Hey Peter, here are some general global points I wanted to draw your attention to. You'll see examples where we've done this throughout:

Always talk through code, even if it's a couple of lines. Pay special attention to any new code

Introduce a new piece of code, let the readers know what we are about to do. That will give them context when they are reading through.

Any new code needs to be in text rather than in screenshots only – that helps a reader know it's an integral part of the book, among other things

Separate output from the program --- much less confusing, and also interesting to see

4

"Mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true."

- Bertrand Russell

Algebra: Transforming and Storing Numbers

Algebra is often introduced in school as the idea of replacing numbers with letters. For example, instead of using 2·3 you might use 2*x* where x is a placeholder that can be any number, meaning "2 times some unknown number." It's just ~~You might think of them~~ like variables, which we've already used to replace numbers we can change the value of: it's a vital tool in programming.

In traditional math class, variables are often used to represent a mystery number that a student is required to find. Figure 4.1 shows a student's cheeky response to the problem, "Find *x*":

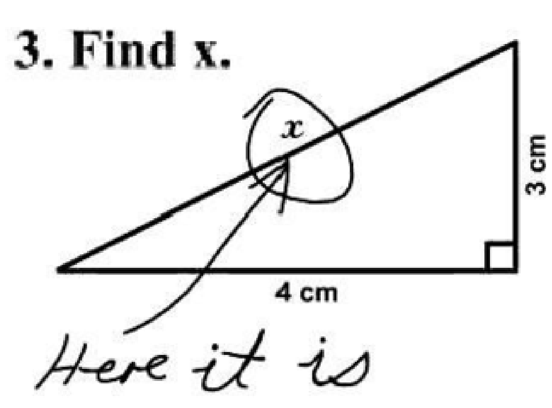

Figure 3-1: Locating the x variable instead of solving for its value.

As you can see, the student has located the variable *x* in the diagram instead of solving for its possible value. Often in algebra, students are given an equation with an unknown element (or two or three…) and are tasked with finding out what number or numbers will make the equation true, like this:

Solve $2x + 5 = 13$.

In this context, solve means to find out which number, when you replace *x* with that number, makes both sides of the equation equal. Equations of this form may not take too much effort to solve, but even at this level we can use Python and the *brute force* method to plug in random numbers and find the right one.

Brute Forcing an Algebra Equation

So we're trying to find a number x that, when you multiply it by 2 and add 5, you get 13. I'll make a guess that it's somewhere between -100 and 100. ~~W~~ So we'll **write** a program that plugs all the integers between -100 and 100 in to the above equation, checks the output and prints out the number that makes the equation true.

~~**Solution:**~~

```
def plug():
❶    x = -100 #start at -100
    while x < 100: #go up to 100
❷        if 2*x + 5 == 13: #if it makes the equation true
            print("x =",x) #print it out
❸        x += 1 #make x go up by 1 to test the next number

plug() #run the plug function
```

Listing 3-1: Brute force program that plugs in numbers to see which satisfies the equation

Here we define the function as `plug()` and initialize the x variable at -100 ❶ On the next line we start a while loop that will keep repeating until x equals 100. On line ❷ we multiply whatever x is by 2 and add 5. If the output equals 13, it prints out the number. On line ❸ we "increment" x by 1 which means we make it go up by 1. It starts the loop over. The last line makes the program run the plug function. If you don't include that line it won't do anything! The output should be something like this:

```
x = 4
```

This is a perfectly valid way to solve this problem if you don't know how to balance equations! The trial and error method of plugging in all those digits by hand is too laborious, but using a computer it's a cinch. If you suspect the solution ~~won't be~~isn't an integer, you might have to increment by smaller numbers by changing the line at ❸ to `x += .25` or some other decimal.

Solving First-degree Equations

Another way to solve an equation like $2x + 5 = 13$ is to find a general formula for this type of equation. We can then use this formula to write a program in Python. The equation $2x + 5 = 13$ is an example of a *first-degree equation*, because the highest exponent a variable has is 1. And a number to the 1 power is just the number itself.All first degree equations fit into this general pattern:

ax + b = cx + d

where a, b, c and d are numbers. Like these ones~~:polynomials examples of first-degree moreHere are some~~

$$3x - 5 = 22$$

$$4x - 12 = 2x - 9$$

$$\frac{1}{2}x + \frac{2}{3} = \frac{1}{5}x + \frac{7}{8}$$

In each equation, you usually have an x-term and a constant (a number with no x) on each side of the equals sign.Sometimes there's no *x* term on one side, so the coefficient of x is zero. You can see this in the first example, $3x - 5 = 22$, where 22 is the only term on the right side of the equal sign. In the general formula, a is represented by 3, b by -5, c by 0 and d by 22. In the second example, a = 4, b = -12, c = 2 and d = -9.

Using a little algebra, you can solve $ax + b = cx + d$ for x, which means you can solve virtually all equations of this form.

To solve this equation we first get all the *x*'s on one side of the equals sign by subtracting c*x* and b from both sides of the equation:

$$ax - cx = d - b$$

Now factor out the *x* from a*x* and c*x*:

$$x(a - c) = d - b$$

Finally, divide both sides by a – c to isolate *x*, which gives you the value of *x* in terms of a, b, c, and d.

$$x = \frac{d - b}{a - c}$$

This is the general equation you can use to solve for any variable *x* when the equation is a first-degree equation and all four numbers (a, b, c and d) are known. Now let's write this into a Python program that can solve algebraic equations of this form for us.

Using Python to Solve for x

We'll write a program that will take the four coefficients of the general equation above and print out the solution for *x*. Open a new Python file in IDLE and create a new blank file. Save it as "algebra.py." We're going to write a function that will take the four numbers a, b, c and d as parameters and plug them into the solution above:

def equation(a,b,c,d):

```
    '''solves equations of the
    form ax + b = cx + d'''
    return (d - b)/(a - c)
```

Listing 3-2: Using programming to solve for x

We had to do a little algebra to find the general solution above, then we wrote a Python function to transform the numbers a, b, c and d into the solution to that family of equations. Does it work? Let's test it with an equation we've solved already, $2x + 5 = 13$:

```
>>> equation(2,5,0,13)
4.0
```

It works!, We get 4 as the solution for the equation..

Exercise 3.1: X-Games

Solve $12x + 18 = -34x + 67$ using the program you wrote in Listing 3-2.

Solution:

In Listing 3-2, we didn't use `print` to display our results, but used `return` instead. The `return` term gives us our result as a number that we can assign to a variable and then use again. If we had used `print` instead we could only print out the solution once:

```
def equation(a,b,c,d):
    '''solves equations of the
    form ax + b = cx + d'''
    print((d - b)/(a – c))

>>> x = equation(2,5,0,13)
4.0
>>> print(x)
None
```

Yes, it gave us the answer once, but it didn't save it. There are many times in programming where you want the program to just save the output of a function and apply it elsewhere. Like this next example. (Make sure your "equation" code uses return.)

Let's reuse the equation 12*x* + 18 = -34*x* + 67 from Exercise 3.1 and assign the result to the *x* variable, as shown here:

```
>>> x = equation(12,18,-34,67)
>>> x
1.065217391304348
```

Here, we pass the coefficients and constants of our equation to the `equation()` function so it calculates *x* for us, and assigns the result to the variable *x*. Then we can see the value of *x* by simply entering it.Now that the variable *x* stores the solution, let's plug it back into the equation to check that it's the correct answer. Enter the following to find out what 12*x* + 18, the left side of the equation, evaluates to:

```
>>> 12*x + 18
30.782608695652176
```

We get `30.782608695652176`. Now enter the following to do the same for -34*x* + 67, the right side of the equation:

```
>>> -34*x + 67
30.782608695652172
```

Despite a slight rounding discrepancy at the 15th decimal place, you can see that both sides of the equation evaluates to around 30.782608. So we can be confident that 1.065217391304348 is the correct solution for *x*! Good thing we returned the solution and saved the value instead of just printing it out once. You wouldn't have wanted to type in "1.065217391304348" again and again, would you?

