ESE 150 – Lab 12: User Interface

LAB 12

Today's Lab has the following objectives:

1. Build a simple Graphical User Interface (GUI) that is easy to use with minimal instructions or understanding of the underlying technology.
2. Assess usability of user interfaces.
3. Design GUI enhancements to increase usability.

Background:

IoT – Internet-of-Things – ubiquitous networking and cheap computing and communication devices is ushering in rapid deployment of the Internet-of-Things. Here, we connect things (objects) in the physical world to the Internet, giving us a common infrastructure to sense, monitor, and control physical objects. This allows us to reach out from our computers and mobile devices to interact with the world. As an example, today we will look at controlling electrical outlets over the IP network.

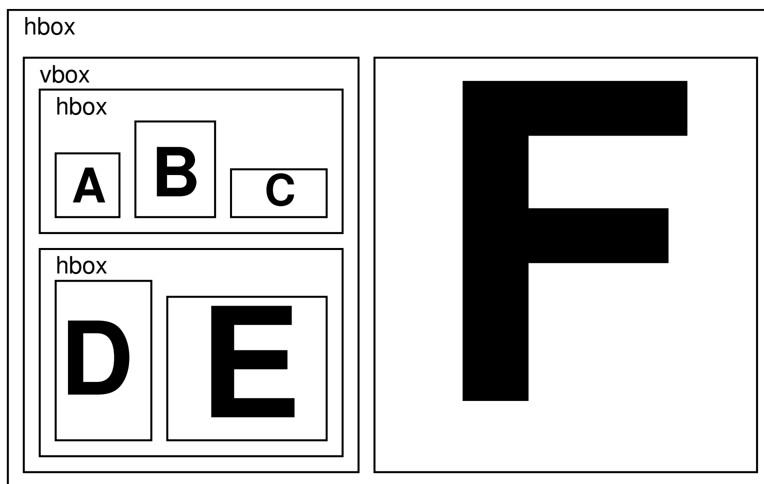
Sockets – as we saw in class and in lab last week, we can create a virtual communication channel between a processes running on different machines by identifying the machines and the port on each machine. Headers in TCP/IP packets specify the sending and receiving (IP-address, port) pairs so that hardware and software can deliver the payload contents to the appropriate process on the machine. From a software standpoint, the way we connect the process to a port is by creating a socket. The socket registers the associated port and provides a stream the process can output data to or read data from. A key part of setting up network communication for a process is to create a socket associated with a particular port.

Concatenate (cat) – cat is the unix command-line tool for combining (concatenating) the output from several files and sending them somewhere. Often this somewhere is the terminal. So, one way to read a file on your computer screen is to simply cat the file. The output of cat can be sent to another unix program using the pipe construct that we saw in Lab 9.

Netcat (nc) – nc is a unix command-line tool that can send and receive data over the network. The "cat" part is by analogy with the cat routine above. Instead of sending data to the console, it can send its data over a network connection. As such, it takes arguments for the IP address of the destination host (when sending) and the port on the destination host. It can also listen on a port, in which case it specifies the port that it should receive data from. Rather than reading from files, it typically takes its input from (and produces its output to) another program using the pipe construct.

GUI: callback – a common idiom in GUI programming is to provide a *callback* routine that should be invoked whenever there is user input. This is an answer to the question: what should happen when a button is pressed? The callback routine tells the GUI which function to call. The function packages up the operations that should be performed then the GUI element receives input. Sometimes the callback routine will have an argument so the GUI can tell the function what input the user provided. You will see your first example of a button with callback in the Prelab.

GUI: Graphical composition – A common idiom when controlling the geometric placement of scalable graphical entities is to organize them into a hierarchy of boxes. That is, at the lowest level, we have some graphical item, like a letter, a paragraph of text, an image, or a button. Each of those entities has its own box size (number of vertical and horizontal pixels). The question is how do we layout these things next to each other? In the simplest form, we might put them one after another like text characters on a line. But, as we deal with images and buttons, we often want to be more deliberately about what's placed above/below other things. Starting with every primitive graphical item having some box size, we can then create larger boxes by putting other boxes side-by-side (vertical or vbox composition) or over-under (horizonal or hbox composition). The resulting box from a vbox or hbox composition is, itself, a box that can be used in another vbox or hbox composition. So, the following allows me to put A B and C side by side and over D and E then beside F.



