2020

Computer Controlled Plant Environment

A-LEVEL OCR PROGRAMMING PROJECT

ADAM SEIDEL

Contents

Analysis

In my project I will be developing a fully automated greenhouse environment that will facilitate the growth of plants in the most efficient way possible. This project on an expanded scale will enable garden centers to optimize the growth of their plants with features that can automatically control the temperature, humidity, light intensity, water, and soil nutrients of the plant environment. Another possible user of my project will be scientists to investigate ideal conditions for plant growth and to generate reports on historic conditions inside the environment. During scientific experiments it is important to control many variables as not to effect results and this project will be able to keep desired variables inside a very small window. Residents in harsh climates could also use this project to grow plants that would not survive due to lack of sun light, rain, humidity, or temperature.

This will be achieved through feedback loops that will monitor various environmental readings and respond accordingly to keep all the environmental variables within the optimal range. For my project I will aim to create a small-scale green house with space for one plant. This greenhouse will link to a raspberry pi that will be responsible for monitoring and controlling the environment whilst also developing a GUI that will display graphs and readings from the greenhouse. The greenhouse will have the ability to be remotely controlled either via a website or an app.

My proposed project is solvable using computation methods because it requires the constant monitoring and adjustment of multiple environmental variables. In a low-tech solution, the greenhouse would have to be controlled manually by a member of staff. This member of staff would be responsible for continually checking sensors in the green house and then manually adjusting the conditions in the green house. In this setup there are large periods of the day such as during the night where conditions are left unmonitored. External environmental changes during this time could leave the plants left in potentially fatal conditions. My project will be able to respond to these changes without the need for external input and alert staff to any issues via the remotely controlled interface.

Using feedback loops it will be possible to keep the greenhouse in an almost constant environmental state. My project will be able to process the many readings coming from sensors in the greenhouse and simultaneously act upon these readings. This will be done in near real time and lead to a much-improved accuracy over what a human could manually achieve. I anticipate that my greenhouse will generate vast amounts of data that I can use to analyze the performance of plant growth. This data collected will be on a new scale to what a human could ever manually record and will allow in depth graphs and metrics to be displayed and calculated.

I will be creating a GUI that will aim to provide the user a visual representation of the readings being generated by the sensors inside my automated plant environment. Key metrics will be displayed to the user such as current temperature and humidity along with options that will allow for a manual override of current conditions. Other potential features for my GUI include a graphs section showing long term sensor data points in an easy and engaging way along with an option to automate condition changes at certain times of the day. This automation of changes will allow the greenhouse to mimic a day with the temperature rising during the day and falling during the nighttime. I will be conducting extra research into how plants best develop and use this to ensure my project has the features required to facilitate this development in an automated and efficient environment.

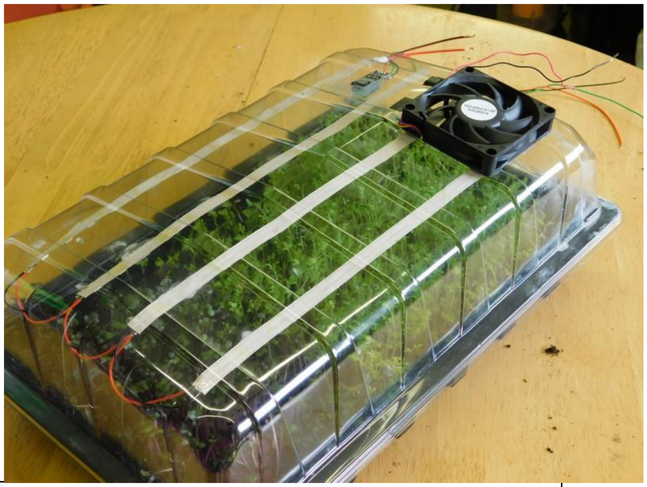
This project can expand with the addition of many complex features. However, at its core this project will be a big jump in accuracy on current manual low-technology solutions. The time saved for greenhouse managers will be large and the increased accuracy will provide an economic impact due to more efficient plant growth. This will lead to an increased plant yield across a year and a reduced fatality rate. The increased stream of data with a higher degree of accuracy to current standards will provide users will the opportunity to discover the optimum parameters for various species to develop. The reduced time required to monitor and grow plants will enable greenhouse owners to either cut staff or to expand their operations with little ongoing costs as one member of staff would now be able to manage a much larger soil area than before and the only upfront cost being another automated green house.

Research of existing solutions

Existing project 1

https://maker.pro/pcb/projects/diy-build-mini-automated-greenhouse-microgreen

This project creates a mini automated greenhouse for growing microgreens. The circuitry has a microcontroller to read the sensors and adjust the environment via the fan, water pump and growing lights. Two sensors are used which are a moisture sensor that detects the amount of water in the soil and a combined temperature and humidity sensor. The project has a main loop that constantly monitors the readings from the sensors and then activates or deactivates the various output devices to alter the environment.



Advantages	Disadvantages
• The program is very simple under 100 lines of codes and its relatively easy to adapt the parameters. • The developer has included a sleep feature for the plans that turns off the growing lamps during the night allowing the plants to rest.	• All the output devices are either on or off there is no ability to vary the fan speed or light intensity. • No readings from the sensors are stored and no reports are generated to show the historical conditions inside the greenhouse. • The project has no remote access feature to allow the user to monitor and change parameters from a wireless device.

Features to include

I will be taking inspiration from the physical design of this project as I like its simplicity along with the ability to move the lid onto a new growing tray easily. In my project I will look to include a sleep function for the plants to give them rest. Including this sleep feature will add another real-world feature to my project. The sleep function could be used in real world implementations when growing plants efficiently or in a laboratory setting when investigating optimum day light hours for plant growth. Ensuring all my output devices have variable outputs will add an extra layer of complexity to my programming implementation whilst also giving greater control over the environment. In a basic implementation such as above the condition for turning on the fan is simply if the temperature drops below a predefined value. Being able to vary the strength of the fan would allow for a feedback loop to be created where the fan speed is varied based on the temperature. These feedback loops can be used with all the sensor and output device pairings to exponentially increase or decrease the output to change the sensor readings the further away the readings get from the desired value.

Existing project 2

https://autogrow.com/

Autogrow is a commercial greenhouse automation solution company. They specialize in controlling all the environmental variables inside a greenhouse on an industrial scale. Plant run-off is measured to ensure that the plants are being grown in a legally compliant environment. The greenhouse can be remotely controlled from any device and the system can send alerts to managers when there's a problem. Autogrow focuses on retrofitting greenhouses with their automated technology that can manage factors such as vents, heating, cooling, lighting, temperature, CO2, irrigation, and a retractable roof. They're advertised advantages of their system are decreased labor costs, increased accuracy and increased quality and yield. Other solutions produced by this company are for automated indoor growing such as inside a shopping center. This allows the owners to reduce time spent maintaining plants and not worry about their decorative plants looking unhealthy / unkept.

Advantages	Disadvantages
• The system controls every possible environmental variable and can monitor and log all this data. • Remote access means the managers don't have to be at the physical greenhouse location when making settings changes. • The alerts system makes sure problems are dealt with quickly.	• The sensors are all very high tech. The plant run-off monitoring especially requires expensive equipment. • There system is aimed at large industrial greenhouses with no options for smaller recreational setups.

Features to include

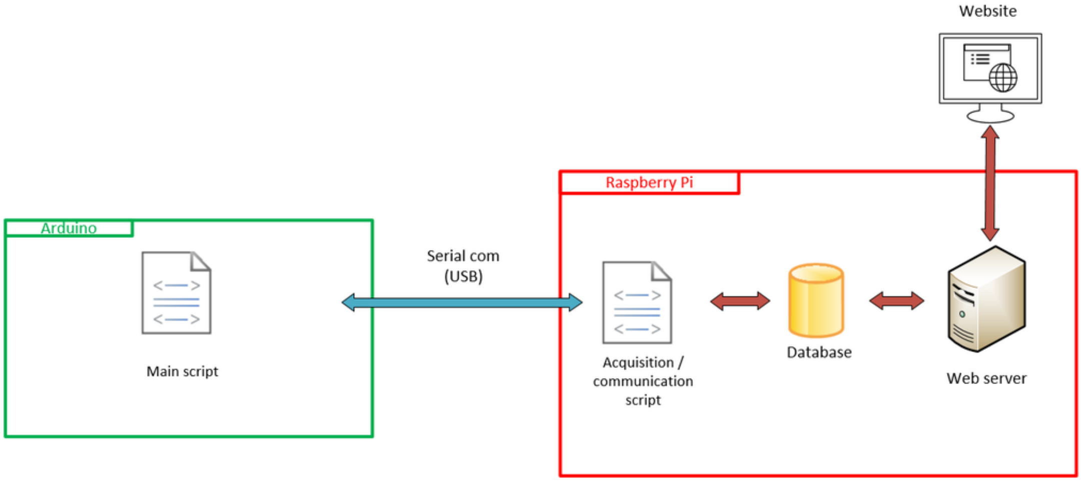
I liked the technical implementation of this project. There remote access feature makes the whole automated environment much more useful in the real world and the alerts system draws the users' attention only when human input is needed. Autogrow also base their sensor readings on relative measurements. This means the readings from the sensor are adjusted to consider the outside environment. This allows the controlled environment to be tailored to reflect the real-world conditions for the plants. Whilst this feature is not needed for all plants it is useful for when you are growing plants to eventually be kept outside the greenhouse. In my own project I'm going to include some sort of alerts system either via email or mobile phone notification. These alerts will give a daily status update and warnings when a failure or issue arises. Another feature I will implement is the remote access. This will probably be through a website that will give the user full access to the system with full remote-control ability.

Existing project 3

https://www.instructables.com/Automated-Greenhouse/

This HelHa Automated greenhouse system has a website interface that connects to a MySql database that stores all the data being generated from the environment. The MySql server is ran on a raspberry pi with python used to update the database with new readings and send signals via USB to the Arduino that is responsible for the motors, sensors, and output devices. The webserver is installed on Apache2 and has a basic main menu to show the live measurements for the environment and a devices state tab to show what the output devices are currently doing. There is also a commands and parameters page on the website that allows the user to switch control for each output device from manual to automatic and set new values for the internal temperature and soil moisture.

Advantages	Disadvantages
• The project has well thought out layers. The system architecture diagram below from the project page shows the data flow and hardware requirements. The layers mean each section can be edited and improved without effecting the other layers. • All the data being stored in a database means it would be easy to generate reports on the data from the sensors. • The website interface makes it simple to control the system without having to install any apps or additional software.	• The website does not feature any login to restrict access to only authorized users. • The input boxes on the website are directly passed and stored in the database. This means there is a possibility for SQL injection attacks.



Features to include

I liked the way this greenhouse is broken down into three layers shown in the system architecture diagram above. I will design my project in distinct layers to allow me to make changes to each of the layers without effecting the operation of the overall system. Using an object-oriented approach will help to mold my project into distinct layers as I will be required to decompose my problem into various classes. The website is also easily accessible for this project with no additional software required. I like this simplicity so will look to produce a control website for my project to allow remote control of the greenhouse. Unlike this project I will look to include some security features such as user login to avoid unauthorized access to the system. The gui is simple to navigate but the developers of this project have not made use of the vast amounts of data that is stored in their database. I will take inspiration from there simple gui but aim to add features that make use of the thousands of sensor readings being generated from the greenhouse. Such as producing graphs for past data and statistics such as mean and range for the different data points.

Stakeholders

Stakeholder 1

Name: Elizabeth Allgar
Age: 43
Job: Computer Science Teacher

Why are they a suitable stakeholder?

Elizabeth Allgar is a recreational gardener who enjoys her hobby but struggles to find time during her day to water and care for her plants. She is looking for ways to keep her plants but reduce the time she must manage them. Elizabeth keeps her plants inside her classroom so has easy access to a power supply but is away from her room during the long summer holidays and over the weekend. Elizabeth will be able to give feedback from the view of a hobbyist and will be able to compare my system to her current routine.

The stakeholder would like the following features:

- Alerts for when there are problems such as the watering system being out of water
- Be able to set how often the plants should be watered
- Have some example settings for different types of plants to help her when setting the parameters such as a plant settings data base
- The system should react to changes in the room such as the air con being turned on or off

Stakeholder 2

Name: Tobias Lester
Age: 28
Job: Commercial Greenhouse owner

Why are they a suitable stakeholder?

Tobias works at a local greenhouse that specializes in high volume and low margin wholesale plant sales. Having spoken to Tobias he has explained that they own a total of 3 large industrial sized greenhouses that are constantly heated and watered. Currently whilst their system is digitalized there is no automation, and the environment is still manually adjusted and controlled via a local onsite control board. When watering is required the site manager on duty must go through the process of turning on the watering system and then deactivating once watering is complete. My stakeholder has mentioned this as a potential area of improvement as manually controlling the system does lead to regular human error and makes it harder for them to grow a variety of plants as each will require different conditions. Automating Tobias' greenhouse will allow him to reduce his staff whilst also increasing the yield of the company.

The stakeholder would like the following features:

- Automated water, light and temperature controls to reduce the dependency on staff
- An alerts system in case there is an unusual issue in the greenhouse that requires staff attention
- A reporting system that delivers Tobias easy to read graphical information informing him on performance

Stakeholder 3

Name: Christopher Mastin
Age: 67
Job: Botanist

Why are they a suitable stakeholder?

Christopher works as a research botanist his work is based on investigating the optimal growing conditions for plants and investigating which factors have the greatest impact on growth. Through his many years of work Christopher has regularly been manually growing hundreds of plants at the same time. Each plant receives slightly different conditions based on the investigation. To keep his experiments, fair his team work tirelessly to control as many control variables as possible. However, over the weekends when the university is closed this proves difficult. Being able to precisely control each individual plant environment will reduce the uncertainty in his investigations and free up his teams' time to manage a much larger number of plants at the same time.

The stakeholder would like the following features:

- The ability to manage multiple plant environments off one system
- Accurate reports to be generated for scientific analysis
- Accurate control of the environment with external effecting factors removed such as external sunlight

Solution Features

Required Features	Desirable Features
The system must be able to automatically control the internal environment continually without any human input. Except to refill the water pump system.	The ability to manage multiple plant environments off one system. Due to hardware, budget and time constrains this is potentially a feature I will not be able to implement.
An easy-to-use graphical user interface must be developed for the system that makes it easy for the user to alter environmental variables and the run schedule without any programming knowledge.	A reporting system that shows graphical representations of the data that is being generated from the sensors in the environment.
Alerts sent via email or notifications to the users' phone to let them know if their attention is	Different default programs that can be used for less experienced users to help them begin to

required due to a fault or issue. The alerts could also provide scheduled updates to the user presenting data from the system.	grow different plants. Whilst these default settings won't be perfect, they should serve as a good starting point.
The system needs to have a scheduling feature to allow for all the environmental controls to be ran on a regular interval. This would mean the user could choose to water the plants every hour or at any other given interval.	Encryption and password protection systems to restrict unauthorized access to the greenhouse. This will be key if I am to implement remote access as this opens the system up.
Sensors will be used to allow the program to react to changes in the greenhouse due to external factors such as aircon systems in the room and external sun light.	Save current settings to make it easier to reconfigure if the system is down for any reason.
All data collected from sensors should be permanently stored. This allows for scientific analysis to be made of the system and its impacts. This data could potentially be stored on an external server to limit data loss in the event of a failure.	Help notes in the GUI will assist new users in getting the system working and let them know what all the different features do.
A login system will be used to ensure only authorized users can access the controls. I will be using a secure hashing algorithm to make sure passwords are stored safely.	

Limitations

I believe that in the given time for this project I will be able to implement all my required features and have them working to a satisfactory level. Due to budget constraints, it might not be possible to address some of the desirable features such as the ability to control multiple environments. It is possible that I will be able to implement this feature in the system without having any capacity to test for this function.

The accuracy of my sensors will also be a liming factor as I will be restricted by their accuracy when recording measurements. This could lead to situations where my system is unaware of slight changes in the environment if they are not detected by the sensors and as such there is no way to respond. This limitation is also true for my output devices as I will only be able to affect the environment within the ranges of my heater and cooling systems. On extremely warm or cold days it could be possible that the automated systems are not able to bring the various variables back to their accepted ranges.

It will not be feasible to produce a fully secure system with all data fully encrypted. Whilst every effort will be made to securely store and authenticate user login details most likely via a hashing algorithm. There will still be large amounts of data that will not be encrypted such as the data generated from the sensors and any information sent via the remote access feature will also be hard to ensure security. Encrypting the sensor data would make it harder for me to manipulate the data and a whole new layer of complexity. For this reason, I will aim to produce a secure login system where user passwords are stored in a secure manner without worrying about securing any other data.

My system will be developed to run on a Raspberry Pi using Python as the main programming language. This will place limitations on which platforms the application can be released on. For example, I will be making use of the Raspberry Pi pins to attach my sensors and other devices. The notification system will also only work with one system either notifying the user via Email or via an app notification API. To reduce time spent setting up the system I will be using sensors with prebuilt libraries to deal with taking readings and brining this data into python. This will make my system dependent on these libraries as I will optimize my code to make best use of my specific sensors functions along with inevitable hard coding of the prebuilt libraries into my own application.

Software and Hardware requirements

Software/Hardware	Why they are needed
Linux/Raspbian	The program will be developed for a Raspberry Pi. To interact with the sensors, I will use raspberry pi specific libraries for those sensors. Using a different OS could mean some of the sensors will not work.
Python 3	The program will be programmed in Python 3 which is not compatible with other python versions.
Mouse	A mouse will be used to navigate the GUI and when using the remote access features.
Raspberry Pi	A Raspberry Pi will be used to run the program and host the remote access features. I will be using a Pi due to its small design and affordability along with its GPIO pins.
Database	Some form of MySQL / MySQL light database will be needed to store all the sensor data in an efficient and accessible manor. This is more desirable than storing data inside a text file as databases are inherently easier to manipulate and analyze.
Webserver	To facilitate the remote access feature an Apache webserver is needed so that the user can interact with a website interface that then communicates with the raspberry pi to control the greenhouse.
Heating element	This is the element that will be used to control the temperature of the greenhouse. Depending on the strength of the sensor multiple may be needed.
Computer Fan	An old computer fan will be used to cool the greenhouse and ensure fresh air enters the greenhouse. This will be attached to the Pi via a relay to allow for variable control of the fan.
LED light strips	LED strips are an efficient and cost-effective way to change the light intensity of the greenhouse environment. The strips will allow for a long line of LEDs to be ran around the greenhouse that are all controlled from one relay.
Water pump	Watering the plants will require a pump to spray water across the plants. The pump will need to

	be high powered so that a good covering of water is achieved over the plans. A ready-made watering pump will be used to save time developing my own.
Soil Moisture sensor	This is the sensor that will indicate when watering is required. These sensors are very easy to get hold of and are placed into the soil giving back readings to the Pi about soil moisture.
Enviro for Raspberry Pi	This all-in-one board for the raspberry pi can monitor multiple variables inside the environment. The board can measure temperature, pressure, humidity, light and gas. This will save the need for connecting many sensors to my breadboard and make it easier to get readings as the pre-built library will be used to interact with the enviro.
Servo motor	To control the window in the greenhouse a servo will be mounted at the side of the window. Activating the servo will open the window and vias versa.

Success Criteria

Usability

- The interface should be designed to minimize the number of clicks to reach any feature
- The interface should be easy to navigate for both new and experienced users
- Must be intuitive for new users without compromising the more in-depth features used by experienced users
- Help buttons placed near key features that describe how to use the related functions
- A home button that is easily accessible to take the users to the main page
- A modular design to keep related areas together

The suitability of the interface is subjective so I will have to ask my stakeholders to review the interface and measure my success based on their response.

Functionality

- Users will find the system easy to control and intuitive to use
- The system should be robust and can run indefinitely with minimal input
- Users can enable notifications to their emails / phones updating them on progress
- There will be a login system to prevent unauthorized access
- Preferences / settings should be saved
- The system will perform regular tests on the sensors and output devices to ensure all are working
- Any data generated should be saved to the database with backups made
- The remote access feature should be easy to interact with

Security

- Sensitive user data such as passwords will be stored in a secure hashed and salted format

Robustness

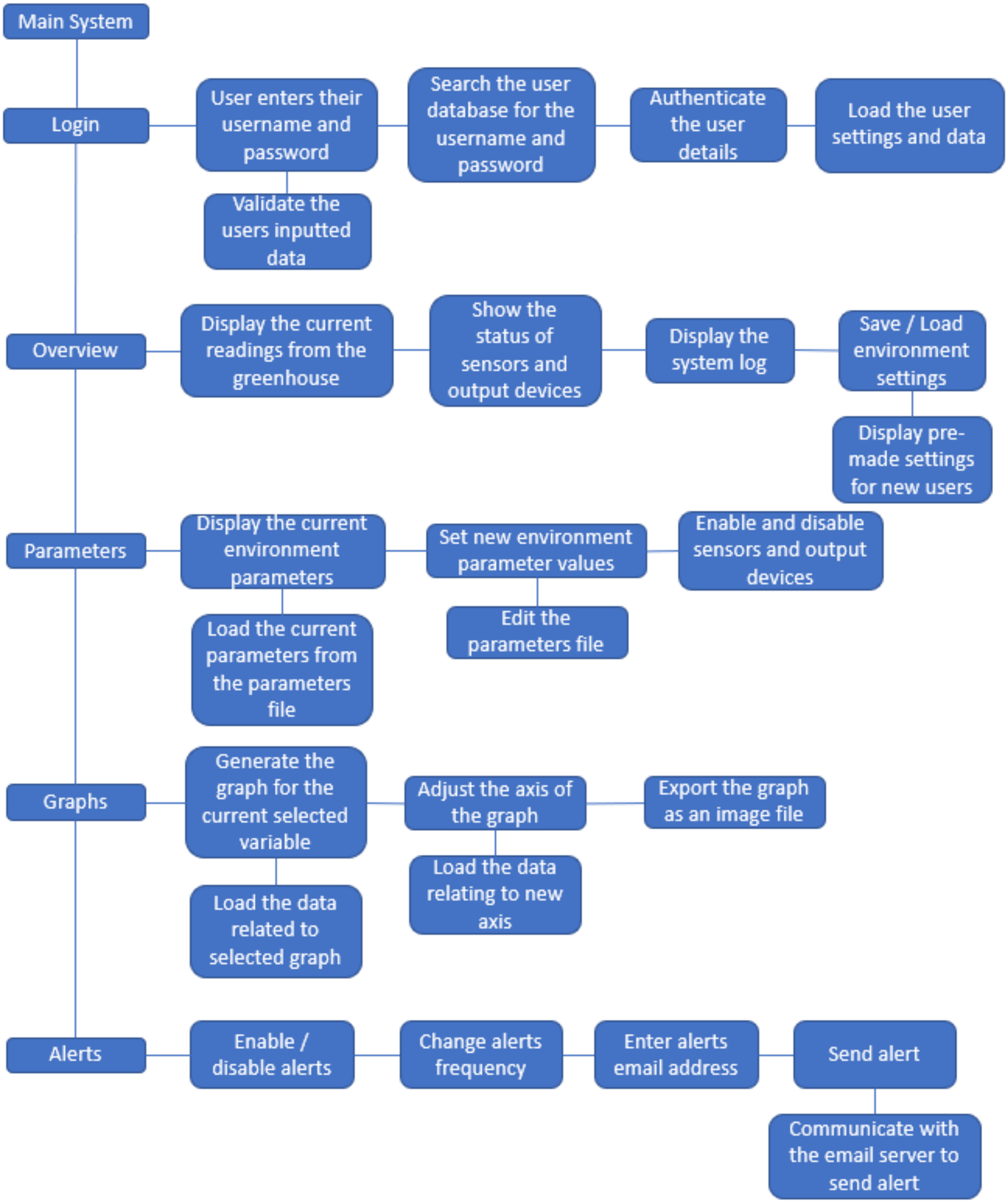
- The system should have the ability to still work when certain sensors or output devices are down or not connected
- All data should be periodically backed up to prevent the risk of data loss in the event of a failure
- Timeouts should be built into the system so that the program is not constantly stuck attempting to access a sensor and ending up in a loop
- Settings should be saved so that the system can be easily restarted without major setup works

Performance

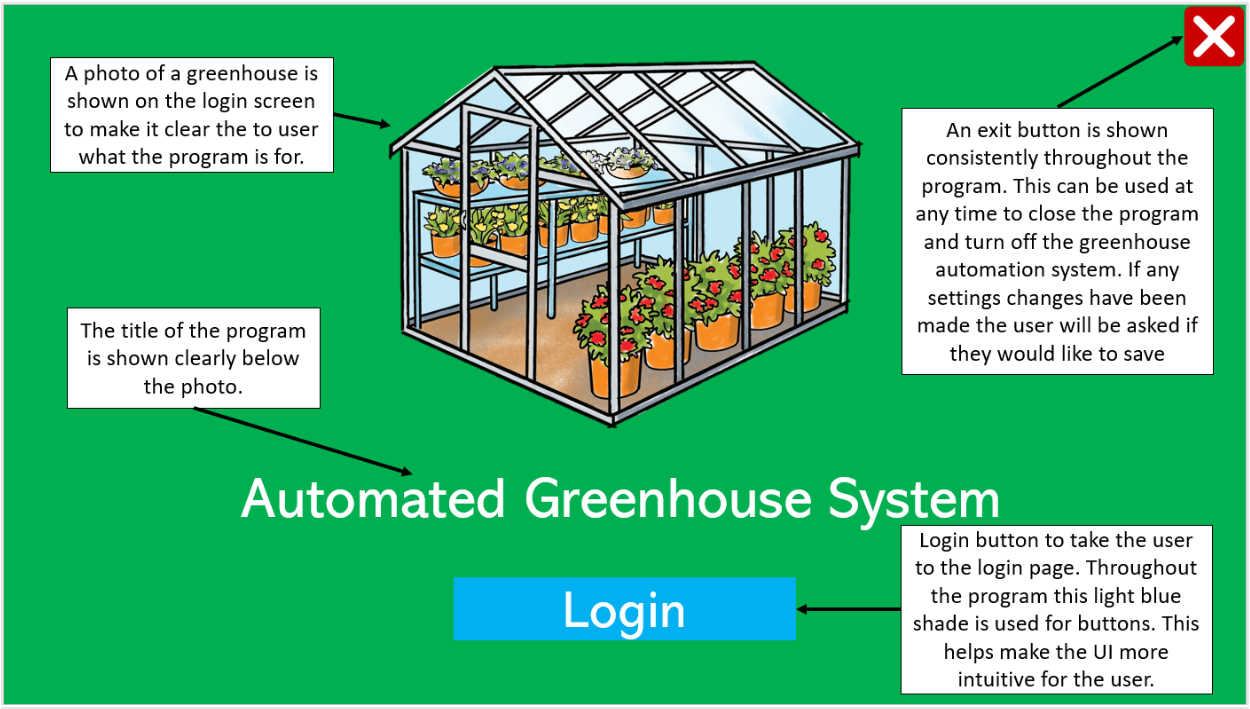
- When generating reports from the data the program should be optimized to generate them in the shortest time possible
- All methods will be reviewed to look for ways to optimize the code
- Areas such as how often a reading is taken and how often that reading is recorded will be reviewed to find the best balance for performance and effectiveness in terms of the plant growth
- When using the remote access feature not all data should be loaded unless it is specifically requested by the user. This will reduce wasted data transfer and stress on the webserver

Design

Decomposition diagram


Main System
Login
User enters their username and password
Search the user database for the username and password
Authenticate the user details
Load the user settings and data
Validate the users inputted data
Overview
Display the current readings from the greenhouse
Show the status of sensors and output devices
Display the system log
Save / Load environment settings
Display pre-made settings for new users
Parameters
Display the current environment parameters
Set new environment parameter values
Enable and disable sensors and output devices
Load the current parameters from the parameters file
Edit the parameters file
Graphs
Generate the graph for the current selected variable
Adjust the axis of the graph
Export the graph as an image file
Load the data related to selected graph
Load the data relating to new axis
Alerts
Enable / disable alerts
Change alerts frequency
Enter alerts email address
Send alert
Communicate with the email server to send alert

Usability features

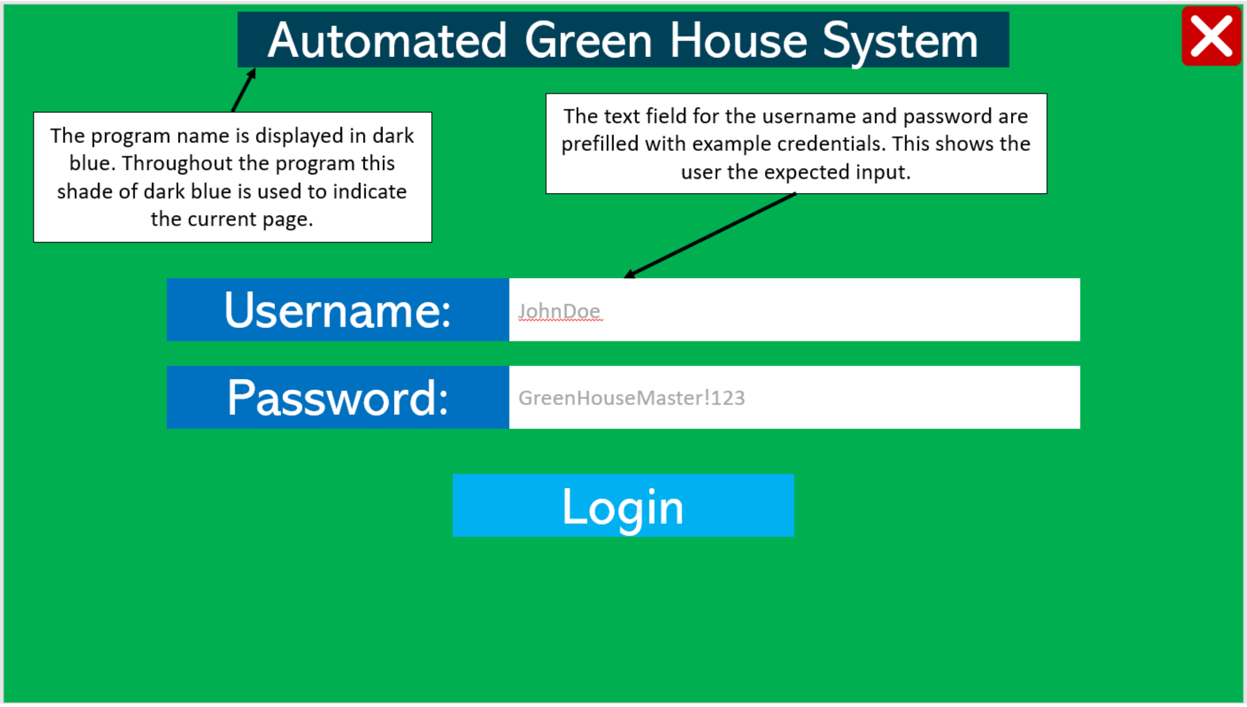


A photo of a greenhouse is shown on the login screen to make it clear the to user what the program is for.

The title of the program is shown clearly below the photo.

An exit button is shown consistently throughout the program. This can be used at any time to close the program and turn off the greenhouse automation system. If any settings changes have been made the user will be asked if they would like to save

Login button to take the user to the login page. Throughout the program this light blue shade is used for buttons. This helps make the UI more intuitive for the user.



The program name is displayed in dark blue. Throughout the program this shade of dark blue is used to indicate the current page.

The text field for the username and password are prefilled with example credentials. This shows the user the expected input.

Username: JohnDoe

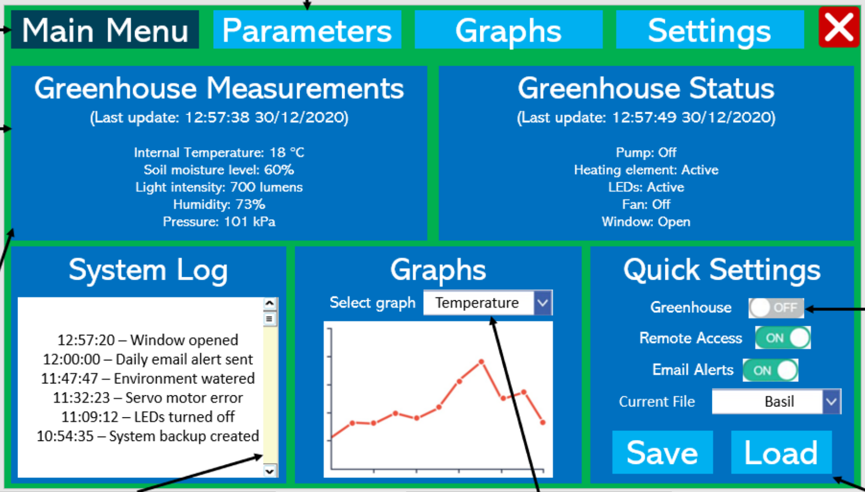
Password: GreenHouseMaster!123

Light blue shade once again indicates buttons to be interacted with.

Dark blue shade to indicate current page

The UI is based on a modular block design. Related features are grouped together inside one box. This supports users to understand what is related.

The main menu page gives an overview of the greenhouse system. This page is intended to be sufficient for daily use. Unless the user wants to make settings changes or view graphs.



Greenhouse Measurements
(Last update: 12:57:38 30/12/2020)

Internal Temperature: 18 °C
Soil moisture level: 60%
Light intensity: 700 lumens
Humidity: 73%
Pressure: 101 kPa

Greenhouse Status
(Last update: 12:57:49 30/12/2020)

Pump: Off
Heating element: Active
LEDs: Active
Fan: Off
Window: Open

System Log

```
12:57:20 – Window opened
12:00:00 – Daily email alert sent
11:47:47 – Environment watered
11:32:23 – Servo motor error
11:09:12 – LEDs turned off
10:54:35 – System backup created
```

Graphs
Select graph: Temperature

Quick Settings
Greenhouse: OFF
Remote Access: ON
Email Alerts: ON
Current File: Basil

On / Off toggles are used when the user has the option to enable or disable a feature. Toggles are used when there is only two options making is super fast to swap between the two options.

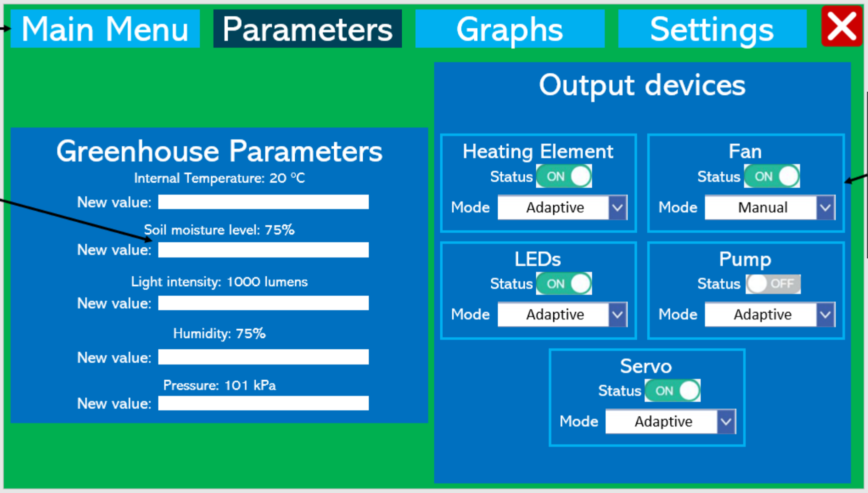
Scroll bar gives the user the ability to move through the system log. The use of a scroll bar means less space is taken up on the main menu page. I used a scroll bar instead of a page system to make the log seem continuous as apposed to splitting it into pages.

A drop-down menu is used to present the user with the available options to choose from. This type of input guides the user to only enter valid options. I chose to use a drop-down menu instead of a text box to stop the user entering an invalid option

Two large save and load buttons in the familiar light blue button colour have been placed in the quick settings box to allow the user to easily save and load settings files.

Fonts have been kept consistent across the UI. Menu buttons have a font of 40 whilst headings are 36 and all other text font size 16.

When larger inputs are needed text boxes are used. These will be confirmed by pressing the enter key to save an input. Data inputted into text boxes will have to be validated.



Greenhouse Parameters
Internal Temperature: 20 °C
New value:
Soil moisture level: 75%
New value:
Light intensity: 1000 lumens
New value:
Humidity: 75%
New value:
Pressure: 101 kPa
New value:

Output devices

Heating Element
Status: ON
Mode: Adaptive

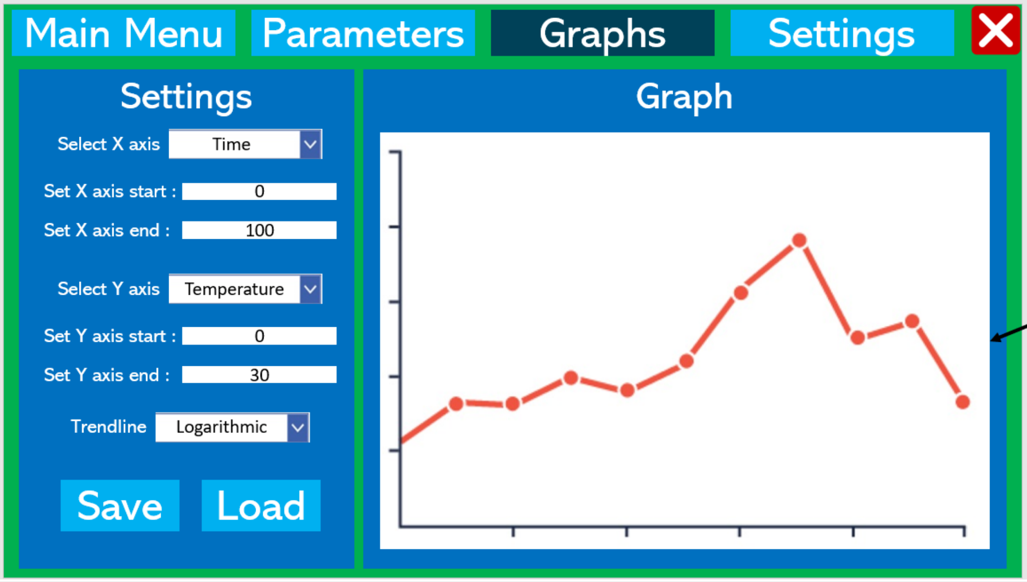
Fan
Status: ON
Mode: Manual

LEDs
Status: ON
Mode: Adaptive

Pump
Status: OFF
Mode: Adaptive

Servo
Status: ON
Mode: Adaptive

Light blue boxes are used to further divide the output devices section. These help to show the user which buttons are related to which device and avoid confusion.



Select X axis Time

Set X axis start : 0

Set X axis end : 100

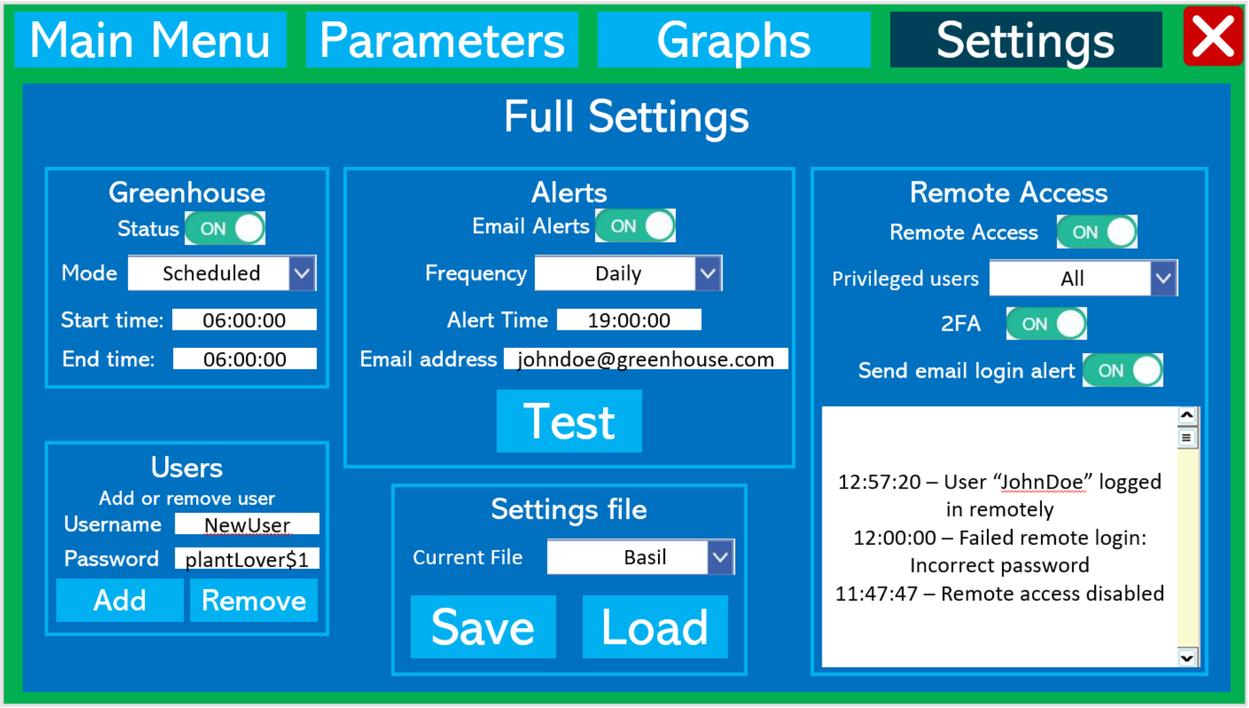
Select Y axis Temperature

Set Y axis start : 0

Set Y axis end : 30

Trendline Logarithmic

Changes made to the graph settings will result in graphical changes on the graph. A graph has been used to help represent the data from the greenhouse sensors in an easy-to-understand manor instead of displaying the raw date.



Status ON

Mode Scheduled

Start time: 06:00:00

End time: 06:00:00

Email Alerts ON

Frequency Daily

Alert Time 19:00:00

Email address johndoe@greenhouse.com

Remote Access ON

Privileged users All

2FA ON

Send email login alert ON

```
12:57:20 – User “JohnDoe” logged
in remotely
12:00:00 – Failed remote login:
Incorrect password
11:47:47 – Remote access disabled
```

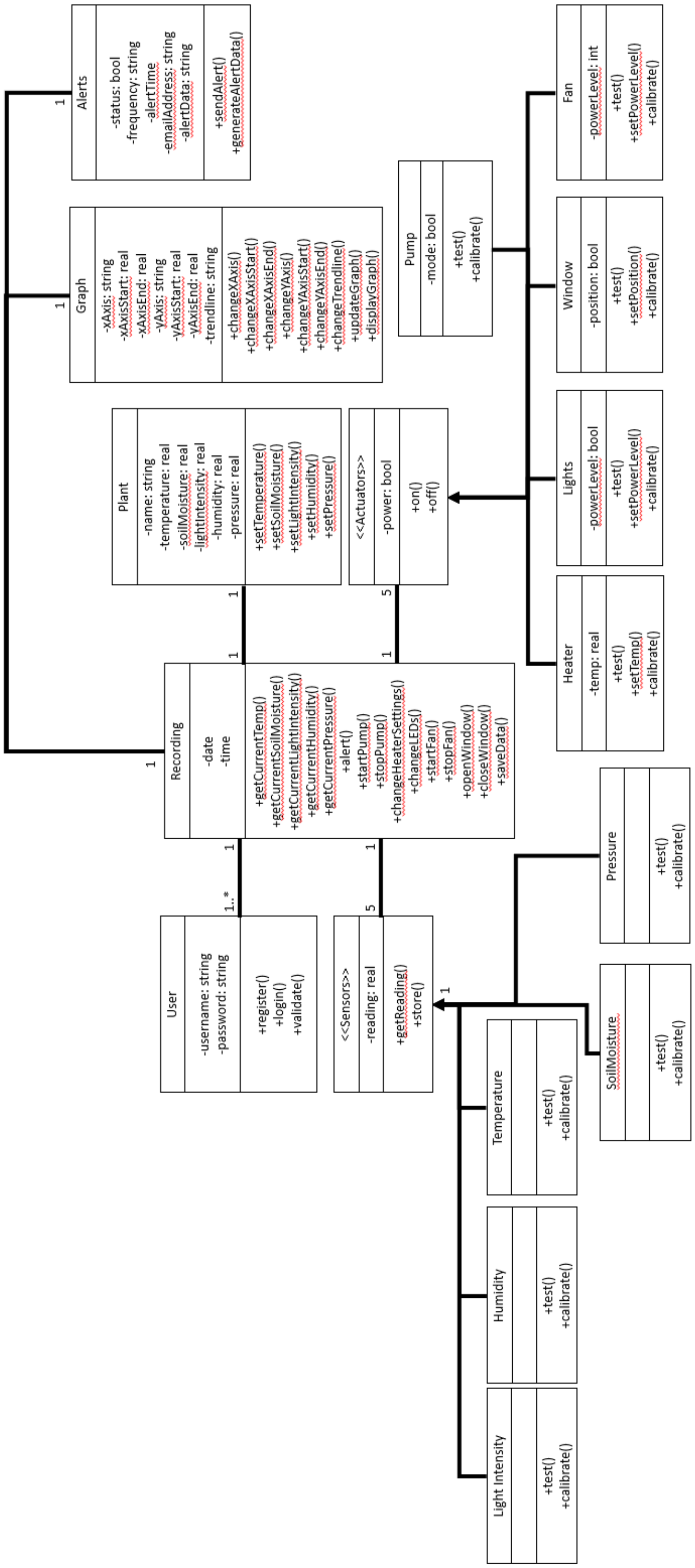
Add or remove user

Username NewUser

Password plantLover$1

Current File Basil

Class Diagram



Testing Strategy

Iterative testing will be carried out during the development of my program. As I program new modules, I will test each one to ensure that it functions as expected. This technique will ensure that all code works as expected and reduce the workload when I come to post-development testing. Whilst each module may work as expected individually this does not mean they will function differently when asked to interact with the wider program. At the end of the development of each class I will perform a complete unit test. This form of white box testing will help me to understand my own code and how data is manipulated inside the class. Any issues or inefficiencies presented will be identified and fixed. During this unit test I will pay particular attention to internal structure, design, and implementation of the class.

Once an error is identified in the program development will be paused whilst a fix can be implemented. This makes sure errors are not compounded by further development that buries the error inside the code making it harder to change without major programmatic changes. Using IDE debugging and testing features I will be able to track the values of variables to ensure no logic errors exist.

Once the program has been developed and tested iteratively, I will perform a full post development test this stage of testing will focus on ensuring the various modules interact with each other in an efficient and expected manor. A comprehensive and challenging testing plan will be developed that will cover every function of the program. Testing each function of the program will make sure all areas of the program function as expected. Any failures in the testing plan will be identified and a solution implemented. This stage of the testing will be carried out using a Black Box testing method where the tester will have no knowledge of the internal structure, design, implementation, and flow of data in the program. This makes sure no bias is introduced by a developer who knows how the program should be operated.

Destructive testing will also be carried out to assess the robustness of the program. Screen capture technology will be used to assess how an end user and the program interact with each other. This will provide insights into how the software is behaving compared to the intended function by the developer. Areas that will particularly be looked at will be time taken for the software to complete certain tasks and ease of use. Screen capture will highlight moments of improper software usage highlighting any changes to the UI that can be made to improve usability and robustness.

Further testing will be carried out to see how the software performs when improper data is inputted to the system. The testing plan will include maximum and minimum value tests to see if the software is able to handle them correctly. Data validation will also be a key focus of testing as the use of a database will introduce the threat of an injection attack that could potentially corrupt the database. Another area to be tested will be proper data output. Comparing the expected output to the produced output. Thorough robustness tests will ensure the software will function well in the real world when end users use the software in ways not envisaged by the developer.

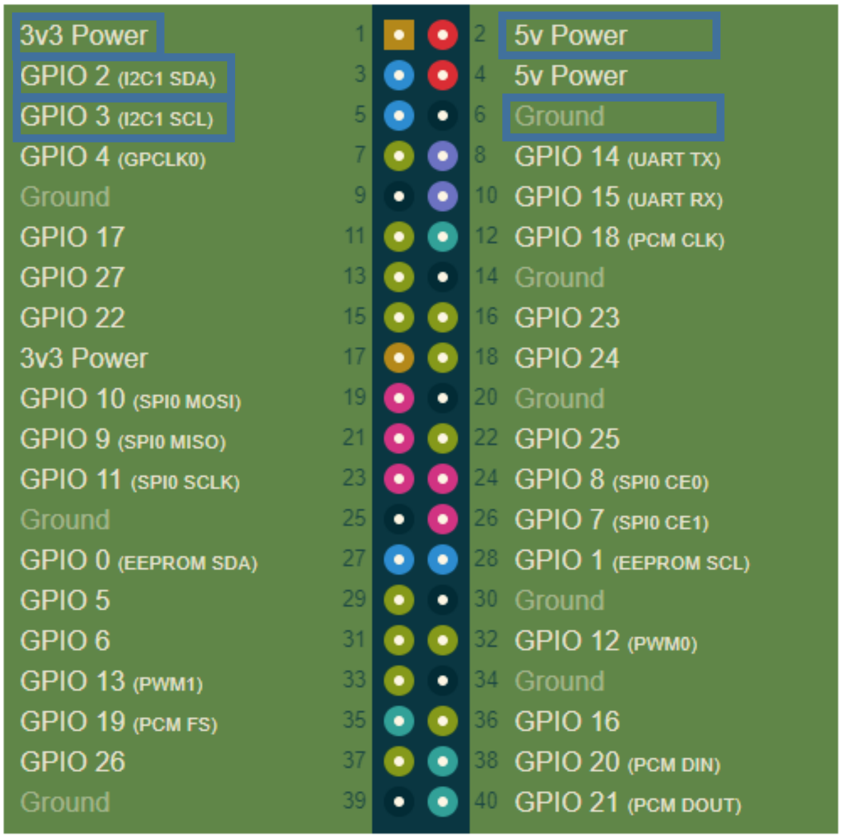
Development & Iterative Testing

Iterative Stage 1 – Relay

Requirement: This class must be able to control the 4 relays connected to the hat in the form of the Relay Hat. The class should be able to handle all combinations of relay states and allow for execution of other code whilst a relay is active.

Hardware: A 4 relay board is connected to the Raspberry Pi directly. This relay board features an LED for each of the relays which is on when the relay is active. This LED will be used for easy testing to determine if the relay is active or not. The relay communicates with the Pi using I2C and requires 3v3 power and 5v power. Figure 1 is a diagram of the GPIO pins that are used by the relay on the Pi. This board is useful as it still allows all the other GPIO pins to be accessed on the Pi. Each relay has a common connection in the middle and then a NC (normal close) connection and a NO (normal open) connection. To this project, we want a component to operate when the relay is active so we will be using only the common and the NC connection. It is intended that the relay will be connected to the Pump, Heating elements and the fan. The components will be connected as shown in figure 2. Another component the Enviro+ also communicates over the I2C pins using the same protocol. Providing the Relay and the Enviro+ are wired in series this will not be an issue as each device is given a 7bit address allowing up to 128 slave devices.

🟦 Relay



3v3 Power	1	2	5v Power
GPIO 2 (I2C1 SDA)	3	4	5v Power
GPIO 3 (I2C1 SCL)	5	6	Ground
GPIO 4 (GPCLK0)	7	8	GPIO 14 (UART TX)
Ground	9	10	GPIO 15 (UART RX)
GPIO 17	11	12	GPIO 18 (PCM CLK)
GPIO 27	13	14	Ground
GPIO 22	15	16	GPIO 23
3v3 Power	17	18	GPIO 24
GPIO 10 (SPI0 MOSI)	19	20	Ground
GPIO 9 (SPI0 MISO)	21	22	GPIO 25
GPIO 11 (SPI0 SCLK)	23	24	GPIO 8 (SPI0 CE0)
Ground	25	26	GPIO 7 (SPI0 CE1)
GPIO 0 (EEPROM SDA)	27	28	GPIO 1 (EEPROM SCL)
GPIO 5	29	30	Ground
GPIO 6	31	32	GPIO 12 (PWM0)
GPIO 13 (PWM1)	33	34	Ground
GPIO 19 (PCM FS)	35	36	GPIO 16
GPIO 26	37	38	GPIO 20 (PCM DIN)
Ground	39	40	GPIO 21 (PCM DOUT)

Figure 1

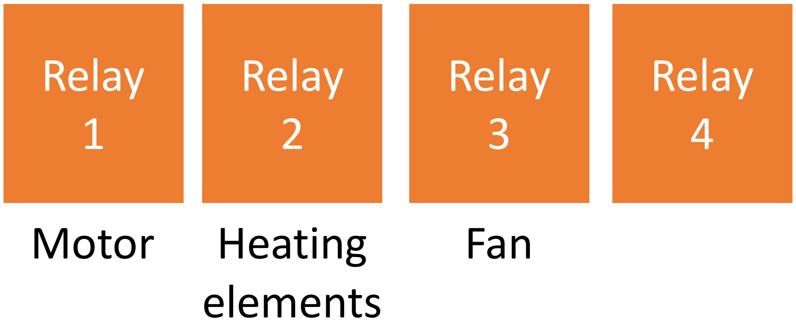
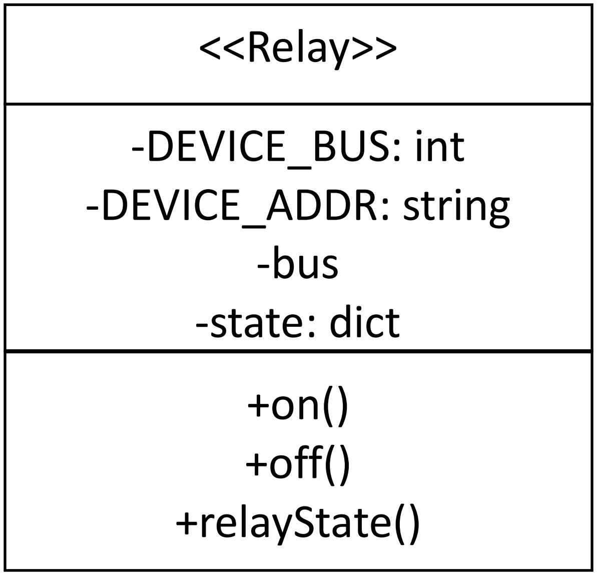


Figure 2

Configuring I2C
By default, the I2C protocol is not activated on the Raspberry Pi so I had to activate it using the steps found on the relay wiki page.

Class Diagram



The relayState() function will be used to return the current state of an individual relay (on/off) this will be key when developing the greenhouse system as it can be used to prevent potential issues such as trying to turn on an already active relay.

Pseudocode

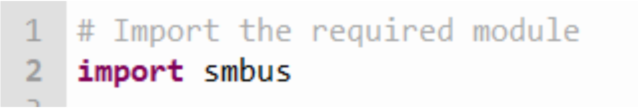
```
class Relay

   public procedure new()
      setup bus
      state = {1:False, 2:False, 3:False, 4:False}

   public procedure on(position)
      Turn relay on
      state[position] = True

   public procedure off(position)
     Turn relay off
     state[position] = False

   public procedure relayState()
      return state[position]

endclass
```

Data Structure	**Data Type**	**Scope**	**Purpose**	**Validation required**
DEVICE_BUS	Int	Local	The smbus class needs to know which bus is being used. The bus number is stored in this variable and later passed as a parameter	
DEVICE_ADDR	String	Local	The smbus class needs to know the I2C address being used for the relay. The address is stored here and passed as a parameter.	
Bus	Object	Local	An instance of the smbus class	
State	Dictionary	Local	The state dictionary is used to store the current state of each relay (on/off)	
Position	Int	Local	A parameter used to signify the	Range check 1 <= x <= 4

			desired relay to be communicated with. There are only 4 relays which are labeled 1,2,3,4. A range check must be carried out to make sure the value is not for a relay that does not exist.	

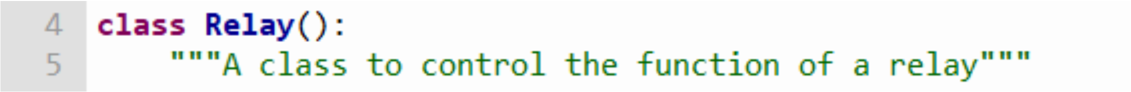
Development Log

The relay class is a small class that will form the backbone of a large part of the project activating and deactivating devices such as the pump. The class features one argument that must be validated to ensure that the class does not attempt to communicate with a relay that does not exist.



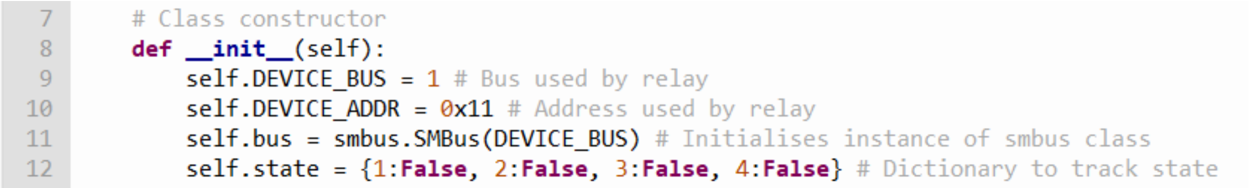
```python
# Import the required module
import smbus
```

To begin with I set out the module that will be needed in this class. The smbus module is used to communicate with devices over the I2C pins.



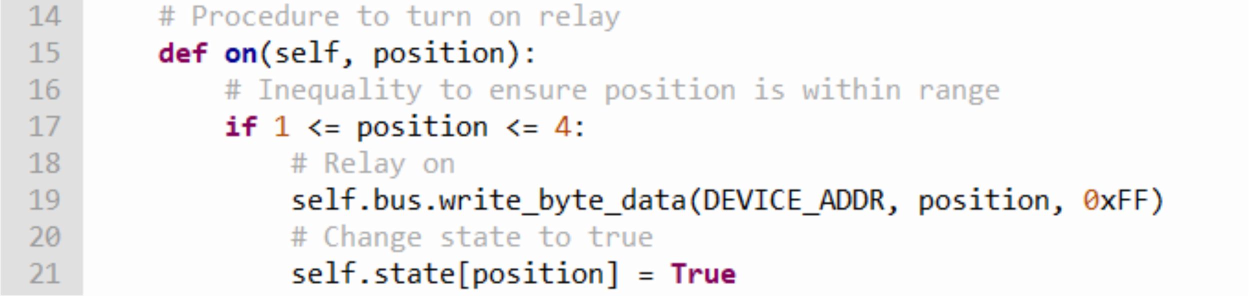
```python
class Relay():
    """A class to control the function of a relay"""
```

Next, I defined the Relay class and added a small docstring to briefly explain the function of this class.



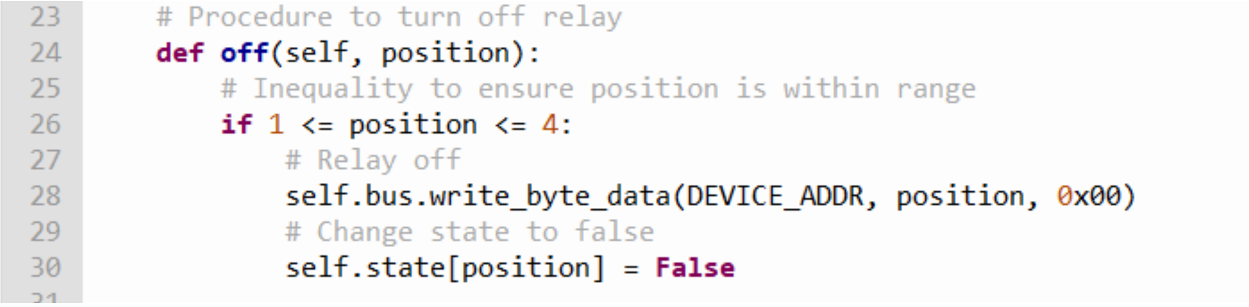
```python
    # Class constructor
    def __init__(self):
        self.DEVICE_BUS = 1 # Bus used by relay
        self.DEVICE_ADDR = 0x11 # Address used by relay
        self.bus = smbus.SMBus(DEVICE_BUS) # Initialises instance of smbus class
        self.state = {1:False, 2:False, 3:False, 4:False} # Dictionary to track state
```

The first three lines of this constructor concern the setup of the smbus object. DEVICE_ADDR refers to the address of the relay board and ensures the correct device receives the data this address can be changed on the relay board using a two-bit switch system. To this project the address will be set too 0x11. DEVICE_BUS signifies which bus is being used to communicate with the I2C devices. Finally, bus is used to create an instance of the smbus class with the correct DEVICE_BUS. The state dictionary is used to store the current state of each of the 4 relays. There is no way to check the current state of a relay on the board. So, it is assumed this class is initialized when all the relays are off. This is denoted by setting all keys in the dictionary to have a value of False.



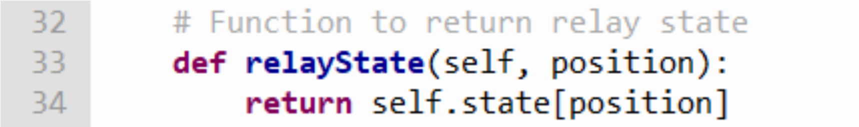
```
14      # Procedure to turn on relay
15      def on(self, position):
16          # Inequality to ensure position is within range
17          if 1 <= position <= 4:
18              # Relay on
19              self.bus.write_byte_data(DEVICE_ADDR, position, 0xFF)
20              # Change state to true
21              self.state[position] = True
```

The on procedure takes one parameter position. This relates to the number of the relay on the board. Each relay is numbered on the board beginning at 1 not 0 going up to 4. For example, a position of 3 would mean the class is changing the state of the 3rd relay on the board. When messing around with the relay I discovered that the relay is cyclical meaning that if you try to activate relay 5 a relay which does not exist this will turn on relay 3. This will cause unneeded issues if we do not validate the position to ensure we don't ever attempt to make use of the cyclical nature of the relay addressing. This parameter is validated to ensure it relates to one of the existing relays. To do this I have used an inequality. Providing the validation is passed a byte is written to instruct the desired relay to be activated. The write_data_byte procedure takes 3 arguments. The first two arguments DEVICE_ADDR and position have previously been explained. The third argument 0xFF is the register used by the board to indicate turning a relay on. After this we change the state of the relay in the state dictionary.



```
23      # Procedure to turn off relay
24      def off(self, position):
25          # Inequality to ensure position is within range
26          if 1 <= position <= 4:
27              # Relay off
28              self.bus.write_byte_data(DEVICE_ADDR, position, 0x00)
29              # Change state to false
30              self.state[position] = False
```

The off procedure is identical to the on procedure apart from the 3rd argument on line 28 which is 0x00 to signify turning off the relay.



```
32      # Function to return relay state
33      def relayState(self, position):
34          return self.state[position]
```

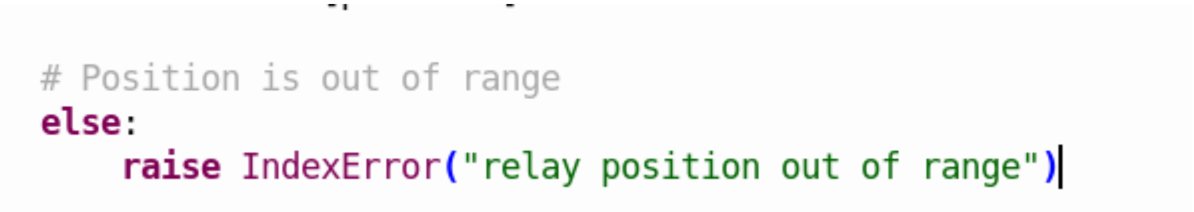
The intended use of this function is to check the state of a relay before it is interacted with. For example, there is no point attempting to turn on a relay that is already activated. A basic return statement is used to return true of false for the requested relay position.

Testing

Test Number	**Test Plan**	**Expected Outcome**	**Actual Outcome**	**Pass/Fail**
1	Attempt to activate each of the relays individually.	The relay should turn on this will be shown by the blue light on the board turning on.	Each relay activated as expected	Pass

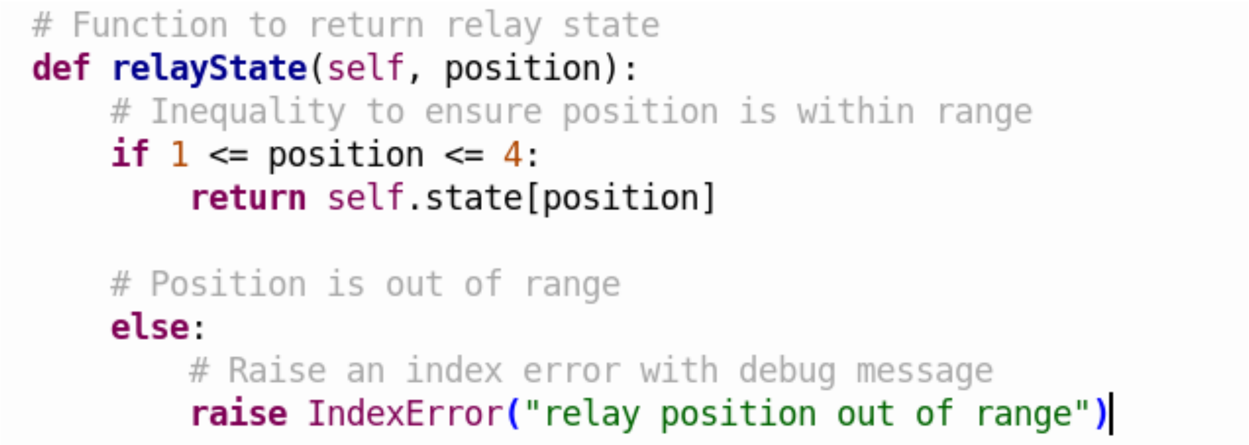
2	Attempt to deactivate each of the relays individually.	The light on the corresponding relay should go out to signify the relay is off.	Each relay turned off as expected	Pass
3	Have multiple relays active at the same time.	The relays should turn on and not be affected by activating a different relay.	The relays stayed on and behaved as expected.	Pass
4	Try to activate a relay position that does not exist such as 5.	The program should not throw an error and no relay should be activated.	The program continued to function, and no relay was activated.	Pass
5	Get the status of a relay.	True should be returned if the relay is active and false if the relay is off.	If the relay was active, then true was returned and the opposite for an inactive relay.	Pass
6	Get the status of a relay that is out of range.	The program should produce an error.	The program produced a key error.	Pass (See notes below)

Whilst all tests were passed, I have decided to modify the code so that when a relay position that is out of range is requested an index error occurs. This will lead to stronger code that is more robust and easier to debug. To do this I will need to modify the On, Off and Relay State class.



```
# Position is out of range
else:
    raise IndexError("relay position out of range")
```

For the on and off class I just had to expand my if statement to have an else for when a position out of range has been entered. In this scenario an exception is raised with a helpful message to help with debugging purposes.



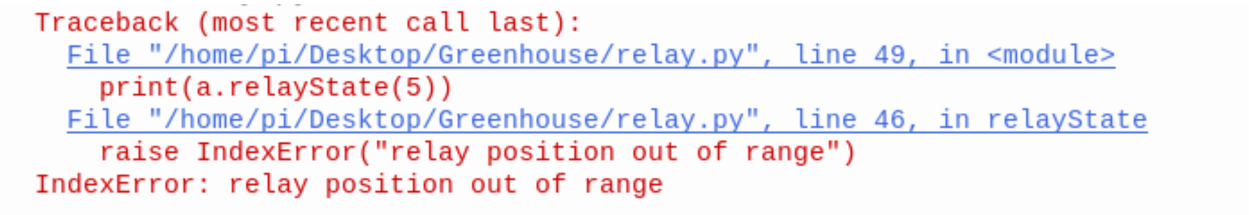
```
# Function to return relay state
def relayState(self, position):
    # Inequality to ensure position is within range
    if 1 <= position <= 4:
        return self.state[position]

    # Position is out of range
    else:
        # Raise an index error with debug message
        raise IndexError("relay position out of range")
```

The process was largely the same for the relayState function however I also added an inequality condition to ensure the exception would be raised by my code and not when attempting to access an index out of range in the state list.

Below is the updated testing plan to reflect the changes made to the code. The only two tests that needed to be amended was test number 4 and 6. Instead of before where the program was expected to continue as usual if a relay out of range was entered the program should now raise the exception.

Test Number	Test Plan	Expected Outcome	Actual Outcome	Pass/Fail
4	Try to activate a relay position that does not exist such as 5.	The program should throw an error and no relay should be activated.	The program produced an error, and no relay was activated	Pass
6	Get the status of a relay that is out of range.	The program should produce an index error.	The program produced a key error.	Pass

Figure 3 shows the error that is produced when a relay position out of range is entered as a parameter.



```
Traceback (most recent call last):
  File "/home/pi/Desktop/Greenhouse/relay.py", line 49, in <module>
    print(a.relayState(5))
  File "/home/pi/Desktop/Greenhouse/relay.py", line 46, in relayState
    raise IndexError("relay position out of range")
IndexError: relay position out of range
```

Figure 3

Bugs encountered during testing

When the code was run for the first time there were a few errors where I had forgotten to add self before variables related to that object. This was quickly fixed by amending the code.

Review
In this first iteration I have developed a robust Relay class to handle the function of the relay board that will be used in my project. The inclusion of raising errors will help with debugging later in development. This class will allow me to begin to develop children classes that can control different hardware devices such as the fan.

Source https://wiki.52pi.com/index.php/DockerPi_4_Channel_Relay_SKU:_EP-0099

Iterative Stage 2 – Servo

Requirement: In this iterative stage I will be developing the servo class. The servo motor will be used to open and close the window in the green house. The servo needs to be moved across an angle of 45 degrees from a vertical to a horizontal position to open and close the window. In initial testing the servo motor would jitter a solution to this will need to be produced.

Hardware: The Tower Pro SG51R servo being used for this project has three wires. A positive (red) wire, a neutral (black) wire and a data wire. As the servo does not require much power, I will be using the Raspberry Pis own 5v power pin and ground to power the servo. Figure 4 shows the GPIO pins that will be used by the servo. Whilst figure 5 shows the currently used GPIO pins including the previous iterative stages.

🟥	Servo

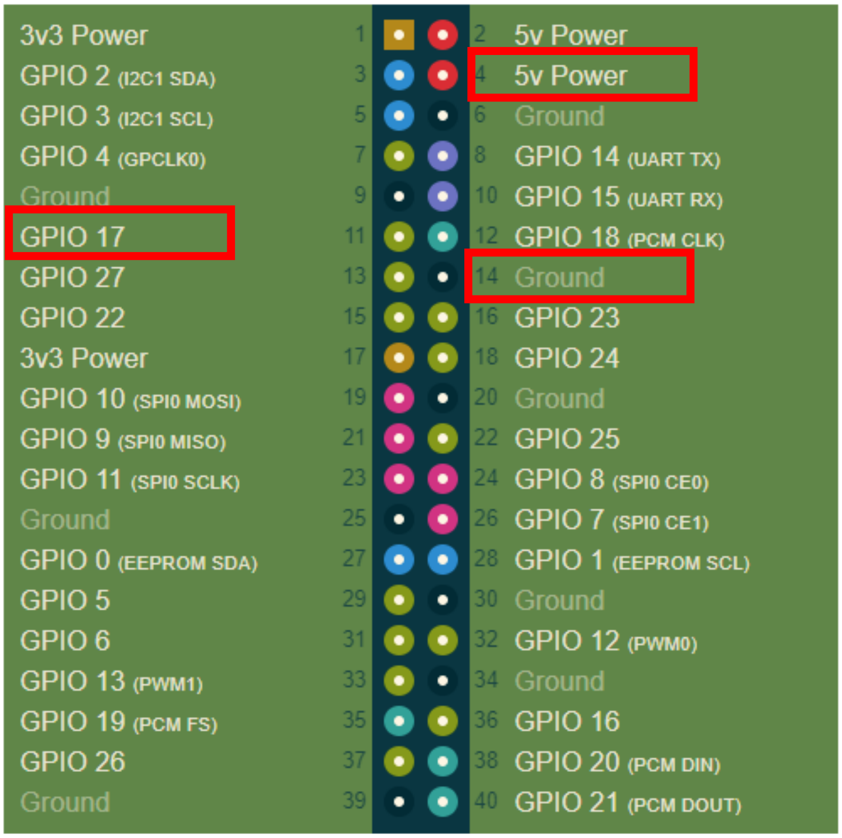
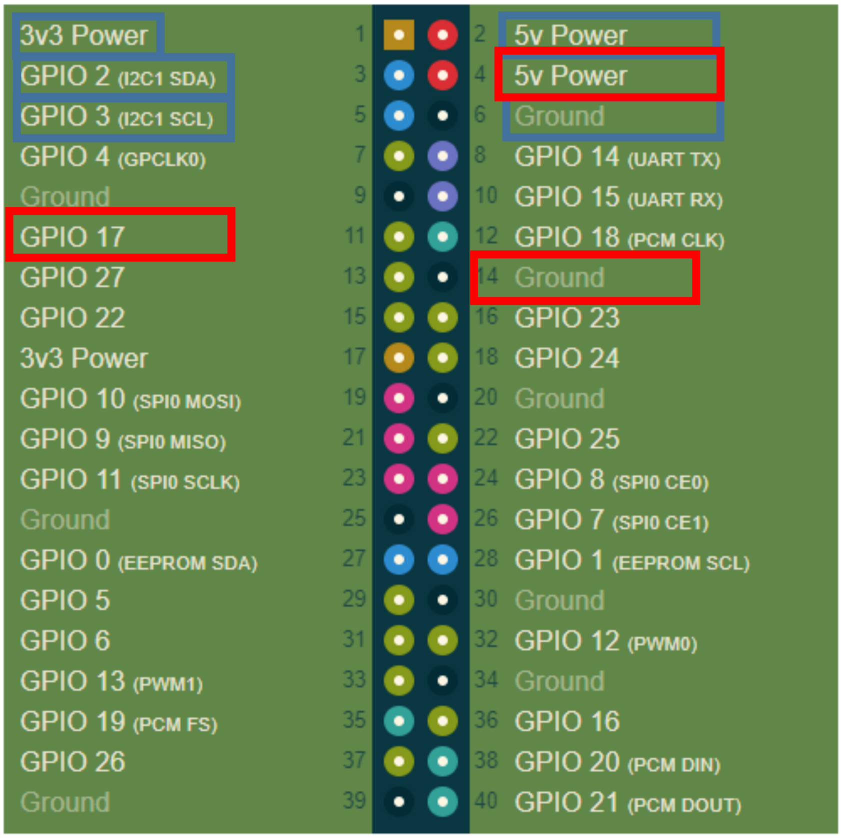

3v3 Power 1 2 5v Power
GPIO 2 (I2C1 SDA) 3 4 5v Power
GPIO 3 (I2C1 SCL) 5 6 Ground
GPIO 4 (GPCLK0) 7 8 GPIO 14 (UART TX)
Ground 9 10 GPIO 15 (UART RX)
GPIO 17 11 12 GPIO 18 (PCM CLK)
GPIO 27 13 14 Ground
GPIO 22 15 16 GPIO 23
3v3 Power 17 18 GPIO 24
GPIO 10 (SPI0 MOSI) 19 20 Ground
GPIO 9 (SPI0 MISO) 21 22 GPIO 25
GPIO 11 (SPI0 SCLK) 23 24 GPIO 8 (SPI0 CE0)
Ground 25 26 GPIO 7 (SPI0 CE1)
GPIO 0 (EEPROM SDA) 27 28 GPIO 1 (EEPROM SCL)
GPIO 5 29 30 Ground
GPIO 6 31 32 GPIO 12 (PWM0)
GPIO 13 (PWM1) 33 34 Ground
GPIO 19 (PCM FS) 35 36 GPIO 16
GPIO 26 37 38 GPIO 20 (PCM DIN)
Ground 39 40 GPIO 21 (PCM DOUT)

Figure 4

- Pin 4 – Servo Red

- Pin 14 – Servo Black
- Pin 11 – Servo Data
-

🟦	Relay
🟥	Servo



3v3 Power	1	2	5v Power
GPIO 2 (I2C1 SDA)	3	4	5v Power
GPIO 3 (I2C1 SCL)	5	6	Ground
GPIO 4 (GPCLK0)	7	8	GPIO 14 (UART TX)
Ground	9	10	GPIO 15 (UART RX)
GPIO 17	11	12	GPIO 18 (PCM CLK)
GPIO 27	13	14	Ground
GPIO 22	15	16	GPIO 23
3v3 Power	17	18	GPIO 24
GPIO 10 (SPI0 MOSI)	19	20	Ground
GPIO 9 (SPI0 MISO)	21	22	GPIO 25
GPIO 11 (SPI0 SCLK)	23	24	GPIO 8 (SPI0 CE0)
Ground	25	26	GPIO 7 (SPI0 CE1)
GPIO 0 (EEPROM SDA)	27	28	GPIO 1 (EEPROM SCL)
GPIO 5	29	30	Ground
GPIO 6	31	32	GPIO 12 (PWM0)
GPIO 13 (PWM1)	33	34	Ground
GPIO 19 (PCM FS)	35	36	GPIO 16
GPIO 26	37	38	GPIO 20 (PCM DIN)
Ground	39	40	GPIO 21 (PCM DOUT)

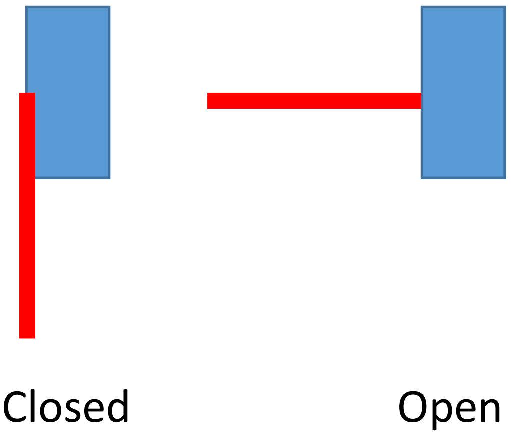
Figure 5

Class Diagram

Window
-position: bool

+open()
+close()
+getPosition()

Open() will set the servo to an angle of 45 degrees and Close() will lower the angle of the servo to 0 degrees from the horizontal. The two diagrams below show the position of the servo for each method. The getPosition() function that will return the current position of the servo.

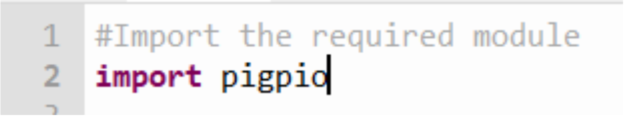


Data Structure	Data Type	Scope	Purpose	Validation required
Position	Int	Local	Store the current position of the servo. This will be returned in the getPosition function	

Development Log

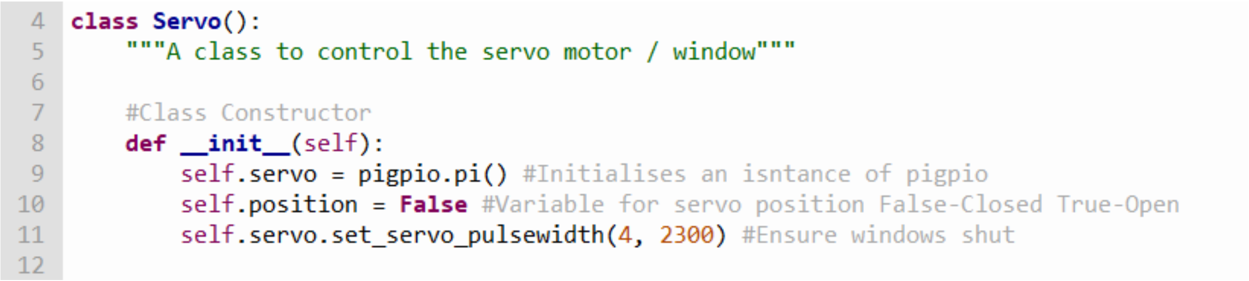
When I was learning how to use the servo, I was initially using the gpiozero module to control the servo. This all worked nicely but there was an issue, once the code had executed the servo would jitter in the position it had been set too. After some research I was able to work out this was because the Raspberry Pi is a fully-fledged computer rather than a microcontroller. Meaning it lacked the ability to maintain a smooth data signal whilst performing other tasks. The solution is to use a ported low-level library called pigpio. This library allows the Pi to produce a smooth data signal and eliminates the jitter. The disadvantages are that it controls the servo based on a pulse width rather than a straight up angle. This will require some code to translate between the desired angle and the pulsewidth. Another issue is that before this library can be used a pigpio daemon must be started. To do this I have added the following line to the raspberry pis crontab file. This means that the daemon is executed on start up and ready for use in python.

```
@reboot              /usr/local/bin/pigpiod
```



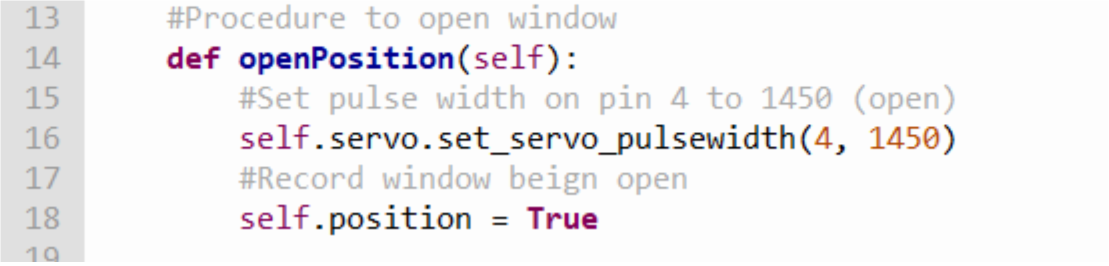
```python
#Import the required module
import pigpio
```

There is just one library that is needed for the servo class which is pigpio. This allows the Pi to communicate with the servo motor without any jitter.



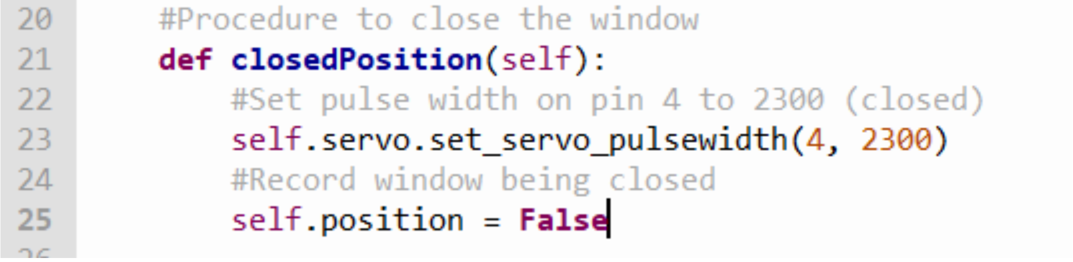
```python
class Servo():
    """A class to control the servo motor / window"""

    #Class Constructor
    def __init__(self):
        self.servo = pigpio.pi() #Initialises an isntance of pigpio
        self.position = False #Variable for servo position False-Closed True-Open
        self.servo.set_servo_pulsewidth(4, 2300) #Ensure windows shut

```

When initializing the servo class an instance of pigpio needs to be created. In my code I have assigned this to the name servo. Next the position of the servo is recorded as closed this variable is used to record the current position of the window and is returned during the getPosition function. Finally, the servo pulsewidth is set to 2300 on GPIO pin 17. This is a precautionary step to ensure that the window is always closed when the class is initialized. There is a scenario where the code could crash leaving the window stuck open so this just accounts for that eventuality when the system is restarted.



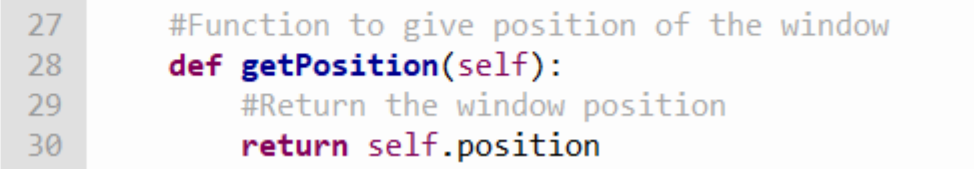
```
    #Procedure to open window
    def openPosition(self):
        #Set pulse width on pin 4 to 1450 (open)
        self.servo.set_servo_pulsewidth(4, 1450)
        #Record window beign open
        self.position = True
```

The openPosition procedure is responsible for opening the window. Originally it was too be called open, but this is already a function in python, so I thought it best to change the name and avoid any naming related bugs. The set_servo_pulsewidth command takes two parameters the first is the pin that the pulse will be broadcast on in this case pin 17. The second parameter is the width of the pulse from experimentation 1450 is the pulse that moves the servo to a horizontal open position. Finally, the position is recorded to be open.



```
    #Procedure to close the window
    def closedPosition(self):
        #Set pulse width on pin 4 to 2300 (closed)
        self.servo.set_servo_pulsewidth(4, 2300)
        #Record window being closed
        self.position = False
```

The closedPosition procedure is identical to the openPosition procedure with the only difference being that the width is 2300 which corresponds to a vertical closed position on the servo and the position recorded as being closed. In this class a position of True corresponds to the window being open and False meaning closed.



```
    #Function to give position of the window
    def getPosition(self):
        #Return the window position
        return self.position
```

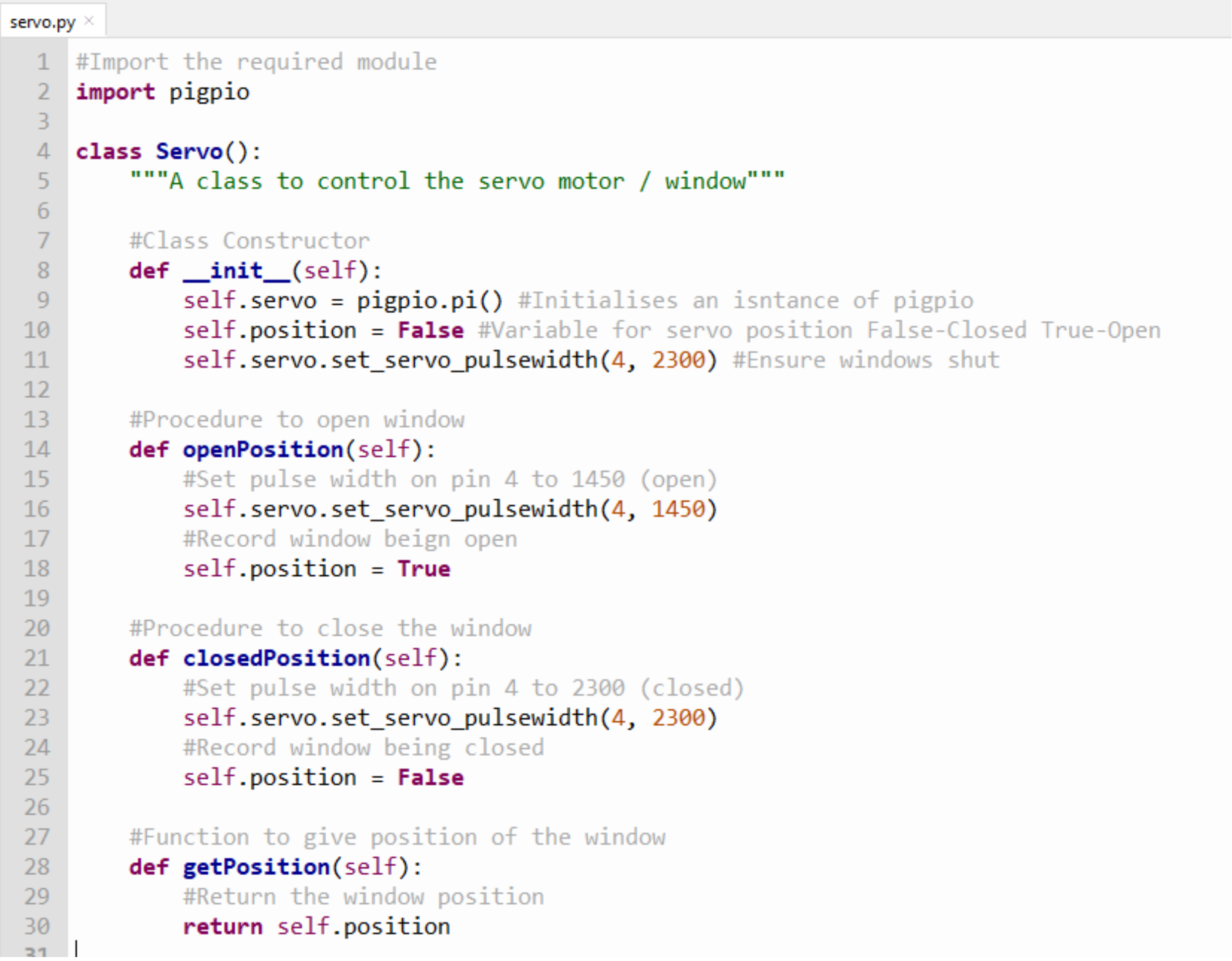
The getPosition function is very simple and just returns the position variable which relates to the current position of the window.

Testing

Test Number	**Test Plan**	**Expected Outcome**	**Actual Outcome**	**Pass/Fail**
1	Open the window	Window moves to open position	The window opened as expected	Pass
2	Close the window	Window moves to the closed position	The window closed as expected	Pass

3	When the window is open run the getPosition function	True will be returned to indicate that the window is open	True was returned	Pass
4	When the window is shut run the getPosition function	False will be returned to indicate a closed window	False was returned	Pass
5	Open the window and then open it again	The window should not move and just stay open	The window did not move	Pass
6	Close the window and then close it again	The window should stay shut	The window stayed shut	Pass

Full Code



```python
#Import the required module
import pigpio

class Servo():
    """A class to control the servo motor / window"""

    #Class Constructor
    def __init__(self):
        self.servo = pigpio.pi() #Initialises an isntance of pigpio
        self.position = False #Variable for servo position False-Closed True-Open
        self.servo.set_servo_pulsewidth(4, 2300) #Ensure windows shut

    #Procedure to open window
    def openPosition(self):
        #Set pulse width on pin 4 to 1450 (open)
        self.servo.set_servo_pulsewidth(4, 1450)
        #Record window beign open
        self.position = True

    #Procedure to close the window
    def closedPosition(self):
        #Set pulse width on pin 4 to 2300 (closed)
        self.servo.set_servo_pulsewidth(4, 2300)
        #Record window being closed
        self.position = False

    #Function to give position of the window
    def getPosition(self):
        #Return the window position
        return self.position
```

Review

The servo class is now complete and can be used to control the window of the greenhouse.

Note – Later in development I have realized that the GPIO 4 pin is used by the enviro. This has required me to swap the Servo Data line to GPIO pin 17. I have updated the text to reflect this, but the code screenshots don’t show this change.

Source - http://abyz.me.uk/rpi/pigpio/

Iterative stage 3 – LED strip

Requirements

The greenhouse features a strip of 60 leds attached to the roof. These can be controlled individually and given unique rbg values. The job of the leds is to provide the plans with light energy for photosynthesis. The user will later be able to select the exact type of light the plans receive such as white light or only blue light. This class will need to have the ability to turn all the led strip one colour, turn off the led strip and at the request of one of my stake holders I will produce two entertainment modes where a rainbow is shown and another where a disco light randomly changes the light every so often.

Hardware

The led strips require 5v power with a decent current to operate properly. In my project I am using a 5v 2A power supply to power the LEDS. The leds can function incorrectly if they do not share a common ground wire with the Raspberry Pi and the power supply. This means the pi is directly wired to the power supply so I will be using a diode to isolate the Pi from the power supply and protect the gpio pins. The makers of the LEDS Adafruit recommend that the data wire is a 5v signal but in my testing, I’ve not had any issues using the Pis 3.3v gpios without any logic level shifting. Below if the wiring diagram for the leds and the raspberry pi. The only difference being that I will be using GPIO 12 not 18 for the LED data due to other pin requirements in my project. Below is also the gpio pins now in use in my project.

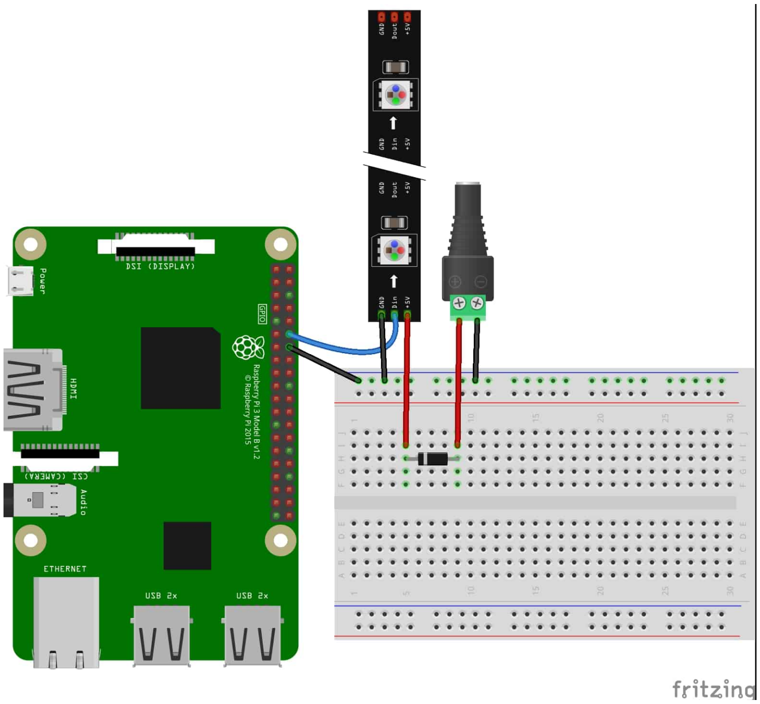
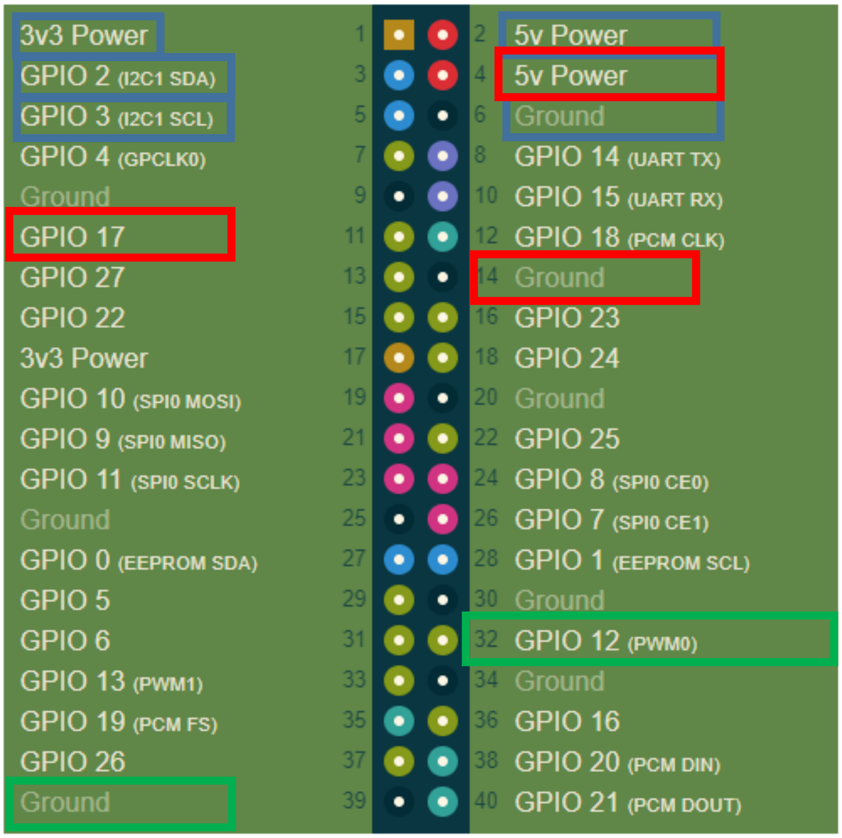


Figure 6

🟦	Relay
🟥	Servo
🟩	LEDs

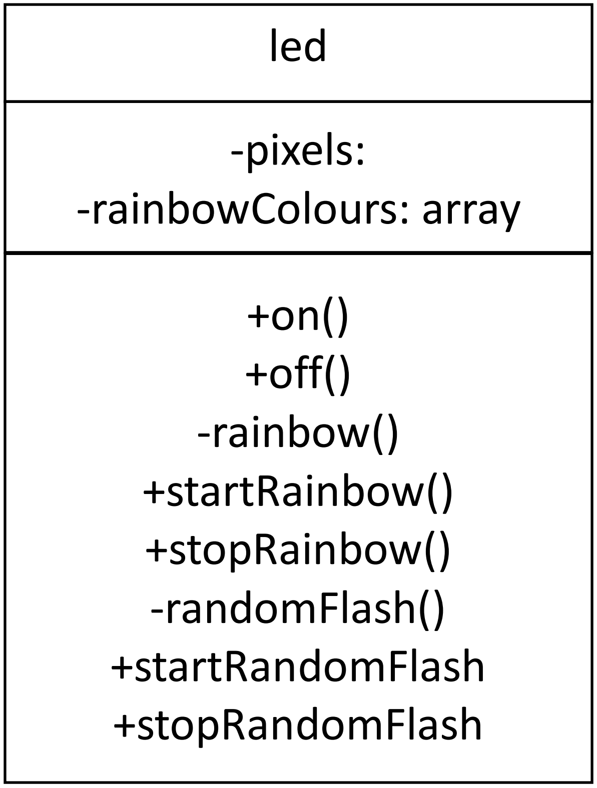


3v3 Power	1	2	5v Power
GPIO 2 (I2C1 SDA)	3	4	5v Power
GPIO 3 (I2C1 SCL)	5	6	Ground
GPIO 4 (GPCLK0)	7	8	GPIO 14 (UART TX)
Ground	9	10	GPIO 15 (UART RX)
GPIO 17	11	12	GPIO 18 (PCM CLK)
GPIO 27	13	14	Ground
GPIO 22	15	16	GPIO 23
3v3 Power	17	18	GPIO 24
GPIO 10 (SPI0 MOSI)	19	20	Ground
GPIO 9 (SPI0 MISO)	21	22	GPIO 25
GPIO 11 (SPI0 SCLK)	23	24	GPIO 8 (SPI0 CE0)
Ground	25	26	GPIO 7 (SPI0 CE1)
GPIO 0 (EEPROM SDA)	27	28	GPIO 1 (EEPROM SCL)
GPIO 5	29	30	Ground
GPIO 6	31	32	GPIO 12 (PWM0)
GPIO 13 (PWM1)	33	34	Ground
GPIO 19 (PCM FS)	35	36	GPIO 16
GPIO 26	37	38	GPIO 20 (PCM DIN)
Ground	39	40	GPIO 21 (PCM DOUT)

Figure 7

- Ground 39 – Led black
- GPIO 12 – Led data blue

Class Diagram



led

-pixels:
-rainbowColours: array

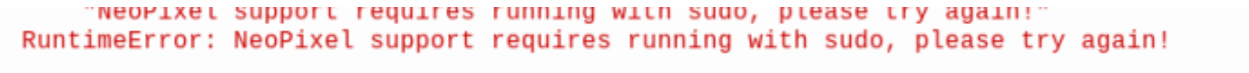
+on()
+off()
-rainbow()
+startRainbow()
+stopRainbow()
-randomFlash()
+startRandomFlash
+stopRandomFlash

- On() – fills the led strip with a given colour
- Off() – turns the led strip off (effectively the same as on but with rgb values 0,0,0)
- Rainbow() – Produces the rainbow effect on the leds
- StartRainbow() – Begins a rainbow thread to allow for concurrent processing
- StopRainbow() – Stop the rainbow thread using a start/stop flag
- RandomFlash() – Randomly changes the leds to a colour of the rainbow at set intervals
- StartRandomFlash() – Begins the random flash thread
- StopRandomFlash – Closes the random flash thread using a flag

Data Structure	Data Type	Scope	Purpose	Validation required
rainbowColours	Array	Local	To store the rgb color values of the rainbow	

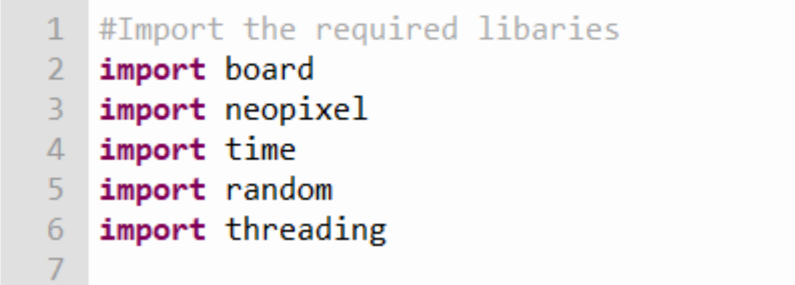
Software considerations

Filling the led strip with one color is straight forward using the fill function in the neopixels library. This will set the strip to a desired color and the strip will stay that way until told otherwise. However, when producing more complex patterns of LEDs such as the rainbow snake the leds need to be constantly updated. This means whilst running the snake or the flash procedures no other computation can be carried out in python. So, it won’t be possible to run the leds in this way and continue the other functions of the greenhouse. To overcome this issue, I will be using threading to allow me to run concurrent python processes. The neopixel also requires that it is launched with sudo privileges meaning the greenhouse will need to be launched from command line. Below is the error produced when not ran from command line using “sudo python3 led.py”.



```
    "NeoPixel support requires running with sudo, please try again!"
RuntimeError: NeoPixel support requires running with sudo, please try again!
```

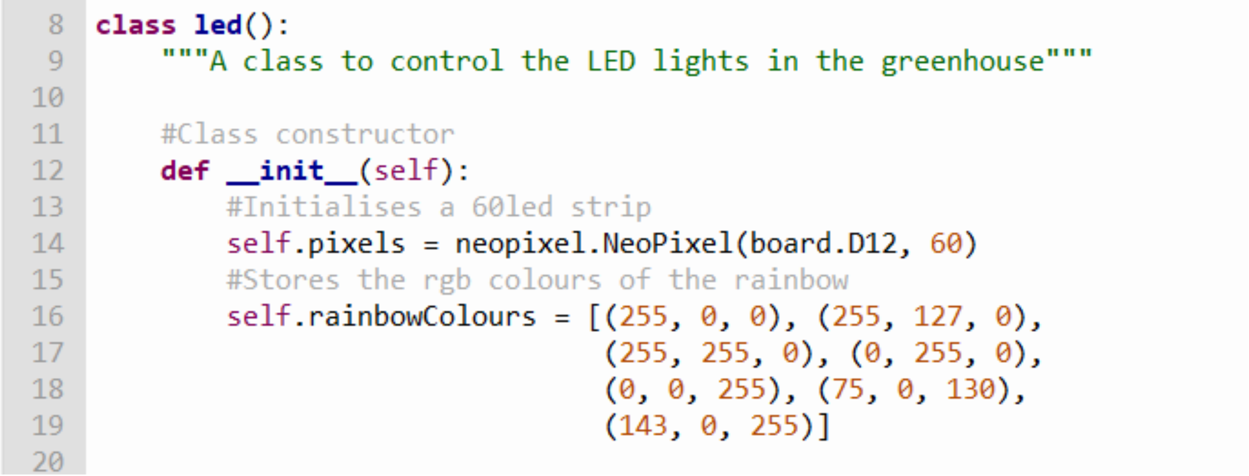
Development log



```
#Import the required libaries
import board
import neopixel
import time
import random
import threading

```

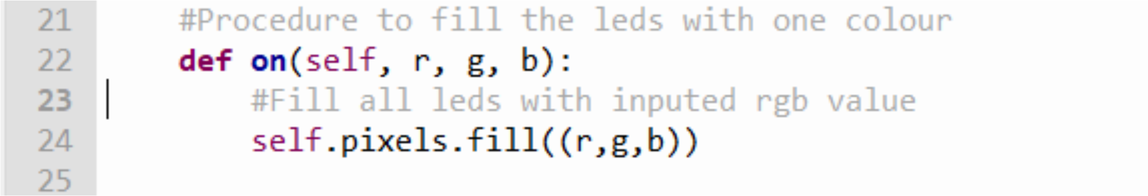
This class requires a couple of libraries to be imported each performing a different task

- Board – Allows the neopixel library to talk to the GPIO pins
- Neopixel – A library that allows python to control led strips
- Time – Used to change the speed of the leds changing in the rainbow and flash procedures
- Random – Used to select a random item from the rainbow colors array
- Threading – Provides the ability to do concurrent processing in python via threads



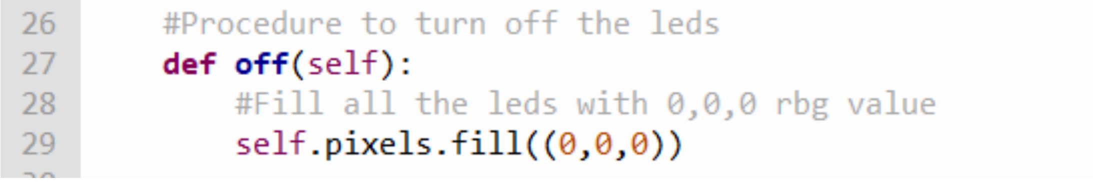
```
class led():
    """A class to control the LED lights in the greenhouse"""

    #Class constructor
    def __init__(self):
        #Initialises a 60led strip
        self.pixels = neopixel.NeoPixel(board.D12, 60)
        #Stores the rgb colours of the rainbow
        self.rainbowColours = [(255, 0, 0), (255, 127, 0),
                               (255, 255, 0), (0, 255, 0),
                               (0, 0, 255), (75, 0, 130),
                               (143, 0, 255)]

```

Inside the class constructor I have set up the neopixel strip to have 60 leds and to communicate over GPIO pin 12. For some reason the neopixel library only works on 4 select pins so my choice of pins was dictated by this and the requirements of the enviro hat. I have also declared a rainbowColours array which contains the 7 main colors of the rainbow in RBG form in order. I will use this later to loop over or make a random selection from.



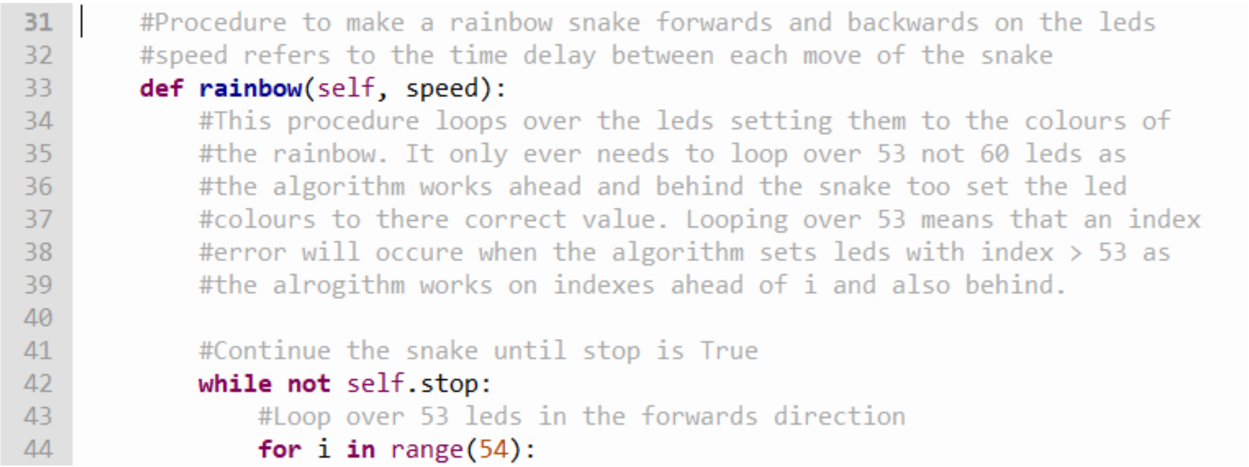
```
    #Procedure to fill the leds with one colour
    def on(self, r, g, b):
        #Fill all leds with inputed rgb value
        self.pixels.fill((r,g,b))

```

Turning the led strip on in one color is straight forward and just requires the use of the fill procedure and a rgb value to be passed. This procedure does not require any threading as once the leds are filled they will maintain this color until another command is sent or power on the strip is lost. Meaning I can continue to execute my python code normally without having to go back and continually update the led strip.



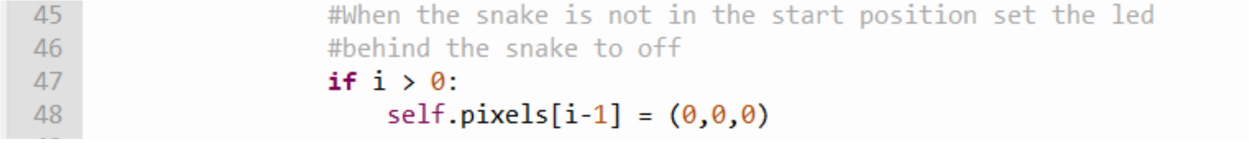
```
26      #Procedure to turn off the leds
27      def off(self):
28          #Fill all the leds with 0,0,0 rbg value
29          self.pixels.fill((0,0,0))
```

The off procedure works by filling the pixel strip with a rgb value of (0, 0 , 0) this achieves the aim of turning off the leds. I considered just calling the on procedure inside the off procedure and passing the parameter (0, 0, 0) to achieve the same effect but decided against it as this made code less readable and more memory intensive.



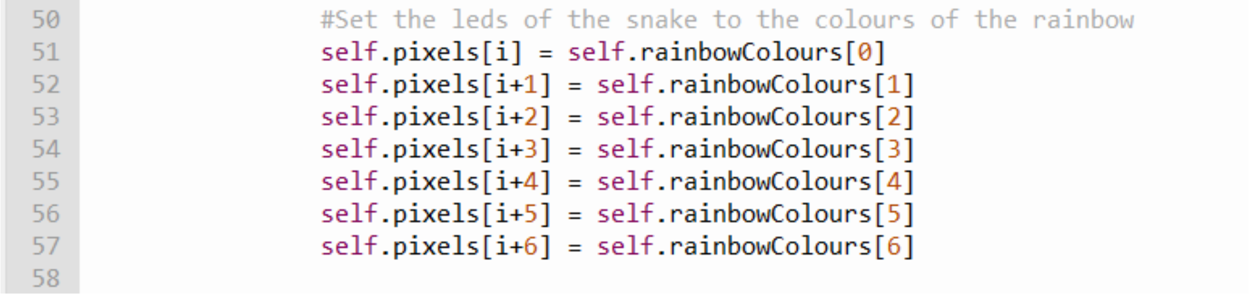
```
31      #Procedure to make a rainbow snake forwards and backwards on the leds
32      #speed refers to the time delay between each move of the snake
33      def rainbow(self, speed):
34          #This procedure loops over the leds setting them to the colours of
35          #the rainbow. It only ever needs to loop over 53 not 60 leds as
36          #the algorithm works ahead and behind the snake too set the led
37          #colours to there correct value. Looping over 53 means that an index
38          #error will occure when the algorithm sets leds with index > 53 as
39          #the alrogithm works on indexes ahead of i and also behind.
40
41          #Continue the snake until stop is True
42          while not self.stop:
43              #Loop over 53 leds in the forwards direction
44              for i in range(54):
```

The rainbow procedure produces a snake of 7 unique colours from the rainbowColours array that begins at the start of the led strip and progresses down the strip shifting forward 1 led at a time until it reaches the end of the stip. At this point the process is reversed, and the snake is moved back to the start. The while loop on line 42 means the snake will continue until the stopRainbow procedure changes the flag too true. Although there are 60 leds in the strip the algorithm only needs to loop over 53 of them as I work ahead of, I too set the rest of the snake. If the loop went all the way too 60 then an index error would occur when trying to set the i+1 led too its rgb value. The parameter speed is used to speed up or slow down the progression of the snake along the strip.



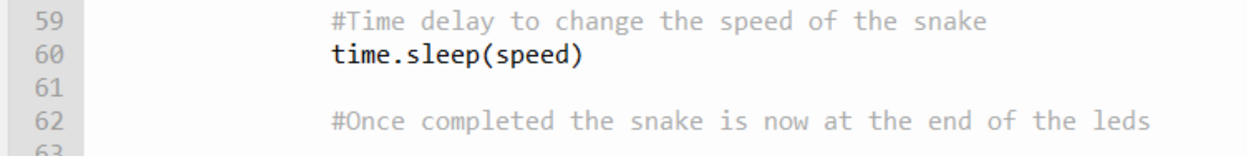
```
45                  #When the snake is not in the start position set the led
46                  #behind the snake to off
47                  if i > 0:
48                      self.pixels[i-1] = (0,0,0)
```

As the snake progresses along the strip the led trailing the snake needs to be set back to off otherwise a trail of red is left behind the snake as this is the color at the start of the snake. So, in the forward direction case when the snake has moved at least 1 led the led trailing the snake is set too off. Without the if statement the snake would begin with i = 0 and then attempt to set led position -1 to off and cause an index error.



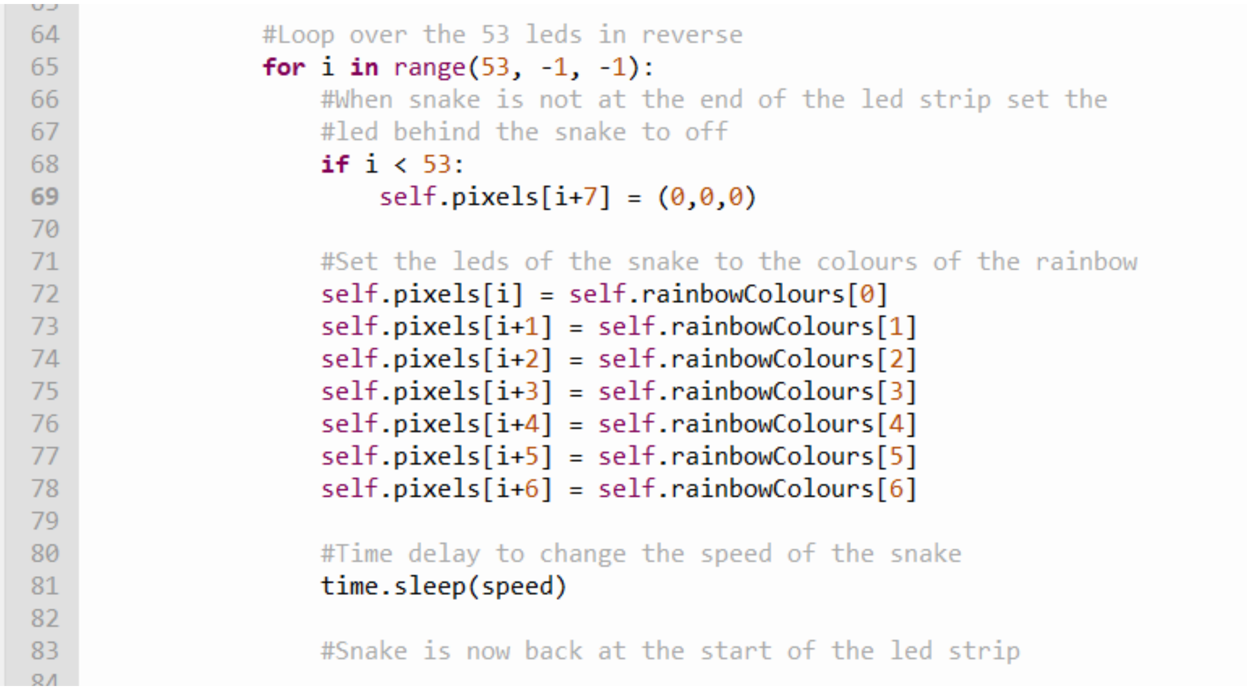
```
50                  #Set the leds of the snake to the colours of the rainbow
51                  self.pixels[i] = self.rainbowColours[0]
52                  self.pixels[i+1] = self.rainbowColours[1]
53                  self.pixels[i+2] = self.rainbowColours[2]
54                  self.pixels[i+3] = self.rainbowColours[3]
55                  self.pixels[i+4] = self.rainbowColours[4]
56                  self.pixels[i+5] = self.rainbowColours[5]
57                  self.pixels[i+6] = self.rainbowColours[6]
58
```

Now the leds of the snake are set. The current i value is the start of the snake and is set to the first value of the rainbowColour array. Then the rest of the leds ahead of the snake start are set moving +1 each time ahead in the pixels index and the rainbowColours array. After this code has been run the snake is shown on the led strip.



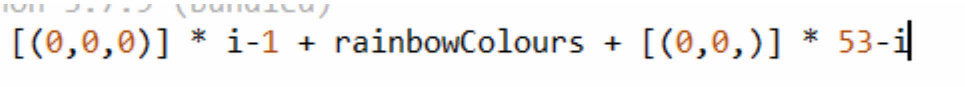
```
59                  #Time delay to change the speed of the snake
60                  time.sleep(speed)
61
62                  #Once completed the snake is now at the end of the leds
```

Line 60 puts a delay into the rainbow snake loop. This has the visual effect of slowing the snakes speed moving along the strip as the next iteration of the loop which moves the snake onwards 1 position won't happen until after this delay. At the end of the first loop of range(54) the snake will have reached the end of the led stip. With the start of the snake sitting 7 pixels back from the end of the strip and the final pixel of the strip being the final colour from the rainbowColours array.



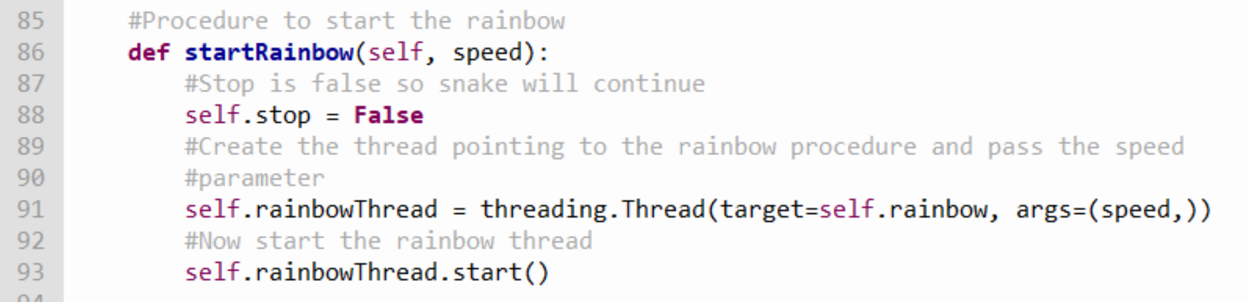
```
64              #Loop over the 53 leds in reverse
65              for i in range(53, -1, -1):
66                  #When snake is not at the end of the led strip set the
67                  #led behind the snake to off
68                  if i < 53:
69                      self.pixels[i+7] = (0,0,0)
70
71                  #Set the leds of the snake to the colours of the rainbow
72                  self.pixels[i] = self.rainbowColours[0]
73                  self.pixels[i+1] = self.rainbowColours[1]
74                  self.pixels[i+2] = self.rainbowColours[2]
75                  self.pixels[i+3] = self.rainbowColours[3]
76                  self.pixels[i+4] = self.rainbowColours[4]
77                  self.pixels[i+5] = self.rainbowColours[5]
78                  self.pixels[i+6] = self.rainbowColours[6]
79
80                  #Time delay to change the speed of the snake
81                  time.sleep(speed)
82
83                  #Snake is now back at the start of the led strip
```

The process is now reversed to move the rainbow snake back to the start of the led strip. Only two things need to be changed to the first loop to do this. Firstly, the parameters of range and changed to loop from 53 down too 0. This means the start of the snake moves towards the start of the strip with each iteration. The other change is that the condition in the if statement changes to make sure the snake is not at the end of the strip and attempts to set led index 61 too off. The rainbow snake is now complete and running this procedure on its own will cause the snake too continually move from the start to the end and back again. When developing this procedure, I thought about different approaches to this problem. At its core the problem is how to move a fixed sequence of 7 values down an array of 60 items and back again. Possible solutions were to implement a circular queue type algorithm to do this however this would only work if I was happy to accept the 7 values also rotating each iteration as the front led color would be popped from the queue and then pushed to the end of the queue. Meaning the colors do a loop of their own constantly changing order. Another idea I had was using list comprehension too produce an array of 60 rgb values with the following structure.



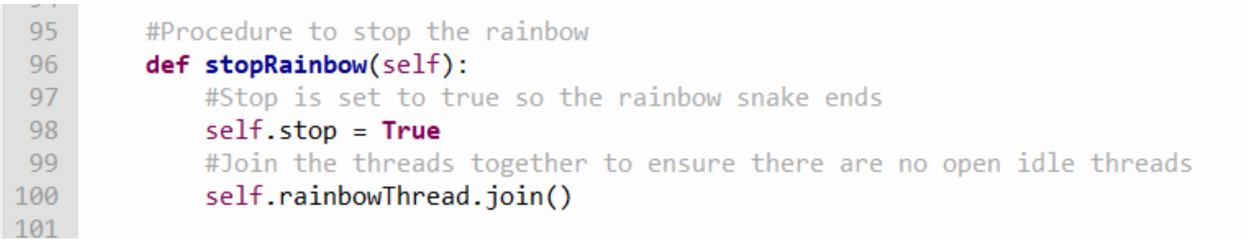
```
[(0,0,0)] * i-1 + rainbowColours + [(0,0,)] * 53-i
```

And then looping through it setting each led value in the strip too the corresponding rgb value from the generated array. It was a close call between using this method and the method I implemented but, in the end, I elected to not go with this method as it would require a lot of iteration when mapping the rgb array too the led strip. As already alluded to the issue is that python will always be inside this loop and never can complete any other functions such as check the temperature or water the plant. So, the options are solving this issue and find a way to do concurrent processing in python or have it so the greenhouse can only light the plans when not doing anything else.



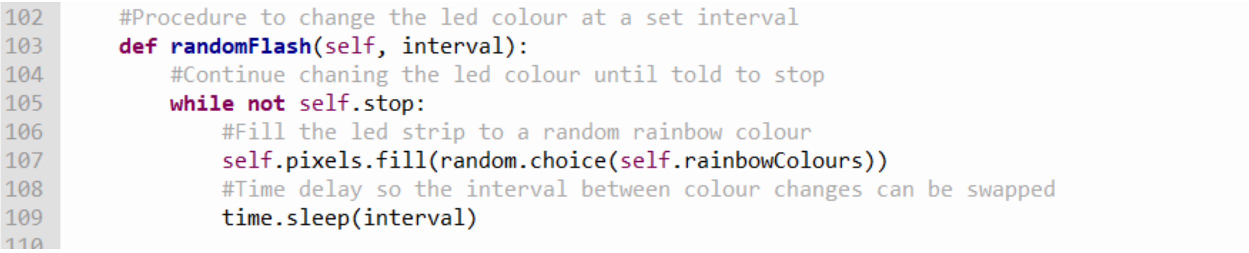
```
85      #Procedure to start the rainbow
86      def startRainbow(self, speed):
87          #Stop is false so snake will continue
88          self.stop = False
89          #Create the thread pointing to the rainbow procedure and pass the speed
90          #parameter
91          self.rainbowThread = threading.Thread(target=self.rainbow, args=(speed,))
92          #Now start the rainbow thread
93          self.rainbowThread.start()
```

The threading module provides the solution to the concurrent processing problem. This module allows you to create threads from inside one python script. A thread runs separately from the main python script and can run at the same time as the main program doing its own computation and moving further down the code flow. A thread can be thought of as a split in a pipe where water can flow two ways at the same time. The startRainbow procedure opens a thread which runs the rainbow procedure indefinitely until the stop flag is flipped. The flag is set too false on line 88 just to be sure that it is set correctly as there is a possibility that it is currently true if a previous rainbow was in operation and then stopped. Then a thread is setup which targets the rainbow procedure this is the procedure that will be ran concurrently when the thread is started, and the speed argument is also passed into the target procedure as required by the rainbow procedure. Finally, the thread that was setup in the line prior is started. From this point on the rainbow procedure is moving a snake of 7 rainbow colored leds up and down the led strip whilst python is still able to do whatever it wants such as open the window.



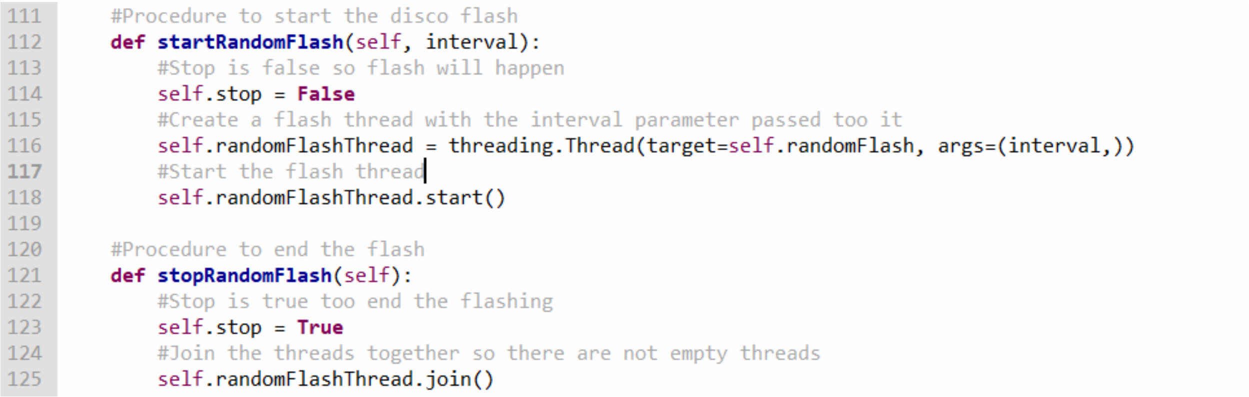
```
95      #Procedure to stop the rainbow
96      def stopRainbow(self):
97          #Stop is set to true so the rainbow snake ends
98          self.stop = True
99          #Join the threads together to ensure there are no open idle threads
100         self.rainbowThread.join()
101
```

There will eventually be a time when the rainbow snake needs to be stopped say when it is nighttime in the greenhouse. Too do this I created the stopRainbow class. This class is nice and short and sets the stop flag to equal True. On the next iteration of the rainbow snake the while loop condition won't evaluate as true, and the snake will end. There will also be an empty but open thread so join() is used to bring the thread to an end. This is like joining the two pipes back up so the water flows in on pipe again. During the write up of this code it has occurred to me that there is a situation where the snake is stopped but it's still going to be displayed on the leds in its current position as nowhere has the off procedure been called to clear the strip. I will fix this issue in the testing phase.



```
    #Procedure to change the led colour at a set interval
    def randomFlash(self, interval):
        #Continue chaning the led colour until told to stop
        while not self.stop:
            #Fill the led strip to a random rainbow colour
            self.pixels.fill(random.choice(self.rainbowColours))
            #Time delay so the interval between colour changes can be swapped
            time.sleep(interval)
```

The second of the two playful lighting modes is the randomFlash procedure. This procedure will change the light of the led strip to a random rainbow color from the rainbowColours array at a set interval. The intended effect of this is a disco room where the lights are constantly changing. The only parameter for this procedure is the interval at which the light will change given as an integer. A while loop means the lights change indefinitely until the flags changed. Each iteration the led strip is filled with a random rbg value that has been chosen from the rainbowColours array using the random library. Just as in the rainbow procedure a time delay is added to change the interval between iterations. As this procedure is going to be threaded this time delay won't slow down the function of the greenhouse only the function of this procedure.



```
    #Procedure to start the disco flash
    def startRandomFlash(self, interval):
        #Stop is false so flash will happen
        self.stop = False
        #Create a flash thread with the interval parameter passed too it
        self.randomFlashThread = threading.Thread(target=self.randomFlash, args=(interval,))
        #Start the flash thread
        self.randomFlashThread.start()

    #Procedure to end the flash
    def stopRandomFlash(self):
        #Stop is true too end the flashing
        self.stop = True
        #Join the threads together so there are not empty threads
        self.randomFlashThread.join()
```

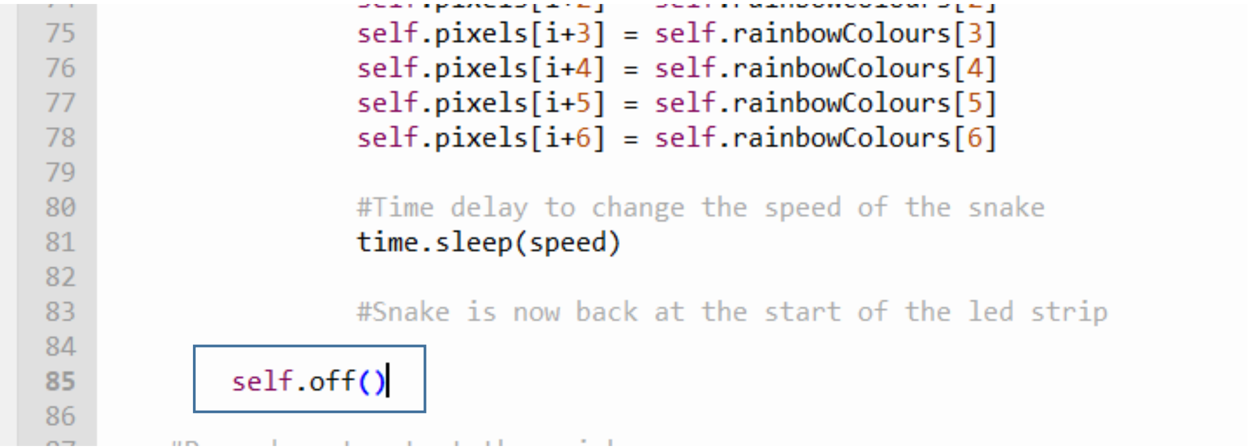
The startRandomFlash and stopRandomFlash procedures are the same as the rainbow start stop functions. They open and close a thread passing any required arguments to the randomFlash procedure.

Testing

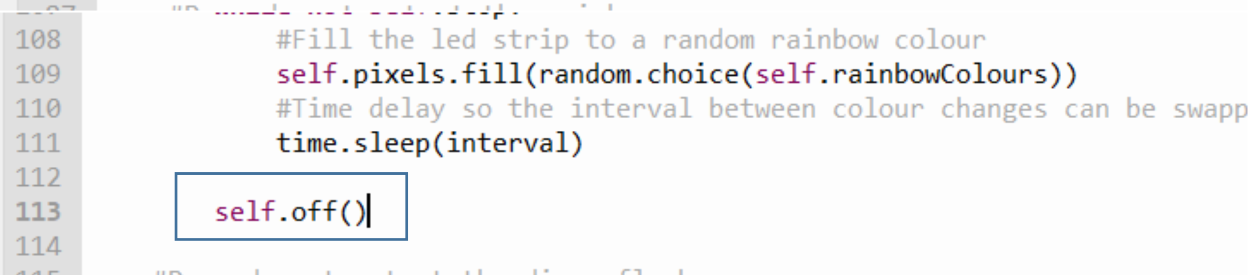
Test Number	Test Plan	Expected Outcome	Actual Outcome	Pass/Fail
1	Fill the led strip with a rgb value	The strip gets filled with that rbg value	The strip was filled with the inputted rgb value	Pass
2	Turn off the led strip	The led strip will turn off	The strip turned off	Pass
3	Start the rainbow without threading	The snake will move up and down the leds until a keyboard interrupt	The snake moved up and down the leds	Pass
4	Start the rainbow using the threading function and see if other processes	The snake will begin to move forward and then back whilst python completes some other code	The snake moved and python continued to function	Pass

	can be competed in python			
4	End the rainbow thread	Rainbow stops and the led strip goes too off	Rainbow stopped but the strip didn't go off it was left with the snake stood still	Fail
5	Start the randomFlash without threading	Leds will flash until keyboard interrupt	Leds flashes	Pass
6	Start the randomFlash using the threading function	RandomFlash should happen and python can continue to process	Flashed and python worked	Pass
7	End the randomFlash using the threading function	Flash will end and the leds go off	Flash stopped but stayed on in the color of the final flash	Fail

There were two failures in my tests which both related to the leds not going back too blank once the procedure controlling them was stopped. To solve this, I'm just going to add self.off() to the end of the rainbow and randomFlash procedures but outside of the loop. So, when the loop ends the offline is executed. Below are the changes made which have fixed the two failures.



```
                self.pixels[i+3] = self.rainbowColours[3]
                self.pixels[i+4] = self.rainbowColours[4]
                self.pixels[i+5] = self.rainbowColours[5]
                self.pixels[i+6] = self.rainbowColours[6]

                #Time delay to change the speed of the snake
                time.sleep(speed)

                #Snake is now back at the start of the led strip

        self.off()

```



```
            #Fill the led strip to a random rainbow colour
            self.pixels.fill(random.choice(self.rainbowColours))
            #Time delay so the interval between colour changes can be swapp
            time.sleep(interval)

        self.off()

```

Review

In practice mainly the led strip will be set to one color too light the plants but there will be the option to active one of the two fun modes. The rainbow and flash lighting modes were produced more as a demo function too create interest in the project rather than to help optimize plant growth. They did provide a nice challenge when programming as particularly the rainbow required some thinking, and the threading was a new library too me.

Source - https://learn.adafruit.com/neopixels-on-raspberry-pi
Source - https://www.thegeekpub.com/15990/wiring-ws2812b-addressable-leds-to-the-raspbery-pi/

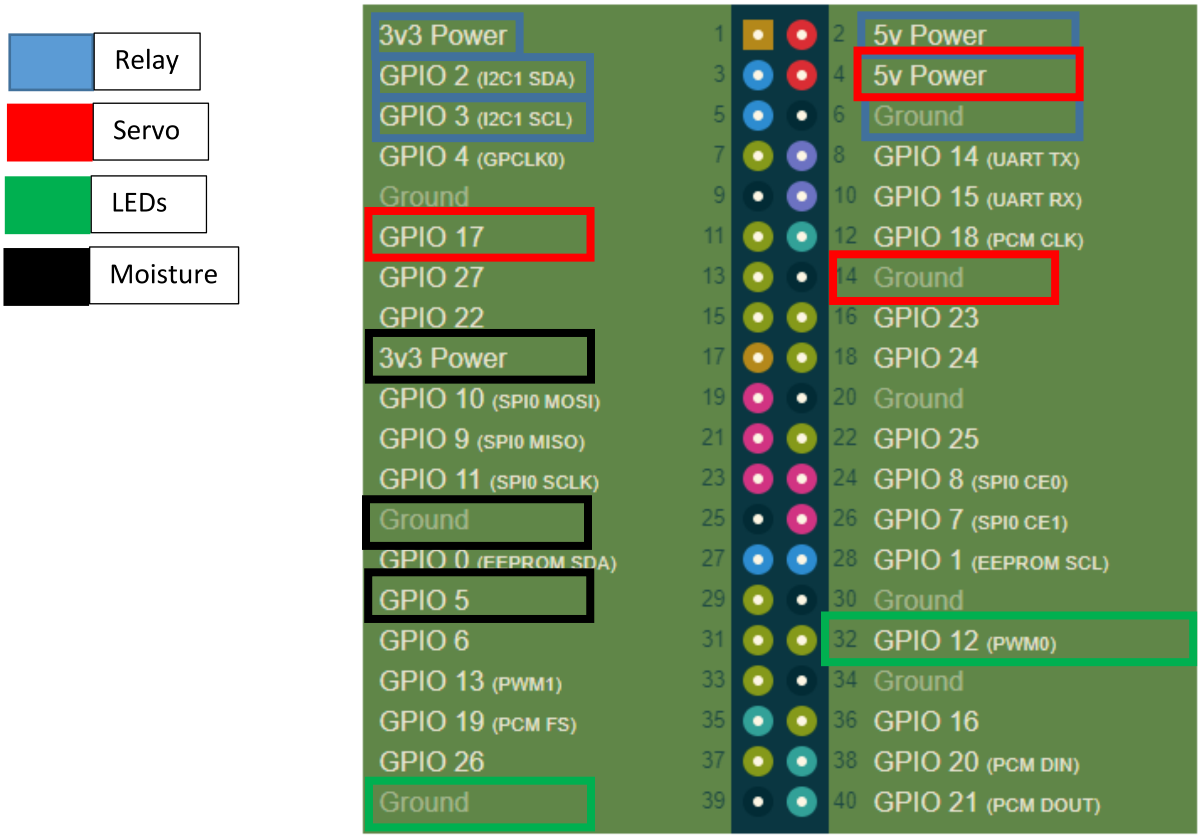
Iterative stage 4 – Moisture sensor

Requirements

This class needs to have a function that will return true if the plant needs watering and false when the plant does not need watering. The moisture sensor has a potentiometer that needs to be set manually which determines when the sensor detects moisture. This means the moisture threshold of the soil will need to be set by the user as it's not possible to do this in software.

