QF205 G2 Team 3

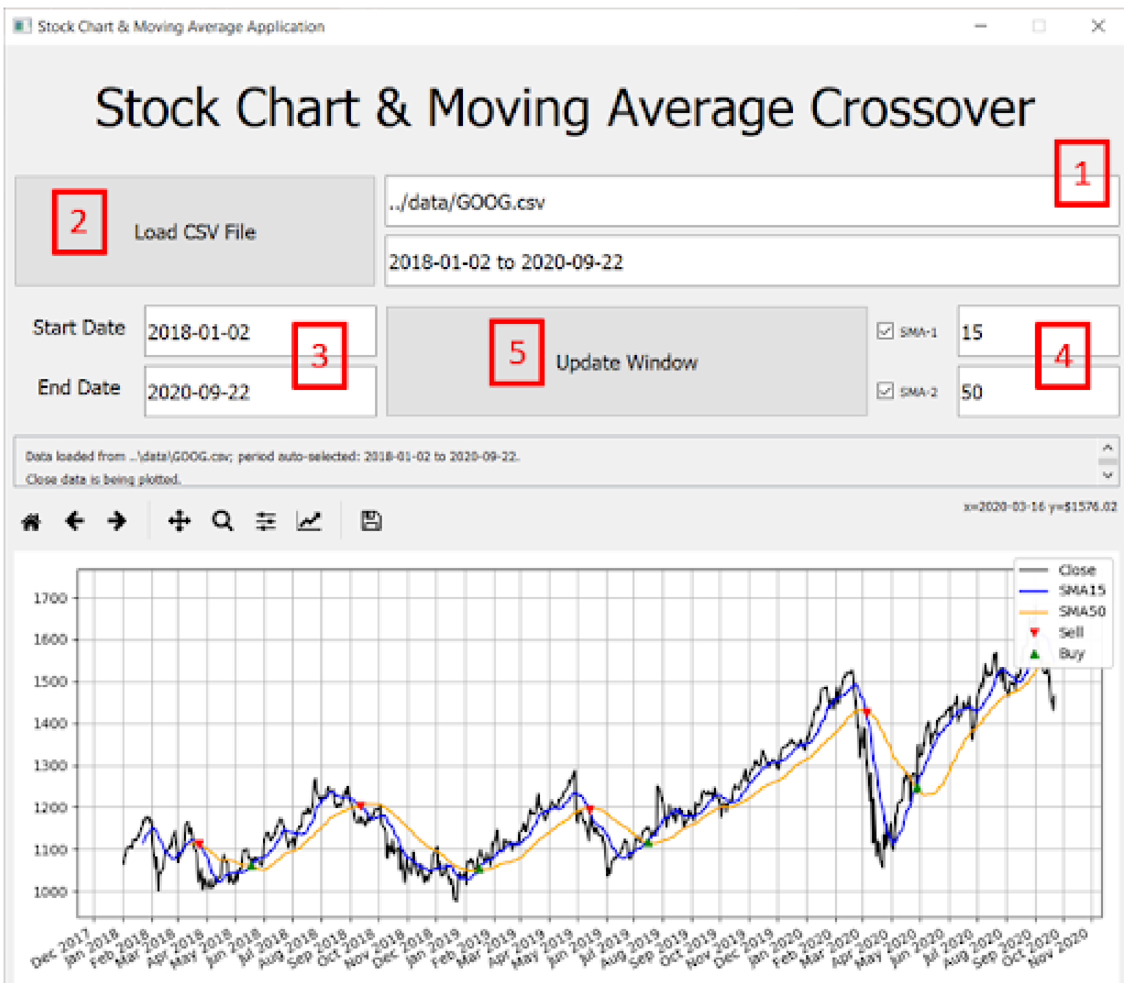
Yusnoveri Calvin, Lim Zhi Xin Casper, Pang Bo Hao Edwin, Lim Wei Yang,Yang Jiaxin

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

November 18, 2020

1 User Manual

1.1 Running the Application from `.exe` (Windows Only)



5 Steps of running the application from `.exe` file (found in `dist` folder):

(Recommended for Users who just want to know how to run and use the application)

0. Go to `dist` folder, double click the `app.exe`
1. Input the location of the stock data file here. In this case we have a relative path that directs to the `data` folder in StockChartApplication, where our sample data file, `GOOG.csv` is located. (`.` means the current folder where `app.exe` is located, `..` means the parent folder (one folder up) from where `app.exe` is located)

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

2. Loads the information in the stock data file into the application by pressing `Load CSV File`
3. The date range to be viewed can be changed here. (format: YYYY-MM-DD).
4. The number of days used to compute the simple moving averages can be changed and toggled here.
5. Press `Update Window` Button to plot the stock price, the two SMAs, the location of crossovers, and the signal presented by the crossover.

1.2 Running the Application using Python shell

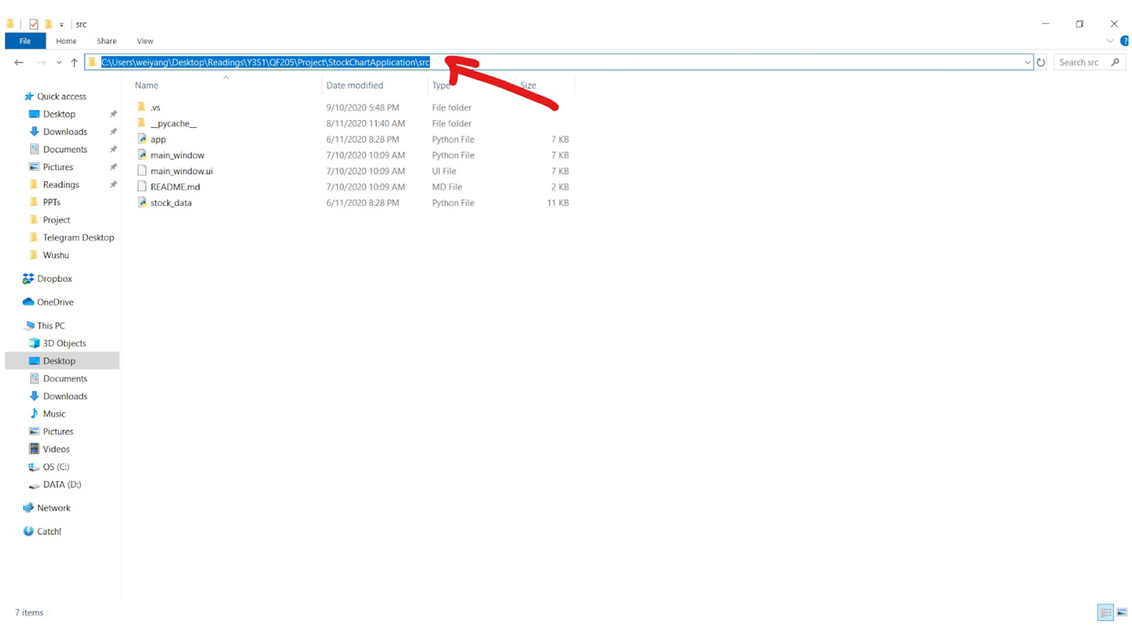
4 Steps of running the application using

(Recommended for Users who also want to know how to develop the application)

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

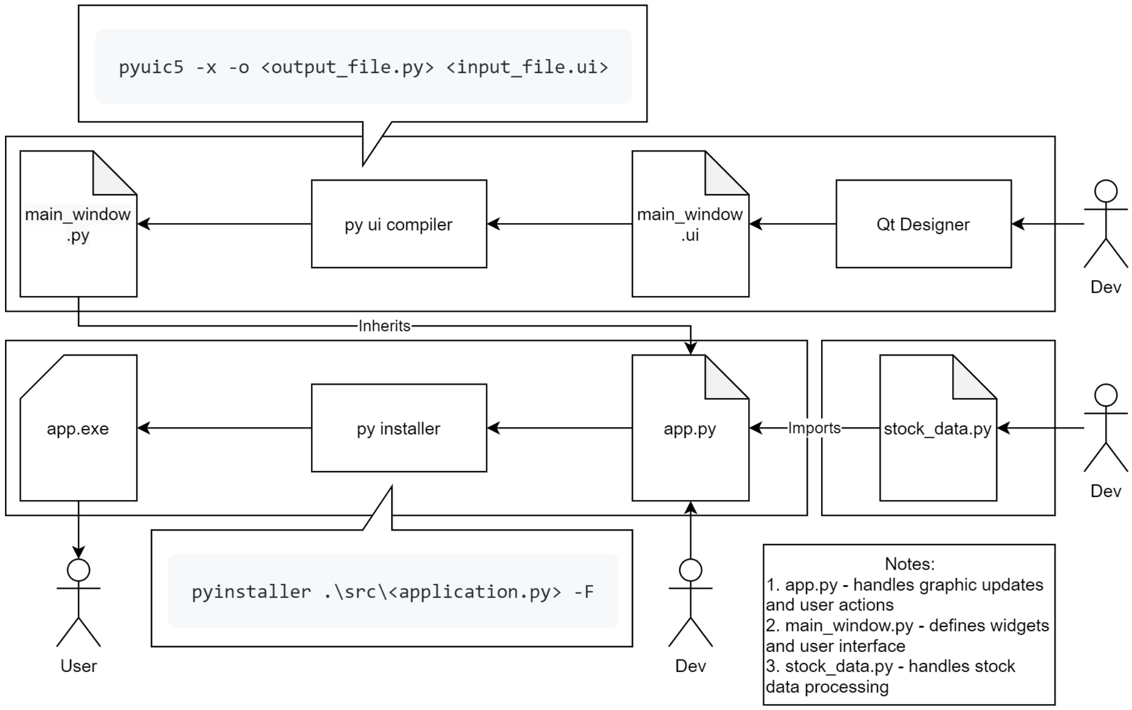
   - (Windows) https://repo.anaconda.com/archive/Anaconda3-2020.07-Windows-x86_64.exe
   - (Mac OS) https://repo.anaconda.com/archive/Anaconda3-2020.07-MacOSX-x86_64.pkg

1. Go to the `src` folder in the StockChartApplication, click on the address bar, and copy the directory location



2. Search for and run the command line shell of your python distributor. In this case we will search for and run `Anaconda` Prompt.
3. In the command line shell, type `cd` and then paste the directory address of the `src` folder then press `Enter` to change the directory
4. Type `python app.py` and press `Enter` to run the Stock Chart Application. The application window will appear, simply follow the previous section's instruction to use the application.

1.3 Application Components



This application is made up of 3 components:

1. `main_window.py`
2. `stock_data.py`
3. `app.exe`

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

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

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

The next section of the report will discuss how each of these 3 components are created (Please refer to the section headers to find the relevant part's information).

- For those **unfamiliar** with Python, it is recommended to read Section 2 and 3 about Python Basics and Packages.
- For those already **familiar** with Python, skip to Section 4, 5 and 6 corresponding to the 3 components.

