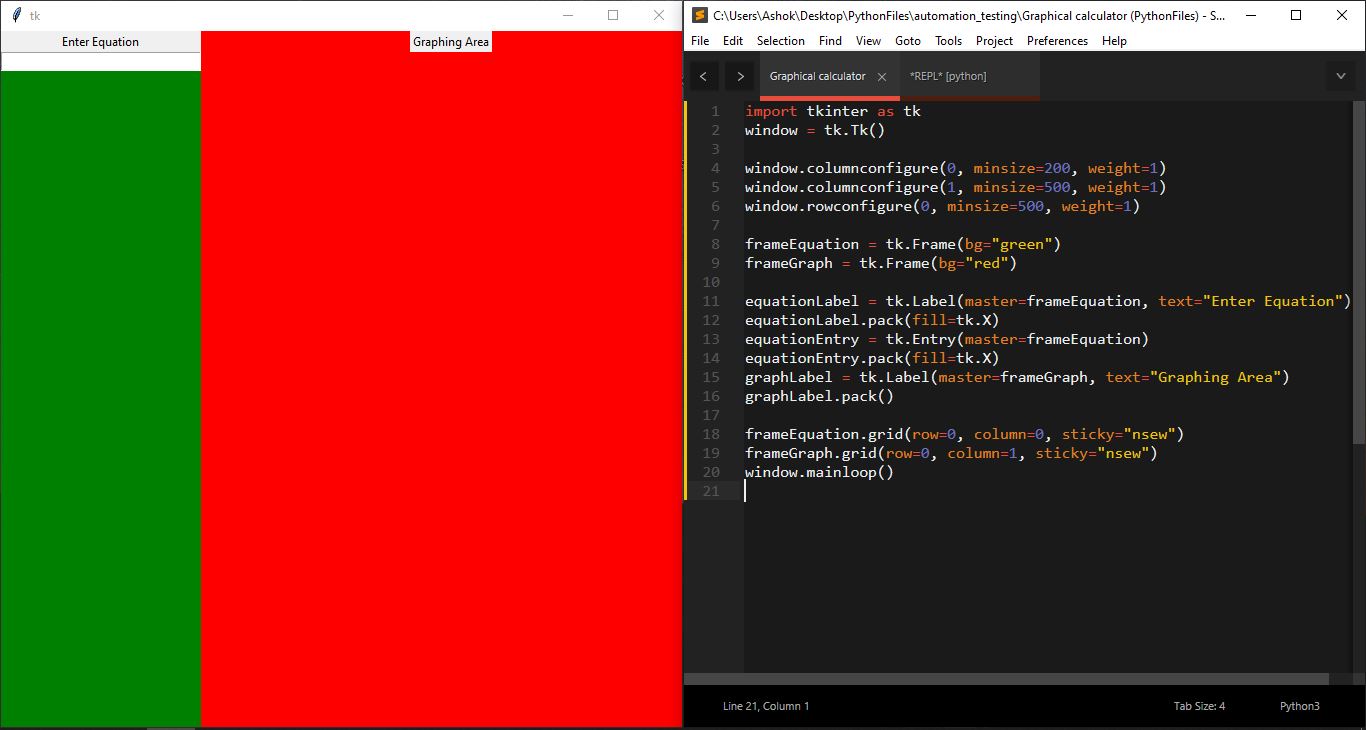
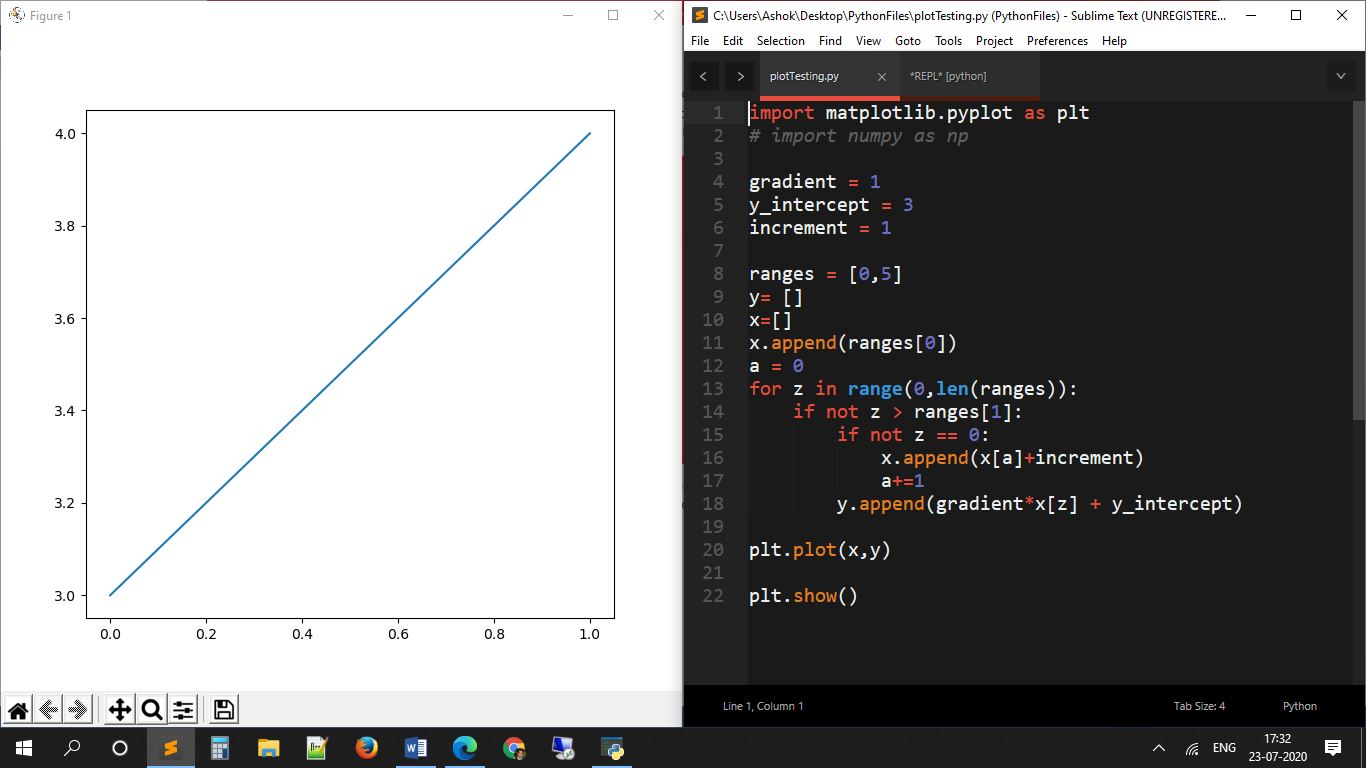
Goal –

To create a graphing calculator that graphs any equation, that finds its usages when studying mathematics, but can also be used in various situations to view the general trend of a situation in a graphical form. I will be using python to create this software.

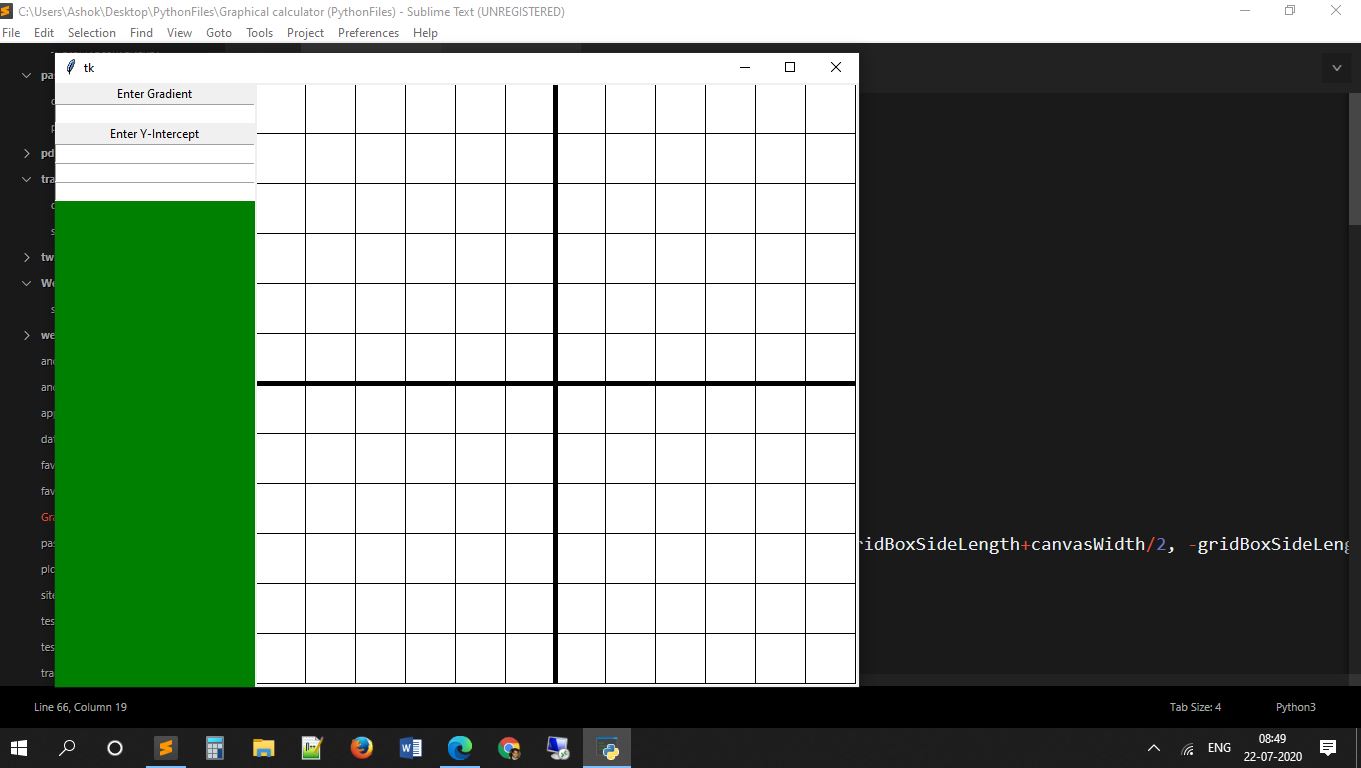
Documentation of the Creation of the Graphic Calculator–

