Foundational Math 2

July 4, 2024

Learn Foundational Math 2 by Building Cartesian Graphs Each of these steps will lead you toward the Certification Project. Once you complete a step, click to expand the next step.

1 ↓ Do this first ↓

Copy this notebook to your own account by clicking the `File` button at the top, and then click `Save a copy in Drive`. You will need to be logged in to Google. The file will be in a folder called "Colab Notebooks" in your Google Drive.

2 Step 0 - Acquire the testing library

Please run this code to get the library file from FreeCodeCamp. Each step will use this library to test your code. You do not need to edit anything; just run this code cell and wait a few seconds until it tells you to go on to the next step.

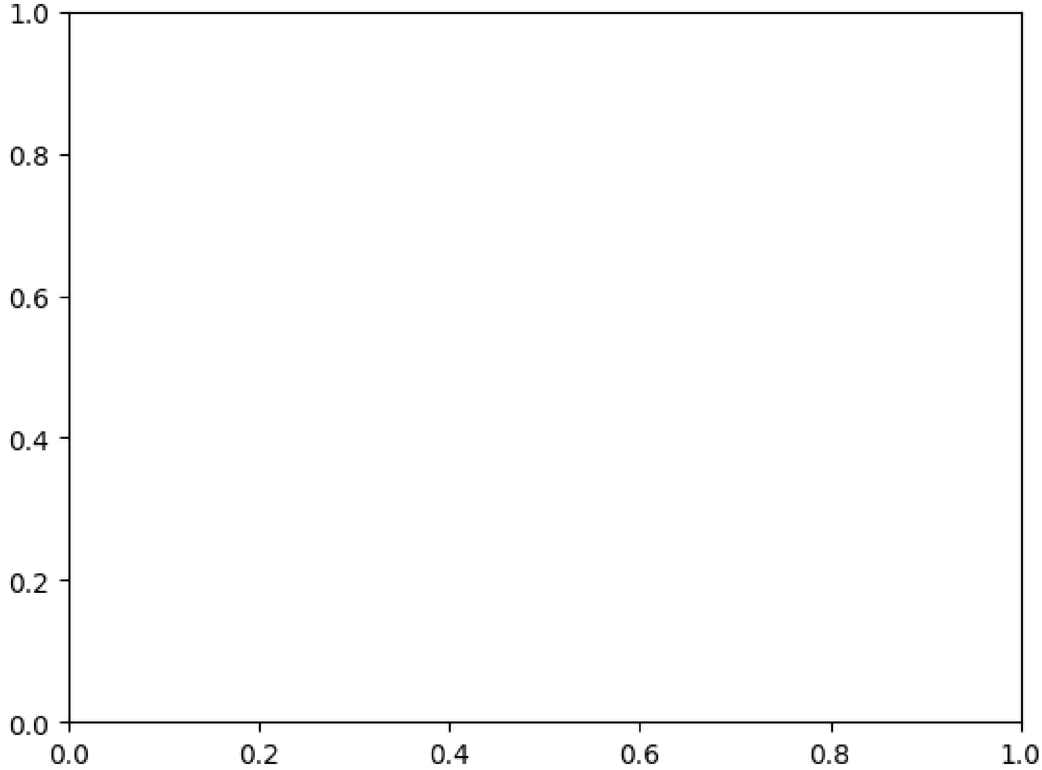
[1]:
```
# You may need to run this cell at the beginning of each new session

!pip install requests

# This will just take a few seconds

import requests

# Get the library from GitHub
url = 'https://raw.githubusercontent.com/edatfreecodecamp/python-math/main/
 ↪math-code-test-b.py'
r = requests.get(url)

# Save the library in a local working directory
with open('math_code_test_b.py', 'w') as f:
    f.write(r.text)

# Now you can import the library
import math_code_test_b as test

# This will tell you if the code works
test.step01()
```

```
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-
packages (2.31.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests) (2024.6.2)
Code test Passed
Go on to the next step
```

3 Step 1 - Cartesian Coordinates

Learn Cartesian coordinates by building a scatterplot game. The Cartesian plane is the classic x-y coordinate grid (invented by René DesCartes) where "x" is the horizontal axis and "y" is the vertical axis. Each (x,y) coordinate pair is a point on the graph. The point (0,0) is the "origin." The x value tells how much to move right (positive) or left (negative) from the origin. The y value tells you how much you move up (positive) or down (negative) from the origin. Notice that you are importing `matplotlib` to create the graph. The following code just displays one quadrant of the Cartesian graph. Just run this code to see how Python displays a graph.

[2]:
```python
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
plt.show()

# Just run this code to see a blank graph
import math_code_test_b as test
test.step01()
```

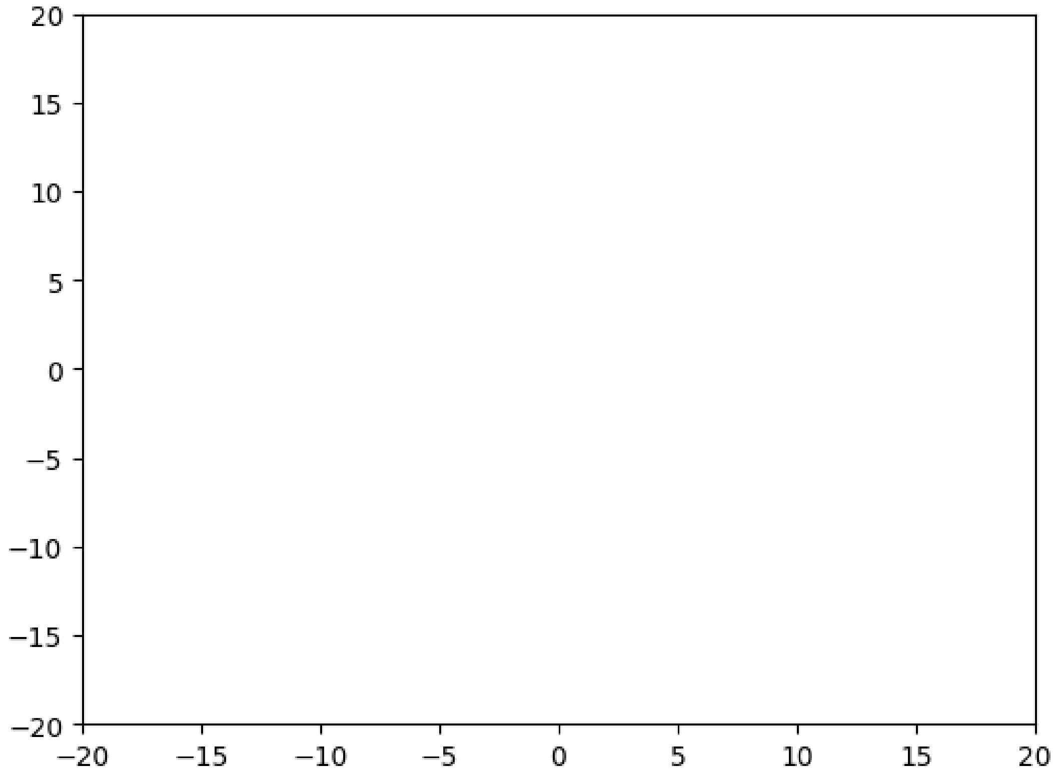



```
Code test Passed
Go on to the next step
```

4 Step 2 - Cartesian Coordinates (Part 2)

Here you will create a standard window but still not highlight each axis. Run this code once, then change the window size to 20 in each direction and run it again.

[3]:
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

# Only change the numbers in the next line:
plt.axis([-20,20,-20,20])

plt.show()


# Only change code above this line
import math_code_test_b as test
test.step02(In[-1].split('# Only change code above this line')[0])
```



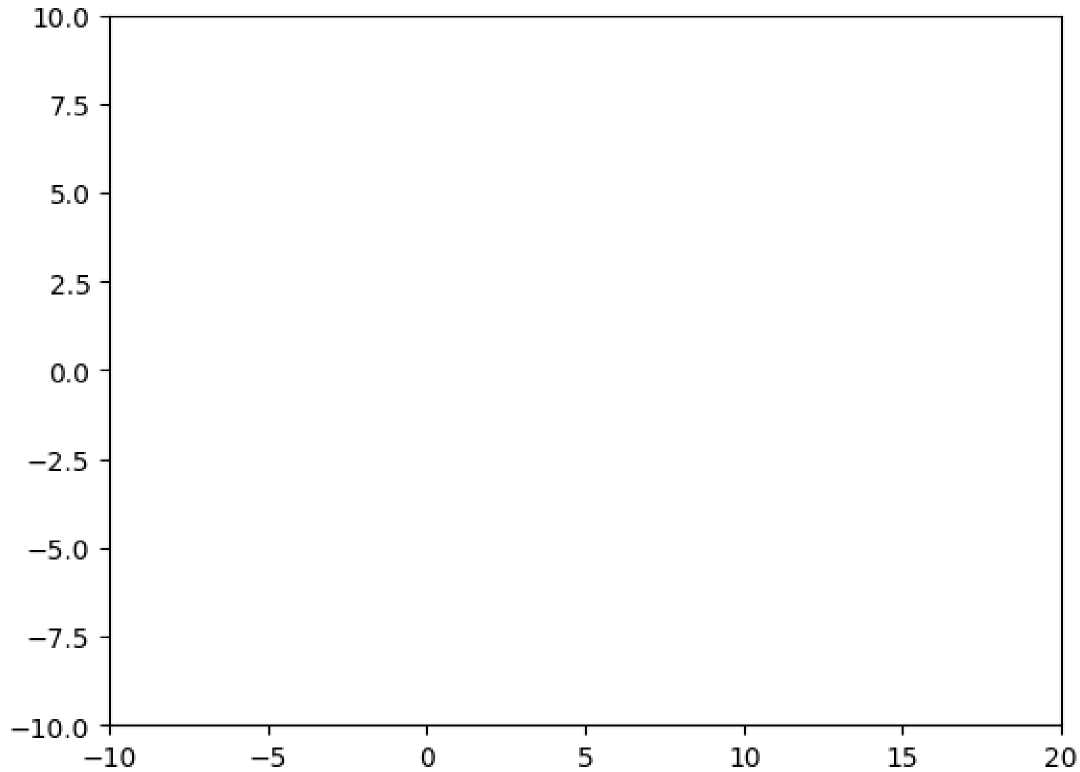

```
Code test passed
Go on to the next step
```

5 Step 3 - Graph Dimensions

When you look at this code, you can see how Python sets up window dimensions. You will also notice that it is easier and more organized to define the dimensions as variables. Run the code, then change just the `xmax` value to 20 and run it again to see the difference.

[4]:
```python
import matplotlib.pyplot as plt

xmin = -10
xmax = 20
ymin = -10
ymax = 10

fig, ax = plt.subplots()
plt.axis([xmin,xmax,ymin,ymax]) # window size
plt.show()

# Only change code above this line
```

```
import math_code_test_b as test
test.step03(In[-1].split('# Only change code above this line')[0])
```



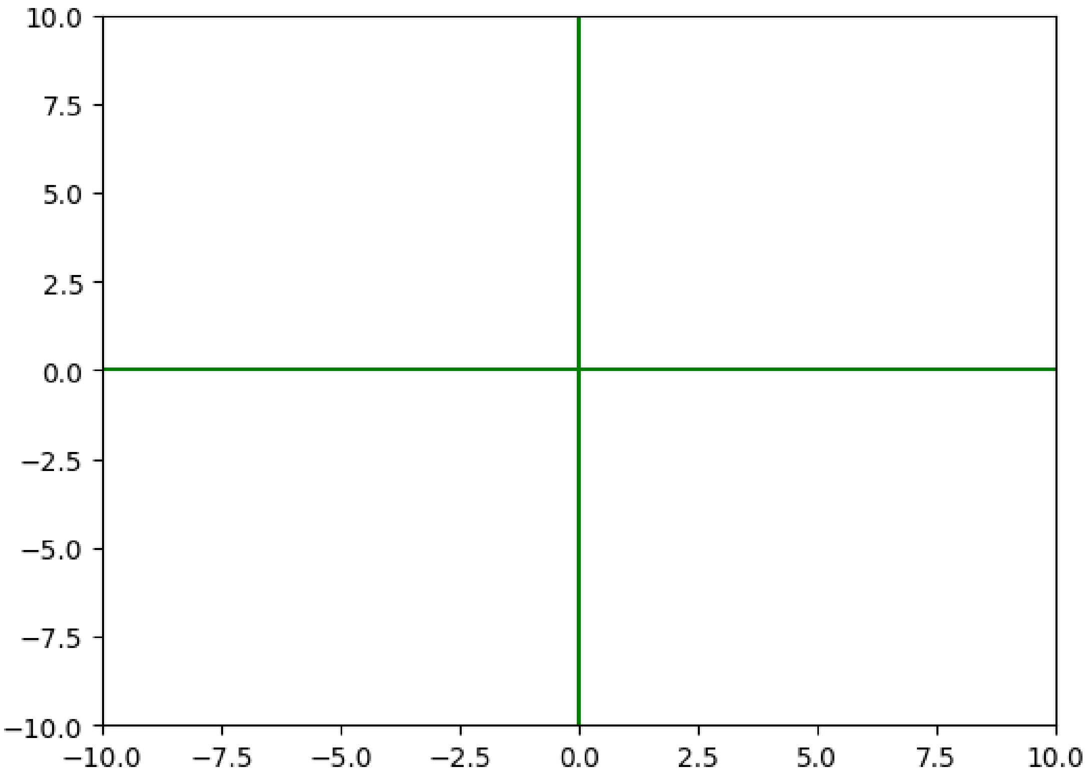