2 Python Basics

2.1 Variables

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

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

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

```
apple 1 2
```

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

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

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

2.2 Data types

Common Data types that are used in the projects includes.

Text type - `String (str)`

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

Sequence Type - `list` - `range`

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

2.2.1 Text type

`String(str)`

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

Examples of String

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

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

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

2.2.2 Numeric type

Integer (`Int`)

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

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

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

Wrong examples:

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

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

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

Float (`float`)

Float are Floating point numbers that contains one or more decimals.

Underscore are allowed between float groupings.

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

```
<class 'float'>
```

`Int` can be converted to `float`, vice versa.

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

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

Wrong examples:

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

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

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

2.2.3 Sequence type

List `List` is a collection which is ordered and changeable.

A single list may contain DataTypes like Integers, Strings, as well as Objects. `List` are mutable, and hence, they can be altered even after their creation.

The elements in a list are indexed according to a definite sequence and the indexing of a `list` is done with 0 being the first index.

[6]:
```
list_example= ['a',1,'c']

print(list_example)
print(list_example[0])
```

```
['a', 1, 'c']
a
```

We can find the length of the `list` by using `len()`

[7]:
```
print(len(list_example))
```

```
3
```

2.3 range

The `range()` function is used to generate a sequence of numbers over time.

We can create a start, stop and step while using the `range()` function.

The syntax of range function is : `range(start,stop,step)`

Start is where the range start (inclusive), stop is where the range stops at (exclusive), step determines each increment

[8]:
```python
print(range(5))
print(range(2,5))
print(range(1,10,2))
```

```
range(0, 5)
range(2, 5)
range(1, 10, 2)
```

2.4 Type conversion

We can convert one type of data to another with `int()`, `float()` and `str()`.

[10]:
```python
x = 1
print(x,type(x))

x = str(x)
print(x,type(x))

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

```
1 <class 'int'>
1 <class 'str'>
1.0 <class 'float'>
```

However, converting data type to an invalid literal would give a `ValueError`.

[10]:
```python
x = 'apple'
x = int(x)
print(type(x))
```

```
␣
↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-10-7753f23209e0> in <module>
         1 x = 'apple'
   ----> 2 x = int(x)
         3 print(type(x))


       ValueError: invalid literal for int() with base 10: 'apple'
```

2.5 Concatenation and Operations

Concatenate by using (+) operator - Multiply using (*) operator - Minus using (-) operator - Divide using (/) operator - Modulus using (%) operator - to find remainder - Exponentiation using (*) operator - Floor division using (//) operator

[11]:
```
x = 'hi'
y = 'bye'
z = 2
print(x+y)
print(x*2)
x = 3
y = 2
print(x+y)
print(x-y)
print(x*y)
print(x/y)
print(x%y)
print(x**y)
print(x//y)
```

```
hibye
hihi
5
1
6
1.5
1
9
1
```

2.6 Operator precedence

Operator have different precedence. It is important to take note of the precedence of the operator in order to accurately represent your equations. We can also use the Brackets to represent the precedence.

This is the operator precedence from Lowest to highest precedence.

1. +,-
2. *,/,//,%
3. **

Examples of Operator precedence

[11]:
```
print(3**2 * 3 +1)

print(2**2**3) # the precedence occurs from the left to right in for
 ↪Exponentiation
```

```
print((20+2*4) / 2)
```

```
28
256
14.0
```

2.7 Logical operators

A logical operator is a symbol or word used to connect two or more expressions such that the value of the compound expression produced depends only on that of the original expressions and on the meaning of the operator. Common logical operators include AND, OR, and NOT.

`and` Returns True if all of the statements is true.

`or` Returns True if one of the statements is true.

`not` Reverse the result, returns False if the result is true.

Examples:

[14]:
```
x = 5
y = 5
z = 5
print(x < 3 and x < 10)
print(y > 3 or y < 10)
print (not(z < 4))
```

```
False
True
True
```

2.8 Assignment Statement

We can (re)bind names to values and also modify attributes or mutable objects using assignment statements.

When assigning the variable x to a different value, we can see that variable x id is also changed by using the in-build function `id()`.

However, it is possible that two objects with non-overlapping lifetimes can have the same `id()` value.

[16]:
```
x=1
print(id(x))
x=3
print(id(x))
print(x)
```

```
140734888321424
140734888321488
3
```

2.9 Comments

We can use write comments on our code by using the hash character(#).

A comment signifies the end of the logical line most of the time, unless implicit line joining rules are invoked which would result in an syntax error.

[21]:
```
x = 1 # x is being assigned the value 1
```

Example of implicit line joining:

[20]:
```
x = 1 \ #explicit line joining
print(x)
```

```
  File "<ipython-input-20-bda2e21dbab5>", line 1
    x = 1 \ #explicit line joining
                                  ^
SyntaxError: unexpected character after line continuation character
```

2.10 Indentation

Indentations begins at the start of the logical line which is used to determine the grouping of the statements.

Indentation are rejected if there are an inconsistent mix of tabs and spaces. This causes a Tab Error.

[22]:
```
if 1<4:
    print('yes')
else:
    print('no')
```

```
yes
```

Bad Example of inconsistent mix of tabs and spaces:

[15]:
```
if 1>4:
    print('yes')
else:
print('no')
```

```
  File "<ipython-input-15-a7c8ff9be281>", line 4
    print('no')
        ^
IndentationError: expected an indented block
```

2.11 Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

You can pass data, known as parameters, into a function. A function can return data as a result.

We use `def` to define a function. `def` is followed by the function name and a parentheses and a colon punctuation.

This is a basic example of a function that prints a string:

[16]:
```
def my_function():
  print("Hello from a function")
```

By typing `my_function()`, this would call the function and run the action it in which is to print

Hello from a function

[17]:
```
my_function()
```

```
Hello from a function
```

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

We use the return function to send back an output.

Examples of Parameters: `Sum(x,y)`

A parameter is the variable listed inside the parentheses in the function definition.

Examples of arguments: `Sum(1,2)`

An argument is the value that is sent to the function when it is called.

Example of how a function with arguments and parameter.

1. Create the function
2. Define the parameters
3. Action in the function
4. The desired output

[18]:
```
def sum(x,y): # function name is sum with parameters x and y
    z= x + y # the action of the function
    return z # output of the function
```

5. call the function and input the arguments.

```
[19]: total_sum = sum(1,2) #calling the function and input of arguments 1 and 2
print(total_sum)
```

```
3
```

It is good practice to make sure you have the same number of arguments and parameter.

If we are missing an argument, there would be a `TypeError` stating that the function is missing a required positional argument.

```
[29]: total_sum = sum(1) #missing an argument
print(total_sum)
```

```
↪---------------------------------------------------------------------------

    TypeError                                 Traceback (most recent call
↪last)

    <ipython-input-29-64f0fd9ad24f> in <module>
----> 1 total_sum = sum(1)
      2 print(total_sum)


    TypeError: sum() missing 1 required positional argument: 'y'
```

If we input more arguments than the parameter, we would also get a `TypeError`stating the function `sum()` takes 2 positional arguments but 3 were given.

```
[30]: total_sum = sum(1,2,3) #extra arguments
print(total_sum)
```

```
↪---------------------------------------------------------------------------

    TypeError                                 Traceback (most recent call
↪last)

    <ipython-input-30-5c272ec8fe3b> in <module>
----> 1 total_sum = sum(1,2,3)
      2 print(total_sum)


    TypeError: sum() takes 2 positional arguments but 3 were given
```

Arbitrary Arguments, `*args` If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a `tuple` of arguments, and can access the items accordingly:

[31]:
```
def name(*friend):
  print("The best friend is " + friend[2])

name("Bob", "Tom", "Eden")
```

```
The best friend is Eden
```

Keyword Arguments You can also send arguments with the `key = value` syntax.

This way the order of the arguments does not matter.

[32]:
```
def fruits(fruit3, fruit2, fruit1):

  print("My favourite fruit is " + fruit2)

fruits(fruit1 = "apple", fruit2 = "pear", fruit3 = "orange")
```

```
My favourite fruit is pear
```

Arbitrary Keyword Arguments `**kwargs` If you do not know how many keyword arguments that will be passed into your function, add two asterisk: `**` before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly:

[33]:
```
%reset -f

def fruit_function(**fruit):
  print("My favourite fruit is " + fruit["fruit2"])


fruit_function(fruit1 = "apple", fruit2 = "pear")
```

```
My favourite fruit is pear
```

2.12 Loops

Loops allow the code to iterate and repeat itself within set parameters. Loops allow you to automate processes within the program using a few lines of code.

Loops typically come in two forms: 1. the `For` statement 2. the `While` statement

The for statement iterates over the elements of a sequence, such as a string, `list`, `range`, or other iterable objects. With the `for` loop you can automatically execute the code once for each item in the sequence.

An example of a `for` loop would be:

[35]:
```python
dateList = ["01-01-2000", "31-12-2020", "05-06-2018", "04-10-2010"]
for date in dateList:
    print(date)
```

```
01-01-2000
31-12-2020
05-06-2018
04-10-2010
```

The date between `for` and `in` in the code is arbitary, and can be replaced by any other letter or string.

In this case date is known as the target_list, while the actual list of dates is known as the expression_list. The target list can also have multiple values matching the number of values within the expression_list such as:

[36]:
```python
for (i,j) in [(1,2), (3,4), (5,6)]:
    print(i)
    print(j)
```

```
1
2
3
4
5
6
```

the number of values in the target_list not matching the number of values in the expression_list will give an error:

[37]:
```python
for (i,j,k) in [(1,2), (3,4), (5,6)]:
    print(i)
    print(j)
    print(k)
```

```
              ␣
↪---------------------------------------------------------------------------

    ValueError                                Traceback (most recent call␣
↪last)

    <ipython-input-37-0e03c00705c4> in <module>
  ----> 1 for (i,j,k) in [(1,2), (3,4), (5,6)]:
        2     print(i)
        3     print(j)
        4     print(k)
```

```
ValueError: not enough values to unpack (expected 3, got 2)
```

For loops can also be used in conjunction with the range() function such as in:

[38]:
```python
n = 5
for i in range(n):
    print(i)
```

```
0
1
2
3
4
```

On the other hand, the `while` statement repeatedly tests an expression and executes the code for as long as the expression is true.

An example of a `while` loop would be:

[ ]:
```python
i = 1
while i < 6:
    print(i)
    i += 1
```

As long as `i` is less than 6, the code will print the current value of `i` before adding 1 to the value of `i`. When the value of `i` reaches 6 the loop will automatically end.

2.13 if/elif/else

`if…elif…else` statement is used for decision making in Python. If statement is represented by `if` and it will only be executed when the statement is true. Else if statement is represented by `elif` and it will only execute when the statement is true and if the previous statement was not true. Else statement is represented by `else` and it will only execute when all previous statement were not true. Note: the statements must have a boolean output and each statement blocks have to be indented.

Example:

[39]:
```python
a = 200
b = 33

if b > a: #if b is greater than a, python would run the action in the if␣
 ↪condition. The following code with continue to run
  print("b is greater than a")
elif a == b: #if a and b are equal and the previous condition is false, python␣
 ↪would run the action in the elif condition.
  print("a and b are equal")
else: # python would run the else condition if all previous conditions are false
  print("a is greater than b")
```

