ANALYSIS OF LARGE-SCALE DATA SETS

Computer Science NEA Project

This project aims to provide a data graphing and manipulation software for data scientists, which utilizes the graphing capabilities of Python's Matplotlib, and a Tkinter-constructed GUI.

Callum Wilkie

Candidate Number:

End of Contents

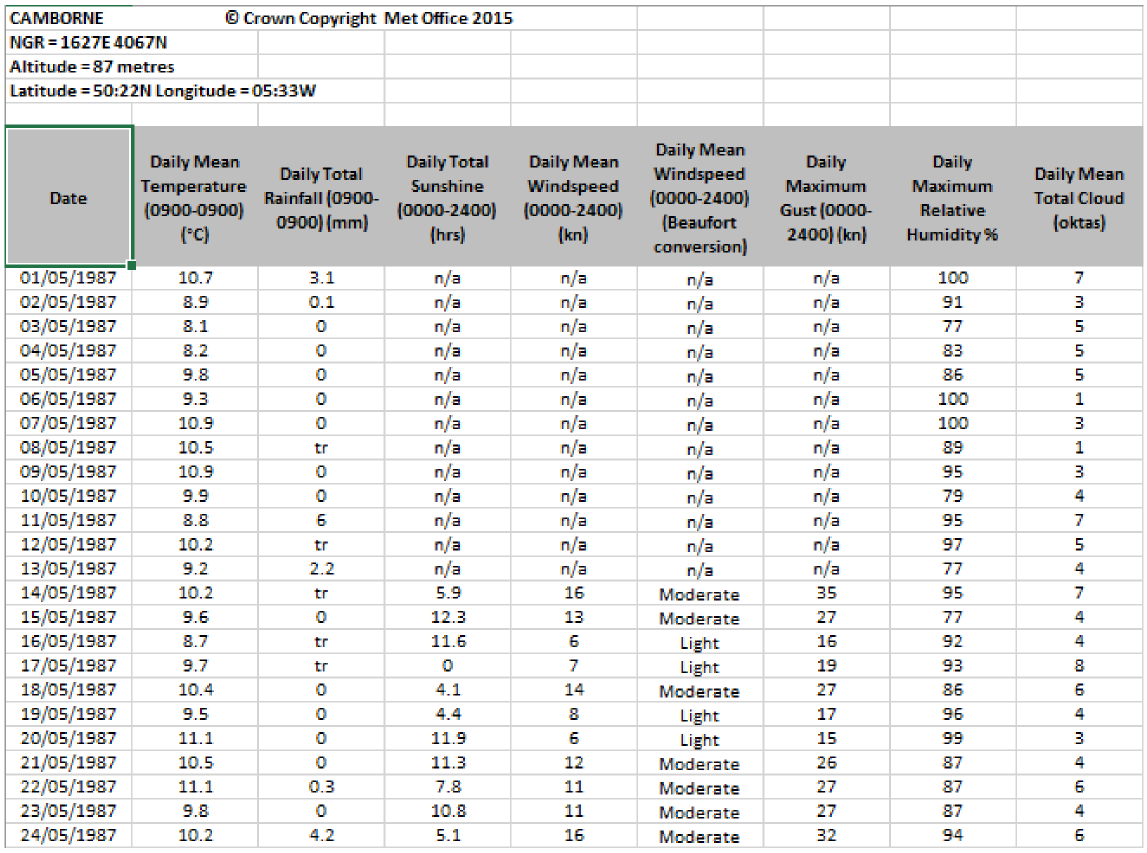
The Large Data Set
https://qualifications.pearson.com/content/dam/pdf/A%20Level/Mathematics/2017/specification-and-sample-assesment/Pearson%20Edexcel%20GCE%20AS%20and%20AL%20Mathematics%20data%20set%20-%20Issue%201%20(1).xls

Taken from the Introduction on the first page,
"Pearson have provided this large data set, which will support the assessment of Statistics in the A level Mathematics Paper 3… The data set consists of weather data samples provided by the Met Office for five UK weather stations and three overseas weather stations in the time periods May to October 1987 and May to October 2015."

Whilst the Data Set was initially intended for teaching Statistics in Mathematics it is also an easily accessible source of raw data, that can be analysed by a python program. This is why it was selected for the Computer Science NEA Project as the primary piece of testing data. It contains multiple sheets of varying statistics and values from which I can test the efficacy of my program. Despite this being the data upon which the program was both inspired by and designed to utilise, I aim for the program to be capable of using numerous compatible files of datasets.

With regards to the 'Large Data Set', it is an excel document with multiple pages covering extensive weather data during the months of May to October. International locations are selected were selected to record data from in the years 1987 and 2015, resulting in the potential to compare two sets of data per City. This data may incorporate the following:

- Date
- Daily Mean Temperature (Celsius)
- Daily Total Rainfall (mm)
- Daily Total Sunshine (Hrs)
- Daily Mean Windspeed (kn + Beaufort)
- Daily Maximum Gust (kn)
- Daily Maximum Relative Humidity (%)
- Daily Mean Total Cloud (oktas)
- Daily Mean Visibility (Dm)
- Daily Mean Pressure (hPa)
- Daily Mean Wind Direction (o)
- Cardinal Direction
- Daily Max Gust Corresponding Direction (o)
- Cardinal Direction (of DMGCD)



CAMBORNE © Crown Copyright Met Office 2015
NGR = 1627E 4067N
Altitude = 87 metres
Latitude = 50:22N Longitude = 05:33W

Date	Daily Mean Temperature (0900-0900) (°C)	Daily Total Rainfall (0900-0900) (mm)	Daily Total Sunshine (0000-2400) (hrs)	Daily Mean Windspeed (0000-2400) (kn)	Daily Mean Windspeed (0000-2400) (Beaufort conversion)	Daily Maximum Gust (0000-2400) (kn)	Daily Maximum Relative Humidity %	Daily Mean Total Cloud (oktas)
01/05/1987	10.7	3.1	n/a	n/a	n/a	n/a	100	7
02/05/1987	8.9	0.1	n/a	n/a	n/a	n/a	91	3
03/05/1987	8.1	0	n/a	n/a	n/a	n/a	77	5
04/05/1987	8.2	0	n/a	n/a	n/a	n/a	83	5
05/05/1987	9.8	0	n/a	n/a	n/a	n/a	86	5
06/05/1987	9.3	0	n/a	n/a	n/a	n/a	100	1
07/05/1987	10.9	0	n/a	n/a	n/a	n/a	100	3
08/05/1987	10.5	tr	n/a	n/a	n/a	n/a	89	1
09/05/1987	10.9	0	n/a	n/a	n/a	n/a	95	3
10/05/1987	9.9	0	n/a	n/a	n/a	n/a	79	4
11/05/1987	8.8	6	n/a	n/a	n/a	n/a	95	7
12/05/1987	10.2	tr	n/a	n/a	n/a	n/a	97	5
13/05/1987	9.2	2.2	n/a	n/a	n/a	n/a	77	4
14/05/1987	10.2	tr	5.9	16	Moderate	35	95	7
15/05/1987	9.6	0	12.3	13	Moderate	27	77	4
16/05/1987	8.7	tr	11.6	6	Light	16	92	4
17/05/1987	9.7	tr	0	7	Light	19	93	8
18/05/1987	10.4	0	4.1	14	Moderate	27	86	6
19/05/1987	9.5	0	4.4	8	Light	17	96	4
20/05/1987	11.1	0	11.9	6	Light	15	99	3
21/05/1987	10.5	0	11.3	12	Moderate	26	87	4
22/05/1987	11.1	0.3	7.8	11	Moderate	27	87	6
23/05/1987	9.8	0	10.8	11	Moderate	27	87	4
24/05/1987	10.2	4.2	5.1	16	Moderate	32	94	6

Furthermore, the data perfectly represents a common issue with large-scale datasets - there is an issue regarding efficiency of manual analysis. This data is difficult to manually read, let alone compare with other cities and timeframes – this being due to the vastness of available information. Searching manually for patterns and trends will take a large amount of time to cover the entire data set, or valuable areas.

Prior to my programming, the document obtained from was altered for ease of use. The Informational first page has been removed, as it provides only a background on the data used within the Large Data Set. This does not affect the program – the first informational sheet could be ignored anyway through the import method's functions. in the selection of other sheets to read.

Benefits of a Computational Approach

Computers are capable of reading, identifying, and outputting data in a manageable form with great speed. This results in far greater efficiency in analysing data through the use of this program than being done by eye or other lengthy means. Furthermore, complex calculations incorporating this data – which may include extrapolating and interpolating data values, or alternatively producing mean values.

In particular, the aims of the program include the creation of mean data spanning multiple useful ranges and weekly (or x days) rolling averages. furthermore, raw data can be quickly transformed and outputted as a digital graph – making the understanding of data and comparisons far more efficient by eye. Alternatively, these comparisons can be carried out by the computer, and provided to the user in multiple forms – graphical or data.
Computers efficacies lie within data analysis and so it would be a misplaced notion to not employ them for this purpose. Such are the benefits to a computational approach within this program.

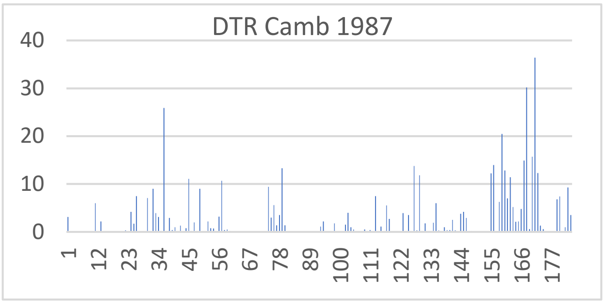
Stakeholders Within this Project

Through this python program, I aim for a large variety of data sets to be downloaded, read and processed in an efficient and understandable manner. It should be feasible for a data scientist researching any field with viable quantitative (measurable or numerically-based) data to use this program. The program should also provide suitable graphing and calculation abilities to be desirable for both amateur and professional researchers looking for an easy method to analyse their data.

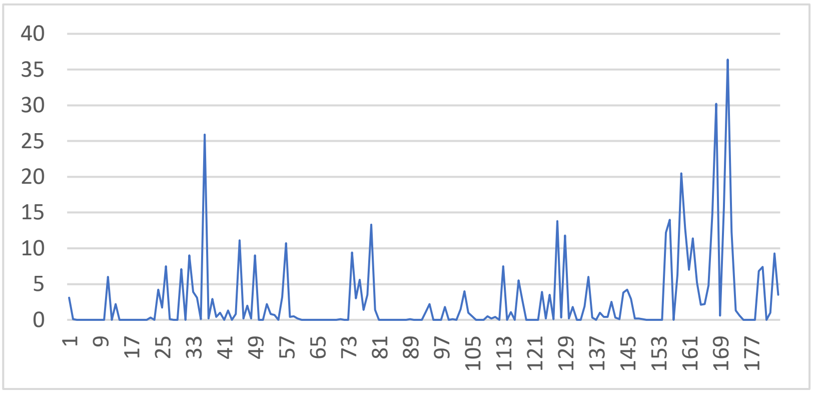
Whilst primarily I aimed for Data Scientists performing weather or climate research to obtain the most use from this program, I found the program would be much more useful if it could be applied to a broad range of data sets than only the Large Data Set – as identified as test data previously. A program limited to only a single data set holds little value, especially as data scientists value having access to large swathes of data to analyse for patterns and trends.

Thus, my stakeholder group expands to be for those with interests in extensive data research and long-term trends or patterns.

Excel Spreadsheets



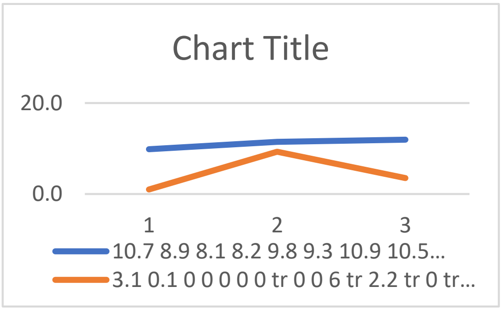
Within an excel document itself, data can be transformed into a graph of multiple forms. This includes pie, bar and line – line graphs being especially useful. Shown right are two graphs produced inside of excel from data in 'Daily Total Rainfall (0900-0900) (mm)' from Camborne May-Oct 1987. Top is a bar graph, and below is a line graph. Both are effective in displaying the data held in a comprehensible format.

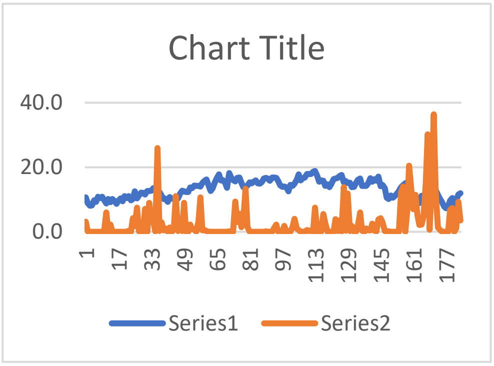


These graphs are produced near immediately (dependant on the size of the data) with pre-set axis values. The X-axis is set to the days that the data covers, and the Y-axis as the Daily Total Rainfall value. Additionally, quality detail on the axis is altered by shrinking / expanding the graph – shown by the comparison of the two graphs on the right.

Observing both graphs, some may find the bar graph makes for a better visual representation of the data in this instance. The line graph is suitable; however it crosses over into dates where the position of the line does not accurately represent the rainfall of the day. Despite this, it is effective in showing change over time, which may make it more suitable for a rolling average instead. However, others may find that they prefer the line graph for its frequent use in data representation, and applicability to numerous sets of data and data values. I aim to allow for users to have multiple potential graphs that they can use, in order to best suit the situation they encounter.

Furthermore, the excel graph is capable of increasing and decreasing the shown level of detail. However, I will not incorporate this into my program as I believe it is an unnecessary inclusion in the excel program made to benefit the graph being used in other Microsoft programs. But, it does show an effective way of allowing data to be read in high detail without giving every single data point. With all 183 datapoints, not all can be visibly titled on the x-axis. A 7day group may work better as it would result in a little under 30 points to mark on the graph. Ultimately, this will depend on the size allotted for the graph, and will require further thought during ongoing program development.





Despite both examples utilising the Date and another data column, this does not have to be the case. Although a little less obvious to produce (the top left graph being the initial attempt at graphing the data) I could produce a graph comparing Daily Mean Temperature and Daily Total Rainfall. The failure of the first graph may be due to uncleaned data which still contains *tr* (trace values), rather than decimal values. Regardless, the second edition displays something I should aim to achieve in my own program.

The flexibility of comparing any two data columns in excel should be achieved in my own program. Furthermore, excel provides a "suggested graphs" function which was used to produce the line graph shown on the left. Whilst such a feature would likely be both difficult and unnecessary to implement into the program, there are alternate features such as a second axes which likely could be implemented (these are applied to the opposite side of the current y-axis).

Python Libraries

Python is a programming language already in heavy use amongst Data Scientists due to its versatility, free use and large additional libraries. Furthermore, the simple syntax and language makes it an easier language for new learners. This will be upheld by my program being coded in an organised and understandable manner.

Of the additional downloadable libraries, **matplotlib** and **Pandas** will come under heavy use for the program: Pandas allows for reading and writing not only its own created 'Dataframes' but also excel documents and CVSs – a common method of storing data. Pandas also can select columns or rows to output or modify using the 'replace()' function. Pandas utility for managing datasets will be imperative for user interactions with the data before making calculations with it. Furthermore, URL links can also be inputted into Pandas and data read from these files. Due to this, options to read already downloaded files or enter an internet URL will be included for the user. URL only will not be included, as pandas may be incapable of passing through some security checks such as required login info, or unable to download data from some websites.

Graphical outputs will be created with Matplotlib. This library allows for a large variety of plots for data imported into the program. These graphs are essential to the program, and being capable of producing high-quality and accurate graphs is capable using this library. Furthermore, the user can modify a graph's style, appearance and incorporated data. Important for the program is the ability to have multiple lines in order to display a comparison graph – comparisons of past and present data is incredibly useful for data scientists. Whilst the library is capable of even 3-Dimensional plots, these are likely to be unused for the program as no greater insight is provided with their use (in comparison to 2-Dimensional plots)

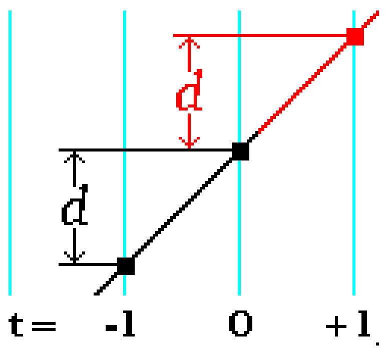
Data Analysis

My aims for the data can be split into roughly three categories. Those are:

- Outputting + Analysing Data
- Creating / Predicting new Data
- Assessing accuracy of the data

The analysis of the data will be relatively simple, and so this research will focus upon the last two categories.

Extrapolation is to assume that existing trends will continue or a current method can remain applicable. In the case of the program, findings from the current data set will be assumed to continue beyond May to October of 1987 or 2015. A data scientist may be wishing to see if actual data records in 2005 match estimates made by the program.



Extrapolation works best the closer to the given range of information, and so it may be worth adding a disclaimer to alert users should they enter a region of low accuracy. Being able to accurately tell the reliability of the extrapolated data would also be useful – although they would likely be the most accurate during May to October months as this is where our data is situated. Due to this, I will limit data predictions to these months, but without limit on years.

From the diagram, a rough visual representation of a potential method is shown. By identifying the difference between two values, it can be added to find the following unknown value. This difference may be a constant value or a proportional change. However, this is a heavily simplified diagram and extrapolating decently accurate data will be much more difficult. General trends within the data must be recognised and employed. Furthermore, a lack of knowledge outside May to October heavily hinders predictions. Due to this, it may be worth using additional weather data.

Generally, more data leads to better results. The data being used could also be cleaned to remove anomalous results, however it is large enough that the sparse anomalies shouldn't affect our data to a noticeable amount. Furthermore, extrapolated data could be reused with original data to make further predictions. Although this could lead to compounding errors. Heavy management of the extrapolated data would be required if it were to be extrapolated again.

Furthermore, how this new data is to be stored must be decided. It could be added onto a new sheet entitled "[Location] Prediction [Year]" as an example. This would allow for rapid and simple use of the predictions – the user can graph the data easily too as it would be part of the main data file. Alternatively, the data could be stored in the program. However, this may require it being recreated frequently, which would be time wasted. Another option would be a secondary excel document to store data predictions. This retains the original data and keeps it separate from any predictions. Furthermore, predictions can be stored and quickly reobtained later.

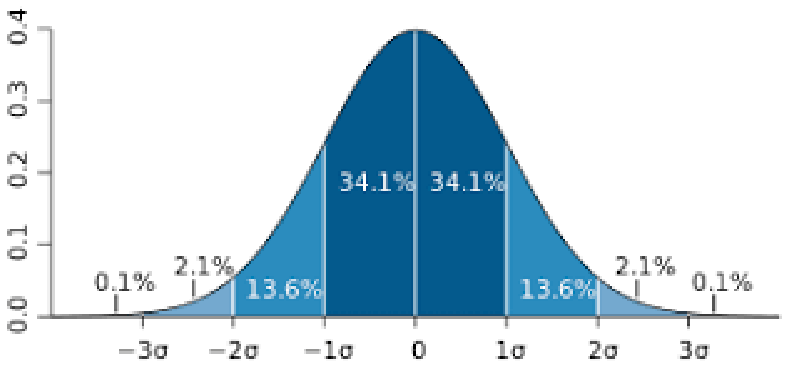
Checking the quality / accuracy of data

Since extrapolation is being used, assumptions must be made with current data. Predicted data values assume a constant pattern that is unaffected by future events. However, this frequently is not the case in reality. Despite this, our extrapolated values should remain accurate to a limit. Furthermore, accuracy of predictions can be improved by acknowledging potential flaws and dealing with them before issues arise.

Limiting how much values can change is a potential measure, and this can be done by using a Normal Distribution to observe the likelihood of extrapolated data values. A normal distribution can be applied as it is a reliable model for making predictions about data, including weather. Using a minimum level of significance – such as testing data to a 5% or 10% level, or x standard deviations limit could be used to ensure extrapolated data is within feasible possibility, else it can be rejected.

The Normal Distribution

Using the median μ and standard deviation $\sigma$of a set of data, a normal distribution can be produced using the model $X \sim N(\mu, \sigma^2)$. Its function produces an integral with an area of 1, allowing for it to be easily split into probability values as shown in the diagram. These probability values centre around the distribution using a "Probability Density", whereby an area of the graph maps a corresponding probability. Whilst the standard normal distribution is modelled with $\mu = 0, \sigma = 1$, these values can be changed to suit the needs of the user. The axis of the function are shifted in order to match with the inputs, however the area always remains the same.
Alternative names of the Normal Distribution are the Gaussian Curve or Bell Curve, used for a multitude of purposes such as IQ tests or setting grade boundaries for exams. The distribution is especially good with large sets of data like the Large Data Set, so it can be employed effectively within my program.



As for producing a normal distribution, variance and standard deviation can be gathered using the same formula, with only a small modification for standard deviation:

$$\sigma^2 = \frac{\Sigma(x_i - \overline{x})^2}{N} \; where \; \overline{x} = \left(\frac{\Sigma x}{N}\right)$$

Alternatively, a more simplified form can be used: $\sigma^2 = Mean\ of\ the\ Squares - Square\ of\ the\ Mean.$
This formula is used to find variance, typically written as σ^2. First, the mean of the data is derived using the $\overline{x}$ formula. The total sum of the data, with x being the data and Σ representing sum, divided by N values within x. then for each value in x, the mean $\overline{x}$ is subtracted from it and the result squared. The mean of these values is what produces the variance σ^2.

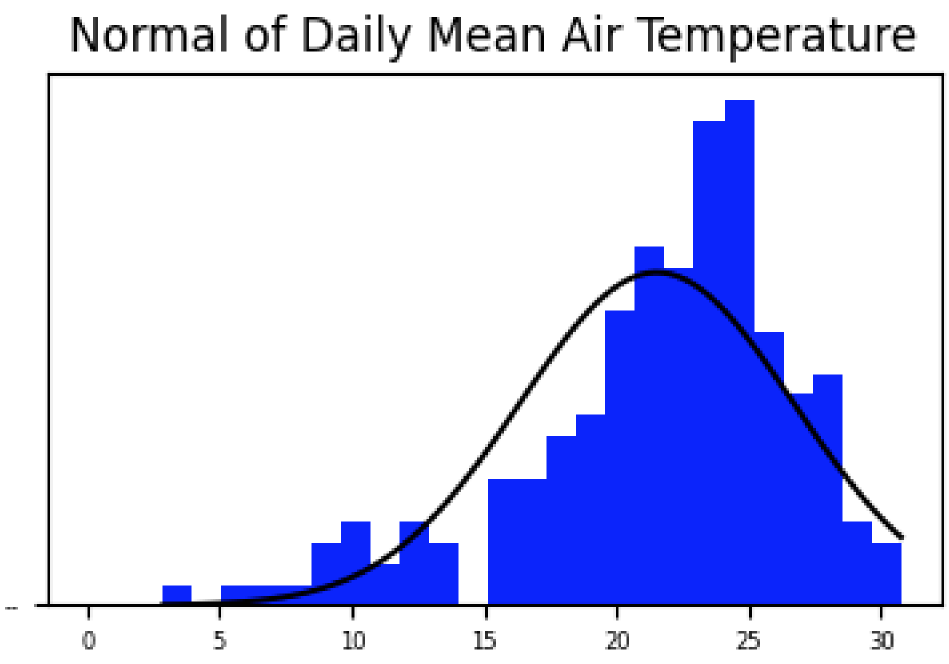
As for the standard deviation, that is the square root of the variance $\sqrt{\sigma^2} = \sigma$. The standard deviation is obtained despite normal distribution being modelled using: $X \sim N(\mu, \sigma^2)$. This is due to the normal distribution formula using the singular σ. Furthermore, both variance and standard deviation can be outputted for the user should they desire those values.

And to find the probability of a given x value, where $P(X = x)$ then the following formula is applied:

$$f(x) = \frac{1}{\sigma\sqrt{(2\pi)}} e^{-0.5\left(\frac{x-\mu}{\sigma}\right)^2}$$

Using the values of x, μ, σ the function produces a probability value equivalent to the probability of data value x occuring. This can be used to assess the validity of data predictions.

For the program, rather than utilising the function directly, I will employ the python library Scipy, which provides the statistical function as part of its library. All that is required to be calculated is the mean and standard deviation, which is then passed into the function.



Here, the Normal Distribution is modelled using a mean and standard deviation derived from the data stored within the Daily Mean Air Temperature of Beijing in 1987. This is then given to a solid black line, overlayed onto a histogram constructed from the same data. This comparison aims to present the user with a view of how well the data matches the assumed normal distribution, and displays regions where irregulatities may be occurring. However, this method of catching anomalies is not always perfect. For example, using collective weather data over a timeframe of multiple months subjects the data to seasonal weather patterns, which will cause inevitable deviations from the normal distribution. Hence, the user will have the final say in what values are removed from the stored information, rather than having the program do it all automatically.

What is Essential?

Via the analysis of these programs, methods, and their features, I have identified numerous desired features that would be beneficial to incorporate into my program. Whilst it may be that the core features of this program will simply be the ability to read an excel document and output the data read from it, the potential uses will be of great use for any Data Scientist.

Additionally, there should be features present that go either beyond other programs' capabilities or alternatively carry out the users demands in a more efficient manner. With the program having a specialised purpose (to analyse data sets) then increasing the efficiency and ease of use would be the more readily obtainable goal. Some of the benefit from using this program will also derive from it being written in Python, whereby the users can have more direct modification of the program compared to something like Excel.

General Programming Desires

This focuses upon features desired within the program's code – be it in creating understandable code which is understandable for another programmer, implementing higher-level methods in my coding, or increasing efficiency. Data being used by the program would need to be reformatted and cleaned into a standardised form. This must be done as data sets on programs such as Excel may come with non-essential sheets of text, and removable footers or headers. If these are not removed, it would likely result in the program mistakenly identifying text as data and potentially crashing when calculations are attempted. The user would have to do so manually when opening a new data set for the program to read. The program should allow for the user to edit a document via removing footers, headers, and the likes before continuing to analyse the data. For example, the Large Data Set contains 5 lines of header text on each sheet that requires removal before processing. Displaying a preview of the data will assist the user in ensuring the data is fit for analysis.

Data being read from the program would do best stored in a standardised form, allowing for all subroutines to read and manipulate stored data efficiently, and without much modification between subroutines. This can be done by using a designated list / array for the data, alongside adhering to a particular formatting layout for the data. This may include the order of stored values in a list, or reducing the need to alter datatypes throughout the program.

Furthermore, this will reduce difficulty in managing the program for myself and increase the understanding of others observing my code. Should an unknown issue arise, a person other than myself may be capable of creating a fix for it themselves. However, it would be best to avoid such flaws in the program for optimal user experience. A User may also wish to introduce their own features into the program. By having the code understandable and its purpose evident this is made far easier.

Additionally, the use of higher-level programming techniques can provide an evidence of greater programming skill. I personally would like to find a way to incorporate a class, however the nature of the program makes this difficult. Alternative methods, like dictionaries and multi-dimensional arrays can be used instead. Furthermore, increasing the efficiency of the code will display good programming skills. Keeping it clean, organised and understandable is a must.

User Interface Features

These features concern the user interface, and making the program as user-friendly as possible. To start with, the user interface should remain simple and easy to use. Complex features being added need to be managed well in order to avoid user confusion on what they do. As more features are incorporated, they should be managed neatly and organised in an easy to understand manner. Looking towards the Microsoft program's toolbars at the top of the screen could make for a good influence.

Whilst I intend to use multiple windows in order to have greater organisation of the program and increase user accessibility, I must take care to control the number of windows present on screen. This should be done in order to avoid unnecessary clutter that makes the program less efficient in handling and use. What could be implemented is a maximum number of open windows related to the program, and oldest windows are removed upon opening a new window.

Data Features

These features will focus upon the data obtained from the Large Data Set and how the user wishes to use it.
With regards to graphing data, the user should be prompted to either produce a graph of data or a comparison graph. With a simple data graph, only 1 set of values from the Data Set will be involved. However, a comparison graph will incorporate 2 sets of values. The user may choose to have both sets drawn onto the same graph for visual comparison, or a single line drawn onto a graph showing a + or – change in values in the same timeframe.

Furthermore, the user should be capable of having more advanced modifications made for graphs rather than a basic (x, y) axis. The program should be capable of using timeframe averages – like a weekly average – to produce graphical output of the data. Alongside this, the program should be able to perform a number of alternate functions upon the data, such as finding Ranges, Median and Mean values for chosen categories.

Additionally, I would like for the program to be capable of making predictions and assessing the validity / likelihood of those predictions. Alongside those predictions, I wish to produce algebraic formula that describe the data within the Large Data Set – likely complex functions that can assist in predictions, or be used in a line graph with mathematical modelling. Deriving such functions for the data will adda unique complexity unseen elsewhere in the program. Producing those modelling formulae and accurate predictions will likely be the two most complex features to implement accurately.

Other Desired Features

Capability to analyse alternate data sets to the Large Data Set. This will require the program to identify any unnecessary text at the header and footer of the excel document and remove it. Alternatively this can be controlled by the user, which may avoid the program making mistakes. By allowing for alternate data sets, then the program will have far greater analysis capabilities. Furthermore, it will improve the predictions made by the program thanks to a greater number of data sets.

The Program could provide default formatting settings, of which the user can change or alter as they require. This should include header and footer settings to remove non-essential text as seen in the Large Data Set, and the ability to determine which Sheets, Columns and Rows the user would like to import. This may also open up a small additional potential for the program to be capable of extracting desired segments of data from a data sheet and saving them separately for ease of use. With this, the user would be able to manage their data with ease, and omitting non-essential or undesired data will save them time. Whilst producing graphs of specific data is possible with Excel, my program will do so better.

Greater access to data will also be favoured by the stakeholders. Employing a wide range of data is crucial for scientific studies and avoiding research bias or anomalous results.

Limitations and Requirements

Avoid using Excel beyond source of data?

Limitations of the program can derive from sourcing the data to use with it. Some data may be stored in an unowned program such as Excel – like the Large Data Set. Whilst the program is capable of avoiding necessary purchases due to Python being a free language, some data sets may be inaccessible. Furthermore, the user would require an internet access in order to download data. However, for the stakeholders in this project it is most likely that access to Excel (and similar data software) is possible. For a non-Excel owner to use this program, it may be possible to utilise an alternate storage method for data – such as csv files, which can also be read into the program. As for obtaining it from Excel formats, one method for online sources is for the program to connect directly and copy the data from the webpage. However, for online-hosted downloadable files like the Large Data Set, this may not be possible. Instead, the user would have to download it themselves into alternative formats, should a direct connection be unsuccessful.

Furthermore, the program will likely store data into it's own Excel document – so that data stored within it can be read immediately upon loading the program, rather than having the user re-load the data each time. An Excel file would be used for this purpose due to its ability to hold multiple worksheets of data, rather than a csv's limitations of only 1 sheet, and thus 1 dataset per file.

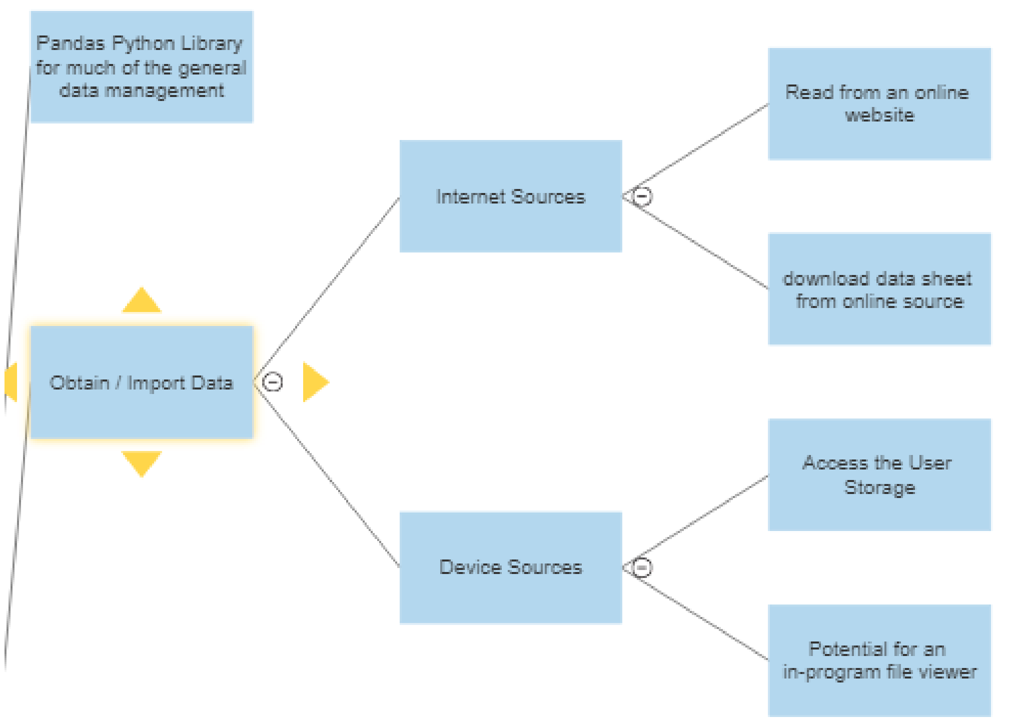
Decomposing the Problems

Breaking down this program into more simplified segments has been done in order to better my own understanding of the process to be taken in order to construct it. Each individual segment is responsible for a crucial element of this data analysis program and must be done correctly.

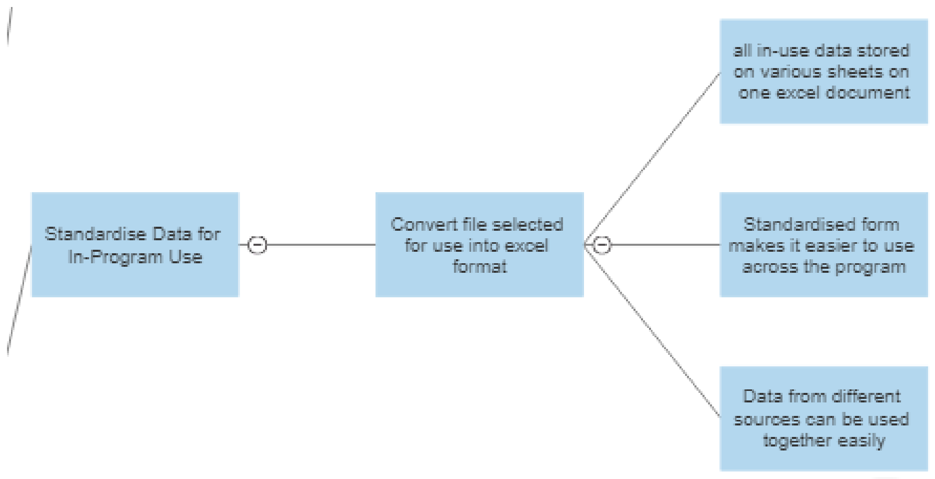
Additionally, the program will incorporate a vast quantity of variables for storing each individual section's various settings and management options that the user can interact with. A general overview of what will be used has been included with each section.

Importing Data

Of all of these segments, the most important steps regard the Importing of data into the program and the Standardisation of that data into a workable form. Data will be able to be introduced into the program either via reading it from an online link, or by accessing data stored on the device. Stored data sets will be the most likely method of importing data into the program – and online data sets can be downloaded onto the users' device - and so this shall receive the most attention in development. An file viewer function can be constructed by utilising the devices' own file viewer. This results in the only issues resulting from device-specific issues and requirements can be avoided or accounted for.


