app

November 16, 2020

1 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

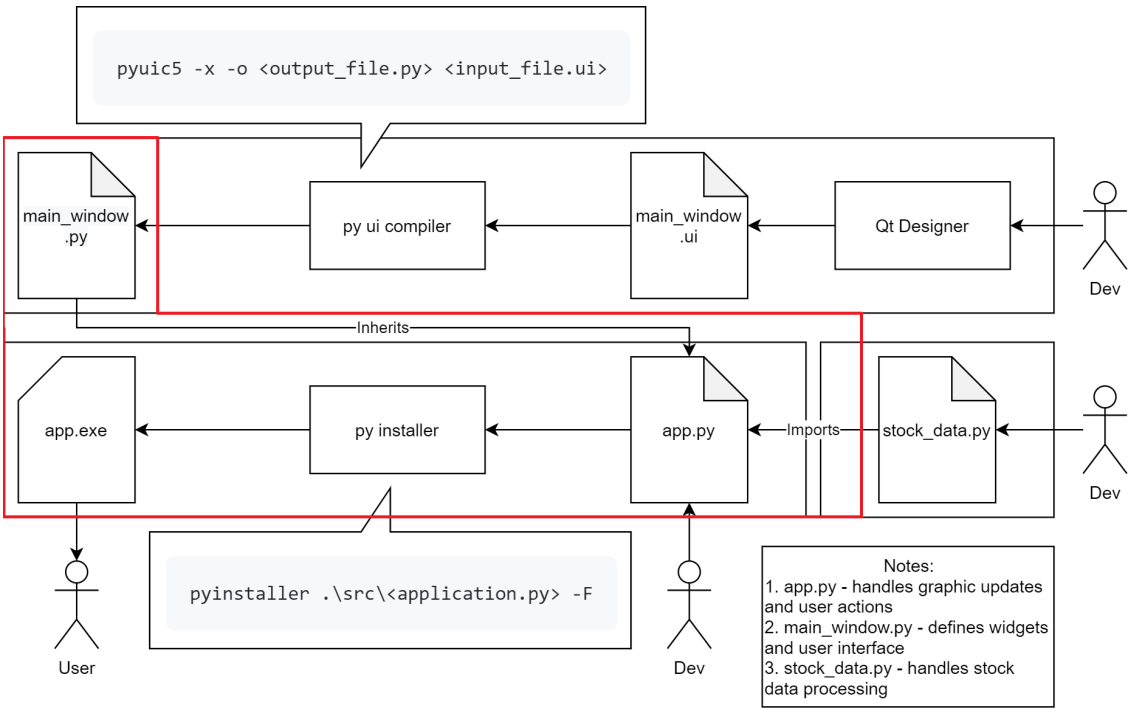
For example, if we want the Update Window Button to plot the stock prices in the GUI's canvas. We will have to create a function that plots the graph into the canvas and then connect the Update Window Button to this function.

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!

