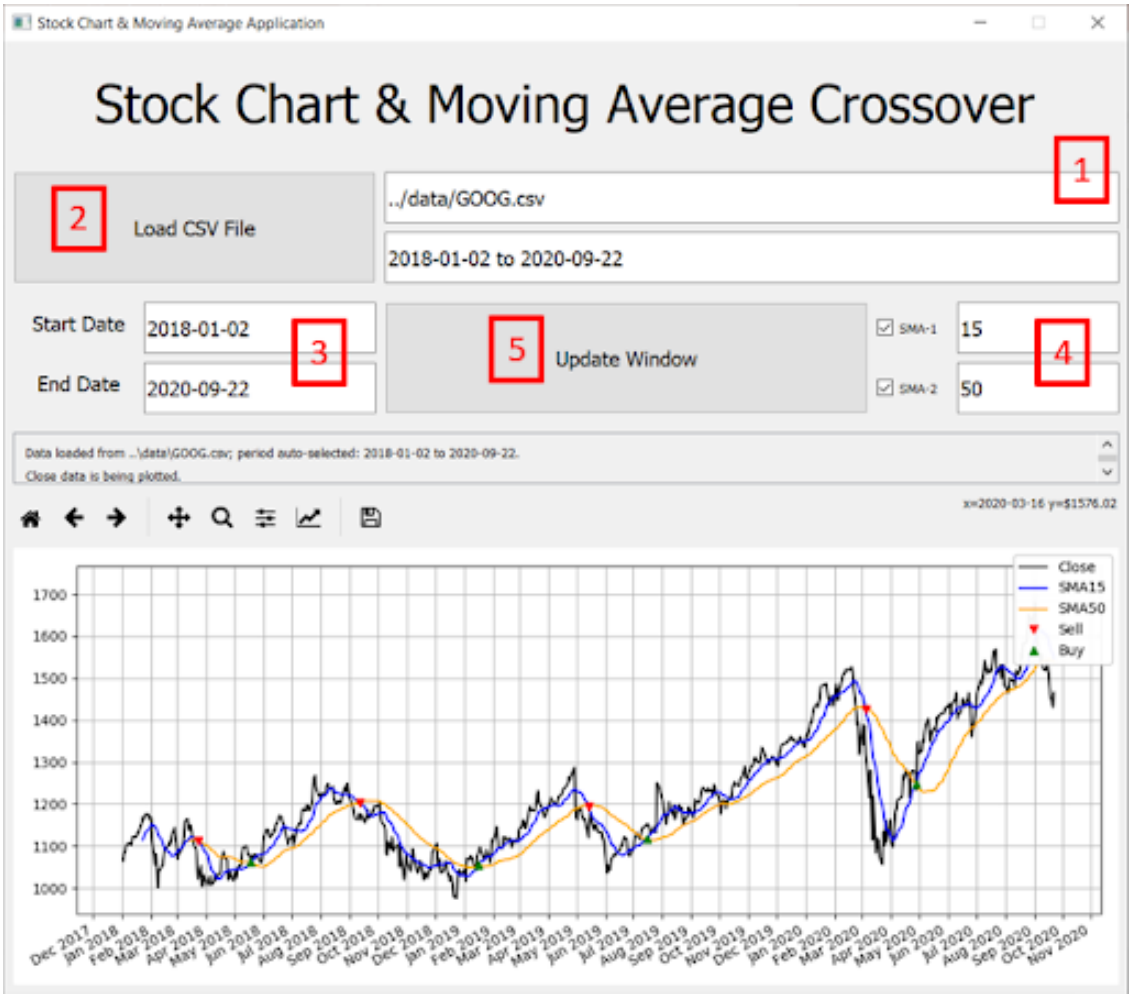
# Python Programming and Its Applications in Stock Chart & Moving Average (MA) Crossover

November 16, 2020

## 1 User Manual




1: Input the location of the stock data file here. In this case we have a relative path that directs to the "data" folder in StockChartApplication, where our sample data file, `GOOG.csv` is located.

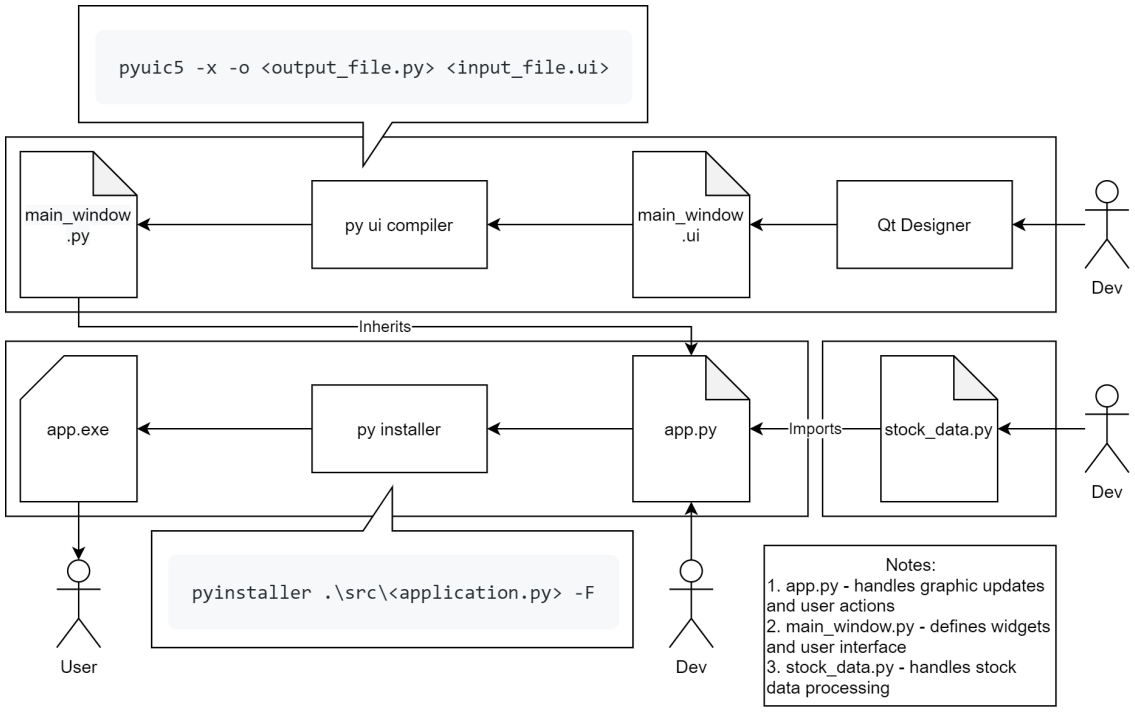
Alternate: an absolute path can also be inputted using the exact directory address of the file.

2: Loads the information in the stock data file into the application

3: The date range to be analysed can be changed here.

4: The number of days used to compute the simple moving averages can be changed and toggled here.

5: Updates the graph to plot the stock price, the two SMAs, the location of crossovers, and the signal presented by the crossover.




The user interface was developed using Qt Designer and saved into `main_window.ui`. This graphical user interface (GUI) file is then converted to `main_window.py` using a py ui compiler. `main_window.py` thus defines the widgets and the user interface for the application.

`stock_data.py` processes the stock data file provided by the user. This data file can be downloaded from sites such as yahoo finance by searching for a company and date range of the user's choice. The`stock_data.py` handles the stock data processing for the application and calculates the simple moving averages as well as identifying the locations of crossovers, before adding them to the stock data file.

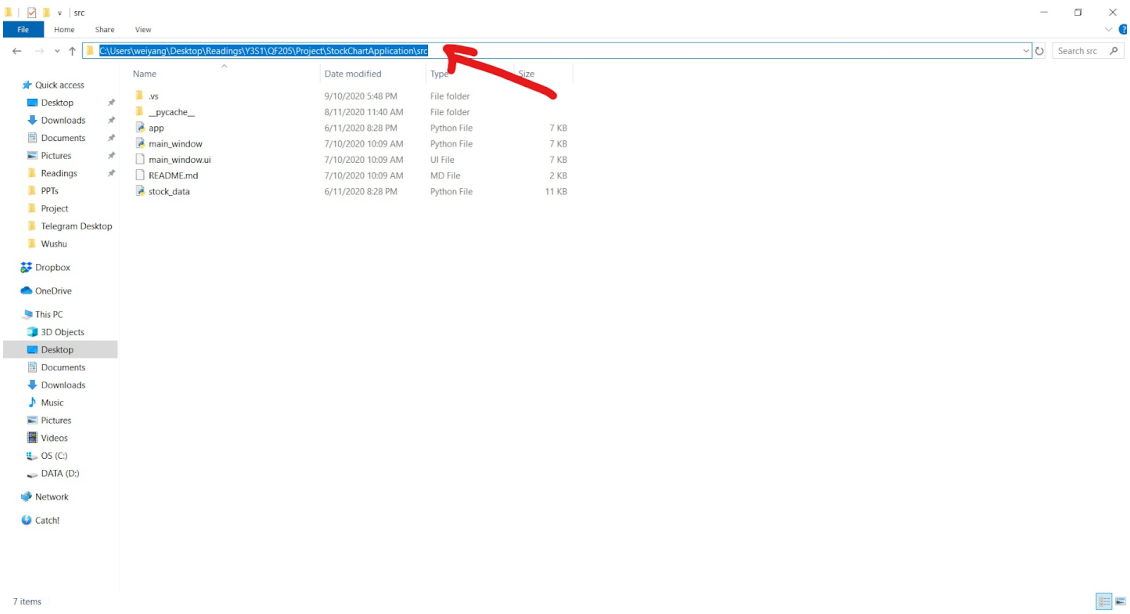
`app.py` inherits the GUI from `main_window.py` and imports the data generated by `stock_data.py`, combining the two and completing the application. Pyinstaller was then used to package `app.py` into `app.exe`, which can be easily run by the user. Running `app.py` through a python shell or just running `app.exe` will launch the application.

