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*":

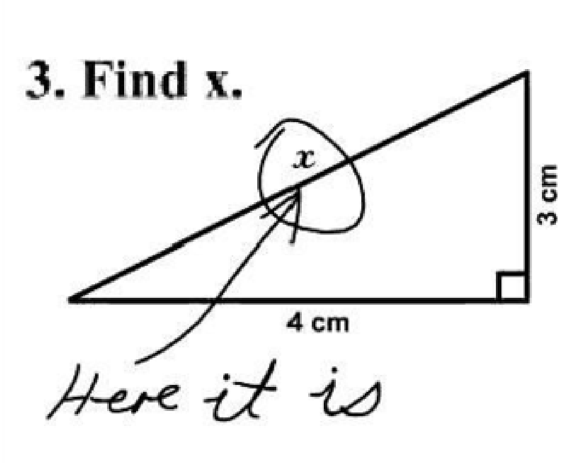


Figure 3-1: Locating the x variable instead of solving for its value.

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 an integer, you might have to increment by smaller numbers by changing the line at ❸ to~~, for example,~~ `x += .25` or some other decimal~~.~~.

Solving First-degree ~~Polynomials~~Equations

Another way to solve an equation like $2x + 5 = 13$ is to find a general formula for the 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* ~~*polynomial*~~*equation*, because the highest exponent a variable has is 1. And a number to the 1 power is just the number itself.~~They one term whose highest exponent is one. more thanwhich refers to an equation that has~~ 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*~~*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 ~~find the general form of~~ 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 ~~polynomial~~equation and all four ~~coefficients~~numbers (a, b, c and d) are known. Now let's write this into a Python program that can solve algebraic equations of thos form for us.

Using Coefficients 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 coefficients 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: ~~to make sure it works~~

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

It works!~~As expected~~, ~~w~~We get 4 as the solution for the equation.~~e x variable~~.

Exercise 3.1: X-Games

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

~~Using Return Instead of Print~~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 ~~in another equation~~ 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 ~~works out~~ calculates *x* for us, and assigns the result ~~-~~to the variable *x*. Then we can see the value of *x* by simply entering it. ~~that result ind"store"e W .returns the value satisfies the equation 12*x* + 18 = -34*x* + 67 and ot ,~~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?

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. But they can all be expressed in that form. To solve an equation with a squared term, you can use the Quadratic Formula:

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

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. Like $x^2 + 3x - 10 = 0$. 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 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:~~write a Python program that uses the quadratic formula to return the solutions of a quadratic equation, which w Let's~~

~~rQuadratic Formula Solve AutomaticMaking the~~

We'll call the function `quad()`. Before we do ~~this~~ anything, ~~you~~we'll need to import the `sqrt` method from the `math` module. The `sqrt` method allows you to find the square root of ~~zerobelow that , not not~~

~~onevaluepositive nya~~ 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 what 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 ~~XXXXXexpressionsreturns the solutions for a quadratic equation that has the form $ax^2 + bx + c = 0$. Because there are two solutions to this type of equation, we need to write two~~ 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 $2x^2 + 7x – 15 = 0$ ~~$x^2 + 3x – 10 = 0$~~. 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.
```

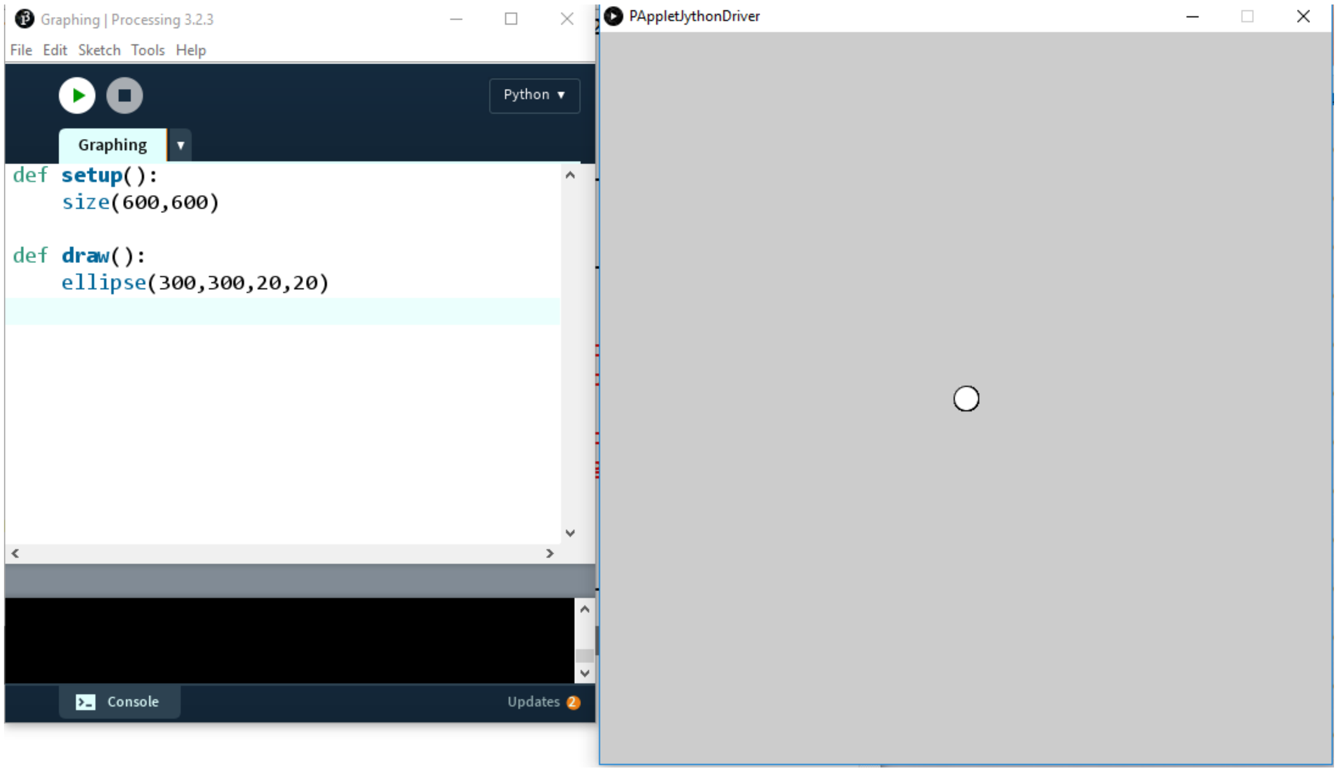
That gives you -5 as the solution, but as you might suspect from the term x^3, there ~~are actually~~ could be as many as three possible solutions to this equation. Fortunately, there's a way to see all the possible inputs and corresponding outputs of a function; it's called *graphing.*

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:**



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

def draw():
    ellipse(300,300,20,20)
```

