4 [ChapterStart]

Algebra: Transforming and Storing Numbers

Algebra is the idea of replacing numbers with letters. Instead of 2·3 you have 2*x* or 2*y*, for example, meaning “2 times some unknown number.” We've already used variables to replace numbers, and it's a vital tool in programming.

Unfortunately, in traditional math class variables are often used to represent a “mystery number” that a student is required to find. Not “find” as in Figure 4.1:

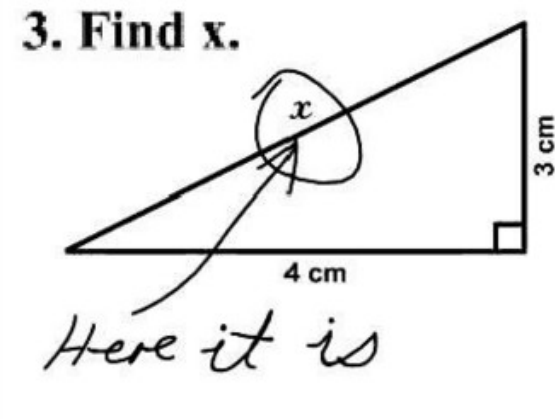


Figure 4.1: Math questions are subject to interpretation.

Every day in Algebra students are given an equation with an unknown (or two or three…) and they have to find what number or numbers make the equation true, like:

1. Solve 2*x* + 5 = 13.

“Solve” means “find out which number, when you replace *x* with that number, makes both sides of the equation equal.” Equations of this form don't take too much effort to solve, but even at this level we can use Python and “brute force” to plug in random numbers.

Exercise 4.1:

Write a program that will plug all the integers between -100 and 100 into the above equation and if one works, print out the number.

Solution:

```
def plug():
    x = -100 #start at -100
    while x < 100: #go up to 100
        if 2*x + 5 == 13: #if it makes the equation true
            print("x =",x) #print it out
```

```
        x += 1 #make x go up by 1

plug() #run the plug function
```

The output is

```
x = 4
```

This is a valid way to solve a problem! It's too laborious for a human to do by hand, but using a computer it's a cinch. If the solution isn't an integer, you might have to increment by smaller numbers.

All “first-degree” equations, meaning those in which the highest exponent is one, fall into this pattern:

ax + b = cx + d

where a,b,c and d are numbers. Like these ones:

3*x* - 5 = 22

4*x* - 12 = 2*x* - 9

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

Sometimes there's no x term on one side, so the coefficient is zero. Using a little Algebra, you can solve the general form of the equation and that will help you solve all equations of that form.

a*x* + b = c*x* + d

Get all the *x*'s on one side of the equals sign:

a*x* - c*x* = d - b

Now factor out an *x*:

x(a - c) = d - b

Divide both sides by a – c and you have the value of *x*:

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

Exercise 4.2:

Write a program that will take the four coefficients of the general equation above and print out the solution.

Solution:

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

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

Let's test it with an equation we've solved already:

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

Now for a more complicated one:

$12x + 18 = -34x + 67$

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

How do we check it? We plug that solution in for x. But why type all those digits? Notice we didn't use “print” in the program, but we used “return” instead. This means we can assign the result to a variable:

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

Now find out what the left side of the equation becomes with that number for x:

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

Is it the same as the right side of the equation?

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

Except for some rounding error at the end, yes!

Higher Degree Equations

If you're faced with an equation with a squared term in it, like $ax^2 + bx + c = 0$, you should use the Quadratic Formula:

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

Exercise 4.3:

Write a program that will take a, b and c above and return the two solutions to the quadratic formula. Use it to solve $x^2 + 3x - 10 = 0$.

