại giữa guitar & trumpet không? Chúng có chơi cùng nhau không? Nên tách chúng ra bằng cách điều chỉnh cân bằng để đưa 1 ra phía trước hay chỉ cần thay đổi âm lượng là đủ? Có lẽ việc đặt chúng vào các không gian riêng biệt trong phổ âm thanh nổi sẽ hiệu quả hoặc sử dụng hiệu ứng trễ 100 ms để đặt chúng vào không gian bên ngoài, cách xa người nghe. Có rất nhiều để thử nghiệm, & cần thời gian để hoàn thiện nghệ thuật phối nhạc. Cần phải thực hành để phát triển kỹ năng nghe này, nhưng với sự thực hành, bạn sẽ nhận thức sâu sắc về các sắc thái. Ví dụ, hãy lắng nghe khi các nhạc cụ có cùng băng thông tần số chồng lên nhau. Với các công cụ kỹ thuật số cho phép phân tích phổ tần số theo thời gian thực, bạn thực sự có thể thấy chúng chồng lên nhau ở đâu. Đừng lo lắng về việc phải làm đúng ngay từ lần đầu tiên; pha trộn là một quá trình có thể đòi hỏi nhiều lần lặp lại trước khi đạt được điểm "ngọt ngào" đó.

In this chap, going to refer to each individual layer of music as a *track* [This use of word "track" has a different meaning than a track on an album.]. Common practice to record vocals, drums, bass, & so on, on separate tracks & then mix them together to form final product. In TunePad, tracks are created using cells that can be assembled on a timeline as parts of a song. On a traditional mixing board, each *track* is a multifaceted tool used to shape sound elements in order to blend them cohesively with other elements in song Fig. 7.6: Mixing console with magnetic tape.

– Trong chương này, chúng ta sẽ gọi từng lớp nhạc riêng lẻ là một *track* [Cách sử dụng từ "track" này có nghĩa khác với một track trong album.]. Thực hành phổ biến là ghi âm giọng hát, trống, bass, & v.v., trên các track riêng biệt & sau đó trộn chúng lại với nhau để tạo thành sản phẩm cuối cùng. Trong TunePad, các track được tạo bằng cách sử dụng các ô có thể được lắp ráp trên một dòng thời gian như các phần của bài hát. Trên một bảng trộn âm truyền thống, mỗi *track* là một công cụ đa năng được sử dụng để định hình các thành phần âm thanh nhằm hòa trộn chúng một cách gắn kết với các thành phần khác trong bài hát Hình 7.6: Bàn trộn âm có băng từ.

A few of most important audio parameters to consider when mixing are panning, frequency manipulation (equalization), & gain (loudness of each track). This is also where you can apply audio effects like reverb, echo, or chorus (some of which cover in next chap).

– 1 số thông số âm thanh quan trọng nhất cần xem xét khi trộn là panning, điều chỉnh tần số (cân bằng), & gain (độ lớn của từng track). Đây cũng là nơi bạn có thể áp dụng các hiệu ứng âm thanh như reverb, echo hoặc chorus (một số trong số đó sẽ được đề cập trong chương tiếp theo).

∗ 7.4.1. Mixing tools. Before widespread availability of digital production tools, recording studios used multi-track magnetic tape to record multiple elements of a song e..g. bass, drums, guitar, keyboards, & vocals. Large mixing consoles were then used to record & play back to each element on tape Fig. 7.6. Studio infrastructure would route signals from individual tracks to & from mixing board. Today this is mainly done using software & visualization tools that allow for more flexibility & precision.

  – Công cụ trộn. Trước khi các công cụ sản xuất kỹ thuật số được sử dụng rộng rãi, các phòng thu âm đã sử dụng băng từ đa rãnh để ghi lại nhiều thành phần của một bài hát, ví dụ như bass, trống, guitar, keyboard, & giọng hát. Sau đó, các bàn điều khiển trộn lớn được sử dụng để ghi lại & phát lại từng thành phần trên băng Hình 7.6. Cơ sở hạ tầng phòng thu sẽ định tuyến tín hiệu từ các rãnh riêng lẻ đến & từ bảng trộn. Ngày nay, điều này chủ yếu được thực hiện bằng phần mềm & các công cụ trực quan hóa cho phép linh hoạt hơn & độ chính xác.

∗ 7.4.2. Panning & stereo. Most of music that you listen to has multiple *channels* of audio data. Stereophonic, or *stereo*, recordings use 2 different channels (left & right) to recreate spatial experience of listening to music in natural acoustic environments [Recordings with only 1 channel of audio data are called *monophonic*, or *mono*, recordings.]. I.e., when listening to music with headphones or earbuds, what you hear in your left ear is subtly (or note so subtly) different from what you hear in your right ear. Try removing 1 of your earbuds next time listening to music to see if can hear difference. When experience live music, you have a physical position in space relative to various musicians, vocalists, & other audio sources in room. Your 2 ears are also pointed in opposite directions, meaning they receive different versions of same audio scene. Music producers use stereo spectrum to recreate this experience. In general, humans have evolved to process sound from these 2 sources to create a mental map of physical space surrounding us. Think about a truck that drives past you. Even if you can't see truck, your brain is able to tell you where truck was & roughly how fast it was going based on frequency, loudness, & phase differences from your left & right ears.

  – Quét & âm thanh nổi. Hầu hết âm nhạc mà bạn nghe đều có nhiều *kênh* dữ liệu âm thanh. Bản ghi âm âm thanh nổi, hay *âm thanh nổi*, sử dụng 2 kênh khác nhau (trái & phải) để tái tạo trải nghiệm không gian khi nghe nhạc trong môi trường âm thanh tự nhiên [Bản ghi âm chỉ có 1 kênh dữ liệu âm thanh được gọi là bản ghi âm *đơn âm*, hay *đơn âm*.]. Ví dụ, khi nghe nhạc bằng tai nghe hoặc nút tai, những gì bạn nghe thấy ở tai trái sẽ hơi khác (hoặc hơi khác một chút) so với những gì bạn nghe thấy ở tai phải. Hãy thử tháo 1 nút tai ra vào lần tới khi nghe nhạc để xem bạn có thể nghe thấy sự khác biệt không. Khi trải nghiệm âm nhạc trực tiếp, bạn có một vị trí vật lý trong không gian so với nhiều nhạc sĩ, ca sĩ, & các nguồn âm thanh khác trong phòng. 2 tai của bạn cũng hướng về các hướng ngược nhau, nghĩa là chúng nhận được các phiên bản khác nhau của cùng một cảnh âm thanh. Các nhà sản xuất âm nhạc sử dụng phổ âm thanh nổi để tái tạo trải nghiệm này. Nhìn chung, con người đã tiến hóa để xử lý âm thanh từ 2 nguồn này để tạo ra bản đồ tinh thần về không gian vật lý xung quanh chúng ta. Hãy nghĩ về một chiếc xe tải chạy ngang qua bạn. Ngay cả khi bạn không thể nhìn thấy xe tải, não của bạn vẫn có thể cho bạn biết xe tải đang ở đâu & tốc độ ước chừng của nó dựa trên tần số, độ lớn, & độ lệch pha từ tai trái & tai phải của bạn. Gửi ý kiến phản hồi

  Panning of a track refers to its position in this stereo spectrum. In practice, this means how much of track comes out of left & right speakers. Producers can create more depth to a song & replicate live recordings by controlling panning of tracks. Can almost think of it like arranging musicians on a stage in front of a live audience. Humans are also better at perceiving directionality of sound at higher frequencies, i.e., can easily tell which direction a high-pitched hi-hat sound is coming from, but we have a hard time telling which direction a bass line is coming from. As a result, producers will often pan higher-pitched sounds to left or right, while leaving lower-pitched sounds more in center of a mix.

  – *Panning* của một bản nhạc đề cập đến vị trí của nó trong phổ âm thanh nổi này. Trong thực tế, điều này có nghĩa là có bao nhiêu bản nhạc phát ra từ loa trái & phải. Nhà sản xuất có thể tạo thêm chiều sâu cho một bài hát & sao chép các bản ghi âm trực tiếp bằng cách kiểm soát việc panning các bản nhạc. Gần như có thể nghĩ về nó giống như việc sắp xếp các nhạc công trên sân khấu trước khán giả trực tiếp. Con người cũng giỏi hơn trong việc nhận biết hướng của âm thanh ở tần số cao hơn, tức là có thể dễ dàng biết được âm thanh hi-hat cao độ phát ra từ hướng nào, nhưng chúng ta khó có thể biết được hướng của dòng âm trầm phát ra từ hướng nào. Do đó, nhà sản xuất thường sẽ pan các âm thanh có cao độ cao sang trái hoặc phải, trong khi để các âm thanh có cao độ thấp hơn ở giữa bản phối.

  In TunePad, can adjust pan, gain, & frequency elements of different cells using mixer interface shown in Fig. 7.7: Mixing interface in TunePad allows you to adjust gain, pan, & frequency response for each track in a mix. Also possible to apply these effects in code using Python's `with` construct. An example of a pan effect that shifts stereo balance of 2 `playNote` instructions to far left speaker.

```
with pan(-1.0):
    playNote([ 31, 35, 38 ], beats = 4)
    playNote([ 31, 35, 38 ], beats = 4)
```

Values of pan parameter ranges from -1.0 (full left speaker) to 1.0 (full right speaker). A value of 0.0 evenly splits sound. `with` keyword in TunePad applies pan effect to all of statements indented directly below it.

∗ 7.4.3. Gain. *Gain* of a track is related to its loudness. Gain isn't quite volume, but works as kind of a multiplier to an audio signal's amplitude. When mixing boards were physical pieces of equipment, gain related to amount of power a signal had at each stage of signal flow. Now, gain has a similar meaning & can be used to make a track more or less prominent. E.g., a producer may choose to make bass drum of a dance track more prominent while decreasing gain of vocal melody. Gain is commonly measured in decibels. Negative values reduce loudness of a track, & positive values increase it from its original volume.

– *Gain* của một bản nhạc liên quan đến độ to của bản nhạc đó. Gain không hẳn là âm lượng, nhưng hoạt động như một loại hệ số nhân với biên độ của tín hiệu âm thanh. Khi bảng trộn là các thiết bị vật lý, gain liên quan đến lượng công suất mà tín hiệu có ở mỗi giai đoạn của luồng tín hiệu. Bây giờ, gain có ý nghĩa tương tự & có thể được sử dụng để làm cho một bản nhạc nổi bật hơn hoặc ít nổi bật hơn. Ví dụ, một nhà sản xuất có thể chọn làm cho tiếng trống trầm của một bản nhạc khiêu vũ nổi bật hơn trong khi giảm gain của giai điệu giọng hát. Gain thường được đo bằng decibel. Các giá trị âm làm giảm độ to của một bản nhạc, & các giá trị dương làm tăng nó so với âm lượng ban đầu của nó.

∗ 7.4.4. Frequency bands. When mixing tracks together, often helpful to break full frequency spectrum into *bands* that correspond to different ranges of frequencies. Each band is meant to capture a particular musical element, although, of course, this varies between genres & specific songs. Producers often split a mix into 7 bands: sub-bass, bass, low midrange, midrange, upper midrange, presence, & brilliance. 1 reason to think in terms of bandwidth: when sounds have same bandwidth an acoustic phenomenon called "masking" can occur. Masking is when 1 sound overpowers another sound s.t. sound that is overpowered is not audible.

[Table: Band: Frequency range: Description]

· Sub-bass: 20–60 Hz: Adds power & deepness to bass & drums

· Bass: 60–250 Hz: Captures core fundamentals of bass & drum sound

· Low midrange: 250–500 Hz: Captures overtones of lower instruments, as well as instruments like viola & alto saxophone

· Midrange: 500–2000 Hz: Captures melodic instruments e.g. violin, flute, & human voice

· Upper midrange: 2000–4000 Hz: Captures overtones of melodic instruments as well as core of some higher instruments

· Presence: 4000–6000 Hz: Captures overtones of higher instruments as well as adding precision & clarity to sounds

· Brilliance: 6000–20000 Hz: Captures upper overtones of all instruments

Humans are most sensitive to frequencies between 1 kHz & 4 kHz. Looking at frequency values for each band, might notice: frequency ranges are not even close to same size. E.g., sub-bass band covers a range of only 40 Hz (from 20 Hz–60 Hz), while presence band covers 2000 Hz (from 4000 Hz–6000 Hz). Reason: human perception of pitch isn't linear. When move up 1 octave, doubling frequency of a pitch, i.e., each consecutive musical octave covers double frequency range (or bandwidth) of octave below it. As a result, higher frequency bands naturally cover large portions of frequency range & generate more energy.

– Con người nhạy cảm nhất với tần số từ 1 kHz & 4 kHz. Khi xem xét các giá trị tần số cho từng băng tần, bạn có thể nhận thấy: các dải tần số thậm chí không gần bằng nhau. Ví dụ, dải âm trầm phụ chỉ bao phủ một dải tần 40 Hz (từ 20 Hz–60 Hz), trong khi dải hiện diện bao phủ 2000 Hz (từ 4000 Hz–6000 Hz). Lý do: nhận thức của con người về cao độ không phải là tuyến tính. Khi di chuyển lên 1 quãng tám, tần số của một cao độ tăng gấp đôi, tức là mỗi quãng tám âm nhạc liên tiếp bao phủ gấp đôi dải tần số (hoặc băng thông) của quãng tám bên dưới nó. Do đó, các dải tần số cao hơn tự nhiên bao phủ các phần lớn của dải tần & tạo ra nhiều năng lượng hơn.

◦ 7.5. Filters & equalization. If 2 instruments overlap in their natural pitch range, it can be difficult to distinguish 1 from the other, which can lead to muddiness. Producer will want to ensure: each musical element is distinct & audible. Think of a painter recreating an ocean scene. Painter wants each element of scene to stand out clearly – perhaps sky, a boat, shore, & ocean itself. If ocean, sky, & land are all same shade of blue, a viewer won't be able to interpret & appreciate scene.

– Bộ lọc & cân bằng. Nếu 2 nhạc cụ chồng lên nhau trong phạm vi cao độ tự nhiên của chúng, có thể khó phân biệt nhạc cụ này với nhạc cụ kia, điều này có thể dẫn đến sự hỗn loạn. Nhà sản xuất sẽ muốn đảm bảo: mỗi yếu tố âm nhạc đều riêng biệt & có thể nghe được. Hãy nghĩ đến một họa sĩ đang tái hiện một cảnh đại dương. Họa sĩ muốn mỗi yếu tố của cảnh nổi bật rõ ràng – có thể là bầu trời, một chiếc thuyền, bờ biển, & chính đại dương. Nếu đại dương, bầu trời, & đất liền đều có cùng một sắc thái xanh, người xem sẽ không thể diễn giải & đánh giá cao cảnh đó.

Most important tools that a producer has to achieve balance across frequency spectrum are *filters & equalizers*. These tools reduce (*attenuate*) or increase (*boost*) certain frequency ranges in a track to make them more or less prominent in a mix, & producers will often "carve out" room in frequency spectrum for each track. Process of adjusting levels of frequency bands within a signal is called *equalization* (or *EQ*). When adjust bass & treble dials of sound system in your car, you are equalizing frequencies just as a producer might while adjusting sound of an instrument.

– Các công cụ quan trọng nhất mà nhà sản xuất phải có để đạt được sự cân bằng trên toàn bộ phổ tần số là *bộ lọc & bộ cân bằng*. Các công cụ này làm giảm (*làm suy yếu*) hoặc tăng (*tăng cường*) các dải tần số nhất định trong một bản nhạc để làm cho chúng nổi bật hơn hoặc ít nổi bật hơn trong bản phối, & nhà sản xuất thường sẽ "tạo ra" chỗ trong phổ tần số cho mỗi bản nhạc. Quá trình điều chỉnh mức độ của các dải tần số trong một tín hiệu được gọi là *cân bằng* (hoặc *EQ*). Khi điều chỉnh núm xoay âm trầm & âm bổng của hệ thống âm thanh trong xe hơi của bạn, bạn đang cân bằng tần số giống như một nhà sản xuất có thể làm khi điều chỉnh âm thanh của một nhạc cụ.

Can think of an audio filter kind of like a filter that you would use to purify drinking water. A water filter is designed to let small particles (like water molecules & minerals) pass through while blocking larger particles (like bacteria). An audio filter achieves a similar effect except for sound, allowing certain frequencies of an audio signal to pass through unaffected while blocking or reducing other frequencies. A filter's *response curve* is a graph that shows which frequencies are allowed to pass through & which are filtered out. There are several types of filters common used in music production including lowpass, highpass, lowshelf, highshelf, bandpass, notch, & peaking. Describe several of these filters below along with an example of applying these filters with Python code in TunePad. Most production software (including TunePad) include built-in equalizer tools that let you combine & precisely adjust various filter types. Understanding how each filter works will help use these tools.

– Có thể nghĩ về bộ lọc âm thanh giống như bộ lọc mà bạn sử dụng để làm sạch nước uống. Bộ lọc nước được thiết kế để cho các hạt nhỏ (như phân tử nước & khoáng chất) đi qua trong khi chặn các hạt lớn hơn (như vi khuẩn). Bộ lọc âm thanh đạt được hiệu ứng tương tự ngoại trừ âm thanh, cho phép một số tần số nhất định của tín hiệu âm thanh đi qua mà không bị ảnh hưởng trong khi chặn hoặc giảm các tần số khác. *response curve* của bộ lọc là đồ thị hiển thị tần số nào được phép đi qua & tần số nào bị lọc ra. Có một số loại bộ lọc thường được sử dụng trong sản xuất âm nhạc bao gồm thông thấp, thông cao, kệ thấp, kệ cao, thông dải, khía, & peaking. Mô tả một số bộ lọc này bên dưới cùng với ví dụ về cách áp dụng các bộ lọc này bằng mã Python trong TunePad. Hầu hết các phần mềm sản xuất (bao gồm TunePad) đều bao gồm các công cụ cân bằng tích hợp cho phép bạn kết hợp & điều chỉnh chính xác nhiều loại bộ lọc khác nhau. Hiểu cách thức hoạt động của từng bộ lọc sẽ giúp sử dụng các công cụ này.

* 7.5.1. Lowpass filer. A *lowpass filter* allows frequencies below a certain threshold – called *cutoff frequency* – to pass through unaltered. Frequencies above this threshold are reduced (or attenuated). A frequency parameter specifies location of cutoff, & a Q parameter determines how sharp or steep this cutoff is Fig. 7.8: Lowpass filter response curve.

  – Bộ lọc thông thấp. Một *bộ lọc thông thấp* cho phép các tần số dưới ngưỡng nhất định – được gọi là *tần số cắt* – đi qua mà không bị thay đổi. Các tần số trên ngưỡng này bị giảm (hoặc suy yếu). Một tham số tần số chỉ định vị trí cắt, & một tham số Q xác định mức độ sắc nét hoặc dốc của điểm cắt này Hình 7.8: Đường cong đáp ứng của bộ lọc thông thấp.

  Lowpass filers might be applied if a track sounds too bright, or to remove some of higher partials of a bass instrument to make room for other instruments in a mix, or even to remove some unwanted studio sound e.g. buzzing from equipment.