To create general UI, I used a python module named tkinter. This module allows one to create basic UI with python code. It also provides a canvas functionality (a widget) where one can draw lines, squares and other shapes that can be useful to plot the graphs. After some playing around with the module, I came up with this very bare and simple layout:

Although much simpler alternatives existed, like using matplotlib, where I would not have to do anything (as the UI is pre-built with enough functionality and graphing equations is not difficult at all), it would not be challenging at all, because it only requires some understanding of python to implement it. The snip below shows how I can use matplotlib to not only present a clean looking UI, but also graph the graphs I want:



After creating this rough area division, I worked on converting the graphing area into looking like a real graph paper with the grids and axes. After my work, the result looked like the following:



To do this, I created individual squares of thickness 1 that I looped. There are 4 individual loops for the 4 quadrants of a graph. To create the axes, I created 2 lines of thickness 5 at the centre of screen. As of now, the graph is rigid and doesn’t move; and it also doesn’t zoom in or out. The code is below:

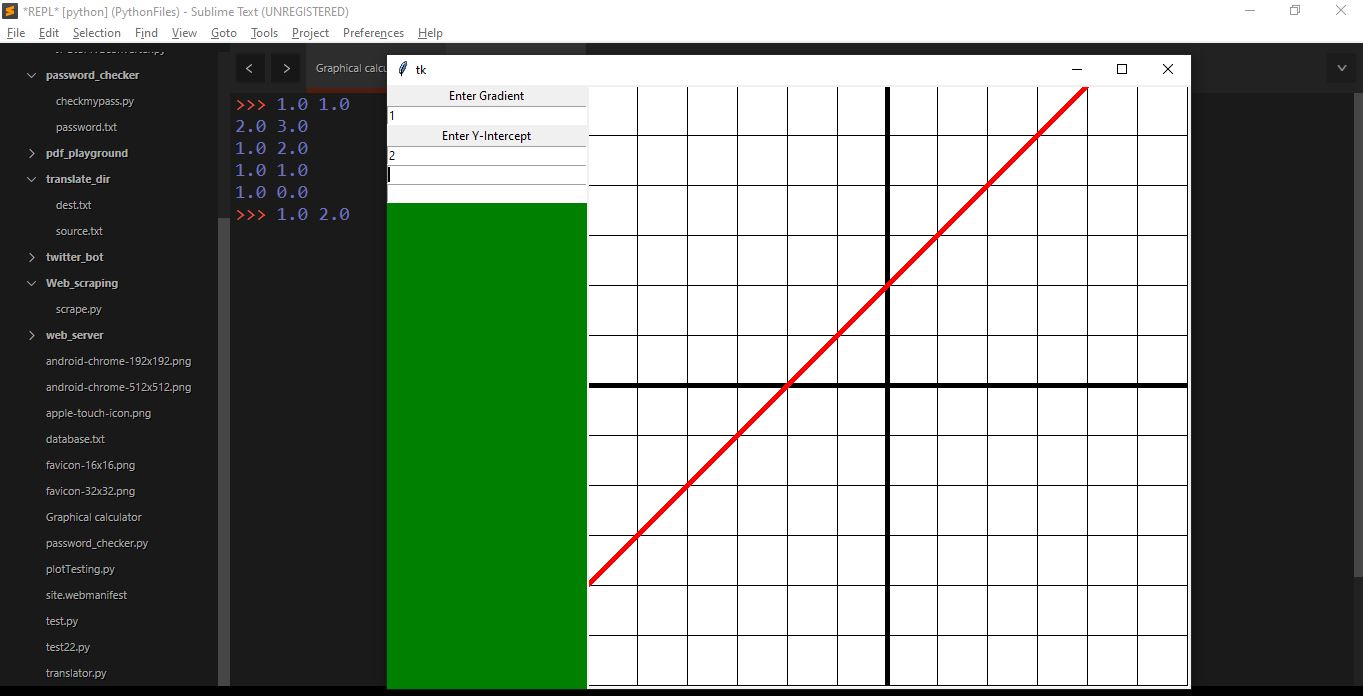


After a basic graphing area in place, I worked on creating the UI to enter the equations. To keep it simple, I only focused on linear equations. For some time in this documentation, you will see that I will use gradient as an entry for the co-efficient of x and the y-intercept as the constant, because

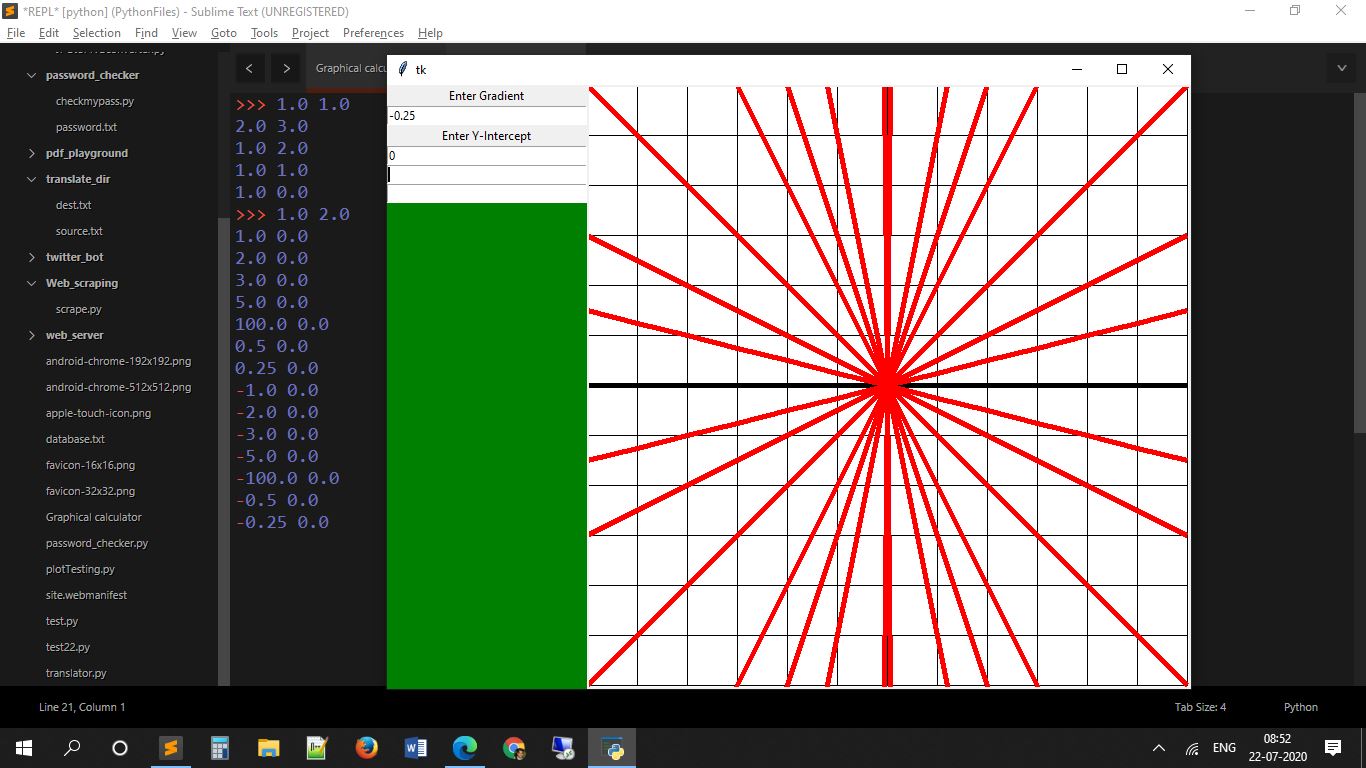
y = (*gradient*)x+(*y\_intercept*)

which is true for all linear equations.

