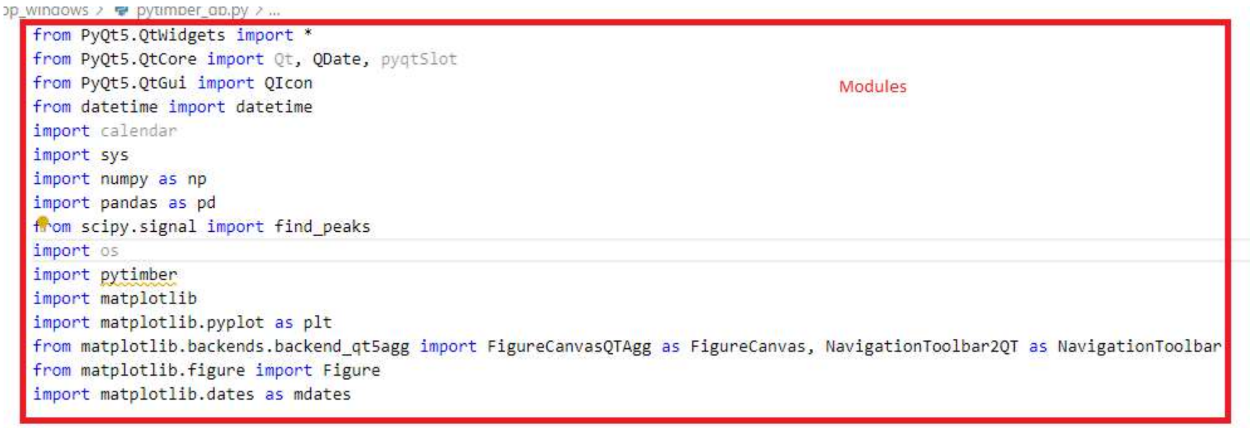
PYTIMBER GUI SCRIPT

- Modules. First 3 modules are PYQT5 modules which are necessary for graphical user interface.



```
from PyQt5.QtWidgets import *
from PyQt5.QtCore import Qt, QDate, pyqtSlot
from PyQt5.QtGui import QIcon
from datetime import datetime
import calendar
import sys
import numpy as np
import pandas as pd
from scipy.signal import find_peaks
import os
import pytimber
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas, NavigationToolbar2QT as NavigationToolbar
from matplotlib.figure import Figure
import matplotlib.dates as mdates
```

- These matplotlib functions make plot more readable.

```
matplotlib.use('Qt5Agg')
myFmt = mdates.DateFormatter('%d')
```

- pytimber.LoggingDB() function connects to the database.

```
db = pytimber.LoggingDB()
```

- Start_date and end_date dictionaries hold start date and end date informations in dictionaries.

```
start_date = {"start_day":datetime.now().day, "start_month":datetime.now().month,
 "start_year":datetime.now().year}

end_date   = {"end_day":datetime.now().day, "end_month":datetime.now().month, "en
d_year":datetime.now().year}
```

- This variable list hold the names of variables of machines which are same as at data base.

```
variable_list = ["LHC.BCTDC.A6R4.B1:BEAM_INTENSITY",
                 "LHC.BCTDC.A6R4.B2:BEAM_INTENSITY",
                 "LHC.BCTDC.B6R4.B1:BEAM_INTENSITY",
                 "LHC.BCTDC.B6R4.B2:BEAM_INTENSITY",
                 "LHC.BCTDC.A6R4.B1:BEAM_INTENSITY_ADC24BIT",
                 "LHC.BCTDC.A6R4.B2:BEAM_INTENSITY_ADC24BIT",
                 "LHC.BCTDC.B6R4.B1:BEAM_INTENSITY_ADC24BIT",
                 "LHC.BCTDC.B6R4.B2:BEAM_INTENSITY_ADC24BIT"]
```

- Another list which is using for put a name to the file while saving them.

```
data_save_list = ["ADC_16BIT_AB1", "ADC_16BIT_AB2", "ADC_16BIT_BB1",
                  "ADC_16BIT_BB2", "ADC_24BIT_AB1", "ADC_24BIT_AB2",
                  "ADC_24BIT_BB1", "ADC_24BIT_BB2"]
```

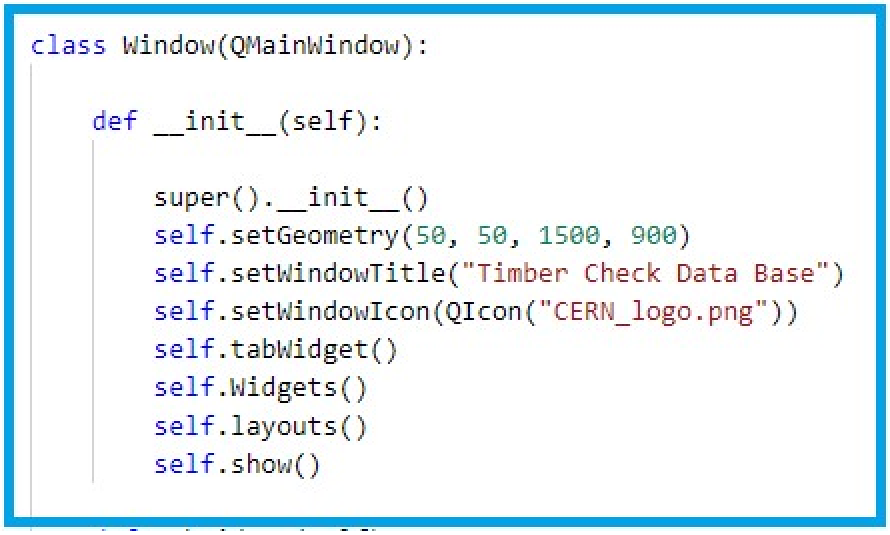
- Data frames generates by Pandas to hold values of variables.

```
df_16bit_AB1 = pd.DataFrame({"time":[], "intensity":[]}, columns = ["time", "inte
nsity"]);
df_16bit_AB2 = pd.DataFrame({"time":[], "intensity":[]}, columns = ["time", "inte
nsity"])
df_16bit_BB1 = pd.DataFrame({"time":[], "intensity":[]}, columns = ["time", "inte
nsity"]);
df_16bit_BB2 = pd.DataFrame({"time":[], "intensity":[]}, columns = ["time", "inte
nsity"])
df_24bit_AB1 = pd.DataFrame({"time":[], "intensity":[]}, columns = ["time", "inte
nsity"]);
df_24bit_AB2 = pd.DataFrame({"time":[], "intensity":[]}, columns = ["time", "inte
nsity"])
df_24bit_BB1 = pd.DataFrame({"time":[], "intensity":[]}, columns = ["time", "inte
nsity"]);
df_24bit_BB2 = pd.DataFrame({"time":[], "intensity":[]}, columns = ["time", "inte
nsity"])
dataFrame_list = [df_16bit_AB1, df_16bit_AB2, df_16bit_BB1, df_16bit_BB2,
                  df_24bit_AB1, df_24bit_AB2, df_24bit_BB1, df_24bit_BB2]
```

- Another global variables. Third one is step number which can be seen on the GUI directly.

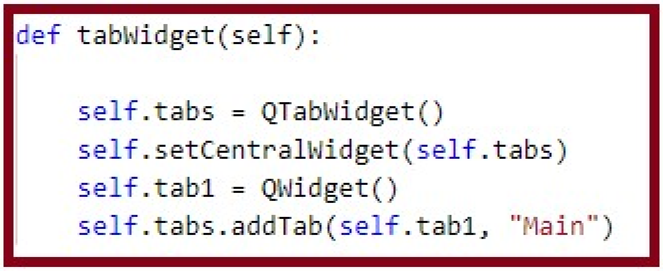
```
time_dict = {"tstart":"", "tend":""}
error_check = {"error":2}
step_number = {'step_number':80}
second_clicked = {'clicked':0}
```

- Window class is child class of QMainWindow. Init function includes super().__init__() which corresponds to during initialization, use every feature of parent class.
  setGeometry(start_x, start_y, height, width) is starting condition.
  setWindowTitle corresponds to a name which when GUI opened, you can see this name at left corner of window.
  SetWindowIcon corresponds to a logo which can be seen at the left of the window name.
  I will explain other function by one by below.



```python
class Window(QMainWindow):

    def __init__(self):

        super().__init__()
        self.setGeometry(50, 50, 1500, 900)
        self.setWindowTitle("Timber Check Data Base")
        self.setWindowIcon(QIcon("CERN_logo.png"))
        self.tabWidget()
        self.Widgets()
        self.layouts()
        self.show()
```

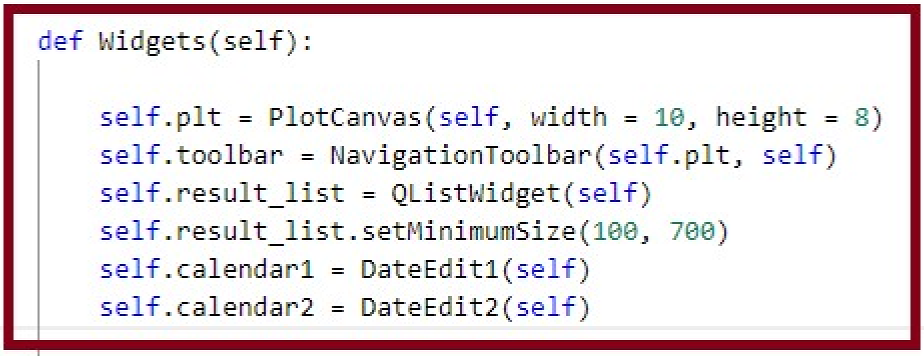
- tabWidget function creates a tab on the GUI. There is only one tab which is MAIN. It can be extend with using another tabs.



```python
def tabWidget(self):

    self.tabs = QTabWidget()
    self.setCentralWidget(self.tabs)
    self.tab1 = QWidget()
    self.tabs.addTab(self.tab1, "Main")
```

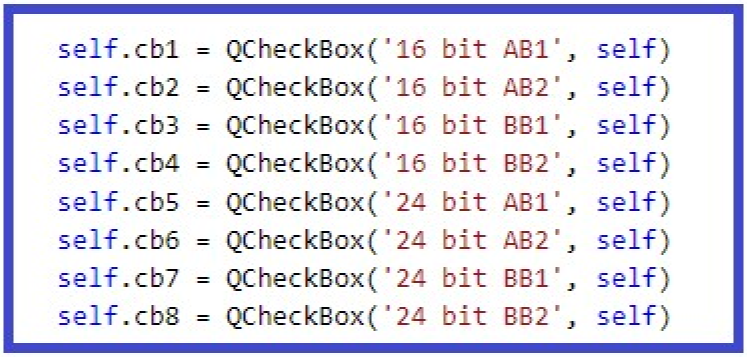
WIDGETS FUNCTION

- Widgets function is very long function which includes all the main programs in the code. First 2 lines represent plot figure which is at right side on the GUI. toolbar feature can be seen on the plot which includes functions such as zoom, save and so on.
  Result_list is the list which shows search results.
  There 2 different calendars which are for start date and end date of the search.



```
def Widgets(self):

    self.plt = PlotCanvas(self, width = 10, height = 8)
    self.toolbar = NavigationToolbar(self.plt, self)
    self.result_list = QListWidget(self)
    self.result_list.setMinimumSize(100, 700)
    self.calendar1 = DateEdit1(self)
    self.calendar2 = DateEdit2(self)
```

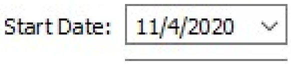
- These are the check box for plot function. If one of them check when plot button clicked, this variable data can be seen on plot.