Step 0: Install a python distributor. In our case we will use Anaconda. Anaconda can be downloaded and installed from:

Windows: https://repo.anaconda.com/archive/Anaconda3-2020.07-Windows-x86_64.exe

Mac OS: https://repo.anaconda.com/archive/Anaconda3-2020.07-MacOSX-x86_64.pkg

Step 1: go to the src folder in the StockChartApplication, click on the address bar, and copy the directory location




Step 2: search for and run the command line shell of your python distributor. In this case we will search for and run Anaconda Prompt.

Step 3: In the command line shell, type `cd` and then paste the directory address of the "src" folder then press `enter` to change the directory

Step 4: type `python app.py` and press `enter` to run the Stock Chart Application

[ ]:

# 2 Python Basics

## 2.1 Variables

Variables are containers for storing data values. Unlike other programming languages, a variable is created the moment you first assign a value to it. to assign a value to a variable, use the (=) sign.

Use print() to see the value assigned to the variable

[3]:
```
x = 'apple'
Y = 1
_1 = 2
print (x,Y,_1)
```

```
apple 1 2
```

Variable names in Python can be any length and can consist of uppercase and lowercase letters (A-Z, a-z), digits (0-9), and the underscore character (_). An additional restriction is that, although a variable name can contain digits, the first character of a variable name cannot be a digit.

[4]:
```
1apple = apple
```

```
  File "<ipython-input-4-f38a6c9f9c71>", line 1
    1apple = apple
     ^
SyntaxError: invalid syntax
```

## 2.2 Data types

Common Data types that are used in the projects includes

Text type - String (str)

Numeric type - Integer (int) - Float (float)

Sequence Type - list - range

Using `Print(type())` we can see the data type.

### 2.2.1 Text type

String(str)

Strings literal are denoted by single (' ') or double quotes (" ")

Examples of String

[ ]:
```
x = 'apple'
y = "pear"
print(type(x))
```

We can find the length of the variable by using the in build function len()

[ ]:
```
print(len(x))
print(len(y))
```

### 2.2.2 Numeric type

Integer (Int)

Int are whole integers which has no decimal points and have unlimited length. Examples of Integers. Underscore are allowed between integer groupings

[2]:
```
x = 1_3
y = -2
print(type(x))
```

```
<class 'int'>
```

Wrong examples:

Example z gives a invalid token error because leading zero in a non-zero decimal number are not allowed

[3]:
```
z = 01
```

```
  File "<ipython-input-3-360f18516b30>", line 1
    z = 01
        ^
SyntaxError: invalid token
```

Float(float)

Float are Floating point numbers that contains one or more decimals

underscore are allowed between float groupings

[ ]:
```
x = 1.0
y = -3.14_24
print(type(y))
```

Int can be coverted to float, vice versa.

[ ]:
```
x = 1
print(type((x)))
print(x)
```

```
x = float(x)
print(type((x)))
print(x)
x= int(x)
print(type((x)))
print(x)
```

Wrong examples:

Example z gives a syntax error because as the underscore is not between digits.

[4]:
```
z = 31._3
```

```
  File "<ipython-input-4-16e45c8381ba>", line 1
    z = 31._3
           ^
SyntaxError: invalid syntax
```

[ ]:
```
```

# 3 stock_data.py

## 3.1 Import for StockData.py

Firstly, we import all the packages that would be used in StockData.py. We used the `import` statement and created an alias for the packages using the `as` statement. We import `numpy` because we would be using some of the in-built functions such as np.nan. `pandas` package would enable us to read and overwrite our CSV datafiles. `matplotlib.pyplot` package would be used for plotting the stock data into graphs.

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

> *Learning points: The programmer can create alias for their imported packages so that it would be easier for them to recognize and use the functions in the packages.*

[ ]:
```
```

# 4 main_window.py

As mentioned, `main_window.py`'s main responsibility is to **define the graphic user interface (GUI) itself**. It does so by:

1. Defining each `Widget` objects' and their names within the GUI

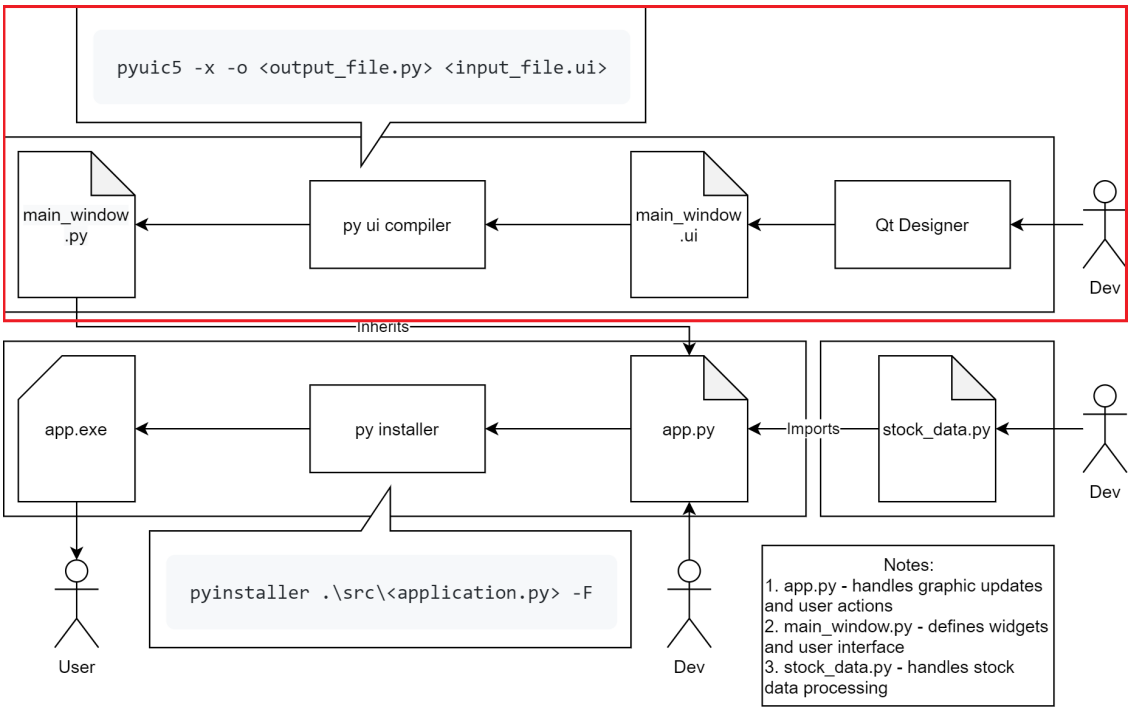
2. Defining the location, size and other physical attributes of each `Widgets`

It does **NOT** define the functionalities of the `Widgets` found in the GUI. That is the job of `app.py`.

While it is possible to create `main_window.py` by manually writing a python script file from scratch, it is cumbersome. Instead, the following method was used develop the Stock Chart Application:

1. Install `Qt Designer` application
2. Use `Qt Designer` to build the GUI file called: `main_window.ui`
3. Pip install `PyQt5` for python
4. Use `pyuic5` (a utility script that comes with `PyQt5`) to compile `main_window.ui` into `main_window.py`

The above-mentioned `main_window.py`'s development process is summarized in the graphics below:




This method is **recommended** because it is user-friendly and changes made can be seen visually on the `Qt Designer` itself before it is applied. Thus, not requiring the developer to run the python file after every changes or even knowing how do so at all.

This section of the report will now go through the 4 steps of developing `main_window.py` mentioned.

## 4.1 Installing `Qt Designer`

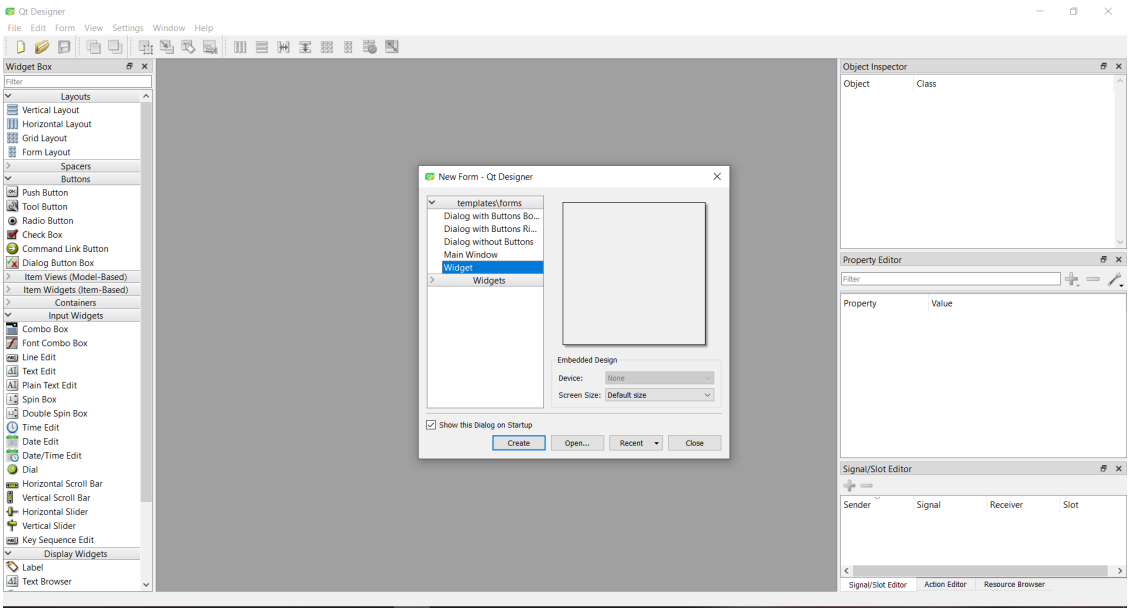
The installation process of `Qt Designer` is similar to any other application.