```
a is greater than b
```

There is also an alternative way to write if/else/elif statements that are one liner.

[40]:
```
a = 200
b = 33
if b > a: print("b is greater than a")
else: print("a is greater than b")
```

```
a is greater than b
```

More examples:

[45]:
```
dateIndex = 5

if dateIndex < 3: #10 is more than 3, therefore this statement is false
    print(dateIndex, 'less than 3');
elif dateIndex < 8: #since previous statement is false, it will land on elif␣
↪statement. Since 10 is more than 8, therefore it is false
    print(dateIndex, 'less than 8');
else: #since all previous statements is false, it will execute the else block
    print(dateIndex);
```

```
5 less than 8
```

Indentations are important in `if/else/elif` conditional statements. If the indentations are not properly used after declaring the `if/else/elif` statements, there will be an indentation error. dateIndex+1 = 5 statement wont give you an boolean output which will give you an error. The correct statement is dateIndex+1 == 5, which will give you either true or false.

Bad example:

[47]:
```
If dateIndex+1 = 5:
print(dateIndex);
else:
print(dateIndex);
```

```
      File "<ipython-input-47-6604c19b7ed8>", line 1
    If dateIndex+1 = 5:
               ^
SyntaxError: invalid syntax
```

2.14 Tuple

Tuple is one of the basic sequences. It is immutable and usually used to store collections of data. Tuple may be constructed in different number of ways: using a pair parentheses to denote the empty tuple: `()` Using a trailing comma for singleton tuple: 1, or `(1,)` Separating items with commas: 1, 2, 3 or `(1, 2, 3)` Using the tuple() built-in: tuple() or `tuple([1,2,3])` #tuple(iterable)

The constructor builds a tuple whose items are the same and in the same order as iterable's items. Iterable may be either a sequence, a container that supports iteration, or an iterator object. If iterable is already a tuple, it is returned unchanged. For example, `tuple('haha')` returns `('h', 'a', 'h', 'a')` and `tuple([a, b, c])` returns `(a, b, c)`. If no argument is given, the constructor creates a new empty tuple, (). Note: Comma is what makes a tuple, not the parentheses. Parentheses are only compulsory when constructing an empty tuple or when they are needed to avoid syntactic ambiguity. Example `f(z, y, x)` #this is a function with 3 arguments `f((z, y, x))` #this is a function with a 3-tuple as the only argument

[48]:
```
tuple('haha')
```

[48]:
```
('h', 'a', 'h', 'a')
```

[49]:
```
tuple('1')
```

[49]:
```
('1',)
```

2.15 F-strings

A new string formatting mechanism known as Literal String Interpolation or more commonly as `F-strings` (because of the leading f character preceding the string literal). The idea behind `f-strings` is to make string interpolation simpler. We would input variables in {} that will be replaced with their values.

Examples:

[50]:
```
name = 'bob'
age = 23
print(f"Hello, My name is {name} and I'm {age} years old.")
```

```
Hello, My name is bob and I'm 23 years old.
```

2.16 Try Except

The try and except block in Python is used to catch and handle exceptions. The `try` block lets you test a block for exceptions. The `except` block lets you catch and handle the exceptions if there are any.

Example:

[51]:
```
num1 = 1
num2 = 0
try: #Python will try to run the code in the try block
        print(num1/num2) #There is a division by 0 situation and python raise a␣
 ↪ZeroDivisionError
except ZeroDivisionError as e:#Except block catches the ZeroDivisionError␣
 ↪exception and run the except block
        print("You cannot divide by 0" ) #Python prints "You cannot divide by 0"
```

```
You cannot divide by 0
```

2.17 Assert Statement

The `assert` Statement: When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an AssertionError exception. If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError.

Examples:

We first have to `assert` a condition

```
[ ]: assert condition
```

We then set a conditional statement to immediately trigger an error if the condition is false. The `AssertionError()` is a function that is predetermined by the user and it would run when the if condition is trigger.

```
[ ]: if not condition: raise AssertionError()
```

2.18 Exception

Exceptions are errors that were detected during execution. Exception can arise even if a statement and expression is syntactically correct, the cause is due to python not being able to cope with the code. It comes in different types and the type is printed as part of the error message.

Examples:

```
[ ]: ValueError # Raised when an operation or function receives an argument that has␣
     ↪the right type but an inappropriate value, and the situation is not␣
     ↪described by a more precise exception such as IndexError.
     AssertionError #Raised when an assert statement fails

     KeyError # Raised when a mapping (dictionary) key is not found in the set of␣
     ↪existing keys.

     ZeroDivisionError # Raised when the second argument of a division or modulo␣
     ↪operation is zero. The associated value is a string indicating the type of␣
     ↪the operands and the operation.
```

Exception error message comes in form of a stack traceback and it shows the context where the exception occurred.

Example:

```
[ ]: #Assertion Stack traceback message
     Traceback (most recent call last):
       File "/home/bafc2f900d9791144fbf59f477cd4059.py", line 4, in
         assert y!=0, "Invalid Operation" # denominator can't be 0

     AssertionError: Invalid Operation
```

```
# We can tell that this is an AssertionError from the name of this exception␣
↪and the assertion error occurred in line 4 of the code.
```

Exceptions are not unconditionally fatal as it is possible to write programs that handle selected exceptions. To handle exceptions, Try Except is required. Look at the following example, it assigns two variables num1 and num2 with a value of 1 and 0 respectively. Python will try to execute the try block but num1 is divided by 0 which raises a ZeroDivisionError. The except block will catch the exception ZeroDivisionError and execute the except block.

[53]:
```
num1 = 1
num2 = 0
try:
    print(num1/num2)
except ZeroDivisionError as e: #except block executes the following action when␣
↪try returns an error.
    print("You cannot divide by 0")
```

```
You cannot divide by 0
```

2.19 Lambda Function

A `lambda` function is a small anonymous function. A `lambda` function can take any number of arguments, but can only have one expression.

Examples:

[54]:
```
x = lambda a : a + 10 # 5 is assigned to a, this would return a value of 15
print(x(5))
```

```
15
```

[55]:
```
x = lambda a, b, c : a + b + c # lamda can take any number of arguments.
print(x(5, 6, 2))
```

```
13
```

The power of `lambda` is better shown when you use them as an anonymous function inside another function. Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number.

Examples:

[56]:
```
def myfunc(n): # we define a function with parameter (n)
  return lambda a : a * n # this returns a value of 3*11=33

mytripler = myfunc(3)

print(mytripler(11))
```

33

2.20 Classes, Object

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods. A `Class` is like an object constructor, or a blueprint for creating objects. Classes allow us to bundle data and functionalities together.

To create a `class` we have to use the keyword `class`.

Example:

[57]:
```
class MyClass:
  x = 5
```

Creating a new class would also create a new type of object. This would create a new instance of that type to be made. Now we can use the class named MyClass to create objects.

Examples:

[58]:
```
p1 = MyClass()
print(p1.x)
```

```
5
```

All classes have a function called `init()`, which is always executed when the class is being initiated. Use the `init()` function to assign values to object properties, or other operations that are necessary to do when the object is being created.

Create a `class` named Person, use the `init()` function to assign values for name and age.

Example:

[59]:
```
class Person:
  def __init__(self, name, age): #create a function with parameters self, name,␣
 ↪age
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name) # this would print the
print(p1.age)
```

```
John
36
```

If one more argument is given when creating an object, there would be a `TypeError`.

Bad Example:

[60]:
```
class Person:
  def __init__(self, name, age): #create a function with parameters self, name,␣
↪age
    self.name = name
    self.age = age

p1 = Person("John", 36, 46)

print(p1.name) # this would print the
print(p1.age)
```

```
       ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-60-25e376e34c50> in <module>
         4     self.age = age
         5
   ----> 6 p1 = Person("John", 36, 46)
         7
         8 print(p1.name) # this would print the


       TypeError: __init__() takes 3 positional arguments but 4 were given
```

If one less argument is given when creating an object, there would be a `TypeError`.

Bad Example:

[61]:
```
class Person:
  def __init__(self, name, age, gender): #create a function with parameters␣
↪self, name, age
    self.name = name
        self.age = age

p1 = Person("John", 36)

print(p1.name) # this would print the
print(p1.age)
```

```
  File "<ipython-input-61-e8fbb26d380d>", line 4
    self.age = age
    ^
```

```
TabError: inconsistent use of tabs and spaces in indentation
```

Objects can also contain methods. Methods in objects are functions that belong to the object.

Example:

[62]:
```python
%reset -f

class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36) # creates a person object with the two argument.
p1.myfunc()#call object methods myfunc(self)
```

```
Hello my name is John
```

A TypeError would occur if there is a missing argument.

Bad Example:

[63]:
```python
%reset -f

class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self, name):
    print("Hello input name is " + self.name)

p1 = Person("John", 36) # creates a person object with the two argument.
p1.myfunc()#call object methods myfunc(self) but there is a missing parameter
```

```
      ␣
↪---------------------------------------------------------------------------

      TypeError                                 Traceback (most recent call␣
↪last)

      <ipython-input-63-f7e5c90d73dd> in <module>
       10
```

```
        11 p1 = Person("John", 36) # creates a person object with the two␣
↪argument.
   ---> 12 p1.myfunc()#call object methods myfunc(self) but there is a missing␣
↪parameter
        13


TypeError: myfunc() missing 1 required positional argument: 'name'
```

2.21 Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class. The class inheritance mechanism allows multiple base classes. A derived class can override any methods of its base class or classes. A method can call the method of a base class with the same name.

There are 2 main classes, Parent `class` and Child `class`. Parent `class` is the class being inherited from, also called base class. Child `class` is the class that inherits from another class, also called derived class. We can create a parent `class`.

Example:

[64]:
```python
%reset -f
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)#Use the Person class to create an␣
↪object, and then execute the printname method:

x = Person("JunJie", "Edwin")
x.printname()
```

```
JunJie Edwin
```

We then create child `class`. This child `class` will inherit the properties and methods from the Person `class`.

Example:

[65]:
```python
class Student(Person):
    pass # we use pass because we don't want to add any other properties or␣
↪methods to class.
```

We can then create the child `class` (student) to create an object.

[66]:
```
x = Student("Casper", "Calvin")
x.printname() #print out the properties in the class.
```