Pandas Python Library for much of the general data management
Internet Sources
Read from an online website
download data sheet from online source
Obtain / Import Data
Access the User Storage
Device Sources
Potential for an in-program file viewer

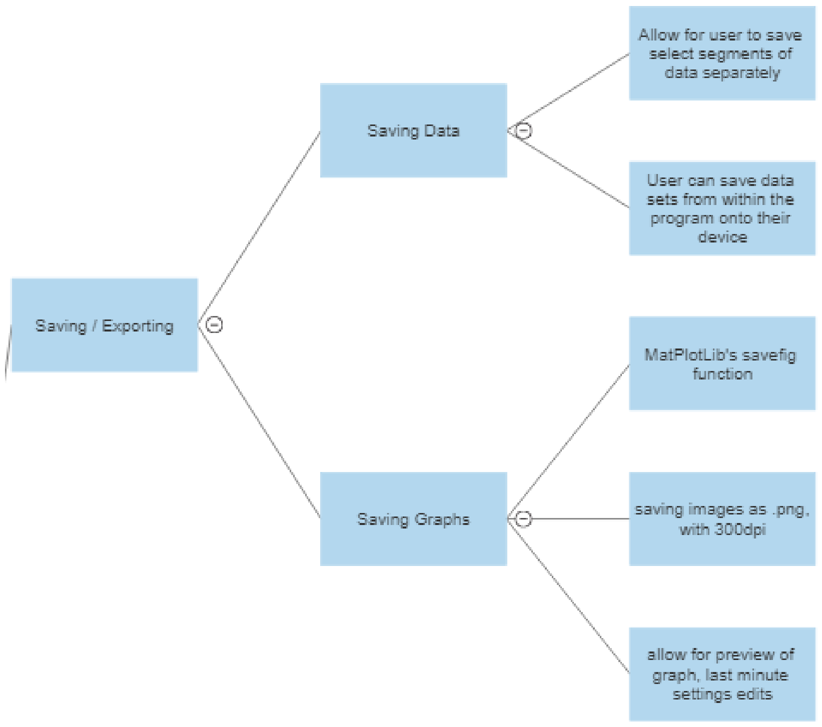
For the standardisation of imported data, all data will be converted into the same excel format and saved onto a singular spreadsheet file. This is to act as the program's "memory" whilst in operation. By utilising a common design and layout with the data, it will make my program easier to code and it can be optimised to fit the format of the data. Furthermore, dealing with organising data the moment it is imported to the program will save time and hassle later on, in case the user forgets to alter or manage the data set themselves. The standardisation of the data will also involve the setting of a header and footer for excel documents, to remove unnecessary descriptive additions, and removing dataset-specific identifiers for relevant values. This may include "tr" for trace, as an example from the Large Data Set.


Standardise Data for In-Program Use
Convert file selected for use into excel format
all in-use data stored on various sheets on one excel document
Standardised form makes it easier to use across the program
Data from different sources can be used together easily

The Variables required for importing data will not need to be considered with regards to the data itself. Pandas dataframe object will be responsible for dealing with imported data, and is more than capable of sorting any encounters with string, int or other data types in the data sets. What would need to be considered istead is the variables in which the Dataframe objects are being stored. For the program, I would like to utilise simple lists to store each dataframe, or any necessary information required to make them function. As an example, the data set in the process of being imported by the user (that is, currently undergoing modifications before the user finalises the import) should be stored in a separate variable for ease of use, and then appeneded to the greater dataset storage list when finalised.

Saving and Storing Data in the Program

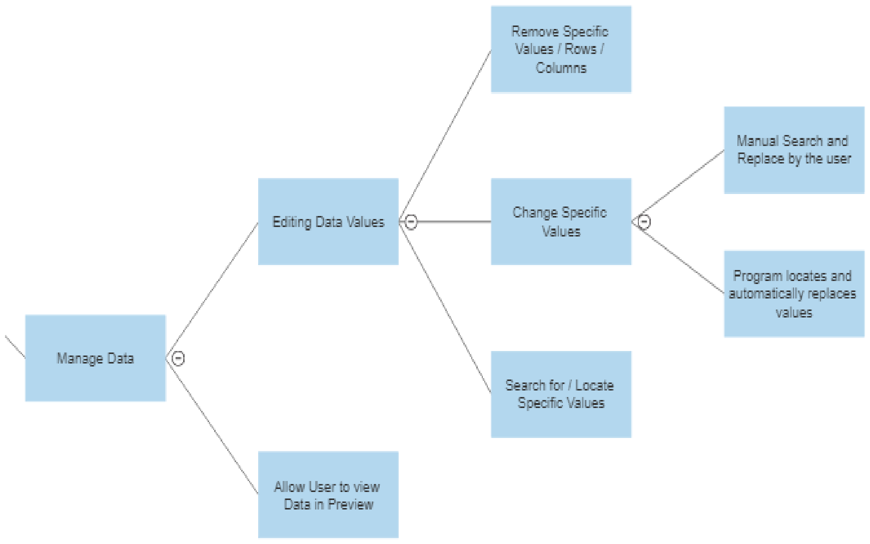
After having imported and cleaned data being used in the program, one of the next options available to the user will be to save parts of, or a whole data set. This option is provided for pure data, as the user may wish to save a cleaned set of data to avoid having to manually alter it on the next use of the program. Furthermore, the user will be able to select particular columns, or individual sheets to save – rather than the entire thing at once. These features are included should the user wish for important values to be more readily separated and confined. “Save Data” and “Save Graph” will be identified as two separate options within the program, and their purpose made evident by their name.


Saving / Exporting
Saving Data
Allow for user to save select segments of data separately
User can save data sets from within the program onto their device
Saving Graphs
MatPlotLib's savefig function
saving images as .png, with 300dpi
allow for preview of graph, last minute settings edits

For Saving Graphs, Matplotlib comes equipped with a function designed around creating a .png (or other file formats) of the graph. This function will also be a required incorporation as the interactive graphs will likely be disabled. When the user goes to save a created graph, options will be presented for some alterations. One of which will be the dpi value (dots-per-inch). A resolution of 300 will be recommended due to it being the minimum for a professional research paper – which a Data Scientist may be producing with the aid of this program. However, he program will allow for increasing or decreasing this value. However, much of the alterations to the appearance of the graph will be done during its creation.

Managing Data

Prior to producing any graphical outputs or utilising any formulas, the user will be presented with Data Management Options. This will regard the user's ability to edit data values, and also a QoL feature for the user to be able to view the Data Set in an organised form – representing how the program will read and view the data.

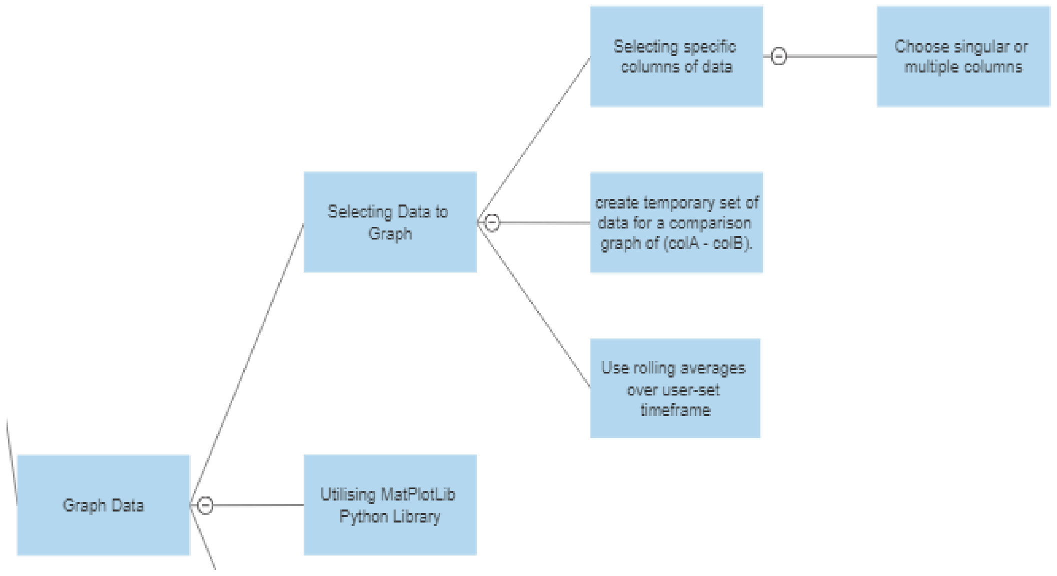

Remove Specific Values / Rows / Columns
Manual Search and Replace by the user
Editing Data Values
Change Specific Values
Program locates and automatically replaces values
Manage Data
Search for / Locate Specific Values
Allow User to view Data in Preview

For editing data values, the minimum requirement is that the user is able to access any individual cells and edit or remove the data from the data set. Beyond this requirement, I would like to incorporate features to simplify the process of removing or altering larger portions of the data set through searching for specific words, data or values. This will allow for the user to specialise the data set and omit unnecessary values that are not important to their investigation. Furthermore, the program will implement warnings should the user attempt to remove large quantities of data – in order to make sure that the process is intended. The program can also be used to speed up the process of managing data by using algorithms to find and replace user-selected values. For instance, replacing the "tr" (trace) value for a 0 in rainfall data. If a particular data scientist is researching recent rainfall and comparing it to previous years, they may wish to remove non-rainfall data from imported data sets in order to reduce micromanaging graphs and formulae later on.

Graphing Data

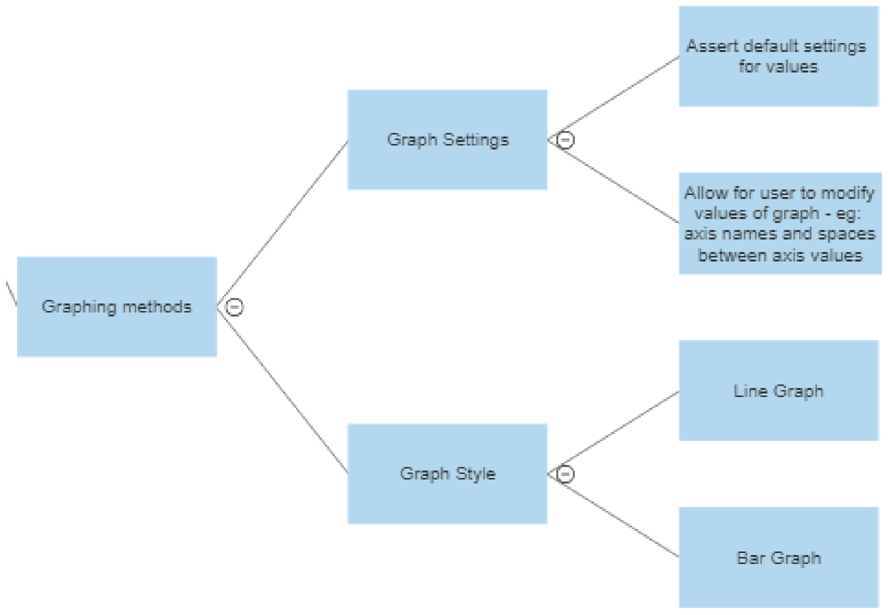
Either graphing data or utilising in-program formulae are likely to be the next step of the user. With the production of graphs, the Matplotlib Python Library will be used for its in-built graph plotting features. The user must select data they desire to produce a graph of, or multiple columns of data for a comparison graph.

Alternatively, comparisons can be made between data and the increase or decrease in values over time can be graphed. Furthermore, rolling averages over a timeframe will be another option – this may be rainfall across 7 (maximum value) days – with 2 as the minimum for using this method. The user may also wish to see the total amount of an item, the cumulative total of rainfall during a timeframe, and this shall be incorporated as an additional option too. Increasing the utility of the graphing function will make it more useful to the stakeholders of the project.


Graph Data
Selecting Data to Graph
Utilising MatPlotLib Python Library
Selecting specific columns of data
Choose singular or multiple columns
create temporary set of data for a comparison graph of (colA - colB).
Use rolling averages over user-set timeframe

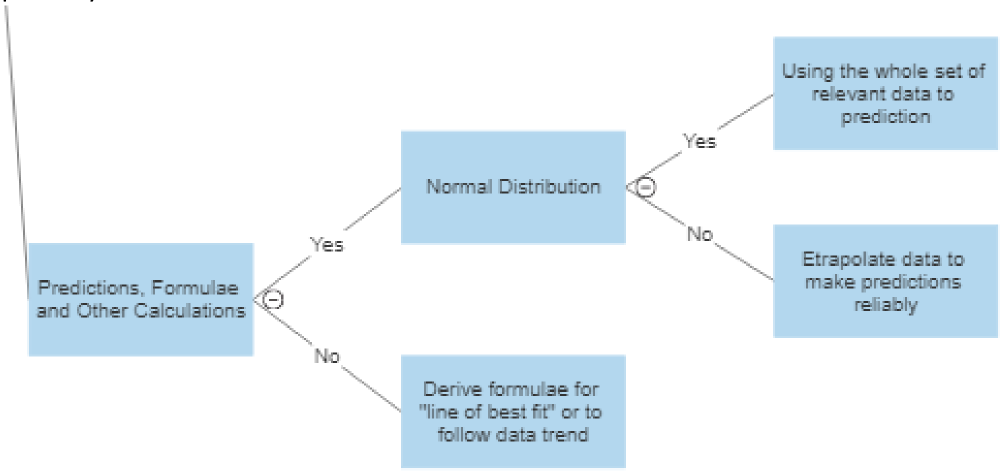
From here, what remains of the Graph is selecting the right method, and making visual modifications. Line and bar graphs are two simple and necessary graphs to be incorporated in the program. In Particular, line graphs are very effective when dealing with data, whilst the bar graph is offered as an alternative style choice. Options like the Pie Chart will not be included, as they are not as affective at displaying large-scale numerical data. Data scientists desire efficient and understandable graphs in order to make quick visual observations that are not as readily visible in numerical data form.

As mentioned briefly, visual modifications will be allowed. With this, the user can alter the names of axis, frequency of axis markings, and the colours of the graphs – which is intended for those who are colour-blind and may have difficulty differentiating between some colours.


Graphing methods
Graph Settings
Assert default settings for values
Allow for user to modify values of graph - eg: axis names and spaces between axis values
Graph Style
Line Graph
Bar Graph

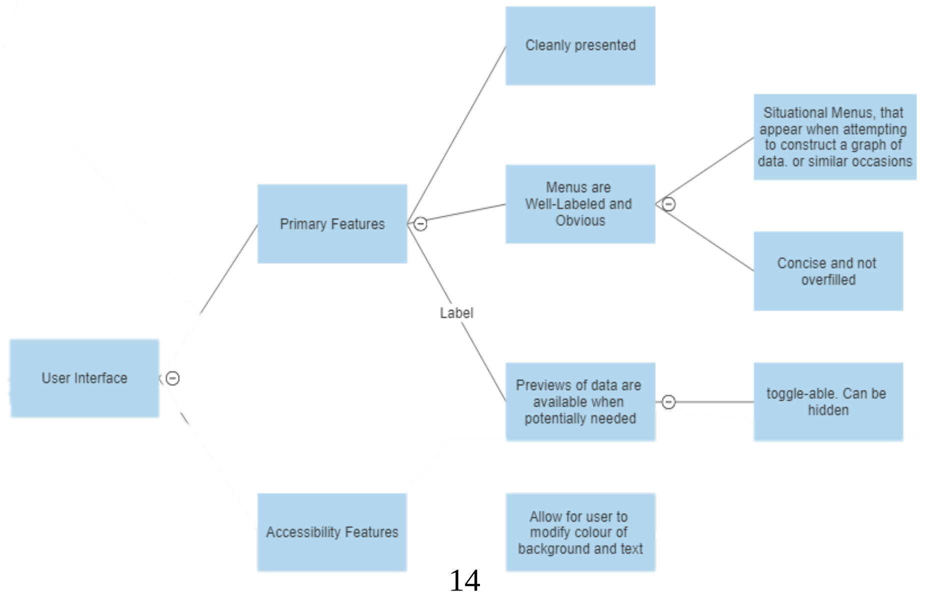
Formulae and Calculations

Formulae is where the Normal Distribution predictions will be implemented, alongside creating functions to model the line of best fit and continuing trends of the data. The formulae will be used in a similar manner to graphing, in which the relevant data sets to be used will be selected by the user. Due to the formulae operating off of raw data, no additional settings need to be implemented – just a data preview window so the user can affirm that they have selected the correct data. The application of the Normal Distribution for predictions and composing formulae for modelling trends is something that data scientists would greatly value. The ability to make predictions about following years and then later compare it to true data sets allows for viewing the accuracy of predictions, and altering formulae to make them more accurate. Being able to predict weather metrics is also important with greater concern over Climate Change – where monitoring extreme and unordinary weather values and the impact they're having is desperately needed.


Predictions, Formulae and Other Calculations
Yes
Normal Distribution
Yes
Using the whole set of relevant data to prediction
No
Etrapolate data to make predictions reliably
No
Derive formulae for "line of best fit" or to follow data trend

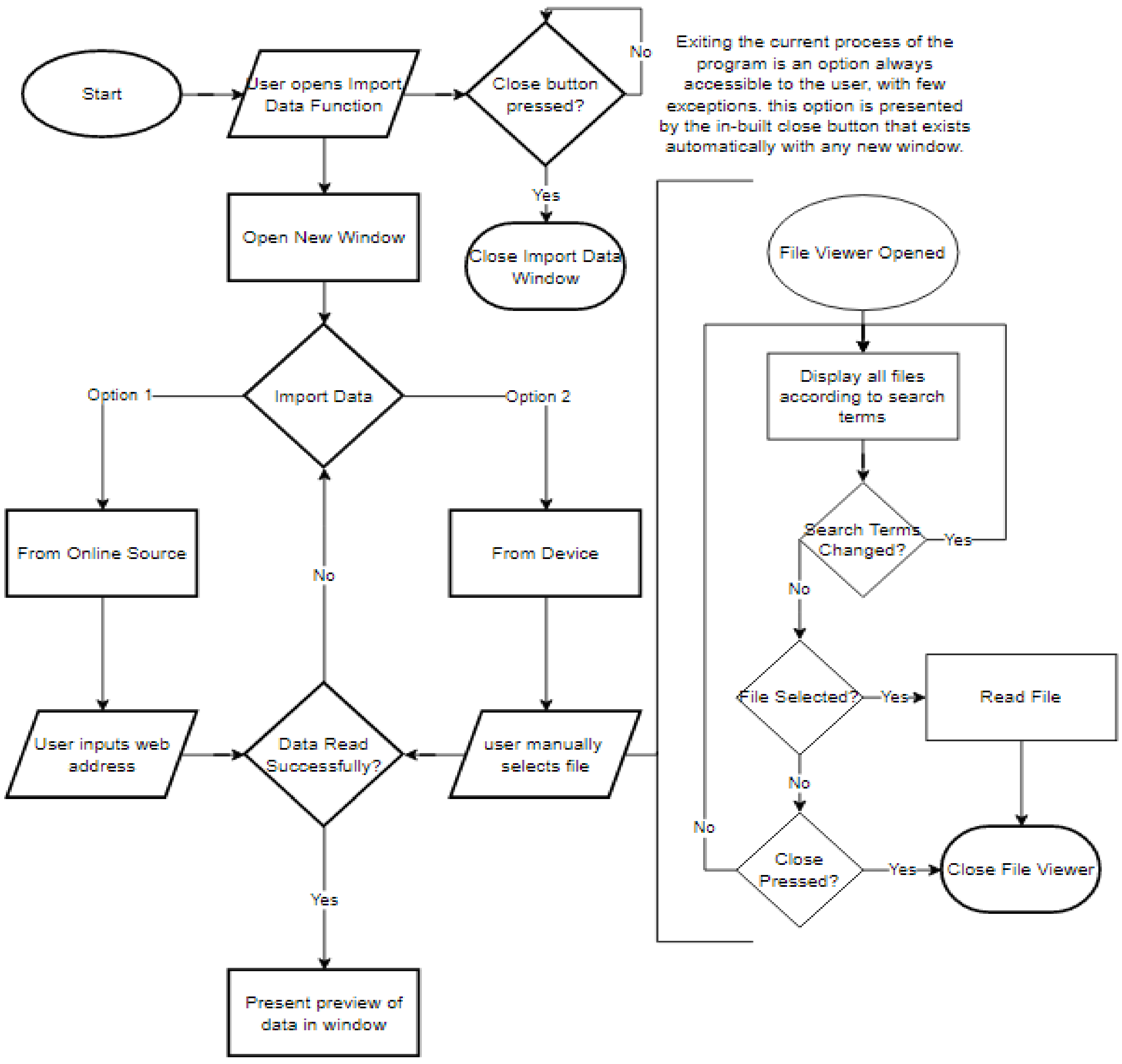
User Interface

Primary features will regard the formatting and layout of the program. It should be designed in an intuitive fashion that the user will likely be accustomed to through common user interfaces found in Windows or Mac programs. Through this, users should have little issue navigating the program. Furthermore, I aim to permit a large amount of user interference in the program and, importantly, avoid it feeling out of place. Alongside the essential user interface features, user-implemented / adjusted alterations to the appearance of the program, which will incorporate various accessibility features such as font, text colour, background colours. Further visual accessibility can be improved through a magnifier to increase the size of words and menus


User Interface
Primary Features
Cleanly presented
Menus are Well-Labeled and Obvious
Situational Menus, that appear when attempting to construct a graph of data. or similar occasions
Concise and not overfilled
Label
Previews of data are available when potentially needed
toggle-able. Can be hidden
Accessibility Features
Allow for user to modify colour of background and text

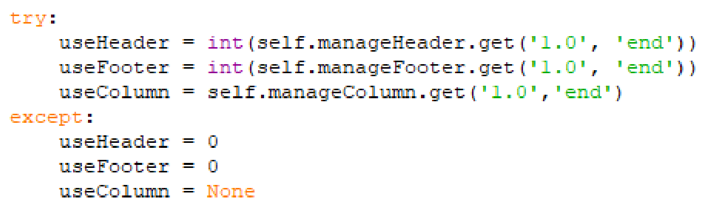
Importing Data

This flow chart shows the sequence of events the user will undergo when initially selecting data to import into the program. The detached flow chart on the right-hand side is a more in-depth look at the events occurring during the "User manually selects file" process. Following this, the user will then select from various options on managing the newly imported data. The user cannot import and manage more than one data set at once. Furthermore, the primary window will be prevented from receiving or utilising inputs from the user whilst the data importing window is open. This feature will be true whenever a secondary window is created to avoid errors arising from making inputs at the wrong time, or leaving a process early.


Start
User opens Import Data Function
Close button pressed?
No
Yes
Exiting the current process of the program is an option always accessible to the user, with few exceptions. this option is presented by the in-built close button that exists automatically with any new window.
Open New Window
Close Import Data Window
File Viewer Opened
Display all files according to search terms
Import Data
Option 1
Option 2
From Online Source
From Device
Search Terms Changed?
Yes
No
User inputs web address
Data Read Successfully?
user manually selects file
File Selected?
Yes
Read File
No
No
Close Pressed?
Yes
Close File Viewer
Yes
Present preview of data in window

Due to tkinter being a constantly running program, general errors which would crash and stop the running of a program can be avoided. As a result, the errors I have encountered whilst programming do not result in the requirement of a “Try: Except:” to prevent crashes. However, there are occasions where I have incorporated this should there be errors I have not encountered. Furthermore, it provides an opportunity to present a reason to the user as to why something has gone wrong, or just to notify them that an error has occurred in the first place – it may be rather small and unnoticeable, but have larger affects on their work whilst using the program.

With regards to entry validation, sensitive fields that deal with the reading of data will require more closely monitored inputs than other areas. Some examples that I will have to deal with are the header, footer and column sections of the read_excel function. If the user enters a non-integer value, or a value greater than the available data, then errors will be raised. The exception here is used to provide default supported values that will not cause errors, but will also be noticeable to the user that their inputs have not been applied – alongside a suitable error message informing them of such.



```
try:
    useHeader = int(self.manageHeader.get('1.0', 'end'))
    useFooter = int(self.manageFooter.get('1.0', 'end'))
    useColumn = self.manageColumn.get('1.0','end')
except:
    useHeader = 0
    useFooter = 0
    useColumn = None
```

In general, the user inputs can be limited in what can be entered and their frequency. A vast number of inputs will be done through program-constructed drop-down boxes, buttons, and slider values. These completely avoid the need for validation as it can only use suitable information. There may be some additional programming needed to convert a StringVar() value into a true string or convert into an integer, however, it is not much issue to incorporate. These methods of input still provide a large range of manipulation for the user, despite the seeming limitations of not allowing for any and all inputs. For example, the dropdown box listing all sheet names from an excel document is far more reliable than expecting the user to enter in the correct sheet name perfectly, each time they need a new sheet.

The User Interface's Design

The UI should be simple and understandable. Elements should have a label to identify them, and potentially even incorporate a help button in-program through which a sort of manual can be given to the user. In order to have these simple and understandable elements, I will not be using much variety in colour or font in order to retain simplicity – albeit I will allow for fonts to be changed in accessibility settings for those who have difficulty reading any text. The program will also be spacious and segmented. Each section of the screen or window will be distinctly defined with a particular group of objects – such as widgets affecting which screen you are importing, and a separate frame to preview this import as changes are applied.

Accessibility Features

One of the more easy-to-implement accessibility features is the previously mentioned font changing. Fonts such as Comic Sans were designed to be easier to read with more individually distinguished alphanumeric values – although mostly focused on letters. For my program, if I incorporate a method to reload the window and apply a font change to all widgets, then in a section of the settings the user can be able to switch between some various pre-set fonts. This would include the default Tk font, which all widgets will be initialised with, Comic Sans, Ariel and I will likely incorporate some others either for personal preference or accessibility and decide on those later.

I have also considered the idea of being able to change the background colours or font colours. Whilst by design the program is intended to be visually clear and free of clutter, some may have preferential colours or find the Grey and Black of the program difficult to differentiate. The only potential issue with incorporating the ability to change colours is that any approach will need to deal with the fact that the Tkinter widgets deal with background and foreground colours differently, and also to what extent the user can manipulate the colour. This feature is something I will likely consider later into the development of the program, as it may be unfeasible to introduce in a non-convoluted manner.

The Benefits to Prototyping

Producing prototypes for the development of my program grants multiple opportunities to better my program, observe its failings and successes, and gather feedback from stakeholders on their desired features or views on implemented features. Overall, the project will feature 3 main prototypes, from which their successes and overall development are accumulated into a single final program. This particular subsection has been appended to the first prototype to explain the layout of each Prototype analysis section:

In each Prototype section, the general aims of the prototype will be covered first. Here, I will grant an insight as to what the program is currently seeking to test and obtain results or information - from which I can improve the tested features as and when necessary. As I move between prototypes, there may be also areas which are tested multiple times due to improvements or alterations introduced from the previous iteration of the program. Their improvements and differences to the previous iteration outlined in their own individual subsections.

Testing of features and functions is the next segment, where I shall myself analyse the performance of the program's algorithms and features. Through this testing I can ensure that the program functions as expected and required of it – according to suitable and proper input. It will report on what was successful or lacking in the features, and how they can be improved. Combined with user-testing (covered in the following subsection), I can derive what is most crucial to work upon for the following prototype. Each feature (or group of features, such as input methods and validation) being tested will receive its own subsection.

The testing subsections will also feature User feedback. This will allow for an additional analysis of the performance of features in the field of user experience. This primarily is intended to test the user experience and collect information on errors that my own testing missed. Furthermore, it grants an opportunity to obtain feedback from stakeholders and test users on the program's features - this can incorporate what they believe to be omitted or unnecessary features, or elements potentially requiring improvement.

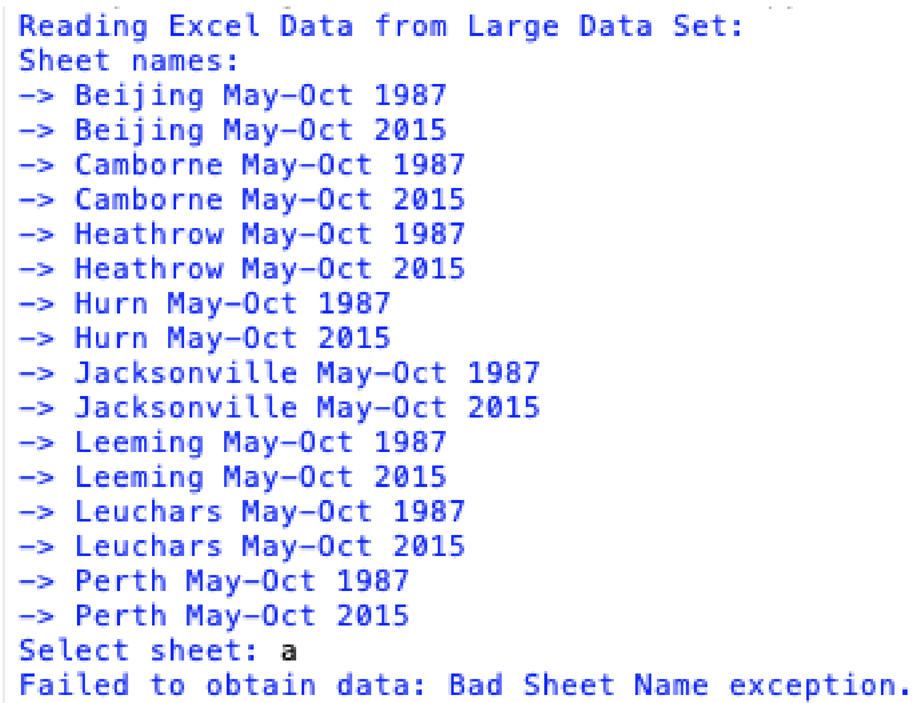
Some subsections of notable areas of development during the program have also been added. These areas cover the progress of developing new features or methods. Improvements to previous prototype functions are elaborated upon in the testing sections, prior to the testing overview and results.

The final subsection to each Prototype will be a general overview or conclusion of its results in testing. It will outline the most important aspects mentioned in the prior subsections, and the actions to be taken for the next prototype.

Aims of The First Prototype

Prototype 1 was intended to be limited in its ambitions and would serve as an introductory setting for the stakeholders to how the program was intended to function in a heavily simplified setting. It would return information for how I should develop and utilise the matplotlib library and obtain any particular requests with regards to the graphing functionality.

The initial prototype would involve a text-based program which the users would interact with through a limited scope to avoid errors deriving from accidental or intentional human input mistakes. It was set to be using data only from the Large Data Set at this time. Fortunately, the Large Data Set provides an extensive list of datasets to use for testing purposes.



```
Reading Excel Data from Large Data Set:
Sheet names:
-> Beijing May-Oct 1987
-> Beijing May-Oct 2015
-> Camborne May-Oct 1987
-> Camborne May-Oct 2015
-> Heathrow May-Oct 1987
-> Heathrow May-Oct 2015
-> Hurn May-Oct 1987
-> Hurn May-Oct 2015
-> Jacksonville May-Oct 1987
-> Jacksonville May-Oct 2015
-> Leeming May-Oct 1987
-> Leeming May-Oct 2015
-> Leuchars May-Oct 1987
-> Leuchars May-Oct 2015
-> Perth May-Oct 1987
-> Perth May-Oct 2015
Select sheet: a
Failed to obtain data: Bad Sheet Name exception.
```

Beyond the scope of the prototype was the importing of data and the prediction of data trends. Importing data will be explored during the graphical user-interface prototypes as it would work most effectively when implemented with a GUI – a file searcher would be rather difficult to use in a text-based interface. Due to the importing of data being unexplored, this largely also left the Pandas module underutilised in the first prototype. Whilst I would implement it for the purposes of parsing data from the Excel Document which the Large Data Set is stored in, it would not fully display its capabilities in producing its own documents, and additional specifications which can be attached.

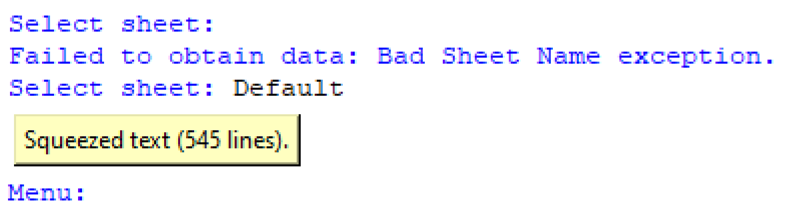
The prediction of data trends is also a complex feature, and I have decided to leave it absent from the initial prototype to save on complexity and time for a basic text-based program.

Inputs and Validation - Testing and Evaluation

Function	Working	Score / 10	Errors / Issues	User Feedback
Crash prevention	Y	10	None	
Re-entry / Re-attempt input	Y	7	None	Avoid program automatically using default values in case of user mis-input. Ensure multiple attempts in ALL selection events
“Default” value	Y	7	None	Default method was useful for going quickly through program Build upon the Default values feature so that it is more explicitly mentioned Increase Default feature’s utility
Menu	Y	10	None	

The Menu subroutine would not provide any direct issues with the program. it was simply responsible for fetching a user input from the user and comparing that to the values of the acceptable options provided to the subroutine. As a result, the program would be largely safe from any user-input related crashes or errors due to the input validation performed by this subroutine.

The program would also function without issues because of the care taken with the application of try and except clauses or utilising lists to detect if a user input is within x list. Preventing errors and crashes being raised upon unsuitable inputs (such as enter, or numerical values) was necessary for a smoothly running program.


```
Select sheet:
Failed to obtain data: Bad Sheet Name exception.
Select sheet: Default
Squeezed text (545 lines).
Menu:
```

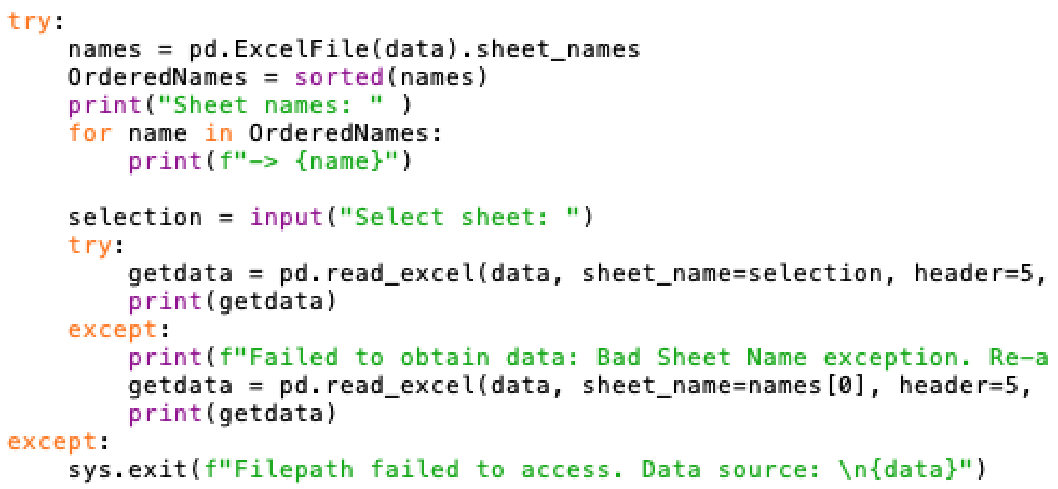
The addition of the “Default” option to selections would also prove useful for quickly using the first sheet in a data set. This feature was not explicity mentioned in the program due to the intital implementation being due to the necessity for a quick function to default select values so that I could move onto the testing of features further in the program. However, test users would find the idea to be a particularly useful one when notified of the feature. As a result of this, being able to use Default values quickly and rapidly will be implemented in future prototypes for the purpose of demonstrations, and rapid testing. Test Users believe that a default options button would remove the unecessary hassle of re-entering values for graphing and data management each time a file is imported.

