[Date]

Harin Vyas

El Generico®

[Document title]

[Document subtitle]

Contents

[Section 1 – Analysis 2](#_Toc21594204)

[Part 1 – Description of the problem 2](#_Toc21594205)

[Part 2 – Stakeholders 2](#_Toc21594206)

[Part 3 – How the problem can be solved by computational methods 3](#_Toc21594207)

[Part 4 – Software and Hardware requirments 4](#_Toc21594208)

[Part 5 – Success Criteria 4](#_Toc21594209)

[Section 2 – Design 8](#_Toc21594210)

[Part 1 - Introduction 8](#_Toc21594211)

[Part 2 - How Kivy works 8](#_Toc21594212)

[Notes 9](#_Toc21594213)

[Part 3 – RegisterPage Class (1.1) 9](#_Toc21594214)

[Part 4 - LoginPage Class (1.2) 15](#_Toc21594215)

[Part 5 - AddLocationForm (1.3) 19](#_Toc21594216)

[Part 5 – ICAOFinder 23](#_Toc21594217)

El Generico® Company

# Section 1 – Analysis

## Part 1 – Description of the problem

El Generico® strives to make high-quality applications for our wide range of clients including myself, myself and myself. The problem which we shall be addressing for our client is the problem of aviation weather. Normally, pilots will get the generic flight info before the flight but the live data is provided using a Meteorological Terminal Aviation Routine Weather Report (METAR). This is what mainly determines what runway shall be in use, how you would land your plane and if you will be visually landing the plane (VFR) for electronically (IFR) (and probably some other stuff too).

Traditionally, this is given via Radio using a Text to Speech (TTS) but the pilot still has to decode this information. This is where the problem is. It is inconvenient to get you METAR like this: EGCC 111420Z 01012KT 5000 RA SCT021 BKN030 10/08 Q1018 TEMPO -RA. When you can get it decoded which makes it easier to understand so making it less likely to have air accidents as well as lowering flight time. The solution must be a way to present METAR in a more readable format

## Part 2 – Stakeholders

Geoffrey Windsor – Pilot for Virgin Atlantic

Using my interview template.

1. What would you expect from this type of app?

I would expect it to tell me the weather.

Search functionality to get ICAO codes for me (I am very forgetful).

Little icons to show weather.

1. What would make this app different from the alternatives?

Using a TTS system so I don’t have to look at my screen.

1. Does [the problem] exist in your field of work and do you think that my app can solve it?

The problem does exist, the normal METAR format is long and decoding it would make life much easier. Yes

1. What sort of customisation would you like?

Colour theme (dark mode minimum) and preferably other themes as well.

Joseph Jones– Flight sim Enthusiast

Using my interview template.

1. What would you expect from this type of app?

Accurate info and data in a correct pilot format. Concise + minimalistic

1. What would make this app different from the alternatives?

Free of charge with no extra money for premium features.

1. Does [the problem] exist in your field of work and do you think that my app can solve it?

Exists for new people and the younger people who come to ‘flight simming’ and find it hard to understand the terms

1. What sort of customisation would you like?

Dark mode. Enhanced brightness for night time use. Airline based themes. Real-time updates.

Summary

From the interviews, I can summarise that the users would like the app to be able to give an accurate decoded METAR report in a readable format. As well as having a search functionality for ICAO codes and for the data to be presented in a concise and minimalistic format that includes appropriate icons.

We shall be able to stand out from the competition as we have a completely free app (except maybe donations) and a TTS system which helps when your hands are not free.

The problem mentioned does exist in the stakeholder’s fields of work/hobby so there is surely a need for the app. My app should be able to solve the problem by being able to just present decoded METAR.

The App should have at least dark mode and light mode. But it is preferred for other themes to be included for example themes for the airline you are flying for. It should also have enhanced brightness for dark conditions so the text is readable and the app should have real-time updates on the weather.

## Part 3 – How the problem can be solved by computational methods

Thinking Abstractly and Visualisation

Abstraction can be used in the following ways:

* When the data is received from the API, it is in a complicated JSON format so the app should extract the stuff which is needed and present it in a clear format. The large JSON data is hidden from the user,
* Instead of worrying about class inheritance to move the user’s data around the screens, we just set it as a global variable to make it easier to code and understand. The unnecessary need for class inheritance is cut out.

Thinking Ahead

* The App should be responsive. It should work on different screen sizes and operating systems as the users could be using a range of different devices to access the app. Without this, we would be narrowing the audience of the app to only Android users for example which would mean that there would be fewer users on the App so it wouldn’t be solving the problem for everyone.
* The App should be accessible. It should have settings where users can change aspects of the app to suit them. Aspects include Dark/Light Mode, Language, Large text and a filter for color-blind people. This again gives a larger audience as well as hopefully standing out against any potential competition. Settings data will probably be stored in the database with numbers representing each option for the settings.

Thinking Procedurally & Decomposition

* There will be a login/ register system, a way to find an ICAO and a way to search for the weather.

Decomposition is done in Section 2.

Thinking Logically

* Naturally, the app running state would be a loop in itself. Kivy (the framework which I am using) has this built into the ‘App’. When the user does anything (types, presses a button, etc), the game state shall stop for the respective code to be run. The app running state will be a constant iteration that will end and close the app when the user exits the app.

## Part 4 – Software and Hardware requirements

Hardware

* A Windows 7 or above PC with basic I/O devices
* At least 2GB of RAM
* 2GB free storage

Software

* A python interpreter
* Kivy module and dependences
* pyttsx3 module for TTS
* google translate module for translation

Other

* A stable internet connection, preferably 15 Mbps or above.

## Part 5 – Success Criteria

The app must have at minimum:

Register System:

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Criteria | Why it’s needed | How to check |
| 1 | A register system | So, we can add new users who can log in easily to the system. Also, requirement from clients | We are hoping for a working register system that can add new users and give error messages based on inputs. |
| 2 | Appropriate validation of the inputs | So, there are no overlaps in the user’s data as well as no duplicate users with different passwords as this can mess up the login system. | Test it with all sorts of inputs e.g. missing inputs, existing users, etc. |
| 3 | The data to be kept secure e.g.: hashing passwords with salt. | This ensures that nobody who may have acquired access to the user’s data cannot read their password off right away. Using salt ensures that hackers cannot use rainbow tables to get the passwords of users. | The passwords should be hashed with salt where the data is stored. |
| 4 | The user’s data is set up in the correct format (TBD) | So, there is consistency in the data and to make it easier to parse and interpret the data. | Check where the data is stored that new user’s data is in the correct format |
| 5 | Error messages if unsuccessful | So, the users know that their registration is unsuccessful and why it is unsuccessful. | Check by putting in data that should return an error message and check if the error message has appeared. |
| 6 | Success message | So, the user doesn’t wonder, why isn’t it done yet? They know that they can now log in. | ^ but with data that would return a success message. |

Login System

|  |  |  |  |
| --- | --- | --- | --- |
| Id | Criteria | Why it’s needed | How to check |
| 7 | A login system | Well, what good is a register system without a login system? Users need to be able to log in once they’ve registered of course. | If there is a working login system with good validation which takes the user to the main screen when successful. |
| 8 | Appropriate validation of the inputs | So, we don’t waste processor time trying to log in on a user which doesn’t exist | Fire in some inputs which should not make the user log in e.g. no password filled or the username not existing |
| 9 | Matching the hashes | This will hash the inputted password (with salt) to be able to match the hashed password as you cannot unhash a hashed password (unless you use PHP). | We need to check that when we input the correct password, it should log in which shows that the hashes have been matched. |
| 10 | Error messages if unsuccessful | So, the user knows that he messed up big time (and why). | Fire in some bad data, if we get an error message then we’re cool. |
| 11 | On success, go to the main screen (TBD) | So, the user knows that he’s logged in as he’s now on the main screen. As well as to reassure them that they have done everything right. | Fire in some correct data (can be done at same time as matching the hashes) if the main screen loads then we’re Gucci |
| 12 | On success also save their data as a global variable until the program shuts down. | So, the user’s data can be used/updated in other parts of the program. For example, recent searches or settings preferences can be read and updated when necessary. | This we can only check once we have linked this data to something else e.g. a recent search system if they load properly then it works. |

The metar search system.

|  |  |  |  |
| --- | --- | --- | --- |
| Id | Criteria | Why it’s needed | How to check |
| 13 | METAR search system | As it is the main point of the program innit. | A system which presents a decoded accurate METAR in a clear and concise format |
| 14 | A recent search system | So, there would be little point in saving the user's data then. It also allows for pilots who fly routes frequently. | Check if the recent searches have merged from the user’s data and when you click it, it automatically goes in the search box. |
| 15 | Backend code which searches for the METAR using an API and returns the results | It is the main purpose of the program of course. | Check-in console with a print statement. |
| 16 | The results to be presented in a clear format | This will most likely be a list view with each section of the decoded metar on a separate line. | If when we show it to our shareholders, they find it sufficient. |
| 17 | A way for the user to search for the ICAO code | As this is a shareholder requirement because people can forget their ICAO codes. This will probably be implemented in another part of the app with buttons linking the 2. | Check that when a button linking the 2 screens is clicked, the screen changes. |

The ICAO search system

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Criteria | Why it’s needed | How to check |
| 18 | ICAO search system | Shareholder requirement from Geoffrey. As users can forget their ICAO code. | A system that accurately returns all results available based on the query in a clear format. |
| 19 | A recent search system | As the users can forget what they are looking for (this one is aimed mostly at enthusiasts as pilots shouldn’t really be forgetting this sort of stuff) | The data from the user’s data should be shown there and when it is clicked, the recent search goes into the search box. |
| 20 | Search results to be clear and accurate | So, it makes it easiest for our users to understand. | Ask the shareholders if it is fine or not. |
| 21 | A way to navigate back to the METAR search system | It allows the app to be smooth and the user to use the newly discovered ICAO code for good use. | Check that when a button linking the 2 screens is clicked, the screen changes. |

General stuff

|  |  |  |  |
| --- | --- | --- | --- |
|  | Criteria | Why it’s needed | How to check |
| 22 | Dark mode and light mode | Dark mode to satisfy our shareholders and for users so be able to see better at night (as well as their eyes not dying from the sun god beaming down on them with light mode). Light mode is better for daytime (or normies) so that should also be an option. | It should be by default dark mode and when the option is set for light mode, it should change to what is generally considered light mode. |
| 23 | A TTS system | So, the users can hear the METAR when their hands are not free e.g. finishing off the checklists. | When the TTS button is clicked, you should be able to hear the METAR. |
| 24 | If the app is disturbed using windows store, app store, or play store the app must be kept free | To make sure that those who cannot afford such an app can still reap the benefits from it. It also lets us keep up with the competition which is generally paid apps. | Kinda obvious innit. |

What isn’t plausible

|  |  |
| --- | --- |
| What isn’t plausible | Why it isn’t |
| Real-time updates | This involves having to keep the app open at all times as well as having to fiddle about with time in Kivy which can mess up the app state horribly so rather not take the risk tbh. |

# Section 2 – Design

## Part 1 - Introduction

I will split this section by class. Each part will include the structure diagram, proposed screen designs, pseudocode, and any test data which we’ll be firing in.

As my structure diagram is too big to fit in one piece, I will have each class diagram below. Note that the parent of each class is the same (the main app).

The structure diagram was hand-drawn on OneNote (with dark mode on, of course). Rectangles at the lowest generation shows a process and circles at the lowest level shows some text

I have decomposed the problem using a Top-down Module Design as it is an easier way to visualize the program showing the parent and child classes, and the widget inheritance so we know what Kivy styling will affect what parts. It also shows how the python code for each class will work. What processes are needed (in the procedures) after clicking certain buttons? The number hierarchy is used in order to match parts of the structure diagram with the screen designs.

All pseudo-code was typed in Notepad ++ then a screenshot of that was used to put the pseudo-code in.

## Part 2 - How Kivy works

Kivy code (henceforth kv) works differently in most programming languages. It is best thought of as CSS as it is generally used to create ‘widgets’ (defined as “elements of a graphical user interface that form part of the User Experience. The kivy.uix module contains classes for creating and managing Widgets. “- <https://kivy.org/doc/stable/api-kivy.uix.html>). This includes things like the screen manager, UX widgets (e.g.: dropdown lists, buttons, text boxes, etc.) and layouts (how UX widgets are laid out on the screen e.g. boxlayout).

You would typically have a ‘root’ class (the parent class) that handles the screens (each of its children classes) e.g.: a login page and a register page would be screens which inherited by the root class. As the screens are children classes, they will also inherit from the root class (the parent) so if you added some code to change the background colour in the root class to pink (note that this does nothing to the root as the root is not a screen so it is not displayed), each of the screens will have a pink background colour.

In the python code, you would have what I will call a grandparent class which I believe can be best explained by the docs:

“The App class is the base for creating Kivy applications. Think of it as your main entry point into the Kivy run loop. In most cases, you subclass this class and make your own app. You create an instance of your specific app class [which includes your instance of the root and screens] and then when you are ready to start the application’s life cycle, you call your instance’s App.run() [so if I called my instance WeatherApp, I call it using WeatherApp().run() (note the ‘()’ at the end of WeatherApp. This is because we define the instance as a class in python)] method.”

(<https://kivy.org/doc/stable/api-kivy.app.html>)

So overall, it is best to think of it just like making a website. The root is the htdocs folder and each screen (or child class) is a webpage. The python code is like HTML scripting, you could use it do the jobs of JS and CSS (by the <script> and <style> respectively) but it is good practice to have separate .js and .css files and call them in (in Kivy, no code is needed for this). So, the kv file is the HTML and CSS which can create and style HTML elements (widgets in Kivy) and moves between the different webpages (screens for the sake of Kivy). The python code defines the screens, the root, and the app life cycle (through the grandparent class) and runs it all. The python code also performs most of the same code as JS can (even if modules are needed) which in my case will be done by adding methods and properties in the screens (as they are classes).