Python – Python is a dynamically-typed, interpreted, high-level programming language. After C and MATLAB (and Java from CIS110), hopefully you're getting more and more comfortable with picking up and modifying code in different languages. There are many languages in use today, and new ones will continue to be invented that make specific things easier to do. So, you will find it useful to continue to pickup new languages throughout your career. We use Python for this lab because it is free and open-source and has some convenient and lightweight interface to GUI toolkits that will allow us to build useful interfaces with small amounts of new code. You will find similar GUI toolkits in most languages (including Java and MATLAB).

User Interface (UI) Assessment Rubric – in class we discussed many desirable properties of good user interfaces. For the purpose of this lab, we'll focus on a specific, somewhat simplified rubric for assessing the UIs we use or generate. This has 4 components:

1. User time -- How much time does it take for the user to accomplish the desired task? Saving the user time is generally good. This can be measured with a stop-watch. You may want to define a particular benchmark task that you can time. Steve Jobs used to motivate his designers by

noting how many users they had and how often each user performed an operation and then computing number of lifetimes that could be saved if specific operations could be completed some number of seconds faster.

2. Cognitive load – How hard does the user have to think in order to perform the task? There is probably some correlation here to user time, but they can diverge depending on the UI. If the user needs to perform computations in their head or remember detailed facts correctly, that can lead to a high cognitive load. Here you can note what things a user needs to know and may need to figure out on their own.
3. Error prone – How likely is it that the user can make an error in interacting with the UI? A good UI will prevent users form making errors. In a more elaborate rubric we might break out how bad errors can be, how easy it is for users to understand what went wrong and how to recover, and how easy it is to recover form errors if they are possible. Here you might note what errors the user could make and perhaps identify what fraction of potential inputs from users would lead to errors.
4. Self describing – How much is the interface clear and usable without instruction? In general, we would be concerned with several things like: how long it takes to learn an interface, how often one needs to refer back to instructions, how easy it is to determine the right way to use the interface. The ultimately goal might be to have something that was immediately usable with no instruction and provided enough guidance so that anyone could learn how to use it quickly just by using the UI. Since our task is simple, we'll try to rate it in terms of how close it comes to this ultimate, self-describing goal. Here you can note how the UI might fall short of being self describing for a wide-range of audiences.

Prelab:

PreLab – Section 1: GUI Tools (optional – but step 2 is necessary if you want to run an app on your iPhone)

- Collect the software you need to build and run the GUIs on your laptop.
- You can always run the python GUI tools on the computers in Detkin and Ketterer, so step 1 is only if you want to be able to run them on your own laptop.
- You can only download an application to your iPhone from a Mac; so step 2 is necessary if you want to run the app on your iPhone. Since we're not certain of the time it will take to (a) perform step 2 here and (b) perform the lab, we are making running the application on your iPhone optional (bonus points). We think it will be fun.
- (Make sure your laptop is plugged into the power source throughout the installation process)

1. Installing python-gtk on your Mac.
   a. First let's check if you have brew installed on your Mac. To do this run
      `$ brew ––version`
      If you get an output that shows versions of Homebrew that is installed on your Mac, then go to step d.
      Note, we show $ here to indicate the prompt form your terminal on the mac. It is not something you type.
   b. To install brew run this command:
      ```
      $ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)" < /dev/null 2> /dev/null
      ```
      When prompted, type in your password. This command takes about 5 mins with good internet speed (AirPennNet).
   c. Run `$ brew ––version` again to see if the terminal outputs the version. Once you have successfully installed brew, move to the next step. If you get errors, please ask a TA for help or post on Piazza.
   d. Run this command on the terminal to see if you have PyGTK installed –
      `$ brew list | grep gtk`
      If your system has pygtk installed you will see the terminal output this:
      ```
      gtk+
      pygtk
      ```
   e. If you don't see this, follow the next few instructions to install it. If it is already installed, please continue to step (2)
      Run the following commands:
      `$ brew install pygtk`
      This installation takes a couple of minutes. Run `$ brew list | grep gtk` again to see if you see gtk+ and pygtk installed.

