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 processes running on different machines by identifying the machines and the port on each machine. Headers in TCP/IP (and UDP/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.

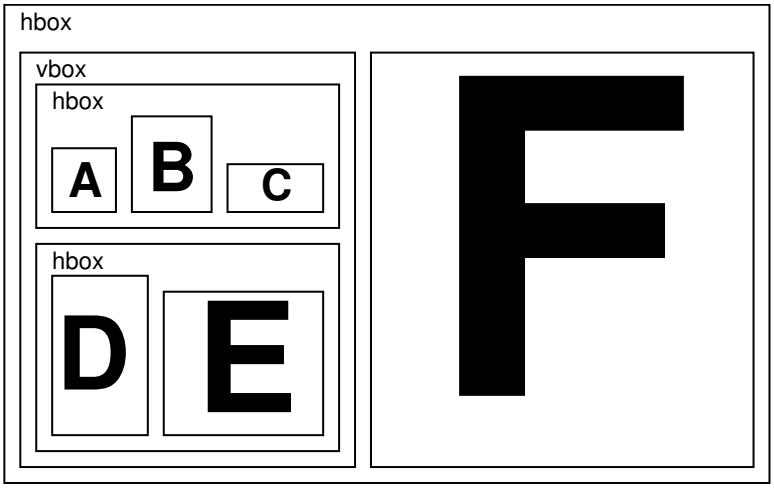
Consistent with Lab 11, we will use UDP packets to send commands. We won't keep an open stream. Since these are UDP (Unreliable Datagram Protocol) packets, some of them will get dropped. So, expect a few of your commands to need to be resent.

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 over-under (vertical or vbox composition) or side-by-side (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 – Section0: GUI Tools (Optional)

- You can run python and gtk on the Detkin Linux computers in Lab.
- If you want to run them on your personal laptop, you will need to collect the software you need to build and run the GUIs on your laptop.
- (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 "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)" < /dev/null 2> /dev/null`
        When prompted, type in your password. This command takes about 5 mins with good internet speed.
     c. Run `$ 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 pygobject3 gtk+3`
        [see: this link if you're having issues]
        This installation takes a couple of minutes. Run `$ brew list | grep gtk` again to see if you see gtk+3.
  2. To install PyGTK on Linux (Debian based Linux), use
     `$ sudo apt-get python-gtk3`
  3. To install python-gtk on Windows,
     a. Follow the instructions to install MSYS2 and python-gtk on the site below:
        https://pygobject.readthedocs.io/en/latest/getting_started.html
        i. In step 5 you may have to do "-Sy" instead of "-S"
     b. Whenever you need to use a terminal, run the following command in PowerShell:
        `C:\msys64\mingw64.exe`
     c. This will start a new terminal window, which starts in the path

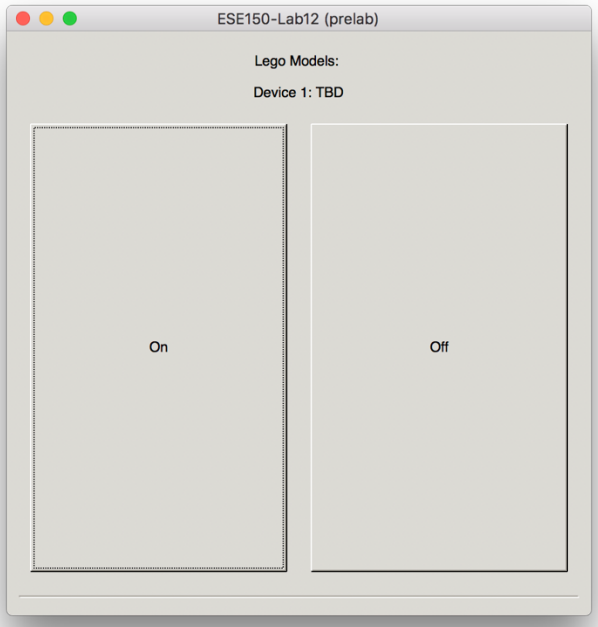
`C:\msys64\home\<username>`

d. You should move files into this location to see them in the terminal.

Prelab – Section 1: Python GTK Buttons

- Build and extend a simple Python GUI.
- Default is to use a Detkin Linux machines for this.

