app

November 14, 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

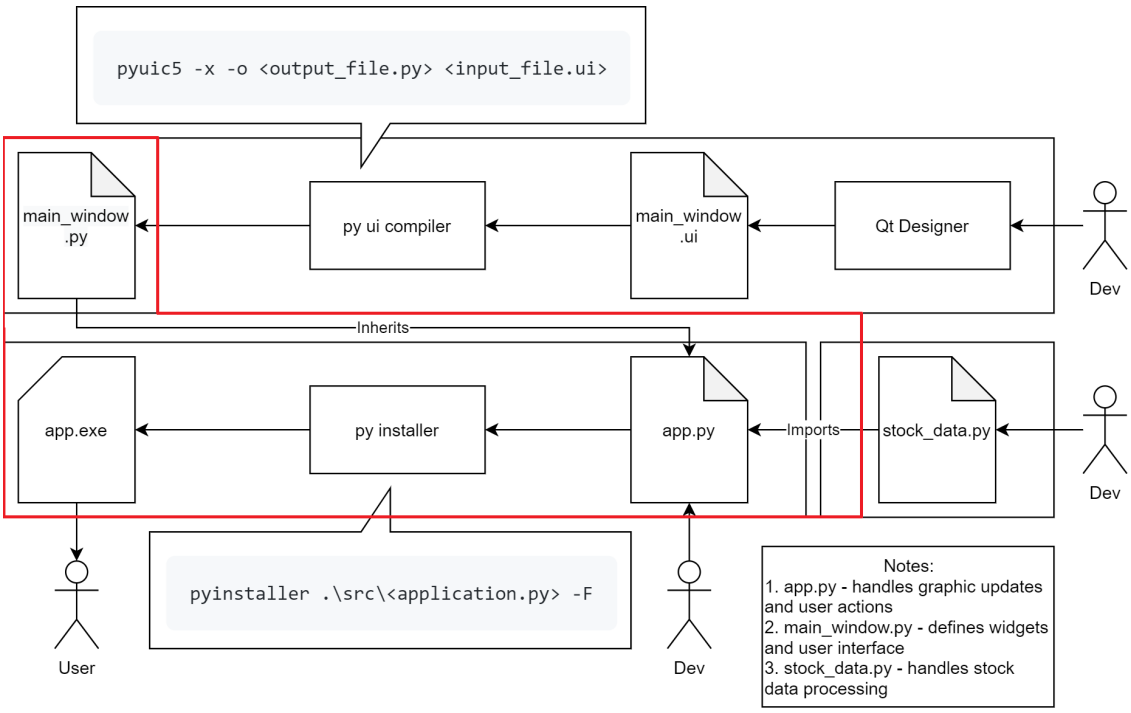
For example, if we want the Update Window Button to plot the stock prices in the GUI's canvas. We will have to create a function that plots the graph into the canvas and then connect the Update Window Button to this function.

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!