Figure ~~3~~4~~-~~.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:

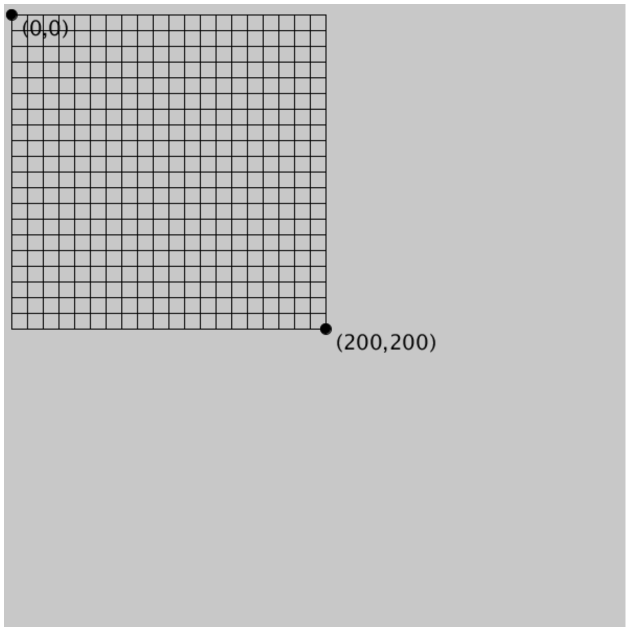


Figure 4.3

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:

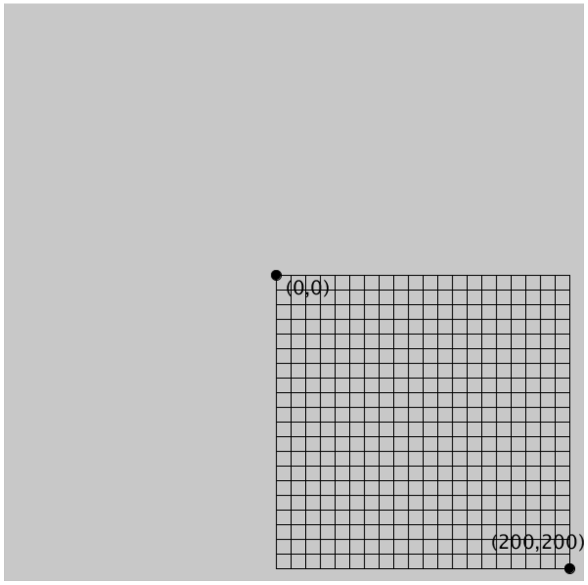


Figure 4.4

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 ~~set them both to 21 here, because the total number of values between -10 and 10 is 21, including zero.~~ 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 ~~pixels~~units by 600 ~~pixels~~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 ~~create~~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! ~~th (600) by `rangex` (21) and assign that value to `xscl`. We do the same for the y-axis by dividing `height` by `rangey` (21) and assigning the value to the `yscl` variable.divide the wid~~