Hardware

The moisture sensor consists of a sensor and a probe. The probe is wired to the sensor by two jumper wires. It does not matter which way round the wires go onto the sensor. Three pins are attached from the sensor too the Raspberry Pi. These are VCC which attaches onto pin 17 for 3v3 power, GND too pin 25 for ground and D0 attaches too GPIO 5 pin 29. Too adjust the threshold at which the sensor detects moisture there is a blue potentiometer on the sensor that can be rotated to change the threshold. The user will need to water some soil and then set the sensor to be off when the probe is placed inside the soil too set up the threshold. Figure 8 shows the gpio pins in use by the moisture sensor and the other components of my project so far. Whilst figure 9 shows the sensor and probe assembly.



🟦	Relay
🟥	Servo
🟩	LEDs
⬛	Moisture

3v3 Power	1	2	5v Power
GPIO 2 (I2C1 SDA)	3	4	5v Power
GPIO 3 (I2C1 SCL)	5	6	Ground
GPIO 4 (GPCLK0)	7	8	GPIO 14 (UART TX)
Ground	9	10	GPIO 15 (UART RX)
GPIO 17	11	12	GPIO 18 (PCM CLK)
GPIO 27	13	14	Ground
GPIO 22	15	16	GPIO 23
3v3 Power	17	18	GPIO 24
GPIO 10 (SPI0 MOSI)	19	20	Ground
GPIO 9 (SPI0 MISO)	21	22	GPIO 25
GPIO 11 (SPI0 SCLK)	23	24	GPIO 8 (SPI0 CE0)
Ground	25	26	GPIO 7 (SPI0 CE1)
GPIO 0 (EEPROM SDA)	27	28	GPIO 1 (EEPROM SCL)
GPIO 5	29	30	Ground
GPIO 6	31	32	GPIO 12 (PWM0)
GPIO 13 (PWM1)	33	34	Ground
GPIO 19 (PCM FS)	35	36	GPIO 16
GPIO 26	37	38	GPIO 20 (PCM DIN)
Ground	39	40	GPIO 21 (PCM DOUT)

Figure 8

- VCC – 3v3 power pin 17
- GND – GND pin 25
- D0 – GPIO 5 pin 29

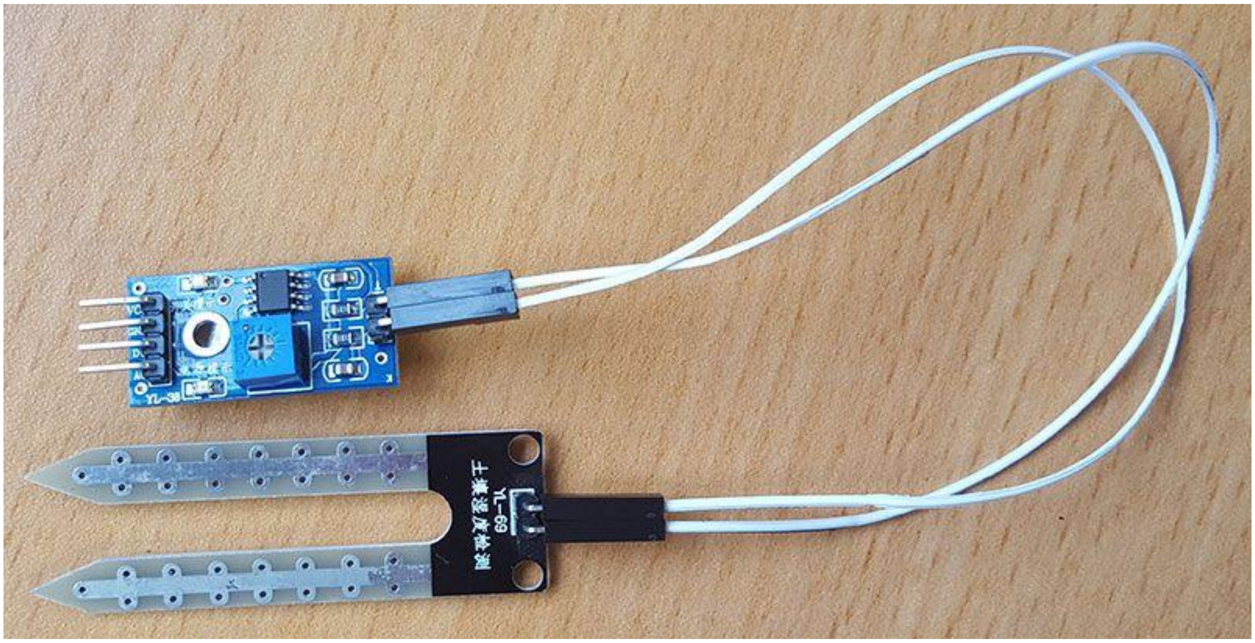


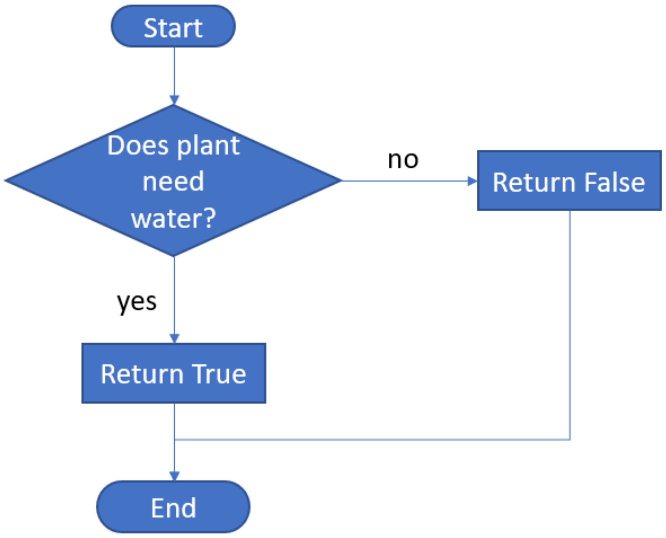
Figure 9

Class Diagram

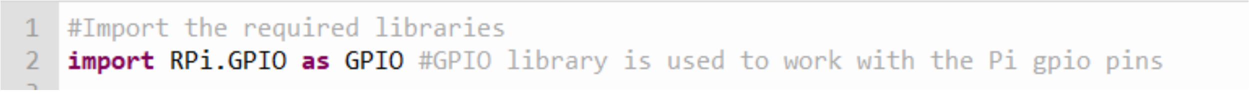
Moisture
+doesPlantNeedWater()

- DoesPlantNeedWater () will return true if the plan needs watering and will return false if the plant does not need watering.

Flow Chart

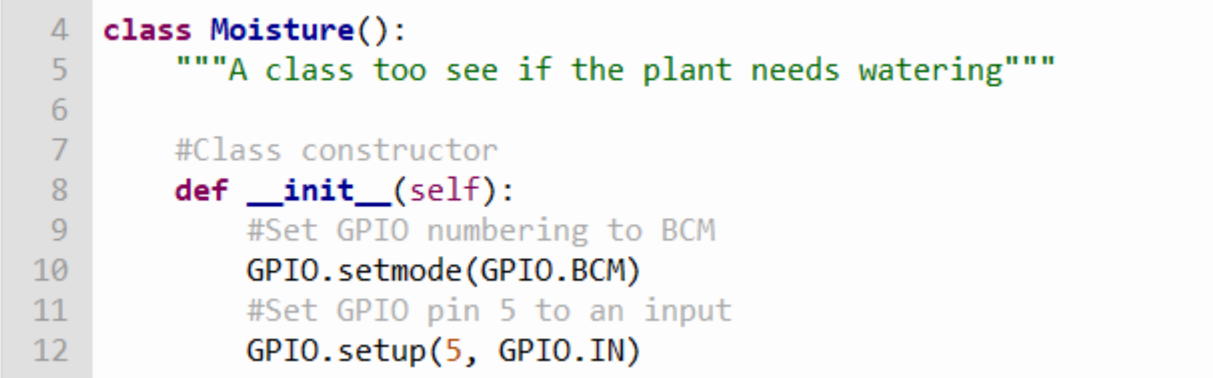


Development Log



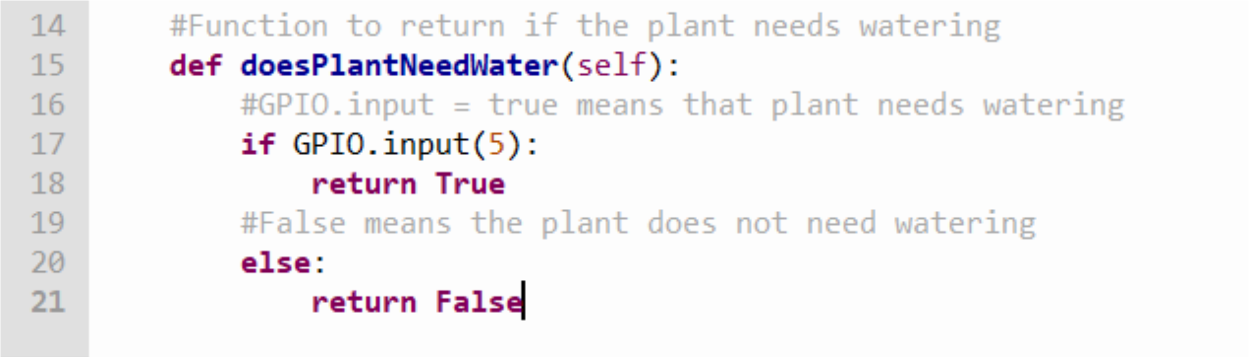
```python
#Import the required libraries
import RPi.GPIO as GPIO #GPIO library is used to work with the Pi gpio pins
```

So that python can communicate with the GPIO pins the RPi.GPIO library is used. This is imported under the identifier GPIO just to help make the code more readable.



```python
class Moisture():
    """A class too see if the plant needs watering"""

    #Class constructor
    def __init__(self):
        #Set GPIO numbering to BCM
        GPIO.setmode(GPIO.BCM)
        #Set GPIO pin 5 to an input
        GPIO.setup(5, GPIO.IN)
```

The class constructor of moisture sets up the GPIO library so that it has the correct settings too work with the signal from the moisture sensor. As this is a digital sensor when the sensor detects moisture the output on GPIO 5 is LOW 0v and then when the sensor can't detect moisture the sensor is HIGH 3.3v. The GPIO mode is set to BCM and GPIO 5 is setup as an input pin to detect a high / low signal.



```python
    #Function to return if the plant needs watering
    def doesPlantNeedWater(self):
        #GPIO.input = true means that plant needs watering
        if GPIO.input(5):
            return True
        #False means the plant does not need watering
        else:
            return False
```

The doesPlantNeedWater function will return true when the plant needs watering and false when the plant does not need watering. The plant will be deemed to need watering when the sensor does not detect moisture this will be when the moisture drops below the manually set potentiometer threshold. GPIO.input(5) will return True when the reading on GPIO 5 is a HIGH 3.3V reading. In this case we return True to indicate that the plant needs water. In the case the sensor detects moisture the reading will be a LOW 0V reading and we return False to indicate that the plant does not need watering at the time the reading was taken.

Testing

Test Number	Test Plan	Expected Outcome	Actual Outcome	Pass/Fail
1	Place the probe into a glass of water	DoesPlantNeedWater will return False	False was returned	Pass
2	Leave the probe out of water	DoesPlantNeedWater will return True	True was returned	Pass

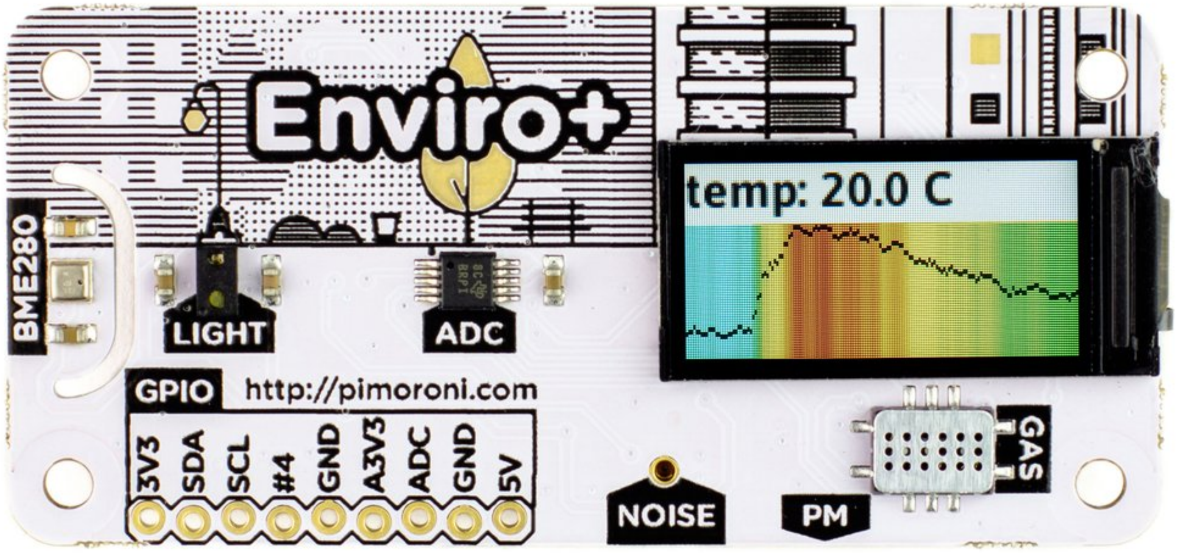
Review

The moisture class nicely abstracts the job of determining if the plant needs water or not into a simple True or False. This class will be used as part of a feedback loop later in my project too regularly check if the plant needs watering and then act accordingly. It is not ideal that the user will have to manually set the potentiometer however once set it should not need to be changed again. The way to avoid this manual setting would be to use the analog signal from the sensor but this would require a microprocessor, and this was extra complexity that I decided against.

Iterative stage 5 – Enviro Plus

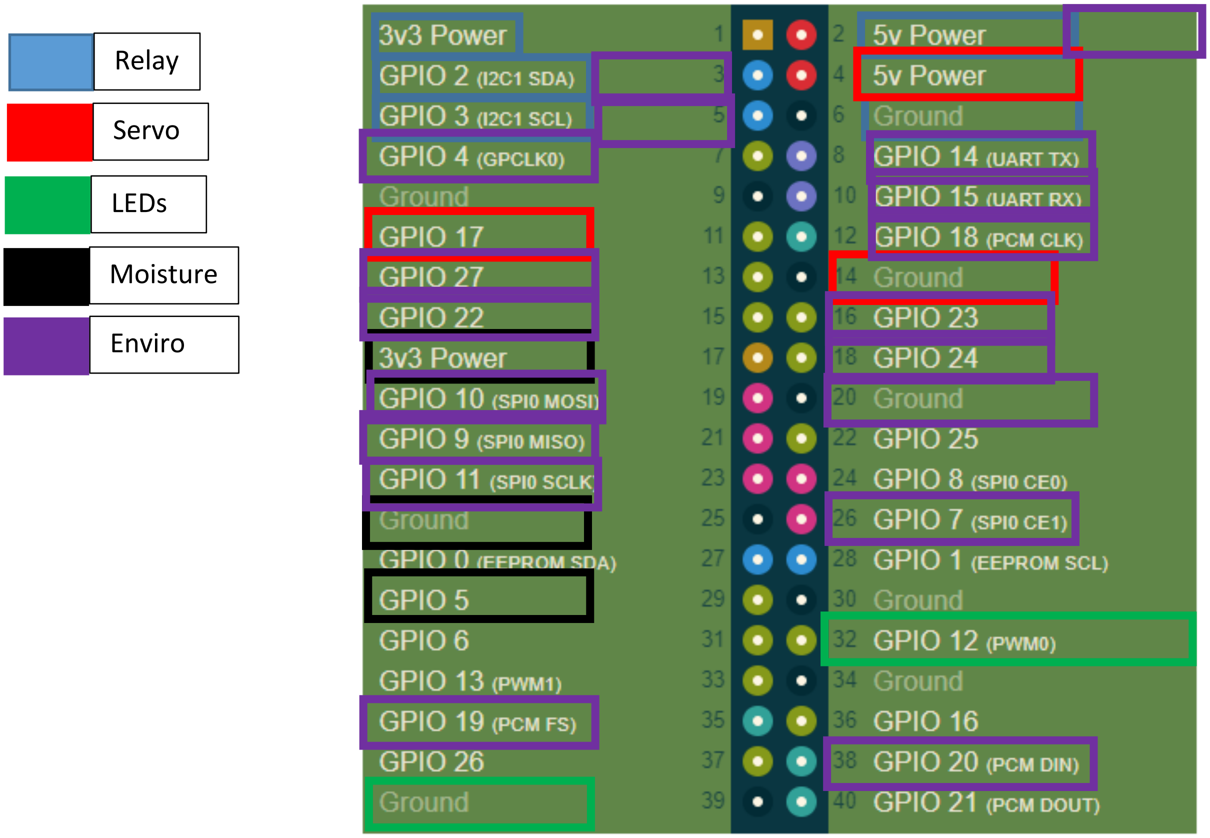
Requirements

The enviro plus board pictured below is a compact sensor board that contains a range of sensors such as temperature, light, gas, pressure and many more. A side from the moisture sensor setup in stage 4 this board will be responsible for taking all sensor readings required by the greenhouse. Not all readings will be utilized such as the gas sensor. The 4 sensor readings that the greenhouse will track will be temperature, pressure, humidity, and light.



Hardware

The board used 16 pins and is by far the largest device in terms of pin requirements. All required pins have been left free for use by the enviro apart from the backlight pin which the LED neopixles library forced me to use. This isn't too much of an issue as the screen on the enviro won't be used so having no backlight on it won't make any difference. Below is the diagram of the pins now in use for my greenhouse. Some of the pins are doubled up at this stage but this won't affect the function of any sensors as the pins are either doubled up on power pins or on the I2C pins with can deal with many parallel devices. All pins are connected to the pi from the enviro via a female to male jumper pin. This is the final diagram as all devices and sensors have been wired to the pi. The only thing left is to add power too some of the relay devices such as the motor and the fan, but this will be dealt with externally power wise from the Pi in the following iterative stages.



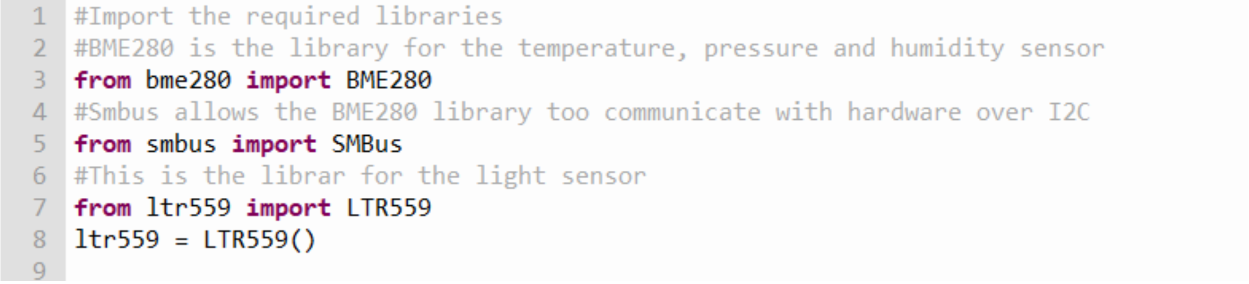
Class Diagram

Enviro

+temperature()
+pressure()
+humidity()
+light()

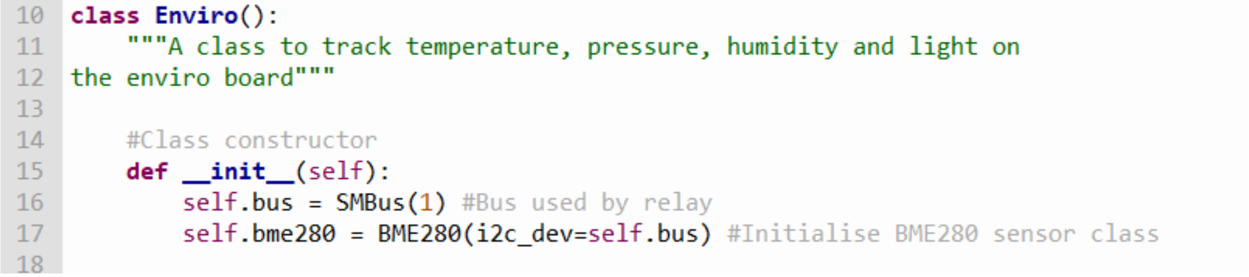
- Temperature() – return the current temperature in the greenhouse (units C)
- Pressure() – return the current pressure in the greenhouse (units hPa)
- Humidity() – return the current humidity inside the greenhouse (units %)
- Light() – return the current light level inside the greenhouse (unit lux)

Development log



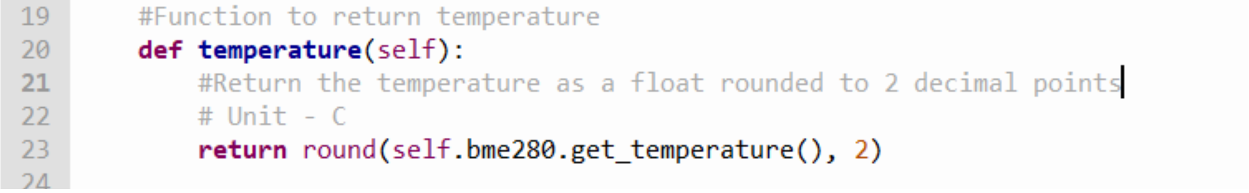
```python
#Import the required libraries
#BME280 is the library for the temperature, pressure and humidity sensor
from bme280 import BME280
#Smbus allows the BME280 library too communicate with hardware over I2C
from smbus import SMBus
#This is the librar for the light sensor
from ltr559 import LTR559
ltr559 = LTR559()

```

Here I have imported all the libraries that are required to communicate with the sensors. The bme280 library is used for communicating with the temperature, pressure, and humidity sensor. The smbus library allows the bme280 module too communicate over I2C protocol. The ltr559 library is used for the light sensor on the board.



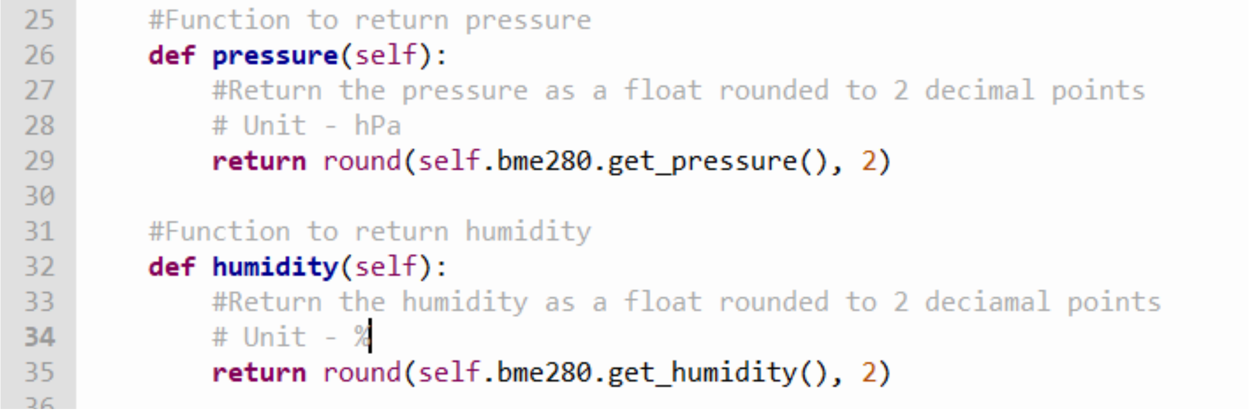
```python
class Enviro():
    """A class to track temperature, pressure, humidity and light on
the enviro board"""

    #Class constructor
    def __init__(self):
        self.bus = SMBus(1) #Bus used by relay
        self.bme280 = BME280(i2c_dev=self.bus) #Initialise BME280 sensor class

```

Inside the class constructor the bus for the I2C protocol is setup and then passed as a parameter when initializing an instance of the BME280 class.



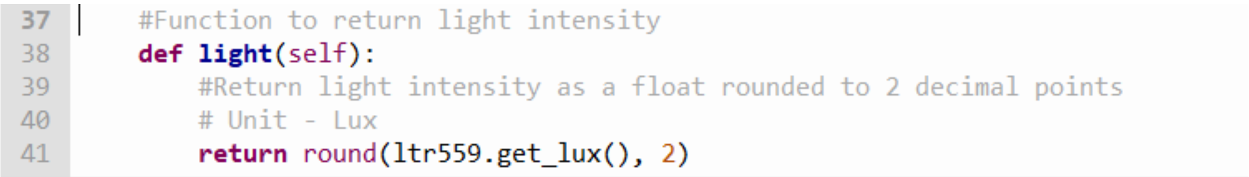
```python
    #Function to return temperature
    def temperature(self):
        #Return the temperature as a float rounded to 2 decimal points
        # Unit - C
        return round(self.bme280.get_temperature(), 2)

```

The temperature function uses bme280.get_temperature to get the current reading from the enviro board. This value by default extends to many decimal places so to sanitize the data have chosen too round this value to 2 decimal places. This rounded value in float data type is then returned by the function.



```python
    #Function to return pressure
    def pressure(self):
        #Return the pressure as a float rounded to 2 decimal points
        # Unit - hPa
        return round(self.bme280.get_pressure(), 2)

    #Function to return humidity
    def humidity(self):
        #Return the humidity as a float rounded to 2 deciamal points
        # Unit - %
        return round(self.bme280.get_humidity(), 2)

```

The pressure and humidity functions follow the same format but using the correct sensor. Once again rounding to 2 decimal places and returning the value in float format.



```
    #Function to return light intensity
    def light(self):
        #Return light intensity as a float rounded to 2 decimal points
        # Unit - Lux
        return round(ltr559.get_lux(), 2)
```

Finally, the light function makes use of the ltr559 library to obtain the luminosity from the light sensor. This value is rounded and retuned.

Testing

Test Number	Test Plan	Expected Outcome	Actual Outcome	Pass/Fail
1	Get the temperature	A sensible value for temperature will be returned in float form rounded to 2 decimal places	22.04 was returned this is in float form and is rounded and seems like a sensible value for temperature	Pass
2	Get the pressure	The pressure will be returned in the correct format and data type	658.97 was returned	Pass
3	Get the humidity	The humidity will be returned	76.71 was returned	Pass
4	Get the light intensity	A value for light intensity will be returned	10.19 was retuned	Pass
5	Turn on the heat lamp and record temperature after a few minutes	The temperature should go up	The readings started at 20 and steadily climbed for every new reading whilst the lamp was on	Pass
6	Turn on the leds and record the luminosity	The luminosity should go up	A value of 0 was returned	Fail

6	Turn on the leds and record the luminosity	The luminosity should go up	A value of 0 was returned	Fail

No error was being shown in the Python shell, but the light sensor appears to be returning a value of 0 no matter the light intensity. To begin with I shined a torch onto the enviro too see if this would change the reading. This did not work so I decided to reboot the Raspberry Pi too see if this made a difference. This also had no effect on the sensor reading it was still returning 0. At this point I decided to go back to the examples provided by the maker of the board and their code was still working and returning the light intensity. After playing around with my code and the example I noticed that the sensor seems to always return 0 for its first reading. When placed in a loop constantly returning light readings the sensor would begin to provide light intensity readings after providing its initial reading of 0. It appears the error has something to do with calling the light intensity too quickly after initializing the ltr559 module. To fix this issue I could have added a time delay into the light function to ensure the sensor was properly setup before a reading was requested from it. However, I elected not to do this as when the greenhouse is started there will always be ample time between the system starting and a light reading being taken as the user will need to login which takes longer than the 0.1 second delay, I found was needed between calling import ltr559 and doing ltr559.get_lux. This is an issue I will monitor as if it proves to be a significant issue, I will have to implement the time delay fix. The reasoning for not introducing this delay is that I did not like the idea of introducing time delays as this is never goo practice unless required. Testing of test 6 produced a pass when I ensured the sensor was initialized before taking a reading.

Review

The enviro class is a key backend component that will be used during every cycle of the Greenhouse too take sensor readings. All actions of the greenhouse will be based off these readings. At this point I have created classes to control all hardware and sensors connected to the Raspberry Pi. All that is left to do hardware wise it to wire up the fan and pump too the relay board which I will cover in the next stage.

Source - https://learn.pimoroni.com/tutorial/sandyj/getting-started-with-enviro-plus

Iterative stage 6 – Relay wiring + Component testing

Overview

This stage is purely hardware focused and won't involve any programming unless bugs are identified. At this stage the following components are connected to the Raspberry Pi, and I have written code to control them the relay, the LED strip, the servo, the moisture sensor, and the enviro sensor board. This leaves the Fan, the heat lamp and the pump that needs to be connected to the relay. These devices will all need to be wired too an external power source and go via separate relays to allow me to control the function of them. After this I will carry out an extensive test plan to check all hardware components are working and specifically, they are working whilst other components are also in operation.

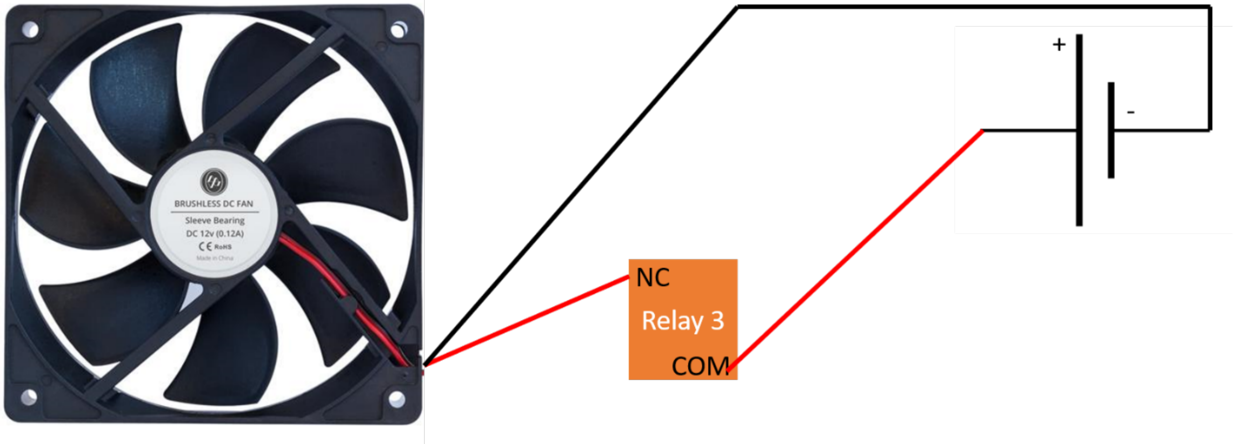
Relay

Since I began development of this project, I have bought a heating lamp that is designed for heating chickens. This light is very powerful and can provide much more heat than the heating pads. For this reason, I have decided to use the heating lamp instead of the heating pads as the primary source of heat for the greenhouse. This means relay 2 will be left empty and relay 4 used for the heating lamp. The main reason for this change is that the heating elements drew so much power from my external 5v power supply that the other devices struggled to operate. The heating lamp comes with its own plug so is not on the same circuit as all other components and as such avoids this issue.

Relay 1	Relay 2	Relay 3	Relay 4
Motor		Fan	Heating Lamp

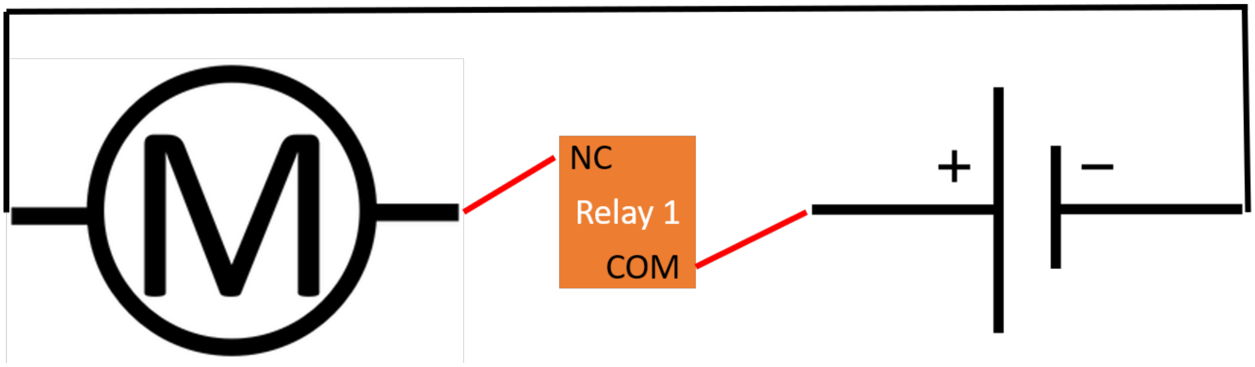
Fan

The fan has a red positive wire and a black negative wire. When power is placed over the fan it begins to spin. To turn the fan on and off it will need to be wired across a relay. The fan will be connected too my 5v external power source via the main breadboard and then wired into the common middle port of relay 3 and then wired out from the NC port on the left which means the fan will turn on when the relay is closed. Below is the wiring diagram for the fan.



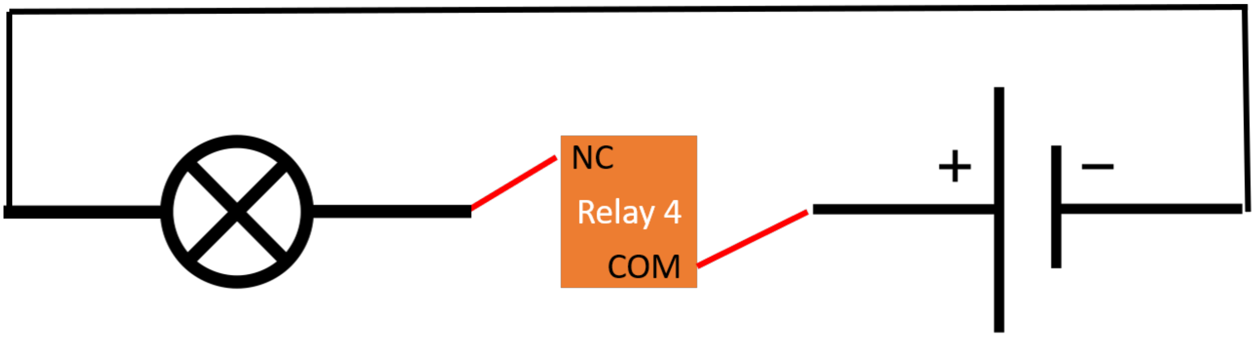
Motor / Water Pump

The pump is also powered by a standard live and neutral wire setup. The pump is connected in the same way as the fan as shown blow. The motor can run either direction so swapping the wires simply reverses the direction of the pump.



Heat Lamp

The heat lamp is connected to the relay in the same fashion as the other two devices. The lamp is powered by its own wall plug as it requires more power than the other devices.



Relay Review

All devices are now connected to the Pi and the greenhouse. Throughout the previous iterative stages I have developed classes to communicate and control all these devices. A testing plan has been developed and carried out for all these classes individually. I will now carry out a larger testing plan to ensure that all devices work simultaneously.

Hardware testing plan

In this testing plan I will produce a python script to run various components simultaneously and check that they work as expected when used in conjunction with other hardware devices and sensors.

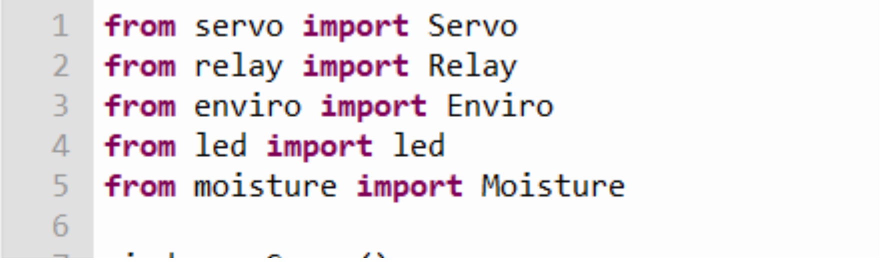
Before commencing testing, plan Raspberry Pi will be rebooted, and all devices connected and powered as the Greenhouse will be during use. During each test all other devices should continue to operate.

Test Number	**Test Plan**	**Expected Outcome**	**Actual Outcome**	**Pass/Fail**
1	Open the window	The window should open without the requirement for “sudo pigpiod” to be ran in terminal		
2	Turn on the heating lamp	The heating lamp should come on		
3	Take a reading of temperature	A float value for temperature should be returned rounded to 2 decimal places		
4	Turn on the fan	The fan will begin to spin		
5	Turn on the pump	The pump will begin to pump water into the soil		
6	At this stage all relay devices are turned on. Check all devices are functioning and not struggling to for power as they share the same power source.	The fan, pump and lamp should be performing their respective jobs to a suitable standard		
7	Fill the LED strip using the following rgb values (255,255,255). The leds share the same power source as the	The LED strip will be filled white. All devices on the 5v supply should be working.		

	relay devices. White is the most power intensive colour for the led strip as each value is at its max. Check that all devices on the 5v power supply is functioning			
8	Turn off all the relay devices. (fan, lamp, pump)	All relay devices should be turned off		
9	Begin the LED Rainbow mode	The LED should start to snake up and down the LEDs		
10	Close the window	The window will shut, and the LED will continue to snake		
11	Take a reading for pressure	Pressure will be returned as a float rounded to 2 decimal places		
12	Take a reading for humidity	Humidity will be retuned as a float rounded to 2 decimal places		
12	Take a reading of light	Light will be returned as a float rounded to 2 decimal places		
13	Check if the plan needs watering	True or false will be returned based on if the soil is too dry. Check this value against the light on the moisture sensor		
14	Turn off the LED Rainbow	The rainbow will stop, and the led strip be off		
15	Start the random flash led mode	Leds will begin to flash		

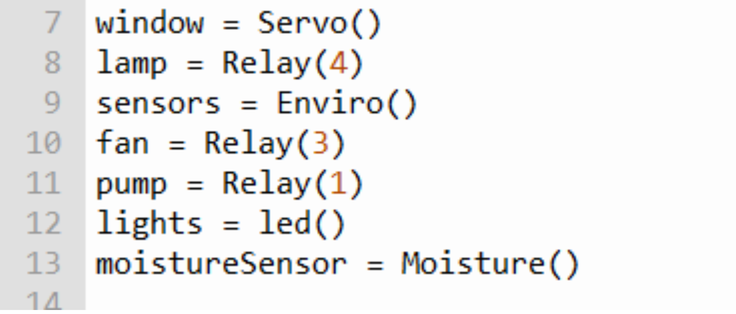
16	Turn off the random flash led mode	Flashing will stop and the led strip turn off		

Test Plan Script



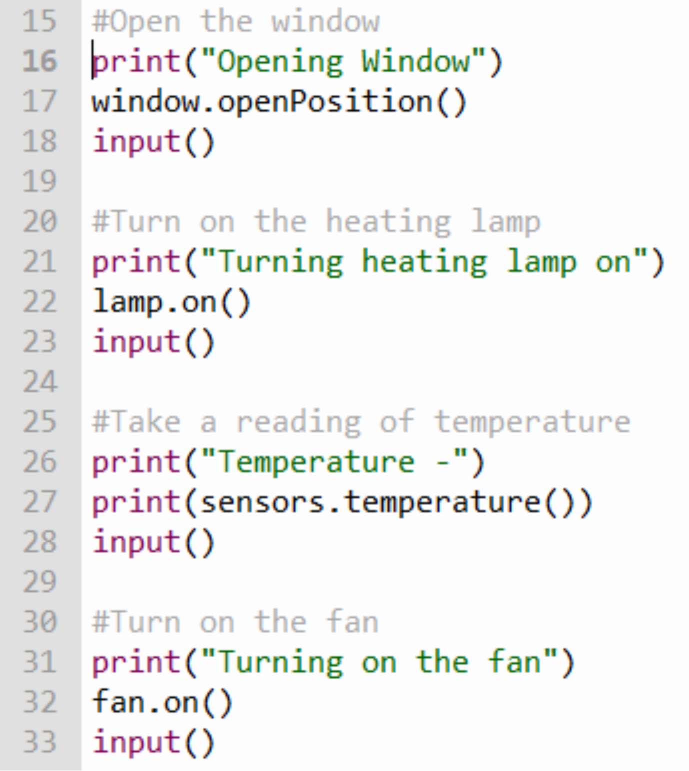
```python
from servo import Servo
from relay import Relay
from enviro import Enviro
from led import led
from moisture import Moisture

```

Import all the classes I have developed.

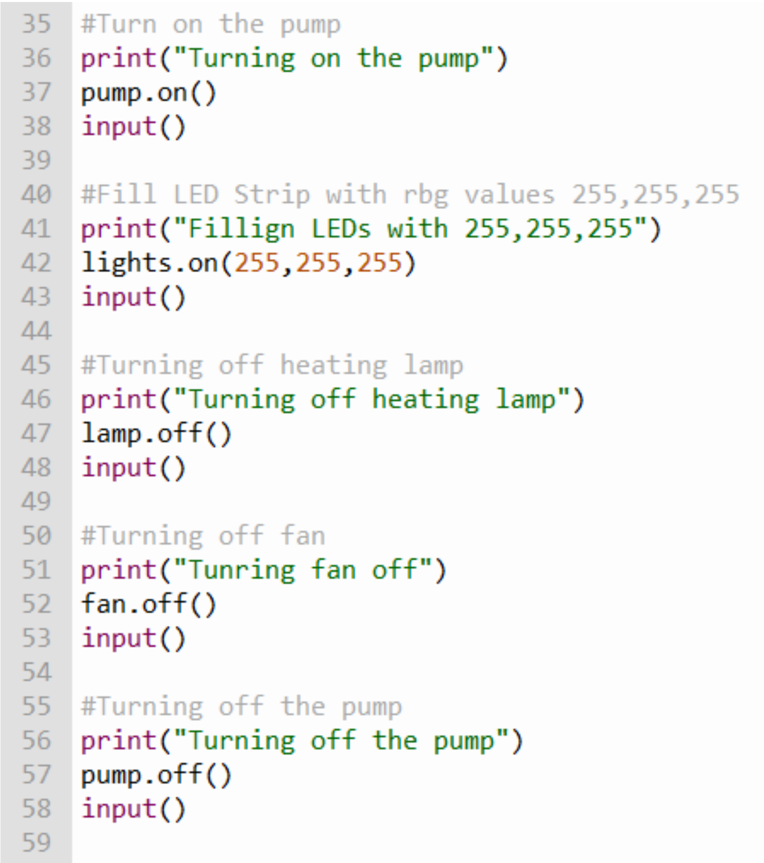


```python
window = Servo()
lamp = Relay(4)
sensors = Enviro()
fan = Relay(3)
pump = Relay(1)
lights = led()
moistureSensor = Moisture()

```

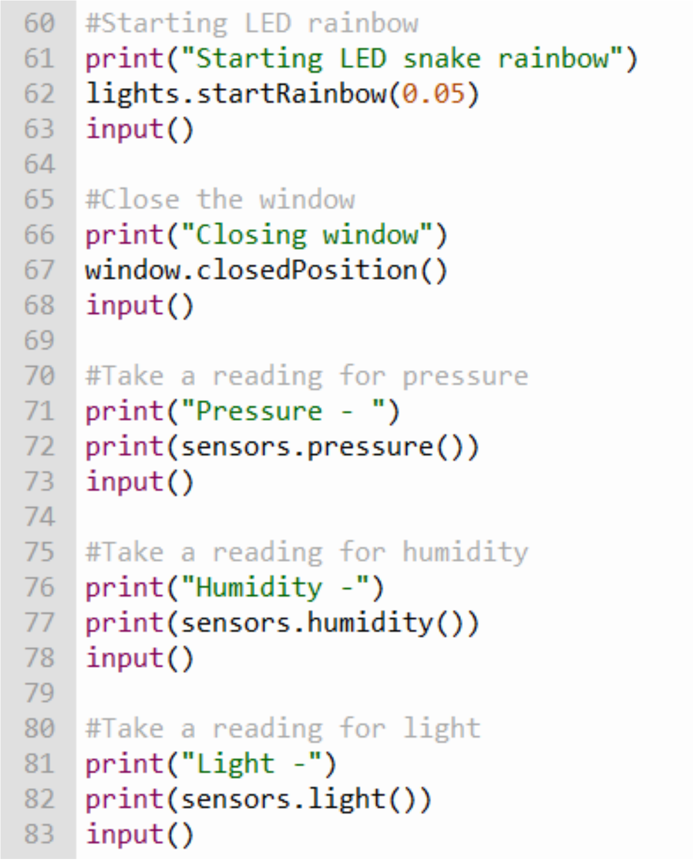
Initialize instances of all the classes. I have slightly changed how the relay class works to make it easier to use I will explain these changes later in this stage.



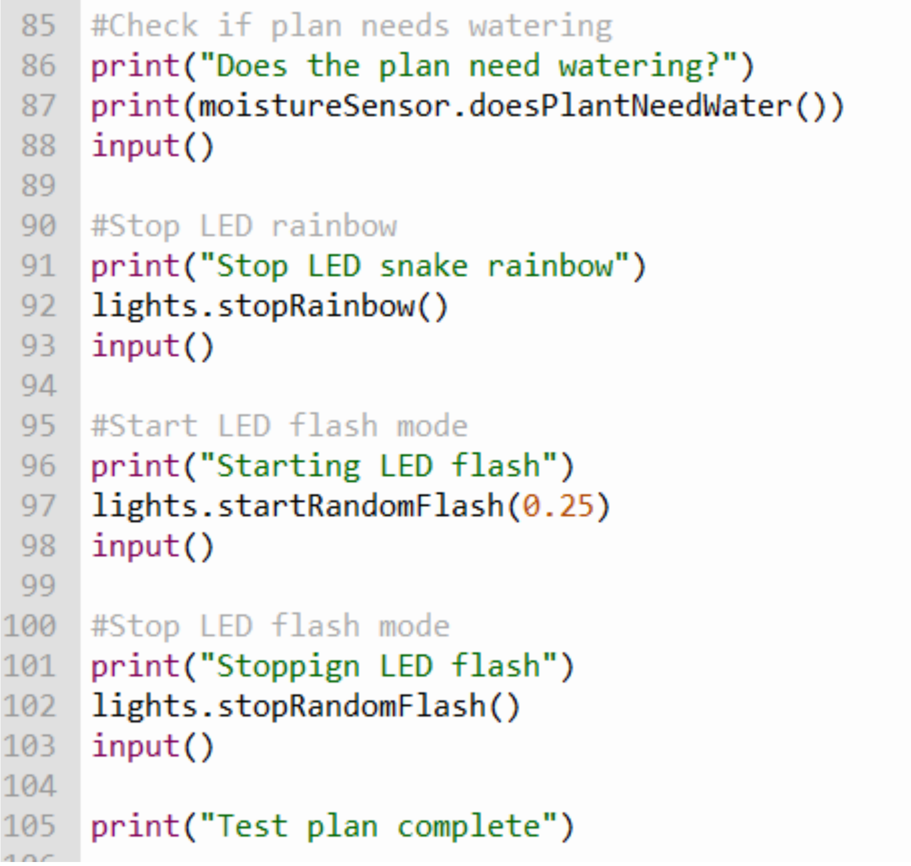
```python
#Open the window
print("Opening Window")
window.openPosition()
input()

#Turn on the heating lamp
print("Turning heating lamp on")
lamp.on()
input()

#Take a reading of temperature
print("Temperature -")
print(sensors.temperature())
input()

#Turn on the fan
print("Turning on the fan")
fan.on()
input()
```



```
#Turn on the pump
print("Turning on the pump")
pump.on()
input()

#Fill LED Strip with rbg values 255,255,255
print("Fillign LEDs with 255,255,255")
lights.on(255,255,255)
input()

#Turning off heating lamp
print("Turning off heating lamp")
lamp.off()
input()

#Turning off fan
print("Tunring fan off")
fan.off()
input()

#Turning off the pump
print("Turning off the pump")
pump.off()
input()

```



```
#Starting LED rainbow
print("Starting LED snake rainbow")
lights.startRainbow(0.05)
input()

#Close the window
print("Closing window")
window.closedPosition()
input()

#Take a reading for pressure
print("Pressure - ")
print(sensors.pressure())
input()

#Take a reading for humidity
print("Humidity -")
print(sensors.humidity())
input()

#Take a reading for light
print("Light -")
print(sensors.light())
input()
```

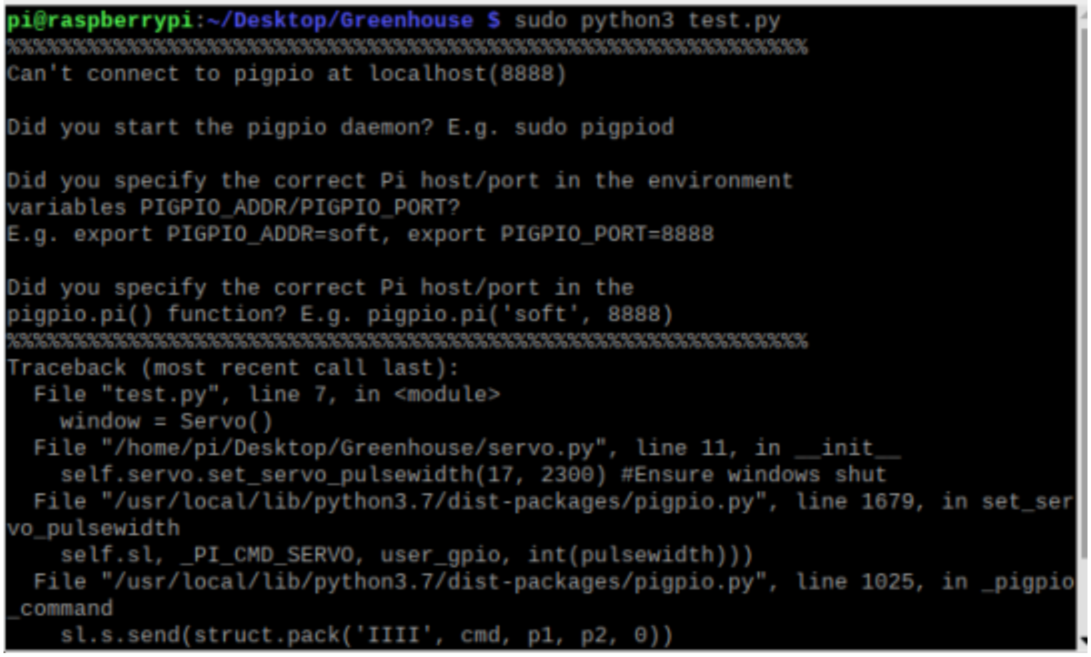


```
#Check if plan needs watering
print("Does the plan need watering?")
print(moistureSensor.doesPlantNeedWater())
input()

#Stop LED rainbow
print("Stop LED snake rainbow")
lights.stopRainbow()
input()

#Start LED flash mode
print("Starting LED flash")
lights.startRandomFlash(0.25)
input()

#Stop LED flash mode
print("Stoppign LED flash")
lights.stopRandomFlash()
input()

print("Test plan complete")
```

Throughout the test script I have used the line “input()” so that the code will wait for me to hit a key on the keyboard before moving onto the next test. This allows me as much time as I need to observe the greenhouse and check everything is working.

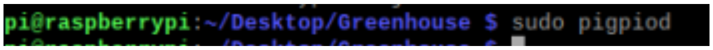
Test plan results

1	Open the window	The window should open without the requirement for “sudo pigpiod” to be ran in terminal	An error was produced say that the deamon was not started	Fail



```
pi@raspberrypi:~/Desktop/Greenhouse $ sudo python3 test.py
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Can't connect to pigpio at localhost(8888)

Did you start the pigpio daemon? E.g. sudo pigpiod

Did you specify the correct Pi host/port in the environment
variables PIGPIO_ADDR/PIGPIO_PORT?
E.g. export PIGPIO_ADDR=soft, export PIGPIO_PORT=8888

Did you specify the correct Pi host/port in the
pigpio.pi() function? E.g. pigpio.pi('soft', 8888)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    window = Servo()
  File "/home/pi/Desktop/Greenhouse/servo.py", line 11, in __init__
    self.servo.set_servo_pulsewidth(17, 2300) #Ensure windows shut
  File "/usr/local/lib/python3.7/dist-packages/pigpio.py", line 1679, in set_ser
vo_pulsewidth
    self.sl, _PI_CMD_SERVO, user_gpio, int(pulsewidth)))
  File "/usr/local/lib/python3.7/dist-packages/pigpio.py", line 1025, in _pigpio
_command
    sl.s.send(struct.pack('IIII', cmd, p1, p2, 0))
```

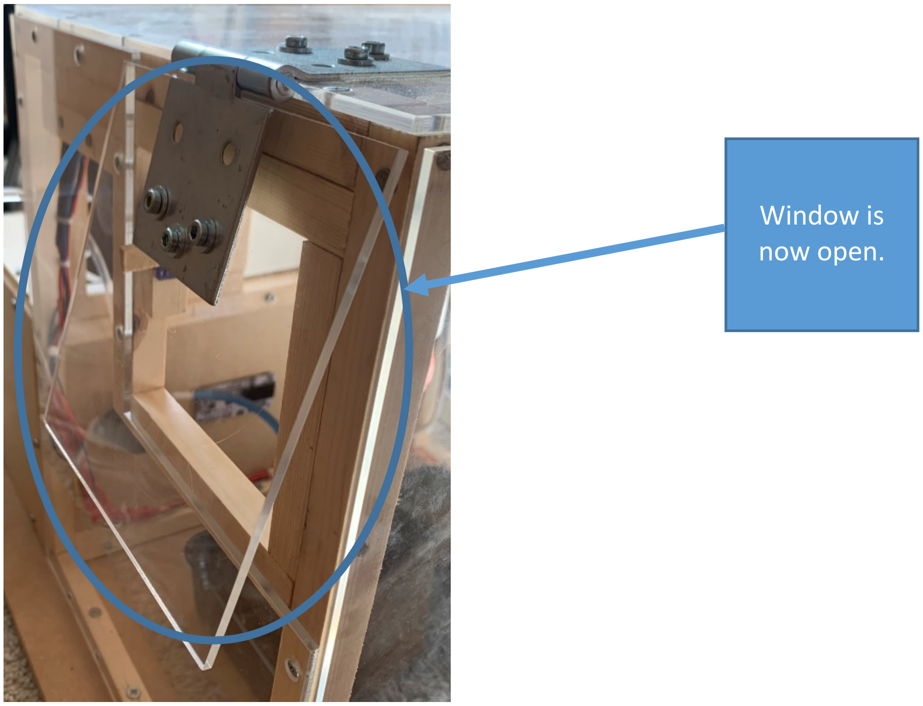
After testing the Servo class, I believed that I had fixed this issue however it appears to have reemerged as an issue. I have already carried out all the instructions in the Pigpiod help documents to run the deamon on startup, but this doesn’t appear to fix the issue. I wanted to avoid having to ask the user to run the following command before starting the greenhouse, but it looks like that’s the only reliable solution to get the servo working. Going forward before running the greenhouse the user will have to enter the following into terminal.



```
pi@raspberrypi:~/Desktop/Greenhouse $ sudo pigpiod
```

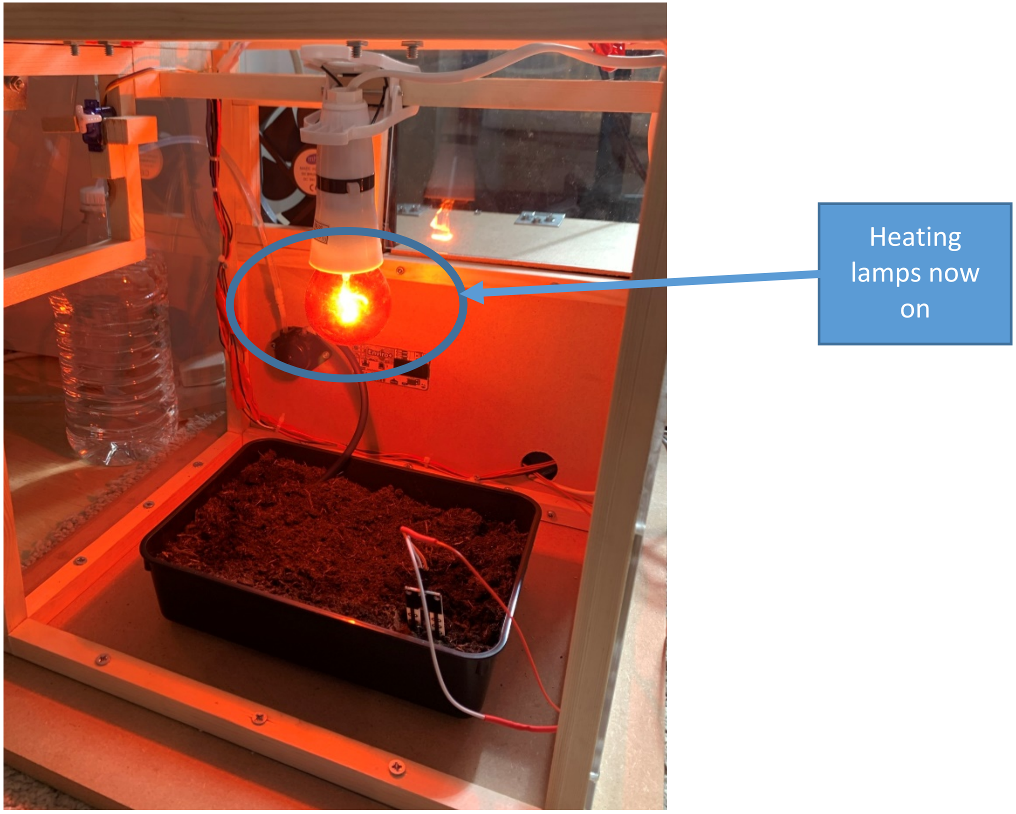
I will add a prompt in the gui after login to ask the user if they have remembered to run this command. Once this line is running the window opens as expected so the test has been passed.

1	Open the window	The window should open without the requirement for “sudo pigpiod” to be ran in terminal	After running the right terminal command, the window opened	Pass



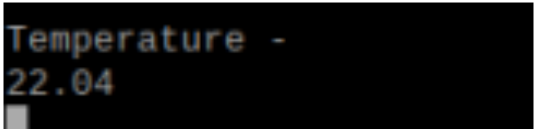
Window is now open.

2	Turn on the heating lamp	The heating lamp should come on	The lamp turned on	Pass



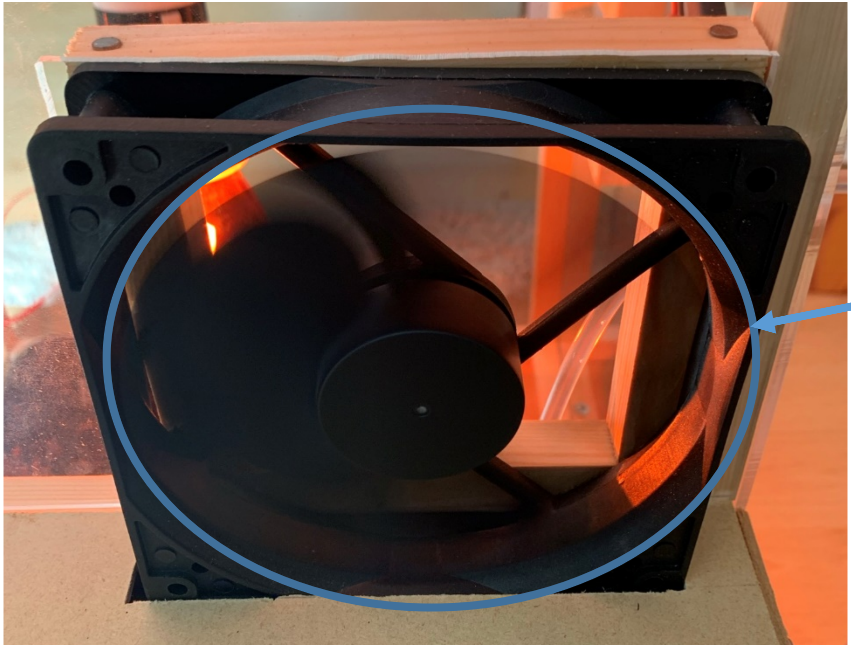
Heating lamps now on

3	Take a reading of temperature	A float value for temperature should be returned rounded to 2 decimal places	22.04 was returned as the temperature reading. This is a sensible reading considering the greenhouse is kept inside at room temperature. The format was float form and rounded to the correct number of decimal points	Pass



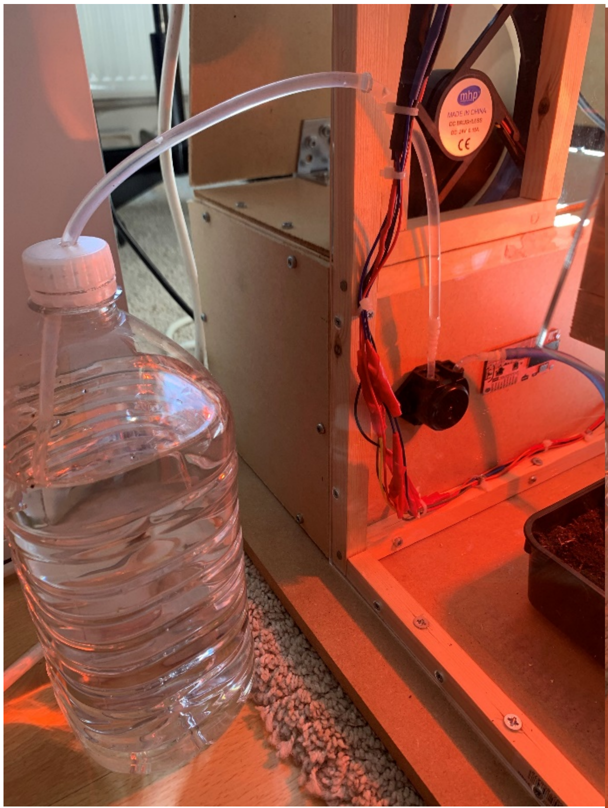
```
Temperature -
22.04
```

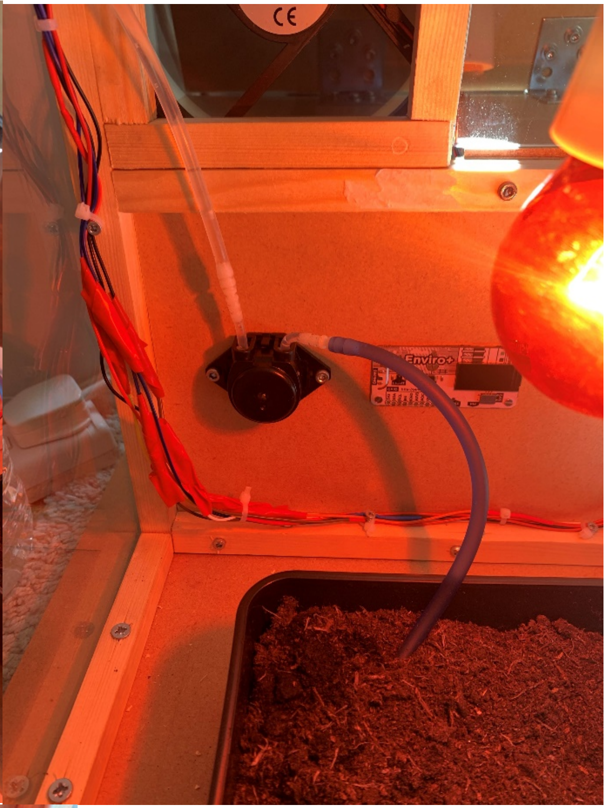
4	Turn on the fan	The fan will begin to spin	The fan started to spin	Pass

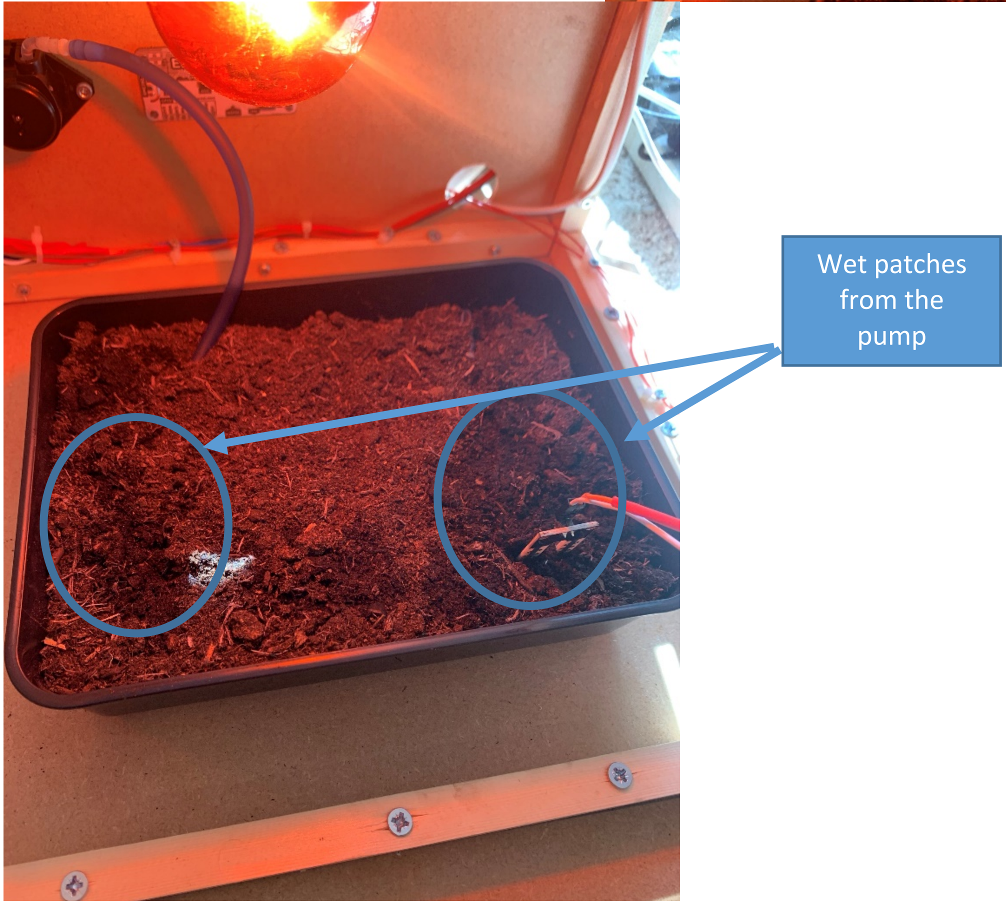


Fan is now on

5	Turn on the pump	The pump will begin to pump water into the soil	The pump turned on and started pumping water into the greenhouse	Pass

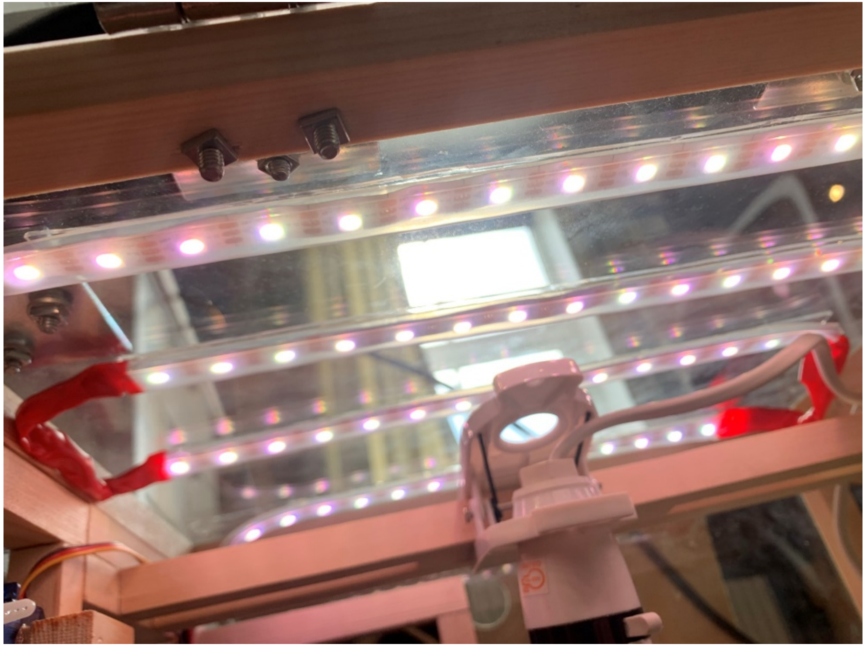




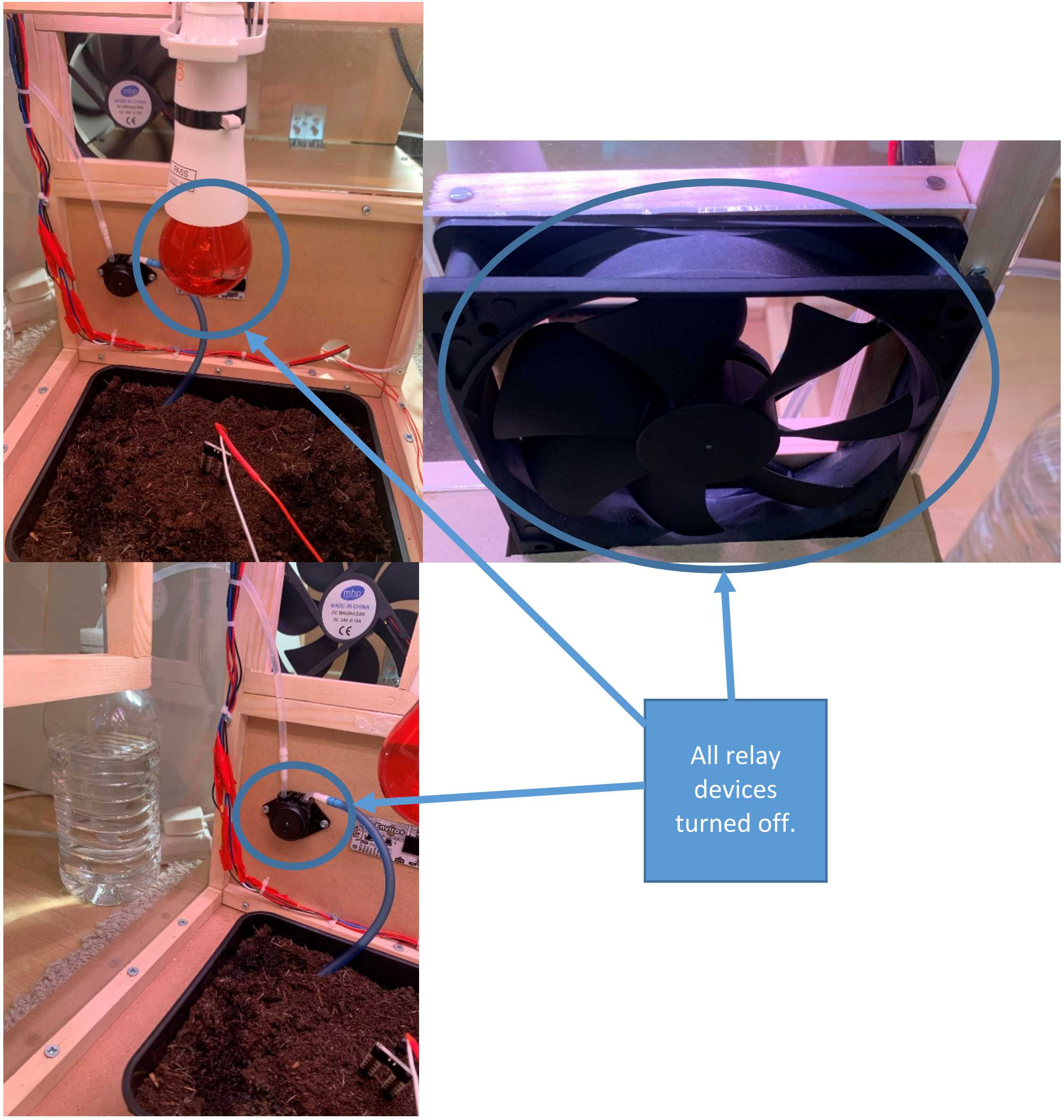

Wet patches from the pump

6	At this stage all relay devices are turned on. Check all devices are functioning and not struggling to for power as they share the same power source.	The fan, pump and lamp should be performing their respective jobs to a suitable standard	The pump, lamp and fan are working at full power and not struggling to perform their jobs	Pass

7	Fill the LED strip using the following rgb values (255,255,255). The leds share the same power source as the relay devices. White is the most power intensive colour for the led strip as each value is at its max. Check that all devices on the 5v power supply is functioning	The LED strip will be filled white. All devices on the 5v supply should be working.	The led strip was filled with the rgb values 255,255,255	Pass

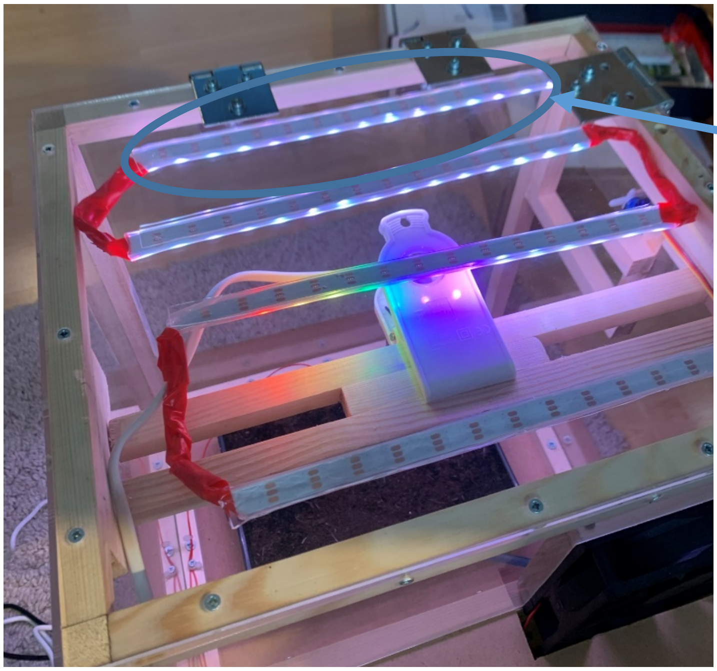


8	Turn off all the relay devices. (fan, lamp, pump)	All relay devices should be turned off	All the relay devices (lamp, pump, and fan) turned off	Pass



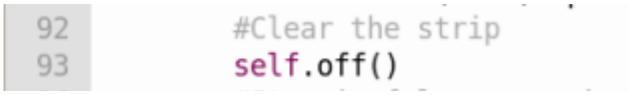
All relay devices turned off.

9	Begin the LED Rainbow mode	The LED should start to snake up and down the LEDs	The LED snake started but it did not clear the led strip before starting	Fail



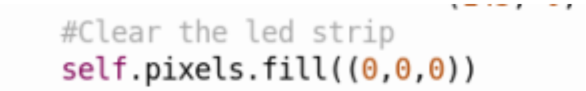
The LEDs that the snake has not yet reached are still white.

This is an issue only relevant on the first pass of the rainbow snake when the strip has previously been filled. The fix for this is to clear the strip before beginning the rainbow thread. I have added the following code to the start of the startRainbow and startRandomFlash methods this will ensure the strip is off before starting the code.



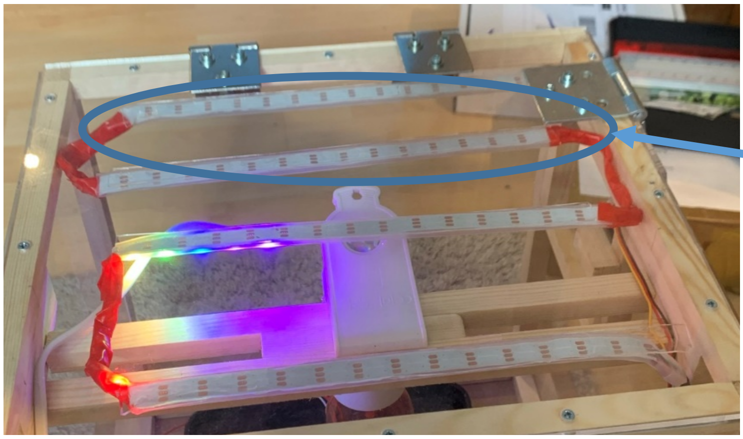
```
        #Clear the strip
        self.off()
```

I have also added the following code to the class constructor of the led class as I have noticed that on bootup of the raspberry pi the first led is sometimes turned on. This will ensure the strip is clear when the greenhouse is started. I would have used the same code as above, but the off method has not been declared when the class constructor is running so i needed to use fill instead.



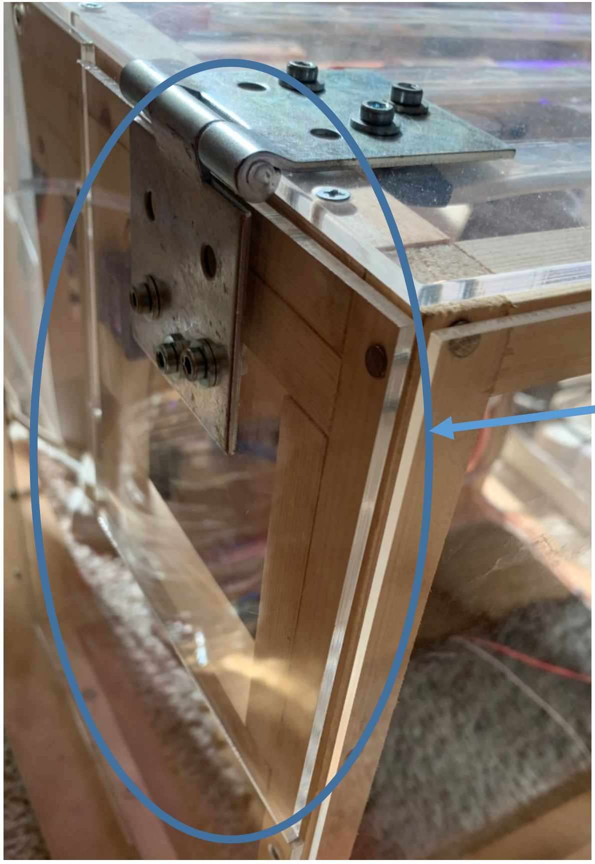
```
    #Clear the led strip
    self.pixels.fill((0,0,0))
```

9	Begin the LED Rainbow mode	The LED should start to snake up and down the LEDs	After implementing the above fixes, the led strip is now clear when the rainbow snake begins	Pass



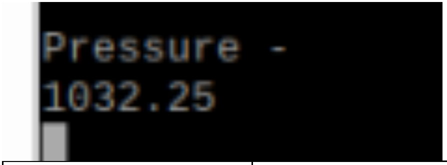
The strip is no longer filled when the rainbow snake starts.

10	Close the window	The window will shut, and the LED will continue to snake	The window shut and the rainbow snake continued to function meaning the threading is working as expected	Pass



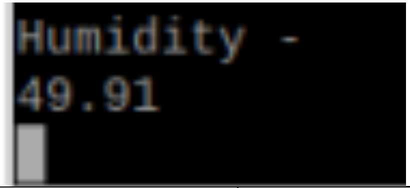
Window is now closed.

11	Take a reading for pressure	Pressure will be returned as a float rounded to 2 decimal places	1032.25 was returned this is roughly pressure at sea level so seems sensible and is correctly rounded	Pass



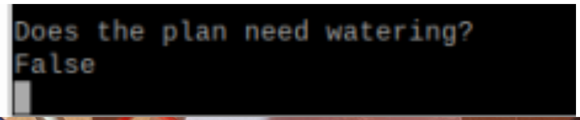
```
Pressure -
1032.25
```

12	Take a reading of light	Light will be returned as a float rounded to 2 decimal places	A humidity reading of 49.91 was returned in correct format	Pass

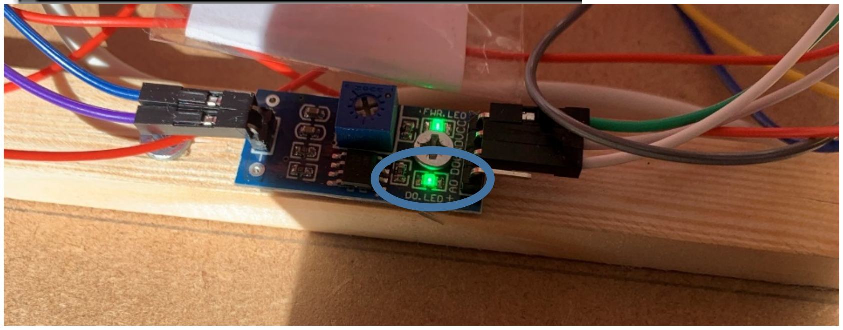


```
Humidity -
49.91
```

13	Check if the plan needs watering	True or false will be returned based on if the soil is too dry. Check this value against the light on the moisture sensor	False was returned indicating that the plan does not need watering. I am confident the pump system is working well as I could see on the sensor before the pump was turned on that it would need water but after pumping during testing that has changed too false	Pass

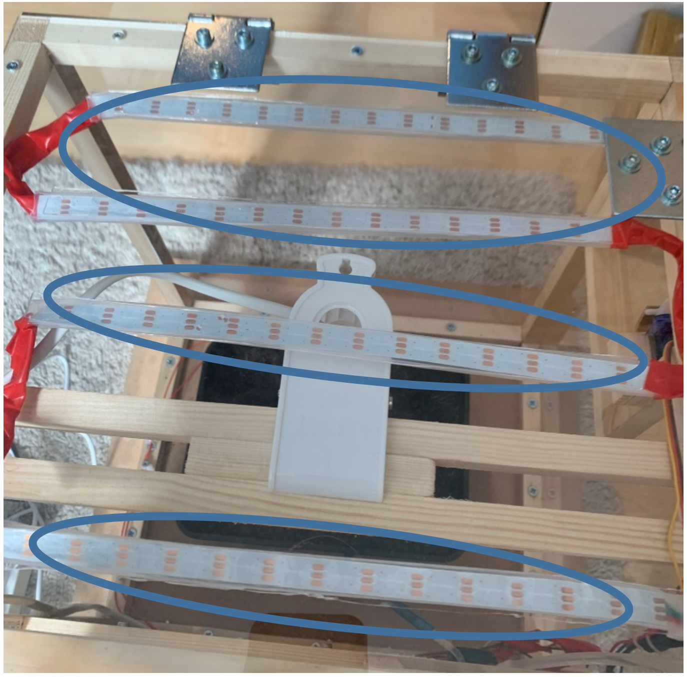


```
Does the plan need watering?
False
```



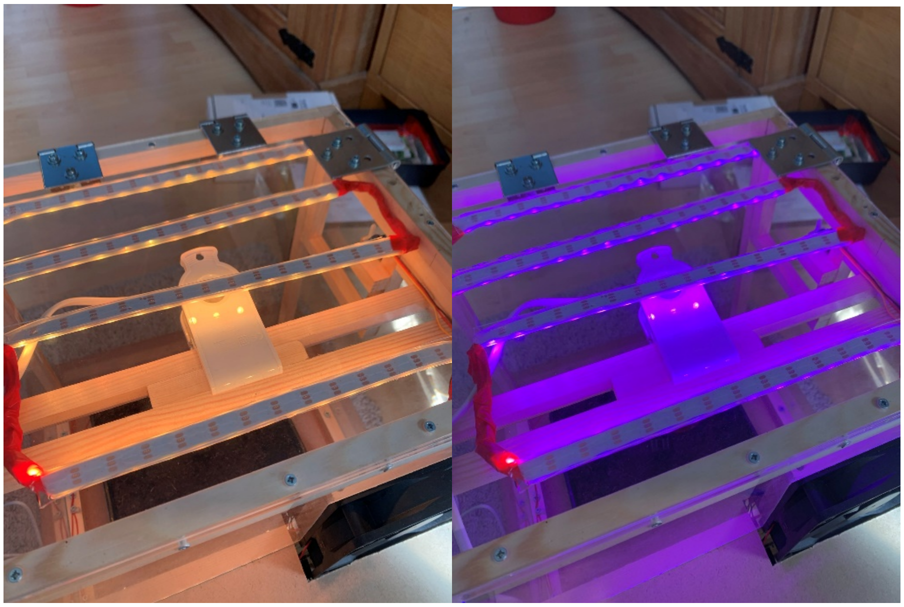
The lower green light on the sensor indicated if the sensor is detecting moisture or not. The light being on means the probe is detecting moisture and that the plan does not need watering. I have mounted the sensor at the lip of the electronics box door so that it is easily accessible if anyone wants to adjust its potentiometer (moisture detection threshold).

14	Turn off the LED Rainbow	The rainbow will stop, and the led strip be off	The LED rainbow snake ended, and the strip was fully off	Pass



The LED strip is fully off.

15	Start the random flash led mode	Leds will begin to flash	The LEDs started to flash	Pass



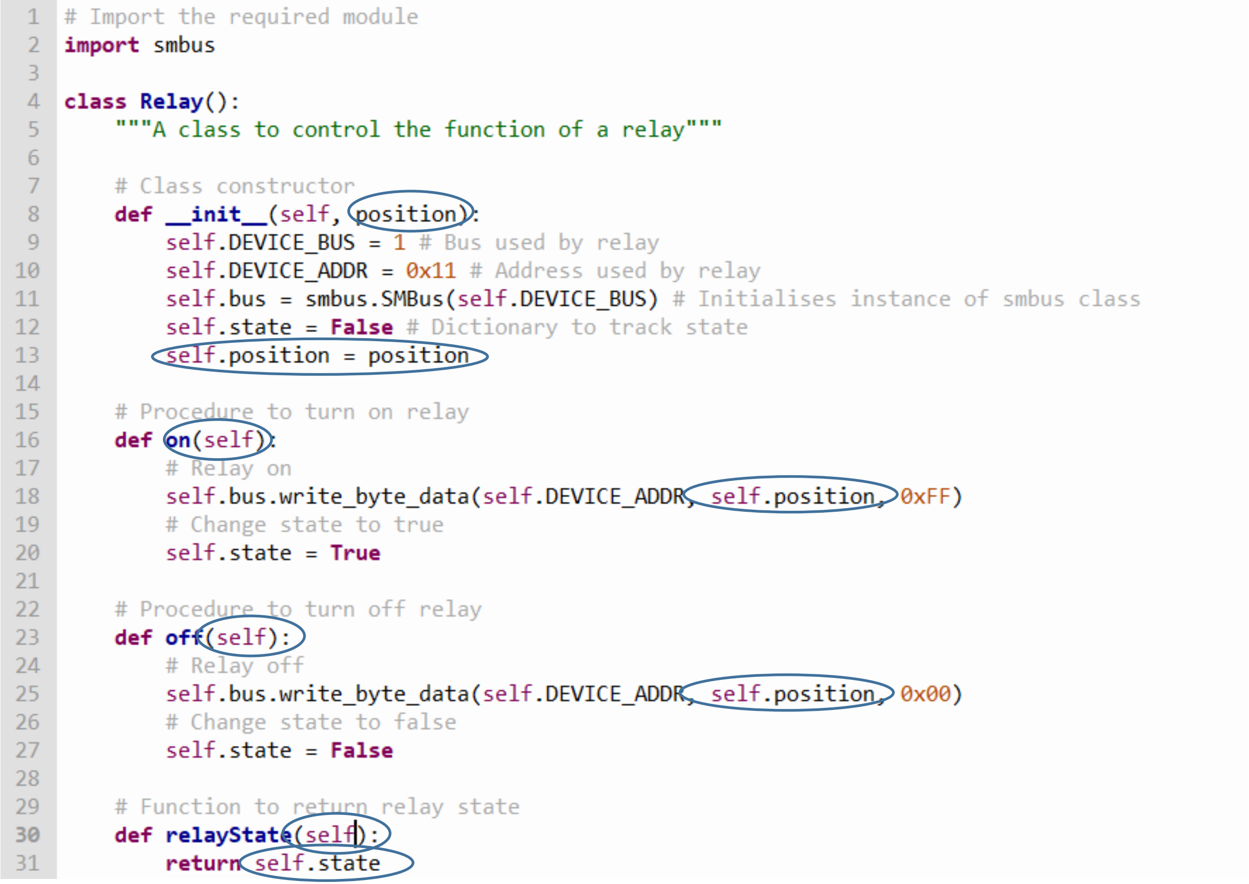
Here you can see the Led strip flashing different colors of the rainbow

16	Turn off the random flash led mode	Flashing will stop and the led strip turn off	The LED strip stopped flashing and turned off	Pass

Amendments to the Relay class

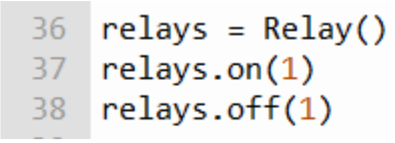
When writing the test script, I realized it would be easier to have a relay class where you pass a relay position on initialization and then that was the relay that that instance would control. This would save me passing the relay position each time I turned on or off a relay and would allow me to have three instances of the relay class one for each of the devices connected to a relay.

Too implement this change I removed the position parameters from the on, off and relay state procedures. I then added a position parameter too the class constructor that would be the relay controlled by that instance of the class. I also swapped the state identifier from an array data type to a boon to reflect the fact we are only dealing with a single relay, so it is either on or off. As we now only must enter the relay position once I have removed the inequalities that check the position is valid as its much less likely to make a mistake when we only need to enter the position when initializing the class. Finally, I changed the arguments to the write_byte_data procedure to reflect the new location of the position variable and changed the relayState function so that it returned the state bool not the state array. I have circled the changed made below.

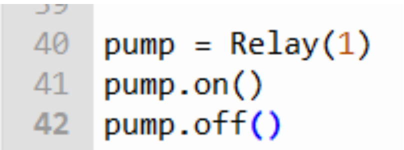


```
# Import the required module
import smbus

class Relay():
    """A class to control the function of a relay"""

    # Class constructor
    def __init__(self, position):
        self.DEVICE_BUS = 1 # Bus used by relay
        self.DEVICE_ADDR = 0x11 # Address used by relay
        self.bus = smbus.SMBus(self.DEVICE_BUS) # Initialises instance of smbus class
        self.state = False # Dictionary to track state
        self.position = position

    # Procedure to turn on relay
    def on(self):
        # Relay on
        self.bus.write_byte_data(self.DEVICE_ADDR, self.position, 0xFF)
        # Change state to true
        self.state = True

    # Procedure to turn off relay
    def off(self):
        # Relay off
        self.bus.write_byte_data(self.DEVICE_ADDR, self.position, 0x00)
        # Change state to false
        self.state = False

    # Function to return relay state
    def relayState(self):
        return self.state
```

These changes make it much easier to account for swapping the relay that a device is connected too and makes our code much more readable. On the left is the old code we would need to do to turn on and off the pump and on the right is the new code to turn on and off the pump which I feel is much more readable and robust.



```
relays = Relay()
relays.on(1)
relays.off(1)
```



```
pump = Relay(1)
pump.on()
pump.off()
```

Old New

Review

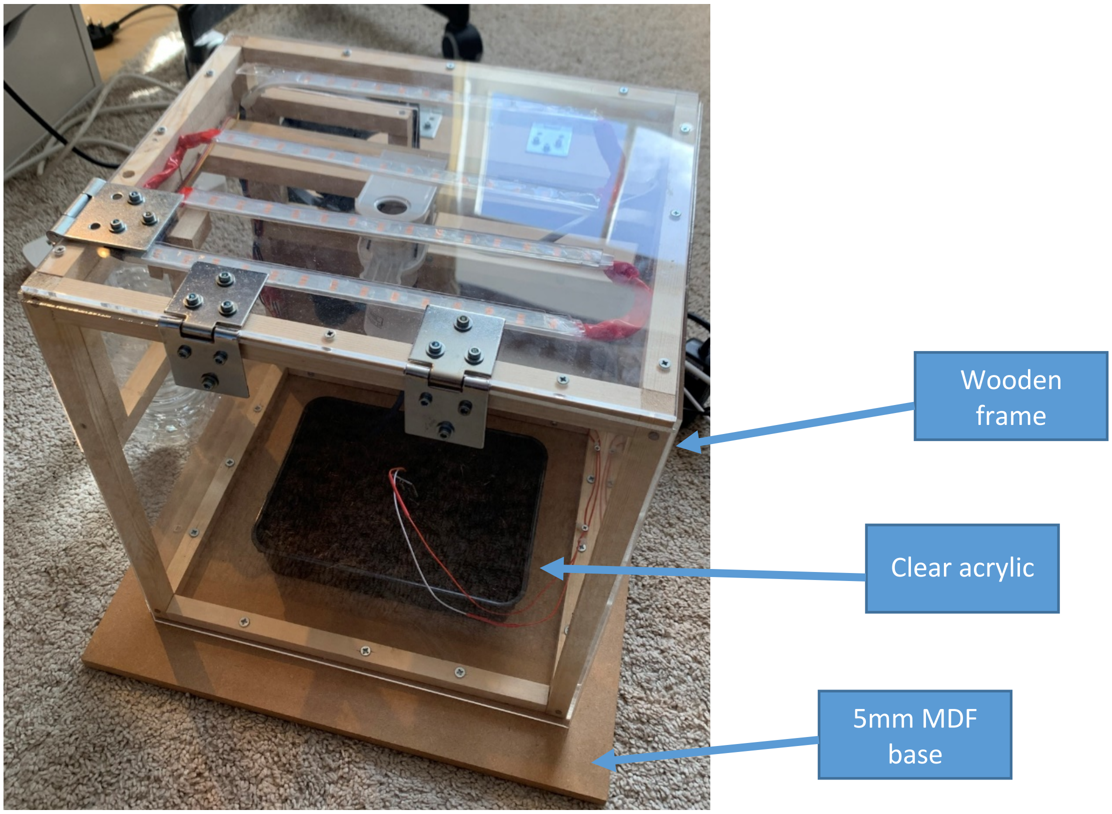
The hardware stage of this project is now complete. I have tested all components of the greenhouse and they function as expected. Only a few minor issues have been encountered and solutions have been implemented for them. It is regrettable that the user will have to run "sudo pigpiod" before running the greenhouse however I have been unable to find a work around and can only assure updates to the Raspberry Pi operating system have stopped the pigpiod daemon from starting on boot.

Iterative Stage 7 – Greenhouse build

Overview

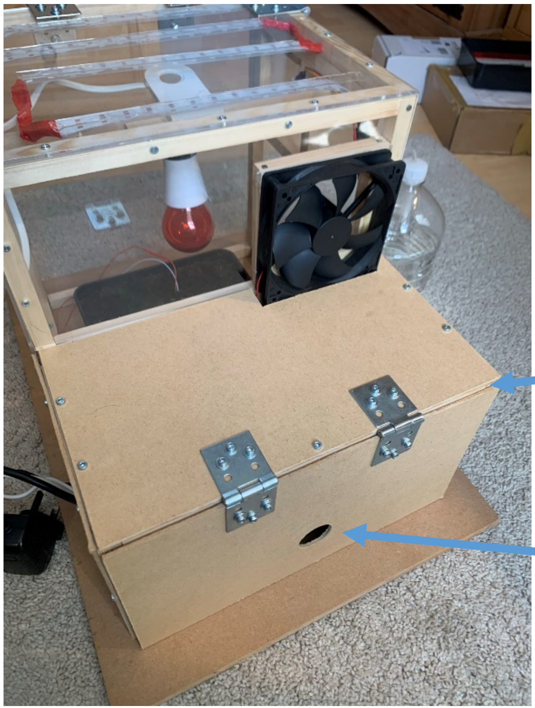
This stage will provide a quick overview of the greenhouse build and the different parts and features of the greenhouse

Greenhouse Body



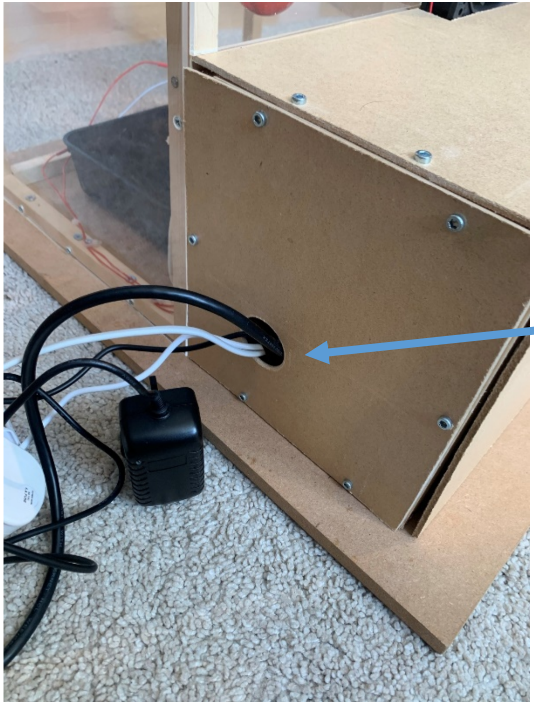
The greenhouse has been constructed from a wooden frame mounted to a MDF base with clear acrylic for the greenhouse section too allow people to observe the plants inside the greenhouse and for natural light to enter. The wooden frame provides ideal mountings for all wires too be attached too.

Electronics Box

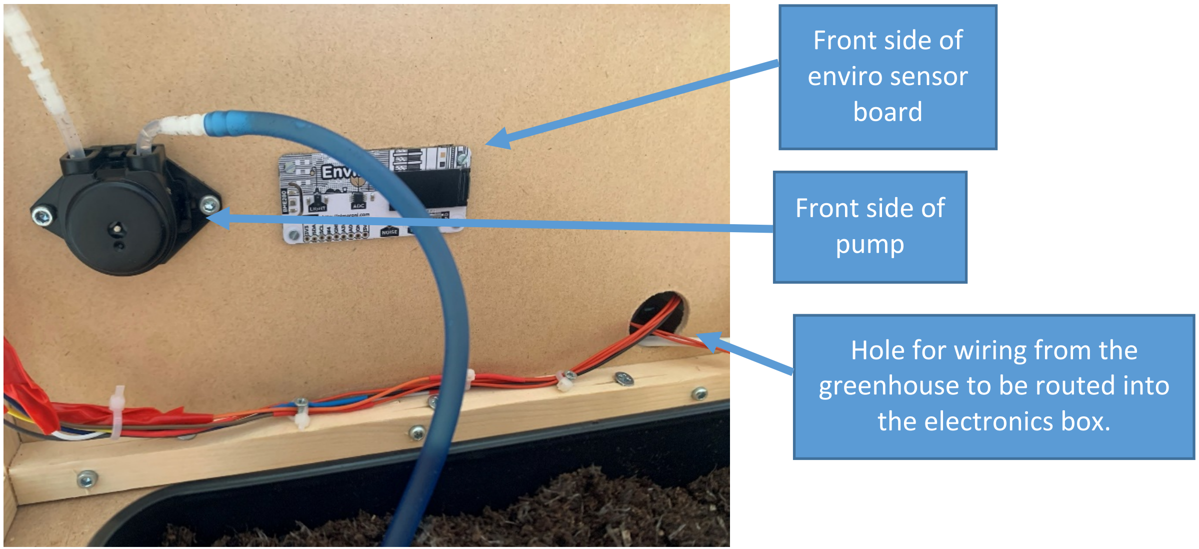


Wooden electronics box

Finger hole to open the door.



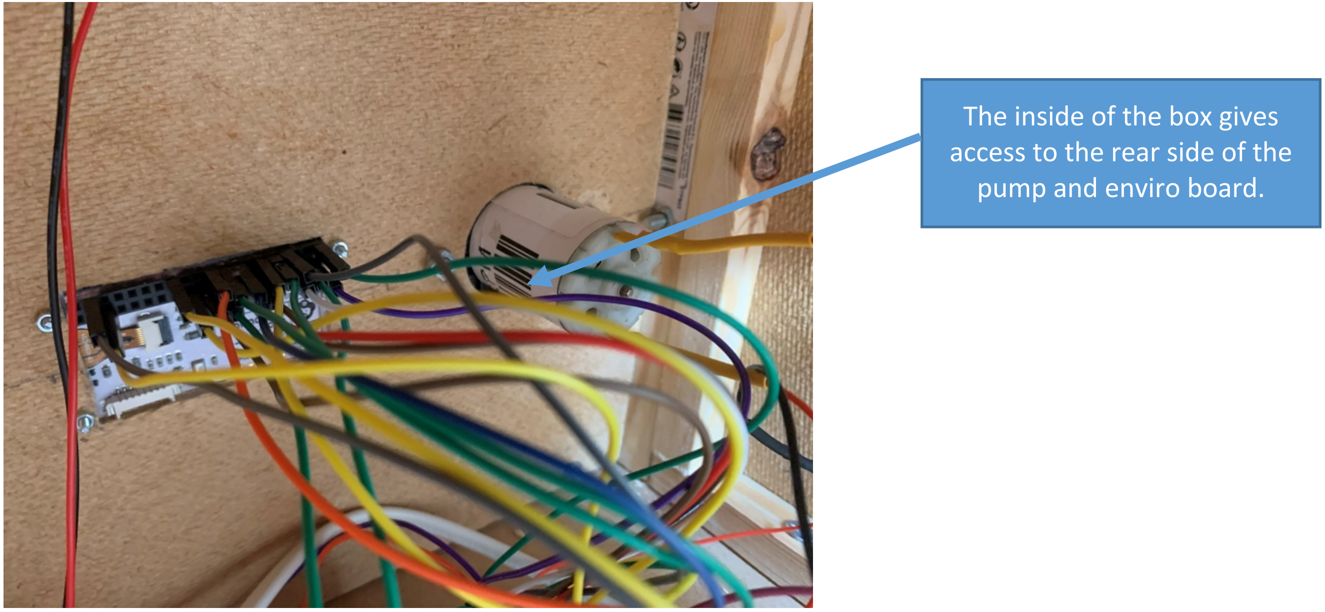
Side hole allows power and data cables to be routed out from the greenhouse.



Front side of enviro sensor board

Front side of pump

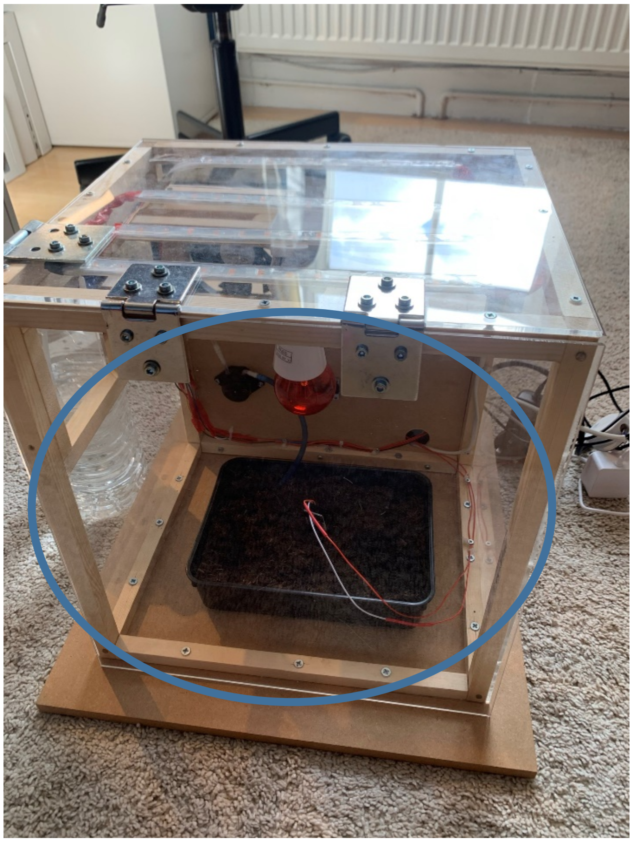
Hole for wiring from the greenhouse to be routed into the electronics box.



The inside of the box gives access to the rear side of the pump and enviro board.

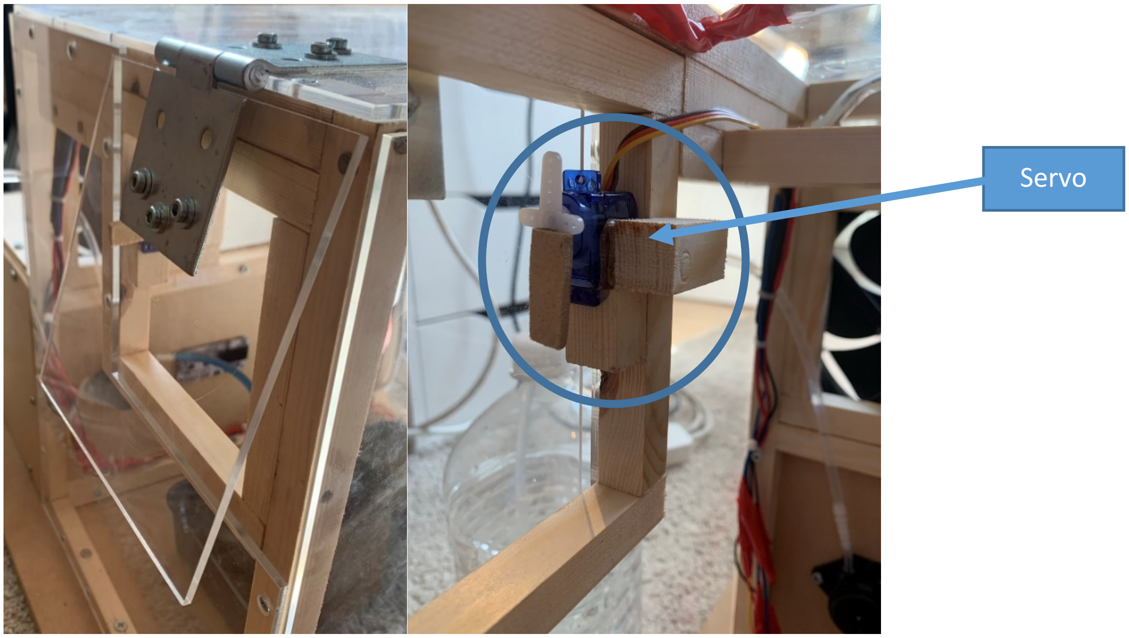
A wooden box half the height of the greenhouse attaches to the back of the greenhouse. This box houses all the electronics such as the Raspberry Pi, Breadboard, moisture sensor and wiring. This helps to protect the components of the Greenhouse from being moved about and accidentally unplugged or damaged. The back side of the box that backs onto the greenhouse holds the enviro sensor board and the pump.

Greenhouse Door



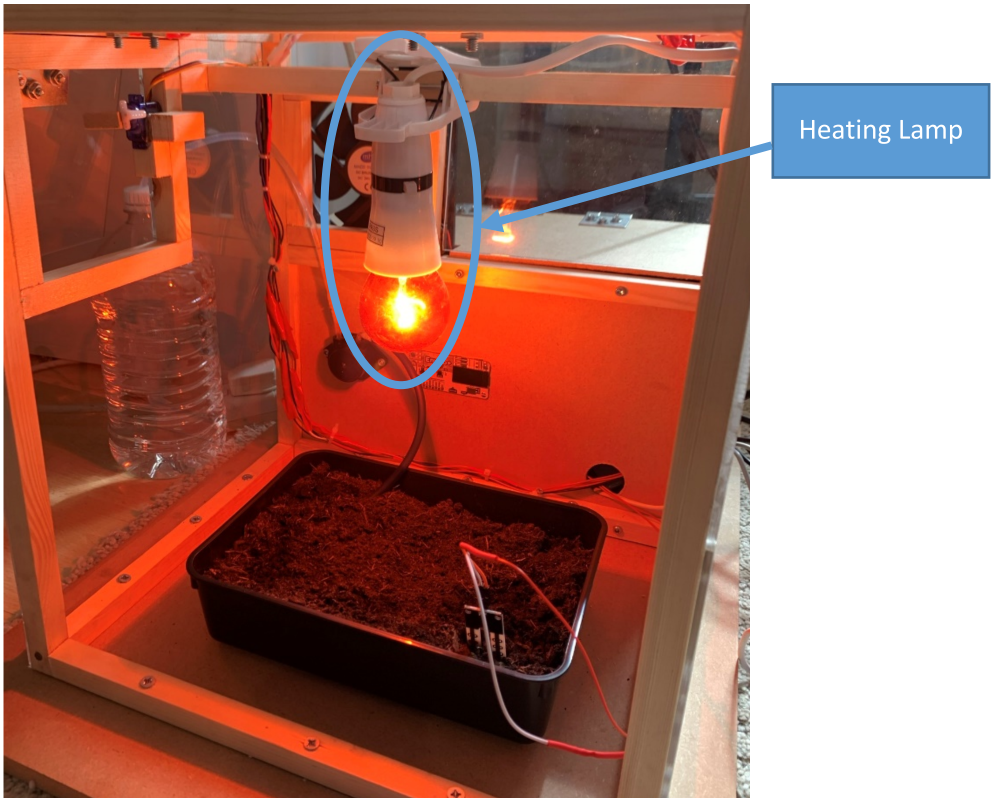
A large door provides easy access into the greenhouse so that the soil tray can be moved in and out of the greenhouse. This door is attached via two large hinges so that the door is sturdy and does not wobble.

Window



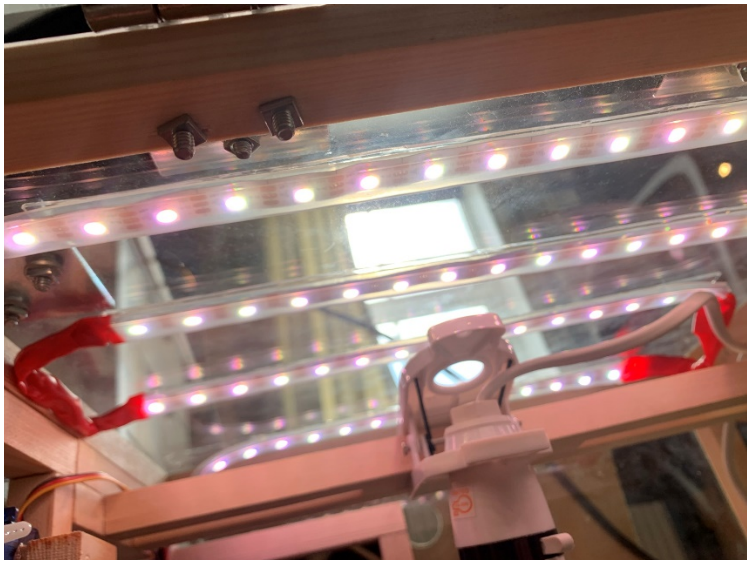
The window is attached to an opening on the side of the greenhouse. The servo motor is mounted behind the window so that when the servo is operated the window will open.

Heating Lamp



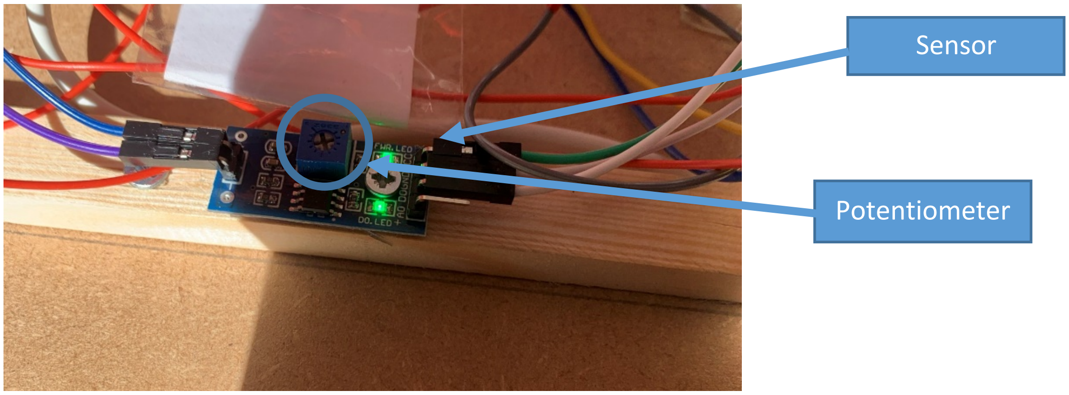
The heating lamp is placed directly above the plant tray so that maximum heating efficiency is achieved. The cable is then routed into the electronics box where it is connected too its relay and then onto its power supply.

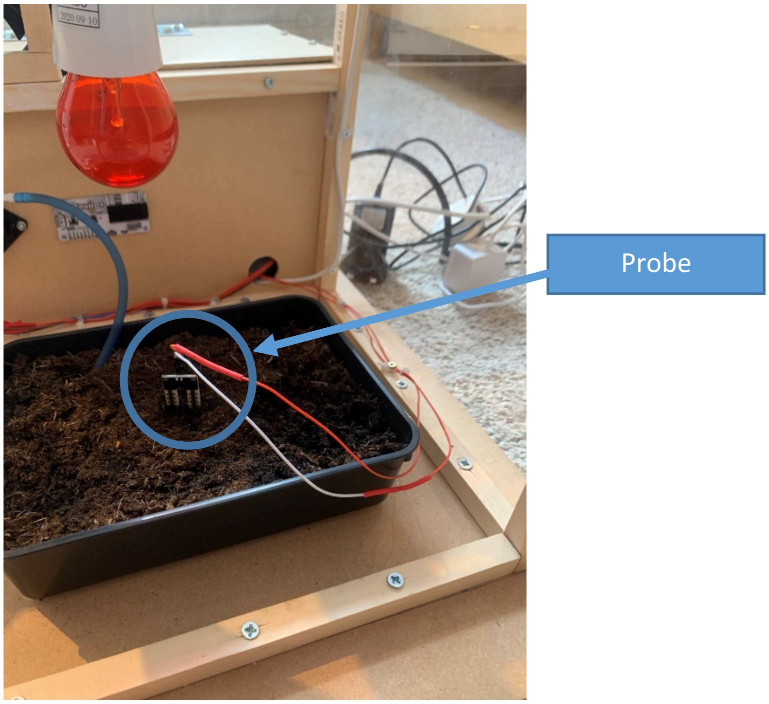
LED Strip



The LED strip has been cut into 4 equal length pieces and then soldered together to allow me to mount it in a snake shape along the top of the greenhouse. To attach the leds to the roof I have used double sided tape.

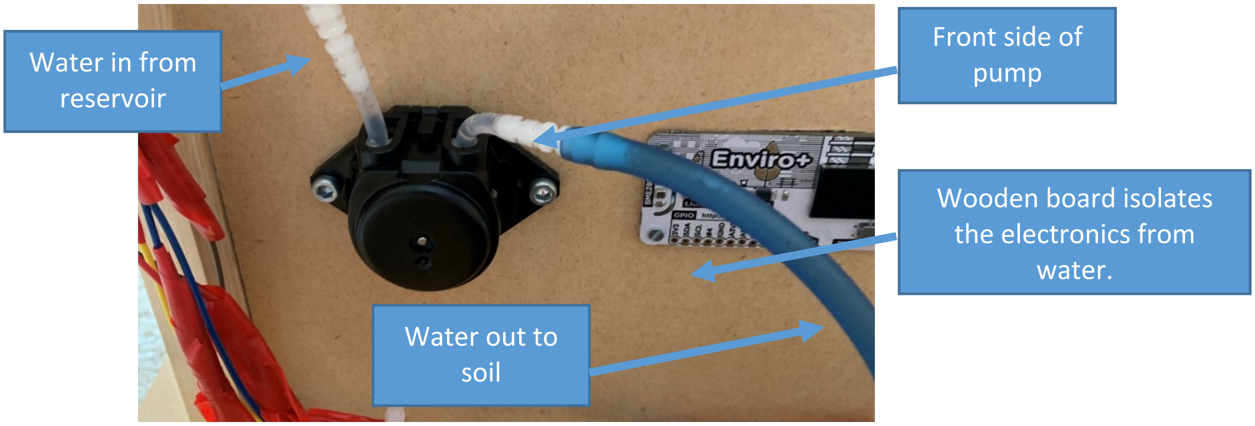
Moisture Sensor / Probe

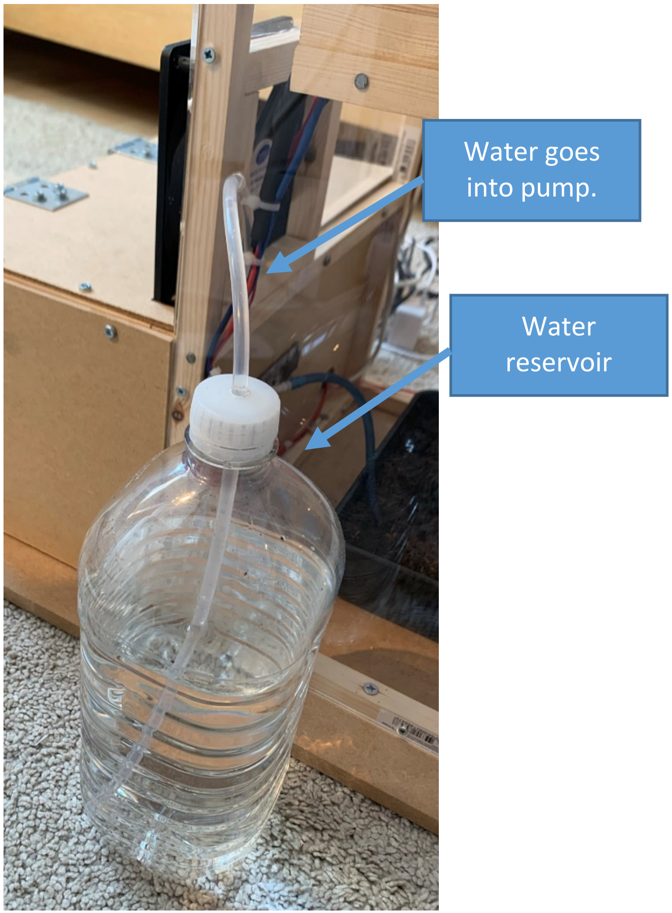


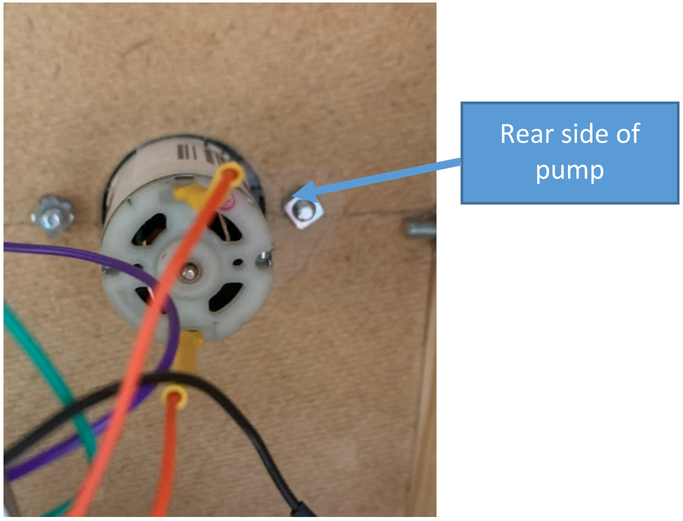


The Moisture probe is routed from the electronics box and into the greenhouse where it is placed into the soil tray. I have mounted the sensor on the lip of the electronics box door so that it is easily accessible when changing the potentiometer.

Pump

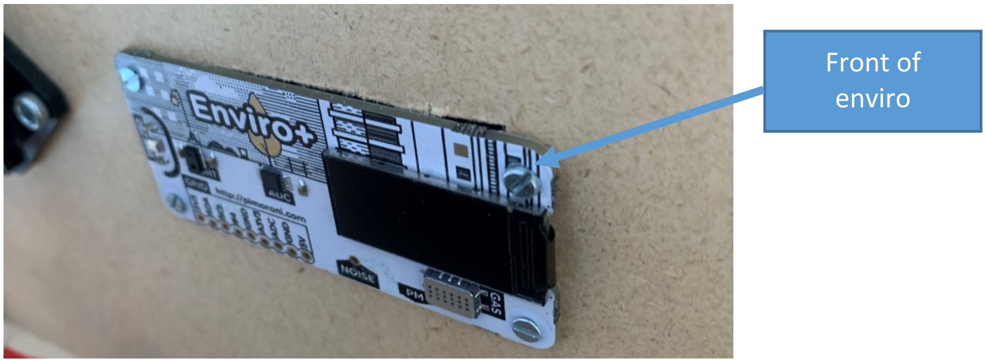


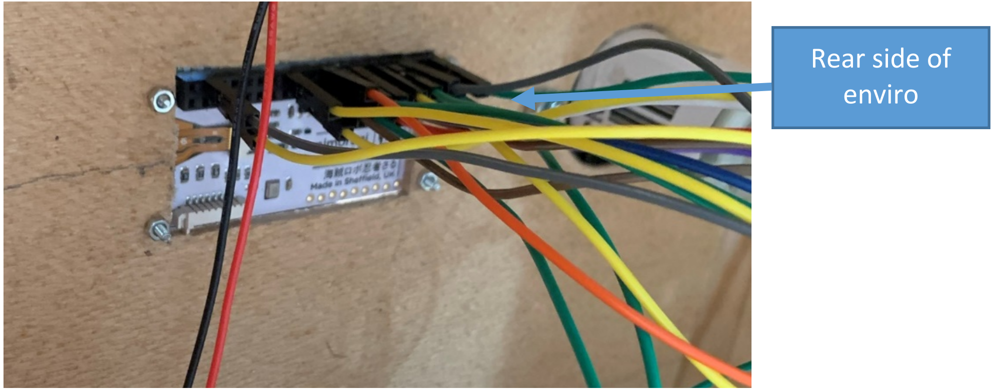




The pump is attached to the back of the electronics box which in turn faces the inside of the greenhouse. The rear of the pump is easily accessible inside the electronics box and is wired into the power source and the relay. The front of the pump is separated from all electronics by the wooden board reducing any risk of water damaging components. A small hole drilled into the side of the greenhouse acrylic allows for the pump pipe to exit the greenhouse and feed into the water reservoir. The other pipe is buried in the soil with a few small holes cut into it so that water is evenly distributed around the soil.

Enviro Board



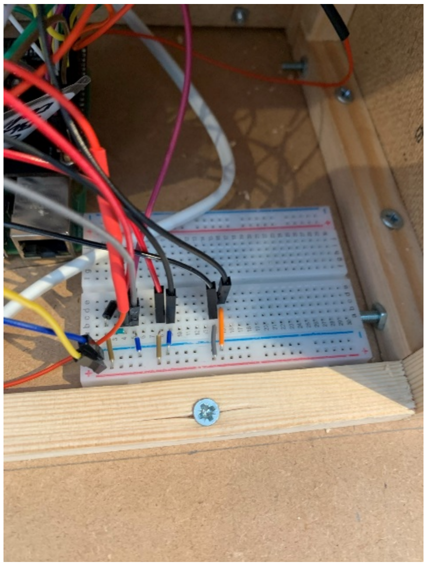


The enviro sensor board is mounted on the same face as the pump. The front of the board is exposed to the inside of the greenhouse where the sensors can make accurate readings of the current environment conditions. The rear side of the board is exposed to the inside of the electronics box where the required pins are wired into the Raspberry Pi.

Fan

The fan is fixed in place above the electronic box and provides fresh air into the greenhouse.

Breadboard



The breadboard attaches to a 5v power supply and provides power in parallel to the fan, pump and then LEDs. I have used the sticky tape on the bottom of the breadboard to stick it down to the MDF base. I positioned the breadboard at the front of the electronics box so that it is easily accessible if anything needs changing.

Review

The greenhouse is fully built and can be controlled manually by writing custom code using the classes I have created. The next stages of my development will now focus on producing the gui and automation features of the Greenhouse.

Iterative stage 8 – GUI

Overview

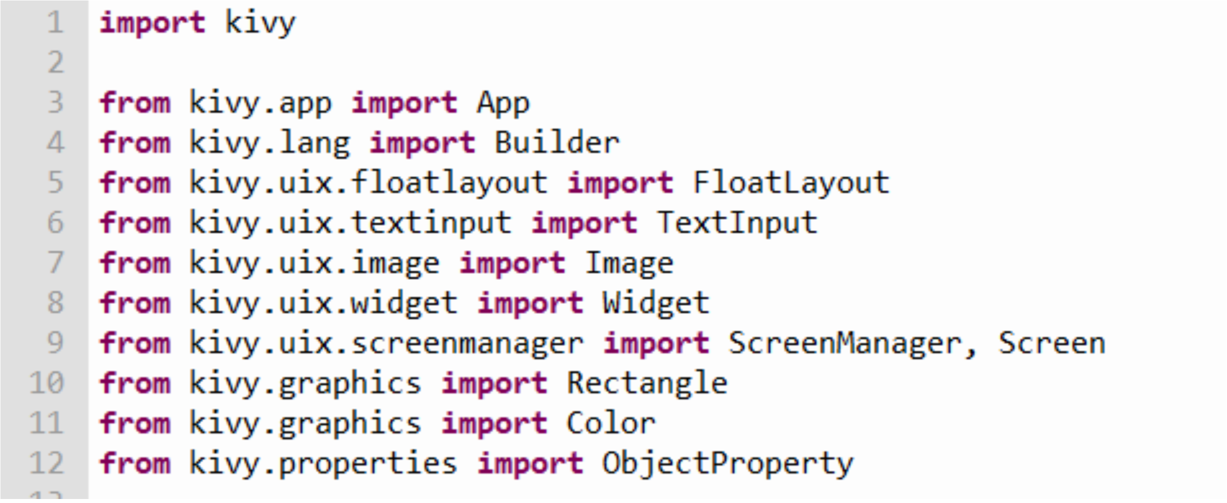
Stage 8 is going to focus on building the graphical user interface that the user will use to communicate and interact with the greenhouse automation system. I will be using the python kivy language too handle the GUI and, in this stage, will build the layout adding all components without developing their function.

Requirements

The GUI must follow the design of the mockups that I have produced earlier in the project. For this stage I will just be placing buttons, text boxes and other components onto the screen they should be in the correct position but interacting with them won't cause anything too happen. I will be adding functionality to the GUI in future stages.

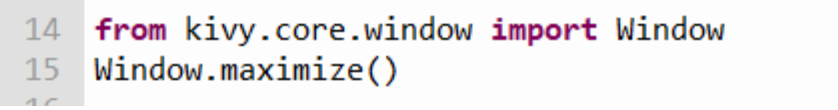
Development Log – Welcome screen

In this development log I will be setting up Kivy so that it is ready to be used in python for the GUI and developing the welcome screen.



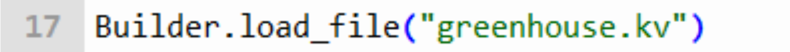
```python
import kivy

from kivy.app import App
from kivy.lang import Builder
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.textinput import TextInput
from kivy.uix.image import Image
from kivy.uix.widget import Widget
from kivy.uix.screenmanager import ScreenManager, Screen
from kivy.graphics import Rectangle
from kivy.graphics import Color
from kivy.properties import ObjectProperty
```

Before I can use Kivy it needs to be imported as a library into the greenhouse.py script. I have also imported all the different classes that will be used to make it easier to access them. This saves me typing out the full-length identifier when using a common class that the kivy library offers.



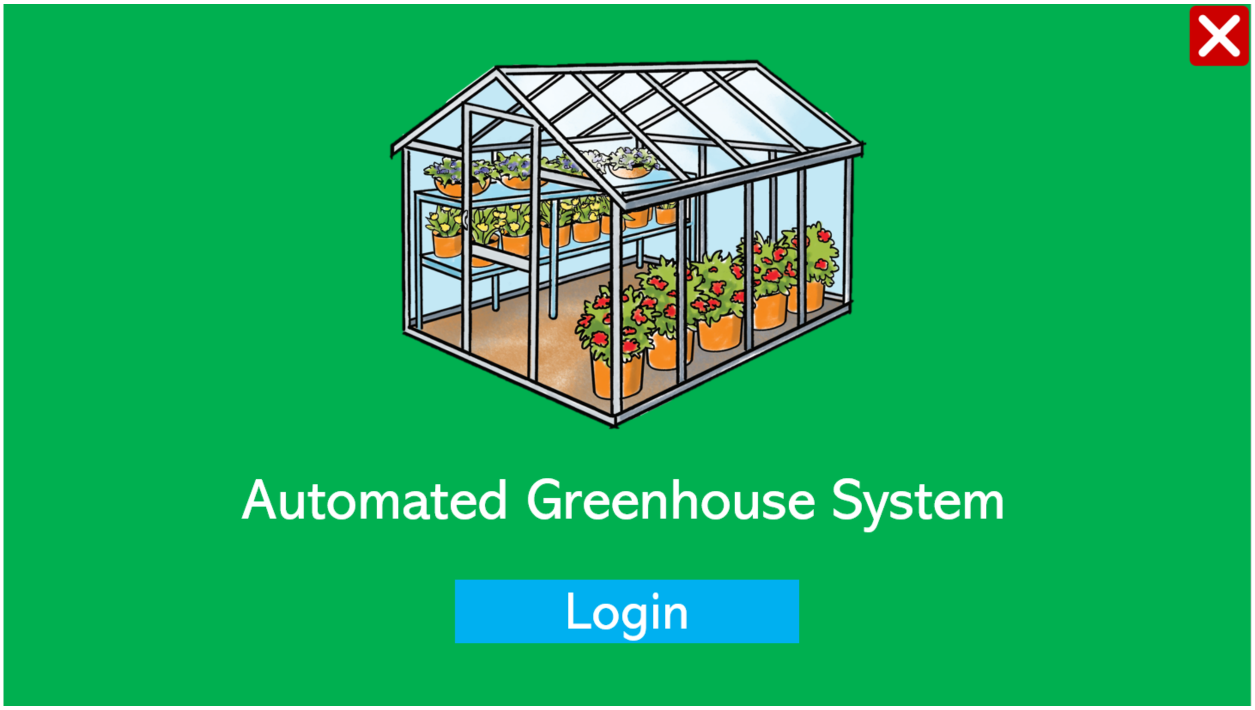
```python
from kivy.core.window import Window
Window.maximize()
```

When the GUI is started, I want it to be automatically full screen. Kivy by default does not maximize the window and so unless specifically set the window will be around ¼ the size of the screen. Here I have imported the window class from the kivy library and then told the window too always be maximized. This comes before any other code to ensure the window is going to be full screen right from the start.

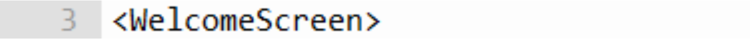


```python
Builder.load_file("greenhouse.kv")
```

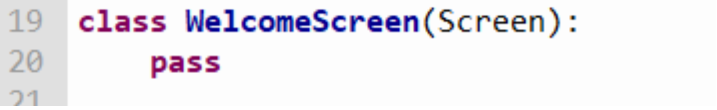
The Kivy library allows you to style the screen in much the same way as CSS. Whilst you can enter kivy objects directly from the python file this can before cluttered and makes it harder to manage many screens. To overcome this kivy allows you to store your objects and screens inside a kv file. Here I am loading the greenhouse.kv file that I am using to style the GUI so that kivy knows to use the contents of this file when rendering the GUI.



When the user starts the application the first screen, they are met with is the welcome screen. The screen is made up for 3 elements a photo, a title and a login button which takes the user to the login page. Above is the mockup of this page that I made earlier in the project I will be basing the Kivy screen off this design. As kivy produces a standard window is already has an exit button in the top right so I won't need to add one myself.



```
<WelcomeScreen>
```

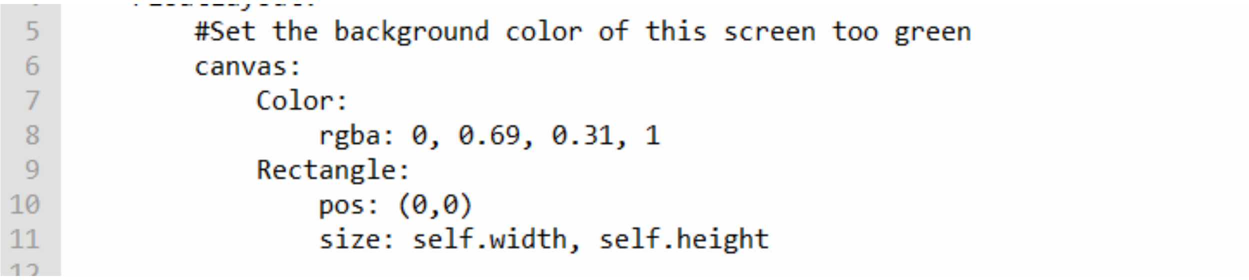


```
class WelcomeScreen(Screen):
    pass
```

Inside the kv file I have defined a new screen called "WelcomeScreen" the kivy syntax to do this is to use the < and > symbols. Each screen in kivy is also a class with the same name as the screen so I have defined the "WelcomeScreen" class inside the python file. All screens will inherit from the screen class which adds required methods from the kivy library. Later in development any functions for a specific page such as the function controlling login validation will need to be added inside its class. However, for the time being I have just added the keyword pass so this class is defined but with no methods.