```
Code test passed
Go on to the next step
```

6 Step 4 - Displaying Axis Lines

Notice the code to `plot` a line for the x axis and a line for the y axis. The `'b'` makes the line blue. Run the code, then change each 'b' to 'g' to make the lines green.

[5]:
```
import matplotlib.pyplot as plt

xmin = -10
xmax = 10
ymin = -10
ymax = 10

fig, ax = plt.subplots()
plt.axis([xmin,xmax,ymin,ymax]) # window size
plt.plot([xmin,xmax],[0,0],'g') # blue x axis
```

```
plt.plot([0,0],[ymin,ymax], 'g') # blue y axis

plt.show()

# Only change code above this line
import math_code_test_b as test
test.step04(In[-1].split('# Only change code above this line')[0])
```



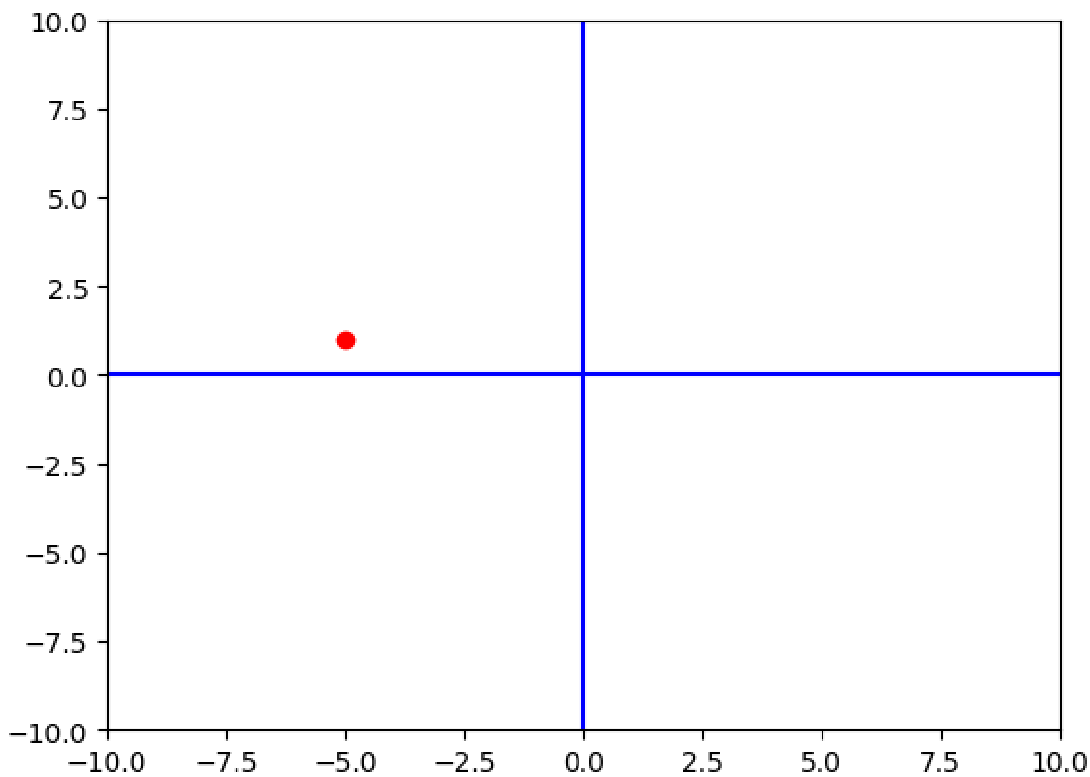

```
Code test passed
Go on to the next step
```

7 Step 5 - Plotting a Point

Now you will plot a point on the graph. Notice the `'ro'` makes the point a red dot. Run the code, then change the location of the point to (-5,1) and run it again. Keep the window size the same. Notice the difference between plotting a point and plotting a line.

[6]:
```
import matplotlib.pyplot as plt

xmin = -10
xmax = 10
```

```
ymin = -10
ymax = 10

fig, ax = plt.subplots()
plt.axis([xmin,xmax,ymin,ymax]) # window size
plt.plot([xmin,xmax],[0,0],'b') # blue x axis
plt.plot([0,0],[ymin,ymax], 'b') # blue y axis

# Change only the numbers in the following line:
plt.plot([-5],[1], 'ro')

plt.show()


# Only change code above this line
import math_code_test_b as test
test.step05(In[-1].split('# Only change code above this line')[0])
```

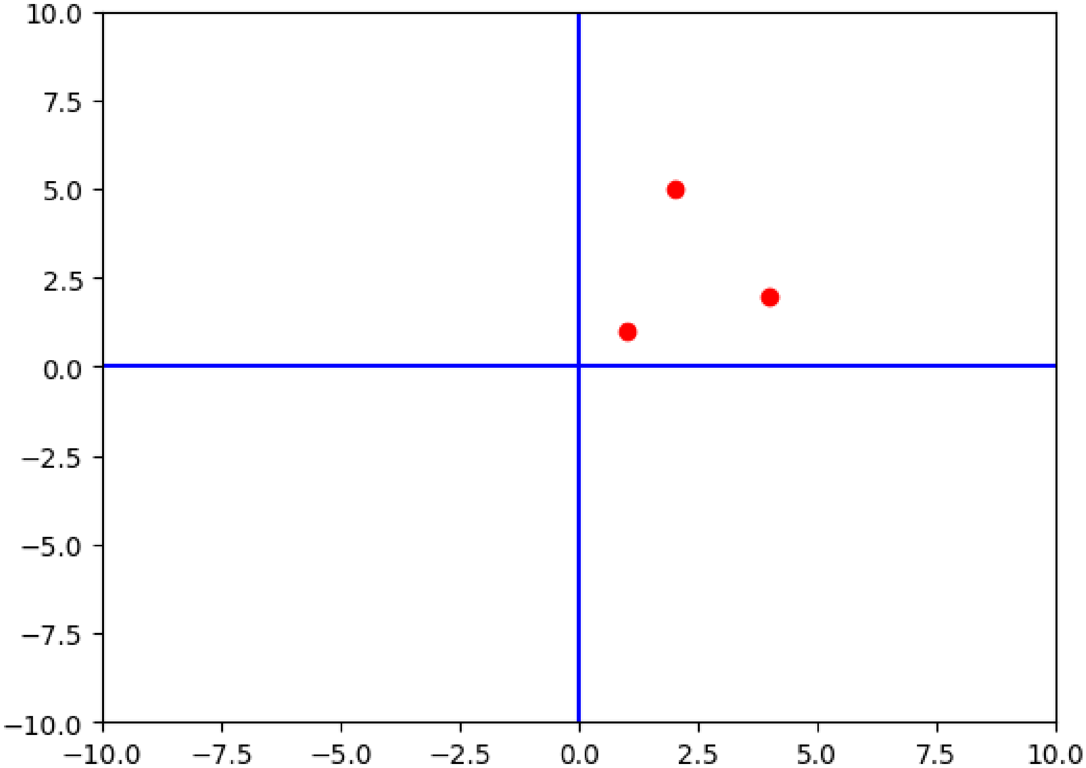



```
Code test passed
Go on to the next step
```

8 Step 6 - Plotting Several Points

You have actually been using arrays to plot each singular point so far. In this step, you will see an array of x values and an array of y values defined before the plot statement. Notice that these two short arrays create one point: (4,2). Add two numbers to each array so that it also plots points (1,1) and (2,5).

[7]:
```
import matplotlib.pyplot as plt

# only change the next two lines:
x = [4, 1, 2]
y = [2, 1, 5]

# Only change code above this line

xmin = -10
xmax = 10
ymin = -10
ymax = 10

fig, ax = plt.subplots()
plt.axis([xmin,xmax,ymin,ymax]) # window size
plt.plot([xmin,xmax],[0,0],'b') # blue x axis
plt.plot([0,0],[ymin,ymax],'b') # blue y axis

plt.plot(x, y, 'ro') # red points
plt.show()


# Only change code above this line
import math_code_test_b as test
test.step06(In[-1].split('# Only change code above this line')[0])
```



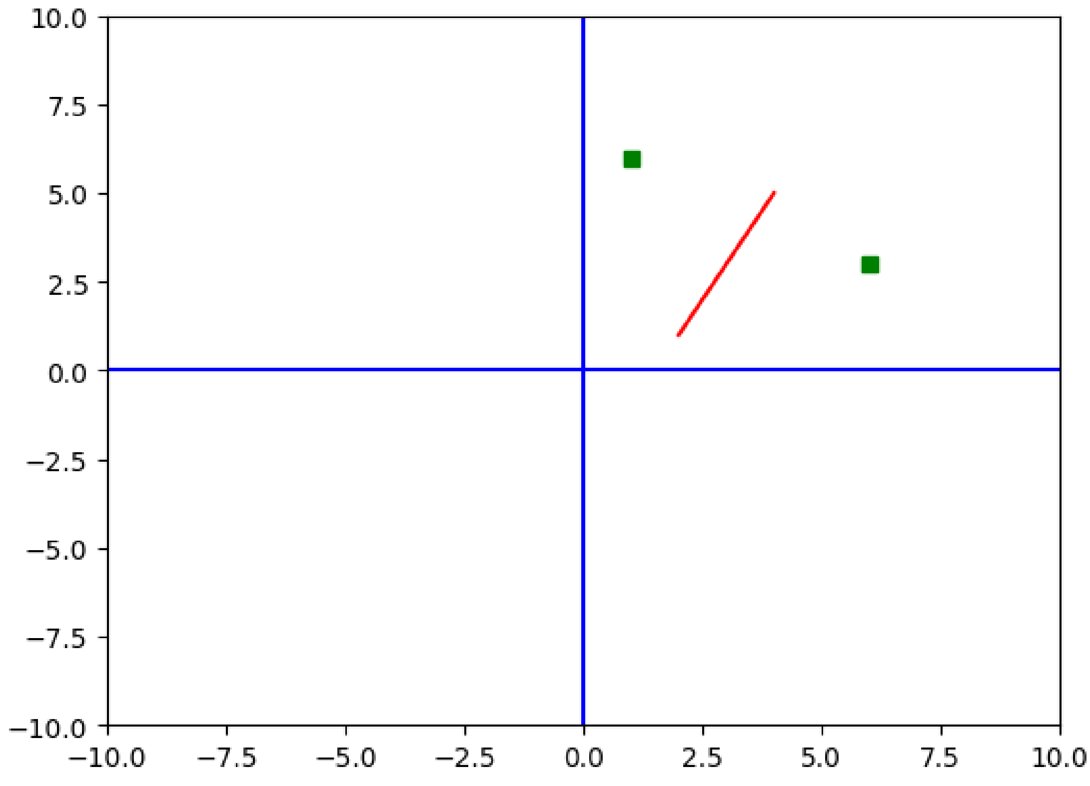

```
Code test passed
Go on to the next step
```

9 Step 7 - Plotting Points and Lines

Notice the subtle difference between plotting points and lines. Each `plot()` statement takes an array of x values, an array of y values, and a third argument to tell what you are plotting. The default plot is a line. The letters `'r'` and `'b'` (and `'g'` and a few others) indicate common colors. The "o" in `'ro'` indicates a dot, where `'rs'` would indicate a red square and `'r^'` would indicate a red triangle. Plot a red line and two green squares.

[8]:
```
import matplotlib.pyplot as plt

# Use these numbers:
linex = [2,4]
liney = [1,5]
pointx = [1,6]
pointy = [6,3]

# Keep these lines:
xmin = -10
```

```
xmax = 10
ymin = -10
ymax = 10

fig, ax = plt.subplots()
plt.axis([xmin,xmax,ymin,ymax]) # window size
plt.plot([xmin,xmax],[0,0],'b') # blue x axis
plt.plot([0,0],[ymin,ymax], 'b') # blue y axis

# Change the next two lines:
plt.plot(linex, liney, 'r')
plt.plot(pointx, pointy, 'gs')

plt.show()