Drawing a Grid

Now we'll draw in lines for the grid like on a graph paper. All the drawing will go in a 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 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 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. 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 (or you might not!) 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)
```

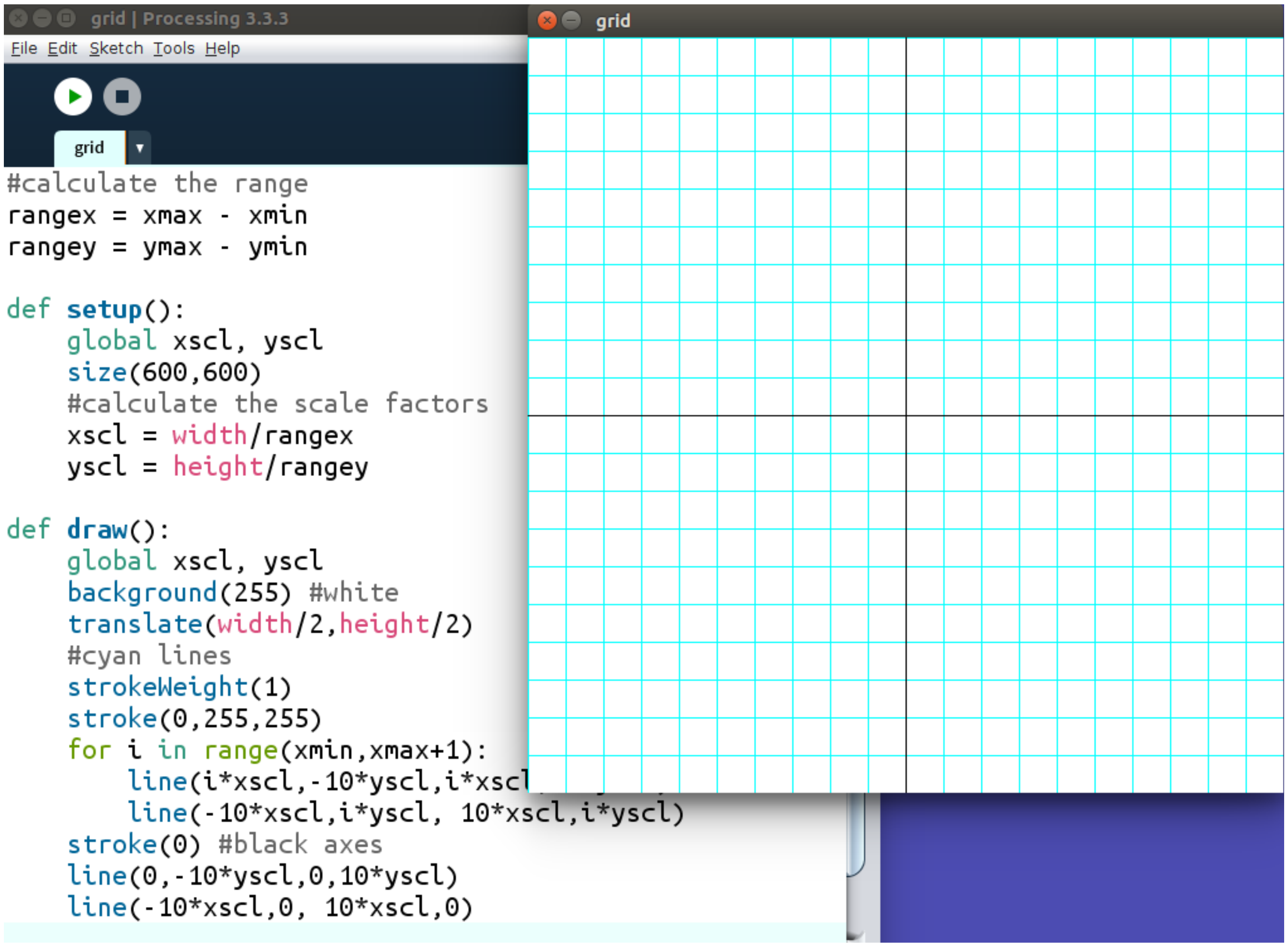
~~In this example, we use the `line()` function to draw the grid on which we'll draw our axes. The `line()` function takes four parameters using the coordinates of two points: (x-coordinate of the first point, y-coordinate of the first point, x-coordinate of the second point, y-coordinate of the second point). Using a `for` loop, we~~ ~~,~~Now we'll create the axes

Draw two black lines for the x- and y-axes.

Solution:

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

~~Add "`set the background white. to  the beginning of the draw function to "background(255)`~~The~~at~~ above code gives us a nice grid, like in Figure 4.5:



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

def setup():
    global xscl, yscl
    size(600,600)
    #calculate the scale factors
    xscl = width/rangex
    yscl = height/rangey

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
        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)
```

Figure 4.5: Creating a grid for graphing. You only have to do it once!

This looks done, but if we try to put a point `, actually) ellipsean(` 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 4.6:

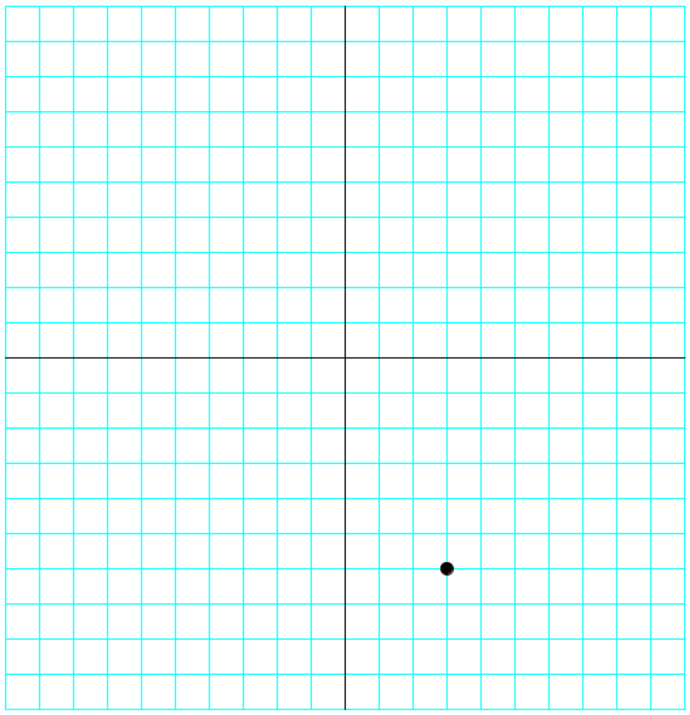


Figure 4.6: Checking our graphing program. Almost there!