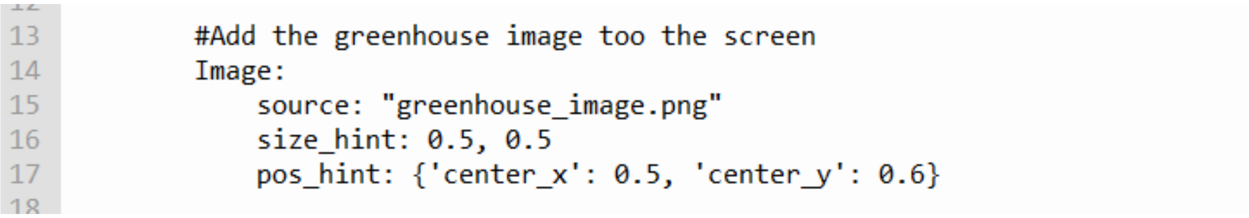
```
FloatLayout:
```

For all the GUI screens I will be using Kivys float layout this allows me to position items on the screen based on a percentage of the screen and size them based on a percentage of the screen. This means my GUI will be responsive too changes in the size of the screen as an object that takes up 50% of the window in the x axis will always take up the same percentage no matter how the user decides to readjust the window width and height.



```
5        #Set the background color of this screen too green
6        canvas:
7            Color:
8                rgba: 0, 0.69, 0.31, 1
9            Rectangle:
10               pos: (0,0)
11               size: self.width, self.height
```

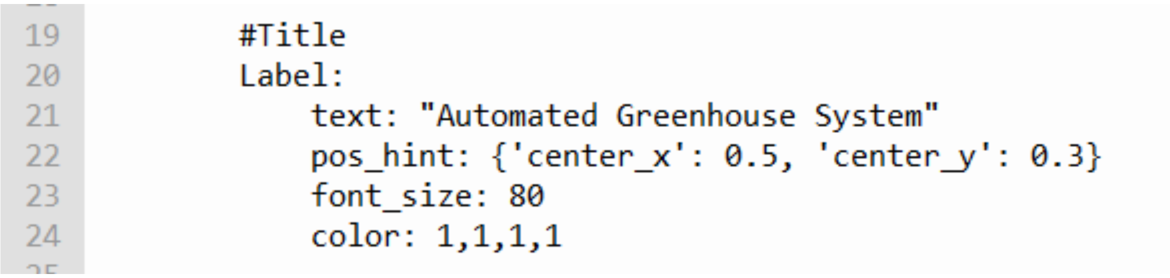
For each screen the background color will be light green. Too do this I have drawn a rectangle onto the screen which has a color of green and a width and heigh equal to that of the window. Kivy uses rgba for its colours with each value expressed as a fraction of 1 so too convert standard rgb into the kivy standard each r, g and b value must be divided between 255. This snippet of code will be repeated at the start of each screen in the kv file.



```
13       #Add the greenhouse image too the screen
14       Image:
15           source: "greenhouse_image.png"
16           size_hint: 0.5, 0.5
17           pos_hint: {'center_x': 0.5, 'center_y': 0.6}
```

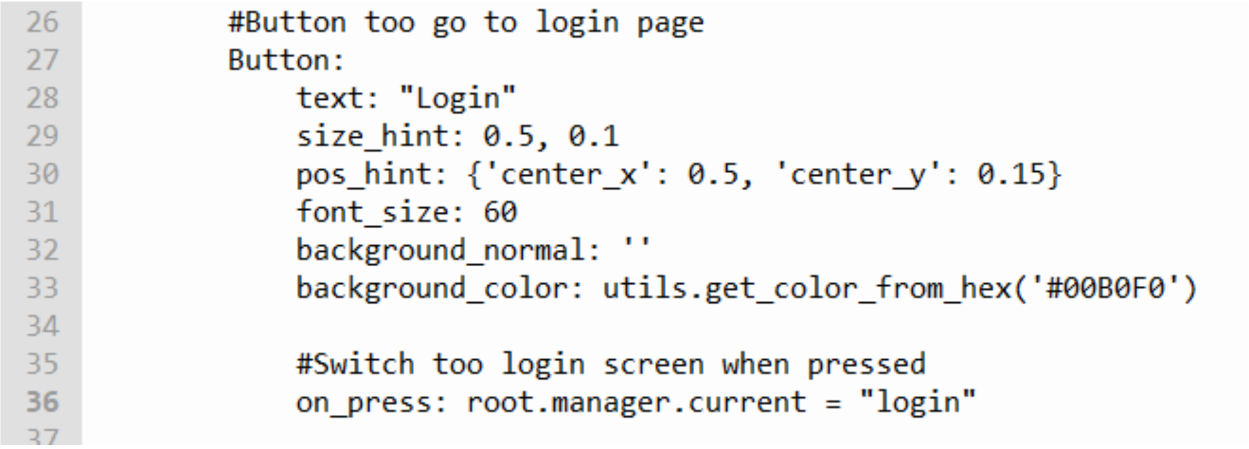
Kivy can render png files onto the screen with a transparent background. Too do this the image keyword is used. Line 15 lets kivy know when the image too be rendered can be found in this case in the same directory with the name “greenhouse_image.png”. Line 16 lets kivy know how big the image should be the first value is what percentage of the x axis the image should take up and the second value what percentage of the y axis. So, in this case I have defined that the image should be the width of 50% of the screen and the height of 50% of the screen. This is useful as if the window is readjusted too be bigger or the GUI is run on a different computer the image will still be in the same proportion as before when compared to the screen as a percentage. In line 17 I have positioned the image too do this I have specified that the x axis center position of the image should be 50% of the screen this is equivalent to the middle of the screen and then I have specified that the image should be centered on the y axis 60% up from the bottom of the screen. It should be noted that kivy takes all measurements from the bottom left of the screen so too position the image kivy calculates the height and width then works out 50% of the width and 60% of the height and positions the image in that location. This is constantly evaluated so any changes too screen size automatically repositions the elements in the screen.

Note on calculating positions:
To help me position the elements of my gui I printed off the GUI renders on paper and measured the height and width of the A4 page. This then allowed me to measure the distance in the x and y directions from the bottom left of the page too the center of an object and then divide this value by the overall width or height of the page. Thus, obtaining the percentage position value which I could then plug into kivy. This made it much easier to work with the float layout in kivy and saved me a lot of time otherwise spent moving elements around in kivy trying to make them sit in the correct position.



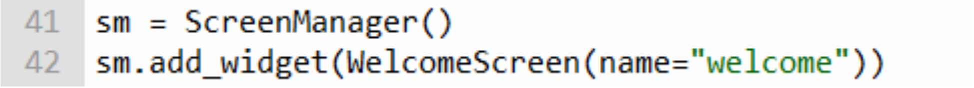
```
19          #Title
20          Label:
21              text: "Automated Greenhouse System"
22              pos_hint: {'center_x': 0.5, 'center_y': 0.3}
23              font_size: 80
24              color: 1,1,1,1
```

Below the image of the greenhouse there will be a title. To add text into a kivy screen the Label keyword is used. Just like with an image it is given a position hint on line 22 to let kivy know where to place the title. A kivy label has a text property that is set on line 21 to specify the text of the label. A font size is also given to the label along with a color.



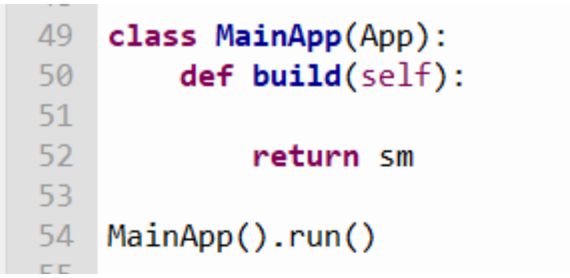
```
26          #Button too go to login page
27          Button:
28              text: "Login"
29              size_hint: 0.5, 0.1
30              pos_hint: {'center_x': 0.5, 'center_y': 0.15}
31              font_size: 60
32              background_normal: ''
33              background_color: utils.get_color_from_hex('#00B0F0')
34
35              #Switch too login screen when pressed
36              on_press: root.manager.current = "login"
```

Finally for this screen a button is added which takes the user to the login page. The button by default has a black background along with a slight opaque tint I have decided to override this by setting the background_normal to remove the tint and the colour too be blue as per my GUI mockups. Kivy has got a library which allows it to use straight up hex colour values and in this case, I have elected to use this function too set the colour of the button on line 33. When the button is pressed the screen needs to be moved to the login screen. This will happen regardless of any python side processing and so this transition can be handled inside the kivy file by setting the screen manage current page too equal “login” which will be the name of the login page. This happens when the button is pressed.



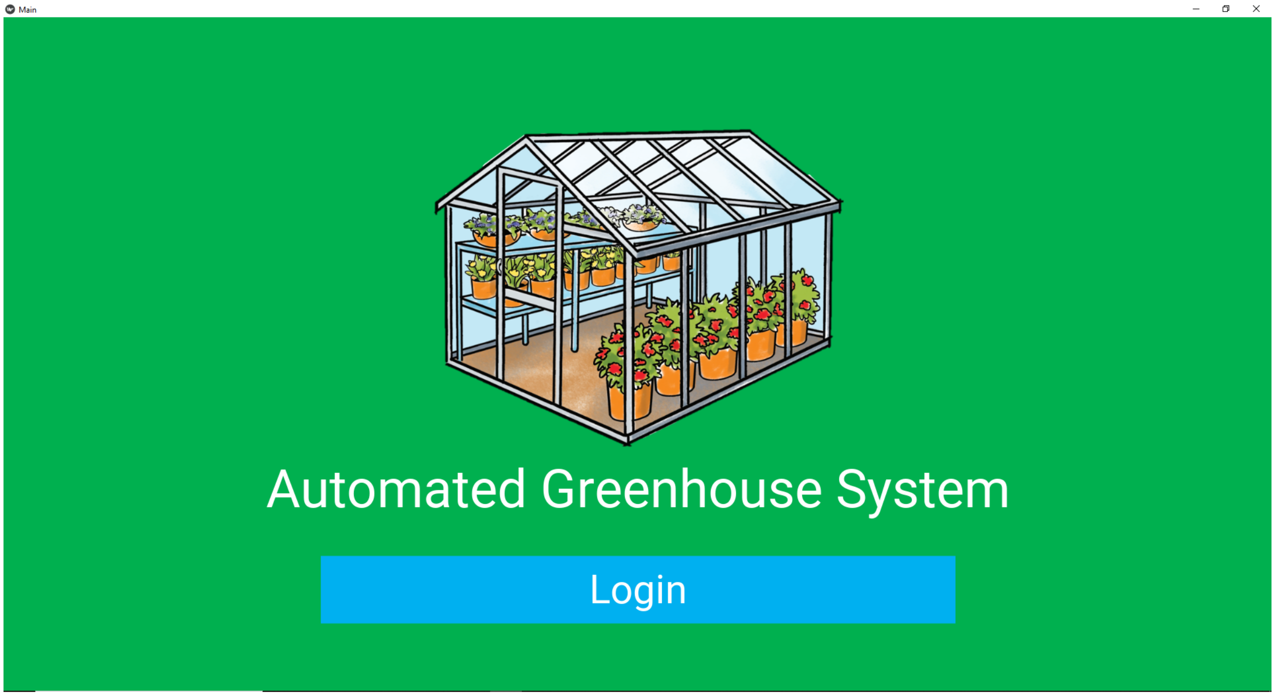
```
41  sm = ScreenManager()
42  sm.add_widget(WelcomeScreen(name="welcome"))
```

The welcome screen is now complete and when ran will look the same as my GUI render of this screen. Before kivy can run the script a screen manager needs to be added inside the python file. The screen manager is responsible for controlling which screen is displayed too the user. This handles transitions of the screen when I want to complete some processing such as login validation inside python and based on the results of that move the user to a specific screen. It also gives the different screens their name which is then used too transition between pages such as in line 36 when I’ve swapped the screen to the login screen. The first widget added to the screen manager will be the one that is shown to the user on startup and so I have added the Welcome Screen first. I have given the screen a name of “welcome” and this is the key word that will be used if I ever need to swap the screen being displayed too the welcome screen.

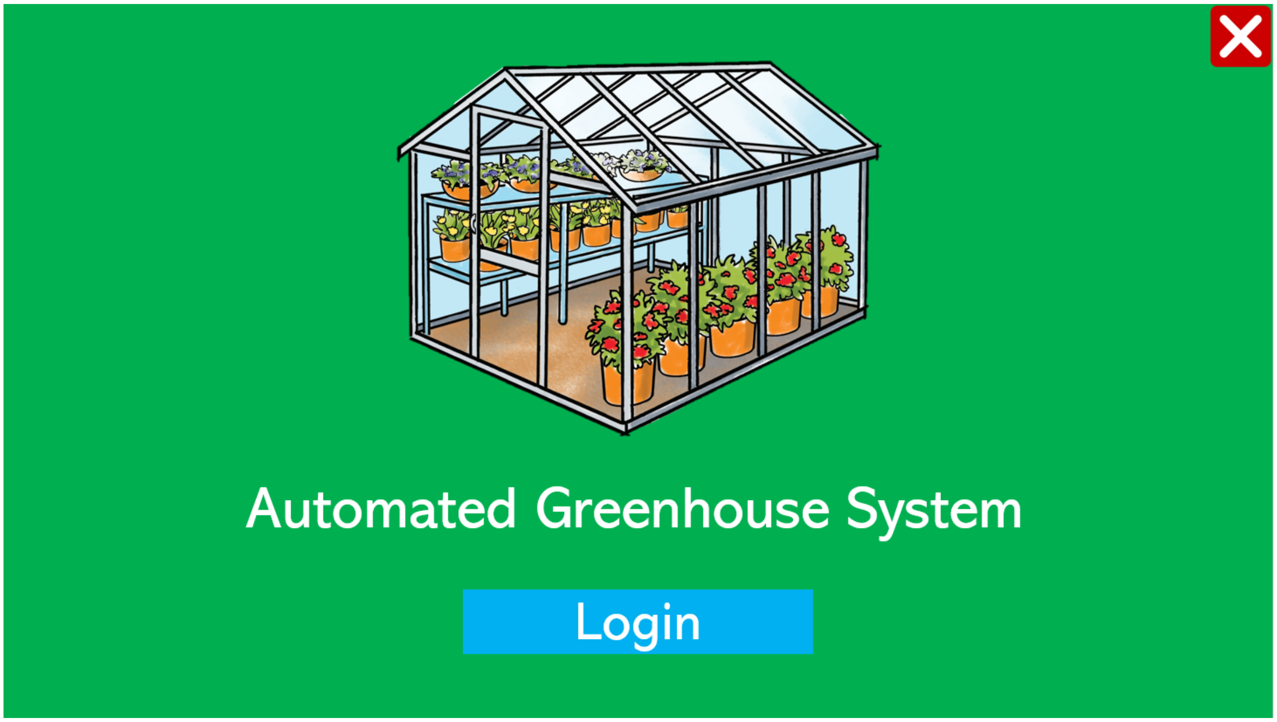


```
49 class MainApp(App):
50     def build(self):
51 
52         return sm
53 
54 MainApp().run()
```

Finally, the MainApp class is declared which inherits from App this is a Kivy class which produces a standard window GUI. When the class is built it will return the screen manager thus allowing me to control the current screen shown to the user. On line 54 the MainApp is ran which will launch the GUI to the user.



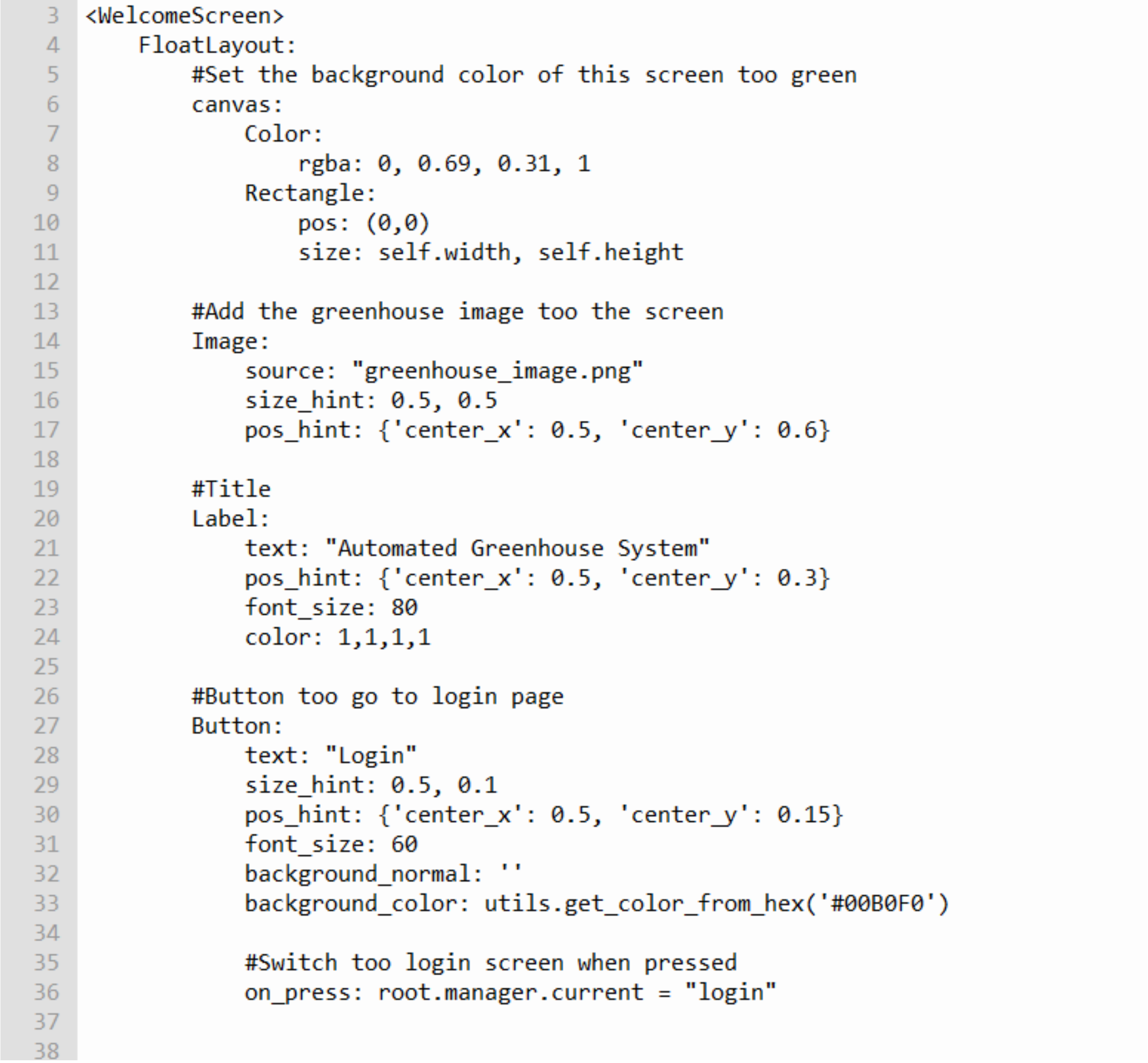
Kivy



Mockup

Above is a screenshot of the welcome page as developed using kivy. I have also included a screen shot of the GUI mockup made previously for comparison. The differences are minimal and mainly relate to the different fonts used by kivy and when producing the mockup.

Complete welcome screen code



```
<WelcomeScreen>
    FloatLayout:
        #Set the background color of this screen too green
        canvas:
            Color:
                rgba: 0, 0.69, 0.31, 1
            Rectangle:
                pos: (0,0)
                size: self.width, self.height

        #Add the greenhouse image too the screen
        Image:
            source: "greenhouse_image.png"
            size_hint: 0.5, 0.5
            pos_hint: {'center_x': 0.5, 'center_y': 0.6}

        #Title
        Label:
            text: "Automated Greenhouse System"
            pos_hint: {'center_x': 0.5, 'center_y': 0.3}
            font_size: 80
            color: 1,1,1,1

        #Button too go to login page
        Button:
            text: "Login"
            size_hint: 0.5, 0.1
            pos_hint: {'center_x': 0.5, 'center_y': 0.15}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #Switch too login screen when pressed
            on_press: root.manager.current = "login"

```

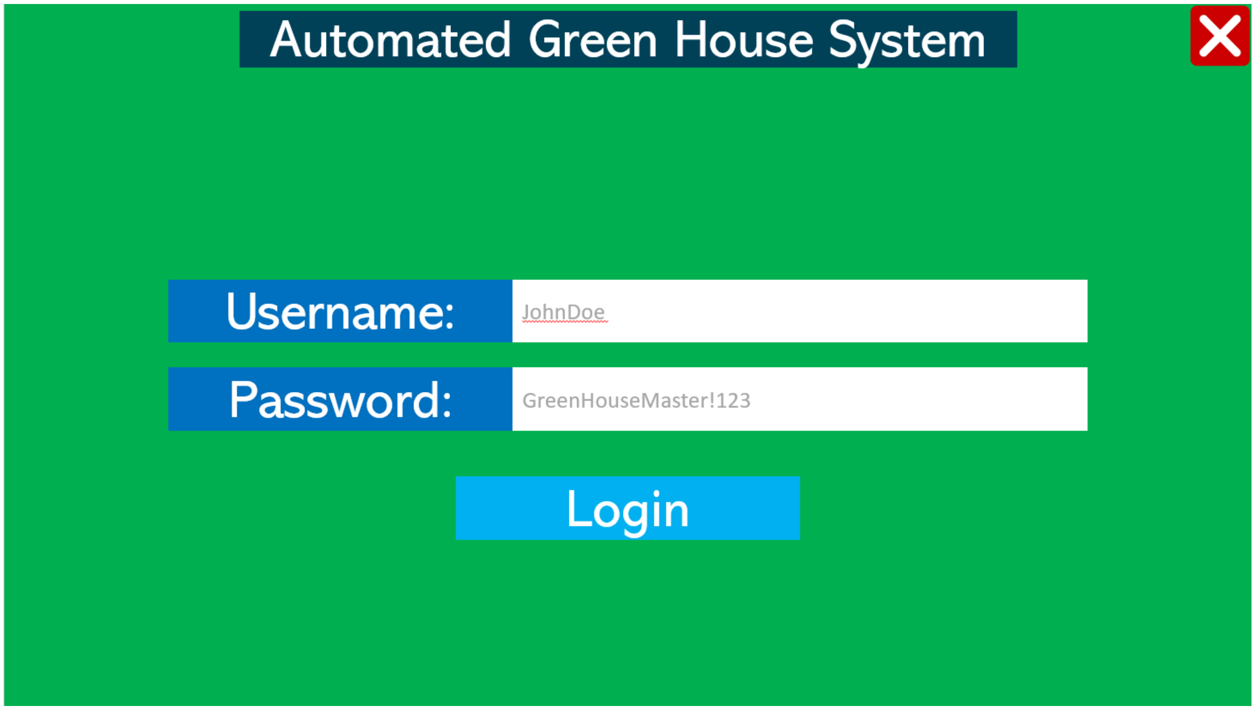
Test Plan – Welcome Screen

The only thing too test on this page is the login button which should move the screen onto the login page. As this page has not yet been developed I have just quickly setup a blank screen so I can test if the screen is indeed changed when the button is pressed.

Test Number	Test Plan	Expected Outcome	Actual Outcome	Pass/Fail
1	Click the login button	The welcome screen will transition to the login screen	The screen swapped too the login screen	Pass

Development Log – Login Screen

The login screen will need to have two text input boxes one for the username to be entered and one for the password of the user to be entered. A login button will also be on the screen which when pressed will eventually validate the user and act accordingly. However, this stage is only focusing on the kivy layout, so I won't be adding the login function yet.

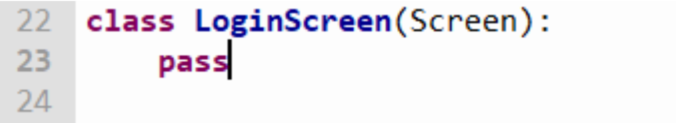


Automated Green House System

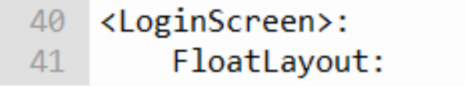
Username: JohnDoe

Password: GreenHouseMaster!123

Here is the mockup for the login screen that I have developed, and I will be using this too layout my kivy screen.

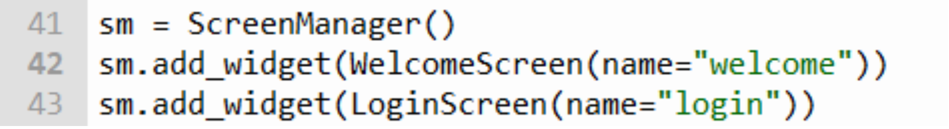


```
class LoginScreen(Screen):
    pass

```



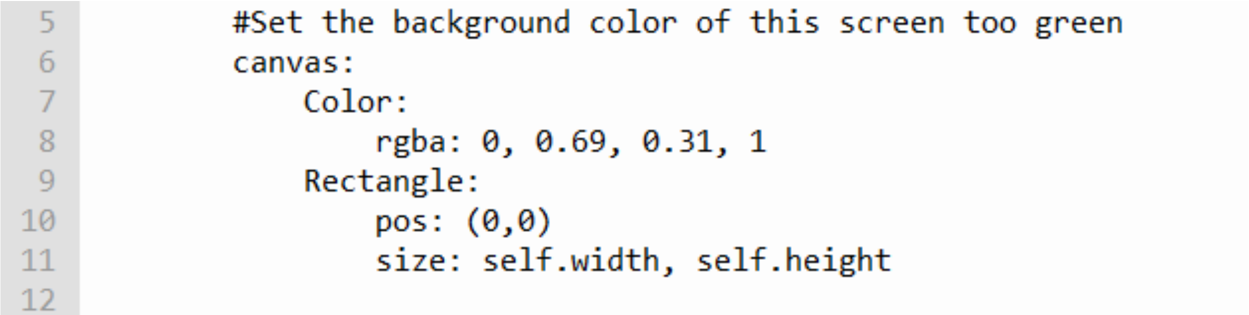
```
<LoginScreen>:
    FloatLayout:
```

Once again, I have declared the loginscreen class inside python and created a new screen inside the kivy file that once again uses float layout. Inside the python loginscreen class there will eventually be a function that will be ran when the user presses the login button. This function will then handle the validation of the user and if successful use the screen manager to swap them to the main menu screen.



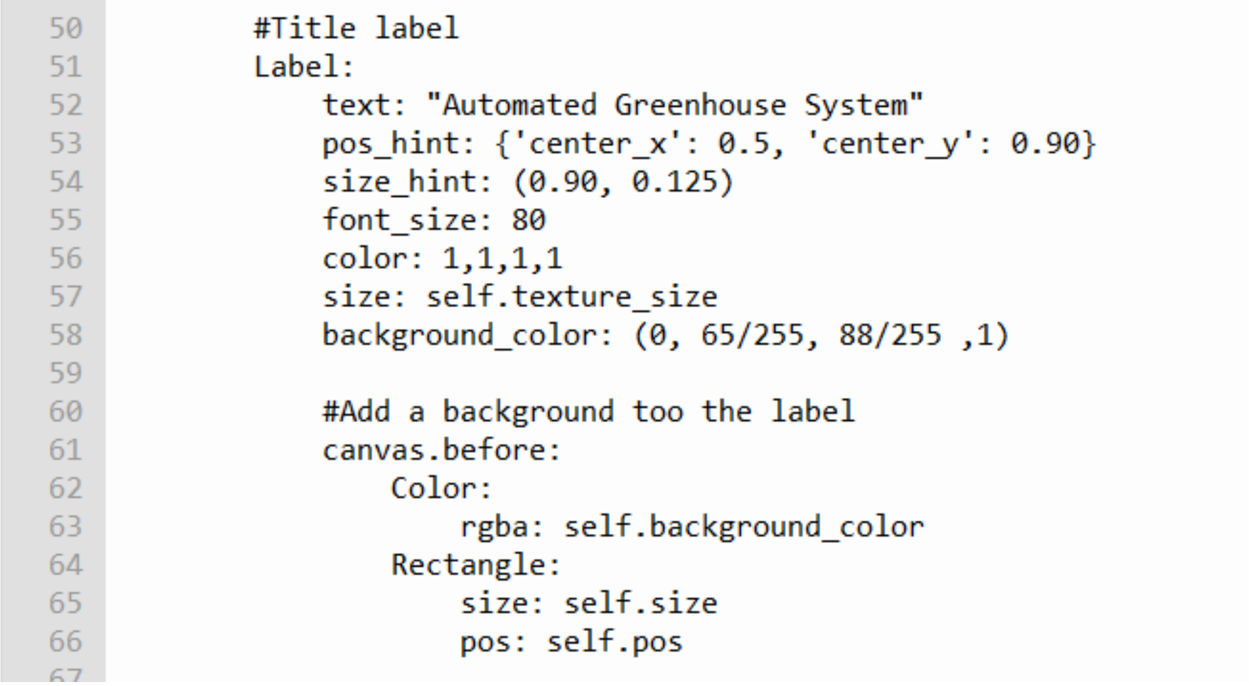
```
sm = ScreenManager()
sm.add_widget(WelcomeScreen(name="welcome"))
sm.add_widget(LoginScreen(name="login"))
```

The screen is added as a widget of the screen manager.



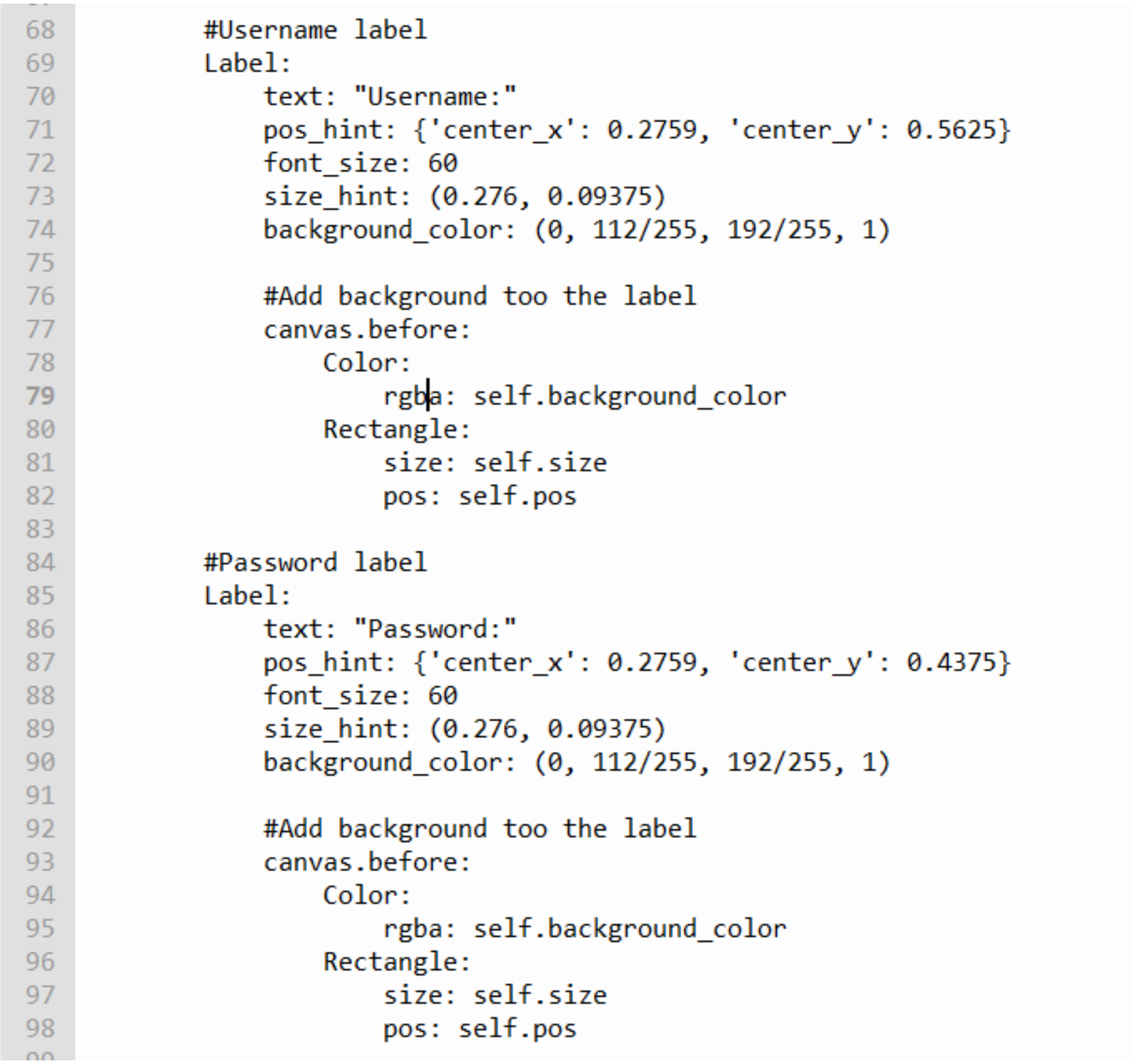
```
5          #Set the background color of this screen too green
6          canvas:
7              Color:
8                  rgba: 0, 0.69, 0.31, 1
9              Rectangle:
10                 pos: (0,0)
11                 size: self.width, self.height
12
```

Background is made green



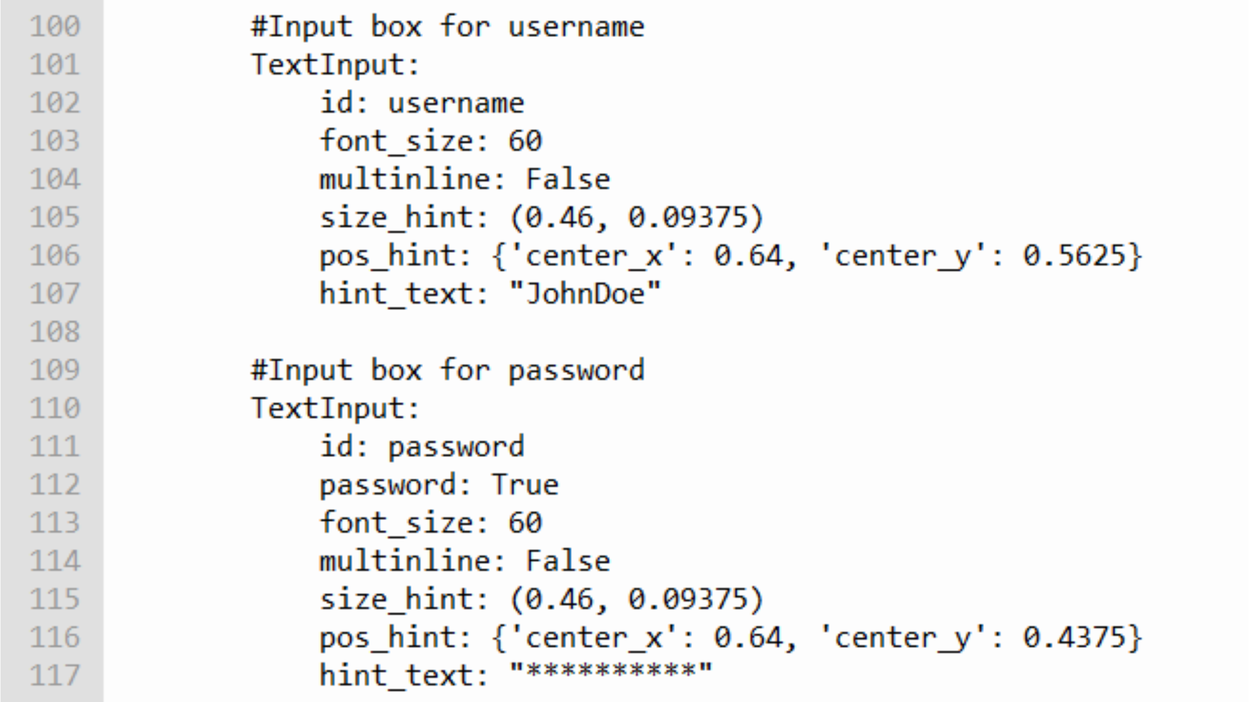
```
50         #Title label
51         Label:
52             text: "Automated Greenhouse System"
53             pos_hint: {'center_x': 0.5, 'center_y': 0.90}
54             size_hint: (0.90, 0.125)
55             font_size: 80
56             color: 1,1,1,1
57             size: self.texture_size
58             background_color: (0, 65/255, 88/255 ,1)
59
60             #Add a background too the label
61             canvas.before:
62                 Color:
63                     rgba: self.background_color
64                 Rectangle:
65                     size: self.size
66                     pos: self.pos
67
```

A label is used to add the title to the top of the screen. As the title will have a dark blue background, I have drawn a rectangle behind the label which will have a background colour equal to the rgba value on line 58 in this case light blue and set the rectangles size and position equal to that of its parent. In this case the parent’s size and position is set when I sized and positioned the label.



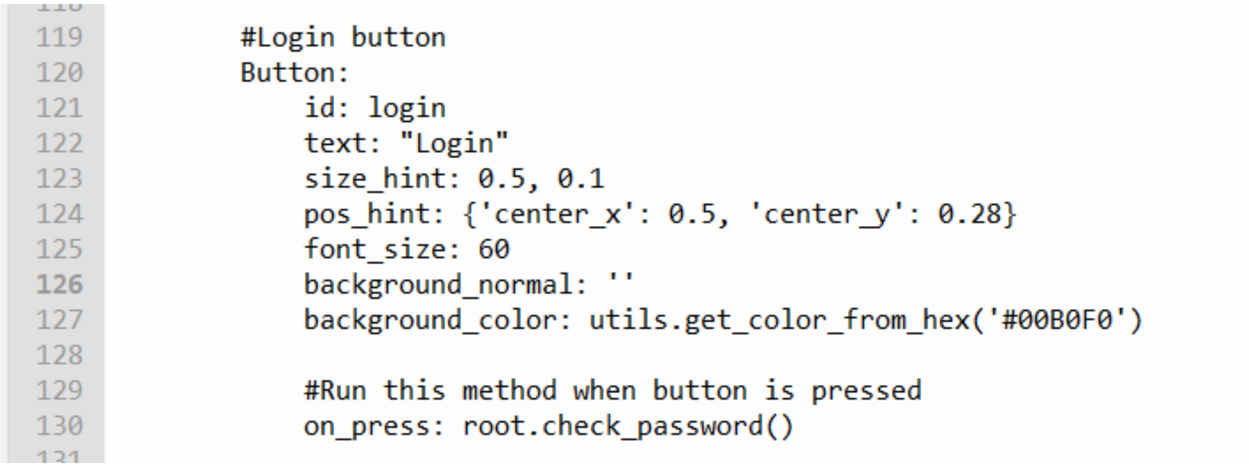
```
        #Username label
        Label:
            text: "Username:"
            pos_hint: {'center_x': 0.2759, 'center_y': 0.5625}
            font_size: 60
            size_hint: (0.276, 0.09375)
            background_color: (0, 112/255, 192/255, 1)

            #Add background too the label
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Password label
        Label:
            text: "Password:"
            pos_hint: {'center_x': 0.2759, 'center_y': 0.4375}
            font_size: 60
            size_hint: (0.276, 0.09375)
            background_color: (0, 112/255, 192/255, 1)

            #Add background too the label
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos
```

Two further labels one is added which sit next to the username and password input boxes too let the user know where to input their login details. Both have their own background boxes which are added in the same was as for the title but with a light blue background.



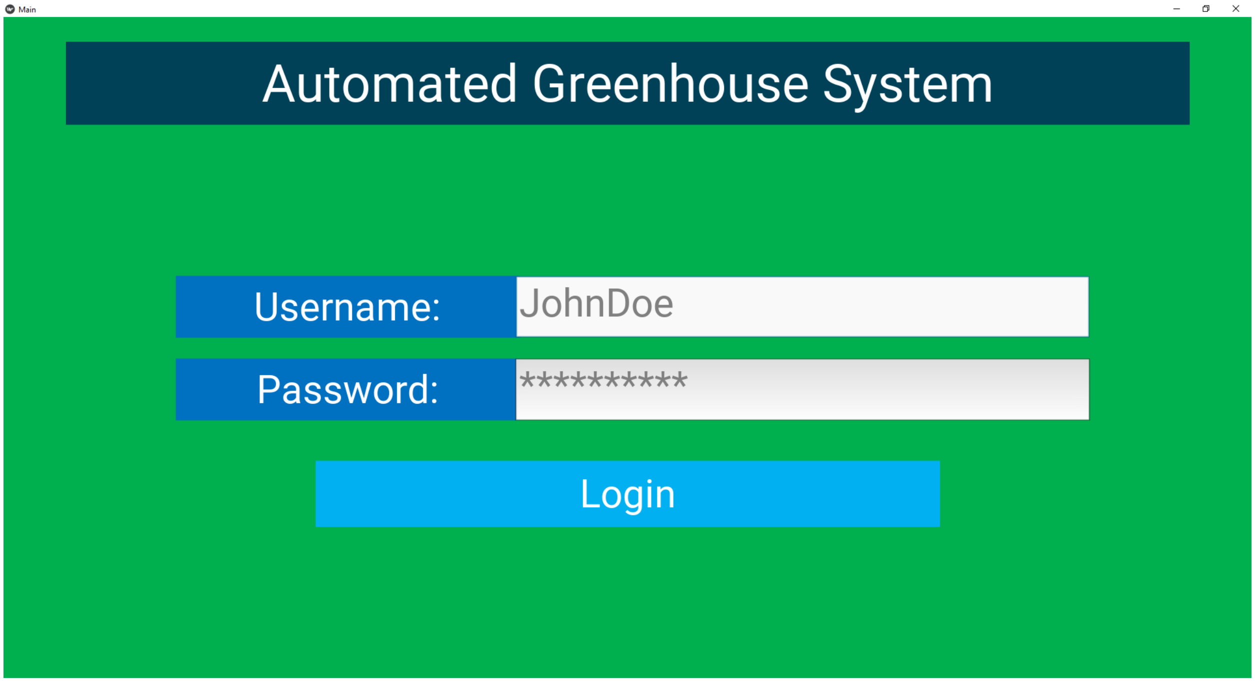
```
        #Input box for username
        TextInput:
            id: username
            font_size: 60
            multinline: False
            size_hint: (0.46, 0.09375)
            pos_hint: {'center_x': 0.64, 'center_y': 0.5625}
            hint_text: "JohnDoe"

        #Input box for password
        TextInput:
            id: password
            password: True
            font_size: 60
            multinline: False
            size_hint: (0.46, 0.09375)
            pos_hint: {'center_x': 0.64, 'center_y': 0.4375}
            hint_text: "**********"
```

I have added two text input boxes onto the page which are used to capture the user's login details. They are both given an id of "username" and "password" respectively this is to allow me to access their value inside of python by referencing their id. The multiline parameter is set as False for both boxes so that the user can't add more than one line of text. A hint text is added so the user sees an example of the information to be entered too prompt them to enter their own details and avoid any confusion. For the password text input box, I have set the password parameter to equal true. This means that any text inputted into the box will be represented as a "*" so that the user's password is hidden from view for security reasons.



```
        #Login button
        Button:
            id: login
            text: "Login"
            size_hint: 0.5, 0.1
            pos_hint: {'center_x': 0.5, 'center_y': 0.28}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #Run this method when button is pressed
            on_press: root.check_password()
```

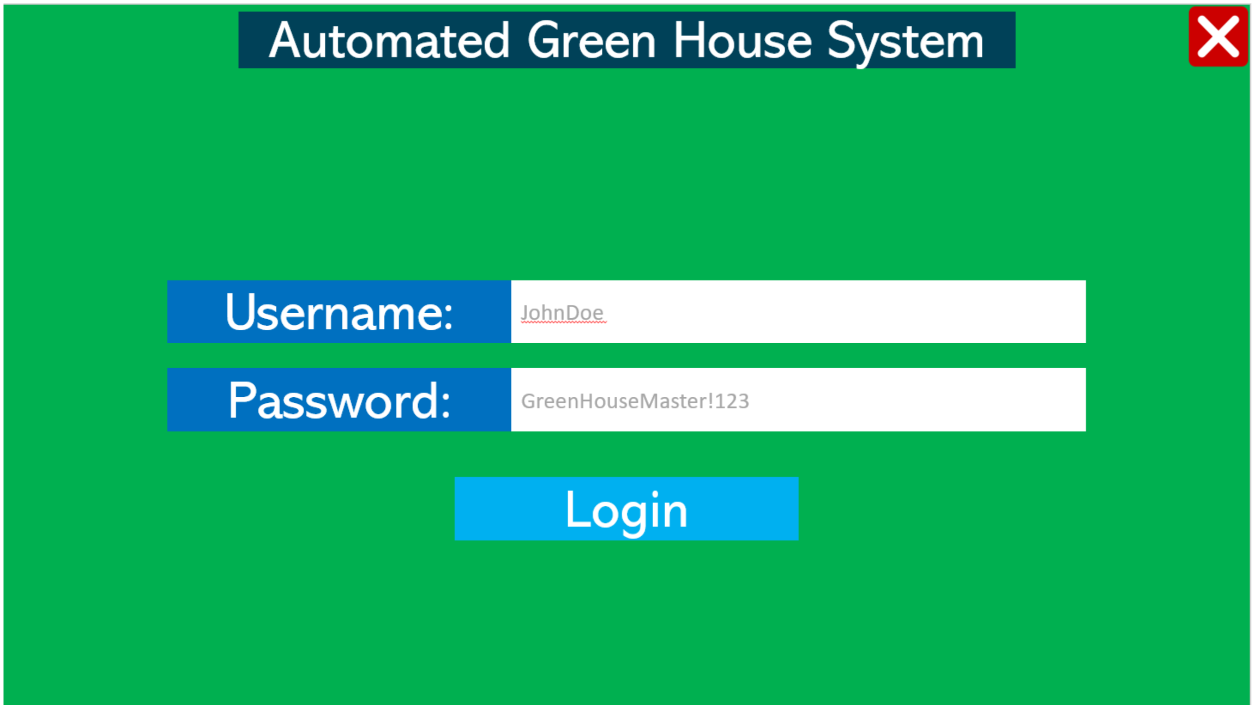
Finally, I have added the login button at the bottom of the page. In the case of this button when it is pressed a function called "check_password" will be ran. This function will be a part of the LoginScreen class and can perform the required actions of the user validation. The "root" means that kivy knows the function belongs to the screens class inside the python file.



Username: JohnDoe

Password: **********

Kivy



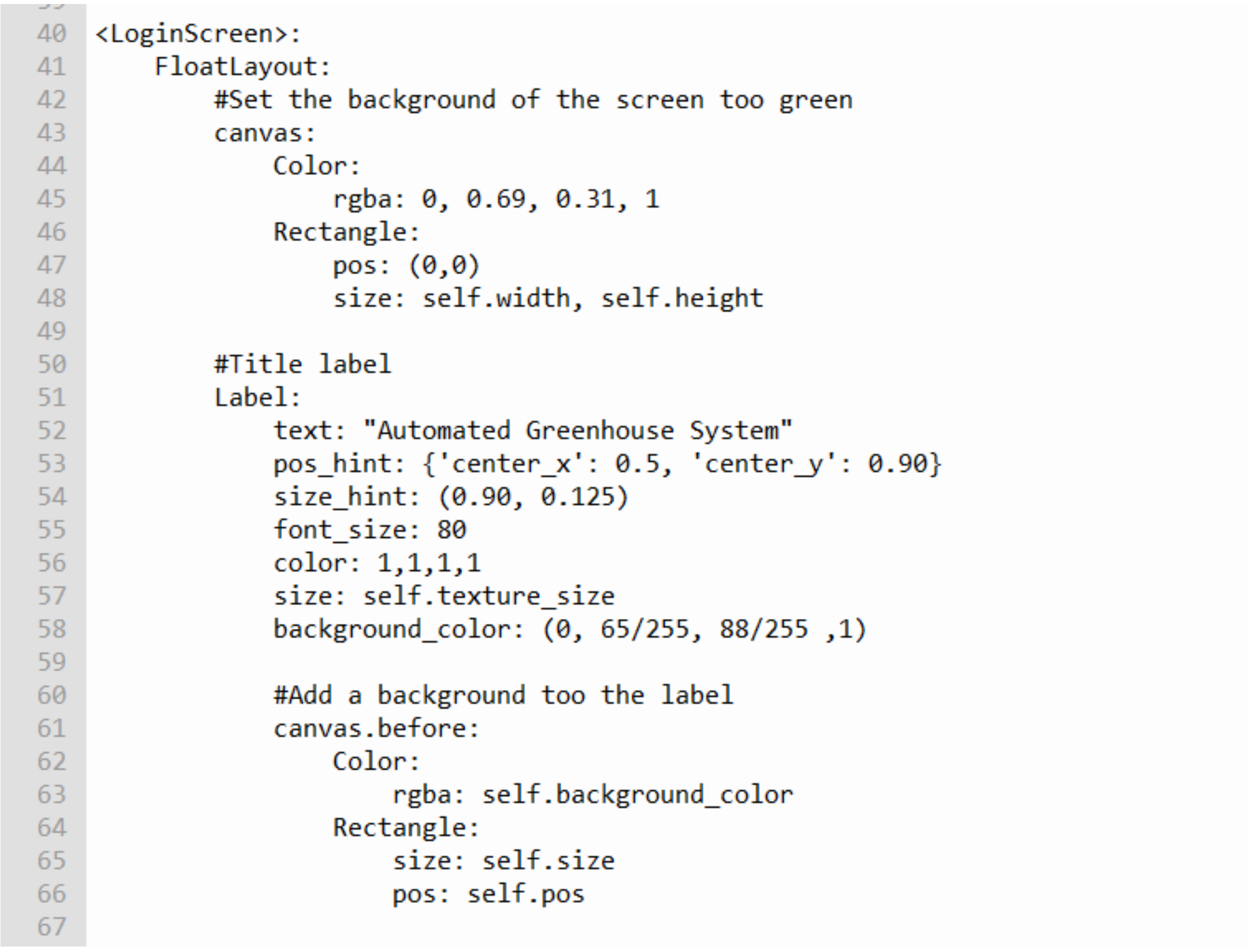
Username: JohnDoe

Password: GreenHouseMaster!123

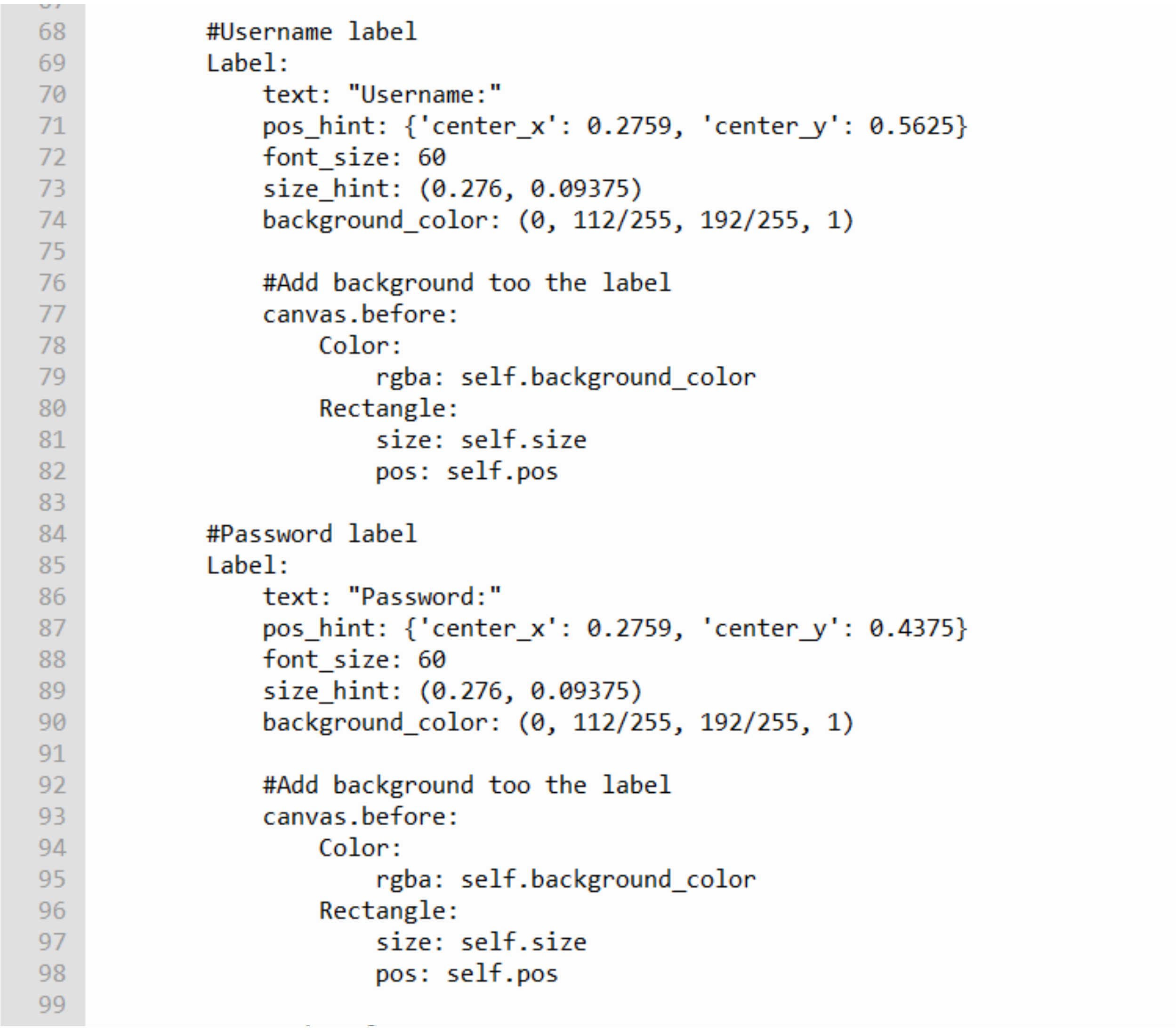
Mockup

Above are screenshots of the kivy login screen and the gui mockup for comparison.

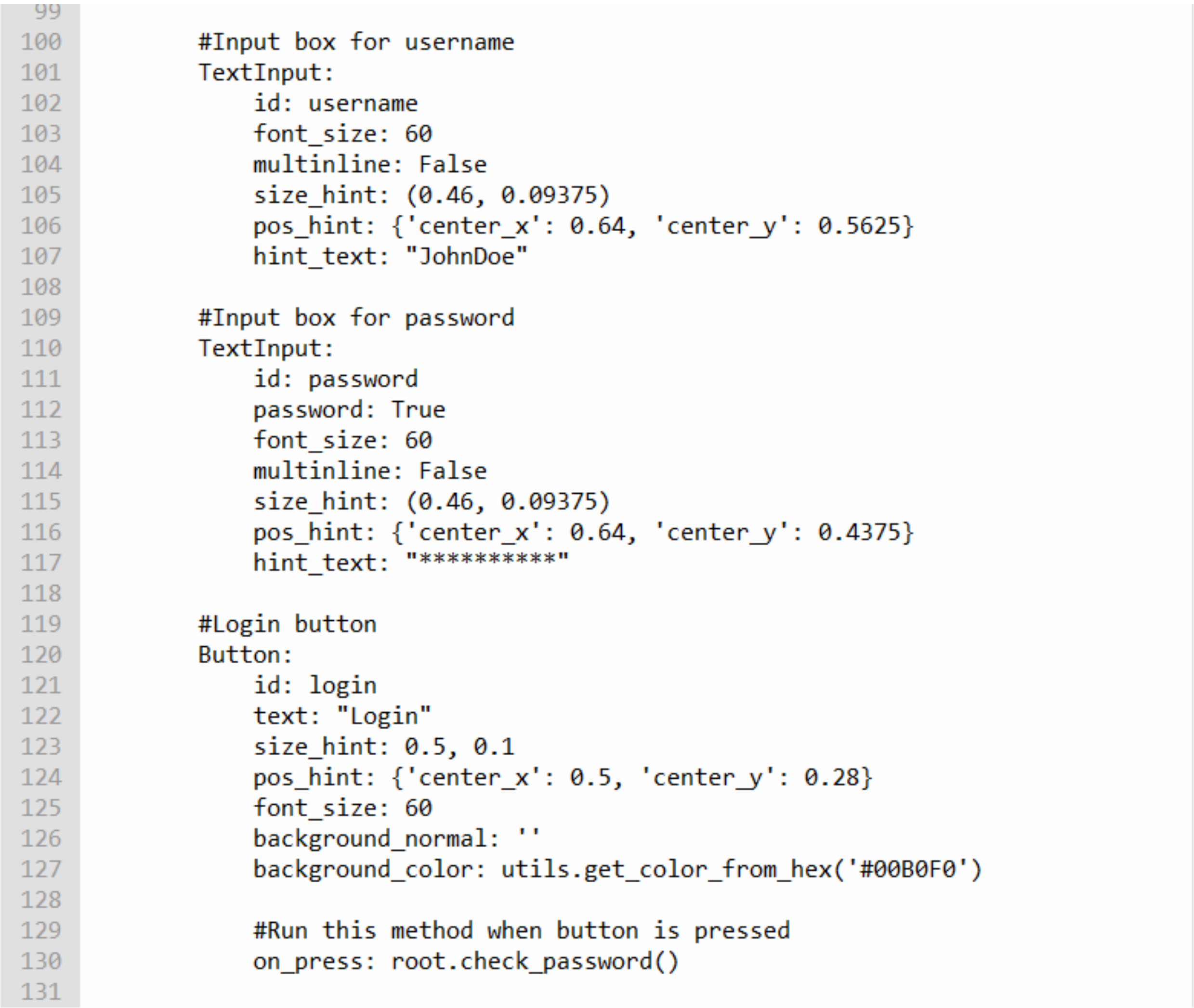
Complete login screen code



```
<LoginScreen>:
    FloatLayout:
        #Set the background of the screen too green
        canvas:
            Color:
                rgba: 0, 0.69, 0.31, 1
            Rectangle:
                pos: (0,0)
                size: self.width, self.height

        #Title label
        Label:
            text: "Automated Greenhouse System"
            pos_hint: {'center_x': 0.5, 'center_y': 0.90}
            size_hint: (0.90, 0.125)
            font_size: 80
            color: 1,1,1,1
            size: self.texture_size
            background_color: (0, 65/255, 88/255 ,1)

            #Add a background too the label
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

```



```
        #Username label
        Label:
            text: "Username:"
            pos_hint: {'center_x': 0.2759, 'center_y': 0.5625}
            font_size: 60
            size_hint: (0.276, 0.09375)
            background_color: (0, 112/255, 192/255, 1)

            #Add background too the label
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Password label
        Label:
            text: "Password:"
            pos_hint: {'center_x': 0.2759, 'center_y': 0.4375}
            font_size: 60
            size_hint: (0.276, 0.09375)
            background_color: (0, 112/255, 192/255, 1)

            #Add background too the label
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

```



```

        #Input box for username
        TextInput:
            id: username
            font_size: 60
            multinline: False
            size_hint: (0.46, 0.09375)
            pos_hint: {'center_x': 0.64, 'center_y': 0.5625}
            hint_text: "JohnDoe"

        #Input box for password
        TextInput:
            id: password
            password: True
            font_size: 60
            multinline: False
            size_hint: (0.46, 0.09375)
            pos_hint: {'center_x': 0.64, 'center_y': 0.4375}
            hint_text: "**********"

        #Login button
        Button:
            id: login
            text: "Login"
            size_hint: 0.5, 0.1
            pos_hint: {'center_x': 0.5, 'center_y': 0.28}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #Run this method when button is pressed
            on_press: root.check_password()

```

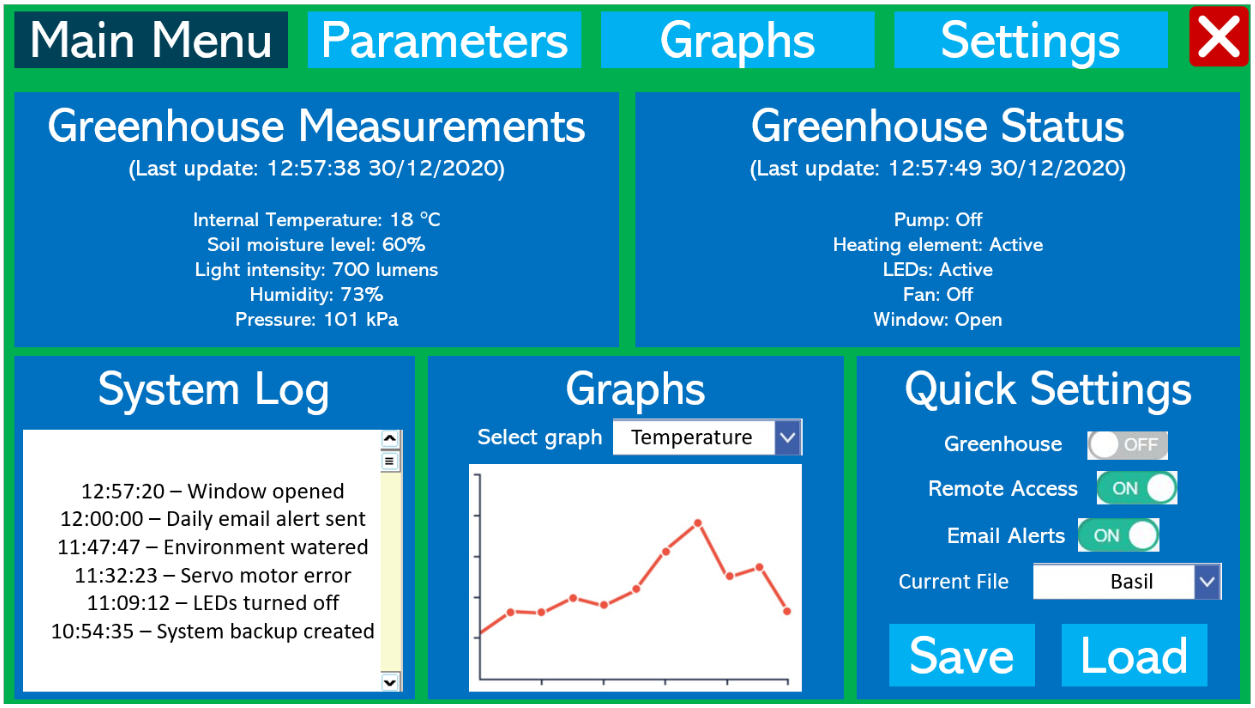
Test Plan – Login Screen

Test Number	**Test Plan**	**Expected Outcome**	**Actual Outcome**	**Pass/Fail**
1	Enter some text into the username text input box	The text will appear in the text box	The text appeared in the text box	Pass
2	Try to enter a new line in the username text input box	No new line should be entered, and the text should stay on one line	The text was limited to one line	Pass
3	Enter some text into the password input box	The text should be entered but hidden by the “*” character	The text was entered and hidden by the “*” symbol	Pass
4	Try to enter a new line in the	No new line should be	The text was limited to one line	Pass

	password text input box	entered, and the text should stay on one line		

Development Log – Main Menu

The main menu screen is designed to give the user a quick overview of the greenhouse system showing them the main measurements coming from the greenhouse and letting them make some quick adjustments to the settings. The menu is split into 5 compartments each representing a different group of related features or information. This is the first screen that will make use of a scroll view for the system log and of dropdown menus.



Greenhouse Measurements

(Last update: 12:57:38 30/12/2020)

Internal Temperature: 18 °C
Soil moisture level: 60%
Light intensity: 700 lumens
Humidity: 73%
Pressure: 101 kPa

Greenhouse Status

(Last update: 12:57:49 30/12/2020)

Pump: Off
Heating element: Active
LEDs: Active
Fan: Off
Window: Open

System Log

```
12:57:20 – Window opened
12:00:00 – Daily email alert sent
11:47:47 – Environment watered
11:32:23 – Servo motor error
11:09:12 – LEDs turned off
10:54:35 – System backup created
```

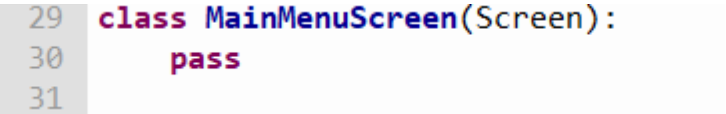
Graphs

Select graph: Temperature

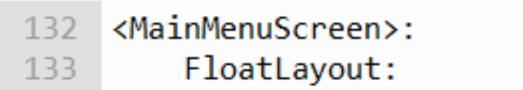
Quick Settings

Greenhouse: OFF
Remote Access: ON
Email Alerts: ON
Current File: Basil

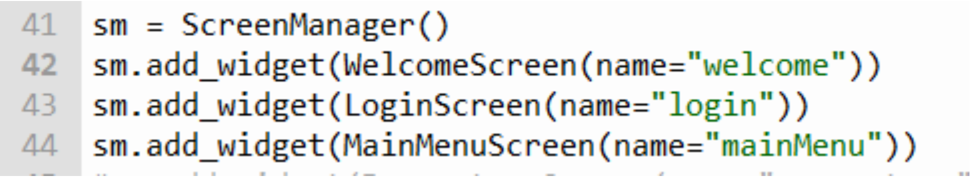
This is the mockup of the main menu. I will be leaving the graph blank as I will need to implement this later.



```python
class MainMenuScreen(Screen):
    pass

```

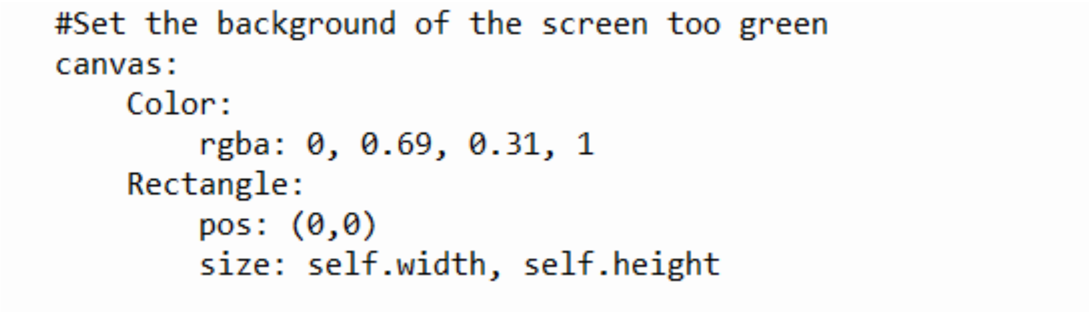


```
<MainMenuScreen>:
    FloatLayout:
```



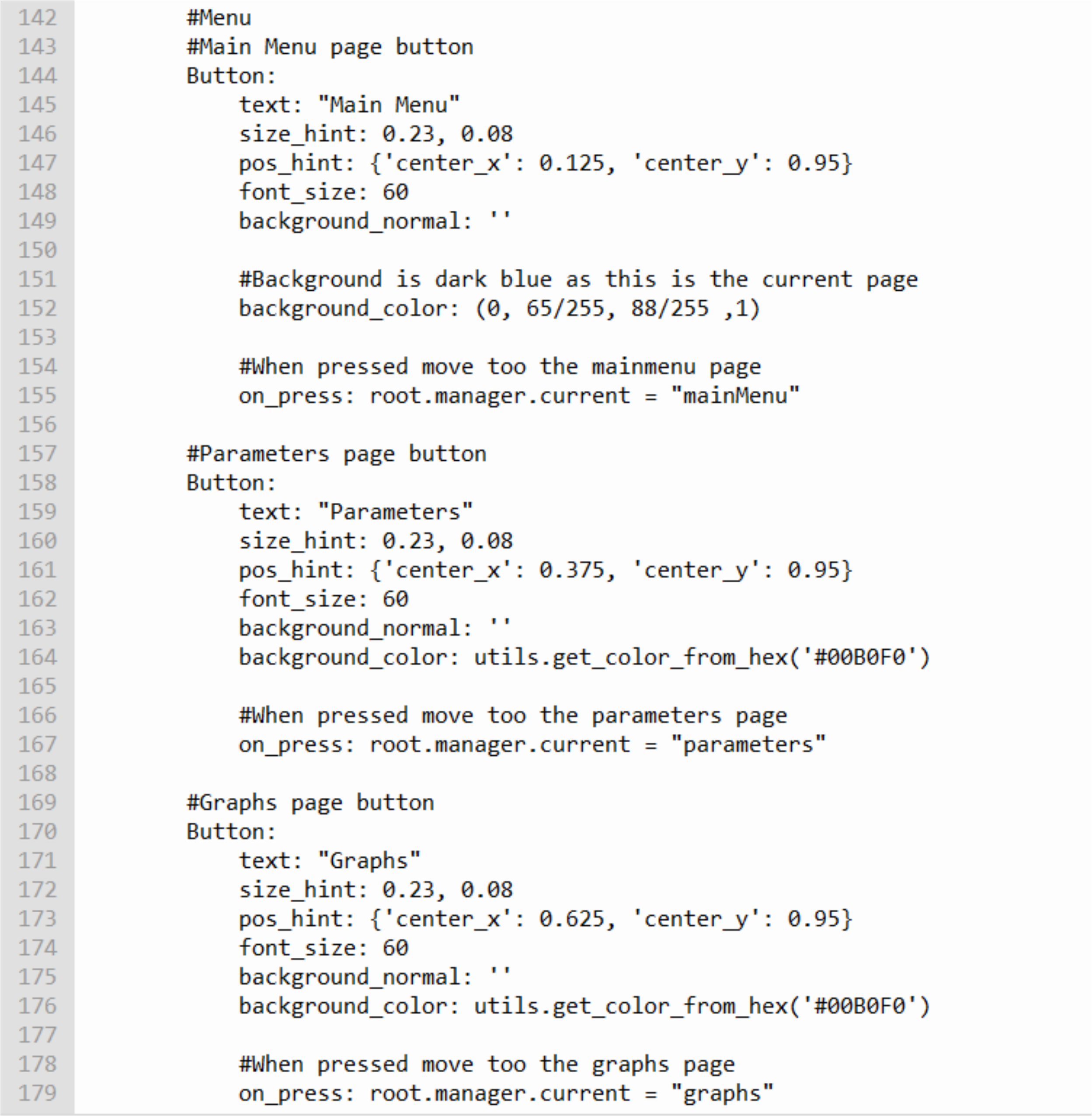
```python
sm = ScreenManager()
sm.add_widget(WelcomeScreen(name="welcome"))
sm.add_widget(LoginScreen(name="login"))
sm.add_widget(MainMenuScreen(name="mainMenu"))
```

The main menu screen class will later have functions for all the buttons, dropdown menus and the scroll view but for the time being is left empty. Inside the Kivy file I have also added a new screen with a matching name. Whilst also adding the new main menu screen too the screen manager.

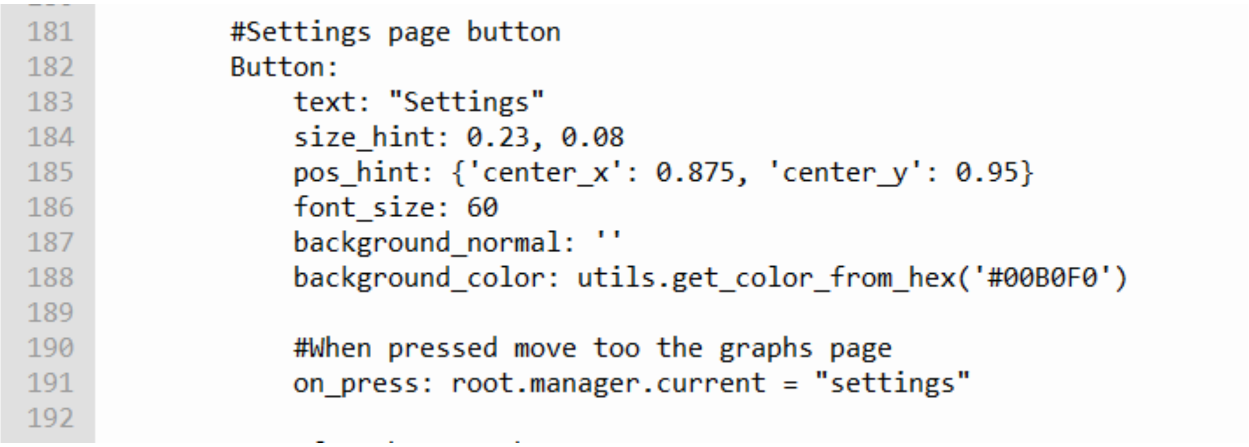


```
#Set the background of the screen too green
canvas:
    Color:
        rgba: 0, 0.69, 0.31, 1
    Rectangle:
        pos: (0,0)
        size: self.width, self.height
```

Here the background is set to the colour green.

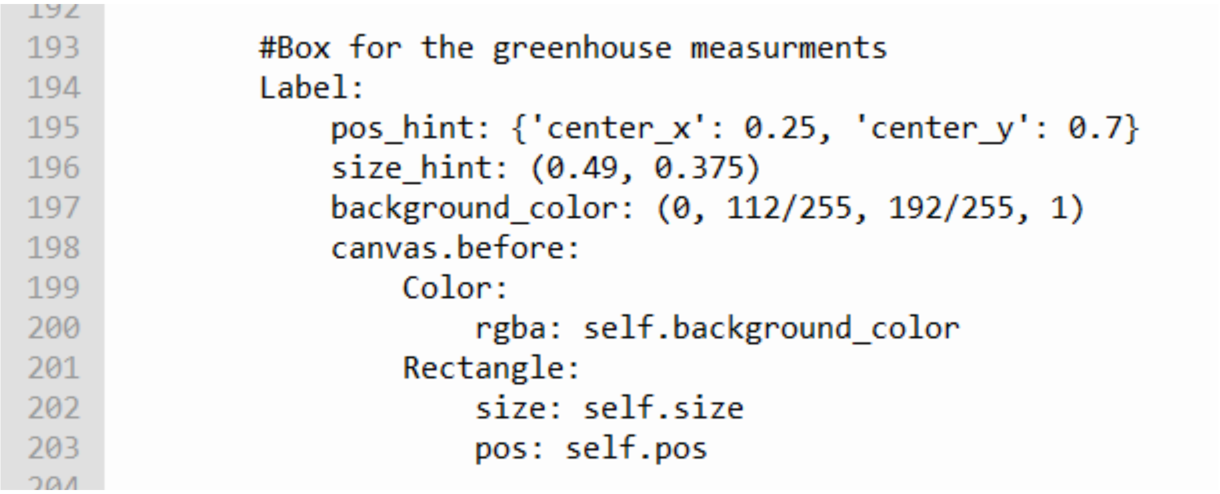


```
        #Menu
        #Main Menu page button
        Button:
            text: "Main Menu"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.125, 'center_y': 0.95}
            font_size: 60
            background_normal: ''

            #Background is dark blue as this is the current page
            background_color: (0, 65/255, 88/255 ,1)

            #When pressed move too the mainmenu page
            on_press: root.manager.current = "mainMenu"

        #Parameters page button
        Button:
            text: "Parameters"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.375, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #When pressed move too the parameters page
            on_press: root.manager.current = "parameters"

        #Graphs page button
        Button:
            text: "Graphs"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.625, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #When pressed move too the graphs page
            on_press: root.manager.current = "graphs"
```



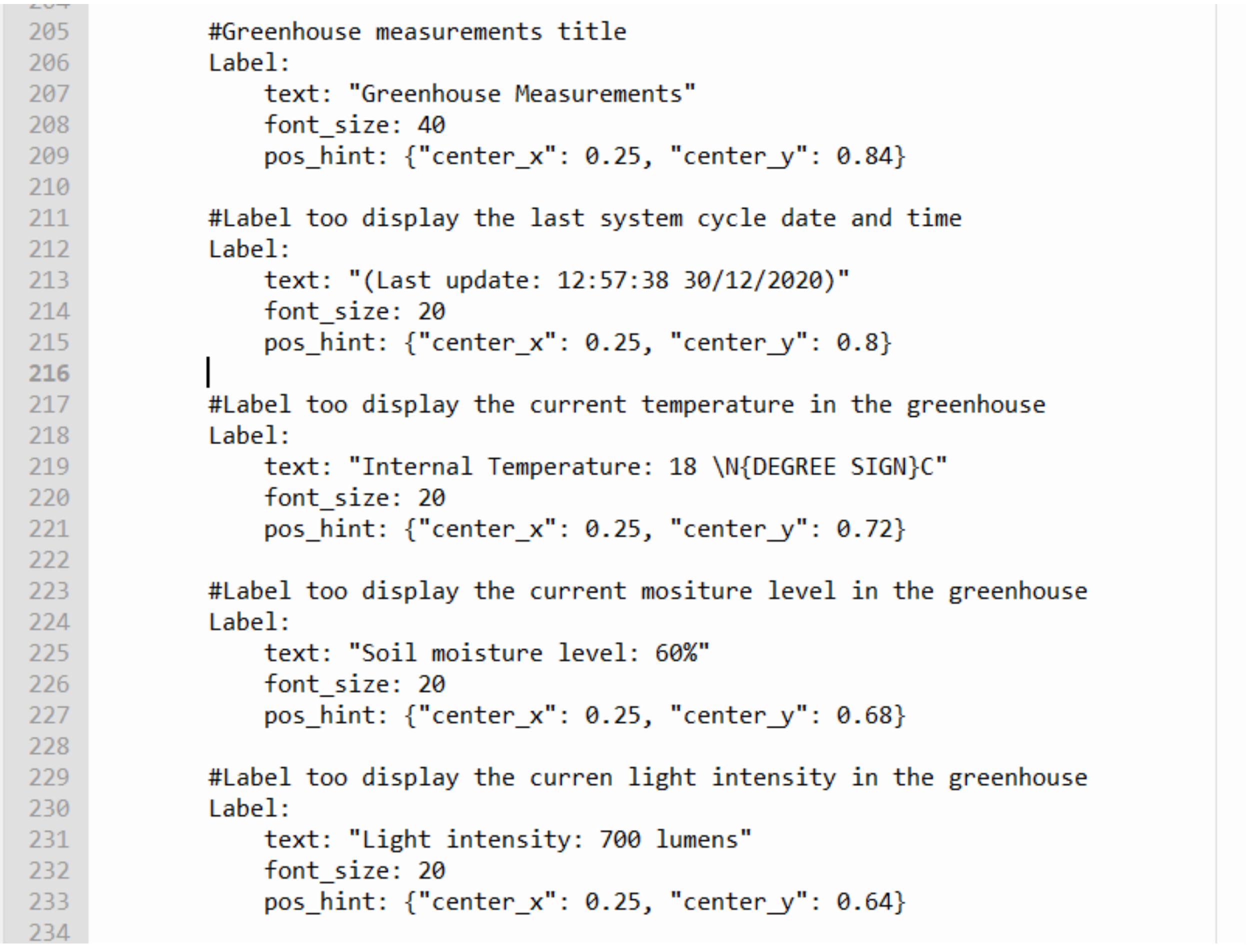
```
        #Settings page button
        Button:
            text: "Settings"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.875, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #When pressed move too the graphs page
            on_press: root.manager.current = "settings"
```

All the pages after the user have logged in feature a menu along the top. This menu will allow the user to navigate between the following the main menu, parameters, graphs, and settings screen. The current page that the user is on will be displayed as dark blue whilst the other pages buttons will be light blue. Once a button in the menu has been pressed the user will be taken to the related screen. The menu code is repeated at the top of each page that includes the menu with the only alteration being the page which is displayed as dark blue to represent the current page.

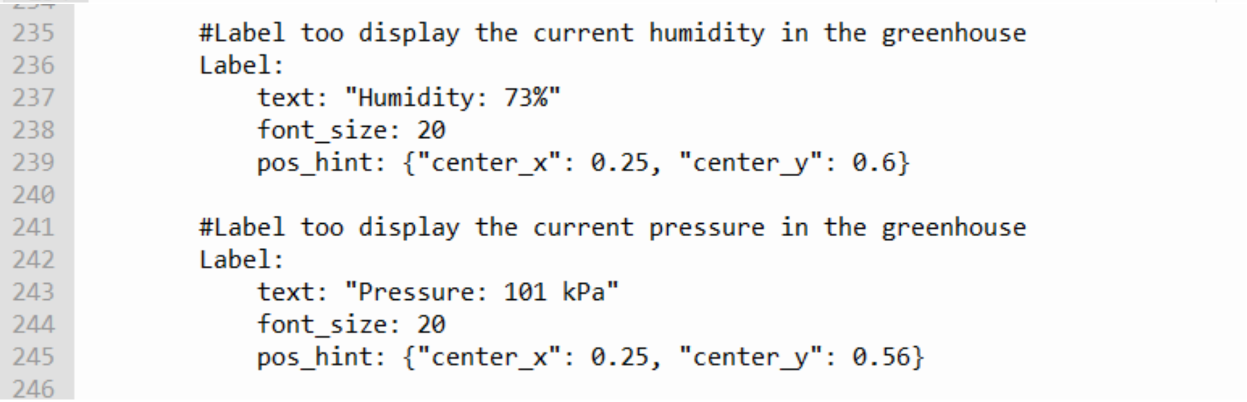


```
        #Box for the greenhouse measurments
        Label:
            pos_hint: {'center_x': 0.25, 'center_y': 0.7}
            size_hint: (0.49, 0.375)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos
```

Behind the greenhouse measurements I have drawn a box too compartmentalize the information regarding the greenhouse readings from the other collections on the main menu. I found that the easiest way to draw this box was too just create a label with no text property and then set the size and position as normal.

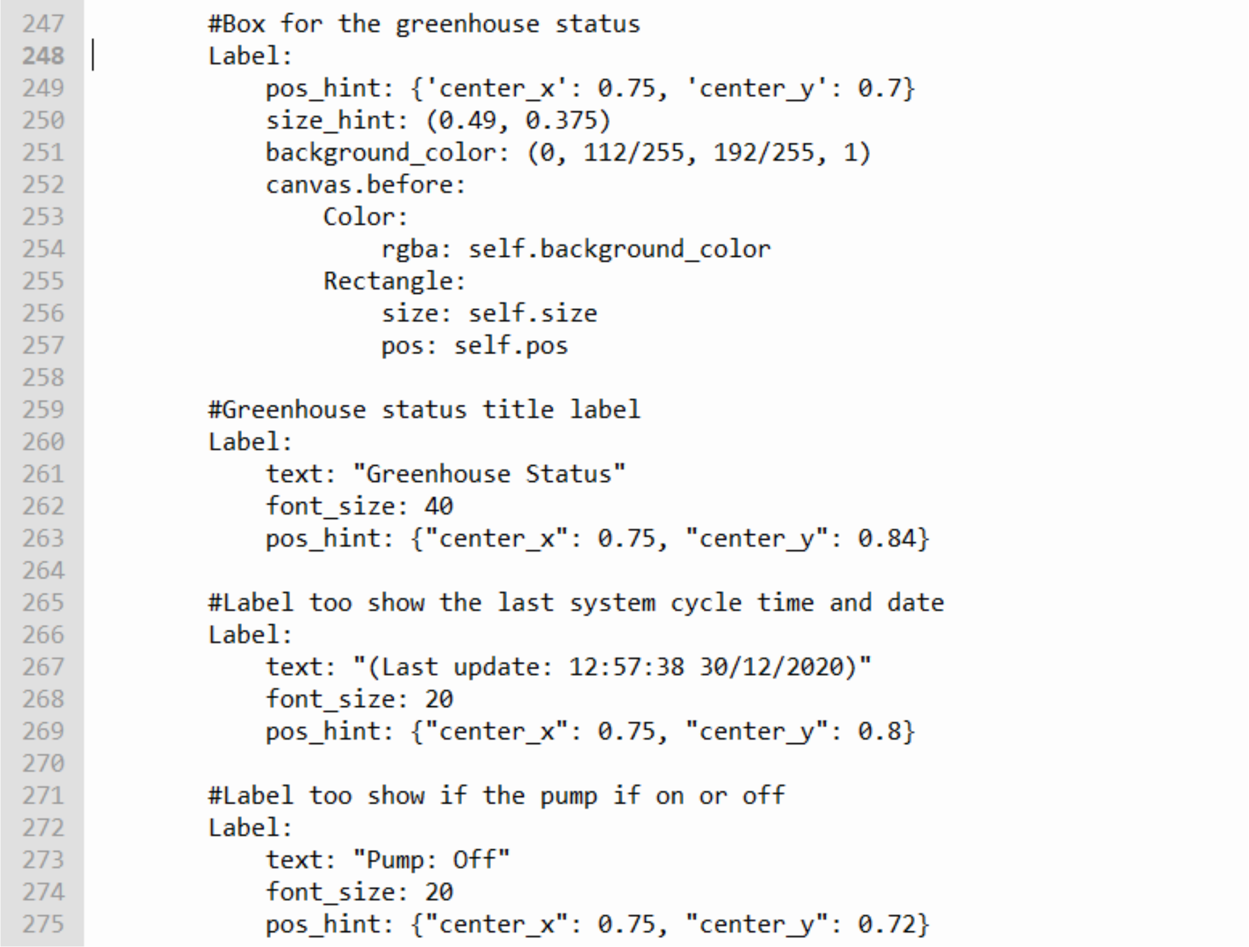


```
        #Greenhouse measurements title
        Label:
            text: "Greenhouse Measurements"
            font_size: 40
            pos_hint: {"center_x": 0.25, "center_y": 0.84}

        #Label too display the last system cycle date and time
        Label:
            text: "(Last update: 12:57:38 30/12/2020)"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.8}

        #Label too display the current temperature in the greenhouse
        Label:
            text: "Internal Temperature: 18 \N{DEGREE SIGN}C"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.72}

        #Label too display the current mositure level in the greenhouse
        Label:
            text: "Soil moisture level: 60%"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.68}

        #Label too display the curren light intensity in the greenhouse
        Label:
            text: "Light intensity: 700 lumens"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.64}

```

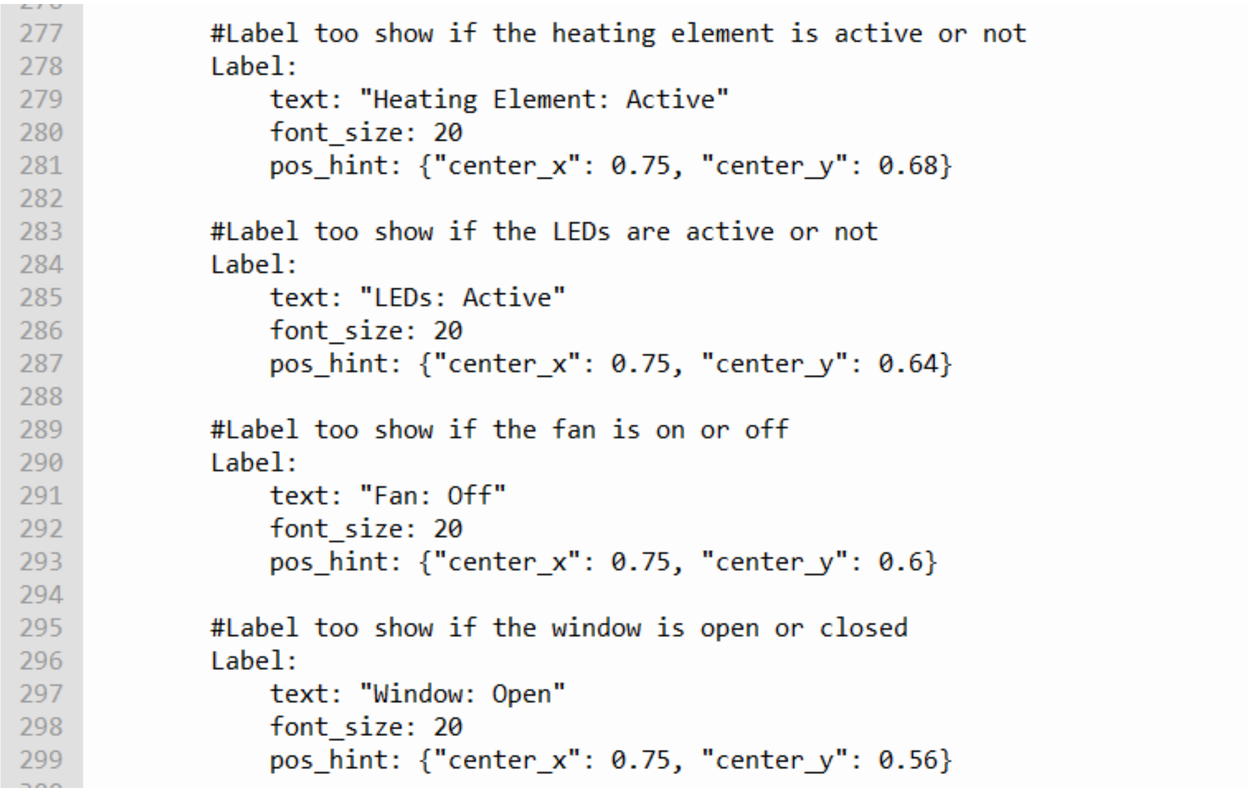


```
        #Label too display the current humidity in the greenhouse
        Label:
            text: "Humidity: 73%"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.6}

        #Label too display the current pressure in the greenhouse
        Label:
            text: "Pressure: 101 kPa"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.56}

```

Too complete the greenhouse measurements section of the main menu I have added a title showing the user what this section is related too and added labels with static text showing the user the readings from the greenhouse. The text in these measurement labels will be made dynamic later in development when their values will be continually updated to match the greenhouse readings.

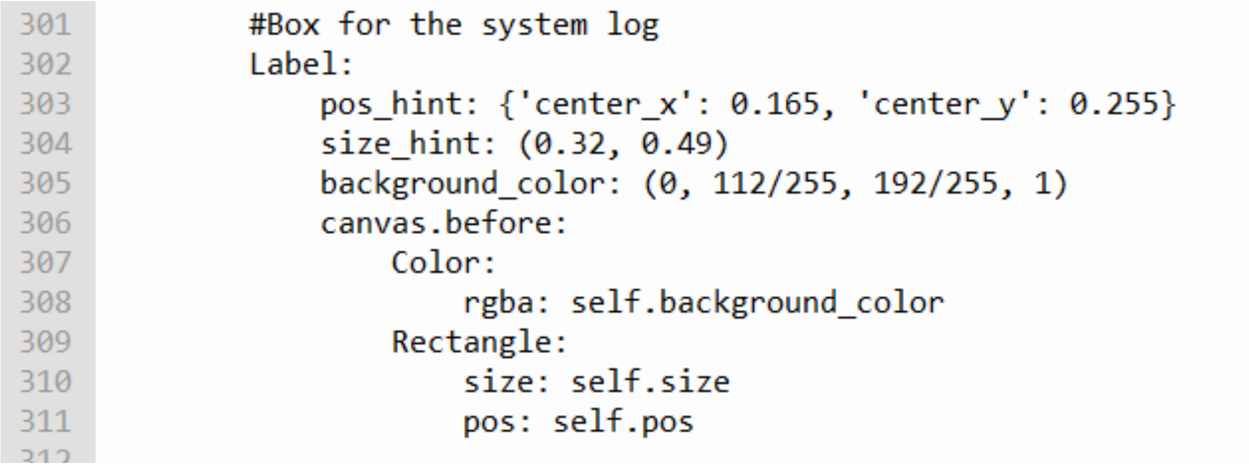


```
        #Box for the greenhouse status
        Label:
            pos_hint: {'center_x': 0.75, 'center_y': 0.7}
            size_hint: (0.49, 0.375)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Greenhouse status title label
        Label:
            text: "Greenhouse Status"
            font_size: 40
            pos_hint: {"center_x": 0.75, "center_y": 0.84}

        #Label too show the last system cycle time and date
        Label:
            text: "(Last update: 12:57:38 30/12/2020)"
            font_size: 20
            pos_hint: {"center_x": 0.75, "center_y": 0.8}

        #Label too show if the pump if on or off
        Label:
            text: "Pump: Off"
            font_size: 20
            pos_hint: {"center_x": 0.75, "center_y": 0.72}
```



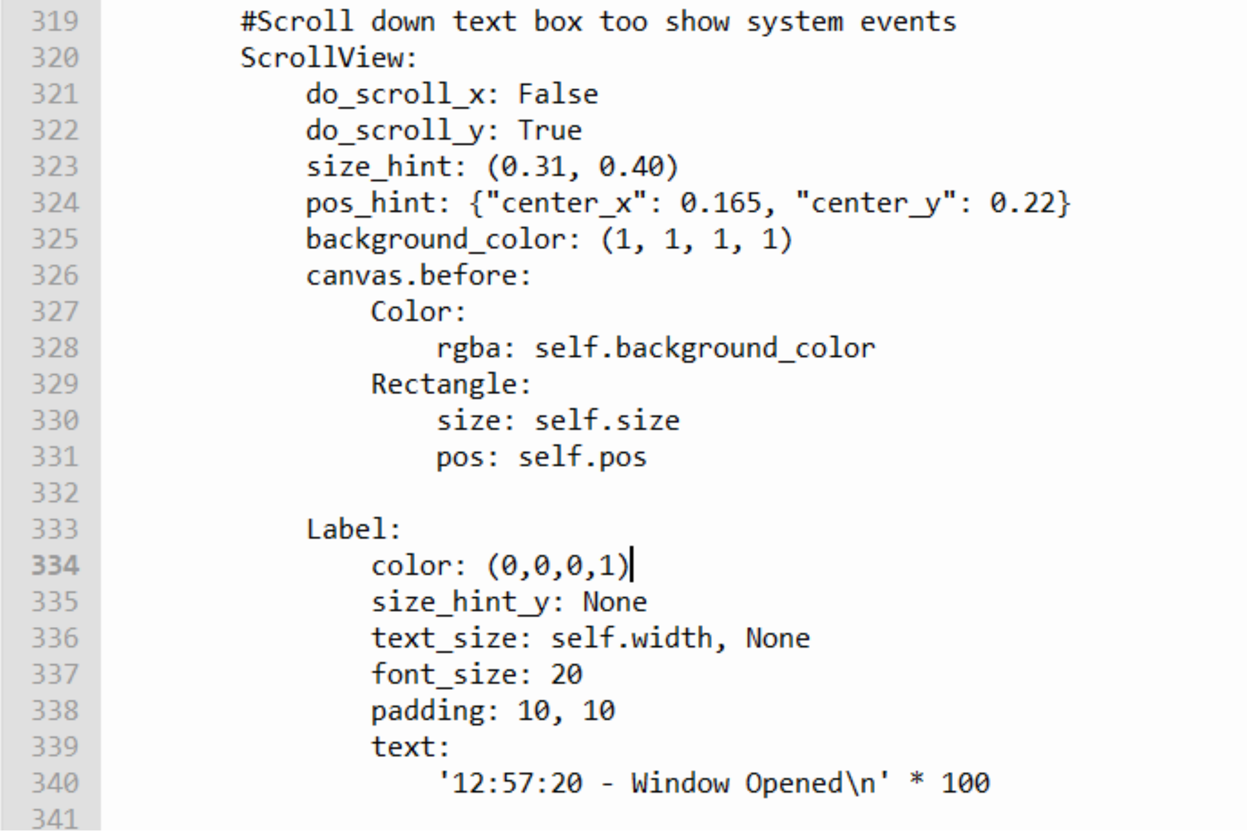
```
        #Label too show if the heating element is active or not
        Label:
            text: "Heating Element: Active"
            font_size: 20
            pos_hint: {"center_x": 0.75, "center_y": 0.68}

        #Label too show if the LEDs are active or not
        Label:
            text: "LEDs: Active"
            font_size: 20
            pos_hint: {"center_x": 0.75, "center_y": 0.64}

        #Label too show if the fan is on or off
        Label:
            text: "Fan: Off"
            font_size: 20
            pos_hint: {"center_x": 0.75, "center_y": 0.6}

        #Label too show if the window is open or closed
        Label:
            text: "Window: Open"
            font_size: 20
            pos_hint: {"center_x": 0.75, "center_y": 0.56}
```

The greenhouse status box follows the same structure as the measurements box. Once again, the values will become dynamic later when they will be updated to match the greenhouse components status.



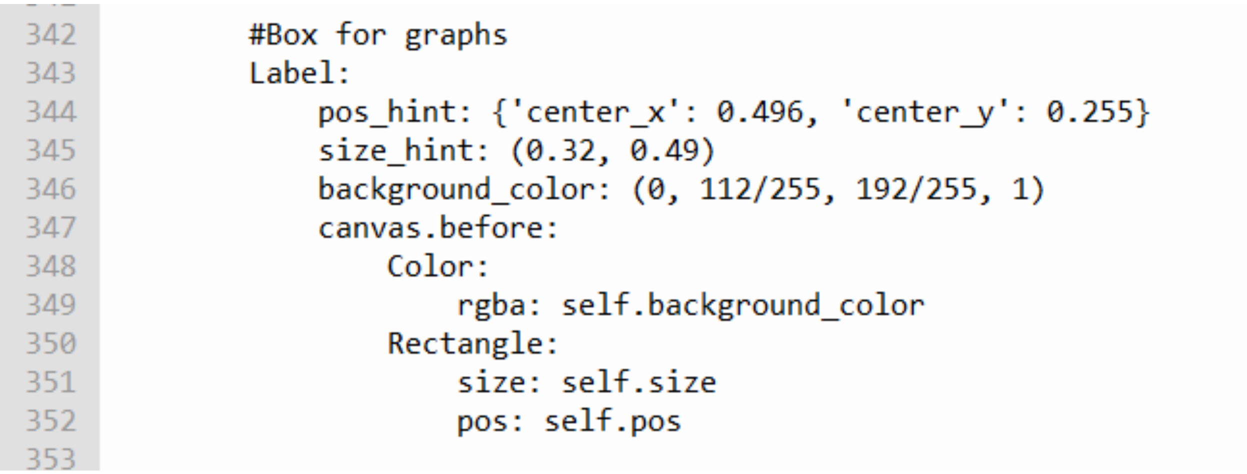
```
        #Box for the system log
        Label:
            pos_hint: {'center_x': 0.165, 'center_y': 0.255}
            size_hint: (0.32, 0.49)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos
```

Moving onto the system log box this will contain a log of the system events and the user will be able to scroll back in time to see previous events. Here I have added the background that sits behind the system log for aesthetics.



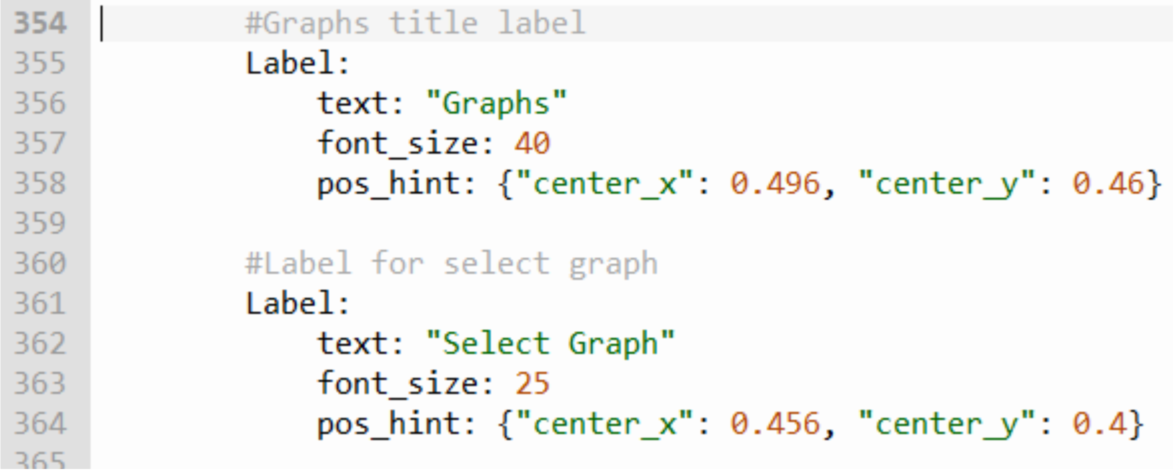
```
        #Scroll down text box too show system events
        ScrollView:
            do_scroll_x: False
            do_scroll_y: True
            size_hint: (0.31, 0.40)
            pos_hint: {"center_x": 0.165, "center_y": 0.22}
            background_color: (1, 1, 1, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

            Label:
                color: (0,0,0,1)
                size_hint_y: None
                text_size: self.width, None
                font_size: 20
                padding: 10, 10
                text:
                    '12:57:20 - Window Opened\n' * 100

```

A scrollview in kivy works by creating a label that is nested inside a scrollview. The scrollview has many properties which can control how it functions. For this scrollview I have selected that the user will be able to scroll up and down in the y direction but not in the x direction. Like will all elements in kivy I have set a size hint and a position hint too position the element on the page. I have also added a white background in the same way I did for our labels so that the text shows us better as opposed to the blue background of the system log box. Nested inside the scrollview is a label which has an unlimited size in the y direction which means its size will exceed the size of the parent scrollview. Since the size is larger than the parent the scroll will kick in allowing the user too to navigate the text. Some padding is also added to the label so that the text is moved in slightly from the side of the scroll view. Currently the text is just a generic string of 100 lines for testing. Later, system events will be added to this label so that the user can scroll through them.



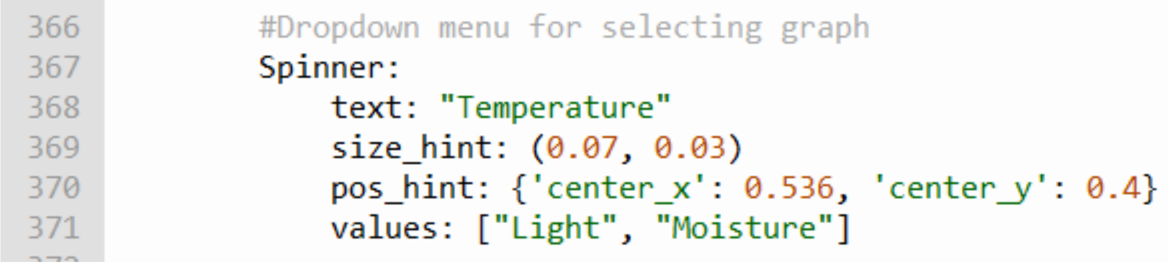
```
        #Box for graphs
        Label:
            pos_hint: {'center_x': 0.496, 'center_y': 0.255}
            size_hint: (0.32, 0.49)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

```

Above is the code that I have used to add a background box for the graphs section of the main menu.



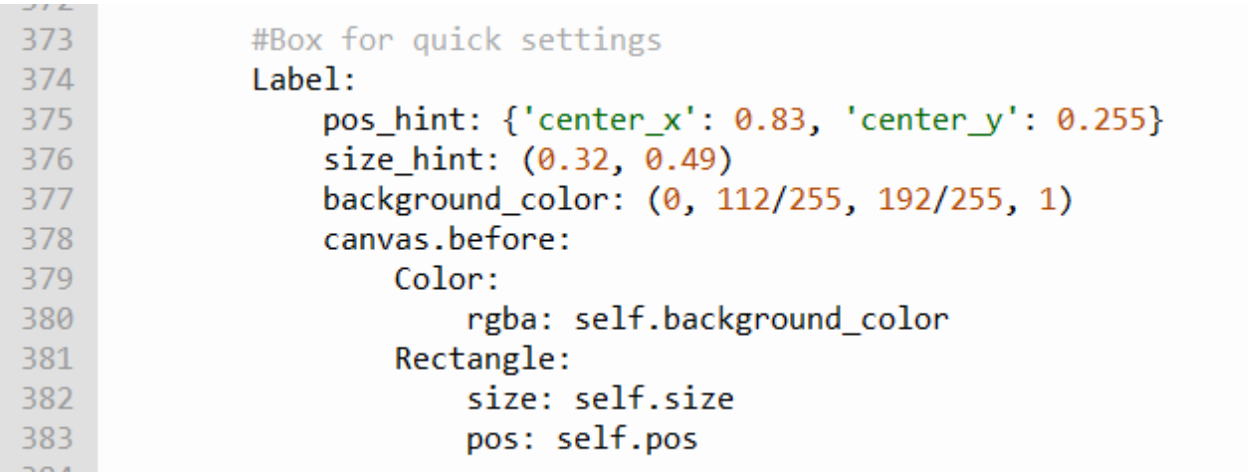
```
        #Graphs title label
        Label:
            text: "Graphs"
            font_size: 40
            pos_hint: {"center_x": 0.496, "center_y": 0.46}

        #Label for select graph
        Label:
            text: "Select Graph"
            font_size: 25
            pos_hint: {"center_x": 0.456, "center_y": 0.4}
```

Below the graphs background box is a title label showing the user what the following section is about. Along with a label that sits next to the dropdown box.



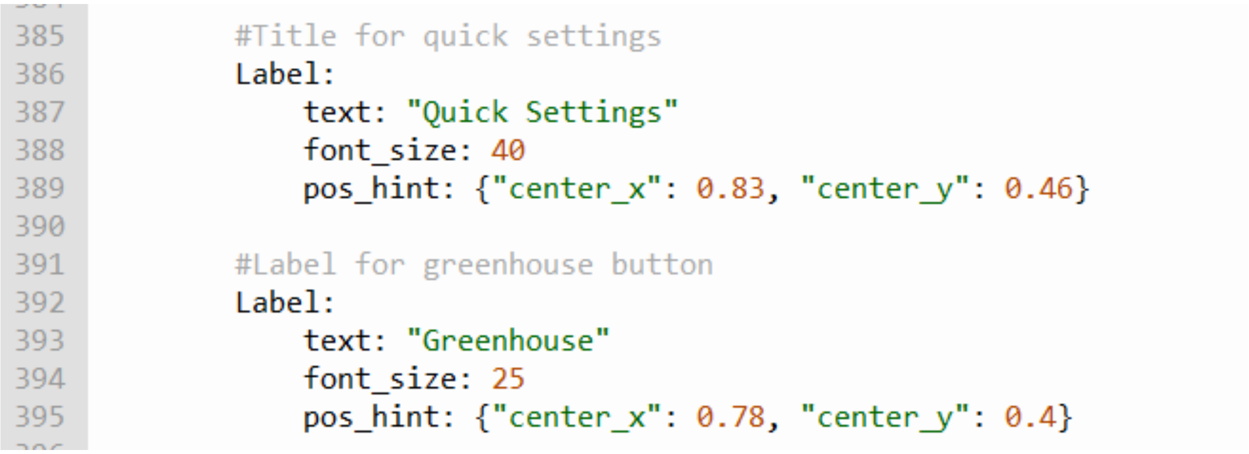
```
        #Dropdown menu for selecting graph
        Spinner:
            text: "Temperature"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.536, 'center_y': 0.4}
            values: ["Light", "Moisture"]
```

A dropdown menu in kivy is called a spinner. It takes a text parameter which in this case is set to “temperature” this is the default value of the dropdown menu before the user selects a different option from the dropdown. The dropdown menu can be positioned in just the same way as all other elements in kivy using the size hint and a position hint. The options available in the dropdown menu are listed in the values list of the spinner. Currently not all available values for the dropdown are in the list but later I will add them in.



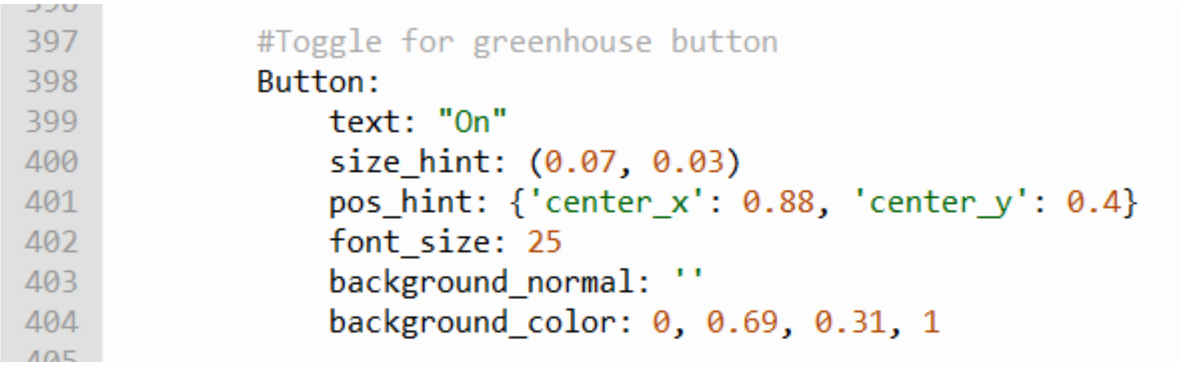
```
        #Box for quick settings
        Label:
            pos_hint: {'center_x': 0.83, 'center_y': 0.255}
            size_hint: (0.32, 0.49)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos
```

The final section of the main menu is the quick settings area where the user can adjust the settings of the greenhouse such as turning the greenhouse on and off. Here I have added a background box for the quick settings area in the bottom right of the screen.



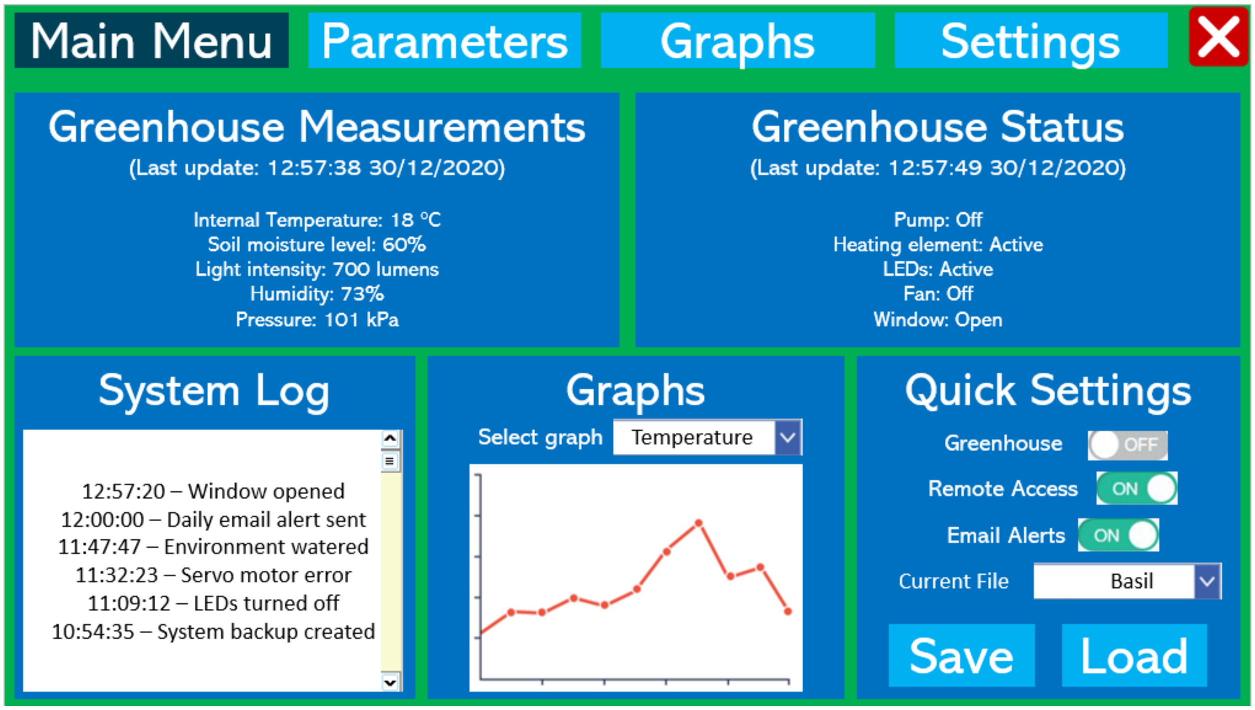
```
        #Title for quick settings
        Label:
            text: "Quick Settings"
            font_size: 40
            pos_hint: {"center_x": 0.83, "center_y": 0.46}

        #Label for greenhouse button
        Label:
            text: "Greenhouse"
            font_size: 25
            pos_hint: {"center_x": 0.78, "center_y": 0.4}
```

I have used two labels too add the title of the quick settings box and for the text next to the greenhouse on off toggle.



```
        #Toggle for greenhouse button
        Button:
            text: "On"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.88, 'center_y': 0.4}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1
```

There will be an on off toggle next to the different settings in the quick settings box. Whilst kivy does have a specific toggle element I found that it was not easy to implement. Instead for my own off toggle buttons I have decided to use a label. This label will be styled as having a green background with the text “on” when the toggle is on and then I will program the label so that once it is pressed the text swaps to red and the text to “off”.



Greenhouse Measurements
(Last update: 12:57:38 30/12/2020)

Internal Temperature: 18 °C
Soil moisture level: 60%
Light intensity: 700 lumens
Humidity: 73%
Pressure: 101 kPa

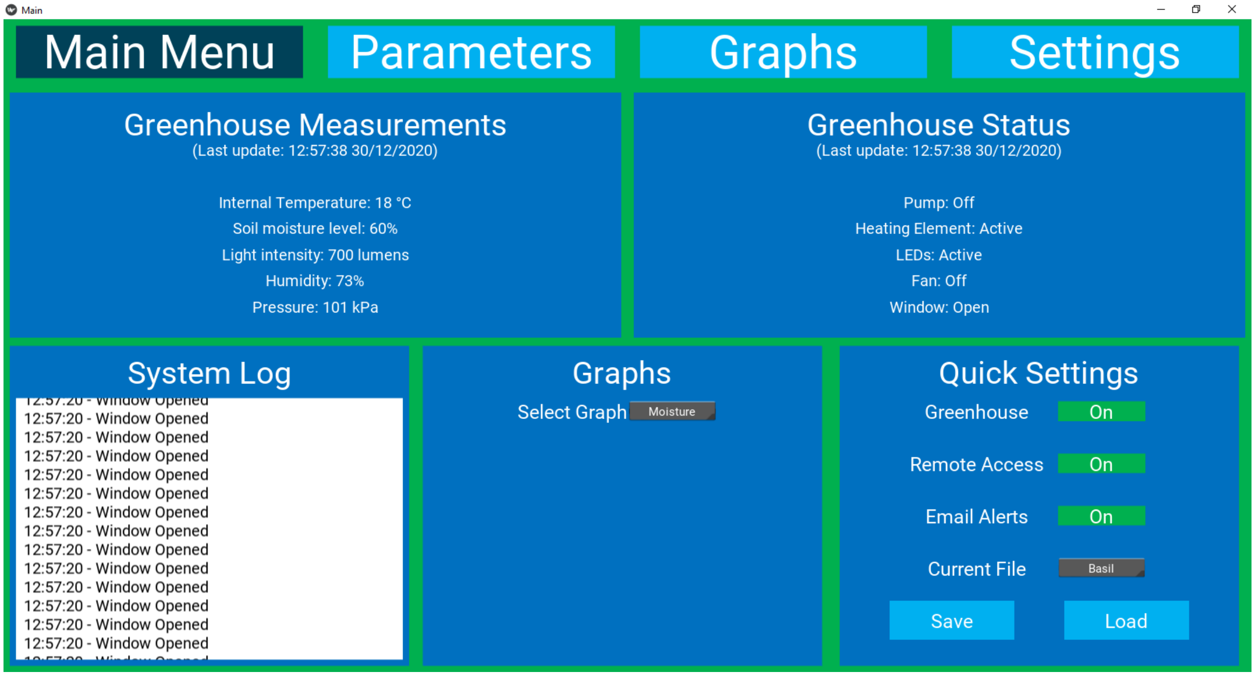
Greenhouse Status
(Last update: 12:57:49 30/12/2020)

Pump: Off
Heating element: Active
LEDs: Active
Fan: Off
Window: Open

System Log

```
12:57:20 – Window opened
12:00:00 – Daily email alert sent
11:47:47 – Environment watered
11:32:23 – Servo motor error
11:09:12 – LEDs turned off
10:54:35 – System backup created
```

Mockup



Greenhouse Measurements
(Last update: 12:57:38 30/12/2020)

Internal Temperature: 18 °C
Soil moisture level: 60%
Light intensity: 700 lumens
Humidity: 73%
Pressure: 101 kPa

Greenhouse Status
(Last update: 12:57:38 30/12/2020)

Pump: Off
Heating Element: Active
LEDs: Active
Fan: Off
Window: Open

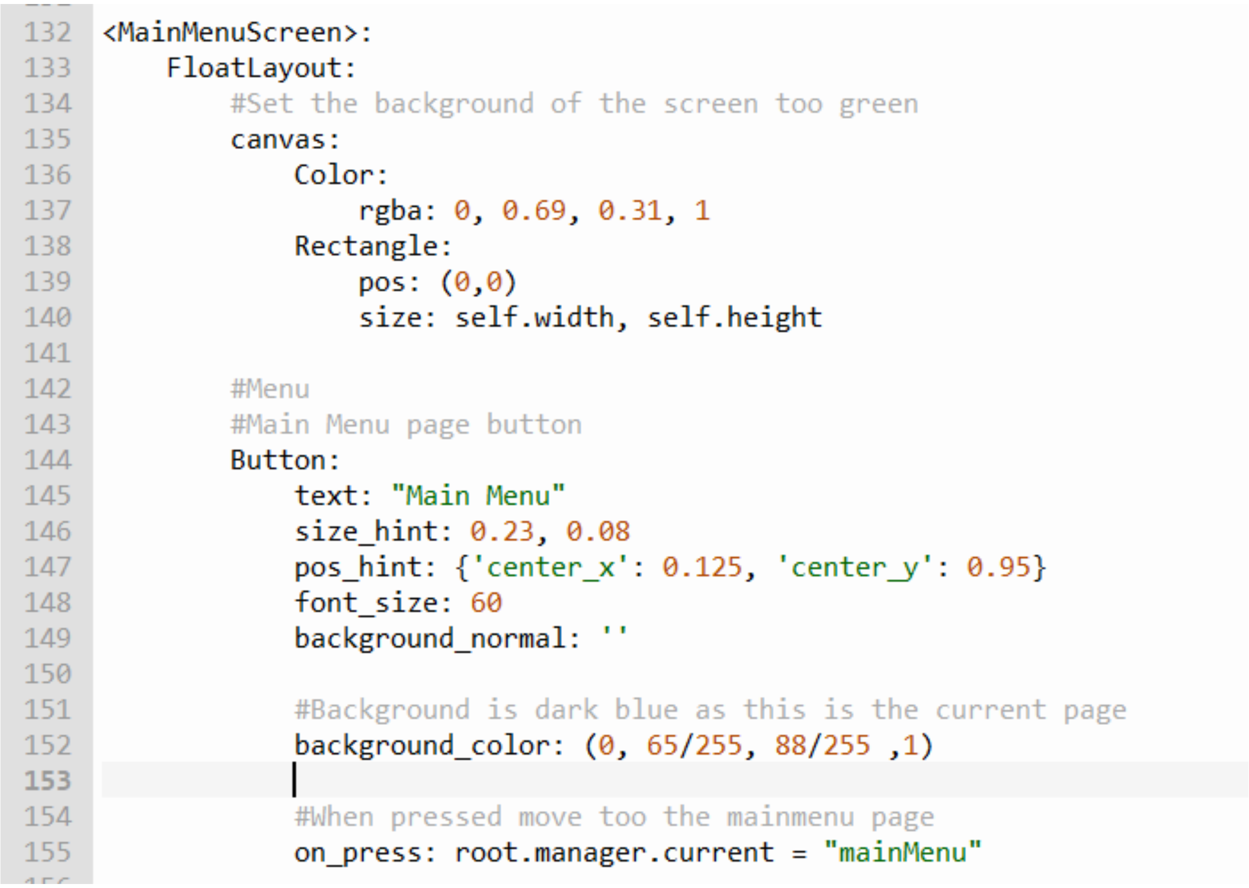
System Log

```
12:57:20 - Window Opened
12:57:20 - Window Opened
12:57:20 - Window Opened
12:57:20 - Window Opened
12:57:20 - Window Opened
12:57:20 - Window Opened
12:57:20 - Window Opened
12:57:20 - Window Opened
12:57:20 - Window Opened
12:57:20 - Window Opened
12:57:20 - Window Opened
12:57:20 - Window Opened
12:57:20 - Window Opened
12:57:20 - Window Opened
12:57:20 - Window Opened
```

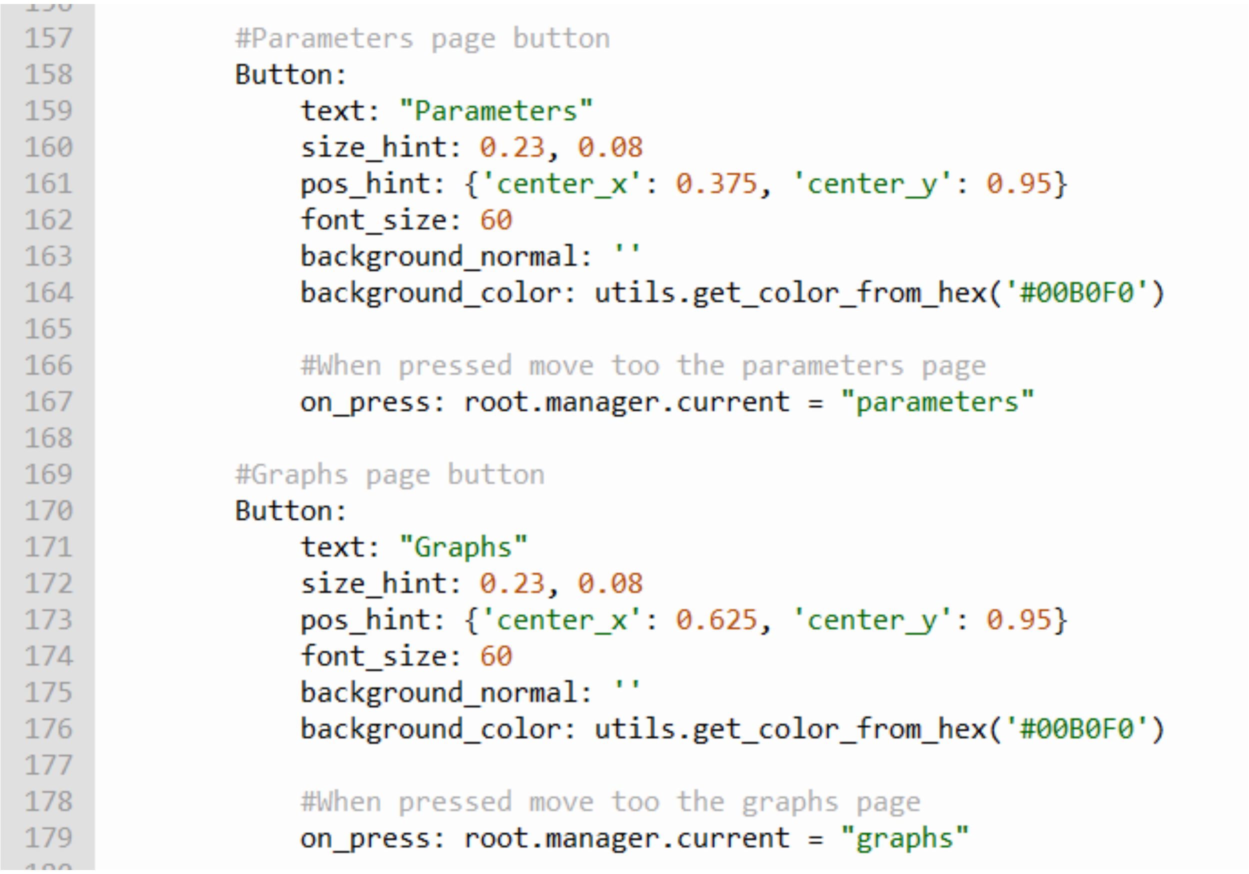
Kivy

Above is the comparison between the mockup and then kivy file final implementation of the main menu. The main differences can be seen in the styling of the drop-down menus, the on off toggle and the scroll view. However, the functions of each are identical. I have struggled with implementing the scroll view in kivy I have not been able to add a scroll bar next to the scroll view. When I come too programming the system log, I will attempt to find a solution to this however it is possible I will change it too a label without any scroll capabilities and instead append new system events to the text property of the label and remove old events from the start of the text property.

