```
    # CapTouch.py
# Starter code taken from https://learn.adafruit.com/adafruit-mpr121-12-key-capacitive-touch-sensor-breakout-tutorial/python-circuitpython

import time
import board
import busio
# Import MPR121 module.
import adafruit_mpr121

from gpiozero import LED, LEDBoard

# Create I2C bus.
i2c = busio.I2C(board.SCL, board.SDA)

# Create MPR121 class.
mpr121 = adafruit_mpr121.MPR121(i2c)

# we will output touches on GPIO Pins of the Pi
TOUCH_LEFT = 12
TOUCH_DOWN = 16
TOUCH_UP = 20
TOUCH_RIGHT = 21

touchOut = LEDBoard(TOUCH_LEFT, TOUCH_DOWN, TOUCH_UP, TOUCH_RIGHT)

# Loop forever testing each input and printing when they're touched.
while True:
        print(mpr121[0].value, mpr121[2].value, mpr121[4].value, mpr121[6].value)
        touchOut.value = (mpr121[0].value, mpr121[2].value, mpr121[4].value, mpr121[6].value)


    # sendSteps.py
from  gpiozero import LED, LEDBoard
from time import *
from omxplayer.player import OMXPlayer
from pathlib import Path
from OneThing import *

STEP3 = 19
STEP2 = 13
STEP1 = 6
STEP0 = 5
BPM_CLK = 26
RESET_PIN = 17

stepLeds = LEDBoard(STEP3, STEP2, STEP1, STEP0)
bpmClockLed = LED(BPM_CLK)
resetLed = LED(RESET_PIN)

def sendSteps():
        # pulse reset signal before sending over steps
        resetLed.on()
        resetLed.off()
        # calculate sleep amount outside loop
        sleepAmount = 30/OneThingBPM
        # asynchronously play audio
        player = OMXPlayer(Path(OneThingMP3Path))
        # send over steps
        for step in OneThingSteps:
                start = time()
                stepLeds.value = tuple(step)
                # pulse BPM clock
                bpmClockLed.on()
                end = time()
                # sleep appropriately
                sleep(sleepAmount - (end - start))
                start = time()
                bpmClockLed.off()
                end = time()
                sleep(sleepAmount - (end - start))
        # make sure steps don't persist on led matrix
        stepLeds.value = tuple([0]*4)


    # OneThing.py (Shortened for Purposes of Example)
OneThingBPM = 127.020
OneThingMP3Path = 'OneThing.mp3'
OneThingSteps = [
[0,1,0,0],
[0,0,0,0],
[0,0,1,0],
[0,0,0,0],
[0,1,0,0]]
```

```
---SYSTEMVERILOG CODE---
//Top-level module - DDR.sv
module DDR(input logic clk,
                        input logic bpmClk,
                        input logic reset,
                        input logic [3:0] step,
                        input logic [3:0] button,
                        output logic [3:0] colEnOut,
                        output logic [7:0] col,
                        output logic redLed,
                        output logic greenLed,
                        output logic [6:0] sevenSegDisplay,
                        output logic leftDigitTransBase,
                        output logic rightDigitTransBase);

        logic [15:0] counter;
        logic multiplexClk;
        logic [3:0] actionStep;
        logic [3:0] colEn;
        logic stepEn;

        always_ff@(posedge clk)
                if (reset) counter <= 0;
                else counter <= counter + 1;

        assign multiplexClk = counter[15];

        levelToPulse l2p(clk, reset, bpmClk, stepEn);
        colMuxer colMux(multiplexClk, reset, colEn);
        stepShiftRegister stepReg(clk, reset, stepEn, colEn, step, col, actionStep);

        assign colEnOut = ~colEn;

        logic [8:0] score;
        scoring scoring0(clk,reset,actionStep,stepEn,button,bpmClk,score,redLed,greenLed);

        scoreDisplay scoreDisplay0(clk,reset,score[8:0],leftDigitTransBase,rightDigitTransBase,sevenSegDisplay);

endmodule


    // scoring.sv - Scoring algorithm determines if good/bad press and updates score and provides immediate feedback via green and red LEDs
module scoring(input logic clk,
                    input logic reset,
                    input logic [3:0] step,
                    input logic beatEn,
                    input logic [3:0] button,
                                                input logic bpmClk,
                    output logic [8:0] score,
                    output logic redLed,