### Notes

Initial Imports:

* App class from kivy.app to create the base for our application (as quoted before)
* BoxLayout class from kivy.uix.boxlayout for the layout system for our app
* DropDown class from kivy.uix.dropdown to be able to use the dropdown widget
* ObjectProperty class from kivy.properties so we can call for an object property from the kv code (explained below)
* UrlRequest from kivy.network.urlrequest to be able to make web requests for our weather
* Screen and ScreenManager classes from kivy.uix.screenmanager to be able to access and switch between different screens.
* JSON, hashlib, and os from standard library for data storage hashing passwords and for our data storage system respectively.

In the python/pseudocode for python code, when something like varname = ObjectProperty() is used, this means that it gets (inherits) varname from the kv code so in this case, it will get the ‘varname’ property from the kv code. This is like the getElementByID method in JS.

## Part 3 – RegisterPage Class (1.1)

Breakdown of 1.1

The register page is split into a backend (with all the python code) and the frontend (with the Kivy code). The frontend will consist of a box layout that has been centred using padding on all sides (to make the page look better basically as centred layouts generally appeal to the user more). This layout will have 3 labels with 3 text-input boxes underneath. These are (in the vertical top to bottom order): Username, Password (with password formatting (\*’s) for security) and email. Each of these inputs will have an id assigned to them (username, password, email respectively) so they can be used in the python code via ObjectProperty when they are assigned as an object property in the Kivy code (e.g.: username\_input: username with the same for new properties password\_input and email\_input).

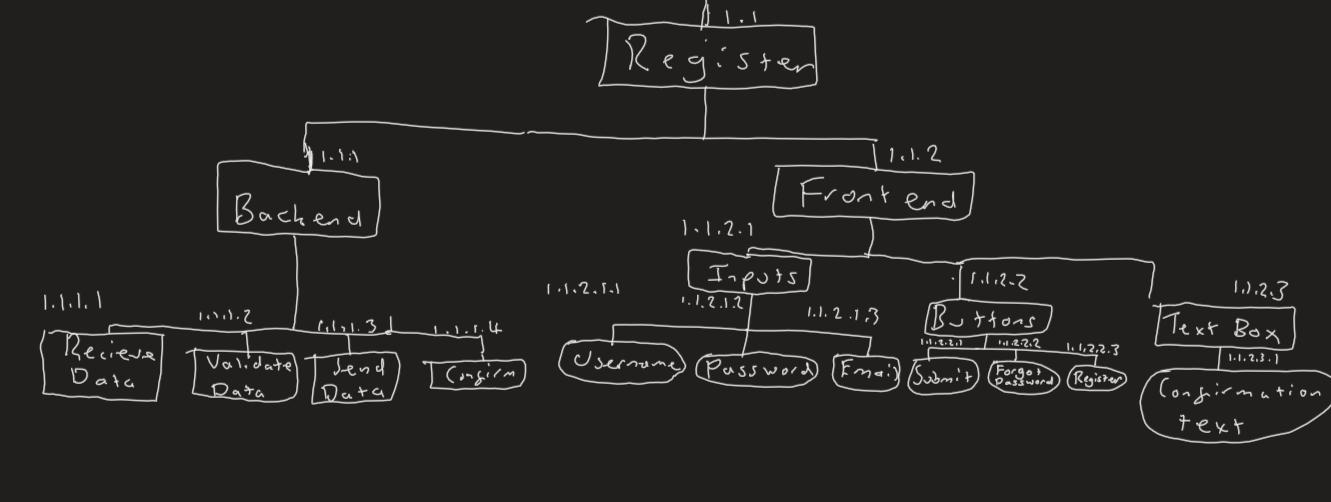
Next, there will be an anchor layout (who’s parent is the box layout) which has 2 buttons: submit which will trigger the backend code to run and log in (CHANGE ON SCREENSHOT) which takes the user to the login page (1.2) when clicked. Finally, there is the confirmation text which is a label with an id assigned to it (‘confo’ so at the top it is assigned to property ‘validation’ via ‘validation: confo’). In the screen design, a box has been shown to show where the confirmation text would appear but in-fact it is invisible until the backend changes the confirmation text. The confirmation text will either report any errors in the python code (e.g. missing the input for email or username is already taken) or will report that the registration was successful.

The backend will firstly get all the object properties from the Kivy code by doing varname = ObjectProperty(), this is done for all the object properties. Next, there should be a procedure that does some validation on the inputs so it should check if the form is properly completed and if the username/email is already taken (exists in the JSON file/DB). To use the JSON file/DB, it should open it using the JSON module/the DB's module and extract each user’s data into an embedded list (then the file closed) which then can be iterated over to check for the validation criteria. If any of these are true then the validation text (which has been brought over as an object property) should be changed to an appropriate message e.g. Form not completed. If it is fine then it moves to the next procedure.

Next, a new procedure will be used for the actual registration. A variable called ‘id’ shall be the highest number id in the JSON file/DB + 1. Next, a salt should be created and stored under a variable. Then a variable with the password hash is created which is equal to an md5 hash (using hashlib module) of the salt + the password\_input object property, this is encoded using utf-8. I am using hashing so a hacker cannot get the passwords and embedded salt to protect against rainbow tables. After this:

For JSON, update the extracted data with the new user’s data (id, username, password, email, etc.), in the correct formatting. Then the JSON file will be closed, then this data will be dumped to a new temporary file, the old one deleted and the temporary file renamed to the old one’s name using the os module.

For a DB, the DB will just be updated using the data at the end then the DB closed.

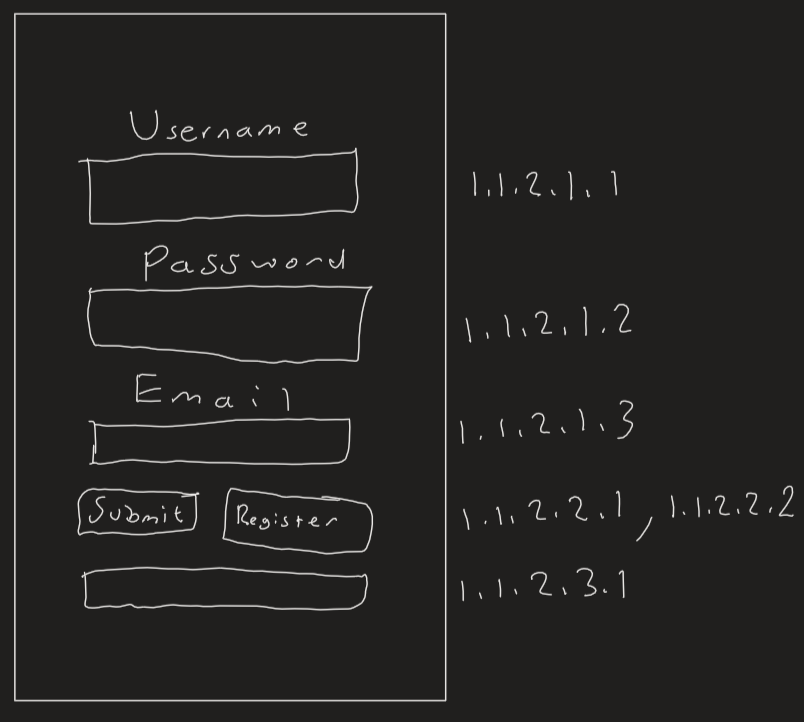
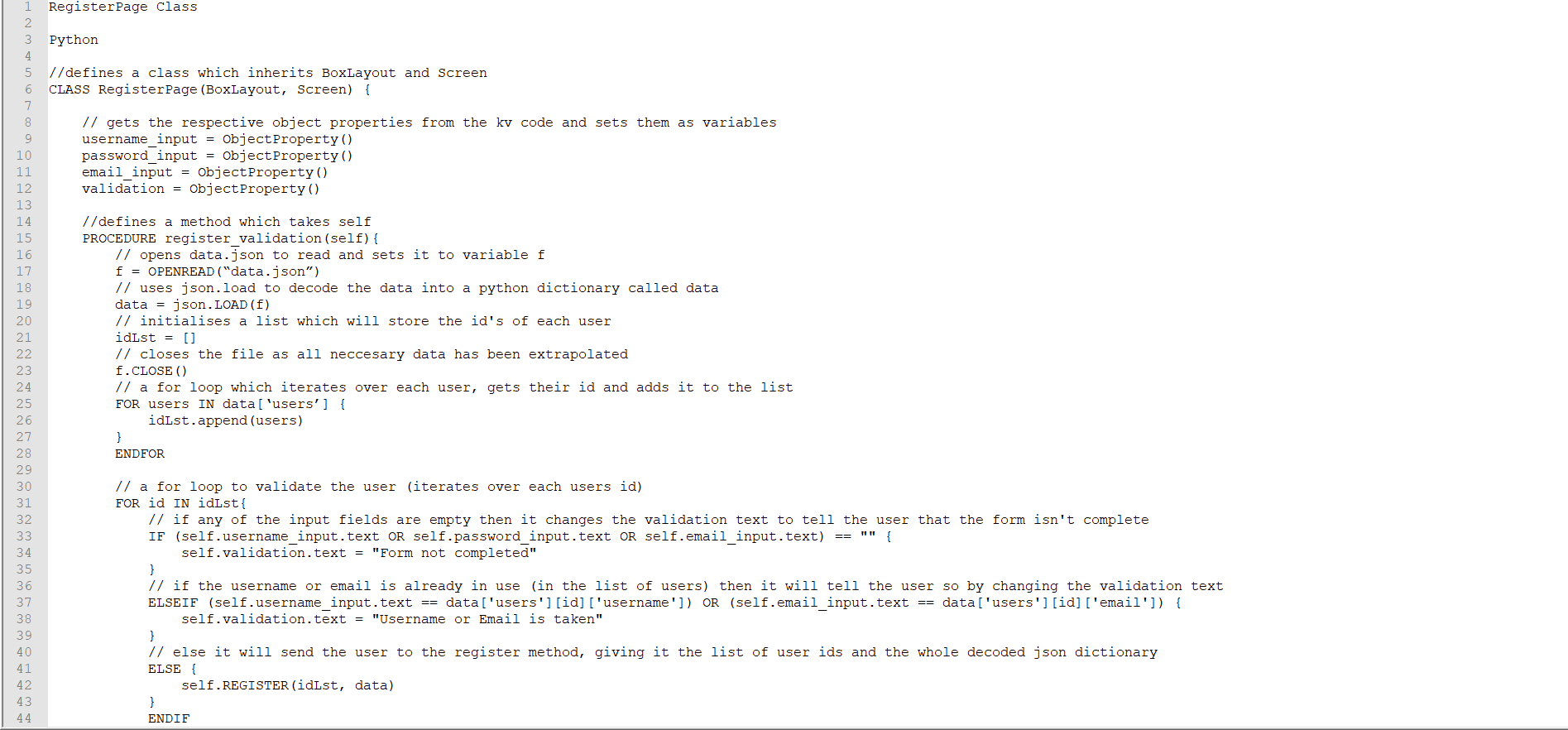
Either way, at the end of the code, the input boxes will be cleared and the validation text object property updated with a success message. MAYBE IN GREEN COLOUR FONT.

Structure Diagram:

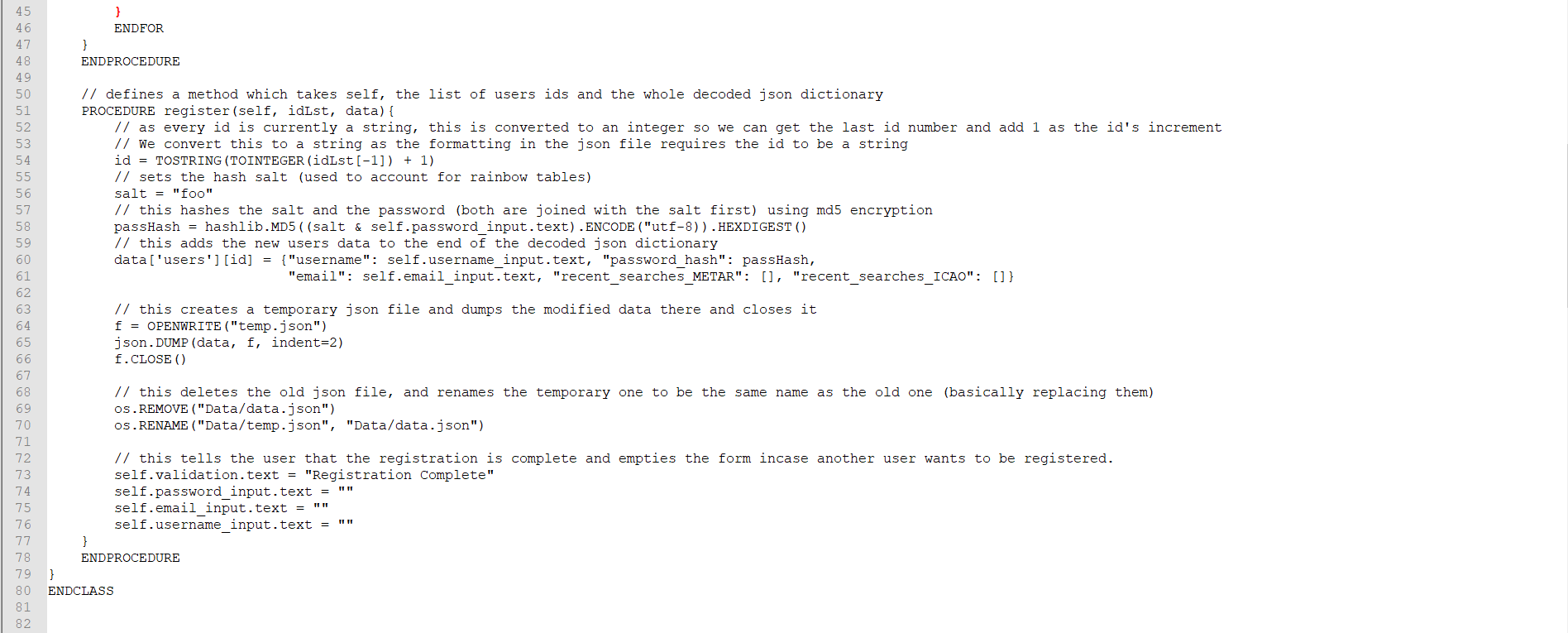
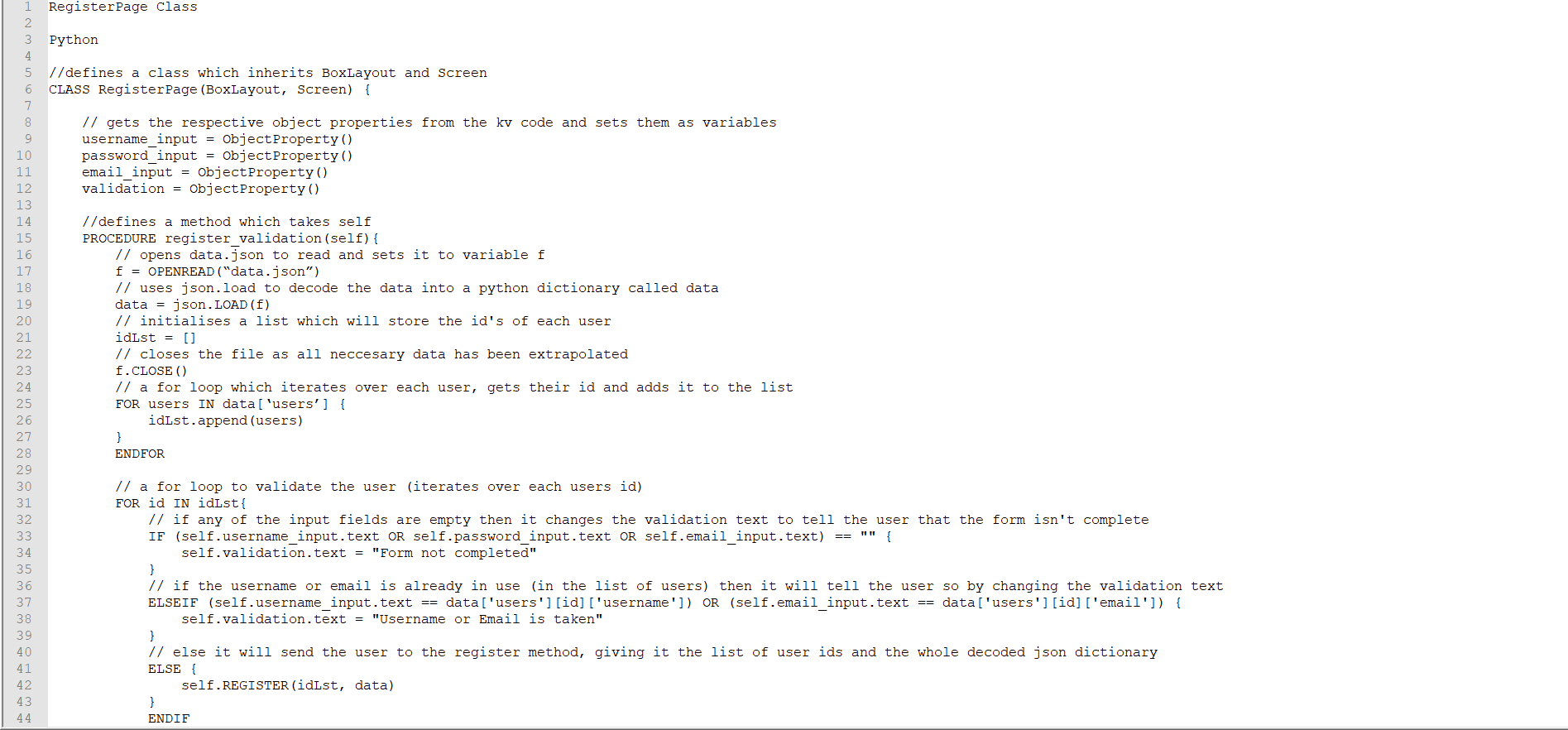
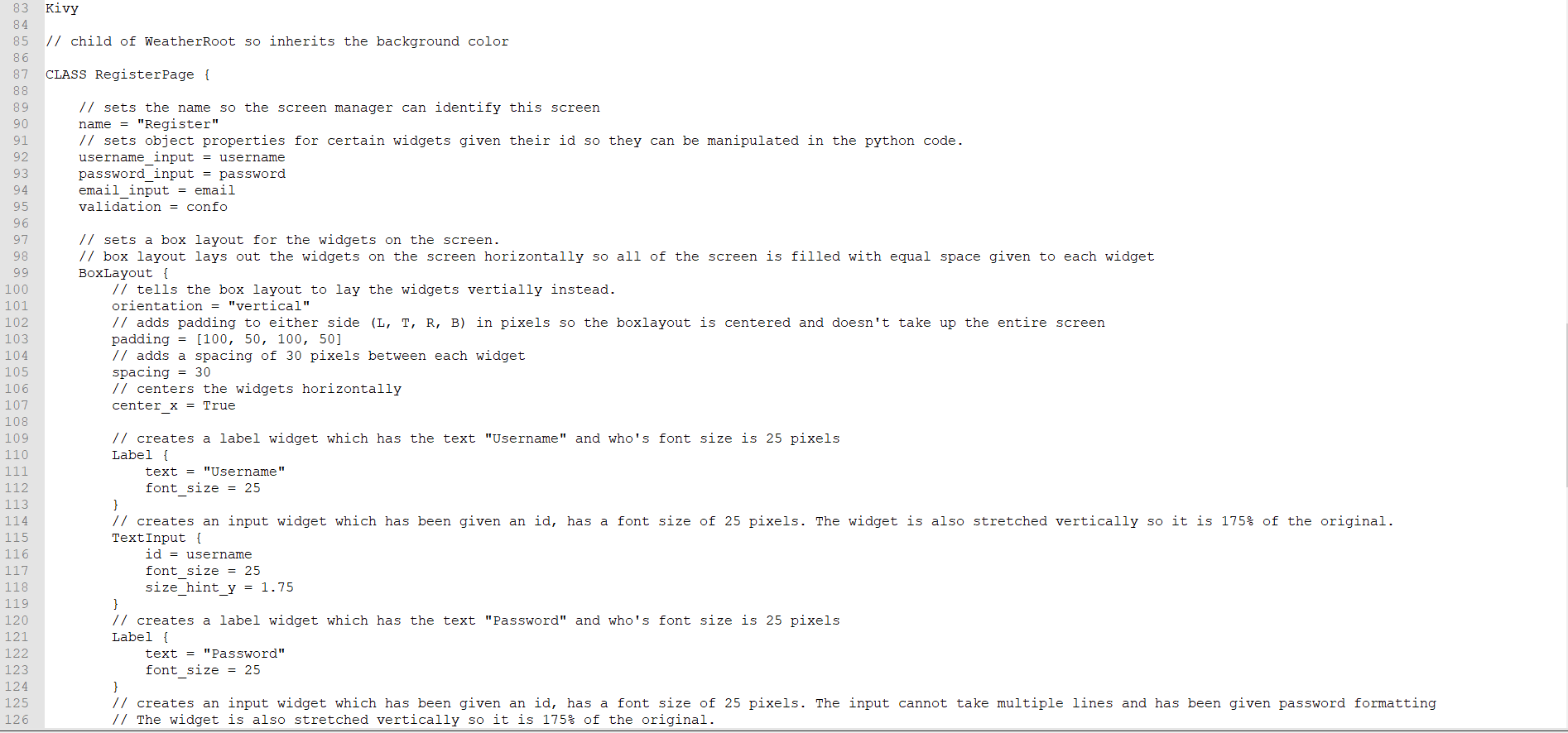
Key Variables:

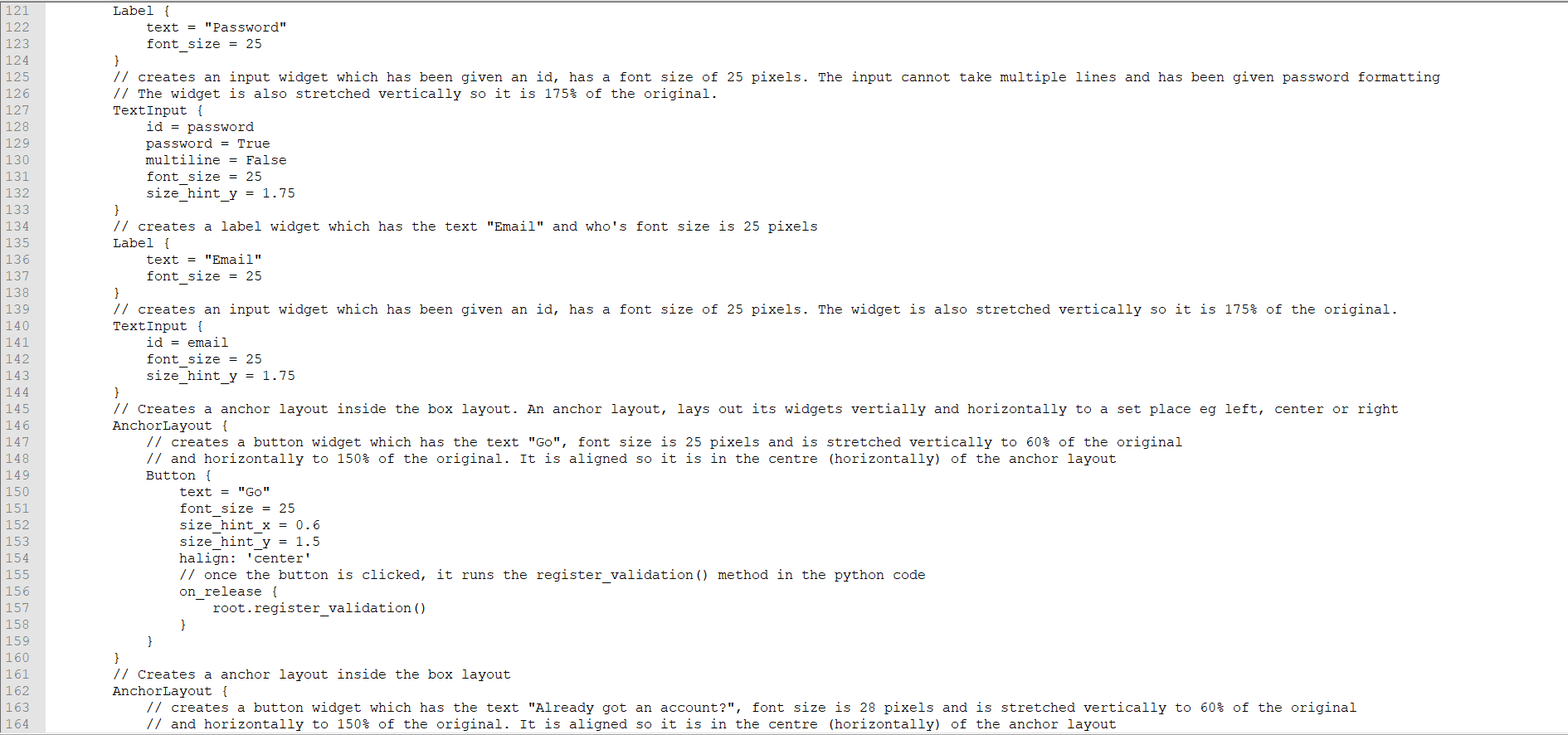
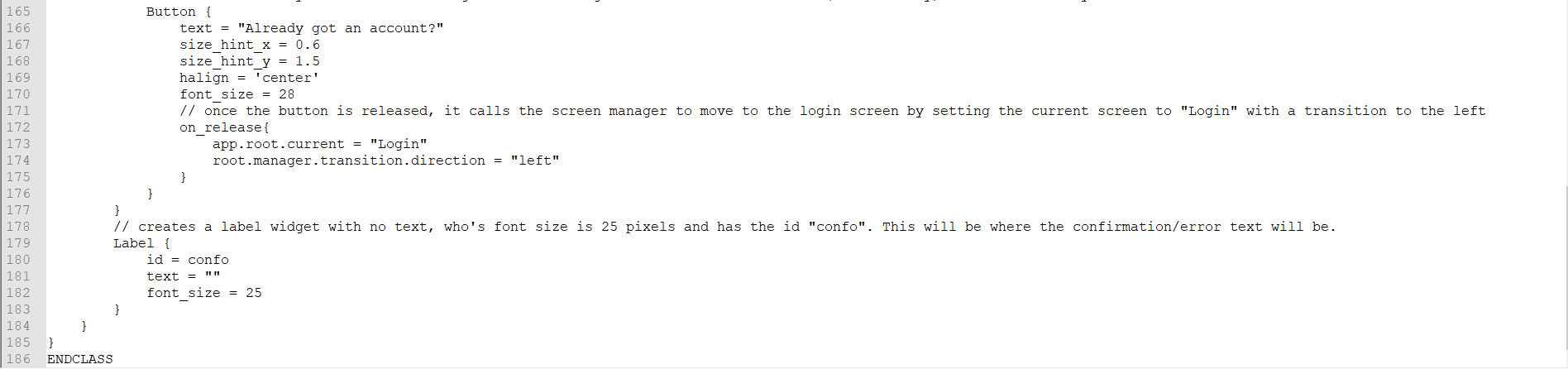
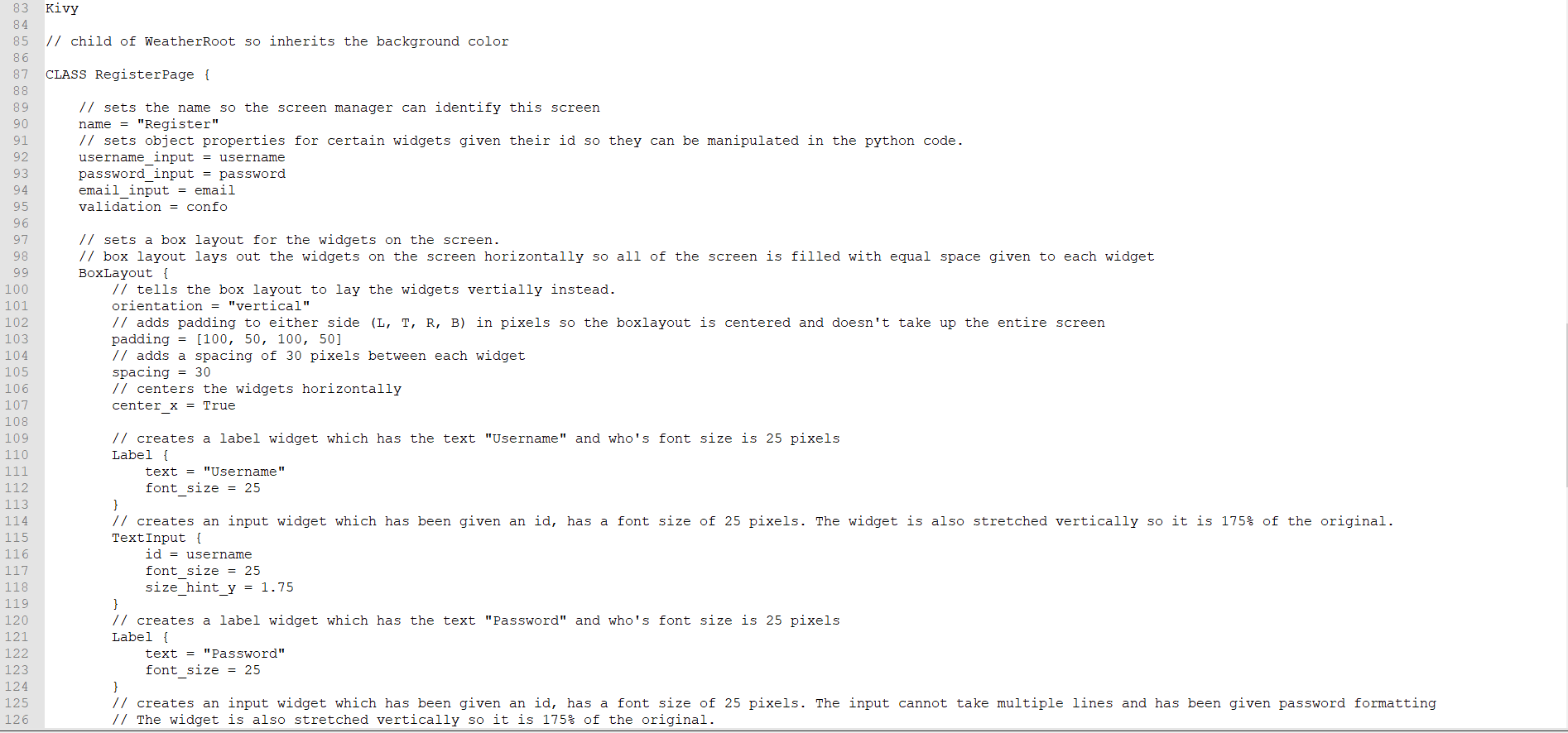
|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Explanation/Usage | Link to success criteria |
| register\_validation | procedure | The procedure for validating the details. Will return error message or will move to register procedure | 1 |
| register | procedure | Registers the user to the data source (adds the user details there). | 1 |
| user\_input | ObjectProperty | Input from the form which will be added to the DB/JSON file. It is also used for validation | 1,2 |
| password\_input | ^ | Input from the form which will be added to the DB/JSON file. | 1 |
| Email\_input | ^ | ^ as well as validation | 1,2 |
| validation | ^ | This is used to tell the user of success/failure messages | 1,5,6 |
| data | List | This is where the DB/JSON files data will be stored then will be edited with the new user’s program then added to the data source. It will also be searched through | 1, 4 |
| idList | Dictionary | Will be iterated over for validation (so we iterate over all existing users) | 1,2 |
| passHash | String | The hash (with the salt) for the user. | 1,3 |
| f | JSON file | What we open our file as so we dump our data | 1,4 |