1. Go to: https://build-system.fman.io/qt-designer-download
2. Click either the `Windows` or `Mac` option. Depending on your computer's Operating System
3. Select a location for the Qt Setup Application `.exe` to be downloaded
4. Double click on the Qt Setup Application `.exe` and follow its installation procedure
5. Check that you have `Qt Designer` installed after the installation has completed

## 4.2 Building `main_window.ui` with `Qt Designer`

### 4.2.1 Defining the GUI

First, open `Qt Designer`. The following window and prompts will appear:



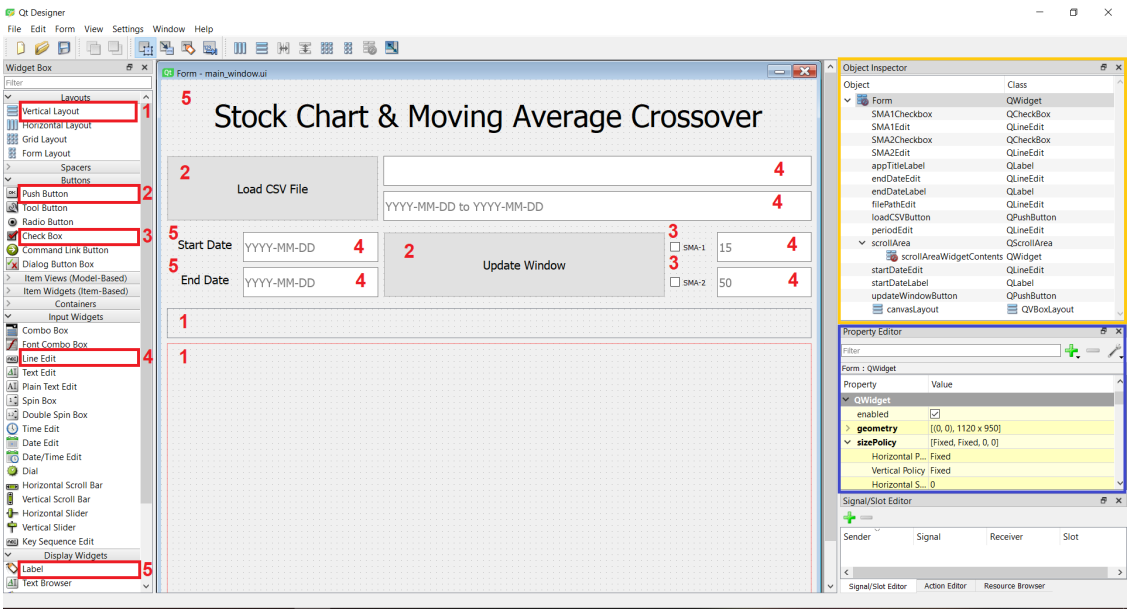

Choose `Widget` under the `template\forms` prompt and press the `Create` Button to begin designing `main_window.ui`.

This is simply a starting template of our GUI, but it is important as the `Widget` option will later be used to inform `app.py` of the type of GUI being inherited.

> *Learning Point: `Qt Designer` + `PyQt5` Template*
>
> *The information about the template is specified when the `.ui` file is started. The information is important because it specifies they **type** of GUI being inherited later. In this case, the `Widget` called `UI_Form` is going to be inherited by `app.py`*

### 4.2.2 Defining the `Widgets` inside the GUI

Second, start designing the `main_window.ui` GUI as shown in the image below:




To 'design' the GUI, simply **drag and drop** the appropriate **type** of `Widget` from the left side-bar called `Widget Box` into the GUI `Widget`.

This does imply that our GUI is a `Widget` (because we specify it as such in the `template\forms` option) containing `Widgets`.

For convenience, the **type** of the `Widget` used to make the GUI shown above has ben annotated with red boxes and numbers to show where to find each **type** of `Widgets` used to build the GUI.

> *Learning Point:* ***`Qt Designer`*** *+* ***`PyQt5 Widget`*** *Types*
>
> 1. `Vertical Layout` *: a layout to mark certain area*
> 2. `Push Button` *: an interactive button*
> 3. `Check Box` *: an interactive checkbox*
> 4. `Line Edit` *: a place to enter a line of text*
> 5. `Label` *: a non-interactive label to display texts*

For each `Widget` being dragged and dropped into the GUI, remember to **name them accordingly** by editing the value of the `objectName` in the Property Editor (blue box). There are also other attributes values to play with!

For instance, this Stock Chart Application has its window **fixed to a specific size**. This can be done by specifying the following properties in the Property Editor of the `UI Form` (found in the Object Inspector):

1. Set `geometry` to: `[(0, 0), 1120 x 950]`
2. Set `sizePolicy` to: `[Fixed, Fixed, 0, 0]`

Tips: To preview the GUI inside `Qt Designer`, press `Ctrl + R` (for Windows users only).

> *Learning Point:* ***`Qt Designer`*** *+* ***`PyQt5 Widget`*** *Attributes*
>
> *Different* ***`Widget`*** *will have different attributes. They can be found in the Property Editor. Some important attributes include:* ***`objectName`****,* ***`geometry`****,* ***`sizePolicy`****,* ***`font`****, etc...*

Also, do refer to the Object Inspector (yellow box) in the `main_window.ui` image for a list of the **names of the widget** and their associated `Widget` **type**.

For example: name (Object): `SMA1CheckBox`, class (type): `QCheckBox`.

In short, these 2 actions: **dragging and dropping `Widgets` and editing values in `Property Editor`** correspond to what were initially meant by:

1. Defining each `Widget` objects' and their names within the GUI
2. Defining the location, size and other physical attributes of each `Widgets`

Finally, to save the `main_window.ui` file, press: `File > Save As` option on the top left hand corner of the window.

### 4.3 Installing PyQt5

Installing `PyQt5` is similar to installing any other python packages using `PIP`. Simply run the following command from the computer's terminal:

```
pip install PyQt5
```

`PyQt5` is a package comprising a comprehensive set of Python bindings for `Qt Designer v5`. As part of its package, it comes with a utility script called `pyuic5` which will be used to compile `.ui` files created using `Qt Designer` into a `.py` python module file.

### 4.4 Compiling `main_window.ui` into `main_window.py`

To compile the `main_window.ui` file into `main_window.py`, simply run the following command from the computer's terminal:

```
pyuic5 -x -o .\src\main_window.py .\src\main_window.ui
```

- The two flags `-x -o` are **required** for the program to work.
- The two arguments passed are also **required** as they are the `output` file path and the `input` file path.

Note: the two file paths assume that the command is run from the `root` directory and the `main_window.ui` file is saved in a directory called `src`.

## 5 `app.py`

While `main_window.py`'s responsibility is to **define the graphics user interface**, `app.py`'s responsibility is to **define the functionalities of the GUI**. This is achieved by doing 2 things:

1. Defining `functions` to accomplish certain actions
2. Connecting `Widget` actions to these `functions`

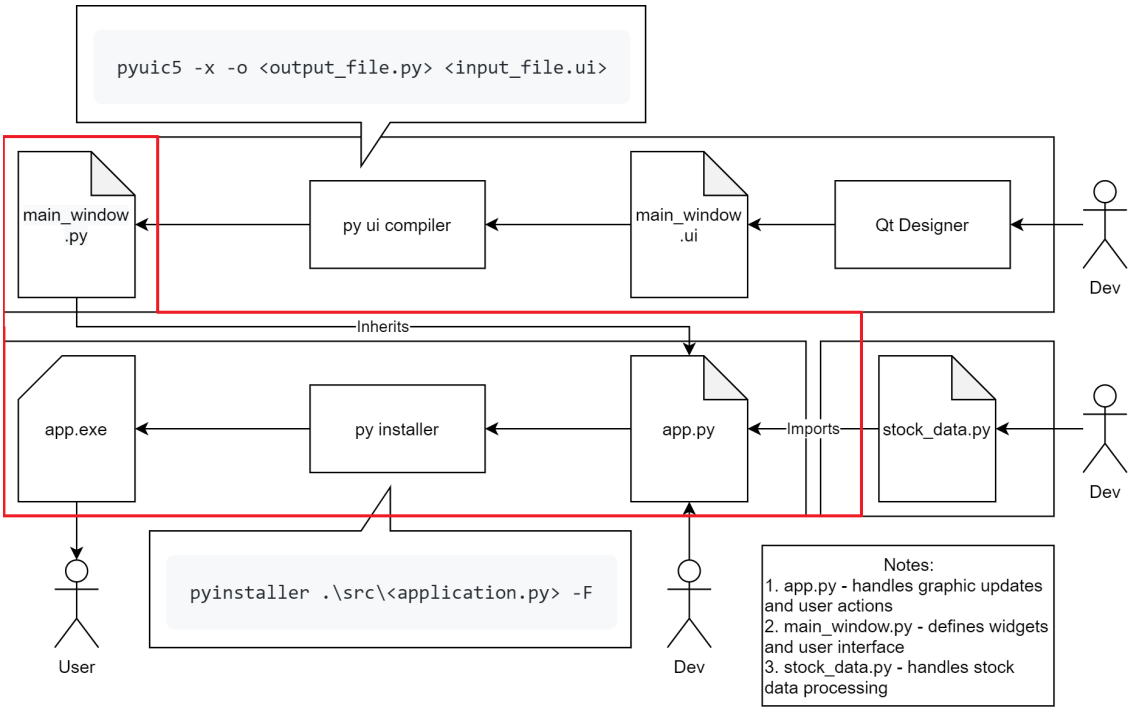
For example, if we want the `Update Window Button` to plot the stock prices in the GUI's canvas. We will have to create a `function` that plots the graph into the canvas and then connect the `Update Window Button` to this `function`.

However, before doing so, `app.py` must first know the `Widget names` defined in `main_window.py`.