# Only change code above this line
import math_code_test_b as test
test.step07(In[-1].split('# Only change code above this line')[0])
```



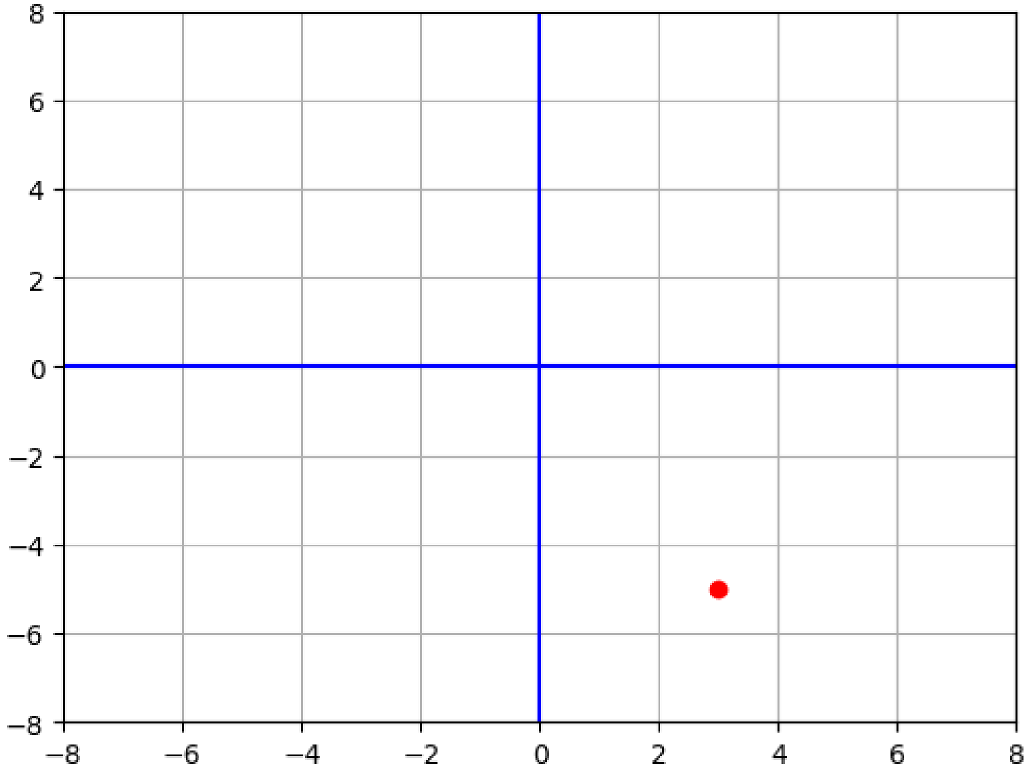

```
Code test passed
```

Go on to the next step

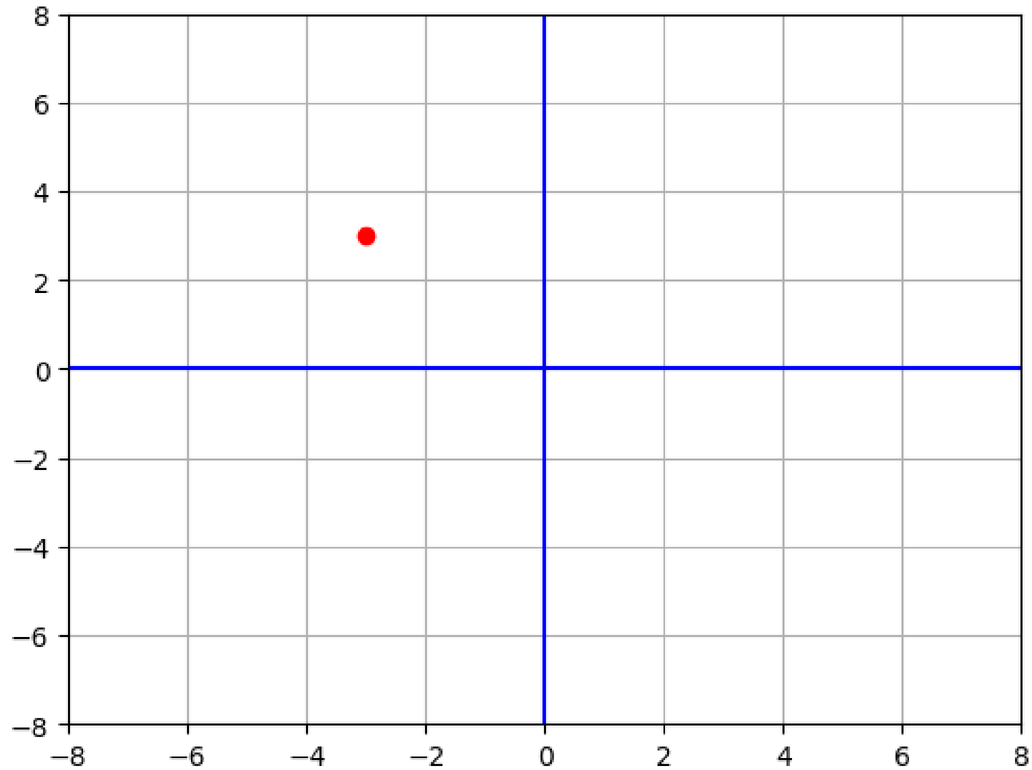
10 Step 8 - Making a Scatterplot Game

To make the game, you can make a loop that plots a random point and asks the user to input the (x,y) coordinates. Notice the `for` loop that runs three rounds of the game. Run the code, play the game, then you can go on to the next step.

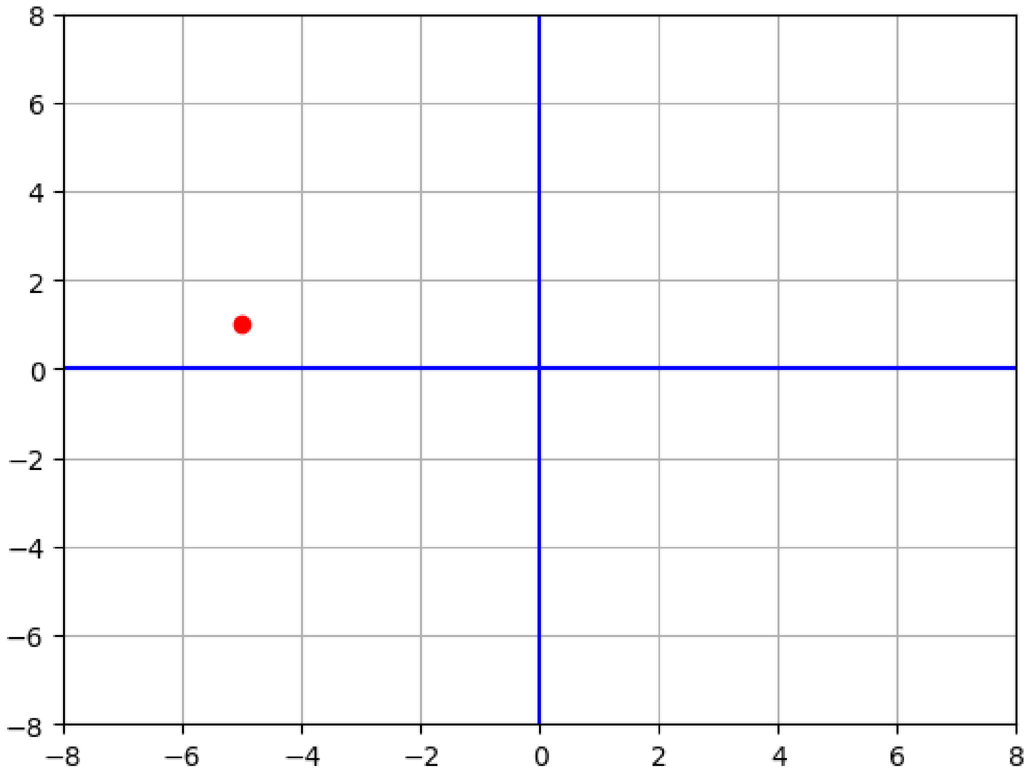
[9]:
```python
import matplotlib.pyplot as plt
import random

score = 0

xmin = -8
xmax = 8
ymin = -8
ymax = 8

fig, ax = plt.subplots()

for i in range(0,3):
    xpoint = random.randint(xmin, xmax)
    ypoint = random.randint(ymin, ymax)
    x = [xpoint]
    y = [ypoint]
    plt.axis([xmin,xmax,ymin,ymax]) # window size
    plt.plot([xmin,xmax],[0,0],'b') # blue x axis
    plt.plot([0,0],[ymin,ymax], 'b') # blue y axis
    plt.plot(x, y, 'ro')
    print(" ")
    plt.grid() # displays grid lines on graph
    plt.show()
    guess = input("Enter the coordinates of the red point point: \n")
    guess_array = guess.split(",")
    xguess = int(guess_array[0])
    yguess = int(guess_array[1])
    if xguess == xpoint and yguess == ypoint:
        score = score + 1

print("Your score: ", score) # notice this is not in the loop


# Only change code above this line
import math_code_test_b as test
test.step08(score)
```




```
Enter the coordinates of the red point point:
3, -5
```




```
Enter the coordinates of the red point point:
-3, 3
```



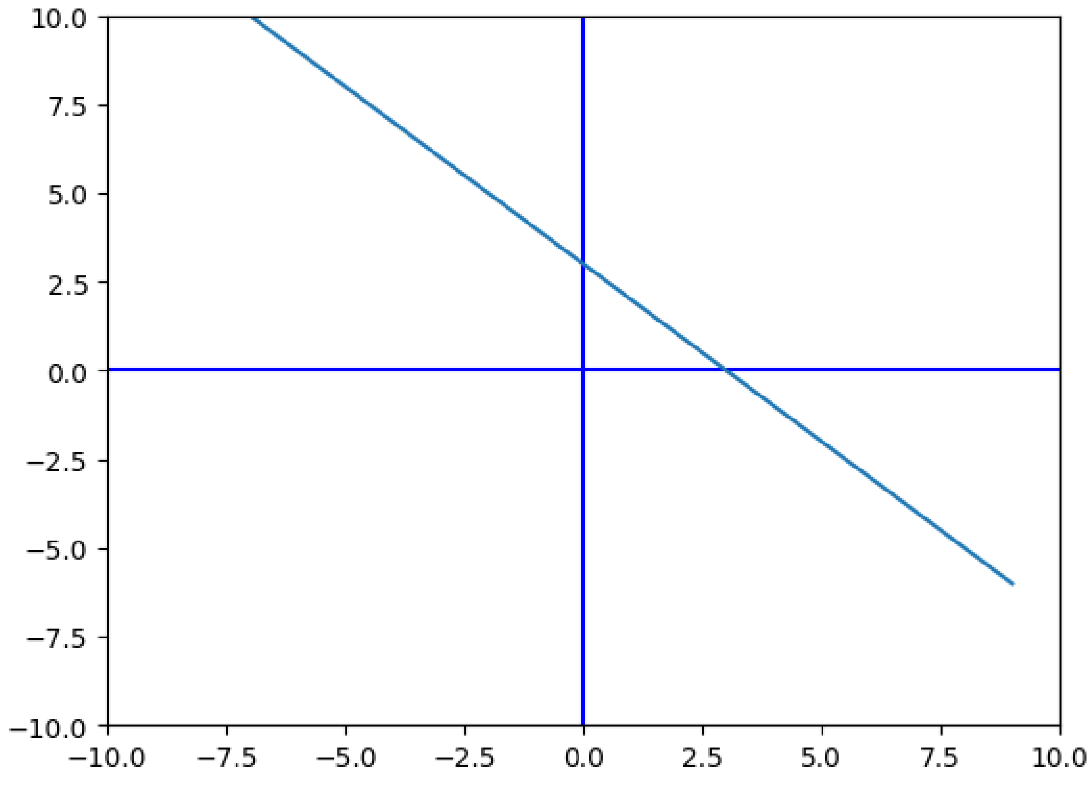

```
Enter the coordinates of the red point point:
-5, 1
Your score:  3
You scored 3 out of 3. Good job!
You can go on to the next step
```

11 Step 9 - Graphing Linear Equations

Besides graphing points, you can graph linear equations (or functions). The graph will be a straight line, and the equation will not have any exponents. For these graphs, you will import `numpy` and use the `linspace()` function to define the x values. That function takes three arguments: starting number, ending number, and number of steps. Notice the `plot()` function only has two arguments: the x values and a function (y = 2x - 3) for the y values. Run this code, then use the same x values to graph `y = -x + 3`.

[10]:
```
import matplotlib.pyplot as plt
import numpy as np

xmin = -10
xmax = 10
ymin = -10
ymax = 10
```

```
fig, ax = plt.subplots()
plt.axis([xmin,xmax,ymin,ymax]) # window size
plt.plot([xmin,xmax],[0,0],'b') # blue x axis
plt.plot([0,0],[ymin,ymax], 'b') # blue y axis

x = np.linspace(-9,9,36)

# Only change the next line to graph y = -x + 3
plt.plot(x, -x + 3)



plt.show()


# Only change code above this line
import math_code_test_b as test
test.step09(In[-1].split('# Only change code above this line')[0])
```




```
Code test passed
```

Go on to the next step

12 Step 10 - Creating Interactive Graphs

Like the previous graphs, you will graph a line. This time, you will create two sliders to change the slope and the y intecept. Notice the additional imports and other changes: You define a function with two arguments. All of the graphing happens within that `f(m,b)` function. The `interactive()` function calls your defined function and sets the boundaries for the sliders. Run this code then adjust the sliders and notice how they affect the graph.

[11]:
```
%matplotlib inline
from ipywidgets import interactive
import matplotlib.pyplot as plt
import numpy as np