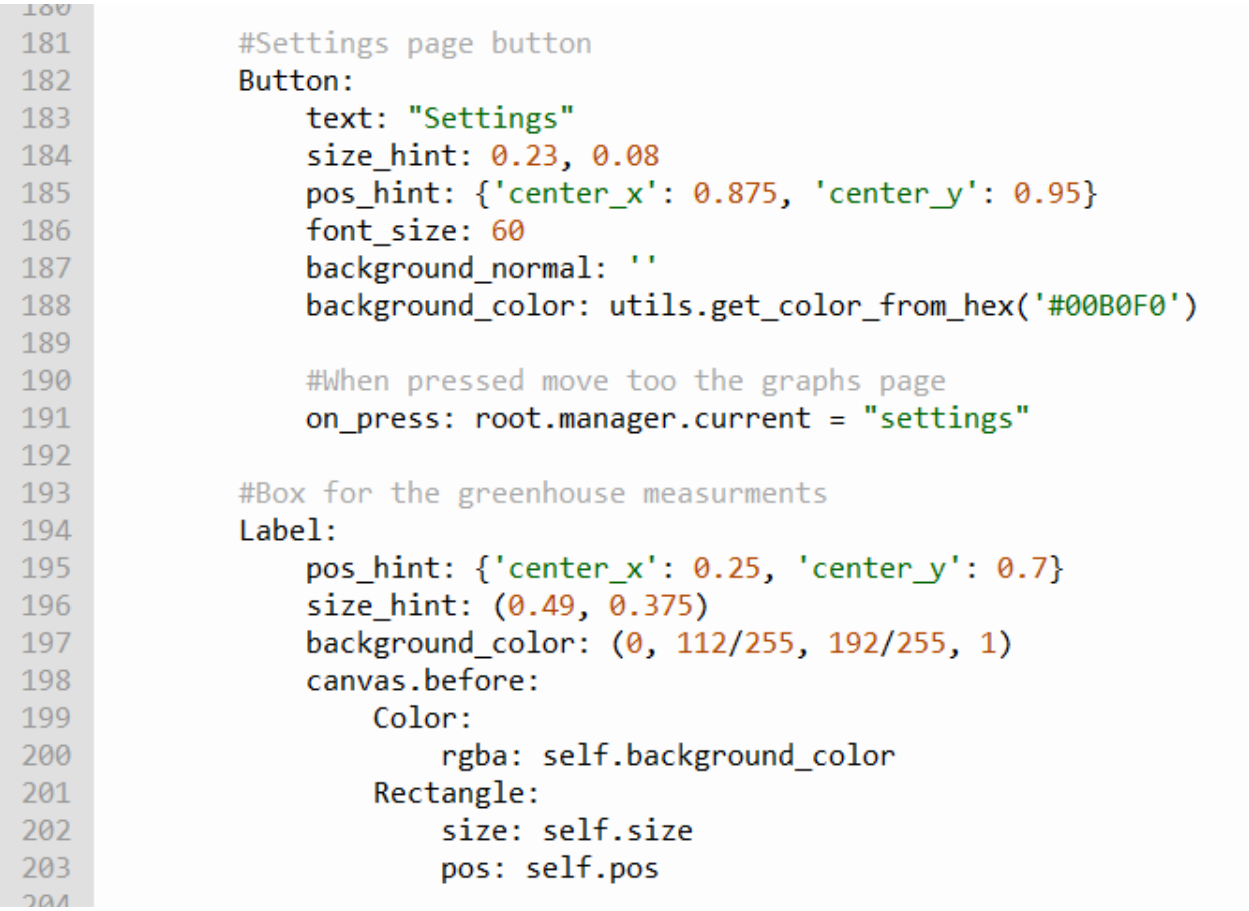
Complete main menu code



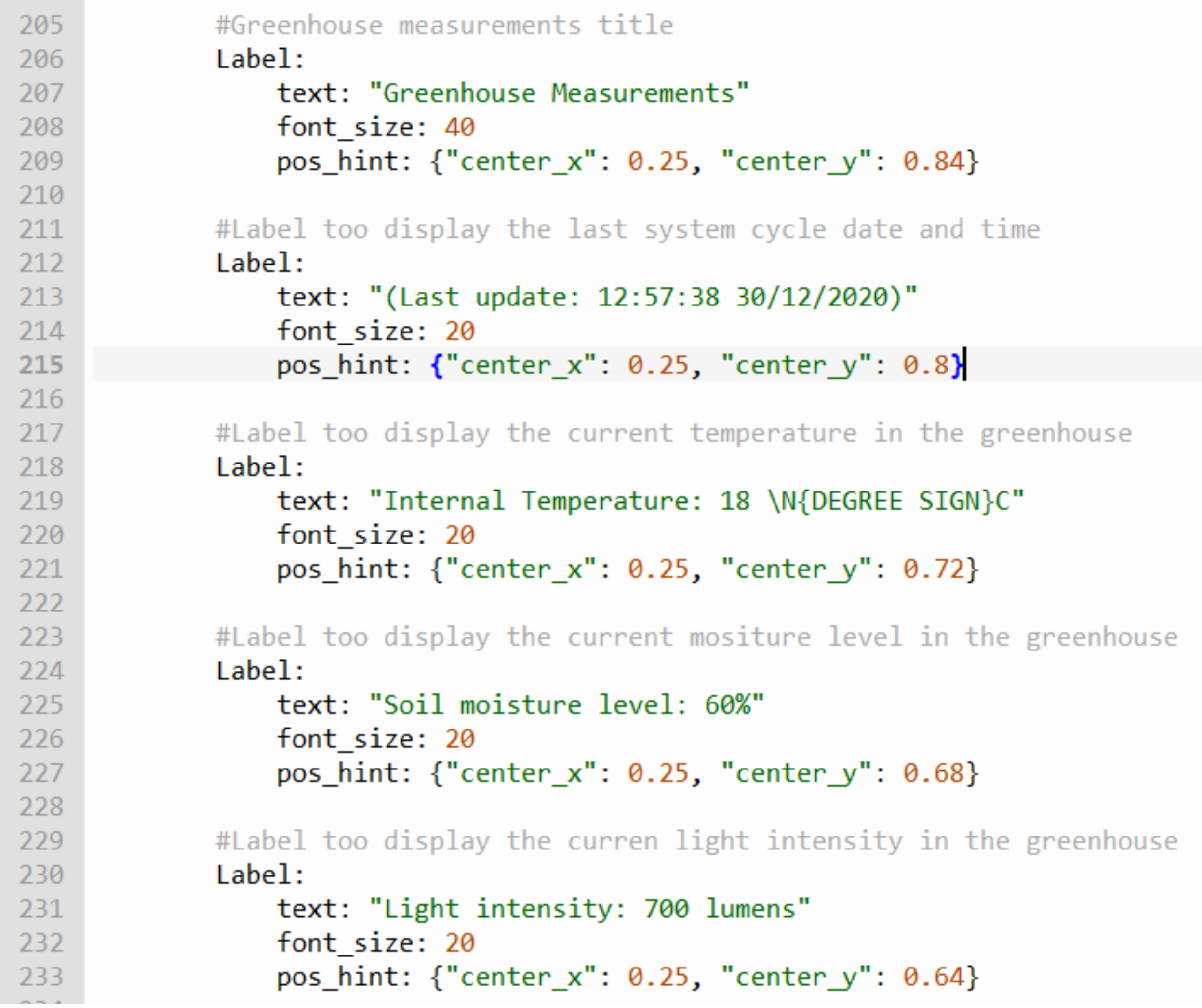
```
<MainMenuScreen>:
    FloatLayout:
        #Set the background of the screen too green
        canvas:
            Color:
                rgba: 0, 0.69, 0.31, 1
            Rectangle:
                pos: (0,0)
                size: self.width, self.height

        #Menu
        #Main Menu page button
        Button:
            text: "Main Menu"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.125, 'center_y': 0.95}
            font_size: 60
            background_normal: ''

            #Background is dark blue as this is the current page
            background_color: (0, 65/255, 88/255 ,1)

            #When pressed move too the mainmenu page
            on_press: root.manager.current = "mainMenu"
```



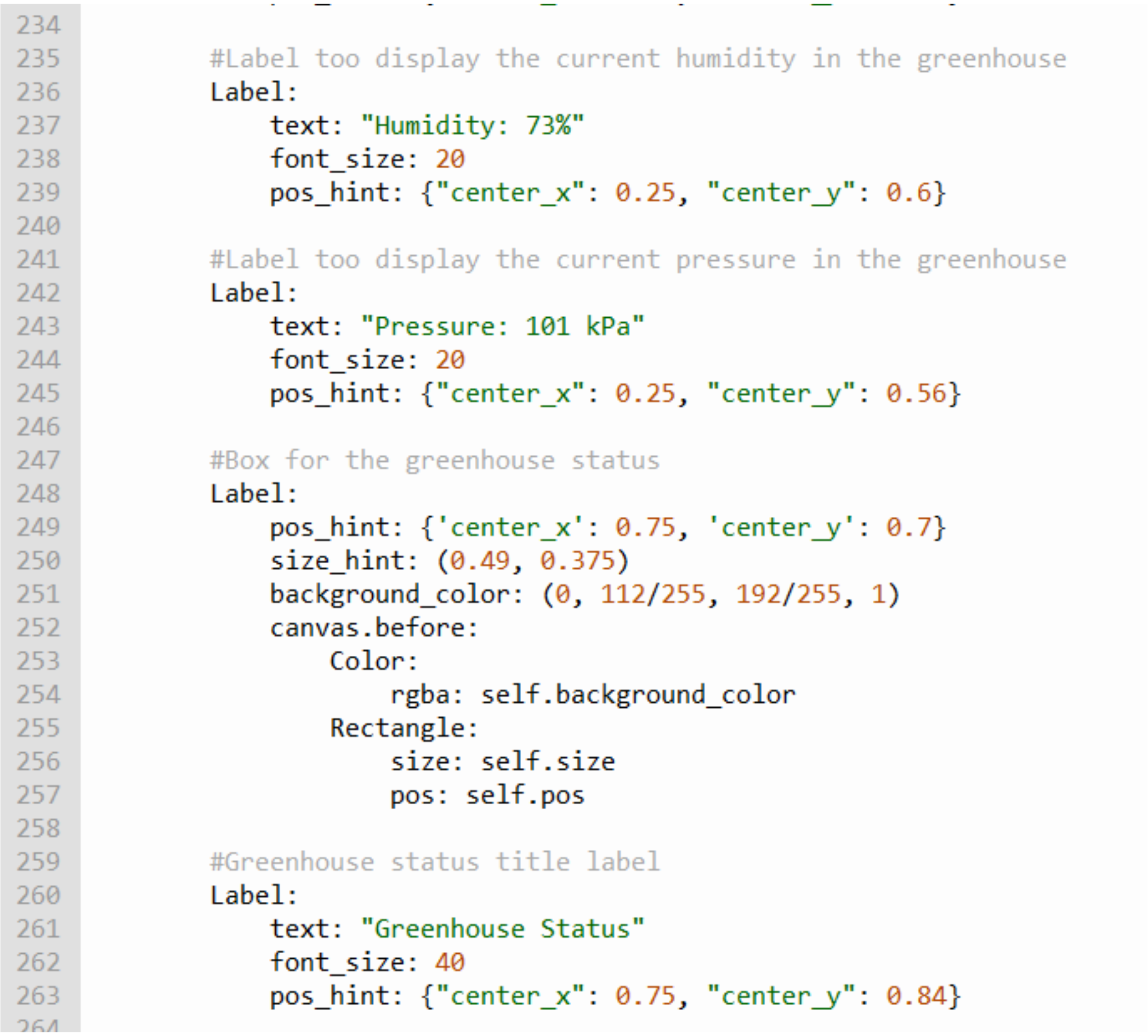
```
        #Parameters page button
        Button:
            text: "Parameters"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.375, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #When pressed move too the parameters page
            on_press: root.manager.current = "parameters"

        #Graphs page button
        Button:
            text: "Graphs"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.625, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #When pressed move too the graphs page
            on_press: root.manager.current = "graphs"
```



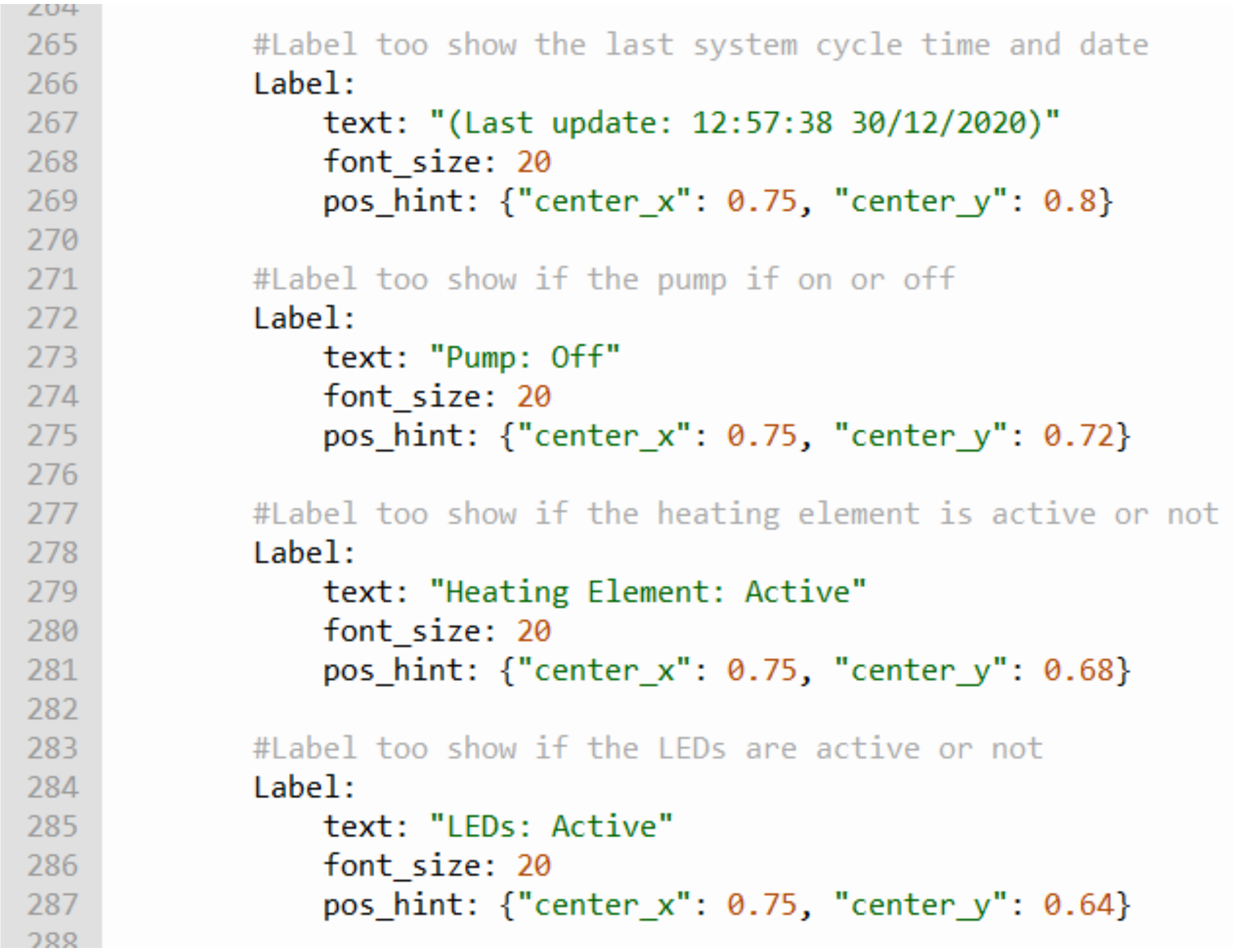
```
        #Settings page button
        Button:
            text: "Settings"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.875, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #When pressed move too the graphs page
            on_press: root.manager.current = "settings"

        #Box for the greenhouse measurments
        Label:
            pos_hint: {'center_x': 0.25, 'center_y': 0.7}
            size_hint: (0.49, 0.375)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos
```



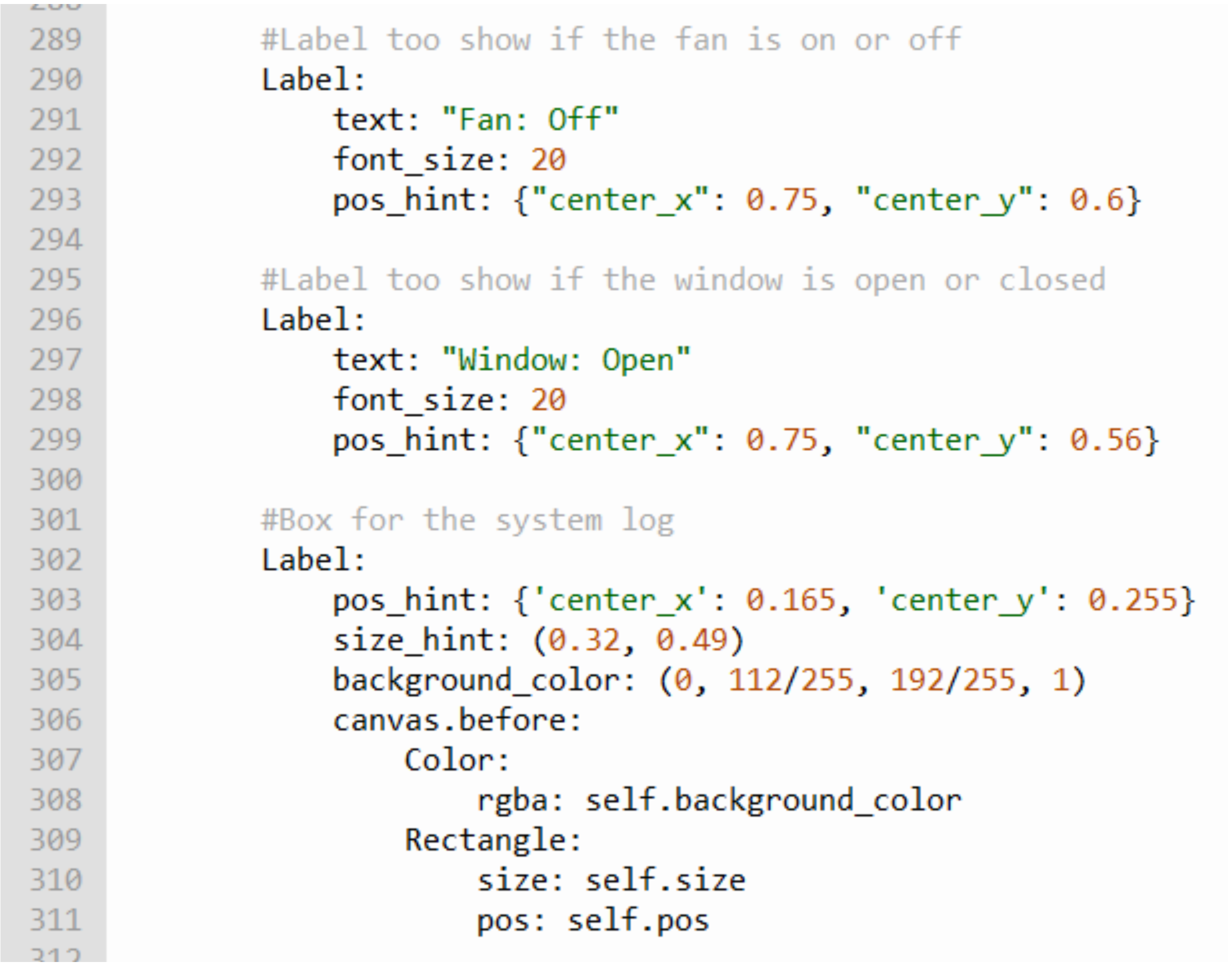
```
        #Greenhouse measurements title
        Label:
            text: "Greenhouse Measurements"
            font_size: 40
            pos_hint: {"center_x": 0.25, "center_y": 0.84}

        #Label too display the last system cycle date and time
        Label:
            text: "(Last update: 12:57:38 30/12/2020)"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.8}

        #Label too display the current temperature in the greenhouse
        Label:
            text: "Internal Temperature: 18 \N{DEGREE SIGN}C"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.72}

        #Label too display the current mositure level in the greenhouse
        Label:
            text: "Soil moisture level: 60%"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.68}

        #Label too display the curren light intensity in the greenhouse
        Label:
            text: "Light intensity: 700 lumens"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.64}
```



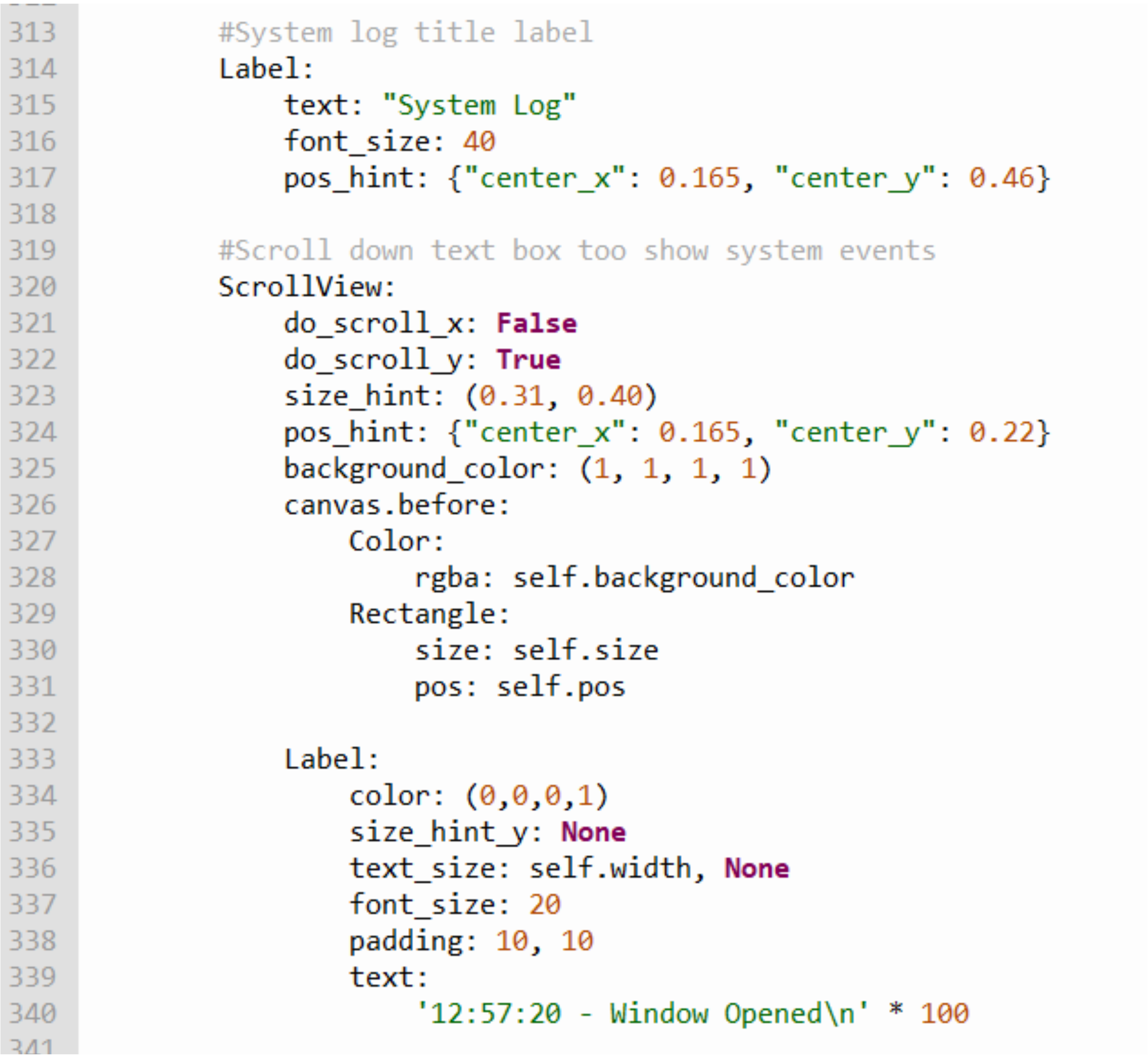
```
        #Label too display the current humidity in the greenhouse
        Label:
            text: "Humidity: 73%"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.6}

        #Label too display the current pressure in the greenhouse
        Label:
            text: "Pressure: 101 kPa"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.56}

        #Box for the greenhouse status
        Label:
            pos_hint: {'center_x': 0.75, 'center_y': 0.7}
            size_hint: (0.49, 0.375)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Greenhouse status title label
        Label:
            text: "Greenhouse Status"
            font_size: 40
            pos_hint: {"center_x": 0.75, "center_y": 0.84}
```



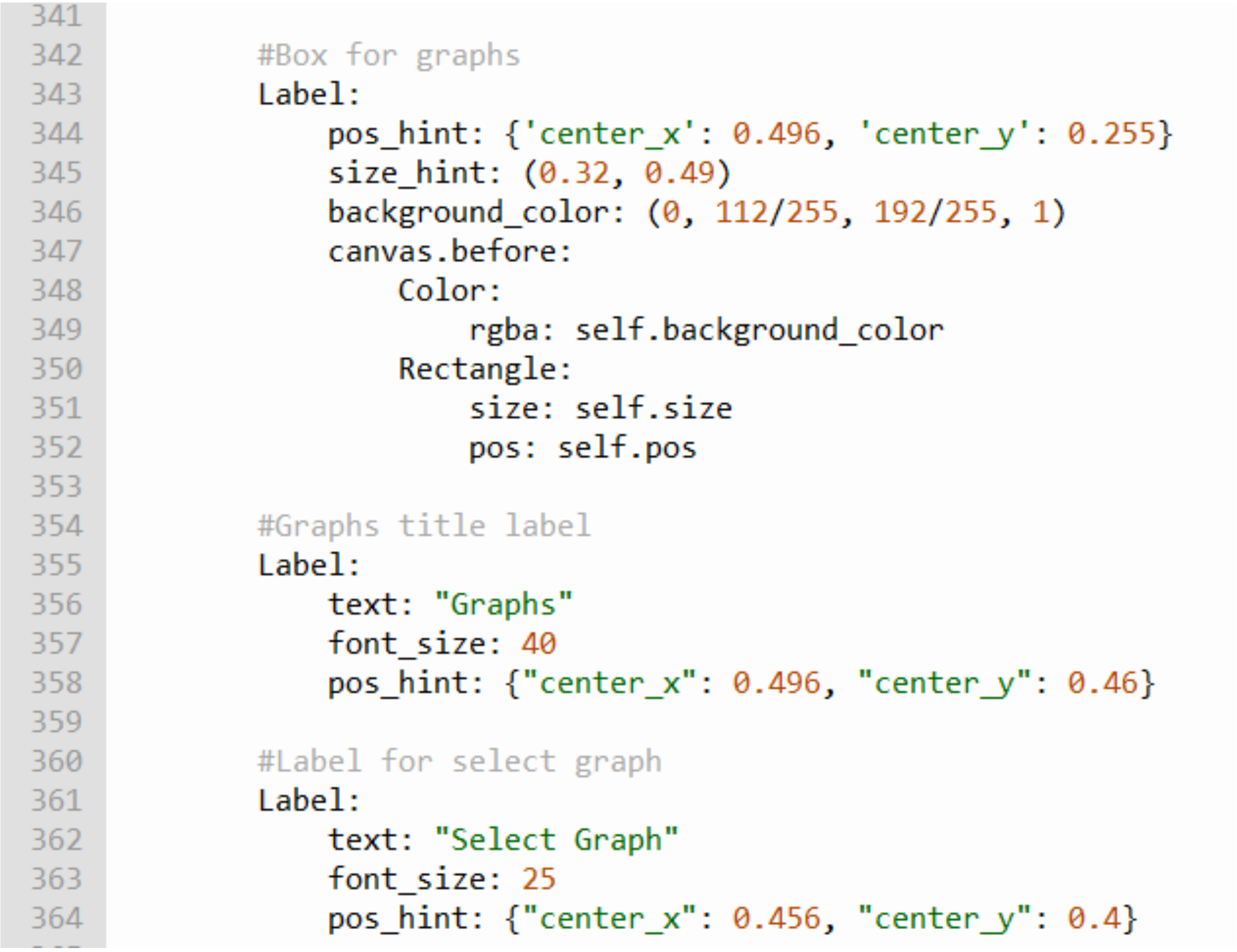
```
        #Label too show the last system cycle time and date
        Label:
            text: "(Last update: 12:57:38 30/12/2020)"
            font_size: 20
            pos_hint: {"center_x": 0.75, "center_y": 0.8}

        #Label too show if the pump if on or off
        Label:
            text: "Pump: Off"
            font_size: 20
            pos_hint: {"center_x": 0.75, "center_y": 0.72}

        #Label too show if the heating element is active or not
        Label:
            text: "Heating Element: Active"
            font_size: 20
            pos_hint: {"center_x": 0.75, "center_y": 0.68}

        #Label too show if the LEDs are active or not
        Label:
            text: "LEDs: Active"
            font_size: 20
            pos_hint: {"center_x": 0.75, "center_y": 0.64}
```



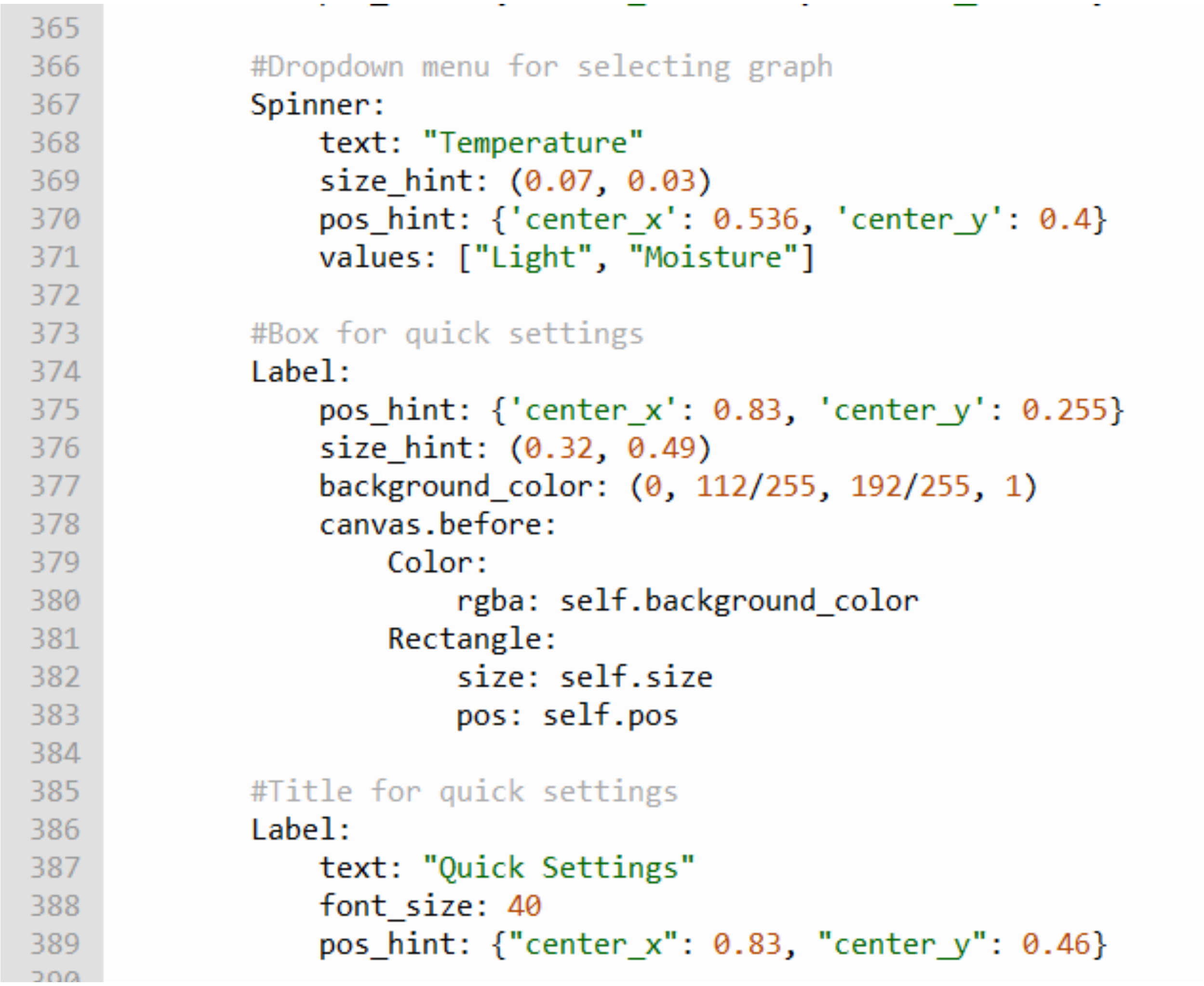
```
        #Label too show if the fan is on or off
        Label:
            text: "Fan: Off"
            font_size: 20
            pos_hint: {"center_x": 0.75, "center_y": 0.6}

        #Label too show if the window is open or closed
        Label:
            text: "Window: Open"
            font_size: 20
            pos_hint: {"center_x": 0.75, "center_y": 0.56}

        #Box for the system log
        Label:
            pos_hint: {'center_x': 0.165, 'center_y': 0.255}
            size_hint: (0.32, 0.49)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos
```



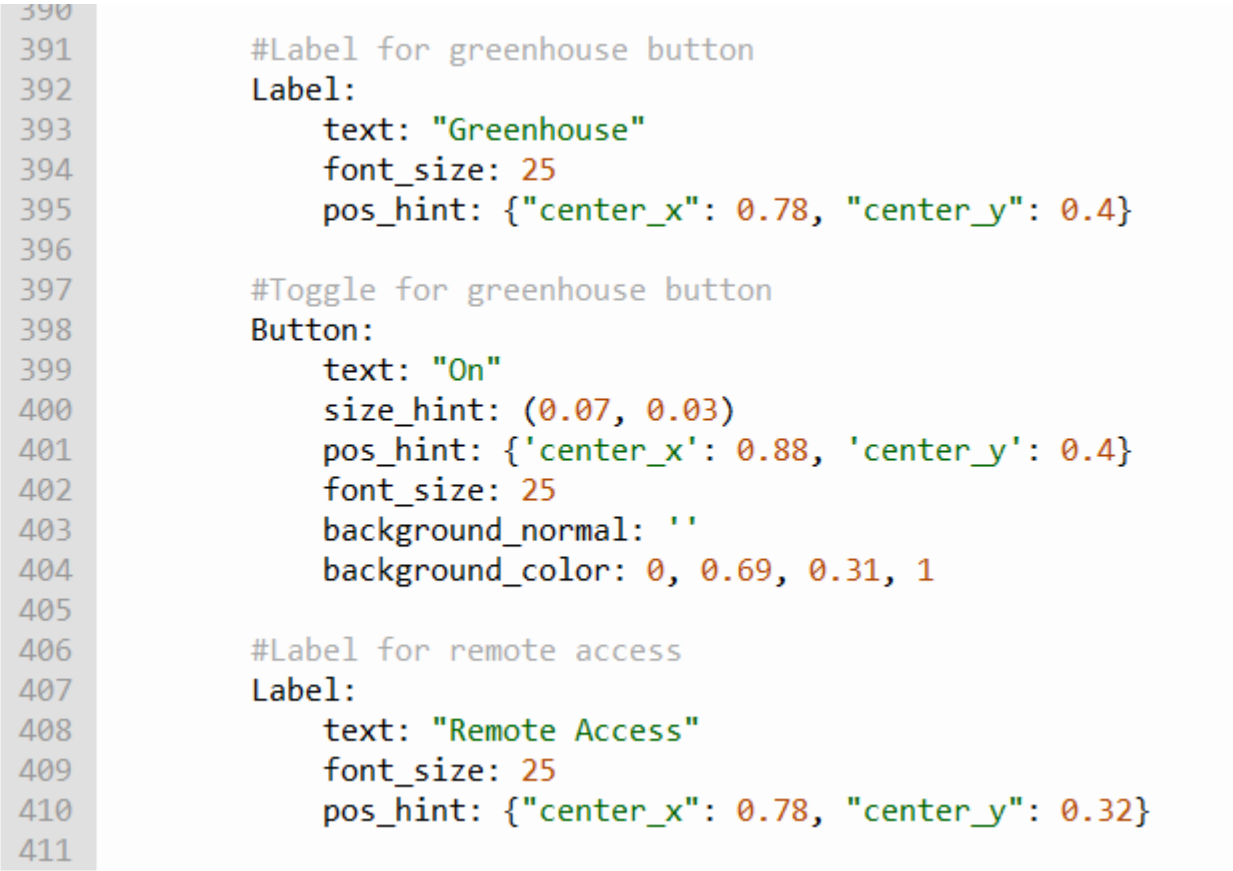
```
        #System log title label
        Label:
            text: "System Log"
            font_size: 40
            pos_hint: {"center_x": 0.165, "center_y": 0.46}

        #Scroll down text box too show system events
        ScrollView:
            do_scroll_x: False
            do_scroll_y: True
            size_hint: (0.31, 0.40)
            pos_hint: {"center_x": 0.165, "center_y": 0.22}
            background_color: (1, 1, 1, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

            Label:
                color: (0,0,0,1)
                size_hint_y: None
                text_size: self.width, None
                font_size: 20
                padding: 10, 10
                text:
                    '12:57:20 - Window Opened\n' * 100
```



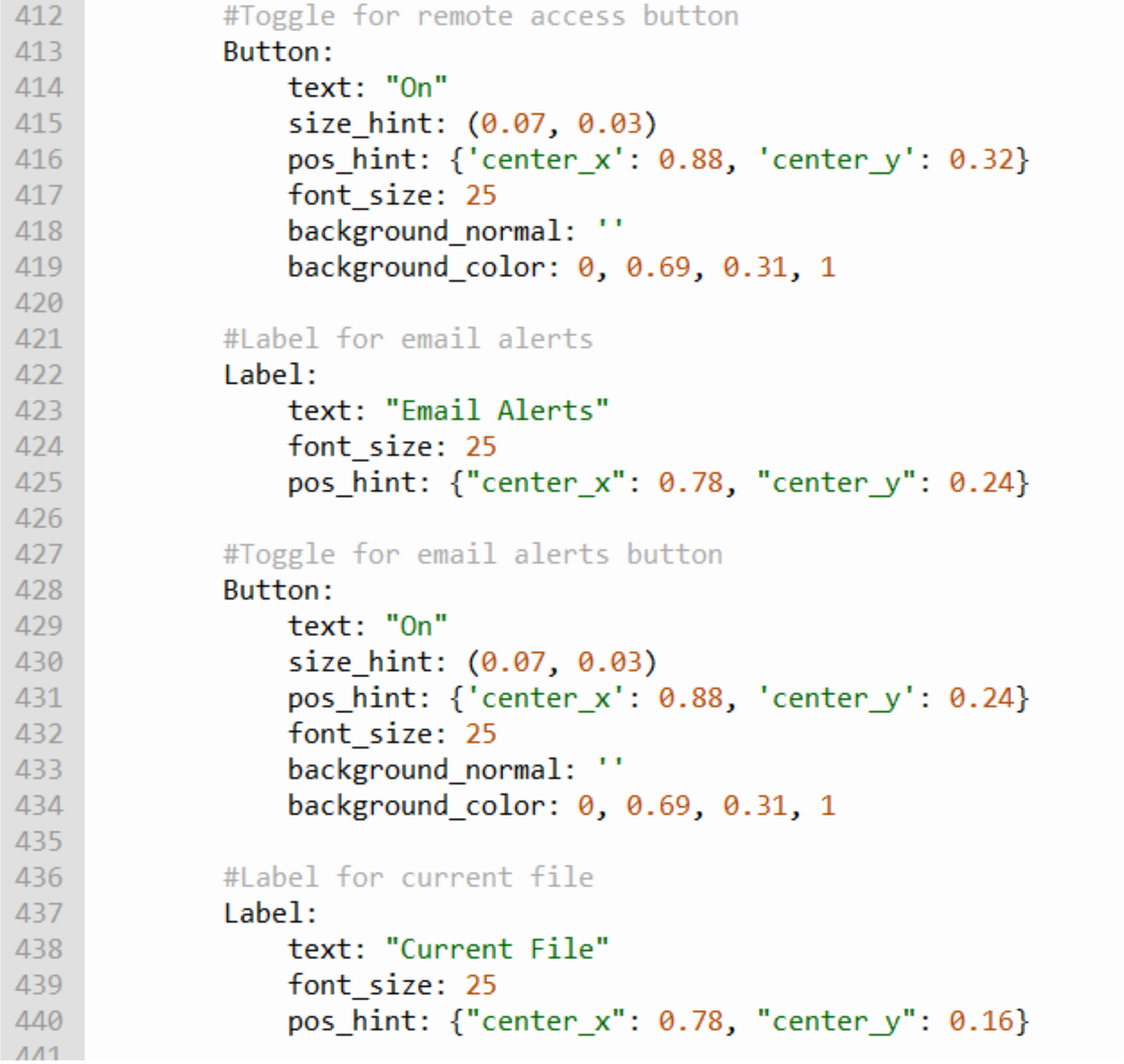
```

        #Box for graphs
        Label:
            pos_hint: {'center_x': 0.496, 'center_y': 0.255}
            size_hint: (0.32, 0.49)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Graphs title label
        Label:
            text: "Graphs"
            font_size: 40
            pos_hint: {"center_x": 0.496, "center_y": 0.46}

        #Label for select graph
        Label:
            text: "Select Graph"
            font_size: 25
            pos_hint: {"center_x": 0.456, "center_y": 0.4}
```



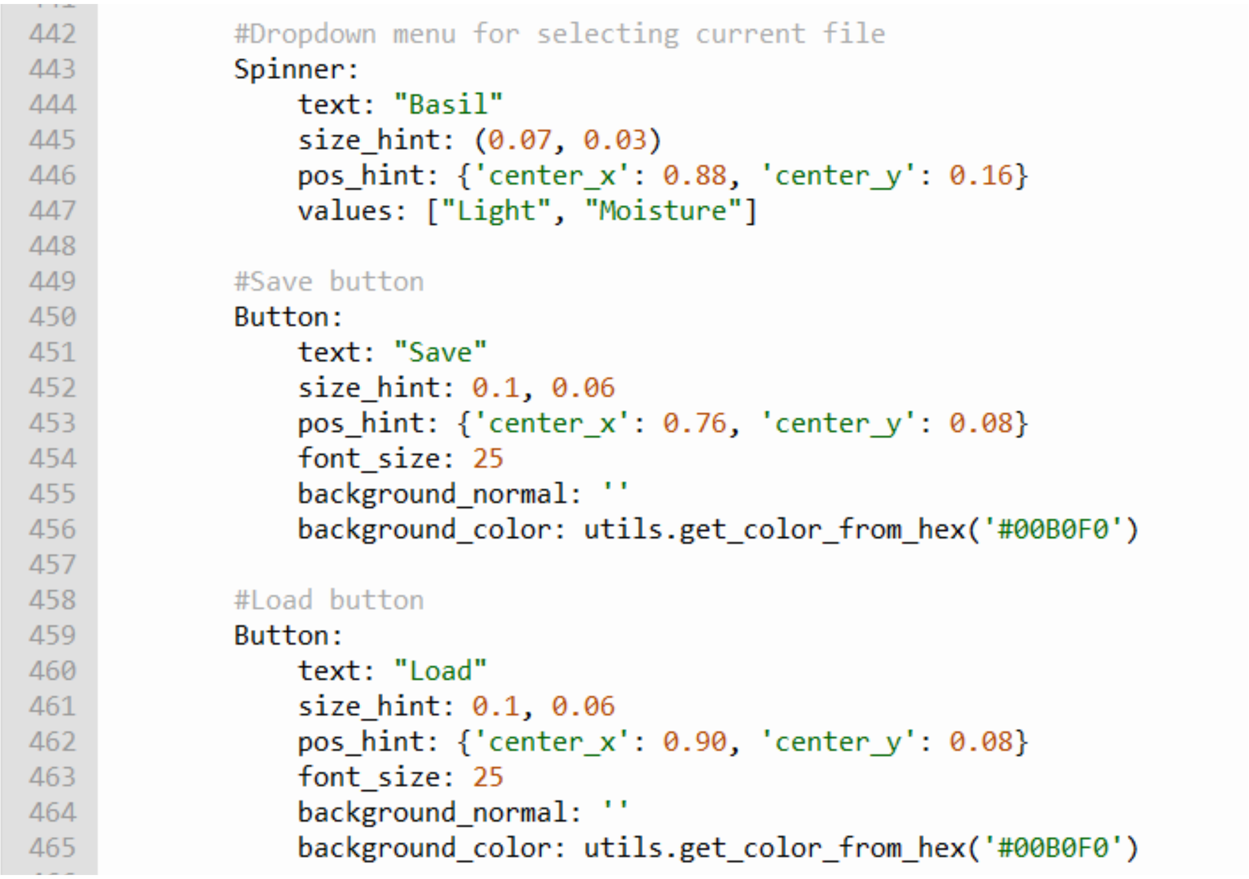
```

        #Dropdown menu for selecting graph
        Spinner:
            text: "Temperature"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.536, 'center_y': 0.4}
            values: ["Light", "Moisture"]

        #Box for quick settings
        Label:
            pos_hint: {'center_x': 0.83, 'center_y': 0.255}
            size_hint: (0.32, 0.49)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Title for quick settings
        Label:
            text: "Quick Settings"
            font_size: 40
            pos_hint: {"center_x": 0.83, "center_y": 0.46}
```



```
        #Label for greenhouse button
        Label:
            text: "Greenhouse"
            font_size: 25
            pos_hint: {"center_x": 0.78, "center_y": 0.4}

        #Toggle for greenhouse button
        Button:
            text: "On"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.88, 'center_y': 0.4}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1

        #Label for remote access
        Label:
            text: "Remote Access"
            font_size: 25
            pos_hint: {"center_x": 0.78, "center_y": 0.32}

```



```
        #Toggle for remote access button
        Button:
            text: "On"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.88, 'center_y': 0.32}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1

        #Label for email alerts
        Label:
            text: "Email Alerts"
            font_size: 25
            pos_hint: {"center_x": 0.78, "center_y": 0.24}

        #Toggle for email alerts button
        Button:
            text: "On"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.88, 'center_y': 0.24}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1

        #Label for current file
        Label:
            text: "Current File"
            font_size: 25
            pos_hint: {"center_x": 0.78, "center_y": 0.16}

```



```
        #Dropdown menu for selecting current file
        Spinner:
            text: "Basil"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.88, 'center_y': 0.16}
            values: ["Light", "Moisture"]

        #Save button
        Button:
            text: "Save"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.76, 'center_y': 0.08}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

        #Load button
        Button:
            text: "Load"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.90, 'center_y': 0.08}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
```

Test Plan – Main Menu

In this testing plan I will be verifying the function of the dropdown menu and the scroll view. As the toggle buttons have not been programmed yet I will not be including these in the testing plan. I am also anticipating that the scroll view will fail the test plan as for reasons previously discussed it is not working as required. I won't be testing any of the labels or other elements as these have already been implemented and tested in previous screens, so it is assumed these are working.

Test Number	Test Plan	Expected Outcome	Actual Outcome	Pass/Fail
1	Scroll down on the scroll view	The scroll view will move the text down the screen	The scroll view did not scroll the text	Fail
2	Scroll up on the scroll view	The scroll view will move the text up the screen	The scroll view did not scroll the text	Fail
3	Select one of the drop-down menus	The list of elements in the drop-down will be shown	The options in the drop down were shown	Pass
4	Select one of the options in the drop-down	The option will be selected, and the current option	The clicked option was selected and then displayed in	Pass

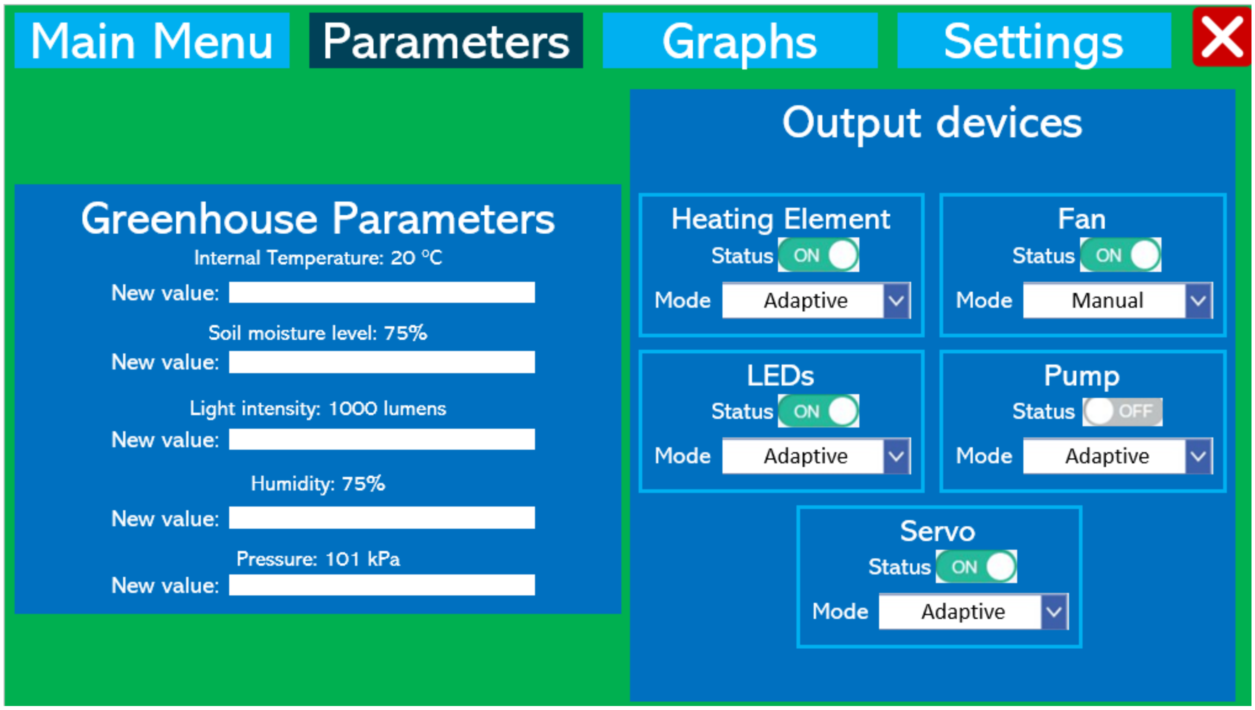
		will replace the text value of the dropdown	the drop-down box	

1	Scroll down on the scroll view	The scroll view will move the text down the screen	The scroll view did not scroll the text	Fail
2	Scroll up on the scroll view	The scroll view will move the text up the screen	The scroll view did not scroll the text	Fail

The scroll view was previously working on my desktop computer however on my laptop the scroll view does not seem to be working. I have decided that I will attempt to find a solution to the scroll view issues once I begin to develop the system log. As depending on how I implement it will have a bearing on the way I implement the scroll view. I suspect that I have not correctly imported the kivy scroll view dependencies as I have followed there help pages for how to create a scroll view.

Development Log – Parameters screen

All the screens left to implement will use a combination of labels, buttons, dropdowns, and text entry boxes. As I have already discussed how these work in kivy I will only provide a brief overview of developing the final pages. These screens are just a different layout of the previously used elements positioned and sized in a different way by changing their size and position hints. For this reason, I won't be completing a testing plan for the final screens as I am confident all the different elements are working apart from the scroll view.



Greenhouse Parameters

Internal Temperature: 20 °C

New value:

Soil moisture level: 75%

New value:

Light intensity: 1000 lumens

New value:

Humidity: 75%

New value:

Pressure: 101 kPa

New value:

Output devices

Heating Element
Status ON
Mode Adaptive

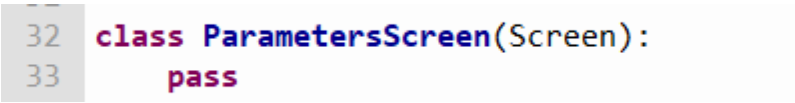
Fan
Status ON
Mode Manual

LEDs
Status ON
Mode Adaptive

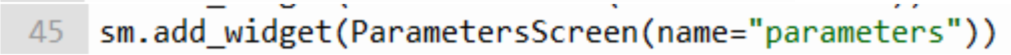
Pump
Status OFF
Mode Adaptive

Servo
Status ON
Mode Adaptive

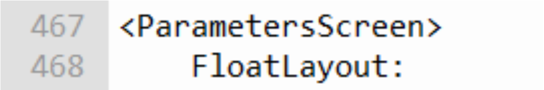
Above is a mockup of the GUI for the parameters page. The page is made up of two sections one for setting the greenhouse parameters and one for controlling the output devices. The screen needs to have a background set in the same way as previously and has the menu along the top with the parameters button having a dark blue background to signify this is the current page. The output devices section has multiple devices with toggle buttons and dropdown menus. Once again, the toggle buttons are not functional at this stage and will be implemented later in development.



```
class ParametersScreen(Screen):
    pass
```

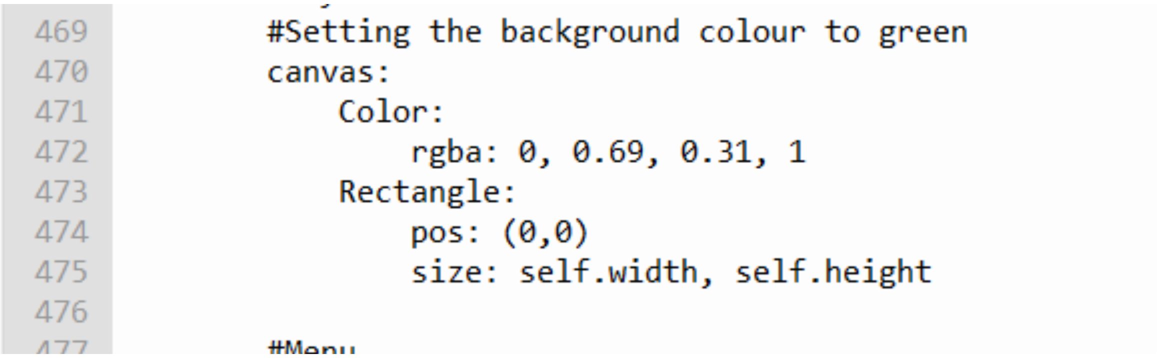


```
sm.add_widget(ParametersScreen(name="parameters"))
```



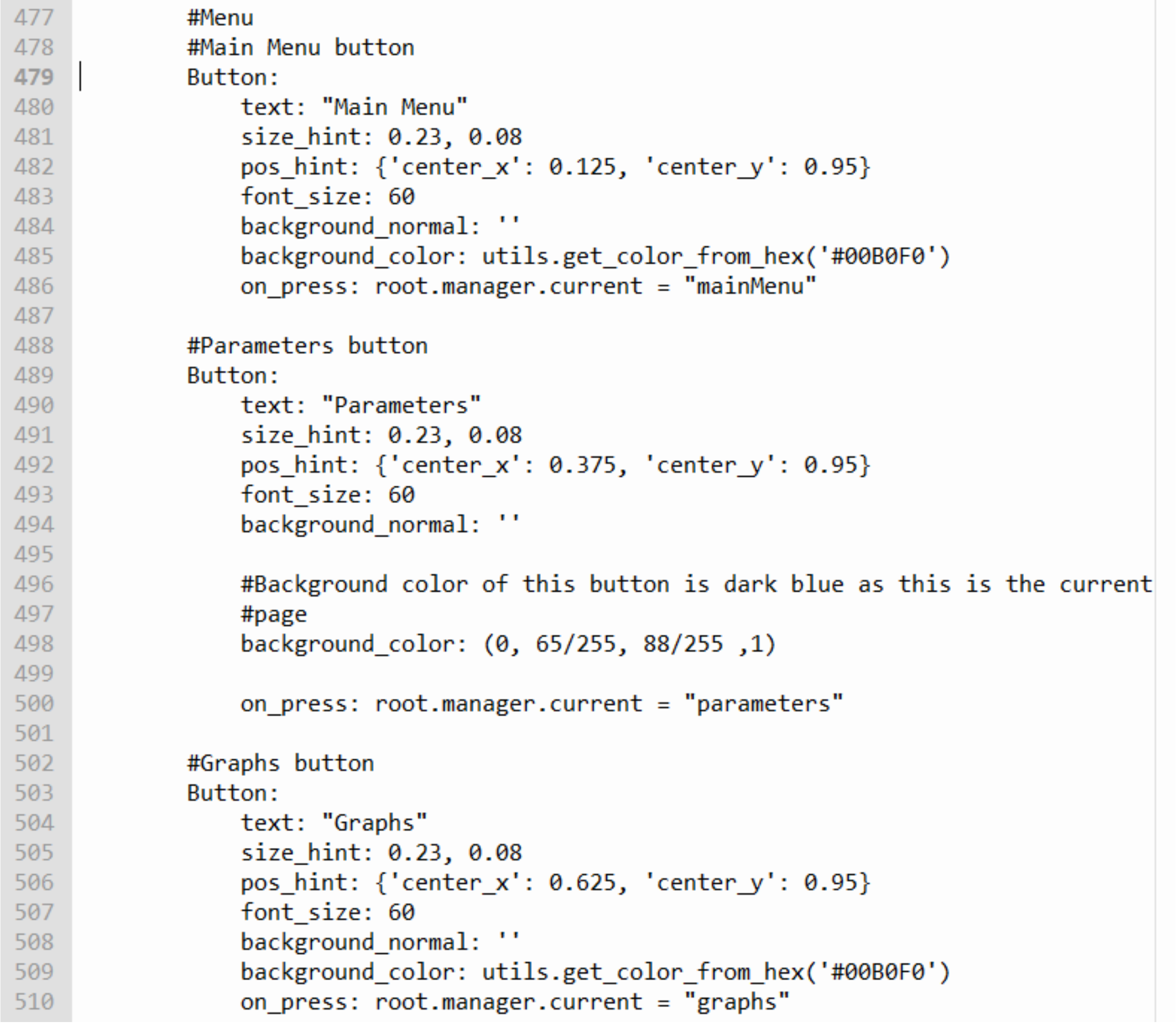
```
<ParametersScreen>
    FloatLayout:
```

I have setup a new screen inside the kivy file and added this too the screen manager widgets along with declaring a new class which is related to the parameters screen. This screen will be using float layout as with all the other screens in this project.

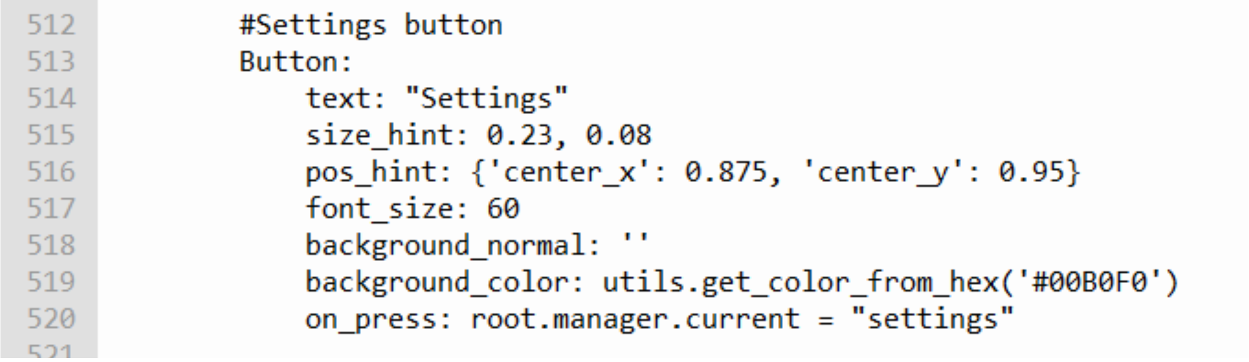


```
        #Setting the background colour to green
        canvas:
            Color:
                rgba: 0, 0.69, 0.31, 1
            Rectangle:
                pos: (0,0)
                size: self.width, self.height

        #Menu
```

The background color of the parameters screen will be green as usual.

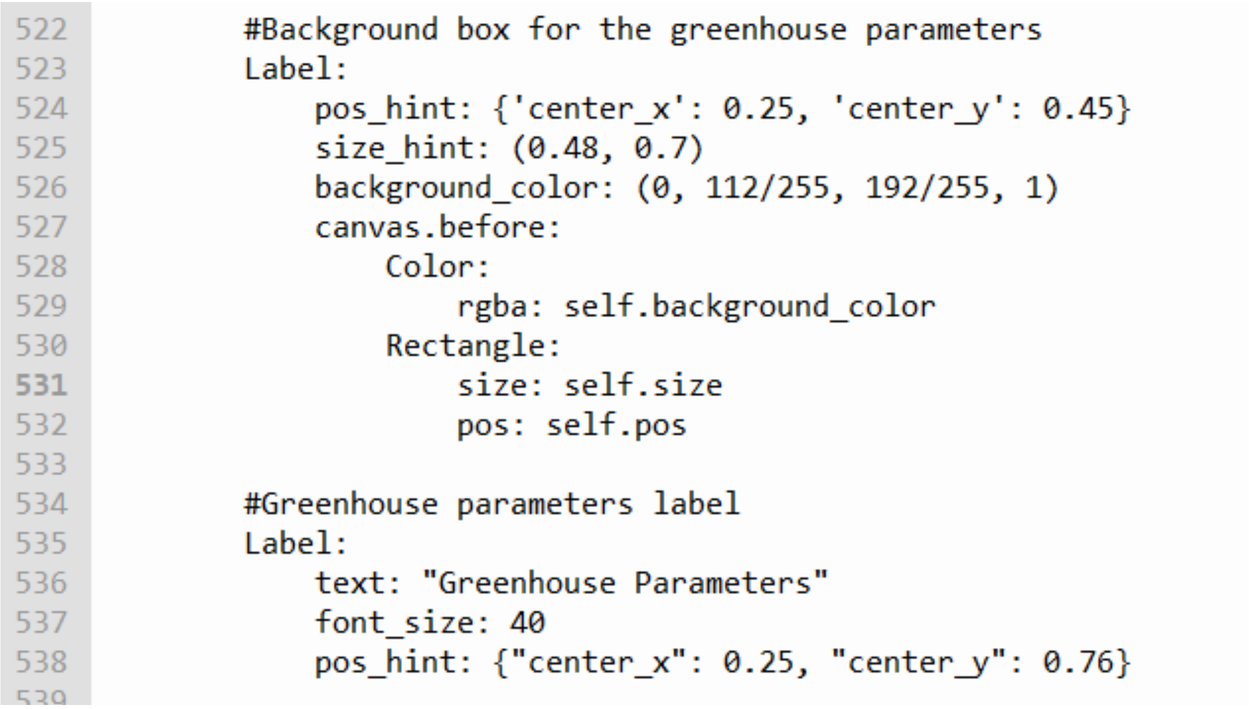


```
        #Menu
        #Main Menu button
        Button:
            text: "Main Menu"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.125, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
            on_press: root.manager.current = "mainMenu"

        #Parameters button
        Button:
            text: "Parameters"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.375, 'center_y': 0.95}
            font_size: 60
            background_normal: ''

            #Background color of this button is dark blue as this is the current
            #page
            background_color: (0, 65/255, 88/255 ,1)

            on_press: root.manager.current = "parameters"

        #Graphs button
        Button:
            text: "Graphs"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.625, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
            on_press: root.manager.current = "graphs"
```



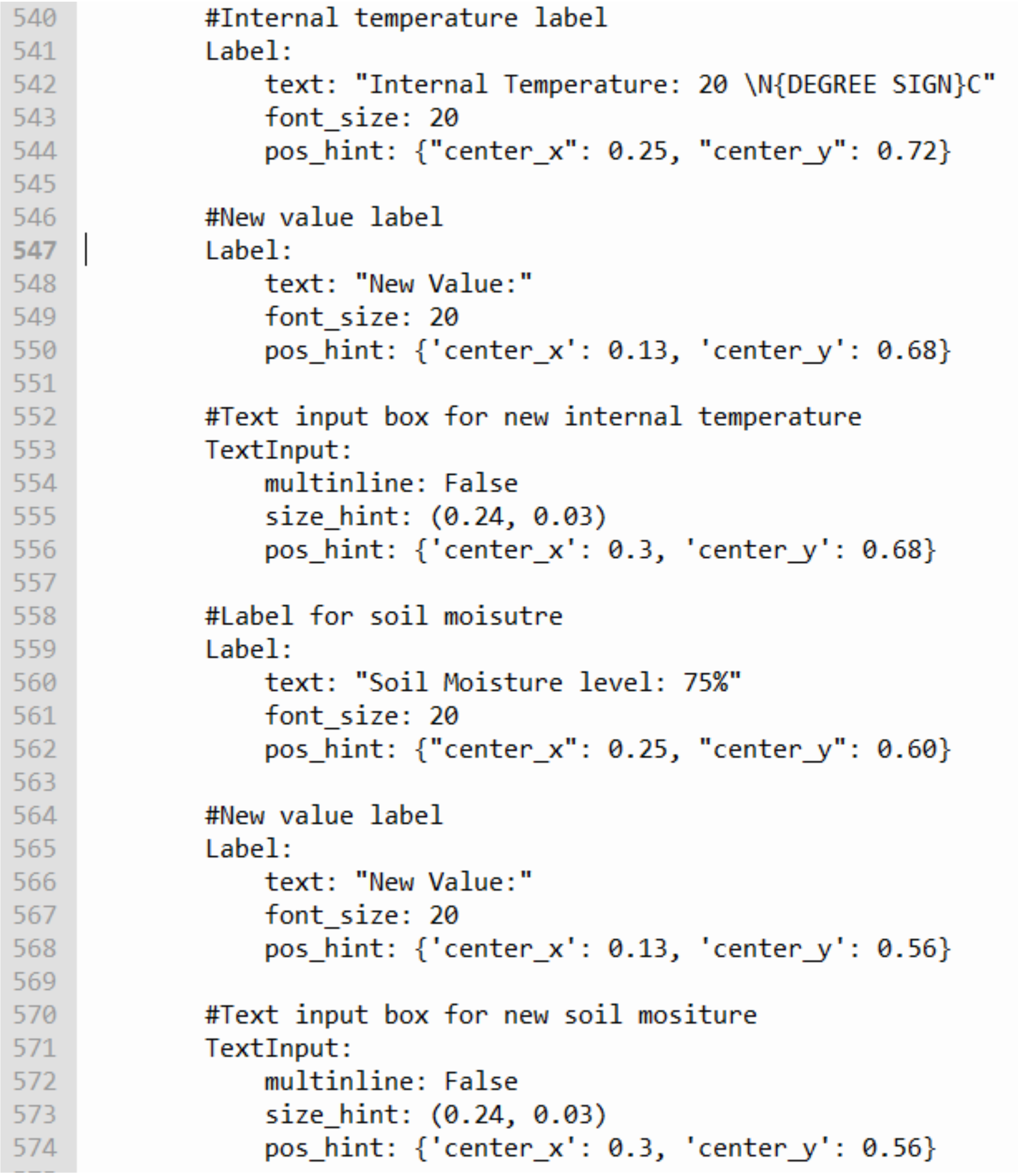
```
        #Settings button
        Button:
            text: "Settings"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.875, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
            on_press: root.manager.current = "settings"
```

The menu along the top of the parameters page is the same as on the main menu with the parameters button being the dark blue one this time.

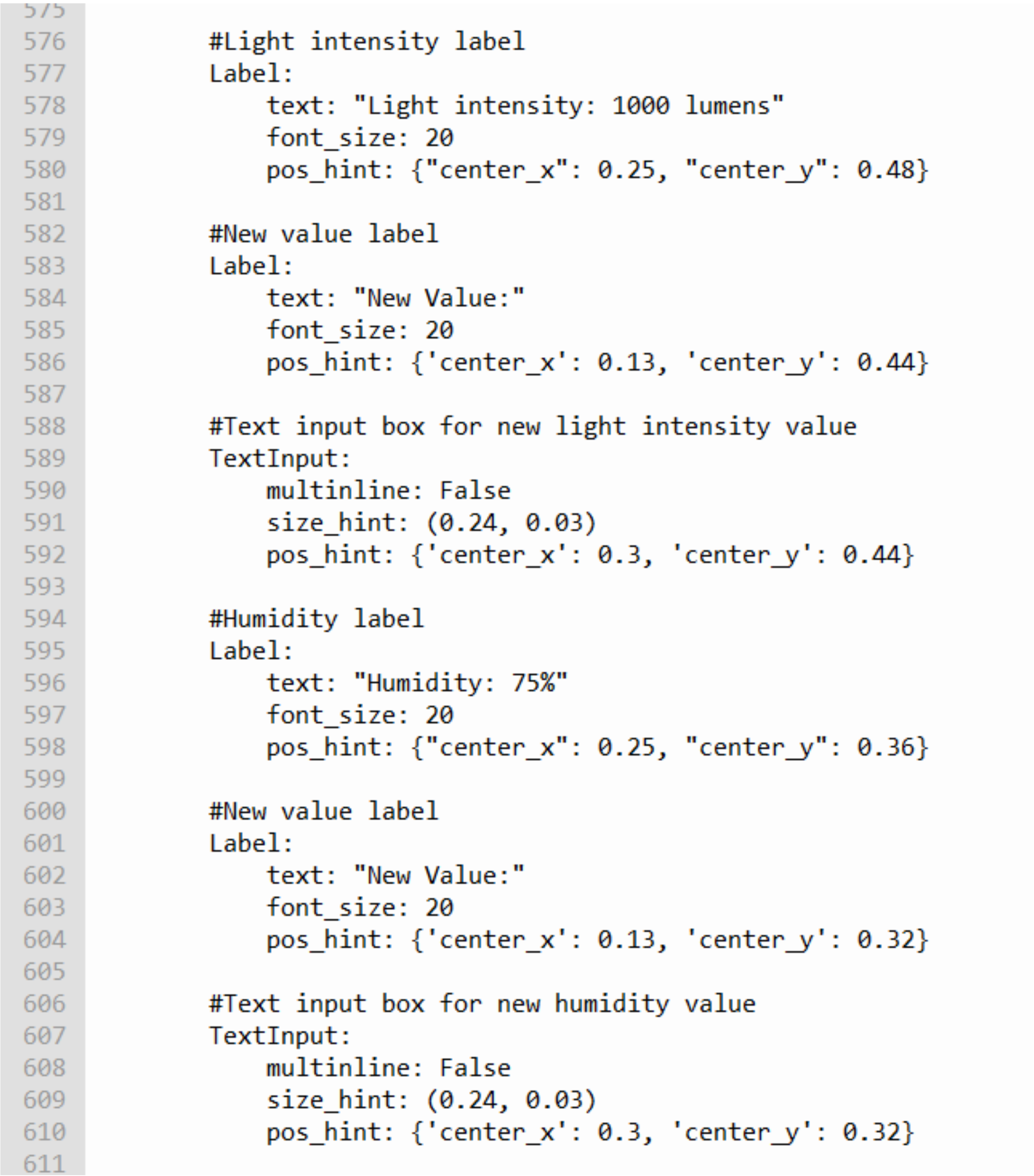


```
        #Background box for the greenhouse parameters
        Label:
            pos_hint: {'center_x': 0.25, 'center_y': 0.45}
            size_hint: (0.48, 0.7)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Greenhouse parameters label
        Label:
            text: "Greenhouse Parameters"
            font_size: 40
            pos_hint: {"center_x": 0.25, "center_y": 0.76}
```

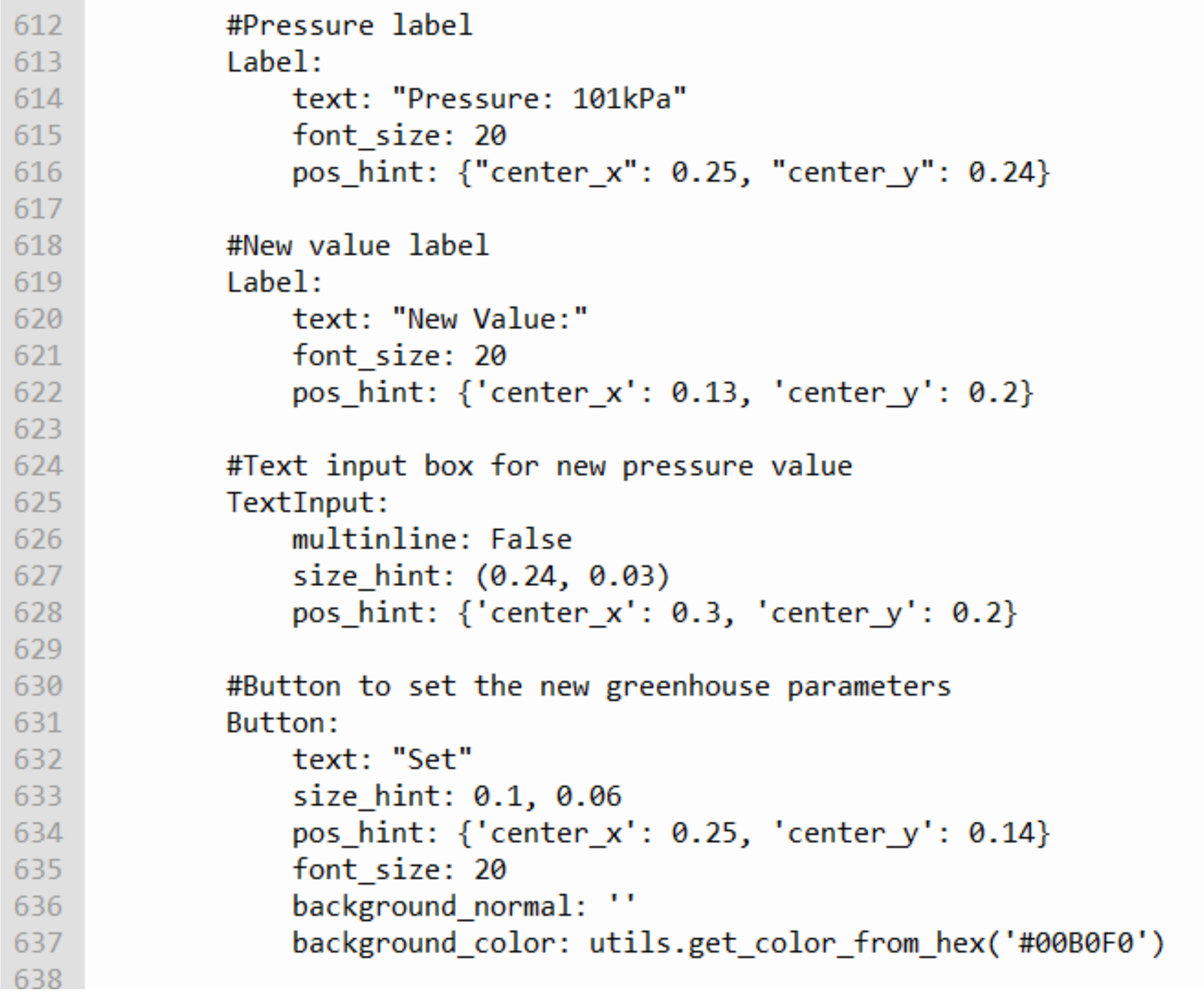
Half of the parameters screen is used for allowing the user to set new greenhouse parameters a background box is added using a label and then the title is added using a label.



```
        #Internal temperature label
        Label:
            text: "Internal Temperature: 20 \N{DEGREE SIGN}C"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.72}

        #New value label
        Label:
            text: "New Value:"
            font_size: 20
            pos_hint: {'center_x': 0.13, 'center_y': 0.68}

        #Text input box for new internal temperature
        TextInput:
            multinline: False
            size_hint: (0.24, 0.03)
            pos_hint: {'center_x': 0.3, 'center_y': 0.68}

        #Label for soil moisutre
        Label:
            text: "Soil Moisture level: 75%"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.60}

        #New value label
        Label:
            text: "New Value:"
            font_size: 20
            pos_hint: {'center_x': 0.13, 'center_y': 0.56}

        #Text input box for new soil mositure
        TextInput:
            multinline: False
            size_hint: (0.24, 0.03)
            pos_hint: {'center_x': 0.3, 'center_y': 0.56}
```

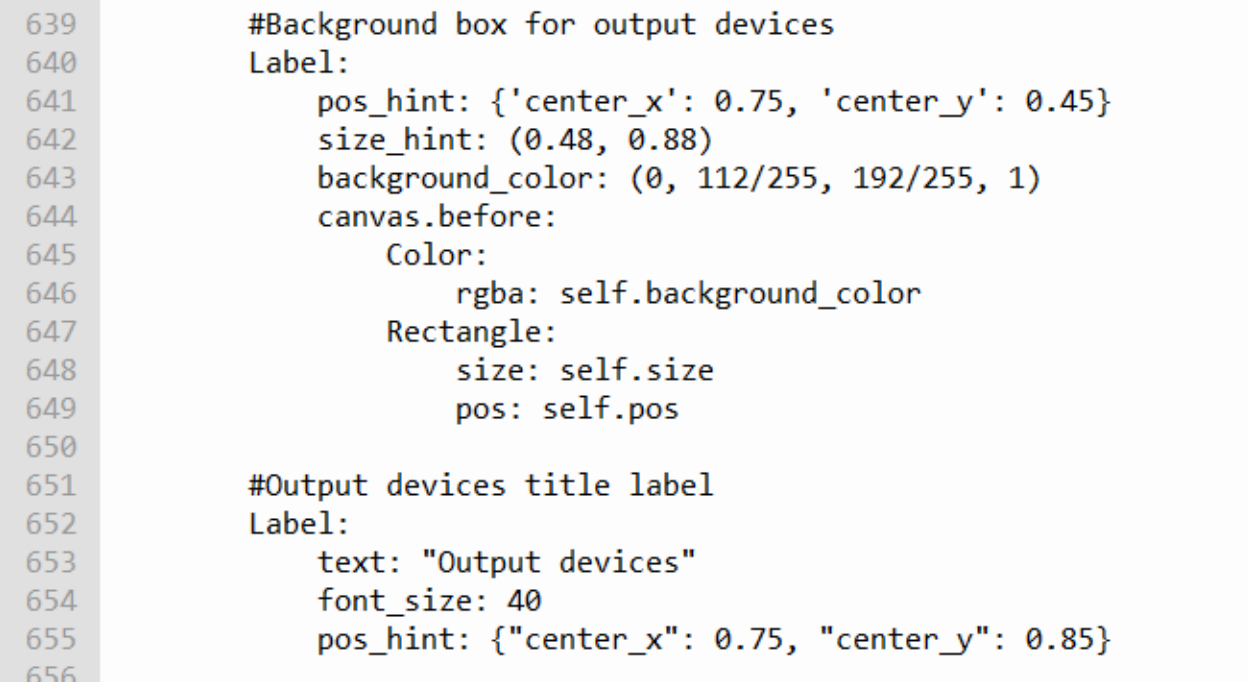


```
        #Light intensity label
        Label:
            text: "Light intensity: 1000 lumens"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.48}

        #New value label
        Label:
            text: "New Value:"
            font_size: 20
            pos_hint: {'center_x': 0.13, 'center_y': 0.44}

        #Text input box for new light intensity value
        TextInput:
            multinline: False
            size_hint: (0.24, 0.03)
            pos_hint: {'center_x': 0.3, 'center_y': 0.44}

        #Humidity label
        Label:
            text: "Humidity: 75%"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.36}

        #New value label
        Label:
            text: "New Value:"
            font_size: 20
            pos_hint: {'center_x': 0.13, 'center_y': 0.32}

        #Text input box for new humidity value
        TextInput:
            multinline: False
            size_hint: (0.24, 0.03)
            pos_hint: {'center_x': 0.3, 'center_y': 0.32}
```



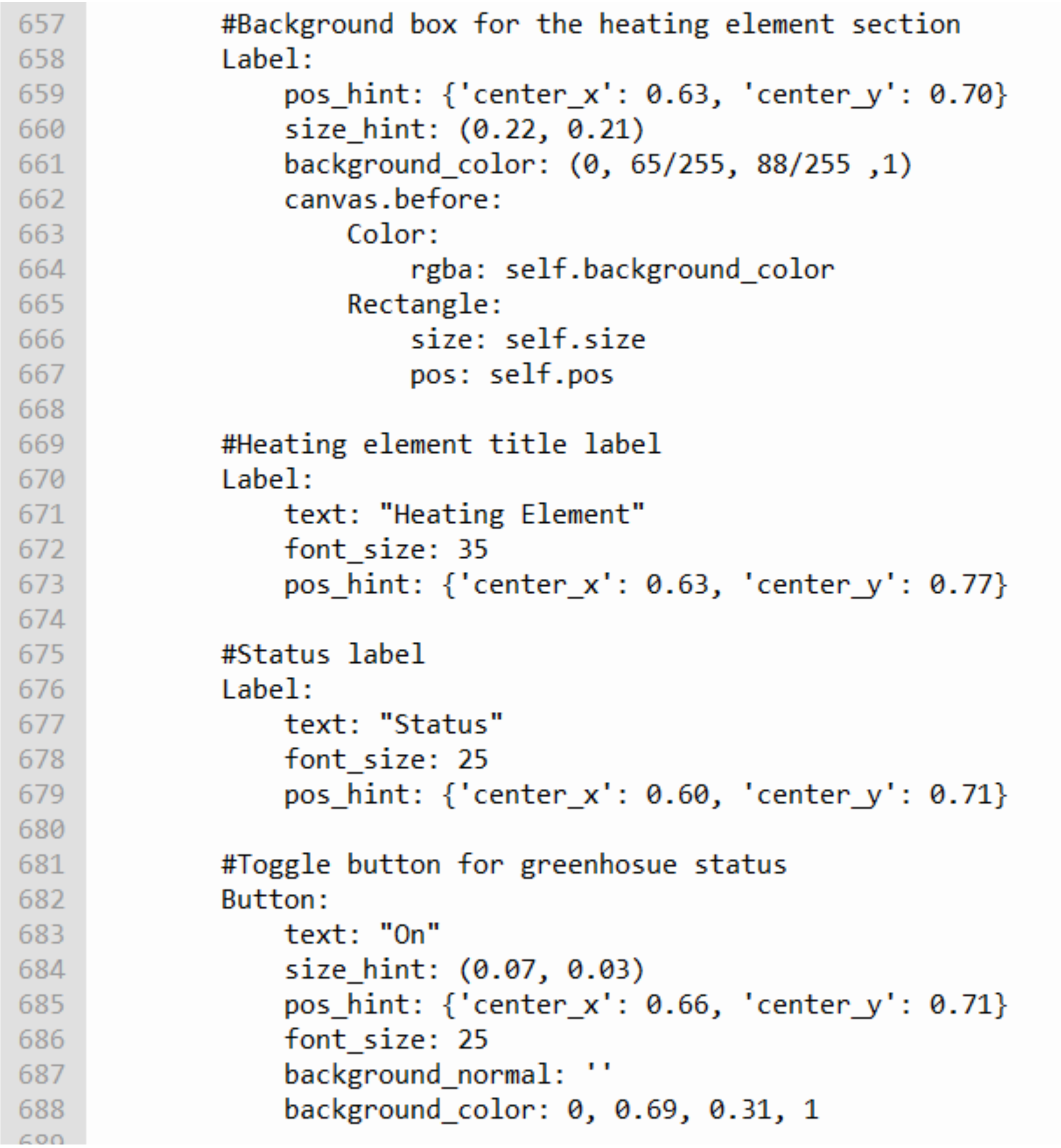
```
        #Pressure label
        Label:
            text: "Pressure: 101kPa"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.24}

        #New value label
        Label:
            text: "New Value:"
            font_size: 20
            pos_hint: {'center_x': 0.13, 'center_y': 0.2}

        #Text input box for new pressure value
        TextInput:
            multinline: False
            size_hint: (0.24, 0.03)
            pos_hint: {'center_x': 0.3, 'center_y': 0.2}

        #Button to set the new greenhouse parameters
        Button:
            text: "Set"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.25, 'center_y': 0.14}
            font_size: 20
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
```

Inside the greenhouse parameters box, I have added labels for each parameter which can showing the current value of that parameter. I have then added a text input box for each parameter which the user can input their new desired value into. At the bottom of the parameters section there is a set button which when pressed will store the new parameter values.

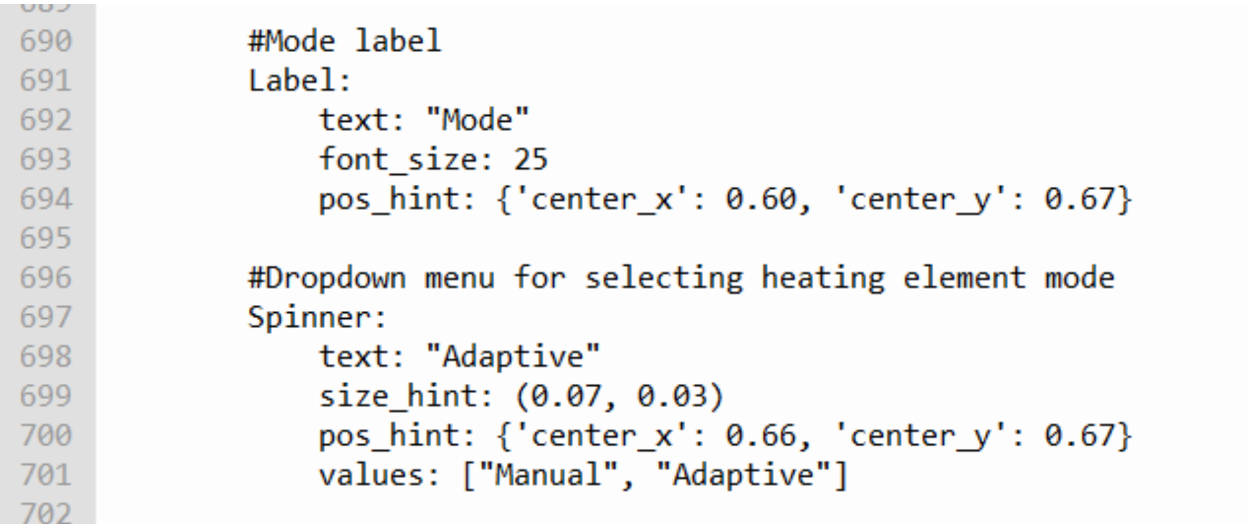


```
        #Background box for output devices
        Label:
            pos_hint: {'center_x': 0.75, 'center_y': 0.45}
            size_hint: (0.48, 0.88)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Output devices title label
        Label:
            text: "Output devices"
            font_size: 40
            pos_hint: {"center_x": 0.75, "center_y": 0.85}
```

On the other half of the page is the output devices section where the user can control how the various output device’s function.

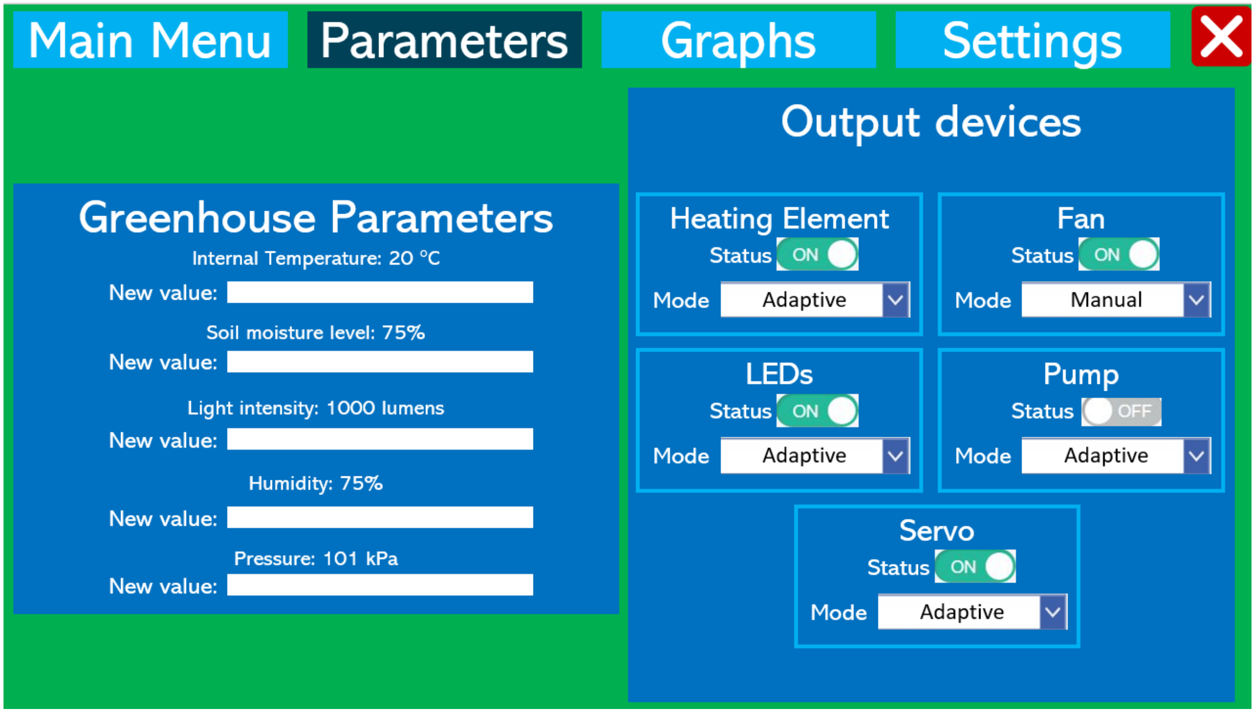


```
        #Background box for the heating element section
        Label:
            pos_hint: {'center_x': 0.63, 'center_y': 0.70}
            size_hint: (0.22, 0.21)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Heating element title label
        Label:
            text: "Heating Element"
            font_size: 35
            pos_hint: {'center_x': 0.63, 'center_y': 0.77}

        #Status label
        Label:
            text: "Status"
            font_size: 25
            pos_hint: {'center_x': 0.60, 'center_y': 0.71}

        #Toggle button for greenhosue status
        Button:
            text: "On"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.66, 'center_y': 0.71}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1
```



```
        #Mode label
        Label:
            text: "Mode"
            font_size: 25
            pos_hint: {'center_x': 0.60, 'center_y': 0.67}

        #Dropdown menu for selecting heating element mode
        Spinner:
            text: "Adaptive"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.66, 'center_y': 0.67}
            values: ["Manual", "Adaptive"]
```

There are 5 output devices which can be controlled they all follow the same structure as above. They have a background box in light blue along with a title saying which device the box is controlling and then a toggle button to turn the device on and off and a mode dropdown menu so the user can select what mode the device is functioning in. In the GUI mockup the boxes were going to be a light blue outline however I could not find a way too have a transparent box with a boarder so instead I swapped to use a dark blue background for the output devices.



Greenhouse Parameters

Internal Temperature: 20 °C
New value:

Soil moisture level: 75%
New value:

Light intensity: 1000 lumens
New value:

Humidity: 75%
New value:

Pressure: 101 kPa
New value:

Output devices

Heating Element
Status ON
Mode Adaptive

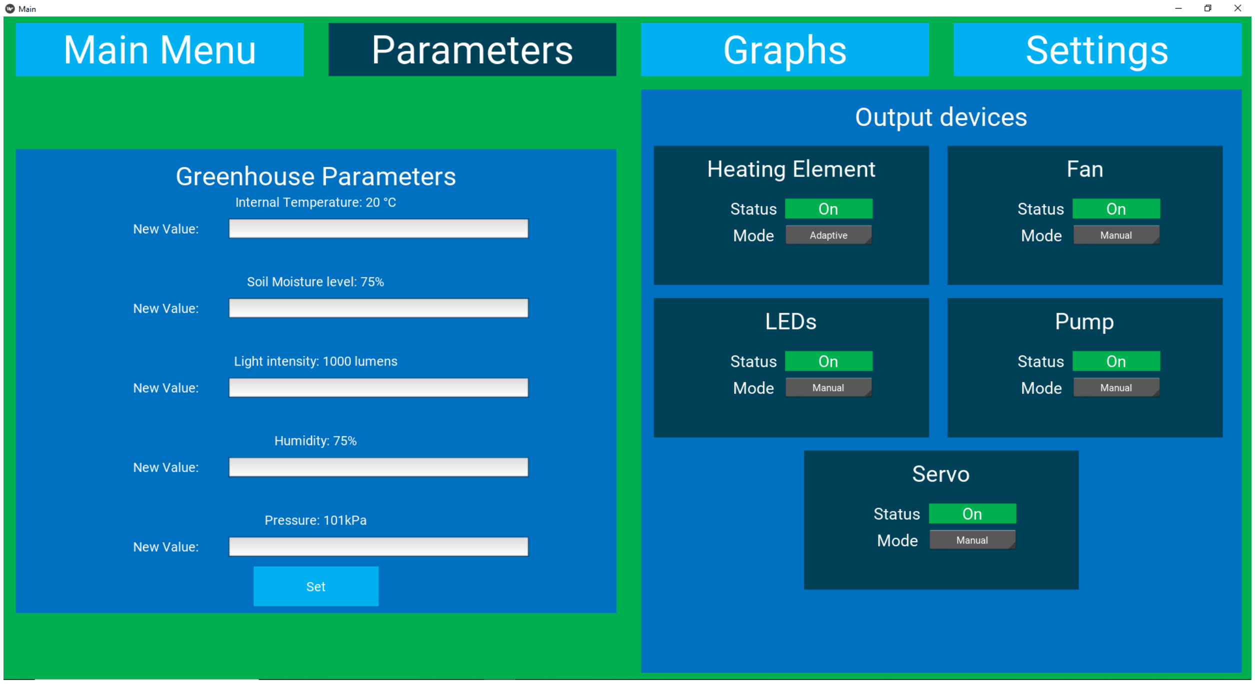
Fan
Status ON
Mode Manual

LEDs
Status ON
Mode Adaptive

Pump
Status OFF
Mode Adaptive

Servo
Status ON
Mode Adaptive

GUI mockup



Greenhouse Parameters

Internal Temperature: 20 °C
New Value:

Soil Moisture level: 75%
New Value:

Light intensity: 1000 lumens
New Value:

Humidity: 75%
New Value:

Pressure: 101kPa
New Value:

Output devices

Heating Element
Status On
Mode Adaptive

Fan
Status On
Mode Manual

LEDs
Status On
Mode Manual

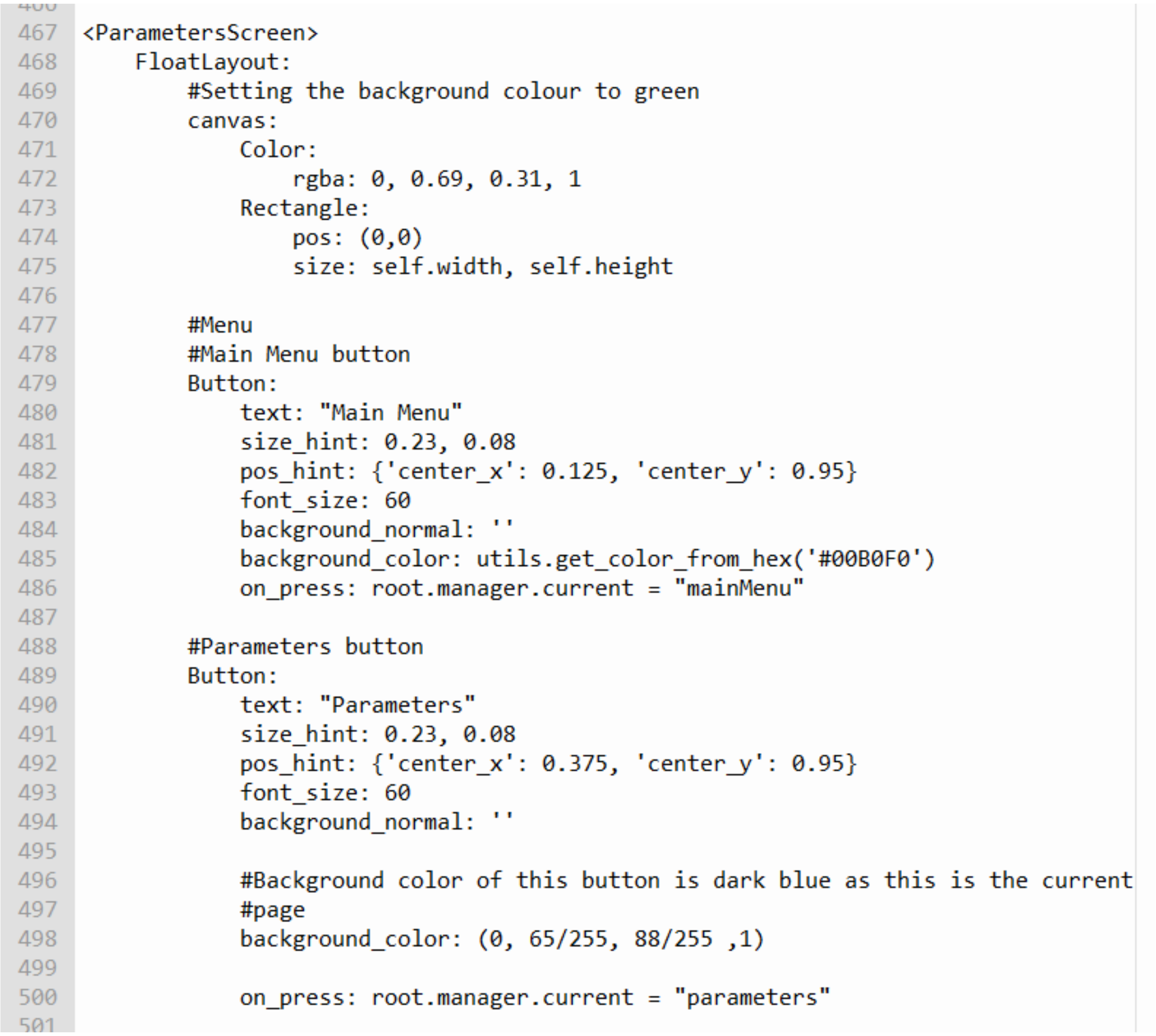
Pump
Status On
Mode Manual

Servo
Status On
Mode Manual

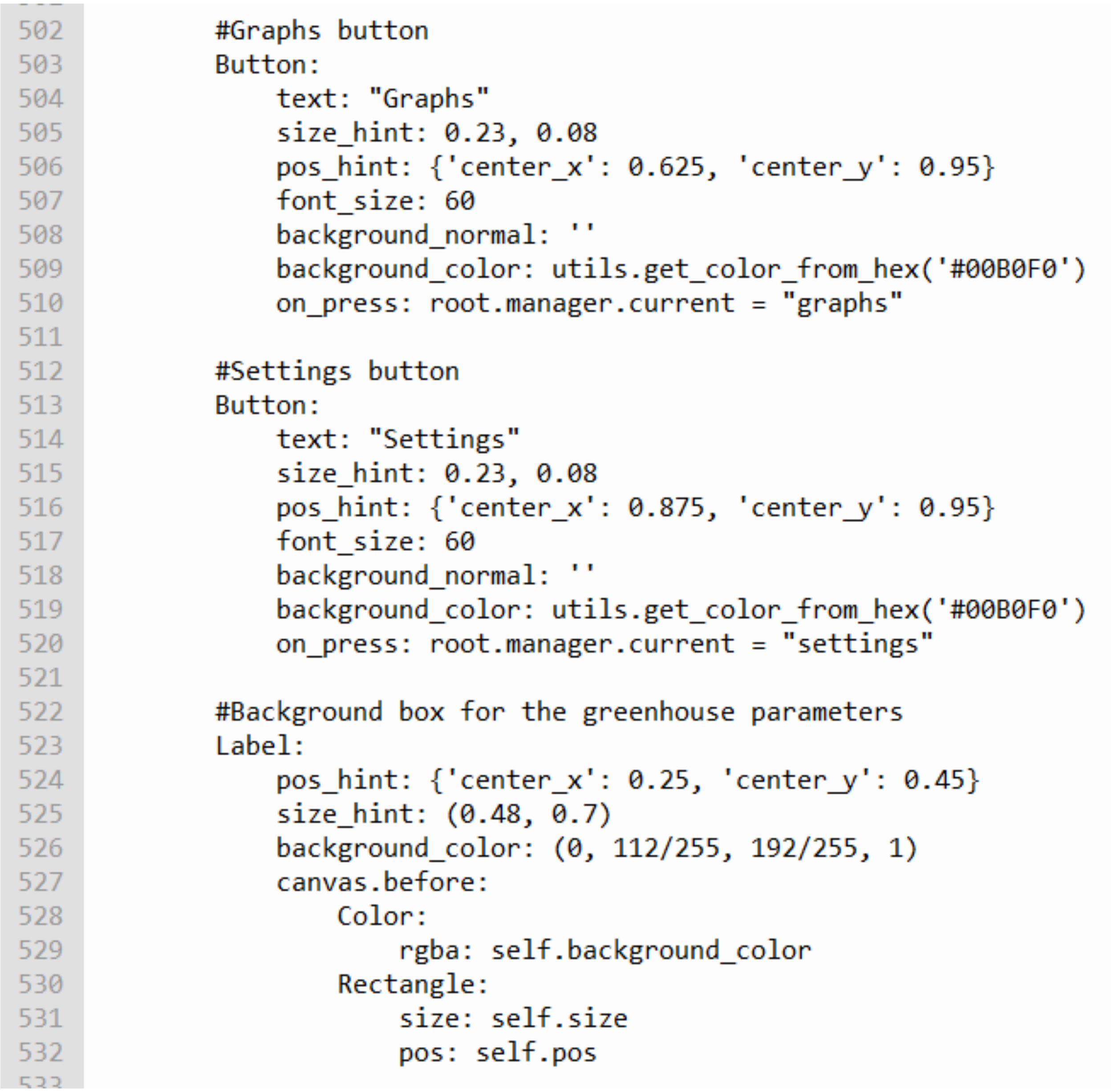
Kivy

The kivy implementation of the parameters page is not completely true to the GUI design. As mentioned in the output devices section I have swapped from using a light blue outline to using a full dark blue background due to kivy restrictions. Other differences relate to the status on off toggle button and the alignment of the boxes in the output devices.

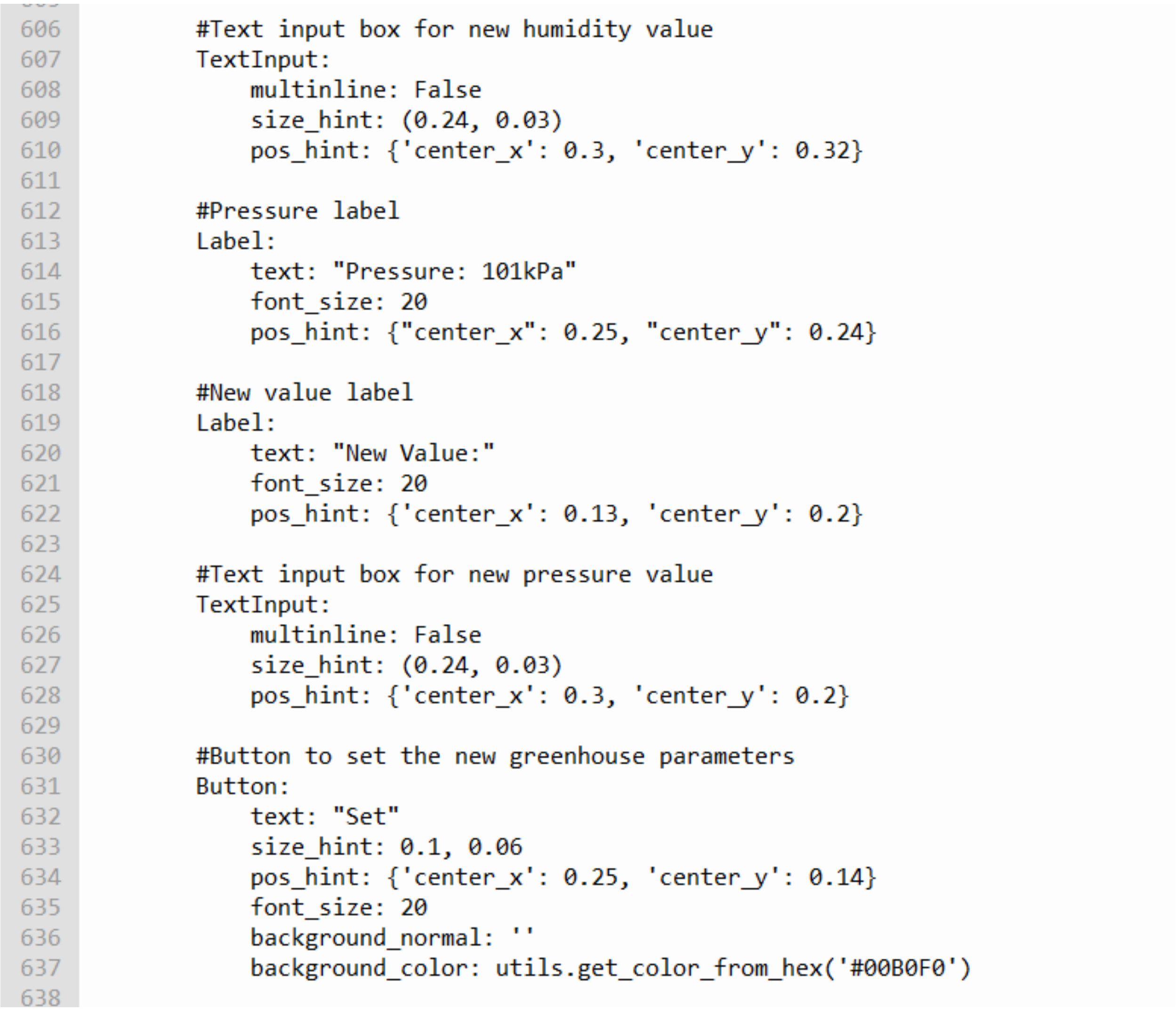
Complete parameters code



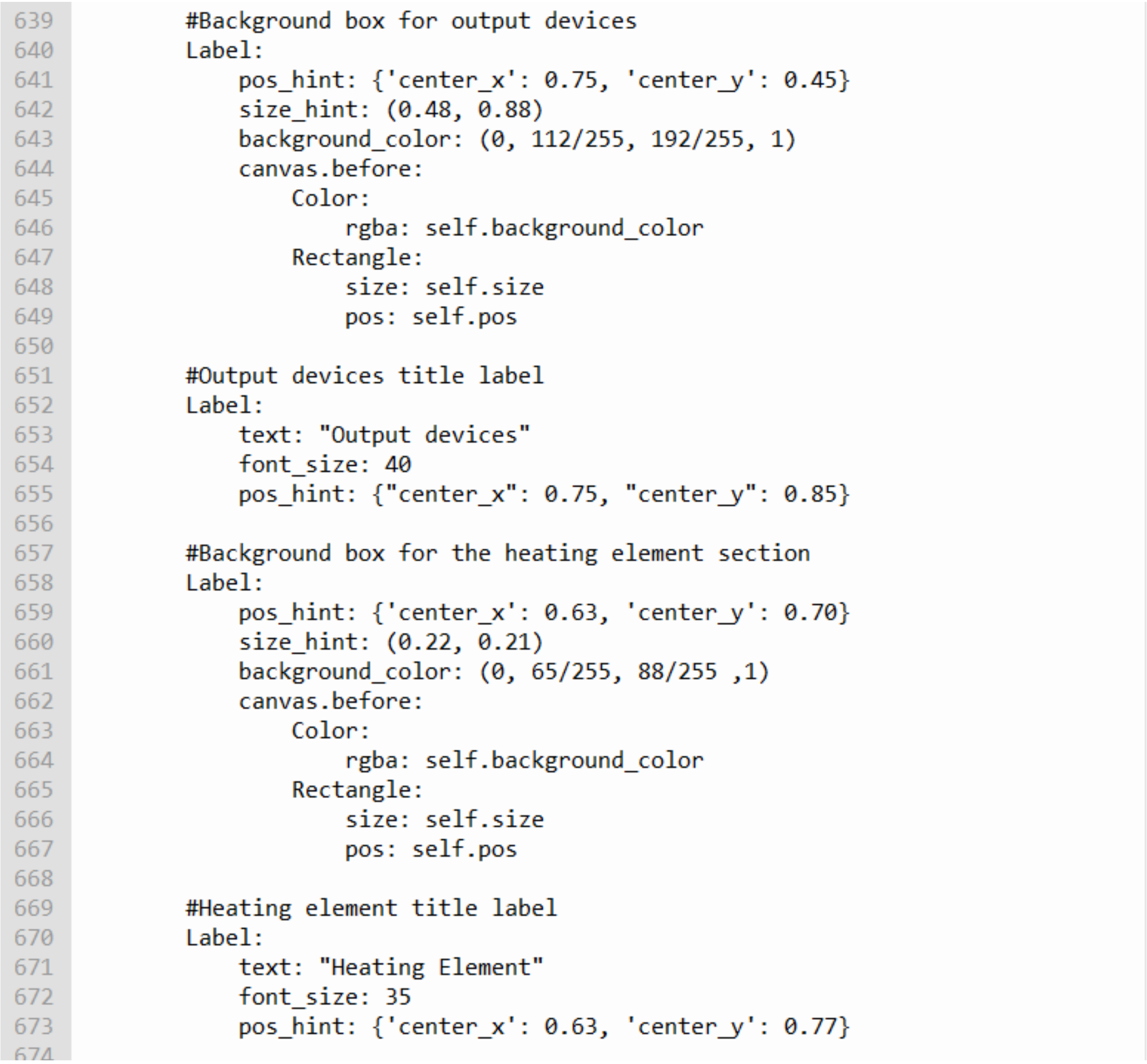
```
<ParametersScreen>
    FloatLayout:
        #Setting the background colour to green
        canvas:
            Color:
                rgba: 0, 0.69, 0.31, 1
            Rectangle:
                pos: (0,0)
                size: self.width, self.height

        #Menu
        #Main Menu button
        Button:
            text: "Main Menu"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.125, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
            on_press: root.manager.current = "mainMenu"

        #Parameters button
        Button:
            text: "Parameters"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.375, 'center_y': 0.95}
            font_size: 60
            background_normal: ''

            #Background color of this button is dark blue as this is the current
            #page
            background_color: (0, 65/255, 88/255 ,1)

            on_press: root.manager.current = "parameters"
```



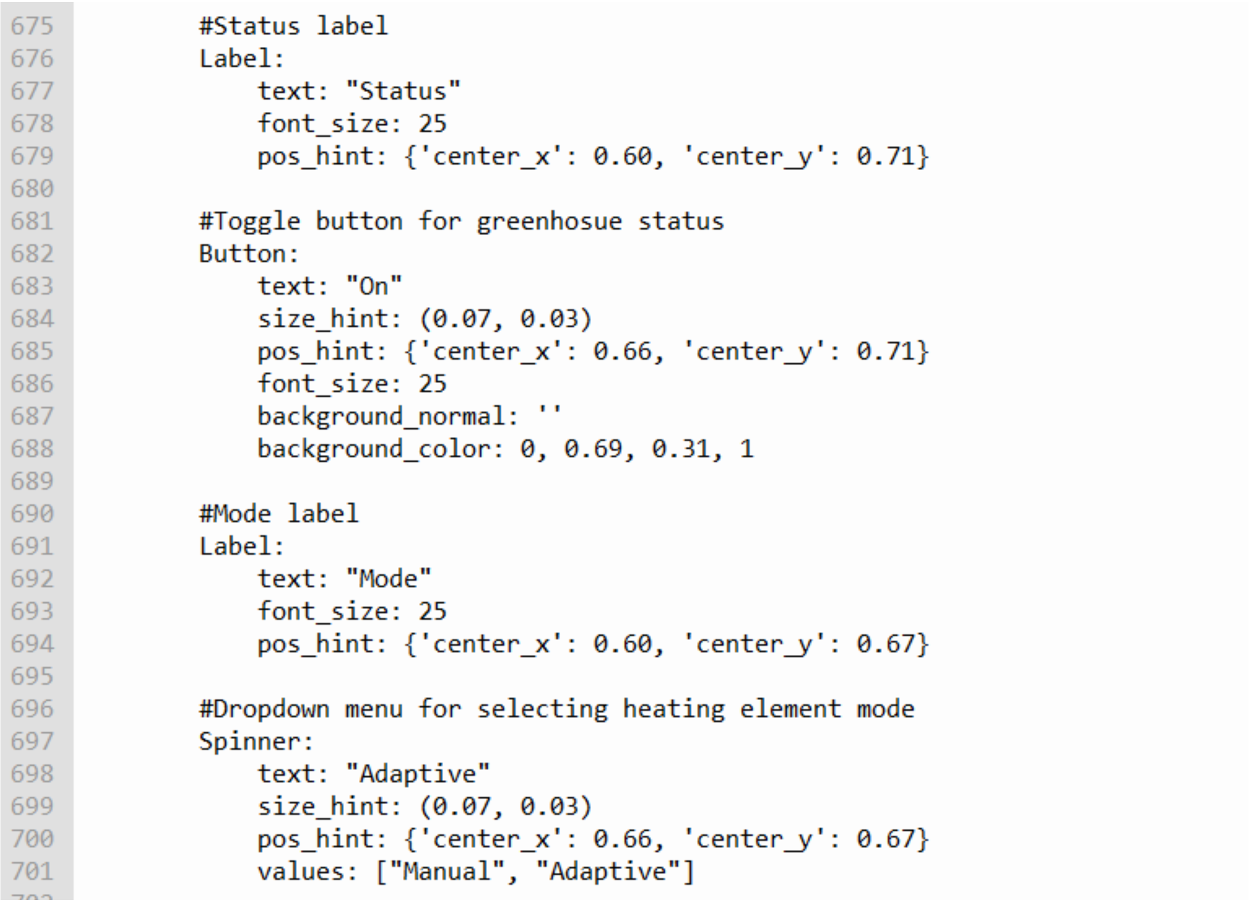
```
        #Graphs button
        Button:
            text: "Graphs"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.625, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
            on_press: root.manager.current = "graphs"

        #Settings button
        Button:
            text: "Settings"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.875, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
            on_press: root.manager.current = "settings"

        #Background box for the greenhouse parameters
        Label:
            pos_hint: {'center_x': 0.25, 'center_y': 0.45}
            size_hint: (0.48, 0.7)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos
        #Greenhouse parameters label
        Label:
            text: "Greenhouse Parameters"
            font_size: 40
            pos_hint: {"center_x": 0.25, "center_y": 0.76}

        #Internal temperature label
        Label:
            text: "Internal Temperature: 20 \N{DEGREE SIGN}C"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.72}

        #New value label
        Label:
            text: "New Value:"
            font_size: 20
            pos_hint: {'center_x': 0.13, 'center_y': 0.68}

        #Text input box for new internal temperature
        TextInput:
            multinline: False
            size_hint: (0.24, 0.03)
            pos_hint: {'center_x': 0.3, 'center_y': 0.68}

        #Label for soil moisutre
        Label:
            text: "Soil Moisture level: 75%"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.60}

        #New value label
        Label:
            text: "New Value:"
            font_size: 20
            pos_hint: {'center_x': 0.13, 'center_y': 0.56}
        #Text input box for new soil mositure
        TextInput:
            multinline: False
            size_hint: (0.24, 0.03)
            pos_hint: {'center_x': 0.3, 'center_y': 0.56}

        #Light intensity label
        Label:
            text: "Light intensity: 1000 lumens"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.48}

        #New value label
        Label:
            text: "New Value:"
            font_size: 20
            pos_hint: {'center_x': 0.13, 'center_y': 0.44}

        #Text input box for new light intensity value
        TextInput:
            multinline: False
            size_hint: (0.24, 0.03)
            pos_hint: {'center_x': 0.3, 'center_y': 0.44}

        #Humidity label
        Label:
            text: "Humidity: 75%"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.36}

        #New value label
        Label:
            text: "New Value:"
            font_size: 20
            pos_hint: {'center_x': 0.13, 'center_y': 0.32}
```



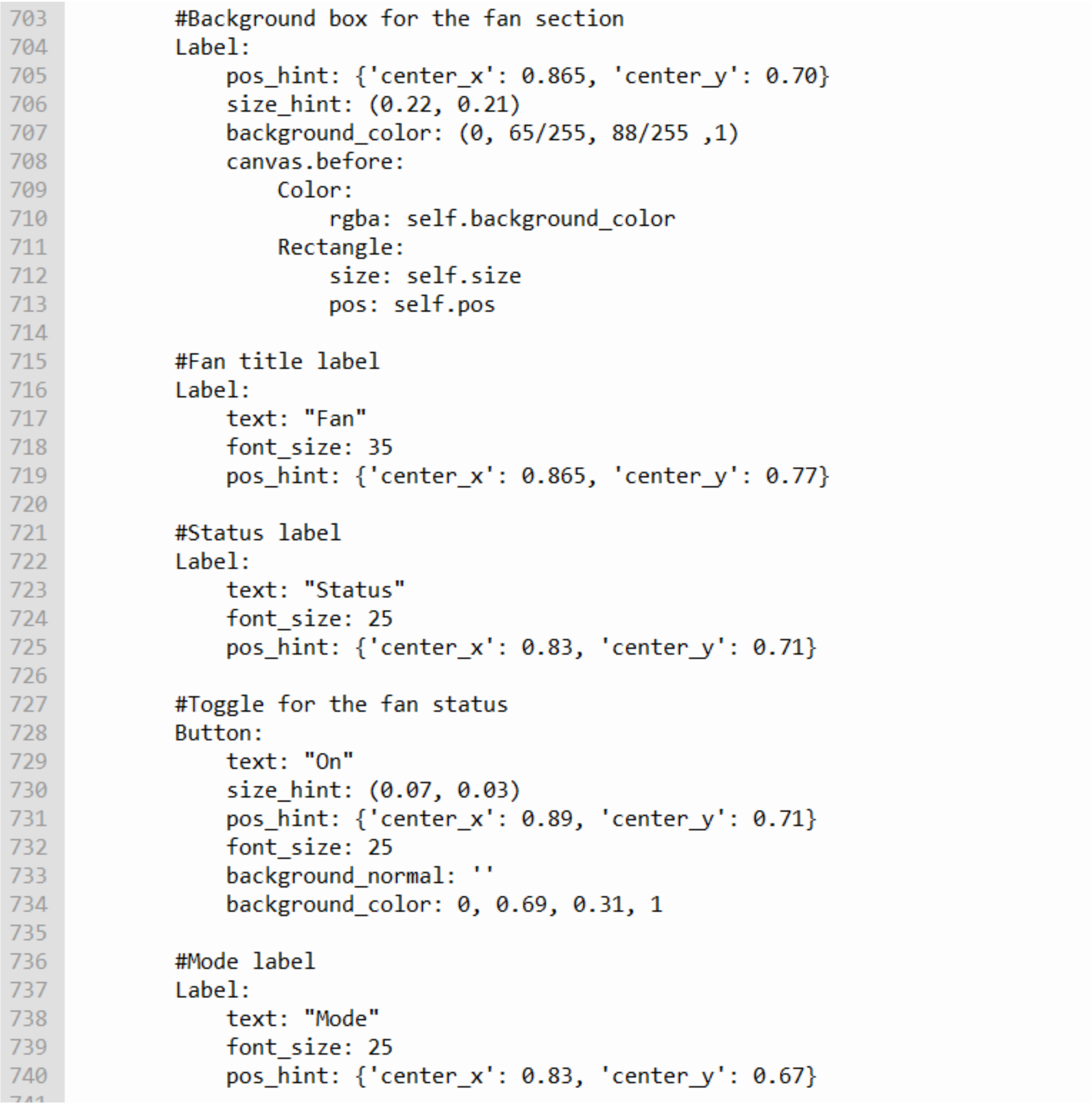
```
        #Text input box for new humidity value
        TextInput:
            multinline: False
            size_hint: (0.24, 0.03)
            pos_hint: {'center_x': 0.3, 'center_y': 0.32}

        #Pressure label
        Label:
            text: "Pressure: 101kPa"
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.24}

        #New value label
        Label:
            text: "New Value:"
            font_size: 20
            pos_hint: {'center_x': 0.13, 'center_y': 0.2}

        #Text input box for new pressure value
        TextInput:
            multinline: False
            size_hint: (0.24, 0.03)
            pos_hint: {'center_x': 0.3, 'center_y': 0.2}

        #Button to set the new greenhouse parameters
        Button:
            text: "Set"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.25, 'center_y': 0.14}
            font_size: 20
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

```



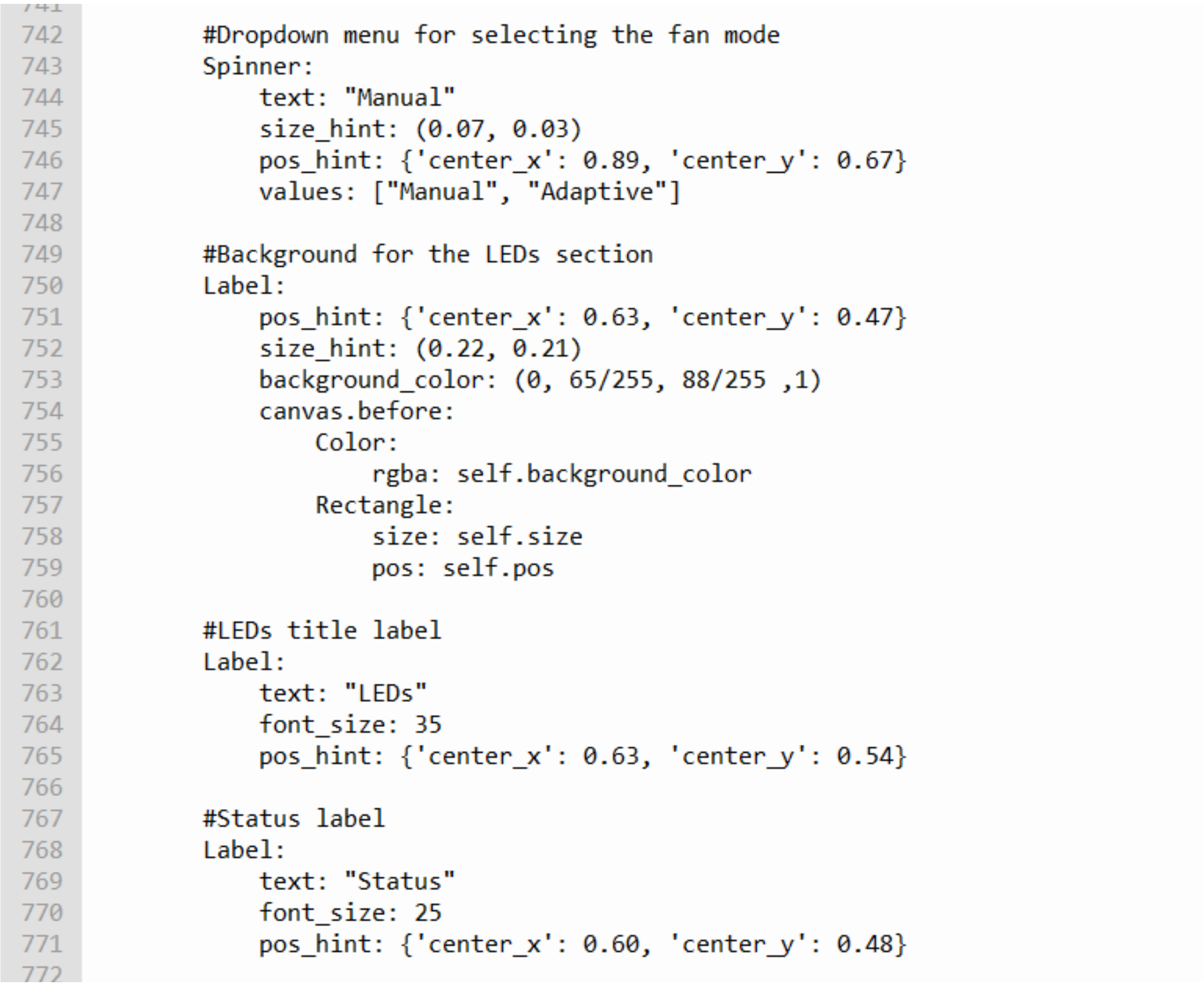
```
        #Background box for output devices
        Label:
            pos_hint: {'center_x': 0.75, 'center_y': 0.45}
            size_hint: (0.48, 0.88)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Output devices title label
        Label:
            text: "Output devices"
            font_size: 40
            pos_hint: {"center_x": 0.75, "center_y": 0.85}

        #Background box for the heating element section
        Label:
            pos_hint: {'center_x': 0.63, 'center_y': 0.70}
            size_hint: (0.22, 0.21)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Heating element title label
        Label:
            text: "Heating Element"
            font_size: 35
            pos_hint: {'center_x': 0.63, 'center_y': 0.77}
```



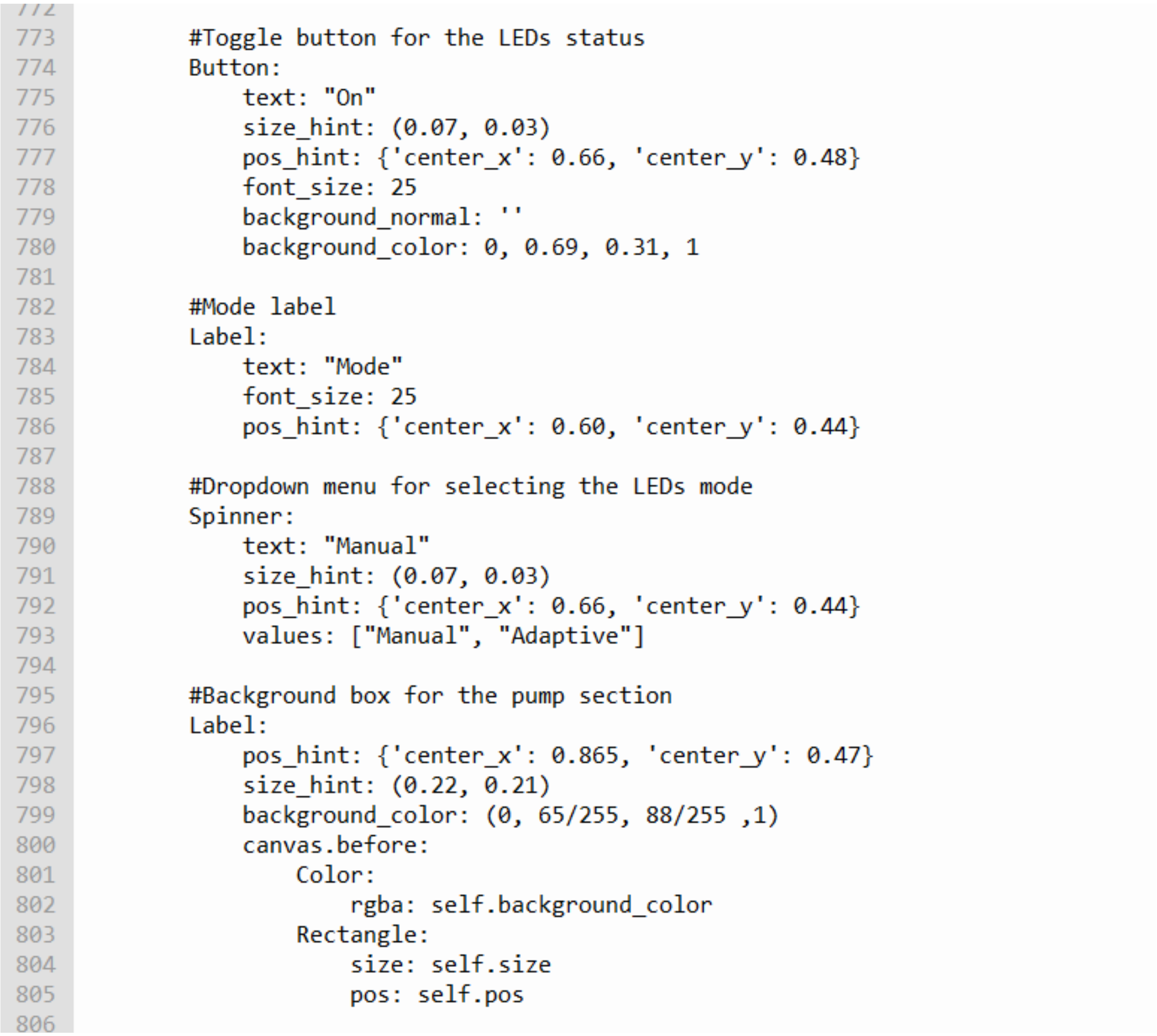
```
        #Status label
        Label:
            text: "Status"
            font_size: 25
            pos_hint: {'center_x': 0.60, 'center_y': 0.71}

        #Toggle button for greenhosue status
        Button:
            text: "On"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.66, 'center_y': 0.71}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1

        #Mode label
        Label:
            text: "Mode"
            font_size: 25
            pos_hint: {'center_x': 0.60, 'center_y': 0.67}

        #Dropdown menu for selecting heating element mode
        Spinner:
            text: "Adaptive"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.66, 'center_y': 0.67}
            values: ["Manual", "Adaptive"]
```



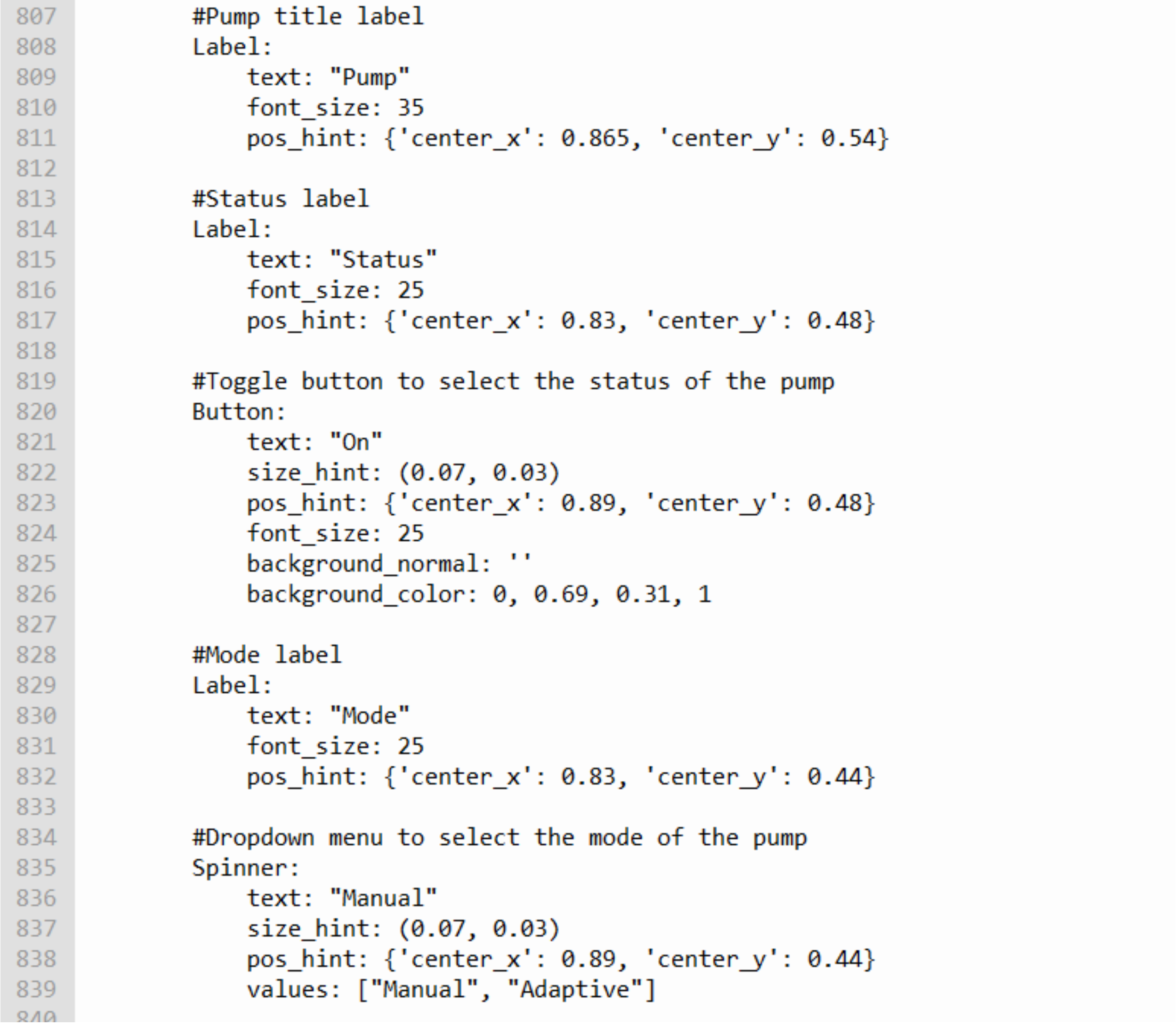
```
        #Background box for the fan section
        Label:
            pos_hint: {'center_x': 0.865, 'center_y': 0.70}
            size_hint: (0.22, 0.21)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Fan title label
        Label:
            text: "Fan"
            font_size: 35
            pos_hint: {'center_x': 0.865, 'center_y': 0.77}

        #Status label
        Label:
            text: "Status"
            font_size: 25
            pos_hint: {'center_x': 0.83, 'center_y': 0.71}

        #Toggle for the fan status
        Button:
            text: "On"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.89, 'center_y': 0.71}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1

        #Mode label
        Label:
            text: "Mode"
            font_size: 25
            pos_hint: {'center_x': 0.83, 'center_y': 0.67}
```



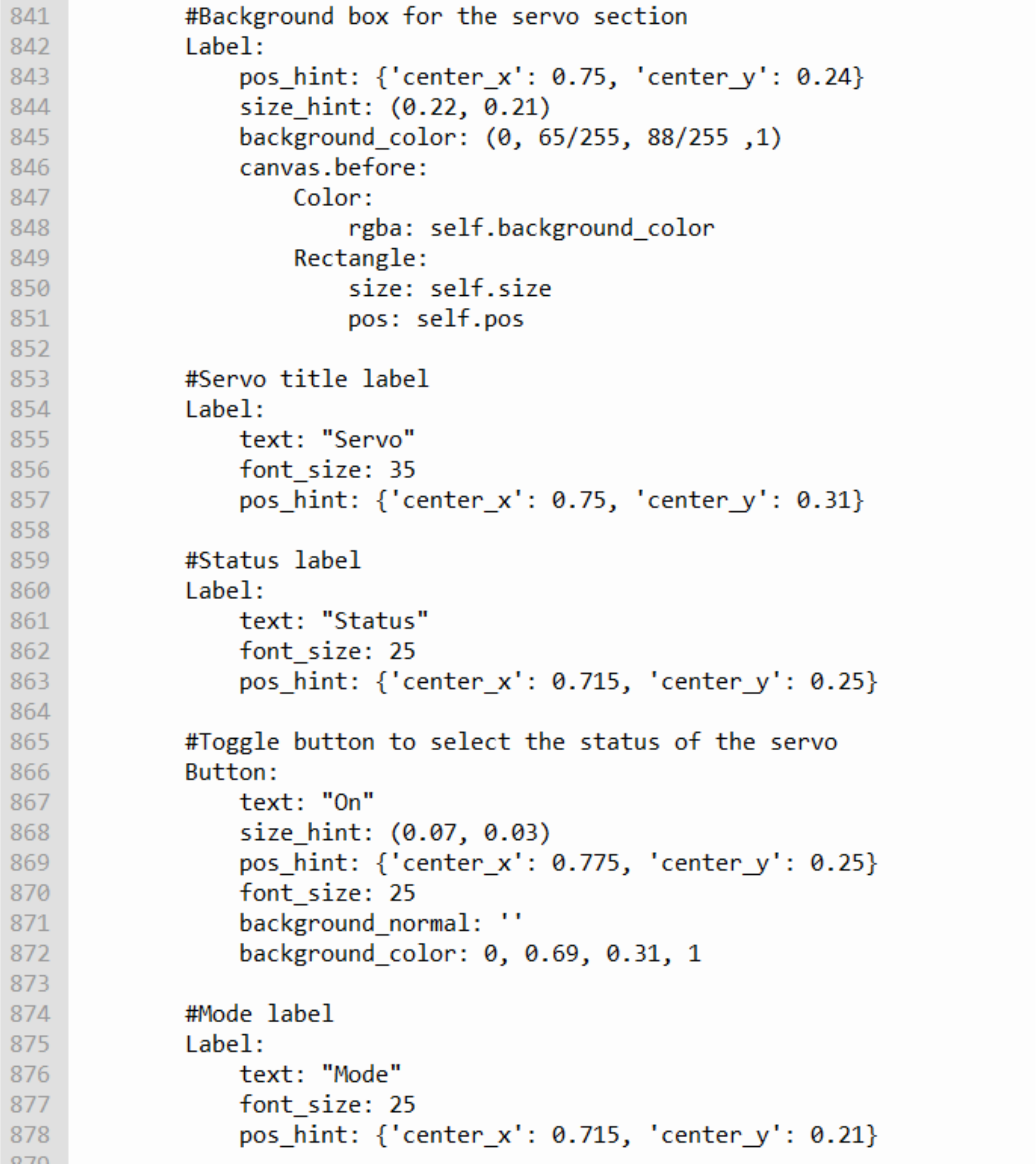
```
        #Dropdown menu for selecting the fan mode
        Spinner:
            text: "Manual"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.89, 'center_y': 0.67}
            values: ["Manual", "Adaptive"]

        #Background for the LEDs section
        Label:
            pos_hint: {'center_x': 0.63, 'center_y': 0.47}
            size_hint: (0.22, 0.21)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #LEDs title label
        Label:
            text: "LEDs"
            font_size: 35
            pos_hint: {'center_x': 0.63, 'center_y': 0.54}

        #Status label
        Label:
            text: "Status"
            font_size: 25
            pos_hint: {'center_x': 0.60, 'center_y': 0.48}
```



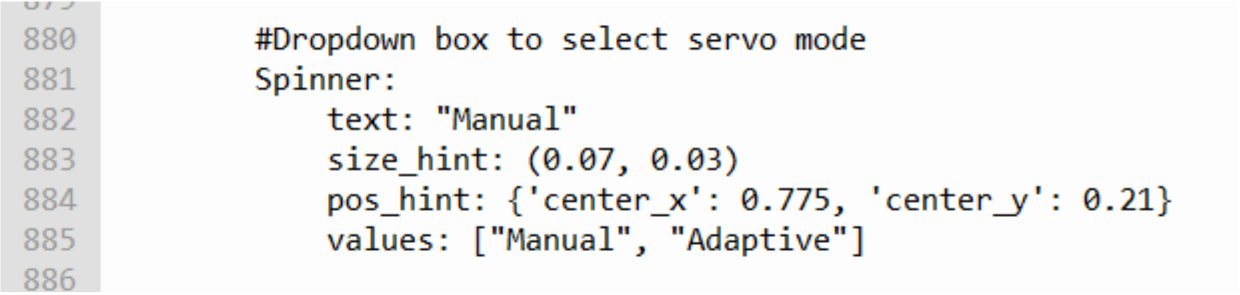
```
        #Toggle button for the LEDs status
        Button:
            text: "On"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.66, 'center_y': 0.48}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1

        #Mode label
        Label:
            text: "Mode"
            font_size: 25
            pos_hint: {'center_x': 0.60, 'center_y': 0.44}

        #Dropdown menu for selecting the LEDs mode
        Spinner:
            text: "Manual"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.66, 'center_y': 0.44}
            values: ["Manual", "Adaptive"]

        #Background box for the pump section
        Label:
            pos_hint: {'center_x': 0.865, 'center_y': 0.47}
            size_hint: (0.22, 0.21)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos
```



```
        #Pump title label
        Label:
            text: "Pump"
            font_size: 35
            pos_hint: {'center_x': 0.865, 'center_y': 0.54}

        #Status label
        Label:
            text: "Status"
            font_size: 25
            pos_hint: {'center_x': 0.83, 'center_y': 0.48}

        #Toggle button to select the status of the pump
        Button:
            text: "On"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.89, 'center_y': 0.48}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1

        #Mode label
        Label:
            text: "Mode"
            font_size: 25
            pos_hint: {'center_x': 0.83, 'center_y': 0.44}

        #Dropdown menu to select the mode of the pump
        Spinner:
            text: "Manual"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.89, 'center_y': 0.44}
            values: ["Manual", "Adaptive"]
```



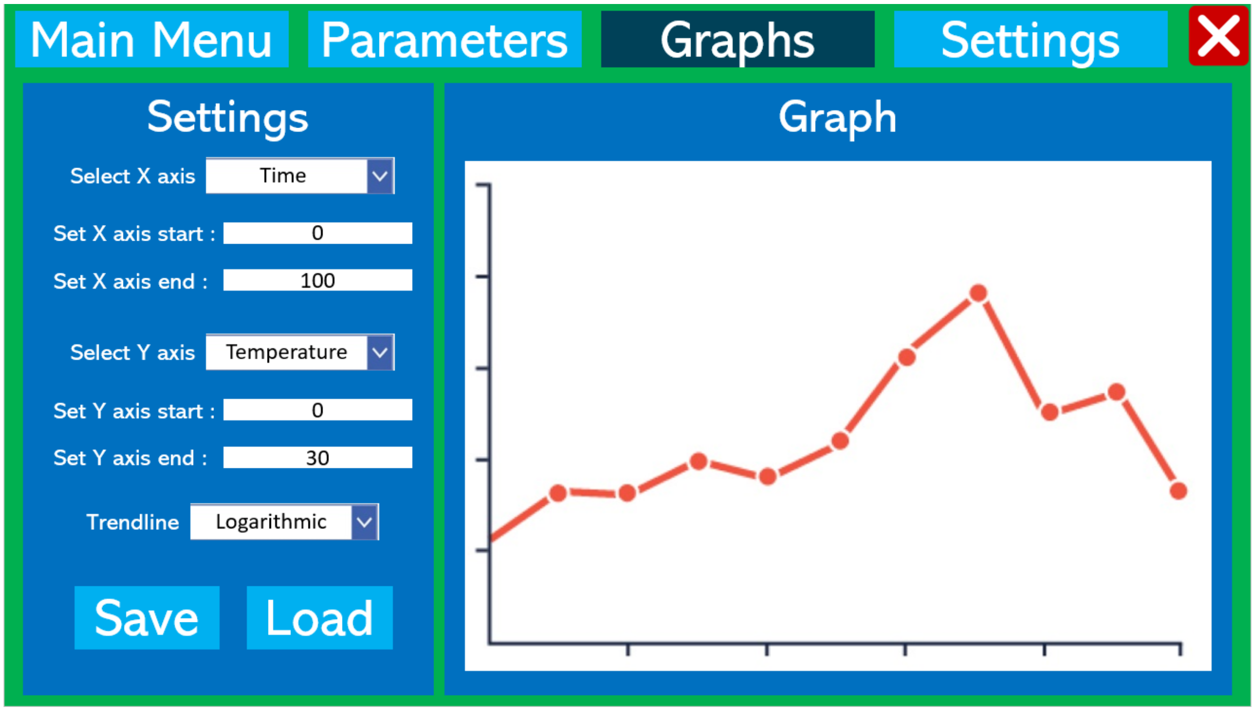
```
        #Background box for the servo section
        Label:
            pos_hint: {'center_x': 0.75, 'center_y': 0.24}
            size_hint: (0.22, 0.21)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Servo title label
        Label:
            text: "Servo"
            font_size: 35
            pos_hint: {'center_x': 0.75, 'center_y': 0.31}

        #Status label
        Label:
            text: "Status"
            font_size: 25
            pos_hint: {'center_x': 0.715, 'center_y': 0.25}

        #Toggle button to select the status of the servo
        Button:
            text: "On"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.775, 'center_y': 0.25}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1

        #Mode label
        Label:
            text: "Mode"
            font_size: 25
            pos_hint: {'center_x': 0.715, 'center_y': 0.21}
```



```
        #Dropdown box to select servo mode
        Spinner:
            text: "Manual"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.775, 'center_y': 0.21}
            values: ["Manual", "Adaptive"]

```

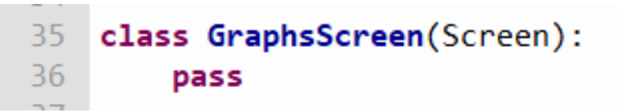
Development log - Graphs screens

The graphs screen will be responsible for allowing the user to produce graphs of the data recorded from the greenhouse. The right-hand side of the screen will be solely for displaying the graph produced and the left side of the screen will feature different dropdown menus and text input boxes to allow the user to adjust the x and y axis ranges and data. There will also be two buttons one to save a graph produced by the user and another to load a graph previously produced by the user. As the graph will be generated and displayed by a different library, I will leave the graph section on the right blank as this will be implemented later.