Eventually, the program was working, but only for linear equations:

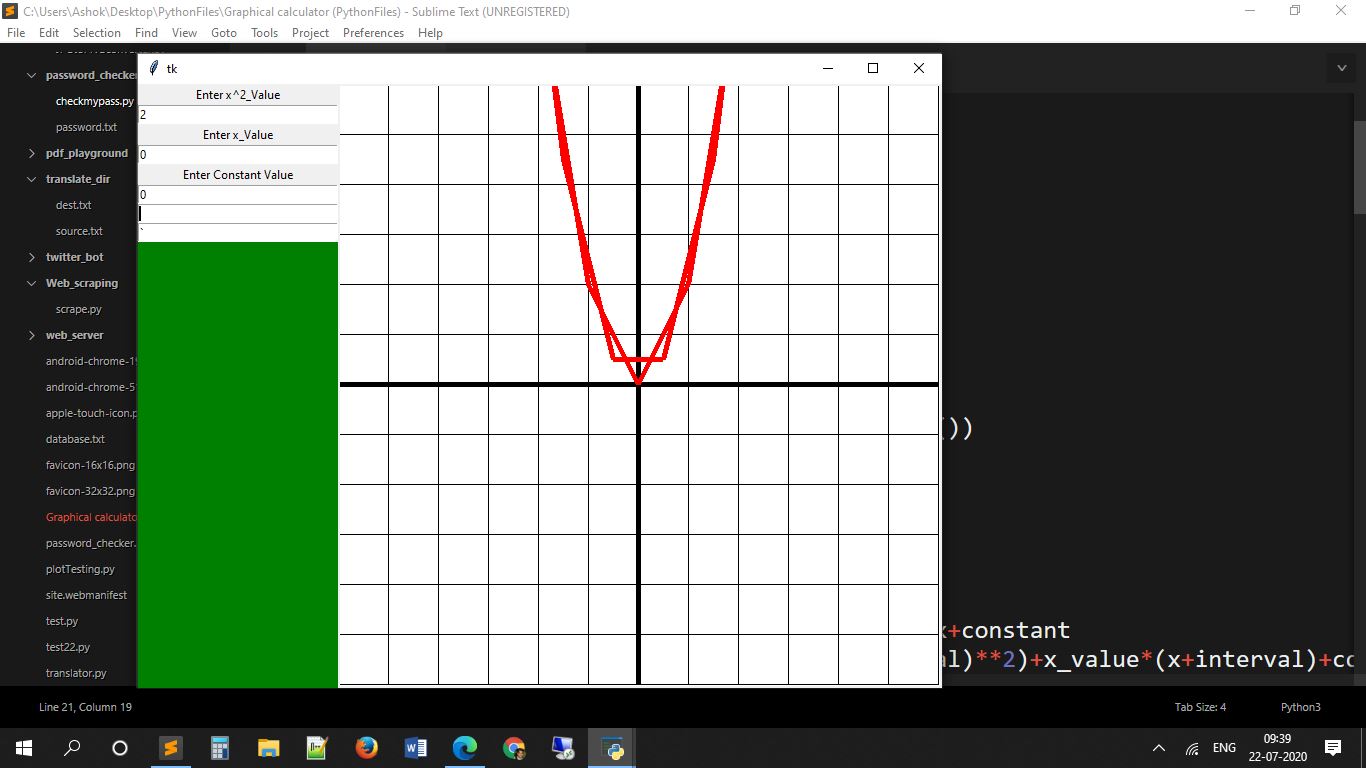


The program also allowed for the over-lapping of equations. Although, later on I think of creating separate slots for each separate equation; it still looks beautiful.

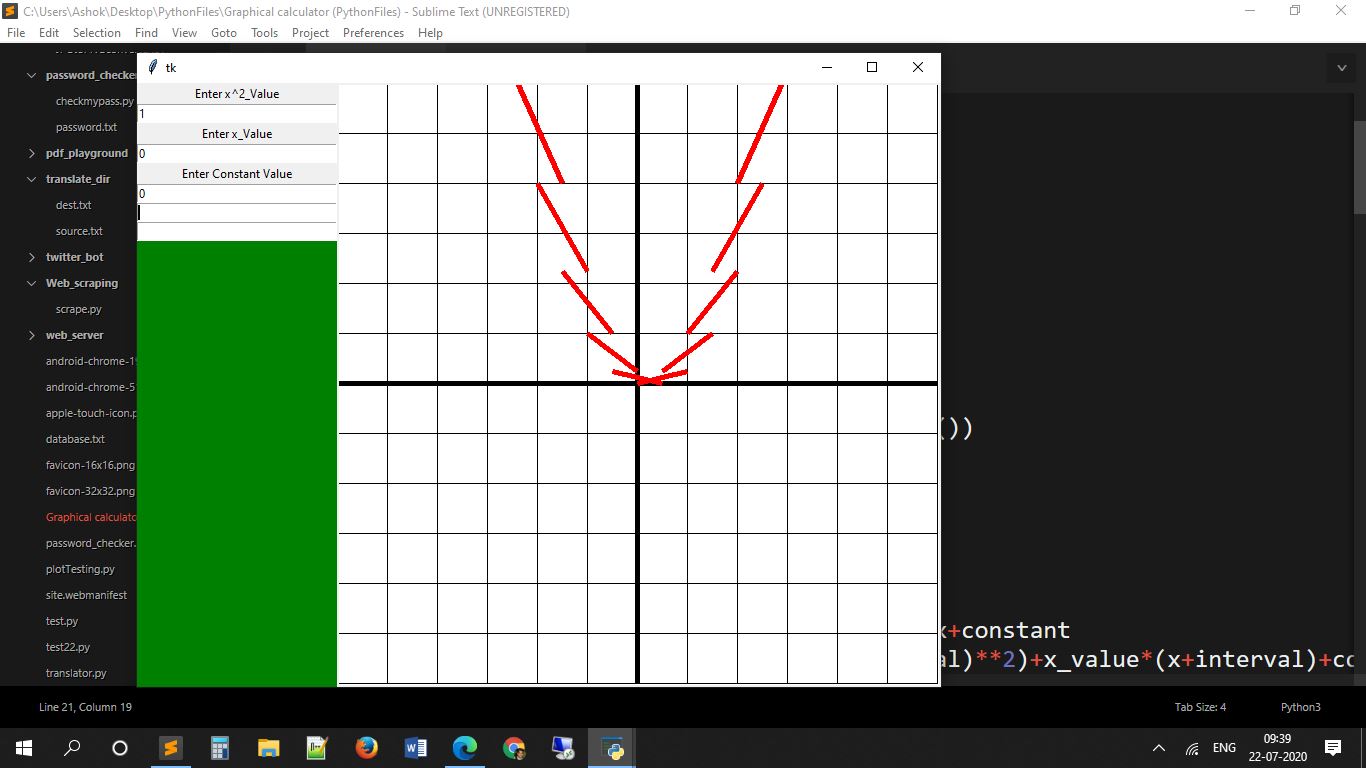


The next day, I tried expanding this to quadratic equations, but I faced some issues.