```
Casper Calvin
```

2.21.1 Super() Function

Python has a function called `super()`. This would make the child class inherit all the methods and properties from its parent. By using `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

[67]:
```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)


x = Student("Michael", "Jackson")
x.printname()
```

```
Michael Jackson
```

3 Python Packages

3.1 Modules

Modules are files containing a set of functions you want to include in your application. Modules are useful because they can store a handy function that you may need to use often into it. You can then call the function from the module whenever you need it.

In python, there is a way to put definitions in a file and use them in a script. These files are known as modules. Definitions and functions in a module can be imported into other modules.

To create a module, you would need to save a code into a file with the suffix `.py` appended.

A docstring is a string literal that usually appears as the first statement in a module. It is used to explain what the module does. The docstring becomes a `__doc__` special attribute of that object.

There are several built-in modules in python which can be imported and called at any time.

Example:

```
import platform

x = platform.system()
print(x)
```

You can also use the `dir()` function to list all the defined names belonging to the platform module.

Example:

```
import platform

x = dir(platform)
```

```
print(x)
```

3.2 Import

Python modules can get access to code from another module by importing the file/function using import. When import is used, it searches for the module initially in the local scope by calling **import**() function. The values returned by the function are then reflected in the output of the initial code.

The basic import statement is executed in two steps. Firstly, it finds a module, loads and initializes it. Secondly, defines names in the local namespace for the scope where the import statement occurs.

Import statements should be carried out in separate lines

Examples:

```
Import os

Import sys
```

When the requested modules is retrieved successfully, there would be 3 main ways it would be made available in the local namespace.

Firstly, using `import` and defining the module. If the module imported is a top level module, its name would be bound to the local namespace as a reference to the imported module.

Example:

```
Import stockdata
```

Secondly, you can create an alias when you import a module by using `as` keyword.

Example:

```
Import stockdata as SD
```

Thirdly, if the module imported is not a top level module, the name of the top level package that contains the module is bound in the local namespace. The imported module must be accessed using its full qualified name.

Example:

import pandas.dataframe

We can also use `from` together with the `import` statement.

Imports the module, and creates references to all public objects defined by that module in the current namespace or whatever name you mentioned. After you've run this statement, you can simply use a plain `()` name to refer to things defined in the `module(x)`. But `X` itself is not defined, so `X.name` doesn't work.

3.3 Packages

There are two types of packages in Python

1. Regular packages

Regular packages are traditional packages as they existed in Python 3.2 and earlier. A regular package is typically implemented as a directory containing an `__init__.py` file. When a regular package is imported, this `__init__.py` file is implicitly executed, and the objects it defines are bound to names in the package's namespace.

2. Namespace packages

   With namespace packages, there is no `parent/__init__.py` file. There may be multiple parent directories found during import search, where each one is provided by a different portion.

A namespace package is a composite of various portions, where each portion contributes a subpackage to the parent package. Portions may reside in different locations on the file system.

3.4 Searching for package

To begin the search, Python needs the fully qualified name of the module (or package, but for the purposes of this discussion, the difference is immaterial) being imported. This name may come from various arguments to the import statement, or from the parameters to the `importlib.import_module()` or `__import__()` functions. This name will be used in various phases of the import search, and it may be the dotted path to a submodule, e.g. `foo.bar.baz`. In this case, Python first tries to import `foo`, then `foo.bar`, and finally `foo.bar.baz`. If any of the intermediate imports fail, a `ModuleNotFoundError` is raised.

3.5 Pandas and dataframe

In computer programming, Pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

Dataframe is a 2-dimensional labelled data structure with columns of potentially different types. There are 3 components in a dataframe: rows, columns and data. Also, dataframe is generally the most commonly used pandas object.

Let us start with an example:

```
Import pandas as pd
df = pd.DataFrame ({"a" : [4 ,5, 6], "b" : [7, 8, 9], "c" : [10, 11, 12]}, index = [1, 2, 3])
```

which would give us:

```
   a  b   c
1  4  7  10
2  5  8  11
3  6  9  12
```

To start, we have to import Pandas using `import pandas as pd` first, which imports Pandas and assign it to a declared name by developer called `pd`.

Here we define the dataframe by putting in the data directly. However, a more common way to install dataframe is to read from other files, such as .csv file. If we want to use a .csv file, type

```
pd.read_csv(filepath)
```

For a complete list for importing different file types, check https://pandas.pydata.org/pandas-docs/stable/reference/io.html.

After importing an entire dataframe, pandas can extract particular sets of data such as:

```
df.head(2)
```

which gives us:

```
   a  b   c
1  4  7  10
2  5  8  11
```

This will return only first 2 rows of data and there are similar functions like `df.tail(n)` which will return only last 2 rows of data.

other functions like `df.loc` allows you to extract data based on specific search terms. For example:

```
df.loc[1]
```

gives us:

```
a      4
b      7
c     10
Name: 1, dtype: int64
```

`df.loc` will allow you to get rows/columns with particular labels from the index. As our example shows, `df.loc[1]` returns the row named '1' as well as its name and data type.

Using `[[]]` will return another dataframe that has all the data for both '1' as well as '2' such as:

```
df.loc[[1,2]]
```

which gives us:

```
    a   b   c
1   4   7   10
2   5   8   11
```

On the other hand, `df.iloc[]` searches for data based on its position. The 'i' in `iloc` refers to integer. For example:

```
df.iloc[1]
```

gives us:

```
a      5
b      8
c     11
Name: 2, dtype: int64
```

`df.iloc` may look similar to `df.loc` but their functions are different. When we type `df.loc[1]` we are finding the row named '1', while if we type `df.iloc[1]` it will return the row in position 1. Remember, in python indexing starts from 0, hence the row in position 1 is actually the second row in the dataframe (`df.iloc[0]` will return the first row).

You can also manipulate and append the dataframe using functions such as:

```
df['d'] = df['a']+df['b']
```

which gives us:

```
   a  b   c   d
1  4  7  10  11
2  5  8  11  13
3  6  9  12  15
```

This creates a new column 'd' in the dataframe that is the sum of column 'a' and 'b'.

Other than that, pandas can also generate useful summaries of the data. For example:

```
df.describe()
```

gives us:

```
    a   b   c
count   3.0 3.0 3.0
mean    5.0 8.0 11.0
std 1.0 1.0 1.0
min 4.0 7.0 10.0
25% 4.5 7.5 10.5
50% 5.0 8.0 11.0
75% 5.5 8.5 11.5
max 6.0 9.0 12.0
```

This will show the summary statistics for the numeral columns. For more specific statistics, you can use functions like `df.mean()`, which describes the means of all columns and `df.corr()`, which returns the correlation between columns in a dataframe.

```
df.to_csv(filename)
```

This will allow you to export the data you have into a .csv file and there are similar functions like df.to_excel (convert data into excel file)

3.6 Numpy

NumPy(Numerical Python) is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects, and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

The ndarray object is the core of the NumPy package. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size at creation which is different from Python lists. To change the size of an ndarray, a new array needs to be created and the original has to be deleted.
- The elements in a NumPy array are all required to be of the same data type. The exception: one can have arrays of objects.

- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

In order to use NumPy, we have to import numpy using `import numpy as np` which imports numpy and assign it to a declared name by developer called `np`. Developer have the freedom to name the declared name, but it is highly recommended to use `np`) To initialize NumPy, we can do it like the following example:

```
Import numpy as np
Index = np.array([0, 1, 2, 3, 4]) #converting array [0, 1, 2, 3, 4] to a numpy array
Index2 = np.array(1) #converting ndarray with the value 1
Index3 = np.array([[0, 1, 2, 3, 4],[0, 1, 2]]) #converting 2d array [[0, 1, 2, 3, 4],[0, 1, 2]]
```

As previously mentioned, ndarray requires all the elements to be the same type, therefore if a list of mixture of integers and float will be converted to a list of float.

```
Index4 = np.array([0,1,2,3.0] #all elements will be converted to float
print(index4)
```

3.7 Numpy round

The numpy.round is an inbuilt function in NumPy that is used to round off every single element in the numpy array(ndarray). The `numpy.round` will take in 2 arguments: the first argument will be the numpy array and the second argument will be the decimal place to round up to. The following example will demonstrate the usage and syntax of numpy.rounds

```
Import numpy as np
Index = np.array([0.5, 1.35, 2.68, 3.10, 4.9000])#converting array [0.5, 1.35, 2.68, 3.10, 4.9
np.round(data, 2) #numpy will round up every element in the numpy array to two decimal places
```

3.8 Numpy NaN

The numpy.nan is a floating point representation(float) of Not a Number (NaN). It is similar to Python's `none`. NaN can be assigned to an index in a numpy array and dataframes. The following example will demonstrate it:

```
Import numpy as np
index = np.array(1) #creating a ndarray with the value 1
Index = np.nan #assign np.nan to index
print(index)

index2 = np.array([1.0, 2.0 ,3.0]) #creating ndarray with the array[1.0, 2.0, 3.0]. At least o
Index2[0] = np.nan #assign index2[0] as np.nan
print(index)
```

3.9 Datetime

`datetime` is a module that provides classes for manipulating dates and times. In Python, date alone is not a data type of its own, but if we use `datetime` to work with dates as date objects.

To use `datetime`, we have to import it first with the following statement. This will import the classes in date time into the Python file.

```
from datetime import datetime
```

OR

```
Import datetime
```

3.10 Strptime

strptime is a function in datetime that creates a datetime object from the given string. Do note that not every string can create a `datetime` object, it needs to be in a certain format. strptime() class function contains two parameters: a string (to be converted to `datetime`) and format code. It will raise a `ValueError` exception if both arguments do not match. For Example:

```
From datetime import datetime
dateString = "12/11/2018 09:15:32"
dateObj = datetime.strptime(dateString, "%d/%m/%Y %H:%M:%S")
print(dateObj)
```

gives us:

```
2018-11-12 09:15:32
```

For more information on the format codes that can be used, check here:https://docs.python.org/3/library/datetime.html

3.11 Matplotlib

Matplotlib is an open-sourced, low level graph plotting library in python and it allows us to visualise data. Similarly, we need to import matplotlib at start:

```
import matplotlib
```

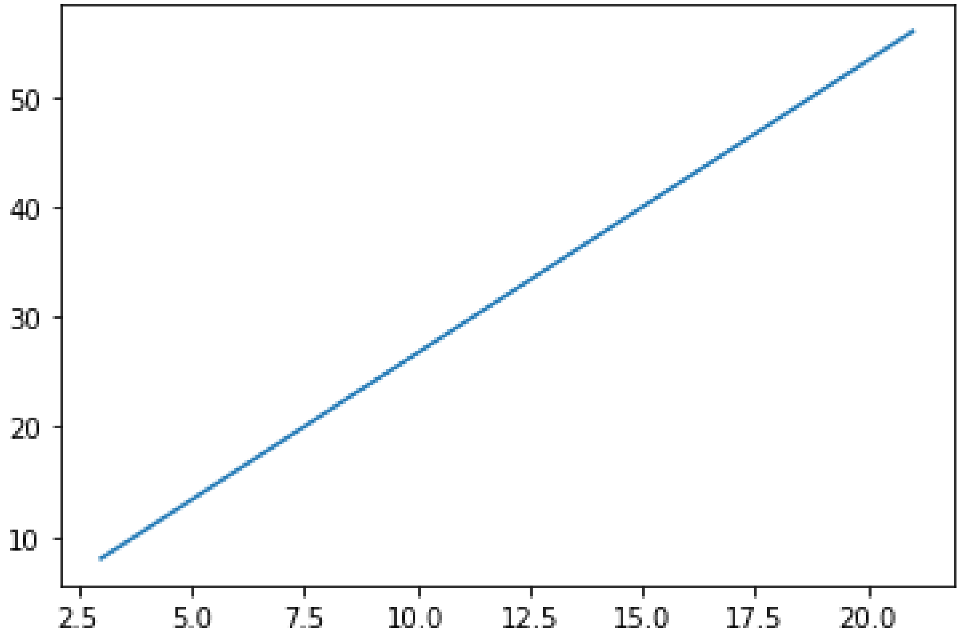
3.12 pyplot

pyplot submodule is mostly used in Matplotlib and we also need to import it first.