  – Bộ lọc thông thấp có thể được áp dụng nếu một bản nhạc nghe quá sáng hoặc để loại bỏ một số phần cao hơn của nhạc cụ trầm để tạo chỗ cho các nhạc cụ khác trong bản phối hoặc thậm chí để loại bỏ một số âm thanh phòng thu không mong muốn, ví dụ như tiếng ù từ thiết bị.

  In TunePad, can add a lowpass filter directly in Python code. Example below applies a constant lowpass filter with a cutoff of 100 Hz to reduce higher frequencies in drums.

  ```
  with lowpass(frequency = 100):
      playNote(0, beats = 1)
      playNote(2, beats = 1)
      playNote(0, beats = 1)
      playNote(2, beats = 1)
  ```

  `with` keyword starts a special Python structure that applies an effect to all of statements indented below it. In this case, TunePad's lowpass filter is applied to 4 drum sounds.

  All of filters that you can code in TunePad have same basic structure. Use `with` keyword followed by filter name. Filters have 1 required parameter & several optional parameters. Only required parameter is frequency, which represents cutoff frequency for each filter. Filters also have an optional Q parameter, which specifies how sharp or spread out frequency cutoff is around target frequency.

  [Table: Parameter: Description: Required?]

  · `Frequency`: Cutoff or central frequency specified in Hz: Yes

  · `Q`: Typically sharpness of cutoff frequency: No

  · `Beats`: How long effect lasts in beats: No

  · `Start`: How long to delay in beats before starting effect: No

  · `Gain`: Some filters like peaking, lowshelf, & highshelf use a gain parameter to specify intensity of boost or attenuation in decibels: No

* 7.5.2. Highpass filter. A *highpass filter* is opposite of a lowpass filter; it passes frequencies *above* cutoff & reduces frequencies below. As with lowpass filters, frequency parameter sets cutoff frequency & Q parameter specifies sharpness of cutoff Fig. 7.9: Highpass filter response curve.

  A highpass filter might be applied if a track sounds muddy because it has too much bass, or to remove unwanted noise e.g. a low hum from equipment. A TunePad example that uses a highpass filter to cut out sounds lower than 4000 Hz (4kHz) for an instrument playing a melody:

  ```
  with highpass(frequency = 4000):
      playNote(31, beats = 0.5)
  ```

```
        playNote(35, beats = 0.5)
        playNote(38, beats = 1)
        playNote(36, beats = 1)
```

- ∗ 7.5.3. Bandpass filter. A *bandpass filter* reduces frequencies above & below a specified band of frequencies; a bandpass is equivalent of applying both a lowpass & a highpass filter. Specify middle of band using frequency parameter & width of band using Q parameter. Higher Q, sharper cutoff, & narrower band of frequencies that can pass through. Bandpass filters allow us to precisely target a track's frequency range. Might use a bandpass filter to bring out vocals or melody of a song by reducing everything else Fig. 7.10: Bandpass filter response curve.
  In example, apply a constant bandpass filter with a center frequency of 130 Hz (≈ C3 or MIDI 48) to a short melody to bring melody out in overall musical texture.

```
    with bandpass(frequency = 130, Q = 0.7):
        playNote(48, beats = 0.5)
        playNote(52, beats = 0.5)
        playNote(55, beats = 1)
        playNote(53, beats = 1)
```

- ∗ 7.5.4. Notch filter. A *notch filter* is opposite of a bandpass filter. Rather than bringing out a band of frequencies, a notch filter *reduces* frequency band while all other frequencies pass through freely. Like with bandpass, frequency parameter specifies center of this frequency band & Q parameter sets width Fig. 7.11: Notch filter response curve.
  In example, apply a constant notch filter with a center frequency of 440 Hz (≈ A4or MIDI 69) to a short selection of chords to reduce prevalence in overall musical texture.

```
    with notch(frequency = 440):
        playNote([69, 72, 76], beats = 4)
        playNote([69, 72, 76], beats = 4)
```

- ∗ 7.5.5. Peaking filter. Peaking filters are frequently used in *parametric equalizers* to boot or attenuate sounds at a target frequency. Parametric equalizers are a type of equalizer that offer precise control of center frequencies & Q (how spread out or tight filter is around center frequency). Using these filters, there's a 3rd parameter called *gain* that controls how much signal is boosted or attenuated. Gain is measured in decibels. A positive gain will boost frequencies targeted by filter, & a negative gain will attenuate them Fig. 7.12: Peaking filer response curve.
  – Bộ lọc đỉnh. Bộ lọc đỉnh thường được sử dụng trong *bộ cân bằng tham số* để khởi động hoặc làm suy yếu âm thanh ở tần số mục tiêu. Bộ cân bằng tham số là một loại bộ cân bằng cung cấp khả năng kiểm soát chính xác tần số trung tâm & Q (mức độ lan tỏa hoặc chặt chẽ của bộ lọc xung quanh tần số trung tâm). Khi sử dụng các bộ lọc này, có một tham số thứ 3 được gọi là *gain* kiểm soát mức độ tín hiệu được tăng cường hoặc suy yếu. Độ khuếch đại được đo bằng decibel. Độ khuếch đại dương sẽ tăng cường tần số mục tiêu của bộ lọc, & độ khuếch đại âm sẽ làm suy yếu chúng Hình 7.12: Đường cong phản hồi của bộ lọc đỉnh.
- ∗ 7.5.6. Lowshelf & highshelf filters. Lowshelf & highshelf filters boost or attenuate sounds beyond target frequency. They are called *shelves* due to plateau shape of their response curves. As with peaking filters, frequency parameter specifies cutoff, & gain parameter specifies how much boost or attenuation to give to frequencies beyond target. Negative gain values attenuate & positive gain values boost Fig. 7.13: Low shelf & high shelf response curves.
  – Bộ lọc Lowshelf & highshelf. Bộ lọc Lowshelf & highshelf tăng cường hoặc làm suy yếu âm thanh vượt quá tần số mục tiêu. Chúng được gọi là *shelves* do hình dạng cao nguyên của đường cong phản hồi của chúng. Giống như bộ lọc đỉnh, tham số tần số chỉ định điểm cắt, tham số & gain chỉ định mức tăng cường hoặc làm suy yếu nào để cung cấp cho các tần số vượt quá mục tiêu. Giá trị khuếch đại âm làm suy yếu & giá trị khuếch đại dương tăng cường Hình 7.13: Đường cong phản hồi low shelf & high shelf.

○ 7.6. Mastering. After individual tracks have been adjusted in relation to 1 another, *mastering* is process of taking this final mix & polishing it by adjusting global parameters e.g. dynamic range & frequency. In early days, there were mastering engineers who specialized in process of mastering final mixes. In fact, there were studios dedicated to mastering, so can imagine that this last leg of production process deserves as much attention as the rest. Mastering is particularly important because you want your mix to sound good on as many devices as possible, so there is a delicate process of balancing elements in mix to optimize listening experience across different media. Want your mix to sound as good over speakers as it does over headphones.

– Mastering. Sau khi từng track riêng lẻ đã được điều chỉnh liên quan đến 1 track khác, *mastering* là quá trình thực hiện bản phối cuối cùng này & đánh bóng nó bằng cách điều chỉnh các thông số toàn cục, ví dụ như dải động & tần số. Vào những ngày đầu, có những kỹ sư mastering chuyên về quá trình mastering các bản phối cuối cùng. Trên thực tế, đã có những studio chuyên về mastering, vì vậy có thể tưởng tượng rằng giai đoạn cuối cùng của quy trình sản xuất này cũng đáng được quan tâm như phần còn lại. Mastering đặc biệt quan trọng vì bạn muốn bản phối của mình nghe hay trên càng nhiều thiết bị càng tốt, do đó, có một quy trình tinh tế để cân bằng các yếu tố trong bản phối nhằm tối ưu hóa trải nghiệm nghe trên các phương tiện khác nhau. Muốn bản phối của bạn nghe hay qua loa cũng như qua tai nghe.

Traditionally, mastering is done using tools like equalization, compression, limiting, & stereo enhancement. Recall from Chap. 3: dynamic range refers to difference between quietest & loudest volumes in a selection of audio. This can be adjusted through use of *dynamic range compression*, or *compressors*. Compressors reduce highest volumes in a mix & amplify lowest volumes, which shrinks overall dynamic range of audio. This ensures: listener can hear full range of volumes clearly. Can think of this like an action movie where a character might whisper a secret right before an explosion. Dynamic range compression is 1 possible tool that could make sure that both of these sounds are clear to audience by reducing volume of explosion & increasing volume of whisper.

– Theo truyền thống, việc master được thực hiện bằng các công cụ như cân bằng, nén, giới hạn, & tăng cường âm thanh nổi. Nhớ lại từ Chương 3: dải động đề cập đến sự khác biệt giữa âm lượng nhỏ nhất & to nhất trong một lựa chọn âm thanh. Điều này có thể được điều chỉnh thông qua việc sử dụng *nén dải động* hoặc *máy nén*. Máy nén giảm âm lượng cao nhất trong bản phối & khuếch đại âm lượng thấp nhất, làm giảm dải động tổng thể của âm thanh. Điều này đảm bảo: người nghe có thể nghe rõ toàn bộ dải âm lượng. Có thể coi đây giống như một bộ phim hành động, trong đó một nhân vật có thể thì thầm một bí mật ngay trước khi xảy ra vụ nổ. Nén dải động là 1 công cụ khả thi có thể đảm bảo rằng cả hai âm thanh này đều rõ ràng đối với khán giả bằng cách giảm âm lượng của vụ nổ & tăng âm lượng của tiếng thì thầm.

Producer also considers frequency domain when creating final product. Instead of thinking on a track-by-track level as in mixing, producer can think in terms of different bands of frequencies. By this stage, our different bands of frequencies should already be well balanced, & goal: polish overall mix using EQ & filters.

– Nhà sản xuất cũng xem xét miền tần số khi tạo ra sản phẩm cuối cùng. Thay vì suy nghĩ theo từng track như trong quá trình trộn, nhà sản xuất có thể suy nghĩ theo các dải tần số khác nhau. Đến giai đoạn này, các dải tần số khác nhau của chúng ta đã được cân bằng tốt, & mục tiêu: đánh bóng bản phối tổng thể bằng EQ & bộ lọc.

Lastly, a final version of track is generated & exported into final format. A major consideration is where & how music is going to be distributed. A producer might think about a person watching a music video on a laptop through YouTube, vs. someone listening to radio in a car, vs. someone streaming audio online, vs. someone with a physical CD or even vinyl recording. Music for streaming & other forms of distribution is almost always *compressed*, i.e. size of final audio file is much, much smaller than original uncompressed audio data. Note: this is not related to dynamic range compression. Compressing audio means: some of data is discarded to decrease amount of information that has to be transmitted over internet to avoid buffering delays or to store more songs on a CD. There are complex computer algorithms that decide what data is discarded so that listeners won't even notice a reduction in quality. Examples of file formats that use this form of compression include `.mp3, .aac` files.

Some audio formats forgo this compression in favor of increased sound quality & fidelity. These files contain raw audio data. These audio files are generally larger, taking up more file space. Examples of this include `.wav & .aiff` file formats.

Mixing & mastering can be a tedious process requiring attention to detail & a keen, well-trained ear. Becoming an accomplished professional can take many years of experience, & have introduced just a few of parameters & tools at your disposal. Don't stress over this, especially at 1st! Mixing & mastering are 2 of most difficult concepts in producing music, but having a grasp of them can greatly elevate music you create. Best way to gain this familiarity is through experimentation & thoughtful listening. Listening critically to music that has been professionally produced will help develop your ear & unlock a whole new world of possibilities to your music.

– Mixing & mastering có thể là một quá trình tẻ nhạt đòi hỏi sự chú ý đến từng chi tiết & một đôi tai nhạy bén, được đào tạo bài bản. Để trở thành một chuyên gia thành đạt có thể mất nhiều năm kinh nghiệm, & đã giới thiệu một vài thông số & công cụ mà bạn có thể sử dụng. Đừng căng thẳng về điều này, đặc biệt là lúc đầu! Mixing & mastering là 2 trong số những khái niệm khó nhất trong sản xuất âm nhạc, nhưng nắm bắt được chúng có thể nâng cao đáng kể âm nhạc bạn tạo ra. Cách tốt nhất để có được sự quen thuộc này là thông qua thử nghiệm & lắng nghe một cách chu đáo. Việc lắng nghe một cách phê phán âm nhạc được sản xuất một cách chuyên nghiệp sẽ giúp phát triển đôi tai của bạn & mở ra một thế giới hoàn toàn mới về khả năng cho âm nhạc của bạn.

- 7.7. Dynamic effects in TunePad. Many of TunePad effects described can also be applied dynamically to create a wide variety of sounds. Basic idea: instead of passing 1 constant value for a parameter, instead pass a list of numbers that describes how that parameter will change over time. Duration of dynamic effect is specified using a `beats` parameter. Can try these dynamic effects & filters for yourself at https://tunepad.com/examples/efects. An example that uses a lowpass filter to create a wha-wha effect at beginning of a piano note. Cutoff frequency of filter moves rapidly back & forth between 200 Hz & 800 Hz over duration of 1 beat. Note itself plays for 3 beats, so only 1st beat of note has effect applied:

```
# creates a wha-wha effect by quickly changing
# the cutoff of a lowpass filter between 200 and 800hz
with lowpass(frequency = [200, 800, 200, 800, 200, 800], beats=1):
    playNote(47, beats=3)
```

Other effects like pan & gain work same way. Can create dynamic changes by passing in a list of values & `beats` parameters. E.g., this code gradually sweeps a lowpass filter from 20 Hz to 750 Hz & back again. At same time, it moves sound across stereo field from left to right & back again. To do this, it nests pan effect inside of lowpass filter effect.

```
with lowpass(frequency = [20, 750, 20], beats = 40):
    with pan(value = [-1, 1, -1], beats = 40):
        playNote(16, beats=40)
```

Interlude following this chap shows how to add other dynamic effects to TunePad projects using Python code.

- Interlude 7: Creative Effects. In this interlude going to work with audio effects – e.g. filtering & gain – as creative compositional tools. In Chap. 7, saw how static effects like filters & gain can be used in mixing process. These same effects can also be used as compositional tools to help craft a cohesive soundscape. They can build tension, add contrast, & drive a song forward. Can follow along online with this TunePad project `https://tunepad.com/interlude/efects`.
  - Option 1: Fades. Fades are 1 of most common creative effects. Often ear fades in or out at beginnings or ends of songs. These fades are essentially gradually moving from 1 volume to another: for a fade-in, low to high; for a fade-out, high to low.

    Can easily add fades to our music in TunePad using gain effect & a list. If gain class is passed a list, it moves evenly from each value to next, over whole duration of beats. For a fade-in, want to move from silent to full volume. Say have a function called `phrase` that plays 8 beats of a melody. Can use gain class with a list as values input:

    ```
    with gain([0.0, 1.0], beats = 8):
        phrase()
    ```

    Or, if want to fade out our phrase:

    ```
    with gain([1.0, 0.0], beats = 8):
        phrase()
    ```

    What if want to control speed of our fade? Because gain moves smoothly from each value, can shape our fade by adding more intermediate values $\in [0.0, 1.0]$. Can think of this as specifying more points along curve. If want our fade-in to be quieter for longer, could add additional values closer to 0:

    ```
    with gain([0.0, 0.05, 0.1, 0.15, 1.0], beats = 8):
        phrase()
    ```

    Can see difference graphically below Fig. 7.14: Graph of 2 methods for fading audio in.

    With method 1, ramp-up in volume is consistent & gradual over entire duration. But with method 2, ramp-up is slow until 7th beat. These last 2 beats have greatest increase in gain, increasing from 0.15–1.0.
  - Option 2: Filter sweeps. Filter sweeps are a great way to build tension in a song. Principle behind a filter sweep: apply a filter to a sect of a song – which blocks some of frequencies – & then gradually remove it, revealing full spectrum of audio. This can either highpass or lowpass filters, each imparting a different sound to our track. A lowpass filter will start with just low frequencies & will gradually reveal upper overtones while a highpass filter will do opposite.

    In TunePad, this is going to look similar to our fades. Pass our filter a list of 2 more numbers. However, these numbers will represent frequencies, so need to decide which values to use. Like with our fades, want a higher value & a lower value. Exact values depend largely on specific bandwidth of our instrument, desired speed of our sweep, & personal taste. For our highpass sweep, want our initial value to block most of frequencies, & our final value to include most of spectrum of audio. Our higher value can be a frequency near upper range of human hearing; for our lower value, could choose a frequency closer to bottom end of human hearing. Look now at code for a basic highpass filter sweep using our phrase function & values 22000 Hz & 22 Hz:

    ```
    with highpass([22000, 22], beats = 8):
        phrase()
    ```

    For lowpass filter sweep, also want to initially block most of component frequencies & gradually reveal upper frequencies of audio. Due to how lowpass filters work, lower number should be 1st. Can use initial value of 22 Hz to block most of frequencies & end at 4000 Hz, revealing most of our spectrum. Look now at code for a basic lowpass filter sweep, again using our phrase function with our chosen values:

    ```
    with lowpass([22, 4000], beats = 8):
        phrase()
    ```

    If want to have greater control of shape our our sweep, can use same principle behind our gain. Adding more values allows us to better sculpt our resulting sound. If want our filter to evolve slowly at 1st, can add additional values near our starting frequency. An example of this with our highpass sweep:

    ```
    with highpass([22000, 18000, 14000, 8000, 22], beats = 8):
        phrase()
    ```

    Most of audio spectrum is revealed between 8000 Hz & 22 Hz values, which occurs on last 2 beats. Try experimenting with different values.

- Option 3: Reverb. 1 of most important effects that producers use is reverb. When sound waves move through a physical space, some of those waves bounce back to listener. Waves that bounce back are heard as softer. Think of how sound reverberates through a concert hall, or even your bathroom. Can recreate this reverberation through applying reverb to our track.

  There are different mathematical strategies for applying reverb to audio. TunePad uses *convolution reverb*, which is 1 of most common varieties. This takes an audio sample called an *Impulse Response* from a real-world space that represents how different frequencies resonate through physical space & essentially maps this Impulse Response over our selection of audio.

  In TunePad, specify our impulse response by choosing from a selection of preset values:

  * Hall
  * Gallery
  * Museum
  * Library
  * Theater
  * Underpass
  * Space Echo 2

  This is 1st argument. Capitalization, spacing, & punctuation are ignored. Like other effect classes, can also optionally specify amount of beats effect should last & a delayed start parameter. This effect also uses a parameter called `wet`. This specifies how much reverb is applied. A value of 1.0 represents maximum reverb, & a value of 0.0 represents no reverb. Can also pass a list of numbers to create a change over time. Each number will be evenly distributed over duration of effect specified by `beats` parameter. In example, gradually applying "Underpass" reverb, which is most reverberant, to our phrase function:

  ```
  with reverb(impulse = "Underpass", wet = [0.0, 1.0], beats = 8):
      phrase()
  ```

  Experiment with different impulse responses!

