Overview

When a guitar string is plucked, the string vibrates and creates sound. The length of the string determines its *fundamental frequency* of vibration. We model a guitar string by sampling its *displacement* (a real number between $-\frac{1}{2}$ and $+\frac{1}{2}$) at N equally-spaced points (in time), where N equals the *sampling rate* (24000 Hz) divided by the fundamental frequency (rounded to the nearest integer). We store these displacement values in a data structure that we will refer to as a *ring buffer*.

The excitation of the string can contain energy at any frequency. We simulate the excitation by filling the buffer with white noise: set each of the N sample displacements to a random real number between $-\frac{1}{2}$ and $+\frac{1}{2}$.

After the string is plucked, the string vibrates. The pluck causes a displacement which spreads wave-like over time. The Karplus–Strong algorithm simulates this vibration by maintaining a ring buffer of the N samples: the algorithm repeatedly deletes the first sample from the buffer and adds to the end of the buffer the average of the first two samples, scaled by an energy decay factor of 0.996.

Why it works

The two primary components that make the Karplus–Strong algorithm work are the ring buffer feedback mechanism and the averaging operation.

- The ring buffer models the medium (a string tied down at both ends) in which the energy travels back and forth. The length of the ring buffer determines the fundamental frequency of the resulting sound. Acoustically, the feedback mechanism reinforces only the fundamental frequency and its harmonics (frequencies at integer multiples of the fundamental). The energy decay factor (0.996 in this case) models the slight dissipation in energy as the wave makes a roundtrip through the string.

- The averaging operation serves as a gentle low-pass filter (which removes higher frequencies while allowing lower frequencies to pass, hence the name). Because it is in the path of the feedback, this has the effect of gradually attenuating the higher harmonics while keeping the lower ones, which corresponds closely with how actually plucked strings sound.

Setup

Download all starter files from this link:

`http://penoy.admu.edu.ph/~guadalupe154884/classes/csci30/guitar_files.zip`

You will need to set up a Python *virtual environment*, which allows us to install Python packages without messing up the system Python installation. To do this, run the command `python -m venv guitarenv`. This will produce a folder called `guitarenv`. Now your project has its own virtual environment. Generally, before you start using it, you'll first activate the environment by executing a script that comes with the installation:

- In Linux/macOS, run `source guitarenv/bin/activate`.
- In Windows, run `guitarenv\Scripts\activate`.

Once you can see the name of your virtual environment (`guitarenv` in this case) in your terminal, then you know that your virtual environment is active.

Next, install the packages used by this project by running `pip install -r requirements.txt`. This file lists the packages, and more importantly, their respective version numbers, which ensures consistency.

Verify that the packages have been installed correctly by running the command `pip freeze`. It should look something like this:

```
brian@tinolang-manok ~/classes/csci30/projects/guitar
(guitarenv) % pip freeze
numpy==1.23.2
pygame==2.1.2
```

> **Important!** If you are having trouble setting up the virtual environment, please don't proceed further. Don't hesitate to ask for help!

Provided as part of the starter files is the `stdaudio` library, which serves as a convenient wrapper for low-level audio synthesis functions from PyGame. To test whether the `stdaudio` library works in your machine, you can run `python stdaudio.py` in the terminal (and additionally, you may want to turn down the volume first since it may be loud). You should hear a short tune being played.

> **Important!** The `stdaudio` library has been tested to work with Linux and Windows, but since I don't have access to a macOS machine, please let me know right away if you encounter any problems with this library.

Once you are done using the virtual environment, you can deactivate it by running the `deactivate` command. After executing the `deactivate` command, your terminal returns to normal. This change means that you've exited your virtual environment. If you interact with Python or `pip` now, you'll interact with your globally configured Python environment.

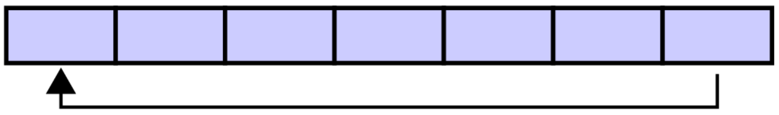
If you want to go back into a virtual environment that you've created before, you again need to run the `activate` script of that virtual environment.