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

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

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

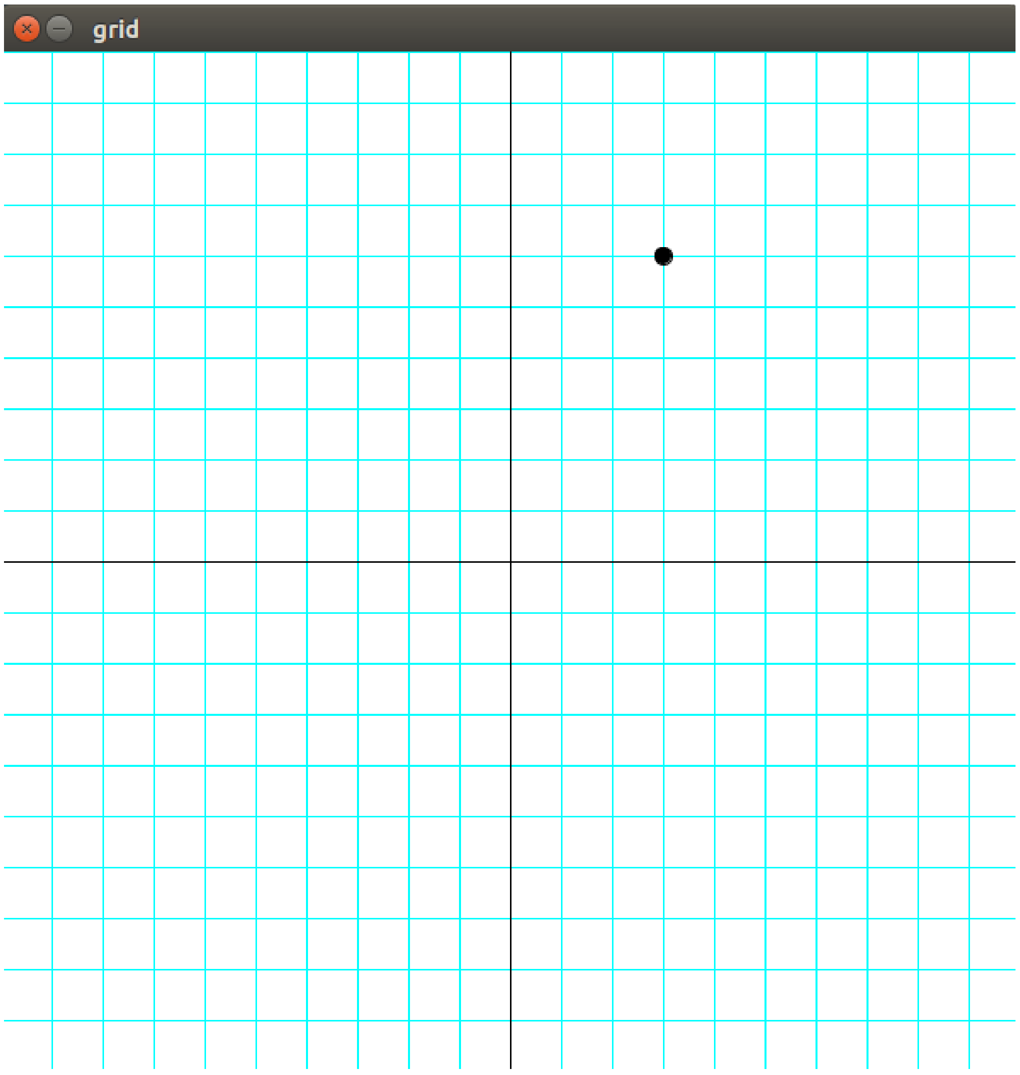


Figure 4.7: The grapher is working!

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 equation:

