[Date]

Harin Vyas

El Generico®

[Document title]

[Document subtitle]

Contents

[Section 1 – Analysis 2](#_Toc19006085)

[Part 1 – Description of the problem 2](#_Toc19006086)

[Part 2 – Stakeholders 2](#_Toc19006087)

[Part 3 – How the problem can be solved by computational methods 3](#_Toc19006088)

[Section 2 – Design 4](#_Toc19006089)

[Part 1 - Introduction 4](#_Toc19006090)

[Part 2 - How Kivy works 4](#_Toc19006091)

[Notes 5](#_Toc19006092)

[Part 3 – RegisterPage Class (1.1) 5](#_Toc19006093)

[Part 4 - LoginPage Class (1.2) 10](#_Toc19006094)

[Part 5 - AddLocationForm (1.3) 14](#_Toc19006095)

[Part 5 – ICAOFinder 18](#_Toc19006096)

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 which 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:

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 less 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 colour-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 system.

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 in to 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 which will end and close the app when the user exists the app.

Thinking Concurrently

## Part 4 – Success Criteria

The app must have at minimum:

A login/register system which links to either a JSON file or a database which has users data in. The data saved will be their settings preferences, recent searches (3), username, email and password. Once the user has logged in, the user’s data is set as a global variable which then can be used in the other classes.

A way to search for METAR reports

# 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 visualise 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 to 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) which 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 in the sake of Kivy). The python code defines the screens, the root and app life cycle (through the grandparent class) and runs it all. The python code also preforms 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 which 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 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).

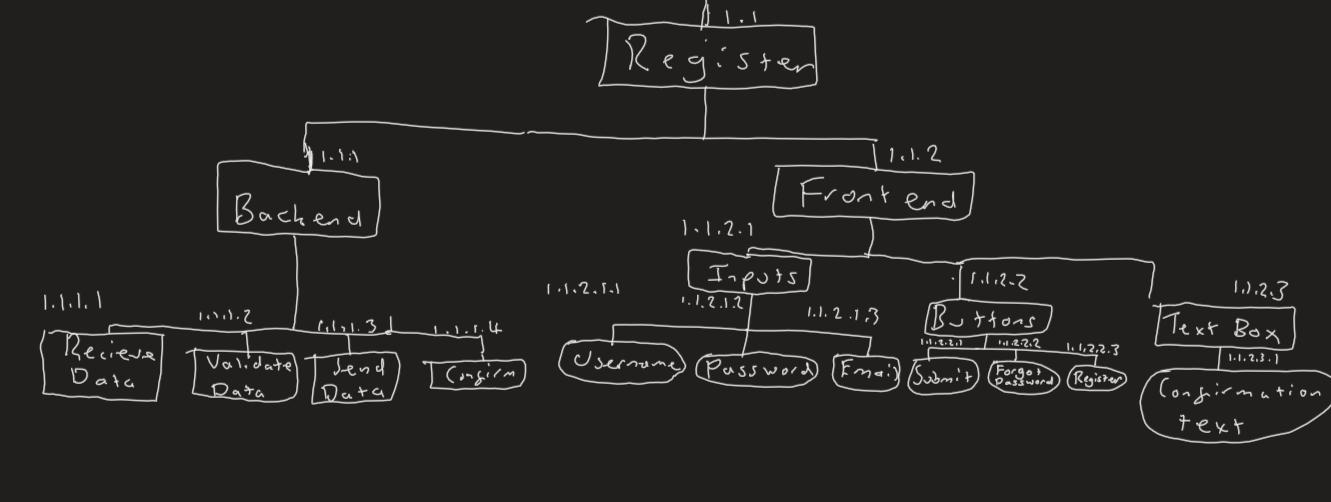
Next, there will be an anchor layout (who’s parent is the box layout) which has 2 buttons: submit which will trigger the backend code to run and login (CHANGE ON SCREENSHOT) which takes the user to the login page (1.2) when clicked. Finally, there is the confirmation text which is a label with an id 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 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 which 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. MAYBE ADD RED COLOUR FONT. If 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 a 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:

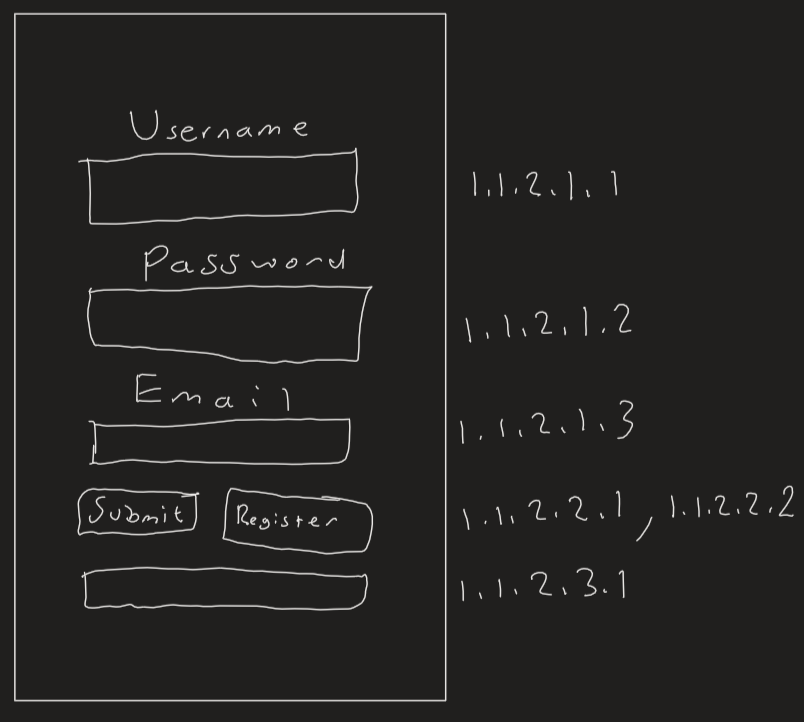
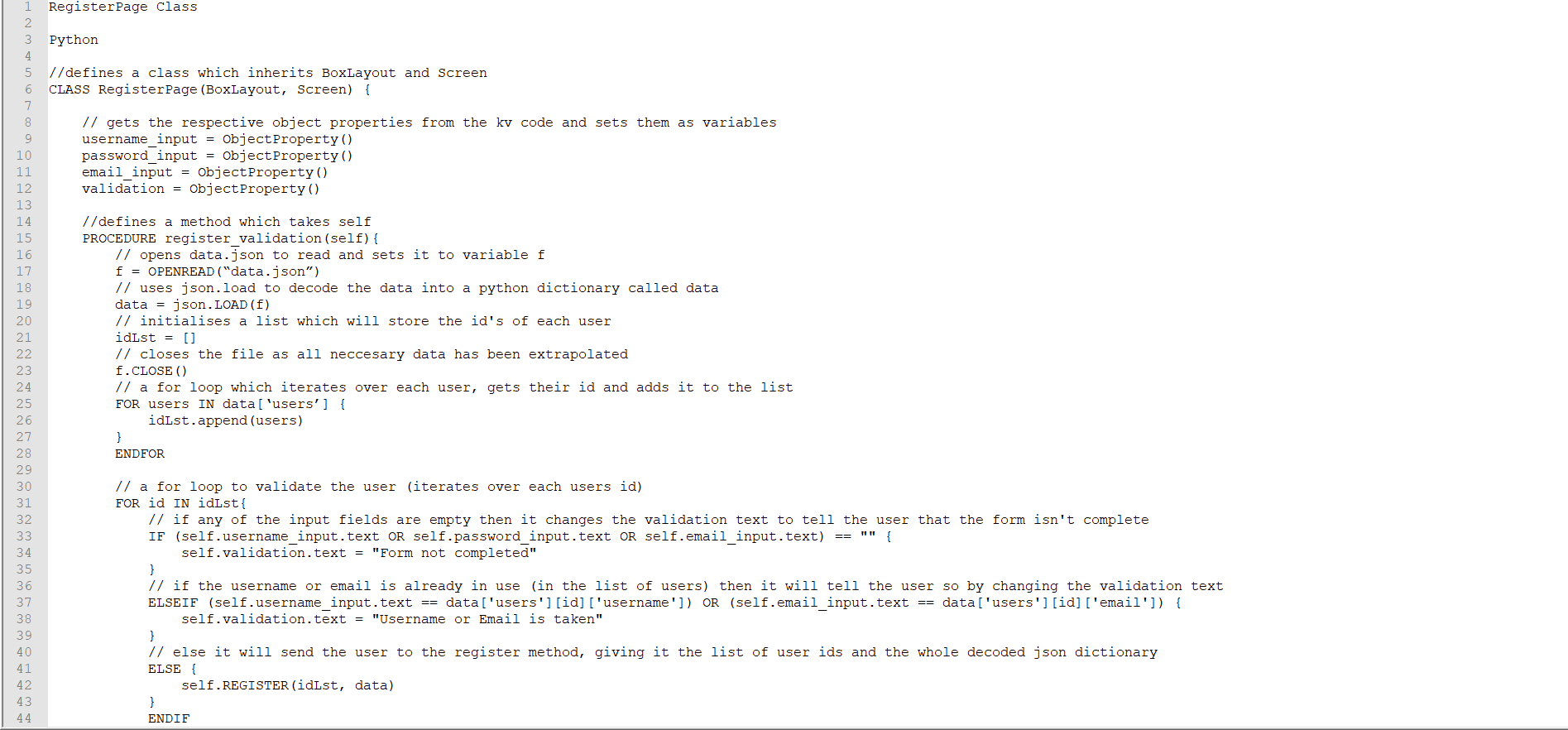
For JSON, update the extracted data with the new user’s data (id, username, password, email, etc.), in the correct formatting. Then the json file will be closed, then this data will be dumped to a new temporary file, the old one deleted and the temporary file renamed to the old one’s name using the os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