For example, the `Update Window Button` is actually named: `updateWindowButton`. This name is defined on the previous section, when `main_window.ui` was designed using `Qt Designer` and the `objectName` is specified inside the Property Editor!

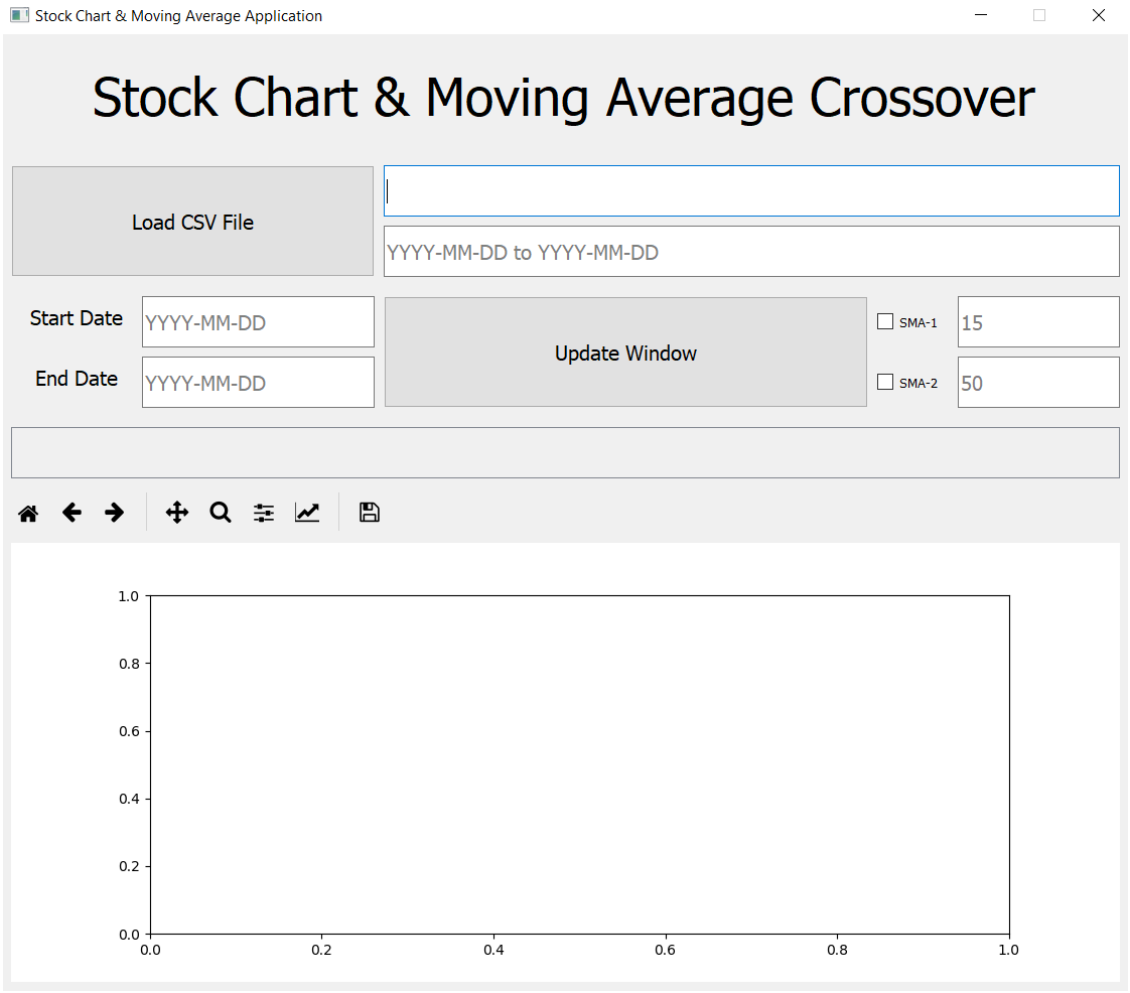
This is why, on the previous step, it is **recommended** to name the `Widgets` **accordingly!**

This section of the report will go through the 3 steps of developing `app.py` + 1 optional step to compile `app.exe`, as summarized in the graphics below.




## 5.1 Inheriting `Widgets` from `main_window.py`

The goal of this section is to ensure that `app.py` is **runnable without any error** and shows the **exact same** GUI as if previewing `main_window.ui`.




This result shows that `app.py` has successfully inherited all the properties of `main_window.py`, which includes all the `Widgets` defined when `main_window.ui` was created! These `Widgets` include `updateWindowButton`, `SMA1Checkbox`, `filePathEdit`, etc…

To achieve this, simply start from the generic starter code for all `PyQt5` application and then add the following:

1. Import `matplotlib`, `PyQt5` and the GUI's Widget class called `UI_Form` from `main_window`
2. Pass `QWidget` and `UI_Form` as argument to `Main` class to specify inheritance from `QWidget` and `UI_Form` class
3. Call the superclass' (`UI_Form`) initializing function and setup function
4. Finally, after the inherited GUI has been initialized, it is still possible to add other `Widgets` programmatically as well

This is exactly shown in the code below, running them should result in the image shown above:

[ ]:
```
import sys
from pathlib import Path
from datetime import datetime

```

```
# Step 1
# standard matplotlib import statements
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

# import matplotlib backend for Qt5
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.backends.backend_qt5agg import NavigationToolbar2QT as␣
 ↪NavigationToolbar

# standard PyQt5 import statements
from PyQt5 import QtCore as qtc
from PyQt5 import QtWidgets as qtw

# importing the class to be inherited from
from main_window import Ui_Form

# importing StockData processing module
from stock_data import StockData

class Main(qtw.QWidget, Ui_Form): # Step 2
    def __init__(self):
        # Step 3
        # calling Ui_Form's initializing and setup function
        super().__init__()
        self.setupUi(self)
        self.setWindowTitle("Stock Chart & Moving Average Application")

        # Step 4
        # sets up figure to plot on, instantiates canvas and toolbar
        self.figure, self.ax = plt.subplots()
        self.canvas = FigureCanvas(self.figure)
        self.toolbar = NavigationToolbar(self.canvas, self)

        # attaches the toolbar and canvas to the canvas layout
        self.canvasLayout.addWidget(self.toolbar)
        self.canvasLayout.addWidget(self.canvas)

        # sets up a scroll area to display GUI statuses
        self.scrollWidget = qtw.QWidget()
        self.scrollLayout = qtw.QVBoxLayout()
        self.scrollWidget.setLayout(self.scrollLayout)
        self.scrollArea.setWidget(self.scrollWidget)

    def function(self):
        # define new functions to do each new actions this way
        pass
```

```
if __name__ == "__main__":
    app = qtw.QApplication([])
    main = Main()
    main.show()
    sys.exit(app.exec_())
```

*Learning Point: Inheriting* ***Widgets*** *from* ***main_window.py***

*When* ***main_window.ui*** *is converted into* ***main_window.py*** *using* ***pyuic5****, the* ***Widget*** *class called* ***Ui_Form*** *is created. This* ***Ui_Form*** *class has access to all the* ***Widgets*** *previously defined inside* ***main_window.ui*** *using* ***Qt Designer!*** *They're accessible to* ***Ui_Form*** *as regular python* ***Attributes****. e.g:* ***self.updateWindowButton****, etc… Thus, by inheriting from* ***Ui_Form****,* ***app.py****'s* ***Main*** *class can also access these* ***Widgets*** *through its* ***Attributes****. LIkewise,* ***functions*** *defined in* ***Ui_Form*** *are also inherited and accessible to* ***Main****.*

*Learning Point: Defining & Adding* ***Widgets*** *programmatically*

*Sometimes, it is more convenient to define* ***Widgets*** *programmatically then through* ***Qt Designer****. As shown from the code snippet above, this is also possible and uses the* ***exact same core principles*** *as in* ***main_window.py*** 1. *Defining each* ***Widget*** *objects' and their names within the GUI. Exemplified with lines such as:* ***self.canvas = FigureCanvas(self.figure)*** *or similar instantiation line:* ***button = QPushButton('Button Name', self)*** 2. *Defining the location, size and other physical attributes of each* ***Widgets****. Exemplified with lines such as:* ***self.canvasLayout.addWidget(self.canvas)***

Now that `app.py` is able to access the `Widgets` defined in `main_window.py` by means of Python inheritance. It is now possible to implement `app.py`'s main responsibility:

1. Defining `functions` to accomplish certain actions
2. Connecting `Widget` actions to these `functions`

## 5.2 Defining `functions` in `app.py`

Before defining the `functions` in `app.py`, it is important to first be aware of the scope of each `functions` needed to execute the app's entire process. By referring to the User Manual's 5-step guide, it is possible to breakdown the entire app's functionalities into 3 major functions + 2 minor functions:

1. `load_data(self)` : invoked when `Load CSV File Button` is pressed

   loads stock data .csv from inputted filepath string on the GUI as StockData object, also autocompletes all inputs using information provided by the csv. (Handles the actions from Step 1-2 of User Manual).

2. `update_canvas(self)` : invoked when `Load Update Window Button` is pressed

   creates a datetime object from the inputted date string of format YYYY-MM-DD. uses it to slice a copy of loaded stock_data to be used to update graphics. checks

checkboxes first to see if SMA1, SMA2, Buy and Sell plots need to be drawn. finally, updates graphic accordingly. (Handles the actions from Step 3-5 of User Manual).

3. `plot_graph(self, column_headers, formats)` : invoked when `update_canvas function` is called

   plots graphs specified under column_headers using the formats specified (Helps to handle the action from Step 5 of User Manual).