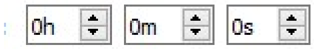
```
self.cb1 = QCheckBox('16 bit AB1', self)
self.cb2 = QCheckBox('16 bit AB2', self)
self.cb3 = QCheckBox('16 bit BB1', self)
self.cb4 = QCheckBox('16 bit BB2', self)
self.cb5 = QCheckBox('24 bit AB1', self)
self.cb6 = QCheckBox('24 bit AB2', self)
self.cb7 = QCheckBox('24 bit BB1', self)
self.cb8 = QCheckBox('24 bit BB2', self)
```

- There are 5 different functions from PyQt5 which repeat

1. QLabel function stands for text part of the GUI. In example below, You can see that there is a QLabel function which is 'Start Date:'



Start Date: 11/4/2020

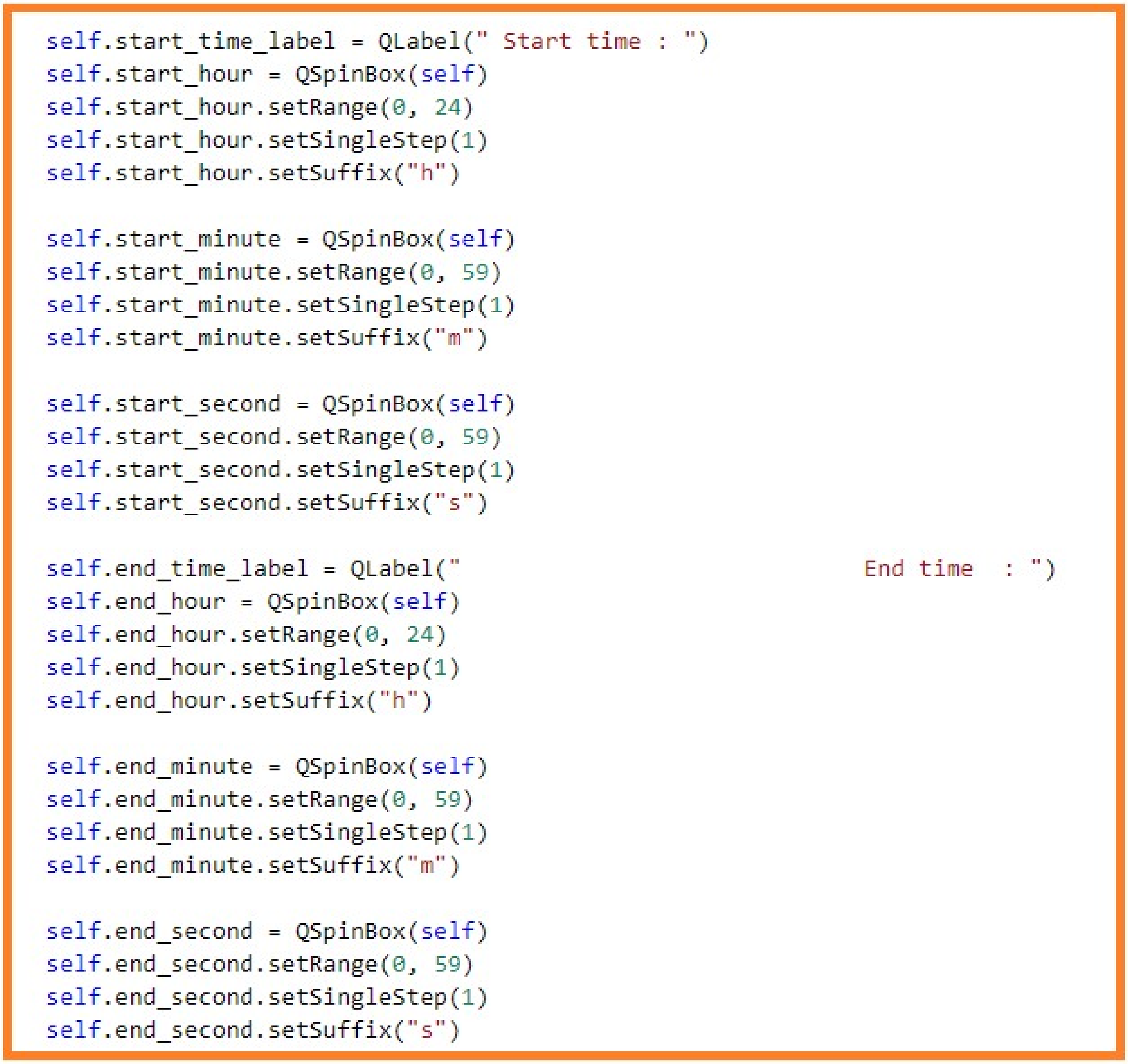
2. QSpinBox function stands for 

0h 0m 0s

3. setRange functions represents boundary of spin box.

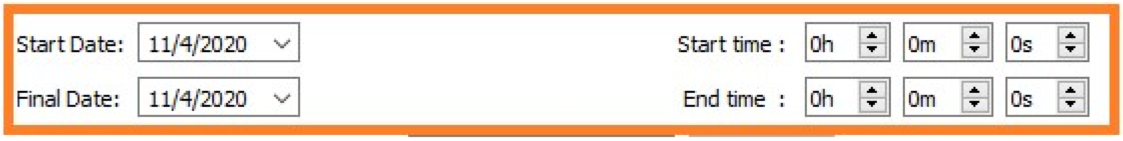
4. setSingleStep stands for when you clicked up arrow, it will increase simply one.

5. setSuffix is a kind of unit of increasing value in the spin box.



```
self.start_time_label = QLabel(" Start time : ")
self.start_hour = QSpinBox(self)
self.start_hour.setRange(0, 24)
self.start_hour.setSingleStep(1)
self.start_hour.setSuffix("h")

self.start_minute = QSpinBox(self)
self.start_minute.setRange(0, 59)
self.start_minute.setSingleStep(1)
self.start_minute.setSuffix("m")

self.start_second = QSpinBox(self)
self.start_second.setRange(0, 59)
self.start_second.setSingleStep(1)
self.start_second.setSuffix("s")

self.end_time_label = QLabel("                           End time   : ")
self.end_hour = QSpinBox(self)
self.end_hour.setRange(0, 24)
self.end_hour.setSingleStep(1)
self.end_hour.setSuffix("h")

self.end_minute = QSpinBox(self)
self.end_minute.setRange(0, 59)
self.end_minute.setSingleStep(1)
self.end_minute.setSuffix("m")

self.end_second = QSpinBox(self)
self.end_second.setRange(0, 59)
self.end_second.setSingleStep(1)
self.end_second.setSuffix("s")
```

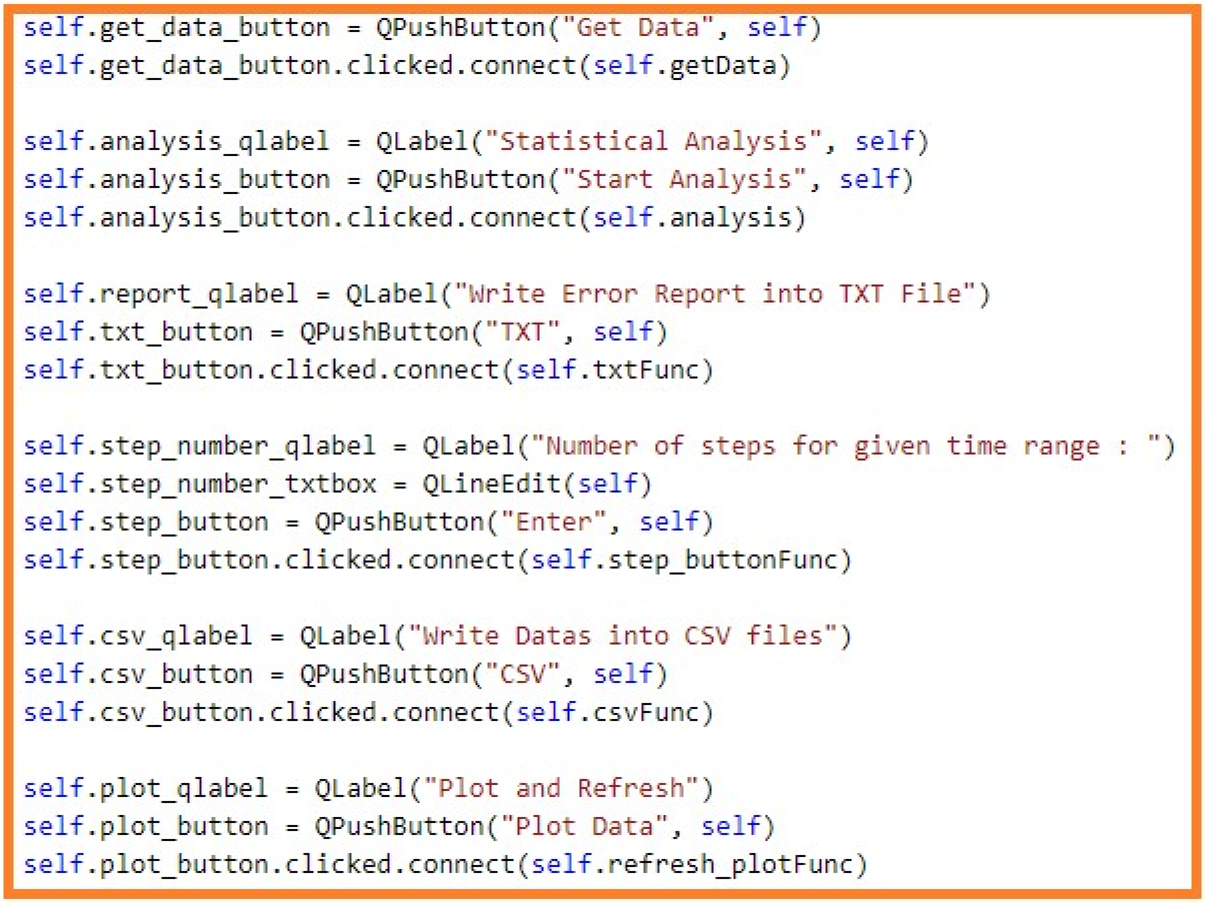
If we visualize the code, it looks like :



Start Date: 11/4/2020 Start time : 0h 0m 0s

Final Date: 11/4/2020 End time : 0h 0m 0s

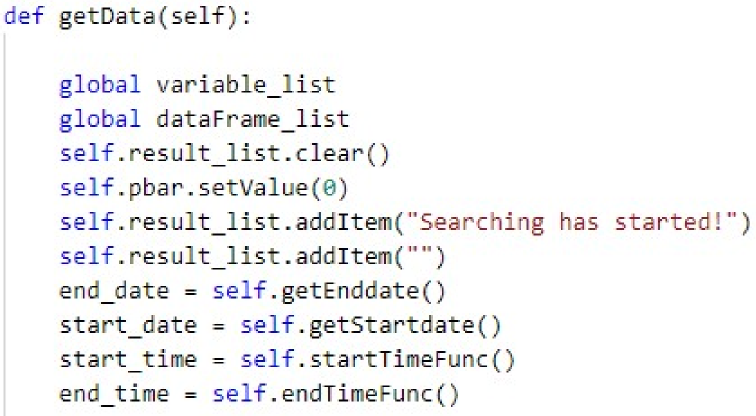
- Button part of the Widgets function:



