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

November 13, 2020

1 `main_window.py`

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

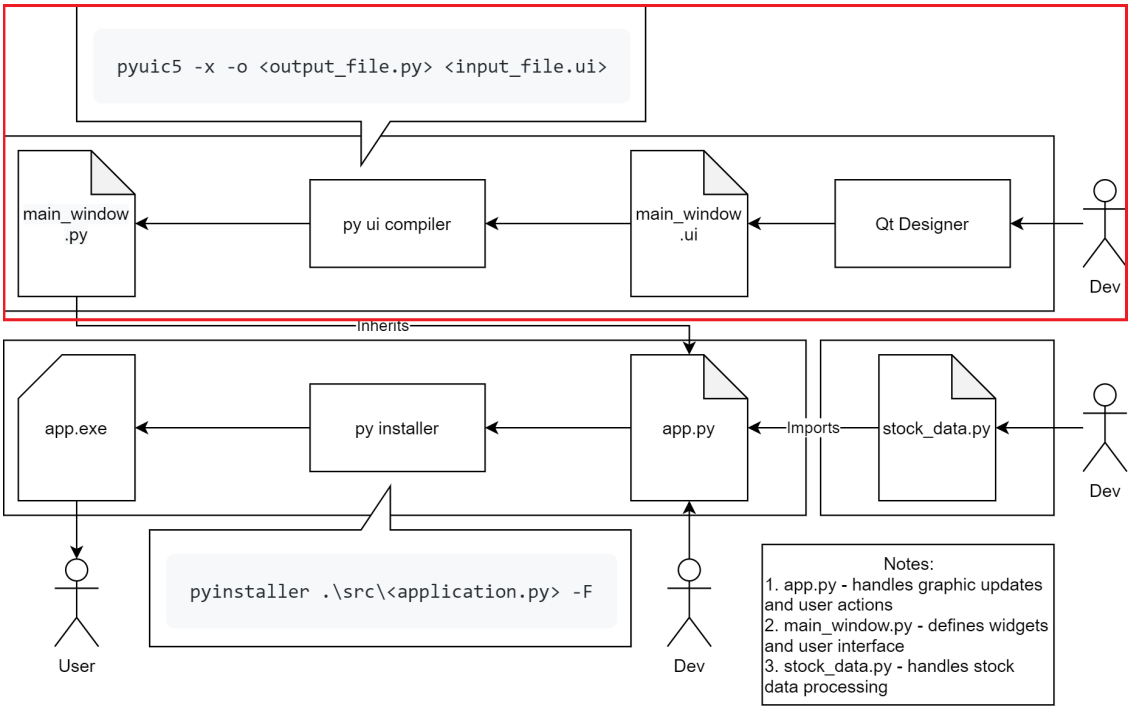
1. Defining each `Widget` objects' and their names within the GUI
2. Defining the location, size and other physical attributes of each `Widgets`

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.