What it should look like:



CHANGE SCREENSHOT ^ Pseudocode



PUT IN TEST DATA

Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Test number | What are we testing for | Expected result | Test data |
| 1 | Check usage of the text boxes and to make sure nothing is missing | When input text into the boxes, it should be presented clearly and be correctly sized. | Just some random strings |
| 2 | To see if the password formatting works | Whatever text we put in the password box; it should be replaced with asterisks. | Random string |

|  |  |  |  |
| --- | --- | --- | --- |
| Test number | What are we testing for | Expected result | Test data |
| 3 | Test validation process, no username | Expect to see “Form not completed” show up in the confirmation text area and for no registration to continue. | Random strings entered for password and email, nothing in username input box. |
| 4 | Test validation process, no password | ^ | Random strings entered for username and email, noting in password input box. |
| 5 | Test validation process, no email | ^ | Random strings entered for password and username, noting in email input box. |
| 6 | Test validation, already existing user | “Username or email is already taken” warning message. | Username = “f”, password and email have random strings |
| 7 | Testing validation, existing email | ^ | Email is “[h@h.h](mailto:h@h.h)”, password and username are random strings. |
| 8 | Test with legitimate data which hasn’t been used | It should register the user and there should be a confirmation message for the user. | Username = “re”  Password = random string  Email = “re@re.re” |

Post Test Data

## Part 4 - LoginPage Class (1.2)

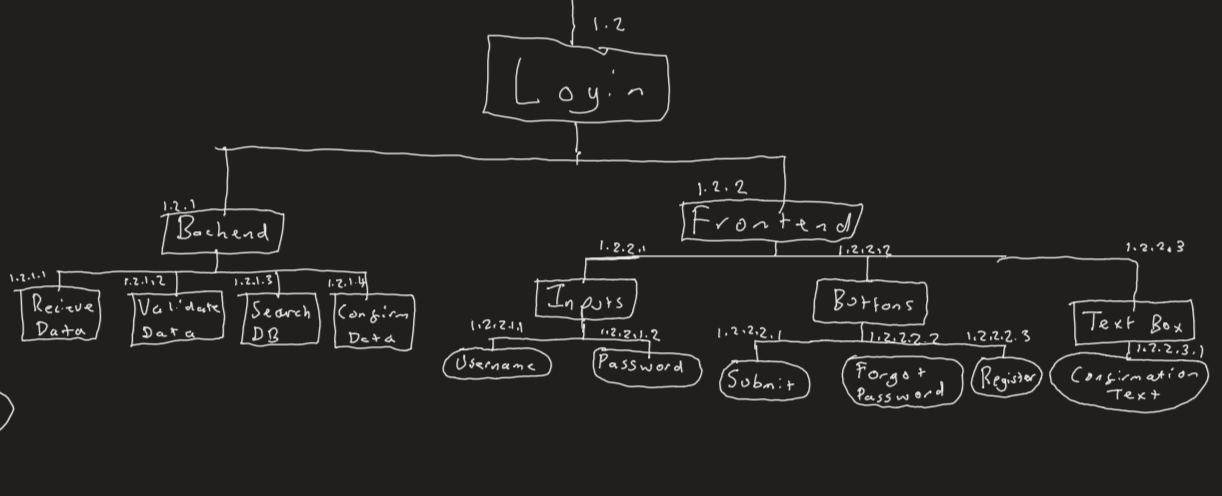
Description

The login page class also consists of a frontend (with Kivy code) and a backend (with python code ofc). The frontend consists of a box layout also but this time its structure is a bit different. This time, there are only 2 label/ input boxes: for username and password, each with their own id’s and have been declared as object properties. Then there is an anchor layout for the Login/Go (PLS CHOOSE ONE OF THESE) button so it can be aligned in the centre using the halign property; once clicked, it will run the backend code. Next, there is an embedded box layout that contains the forgot password and register buttons this is used because we can stack these 2 buttons horizontally and box layout is the best for stacking widgets. The forgot password DOES THIS and the register button moves the user to the register screen (1.1). Finally, there is the confirmation text which works the exact same as in 1.1.

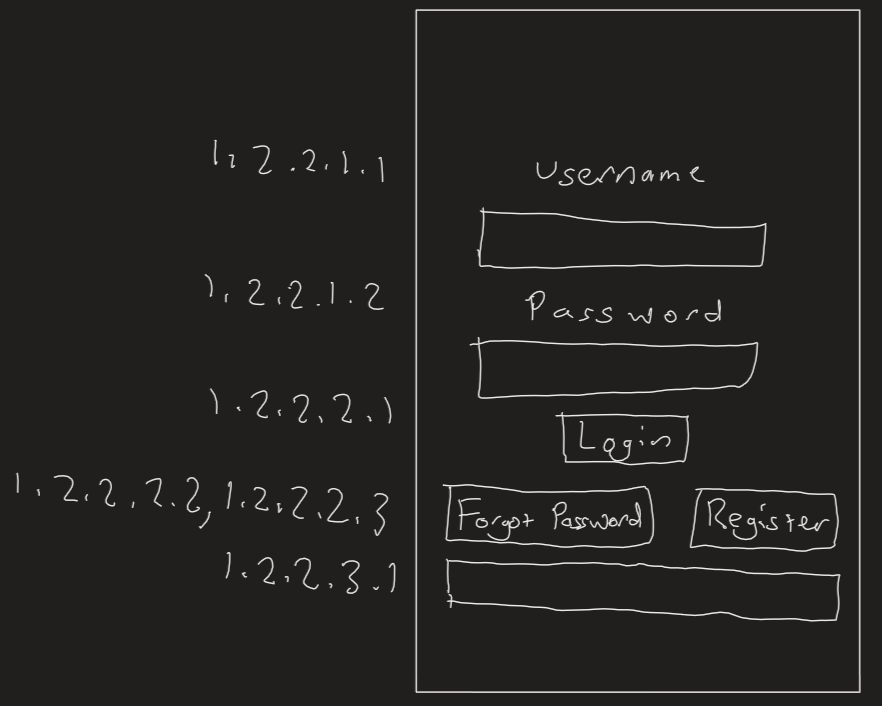
The backend would firstly, get the object properties from the Kivy code using varname = ObjectProperty(), this is done for the username input, password input, and validation text. Next, there is a procedure that will extract the data from the JSON file/ DB into a dictionary. This is then iterated over to create a list of id’s and users (which has the user info in it). Next, it would validate by checking if the form is complete and if the user exists in the list. If not then an appropriate message is set as the validation text object property. If it passes the tests then it is moved to the next procedure

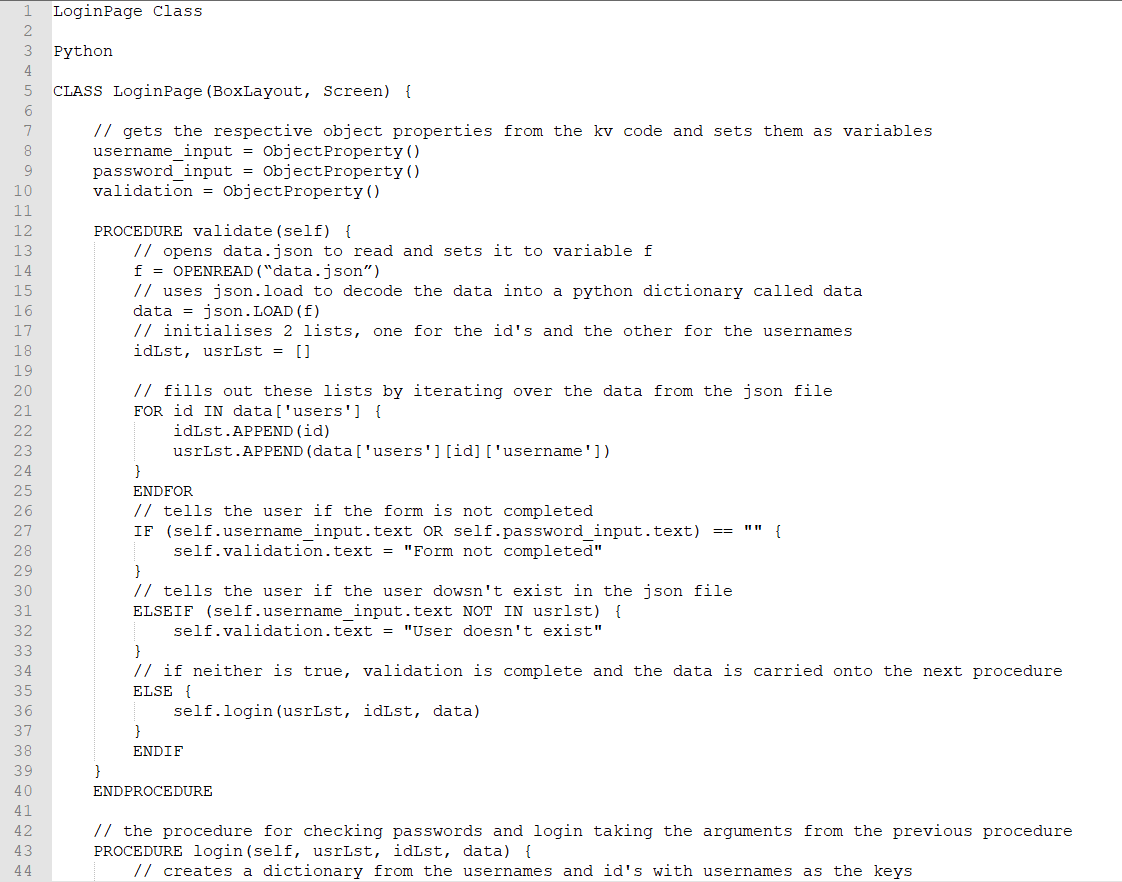
There will then be another procedure which opens the user’s info based on the username input and sets it to a dictionary. Next, we create a password hash using the same salt + password input method as in 1.1. Then we check this against the password hash in the database. If they don’t match then the validation text is set saying that the password input is wrong. If they match then the user is moved to the AddLocationForm class (1.3).

