Table of Contents

Geometry

-*In the teahouse one day Nasrudin announced he was selling his house. When the other patrons asked him to describe it, he brought out a brick. "It's just a collection of these."*
- Idries Shah

In geometry class everything you learn about involves dimensions in space using shapes. You typically start with examining one-dimensional lines and two-dimensional circles, squares, or triangles, then move on to three-dimensional objects like spheres and cubes. These days, creating geometric shapes is easy with technology and free software, though manipulating and changing the shapes you create can be more of a challenge. In this chapter, you'll learn how to manipulate and transform geometric shapes using the Processing graphics package. You'll start with basic shapes like circles and triangles, and in future chapters ~~will soon~~ you'll be working with complicated shapes like fractals and cellular automata.

Downloading and Installing the Processing Python Package

Processing is a programming environment that makes it easy to visualize your code. You can think of it as a sketchbook for your programming ideas. In fact, each Processing program you create is called a *sketch*. In this chapter, we'll use Processing to visualize our code output as shapes and movement.

Before you can begin creating shapes, you'll need to download Processing. Navigate to *https://processing.org/download/?processing* and download the appropriate version of Processing for your operating system. Once downloaded, unzip the files and save them somewhere appropriate on your computer, then open the *Processing.exe* program.

Processing actually has its own programming language, but you can choose to use Python within the Processing sketchbook. Click the box on the upper right corner that should currently say *Java*. This should open a drop down menu. Click **Add Mode...** to open a list of modes you can install. Choose **Python Mode for Processing 3** and click **Install**. When you're done downloading the Python Mode, close this window to return to your sketch and select **Python** to turn on Python Mode.

You should see a blank Python workspace labeled "sketch_XXXX." Now you're ready to start creating shapes.

Drawing a Circle

Let's start with a simple one-dimensional circle. With a new sketch open in Processing, use the code shown in Figure 5-1 to create a circle on the screen:

```
#geometry.pyde

def setup():
    size(600,600)

def draw():
    ellipse(200,100,20,20)
```

Figure 5-1: Code to draw a circle in Processing's Python mode

Before we draw the shape, we first define the size of our sketchbook, known as the *coordinate plane*. In this example, we use the size`()` function to say that our grid will be 600 pixels wide and 600 pixels tall. With our coordinate plane set up, we use the drawing function `ellipse()` to create our circle on this plane. The first two parameters show where the center of the circle is located. Here, `200` is the x-coordinate and the second number `100` is the y-coordinate of this circle's center, placing it at (200,100) on the 600x600 plane. The last two parameters determine the width and height of the shape in pixels. In the example, the shape is `20` pixels wide and `20` pixels tall. Because the points on the circumference are equidistant from its center, this forms a perfectly round circle.

When you click the **Run** button (it looks like the Play symbol), a new window should open, in which you should see a small circle like in Figure 5-2:

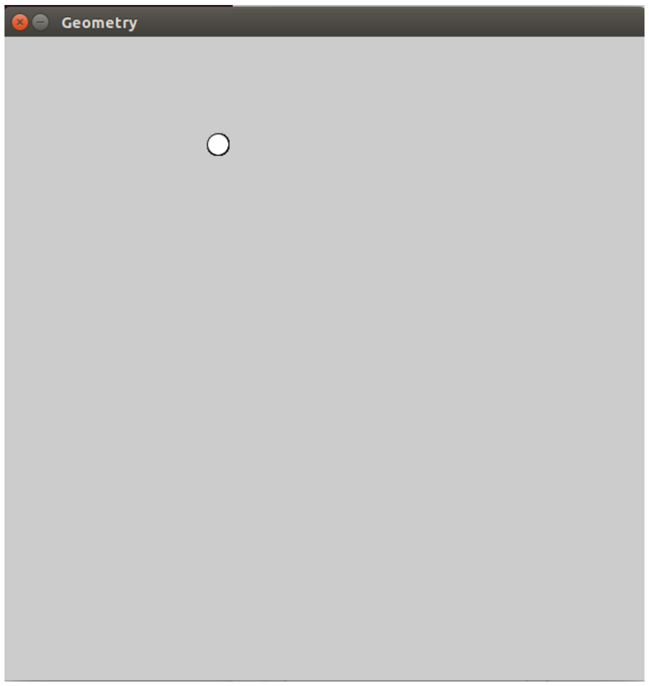
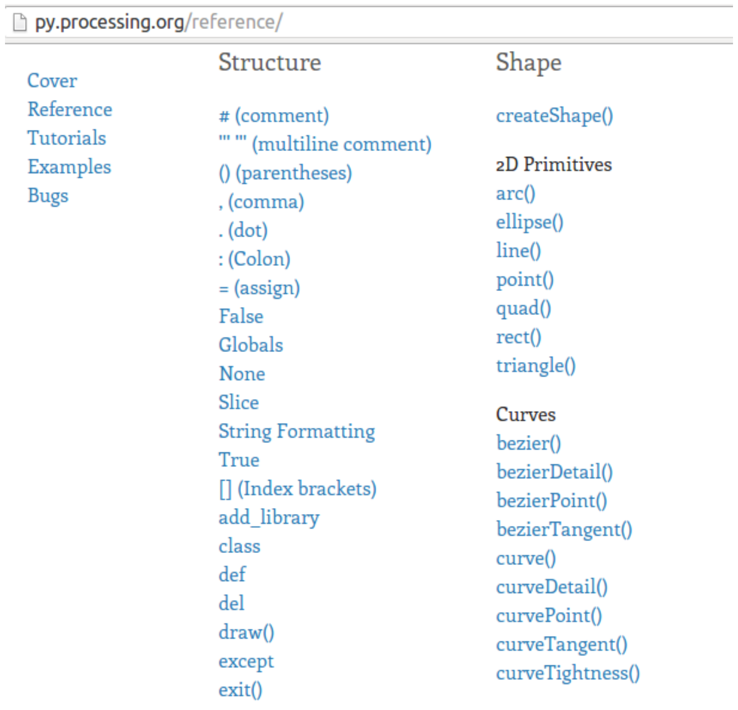


