Before you begin

- Review your knowledge of Fourier transforms. Here's a good place to start if you don't remember. Long story short, all signals can be approximated by sine waves (and if they are discrete signals, you can get a perfect reconstruction).
- Remind yourself of what the term Nyquist frequency means in the context of sampled signals. Long story short, the higher your sampling rate, the higher ranges of frequencies you can record in your signal.
- Download the `sinewave1000hz.wav` and `starwars.wav` files on Canvas.

Learning Objectives:

The goal of this homework is to have you write an application that does something useful. In this case, we're having you work with digital signal processing, but we could just as easily have you working with applications like dynamic system simulation, etc. We just chose music files because, well, who doesn't like music?

Thoughts, (come back after you have read the problems):

- There are two parts to this assignment. The first part contains the logic for the digital signal processing. The second involves the GUI. The idea is that you can write the digital signal code and then import it inside of your GUI. Even though we tell you to write the digital signal stuff first, you should feel free to write methods that may help you out for the GUI.
- Remember that if you're copy-pasting code, there's usually a way you can refactor the code so that it's reusable. For instance, if you find that while making your GUI you're copy and pasting 5-line chunks of code multiple times, there's probably a way to condense those 5 lines into a single reusable command, which will make your life easier if you find out you did something wrong and have to go back and edit each of the chunks again. You may also find it useful to reuse some of the code in Lab 8.
- You should sanity check your `bandpass()` function by running it on the provided sine wave file, which is 1000 Hz, and saving it back out. If 1000 Hz is in your frequency range, the signal should sound identical. If it's outside, you should hear silence.

Grading Checkpoints (20 points + 3 extra credit + 4 standard)

Please see associated footnotes for each item * .

- **Digital signal (10 pts)**: (Autograder will give immediate feedback)
  - [2 points] Class properly initializes and sets the initial attributes correctly. *
  - [1 points] `DigitalSignal.from_wav()` works. *
  - [1 points] `bandpass()` sets `self.freq_low` and `self.freq_high` correctly *
  - [2 points] `bandpass()` sets `self.filtered_data` to the correctly transformed signal *
  - [1 points] `bandpass()` has a consistent datatype as the input *
  - [2 points] `subset_signal()` works. *
  - [1 points] `save_wav()` works. *
- **GUI (10 points, but can earn up to 13)**: (Manual feedback)
  - [1 points] We can load in a valid file.
  - [2 points] Your code does not emit any exceptions or crash during the course of normal operation.
  - [7-10 points] Some combination of the additional features listed at the bottom of the assignment. You must complete 7 points minimum, can earn up to 10 points (3 points extra credit)
- **Standard Grading Checkpoints (3.5 points total)**

*Indicates that the autograder will tell you if this problem is correct on our set of test cases and assign you credit.

- [2 points] Code passes PEP8 checks with 10 errors max.
- [2 points] Code passes TA-reviewed style checks for cleanliness, layout, readability, etc.

Hand in your `digital_signal.py` and `audio_gui.py` file to Gradescope.

Problem 1 Signal Processing Class

Create a new file called `digital_signal.py`. All your code from Problem 1 and 2 should go in there.

The first part of this assignment will not involve the GUI at all; however, it will provide you with some useful interfaces to which you can connect the GUI actions.

- Write a class called `DigitalSignal` whose `__init__` method takes in two arguments: a 1-D integer Numpy array and a positive integer representing the sampling frequency. It should create the following attributes (as well as any others you find useful):
  - `sampling_frequency`: The passed-in sampling frequency.
  - `source_data`: The original Numpy array which was passed in.
  - `filtered_data`: The same as `source_data`, but as a copy (do `your_array.copy()`)
  - `freq_low`: Initialized to 0
  - `freq_high`: Initialized to the maximum frequency present in the signal, i.e. Nyquist frequency
- Write a class method called `from_wav()` which takes in a single argument, a file name string. It should return a `DigitalSignal` with the data and sampling rate from the WAV file. You may assume that the WAV file is mono (so there will only be a single channel, returned by scipy's `wavfile read` function as a 1-D array of some length). However, for your own sake, if you want to work with stereo, you should extract simply extract the first channel to work with (we will not grade you on this).
  Note that with a class method, we should be able to call your code like this:

  ```
  my_signal = DigitalSignal.from_wav('music.wav')
  ```

  Be sure that you use the passed in `cls` variable to initialize the `DigitalSignal` instance. (The autograder will tell you if you're doing it correctly.)