1. Open a terminal and create ESE150_Lab12.
   If running on Windows, this must be under your `C:\msys64\home\<username>` subtree as noted in the previous section
   ```
   $ mkdir ESE150_Lab12
   ```
2. Pickup a starting UI from ~ese150/lab12/prelab_ui.py
   a. ~ese150 is a unix shorthand for "the home directory associated with ese150"; this resolves to /home1/e/ese150
   ```
   scp PENNKEY@eniac.seas.upenn.edu:~ese150/lab12/prelab_ui_python3.py .
   ```
   b. Alternately, running on a Detkin linux machine, you should be able to just copy it to your ESE150_Lab12 directory:
   ```
   $ cd ESE150_Lab12
   $ cp ~ese150/lab12/prelab_ui_python3.py .
   ```
3. Run the python script by typing:`$ python3 prelab_ui_python3.py`
   If the code does not run, double check that you are using python version 3 and that any Anaconda environments are disabled with `conda deactivate`. You should see a GUI open up showing this:



   Now click on these 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 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. Note that while Python can tolerate spaces and tabs, it does not work to mix spaces and tabs. We provide you Python code with spaces so do not use tabs.
    ii. 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>”)
    iii. 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
    iv. Make this part of the hbox by using pack_start on hbox.
    v. Test and debug UI.
b. Change the layout so the new button is below the previous on-off buttons
    i. Copy prelab_ui_python3.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.

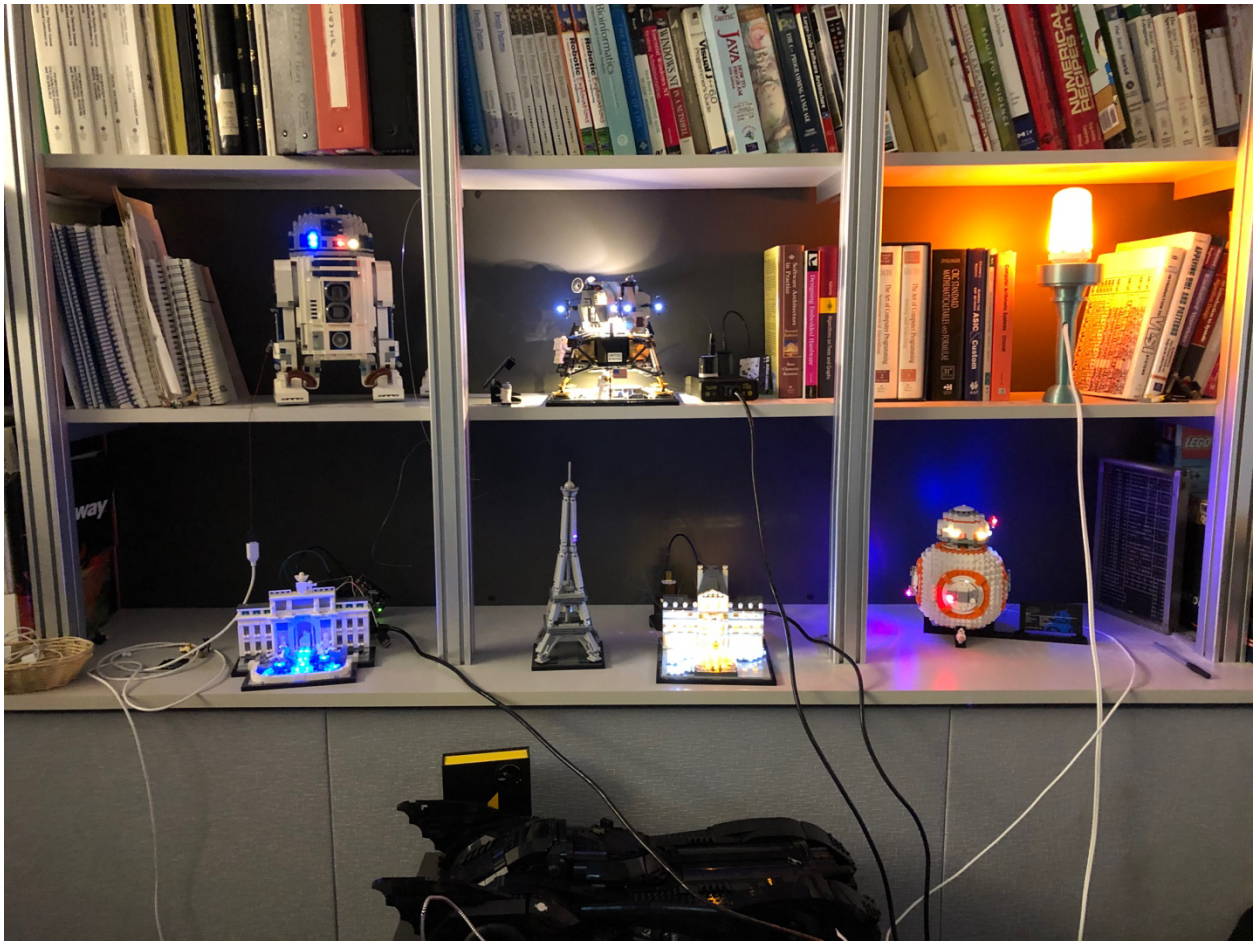
5. Include your final prelab_ui_python3.py, prelab_ui_vert.py and a screenshot of each GUI in your writeup.
6. Show your GUI to your TA for prelab checkoff.