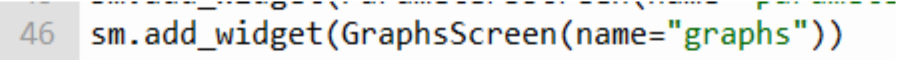


Select X axis: Time
Set X axis start : 0
Set X axis end : 100
Select Y axis: Temperature
Set Y axis start : 0
Set Y axis end : 30
Trendline: Logarithmic

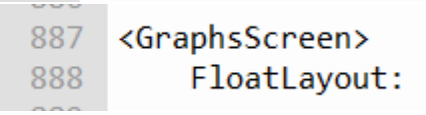
The GUI for the graphs screen has been designed to allow the user to produce meaningful and understandable graphs of the data recorded in the greenhouse. As opposed to viewing raw date which is harder to interpret. This can be implemented in kivy using our previously used labels, buttons, and drop-down menus.



```
class GraphsScreen(Screen):
    pass
```

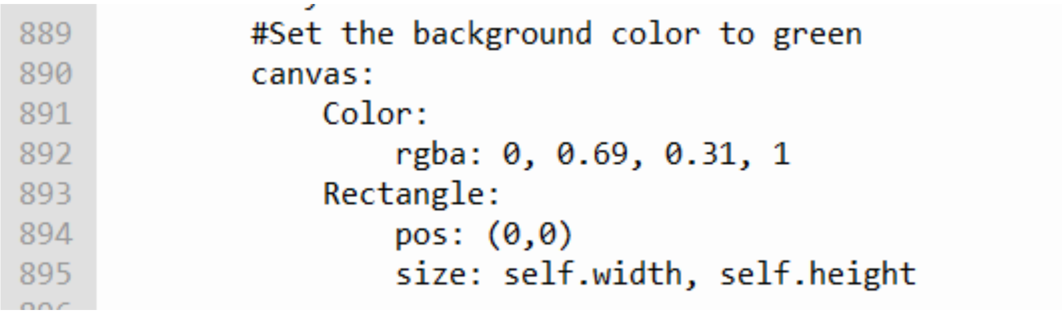


```
sm.add_widget(GraphsScreen(name="graphs"))
```



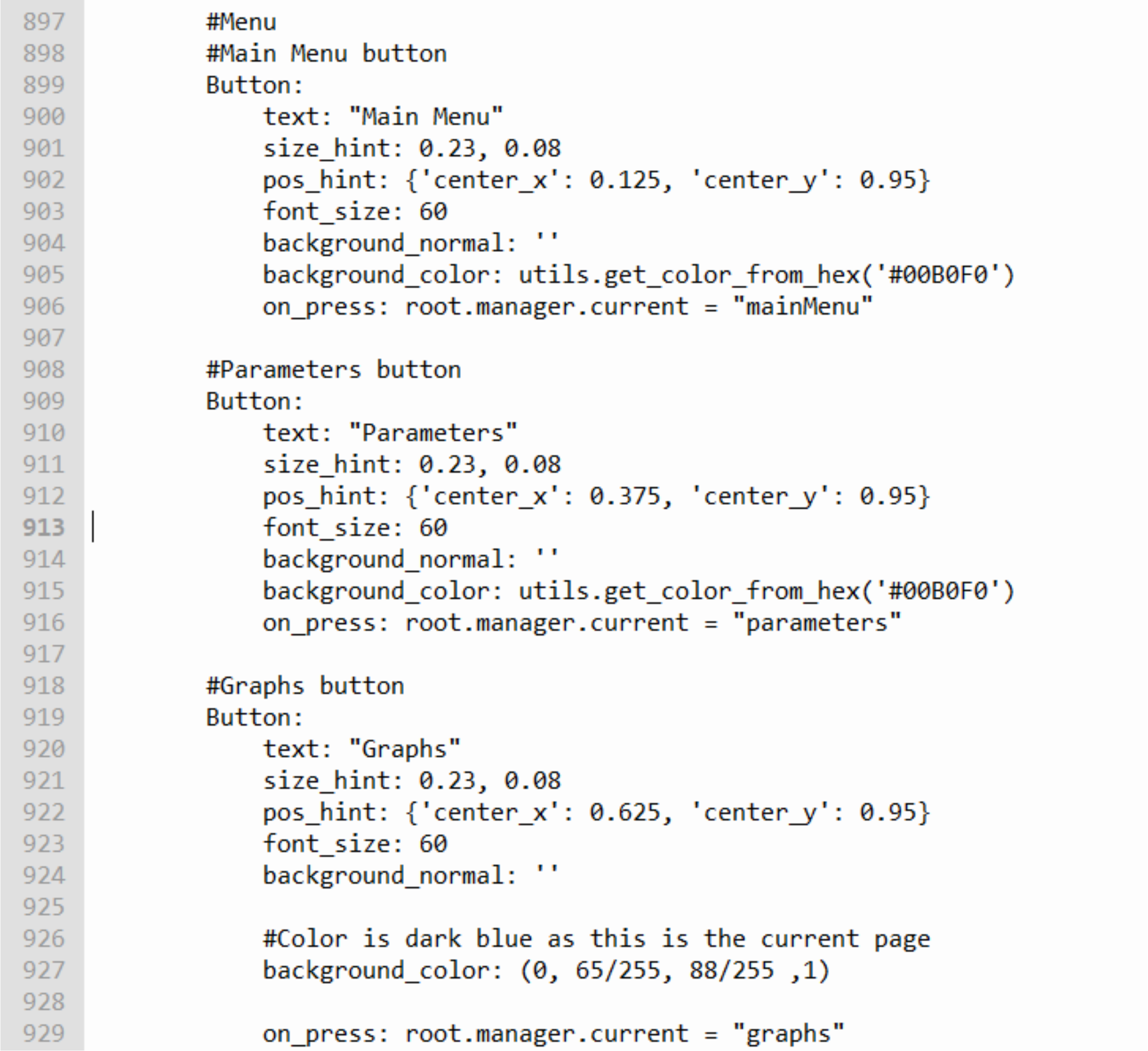
```
<GraphsScreen>
    FloatLayout:
```

I’ve setup the screen inside the python file and the kivy file.

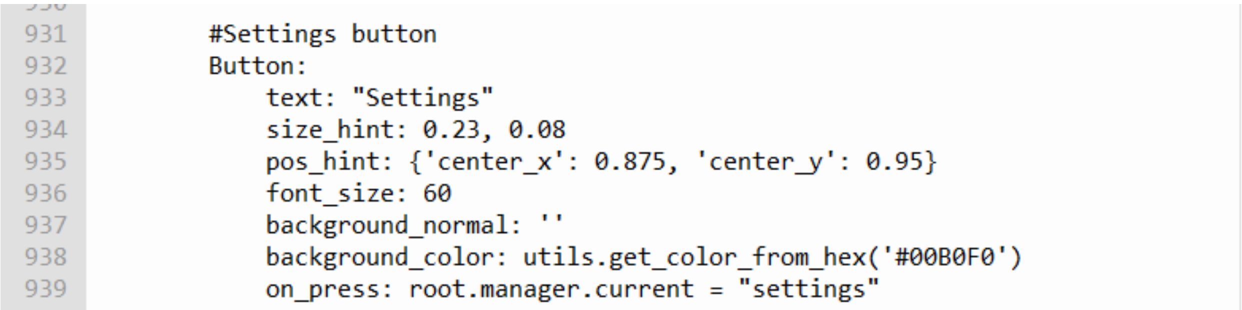


```
        #Set the background color to green
        canvas:
            Color:
                rgba: 0, 0.69, 0.31, 1
            Rectangle:
                pos: (0,0)
                size: self.width, self.height
```

Background color is green.

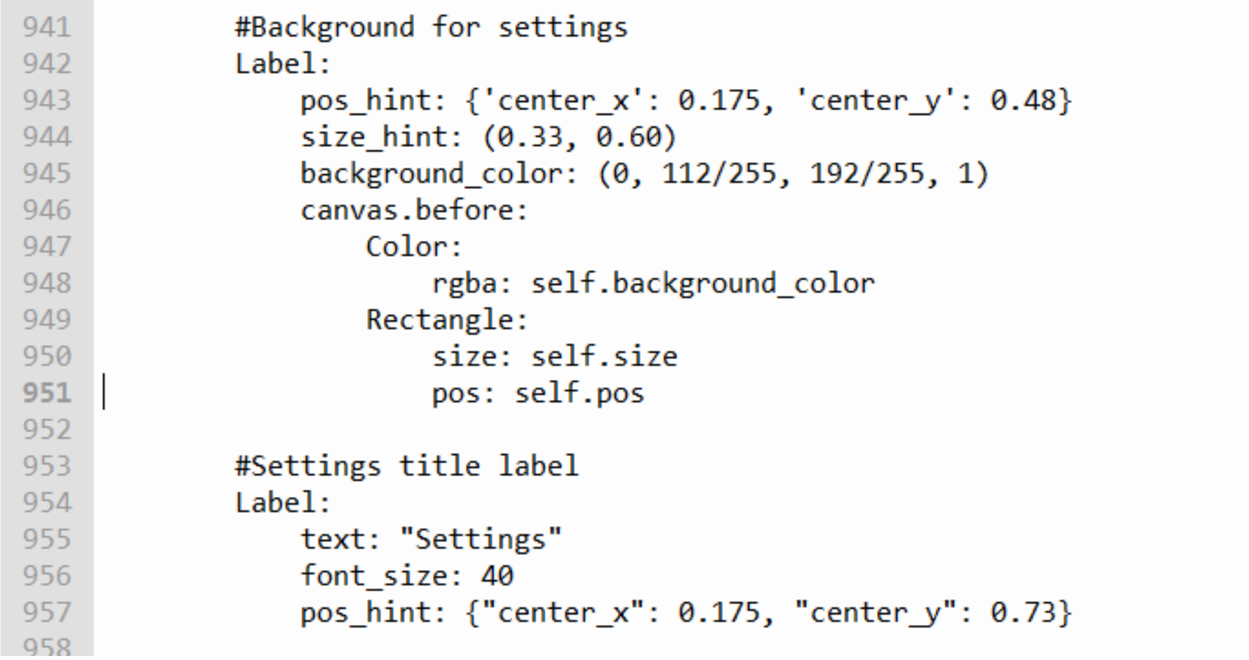


```
        #Menu
        #Main Menu button
        Button:
            text: "Main Menu"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.125, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
            on_press: root.manager.current = "mainMenu"

        #Parameters button
        Button:
            text: "Parameters"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.375, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
            on_press: root.manager.current = "parameters"

        #Graphs button
        Button:
            text: "Graphs"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.625, 'center_y': 0.95}
            font_size: 60
            background_normal: ''

            #Color is dark blue as this is the current page
            background_color: (0, 65/255, 88/255 ,1)

            on_press: root.manager.current = "graphs"
```



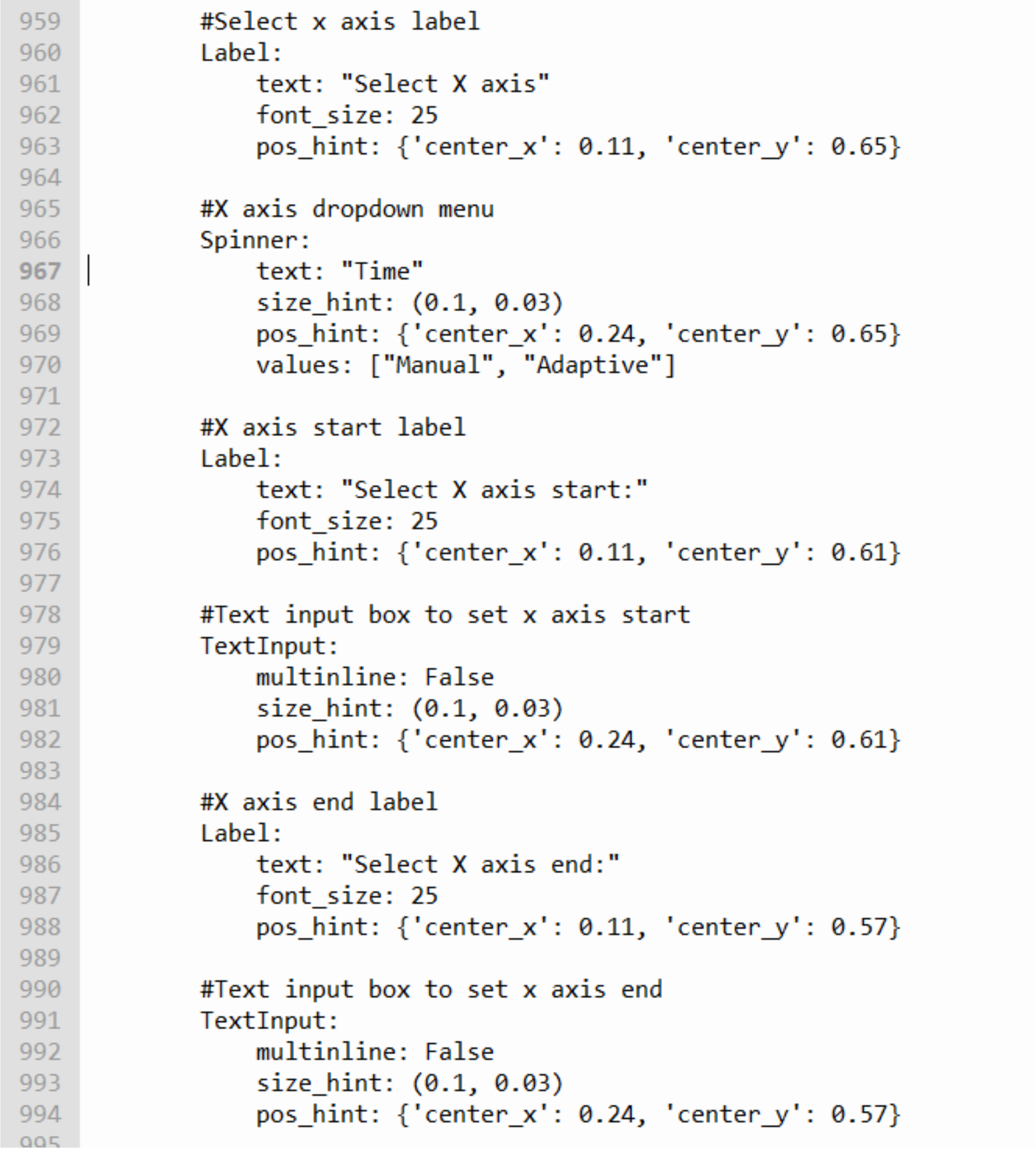
```
        #Settings button
        Button:
            text: "Settings"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.875, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
            on_press: root.manager.current = "settings"
```

Menu is the same as always with the graphs button made dark blue this time.

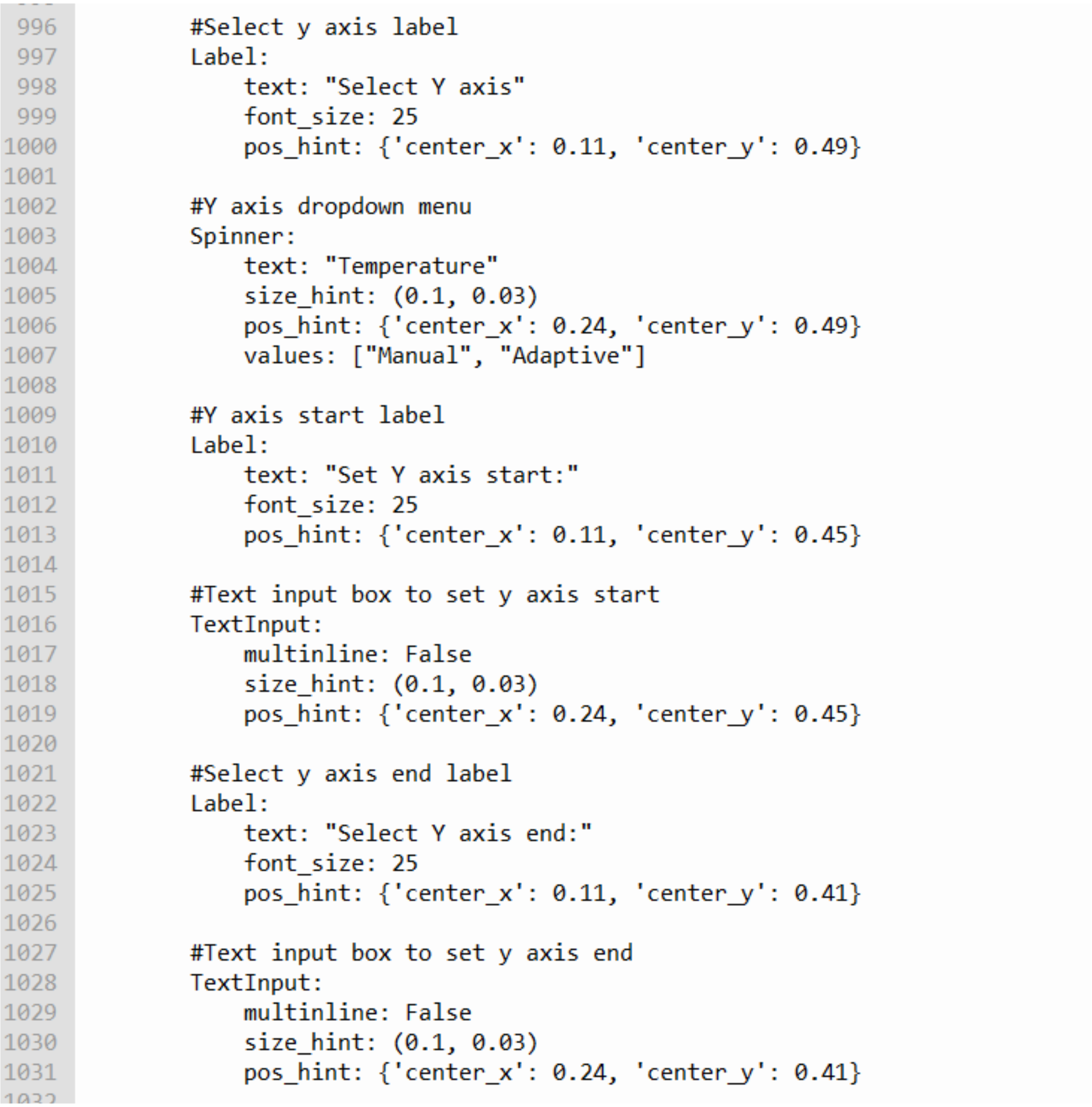


```
        #Background for settings
        Label:
            pos_hint: {'center_x': 0.175, 'center_y': 0.48}
            size_hint: (0.33, 0.60)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Settings title label
        Label:
            text: "Settings"
            font_size: 40
            pos_hint: {"center_x": 0.175, "center_y": 0.73}
```

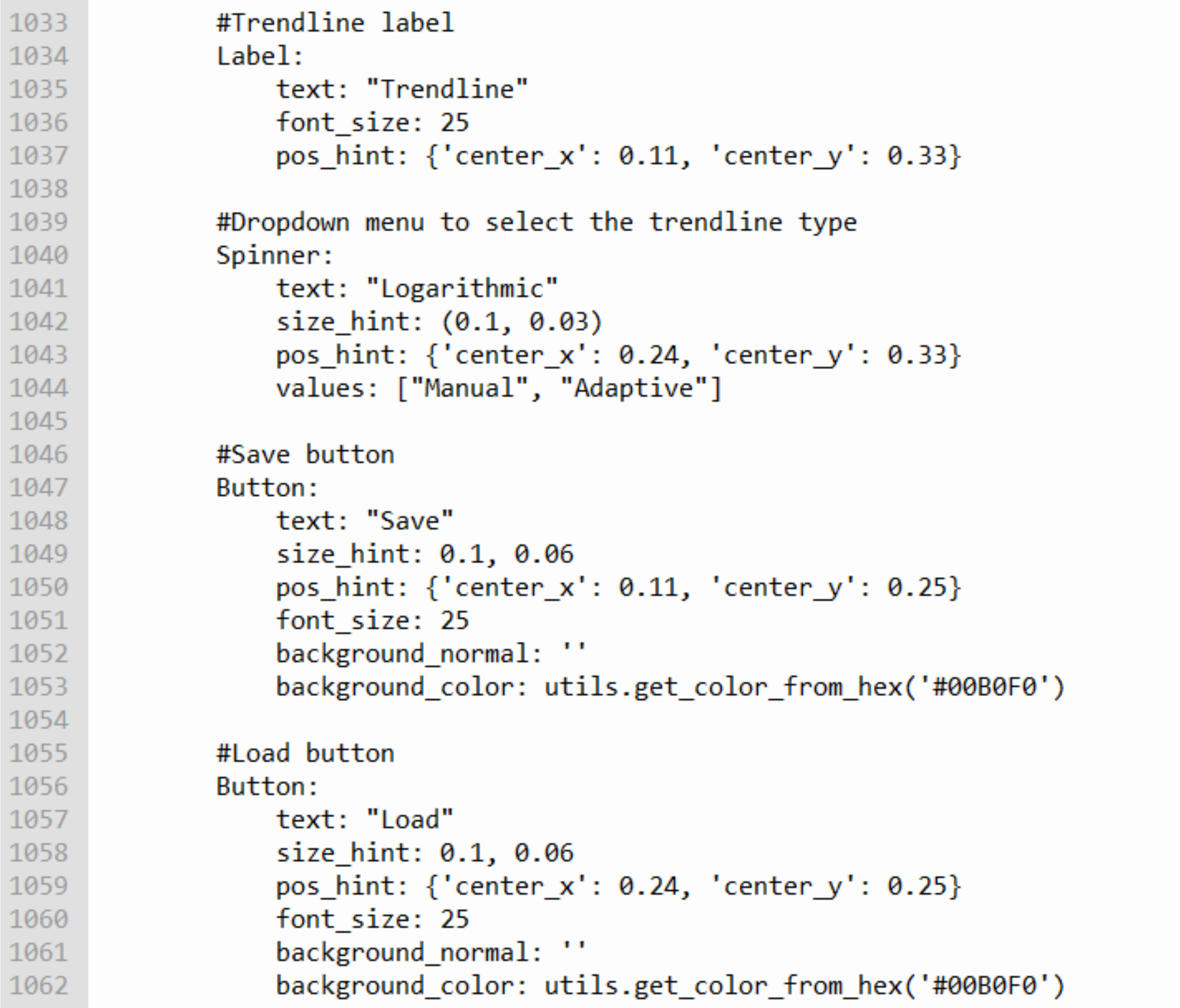
The settings section on the left has a background box and a title.



```
        #Select x axis label
        Label:
            text: "Select X axis"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.65}

        #X axis dropdown menu
        Spinner:
            text: "Time"
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.65}
            values: ["Manual", "Adaptive"]

        #X axis start label
        Label:
            text: "Select X axis start:"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.61}

        #Text input box to set x axis start
        TextInput:
            multinline: False
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.61}

        #X axis end label
        Label:
            text: "Select X axis end:"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.57}

        #Text input box to set x axis end
        TextInput:
            multinline: False
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.57}
```

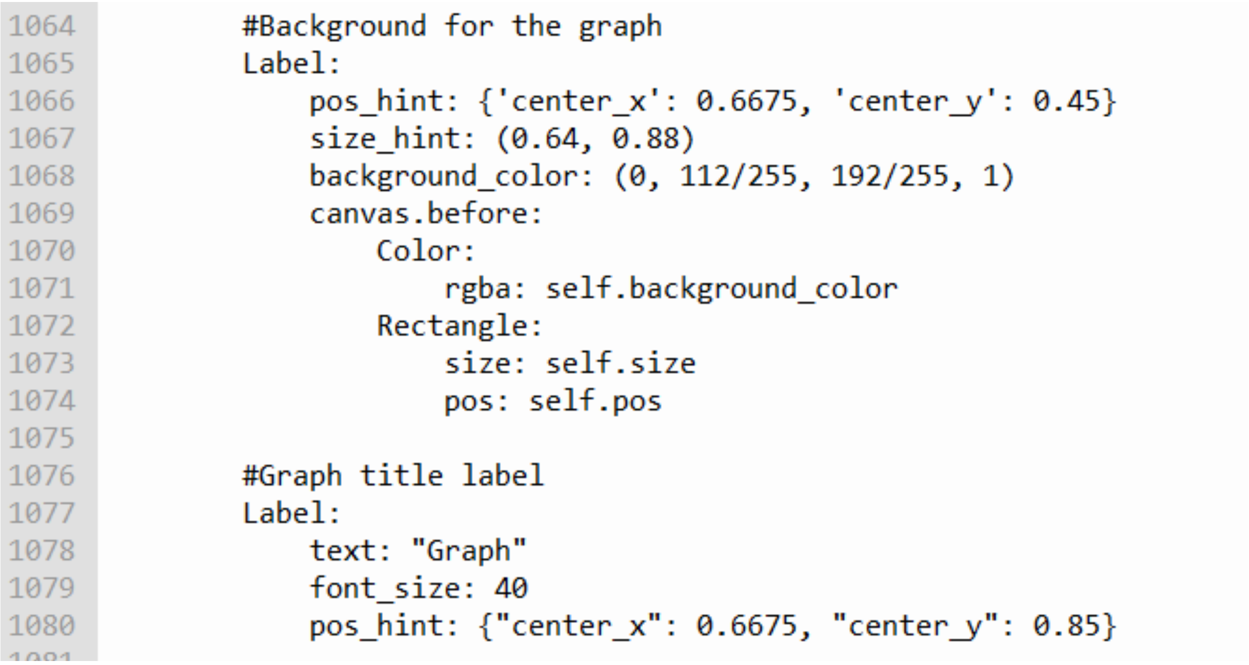


```
        #Select y axis label
        Label:
            text: "Select Y axis"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.49}

        #Y axis dropdown menu
        Spinner:
            text: "Temperature"
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.49}
            values: ["Manual", "Adaptive"]

        #Y axis start label
        Label:
            text: "Set Y axis start:"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.45}

        #Text input box to set y axis start
        TextInput:
            multinline: False
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.45}

        #Select y axis end label
        Label:
            text: "Select Y axis end:"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.41}

        #Text input box to set y axis end
        TextInput:
            multinline: False
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.41}
```



```
        #Trendline label
        Label:
            text: "Trendline"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.33}

        #Dropdown menu to select the trendline type
        Spinner:
            text: "Logarithmic"
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.33}
            values: ["Manual", "Adaptive"]

        #Save button
        Button:
            text: "Save"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.11, 'center_y': 0.25}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

        #Load button
        Button:
            text: "Load"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.24, 'center_y': 0.25}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
```

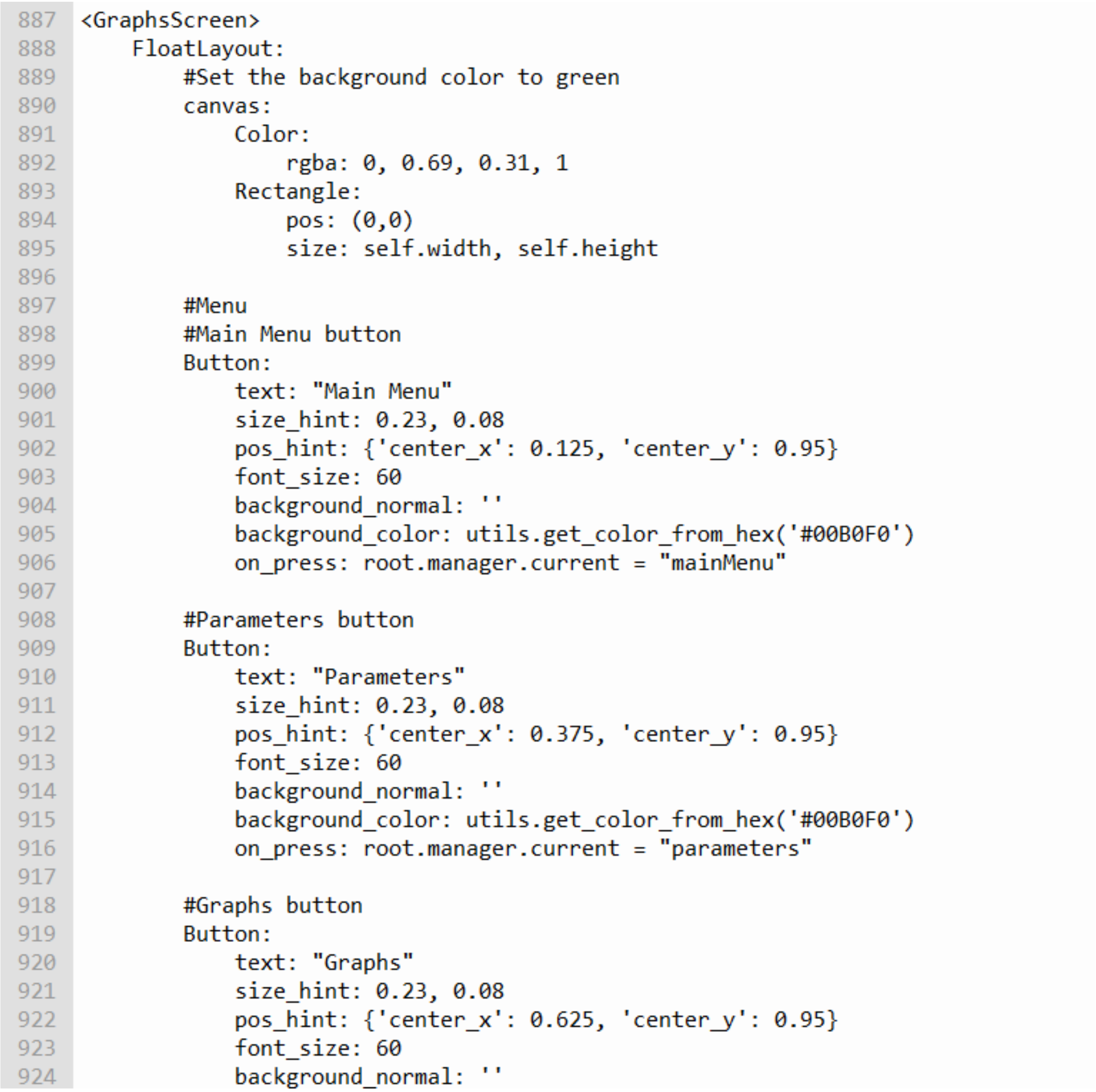
Using labels, dropdown menus, text input boxes and two buttons I have set out all the different options that the user must adjust the graph. The save and load button will later allow the user to save the graph they have generated and too load graphs that they have previously made.



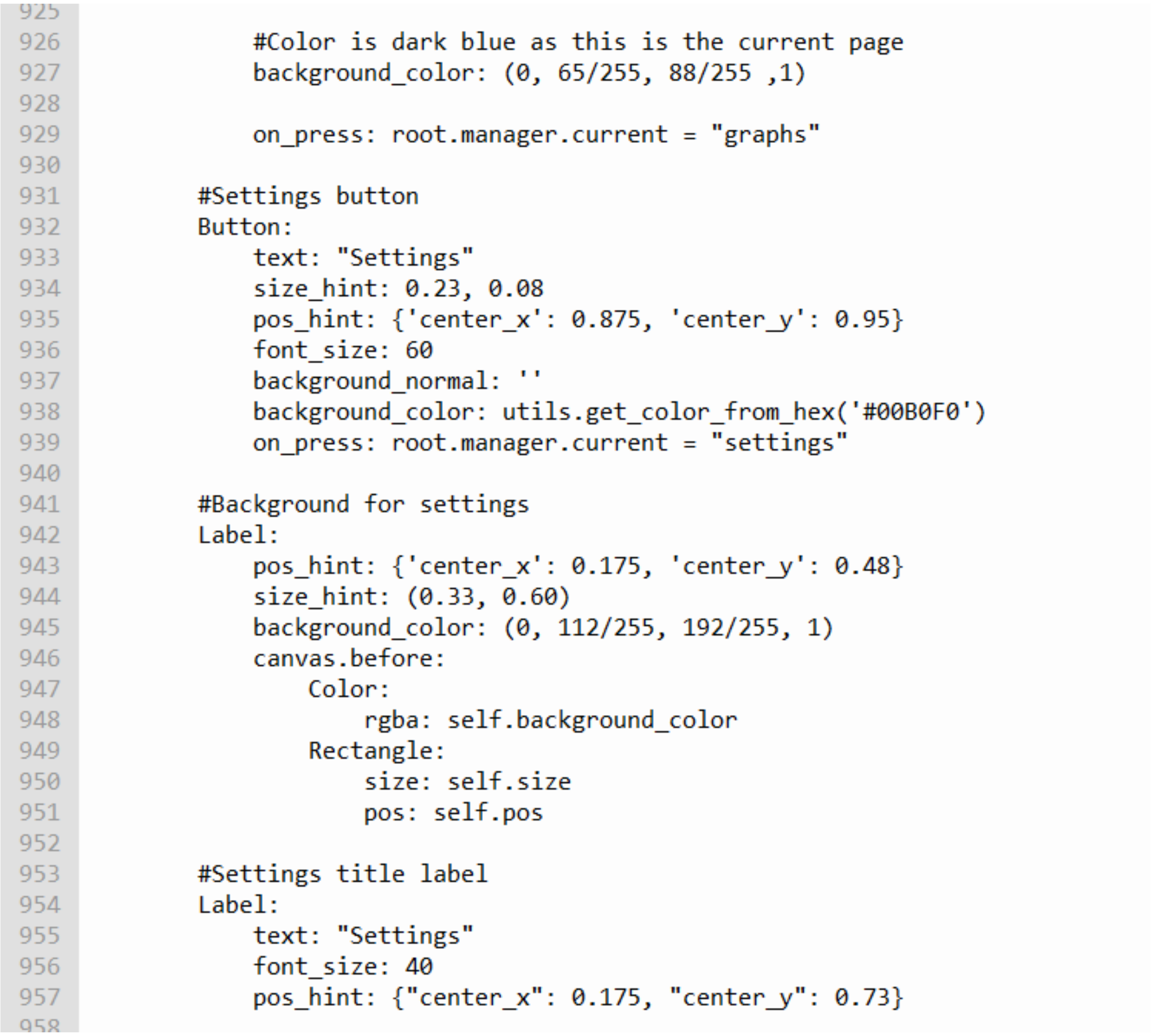
```
        #Background for the graph
        Label:
            pos_hint: {'center_x': 0.6675, 'center_y': 0.45}
            size_hint: (0.64, 0.88)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Graph title label
        Label:
            text: "Graph"
            font_size: 40
            pos_hint: {"center_x": 0.6675, "center_y": 0.85}
```

The right-hand side of the screen will be for the graph to be displayed on. I will be implementing the graph at a later stage of development and so this section just consists of the background box and the title for the time being.

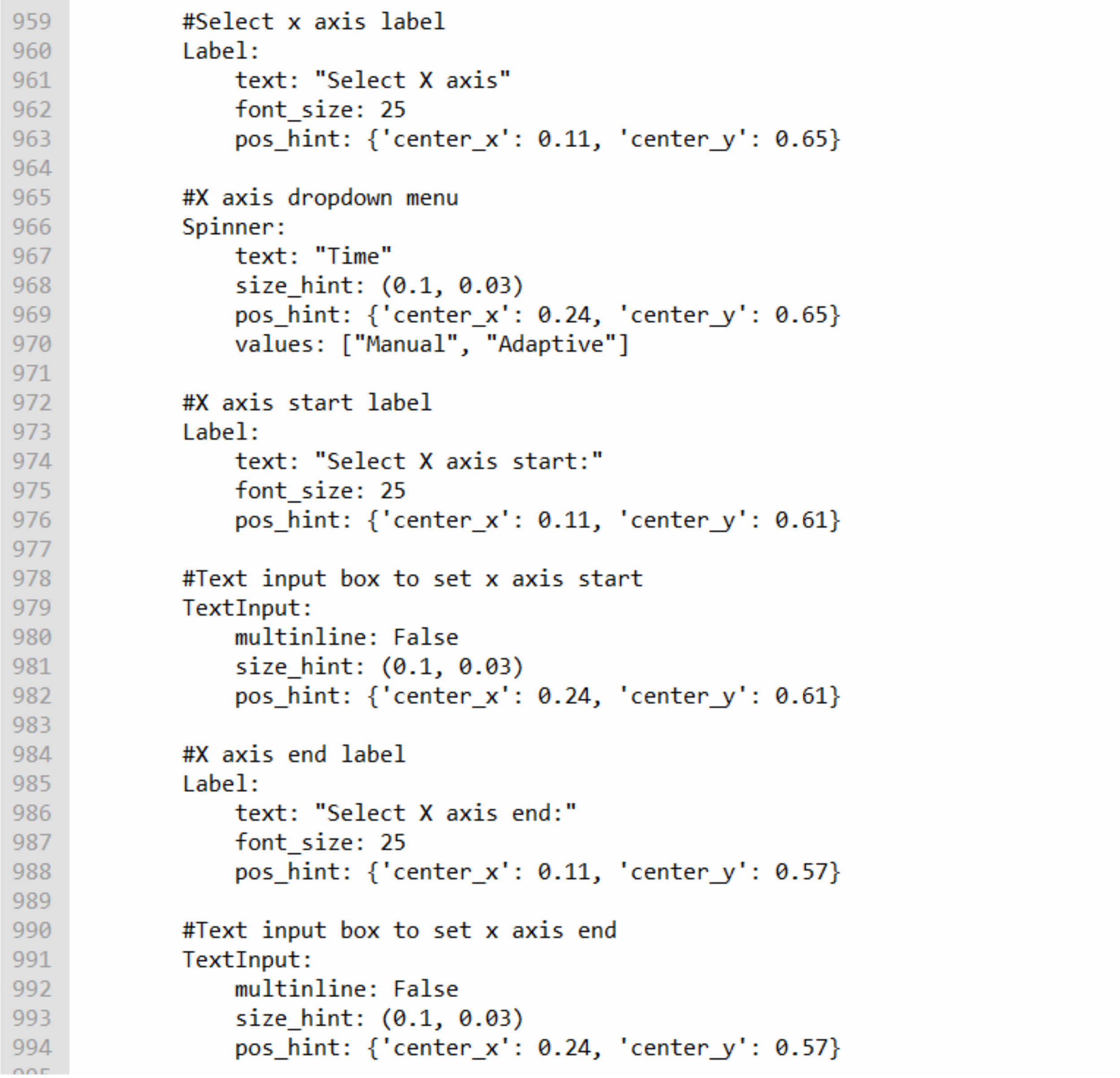
Complete graphs screen code



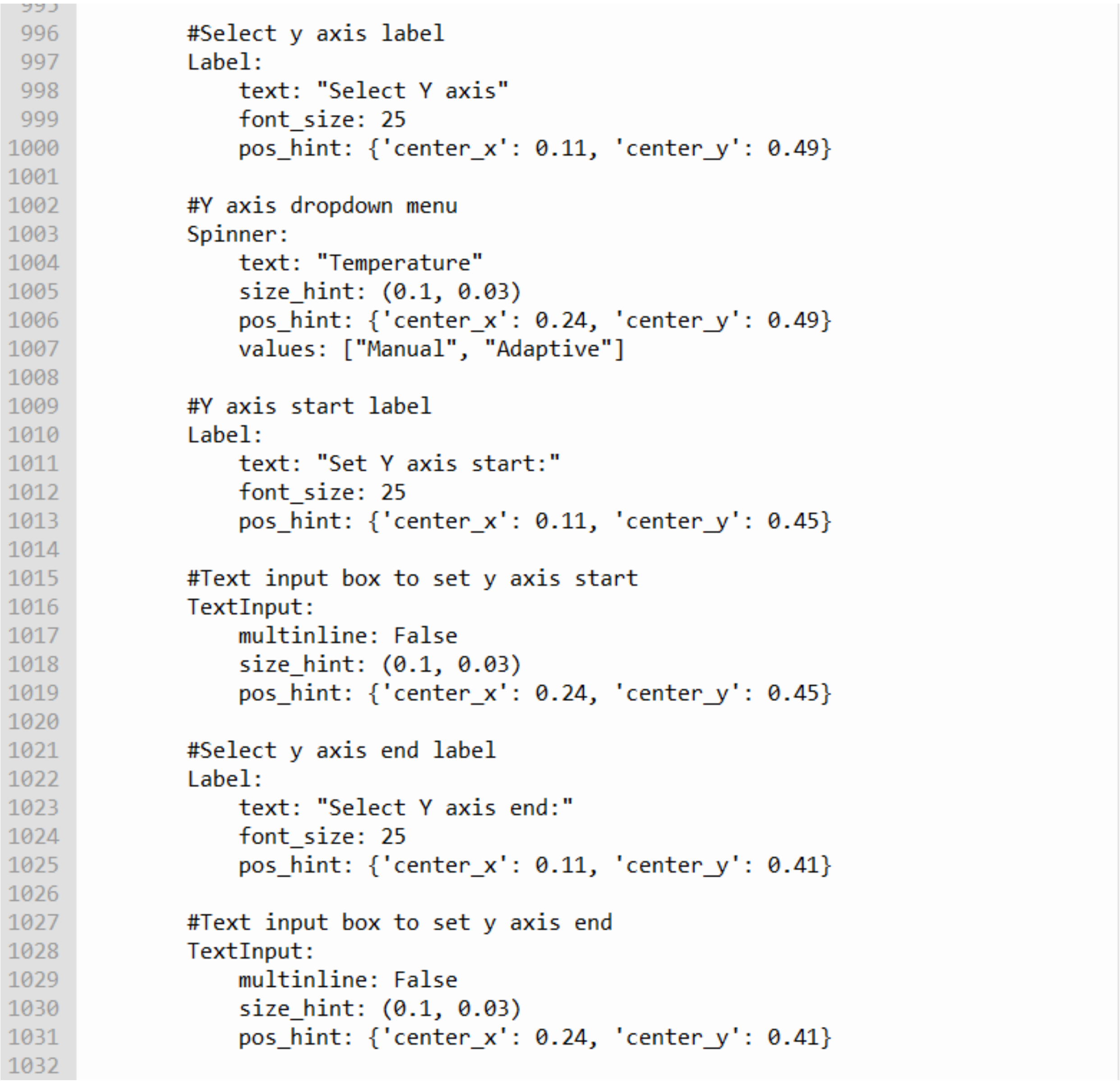
```
<GraphsScreen>
    FloatLayout:
        #Set the background color to green
        canvas:
            Color:
                rgba: 0, 0.69, 0.31, 1
            Rectangle:
                pos: (0,0)
                size: self.width, self.height

        #Menu
        #Main Menu button
        Button:
            text: "Main Menu"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.125, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
            on_press: root.manager.current = "mainMenu"

        #Parameters button
        Button:
            text: "Parameters"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.375, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
            on_press: root.manager.current = "parameters"

        #Graphs button
        Button:
            text: "Graphs"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.625, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
```



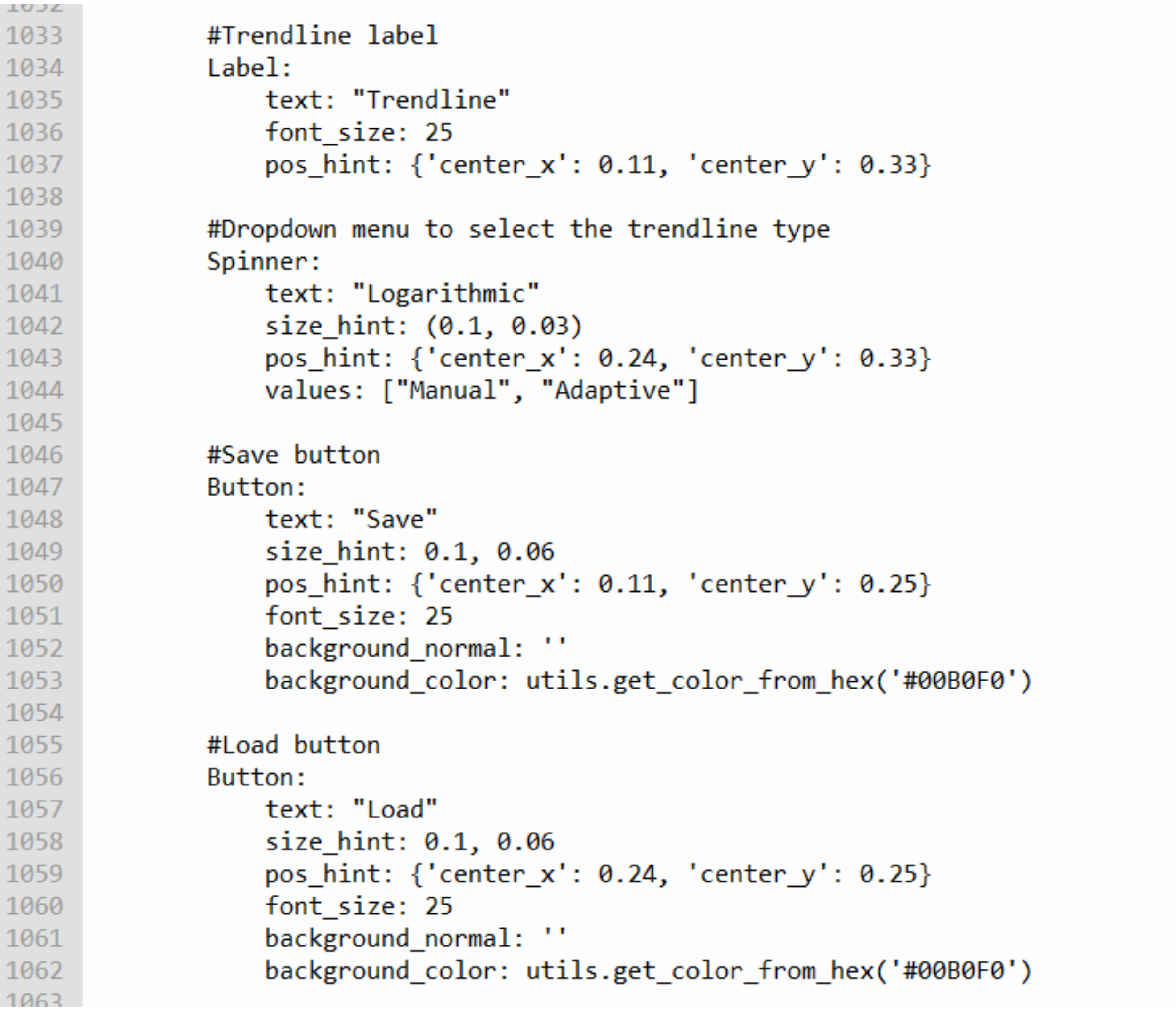
```

            #Color is dark blue as this is the current page
            background_color: (0, 65/255, 88/255 ,1)

            on_press: root.manager.current = "graphs"

        #Settings button
        Button:
            text: "Settings"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.875, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
            on_press: root.manager.current = "settings"

        #Background for settings
        Label:
            pos_hint: {'center_x': 0.175, 'center_y': 0.48}
            size_hint: (0.33, 0.60)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Settings title label
        Label:
            text: "Settings"
            font_size: 40
            pos_hint: {"center_x": 0.175, "center_y": 0.73}

```



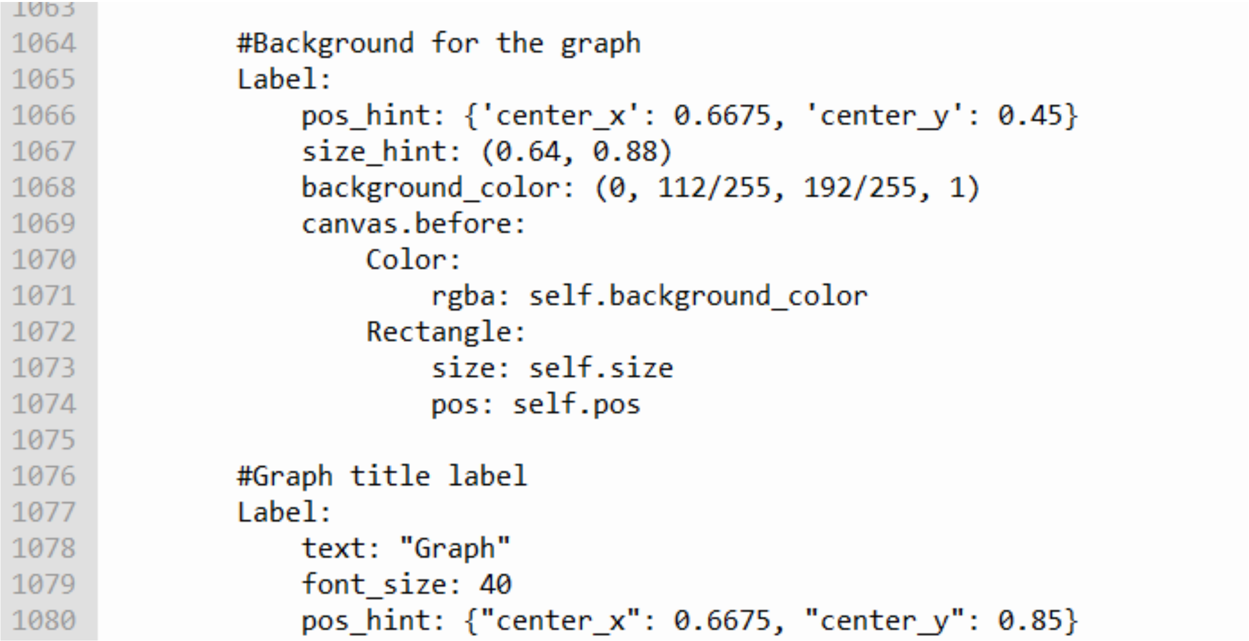
```
        #Select x axis label
        Label:
            text: "Select X axis"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.65}

        #X axis dropdown menu
        Spinner:
            text: "Time"
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.65}
            values: ["Manual", "Adaptive"]

        #X axis start label
        Label:
            text: "Select X axis start:"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.61}

        #Text input box to set x axis start
        TextInput:
            multinline: False
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.61}

        #X axis end label
        Label:
            text: "Select X axis end:"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.57}

        #Text input box to set x axis end
        TextInput:
            multinline: False
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.57}
```



```
        #Select y axis label
        Label:
            text: "Select Y axis"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.49}

        #Y axis dropdown menu
        Spinner:
            text: "Temperature"
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.49}
            values: ["Manual", "Adaptive"]

        #Y axis start label
        Label:
            text: "Set Y axis start:"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.45}

        #Text input box to set y axis start
        TextInput:
            multinline: False
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.45}

        #Select y axis end label
        Label:
            text: "Select Y axis end:"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.41}

        #Text input box to set y axis end
        TextInput:
            multinline: False
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.41}
```



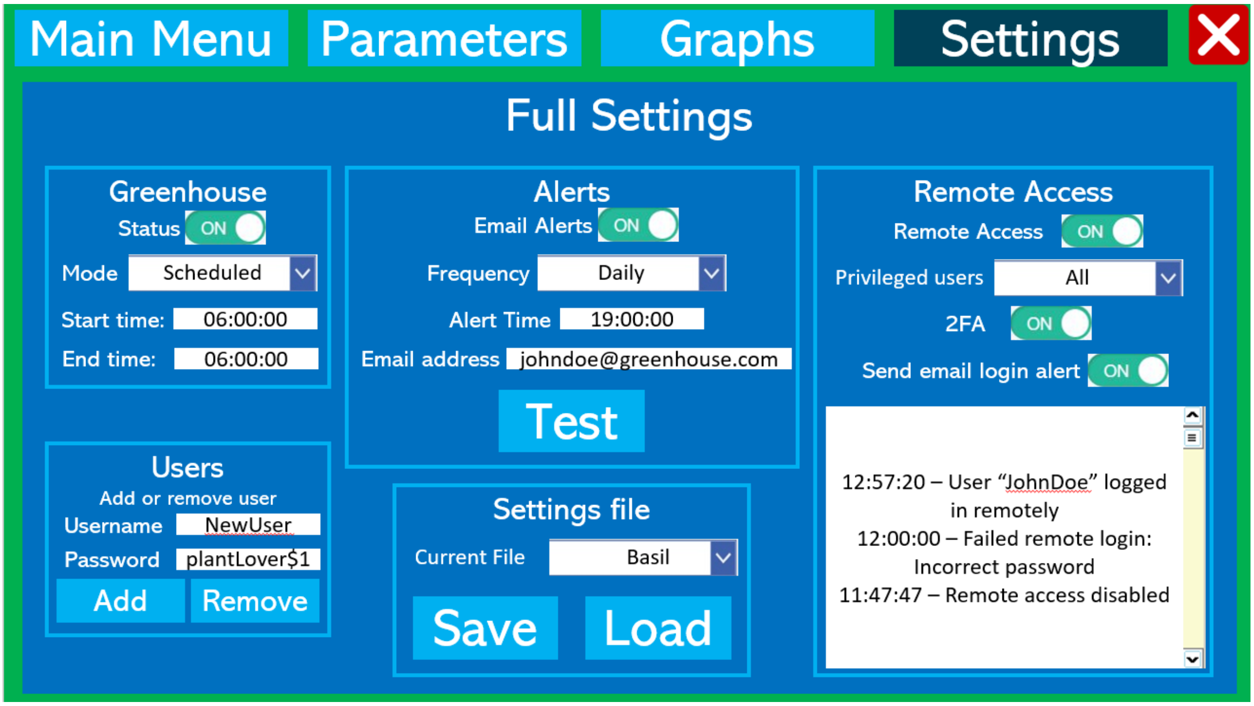
```
        #Trendline label
        Label:
            text: "Trendline"
            font_size: 25
            pos_hint: {'center_x': 0.11, 'center_y': 0.33}

        #Dropdown menu to select the trendline type
        Spinner:
            text: "Logarithmic"
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.24, 'center_y': 0.33}
            values: ["Manual", "Adaptive"]

        #Save button
        Button:
            text: "Save"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.11, 'center_y': 0.25}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

        #Load button
        Button:
            text: "Load"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.24, 'center_y': 0.25}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
```



```
        #Background for the graph
        Label:
            pos_hint: {'center_x': 0.6675, 'center_y': 0.45}
            size_hint: (0.64, 0.88)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Graph title label
        Label:
            text: "Graph"
            font_size: 40
            pos_hint: {"center_x": 0.6675, "center_y": 0.85}
```

Development log – Settings screen

The settings screen is the final screen in the GUI. On this screen the user can control all other settings that have not already been shown on any of the other screens. The screen is split into 5 sections with one section relating to the greenhouse status and scheduling another regarding email alerts and another regarding adding and removing users also a settings file section allowing the user to load in saved settings and finally a remote access section. Due to time constraints, I will not be implementing the remote access feature of the greenhouse and so this section will simply show a label saying, "coming soon".



Status ON
Mode Scheduled
Start time: 06:00:00
End time: 06:00:00

Email Alerts ON
Frequency Daily
Alert Time 19:00:00
Email address johndoe@greenhouse.com

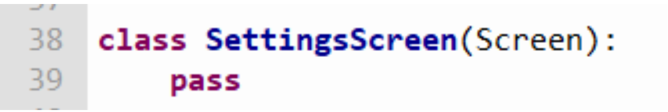
Remote Access ON
Privileged users All
2FA ON
Send email login alert ON

12:57:20 – User "JohnDoe" logged in remotely
12:00:00 – Failed remote login: Incorrect password
11:47:47 – Remote access disabled

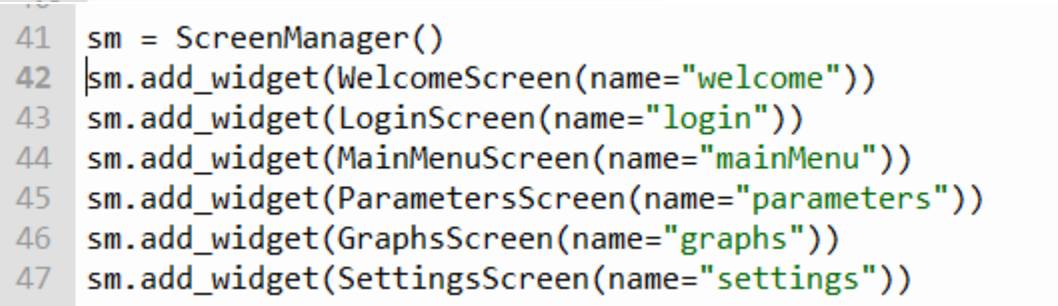
Username NewUser
Password plantLover$1

Current File Basil

Here is the mockup I have made for the settings screen. As with the graphs screen, I will be replacing the blue outlines with a solid dark blue background for the different sections of the screen. The screen does not feature any new elements and so is just a case of positioning different elements and sizing them.

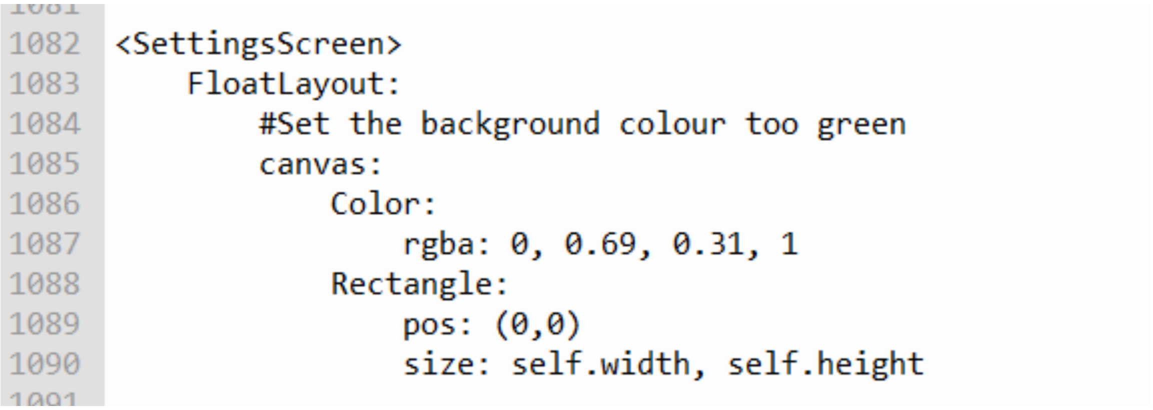


```
class SettingsScreen(Screen):
    pass
```



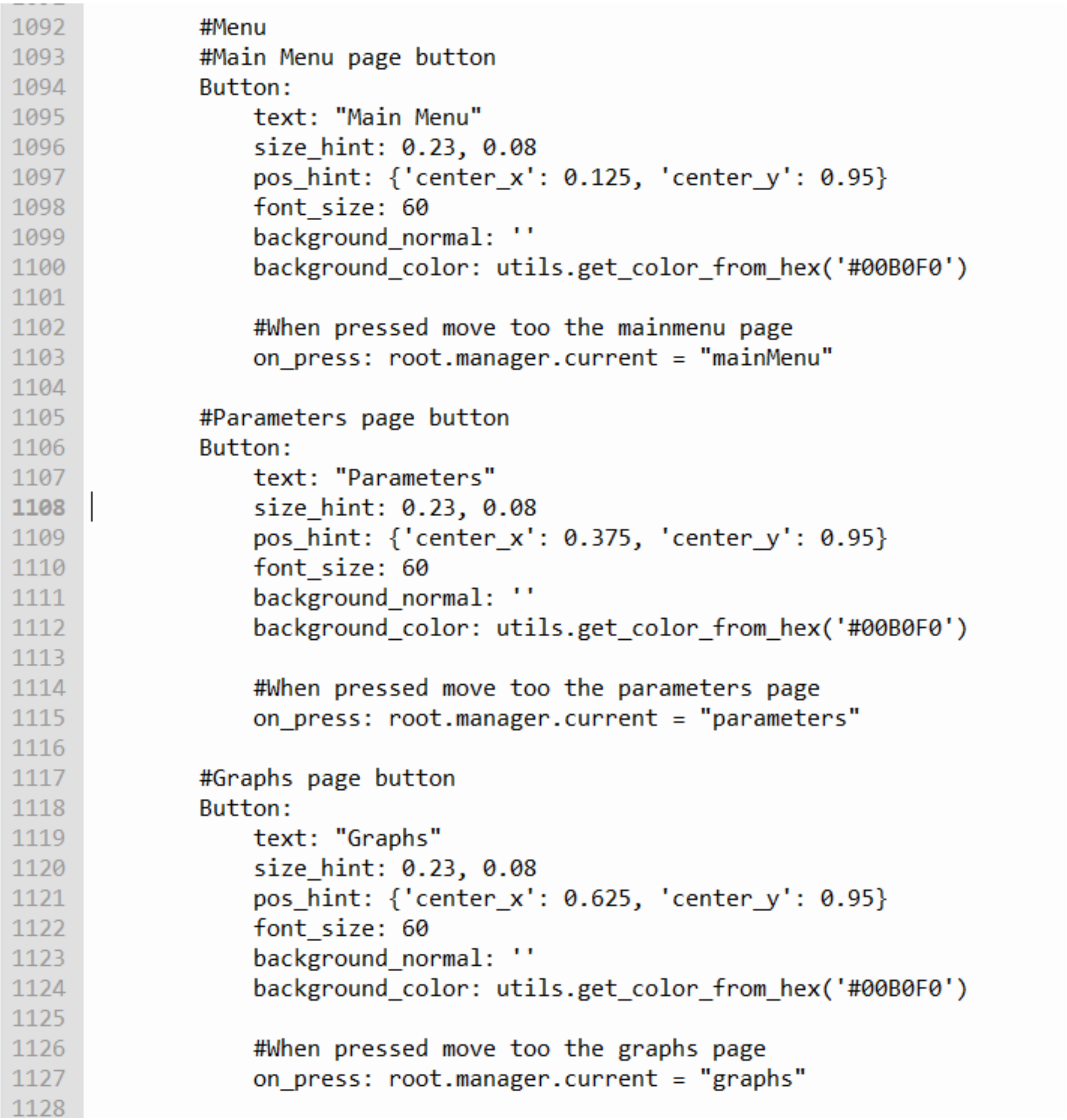
```
sm = ScreenManager()
sm.add_widget(WelcomeScreen(name="welcome"))
sm.add_widget(LoginScreen(name="login"))
sm.add_widget(MainMenuScreen(name="mainMenu"))
sm.add_widget(ParametersScreen(name="parameters"))
sm.add_widget(GraphsScreen(name="graphs"))
sm.add_widget(SettingsScreen(name="settings"))
```

A class has been added to the python file to relate to the settings screen. The final widget in the screen manager has also been added. At this stage all the different screens of the greenhouse have been added I just need to write the kivy code to define its layout.

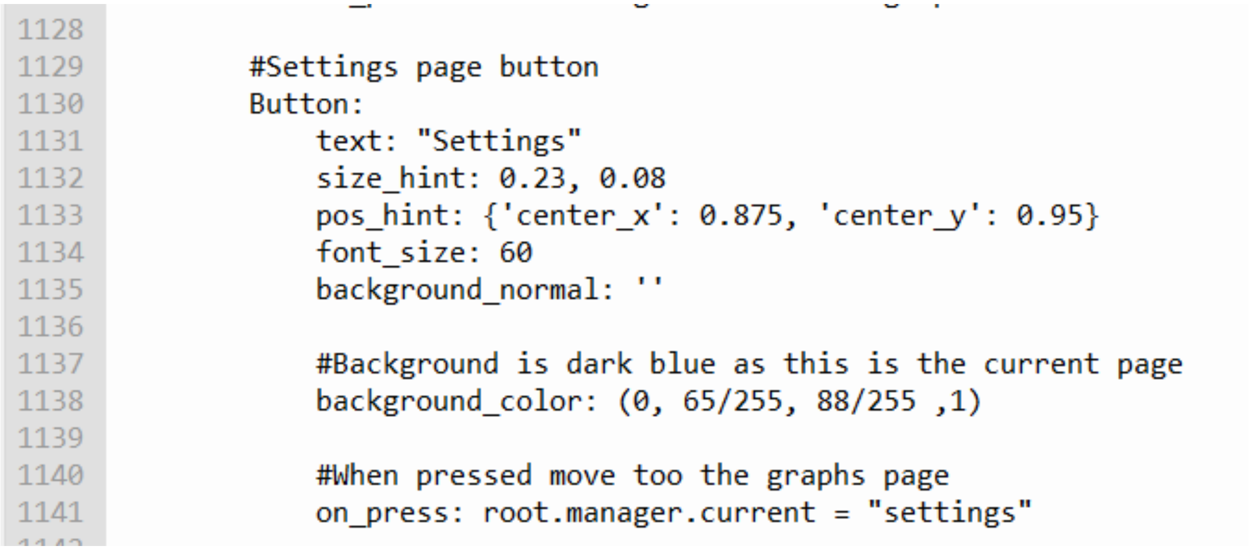


```
<SettingsScreen>
    FloatLayout:
        #Set the background colour too green
        canvas:
            Color:
                rgba: 0, 0.69, 0.31, 1
            Rectangle:
                pos: (0,0)
                size: self.width, self.height
```

I have defined a new screen inside the kivy file which has the same name as the screen class I made in python so that kivy knows they are the same. The screen is using float layout and has a green background.

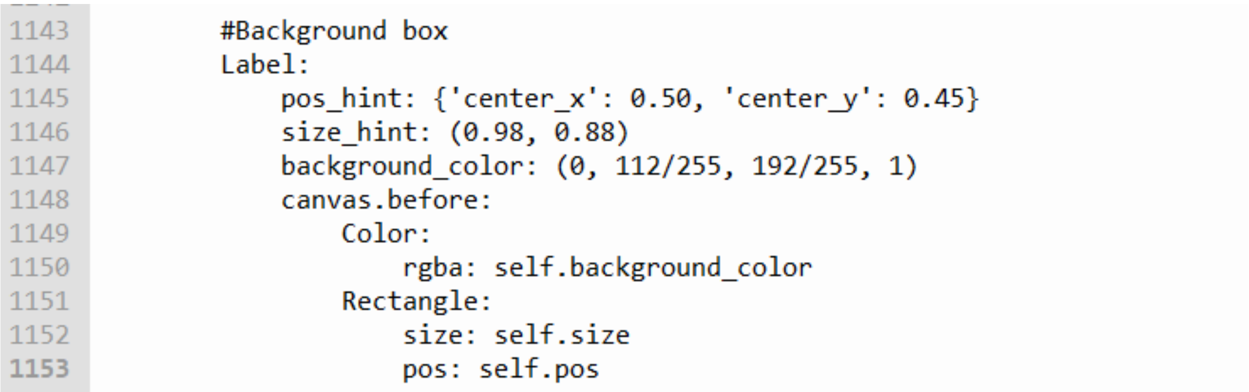


```
        #Menu
        #Main Menu page button
        Button:
            text: "Main Menu"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.125, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #When pressed move too the mainmenu page
            on_press: root.manager.current = "mainMenu"

        #Parameters page button
        Button:
            text: "Parameters"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.375, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #When pressed move too the parameters page
            on_press: root.manager.current = "parameters"

        #Graphs page button
        Button:
            text: "Graphs"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.625, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #When pressed move too the graphs page
            on_press: root.manager.current = "graphs"

```



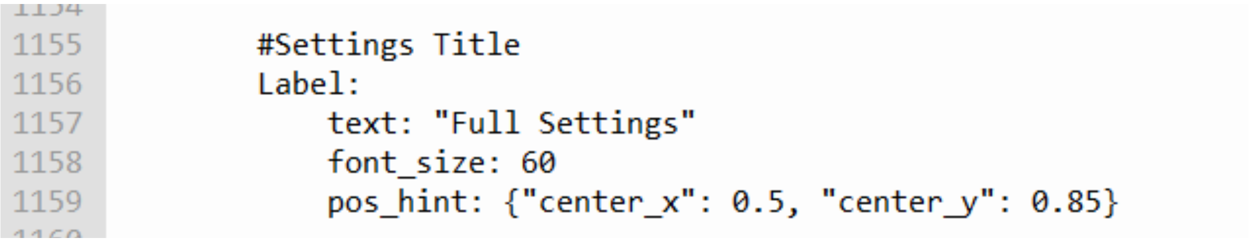
```
        #Settings page button
        Button:
            text: "Settings"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.875, 'center_y': 0.95}
            font_size: 60
            background_normal: ''

            #Background is dark blue as this is the current page
            background_color: (0, 65/255, 88/255 ,1)

            #When pressed move too the graphs page
            on_press: root.manager.current = "settings"
```

The menu has been adjusted so that the settings button has the dark blue background.



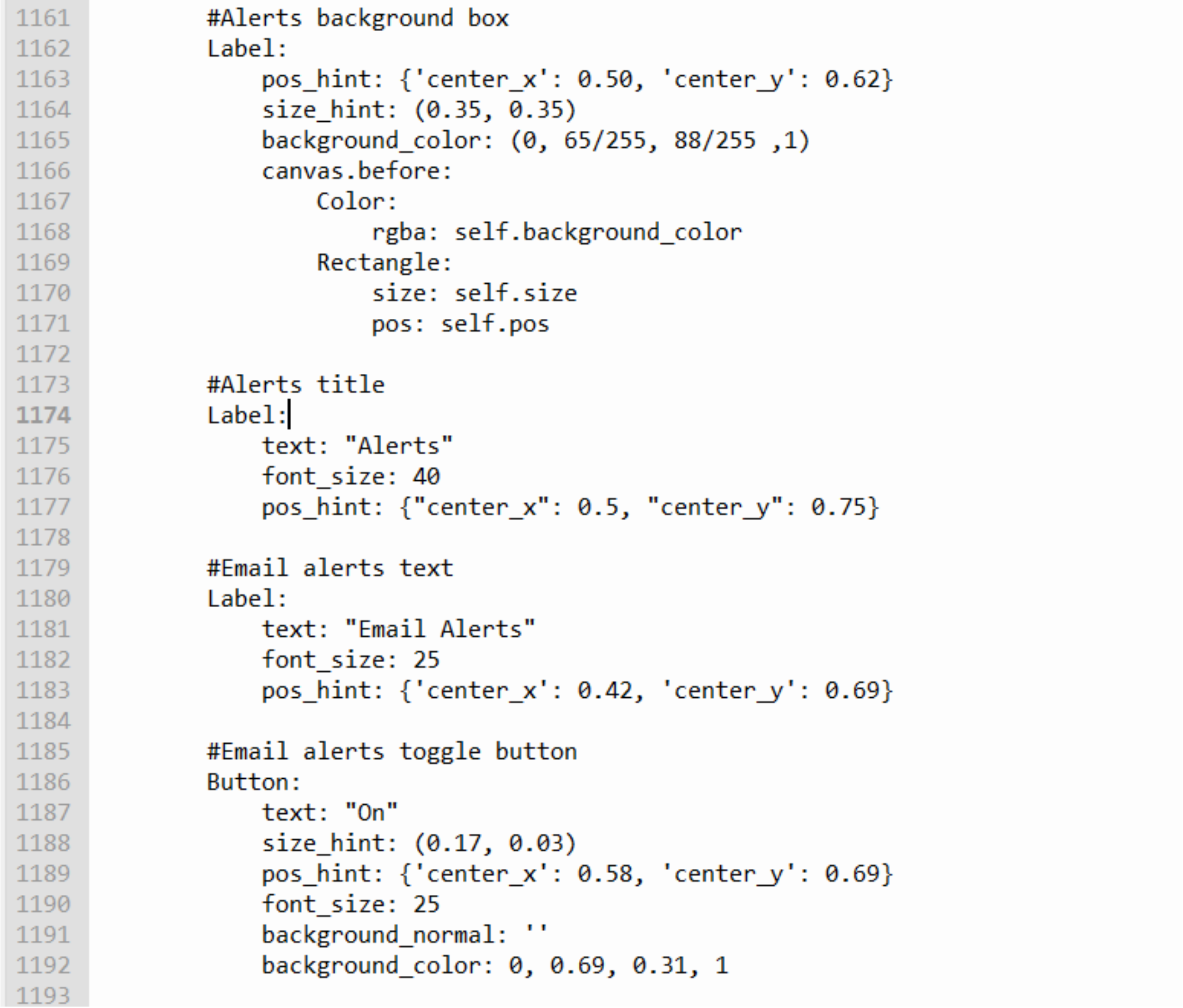
```
        #Background box
        Label:
            pos_hint: {'center_x': 0.50, 'center_y': 0.45}
            size_hint: (0.98, 0.88)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos
```

A large background box covers the rest of the screen which will contain the full settings section.

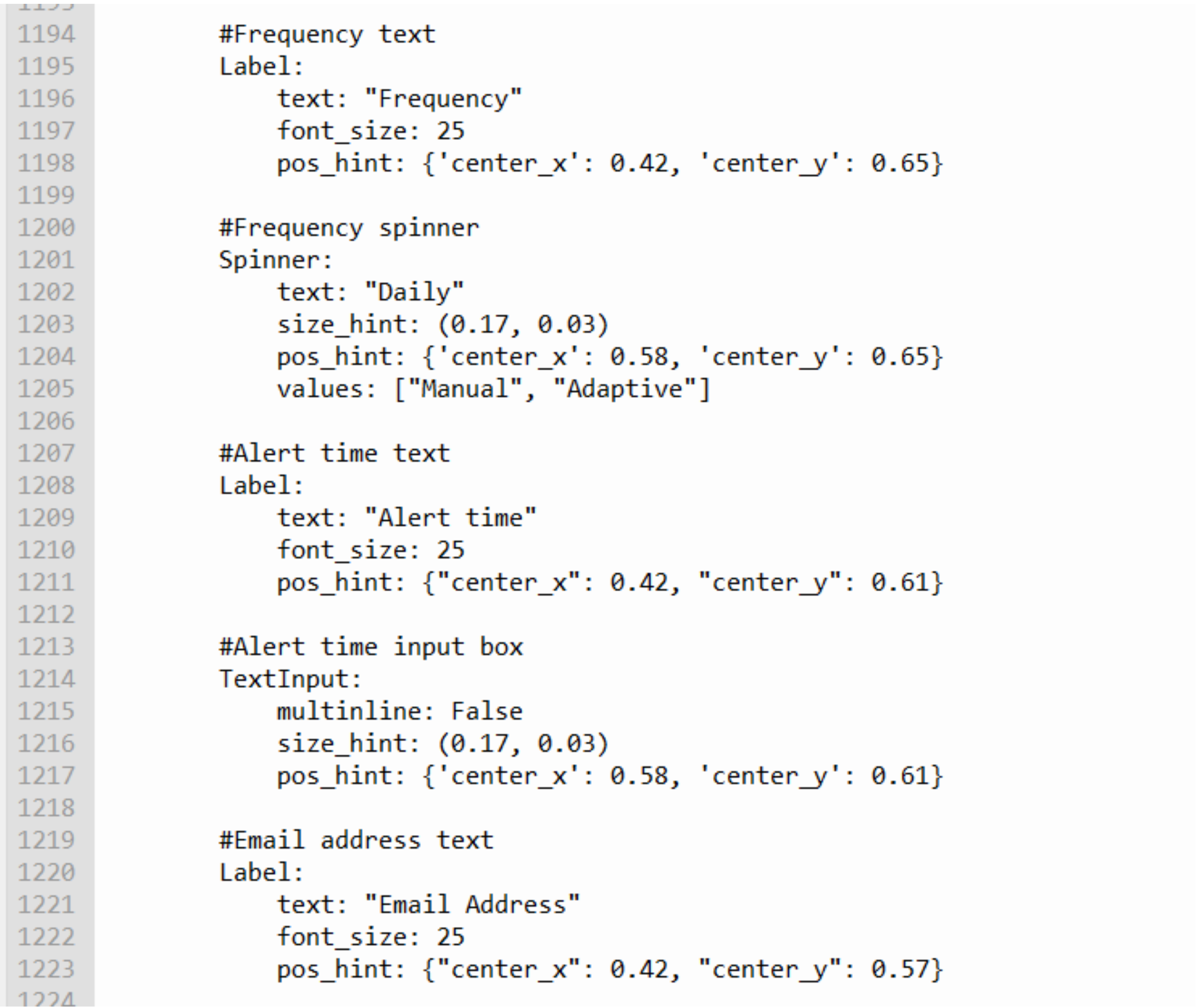


```
        #Settings Title
        Label:
            text: "Full Settings"
            font_size: 60
            pos_hint: {"center_x": 0.5, "center_y": 0.85}
```

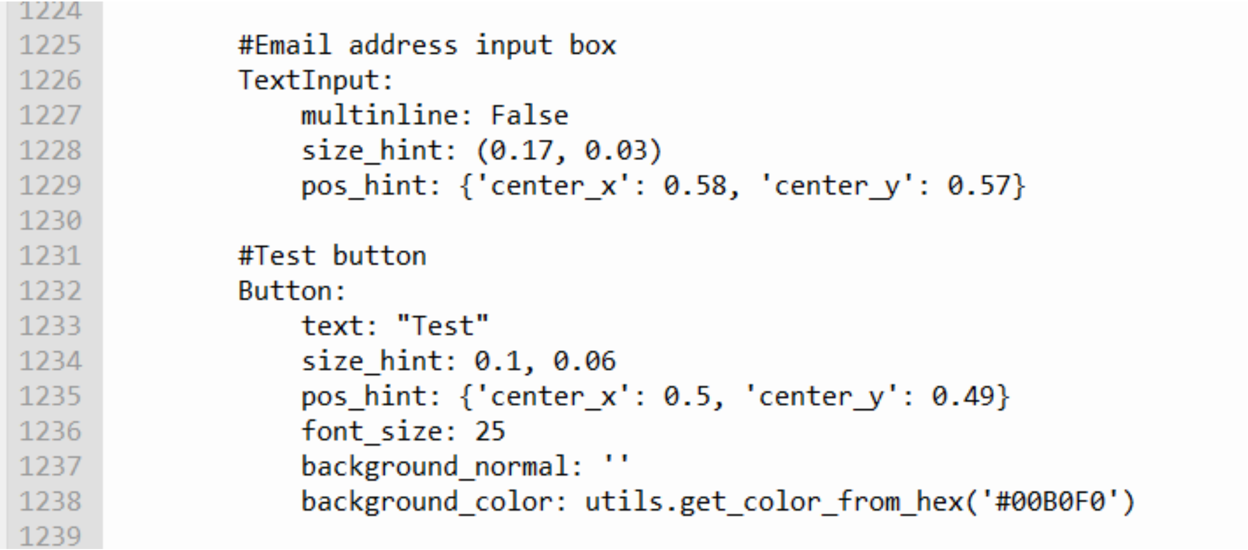
I have used a label too add a large label showing the user that this is the full settings page.



```
        #Alerts background box
        Label:
            pos_hint: {'center_x': 0.50, 'center_y': 0.62}
            size_hint: (0.35, 0.35)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Alerts title
        Label:
            text: "Alerts"
            font_size: 40
            pos_hint: {"center_x": 0.5, "center_y": 0.75}

        #Email alerts text
        Label:
            text: "Email Alerts"
            font_size: 25
            pos_hint: {'center_x': 0.42, 'center_y': 0.69}

        #Email alerts toggle button
        Button:
            text: "On"
            size_hint: (0.17, 0.03)
            pos_hint: {'center_x': 0.58, 'center_y': 0.69}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1

```

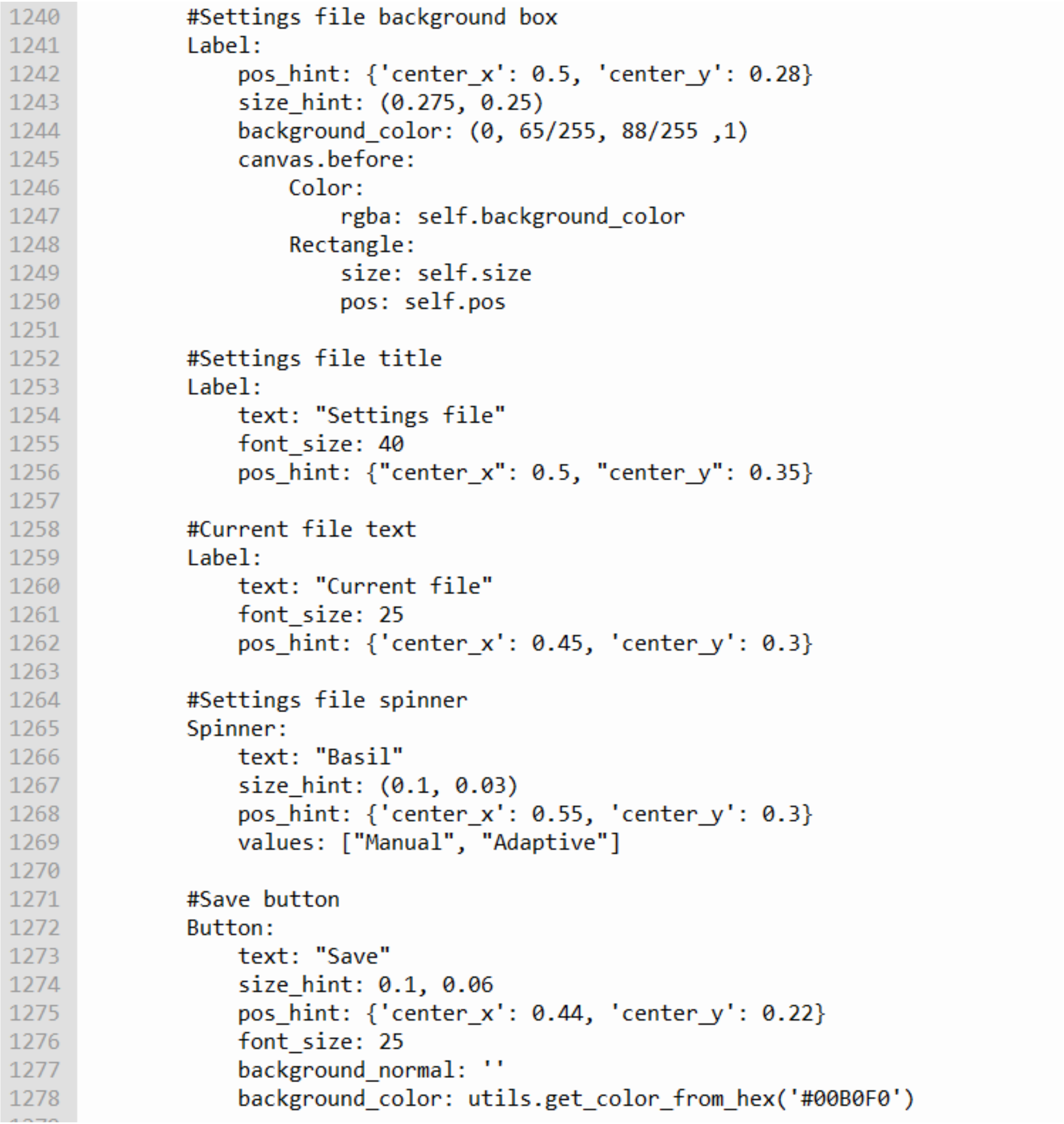


```
        #Frequency text
        Label:
            text: "Frequency"
            font_size: 25
            pos_hint: {'center_x': 0.42, 'center_y': 0.65}

        #Frequency spinner
        Spinner:
            text: "Daily"
            size_hint: (0.17, 0.03)
            pos_hint: {'center_x': 0.58, 'center_y': 0.65}
            values: ["Manual", "Adaptive"]

        #Alert time text
        Label:
            text: "Alert time"
            font_size: 25
            pos_hint: {"center_x": 0.42, "center_y": 0.61}

        #Alert time input box
        TextInput:
            multinline: False
            size_hint: (0.17, 0.03)
            pos_hint: {'center_x': 0.58, 'center_y': 0.61}

        #Email address text
        Label:
            text: "Email Address"
            font_size: 25
            pos_hint: {"center_x": 0.42, "center_y": 0.57}
```

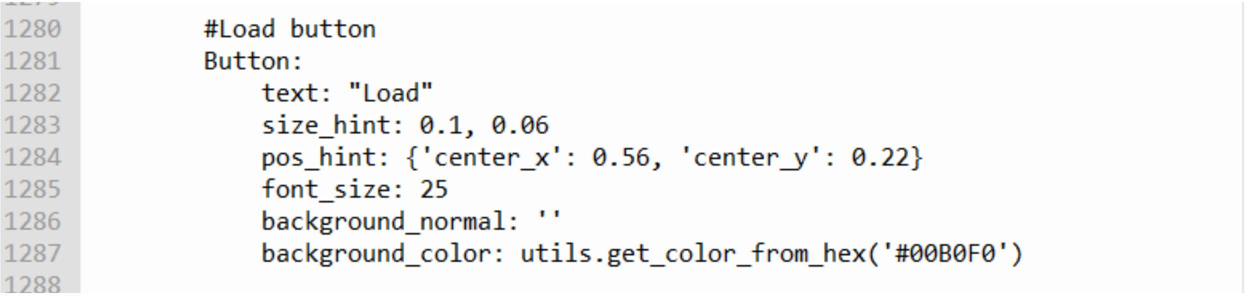


```

        #Email address input box
        TextInput:
            multinline: False
            size_hint: (0.17, 0.03)
            pos_hint: {'center_x': 0.58, 'center_y': 0.57}

        #Test button
        Button:
            text: "Test"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.5, 'center_y': 0.49}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

```

The alerts section consists of a toggle button, two text input boxes and a test button which will be used to send a test email to the users email address to ensure the email alerts are working as expected.

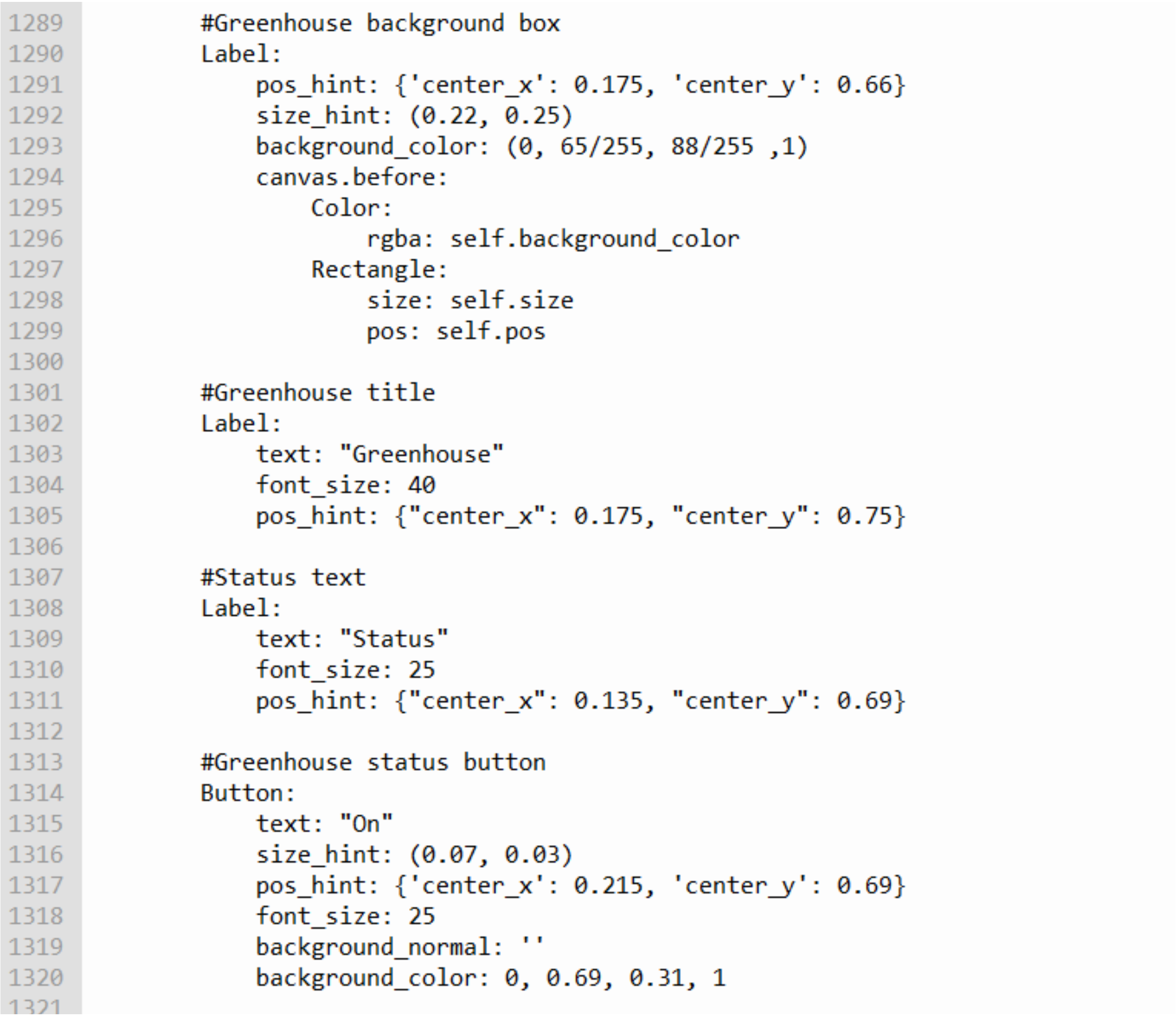


```
        #Settings file background box
        Label:
            pos_hint: {'center_x': 0.5, 'center_y': 0.28}
            size_hint: (0.275, 0.25)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Settings file title
        Label:
            text: "Settings file"
            font_size: 40
            pos_hint: {"center_x": 0.5, "center_y": 0.35}

        #Current file text
        Label:
            text: "Current file"
            font_size: 25
            pos_hint: {'center_x': 0.45, 'center_y': 0.3}

        #Settings file spinner
        Spinner:
            text: "Basil"
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.55, 'center_y': 0.3}
            values: ["Manual", "Adaptive"]

        #Save button
        Button:
            text: "Save"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.44, 'center_y': 0.22}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
```

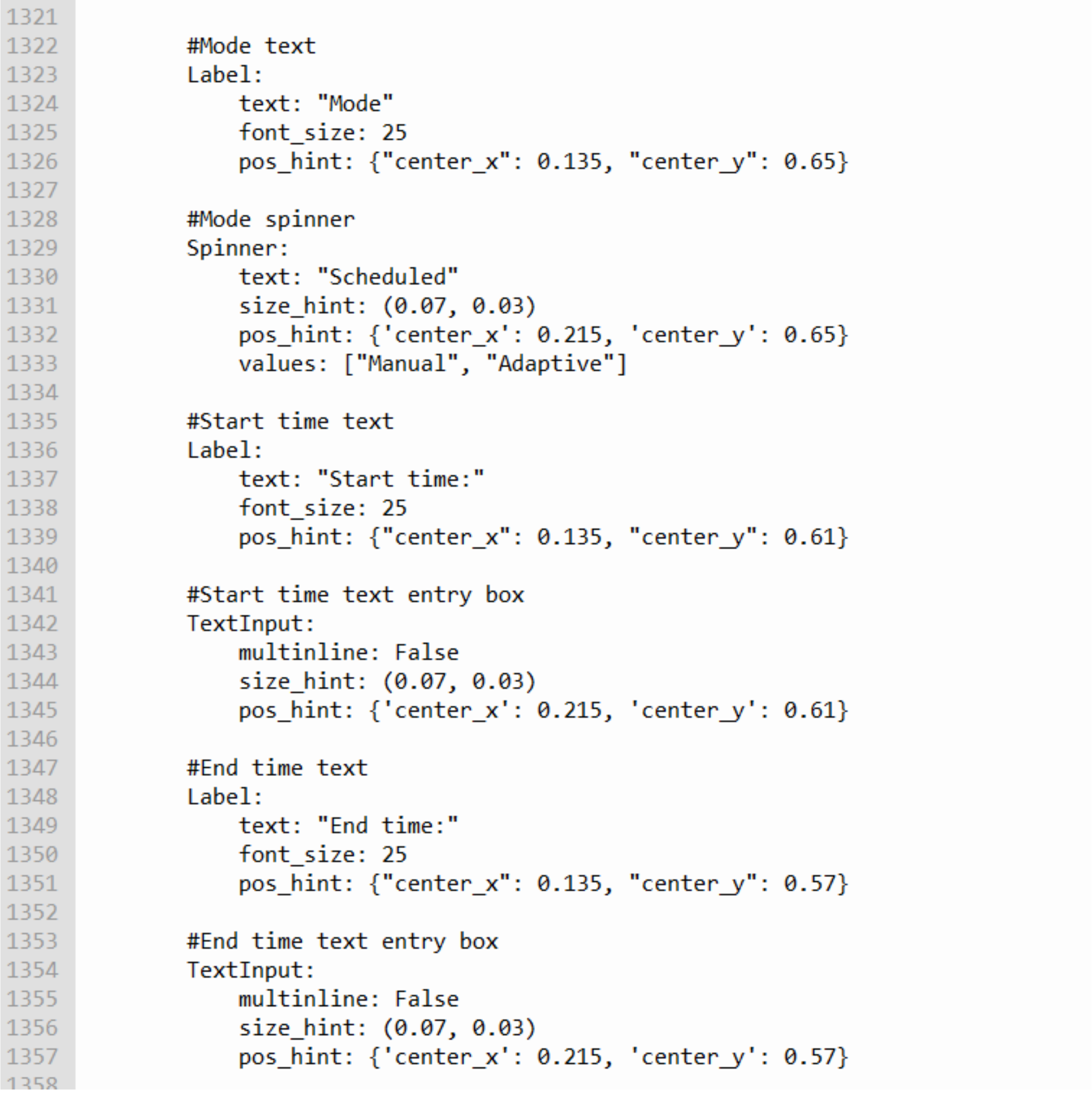


```
        #Load button
        Button:
            text: "Load"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.56, 'center_y': 0.22}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
```

The settings file area will let the user load a pre saved file into the greenhouse altering all the settings to the settings of that file. This section features a drop-down menu and two buttons to save and load files into the greenhouse.

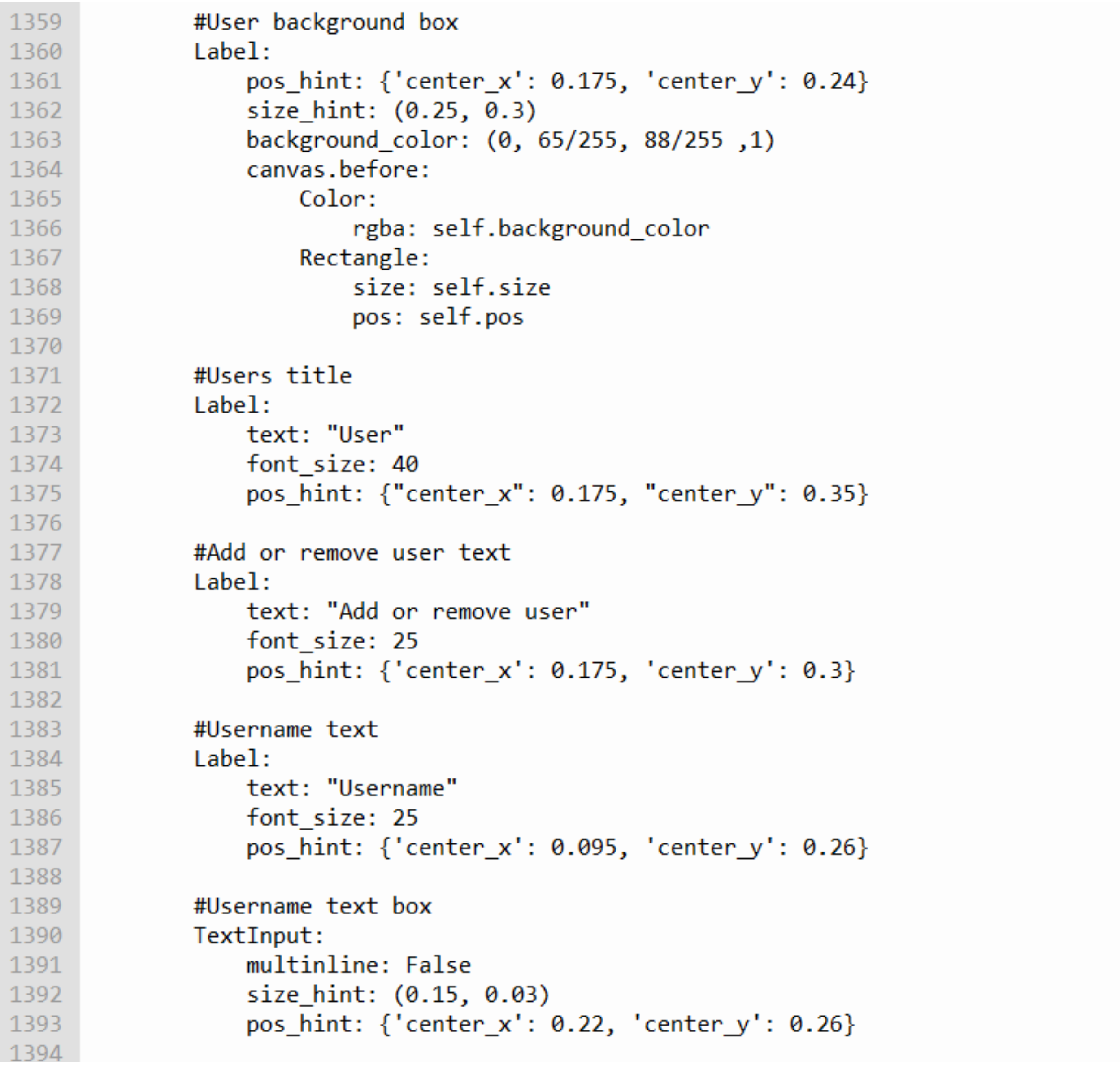


```
        #Greenhouse background box
        Label:
            pos_hint: {'center_x': 0.175, 'center_y': 0.66}
            size_hint: (0.22, 0.25)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Greenhouse title
        Label:
            text: "Greenhouse"
            font_size: 40
            pos_hint: {"center_x": 0.175, "center_y": 0.75}

        #Status text
        Label:
            text: "Status"
            font_size: 25
            pos_hint: {"center_x": 0.135, "center_y": 0.69}

        #Greenhouse status button
        Button:
            text: "On"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.215, 'center_y': 0.69}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1
```

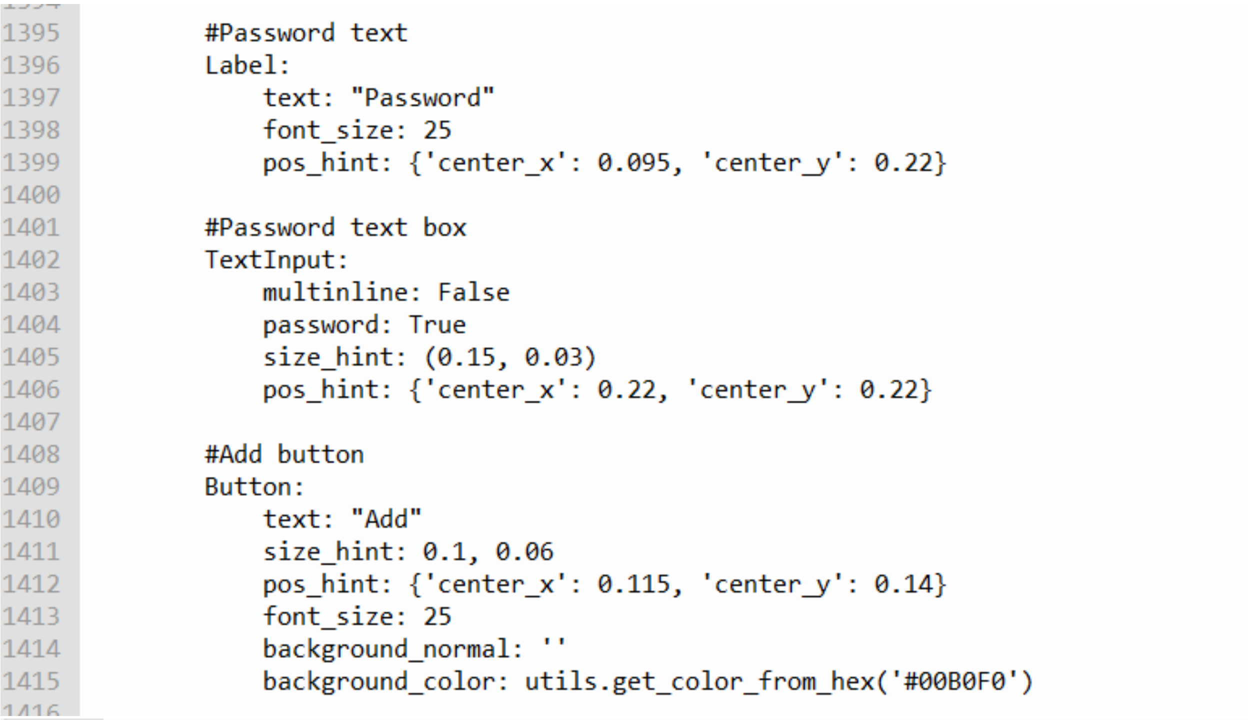


```

        #Mode text
        Label:
            text: "Mode"
            font_size: 25
            pos_hint: {"center_x": 0.135, "center_y": 0.65}

        #Mode spinner
        Spinner:
            text: "Scheduled"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.215, 'center_y': 0.65}
            values: ["Manual", "Adaptive"]

        #Start time text
        Label:
            text: "Start time:"
            font_size: 25
            pos_hint: {"center_x": 0.135, "center_y": 0.61}

        #Start time text entry box
        TextInput:
            multinline: False
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.215, 'center_y': 0.61}

        #End time text
        Label:
            text: "End time:"
            font_size: 25
            pos_hint: {"center_x": 0.135, "center_y": 0.57}

        #End time text entry box
        TextInput:
            multinline: False
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.215, 'center_y': 0.57}
```

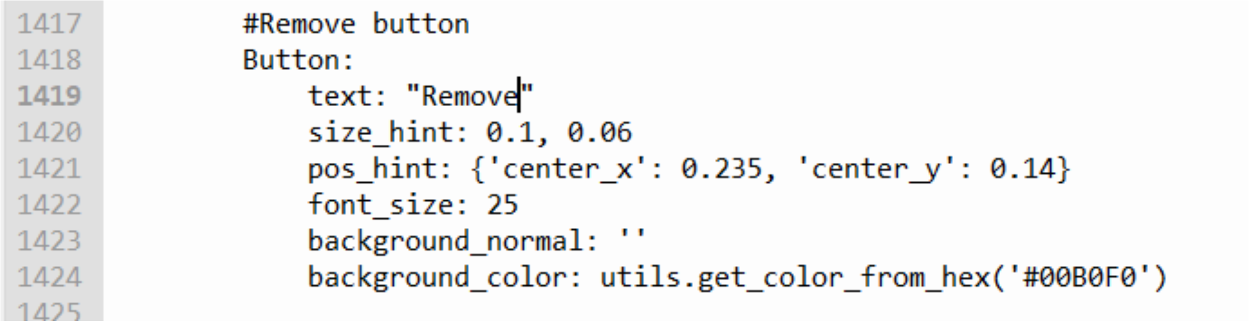
The greenhouse will be turned on and off using a toggle button and can be scheduled by entering the start and end operating time of the greenhouse into two text input boxes.



```
        #User background box
        Label:
            pos_hint: {'center_x': 0.175, 'center_y': 0.24}
            size_hint: (0.25, 0.3)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Users title
        Label:
            text: "User"
            font_size: 40
            pos_hint: {"center_x": 0.175, "center_y": 0.35}

        #Add or remove user text
        Label:
            text: "Add or remove user"
            font_size: 25
            pos_hint: {'center_x': 0.175, 'center_y': 0.3}

        #Username text
        Label:
            text: "Username"
            font_size: 25
            pos_hint: {'center_x': 0.095, 'center_y': 0.26}

        #Username text box
        TextInput:
            multinline: False
            size_hint: (0.15, 0.03)
            pos_hint: {'center_x': 0.22, 'center_y': 0.26}

```

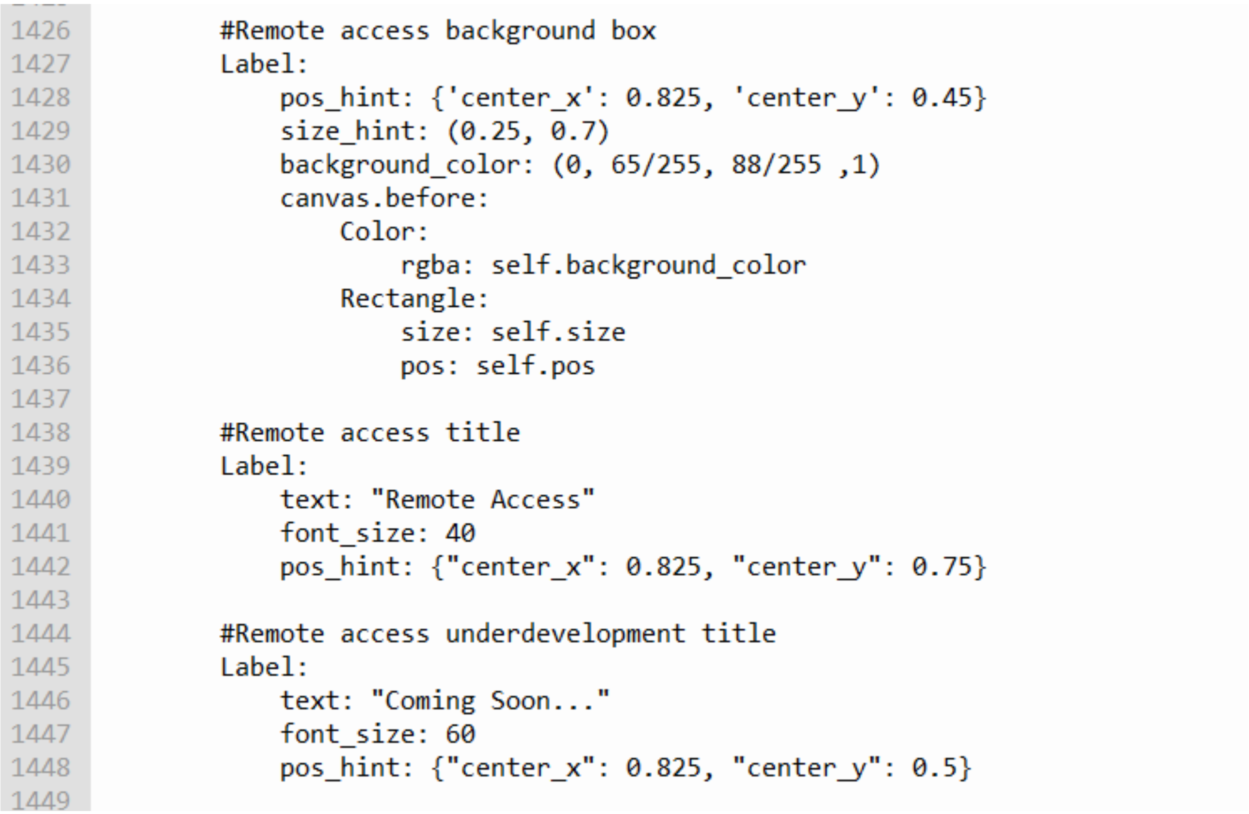


```
        #Password text
        Label:
            text: "Password"
            font_size: 25
            pos_hint: {'center_x': 0.095, 'center_y': 0.22}

        #Password text box
        TextInput:
            multinline: False
            password: True
            size_hint: (0.15, 0.03)
            pos_hint: {'center_x': 0.22, 'center_y': 0.22}

        #Add button
        Button:
            text: "Add"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.115, 'center_y': 0.14}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
```



```
        #Remove button
        Button:
            text: "Remove"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.235, 'center_y': 0.14}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')
```

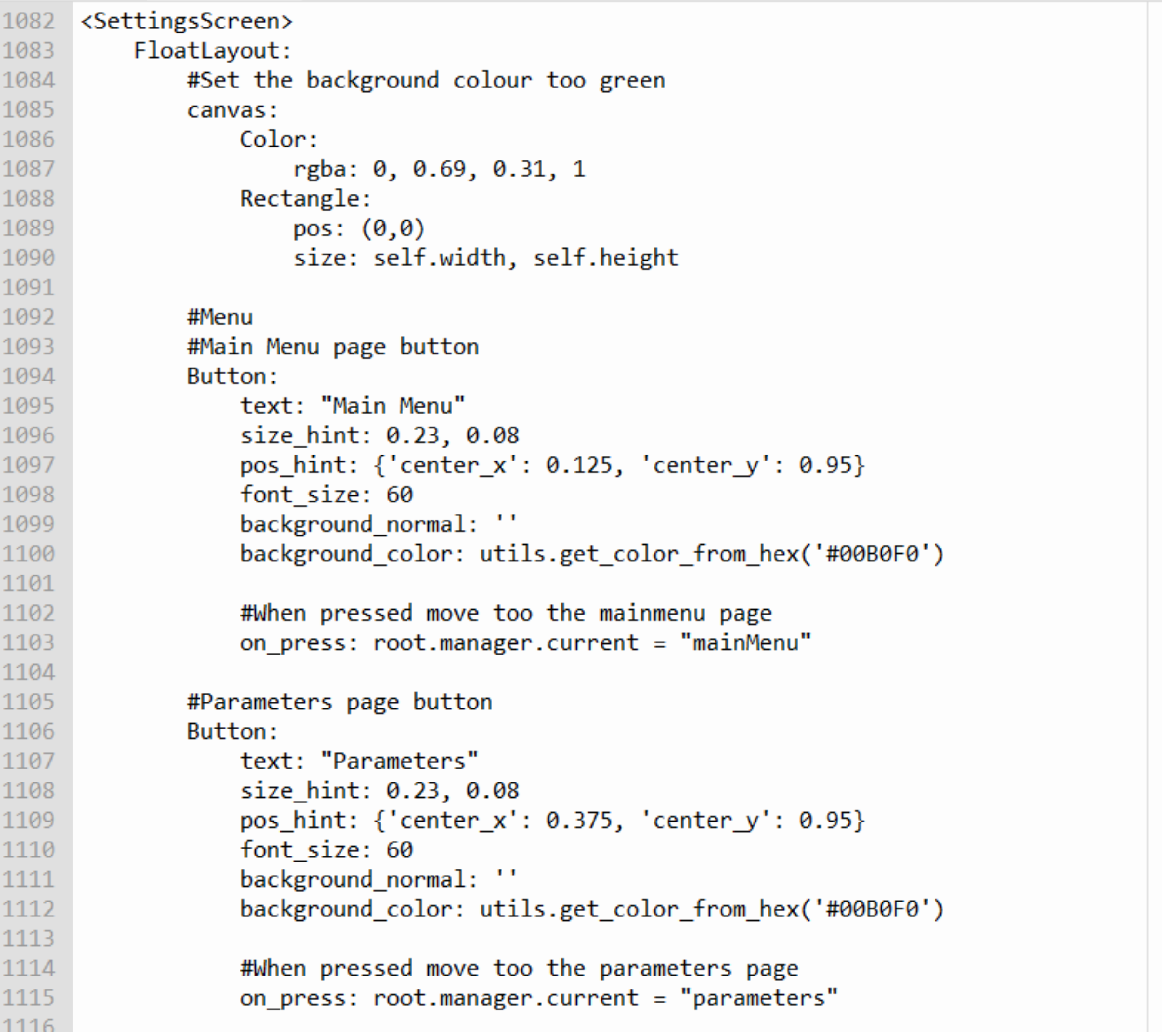
The user will be able to add and remove users using the following area of the full settings page. There is a text entry box for entering the username and one for the password. There are then two buttons one for adding the user and another for removing the user.



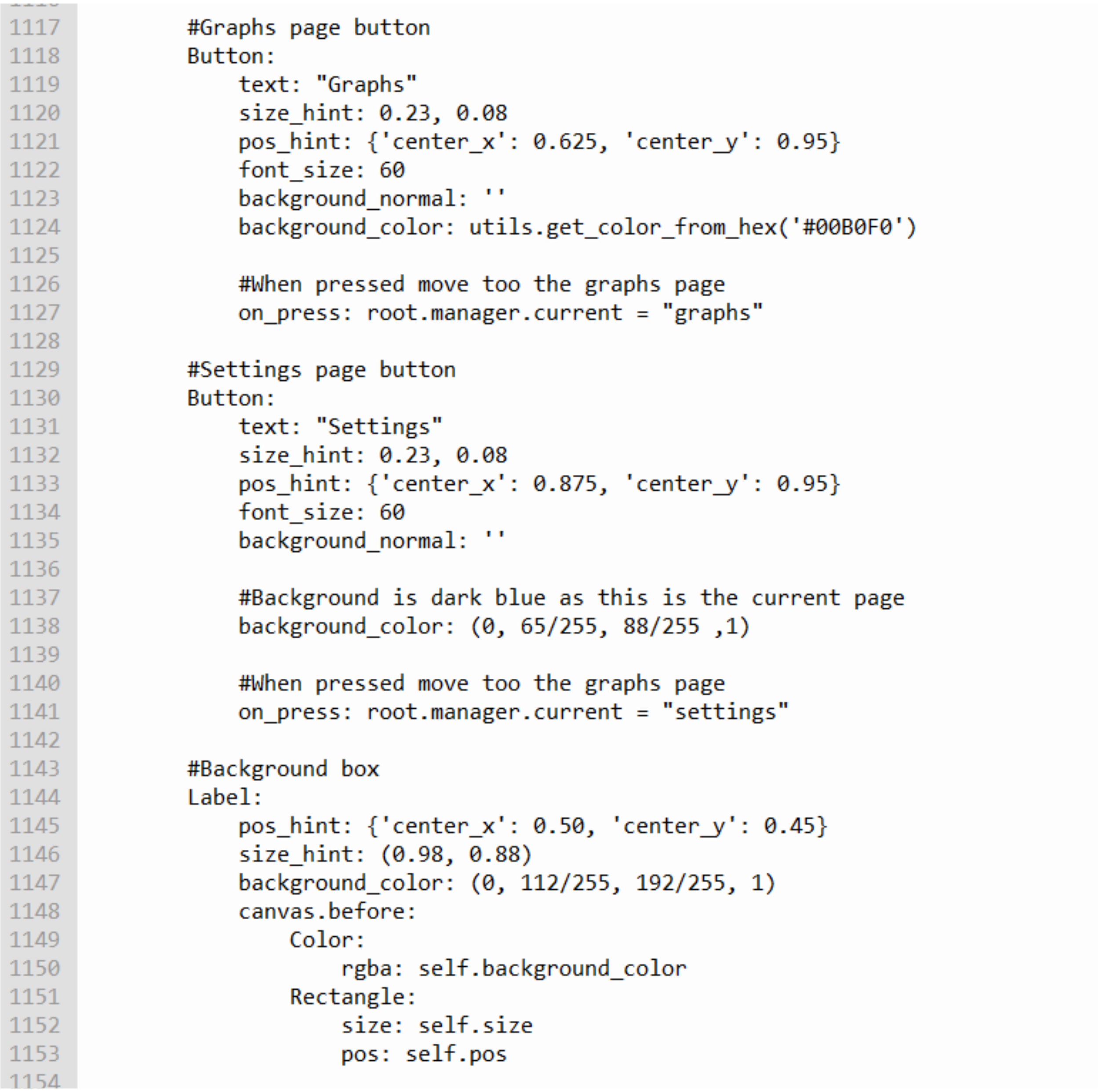
```
        #Remote access background box
        Label:
            pos_hint: {'center_x': 0.825, 'center_y': 0.45}
            size_hint: (0.25, 0.7)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Remote access title
        Label:
            text: "Remote Access"
            font_size: 40
            pos_hint: {"center_x": 0.825, "center_y": 0.75}

        #Remote access underdevelopment title
        Label:
            text: "Coming Soon..."
            font_size: 60
            pos_hint: {"center_x": 0.825, "center_y": 0.5}
```

The final section of the settings screen is the remote access area as discussed due to time constraints this feature won't be included so I have added a simple coming soon sign to this area of the GUI.

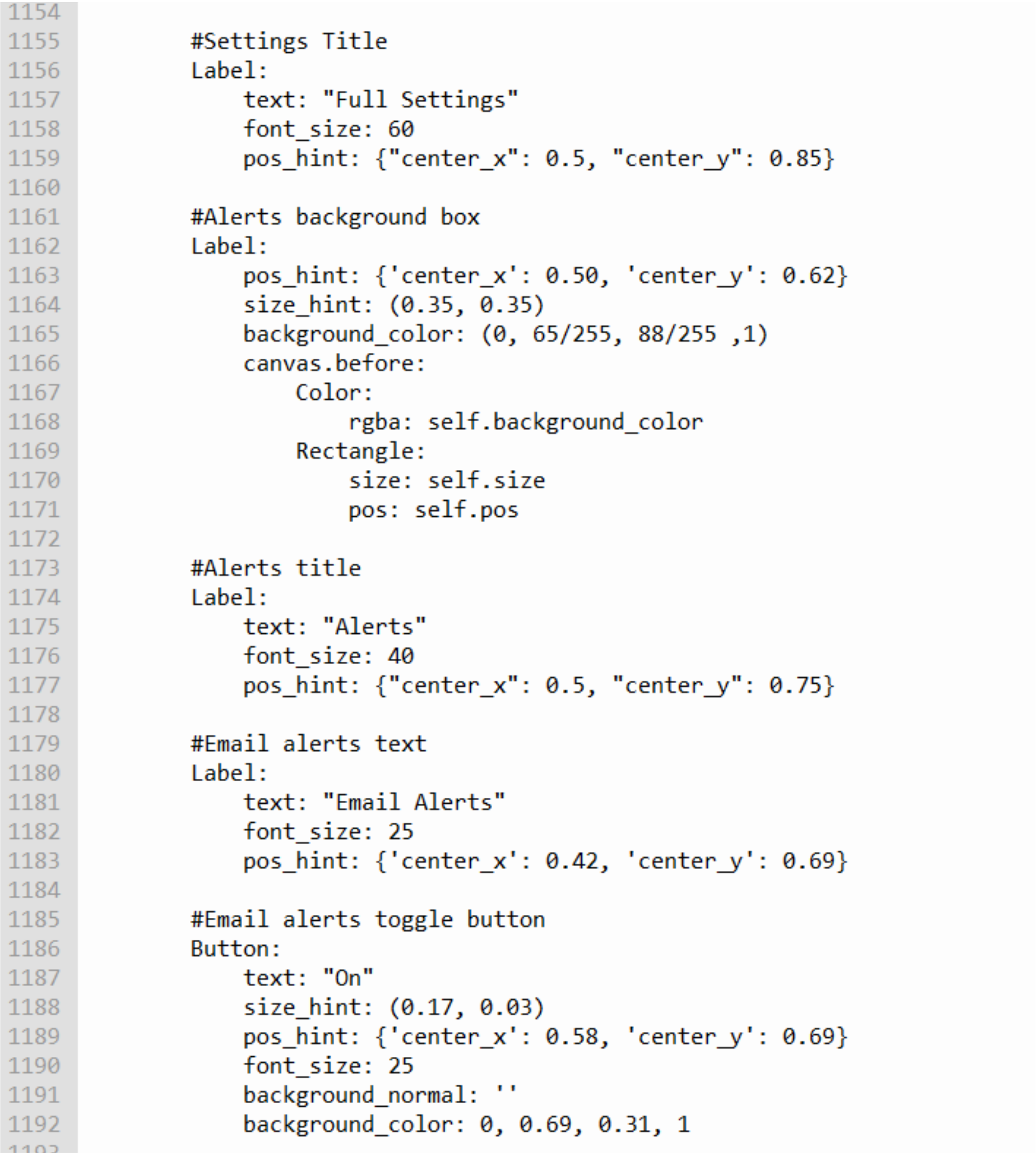
Complete settings page code



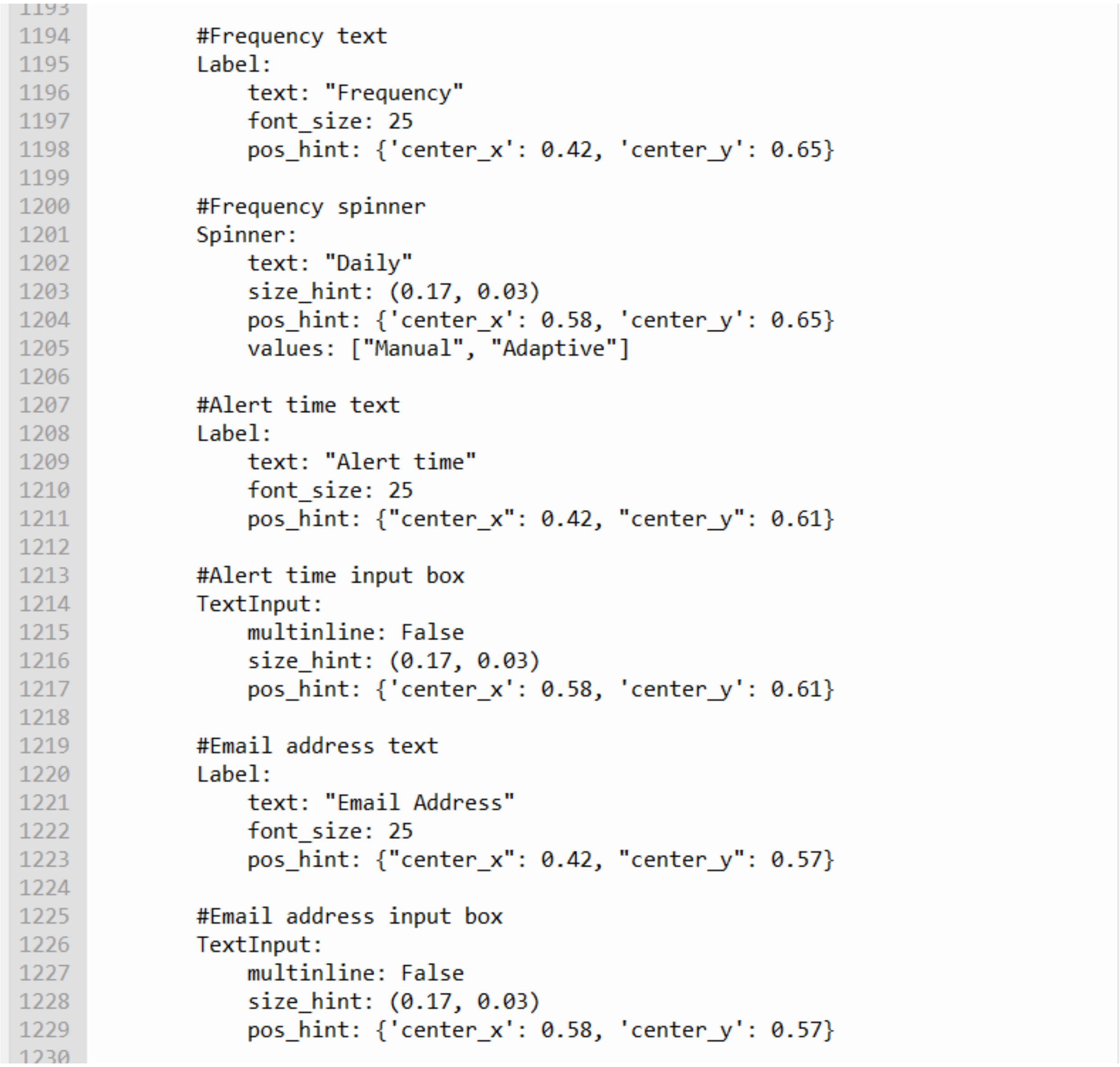
```
<SettingsScreen>
    FloatLayout:
        #Set the background colour too green
        canvas:
            Color:
                rgba: 0, 0.69, 0.31, 1
            Rectangle:
                pos: (0,0)
                size: self.width, self.height

        #Menu
        #Main Menu page button
        Button:
            text: "Main Menu"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.125, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #When pressed move too the mainmenu page
            on_press: root.manager.current = "mainMenu"

        #Parameters page button
        Button:
            text: "Parameters"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.375, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #When pressed move too the parameters page
            on_press: root.manager.current = "parameters"
```



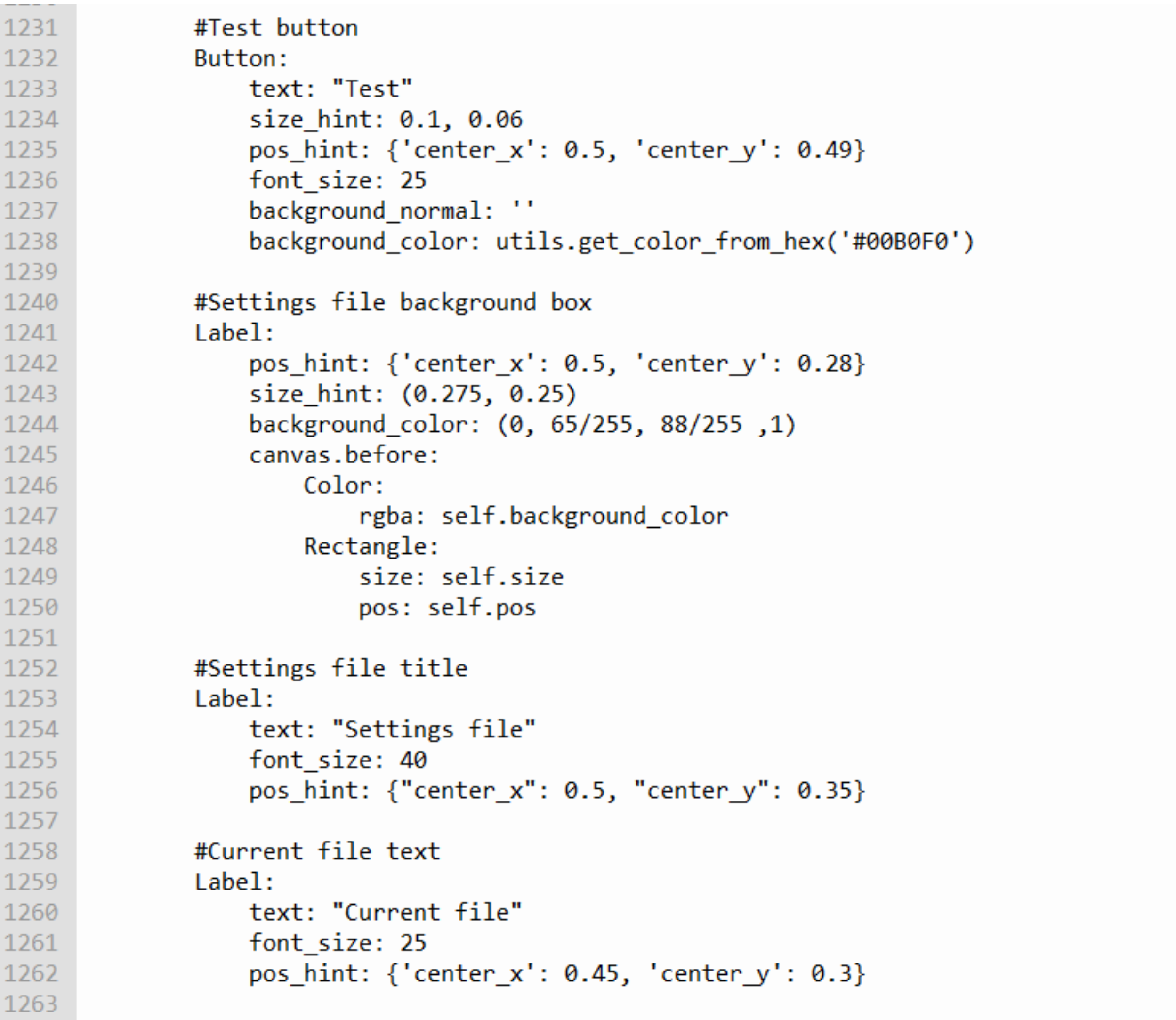
```
        #Graphs page button
        Button:
            text: "Graphs"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.625, 'center_y': 0.95}
            font_size: 60
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #When pressed move too the graphs page
            on_press: root.manager.current = "graphs"

        #Settings page button
        Button:
            text: "Settings"
            size_hint: 0.23, 0.08
            pos_hint: {'center_x': 0.875, 'center_y': 0.95}
            font_size: 60
            background_normal: ''

            #Background is dark blue as this is the current page
            background_color: (0, 65/255, 88/255 ,1)

            #When pressed move too the graphs page
            on_press: root.manager.current = "settings"

        #Background box
        Label:
            pos_hint: {'center_x': 0.50, 'center_y': 0.45}
            size_hint: (0.98, 0.88)
            background_color: (0, 112/255, 192/255, 1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos
```



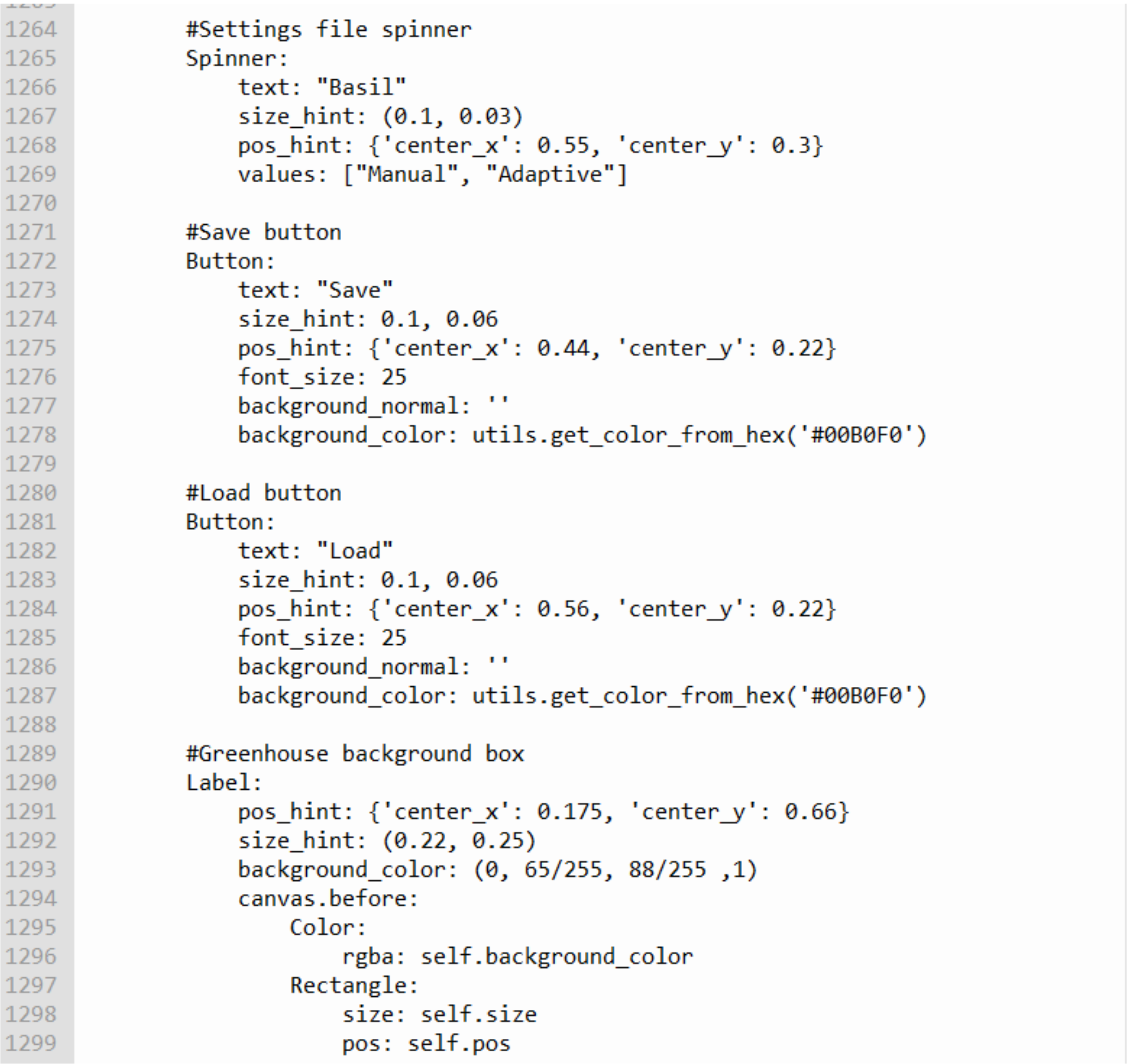
```

        #Settings Title
        Label:
            text: "Full Settings"
            font_size: 60
            pos_hint: {"center_x": 0.5, "center_y": 0.85}

        #Alerts background box
        Label:
            pos_hint: {'center_x': 0.50, 'center_y': 0.62}
            size_hint: (0.35, 0.35)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Alerts title
        Label:
            text: "Alerts"
            font_size: 40
            pos_hint: {"center_x": 0.5, "center_y": 0.75}

        #Email alerts text
        Label:
            text: "Email Alerts"
            font_size: 25
            pos_hint: {'center_x': 0.42, 'center_y': 0.69}

        #Email alerts toggle button
        Button:
            text: "On"
            size_hint: (0.17, 0.03)
            pos_hint: {'center_x': 0.58, 'center_y': 0.69}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1
```



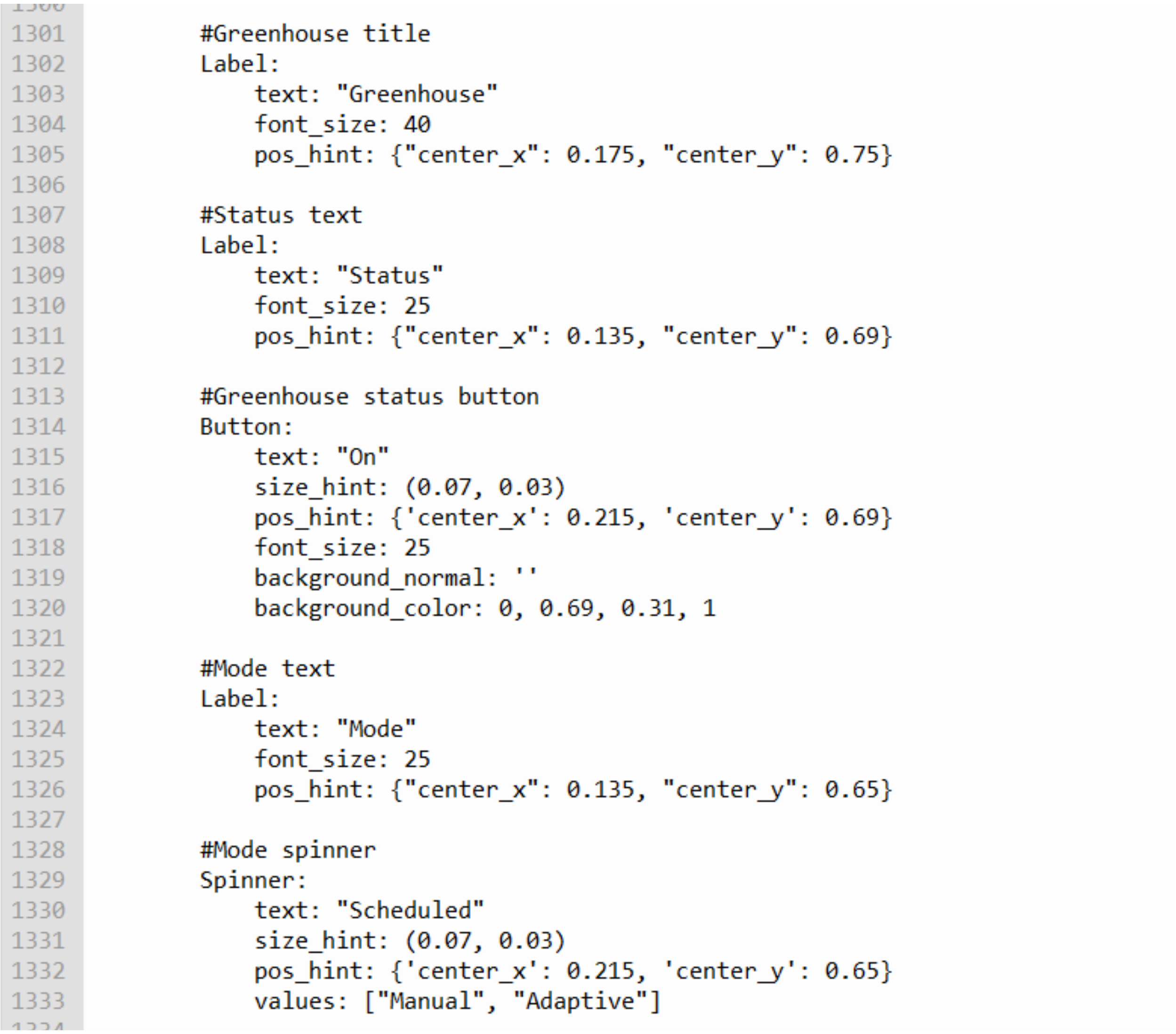
```
        #Frequency text
        Label:
            text: "Frequency"
            font_size: 25
            pos_hint: {'center_x': 0.42, 'center_y': 0.65}

        #Frequency spinner
        Spinner:
            text: "Daily"
            size_hint: (0.17, 0.03)
            pos_hint: {'center_x': 0.58, 'center_y': 0.65}
            values: ["Manual", "Adaptive"]

        #Alert time text
        Label:
            text: "Alert time"
            font_size: 25
            pos_hint: {"center_x": 0.42, "center_y": 0.61}

        #Alert time input box
        TextInput:
            multinline: False
            size_hint: (0.17, 0.03)
            pos_hint: {'center_x': 0.58, 'center_y': 0.61}

        #Email address text
        Label:
            text: "Email Address"
            font_size: 25
            pos_hint: {"center_x": 0.42, "center_y": 0.57}

        #Email address input box
        TextInput:
            multinline: False
            size_hint: (0.17, 0.03)
            pos_hint: {'center_x': 0.58, 'center_y': 0.57}
```



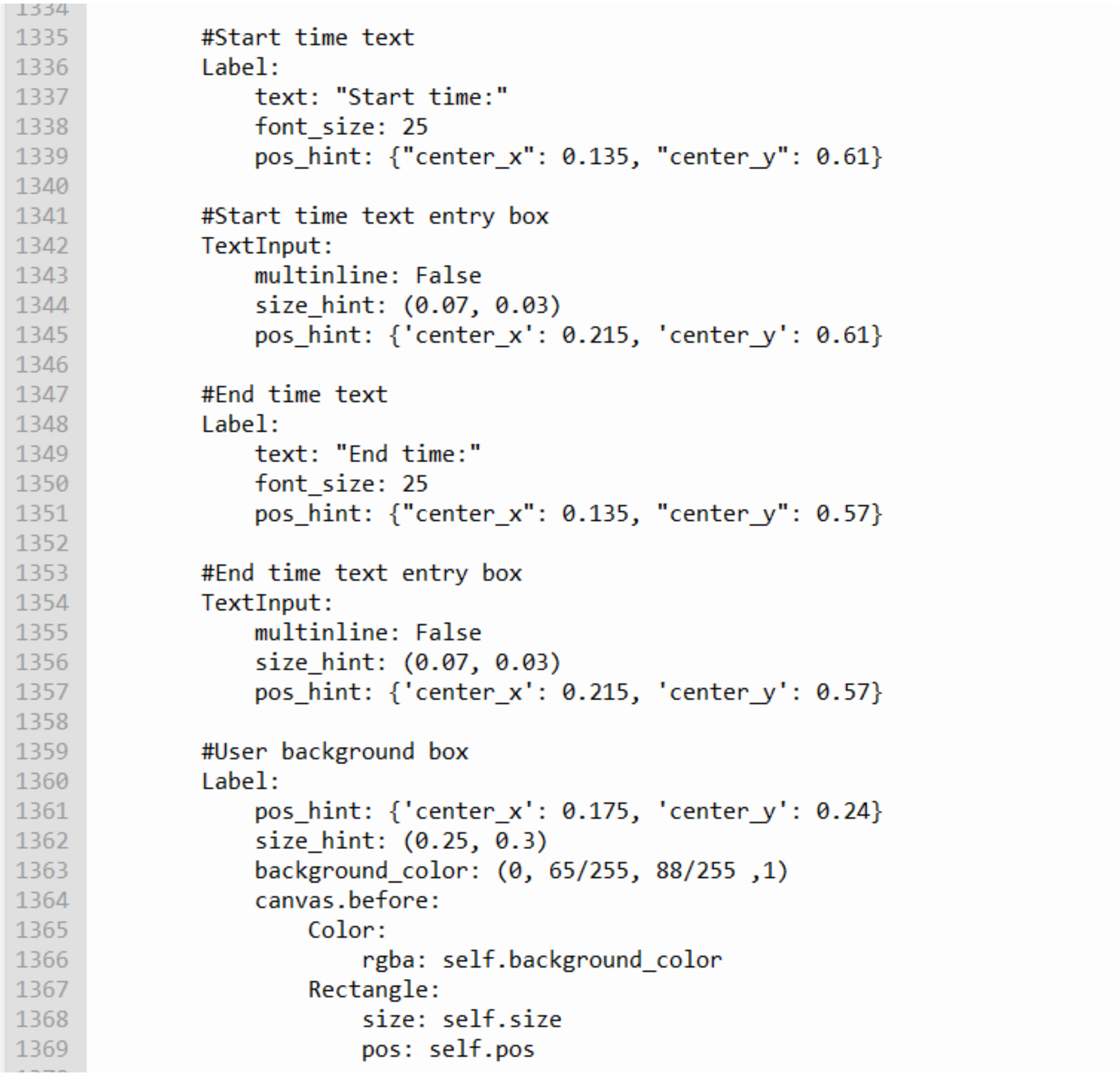
```
        #Test button
        Button:
            text: "Test"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.5, 'center_y': 0.49}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

        #Settings file background box
        Label:
            pos_hint: {'center_x': 0.5, 'center_y': 0.28}
            size_hint: (0.275, 0.25)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Settings file title
        Label:
            text: "Settings file"
            font_size: 40
            pos_hint: {"center_x": 0.5, "center_y": 0.35}

        #Current file text
        Label:
            text: "Current file"
            font_size: 25
            pos_hint: {'center_x': 0.45, 'center_y': 0.3}
```



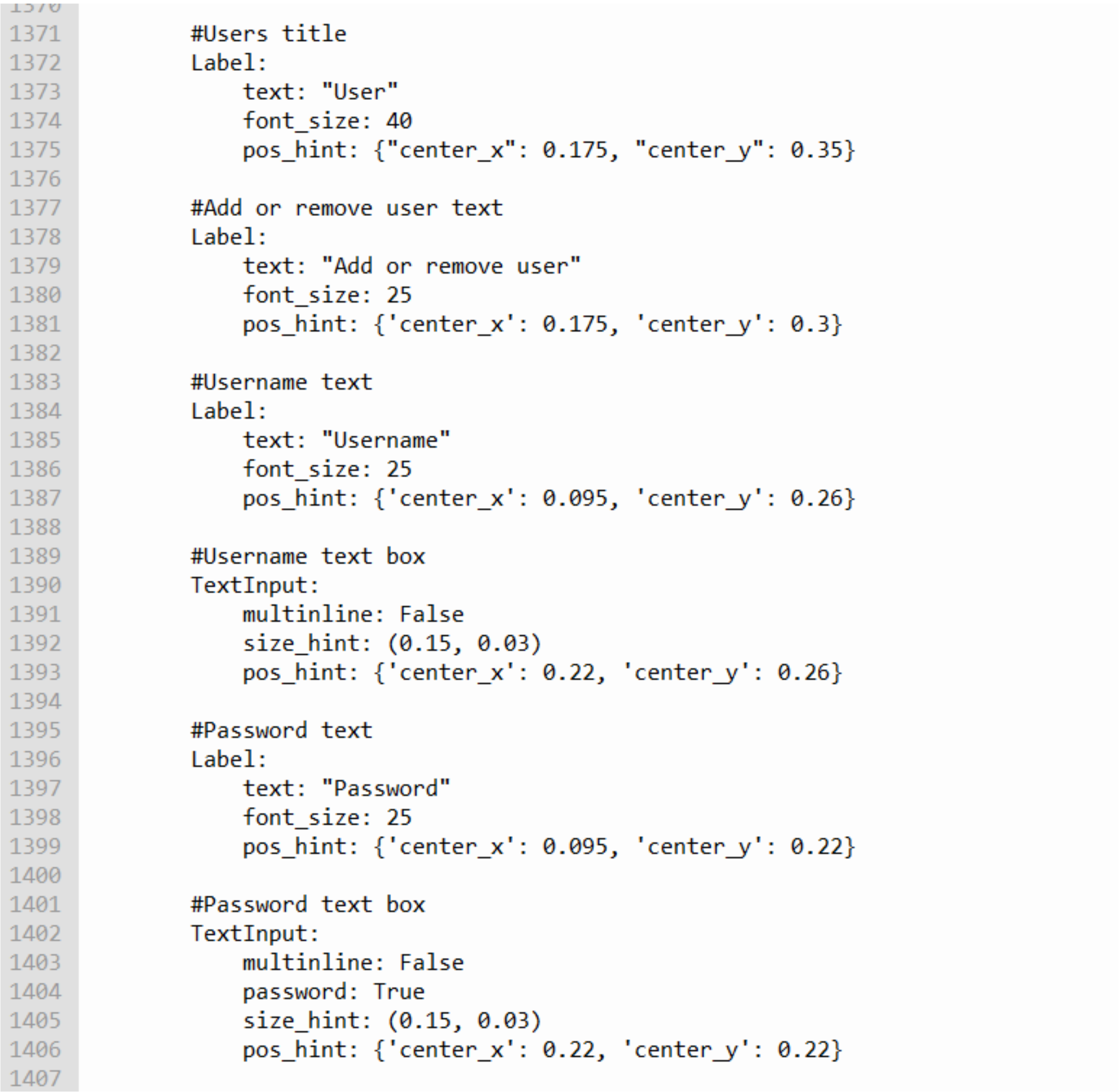
```
        #Settings file spinner
        Spinner:
            text: "Basil"
            size_hint: (0.1, 0.03)
            pos_hint: {'center_x': 0.55, 'center_y': 0.3}
            values: ["Manual", "Adaptive"]

        #Save button
        Button:
            text: "Save"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.44, 'center_y': 0.22}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

        #Load button
        Button:
            text: "Load"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.56, 'center_y': 0.22}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

        #Greenhouse background box
        Label:
            pos_hint: {'center_x': 0.175, 'center_y': 0.66}
            size_hint: (0.22, 0.25)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos
```



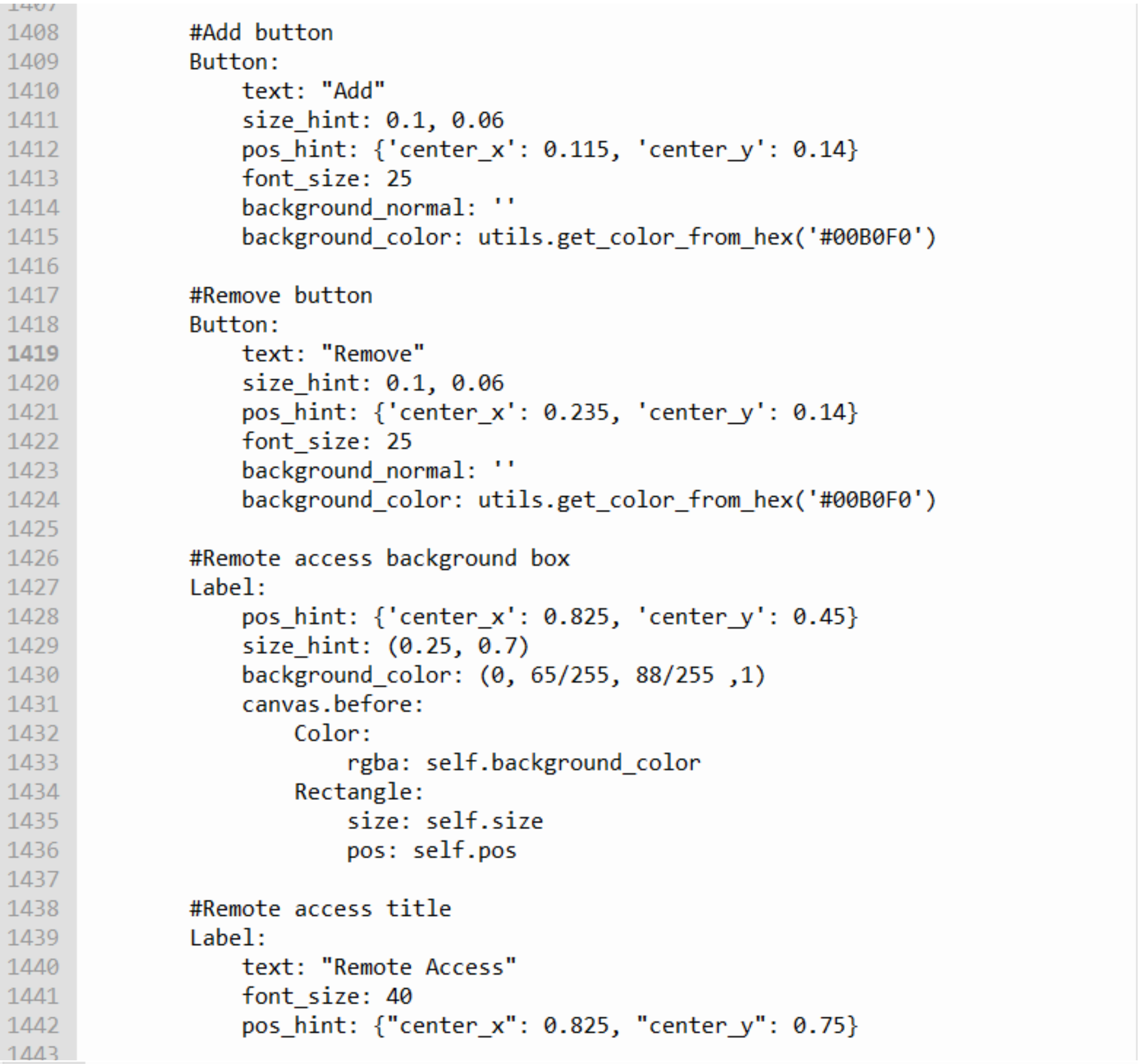
```
        #Greenhouse title
        Label:
            text: "Greenhouse"
            font_size: 40
            pos_hint: {"center_x": 0.175, "center_y": 0.75}

        #Status text
        Label:
            text: "Status"
            font_size: 25
            pos_hint: {"center_x": 0.135, "center_y": 0.69}

        #Greenhouse status button
        Button:
            text: "On"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.215, 'center_y': 0.69}
            font_size: 25
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1

        #Mode text
        Label:
            text: "Mode"
            font_size: 25
            pos_hint: {"center_x": 0.135, "center_y": 0.65}

        #Mode spinner
        Spinner:
            text: "Scheduled"
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.215, 'center_y': 0.65}
            values: ["Manual", "Adaptive"]
```



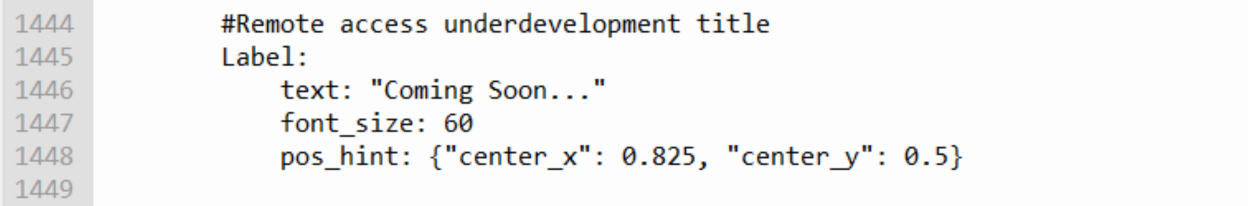
```
        #Start time text
        Label:
            text: "Start time:"
            font_size: 25
            pos_hint: {"center_x": 0.135, "center_y": 0.61}

        #Start time text entry box
        TextInput:
            multinline: False
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.215, 'center_y': 0.61}

        #End time text
        Label:
            text: "End time:"
            font_size: 25
            pos_hint: {"center_x": 0.135, "center_y": 0.57}

        #End time text entry box
        TextInput:
            multinline: False
            size_hint: (0.07, 0.03)
            pos_hint: {'center_x': 0.215, 'center_y': 0.57}

        #User background box
        Label:
            pos_hint: {'center_x': 0.175, 'center_y': 0.24}
            size_hint: (0.25, 0.3)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos
```



```
        #Users title
        Label:
            text: "User"
            font_size: 40
            pos_hint: {"center_x": 0.175, "center_y": 0.35}

        #Add or remove user text
        Label:
            text: "Add or remove user"
            font_size: 25
            pos_hint: {'center_x': 0.175, 'center_y': 0.3}

        #Username text
        Label:
            text: "Username"
            font_size: 25
            pos_hint: {'center_x': 0.095, 'center_y': 0.26}

        #Username text box
        TextInput:
            multinline: False
            size_hint: (0.15, 0.03)
            pos_hint: {'center_x': 0.22, 'center_y': 0.26}

        #Password text
        Label:
            text: "Password"
            font_size: 25
            pos_hint: {'center_x': 0.095, 'center_y': 0.22}

        #Password text box
        TextInput:
            multinline: False
            password: True
            size_hint: (0.15, 0.03)
            pos_hint: {'center_x': 0.22, 'center_y': 0.22}

```



```
        #Add button
        Button:
            text: "Add"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.115, 'center_y': 0.14}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

        #Remove button
        Button:
            text: "Remove"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.235, 'center_y': 0.14}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

        #Remote access background box
        Label:
            pos_hint: {'center_x': 0.825, 'center_y': 0.45}
            size_hint: (0.25, 0.7)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Remote access title
        Label:
            text: "Remote Access"
            font_size: 40
            pos_hint: {"center_x": 0.825, "center_y": 0.75}
```



```
        #Remote access underdevelopment title
        Label:
            text: "Coming Soon..."
            font_size: 60
            pos_hint: {"center_x": 0.825, "center_y": 0.5}
```

Review

In this iterative stage I have developing the graphical user interface of my greenhouse system. The layout of the GUI is complete and should not require any modifications unless further into development I deem it necessary to as some more features. The GUI is responsive to different screen sizes and is cross platform compatible with any device that can run python. The remaining focus of this project will be to develop the back end so that the greenhouse and the GUI work together displaying the correct information and carrying out the correct functions for the plan environment.