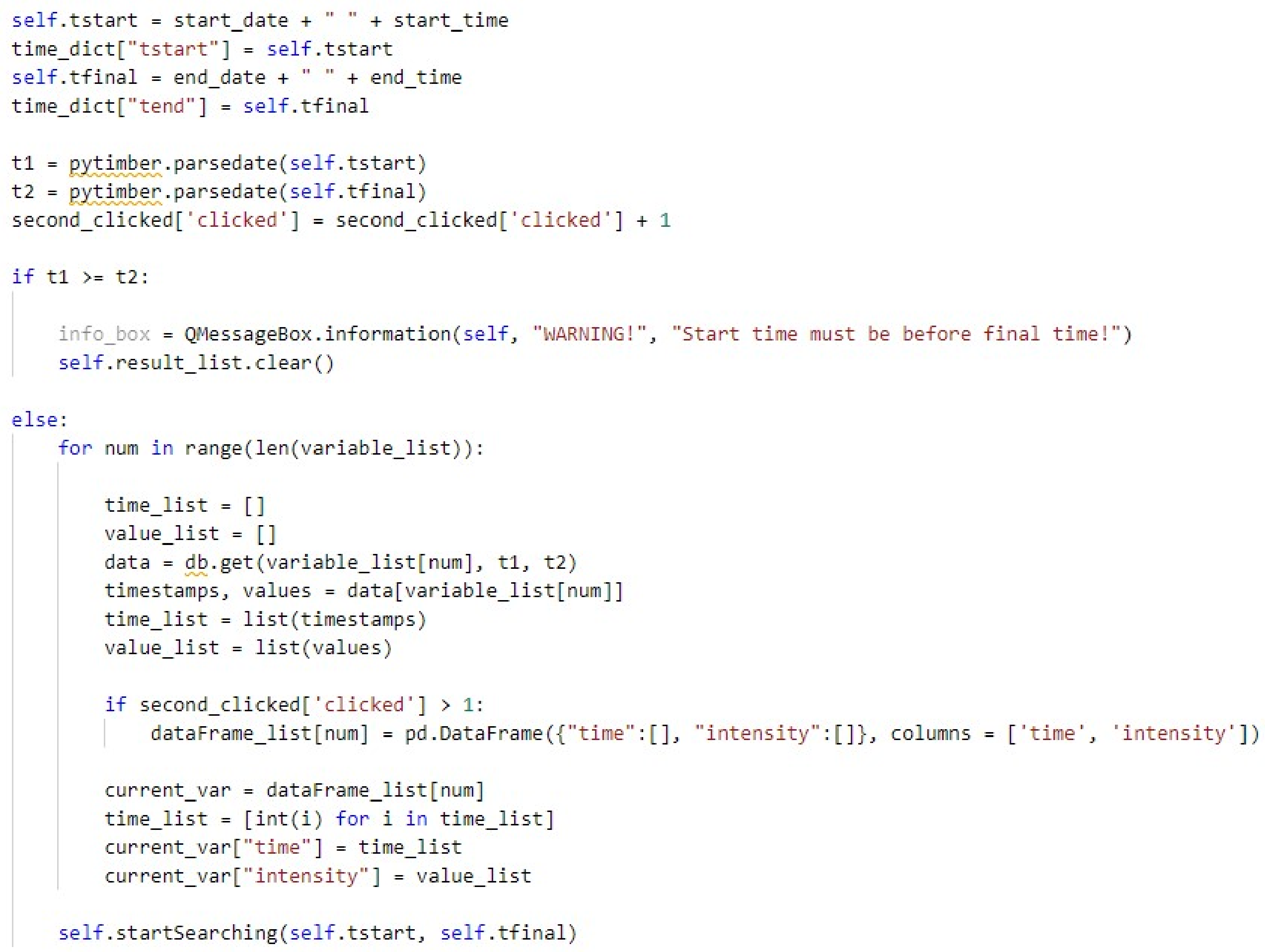
```
self.get_data_button = QPushButton("Get Data", self)
self.get_data_button.clicked.connect(self.getData)

self.analysis_qlabel = QLabel("Statistical Analysis", self)
self.analysis_button = QPushButton("Start Analysis", self)
self.analysis_button.clicked.connect(self.analysis)

self.report_qlabel = QLabel("Write Error Report into TXT File")
self.txt_button = QPushButton("TXT", self)
self.txt_button.clicked.connect(self.txtFunc)

self.step_number_qlabel = QLabel("Number of steps for given time range : ")
self.step_number_txtbox = QLineEdit(self)
self.step_button = QPushButton("Enter", self)
self.step_button.clicked.connect(self.step_buttonFunc)

self.csv_qlabel = QLabel("Write Datas into CSV files")
self.csv_button = QPushButton("CSV", self)
self.csv_button.clicked.connect(self.csvFunc)

self.plot_qlabel = QLabel("Plot and Refresh")
self.plot_button = QPushButton("Plot Data", self)
self.plot_button.clicked.connect(self.refresh_plotFunc)
```

Every button connects a function. When button clicked function get involved the program. There are explanations of buttons and their functions below:

1. Get_data button:



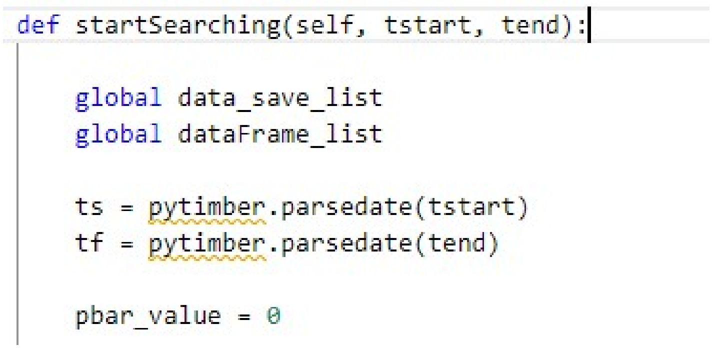
```
def getData(self):

    global variable_list
    global dataFrame_list
    self.result_list.clear()
    self.pbar.setValue(0)
    self.result_list.addItem("Searching has started!")
    self.result_list.addItem("")
    end_date = self.getEnddate()
    start_date = self.getStartdate()
    start_time = self.startTimeFunc()
    end_time = self.endTimeFunc()
```



```
self.tstart = start_date + " " + start_time
time_dict["tstart"] = self.tstart
self.tfinal = end_date + " " + end_time
time_dict["tend"] = self.tfinal

t1 = pytimber.parsedate(self.tstart)
t2 = pytimber.parsedate(self.tfinal)
second_clicked['clicked'] = second_clicked['clicked'] + 1

if t1 >= t2:

    info_box = QMessageBox.information(self, "WARNING!", "Start time must be before final time!")
    self.result_list.clear()

else:
    for num in range(len(variable_list)):

        time_list = []
        value_list = []
        data = db.get(variable_list[num], t1, t2)
        timestamps, values = data[variable_list[num]]
        time_list = list(timestamps)
        value_list = list(values)

        if second_clicked['clicked'] > 1:
            dataFrame_list[num] = pd.DataFrame({"time":[], "intensity":[]}, columns = ['time', 'intensity'])

        current_var = dataFrame_list[num]
        time_list = [int(i) for i in time_list]
        current_var["time"] = time_list
        current_var["intensity"] = value_list

    self.startSearching(self.tstart, self.tfinal)
```

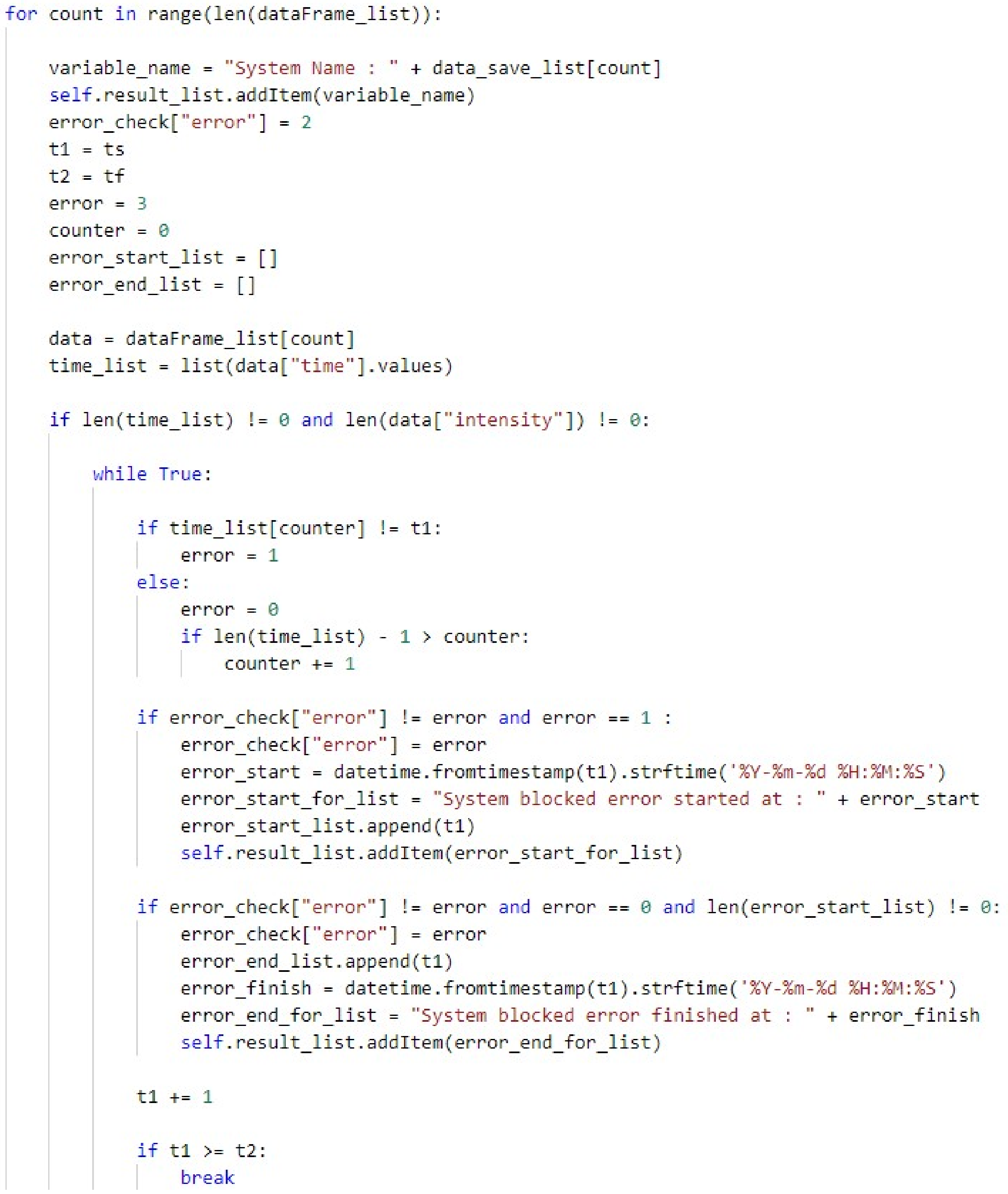
As you can see, when getData button clicked, it acquires given start time and end time. After that until all variables taken from data base, it works. End of the code there is also another function. After all data has taken from database, it triggers to search of errors in data bases one by one.

startSearching function is more complicated then the others, because it tries to find both system blocks as well as glitches. For every variable when searching process finished our bar fills.



