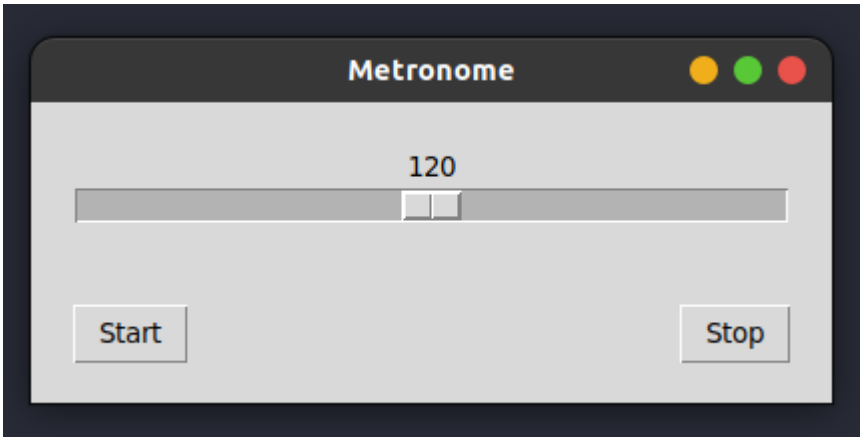
Adit Jain, 20038

DYNAMIC METRONOME GENERATOR

I started by making a basic window in python using the *tkinter* library. It provides a minimalist framework with which one can create modular apps. In this I added a slider that goes from the value of 40 to 200, essentially representing the tempo, a.k.a., the BPM (beats per minute). I also added two other buttons, namely the “Start” and “Stop” button that works as you think it does. Here’s a screenshot of it:



With the GUI out of the way, I focused on the core aspect of the application, i.e. the metronome generation. In order to do so, I declared a Metronome class, employing the use of the in-built *time* and *threading* libraries. The Metronome class has the following 5 functions, as can be seen in the image on the right. A brief overview of the functions are as follows:

__init__: Initialises the class with the default values. (Constructor)
change_tempo: Changes the tempo to the input value. It calculates the interval accordingly. The slider takes in this as its call-back function, providing us with the user updated value of the tempo.
beat: Responsible for actually playing the sound (printing ‘Tick’ here).
start: Self explanatory, mapped to the start button of the GUI.
stop: Self explanatory, mapped to the stop button of the GUI.

```
class Metronome:
  def __init__(self):
    self.tempo = 120
    self.interval = 60 / self.tempo
    self.running = False
    self.thread = None

  def change_tempo(self, new_tempo):
    self.tempo = int(new_tempo)
    self.interval = 60 / self.tempo

  def beat(self):
    if self.running:
      print('[METRONOME] Tick')
      time.sleep(self.interval)
      self.beat()

  def start(self):
    if not self.running:
      print('[METRONOME] Start')
      self.running = True
      self.thread = threading.Thread(target=self.beat)
      self.thread.start()

  def stop(self):
    if self.running:
      print('[METRONOME] Stop')
      self.running = False
      self.thread.join()
      self.thread = None
```

REAL TIME NOTE DETECTOR

In order to achieve real time note detection, I won't be using any GUI, as it is not required. Rather, I would spend more time on making my code readable and fine-tuned for utmost precision. The note being played can simply be displayed on the terminal. A high-level overview of my approach (yet to be implemented) would be as follows:

- **Audio Input:** For capturing real-time audio, I explored a lot of libraries and settled on *pyaudio*, a library with easy to understand and implement functions.
- **Pre-processing:** The audio captured will most likely have noise and certain artifacts that may affect the accuracy of note detection. Preprocessing techniques such as filtering and normalization that are widely made use of in signal processing and available in the *scipy* library of python can be made use of to reduce the noise as much as possible, assuming I'm calibrating my program to listen for only certain frequency range, such as 200*Hz* - 1000*Hz*, in which case a band-pass filter is nice.
- **Pitch Detection:** Music notes are decided by the pitch of the sound they make. I need to identify the primary frequency component of the available sound signal. The first approach that comes to mind is Fast Fourier Transform (FFT) to convert the time-series signal to a signal in the frequency domain. The peak frequency here should theoretically correspond to the pitch of the note being played. But in popular words, theory can only take you so far. So a lot of testing and fine-tuning will need to be done in order to accurately judge the actual peak.
- **Frequency to Note:** To get the node from the frequency could be as simple and inefficient as a look-up table, or we could use a theoretical formule, to find out both the note and the octave (using A4 at 440*Hz*) as follows:

```python
A4_FREQ = 440.0
SEMI_TONE_RATIO = 2 ** (1/12)
NOTES = ['C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A', 'A#', 'B']

def get_note(frequency):
  num_semitones = 12 * np.log2(frequency / A4_FREQ)
  octave = 4 + int(num_semitones / 12)
  note_index = int(round(num_semitones)) % 12
  note = NOTES[note_index]
  return note, octave
```

- **Real Time implementation:** To make all this process work in real-time, what I thought of doing is using buffer techniques to process chunks of audio data at a time, thereby ensuring an accurate trade-off between performance and accuracy. I feel that a buffer time of as low as 100*ms* should work fine, but I'll decide upon the actual figure post implementation and rigorous testing.