1.1 Installing `Qt Designer`

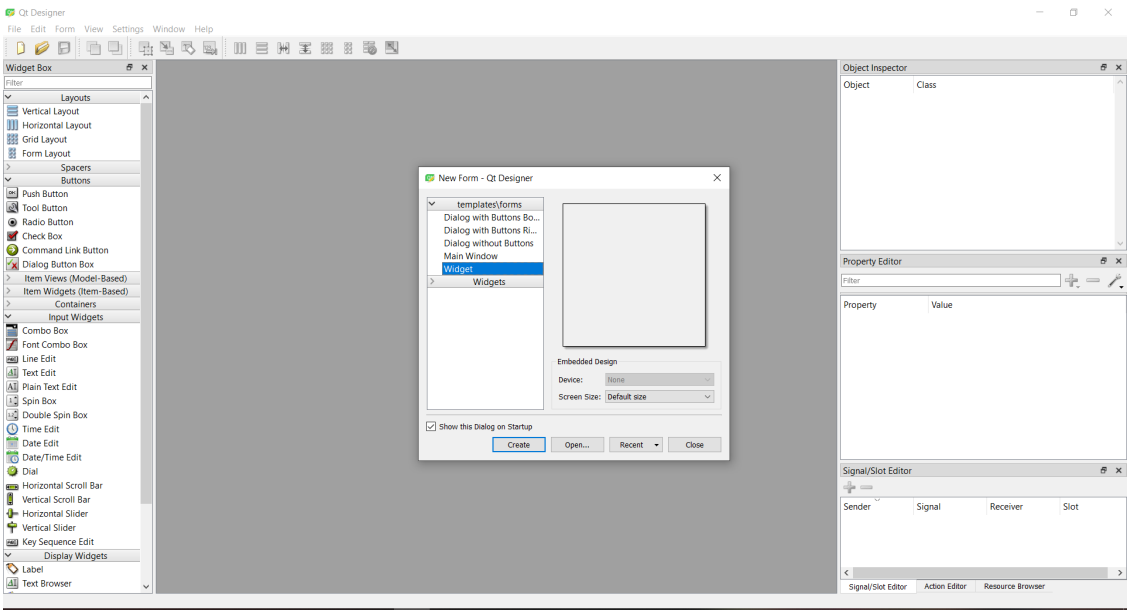
The installation process of `Qt Designer` is similar to any other application.

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

1.2 Building `main_window.ui` with `Qt Designer`

1.2.1 Defining the GUI

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



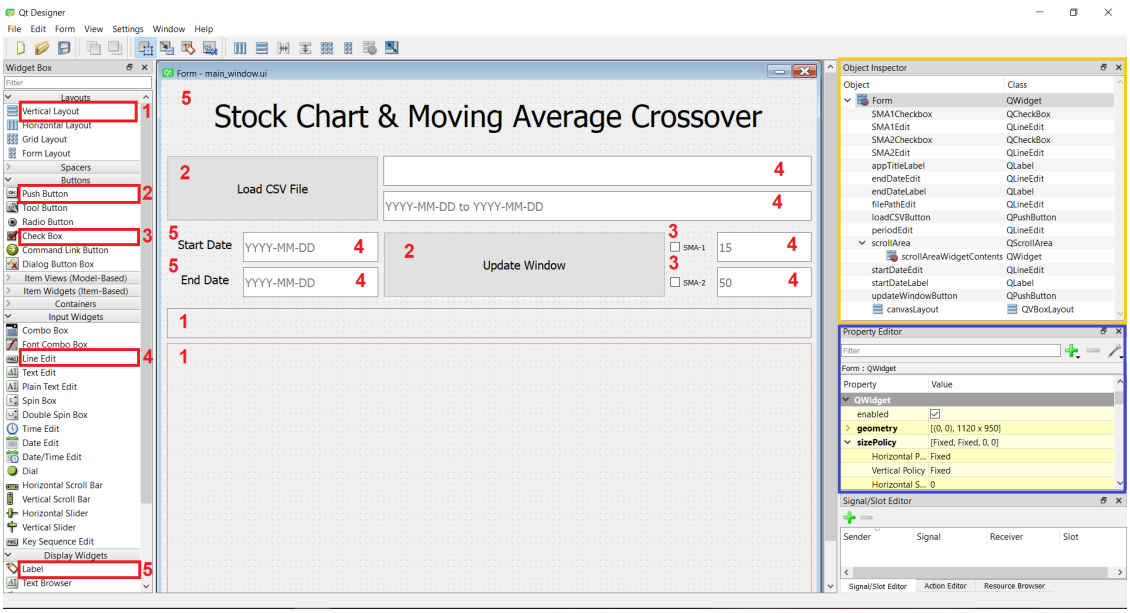
Choose `Widget` under the `template\forms` prompt and press the `Create` Button to begin designing `main_window.ui`.

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`*

1.2.2 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.

1.3 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.

1.4 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`.