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

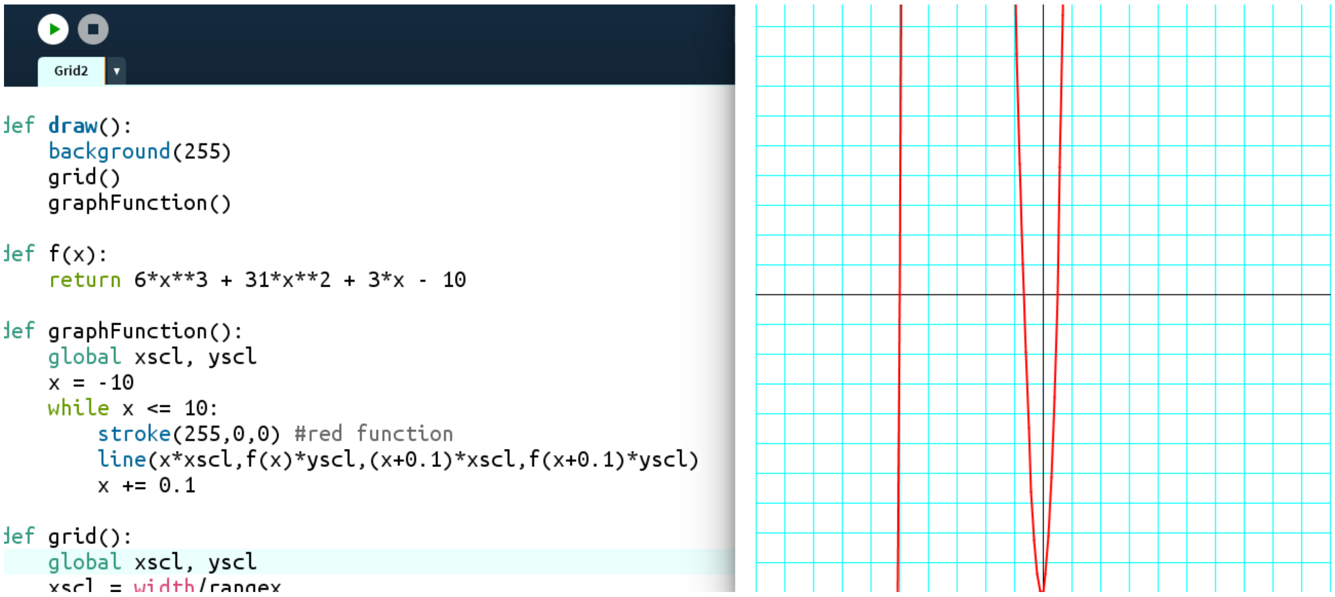
We'll add this function

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

draw lines from every point to every “next” point, going up a tenth of a unit at a time andto our program:

```
def graphFunction():
    x = -10
    while x <= 10:
        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 “`graphFunction”` in `draw and you'll seefunction` the output in Figure 4.8:



```
def draw():
    background(255)
    grid()
    graphFunction()

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

def graphFunction():
    global xscl, yscl
    x = -10
    while x <= 10:
        stroke(255,0,0) #red function
        line(x*xscl,f(x)*yscl,(x+0.1)*xscl,f(x+0.1)*yscl)
        x += 0.1

def grid():
    global xscl, yscl
    xscl = width/rangex
```

Figure 4.8: Graphing a polynomial function

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

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
```

Another solution to our equation is 0.5. Finally, we'll try the root between -1 and 0.

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

Looking at the graph, now we can tell we guessed too high: 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 4.5 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:

```
'''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))
```

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. In decimal terms, 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 4.6: Rising and Running

Write a program to find the slope of the line between two points.

Solution:

```
>>> slope((1,2),(5,6))
1.0
>>> pt1 = (-2,5)
>>> pt2 = (10,-4)
>>> slope(pt1,pt2)
-0.75You should be able to enter any two points and it should return the slope as a
number, like this:
```

```
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
```

It's not so hard to solve this equation for b:

b = y – mxRemember the equation of the line is y = mx + b, where m is the slope and b is the y-intercept. The other half of the linear function 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

Exercise 4.7: A Tale of Two Points:

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

Solution:

```
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
```

The equation of a line is y = mx + b, meaning all the x- and y-values of every point in the line are connected by the same formula. In y = 2x + 5, you multiply the x-value by 2 and add 5, and you get the y-value of that point. For example, for x = 3, the y-value of the point on the line y = 2x + 5 where the x-value is 3 is 2(3) + 5 = 11. The point on that line whose x-value is 3 is (3, 11). Let's graph it and see.

Lists of Points

Remember when we learned about variables? They were the way to save one value:

```
x = 5
playing = True
name = “Peter”
```

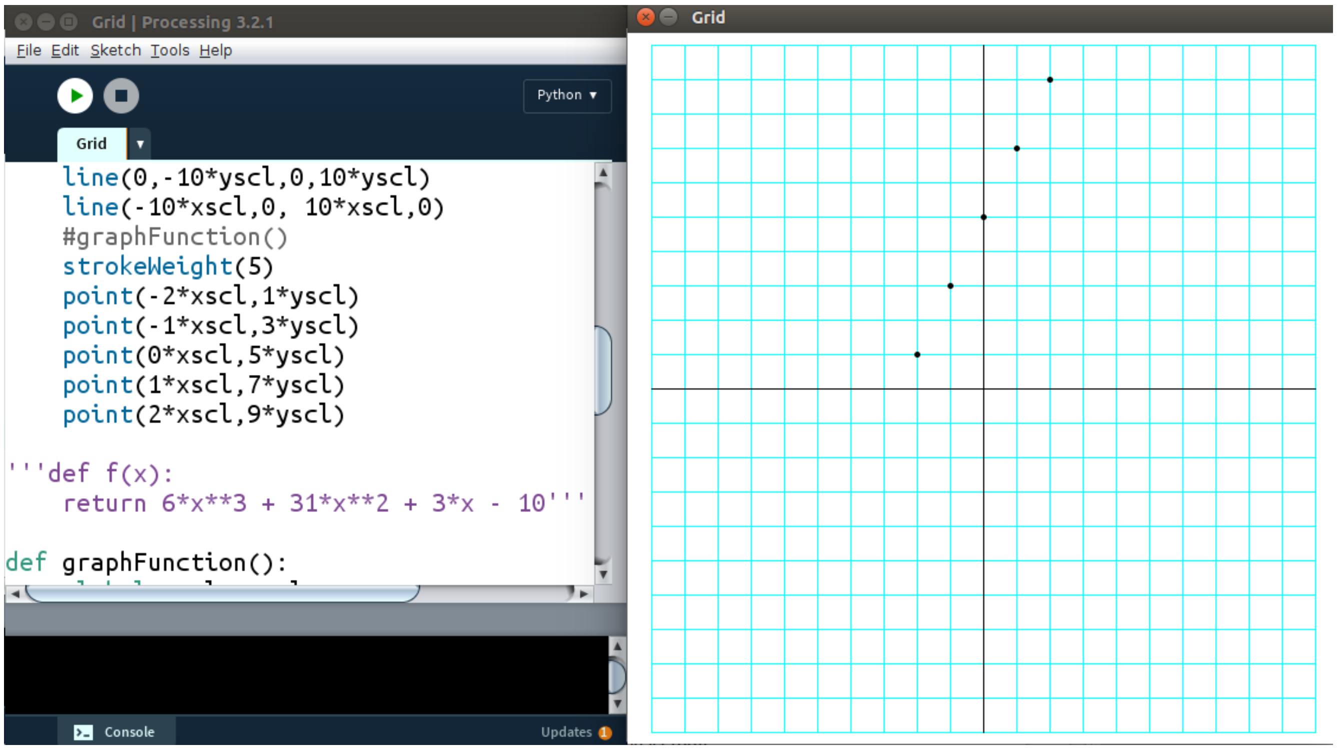
Then we wanted to save more than one number, so we learned to make lists:

```
myList = [2,4,6,8]
```

