CPSC 231: Introduction to Computer Science for Computer Science Majors I

Assignment 2: Graphing Calculator

Weight: 7%

Collaboration

Discussing the assignment requirements with others is a reasonable thing to do, and an excellent way to learn. However, the work you hand-in must ultimately be your work. This is essential for you to benefit from the learning experience, and for the instructors and TAs to grade you fairly. Handing in work that is not your original work, but is represented as such, is plagiarism and academic misconduct. Penalties for academic misconduct are outlined in the university calendar.

Here are some tips to avoid plagiarism in your programming assignments.

1. Cite all sources of code that you hand-in that are not your original work. You can put the citation into comments in your program. For example, if you find and use code found on a web site, include a comment that says, for example:

   ```
   # the following code is from
   https://www.quackit.com/python/tutorial/python_hello_world.cfm.
   ```

   Use the complete URL so that the marker can check the source.

2. Citing sources avoids accusations of plagiarism and penalties for academic misconduct. **However, you may still get a low grade if you submit code that is not primarily developed by yourself. Cited material should never be used to complete core assignment specifications. You can and should verify and code you are concerned with your instructor/TA before submit.**
3. Discuss and share ideas with other programmers as much as you like, but make sure that when you write your code that it is your own. A good rule of thumb is to wait 20 minutes after talking with somebody before writing your code. If you exchange code with another student, write code while discussing it with a fellow student, or copy code from another person's screen, then this code is not yours.
4. **Collaborative coding is strictly prohibited**. **Your assignment submission must be strictly your code**. Discussing anything beyond assignment requirements and ideas is a strictly forbidden form of collaboration. This includes sharing code, discussing code itself, or modelling code after another student's algorithm. **You can not use (even with citation) another student's code.**
5. Making your code available, even passively, for others to copy, or potentially copy, is also plagiarism.
6. We will be looking for plagiarism in all code submissions, possibly using automated software designed for the task. For example, see Measures of Software Similarity (MOSS - https://theory.stanford.edu/~aiken/moss/).
7. Remember, if you are having trouble with an assignment, it is always better to go to your TA and/or instructor to get help than it is to plagiarize. A common penalty is an F on a plagiarized assignment.

Late Penalty

Late assignments will not be accepted.

Goal

Writing a program with loops and functions

Technology

Python 3

Submission Instructions

You must submit your assignment electronically. Use the Assignment 2 dropbox in D2L for the electronic submission. You can submit multiple times over the top of a previous submission. Do not wait until the last minute to attempt to submit. You are responsible if you attempt this and time runs out. Your assignment must be completed in **Python 3** and be executable with Python version 3.6.8+. For graphical drawing you **must** use the **Python turtle** library.

Description

Charting Expressions (Graphing Calculator)

You will be creating a small graphical **Python 3** program. This program will use the **turtle** library to draw based on information taken from the user. You should already have all the experience you need with this library from Assignment 1.