Exercise : Do you do fractions?

Now use your equation function to solve the last, most sinister looking, equation in the examples:

$$\frac{1}{2}x+\frac{2}{3}=\frac{1}{5}x+\frac{7}{8}$$

Solution:

Plug the coefficients and constants in to the equation function:

```
>>> x = equation(1/2,2/3,1/5,7/8)
>>> x
```

```
0.6944444444444446
>>> 1/2*x + 2/3
1.0138888888888888
>>> 1/5*x + 7/8
1.0138888888888888
```

We passed the fractions in the equation to the equation function, and saved the output to a variable x. When we enter "x" it gives us **the solution, around 0.694**. Without having to type the long decimal again, we can make sure the left side of the equation, using the solution for x, equals the right side of the equation. They both equal 1.0138888888888888. Check!

Solving Higher Degree Equations

Things get a little more complicated when an equation has a term raised to the second degree, like $x^2 + 3x - 10 = 0$. Equations of this form are called *quadratic equations* and have a general format that looks like this: $ax^2 + bx + c = 0$. a, b and c can be any number (well, a can't be 0 or you'd be back to a first degree equation), positive or negative, whole numbers, fractions or decimals. To solve an equation with a squared term, you can use the Quadratic Formula:

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}\quad.$$

The Quadratic Formula is a very powerful tool for solving equations. No matter what a, b and c are in $ax^2 + bx + c = 0$, you just plug them in to the Formula and do the arithmetic to find your solutions. In $x^2 + 3x - 10 = 0$, a = 1, b = 3 and c = -10. Plugging those in to the Quadratic Formula, we get

$$x=\frac{-3\pm\sqrt{3^2-4(1)(-10)}}{2(1)}$$

Which simplifies to

$$x=\frac{-3\pm\sqrt{49}}{2}=\frac{-3\pm 7}{2}$$

There are two solutions, one for $\frac{-3+7}{2}$ which is 4/2 or 2

The other is $\frac{-3-7}{2}$ which is -10/2 or -5.

Both of those numbers, when you replace x with them, make the original formula true:

$(2)^2 + 3(2) - 10 = 4 + 6 - 10 = 0$

$(-5)^2 + 3(-5) - 10 = 25 - 15 - 10 = 0$

Exercise 3.2~~4.3~~: Working on our Quads

Write a function that will take the three numbers (a, b and c), from a quadratic equation ax^2 + bx + c = 0 and return the two solutions to the quadratic formula. Use it to solve 2x^2 + 7x – 15 = 0.

Solution:

We'll call the function `quad()`. Before we do anything, we'll need to import the `sqrt` method from the `math` module. The `sqrt` method allows you to find the square root of a number, just like a square root button on a calculator. It works great for positive numbers, but if you try finding the square root of a negative number this is the error you'll see:

```
>>> from math import sqrt
>>> sqrt(-4)
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    sqrt(-4)
ValueError: math domain error
```

Open a new Python file in IDLE and add the following line to the top of your file to import `sqrt` from `math`:

```
from math import sqrt
```

Then enter the following to create the `quad()` function:

```
def quad(a,b,c):
    '''Returns the solutions of an equation
    of the form a*x**2 + b*x + c = 0'''
    x1 = (-b + sqrt(b**2 - 4*a*c))/(2*a)
    x2 = (-b - sqrt(b**2 - 4*a*c))/(2*a)
    return x1,x2
```

Listing 3-3: Using the quadratic equation to solve an equation

The `quad()` function takes the numbers a, b and c as parameters and plugs them in to the Quadratic Formula. x1 is the solution when we're adding $-b+\sqrt{b^2-4ac}$ and x2 is the solution when we're subtracting $-b-\sqrt{b^2-4ac}$

Now, let's test this program using the equation $2x^2 + 7x - 15 = 0$. Plugging in the numbers 2, 7 and -15 for a, b, and c should return the following output:

```
>>> quad(2,7,-15)
(1.5, -5.0)
```

The two solutions for x are 1.5 and -5, which means both values should satisfy the equation $2x^2 + 7x - 15 = 0$. To test this, we check the answers by replacing all the x's in the original equation $2x^2 + 7x - 15 = 0$ with 1.5, the first solution, and then with -5, the second solution, as shown here:

```
>>> 2*1.5**2 + 7*1.5 - 15
0.0
>>> 2*(-5)**2 + 7*(-5) - 15
0
```

Yes! Both values work in the original equation. You can use your "equation" and "quad" functions anytime in the future. Now that you've learned to write functions to solve first-degree and second-degree equations, let's learn how to solve even higher degree equations!

Solving Higher Degree Equations with Visualizations

In algebra class students are often asked to solve a *cubic equation* like $6x^3 + 31x^2 + 3x - 10 = 0$, which has a term raised to the third degree. We can use the `plug()` function we made in Listing 3-1 to look for solutions manually. Enter the following into IDLE to see this in action:

```
def g(x):
    return 6*x**3 + 31*x**2 + 3*x – 10

def plug():
    x = -100
    while x < 100:
        if g(x) == 0:
            print("x =",x)
        x += 1
    print("done.")
```

Listing 3-4: Using plug() to solve a cubic polynomial

First, we define `g(x)` to be a function that evaluates the expression `6*x**3 + 31*x**2 + 3*x – 10`. Then we tell the program to plug all numbers between -100 and 100 into `g(x)` we defined initially. If the program finds a number that makes `g(x)` equal zero, it prints it for the user.

When you call `plug()` you should see the following output:

```
>>> plug()
```

```
x = -5
done.
```

That gives you -5 as the solution, but as you might suspect from the term x^3, there ~~are actually~~ could be as many as three possible solutions to this equation. Fortunately, there's a way to see all the possible inputs and corresponding outputs of a function; it's called *graphing.*

In the next section, you'll build a graph using a nifty tool called Processing to make visualizations that will help you solve higher degree polynomials.

Downloading and Setting up Processing

~~Creating Your Own Graphing Tool~~

~~We're going to use Processing to create a graphing tool that will allow us to see how many solutions our equation has.~~ **The basic setup of a Processing sketch in Python mode is shown in Figure 4.2:**