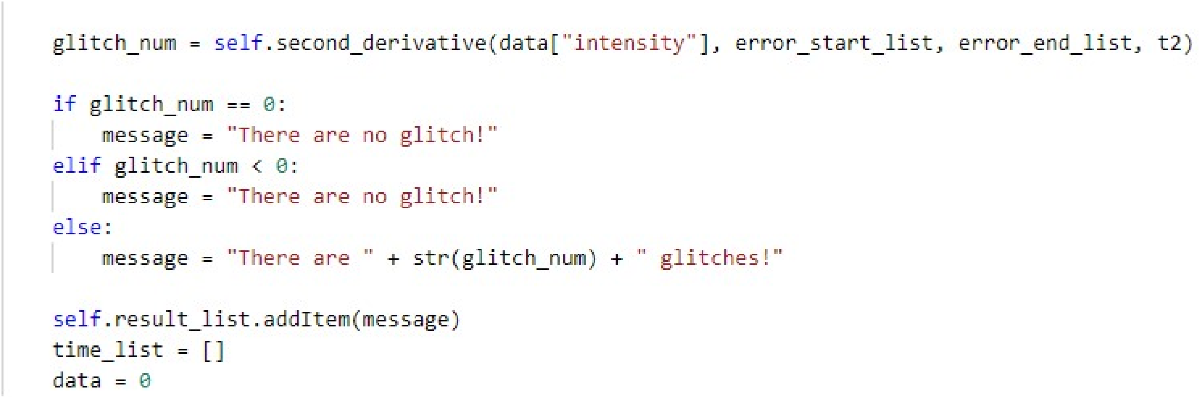
```
def startSearching(self, tstart, tend):

    global data_save_list
    global dataFrame_list

    ts = pytimber.parsedate(tstart)
    tf = pytimber.parsedate(tend)

    pbar_value = 0
```

This part of code finds the system blocks if there is any.



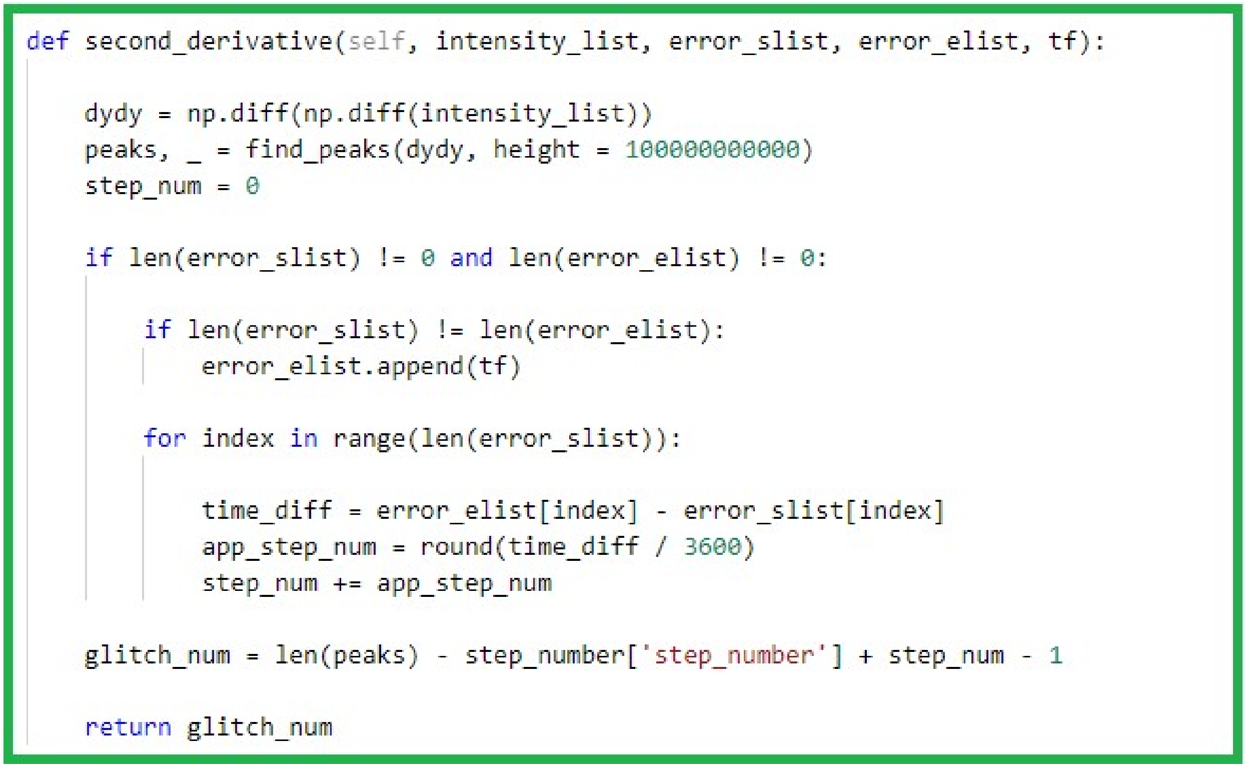
```
for count in range(len(dataFrame_list)):

    variable_name = "System Name : " + data_save_list[count]
    self.result_list.addItem(variable_name)
    error_check["error"] = 2
    t1 = ts
    t2 = tf
    error = 3
    counter = 0
    error_start_list = []
    error_end_list = []

    data = dataFrame_list[count]
    time_list = list(data["time"].values)

    if len(time_list) != 0 and len(data["intensity"]) != 0:

        while True:

            if time_list[counter] != t1:
                error = 1
            else:
                error = 0
                if len(time_list) - 1 > counter:
                    counter += 1

            if error_check["error"] != error and error == 1 :
                error_check["error"] = error
                error_start = datetime.fromtimestamp(t1).strftime('%Y-%m-%d %H:%M:%S')
                error_start_for_list = "System blocked error started at : " + error_start
                error_start_list.append(t1)
                self.result_list.addItem(error_start_for_list)

            if error_check["error"] != error and error == 0 and len(error_start_list) != 0:
                error_check["error"] = error
                error_end_list.append(t1)
                error_finish = datetime.fromtimestamp(t1).strftime('%Y-%m-%d %H:%M:%S')
                error_end_for_list = "System blocked error finished at : " + error_finish
                self.result_list.addItem(error_end_for_list)

            t1 += 1

            if t1 >= t2:
                break
```

After system blocked search finishes, second_derivative function searches for glitches.



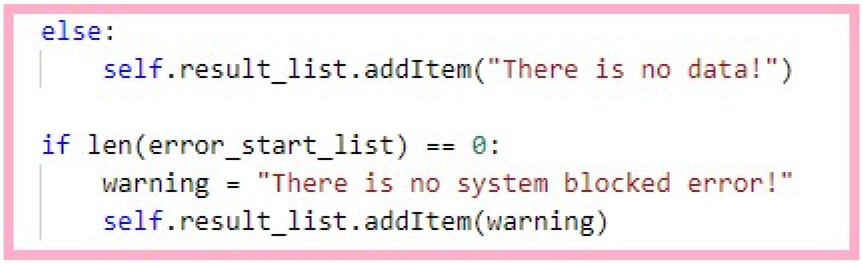
```
glitch_num = self.second_derivative(data["intensity"], error_start_list, error_end_list, t2)

if glitch_num == 0:
    message = "There are no glitch!"
elif glitch_num < 0:
    message = "There are no glitch!"
else:
    message = "There are " + str(glitch_num) + " glitches!"

self.result_list.addItem(message)
time_list = []
data = 0
```

Second_derivative functions takes 2 times derivative of intensities and find peaks to see if there is any glitch. I also use step number to decrease number of peaks which are steps. So it is necessary to give true number of steps before use getData button if the steps are different than normal step number. Step number has initial value as 80.



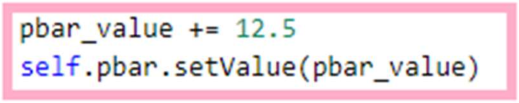
```
def second_derivative(self, intensity_list, error_slist, error_elist, tf):

    dydy = np.diff(np.diff(intensity_list))
    peaks, _ = find_peaks(dydy, height = 100000000000)
    step_num = 0

    if len(error_slist) != 0 and len(error_elist) != 0:

        if len(error_slist) != len(error_elist):
            error_elist.append(tf)

        for index in range(len(error_slist)):

            time_diff = error_elist[index] - error_slist[index]
            app_step_num = round(time_diff / 3600)
            step_num += app_step_num

    glitch_num = len(peaks) - step_number['step_number'] + step_num - 1

    return glitch_num
```

If there is no data, it goes to else condition which can be seen in the result list.



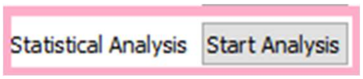
```
else:
    self.result_list.addItem("There is no data!")

if len(error_start_list) == 0:
    warning = "There is no system blocked error!"
    self.result_list.addItem(warning)
```

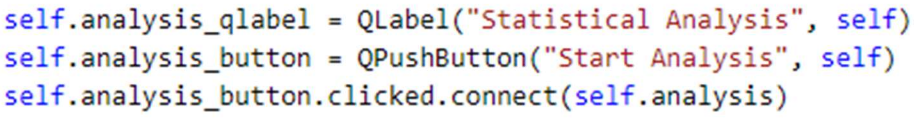
Finally pbar value increases :



```
pbar_value += 12.5
self.pbar.setValue(pbar_value)
```

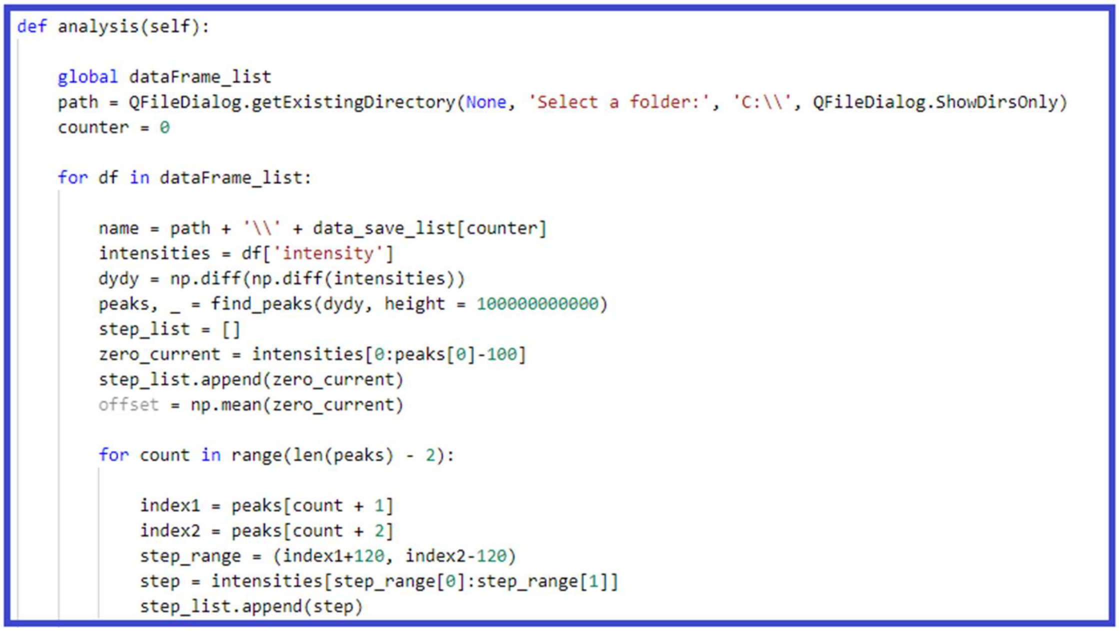
2. Start Analysis Button:





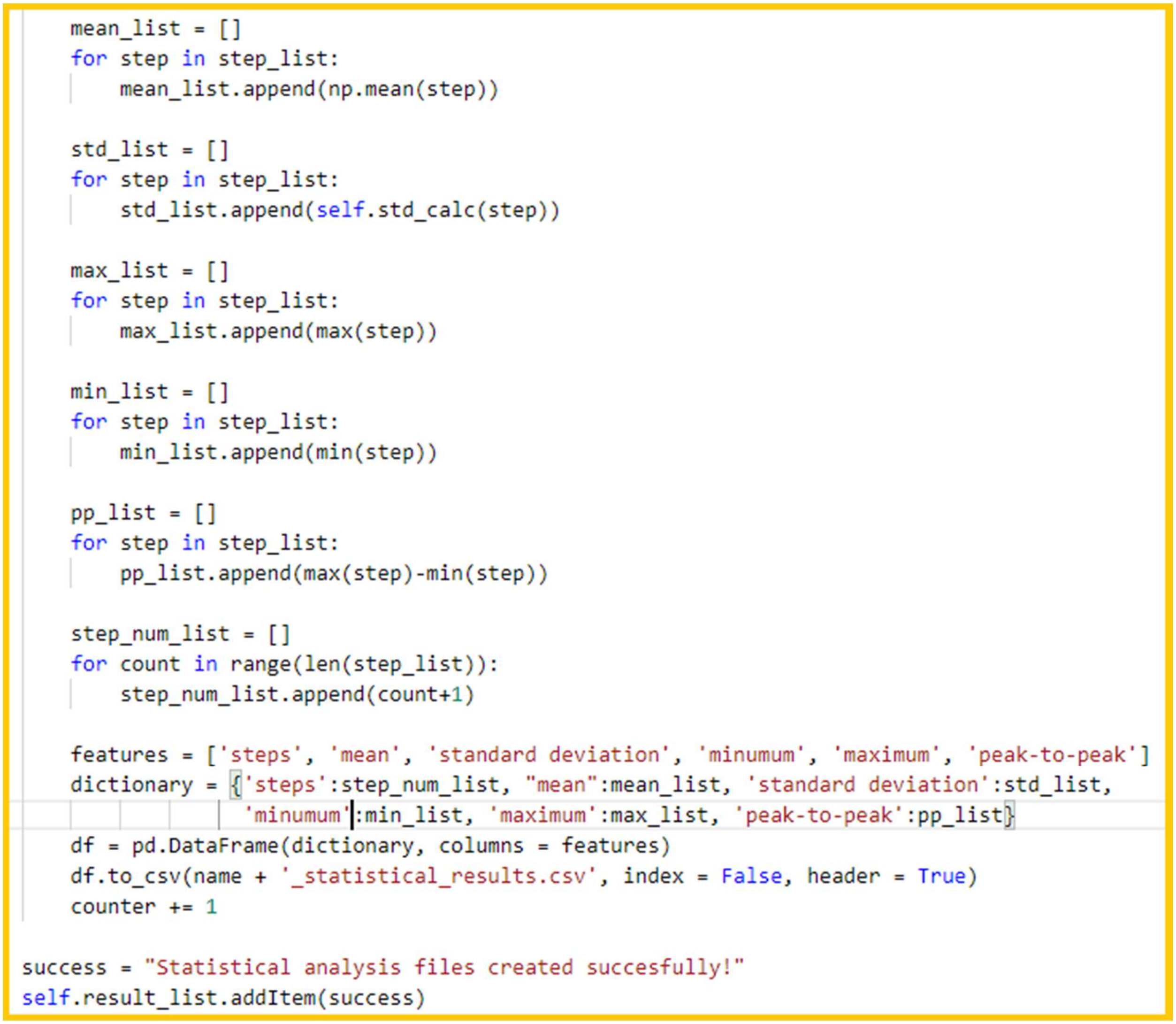
```
self.analysis_qlabel = QLabel("Statistical Analysis", self)
self.analysis_button = QPushButton("Start Analysis", self)
self.analysis_button.clicked.connect(self.analysis)
```

When this button clicked, it goes to the analysis function which is



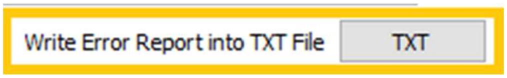
```
def analysis(self):

    global dataFrame_list
    path = QFileDialog.getExistingDirectory(None, 'Select a folder:', 'C:\\', QFileDialog.ShowDirsOnly)
    counter = 0

    for df in dataFrame_list:

        name = path + '\\' + data_save_list[counter]
        intensities = df['intensity']
        dydy = np.diff(np.diff(intensities))
        peaks, _ = find_peaks(dydy, height = 100000000000)
        step_list = []
        zero_current = intensities[0:peaks[0]-100]
        step_list.append(zero_current)
        offset = np.mean(zero_current)

        for count in range(len(peaks) - 2):

            index1 = peaks[count + 1]
            index2 = peaks[count + 2]
            step_range = (index1+120, index2-120)
            step = intensities[step_range[0]:step_range[1]]
            step_list.append(step)
```

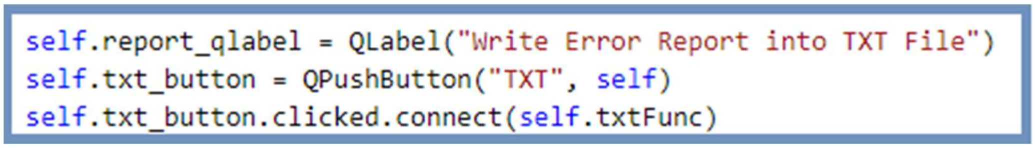
When function is activated, it opens a folder discovery which expects a folder for saving analysis results as CSV file. After that it starts to analyze all data frames one by one. It finds mean values, standard deviation, maximum value, minimum value and peak to peak values of steps. After it has done, you can see the response of function in the result list.



```
    mean_list = []
    for step in step_list:
        mean_list.append(np.mean(step))

    std_list = []
    for step in step_list:
        std_list.append(self.std_calc(step))

    max_list = []
    for step in step_list:
        max_list.append(max(step))

    min_list = []
    for step in step_list:
        min_list.append(min(step))

    pp_list = []
    for step in step_list:
        pp_list.append(max(step)-min(step))

    step_num_list = []
    for count in range(len(step_list)):
        step_num_list.append(count+1)

    features = ['steps', 'mean', 'standard deviation', 'minumum', 'maximum', 'peak-to-peak']
    dictionary = {'steps':step_num_list, "mean":mean_list, 'standard deviation':std_list,
                  'minumum':min_list, 'maximum':max_list, 'peak-to-peak':pp_list}
    df = pd.DataFrame(dictionary, columns = features)
    df.to_csv(name + '_statistical_results.csv', index = False, header = True)
    counter += 1

success = "Statistical analysis files created succesfully!"
self.result_list.addItem(success)
```

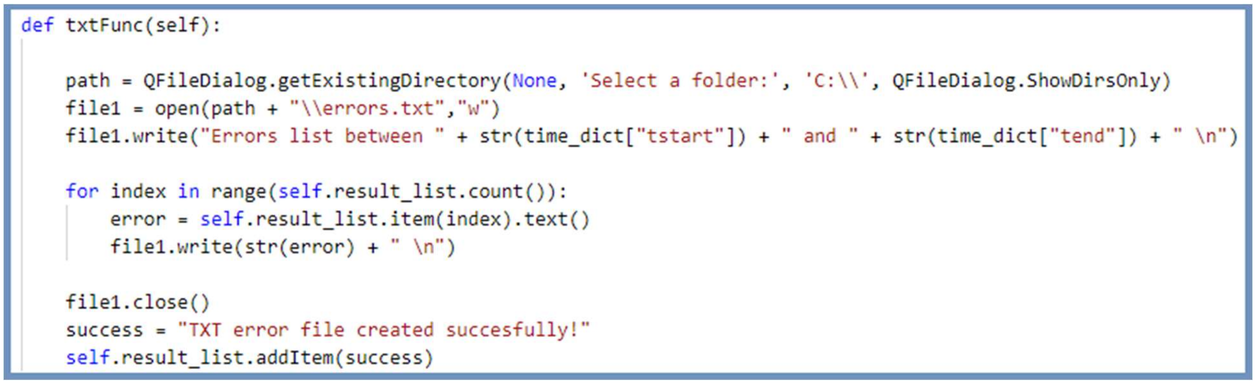
3. TXT button:





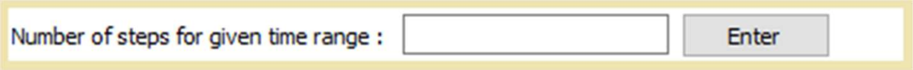
```
self.report_qlabel = QLabel("Write Error Report into TXT File")
self.txt_button = QPushButton("TXT", self)
self.txt_button.clicked.connect(self.txtFunc)
```

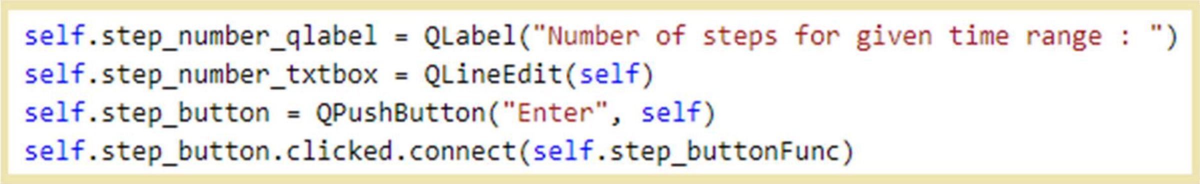
txtFunc duty is very basic. When you click it, again folder discovery opens and you choose suitable for txt file which includes all of the rows in the result list.



```
def txtFunc(self):

    path = QFileDialog.getExistingDirectory(None, 'Select a folder:', 'C:\\', QFileDialog.ShowDirsOnly)
    file1 = open(path + "\\errors.txt","w")
    file1.write("Errors list between " + str(time_dict["tstart"]) + " and " + str(time_dict["tend"]) + " \n")

    for index in range(self.result_list.count()):
        error = self.result_list.item(index).text()
        file1.write(str(error) + " \n")

    file1.close()
    success = "TXT error file created succesfully!"
    self.result_list.addItem(success)
```

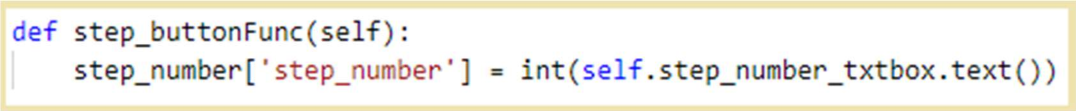
4. Step number txt box:





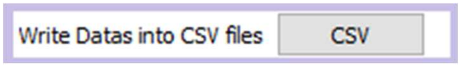
```
self.step_number_qlabel = QLabel("Number of steps for given time range : ")
self.step_number_txtbox = QLineEdit(self)
self.step_button = QPushButton("Enter", self)
self.step_button.clicked.connect(self.step_buttonFunc)
```

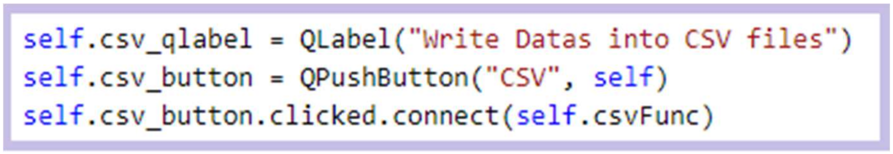
When you write something into txt box and click enter, it will assign input as step number. Lets have a look to the step_buttonFunc which is also very short and easy :



```
def step_buttonFunc(self):
    step_number['step_number'] = int(self.step_number_txtbox.text())
```