Whilst I had anticipated that user-inputs would be preferred to be re-attempted in areas such as sheet selection when reading from the Large Data Set document, I did not apply the while method in column selection for the user to produce a graph. This lead to Test users accidentally being forced into using default column value “Date”, which may then have been selected for both axis – providing a useless graph, and effectively requiring a restart. As a result of this incident, I would not omit the user's ability to re-attempt entry of values from any user-input area for future prototypes.

Overall, this section of Input Validation was largely successful, with no errors, and only a few minor details to improve in the next iterations.

Inputs and Validation – Development

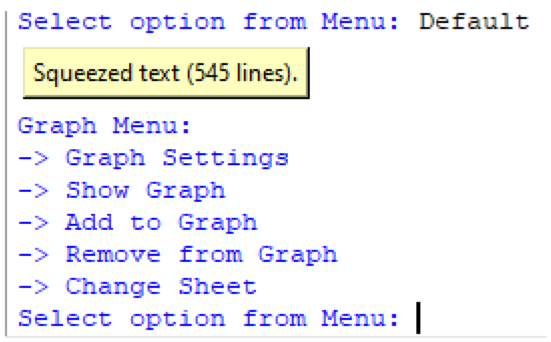
The program in this prototype would be designed to import the Large Data Set with default headers and footer values to avoid complexity with importing specific data sets. However, I could permit for the user to enter a particular sheet name for them to observe the part of the utilisation and necessity of Pandas in this program – other than parsing from Excel or CSV documents. The inputs would have to be treated carefully, however, as they could easily result in the program crashing or breaking with incorrect characters or misspellings. This led to the arguable overuse of the "try:" and "except:" clauses in preventing crashes from the user. However, without these the test users would be required to reload the program each time a crash was encountered. This was an appreciated feature in the test-users experience, as with large amounts of data and extensive fiddling with various settings and values would quickly become wasted upon a single crash of the program.



```
try:
    names = pd.ExcelFile(data).sheet_names
    OrderedNames = sorted(names)
    print("Sheet names: " )
    for name in OrderedNames:
        print(f"-> {name}")

    selection = input("Select sheet: ")
    try:
        getdata = pd.read_excel(data, sheet_name=selection, header=5,
        print(getdata)
    except:
        print(f"Failed to obtain data: Bad Sheet Name exception. Re-a
        getdata = pd.read_excel(data, sheet_name=names[0], header=5,
        print(getdata)
except:
    sys.exit(f"Filepath failed to access. Data source: \n{data}")
```

Shown above is an example for the selection of the user input and how the try method was implemented in the program. Either the try could be used to employ default values upon the exception case being necessary, or it could be utilised in a While loop – designed to break when the code executes successfully and without error. I expected that the end-users would prefer to be able to reattempt inputs should a mistake be made, and stakeholders would provide feedback matching with this. Exceptions which resulted in a forced selection of the default program-provided values still effectively forced a restart in situations where the test-users intended to use a non-default input or value. Future iterations will not use as many try clauses, as they ended up unnecessary. Instead, the user inputs will be compared to a list of acceptable inputs – such as the sheet names imported from a document.

Whilst menus would be difficult to implement for a text-based program, it was somewhat doable through allowing the user to interact with the various options in a text-based menu system. Here, the user will cycle between 4 different subroutines which repetitively call each other to simulate a menu. I presented the test users with the options to add, remove and produce a graph using data downloaded from their selected Sheet of the Large Data Set. Alongside this main menu, additional options would greet the user once they access the "Add to Graph" option. In this, the design elements of the graph can be altered through



```
Select option from Menu: Default
Squeezed text (545 lines).
Graph Menu:
-> Graph Settings
-> Show Graph
-> Add to Graph
-> Remove from Graph
-> Change Sheet
Select option from Menu: |
```

line colour and point markings. Furthermore, they can access different options for the graph – such as bar, scatter and line.

Since the graphing feature of the program would require numerous menus to navigate through all the options, I designed and implemented a subroutine for a “Menu” function. When provided with a list of options and a title card for the menu to display, it will output a Menu list whenever the function is called, and return the selected value. Furthermore, it also checks within the function if the user’s input is a part of the options, or “Default”, before returning the value. The Menu subroutine is shown below:

```python
def menu(menutitle, options):
    print(menutitle)
    for opt in options:
        print(f"-> {opt}")
    userinput = input('Select option from Menu: ')
    while True:
        if userinput in options:
            return userinput
        elif userinput == 'Default':
                return options[0]
        else:
            userinput = input('userinput not in options. retry input: ')
```

Upon receiving a value returned from the subroutine, there is little need for over-precautionary measures such as try and except to prevent crashes, as the function has effectively cleaned the user inputs already. Furthermore, due to the user only being able to select a value that is provided, issues with the read_excel functions or selecting a particular column and entering a non-existent name are avoided – provided that the options given to the Menu subroutine to display and for the user to select from do not cause issues. Which they should not, being derived directly from the Large Data Set excel sheet.

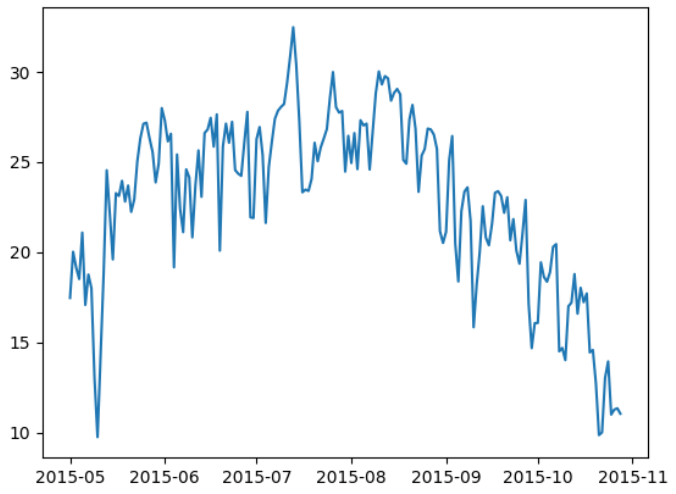
Graphing - Testing and Evalutation

Function	Working	Score	Errors / Issues	User Feedback
Add to Graph	Y	10		Add non-data set elements to graph?
Remove From Graph	Y	10		
Graph Settings	Y	8		Only design elements are present. Could there be more features applied?
Show Graph	Y	10		Graph be shown whilst being worked on simultaneously?
Close Graph	Y -	6	Sometimes closes the whole program – no errors raised. Unable to get it consistently occurring	

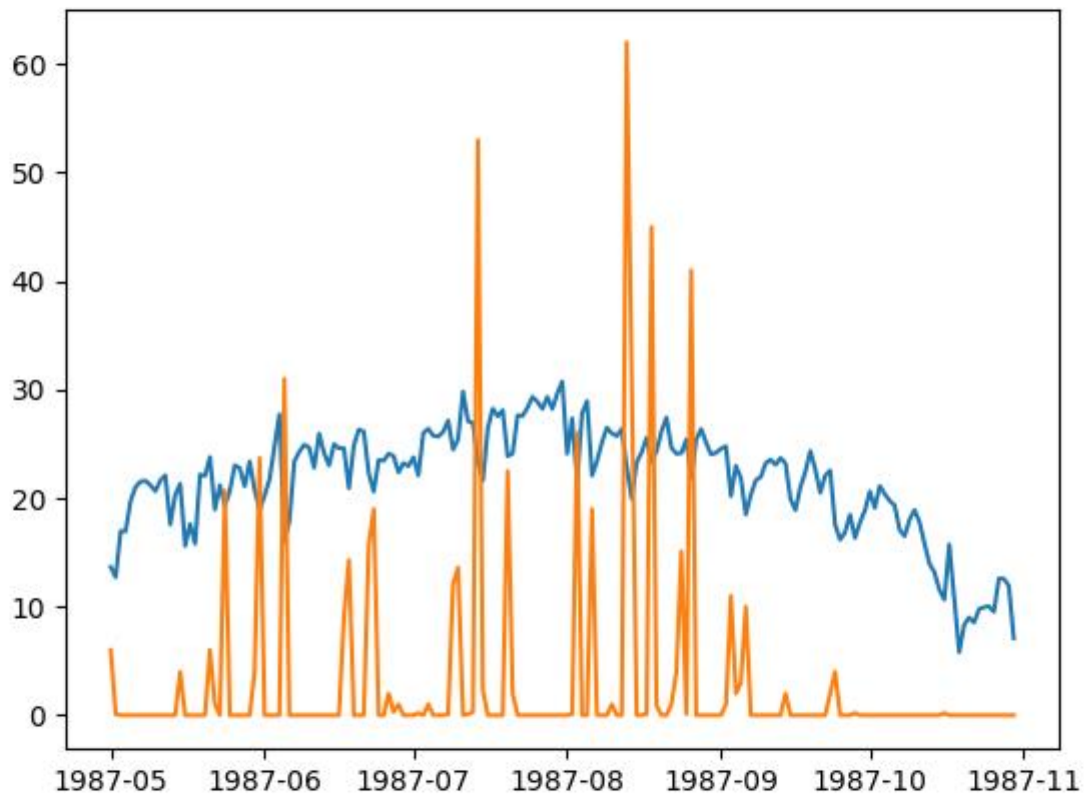
```python
import matplotlib.pyplot as plt
xAxis = ['values']
yAxis = ['values']
plt.plot(xAxis,yAxis)
plt.show()
```

Matplotlib is simple to utilise for producing a graph which can contain multiple sets of data. At its most basic, all that is required is a set of x-Axis values, and a set of y-Axis values. Matplotlib will derive the points to mark on the graph by assuming that each x-value corresponds to a y-value at the same index. And for the axis length, unless otherwise specified, they will be set equal to the maximum and minimum values in the axis lists.

This is demonstrated in the next image, whereby I have assigned the assessed Dates from the LargeDataSet into the x-axis, and Temperature into the y-axis for Beijing's 2015 data. Furthermore, the plot() function can accept multiple x and y axis value sets, allowing for a user to efficiently produce a comparison between two sets of data – such as the rainfall and temperature. Additionally, there are also the capabilities for design elements – such as the colour of the lines, bars or points in graphs, adding labels – which I could implement through automatic default names and allow for the user to make their own – and titles for the graph.



What would not function so effectively wa the comparison of data between differing times. Due to data being added to the graph being dependant upon having a dataset column for both the x and y axis, you could not truly compare data between different sets. For the next prototype I aim to allow for a "None" option for the x-axis which will simply result in the y-axis data being mapped to x incrementing by 1 each time.



Overview of Prototype 1

Prototype 1 would prove to be a successful method of introducing the graphing capabilities to the stakeholders and displaying the potential it held for the purposes of data analysis. For the future prototypes, however, there was useful feedback from which I aim to improve the current features. The ability to utilise default values will be incorporated – likely with values being entered as a default and hence skipped over, yet can still be altered by user input. Furthermore, all methods of input will continue to avoid crashes due to user error or misinput, alongside permitting the re-entry of data values. However, I believe that utilising a GUI for the next iterations would automatically solve this issue without needing to take any direct action.

With regards to the graphing of data, I believe the suggestion of allowing for non-data graphs to be a good idea. Linear graphs can be used to mimic data trends easily, and allowing for quick and easy modifications to them should allow for users to have a lot more freedom in their graphing. I may also be able to implement a function to produce a line of best fit

Aims of the Second Prototype

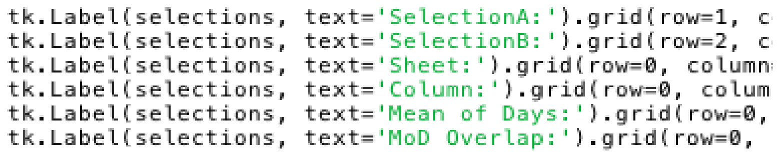
The first prototype provided information on the graphing function. In this prototype, I aim to introduce the improvements mentioned in the Overview of Prototype 1 and also introduce a GUI that the user can interact with. Furthermore, I aim to introduce greater interaction for the user when importing files through the use of a file finder and permitting them to alter settings to alter how data is imported.

The predominant purpose of this prototype is to construct the GUI environment for the user and develop a user-friendly and intuitive space. Feedback from test-users will be looking for how the GUI environment can be improved or modified. The development of the GUI has been highly documented in this section, as I aimed at securing a largely stable GUI that does not change much over the next prototypes. The development of the GUI features as implemented in Prototype 2 were heavily influenced by user feedback during development, as it was quick and efficient to produce multiple small exemplar designs.

Widget - Development

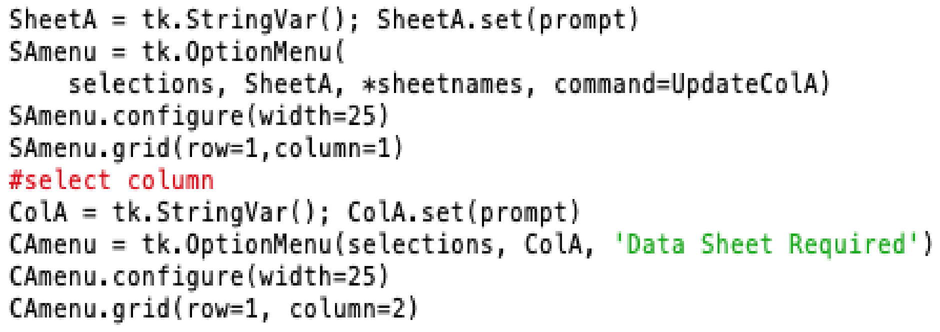
Producing the required Tkinter widgets will require the declaration of the widget and then positioning it using grid() - which provides greater positioning management than the pack() method, and I can produce a more efficient layout. A lot of lines will be repetitive code simply declaring these widgets, with their functions being declared in subroutines as per the functionality of the 'command' component in buttons and similar widgets.

Tkinter does not require variables for every widget, and can in-fact correctly produce and display widgets without it. However, I will need a method to apply changes to these widgets and configure them at a later event. For example: the user changing the font. The easiest method through which this can be done is providing a variable in which the widget is stored. From there, the .configure() method can be applied to the variable to allow for post-initialisation changes to be applied to the widget. As a result, all of the widgets used in the program – even with minimal function – will be utilising their own individual widget.



```
tk.Label(selections, text='SelectionA:').grid(row=1, c
tk.Label(selections, text='SelectionB:').grid(row=2, c
tk.Label(selections, text='Sheet:').grid(row=0, column
tk.Label(selections, text='Column:').grid(row=0, colum
tk.Label(selections, text='Mean of Days:').grid(row=0,
tk.Label(selections, text='MoD Overlap:').grid(row=0,
```

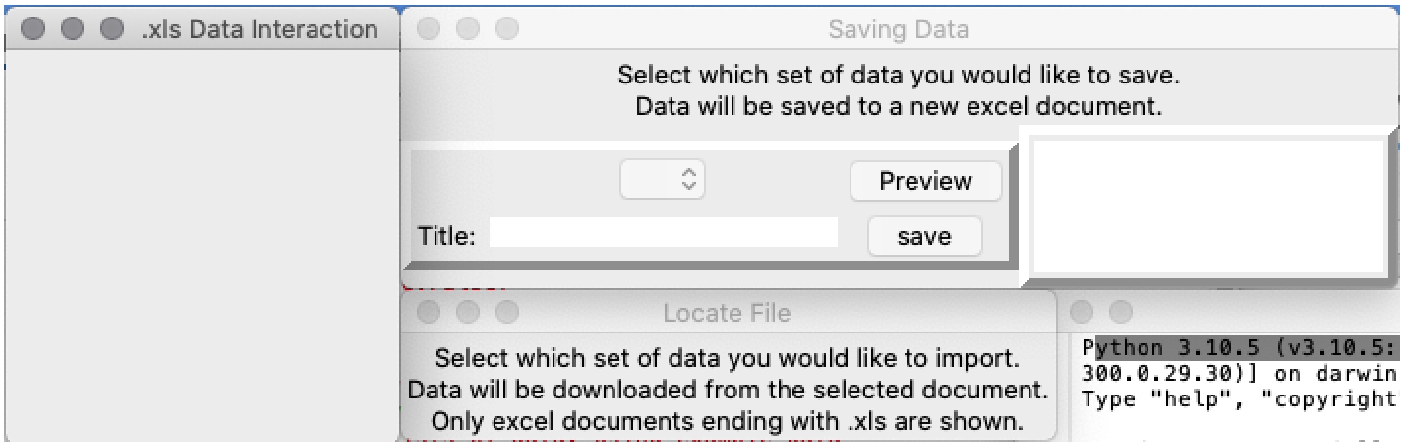
Due to the way in which the widgets are being managed through constant re-configurations and unloading, I would utilise variable names for each of the widgets. There was a potential complication whereby unsuitable variable names for the widgets would result in confusion when errors arose or with identifying their purposes. To deal with such, for my prototype I determined a naming scheme to avoid such issues. With regards to naming the widget variables, they must be coherent and relate to the function they serve as a basic minimum. Additionally the names will also incorporate their position and role in the program – such as the window it is in, or its parent frame, and its action. This would help differentiate between similar functionality widgets with otherwise confusingly similar names. These variables can also be appended into a list, which can be linearly processed to grid each widget stored inside, allowing for the loading functions to iterate linearly through a list of widgets.



```
SheetA = tk.StringVar(); SheetA.set(prompt)
SAmenu = tk.OptionMenu(
    selections, SheetA, *sheetnames, command=UpdateColA)
SAmenu.configure(width=25)
SAmenu.grid(row=1,column=1)
#select column
ColA = tk.StringVar(); ColA.set(prompt)
CAmenu = tk.OptionMenu(selections, ColA, 'Data Sheet Required')
CAmenu.configure(width=25)
CAmenu.grid(row=1, column=2)
```

Window - Development

Initially I planned to utilise multiple windows to keep each part of the program separate and concise. However, the result was in fact the opposite. Multiple windows results in a large amount of clutter quickly taking up space on the screen. Furthermore, I would have to implement more protections against user input in windows when other processes are ongoing. This may result in significant issues where the user finds a method to enter and alter values in windows that should not be in current use. Shown here, both the Saving Data and Import Data windows are being opened simultaneously. With an additional file viewer being opened to search for files, the user quickly gets bogged down sifting between windows which all look near identical at first glance.

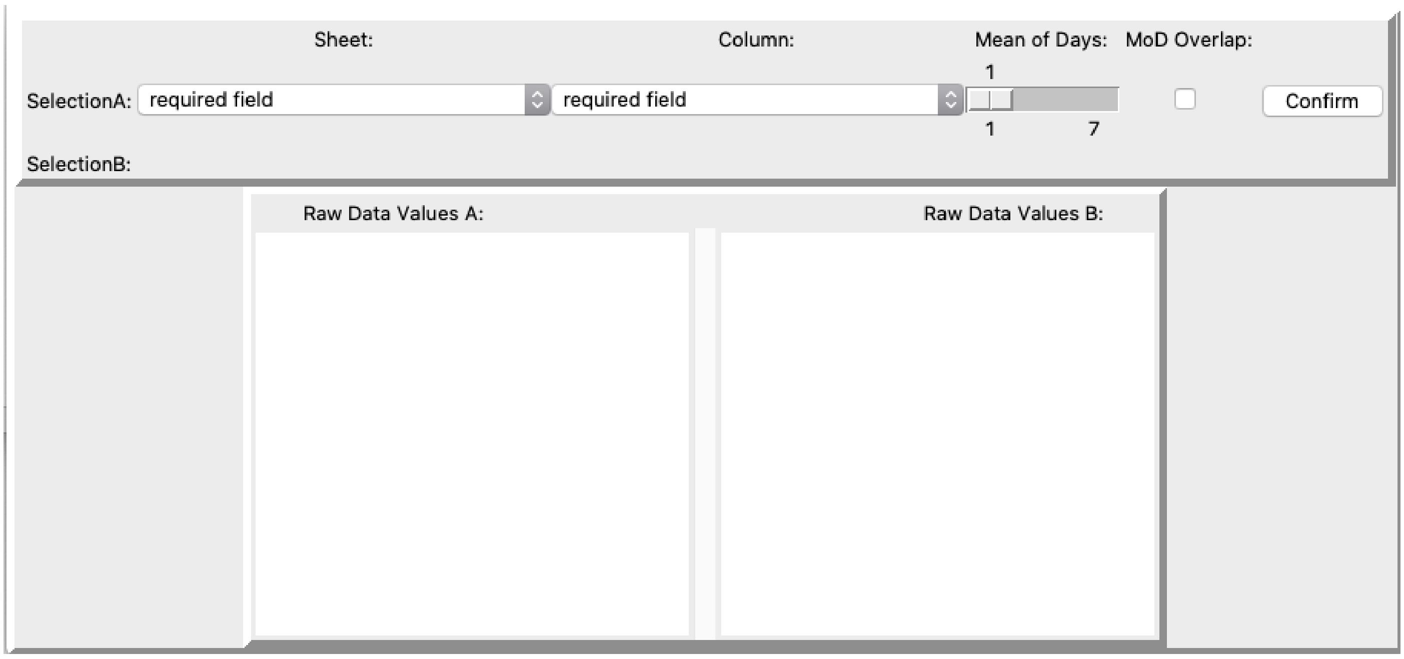


Select which set of data you would like to save.
Data will be saved to a new excel document.

Select which set of data you would like to import.
Data will be downloaded from the selected document.
Only excel documents ending with .xls are shown.

```
Python 3.10.5 (v3.10.5:
300.0.29.30)] on darwin
Type "help", "copyright
```

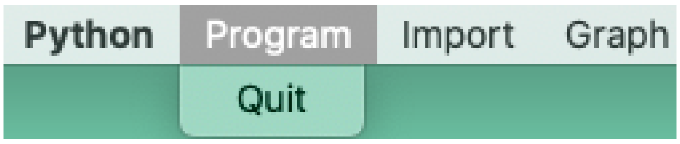
An alternative method for sectioning the program is rather than using individual windows, I put into use the forget_grid() method of Tkinter. Using this, widgets will be removed from their position, hidden and cannot be interacted with. This will also allow for other widgets to take previously occupied positions. The requirement of multiple windows to house all the various widgets of the program is removed, instead multiple interchangeable frames can be used to simulate different screens in the program. This led to the significantly less visibly cluttered prototypes. There were no particular downsides to this method of using only 1 window, as effectively multiple windows simply shared one output location. All that is required is a method to switch between the currently presenting screen and another of the user's choosing. Below is an image for the incomplete graphing screen. Other screens can follow in its example with frames being used to section off areas, such as the area where raw data is shown and the selection of data columns.



This resulted in a question of switching between screens. For the purposes of this, I considered buttons, however a less space-consuming and more intuitive method can be utilised. Hence, I utilised the menu bar widgets can be used to produce a menu bar for the program – much like Word and other applications have at the top of the screen. Furthermore, when selecting a menu item, a command can be attached. This avoids the use of large and rather cumbersome buttons for changing screens, and instead makes for an intuitive format which can be kept out of the way of the rest of the important features in the program.

Shown here with the menu bar on Mac, it appears at the very top of the screen. In windows, it appears in the program, however, it functions the same on both devices. There are no difficult parity issues to address through this method, making it particularly suitable. The menu bar can also allow for cascading menus – this will allow for the screen selection to be tucked away and not take up a large amount of space. Furthermore, the vast majority of people have encountered cascading menus previously – especially those I aim to utilise the program.

```
programMenu = tk.Menu(menubar)
programMenu.add_command(label='Quit', command=exit)
importMenu = tk.Menu(menubar)
importMenu.add_command(label='unused')
graphMenu = tk.Menu(menubar)
graphMenu.add_command(label='unused')


menubar.add_cascade(label='Program', menu=programMenu)
menubar.add_cascade(label='Import', menu=importMenu)
menubar.add_cascade(label='Graph', menu=graphMenu)
```



What was left to consider was how the currently placed widgets could be removed from the screen and temporarily replaced with an alternate screen's widgets. Previously I mentioned the forget_grid() method, which comes into use in the loading and unloading functions I will cover in the next subsection.

Loading and Un-Loading - Development

I required a callable subroutine that could provide the grid placement to widgets provided to it. A second separately callable function would perform the un-loading of the provided widgets. This essentially becomes the two basic functions of:

```
def unload(self, WidgetsList):
    for widget in WidgetsList:
            widget.grid_forget()

def load(self, WidgetsList, gridValsList):
    i = 0
    for widget in WidgetsList:
            widget.grid(row=gridValsList[i],
                        column=gridValsList[i+1])
            i += 2
```

All that was required for the functions to operate is a WidgetsList which houses all of the widgets which grid is being applied to, and gridValsList to store the row and column values for which the widget is to be placed at. With the current designs of how the widgets are created – with each having its own independent variable – I can easily append all the variables to a list, which is then passed into the necessary function. The same would apply to the grid values.

Further development of the load function would be necessary as the program developed. One aim was for the ability for users to change font type and colour, and so this will be introduced into the load subroutine. Upon the user changing font, a load can be called and the current screen will change font. Furthermore, whenever a different screen is called to load, the font will apply to those widgets too. However, not every widget can have the font type

```
try: widget.config(font=defaultFont,fg=fontColour)
except: pass
```

applied to it. To avoid crashes in the case of the load function attempting to apply a font to Frame or other unsuitable widgets, I incorporated a simple try method to simply skip the font process if it is not possible.

The defaultFont and fontColour variables here refer to the users selection of font, and font colour. Furthermore, the font is default set to being the 'TkDefaultFont' and colour black at the start of the program.

```
defaultFont = tkFont.Font(family='TkDefaultFont', size=8)
fontColour = 'black'
```

However, there would be some necessary widget-specific functions to implement into the loading subroutine. Scrollbars would require a value in the "sticky" value of the grid method, which could be added to the gridValsList list. Fortunately, this was also safe to apply to any widget as it could be set to '' – meaning no value. The most complicated item to work around would be the Option Menu widget. I needed a way in which the list of values could be altered and changed, depending on user input – for example, an Option Menu displaying the sheet names of an excel document, which updates when a new document is being imported and managed.

Fortunately, the "Menu" of the Option Menu can be directly accessed, and each item deleted individually and new ones appended. With the addition of an if statement to detect option menus, a solution would be added into the loading function. However, it would also require a small alteration as to how the gridValsList would work.
Whilst it had previously stored only the row, column and sticky values, it would now be utilised to store additional

```
if isinstance(widget, tk.OptionMenu):
    widget['menu'].delete(0,'end')
    for item in gridValsList[5]:
        widget['menu'].add_command(label=item,
                command=tk._setit(gridValsList[4], item))
    gridValsList[4].set(gridValsList[5][0])
```

information about the widget. In this case, the gridValsList[4] and gridValsList[5] positions are variables used in the creation of Optionmenus. Position 4 is a StringVar() variable used for identifying the currently selected value of the Optionmenu. Position 5 is a list containing the items that should be in the optionmenu. Below is shown the finished version of the load subroutine as used in prototype 2.

```
def load(self, WidgetsList, gridValsList):
    i = 0
    for widget in WidgetsList:
        try: widget.config(font=defaultFont,fg='black')
        except: pass
        widget.grid(row=gridValsList[i],
                    column=gridValsList[i+1])
        if isinstance(widget, tk.OptionMenu):
            widget['menu'].delete(0,'end')
            for item in gridValsList[5]:
                widget['menu'].add_command(
                    label=item,command=tk._setit(gridValsList[4], item))
            gridValsList[4].set(gridValsList[5])
        i += 2
```

The loading functions are suitable enough such that they should be able to provide for following prototypes. Furthermore, they are quickly re-configurable to account for additional exception cases like how Option Menu is currently treated.

Loading and Unloading – Testing and Evaluation

Function	Working	Score	Errors / Issues	User Feedback
Load Screen	Y	10		
Un-load Screen	Y	10		Doesn't remove previously entered values when screen switched – Good thing!
Load specific Widgets	Y	10		
Switch screens	Y	10		

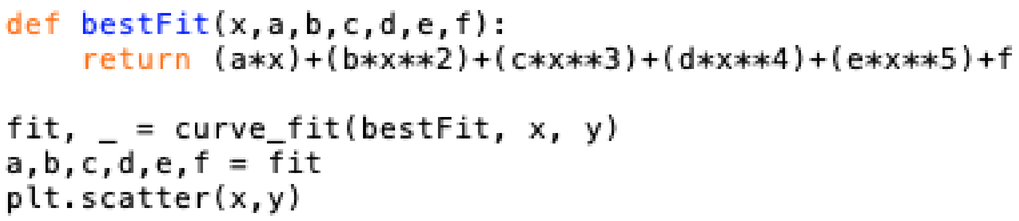
The loading section was crucial to have working perfectly, else the program as a whole would not function. for the purposes of Prototype 2, the function is working perfectly and as intended. Furthermore, it had an additional capability that I did not originally intend to permit whereby the passing of a single widget in a list – ie: load([WidgetName]) will allow for a single (or multiple) specific widgets to be loaded, rather than refreshing the whole screen.

Due to the success of this function in personal testing and user testing, there are no major or even minor flaws to iron out for the following prototypes, other than ensuring that the Widget lists and grid values being passed into the function are correctly organised.

Graphing - Development

Prototype 1 permitted Axis titles, a general figure title and altering of line, point or bar colours and styles. This would be continued in prototype 2, but the Graphing functions would have additions to significantly improve the feature in comparison to the original – which was largely intended to be an introduction to the program for the stakeholders.

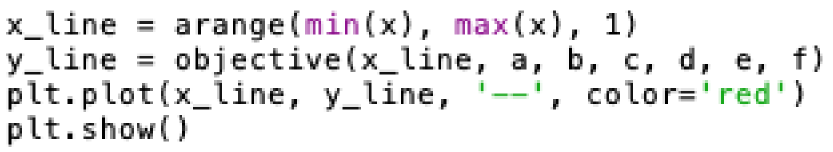
This graphing function would permit for the creation of lines of best fit – to match data plotted into line or scatter graphs. Furthermore, the user can graph – to a very limited functionality – their own polynomials into the graph plotter. Currently with prototype 2 I do not yet have a solution to parsing the majority of mathematical functions such as trig or exponents. As a result, I opted to push forward the prototype with limited functionality and to note to the stakeholders that it is in progress.



```
def bestFit(x,a,b,c,d,e,f):
    return (a*x)+(b*x**2)+(c*x**3)+(d*x**4)+(e*x**5)+f

fit, _ = curve_fit(bestFit, x, y)
a,b,c,d,e,f = fit
plt.scatter(x,y)
```

The line of best fit is a solution I introduced for finding the best fitting line to model a given set of x and y values. The Scipy module is utilised to provide the curve_fit function, which returns the best coefficients of ascending powers of x to fit the data x, y. it returns coefficients based upon the function being passed into curve_fit. In this case, my program will be dealing with up to x^5 values to provide a strong estimate of the best fit without requiring extensive calculations. Curve_fit also returns a covariance matrix, which is not necessary and hence stored in a _ variable to be ignored.

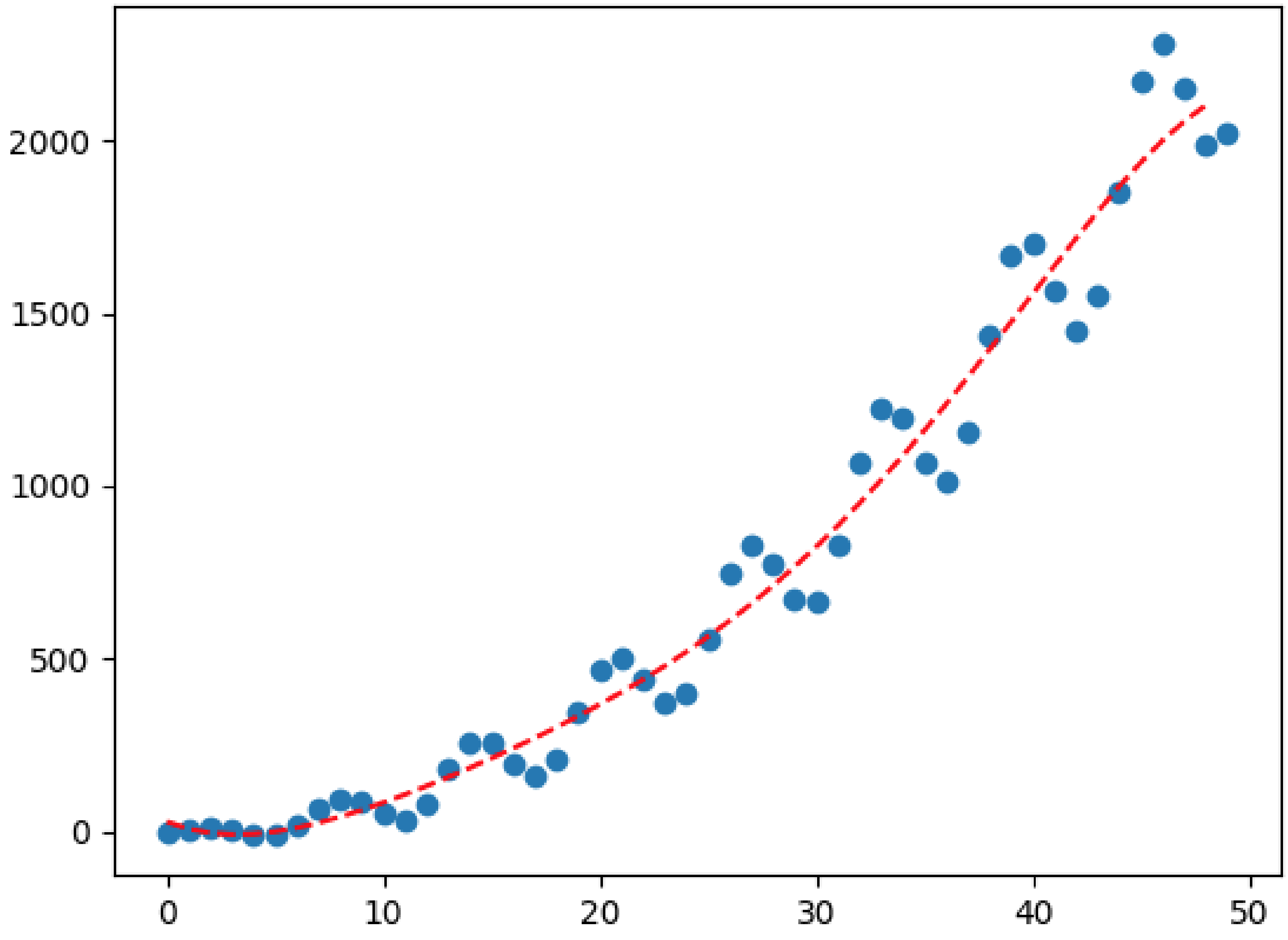
Following the obtaining of the coefficient values, the line of best fit is then plotted utilising an ordered list of x values, which are then applied into the bestFit function to return a set of yvalues corresponding to respective x positions.



```
x_line = arange(min(x), max(x), 1)
y_line = objective(x_line, a, b, c, d, e, f)
plt.plot(x_line, y_line, '--', color='red')
plt.show()
```

Hence, the result is a red dotted line that is mapped to a set of values. In the example case below, I created a complex and interesting looking function, as the feature will be tested with weather data during later tests.

```
for i in range(50):
    x.append(i)
    yVal = int(-2*x[i] + x[i]**2 + 6*x[i]*np.sin(x[i]) + random.randint(0,5))
    y.append(yVal)
```



However, it intends to show my goals in prototype 3 with the user-inputted functions, rather than only having polynomials entered. Here I utilise the trig function sin to model a small wave in the graph.

Also not included is the ability to work on the graph and have changes displayed without having to click the "show graph" button each time. Furthermore, the graph itself remains as an external window to the program.