2. To install PyGTK on Linux (Debian based Linux), use
   ```
   $ sudo apt-get python-gtk2
   ```
3. Installing Flutter for Mac OS (Partners have been assigned such that one person in each team has a Mac. So we advise for the partners to install Flutter SDK together on Mac. More detailed instructions can be found here - https://flutter.dev/docs/get-started/install/macos). Plan for this to take 1.5 hours. One strategy is to work on Prelab 2 while waiting for the various steps in this installation to complete.
   a. Download the .zip file from this link - https://storage.googleapis.com/flutter_infra/releases/stable/macos/flutter_macos_v1.2.1-stable.zip
      If the link doesn't work, go to - https://flutter.dev/docs/get-started/install/macos and click on .zip file to download it.

      ## System requirements

      To install and run Flutter, your development environment must meet these minimum requirements:

      - **Operating Systems**: macOS (64-bit)
      - **Disk Space**: 700 MB (does not include disk space for IDE/tools).
      - **Tools**: Flutter depends on these command-line tools being available in your environment.
        - bash
        - curl
        - git 2.x
        - mkdir
        - rm
        - unzip
        - which

      ## Get the Flutter SDK

      1. Download the following installation bundle to get the latest stable release of the Flutter SDK:

         flutter_macos_v1.2.1-stable.zip

         For other release channels, and older builds, see the SDK archive page.

   b. Open a terminal and from your home directory create a new directory using-
      ```
      $ mkdir ESE150_Lab12
      ```
   c. Use cd and open the newly created directory-
      ```
      $ cd ESE150_Lab12
      ```
   d. Extract the file to this location, if already extracted, drag – copy the file to this location
   e. Type `$ pwd` to get the current directory's full path
   f. Now type–
      ```
      $ export PATH=”$PATH:<type pwd here>/flutter/bin”
      ```
   g. Run the following commands which helps to install and deploy the app to your iPhone (brew update will take about 5 minutes)
      ```
      $ brew update
      $ brew install --HEAD usbmuxd
      $ brew link usbmuxd
      $ brew install --HEAD libimobiledevice
      $ brew install ideviceinstaller ios-deploy cocoapods
      $ pod setup
      ```
      (pod setup takes about 5-10 minutes to finish)

If any of these commands fail, run $ `brew doctor` and follow the instructions to resolve any issues.

h. Now download Xcode - https://developer.apple.com/xcode/ [This takes about half an hour on a wired network on campus. Possibly an hour on wireless or on an off-campus network. We strongly suggest connect to a wired network on campus unless you have higher off-campus bandwidth.]

i. Next let's install Xcode by typing in the terminal –

```
$ sudo xcode-select –switch
/Applications/Xcode.app/Contents/Developer
```

(above all goes on one line; just split so fits on page.)

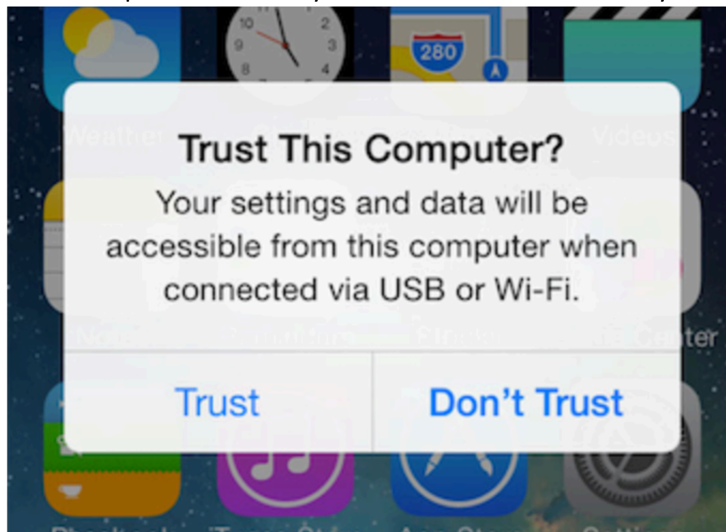
j. Next run – $ `flutter doctor`

This tells you what additional installations are required. Do as asked from the output. (Ignore the Andriod SDK installation)

k. Now we can create a sample app. First connect your iPhone to the Mac and type-

```
$ flutter create my_app
$ cd my_app
```

l. Check for permissions on your iPhone and select Trust your Computer.