Iterative Stage 8 – Login

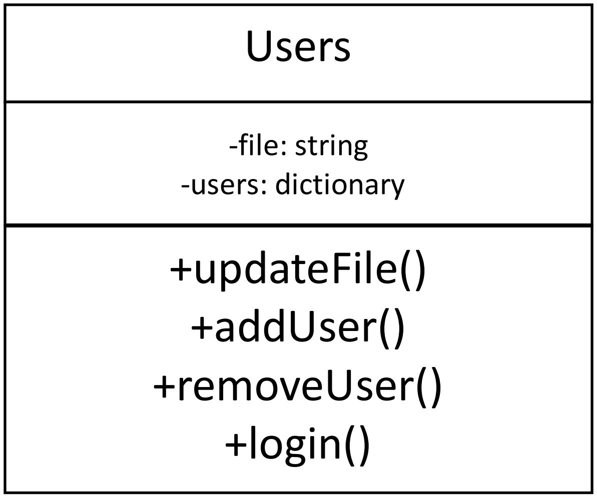
Overview

In this iterative stage I will be developing the login section of my project. The main aims of this stage will be to produce class with the ability to validate user details and to add new users. The login must be validated to check the user exists and ensure that the password is dealt with securely. I will be using the hash lib library on python to deal with hashing the password.

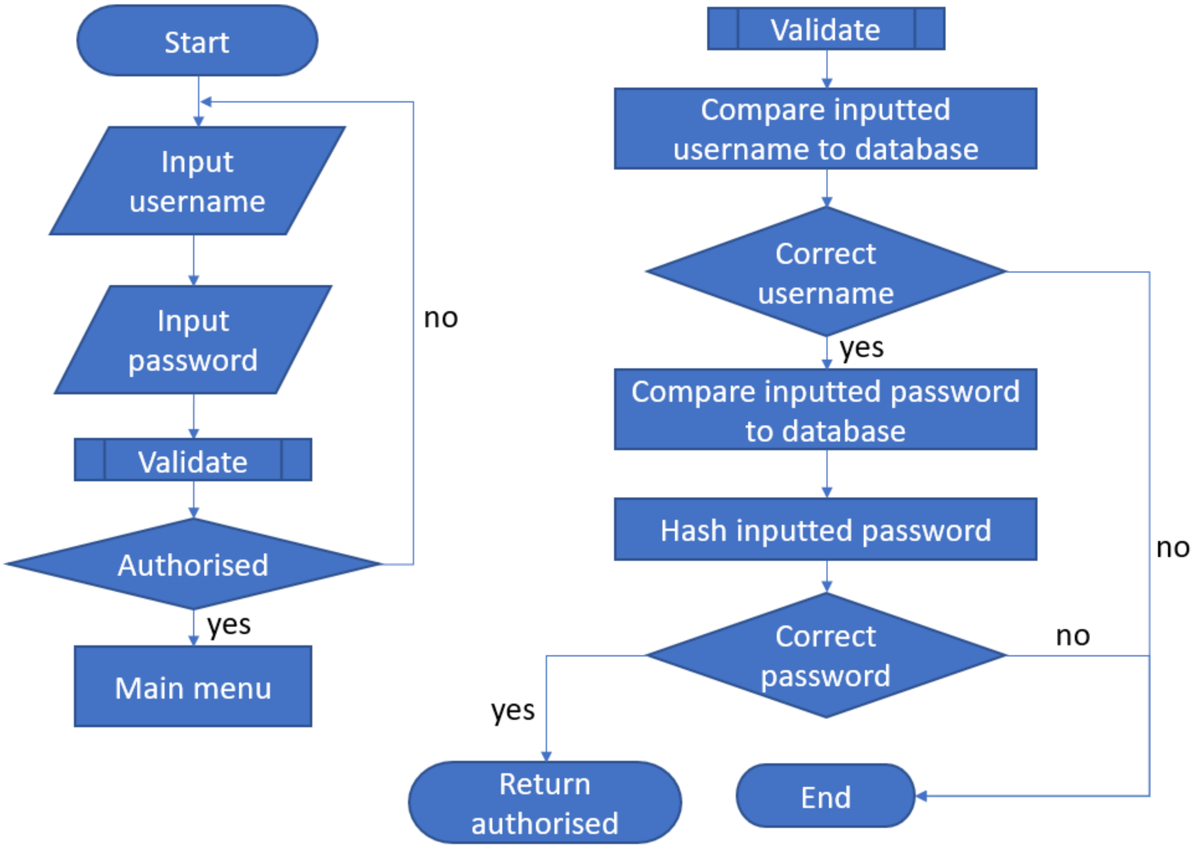
Requirements

All passwords will be stored in hashed form this will mean even if somebody gets hold of the users file there is no way for them to read the users passwords. In my greenhouse project all users will access the same data so there are no requirements for different user environments. Once a user logs in they have access to the same interface and data as all other users. The user details will be stored in a text file inside the same directory as the main system files. I have decided that all validation of user inputted data will be carried out on the side of the kivy class. So, the user’s class will not be responsible for making sure the user’s password is long enough and other validation requirements this will be handled inside the login kivy screen class.

Class diagram

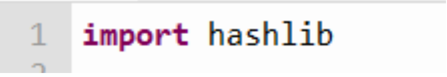


Flow chart



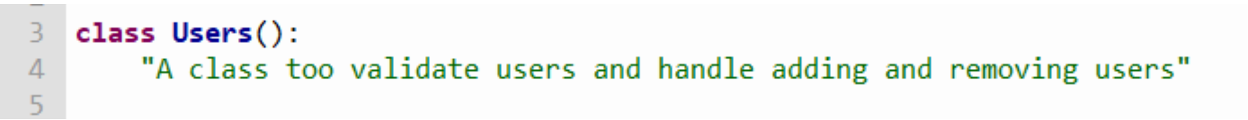
Data Structure	Data Type	Scope	Purpose	Validation required
File	String	Local	Stores the file path of the user's text file	
users	Dictionary	Local	Store all the users and the associated passwords	

Development Log



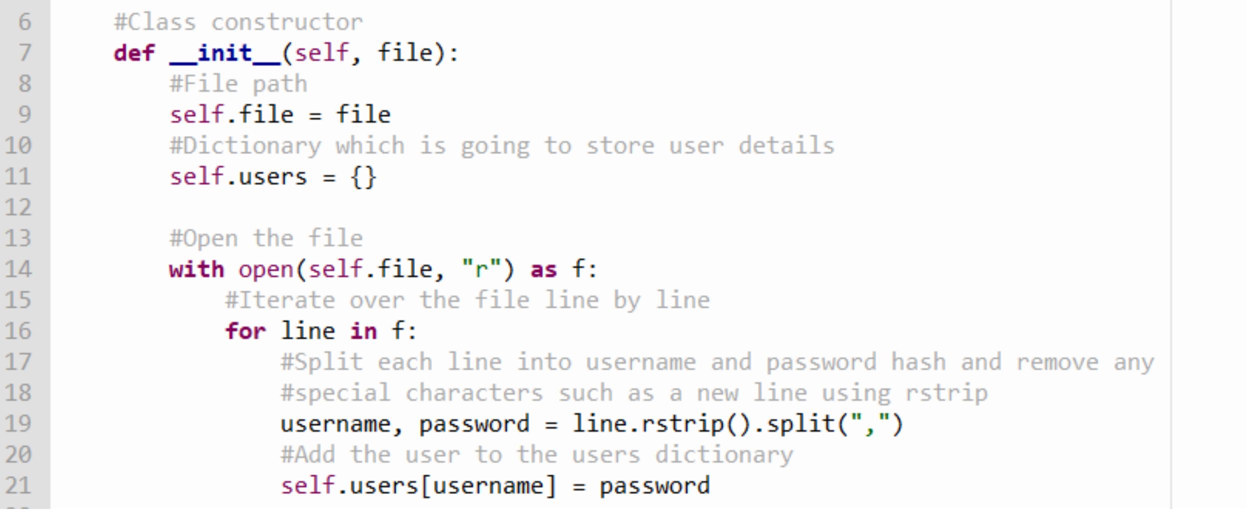
```python
import hashlib
```

In the user's class I will be using the hashlib library to handle the hashing of passwords. A hash is a one way encoding of data which cannot be undone. This allows me to store the users' passwords without much security as even If a malicious party got hold of the file, they would only see the hash of the password. Since this means I cannot decode the hash to compare it to the password which the user enters on login I will have to hash the password which the user enters and compare this to the stored hash for that user inside the user's file.



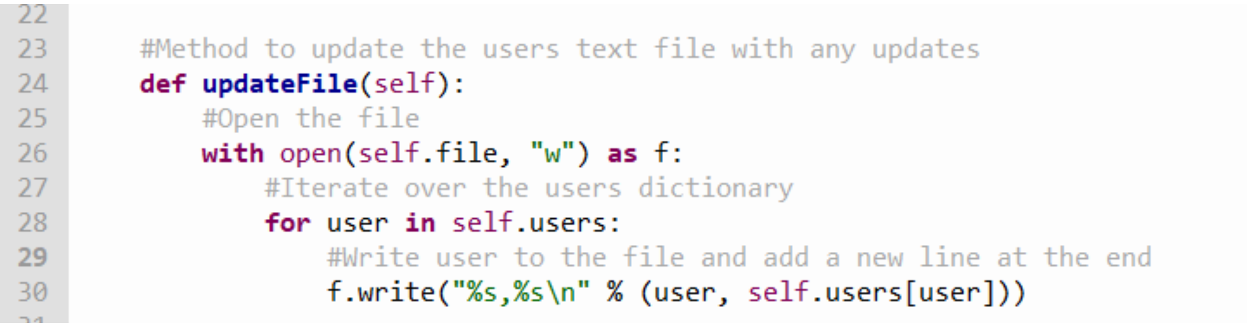
```python
class Users():
    "A class too validate users and handle adding and removing users"
```

The purpose of the user's class is to validate user logins and also to handle adding and removing users from the users' file.



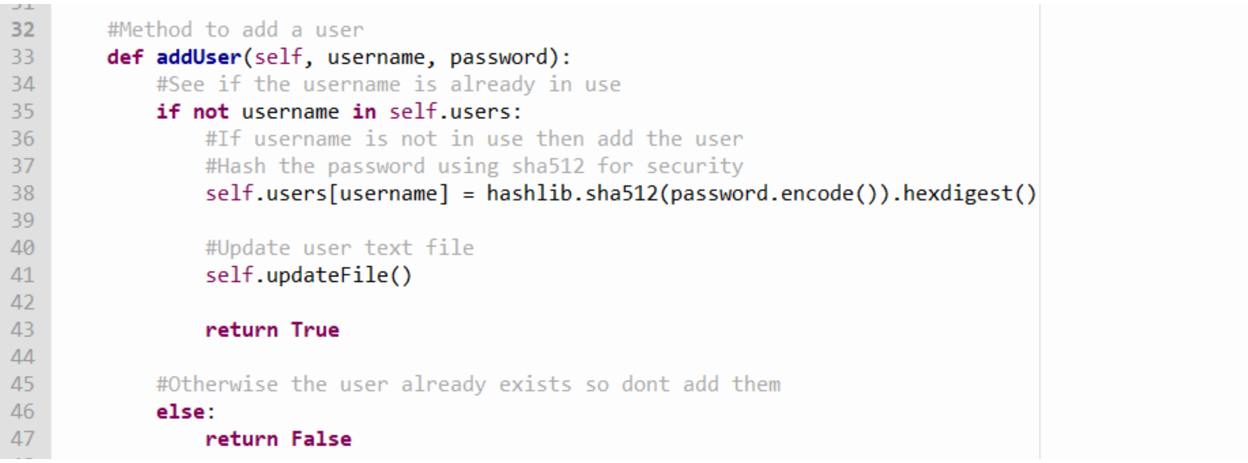
```python
    #Class constructor
    def __init__(self, file):
        #File path
        self.file = file
        #Dictionary which is going to store user details
        self.users = {}

        #Open the file
        with open(self.file, "r") as f:
            #Iterate over the file line by line
            for line in f:
                #Split each line into username and password hash and remove any
                #special characters such as a new line using rstrip
                username, password = line.rstrip().split(",")
                #Add the user to the users dictionary
                self.users[username] = password
```

When the class is initialized, it is passed the path of the user's file. The users file is then opened with the identifier f. A loop then iterates over all the lines inside the file. Each line in the users file consists of the username and the password for that user stored in a hashed form separated by a comma. First, I have used rstrip to remove any special characters specifically in this case we are concerned about removing the "\n" new line character at the end of each line. Once this has been removed the line is then split about the comma and each value either side of the comma assigned to a variable. The username is then added to the dictionary as a key with the value being the user's password hash. Once this loop is complete there is a dictionary called users which stores all the user details inside the users' file.



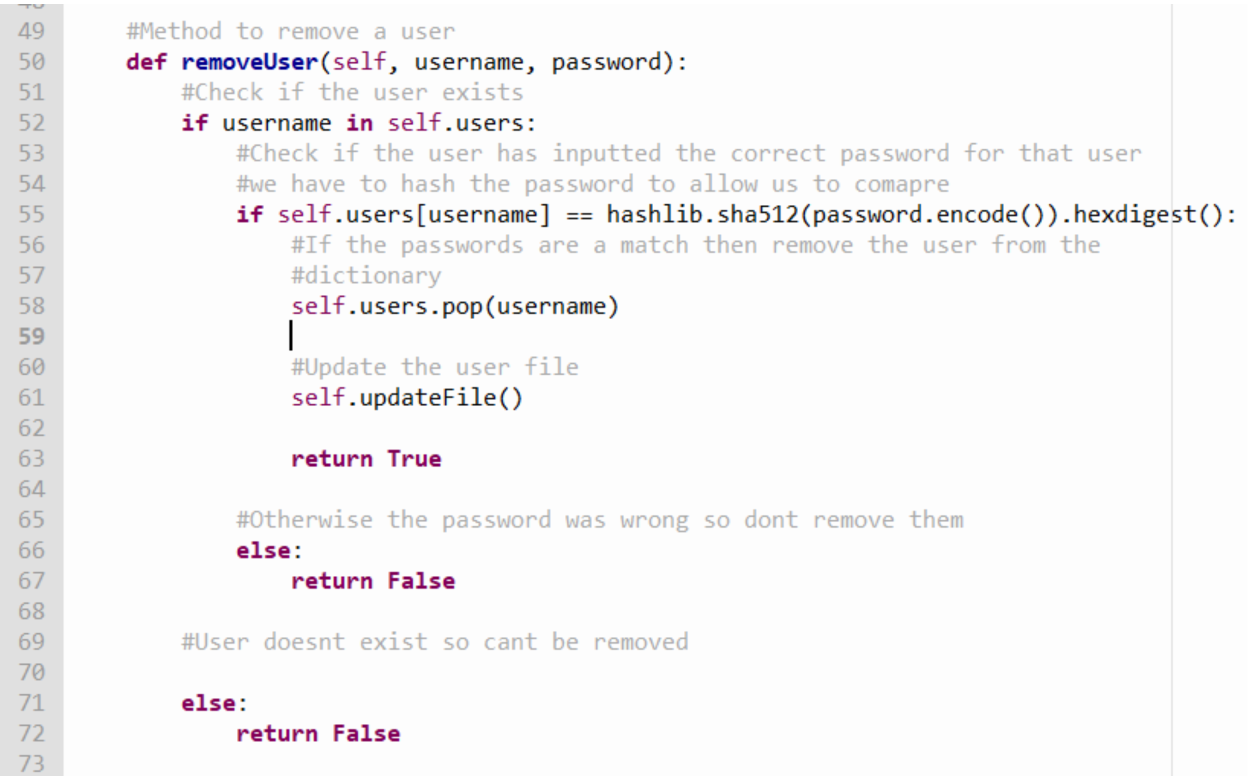
```python

    #Method to update the users text file with any updates
    def updateFile(self):
        #Open the file
        with open(self.file, "w") as f:
            #Iterate over the users dictionary
            for user in self.users:
                #Write user to the file and add a new line at the end
                f.write("%s,%s\n" % (user, self.users[user]))
```

When a user is added or removed the change will be made to the classes user's dictionary. However, this will not mean the change has been saved into the user's text file. The job of the updatefile method is to write the contents of the user's dictionary to the user's text file. This way there will be no differences between the two. This class will only be called when a user has been added or removed from our users' group. The method begins by opening the users text file but this time in write mode. The write mode means that we overwrite all data in the file. Next a loop goes through all the keys inside the user's dictionary. For each key a line is written to the file with the user going first then a comma followed by the users recorded password hash. A new line is also included using the "\n" character so that users each have their own line in the text file.



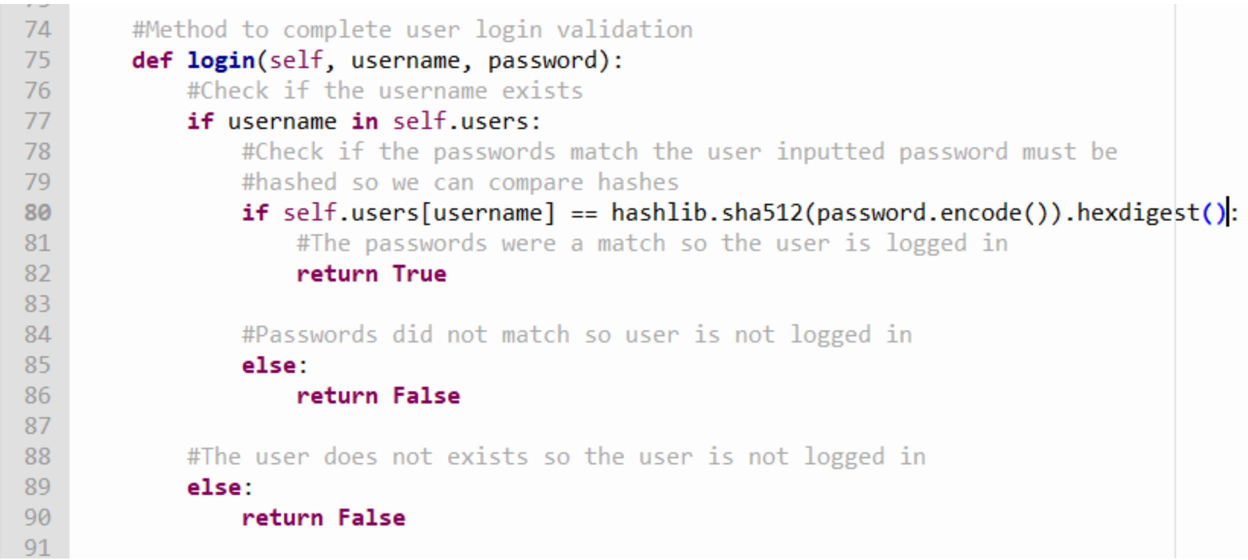
```
32     #Method to add a user
33     def addUser(self, username, password):
34         #See if the username is already in use
35         if not username in self.users:
36             #If username is not in use then add the user
37             #Hash the password using sha512 for security
38             self.users[username] = hashlib.sha512(password.encode()).hexdigest()
39 
40             #Update user text file
41             self.updateFile()
42 
43             return True
44 
45         #Otherwise the user already exists so dont add them
46         else:
47             return False
```

When adding a user to the group of users two parameters are required the username of the new user and the plain text version of their desired password. First there is a check on line 35 to make sure that the username is not already in use as there can only be one user with a specific username. To do this check I have checked to see if the username is in the user's dictionary and then used the NOT keyword so that the selection is only carried out if the user does not exist. Providing the username is not already in use a new user is created with a value of the password after it has been salted. In the case a user has been added the updateFile method is called so that the changes made to the dictionary are also reflected inside the user's text file. I have then returned true so that I am able to confirm if a user has been added or not. In the case a user is not added then false is returned.



```
49     #Method to remove a user
50     def removeUser(self, username, password):
51         #Check if the user exists
52         if username in self.users:
53             #Check if the user has inputted the correct password for that user
54             #we have to hash the password to allow us to comapre
55             if self.users[username] == hashlib.sha512(password.encode()).hexdigest():
56                 #If the passwords are a match then remove the user from the
57                 #dictionary
58                 self.users.pop(username)
59 
60                 #Update the user file
61                 self.updateFile()
62 
63                 return True
64 
65             #Otherwise the password was wrong so dont remove them
66             else:
67                 return False
68 
69         #User doesnt exist so cant be removed
70 
71         else:
72             return False
73 
```

The remote user method has the same parameters as the previous add user class. For this method, the given password must match the password of the user that is being removed for the action to be completed. First there is a conditional If statement to see if the user exists. As long as the user exists, we then check to see if the given password as a parameter of the method matches the stored password of the user. As long as the password hashes match and the user exist the user is removed from the dictionary and the users text file updated. In the case that the user either doesn't exist or that the wrong password was given then false is returned.



```
    #Method to complete user login validation
    def login(self, username, password):
        #Check if the username exists
        if username in self.users:
            #Check if the passwords match the user inputted password must be
            #hashed so we can compare hashes
            if self.users[username] == hashlib.sha512(password.encode()).hexdigest():
                #The passwords were a match so the user is logged in
                return True

            #Passwords did not match so user is not logged in
            else:
                return False

        #The user does not exists so the user is not logged in
        else:
            return False
```

The method that will be used most from the user's class is the login method. This will be used when the user logs in. The job of this method is to compare a given username and password against the stored usernames and passwords and either log the user in or deny access. The two parameters are the username which will come from the username box in the kivy login screen and the password which is also captured from the kivy login screen password box. The method then sees if the user exists and validates the user by comparing the saved hash of the password and the inputted password from the user. If the user Is logged in, then the Boolean True is returned otherwise False is returned. No indication is given as to if the issue was with the user's password or there username in the event that login fails this is for security reasons.

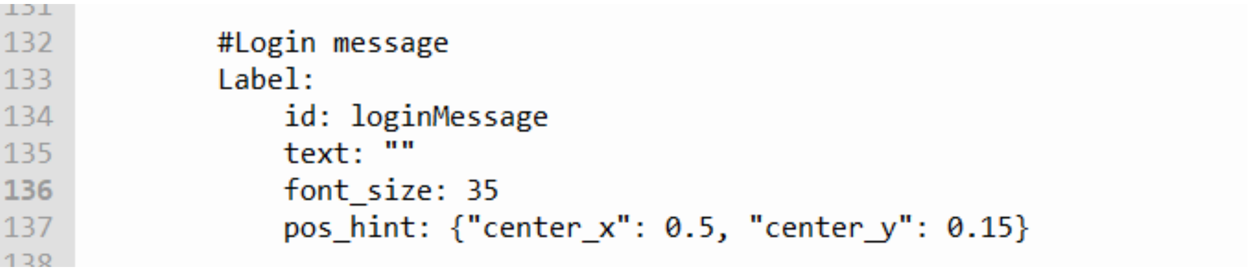
Test plan – Users class

Test Number	**Test Plan**	**Expected Outcome**	**Actual Outcome**	**Pass/Fail**
1	Add a user	The user will be stored in the user's text file along with their password in hashed form on a new line	The user was added and stored in the user's text file	Pass
2	Add a second user	Same as test number 1 but this is to check that a new line is being added when the users are written into the file	The second user was added and stored correctly on a new line in the text file	Pass

3	Remove one of the added users	The user will be taken out of the user's text file	The user was removed from the group of users	Pass
4	Login using valid user details	True will be returned to indicate that the user details were correct	Login was successful	Pass
5	Try to login with invalid details	False will be returned to show the user has not been logged in	Login was unsuccessful	Pass
6	Try to add a user which already exists	The user won't be added again as the user already exists. False should be returned	The user was not added, and all other users were unaffected	Pass
7	Try to remove a user which does not exist	False will be returned and no other users will be removed from the group of users	No users were removed	Pass

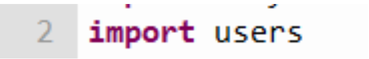
The testing plans has shown that the user's class is very robust and can handle the requirements of managing the users for the greenhouse GUI. It is now time to implement this class into the GUI so that the user can login.

Development log – Implementing into Gui

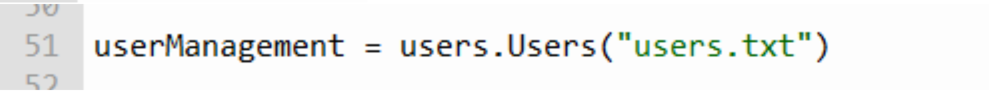


```
        #Login message
        Label:
            id: loginMessage
            text: ""
            font_size: 35
            pos_hint: {"center_x": 0.5, "center_y": 0.15}
```

When the user fails to login a message will need to be displayed informing them that there is an issue with the entered login details. To do this I have added a label which sits below the login button on the login screen. The label has an id of "loginMessage" to allow me to access its properties from inside python by referencing this id. The text property is initially empty as we will only be displaying a message to the user if they unsuccessfully login.

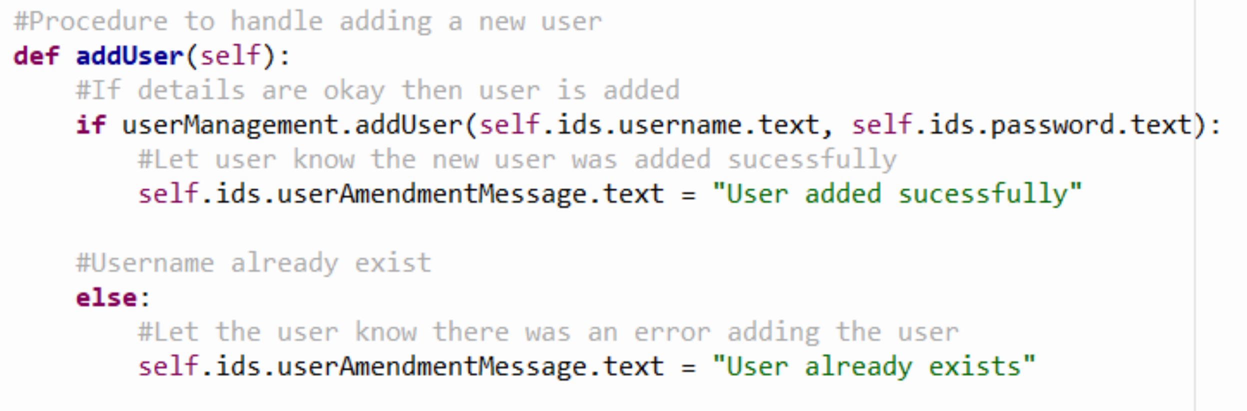


```
import users
```



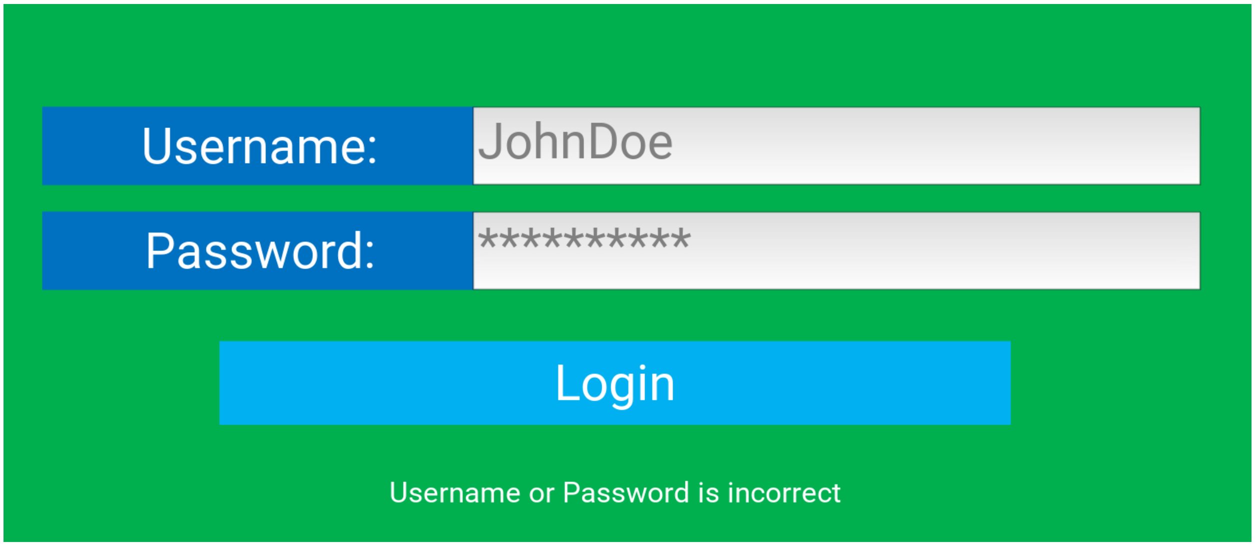
```
userManagement = users.Users("users.txt")
```

Inside the greenhouse.py python file I have imported the user's class which has just been developed. Further to this I have initialized an instance of this class with the users file passed upon initialization.



```
#Procedure to handle adding a new user
def addUser(self):
    #If details are okay then user is added
    if userManagement.addUser(self.ids.username.text, self.ids.password.text):
        #Let user know the new user was added sucessfully
        self.ids.userAmendmentMessage.text = "User added sucessfully"

    #Username already exist
    else:
        #Let the user know there was an error adding the user
        self.ids.userAmendmentMessage.text = "User already exists"
```

I have added a method inside the login screen class called check password. This will be called when the user selects the login button. To validate the users inputted details the login method of the user's class is called. This will evaluate as true if the details are correct and so I have passed to it the text inside the username and password text input box at the time when the user selects the login button. Kivy allows us to access the properties of elements using their id. In this case the element with the id username is belonging to the loginscreen class and getting its text property will return the current text inside the text input box. The same is done for the password text input box. To log the user in the screen manager is used to change the current screen to the main menu. If the user details inputted are not correct, then this is when we adjust the text property of the new element we added to inform the user that there login attempt was unsuccessful.

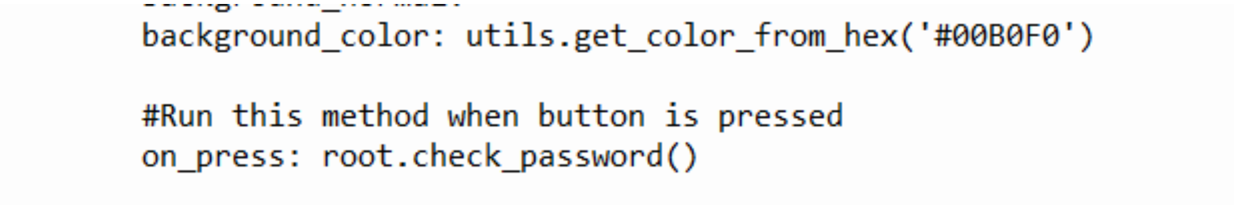


Username: JohnDoe

Password: **********

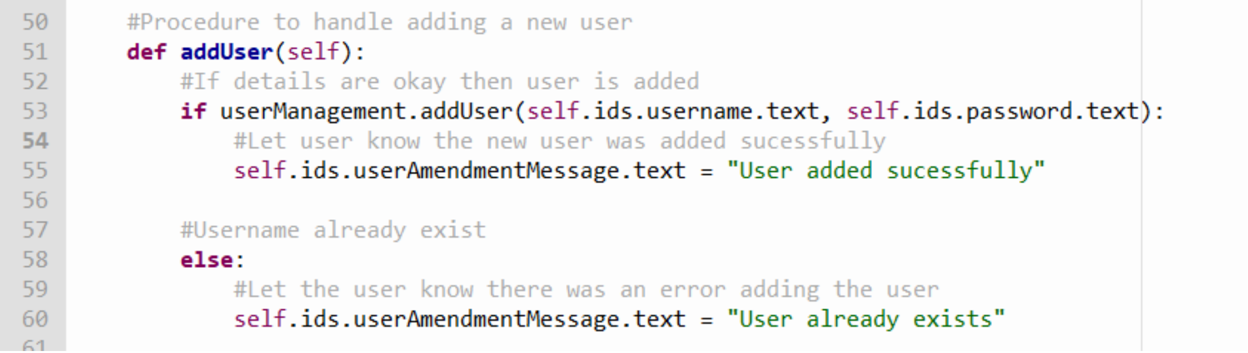
Username or Password is incorrect

Here you can see the message which is displayed to the user when the incorrect details are entered, and they try to login.

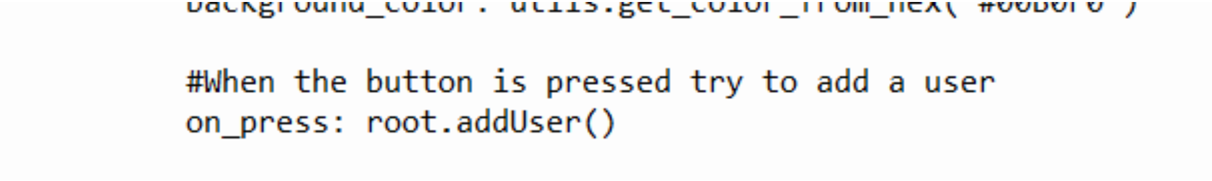


```
        background_color: utils.get_color_from_hex('#00B0F0')

        #Run this method when button is pressed
        on_press: root.check_password()
```

When the login button is pressed, I need the check password method to be called to do this I can use the on-press property in the login button element inside the kivy file. I have decided to save time that I will not include any validation of the username and password entered by the user.

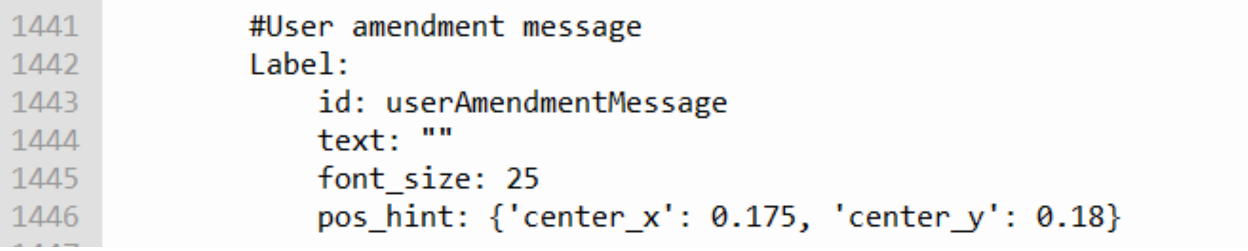


```
    #Procedure to handle adding a new user
    def addUser(self):
        #If details are okay then user is added
        if userManagement.addUser(self.ids.username.text, self.ids.password.text):
            #Let user know the new user was added sucessfully
            self.ids.userAmendmentMessage.text = "User added sucessfully"

        #Username already exist
        else:
            #Let the user know there was an error adding the user
            self.ids.userAmendmentMessage.text = "User already exists"
```



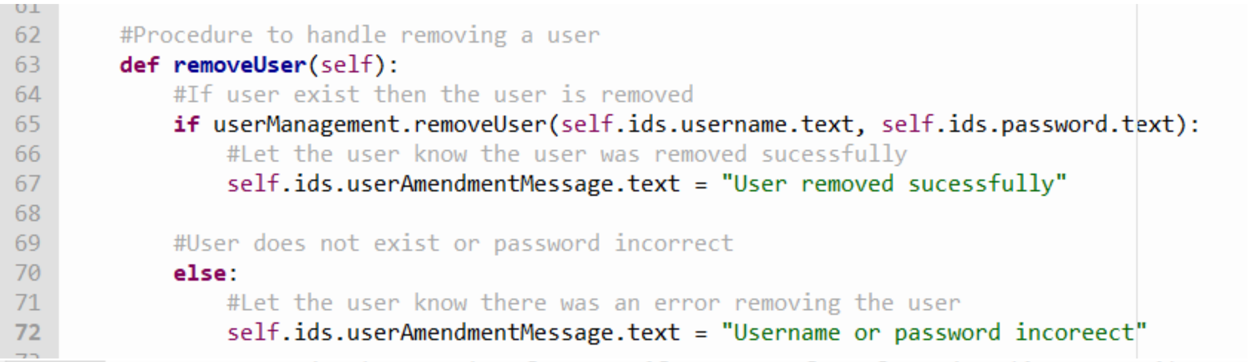
```
background_color: utils.get_color_from_hex('#00B0F0')

#When the button is pressed try to add a user
on_press: root.addUser()
```

Inside the settings screen the user can add and remove users from the group of authorized users. Here is the implementation for handling the add user event. When the user selects the add user button the add user procedure is called. We attempt to add the user using the details provided by the client. If this is successful, then a message is displayed to the user and otherwise if there is an issue we notify the user that the username already exists and so couldn't be added.

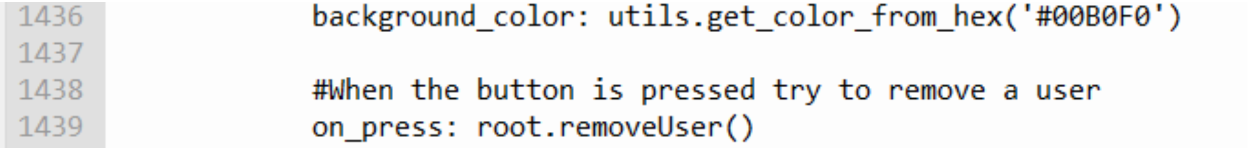


```
        #User amendment message
        Label:
            id: userAmendmentMessage
            text: ""
            font_size: 25
            pos_hint: {'center_x': 0.175, 'center_y': 0.18}
```

To display messages to the user regarding the success and failure of adding and removing users I have added a new label inside the user's section of the settings page. This label is initially blank and has an id of user amendment message to allow it to be accessed inside python.

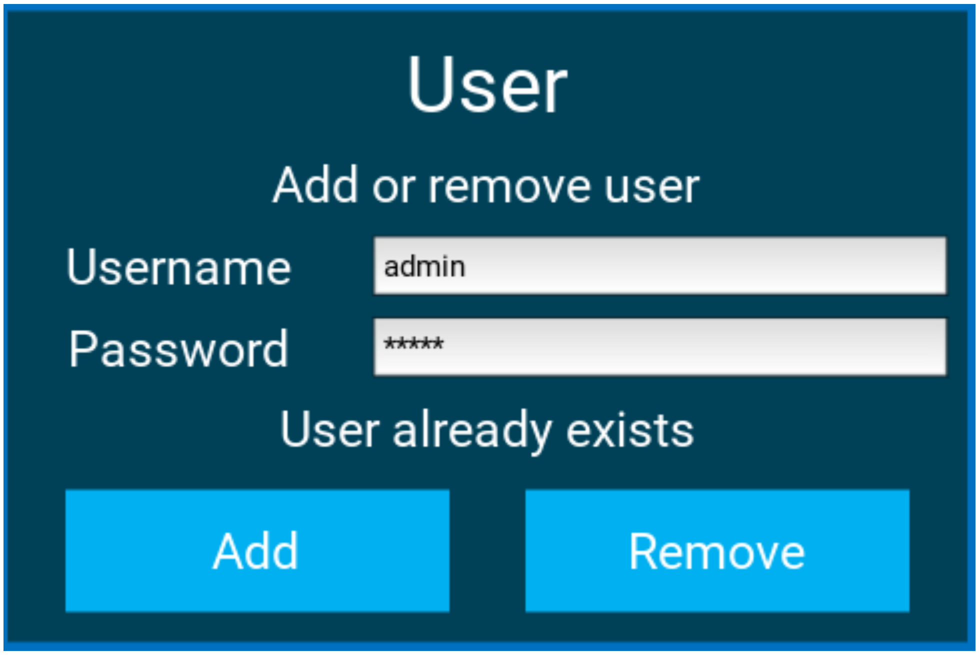


```
    #Procedure to handle removing a user
    def removeUser(self):
        #If user exist then the user is removed
        if userManagement.removeUser(self.ids.username.text, self.ids.password.text):
            #Let the user know the user was removed sucessfully
            self.ids.userAmendmentMessage.text = "User removed sucessfully"

        #User does not exist or password incorrect
        else:
            #Let the user know there was an error removing the user
            self.ids.userAmendmentMessage.text = "Username or password incoreect"
```



```
            background_color: utils.get_color_from_hex('#00B0F0')

            #When the button is pressed try to remove a user
            on_press: root.removeUser()
```

The process for removing a user is the same. A procedure called remove user is added inside the settings screen class which is called when the user clicks the remove button on the settings screen. This time if a user is added successfully, we let the user know using our label we just added or if the user does not exist or the password is incorrect so the user can't be removed we let them know that.


User

Add or remove user

Username: admin

Password: *****

User already exists

Above you can see the location of the message displayed to the user when they add or remove a user.

Test Plan

Test Number	Test Plan	Expected Outcome	Actual Outcome	Pass/Fail
1	Login using valid user details	Login should be successful	The user was logged in successfully	Pass
2	Login using invalid user details	Login should not be successful	The user was not logged in and an error message shown	Pass
3	Add a user using valid new user details	The user should be added	User was added successfully to the user's text file	Pass
4	Try adding a user which already exists	The user should not be added	The user was not added, and an error message was shown	Pass
5	Remove a current user	The user should be removed	The user was removed	Pass
5	Remove a user which does not exists	The user should not be removed	No users were removed, and an error message was shown to the user	Pass
6	Remove a user which exists but	The user should not be removed	The user was not removed, and an error shown	Pass

	enter an incorrect password			

Review

The login side of this project is now completed. The user can authenticate themselves to gain access to the greenhouse system and are also able to add new users and remove existing users. Due to time constrains I have not added any validation to the usernames and passwords which the user enters. In an ideal world I would have some restrictions on minimum password and username lengths along with requirements for including a special character and a capital letter in the user's password. However, the passwords are held securely using hashing so the login system is suitably secure.

Iterative stage 9 – Greenhouse Parameters

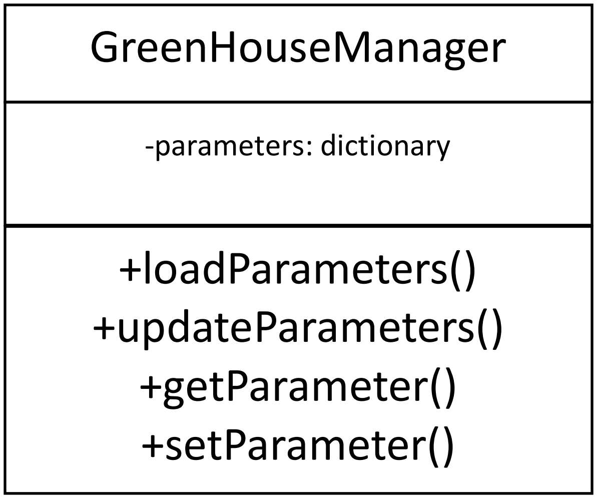
Overview

The greenhouse system will have 5 environmental parameters which will be controlled by the various devices inside the greenhouse. The user will be able to see the currently set parameter values and change these values on the parameters page. In this iterative stage I will be implementing the code to display the saved parameter values to the user and the code that will allow them to change these values. The parameters will be saved inside a text file.

Requirements

In this iterative stage I will begin development of the key class of this whole project called green house manager. This class will be responsible for controlling the greenhouse and managing all devices and settings.

Class Diagram



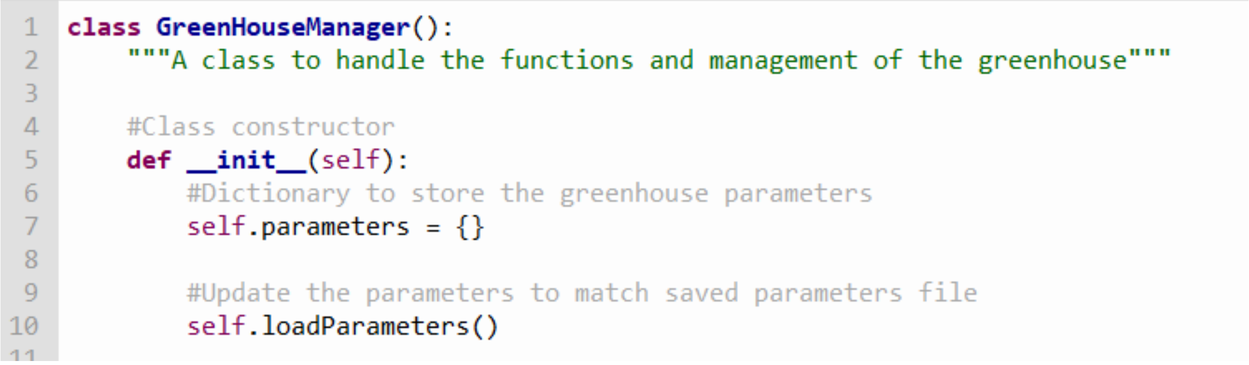
GreenHouseManager

-parameters: dictionary

+loadParameters()
+updateParameters()
+getParameter()
+setParameter()

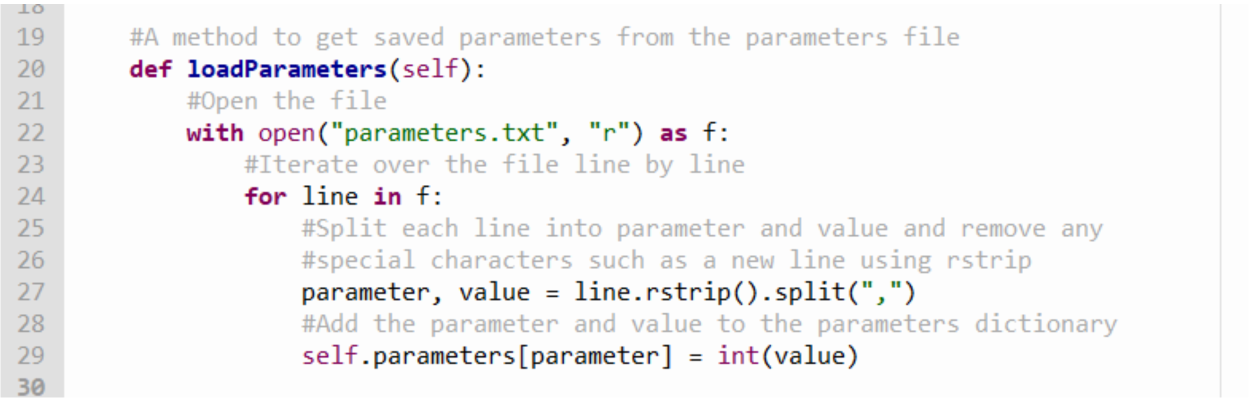
Development log

The green house manager class will extend far beyond managing the saved parameters however in this stage I will be developing just the parameters side of the class. The green house manager won't be able to control the greenhouse unless it knows the values it is trying to achieve in the greenhouse. Hence it would seem smart to begin developing the parameters functions first.



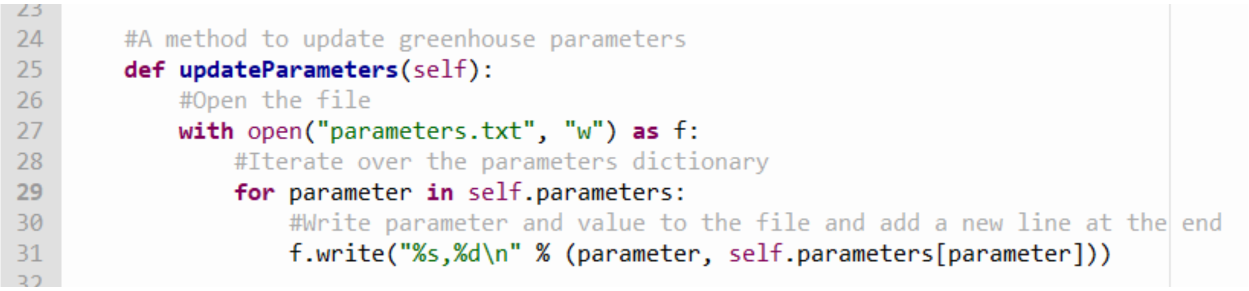
```
class GreenHouseManager():
    """A class to handle the functions and management of the greenhouse"""

    #Class constructor
    def __init__(self):
        #Dictionary to store the greenhouse parameters
        self.parameters = {}

        #Update the parameters to match saved parameters file
        self.loadParameters()
```

When the class is initialized, a dictionary is made which will store the parameters for the greenhouse. A method of this class called load parameters is ran which will be responsible for loading the parameters from the text file. This method will be developed in a moment.



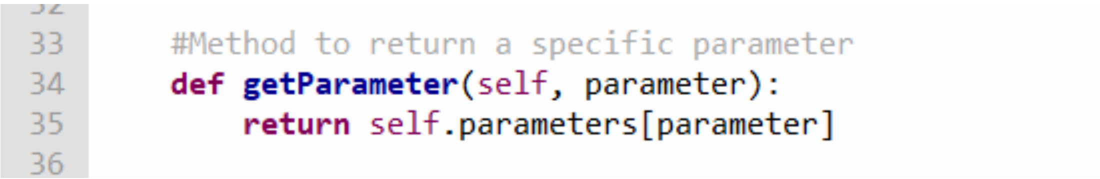
```
    #A method to get saved parameters from the parameters file
    def loadParameters(self):
        #Open the file
        with open("parameters.txt", "r") as f:
            #Iterate over the file line by line
            for line in f:
                #Split each line into parameter and value and remove any
                #special characters such as a new line using rstrip
                parameter, value = line.rstrip().split(",")
                #Add the parameter and value to the parameters dictionary
                self.parameters[parameter] = int(value)
```

The load parameters method works just the same as when we read the users text file. The method opens the text file in this case parameters.txt in read mode as the identifier f. Then a loop moves through the file line by line splitting each line at our designated split character in this case "," and then assigns the parameters and their values to the parameters dictionary. The advantage of opening the file using "with" is that once the nested code inside the "with" statement is complete the file is automatically closed. This just helps to avoid situations where the file is open twice or being written and read from at the same time which obviously will cause issues.



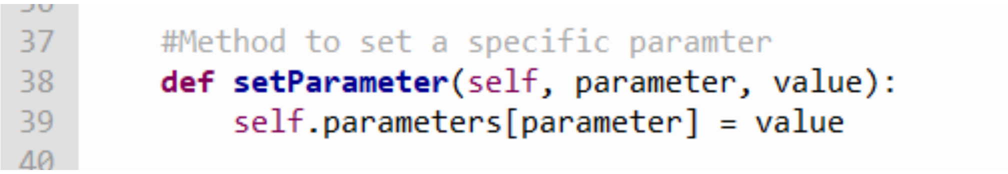
```
    #A method to update greenhouse parameters
    def updateParameters(self):
        #Open the file
        with open("parameters.txt", "w") as f:
            #Iterate over the parameters dictionary
            for parameter in self.parameters:
                #Write parameter and value to the file and add a new line at the end
                f.write("%s,%d\n" % (parameter, self.parameters[parameter]))
```

When the user has made changes to one or more of the greenhouse parameters then the parameters text file ill need to be updated to save these changes. This is the job of the update parameters method. The method does the opposite of the load parameters method by opening the file in write mode and writing the contents of the parameters dictionary to the file.



```
    #Method to return a specific parameter
    def getParameter(self, parameter):
        return self.parameters[parameter]
```

The getter method get parameter is responsible for returning the value of a specific parameter. The desired parameter is passed as an argument to this method and its corresponding value is returned.



```
    #Method to set a specific paramter
    def setParameter(self, parameter, value):
        self.parameters[parameter] = value
```

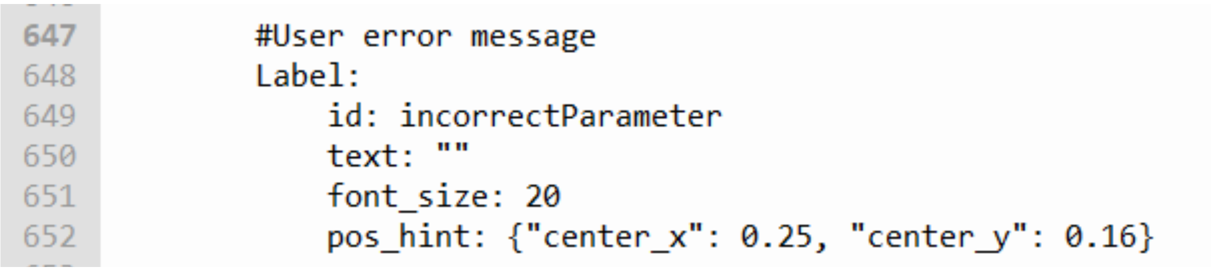
This setter method is responsible for changing the value of specific parameter. Both the parameter to be changed and its new value are passed as parameters.

Test plan – Green house manager class

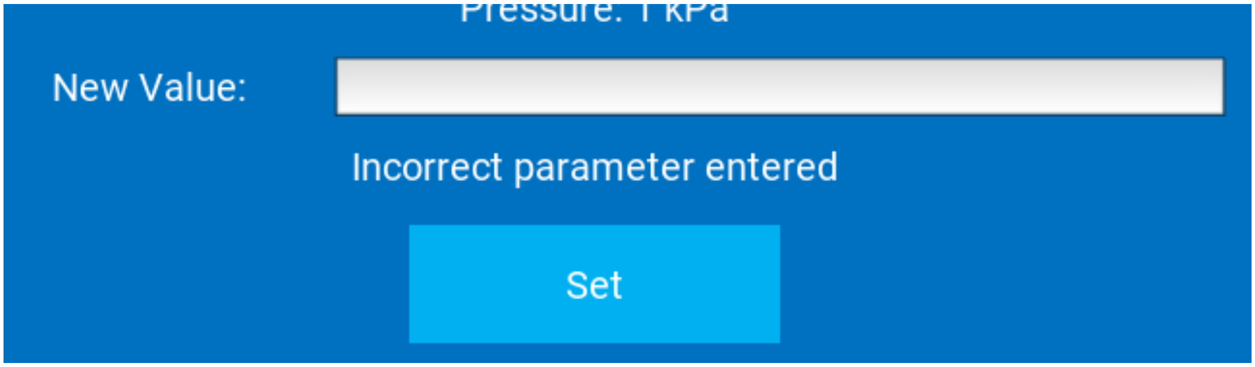
Test Number	Test Plan	Expected Outcome	Actual Outcome	Pass/Fail
1	load parameters	The parameters inside the text file should be loaded	The correct values were added to the parameters dictionary	Pass
2	Use the get parameter method to print out a parameter value	The value of the desired parameter which was passed to the method should be printed	The given parameters value was returned	Pass
3	Use the set parameter method to adjust the value of a parameter	That parameters value should be updated in the dictionary	The value was updated	Pass
4	Use the update parameters method to save the changes from test 3 to the text file	The changes should be saved into the text file	The changes were indeed saved	Pass

Development log – Implementing into kivy

The ability the edit the saved parameters and to view the current parameters now needs to be implemented into kivy.

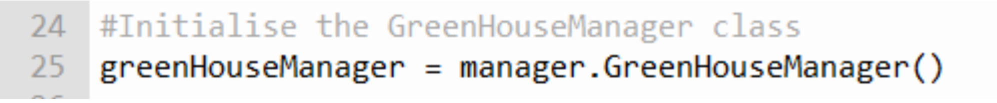


```
        #User error message
        Label:
            id: incorrectParameter
            text: ""
            font_size: 20
            pos_hint: {"center_x": 0.25, "center_y": 0.16}
```



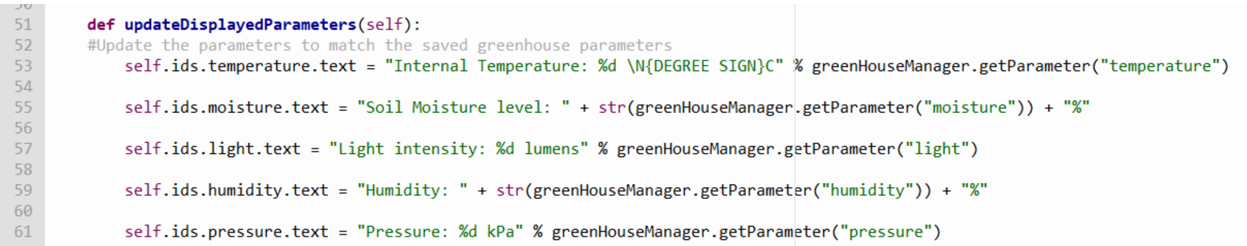
Incorrect parameter entered

I have added an error message into the parameters section of the gui. Just like for the login this error message is going to be used if the user enters some illegal data such as a string instead of a numerical parameter value.



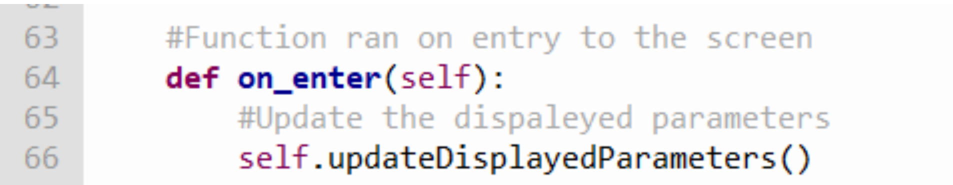
```
#Initialise the GreenHouseManager class
greenHouseManager = manager.GreenHouseManager()
```

The green house manager class has been initialized to allow us to use its methods.



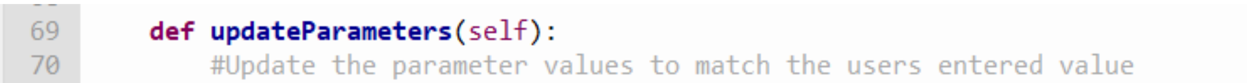
```
    def updateDisplayedParameters(self):
    #Update the parameters to match the saved greenhouse parameters
        self.ids.temperature.text = "Internal Temperature: %d \N{DEGREE SIGN}C" % greenHouseManager.getParameter("temperature")

        self.ids.moisture.text = "Soil Moisture level: " + str(greenHouseManager.getParameter("moisture")) + "%"

        self.ids.light.text = "Light intensity: %d lumens" % greenHouseManager.getParameter("light")

        self.ids.humidity.text = "Humidity: " + str(greenHouseManager.getParameter("humidity")) + "%"

        self.ids.pressure.text = "Pressure: %d kPa" % greenHouseManager.getParameter("pressure")
```

Inside the parameters screen class, I have made the method update displayed parameters. This class sets the text value of the labels responsible for showing the user the current parameter values. For each parameter I have set the corresponding label to be equal to the parameter value. Here I am using the getter method get parameter to get the value of each parameter.



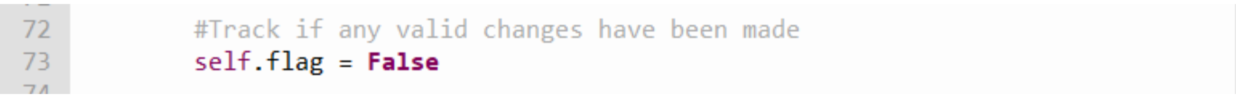
```
    #Function ran on entry to the screen
    def on_enter(self):
        #Update the dispaleyed parameters
        self.updateDisplayedParameters()
```

When the kivy screen manager transitions into a different screen it automatically calls a procedure called on enter. In the case of the parameters screen we want the parameter values shown to be updated each time the user goes to the screen so that they match the values of the saved parameters inside the text file. So, when this function is called, we run the update displayed parameters method so that the parameters are ensured to be up to date.



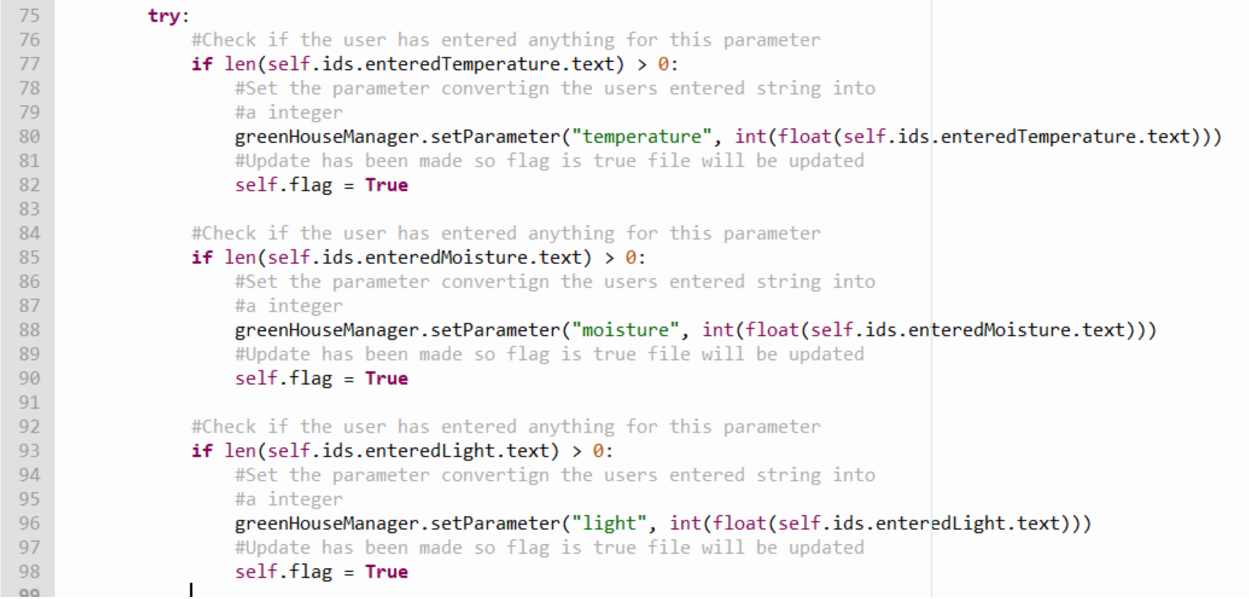
```
    def updateParameters(self):
        #Update the parameter values to match the users entered value
```

The parameters screen also gives the user the ability to enter new parameter values which the green house manager will then aim to keep inside the greenhouse. They can do this by entering new values inside text entry boxes and then selecting the save button. The update parameters method will be run when the user clicks the save button and will see if any valid changes have been made and if so will save these changes.

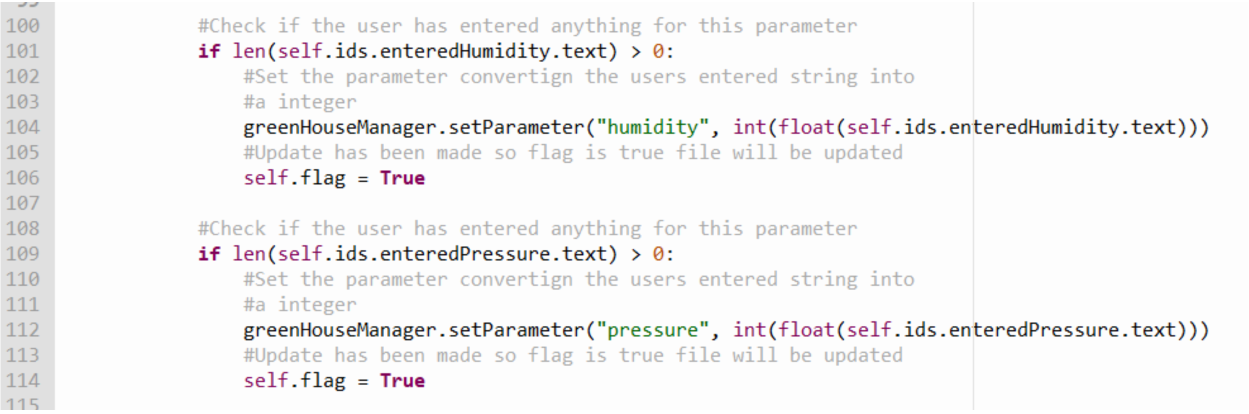


```
        #Track if any valid changes have been made
        self.flag = False
```

As saving to a file is time consuming, I am using a flag to check if any valid changes have been made by the user as if none have been made I can then avoid writing to the parameters file.

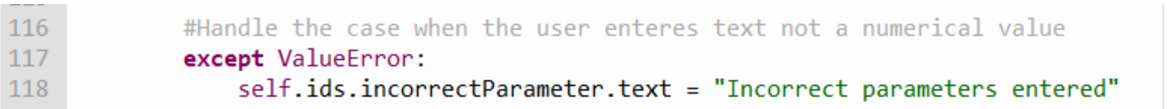


```
        try:
            #Check if the user has entered anything for this parameter
            if len(self.ids.enteredTemperature.text) > 0:
                #Set the parameter convertign the users entered string into
                #a integer
                greenHouseManager.setParameter("temperature", int(float(self.ids.enteredTemperature.text)))
                #Update has been made so flag is true file will be updated
                self.flag = True

            #Check if the user has entered anything for this parameter
            if len(self.ids.enteredMoisture.text) > 0:
                #Set the parameter convertign the users entered string into
                #a integer
                greenHouseManager.setParameter("moisture", int(float(self.ids.enteredMoisture.text)))
                #Update has been made so flag is true file will be updated
                self.flag = True

            #Check if the user has entered anything for this parameter
            if len(self.ids.enteredLight.text) > 0:
                #Set the parameter convertign the users entered string into
                #a integer
                greenHouseManager.setParameter("light", int(float(self.ids.enteredLight.text)))
                #Update has been made so flag is true file will be updated
                self.flag = True
```



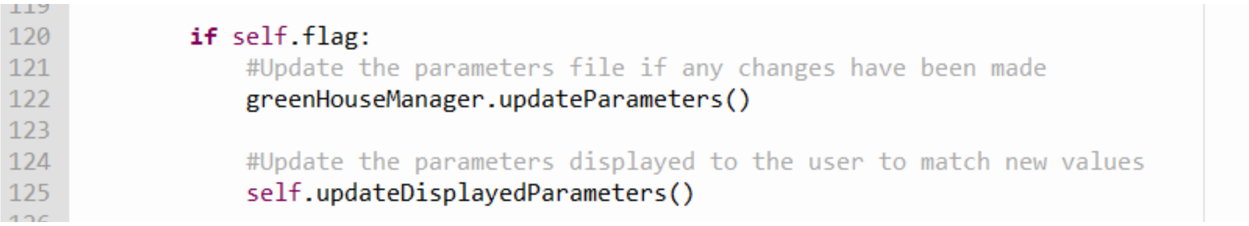
```
            #Check if the user has entered anything for this parameter
            if len(self.ids.enteredHumidity.text) > 0:
                #Set the parameter convertign the users entered string into
                #a integer
                greenHouseManager.setParameter("humidity", int(float(self.ids.enteredHumidity.text)))
                #Update has been made so flag is true file will be updated
                self.flag = True

            #Check if the user has entered anything for this parameter
            if len(self.ids.enteredPressure.text) > 0:
                #Set the parameter convertign the users entered string into
                #a integer
                greenHouseManager.setParameter("pressure", int(float(self.ids.enteredPressure.text)))
                #Update has been made so flag is true file will be updated
                self.flag = True
```

For each of the 5 parameter values there is a check to see if the user has entered anything inside its new value text box. This check is done by seeing if the length of the text entry box text parameter is larger than 0. If this is the case, then it is clear the user has entered a value. When the user enters a new parameter value, I use the green house manager setter method to update the value of that parameter inside the parameters dictionary to match the value inputted into the text box. As the text box records strings the text value must be changed from a string into an integer. For some unknown reason python doesn't seem to like converting the kivy text values straight into an integer so I've had to add an intermediary step of converting to a float to solve this issue. If a change is detected in any of the text input boxes the flag becomes true so that the file will be updated.



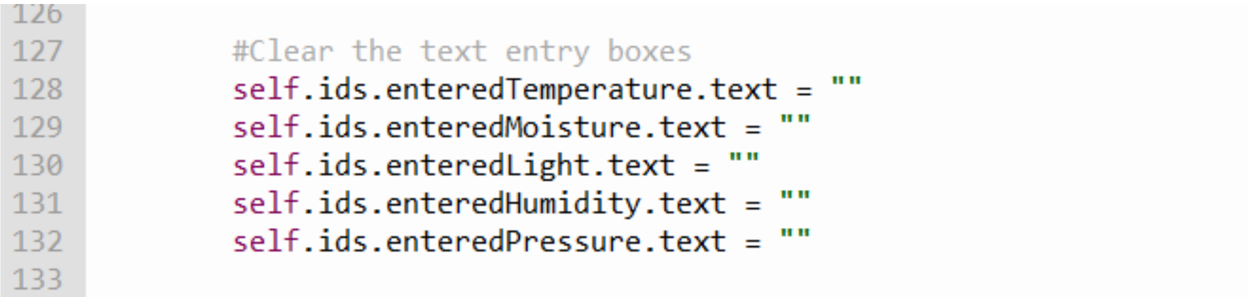
```
        #Handle the case when the user enteres text not a numerical value
        except ValueError:
            self.ids.incorrectParameter.text = "Incorrect parameters entered"
```

Of course, this introduces a case where the user could have entered a string into the text box and then when the program converts this into a float a value error will occur. When I say a string, I mean the user could have entered "lorry" as opposed to "100" of course both are strings but only the latter can be converted and represented as an integer. To handle this event a try except statement is used to catch the value error and display an error message to the user without interrupting the program flow.



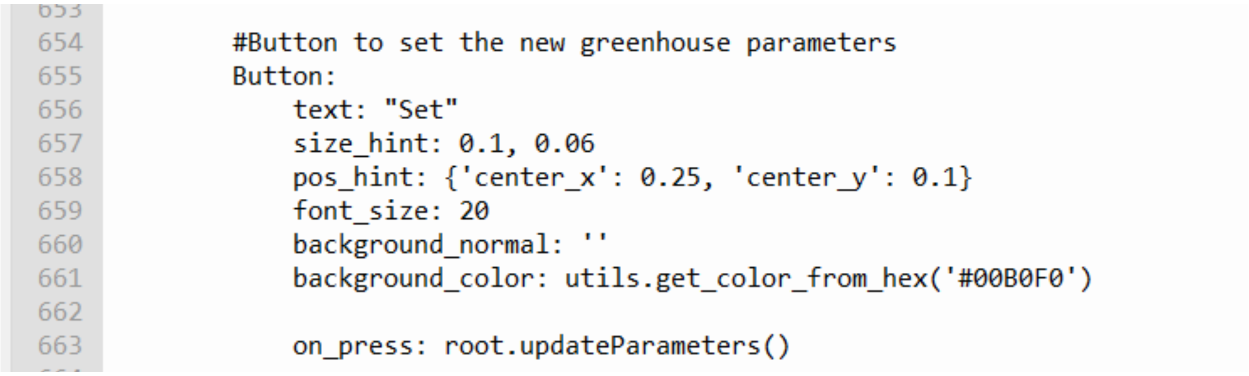
```
        if self.flag:
            #Update the parameters file if any changes have been made
            greenHouseManager.updateParameters()

            #Update the parameters displayed to the user to match new values
            self.updateDisplayedParameters()
```

If the flag is true, then a valid change has been made to one or more of the parameters. So, these changes will need to be saved into the parameters text file to do this the green house manager update parameters method is called which will write the current contents of the parameters dictionary into the text file for permeant storage. Once the changes have been saved the update displayed parameters method is called so that the parameters displayed on the screen are adjusted to match their new values.



```
        #Clear the text entry boxes
        self.ids.enteredTemperature.text = ""
        self.ids.enteredMoisture.text = ""
        self.ids.enteredLight.text = ""
        self.ids.enteredHumidity.text = ""
        self.ids.enteredPressure.text = ""

```

Finally, the 5 text entry boxes for each of the parameters are cleared so that they are ready for when the user next wants to enter a new parameter.



```
        #Button to set the new greenhouse parameters
        Button:
            text: "Set"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.25, 'center_y': 0.1}
            font_size: 20
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            on_press: root.updateParameters()
```

I have connected the update parameters method to the set button in the parameters page so that when its pressed all the actions described above are carried out to result in the parameters being saved.

Test Plan

Test Number	**Test Plan**	**Expected Outcome**	**Actual Outcome**	**Pass/Fail**
1	Open the GUI	The parameters should be loaded into the GUI and displayed to the user	The correct values were added shown on the parameters screen in kivy	Pass
2	Enter a new value for one of the parameters and set it	The new parameter value should be saved to the parameters file and the text	The value was saved and cleared from the text box and shown to the user	Pass

		box should be cleared and the new value should also be shown on the parameters page		
3	Try to enter a string such as “test” and set this as a value	An error message should be shown to the user and the value should not be saved to the file	The error message was shown asking the user to enter a number	Pass

Review
The greenhouse parameters are now saved inside a text file and loaded into kivy for the user to view and change. These parameters will be the target values which the greenhouse will try stick to. The system for setting values is robust and does not allow any invalid data to be entered.

Iterative stage 10 – Output devices

Overview
The second half of the parameters screen is where the user can turn on and off the different output devices inside the greenhouse and change the mode of these devices. Each device will have two modes manual and adaptive. In manual mode the device will always be on, and the greenhouse manager won’t turn it off if the parameter which it governs is exceeded. So, if the heating element is in manual mode it will always be on regardless of if the greenhouse surpasses the parameter set by the user. The second mode called adaptive is when the greenhouse manager will turn the device on and off to control the greenhouse environment. So, if the temperature gets to warm then the lamp goes off and the fan will open.

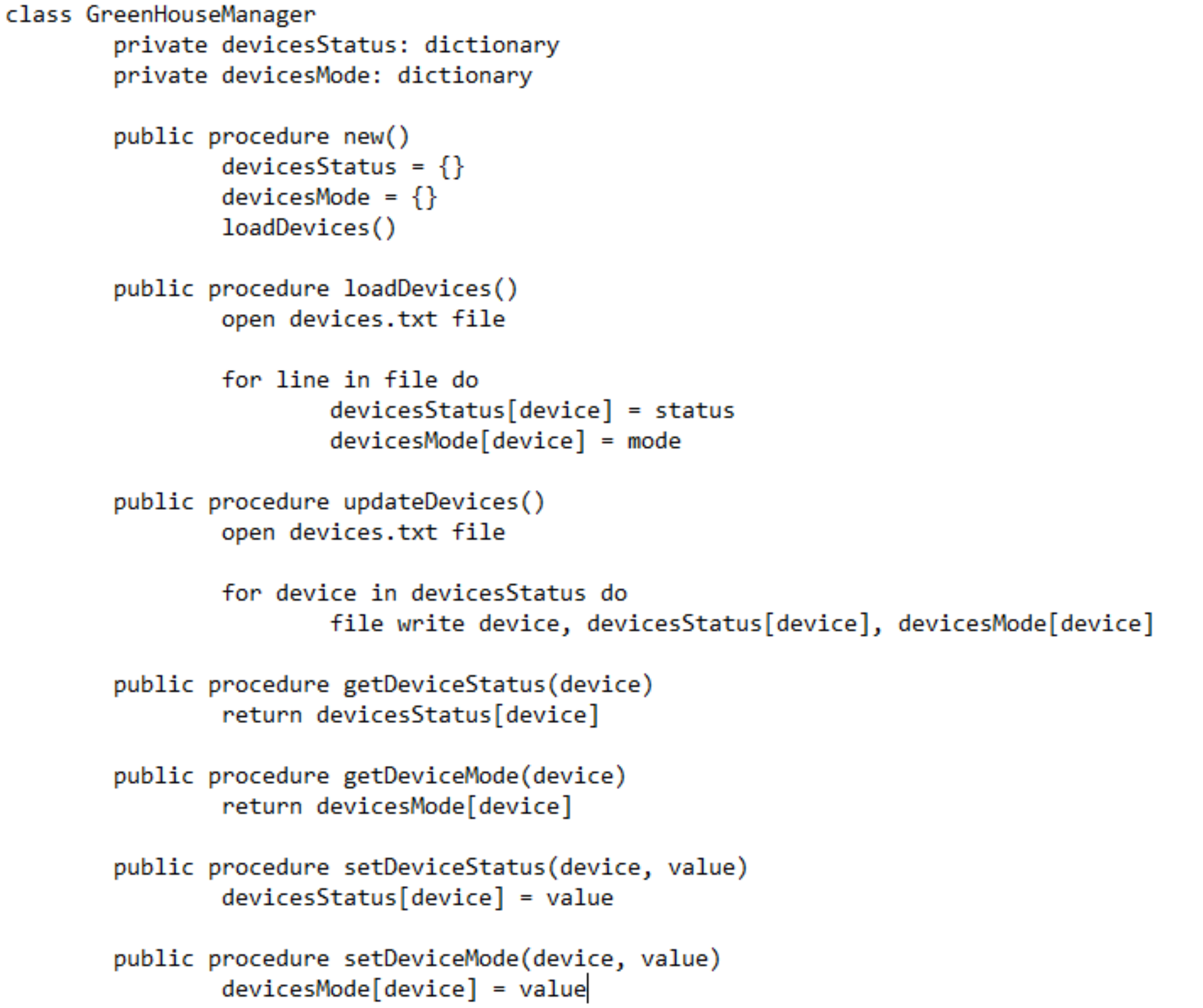
Requirements
In this stage I will be creating a file to save the status and mode of each output device. I will also be adding methods to the greenhouse manager class to load, update, get and set these different values. During this stage I will also be writing the code to make the on and off toggle buttons function.

Class Diagram

GreenHouseManager
-devicesStatus: dictionary -devicesMode: dictionary

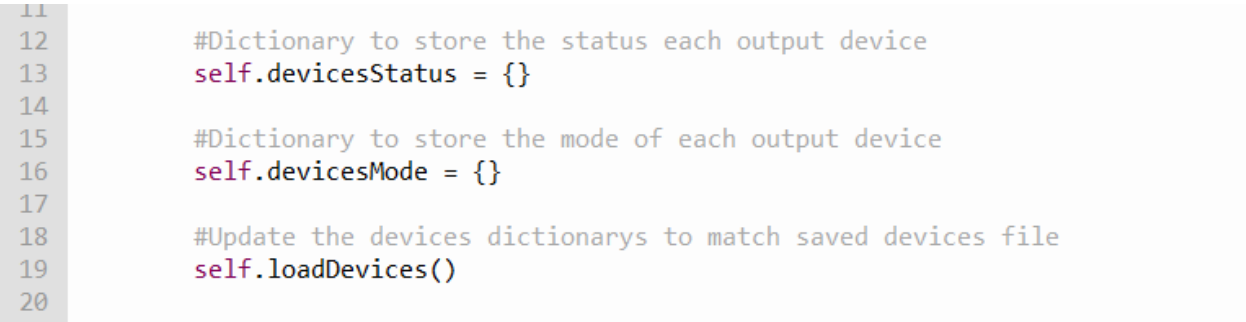
+loadDevices()
+updateDevices()
+getDeviceStatus()
+getDeviceMode()
+setDeviceStatus()
+setDeviceMode()

Pseudocode



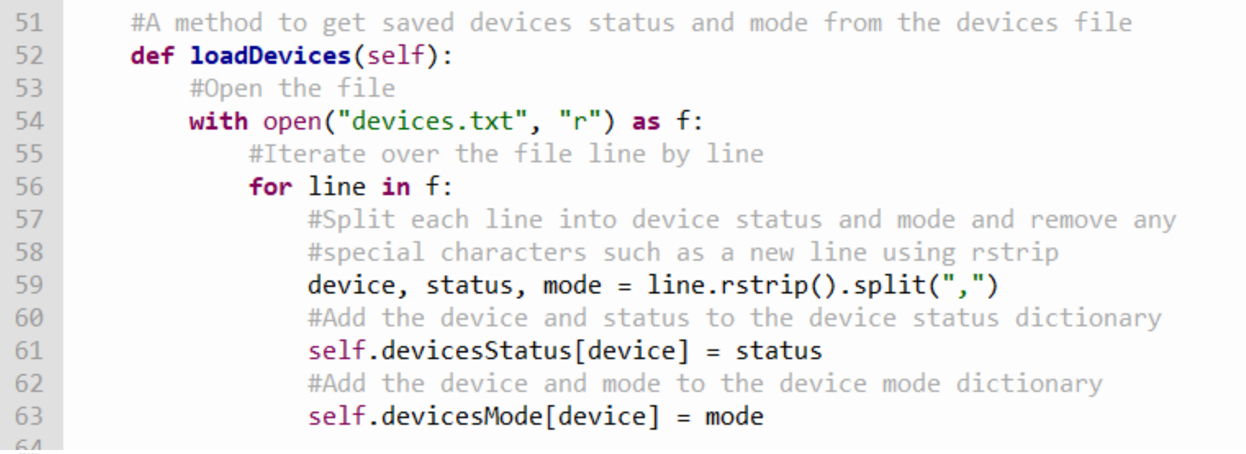
```
class GreenHouseManager
        private devicesStatus: dictionary
        private devicesMode: dictionary

        public procedure new()
                devicesStatus = {}
                devicesMode = {}
                loadDevices()

        public procedure loadDevices()
                open devices.txt file

                for line in file do
                        devicesStatus[device] = status
                        devicesMode[device] = mode

        public procedure updateDevices()
                open devices.txt file

                for device in devicesStatus do
                        file write device, devicesStatus[device], devicesMode[device]

        public procedure getDeviceStatus(device)
                return devicesStatus[device]

        public procedure getDeviceMode(device)
                return devicesMode[device]

        public procedure setDeviceStatus(device, value)
                devicesStatus[device] = value

        public procedure setDeviceMode(device, value)
                devicesMode[device] = value
```

Development log

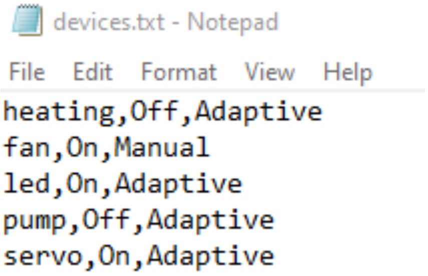


```
        #Dictionary to store the status each output device
        self.devicesStatus = {}

        #Dictionary to store the mode of each output device
        self.devicesMode = {}

        #Update the devices dictionarys to match saved devices file
        self.loadDevices()

```

Inside the green house manager class constructor, I have created two dictionaries which will be used to store the status of each device and also the mode of each device. After this I have called the method load devices which will then read the data from our devices file and add it to the two dictionaries. When I was deciding how to implement the data structures to store the devices status and mode, I considered using a single dictionary with a key equal to the device name and then an array as the value with the first value of the array holding the status and the second the mode. However, I elected against this approach as its wasn't obvious when accessing the data if you were getting the status or the mode unless you remembered that the status was index 0 and mode index 1. I feel this was using two dictionaries is more readable and will lead to less issues down the line.

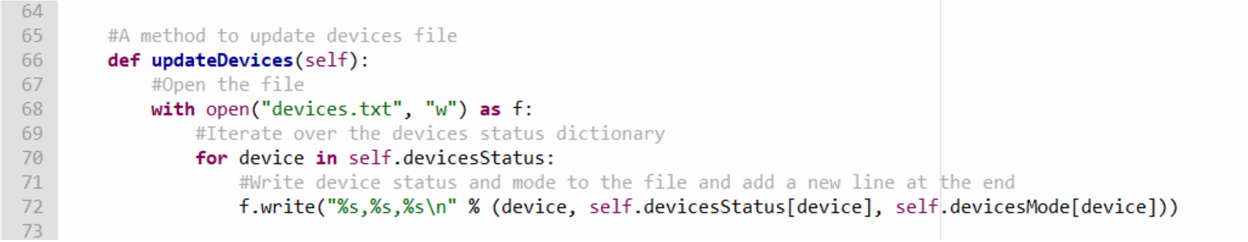


```
    #A method to get saved devices status and mode from the devices file
    def loadDevices(self):
        #Open the file
        with open("devices.txt", "r") as f:
            #Iterate over the file line by line
            for line in f:
                #Split each line into device status and mode and remove any
                #special characters such as a new line using rstrip
                device, status, mode = line.rstrip().split(",")
                #Add the device and status to the device status dictionary
                self.devicesStatus[device] = status
                #Add the device and mode to the device mode dictionary
                self.devicesMode[device] = mode
```



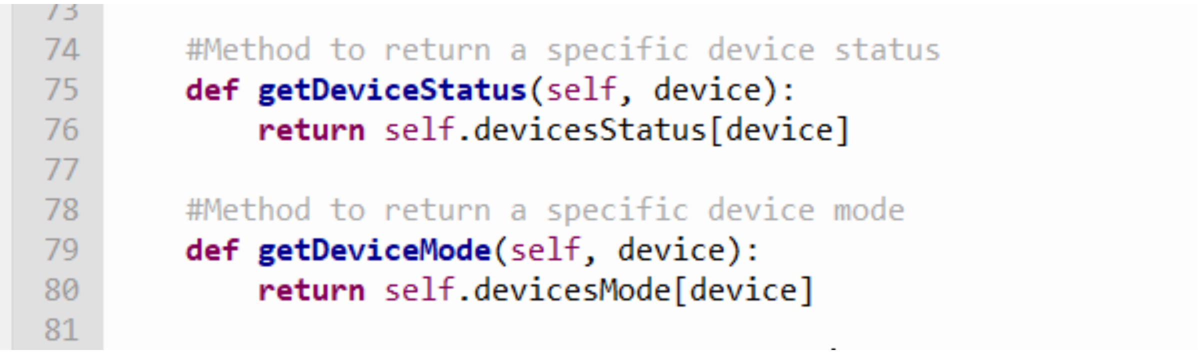
```
heating,Off,Adaptive
fan,On,Manual
led,On,Adaptive
pump,Off,Adaptive
servo,On,Adaptive
```

The load devices method needs to open the devices.txt file which I am using to store the status and mode of each device and add them to the status and mode dictionary. To store the data, I am using a text file just like with the parameters. Each value is separated by a comma and each different record is separated by a new line. In this case we are storing three values the name of the device its status and its mode. In the method the file is first opened in read mode with the identifier f. Then the file is iterated over line by line. Each time the line is split up at the designated special character and assigned to three variables. Finally, the devices status is added to the status dictionary with the device name as key and the same for the device mode in the mode dictionary. As I'm using with open there is no need to close the file this is automatically done at the end of that code block.



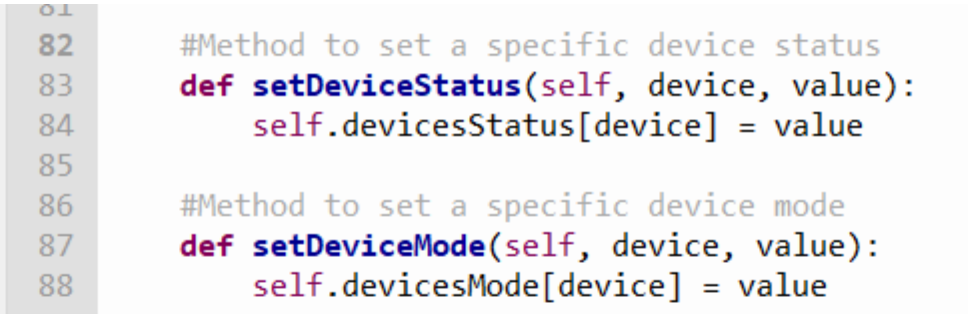
```
    #A method to update devices file
    def updateDevices(self):
        #Open the file
        with open("devices.txt", "w") as f:
            #Iterate over the devices status dictionary
            for device in self.devicesStatus:
                #Write device status and mode to the file and add a new line at the end
                f.write("%s,%s,%s\n" % (device, self.devicesStatus[device], self.devicesMode[device]))
```

When a change is made to a device mode or status the devices file will need to be updated so the change is saved into memory. The file is opened in write mode this time with the same identifier. Then we iterate over each key inside the devices status and write a new line for each device with the device, status and mode being written. Since each device has a status and mode it does not matter if we loop over the keys of the devices Status or devices mode dictionary as they both have the same number of identical keys.



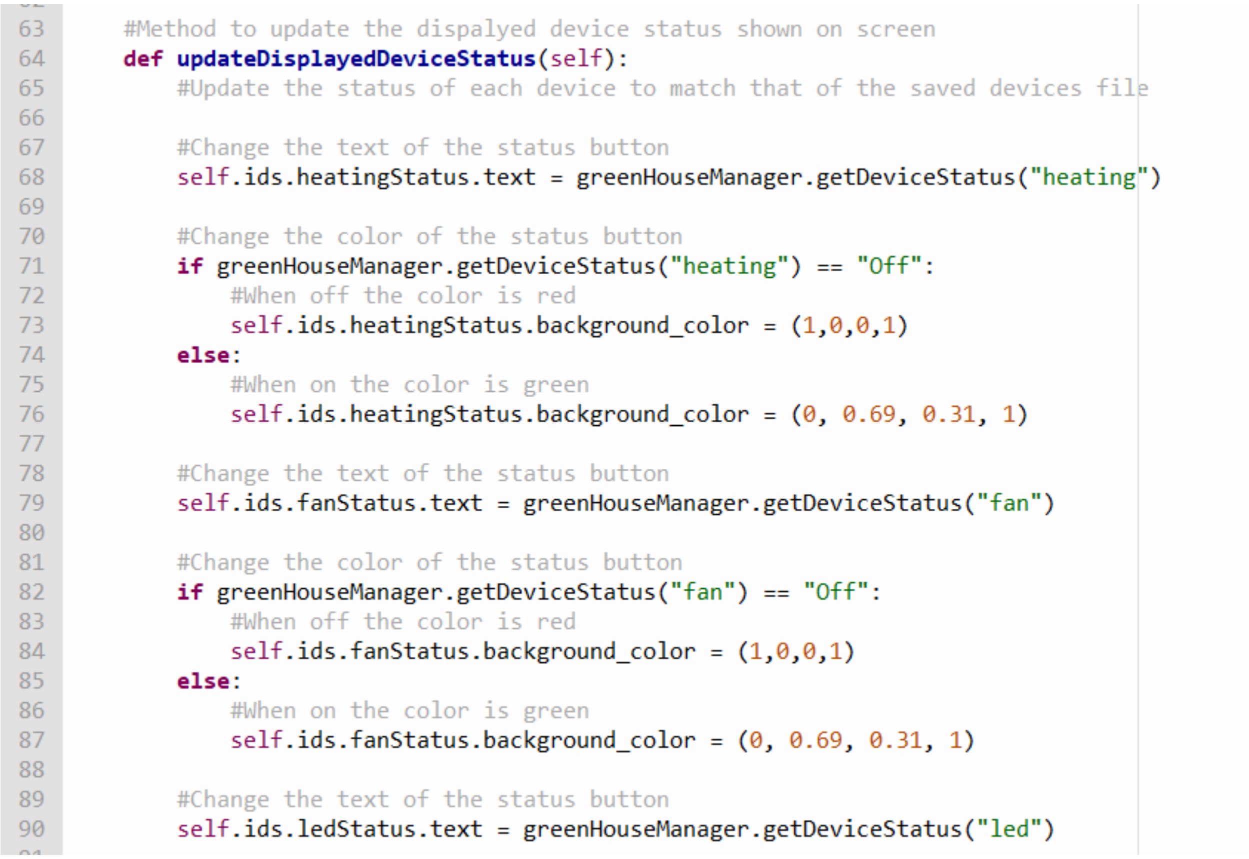
```
    #Method to return a specific device status
    def getDeviceStatus(self, device):
        return self.devicesStatus[device]

    #Method to return a specific device mode
    def getDeviceMode(self, device):
        return self.devicesMode[device]
```

The devices need two getter methods to return the value of a specific device's status and mode.

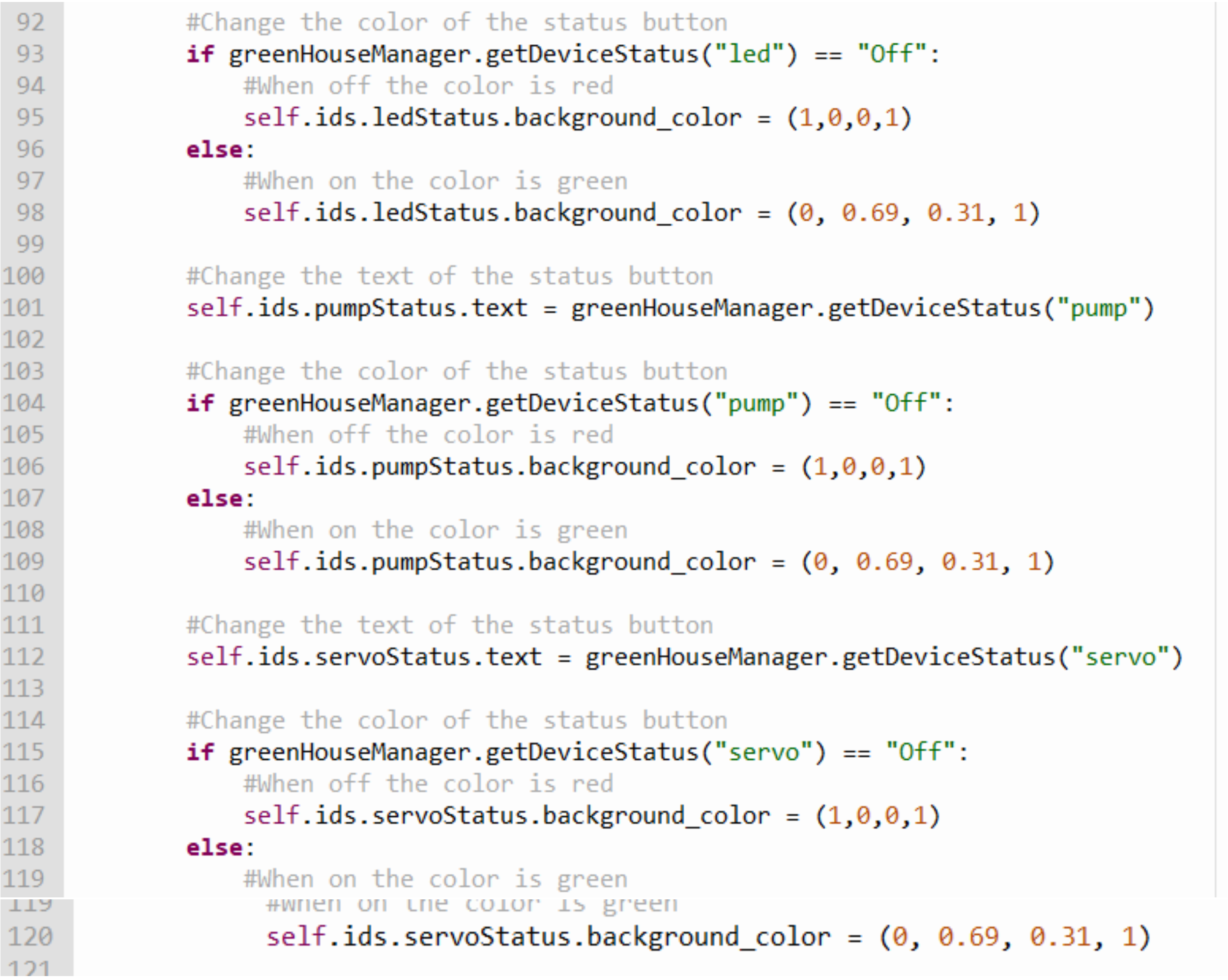


```
    #Method to set a specific device status
    def setDeviceStatus(self, device, value):
        self.devicesStatus[device] = value

    #Method to set a specific device mode
    def setDeviceMode(self, device, value):
        self.devicesMode[device] = value
```

Two setter methods are also implemented to set the values of different devices.

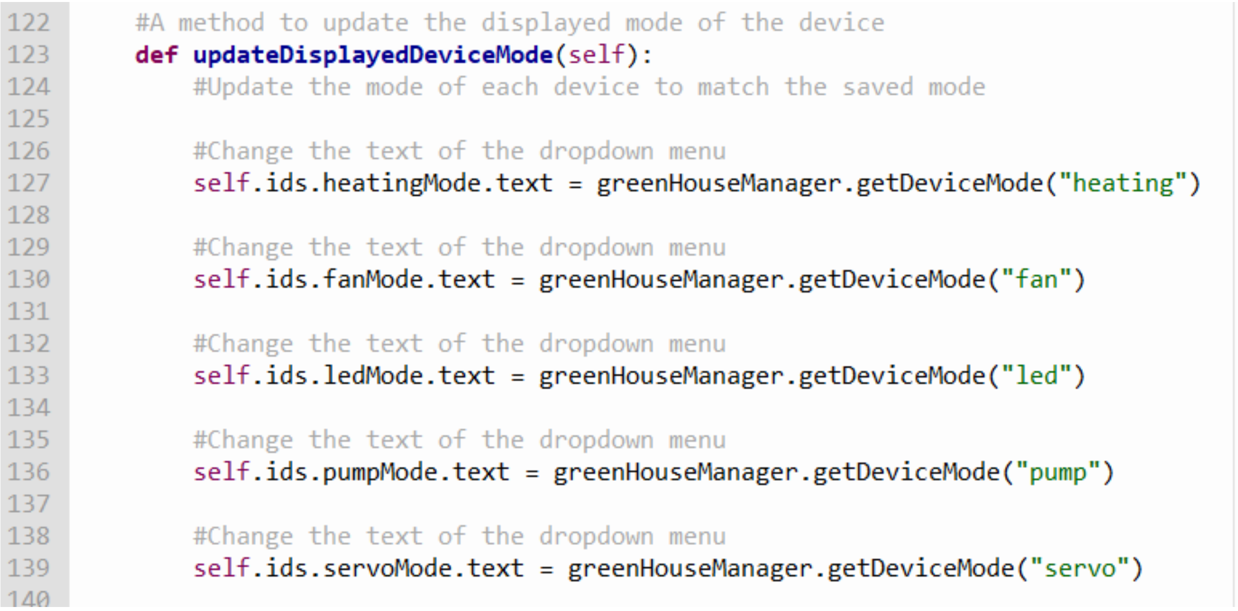


```
    #Method to update the dispalyed device status shown on screen
    def updateDisplayedDeviceStatus(self):
        #Update the status of each device to match that of the saved devices file

        #Change the text of the status button
        self.ids.heatingStatus.text = greenHouseManager.getDeviceStatus("heating")

        #Change the color of the status button
        if greenHouseManager.getDeviceStatus("heating") == "Off":
            #When off the color is red
            self.ids.heatingStatus.background_color = (1,0,0,1)
        else:
            #When on the color is green
            self.ids.heatingStatus.background_color = (0, 0.69, 0.31, 1)

        #Change the text of the status button
        self.ids.fanStatus.text = greenHouseManager.getDeviceStatus("fan")

        #Change the color of the status button
        if greenHouseManager.getDeviceStatus("fan") == "Off":
            #When off the color is red
            self.ids.fanStatus.background_color = (1,0,0,1)
        else:
            #When on the color is green
            self.ids.fanStatus.background_color = (0, 0.69, 0.31, 1)

        #Change the text of the status button
        self.ids.ledStatus.text = greenHouseManager.getDeviceStatus("led")
```



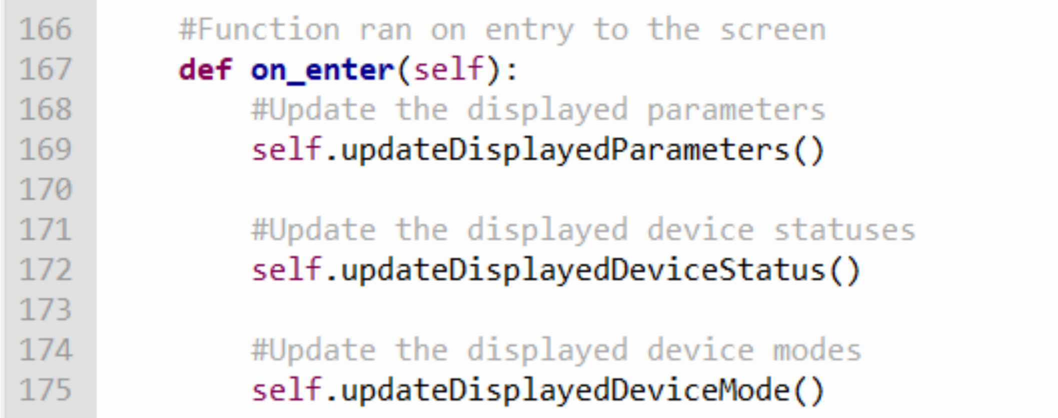
```
        #Change the color of the status button
        if greenHouseManager.getDeviceStatus("led") == "Off":
            #When off the color is red
            self.ids.ledStatus.background_color = (1,0,0,1)
        else:
            #When on the color is green
            self.ids.ledStatus.background_color = (0, 0.69, 0.31, 1)

        #Change the text of the status button
        self.ids.pumpStatus.text = greenHouseManager.getDeviceStatus("pump")

        #Change the color of the status button
        if greenHouseManager.getDeviceStatus("pump") == "Off":
            #When off the color is red
            self.ids.pumpStatus.background_color = (1,0,0,1)
        else:
            #When on the color is green
            self.ids.pumpStatus.background_color = (0, 0.69, 0.31, 1)

        #Change the text of the status button
        self.ids.servoStatus.text = greenHouseManager.getDeviceStatus("servo")

        #Change the color of the status button
        if greenHouseManager.getDeviceStatus("servo") == "Off":
            #When off the color is red
            self.ids.servoStatus.background_color = (1,0,0,1)
        else:
            #When on the color is green
            self.ids.servoStatus.background_color = (0, 0.69, 0.31, 1)
```

The update displayed device status method inside the parameters screen class will be used to set the value out on off status toggles for each device. This method will be called on entry to the parameters screen to ensure the values shown on screen for device status match that of the saved values. For each device the text of the toggle is set to the value of its status. Which will be either on or off to find its value I'm using the getter method get device status with the parameter corresponding to the right device. Then a selection if statement looks to see if the device status is off if this is the case then the toggle buttons background color is swapped to be red. In the alternative case that the text is on then the background color is made green. As by default the background color of all these toggle buttons are green as I defined inside the kv file I could probably do away with the else part of the if statements. However, I've decided to keep it for robustness it could be useful if I ever need to refresh all the toggles to make sure their values are correct.



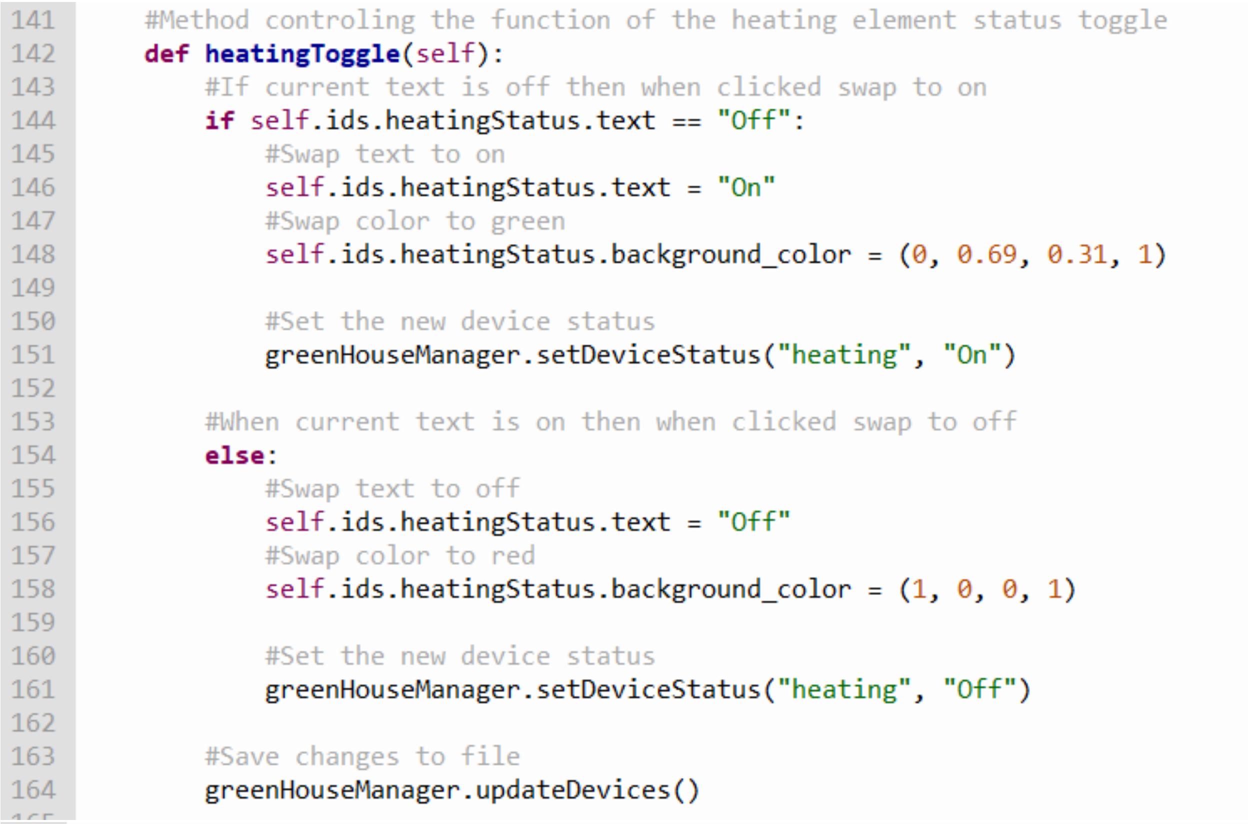
```
    #A method to update the displayed mode of the device
    def updateDisplayedDeviceMode(self):
        #Update the mode of each device to match the saved mode

        #Change the text of the dropdown menu
        self.ids.heatingMode.text = greenHouseManager.getDeviceMode("heating")

        #Change the text of the dropdown menu
        self.ids.fanMode.text = greenHouseManager.getDeviceMode("fan")

        #Change the text of the dropdown menu
        self.ids.ledMode.text = greenHouseManager.getDeviceMode("led")

        #Change the text of the dropdown menu
        self.ids.pumpMode.text = greenHouseManager.getDeviceMode("pump")

        #Change the text of the dropdown menu
        self.ids.servoMode.text = greenHouseManager.getDeviceMode("servo")
```

The dropdown menus to select the device mode also need to be updated to match the saved value for that device upon entry to the parameters screen. To do this I have set the text value of each dropdown menu to equal the current mode of the related device. For the last two methods I have added ids to the elements in question to allow me to access their properties from inside python.

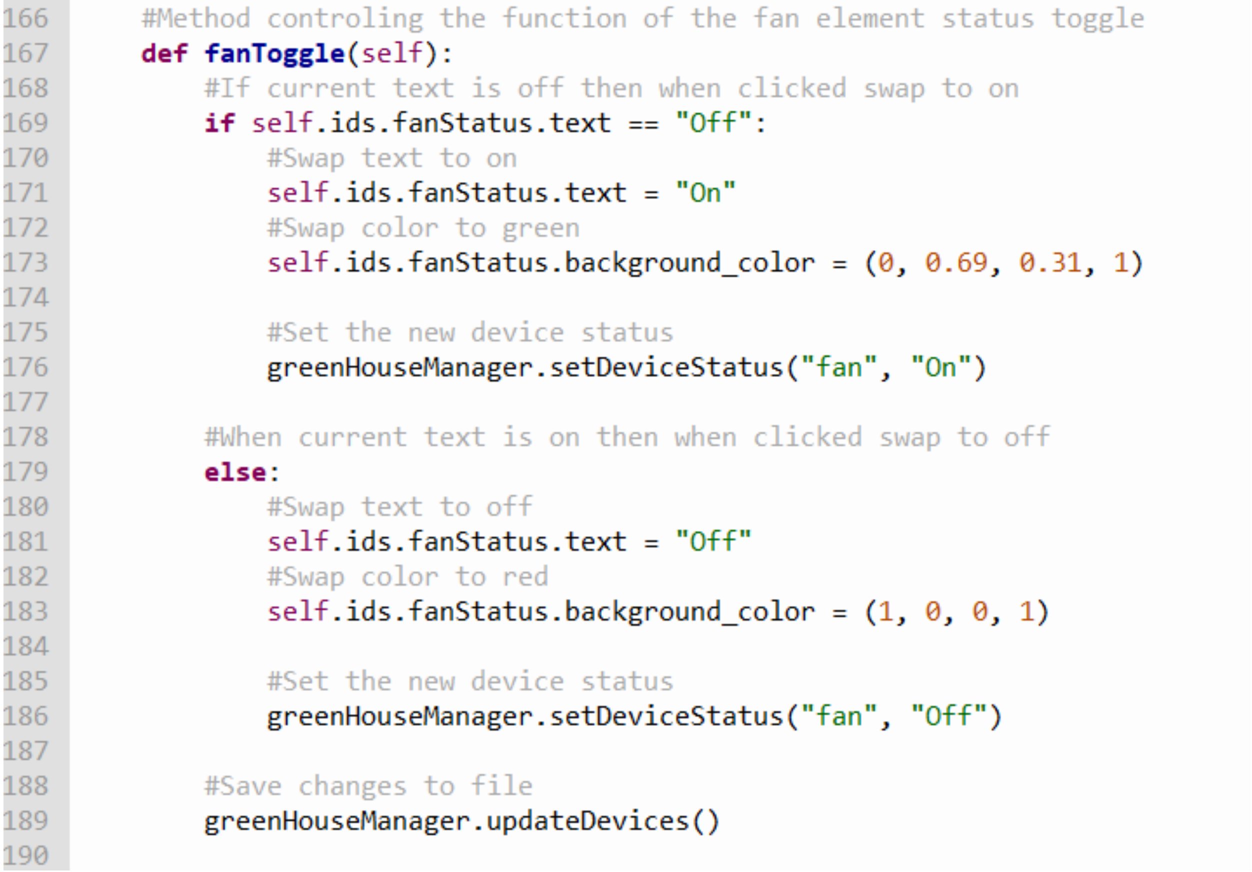


```
    #Function ran on entry to the screen
    def on_enter(self):
        #Update the displayed parameters
        self.updateDisplayedParameters()

        #Update the displayed device statuses
        self.updateDisplayedDeviceStatus()

        #Update the displayed device modes
        self.updateDisplayedDeviceMode()
```

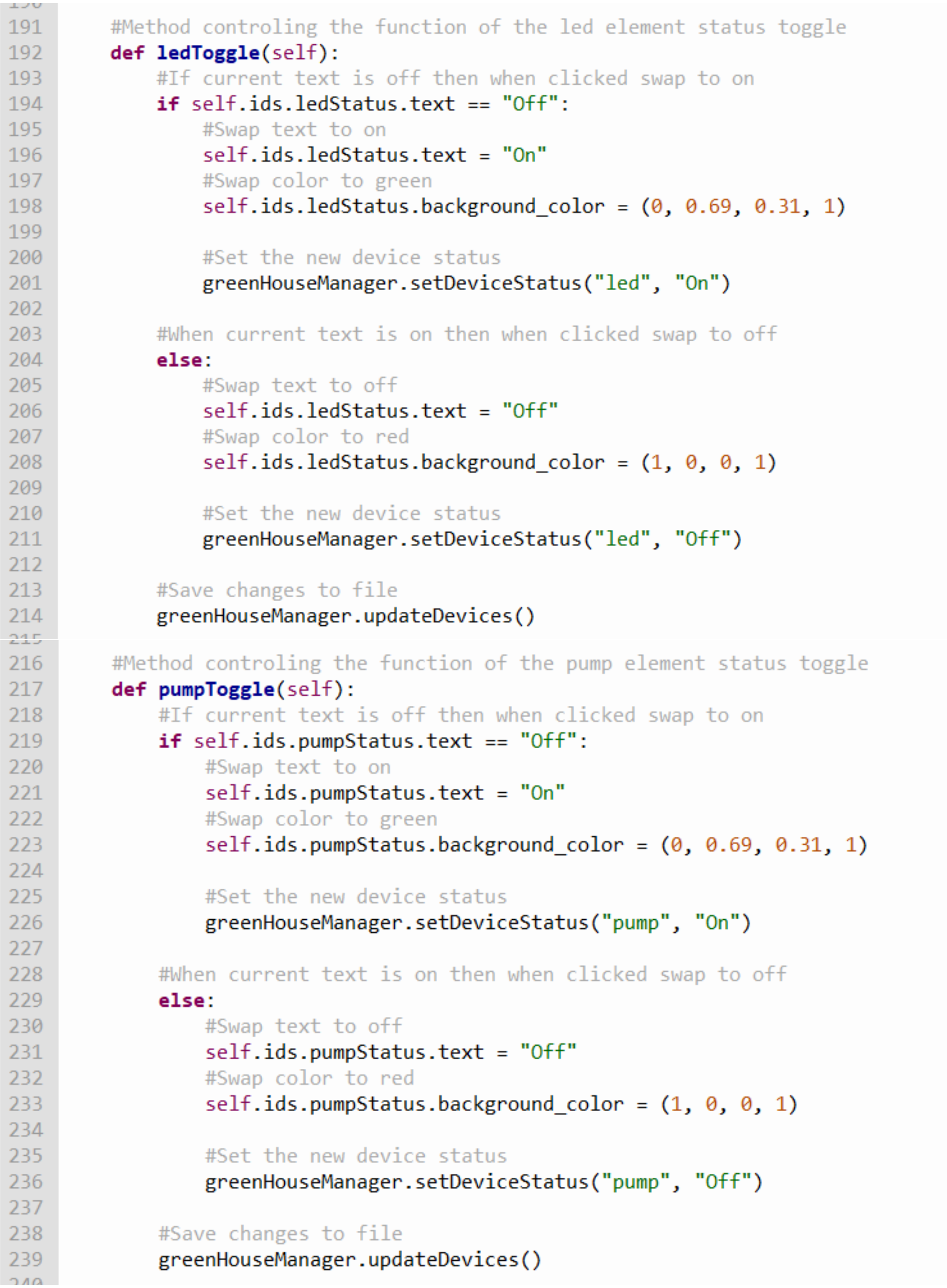
Both these two new methods are run on entry to the parameters screen to make sure the values for device status and also mode are matching to the current saved value.