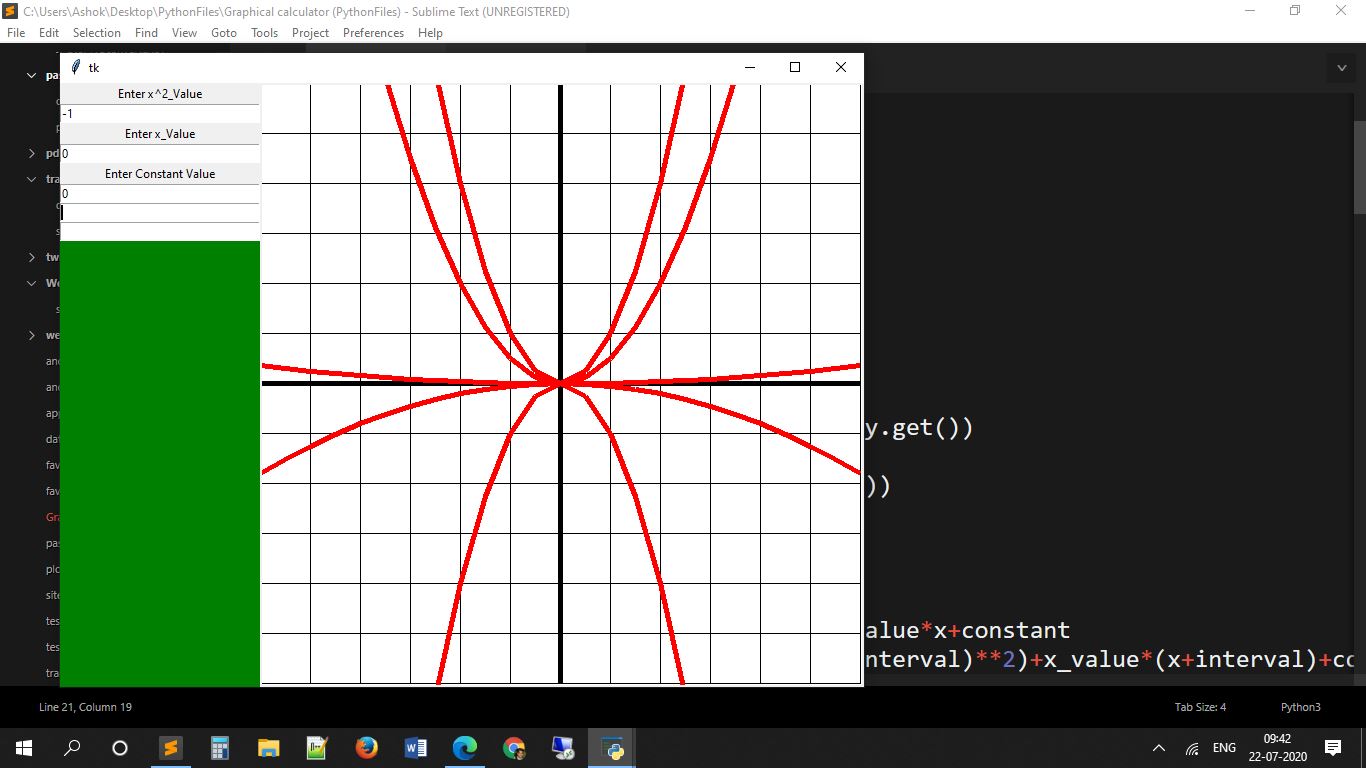
In my first attempt, the line was jagged, but the fix was very simple.



Later, the individual line segments were disjoint:

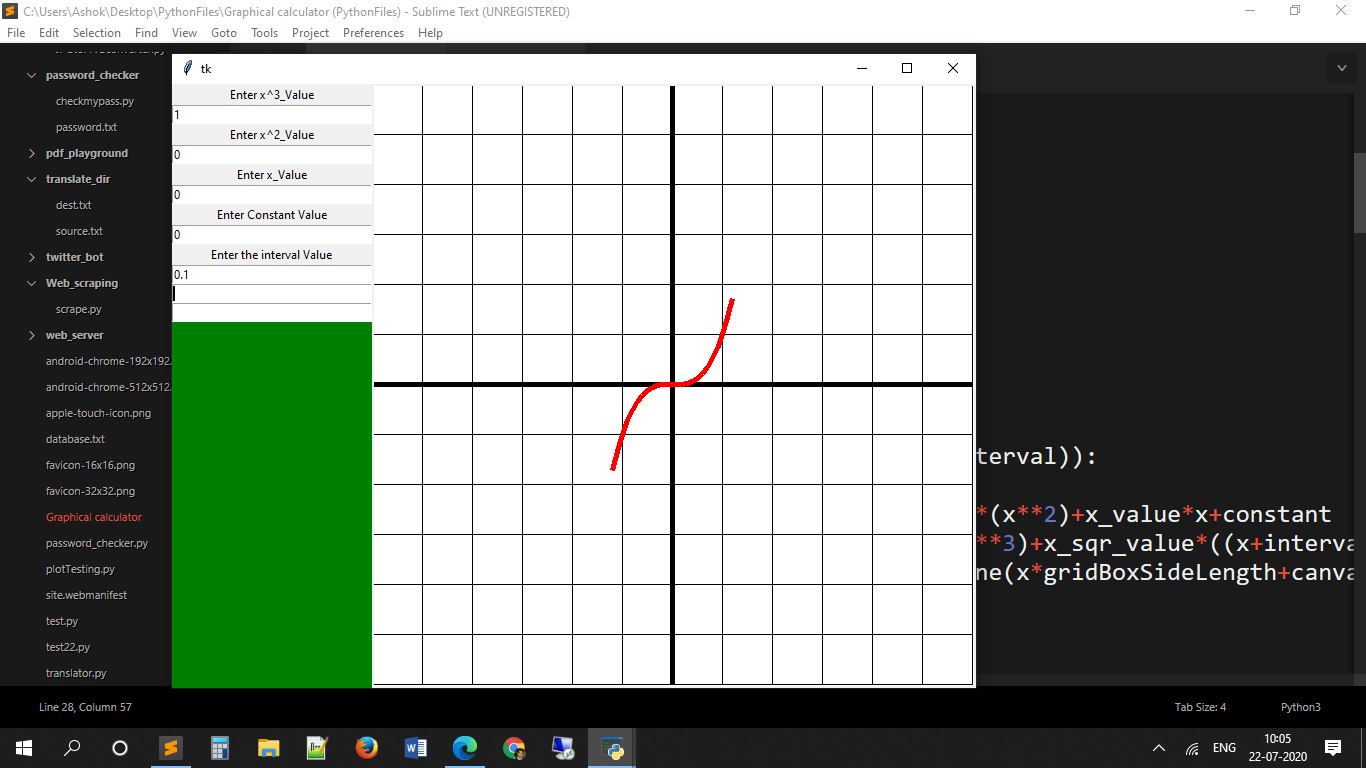


Finally this issue was fixed and some accurate graphs could be plotted.

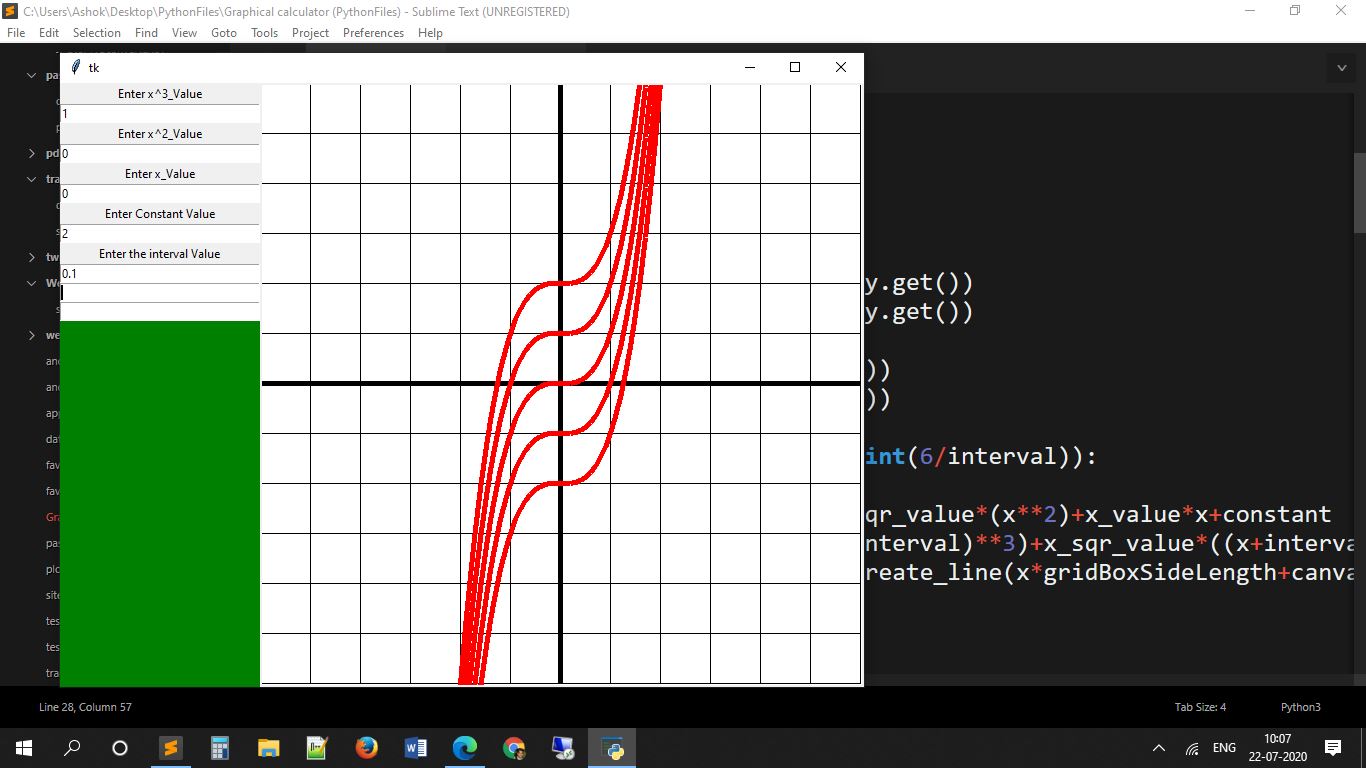


Even these graphs that have a smooth looking appearance are made out of straight lines. For the above image, there were 10 individual straight lines for each unit (length of 1 grid box) that gives the illusion of smoothness.