4. `report(self, string)` : invoked when any of the 3 major functions are called

   given a report (string), update the scroll area with this report

5. `center(self)` : invoked when `__init__(self)` is called (i.e. during the startup of app)

   centers the fixed main window size according to user screen size

The following part of the report will attempt to explain each of these 5 functions in detail. However, due to space limitation and the need for conciseness, only **parts of the code with its line number will be referenced!** We highly recommend that readers refer to the **full code in the Appendix or the python file itself should it become necessary**.

#### 5.2.1 `load_data(self)`

First, this function attempts to parse the `text` specified by user in the `Line Edit Widget` called `filePathEdit`for a `filepath` .

```
102 filepath = Path(self.filePathEdit.text())
```

*Learning Point: Getting `Line Edit Widget` Value*

*To extract the `string` value from `Line Edit Widget`, use: `.text()` method*

The parsing of this `filepath` is outsourced to Python's `pathlib` library.

*Learning Point: Using `Path` from `pathlib` to parse `filepath`*

*To parse the `filepath` from `string`, simply use the standard python `pathlib`. Instantiate a `Path` object by passing the `string` as follows: `Path(string)`. This guarantees that the resultant `filepath` follows the proper format that the computer OS uses.*

Next, it will attempt to instantiate a `StockData` data object using this `filepath`. However, to prevent crashes due to invalid `filepath` or `.csv` file, it is important to wrap the previous instantiation line with a `try... except....`

```
104 try:
105     self.stock_data = StockData(filepath)
...
121 except IOError as e:
122     self.report(f"Filepath provided is invalid or fail to open .csv file. {e}")
123
124 except TypeError as e:
125     self.report(f"The return tuple is probably (nan, nan) because .csv is empty")
```

Each of this `except` corresponds to the the errors mentioned in the function's docstring line 96 to 100 (see Appendix).

*Learning Point: Preventing Crashes with `try... except...`*

*To prevent crashes, simply encapsulate the line inside a `try... except...`. Each type of error can then be handled individually.*

Once `StockData` has been initialized, the function attempts to get the `start_date` and `end_date` of the `stock_data` by `StockData`'s method called `get_period()`.

```
106     start_date, end_date = self.stock_data.get_period()
107     period = f"{start_date} to {end_date}"
```

Finally, the function will attempt to 'auto-complete' the various `Widgets` using information such as the `start_date` and `end_date`.

```
109     # auto-complete feauture
110     self.startDateEdit.setText(start_date)
111     self.endDateEdit.setText(end_date)
112     self.periodEdit.setText(period)
113     self.SMA1Edit.setText("15")
114     self.SMA2Edit.setText("50")
115     self.SMA1Checkbox.setChecked(False)
116     self.SMA2Checkbox.setChecked(False)
```

*Learning Point: Setting `Widget` Values Programmatically.*

*To set values to `Widgets` there are various methods specific to each type of `Widget`. `Line Edit Widget` uses `.setText(string)` whereas `Checkbox Widget` uses `.setChecked(bool)`.*

### 5.2.2 `update_canvas(self)`

Similar to `load_data(self)`, this function begins by parsing an input. This time, the input is read from `startDateEdit` and `endDateEdit`. While `load_data(self)` attempts to parse `filepath`, `update_canvas(self)` is attempting to read `datetime`. Hence, python's standard `datetime` library is used:

```
150 try:
151     start_date = str(datetime.strptime(self.startDateEdit.text(), self.date_format).date()
152     end_date = str(datetime.strptime(self.endDateEdit.text(), self.date_format).date())
```

To convert a datetime `string` into a `datetime` object, the method `datetime.strptime(string, format)` can be used. However, it requires that the specified `string` follows a certain format, the chosen format is: YYYY-MM-DD, represented by:

```
148 self.date_format = '%Y-%m-%d'
```

Similar to `load_data(self)`, these functions are encapsulated inside a `try... except...` to prevent crashes and catch errors.

More detailed information about this `datetime` package can be found in the "Python Packages" section.

*Learning Point: Parsing date `string` using `datetime`*

*To parse a `datetime` string into a `datetime object`, use the `datetime.strptime(string, format)` method. This method requires that the `string` specified follows a `format`. For YYYY-MM-DD, its `format` is represented as: `%Y-%n-%d`. Then finally, to return a `datetime object` in a certain format, simply use the objetc's method. In the application, `.date()` is used to return the `datetime object` with a YYYY-MM-DD format.*

Unlike `load_data(self)` that attempts to simply process the entire `StockData`, the goal of `update_canvas` is to:

1. Determine a `range` of of data to be plotted
2. Determine what `columns` of data to be plotted

The first goal is simple as the function has already parsed the `start_date` and `end_date strings` from their respective `Line Edit Widgets` using `datetime` package mentioned previously. All that is left is to call the `StockData`'s method that has been written to return a copy of the `DataFrame` for the specified `range` of data.

```
175     self.selected_stock_data = self.stock_data.get_data(start_date, end_date)
```

The second goal is a little more complex. The function needs to build a `list` of `column_headers` by checking whether or not the two SMA `Checkbox Widgets` are 'ticked' using the method `Checkbox.isChecked()`.

There are in total 3 different possibilities:

1. No Checkbox is ticked. Then, only the stock price under the `Close` header needs to be plotted. This means by default, the `Close` stock price data will always be plotted. Hence, the `column_headers list` is always instantiated with this value inside:

```
156     # builds a list of graphs to plot by checking the tickboxes
157     column_headers = ['Close']
```

2. Only 1 of the SMA Checkbox is ticked. Then, it is only necessary to calculate 1 SMA using the `StockData` method `_calculate_SMA(int)`, and append 1 `column_head string` into the `column_headers list`. Thus, we check for this condition using 2 `if` clauses, 1 for each SMA `Checkbox Widget`resulting in a `column_headers list` of length 2:

```
160     if self.SMA1Checkbox.isChecked():
161         self.stock_data._calculate_SMA(int(self.SMA1Edit.text()))
162         column_headers.append(f"SMA{self.SMA1Edit.text()}")
...
164     if self.SMA2Checkbox.isChecked():
165         self.stock_data._calculate_SMA(int(self.SMA2Edit.text()))
166         column_headers.append(f"SMA{self.SMA2Edit.text()}")
```

3. Both of the SMA Checkboxes are ticked. Then, 2 SMAs must be calculated and 2 `column_head string` must be appended. However, on top of these, SMA crossover data can now be calculated using the 2 SMA data with `_calculate_crossover(SMA1, SMA2, value)` resulting in 2 additional columns of signal data to be plotted called: `Buy` and `Sell`. This results in a `column_headers list` of length 5. We check for this condition by checking if the length of `column_headers list` is 3:

```
168     if len(column_headers) == 3:
```

```
169         self.stock_data._calculate_crossover(column_headers[1], column_headers[2], column_
170         column_headers.append('Sell')
171         formats.append('rv')
172         column_headers.append('Buy')
173         formats.append('g^')
```

Finally, we can then plot these datapoints found in the `column headers` according to specific `formats` by calling:

```
176     self.plot_graph(column_headers, formats)
```

The `formats` is also a list of `string` that tells `matplotlib` of the **marker type and color** of the different data plots. The process of building the `formats list` is exactly the same as `column_headers list`, and therefore, the length of the two lists **must always be the same** by the time line 176 is called.

> *Learning Point: Getting `Checkbox Widget` Value*
>
> *While `Line Edit Widget` uses the method `.text()` to get its `string` value. `Checkbox Widget` uses `.isChecked()` to get its current value which returns `boolean: True` or `False` depending whether the it is 'ticked' or not.*
>
> *Learning Point: `matplotlib` plot format `strings`*
>
> *Format `strings` inform `matplotlib` of both **color and type** of plot. Some common ones include: `k-`, where `k` tells `matplotlib` to color the plot black and the `-` tells `matplotlib` to plot the data as line graph. `ro` tells `matplolib` to plot the data red and as scatter plot. Finally, `g^` tells `matplotlib` to use the green color and upper triangle for the scatter plot's marker instead of a dot which the previous `o` command specifies.*

### 5.2.3 `plot_graph(self, column_headers, formats)`

This function implements the standard `matplotlib`'s method of plotting datapoints into an `Axes`.

First ensure that the `Axes` to plot on is cleared before a new plot is drawn by calling:

```
210 self.ax.clear()
```

This is to prevent multiple plots being plotted on the same `Axes` when the `Update Window Button` is pressed multiple times.

Next, prevent any crashing due to empty dataframe by using `assert` statement to raise error when such occasions do happen, for example: the user selects a start and end date containing no data points.

```
211 assert not self.selected_stock_data.empty
```

> *Learning Point: Clearing `Axes`*
>
> *`Axes` is the plot area in which the datapoints are plotted. It is important to clear this area, otherwise multiple plots will be plotted in it. To clear it use the `.clear()` method.*
>
> *Learning Point: Preventing Crashes with `assert`*

*The `assert` keyword tests if a condition is true. If it is **NOT**, the program will raise an `AssertionError`. which can then be handled. This can be used to prevent crashes, in combination with `try... except` mentioned previously.*

Only after doing these checks, do we implement the plotting method which is simply just:

```
223         self.ax.plot(x_data, y_data, formats[i], label=column_headers[i])
```

This is the standard `matplotlib` function to use to plot any X-Y datas in an `Axes`.