Lab Procedure:

Lab – Section 1: Routes between computers

- Establish primitive network communication with an Internet-controlled power outlet
- Default is to use a Detkin Linux machine for this lab.

We will be connected to AirLift units programmed almost identically to the ones you developed for your WiFi Light Control in Lab 11. In fact, you will use your Lab 11 light controller as your first control target.
We will provide additional units controlling a number of lighted LEGO models:



1. Start by making sure you Lab 11 Light Controller is up and you know its IP address.
    a. Review Lab 11 instructions as necessary.
2. Turn the outlet on and off using the same nc that you used in Lab 11:
    a. `$ echo "1" | nc -u [your IP] 2390`
    b. `$ echo "0" | nc -u [your IP] 2390`
    c. Make sure the outlet goes on and off based on what you send it.

   d. Recall UDP is unreliable, so it may be necessary to send a command more than once. Use CTRL + C to regain terminal control if the program hangs.

3. Create a unix shell script that contains the unix command line that you used above (Step 5) to turn the device on and a second script to turn the device off.
   a. Name appropriately for where you are using your Lighted Control Module.
      i. Start a text editor like nano that you used in Lab 9.
      ii. Add a first line with:
      ```
      #!/bin/bash
      ```
      iii. On another line, place the "on" command you used in step 2a.
      iv. Save the file with a suitable name (like lego_on.sh)
      v. Repeat the same steps for the "off" command with a suitable name.
   b. Use chmod +x to make the shell scripts executable.
      ```
      $ chmod +x lego_on.sh
      ```
   c. Test that you can turn the outlet on and off with your shell scripts.
      i. You run the script by specifying it: `$ ./lego_on.sh`
   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 light that goes with the name of the shell script.

Lab – Section 2: GUI

- Revise/extend your GUI to control your outlet and other outlets.
- Continue to use a Detkin Linux machine.

1. Copy ~ese150/lab12/single_outlet_python3.py to your ESE150_Lab12 directory.
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 port number argument.
    f. Note that it takes a string to send in the command argument.
    g. You will formulate these 3 arguments into a 3-tuple and pass it to sendCallback.
    h. Note that single_outlet_python3.py also contains string variables that hold the "0" and "1" that can be used as the command string argument for this sendCommand sendCallback.
        i. For some other modules, the on/off might not be as simple as the ones we chose to use for the AirLift.
3. Revise single_outlet_python3.py GUI to control your Light Control unit:
    a. As provided, it has two buttons, on and off; they 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.
    c. If you get the IP or port wrong (or if your Light Control unit is unplugged), the buttons will do nothing. Double check the IP and port number are correct and passed in the correct order.
4. Test and debug your GUI.
5. Include your final single_outlet_python3.py in your writeup.
6. Copy single_outlet_python3.py to multi_outlet.py.
7. Revise multi_outlet.py by adding two more buttons so that it can control two lighted structures in lab in addition to your own Light Control unit.
    a. See Lab spreadsheet for models available and their IP addresses.
    b. You will need to be in Detkin or on the Penn VPN to be able to use the addresses provided and for your packets to go through.
    c. Label so it is clear which buttons do which things to which models.
    d. Coordinate with other teams for testing.
    e. Test and Debug.
    f. Include multi_outlet.py in your writeup.
8. Copy multi_outlet.py to visual_outlet.py.

9. Copy over the jpg images from ~ese150/lab12/
    a. Also take pictures to represent the on and off state of your local Light Controller and add those to your set of images.
10. Revise visual_outlet.py so that it uses the images to visually denote the lighted structure controlled by the buttons on the GUI.
    a. prelab_image_ui_python3.py in ~ese150/lab12 is the same as prelab_ui_python3.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.
        i. It may be useful to use the diff command to get you started:
        $ diff prelab_ui_python3.py prelab_image_ui_python3.py
        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.
    b. You may want to rearrange on and off buttons so they are a vertical rather than horizontal pair.
    c. Coordinate with other teams for testing.
    d. Test and Debug.
    e. 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 result in your writeup.
        iii. This is the required part of the lab checkoff.
    f. Include visual_outlet.py in your writeup.
    g. Include a screenshot of visual_outlet.py UI window in your writeup.
11. Use the lab UI rubric to compare 4 UIs and include in report:
    a. Using netcat -- Section 1, Step 2
    b. Using a script – Section 1, Step 3
    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.

Postlab:

There is no separate postlab. Remember to complete the UI rubric assessment comparison among all 5 UIs (4 noted in Section 2, Step 11; then Step 12).

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.
    - Screenshot of visual_outlet.py UI window.
    - Improved GUI (Section 2, Step 12).
    - UI Assessments.