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

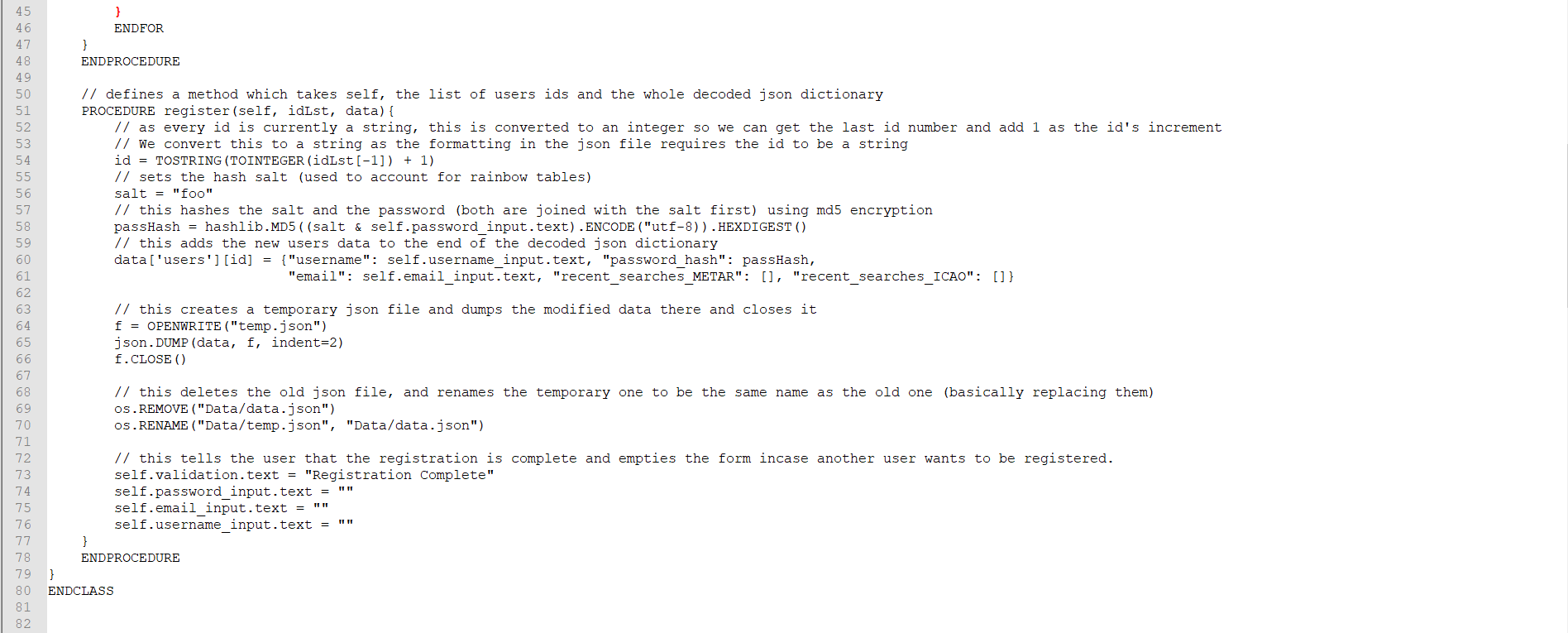
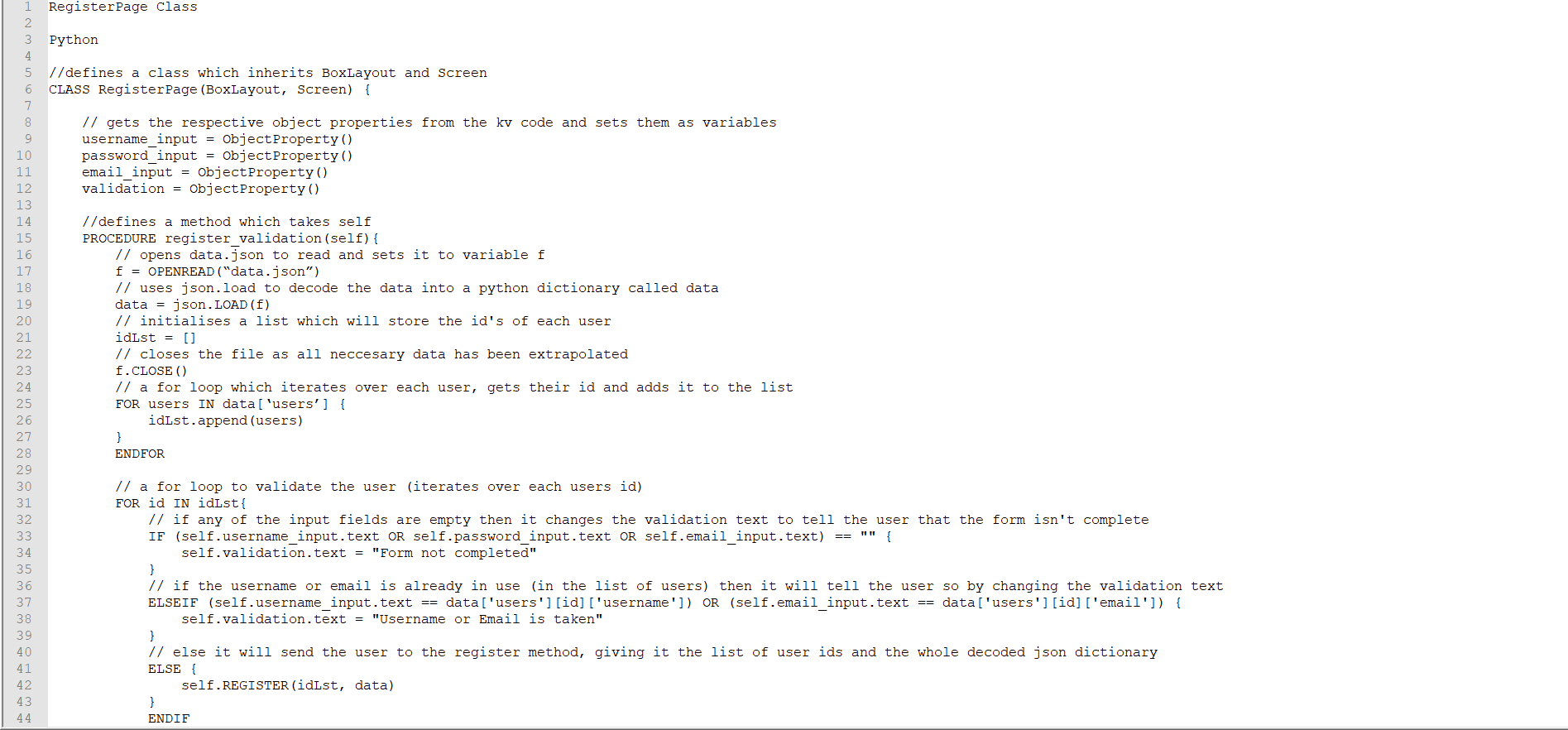
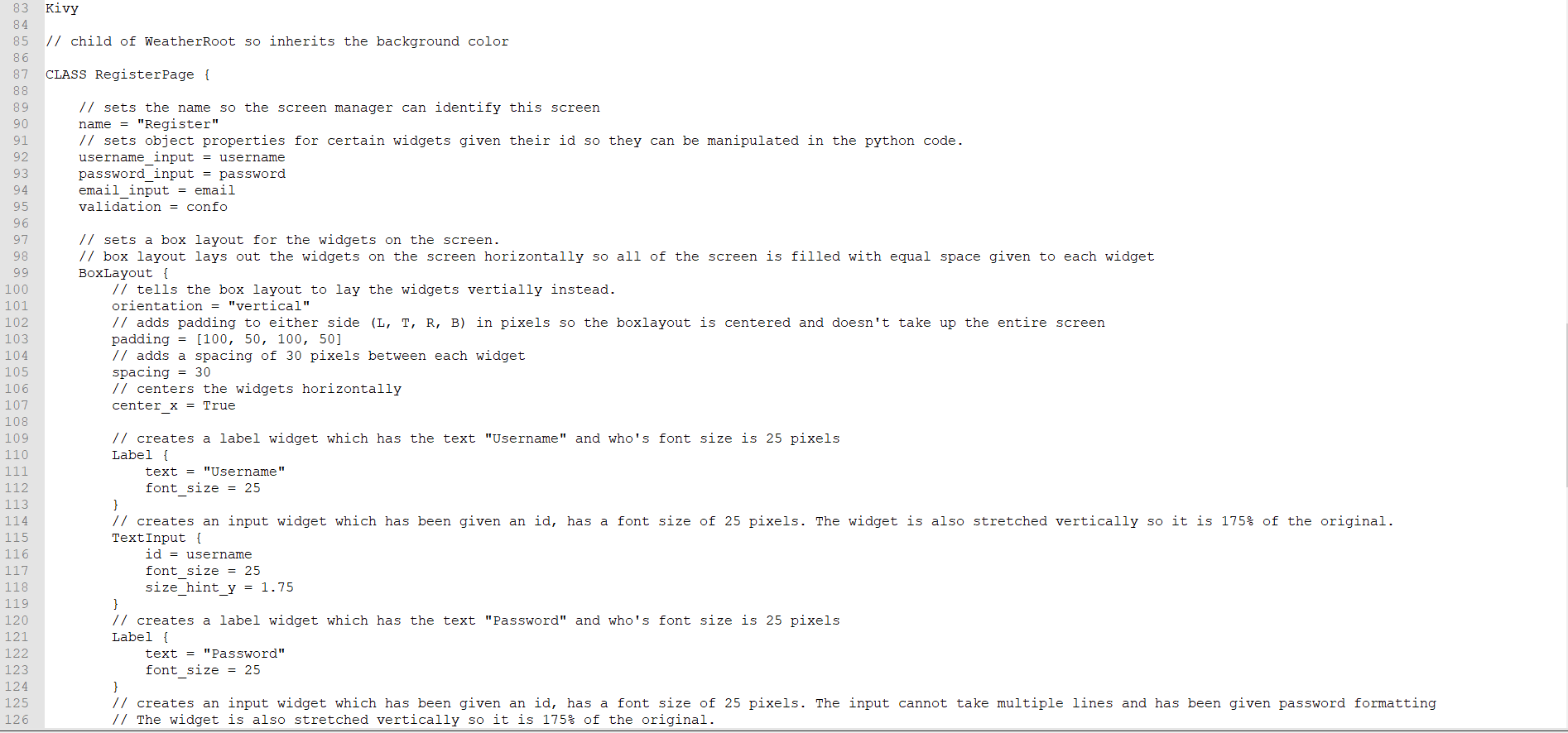
Either way, at the end the code, the input boxes will be cleared and the validation text object property updated with a success message. MAYBE IN GREEN COLOUR FONT.