For the `x_data`, we have the `list` containing `dates` of each prices. However, specifically for a time-series `x_data`, `matplotlib` does not accept `string` or `datetime` objects. Instead it has its own internal way of representing `datetime`. As such, it is mandatory to convert `datetime` objects into this internal representation with `mdates.date2num(datetime_list)`.

```
213 # matplotlib has its own internal representation of datetime
214 # date2num converts datetime.datetime to this internal representation
215 x_data = list(mdates.date2num(
216                               [datetime.strptime(dates, self.date_format).date()
217                               for dates in self.selected_stock_data.index.values]
218                               ))
```

For the `y_data`, we can use anything as it is a simple stock price values. In this case, it is just a `list`. Furthermore, if we want to plot multiple datasets in the same `Axes`, we can simply call the method in line 223 mutiple times with different `y_data`. For example, we use loops to call `ax.plot()` on each `y_data` dataset of every `column_headers`:

```
220 for i in range(len(column_headers)):
221     if column_headers[i] in self.selected_stock_data.columns:
222         y_data = list(self.selected_stock_data[column_headers[i]])
223         self.ax.plot(x_data, y_data, formats[i], label=column_headers[i])``
```

*Learning Point: The "Standard Way" of Plotting Using `matplotlib`*

*The standard method of plotting using `matplotlib` is to use the method: `ax.plot(x_data, y_data)`.*

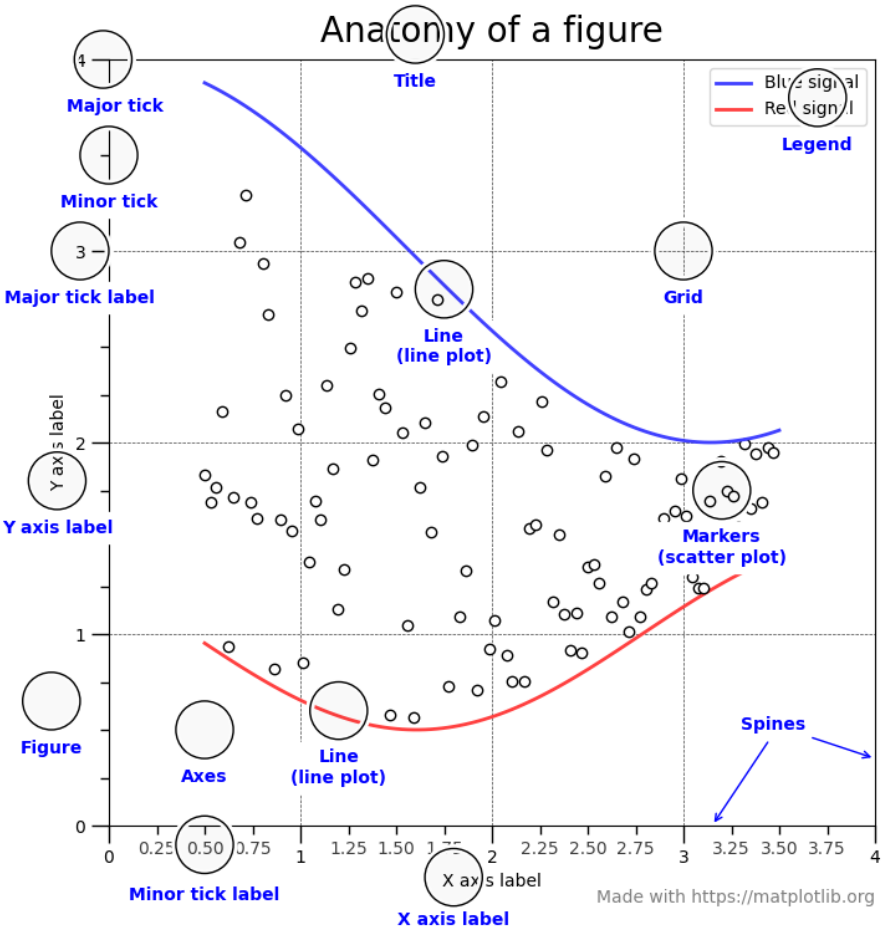
Once the plots are drawn, there may be some formatting that needs to be done on how either the `Axes` or the `Figure` looks like:

```
227 # formatting
228 months_locator = mdates.MonthLocator()
229 months_format = mdates.DateFormatter('%b %Y')
230 self.ax.xaxis.set_major_locator(months_locator)
231 self.ax.xaxis.set_major_formatter(months_format)
232 self.ax.format_xdata = mdates.DateFormatter(self.date_format)
233 self.ax.format_ydata = lambda y: '$%1.2f' % y
234 self.ax.grid(True)
235 self.figure.autofmt_xdate()
236 self.figure.legend()
237 self.figure.tight_layout()
238 self.canvas.draw()
```

Line 238 is important as it tells the GUI to redraw the plot itself with the new formatting!

There are many components that are editable to make a plot looks just right! Thus, it is important to know what is in fact editable by understanding the parts of a `Figure`.

*Learning Point: Anatomy `matplotlib`'s `Figure`*




*One important thing to note is that, the `Figure` encompasses the `Axes` and other things like the `legend`, `layout`, `title`, etc... Whereas the `Axes` of a `Figure` is just the area where the data are plotted! There can be multiple `Axes` to a single `Figure` but not the reverse!*

An alternative to this method is to simply call `Dataframe.plot(column_headers, formats)` on the `Dataframe` containing the selected data. However, this method requires that the format of the `x_data` is already in correct (in this case: `mdates`). Otherwise it will result in an inaccurate/missing `x_data ticks`. As shown here:


Which is why, using the standard method with `ax.plot()`, is recommended and chosen for this application as it guarantees a correct plot as the data are **explicitly** handled.

### 5.2.4 `report(self, string)`

This is a simple function to replicate the act of printing statements to terminal to check on the current progress of the code. It is not necessary to have this statement if the user is running the app using python. However, it is necessary to have it if the user runs the `.exe` file instead, because there is no terminal to see the progress of the app.

```
248 report_text = qtw.QLabel(string)
249 self.scrollLayout.addWidget(report_text)
250 print(string)
```

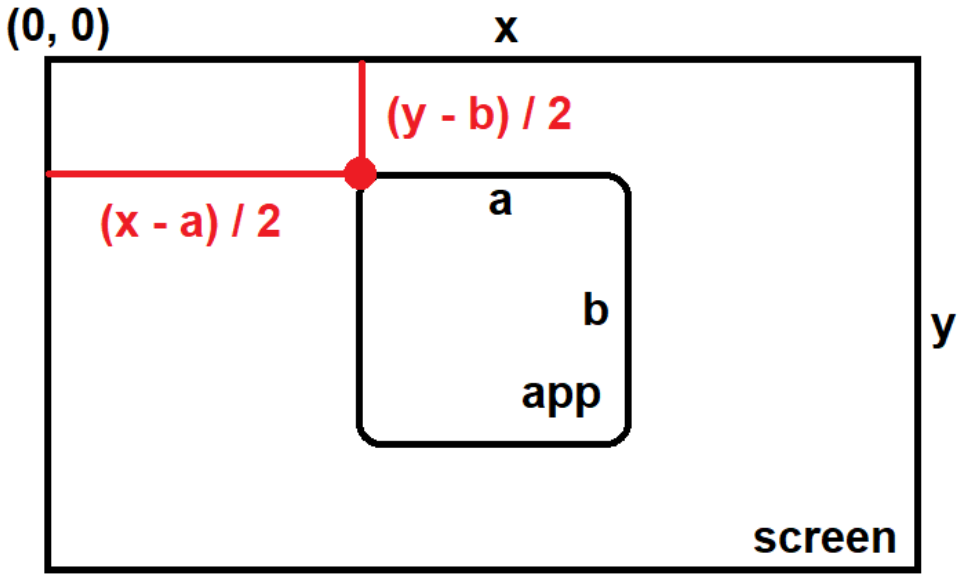
To simulate `print` statements, simply add new `Label Widget` with the `string` statement as its value. This is attached to a `Layout` that can be scrolled.

### 5.2.5 `center(self)`

This method is called to programmatically center the main window of the app according to the screen size of the user's computer. First, the `screen` and app's `main_window` geometries are acquired.

```
256 screen = qtw.QDesktopWidget().screenGeometry()
257 main_window = self.geometry()
```

Using the `width()` and `height()` methods, the values of the width and height of the two geometries can be acquired, and be used to calculate the center pixel. The following diagram illustrates this:




As such, we have the following `x` and `y` coordinates to move towards, using: `.move(x, y)` method.

```
258 x = (screen.width() - main_window.width()) / 2
259
260 # pulls the window up slightly (arbitrary)
261 y = (screen.height() - main_window.height()) / 2 - 50
262 self.setFixedSize(main_window.width(), main_window.height())
263 self.move(x, y)
```

Note: top-left corner is the zero coordinate. Hence, `- 50` pixel will pull the app's window up slightly.

### 5.3 Connecting `Widget` actions to `functions`

Fortunately, connecting `Widget` actions to `functions`are much simpler than defining the `functions`. These are all done inside the `__init__(self)` function. i.e. The app will attempt to connect these functions when it is first initialized/started by the user.

The method used to connect `Widgets` to `functions` is: `Widget.connect(function)`

Simply add the following code to the the starter code given in section: "Inheriting `Widgets` from `main_window.py`" to complete `app.py`.

```
__ init __(self)
```

```
...
81 # button & checkbox connections
82 self.loadCSVButton.clicked.connect(self.load_data)
83 self.updateWindowButton.clicked.connect(self.update_canvas)
84 self.SMA1Checkbox.stateChanged.connect(self.update_canvas)
85 self.SMA2Checkbox.stateChanged.connect(self.update_canvas)
86
87 # auto-complete feauture
88 self.filePathEdit.setText("../data/GOOG.csv")
```

> *Learning Point: Connecting `Widgets` to `functions`*
>
> *To connect `Widgets` to `functions` use the following method: `Widget.connect(function)`. This ensures that when users interact with the `Widget` e.g. by pressing `Button`, checking `Checkbox`, etc…, it will trigger the appropriate `functions`*

## 5.4 (Optional) Compiling `app.exe`

To compile `app.py` application into an executable, first install `pysinstaller` using `PIP` by running the following command:

```
pip install pyinstaller
```

Having installed `pyinstaller`, then use the following command from `root` folder:

```
pyinstaller .\src\app.py -F
```

The `app.exe` file can be found inside the `dist` folder.