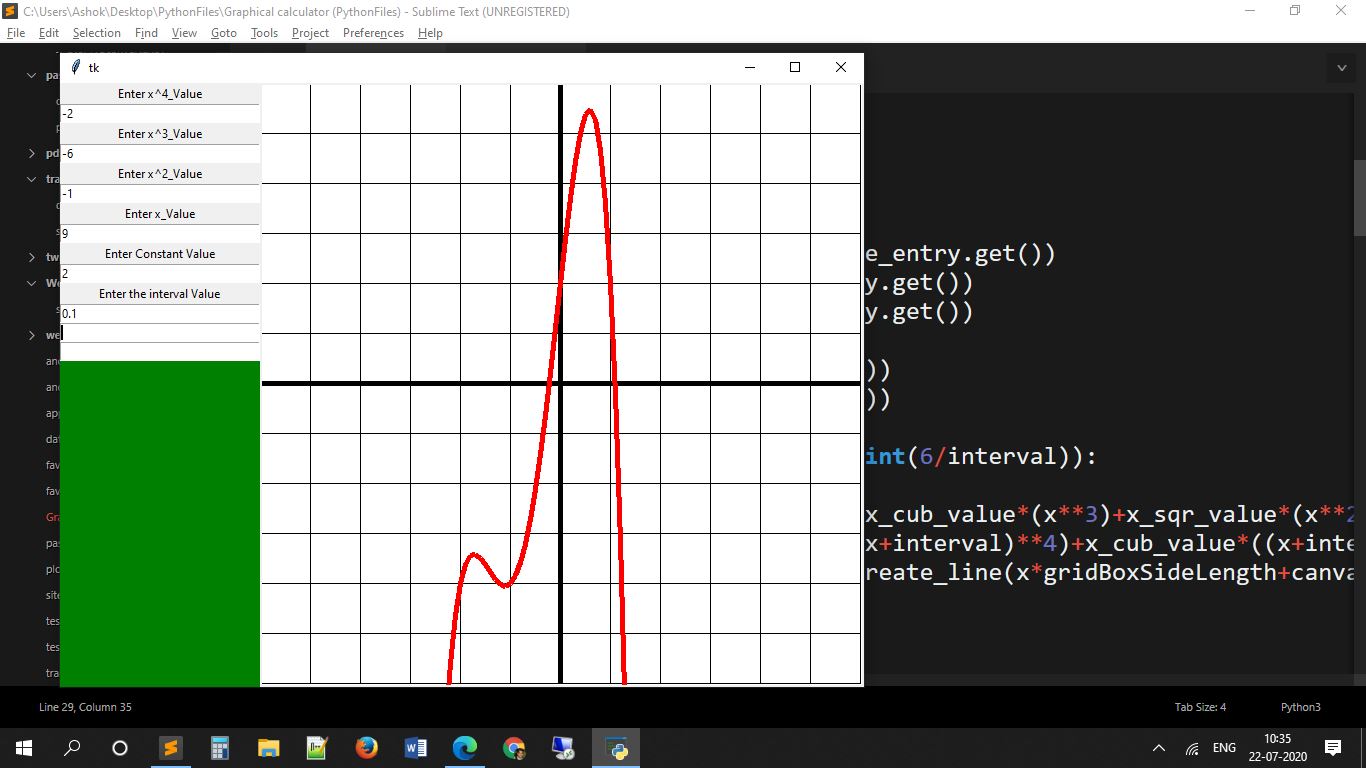
Expanding from linear to quadratic was the big leap. Now expanding to cubic is just a matter of copy-paste and small modifications. However, I faced a minor issue that was related with the scale of the graph in comparison to the grid.



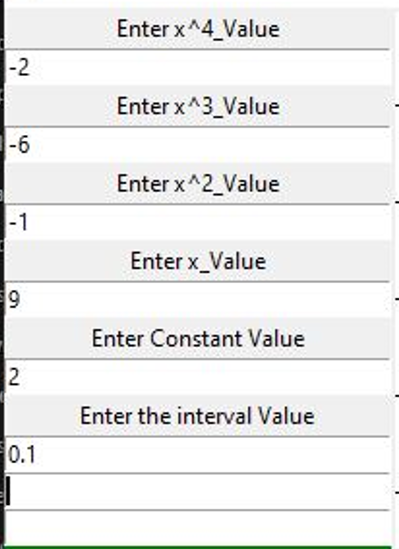
It was fixed, and I could start plotting cubic equations:



I then thought to myself, why stop here when the process of expanding to more degrees is so easy? I then added quartic functionality to the software:

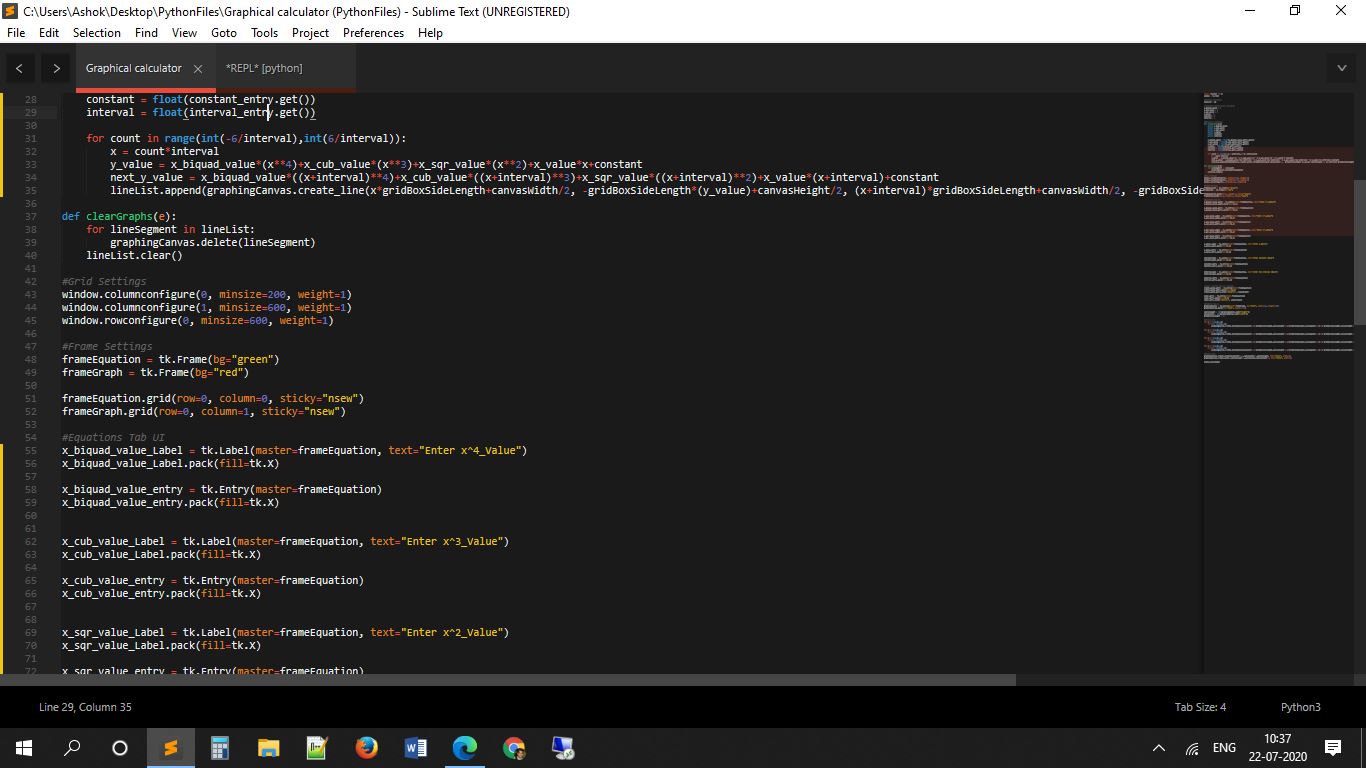


After adding this in, I started to imagine of the inconvenience of this method of mine. I have to individually fill in the coefficients for each term.



It also severely limits my application. If someone wishes to have a degree 100 equation to be plotted, I will not be able to ever add that functionality. Also, the lefthand-side UI bar would never be able to fit so much information. I was time to make the input system extremely flexible that can only be achieved by a change in approach. In my knowledge, regular expressions would be very handy for me in this situation, but I had to brush up on that knowledge first.