- 8. Note-based production effects. This chap covers a variety of production effects that can add sophistication & depth to your sound. All of effects are variations on same theme – instead of playing 1 note, play a series of notes, each offset slightly in time or pitch. From this basic technique, work through a wide variety of effects including echos, arpeggiation, chorus sounds, a swung beat, & a phaser. To create these effects, define some of our own functions that make use of TunePad's `rewind, fastForward` features. This will also give us a good chance to review loops, variables, & parameters from earlier chaps. Once master these basic techniques, it will open a range of audio effects that you can expand & customize to define your own unique sound. Also cover how to combine these techniques with other effects & filters to add even more flexibility.

  - 8.1. Out-of-tune piano effect. Start with 1 of more straightforward effects, an out-of-tune piano. As always, can try this example by visiting `https://tunepad.com/examples/out-of-tune`. Recall from Chap. 3: space between separate notes on a 12-tone chromatic scale can be subdivided into even units called *cents* – just imagine 100 individual smaller notes between each adjacent key on piano keyboard. On an instrument like a violin or trombone, can play notes that are slightly out of tune or that glide between 1 note & another. Can do sth similar in TunePad by using decimal numbers instead of integer values when call `playNote` function:

    ```
    playNote(36.5) # plays an out-of-tune C
    ```

    For this line of code, a value of 36.5 sits exactly halfway between a C (MIDI value 36) & a C# (MIDI value 37). I.e., it's a pure C *detuned* by 50 cents Fig. 8.1: Intermediate pitches between C & C#.

    There are plenty of artistic reasons to create sounds that are out of tune – like if wanted to create an eerie melody for a horror movie. To get this kind of effect, could just randomly *detune* some of notes in our melody by small amounts, & get sth that sounded off key. But a more interesting approach that gives us additional texture (& eerie dissonant overtones) would be to play several notes at same time, each slightly detuned from 1 another. This actually approximates what a real out-of-tune piano sounds like. Notes on a piano are produced when a hammer mechanism inside instrument his multiple individual strings at same time (3 strings for most notes). When those individual strings are off from 1 another, hear an entirely different sound than you would get from just 1 string being out of tune. Honky tonk pianos are intentionally tuned so that 3 strings for each note are detuned slightly from 1 another to get warped harmonics. A simple Python function that approximates that dissonant sound in TunePad:

    ```
    def errieNote(note, beats = 1, velocity = 100):
        volume = velocity / 3.0
        for i in range(3):
            offset = random() -0.5
            playNote(note + offset, beats = beats, velocity = volume)
            rewind(beats)
        fastForward(beats)
    ```

Walk through this function 1 line at a time. If this example makes sense, all of other functions that we create later in this chap should be easier to understand. On line 1, use `def` keyword to define our own Python function.

```
def errieNote(note, beats = 1, velocity = 100):
```

Flip back to Chap. 4 for an in-depth description of function defs, but main thing to remember: creating our own functions lets us build up our own musical toolbox to help create more complex compositions. In this case adding `eerieNote` as our newest tool. Also define 3 *parameters* for this function called `note, beats, & velocity`, each of which is listed inside parentheses. Use `note` for pitch value want to play; `beats` for duration of note; & `velocity` to approximate overall volume of sound. Might notice: these parameters are exactly same as `playNote` function. That's intentional because it will make it easy to swap out `playNote` function for our new `eerieNote` function in other parts of our project. 1 other thing to notice: `beats, velocity` are examples of what's called an *optional* parameter. I.e., have defined a default value that Python will use if don't otherwise specify sth. Default value for `beats` is 1, & default value for `velocity` is 100. So, e.g., each of these lines of code will do same thing, & can use them interchangeably:

```
eerieNote(36)
eerieNote(36, beats = 1)
eerieNote(36, beats = 1, velocity = 100)
eerieNote(36, 1, 100)
eerieNote(36)
```

In all of these cases, `beats, velocity` are same as their default values. Can call function with other values too:

```
eerieNote(40, beats = 0.5, velocity = 50)
eerieNote(40, beats = 1.5)
eerieNote(40, velocity = 120)
eerieNote(40, 2.0, 50)
```

Go back to `eerieNote` function. On line 2 define a *variable* called `volume` that will help us adjust loudness of our individual notes.

```
volume = velocity / 3.0
```

Doing this because playing all notes at full volume would end up being much louder than sound of a single note. Set its value to `velocity` parameter divided by 3.0 because going to end up playing 3 notes instead of 1, so want each individual note to make up about $\frac{1}{3}$ of overall volume.

On line 3, set up our `for` loop to play 3 notes. Can also experiment with loops that repeat different numbers of times. Remember `range` is just a Python function that generates a sequence of numbers $[0, 1, 2]$ that variable i will walk through, 1 number at a time.

```
for i in range(0, 3):
```

On line 4, use Python's `random` function to generate a random decimal number somewhere between 0 & 1. Subtracting 0.5 will then gives us a number in range of negative -0.5 to 0.5. Save result in variable called `offset` that will use on next line of code to shift our note's pitch.

```
offset = random() - 0.5
```

Line 5 then plays note with our random pitch offset added in ($\in (-0.5, 0.5)$) to make it sound out of tune. Notice: use our `beats, volume` variables to control duration & volume of note that gets played, just as we might if were calling `playNote`.

```
playNote(note + offset, beats = beats, velocity = volume)
```

Last 2 lines of function make it so that all of notes get played at same time. On line 6, going to use a TunePad command called `rewind` to move playhead back to where we were before we played note. Remember: `playNote` automatically advances playhead forward by given number of beats, so need to rewind to get us back where we started. Effect of calling rewind is instantaneous; all it does is reposition playhead.

```
rewind(beats)
```

Calling `rewind` is important because want all 3 notes to play at exactly same time to give us right effect. Later in chap, experiment with playing notes that don't all get triggered at same time.

Lines 4–6 are all part of `for` loop – they all get repeated 3 times in a row because they're indented below loop statement on line 3. Finish function outside of loop with line 7 that calls another TunePad command called `fastForward`.

```
fastForward(beats)
```

As might guess, this does opposite of `reward` – instead of moving playhead backward it moves it forward. Call this last `fastForward` to make `eerieNote` behave exactly as if we had played a single note using our standard `playNote` function. Playhead will be moved by value of `beats`. This combination of using `rewind` inside of a loop with a `fastForward` at end of loop will be standard template for all of remaining effects that will cover in chap. Mix things up, but basic ideas will be same.

To put this all together, a childhood favorite that merrily reminisces about black plague. Sticking with horror movie theme, going to make this extra menacing by setting tempo way down to 60 BPM & playing whole thing an octave lower than usual. Snippet listed below defines entire song (including pitches, durations, & lyrics) in a Python list starting on line 1. Each note is represented by a Python structure called a `tuple`. Tuples are written as values separated by commas – just like lists except that you enclose values inside of parentheses instead of square brackets. Can use tuples exactly as we would use a list except values inside a tuple can't be changed. Saying same thing in Python lingo, tuples are *immutable* objects. Lines 18–19 step through notes in song, tuple by tuple, calling `eerieNote` for each one. Saying `note[0]` gets pitch of note & `note[1]` gets duration. Could also include this line inside loop if wanted to print lyrics.

```
print(note[3])
```

```
song = [
    (48, 1, "Ring"), (48, 0.5, "a-"), (45, 1, "round"), (50, 0.5, "the"),
    (48, 1.5, "ro-"), (45, 1, "sie, "), (47, 0.5, "a"), (48, 1, "pock-"),
    (48, 0.5, "et"), (45, 1, "full"), (50, 0.5, "of"), (48, 1.5, "po-"),
    (45, 1.5, "sies!"), (48, 1.5, "Ash-"), (45, 1.5, "es!"), (48, 1.5, "Ash-"),
    (45, 1, "es!"), (45, 0.5, "We"), (48, 1.5, "all"), (48, 1.5, "fall"),
    (41, 2.5, "down.")
]

def eerieNote(note, beats=1, velocity=100):
    volume = velocity
    for i in range(0, 3):
        offset = (random()-0.5) # random detune amount
        playNote(note + offset, beats, velocity=volume)
        rewind(beats)
    fastForward(beats)

for note in song:
    eerieNote(note[0] -12, beats=note[1])
```

Note values & associated TunePad commands for entire song: [Table: Note: Beats: Lyrics: Code]

- 8.2. Phaser effect. If play song in prev example, might hear some interesting & unexpected effects, especially on lower notes that come about from playing several notes together with very similar pitches. It turns out: this approximates a common musical effect called a *phaser* or *phase shifters*. Real phaser effects are produced by playing multiple versions of same sound together at same time, but changing frequency profile of each individual sound to get gaps or dips in spectrum.

  A very simple variation of `eerieNote` function replaces use of `random` function & instead just increments pitch `offset` variable by a fixed amount. This change gives us a cool approximation of a phaser effect that's easy to manipulate by changing number of simultaneous notes that get played or by changing pitch offset. Try this effect with some of built-in TunePad instruments, or use Sampler instrument to record & playback your own voice with `phaserNote` function.

```
def phaserNote(note, beats = 1, velocity = 100):
    note_count = 5
    volume = velocity / note_count
    offset = 0.0
    for i in range(0, note_count):
        playNote(note + offset, beats = beats, velocity = volume)
        offset += 0.15
        rewind(beats)
    fastForward(beats)
```

  Can try this example online by going to https://tunepad.com/examples/phaser.

- 8.3. Echo effect. For next set of examples, going to move from pitch-based effects to time-based effects where spread multiple notes out over time. Simplest version of this is an *echo effect* that plays an initial sound at full volume followed by a rapid succession of softer notes that fall off into silence. This is an extremely common technique used in a wide variety of digitally produced music. Can also throw in pitch manipulations & reverb effects to add another layer of complexity, but start with foundational repeated sound. As before, going to define our own function that looks similar to basic `playNote` function. Can follow along in TunePad by visiting https://tunepad.com/examples/echo.

```
def echoNote(note, beats = 1, delay = 0.125):
    volume = 100 # start at full volume
    offset = 0 # keep track of delays
    while volume > 1: # loop until silence
        playNote(note, beats = beats, velocity = volume)
        remind(beats - delay)
        volume *= 0.5 # reduce volume by 50%
        offset += delay # keep track of delays
    rewind(offset) # rewind by accumlated delay amount
    fastForward(beats) # move playhead to end of note
```

Line 1 looks similar to prev 2 examples except that we have added another optional parameter called `delay` that specifies how spread out each echoed note is in time. By default have set this to 0.125 beats, but by making this an optional parameter, have given composer ability to change this delay time on fly if they want to.

Line 2 should also look familiar except this time we are going to start with 1st note at full volume & then rapidly decay volume for each successive note. Then on line 3, define a variable called `offset` that keeps track of total amount of delay accumulated so far. This is a bookkeeping variable that's important for us to leave playhead in correct location after our function completes. Use this on line 9.

On line 4, set up a new kind of Python loop called a `while` loop. Until now, have always used `for` loops to iterate through a list or repeat sth for a *fixed* number of times. With a `while` loop on other hand, will repeat sth over & over again *until* a certain condition is met – in this case loop will repeat until volume is ≤ 1.

```
while volume > 1:
```

Basic idea: repeat sound until it's too quiet to hear. Trick to making this work: decrease value of `volume` inside while loop. Otherwise, loop would keep repeating over & over again forever (an infinite loop). Do this on line 7 where multiply volume by 0.5 (50% of its prev value). Could try other values here or even make this a parameter of function instead of a hard-coded value. If want sth to echo out longer, could set this up to 0.6 or even 0.7. Again, be careful here because if this value is 1.0 or higher note will echo forever & TunePad will complain that have created an infinite loop. Higher values can also cause note to echo out too long & run into other notes, creating unwanted interference.

Line 6 is a little tricky. Instead of rewinding all way back to beginning of note as we did in prev example, going to rewind by a smaller amount so that successive notes are spread out in time.

```
rewind(beats - delay)
```

This total amount = `delay` parameter that gets passed into function. Diagram below shows what this looks like over 4 iterations of `while` loop. Can also see value of `volume` as it decreases on each successive pass.

Last 2 lines of function are responsible for fixing up playhead position so that it's exactly at end of 1st note we played, which is why we kept track of number of total accumulated delay.

- 8.4. Chorus effect. A closely related effect to echo: randomly scatter notes around a central point in time. Our strategy for this effect should look familiar by now. Only tricky part: have to generate a random time offset ± a 32nd note that we use to fast forward. Because this number can be positive, negative, or 0, playhead will move forward, backward, or stay in place when call `fastForward`. Another way to think of it: `fastForward` with a negative number is same thing as calling `rewind` with a positive number. This function takes an optional parameter called `count` that says how many scattered notes we should play.

```
def chorusNote(note, beats = 1, count = 4):
    volume = 40
    spread = 0.125
    for i in range(0, count):
        offset = (random()-0.5) * spread
        fastForward(offset)
        playNote(note, beats - offset, volume)
        rewind(beats)
    fastForward(beats)
```

Then, when rewind on line 8, going to back up to where we started. Together this has effect of scattering several notes randomly around time the note was to be played. Can also try making spread an additional optional parameter that you pass into function. A simple stomp/clap pattern that uses `chorusNote`:

```
stomp = [ 0, 1 ]
clap = 22
chorusNote(stomp)
chorusNote(stomp)
chorusNote(clap)
rest(1)
```

This is a great effect to combine with a reverb effect to make it sound like you have a crowd in a large room. Another common technique: detune pitch of each note slightly from 1 another, which was a common technique for chorus effects applied to electric guitars in 1980s. Can try this function by going to https://tunepad.com/examples/chorus.

- 8.5. Arpeggiation effect. (Hiệu ứng rải âm) Another extremely common effect called arpeggiation is used to transform single sustained notes into rhythmic patterns that rapidly climb up & down a chord structure. Saw this in Interlude 4 as a method of playing chords. Arpeggios are commonly used across a wide variety of musical genres from classical to hip hop to dance music, & most digital audio workstations (DAWs) provide built-in arpeggiators with a variety of options to change speed, direction, & note pattern. Following pattern we saw with prev functions, can build an arpeggiator with TunePad by creating an `ARPNote` function with parameters for note pattern & speed. This effect combines both pitch-based & timing-based variations of notes.

```
def ARPNote(note, beats = 1, pattern = [0,4,7], speed = 0.125):
    offset = 0
    while offset < beats:
        for step in pattern:
            playNote(note + step, speed)
            offset += speed
    rewind(offset)
    fastForward(beats)
```

Before put this together in a complete example, look at `pattern` parameter. this parameter is a list that describes a pitch offset from 1st note (given by `note` parameter). Function will step through this list, & each element will be added to `note` to calculate which note to play. `pattern` parameter can be as simple or complex as you want, but a good starting point are some of chords introduced in Chap. 3. Some common arpeggiation patterns:

```
Major Triad [ 0, 4, 7 ]
Minor Triad [ 0, 3, 7 ]
Major Triad Up/Down [ 0, 4, 7, 12, 7, 4 ]
Minor Triad Up/Down [ 0, 3, 7, 12, 7, 3 ]
```

This function is a great example of using *nested* loops in Python, which just means: 1 loop is embedded inside of another. Outside loop gets set up on line 3 & repeats until variable `offset` > duration of note. Inner loop is set up on line 4 & steps through each value in `pattern` parameter. What all this means: inner for loop will get run at least once & possibly many times – as long as it takes to completely fill duration of note provided by `beats` parameter. Line 5 plays notes in arpeggio by adding `step` to base note s.t. it moves up or down pattern, & set duration of note to speed. Then on line 6, increment `offset` bookkeeping variable by duration of an arpeggiation step. Last 2 lines of function clean up timing to make sure playhead advances by correct amount.

`speed` parameter is a matter of taste & musical convention, but common to use 16th notes (0.25 beats) or 32nd notes (0.125 beats) for this value. 1 thing you might notice: this function will repeat arpeggiation chord pattern for as long as it takes to fill out entire note duration, but it might also play out beyond end of note. This extra holdover could be undesirable depending on music we're trying to make. To fix this, can insert 1 additional line right after line 6 inside for loop.

```
if offset >= beats: break
```

This has effect of exiting out of loop as soon as hit desired length, possibly short circuiting ARP pattern. Before look at an example of `ARPNote` function in action, add 1 more feature. It sounds a little heavy to have every note of arpeggio hit with same velocity. To add a more interesting rhythm & texture, could instead emphasize 1st note of arpeggio & then drop volume down for remaining notes. A variation of `ARPNote` function that creates this effect with sth like technique we used in `echoNote` example.

```
def ARPNote(note, beats = 1, velocity = 100, pattern = [0,4,7], speed = 0.125):
    offset = 0
    while offset < beats:
        volume = velocity # reset volume every iteration
        for step in pattern:
            playNote(note + step, speed, volume)
            volume = velocity * 0.25 # reduce volume after 1st note
            offset += speed
            if offset >= beats: break
    rewind(offset)
    fastForward(beats)
```

What's happening: create a `volume` variable that gets reset every iteration of while loop to full value of `velocity` parameter. Then, after play 1st note of arpeggio inside for loop, reduce `volume` to a quarter of `velocity` for remaining notes in arpeggio. When arpeggio is finished, `volume` is restored to its original value for next arpeggio.

This project puts everything together with an example from The Police. Try this code at https://tunepad.com/examples/arp.

When try this online, might notice: have added 1 more trick to `ARPNote` function on line 5. This uses `sustain` parameter of `playNote` to have 1st note of arpeggio ring out for entire duration of set, but at a reduced volume. This gives arpeggio a nice resonant quality.

- 8.6. Swing effect. Last effect in this chap creates a swing beat from a standard straight rhythm. Essentially this means adding a bounce or swing to a rhythm's timing so that we move from mechanical precision to more of a human feel. Most DAWs provide a variety of options for swinging a beat. Typically this involves altering a beat at either 9th or 16th note level so that odd-numbered notes are stretched out slightly in time, while even-numbered notes are compressed by same amount. Most DAWs let you select ratio between even- & odd-numbered notes by either offering a selection of fixed choices or a continuous dial. Creating sth similar in TunePad turns out to be fairly straightforward. Hardest part is figuring out whether we're on an even or an odd beat. To do this, can use built-in TunePad function called `getPlayHead()` that returns current position of playhead in beats (quarter notes). To figure out which 8th note we're on, can divide by an 8th note duration (0.5 on line 3). use Python's `round` function to make sure this is an integer value. Next step: ask whether we're on an even or an odd 8th note. Do that in line 4 using modulus operator `%`. Can think of this as returning remainder of a division operation. I.e., divide step variable by 2. If remainder is 1, it's an even 8th note, & need to push note forward a little by swing factor that we defined on line 2. Otherwise, it's an odd 8th note, & leave it in place. Together this has same effect as stretching out duration of odd-numbered notes.

  ```
  def swingNote(note, beats=1, velocity=100):
      swing = 0.25
      step = round(getPlayhead() / 0.5)
      if step % 2 == 1:
          fastForward(beats * swing)
          playNote(note, beats, velocity)
          rewind(beats * swing)
      else:
          playNote(note, beats, velocity)
  ```

  This example codes a simple rock beat that uses `swingNote`. Try this online at https://tunepad.com/examples/swing. Try adjusting value of swing factor on line 2 to get a feel for how it adds bounce to rhythm. Set swing down to 0.0 to get a straight beat.