Solution:

Add this line to the top of your Python file to import the square root function:

```
from math import sqrt
```

Here's the 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

>>> quad(1,3,-10)
(2.0, -5.0)
```

Here's how we check the answers:

```
>>> 2**2 + 3*2 - 10
0
>>> (-5)**2 + 3*(-5) - 10
0
```

Factoring

Many problems require students to solve those quadratic equations (with an x^2 term) by factoring. Factoring means reducing one number into the product of two or more numbers, like 30 = (5)(6). That means $ax^2 + bx + c$ (if it's factorable) can be split into the form

$$(dx + e)(fx + g)$$

d and f are factors of a, and e and g are factors of c. All you have to do is go through all the factors of a and all the factors of c, and see if d*g + e*f equals b. It would be annoying for a human to go through every possible combination, but computers were made for such repetition. Remember the program you wrote that factored an integer? We need to adapt it to include the negative factors, too.

Exercise 4.4:

Write a program that will factor an integer and put all its factors (including negative ones!) into a list, like this:

```
>>> factor(6)
[1, -1, 2, -2, 3, -3, 6, -6]
```

Solution:

```
def factor(num):
    factors = [] #factor list
    num = abs(num) #use the absolute value
    for i in range(1,num+1):
        if num % i == 0:
            factors.append(i)
            factors.append(-i)
    return factors
```

Now all you have to do is go through all the factors, add up d*g + e*f and see which combination adds up to b.

Exercise 4.5:

Write a program that will take a, b and c and print out the factored form of the polynomial, if it is factorable.

Solution:

```
def factorPoly(a,b,c):
    afactors = factor(a) #get the factors of a
    cfactors = factor(c) #get the factors of c
    for afactor in afactors: #try all the factors of a
        for cfactor in cfactors: #and c
            #see which combination adds up to b
            if afactor * c/cfactor + a/afactor * cfactor == b:
                print("(",afactor,"x +",cfactor,")(",
                      int(a/afactor),"x +",int(c/cfactor),")")
                return
```

```
>>> factorPoly(6,-1,-15)
( 2 x + 3 )( 3 x + -5 )
```

Unfortunately, not all polynomials are factorable. But using the Quadratic Formula will always work, even if the solutions aren't integers, rational numbers or even real numbers!

Solving Higher Degree Equations

In Algebra class students are often asked to solve a higher degree equation, like

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

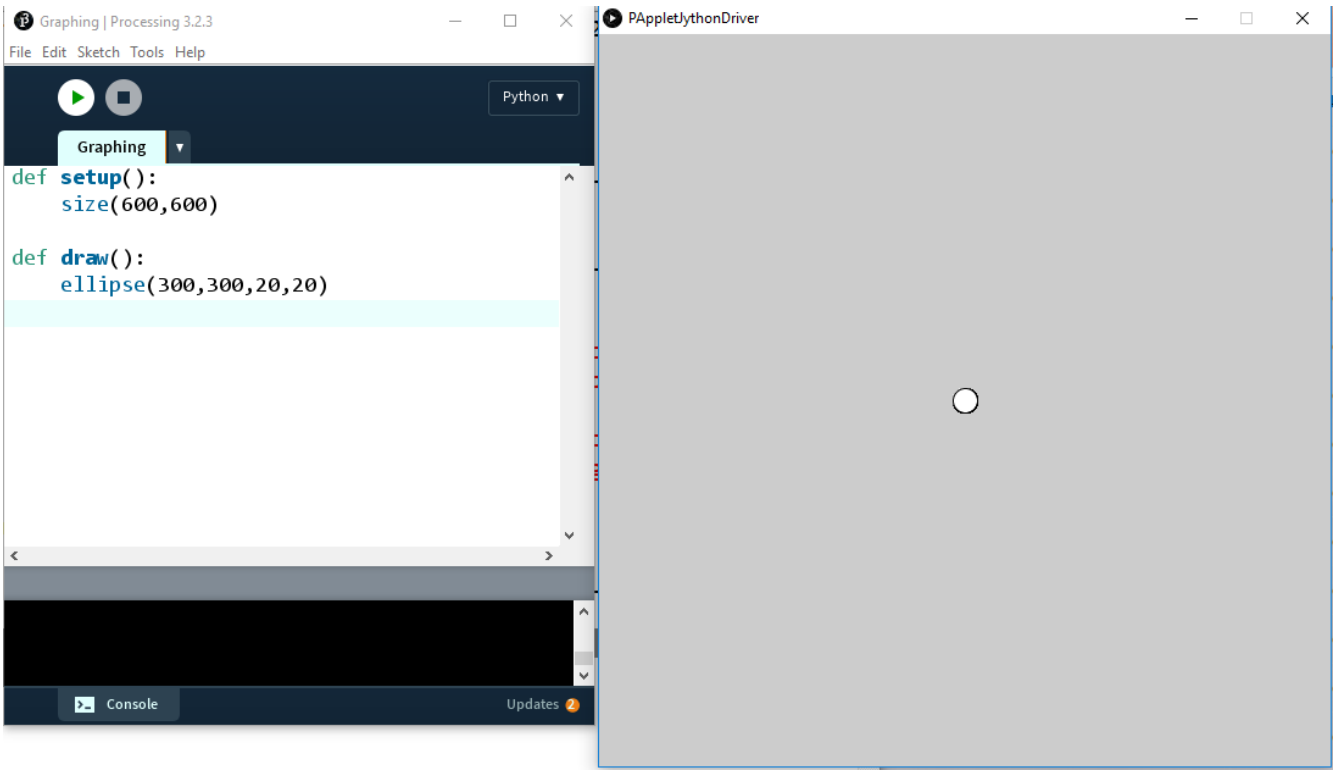
You can use the "plug" function and look for solutions manually:

```
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.")

>>> plug()
x = -5
done.
```

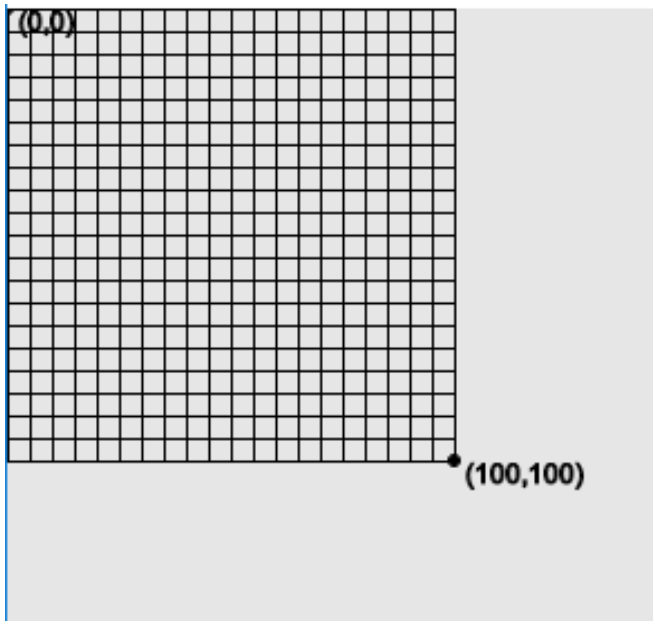
That gives you one of the solutions. What if there were a way to see all the possible inputs and their corresponding outputs to a function? Well, there is, and it's called **graphing**.

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 4.1:



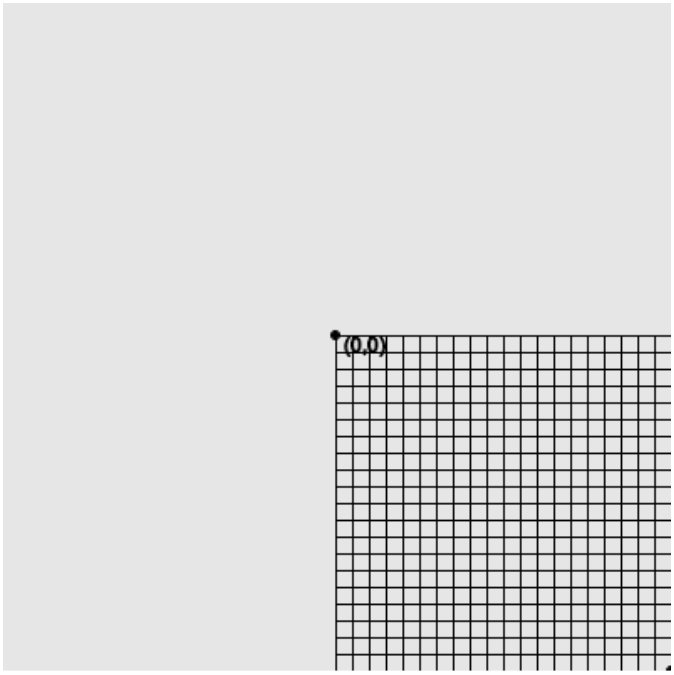
Unfortunately for us, computer graphics usually have their “origin” in the top left corner of the screen and the y-coordinates get larger as they go down the screen, not up, like in Figure 4.2:



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

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

So the origin will be in the middle of the screen. Figure 4.3 gives you the idea:



Our screen is still 600 pixels by 600 pixels but we probably will only need x- and y- values from -10 to 10. We can use Processing's scale function to scale it down, but we could also declare a range of x- and y-values we're interested in displaying:

```
rangex = 21
rangey = 21
```

We'll scale the coordinates down (or up) by simply multiplying the x- or y-coordinates by these factors:

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

Now we'll draw lines for the grid, in blue, like graph paper. The lines will go from (x, -10) to (x, 10) for each x value and from (-10, y) to (10, y) for each y-value.

```
    #cyan lines
    stroke(0,255,255)
    for i in range(-10,11):
```

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

Now we'll create the axes, a line between (0,-10) and (0,10) and another between (-10,0) and (10,0).

Exercise

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 “background(255)” to the beginning of the draw function to set the background white. That gives us a nice grid, like in Figure 4.4:

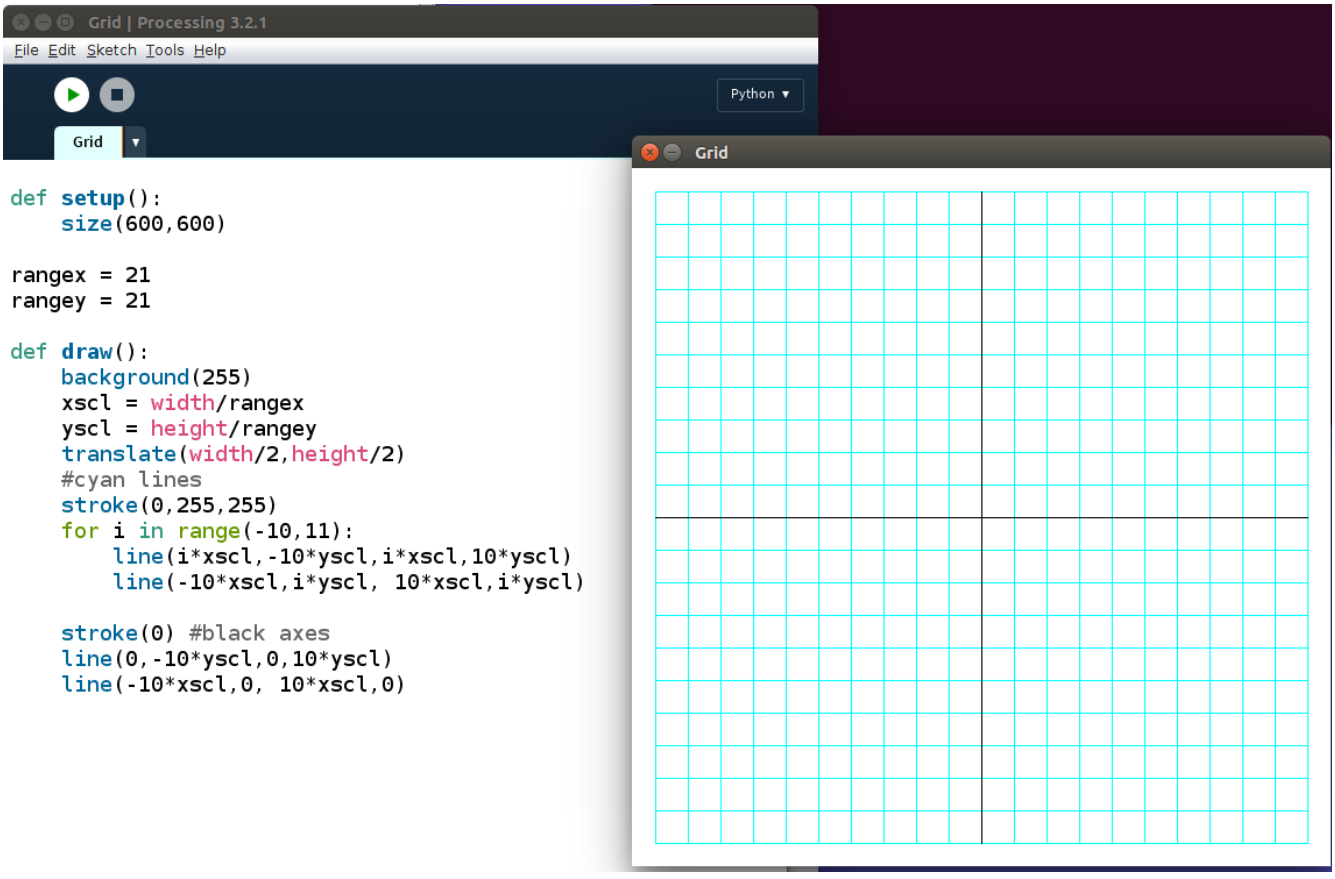


Figure 4.4: Creating a grid for graphing. You only have to do it once!

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 4.5:

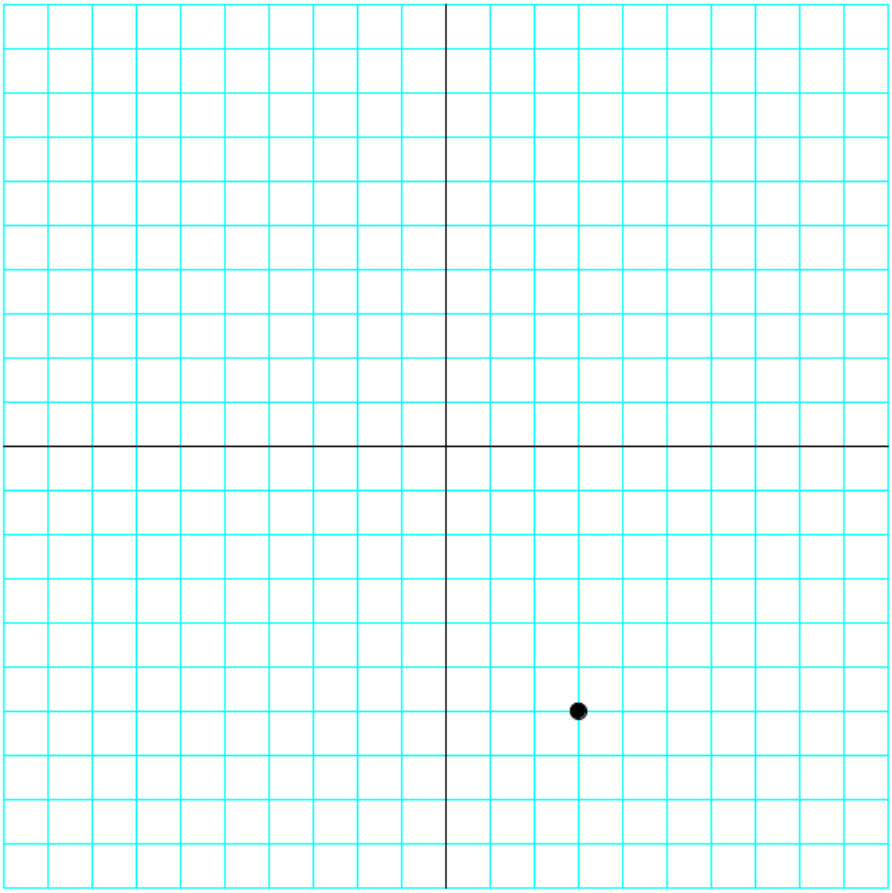


Figure 4.5: Checking our graphing program. Almost there!