                    output logic greenLed);

    //INPUTS AND TIME CAPTURE
    logic [3:0] buttonPulse;
    levelToPulse l2p_0(clk, reset, button[0], buttonPulse[0]);
    levelToPulse l2p_1(clk, reset, button[1], buttonPulse[1]);
    levelToPulse l2p_2(clk, reset, button[2], buttonPulse[2]);
    levelToPulse l2p_3(clk, reset, button[3], buttonPulse[3]);

    logic [63:0] currentTime;
    logic [63:0] beatTime;
    always_ff @(posedge clk) begin
        if (reset) begin
            currentTime <= 0;
            beatTime <= 0;
        end
        else begin
            currentTime <= currentTime + 1;
            if (beatEn)
                beatTime <= currentTime;
        end
    end

    logic beatReset;
    //Beat is reset halfway between beatEnables
    levelToPulse l2p_4(clk, reset, ~bpmClk, beatReset);

    logic [3:0] buttonPressedDuringStep;
    always_ff @(posedge clk) begin
        if (reset || beatReset)
            buttonPressedDuringStep <= 0;
        else if (buttonPulse[0])
            buttonPressedDuringStep[0] <= 1;
```

```
        else if (buttonPulse[1])
            buttonPressedDuringStep[1] <= 1;
        else if (buttonPulse[2])
            buttonPressedDuringStep[2] <= 1;
        else if (buttonPulse[3])
            buttonPressedDuringStep[3] <= 1;
    end

    logic goodStep;
    assign goodStep = (step == buttonPressedDuringStep) && (step != 4'b0000);

    //CALCULATE SCORE
    logic addGoodStepToScore;
    logic deltaScore;
    always_ff @(posedge clk) begin
        if (reset) begin
            deltaScore <= 0;
            score <= 0;
        end
        else if (beatReset || deltaScore) begin //score as soon as step equals buttonPressedDuringStep
            deltaScore <= addGoodStepToScore;
            score <= score + deltaScore;
        end
    end
    assign addGoodStepToScore = goodStep && ~deltaScore;

    //LED FEEDBACK
    logic redLedEn;
    logic ledReset;
         logic stepExpected;
    always_ff @(posedge clk) begin
        if (ledReset)
            greenLed <= 0;
        else if (deltaScore)
            greenLed <= 1;
    end

    always_ff @(posedge clk) begin
        if (beatEn)
            redLed <= 0;
        else if (redLedEn)
            redLed <= 1;
    end
         assign stepExpected = (| step);
    assign redLedEn = (beatReset && !deltaScore && stepExpected && !goodStep);
    assign ledReset = (reset || beatReset);

endmodule


    // stepShiftRegister.sv
module stepShiftRegister(input logic clk,
                                                                input logic reset,
                                                                input logic stepEn,
                                                                input logic [3:0] colEn,
                                                                input logic [3:0] inputStep,
                                                                output logic [7:0] col,
                                                                output logic [3:0] actionStep);

        logic [3:0] step0, step1, step2, step3, step4, step5, step6, step7;
        // shift register to shift steps every beat
        always_ff@(posedge clk)
                if (reset) begin
                                step0 <= 4'h0;
                                step1 <= 4'h0;
                                step2 <= 4'h0;
                                step3 <= 4'h0;
                                step4 <= 4'h0;
                                step5 <= 4'h0;
                                step6 <= 4'h0;
                                step7 <= 4'h0;
                        end
                else if (stepEn) begin
                        step0 <= inputStep;
                        step1 <= step0;
                        step2 <= step1;
                        step3 <= step2;
                        step4 <= step3;
                        step5 <= step4;
                        step6 <= step5;
                        step7 <= step6;
                end

        assign actionStep = step7;
```

```
        always_comb