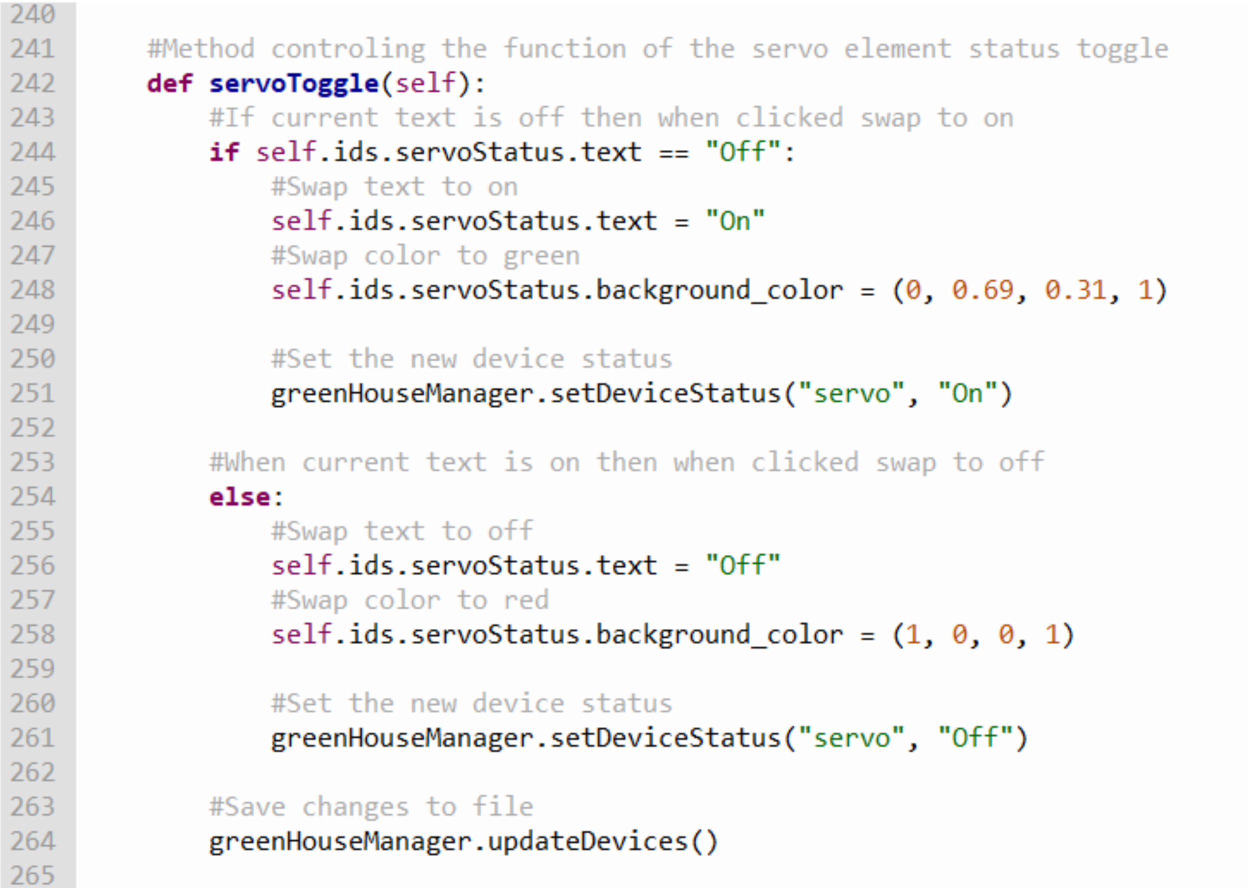
```
    #Method controling the function of the heating element status toggle
    def heatingToggle(self):
        #If current text is off then when clicked swap to on
        if self.ids.heatingStatus.text == "Off":
            #Swap text to on
            self.ids.heatingStatus.text = "On"
            #Swap color to green
            self.ids.heatingStatus.background_color = (0, 0.69, 0.31, 1)

            #Set the new device status
            greenHouseManager.setDeviceStatus("heating", "On")

        #When current text is on then when clicked swap to off
        else:
            #Swap text to off
            self.ids.heatingStatus.text = "Off"
            #Swap color to red
            self.ids.heatingStatus.background_color = (1, 0, 0, 1)

            #Set the new device status
            greenHouseManager.setDeviceStatus("heating", "Off")

        #Save changes to file
        greenHouseManager.updateDevices()
```



```
    #Method controling the function of the fan element status toggle
    def fanToggle(self):
        #If current text is off then when clicked swap to on
        if self.ids.fanStatus.text == "Off":
            #Swap text to on
            self.ids.fanStatus.text = "On"
            #Swap color to green
            self.ids.fanStatus.background_color = (0, 0.69, 0.31, 1)

            #Set the new device status
            greenHouseManager.setDeviceStatus("fan", "On")

        #When current text is on then when clicked swap to off
        else:
            #Swap text to off
            self.ids.fanStatus.text = "Off"
            #Swap color to red
            self.ids.fanStatus.background_color = (1, 0, 0, 1)

            #Set the new device status
            greenHouseManager.setDeviceStatus("fan", "Off")

        #Save changes to file
        greenHouseManager.updateDevices()

```

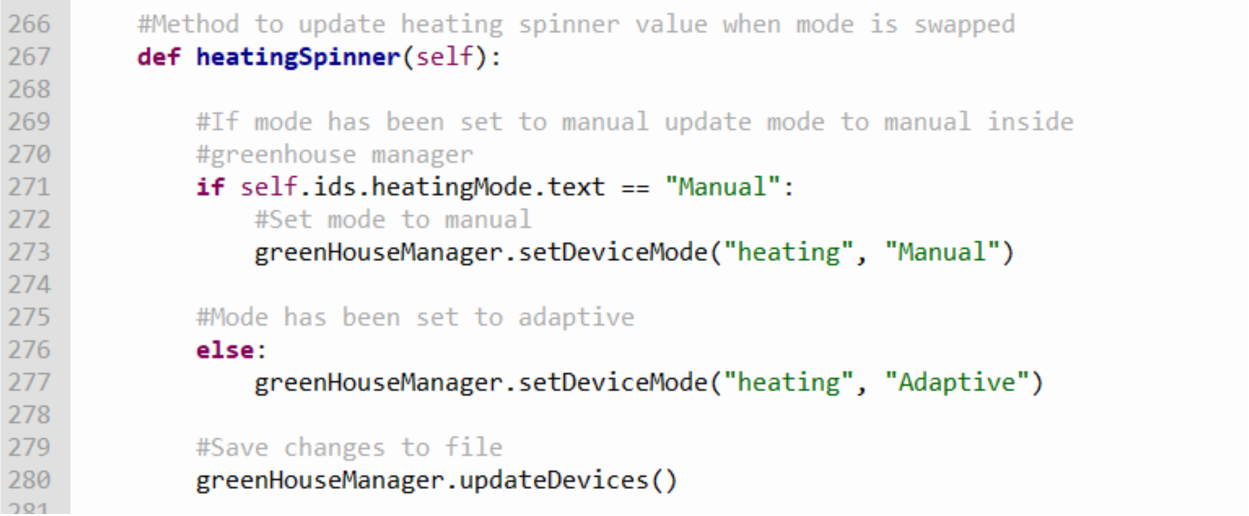


```
    #Method controling the function of the led element status toggle
    def ledToggle(self):
        #If current text is off then when clicked swap to on
        if self.ids.ledStatus.text == "Off":
            #Swap text to on
            self.ids.ledStatus.text = "On"
            #Swap color to green
            self.ids.ledStatus.background_color = (0, 0.69, 0.31, 1)

            #Set the new device status
            greenHouseManager.setDeviceStatus("led", "On")

        #When current text is on then when clicked swap to off
        else:
            #Swap text to off
            self.ids.ledStatus.text = "Off"
            #Swap color to red
            self.ids.ledStatus.background_color = (1, 0, 0, 1)

            #Set the new device status
            greenHouseManager.setDeviceStatus("led", "Off")

        #Save changes to file
        greenHouseManager.updateDevices()

    #Method controling the function of the pump element status toggle
    def pumpToggle(self):
        #If current text is off then when clicked swap to on
        if self.ids.pumpStatus.text == "Off":
            #Swap text to on
            self.ids.pumpStatus.text = "On"
            #Swap color to green
            self.ids.pumpStatus.background_color = (0, 0.69, 0.31, 1)

            #Set the new device status
            greenHouseManager.setDeviceStatus("pump", "On")

        #When current text is on then when clicked swap to off
        else:
            #Swap text to off
            self.ids.pumpStatus.text = "Off"
            #Swap color to red
            self.ids.pumpStatus.background_color = (1, 0, 0, 1)

            #Set the new device status
            greenHouseManager.setDeviceStatus("pump", "Off")

        #Save changes to file
        greenHouseManager.updateDevices()
```

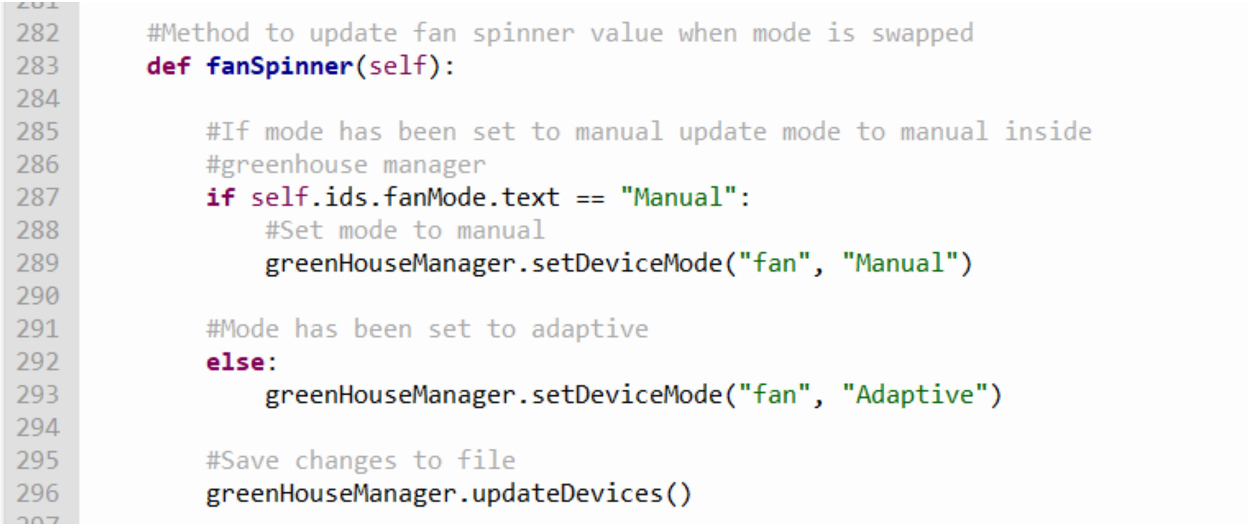


```

    #Method controling the function of the servo element status toggle
    def servoToggle(self):
        #If current text is off then when clicked swap to on
        if self.ids.servoStatus.text == "Off":
            #Swap text to on
            self.ids.servoStatus.text = "On"
            #Swap color to green
            self.ids.servoStatus.background_color = (0, 0.69, 0.31, 1)

            #Set the new device status
            greenHouseManager.setDeviceStatus("servo", "On")

        #When current text is on then when clicked swap to off
        else:
            #Swap text to off
            self.ids.servoStatus.text = "Off"
            #Swap color to red
            self.ids.servoStatus.background_color = (1, 0, 0, 1)

            #Set the new device status
            greenHouseManager.setDeviceStatus("servo", "Off")

        #Save changes to file
        greenHouseManager.updateDevices()

```

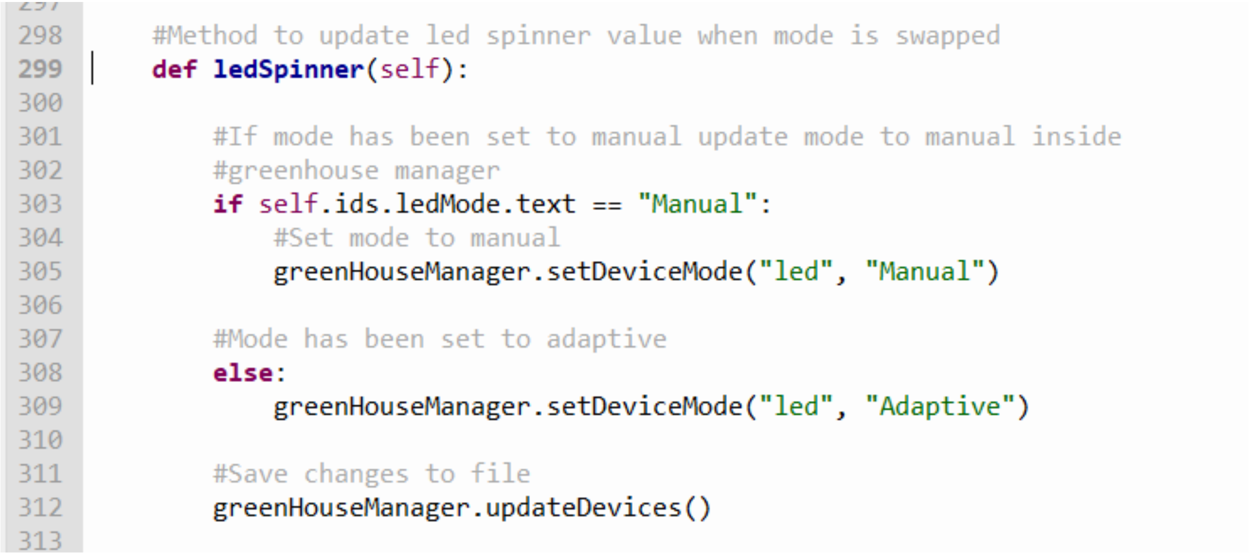
The code for each of the five toggles on off buttons is the same just with a different function name and also the right id for that toggle used. When the toggle is clicked the user wants to change the button from either on to off or from off to on. When the button is pressed the toggle method for that button is called. The function checks the current text of the toggle. If the text is currently “off” then the toggle needs to be set into the “on” position. So, the text for the toggle is changed to “on” and the background color is set to be green. The status of that device is also changed using the setter method set device status. If the text is currently “On” then the reverse happens. The text is set to equal “off” and the color becomes red. After this the update devices method of the green house manager is called so that the changes made are saved into memory. When I was implementing the toggles, I considered having one function with a device argument which took the device which was being turned on or off. However, I was not able to find a way to concatenate kivy ids to include the id of the device passed as a parameter. So, I was forced to make separate functions for each of the toggles.



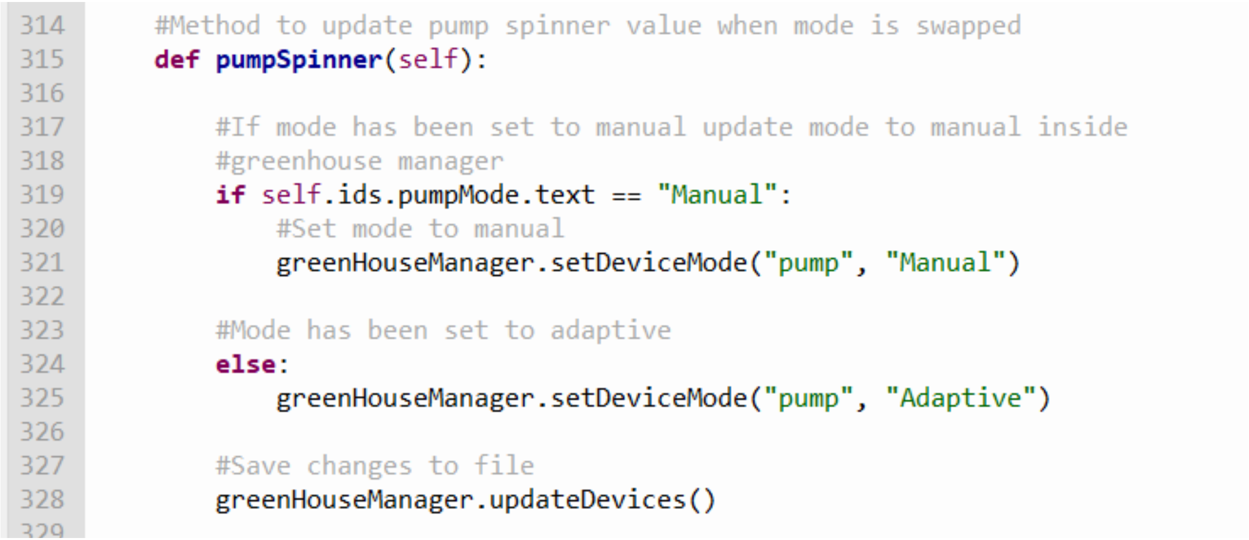
```
    #Method to update heating spinner value when mode is swapped
    def heatingSpinner(self):

        #If mode has been set to manual update mode to manual inside
        #greenhouse manager
        if self.ids.heatingMode.text == "Manual":
            #Set mode to manual
            greenHouseManager.setDeviceMode("heating", "Manual")

        #Mode has been set to adaptive
        else:
            greenHouseManager.setDeviceMode("heating", "Adaptive")

        #Save changes to file
        greenHouseManager.updateDevices()
```



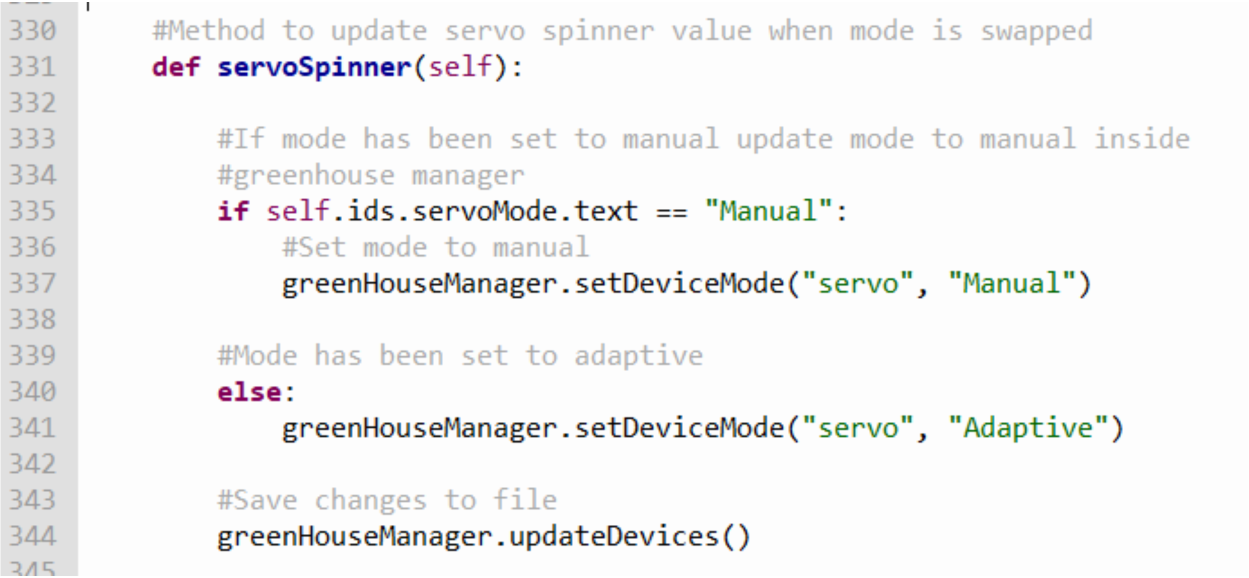
```
    #Method to update fan spinner value when mode is swapped
    def fanSpinner(self):

        #If mode has been set to manual update mode to manual inside
        #greenhouse manager
        if self.ids.fanMode.text == "Manual":
            #Set mode to manual
            greenHouseManager.setDeviceMode("fan", "Manual")

        #Mode has been set to adaptive
        else:
            greenHouseManager.setDeviceMode("fan", "Adaptive")

        #Save changes to file
        greenHouseManager.updateDevices()
```



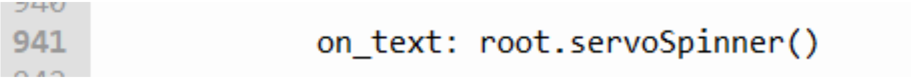
```
    #Method to update led spinner value when mode is swapped
    def ledSpinner(self):

        #If mode has been set to manual update mode to manual inside
        #greenhouse manager
        if self.ids.ledMode.text == "Manual":
            #Set mode to manual
            greenHouseManager.setDeviceMode("led", "Manual")

        #Mode has been set to adaptive
        else:
            greenHouseManager.setDeviceMode("led", "Adaptive")

        #Save changes to file
        greenHouseManager.updateDevices()
```



```
    #Method to update pump spinner value when mode is swapped
    def pumpSpinner(self):

        #If mode has been set to manual update mode to manual inside
        #greenhouse manager
        if self.ids.pumpMode.text == "Manual":
            #Set mode to manual
            greenHouseManager.setDeviceMode("pump", "Manual")

        #Mode has been set to adaptive
        else:
            greenHouseManager.setDeviceMode("pump", "Adaptive")

        #Save changes to file
        greenHouseManager.updateDevices()
```



```
    #Method to update servo spinner value when mode is swapped
    def servoSpinner(self):

        #If mode has been set to manual update mode to manual inside
        #greenhouse manager
        if self.ids.servoMode.text == "Manual":
            #Set mode to manual
            greenHouseManager.setDeviceMode("servo", "Manual")

        #Mode has been set to adaptive
        else:
            greenHouseManager.setDeviceMode("servo", "Adaptive")

        #Save changes to file
        greenHouseManager.updateDevices()
```



```
            on_text: root.servoSpinner()
```

When the user selects a new mode for a device using the dropdown menu or as kivy call it spinner that change needs to be recorded in the greenhouse manager class and saved to the devices text file. Unlike buttons in kivy spinners don’t have a “on_press” attribute instead for spinners you need to use the “on_text” attribute which is called when the user selects a new value and hence changes the text of the dropdown menu. For each dropdown I have created a device spinner method and binded this to the “on_text” property of the delated spinner. When the spinners’ function is called there is a check to see if the text is manual if this is the case then the greenhouse manager mode for that device is updated to be manual and otherwise it is set to adaptive. Finally, the greenhouse manger update devices method is called to update the devices.txt file. Rather annoyingly the “on_text” parameter is called whenever the text of a drop-down menu is changed not only by the user in the gui but also when the text is changed via id reference inside python. This means that when the GUI loads, and the update displayed device mode method is ran on entry to the screen for each of the spinners a change of text is occurring meaning this then sets off the spinners “on_text” parameter. So, in effect the screen is now being loaded then getting the mode values from the devices text file and assigning them to the spinners for each device which then sets off "on_text" meaning the value is then written back to the file. Unfortunately, there is no way to get around this.

Testing plan

Test Number	Test Plan	Expected Outcome	Actual Outcome	Pass/Fail
1	Set the heating element status to on	Status will be on in the gui and in the devices file		Pass
2	Set the heating element status to off	Status will be off in the gui and the devices file		Pass
3	Set the fan status to on	Status will be on in the gui and in devices file		Pass
4	Set the fan status to off	Status will be off in the gui and in the devices file		Pass
5	Set the leds status to on	Status will be off in the gui and in the devices file		Pass
6	Set the leds status to off	Status will be off in the gui and in the devices file		Pass
7	Set the pump status to on	Status will be on in the gui and in the devices file		Pass
8	Set the pump status to off	Status will be off in the gui and in the devices file		Pass
9	Set the heating mode to manual	Mode will be manual in gui and in the devices file		Pass
10	Set the heating mode to adaptive	Mode will be adaptive in gui and in the devices file		Pass
11	Set the fan mode to adaptive	Mode will be adaptive in the gui and in the devices file		Pass
12	Set the fan mode to manual	Mode will be adaptive in the gui and in the devices file		Pass
13	Set the LEDs mode to manual	Mode will be manual in the gui		Pass

		and in the devices file		
14	Set the LEDs mode to adaptive	Mode will be adaptive in the gui and in the devices file		Pass
15	Set the pump mode to manual	Mode will be manual in the gui and in the devices file		Pass
16	Set the pump mode to adaptive	Mode will be adaptive in the gui and in the devices file		Pass
17	Set the servo mode to manual	Mode will be manual in the gui and in the devices file		Pass
18	Set the servo mode to adaptive	Mode will be adaptive in the gui and in the devices file		Pass

Review

The parameters page is now fully implemented with the ability to view current device modes and status along with seeing the current greenhouse parameters along with ability to change all these values and that be reflected inside their text files.

Iterative stage 11 – Greenhouse settings

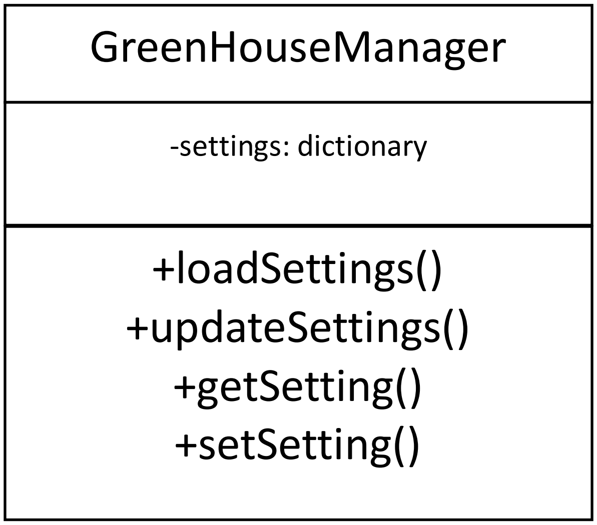
Overview

There are a few final settings which need to be stored before I can begin to implement the greenhouse environment management functions which will continually monitor and adapt the greenhouse environment. In this iterative stage I will be implementing the overall greenhouse settings file which will store certain values regarding the greenhouse such as the status of the greenhouse, the mode of the greenhouse and schedule of the greenhouse.

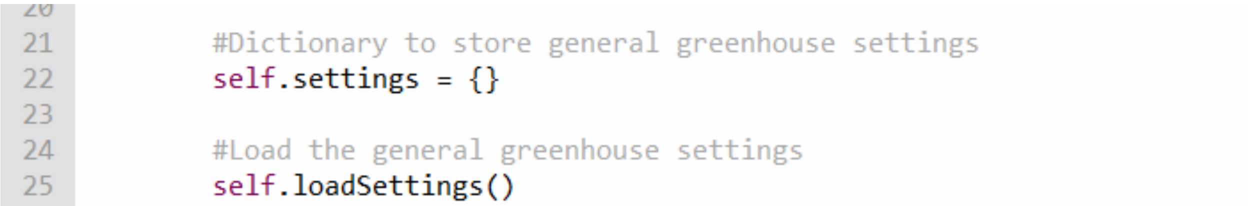
Requirements

During this stage I will be setting up the general greenhouse settings. This will require a settings text file too store the 4 general settings. Which is greenhouse status this will decide if the greenhouse will be on or off, mode which will determine if the greenhouse runs continually or only during a set time period and start / end time which will determine when the greenhouse will run if it is in scheduled mode.

Class Diagram

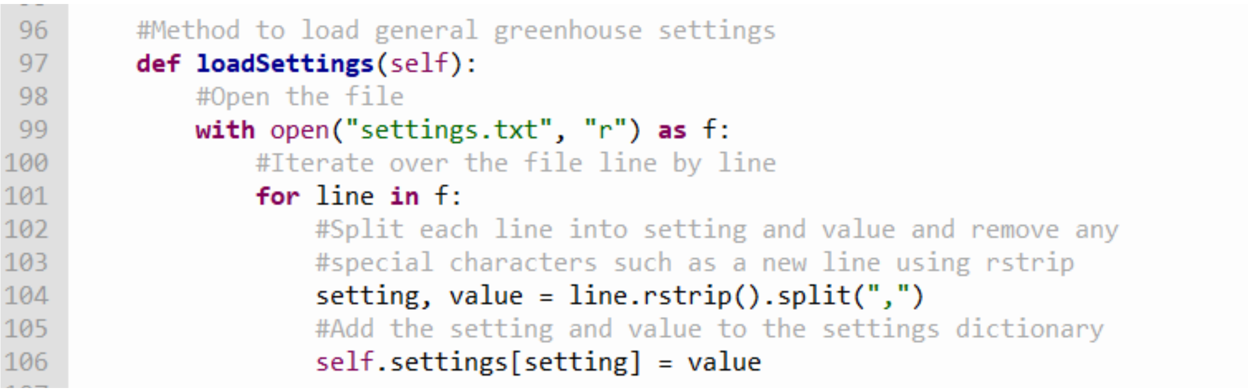


Development log



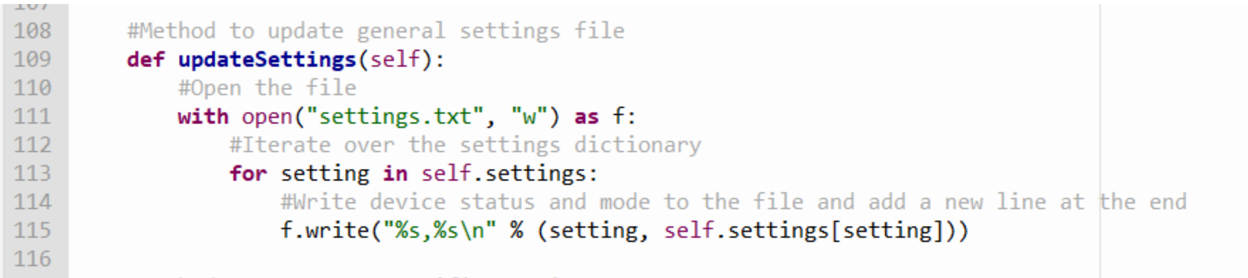
```
        #Dictionary to store general greenhouse settings
        self.settings = {}

        #Load the general greenhouse settings
        self.loadSettings()
```

The settings methods of the greenhouse manager class are essentially the same as the parameters and devices methods. Inside the constructor I have created a settings dictionary to store the general settings of the greenhouse. I have also called the load settings method to fill the settings dictionary.



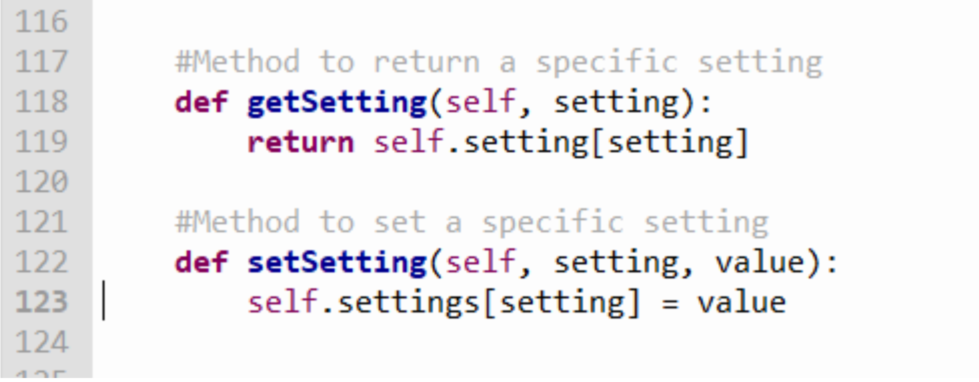
```
    #Method to load general greenhouse settings
    def loadSettings(self):
        #Open the file
        with open("settings.txt", "r") as f:
            #Iterate over the file line by line
            for line in f:
                #Split each line into setting and value and remove any
                #special characters such as a new line using rstrip
                setting, value = line.rstrip().split(",")
                #Add the setting and value to the settings dictionary
                self.settings[setting] = value
```

The load settings method opens the settings file and writes the settings values to the settings dictionary.



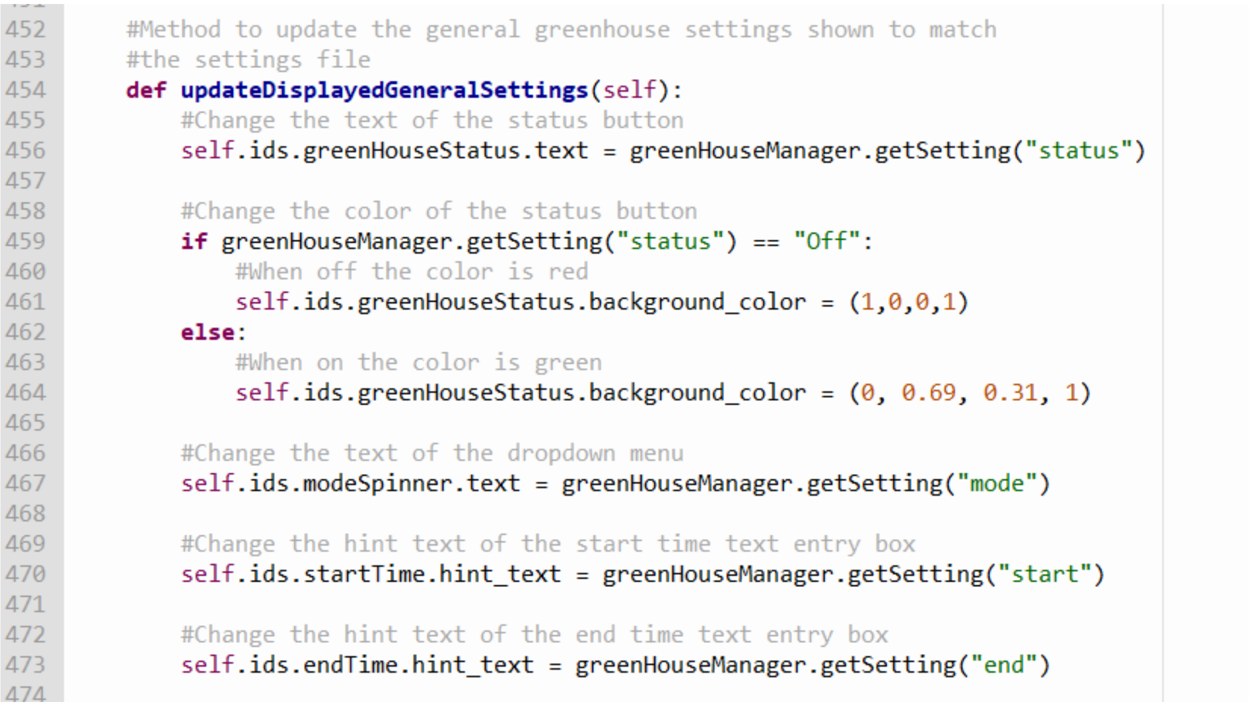
```
    #Method to update general settings file
    def updateSettings(self):
        #Open the file
        with open("settings.txt", "w") as f:
            #Iterate over the settings dictionary
            for setting in self.settings:
                #Write device status and mode to the file and add a new line at the end
                f.write("%s,%s\n" % (setting, self.settings[setting]))

```

Whilst the update settings method writes the contents of the settings dictionary to the settings text file.



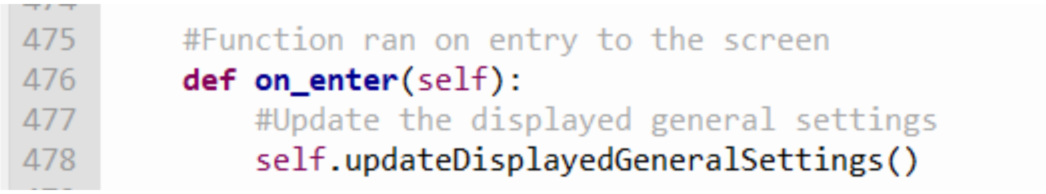
```python

    #Method to return a specific setting
    def getSetting(self, setting):
        return self.setting[setting]

    #Method to set a specific setting
    def setSetting(self, setting, value):
        self.settings[setting] = value

```

A getter and a setter method are used to allow for the setting and getting of general setting values.



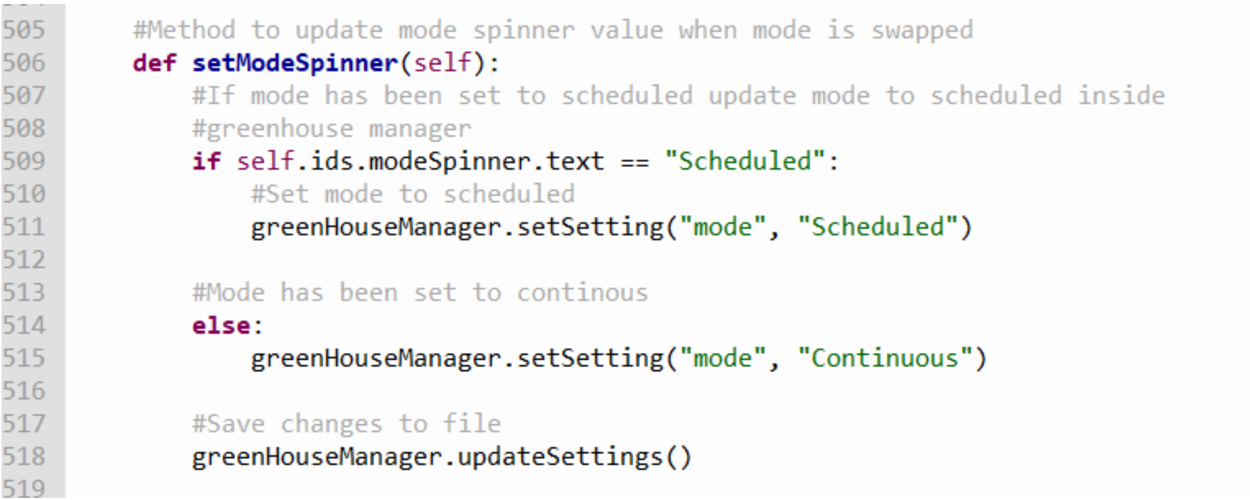
```python
    #Method to update the general greenhouse settings shown to match
    #the settings file
    def updateDisplayedGeneralSettings(self):
        #Change the text of the status button
        self.ids.greenHouseStatus.text = greenHouseManager.getSetting("status")

        #Change the color of the status button
        if greenHouseManager.getSetting("status") == "Off":
            #When off the color is red
            self.ids.greenHouseStatus.background_color = (1,0,0,1)
        else:
            #When on the color is green
            self.ids.greenHouseStatus.background_color = (0, 0.69, 0.31, 1)

        #Change the text of the dropdown menu
        self.ids.modeSpinner.text = greenHouseManager.getSetting("mode")

        #Change the hint text of the start time text entry box
        self.ids.startTime.hint_text = greenHouseManager.getSetting("start")

        #Change the hint text of the end time text entry box
        self.ids.endTime.hint_text = greenHouseManager.getSetting("end")

```

Inside the settings screen class I have made an update displayed general settings method which is responsible for setting the status, mode, and time values of the 3 greenhouse general settings on the settings page. This method uses the get setting method to get the required value and then assigns that to the label inside kivy. The status toggle button is also set to the right color.

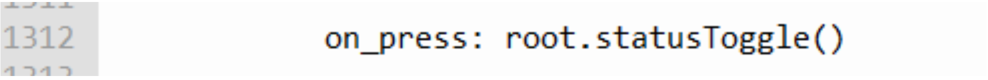


```python
    #Function ran on entry to the screen
    def on_enter(self):
        #Update the displayed general settings
        self.updateDisplayedGeneralSettings()
```

The on enter function is used to make sure that each time the user enters the settings screen the general settings are updated. This ensures they are always up to date.

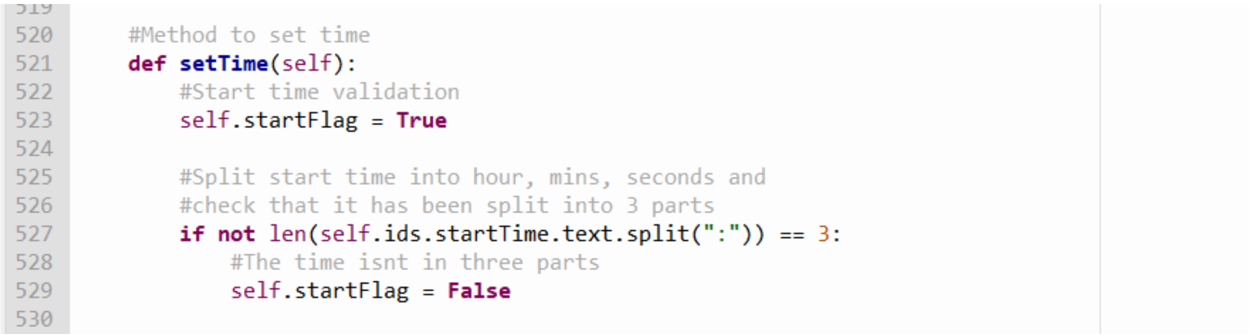


```
    #Method to update mode spinner value when mode is swapped
    def setModeSpinner(self):
        #If mode has been set to scheduled update mode to scheduled inside
        #greenhouse manager
        if self.ids.modeSpinner.text == "Scheduled":
            #Set mode to scheduled
            greenHouseManager.setSetting("mode", "Scheduled")

        #Mode has been set to continous
        else:
            greenHouseManager.setSetting("mode", "Continuous")

        #Save changes to file
        greenHouseManager.updateSettings()
```



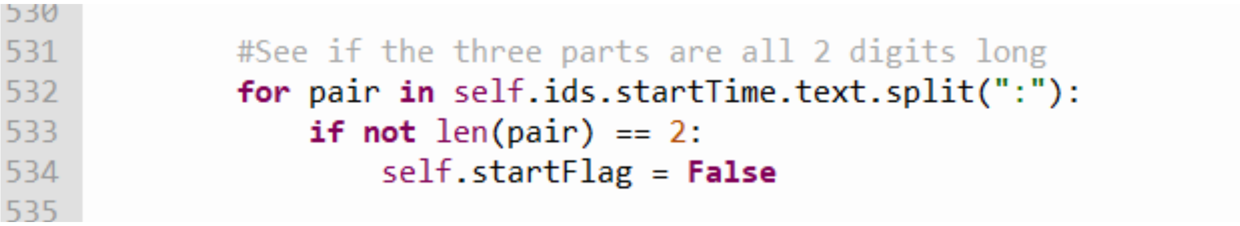
```
                on_press: root.statusToggle()
```

When the user decides to change the mode of the greenhouse, they do this via the dropdown menu. The set mode spinner method is responsible for updating the stored value of the greenhouse mode to reflect the user selected mode. The method simply checks the text value of the dropdown menu and then sets the setting equal to that value. At the end of the method the update settings method is called so that these changes are recorded into the text file. I have binded this function to the on-text property of the dropdown menu so it’s ran when a new option is changed.



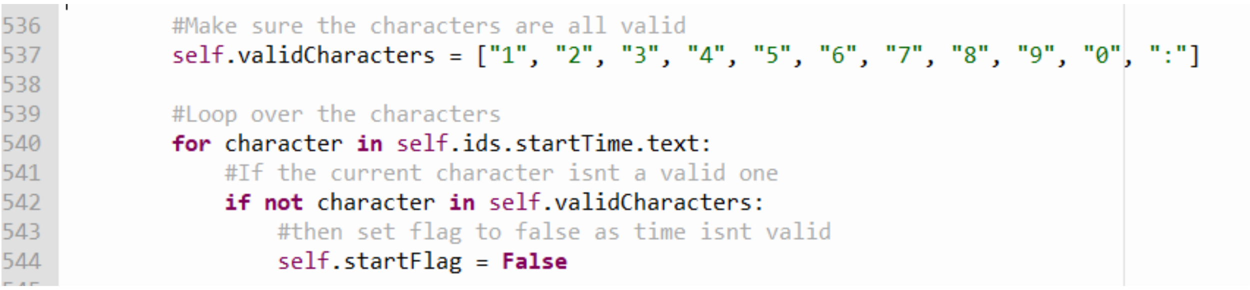
```
    #Method to set time
    def setTime(self):
        #Start time validation
        self.startFlag = True

        #Split start time into hour, mins, seconds and
        #check that it has been split into 3 parts
        if not len(self.ids.startTime.text.split(":")) == 3:
            #The time isnt in three parts
            self.startFlag = False
```

The user is given the ability to set the start and end time of the greenhouse operations. These times are set via two text input boxes and then saved when the user selects the set button. The set time method will be governing the validation and saving of this data and will be called when the set button is clicked. The process for validating the start time and end time is the same the process is repeated twice inside this method just using the other text input the second time. To begin with a flag is set to be true. Providing this flag is still true once validation is complete then the data entered by the user is okay. The first validation step is to try and split the input at the “:” character. If the user has entered the correct time format, then this will result in 3 list items inside an array. The length of the produced array is compared to 3 and if not equal then the flag becomes false as the format can’t be “hh:mm:ss” as required. The split function in python does not produce an error if there are no “:” characters present it just returns the original text so its safe to do this test if the user enters no “:” characters.



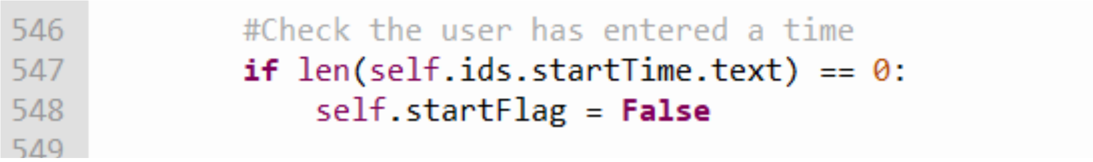
```
        #See if the three parts are all 2 digits long
        for pair in self.ids.startTime.text.split(":"):
            if not len(pair) == 2:
                self.startFlag = False
```

The next test is to see if the 3 constituent parts are all 2 digits long. The for loop iterates over the time which has been split into the following “hh”, “mm” and “ss”. A check is made to see if the length is not equal to 2 then the flag becomes false.



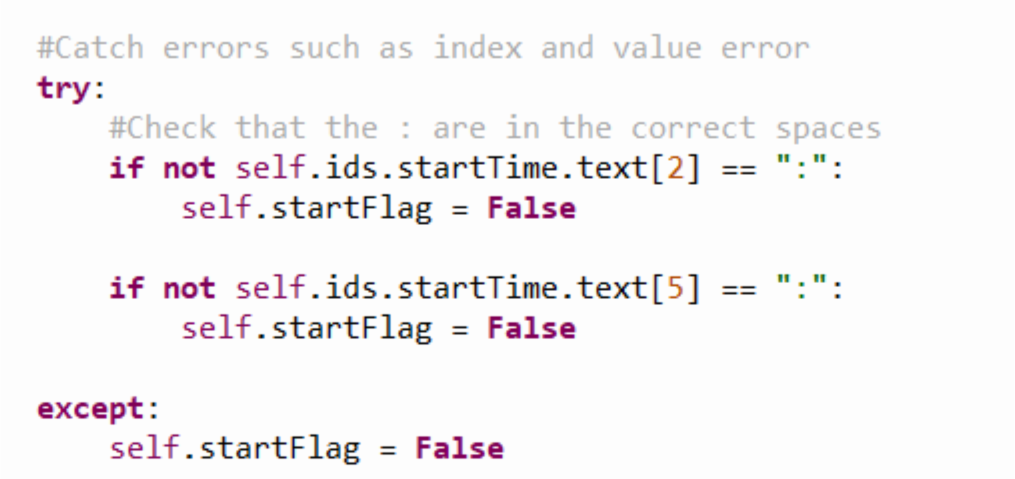
```
        #Make sure the characters are all valid
        self.validCharacters = ["1", "2", "3", "4", "5", "6", "7", "8", "9", "0", ":"]

        #Loop over the characters
        for character in self.ids.startTime.text:
            #If the current character isnt a valid one
            if not character in self.validCharacters:
                #then set flag to false as time isnt valid
                self.startFlag = False
```

There are only 11 valid characters which the user can enter the time. These are defined in the valid characters array. An iteration goes through the time text and checks if the letter is not a valid one then the flag is false to fail the validation.



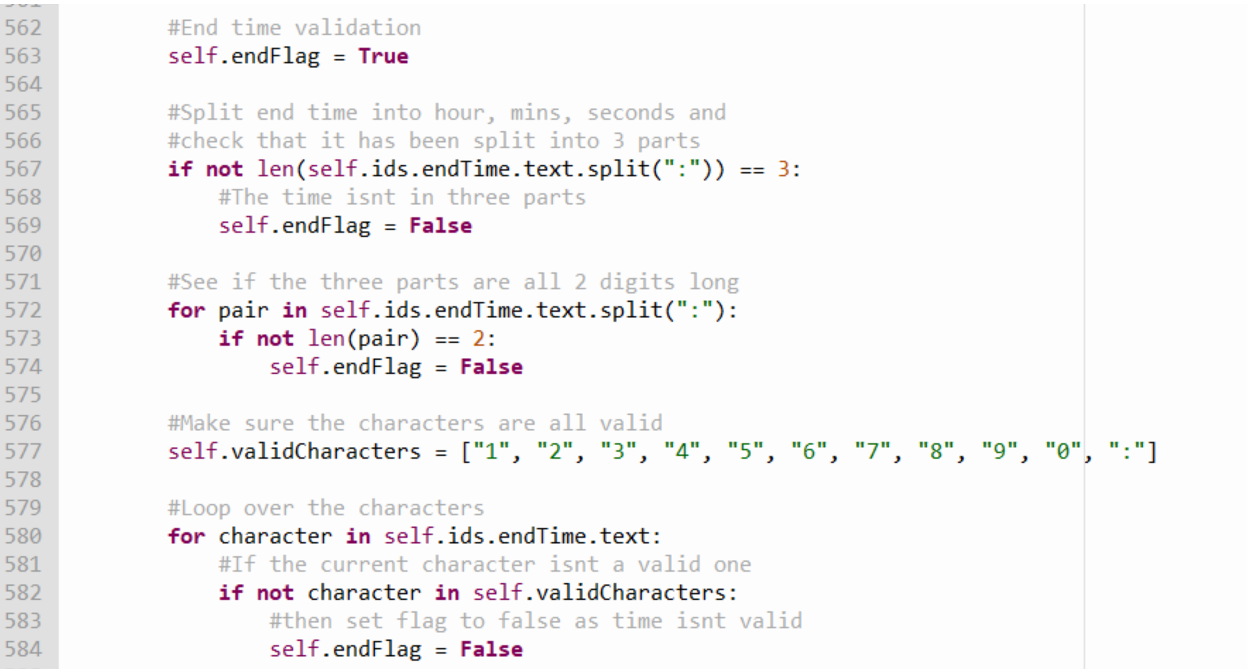
```
        #Check the user has entered a time
        if len(self.ids.startTime.text) == 0:
            self.startFlag = False
```

This check looks to see if the user has entered any characters if that’s the case then the flag is false.

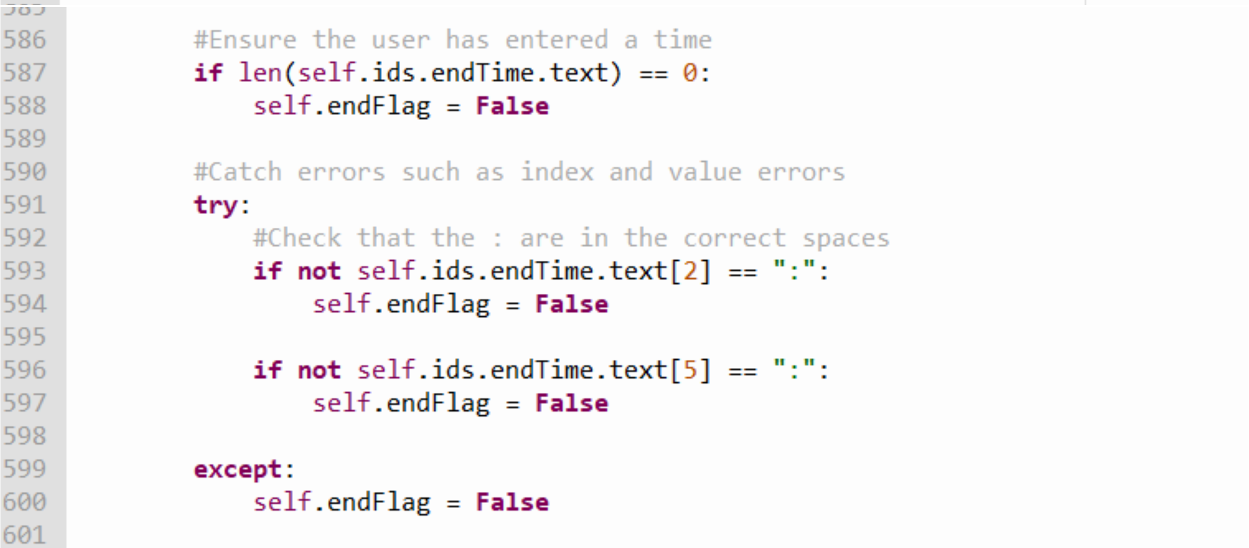


```
#Catch errors such as index and value error
try:
    #Check that the : are in the correct spaces
    if not self.ids.startTime.text[2] == ":":
        self.startFlag = False

    if not self.ids.startTime.text[5] == ":":
        self.startFlag = False

except:
    self.startFlag = False
```

The final validation check is to see that the second and fifth characters are “:”. This check creates a situation where if the user has not entered enough characters, then we will be trying to access an index out of range. To account for this case, I am using a try except statement. In the event that the index Is out of range an index error will occur which will be handled by the except statement which will set the flag as false.

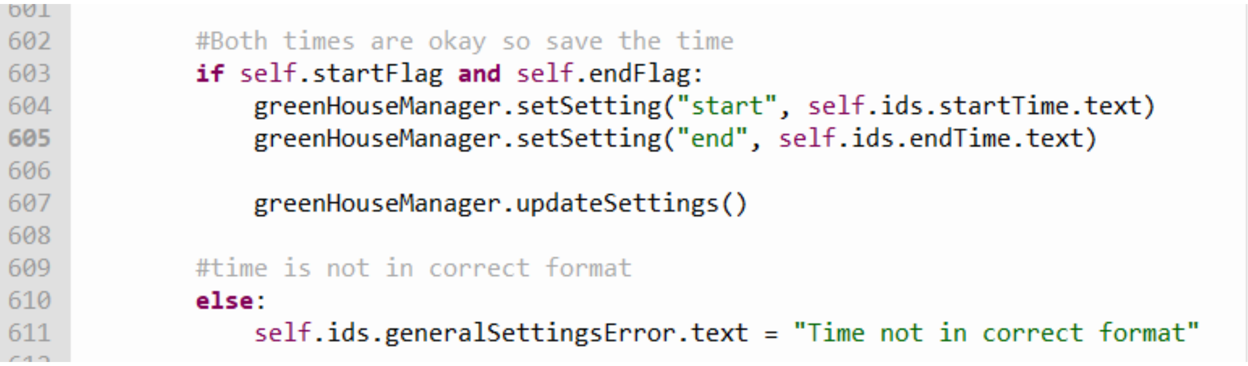


```
        #End time validation
        self.endFlag = True

        #Split end time into hour, mins, seconds and
        #check that it has been split into 3 parts
        if not len(self.ids.endTime.text.split(":")) == 3:
            #The time isnt in three parts
            self.endFlag = False

        #See if the three parts are all 2 digits long
        for pair in self.ids.endTime.text.split(":"):
            if not len(pair) == 2:
                self.endFlag = False

        #Make sure the characters are all valid
        self.validCharacters = ["1", "2", "3", "4", "5", "6", "7", "8", "9", "0", ":"]

        #Loop over the characters
        for character in self.ids.endTime.text:
            #If the current character isnt a valid one
            if not character in self.validCharacters:
                #then set flag to false as time isnt valid
                self.endFlag = False
```



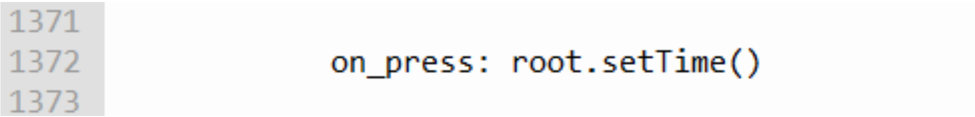
```
        #Ensure the user has entered a time
        if len(self.ids.endTime.text) == 0:
            self.endFlag = False

        #Catch errors such as index and value errors
        try:
            #Check that the : are in the correct spaces
            if not self.ids.endTime.text[2] == ":":
                self.endFlag = False

            if not self.ids.endTime.text[5] == ":":
                self.endFlag = False

        except:
            self.endFlag = False
```

The same validation process is carried out on the second time.



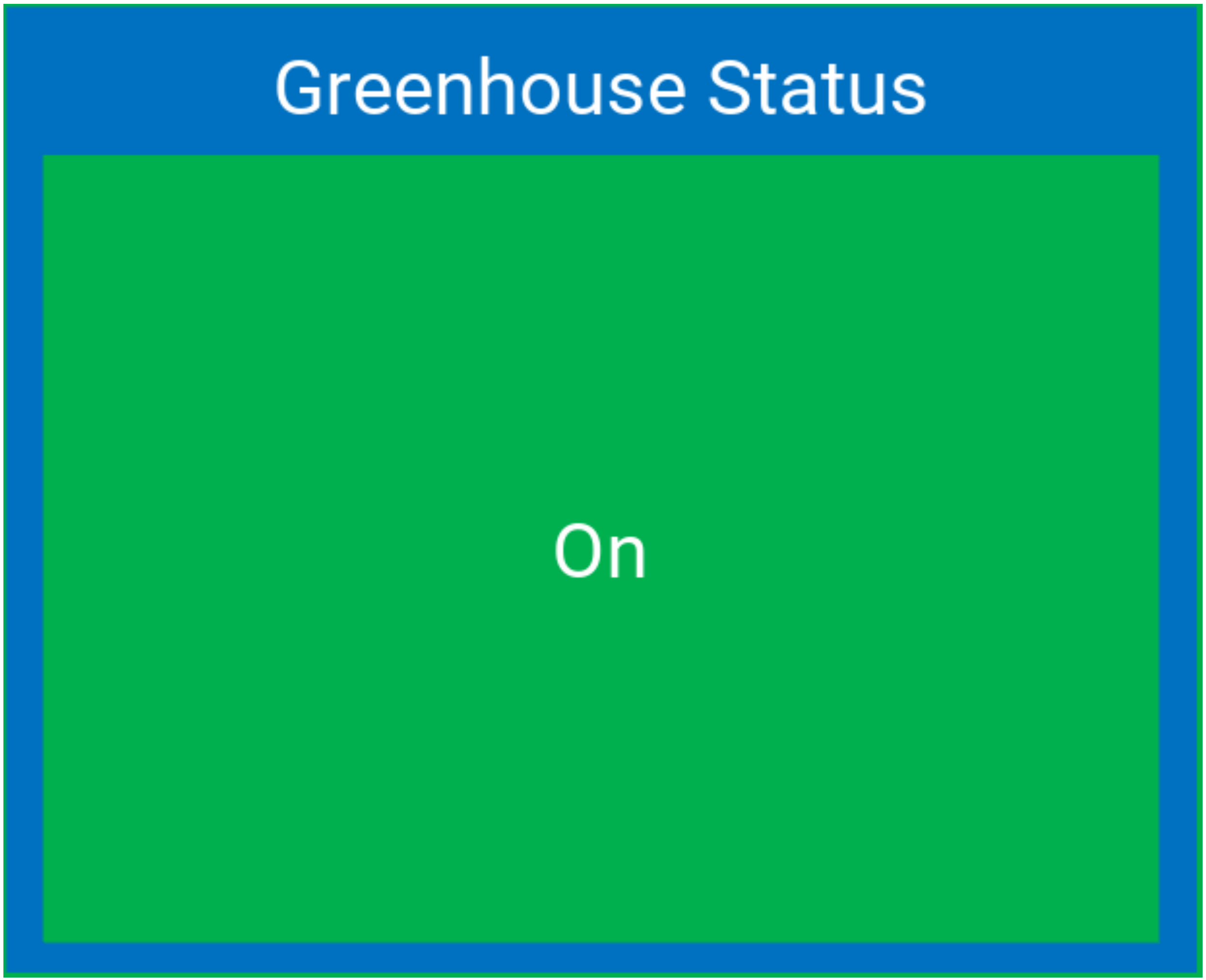
```
        #Both times are okay so save the time
        if self.startFlag and self.endFlag:
            greenHouseManager.setSetting("start", self.ids.startTime.text)
            greenHouseManager.setSetting("end", self.ids.endTime.text)

            greenHouseManager.updateSettings()

        #time is not in correct format
        else:
            self.ids.generalSettingsError.text = "Time not in correct format"
```

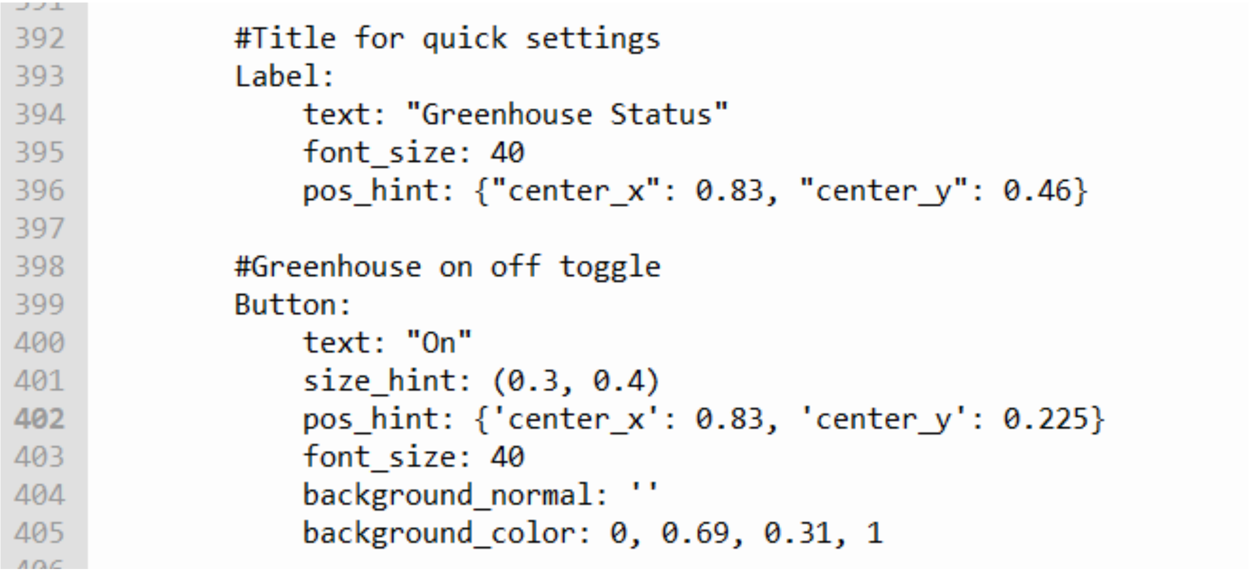
Providing both the flags are still true then it is okay to update the values for the start and end time. These values are set using the settings setter method and saved using the update settings method. In the case validation has not been passed then an error message is shown to the user.



```
                on_press: root.setTime()
```

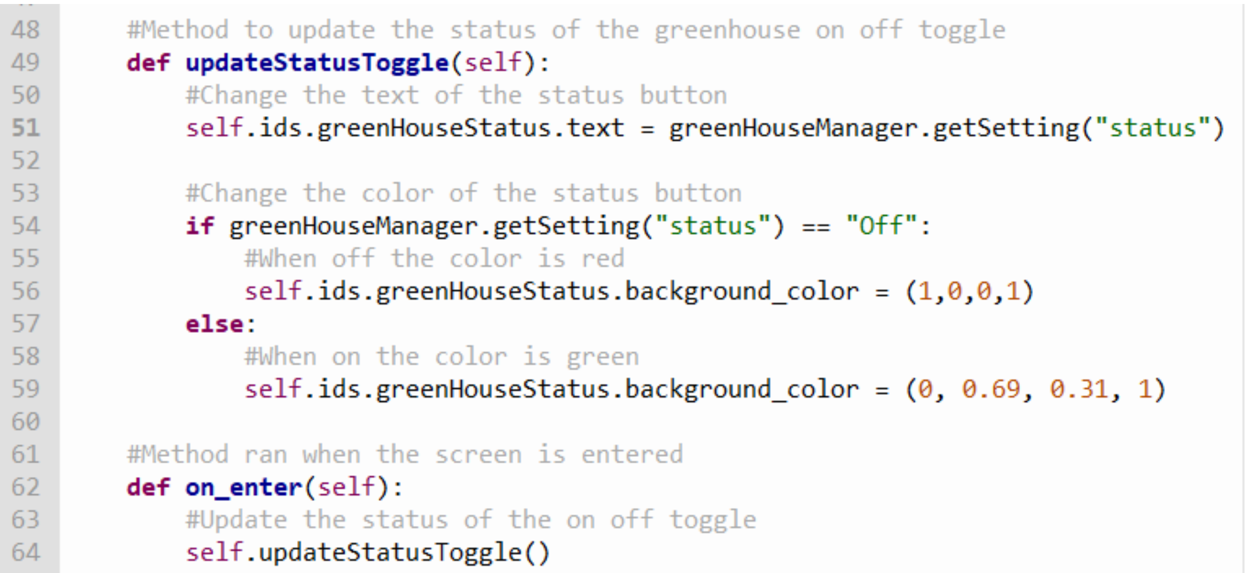
The set time method is run when the set button is pressed inside the GUI.





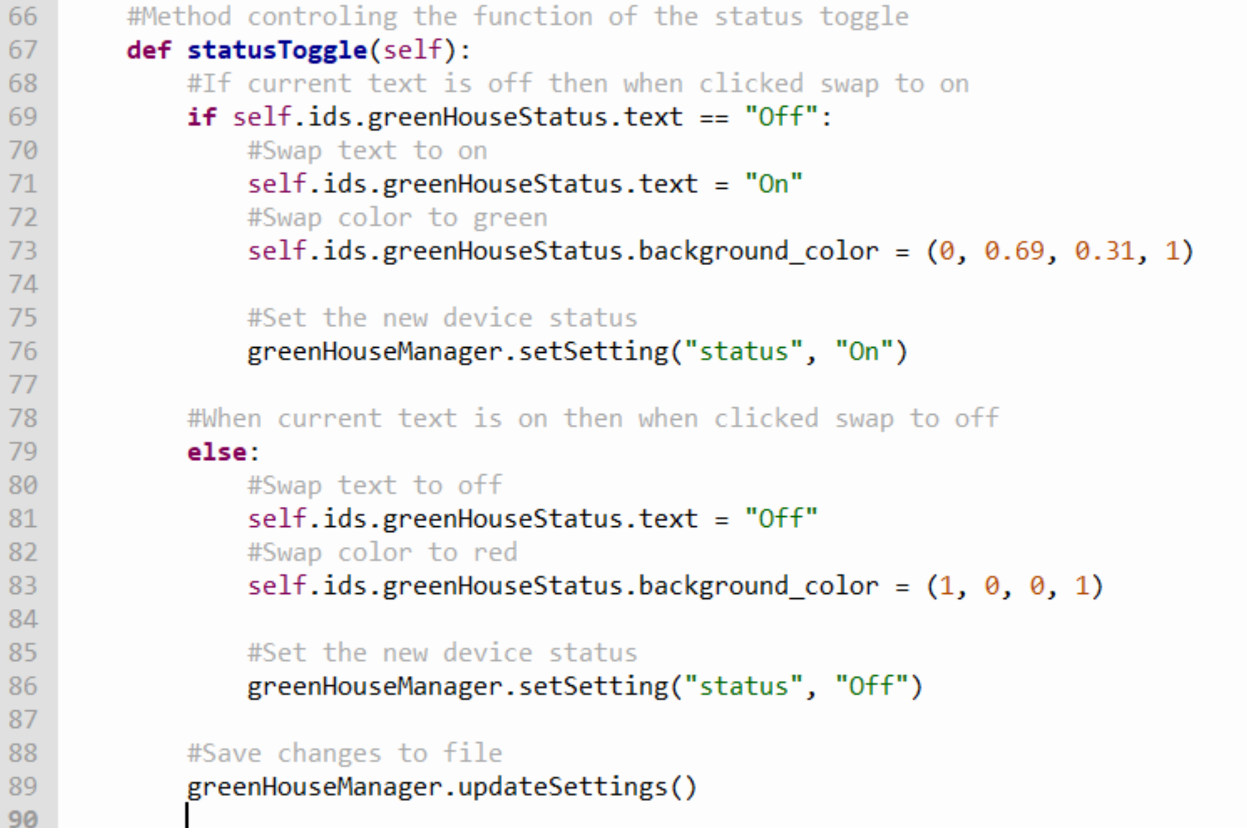
```
        #Title for quick settings
        Label:
            text: "Greenhouse Status"
            font_size: 40
            pos_hint: {"center_x": 0.83, "center_y": 0.46}

        #Greenhouse on off toggle
        Button:
            text: "On"
            size_hint: (0.3, 0.4)
            pos_hint: {'center_x': 0.83, 'center_y': 0.225}
            font_size: 40
            background_normal: ''
            background_color: 0, 0.69, 0.31, 1
```

Due to time constrains I am stripping out luxury features such as the remote access and the email alerts. This means the only setting inside the main menu quick settings section will be the on off toggle button to turn the greenhouse on and off. Due to this I've changed the quick settings area into a large greenhouse on off toggle switch. Clicking this will make the greenhouse turn on and off. Above is the kivy code and a screenshot of the button.



```
    #Method to update the status of the greenhouse on off toggle
    def updateStatusToggle(self):
        #Change the text of the status button
        self.ids.greenHouseStatus.text = greenHouseManager.getSetting("status")

        #Change the color of the status button
        if greenHouseManager.getSetting("status") == "Off":
            #When off the color is red
            self.ids.greenHouseStatus.background_color = (1,0,0,1)
        else:
            #When on the color is green
            self.ids.greenHouseStatus.background_color = (0, 0.69, 0.31, 1)

    #Method ran when the screen is entered
    def on_enter(self):
        #Update the status of the on off toggle
        self.updateStatusToggle()
```

The update status toggle sets the value and color of the status toggle and is ran on entry to the screen.



```
    #Method controling the function of the status toggle
    def statusToggle(self):
        #If current text is off then when clicked swap to on
        if self.ids.greenHouseStatus.text == "Off":
            #Swap text to on
            self.ids.greenHouseStatus.text = "On"
            #Swap color to green
            self.ids.greenHouseStatus.background_color = (0, 0.69, 0.31, 1)

            #Set the new device status
            greenHouseManager.setSetting("status", "On")

        #When current text is on then when clicked swap to off
        else:
            #Swap text to off
            self.ids.greenHouseStatus.text = "Off"
            #Swap color to red
            self.ids.greenHouseStatus.background_color = (1, 0, 0, 1)

            #Set the new device status
            greenHouseManager.setSetting("status", "Off")

        #Save changes to file
        greenHouseManager.updateSettings()
```

Finally, the status toggle method is duplicated in the main menu screen and binded to the status button.

Test plan

Test Number	Test Plan	Expected Outcome	Actual Outcome	Pass/Fail
1	Turn the status on the main menu to on	Status will be on and saved to file	Status was on and correctly saved	Pass
2	Turn the status to off in the main menu	Status will be off and saved to file	Status was off and was correctly saved	Pass
3	Turn the status to on in the settings page	Status will be on and saved to file	Status was on and saved	Pass
4	Turn the status to off in the settings page	Status will be off and saved to file	Status was off and saved	Pass
5	Enter a valid time into the start and end time boxes and set it	Time should be accepted and saved	Times were saved	Pass
6	Enter an invalid time into the time boxes and set it	Error message should be shown, and times not saved	Error message shown and no changes made to settings file	Pass
7	Enter “22:00” into the time box.	The time is not valid so will be denied	Error message shown and no changes made to settings file	Pass
8	Enter “111:00:23” into the time box	The time is not valid so will be denied	Error message shown and no changes made to settings file	Pass
9	Enter “aa:ff:ss” into the time box	Time is not valid so will be denied	Error message shown and no changes made to settings file	Pass
10	Enter “” into the time box	Time is not valid so will be denied	Error message shown and no changes made to settings file	Pass
11	Enter “10:2345” into the time box	Time is not valid so will be denied	Error message shown and no changes made to settings file	Pass

Review

The final settings from the greenhouse are now being saved into the settings.txt file. These settings are displayed to the user on both the settings page and in the main menu where the greenhouse status is shown.

Iterative stage 12- Greenhouse live measurements and device status

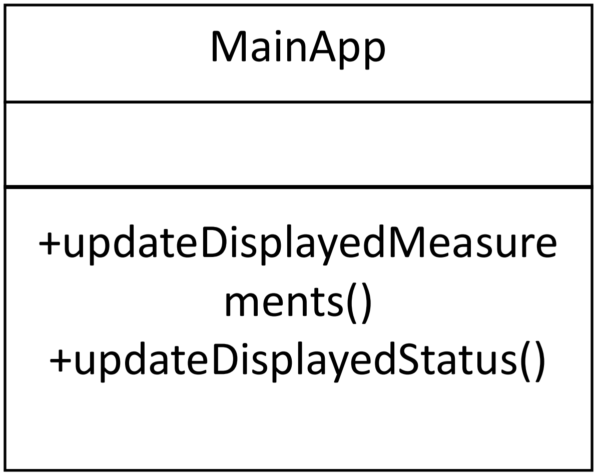
Overview

Whilst the greenhouse Gui is running the live measurements from the greenhouse and the current status of each device needs to be shown to the user. These values will need to be periodically refreshed to ensure that they are up to date. The device status will need to be stored inside the greenhouse manager class and then fetched. Whilst the enviro and moisture class will be used for the measurements.

Requirements

Two functions to update the status and measurements must be ran periodically. They should both display the time at which the reading was made. They will make use of the greenhouse manager class to get current device status, the enviro class to take sensor readings and the moisture class to see if the plant needs water. The current device status is different to the device status which the user can set on the parameters page. The current device status is to do with if a device is currently in operation such as the light being on whereas the device status is if the device is enabled by the user.

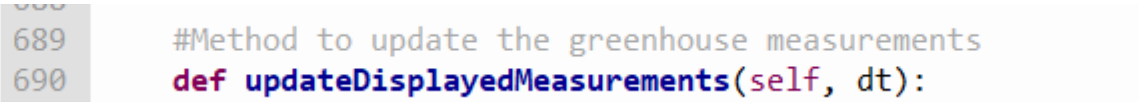
Class Diagram



The two functions will belong to the main kivy app class and will be added to the kivy clock inside the build method of the main app.

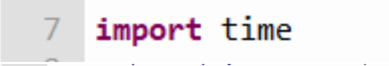
Development log

The kivy clock object allows for a function to be scheduled repeatedly without causing any interruption to the kivy gui. Without using some form of multiprocessing any functions called would cause the kivy gui to freeze for the time the function takes to execute. The clock object handles the execution of any given functions concurrently without interrupting the gui. The clock has a schedule interval method which will be most useful for this project. The method takes the function to be ran repeatedly and a time interval at which the function will be executed.

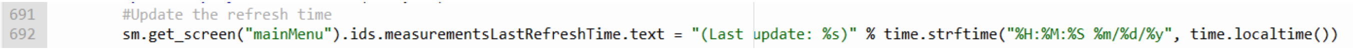


```
    #Method to update the greenhouse measurements
    def updateDisplayedMeasurements(self, dt):
```

The first method is called update displayed measurements and is responsible for updating the label values for the different measurements. This method has one parameter called dt which won't have any use inside the code I will be writing but is a required parameter for the clock object.

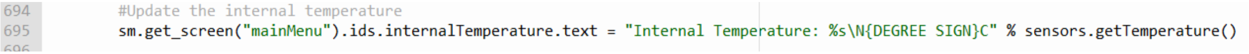


```
import time
```

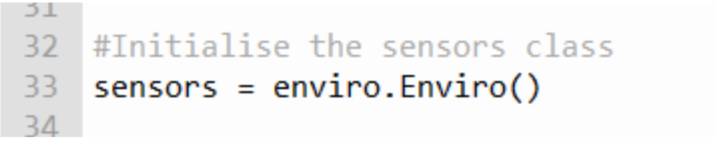


```
        #Update the refresh time
        sm.get_screen("mainMenu").ids.measurementsLastRefreshTime.text = "(Last update: %s)" % time.strftime("%H:%M:%S %m/%d/%y", time.localtime())
```

The first label to update is the time stamp label which will show the time that this function was ran and hence the time at which all the measurements were taken. Since this method belongs to the main app class of kivy it can't access kivy ids using the self-keyword. Since self can only refer to objects belonging to the current object a different method needs to be used to access ids. Each screen is added to the screen manager, so all objects' parents is the screen manager. This is the root into accessing elements from outside their class. To access an id first a screen is accessed by using get screen with the desired screen as a parameter. Then the id can be accessed as usual by referencing the id and then the parameter which is required in this case text. The value of this label needs to be set to the current time and date. The current time and date are loaded using the time library which I've imported for this job. The time library has a feature called strftime which takes a local time object as a parameter and allows for it to be formatted into a desired format. I have specified the format should be hours, minutes, and seconds and then month, day, and years. A time object is generated by doing local time which is passed as the object to be formatted into a string by the strftime method.

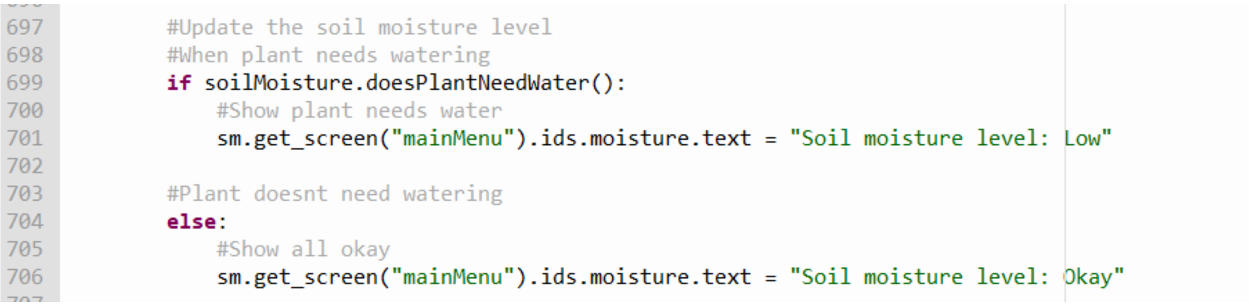


```
        #Update the internal temperature
        sm.get_screen("mainMenu").ids.internalTemperature.text = "Internal Temperature: %s\N{DEGREE SIGN}C" % sensors.getTemperature()
```



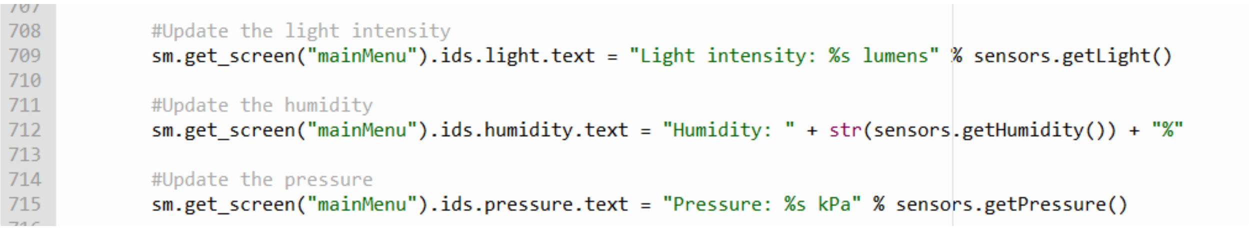
```
#Initialise the sensors class
sensors = enviro.Enviro()
```

To set the value of the internal temperature string the element is referenced in the same way as described above and then its text property is set to equal the current sensor reading from the greenhouse. Sensors is an instance of the enviro class which is responsible for getting values from the greenhouse.



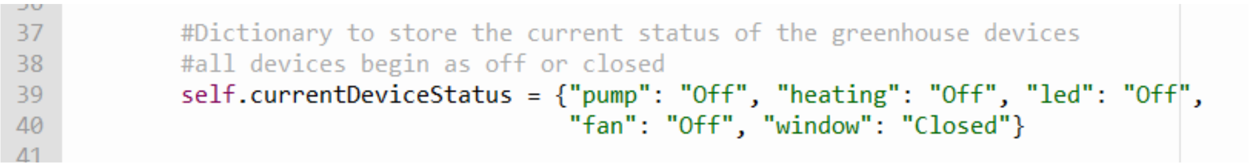
```
        #Update the soil moisture level
        #When plant needs watering
        if soilMoisture.doesPlantNeedWater():
            #Show plant needs water
            sm.get_screen("mainMenu").ids.moisture.text = "Soil moisture level: Low"

        #Plant doesnt need watering
        else:
            #Show all okay
            sm.get_screen("mainMenu").ids.moisture.text = "Soil moisture level: Okay"
```

Updating the soil moisture is a little more complex since the moisture class is written to return true when the plant needs watering and false when it does not. This would look odd if the label read "Soil moisture level: True/False". To overcome this an instance of the moisture class called soil moisture is queried. If the result is true, then the plant needs more water, and the moisture label is set to equal "soil moisture level: low" otherwise the water levels are okay, and the text is set to be "Soil moisture level: Okay".

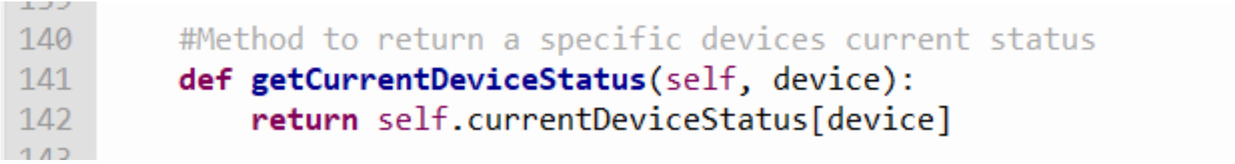


```
        #Update the light intensity
        sm.get_screen("mainMenu").ids.light.text = "Light intensity: %s lumens" % sensors.getLight()

        #Update the humidity
        sm.get_screen("mainMenu").ids.humidity.text = "Humidity: " + str(sensors.getHumidity()) + "%"

        #Update the pressure
        sm.get_screen("mainMenu").ids.pressure.text = "Pressure: %s kPa" % sensors.getPressure()
```

The final three measurements are set by getting values from the enviro class. A quick note is that I have changed the name of all the methods inside the enviro class to have get in front of them so where the method was once called "temperature" it's now called "getTemperature" this was just to better explain its job and make the code more understandable.

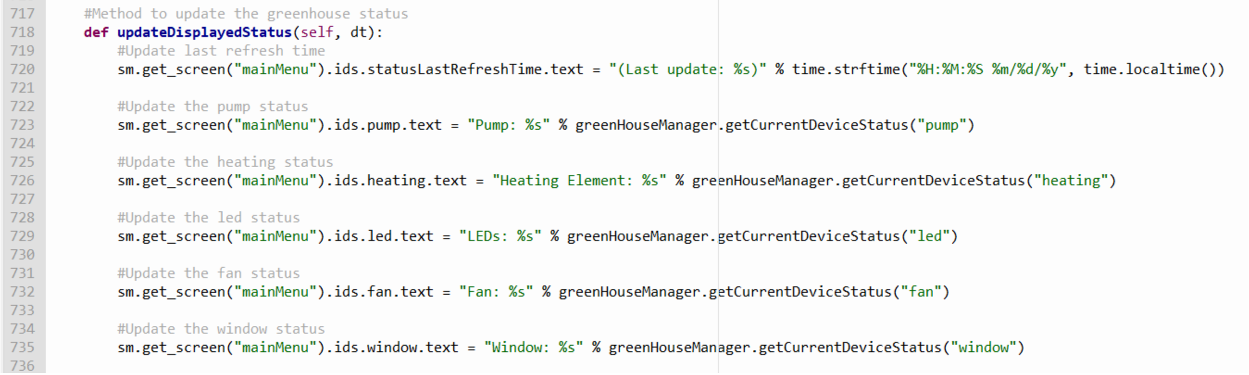


```
        #Dictionary to store the current status of the greenhouse devices
        #all devices begin as off or closed
        self.currentDeviceStatus = {"pump": "Off", "heating": "Off", "led": "Off",
                                    "fan": "Off", "window": "Closed"}
```



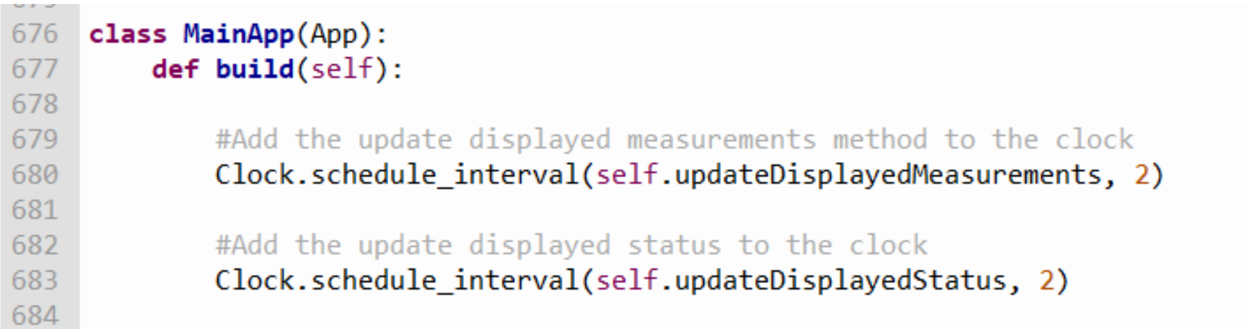
```
    #Method to return a specific devices current status
    def getCurrentDeviceStatus(self, device):
        return self.currentDeviceStatus[device]
```

To record the status of a device I'm going to be using a dictionary inside the manager class. All devices are off when the greenhouse is started so their values reflect this. I've also made a getter method to get the status of a device. It takes the device as a parameter and then returns its current value.



```
    #Method to update the greenhouse status
    def updateDisplayedStatus(self, dt):
        #Update last refresh time
        sm.get_screen("mainMenu").ids.statusLastRefreshTime.text = "(Last update: %s)" % time.strftime("%H:%M:%S %m/%d/%y", time.localtime())

        #Update the pump status
        sm.get_screen("mainMenu").ids.pump.text = "Pump: %s" % greenHouseManager.getCurrentDeviceStatus("pump")

        #Update the heating status
        sm.get_screen("mainMenu").ids.heating.text = "Heating Element: %s" % greenHouseManager.getCurrentDeviceStatus("heating")

        #Update the led status
        sm.get_screen("mainMenu").ids.led.text = "LEDs: %s" % greenHouseManager.getCurrentDeviceStatus("led")

        #Update the fan status
        sm.get_screen("mainMenu").ids.fan.text = "Fan: %s" % greenHouseManager.getCurrentDeviceStatus("fan")

        #Update the window status
        sm.get_screen("mainMenu").ids.window.text = "Window: %s" % greenHouseManager.getCurrentDeviceStatus("window")
```

The update displayed status method is very similar to the update displayed measurements method. The update time is set in the same way. For the devices, the getter method created above is used to display the current value onto the screen.



```
class MainApp(App):
    def build(self):

        #Add the update displayed measurements method to the clock
        Clock.schedule_interval(self.updateDisplayedMeasurements, 2)

        #Add the update displayed status to the clock
        Clock.schedule_interval(self.updateDisplayedStatus, 2)
```

Inside the kivy build method I have added both methods I have just created which are responsible for updating the measurements and status to the system clock. This means that the moment the app is built the values will be constantly updated. I have scheduled them to be ran every 2 seconds as during development I found this was the ideal time so that the user had time to read values, but they were not massively out of date when they did.

Test Plan

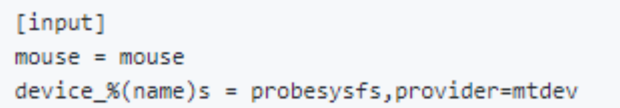
Test Number	**Test Plan**	**Expected Outcome**	**Actual Outcome**	**Pass/Fail**
1	Turn on the greenhouse and check that the internal temperature is updating	The temperature value will be updated every 2 seconds	The temperature value was updated every 2 seconds	Pass
2	Turn on the greenhouse and check that the soil moisture is updating	After watering the plant, the label should go from low to okay	The value was updated after I watered the plant	Pass
3	Turn on the greenhouse and check that the light intensity is updating	The light intensity value should be changing every 2 seconds	The value was changing every 2 seconds	Pass
4	Turn on the greenhouse and check that the humidity is updating	The humidity should be changing every 2 seconds	The value was changing every 2 seconds	Pass
5	Turn on the greenhouse and check that the pressure is updating	The pressure should be changing every 2 seconds	The value was changing every 2 seconds	Pass
6	Turn on the greenhouse and check the pump matches the dictionary value	The value from the current device status dictionary should be shown on the screen	The correct value was shown	Pass
7	Turn on the greenhouse and check the heating element matches the dictionary value	The value from the current device status dictionary should be shown on the screen	The correct value was shown	Pass
8	Turn on the greenhouse and check the led element matches	The value from the current device status dictionary	The correct value was shown	Pass

	the dictionary value	should be shown on the screen		
9	Turn on the greenhouse and check the fan element matches the dictionary value	The value from the current device status dictionary should be shown on the screen	The correct value was shown	Pass
10	Turn on the greenhouse and check the window element matches the dictionary value	The value from the current device status dictionary should be shown on the screen	The correct value was shown	Pass

Review

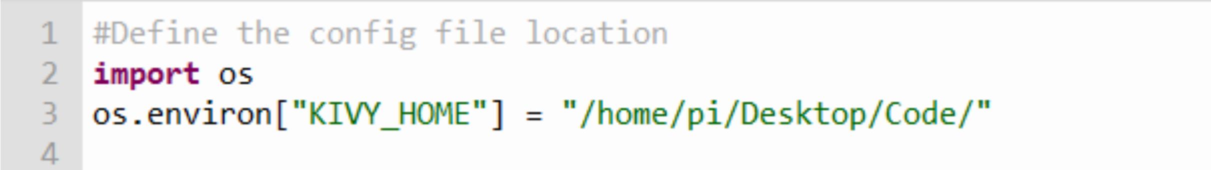
Whilst the gui is ran the greenhouse measurements and device statues are updated every 2 seconds. The system is not stopping the gui due to the clock object being used. This means that the GUI continues to function whilst the values are updated. Up until this stage I have been developing the GUI on my windows computer. However, this stage required the GUI to be ran on the Raspberry Pi for the first time as live data values from the enviro class are being taken and then displayed in the GUI. When moving over I realized that kivy was acting very strangely. One clicks of the mouse was being detected as multiple clicks in random locations on the screen by the kivy backend. This is obviously a major issue and was rendering the GUI unusable. It seems this is a raspberry pi specific issue as I've not been able to replicate the issue on my desktop. Having looked online I was not able to find any obvious solutions to this issue. Below is a table of the different steps I tired to solve the issue which has had no effect.

- Reinstall kivy
- Downgrade kivy to version 1 from version 2
- Swapped mouse
- Changed the backend window provide used by kivy
- Connecting via VNC
- Reinstalling the whole Os on the PI
- Installing a custom OS which had kivy supposedly "setup" on It



```
[input]
mouse = mouse
device_%(name)s = probesysfs,provider=mtdev
```

The only solution I found to this problem was to remove a line from the kivy config file relating to the function of the mouse. Above is a screenshot of the default kivy config file on the raspberry pi. For some reason removing the "device_%" line fixed all the issues with the mouse. The kivy config file is stored in the following path by default "<HOME_DIRECTORY>/.kivy/config.ini". I was successfully able to modify the config file for the "Pi" user account. However, if you recall to the LEDs iterative stage, I am being forced to run the neopixels library using sudo. This means the python installation used will be the sudo root accounts and not the pi account. The root account is a protected directory, and I was not able to edit the root accounts config file so the issue would persist when running the complete program.



```
#Define the config file location
import os
os.environ["KIVY_HOME"] = "/home/pi/Desktop/Code/"

```

To get around this issue for good I am now having to specify the path to the corrected config file. In this case I am storing the correct config file with the line removed inside the same directory as the greenhouse code. The os library is used to set kivy environmental variables. I have set the kivy home directory to the path of folder containing the correct config file. This gets around the restriction on editing the root users kivy config file and means the mouse click issue is solved.

Iterative stage 13 – System log

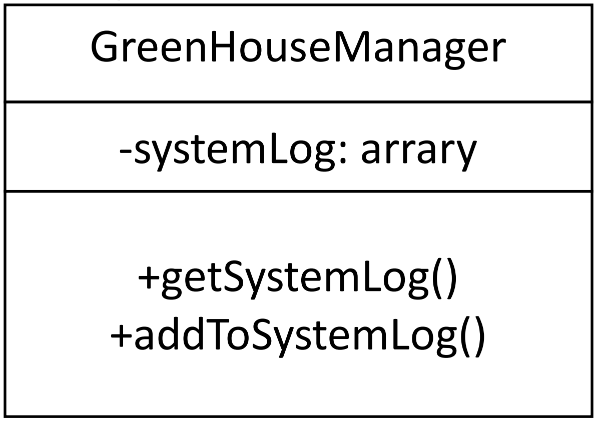
Overview

This stage will focus on implementing the required methods to perform the functions of the system log. The system logs job is to display to the user any events that are performed by the to be developed main management method such as the window being opened. As mentioned during the Kivy GUI iterative stage I have not been able to get the scroll view working. In the interests of time, I have decided to not mess about any further with the scroll view and instead swap to a label which I can just update the text of. This means the user will not be able to scroll back to view past events, but I feel there is not much to be gained from a historic view.

Requirements

The system log will be responsible for showing the user recent actions carried out by the greenhouse. The log will work in the style of a queue with new events being added to the end of the log and then when the maximum number of events which can be shown on the screen is reached the oldest element in the log will be removed. The system log will be stored in the form of an array so I can keep the items in order. One method will need to govern adding events to the system log and another will return the log. The system log will be implemented into the greenhouse manager class which I'm using to perform all the management tasks of the greenhouse.

Class Diagram



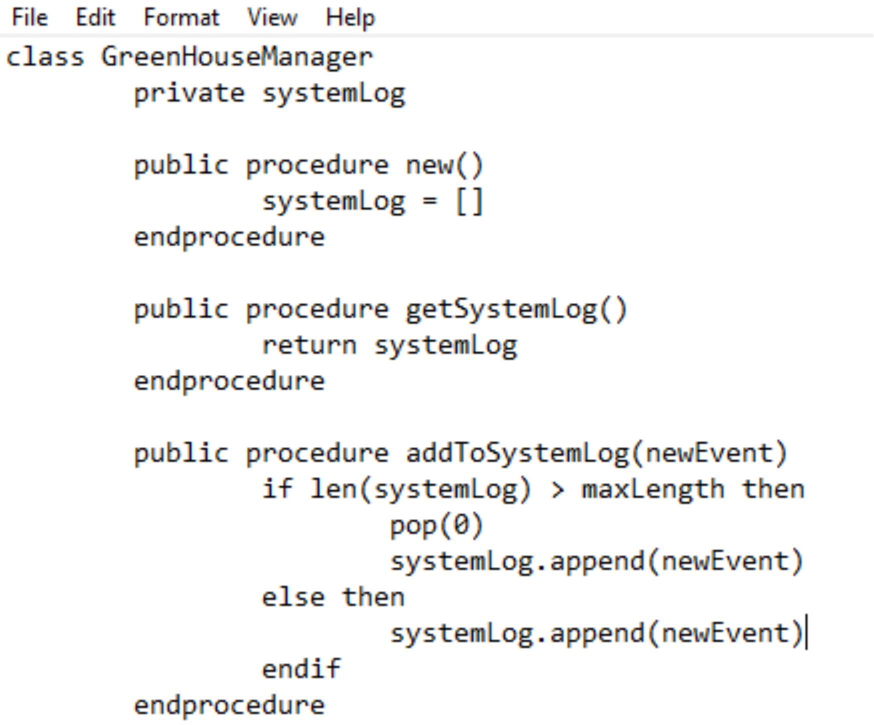
GreenHouseManager

-systemLog: array

+getSystemLog()
+addToSystemLog()

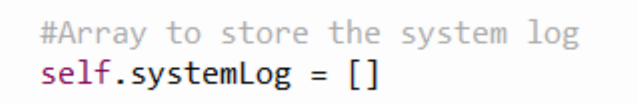
The system log array is going to be created when the class is initialized and as a private array. The public getter method get system log will be used to return the contents of the array. Whilst the add to system log method Is going to add new events to the array and make sure it does not go over size.

Pseudocode



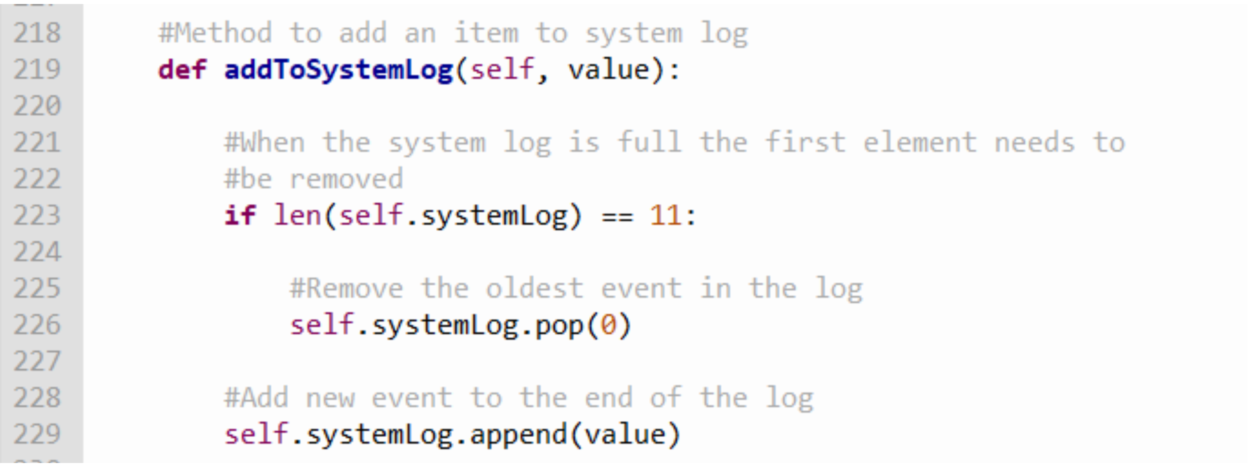
```
class GreenHouseManager
    private systemLog

    public procedure new()
        systemLog = []
    endprocedure

    public procedure getSystemLog()
        return systemLog
    endprocedure

    public procedure addToSystemLog(newEvent)
        if len(systemLog) > maxLength then
            pop(0)
            systemLog.append(newEvent)
        else then
            systemLog.append(newEvent)
        endif
    endprocedure
```

Development log



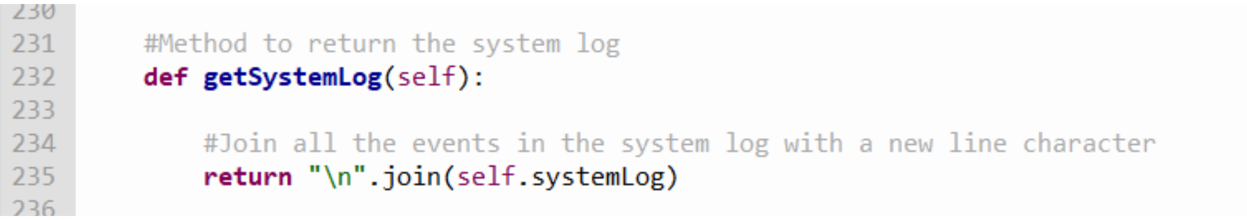
```
#Array to store the system log
self.systemLog = []
```

Following my pseudocode, I have created a blank array inside the class constructor of the manager class which I will use to store the greenhouse events.



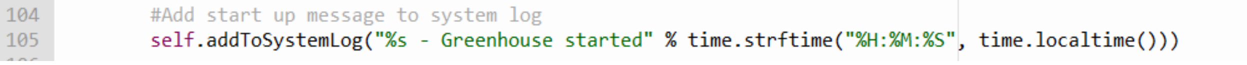
```
    #Method to add an item to system log
    def addToSystemLog(self, value):

        #When the system log is full the first element needs to
        #be removed
        if len(self.systemLog) == 11:

            #Remove the oldest event in the log
            self.systemLog.pop(0)

        #Add new event to the end of the log
        self.systemLog.append(value)
```

Having done some quick testing inside kivy I feel the maximum number of lines that can be shown in the space I've left for the system log is 11. Since the array is private there is no scenario the length of this array will ever be allowed to go above 11 as every time a new event is added it must use the setter method add to system log. For this reason, I am just checking if the length of the array is 11 as apposed to using an inequality sign. This is going to be true for all, but the first 11 events added to the log after system start. In this case adding another event is going to make the log too large so I pop the oldest event from the list in index 0. After that regardless of whether the array is oversize of now, I want to append the value passed to the method to the end of the system log.



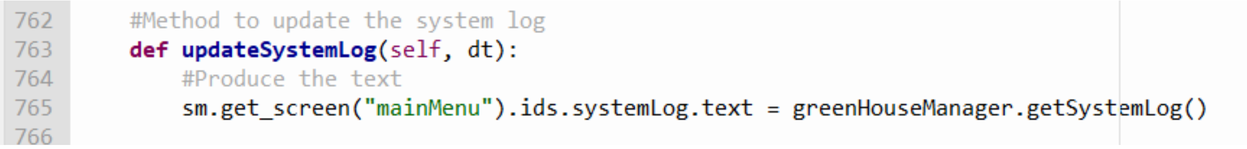
```
    #Method to return the system log
    def getSystemLog(self):

        #Join all the events in the system log with a new line character
        return "\n".join(self.systemLog)
```

Originally, I was going to just return the array via the get system log method. However, I decided it would work better if I formatted the array into a string which can then easily be displayed inside python. To do this I'm just the join method. Which joins all the elements inside the system log with a new line character. This means when the string is displayed in kivy it will appear as 11 lines each of a unique system event.



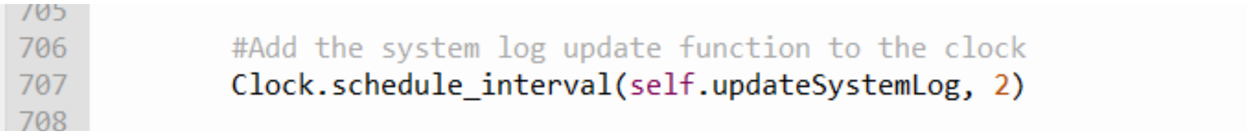
```
        #Add start up message to system log
        self.addToSystemLog("%s - Greenhouse started" % time.strftime("%H:%M:%S", time.localtime()))
```

I have taken the liberty to add a system event to the log inside the class constructor to notify the user that the greenhouse has started. All system events will begin with a time stamp to let the user know when the event happened. To do this I am using the same time method as before with slightly different formatting so that the hour, mins, and seconds are displayed. This is the format that will be used for all system log events time stamp and then the event that has occurred.



```
    #Method to update the system log
    def updateSystemLog(self, dt):
        #Produce the text
        sm.get_screen("mainMenu").ids.systemLog.text = greenHouseManager.getSystemLog()
```

The system log is going to need to be continually updated during the running of the greenhouse just like for the device measurements and status. To do this a method called update system log has been written inside the kivy main app class. As this method will be added to the clock it has the unused dt parameter. This method simply sets the text value of the system log element to be equal to the string that is returned by the get system log.



```
        #Add the system log update function to the clock
        Clock.schedule_interval(self.updateSystemLog, 2)
```

This method has then been added to the clock during the built method of the main app so that it will be ran continually at 2 second intervals if the greenhouse is in operation.

Testing plan

As I have not developed the main management class which will be controlling the greenhouse environment no events will be automatically added to the system log. For this reason, I am going to have to manually add events to the log to check that is working. I will add the events using the add event method so that it is done in the same way as it will be used later in development when the events are added depending on actions taken by the greenhouse.

Test Number	**Test Plan**	**Expected Outcome**	**Actual Outcome**	**Pass/Fail**
1	Add 1 event to the system log	It should be displayed on the system log	The event was displayed in the system log	Pass

2	Add 11 events to the system log	They should all be displayed on the screen	The events were all added to the screen	Pass
3	Add a 12th event	The oldest event should be removed, and then newest event should be added to the bottom of the log	The oldest event was removed, and the new event added to the end of the log	Pass

Review

The system log would at first seem like quite a complex problem however it was one of the faster features to implement. Given more time I would have tried to get the scroll view working so that the user could scroll through all the system events that have occurred. I do have a feeling that the issues with the scrollview not working were more on the side of kivy. If I was to do this project again, I would be using a different graphical user interface module which is more robust than kivy. So far, all the major issues I have faced have been down to external libraries such as kivy and neopixels as opposed to logic errors with my own written code. This is very frustrating as despite following the help documents for these libraries problems still occur which take countless hours to fix which could be better spent. Another feature I have not included is the ability to export and save system log events. This would be a useful feature for debugging for the end user which would have been nice to implement given more time.

Iterative stage 14 – A few adjustments to the GUI

Overview

As previously alluded to I am going to be removing the remote access section of this project. I have also regrettably decided to strip out the email alerts feature and the ability to change the current setting file. Below I will discuss how I would have implemented these features and outline the changes to the GUI that I have done to remove these sections. When thinking through how the greenhouse will function it has come to my attention that the user will want to be able to select the speed at which the LEDs run at and that a demo feature might be handy. This feature would just turn on all devices and would be ideal for demonstration purposes to potential clients. So, I will be implementing these two new features quickly as they both draw on code that has either been developed or will be developed later.

Development log –

Remote access

The remote access feature was going to include a login log much like the system log I've just implemented this would let the user know when somebody logs in to the system remotely and other login events. I would have implemented this log in the same was as the system log. For the actual remote access part, I had not completed much research into how to implement this. However, I would have been looking for a library that supported the implementation of remote access to a specific raspberry pi application. The libraries that spring to mind is putty, VNC or a variant of SSH which supported remote desktop. The key to this would have been the ability to limit the access to just the greenhouse application as there are many applications such as vnc which out of the box provide remote access to the raspberry pi. Since this feature is no longer going to be part of the project, I have removed the section inside the settings page for it.

Settings file

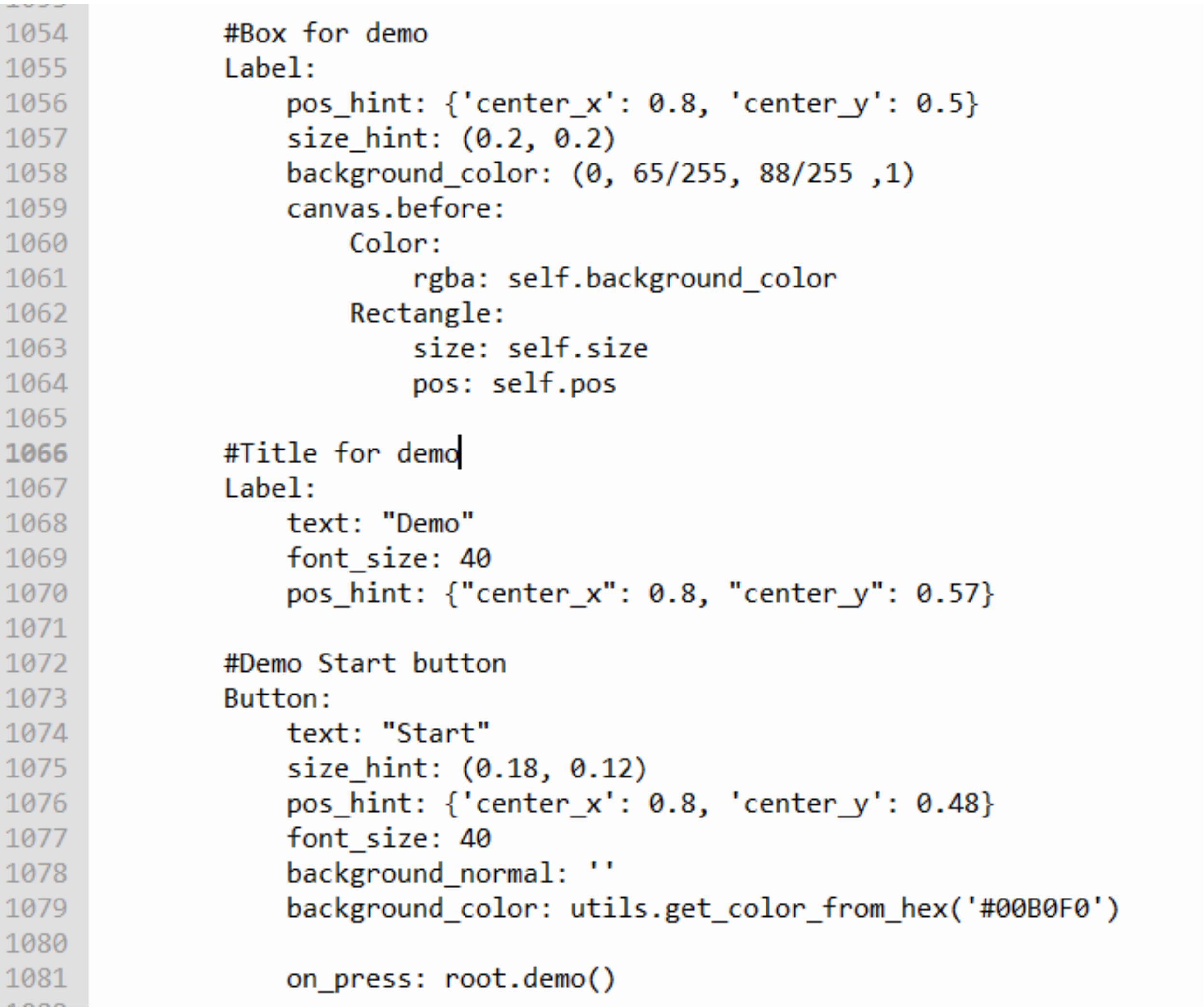
The user was originally going to have the ability to swap between different saves so that if they swapped the plant inside the greenhouse, they could select a previously used settings file to load the right parameters etc for that plant. To implement this feature, I was planning on creating a new folder each time a user made a new save file. This folder would be the name of the save which the user would see when they select a save from the dropdown. This folder would contain the settings, parameters and devices text files which are the 3 files which store all the data for this project. When the user selected a new save I would either copy the contents of the folder into the same path as the main python files or adjust the path inside the various functions which accessed and wrote to these files so that they point to the correct folder. I would have likely gone for the copy method as it would have saved me having to edit the file paths wherever I have opened the files inside python. I've simply removed the kivy code inside the kv file so that the settings section of the full settings page is no longer shown.

Email alerts

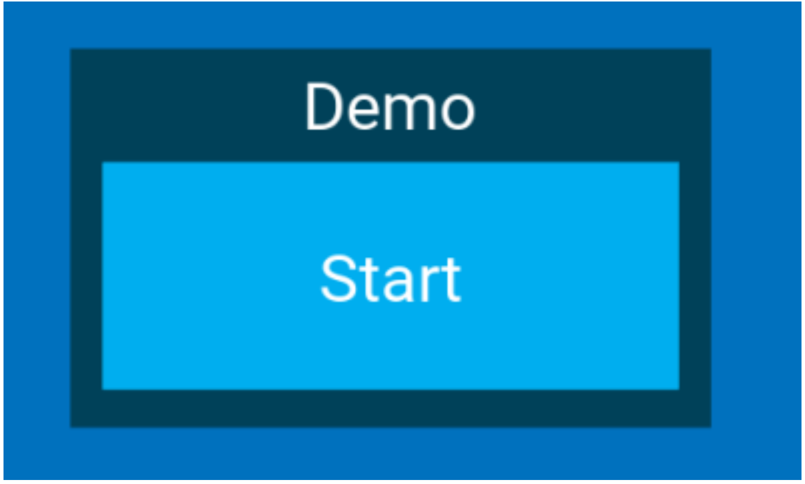
The email alerts feature would have been straight forward to implement using the python smtpd library. I would have written a function which when called sent an email to the user's email detailing the current readings of the greenhouse and a couple of other stats such as the average temperature during the day. The smtpd library needs an email server to send the mail from and for this I would have probably used gmail as its free and they give full access to the required features to link to smtpd. I would have then added this function to the clock at the interval set by the user so that an email was periodically sent out. This would have been a nice feature to implement but I've had to axe it due to time constraints. This section has been removed from the settings page too.

Demo button

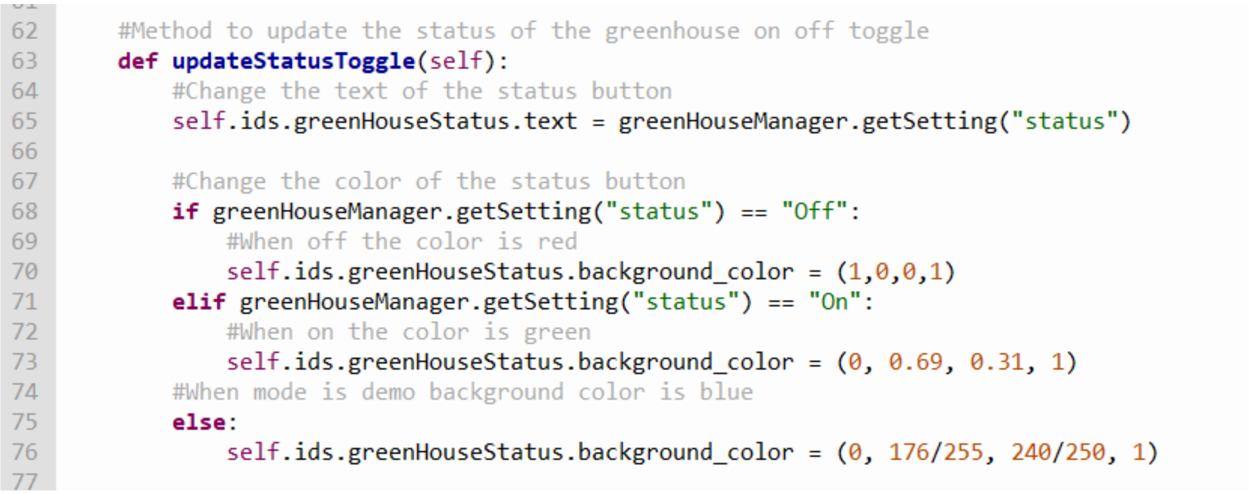
In place of the free space created on the full settings page I am going to be adding a demo button which when pressed will run through a demo of the greenhouse.



```
        #Box for demo
        Label:
            pos_hint: {'center_x': 0.8, 'center_y': 0.5}
            size_hint: (0.2, 0.2)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Title for demo
        Label:
            text: "Demo"
            font_size: 40
            pos_hint: {"center_x": 0.8, "center_y": 0.57}

        #Demo Start button
        Button:
            text: "Start"
            size_hint: (0.18, 0.12)
            pos_hint: {'center_x': 0.8, 'center_y': 0.48}
            font_size: 40
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            on_press: root.demo()
```

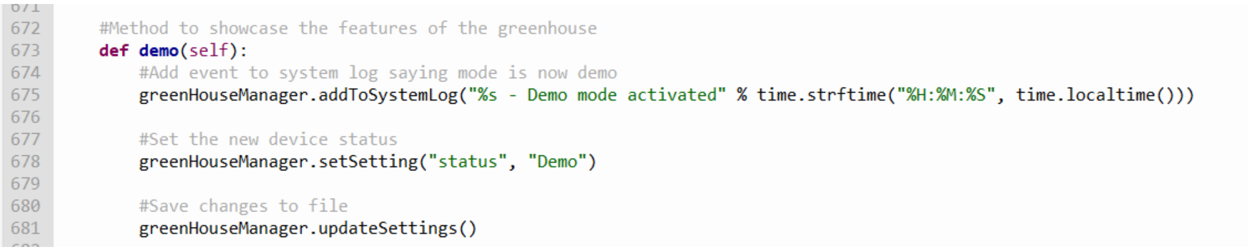


Above is the code for the demo button. It features a dark blue background container that I've made using a label. A title to let the user know what the button will do and the button itself which when pressed is going to run a method called demo which will put into action the steps required to turn on all the devices in the greenhouse. There is also a screenshot of how this demo button looks above.



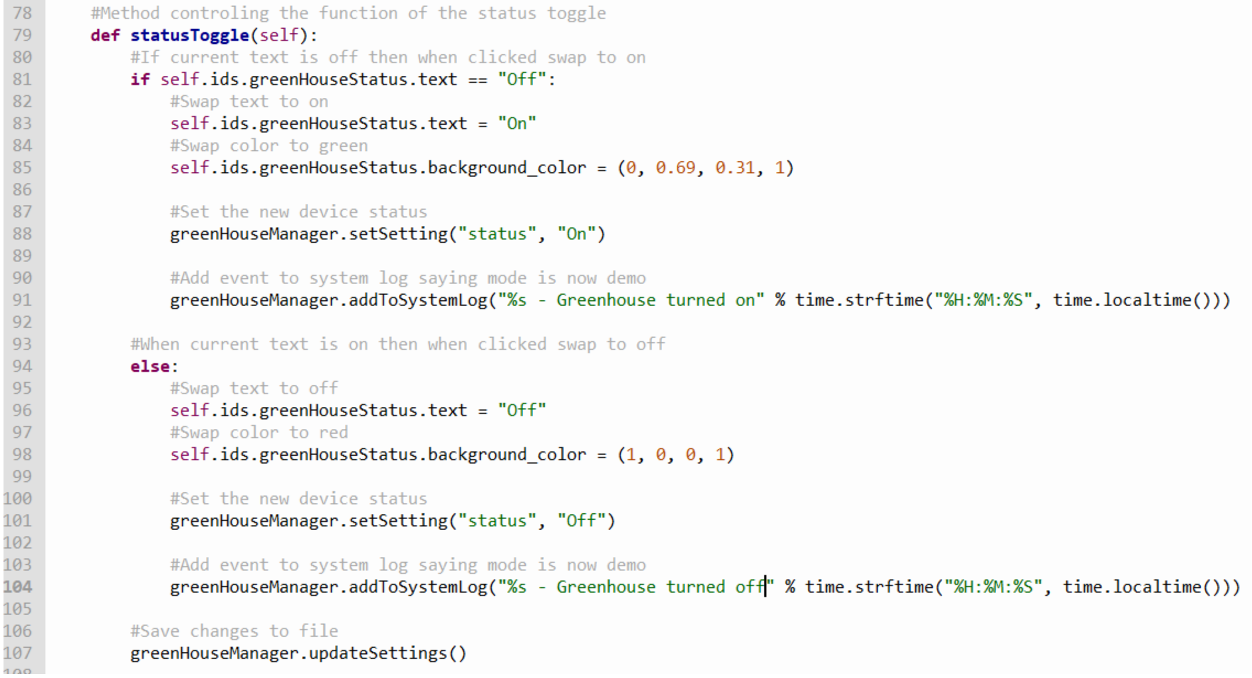
```
62      #Method to update the status of the greenhouse on off toggle
63      def updateStatusToggle(self):
64          #Change the text of the status button
65          self.ids.greenHouseStatus.text = greenHouseManager.getSetting("status")
66
67          #Change the color of the status button
68          if greenHouseManager.getSetting("status") == "Off":
69              #When off the color is red
70              self.ids.greenHouseStatus.background_color = (1,0,0,1)
71          elif greenHouseManager.getSetting("status") == "On":
72              #When on the color is green
73              self.ids.greenHouseStatus.background_color = (0, 0.69, 0.31, 1)
74          #When mode is demo background color is blue
75          else:
76              self.ids.greenHouseStatus.background_color = (0, 176/255, 240/250, 1)
```

Currently there are two modes for the greenhouse on and off. These are both selected using the big status button on the main screen and are then saved into the settings file. Later, I'm going to be using this value as a flag for if the greenhouse manager runs its main management function to turn on and off devices. I'm going to add a 3rd mode called demo which if equal to the current setting will trigger a special demo function as opposed to the normal management algorithm. I want the status button on the front page to shown when the mode is demo. So, to do this I have added a new section to the update status toggle method which is responsible for updating the appearance of the button when the user enters the main menu screen. Lines 74-76 now account for the final case where the mode is not off and not on so hence must be demo. In this case the color of the button is made light blue which is going to be the theme of demo which matches the demo button is made a second ago on the settings page. I don't want the user to be able to select demo mode from the main screen, so I've left the status toggle method the same which is responsible for changing the mode when the user clicks the status button on the main menu. As it stands when the user is in demo mode and clicks on the status button the greenhouse will swap to off mode and from there the user can click again to go to on mode.



```
    #Method to showcase the features of the greenhouse
    def demo(self):
        #Add event to system log saying mode is now demo
        greenHouseManager.addToSystemLog("%s - Demo mode activated" % time.strftime("%H:%M:%S", time.localtime()))

        #Set the new device status
        greenHouseManager.setSetting("status", "Demo")

        #Save changes to file
        greenHouseManager.updateSettings()
```

The demo method belongs to the settings screen class and is going to be called when the user clicks the demo button. Making use of the new system log I have first added a system event which will pop up on the system log to let the user know demo mode has been entered. Then the status setting is changed to demo mode and finally the settings file is saved so that the changes will be loaded next time the greenhouse is started.

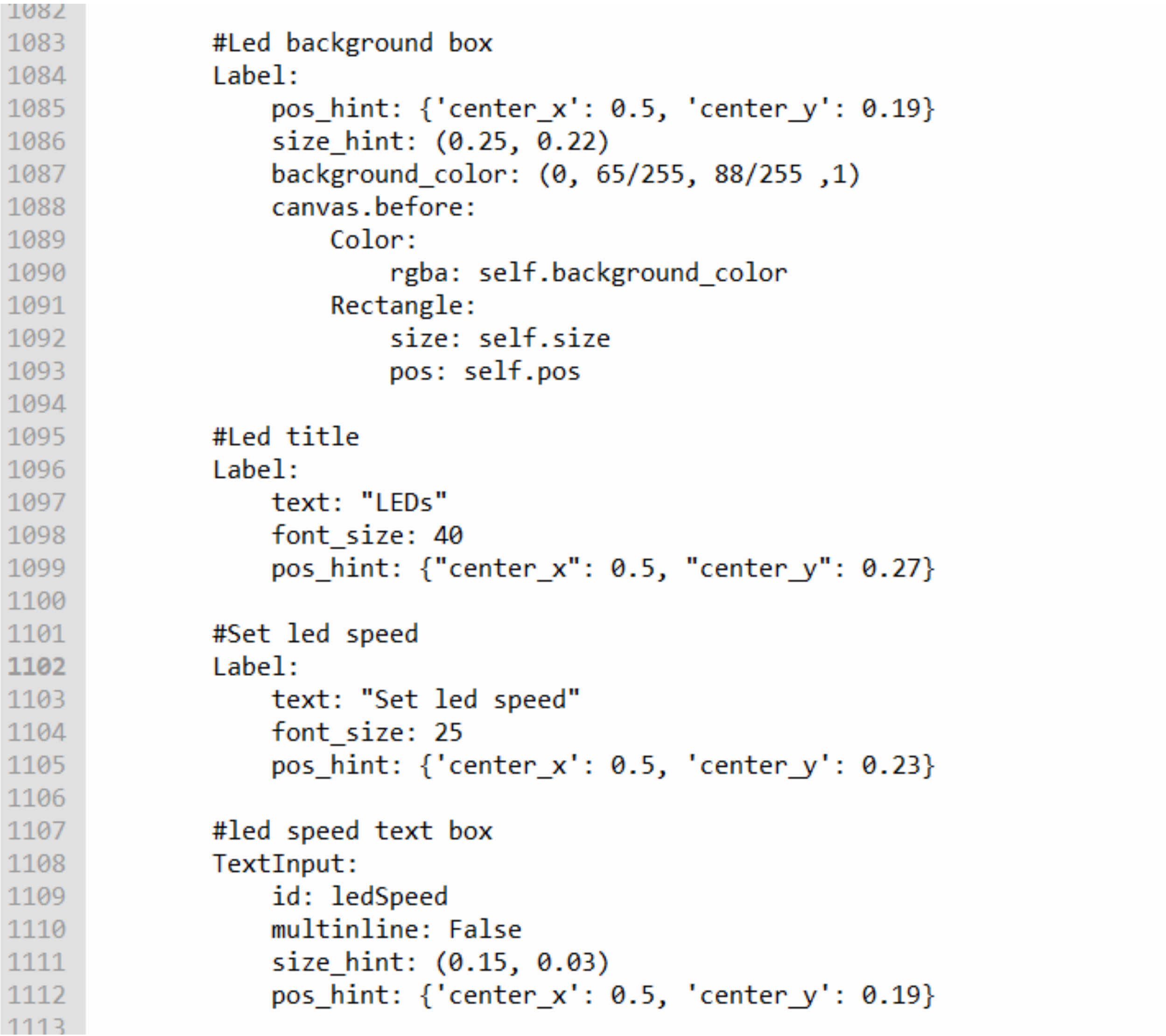


```
78      #Method controling the function of the status toggle
79      def statusToggle(self):
80          #If current text is off then when clicked swap to on
81          if self.ids.greenHouseStatus.text == "Off":
82              #Swap text to on
83              self.ids.greenHouseStatus.text = "On"
84              #Swap color to green
85              self.ids.greenHouseStatus.background_color = (0, 0.69, 0.31, 1)
86
87              #Set the new device status
88              greenHouseManager.setSetting("status", "On")
89
90              #Add event to system log saying mode is now demo
91              greenHouseManager.addToSystemLog("%s - Greenhouse turned on" % time.strftime("%H:%M:%S", time.localtime()))
92
93          #When current text is on then when clicked swap to off
94          else:
95              #Swap text to off
96              self.ids.greenHouseStatus.text = "Off"
97              #Swap color to red
98              self.ids.greenHouseStatus.background_color = (1, 0, 0, 1)
99
100             #Set the new device status
101             greenHouseManager.setSetting("status", "Off")
102
103             #Add event to system log saying mode is now demo
104             greenHouseManager.addToSystemLog("%s - Greenhouse turned off" % time.strftime("%H:%M:%S", time.localtime()))
105
106         #Save changes to file
107         greenHouseManager.updateSettings()
```

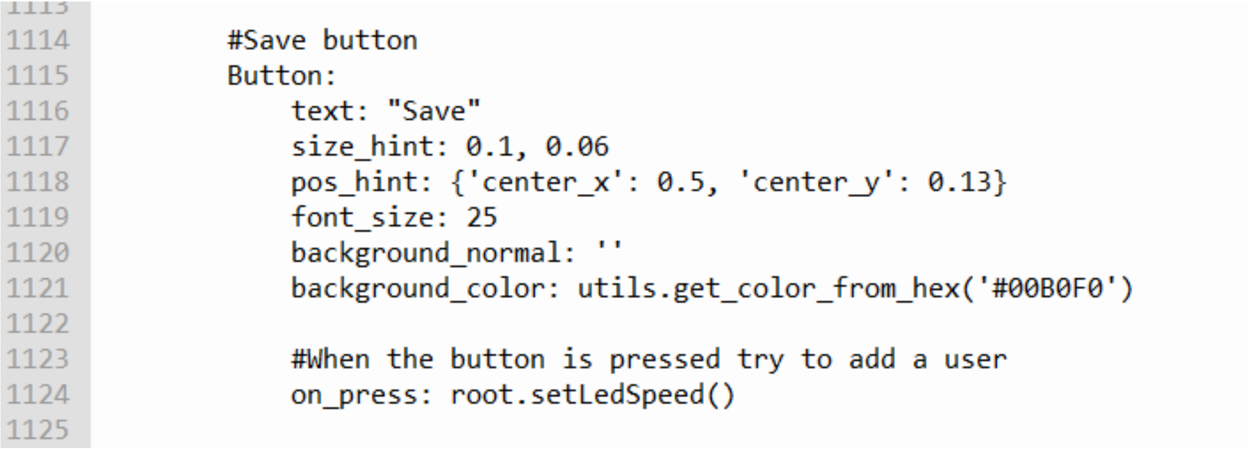
Now that the system log has been implemented, I have added two events to the status toggle method on line 91 and 104 which will let the user know that they have turned the greenhouse on and off using the big status toggle button on the main menu.

LED speed setting

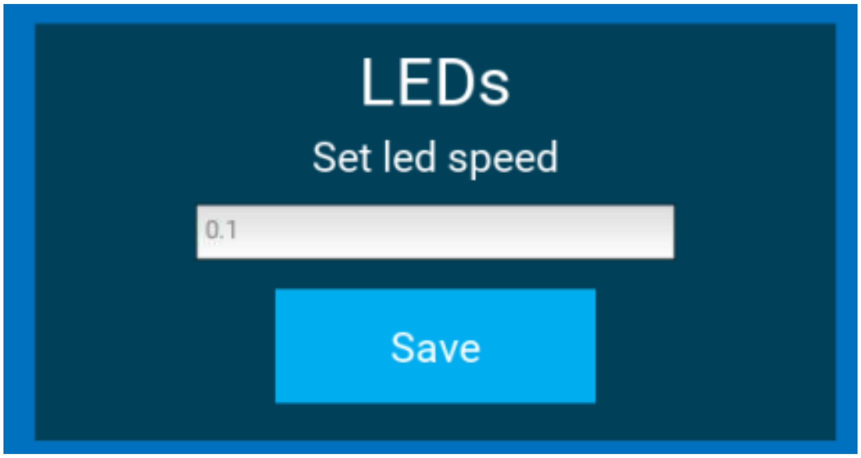
The led class has two functions the snake and flash which both make use of delays to dictate how fast they move. I have decided that the user will be able to select the time delay themselves. To do this I'm going to be adding a new setting to the settings text file and then adding a section on the settings page to allow the user to enter the speed.



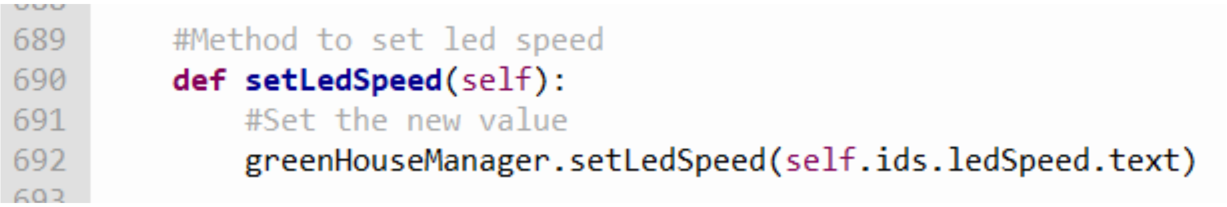
```

        #Led background box
        Label:
            pos_hint: {'center_x': 0.5, 'center_y': 0.19}
            size_hint: (0.25, 0.22)
            background_color: (0, 65/255, 88/255 ,1)
            canvas.before:
                Color:
                    rgba: self.background_color
                Rectangle:
                    size: self.size
                    pos: self.pos

        #Led title
        Label:
            text: "LEDs"
            font_size: 40
            pos_hint: {"center_x": 0.5, "center_y": 0.27}

        #Set led speed
        Label:
            text: "Set led speed"
            font_size: 25
            pos_hint: {'center_x': 0.5, 'center_y': 0.23}

        #led speed text box
        TextInput:
            id: ledSpeed
            multinline: False
            size_hint: (0.15, 0.03)
            pos_hint: {'center_x': 0.5, 'center_y': 0.19}

```



```

        #Save button
        Button:
            text: "Save"
            size_hint: 0.1, 0.06
            pos_hint: {'center_x': 0.5, 'center_y': 0.13}
            font_size: 25
            background_normal: ''
            background_color: utils.get_color_from_hex('#00B0F0')

            #When the button is pressed try to add a user
            on_press: root.setLedSpeed()

```

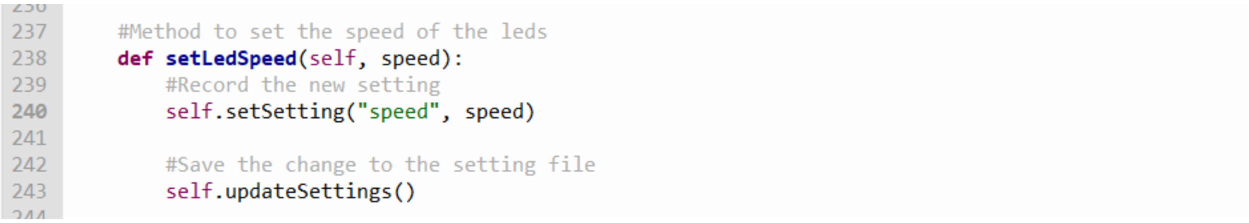


Above is a screenshot of the kivy code which produces the LEDs section where the user is going to be able to enter a custom speed/time delay for the leds.



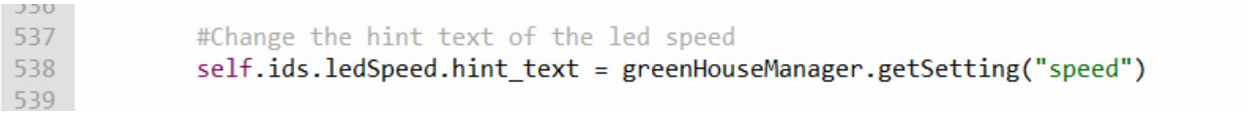
```
    #Method to set led speed
    def setLedSpeed(self):
        #Set the new value
        greenHouseManager.setLedSpeed(self.ids.ledSpeed.text)
```

The save button is binded to a method called set led speed which in turn calls a method of green house manager called set led speed with an argument of the text value of the text input box passed to it.



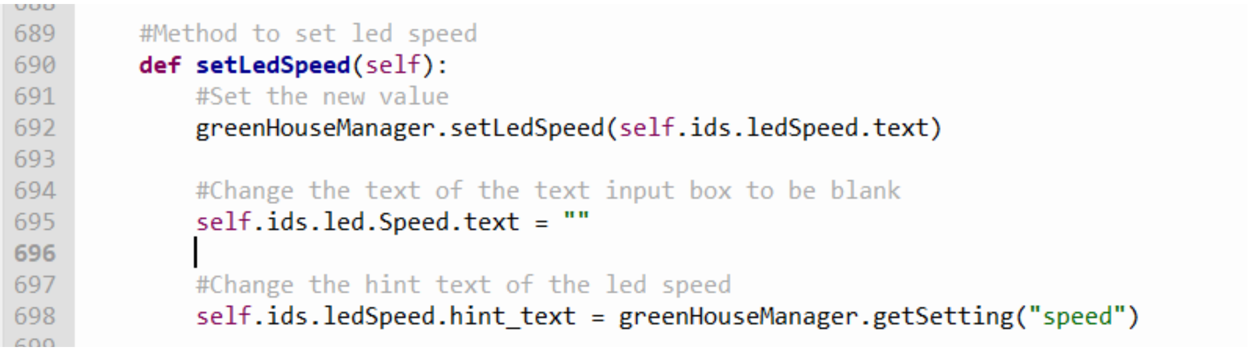
```
    #Method to set the speed of the leds
    def setLedSpeed(self, speed):
        #Record the new setting
        self.setSetting("speed", speed)

        #Save the change to the setting file
        self.updateSettings()
```

The set led speed method of the greenhouse manager class calls the set setting method to set the value of speed and then uses update settings to save this to the file. The advantage of having my setting stored in a dictionary as opposed to an array is situations such as this. Where I am adding a brand-new setting. If I was using an array, then passing speed for the first time to set setting would cause an index error as the method would try to assign the speed value to the index which does not currently exist. A dictionary on the other hand first looks to see if that key is in the dictionary and if so, updates its value and if not just makes a new key with no error.



```
        #Change the hint text of the led speed
        self.ids.ledSpeed.hint_text = greenHouseManager.getSetting("speed")
```

At the end of the update displayed general settings function I have added this line so that the hint text of the speed input box is updated to be equal to the current value of the speed. This just helps to let the user know what they need to enter and the current value for references.

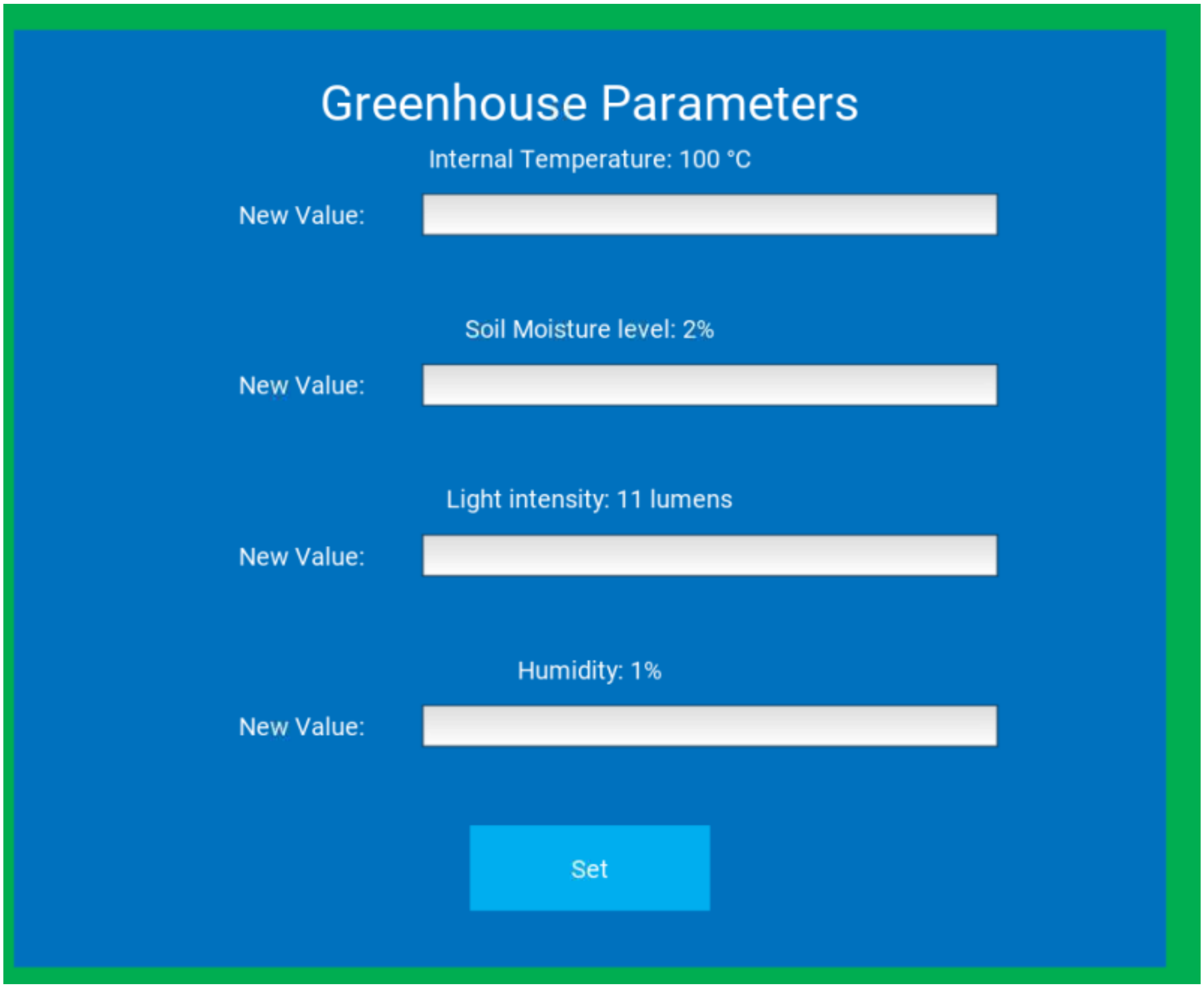


```
    #Method to set led speed
    def setLedSpeed(self):
        #Set the new value
        greenHouseManager.setLedSpeed(self.ids.ledSpeed.text)

        #Change the text of the text input box to be blank
        self.ids.led.Speed.text = ""

        #Change the hint text of the led speed
        self.ids.ledSpeed.hint_text = greenHouseManager.getSetting("speed")
```

As the update displayed general settings function is only called on entry to the page there is a situation where the user updates the value and then the hint text is still equal to the old value until the user leaves the page and reenters the page so for this reason I've added the same line to the end of the set led speed method so that it's also updated when the value is set. This line needs to be in both functions as the on-entry case makes sure its up to date when the gui is loaded and the set led speed case if for when the user makes a change to the speeds value. I've also set the text value of the text input box to blank so that after a user enters a new value and saves it the box is blank and ready for next use.