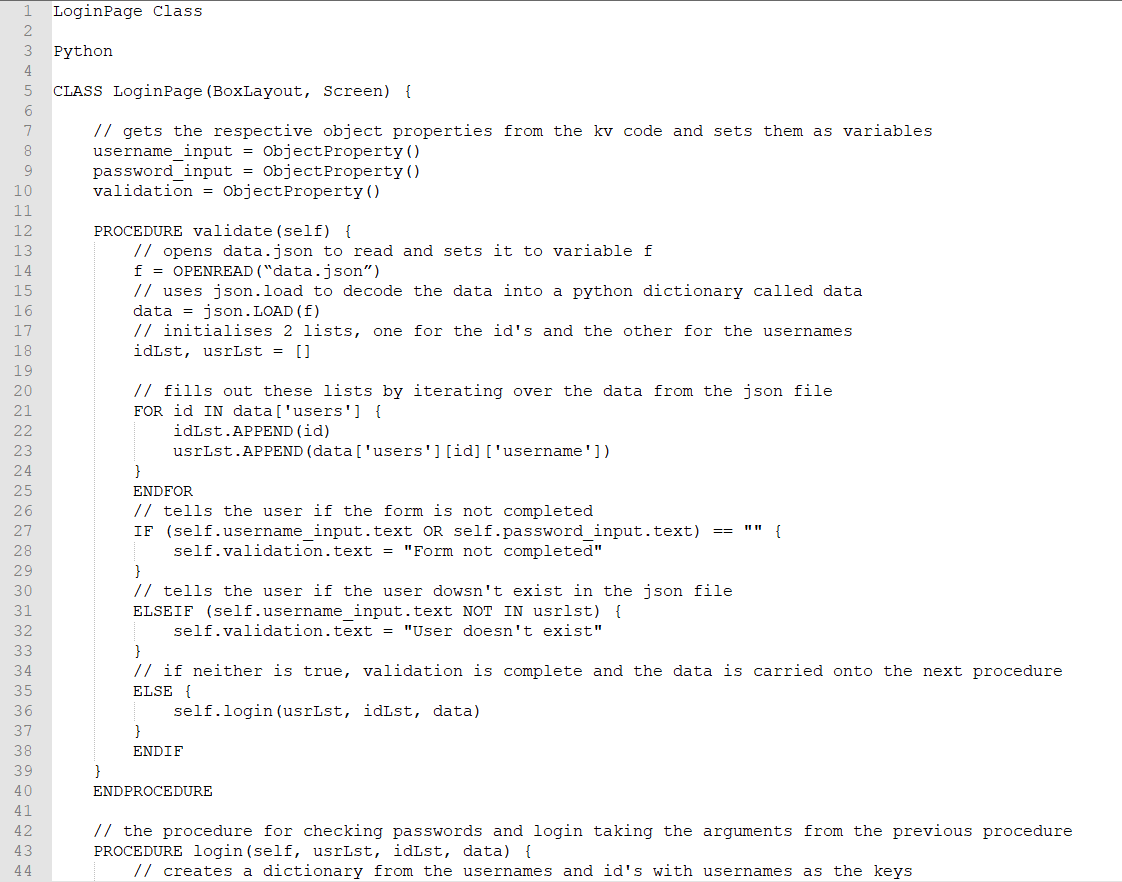
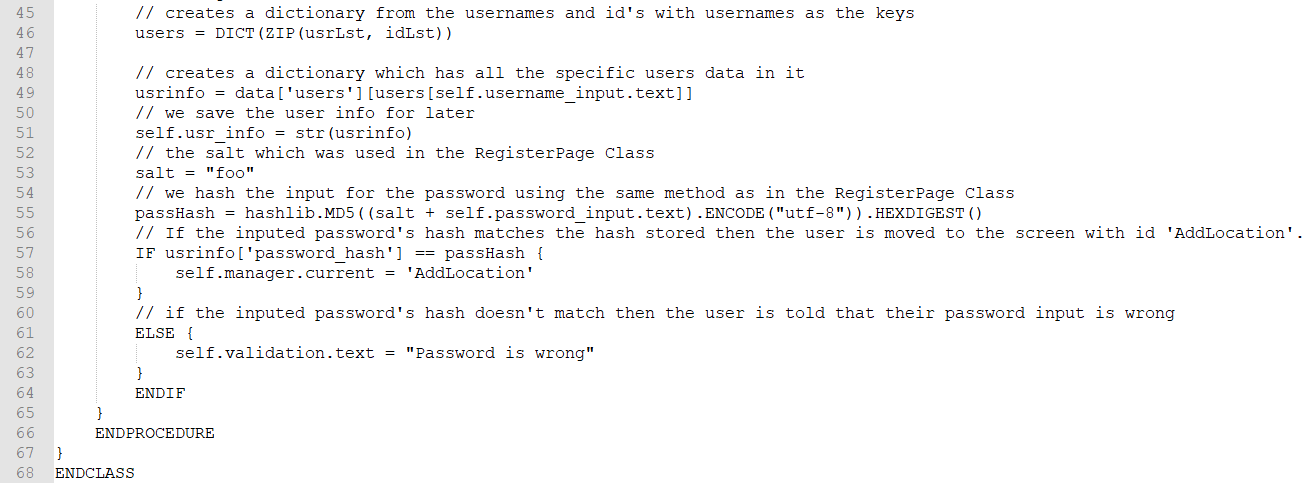
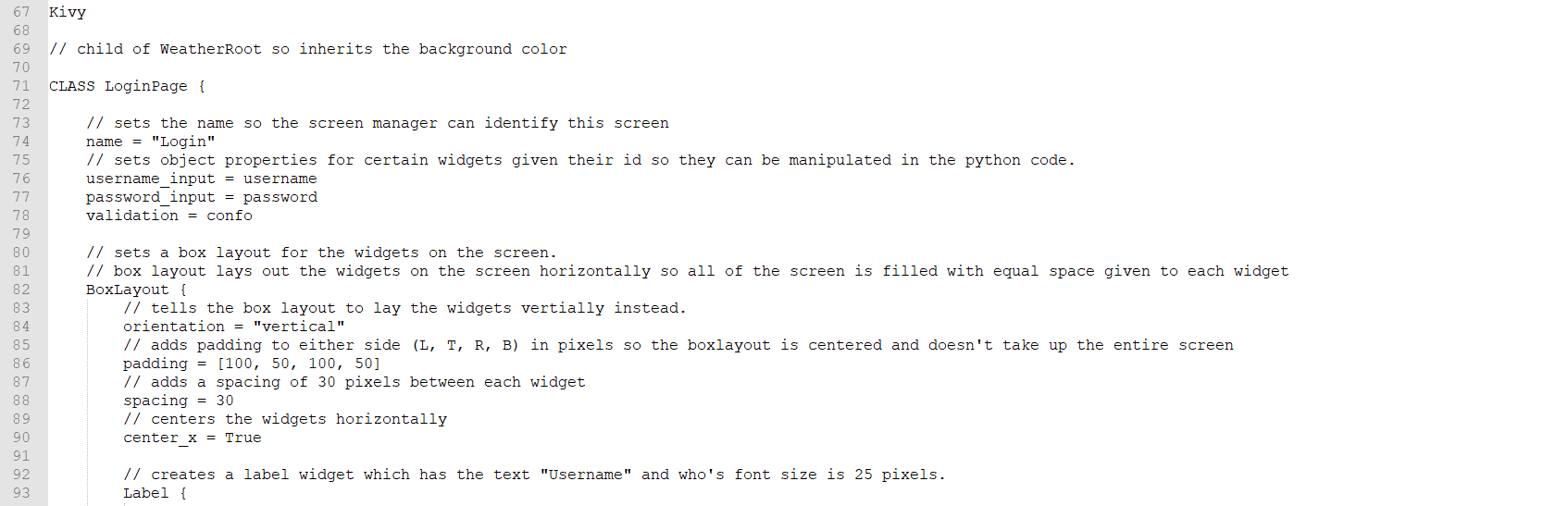
Structure Diagram

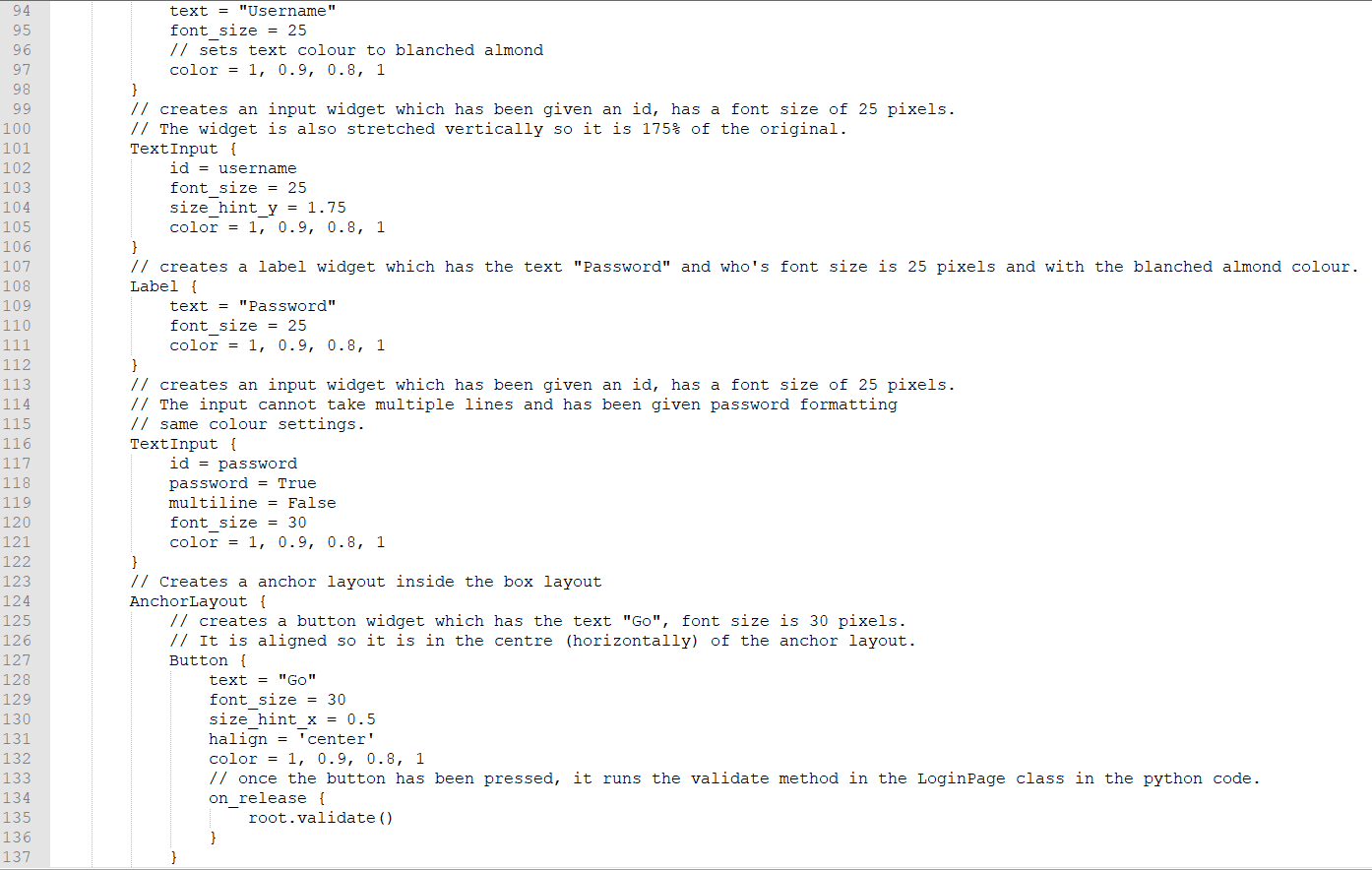
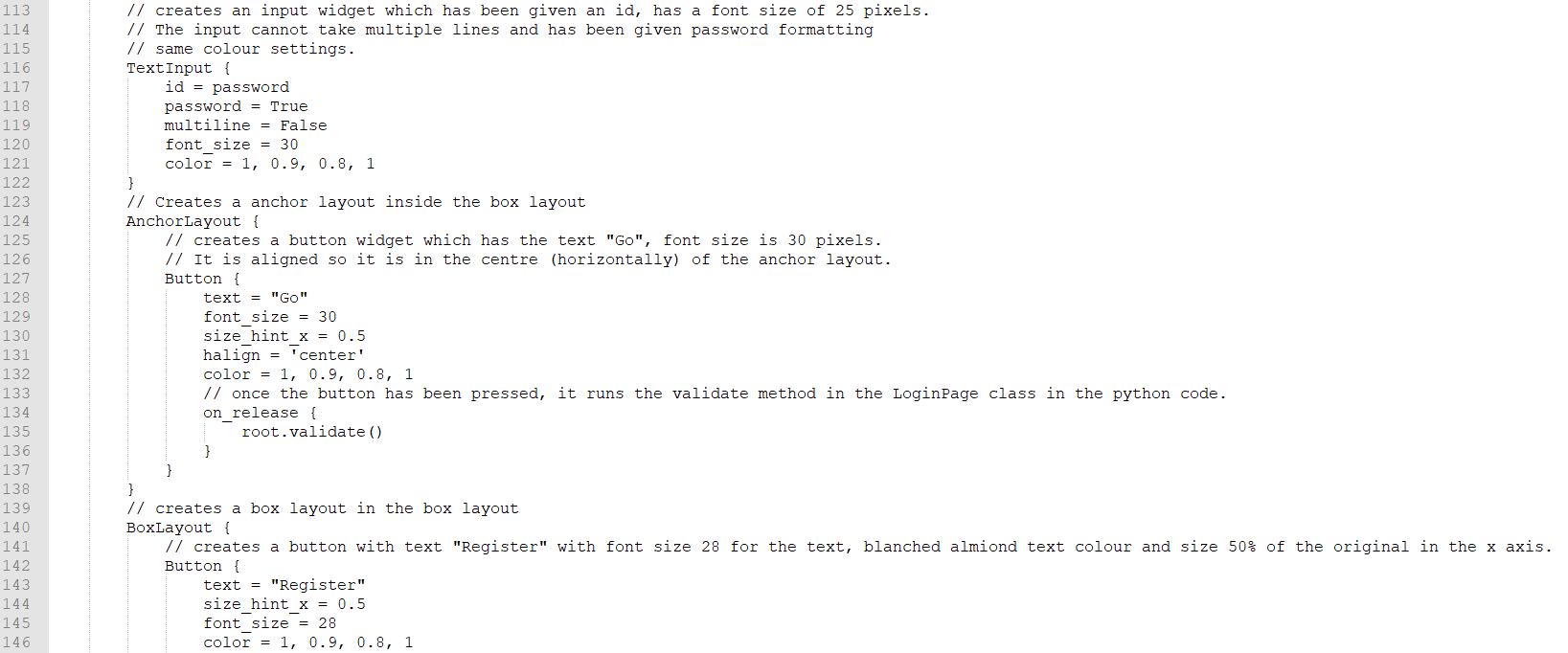