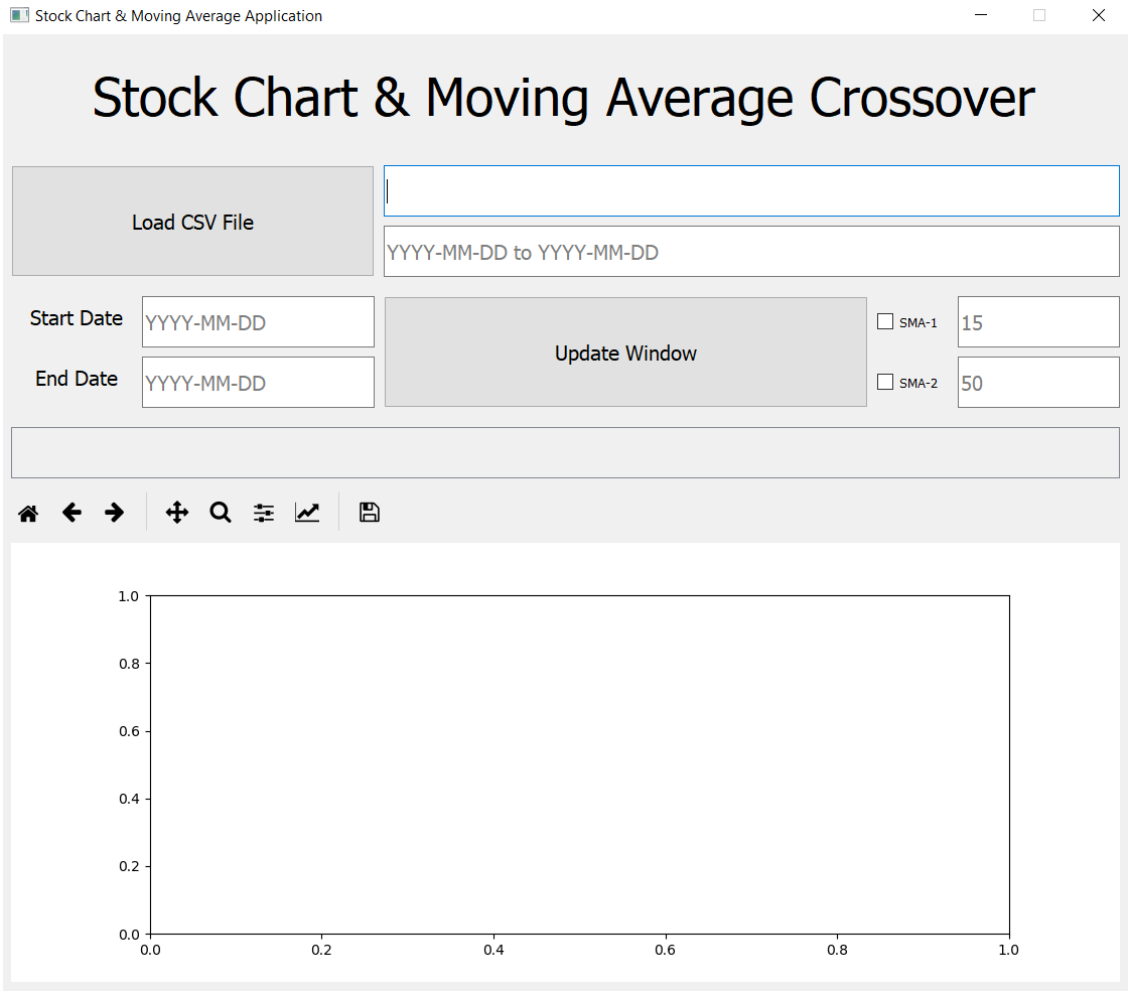
This is why, on the previous step, it is **recommended** to name the Widgets **accordingly!**

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, 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

# 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)

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 columnd_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 `__init__(self)` is called (i.e. during the startup of app)

   centers the fixed main window size according to user screen size

[2]:
```
import sys
sys.path.insert(1, '../src')

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

def beautify(code): # prints code with 2 less indent and without the def header
    print("".join([text[2:] if len(text) > 1 else text for text in code[0][1:
 ↪]]))
```

1.2.1 load_data(self)

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

```
"""
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.
        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")
```

1.2.2 update_canvas(self)

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

```
"""
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.

Error handling
invalid date format:
        date format inside the .csv file is not 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}")
```

1.2.3 plot_graph(self, column_headers, formats)

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

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

Parameters
column_headers : [str, str, …]
        a list containing column header names with data 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()
```

1.2.4 report(self, string)

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

```
"""
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)
```

1.2.5 center(self)

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

```
"""
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)
```

1.3 Connecting Widget actions to functions

blabla

[8]:
```
beautify(inspect.getsourcelines(Main.__init__))
```

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

# 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)

# 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")
```

1.4 (Optional) Compiling `app.exe`