What shapes really do is **store a bunch of values,** usually points. When those points are put on a screen, it looks like an ellipse or a rectangle. Even the equation of a line or curve is a way to link the x-values and y-values of all the points that make up that shape.

y = 2x + 5

Obviously we can think of a bunch of points that work in this equation: (0,5), (1,7) and so on. If we were to use our graphing tool and put those points on the grid, we'd see the shape that equation represents, as in Figure 4.9:



```
    line(0,-10*yscl,0,10*yscl)
    line(-10*xscl,0, 10*xscl,0)
    #graphFunction()
    strokeWeight(5)
    point(-2*xscl,1*yscl)
    point(-1*xscl,3*yscl)
    point(0*xscl,5*yscl)
    point(1*xscl,7*yscl)
    point(2*xscl,9*yscl)

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

def graphFunction():
```

Figure 4.9: Plotting points one by one.

That's very time consuming and there is an easier way. We just need a list of points and we can put them in the code automatically. Here's a function to make a list full of points from that function:

```
a = list() #or a = []
for i in range(-10,11):
    a.append([i*xscl,(2*i+5)*yscl])
```

The list looks like this:

```
[[-10, -15], [-9, -13], [-8, -11], [-7, -9], [-6, -7], [-5, -5], [-4, -3],
[-3, -1], [-2, 1], [-1, 3], [0, 5], [1, 7], [2, 9], [3, 11], [4, 13], [5, 15],
[6, 17], [7, 19], [8, 21], [9, 23], [10, 25]]
```

List Comprehensions

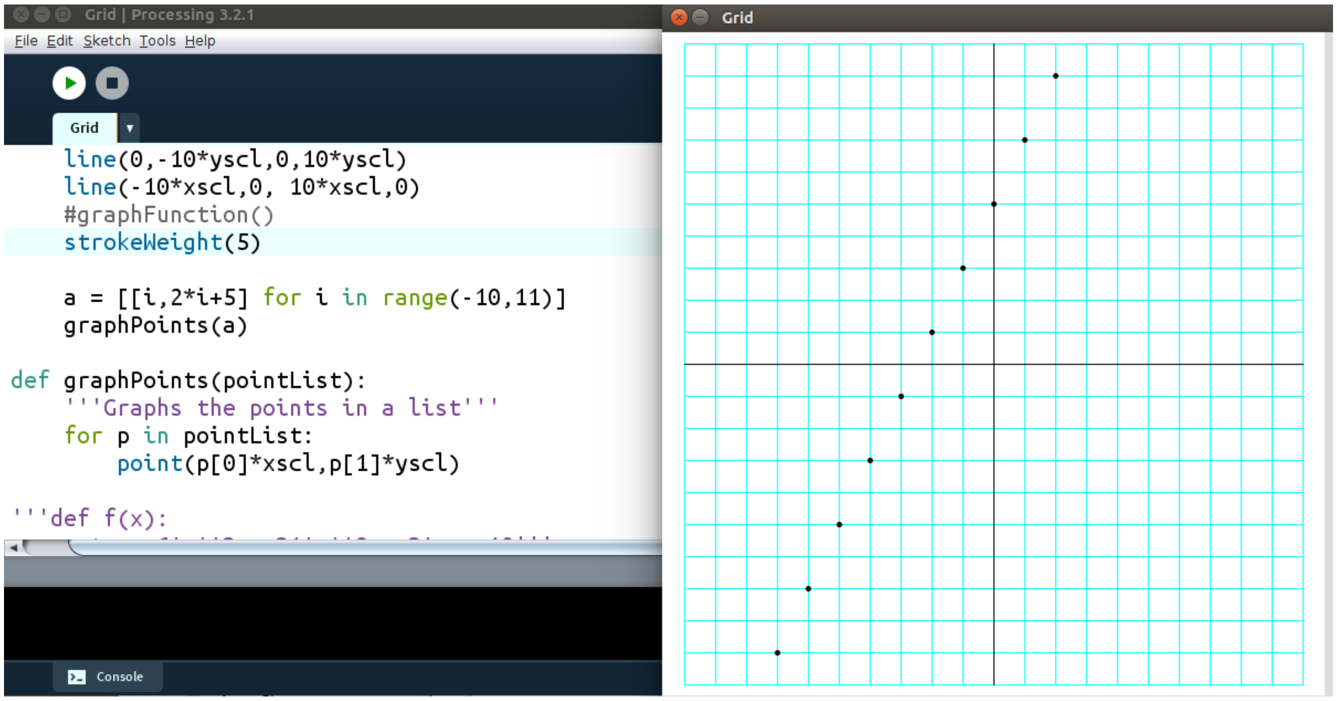
Python has a way to create lists in a compact form. Instead of the three lines above, you can do it all in one line:

```
a = [[i,2*i+5] for i in range(-10,11)]
```