Pressure

I have also realised that there is no effective way with the equipment I have installed in the greenhouse to affect the pressure inside the greenhouse. This means the user should not be able to adjust the pressure parameter on the parameters page as I'm not going to be monitoring this value.



Due to this I have taken out the pressure text box from the green house parameters area and rearranged the other elements, so they fit together. The user is no longer able to set a desired value for pressure and I have also taken the parameter out of the parameter text file. I also had to remove the validation in the update parameters method so that the method did not try access a text input box which no longer exists, and I also removed the section of update displayed device parameter which set the pressure value on the parameters page. The pressure is still displayed to the user on the main menu they just no longer can tell the greenhouse what their ideal pressure is since the greenhouse hasn't got any mechanism to reach that target.

Test plan

Test Number	**Test Plan**	**Expected Outcome**	**Actual Outcome**	**Pass/Fail**
1	Enter a led speed and click save	The value should be saved to the settings file	The value was saved to the setting file	Pass
2	Click the demo button	The mode should swap to demo and be saved in the settings file	The mode was set to demo and saved and	Pass

		and be shown on the main menu in the status toggle.	displayed in the status button	

Review

The gui has now been tidied up a little so that redundant sections have been removed and two new features have been accounted for and implemented into the gui. In later stages I will be making use of the new demo mode and the led speed to change how the greenhouse is functioning. It would have been nice to implement the features which i have had to remove in this section however without doing so I believe the project would have stretched on and possibly doubled in size. I hope that this iterative stage has demonstrated how I would have implemented these features and that it was time which led me to remove them from the project.

Iterative stage 15 – Turning off all devices

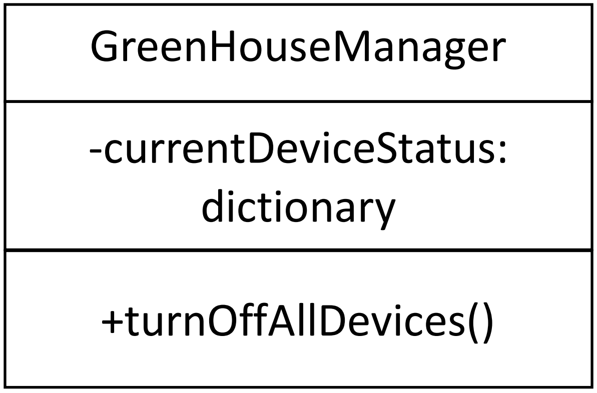
Overview

There are two scenarios where I am going to want to turn off all devices. When the greenhouse system is started there needs to be code to ensure that all devices are off/closed to account for situations where the greenhouse has say crashed and then been restarted leaving certain devices such as the fan still in operation. Since the current device statuses are set as off during initialization of the greenhouse manager class and there is no way to query the state of a device after a crash, I need to ensure the actual state and the recorded device state match to avoid and unexpected behavior from the greenhouse. The other situation is when the user sets the greenhouse status to off and so any active devices should be turned off.

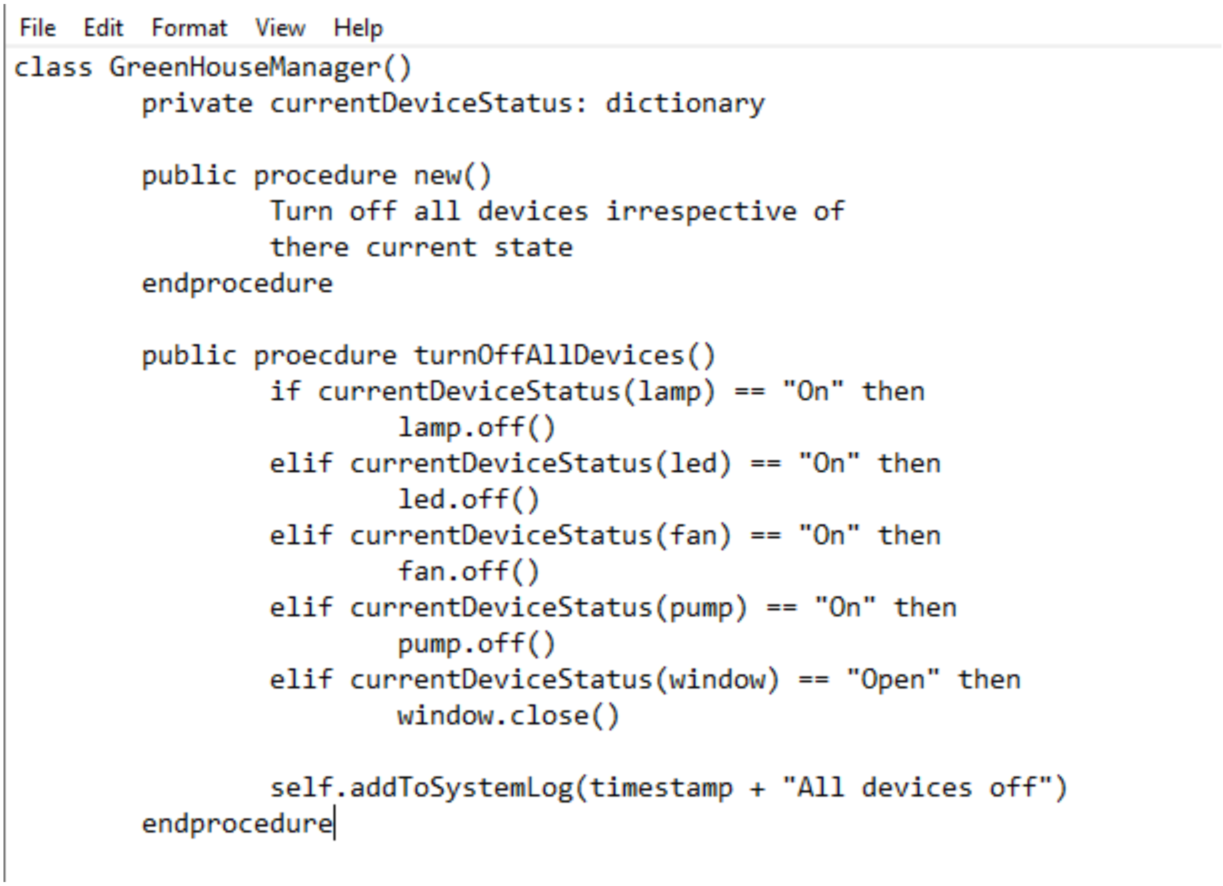
Requirements

Code will be written as the first code executed inside the class constrictor which will turn off all devices regardless of the state that is recorded for them inside the current device status dictionary. This can potentially cause errors if for example an attempt is made to turn off a led thread, but none exists. On the other hand, devices such as the relay do not care if you turn if off and its already off. So adequate error handling needs to be implemented on a case-by-case basis. Another function is going to carry out the same actions, but this time will be dictated by the current device status and so if a device is recorded as being already off no attempt will be made to turn it off. This method will only be used after the class has been initialized and thanks to the code about to be implemented, we can be confident that all devices will match their recorded status.

Class diagram

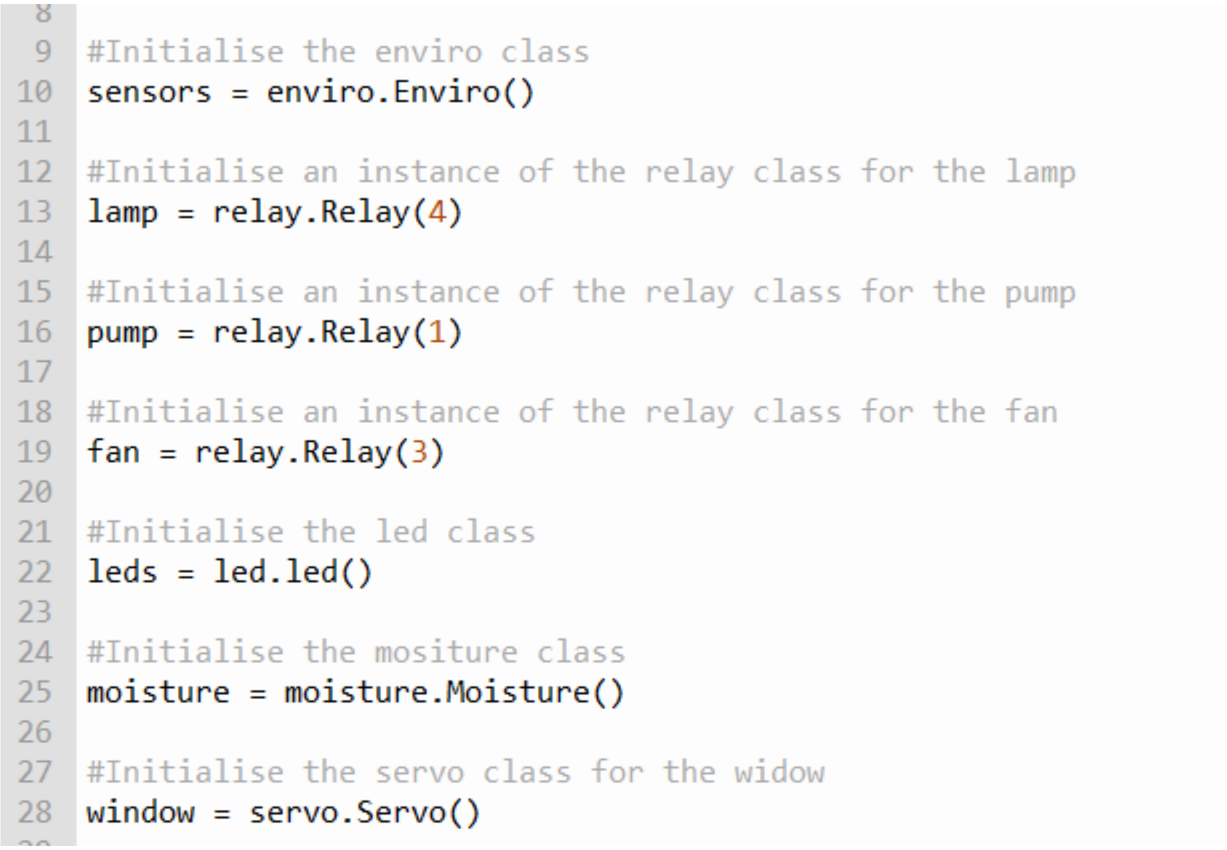


Pesudocode



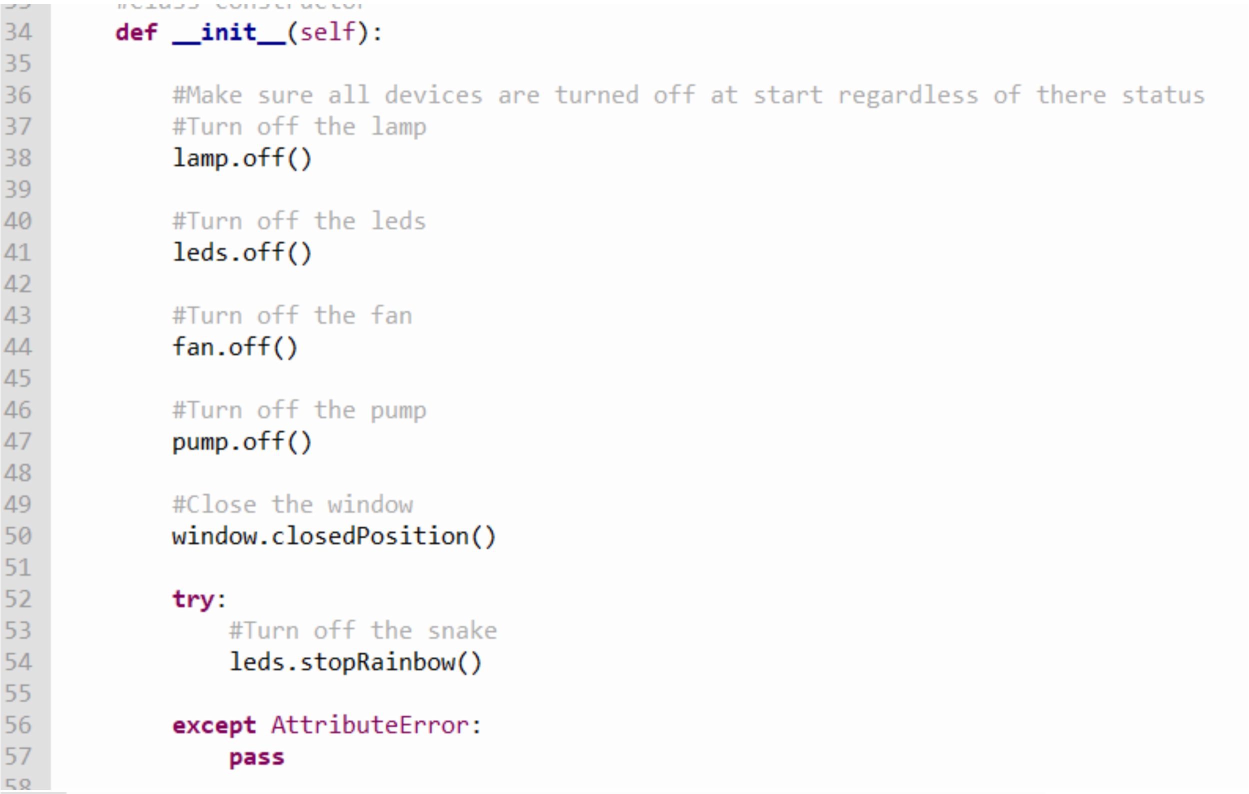
```
class GreenHouseManager()
        private currentDeviceStatus: dictionary

        public procedure new()
                Turn off all devices irrespective of
                there current state
        endprocedure

        public proecdure turnOffAllDevices()
                if currentDeviceStatus(lamp) == "On" then
                        lamp.off()
                elif currentDeviceStatus(led) == "On" then
                        led.off()
                elif currentDeviceStatus(fan) == "On" then
                        fan.off()
                elif currentDeviceStatus(pump) == "On" then
                        pump.off()
                elif currentDeviceStatus(window) == "Open" then
                        window.close()

                self.addToSystemLog(timestamp + "All devices off")
        endprocedure
```

Development log

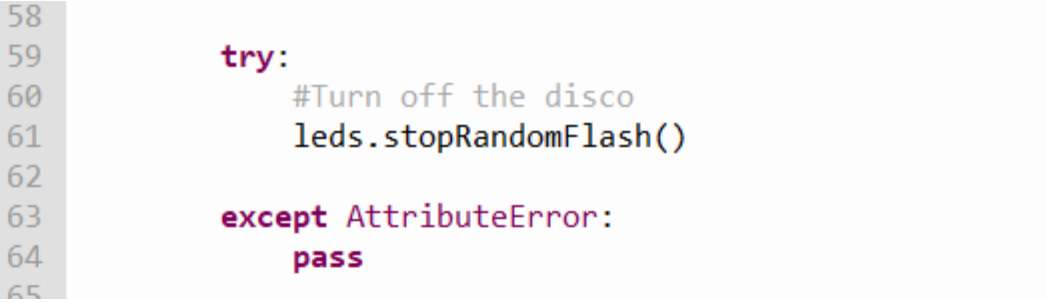


```
#Initialise the enviro class
sensors = enviro.Enviro()

#Initialise an instance of the relay class for the lamp
lamp = relay.Relay(4)

#Initialise an instance of the relay class for the pump
pump = relay.Relay(1)

#Initialise an instance of the relay class for the fan
fan = relay.Relay(3)

#Initialise the led class
leds = led.led()

#Initialise the mositure class
moisture = moisture.Moisture()

#Initialise the servo class for the widow
window = servo.Servo()
```

The greenhouse manager is going to need to be able to directly control all devices inside the greenhouse using the classes I have developed and to get readings from the greenhouse using the greenhouse. Due to this I have initialized objects for all the different sensors and devices. So that the class can control them. I believe this shows the justification for the changes I made earlier to the relay class so that I was able to pass the relay number once instead of each time I ran on or off. Instead, the relay is passed upon initialization of the class and then from then on, I can just call that object and then the on or off method without worrying about trying to remember which bus it's on.

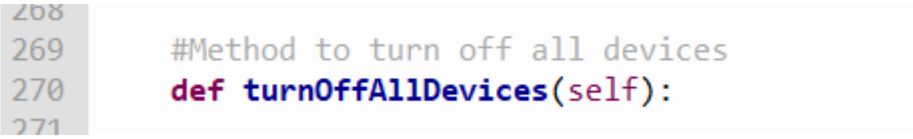


```
33      #class constructor
34      def __init__(self):
35
36          #Make sure all devices are turned off at start regardless of there status
37          #Turn off the lamp
38          lamp.off()
39
40          #Turn off the leds
41          leds.off()
42
43          #Turn off the fan
44          fan.off()
45
46          #Turn off the pump
47          pump.off()
48
49          #Close the window
50          window.closedPosition()
51
52          try:
53              #Turn off the snake
54              leds.stopRainbow()
55
56          except AttributeError:
57              pass
58
```



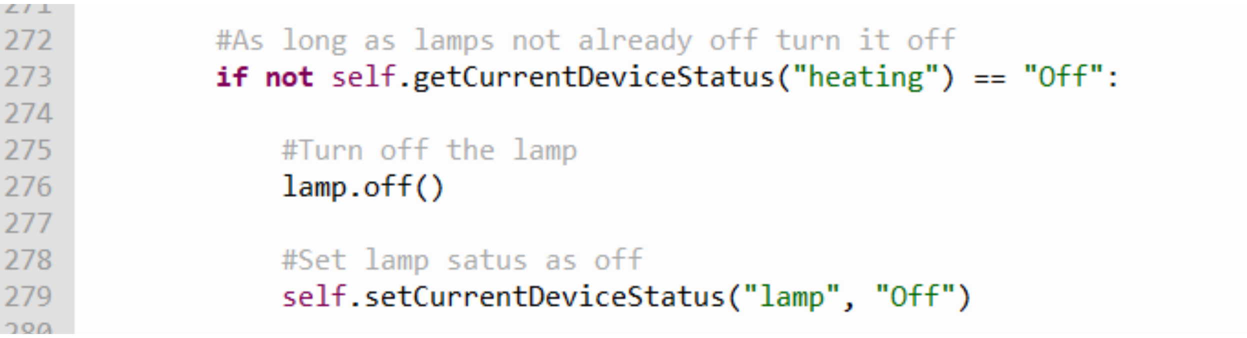
```
58
59          try:
60              #Turn off the disco
61              leds.stopRandomFlash()
62
63          except AttributeError:
64              pass
65
```

Inside the class constructor of the greenhouse manager, I have added the code to turn off all the different devices. To understand which devices would need error handling I made sure all devices were off inside the greenhouse and then ran the code without any error handling implemented. The only error produced was an Attribute error when trying to end and running led threads. To account for this I have added a try except to line 54 and 61 where I am attempting to close any currently running threads. Two try except statements are used despite the error being the same for both lines as if line 54 is ran and no threads are running an error would occur and the except part ran instead of an attempt being made to close any running random flash threads if they were nested inside the same statement. Now that this code is implemented the moment the greenhouse is ran all devices are ensured to be in there off state and no possible collisions or unexpected behaviors can occur whereby the greenhouse thinks a device is off but it's on. Which could be fatal for the plant if it's cooked by the lamp of flooded by the pump.



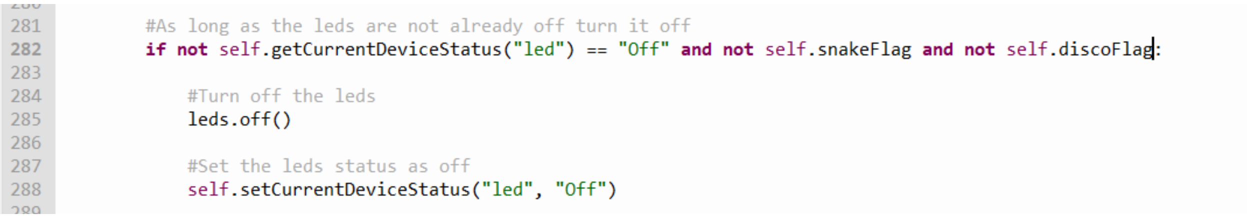
```
268
269     #Method to turn off all devices
270     def turnOffAllDevices(self):
271
```

The turn off all devices method is a much stricter method which will achieve the same results as the code written into the class constructor. This method will check the status of each device and if it is not already off/closed then it will try to turn off the device. Since we can be confident the stored current device status is the same as the actual device status no error handling is needed as there should never be a situation where a device is incorrectly turned off and an error created.



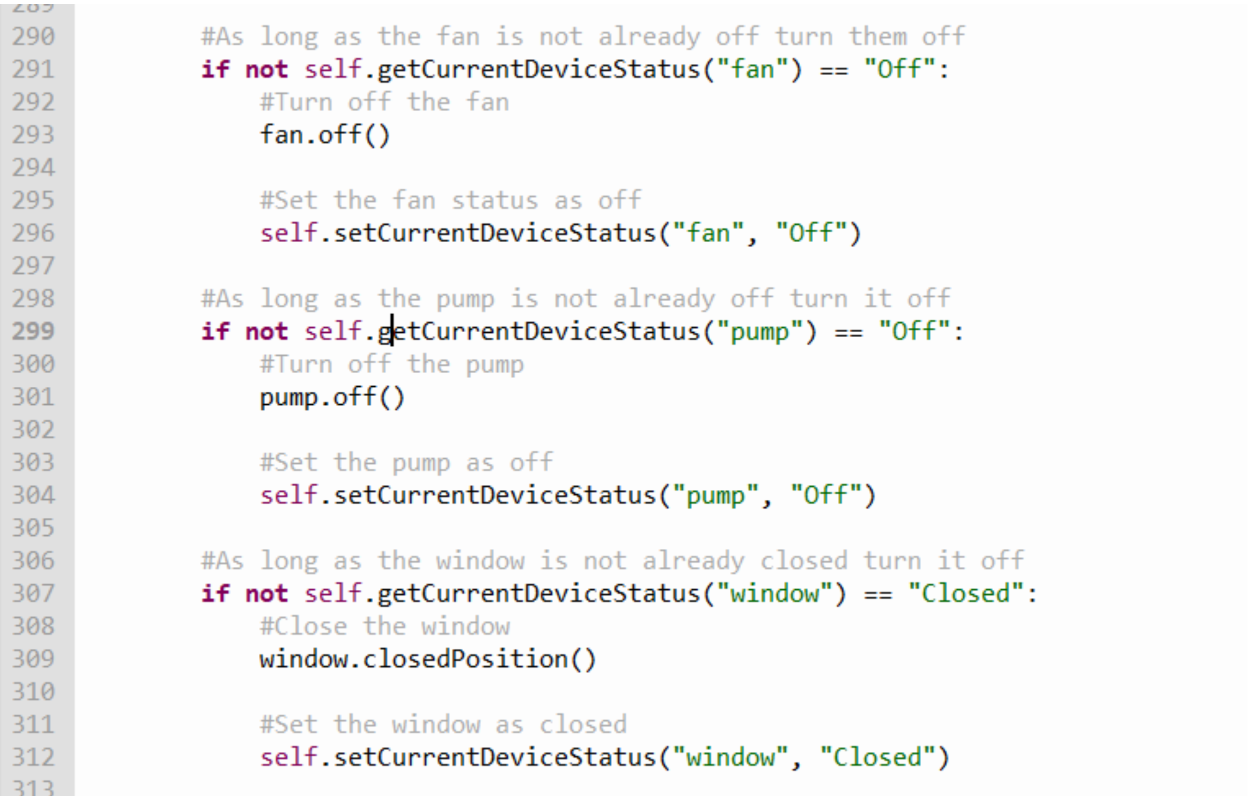
```
        #As long as lamps not already off turn it off
        if not self.getCurrentDeviceStatus("heating") == "Off":

            #Turn off the lamp
            lamp.off()

            #Set lamp satus as off
            self.setCurrentDeviceStatus("lamp", "Off")
```

The first device this method deals with is the heating. I am a big fan of beautiful code readable code and so I have used the not keyword to create a very readable statement to see if the current device is not off. In the case the status is not off then the lamp is turned off and the status of the lamp is set to off.



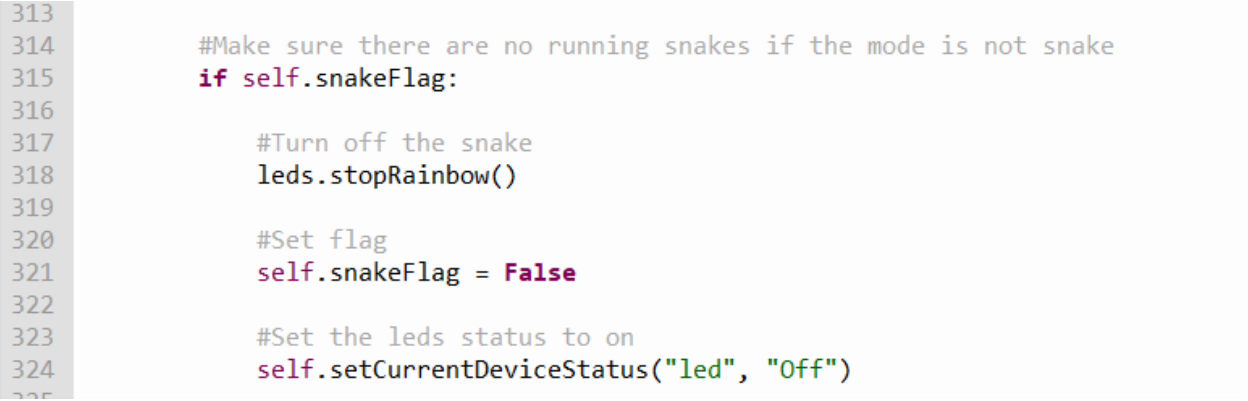
```
        #As long as the leds are not already off turn it off
        if not self.getCurrentDeviceStatus("led") == "Off" and not self.snakeFlag and not self.discoFlag:

            #Turn off the leds
            leds.off()

            #Set the leds status as off
            self.setCurrentDeviceStatus("led", "Off")
```

The same process is carried out for the leds. However, there are two extra checks to ensure that neither of two flags are true. Since I am using threading for the function of the led snake and disco extra care needs to be taken to ensure there is never an empty thread left running eating up processing power or even worse multiple threads at the same time. For these reasons I'm going to be using a flag to record If I have ever started a snake thread or a disco thread. This will allow me to periodically check if a thread is running and it should not be then turn it on. The flags are declared in the class constructor and have an initial value of false. So far, no code will make them true as I've not implemented anything which begins a led thread yet. The reason I'm not turning the leds off if a thread is running is that the led off method effectively sets the lights to a rgb value of 0 which won't stop the threads it will only momentarily turn off the led strip. I will deal with the threads in a moment. For the mean time if the leds are not off and no threads are running then they can be turned off using the leds off method and then their status is set to be equal to off.

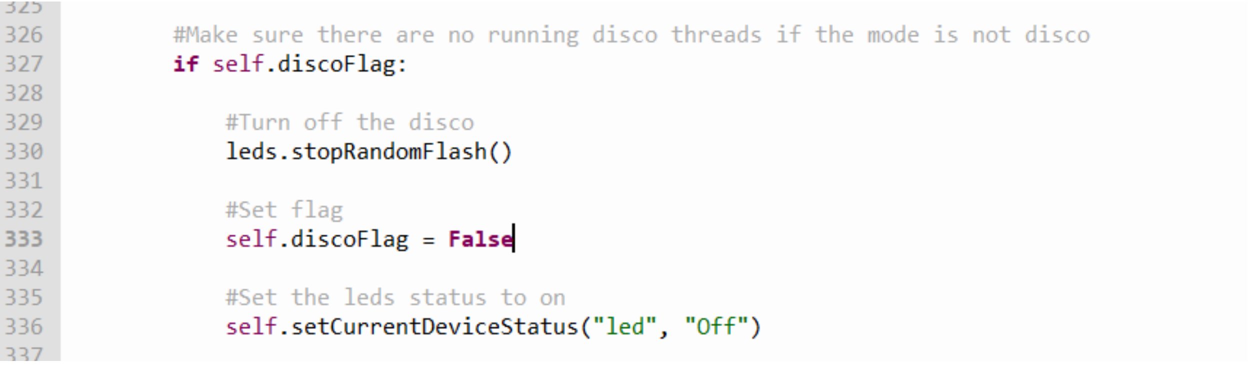


```
        #As long as the fan is not already off turn them off
        if not self.getCurrentDeviceStatus("fan") == "Off":
            #Turn off the fan
            fan.off()

            #Set the fan status as off
            self.setCurrentDeviceStatus("fan", "Off")

        #As long as the pump is not already off turn it off
        if not self.getCurrentDeviceStatus("pump") == "Off":
            #Turn off the pump
            pump.off()

            #Set the pump as off
            self.setCurrentDeviceStatus("pump", "Off")

        #As long as the window is not already closed turn it off
        if not self.getCurrentDeviceStatus("window") == "Closed":
            #Close the window
            window.closedPosition()

            #Set the window as closed
            self.setCurrentDeviceStatus("window", "Closed")
```

The fan, pump and window are all turned off in the same way as the lamp.

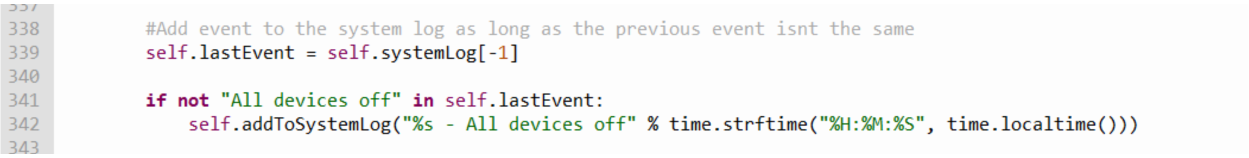


```
        #Make sure there are no running snakes if the mode is not snake
        if self.snakeFlag:

            #Turn off the snake
            leds.stopRainbow()

            #Set flag
            self.snakeFlag = False

            #Set the leds status to on
            self.setCurrentDeviceStatus("led", "Off")
```



```
        #Make sure there are no running disco threads if the mode is not disco
        if self.discoFlag:

            #Turn off the disco
            leds.stopRandomFlash()

            #Set flag
            self.discoFlag = False

            #Set the leds status to on
            self.setCurrentDeviceStatus("led", "Off")
```

The snake flag is only going to be true if I have set it so after starting a led snake thread. If the turn off all device's method is called then its time to turn off this thread. To do this I query the flag and if its true a thread is indeed running and needs turning off, so I then call the led stop rainbow method to stop the thread. Now the thread is running the flag is set to false and finally the status of the leds are set to be off. The process is the same for any running disco threads. This may seem like being over cautious however implementing this robustness removes any potential issues which could be a nightmare to solve if many hundreds of threads are running at once and is also just good practice.



```
        #Add event to the system log as long as the previous event isnt the same
        self.lastEvent = self.systemLog[-1]

        if not "All devices off" in self.lastEvent:
            self.addToSystemLog("%s - All devices off" % time.strftime("%H:%M:%S", time.localtime()))
```

The main function of this greenhouse responsible for making decisions regarding turning devices on and off will most likely be ran in a loop. I want to post an event to the system log when all devices are turned off informing the user that this has happened. If the state of the greenhouse is off, then on each iteration this turn off all device's method will be called to make sure all devices are off. This means that without any limitation each iteration would add a new all devices off event to the system log. This would spam the log and just look like a bit of a mess. For this reason, I've added a check to see if the last element in the system log array and hence the latest event to be added contains the words all devices off. If it does then no new event is added. A straight up comparison between the last event and the new event can't be made since the time stamps will be different meaning, they won't return true when being compared. Therefore, I've used the in keyword to see if the string is inside the last event in the array.

Test Plan

Test Number	Test Plan	Expected Outcome	Actual Outcome	Pass/Fail
1	Manually turn on all devices and then create an instance of the greenhouse manager class	All the devices should be turned off when the class is initialized	All the devices were turned off	Pass
2	After an object has been created turn on all the devices manually then call the turn off all device's method	All the devices should be turned off and their status set to off	All the devices were turned off and their status was set to off	Pass

Review

The implemented code inside the class constructor in this stage is going to be crucial to ensuring there are no scenarios where the greenhouse has a different recorded device status to the actual device in question. Whilst the turn off all device function will be used when the greenhouse mode is off to ensure there are no functioning devices.

Iterative stage 16 – Main Greenhouse manager function

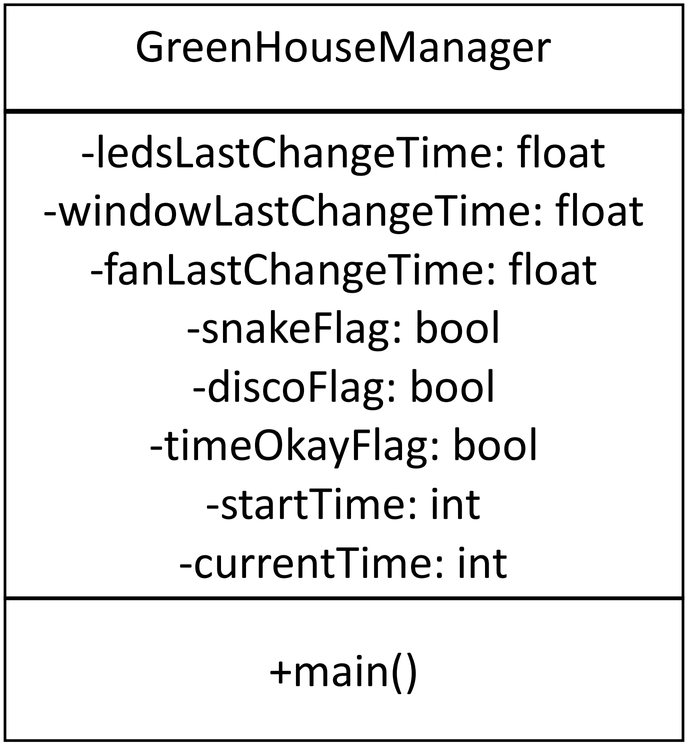
Overview

This stage is going to focus on building the main part of the greenhouse which is going to control all the different devices based on the different environment readings coming from inside the greenhouse. I am going to run this function on the clock at an interval of 2 seconds. Each time the function is called the current greenhouse status will be checked and depending on the mode certain actions will be taken. The function should add any changes made to the system log and make sure to set the status of devices when they are turned on or off.

Requirements

The main greenhouse manager function is going to track all running processes and take particular care to shut down any open threads related to the leds if they are not currently needed. When the greenhouse status is demo then a special script will be run to turn on all the devices so that the user can observe the greenhouse in action. Continuous mode is going to just carryout the management and monitoring of the greenhouse regardless of the time whilst when in scheduled mode the greenhouse will only be operating if the current time is inside the operating time specified by the user.

Class diagram



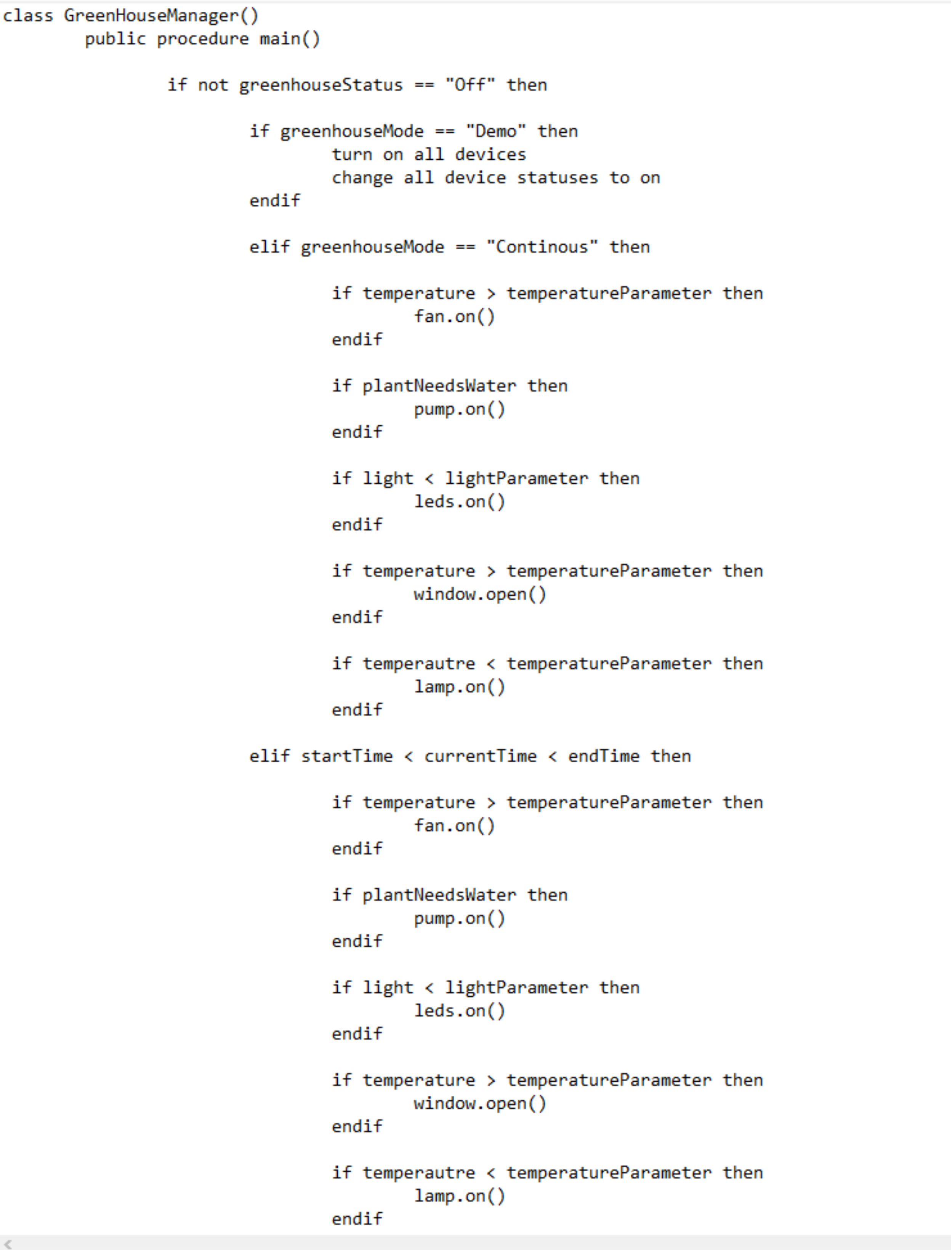
GreenHouseManager

-ledsLastChangeTime: float
-windowLastChangeTime: float
-fanLastChangeTime: float
-snakeFlag: bool
-discoFlag: bool
-timeOkayFlag: bool
-startTime: int
-currentTime: int

+main()

- ledsLastChangeTime is going to store the time at which the leds were last turned on or off
- windowLastChangeTime is going to store the time at which the window was last opened or closed
- fanLastChangeTime is to record the time at which the fan was last turned on or off
- snakeFlag is to record if a snake thread is active
- discoFlag is to record if a disco thread is active
- timeOkayFlag is to record if the current time is between the set interval of the user
- startTime is to hold the time which the user wants the greenhouse to function from
- endTime is to hold the time which the user wants the greenhouse to function until
- The main function is going to be the code to carry out the greenhouse functions

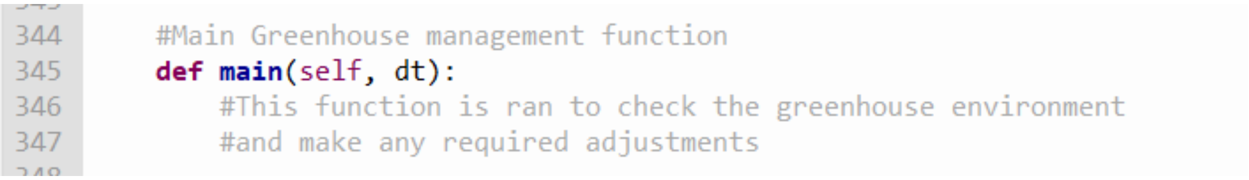
Pseudocode



```
class GreenHouseManager()
        public procedure main()

                if not greenhouseStatus == "Off" then

                        if greenhouseMode == "Demo" then
                                turn on all devices
                                change all device statuses to on
                        endif

                        elif greenhouseMode == "Continous" then

                                if temperature > temperatureParameter then
                                        fan.on()
                                endif

                                if plantNeedsWater then
                                        pump.on()
                                endif

                                if light < lightParameter then
                                        leds.on()
                                endif

                                if temperature > temperatureParameter then
                                        window.open()
                                endif

                                if temperautre < temperatureParameter then
                                        lamp.on()
                                endif

                        elif startTime < currentTime < endTime then

                                if temperature > temperatureParameter then
                                        fan.on()
                                endif

                                if plantNeedsWater then
                                        pump.on()
                                endif

                                if light < lightParameter then
                                        leds.on()
                                endif

                                if temperature > temperatureParameter then
                                        window.open()
                                endif

                                if temperautre < temperatureParameter then
                                        lamp.on()
                                endif
```

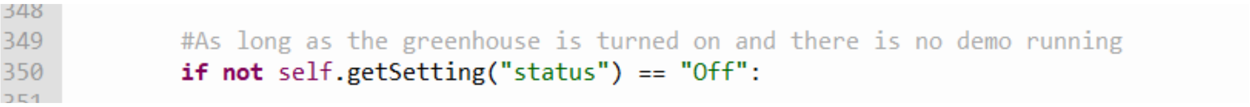
```
                    endif
                endif

            else then
                turnOffAllDevices()
```

Development log –



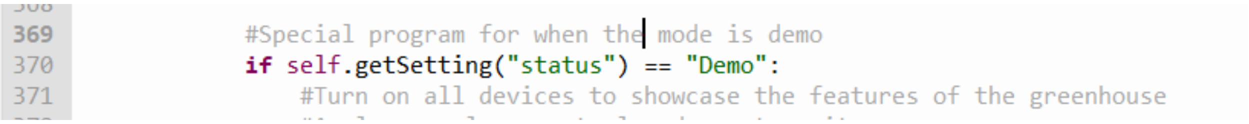
```
    #Main Greenhouse management function
    def main(self, dt):
        #This function is ran to check the greenhouse environment
        #and make any required adjustments
```

The main function is going to be added to the clock and needs to have the parameter dt so that the clock does not create an error.



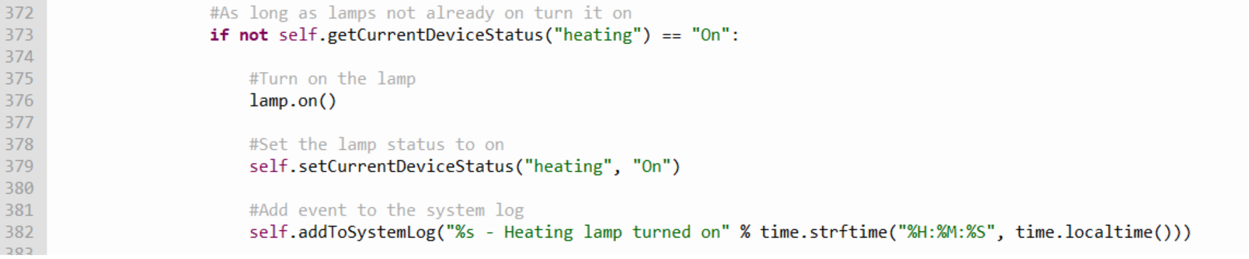
```
        #As long as the greenhouse is turned on and there is no demo running
        if not self.getSetting("status") == "Off":
```

When the function is running the first thing this is checked is if the greenhouse status is off. If the greenhouse is not off, then the statement evaluates to true, and the indented code block will be run.



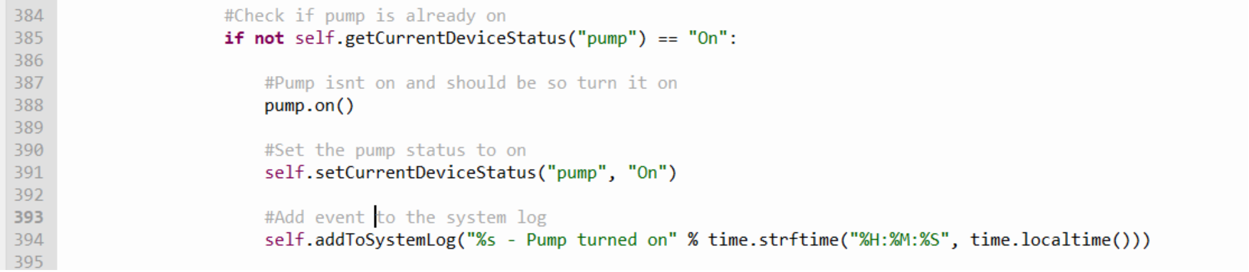
```
            #Special program for when the mode is demo
            if self.getSetting("status") == "Demo":
                #Turn on all devices to showcase the features of the greenhouse
```

When the greenhouse mode is demo, this means that all the devices need to be turned on.



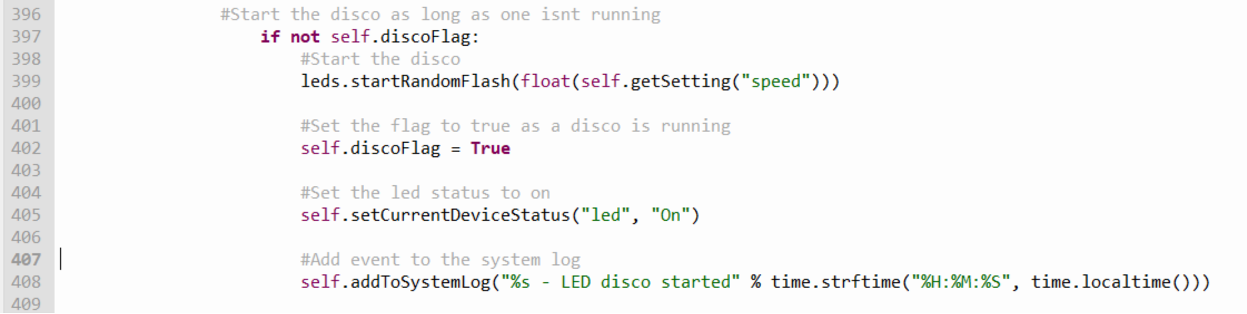
```
                #As long as lamps not already on turn it on
                if not self.getCurrentDeviceStatus("heating") == "On":

                    #Turn on the lamp
                    lamp.on()

                    #Set the lamp status to on
                    self.setCurrentDeviceStatus("heating", "On")

                    #Add event to the system log
                    self.addToSystemLog("%s - Heating lamp turned on" % time.strftime("%H:%M:%S", time.localtime()))
```

Firstly, the lamp is checked if the lamp is not already on then the lamp is turned on and its status set to be equal to on. An event is also added into the system log. The code for this is like the turn off all devices function just with the device being turned on and the check seeing if the device is not on as opposed to not off.

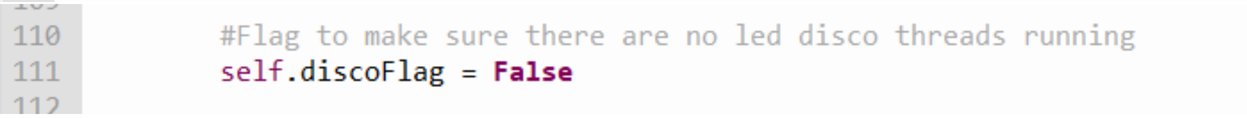


```
                #Check if pump is already on
                if not self.getCurrentDeviceStatus("pump") == "On":

                    #Pump isnt on and should be so turn it on
                    pump.on()

                    #Set the pump status to on
                    self.setCurrentDeviceStatus("pump", "On")

                    #Add event to the system log
                    self.addToSystemLog("%s - Pump turned on" % time.strftime("%H:%M:%S", time.localtime()))
```

The process is the same for the pump.

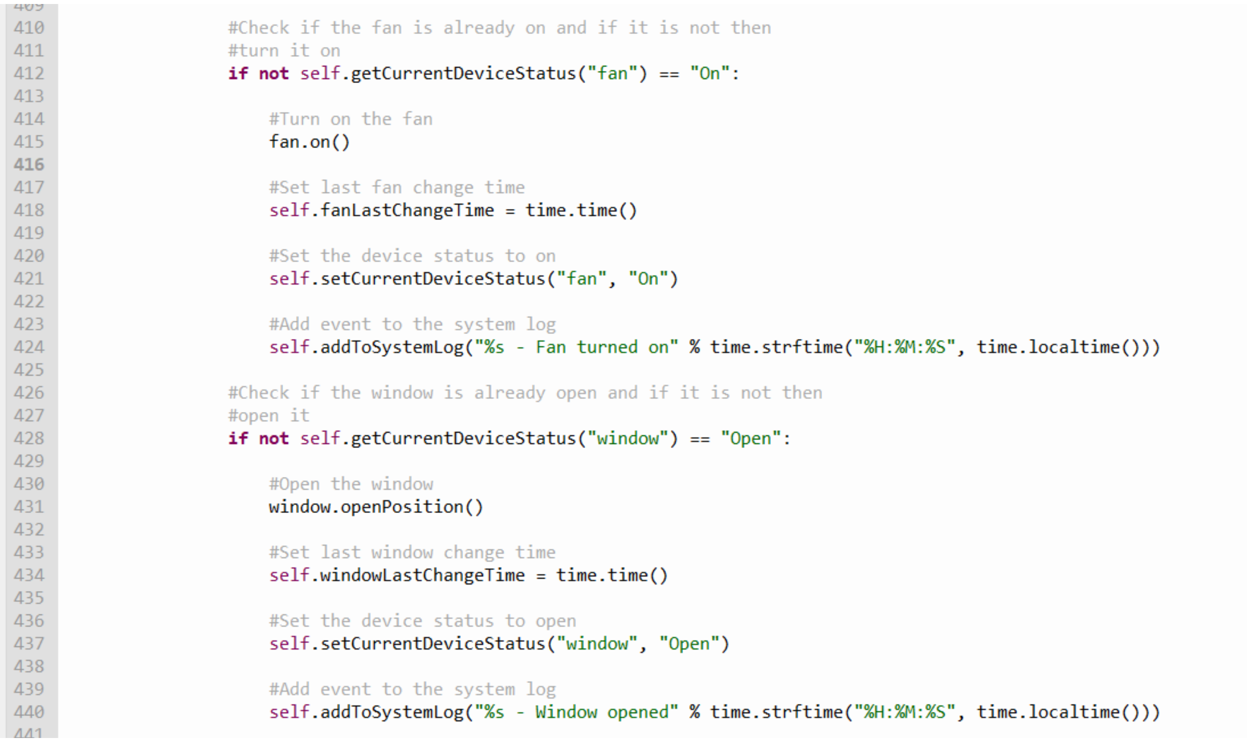


```
#Start the disco as long as one isnt running
    if not self.discoFlag:
        #Start the disco
        leds.startRandomFlash(float(self.getSetting("speed")))

        #Set the flag to true as a disco is running
        self.discoFlag = True

        #Set the led status to on
        self.setCurrentDeviceStatus("led", "On")

        #Add event to the system log
        self.addToSystemLog("%s - LED disco started" % time.strftime("%H:%M:%S", time.localtime()))
```



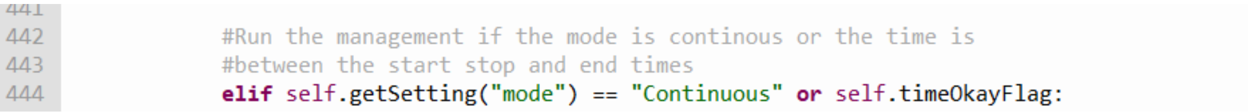
```
#Flag to make sure there are no led disco threads running
self.discoFlag = False
```

Since this mode is a demo designed to show off the features of the greenhouse, I am thought it would be good to turn on the random flash mode which I developed for the led class. As threads pose a potential issue if left unchecked, I am using the disco flag to only try and start a thread if there are no running threads. The flag is defined inside the class constructor and is false by default as no threads will be running when the greenhouse is started. Without this each clock cycle a new random flash thread would be created causing major issues related to memory and the led strip would be functioning erratically. The function start random flash is called to being a new random flash and is passes the users saved speed value giving them control over how fast or slow the flashes happen. As the speed is stored as a text value from the text input button, I have casted the value to a float. I went for a float over an integer so the user could tune the speed more precisely. Once the thread is started the disco flag is set to be equal to true. The device status of the leds is also changed to on. Finally, an event is added to the system log.



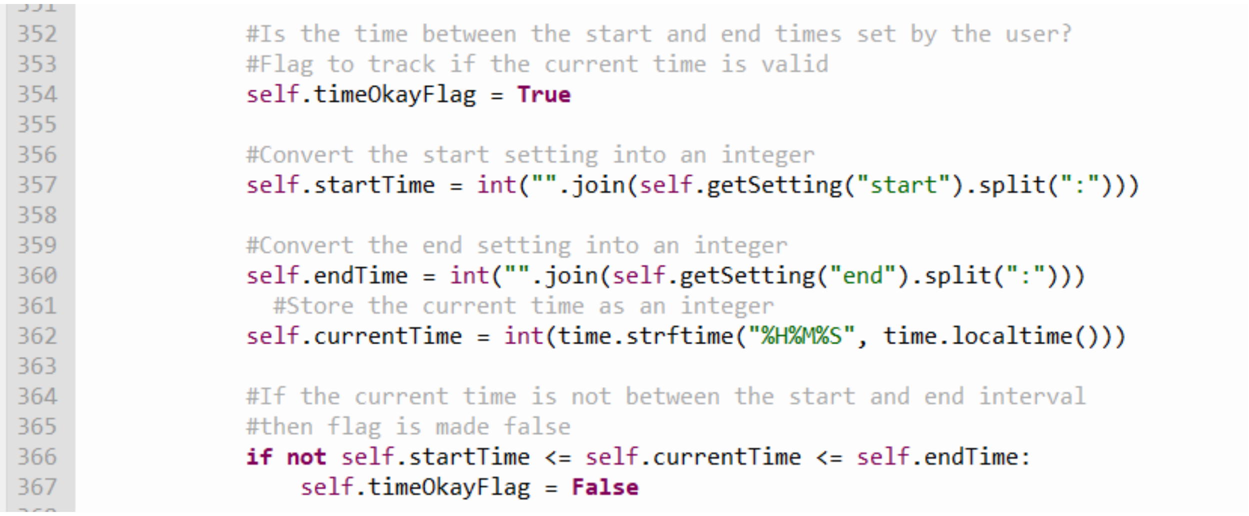
```
#Check if the fan is already on and if it is not then
#turn it on
if not self.getCurrentDeviceStatus("fan") == "On":

    #Turn on the fan
    fan.on()

    #Set last fan change time
    self.fanLastChangeTime = time.time()

    #Set the device status to on
    self.setCurrentDeviceStatus("fan", "On")

    #Add event to the system log
    self.addToSystemLog("%s - Fan turned on" % time.strftime("%H:%M:%S", time.localtime()))

#Check if the window is already open and if it is not then
#open it
if not self.getCurrentDeviceStatus("window") == "Open":

    #Open the window
    window.openPosition()

    #Set last window change time
    self.windowLastChangeTime = time.time()

    #Set the device status to open
    self.setCurrentDeviceStatus("window", "Open")

    #Add event to the system log
    self.addToSystemLog("%s - Window opened" % time.strftime("%H:%M:%S", time.localtime()))
```

The last two devices the fan and the window are both turned on or opened in the case of the window.



```
            #Run the management if the mode is continous or the time is
            #between the start stop and end times
            elif self.getSetting("mode") == "Continuous" or self.timeOkayFlag:
```

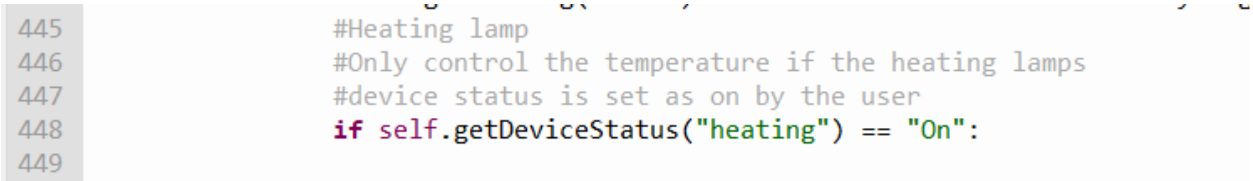
When the greenhouse is in the continuous mode, I want the greenhouse conditions to be monitored. The greenhouse conditions also need to be monitored when the mode is scheduled, and the time is between the start and end time that the user has set. Since the code for continuous and scheduled mode is the same the only difference being when it is run, they can be combined into the same statement. Providing the mode is continuous or the time is between the start and end time the main code should be ran. I am using the time okay flag to signify if the current time is within the set range by the user that the greenhouse should function in. No check is needed to see if the greenhouse is actual in scheduled mode as all other modes are covered previously and in the case the mode is scheduled then all that matters is that the time is okay.



```
            #Is the time between the start and end times set by the user?
            #Flag to track if the current time is valid
            self.timeOkayFlag = True

            #Convert the start setting into an integer
            self.startTime = int("".join(self.getSetting("start").split(":")))

            #Convert the end setting into an integer
            self.endTime = int("".join(self.getSetting("end").split(":")))
              #Store the current time as an integer
            self.currentTime = int(time.strftime("%H%M%S", time.localtime()))

            #If the current time is not between the start and end interval
            #then flag is made false
            if not self.startTime <= self.currentTime <= self.endTime:
                self.timeOkayFlag = False
```

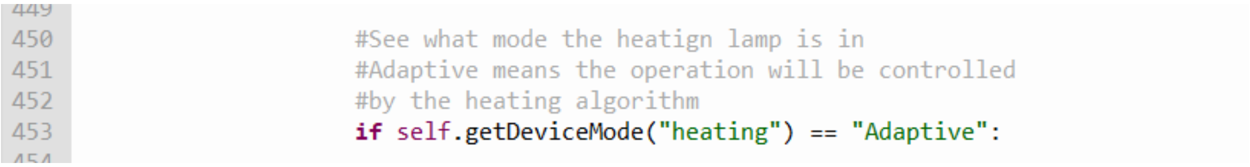
At the beginning of the main function the time okay flag is set. Its initial value is true as the time is thought to be within the allowed boundaries unless told otherwise. I decided the best way to compare time was to simply convert it into a number and then compare values. For example, when seeing if 09:00:00 Is larger than 10:00:00 both can be converted into numbers with the colons removed as follows 090000 and 100000 and then comparing these two values shows that 090000 is not larger than 100000. To convert the start time into this format It is first got from settings using the get setting method. At this stage, the time is in string format as follows hhmm:ss. The time is then split into 3 parts using the split method in python to sperate the values about the ":". The three values are stored in an array so can then be joined together using the join method without any character separating the joined values. The time is now in the following format as a string hhmmss. Finally, the time is converted into an integer to allow me to compare it to other time values. This process is carried out for both the start and end time. The current time is also got in the format hhmmss and then converted to an integer so a comparison can be made to see if the current time is between the start and end boundaries. An inequality looks to see if the current time is indeed within the start and end boundaries and if it is not then the flag is false so that the greenhouse wont function during the current cycle.

Heading



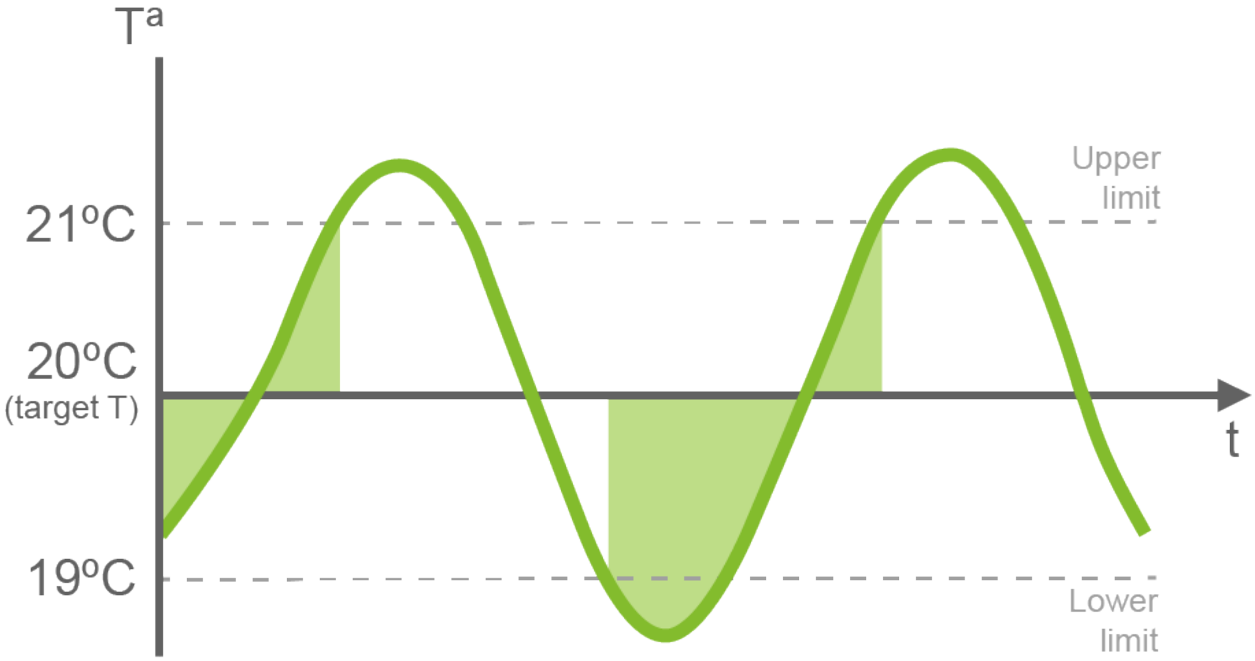
```
            #Heating lamp
            #Only control the temperature if the heating lamps
            #device status is set as on by the user
            if self.getDeviceStatus("heating") == "On":
```

The first device inside the greenhouse which I will be automating is the heating lamp. When the heat is too low the heating lamp needs to come on until the temperature reaches the set level by the user. If the user has selected the heating lamp status as on, then heating lamp algorithm will be carried out.

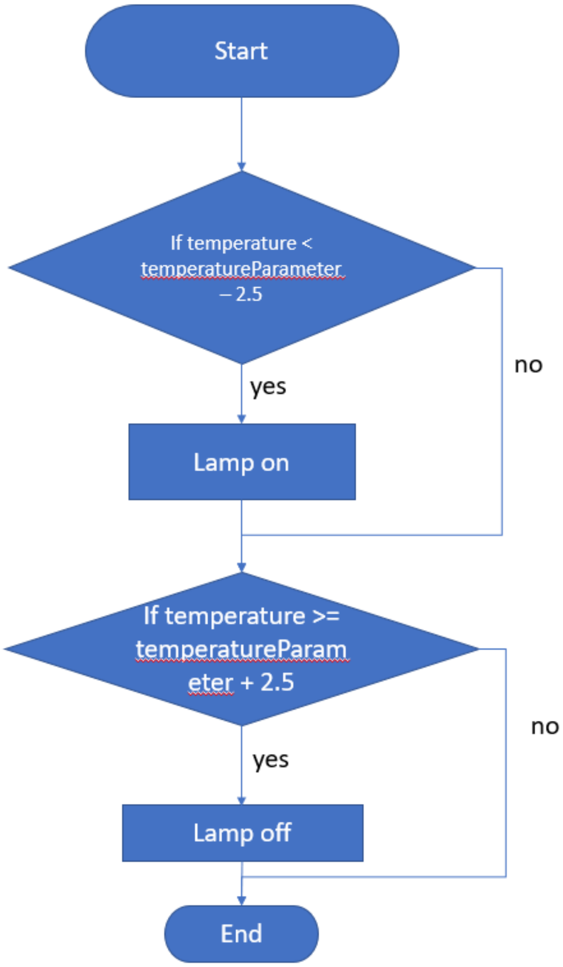


```
                #See what mode the heatign lamp is in
                #Adaptive means the operation will be controlled
                #by the heating algorithm
                if self.getDeviceMode("heating") == "Adaptive":
```

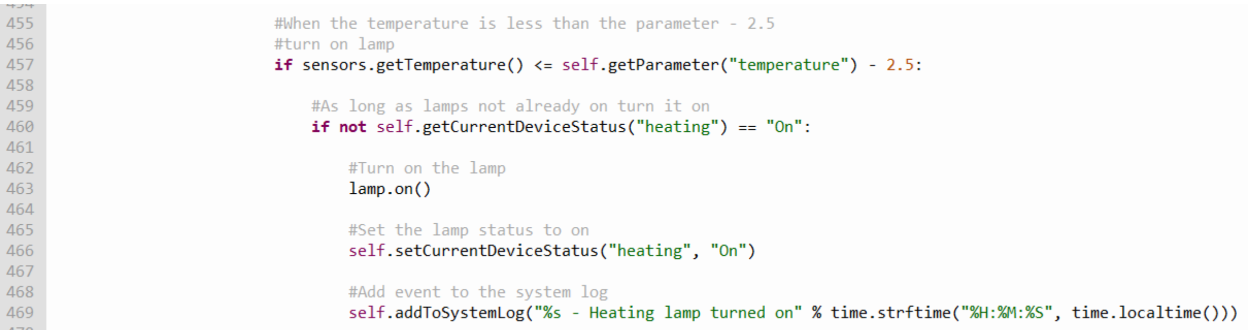
The first mode the heating lamp can be in is adaptive this means that each cycle the temperature will be checked to see if the lamp needs turning on. This is as opposed to the heating lamp being in manual mode which I will implement later.



When I initially implemented the code to turn the heating lamp on and off, I was checking to see if the temperature was less than the temperature parameter and if that was the case then the lamp was turned on and if the temperature was too high then the lamp was turned off. This caused an issue when the temperature reached the parameter value the lamp would be turned off however also instantly the temperature would then drop causing the lamp to turn back on and the temperature to rise above the parameter value and the whole process to repeat. This made a continuous loop where the lamp was being turned on and off continually when the temperature was near to the parameter value. After some research into how thermostats operate, I decided to implement an algorithm where the greenhouse would be heated a little above the parameter value and then allowed to fall a little below the parameter value before the lamp was turned on again. As seen in the diagram above this there will be an upper and lower limit based on the target temperature. The effect of this is that the average temperature will be the desired parameter value without the issue of the light flickering on and off.

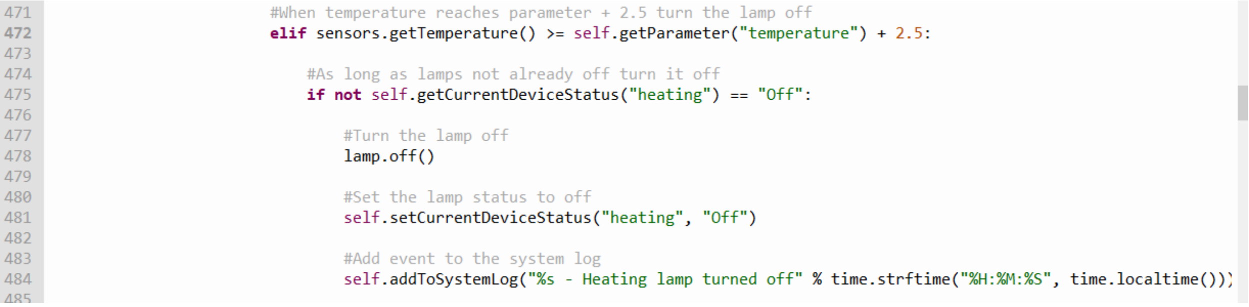

Start
If temperature < temperatureParameter – 2.5
yes
no
Lamp on
If temperature >= temperatureParameter + 2.5
yes
no
Lamp off
End

Above is the flow chart for the heating algorithm that I am going to be using to monitor the temperature. When the temperature is 2.5 degrees less than the desired temperature value the heating lamp is turned on. Otherwise, if the temperature is greater than or equal to the temperature parameter value + 2.5 then the lamp is turned off. I have recorded that it takes the greenhouse roughly 3 minuets for the temperature to fall by 2.5 degrees so with this algorithm implemented it should take 6 minuets from the maximum temperature being reached and the lamp being turned off to the temperature reaching the lower bound of the temperature and the lamp coming back on.



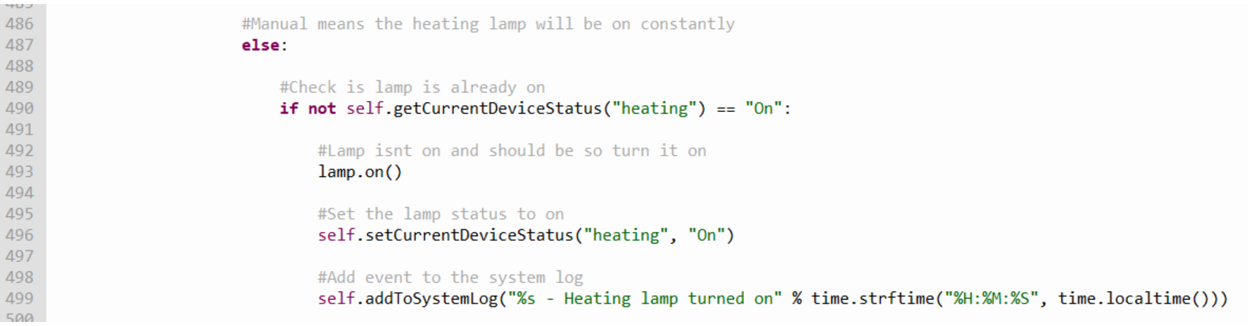
```
                        #When the temperature is less than the parameter - 2.5
                        #turn on lamp
                        if sensors.getTemperature() <= self.getParameter("temperature") - 2.5:

                            #As long as lamps not already on turn it on
                            if not self.getCurrentDeviceStatus("heating") == "On":

                                #Turn on the lamp
                                lamp.on()

                                #Set the lamp status to on
                                self.setCurrentDeviceStatus("heating", "On")

                                #Add event to the system log
                                self.addToSystemLog("%s - Heating lamp turned on" % time.strftime("%H:%M:%S", time.localtime()))
```

Here is the implementation of the temperature algorithm which I have created. When the temperature is less than or equal to the user desired temperature parameter – 2.5 degrees the lamp is turned on as long as it's not already on.



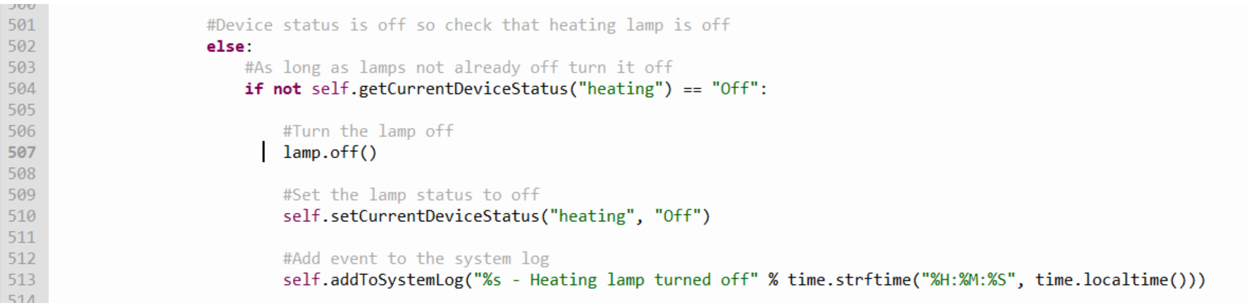
```
#When temperature reaches parameter + 2.5 turn the lamp off
elif sensors.getTemperature() >= self.getParameter("temperature") + 2.5:

    #As long as lamps not already off turn it off
    if not self.getCurrentDeviceStatus("heating") == "Off":

        #Turn the lamp off
        lamp.off()

        #Set the lamp status to off
        self.setCurrentDeviceStatus("heating", "Off")

        #Add event to the system log
        self.addToSystemLog("%s - Heating lamp turned off" % time.strftime("%H:%M:%S", time.localtime()))
```

When the temperature inside the greenhouse reaches the user set temperature parameter + 2.5 the lamp is turned off. This leaves a 5-degree window where the temperature will fall from the upper bound until it hits the lower bound and the process begins again.



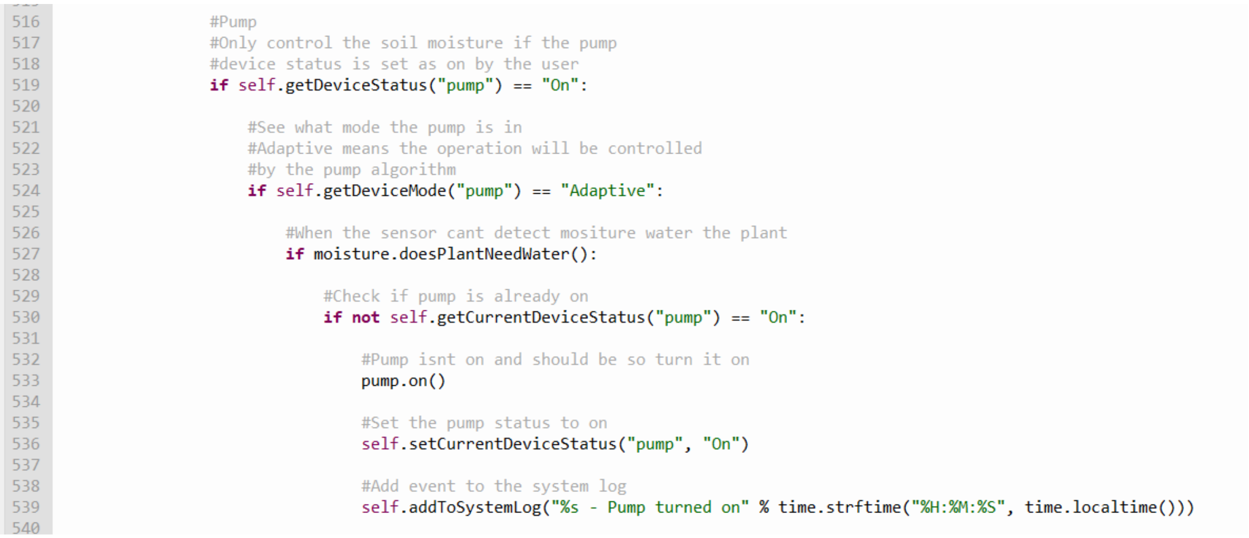
```
#Manual means the heating lamp will be on constantly
else:

    #Check is lamp is already on
    if not self.getCurrentDeviceStatus("heating") == "On":

        #Lamp isnt on and should be so turn it on
        lamp.on()

        #Set the lamp status to on
        self.setCurrentDeviceStatus("heating", "On")

        #Add event to the system log
        self.addToSystemLog("%s - Heating lamp turned on" % time.strftime("%H:%M:%S", time.localtime()))
```

When the mode is not adaptive it must be manual. In manual mode the lamp needs to be always on. So, each cycle there is a check to see that the lamp is on and if it is not then the lamp is turned on.



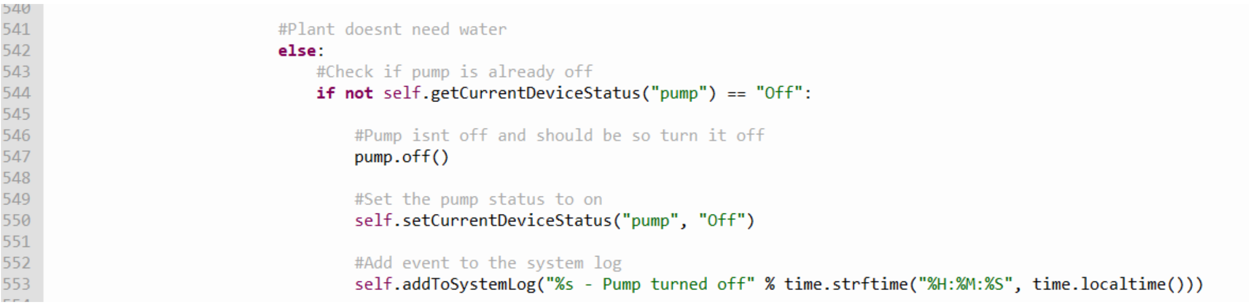
```
#Device status is off so check that heating lamp is off
else:
    #As long as lamps not already off turn it off
    if not self.getCurrentDeviceStatus("heating") == "Off":

        #Turn the lamp off
        lamp.off()

        #Set the lamp status to off
        self.setCurrentDeviceStatus("heating", "Off")

        #Add event to the system log
        self.addToSystemLog("%s - Heating lamp turned off" % time.strftime("%H:%M:%S", time.localtime()))
```

If the device status is not on, then it must be off. In this case the lamp is turned off so that it is not functioning as the user has set that the lamp should not be being used by the greenhouse.



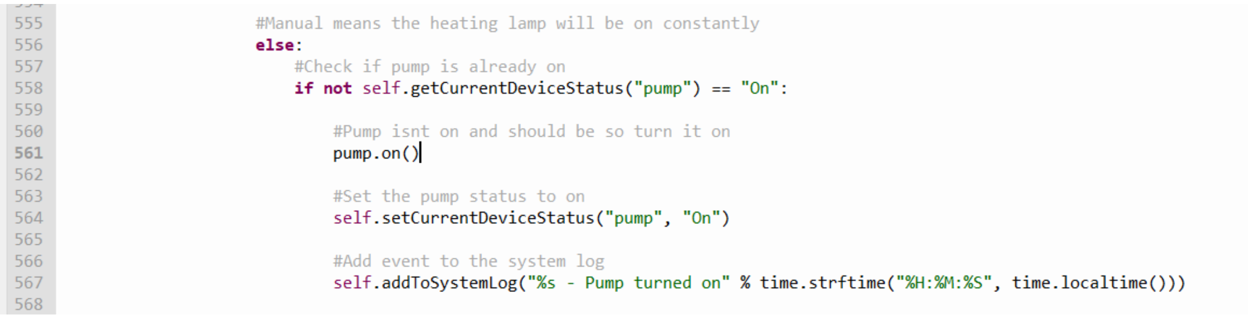
```
                #Pump
                #Only control the soil moisture if the pump
                #device status is set as on by the user
                if self.getDeviceStatus("pump") == "On":

                    #See what mode the pump is in
                    #Adaptive means the operation will be controlled
                    #by the pump algorithm
                    if self.getDeviceMode("pump") == "Adaptive":

                        #When the sensor cant detect mositure water the plant
                        if moisture.doesPlantNeedWater():

                            #Check if pump is already on
                            if not self.getCurrentDeviceStatus("pump") == "On":

                                #Pump isnt on and should be so turn it on
                                pump.on()

                                #Set the pump status to on
                                self.setCurrentDeviceStatus("pump", "On")

                                #Add event to the system log
                                self.addToSystemLog("%s - Pump turned on" % time.strftime("%H:%M:%S", time.localtime()))
```

The next device to be controlled is the pump. As soil absorbs water, I have found that once the soil moisture level set by the user on the soil moisture sensor is reached there is a suitable time gap before the sensor then reads as being too low on moisture. For this reason, the pump can simply be turned on when the plant needs water and then turned off when it does not without any special algorithm to stop the pump cycling between being on and off. This is good since the soil moisture is binary in the sense that it can only be detecting moisture or not detecting moisture so the previously implemented algorithm for the lamp could not work in this case. The moisture class is used to see if the plant needs any water and if this is the case the pump is turned on.



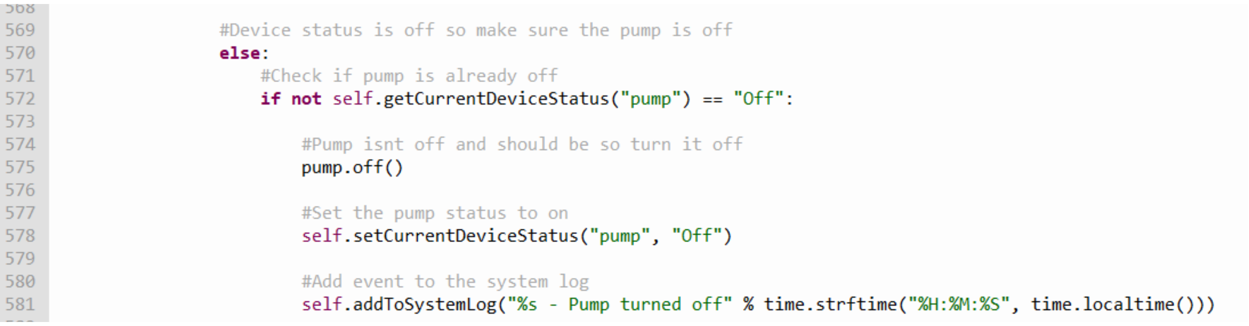
```
                        #Plant doesnt need water
                        else:
                            #Check if pump is already off
                            if not self.getCurrentDeviceStatus("pump") == "Off":

                                #Pump isnt off and should be so turn it off
                                pump.off()

                                #Set the pump status to on
                                self.setCurrentDeviceStatus("pump", "Off")

                                #Add event to the system log
                                self.addToSystemLog("%s - Pump turned off" % time.strftime("%H:%M:%S", time.localtime()))
```

When the moisture sensor is not detecting moisture, the plant does not need to be watered so the pump is turned off.



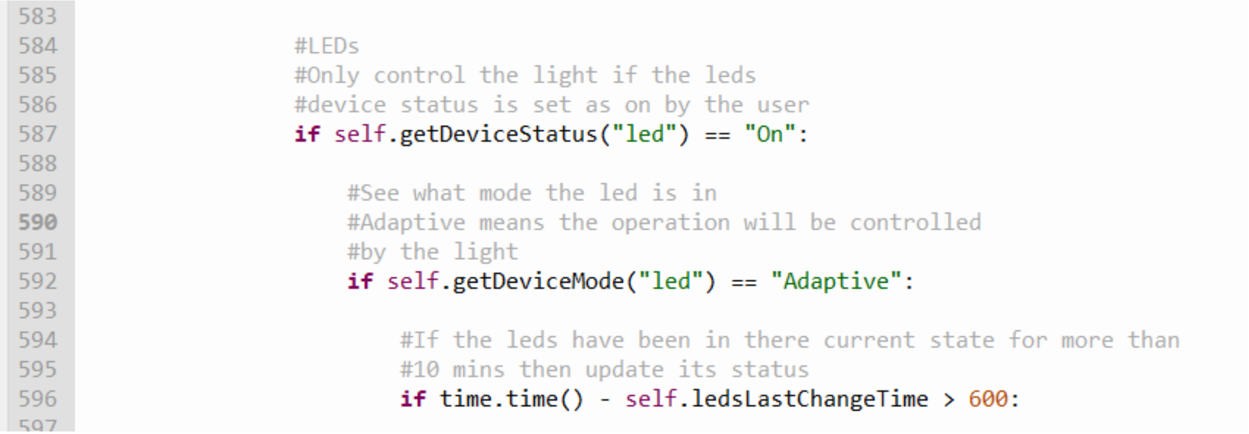
```
                    #Manual means the heating lamp will be on constantly
                    else:
                        #Check if pump is already on
                        if not self.getCurrentDeviceStatus("pump") == "On":

                            #Pump isnt on and should be so turn it on
                            pump.on()

                            #Set the pump status to on
                            self.setCurrentDeviceStatus("pump", "On")

                            #Add event to the system log
                            self.addToSystemLog("%s - Pump turned on" % time.strftime("%H:%M:%S", time.localtime()))
```

Just like for the lamp when the pump is in manual mode it is always on.

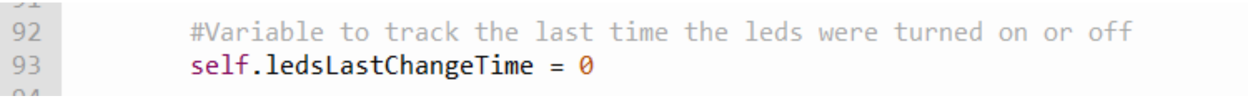


```
                #Device status is off so make sure the pump is off
                else:
                    #Check if pump is already off
                    if not self.getCurrentDeviceStatus("pump") == "Off":

                        #Pump isnt off and should be so turn it off
                        pump.off()

                        #Set the pump status to on
                        self.setCurrentDeviceStatus("pump", "Off")

                        #Add event to the system log
                        self.addToSystemLog("%s - Pump turned off" % time.strftime("%H:%M:%S", time.localtime()))
```

When the pump is disabled by the user the pump is turned off so that it is not functioning.

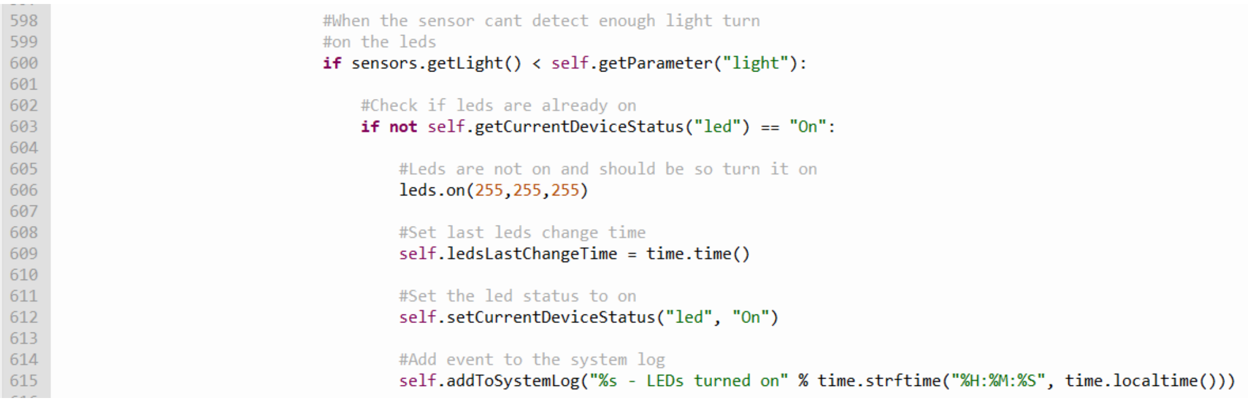


```

                #LEDs
                #Only control the light if the leds
                #device status is set as on by the user
                if self.getDeviceStatus("led") == "On":

                    #See what mode the led is in
                    #Adaptive means the operation will be controlled
                    #by the light
                    if self.getDeviceMode("led") == "Adaptive":

                        #If the leds have been in there current state for more than
                        #10 mins then update its status
                        if time.time() - self.ledsLastChangeTime > 600:
```



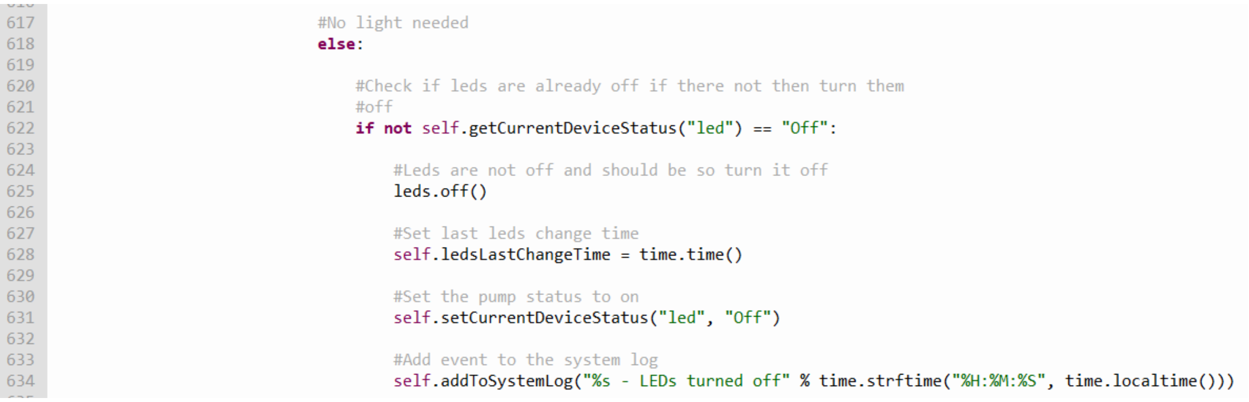
```
        #Variable to track the last time the leds were turned on or off
        self.ledsLastChangeTime = 0
```

Controlling the leds are a little different since unlike the lamp I cannot wait for the light value to slowly fall away before turning the leds on since the moment the leds are turned off the light will drop instantly. Without some sort of algorithm governing the leds they would just constantly turn on surpass the light parameter then turn off and drop below the parameter value instantly and then repeat the whole process again. For this reason, I'm going to make it so that the leds must have been in their current state for more than 10 minuets before there status can be changed. To do this I am using the leds last change time variable. This is defined inside the greenhouse manager class constructor and initially has a value of 0. When I need to check if the leds have been in there current state for more than 10 minutes I get the current time using time.time(). This just provides the time in seconds since an arbitrary moment in the 1970s called the epoch. Since I am only worried about change in time it does not matter that the actual time is not related to the current time what matters is the difference between two readings. I then deduct the leds last change time value from this value and if it is bigger than 600 seconds then the leds have been in their current state for more than 10 minuets and so the algorithm will change their state if needs be. As the led last change time is 0 to begin with the algorithm will change their state on first pass as the time – 0 is larger than 600. From then onwards the leds last change time will be updated when I change the led state and the value produced from deducting this value from time will be the number of seconds since the leds were last turned on or off.



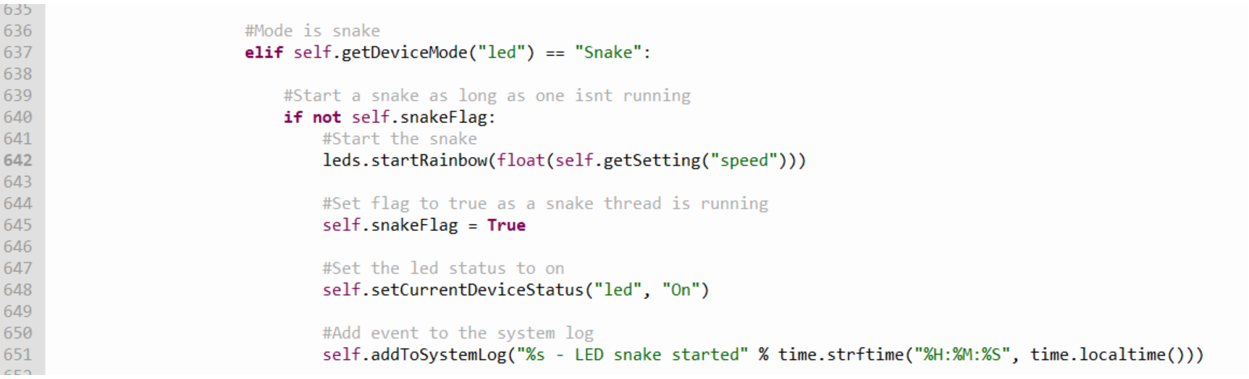
```
                            #When the sensor cant detect enough light turn
                            #on the leds
                            if sensors.getLight() < self.getParameter("light"):

                                #Check if leds are already on
                                if not self.getCurrentDeviceStatus("led") == "On":

                                    #Leds are not on and should be so turn it on
                                    leds.on(255,255,255)

                                    #Set last leds change time
                                    self.ledsLastChangeTime = time.time()

                                    #Set the led status to on
                                    self.setCurrentDeviceStatus("led", "On")

                                    #Add event to the system log
                                    self.addToSystemLog("%s - LEDs turned on" % time.strftime("%H:%M:%S", time.localtime()))
```

Providing the leds have been in their current state for more than 10 minutes then I check to see if the light reading from the greenhouse is less than the light parameter and if it is the greenhouse led strip is turned on. In this mode I am setting the whole strip to be white. When the leds are turned on I also set the leds last change time to be equal to the current time.time value.



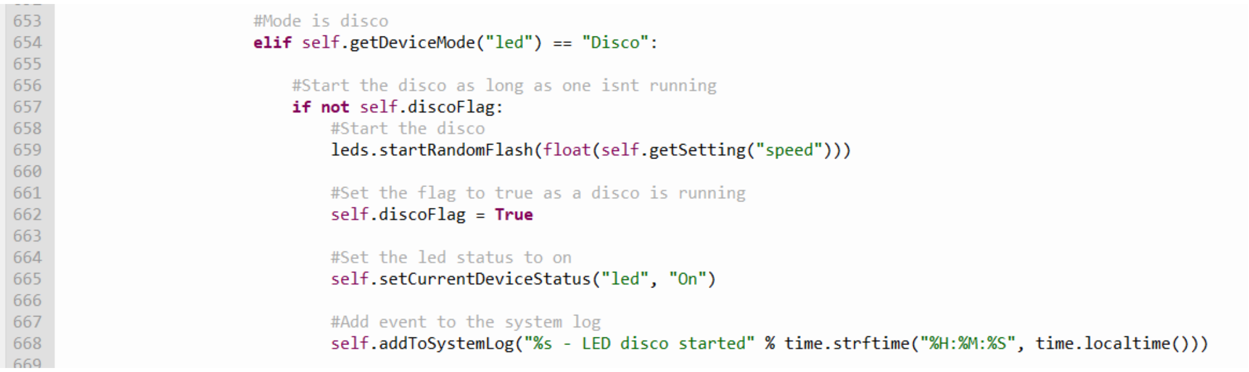
```
                            #No light needed
                            else:

                                #Check if leds are already off if there not then turn them
                                #off
                                if not self.getCurrentDeviceStatus("led") == "Off":

                                    #Leds are not off and should be so turn it off
                                    leds.off()

                                    #Set last leds change time
                                    self.ledsLastChangeTime = time.time()

                                    #Set the pump status to on
                                    self.setCurrentDeviceStatus("led", "Off")

                                    #Add event to the system log
                                    self.addToSystemLog("%s - LEDs turned off" % time.strftime("%H:%M:%S", time.localtime()))
```

If the light value is high enough then the leds need to be off. In this case I only set the leds last change time if the lights are on and I turn them off. If the last change time was set regardless of whether the leds being turned off from the on state, then effectively my algorithm would just be checking every 10 minutes if the light value is too high or low.



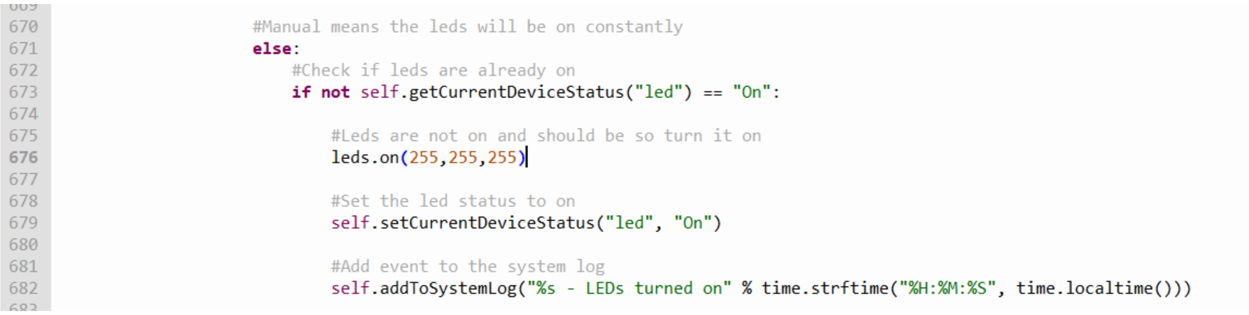
```
                    #Mode is snake
                    elif self.getDeviceMode("led") == "Snake":

                        #Start a snake as long as one isnt running
                        if not self.snakeFlag:
                            #Start the snake
                            leds.startRainbow(float(self.getSetting("speed")))

                            #Set flag to true as a snake thread is running
                            self.snakeFlag = True

                            #Set the led status to on
                            self.setCurrentDeviceStatus("led", "On")

                            #Add event to the system log
                            self.addToSystemLog("%s - LED snake started" % time.strftime("%H:%M:%S", time.localtime()))
```

If the user has selected that the leds should be working in snake mode, then providing no snake thread is currently running I am being a new led rainbow with a speed equal to the value entered by the user. The snake flag is made true when I begin a new thread to make sure that only one thread is in operation at once.



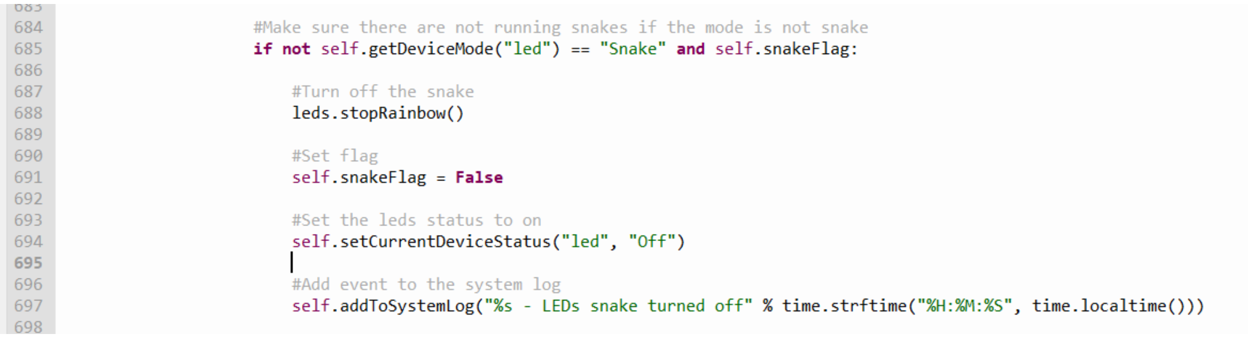
```
#Mode is disco
elif self.getDeviceMode("led") == "Disco":

    #Start the disco as long as one isnt running
    if not self.discoFlag:
        #Start the disco
        leds.startRandomFlash(float(self.getSetting("speed")))

        #Set the flag to true as a disco is running
        self.discoFlag = True

        #Set the led status to on
        self.setCurrentDeviceStatus("led", "On")

        #Add event to the system log
        self.addToSystemLog("%s - LED disco started" % time.strftime("%H:%M:%S", time.localtime()))
```

The same process is carried out for if the mode is disco where a random flash is started. This time the disco flag is made true.



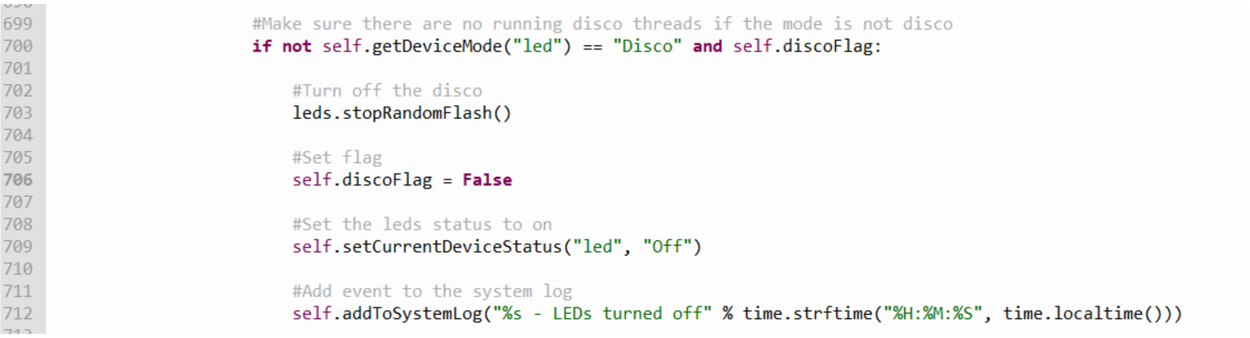
```
#Manual means the leds will be on constantly
else:
    #Check if leds are already on
    if not self.getCurrentDeviceStatus("led") == "On":

        #Leds are not on and should be so turn it on
        leds.on(255,255,255)

        #Set the led status to on
        self.setCurrentDeviceStatus("led", "On")

        #Add event to the system log
        self.addToSystemLog("%s - LEDs turned on" % time.strftime("%H:%M:%S", time.localtime()))
```

The final mode is manual which means the leds need to be on continually. In this case the led strip is set to be white.



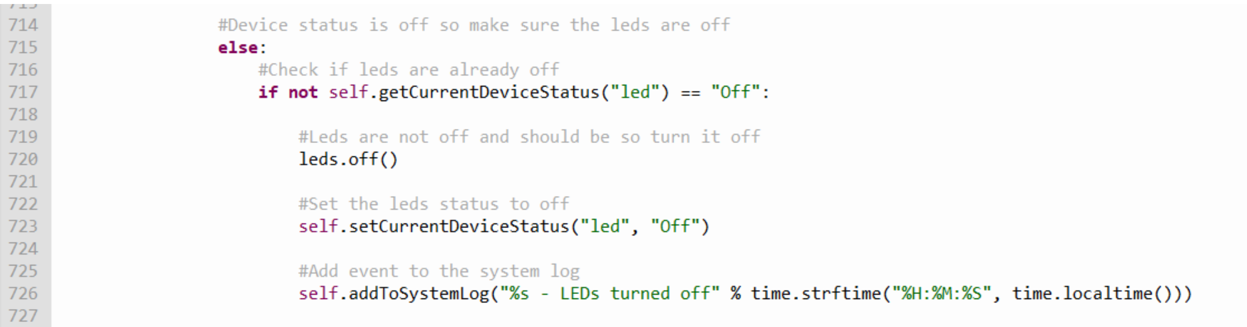
```
#Make sure there are not running snakes if the mode is not snake
if not self.getDeviceMode("led") == "Snake" and self.snakeFlag:

    #Turn off the snake
    leds.stopRainbow()

    #Set flag
    self.snakeFlag = False

    #Set the leds status to on
    self.setCurrentDeviceStatus("led", "Off")

    #Add event to the system log
    self.addToSystemLog("%s - LEDs snake turned off" % time.strftime("%H:%M:%S", time.localtime()))
```

At the end of the snake algorithm I am checking to see if the mode of the led strip is not snake mode but the snake flag is true indicating there is a runnign snake thread. When this is the case the snake thread should not be running and needs to be stopped. The rainbow is stopped and the flag made false as there is now no running thread.



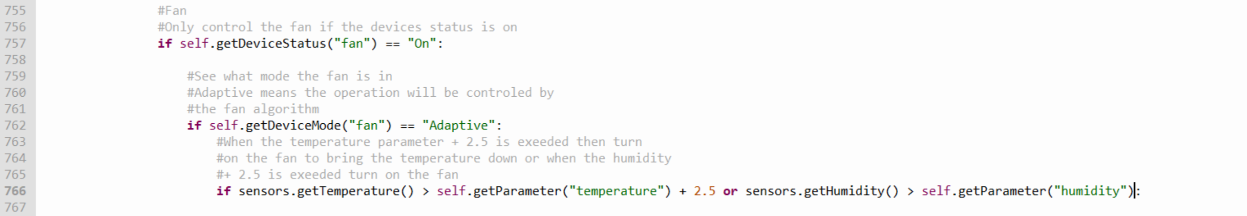
```
#Make sure there are no running disco threads if the mode is not disco
if not self.getDeviceMode("led") == "Disco" and self.discoFlag:

    #Turn off the disco
    leds.stopRandomFlash()

    #Set flag
    self.discoFlag = False

    #Set the leds status to on
    self.setCurrentDeviceStatus("led", "Off")

    #Add event to the system log
    self.addToSystemLog("%s - LEDs turned off" % time.strftime("%H:%M:%S", time.localtime()))
```

The same process is carried out for the random flash thread to make sure there are no loose disco threads which should not be running.



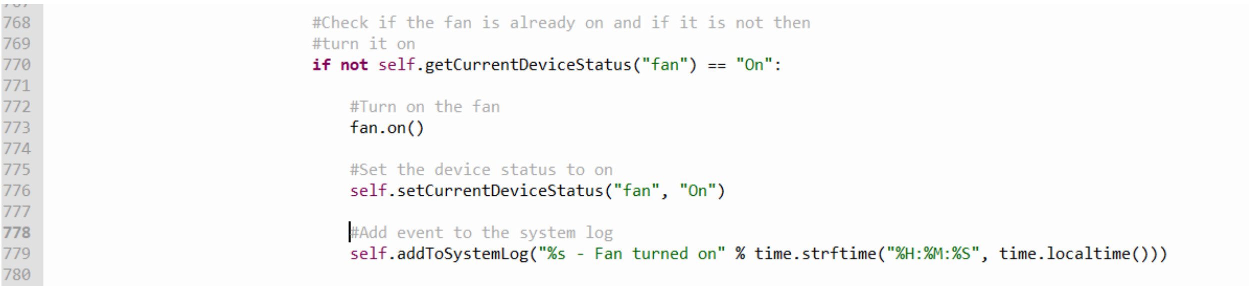
```
            #Device status is off so make sure the leds are off
            else:
                #Check if leds are already off
                if not self.getCurrentDeviceStatus("led") == "Off":

                    #Leds are not off and should be so turn it off
                    leds.off()

                    #Set the leds status to off
                    self.setCurrentDeviceStatus("led", "Off")

                    #Add event to the system log
                    self.addToSystemLog("%s - LEDs turned off" % time.strftime("%H:%M:%S", time.localtime()))

```

When the led device status is off the leds are turned off.



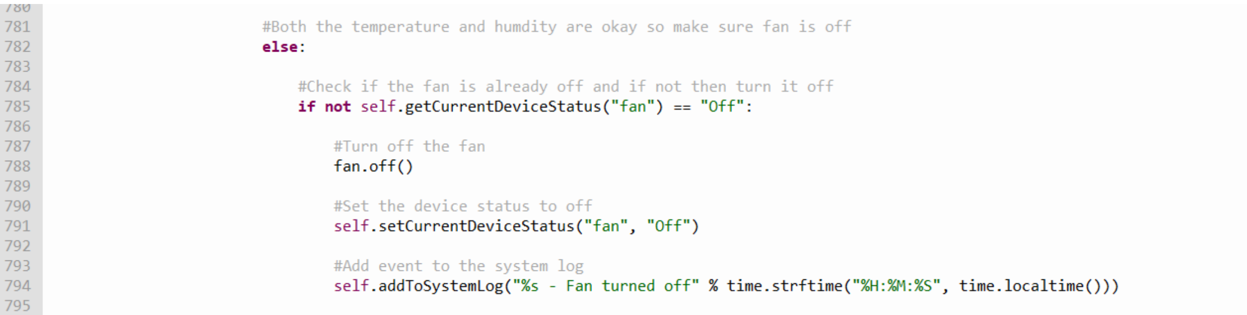
```
        #Fan
        #Only control the fan if the devices status is on
        if self.getDeviceStatus("fan") == "On":

            #See what mode the fan is in
            #Adaptive means the operation will be controled by
            #the fan algorithm
            if self.getDeviceMode("fan") == "Adaptive":
                #When the temperature parameter + 2.5 is exeeded then turn
                #on the fan to bring the temperature down or when the humidity
                #+ 2.5 is exeeded turn on the fan
                if sensors.getTemperature() > self.getParameter("temperature") + 2.5 or sensors.getHumidity() > self.getParameter("humidity"):

```

The fan is used to control two different environmental values inside the greenhouse. When the temperature is too hot then the fan needs to be turned on and when the humidity is too high then the fan needs to be turned on. Since the heating algorithm purposely overheats the greenhouse, we need to wait until the temperature goes 2.5 above the temperature parameter reading before the fan comes on. Otherwise, it will be working against the heating algorithm. When the humidity goes above the humidity parameter the fan also needs to turn on.



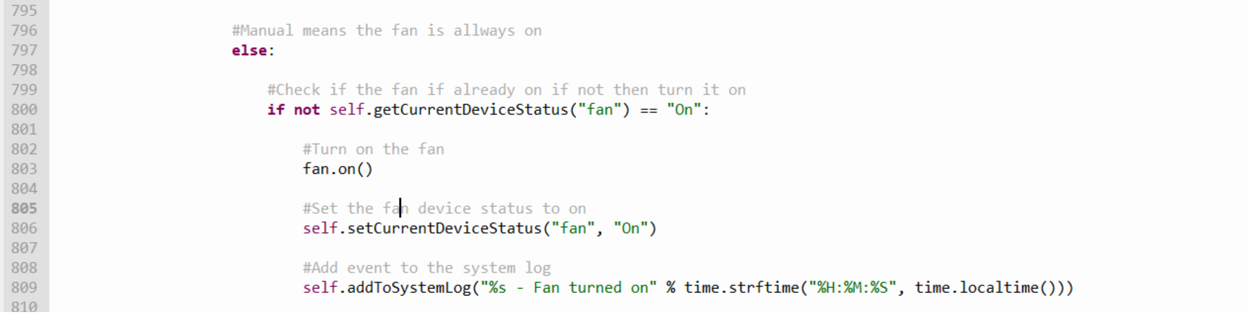
```
                    #Check if the fan is already on and if it is not then
                    #turn it on
                    if not self.getCurrentDeviceStatus("fan") == "On":

                        #Turn on the fan
                        fan.on()

                        #Set the device status to on
                        self.setCurrentDeviceStatus("fan", "On")

                        #Add event to the system log
                        self.addToSystemLog("%s - Fan turned on" % time.strftime("%H:%M:%S", time.localtime()))

```

When the temperature or humidity is too high the fan Is turned on.



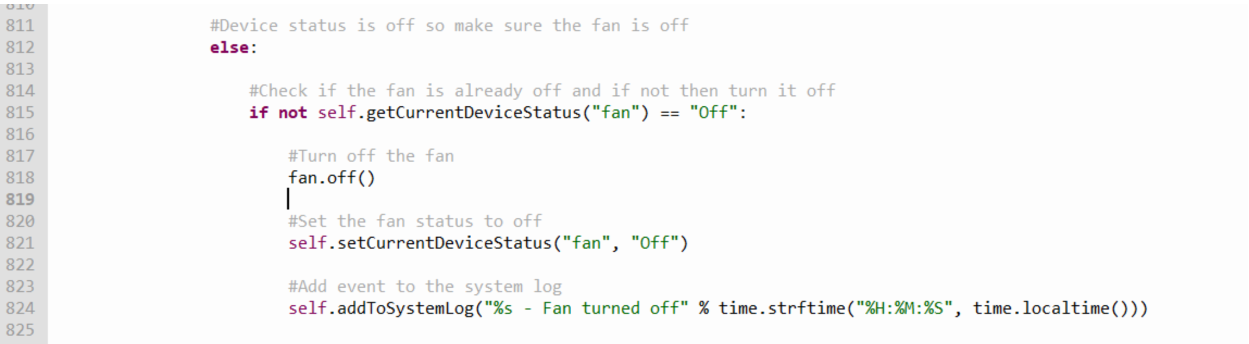
```
                #Both the temperature and humdity are okay so make sure fan is off
                else:

                    #Check if the fan is already off and if not then turn it off
                    if not self.getCurrentDeviceStatus("fan") == "Off":

                        #Turn off the fan
                        fan.off()

                        #Set the device status to off
                        self.setCurrentDeviceStatus("fan", "Off")

                        #Add event to the system log
                        self.addToSystemLog("%s - Fan turned off" % time.strftime("%H:%M:%S", time.localtime()))

```

When both the temperature and the humidity are low enough the fan is off.



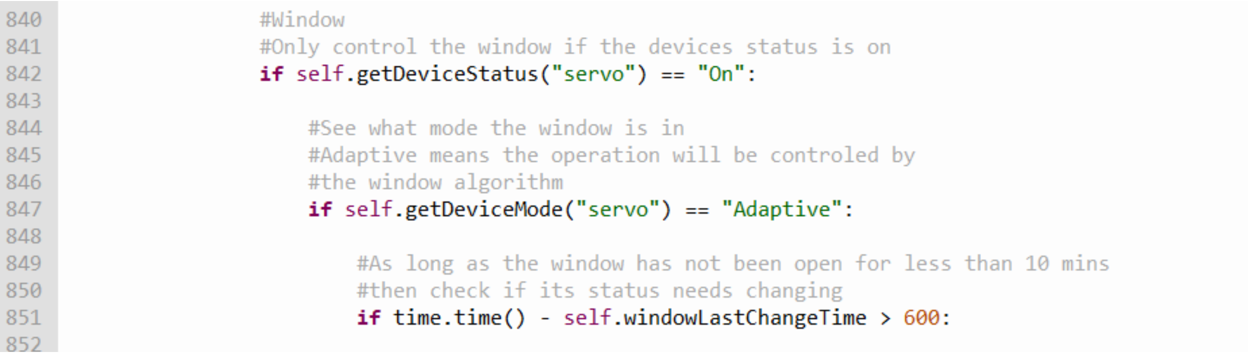
```
                    #Manual means the fan is allways on
                    else:

                        #Check if the fan if already on if not then turn it on
                        if not self.getCurrentDeviceStatus("fan") == "On":

                            #Turn on the fan
                            fan.on()

                            #Set the fan device status to on
                            self.setCurrentDeviceStatus("fan", "On")

                            #Add event to the system log
                            self.addToSystemLog("%s - Fan turned on" % time.strftime("%H:%M:%S", time.localtime()))
```

When the fan is in manual mode the fan is always on.



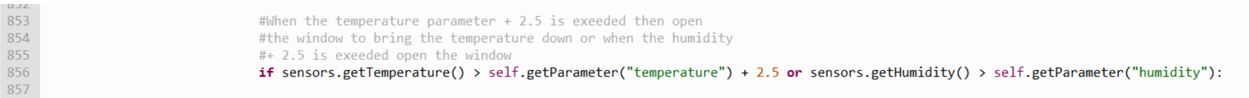
```
                #Device status is off so make sure the fan is off
                else:

                    #Check if the fan is already off and if not then turn it off
                    if not self.getCurrentDeviceStatus("fan") == "Off":

                        #Turn off the fan
                        fan.off()

                        #Set the fan status to off
                        self.setCurrentDeviceStatus("fan", "Off")

                        #Add event to the system log
                        self.addToSystemLog("%s - Fan turned off" % time.strftime("%H:%M:%S", time.localtime()))
```

The fan being in device status off means that the fan is always turned off.

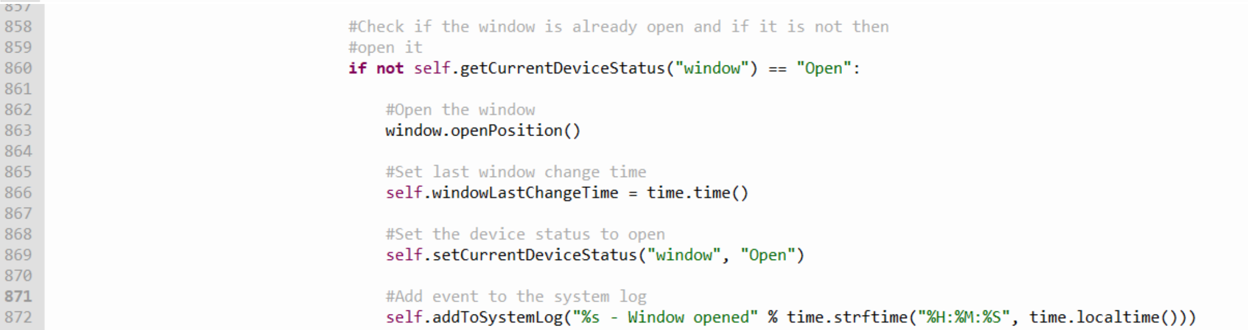


```
                #Window
                #Only control the window if the devices status is on
                if self.getDeviceStatus("servo") == "On":

                    #See what mode the window is in
                    #Adaptive means the operation will be controled by
                    #the window algorithm
                    if self.getDeviceMode("servo") == "Adaptive":

                        #As long as the window has not been open for less than 10 mins
                        #then check if its status needs changing
                        if time.time() - self.windowLastChangeTime > 600:
```

The final device is the window. The window mirrors the function of the fan meaning when the temperature is too high the fan and window both turn on and when the humidity is too high. The window is always left open for at least 10 minutes to avoid the window opening and closing rapidly when the parameter readings from the greenhouse are around their user set values.

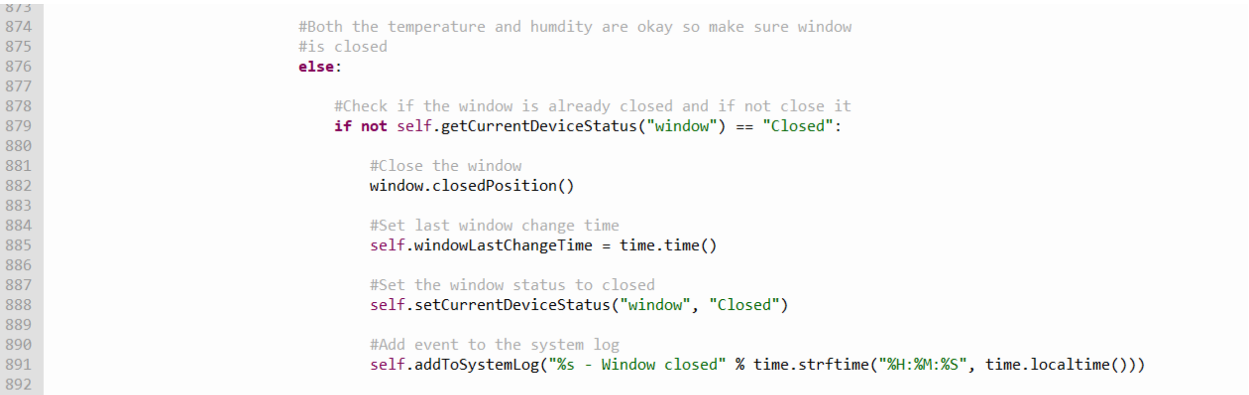


```
                            #When the temperature parameter + 2.5 is exeeded then open
                            #the window to bring the temperature down or when the humidity
                            #+ 2.5 is exeeded open the window
                            if sensors.getTemperature() > self.getParameter("temperature") + 2.5 or sensors.getHumidity() > self.getParameter("humidity"):
```



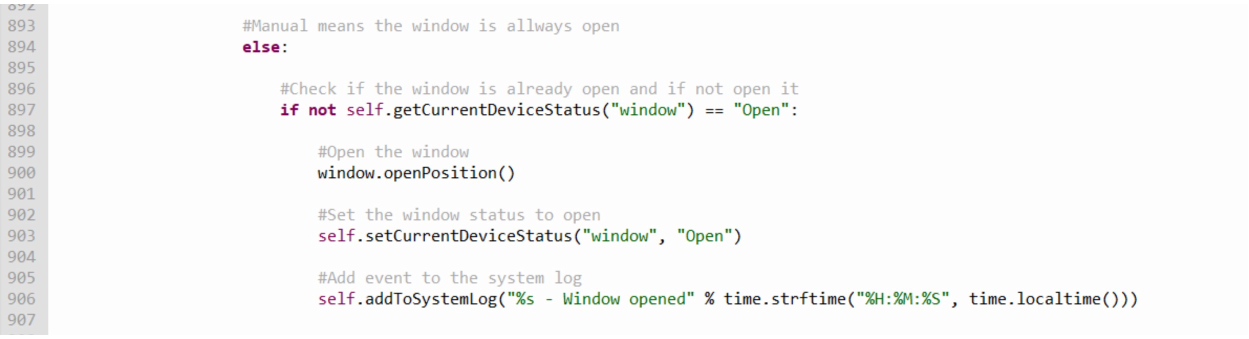
```
                                #Check if the window is already open and if it is not then
                                #open it
                                if not self.getCurrentDeviceStatus("window") == "Open":

                                    #Open the window
                                    window.openPosition()

                                    #Set last window change time
                                    self.windowLastChangeTime = time.time()

                                    #Set the device status to open
                                    self.setCurrentDeviceStatus("window", "Open")

                                    #Add event to the system log
                                    self.addToSystemLog("%s - Window opened" % time.strftime("%H:%M:%S", time.localtime()))
```

When the temperature gets too high, or the humidity gets too high just like with the fan the window is opened.



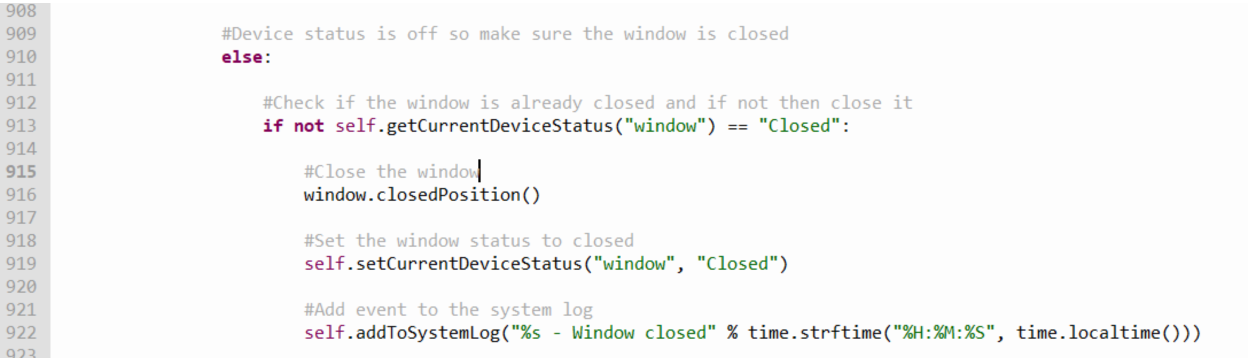
```
                            #Both the temperature and humdity are okay so make sure window
                            #is closed
                            else:

                                #Check if the window is already closed and if not close it
                                if not self.getCurrentDeviceStatus("window") == "Closed":

                                    #Close the window
                                    window.closedPosition()

                                    #Set last window change time
                                    self.windowLastChangeTime = time.time()

                                    #Set the window status to closed
                                    self.setCurrentDeviceStatus("window", "Closed")

                                    #Add event to the system log
                                    self.addToSystemLog("%s - Window closed" % time.strftime("%H:%M:%S", time.localtime()))
```

If the readings are okay, then the window is closed.



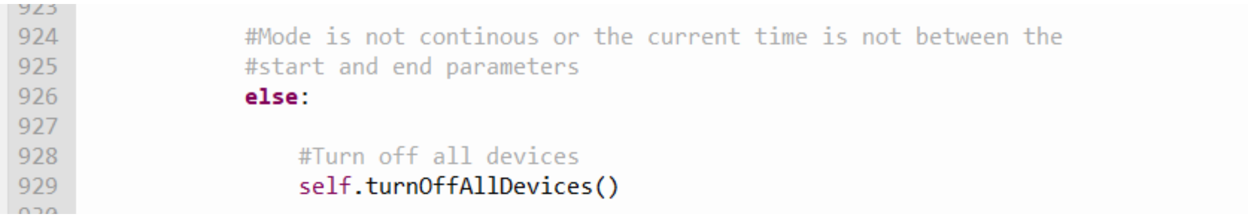
```
                    #Manual means the window is allways open
                    else:

                        #Check if the window is already open and if not open it
                        if not self.getCurrentDeviceStatus("window") == "Open":

                            #Open the window
                            window.openPosition()

                            #Set the window status to open
                            self.setCurrentDeviceStatus("window", "Open")

                            #Add event to the system log
                            self.addToSystemLog("%s - Window opened" % time.strftime("%H:%M:%S", time.localtime()))
```

When the greenhouse mode is manual the window is always in the open position.



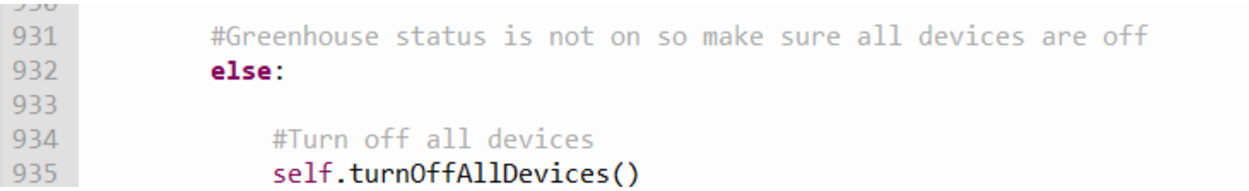
```
                #Device status is off so make sure the window is closed
                else:

                    #Check if the window is already closed and if not then close it
                    if not self.getCurrentDeviceStatus("window") == "Closed":

                        #Close the window
                        window.closedPosition()

                        #Set the window status to closed
                        self.setCurrentDeviceStatus("window", "Closed")

                        #Add event to the system log
                        self.addToSystemLog("%s - Window closed" % time.strftime("%H:%M:%S", time.localtime()))
```

Finally, if the window status is set as off the window Is closed.



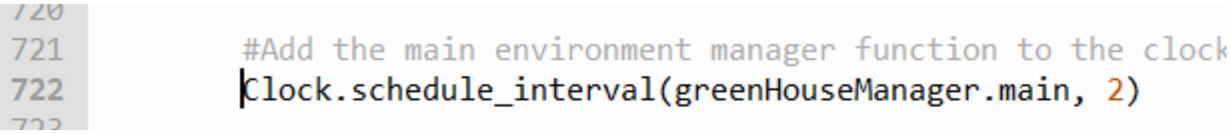
```
            #Mode is not continous or the current time is not between the
            #start and end parameters
            else:

                #Turn off all devices
                self.turnOffAllDevices()
```

When the mode of the greenhouse is not continuous, or the current time is not between the start and end parameters of the greenhouse then no devices should be functioning. In this case the turn off all device's method is used to ensure all devices are off.



```
        #Greenhouse status is not on so make sure all devices are off
        else:

            #Turn off all devices
            self.turnOffAllDevices()
```

The greenhouse can also be set to have a status of off and in this case the turn off all device's method is used to make sure all devices are off. At this stage the main method is complete and can monitor all the different parameters inside the greenhouse and act accordingly as they change.



```
        #Add the main environment manager function to the clock
        Clock.schedule_interval(greenHouseManager.main, 2)
```

The greenhouse manage main method is added to the system clock inside the kivy main app build method. I am calling the method every 2 seconds meaning that the greenhouse will be constantly monitored and controlled.

Testing

Test Number	**Test Plan**	**Expected Outcome**	**Actual Outcome**	**Pass/Fail**
1	Run the demo mode	All the devices should come on	All the devices came on	Pass
2	Set the greenhouse mode to continuous	The program should begin to monitor the greenhouse	The program started to turn on some of the devices inside the greenhouse showing that it was working	Pass
3	With the heating in adaptive mode set the temperature parameter to a value above the current temperature reading	The lamp should come on and heat the greenhouse until the parameter + 2.5 is reached	The lamp came on and then went off once the greenhouse was heated	Pass
4	Wait for the temperature to drop by 5 degrees	The lamp will come back on and reheat the greenhouse	The lamp came back on and heated the greenhouse up again	Pass
5	Set the heating mode to manual	The lamp will come on constantly	The lamp came on	Pass
6	Set the heating status to off	The lamp will turn off	The lamp was off	Pass
7	With the pump in adaptive mode and the soil dry see that the pump comes on	The pump should come on	The pump came on	Pass

8	Wait for the pump to turn off the moisture sensor	The pump should turn off once the plant is watered	The pump was turned off when the moisture sensor detected moisture	Pass
9	Make the pump mode manual	The pump should be constantly on	The pump was on	Pass
10	Set pump status as off	The pump should be off	The pump was off	Pass
11	With the leds in adaptive mode and the light parameter above the current light level see that the lights come on	The lights should come on and stay on for 10 minuets	The lights stayed on for 10 minuets	Pass
12	See that the lights stay off for the next 10 minuets	The lights should stay off for 10 minuets	The leds were off for 10 minuets	Pass
13	Set the light parameter below the current light level	The leds should be off	The leds were off	Pass
14	Set the led mode to snake	The snake should begin on the led strip	The led snake began	Pass
15	Change the led mode from snake to disco	The leds should swap from snake mode to disco and end the snake thread	The mode swapped to disco, and the thread was closed	Pass
16	Make the led mode manual	The leds should be filled with white	The leds went white	Pass

17	Set the led status as off	The leds should go off	The leds were turned off	Pass
18	Set the temperature parameter at least 2.5 below the current temperature	The fan and window should come on for 10 minuets	The fan and window came on for 10 minuets	Pass
18	Set the humidity parameter lower than the current humidity	The fan and window should come on for 10 minuets	The fan and window came on for 10 minuets	Pass
19	Set fan mode to manual	The fan should be constantly on	The fan was always on	Pass
20	Set fan status to off	Fan should be turned off	The fan was turned off	Pass
21	Set the window mode to manual	The window should open	The window opened	Pass
22	Set the window status to off	The window should close	The window closed	Pass
23	Set the greenhouse mode to scheduled and make sure the current time is between the start and end times	The greenhouse should function as usual	The greenhouse was functioning and controlling devices	Pass
24	Set the start and end time so that the current time is not inside the range	The greenhouse should not function	The greenhouse did not operate, and all devices were off.	Pass

25	Set the greenhouse status as off	The greenhouse should not function	The greenhouse did not function, and all devices were off	Pass

Review

This stage saw the development of the main function which acts as the backbone of this greenhouse. I believe this stage was a success as no errors were produced in the test plan thanks to the robustness of the previous classes I have developed allowing me to control devices safely, the use of a dictionary to track device states so that no collisions happen like trying to turn on a device which is already on and via the use of two flags to keep track of and effectively close down threads from the led class. At this stage the greenhouse is fully functional and can carry out the control of a plant environment inside the greenhouse.

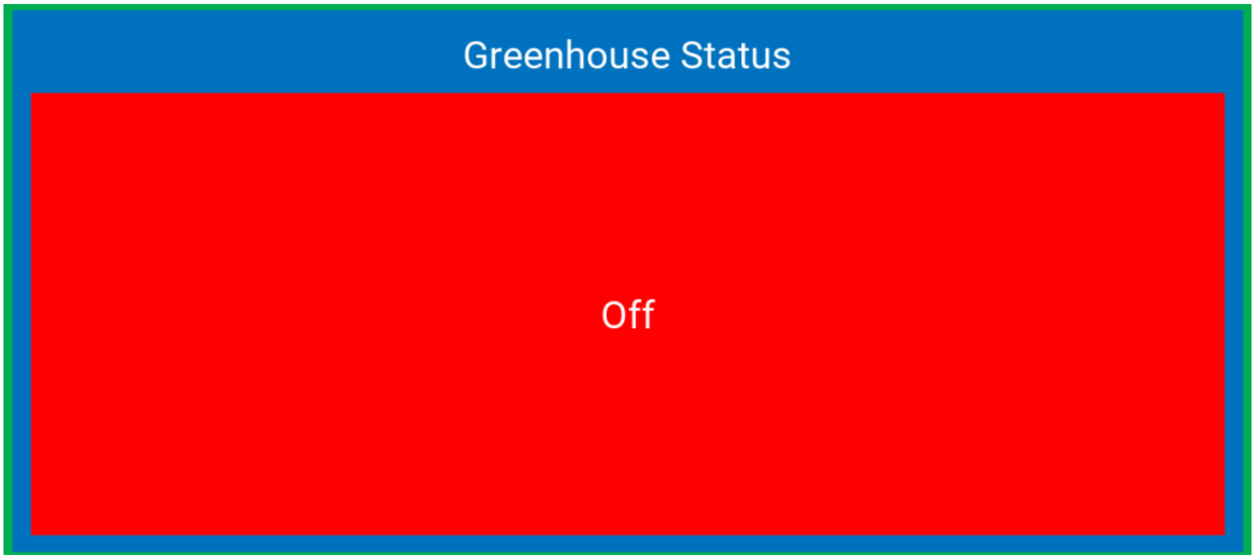
Iterative stage 17 – Graphs

Overview

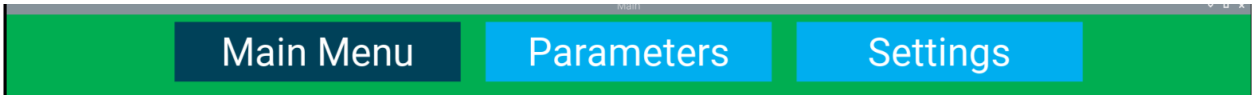
This stage will be focused on the implementation of the graph page and the mini graph on the main menu. These graphs will display to the user historic sensor readings from the greenhouse and give the user the ability to change the graph axis and time scales. This stage will see the implementation of sensors readings being saved to a file for use by the graph section of this project.

Development log

Unfortunately, after many hours spent attempting to implement graphs into kivy I have conclude that it is no possible. Whilst there is a graphs feature in kivy its function is erratic and cannot plot new data points as they are taken from the enviro class. Installing the kivy graph garden took in itself a good few hour. Mainly since kivy somewhere along the line decided to change the way open source kivy plugins should be installed but neglected to mention this fact anywhere. Due to this and other issues encountered trying to implement graphs into kivy I have decided to remove this feature from my program. This means my project will end here as the main functions of the greenhouse are now being carried out with the user having full customization ability over the environment. The remainder of this stage will show the changes I have made to the gui to take out the graph's sections.



On the home page I have removed the section which was originally designed to show a mini graph of live data from the greenhouse and in its place extended the size of the greenhouse status toggle button.



The graphs page has been completely removed from my code and the page has been taken out of the screen manager. I have rearranged the navigation bar so that the 3 pages are grouped together centrally.

Review

It would have been nice to implement graphs into this project as it would have served a useful function for the user to see how the different environmental variables inside the greenhouse were changing. I feel the size of this iterative stage does not do the amount of time I've spent trying to implement graphs justice. Nevertheless, the greenhouse is now in a fully functional and complete stage. Which can be used by an end user without any issues.

Overview

The greenhouse is now complete in this overview section I will briefly talk about the development process outlining things that went well and things that proved to be a challenge.

The greenhouse system is developed so that it has the following features -

- Live greenhouse readings
- Live device status
- System Log
- Demo Mode
- Continuous mode
- Scheduled mode
- Update parameters
- Update settings
- Change device mode and status

From the beginning of this project my aim was to produce the system in an object-oriented modular manor. I feel that using various classes contained inside their own files I have created robust libraries which can be used for their respective uses by other programmers trying to solve the same problems as me. When implemented inside my greenhouse system they perform the various functions required by the greenhouse. I feel that the use of an object-oriented approach allowed me to never feel overwhelmed by the amount of code that I have written and always have a good handle on how different classes needed to interact inside the program.

The use of comments in my code meant I was quickly able to familiarize myself with the function of different areas of my software when I had not been developing that section for a while. If I was going to do this project again, I would probably do less commenting of code as I feel there was not the need to comment every line of code.

The implementation of a thermostat style algorithm for the heating lamp was an interesting section to develop. I had debated from the start of the project how I would solve the issue of the flickering lamp and in the end, I felt the thermostat algorithm was the most elegant solution as this ensured the average temperature would be equal to the users entered parameter. Without moving the temperature too far either side of the desired value.

A major issue in this project was the use of Kivy which proved time and time again to be a source of great frustration. With issues such as the phantom mouse clicks, countless installation errors and the graphs implementation taking up a great chunk of the time I spent developing this project. One of my greatest frustrations was the fact that the vast majority of the kivy online help documents would not work when implemented into kivy.

Overall, I am very happy how this project has turned out I feel with more time I would have been able to implement some of the bonus features such as the remote access and email alerts. However, at this stage the greenhouse is fully functioning and capable of performing its main job of controlling a greenhouse environment.