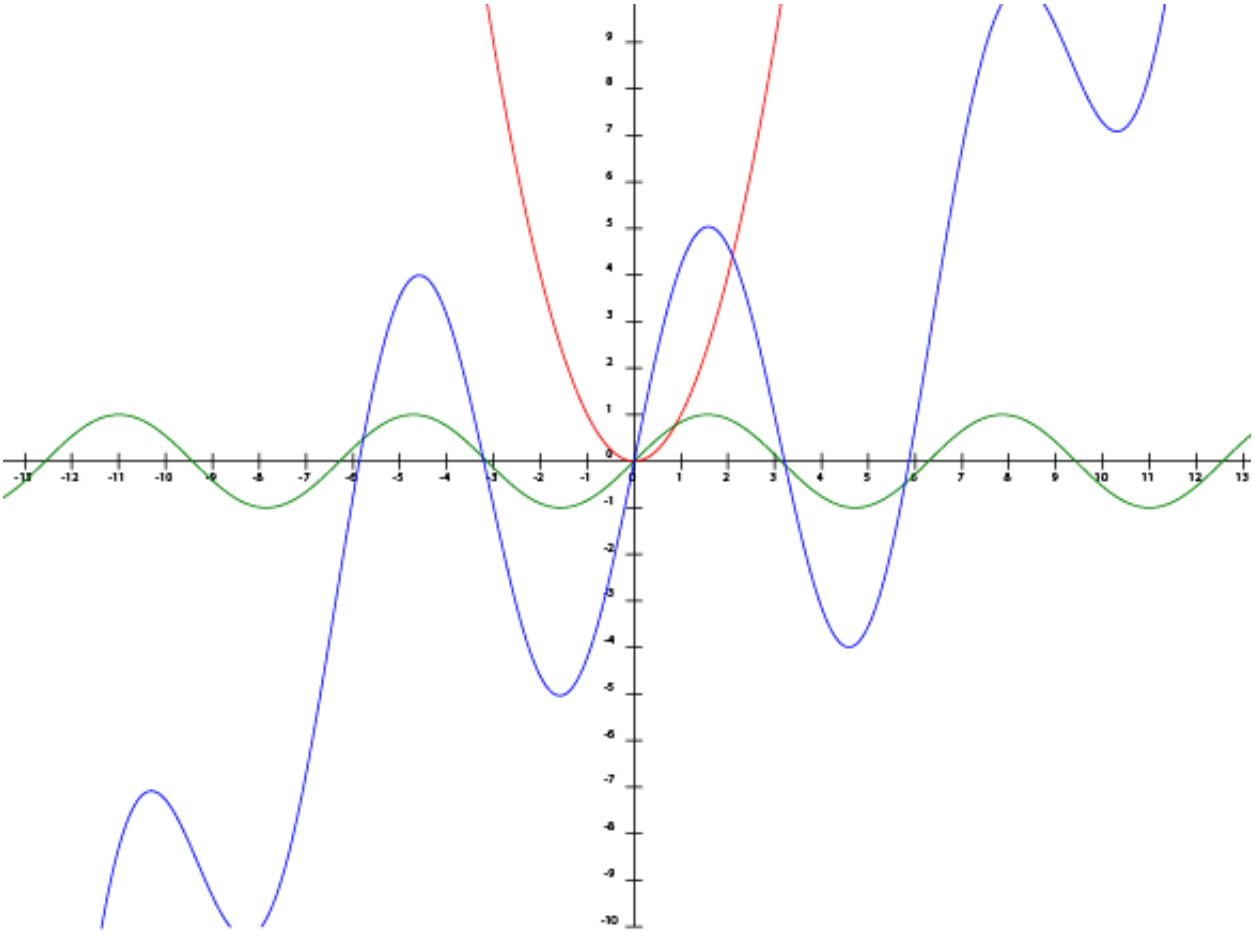
The assignment requires an additional understanding of converting between Cartesian coordinate systems, using loops, and completing functions as outlined by someone else. You will be provided with a starter code that already implements the basic interaction with the user (something you learned in the previous assignment). This code consists of a completed **main** function, which sets up the window using a **setup** function. There is also a **calc** expression that takes a string of an expression (Ex. expr="x**2") and returns the y for a given x value (Ex. expr="x**2" for x=3 will return 9) **None of these three functions should be changed.**

The **main** function makes use of several incomplete functions. Your assignment is to **complete these functions** so that the **main** function can use them to draw what the user requests. **All your code should be written within the incomplete functions. You are allowed to create constants and new functions as long as they are not duplications of existing ones.** There should be no global variables in your code (except if you are doing the bonus). There is a bonus that involves identify local minima/maxima and labelling them described at the end of the assignment.

Your program will be drawing in an 800-pixel by 600-pixel window with $(0,0)$ as the bottom left corner and $(800,600)$ as the top right corner. The starter code has a **setup()** function, which sets up the window and returns the turtle drawing object **pointer** so you can use it. This **pointer** is passed into the existing starter code functions for you to use. The included starter code already prompts the user for the pixel coordinates for where the chart origin should be placed in the window. The code then prompts the user for a ratio of how many pixels are 1 step of the chart. For example, an origin of $(x_o, y_o) = (400, 300)$ should be the centre of the screen. If the ratio of pixels per single step is 100 pixels per step. Then a position one step up from the origin point to $(x, y) = (0, 1)$ in the charting coordinate system is

$$(x_o + ratio * x, \quad y_o + ratio * y)$$
$$= (400 + 100 * 0, \quad 300 + 100 * 1)$$
$$= (400, \quad 300 + 100)$$
$$= (400, 400).$$

Below is an image of 3 curves drawn for a chart with an origin at 400,300 and a ratio of 30.