Here's the function for graphing points, given a list:

```
def graphPoints(pointList):
    '''Graphs the points in a list'''
    for p in pointList:
        point(p[0]*xscl,p[1]*yscl)
```

However, you might remember “range” only contains integers. That means when we graph the points in list `a`, we'll see what's in Figure 4.10:



```
    line(0,-10*yscl,0,10*yscl)
    line(-10*xscl,0, 10*xscl,0)
    #graphFunction()
    strokeWeight(5)

    a = [[i,2*i+5] for i in range(-10,11)]
    graphPoints(a)

def graphPoints(pointList):
    '''Graphs the points in a list'''
    for p in pointList:
        point(p[0]*xscl,p[1]*yscl)

'''def f(x):
```

Figure 4.10: The points generated by the list comprehension.

Python has a terrific numerical package called Numpy which contains "`arange,`" which can go up by decimals. However, you can't import Numpy into Processing yet. Let's create our own "`arange`" function:

Exercise 4.8: Home on arange

Create a function called "`arange`" that creates a list. It should start at a given "start" value, go up by a given "step" value and put all the values into the list until it reaches a "stop" number, like this, which means "all the numbers from 1 to 3, going up by 0.25." Like "range," it doesn't contain the "stop" value.

```
>>> print(arange(1,3,0.25))
[1, 1.25, 1.5, 1.75, 2.0, 2.25, 2.5, 2.75]
```

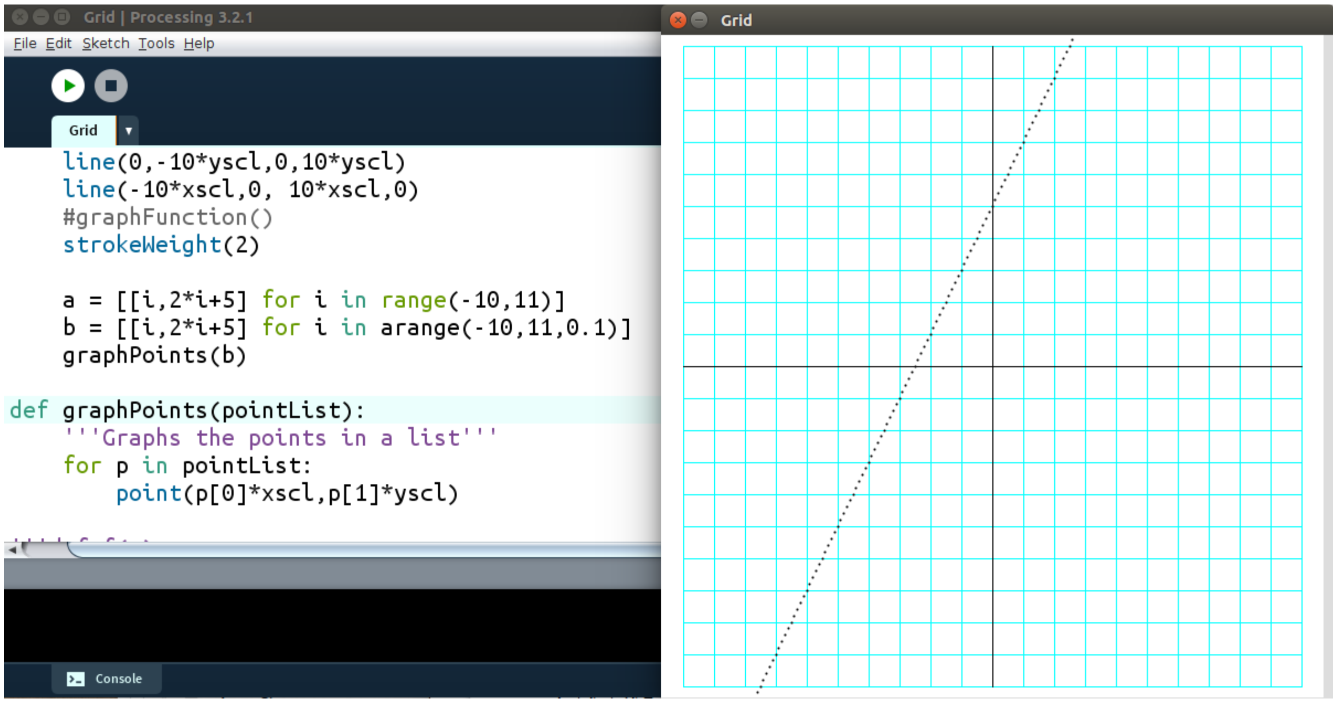
Solution:

```
def arange(start,stop,step):
    output = []
    x = start
    while x < stop:
        output.append(x)
        x += step
    return output
```