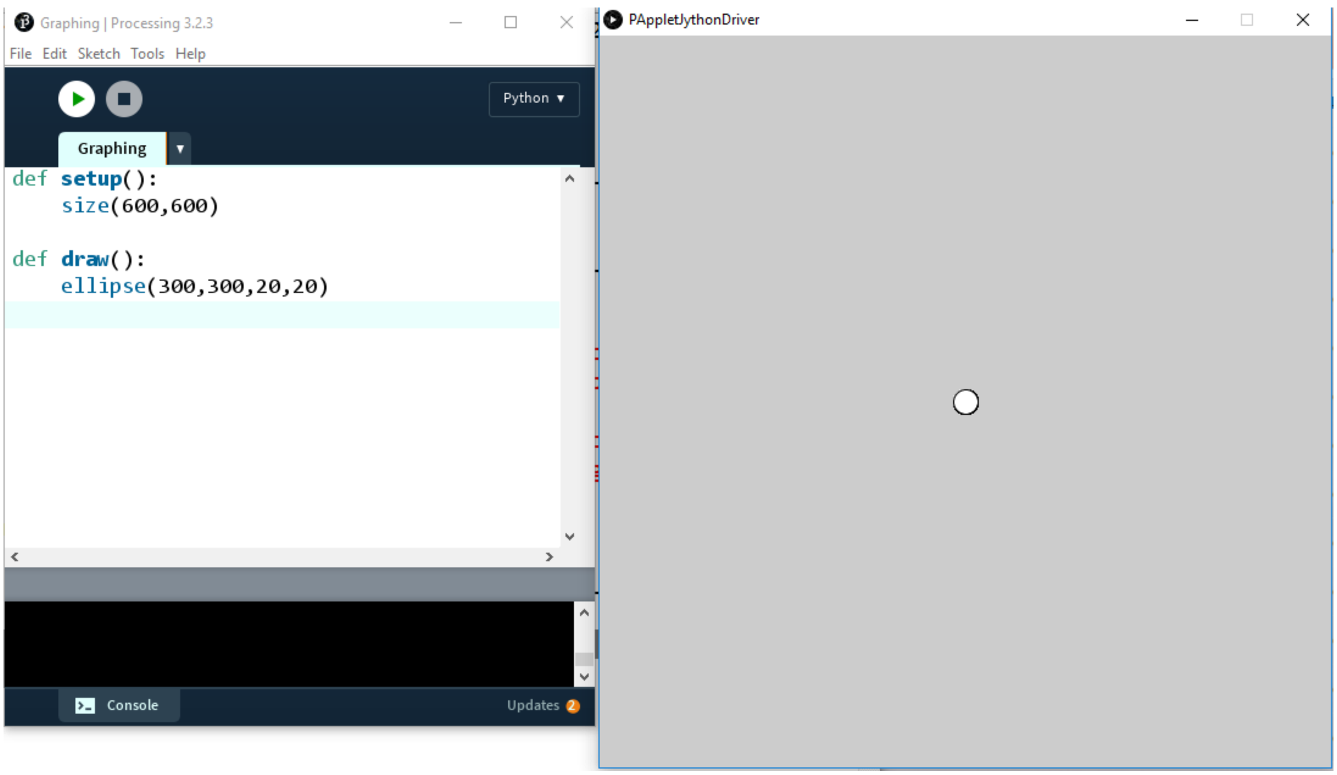

Figure 3-2:

Surprisingly, computer graphics usually have their “origin” in the top left corner of the screen and the y-coordinates get larger as they go down the screen, not up, like in Figure 4.3:

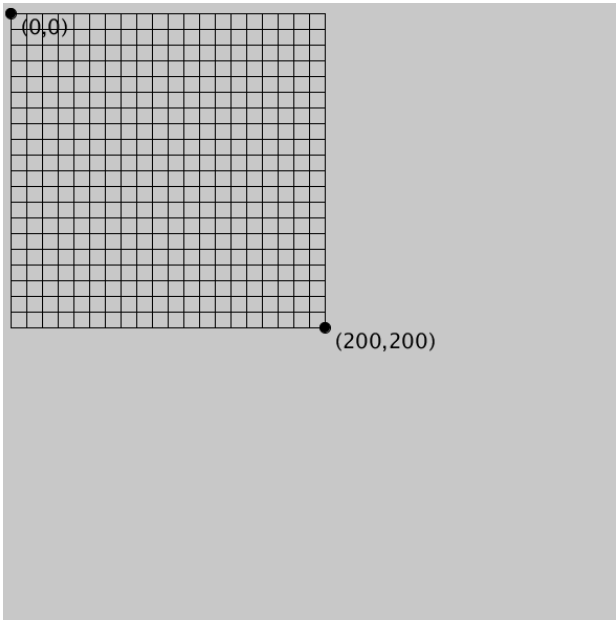

Figure 3~~4~~.3

To move the origin to the center of the screen, we use the “translate” function, like this:

```
translate(width/2, height/2)
```

Now the origin will be in the middle of the screen. Figure 4.4 ~~gives~~shows you ~~the idea~~what the grid looks like after translating the origin to the middle of the screen:

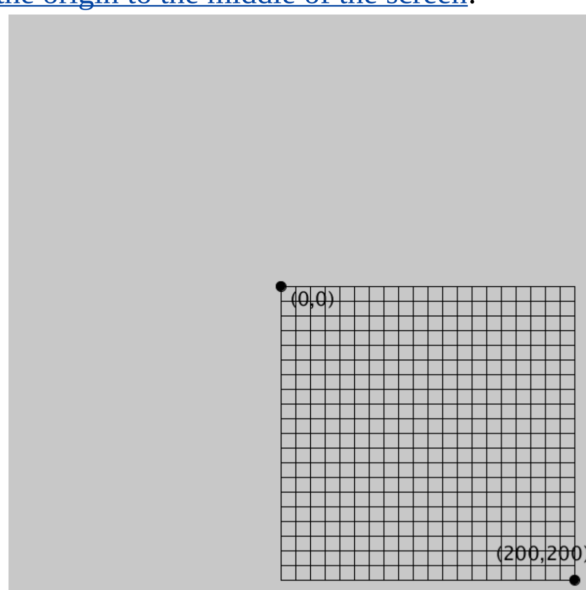

Figure 3~~4~~.4

Creating Your Own Graphing Tool

Now that you've downloaded Processing, let's use it to create a graphing tool that will allow us to see how many solutions an equation has. First, we'll create a grid of blue lines that will look like graph paper. Then, we'll create the x- and y-axes using black lines.

Setting Graph Dimensions

Before we can make a grid for our graphing tool, we'll need to set the dimensions of the display window. In Processing, you can use the `size()` function to indicate the width and height of the screen in pixels. The default screen is 600 pixels by 600 pixels, but for our graphing tool we'll create a graph that includes x- and y- values ranging from -10 to 10. Open a new file in Processing and enter the following code to declare the range of x- and y-values we're interested in displaying for our graph:

```
#grid.pyde

#set the range of x-values
xmin = -10
xmax = 10

#range of y-values
ymin = -10
ymax = 10

#calculate the range
rangex = xmax - xmin
rangey = ymax - ymin

def setup()
    size(600,600)
```

Listing 3-5: Setting the range of x- and y-values for the graph

In Listing 3-5 we can create two variables: xmin and xmax for the minimum and maximum x-values in our grid. Then we declare `rangex` for the x-range and `rangey` variable for the y-range. We calculate rangex by subtracting the xmin from the xmax and do the same for the y-values.

Because we don't need a graph that's 600 units by 600 units, we'll need to scale the coordinates down by multiplying the x- or y-coordinates by scale factors. When graphing we'll have to remember to multiply all our x-coordinates and y-coordinates by these scale factors or they won't show up correctly on the screen. To do this, add the following lines of code below

```
def setup()
```

```
    global xscl, yscl
    size(600,600)

    xscl = width/rangex
    yscl = height/rangey
```

Listing 3-6: Scaling coordinates using scale factors

First, we create a setup`()` function and declare global variables `xscl` and `yscl`, which we'll use to scale our screen. `xscl` and `yscl` stand for the x-scale factor and y-scale factor, respectively. For example, the x-scale factor would be 1 if we want our x-range to be 600 pixels, or the full width of the screen. But if we want our screen to be between -300 and 300, the x-scale factor would be 2, which we get by dividing the width (600) by the x-range (300). In our case, the scale factor would be 600 divided by the x-range, which is 20. So the scale factor is 30. All our x- and y-coordinates will have to be scaled up by a factor of 30 to be able to see them on the screen. The good news is the computer will do all the dividing and scaling for us from now on. We just have to remember to use the xscl and yscl when graphing!