Problem 2 Signal Processing

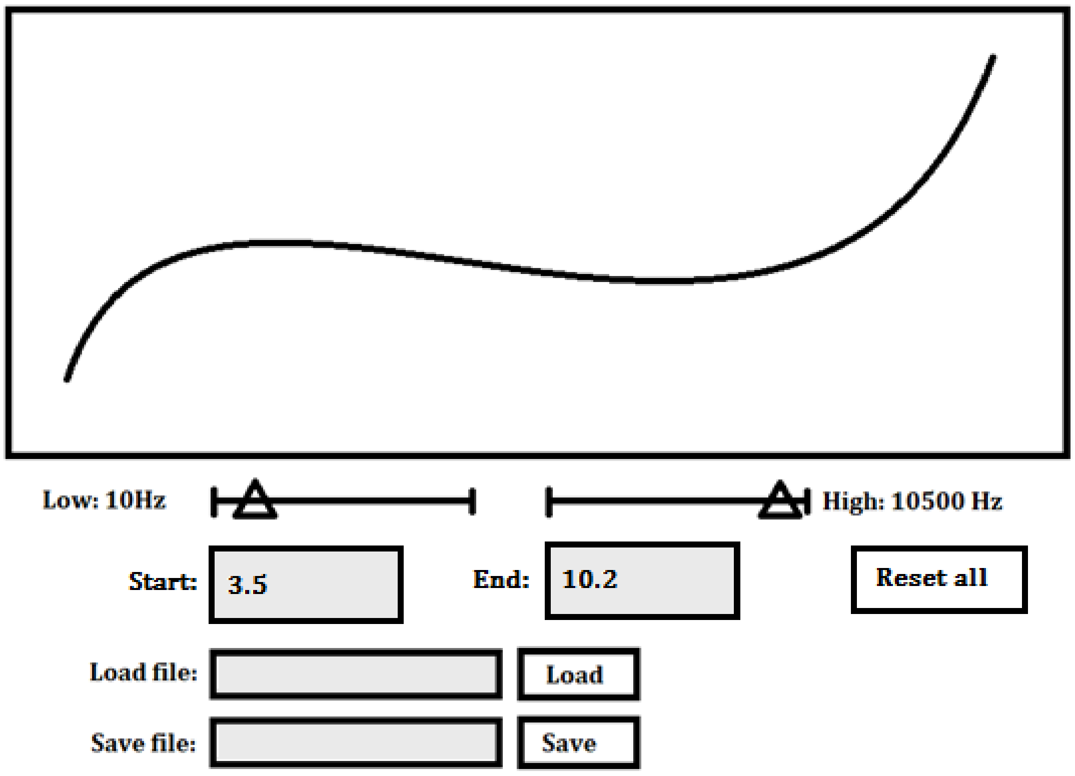
- Write an instance method called `bandpass()` which takes in 2 **optional** arguments, `low` and `high`, representing the bounds of the frequency ranges to leave in. If not specified, `low` should default to 0, and `high` should default to the Nyquist frequency.
- It should first take the entire audio signal and compute its Fourier transform using `scipy.fft.rfft`.
- It should then use the `scipy.fft.rfftfreq` function. This function produces an array which, when given the number of samples in the original signal and the sampling *period*, produces an array which contains the frequency value associated with each element of the Fourier transform.
- You should then modify the Fourier transform so that any value which corresponds with a frequency component strictly less than `low` and strictly greater than `high` is set to 0.
- Finally, you should compute the inverse Fourier transform of the modified Fourier transform with `scipy.fft.irfft` and convert the output back to the same datatype as self.`source_data`.
- You should then do the following:
  - Set `self.filtered_data` equal to the newly-transformed data.
  - Set `self.freq_low` and `self.freq_high` to the corresponding `low` and `high` frequencies.

Note: For consistency, please use the `fft`-related functions **Scipy's `fft`** module shown above, **NOT Numpy's FFT** functions.

- Write an instance method called `subset_signal()`, which takes in two optional numeric arguments, `start` and `end`. If the user does not specify them, `start` should default to 0, and `end` should default to the length of the audio signal in seconds.
- It should take the data currently in `self.filtered_data` and subset the signal to only those values whose time values are greater than or equal to `start` and less than or equal to `end`. The first sample in the audio signal is associated with `t=0`, the second sample `t=1/samplingRate`, the third `t=2/samplingRate`, etc.
- **Hint**: This part is most easily done using `np.arrange` and truth indexing.
- Write an instance method called `save_wav()` which takes in one required argument, a file name, and 2 optional arguments, `start` and `end` similar to in `subset_signal()`. It should write out the currently saved filtered audio signal with the corresponding time bounds to the filename.

Problem 3: GUI

Create a new file called `audio_gui.py`. When you submit your code Gradescope, **make sure to submit all files necessary to run your GUI.** In this section, your goal is to implement a GUI which provides easy access to the features of the class. It will look something like the prototype below:



It does not need to look exactly like the above. It must have a `Load` button which reads user input from a form field and loads the file in. It also should not crash or emit any exceptions during the normal course of operation.

From there, **you may choose the following features to implement**. You only need to do 7 points worth of them to get full credit, but you can earn up to 10 points (i.e. 3 extra credit points). (Note that there are 12 points available total; if you do all 12, you will only get 10, but you can pat yourself on the back for a job well done.)

- [1 points] A plot showing the audio signal of the loaded file
- [2 points] Two sliders representing the low and the high frequencies of the signal. The min and the max should be set to 0 and the Nyquist frequency respectively. These values should update each time we load a new file in. Each slider should have a label showing the current value of the slider.

- [2 points] Two text inputs (i.e. `QLineEdit` objects) for the start and end of the audio signal to be displayed. When we load a new file in, the values in these fields should be set to 0 and the maximum time respectively.
  Also note that your code should make sure **not to throw an exception** if the user types in something that's not a number into the field. You can decide what the correct behavior for the code is if this happens, as long as it doesn't emit an `Exception`.
- [2 points] The plot updates when the sliders are moved and the text inputs are edited. For the `QLineEdit`, the trigger you want is `.editingFinished`
  Note: The graph may update slowly when changing the frequency; this is OK. Changes in the time bounds should be near-instantaneous.
- [2 points] A `reset all` button which automatically sets the sliders and text inputs back to their default values for the current audio file.
- [1 points] A `save file` field and button which saves the audio file to the desired name with the values from the sliders.
- [2 points] The frequency sliders cannot slide past each other, i.e. if the low bound exceeds the upper bound, the upper bound gets "pushed" along, and vice versa.