**5 Implementation**

**5.1 Use of software packages**

Implementation Language: Python

**Python** was the chosen implementation language for the system. The stakeholder decided the system should be written in Python to explore the different data analysis and visualisation libraries and application of statistical techniques that can expand upon the research previously done on the SONI dataset in MATLAB. Python is open source unlike MATLAB and has a focus on readability, with a low learning curve for those new to the language [1]. For an academic user this enables them to efficiently understand the system’s code base and develop new functionality without the requirement of a commercial licence. Python is a general-purpose language and has a built-in testing framework to ensure test coverage [1]. This is important to an industrial user as when deploying the system to a production environment they can ensure the data analysis tasks performed provide accurate results.

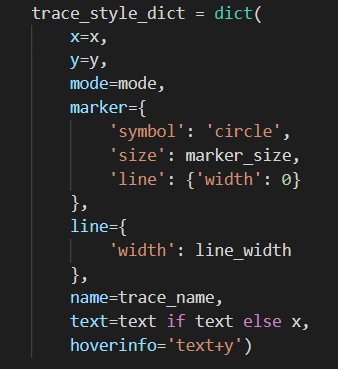
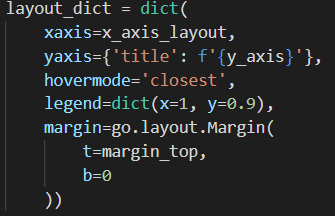
Previous analysis work performed in **chapter 2** used Python to develop functionality that the system is required to automate for the target users. The associated libraries and the functionality they provide were mapped to the solution:

* **Pandas** for dataset manipulation
* **Scikit-learn** for linear regression model construction
* **Holidays** for holiday processing
* **Numpy** for advanced mathematical functions, including calculating model performance error statistics

Python Dependency Installation: Anaconda

**Anaconda** was chosen to organise the required library dependencies for the system. Anaconda is cross platform and unlike popular Python package manager PIP can install the Python interpreter for the user along with Python packages, which reduces the number of external binaries the user must install to run the system. Anaconda is especially suitable for users performing data science as working with different analysis tools the users may have to install requirements that conflict with each other [2]. A user can create a virtual environment isolated from the Python versions and dependencies installed in the user’s system from a list of requirements (environment.yml) specific to the system. This can then be used to run the solution with no external conflicts. The Anaconda repository contains libraries targeted for data science and machine learning, which a user can install to their virtual environment to extend the system’s functionality.

Data Visualisation Library: Plotly

Matplotlib was used in **chapter 2** to produce static SVG images. This is not suitable for the system as **FTR00** the system is required to enable the user to interact with visualisations by changing different parameters and having the visualisatiosn update dynamically. Hence, python data visualisation library **Plotly** was chosen as it produces dynamically graphical visualisations suitable for time series load data. Visualisations in code are written declaratively with a focus on the data presented by the graph rather than developing a graph through a linear list of functions, **see Figure 5.1** [3].

*(a) (b)*

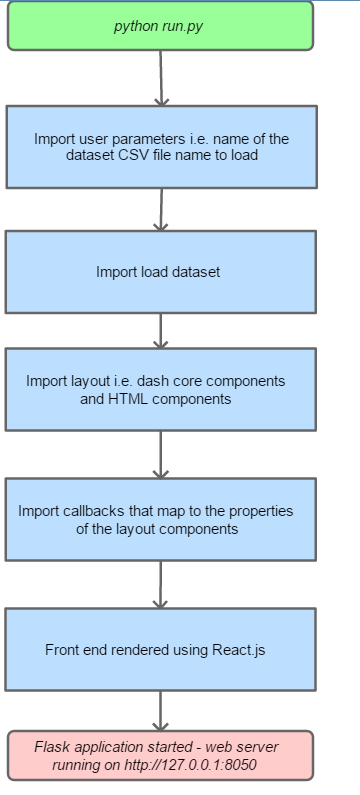
**Figure 5.1** Construction of elements of the Plotly graph in the system. (a) Arranging model data points (traces) and customising their presentation plotted on the graph. (b) Defining the layout of the visualisation.

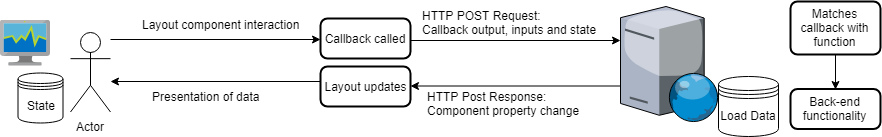
**5.2 Dash Framework**

Overview

The Python framework **Dash** was chosen to create an open source interactive dashboard that hosts Plotly graphs and the UI components to interact with the visualisations. Dash is supported by all modern browsers and is mobile ready which meets the requirement of the system to be cross platform [4]. The implementation of the solution was based heavily on using the Dash framework to meet the requirements of the target users. Dash is built upon the Python web framework **Flask**. Flask is widely supported by web deployment tools, meeting the requirement of the system having the ability to be hosted on the internet. The dashboard front-end and back-end is implemented using pure python. Dash provides components libraries that are useful for creating controls to enable user interaction with visualisations. The dash-core-components library provides interactive components that are written in Python and then translated to React.js which bundles the required HTML, Javascript and CSS to render them on a web page [5]. This was beneficial to producing the visualisations as the API used to create Plotly visualisations is available through the ‘Graph’ class in dash-core-components [4]. Furthermore, the dash-html-components library provides all web html components. Hence, there was no need to have knowledge of additional web technologies and protocols as these are abstracted by Dash. To create a component for presentation on the web-based application only the declarative format written in Python, syntactically like Plotly graph construction, is used. Dash achieved the responsiveness required by the target users by rendering the dashboard a single-page application. User interaction and the dynamic updating of the visualisations is seamless to the user, not requiring a reload of the web page.