Drawing a Grid

Now we'll draw in lines for the grid like on graph paper. All the drawing will go in the draw() function, the infinite loop.

```
def draw():
    global xscl, yscl
    background(255) #white
    translate(width/2,height/2)
    #cyan lines
    strokeWeight(1)
    stroke(0,255,255)
    for i in range(xmin,xmax+1):
        line(i*xscl,-10*yscl,i*xscl,10*yscl)
        line(-10*xscl,i*yscl, 10*xscl,i*yscl)
```

Listing 3-7: Creating blue grid lines for the graph

First we have to tell Python we’re not creating new variables called xscl and yscl, we want to use the global ones that were already created. Then we set the background color to white using the value 255. Translate means move or slide, and we move the origin (where x and y are both 0) to the center of the screen using `translate(width/2,height/2)`. “strokeWeight” is the thickness of the lines and 1 is the thinnest. You can make them thicker if you want by using higher numbers. “Stroke” means the color of the lines, and for cyan (“sky blue”) the RGB value is (0,255,255), meaning no red, maximum green and maximum blue.

After that we use a for loop to save us having to type 40 lines of code to draw 40 blue lines. We want the blue lines to go from xmin to xmax, including xmax. But remember, we saw “range(4)” contains 4 numbers, *not* including 4:

```
>>> for i in range(4):
    print(i)
0
1
2
3
```

In order for us to get from 1 to 4 we have to “start” at 1 and “stop” at 5:

```
>>> for i in range(1,5):
    print(i)
1
2
3
4
```

So our for loop goes from xmin to xmax + 1 to include xmax.

The “line” code might be hard to figure out at first. In Processing, you draw a line by declaring its endpoints, so you need 4 numbers: the x and y of the beginning of the line and the x and y of the end. Written out, the vertical lines would be

```
line(-10,-10, -10,10)
line(-9,-10, -9,10)
line(-8,-10,-8,10)
```

and so on. See a pattern? The x-values go from xmin to xmax. So we could put that in a for loop like this:

```
for i in range(xmin,xmax+1):
    line(i,-10,i,10)
```

Similarly, the horizontal lines would go like this:

```
line(-10,-10,10, -10)
line(-10,-9, 10, -9)
line(-10,-8, 10, -8)
```

and so on. It’s the y-values that are changing this time. We could just add another line inside our loop:

```
for i in range(xmin,xmax+1):
    line(i,-10,i,10)
    line(-10,i,10,i)
```

If you graphed this right now you might see a tiny splotch in the middle of the screen because the x and y-coordinates go from -10 to 10 but the screen goes from 0 to 600. We need to multiply all our x- and y-coordinates by their scale factor in order for them to display properly. That’s why the lines are

```
line(i*xscl,-10*yscl,i*xscl,10*yscl)
line(-10*xscl,i*yscl, 10*xscl,i*yscl)
```

Now we'll create the axes.

Exercise 3-3: I Walk The Line

Now add the two black lines for the x- and y-axes.

Solution:

First we set the stroke color to black by calling the `stroke()` function, with 0 being black (255 would be white). Then we draw a vertical line from the point (0, -10) to the point(0, 10) and a horizontal line from (-10,0) to (10,0). Don’t forget to multiply the values by their respective scale factors, unless they’re 0, in which case, multiplying them wouldn’t change them anyway.

```
    stroke(0) #black axes
    line(0,-10*yscl,0,10*yscl)
    line(-10*xscl,0, 10*xscl,0)
```

The above code gives us a nice grid, like in Figure 3~~4~~.5:

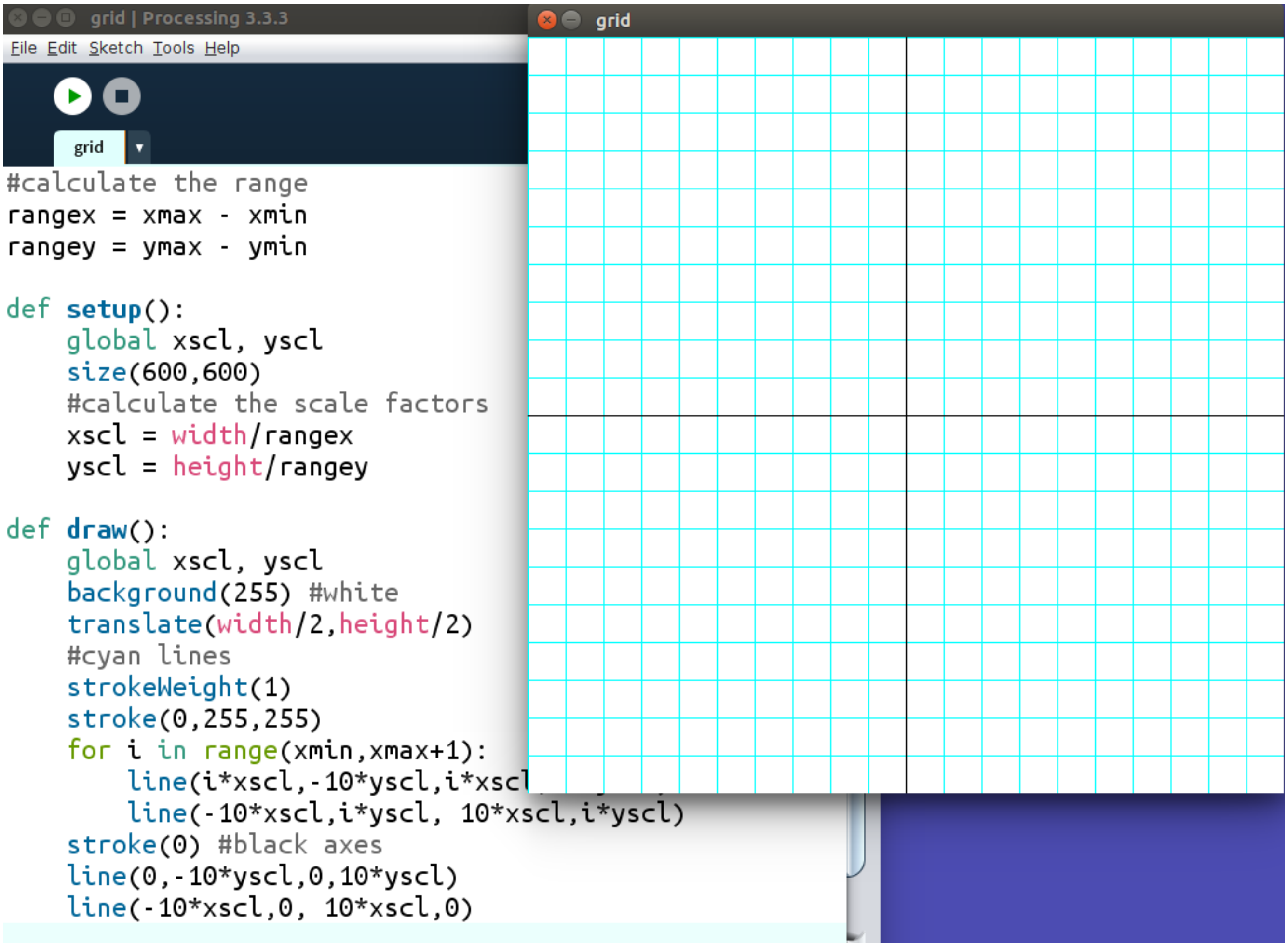

Figure 34.5: Creating a grid for graphing. You only have to do it once!