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 draw function to flip that over:

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

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

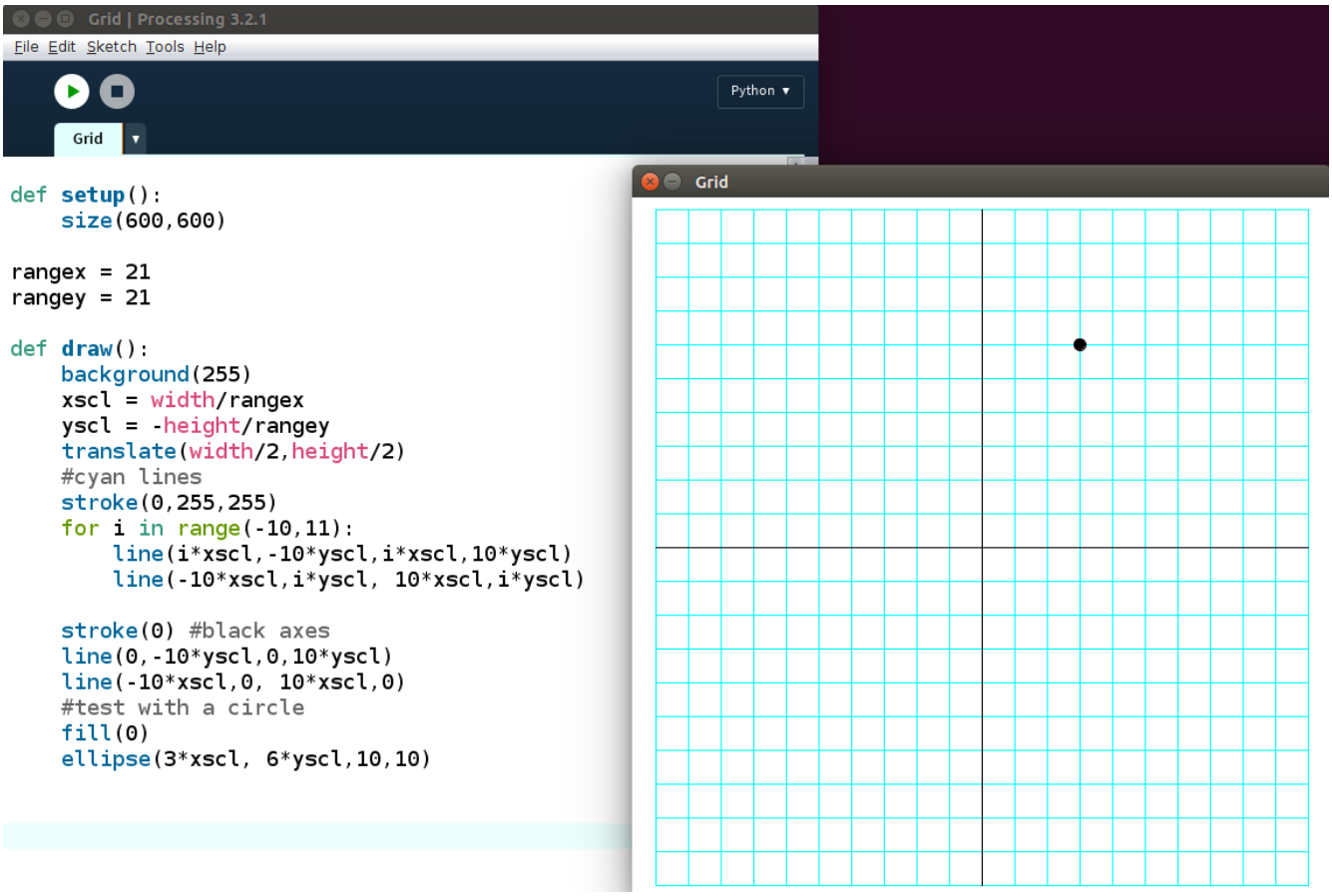


Figure 4.7: The grapher is working!

Now we're ready to solve our equation:

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

We'll add the function

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

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