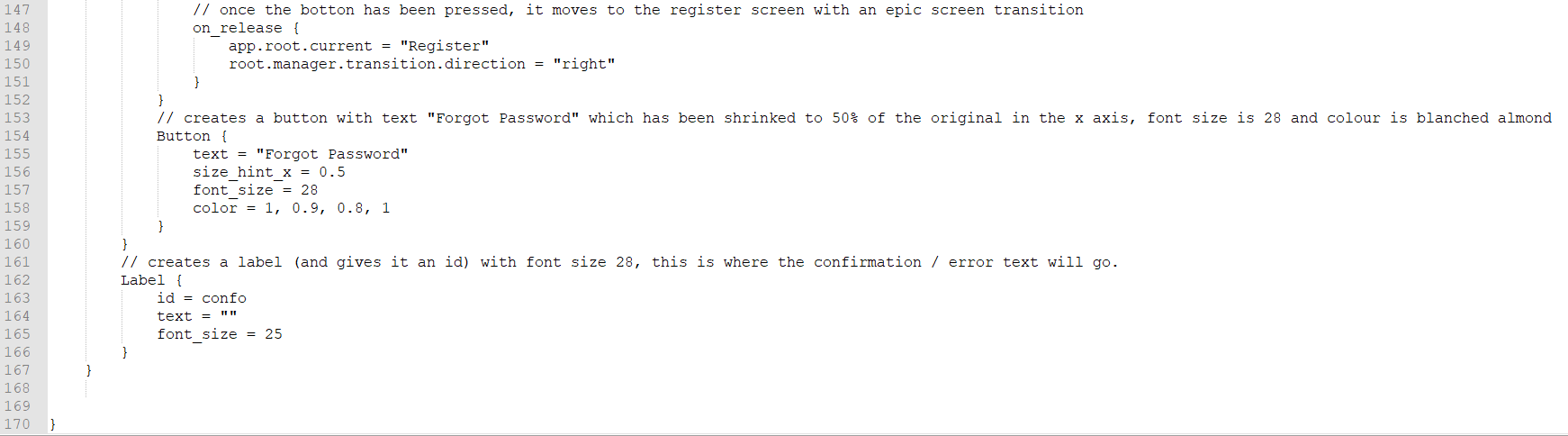
|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Explanation/Usage | Link to success criteria |
| validate | procedure | The procedure for validating the details. Will return error message or will move to register procedure | 7 |
| login | procedure | Registers the user to the data source (adds the user details there). | 7 |
| user\_input | ObjectProperty | Input from the form which will be used for validation | 7,8 |
| password\_input | ^ | Input from the form which will be used for validation | 7,8,9 |
| validation | ^ | This is used to tell the user of success/failure messages | 7,8 |
| Data | List | This will be searched through to validate the user's data. | 7, 8 |
| idList | Dictionary | Will be iterated over for validation (so we iterate over all existing users) | 7,8 |
| passHash | String | The hash (with the salt) to match with the data. | 7,8,9 |
| f | JSON file | What we open our file as so we dump our data | 7 |
| Usr\_data | ObjectProperty | This is where the user’s data is stored for future usage | 7, 12 |

Screen Design

Pseudocode





Test Data

Post Data

## Part 5 - AddLocationForm (1.3)

This is the main part of the program., where we get the METAR reports. Just like the other classes, it will consist of a frontend and a backend. The frontend will consist of a title at the top which will be centred in the middle horizontally. Underneath that, there will be a horizontally orientated box layout. This will consist of:

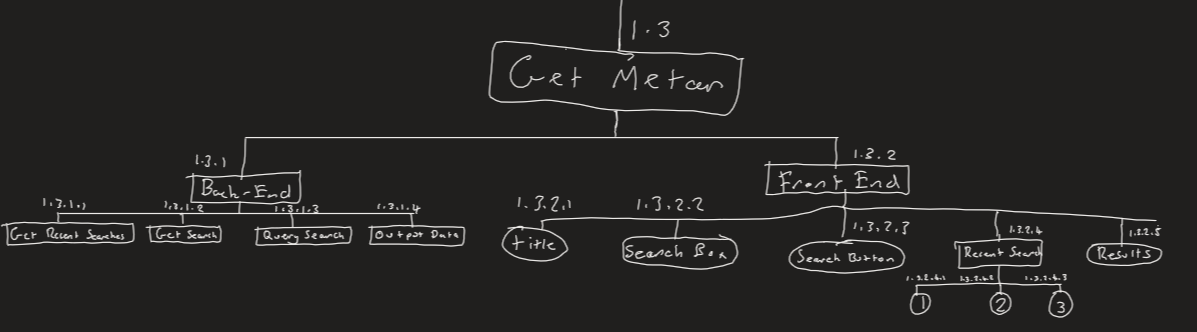
* A search box, which will be id’s so its contents can be retrieved in the backend.
* A search button, to trigger the backend code for searching.
* A recent search button which will open a dropdown box directly below it. See below
* Forgot the ICAO button which takes the user to 1.4.

The dropdown is id’d and only opens when the recent search button is opened. If the screen elsewhere is clicked then it closes (so it doesn’t stay opened permanently). It has 3 buttons inside. Once the recent search button is clicked, a backend script will run which collects the backend scripts from the user’s data than will fit the 3 buttons with the recent searches as their text so when one of these 3 buttons are clicked, the search box contents changes to that of the button. The recent search system allows a pilot or enthusiast to quickly search for METAR for frequently flown routes hence helping our target market.

Finally, there is the big space where (for now) there will just be a ListView widget (which displays a list/ array in vertical order filling up the screen). This is where we will display the decoded METAR in the ListView way so each bit of the metar is stacked vertically like bullet points. In future versions, this can include such things as a map, runway information, etc.

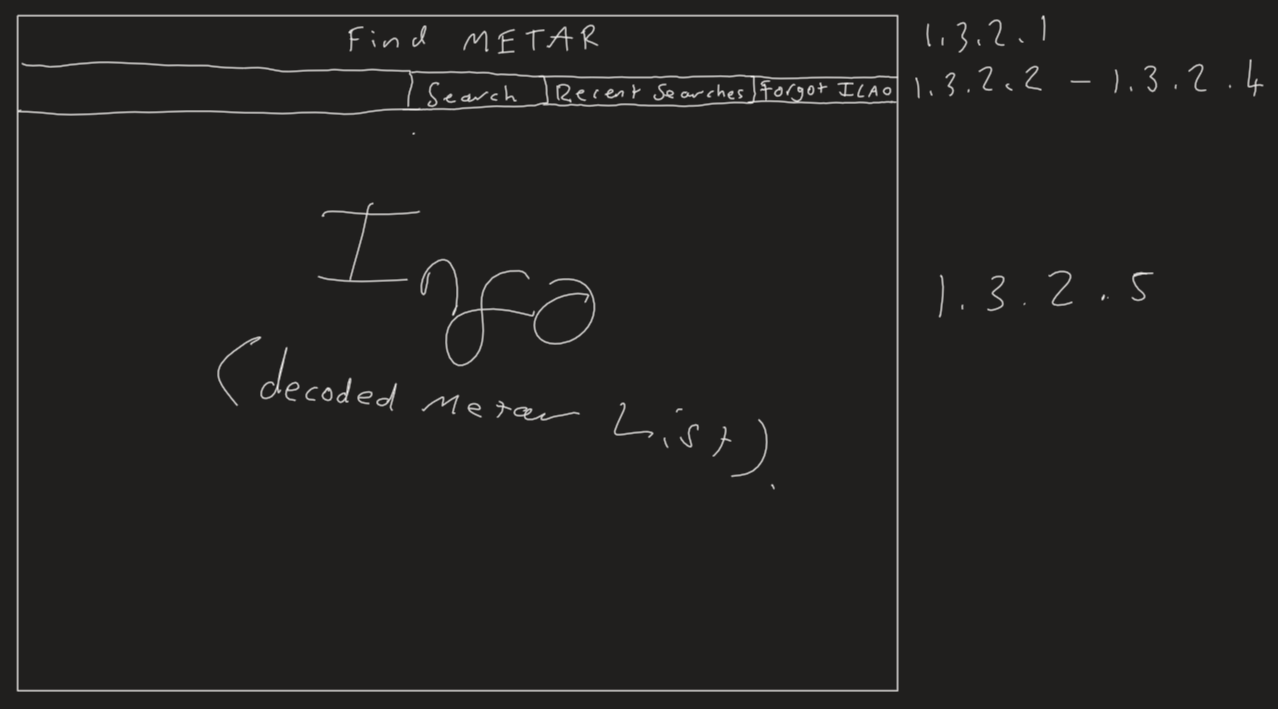
For the backend, there will be 3 procedures, one to fetch the recent searches and 2 to handle the search (fetch it, query it and output the results). The first one will get the users data from their login (saved a global variable), format it into a dictionary format then fill them in for the dropdown boxes.