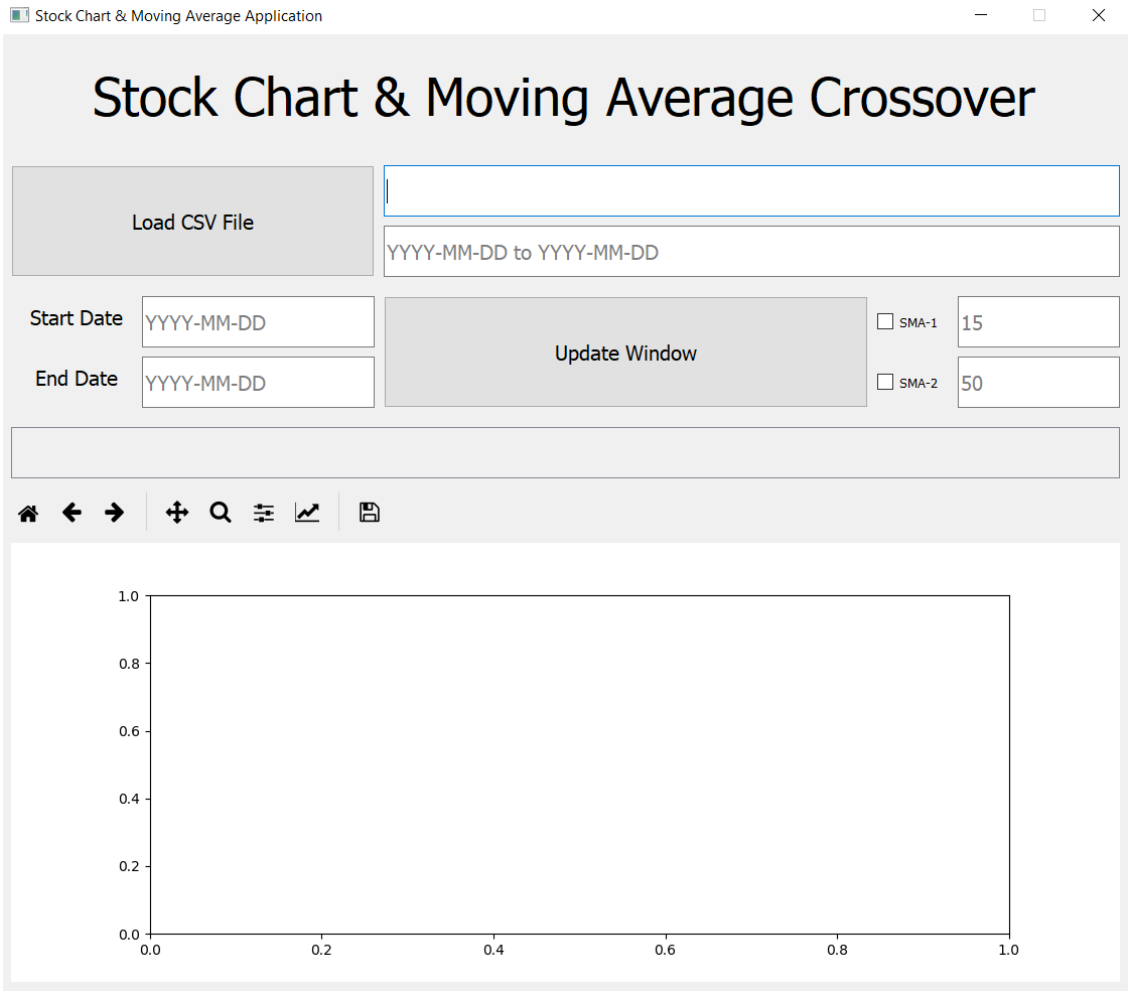
This is why, on the previous step, it is **recommended** to name the Widgets **accordingly!**

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.



1.1 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`

1.2 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**.

1.2.1 `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)`.

1.2.2 `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.*

1.2.3 `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)`.