Note: the above command assumes that all source code (such as `app.py`, `stock_data.py` and `main_window.py`) are all found inside the `src` folder!

`app.exe` is a binary executable file for Windows (not Mac!). It allows users to simply double-click this file to start the application without requiring installation of any python modules at all.

> *Learning Point: Compiling Python Modules into an `.exe`*
>
> *PyInstaller is a standard package to bundle a Python application and all of its dependencies into a single executable. The user can then run the packaged app without installing a Python interpreter or any modules. However, this is only possible for Windows!*

# 6 Appendix

## 6.1 Code Reference

[1]:
```
import sys
sys.path.insert(1, '../src')

from app import Main
from stock_data import StockData
import inspect # standard library used later to get info about the source code

def print_code(code): # prints '{line} {code}' with 2 less indent and without␣
↪the def header
    codeline = lambda code, start : [(start + 1 + i, code[i]) for i in␣
↪range(len(code))]
    print("".join([f"{line} {text[2:]}" if len(text) > 1 else f"{line} {text}"␣
↪for line, text in codeline(code[0][1:], code[1])]))
```

### 6.1.1 app.py

**___ init ___(self)**

[2]:
```
print_code(inspect.getsourcelines(Main.__init__))
```

```
59 """
60 initializes and sets up GUI widgets and its connections
61 """
62 super().__init__()
63 self.setupUi(self)
64 self.setWindowTitle("Stock Chart & Moving Average Application")
65
66 # sets up figure to plot on, instantiates canvas and toolbar
67 self.figure, self.ax = plt.subplots()
68 self.canvas = FigureCanvas(self.figure)
69 self.toolbar = NavigationToolbar(self.canvas, self)
70
71 # attaches the toolbar and canvas to the canvas layout
72 self.canvasLayout.addWidget(self.toolbar)
73 self.canvasLayout.addWidget(self.canvas)
74
75 # sets up a scroll area to display GUI statuses
76 self.scrollWidget = qtw.QWidget()
77 self.scrollLayout = qtw.QVBoxLayout()
78 self.scrollWidget.setLayout(self.scrollLayout)
79 self.scrollArea.setWidget(self.scrollWidget)
80
81 # button & checkbox connections
82 self.loadCSVButton.clicked.connect(self.load_data)
83 self.updateWindowButton.clicked.connect(self.update_canvas)
```

```
84 self.SMA1Checkbox.stateChanged.connect(self.update_canvas)
85 self.SMA2Checkbox.stateChanged.connect(self.update_canvas)
86
87 # auto-complete feauture
88 self.filePathEdit.setText("../data/GOOG.csv")
```

**load_data(self)**

[3]:
```
print_code(inspect.getsourcelines(Main.load_data))
```

```
91 """
92 loads stock data .csv from inputted filepath string on the GUI
93 as StockData object, also autocompletes all inputs
94 using information provided by the csv.
95
96 Error handling
97      invalid filepath :
98              empty filepath or file could not be found.
99      invalid .csv :
100             .csv file is empty, missing date column, etc.
101 """
102 filepath = Path(self.filePathEdit.text())
103
104 try:
105     self.stock_data = StockData(filepath)
106     start_date, end_date = self.stock_data.get_period()
107     period = f"{start_date} to {end_date}"
108
109     # auto-complete feauture
110     self.startDateEdit.setText(start_date)
111     self.endDateEdit.setText(end_date)
112     self.periodEdit.setText(period)
113     self.SMA1Edit.setText("15")
114     self.SMA2Edit.setText("50")
115     self.SMA1Checkbox.setChecked(False)
116     self.SMA2Checkbox.setChecked(False)
117
118     self.report(f"Data loaded from {filepath}; period auto-selected:
{start_date} to {end_date}.")
119     print(self.stock_data.data)
120
121 except IOError as e:
122     self.report(f"Filepath provided is invalid or fail to open .csv file.
{e}")
123
124 except TypeError as e:
125     self.report(f"The return tuple is probably (nan, nan) because .csv is
```

```
empty")
```

**update_canvas(self)**

[4]:
```
print_code(inspect.getsourcelines(Main.update_canvas))
```

```
128 """
129 creates a datetime object from the inputted date string
130 of format YYYY-MM-DD. uses it to slice a copy of loaded
131 stock_data to be used to update graphics. checks
132 checkboxes first to see if SMA1, SMA2, Buy and Sell plots
133 need to be drawn. finally, updates graphic accordingly.
134
135 Error handling
136 invalid date format:
137     date format inside the .csv file is not YYYY-MM-DD
138 non-existent stock_data :
139     the selected range results in an empty dataframe
140     or end date < start date
141 non-existent data point :
142     data of that date does not exist,
143     or maybe because it is Out-Of-Bound
144 raised exceptions :
145     SMA1 and SMA2 values are the same,
146     or other exceptions raised
147 """
148 self.date_format = '%Y-%m-%d'
149
150 try:
151     start_date = str(datetime.strptime(self.startDateEdit.text(),
self.date_format).date())
152     end_date = str(datetime.strptime(self.endDateEdit.text(),
self.date_format).date())
153     period = f"{start_date} to {end_date}"
154     self.periodEdit.setText(period)
155
156     # builds a list of graphs to plot by checking the tickboxes
157     column_headers = ['Close']
158     formats = ['k-']
159
160     if self.SMA1Checkbox.isChecked():
161             self.stock_data._calculate_SMA(int(self.SMA1Edit.text()))
162             column_headers.append(f"SMA{self.SMA1Edit.text()}")
163             formats.append('b-')
164     if self.SMA2Checkbox.isChecked():
165             self.stock_data._calculate_SMA(int(self.SMA2Edit.text()))
166             column_headers.append(f"SMA{self.SMA2Edit.text()}")
```

```
167             formats.append('m-')
168     if len(column_headers) == 3:
169             self.stock_data._calculate_crossover(column_headers[1],
column_headers[2], column_headers[1])
170             column_headers.append('Sell')
171             formats.append('rv')
172             column_headers.append('Buy')
173             formats.append('g^')
174
175     self.selected_stock_data = self.stock_data.get_data(start_date,
end_date)
176     self.plot_graph(column_headers, formats)
177
178     self.report(f"Plotting {column_headers} data from period: {start_date}
to {end_date}.")
179     print(self.selected_stock_data)
180
181 except ValueError as e:
182     self.report(f"Time period has not been specified or does not match YYYY-
MM-DD format, {e}.")
183
184 except AssertionError as e:
185     self.report(f"Selected range is empty, {e}")
186
187 except KeyError as e:
188     self.report(f"Data for this date does not exist: {e}")
189
190 except Exception as e:
191     self.report(f"Exception encountered: {e}")
```

**plot_graph(self, column_headers, formats)**

[5]:
```
print_code(inspect.getsourcelines(Main.plot_graph))
```

```
194 """
195 plots graphs specified under column_headers using the formats
196
197 Parameters
198 column_headers : [str, str, …]
199     a list containing column header names with data to be plotted
200 formats : [str, str, …]
201     a list of matplotlib built-in style strings to indicate
202     whether to plot line or scatterplot and the colours
203     corresponding to each value in col_headers
204     (hence, must be same length)
205
206 Error handling
```

```
207 empty dataframe :
208     selected dataframe is empty
209 """
210 self.ax.clear()
211 assert not self.selected_stock_data.empty
212
213 # matplotlib has its own internal representation of datetime
214 # date2num converts datetime.datetime to this internal representation
215 x_data = list(mdates.date2num(
216                                 [datetime.strptime(dates,
self.date_format).date()
217                                 for dates in
self.selected_stock_data.index.values]
218                                 ))
219
220 for i in range(len(column_headers)):
221     if column_headers[i] in self.selected_stock_data.columns:
222             y_data = list(self.selected_stock_data[column_headers[i]])
223             self.ax.plot(x_data, y_data, formats[i],
label=column_headers[i])
224             self.report(f"{column_headers[i]} data is being plotted.")
225     else: self.report(f"{column_headers[i]} data does not exist.")
226
227 # formatting
228 months_locator = mdates.MonthLocator()
229 months_format = mdates.DateFormatter('%b %Y')
230 self.ax.xaxis.set_major_locator(months_locator)
231 self.ax.xaxis.set_major_formatter(months_format)
232 self.ax.format_xdata = mdates.DateFormatter(self.date_format)
233 self.ax.format_ydata = lambda y: '$%1.2f' % y
234 self.ax.grid(True)
235 self.figure.autofmt_xdate()
236 self.figure.legend()
237 self.figure.tight_layout()
238 self.canvas.draw()
```

**report(self, string)**

[6]:
```
print_code(inspect.getsourcelines(Main.report))
```

```
241 """
242 given a report (string), update the scroll area with this report
243
244 Parameters
245 string : str
246     string of the report, usually the error message itself.
247 """
```

```
248 report_text = qtw.QLabel(string)
249 self.scrollLayout.addWidget(report_text)
250 print(string)
```

**center(self)**

[7]:
```
print_code(inspect.getsourcelines(Main.center))
```

```
253 """
254 centers the fixed main window size according to user screen size
255 """
256 screen = qtw.QDesktopWidget().screenGeometry()
257 main_window = self.geometry()
258 x = (screen.width() - main_window.width()) / 2
259
260 # pulls the window up slightly (arbitrary)
261 y = (screen.height() - main_window.height()) / 2 - 50
262 self.setFixedSize(main_window.width(), main_window.height())
263 self.move(x, y)
```

### 6.1.2 stock_data.py

**___ init ___(self)**

[8]:
```
print_code(inspect.getsourcelines(StockData.__init__))
```

```
18 """
19 initializes StockData object by parsing stock data .csv file into a dataframe
20 (assumes 'Date' column exists and uses it for index),
21 also checks and handles missing data
22
23 Parameters
24 filepath : str
25      filepath to the stock data .csv file, can be relative or absolute
26
27 Raises
28 IOError :
29      failed I/O operation, e.g: invalid filepath, fail to open .csv
30 """
31 self.filepath = filepath
32 self.data = pd.read_csv(filepath).set_index('Date')
33 self.check_data()
```