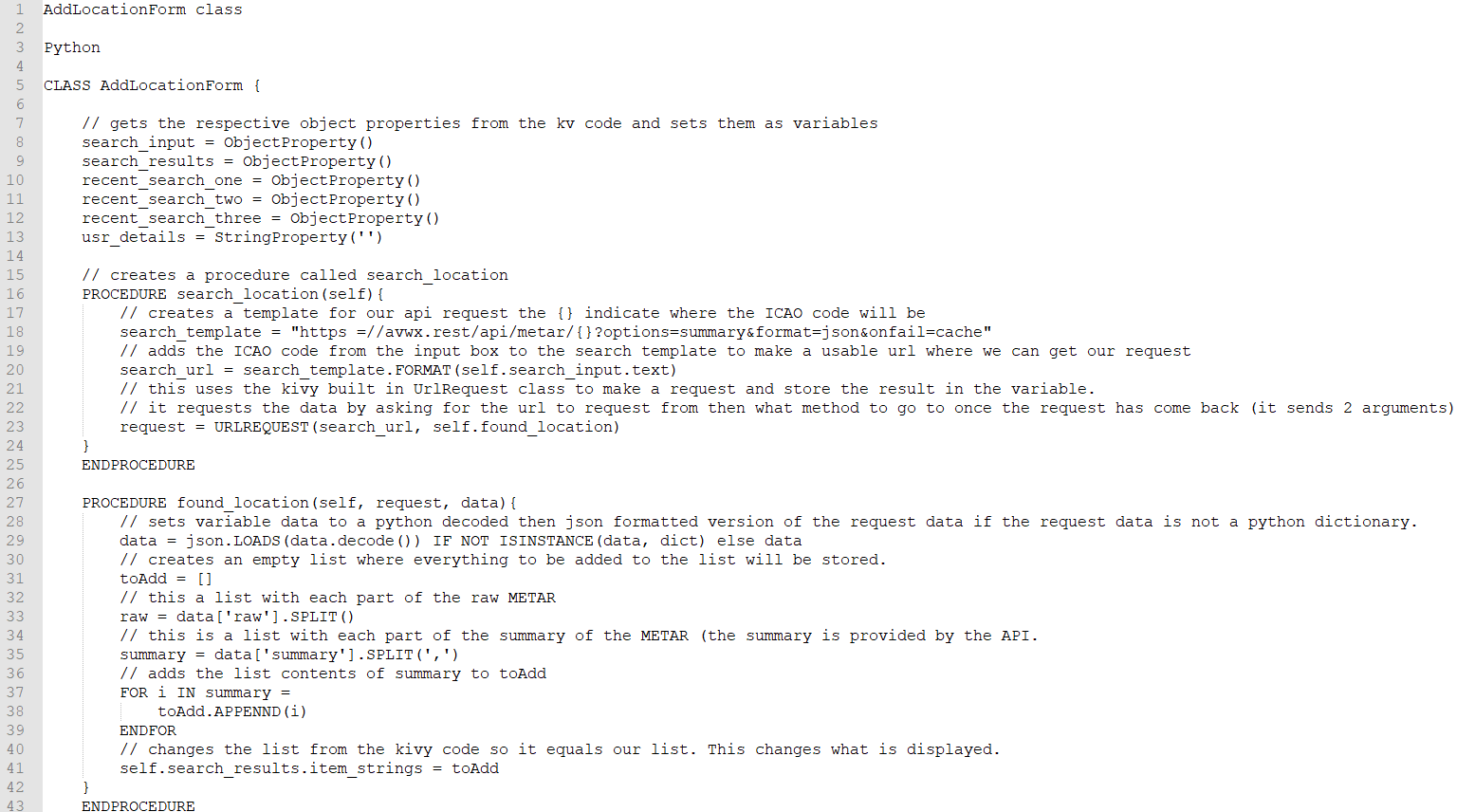
For the search, there will be 2 procedures, one to send the request and one which triggers the request once the request comes back (auto-triggered with Kivy’s URL request class). The first will put the search request in an already defined template then fire it off with the URL request class. The second will decode the request data using JSON.loads into a python dictionary. An empty list is created for the data to add, this is populated with the summary of the metar for now. Once done, it will update the list view widget on the screen.

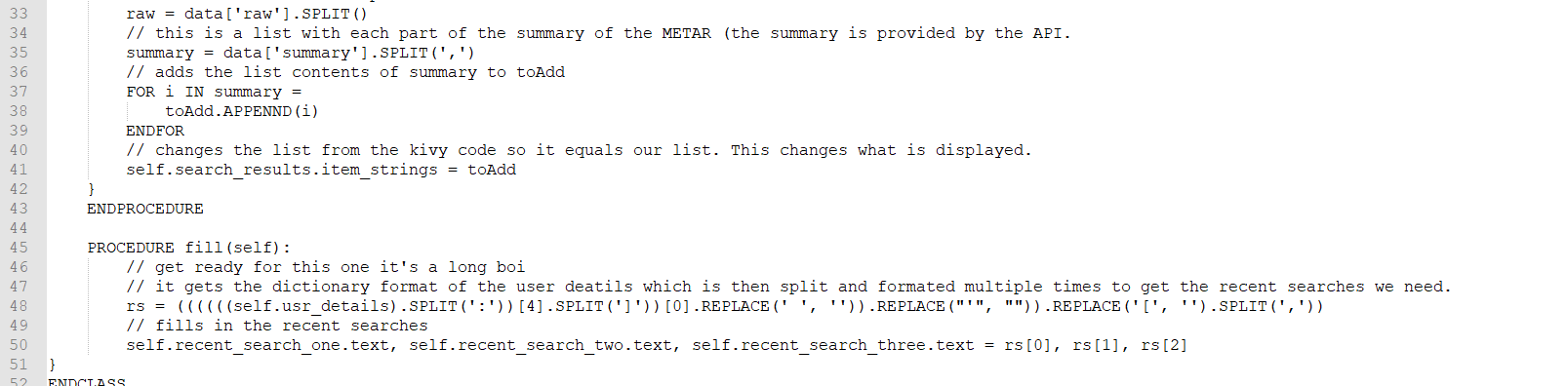
Structure Diagram

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Explanation/Usage | Link to success criteria |
| search\_location | procedure | This creates the template for the API request and moves to the next procedure when completed. | 13, 15, 17 |
| found\_location | procedure | Gives the user the details from the request | 13, 16 |
| fill | procedure | Fills in the contents of the recent search dropdown. | 14 |
| search\_input | ObjectProperty | The input from the search box used to search. | 17 |
| search\_results | ^ | The results from the search which will be displayed to the user | 16 |
| recent\_search\_one, recent\_search\_two, recent\_search\_three | ^ | The recent searches. Used for convenience. | 14 |
| usr\_details | StringProperty | The user’s details | 14 |
| Search\_url | String | The URL where we request our data from. | 15 |
| Data | Dictionary | The entire API request data | 15,16 |
| toAdd | List | What will be displayed to the user. By adding to the search\_results ObjectProperty. | 16 |
| rs | list | The recent searches | 14 |

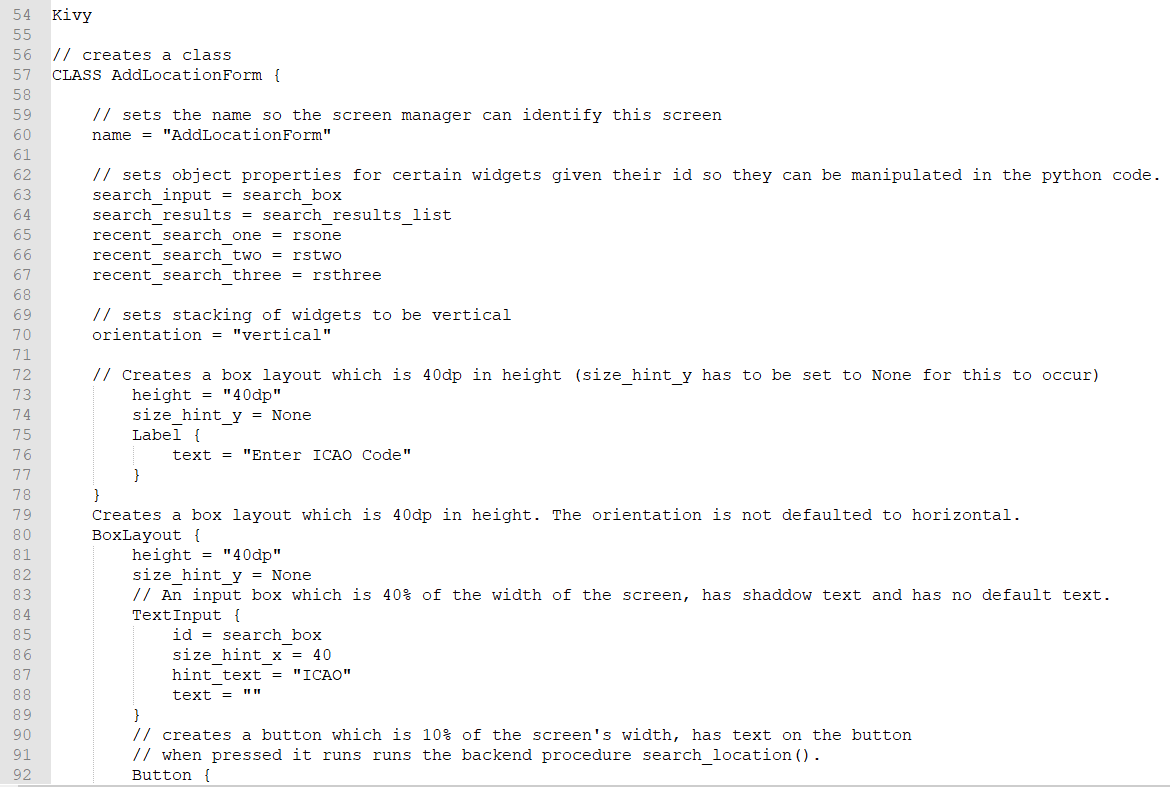
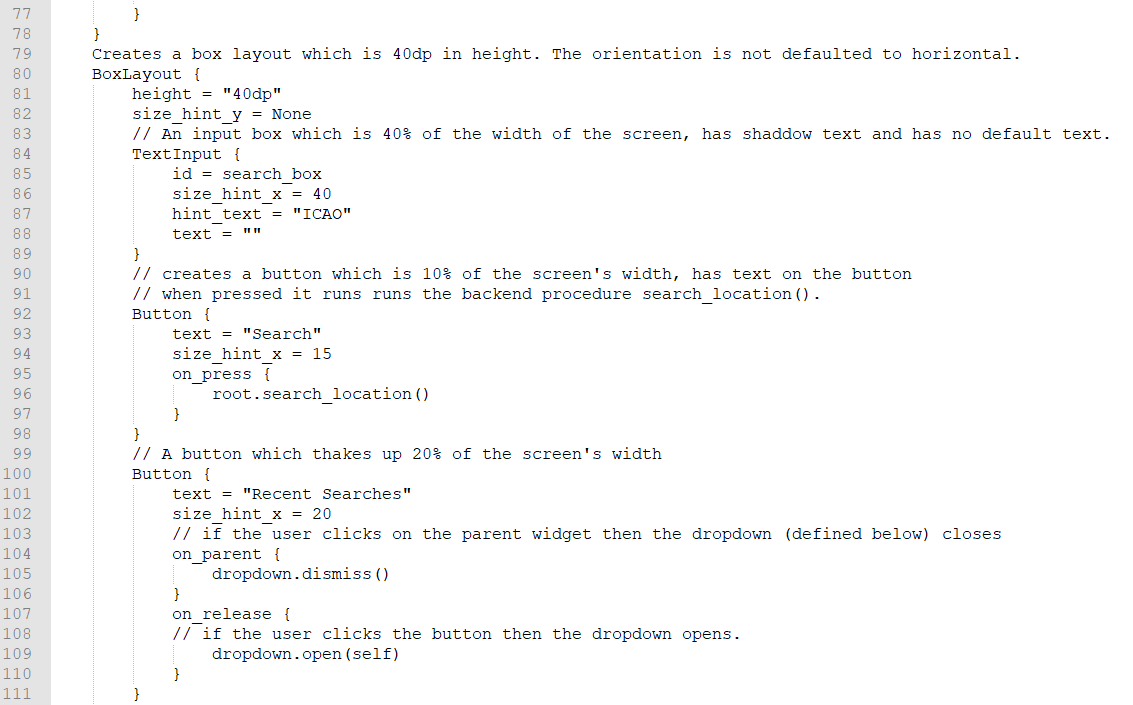
Screen Design

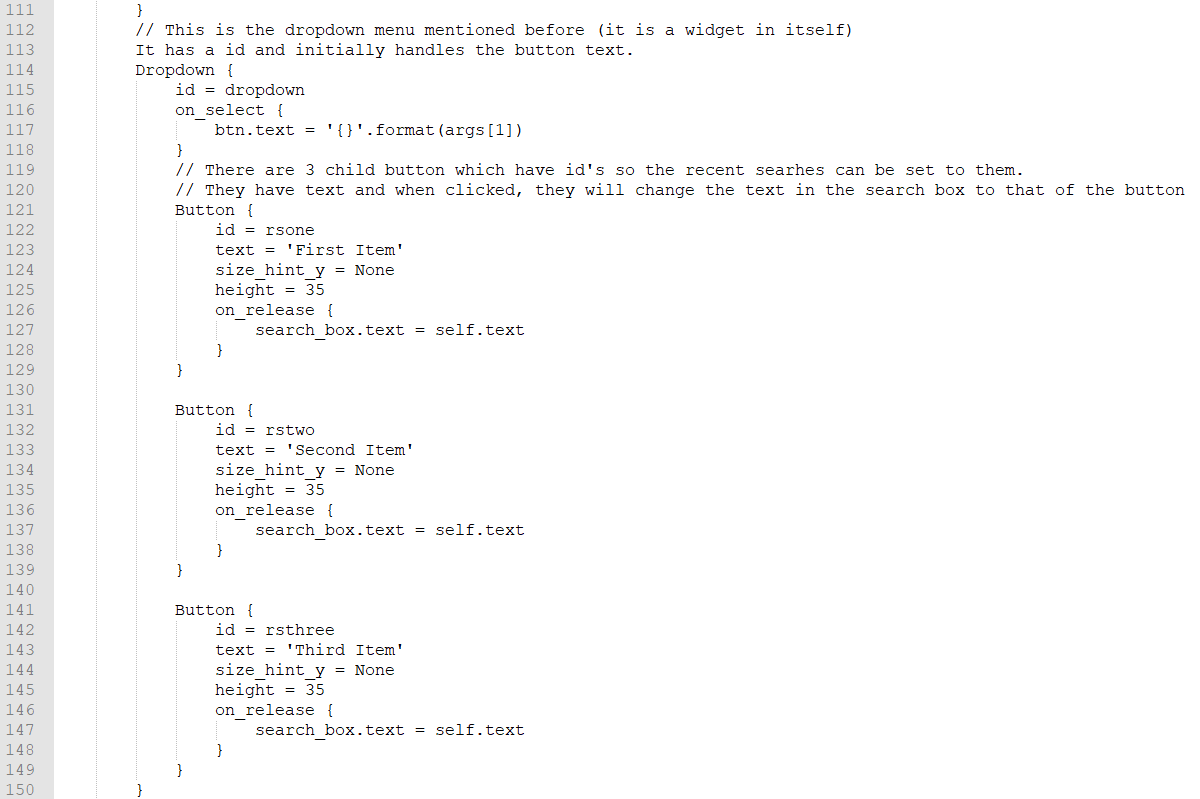
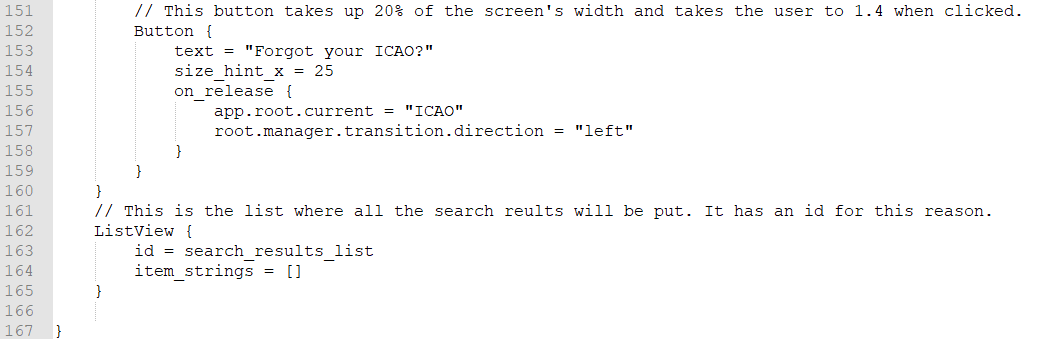


Pseudocode



Kivy





TEST DATA

## Part 5 – ICAOFinder

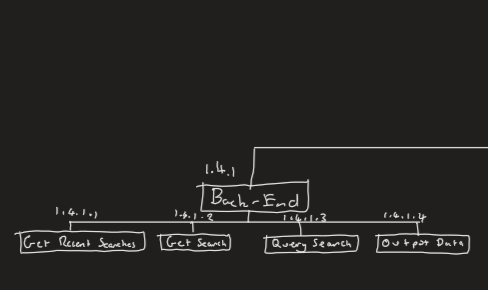
This part was added because of the demands of our clients as pilots and enthusiasts can forget their ICAO code so we thought of a system where the user could search by airport name and it would show the airports ICAO.