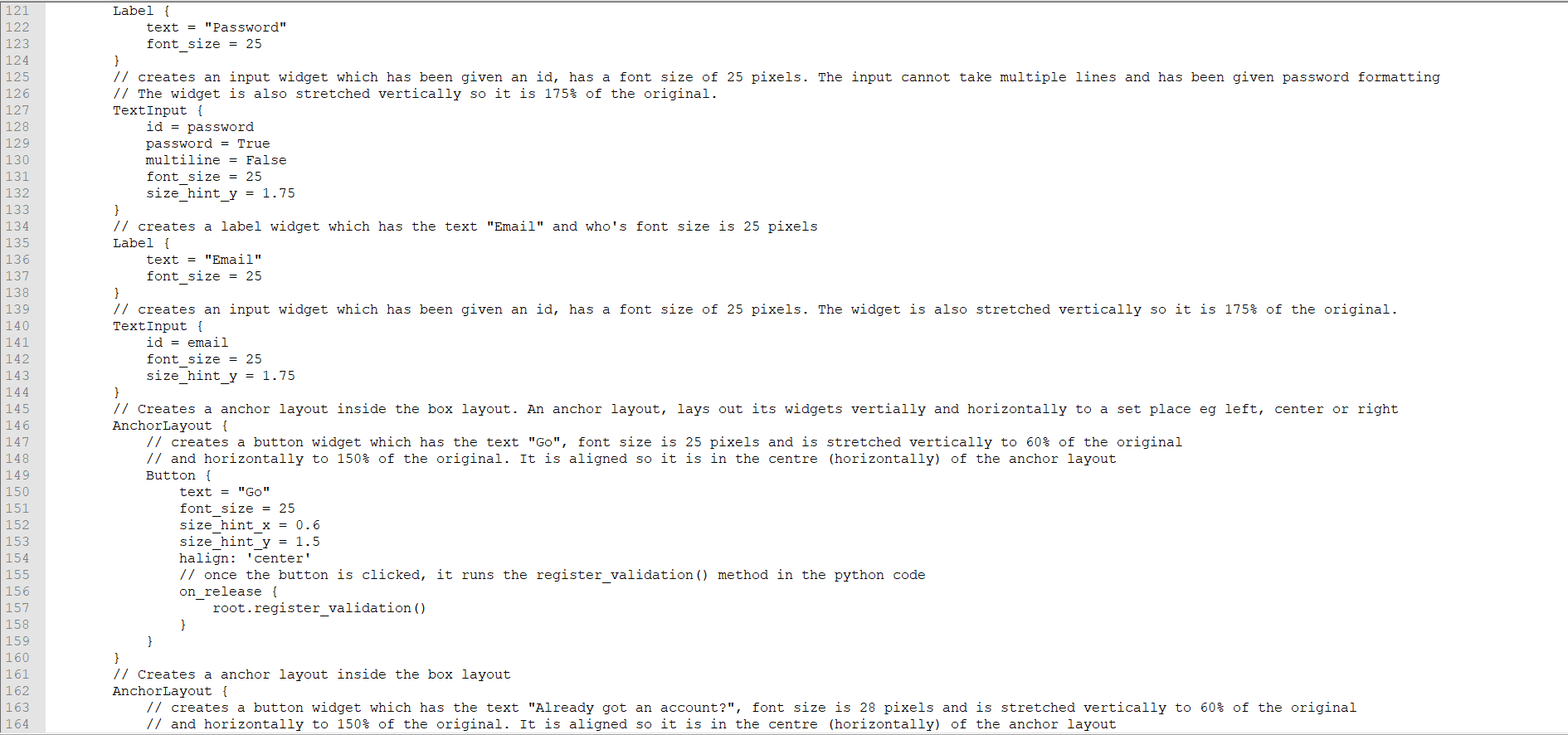
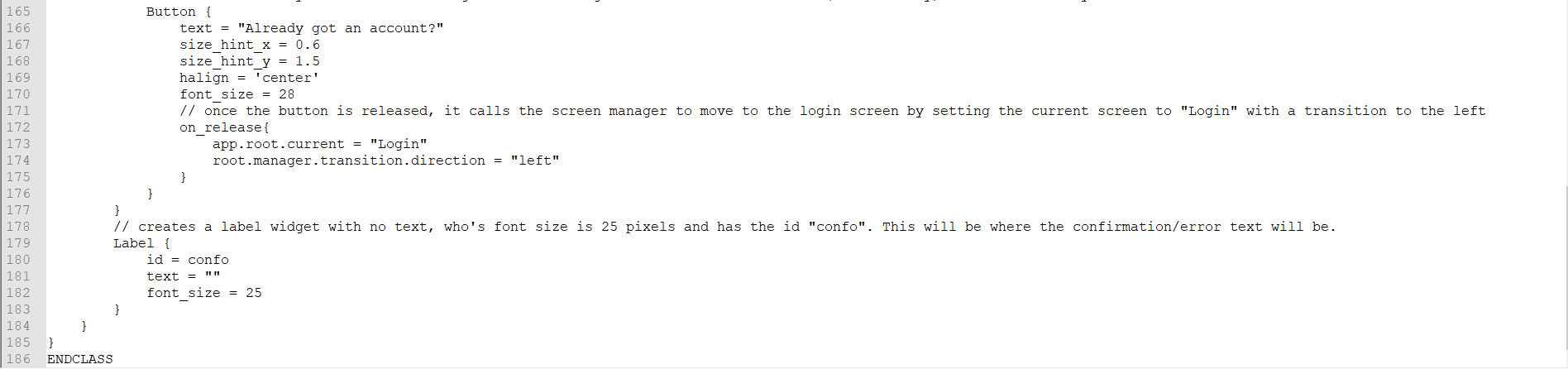
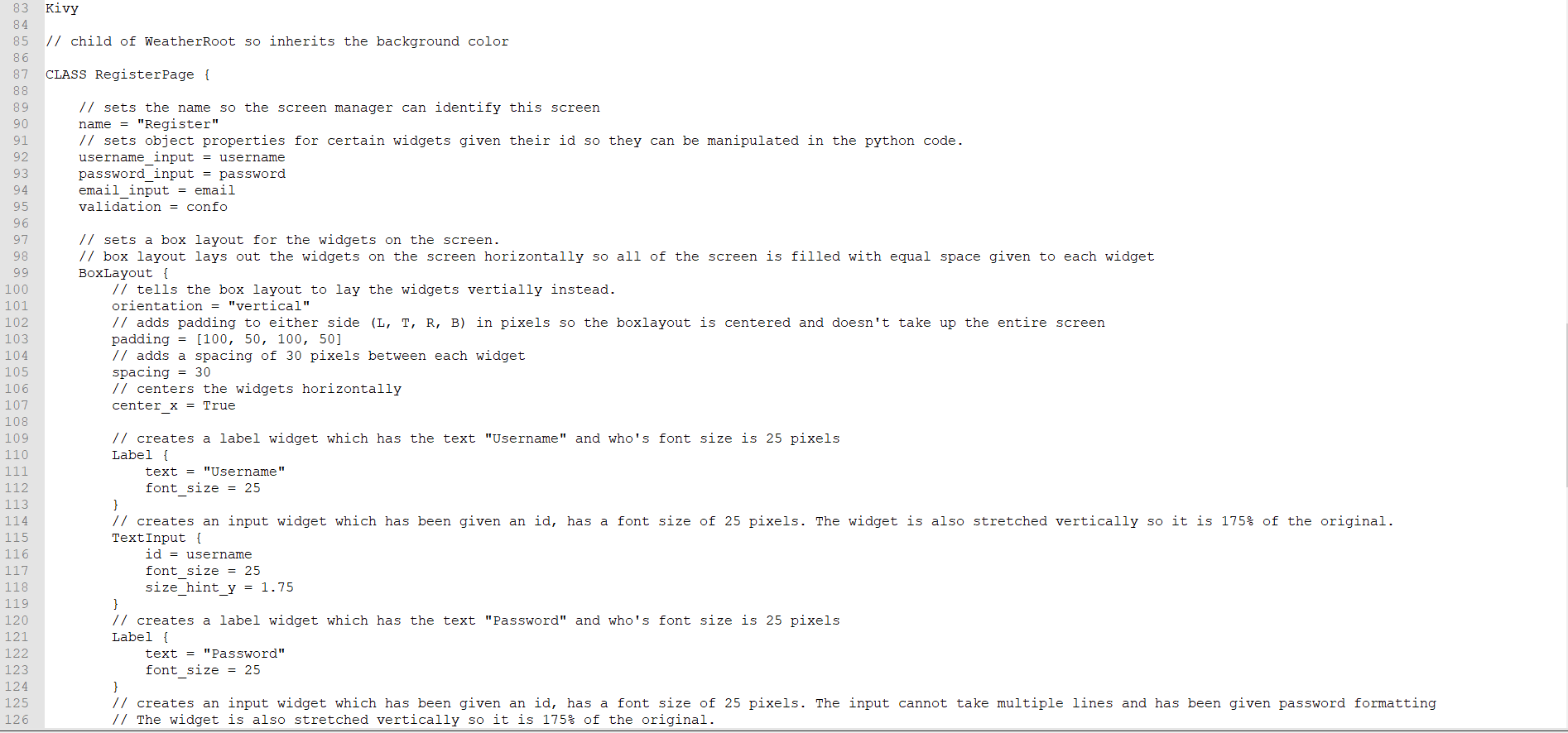
Structure Diagram:

What it should look like:



CHANGE SCREENSHOT ^ Pseudocode



PUT IN TEST DATA

## Part 4 - LoginPage Class (1.2)

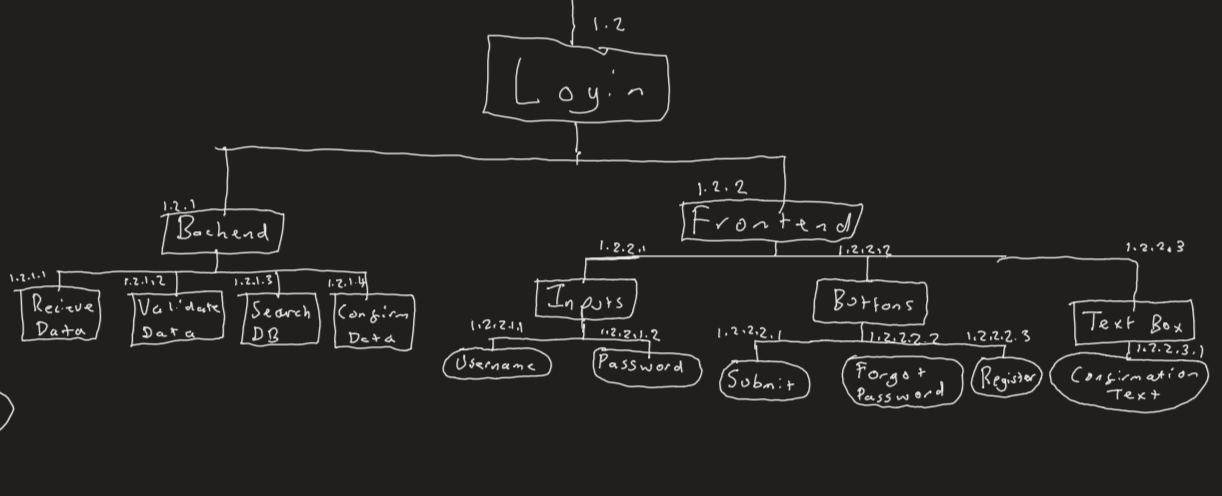
Description

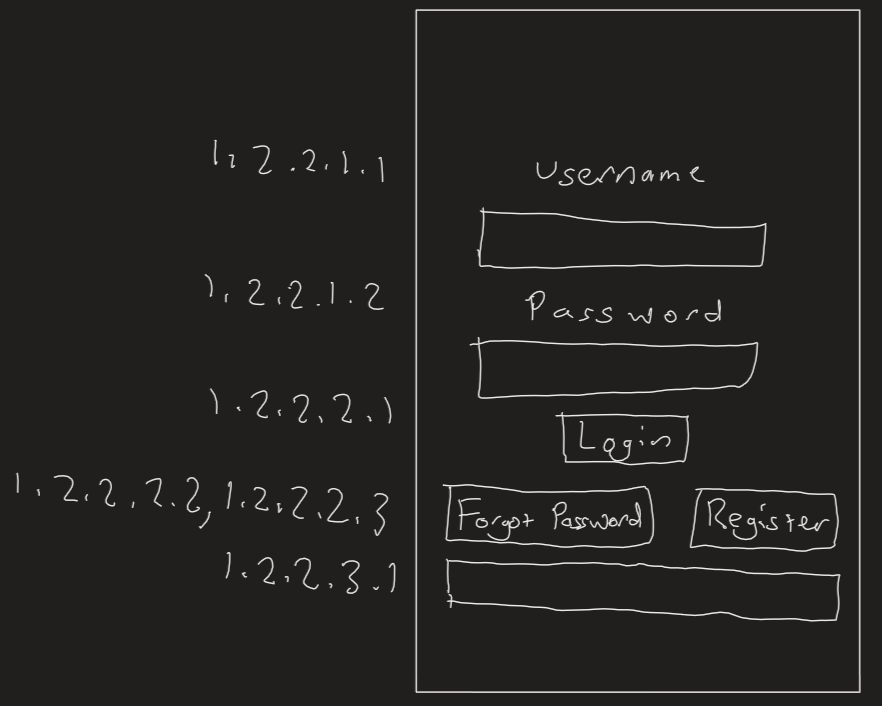
The login page class also consists of a frontend (with Kivy code) and a backend (with python code ofc). The frontend consists of a box layout also but this time its structure is a bit different. This time, there are only 2 label/ input boxes: for username and password each with their own id’s and have 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 which 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.