2 `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`

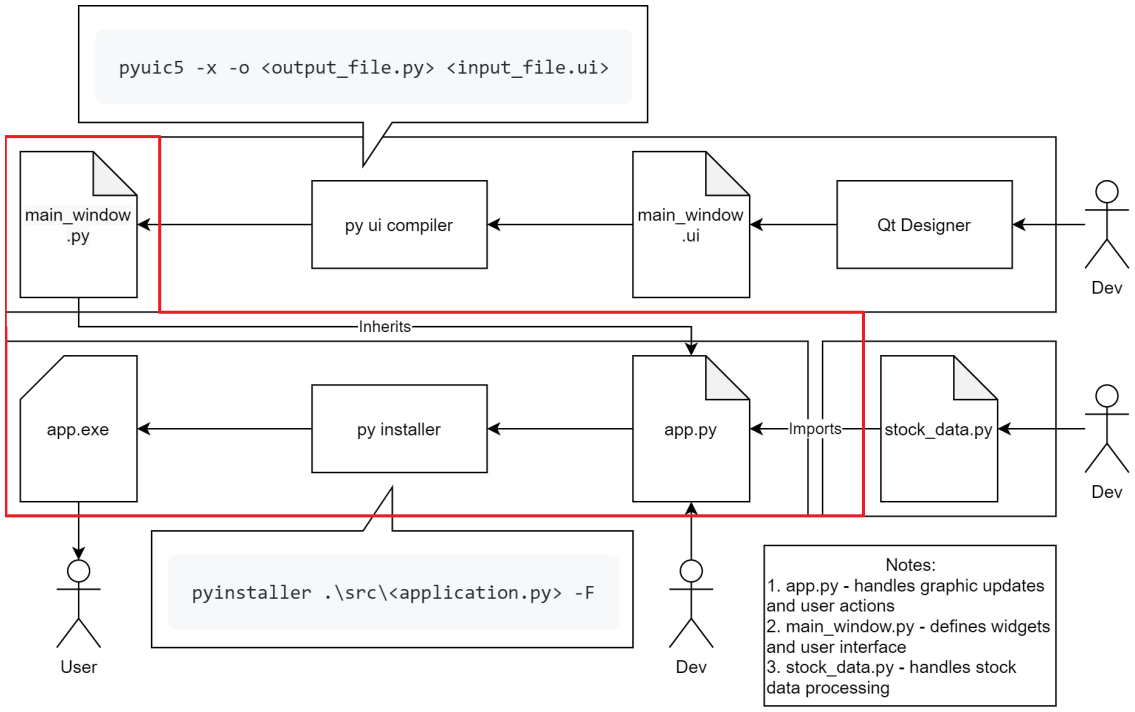
For example, if we want the `Update Window Button` to plot the stock prices in the GUI's canvas. We will have to create a `function` that plots the graph into the canvas and then connect the `Update Window Button` to this `function`.

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!