# Define the graphing function
def f(m, b):
    xmin = -10
    xmax = 10
    ymin = -10
    ymax = 10
    plt.axis([xmin,xmax,ymin,ymax]) # window size
    plt.plot([xmin,xmax],[0,0],'black') # black x axis
    plt.plot([0,0],[ymin,ymax], 'black') # black y axis
    plt.title('y = mx + b')
    x = np.linspace(-10, 10, 1000)
    plt.plot(x, m*x+b)
    plt.show()

# Set up the sliders
interactive_plot = interactive(f, m=(-9, 9), b=(-9, 9))
interactive_plot


# Just run this code and use the sliders
import math_code_test_b as test
test.step01()
interactive_plot
```

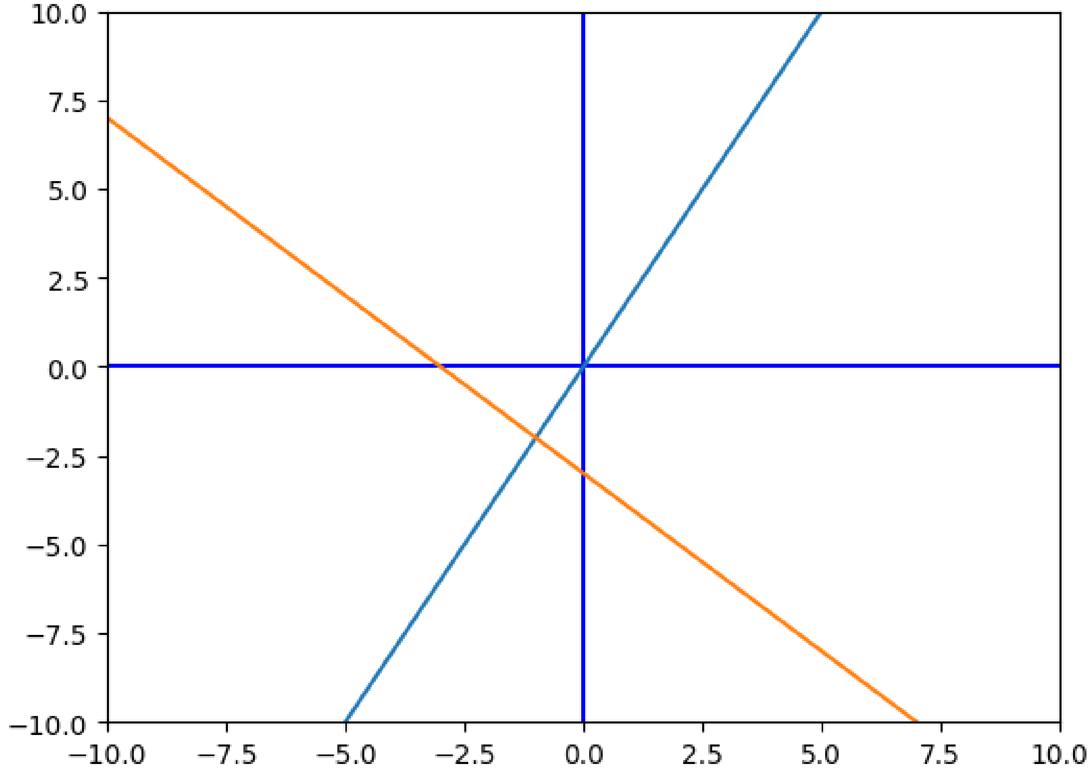
```
Code test Passed
Go on to the next step
```

```
interactive(children=(IntSlider(value=0, description='m', max=9, min=-9),␣
↪IntSlider(value=0, description='b', …
```

13 Step 11 - Graphing Systems

When you graph two equations on the same coordinate plane, they are a system of equations. Notice how the `points` variable defines the number of points and the `linspace()` function uses that variable. Run this code to see the graph, then change `y2` so that it graphs y = -x - 3.

[12]:
```
import matplotlib.pyplot as plt
import numpy as np



xmin = -10
xmax = 10
ymin = -10
ymax = 10
points = 2*(xmax-xmin)

# Define the x values once
x = np.linspace(xmin,xmax,points)

fig, ax = plt.subplots()
plt.axis([xmin,xmax,ymin,ymax]) # window size
plt.plot([xmin,xmax],[0,0],'b') # blue x axis
plt.plot([0,0],[ymin,ymax], 'b') # blue y axis

# line 1
y1 = 2*x
plt.plot(x, y1)

# line 2
#y2 = x**2 - 3
y2 = -x - 3
plt.plot(x, y2)

plt.show()


# Only change code above this line
import math_code_test_b as test
test.step11(In[-1].split('# Only change code above this line')[0])
```




```
Code test passed
Go on to the next step
```

14 Step 12 - Systems of Equations - Algebra

In a system of equations, the solution is the point where the two equations intersect, the (x,y) values that work in each equation. To work with algabraic expressions, you will import `sympy` and define x and y as symbols. If you have two equations and two variables, set each equation equal to zero. The `linsolve()` function takes the non-zero side of each equation and the variables used. Notice the syntax. Run the code, then change the two equations to solve 2x + y - 15 = 0 and 3x - y = 0.

[13]:
```
from sympy import *
x,y = symbols('x y')



# Change the equations in the following line:
#print(linsolve([2*x + y - 1, x - 2*y + 7], (x, y)))
print(linsolve([2*x + y - 15, 3*x - y], (x, y)))

# Only change code above this line
```

```
import math_code_test_b as test
test.step12(In[-1].split('# Only change code above this line')[0])
```

```
{(3, 9)}

Code test passed
Go on to the next step
```

15 Step 13 - Solutions as Coordinates

The `linsolve()` function returns a finite set, and you can convert that "finite set" into (x,y) coordinates. Notice how the code parses the `solution` variable into two separate variables. Just run the code to see how this works.

[14]:
```
from sympy import *
x,y = symbols('x y')

# Use variables for each equation
first = x + y
second = x - y

# parse finite set answer as coordinate pair
solution = linsolve([first, second], (x, y))
x_solution = solution.args[0][0]
y_solution = solution.args[0][1]

print("x = ", x_solution)
print("y = ", y_solution)
print(" ")
print("Solution: (",x_solution,",",y_solution,")")


# Just run this code
import math_code_test_b as test
test.step01()
```

```
x =  0
y =  0

Solution: ( 0 , 0 )
Code test Passed
Go on to the next step
```

16 Step 14 - Systems from User Input

For more flexibility, you can get each equation as user input (instead of defining them in the code). Run this code and try it out - to solve any system of two equations.

[15]:
```
from sympy import *

x,y = symbols('x y')
print("Remember to use Python syntax with x and y as variables")
print("Notice how each equation is already set equal to zero")
first = input("Enter the first equation: 0 = ")
second = input("Enter the second equation: 0 = ")
solution = linsolve([first, second], (x, y))
x_solution = solution.args[0][0]
y_solution = solution.args[0][1]

print("x = ", x_solution)
print("y = ", y_solution)


# Just run this code and test it with different equations
import math_code_test_b as test
test.step14()
```

```
Remember to use Python syntax with x and y as variables
Notice how each equation is already set equal to zero
Enter the first equation: 0 = 2*x - y
Enter the second equation: 0 = x + y + 1
x =  -1/3
y =  -2/3

If you didn't get a syntax error, code test passed
```

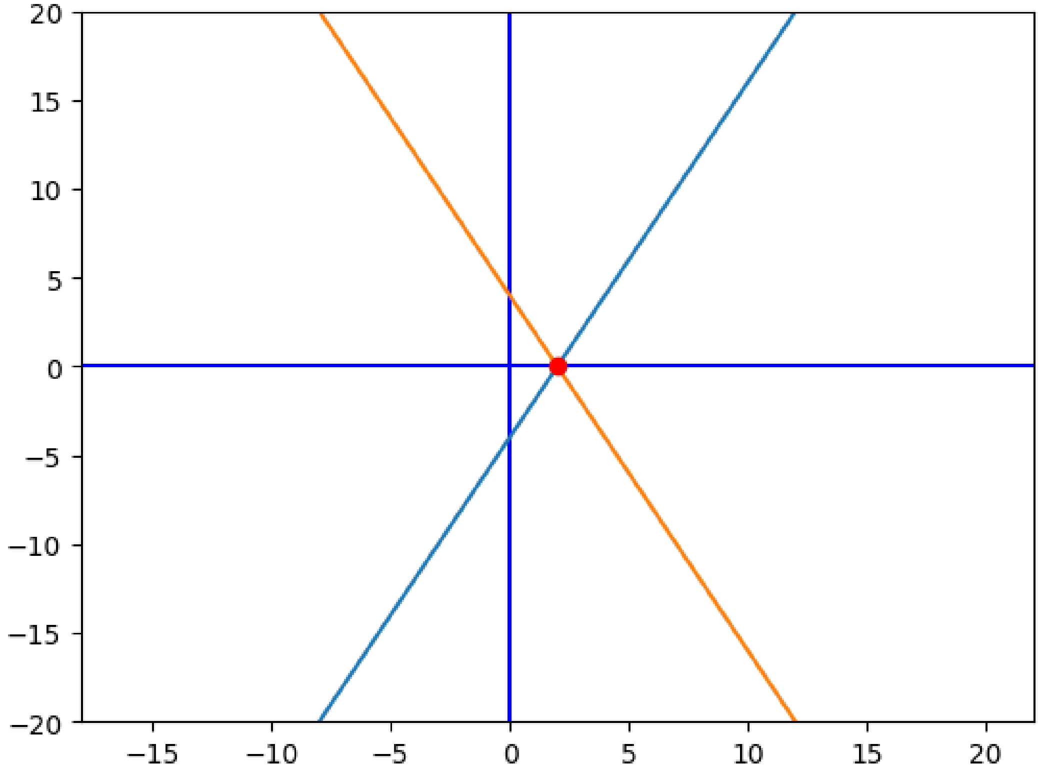
17 Step 15 - Solve and graph a system

Now you can put it all together: solve a system of equations, graph the system, and plot a point where the two lines intersect. Notice how this code is not like the previous solving equations code or the graphing code or the user input code. Python uses `sympy` to get the (x,y) solution and `numpy` to get the values to graph, so the user inputs nummerical values and the code uses them in two different ways. Think about how you would do this if the user input values for ax + by = c.

[16]:
```
from sympy import *
import matplotlib.pyplot as plt
import numpy as np

print("First equation: y = mx + b")
mb_1 = input("Enter m and b, separated by a comma: ")
mb_in1 = mb_1.split(",")
m1 = float(mb_in1[0])
b1 = float(mb_in1[1])

print("Second equation: y = mx + b")
```

```python
mb_2 = input("Enter m and b, separated by a comma: ")
mb_in2 = mb_2.split(",")
m2 = float(mb_in2[0])
b2 = float(mb_in2[1])

# Solve the system of equations
x,y = symbols('x y')
first = m1*x + b1 - y
second = m2*x + b2 - y
solution = linsolve([first, second], (x, y))
x_solution = round(float(solution.args[0][0]),3)
y_solution = round(float(solution.args[0][1]),3)

# Make sure the window includes the solution
xmin = int(x_solution) - 20
xmax = int(x_solution) + 20
ymin = int(y_solution) - 20
ymax = int(y_solution) + 20
points = 2*(xmax-xmin)

# Define the x values once for the graph
graph_x = np.linspace(xmin,xmax,points)

# Define the y values for the graph
y1 = m1*graph_x + b1
y2 = m2*graph_x + b2

fig, ax = plt.subplots()
plt.axis([xmin,xmax,ymin,ymax]) # window size
plt.plot([xmin,xmax],[0,0],'b') # blue x axis
plt.plot([0,0],[ymin,ymax], 'b') # blue y axis2

# line 1
plt.plot(graph_x, y1)

# line 2
plt.plot(graph_x, y2)

# point
plt.plot([x_solution],[y_solution],'ro')

plt.show()
print(" ")
print("Solution: (", x_solution, ",", y_solution, ")")