**5.3 Dash Lifecycle**

****The system processes described in **Chapter 3.1.2** were facilitated by the use of the Dash framework, **see Figure 5.2.** Callbacks are associated with the Dash components with the input of callbacks being the properties of components that are listened and called when changed through user interaction. There are also state properties in callbacks that are input to callbacks but do not call the callbacks when changed. The return value of the callback is then a new property of the component described in the callback output. The start up processes in **Figure 5.3**  bring together all the components of the dash framework to produce an interactive data visualisation dashboard.

****

***Figure 5.2*** *The Dash components working together to make the system responsive to the user’s interaction.*

**Figure 5.3** The start-up processes of the system.

**5.4 Dash Component Implementation Decisions**

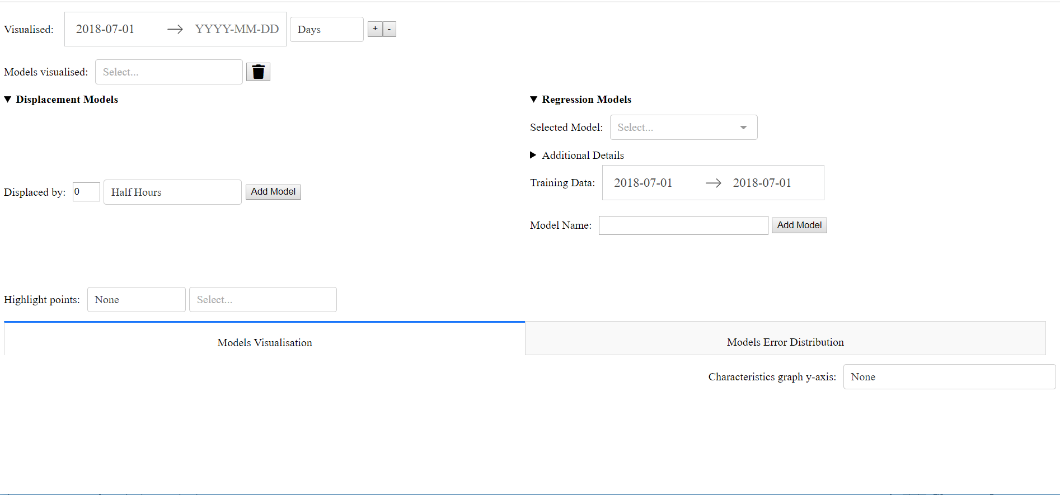
**5.4.1 Layout Implementation Decisions**

The layout of the system was defined in the *layout.py* file located in the root of the system’s code base. The Dash component libraries: dash-html-components and dash-core-components were used to create the layout.

Naming Convention

There was a naming convention for the various elements in the layout. It is important to have a consistent naming convention for a project [6] with an additional importance to the system as the ids in the layout relate directly to callbacks. Any inconsistency in naming could cause mapping errors with interaction not resulting in expected functionality. The ids of the components and stylesheet classes names were named using kebab case, which replaces spaces with hyphens and is all lower case e.g. variable-id-1. All components were appended with a general description of their purpose in the layout e.g. all dropdown components had ‘\_selection’ appended at the end. The purpose of this was to make them as descriptive as possible to developers who want to understand what function the component has.

Component Styling



Height:

35% vh

Width: 50% vw

Width: 50% vw

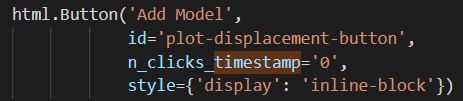
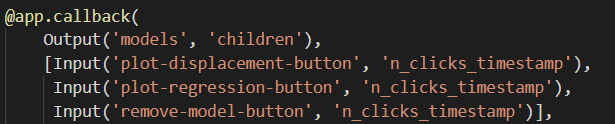
**Figure 5.2** Example of two flex items contained in a single flex box container. The model construction user interface elements for displacement and linear regression models can shrink and grow to match the constraints stated. vw – viewport width. vh – viewport height

There were several considerations made when styling user interface components. The system was required to be scalable to the user defined window size in **chapter 4.3.1.** To meet this requirement components were styled to be contained within flexbox containers that shrink and grow the components to the available window space [7]. Furthermore, instead of using hard coded pixel sizes for the flex box container the height and width of the viewport (browser view) were used to make the layout scalable to the user’s defined window size. This is especially important for users wanting to use the solution on a mobile device. For an example of a flexbox container implemented in the system’s layout **see** **Figure 5.2.**