```
import matplotlib.pyplot as plt #R
import numpy as np

xpoints = np.array([3, 21])
ypoints = np.array([8, 56])

plt.plot(xpoints, ypoints)
plt.show()
```



will plot the following graph:

The plot(x,y) function allows us to draws a line from point to point. Parameter 1 is an array containing the points on the x-axis, which is horizontal, and parameter 2 is an array containing the points on the y-axis, which is vertical. plot(y) is also possible and here x is an index array which start from 0 to N-1.

3.13 Date

Beyond python, Matplotlib provides sophisticated date plotting capabilities. Start with importing matplotlib.dates as mdates.

```
import matplotlib.dates as mdates
```

As the datetime objects are different in python and matplotlib, date2num function allows us to convert datetime objects to Matplotlib dates and num2date provides the opposite, which coverts Matplotlib dates to datetime objects.

Also, syntax like DateFormatter(fmt,tz) use strftime format strings. Here, fmt means a strftime format string and is always required; tz means the timezone and can be set to none to ignore the timezone information.

4 `stock_data.py`

4.1 `import` statements

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

```
import numpy as np
```

```
import pandas as pd
import matplotlib.pyplot as plt
```

Learning points: Package Aliasing

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

4.2 `class` statement

We create a class name `StockData` which will contain the attributes and functions.

```
class StockData():
```

Learning point: Classes

Classes are often created because it allows us to bundle data and functionalities together.

4.3 `__ init __(self)` (constructor statement)

We create a constructor using `__init__` which requires one string parameter: `filepath`. The attribute `filepath` stores the parameter `filepath`. The attribute `data` stores the pandas dataframe which is extracted from a CSV file found in the `filepath`. When constructing a `StockData` object, it will call and run the `check_data()` function.

```
    self.filepath = filepath
    self.data = pd.read_csv(filepath).set_index('Date')
    self.check_data()
```

Learning point: Default Constructor

If you do not create a Constructor, Python will automatically create a default constructor that does not do anything.

4.4 `check_data(self, overwrite=True)`

We first start by checking for any missing data and then filling in any missing values by interpolation in the csv data. We use the `interpolate()` function to fill in the estimated values. The interpolate function uses a linear interpolation which takes the average of the value before and after the data point to come out with an estimation. We started with this step so that our dataset would be cleaned and have no missing values.

We define a function name `check_data()`. This functions checks and handles missing data by filling in missing values by interpolation. The parameter `(overwrite = True)` takes a boolean value and overwrites the original source stock data .csv file.

```
    self.data = self.data.interpolate()
```

Learning point: Indentation

When creating a function, we would need to make sure there is proper indentation after the colon. All the code that is in the function would need to have the same indentation.

The next part is to overwrite the original stock data.csv file. We would use a pandas inbuilt function`to_csv()` with the parameters (self.filepath) as the filepath and (index= overwrite) to overwrite the csv file.

```
self.data.to_csv(self.filepath, index=overwrite)
```

We then use `return` to send the StockData to any code that calls this function.

```
return self
```

> *Learning point:* `return` *statement*
>
> `return` statement is often used at the end of the function to returns the results (values) of the expression to the caller. Statements after the return statement are not executed. If the return statement is without any expression, the value returned would be `none`.*

4.5 `get_data(self, start_date, end_date)`

The get_data function to return a subset of the stock data from start_date to end_date inclusive. The parameter `start_date` and `end_date` has a type `str` that is the start date and end_data of stock data range, must be of format YYYY-MM-DD.

The variable self.selected_data would store a dataframe indexed from the specified start to end date inclusive.

```
self.selected_data = self.data[str(start_date):str(end_date)]
```

We then use `return` to send the `selected_data` that consist of start and end dates to any code that calls this function.

```
return self.selected_data
```

4.6 `get_period(self)`

The get_period function is used to obtain the earliest and latest date in the `data` dataframe. Since `data` have index based on the date, we can obtain a list of date with `list(self.data.index)`. With the list, we can obtain the first and last index in the list and return them in a tuple.

```
 index = list(self.data.index)
 (first, last) = (index[0], index[-1])
 return (first, last)
```

> *Learning point: Returning Multiple Variables*
>
> *If you want to return more than one variable, you can return them in heterogeneous containers like tuple or list.*

4.7 `calculate_SMA(self, n)`

In the calculate_SMA function, we take in 1 parameter: n which is the number of days used to calculate the simple moving average (SMA). With n, we will create a column label named `SMA + n`.

```
col_head = 'SMA' + str(n)
```

For example, if n is 15, the column label will be named 'SMA15'

Due to the dataframe of the self.data having an index using the date, we use reset_index() to undo the index and reinclude date into one of the columns.

```
df = self.data.reset_index()
```

Learning point: Built-in Functions

To speed the working progress, we should use in-built functions provided by packages if it fulfil the requirements.

Then we check if the column name `col_head` is found in `df` by using the following code:

```
if col_head not in df.columns:
```

Learning point: `not` *statement*

`not` *is a logical operator commonly used with conditional statements such as* `if else` *or* `while`.

If it is found in `df`, we will `return self` and leave the dataframe untouched as the SMA of `n` number of days has already been calculated. Otherwise, we will begin the calculation.

We begin by retrieving the list of date found in the self.data(portion of the full data) and creating `returnList` which will store the calculated SMA later on by using the following code:

```
    dateList = self.data.index.values.tolist()
    returnList = []
```

With this list of data, we will do a for loop with each of the date in the list and find the index of each specific date in the full dataset. We will then use these dateIndex to see if there are enough datasets to calculate the SMA. For example, we need 15 data set prior to the current day in order to calculate the SMA of 15 days. If there is not enough data prior to the current date, we will append NaN into the `returnList` to show that we do not have SMA for that current date.

```
    for date in dateList:
        dateIndex = df[df["Date"]==date].index.values[0]
        if dateIndex < n: # if date index is less than n: append None
            returnList.append(np.nan)
        else:
            sum = 0
            for i in range(n):
                sum += df.iloc[dateIndex-i]["Close"]
            returnList.append(sum/n)
```

If there is enough data, we will do a for loop with `n` number of iterations to calculate the sum of adjusted close values for n number of days which is the SMA value. At the end of the loop, we will append the SMA into `returnList`.

After calculating all the SMA for every date in self.data, we insert the `returnList` containing all the SMA value with a column name stored in `col_head`. At the end of the function, we save the dataframe with SMA into a CSV file.

```
    self.data[col_head] = returnList
    self.data.to_csv(self.filepath, index=True)
```

4.8 calculate_crossover(self, SMAa, SMAb)

We first start by creating and defining the shell of the calculate_crossover function:

```
def calculate_crossover(self, SMAa,SMAb):
...
    return self
```

This function takes in the two SMA values previously calculated in the calculate_sma function as inputs to calculate the crossover locations.

Next we will start to write the code inside the function. We first define the columns we plan to add to the .csv file and extract the all data in the .csv file:

```
col_head3 = 'Buy'
col_head4 = 'Sell'
df = self.data
```

We convert the data into a list which we will use as a reference to ensure our subsequent calculations have the correct number of elements

```
SMAlist = self.data.index.values.tolist()
```

We then use an if, elif, and else statement to assign the lower SMA to SMA1 from the and the higher SMA to SMA2. This is useful later in the calculations to ensure that buy and sell signals are correctly identified.

```
if SMAa < SMAb:
    SMA1 = df[SMAa].tolist()
    SMA2 = df[SMAb].tolist()
elif SMAa > SMAb:
    SMA1 = df[SMAb].tolist()
    SMA2 = df[SMAa].tolist()
else: # SMAa == SMAb
    raise ValueError(f"Given {SMAa} & {SMAb} are the same. Must be different SMA.")
```

> *Learning point:* `if, elif, and else` *statements*
>
> `elif` *is used here because there are multiple distinct different possibilities with how SMAa and SMAb are related. It is common to list the expected possibilities first in the* `if` *and* `elif` *statements, and* `else` *would normally be reserved for unexpected outcomes or errors*

`df.[SMAa].tolist()` extracts the column `SMAa` from the dataframe `df` and converts it to a list. Likewise for `df.[SMAb].tolist()`. If the two SMA values are equal, the code will raise a value error and the error message.

We create empty lists for the relative position of the two SMAs (`stockPosition`), the combined list of crossover signals (`stockSignal`), and finally separate lists for the buy and sell signals (`buySignal`, `sellSignal`). These lists will be referenced and used in the next few lines of code.