                case (colEn)
                        4'b0001: col = {~step0[3], ~step1[3], ~step2[3], ~step3[3], ~step4[3], ~step5[3], ~step6[3], ~step7[3]};
                        4'b0010: col = {~step0[2], ~step1[2], ~step2[2], ~step3[2], ~step4[2], ~step5[2], ~step6[2], ~step7[2]};
                        4'b0100: col = {~step0[1], ~step1[1], ~step2[1], ~step3[1], ~step4[1], ~step5[1], ~step6[1], ~step7[1]};
                        4'b1000: col = {~step0[0], ~step1[0], ~step2[0], ~step3[0], ~step4[0], ~step5[0], ~step6[0], ~step7[0]};
                        default: col = {~step0[3], ~step1[3], ~step2[3], ~step3[3], ~step4[3], ~step5[3], ~step6[3], ~step7[3]};
                endcase


endmodule



    // scoreDisplay.sv
module scoreDisplay(input logic clk, reset,
                                                                input logic [8:0] score,
                                                                output logic leftDigitTransBase, rightDigitTransBase,
                                                                output logic [6:0] seg);

        //Alternate LEDs at clk/(32768)=1220 Hz
        logic [16:0] counter; //16-bit counter
        logic switch; //LED group to drive (0 or 1)
        assign switch = counter[16]; //highest bit of counter
        always_ff @(posedge clk)
                counter <= counter + 1;
        //Only one transistor base low at a time
        assign leftDigitTransBase = switch;
        assign rightDigitTransBase = ~switch;
        //Switch to 7-segment display decoder
        logic [3:0] scoreHalf;
        assign scoreHalf = (switch ? score[7:4] : score[3:0]);
        hexto7seg dispDecoder(clk,scoreHalf,seg);
endmodule

//Module that decodes hexadecimal value to 7- segment display
module hexto7seg(input logic clk, input logic [3:0] hex, output logic [6:0] seg);
        //Common logic-high anode, output is cathode value
        //Logic-low output indicates turning segment
        always_comb begin
                case (hex)
                        4'h0: seg <= 7'b1000000;
                        4'h1: seg <= 7'b1111001;
                        4'h2: seg <= 7'b0100100;
                        4'h3: seg <= 7'b0110000;
                        4'h4: seg <= 7'b0011001;
                        4'h5: seg <= 7'b0010010;
                        4'h6: seg <= 7'b0000010;
                        4'h7: seg <= 7'b1111000;
                        4'h8: seg <= 7'b0000000;
                        4'h9: seg <= 7'b0011000;
                        4'hA: seg <= 7'b0001000; //A
                        4'hB: seg <= 7'b0000011; //b
                        4'hC: seg <= 7'b1000110; //C
                        4'hD: seg <= 7'b0100001; //d
                        4'hE: seg <= 7'b0000110; //E
                        4'hF: seg <= 7'b0001110; //F
                endcase
        end
endmodule



    //levelToPulse.sv
module levelToPulse(input logic clk,
        input logic reset,
        input logic a,
        output logic en);
 typedef enum {s0, s1, s2} StateType;

 StateType state;
 StateType nextState;

 always_ff@(posedge clk, posedge reset)
  if (reset) state <= s0;
  else       state <= nextState;

 always_comb
  case(state)
   s0: nextState = a ? s1 : s0;
   s1: nextState = a ? s2 : s0;
   s2: nextState = a ? s2: s0;
  endcase
```

```
 // emit pulse for one clk cycle
 assign en = state == s1;
endmodule


    // colMuxer.sv
module colMuxer(input logic multiplexClk,
                                        input logic reset,
                                        output logic [3:0] colEn);

        logic [3:0] nextColEn;

        always_ff@(posedge multiplexClk)
                if (reset) colEn <= 4'b0001;
                else               colEn <= nextColEn;

        always_comb
                case (colEn)
                        4'b0001: nextColEn <= 4'b0010;
                        4'b0010: nextColEn <= 4'b0100;
                        4'b0100: nextColEn <= 4'b1000;
                        4'b1000: nextColEn <= 4'b0001;
                        default: nextColEn <= 4'b0001;
                endcase

endmodule
```