**check_data(self, overwrite=True)**

[9]:
```
print_code(inspect.getsourcelines(StockData.check_data))
```

```
36 """
37 checks and handles missing data by filling in missing values by interpolation
38
39 Parameters
40 overwrite : bool (True)
41      if True, overwrites original source stock data .csv file
42
43 Returns
44 self : StockData
45 """
46 # function to fill in missing values
47 # by averaging previous data and after (interpolation)
48 self.data = self.data.interpolate()
49 self.data.to_csv(self.filepath, index=overwrite)
50 return self
```

**get_data(self, start_date, end_date)**

[10]:
```
print_code(inspect.getsourcelines(StockData.get_data))
```

```
53 """
54 returns a subset of the stock data from start_date to end_date inclusive
55
56 Parameters
57 start_date : str
58      start date of stock data range, must be of format YYYY-MM-DD
59 end_date : str
60      end date of stokc data range, must be of format YYYY-MM-DD
61
62 Returns:
63 selected_data : DataFrame
64      stock data dataframe indexed from specified start to end date inclusive
65
66 Raises
67 KeyError :
68      data for this date does not exist
69 AssertionError :
70      selected range is empty
71 """
72 self.selected_data = self.data[str(start_date):str(end_date)]
73 return self.selected_data
```

**get_period(self)**

[11]:
```
print_code(inspect.getsourcelines(StockData.get_period))
```

```
76 """
```

```
77 returns a string tuple of the first and last index
78 which make up the maximum period of StockData
79
80 Returns
81 period : (str, str)
82
83 Raises
84 TypeError :
85      the return tuple is probably (nan, nan) because .csv is empty
86 """
87 index = list(self.data.index)
88 (first, last) = (index[0], index[-1])
89 return (first, last)
```

**_calculate_SMA(self, n, col='Close')**

[12]:
```
print_code(inspect.getsourcelines(StockData._calculate_SMA))
```

```
92 """
93 calculates simple moving average (SMA) and augments the stock dataframe
94 with this SMA(n) data as a new column
95
96 Parameters
97 n : int
98      the amount of stock data to use to calculate average
99 col : str ('Close')
100     the column head title of the values to use to calculate average
101
102 Returns
103 self : StockData
104 """
105 col_head = f'SMA{n}'
106 if col_head not in self.data.columns:
107     sma = self.data[col].rolling(n).mean()
108     self.data[f'SMA{n}'] = np.round(sma, 4)
109     self.data.to_csv(self.filepath, index=True)
110 return self
```

**_calculate_crossover(self, SMA1, SMA2, col='Close')**

[13]:
```
print_code(inspect.getsourcelines(StockData._calculate_crossover))
```

```
113 """
114 calculates the crossover positions and values,
115 augments the stock dataframe with 2 new columns
116 'Sell' and 'Buy' containing the value at which SMA crossover happens
117
```

```
118 Parameters
119 SMA1 : str
120     the first column head title containing the SMA values
121 SMA2 : str
122     the second column head title containing the SMA values
123 col : str ('Close')
124     the column head title whose values will copied into 'Buy' and 'Sell'
125     columns to indicate crossovers had happen on that index
126
127 Returns
128 self : StockData
129
130 Raises
131 Exception :
132     SMA1 and SMA2 provided are the same, they must be different
133 """
134 if SMA1 < SMA2: signal = self.data[SMA1] - self.data[SMA2]
135 elif SMA1 > SMA2: signal = self.data[SMA2] - self.data[SMA1]
136 else: raise Exception(f"{SMA1} & {SMA2} provided are the same. They must be
different SMA.")
137
138 signal[signal > 0] = 1
139 signal[signal <= 0] = 0
140 diff = signal.diff()
141
142 self.data['Sell'] = np.nan
143 self.data['Buy'] = np.nan
144 self.data.loc[diff.index[diff < 0], 'Sell'] = self.data.loc[diff.index[diff
< 0], col]
145 self.data.loc[diff.index[diff > 0], 'Buy'] = self.data.loc[diff.index[diff >
0], col]
146
147 self.data.to_csv(self.filepath, index=True)
148 return self
```

**plot_graph(self, col_headers, style, ax, show=True)**

[14]:
```
print_code(inspect.getsourcelines(StockData.plot_graph))
```

```
151 """
152 plots columns of selected values as line plot and/or columns of values
153 as scatter plot as specified by style to an Axes object
154
155 Parameters
156 col_headers : [str, str, …]
157     a list containing column header names whose data are to be plotted
158 style : [str, str, …]
```

```
    a list of matplotlib built-in style strings to indicate whether to plot
    line or scatterplot and the colours corresponding to each value in
    col_headers (hence, must be same length)
ax : Axes
    matplotlib axes object on which the plot will be drawn

Raises
AttributeError :
    self.selected_data has not been specified,
    call StockData.get_data(start, end) before plotting
AssertionError :
    self.selected_data is empty, perhaps due to OOB or invalid range
"""
assert not self.selected_data.empty
self.selected_data[col_headers].plot(style=style,
                                     ax=ax,
                                     grid=True,
                                     x_compat=True,
                                     linewidth=1)
if show: plt.show()
```

**calculate_SMA(self, n)**

[15]:
```
print_code(inspect.getsourcelines(StockData.calculate_SMA))
```

```
"""
calculates simple moving average (SMA) and augments the stock dataframe
with this SMA(n) data as a new column

Parameters
n : int
    the amount of stock data to use to calculate average
col : str ('Close')
    the column head title of the values to use to calculate average

Returns
self : StockData
"""
col_head = 'SMA' + str(n)
df = self.data.reset_index()

if col_head not in df.columns:
    # Extract full dataframe from the actual data
    # (to check if there is enough data for sma)
    dateList = self.data.index.values.tolist()
    returnList = []
    for date in dateList: # for date in dateList
```

```
203     # find the index of date in the full data
204             dateIndex = df[df["Date"]==date].index.values[0]
205             if dateIndex < n: # if date index is less than n: append None
206                     returnList.append(np.nan)
207             else:
208                     sum = 0
209                     for i in range(n):
210                             sum += df.iloc[dateIndex-i]["Close"]
211                     # append the SMA for each day to a list
212                     returnList.append(sum/n)
213
214     self.data[col_head] = returnList
215     print(self.data)
216     self.data.to_csv(self.filepath, index=True)
217
218 return self
```

**calculate_crossover(self, SMAa, SMAb)**

[16]:
```
print_code(inspect.getsourcelines(StockData.calculate_crossover))
```

```
221 """
222 calculates the crossover positions and values,
223 augments the stock dataframe with 2 new columns
224 'Sell' and 'Buy' containing the value at which SMA crossover happens
225
226 Parameters
227 SMA1 : str
228     the first column head title containing the SMA values
229 SMA2 : str
230     the second column head title containing the SMA values
231 col : str ('Close')
232     the column head title whose values will copied into 'Buy' and 'Sell'
233     columns to indicate crossovers had happen on that index
234
235 Returns
236 self : StockData
237
238 Raises
239 Exception :
240     SMA1 and SMA2 provided are the same, they must be different
241 """
242 col_head1 = 'Position'
243 col_head2 = 'Signal'
244 col_head3 = 'Buy'
245 col_head4 = 'Sell'
246 df = self.data
```

```
247
248 # to ensure the correct number of elements in the loop
249 SMAlist = self.data.index.values.tolist()
250 # extracts the SMA from the specific column in self.data
251 if SMAa < SMAb:
252     SMA1 = df[SMAa].tolist()
253     SMA2 = df[SMAb].tolist()
254 elif SMAa > SMAb:
255     SMA1 = df[SMAb].tolist()
256     SMA2 = df[SMAa].tolist()
257 else: # SMAa == SMAb
258     raise ValueError(f"Given {SMAa} & {SMAb} are the same. Must be different
SMA.")
259
260 stockPosition = []  # which SMA line is on top
261 stockSignal = []    # the buy/sell signal --> the 1s and -1s
262 buySignal = []              # filtered out location of buy signals
263 sellSignal = []             # filtered out location of sell signals
264
265 # goes through every element
266 for i in range(len(SMAlist)):
267     if SMA1[i] > SMA2[i]: stockPosition.append(1)   # SMA1 above SMA2
268     elif SMA1[i] < SMA2[i]: stockPosition.append(0) # SMA2 above SMA1
269     # if the SMAs are equal, repeat the previous entry
270     # because no crossover has occured yet
271     elif SMA1[i] == SMA2[i]: stockPosition.append(stockPosition[i-1])
272     else: stockPosition.append(np.nan) # if no data, leave blank
273
274 # find the places where crossover occurs
275 for j in range(len(stockPosition)):
276     # 'shifts' the data one period to the right
277     # to ensure crossovers are reflected on the correct date
278     if j == 0: stockSignal.append(np.nan)
279     # calculation for the crossover signals
280     else: stockSignal.append(stockPosition[j] - stockPosition[j-1])
281
282
283 for k in range(len(stockSignal)): # finding location of buy signals
284     if stockSignal[k] == 1:
285             value = self.data[SMAa].tolist()[k]
286             buySignal.append(value)
287     else: buySignal.append(np.nan) # if no signal leave blank
288
289 for k in range(len(stockSignal)): # finding location of sell signals
290     if stockSignal[k] == -1:
291             value = self.data[SMAa].tolist()[k]
292             sellSignal.append(value)
293     else: sellSignal.append(np.nan) # if no signal leave blank
```

```
294
295 self.data[col_head3] = buySignal
296 self.data[col_head4] = sellSignal
297
298 print(self.data)
299 self.data.to_csv(self.filepath, index=True)
300 return self
```