Dash HTML Components

The following html components were implemented in the layout of the system:

* **divs** used to separate different interactive components and contain the plotly graphical visualisations and model error statistics table. In addition, a hidden div design is used for the user’s client-side state. This is elaborated on in the **forecasting model implementation decisions chapter.**
* **button** is used for user input confirmation. This enables the user to:
* pan the visualised dataset
* remove a plotted model
* add a model, **see Figure 5.3 (a)**

1.  (b)

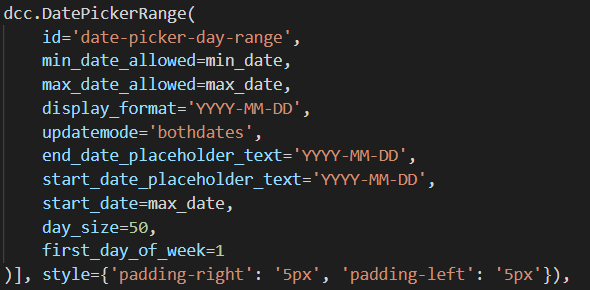
**Figure 5.3** Functionality to add a displacement model. (a) The button component enabling the user to add a displacement model. (b) Call back wrapper for the function that updates the model state. It includes the n\_clicks\_timestamp property of (a) as input.

There was an issue with different buttons being used as inputs for the same call back function. The receiving function needed a way to know which button was clicked first in order to execute the correct conditional logic. To resolve this a property ‘n\_clicks\_stamp’ was added to the buttons in the layout. This property contained the date-time the button was clicked and updated upon clicking the button. This was then included as input to the callback functions that correspond to the button’s functionality, **see Figure 5.3 (b)**. The conditional logic in the backend then recognised the button with the maximum n\_click\_stamp was the one clicked last by the user, and hence the correct conditional logic was executed.

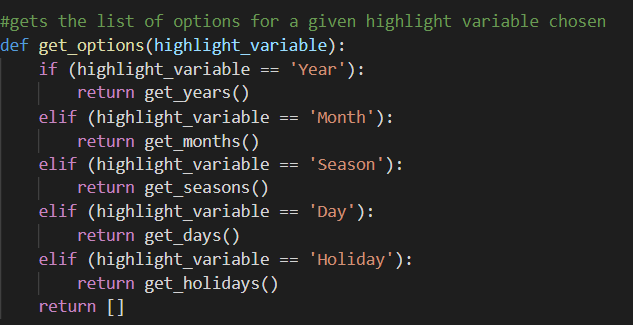
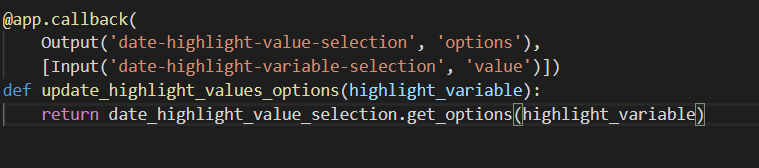
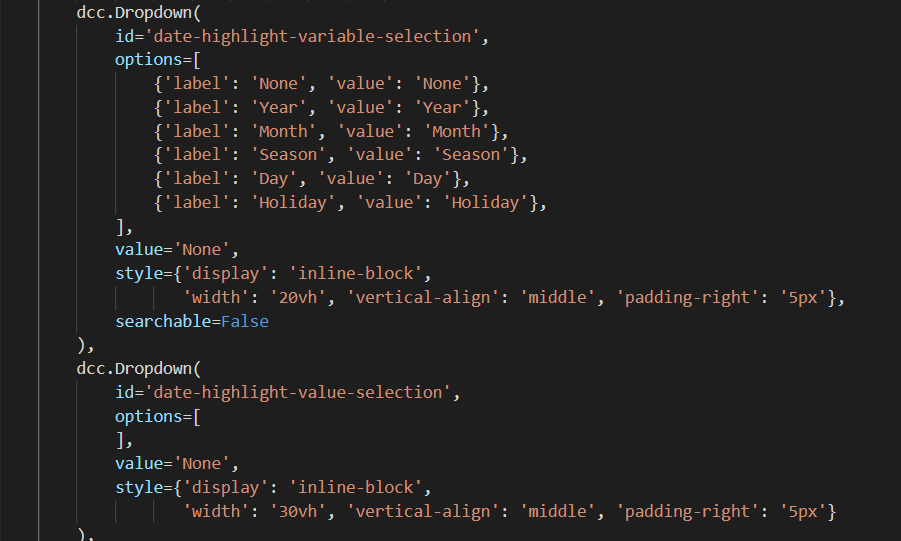
* **Details** and **Summary** was used for hiding additional details and functionality not essentially required to fulfilling the user’s user flow upon initial load. They can be opened with a mouse click of the bold label with a carat. This enables the user to:
* Only be presented with the components enabling them to visualise the load dataset upon initial load. The model construction interface is hidden.
* Optionally view the details of the linear regression model chosen
* **H4** was used for labelling distinct sections of the system to make the system more accessible to the user.