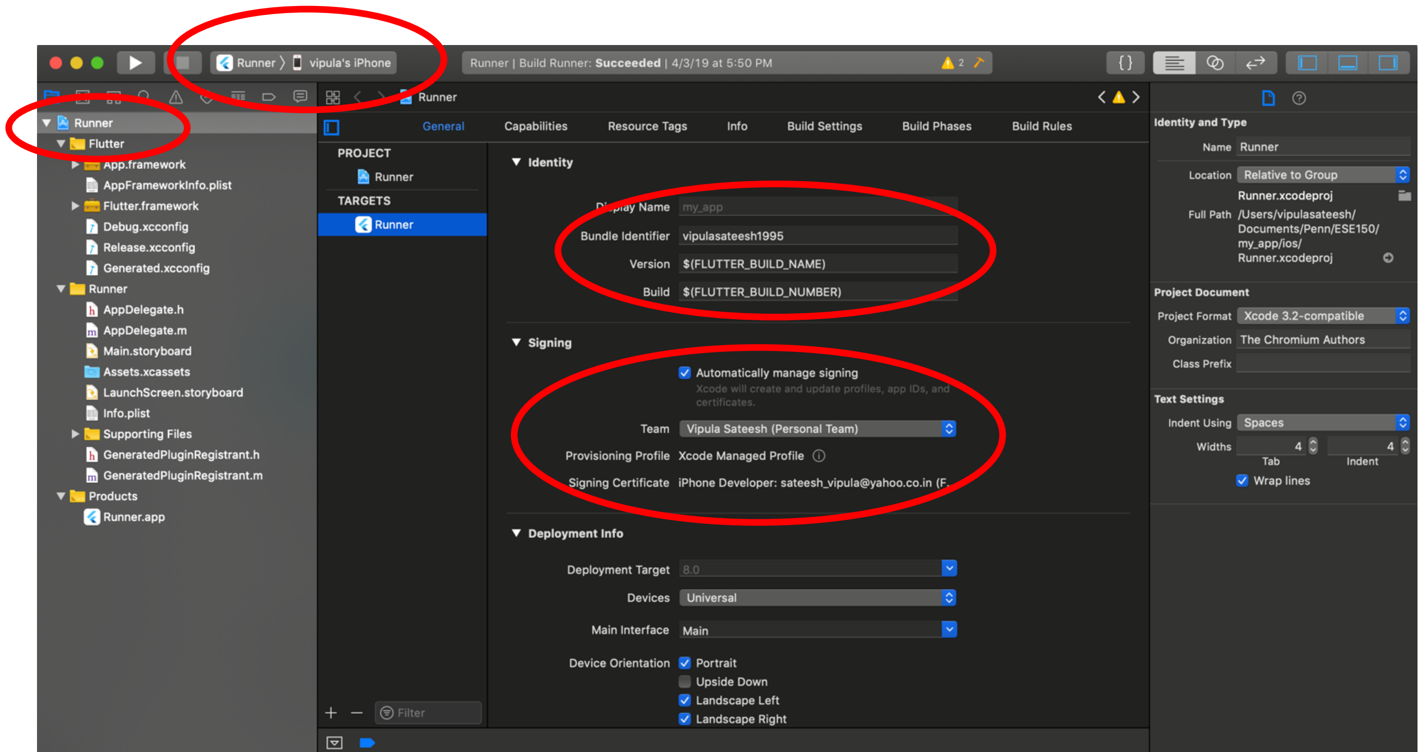
m. Now Open the Xcode Application by typing

```
$ open ios/Runner.xcworkspace
```

And select the Runner project on the left top corner. (Shown in the image below).

n. Now sign in to your Apple ID. Select your name from the Team drop down option.

o. Create a unique Bundle Identifier under Identity. Make sure your phone is selected in the top left corner. And then click on Try again under Failed to create Provisional Profile.



p. Now open your iPhone and go to Settings→ General→Scroll down to see Profiles and Device Management. Select your apple ID and give permissions.

q. Run.

```
$ flutter run
```

When prompted type your iOS password and select always allow.