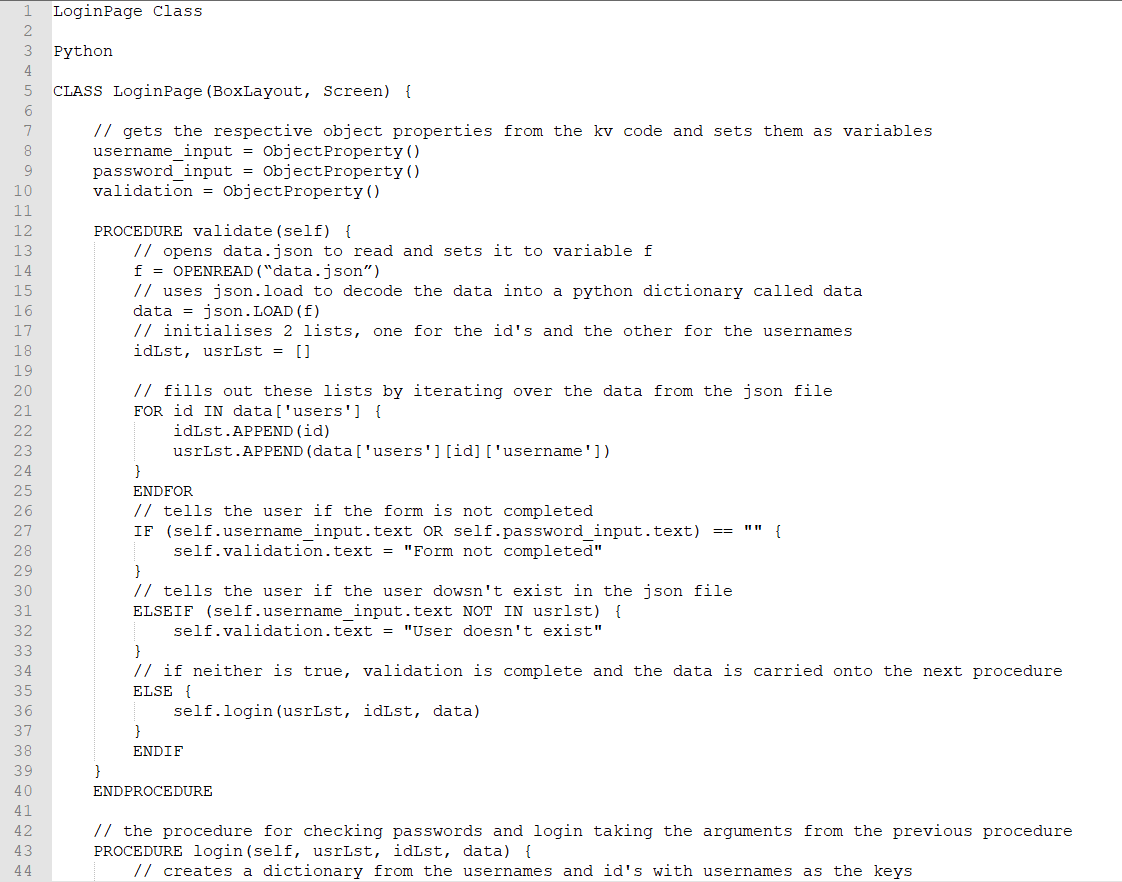
The backend would firstly, get the object properties from the Kivy code using varname = ObjectProperty(), this is done for the username input, password input and validation text. Next, there is a procedure which will extract the data from the json file/ db into a dictionary. This is then iterated over 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

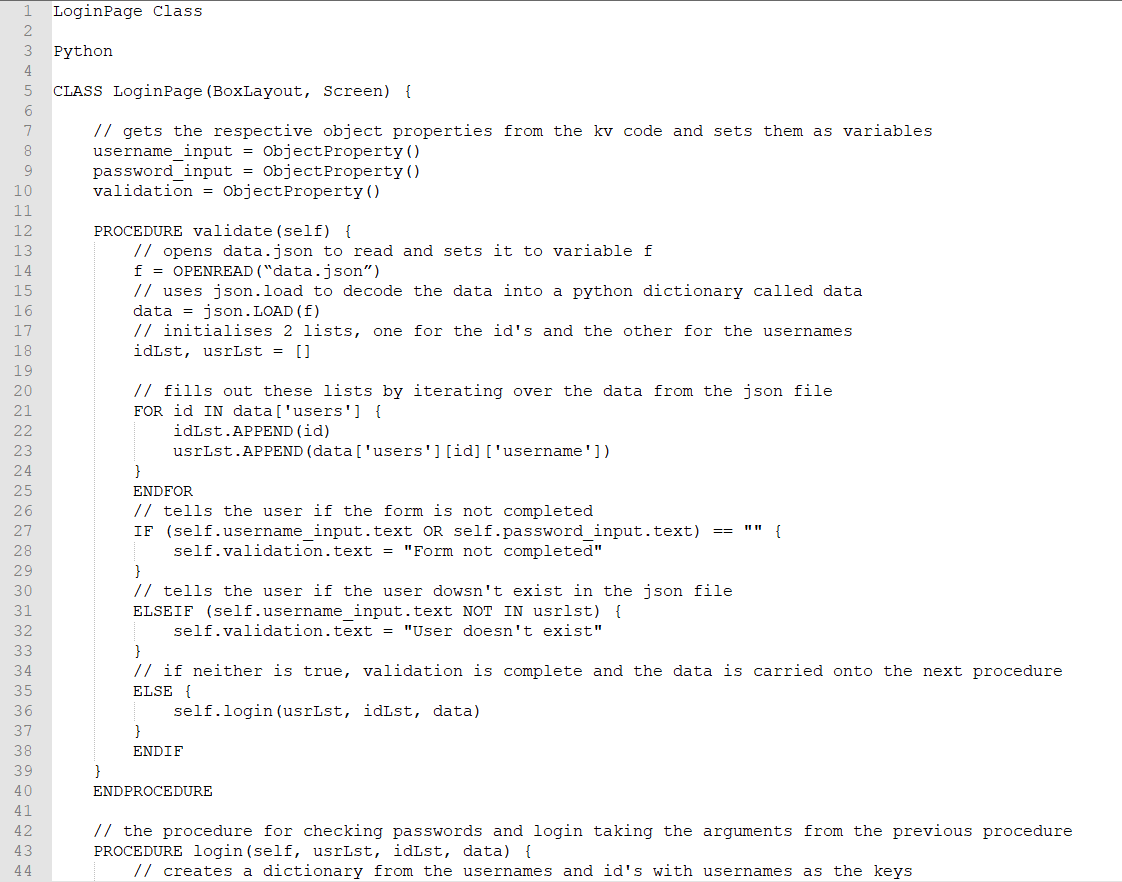
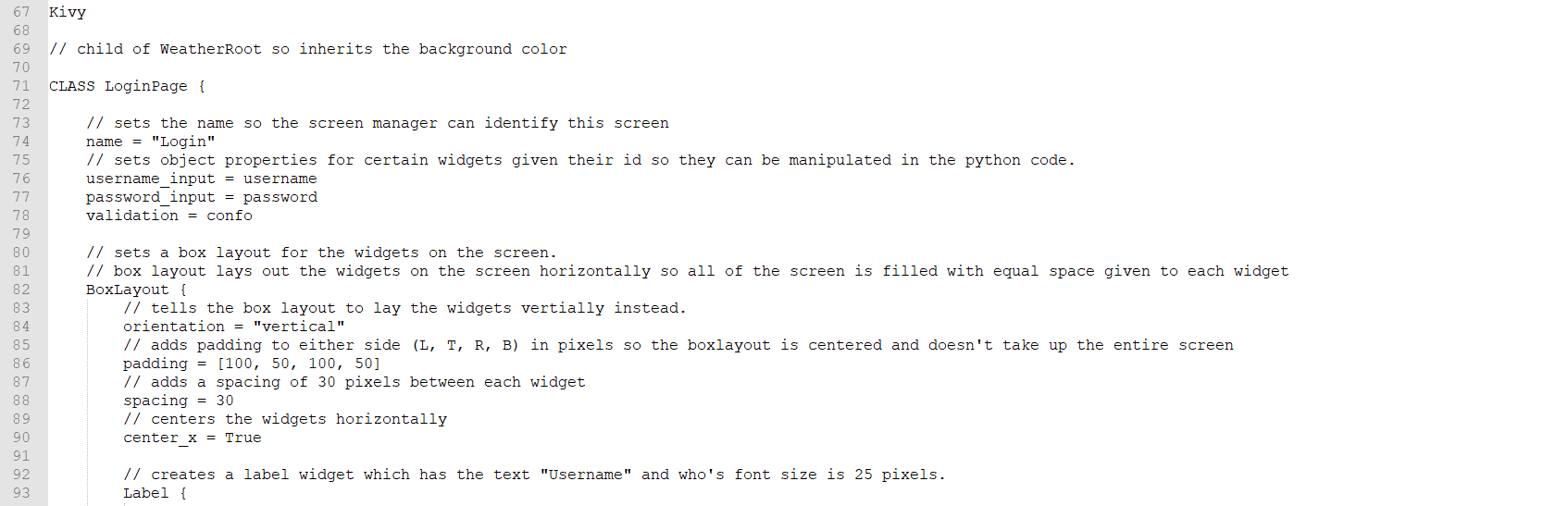
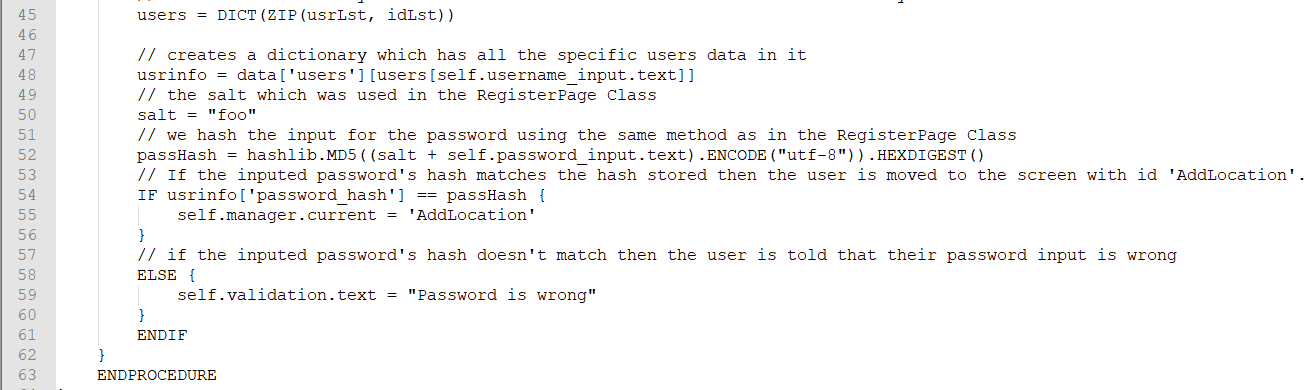
There will then be another procedure which opens the user’s info based on the username input and sets it to a dictionary. Next we create a password hash using the same salt + password input method as in 1.1. Then we check this against the password hash in the database. If they don’t match then the validation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