- Interlude 8: How to Make a Drum Fill. In this interlude, explore 4 kinds of drum fills. A *drum fill* is a short phrase dropped into main groove of a drum track, usually every 9 or 16 bars. Fills add variety & support transition between sects of a song (e.g., verse to chorus). Can follow along online with this TunePad project https://tunepad.com/interlude/drum-fills.

  - **Step 1: Groove.** (Rãnh) Start by creating a 4-beat *groove*. For this tutorial, use a sparse rock-style drum pattern. Keep it simple because going to add decoration with various fills. Code below wraps drum pattern inside a function def so that it's easy to reuse in examples below. Define *groove* function inside of a *Code* cell in TunePad so that can import it into other cells.

    ```
    def groove():
        playNote(0, beats = 0.5)
        playNote(4, beats = 0.5)
        playNote(2, beats = 0.5)
        playNote(4, beats = 0.5)
        playNote(0, beats = 0.5)
        playNote([0, 4], beats = 0.5) # double kick with hat
        playNote(2, beats = 0.5)
        playNote(4, beats = 0.5)
    ```

    Set title of your code cell to "groove" & then note Python import statement that it generates for us directly below title Fig. 8.2: Code cell in TunePad showing import statement.

- **Pattern A: Tom runs.** Start with easiest drum fill. This fill takes up 1 measure (4 beats) & consists of 16th note runs on high tom, mid tom, low tom, & kick drum in order Fig. 8.3: Drum fill pattern A.

  Can code this with simple for loops to play each of drum sounds:

  ```
  from groove import * # import our groove function
  def fillA():
      for i in range(4):
          playNote(6, beats = 0.25) # high tom
      for i in range(4):
          playNote(7, beats = 0.25) # mid tom
      for i in range(4):
          playNote(8, beats = 0.25) # low tom
      for i in range(4):
          playNote(0, beats = 0.25) # kick drum
      playNote(9, beats = 0, sustain = 4) # crash cymbal
  ```

  Fill finishes with a crash cymbal hit on line 16. Use `sustain` (duy trì) parameter to let crash cymbal ring out & overlap with next drum measure. Code to play fill:

  ```
  # let's try it!
  groove()
  groove()
  fillA()
  groove()
  ```

- **Pattern B: Triplets.** Our next pattern plays triplet notes that subivide each beat into 3 equal parts. Pattern repeats 4times in a row with HIGH TOM - LOW TOM - KICK DRUM combos Fig. 8.4: Drum fill pattern B.

  Can code this with just 1 for loop where set beat duration to 1/3.0.

  ```
  from groove import *
  def fillB():
      for i in range(4):
          playNote(6, beats = 1/3.0)
          playNote(8, beats = 1/3.0)
          playNote(0, beats = 1/3.0)
      playNote(9, beats = 0, sustain = 4)
  ```

  Use this code to play `fillB` with groove.

  ```
  groove()
  groove()
  fillB()
  groove()
  ```

- **Pattern C: Random 16ths.** Next pattern sprinkles random 16th note hits in to decorate an otherwise boring pattern.

  – Mẫu tiếp theo rắc ngẫu nhiên nốt nhạc móc đơn 16 để trang trí cho một mẫu nhàm chán.

  To do this, going to use Python's `choice` function which is part of built-in `random` module. This function works by picking an element from a list at random. Can think of it as drawing a random card from a deck. To use `choice`, have to add a line at top of our code to import function:

  ```
  from random import choice
  ```

  Next secret ingredient: define a list of possible drum sounds that will get selected at random. 1 trick: pad this list with a few `None` values, which will result in random empty spaces in our pattern. Can experiment with different drum sounds in this list or with using a larger or smaller number of `None`s. Might also notice: doubled up on number of 10 notes (claps) because like how they sound.

  ```
  notes = [ 0, 2, 3, 4, 6, 7, 8, 10, 10, 11, None, None, None, None ]
  ```

  Once have defined our note array, function is simple to write. Just select & play 8 notes from our list at random.

  ```
  from random import choice
  from groove import *
  notes = [ 0, 2, 3, 4, 6, 7, 8, 10, 10, 11, None, None, None, None ]
  ```

```
    def fillC():
        for i in range(8):
            playNote(choice(notes), 0.25)
        rewind(2)

    # play it with our groove
    for i in range(10):
        groove()
        fillC()
        groove()
    ```

    1 trick on line 9 where rewind playhead by 2 beats. Do this so that drum fill overlaps with beat instead of pausing it while fill plays.

  - **Pattern D: Random triplets.** Our last pattern is a combination of Pattern B & C. use triplets but with random notes instead of a fixed pattern. Also sue `choice` function again, but with a smaller set of notes:

    ```
    notes = [ 0, 6, 7, 8, 10 ]
    ```

    Then can play 6 triplets (2 beats) at random followed by a crash.

    ```
    def fillD():
        notes = [ 0, 6, 7, 8, 10 ]
        for i in range(6):
            playNote(choice(notes), 1/3.0)
        playNote(9, beats = 0, sustain = 4)
    ```

    Code to play this full with our groove.

    ```
    for i in range(10):
        groove()
        groove()
        fillD()
        groove()
    ```

- 9. Song composition & EarSketch. [Guest chap by Lauren McCall] Art of sampling & remixing has become a central practice in modern digital music production. Artists from Dr. Dre to Cardi B have used new technologies to blend short samples from existing music into entirely new compositions. This chap introduces sampling, remixing, & song composition using a free online platform called EarSketch. This platform is similar to TunePad in that it combines Python programming with music creation. But with EarSketch focus is more on creating full-length songs by combining & remixing pre-recorded samples instead of composing music from individual notes & percussion sounds. Cover basics of working with EarSketch & show how to structure musical samples into full-length compositions.

  - 9.1. Song structure. There's more to writing music than coming up with a beat, harmony, & melody. Nearly every popular song follows some kind of codified structure. In days before recorded audio, structure & repetition helped listeners develop relationships with music. Material would get introduced early in a song & then elaborated throughout. Structure of a Blues song is an excellent sample, but this is true across almost every genre of popular music from punk to hip-hop to country. Structure & repetition is equally important to music we listen to today in that it gives us an opportunity to notice contrasting elements, remember a melody, or sing along with chorus.

    Song structure – sometimes known as *musical form* – describes how musical ideas & material play out over a piece of music. Earlier in book introduced idea of musical phrase: a self-contained musical thought, like a sentence. Can group these phrases into larger musical sects. These sects can repeat to form even larger structures Fig. 9.1: Songs are composed of nested & repeating notes, phrases, & sects. Notes ⊂ Phrase ⊂ Section ⊂ Song.

    Most common way for musicians to clarify & discuss different parts of a song: label them with letters. Every time there's a new sect, it gets next letter in alphabet. If a sect reappears, reuse letter we had already applied to that sect. If a sect reappears but is slightly different – maybe with different lyrics – can give it a number. This works as a timeline or map of a song. In popular music, *verse-chorus form* is most common song structure. At its simplest, this form is built on 2 repeating sect called `verse, chorus`. There are endless variations on this basic structure, but form tends to include: Intro → A1 Verse → B Chorus → A2 Verse → B Chorus → C bridge → B Chorus → Outro.

    Some songs will feature an introduction that builds up energy & introduces musical material. Verse is a repeated sect that helps tell story of a song. Generally, melody of verse is same each time, but lyrics are different. Chorus, or hook, usually contains an important musical or lyrical motif to song. This is usually most memorable part of a song, & lyrics are almost always same for each repetition. Bridge provides contrast with rest of song, often using sth special like a change in harmony or tempo. Not every song has a bridge; some have another repetition of verse. Final sect, outro, is a way to finish song. This might include sth like a slow fade out of chorus.

Take a song like Carrie Underwood's hit single, *Before He Cheats* (2005). This follows exactly structure in table above. There's a short intro followed by 1st verse ("Right now, he's probably slow dancin' ..."), followed by chorus ("I dug my key into side of his pretty little souped-up 4-wheel drive ..."). A 2nd verse & chorus is then followed by a bridge ("I might have saved a little trouble for the next girl"), 1 last chorus, & outro ("Oh, maybe next time he'll think before he cheats"). Like many of concepts discussed in this book, these rules are more of rough guidelines than hard rules. Being aware of song form can make writing music much easier, & can thin in larger sects. Using variations of common forms gives listeners an entry point as to what to expect from your music.

- 9.2. Sampling. Sampling involves taking sects from existing pieces of music & repurposing or remixing them in order to make a new creation. A sample might consist of a specific musical element of original song e.g. a bassline, drum break, vocals, or even a snippet of speech. Samples often have creative effects applied, e.g. reverb, chorus, or filters, & they might be looped, pitch shifted, or even reversed. Examples includes songs like Kayne West's *Good Morning* (2007), which sampled Elton John's *Someone Saved My Life Tonight* (1975) or *Naughty* By Nature's O.P.P. (1991), which sampled multiple elements from Jackson 5's *ABC* (1970). Sampling often occurs in dialogue with or homage to original work. An artist might include a reference to a work that inspired them by incorporating signature chord progressions, instrumentation, or melodies.

  Availability of new recording technology in 20th century gave artists opportunity to engage with prior generations of music in innovative ways. Although sampling is common to many genres of music, it's 1 of defining characteristics of hip-hop music. Roots of hip-hop were in live performance – DJs manipulated existing records on turntables to create completely new soundscapes. Techniques from live performance then carried over into original works driven by advances in recording technology & equipment specifically designed for sampling. Advanced sampling tools are now built into digital audio workstation (DAW) software in which vast majority of modern music is created.

  When combining samples in your composition, be mindful of how different musical elements work with 1 another. If your song is in E major & add a sample that's in D minor, it might clash harmonically. If sample is in a different tempo, it might not line up rhythmically with rest of your song. Samples get manipulated in pitch or tempo or a variety of other ways to make them work stylistically with rest of a song.

  Remark 2 (Copyright law). *Be aware of copyright issues around music that you sample, especially if you're thinking about sharing it online or licensing it to make money. Most of music on TunePad & EarSketch websites is licensed so that you can use them however you want, but important to get permission or a license when sampling other artists' music.*

- 9.3. Introduction to EarSketch. EarSketch Fig. 9.2: Main EarSketch interface features a large library of samples (left), an interactive timeline (middle top), a code editor (center), & extensive documentation & curriculum (right). is a free online platform for creating music with code that was developed by researchers at Georgia Institute of Technology in Atlanta. EarSketch works with a few different programming languages including Python & JavaScript. All of Python concepts you've been learning about (lists, loops, variables, functions, & parameters) still apply, although music-making functions are different. E.g., EarSketch doesn't have a `playNote` or `rest` function. Instead, it has a `fitMedia` function that places musical samples on timeline of a DAW.

  Remark 3 (JavaScript). *JavaScript is another important & widely used programming language. It's often called* language of web *because web sites use JavaScript to add logic, user interaction, & dynamic effects. Every time click on a button on a web page, it's almost certainly running JavaScript code. If were to look under hood at TunePad & EarSketch, you would see: they both use JavaScript to generate & play music. If interested in trying it out, EarSketch has excellent resources for learning JavaScript.*

  Once log into main EarSketch site at https://earsketch.gatech.edu, can write programs in *Code Editor*. Note: both EarSketch & TunePad sometimes use term *script* to refer to a program. A script is just another word for a short computer program that performs smaller tasks like putting together a song. Creating a new script in EarSketch generates a small amount of *boilerplate* code to set up your project. In computer programming "boilerplate" means standard code that you use across multiple projects. EarSketch boilerplate:

  ```
  from earsketch import *
  init()
  setTempo(120)
  finish()
  ```

  1st line imports core EarSketch module. In TunePad have used Python's import functionality as well, but can read more about how it works in Python appendix at end of book. 2nd line `init()`, sets up DAW. Next, `setTempo` specifies project's tempo in beats per minute (bpm). Code you write for your project should be added in between `setTempo, finish()` function calls.

  1 of best parts about EarSketch is its extensive sound library. This library has nearly 4000 premade audio clips created by producers & musicians that you can use in your projects for free. Can browse through sample library on main EarSketch page, filtering by musical genre, artist, & instrument type. Can add samples to your project with code using their predefined variable names & `fitMedia` function.

  * 9.3.1. `fitMedia` function. p. 163+++

- 10. Modular synthesis. [Guest chap by DILLON HAL] Modular synthesis is a set of tools & techniques for electronically or digitally "synthesizing" musical sounds. *Modular* means "made up of smaller pieces that can be taken apart & rearranged". *Synthesis* means "creating sth new from existing parts or ideas". Modular synthesizers were 1st introduced in 1950s & 1960s with devices like Moog System 55 Fig. 10.1: Moog System 55 modular synthesizer., which were sometimes size of entire rooms. These synthesizers were *analog*, meaning they generated sounds using electrical circuits instead of starting with some kind of physical vibration (like a guitar string). By chaining together electronically generated waveforms with filters, effects, & envelopes, they could not only approximate traditional acoustic instruments – like bass, piano, brass, woodwinds, & drums – but also create entirely new sounds altogether. Today, can make these same sounds on a computer using *digital signal processing* techniques. While process has changed, basic ideas are same: take several different *modules* & combine them together into a *patch*. A module is a single device with a single purpose e.g. creating, transforming, or controlling sound. A patch is formed by chaining many modules together to combine sounds & layer effects. This chap provides a high-level overview of modular synthesis concepts using an interface built into TunePad.

  ○ 10.1. Signals. In modular synthesis, a *signal* is a collection of values that varies over time & can carry information. In physical world, when sound waves collide with membrane of a microphone, they cause fluctuations in voltage levels of an electric circuit. Microphone has transformed physical sound waves into an electric signal that consists of fluctuating voltage levels. These recorded voltage levels are just 1 example of a signal Fig. 10.2: Audio signal from a microphone & an audio signal from an electric circuit.

    – Tín hiệu. Trong tổng hợp mô-đun, *tín hiệu* là tập hợp các giá trị thay đổi theo thời gian & có thể mang thông tin. Trong thế giới vật lý, khi sóng âm va chạm với màng của micrô, chúng gây ra sự dao động ở mức điện áp của mạch điện. Micrô đã biến đổi sóng âm vật lý thành tín hiệu điện bao gồm các mức điện áp dao động. Các mức điện áp được ghi lại này chỉ là 1 ví dụ về tín hiệu Hình 10.2: Tín hiệu âm thanh từ micrô & tín hiệu âm thanh từ mạch điện.

    For analog synthesizers, this process works in reverse. Audio signals are generated synthetically using simple electronic components – e.g. resistors, capacitors, & transistors – rather than a microphone. By varying voltage input levels, can change frequency at which circuit oscillates or "vibrates". In digital world, instead of using oscillator circuits, computers generate streams of numbers that simulate voltage output of original electronic components.

    – Đối với bộ tổng hợp tương tự, quá trình này hoạt động ngược lại. Tín hiệu âm thanh được tạo ra tổng hợp bằng cách sử dụng các thành phần điện tử đơn giản – ví dụ như điện trở, tụ điện, & bóng bán dẫn – thay vì micrô. Bằng cách thay đổi mức điện áp đầu vào, có thể thay đổi tần số mà mạch dao động hoặc "rung". Trong thế giới kỹ thuật số, thay vì sử dụng mạch dao động, máy tính tạo ra các luồng số mô phỏng điện áp đầu ra của các thành phần điện tử gốc.

    For modular synthesis, there are 2 basic kinds of signals: *audio signals & control signals*. Audio signals oscillate in range of human hearing, i.e., they can create musical notes. Faster they move, higher their *frequency* & higher pitch we hear. Control signals, on other hand, tend to vibrate more slowly, below range of human hearing. Instead of sending them directly to our speakers, use them to change parameters of audio signals. A good example of this is *vibrato*. A violinist draws a bow across a string to generate a high-frequency sound that falls in range of human hearing. As bow is drawn, violinist quickly rocks a finger on fretboard to modulate pitch. Violinist's finger operates like a control signal that modifies audio signal of violin string.

    – Đối với tổng hợp mô-đun, có 2 loại tín hiệu cơ bản: *tín hiệu âm thanh & tín hiệu điều khiển*. Tín hiệu âm thanh dao động trong phạm vi thính giác của con người, tức là chúng có thể tạo ra các nốt nhạc. Chúng di chuyển càng nhanh, *tần số* của chúng & cao độ mà chúng ta nghe được càng cao. Mặt khác, tín hiệu điều khiển có xu hướng rung chậm hơn, thấp hơn phạm vi thính giác của con người. Thay vì gửi chúng trực tiếp đến loa của chúng ta, hãy sử dụng chúng để thay đổi các thông số của tín hiệu âm thanh. Một ví dụ điển hình về điều này là *rung*. Một nghệ sĩ vĩ cầm kéo một cây vĩ trên một dây đàn để tạo ra âm thanh tần số cao nằm trong phạm vi thính giác của con người. Khi kéo vĩ, nghệ sĩ vĩ cầm nhanh chóng lắc ngón tay trên cần đàn để điều chỉnh cao độ. Ngón tay của nghệ sĩ vĩ cầm hoạt động giống như một tín hiệu điều khiển để điều chỉnh tín hiệu âm thanh của dây đàn vĩ cầm.

  ○ 10.2. Modules. 1 way to think about modules is as physical pieces of equipment with 1 or more sockets for inputs & an output signal. Inputs may be audio signals or control signals. Outputs may feed into other modules or be sent to our speakers. In simple TunePad patch below, sine wave module generates an audio signal that gets plugged into Output module & sent to speakers Fig. 10.3: A simple Modular Synthesis patch created in TunePad.

    ∗ 10.2.1. Source modules. A source module is a type of module that generates a signal. Sources can also have inputs to control parameters like pitch & amplitude.
      – Mô-đun nguồn. Mô-đun nguồn là loại mô-đun tạo ra tín hiệu. Các nguồn cũng có thể có đầu vào để điều khiển các thông số như cao độ & biên độ.
      p. 180+++

    ∗ 10.2.2. Control modules. Control modules generate signals that aren't typically audible to humans. Control modules can feed into input sockets of other modules.

    ∗ 10.2.3. Creating patches.

- 11. History of music & computing. [Guest chap by JASON FREEMAN] Throughout this book, you've seen that music & computing are connected. Musicians think about things like notes, meter, & phrases just like programmers think about things like variables, functions, & loops. Code can help us understand inner logic of how music is created. Writing music with code can help us create interesting music & express ourselves in new ways.

These connections between music & computing are no accident. Since earliest computers, musicians have been inventing ways to use computers in music. Today, when listen to your favorite song, computers have likely been involved in many different ways. E.g.:

- Songwriter may have used music notation or audio recording software to capture their initial inspiration for song on a laptop, tablet, or mobile phone.
- Musicians in band may have performed on digital keyboards, drum machines, or control surfaces. Guitarists & bass players may have routed their audio through digital effects pedals. Each of these devices makes or transforms sound through an embedded computer.
- In recording studio, a producer & audio engineer probably recorded, edited, & mixed song using digital audio workstation software on a computer. They may have even used additional software plugins for special tasks like eliminating background noise or fixing notes that were out of tune.
- A music streaming service had to store music track & information about it in a cloud-based database. Service also developed an app to stream track to your device. In addition, service probably developed ML algorithms to generate personalized playlists or radio stations for you.

In this chap, explore some of key moments in history of computer music. This history will help us understand how computing is continuing to revolutionize creation, performance, & distribution of music today.

- 11.1. What is computer music? Computer music refers to any process of creating, performing, analyzing, or distributing music that involves a computer. This includes a wide range of activities, from writing code to creating music (like in EarSketch & TunePad) to producing new music with an app to watching your favorite music videos online.

  Because computer music is such a broad field, it helps to ask a few basic questions as we look at developments in computer music history or think about things we want to do with computers ourselves:

  What is role of computer? Often think of computers as tools that help us get a job done, like writing a paper or reading a message. Computers can be musical tools too. They can be musical instruments that create sound in response to our actions, like a digital keyboard that plays a note every time we press a key. Most of instruments in TunePad, e.g., work in this way. Press keys on computer keyboard & immediately hear corresponding instrument sound from TunePad.

  Computers can also act more like a musician who performs a piece, taking each note in a score & turning it into sound. Much of code we write in TunePad works this way. Specify a series of notes to be played. Then TunePad displays notes in a grid & plays them back for us. Each instrument in a TunePad dropbook acts like a single musician. When play them all back together, computer becomes like a band of musicians.

  Many computer musicians also explore how computers can act like intelligent musicians. These intelligent musicians don't simply follow instructions in a score. An artificially intelligent computer composer may create its own musical scores that mimic style of a human musician. An artificially intelligent computer performer may listen to human musicians & improvise along with them.

  How does computer represent music? Human musicians have many different ways of representing music: create notated scores, create shorthand lead sheets, talk about it in words, & teach it by oral tradition. Computer musicians also have many different ways of representing music. 2 approaches are most common. Symbolic representations describe music as a series of musical notes, each with a specific pitch, loudness, start time, & duration. Audio representations describe music in terms of actual sound that we hear.

  Have already seen these 2 approaches with TunePad & EarSketch. TunePad's `playNote()` function, e.g., is a symbolic approach that defines properties of a single musical note. Music in TunePad consists of a bunch of these notes defined by `playNote()` & other TunePad functions.

  In contrast, EarSketch's `fitMedia()` function adds an audio file into your song. That audio file can be anything: it can be a single musical note, it can be an entire musical phrase, or it can be a sound effect that might not even seem like music. A song in EarSketch is made up of a bunch of these audio files placed on a multi-track timeline.

  There are advantages & disadvantages to each approach. With a symbolic representation, computer can easily represent & manipulate each individual note. TunePad can easily draw music on a grid that shows each note, & if want to change a single note in your song, easy to find corresponding line of code & update it. With an audio representation, those kinds of visualizations & edits are often not possible. But there is more flexibility to capture, generate, & edit any kind of sound.

  Does computer create music in real time? Early computers often took hours to generate a few secs of sound, & even today, computer music applications that use ML & AI can take days to analyze large datasets. Many computer music applications, like musical instruments, require immediate, real-time responses, while with other applications, like composition, real-time operation may not always be as important.

  Consider these 3 questions as look at some of earliest examples of computer music.

- 11.2. Computer music on mainframe computers. In 1950s, Bell Labs in New Jersey was 1 of most famous research & development labs in world. Building from legacy of Alexander Graham Bell's invention of telephone, this division of AT&T had created innovative new technologies that sill impact how transmit radio signals, synthesize speech, encrypt communications, & build computers.

  Max Matthews was an electrical engineer & violinist working at Bell Labs. He was working with giant mainframe computers like IBM 7094, which cost millions of dollars, occupied an entire room, & relied on punch cards & magnetic tapes to read &

write data. These computers were used for many purposes, including during some of early NASA space missions Fig. 11.1: IBM 7094 computer at NASA. Public domain. Available at Wikipedia.

MATTHEWS wanted to make music with these mainframe computers. He created a programming language called MUSIC, then later MUSIC II, MUSIC III, & so on. Today call his languages MUSIC-N (N is like a variable representing specific version number of MUSIC).

In MUSIC-N languages, programmers created an "orchestra" of musical instruments. Orchestra defined a set of instruments & configured how each instrument created sound. An example of an instrument definition in CSound, a music programming language inspired by MUSIC-N that can still use today: [Codes]

Even though syntax of programming language may look strange, elements are actually familiar. 1st line defines instrument, which is similar to defining your own function in TunePad or EarSketch. 3rd line outputs sound, which is similar to a return statement in your EarSketch or TunePad function def. 2nd line calls a unit generator (oscil), which is similar to calling a function, & passes 3 arguments to it: `p4, p5, 1`. Oscil is an oscillator that synthesizes a simple waveform, like a sine wave or square wave. Arguments configure things like how low or high sound is (frequency) & how loud it is (amplitude).

– Mặc dù cú pháp của ngôn ngữ lập trình có vẻ lạ, nhưng các phần tử thực sự quen thuộc. Dòng thứ nhất định nghĩa nhạc cụ, tương tự như định nghĩa hàm của riêng bạn trong TunePad hoặc EarSketch. Dòng thứ ba xuất ra âm thanh, tương tự như câu lệnh return trong hàm EarSketch hoặc TunePad của bạn def. Dòng thứ hai gọi một trình tạo đơn vị (oscil), tương tự như gọi một hàm, & truyền 3 đối số cho nó: `p4, p5, 1`. Oscil là một bộ dao động tổng hợp một dạng sóng đơn giản, như sóng sin hoặc sóng vuông. Các đối số cấu hình những thứ như âm thanh thấp hay cao (tần số) & âm lượng lớn như thế nào (biên độ).

p. 197+++

  ◦ 11.3. Digital synthesizers & personal computers.

- Appendix A: Python ref.
- Appendix B: TunePad programming ref.
- Appendix C: Music ref.

1.3 [Väl+06]. VESA VÄLIMÄKI, JYRI PAKARINEN, CUMHUR ERKUT, MATTI KARJALAINEN. Discrete-Time Modeling of Musical Instruments

[283 citations]

Question 1 (Cf. Continuous-time modeling vs. Discrete-time modeling)**.** *How about continuous-time modeling of musical instruments?*

Question 2 (Cf. Mathematical modeling technique vs. Physical modeling technique)**.** *Compare Mathematical modeling technique vs. Physical modeling technique.*

- **Abstract.** This article describes physical modeling techniques that can be used for simulating musical instruments. Methods are closely related to digital signal processing. They discretize system w.r.t. time, because aim: run simulation using a computer. Physics-based modeling methods can be classified as mass-spring, modal, wave digital, finite difference, digital waveguide & source-filter models. Present basic theory & a discussion on possible extensions for each modeling technique. For some methods, a simple model example is chosen from existing literature demonstrating a typical use of method. E.g., in case of digital waveguide modeling technique a vibrating string model is discussed, & in case of wave digital filter technique, present a classical piano hammer model. Tackle some nonlinear & time-varying models & include new results on digital waveguide modeling of a nonlinear string. Discuss current trends & future directions in physical modeling of musical instruments.

- 1. Introduction. Musical instruments have historically been among most complicated mechanical systems made by humans. They have been a topic of interest for physicists & acousticians for over a century. Modeling of musical instruments using computers is newest approach to understanding how these instruments work.

  This paper presents an overview of physics-based modeling of musical instruments. Specifically, this paper focuses on sound synthesis methods derived using physical modeling approach. Several previously published tutorial & review papers discussed physical modeling synthesis techniques for musical instruments sounds [73, 129, 251, 255, 256, 274, 284, 294]. Purpose of this paper: give a unified introduction to 6 main classes of discrete-time physical modeling methods, namely mass–spring, modal, wave digital, finite difference, digital waveguide & source–filter models. This review also tackles mixed & hybrid models in which usually 2 different modeling techniques are combined.

  Physical models of musical instruments have been developed for 2 main purposes: research of acoustical properties & sound synthesis. Methods discussed in this paper can be applied to both purpose, but here main focus is sound synthesis. Basic idea of physics-based sound synthesis: build a simulation model of sound production mechanism of a musical instrument & to generate sound with a computer program or signal processing hardware that implements that model. Motto of physical modeling synthesis: when a model has been designed properly, so that it behaves much like actual acoustic instrument, synthetic sound will automatically be natural in response to performance. In practice, various simplifications of model cause sound output to be similar to, but still clearly different from, original sound. Simplifications may be caused by intentional approximations that reduce computational cost or by inadequate knowledge of what is actually happening in acoustic instrument. A typical & desirable simplification: linearization of slightly nonlinear phenomena, which may avert unnecessary complexities, & hence may improve computational efficiency.

In speech technology, idea of accounting for physics of sound source, human voice production organs, is an old tradition, which has led to useful results in speech coding & synthesis. While 1st experiments on physics-based musical sound synthesis were documented several decades ago, 1st commercial products based on physical modeling synthesis were introduced in 1990s. Thus, topic is still relatively young. Research in field has been very active in recent years.

1 of motivations for developing a physically based sound synthesis: musicians, composers, & other users of electronic musical instruments have a constant hunger for better digital instruments & for new tools for organizing sonic events. A major problem in digital musical instruments has always been how to control them. For some time, researchers of physical models have hoped: these models would offer more intuitive, & in some ways better, controllability than previous sound synthesis methods. In addition to its practical applications, physical modeling of musical instruments is an interesting research topic for other reasons. It helps to solve old open questions, e.g. which specific features in a musical instrument's sound make it recognizable to human listeners or why some musical instruments sound sophisticated while others sound cheap. Yet another fascinating aspect of this field: when physical principles are converted into computational methods, possible to discover new algorithms. This way, possible to learn new signal processing methods from nature.

- 2. Brief history. Modeling of musical instruments is fundamentally based on understanding of their sound production principles. 1st person attempting to understand how musical instruments work might have been Pythagoras, who lived in ancient Greece around 500 BC. At that time, understanding of musical acoustics was very limited & investigations focused on tuning of string instruments. Only after late 18th century, when rigorous mathematical methods e.g. PDEs were developed, was it possible to build formal models of vibrating strings & plates.

  Earliest work on physics-based discrete-time sound synthesis was probably conducted by Kelly & Lochbaum in context of vocal-tract modeling [145]. A famous early musical example is 'Bicycle Built for 2' (1961), where singing voice was produced using a discrete-time model of human vocal tract. This was result of collaboration between Mathews, Kelly & Lochbaum [43]. 1st vibrating string simulations were conducted in early 1970s by Hiller & Ruiz [113, 114], who discretized wave equation to calculate waveform of a single point of a vibrating string. Computing 1 s of sampled waveform took minutes. A few years later, Cadoz & his colleagues developed discrete-time mass-spring models & built dedicated computing hardware to run real-time simulations [38].

  In late 1970s & early 1980s, McIntyre, Woodhouse, & Schumacher made important contributions by introducing simplified discrete-time models of bowed strings, clarinet & flute [173, 174, 235], & Karplus & Strong [144] invented a simple algorithm that produces string-instrument-like sounds with few arithmetic operations. Based on these ideas & their generalizations, Smith & Jaffe introduced a signal-processing oriented simulation technique for vibrating strings [120, 244]. Soon thereafter, Smith proposed term 'digital waveguide' & developed general theory [247, 249, 253].

  1st commercial product based on physical modeling synthesis, an electronic keyboard instrument by Yamaha, was introduced in 1994 [168]; it used digital waveguide techniques. More recently, digital waveguide techniques have been also employed in MIDI synthesizers on personal computer soundcards. Currently, much of practical sound synthesis is based on software, & there are many commercial & freely available pieces of synthesis software that apply 1 or more physical modeling methods.

- 3. General concepts of physics-based modeling. In this sect, discuss a number of physical & signal processing concepts & terminology that are important in understanding modeling paradigms discussed in subsequent sects. Each paradigm is also characterized briefly in end of this sect. A reader familiar with basic concepts in context of physical modeling & sound synthesis may go directly to Sect. 4.

  - 3.1. Physical domains, variables, & parameters. Physical phenomena can be categorized as belonging to different 'physical domains'. Most important ones for sound sources e.g. musical instruments are acoustical & mechanical domains. In addition, electrical domain is needed for electroacoustic instruments & as a domain to which phenomena from other domains are often mapped. Domains may interact with one another, or they can be used as analogies (equivalent models) of each other. Electrical circuits & networks are often applied as analogies to describe phenomena of other physical domains.

    Quantitative description of a physical system is obtained through measurable quantities that typically come in pairs of variables, e.g. force & velocity in mechanical domain, pressure & volume velocity in acoustical domain or voltage & current in electrical domain. Members of such dual variable pairs are categorized generically as 'across variable' or 'potential variable', e.g. voltage, force or pressure, & 'through variable' or 'kinetic variable', e.g. current, velocity or volume velocity. If there is a linear relationship between dual variables, this relation can be expressed as a parameter, e.g. impedance $Z = \frac{U}{I}$ being ratio of voltage U & current I, or by its inverse, admittance $Y = \frac{I}{U}$. An example from mechanical domain is mobility (mechanical admittance) defined as ratio of velocity & force. When using such parameters, only 1 of dual variables is needed explicitly, because the other one is achieved through constraint rule.

    Modeling methods discussed in this paper use 2 types of variables for computation, 'K-variables' & 'wave variables' (also denoted as 'W-variables'). 'K' comes from Kirchhoff & refers to Kirchhoff continuity rules of quantities in electric circuits & networks [185]. 'W' is shortform for wave, referring to wave components of physical variables. Instead of pairs of across & through as with K-variables, wave variables come in pairs of incident & reflected wave components. Details of wave modeling are discussed in Sects. 7–8, while K-modeling is discussed particularly in Sects. 4 & 10. It will become obvious: these are different formulations of same phenomenon, & possibility to combine both approaches in hybrid modeling will be discussed in Sect. 10.

Decomposition into wave components is prominent in such wave propagation phenomena where opposite-traveling waves add up to actual observable K-quantities. A wave quantity is directly observable only when there is no other counterpart. It is, however, a highly useful abstraction to apply wave components to any physical case, since this helps in solving computability (causality) problems in discrete-time modeling.

- 3.2. Modeling of physical structure & interaction. Physical phenomena are observed as structures & processes in space & time. In sound source modeling, interested in dynamic behavior that is modeled by variables, while slowly varying or constant properties are parameters. Physical interaction between entities in space always propagates with a finite velocity, which may differ by orders of magnitude in different physical domains, speed of light being upper limit.

  'Causality' is a fundamental physical property that follows from finite velocity of interaction from a cause to corresponding effect. In many mathematical relations used in physical models causality is not directly observable. E.g., relation of voltage across & current through an impedance is only a constraint, & variables can be solved only within context of whole circuit. Requirement of causality (more precisely temporal order of cause preceding effect) introduces special computability problems in discrete-time simulation, because 2-way interaction with a delay shorter than a unit delay (sampling period) leads to 'delay-free loop problem'. Use of wave variables is advantageous, since incident & reflected waves have a causal relationship. In particular, wave digital filter (WDF) theory, discussed in Sect. 8, carefully treats this problem through use of wave variables & specific scheduling of computation operations.

  Taking finite propagation speed into account requires using a spatially distributed model. Depending on case at hand, this can be a full 3D model e.g. used for room acoustics, a 2D model e.g. for a drum membrane (discarding air loading) or a 1D model e.g. for a vibrating sting. If object to be modeled behaves homogeneously enough as a whole, e.g. due to its small size compared with wavelength of wave propagation, it can be considered a lumped entity that does not need a description of spatial dimensions.

  – Việc tính đến tốc độ lan truyền hữu hạn đòi hỏi phải sử dụng 1 mô hình phân bố không gian. Tùy thuộc vào trường hợp cụ thể, đây có thể là mô hình 3D đầy đủ, ví dụ như được sử dụng cho âm học phòng, mô hình 2D, ví dụ như cho màng trống (loại bỏ tải trọng không khí) hoặc mô hình 1D, ví dụ như cho 1 cú chích rung. Nếu vật thể được mô hình hóa hoạt động đủ đồng nhất như 1 tổng thể, ví dụ như do kích thước nhỏ so với bước sóng truyền sóng, thì nó có thể được coi là 1 thực thể tập trung không cần mô tả về kích thước không gian.

- 3.3. Signals, signal processing, & discrete-time modeling. In signal processing, signal relationships are typically represented as 1-directional cause-effect chains. Contrary to this, bi-directional interaction is common in (passive) physical systems, e.g. in systems where reciprocity principle is valid. In true physics-based modeling, 2-way interaction must be taken into account. I.e., from signal processing viewpoint, such models are full of feedback loops, which further implicates: concepts of computability (causality) & stability become crucial.

  In this paper, apply digital signal processing (DSP) approach to physics-based modeling whenever possible. Motivation for this: DSP is an advanced theory & tool that emphasizes computational issues, particularly maximal efficiency. This efficiency is crucial for real-time simulation & sound synthesis. Signal flow diagrams are also a good graphical means to illustrate algorithms underlying simulations. Assume: reader is fmailiar with fundamentals of DSP, e.g. sampling theorem [242] to avoid aliasing (also spatial aliasing) due to sampling in time & space as well as quantization effects due to finite numerical precision.

  An important class of systems is those that are linear & time invariant (LTI). They can be modeled & simulated efficiently by digital filters. They can be analyzed & processed in frequency domain through linear transforms, particularly by Z-transform & discrete Fourier transform (DFT) in discrete-time case. While DFT processing through fast Fourier transform (FFT) is a powerful tool, it introduces a block delay & does not easily fit to sample-by-sample simulation, particularly when bi-directional physical interaction is modeled.