If you get an error, go back to step p. and select your apple ID under Profiles and Device management.

r. Now you should be able to see an app on your iphone by the name "my_app"

s. This is a sample code that has an increment button which displays the number of the times, the button was clicked. To see the source code, type `$ cat lib/main.dart`

Congrats! You have successfully created an app on your iPhone. Click the button (+) on your phone, the app should display how many times you have clicked the button

Prelab – Section 2: Python GTK Buttons

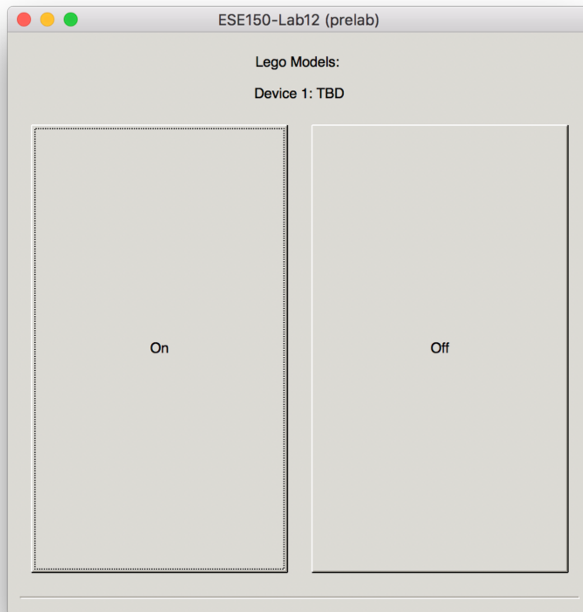
- Build and extend a simple Python GUI.

1. If you did not already create ESE150_Lab12 during the Section 1 install, open a terminal and create it now.

```
$ mkdir ESE150_Lab12
```

2. Pickup a starting UI from ~ese150/lab12/prelab_ui.py (or by downloading the support code for the lab from the course syllabus)

3. Copy it to your ESE150_Lab12 directory:
   ```
   $ cd ESE150-Lab12
   $ cp ~ese150/lab12/prelab_ui.py
   ```
   Run the python script by typing- `$ python prelab_ui.py`
   You should see a gui open up showing this :



   Now click on this two text boxes (Which are actually buttons)
   What do you see? From looking at the code, explain what is happening.
4. Modify it to add some functionality.
   a. Open the python script. And read through the code. Make sure you understand it. Also note that the you are only required to add a small section of the code which is commented #YOUR CODE GOES HERE. There will be a total of 3 line of added code.
      i. Here add a button labeled "Do not press this button".
         Use the syntax- button_name = gtk.Button("<string that should be displayed on the button>")
      ii. Add a call back function to this button which when pressed it puts up a window that says "Please, do not press this button again." ☺ Please refer to the first section for this. (Use #Syntax - button_name.connect("clicked", <call back function name>, "<Data that is sent to the function>"))
         1. For amusement value on one source of this joke, see: https://www.youtube.com/watch?v=kLC8qPNU6_0
      iii. Make this part of the hbox by using pack_start on hbox.
      iv. Test and debug UI.
   b. Change the layout so the new button is below the previous on-off buttons
      i. Copy prelab_ui.py to prelab_ui_vert.py

   ii. Change the packing command for your added button to pack on vbox_app instead of hbox
      1. This is a small change to only one line of code.
   iii. Test and debug revised UI.

5. Include your final prelab_ui.py, prelab_ui_vert.py and a screenshot of each gui in your writeup.
6. Show your GUI to your TA for prelab checkoff.