Figure 5~~.~~-2: The output of the above code showing a small circle

Processing has a number of other functions that make it easy to draw shapes. To see the full list, look at the reference pages at *https://processing.org/reference/* to find functions for drawing ellipses, triangles, rectangles, arcs and much more. Figure 5-3 shows a screenshot of the reference~~s~~ page:



py.processing.org/reference/

Cover
Reference
Tutorials
Examples
Bugs

Structure

(comment)
""" """ (multiline comment)
() (parentheses)
, (comma)
. (dot)
: (Colon)
= (assign)
False
Globals
None
Slice
String Formatting
True
[] (Index brackets)
add_library
class
def
del
draw()
except
exit()

Shape

createShape()

2D Primitives
arc()
ellipse()
line()
point()
quad()
rect()
triangle()

Curves
bezier()
bezierDetail()
bezierPoint()
bezierTangent()
curve()
curveDetail()
curvePoint()
curveTangent()
curveTightness()

Figure 5-3: The reference page of the Python mode of Processing~~.~~

Now that you know how to draw a circle in Processing, you'll learn how to use these simple shapes to create dynamic, interactive graphics. In order to do that, you'll first need to learn about location and transformations. Let's start with location.

Specifying Location Using Coordinates

In Figure 3-X, we used the first two parameters of the `ellipse()` to specify our circle's location on the grid. Likewise, each shape we create using Processing needs a location that we specify with the coordinate system, where each point on the graph is represented by two numbers: (x, y). The first number tells you where it is on the x-coordinate (horizontal line) and the second number tells you the y-coordinate (vertical line). In traditional math graphs, as you likely know, e the *origin* where x=0 and y = 0 is at the center of the graph as in Figure 5.4

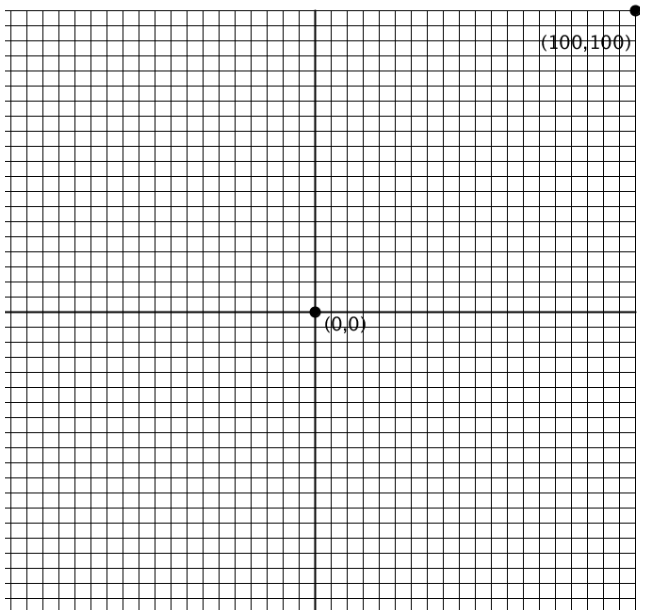


Figure 5-4: A traditional coordinate system with the origin in the center

In computer graphics, however, the coordinate system is a little different,as the origin is in the top-left corner of the screen, so that x and y increase as you move right and down respectively, as in Figure 5.5:

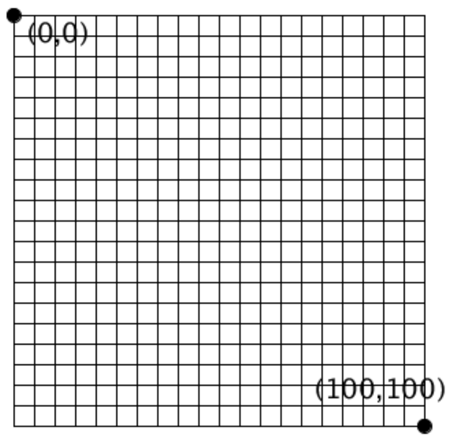


Figure 5-~~.~~5 The coordinate system for computer graphics with the origin in the top-left corner

Each coordinate on this plane represents a pixel on the screen. As you can see with this graph, this means you don't have to deal with negative coordinates. ~~numbers i(unless you really want to).~~ We'll use ~~coordinates~~functions to transform and translate increasingly complex shapes.