Dash Core Components

The following core components were implemented in the layout of the system:

* **DatePickerRange** was used for defining range values constrained by the date range of the load dataset. The format of the date for user input conforms with the ISO standard defined in **chapter 4.3.2.** This enables the user to:
* Change the range of visualised load data **see Figure 5.4**.
* When constructing a linear regression model, change the range of data used for training the model
* ****When evaluating a model’s performance, change the range of test data.

**Figure 5.4** The Date Range Picker component. The range of data shown is set by the property ‘min\_date\_allowed’ and ‘max\_date\_allowed’. The format of the dates presented by the component is set by the property ‘display\_format’. The date picker range requires both the start date and end date to be set for the input to be with the property ‘updatemode’ being set to bothdates.

* **Dropdown** are used to list options for the user to toggle and use as parameters for visualisations and model construction. This enables the user to:
* Choose the unit to pan the date range of the dataset by
* Choose from a list of the models to remove
* Choose from a list of the included linear regression models to add
* When constructing a displacement model, choose the unit of displacement
* Choose the type of variable and the value of the variable to highlight the dataset by, **see Figure 5.5**.
* ****Choose the data variable to visualise below the load forecasting graph

***Figure 5.5*** *The dropdown components enabling the user to choose what type of day they want to highlight in the visualisations. The default list of items for the user to choose from are set by the ‘options’ property. The highlight value selection dropdown does not have any options by default as the option list is dependent on the value chosen in the highlight variable dropdown.*

*(1) Option is chosen in the variable selection dropdown*

*(2) Option selected is input to the call back function*

*(3) The list of values that correspond to the highlight variable chosen is returned.*

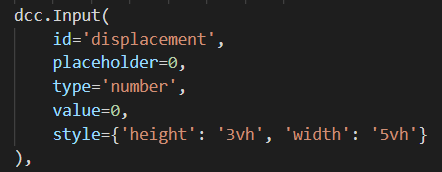
*(4) The output of the call back function is outputted as the list of options for the value selection dropdown.*

3

2

4

1

* Tabs were used to separate the different visualisations, conforming to the tabbed design requirement in **chapter** **4.3.**
* **Input** is used for user customisable parameters. This enables the user to:
* construct a displacement model by setting the displacement value to use as the load forecast. Input uses client-side-validation to ensure the entered value is a numeric value by setting the ‘type’ property **see Figure 5.6**.
* setting the linear regression model name

**Figure 5.6** The input component enabling the user to enter a numeric displacement value.

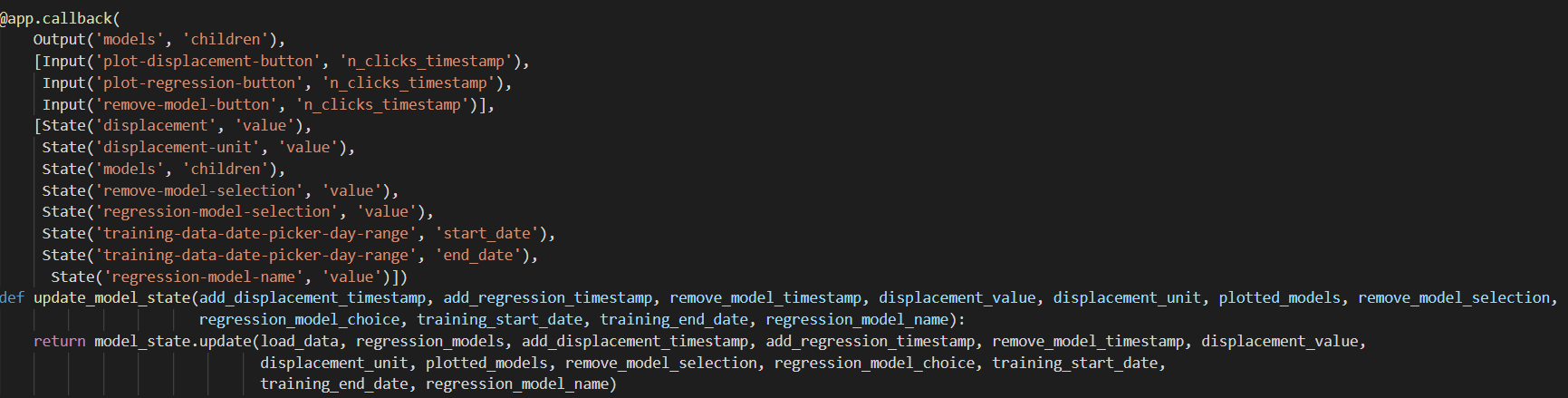
**5.4.2 Callback Implementation Decisions**