The existing starter code uses the included (but incomplete) functions to draw the x and y axes in black, and then loops to get expressions from the user. The colour used to draw an expression is determined based on the number of previously drawn expressions. So this number is tracked using a counter variable, that increases by 1 for each new expression entered.

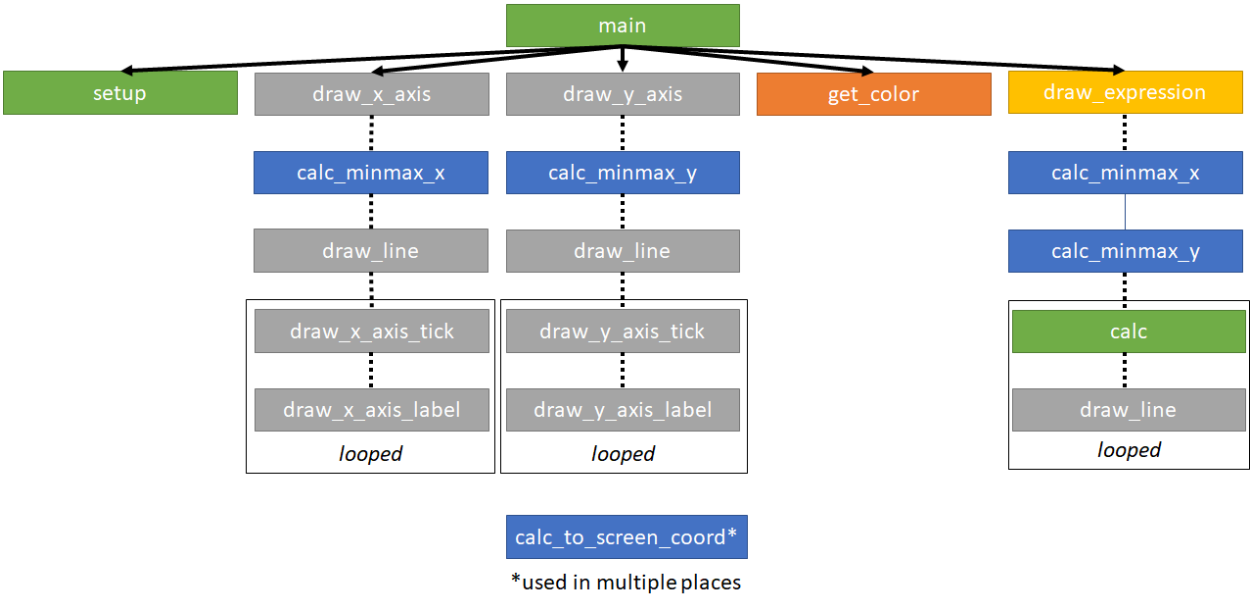
Your task is to complete the included functions, so they function in the way that the comments and assignment description require. You will likely find the visual examples of operation included in the assignment description helpful.

In the following diagram. Green functions are complete and provided. Order of parts is

Part 2 Blue – calc_to_screen_coord, calc_minmax_x, calc_minmax_y

Part 3 Grey – draw_x_axis, draw_y_axis, draw_line, draw_x_axis_label, draw_x_axis_tick, draw_y_axis_label, draw_y_axis_tick

Part 4 Yellow – draw_expression



I recommend approaching the functions in stages:

Part 1: get_color:

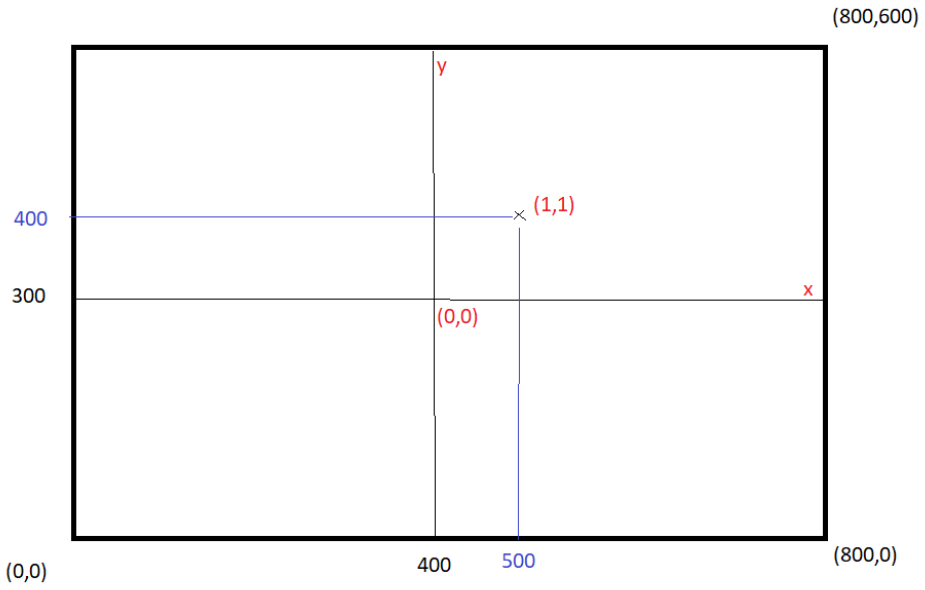
First, complete the colour-determination function **get_color**. This function takes the counter tracked in the expression input loop and returns one of three different colours: **red**, **green**, or **blue**. You should find the remainder division operator (also called modulus) helpful here because it allows you to change an infinitely increasing integer sequence into a range of integers.

Part 2: calc_to_screen_coord, calc_minmax_x, calc_minmax_y:

There are 3 sub-functions to this part. Each does not take a lot of code, but needs to be mathematically sound.

First, complete the coordinate conversion function **calc_to_screen_coord**. This function uses the parameters that the main function obtains user defined pixel coordinate system (**x_origin, y_origin, ratio**), along with a calculator coordinate system (x, y) to perform the conversion. The (**x_origin, y_origin**) is the pixel coordinates of where (**0, 0**) in the calculator should be located on screen, and ratio is how many pixels are equal to 1 in the calculator coordinate system. **calc_to_screen_coord** returns the (**x, y**) coordinate as a pixel screen coordinate (**screen_x, screen_y)** following the math described previously in the example. Use this function in your code any time you want to convert and draw a calculator location. You will lose marks if you duplicate the math/code functionality of this function somewhere else instead of using this function designed for that purpose.

The following image diagrams another example. In it an 800 by 600 window with a 400,300 origin and a ratio of 100. This would mean a (1,1) coordinate system point would fall at (500,400).



x_origin = 400, y_origin =300, ratio - 100

Now complete the other two functions. These functions are utility functions useful later in your code to determine what coordinates in each axis to draw values in between. To simplify things we will be deciding what axis value is just off the visible screen on each axis. **calc_minmax_x** is for the x-axis, **calc_minmax_y** is for the y-axis. For the x-axis we want to know the calculator location of **x=0** and **x=WIDTH** as these are the smallest and largest pixel locations we will be drawing. We can invert the math of calculating a pixel screen coordinate to determine what calculator x values match each of these. Then we will take the floor of 0-conversion value to be the min, and the ceiling of the WIDTH-conversion to be the max. We will do the same for the y-axis function, but for **y=0** and **y=HEIGHT**.

Ex. $x\_origin = 400, y\_origin = 300, ratio = 100$

$$screen_x = x_{origin} + x * ratio \qquad \text{for } screen\_x = 0$$

$$0 = 400 + x * 100$$

$$-400/100 = x$$

$$x = -4.0$$

$$min_x = int(floor(-4.0)) = -4$$

Part 3: draw_line, draw_x_axis, draw_y_axis (tick/label helpers)

Third, complete the functions that draw the x and y axes. These two functions are structurally similar, so complete one of them first then you can copy and modify it to complete the other function.