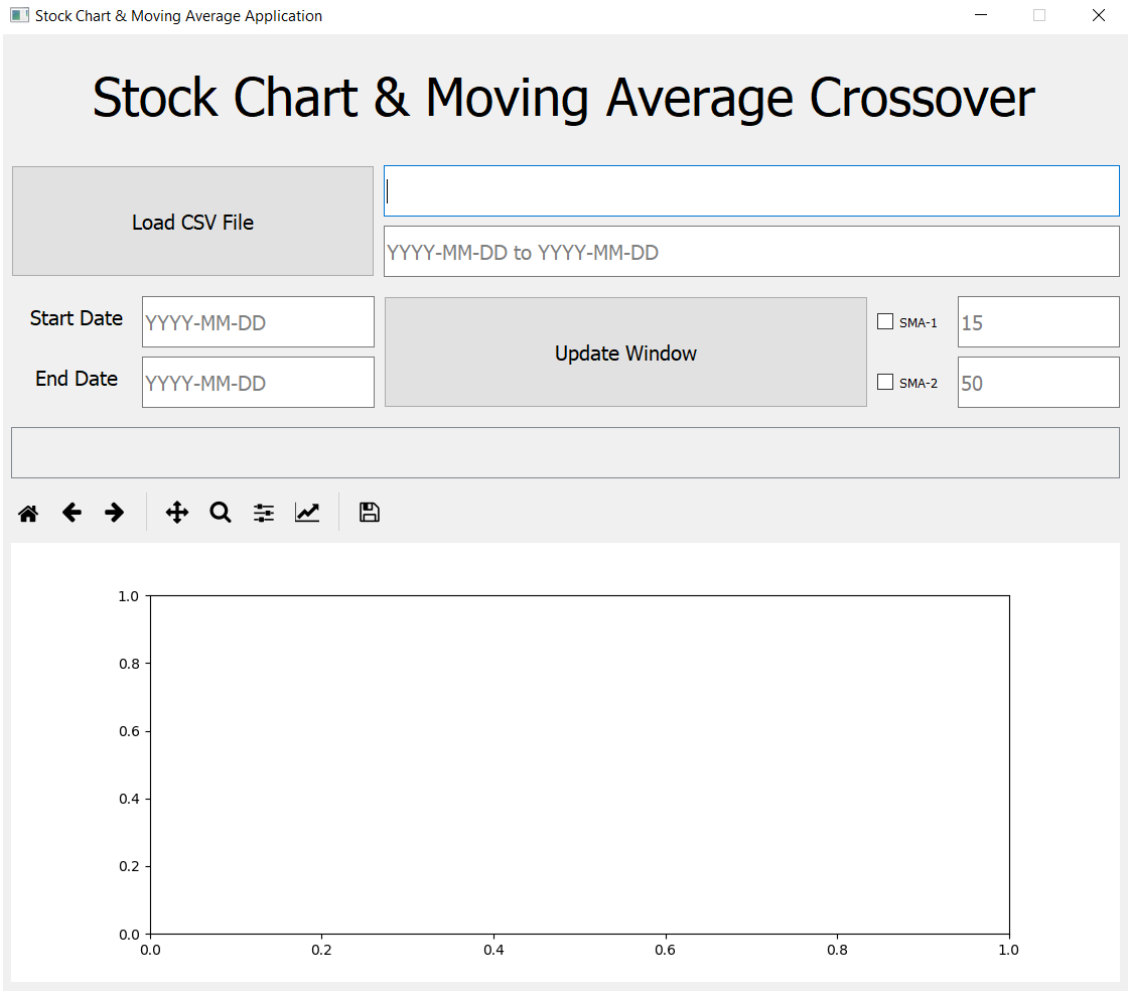
This is why, on the previous step, it is **recommended** to name the `Widgets` **accordingly!**

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.



2.1 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, add the `canvas` and `toolbar` widget to the `canvasLayout`

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

[ ]:
```
import sys

# 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

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)

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`

2.2 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, Buya 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 columnd_headers using the formats specified (Helps to handles the actions 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 `__init__(self)` is called (i.e. during the startup of application)

   centers the fixed main window size according to user screen size

[15]:
```
import sys
sys.path.insert(1, '../src')
```

```python
from app import Main
import main_window
import inspect # standard library used later to get info about the source code

function_1 = Main.load_data.__doc__
function_2 = Main.update_canvas.__doc__
function_3 = Main.plot_graph.__doc__
function_4 = Main.report.__doc__
function_5 = Main.center.__doc__

print(f"""
3 Major Functions' Scope:
    1. load_data(self)
    {function_1}
    2. update_canvas(self)
    {function_2}
    3. plot_graph(self, column_headers, formats)
    {function_3}
2 Minor Functions' Scope:
    1. report(self, string)
    {function_4}
    2. center(self)
    {function_5}
""")
```

```
3 Major Functions' Scope:
    1. load_data(self)

                loads stock data .csv from inputted filepath string on the GUI
as StockData object,
                also autocompletes all inputs using information provided by the
csv.

                Error handling
                        invalid filepath :
                                empty filepath or file could not be found or
opened.
                        invalid .csv :
                                .csv file is empty, missing date column, etc.