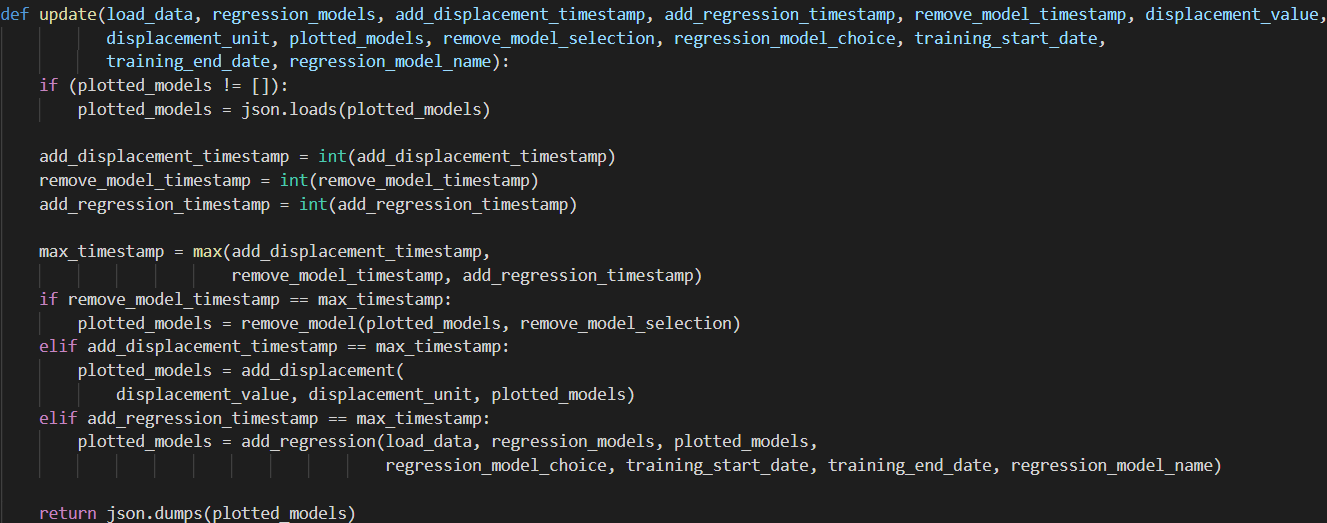
All the callbacks included in the system are in the *callbacks* folder which each callback file ending in *\_callbacks.py.* There is an *\_\_init\_\_.py ­*file contained in *callbacks* that that packages all the callback files into a single module with the namespace ‘callbacks’. This module is imported in *app.py* which is executed when the system is ran, binding the callbacks with the system’s layout. There were challenges identified with callbacks during implementation.

Unique Output Restriction

Callbacks must have a unique output property. This meant the interface to the backend of the solution required functions with many arguments if multiple flows shared the same output component. The logic the user wanted had to be determined through handling code in the back-end by inspecting the arguments passed. An example of this is in **Figure 5.7.** Updating the client-side model state of the user may involve the user doing one of the following:

* Removing a model
* Adding a displacement model
* Adding a regression model

The callback in **Figure 5.7(a)** needs to accept the layout component properties required to filled by the user to this this functionality in the back-end as input or state variables. Then, in **Figure 5.7(b)** the flow must be determined from the component properties the user has changed in the layout.

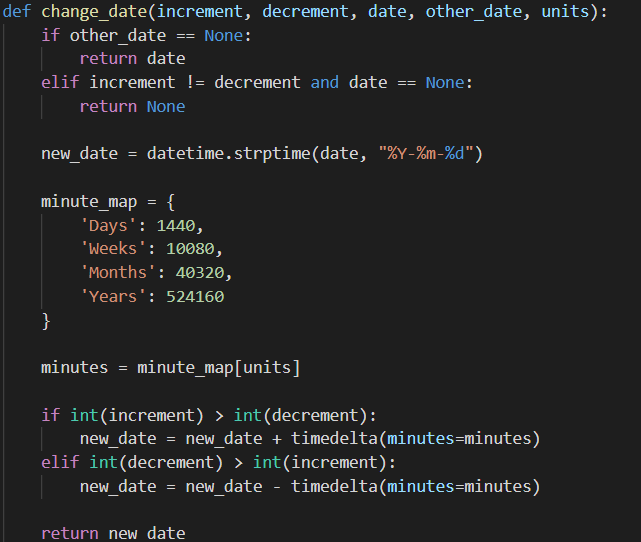
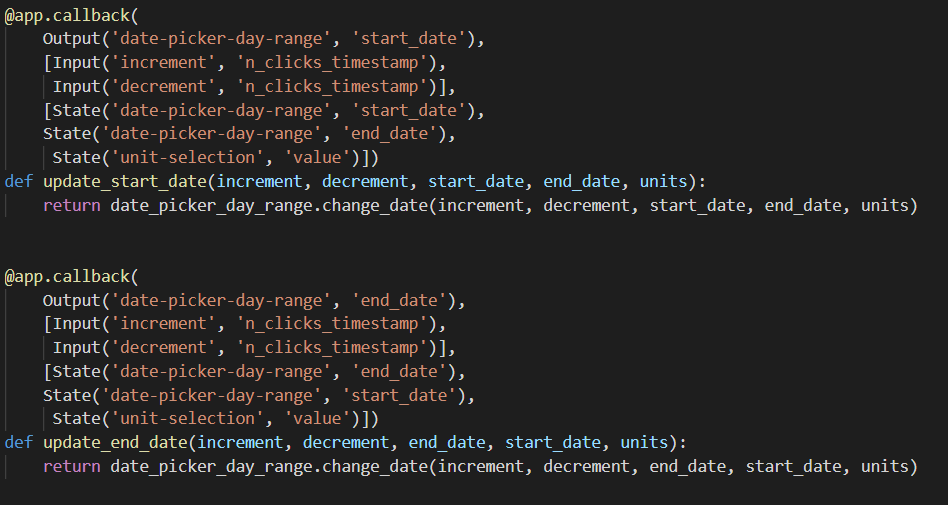
*(a)*

*(b)*

**Figure 5.7** Updating the model state logic (a) Callback for the model state component. (b) the interface to logic that returns a change to the model state. Contains handling code to determine which flow the user has selected by inspecting the timestamp of the button pressed. When one of the conditions is met in one of the if statements its logic path is followed and then returned as output to the callback for the component.