When first drawing your axes, you can ignore the tick marks and labels. These can be added in once you have the lines drawn successfully. Use your part 2 functions to determine the min/max calculator locations for the x-axis/y-axis. Then draw a line between those two locations. To draw this line complete and use the **draw_line** function.

To draw the ticks/labels my advice is to use loops. Start at the minimum location of the axis and loop until you reach the maximum location. Loop through each label location and then complete and use the **draw_x_axis_label, draw_x_axis_tick** functions (or y-axis variants) to draw the label/tick for that location.

Part 4: draw_expression

Finally, you will draw the actual expression that the user inputted. Python can evaluate a string using its **eval** function. Your code does not need to be capable of anything more than the **eval** function can do. For example, the following 3-line program is a simple calculator that works for input like "1+1":

```
expression = input("Enter an arithmetic expression: ")
result = eval(expression)
print("The result of that expression is", result)
```

Python is also able to evaluate expressions which include a single variable **x**. The implementation of this function can be surprising at first. When the **eval** function is called, it uses whatever value is stored in the variable called **x** to evaluate the given expression.

To make things easier a **calc** function has already been completed that will take an expression and an x and return the y value. (Ex. **y = calc(expression, x)**). You will use the **eval()** method to do all of your math for you for the calculator. You will notice that we have the import at the top **from math import *** which allows **eval**() to use things like cos/sin.

As a result, your program will need to include a variable named **x**, which represents the **x** coordinate in the calculator coordinate system. The value of **x** will change in a loop. Call **calc** inside of the loop so that you can compute the value of **y** for many different **x** values. For example, the following program evaluates an expression that includes the variable **x** for each integer value of **x** from 0 up to and including 5:

```
expr = input("Enter an arithmetic expression: ")
for x in range(0, 6):
        y  = calc(expr, x)
        print("When x is", x, "the value of the expression is", y)
```

You can use a similar technique to compute the **y** position of the curve for many different values of **x**. Those **x** and **y** values are what you need to draw short lines with turtle that will visually emulate the expression's lines and possible curvature.

You cannot complete this function correctly unless the previous functions are operational. In particular, it will be useful to use **calc_minmax_x** to determine which values of **x** you want to draw a line between. You will need to pick a **delta** that makes the curve smooth. A **delta** of 0.1 or 0.2 should provide sufficient smoothness. Note that, once you are using floating point values for your steps size, you will be unable to use a **for-range()** loop to loop through them but instead should use a **while** loop.

Note: Two challenges you may have as you implement expression drawing are:

(1) parts of the expression that exist outside the visible drawing window and,
(2) discontinuities in expressions (Ex. tan(x)).

For expressions outside the draw window (for example y values out of screen), you are free to draw these parts of the expression, or alternately you may implement conditionals that will avoid drawing them. **calc_minmax_y** will assist if you choose to do choose the second solution as you can use it to determine the minimum and maximum y axis values for the drawing window.

For expressions with discontinuities, or undefined points, you can write your code under the knowledge they will either not be considered valid input, or if they are given as input, then the discontinuity does not have to be drawn. (Ex. depending on your solution tan(x) may end up being drawn with a line connecting across the vertical discontinuity that exists at every odd multiple of π/2. This is an acceptable submission.)

Additional Specifications:

Ensure that your program meets all the following requirements:

- You should have the class, your name, your tutorial, your student id, the date, and description at the top of your code file in comments. Marks are given for these.
- The code file should be CPSC231F21A2-<Name>.py (ex. Mine would be CPSC231F21A2-Hudson.py)
- Do not import more than math/turtle as libraries
- Use constants appropriately. Your TA may note one or two magic numbers as a comment, but more will result in lost marks.
- Use descriptive variable names. Using the same variable names that appear in mathematical equations is OK.
- Draw your axes black, alternate your colours for curves through red, green, and blue.
- Use in-line comments to indicate blocks of code and describe decisions or complex expressions.
- Do not put any code outside the functions given. You can add functions, but none should duplicate functionality described in a current function. For example, don't create functions to draw the labels/tick marks as two already exist.
- Break and continue commands are generally considered bad form. As a result, you are **NOT** allowed to use them in this assignment. In general, their use can be avoided by using a combination of if statements and writing better conditions on your while loops.
- You do not need to perform any error checking in your program. You may assume that the user always enters a valid expression or a blank line indicating that they want to quit. A valid expression contains x, operators (+/-,*,/), numeric constants (1, 4.5) and standard mathematical functions (sin, cos).