Even though a ton of code had already been written using this old system, I would have to now drastically change some parts of it. This was an overview of the amount of code needed to be changed:



**Logic Behind The Input system to accept any equation in the form y=ax^n+bx^(n-1)+cx^(n-2)….**

To begin with, I used regular expressions to find break an equation into various parts. The random jumble of characters below is infact a regular expression that I created for my purpose. This regular expression identifies the first term of an equation.

What is a term in an equation?

In the equation: y=5x^3-2x^2+6x-1

The terms of the equation are 3x^3, -2x^2, +6x, -1

The following regular expression identifies the first term:

algeabricTermDivider = re.compile(r"y?=?(((\+|\-)?\d\*\.?\d\*x?(?:\^\d\*\.?\d\*)?)\)?)")

when I execute this line of code, it retrieves the first term in the form of a string:

algeabricTermDivider.search(equation).group(1)

So, I will get 5x^3 as a result.

Next, I separate the co-efficient and exponent from the term.

Using coefficientPattern = re.compile(r"^(\+|\-)?\d\*\.?\d\*")

And this line of code: coefficientPattern.search(segmentBlock).group(0)

I get 5 from 5x^3 and I store this in a variable name var\_coefficient

Then, I extract the exponent:

Using exponentPattern = re.compile(r"x(?:\^(\+|\-)?\d\*\.?\d\*)")

And this line of code exponentPattern.search(segmentBlock).group(0)

I get 3 from 5x^3, and I store this is a variable named exponent

After this, I assemble the terms total value, by:

y\_val += var\_coefficient \* (x\*\*exponent)

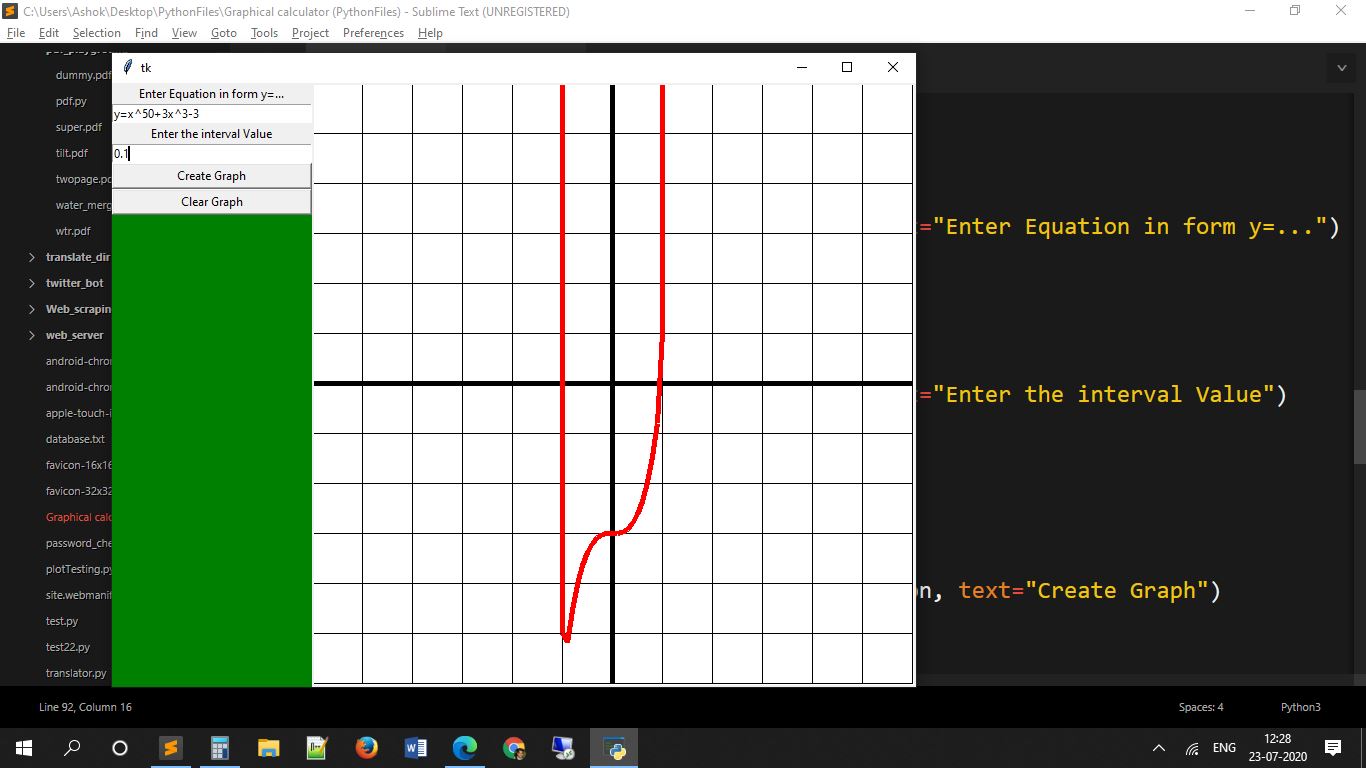
and I loop to do this again, however before doing that, I remove 5x^3 from the whole equation. This leaves -2x^2+6x-1

From here, the first term is -2^2

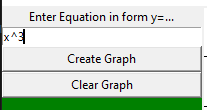
And the process repeats until the equation has no more length, or all the terms have been worked upon. This way, I am able to take any equation in the format mentioned in the subtitle, and I will be able to substitute a value of x in the equation. Later, I just loop the whole thing for many values of x, and then draw a graph between the yvalues of the xvalues to plot the graph for the equation.

Obviously this is a very big achievement, as I will now be able to plot equations above degree4. I am not limited to 4 degrees anymore. However, I will be unable to plot trigonometric functions with the current method, because no arrangement is made for it.

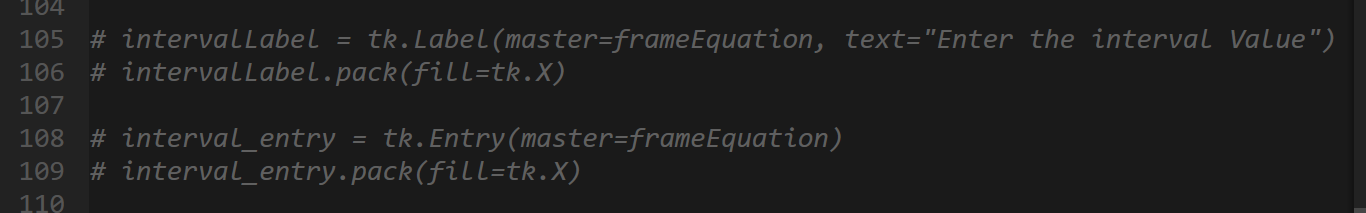
The image below shows how can plot graphs for absurd degrees with this logic change:

****

After showing the product in this stage to my maths teacher, she gave me a suggestion that I implemented, because it would increase the ease to use the product. I shortly after removed the option to change the increment (that directly affects how smooth the graph would be) because it is unnecessary that the user plays with that setting.



But the code for that functionality still remains in the main code hidden as a comment.



After enjoying the success of my program and testing it on different equations, I was exhausted after working for 4 hours straight. I finally took a break. But then I started thinking about how to integrate trigonometric functions that are necessary for any graphing calculator. My break only lasted 2 minutes, unfortunately, and I was back on.

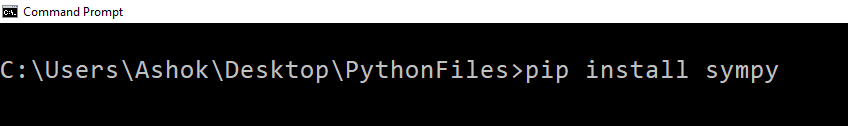
Now initially I thought that I would be able to do something with regular expressions as they had come in handy till now. So I set on to create a new and modified regular expression. This is what I came up with:

re.compile(r"y?=?(((?:\+|\-)?\d\*\.?\d\*)((?:sin|cos|tan|cot|sec|cosec)\()?((\+|\-)?\d\*\.?\d\*x?(?:\^\d\*\.?\d\*)?)\)?(\^\d\*\.?\d\*)?)")

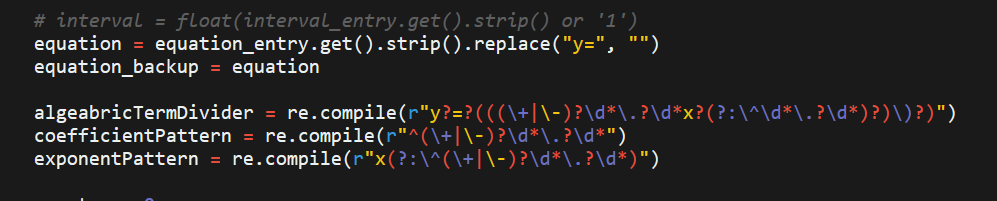
Even though this was *extremely easy* to understand, I began to think that the length of this regular equation would increase exponentially as I incorporated more and more features into the product. It would be shy of a few hundred characters at the end that was extremely ineffective. I sadly had to dump all that I had done till now and opt for a code restructure, that would let me easily add in trigonometric ratios, but later on logarithmic functions, etc. too.