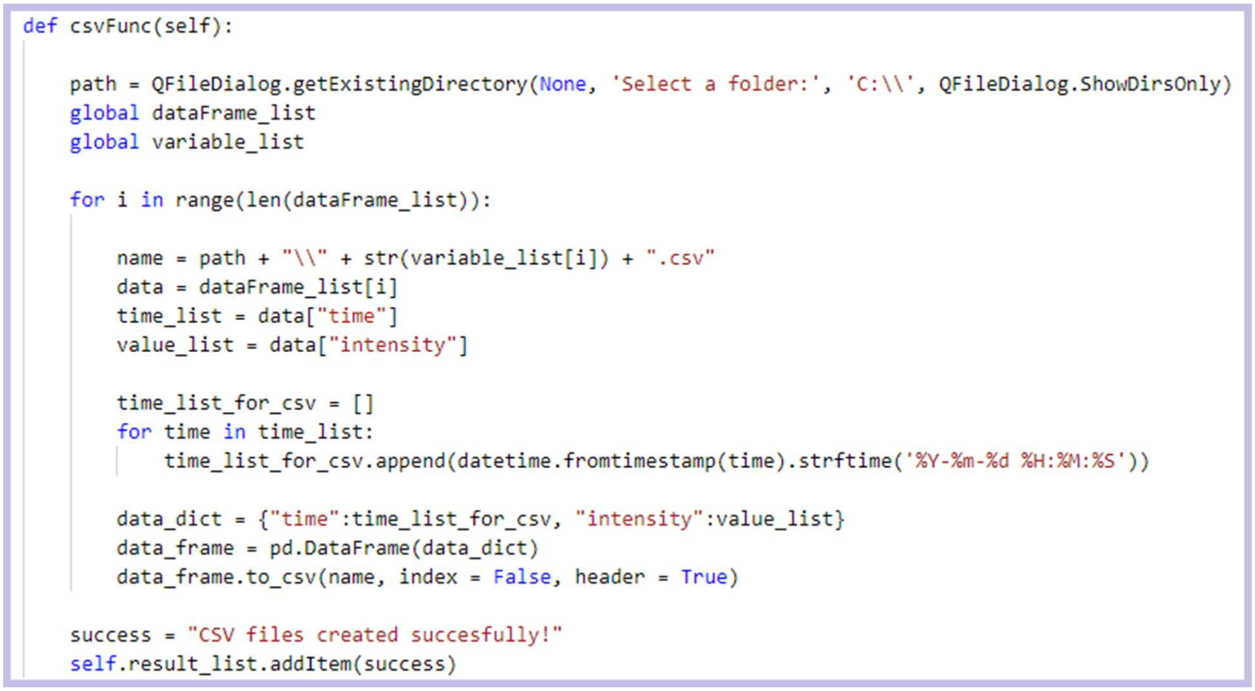
5. CSV button:





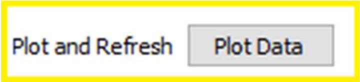
```
self.csv_qlabel = QLabel("Write Datas into CSV files")
self.csv_button = QPushButton("CSV", self)
self.csv_button.clicked.connect(self.csvFunc)
```

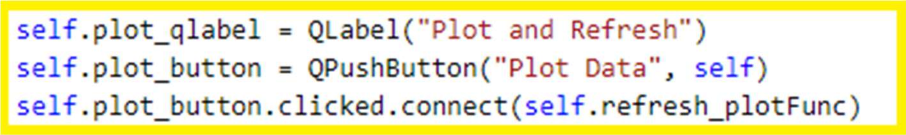
csvFunc also opens a folder discovery choose a path to save taken data as CSV file. It uses variable list as name of taken data during saving process. It also change time stamps to the readable form which is YEAR-MONTH-DAY HOUR-MINUTE-SECOND After its execution complete, success message can be seen in result list.



```
def csvFunc(self):

    path = QFileDialog.getExistingDirectory(None, 'Select a folder:', 'C:\\', QFileDialog.ShowDirsOnly)
    global dataFrame_list
    global variable_list

    for i in range(len(dataFrame_list)):

        name = path + "\\" + str(variable_list[i]) + ".csv"
        data = dataFrame_list[i]
        time_list = data["time"]
        value_list = data["intensity"]

        time_list_for_csv = []
        for time in time_list:
            time_list_for_csv.append(datetime.fromtimestamp(time).strftime('%Y-%m-%d %H:%M:%S'))

        data_dict = {"time":time_list_for_csv, "intensity":value_list}
        data_frame = pd.DataFrame(data_dict)
        data_frame.to_csv(name, index = False, header = True)

    success = "CSV files created succesfully!"
    self.result_list.addItem(success)
```

6. Plot Data button:



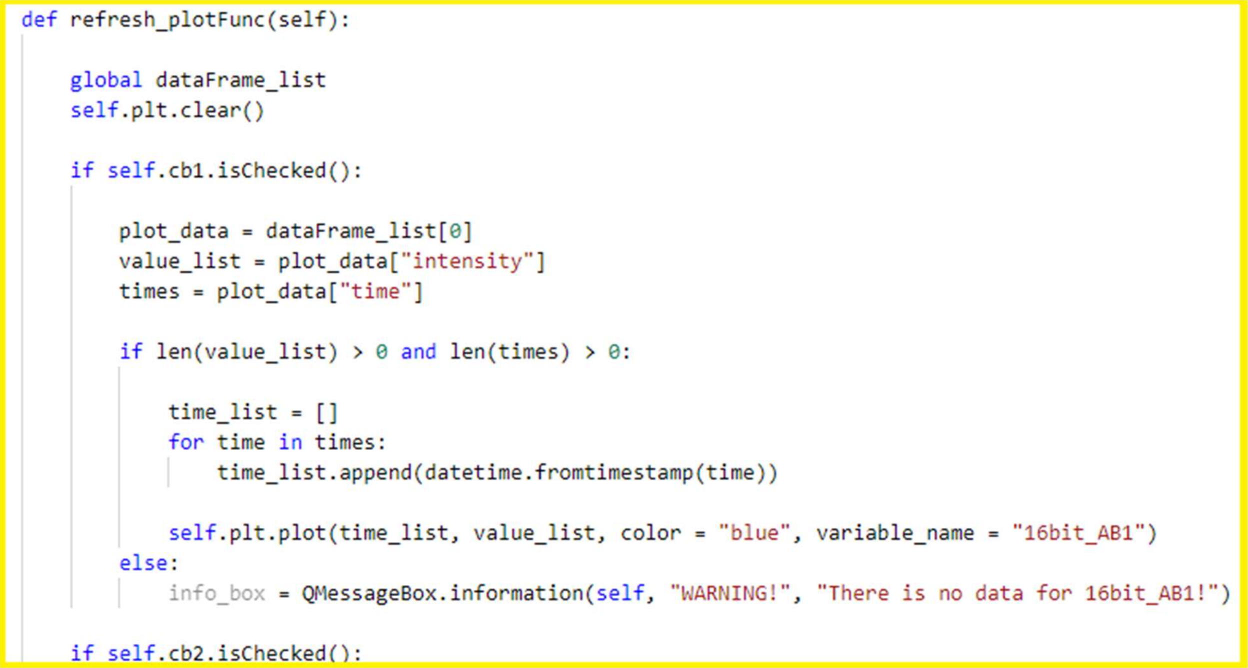


```
self.plot_qlabel = QLabel("Plot and Refresh")
self.plot_button = QPushButton("Plot Data", self)
self.plot_button.clicked.connect(self.refresh_plotFunc)
```

Refresh_plotFunc is little longer than others, but it repeats itself. I will try to explain one. When one of the check box is checked and clicked Plot Data button, it plots chosen variable. Data comes from data frames which is filled by getData. If there is no data it gives error such as

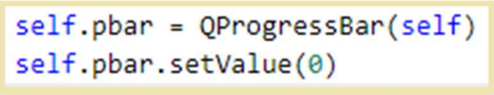
‘ There is no data for (chosen data frame name)!’

It continues with other data frames. Every time you clicked Plot Data, it clears plot and replot it again.



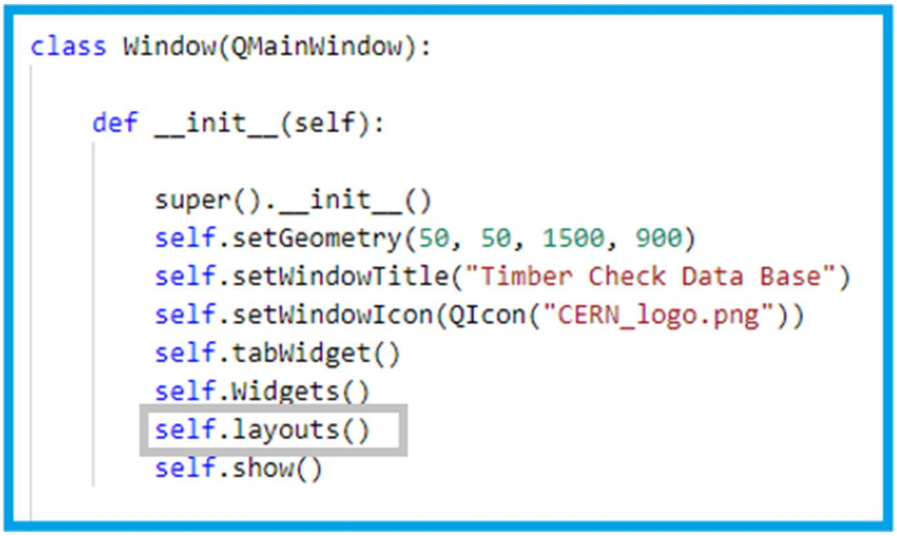
```
def refresh_plotFunc(self):

    global dataFrame_list
    self.plt.clear()

    if self.cb1.isChecked():

        plot_data = dataFrame_list[0]
        value_list = plot_data["intensity"]
        times = plot_data["time"]

        if len(value_list) > 0 and len(times) > 0:

            time_list = []
            for time in times:
                time_list.append(datetime.fromtimestamp(time))

            self.plt.plot(time_list, value_list, color = "blue", variable_name = "16bit_AB1")
        else:
            info_box = QMessageBox.information(self, "WARNING!", "There is no data for 16bit_AB1!")

    if self.cb2.isChecked():
```

Finally we have another function which is initial condition of progress bar in Widgets.



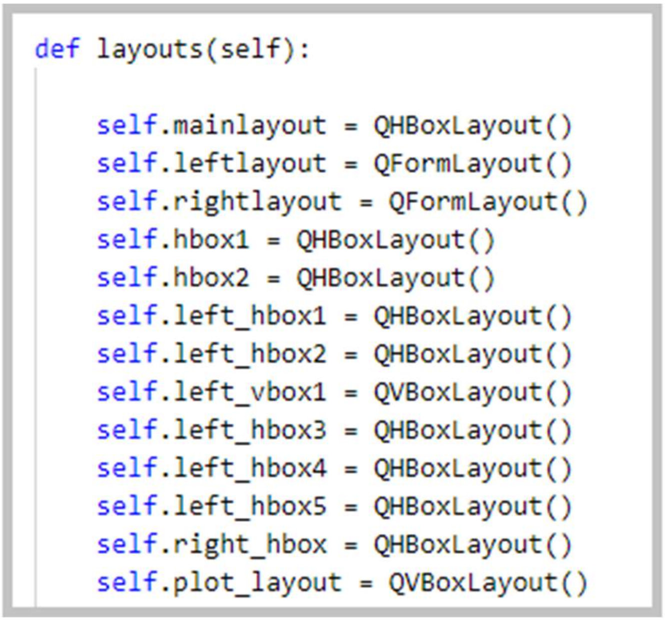
```
self.pbar = QProgressBar(self)
self.pbar.setValue(0)
```

LAYOUTS(skeleton of GUI)