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

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

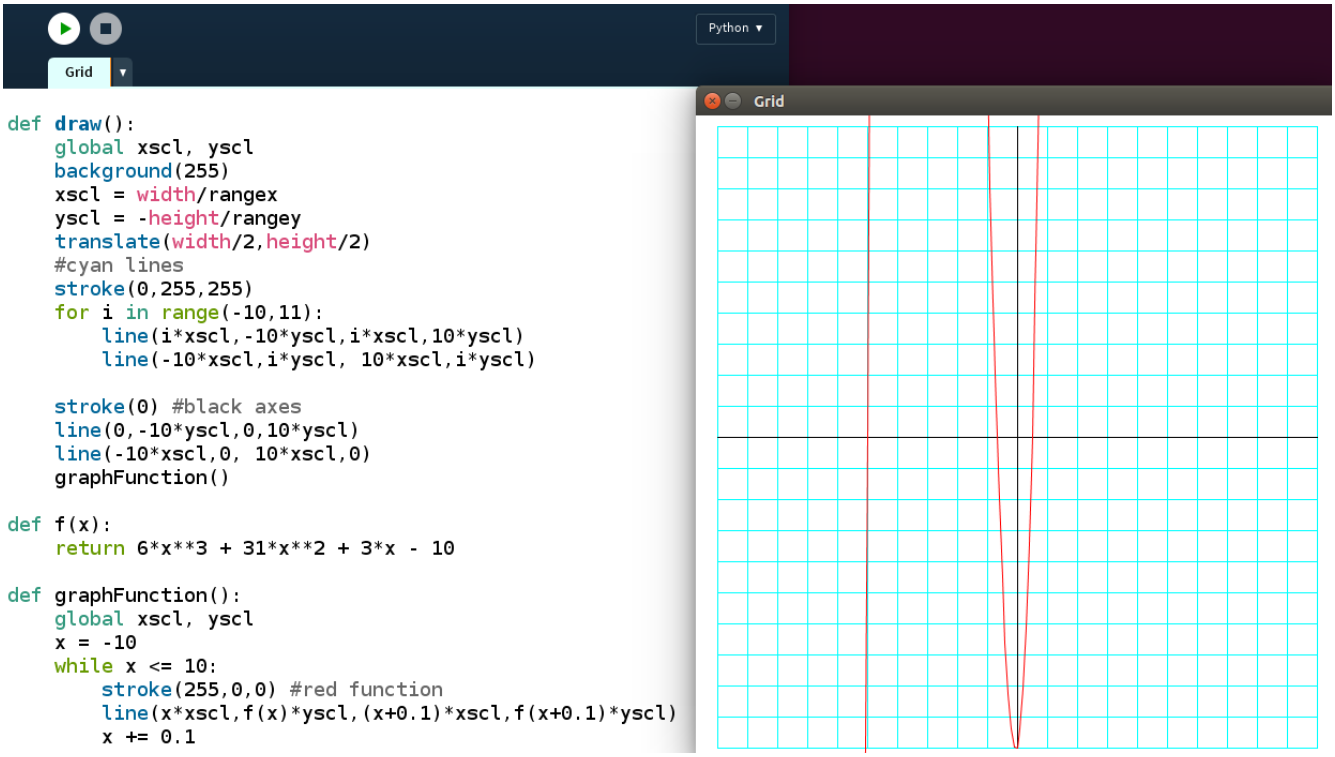


Figure 4.8: Graphing a polynomial function

The solutions to the equation 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

The easiest way to approximate the roots is by halving the range. 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 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 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?