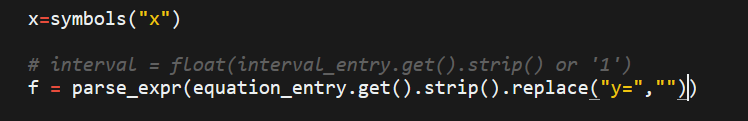
After a ton of research, from a stack overflow thread I found that there existed a module that would substitute values into a given function that has been specified, called sympy for python. This is what I needed.

After installing the module on my PC, I eventually incorporated it into the program and replaced all the existing regular expressions. Only one regular expression remains that I had created for converting user input to a format that sympy can understand.



The first image changed to the second:



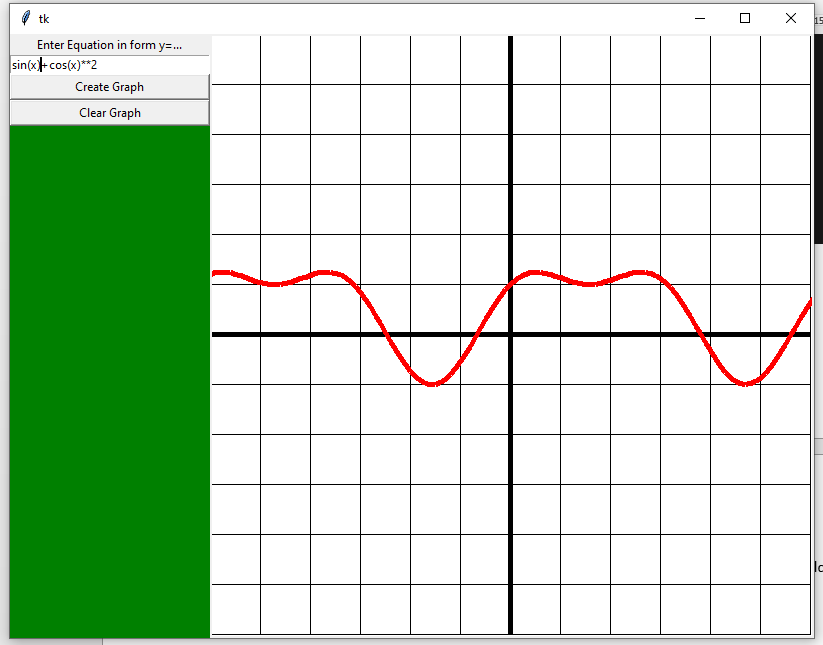


And the whole of the first image below changed to the second:

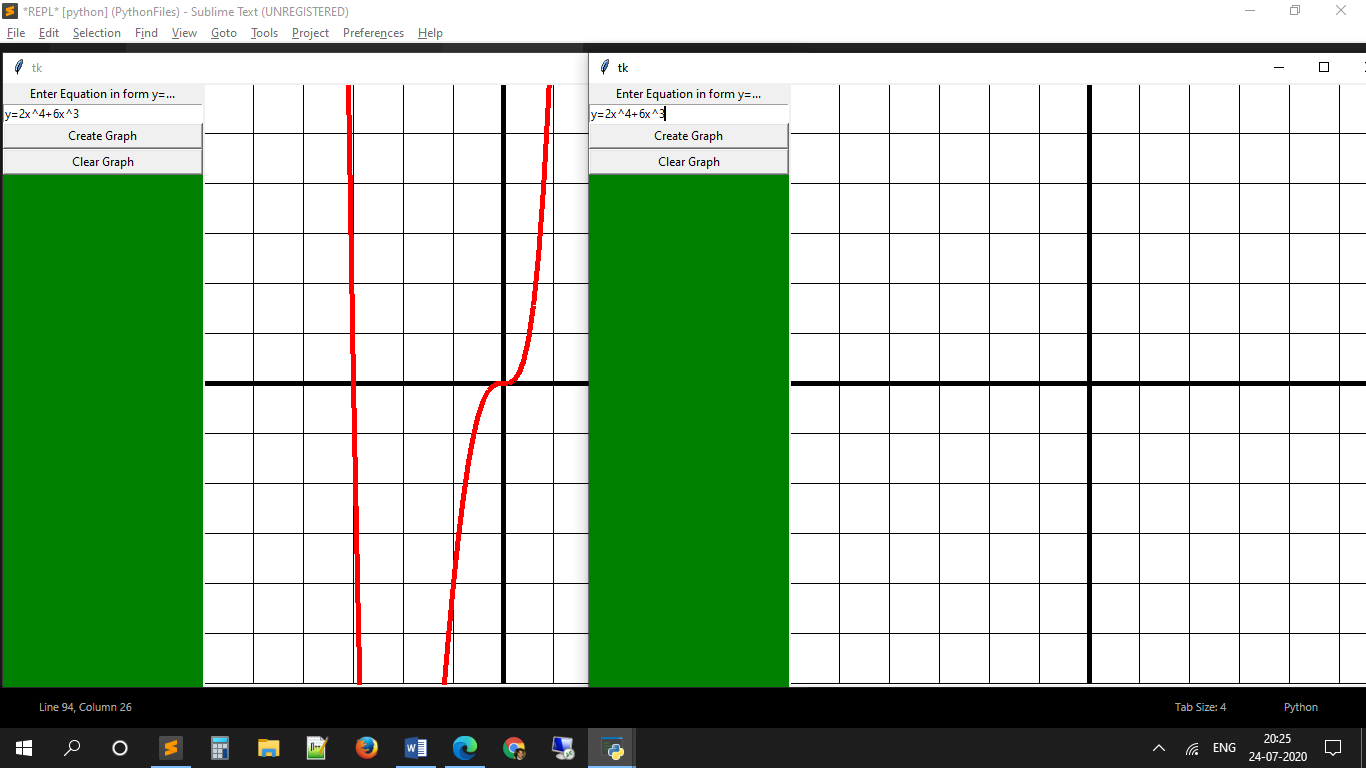




With the incorporation of sympy, not only could I do all that I could have done before, but I could also plot trigonometric equations!



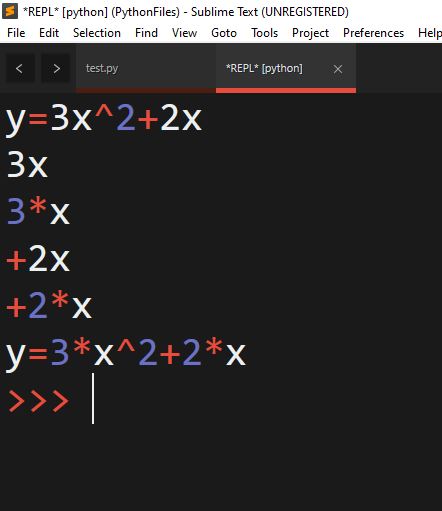
However, there were some issues that were faced. For example, sympy accepts exponent notation as ‘\*\*’ instead of ‘^’, but only python programmers use this syntax for an exponent. This was solved easily by replacing all ‘^’ that may come from user input to ‘\*\*’ before feeding it into sympy. Another example, sympy does not understand this notation of multiplication: 2x. 2x = 2\*x for humans, but sympy doesn’t understand that. The leftwards app uses the old version where I didn’t use sympy, but the regular expressions. Rightwards is sympy, that can’t understand the input due to the above reasons so doesn’t plot a graph:



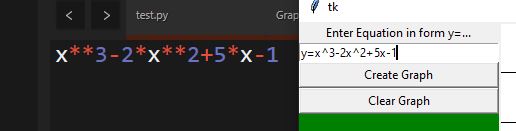
To solve this, I used a regular expression I made:

(?:\+|\-)?\d+\.?\d\*(?:x|sin|cos|tan|sec|cosec|cot)

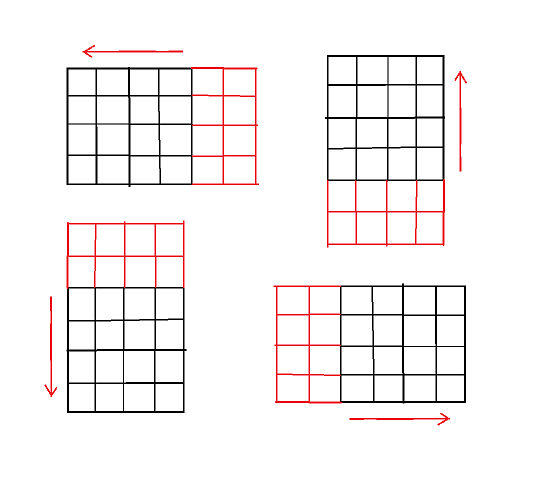
That would search for a number (that may even be a decimal) directly proceeded by ‘x’ or any of the trig functions notations (sin, cos , tan, etc.). The Image below gives a demonstration of how the modifications are conducted. The first line is the user input. The 2nd line is one of the 2 cases (in this equation) where 3 is multiplied by x. line 3 is after an asterisk is added to the 3 and x. Line 4 and 5 are the same as 3 and 4, but for the other case (2x). line 6 is end equation where invisible multiplication has been brought to sympy format by adding the obvious (for humans) asterisk sign denoting multiplication.