# Run this code and test it with different equations
```

```
First equation: y = mx + b
Enter m and b, separated by a comma: 2, -4
Second equation: y = mx + b
Enter m and b, separated by a comma: -2, 4
```



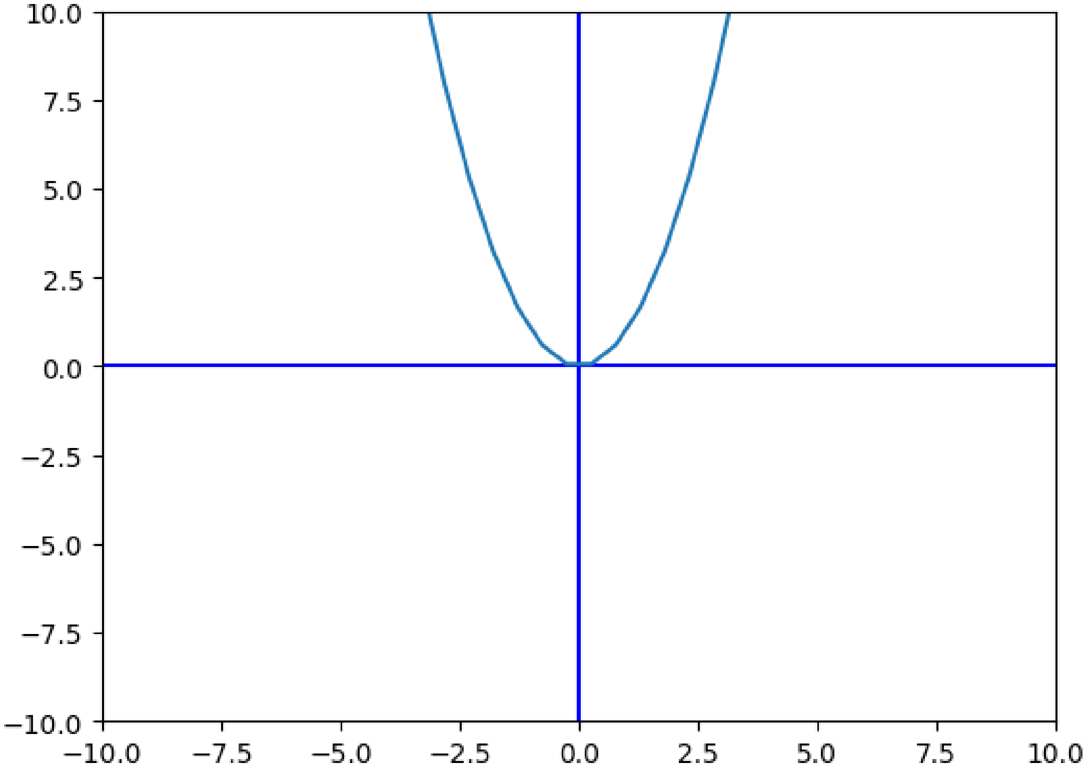

```
Solution: ( 2.0 , 0.0 )
```

18 Step 16 - Quadratic Functions

Any function that involves x2 is a “quadratic” function because “x squared” could be the area of a square. The graph is a parabola. The formula is y = ax2 + bx + c, where `b` and `c` can be zero but `a` has to be a number. Here is a graph of the simplest parabola.

[17]:
```
import matplotlib.pyplot as plt
import numpy as np

xmin = -10
xmax = 10
ymin = -10
ymax = 10
points = 2*(xmax-xmin)
```

```
x = np.linspace(xmin,xmax,points)

fig, ax = plt.subplots()
plt.axis([xmin,xmax,ymin,ymax]) # window size
plt.plot([xmin,xmax],[0,0],'b') # blue x axis
plt.plot([0,0],[ymin,ymax], 'b') # blue y axis

y = x**2

plt.plot(x,y)
plt.show()

# Just run this code. The next step will transform the graph
import math_code_test_b as test
test.step01()
```




```
Code test Passed
Go on to the next step
```

19 Step 17 - Quadratic Function ABC's

Using the parabola formula y = ax2 + bx + c, you will change the values of `a`, `b`, and `c` to see how they affect the graph. Run the code and use the sliders to change the values of `a` and `b`. Then change the code in the three places indicated to add a slider for `c`. You may remember this type of interactive graph from an earlier step. Move each slider to see how it affects the graph.

[18]:
```
%matplotlib inline
from ipywidgets import interactive
import matplotlib.pyplot as plt
import numpy as np

# Change the next line to include c:
def f(a,b,c):
    plt.axis([-10,10,-10,10]) # window size
    plt.plot([-10,10],[0,0],'k') # blue x axis
    plt.plot([0,0],[-10,10], 'k') # blue y axis
    x = np.linspace(-10, 10, 1000)

    # Change the next line to add c to the end of the function:
    plt.plot(x, a*x**2 + b*x + c)
    plt.show()

# Change the next line to add a slider to change the c value
interactive_plot = interactive(f, a=(-9, 9), b=(-9,9), c=(-9,9))
interactive_plot


# Run the code once, then change the code and run it again

# Only change code above this line
import math_code_test_b as test
test.step17(In[-1].split('# Only change code above this line')[0])
interactive_plot
```

```
Code test passed
Go on to the next step
```

```
interactive(children=(IntSlider(value=0, description='a', max=9, min=-9),␣
↪IntSlider(value=0, description='b', …
```

20 Step 18 - Quadratic Functions - Vertex

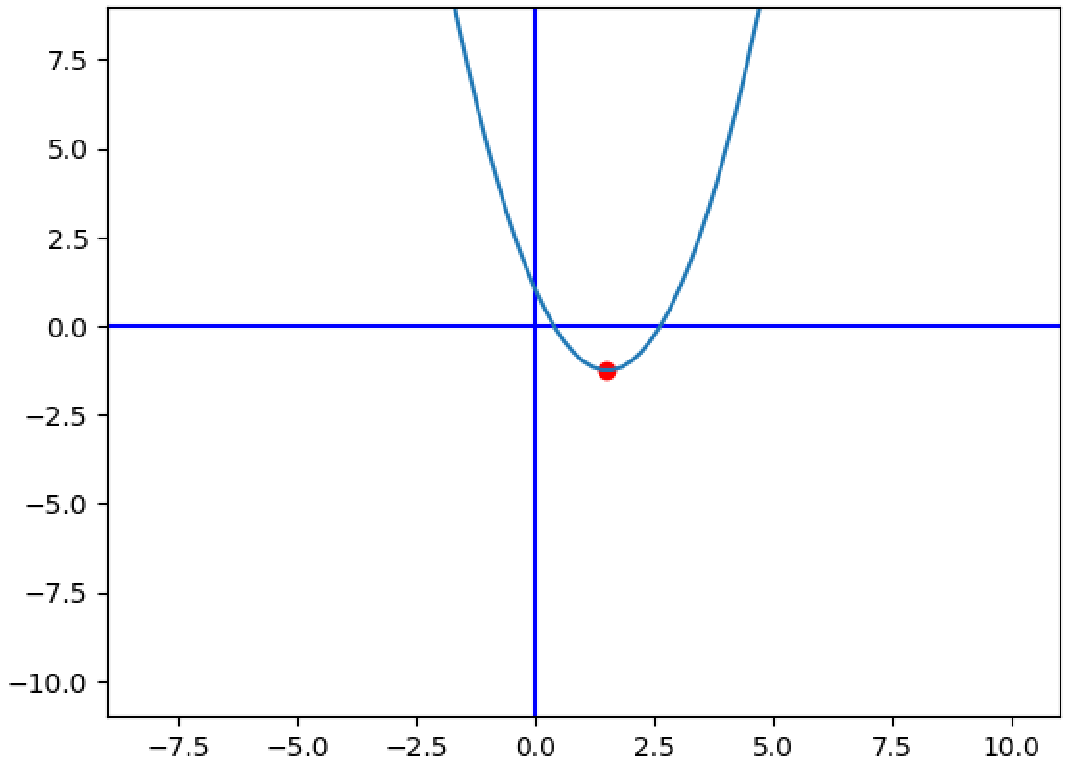
The vertex is the point where the parabola turns around. The x value of the vertex is $\frac{-b}{2a}$ (and then you would calculate the y value to get the point). Write the code to find the vertex, given `a`, `b`, and `c` as inputs. Remember the parabola forumula is y = ax2 + bx + c

[19]:

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

# \u00b2 prints 2 as an exponent
print("y = ax\u00b2 + bx + c")

a = float(input("a = "))
b = float(input("b = "))
c = float(input("c = "))

# Write your code here, changing vx and vy
vx = -b/(2*a)                  # "vertex formula"
#vy = -(b**2 - 4*a*c)/(4*a)    # "vertex formula"
vy = a*vx**2 + b*vx + c        # "vertex formula"

# Only change the code above this line

print(" (", vx, " , ", vy, ")")
print(" ")

xmin = int(vx)-10
xmax = int(vx)+10
ymin = int(vy)-10
ymax = int(vy)+10
points = 2*(xmax-xmin)
x = np.linspace(xmin,xmax,points)

fig, ax = plt.subplots()
plt.axis([xmin,xmax,ymin,ymax]) # window size
plt.plot([xmin,xmax],[0,0],'b') # blue x axis
plt.plot([0,0],[ymin,ymax], 'b') # blue y axis

plt.plot([vx],[vy],'ro') # vertex

x = np.linspace(vx-10,vx+10,100)
y = a*x**2 + b*x + c
plt.plot(x,y)

plt.show()


# Only change code above this line
import math_code_test_b as test
test.step18(In[-1].split('# Only change code above this line')[0])
```

```
y = ax² + bx + c
a = 1
```

```
b = -3
c = 1
 ( 1.5  ,  -1.25 )
```




```
Code test passed
Go on to the next step
```

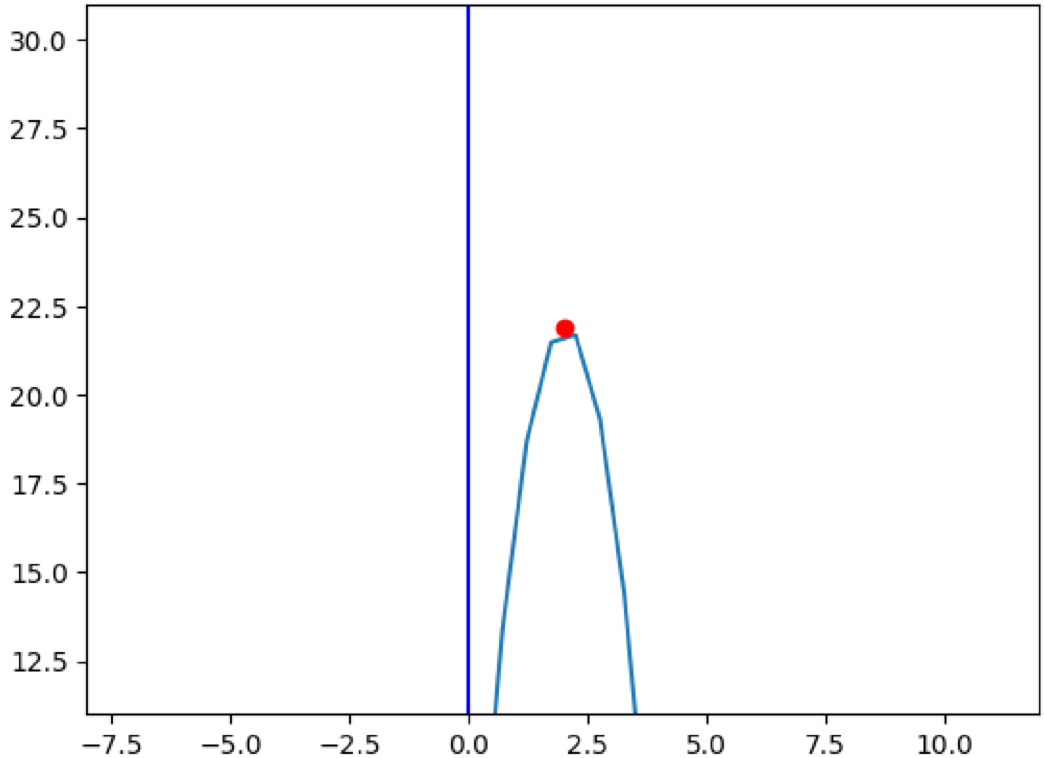
21 Step 19 - Projectile Motion

The path of every projectile is a parabola. For something thrown or launched upward, the `a` value is -4.9 (meters per second squared); the `b` value is the initial velocity (in meters per second); the `c` value is the initial height (in meters); the `x` value is time (in seconds); and the `y` value is the height at that time. In this code, change `vx` and `vy` to represent the vertex. Plotting that (x,y) vertex point is already in the code.