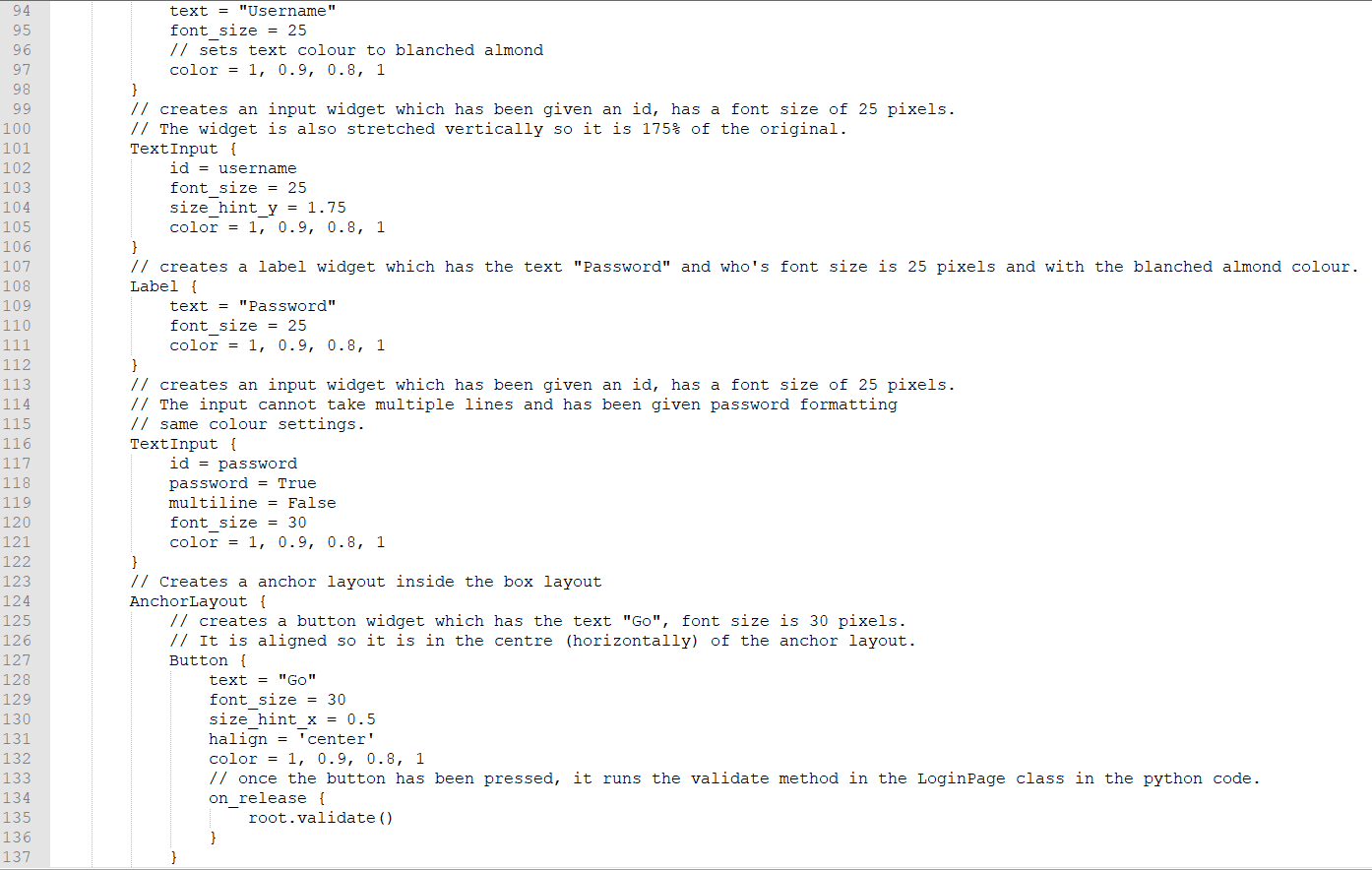
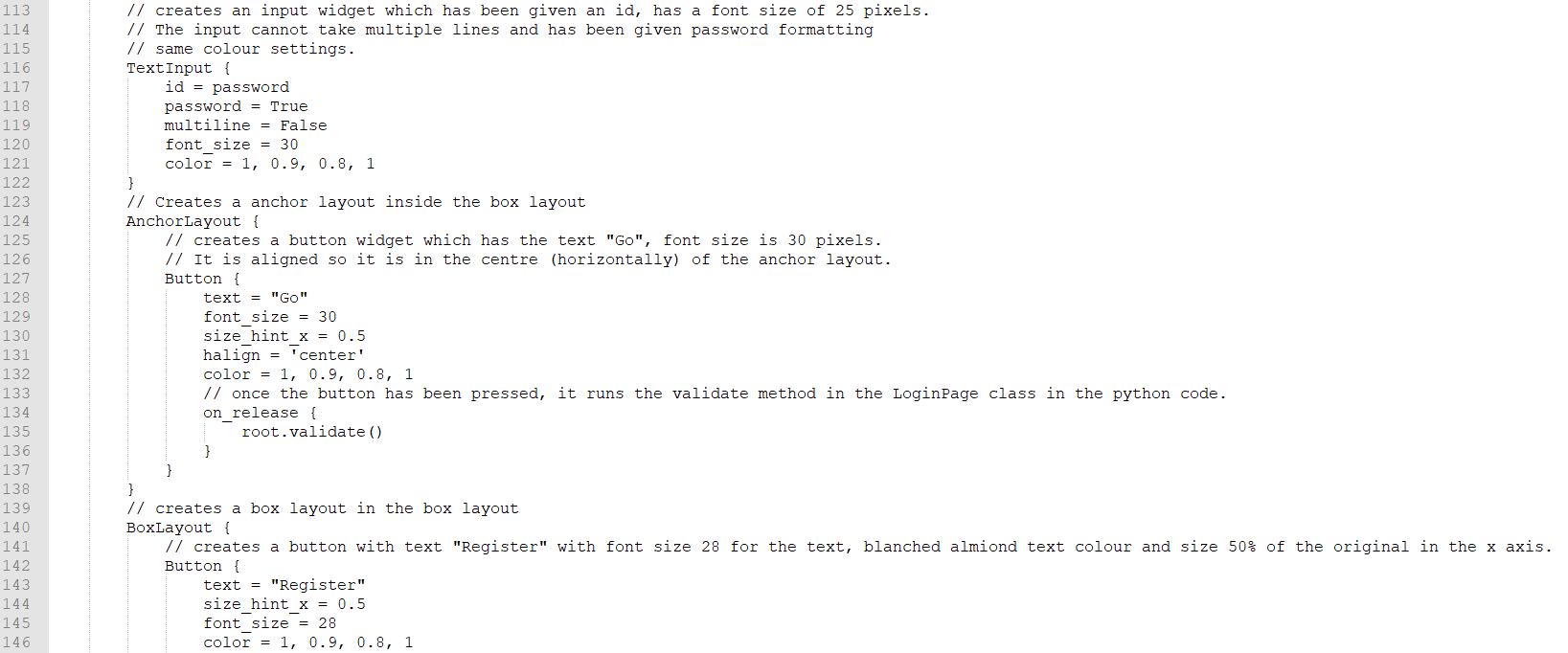
Structure Diagram



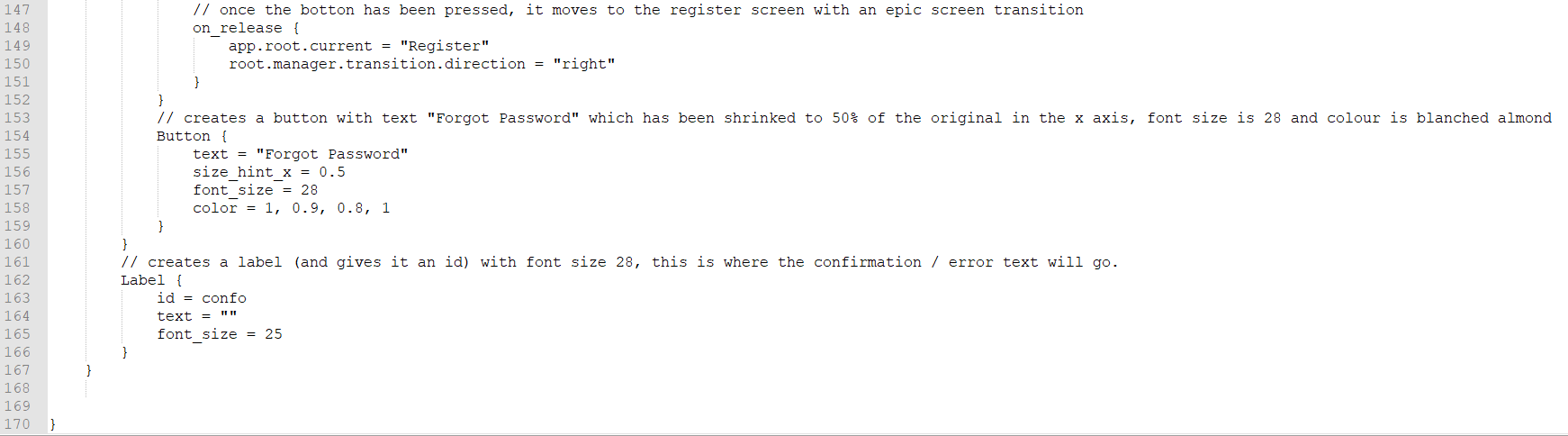
Screen Design

Pseudocode





UPDATE FOR RECENT SEARCH SYSTEM

ADD TEST DATA

## Part 5 - AddLocationForm (1.3)

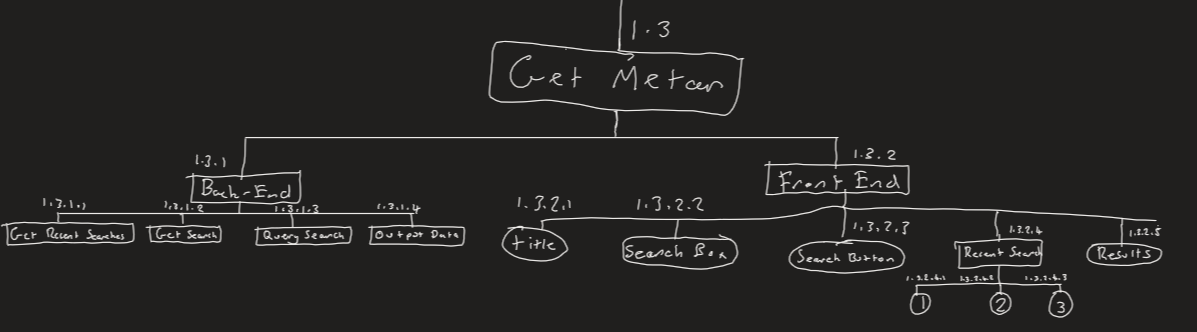
This is the main part of the program., where we get the METAR reports. Just like the other classes, it will consist of a frontend and a backend. The frontend will consist of a title at the top which will be centered in the middle horizontally. Underneath that, there will be a horizontally orientated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