Bonus:

Looking for an A+?

Improve the program so that it marks every local minimum in orange and every local maximum in purple. Do this for every one found between **min_x** and **max_x** calculated by **calc_minmax_x**. You should only consider points that you calculated using the **calc** method during your **draw_expression** loop. You do not, and should not, try to analytically determine these in a more complicated way as there is no more credit to be gained if you do so.

Use a small circle to do this to indicate the location of these points. Create a function that draws a circle for you around a given point.

Make sure that your program does not mark inflection points that aren't minima/maxima, such as (0, 0) when graphing y = x ** 3 (and other similar functions that include one or more inflection points that are not minima/maxima).

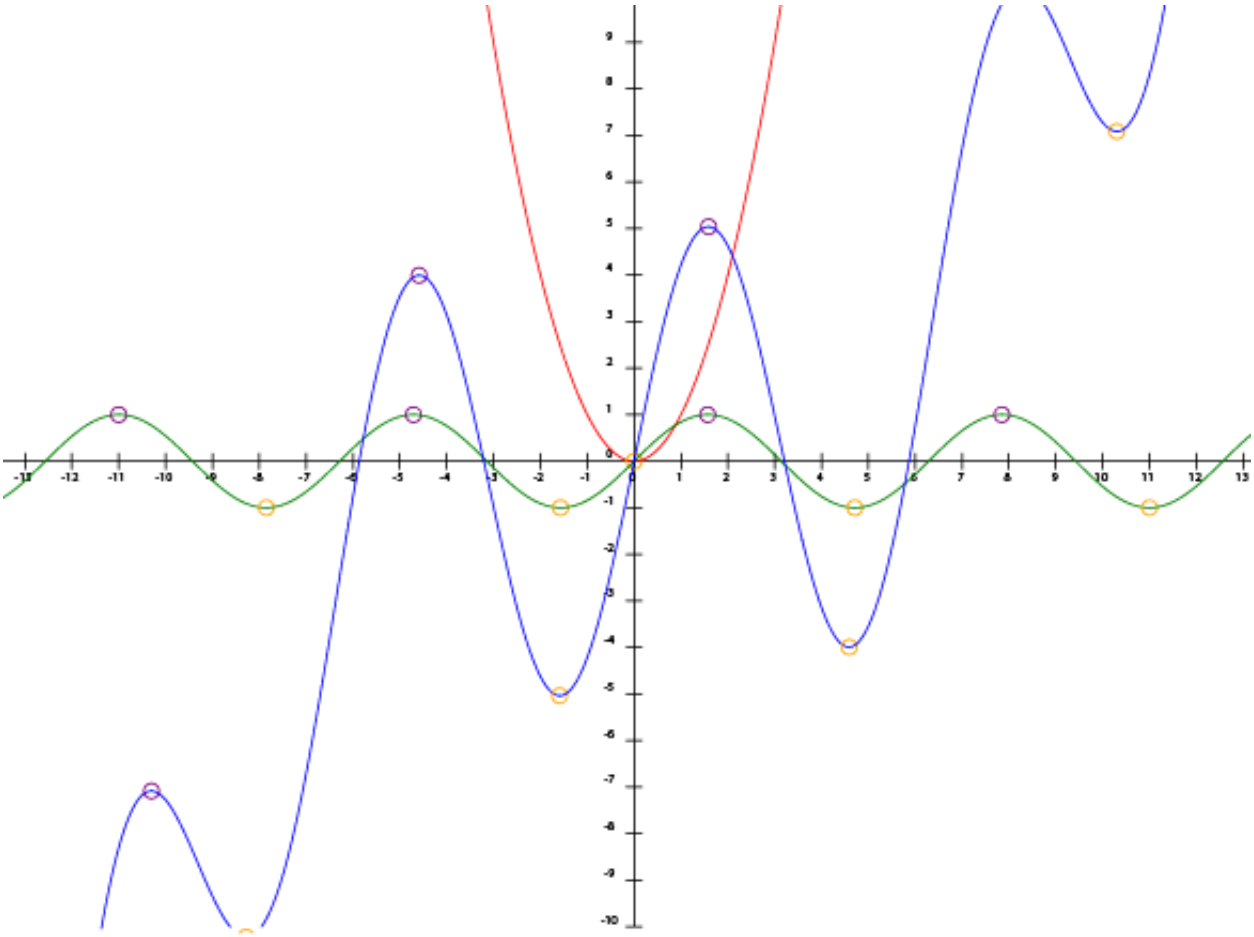
1. At that same time as you draw, print the largest local minimum (global maximum) and lowest local minimum (global minimum) to the shell/terminal window **for each expression** entered (ignore infinite values).
2. Concurrently, track the largest global maximum and global minimum **across all expression** entered. To track these values across expressions, please use **global** variables. These variables should be the only **global** variables in your program.