```
'''The halving method'''

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

def average(a,b):
    return (a + b)/2

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.

So now 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 really learn to use matrices, we'll learn how to solve *systems* of equations, like these:

a + 2b -4c + d = -9

2a – 3b + 2c – 5d = 21

5a + b + 8c + 6d = -1

12a – 7b + 6c – 2d = -8

Variables are Supposed to Vary

People familiar with x's and y's in Algebra class might think variables represent a specific (and secret) number that people have to figure out. But that's not how variables are used in the real world. As you might suspect from their name, "variables" are letters that represent numbers that will vary, or change. We've already seen "i" used in this way in loops:

```
>>> for i in range(5):
      print(i)
0
1
2
3
4
```

The variable i starts off as 0 and changes (or "varies") every loop. This is incredibly useful, since it allows us to easily deal with situations when numbers change. We also saw how easy variables make

things when a number pops up repeatedly, and changing a number once is easier than changing it throughout the code. A simple example is the polygon function:

```
def polygon(num):
    for i in range(num):
        fd(100)
        rt(360/num)
```

Whenever you want to change the number of sides in the polygon, the variable called “num” changes the value automatically throughout the function.

1 [ChapterStart]

ChapterTitle

1st Para (large font paragraph at start of chapter; do not use for NXT books)

BodyFirst (standard paragraph following heading; no indent)

Body (standard paragraph; indented)

Basic: For manga speech bubble text (manga template only)

Italic: For italic manga speech bubble text (manga template only)

HeadA (first level heading)

HeadB (second level heading)

HeadC (third level heading)

```
CodeA (first line of code listings -- 78 characters available in layout)
CodeB (middle lines of code listings)
CodeC (last line of code listings)
```

```
CodeSingle (code listing of only one line)
```

```
CodeA Wide (listings set wide into the margin -- 95 characters available in layout)
CodeB Wide
CodeC Wide
```

```
CodeSingle Wide
```

```
CodeA Indent (listings indented in lists -- 74 characters available in layout)
CodeB Indent
```

```
CodeC Indent
```

```
CodeSingle Indent
```

```
1 CodeA Wingding (for putting wingdings in the margin)
2 CodeB Wingding (leave a default font space after the margin wingding)
3 CodeC Wingding
```

```
4 CodeSingle Wingding
```

Note (for short aside)

List Plain A (first item; shorter “term/description” lists -- begin with EmphasisBold + em space)

List Plain B (middle items)

List Plain C (final item)

ListHead

ListBody (use ListHead/ListBody for longer “term/description” lists)

NumListA (first item, numbered list)

NumListB (middle items)

NumListC (final item)

SubNumberA (first item, numbered sublist)

SubNumberB (second items)

BulletA (first item, bulleted list)

BulletB (middle items)

BulletC (final item)

SubBullet (bulleted sublist)

ListSimple (second paragraph for any kind of list item)

Block Quote (for longer quotations)

Epigraph

Caption (for images – numbered: Figure 1-1, etc.)

Listing (caption for code listings that need references – numbered: Listing 1-1, etc.)

Table Title (title for tables – numbered: Table 1-1, etc.)

Table Header (for header row paragraphs in tables)

Table Body (for body row paragraphs in tables)

Footnote

Author Query (use this for conversation between authors/editors – do not use Comments)

PRODUCTION DIRECTIVE (USE THIS FOR DIRECTIONS TO PRODUCTION FOLKS)

EmphasisBold (regular bold text – for menu paths, buttons, terms in plain lists)

EmphasisBoldBox (regular bold text in text boxes)

EmphasisItalic (regular italic text – for introduced vocab terms, etc.)

EmphasisItalicBox (regular italic in text boxes)

EmphasisItalicFoot (regular italic in footnotes)

EmphasisBoldItal (for bold text in italic paragraphs, italic text in bold phrases, etc.)

EmphasisNote (for italic in Notes; allows for different Note styles in different templates)

EmphasisRevItal (for text that would be italic in italic paragraphs or phrases)

EmphasisRevCaption (for text that would be italic in captions)

KEYCAP (for keyboard key names – underlying text should be lowercase)

`Literal` (for code text in regular paragraphs)

`LiteralBox` (for code text in text boxes)

`LiteralFootnote` (for code text in footnotes)

`Literal1st` (for code text in 1st Para)

`LiteralCaption` (for code text in captions)

`LiteralBold` (for highlighting user input in code listings, etc.)

`LiteralItal` (for placeholder words in code listings, etc.)

`LiteralBoldItal` (for italic in bold phrases or bold in italic phrases in code)

□□● □□□□□□ (MenuArrow) (numeral 4 arrow character for menu paths)

□⓪⑤□□⓪⑤□⑩ (Wingdings, ❶❷❸❹❺❻❼❽❾❿ for code annotation references in regular text paragraphs)

□⓪⑤□□⓪⑤□⑩ (Wingdings Small, ❶❷❸❹❺❻❼❽❾❿ for code paragraphs)

HeadANum (first level numbered head: 1.1, etc.)

HeadBNum (second level numbered head: 1.1.1, etc.)

HeadCNum (third level numbered head: 1.1.1.1, etc.)

Note Warning (for asides with "Warning" flag rather than "Note")

STYLES FOR TEXT BOXES:

HeadBox

BodyFirstBox.*

BodyBox.

List Plain A Box Text text text.

List Plain B Box Text text text.

List Plain C Box Text text text.

ListHeadBox

ListBodyBox.

BulletA Box

BulletB Box

BulletC Box

NumListA Box.

NumListB Box.

NumListC Box.

CaptionBox.

* FootnoteBox.