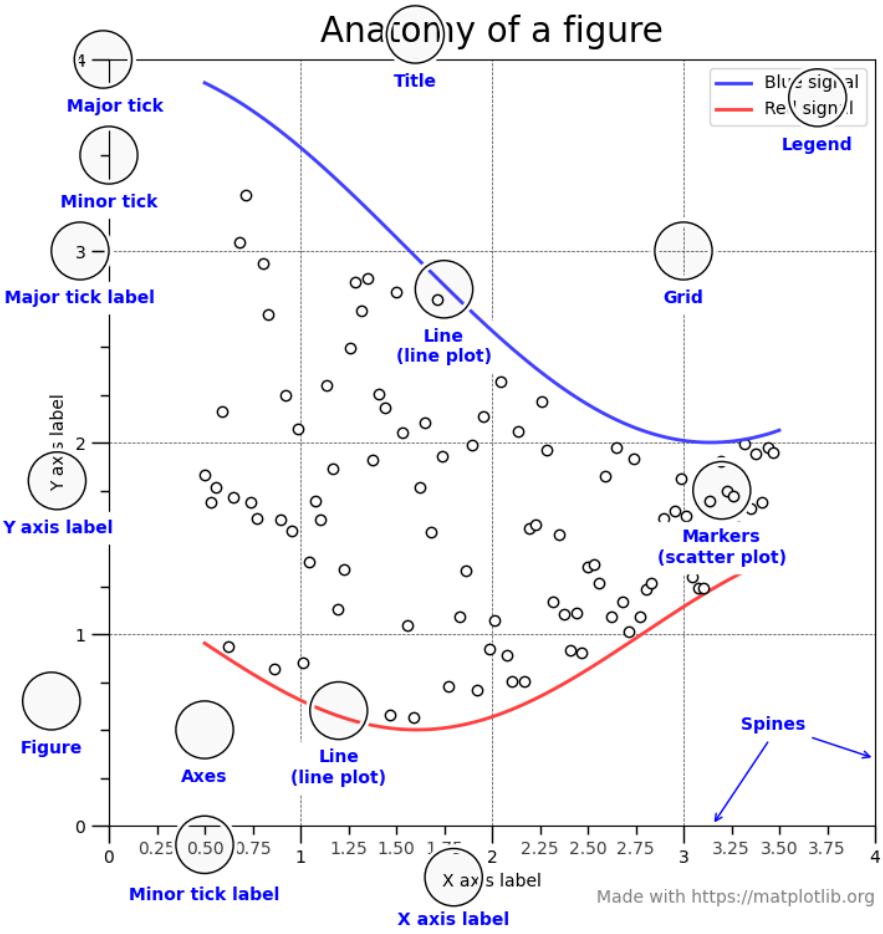
Once the plots are drawn, there may be some formatting that needs to be done on how either the `Axes` or the `Figure` looks like:

```
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.

1.2.4 `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)
```

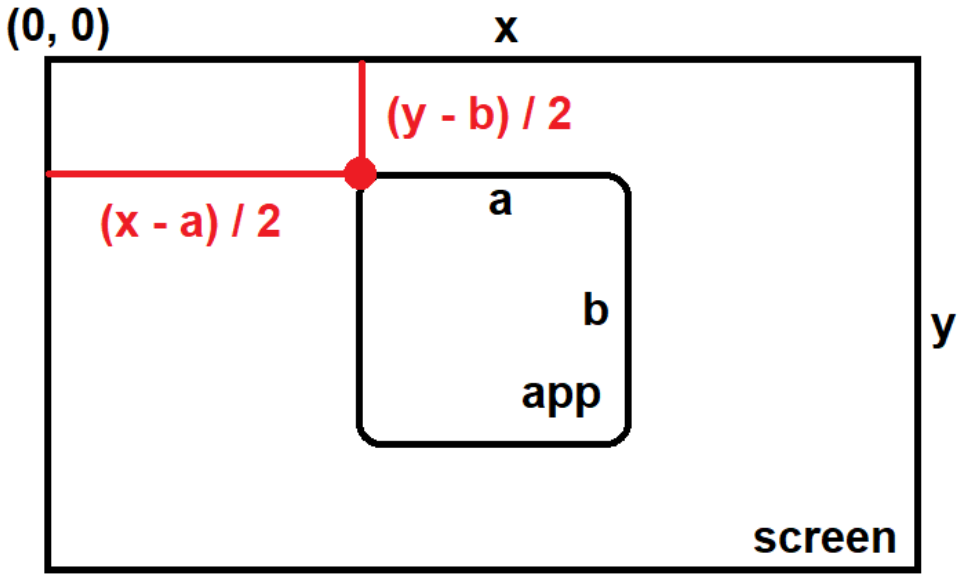
To simulate `print` statements, simply add new `Label Widget` with the `string` statement as its value. This is attached to a `Layout` that can be scrolled.

1.2.5 `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.

1.3 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`*

1.4 (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!*