```
stockPosition = []
stockSignal = []
buySignal = []
```

```
sellSignal = []
```

To create a list of relative SMA positions, we use a `for` loop:

```
for i in range(len(SMAlist)):
    if SMA1[i] > SMA2[i]: stockPosition.append(1)
    elif SMA1[i] < SMA2[i]: stockPosition.append(0)
    elif SMA1[i] == SMA2[i]: stockPosition.append(stockPosition[i-1])
    else: stockPosition.append(np.nan)
```

By setting the range of the for loop to be the length of `SMAlist`, we ensure that the loop iterates over every single element in the dataframe.

Any day that `SMA1` (the smaller one) is higher than `SMA2` will add a `1` to the stockPosition list. Days where `SMA2` is higher than SMA1 will add a `0` to the `stockPosition` list. The end result will be a list of 1s and 0s showing which SMA is higher on any given day.

In the unlikely case that the two SMA vaues are equal in a day, the number added will be a repeat of the previous day, as no crossover has occured yet.

On days where either SMA is missing data, such as in the first few days when there is not enough data to compute the SMA, we will add `np.nan` to the list as a filler.

After getting the full `stockPosition` list, we need to identify the days where crossover occurs. For this, another for loop is used:

```
for j in range(len(stockPosition)):
    if j == 0: stockSignal.append(np.nan)
    else: stockSignal.append(stockPosition[j] - stockPosition[j-1])
```

Again we set the range for the loop to be the length of `stockPosition` to ensure the code iterates over every element.

The `stockSignal` list 'lags' behind the stockPosition list by one day, hence we add a `np.nan` as the very first value in the list to align the `stockSignal` list with the `stockPosition` list and ensure that both lists have the same number of elements.

Following that we take the difference between the stockPosition that day and the `stockPosition` the previous day to identify the locations of crossovers. Crossovers show up in the list as `1` for a buy signal, and a `-1` for sell signals. `0` indicates that there has been no crossover that day.

> *Learning point: indexing*
>
> *Remember that in python, sequences start with 0, not 1! Hence,* `j == 0` *just refers to the first element in the range*
>
> *Learning point:* ***`np.nan`***
>
> *Remember that any arithmetic operation on* `NaN` *will result in* `NaN`*. This allows us to append the list with null values without generating a value error*

The next step would be to filter out the buy and sell signals, which will be processed separately by the application:

```
for k in range(len(stockSignal)):
    if stockSignal[k] == 1:
```

```
        value = self.data[SMAa].tolist()[k]
        buySignal.append(value)
    else: buySignal.append(np.nan)

for k in range(len(stockSignal)):
    if stockSignal[k] == -1:
        value = self.data[SMAa].tolist()[k]
        sellSignal.append(value)
    else: sellSignal.append(np.nan)
```

Using yet another set of for loops, we identify the crossover locations in the `stockSignal` list. At the crossover locations, we append the average SMA values of that particular day to the appropriate buy or sell list. This value will then be used as the y-axis value that the application uses to plot the crossover signals on the graph.

The else condition appends `np.nan` to the list on days that do not contain the respective crossover signals, and ensures that the signals are correctly aligned to the dates where the crossover occurred.

Finally, with the locations of buy and sell crossover signals, the function will append the buy and sell signals to the .csv file as new columns while also printing the results in the application:

```
self.data[col_head3] = buySignal
self.data[col_head4] = sellSignal

print(self.data)
self.data.to_csv(self.filepath, index=True)
```

> *Learning point: Testing*
>
> *The reason why the function prints the results is so we can independently test whether the function works even before the rest of the app is completed. Splitting work up in such a complex application is crucial so you can identify exactly which part of the app is causing errors!*

5 main_window.py

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

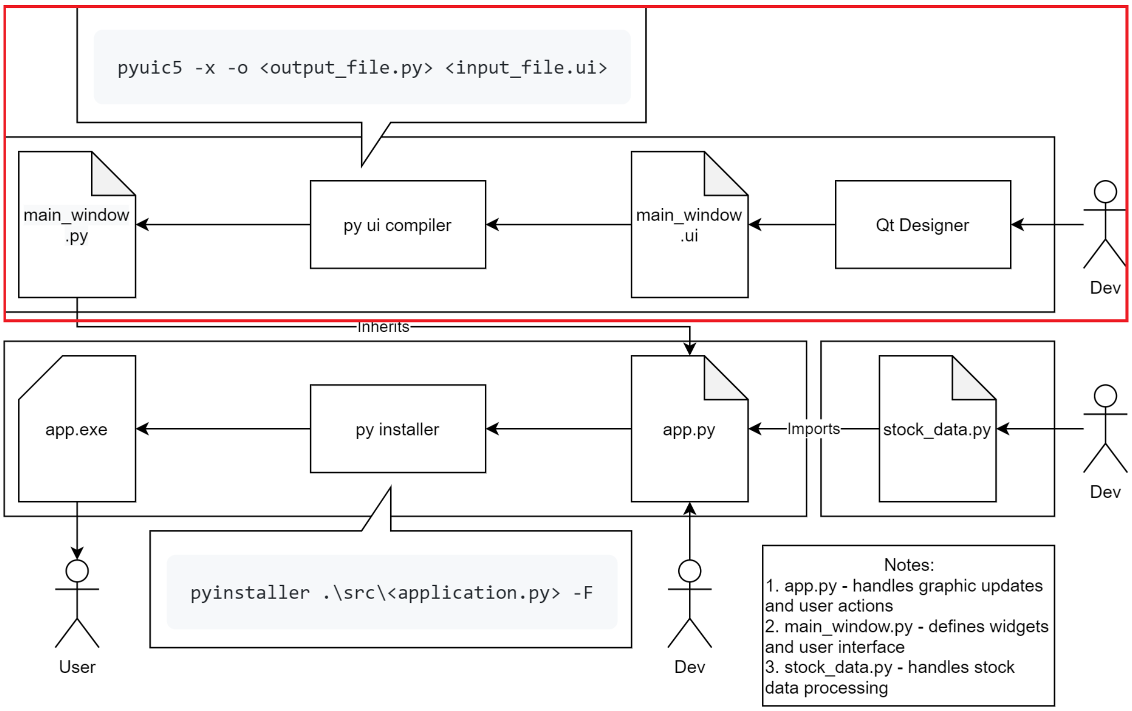
1. Defining each `Widget` objects' and their names within the GUI
2. Defining the location, size and other physical attributes of each `Widgets`

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.

5.1 Installing `Qt Designer`

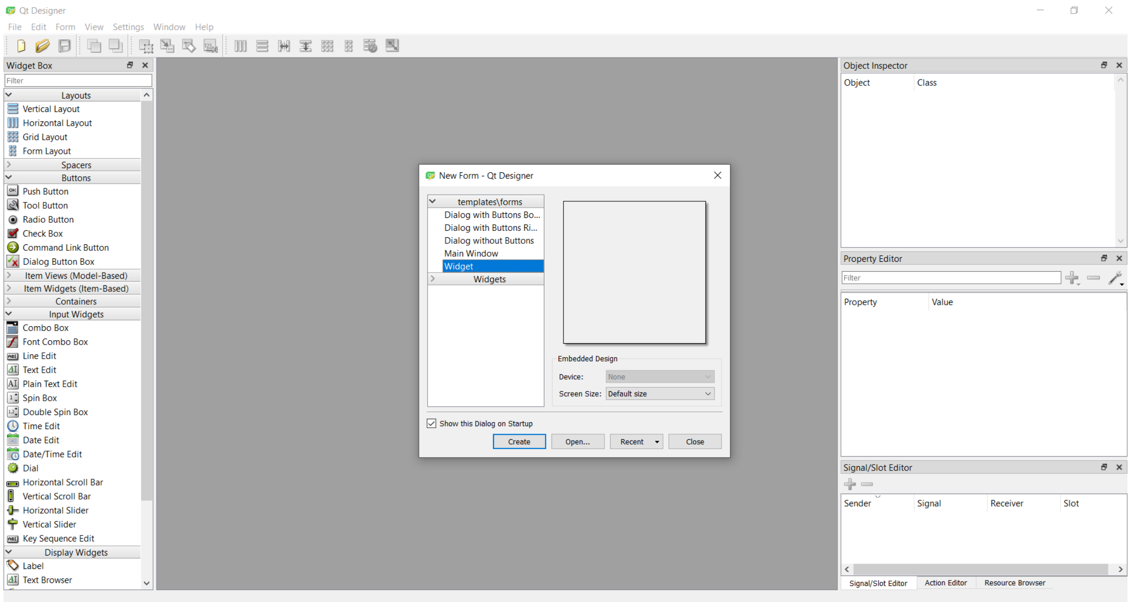
The installation process of `Qt Designer` is similar to any other application.

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

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

5.2.1 Defining the GUI

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



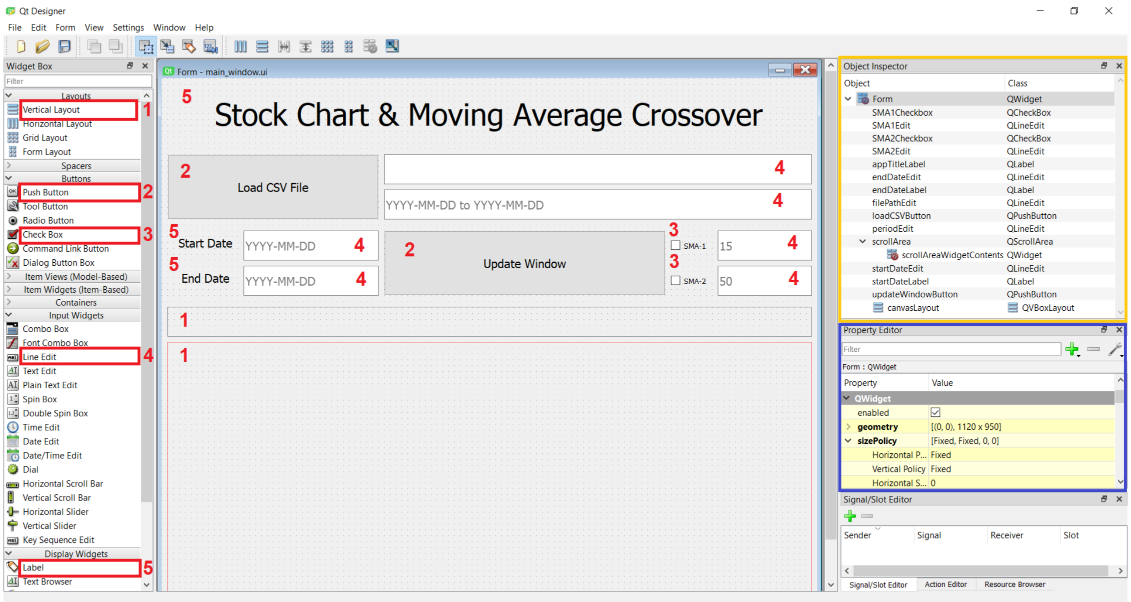
Choose `Widget` under the `template\forms` prompt and press the `Create` Button to begin designing `main_window.ui`.

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`*

5.2.2 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.

5.3 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.

5.4 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`.

6 `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`

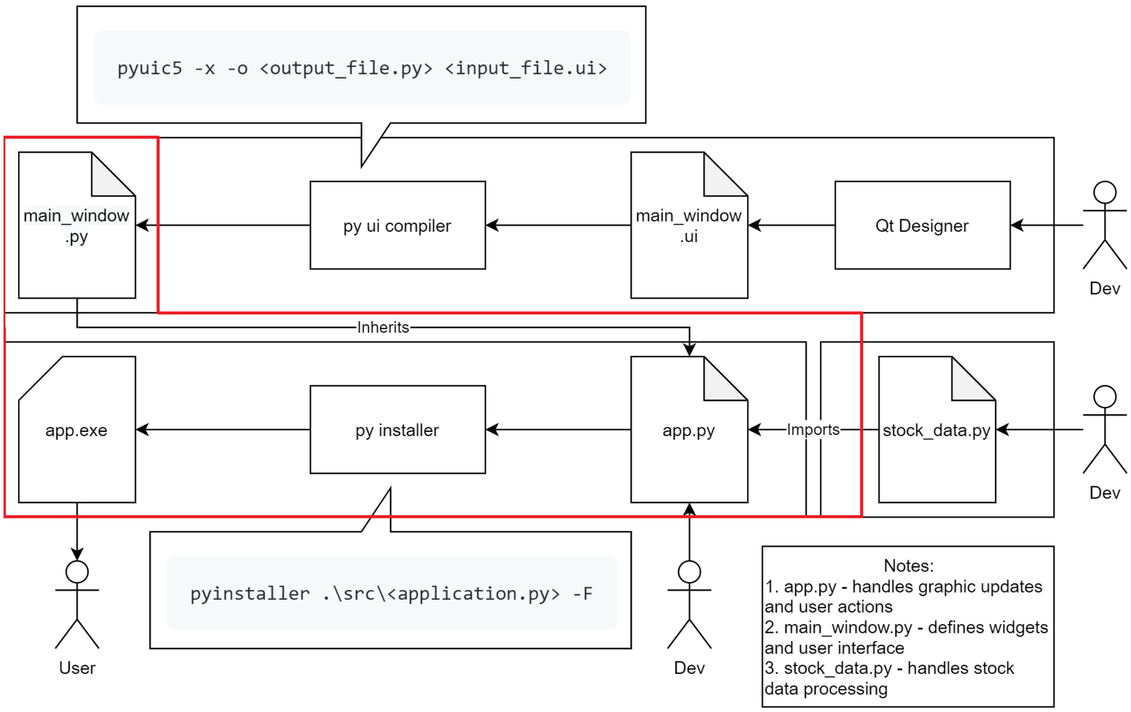
For example, if we want the `Update Window Button` to plot the stock prices in the GUI's canvas. We will have to create a `function` that plots the graph into the canvas and then connect the `Update Window Button` to this `function`.

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!

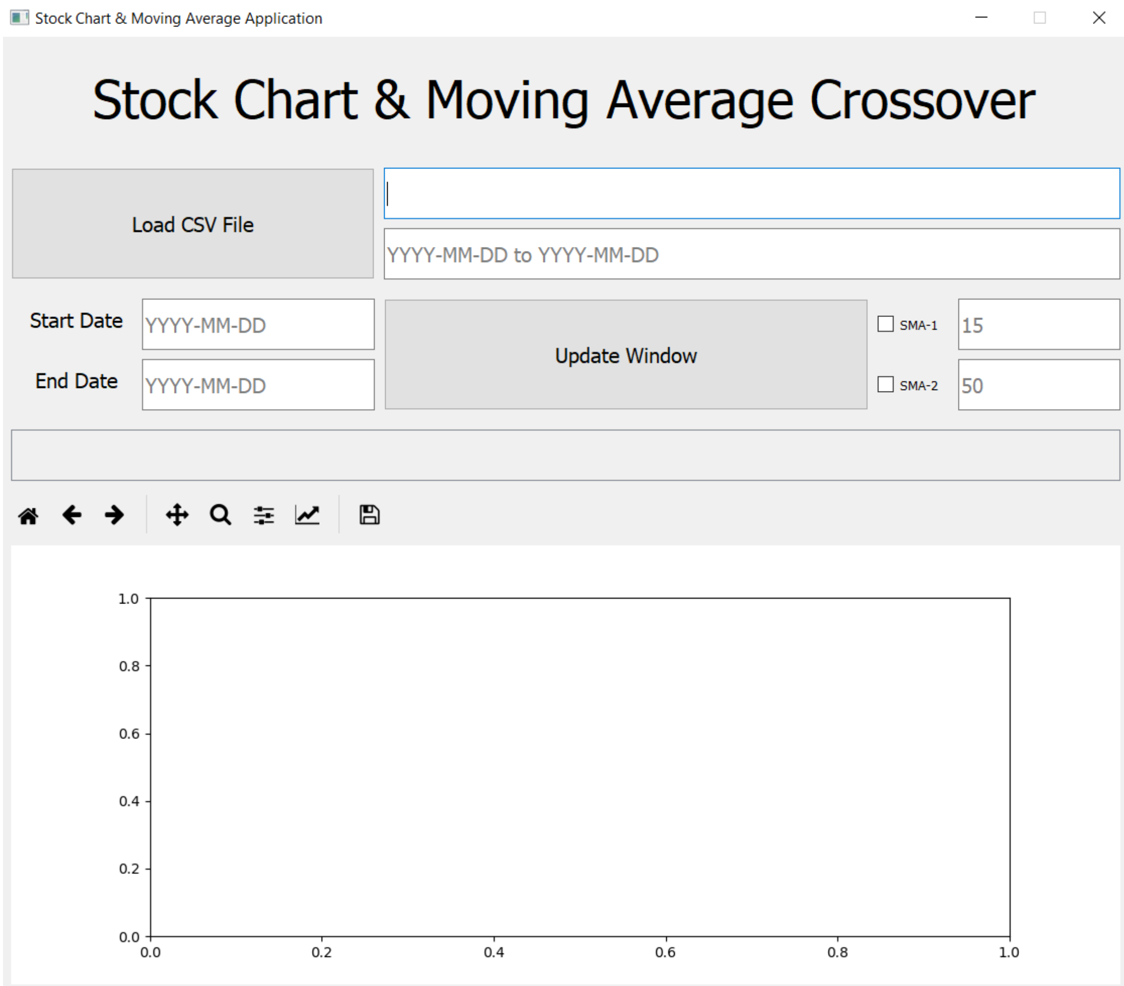
This is why, on the previous step, it is **recommended** to name the `Widgets` **accordingly!**

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.



6.1 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
```

```python
# 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`

6.2 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**.

6.2.1 `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()`.