This looks done, but if we try to put a point (an ellipse actually) at (3,6), we see a problem:

```
#test with a circle
fill(0)
ellipse(3*xscl, 6*yscl,10,10)
```

You'll see what's in Figure 34.6:

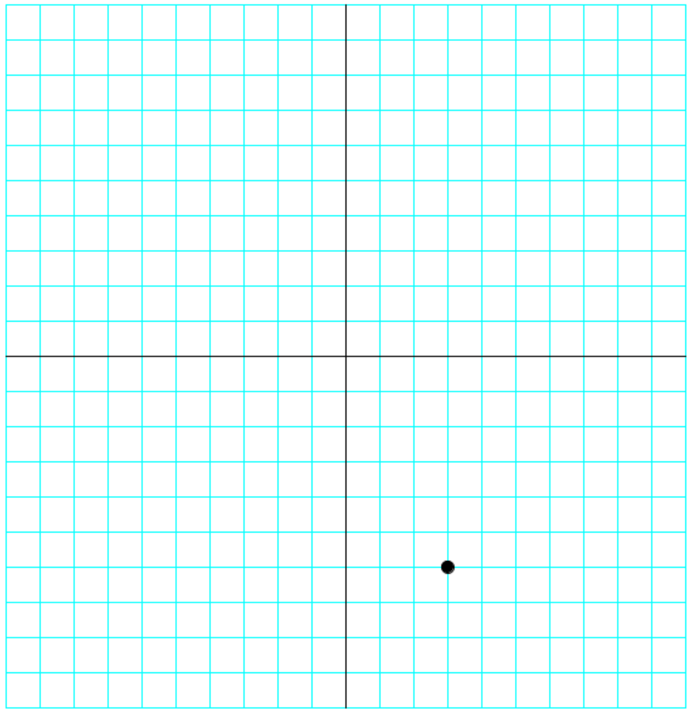

Figure 3~~4~~.6: Checking our graphing program. Almost there!

The y-coordinates go up as we go **down** the screen, so our graph is upside-down. We can add a negative sign to the y-scale factor in the setup function to flip that over:

```
yscl = -height/rangey
```

This flips the output to how we want it , like in Figure 4.7:

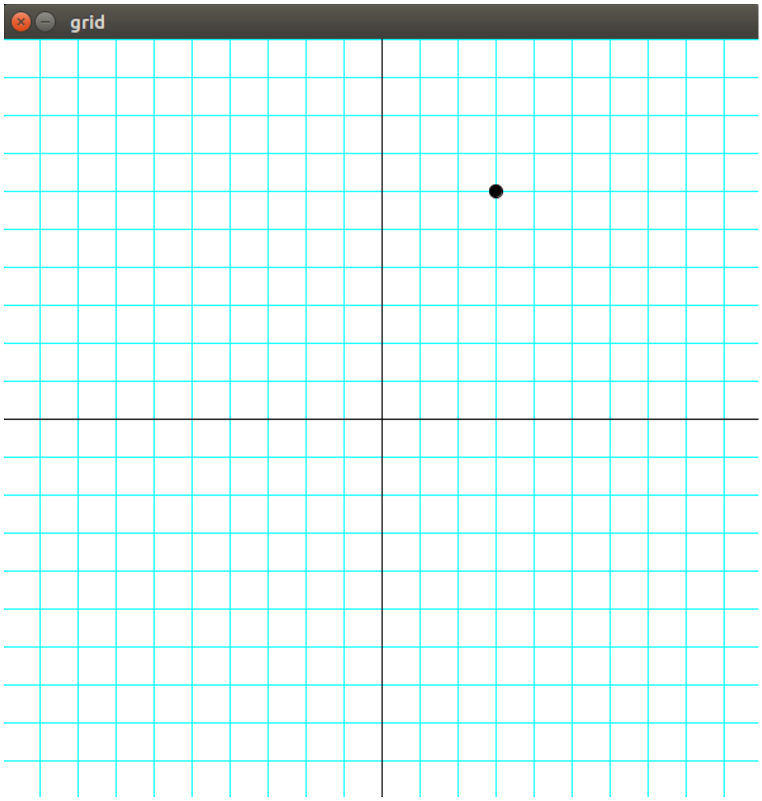

Figure 34.7: The grapher is working!

The grid function

To keep our code organized, we'll separate all the code that makes the grid into its own function and call it "`grid`." Then we'll call the "`grid`" function in the draw function. It should look like this:

```
def draw():
    global xscl, yscl
    background(255)
    translate(width/2,height/2)
    grid(xscl,yscl) #draw the grid

def grid(xscl,yscl):
    #cyan lines
    strokeWeight(1)
    stroke(0,255,255)
    for i in range(xmin,xmax+1):
        line(i*xscl,-10*yscl,i*xscl,10*yscl)
        line(-10*xscl,i*yscl, 10*xscl,i*yscl)
    stroke(0) #black axes
```

```
    line(0,-10*yscl,0,10*yscl)
    line(-10*xscl,0, 10*xscl,0)
```

Now we're ready to solve our cubic equation:

$6x^3 + 31x^2 + 3x - 10 = 0$

We'll add this function after the draw function:

```
def f(x):
    return 6*x**3 + 31*x**2 + 3*x – 10
```

It defines the function we're calling f. We're telling Python what to do with a number x to produce the output of the function.

Next we'll draw lines from every point to every "next" point, going up a tenth of a unit at a time. Even if our function produces a curve, you probably won't notice if we're drawing a straight lines between two points that are really close together. For example, the distance from (2, f(2)) to (2.1, f(2.1)) is tiny, so overall the output will look curved. We'll define a function to draw a graph of f(x) by starting at xmin and going all the way up to xmax. While the x-value is less than xmax, we'll draw a line from (x, f(x)) to ((x + 0.1), f(x + 0.1)). We can't forget to increment x by 0.1 at the end of the loop.

```
def graphFunction():
    x = xmin
    while x <= xmax:
        stroke(255,0,0) #red function
        line(x*xscl,f(x)*yscl,(x+0.1)*xscl,f(x+0.1)*yscl)
        x += 0.1
```

Finally, call the "`graphFunction`" function in `draw` and you'll see the output in Figure 4.8:

```
def draw():
    global xscl, yscl
    background(255) #white
    translate(width/2,height/2)
    grid(xscl,yscl)
    graphFunction()
```