Once we put that `in our Processing sketch`, we can generate a list of points using `arange`:

```
b = [[i,2*i+5] for i in arange(-10,11,0.1)]
```

and graph the points in that list. The output looks like figure 4.11:



```
    line(0,-10*yscl,0,10*yscl)
    line(-10*xscl,0, 10*xscl,0)
    #graphFunction()
    strokeWeight(2)

    a = [[i,2*i+5] for i in range(-10,11)]
    b = [[i,2*i+5] for i in arange(-10,11,0.1)]
    graphPoints(b)

def graphPoints(pointList):
    '''Graphs the points in a list'''
    for p in pointList:
        point(p[0]*xscl,p[1]*yscl)
```

Figure 4.11: Graphing a function by generating the points

We could get a better looking graph by connecting the dots with lines:

```
def graphPoints2(pointList):
    '''Graphs the points in a list using segments'''
    for i,p in enumerate(pointList):
        if i<len(pointList)-1:
            line(p[0]*xscl,p[1]*yscl,
                 pointList[i+1][0]*xscl,pointList[i+1][1]*yscl)
```

This gets us the output in Figure 4.12:

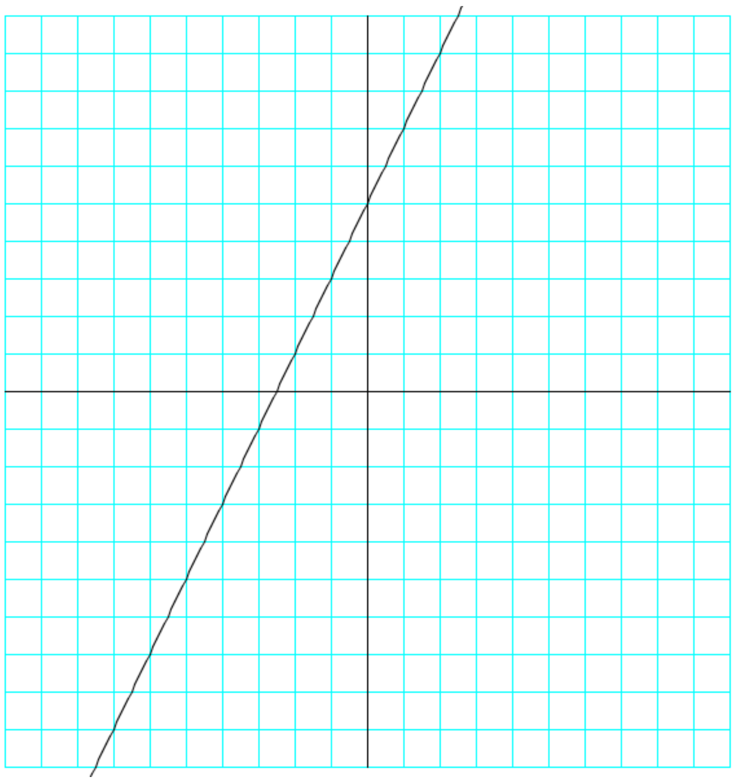


Figure 4.12

Exercise 4.9: 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 beequal:

2x + 5 = -3x – 10

But we can easily solve that with our “equation” function from earlier in this chapter! Use thtat 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:

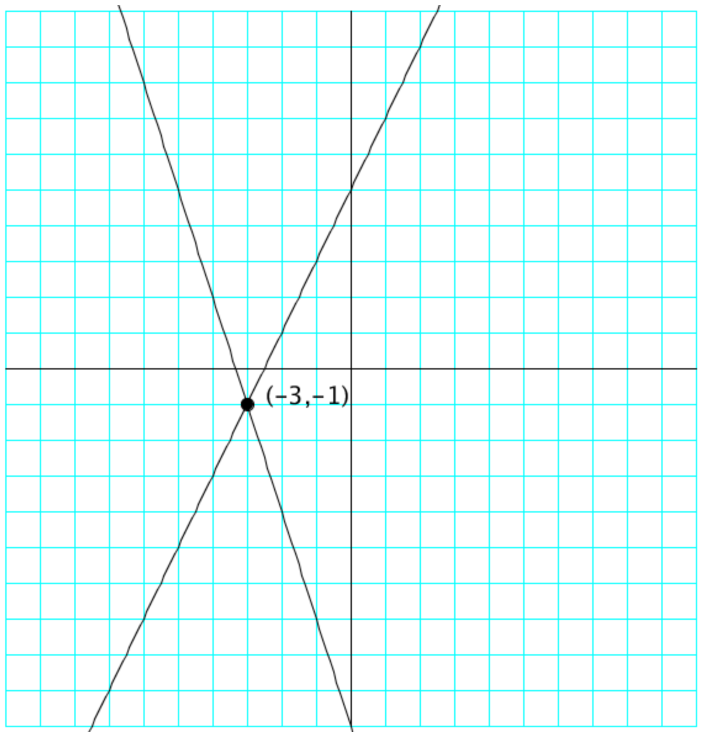


Figure 4.13: The intersection of two lines.

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

List Comprehensions

arange

PointList

Graphing from a Point List

Find Line Through Two Points

Intersection of Two Lines

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