[20]:
```
import matplotlib.pyplot as plt
import numpy as np

a = -4.9
b = float(input("Initial velocity = "))
```

```
c = float(input("Initial height = "))

# Change vx and vy to represent the vertex
vx = -b/(2*a)
vy = a*vx**2 + b*vx + c

# Also change the following dimensions to display the vertex
xmin = int(vx)-10
xmax = int(vx)+10
ymin = int(vy)-10
ymax = int(vy)+10

# You do not need to change anything below this line
points = 2*(xmax-xmin)
x = np.linspace(xmin,xmax,points)
y = a*x**2 + b*x + c

fig, ax = plt.subplots()
plt.axis([xmin,xmax,ymin,ymax]) # window size
plt.plot([xmin,xmax],[0,0],'b') # blue x axis
plt.plot([0,0],[ymin,ymax], 'b') # blue y axis

plt.plot(x,y) # plot the line for the equation
plt.plot([vx],[vy],'ro') # plot the vertex point

print(" (", vx, ",", vy, ")")
print(" ")
plt.show()


# Only change code above this line
import math_code_test_b as test
test.step19(In[-1].split('# Only change code above this line')[0])
```

```
Initial velocity = 20
Initial height = 1.5
 ( 2.0408163265306123 , 21.90816326530612 )
```




```
Code test passed
Go on to the next step
```

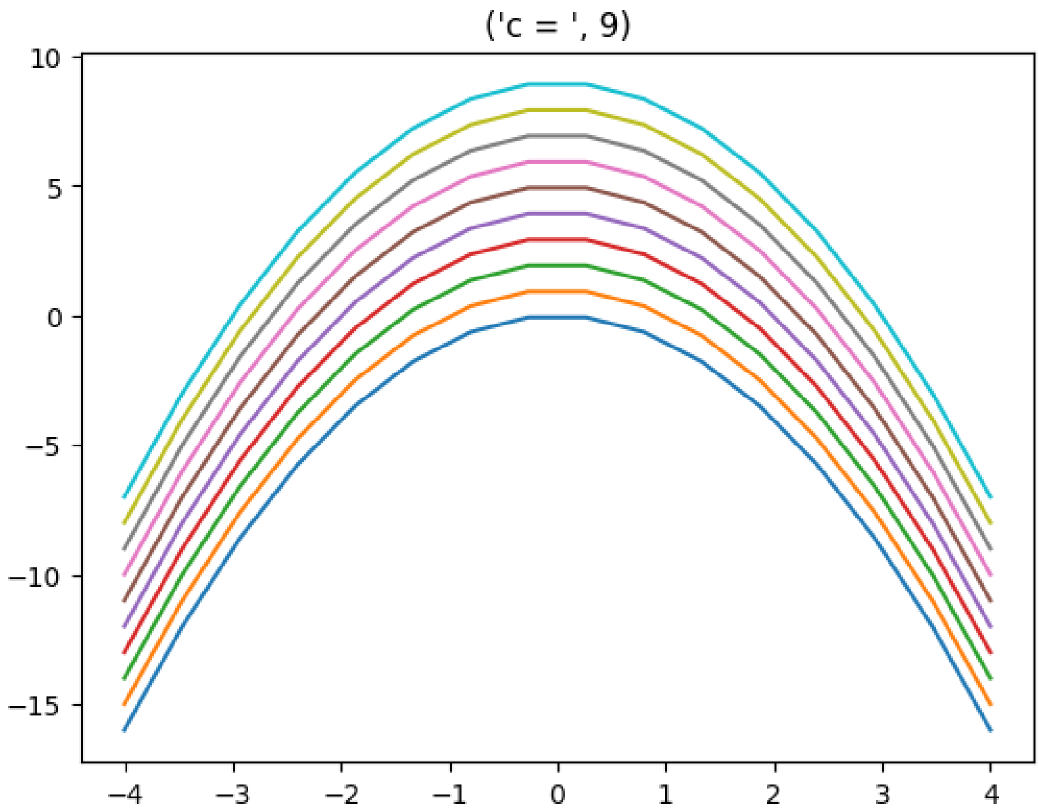
22 Step 20 - Quadratic Functions - C

Like many other functions, the `c` value (also called the "constant" because it is not a variable) affects the vertical shift of the graph. Run the following code to see how changing the `c` value changes the graph.

[21]:
```
import matplotlib.pyplot as plt
import numpy as np
import time
from IPython import display

x = np.linspace(-4,4,16)
fig, ax = plt.subplots()
cvalue = "c = "

for c in range(10):
    y = -x**2+c
    plt.plot(x,y)
```

```
    cvalue = "c = ", c
    ax.set_title(cvalue)
    display.display(plt.gcf())
    time.sleep(0.5)
    display.clear_output(wait=True)

# Just run this code
import math_code_test_b as test
test.step01()
```

```
Code test Passed
Go on to the next step
```



23 Step 21 - The Quadratic Formula

For a projectile, you also need to find the point when it hits the ground. On a graph, you would call these points the "roots" or "x intercepts" or "zeros" (because y = 0 at these points). The quadratic formula gives you the x value when y = 0. Given `a`,`b` and `c`, here is the quadratic formula: $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$ Notice it's the vertex plus or minus something: $\frac{-b}{2a} + \frac{\sqrt{b^2-4ac}}{2a}$ and $\frac{-b}{2a} - \frac{\sqrt{b^2-4ac}}{2a}$ Write the code to output two x values, given a, b, and c as input. Use `math.sqrt()` for the square root.

[22]:
```
import math

# \u00b2 prints 2 as an exponent
print("0 = ax\u00b2 + bx + c")
a = float(input("a = "))
b = float(input("b = "))
c = float(input("c = "))
x1 = 0
x2 = 0

# Check for non-real answers:
if b**2-4*a*c < 0:
    print("No real roots")
else:
    # Write your code here, changing x1 and x2
    x1 = (-b + math.sqrt(b**2 - 4*a*c))/(2*a)
    x2 = (-b - math.sqrt(b**2 - 4*a*c))/(2*a)
    print("The roots are ", x1, " and ", x2)


# Only change code above this line
import math_code_test_b as test
test.step21(In[-1].split('# Only change code above this line')[0])
```

```
0 = ax² + bx + c
a = 1
b = -5
c = 4
The roots are  4.0  and  1.0

Code test passed
Go on to the next step
```

24 Step 22 - Table of Values

In addition to graphing a function, you may need a table of values. This code shows how to make a simple table of (x,y) values. Run the code, then change the title to "y = 3x + 2" and change the function in the table.

[23]:
```
import numpy as np
import matplotlib.pyplot as plt

ax = plt.subplot()
ax.set_axis_off()
title = "y = 3x + 2"  # Change this title
cols = ('x', 'y')
rows = [[0,0]]
```

```
for a in range(1,10):
    rows.append([a, 3*a+2]) # Change only the function in this line

ax.set_title(title)
plt.table(cellText=rows, colLabels=cols, cellLoc='center', loc='upper left')
plt.show()


# Only change code above this line
import math_code_test_b as test
test.step22(In[-1].split('# Only change code above this line')[0])
```

y = 3x + 2

x	y
0	0
1	5
2	8
3	11
4	14
5	17
6	20
7	23
8	26
9	29

```
Code test passed
Go on to the next step
```

25 Step 23 - Projectile Game

Learn quadratic functions by building a projectile game. Starting at (0,0) you launch a toy rocket that must clear a wall. You can randomize the height and location of the wall. The goal is to determine what initial velocity would get the rocket over the wall. Bonus: make an animation of the path of the rocket.

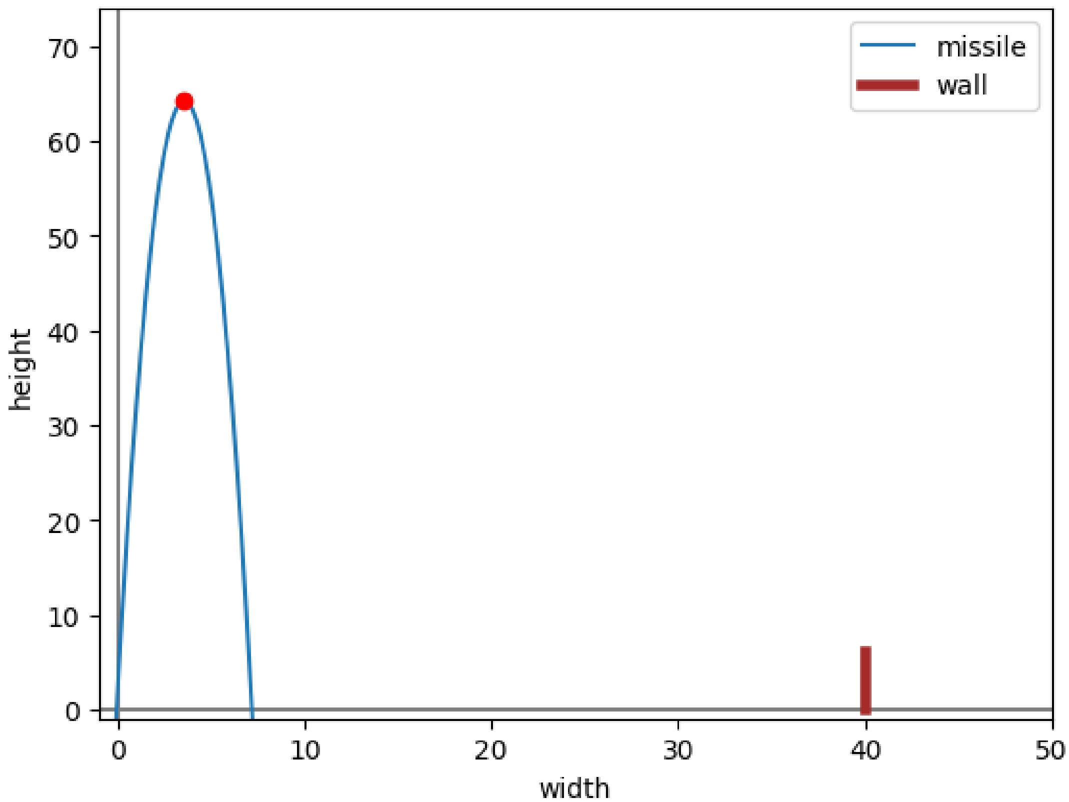
[24]:

```python
import matplotlib.pyplot as plt
import numpy as np
import math
import random

# variable to define the permitted number of attempts
attempts = 3
# variable to count the number of guesses
guesses = 0
# variable to register the user's success
successful = False

# define coordinates for the target: the wall that has to be cleared
loc = random.randint(0, 100)
height = random.randint(0, 50)
print(f"Your toy rocket has to clear a wall at a distance of {loc} meters.")

fig, ax = plt.subplots()
a = -4.9
for i in range(0, attempts):
    # ask for and validate the user's input regarding the missile's velocity␣
    ↪and its starting height
    running = True
    while running:
        try:
            b = float(input("- define an initial velocity (in m/s): "))
            c = float(input("- define an initial height (in m): "))
            running = False
        except:
            print("# ERROR: At least one of your inputs was not valid. - Try it␣
    ↪again ...")

    # calculate the coordinates of the vertex of the missile's trajectory
    vx = -b/(2*a)
    vy = a*vx**2 + b*vx + c
    print(f"--> trajectory's vertex: ({vx:g}, {vy:g})")
    print(" ")

    # define dimensions to display the missile's trajectory and the wall
    xmin = -1
    if vx < (0.5*loc):
        xmax = loc+10
    else:
        xmax = int(vx)+10
    ymin = -1
    ymax = int(vy)+10
```

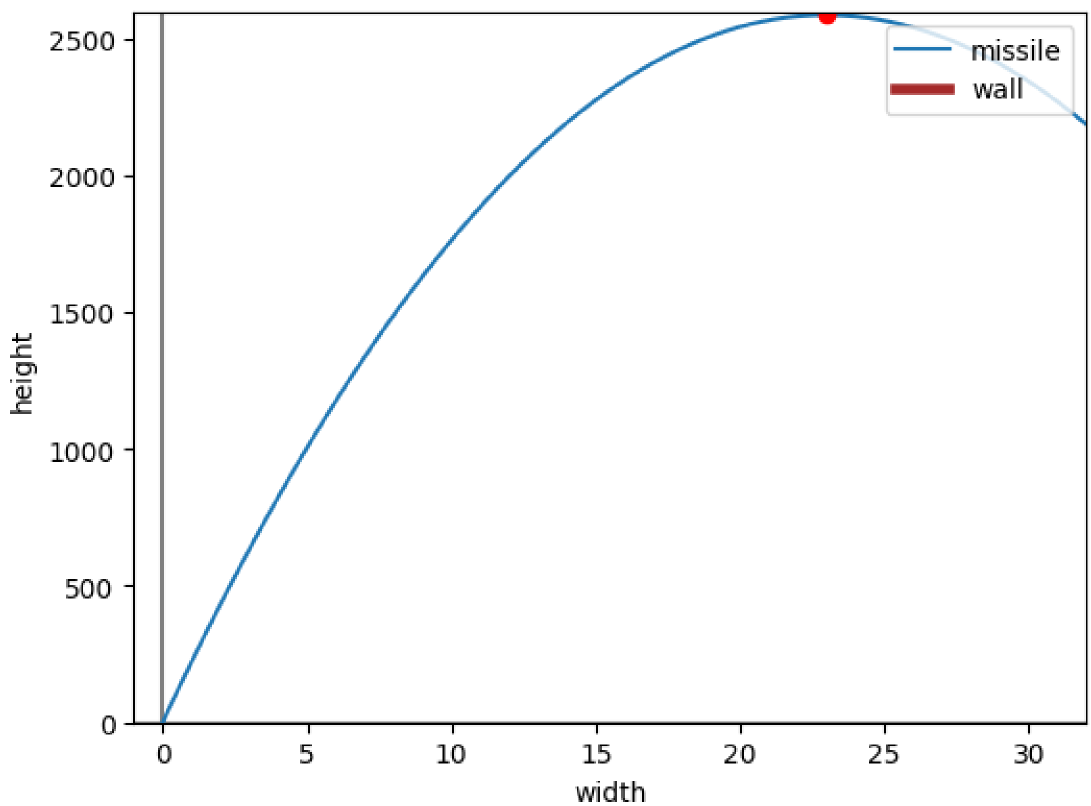
```
    # plot the coordinate system with the missile's trajectory and the wall
    points = 8*(xmax-xmin)
    x = np.linspace(xmin,xmax,points)
    y = a*x**2 + b*x + c
    plt.axis([xmin,xmax,ymin,ymax])           # window dimensions
    plt.plot([xmin,xmax],[0,0],'grey')        # x axis
    plt.plot([0,0],[ymin,ymax], 'grey')       # y axis
    plt.plot(x,y, label="missile")            # missile's trajectory
    plt.plot([vx],[vy],'ro')                  # trajectory's vertex point
    wall = plt.plot([loc, loc], [0, height], "brown", label="wall")   # wall␣
↪(target)
    plt.setp(wall, linewidth=4)
    # tx = loc + 1
    # ty = height
    # ax.text(tx, ty, "wall", style="italic")
    plt.ylabel("height")
    plt.xlabel("width")
    plt.legend(loc="upper right")
    plt.show(block=False)
    plt.pause(0.01)

    # evaluate the trajectory based on the user's input
    guesses += 1
    user_trajectory = round((-b - math.sqrt(b**2 - 4*a*c)) /(2*a), 2)
    if user_trajectory >= loc:
        print(f"--> Distance: {user_trajectory}m --> SUCCESS!")
        successful = True
        break
    else:
        distance = loc - user_trajectory
        if guesses < attempts:
            print(f"--> SORRY, you've missed the target (by {distance:g}m) ...␣
↪please try it again!")
        else:
            print(f"--> SORRY, you've missed the target again! (by {distance:
↪g}m this time)")
        print()

# provide a final feedback
print()
print("*** E V A L U A T I O N ***")
print("===========================")
if successful:
    print(f"CONGRATULATIONS! - You have cleared the wall in {guesses} attempt/s!
↪")
else:
```

```
    print(f"GAME OVER! - Unfortunately, you have not cleared the wall in␣
↪{attempts} attempts!")


# This step does not have a test
```

```
Your toy rocket has to clear a wall at a distance of 40 meters.
- define an initial velocity (in m/s): 35
- define an initial height (in m): 1.8
--> trajectory's vertex: (3.57143, 64.3)
```




```
--> SORRY, you've missed the target (by 32.81m) … please try it again!

- define an initial velocity (in m/s): 225
- define an initial height (in m): 1.8
--> trajectory's vertex: (22.9592, 2584.71)
```




```
--> Distance: 45.93m --> SUCCESS!

*** E V A L U A T I O N ***
===========================
CONGRATULATIONS! - You have cleared the wall in 2 attempt/s!
```