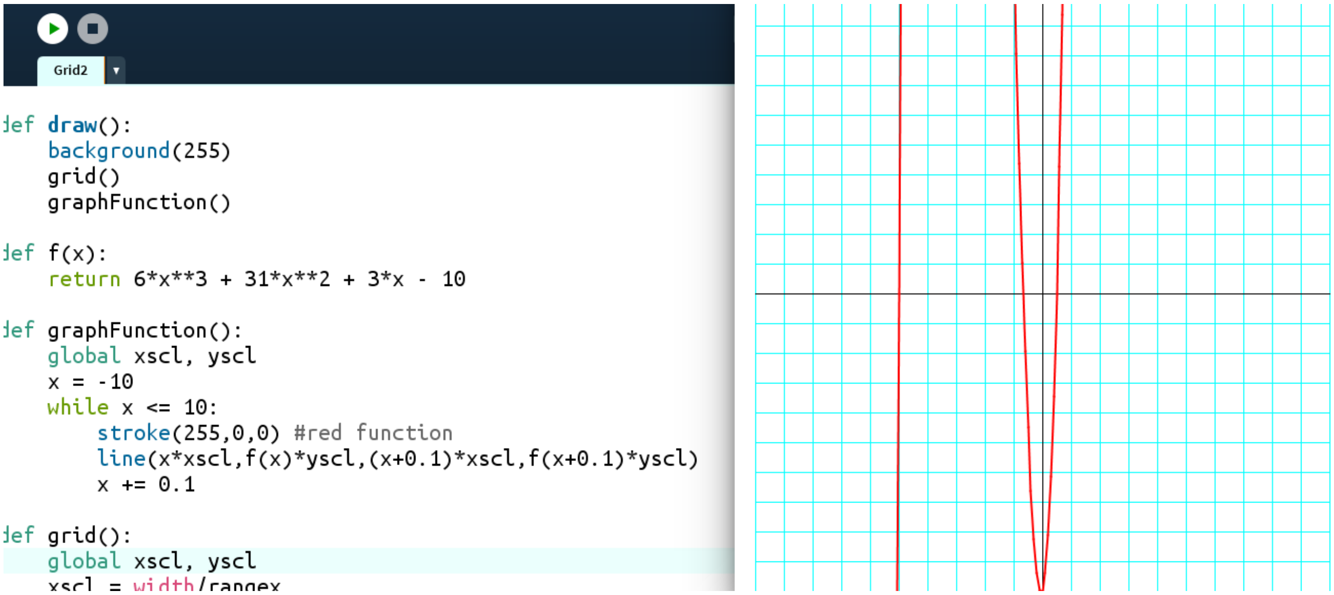

Figure 34.8: Graphing a polynomial function

The solutions (called the “roots”) of the equation are where the graph crosses the x-axis. We can see three places: one where x = -5, another where x is between -1 and 0, and a third where x is between 0 and 1.

Exercise 3-4: Back to the Quad

Use this grapher to check the solutions to the quadratic equations you solved, like $2x^2 + 7x – 15 = 0$.

Solution:

Change f(x) to $2x^2 + 7x – 15$ and find out where the graph crosses the x-axis. That’s where the function equals 0.

```
def f(x):
    return 2*x**2 + 7*x – 15
```

Run it and you’ll see the solutions you got algebraically: x = -5 and x = 1.5

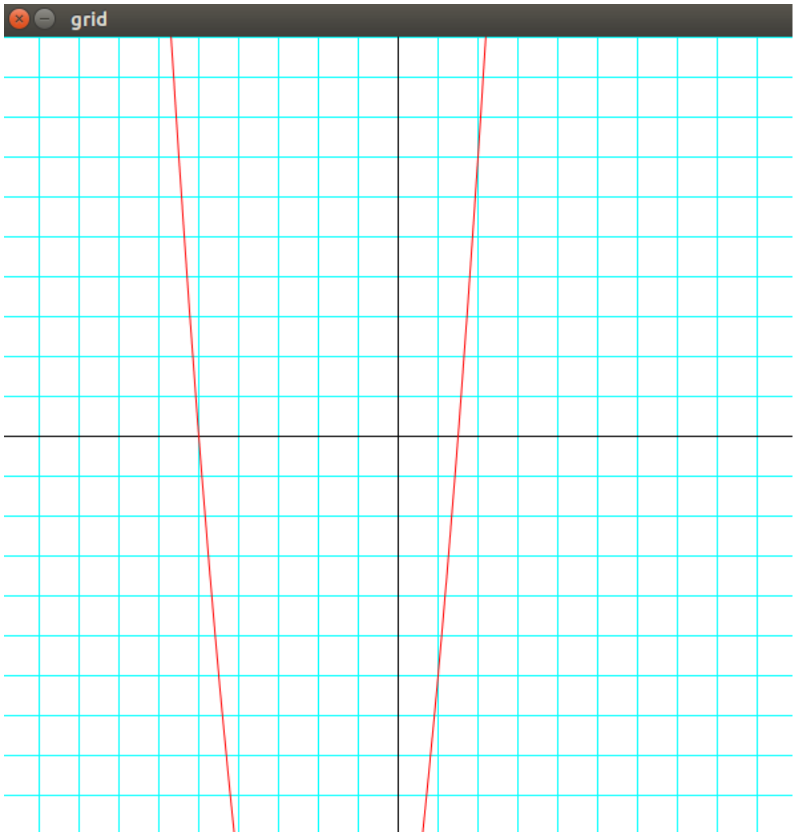

Figure 3-9: The solutions to a quadratic equation

The Halving Method

We already saw how effective our “Halving Method” was in the last chapter. Now we can use that method to approximate the roots. Let's start with the root between 0 and 1. Is it 0.5? We can easily plug in 0.5 and see. We'll move to IDLE for this.

```
def f(x):
    return 6*x**3 + 31*x**2 + 3*x – 10

>>> f(0.5)
0.0
```

Yes! x = 0.5 makes the function 0, so another solution of our equation is 0.5. Finally, we'll try the root between -1 and 0. We’ll try the average of -1 and 0:

```
>>> f(-0.5)
-4.5
```

At x = -0.5, the function is negative. Looking at the graph, ~~now~~ we can tell we guessed too high, so the root is somewhere between -1 and -0.5. We'll average those endpoints and try again:

```
>>> f(-0.75)
2.65625
```

That's positive, so we guessed too low: the solution is between -0.75 and -0.5.

```
>>> f(-0.625)
-1.23046875
```

Too high. Do you see how we might use Python to do these steps for us?

Exercise 3.5 Halving Your Roots Done

Create a function that will use the “halving method” to check for roots to the polynomial between a lower and upper value and adjust its “guesses” accordingly.

Solution:

First we’ll need to declare the function we’re looking for the equation we’re trying to solve. That’s f(x). Then we’ll need the function to average two numbers, since we’ll do that every step. Finally, we’ll write a “halving” function that will start with a lower limit of -1 and and upper limit of 0. We’ll create a loop so we’ll “half” the range 20 times. Our “guess” will be the average, or midpoint, of the upper and lower limits. We’ll put that midpoint into our function and if the output equals 0, we’ll know that’s our root. If the output is negative, we’ll know we guessed too high, like in our first guess above. The midpoint will replace our upper limit and we’ll take another guess. Otherwise, we guessed too low and the midpoint will become our lower limit and we’ll guess again.

If we haven’t returned the solution in 20 guesses we’ll return the latest midpoint and the function of that midpoint. Here’s what the code looks like:

```
'''The halving method'''
def f(x):
    return 6*x**3 + 31*x**2 + 3*x - 10

def average(a,b):
    return (a + b)/2.0

def halving():
    lower = -1
    upper = 0
    for i in range(20):
        midpt = average(lower,upper)
        if f(midpt) == 0:
```

```
            return midpt
        elif f(midpt) < 0:
            upper = midpt
        else:
            lower = midpt
    return midpt

x = halving()

print(x, f(x))
```

When we run this the output is

```
-0.6666669845581055 9.642708896251406e-06
```

x looks like it's around -2/3 and `f(x)` is close to 0. The “e-06” at the end means scientific notation. You take 9.64 and move the decimal place to the left 6 places. That number is 0.00000964. If we increase the number of iterations from 20 to 40, we'll get a number even closer to 0:

```
-0.6666666666669698 9.196199357575097e-12
```

Let's check f(-2/3):

```
>>> f(-2/3)
0.0
```

The three solutions to the equation

$$6x^3 + 31x^2 + 3x - 10 = 0$$

are x = -5, -2/3 and ½.

Algebra Has Been Transformed!

All we have to do to solve an equation, no matter how complicated, is graph it and approximate where it crosses the x-axis. By iterating and “halving” the range of values that work, we can get as accurate as we want. When we learn a Calculus trick called the derivative, we'll have another method to solve equations like this. And when we learn to use lists of lists, called **arrays** or **matrices**, we'll learn how to solve *systems* of equations, too!