This was then coupled with the solution to replace ^ with \*\*. An example of it functioning properly is given below where right-side is the user entry and left is the converted form of it:



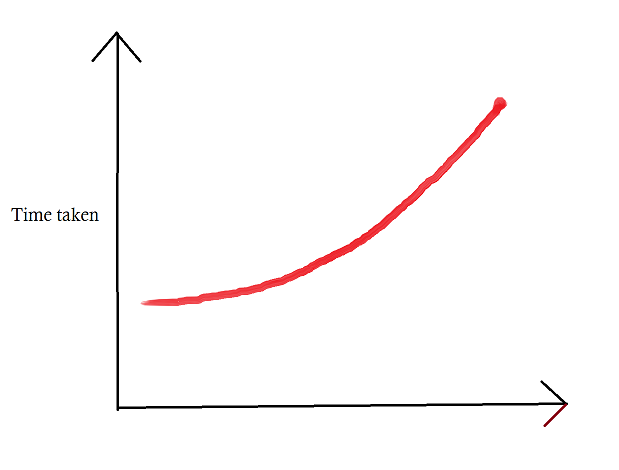
After adding so much flexibility into the input tab, I decided that I would have to add increased functionality at the graphing tab. These functionalities would be movement around the graph and magnification. Implementing these functionalilies were very difficult and required multiple days of constant effort. I began with the more fundamental function that was movement. At the beginning I invested unhealthy amounts of time to create a system in which only the minimum amount of grid boxes required to show to the user were made. I explain the working below.



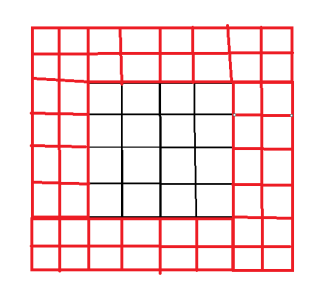
Taking the first diagram as an example, the arrows indicate that if the user wants to go in the right direction, the grid itself will actually be moved in the left direction and additional grid boxes (red in colour) will be made to the right so that those boxes never run out.

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| It requires very little time to begin as only one small square grid has to be created, and when the user moves it will extend the grid spontaneously | It has the effect of getting slower the more one moves because there is no way to my knowledge of deleting grid boxes at a certain distance in my set-up. So, eventually the program is moving thousands of boxes and the program crashes |
| At the start the speed is pretty high since only a few grid boxes are made and have to be removed. | It took me hours to make this, and in the end I had to scrap it due to the first disadvantage |

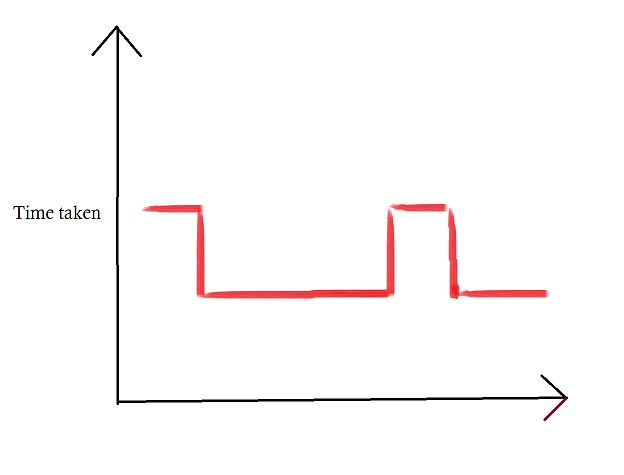
The graph below shows the amount of time it will take to move at set increments as I continue to move in one fixed direction. It increases exponentially. There will be a point when the program will crash.

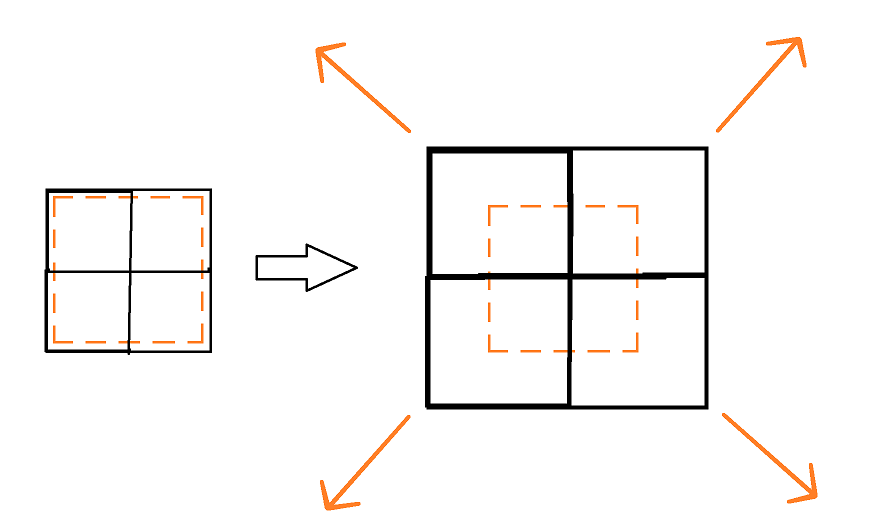


In the end, I stuck with a very simple method that was much easier to code and much easier to understand. In the final method, what happens is that a fixed number of grid boxes are created around the grid boxes that are in view of the user, and in the diagram these boxes are red. There is no addition of grid-boxes as the user moves; instead, when the user reaches the edge of the grid (including the red ones), the whole grid is deleted and a new grid is created around the visible grid boxes at the boundary.

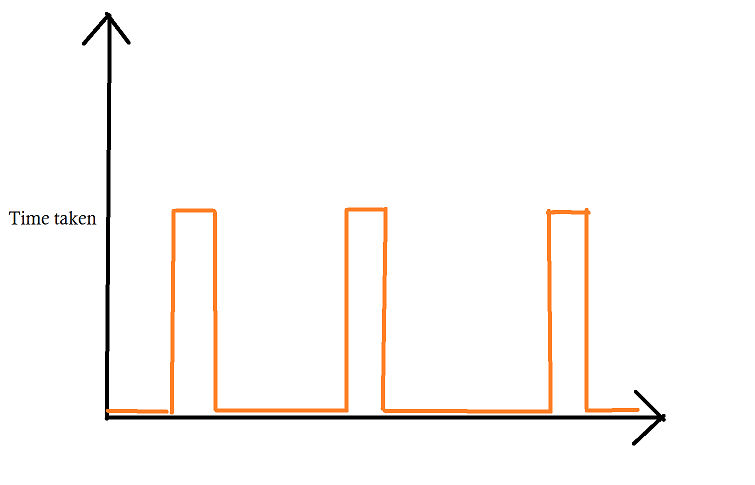


The only times the time taken increases is at the start and when a completely new grid needs to be created. Apart from those times, it moves at a constant speed.

The next function that I decided to add that is pretty standard and useful is magnification. Magnification by itself is very easy, all you need to do is increase the grid box side lengths, and replot the graph on this larger graphing paper. Yet, when we bring into account the integration of movement and magnification, it becomes much more difficult (since I had first integrated method 1 of movement with magnification, then later converted it to method 2). The diagram below attempts to show how magnification works. Here the orange box is what the viewer sees. The second diagram seems magnified to the viewer, because he sees less of a larger image.



Following is a graph for the time taken when magnification is in play. The peaks are when one changes magnification:



At the end, I began experimenting with inputs and was able to allow all standard inputs to be incorporated such as the mouse wheel for magnification, and dragging for movement. I even incorporated WASD keys for more precise movement (although it’s not of any real use).

After completing all of this, I began testing the software and began noticing issues whenever it had to plot infinity. This is because sympy recognises positive infinity (oo), negative infinity (-oo) and complex infinity (zoo). Tkinter on the other hand recognizes none. Also, in some cases sympy can give a NAN result (not a number). Hence, at the time of plotting the graph I nested it in if statements, with solutions to deal with each. My solution is simply to plot it as a very large number like 999999999 instead of infinity, etc.