Part 1: Ring buffer

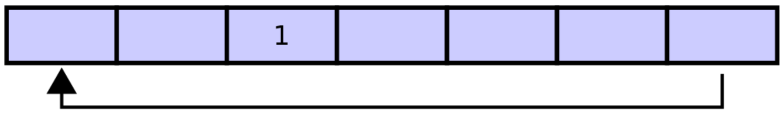
Your first task is to create a data type to model the ring buffer. Write a class named `RingBuffer` that implements the following interface:

```python
class RingBuffer:
    def __init__(self, capacity: int):
        # create an empty ring buffer, with given max capacity
    def size(self) -> int:
        # return number of items currently in the buffer
    def is_empty(self) -> bool:
        # is the buffer empty (size equals zero)?
    def is_full(self) -> bool:
        # is the buffer full (size equals capacity)?
    def enqueue(self, x: float):
        # add item x to the end
    def dequeue(self) -> float:
        # return and remove item from the front
    def peek(self) -> float:
        # return (but do not delete) item from the front
```

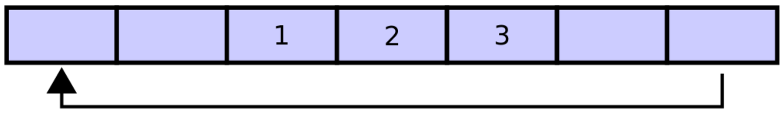
Since the ring buffer has a known maximum capacity, implement it using a fixed array of that length. For efficiency, use a *circular queue*. To see how it works, here is a 7-element buffer:



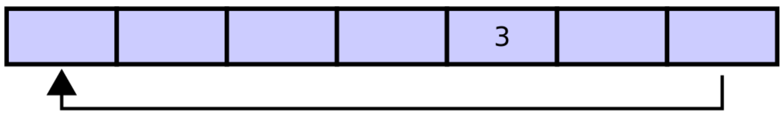
Assume that a 1 is written into the middle of the buffer (exact starting location does not matter in a ring buffer):



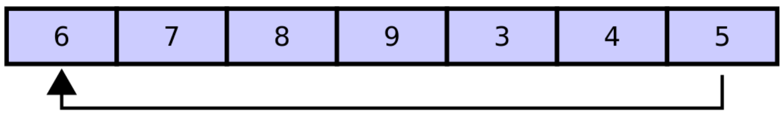
Then assume that two more elements are added (2 and 3) which get appended after the 1. Here, it is important that the 2 and 3 are placed in the exact order and places shown:



If two elements are then removed from the buffer, the oldest two values inside the buffer are removed. The two elements removed, in this case, are 1 and 2, leaving the buffer with just a 3:



If we then enqueue 4, 5, 6, 7, 8, 9, the ring buffer is now as shown below:



Note that the 6 was enqueued at the leftmost entry of the array (i.e., the buffer wraps around, like a ring). At this point, the ring buffer is full, and if another `enqueue()` is performed, then an exception will occur.

I recommend you maintain one variable `_front` that stores the index of the least recently inserted item. Maintain a second variable `_rear` that stores the index one beyond the most recently inserted item. To insert an item, put it at index `_rear` and increment `_rear`. To remove an item, take it from index `_front` and increment `_front`. When either index equals `capacity`, make it wrap-around by changing the index to 0. You're welcome to do something else if you'd like, since my tester doesn't explictly check for these variables and it will not be able to see them anyway.

Implement `RingBuffer` to raise an exception if the client attempts to `dequeue()` or `peek()` from an empty buffer or `enqueue()` into a full buffer.

A tester (which is a proper subset of my actual tester) will be soon provided to test your ring buffer implementation.

Part 2: Guitar string

Next, create a data type to model a vibrating guitar string, which uses `RingBuffer` to replicate the sound of a plucked string. Write a class named `GuitarString` that implements the following interface:

```
class GuitarString:
    def __init__(self, frequency: float):
        # create a guitar string of the given frequency,
        #   using a sampling rate of 24000 Hz
    def make_from_array(self, init: list[int]):
        # create a guitar string whose size and initial values are
        #   given by the array
    def pluck(self):
        # set the buffer to white noise
    def tick(self):
        # advance the simulation one time step
    def sample(self) -> float:
        # return the current sample
    def time(self) -> int:
        # return the number of ticks so far
```

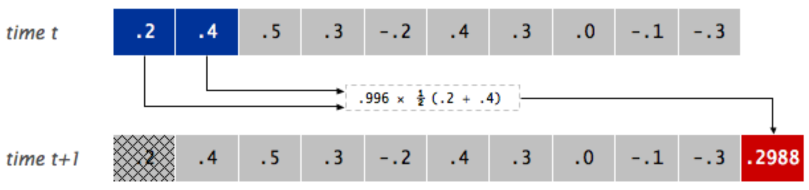
The constructor creates a `RingBuffer` of the desired capacity N (sampling rate 24000 Hz divided by frequency, rounded up to the nearest integer), and initializes it to represent a guitar string at rest by enqueueing N zeros.

The `make_from_array(init)` method creates a `RingBuffer` of capacity equal to the size of the array `init`, and initializes the contents of the buffer to the values in the array `init`. For this project, its main purpose is for debugging and grading; you won't actually use this yourself.