Linear Functions

In math we often need to be able to deal with functions like y = 2x + 5, which are called "*linear functions*," because when you graph them, they make a straight line. The *slope*, or direction, from one point to the next is always the same in a line. The formula is

slope = rise / run

which means we calculate the slope of a line by dividing the vertical change between two points by the horizontal change between the same two points.. The slope is usually represented by the letter m

Exercise 3.6: Rising and Running

Write a program to find the slope of the line between two points. You should be able to enter any two points and it should return the slope as a number, like this:

```
>>> slope((1,2),(5,6))
1.0
>>> pt1 = (-2,5)
>>> pt2 = (10,-4)
>>> slope(pt1,pt2)
-0.75
```

Solution:

Using the points above, (1,2) and (5,6), you calculate the slope by dividing the rise by the run. We find the rise by subtracting the y-values: 6 – 2 = 4. If in Python we're using lists a and b to represent those points, then it's the second element in b minus the second element in a, or b[1] – a[1]. Remember, the first element in b would be b[0].

Then we find the run by subtracting the x-values in the same order as the y-values. So the run would be b[0] – a[0]. Finally we return the rise divided by the run.

```
def slope(a,b):
    '''Returns the slope of the line
    between points a and b'''
    #find the difference between the
    #y-values:
    rise = a[1] - b[1]
    #the difference between the
    #x-values:
    run = a[0] - b[0]
    return rise/run
```

Exercise 3.7: A Tale of Two Points:

Remember the equation of a line is y = mx + b, where m is the slope and b is the y-intercept, where the graphed line crosses the vertical y-axis. Write a function (you can use your slope function!) that will return the slope and y-intercept of a line between two given points. It should return values like this:

```
>>> line2pts((1,2),(5,6))
(1.0, 1.0)
>>> pt1 = (-2,5)
>>> pt2 = (10,-4)
>>> line2pts(pt1,pt2)
(-0.75, 3.5)
```

Solution:

It's not so hard to solve y = mx + b for b, the y-intercept of the line:

$$b = y - mx$$

y is the second element in one of the points, and x is the first element in the point list. The slope is the output of the slope function we just wrote. So we only have to plug y, m and x into the equation above to find the y-intercept, b:

```
def line2pts(a,b):
    '''Returns the slope and y-intercept
    of the line between points a and b'''
    #find the slope:
    rise = a[1] - b[1]
    run = a[0] - b[0]
    slope = rise/run
    #y-intercept: b = y - mx
    yint = a[1] - slope*a[0]
    return slope, yint
```

Exercise 3.8: Making connections

Write a program that will find the intersection point of two lines, given their slope and y-intercept. For example, if I want the intersection of the lines y = 2x + 5 and y = -3x – 10, I can just enter the numbers like this:

```
>>> intersection([2,5],[-3,-10])
(-3.0, -1.0)
```

Solution

If the y-value of the intersection point has to work in both equations y = 2x + 5 and y = -3x – 10, then the right sides must be equal:

2x + 5 = -3x – 10

But we can easily solve that with our "equation" function from earlier in this chapter! Use that to solve for the x-value of the intersection, then plug x into either line to find the y-value:

```
def intersection(line1, line2):
    '''returns the intersection point of two
    lines, given their slopes and y-intercepts'''
    #find the x-value of the intersection
    x = equation(line1[0],line1[1],line2[0],line2[1])
    #find the y-value by plugging it in to either line
    y = line1[0]*x + line1[1]
    return x,y
```

To check, the graph of the lines and their intersection is in Figure 4.13:

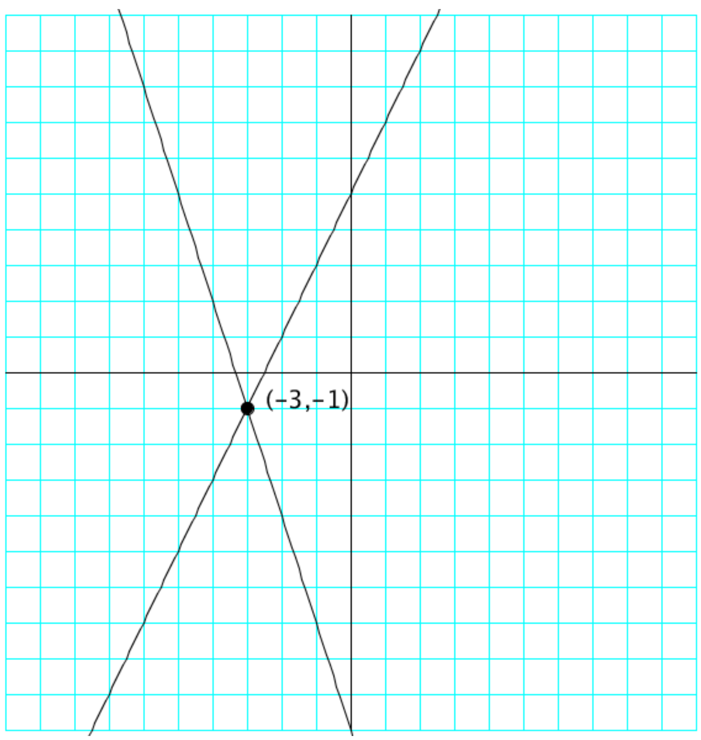

Figure 3-109: The intersection of two lines.

Interactive Lines

Let's use the mouse to make a number change in a line. We'll use the graphing program we made earlier but use the mouse location to change the slope and y-intercept. There's nothing else you can change in a line, is there?

Change f(x) to the general form of a line f(x) = m*x + b:

```
def f(x,m,b):
    return m*x + b
```

In the draw function, just before graphing the function, create a slope function to be the mouse's x-coordinate, and the y-intercept to be the mouse's y-intercept.

```
def draw():
    global xscl, yscl
    background(255) #white
    translate(width/2,height/2)
    grid(xscl,yscl)
    m = mouseX
    b = mouseY
    graphFunction(m,b)
```

And we'll have to change the graphFunction function to this:

```
def graphFunction(m,b):
    x = xmin
    while x <= xmax:
        stroke(255,0,0) #red function
        line(x*xscl,f(x,m,b)*yscl,(x+0.1)*xscl,f(x+0.1,m,b)*yscl)
        x += 0.1
```

But when you run this it doesn't change the slope or y-intercept very dynamically. That's because they're taking the value of the mouse's x- and y-coordinates, which go from 0 to 600. We'd like our slope and y-intercept to go from, for example, -10 to 10. We could use xscl and yscl but there's a function in Processing which allows us to map a variable from one range to another. It'll make it easier to see the slope and y-intercept.

We have to tell Processing which variable we're mapping, in this case mouseX or mouseY. Then we tell it the old range, in this case 0 to 600, and then the new range: -10 to 10. This is how it should look:

```
    m = map(mouseX, 0, 600, -10,10)
    b = map(mouseY, 0, 600, 10,-10)
```

Now you can run this, move the mouse around and see the slope and y-intercept of the line change dynamically!