Lab Procedure:

Lab – Section 1: Routes between computers

- Establish primitive network communication with an Internet-controlled power outlet

1. Use nc (netcat) to send a string between the Detkin Linux workstation and your Linux or OS-X laptop
    a. Determine the IP address of the receiving machine
    b. On the receiving machine run: $ nc -l 47253
    c. On the sending machine run:
       $ echo "Hello World" | nc Receiving-machine-IP-address 47253
    d. [The receiving machine may need to be your laptop. It's not clear the Detkin machines will support the incoming connections.]
2. Determine the IP address for the Wemo Insight power outlet at your station.
    a. Read the mac address from the back of the outlet
    b. Use the provided table to identify the IP address assigned to that mac address
3. Copy the wemo_on.txt and wemo_off.txt files into your working directory for the lab.
       $ cp ~ese150/lab12/wemo_*.txt .
4. Turn the outlet on and off using:
    a. $ cat wemo_on.txt | nc identified-wemo-IP-address 49153
    b. $ cat wemo_off.txt | nc identified-wemo-IP-address 49153
    c. Note this is the same command you used in Step 1; you are just sending specific content to a specific port on the Wemo outlet.
    d. Make sure the outlet goes on and off based on what you send it.
5. Look at the wemo_on.txt file (use cat, more, or nano)
    a. Note how the beginning of the file looks like the beginning of the HTTP request you made last week. Identify the part that is similar and include in your report.
    b. Identify what is different between the wemo_on.txt and wemo_off.txt file. Include this in your report.
        i. It may be useful to use the diff command to get you started:
           $ diff wemo_on.txt wemo_off.txt
        ii. If you've never used diff before, see how the output of diff relates to the two files you give it. You can also see the man page for diff.
        iii. The diff will narrow down what you need to look at to one or more lines, but there is still quite a bit common between the lines. So, look further and identify what part of the lines are the same and different.
    c. Based on your identified headers and difference, describe how these network messages are is specifying that the outlet should be on or off.
6. Create a unix shell script that contains the unix command line that you used above (Step 4) to turn the device on and a second to turn the device off.
    a. Name based on the lighted structure your outlet is controlling.
    b. Use chmod -x to make the shell scripts executable.

```
$ chmod -x script-name
```

c. Test that you can turn the outlet on and off with your shell scripts.

d. Include your shell scripts in your writeup.

e. In your writeup,

    i. Describe how using the shell scripts is easier for you than using the raw unix command line? [assume you're coming back to use it 2 weeks from now.]

    ii. Describe how it is easier to use the shell script than to use the raw nc command for one of your classmates who only knows where to find the shell script and that it controls the lighted structure that goes with the name of the shell script.

Lab – Section 2: GUI

- Revise/extend your GUI to control your outlet and other outlets

  1. Copy ~ese150/lab12/single_outlet.py to your ESE150_Lab12 directory (or grab from the downloaded support code).
  2. Review the code and comments
     a. This code includes a basic on/off UI like the prelab.
     b. It also contains code to perform a similar function to the netcat (nc) unix command you used to send messages between computers and to turn on and off your outlet. See sendCommand(…).
     c. Read through the comments in the code to see how it does that.
     d. Note that it takes an IP address in the host argument.
     e. Note that it takes a string to send in the command argument.
     f. Note that it includes the port number you used earlier when it opens a connection.
     g. Note that single_outlet.py also contains string variables that hold the same contents as wemo_on.txt and wemo_off.txt that can be used as the command string argument for this sendCommand sendCallback.
  3. Revise single_outlet.py GUI to control the Wemo outlet:
     a. As provided it has two buttons, on and off, the simply pop up windows like the prelab version.
     b. Change the callBacks for these buttons so that pushing a button calls the sendCallback/sendCommand routines with suitable arguments to turn your outlet on or off
  4. Test and debug your GUI.
     a. Remember you can use nc to listen on a port (Section 1, step 1); this may be useful in debugging if you are trying to determine if your GUI is sending messages and if the messages are received as you intend them.
  5. Include your final single_outlet.py in your writeup.
  6. Copy single_outlet.py to multi_outlet.py.
  7. Revise multi_outlet.py so that it can control 3 lighted structures: yours, the one on your right (lower station number) and the one on your left (higher station number, unless you are at Station 13, then it will be Station 2).
     a. Get the IP address from the appropriate team.
     b. Coordinate with the adjacent teams for testing.
     c. Test and Debug.
     d. Include multi_outlet.py in your writeup.
  8. Copy multi_outlet.py to visual_outlet.py.
  9. Copy over the png images from ~ese150/lab12/
  10. Revise visual_outlet.py so that the it uses the images to visually denote the lighted structure controlled by the buttons on the GUI.