    2. update_canvas(self)

                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, Buya and Sell
plots need to be drawn.
                finally, updates graphic accordingly

                Error handling
                invalid date format:
                        date format inside the .csv file is not of form YYYY-MM-
DD
                non-existent stock_data :
                        the selected range results in an empty dataframe or end
date < start date
                non-existent data point :
                        data of that date does not exist, or maybe because it is
Out-Of-Bound
                raised exceptions :
                        SMA1 and SMA2 values are the same, or other exceptions
raised
```

3. plot_graph(self, column_headers, formats)

```
                plots graphs specified under columnd_headers using the formats
specified

                Parameters
                column_headers : [str, str, …]
                        a list containing column header names whose data are to
be plotted
                formats : [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)

                Error handling
                empty dataframe :
                        selected dataframe is empty
```

2 Minor Functions' Scope:

1. report(self, string)

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

                Parameters
                string : str
                        string of the report, usually the error message itself.
```

2. center(self)

```
                centers the fixed main window size according to user screen size
```

[ ]:

[ ]:

2.2.1 load_data(self)

[7]:
```
print(inspect.getsource(Main.load_data))
```

```
        def load_data(self):
                """
                loads stock data .csv from inputted filepath string on the GUI
as StockData object,
                also autocompletes all inputs using information provided by the
csv.

                Error handling
                        invalid filepath :
                                empty filepath or file could not be found or
opened.
                        invalid .csv :
                                .csv file is empty, missing date column, etc.
                """
                filepath = Path(self.filePathEdit.text())

                try:
                        self.stock_data = StockData(filepath)
                        start_date, end_date = self.stock_data.get_period()
                        period = f"{start_date} to {end_date}"

                        # auto-complete feauture
                        self.startDateEdit.setText(start_date)
                        self.endDateEdit.setText(end_date)
                        self.periodEdit.setText(period)
                        self.SMA1Edit.setText("15")
                        self.SMA2Edit.setText("50")
                        self.SMA1Checkbox.setChecked(False)
                        self.SMA2Checkbox.setChecked(False)

                        self.report(f"Data loaded from {filepath}; period auto-
selected: {start_date} to {end_date}.")
                        print(self.stock_data.data)

                except IOError as e:
```

```
                        self.report(f"Filepath provided is invalid or fail to
open .csv file. {e}")

                except TypeError as e:
                        self.report(f"The return tuple is probably (nan, nan)
because .csv is empty")
```

2.2.2 update_canvas(self)

[13]:
```
print(inspect.getsource(Main.update_canvas))
```

```
        def update_canvas(self):
                """
                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, Buya and Sell
plots need to be drawn.
                finally, updates graphic accordingly

                Error handling
                invalid date format:
                        date format inside the .csv file is not of form YYYY-MM-
DD
                non-existent stock_data :
                        the selected range results in an empty dataframe or end
date < start date
                non-existent data point :
                        data of that date does not exist, or maybe because it is
Out-Of-Bound
                raised exceptions :
                        SMA1 and SMA2 values are the same, or other exceptions
raised
                """
                self.ax.clear()
                self.date_format = '%Y-%m-%d'

                try:
                        start_date =
str(datetime.strptime(self.startDateEdit.text(), self.date_format).date())
                        end_date =
str(datetime.strptime(self.endDateEdit.text(), self.date_format).date())
                        period = f"{start_date} to {end_date}"
                        self.periodEdit.setText(period)

                        # builds a list of graphs to plot by checking the
```

```
tickboxes
                        column_headers = ['Close']
                        formats = ['k-']

                        if self.SMA1Checkbox.isChecked():
self.stock_data._calculate_SMA(int(self.SMA1Edit.text()))
column_headers.append(f"SMA{self.SMA1Edit.text()}")
                                formats.append('b-')
                        if self.SMA2Checkbox.isChecked():
self.stock_data._calculate_SMA(int(self.SMA2Edit.text()))
column_headers.append(f"SMA{self.SMA2Edit.text()}")
                                formats.append('c-')
                        if len(column_headers) == 3:
self.stock_data._calculate_crossover(column_headers[1], column_headers[2],
column_headers[1])
                                column_headers.append('Sell')
                                formats.append('rv')
                                column_headers.append('Buy')
                                formats.append('g^')

                        self.selected_stock_data =
self.stock_data.get_data(start_date, end_date)
                        self.plot_graph(column_headers, formats)

                        self.report(f"Plotting {column_headers} data from
period: {start_date} to {end_date}.")
                        print(self.selected_stock_data)

                except ValueError as e:
                        self.report(f"Time period has not been specified or does
not match YYYY-MM-DD format, {e}.")

                except AssertionError as e:
                        self.report(f"Selected range is empty, {e}")

                except KeyError as e:
                        self.report(f"Data for this date does not exist: {e}")

                except Exception as e:
                        self.report(f"Exception encountered: {e}")
```