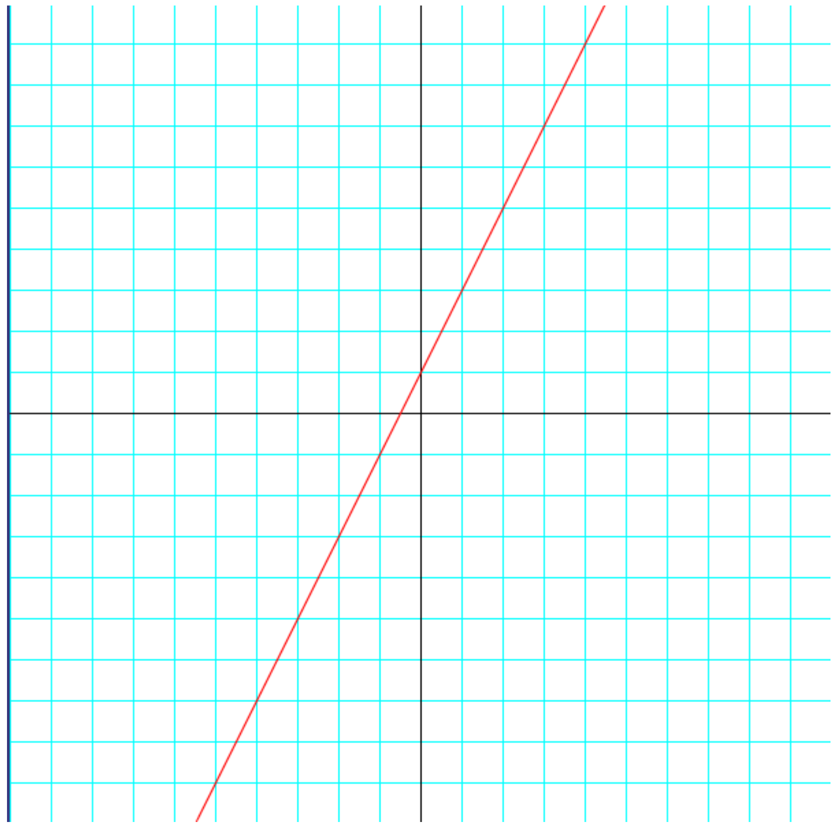

Figure 3-11~~0~~ Our dynamic line!

Adding Text

Finally, let's add some text to our sketch so we can display the exact equation of the line and update it as we move the mouse around. To add text, all you have to do is use Processing's "text" function in the draw function. You tell it what to write, and where to write it. So you need a string, an x-coordinate and a y-coordinate.

```
grid(xscl,yscl)
m=map(mouseX, 0, 600, -10,10)
b=map(mouseY, 0, 600, 10,-10)
graphFunction(m,b)
fill(0)#black text
textSize(48)
text("Hello!",-250,-250)
```

That'll look like this:

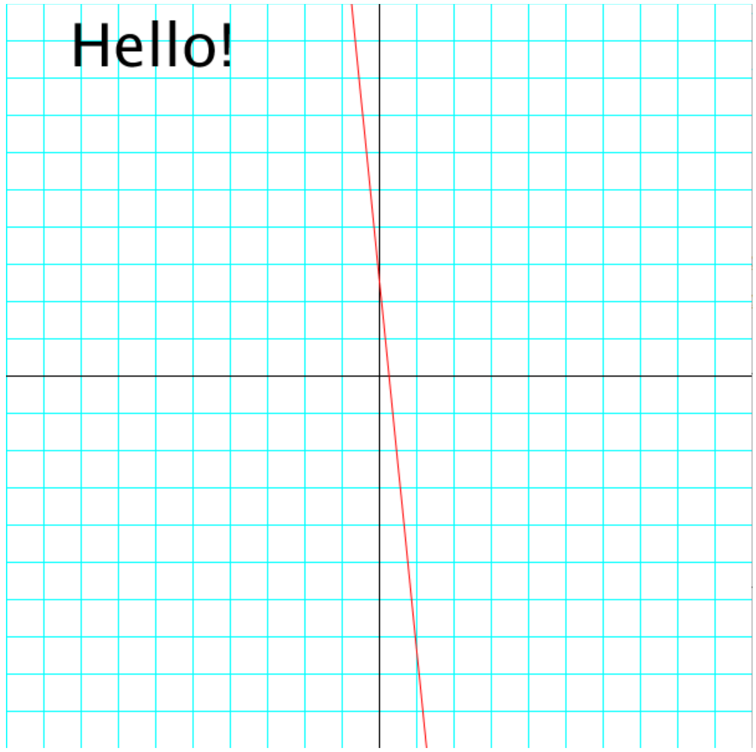

Figure 3-12~~1~~: Writing text on our screen

We’ll just replace the string “Hello” with the string of values for the equation of our line. We can write “y = mx + b” but we need to replace m and b with their values. Processing wants the text to be a string, so we have to convert our numbers m and b to strings. But if you ran this code you’d get some long decimals:

```
text("y="+str(m)+"x+"+str(b),-250,-250)
```

So we’ll use the Python code for rounding numbers. There’s a function called “round” and you tell it how many decimal places you want:

```
>>> number = 0.123456789
>>> round(number,1)
0.1
>>> round(number,4)
0.1235
```

So our text function can look like this so we’ll only get 1 decimal place:

```
text("y="+str(round(m,1))+"x+"+str(round(b,1)),-250,-250)
```

Now we can see the equation of all the lines we make!

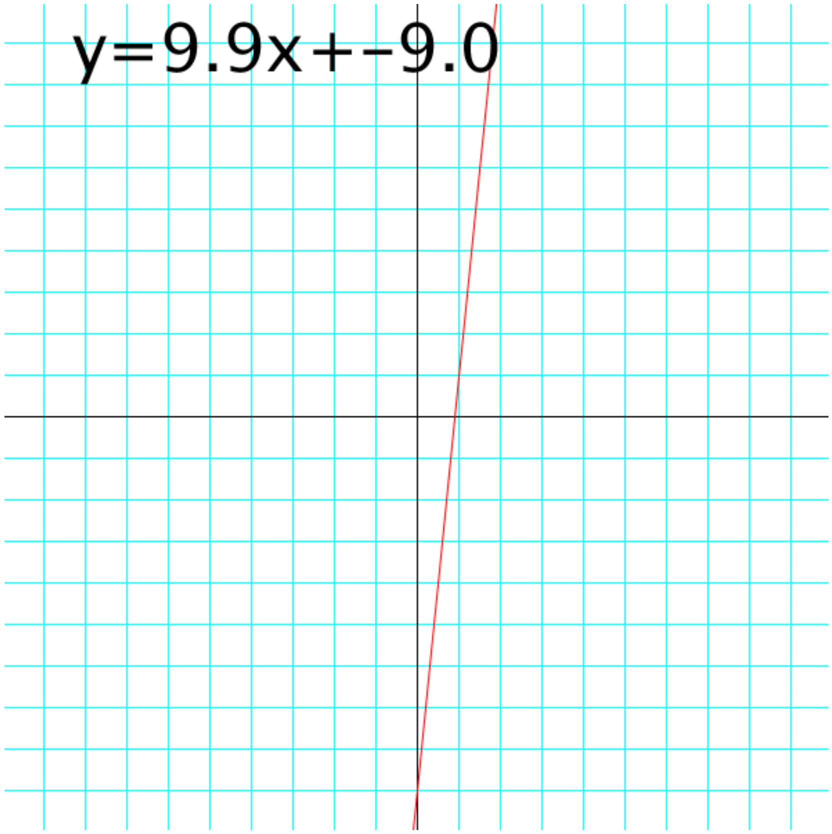

Figure 3-13: The equation of the line

In Conclusion…

We've created a lot of tools in this chapter:

Plug and Check

Equation Solver

Quadratic Formula

Grid for Graphing

Graphing a Function

Find Roots by Halving

Find Line Through Two Points

Intersection of Two Lines

Mapping Variables

Using the Mouse Location

Writing Text on the Screen

Rounding Numbers

Now we're going to use these to build even more powerful tools!