Graphing – Testing and Evaluation

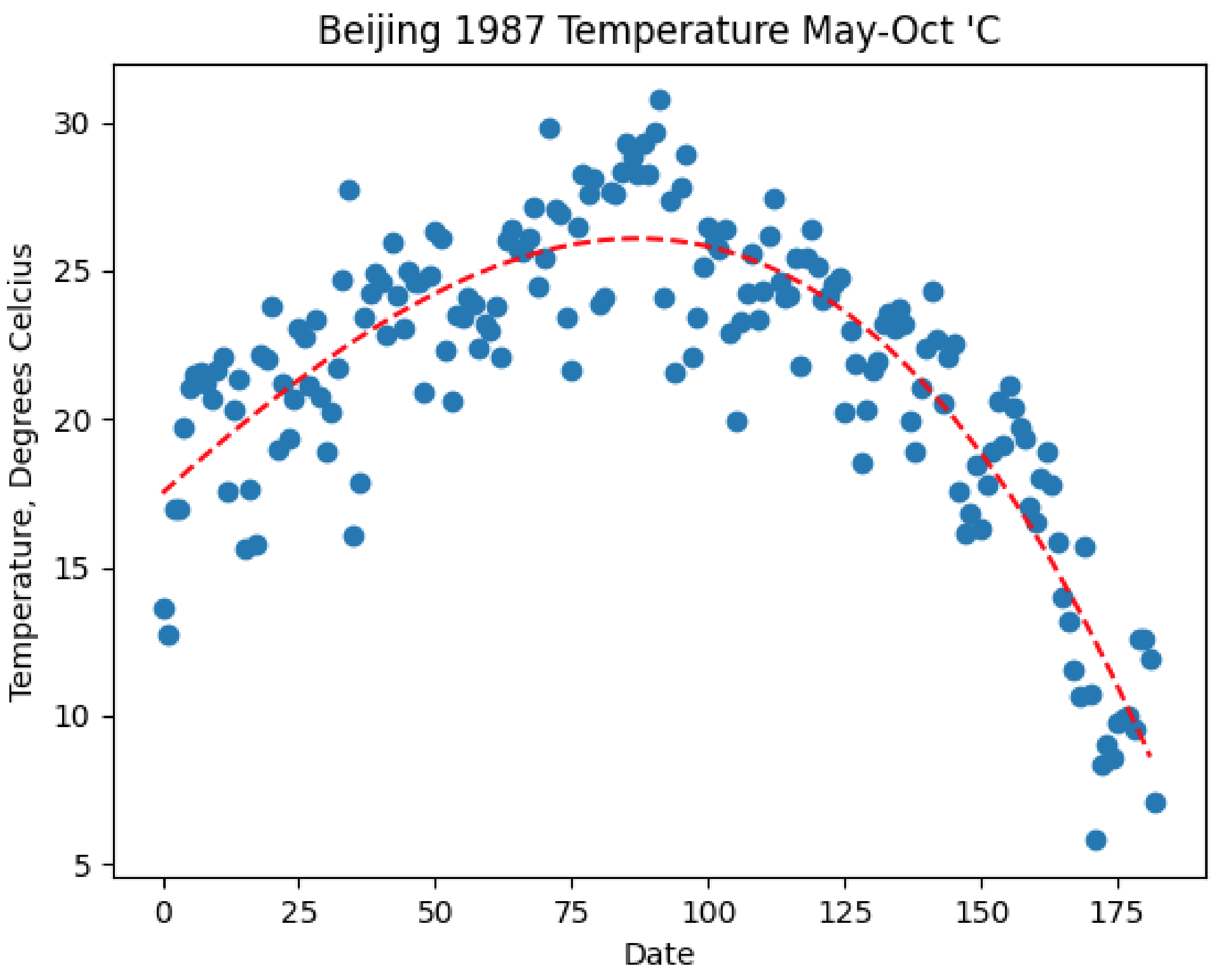
Function	Working	Score / 10	Errors / Issues	User Feedback
Add to Graph	Y	10		
Remove from Graph	Y	10		
Show Graph	Y	8		Have graph shown whilst being worked on to display real-time updates and

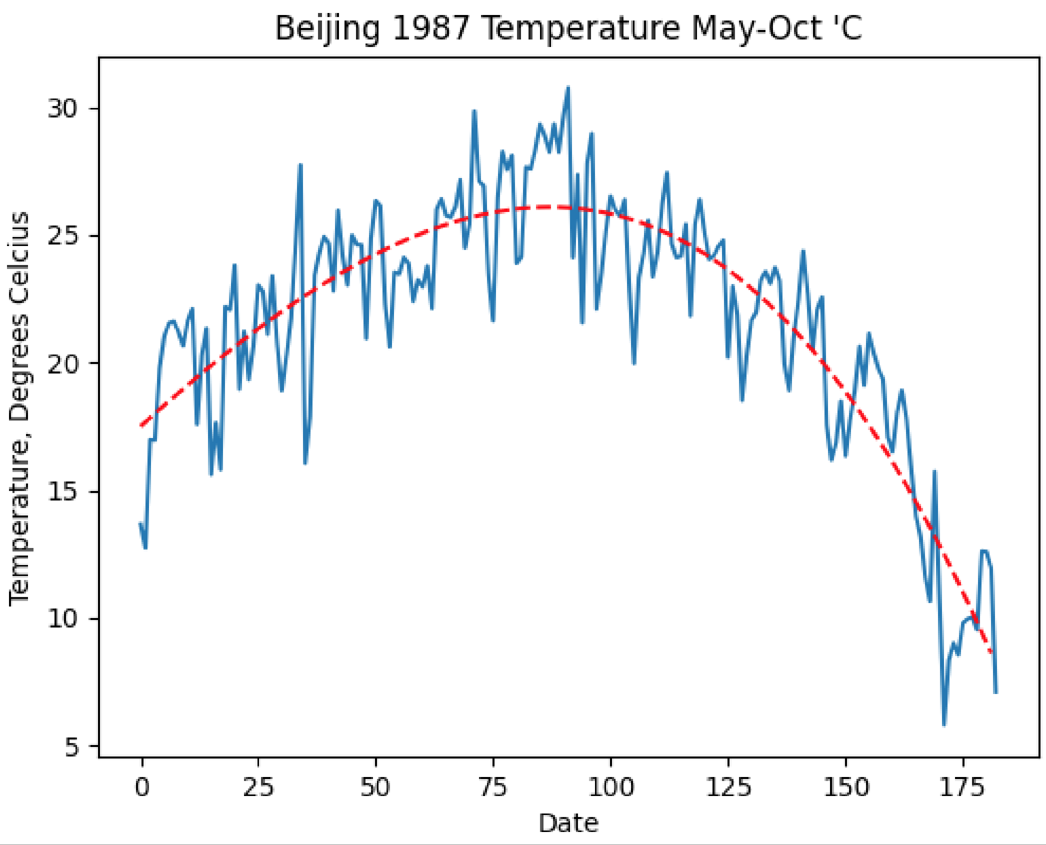
				changes
Graph Settings	Y	10		
Line of Best Fit	Y	7	Cannot calculate when using Date x-values	Currently extends to x^5 term, allow for max term to be based on user input? I.e.: x^n.
User polynomials	Y	3		Limited in use, often causing issues when used with the rest of the dataset graphs
Close Graph	Y	10		

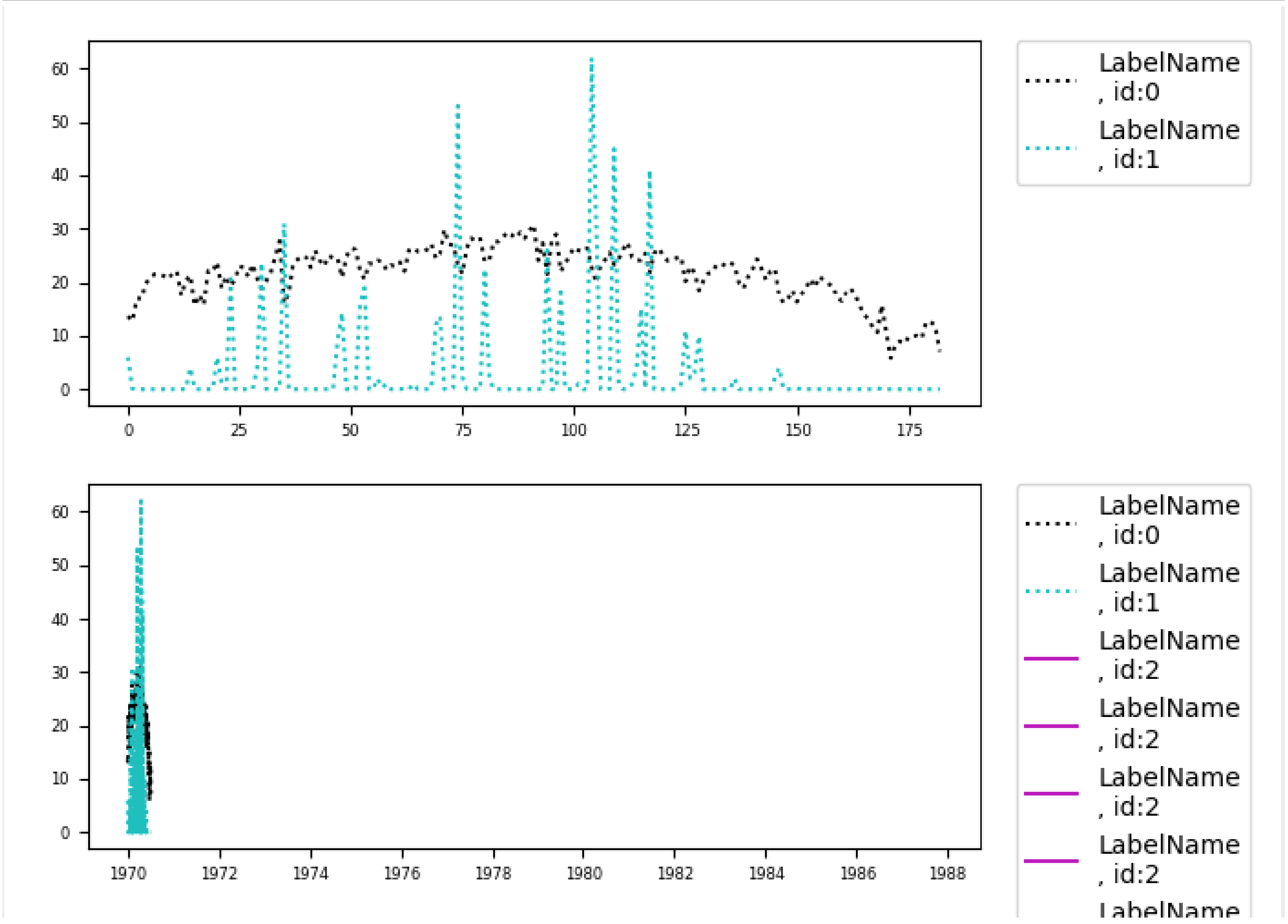
The ability for changes on the graph to update in real-time is a carry over from the first prototype. Due to its demand after 2 iterations, I shall ensure its addition into the next program. furthermore, the line of best fit will be granted additional strength and accuracy through users being able to define it beyond x^5.

During this round of user testing and my own testing, closing the graph would not result in the program as a whole closing. I remain unsure of this was an issue that has been unknowingly fixed from the prior prototype, and so I shall remain on the lookout for this reoccurring later on.

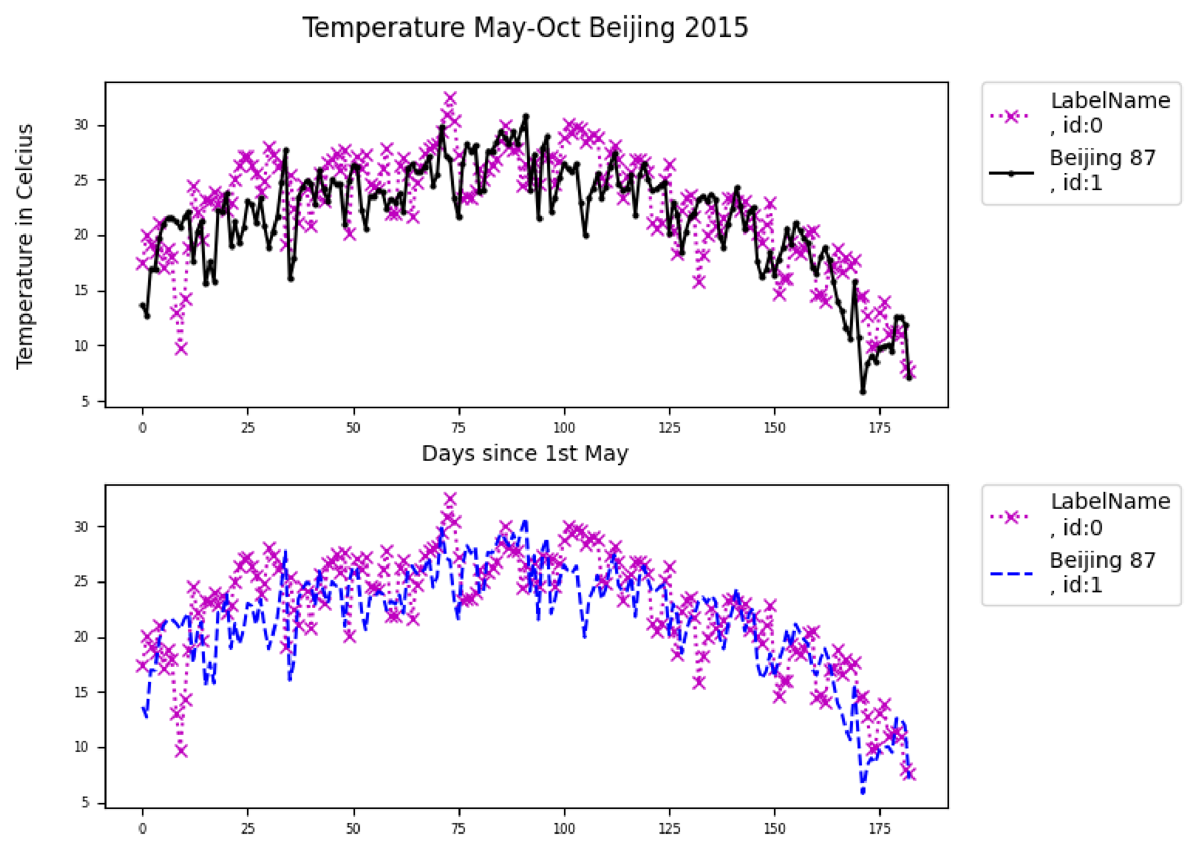
Furthermore, the best fit line would not be able to produce an output when provided with the date-time formatting of the Date columns from Beijing 1987 data, or other sets in the Large Data Set. However, I had applied the ability to chose an additional None column, whereby x values are set to increment by 1 for each y value. Thus, a line could be calculated.







Above is the line being implemented on a scatter graph of the Temperature in 1987 Beijing, and also an example of the titles for the graph, x and y axis. Furthermore, the functions can be applied to alternate graph types, such as the line graph below. Importantly, the graphs also appear the same in formatting and style across platforms, so there are no device-specific discrepancies to account for with the users. The additional confirmation step introduced for the user to confirm changes being made to a graph would be particularly useful in avoiding errors being carried over and permanently affecting the graph. Shown above is an example whereby inputting a set of data has lead to a ruined graph. Whilst a remove data feature is also present, having an additional barrier is much safer for the user.

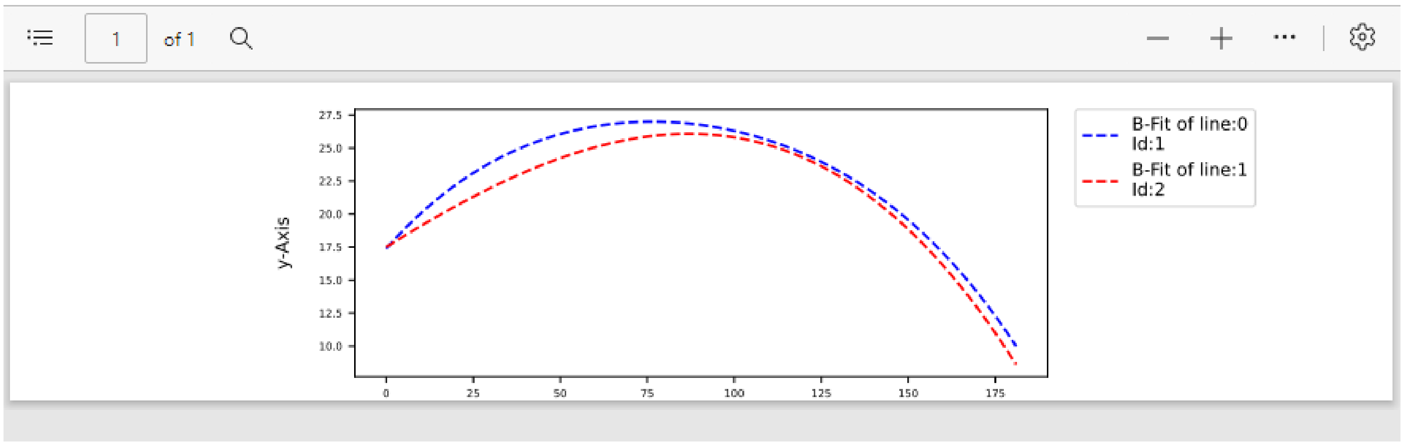


Graph Management – Evaluation and Testing

Function	Working	Score / 10	Errors / Issues	User Feedback
Labels and IDs	Y	5	Ids don't update despite being necessary to refer to lines index in graph	Make Id's either update or consistent
Design Elements	Y	8		Line width options, and alternative grid formatting (logarithmic etc)
Titles Application	Y	10		
Save Graph as pdf	Y	5		Title, x-axis cut off

Another section to the Graphing screen would be methods through which the user could manage already implemented graphs. However, there were multiple recommended improvements for me to make after feedback from the test users. Most important would be how the IDs were utilised – whilst constructing the program I neglected to note that IDs shown in graph labels would not necessarily continue to apply once a line was removed from he graph. As a result of this, line IDs quickly become obsolete, and it is a matter of the user remembering how to identify a line. For prototype 3 I shall rework how the ID system works, such that the IDs are constantly updated to accurately represent the line when labelled in the graph legend. A little more niche was the requests for alternative styles of graph formatting – the most prominent of which was a logarithmic style or format. I aim to also permit this in the next prototype, and the line width formatting in the next prototype.

The graph saving function would be difficult to program due to the constant changing of the axis formatting. As a result, when the final testing came around in must have made a change that was not accounted for in the bounding of the pdf image, resulting in the title and x-axis being cut off. This is shown in the example below.



This also provides an example for another small but necessary complaint to deal with. With no method to change label names, and the lines of best fit only referring to the line id that produced them (which is also liable to error, per the previously mentioned complaint), it can lead to confusing legends where there is no suitable name able to be entered for graph values.

Importing and Reading Datasets – Evaluation and Testing

Function	Working	Score	Errors / Issues	User Feedback
File Finder	Y	7	Device Specific issues with “options” for filters	Remove the filters option, not necessary. Importing files each time program opened is slow. Auto import chosen files?
Read_excel	Y	10		
Parse excel	Y	10		
Read_csv	Y	9	One csv file had issue opening in python, however, may just be an error in the file as it was an isolated case.	
Parse csv	N	0	Did not work.	
Dialogue Box	Y	10		Whilst intended to notify the user of events occuring inside the import screen, would be very useful as a general dialogue box to inform users of errors in the whole program
Default Option	Y	10		
Excel Import Configurations	Y	10		
Sheet Selection	Y	10		Automatic update is good, can it be carried over to

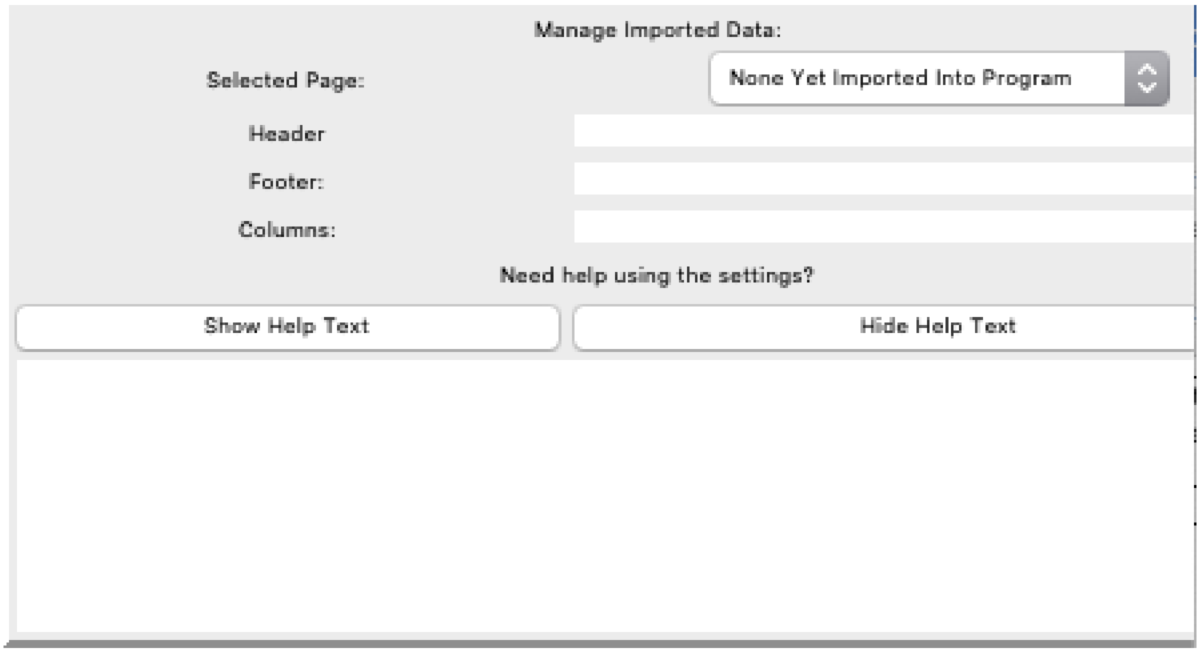
				other areas to replace "update" function buttons?

The File searcher in the program leads to version conflicts with how devices handle file searching. Testing between MacOS and Windows, results in an almost identical execution of the code, however, MacOS cannot use the filetypes filters. These result in the user being unable to select any file. In future prototypes, I will simply remove the filetypes as the test-users have not marked the feature as being particularly necessary, and the file type filters do not save much time when searching through for a dataset in storage.

Whilst coding for the read_csv function, I would not properly ensure its functionality and assume that the same processes could be applied to it as were with read_excel. Initially, the program would successfully import the csv files as requested and without issue. However, upon trying to apply the files to methods for graphing and displaying the contents they would raise numerous errors over the methods being applied. Unfortunately, this would only be caught in the user testing of the program, as I had done my personal testing insufficiently – Only using excel data sheets for the purposes of testing features. For future prototypes I shall take greater care in ensuring that all features and functions are tested properly.

Another request of the test users was a method through which the program automatically imported previously selected files with the settings that were previously applied. Currently, upon closing the program the user will not have any data retained in the program. Hence, they must re-import all prior files to use them again. Going through all the settings and finding the particular files can take a bit of time, especially when the user imports a large number of files, and so I shall aim to introduce a solution in the next prototype.

The dialogue box would also turn out to be a user-favourite feature for its potential as a general dialogue box. In this prototype, it only serves as a way through which the program can notify the user of successful imports into the program, when the program has failed to import anything or what is necessary to fix those issues. Users believe that having it be retained as a permanent feature across the screens would be very helpful to guide them for what the program believes is missing or currently occurring, and assist when issues arise. As a result of this feedback, I will

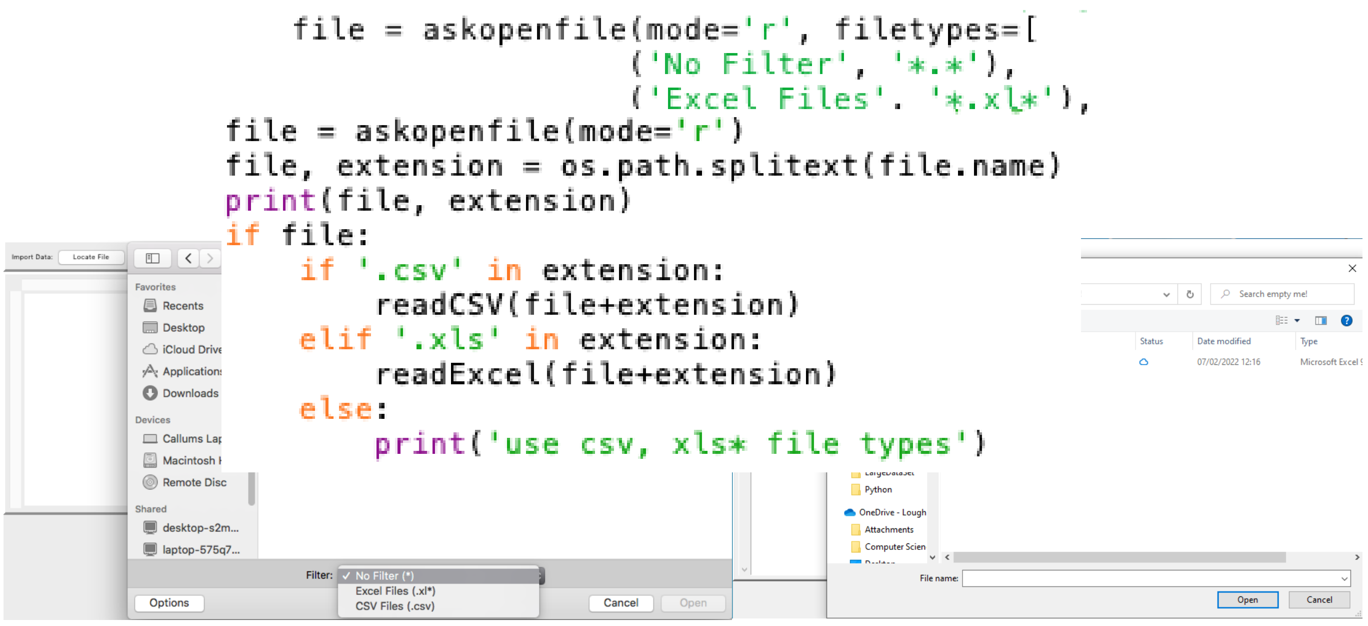


ensure greater functionality for the box, as was present previously.

With regards to the Default options, they were once again present and in the case of file importing, the default values would see Header, Footer set to 0, and Columns set to "a:e". furthermore, I implemented a method through which the user could change the default values for Header, Footer, and other non-import screen values in the settings screen. This was greatly appreciated by users as an improvement and follow up to the prior Default values implementation in prototype 1. The ability to select Headers, Footers and Columns would remove the often present explanatory text in Excel Files, which can also lead to errors in how Pandas reads the file. Furthermore, users noted that the immediate updating of the Sheet selection optionmenu (labelled with "Selected Page" to its left) was a great quality of life feature rather than having to click "update" or similar function buttons. These buttons sometimes resulted in confusion elsewhere in the program and could not provide the immediate response or feedback in the graphs that users desired to see. Hence, I shall ensure that Prototype 3 utilises immediately updating functions and remove the buttons like "Preview" and "Update Columns" that only serve to refresh a particular widget.

Importing and Reading Data Sets - Development

Askopenfile is an additional Tkinter tool that allows for the program to access the devices' in-built file finder application. This does open a new second window, however, it should not be the cause of any significant issue. Whilst I did move to a singular window design to avoid clutter, a short-term temporary second window does not cause significant enough mess or clutter that I feel it to be a hindrance to the user. The method works through opening the file finder, and detecting when a file is selected. This file is saved to the temporary variable "file", and then its path is obtained. Furthermore, additions such as "filetypes" can be added for the user to apply the provided filters to their file search.

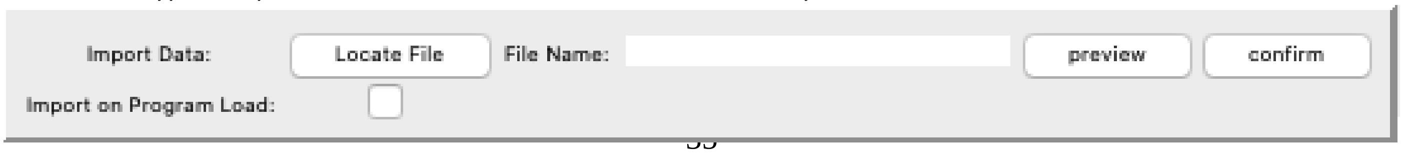


```
file = askopenfile(mode='r', filetypes=[
                        ('No Filter', '*.*'),
                        ('Excel Files'. '*.xl*'),
```

```
file = askopenfile(mode='r')
file, extension = os.path.splitext(file.name)
print(file, extension)
if file:
    if '.csv' in extension:
        readCSV(file+extension)
    elif '.xls' in extension:
        readExcel(file+extension)
    else:
        print('use csv, xls* file types')
```

Upon detecting that a file has been selected, the program then simply uses a number of if statements to check the file extension of the document being imported. Depending on the file extension, the program calls for a particular subroutine to deal with it. In the case of a .xls* file, it will utilise the read_excel function, and so forth.

File.name is employed as the askopenfile method returns an IO text-wrapper, from which the os.path.splitext method cannot be directly applied. The splitext method itself serves as to split the file name into its file path and file extension / type components. The extension is then used to identify what method should be used to read the file.



Prototype 2 Overview

This prototype would follow up on and complete most of the user suggestions and issues from the first prototype. The new Default values implementation was a success, as was the importing of files through a GUI. The only note that has not yet been fully met from the prior prototype is the incorporation of users being able to graph their own functions. Functions can be used to model a vast variety of situations, which increases in potential with the more functions that can be applied. For the next prototype, I will attempt to find a parsing method in which the user can enter a function using trigonometric functions, exponents, and potentially even map x and y to another independent variable for simultaneous equations. Particularly useful for modelling situations are differential equations, whereby the user can model populations of animals or concentrations of chemicals given a set of initial conditions.

For the next prototype, I will also look towards classes to condense the program into a more organised and manageable format. This may also be applicable to the Tkinter widgets, which could save time in creating and organising them.

The most crucial aspect to ensure is working in the next prototype is the non-excel document reading functions, which will apply to csv files and online URLs. Currently the CSV parser is broken, however, it should be a simple fix. Ensuring these features are working is necessary as the stakeholders take interest in being able to utilise and graph a wide variety of information sources and databases.

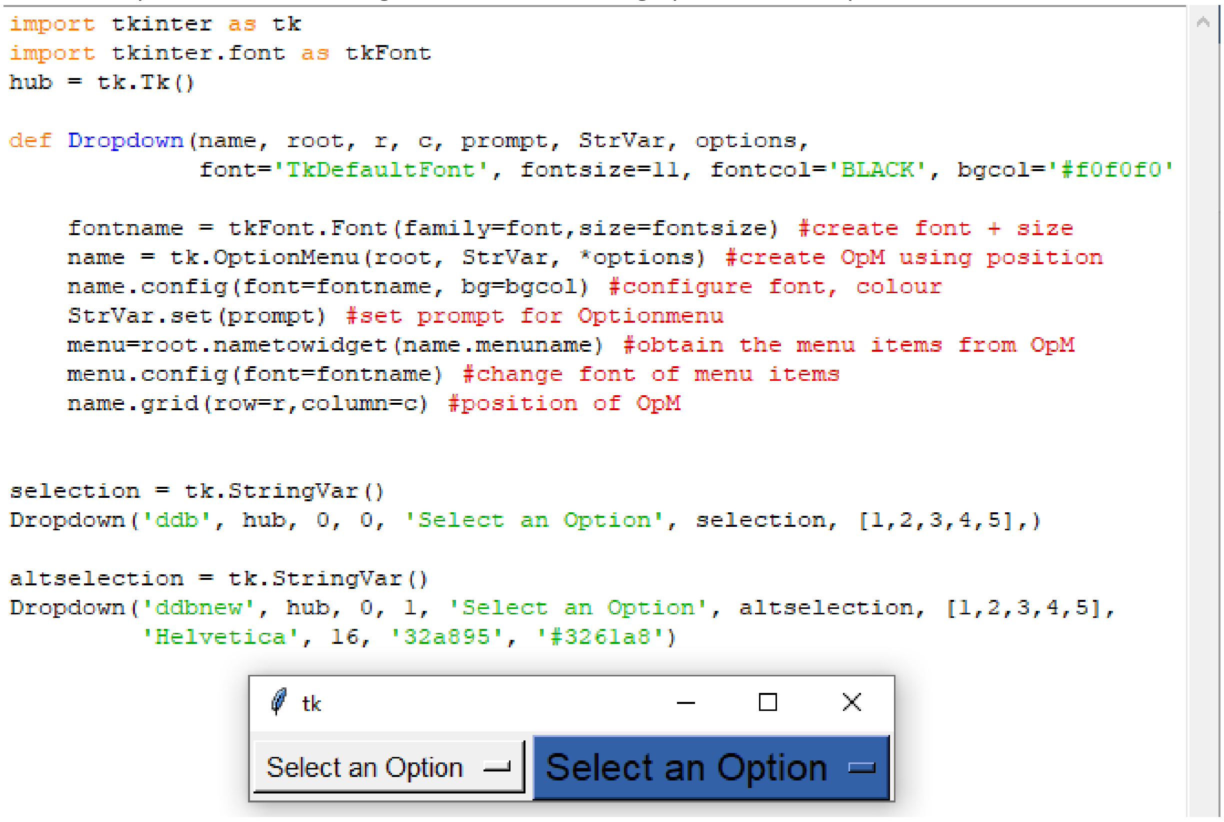
Aims of Prototype 3

For prototype 3 I intended to introduce classes to manage the Tkinter widgets, alongside for constructing the various screens of the program.

With regards to the actual content of the program, I intended to introduce a function constructer that had far more utility than the current polynomial constructer, alongside a normal distribution being available to check for anomalous results in data sets.

Class Widgets – Development

My first attempt at optimising the program's code would be with the implementation of classes for the various widgets to be incorporated in the program. This appeared to be working with the OptionMenu Classes, and Labels, and allowed for quick creation of widgets rather than taking up numerous repetitive lines.

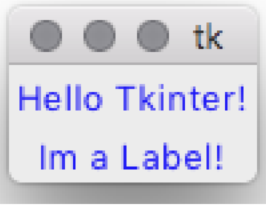


```
import tkinter as tk
import tkinter.font as tkFont
hub = tk.Tk()

def Dropdown(name, root, r, c, prompt, StrVar, options,
             font='TkDefaultFont', fontsize=11, fontcol='BLACK', bgcol='#f0f0f0'

    fontname = tkFont.Font(family=font,size=fontsize) #create font + size
    name = tk.OptionMenu(root, StrVar, *options) #create OpM using position
    name.config(font=fontname, bg=bgcol) #configure font, colour
    StrVar.set(prompt) #set prompt for Optionmenu
    menu=root.nametowidget(name.menuname) #obtain the menu items from OpM
    menu.config(font=fontname) #change font of menu items
    name.grid(row=r,column=c) #position of OpM


selection = tk.StringVar()
Dropdown('ddb', hub, 0, 0, 'Select an Option', selection, [1,2,3,4,5],)

altselection = tk.StringVar()
Dropdown('ddbnew', hub, 0, 1, 'Select an Option', altselection, [1,2,3,4,5],
         'Helvetica', 16, '32a895', '#3261a8')
```

However, I began to encounter issues when attempting to produce a similar class for the frames of the program. I had difficulties in being able to select a specific frame to house a widget, which was a significant hinderance to producing a good layout of the program. Furthermore, the classes were prone to bugs with displaying and later interactions with the widgets. There is potential in saving numerous repetitive lines of code by implementing classes for the various widgets, however, it was causing too many troubles for being immediately worthwhile.