  Nonlinear & time-varying systems bring several complications to modeling. Nonlinearities create new signal frequencies that easily spread beyond Nyquist limit, thus causing aliasing, which is perceived as very disturbing distortion. In addition to aliasing, delay-free loop problem & stability problems can become worse than they are in linear systems. If nonlinearities in a system to be modeled are spatially distributed, modeling task is even more difficult than with a localized nonlinearity. Nonlinearities will be discussed in several sects of this paper, most completely in Sect. 11.

- 3.4. Energetic behavior & stability. Product of dual variables e.g. voltage & current gives power, which, when integrated in time, yields energy. Conservation of energy in a closed system is a fundamental law of physics that should also be obeyed in true physics-based modeling. In musical instruments, resonators are typically passive, i.e. they do not produce energy, while excitation (plucking, bowing, blowing, etc.) is an active process that injects energy to passive resonators.

  Stability of a physical system is closely related to its energetic behavior. Stability can be defined so that energy of system remains finite for finite energy excitations. From a signal processing viewpoint, stability may also be defined so that variables, e.g. voltages, remain within a linear operating range for possible inputs in order to avoid signal clipping & distortion.

  In signal processing systems with 1-directional input–output connections between stable subblocks, an instability can appear only if there are feedback loops. In general, impossible to analyze such a system's stability without knowing its whole feedback structure. Contrary to this, in models with physical 2-way interaction, if each element is passive, then any arbitrary network of such elements remains stable.

- 3.5. Modularity & locality of computation. For a computational realization, desirable to decompose a model systematically into blocks & their interconnections. Such an object-based approach helps manage complex models through use of modularity principle. Abstractions to macro blocks on basis of more elementary ones helps hiding details when building excessively complex models.

For 1-directional interactions used in signal processing, enough to provide input & output terminals for connecting blocks. For physical interaction, connections need to be done through ports, with each port having a pair of K- or wave variables depending on modeling method used. This allows mathematical principles used for electrical networks [185]. Details on block-wise construction of models will be discussed in following sects for each modeling paradigm.

Locality of interaction is a desirable modeling feature, which is also related to concept of causality. For a physical system with a finite propagation speed of waves, enough: a block interacts only with its nearest neighbors; it does not need global connections to compute its task & effect automatically propagates throughout system.

In a discrete-time simulation with bi-directional interactions, delays shorter than a unit delay (including 0 delay) introduce delay-free loop problem that we face several times in this paper. While possible to realize fractional delays [154], delayers shorter than unit delay contain a delay-free component. There are ways to make such 'implicit' system computable, but cost in time (or accuracy) may become prohibitive for real-time processing.

- 3.6. Physics-based discrete-time modeling paradigms. This paper presents an overview of physics-based methods & techniques for modeling & synthesizing musical instruments. Have excluded some methods often used in acoustics, because they do not easily solve task of efficient discrete-time modeling & synthesis. E.g., finite element & boundary element methods (FEM & BEM) are generic & powerful for solving system behavior numerically, particularly for linear systems, but focus on inherently time-domain methods for sample-by-sample computation.

  Main paradigms in discrete-time modeling of musical instruments can be briefly characterized as follows.

  - 3.6.1. Finite difference models. In Sect. 4, finite difference models are numerical replacement for solving PDEs. Differentials are approximated by finite differences so that time & position will be discretized. Through proper selection of discretization to regular meshes, computational algorithms become simple & relatively efficient. Finite difference time domain (FDTD) schemes are K-modeling methods, since wave components are not explicitly utilized in computation. FDTD schemes have been applied successfully to 1D, 2D, & 3D systems, although in linear 1D cases digital waveguides are typically superior in computational efficiency & robustness. In multidimensional mesh structures, FDTD approach is more efficient. It also shows potential to deal systematically with nonlinearities (Sect. 11). FDTD algorithms can be problematic due to lack of numerical robustness & stability, unless carefully designed.
  - 3.6.2. Mass–spring networks. In Sect. 5, mass–spring networks are a modeling approach, where intuitive basic elements in mechanics – masses, springs, & damping elements – are used to construct vibrating structures. It is inherently a K-modeling methodology, which has been used to construct small- & large-scale mesh-like & other structures. It has resemblance to FDTD schemes in mesh structures & to WDFs for lumped element modeling. Mass–spring networks can be realized systematically also by WDFs using wave variables (Sect. 8).
  - 3.6.3. Modal decomposition methods. In Sect. 6 modal decomposition methods represent another approach to look at vibrating systems, conceptually from a frequency-domain viewpoint. Eigenmodes of a linear system are exponentially decaying sinusoids at eigenfrequencies in response of a system to impulse excitation. Although thinking by modes is normally related to frequency domain, time-domain simulation by modal methods can be relatively efficient, & therefore suitable to discrete-time computation. Modal decomposition methods are inherently based on use of K-variables. Modal synthesis has been applied to make convincing sound synthesis of different musical instruments. Functional transform method (FTM) is a recent development of systematically exploiting idea of spatially distributed modal behavior, & it has also been extended to nonlinear system modeling.
  - 3.6.4. Digital waveguides. Digital waveguides (DWGs) in Sect. 7 are most popular physics-based method of modeling & synthesizing musical instruments that are based on 1D resonators, e.g. strings & wind instruments. Reason for this is their extreme computational efficiency in their basic formulations. DWGs have been used also in 2D & 3D mesh structures, but in such cases wave-based DWGs are not superior in efficiency. Digital waveguides are based on use of traveling wave components; thus, they form a wave modeling (W-modeling) paradigm [Term digital waveguide is used also to denote K-modeling, e.g. FDTD mesh-structures, & source-filter models derived from traveling wave solutions, which may cause methodological confusion.]. Therefore, they are also compatible with WDFs (Sect. 8), but in order to be compatible with K-modeling techniques, special conversion algorithms must be applied to construct hybrid models, as discussed in Sect. 10.
  - 3.6.5. Wave digital filters. WDFs in Sect. 8 are another wave-based modeling technique, originally developed for discrete-time simulation of analog electric circuits & networks. In their original form, WDFs are best suited for lumped element modeling; thus, they can be easily applied to wave-based mass–spring modeling. Due to their compatibility with digital waveguides, these methods complement each other. WDFs have also been extended to multidimensional networks & to systematic & energetically consistent modeling of nonlinearities. They have been applied particularly to deal with lumped & nonlinear elements in models, where wave propagation parts are typically realized by digital waveguides.
  - 3.6.6. Source–filter models. In Sect. 9 source–filter models form a paradigm between physics-based modeling & signal processing models. True spatial structure & bi-directional interactions are not visible, but are transformed into a transfer function that can be realized as a digital filter. Approach is attractive in sound synthesis because digital filters are optimized to implement transfer functions efficiently. Source part of a source–filter model is often a wavetable, consolidating different physical or synthetic signal components needed to feed filter part. Source–filter paradigm is frequently used in combination with other modeling paradigms in more or less ad hoc ways.

- 4. Finite difference models. Finite difference schemes can be used for solving PDEs, e.g. those describing vibration of a string, a membrane or an air column inside a tube [264]. Key idea in finite difference scheme: replace derivatives with finite difference approximations. An early example of this approach in physical modeling of musical instruments is work done by HILLER & RUIZ in early 1970s [113, 114]. This line of research has been continued & extended by CHAIGNE & colleagues [45, 46, 48] & recently by others [25, 26, 29, 30, 81, 103, 131].

  Finite difference approach leads to a simulation algorithm that is based on a difference equation, which can be easily programmed with a computer. E.g., how basic wave equation, which describes small-amplitude vibration of a lossless, ideally flexible string, is discretized using this principle. Here present a formulation after Smith [253] using an ideal string as a starting point for discrete-time modeling. A more thorough continuous-time analysis of physics of strings can be found in [96].

  - 4.1. Finite difference models for an ideal vibrating string. Fig. 1: Part of an ideal vibrating string. depicts a snapshot of an ideal (lossless, linear, flexible) vibrating string by showing displacement as a function of position. Wave equation for string is given by $\boxed{Ky'' = \epsilon\ddot{y}}$
  - 4.2. Boundary conditions & string excitation.
  - 4.3. Finite difference approximation of a lossy string.
  - 4.4. Stiffness in finite difference strings.

- 5. Mass-spring networks.
  - 5.1. Basic theory.
  - 5.2. CORDIS-ANIMA.
  - 5.3. Other mass-spring systems.

- 6. Modal decomposition methods.
  - 6.1. Modal synthesis.
  - 6.2. Filter-based modal methods.
  - 6.3. Functional transform method.

- 7. Digital waveguides.
  - 7.1. From wave propagation to digital waveguides.
  - 7.2. Modeling of losses & dispersion.
  - 7.3. Modeling of waveguide termination & scattering.
  - 7.4. Digital waveguide meshes & networks.
  - 7.5. Reduction of a DWG model to a single delay loop structure.
  - 7.6. Commuted DWG synthesis.
  - 7.7. Case study: modeling & synthesis of acoustic guitar. Acoustic guitar is an example of a musical instruments for which DWG modeling is found to be an efficient method, especially for real-time sound synthesis [134, 137, 142, 160, 286, 295]. DWG principle in Fig. 14: A DWG block diagram of 2 strings coupled through a common bridge impedance $Z_{\rm b}$ & terminated at other end by nut impedances $Z_{\rm t1}, Z_{\rm t2}$. Plucking points are for force insertion from wavetables WT_i into junctions in delay-lines DL_{ij}. Output is taken as bridge velocity. allows for true physically distributed modeling of strings & their interaction, while SDL commuted synthesis (Fig. 17: Reduction of bi-directional delay-line waveguide model (top) to a single delay line loop structure (bottom). & Fig. 18: Principles of commuted DWG synthesis: (a) cascaded excitation, string & body, (b) body & string blocks commuted & (c) excitation & body blocks consolidated into a wavetable for feeding string model.) allows for more efficient computation. In this subsect discuss principles of commuted waveguide synthesis as applied to high-quality synthesis of acoustic guitar.

    There are several features that must be added to simple commuted SDL structure in order to achieve natural sound & control of playing features. Fig. 19: Degrees of freedom for string vibration in guitar: Torsional, Longitudinal, Vertical, Horizontal. depicts degrees of freedom for vibration of strings in guitar. Transversal directions, i.e. vertical & horizontal polarizations of vibration, are most prominent ones. Vertical vibration connects strongly to bridge, resulting in stronger initial sound & faster decay than horizontal vibrations that start more weakly & decay more slowly. Effect of longitudinal vibration is weak but can be observed in generation of some partials of sound [320]. Longitudinal effects are more prominent in piano [16, 58], but are particularly important in such instruments as kantele [82] through nonlinear effect of tension modulation (Sect. 11). Torsional vibration of strings in guitar is not shown to have a remarkable effect on sound. In violin it has a more prominent physical role, although it makes virtually no contribution to sound.

    In commuted waveguide synthesis, 2 transversal polarizations can be realized by 2 separate SDL string models, $S_{\rm v}(z)$ for vertical & $S_{\rm h}(z)$ for horizontal polarization in Fig. 20: Dual-polarization string model with sympathetic vibration coupling between strings. Multiple wavetables are used for varying plucking styles. Filter $E(z)$ can control detailed timbre of plucking & $P(z)$ is a plucking point comb filter., each one with slightly different delay & decay parameters. Coefficient $m_{\rm p}$ is used to control relative excitation amplitudes of each polarization, depending on initial direction of string movement after plucking. Coefficients $m_{\rm o}$ can be used to mix vibration signal components at bridge.

Fig. 20 also shows another inherent feature of guitar, sympathetic coupling between strings at bridge, which causes an undamped string to gain energy from another string set in vibration. While principle shown in Fig. 14 implements this automatically if string & bridge admittances are correctly set, model in Fig. 20 requires special signal connections from point C to vertical polarization model of other strings. This is just a rough approximation of physical phenomenon that guarantees stability of model. There is also a connection through $g_{\rm c}$ that allows for simple coupling from horizontal polarization to excite vertical vibration, with a final result of a coupling between polarizations.

Dual-polarization model in Fig. 20 is excited by wavetables containing commuted waveguide excitations for different plucking styles. Filter $E(z)$ can be used to control timbre details of selected excitation, & filter $E(z)$ can be used to control timbre details of selected excitation, & filter $P(z)$ is a plucking point comb filter, as prev discussed.

– Mô hình phân cực kép trong Hình 20 được kích thích bằng các bảng sóng chứa các kích thích ống dẫn sóng chuyển mạch cho các kiểu gảy khác nhau. Bộ lọc $E(z)$ có thể được sử dụng để kiểm soát các chi tiết âm sắc của kích thích đã chọn, & bộ lọc $E(z)$ có thể được sử dụng để kiểm soát các chi tiết âm sắc của kích thích đã chọn, & bộ lọc $P(z)$ là bộ lọc lược điểm gảy, như đã thảo luận trước đó.

For solid body electric guitars, a magnetic pickup model is needed, but body effect can be neglected. Magnetic pickup can be modeled as a lowpass filter [124,137] in series with a comb filter similar to plucking point filter, but in this case corresponding to pickup position.

Calibration of model parameters is an important task when simulating a particular instrument. Methods for calibrating a string instrument model are presented, e.g., in [8, 14, 24, 27, 137, 142, 211, 244, 286, 295, 320].

– Hiệu chuẩn các tham số mô hình là 1 nhiệm vụ quan trọng khi mô phỏng 1 nhạc cụ cụ thể. Các phương pháp hiệu chuẩn mô hình nhạc cụ dây được trình bày . . .

A typical procedure: apply time-frequency analysis to recorded sound of plucked or struck string, in order to estimate decay rate of each harmonic. Parametric models e.g. FZ-ARMA analysis [133, 138] may yield more complete information of modal components in string behavior. This information is used to design a low-order loop filter which approximates frequency-dependent losses in SDL loop structure [14,17,79,244,286]. A recent novel idea has been to design a sparse FIR loop filter, which is of high order but has few nonzero coefficients [163, 209, 293]. This approach offers a computationally efficient way to imitate large deviations in decay rates of harmonic components. Through implementing a slight difference in delays & decay rates of 2 polarizations, beating or 2-stage decay of signal envelope can be approximated. for plucking point comb filter: required to estimate plucking point from a recorded tone [199, 276, 277, 286].

Fig. 21: Detailed SDL loop structure for string instrument sound synthesis. depicts a detailed structure used in practice to realize SDL loop. Fundamental frequency of string sound is inversely proportional to total delay of loop blocks. Accurate tuning requires application of a fractional delay, because an integral number of unit delays is not accurate enough when a fixed sampling rate is used. Fractional delays are typically approximated by 1st-order allpass filters or 1st- to 5th-order Lagrange interpolators as discussed in [154].

When loop filter properties are estimated properly, excitation wavetable signal is obtained by inverse filtering (deconvolution) of recorded sound by SDL response. For practical synthesis, only initial transient part of inverse-filtered excitation is used, typically covering several 10s of milliseconds.

After careful calibration of model, a highly realistic sounding synthesis can be obtained by parametric control & modification of sound features. Synthesis is possible even in cases which are not achievable in practice in real acoustic instruments.

- 7.8. DWG modeling of various musical instruments. Digital waveguide modeling has been applied to a variety of musical instruments other than acoustic guitar. In this subsect, present a brief overview of such models & features that need special attention to each case. For an in-depth presentation on DWG modeling techniques applied to different instrument families, see [254].
  - 7.8.1. Other plucked string instruments.
  - 7.8.2. Struck string instruments.
  - 7.8.3. Bowed string instruments.
  - 7.8.4. Wind instruments.
  - 7.8.5. Percussion instruments.
  - 7.8.6. Speech & singing voice.
  - 7.8.7. Inharmonic SDL type of DWG models.

- 8. Wave digital filters. Purpose of this sect: provide a general overview of physical modeling using WDFs in context of musical instruments. Only essential basics of topic will be discussed in detail; the rest will be glossed over. For more information about project, reader is encouraged to refer to [254]. Also, another definitive work can be found in [94].

  - 8.1. What are wave digital filters? WDFs were developed in late 1960s by Alfred Fettweis [93] for digitizing lumped analog electrical circuits. Traveling-wave formulation of lumped electrical elements, where WDF approach is based, was introduced earlier by Belevitch [21, 254].

    WDFs are certain types of digital filters with valid interpretations in physical world. I.e., can simulate behavior of a lumped physical system (hệ thống vật lý tập trung) using a digital filter whose coefficients depend on parameters of this physical system. Alternatively, WDFs can be seen as a particular type of finite difference schemes with excellent numerical properties [254]. As discussed in Sect. 4, task of finite difference schemes in general: provide discrete versions of PDEs for simulation & analysis purposes.

    WDFs are useful for physical modeling in many respects. 1stly, they are modular: same building blocks can be used for modeling very different systems; all that needs to be changed: topology of wave digital network. 2ndly, preservation of energy & hence also stability is usually addressed, since elementary blocks can be made passive, & energy preservation between blocks are evaluated using Kirchhoff's laws. Finally, WDFs have good numerical properties, i.e., they do not experience artificial damping at high frequencies.

    Physical systems were originally considered to be lumped in basic wave digital formalism. I.e., system to be modeled, say a drum, will become a point-like black box, which has functionality of drum. However, its inner representation, as well as its spatial dimensions, is lost. Must bear in mind, however: question of whether a physical system can be considered lumped depends naturally not only on which of its aspects wish to model but also on frequency scale want to use in modeling (Sect. 3).

  - 8.2. Analog circuit theory.
  - 8.3. Wave digital building blocks.
  - 8.4. Interconnection & adaptors.
  - 8.5. Physical modeling using WDFs.
  - 8.6. Current research.

- 9. Source-filter models.
  - 9.1. Subtractive synthesis in computer music.
  - 9.2. Source-filter models in speech synthesis.
  - 9.3. Instrument body modeling by digital filters.
  - 9.4. Karplus–Strong algorithm.
  - 9.5. Virtual analog synthesis.

- 10. Hybrid models.
  - 10.1. KW-hybrids.
  - 10.2. KW-hybrids modeling examples.

- 11. Modeling of nonlinear & time-varying phenomena.
  - 11.1. Modeling of nonlinearities in musical instruments.
  - 11.2. Case study: nonlinear string model using generalized time-varying allpass filters.
  - 11.3. Modeling of time-varying phenomena.

- 12. Current trends & further research.

- 13. Conclusions.

2 Librosa

`librosa` is a Python package for music & audio analysis. In provides building blocks necessary to create music information retrieval systems.

For a quick introduction to using librosa, see Tutorial. For a more advanced introduction which describes package design principles, refer to librosa paper at SciPy 2015.

2.1 Brian McFee, Colin Raffel, Dawen Liang, Daniel P. W. Ellis, Matt McVicar, Eric Battenberg, Oriol Nieto. librosa: Audio & Music Signal Analysis in Python

[3562 citations]

- Abstract. This document describes version 0.4.0 of `librosa`: a Python package for audio & music signal processing. At a high level, `librosa` provides implementations of a variety of common functions used throughout field of music information retrieval. In this document, a brief overview of library's functionality is provided, along with explanations of design goals, software development practices, & notational conventions.

- Index terms. audio, music, signal processing.

- Introduction. Emerging research field of music information retrieval (MIR) broadly covers topics at intersection of musicology, digital signal processing, ML, information retrieval, & library science. Although field is relatively young – 1st international symposium on music information retrieval (ISMIR) http://ismir.net was held in Oct of 2000 – it is rapidly developing, thanks in part to proliferation & practical scientific needs of digital music services, e.g. iTunes, Pandora, & Spotify. While preponderance of MIR research has been conducted with custom tools & scripts developed by researchers in a variety of languages e.g. MATLAB or C++, stability, scalability, & ease of use these tools has often left much to be desired.

  – Lĩnh vực nghiên cứu mới nổi về truy xuất thông tin âm nhạc (MIR) bao gồm rộng rãi các chủ đề giao thoa giữa âm nhạc học, xử lý tín hiệu số, ML, truy xuất thông tin, & khoa học thư viện. Mặc dù lĩnh vực này còn khá mới mẻ – hội nghị chuyên đề quốc tế đầu tiên về truy xuất thông tin âm nhạc (ISMIR) http://ismir.net đã được tổ chức vào tháng 10 năm 2000 – nhưng nó đang phát triển nhanh chóng, một phần là nhờ sự gia tăng & nhu cầu khoa học thực tế của các dịch vụ âm nhạc số, ví dụ như iTunes, Pandora, & Spotify. Trong khi phần lớn nghiên cứu MIR được tiến hành bằng các công cụ tùy chỉnh & tập lệnh do các nhà nghiên cứu phát triển bằng nhiều ngôn ngữ khác nhau, ví dụ như MATLAB hoặc C++, tính ổn định, khả năng mở rộng, & dễ sử dụng của các công cụ này thường không được như mong đợi.

  In recent years, interest has grown within MIR community in using (scientific) Python as a viable alternative. This has been driven by a confluence of several factors, including availability of high-quality ML libraries e.g. `scikit-learn` [Pedregosa11] & tools based on `Theano` [Bergstra11], as well as Python's vast catalog of packages for dealing with text data & web services. However, adoption of Python has been slowed by absence of a stable core library that provides basic routines upon which many MIR applications are built. To remedy this situation, have developed librosa: https://github.com/bmcfee/librosa a Python package for audio & music signal processing. [Name `librosa` is borrowed from *LabROSA*: LABoratory for Recognition & Organization of Speech & Audio at Columbia University, where initial development of librosa took place.] In doing so, hope to both ease transition of MIR researchers into Python (& modern software development practices), & also to make core MIR techniques readily available to broader community of scientists & Python programmers.

  ○ Design principles. In designing librosa, have prioritized a few key concepts. 1st, strive for a low barrier to entry for researchers familiar with MATLAB. In particular, opted for a relatively flat package layout, & following `scipy` [Jones01] rely upon `numpy` data types & functions [VanDerWalt11], rather than abstract class hierarchies.

    2nd, expended considerable effort in standardizing interfaces, variable names, & (default) parameter settings across various analysis functions. This task was complicated by fact: reference implementations from which our implementations are derived come from various authors, & are often designed as 1-off scripts rather than proper library functions with well-defined interfaces.

    – Thứ 2, dành nhiều công sức để chuẩn hóa giao diện, tên biến, cài đặt tham số & (mặc định) trên nhiều hàm phân tích khác nhau. Nhiệm vụ này phức tạp hơn vì thực tế: các triển khai tham chiếu mà các triển khai của chúng tôi bắt nguồn từ nhiều tác giả khác nhau, & thường được thiết kế dưới dạng tập lệnh 1 lần thay vì các hàm thư viện phù hợp với giao diện được xác định rõ ràng.

    3rd, wherever possible, retain backwards compatibility against existing reference implementations. This is achieved via regression testing for numerical equivalence of outputs. All tests are implemented in `nose` framework https://nose.readthedocs.org/en/latest/.

    – Thứ 3, bất cứ khi nào có thể, hãy duy trì khả năng tương thích ngược với các triển khai tham chiếu hiện có. Điều này đạt được thông qua thử nghiệm hồi quy để có sự tương đương về mặt số của đầu ra. Tất cả các thử nghiệm đều được triển khai trong `nose`.

    4th, because MIR is a rapidly evolving field, recognize: exact implementations provided by librosa may not represent state of art for any particular task. Consequently, functions are designed to be *modular*, allowing practitioners to provide their own functions when appropriate, e.g., a custom onset strength estimate may be provided to beat tracker as a function argument. This allows researchers to leverage existing library functions while experimenting with improvements to specific components. Although this seems simple & obvious, from a practical standpoint monolithic designs & lack of interoperability between different research codebases have historically made this difficult.

    – Thứ 4, vì MIR là một lĩnh vực phát triển nhanh chóng, hãy nhận ra: các triển khai chính xác do librosa cung cấp có thể không đại diện cho trạng thái nghệ thuật cho bất kỳ nhiệm vụ cụ thể nào. Do đó, các hàm được thiết kế theo dạng *mô-đun*, cho phép các học viên cung cấp các hàm của riêng họ khi thích hợp, ví dụ, có thể cung cấp ước tính cường độ khởi phát tùy chỉnh cho beat tracker dưới dạng đối số hàm. Điều này cho phép các nhà nghiên cứu tận dụng các hàm thư viện hiện có trong khi thử nghiệm các cải tiến đối với các thành phần cụ thể. Mặc dù điều này có vẻ đơn giản & hiển nhiên, nhưng theo quan điểm thực tế, các thiết kế nguyên khối & thiếu khả năng tương tác giữa các cơ sở mã nghiên cứu khác nhau trong lịch sử đã khiến điều này trở nên khó khăn.

    Finally, strive for readable code, thorough documentation & exhaustive testing. All development is conducted on GitHub. Apply modern software development practices, e.g. continuous integration testing (via Travis https://travis-ci.org) & coverage(via Coveralls https://coveralls.io). All functions are implemented in pure Python, thoroughly documented using Sphinx, & include example code demonstrating usage. Implementation mostly complies with PEP-8 recommendations, with a small set of exceptions for variable names that make code more concise without sacrificing clarity: e.g., `y, sr` are preferred over more verbose names e.g. `audio_buffer, sampling_rate`.

    – Cuối cùng, hãy cố gắng tạo ra mã dễ đọc, tài liệu hướng dẫn đầy đủ & thử nghiệm toàn diện. Mọi hoạt động phát triển đều được thực hiện trên GitHub. Áp dụng các phương pháp phát triển phần mềm hiện đại, ví dụ như thử nghiệm tích hợp liên tục (qua Travis https://travis-ci.org) & phạm vi phủ sóng (qua Coveralls https://coveralls.io). Tất cả các chức năng đều được triển khai bằng Python thuần túy, được ghi chép đầy đủ bằng Sphinx, & bao gồm mã ví dụ minh họa cách sử dụng. Việc triển khai phần lớn tuân thủ các khuyến nghị của PEP-8, với một tập hợp nhỏ các ngoại lệ cho tên biến giúp mã ngắn gọn hơn mà không làm mất đi tính rõ ràng: ví dụ: `y, sr` được ưu tiên hơn các tên dài dòng hơn, ví dụ: `audio_buffer, sampling_rate`.

- Conventions. In general, librosa's functions tend to expose all relevant parameters to caller. While this provides a great deal of flexibility to expert users, it can be overwhelming to novice users who simply need a consistent interface to process audio files. To satisfy both needs, define a set of general conventions & standardized default parameter values shared across many functions.

  An audio signal is represented as a 1D `numpy` array, denoted as `y` throughout librosa. Typically signal `y` is accompanied by *sampling rate* (denoted `sr`) which denotes frequency (in Hz) at which values of `y` are sampled. Duration of a signal can then be computed by dividing number of samples by sampling rate:

  ```
  >>> duration_seconds = float(len(y)) / sr
  >>> duration_seconds = float(len(audio_buffer)) / sampling_rate
  ```

  By default, when loading stereo audio files, `librosa.load()` function downmixes to mono by averaging left- & right-channels, & then resamples monophonic signal to default rate `sr = 22050` Hz.

  Most audio analysis methods operate not at native sampling rate of signal, but over small *frames* of signal which are spaced by a *hop length* (in samples). Default frame & hop lengths are set to 2048 & 512 samples, resp. At default sampling rate of 22050 Hz, this corresponds to overlapping frames of ≈ 93 ms spaced by 23 ms. Frames are centered by default, so frame index `t` corresponds to slice:

  ```
  y[(t * hop_length - frame_length / 2):
  (t * hop_length + frame_length / 2)],
  ```

  where boundary conditions are handled by reflection-padding input signal y. Unless otherwise specified, all sliding-window analyses use Hann windows by default. For analyses that do not use fixed-width frames (e.g. constant-Q transform), default hop length of 512 is retained to facilitate alignment of results.

  Majority of feature analyses implemented by librosa produce 2D outputs stored as `numpy.ndarray`, e.g., `S[f, t]` might contain energy within a particular frequency band `f` at frame index `t`. Follow convention: final dimension provides index over time, e.g., `S[:, 0], S[:, 1]` access features at 1st & 2nd frames. Feature arrays are organized column-major (Fortran style) in memory, so that common access patterns benefit from cache locality.

  By default, all pitch-based analyses are assumed to be relative to a 12-bin equal-tempered chromatic scale with a reference tuning of `A440 = 440.0 Hz`. Pitch & pitch-class analyses are arranged s.t. 0th bin corresponds to `C` for pitch class or `C1` (32.7 Hz) for absolute pitch measurements.

- Package organization. In this sect, give a brief overview of structure of librosa software package. This overview is intended to be superficial & cover only most commonly used functionality. A complete API reference can be found.

  - *Core functionality.* `librosa.core` submodule includes a range of commonly used functions. Broadly, `core` functionality falls into 4 categories: audio & time-series operations, spectrogram calculation, time & frequency conversion, & pitch operations. For convenience, all functions within `core` submodule are aliased at top level of package hierarchy, e.g., `librosa.core.load` is aliased to `librosa.load`.

    Audio & time-series operations include functions e.g.: reading audio from disk via `audioread` package https://github.com/sampsyo/audioread `core.load`, resampling a signal at a desired rate `core.resample`, stereo to mono conversion `core.to_mono`, time-domain bounded auto-correlation `core.autocorrelate`, & 0-crossing detection `core.zero_crossings`.

    Spectrogram operations include short-time Fourier transform `stft`, inverse STFT `istft`, & instantaneous frequency spectrogram `ifgram` [Abe95], which provide much of core functionality for down-stream feature analysis. Additionally, an efficient constant-Q transform `cqt` implementation based upon recursive down-sampling method of Schokerkhuber & Klapuri [Schoerkhuber10] is provided, which provides logarithmically-spaced frequency representations suitable for pitch-based signal analysis. Finally, `logamplitude` provides a flexible & robust implementation of log-amplitude scaling, which can be used to avoid numerical underflow & set an adaptive noise floor when converting from linear amplitude.

    – Các hoạt động phổ đồ bao gồm biến đổi Fourier thời gian ngắn `stft`, STFT nghịch đảo `istft`, phổ đồ tần số tức thời `ifgram` [Abe95], cung cấp nhiều chức năng cốt lõi cho phân tích tính năng hạ lưu. Ngoài ra, một triển khai biến đổi Q hằng số `cqt` hiệu quả dựa trên phương pháp lấy mẫu xuống đệ quy của Schokerkhuber & Klapuri [Schoerkhuber10] được cung cấp, cung cấp các biểu diễn tần số cách đều theo logarit phù hợp cho phân tích tín hiệu dựa trên cao độ. Cuối cùng, `logamplitude` cung cấp một triển khai linh hoạt & mạnh mẽ của tỷ lệ biên độ logarit, có thể được sử dụng để tránh tràn số & thiết lập sàn nhiễu thích ứng khi chuyển đổi từ biên độ tuyến tính.

    Because data may be represented in a variety of time or frequency units, provide a comprehensive set of convenience functions to map between different time representations: secs, frames, or samples; & frequency representations: hertz, constant-Q basis index, Fourier basis index, Mel basis index, MIDI note number, or note in scientific pitch notation.

    Finally, core submodule provides functionality to estimate dominant frequency of STFT bins via parabolic interpolation `piptrack` [Smith11], & estimation of tuning deviation (in cents) from reference `A440`. These functions allow pitch-based analyses (e.g., `cqt`) to dynamically adapt filter banks to match global tuning offset of a particular audio signal.

- Spectral features. Spectral representations – distributions of energy over a set of frequencies – form basis of many analysis techniques in MIR & digital signal processing in general. `librosa.feature` module implements a variety of spectral representations, most of which are based upon short-time Fourier transform.

  Mel frequency scale is commonly used to represent audio signals, as it provides a rough model of human frequency perception [Stevens37]. Both a Mel-scale spectrogram `librosa.feature.melspectrogram` & commonly used Mel-frequency Cepstral Coefficients (MFCC) `librosa.feature.mfcc` are provided. By default, Mel scales are defined to match implementation provided by SLANEY's auditory toolbox [Slaney98], but they can be made to match Hidden Markov Model Toolkit (HTK) by setting flag `htk = True` [Young97].

  While Mel-scaled representations are commonly used to capture timbral aspects of music, they provide poor resolution of pitches & pitch classes. Pitch class (or *chroma*) representations are often used to encode harmony while suppressing variations in octave height, loudness, or timbre. 2 flexible chroma implementations are provided: one uses a fixed-window STFT analysis `chroma_stft` [`chroma_stft` is based upon reference implementation provided at http://labrosa.ee.columbia.edu/matlab/chroma-ansyn/] (dead link) & other uses variable-window constant-Q transform analysis `chroma_cqt`. An alternative representation of pitch & harmony can be obtained by `tonnetz` function, which estimates tonal centroids as coordinates in a 6D interval space using method of Harte et al. [Harte06]. Fig. 1: 1st: short-time Fourier transform of a 20-sec audio clip `librosa.stft`. 2nd: corresponding Mel spectrogram, using 128 Mel bands `librosa.feature.melspectrogram`. 3rd: corresponding chromagram `librosa.feature.chroma_cqt`. 4: Tonnetz features `librosa.feature.tonnetz` illustrates difference between STFT, Mel spectrogram, chromagram, & Tonnetz representations, as constructed by following code fragment: [For display purposes, spectrograms are scaled by `librosa.logamplitude`. Refer readers to accompanying IPython notebook for full source code to reconstruct figures.]

  ```
  >>> filename = librosa.util.example_audio_file()
  >>> y, sr = librosa.load(filename, offset=25.0, duration=20.0)
  >>> spectrogram = np.abs(librosa.stft(y))
  >>> melspec = librosa.feature.melspectrogram(y=y, sr=sr)
  >>> chroma = librosa.feature.chroma_cqt(y=y, sr=sr)
  >>> tonnetz = librosa.feature.tonnetz(y=y, sr=sr)
  ```

  In addition to Mel & chroma features, `feature` submodule provides a number of spectral statistic representations, including `spectral_centroid, spectral_bandwidth, spectral_rolloff` [Klapuri07], & `spectral_contrast` [Jiang02]. [`spectral_*` functions are derived from MATLAB reference implementations provided by METLab at Drexel University http://music.ece.drexel.edu/.]

  Finally, `feature` submodule provides a new functions to implement common transformations of time-series features in MIR. This includes `delta`, which provides a smoothed estimate of time derivative; `stack_memory`, which concatenates an input feature array with time-lagged copies of itself (effectively simulating feature n-grams); & `sync`, which applies a user-supplied aggregation function, e.g., `numpy.mean` or `median`, across specified column intervals.

- Display. `display` module provides simple interfaces to visually render audio data through `matplotlib` [Hunter07]. 1st function, `display.waveplot` simply renders amplitude envelope of an audio signal `y` using matplotlib's `fill_between` function. For efficiency purposes, signal is dynamically down-sampled. Mono signals are rendered symmetrically about horizontal axis; stereo signals are rendered with left-channel's amplitude above axis & right-channel's below. An example of `waveplot` is depicted in Fig. 2: Top: a waveform plot for a 20-sec audio clip `y`, generated by `librosa.display.waveplot`. Middle: log-power short-time Fourier transform (STFT) spectrum for `y` plotted on a logarithmic frequency scale, generated by `librosa.display.specshow`. Bottom: onset strength function `librosa.onset.onset_strength`, detected onset events `librosa.onset.onset_detect`, & detected beat events `librosa.beat.beat_track` for `y`.

  2nd function, `display.specshow` wraps matplotlib's `imshow` function with default settings (`origin, aspect`) adapted to expected defaults for visualizing spectrograms. Additionally, `specshow` dynamically selects appropriate colormaps (binary, sequential, or diverging) from data type & range. [If `seaborn` package [Waskom14] is available, its version of *cubehelix* is used for sequential data.] Finally, `specshow` provides a variety of acoustically relevant axis labeling & scaling parameters. Examples of `specshow` output are displayed in Figs. 1–2 (middle).

- Onsets, tempo, & beats. While spectral feature representations described above capture frequency information, time information is equally important for many applications in MIR. E.g., it can be beneficial to analyze signals indexed by note or beat events, rather than absolute time. `onset, beat` submodules implement functions to estimate various aspects of timing in music.

  More specially, `onset` module provides 2 functions: `onset_strength, onset_detect`. `onset_strength` function calculates a thresholded spectral flux operation over a spectrogram, & returns a 1D array representing amount of increasing spectral energy at each frame. This is illustrated as blue curve in bottom panel of Fig. 2. `onset_detect` function, on other hand, selects peak positions from onset strength curve following heuristic described by Boeck et al. [Boeck12]. Output of `onset_detect` is depicted as red circles in bottom panel of Fig. 2.

  – Cụ thể hơn, mô-đun `starts` cung cấp 2 hàm: `onset_strength, starts_detect`. Hàm `onset_strength` tính toán một phép toán thông lượng phổ ngưỡng trên một phổ đồ, & trả về một mảng 1D biểu diễn lượng năng lượng phổ tăng dần tại mỗi khung. Điều này được minh họa bằng đường cong màu xanh lam ở bảng dưới cùng của Hình 2. Mặt khác, hàm `onset_detect` chọn các vị trí đỉnh từ đường cong cường độ khởi đầu theo phương pháp tìm kiếm được mô tả bởi Boeck et al. [Boeck12]. Đầu ra của `onset_detect` được mô tả bằng các vòng tròn màu đỏ ở bảng dưới cùng của Hình 2.

beat module provides functions to estimate global tempo & positions of beat events from onset strength function, using method of Ellis [Ellis07]. More specifically, beat tracker 1st estimates tempo, which is then used to set target spacing between peaks in an onset strength function. Output of beat tracker is displayed as dashed green lines in Fig. 2 (bottom).

Typing this all together, tempo & beat positions for an input signal can be easily calculated by following code fragment:

```
>>> y, sr = librosa.load(FILENAME)
>>> tempo, frames = librosa.beat.beat_track(y=y, sr=sr)
>>> beat_times = librosa.frames_to_time(frames, sr=sr)
```

Any of default parameters & analyzes may be overridden. E.g., if user has calculated an onset strength envelope by some other means, it can be provided to beat tracker as follows:

```
>>> oenv = some_other_onset_function(y, sr)
>>> librosa.beat.beat_track(onset_envelope=oenv)
```

All detection functions (beat & onset) return events as frame indices, rather than absolute timing. Downside of this: left to user to convert frame indices back to absolute time. However, in our opinion, this is outweighed by 2 practical benefits: it simplifies implementations, & it makes results directly accessible to frame-indexed functions e.g. librosa.feature.sync.

- Structural analysis. Onsets & beats provide relatively low-level timing cues for music signal processing. Higher-level analyses attempt to detect larger structure in music, e.g., at level of bars or functional components e.g. *verse, chorus*. While this is an active area of research that has seen rapid progress in recent years, there are some useful features common to many approaches. segment submodule contains a few useful functions to facilitate structural analysis in music, falling broadly into 2 categories.

  – Phân tích cấu trúc. Các nhịp khởi đầu & cung cấp tín hiệu thời gian ở mức tương đối thấp để xử lý tín hiệu âm nhạc. Các phân tích ở mức cao hơn cố gắng phát hiện cấu trúc lớn hơn trong âm nhạc, ví dụ, ở mức ô nhịp hoặc các thành phần chức năng ví dụ *verse, chorus*. Mặc dù đây là một lĩnh vực nghiên cứu tích cực đã chứng kiến sự tiến bộ nhanh chóng trong những năm gần đây, nhưng có một số tính năng hữu ích chung cho nhiều phương pháp tiếp cận. Mô-đun con segment chứa một số hàm hữu ích để tạo điều kiện thuận lợi cho phân tích cấu trúc trong âm nhạc, về cơ bản được chia thành 2 loại.

  1st, there are functions to calculate & manipulate *recurrence* or *self-similarity* plots. segment.recurrence_matrix constructs a binary k-nearest-neighbor similarity matrix from a given feature array & a user-specified distance function. As displayed in Fig. 3: left: recurrence plot derived from chroma features displayed in Fig. 1., repeating sequences often appear as diagonal bands in recurrence plot, which can be used to detect musical structure. Sometimes more convenient to operate in *time-lag* coordinates, rather than *time-time*, which transforms diagonal structures into more easily detectable horizontal structure Fig. 3: right: corresponding time-lag plot. [Serra12]. This is facilitated by recurrence_to_lag (& lag_to_recurrence) functions.

  2nd, temporally constrained clustering can be used to detect feature change-points without relying upon repetition. This is implemented in librosa by segment.agglomerative function, which uses scikit-learn's implementation of WARD's agglomerative clustering method [Ward63] to partition input into a user-defined number of contiguous components. In practice, a user can override default clustering parameters by providing an existing sklearn.cluster.AgglomerativeClustering object as an argument to segment.agglomerative().

- Decompositions. Many applications in MIR operate upon latent factor representations, or other decompositions of spectrograms. E.g., common to apply nonnegative matrix factorization (NMF) [Lee99] to magnitude spectra, & analyze statistics of resulting time-varying activation functions, rather than raw observations.

  decompose module provides a simple interface to factor spectrograms (or general feature arrays) into *components & activations*:

```
>>> comps, acts = librosa.decompose.decompose(S)
```

  By default, decompose() function constructs a scikit-learn NMF object, & applies its fit_transform() method to transpose of S. Resulting basis components & activations are accordingly transposed, so that comps.dot(acts) approximates S. If user wishes to apply some other decomposition technique, any object fitting sklearn.decomposition interface may be substituted:

```
>>> T = SomeDecomposer()
>>> librosa.decompose.decompose(S, transformer=T)
```

  In addition to general-purpose matrix decomposition techniques, librosa also implements harmonic-percussion source separation (HPSS) method of FITZGERALD [Fitzgerald10] as decompose.hpss. This technique is commonly used in MIR to suppress transients when analyzing pitch content, or suppress stationary signals when detecting onsets or other rhythmic elements. An example application of HPSS is illustrated in Fig. 4: Top: separated harmonic & percussive waveforms. Middle: Mel spectrogram of harmonic component. Bottom: Mel spectrogram of percussive component.

- Effects. effects module provides convenience functions to applying spectrogram-based transformations to time-domain signals. E.g., rather than writing

```
>>> D = librosa.stft(y)
>>> Dh, Dp = librosa.decompose.hpss(D)
>>> y_harmonic = librosa.istft(Dh)
```

one may simply write

```
>>> y_harmonic = librosa.effects.harmonic(y)
```

Convenience functions are provided for HPSS (retaining harmonic, percussive, or both components), time-stretching & pitch shifting. Although these functions provide no additional functionality, their inclusion results in simpler, more readable application code.

- Output. `output` module includes utility functions to save results of audio analysis to disk. Most often, this takes form of annotated instantaneous event timings or time intervals, which are saved in plain text (comma- or tab-separated values) via `output.times_csv, output.annotation`, resp. These functions are somewhat redundant with alternative functions for text output (e.g., `numpy.savetxt`), but provide sanity checks for length agreement & semantic validation of time intervals. Resulting outputs are designed to work with other common MIR tools, e.g. `mir_eval` [Raffel14] & `sonic-visualiser` [Cannam10].

  `output` module also provides `wirte_wav` function for saving audio in `.wave` format. `write_wav` simply wraps built-in `scipy` wave-file writer `scipy.io.wavfile.write` with validation & optional normalization, thus ensuring: resulting audio files are well-formed.

- Caching. MIR applications typically require computing a variety of features (e.g., MFCCs, chroma, beat timings, etc.) from each audio signal in a collection. Assuming application programmer is content with default parameters, simplest way to achieve this: call each function using audio time-series input, e.g.:

```
>>> mfcc = librosa.feature.mfcc(y=y, sr=sr)
>>> tempo, beats = librosa.beat.beat_track(y=y, sr=sr)
```

However, because there are shared computations between different functions – `mfcc` & `beat_track` both compute log-scaled Mel spectrograms, e.g. – this results in redundant (& inefficient) computation. A more efficient implementation of above example would factor out redundant features:

```
>>> lms = librosa.logamplitude(librosa.feature.melspectrogram(y=y, sr=sr))
>>> mfcc = librosa.feature.mfcc(S=lms)
>>> tempo, beats = librosa.beat.beat_track(S=lms, sr=sr)
```

Although it is more computationally efficient, above example is less concise, & it requires more knowledge of implementations on behalf of application programmer. More generally, nearly all functions in librosa eventually depend upon STFT calculation, but rare: application programmer will need STFT matrix as an end-result.

1 approach to eliminate redundant computation: decompose various functions into blocks which can be arranged in a computation graph, as is done in Essentia [Bogdanov13]. However, this approach necessarily constrains function interfaces, & may become unwieldy for common, simple applications.

Instead, librosa takes a lazy approach to eliminating redundancy via *output caching*. Caching is implemented through an extension of `Memory` class from `joblib` package `https://github.com/joblib/joblib`, which provides disk-backed memoization of function outputs. Cache object `librosa.cache` operates as a decorator on all non-trivial computations. This way, a user can write simple application code (i.e., 1st example above) while transparently eliminating redundancies & achieving speed comparable to more advanced implementation (2nd example).

Cache object is disabled by default, but can be activated by setting environment variable `LIBROSA_CACHE_DIR` prior to importing package. Because `Memory` object does not implement a cache eviction policy (as of version 0.8.4), recommended: users purge cache after processing each audio file to prevent cache from filling all available disk space [Cache can be purged by calling `librosa.cache.clear()`.] Note: this can potentially introduce race conditions in multi-processing environments (i.e., parallel batch processing of a corpus), so care must be taken when scheduling cache purges.

– Đối tượng bộ nhớ đệm bị vô hiệu hóa theo mặc định, nhưng có thể được kích hoạt bằng cách thiết lập biến môi trường `LIBROSA_CACHE_DIR` trước khi nhập gói. Vì đối tượng `Memory` không triển khai chính sách xóa bộ nhớ đệm (kể từ phiên bản 0.8.4), nên khuyến nghị: người dùng xóa bộ nhớ đệm sau khi xử lý từng tệp âm thanh để ngăn bộ nhớ đệm lấp đầy toàn bộ dung lượng đĩa khả dụng [Bộ nhớ đệm có thể được xóa bằng cách gọi `librosa.cache.clear()`.] Lưu ý: điều này có khả năng gây ra tình trạng chạy đua trong môi trường đa xử lý (tức là xử lý hàng loạt song song của một ngữ liệu), do đó phải cẩn thận khi lên lịch xóa bộ nhớ đệm.

- Parameter tuning. Some of librosa's functions have parameters that require some degree of tuning to optimize performance. In particular, performance of beat tracker & onset detection functions can vary substantially with small changes in certain key parameters.

  After standardizing certain default parameters – sampling rate, frame length, & hop length – across all functions, optimized beat tracker settings using parameter grid given in Table 1: Parameter grid for beat tracking optimization. Best configuration is indicated in bold. To select best-performing configuration, evaluated performance on a data set comprised of Isophonics Beatles corpus http://isophonics.net/content/reference-annotations & SMC Dataset 2 [Holzapfel12] beat annotations. Each configuration was evaluated using `mir_eval` [Raffel14], & configuration was chosen to maximize Correct Metric Level (Total) metric [Davies14].

  Similarly, onset detection parameters (listed in Table 2: Parameter grid for onest detection optimization. Best configuration is indicated in bold.) were selected to optimize F1-score on Johannes Kepler University onset database https://github.com/CPJKU/onset_db.

  Note: "optimal" default parameter settings are merely estimates, & depend upon datasets over which they are selected. Parameter settings are therefore subject to change in future as larger reference collections become available. Optimization framework has been factored out into a separate repository, which may in subsequent versions grow to include additional parameters. https://github.com/bmcfee/librosa_parameters

- Conclusion. This document provides a brief summary of design considerations & functionality of librosa. More detailed examples, notebooks, & documentation can be found in development repository & project website. Project is under active development, & roadmap for future work includes efficiency improvements & enhanced functionality of audio coding & file system interactions.

3 Wikipedia

3.1 Wikipedia/computer music

"*Computer music* is application of computing technology in music composition, to help human composers create new music or to have computers independently create music, e.g. with algorithmic composition programs. it includes theory & application of new & existing computer software technologies & basic aspects of music, e.g. sound synthesis, digital signal processing, sound design, sonic diffusion, acoustics, electrical engineering, & psychoacoustics. Field of computer music can trace its roots back to origins of electric music, & 1st experiments & innovations with electronic instruments at turn of 20th century.

3.1.1 History

3.1.2 Advances

3.1.3 Research

3.1.4 Machine improvisation

3.1.5 Live coding

" – Wikipedia/computer music

3.2 Wikipedia/transcription (music)

"A J. S. Bach keyboard piece transcribed for guitar. In music, *transcription* is practice of notating a piece or a sound which was previously unnotated &/or unpopular as a written music, e.g., a jazz improvisation or a video game soundtrack. When a musician is tasked with creating sheet music from a recording & they write down notes that make up piece in music notation, it is said: they created a *musical transcription* of that recording. Transcription may also mean rewriting a piece of music, either solo or ensemble, for another instrument or other instruments than which it was originally intended. Beethoven Symphonies transcribed for solo piano by Franz Liszt are an example. Transcription in this sense is sometimes called *arrangement*, although strictly speaking transcriptions are faithful adaptations, whereas arrangements change significant aspects of original piece.

Further examples of music transcription include ethnomusicological notation of oral traditions of folk music, e.g. Béla Bartók's & Ralph Vaughan Williams' collections of national folk music of Hungary & England resp. French composer Olivier Messiaen transcribed birdsong in wild, & incorporated it into many of his compositions, e.g. his Catalogue d'oiseaux for solo piano. Transcription of this nature involves scale degree recognition & harmonic analysis, both of which transcriber will need relative or perfect pitch to perform.

In popular music & rock, there are 2 forms of transcription. Individual performers copy a note-for-note guitar solo or other melodic line. As well, music publishers transcribe entire recordings of guitar solos & bass lines & sell sheet music in bound books. Music publishers also publish PVG (piano/vocal/guitar) transcriptions of popular music, where melody line is transcribed, & then accompaniment on recording is arranged as a piano part. Guitar aspect of PVG label is achieved through guitar chords written above melody. Lyrics are also included below melody.

3.2.1 Adaptation

Some composers have rendered homage to other composers by creating "identical" versions of earlier composers' pieces while adding their own creativity through use of completely new sounds arising from difference in instrumentation. Most widely known example of this is RAVEL's arrangement for orchestra of MUSSORGSKY's piano piece *Pictures at an Exhibition*. WEBERN used his transcription for orchestra of 6-part ricercar from BACH's *The Musical Offering* to analyze structure of Bach piece, by using different instruments to play different subordinate motifs of Bach's themes & melodies.

In transcription of this form, new piece can simultaneously imitate original sounds while recomposing them with all technical skills of an expert composer in such a way that it seems: piece was originally written for new medium. But some transcriptions & arrangements have been done for purely pragmatic or contextual reasons. E.g., in Mozart's time, overtures & songs from this popular operas were transcribed for small wind ensemble simply because such ensembles were common ways of providing popular entertainment in public places. MOZART himself did this in his own opera *The Marriage of Figaro*. A more contemporary example is STRAVINSKY's transcription for 4 hands piano of *The Rite of Spring*, to be used on ballet's rehearsals. Today musicians who play in cafes or restaurants will sometimes play transcriptions or arrangements of pieces written for a larger group of instruments.

Other examples of this type of transcription include BACH's arrangement of VIVALDI's 4-violin concerti for 4 keyboard instruments & orchestra; MOZART's arrangement of some Bach fugues from *The Well-Tempered Clavier* for string trio; BEETHOVEN's arrangement of his *Große Fuge*, originally written for string quartet, for piano duet, & his arrangement of his Violin Concerto as a piano concerto; Franz Liszt's piano arrangements of works of many composers, including symphonies of Beethoven; TCHAIKOVSKY's arrangement of 4 Mozart piano pieces into an orchestral suite called "Mozartiana"; MAHLER's re-orchestration of SCHUMANN symphonies; & SCHOENBERG's arrangement for orchestra of BRAHMS's piano quintet & BACH's "St. Anne" Prelude & Fugue for organ.

Since piano became a popular instrument, a large literature has sprung up of transcriptions & arrangements for piano of works for orchestra or chamber music ensemble. These are sometimes called "piano reductions", because multiplicity of orchestral parts – in an orchestral piece there may be as many as 2 dozen separate instrumental parts being played simultaneously – has to be reduced to what a single pianist (or occasionally 2 pianists, or 1 or 2 pianos, e.g. different arrangements for GEORGE GERSHWIN's *Rhapsody in Blue*) can manage to play.

Piano reductions are frequently made of orchestral accompaniments to choral works, for purposes of rehearsal or of performance with keyboard alone.

Many orchestral pieces have been transcribed for concert band.

3.2.2 Transcription aids

- **Notation software.** Since advent of desktop publishing, musicians can acquire music notation software, which can receive user's mental analysis of notes & then store & format those notes into standard music notation for personal printing or professional publishing of sheet music. Some notation software can accept a Standard MIDI File (SMF) or MIDI performance as input instead of manual note entry. These notation applications can export their scores in a variety of formats like EPS, PNG, & SVG. Often software contains a sound library that allows user's score to be played aloud by application for verification.

- **Slow-down software.** Prior to invention of digital transcription aids, musicians would slow down a record or a tape recording to be able to hear melodic lines & chords at a slower, more digestible pace. Problem with this approach was: it also changed pitches, so once a piece was transcribed, it would then have to be transposed into correct key. Software designed to slow down tempo of music without changing pitch of music can be very helpful for recognizing pitches, melodies, chords, rhythms, & lyrics when transcribing music. However, unlike slow-down effect of a record player, pitch & original octave of notes will stay same, & not descend in pitch. This technology is simple enough that it is available in many free software applications.

  Software generally goes through a 2-step process to accomplish this. 1st, audio file is played back at a lower sample rate than that of original file. This has same effect as playing a tape or vinyl record at slower speed – pitch is lowered meaning music can sound like it is in a different key. 2nd step: use Digital Signal Processing (or DSP) to shift pitch back up to original pitch level or musical key.

- **Pitch tracking software.** Main article: Wikipedia/pitch tracker. As mentioned in the Automatic music transcription sect, some commercial software can roughly track pitch of dominant melodies in polyphonic musical recordings. Note scans are not exact, & often need to be manually edited by user before saving to file in either a proprietary file format or in Standard MIDI File Format. Some pitch tracking software also allows scanned note lists to be animated during audio playback.

3.2.3 Automatic music transcription (AMT)

Term "automatic music transcription" was 1st used by audio researchers JAMES A. MOORER, MARTIN PISZCZALSKI, & BERNARD GALLER in 1977. With their knowledge of digital audio engineering, these researchers believed: a computer could be programmed to analyze a digital recording of music s.t. pitches of melody lines & chord patterns could be detected, along with rhythmic accents of percussion instruments. Task of AMT concerns 2 separate activities: making an analysis of a musical piece, & printing out a score from that analysis.

This was not a simple goal, but one that would encourage academic research for at least another 3 decades. Because of close scientific relationship of speech to music, much academic & commercial research that was directed toward more financially resourced speech recognitions technology would be recycled into research about music recognition technology. While many musicians & educators insist that manually doing transcriptions is a valuable exercise for developing musicians, motivation for

AMT remains same as motivation for sheet music: musicians who do not have intuitive transcription skills will search for sheet music or a chord chart, so that they may quickly learn how to play a song. A collection of tools created by this ongoing research could be of great aid to musicians. Since much recorded music does not have available sheet music, an automatic transcription device could also offer transcriptions that are otherwise unavailable in sheet music. To date, no software application can yet completely fulfill James Moorer;s definition of AMT. However, pursuit of AMT has spawned creation of many software applications that can aid in manual transcription. Some can slow down music while maintaining original pitch & octave, some can track pitch of melodies, some can track chord changes, & others can track beat of music.

Automatic transcription most fundamentally involves identifying pitch & duration of performed notes. This entails tracking pitch & identifying note onsets. After capturing those physical measurements, this information is mapped into traditional music notation, i.e., sheet music.

Digital Signal Processing is branch of engineering that provides software engineers with tools & algorithms needed to analyze a digital recording in terms of pitch (note detection of melodic instruments), & energy content of un-pitched sounds (detection of percussion instruments). Musical recordings are sampled at a given recording rate & its frequency data is stored in any digital wave format in computer. Such format represents sound by digital sampling.

- **Pitch detection.** Pitch detection is often detection of individual notes that might make up a melody in music, or notes in a chord. When a single key is pressed upon a piano, what we hear is not just *1* frequency of sound vibration, but a *composite* of multiple sound vibrations occurring at different mathematically related frequencies. Elements of this composite of vibrations at differing frequencies are referred to as harmonics or partials.

  E.g., if note A_3 (220 Hz) is played, individual frequencies of composite's harmonic series will start at 220 Hz as fundamental frequency: 440 Hz would be 2nd harmonic, 660 Hz would be 3rd harmonic, 880 Hz would be 4th harmonic, etc. These are integer multiples of fundamental frequency (e.g., $2 \cdot 220 = 440$, 2nd harmonic). While only about 8 harmonics are really needed to audibly recreate note, total number of harmonics in this mathematical series can be large, although higher harmonic's numerical weaker magnitude & contribution of that harmonic. Contrary to intuition, a musical recording at its lowest physical level is not a collection of individual notes, but is really a collection of individual harmonics. That is why very similar-sounding recordings can be created with differing collections of instruments & their assigned notes. As long as total harmonics of recording are recreated to some degree, it does not really matter which instruments or which notes were used.
- **Beat detection.**
- **How ATM works.**
- **Detailed computer steps behind AMT.**

" – Wikipedia/transcription (music)

4 Miscellaneous

Tài liệu

[HWR22] Michael S. Horn, Melanie West, and Cameron Roberts. *Introduction to Digital Music with Python Programming: Learning Music with Code.* 1st edition. Focal Press, 2022, p. 262.

[Väl+06] Vesa Välimäki, Jyri Pakarinen, Cumhur Erkut, and Matti Karjalainen. "Discrete-time modelling of musical instruments". In: *Rep. Prog. Phys.* 69.1 (2006), pp. 1–78. DOI: 10.1088/0034-4885/69/1/R01. URL: https://iopscience.iop.org/article/10.1088/0034-4885/69/1/R01.