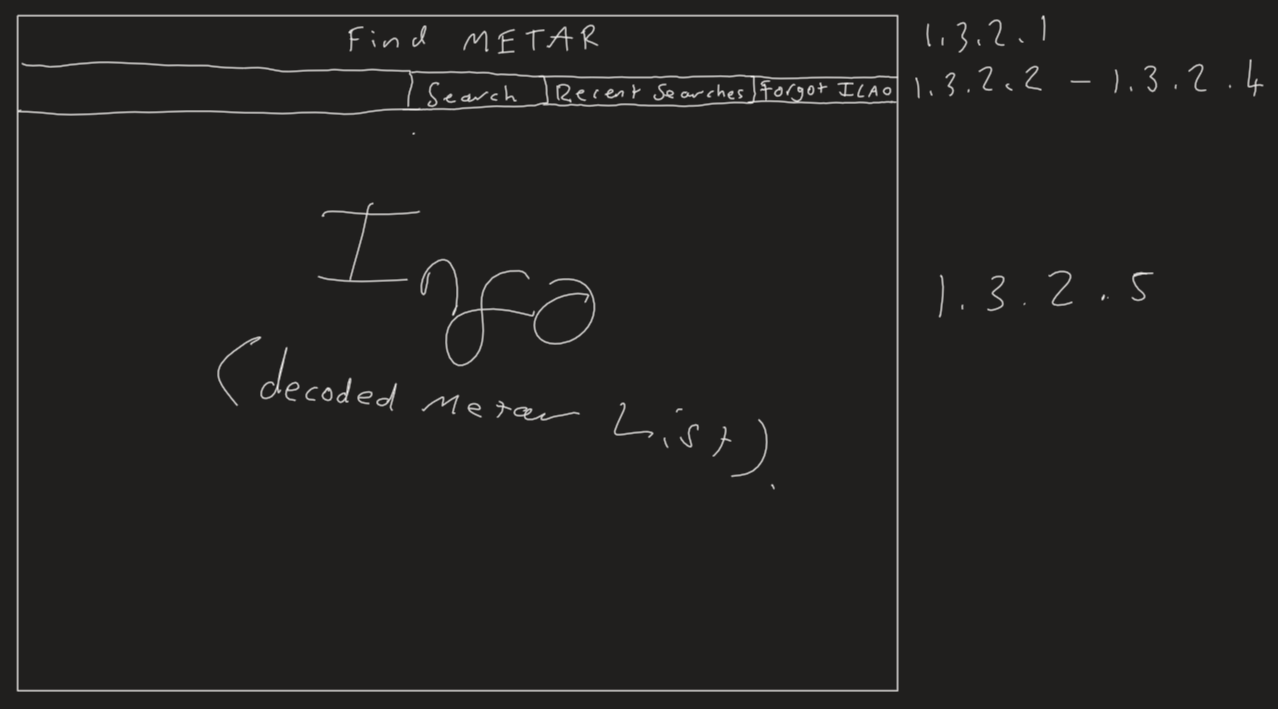
Finally, there is the big space where (for now) there will just be a ListView widget (which displays a list/ array in vertical order filling up the screen). This is where we will display the decoded METAR in the ListView way so each bit of the metar is stacked vertically like bullet points. In future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

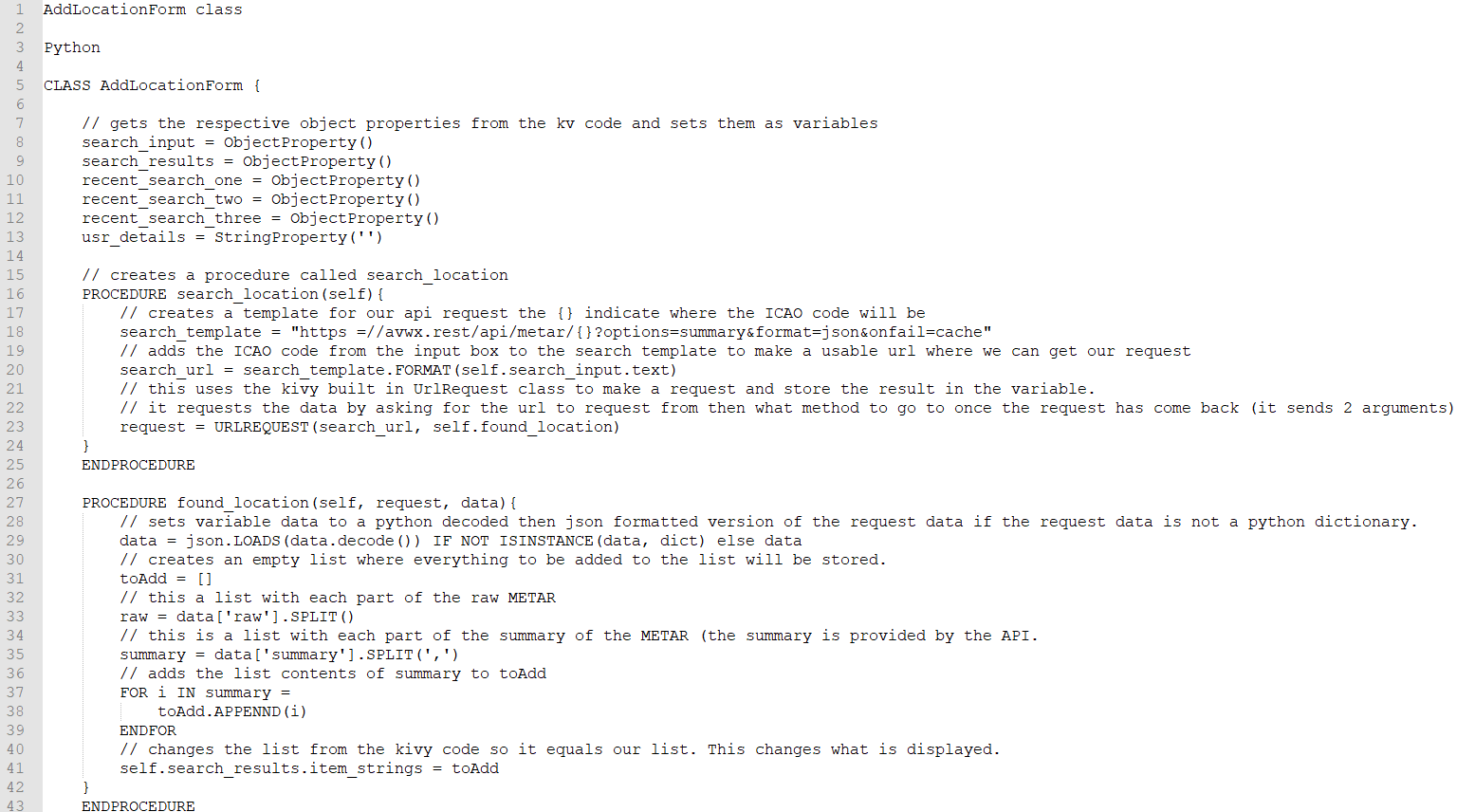
For the backend, there will be 3 procedures, one to fetch the recent searches and 2 to handle the search (fetch it, query it and output the results). The first one will get the users data from their login (saved a global variable), format it into a dictionary format then fill them in for the dropdown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

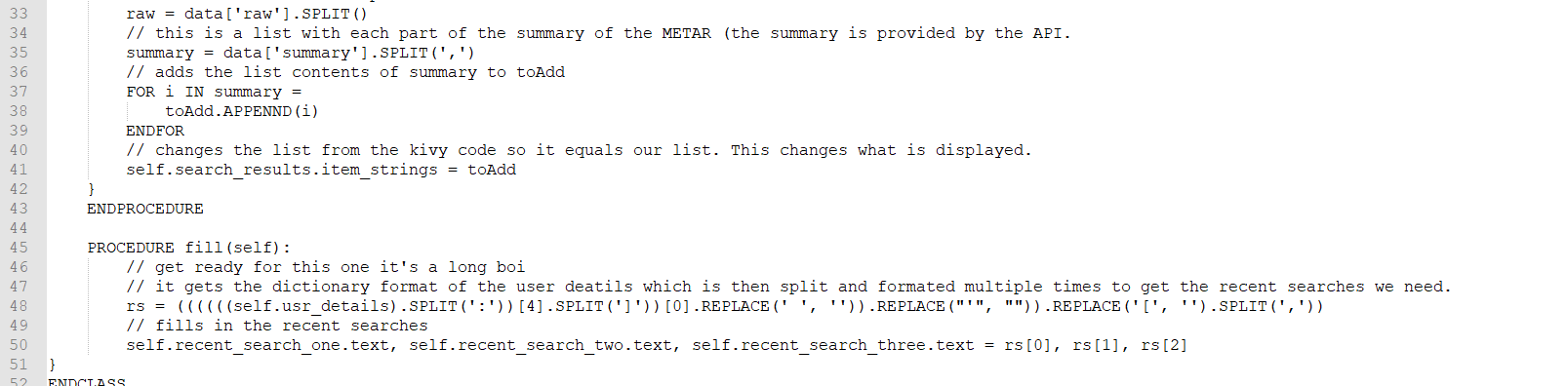
For the search, there will be 2 procedures, one to send the request and one which triggers the request once the request comes back (auto triggered with Kivy’s url request class). The first will put the search request in an already defined template then fire it off with the url request class. The second 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