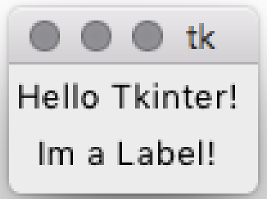
Using variables for widget names would be used in the first versions and work done on the program. It was also a fast method to set up for quick testing of features and widgets, and allowed for an easy way to interact with the widget at a later occasion, or upon a particular event. Despite this, in the larger prototype programming, containing all the numerous variable names was getting somewhat overwhelming. Whilst the variables were quick for setting up small test programs, they would become slower when implementing lengthy variable names into a program. Lengthy names were also required to differentiate themselves uniquely, and could not be avoided. As a result of this, I would look towards alternative ways in producing the widgets again. Over the next developments, I would re-utilise producing widgets without variables for a new method of declaring Tkinter widgets:

```
import tkinter as tk

window = tk.Tk()

labels_List = []
labels_List.append(tk.Label(window, text='Hello Tkinter!'))
labels_List.append(tk.Label(window, text='Im a Label!'))

i=0
for label in labels_List:
    label.grid(row=i,column=0)
    label.configure(fg='blue')
    i+=1
```



Shown here is the original form of the label widget without a variable inside of a list. Through appending them to a list, I effectively grant them a variable through which I can interact with each label – but they do not have any individual name, only the list name and their position in the list. But this method does have additional benefits as a 2D list could be used to store multiple lists in a single overarching list. With this, the widget itself and necessary information to correctly place it can be used – this would include values for row, column, sticky and potentially any extra functions or variables needed by particular widgets to function – such as the StringVar() and list in an OptionMenu widget. This idea is implemented in the code below:

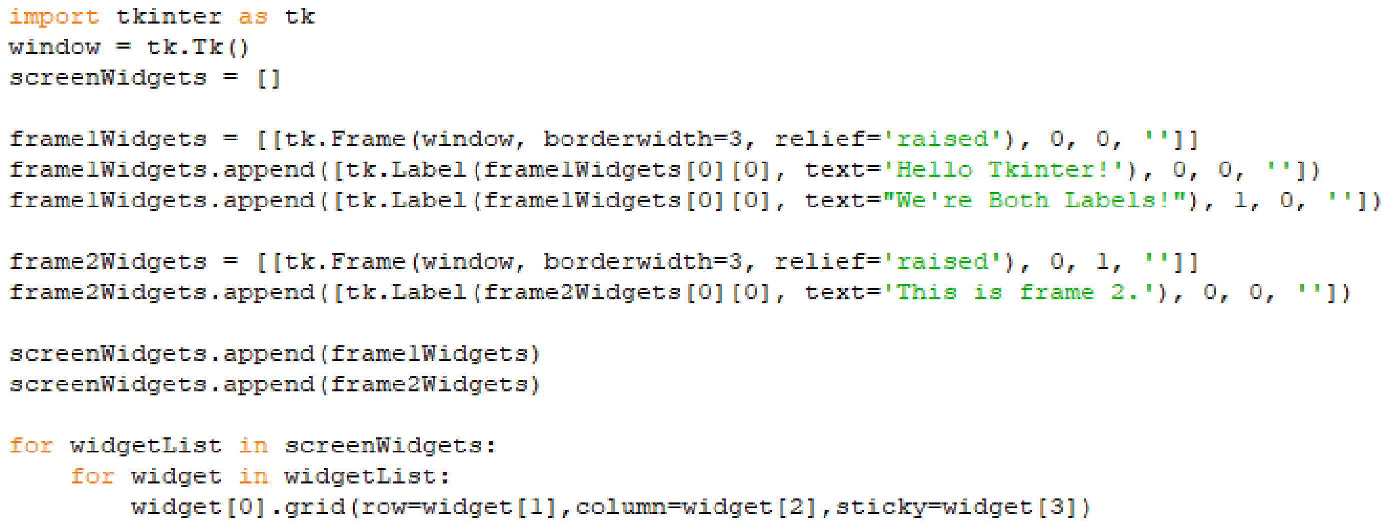
```
import tkinter as tk

window = tk.Tk()

widgetList = []
widgetList.append([tk.Frame(window),0,0,''])
widgetList.append([tk.Label(widgetList[0][0], text='Hello Tkinter!'),0,0,''])
widgetList.append([tk.Label(widgetList[0][0], text='Im a Label!'),1,0,''])

for label in widgetList:
    label[0].grid(row=label[1],column=label[2])
```



The code is modified to a small degree in order to accommodate the grid positions, sticky value and a Frame into a general widget list, and a near-finalised alternate method for widget declaration. This method makes for a far more optimised and efficient way to produce Tkinter widgets than declaring them individually under their own variable and having the necessary grid information in a separate list. However, it loses the benefits of individual variables in that the widgets are largely lost in meaning when directly inserted into a List. Furthermore, coding becomes very difficult when requiring a reference to a list position for each use of a Tkinter widget.

```
                    widgetList[widgetPosition][tk.Widget index]
self.Widgets.append([tk.Label(self.Widgets[0][0],text='Import Data:'),
                    0,0,''])
self.Widgets.append([tk.Button(self.Widgets[0][0], text='Locate File',command=lambda:self.openData()),
                    0,1,''])
self.Widgets.append([tk.Label(self.Widgets[0][0],text='File Name:'),
                    0,2,''])
self.Widgets.append([tk.Text(self.Widgets[0][0], width=25, height=1),
                    0,3,''])#Widgets[6]
self.Widgets.append([tk.Button(self.Widgets[0][0], text='preview',command=lambda:self.previewImport()),
                    0,4,''])
self.Widgets.append([tk.Button(self.Widgets[0][0], text='confirm',command=lambda:self.confirmImport()),
                    0,5,''])
```

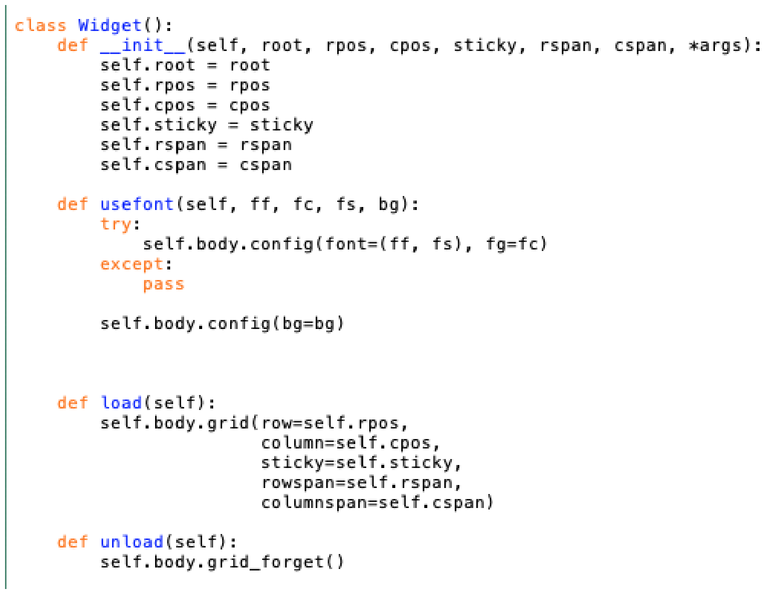
Whilst the tk.Widget index to directly access the widget for .grid and .config to be called can always be the same through using a generalised formatting method, the widget position in the list will grow harder to identify with the greater the program gets in size and number of widgets. I would employ a method of easing these difficulties through having multiple widgets lists in each screen - named according to the Frame that its respective widgets are placed in. Through this, there would be less widgets in each individual frame's list and it would make the programming easier though making the required indexes of widgets easier to find. The lists could be structured as shown in the previous images, with only the addition of a ScreenWidgets list to contain all individual frame's Widget Lists.



```
import tkinter as tk
window = tk.Tk()
screenWidgets = []

frame1Widgets = [[tk.Frame(window, borderwidth=3, relief='raised'), 0, 0, '']]
frame1Widgets.append([tk.Label(frame1Widgets[0][0], text='Hello Tkinter!'), 0, 0, ''])
frame1Widgets.append([tk.Label(frame1Widgets[0][0], text="We're Both Labels!"), 1, 0, ''])

frame2Widgets = [[tk.Frame(window, borderwidth=3, relief='raised'), 0, 1, '']]
frame2Widgets.append([tk.Label(frame2Widgets[0][0], text='This is frame 2.'), 0, 0, ''])

screenWidgets.append(frame1Widgets)
screenWidgets.append(frame2Widgets)

for widgetList in screenWidgets:
    for widget in widgetList:
        widget[0].grid(row=widget[1],column=widget[2],sticky=widget[3])
```

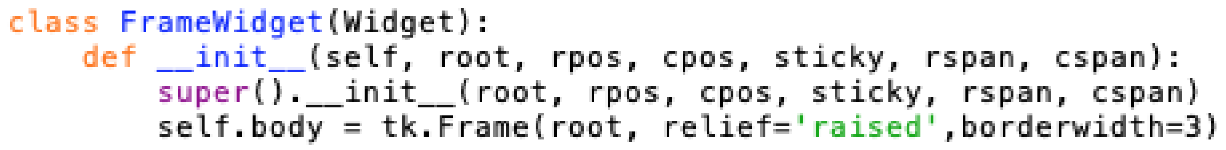
What I now aimed to do was to increase the efficiency through which I could create and manipulate the Tkinter Widgets. Whilst the process of actually creating them would not be much faster than a single line declaration, for features such as entry validation, updating option menus and tracing changes, each function had to be constructed separately. Through introducing a class, elements such as a StringVar could be built into the Widget's class, removing any need to go out of my way to create each one as a separate variable. Rather than checking for option menu widgets in the main program and employing a particular loading sequence to deal with their menus being updated, I could provide an "update" function directly to the Option Menu class.

The difficulty would arise due to the variety of widgets, and that many had exclusive tags that could be applied. For example, Text Widgets could not hold a StringVar or IntVar, whilst Entry Widgets could. Hence, my first action was to create a general widget class that could house universally applicable values.



```
class Widget():
    def __init__(self, root, rpos, cpos, sticky, rspan, cspan, *args):
        self.root = root
        self.rpos = rpos
        self.cpos = cpos
        self.sticky = sticky
        self.rspan = rspan
        self.cspan = cspan

    def usefont(self, ff, fc, fs, bg):
        try:
            self.body.config(font=(ff, fs), fg=fc)
        except:
            pass

        self.body.config(bg=bg)



    def load(self):
        self.body.grid(row=self.rpos,
                       column=self.cpos,
                       sticky=self.sticky,
                       rowspan=self.rspan,
                       columnspan=self.cspan)

    def unload(self):
        self.body.grid_forget()
```

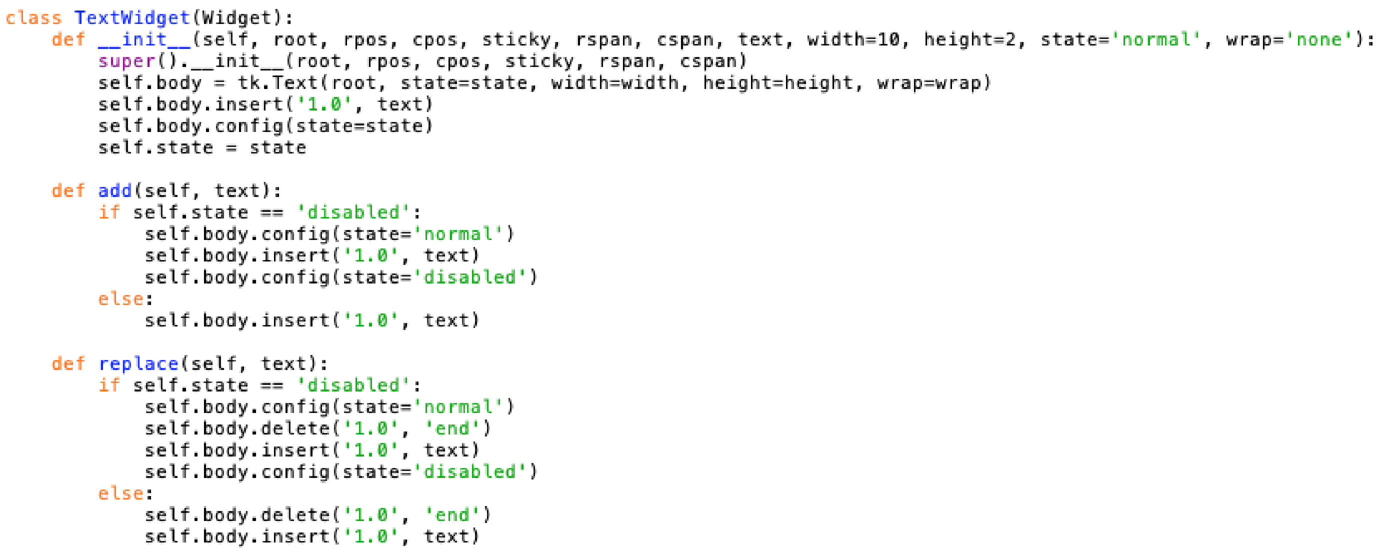
Shown above is the main “Widget” class, utilised as a superclass for each individual widget. It houses the common values, alongside the load and unload functions. Furthermore, a small section to apply font and background colours is added, to apply the user-selected accessibility changes. These 3 functions, however, are all build around another value not stored within the Widget Class itself. “self.body” is a variable used by each subclass to store the actual Tkinter Widget object. From there, Tkinter’s own widget methods of grid and grid_forget() can be directly applied. These subclasses also inherit the Widget Class’s functions and variables, and are the ones called for upon widget creation hence allowing for them to work.



```
class FrameWidget(Widget):
    def __init__(self, root, rpos, cpos, sticky, rspan, cspan):
        super().__init__(root, rpos, cpos, sticky, rspan, cspan)
        self.body = tk.Frame(root, relief='raised',borderwidth=3)
```

The frame class is the simplest, requiring only a root object, row and grid positions to be constructed. Beyond frames, each widget may utilise additional tags, such as text, width, height, commands and others, the majority of which are only needed to initialise the widget and can be dismissed afterwards.

As an example of widget-specific commands, the Text widget employs an “add” and a “replace” command, which either appends text to the end of the current string, or replaces it entirely. Rather than having to alter the state of the Text Widget from “disabled” to “normal” each time I need to edit the text within a Text Widget, utilising a common function makes coding much easier and quicker.



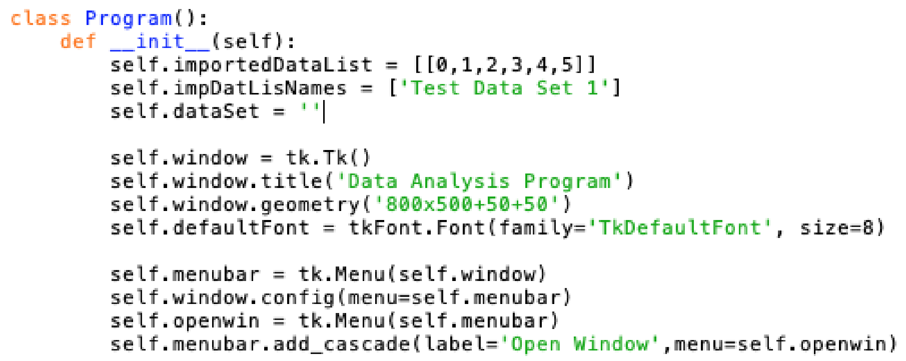
```
class TextWidget(Widget):
    def __init__(self, root, rpos, cpos, sticky, rspan, cspan, text, width=10, height=2, state='normal', wrap='none'):
        super().__init__(root, rpos, cpos, sticky, rspan, cspan)
        self.body = tk.Text(root, state=state, width=width, height=height, wrap=wrap)
        self.body.insert('1.0', text)
        self.body.config(state=state)
        self.state = state

    def add(self, text):
        if self.state == 'disabled':
            self.body.config(state='normal')
            self.body.insert('1.0', text)
            self.body.config(state='disabled')
        else:
            self.body.insert('1.0', text)

    def replace(self, text):
        if self.state == 'disabled':
            self.body.config(state='normal')
            self.body.delete('1.0', 'end')
            self.body.insert('1.0', text)
            self.body.config(state='disabled')
        else:
            self.body.delete('1.0', 'end')
            self.body.insert('1.0', text)
```

Shown above is the Text Widget Class, and it is evident as to why a superclass and subclass structure was necessary. The Widget utilises numerous more tags than frame, many of which would either raise errors or function unexpectedly if applied to other classes. hence, necessary elements to be stored are stored within separate Widget classes, and only the common elements are passed into the superclass.

Program and Screen Class - Development

The only other remaining area where I could incorporate a Class to save time in coding was with the program screens. As previously mentioned, all “screens” would be presented through a single window, being interchangeable with each other through the menu bar system. At first, the work would be split into the Program Class and Screen Class. The Program Class would be a general code class – a parent class for Screen and to house necessary background code. It would house any data sets imported into the program, construct the window in which the program operates, and construct the menu to switch between the other screens. It is not intended to be a framework for multiple Program objects, but instead a framework for multiple Screen Objects. The functions for unloading and loading would also have to be incorporated into this Class, as they would require access to each screen's widgets and their position values.

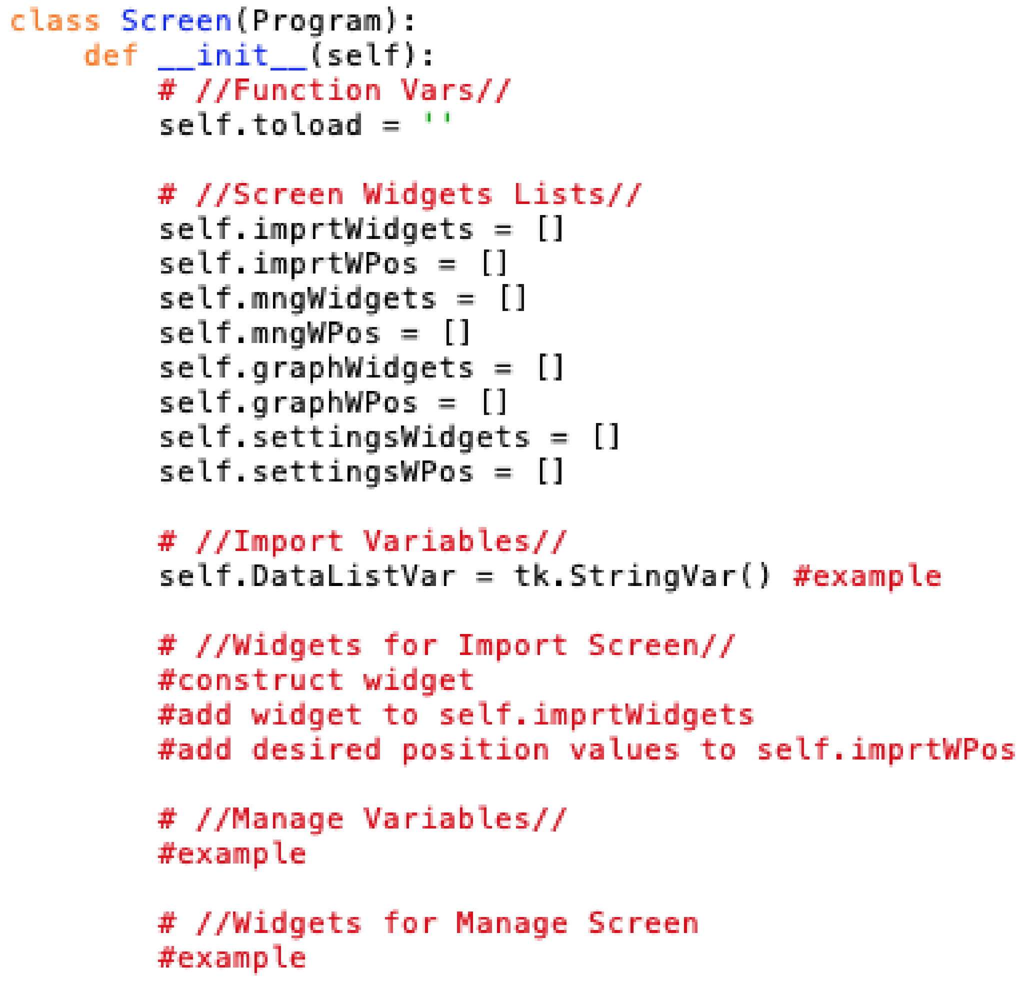
Largely, the Program Class would not change in development much. Its position has always been as the background



```
class Program():
    def __init__(self):
        self.importedDataList = [[0,1,2,3,4,5]]
        self.impDatLisNames = ['Test Data Set 1']
        self.dataSet = ''

        self.window = tk.Tk()
        self.window.title('Data Analysis Program')
        self.window.geometry('800x500+50+50')
        self.defaultFont = tkFont.Font(family='TkDefaultFont', size=8)

        self.menubar = tk.Menu(self.window)
        self.window.config(menu=self.menubar)
        self.openwin = tk.Menu(self.menubar)
        self.menubar.add_cascade(label='Open Window',menu=self.openwin)
```

worker, uniting the various subclasses and providing a global hub and simply performing set-up for the greater program.

The Screen Class would be the more complicated to code, as it was intended to be capable of constructing the foundation of each vastly different screen in the program. However, each screen would have its own independent functions – if they were inserted into the Screen Class, then they would be duplicated for each new screen being made. These functions would have specific variables being used with each class, resulting in errors arising by simply existing outside of their correct screen. Furthermore, it may lead to name conflicts if not closely monitored – not to mention that duplication of unnecessary functions is inefficient. Out of concern for the likely difficulties in coding such a design, I would change my approach to the Screen Class (and all classes incorporated following this instance). This led to the idea of utilising the Screen Class to create only a single "Screen" Object – a single-use method of housing widgets and other functions, rather than as a shell for multiple objects to be constructed. The intended layout for that Screen Class is shown below:



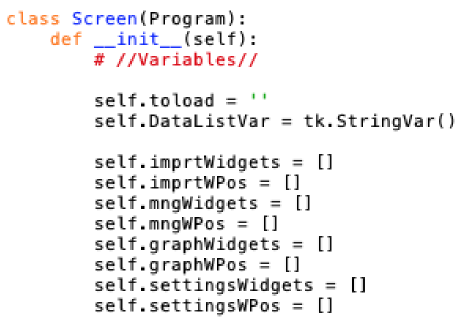
```
class Screen(Program):
    def __init__(self):
        # //Function Vars//
        self.toload = ''

        # //Screen Widgets Lists//
        self.imprtWidgets = []
        self.imprtWPos = []
        self.mngWidgets = []
        self.mngWPos = []
        self.graphWidgets = []
        self.graphWPos = []
        self.settingsWidgets = []
        self.settingsWPos = []

        # //Import Variables//
        self.DataListVar = tk.StringVar() #example

        # //Widgets for Import Screen//
        #construct widget
        #add widget to self.imprtWidgets
        #add desired position values to self.imprtWPos

        # //Manage Variables//
        #example

        # //Widgets for Manage Screen
        #example
```

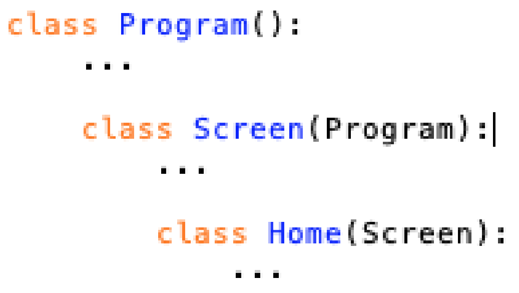
The new Screen Class would have some of its own background variables for the purpose of operating functions such as loading and unloading screens – these could be moved out of the Program Class and more connected to the screens. Due to all necessary variables relating to the widgets and their positions being in a single instance of Screen, the loading and unload functions could also be present.

Whilst the design would function, it would leave the code in a rather dense state, whereby it is quick and easy to get lost amongst the various widgets. Additionally, all the subroutines would be mixed together: which screen is using which variables / subroutines / widgets would not be immediately obvious. As a result, I would have difficulties in troubleshooting errors and general coding with this prototype iteration.

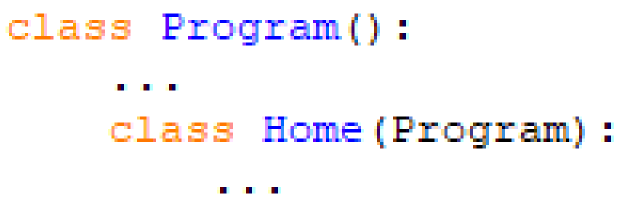
Whilst the design was not a "failure", it was sub-optimal to continue with, and I believed a far better method could be obtained. The method of constructing a single screen in the Screen Class was simple enough. All methods and subroutines, alongside necessary variables are kept local and accessible. There is no overlap with other objects in the Screen Class. Issues were only arising when attempting to utilise the singular Class for multiple objects. This led to the development of a long-standing Class construct: each screen would be built using its own individual Class.

Initially, I would incorporate the Screen Class as a method of interconnecting the different screens and their individual Classes. It would store variables to house the Widgets and their positions from each of the various screens (Home, Import, Graph, Settings, etc) for utilisation in the loading and unload functions. Furthermore, it would also serve as a bridge between the Program Class and the individual screen Classes in a "One to One" and "One to Many" relationship. A method of providing management over all the different screen subclasses.


```
class Screen(Program):
    def __init__(self):
        # //Variables//

        self.toload = ''
        self.DataListVar = tk.StringVar()

        self.imprtWidgets = []
        self.imprtWPos = []
        self.mngWidgets = []
        self.mngWPos = []
        self.graphWidgets = []
        self.graphWPos = []
        self.settingsWidgets = []
        self.settingsWPos = []
```


```
class Program():
    ...

    class Screen(Program):
        ...

        class Home(Screen):
            ...
```

But, due to the fact that this resulted in an unnecessary middle-man between Program and the Screen Subclasses, I opted to instead remove the Screen Class entirely. Instead, all universal functions would be returned to the Program Class, and the individual screen classes are all direct subclasses to Program. The Program class does become more inflated with subroutines that I would have preferred to be more explicitly attached to the screens, however, the program is simpler and functions efficiently.


```
class Program():
    ...
    class Home(Program):
        ...
```

The Info Box – Removal from the Program

The info box was an element from prototype 2 which I believed could be implemented further in this prototype. The test users had acknowledged it as a potentially beneficial feature too. However, I decided to drop this feature as being largely not useful. Much of the program's feedback to the user is done through various text boxes and graphs being updated with new or altered information. Failure to read values, or success, is clearly corroborated through this, without the need of a little box to directly say "Program failed to do x". whilst some test users believed there could have been potential, I did not deem it to be worth the time to implement.

Loading Widgets and Screens – Development

There would be some minor changes introduced to Prototype 2's loading and unloading subroutines due to the introduction of widgets in frame lists, and the screen.Widgets lists. Primarily this involved adding an additional loop so that both the framesList, and the widgets inside each frame can be iteratively processed. Each of these Widgets itself is also a list, consisting of the widget object and then the necessary grid values or additional functionality variables it requries. This, grid values is removed from the loading functions and replaced with referring to specific positions within the Widget list. These positions are kept consistent, such that Widget[0] always referrs to the actual Tkinter object, positions [1,2,3] refer to the x, y, and sticky grid values, and [4,5] are reserved for additional values, or alternatively not used with the list ending on the sticky value.

Whilst the screenSwitch method was also utilised in Prototype 2, it wasn't explicitly shown. But like the loading subroutines, it features little change – save for the references to specific screen objects. Ultimately it still exists to manage the transfer from one screen of Widgets to another.

```
def screenSwitch(self, screen):
    '''screen is where the subclass screen object is used to obtain its widgets'''
        if self.currentload != '':
            self.unload(self.currentload.Widgets)
        self.currentload = screen
        self.load(screen.Widgets)

def load(self, framesList):
    '''Load all widgets in framesList that is passed to load function.
        may be 1+ widgets in length.'''
    for frameWidgets in framesList:
        for widget in frameWidgets:
            try: widget[0].config(font=self.defaultFont,fg='black')         #attempt to apply font
            except: pass                                                    #exception for widgets like Frame
            widget[0].grid(row=widget[1],                                   #apply grid values
                           column=widget[2],
                           sticky=widget[3])
            if len(widget) == 6:                                            #check for extra values
                if isinstance(widget[0], tk.OptionMenu):                    #OptionMenu always given extra values
                    widget[0]['menu'].delete(0,'end')                       #to hold StringVar selection variable
                    for item in widget[5]:                                  #and list of options to be used
                        widget[0]['menu'].add_command(label=item,
                                command=tk._setit(widget[4], item))
                    widget[4].set(widget[5][0])
                else:
                    widget[0].grid(rowspan=widget[4],columnspan=widget[5])  #no other exceptions, thus assumption

def unload(self, framesList):
    '''remove passed widgets from grid positions'''
    for frameWidgets in framesList:
        for widget in frameWidgets:
            widget[0].grid_forget()
```

However, this was built to serve for an alternative method of widget construction than would ultimately be used. It anticipates multiple lists of multiple widgets, and only serves a maximum of 5 additional values – some of which can be utilised for vastly different purposes, and raise numerous errors if declared incorrectly.

```
frame1Widgets = [[tk.Frame(window, borderwidth=3, relief='raised'), 0, 0, '']]
frame1Widgets.append([tk.Label(frame1Widgets[0][0], text='Hello Tkinter!'), 0, 0, ''])
frame1Widgets.append([tk.Label(frame1Widgets[0][0], text="We're Both Labels!"), 1, 0, ''])
```

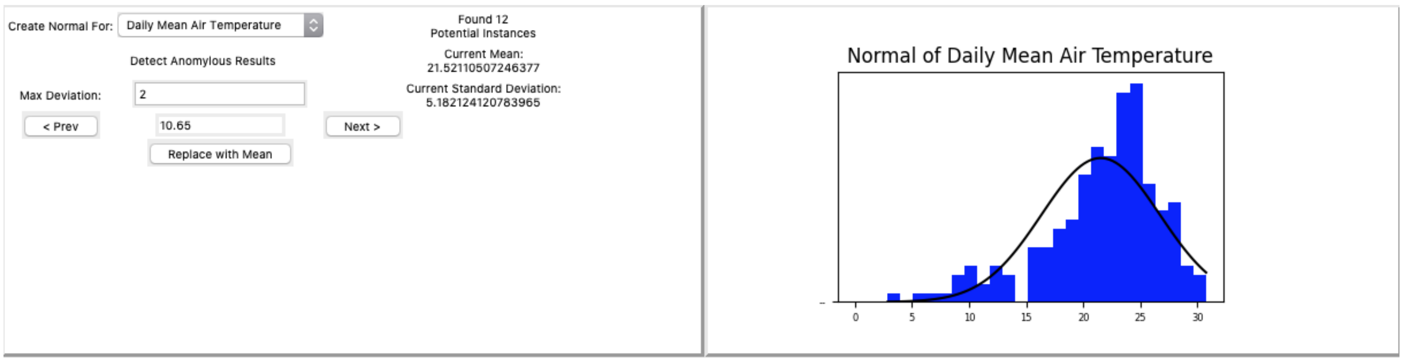
With the new method of creating Widget Classes, there was a built-in grid function for the Widgets which could be called. Furthermore, checking for changes to widgets – such as the option menu's menu configuration shown in the load function – were not necessary, as they were done within other subroutines and did not require an extensive loading sequence. Instead, the program had to apply new fonts to the labels – should the font settings be changed – and apply them when loading a widget. Furthermore, the previous design assumed that the entered value would be a list of lists, each of which was a list of widgets.

$$[\,[\,Frame0\,], [\,[Widget1], [Widget2], [Widget3]\,], [Frame2]]$$

For the new load function design, it would check if the entered value is a list of lists, or just a single list of widgets. This avoids confusion I had previously, where forgetting to surround a single – or list of -widgets in the correct number of brackets [ ] would result in errors as the program attempted to iterate through a non-existent list. The new design does not check for a single widget being passed into the function, as single-widget loading does not need to occur.

Normal Distribution Anomalies – Development

The Normal Distribution comparisons would be an addition to the Import Screen, whereby the user would compare a Histogram of the currently selected Data column to a normal distribution model also derived from the original data. The User would first import a file, with the program reading the column headings to identify each column of data – a method later used to select a column of DateTime objects which could be converted to alternate formats. The methods I had learnt previously to introduce embedded graphs in prototype 2 would be re-utilised to create a single subplot graph in the Import Screen to house this graph. However, the majority of user interaction would be



Create Normal For: Daily Mean Air Temperature

Max Deviation: 2

10.65

Found 12
Potential Instances

Current Mean:
21.52110507246377

Current Standard Deviation:
5.182124120783965

kept to the side of this image.

Shown on the left is the layout of the user-interactions area. The basic features are labels to display the current mean and standard deviation, alongside the potential anomalies that the program has detected. As a default, anomalies are defined as being 2 standard deviations away from the mean – within 5% or less probability of occurring. However, the user can update this to be a greater or lesser integer.

The functions do result in there being no x-axis values displayed upon the graph, despite the y-axis displaying perfectly. I am not sure as to why this is occurring, however, the axis would only display probabilities of the Normal distribution, or the frequency densities of the data. As a result, it is not a significant issue that the x-axis is not present, and will not be something I seek to resolve for the Final Design.

Normal Distribution Anomalies – Testing and Evaluation

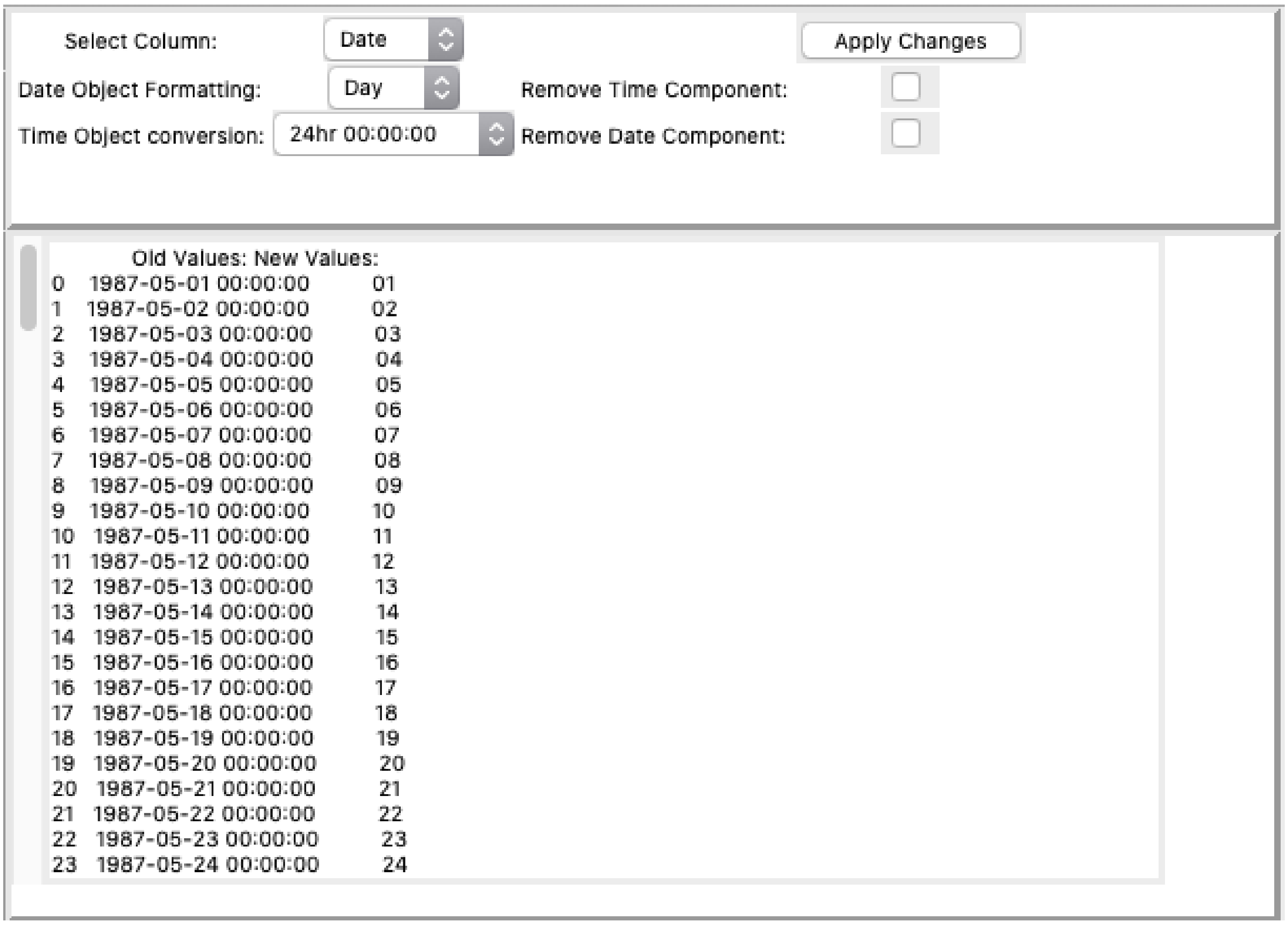
Function	Score	Score Determinants	User Feedback
Normal Graph	9	Missing x-axis is unfortunate and a potential indicator of further issues, but not much of an influence. Hence, only removing 1 point for both graphs.	Create normal distribution for a user-inputted range, to avoid weather patterns raising false anomalies
Histogram	9		
Identifying Anomalies	10	Correctly identifies values which fall outside the user-defined standard deviations, and stores mean values in the correct positions in the data set when replaced.	