```
    start_date, end_date = self.stock_data.get_period()
    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`.

```
    # 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)
```

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)`.

6.2.2 `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:

```
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())
```

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:

```
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.

```
    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:

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

```
    if self.SMA1Checkbox.isChecked():
        self.stock_data._calculate_SMA(int(self.SMA1Edit.text()))
        column_headers.append(f"SMA{self.SMA1Edit.text()}")
...
    if self.SMA2Checkbox.isChecked():
        self.stock_data._calculate_SMA(int(self.SMA2Edit.text()))
        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:

```
    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.*

6.2.3 `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)`.

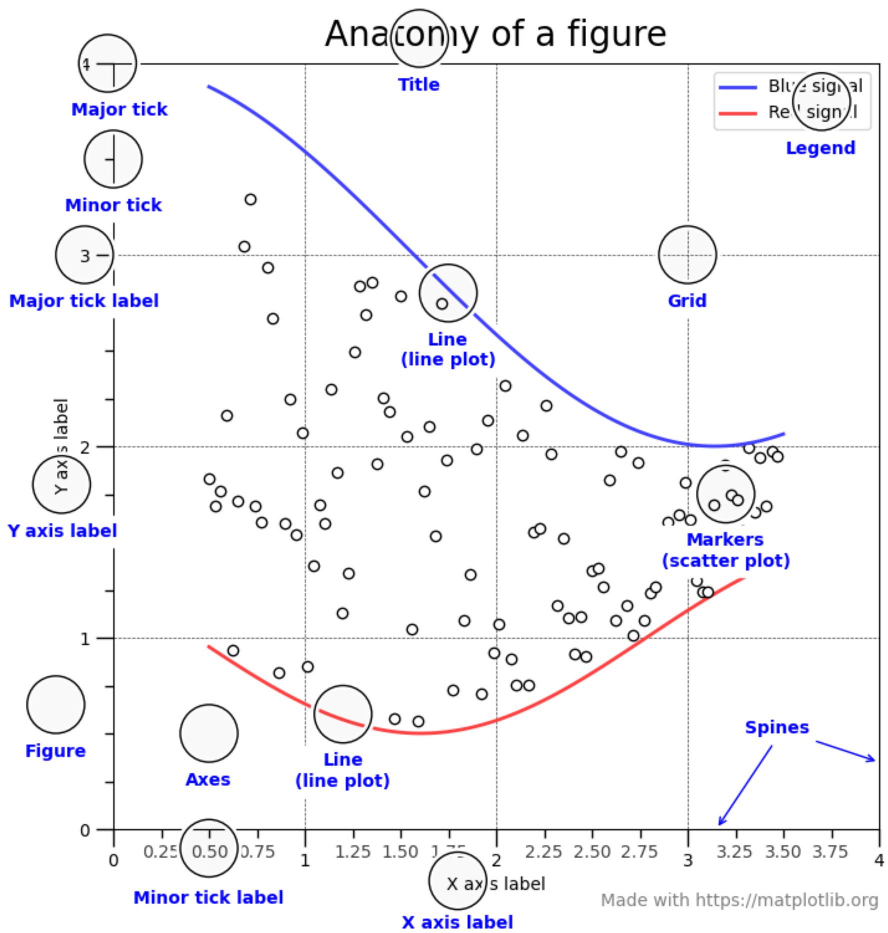
Once the plots are drawn, there may be some formatting that needs to be done on how either the `Axes` or the `Figure` looks like:

```
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!

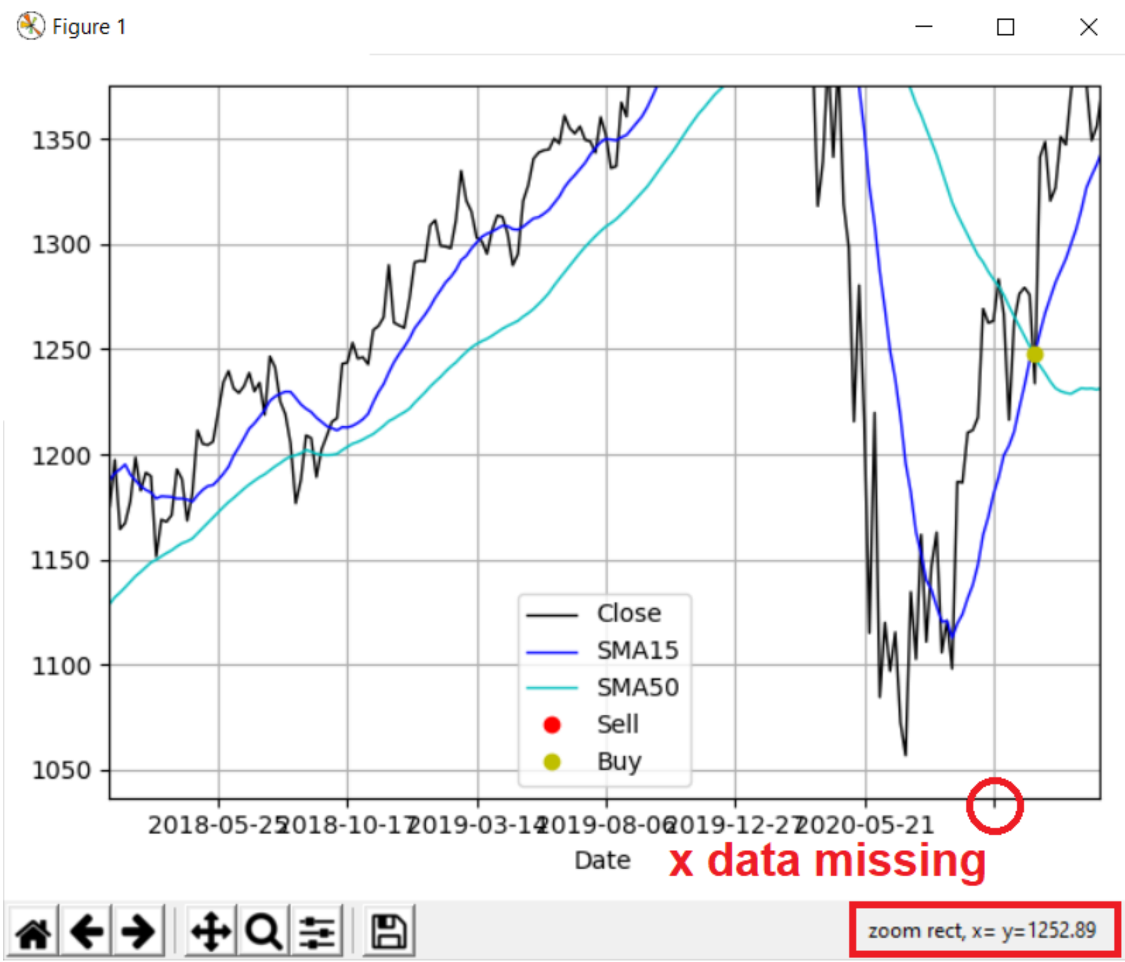
There are many components that are editable to make a plot looks just right! Thus, it is important to know what is in fact editable by understanding the parts of a `Figure`.

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.

6.2.4 `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.

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

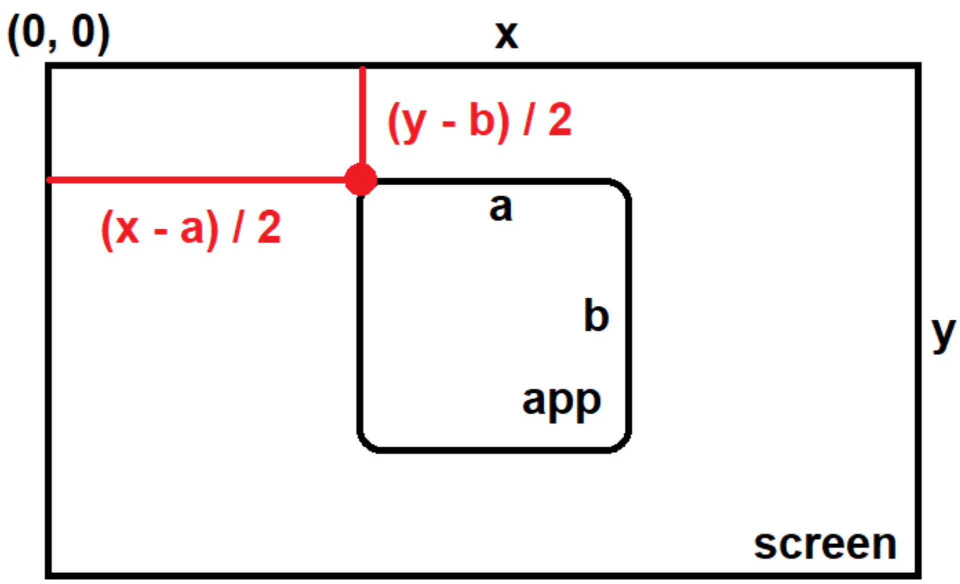
To simulate `print` statements, simply add new `Label Widget` with the `string` statement as its value. This is attached to a `Layout` that can be scrolled.

6.2.5 `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.