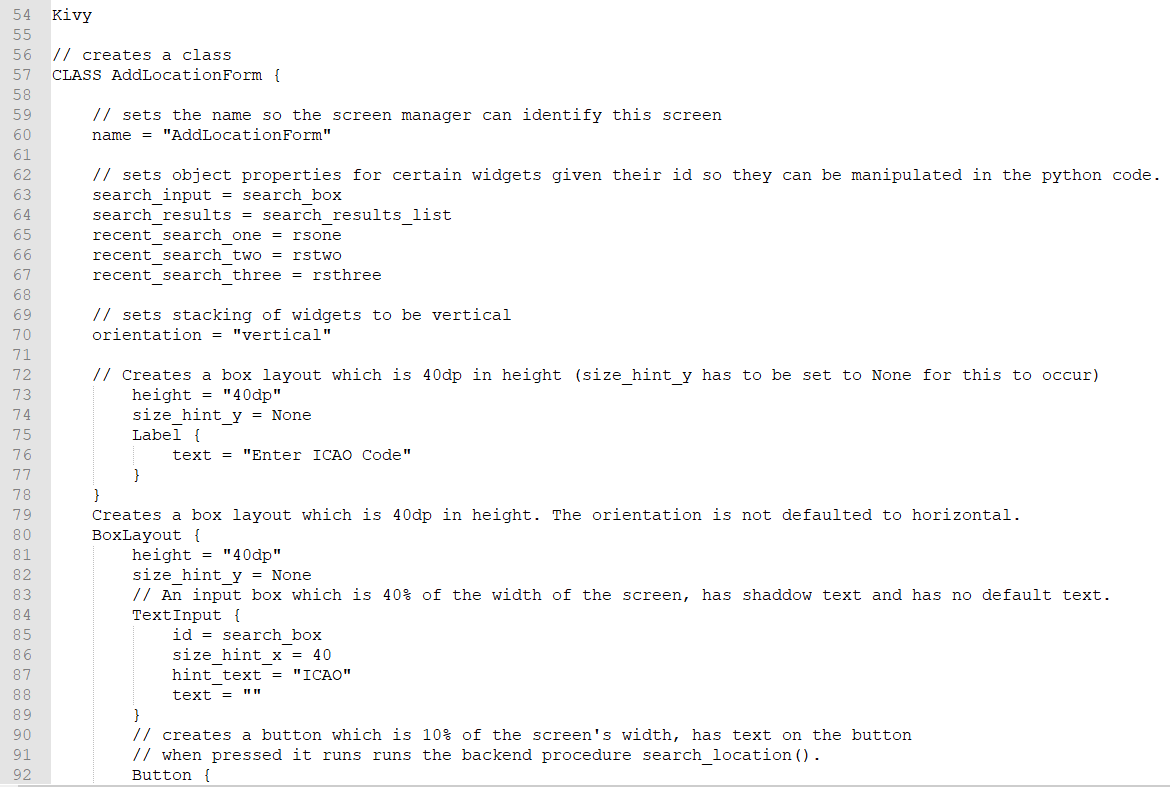
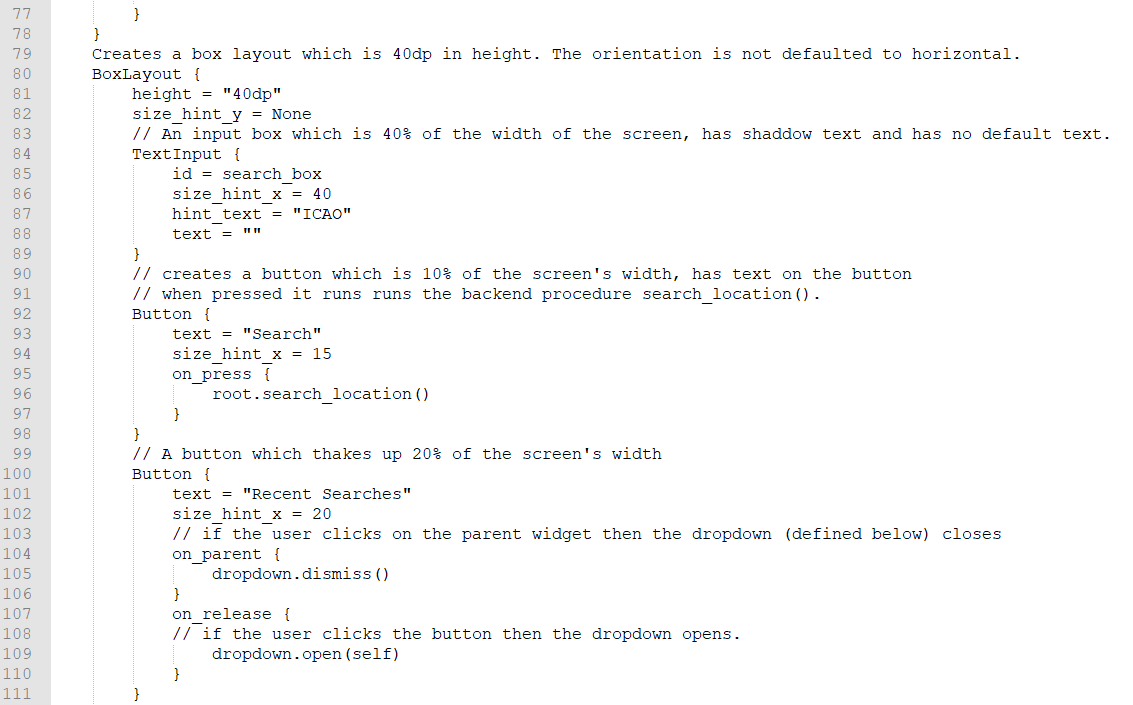
Screen Design

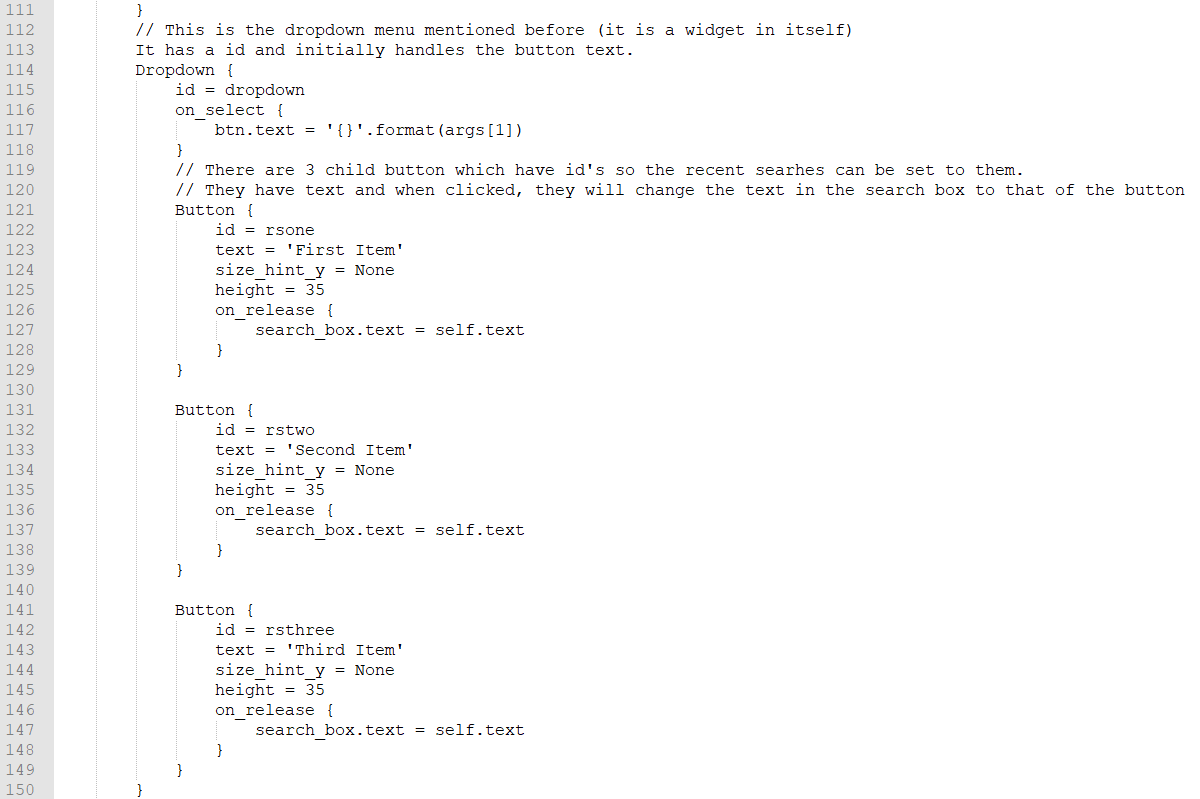
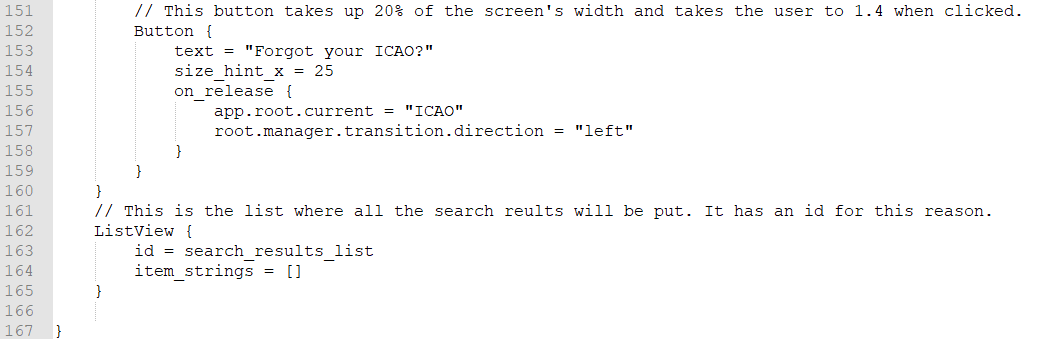


Pseudocode



Kivy





TEST DATA

## Part 5 – ICAOFinder

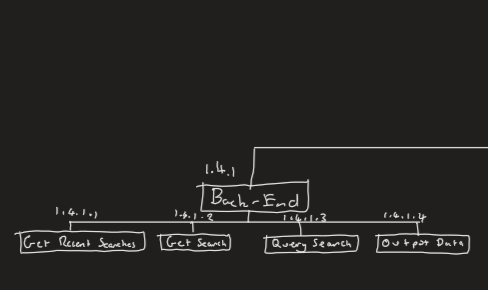
This part was added because of the demands of our clients as pilots and enthusiasts can forget their ICAO code so we thought of a system where the user could search by airport name and it would show the airports ICAO.

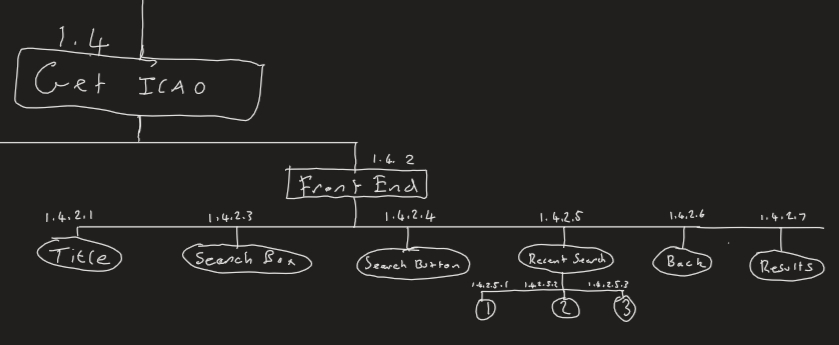
This app consists of a frontend and a backend (you’ve got the jist by now for sure). The frontend will consist of a title at the top which will be centred in the middle horizontally. Underneath that, there will be a horizontally orientated box layout. This will consist of:

* 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.
* A recent search button which will open a dropdown box directly below it. See part 4
* Back button which takes the user to 1.3.