Noted within the evaluative grid is a particularly interesting idea from the test users, whereby they proposed that a range could be applied to the Normal distribution to limit the values it takes in. Imposing limits in this manner would serve to remove many false anomalies because of otherwise acceptable patterns. However, the main priority of the normal comparison is to provide an overview of the data as a whole and determine anomalous results in comparison to that. It was not expected that this method would perfectly apply to every situation, but was aimed to be a generally applicable and “decent” comparison method.

Despite this, I will consider introducing it into the final design if I have additional time or simply don’t forget about it. I believe the feature would certainly be useful, even if not immediately necessary.

DateTime Management and Graphing – Development

Another new addition for this prototype would be the ability to edit the DateTime objects often used to represent dates / times in excel documents. Typically these cause issues when creating comparison graphs – due to alternate years – or when attempting to calculate lines of best fit, due to the dates not being a numerical value, or having differing years, resulting in a unreadable plot. Through providing a method to remove unnecessary Date or Time information, I aimed to permit far greater potential for graph comparisons. The only peculiarity I noticed when dealing with DateTime conversions was that reading Time objects stored as “00:00:00” returned no value, hence forcing them to be omitted from the final formatting of the DateTime, regardless of this being the user’s desire.



Select Column: Date

Date Object Formatting: Day

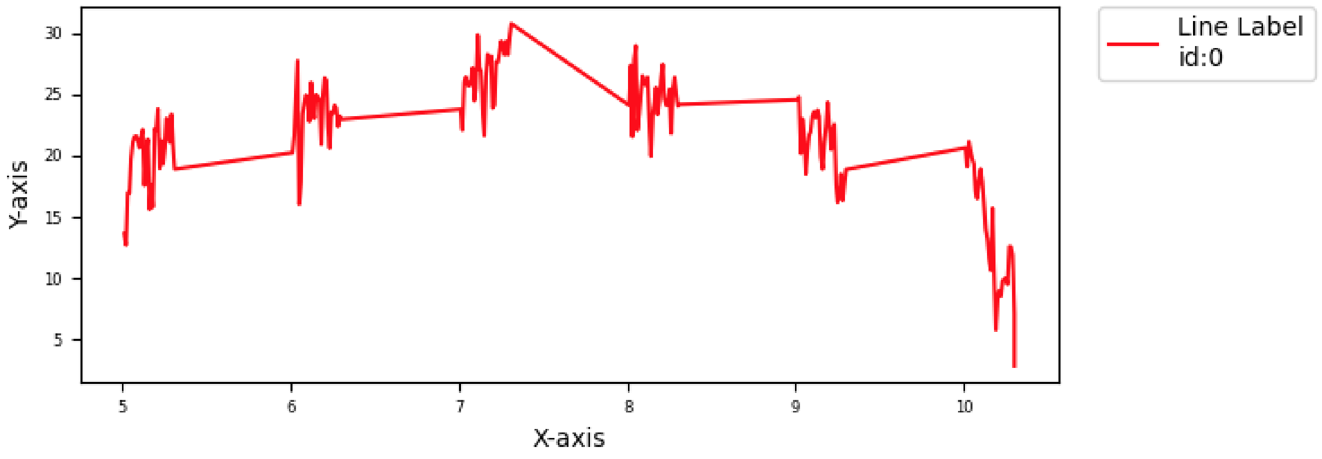
Time Object conversion: 24hr 00:00:00

Remove Time Component: ☐

Remove Date Component: ☐

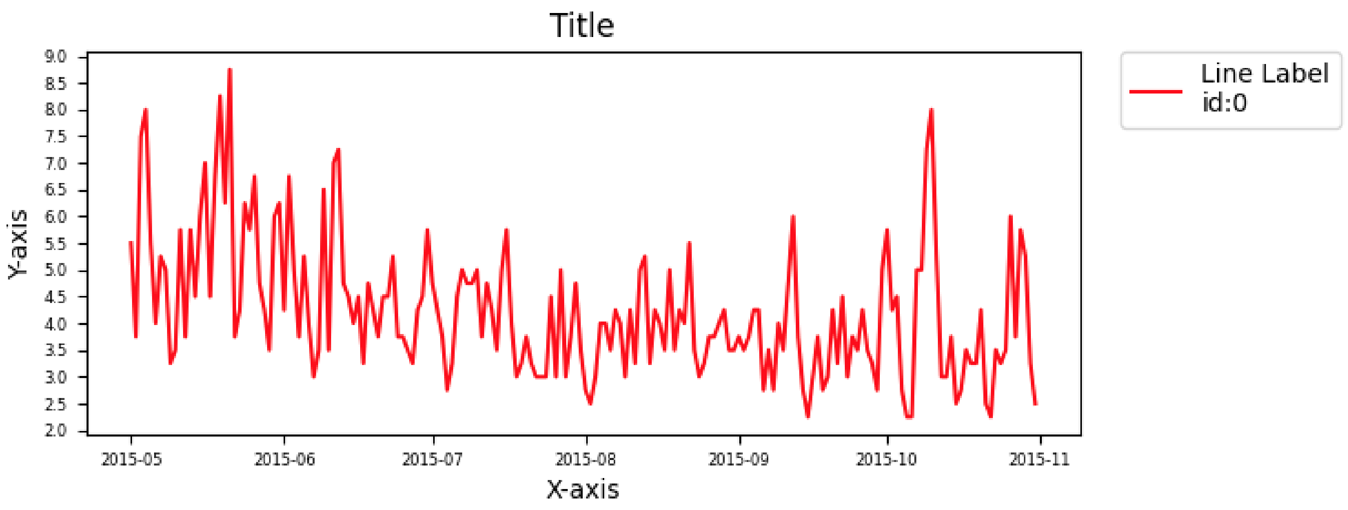
```
       Old Values: New Values:
0   1987-05-01 00:00:00       01
1   1987-05-02 00:00:00       02
2   1987-05-03 00:00:00       03
3   1987-05-04 00:00:00       04
4   1987-05-05 00:00:00       05
5   1987-05-06 00:00:00       06
6   1987-05-07 00:00:00       07
7   1987-05-08 00:00:00       08
8   1987-05-09 00:00:00       09
9   1987-05-10 00:00:00       10
10  1987-05-11 00:00:00       11
11  1987-05-12 00:00:00       12
12  1987-05-13 00:00:00       13
13  1987-05-14 00:00:00       14
14  1987-05-15 00:00:00       15
15  1987-05-16 00:00:00       16
16  1987-05-17 00:00:00       17
17  1987-05-18 00:00:00       18
18  1987-05-19 00:00:00       19
19  1987-05-20 00:00:00       20
20  1987-05-21 00:00:00       21
21  1987-05-22 00:00:00       22
22  1987-05-23 00:00:00       23
23  1987-05-24 00:00:00       24
```

I first would store converted values in the form "Year.Month.Day", with the idea that decimal could be utilised in 2-value pairs to permit numerical graphing of dates. With the most significant Date factor always remaining on the left-hand side of the decimal, it would also prevent dates jumping from 31.5 (31st May) to 1.6 (1st June) through having the format 5.31 (31st May) and the next date: 6.1 (1st June). Due to the decimals ending at .3 or .31 (Year > Month > Day formatting), and jumping to the next month integer, large lines would be drawn to connect these values, which did not represent true graphed values Best Fit lines would also be forced out of shape through this.



As a result, I decided to return to having '/' Separators for Days, Months and Years. This would raise a new issue, whereby I had too many ticks present. Matplotlib natively handles DateTime objects, however, my conversions to alternative formats would convert the DateTimes into string objects. This resulted in each plot point receiving an individual tick to mark it's x-axis position. This was not an issue with the "Month.Day" formatting, as they could be dealt with as decimals, and not require individual ticks.

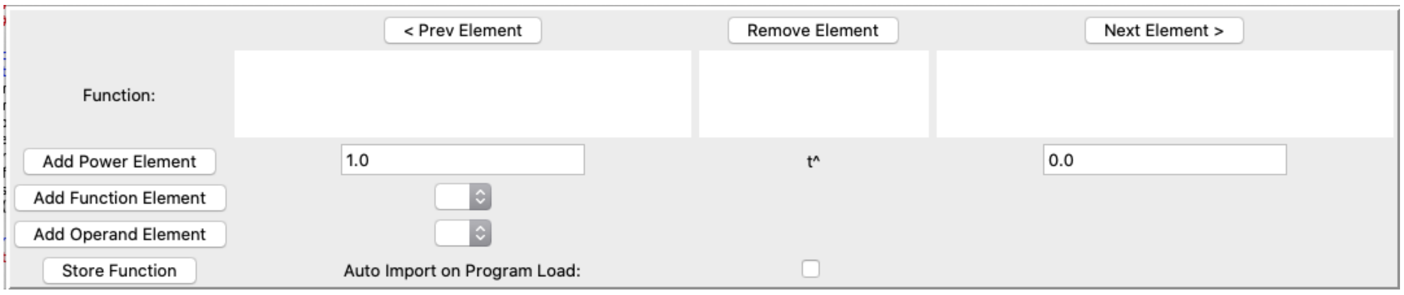
To fix this, I would introduce a new argument to set a maximum number of ticks – which Matplotlib automatically configures it's tick labels around. The graph of windspeed here is limited in the y-axis to 15 ticks. Whilst this does somewhat limit readability of the graphs, it still provides a perfectly suitable presentation. Furthermore, having greater than 15 ticks often results in a difficult to read axis – negating what assistance more axis information may have provided.



DateTime Management and Graphing – Testing and Evaluation

Function	Score	Score Determinants
Read DateTime Objects	10	
Date Management	10	Working perfectly.
Time Management	5	For values of time matching "00:00:00", the program returns nothing, as shown in the bottom image. However, it does not seem to have a negative effect.
Preview changes	10	Both old and New Values are correctly displayed. Although titles do overlap, but not a large issue
Applying changes	9	Irreversible, but the user does permit the action themselves
Graphing with new DateTimes	7	It works, however I would have preferred to find a method through which actual Dates can be displayed, whilst also calculating lines of best fit.

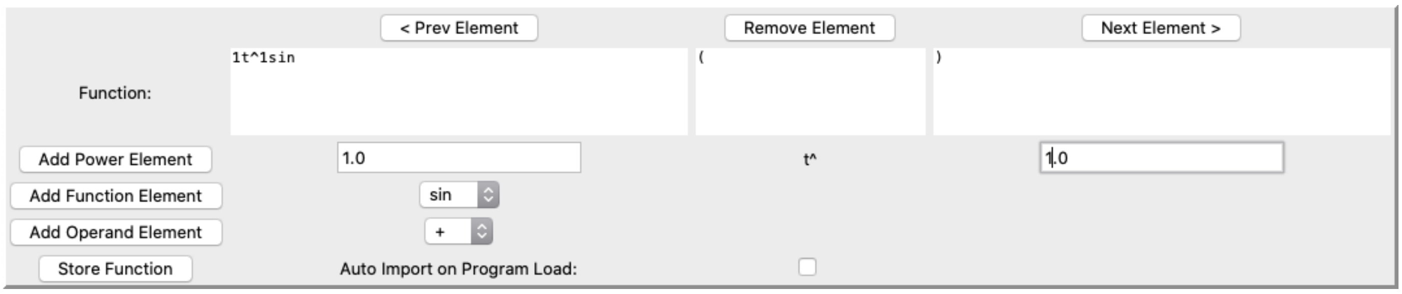
Function Constructer – Development



1.0

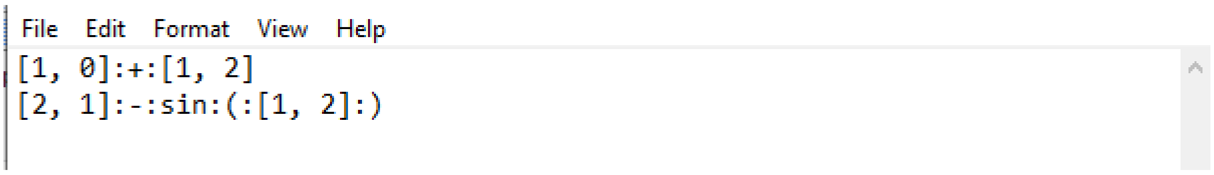
0.0

With the function constructer, I would seek to allow for the user to produce higher-level functions than the previous polynomial constructor. This section would largely be successful, providing the ability to employ trigonometric functions, and their reciprocals, alongside exponentials and non-integer powers of t. the program was correctly obtaining user inputs, and storing them inside a list.



The user was capable of moving between positions in the list, using the buttons as the top of the screen. The current element would be shown in the central box, and elements before and after being displayed in the Left and Right side boxes respectively. When adding an element to the function, it would be added to the right of the element being displayed in the central box. For removing an element, the element being displayed in the central box would be removed.

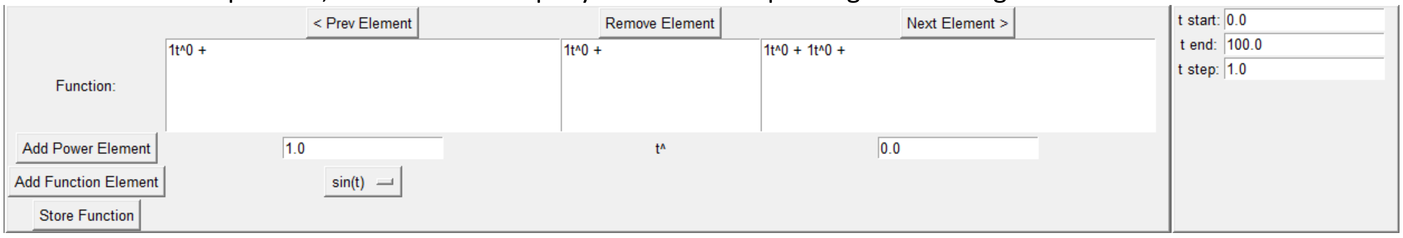
Another addition would be permitting the user to store the function – as unparsed element information – in a txt file, which was read upon loading the program, the same as the excel data. Below I have provided an image of the formatting in which each element was stored. **:** would be used as a separator such that it would not be confused with alternative elements – like the , used to separate coefficient and power elements of kt^n elements.



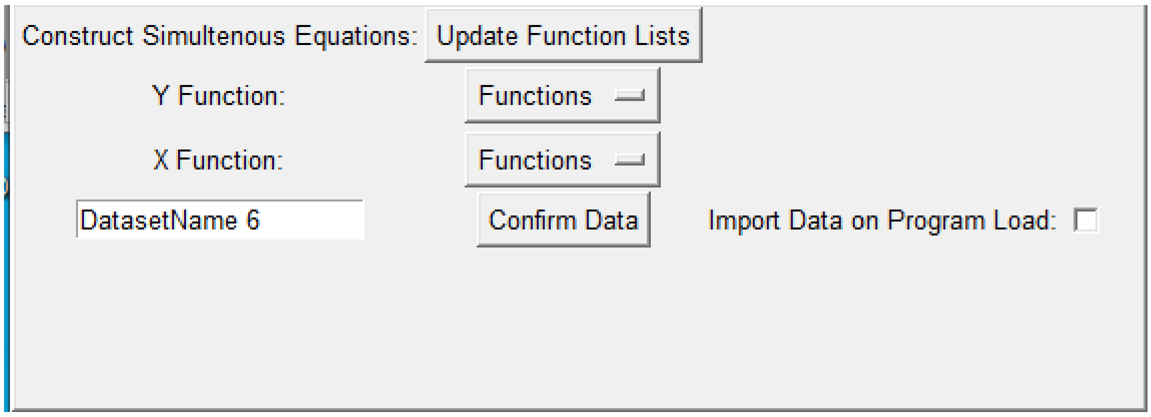
```
[1, 0]:+:[1, 2]
[2, 1]:-:sin:(:[1, 2]:)
```

The functions would only receive once crucial flaw. I had permitted the function parser to be introduced late in prototype 3 as I expected the process to be simple and efficient. I did not anticipate that the vast variety of user inputs would result in numerous potential situations needing to be considered. Furthermore, it would be wholly reliant upon perfect user input, employing mathematical rules for dealing with elements such as BIDMAS, and detecting brackets surrounding subfunctions within the greater function.

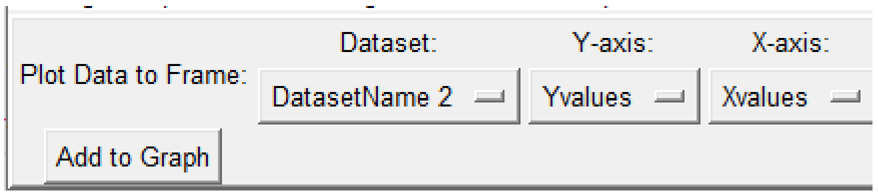
As a result of this, I would be forced to largely scrap the new function parser due to difficulties in parsing. Instead, I would return to the previous polynomial constructor and aim to improve it instead. I would employ the function elements I had explored in prototype 3 – such as trigonometric functions, exponents and logarithms, but would return to just series of operations. The function storage would also be made simpler, such that the values computed from the function would be stored, rather than the function itself. This permitted the values to be stored alongside data in the AutoImport file, and was more rapidly accessible for plotting than having to recalculate values each time.



When a function is constructed, it is stored locally. This function can then be applied as either an X or Y function, with the user able to create simulteneous equations through having X and Y values determined through different equations, or setting X = to a function of just t, effectively producing a direct equation of y=F(x).



I would have difficulties in automatically updating the list, and so would employ a simple list refresher button at the top: “Update Function Lists”. When pressed, each function stored in the internal function list would be applied and shown in the Option Menus titled “Functions”. From there, upon selecting “Confirm Data”, the program stores the selected X and Y functions in a pandas database, with “Xvalues” and “Yvalues” as their respective headings. This allows for it to be easily stored within an Excel file, and being read as a dataframe keeps it similar to the rest of the program. Below is an image of how the user can select the data to graph, showing that the user has stored the data under it’s default namesake.



As a small explanation of DefaultName, it will change the number to match the length of the stored datasets list, in an attempt to reduce instances of data being stored under the same name. however, since data being stored under the same is only a matter of the user selecting the right dataset they want, and names not causing issues in graphing, I did not take many measures to protect against this. It is up to the user to provide a suitable and unique identifier for their data.

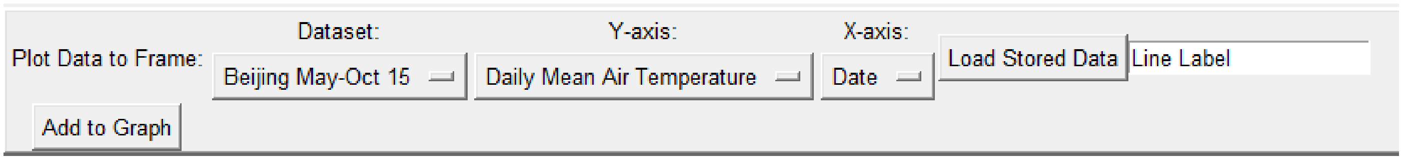
Function Constructer – Testing and Evaluation

Function	Working	Score	Score Afflicters
Add Function	Y	7	Far more limited than I would have liked to initially introduce
Add kt^n Element	Y	10	It is obvious as to how the element is structured, and what values are coefficients or orders
Remove Element	Y	10	It is obvious what will be removed, and works perfectly
Storing Functions	Y	8	Functions are only stored in the program whilst it is running
Get F Values	Y	10	Correctly reads each element and calculates the functions value for a given (t) value.
Saving Values	Y	10	
Displaying Function	Y	7	Displays each element correctly, and in the correct positions, however, but could do with a better indicator of where elements will be added at
Displaying Values	Y	10	All values displayed correctly

There is little to note for user feedback or errors in this section, as it largely built upon the initial polynomial constructor in prototype 2.

Graphing Data – Development

The graphing section would need multiple improvements from the previous version, due to the label ids being largely incorrect and thus unusable. Editing plots was liable to great issue, and the user had to split between scatter, line and bar plots.
For the new data grapher, the first change I would make was to remove the options to select line, scatter and bar graphs entirely. Instead, the program is entirely build around the standard line construction functions. This meant that bar charts would be scrapped entirely, however, they are largely only useful for descriptive data, which was not the target of this project anyway. For creating scatter diagrams, the user would instead remove the connecting line of the line graph to display only the point markers of data values. This also permitted for the user to remove scatter values in a far easier manner, as they were all tied to a single plot line, rather than multiple individually plotted values.

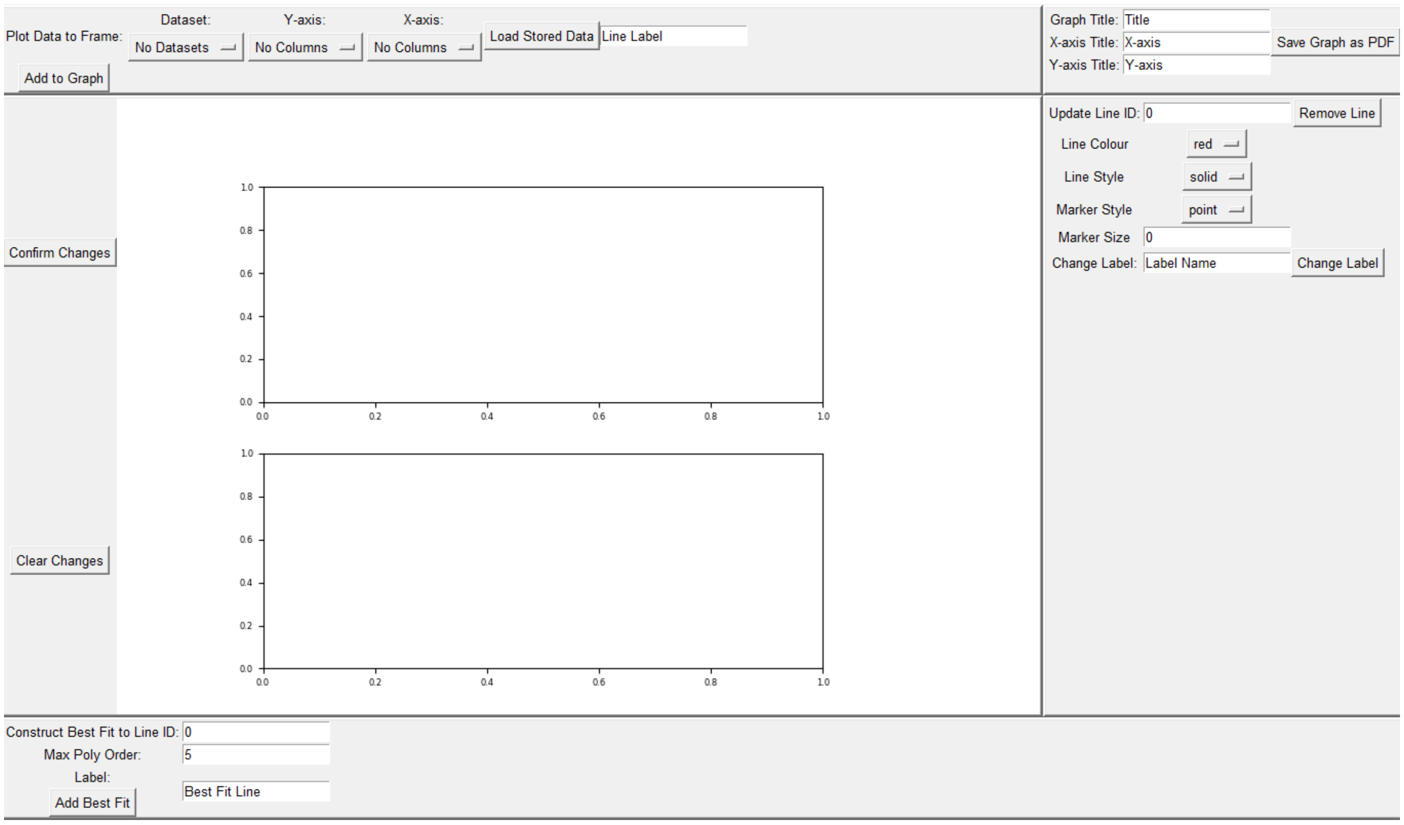


The remainder of the initial data selection is the same, with the user choosing a dataset by name, and then applying particular values to either the x or y axis. Whilst coding this design, I also ensured a refresh button was present – like was with the function creation screen. Thus, when the user introduces a new data set, they can reload the optionmenus and it will be present.

The main features of the line editor remain the same as the previous prototype, only with an added Label Name changer, and “none” line style. They are shown on the right-hand side of the graph. The more prominent change from the previous prototype is that alterations to lines are implemented immediately upon the bottom graph – the working / unconfirmed graph. The only exception to this being the Remove Line feature, which requires button activation, and the Change Label Name feature, which may otherwise immediately change names of Labels the moment an id is entered, which is likely not desirable for the user.

The Graph Figure is also embedded and split into two graphs like the previous version. However, there is less liability for errors when attempting to clear and confirm changes. The bottom graph I continued as the temporary or in-progress graph for this prototype, and the top graph as the confirmed and finalised graph, of which pdf images can be stored.

The polynomial constructer – being a part of the more expensive function constructor on a different screen – is also removed from the bottom of the graph screen. This leaves only the Best Fit creator, which functions the same as the previous prototype, but with an editable label now implemented.



Dataset: No Datasets
Y-axis: No Columns
X-axis: No Columns
Line Label
Graph Title: Title
X-axis Title: X-axis
Y-axis Title: Y-axis
Update Line ID: 0
Line Colour: red
Line Style: solid
Marker Style: point
Marker Size: 0
Change Label: Label Name
Construct Best Fit to Line ID: 0
Max Poly Order: 5
Label: Best Fit Line

Line Class Development

To assist with graphing, I would significantly improve my prior system of storing line information as lists through designing each line as a class object. A line would require the same variables for the x and y values, a label, reference id, housing Axis and Frame, marker style settings and line style settings to be stored. However, I had two graphs to deal with, and a line which could be changed between these graphs. In order to have a single Line object be responsible for its Working and Confirmed value, I would create two internal lists that the object would manage.

Previously, this job was split into two lists stored within the Graphing screen itself:

```
self.storedGraphInfo = []
self.tempStoredGraphInfo = []
```

Which would become

```
self.workingArgs = [self.xvals, self.yvals, root, rootfig, name, lineid, linecol, linestyle, markerstyle, int(markersize)]
self.confirmedArgs = [] #rootaxis, rootfig, name, id, linecol, linestyle, markerstyle, markercol
```

The first subroutine to add into the class was a method through which the Line would graph itself. As the line would house two sets of values, I would make the function dependant on being given a set of values from which it can graph. Thus, the correct values are always used, with only a single graphing function.
This would require that the order in which values were stored in the workingArgs or confirmedArgs Lists was consistent, however, that was not an issue.

```
    def graph(self, vals):
        #xvals, yvals, root, rootfig, name, lineid, linecol, linestyle, markerstyle, markersize
        vals[2].plot(vals[0], vals[1],
                     label=f'{vals[4]}\nid:{vals[5]}',
                     color=vals[6],
                     marker=vals[8],
                     markersize=vals[9],
                     linestyle=vals[7])
```

the most common items which would be updated in the Line object would be the 4 automatically changed values for line colour, style, marker style and size. Each time any of these values is altered, the same tracer function is called, which will call for the Line object's update function. here, since changes are being applied, the only necessary list is workingArgs, and the values within it are replaced with the updated values obtained from the user modification section. Altering the Label Name would be called through a button press, and separate to the automatic changes. As a result of this, I would directly access the Line's Label value to replace it.

the final necessary segment for the Line object was a method through which lines could be temporarily hidden – for

```
def update(self, linecol, linestyle, markerstyle, markersize):
    self.workingArgs = [self.workingArgs[0], self.workingArgs[1], self.workingArgs[2], self.workingArgs[3],
                        self.workingArgs[4], self.workingArgs[5],
                        linecol, linestyle, markerstyle, int(markersize)]
```

the user to view the graph without the line – and for those hidden lines to be destroyed once the user confirms the changes. However, organising the removed lines would largely take place outside of the Line Object itself, and instead inside of the Graph Update functions. As a result, the final Line Object would be constructed as shown below:

```
class Line():
    def __init__(self, xvals, yvals, root, rootfig, name, lineid, linecol, linestyle, markerstyle, markersize):
        p.linesList.append(self)
        self.xvals = xvals
        self.yvals = yvals
        self.removed = False
        if xvals == 'axis':
            for i in range(len(yvals)):
                self.xvals.append(i)
        if yvals == 'axis':
            for i in range(len(xaxis)):
                self.xvals.append(i)
        self.workingArgs = [self.xvals, self.yvals, root, rootfig, name, lineid, linecol, linestyle, markerstyle, int(markersize)]
        self.confirmedArgs = [] #rootaxis, rootfig, name, id, linecol, linestyle, markerstyle, markercol
        self.markersize = int(markersize)

    def graph(self, vals):
        #xvals, yvals, root, rootfig, name, lineid, linecol, linestyle, markerstyle, markersize
        vals[2].plot(vals[0], vals[1],
                     label=f'{vals[4]}\nid:{vals[5]}',
                     color=vals[6],
                     marker=vals[8],
                     markersize=vals[9],
                     linestyle=vals[7])

    def update(self, linecol, linestyle, markerstyle, markersize):
        self.workingArgs = [self.workingArgs[0], self.workingArgs[1], self.workingArgs[2], self.workingArgs[3],
                            self.workingArgs[4], self.workingArgs[5],
                            linecol, linestyle, markerstyle, int(markersize)]

    def remove(self):
        self.removed = True

    def delete(self):
        del self
```

Graphing Data – Testing and Evaluation

Function	Working	Score	Score Afflicters
Add to / Remove from Graph	Y	10	All working perfectly. Improved in quality of use over the previous prototype. Feels more fluid.
Confirm Changes	Y	10	
Clear Changes	Y	9	Sometimes takes 2 button presses, and rarely will cause issues that stop the graph from continuing to function. necessary to sort in the final design.
Line Editing	Y	9	Potentially add some other factors, like line width. But significant improvements with line id and label changing
Graph type	Y	6	Bit limited with only line and scatter, could do with alternative graphs like histograms. Also try non-linear axis scales?

The graphing data section was very messy in the previous prototype. In particular, the broken line id system resulted in numerous features becoming near impossible to feasibly work with. With this functioning as intended and in an easily understandable manner in the current program, the users can apply their alterations to lines and produce best fits easily. Furthermore, the ability to change labels was particularly appreciated, as it allowed the test users to rapidly construct lines and then edit afterwards.

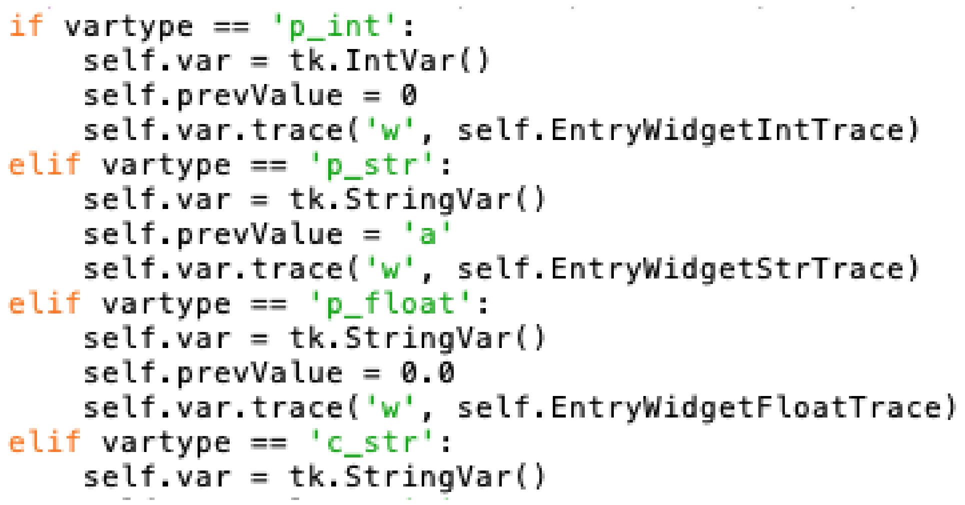
I would also follow up on the prior requests for more instantaneous changes, with alterations to line style values being applied and displayed immediately after being selected – with the exception being the label name. for the most part, this section has met the stakeholders' previous demands.

Input Validation – Development

Another small element I would introduce for this prototype would be input validation. I aimed to utilise the Entry Widget's ability to utilise a StringVar or IntVar variable for storing text within the Widget as a means to trace changes to the stored data and validate entries as they are typed.

I already had the means to define variables as class objects, and so I would aim to produce a callable function specific to the Entry Widget Class, through utilising the variables stored within the object, alongside obtaining, and executing an optional command should a value change.