The `pluck()` method sets the buffer to white noise by replacing the N items in the ring buffer with N random values from $-\frac{1}{2}$ to $+\frac{1}{2}$. You may use the built-in `random.uniform()` function to generate random real numbers for this purpose.

The `tick()` method applies the Karplus–Strong update by:

1. deleting the sample at the front of the ring buffer, and
2. adding to the end of the ring buffer the average of the first two samples, multiplied by the energy decay factor.



the Karplus-Strong update

The `sample()` method simply returns the value of the item at the front of the ring buffer.

The `time()` method simply returns the number of times `tick()` was called.

A tester (which is a proper subset of my actual tester) will be soon provided to test your `GuitarString` implementation.

Part 3: Guitar player

Also provided is `guitarlite.py`, a sample client that plays the guitar in real-time, using the keyboard to input notes. When the user types the lowercase letter a or c, the program plucks the corresponding string. Since the combined result of several sound waves is the superposition of the individual sound waves, we play the sum of all string samples.

Write a program `guitar.py` that is similar to `guitarlite.py`, but supports a total of 16 notes on the chromatic scale from 220 Hz to about 523 Hz. In general, make the ith character of the string below play the ith note.

```
keyboard = "q2we4r5ty7u8i9op"
```

This keyboard arrangement imitates a piano keyboard: the "white keys" are on the `qwerty` row and the "black keys" on the `12345` row of the keyboard.

The *i*th character of the string corresponds to a frequency of $440 \times 1.059463^{i-12}$ Hz,[1] so that the character `q` is close to 220 Hz, `i` is close to 440 Hz, and `p` is approximately 523 Hz.

Please don't even think of including 16 individual `GuitarString` variables or a 16-way if statement! Instead, create a list of 16 `GuitarString` objects and use `keyboard.index(key)` (or something similar) to figure out which key was typed. Make sure your program does not crash if a key is played that is not one of your 16 notes.

Submission

Submit the following files:

- The Python source files `ringbuffer.py`, `guitarstring.py`, `guitar.py`
- A fully-accomplished Certificate of Authorship (use whichever version is appropriate), with the filename `guitar-[ID1]-coa.pdf` or `guitar-[ID1]-[ID2]-coa.pdf` (without the brackets)
- **Do not** include the Python virtual environment folder and the __pycache__ folder

Compress these files into a zip archive (`.zip`) or a gzipped tarball (`.tar.gz`) with the filename `guitar-[ID1]` or `guitar-[ID1]-[ID2]` (without the brackets) with the appropriate file extension, and submit it in the Canvas submission module for this project.

The midterm project is due on **October 10 (Monday) at 6 PM.** See the section below for the late submission policy.

Project defense

Upon submission, you will need to sign up for a project defense. Project defense days are tentatively scheduled from **October 10 (Monday)** to **October 13 (Thursday)**; the last day will be online-only, while the others will be done onsite.

More details about the logistics of the project defense will be announced at least a week before.

Grading

Your midterm project is graded using the following criteria:

[1] In case you're wondering, 1.059463 is just the twelfth root of 2. This number represents the frequency ratio of a semitone in twelve-tone equal temperament.

- 30 points — `RingBuffer` correctness/efficiency
- 30 points — `GuitarString` correctness/efficiency
- 20 points — `guitar.py` functionality
- 20 points — project defense

The last two criteria are graded **subtractively.** That means, the default score is 20 points if there are no problems, but this will be subject to deduction for any issues/errors found.

Extra credit opportunity

Bring your laptop to class on **October 14 (Friday)** and perform a piece for your classmates. Partners may perform a duet and both will receive extra credit, or a solo for individual extra credit.

Other matters

Groupings

You may work alone or you may work with a partner (this can be different from your pset partners). Your partner can come from a different section.

Despite this, a peer grading system will **not** be implemented. That means, with rare exceptions, both partners will receive the same grade even if partnerships ended up being somewhat inequitable. Please let me know immediately if your partnership went really off the rails.

Late submission policy

Late submissions are allowed but strongly discouraged. Submissions made after the due date will be penalized with an 85% cap on the grade. You will not be allowed to sign up for a slot for the project defense until you have submitted.

Rescheduling a project defense

In situations where you or your partner (or both of you) became suddenly unavailable for your chosen time slot due to extenuating circumstances, you are advised to inform your instructor immediately so that necessary adjustments can be made. In most cases, you will be asked to reschedule your project defense.

You have at most 48 hours to sign up for another slot if asked to reschedule. Failure to reschedule your project defense may lead to a grade of zero for the project defense component of your grade.

Any questions?

Any questions, clarifications, and concerns about the midterm project specs should be directed to the `#general` channel on the Discord server, or via email or Canvas.

Changelog

- Sep 17: Initial release