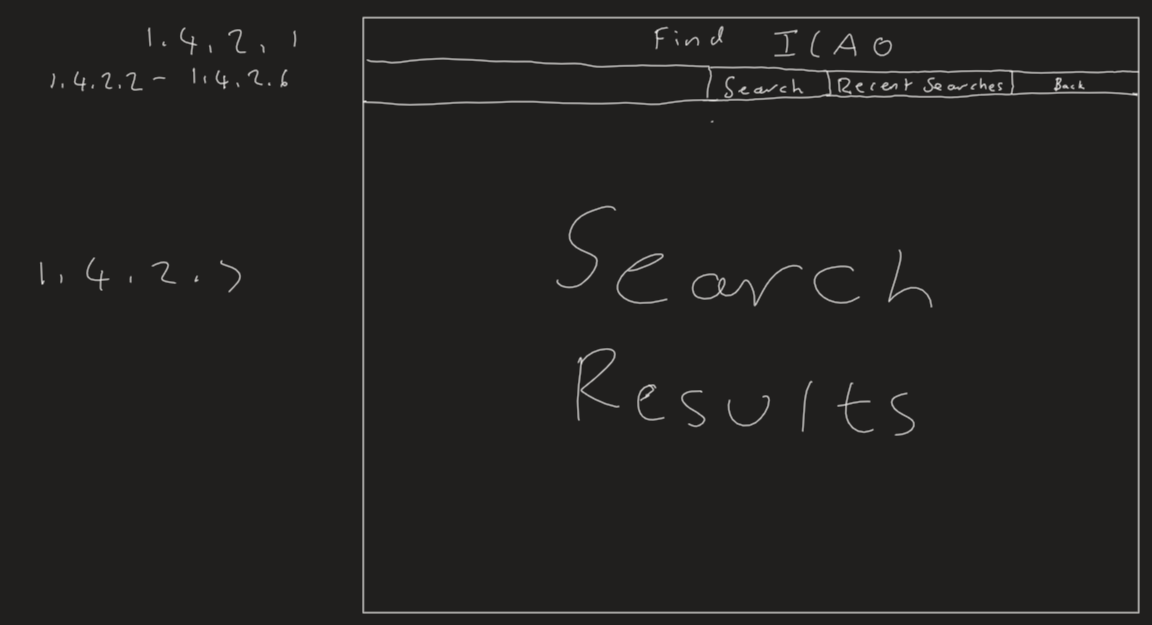
The backend will consist of 2 procedures, the first will get the recent searches from the global variable, format it and add it into the dropdown buttons. The second shall be triggered from the search button and it will get the search, query it and update the list view based on the results. The search 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

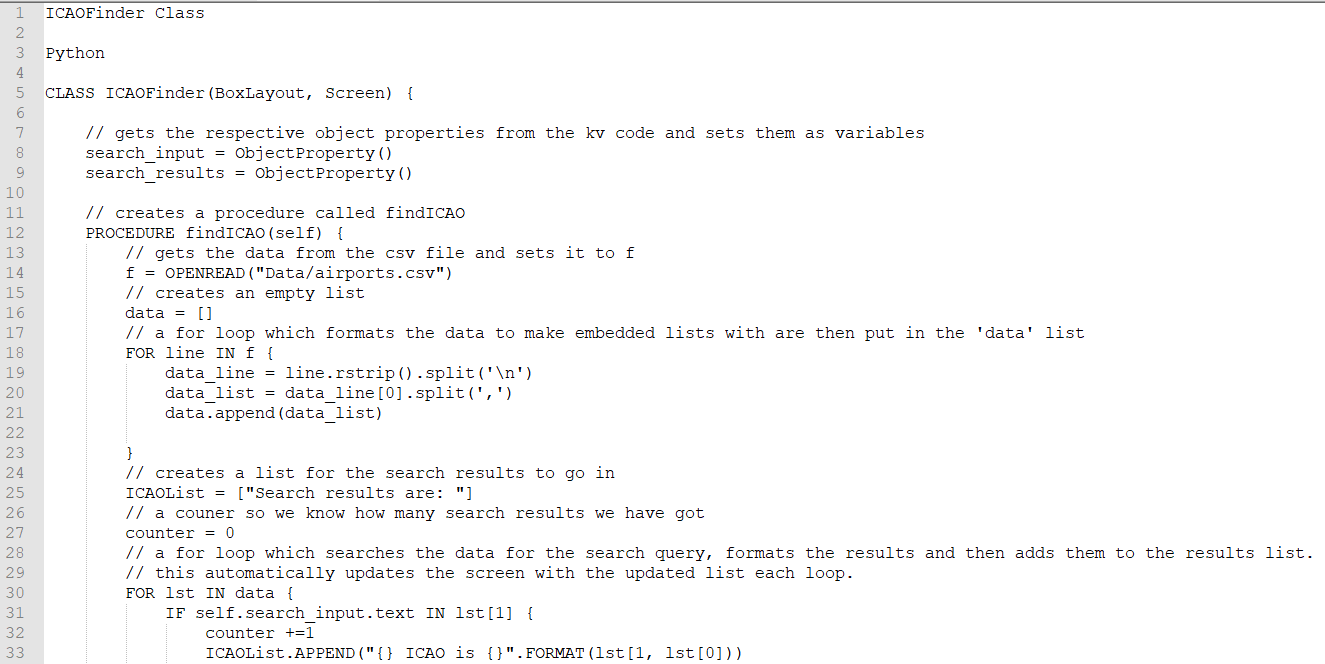


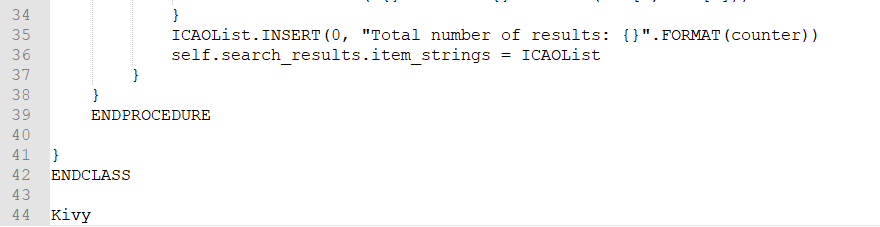


Screen Design

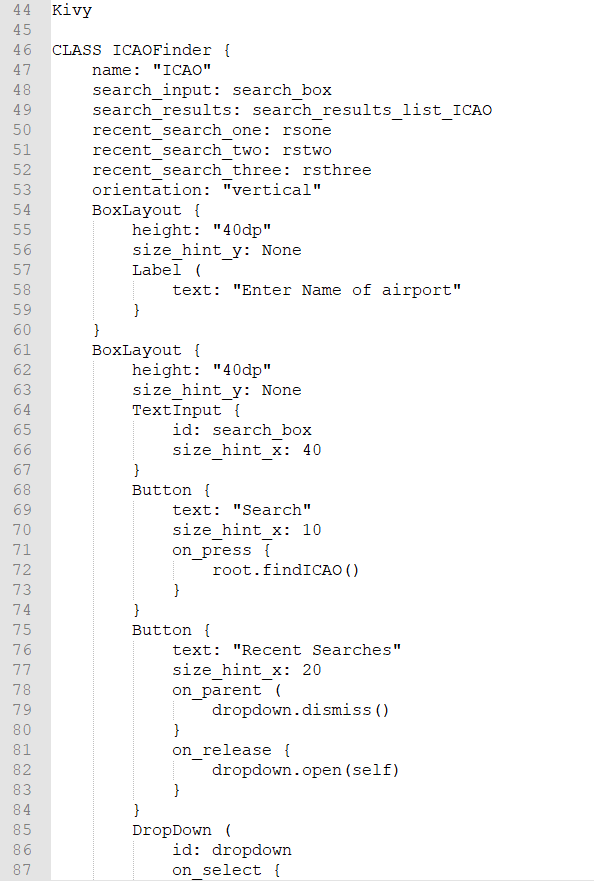


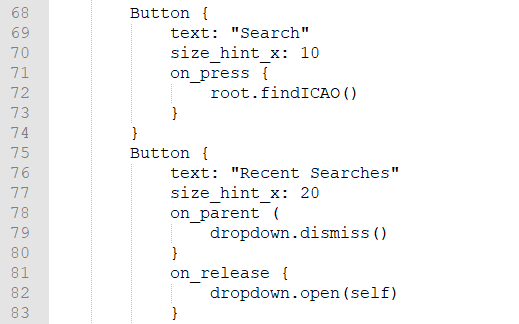
Pseudocode

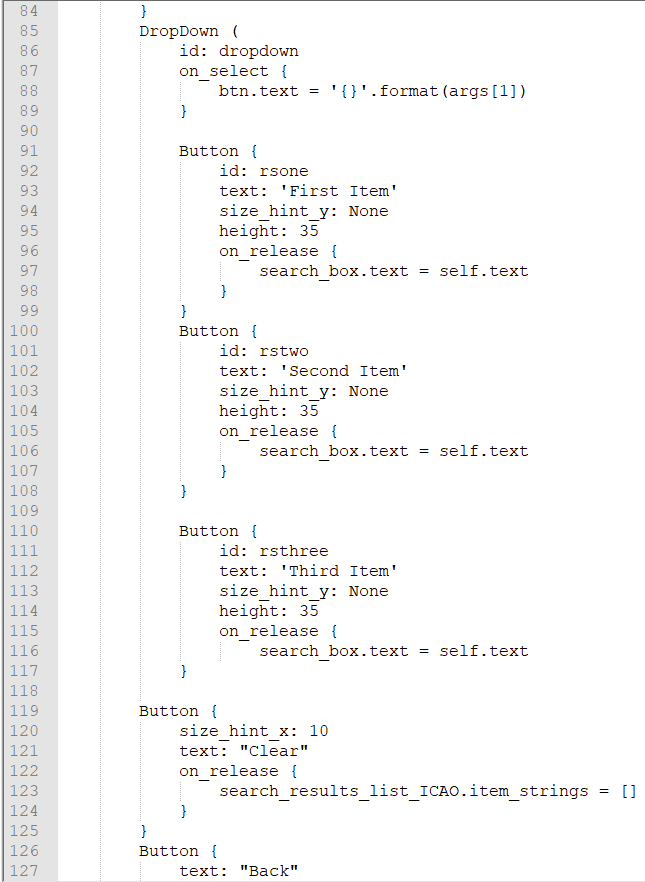
Python

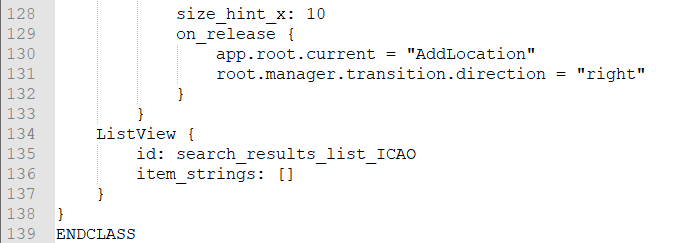


Kivy









ADD TEST DATA