Print to the shell/terminal/console the updated values tracked for each of these two requirements after each expression is entered.

Bonus Example:

The following image shows the result of plotting three curves (w/ bonus). User input is shown in bold. (It is the bonus variant as the local min/max are circled as well)

Enter pixel coordinates of chart origin (x,y): **400,300**
Enter ratio of pixels per step: **30**
Enter an arithmetic expression: **x**2**
Enter an arithmetic expression: **sin(x)**
Enter an arithmetic expression: **0.01*x**3+5*sin(x)**
Enter an arithmetic expression: **<Enter>**



Note, the exact values for min/max values will vary between solutions based on method and accuracy of method chosen to find them. Sometimes more than one location is tied (such as for sin(x)), in such case any of the tried minima/maxima is just as valid. TAs will give credit as long as answer are approximately correct.

Example Output for Bonus: (not the only output)
Enter pixel coordinates of chart origin (x,y): **400,300**
Enter ratio of pixels per step: **30**

Enter an arithmetic expression: **x**2**
No expression global maximum

No global maximums
Global minimum for all expressions (0.00000, 0.00000)

Enter an arithmetic expression: **sin(x)**
Expression global maximum (1.57000, 1.00000)
Expression global minimum (-1.57000, -1.00000)
Global minimum for all expressions (1.57000, 1.00000)
Global minimum for all expressions (-1.57000, -1.00000)

Enter an arithmetic expression: **0.01*x**3+5*sin(x)**
Expression global maximum (8.28000, 10.22973)
Expression global minimum (-8.28000, -10.22973)
Global minimum for all expressions (8.28000, 10.22973)
Global minimum for all expressions (-8.28000, -10.22973)

Enter an arithmetic expression: **<Enter>**

Grading:

Part one of the assignment will be graded out of 12, with the grade based on the program's level of functionality and conformance to the specifications. To get more than an F, your code must not have syntax errors. Code with runtime errors that occur during proper usage of the program, every time it runs, will not get better than a C grade. Runtime errors are your program crashing.

Your TA will begin by grading your code with a general functionality grade starting point and subtract marks when smaller specifications are unfilled.

The total mark achieved for the assignment will be translated into a letter grade using following table:

Submit the following using the Assignment 2 Dropbox in D2L:

1. CPSC231F21A2-Name.py

Grading:

Mark	Letter Grade	Starting Point Guidelines (not a final grading scheme!)
13	A+	Appears to fulfill assignment and bonus spec
12	A	Appears to fulfill assignment spec
11	A-	
10	B+	
9	B	Code draws axes and is partially able to draw curves
8	B-	
7	C+	
6	C	Code draws axes but can't draw curves
5	C-	
4	D+	
3	D	Code has only get_color/calc_to_screen_coord completed
0-2	F	Syntax errors or barely started code

As a reminder, the University of Calgary assigns the following meaning to letter grades:

A: Excellent – Superior performance showing a comprehensive understanding of the subject matter

B: Good – Clearly above average performance with generally complete knowledge of the subject matter

C: Satisfactory – Basic understanding of the subject matter

D: Minimal Pass – Marginal performance; Generally insufficient preparation for subsequent courses in the same subject

F: Fail – Unsatisfactory performance