26 Step 24 - Define Graphing Functions

Building on what you have already done, create a menu with the following options:

Display the graph and a table of values for any “y=” equation input

Solve a system of two equations without graphing

Graph two equations and plot the point of intersection

Given a, b and c in a quadratic equation, plot the roots and vertex

Then think about how you will define a function for each item.

[25]:
```
# Write your code here
# MENU
print("*** SOLVING & GRAPHING FUNCTIONS ***")
print("=====================================")
```

```
print("You can select between one of the following options:")
print("--> 1 = display the graph and a table of values for any 'y=' equation␣
↪input")
print("--> 2 = solve a system of two equations, without graphing")
print("--> 3 = graph two equations and plot the point of intersection")
print("--> 4 = given a, b and c in a quadratic equation, plot the roots and␣
↪vertex")
print()

user_choice = input("Please enter the number corresponding to the action of␣
↪your choice: ")

# This step does not have a test
```

```
*** SOLVING & GRAPHING FUNCTIONS ***
====================================
You can select between one of the following options:
--> 1 = display the graph and a table of values for any 'y=' equation input
--> 2 = solve a system of two equations, without graphing
--> 3 = graph two equations and plot the point of intersection
--> 4 = given a, b and c in a quadratic equation, plot the roots and vertex

Please enter the number corresponding to the action of your choice: 3
```

27 Step 25 - Certification Project 2

Build a graphing calculator that performs the functions mentioned in the previous step:

Display the graph and a table of values for any "y=" equation input

Solve a system of two equations without graphing

Graph two equations and plot the point of intersection

Given a, b and c in a quadratic equation, plot the roots and vertex

Define each of the functions, and make each option call a function.

[26]:
```
import matplotlib.pyplot as plt
import numpy as np
import math
from sympy import *

# CONSTANTS
reminder_python_syntax = "(--> Remember to use Python syntax with x and y␣
↪variables!)"
integer_note = "(For the sake of simplicity, the permitted entries of this␣
↪exercise are limited to integers!)"
```

```
error_msg = "# ERROR! At least one of your inputs must have been incorrect. -␣
↪Try it again ..."
xmin = -10
xmax = 10
ymin = -10
ymax = 10
points = 4*(xmax-xmin)


# FUNCTIONS ...

def display_graph_and_values():
    print("[ 1 ]")
    print("This function allows you to plot the graph for a simple linear␣
↪equation (y = mx + b) and\nto display a corresponding table of values.")
    print("-------------------------------")
    running = True
    while running:
        try:
            print(reminder_python_syntax)
            print("Based on the general form of a linear equation (y = mx + b),␣
↪you have to enter a slope (m) and an intercept (b).")
            print(integer_note)
            m = int(input(" - your slope: m = "))
            b = int(input(" - your intercept: b = "))
            x = np.linspace(xmin, xmax, points)
            y = m*x + b
            running = False
        except:
            print(error_msg)

    if m != 0:
        if m == 1:
            c = ""
        elif m == -1:
            c = "-"
        else:
            c = m
        if b > 0:
            equation = f"y = {c}x + {b}"
        elif b < 0:
            equation = f"y = {c}x - {abs(b)}"
        else:
            equation = f"y = {c}x"
    else:
        equation = f"y = {b}"
```

```
    # plot the graph
    fig, ax = plt.subplots()
    plt.axis([xmin, xmax, ymin, ymax])
    plt.plot([xmin, xmax], [0, 0], "lightgrey")
    plt.plot([0, 0], [ymin, ymax], "lightgrey")
    plt.plot(x, y)
    plt.title(f"graph for {equation}")
    plt.show()

    # plot a table of values
    ax_2 = plt.subplot()
    ax_2.set_axis_off()
    title = f"table of values for {equation}"
    cols = ('x', 'y')
    rows = []
    for n in range(-10, 11, 2):
        rows.append([n, m*n + b])
    ax_2.set_title(title)
    val_table = plt.table(cellText=rows, colLabels=cols, cellLoc='center',␣
 ↪loc='upper left')
    for cell in val_table._cells:
        text = val_table._cells[cell].get_text()
        text.set_fontstyle('italic')

    plt.show()


def solve_two_equations():
    print("[ 2 ]")
    print("This function allows you to solve a system of two linear equations␣
 ↪with two variables each.")
    x, y = symbols ("x y")
    running = True
    while running:
        try:
            print(reminder_python_syntax)
            equation_1 = input(" - enter your first equation:  0 = ")
            equation_2 = input(" - enter your second equation: 0 = ")
            a = Eq(parse_expr(equation_1), 0)
            b = Eq(parse_expr(equation_2), 0)
            a = simplify(a)
            b = simplify(b)
            if (a.lhs-a.rhs) == (b.lhs-b.rhs):
                print("--> Your system of equations is underdetermined, it has␣
 ↪an infinite number of solutions.")
                print("(One of the variables (x, y) is free, that is you can␣
 ↪assign an arbitrary value to it.)")
```

```
            else:
                solution = linsolve([equation_1, equation_2], (x, y))
                if solution:
                    x_solution = round(float(solution.args[0][0]),3)
                    y_solution = round(float(solution.args[0][1]),3)
                    print(f"--> Your system of equations has a unique solution:␣
↪x = {x_solution}, y = {y_solution}.")
                else:
                    print("--> Your system of equations is inconsistent, i.e.␣
↪it has no solution.")
            running = False
        except:
            print(error_msg)


def graph_equations_and_intersection():
    print("[ 3 ]")
    print("This function allows you to plot the graphs of two (linear or␣
↪quadratic) equations of your choice and, provided that\nthese graphs␣
↪intersect, to mark their point(s) of intersection.")
    print("--------------------------------")
    print(reminder_python_syntax)
    running = True
    x, y = symbols("x y")

    while running:
        try:
            choice_1 = int(input(" - regarding your first equation, enter '1'␣
↪for a linear or '2' for a quadratic equation: "))
            choice_2 = int(input(" - regarding your second equation, enter '1'␣
↪for a linear or '2' for a quadratic equation: "))
            if choice_1 in [1, 2] and choice_2 in [1, 2]:
                print("\nFIRST FUNCTION:")
                params_fct_1 = define_function(choice_1)
                if params_fct_1["type"] == "linear":
                    slope = params_fct_1["slope"]
                    intercept = params_fct_1["intercept"]
                    str_fct_1 = f"{slope}*x + {intercept}"
                elif params_fct_1["type"] == "quadratic":
                    coeff_quadr = params_fct_1["coeff_quadr"]
                    coeff_lin = params_fct_1["coeff_lin"]
                    coeff_const = params_fct_1["coeff_const"]
                    str_fct_1 = f"{coeff_quadr}*x**2 + {coeff_lin}*x +␣
↪{coeff_const}"
```

```
                fct_1 = str_fct_1.replace("**2", "\u00b2").replace("*", "").
↪replace("1x", "x").replace("0x\u00b2 +", "").replace("+ 0x", "").replace("+␣
↪0", "").replace("+ -", "- ")
                fct_1y = parse_expr(str_fct_1 + "- y")

                print("\nSECOND FUNCTION:")
                params_fct_2 = define_function(choice_2)
                if params_fct_2["type"] == "linear":
                    slope = params_fct_2["slope"]
                    intercept = params_fct_2["intercept"]
                    str_fct_2 = f"{slope}*x + {intercept}"
                elif params_fct_2["type"] == "quadratic":
                    coeff_quadr = params_fct_2["coeff_quadr"]
                    coeff_lin = params_fct_2["coeff_lin"]
                    coeff_const = params_fct_2["coeff_const"]
                    str_fct_2 = f"{coeff_quadr}*x**2 + {coeff_lin}*x +␣
↪{coeff_const}"
                fct_2 = str_fct_2.replace("**2", "\u00b2").replace("*", "").
↪replace("1x", "x").replace("0x\u00b2 +", "").replace("+ 0x", "").replace("+␣
↪0", "").replace("+ -", "- ")
                fct_2y = parse_expr(str_fct_2 + "- y")

                running = False

                # calculate the functions' intersection points
                solutions = solve([fct_1y, fct_2y], (x, y), dict=True)
                # ??? how to get rid of complex "solutions"?
                # cf. https://stackoverflow.com/questions/15210704/
↪ignore-imaginary-roots-in-sympy?rq=3
                # cf. https://stackoverflow.com/questions/63585690/
↪cant-constrain-sympy-variables-to-real-numbers?noredirect=1&lq=1
                # --> an "open issue" of SymPy?
                # real=True doesn't work; complex=False doesn't work;␣
↪real_roots()?
                print()
                print("The following intersections have been calculated:")
                try:
                    for s in solutions:
                        print(s)
                except Exception as e:
                    print(e)

                # plot the functions and, if they exist, their intersection␣
↪points
                fig, ax = plt.subplots()
                plt.axis([xmin, xmax, ymin, ymax])
```

```
                plt.plot([xmin, xmax], [0, 0], "lightgrey")
                plt.plot([0, 0], [ymin, ymax], "lightgrey")
                graph_x = np.linspace(xmin, xmax, points)
                if params_fct_1["type"] == "linear":
                    y_1 = params_fct_1["slope"] * graph_x +␣
↪params_fct_1["intercept"]
                elif params_fct_1["type"] == "quadratic":
                    y_1 = params_fct_1["coeff_quadr"] * graph_x**2 +␣
↪params_fct_1["coeff_lin"] * graph_x + params_fct_1["coeff_const"]
                plt.plot(graph_x, y_1)
                if params_fct_2["type"] == "linear":
                    y_2 = params_fct_2["slope"] * graph_x +␣
↪params_fct_2["intercept"]
                elif params_fct_2["type"] == "quadratic":
                    y_2 = params_fct_2["coeff_quadr"] * graph_x**2 +␣
↪params_fct_2["coeff_lin"] * graph_x + params_fct_2["coeff_const"]
                plt.plot(graph_x, y_2)
                plt.title(f"graphs for y = {fct_1} and y = {fct_2}")

                if len(solutions) > 0:
                    x_vals = list()
                    y_vals = list()
                    for s in solutions:
                        x_vals.append(s[x])
                        y_vals.append(s[y])
                    plt.plot(x_vals, y_vals, "gs")

                plt.show(block=False)
                plt.pause(0.01)

            else:
                print(error_msg)
        except Exception as e:
            print(e)
            print(error_msg)

def define_function(option_nr):
    params = {
        "type": None,
        "coeff_quadr": None,
        "coeff_lin": None,
        "coeff_const": None,
        "slope": None,
        "intercept": None
    }
    # define a linear function
    if option_nr == 1:
```

```
        running = True
        print("Based on the general form of a linear equation (y = mx + b), you␣
↪have to enter a slope (m) and an intercept (b).")
        print(integer_note)
        while running:
            try:
                m = int(input(" - your slope: m = "))
                b = int(input(" - your intercept: b = "))
                params["type"] = "linear"
                params["slope"] = m
                params["intercept"] = b
                return params
            except:
                print(error_msg)
    # define a quadratic function
    if option_nr == 2:
        running = True
        print("Based on the general form of a quadratic function, that is: f(x)␣
↪= ax\u00b2 + bx + c,\nyou have to enter (integer) values for the␣
↪coefficients a, b, c.")
        print(integer_note)
        while running:
            try:
                a = int(input(" - your quadratic coefficient:  a = "))
                b = int(input(" - your linear coefficient:     b = "))
                c = int(input(" - your constant coefficient:   c = "))
                params["type"] = "quadratic"
                params["coeff_quadr"] = a
                params["coeff_lin"] = b
                params["coeff_const"] = c
                return params
            except:
                print(error_msg)


def plot_quadratic_function():
    print("[ 4 ]")
    print("This function allows you to plot the roots and the vertex of a␣
↪simple quadratic function, based on your input\nof values for the three␣
↪relevant coefficients (quadratic / linear / constant).")
    print(integer_note)
    print("--------------------------------")
    running = True
    while running:
        try:
```

```
            print("Based on the general form of a quadratic function, that is:␣
↪f(x) = ax\u00b2 + bx + c,\nyou have to enter (integer) values for the␣
↪coefficients a, b, c.")
            a = int(input(" - your quadratic coefficient:  a = "))
            b = int(input(" - your linear coefficient:     b = "))
            c = int(input(" - your constant coefficient:   c = "))

            # calculate the roots
            x_1, x_2 = None, None
            print("RESULT:")
            if (b**2 - 4*a*c) < 0:
                print("--> Your function has no real roots.")
            else:
                if a == 0:
                    if b == 0:
                        if c == 0:
                            print("--> Yours is a constant function with c = 0,␣
↪so it has an infinite number of roots.")
                        else:
                            print("--> Yours is a constant function with c !=␣
↪0, so it has no roots.")
                    else:
                        x_1 = -c/b
                        print(f"--> Your function has the root {x_1:g}.")
                else:
                    if (b**2 - 4*a*c) == 0:
                        x_1 = -b/(2*a)
                        print(f"--> Your function has the root {x_1:g}.")
                    elif (b**2 - 4*a*c) > 0:
                        x_1 = (-b + math.sqrt(b**2 - 4*a*c))/(2*a)
                        x_2 = (-b - math.sqrt(b**2 - 4*a*c))/(2*a)
                        print(f"--> Your function has the roots {x_1:g} and␣
↪{x_2:g}.")

            # calculate the vertex (using the "vertex formula")
            vx, vy = None, None
            if a != 0:
                vx = -b/(2*a)
                vy = a*vx**2 + b*vx + c
                print(f"--> The vertex of its graph is to be found at ({vx:g},␣
↪{vy:g}).")
            else:
                print("--> Since your function does not contain a quadratic␣
↪term, its graph is not a parabola - so there is no vertex.")

            running = False
```

```
        if a == 0:
            if b == 0:
                equation = c
            else:
                if b == 1:
                    v = ""
                elif b == -1:
                    v = "-"
                else:
                    v = b
                if c == 0:
                    equation = f"{v}x"
                elif c > 0:
                    equation = f"{v}x + {c}"
                else:
                    equation = f"{v}x - {abs(c)}"
        elif a == 1:
            if b == 0:
                if c == 0:
                    equation = "x\u00b2"
                elif c > 0:
                    equation = f"x\u00b2 + {c}"
                else:
                    equation = f"x\u00b2 - {abs(c)}"
            else:
                if b == 1:
                    if c == 0:
                        equation = f"x\u00b2 + x"
                    elif c > 0:
                        equation = f"x\u00b2 + x + {c}"
                    else:
                        equation = f"x\u00b2 + x - {abs(c)}"
                elif b == -1:
                    if c == 0:
                        equation = f"x\u00b2 - x"
                    elif c > 0:
                        equation = f"x\u00b2 - x + {c}"
                    else:
                        equation = f"x\u00b2 - x - {abs(c)}"
                elif b > 1:
                    if c == 0:
                        equation = f"x\u00b2 + {b}x"
                    elif c > 0:
                        equation = f"x\u00b2 + {b}x + {c}"
                    else:
                        equation = f"x\u00b2 + {b}x - {abs(c)}"
```