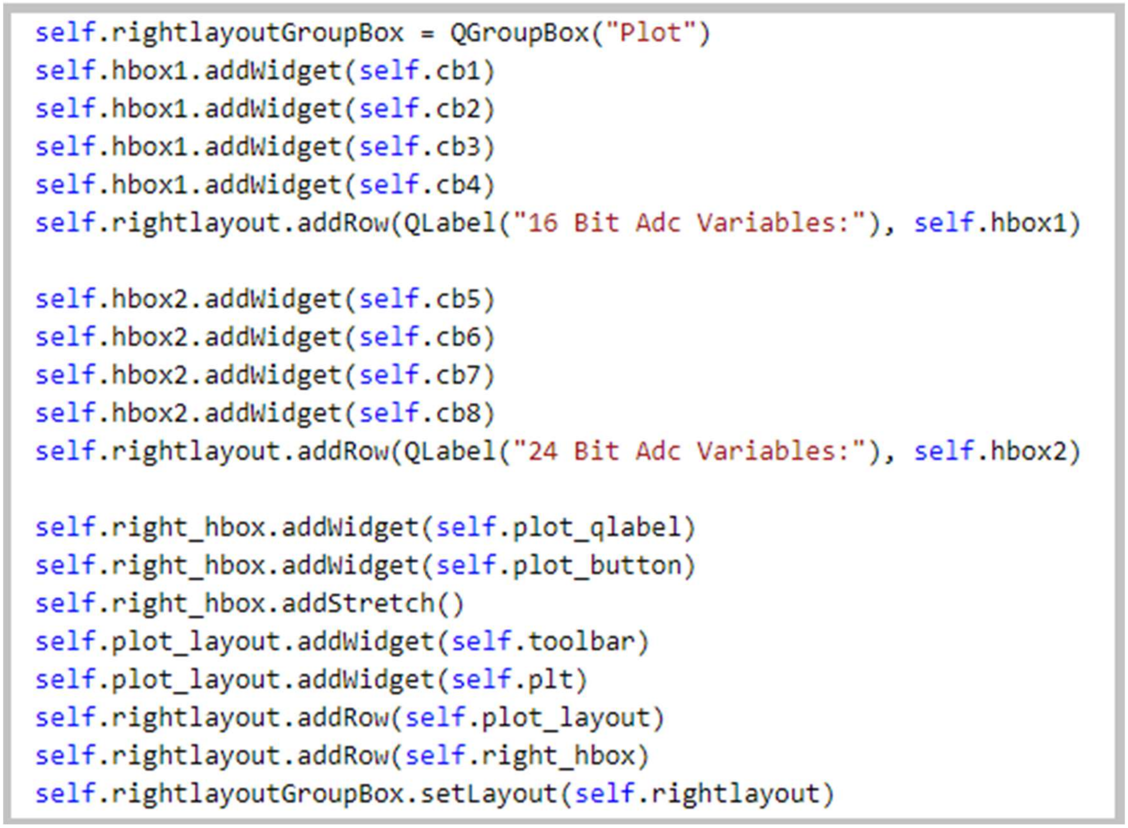
```
class Window(QMainWindow):

    def __init__(self):

        super().__init__()
        self.setGeometry(50, 50, 1500, 900)
        self.setWindowTitle("Timber Check Data Base")
        self.setWindowIcon(QIcon("CERN_logo.png"))
        self.tabWidget()
        self.Widgets()
        self.layouts()
        self.show()
```

Layouts function is an important function which basically visualization of GUI. GUI has one main layout which includes all other layouts. It is divided into 2 sides which are another 2 main layouts right and left. Right layout includes plot figure, figure navigation tool bar, check boxes for plots and plot button. Left includes rest. Lets have a look smaller parts of right and left layouts.



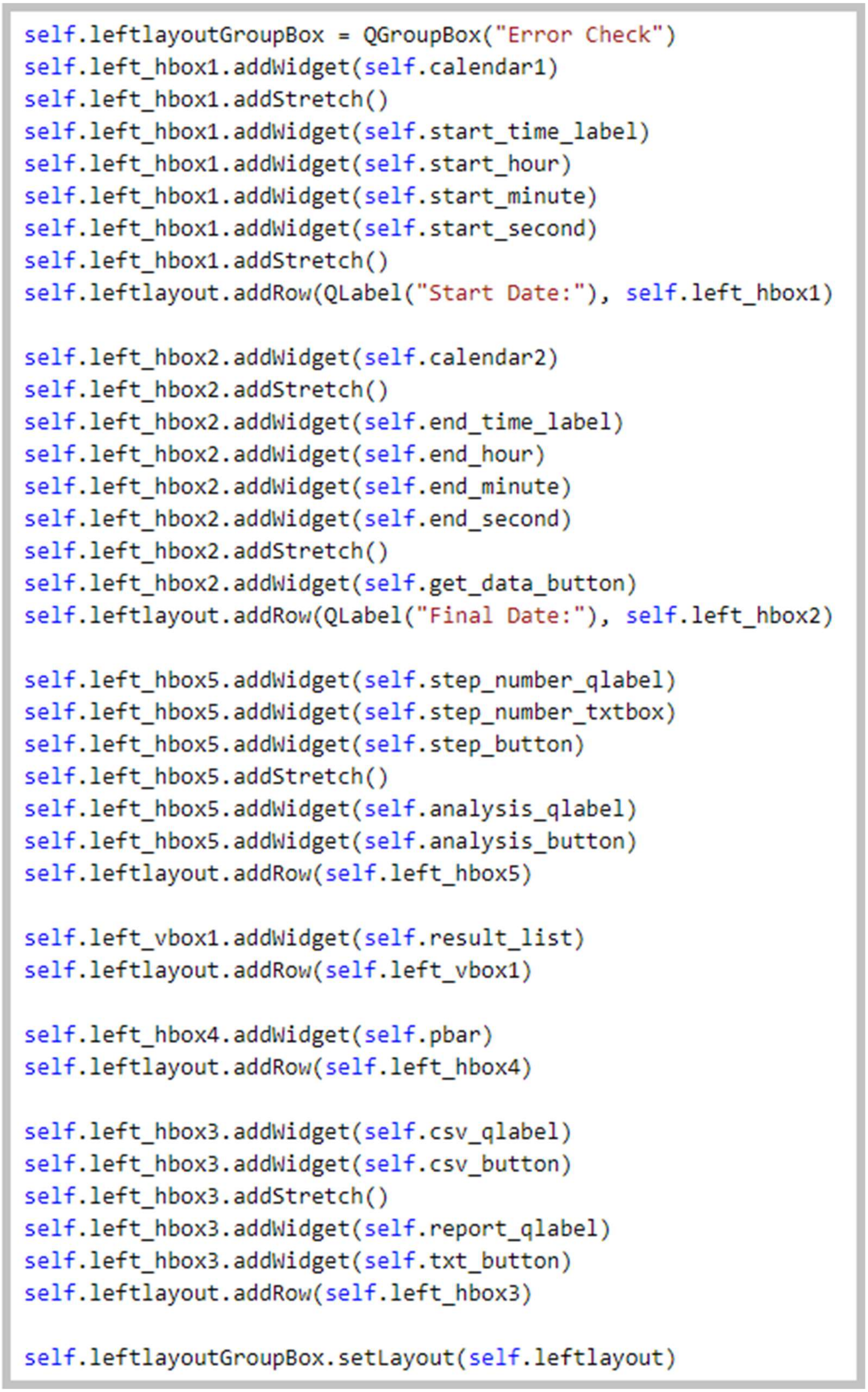
```
def layouts(self):

    self.mainlayout = QHBoxLayout()
    self.leftlayout = QFormLayout()
    self.rightlayout = QFormLayout()
    self.hbox1 = QHBoxLayout()
    self.hbox2 = QHBoxLayout()
    self.left_hbox1 = QHBoxLayout()
    self.left_hbox2 = QHBoxLayout()
    self.left_vbox1 = QVBoxLayout()
    self.left_hbox3 = QHBoxLayout()
    self.left_hbox4 = QHBoxLayout()
    self.left_hbox5 = QHBoxLayout()
    self.right_hbox = QHBoxLayout()
    self.plot_layout = QVBoxLayout()
```

Right layout has a group box feature to cumulate small right layouts. For example after define hbox1, which represents a horizontal box, I added check boxes into it with using .addWidget() function. After that hbox1 added into rightlayout with .addRow() function. Same process has written for hbox2 and for other layouts. After adding widgets into label process finished, group box set by rightlayout which includes all other small layouts and widgets.



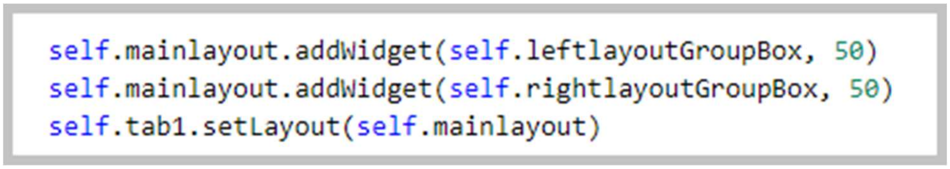
```
self.rightlayoutGroupBox = QGroupBox("Plot")
self.hbox1.addWidget(self.cb1)
self.hbox1.addWidget(self.cb2)
self.hbox1.addWidget(self.cb3)
self.hbox1.addWidget(self.cb4)
self.rightlayout.addRow(QLabel("16 Bit Adc Variables:"), self.hbox1)

self.hbox2.addWidget(self.cb5)
self.hbox2.addWidget(self.cb6)
self.hbox2.addWidget(self.cb7)
self.hbox2.addWidget(self.cb8)
self.rightlayout.addRow(QLabel("24 Bit Adc Variables:"), self.hbox2)

self.right_hbox.addWidget(self.plot_qlabel)
self.right_hbox.addWidget(self.plot_button)
self.right_hbox.addStretch()
self.plot_layout.addWidget(self.toolbar)
self.plot_layout.addWidget(self.plt)
self.rightlayout.addRow(self.plot_layout)
self.rightlayout.addRow(self.right_hbox)
self.rightlayoutGroupBox.setLayout(self.rightlayout)
```

Same processes can be seen in leftlayout lines. First there is a group box created for includes all small layouts and set main layout. There is an additional function in the code which is addStretch(). Its duty is prop up the widgets left or right. If it comes after line, it will prop up to the left, or vice versa. Again after add widgets into small horizontal and vertical boxes, they are added as a row into main left layout. Left layout set into group box to complete right side of GUI.



```python
self.leftlayoutGroupBox = QGroupBox("Error Check")
self.left_hbox1.addWidget(self.calendar1)
self.left_hbox1.addStretch()
self.left_hbox1.addWidget(self.start_time_label)
self.left_hbox1.addWidget(self.start_hour)
self.left_hbox1.addWidget(self.start_minute)
self.left_hbox1.addWidget(self.start_second)
self.left_hbox1.addStretch()
self.leftlayout.addRow(QLabel("Start Date:"), self.left_hbox1)

self.left_hbox2.addWidget(self.calendar2)
self.left_hbox2.addStretch()
self.left_hbox2.addWidget(self.end_time_label)
self.left_hbox2.addWidget(self.end_hour)
self.left_hbox2.addWidget(self.end_minute)
self.left_hbox2.addWidget(self.end_second)
self.left_hbox2.addStretch()
self.left_hbox2.addWidget(self.get_data_button)
self.leftlayout.addRow(QLabel("Final Date:"), self.left_hbox2)

self.left_hbox5.addWidget(self.step_number_qlabel)
self.left_hbox5.addWidget(self.step_number_txtbox)
self.left_hbox5.addWidget(self.step_button)
self.left_hbox5.addStretch()
self.left_hbox5.addWidget(self.analysis_qlabel)
self.left_hbox5.addWidget(self.analysis_button)
self.leftlayout.addRow(self.left_hbox5)

self.left_vbox1.addWidget(self.result_list)
self.leftlayout.addRow(self.left_vbox1)

self.left_hbox4.addWidget(self.pbar)
self.leftlayout.addRow(self.left_hbox4)

self.left_hbox3.addWidget(self.csv_qlabel)
self.left_hbox3.addWidget(self.csv_button)
self.left_hbox3.addStretch()
self.left_hbox3.addWidget(self.report_qlabel)
self.left_hbox3.addWidget(self.txt_button)
self.leftlayout.addRow(self.left_hbox3)

self.leftlayoutGroupBox.setLayout(self.leftlayout)
```