   a. prelab_image_ui.py in ~ese150/lab12 is the same as prelab_ui.py except that it uses an image for the "on" button instead of text. Use that as a template to see how to use images on buttons. Again, diff may be helpful in identifying the exact differences between the two files.
   b. Coordinate with adjacent teams for testing.
   c. Test and Debug.
   d. Have a TA use your GUI.
      i. Note how easily the TA could use your GUI and/or what problems they had.
      ii. Include that results in your writeup.
      iii. This is the required part of the lab checkoff.
   e. Include visual_outlet.py in your writeup.
11. Use the lab UI rubric to compare 4 UIs and include in outlet
   a. Using netcat -- Section 1, Step 2
   b. Using a script – Section 1, Step 6
   c. multi_outlet.py
   d. visual_outlet.py

   You probably want to write this up properly outside of lab time.
   Make sure you take adequate notes on your experience during lab that you can complete this outside of lab.
12. Improve the GUI further.
   a. How would you make the GUI even more usable?
   b. Draw up the revised GUI in PowerPoint and describe the interactions.
      i. Include in your report.
   c. Assess the improved GUI using the UI rubric.
   d. (optional) implement the GUI or a step between the visual_outlet.py GUI and this envisioned GUI.

Lab – Section 3: Smart Phone GUI (encouraged, but optional – worth 2 bonus points)

- Develop and deploy a GUI for your phone.

1. Copy over the starter GUI from ~ese150/lab12/mobile_outlet.dart to your flutter/my_app/lib directory.
2. Look through the code; note wemo_send() which performs a similar role to sendCallback in the python code.
3. Note that as given it behaves like the preclass, printing out a string when a button is pressed.
    a. It does this in two ways: (1) updating a text message on the phone, (2) printing to the connected console.
    b. It includes several different examples of buttons. Idea is that this demonstrates different button types and shows you how to assemble a vertical collection of buttons.
    c. Run the as-provided mobile_outlet.dart to see how it works.
        run $ flutter run -t <path to your mobile_outlet.dart>
4. Revise the code to control the same 3 outlets as you did at the end of Section 2.
    a. Replace the print/text-update operations with the callback to call wemoSend for a particular IP and suitable on or off message.
    b. Elaborate on and off buttons for each of the 3 outlets.
5. Download to your phone.
   First make sure your iPhone is connected to AirPennNet.
   To download your application, run $ flutter run -t <path to your mobile_outlet.dart>
    a. Test and debug.
    b. Coordinate with adjacent teams as before.
6. Show your interface to a TA.
    a. Record notes on the TAs experience using your GUI.
    b. This is the second (optional) part of lab checkoff.
7. Include your final mobile_outlet.dart in your writeup.
8. Assess this UI using the UI rubric.
9. If you do this, let us know how long it took to do this part in your writeup.

Postlab:

There is no separate postlab. Remember to complete the UI rubric assessment comparison among all 6 UIs (4 noted in Section 2, Step 11; then Step 12 and (optionally) Section 3, Step 8).

HOW TO TURN IN THE LAB

- Each individual student should author their own, independent writeup and submit a PDF document to canvas containing:
    - Recorded data and descriptions/explanations where asked.
    - Code developed.
    - Improved GUI (Section 2, Step 12).
    - UI Assessments.
- Note that this is due Wednesday, May 1st (last day of classes) at 11:59pm.
    - By Penn regulations, in a class with a final (which we have), nothing can be due after classes end.