Multiple Outputs Limitation

There is no support for multiple outputs in callbacks as of the version used by system (0.28.5). A callback function had to be written for each specific component property to update. In **Figure 5.8** the back-end functionality for the user panning the dataset by a defined unit and by choosing a new start and end date to visualise the dataset is shared. This was done to ensure logic performing the same operation – changing the date picker’s start and end date, was not being repeated. **In Figure 5.8 (a)** the input variables are the same for both callbacks, meaning they are called separately when an input property change. Hence, with multiple outputs to the start date and end date the function in **Figure 5.8(a)** would not be called twice and instead return both the new start date and end date of the date picker. The handling code to determine whether the increment or decrement panning button was pressed would not be executed twice. Performance optimisations like this would be optimal as Dash is single threaded with callback functions not being performed concurrently. As of version 0.39.0 multiple outputs are supported and therefore work to rewrite this functionality is backlogged.



**Figure 5.8** Updating the load visualisation date range picker logic (a) The callbacks for updating the date picker start date and end date. (b) The back-end functionality that returns a new date for the date ranger picker depending on the whether the user has chosen to pan the dataset or select a new date.

(a)

(b)

**5.4.3 Back-end Implementation Decisions**

The functionality that provided the output results to callbacks for presentation in the user interface components is in the *component* folder, for preprocessing the dataset in the *preprocessing* folder and for handling model state in the *state* folder. The naming convention for variables in these files was snake case, which follows the best practices outlined in PE 8 [6]. There were several key implementation decisions made for the system.

Graph visualisation implementation

**Figure 5.9** The function calls made for each different type of graph in the system. Shared processes are in orange. add\_visualised\_graph: returns a generated load visualisation graph with the trace datapoints dependent on the input values. generate\_graph: returns a plotly graph with the data and layout encapsulated.

All graph visualisations in the solution have an associated callback output in *graph\_callbacks.py*. The callback is called when one of the input values that change when the visualisation of the graph visualisation change. The input and state values are passed onto the update function. Each graph has different functionality in the update function, **see Figure 5.9**. There is shared functionality between all the graphs that is contained within *graph\_util.py*, which contains functions that all graph visualisations need to call e.g. generating data point traces, generating highlighted data points and creating the Plotly graph object. There is different functionality with add\_error\_graph being called to update the error distribution graph, which generates a graph with data point traces unique to the graph. The load graph and characteristics graph both share the same date x-axis but with different layout styles and different y-axis values.

An object-oriented programming (OOP) design to wrap the functionality and data associated with creating a graph inside graph object [8]. A graph class would have behaviour which can be shared by all inherited subclasses (load, characteristics and error distribution graphs). Graph objects (instances of a graph subclass) in the system would be constructed with the input data to define their instance variables, and methods would be called to perform the data manipulation necessary to generate the visualised data points. This would enforce a contract of using reusable functions rather than the calling the utility functions in the current system.

|  |  |  |
| --- | --- | --- |
| Browser | SVG | ScatterGL |
| Chrome | 28.6 | 2.33 |
| Internet Explorer | 38.96 | 4.46 |
| Firefox | 65.34 | 2.28 |

However, the data the graph object cannot be stored indefinitely as the system is stateless. There would no benefit for polymorphism for creating generic functions accepting instances of all graph subclasses as the superclass Graph object instance. The only function shared by all graphs is *generate\_*graph, which is already called by the three graph types. Furthermore, the result of the update function is a python dictionary encapsulating a Plotly ScatterGL object and Graph Layout object. This would require accessing the data in the system defined graph object to create these objects, an unneeded extra level of abstraction that requires extra code. A pure function (graph update function) given the same input will always return the same result was determined to be the best solution. This is called a procedure-oriented way of programming [8].

**Table 5.1** List of times taken from initial system load to visualisation of load from 2017 to 2018 using different web browsers.

To prevent an error with the Plotly API not changing the the graph visualisation because the ID of the Plotly graph object did not change, the returned generate graph object ID has the current system date time appended to ensure it is unique.

There were performance considerations made when deciding how to visualise the graphs. Plotly Scatter graph component was initially used but took an extensive time to load, **see Table 5.1,** and was sluggish when interacting with a high density of visualised load data points. The Plotly ScatterGL graph component was implemented. The component is built upon WebGL which is well supported by all major web browsers. It more performant in rendering graphs with larger density of plots and as the SONI electronic load dataset contains over 140,000 rows, it is the preferred implementation for visualising graphs in the system.