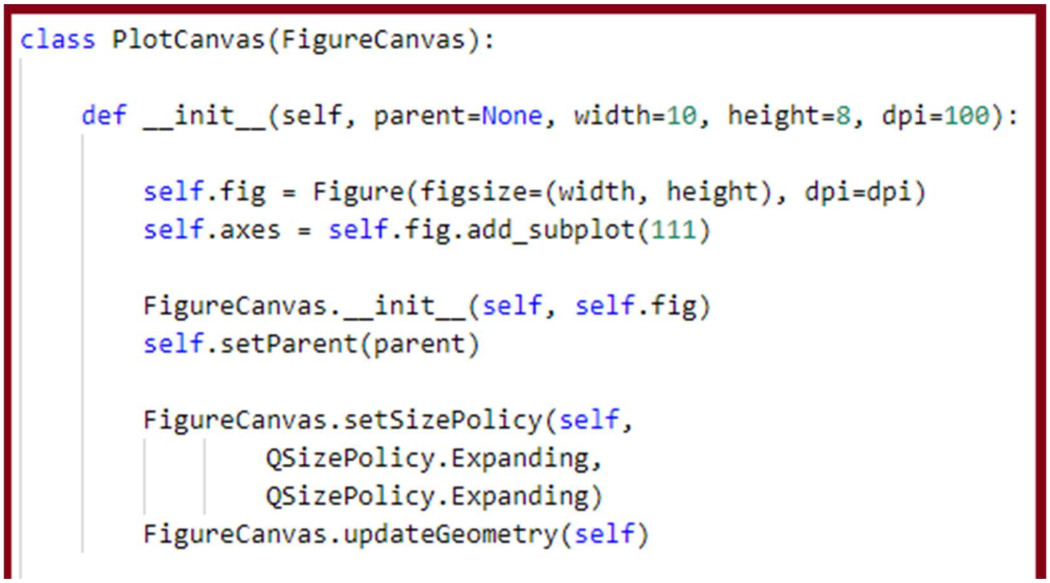
After filling process, right group box and left group box share main layout and main layout added into tab1.



```
self.mainlayout.addWidget(self.leftlayoutGroupBox, 50)
self.mainlayout.addWidget(self.rightlayoutGroupBox, 50)
self.tab1.setLayout(self.mainlayout)
```

PLOT CLASS

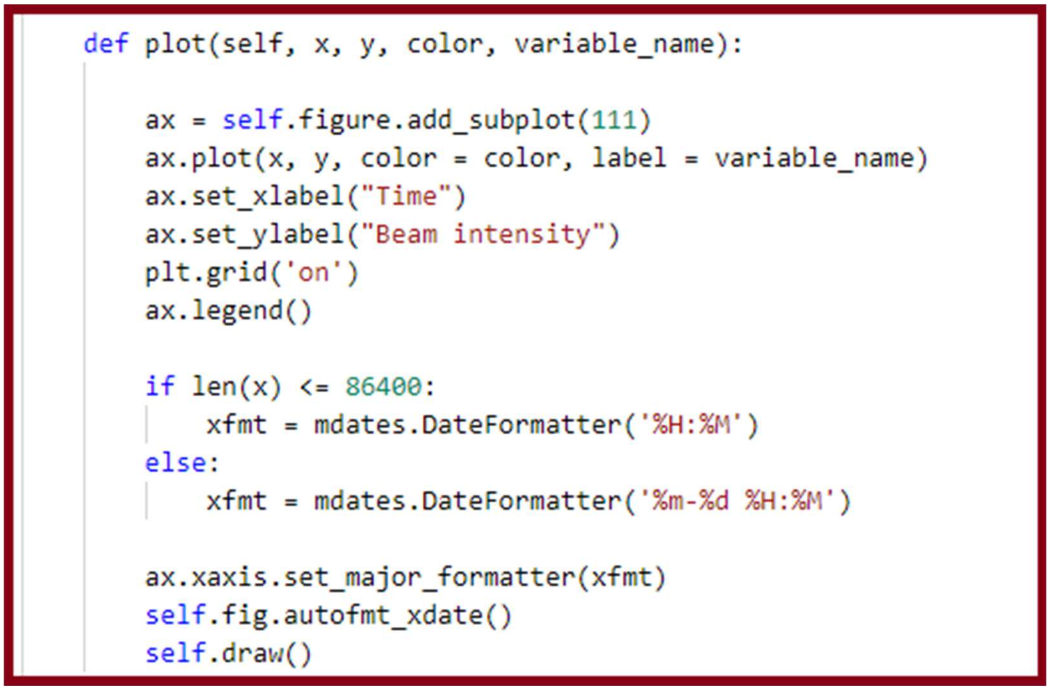
Plot class stands for plot and figure which is special for PyQt5. Figure width and height can be change by changing width and height variables in the initializer.



```
class PlotCanvas(FigureCanvas):

    def __init__(self, parent=None, width=10, height=8, dpi=100):

        self.fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = self.fig.add_subplot(111)

        FigureCanvas.__init__(self, self.fig)
        self.setParent(parent)

        FigureCanvas.setSizePolicy(self,
                QSizePolicy.Expanding,
                QSizePolicy.Expanding)
        FigureCanvas.updateGeometry(self)
```

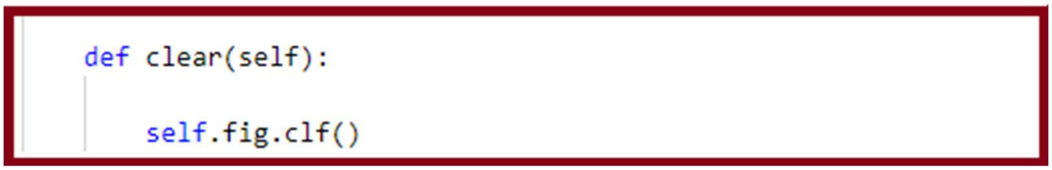
There are 2 functions which have specially written for a reason.

1. Plot function includes plot features. For example; color of plot, xlabel, ylabel, grid feature. This features can be change with changing this function.



```
def plot(self, x, y, color, variable_name):

    ax = self.figure.add_subplot(111)
    ax.plot(x, y, color = color, label = variable_name)
    ax.set_xlabel("Time")
    ax.set_ylabel("Beam intensity")
    plt.grid('on')
    ax.legend()

    if len(x) <= 86400:
        xfmt = mdates.DateFormatter('%H:%M')
    else:
        xfmt = mdates.DateFormatter('%m-%d %H:%M')

    ax.xaxis.set_major_formatter(xfmt)
    self.fig.autofmt_xdate()
    self.draw()
```

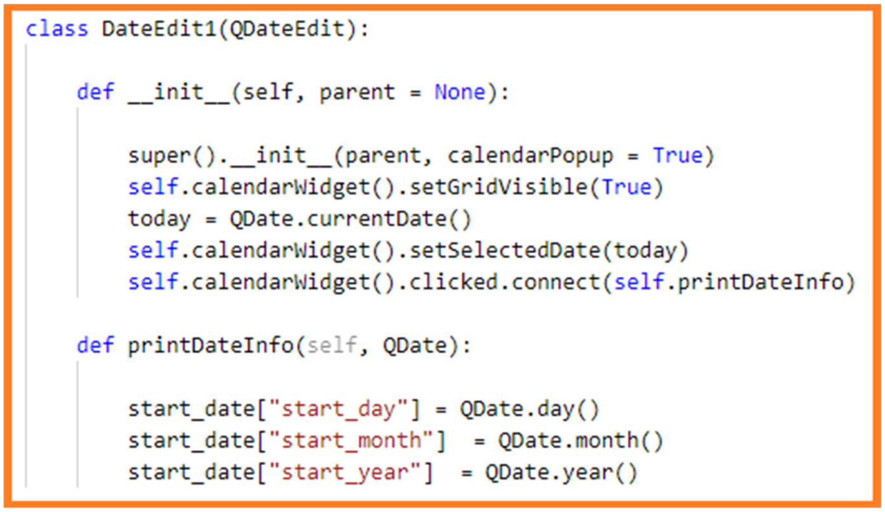
2. Clear function is basically cleans figure.



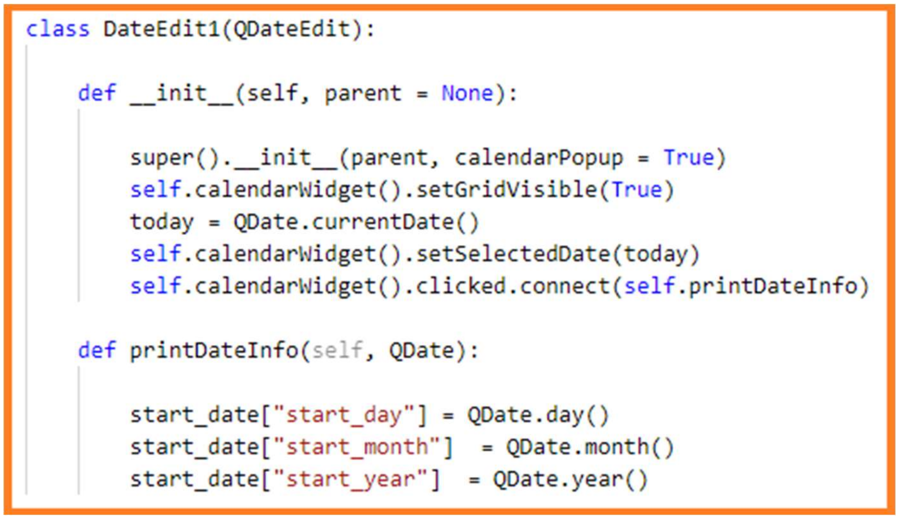
```
def clear(self):

    self.fig.clf()
```

DATE EDIT CLASS

Date edit class stands for calendar to choose date. It initialize with using its module. Today variable takes todays date and it initialize calendar today’s date with using setSelectedDate() function. When if calendar clicked, it connects printDateInfo() function. This function acquires date information from calendar and it saves into date dictionary. Because there 2 different dates, there are 2 different calendar classes.

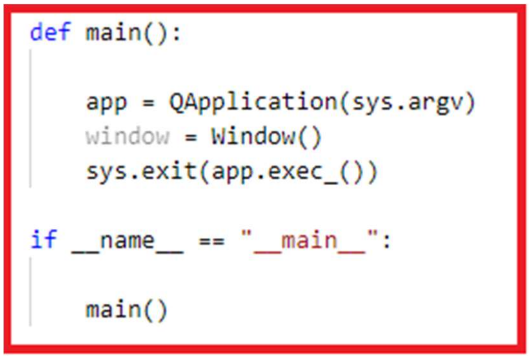


```
class DateEdit1(QDateEdit):

    def __init__(self, parent = None):

        super().__init__(parent, calendarPopup = True)
        self.calendarWidget().setGridVisible(True)
        today = QDate.currentDate()
        self.calendarWidget().setSelectedDate(today)
        self.calendarWidget().clicked.connect(self.printDateInfo)

    def printDateInfo(self, QDate):

        start_date["start_day"] = QDate.day()
        start_date["start_month"]  = QDate.month()
        start_date["start_year"]  = QDate.year()
```



```
class DateEdit1(QDateEdit):

    def __init__(self, parent = None):

        super().__init__(parent, calendarPopup = True)
        self.calendarWidget().setGridVisible(True)
        today = QDate.currentDate()
        self.calendarWidget().setSelectedDate(today)
        self.calendarWidget().clicked.connect(self.printDateInfo)

    def printDateInfo(self, QDate):

        start_date["start_day"] = QDate.day()
        start_date["start_month"]  = QDate.month()
        start_date["start_year"]  = QDate.year()
```

Rest of the code stands for main function which triggers window when code executed.



```
def main():

    app = QApplication(sys.argv)
    window = Window()
    sys.exit(app.exec_())

if __name__ == "__main__":

    main()
```