The first question was how the class would determine what form of variable was required. For Entry widgets where I desired only integers, only floats, only strings or were to permit anything, different variables would be required. For this, I would create my own class tag named "vartype". With this, the first non-common (that is shared with other widgets) argument passed into an Entry Widget Class will be used to determine the storage variable.



```
if vartype == 'p_int':
    self.var = tk.IntVar()
    self.prevValue = 0
    self.var.trace('w', self.EntryWidgetIntTrace)
elif vartype == 'p_str':
    self.var = tk.StringVar()
    self.prevValue = 'a'
    self.var.trace('w', self.EntryWidgetStrTrace)
elif vartype == 'p_float':
    self.var = tk.StringVar()
    self.prevValue = 0.0
    self.var.trace('w', self.EntryWidgetFloatTrace)
elif vartype == 'c_str':
    self.var = tk.StringVar()
```

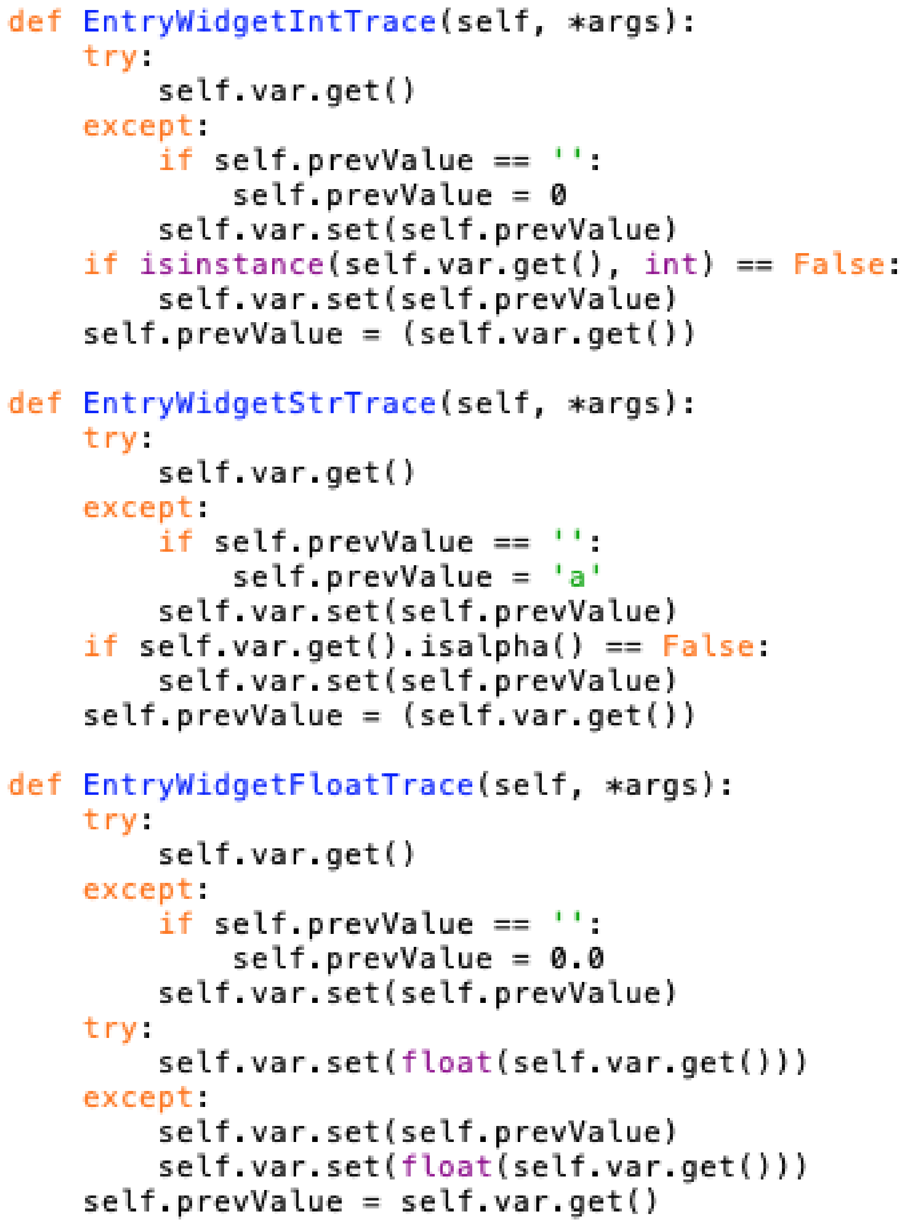
self.var.trace is the code used to initiate a tracer on the object's specific storage variable. Whenever this value is altered, the command (right-side inside the brackets) is called. Since a check would not need to be performed for the "c_str" type, no trace is used to detect variable changes.

The next task was to provide the commands which would ensure that the text stored within the Entry widget was what was supposed to be stored, and if not would be replaced with the previous (acceptable) value. Furthermore, should the values be empty, then they should be replaced with a value to avoid errors. Whilst I recognised that this may cause some mild issues when attempting to insert data and being unaware that the final character cannot be deleted, it was effective in preventing crashes or errors arising due to a lack of values.

The easiest to introduce checks for was the integer type, as an "isinstance" could be used to check the type of the stored data. Should it return false, it indicated that the stored value was not an integer and could be replaced with the previous value in the widget. If the function returns true, then the stored value passes the check, and the previous value is updated to be the current value – in anticipation of future changes.

A similar method could be used for detecting pure strings, however, an "isinstance" itself would fail as strings can contain numbers without raising errors. Instead, the function "isalpha" would be used, with true indicating that the stored value is entirely composed of alphabetical characters. The previous value and empty value work the same as int checking.

For floats, this too was a similar method. But I instead used try and except clauses to attempt to either set the value as the float of itself, or – in the case of an error being raised, thus indicating the stored value to not be a float – set it back to the previous value. The resultant 3 functions were:



```
def EntryWidgetIntTrace(self, *args):
    try:
        self.var.get()
    except:
        if self.prevValue == '':
            self.prevValue = 0
        self.var.set(self.prevValue)
    if isinstance(self.var.get(), int) == False:
        self.var.set(self.prevValue)
    self.prevValue = (self.var.get())

def EntryWidgetStrTrace(self, *args):
    try:
        self.var.get()
    except:
        if self.prevValue == '':
            self.prevValue = 'a'
        self.var.set(self.prevValue)
    if self.var.get().isalpha() == False:
        self.var.set(self.prevValue)
    self.prevValue = (self.var.get())

def EntryWidgetFloatTrace(self, *args):
    try:
        self.var.get()
    except:
        if self.prevValue == '':
            self.prevValue = 0.0
        self.var.set(self.prevValue)
    try:
        self.var.set(float(self.var.get()))
    except:
        self.var.set(self.prevValue)
        self.var.set(float(self.var.get()))
    self.prevValue = self.var.get()
```

Input Validation – Testing and Evaluation

Function	Working	Score	Score Afflicters	Feedback
Check for Int	Y	10		
Check for String	Y	7	Limited in use. Often better going without a check. Could be useful to implement max str length	Make it check for non-suitable characters for saving file names.
Check for Float	Y	10		
No Check	Y	8	No check also meant no protection from empty values	
Ensure no Empty Value	N	5	Prevented some crashes from having no value, but due to multiple tracers observing a single var, they could track and use an empty value before it gets replaced.	Made entering values more difficult to deal with for test users. Very unnecessary effort.

Conclusion

Prototype 3 has introduced – or attempted to – the remaining necessary features of normal data comparisons, and function construction. Whilst there are still goals left unmet from my initial ideas, alongside test user feedback that is still not employed within the program, it will be difficult to introduce those additional features into the final design within a reasonable timeframe. Many of these ideas are small pieces, of which the omission is not largely detrimental to the product. However, they are worth considering as they do come from my stakeholders.

Instead, I aim to iron out any significant errors which threaten the program through crashing, or are detrimental to user experience. This includes attempting to solve screens not displaying widgets immediately – albeit I still believe this may be a device-specific issue rather than a program issue – and ensuring 100% reliability in the importing data and graphing sections.

Importing Data – Video Demonstrations

Function	Video Demonstration	Description
Import Excel File	https://youtu.be/SMKHSHfNjXU	Displaying how the file finder is used, and then selecting an Excel File from the current folder. Also demonstrates how the program parses the excel data into a text box on the screen
Import CSV	https://youtu.be/wVm_Lcx8LEs	Same as above, however, utilising a csv file.
Normal Comparison + Anomalies	https://youtu.be/yAmLzfMbPno	Displaying how a normal graph and histogram are overlaid to produce a data comparison, and then how the user can modify potential anomalies.
Date Time Conversions	https://youtu.be/_b-5SWNo4YQ	Altering stored Dates and Times into the user's preferred formatting style.
Automatic Import File	https://youtu.be/kP070ZzrjOE	Starting from the initial loading of the program, shows that data is imported automatically and ready to graph.
Save in Program	https://youtu.be/D5ziYOTl8nQ	Shows the program's internal data storage updating to include newly imported data values

Importing Data – Goal Completion

Function	Description	Complete?
Import Excel Data	Can the program successfully import and read from an Excel File?	Y
Import CSV Data	Can the program successfully import and read from a CSV File?	Y
Import URL Data	Can the program successfully locate a URL Link and read the data at its destination?	N
Normal Distribution Anomalies	Has a Normal Distribution Model been utilised to provide estimations of values?	Y
Direct Editing of Values	Can the user directly access and edit stored data?	N
Specific Columns, Headers, Footers, etc	Can the user specify data to import through column selections, headers, footers or otherwise?	Y
Saving to Program Storage	Can the program successfully store data into a form of storage?	Y
Saving to External File	Can the program successfully store data into an external file?	Y
Edit Externally Saved Data	Can the program edit and manage externally stored data?	N

For importing data, the most prominent goals have all been successfully attained. Of those which were not, importing URL data would be as I believed it to be unnecessary for this program. I anticipated that most data stored online can be downloaded to devices, thus permitting the reading of the data through the file finder methods. Furthermore, URL links may not lead directly to a readable dataset – the dataset instead being a part of a larger webpage, hence leading to numerous errors when I attempt to provide a general parsing method. Meanwhile, the reading of CSV and Excel files is far more guaranteed in the expected results when the function is called.

The direct editing of values was a feature that I did not want to omit, as it sounds like a basic expected feature. However, providing a way for the user to access and edit values stored within the data frame would be too difficult. Whilst the user can interact easily with Text widgets, interpreting data written into these widgets as a data frame would likely result in numerous errors due to formatting issues or user input error. Alternatively, a single text widget could be used to store a single value for editing, however, this would require a vast amount of screen space. The import screen was already utilising much of its screen space, and making the raw data parsing section larger would make an ergonomic design difficult.
The remaining goal that was not met was the ability to edit the external storage file. This was a matter of running out of time to add more features, resulting in it being omitted. If the user desires, they can still access the external

storage file directly and remove data in there. However, being a part of the program itself would provide for a much more streamlined approach.

Graphing Data – Video Demonstrations

Function	Video Demonstration	Description
Adding to Graph	https://youtu.be/BvXXwEgjZV4	Adding stored data values to graph, and the default options applied.
Removing from Graph	https://youtu.be/hmiFyuWHeOo	Removing already plotted lines from the graph
Line Formatting	https://youtu.be/y_py2rpMKNo	Updating plotted lines to have different formatting and styles.
Confirming / Clearing Changes	https://youtu.be/I74R6_yut7M	How confirming changes will effectively save all currently in-progress work as a backup, and clearing changes will undo all changes made after the last backup was made.
Best Fit	https://youtu.be/5wf88BOlhTA	Produce a line of best fit for a particular set of data in the graph. Shows how the user can alter the max polynomial function, and the effects it has upon the line.
Title Application	https://youtu.be/kL0cN0xUv8Y	How titles automatically update once the user edits them.
Saving PDF Image	https://youtu.be/VTRhcV0q35c	Shows the newly created PDF image file being stored in the program folder, with the Graph Title as it's name.

Graphing Data – Goal Completion

Function	Description	Complete?
Add to Graph	Produce a graph of provided x and y values	Y
Remove from Graph	Remove a set of x and y values from the graph	Y
Preview Graph before Confirming	User can check over the newly constructed graph before finalising their changes through the use of an additional preview graph	Y
Graph Axis Titles	Suitable titles can be entered and applied to the graph by the user	Y
Rolling Averages	Rolling Averages are able to be applied to an axis, to obtain and then graph mean values of data for pattern identification.	N
Alternative Graphs	User can select between line, scatter, bar or histograms, or alternative potential graphs – rather than just a single formatting type.	Y
Alternative Grid Styles	User can select alternative scales for the graph axes, such as logarithmic scales.	N
Graphing Change in Data Values	+ or - Change between the values of X and Y can be graphed.	N
Alter Line Values	Change the appearance of graphed lines	Y
Alter Marker Values	Change the appearance of graphed markers	Y
Save Graph Image	Save a graph as a pdf vector image.	Y

The majority of unadded features involved in graphing data would be caused by a requirement of managing or manipulating data outside of the importing section. This refers to the Rolling Averages, and Graphing Change in Data Values. Direct approaches to graph the change in Y and Z over X would be removed as I believed it to be an unnecessary approach to graphing the changes of data over time. The user could always graph two lines to compare data, the only use this feature would provide is directly displaying the change, which is already rather evident.

Furthermore, with the addition of lines of best fit to represent sets of data, they act even better as comparison lines than just the raw data in a line or scatter graph. As this feature was added, it somewhat took over the place that graphing change held in the program, resulting in the feature being omitted.

With regards to rolling averages, this remains as a potentially useful feature to have in the program, and would likely be part of my next features to add into the program, alongside alternative graph scales – such as logarithmic scales – which I did not have time to incorporate into the program.

Function Creation – Video Demonstrations

Function	Video Demonstration	Description
Adding / Removing Elements	https://youtu.be/zbOMt9nGJDg	Shows the user adding and removing power and function elements. Also shows the user moving between values and how they are displayed.
Creating and Storing Values	https://youtu.be/q7lkqDENY78	How values are created by applying a function to X and Y, and utilising a user-input range system for t, whereby a starting value, final value and step can be entered.
Graphing Functions	https://youtu.be/N2by3UUE8Ic	How stored function data can be added to a graph. The same methods available to imported data lines can be applied to lines drawn from function data.

Function Creation – Goal Completion

Function	Description	Complete?
Polynomial Series Construction		Y
Trigonometric Functions		Y
Exponents and Logarithms		Y
Non-Integer Powers		Y
Non-Series Functions (multiplication and division)		N
Display of raw data from function		Y
Graph display of Function		N
Provide Range for Function/s	User can enter values for start, stop and step values, for which a function will return a value for.	Y
Create Direct Equation	Create a direct equation in the form y=F(x)	Y
Create Simultaneous Equations	Create simultaneous equations in forms y=F(t), x = F(t)	Y
Create 1st Order Differentials	Create 1st Order Differential Equations for modelling.	N
Create 2nd Order Differentials	Create 2nd Order Differential Equations for modelling.	N
Solve Direct / Simultaneous Functions	Parse and find values in a given range	Y
Estimate Differentials using Euler's Formulae	Utilise Euler's formulae to create estimations for values in differential equations. Note them as estimations rather than true values.	N

The failure to create a suitable function parser would have a significant impact upon being able to produce Differential equations. These equations are constructed in the form of:
$\frac{dy}{dx} + F(x)y = G(x)$ for First Order, and for Second Order: $\frac{d2y}{dx2} + F(x)\frac{dy}{dx} + G(X)y = H(x)$

Without being able to construct non-series functions, or multiple variable functions, the differentials cannot be constructed. This is a significant limitation of the program, as differential equations are extensively utilised for modelling rates of change.

Another feature missing from the function construction screen was an additional graph to display the raw data values in a more understandable format. this feature was omitted as users could already graph the data – just in a different screen – however, they would not be able to check for and remove potential errors, which are made much more obvious with unexpected graph plots. Introducing this graph would have been a strong quality-of-use feature for the program.

Non-series functions were not implemented due to extensive difficulties in creating a parser for entered functions. As a result of this, all functions which can be constructed are limited to being addition series, and hence the user is severely limited in constructable graphs or models.

Conclusion

Overall, the program managed to meet the primary aims of the stakeholders through providing an easier to produce and more manageable graph producer. The stakeholders did anticipate a few more features being present within the final program, but with regards to what was delivered, are wholly satisfied. They recognise that the program was severely impacted and thus limited by a bad judgement on a critical feature, but being data scientists, there is still large swathes of data open to analysis beyond producing their own.

What I would aim to do Next

I believe the most crucial element of the program to get functioning is the function parser. My failure to introduce a sufficient version of this resulted in the forced omission of differential equations and more complex simultaneous equations. When I began the project, I did not anticipate that the function parser would take such a lengthy amount of time to program, understand and complete. My experience in mathematics with these functions gave a false sense as to how easily or rapidly I could parse and deal with them. As a result, this feature was left too late into the project to implement in a meaningful manner.

However, if it were to continue, producing a sufficient parser would be the next step. I could continue utilising the function constructor as this worked as intended for creating functions – albeit some various issues and errors needed fixing before it worked perfectly. From there, I would have to break down my own knowledge of dealing with functions and equations down to the fundamental steps. Locating brackets, powers, functions etc. the issues with parsing functions derived from that it was not so simple as reading from left to right, but often originated from multiple positions and outwards.

Beyond this, the next step would be to provide further improvements to the graphing capabilities of the program. intended to be the main feature, it should utilise the requests of the stakeholders for additional graphing types. Histograms are used elsewhere in the program and were reportedly somewhat of a false indicator of the potential capabilities of the graphing feature. Whilst not directly suggested by the stakeholders, being able to produce Normal Distributions of other data sets into the main graph would likely be appreciated.

Third, another area to improve would be the level to which users could manage and edit the data being imported. The ability to select specific columns was removed after the program kept crashing, and I was unable to find the cause – deciding that users could select particular columns to graph anyway, and so should not provide much impact. Whilst still not necessary to bring back, it would likely be appreciated for removing completely useless columns. Furthermore, the import screen should be wholly redesigned to fit in a new method whereby the user can edit specific stored data values. Currently, the data frame is pasted into a single text widget, but with the use of 1 text widget to store 1 value, and then employing an array of them into a grid format, it may be possible to re-read each of the values to reconstruct a data frame from the widgets. This would require significantly more space than a single text widget, however, it may be possible if the scrollbars could be used to scroll through Text widgets rather than the text itself.

Improvements to anomaly detection could also be made by permitting ranges to be defined for the normal functions. Thus, anomalies are relative to a smaller region, and the distribution can account for changing data patterns

Less necessary features would largely be composed of the more Quality-of-Life elements, such as managing the data stored In the Excel file used to provide the automatic data imports upon loading the program. Furthermore, I would aim to improve the accessibility features through introducing preset options for dark modes to reduce the harshness of the default bright UI, and make selecting desired fonts or font styles easier.

Widgets Code

```
import tkinter as tk
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

class Widget():
   def __init__(self, root, rpos, cpos, sticky, rspan, cspan, *args):
      self.root = root
      self.rpos = rpos
      self.cpos = cpos
      self.sticky = sticky
      self.rspan = rspan
      self.cspan = cspan

   def usefont(self, ff, fc, fs, bg):
      try:
         self.body.config(font=(ff, fs), fg=fc)
      except:
         pass

      self.body.config(bg=bg)



   def load(self):
      self.body.grid(row=self.rpos,
                  column=self.cpos,
                  sticky=self.sticky,
                  rowspan=self.rspan,
                  columnspan=self.cspan)

   def unload(self):
      self.body.grid_forget()




class FrameWidget(Widget):
   def __init__(self, root, rpos, cpos, sticky, rspan, cspan):
      super().__init__(root, rpos, cpos, sticky, rspan, cspan)
      self.body = tk.Frame(root, relief='raised',borderwidth=3)

class LabelWidget(Widget):
   def __init__(self, root, rpos, cpos, sticky, rspan, cspan, text='NoTagProvided'):
      super().__init__(root, rpos, cpos, sticky, rspan, cspan)
      self.body = tk.Label(root, text=text)

class ButtonWidget(Widget):
   def __init__(self, root, rpos, cpos, sticky, rspan, cspan, text='', command=None):
      super().__init__(root, rpos, cpos, sticky, rspan, cspan)
      self.body = tk.Button(root, text=text, command=command)
```

```
class TextWidget(Widget):
   def __init__(self, root, rpos, cpos, sticky, rspan, cspan, text, width=10, height=2, state='normal', wrap='none'):
      super().__init__(root, rpos, cpos, sticky, rspan, cspan)
      self.body = tk.Text(root, state=state, width=width, height=height, wrap=wrap)
      self.body.insert('1.0', text)
      self.body.config(state=state)
      self.state = state

   def add(self, text):
      if self.state == 'disabled':
         self.body.config(state='normal')
         self.body.insert('1.0', text)
         self.body.config(state='disabled')
      else:
         self.body.insert('1.0', text)

   def replace(self, text):
      if self.state == 'disabled':
         self.body.config(state='normal')
         self.body.delete('1.0', 'end')
         self.body.insert('1.0', text)
         self.body.config(state='disabled')
      else:
         self.body.delete('1.0', 'end')
         self.body.insert('1.0', text)

class EntryWidget(Widget):
   def __init__(self, root, rpos, cpos, sticky, rspan, cspan, text, vartype='c_str', tr_cmd=None):
      super().__init__(root, rpos, cpos, sticky, rspan, cspan)
      if vartype == 'p_int':
         self.var = tk.IntVar()
         self.prevValue = 0
         self.var.trace('w', self.EntryWidgetIntTrace)
      elif vartype == 'p_str':
         self.var = tk.StringVar()
         self.prevValue = 'a'
         self.var.trace('w', self.EntryWidgetStrTrace)
      elif vartype == 'p_float':
         self.var = tk.StringVar()
         self.prevValue = 0.0
         self.var.trace('w', self.EntryWidgetFloatTrace)
      elif vartype == 'c_str':
         self.var = tk.StringVar()
         self.prevValue = 'a'
      self.body = tk.Entry(root, textvariable=self.var)
      self.body.insert(0, text)
      if tr_cmd != None:
         self.var.trace('w', tr_cmd)
      self.var.set(text)

   def EntryWidgetIntTrace(self, *args):
        try:
            self.var.get()
        except:
            if self.prevValue == '':
                self.prevValue = 0
            self.var.set(self.prevValue)
        if isinstance(self.var.get(), int) == False:
            self.var.set(self.prevValue)
        self.prevValue = (self.var.get())

    def EntryWidgetStrTrace(self, *args):
        try:
            self.var.get()
        except:
            if self.prevValue == '':
                self.prevValue = 'a'
            self.var.set(self.prevValue)
        if self.var.get().isalpha() == False:
            self.var.set(self.prevValue)
        self.prevValue = (self.var.get())

    def EntryWidgetFloatTrace(self, *args):
        try:
            self.var.get()
        except:
            if self.prevValue == '':
                self.prevValue = 0.0
            self.var.set(self.prevValue)
        try:
            self.var.set(float(self.var.get()))
        except:
            self.var.set(self.prevValue)
            self.var.set(float(self.var.get()))
        self.prevValue = self.var.get()



class OptionMenuWidget(Widget):
    def __init__(self, root, rpos, cpos, sticky, rspan, cspan, options=['No Options'], tr_cmd=None):
        super().__init__(root, rpos, cpos, sticky, rspan, cspan)
        self.var = tk.StringVar()
        if tr_cmd != None:
            self.var.trace('w', tr_cmd)
        self.body = tk.OptionMenu(root, self.var, *options)
        self.var.set(options[0])

    def changeOptions(self, options=['No Options']):
        self.body['menu'].delete(0, 'end')
        for item in options:
            self.body['menu'].add_command(label=item, command=tk._setit(self.var, item))
        self.var.set(options[0])
        self.load()

class CheckbuttonWidget(Widget):
    def __init__(self, root, rpos, cpos, sticky, rspan, cspan, tr_cmd=None):
        super().__init__(root, rpos, cpos, sticky, rspan, cspan)
        self.var = tk.IntVar()
        if tr_cmd != None:
            self.var.trace('w', tr_cmd)
        self.body = tk.Checkbutton(root, variable=self.var)

class ScrollbarWidget(Widget):
    def __init__(self, root, rpos, cpos, sticky, rspan, cspan, orient='vertical', target=None):
        super().__init__(root, rpos, cpos, sticky, rspan, cspan)
        if orient == 'vertical':
            self.body = tk.Scrollbar(root, orient=orient, command=target.yview)
            target.config(yscrollcommand=self.body.set)
        elif orient == 'horizontal':
            self.body = tk.Scrollbar(root, orient=orient, command=target.xview)
            target.config(xscrollcommand=self.body.set)

class GraphFigWidget(Widget):
    def __init__(self, root, rpos, cpos, sticky, rspan, cspan, figure):
        super().__init__(root, rpos, cpos, sticky, rspan, cspan)
        self.body = FigureCanvasTkAgg(figure, root).get_tk_widget()
```