2.2.3 plot_graph(self, column_headers, formats)

[14]:
```
print(inspect.getsource(Main.plot_graph))
```

```
        def plot_graph(self, column_headers, formats):
                """
```

```
                plots graphs specified under columnd_headers using the formats
specified

                Parameters
                column_headers : [str, str, …]
                        a list containing column header names whose data are to
be plotted
                formats : [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)

                Error handling
                empty dataframe :
                        selected dataframe is empty
                """
                self.ax.clear()
                assert not self.selected_stock_data.empty

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

                colors = ['black', 'blue', 'orange', 'red', 'green']
                for i in range(len(column_headers)):
                        if column_headers[i] in
self.selected_stock_data.columns:
                                y_data =
list(self.selected_stock_data[column_headers[i]])
                                self.ax.plot(x_data, y_data, formats[i],
label=column_headers[i], color=colors[i])
                                self.report(f"{column_headers[i]} data is being
plotted.")
                        else: self.report(f"{column_headers[i]} data does not
exist.")

                # formatting
                months_locator = mdates.MonthLocator()
                months_format = mdates.DateFormatter('%b %Y')
                self.ax.xaxis.set_major_locator(months_locator)
                self.ax.xaxis.set_major_formatter(months_format)
```

```
                self.ax.format_xdata = mdates.DateFormatter(self.date_format)
                self.ax.format_ydata = lambda y: '$%1.2f' % y
                self.ax.grid(True)
                self.figure.autofmt_xdate()
                self.figure.legend()
                self.figure.tight_layout()
                self.canvas.draw()
```

2.2.4 report(self, string)

[15]:
```
print(inspect.getsource(Main.report))
```

```
        def report(self, string):
                """
                given a report (string), update the scroll area with this report

                Parameters
                string : str
                        string of the report, usually the error message itself.
                """
                report_text = qtw.QLabel(string)
                self.scrollLayout.addWidget(report_text)
                print(string)
```

2.2.5 center(self)

[16]:
```
print(inspect.getsource(Main.center))
```

```
        def center(self):
                """
                centers the fixed main window size according to user screen size
                """
                screen = qtw.QDesktopWidget().screenGeometry()
                main_window = self.geometry()
                x = (screen.width() - main_window.width()) / 2
                y = (screen.height() - main_window.height()) / 2 - 50   # pulls
the window up slightly (arbitrary)
                self.setFixedSize(main_window.width(), main_window.height())
                self.move(x, y)
```

2.3 Connecting Widget actions to functions

blabla

[11]:
```
print(inspect.getsource(Main.__init__))
```

```
        def __init__(self):
                """
                initializes and sets up GUI widgets and its connections
                """
                super().__init__()
                self.setupUi(self)
                self.setWindowTitle("Stock Chart & Moving Average Application")

                # sets up a new figure to plot on, then instantiates a canvas
and toolbar object
                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)

                # button & checkbox connections
                self.loadCSVButton.clicked.connect(self.load_data)
                self.updateWindowButton.clicked.connect(self.update_canvas)
                self.SMA1Checkbox.stateChanged.connect(self.update_canvas)
                self.SMA2Checkbox.stateChanged.connect(self.update_canvas)

                # auto-complete feauture
                self.filePathEdit.setText("../data/GOOG.csv")
```

2.4 (Optional) Compiling `app.exe`