The error distribution graph proposed in **chapter 4.3.2** had the APE of the visualised data points on the x-axis and the cumulative percentage on the y-axis which requires specific processing to generate the required values. In **Figure 5.9** the processing is optimized by using the load visualisation graph plotted forecasting model data as the load forecast to calculate the Absolute Percentage Error (APE) statistic with actual load. This optimization means that the error visualisation graph does not need to recalculate load forecasts already executed to generate the load visualisation graph data points. The data of the load error visualisation graph is the input to the callback function that updates the error visualisation graph, thus the models visualised in the error visualisation graph is updated synchronously. A consequence of this is that rendering the load visualisation graph is a prerequisite of rendering the load visualisation graph, needlessly increasing processing time for the user if they only want to visualise one of the graphs.

**Figure 5.9** Pseudocode representation of the processing of data points of forecasting models visualised in the load visualisation graph for the error visualisation graph.

List of error traces to plot = empty

x = the load visualisation graph

iterate through all the model plots in x:

plotted model = the x,y model data points

calculate the APE between the actual load and the plotted model forecasted load of the model for the same data entry and record this

sort all the data entries by the APE ascending

give every data entry an increasing index

calculate the cumulative percentage by dividing the entry index by the number of datapoints \* 100 and record this

set name visible on the legend as the name of plotted model with (Visualised) appended

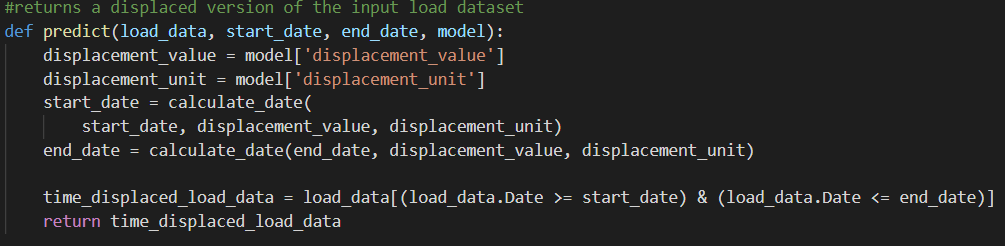
set colour of model as the colour of the plotted model

generate x,y error traces using the APE and cumulative percentage

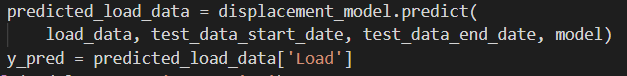
add model error traces to the list of error traces

Forecasting Models Implementation

The processes to create and evaluate forecasting models were adapted from functions written in the analysis. The model functionality is contained in the *models* folder with subfolders *displacement* and *regression*. The entry point for both models to generate load forecasts is the *predict* function. For the displacement models an approach was considered to match each entry in the dataset indexed by date time with the load value at the displaced date index. This approach operating sequentially on scalars would have had at worst case O(N) time complexity. Instead, a more performant approach that took advantage of Pandas data structures being built on arrays that can take advantage of vectorized functions. The load values of data entries to be forecasted were set as the entire array of the displaced load entries **see Figure 5.10.**

****

(a)

****

(b)

**Figure 5.10** The displacement model forecasting functionality (a) The predict function is called returns displaced load data used as the load forecast. (b) An excerpt from error\_model\_graph\_util.py of calling the displacement model functionality and using the return value as a load prediction.

The approach for constructing the linear regression forecasting models used the Scikit-learn library for training the model coefficients and intercept using the *fit\_to\_training* function. The trained coefficients and intercept are used with the input x variables to create forecast predictions using the *predict* function. The process for creating linear regression models that correct the displacement forecast is different from linear regression models with only explanatory variables and associated coefficients. Instead of training the model with the actual load forecast as the dependent variable, the dependent variable is the sum of a displacement model forecast subtracted by the actual load forecast. The load forecast is the displacement model forecasts subtracted by the model output. The current implementation is restricted to only certain constructions of linear regression models, and hence more complex models will require additional conditional logic to process to train the forecasting model and generate load forecasting predictions. Non-linear regression models will require their own folder for training and prediction. However, with the decision to use a common *predict* call to generate model forecasts this ensures the call for model forecast predictions made at the component level are consistent regardless of the type of model.

Forecasting Model State implementation

The stateless design of the system was proposed in **Chapter 4.1.2**, with the users list of models added being stored client-side. The approach to this was to use a hidden DIV in the layout to store a JSON representation of the list of models added. This div contains the specific data fields required to create load forecasts using the model:

* Model name
* Model type (either displacement or regression)
* Colour of the model plotted

Specific to displacement models:

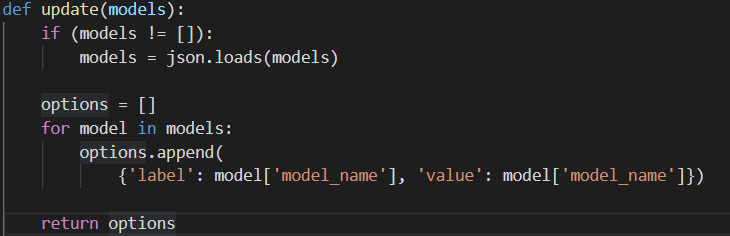
* Value of displacement
* Unit of displacement