Main Program Code

```
import tkinter as tk                    #user interface constructor
import tkinter.font as tkFont               #for ease of use
```

```
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

import matplotlib.pyplot as plt                  #calculations / graphing and
import pandas as pd                              #dealing with datasets
pd.set_option('display.max_rows', None,     #visual enhancements for user
           'display.max_columns', None,  #quality of use. prevents program
           'display.width', None)       #condensing lines

from scipy.optimize import curve_fit
from scipy import stats
import datetime as dt
import numpy as np

from tkinter.filedialog import askopenfile        #only used for importing files
import os.path
import platform

from Class_Tk_Widgets_Cur_3 import *            #My widgets classes

##########################################################################################
#############################################################
##########################################################################################
#############################################################
##########################################################################################
#############################################################

class Program():
   def __init__(self):
      #tk interface setup
      self.window = self.window()
      self.menubar = self.menubar()

      #tk management vars
      self.currentScreen = ''
      self.fontOptions = list(tkFont.families())
      self.fontOptions.insert(0, 'TkDefaultFont')
      self.fontColours = ['black', 'blue', 'red', 'green', 'white']
      self.bgOptions = ['white', '#CFCFCF', '#A9A9A9', 'grey', '#404040', '#5A5A5A', 'black']
      self.defaultStyles = ['TkDefaultFont', 'Black', 10, 'White', 'CFCFCF']
      self.userFont = 'TkDefaultFont'
      self.userFontColour = 'Black'
      self.userFontSize = 10
      self.userBGOption = 'white'
      self.userWBGOption = '#CFCFCF'


      #universal vars:
      self.datapath = os.path.join('AutoImportDataSheetsFile.xlsx') #avoid system specific input using path object
      self.importedData = []
      self.importingData = ''
      self.importingDataPages = []
```

```
        self.dateFormat = ['Day', 'Month', 'Year', 'Year/Month/Day', 'Month/Day', 'Year/Month']
        self.timeFormat = ['24hr 00:00:00', 'Hour/Min 00:00:_', 'Min/Sec _:00:00', 'Hour 00:_:_', 'Min _:00:_', 'Sec
_:_:00']
        self.tFunctions = []
        self.tFunctionsDisplay = []

        self.linesList = []
        self.compiledLines = []

        self.autoImportData()

    def window(self):
        w = tk.Tk()
        w.title('Data Analysis Program')
        w.geometry('2000x1050')
        return w

    def menubar(self):
        mb = tk.Menu(self.window)            #initialise the Menubar
        self.window.config(menu=mb)        #assign menubar to window
        if platform.system() != 'Windows':
            cas_mb = tk.Menu(mb)
            mb.add_cascade(label='Select Screen', menu=cas_mb)
            return cas_mb
        else:
            return mb

    def autoImportData(self):
        if os.path.exists(self.datapath):
            autoFile = pd.ExcelFile(self.datapath)
            sheetNames = sorted(autoFile.sheet_names)
            for sheet in sheetNames:
                self.importedData.append([sheet, autoFile.parse(sheet_name=sheet)])
        else:
            pass

    def parseElement(self, element):
        if len(element) == 2:
            return f'{element[0]}t^{element[1]}'

        if len(element) == 1:
            return f'{element[0]}'




p = Program()
```

```
##############################################################################################################################################################
##############################################################################################################################################################
##############################################################################################################################################################

class Screen():
    def __init__(self, framesList, scrlabel):
        if platform.system() == 'Windows':
            p.menubar.add_command(label=scrlabel, command=lambda:self.screenSwitch(self))
        else:
            p.menubar.add_cascade(label=scrlabel, command=lambda:self.screenSwitch(self))

        self.framesList = framesList

    def screenSwitch(self, switchto):
        if p.currentScreen != '':
            self.unload(p.currentScreen.framesList)
        p.currentScreen = switchto
        self.load(switchto.framesList)

    def load(self, frames):
        try:
            for frame in frames:
                for widget in frame:
                    widget.usefont(p.userFont, p.userFontColour, p.userFontSize, p.userBGOption)
                    widget.load()
        except:
            for widget in frames:
                widget.usefont(p.userFont, p.userFontColour, p.userFontSize, p.userBGOption)
                widget.load()

    def unload(self, frames):
        try:
            for frame in frames:
                for widget in frame:
                    widget.unload()
        except:
            for widget in frames:
                widget.unload()

##############################################################################################################################################################
##############################################################################################################################################################
##############################################################################################################################################################

class ConstructSettingsScreen(Screen):
    def __init__(self):
        framesList = [self.constructFSF()]
        super().__init__(framesList, 'Settings')

    def constructFSF(self):
        #Font Settings Frame
        frame = [FrameWidget(p.window, 0,0,'nsew',1,1)]
        root = frame[0].body

        frame.append(LabelWidget(root, 0, 0, '', 1, 1, 'Font Style:'))
        frame.append(OptionMenuWidget(root, 0, 1, '', 1, 1, p.fontOptions, self.userfont))
        frame.append(LabelWidget(root, 1, 0, '', 1, 1, 'Font Colour:'))
        frame.append(OptionMenuWidget(root, 1, 1, '', 1, 1, p.fontColours, self.fontColour))
        frame.append(LabelWidget(root, 2, 0, '', 1, 1, 'Font Size:'))
        frame.append(EntryWidget(root, 2, 1, '', 1, 1, '10', 'p_int', self.fontSize))
        frame.append(LabelWidget(root, 3, 0, '', 1, 1, 'Frame Background Colour:'))
        frame.append(OptionMenuWidget(root, 3, 1, '', 1, 1, p.bgOptions, self.fbgTheme))
        frame.append(LabelWidget(root, 4, 0, '', 1, 1, 'Window Background Colour:'))
        frame.append(OptionMenuWidget(root, 4, 1, '', 1, 1, p.bgOptions, self.wbgTheme))
        frame.append(ButtonWidget(root, 0, 3, '', 1, 1, 'Reload Screen', self.reloadScreen))
        return frame

    def userfont(self, *args):
        p.userFont = self.framesList[0][2].var.get()

    def fontColour(self, *args):
        p.userFontColour = self.framesList[0][4].var.get()

    def fontSize(self, *args):
        p.userFontSize = self.framesList[0][6].var.get()

    def fbgTheme(self, *args):
        p.userBGOption = self.framesList[0][8].var.get()

    def wbgTheme(self, *args):
        p.userWBGOption = self.framesList[0][10].var.get()

    def reloadScreen(self):
        self.unload(self.framesList)
        p.window.config(bg=p.userWBGOption)
        self.load(self.framesList)

ConstructSettingsScreen()
###########################################################################################
######################################################
###########################################################################################
######################################################
###########################################################################################
######################################################

class ConstructImportDataScreen(Screen):
    def __init__(self):
        self.normFig, self.normAxis = plt.subplots(1,1,figsize=(6,3),dpi=100)
        self.normFig.tight_layout(rect=[0.1,0,0.8,0.9], pad=1.5)
        self.normAxis.tick_params(axis='x', labelsize=6)
        self.normAxis.tick_params(axis='y', labelsize=6)
        self.normAxis.locator_params(axis='x', nbins=20)
        self.normAxis.locator_params(axis='y', nbins=20)

        self.path = ''
        self.columns = []
        self.usecols = ''
        self.fileType = ''

        self.mean = 0
        self.stdev = 1
        self.cleandata = []
        self.anomalies = []
        self.posInAnom = 0
        self.tempdtvals = []
        self.olddtvals = []
        self.newtvals = []
        self.newdvals = []
        self.oldtvals = []
        self.olddvals = []

        framesList = [self.constructDFF(), self.constructPTF(), self.constructMIF(), self.constructMTDF(),
                self.constructNDA(), self.constructNDGF(), self.constructMTPF()]
        super().__init__(framesList, 'Import Data')



    def constructDFF(self):
        # data file finder
        frame = [FrameWidget(p.window, 0,0,'nsew',1,1)]
        root = frame[0].body

        frame.append(ButtonWidget(root, 0, 0, '', 1, 1, 'Import File', self.fileFinder))
        frame.append(LabelWidget(root, 0, 1, '', 1, 1, 'Set File Name:'))
        frame.append(EntryWidget(root, 0, 2, '', 1, 1, f'Data Set {len(p.importedData)+1}', 'c_str'))
        frame.append(ButtonWidget(root, 0, 3, '', 1, 1, 'Confirm Import', self.confirmImport))
        frame.append(LabelWidget(root, 0, 4, '', 1, 1, 'Auto Import on Program Load:'))
        frame.append(CheckbuttonWidget(root, 0, 5, '', 1, 1))
        return frame

    def constructPTF(self):
        #Preview Text Frame
        frame = [FrameWidget(p.window, 2, 0, 'ewns', 1, 1)]
        root = frame[0].body

        frame.append(TextWidget(root, 1, 1, 'ewns', 1, 1, '', 75, 25, 'disabled', 'none'))
        frame.append(ScrollbarWidget(root, 0, 1, 'ew', 1, 1, 'horizontal', frame[1].body))
        frame.append(ScrollbarWidget(root, 1, 0, 'ns', 1, 1, 'vertical', frame[1].body))
        return frame

    def constructMIF(self):
        #Manage Import Frame
        frame = [FrameWidget(p.window, 1, 0, 'nsew', 1, 1)]
        root = frame[0].body

        frame.append(LabelWidget(root, 0, 0, '', 1, 1, 'Select Sheet:'))
        frame.append(OptionMenuWidget(root, 0, 1, '', 1, 1, ['No Sheets'], self.updateImport))
        frame.append(LabelWidget(root, 2, 0, '', 1, 1, 'Apply Header:'))
        frame.append(EntryWidget(root, 2, 1, '', 1, 1, '0', 'p_int', self.updateImport))
        frame.append(LabelWidget(root, 3, 0, '', 1, 1, 'Apply Footer:'))
        frame.append(EntryWidget(root, 3, 1, '', 1, 1, '0', 'p_int', self.updateImport))
        return frame

    def constructMTDF(self):
        #Manage Time Data Frame
        frame = [FrameWidget(p.window, 0, 1, 'ewns', 2, 1)]
        root = frame[0].body

        frame.append(LabelWidget(root, 0, 0, '', 1, 1, 'Select Column:'))
        frame.append(OptionMenuWidget(root, 0, 1, '', 1, 1, ['No Columns'], self.selectedDTcolumn))
        frame.append(LabelWidget(root, 1, 0, '', 1, 1, 'Date Object Formatting:'))
        frame.append(OptionMenuWidget(root, 1, 1, '', 1, 1, p.dateFormat, self.updateDateTimeFormat))
        frame.append(LabelWidget(root, 2, 0, '', 1, 1, 'Time Object conversion:'))
        frame.append(OptionMenuWidget(root, 2, 1, '', 1, 1, p.timeFormat, self.updateDateTimeFormat))
        frame.append(LabelWidget(root, 1, 2, '', 1, 1, 'Remove Time Component:'))
        frame.append(CheckbuttonWidget(root, 1, 3, '', 1, 1))
        frame.append(LabelWidget(root, 2, 2, '', 1, 1, 'Remove Date Component:'))
        frame.append(CheckbuttonWidget(root, 2, 3, '', 1, 1))
        frame.append(ButtonWidget(root, 0,3, '', 1, 1, 'Apply Changes', self.applyDTchanges))
        return frame

    def constructNDA(self):
        #normal distribution anomalies
        frame = [FrameWidget(p.window, 3, 0, 'ewns', 1, 1)]
        root = frame[0].body

        frame.append(LabelWidget(root, 0, 0, '', 1, 1, 'Create Normal For:'))
        frame.append(OptionMenuWidget(root, 0, 1, '', 1, 1, ['No Columns'], self.normalGraph))
        frame.append(LabelWidget(root, 1, 0, '', 1, 3, 'Detect Anomylous Results'))
        frame.append(LabelWidget(root, 2, 0, '', 1, 1, 'Max Deviation:'))
        frame.append(EntryWidget(root, 2, 1, '', 1, 1, '2', 'p_int', self.obtainAnomalies))
        frame.append(ButtonWidget(root, 3,0, '', 1, 1, '< Prev', self.prevAnom))
        frame.append(TextWidget(root, 3, 1, '', 1, 1, 'None', 15, 1, 'disabled', 'none'))
        frame.append(ButtonWidget(root, 3,2, '', 1, 1, 'Next >', self.nextAnom))
        frame.append(ButtonWidget(root, 4,1, '', 1, 1, 'Replace with Mean', self.removeAnomaly))
        frame.append(LabelWidget(root, 0, 4, '', 1, 1, f'Found {len(self.anomalies)}\nPotential Instances'))
        frame.append(LabelWidget(root, 1, 4, '', 1, 1, f'Current Mean:\n{self.mean}'))
        frame.append(LabelWidget(root, 2, 4, '', 1, 1, f'Current Standard Deviation:\n{self.mean}'))
        return frame
```

```
def constructNDGF(self):
    #normal distribution graph frame
    frame = [FrameWidget(p.window, 3, 1, 'ewns', 1, 1)]
    root = frame[0].body

    frame.append(GraphFigWidget(root, 0, 1, '', 2, 1, self.normFig))
    return frame

def constructMTPF(self):
    #manage time preview frame
    frame = [FrameWidget(p.window, 2, 1, 'ewns', 1, 1)]
    root = frame[0].body

    frame.append(TextWidget(root, 1, 1, 'ewns', 1, 1, '', 75, 25, 'disabled', 'none'))
    frame.append(ScrollbarWidget(root, 1, 0, 'ns', 1, 1, 'vertical', frame[1].body))
    return frame

def normalGraph(self, *args):
    self.normAxis.clear()
    columndata = p.importingData[self.framesList[4][2].var.get()]
    self.cleandata = []
    try:
        for data in columndata:
            self.cleandata.append(float(data))
    except:
        return False

    if len(self.cleandata) < 25:
        bins = len(self.cleandata)
    else:
        bins = 25
    self.normAxis.hist(self.cleandata, bins=bins, density=True, color='b')

    self.mean, self.stdev = stats.norm.fit(self.cleandata)
    x = np.linspace(min(self.cleandata), max(self.cleandata), 100)
    self.normAxis.plot(x, stats.norm.pdf(x, self.mean, self.stdev), 'k', '--')
    self.normAxis.title.set_text(f'Normal of {self.framesList[4][2].var.get()}')
    self.normFig.canvas.draw()
    self.obtainAnomalies()

def obtainAnomalies(self, *args):
    maxdev = int(self.framesList[4][5].var.get())
    self.anomalies = []
    upperlimit = (self.mean + (maxdev * self.stdev) )
    lowerlimit = (self.mean - (maxdev * self.stdev) )
    for value in self.cleandata:
        if value > upperlimit or value < lowerlimit:
            self.anomalies.append(value)

    if len(self.anomalies) > 0:
        self.framesList[4][7].replace(self.anomalies[0])
        self.framesList[4][10].body.config(text=f'Found {len(self.anomalies)}\nPotential Instances')
        self.framesList[4][11].body.config(text=f'Current Mean:\n{self.mean}')
        self.framesList[4][12].body.config(text=f'Current Standard Deviation:\n{self.stdev}')
        self.posInAnom = 0

def prevAnom(self):
    if self.posInAnom > 0:
        self.posInAnom -= 1
        self.framesList[4][7].replace(self.anomalies[self.posInAnom])
    else:
        return False

def nextAnom(self):
    if self.posInAnom < len(self.anomalies) - 1:
        self.posInAnom += 1
        self.framesList[4][7].replace(self.anomalies[self.posInAnom])
    else:
        return False

def removeAnomaly(self):
    colid = self.framesList[4][2].var.get()
    anomaly = self.anomalies[self.posInAnom]
    index = self.cleandata.index(anomaly)

    newcol = []
    for val in p.importingData[colid]:
        if val == anomaly:
            newcol.append(self.mean)
        else:
            newcol.append(val)

    newcoldict = {f'{colid}':newcol}
    newcoldf = pd.DataFrame(newcoldict)
    p.importingData[colid] = newcoldf[colid]
    self.anomalies.pop(self.posInAnom)
    self.framesList[4][10].body.config(text=f'Found {len(self.anomalies)}\nPotential Instances')
    self.framesList[1][1].replace(p.importingData)
    if self.posInAnom > len(self.anomalies)-1:
        self.prevAnom()
    else:
        self.nextAnom()

def selectedDTcolumn(self, *args):
    column = self.framesList[3][2].var.get()
    self.olddtvals = p.importingData[column]
    self.oldtvals = []
    self.olddvals = []
    self.newtvals = []
    self.newdvals = []
    self.tempdtvals = []
    for i, val in enumerate(self.olddtvals):
        if isinstance(val, dt.datetime):
            d, t = val.strftime('%d/%m/%Y %X').split(' ')
            d = dt.datetime.strptime(d, '%d/%m/%Y')
            t = dt.datetime.strptime(t, '%H:%M:%S')
        else:
            if i > 0:
                d, t = self.olddvals[i-1], self.oldtvals[i-1]
            else:
                d, t = self.olddvals[i], self.oldtvals[i]
        self.olddvals.append(d)
        self.oldtvals.append(t)
    self.updateDateTimeFormat()


def updateDateTimeFormat(self, *args):
    self.tempdtvals = []
    self.newdvals = []
    self.newtvals = []
    dformat = self.framesList[3][4].var.get()
    tformat = self.framesList[3][6].var.get()
    #['Day', 'Month', 'Year', 'Day/Month/Year', 'Day/Month', 'Month/Year', 'x-axis (from 0)', 'Remove Date Element']
    if len(self.olddtvals) == 0:
        return False
    incdate = self.framesList[3][10].var.get() #1=no, 0=yes
    inctime = self.framesList[3][8].var.get()
    if incdate == 1 and inctime == 1:
        return False
    for i, datetime in enumerate(self.olddvals):
        if dformat == 'Day':
            self.newdvals.append(datetime.strftime('%d'))
        elif dformat == 'Month':
            self.newdvals.append(datetime.strftime('%m'))
        elif dformat == 'Year':
            self.newdvals.append(datetime.strftime('%Y'))
        elif dformat == 'Year/Month/Day':
            self.newdvals.append(datetime.strftime('%Y/%m/%d'))
        elif dformat == 'Month/Day':
            self.newdvals.append(datetime.strftime('%m/%d'))
        elif dformat == 'Year/Month':
            self.newdvals.append(datetime.strftime('%Y/%m'))
        elif tformat == '24hr 00:00:00':
            self.newtvals.append(datetime.strftime('%H:%M:%S'))
        elif tformat == 'Hour/Min 00:00:_':
            self.newtvals.append(datetime.strftime('%H:%M'))
        elif tformat == 'Min/Sec _:00:00':
            self.newtvals.append(datetime.strftime('%M:%S'))
        elif tformat == 'Hour 00:_:_':
            self.newtvals.append(datetime.strftime('%H'))
        elif tformat == 'Min _:00:_':
            self.newtvals.append(datetime.strftime('%M'))
        elif tformat == 'Sec _:_:00':
            self.newtvals.append(datetime.strftime('%S'))

    if inctime == 1 or len(self.newtvals) == 0:
        self.tempdtvals = self.newdvals
    elif incdate == 1 or len(self.newdvals) == 0:
        self.tempdtvals = self.newtvals
    else:
        for i in range(len(self.newdvals)):
            self.tempdtvals.append(f'{self.newdvals[i]} {self.newtvals[i]}')

    dateDict = {'Old Values:':self.olddtvals, 'New Values:':self.tempdtvals}
    comparison = pd.DataFrame(dateDict)
    self.framesList[6][1].replace(comparison)

def applyDTchanges(self):
    newdtdict = {f'{self.framesList[3][2].var.get()}':self.tempdtvals}
    newdtdf = pd.DataFrame(newdtdict)
    p.importingData[self.framesList[3][2].var.get()] = newdtdf[self.framesList[3][2].var.get()]
    self.framesList[1][1].replace(p.importingData)

def fileFinder(self):
    file = askopenfile(mode='r')
    if file:
        self.path = os.path.join(file.name)
        file, extension = os.path.splitext(file.name)
        if '.csv' in extension:
            self.fileType = 'csv'
            p.importingData = pd.read_csv(self.path)
            self.columns = p.importingData.columns.tolist()
            self.framesList[3][2].changeOptions(self.columns)
            self.framesList[4][2].changeOptions(self.columns)
            self.framesList[1][1].replace(p.importingData)

        elif '.xls' in extension:
            self.fileType = 'xls'
            sheets = sorted(pd.ExcelFile(self.path).sheet_names)
            self.framesList[2][2].changeOptions(sheets)
            self.columns = []

        else:
            pass

def updateImport(self, *args):
    if self.fileType == 'xls':
        p.importingData = pd.read_excel(self.path,
                                        sheet_name=self.framesList[2][2].var.get(),
                                        header=self.framesList[2][4].var.get(),
                                        skipfooter=self.framesList[2][6].var.get())
        self.columns = p.importingData.columns.tolist()
        self.framesList[3][2].changeOptions(self.columns)
        self.framesList[4][2].changeOptions(self.columns)
        self.framesList[1][1].replace(p.importingData)

def confirmImport(self):
    data = p.importingData
    dataName = self.framesList[0][3].var.get().strip()
    if dataName == '':
        return False
    else:
        p.importedData.append([dataName, data])
        if self.framesList[0][6].var.get() == 1:
            if os.path.exists(p.datapath):
                with pd.ExcelWriter(p.datapath, engine='openpyxl', mode='a') as writer:
                    data.to_excel(writer, sheet_name=dataName, index=False)
            else:
                with pd.ExcelWriter(p.datapath, engine='openpyxl', mode='w') as writer:
                    data.to_excel(writer, sheet_name=dataName, index=False)


ConstructImportDataScreen()

###################################################################################################
###########################################################
###################################################################################################
###########################################################
###################################################################################################
###########################################################

class ConstructFunctionDataScreen(Screen):
def __init__(self):
    self.elements = []
    self.positionInFunction = 0
    self.xValues = []
    self.yValues = []
    self.presetFunctions = ['sin(t)', 'cos(t)', 'tan(t)', 'e^(t)', 'e^(t^2)', 'ln(t)']
    framesList = [self.constructSET(), self.constructFOE(), self.constructTRF(), self.constructXYV()]
    super().__init__(framesList, 'Creating Function Data ')

def constructSET(self):
    #simultenous Equations in T
    frame = [FrameWidget(p.window, 0, 0, 'ewns', 1, 1)]
    root = frame[0].body

    frame.append(LabelWidget(root, 0, 0, '', 1, 1, 'Construct Simultenous Equations:'))
    frame.append(ButtonWidget(root, 0, 1, '', 1, 1, 'Update Function Lists', self.updateFunctionLists))
    frame.append(LabelWidget(root, 1, 0, '', 1, 1, 'Y Function:'))
    frame.append(OptionMenuWidget(root, 1, 1, '', 1, 1, ['Functions'], self.createYValues))
    frame.append(LabelWidget(root, 2, 0, '', 1, 1, 'X Function:'))
    frame.append(OptionMenuWidget(root, 2, 1, '', 1, 1, ['Functions'], self.createXValues))

    frame.append(EntryWidget(root, 3, 0, '', 1, 1, f'DatasetName {len(p.importedData)}', 'c_str', None))
    frame.append(ButtonWidget(root, 3, 1, '', 1, 1, 'Confirm Data', self.confirmData))
    frame.append(LabelWidget(root, 3, 2, '', 1, 1, 'Import Data on Program Load:'))
    frame.append(CheckbuttonWidget(root, 3, 3, '', 1, 1))
    return frame

def constructFOE(self):
    #Function Of Elements
    frame = [FrameWidget(p.window, 0, 1, 'ewns', 1, 1)]
    root = frame[0].body

    frame.append(ButtonWidget(root, 0, 1, '', 1, 1, '< Prev Element', self.scrollFunctionR))
    frame.append(ButtonWidget(root, 0, 2, '', 1, 1, 'Remove Element', self.removeElement))
    frame.append(ButtonWidget(root, 0, 3, '', 1, 1, 'Next Element >', self.scrollFunctionL))

    frame.append(LabelWidget(root, 1, 0, '', 1, 1, 'Function:'))
    frame.append(TextWidget(root, 1, 1, '', 1, 1, '', 50, 5, 'disabled', None))
    frame.append(TextWidget(root, 1, 2, '', 1, 1, '', 25, 5, 'disabled', None))
    frame.append(TextWidget(root, 1, 3, '', 1, 1, '', 50, 5, 'disabled', None))

    frame.append(ButtonWidget(root, 2, 0, '', 1, 1, 'Add Power Element', self.addPowerElement))
    frame.append(EntryWidget(root, 2, 1, '', 1, 1, '1', 'p_float', None))
    frame.append(LabelWidget(root, 2, 2, '', 1, 1, 't^'))
    frame.append(EntryWidget(root, 2, 3, '', 1, 1, '0', 'p_float', None))

    frame.append(ButtonWidget(root, 3, 0, '', 1, 1, 'Add Function Element', self.addFunctionElement))
    frame.append(OptionMenuWidget(root, 3, 1, '', 1, 1, self.presetFunctions, None))

    frame.append(ButtonWidget(root, 4, 0, '', 1, 1, 'Store Function', self.confirmFunction))
    return frame

def constructTRF(self):
    #t range frame
    frame = [FrameWidget(p.window, 0, 2, 'ewns', 1, 1)]
    root = frame[0].body

    frame.append(LabelWidget(root, 6, 0, '', 1, 1, 't start:'))
    frame.append(EntryWidget(root, 6, 1, '', 1, 1, '0', 'p_float', None))
    frame.append(LabelWidget(root, 7, 0, '', 1, 1, 't end:'))
    frame.append(EntryWidget(root, 7, 1, '', 1, 1, '100', 'p_float', None))
    frame.append(LabelWidget(root, 8, 0, '', 1, 1, 't step:'))
    frame.append(EntryWidget(root, 8, 1, '', 1, 1, '1', 'p_float', None))
    return frame

def constructXYV(self):
    #X Y values
    frame = [FrameWidget(p.window, 1, 0, 'ew', 1, 3)]
    root = frame[0].body

    frame.append(LabelWidget(root, 1, 0, '', 1, 1, 'Y values:'))
    frame.append(TextWidget(root, 1, 1, 'ew', 1, 1, '', 250, 1, 'disabled', 'none'))
    frame.append(ScrollbarWidget(root, 0, 1, 'ew', 1, 1, 'horizontal', frame[2].body))
    frame.append(LabelWidget(root, 2, 0, '', 1, 1, 'X values:'))
    frame.append(TextWidget(root, 2, 1, 'ew', 1, 1, '', 250, 1, 'disabled', 'none'))
    frame.append(ScrollbarWidget(root, 3, 1, 'ew', 1, 1, 'horizontal', frame[5].body))
    return frame

def parseElement(self, element):
    if len(element) == 2:
        return f'{element[0]}t^{element[1]}'

    if len(element) == 1:
        return f'{element[0]}'

def displayFunctionElements(self):
    L, R, C = '', '', ''
    for i, element in enumerate(self.elements):
        if i < self.positionInFunction:
            L += f'{self.parseElement(element)} + '
        if i == self.positionInFunction:
            C += f'{self.parseElement(element)} + '
        if i > self.positionInFunction:
            R += f'{self.parseElement(element)} + '

    self.framesList[1][5].replace(L)
    self.framesList[1][6].replace(C)
    self.framesList[1][7].replace(R)

def scrollFunctionR(self):
    if self.positionInFunction > 0:
        self.positionInFunction += -1
    self.displayFunctionElements()

def scrollFunctionL(self):
    if self.positionInFunction < len(self.elements) - 1:
        self.positionInFunction += 1
    self.displayFunctionElements()

def addPowerElement(self):
    elementToAdd = [float(self.framesList[1][9].var.get()), float(self.framesList[1][11].var.get())]
    for i, val in enumerate(elementToAdd):
        if int(val) == val:
            elementToAdd[i] = int(val)
    newElementList = []

    if len(self.elements) == 0:
        newElementList.append(elementToAdd)
    else:
        for i, element in enumerate(self.elements):
            if i == self.positionInFunction:
                newElementList.append(element)
                newElementList.append(elementToAdd)
            else:
```

```
                newElementList.append(element)
            self.positionInFunction += 1
        self.elements = newElementList
        self.displayFunctionElements()

    def addFunctionElement(self):
        elementToAdd = [self.framesList[1][13].var.get()]
        newElementList = []
        if len(self.elements) == 0:
            newElementList.append(elementToAdd)
        else:
            for i, element in enumerate(self.elements):
                if i == self.positionInFunction:
                    newElementList.append(element)
                    newElementList.append(elementToAdd)
                else:
                    newElementList.append(element)
            self.positionInFunction += 1
        self.elements = newElementList
        self.displayFunctionElements()

    def removeElement(self):
        for i, element in enumerate(self.elements):
            if i == self.positionInFunction:
                self.elements.pop(i)
        if self.positionInFunction > len(self.elements) -1:
            self.positionInFunction -= 1
        self.displayFunctionElements()

    def confirmFunction(self):
        p.tFunctions.append(self.elements)
        elementsName = ''
        for element in self.elements:
            elementsName += f'{self.parseElement(element)}'
        p.tFunctionsDisplay.append(elementsName)
        self.updateFunctionLists()

    def evaluateFunction(self, functionElements, t):
        pfDict = {'sin(t)':np.sin, 'cos(t)':np.cos, 'tan(t)':np.tan,
                  'e^(t)':np.exp, 'e^(t^2)':np.exp, 'ln(t)':np.log}
        value = 0
        for element in functionElements:
            if len(element) == 2:
                value += element[0] * (t**element[1])
            else:
                if element[0] == 'e^(t^2)':
                    value += pfDict[element[0]](t**2)
                else:
                    value += pfDict[element[0]](t)
        return value
```

```
def updateFunctionLists(self):
    self.framesList[0][4].changeOptions(p.tFunctionsDisplay)
    self.framesList[0][6].changeOptions(p.tFunctionsDisplay)

def obtainSSS(self):
    values = []
    multipliers = []
    intvalues = []
    for s in range(3):
        values.append(str(self.framesList[2][2+ (2*s)].var.get()))
        intval, decval = values[s].split('.')
        if decval != '0':
            multipliers.append(len(decval))
        else:
            multipliers.append(0)
        values[s] = float(values[s])
    x10 = max(multipliers)
    for s in range(3):
        intvalues.append(int(values[s] * (10**x10)))
    return intvalues, values

def createXValues(self, *args):
    index = p.tFunctionsDisplay.index(self.framesList[0][6].var.get())
    self.XfunctionElements = p.tFunctions[index]
    self.xValues = []

    #start, stop, step
    intvals, decvals = self.obtainSSS()
    t, step = decvals[0], decvals[2]
    for i in range(intvals[0], intvals[1], intvals[2]):
        value = self.evaluateFunction(self.XfunctionElements, t)
        self.xValues.append(value)
        t += step
    self.framesList[3][5].replace(self.xValues)

def createYValues(self, *args):
    index = p.tFunctionsDisplay.index(self.framesList[0][4].var.get())
    self.YfunctionElements = p.tFunctions[index]
    self.yValues = []

    #start, stop, step
    intvals, decvals = self.obtainSSS()
    t, step = decvals[0], decvals[2]
    for i in range(intvals[0], intvals[1], intvals[2]):
        value = self.evaluateFunction(self.YfunctionElements, t)
        self.yValues.append(value)
        t += step
    self.framesList[3][2].replace(self.yValues)

def confirmData(self):
    name = self.framesList[0][7].var.get().strip()
```

```
        if name != '':
            data = {'Yvalues':self.yValues, 'Xvalues':self.xValues}
            data = pd.DataFrame(data)
            p.importedData.append([name, data])
            if self.framesList[0][10].var.get() == 1:
                if os.path.exists(p.datapath):
                    with pd.ExcelWriter(p.datapath, engine='openpyxl', mode='a') as writer:
                        data.to_excel(writer, sheet_name=name, index=False)
                else:
                    with pd.ExcelWriter(p.datapath, engine='openpyxl', mode='w') as writer:
                        data.to_excel(writer, sheet_name=name, index=False)



ConstructFunctionDataScreen()



####################################################################################################
#############################################################
####################################################################################################
#############################################################
####################################################################################################
#############################################################

class Line():
    def __init__(self, xvals, yvals, root, rootfig, name, lineid, linecol, linestyle, markerstyle, markersize):
        p.linesList.append(self)
        self.xvals = xvals
        self.yvals = yvals
        self.removed = False
        if xvals == 'axis':
            for i in range(len(yvals)):
                self.xvals.append(i)
        if yvals == 'axis':
            for i in range(len(xaxis)):
                self.xvals.append(i)
        self.workingArgs = [self.xvals, self.yvals, root, rootfig, name, lineid, linecol, linestyle, markerstyle,
int(markersize)]
        self.confirmedArgs = [] #rootaxis, rootfig, name, id, linecol, linestyle, markerstyle, markercol
        self.markersize = int(markersize)

    def graph(self, vals):
        #xvals, yvals, root, rootfig, name, lineid, linecol, linestyle, markerstyle, markersize
        vals[2].plot(vals[0], vals[1],
                     label=f'{vals[4]}\nid:{vals[5]}',
                     color=vals[6],
                     marker=vals[8],
                     markersize=vals[9],
                     linestyle=vals[7])
```

```
  def update(self, linecol, linestyle, markerstyle, markersize):
    self.workingArgs = [self.workingArgs[0], self.workingArgs[1], self.workingArgs[2], self.workingArgs[3],
                self.workingArgs[4], self.workingArgs[5],
                linecol, linestyle, markerstyle, int(markersize)]

  def remove(self):
    self.removed = True

  def delete(self):
    del self


##########################################################################################
#####################################################
##########################################################################################
#####################################################
##########################################################################################
#####################################################

class ConstructDataGraphScreen(Screen):
  def __init__(self):
    self.graphFig, self.graphAxis = plt.subplots(2,1,figsize=(9,6),dpi=100)
    self.graphFig.tight_layout(rect=[0.1,0,0.8,0.9], pad=1.5)
    #axis0 for completed, axis1 for working
    self.graphAxis[0].tick_params(axis='x', labelsize=6)
    self.graphAxis[0].tick_params(axis='y', labelsize=6)
    self.graphAxis[1].tick_params(axis='x', labelsize=6)
    self.graphAxis[1].tick_params(axis='y', labelsize=6)

    self.bflineid = ''
    self.linetofit = ''

    self.storedGraphInfo = []
    self.tempStoredGraphInfo = []
    self.colour = ['red', 'blue', 'green', 'cyan', 'magenta', 'black']
    self.marker = ['point', 'circle', 'triangle', 'cross']
    self.style = ['solid','dashed','dotted','dot/dash', 'none']
    self.colourDict = {'red':'r', 'blue':'b', 'green':'g', 'cyan':'c', 'magenta':'m', 'black':'k'}
    self.markerDict = {'point':'.', 'circle':'o', 'triangle':'v', 'cross':'x'}
    self.styleDict = {'solid':'-','dashed':'--','dotted':':','dot/dash':'-.', 'none':'none'}
    framesList = [self.constructDPF(), self.constructMGF(), self.constructMDF(), self.constructGraphs(),
self.constructLBF()]
    super().__init__(framesList, 'Graph Data')



  def constructDPF(self):
    #data plotter frame
    frame = [FrameWidget(p.window, 0, 0, 'ewns', 1, 1)]
    root = frame[0].body
```

```
        frame.append(LabelWidget(root, 0, 0, '', 2, 1, 'Plot Data to Frame:'))
        frame.append(LabelWidget(root, 0, 1, '', 1, 1, 'Dataset:'))
        frame.append(LabelWidget(root, 0, 2, '', 1, 1, 'Y-axis:'))
        frame.append(LabelWidget(root, 0, 3, '', 1, 1, 'X-axis:'))
        frame.append(OptionMenuWidget(root, 1, 1, '', 1, 1, ['No Datasets'], self.updateDataSelection))
        frame.append(OptionMenuWidget(root, 1, 2, '', 1, 1, ['No Columns'], None))
        frame.append(OptionMenuWidget(root, 1, 3, '', 1, 1, ['No Columns'], None))
        frame.append(ButtonWidget(root, 0, 4, '', 2, 1, 'Load Stored Data', self.updateDataOptions))
        frame.append(EntryWidget(root, 0, 5, '', 2, 1, 'Line Label', 'c_str', None))
        frame.append(ButtonWidget(root, 2, 0, '', 1, 1, 'Add to Graph', self.addGraphElement))
        return frame

    def constructMGF(self):
        #manage graph frame
        frame = [FrameWidget(p.window, 0, 1, 'ewns', 1, 1)]
        root = frame[0].body

        frame.append(LabelWidget(root, 0, 0, '', 1, 1, 'Graph Title:'))
        frame.append(LabelWidget(root, 1, 0, '', 1, 1, 'X-axis Title:'))
        frame.append(LabelWidget(root, 2, 0, '', 1, 1, 'Y-axis Title:'))
        frame.append(EntryWidget(root, 0, 1, '', 1, 1, 'Title', 'c_str', self.applyTitles))
        frame.append(EntryWidget(root, 1, 1, '', 1, 1, 'X-axis', 'c_str', self.applyTitles))
        frame.append(EntryWidget(root, 2, 1, '', 1, 1, 'Y-axis', 'c_str', self.applyTitles))
        frame.append(ButtonWidget(root, 0, 2, '', 3, 1, 'Save Graph as PDF', self.saveImage))
        return frame

    def constructMDF(self):
        #manage data frame
        frame = [FrameWidget(p.window, 1, 1, 'ewns', 1, 1)]
        root = frame[0].body

        frame.append(LabelWidget(root, 0, 0, '', 1, 1, 'Update Line ID:'))
        frame.append(EntryWidget(root, 0, 1, '', 1, 1, '0', 'p_int', None))
        frame.append(ButtonWidget(root, 0, 2, '', 1, 1, 'Remove Line', self.removeLine))
        frame.append(LabelWidget(root, 1, 0, '', 1, 1, 'Line Colour'))
        frame.append(OptionMenuWidget(root, 1, 1, '', 1, 1, self.colour, self.updateLine))
        frame.append(LabelWidget(root, 2, 0, '', 1, 1, 'Line Style'))
        frame.append(OptionMenuWidget(root, 2, 1, '', 1, 1, self.style, self.updateLine))
        frame.append(LabelWidget(root, 3, 0, '', 1, 1, 'Marker Style'))
        frame.append(OptionMenuWidget(root, 3, 1, '', 1, 1, self.marker, self.updateLine))
        frame.append(LabelWidget(root, 4, 0, '', 1, 1, 'Marker Size'))
        frame.append(EntryWidget(root, 4, 1, '', 1, 1, '0', 'p_int', self.updateLine))
        frame.append(LabelWidget(root, 5, 0, '', 1, 1, 'Change Label:'))
        frame.append(EntryWidget(root, 5, 1, '', 1, 1, 'Label Name', 'c_str', None))
        frame.append(ButtonWidget(root, 5, 2, '', 1, 1, 'Change Label', self.updateLabel))
        return frame

    def constructGraphs(self):
        #make graph widget
        frame = [FrameWidget(p.window, 1, 0, 'ewns', 1, 1)]
        root = frame[0].body
        frame.append(GraphFigWidget(root, 0, 1, '', 2, 1, self.graphFig))
        frame.append(ButtonWidget(root, 0, 0, '', 1, 1, 'Confirm Changes', self.confirmChanges))
        frame.append(ButtonWidget(root, 1, 0, '', 1, 1, 'Clear Changes', self.clearChanges))
        return frame

    def constructLBF(self):
        # line of best fit
        frame = [FrameWidget(p.window, 2, 0, 'ewns', 1, 2)]
        root = frame[0].body

        frame.append(LabelWidget(root, 0, 0, '', 1, 1, 'Construct Best Fit to Line ID:'))
        frame.append(EntryWidget(root, 0, 1, '', 1, 1, '0', 'p_int', None))
        frame.append(LabelWidget(root, 1, 0, '', 1, 1, 'Max Poly Order:'))
        frame.append(EntryWidget(root, 1, 1, '', 1, 1, '5', 'p_int', None))
        frame.append(LabelWidget(root, 2, 0, '', 1, 1, 'Label:'))
        frame.append(EntryWidget(root, 2, 1, '', 2, 1, 'Best Fit Line', 'c_str', None))
        frame.append(ButtonWidget(root, 3, 0, '', 1, 1, 'Add Best Fit', self.addBestFit))
        return frame

    def updateDataOptions(self, *args):
        options = []
        for data in p.importedData:
            options.append(data[0])
        self.framesList[0][5].changeOptions(options)

    def updateDataSelection(self, *args):
        names = []
        for data in p.importedData:
            names.append(data[0])
        index = names.index(self.framesList[0][5].var.get())
        dataset = p.importedData[index][1]
        columns = list(dataset.columns)
        columns.append('axis')
        self.framesList[0][6].changeOptions(columns)
        self.framesList[0][7].changeOptions(columns)

    def addGraphElement(self):
        #xvals, yvals, root, rootfig, name, lineid, linecol, linestyle, markerstyle, markersize
        dataname = self.framesList[0][5].var.get()
        yid = self.framesList[0][6].var.get()
        xid = self.framesList[0][7].var.get()
        dataid = -1
        for i, data in enumerate(p.importedData):
            if data[0] == dataname:
                dataid = i

        yvals = []
        xvals = []
        if xid == 'axis' and yid == 'axis':
            return False
        elif xid == 'axis':
            for val in p.importedData[dataid][1][yid]:
                yvals.append(val)
            for i in range(len(yvals)):
                xvals.append(i)
        elif yid == 'axis':
            for val in p.importedData[dataid][1][xid]:
                xvals.append(val)
            for i in range(len(xvals)):
                yvals.append(i)
        else:
            for val in p.importedData[dataid][1][xid]:
                xvals.append(val)
            for val in p.importedData[dataid][1][yid]:
                yvals.append(val)

        root = self.graphAxis[1]
        rootfig = self.graphFig
        name = self.framesList[0][9].var.get()
        lineid = len(p.linesList)
        linecol = self.colourDict[self.framesList[2][5].var.get()]
        linestyle = self.styleDict[self.framesList[2][7].var.get()]
        markerstyle = self.markerDict[self.framesList[2][9].var.get()]
        markersize = self.framesList[2][11].var.get()
        Line(xvals, yvals, root, rootfig, name, lineid, linecol, linestyle, markerstyle, markersize)
        self.updateGraphAxis(self.graphAxis[1], self.graphFig, p.linesList)

    def applyTitles(self, *args):
        self.graphAxis[0].set_xlabel(self.framesList[1][5].var.get())
        self.graphAxis[0].set_ylabel(self.framesList[1][6].var.get())
        self.graphAxis[1].set_xlabel(self.framesList[1][5].var.get())
        self.graphAxis[1].set_ylabel(self.framesList[1][6].var.get())
        self.graphAxis[0].title.set_text(self.framesList[1][4].var.get())
        self.graphFig.canvas.draw()

    def updateLine(self, *args):
        colourDict = {'red':'r', 'blue':'b', 'green':'g', 'cyan':'c', 'magenta':'m', 'black':'k'}
        markerDict = {'point':'.', 'circle':'o', 'triangle':'v', 'cross':'x'}
        styleDict = {'solid':'-','dashed':'--','dotted':':','dot/dash':'-.', 'none':'none'}
        #linecol, linestyle, markerstyle, markersize
        lineid = int(self.framesList[2][2].var.get())
        line = p.linesList[lineid]

        linecol = colourDict[self.framesList[2][5].var.get()]
        linestyle = styleDict[self.framesList[2][7].var.get()]
        markerstyle = markerDict[self.framesList[2][9].var.get()]
        markersize = self.framesList[2][11].var.get()
        line.update(linecol, linestyle, markerstyle, markersize)
        self.updateGraphAxis(self.graphAxis[1], self.graphFig, p.linesList)

    def updateLabel(self, *args):
        label = self.framesList[2][13].var.get()
        lineid = int(self.framesList[2][2].var.get())
        line = p.linesList[lineid]
        line.workingArgs[4] = label
        self.updateGraphAxis(self.graphAxis[1], self.graphFig, p.linesList)

    def updateGraphAxis(self, axis, fig, lines):
        axis.clear()
        print(len(lines), lines)
        xvalslen = []
        for line in lines:
            print('line')
            if line.removed == False:
                if axis == self.graphAxis[0]:
                    line.graph(line.confirmedArgs)
                else:
                    line.graph(line.workingArgs)
        axis.legend(bbox_to_anchor=(1.04,1),loc='upper left', borderaxespad=0)
        axis.locator_params(axis='both', nbins=15)
        self.applyTitles()
        fig.canvas.draw()

    def removeLine(self):
        lineid = int(self.framesList[2][2].var.get())
        line = p.linesList[lineid]
        line.remove()
        self.updateGraphAxis(self.graphAxis[1], self.graphFig, p.linesList)

    def confirmChanges(self):
        for line in p.linesList:
            line.confirmedArgs = []
            if line.removed == True:
                removeid = 0
                for i, linetoshift in enumerate(p.linesList):
                    if linetoshift.workingArgs[5] > line.workingArgs[5]:
                        linetoshift.workingArgs[5] -= 1
                    if line == linetoshift:
                        removeid = i
                p.linesList.pop(removeid)
                del line
            else:
                if len(p.linesList) == 0:
                    break

                for element in line.workingArgs:
                    line.confirmedArgs.append(element)
                line.confirmedArgs[2] = self.graphAxis[0]
        self.updateGraphAxis(self.graphAxis[0], self.graphFig, p.linesList)

    def clearChanges(self):
        for line in p.linesList:
            if line.confirmedArgs == []:
                del p.linesList[line.workingArgs[5]]
            else:
                if line.removed == True:
                    line.removed = False
                line.workingArgs = []
                for element in line.confirmedArgs:
                    line.workingArgs.append(element)
                line.workingArgs[2] = self.graphAxis[1]
        self.updateGraphAxis(self.graphAxis[1], self.graphFig, p.linesList)

    def saveImage(self):
        title = self.framesList[1][4].var.get().strip()
        title = f'{title}.pdf'
        try:
            bounds = self.graphAxis[0].get_window_extent().transformed(self.graphFig.dpi_scale_trans.inverted())
            self.graphFig.savefig(title, bbox_inches=bounds.expanded(1.45,1.5))
        except:
            print('file name error')

    def orderFunction(self, x, *coeffs):
        function = 0
        for order, coeff in enumerate(coeffs):
            function += coeff*(x**order)
        return function

    def addBestFit(self):

        lineid = int(self.framesList[4][2].var.get())
        self.linetofit = p.linesList[lineid]
        linename = self.framesList[4][6].var.get()
        xvals = self.linetofit.confirmedArgs[0]
        yvals = self.linetofit.confirmedArgs[1]
        maxpoly = int(self.framesList[4][4].var.get())

        #create Best Fit Line
        #create function for best fit line:
        for f in range(maxpoly):
            coeffs, _ = curve_fit(self.orderFunction, xvals, yvals, p0 = [0]*(f+1))


        #determine (x,y) values
        bfxvals = np.arange(min(xvals), max(xvals), 1)
        bfyvals = self.orderFunction(bfxvals, *coeffs)

        #create line object:
        #xvals, yvals, root, rootfig, name, lineid, linecol, linestyle, markerstyle, markersize
        self.bflineid = len(p.linesList)
        Line(bfxvals, bfyvals, self.graphAxis[1], self.graphFig, linename, self.bflineid, 'r', '--', '.', '0')
        self.updateGraphAxis(self.graphAxis[1], self.graphFig, p.linesList)
```

```
ConstructDataGraphScreen()

##############################################################################################################
##################################################################
##############################################################################################################
##################################################################
##############################################################################################################
##################################################################
```