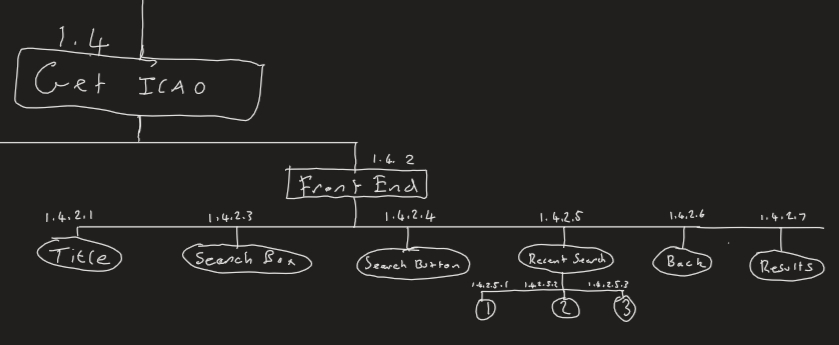
This app consists of a frontend and a backend (you’ve got the gist by now for sure). The frontend will consist of a title at the top which will be centred in the middle horizontally. Underneath that, there will be a horizontally orientated box layout. This will consist of:

* A search box, which will be id’s so its contents can be retrieved in the backend.
* A search button, to trigger the backend code for searching.
* Back button which takes the user to 1.3.

The backend will consist of 2 procedures, the first will get the recent searches from the global variable, format it and add it into the dropdown buttons. The second shall be triggered from the search button and it will get the search, query it and update the list view based on the results. The search system would get its data from an external CSV file which will be formatted to get the results in a dictionary then it would see if the query is in the key or not. If it is then the key and value are added to a list which will then be used to update the list view list.

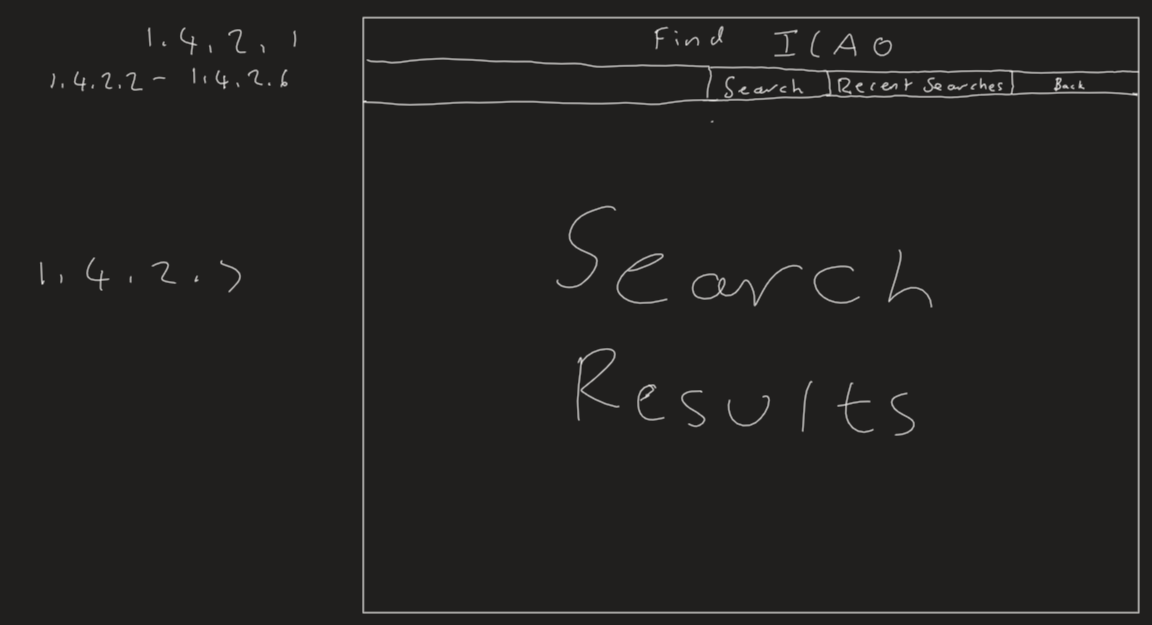
Structure diagram



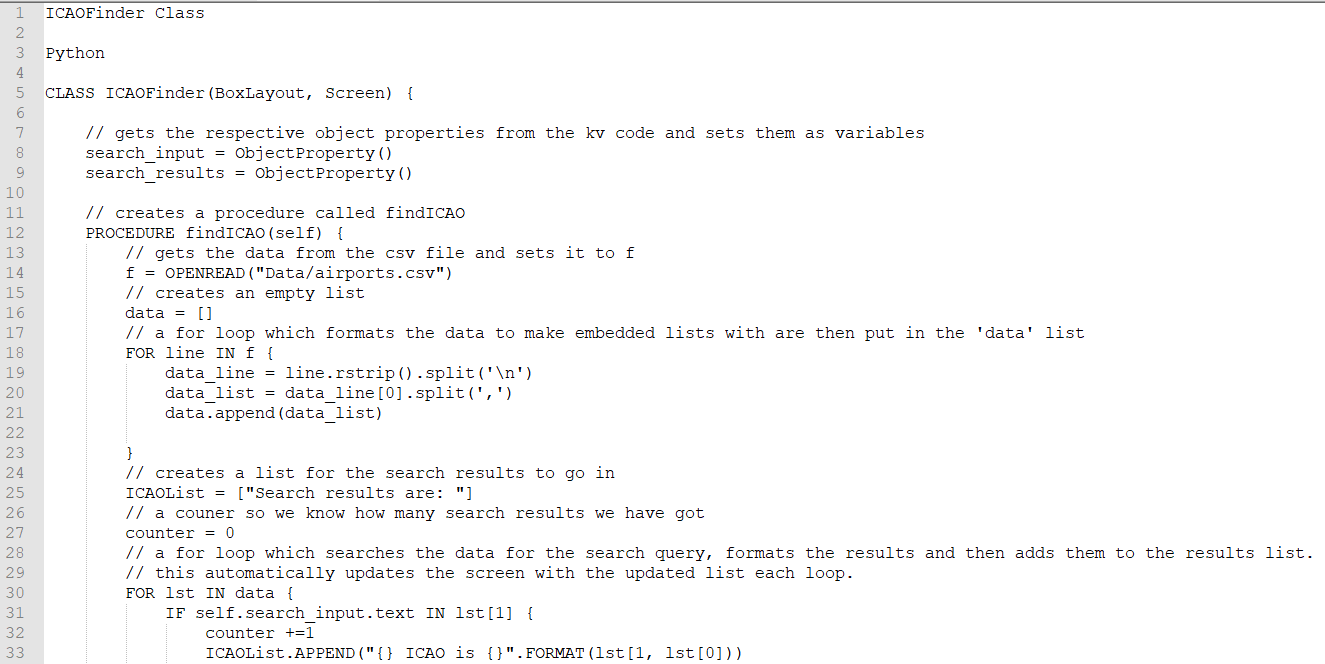


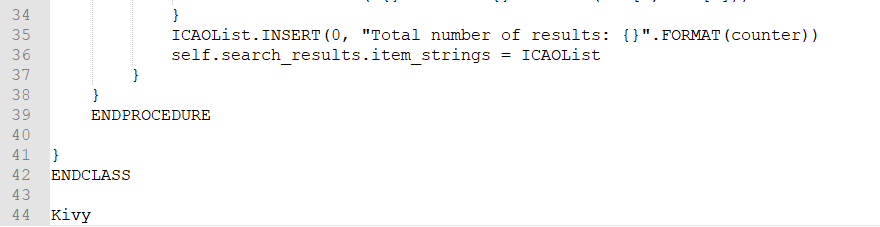
|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Explanation/Usage | Link to success criteria |
| findICAO | procedure | This creates the template for the API request and moves to the next procedure when completed. | 13, 15, 17 |
| search\_input | ObjectProperty | The input from the search box used to search. | 17 |
| search\_results | ^ | The results from the search which will be displayed to the user | 16 |
| Data | List | All the ICAO codes and their names | 15,16 |
| ICAOList | List | What will be displayed to the user. By adding to the search\_results ObjectProperty. | 16 |

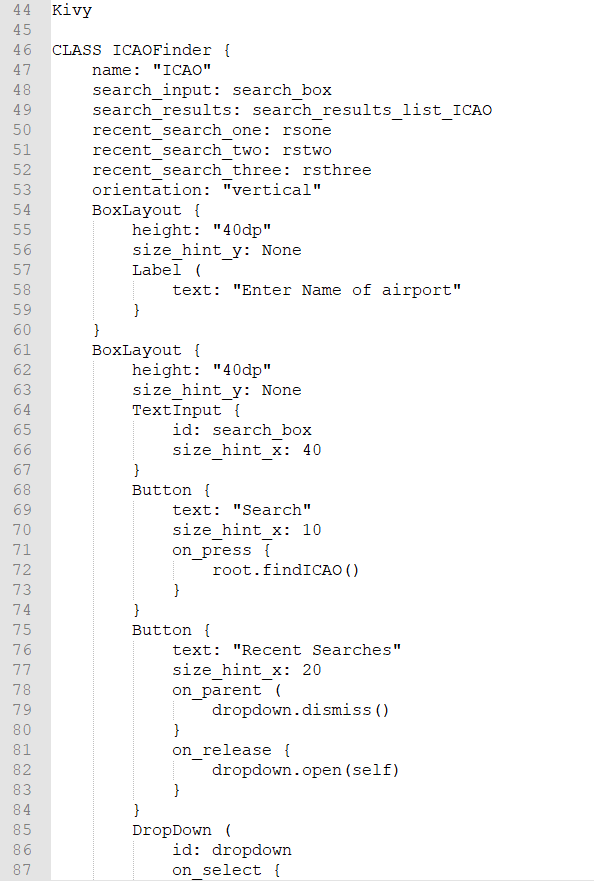
Screen Design

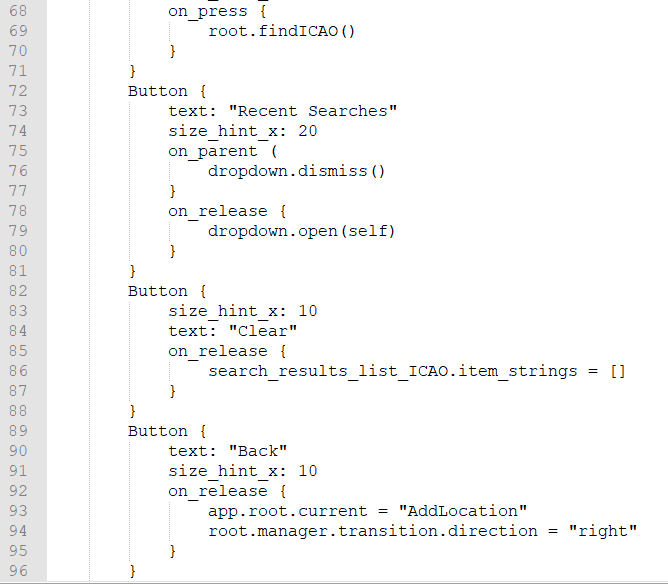


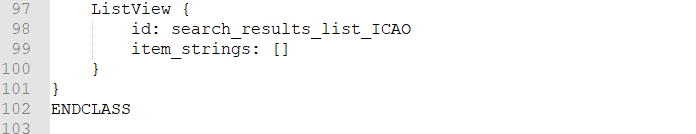
Pseudocode

Python



Kivy





Test Data

Post-Test Data

## Part 6 – Meeting with shareholders

I decided to meet with them separately, showed them this document and asked for their opinion and any questions they may have.

Joseph

Geoffrey