Specific to linear regression models:

* Trained coefficients for input variables
* Trained intercept
* List of dataset columns to use as the input variables
* The column to correct
* Dataset column to correct forecasts for (‘null’ value if not a correction model)

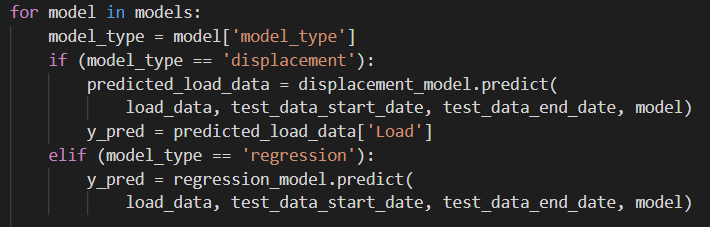
In the code the displacement and linear regression models in back-end functionality are loaded in the JSON structure and are then parsed to a Python dictionary representation. Separate DIVs were not used because it would require the merging of two data structures when displaying a list of models and allowing the users to remove a model, **see Figure 5.11.** The different functions that need to be called to create forecasts for a model for use in visualisations and statistical evaluations are determined through the model type of the entry in the dictionary, **see Figure 5.12.** The coefficients and intercept trained on the training data for a linear regression are included instead of the training data range. This was to reduce processing time by not retraining the model every time the linear regression model forecasts are plotted.

**Figure 5.11** The functionality that generates a list of dropdown items for the model selection dropdown that uses the model state as input.



Preprocessing Dataset Implementation

**Figure 5.12** Excerpt from functionality generating error statistics for test data. For each model the type of model is checked and the condition with the function that facilitates the creation of load forecast prediction using the type of model is called.



The functionality for preprocessing as designed in chapter **3.3.2** the dataset is contained within the *preprocessing* folder. The entry point to the preprocessing functionality is executing the *process\_csv.py* file which accepts console input, **see Figure 5.13.** There is no error handling for erroneous user input in the preprocessing program because the original dataset is not modified. The dataset is accessed, copied to memory, manipulated and then saved as a new load dataset to the user’s file system. The model configuration file was written in JSON. JSON was chosen as it was initially designed to represent data structures and associative lists of objects in JavaScript [9]. For the system, the models are the objects and the associated lists of objects are the model variables required to be added to the dataset. Furthermore, the JSONSchema Python library enables the parsing of a JSON file to a Python dictionary structure, eliminating the need to write a data mapping function.

**Figure 5.13** The process for preprocessing a dataset in the process\_csv.py file.

There was an unexpected error encountered when performing calculations on the entire dataset of displaced variables included in the configuration for linear regression models. There was forecasting of data entries without displaced data available. This occurred as the start date of the displaced values used for load forecasts is before the start date of the master dataset. To resolve this, a correction function was created that concatenated additional data to the start of the master dataset with the load and other numeric values being set to 0, **see Figure 5.14**. The calculation of displaced variables for linear regression models could then be performed over the entire dataset. However, a consequence of this is that linear regression models cannot accurately produce forecasts with the data that has zero variable values set as their displacement forecast. However, as the target user is an academic user it is expected they would understand the displaced input variables for linear regression models exceed the limits of the dataset.

(a)

26th April

27th April

28th April

29th April

30th April

1st May

29th April

30th April

1st May

(b)

difference = start date of the master data – start date of the displacement data

if difference > 0:

no. data entries = difference / time between each data entry in master data

dates = empty list

loop from 0 to the no. data entries as i:

add to dates: start date of the displacement data + i \* time between each data entry

concatenate dates to the start of master dataset

**Figure 5.14** The correction process to handle the absence of load values to calculate a displacement variable to include in the dataset. (a) Diagram showing a displacement of the start of the dataset by 1 day and the data not being available and the correction concatenating load data to the start of the dataset to use for displacement (b) Pseudocode representation of the correction function correct\_displacement\_start\_date\_not\_in\_data.

**Citations**

[1] <https://www.kdnuggets.com/2015/05/r-vs-python-data-science.html>

[2] <https://www.anaconda.com/understanding-conda-and-pip/>

[3] <https://towardsdatascience.com/introduction-to-interactive-time-series-visualizations-with-plotly-in-python-d3219eb7a7af>

[4] <https://dash.plot.ly/introduction>

[5] Appsilon tech talks <https://www.slideshare.net/appsilon/tech-talk-overview-of-dash-framework-for-building-dashboards?from_action=save>

[6] <https://chaseonsoftware.com/most-common-programming-case-types/>

[7] <https://www.w3.org/TR/css-flexbox-1/>

[8] https://python.swaroopch.com/oop.html

[9] <https://www.sitepoint.com/json-vs-xml/>

**Appendices**

[ {

"model\_name":"Test Model",

"model\_type":"regression",

"model\_color":"#2ca02c",

"coefficients":[

-0.5944204249699635,

-0.8933426064242177

],

"intercept":-2.261462109133584,

"x\_columns":[

"Temperature Last Day",

"Wind Speed Difference Last Day"

],

"corrected\_column":"Load Last Day"

}

]