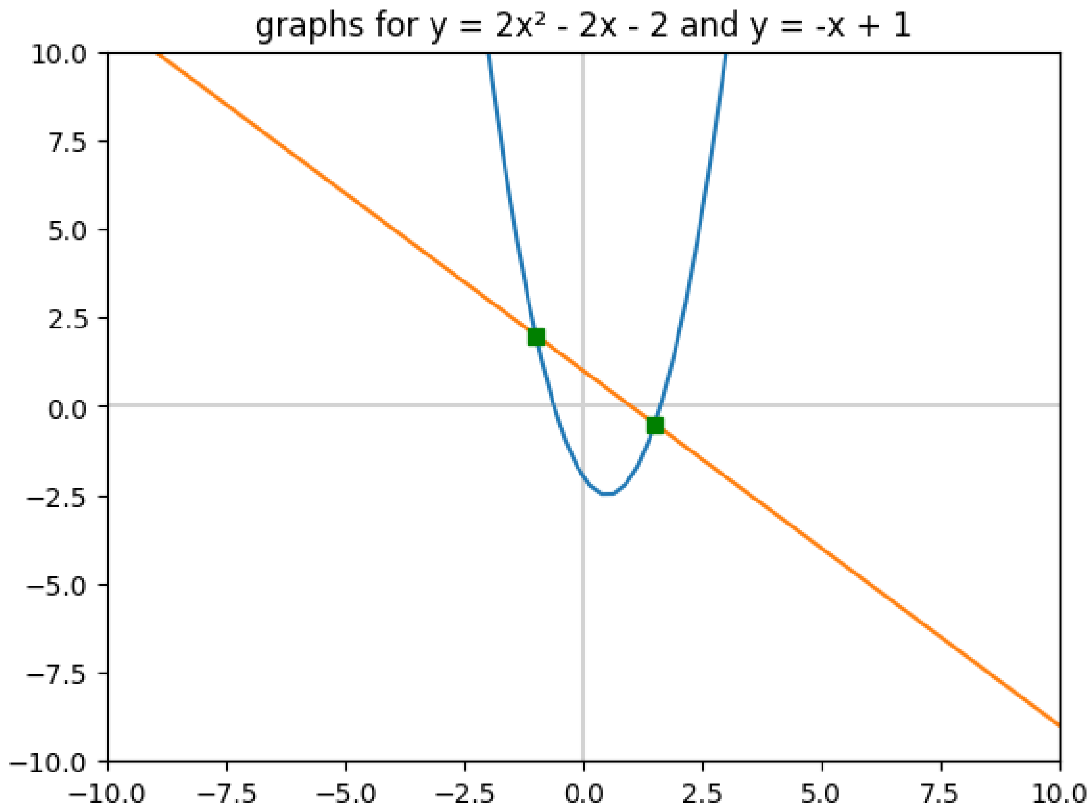
```
            elif b < -1:
                if c == 0:
                    equation = f"x\u00b2 - {abs(b)}x"
                elif c > 0:
                    equation = f"x\u00b2 - {abs(b)}x + {c}"
                else:
                    equation = f"x\u00b2 - {abs(b)}x - {abs(c)}"
    elif a == -1:
        if b == 0:
            if c == 0:
                equation = "-x\u00b2"
            elif c > 0:
                equation = f"-x\u00b2 + {c}"
            else:
                equation = f"-x\u00b2 - {abs(c)}"
        else:
            if b == 1:
                if c == 0:
                    equation = "-x\u00b2 + x"
                elif c > 0:
                    equation = f"-x\u00b2 + x + {c}"
                else:
                    equation = f"-x\u00b2 + x - {abs(c)}"
            elif b == -1:
                if c == 0:
                    equation = "-x\u00b2 - x"
                elif c > 0:
                    equation = f"-x\u00b2 - x + {c}"
                else:
                    equation = f"-x\u00b2 - x - {abs(c)}"
            elif b > 1:
                if c == 0:
                    equation = f"-x\u00b2 + {b}x"
                elif c > 0:
                    equation = f"-x\u00b2 + {b}x + {c}"
                else:
                    equation = f"-x\u00b2 + {b}x - {abs(c)}"
            elif b < -1:
                if c == 0:
                    equation = f"-x\u00b2 - {abs(b)}x"
                elif c > 0:
                    equation = f"-x\u00b2 - {abs(b)}x + {c}"
                else:
                    equation = f"-x\u00b2 - {abs(b)}x - {abs(c)}"
    elif a > 1 or a < -1:
        if b == 0:
            if c == 0:
```

```
                equation = f"{a}x\u00b2"
            elif c > 0:
                equation = f"{a}x\u00b2 + {c}"
            else:
                equation = f"{a}x\u00b2 - {abs(c)}"
        else:
            if b == 1:
                if c == 0:
                    equation = f"{a}x\u00b2 + x"
                elif c > 0:
                    equation = f"{a}x\u00b2 + x + {c}"
                else:
                    equation = f"{a}x\u00b2 + x - {abs(c)}"
            elif b == -1:
                if c == 0:
                    equation = f"{a}x\u00b2 - x"
                elif c > 0:
                    equation = f"{a}x\u00b2 - x + {c}"
                else:
                    equation = f"{a}x\u00b2 - x - {abs(c)}"
            elif b > 1:
                if c == 0:
                    equation = f"{a}x\u00b2 + {b}x"
                elif c > 0:
                    equation = f"{a}x\u00b2 + {b}x + {c}"
                else:
                    equation = f"{a}x\u00b2 + {b}x - {abs(c)}"
            elif b < -1:
                if c == 0:
                    equation = f"{a}x\u00b2 - {abs(b)}x"
                elif c > 0:
                    equation = f"{a}x\u00b2 - {abs(b)}x + {c}"
                else:
                    equation = f"{a}x\u00b2 - {abs(b)}x - {abs(c)}"

    # plot the graph, the roots and the vertex
    fig, ax = plt.subplots()
    plt.axis([xmin, xmax, ymin, ymax])
    plt.plot([xmin, xmax], [0, 0], "lightgrey")
    plt.plot([0, 0], [ymin, ymax], "lightgrey")
    x = np.linspace(xmin, xmax, points)
    y = a*x**2 + b*x + c
    plt.plot(x, y)
    plt.title(f"graph for {equation}")
    plt.plot([vx], [vy], "ro")
    if x_1 is not None:
        plt.plot([x_1], [0], "gs")
```

```
            if x_2 is not None:
                plt.plot([x_2], [0], "gs")
            plt.show()

        except Exception as e:
            print(e)
            print(error_msg)



# USER MENU
print("*** SOLVING & GRAPHING FUNCTIONS ***")
print("====================================")
print()
print("You can select between one of the following options:")
print("--> 1 = display the graph and a table of values for any 'y=' equation␣
 ↪input")
print("--> 2 = solve a system of two equations, without graphing")
print("--> 3 = graph two equations and plot the point of intersection")
print("--> 4 = given a, b and c in a quadratic equation, plot the roots and␣
 ↪vertex")
print()
print("--> 0 = quit the program")
# print()

# PROCESSING THE USER'S CHOICE
programm_running = True
while programm_running:
    print()
    user_choice = input("Please enter the number corresponding to the action of␣
 ↪your choice: ")
    match user_choice:
        case "0":
            print()
            print("****************")
            print(" G O O D B Y E !")
            programm_running = False
        case "1":
            display_graph_and_values()
        case "2":
            solve_two_equations()
        case "3":
            graph_equations_and_intersection()
        case "4":
            plot_quadratic_function()
        case _:
            print("# No valid input! - Try it again ...")
```

```
# This step does not have a test
```

```
*** SOLVING & GRAPHING FUNCTIONS ***
====================================

You can select between one of the following options:
--> 1 = display the graph and a table of values for any 'y=' equation input
--> 2 = solve a system of two equations, without graphing
--> 3 = graph two equations and plot the point of intersection
--> 4 = given a, b and c in a quadratic equation, plot the roots and vertex

--> 0 = quit the program

Please enter the number corresponding to the action of your choice: 3
[ 3 ]
This function allows you to plot the graphs of two (linear or quadratic)
equations of your choice and, provided that
these graphs intersect, to mark their point(s) of intersection.
--------------------------------
(--> Remember to use Python syntax with x and y variables!)
 - regarding your first equation, enter '1' for a linear or '2' for a quadratic
equation: 2
 - regarding your second equation, enter '1' for a linear or '2' for a quadratic
equation: 1

FIRST FUNCTION:
Based on the general form of a quadratic function, that is: f(x) = ax² + bx + c,
you have to enter (integer) values for the coefficients a, b, c.
(For the sake of simplicity, the permitted entries of this exercise are limited
to integers!)
 - your quadratic coefficient:  a = 2
 - your linear coefficient:     b = -2
 - your constant coefficient:   c = -2

SECOND FUNCTION:
Based on the general form of a linear equation (y = mx + b), you have to enter a
slope (m) and an intercept (b).
(For the sake of simplicity, the permitted entries of this exercise are limited
to integers!)
 - your slope: m = -1
 - your intercept: b = 1

The following intersections have been calculated:
{x: -1, y: 2}
{x: 3/2, y: -1/2}
```




```
Please enter the number corresponding to the action of your choice: 0

*****************
 G O O D B Y E !
```

```
[ ]:
```