```
screen = qtw.QDesktopWidget().screenGeometry()
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.

```
x = (screen.width() - main_window.width()) / 2

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

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

6.3 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)`

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

> *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`*

6.4 (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!*

7 Appendix

7.1 Code Reference

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

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

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

7.1.1 app.py

____ init ____(self)

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

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

```
self.SMA1Checkbox.stateChanged.connect(self.update_canvas)
self.SMA2Checkbox.stateChanged.connect(self.update_canvas)

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

load_data(self)

[3]:
```
print_code(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")
```

update_canvas(self)

[4]:
```
print_code(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.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('m-')
    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}")
```

plot_graph(self, column_headers, formats)

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

```
"""
plots graphs specified under column_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]
                               ))

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])
            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()
```

report(self, string)

[6]:
```
print_code(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)
```

center(self)

[7]:
```
print_code(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

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

7.1.2 stock_data.py

__init__(self)

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

```
"""
initializes StockData object by parsing stock data .csv file into a dataframe
(assumes 'Date' column exists and uses it for index),
also checks and handles missing data

Parameters
filepath : str
    filepath to the stock data .csv file, can be relative or absolute

Raises
IOError :
    failed I/O operation, e.g: invalid filepath, fail to open .csv
"""
self.filepath = filepath
self.data = pd.read_csv(filepath).set_index('Date')
self.check_data()
```

check_data(self, overwrite=True)

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

```
"""
checks and handles missing data by filling in missing values by interpolation

Parameters
overwrite : bool (True)
     if True, overwrites original source stock data .csv file

Returns
self : StockData
"""
# function to fill in missing values
# by averaging previous data and after (interpolation)
self.data = self.data.interpolate()
self.data.to_csv(self.filepath, index=overwrite)
return self
```

get_data(self, start_date, end_date)

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

```
"""
returns a subset of the stock data from start_date to end_date inclusive

Parameters
start_date : str
     start date of stock data range, must be of format YYYY-MM-DD
end_date : str
     end date of stokc data range, must be of format YYYY-MM-DD

Returns:
selected_data : DataFrame
     stock data dataframe indexed from specified start to end date inclusive

Raises
KeyError :
     data for this date does not exist
AssertionError :
     selected range is empty
"""
self.selected_data = self.data[str(start_date):str(end_date)]
return self.selected_data
```

get_period(self)

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

```
"""
```

```
returns a string tuple of the first and last index
which make up the maximum period of StockData

Returns
period : (str, str)

Raises
TypeError :
     the return tuple is probably (nan, nan) because .csv is empty
"""
index = list(self.data.index)
(first, last) = (index[0], index[-1])
return (first, last)
```

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

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

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

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

Returns
self : StockData
"""
col_head = f'SMA{n}'
if col_head not in self.data.columns:
     sma = self.data[col].rolling(n).mean()
     self.data[f'SMA{n}'] = np.round(sma, 4)
     self.data.to_csv(self.filepath, index=True)
return self
```

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

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

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

```

```
Parameters
SMA1 : str
    the first column head title containing the SMA values
SMA2 : str
    the second column head title containing the SMA values
col : str ('Close')
    the column head title whose values will copied into 'Buy' and 'Sell'
    columns to indicate crossovers had happen on that index

Returns
self : StockData

Raises
Exception :
    SMA1 and SMA2 provided are the same, they must be different
"""
if SMA1 < SMA2: signal = self.data[SMA1] - self.data[SMA2]
elif SMA1 > SMA2: signal = self.data[SMA2] - self.data[SMA1]
else: raise Exception(f"{SMA1} & {SMA2} provided are the same. They must be different SMA.")

signal[signal > 0] = 1
signal[signal <= 0] = 0
diff = signal.diff()

self.data['Sell'] = np.nan
self.data['Buy'] = np.nan
self.data.loc[diff.index[diff < 0], 'Sell'] = self.data.loc[diff.index[diff < 0], col]
self.data.loc[diff.index[diff > 0], 'Buy'] = self.data.loc[diff.index[diff > 0], col]

self.data.to_csv(self.filepath, index=True)
return self
```

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

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

```
"""
plots columns of selected values as line plot and/or columns of values
as scatter plot as specified by style to an Axes object

Parameters
col_headers : [str, str, …]
    a list containing column header names whose data are to be plotted
style : [str, str, …]
```

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

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

calculate_SMA(self, n)

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

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

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

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

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

```
    # find the index of date in the full data
            dateIndex = df[df["Date"]==date].index.values[0]
            if dateIndex < n: # if date index is less than n: append None
                    returnList.append(np.nan)
            else:
                    sum = 0
                    for i in range(n):
                            sum += df.iloc[dateIndex-i]["Close"]
                    # append the SMA for each day to a list
                    returnList.append(sum/n)

    self.data[col_head] = returnList
    print(self.data)
    self.data.to_csv(self.filepath, index=True)

return self
```

calculate_crossover(self, SMAa, SMAb)

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

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

Parameters
SMA1 : str
    the first column head title containing the SMA values
SMA2 : str
    the second column head title containing the SMA values
col : str ('Close')
    the column head title whose values will copied into 'Buy' and 'Sell'
    columns to indicate crossovers had happen on that index

Returns
self : StockData

Raises
Exception :
    SMA1 and SMA2 provided are the same, they must be different
"""
col_head1 = 'Position'
col_head2 = 'Signal'
col_head3 = 'Buy'
col_head4 = 'Sell'
df = self.data
```

```

# to ensure the correct number of elements in the loop
SMAlist = self.data.index.values.tolist()
# extracts the SMA from the specific column in self.data
if SMAa < SMAb:
    SMA1 = df[SMAa].tolist()
    SMA2 = df[SMAb].tolist()
elif SMAa > SMAb:
    SMA1 = df[SMAb].tolist()
    SMA2 = df[SMAa].tolist()
else: # SMAa == SMAb
    raise ValueError(f"Given {SMAa} & {SMAb} are the same. Must be different SMA.")

stockPosition = []  # which SMA line is on top
stockSignal = []    # the buy/sell signal --> the 1s and -1s
buySignal = []              # filtered out location of buy signals
sellSignal = []             # filtered out location of sell signals

# goes through every element
for i in range(len(SMAlist)):
    if SMA1[i] > SMA2[i]: stockPosition.append(1)   # SMA1 above SMA2
    elif SMA1[i] < SMA2[i]: stockPosition.append(0) # SMA2 above SMA1
    # if the SMAs are equal, repeat the previous entry
    # because no crossover has occured yet
    elif SMA1[i] == SMA2[i]: stockPosition.append(stockPosition[i-1])
    else: stockPosition.append(np.nan) # if no data, leave blank

# find the places where crossover occurs
for j in range(len(stockPosition)):
    # 'shifts' the data one period to the right
    # to ensure crossovers are reflected on the correct date
    if j == 0: stockSignal.append(np.nan)
    # calculation for the crossover signals
    else: stockSignal.append(stockPosition[j] - stockPosition[j-1])


for k in range(len(stockSignal)): # finding location of buy signals
    if stockSignal[k] == 1:
            value = self.data[SMAa].tolist()[k]
            buySignal.append(value)
    else: buySignal.append(np.nan) # if no signal leave blank

for k in range(len(stockSignal)): # finding location of sell signals
    if stockSignal[k] == -1:
            value = self.data[SMAa].tolist()[k]
            sellSignal.append(value)
    else: sellSignal.append(np.nan) # if no signal leave blank
```

```
self.data[col_head3] = buySignal
self.data[col_head4] = sellSignal

print(self.data)
self.data.to_csv(self.filepath, index=True)
return self
```