Drawing a single circle was fairly easy, but drawing multiple shapes can get complicated pretty quickly. For example, imagine drawing a design like the one shown in Figure 5-6:

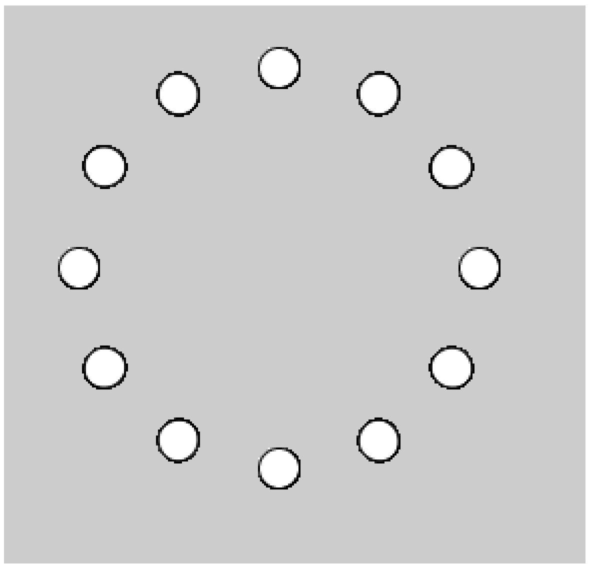


Figure 5-6: A circle made up of circles

This seems like it would take a lot of repeated code, specifying the size and location of each individual circle, spaced perfectly equally. Fortunately, you don't really need to know the absolute x- and y-coordinates of each circle to do this. With Processing, you can easily place objects wherever you want on the grid. Let's see how you can do this.

Transformation Functions

You might remember doing transformations with pencil and paper in geometry class. Geometric transformations, like translation and rotation, change where and how your objects appear without altering the objects themselves. For example, a triangle can be moved to a different location or spun around without changing its shape or size.~~, for example flipping a triangle on its base~~. Processing has a number of built-in transformation functions that make it easy to translate and rotate objects.

Translating Objects with translate()

To translate means to move a shape on a grid so that all points of the shape move the same direction and the same distance. It's basically just moving a shape on a grid without changing the shape at all, not even tilting it. In math class that means all the points in the object change their coordinates. But i~~I~~n Processing, you ~~actually~~ translate an object by moving the *grid* itself, while the object's coordinates stay the same.~~ remains in place.~~ For an example of this, let's put a rectangle on the screen. You just need to tell Processing the x- and y-coordinates of the rectangle (it's measured from the top-left corner), its width and its height:

```
#geometry.pyde

def setup():
    size(600,600)

def draw():
    rect(20,40,50,30)
```

Run this and you should see the rectangle, as in Figure 5-7:

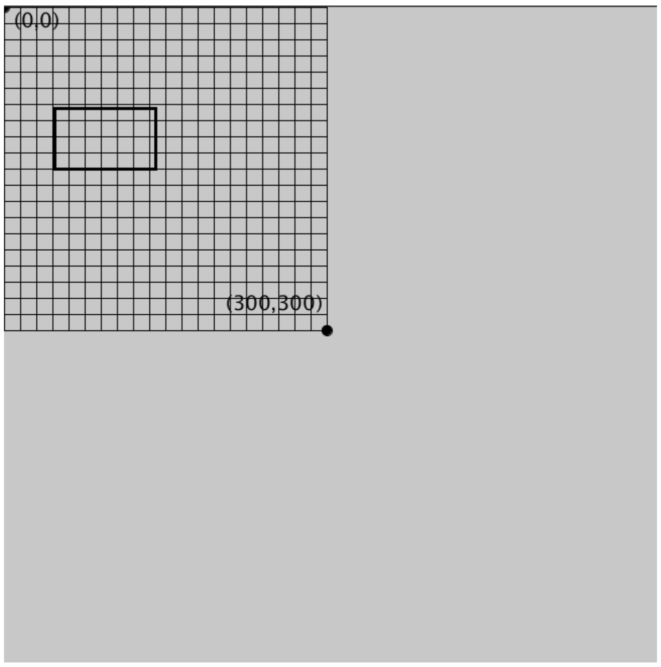


Figure 5-7: The default coordinate setup with the origin at the top left

You won't see the grid. I just put that in there to show you how it'll move. Now we'll tell Processing to translate the rectangle using the following code. Notice the coordinates of the rectangle don't change/~~:~~

```
#geometry.pyde

def setup():
    size(600,600)

def draw():
    translate(50,80);
    rect(50,100,100,60)
```

This~~-~~ should move the entire grid 50 pixels to the right and 80 pixels down, as shown in Figure 5-7:

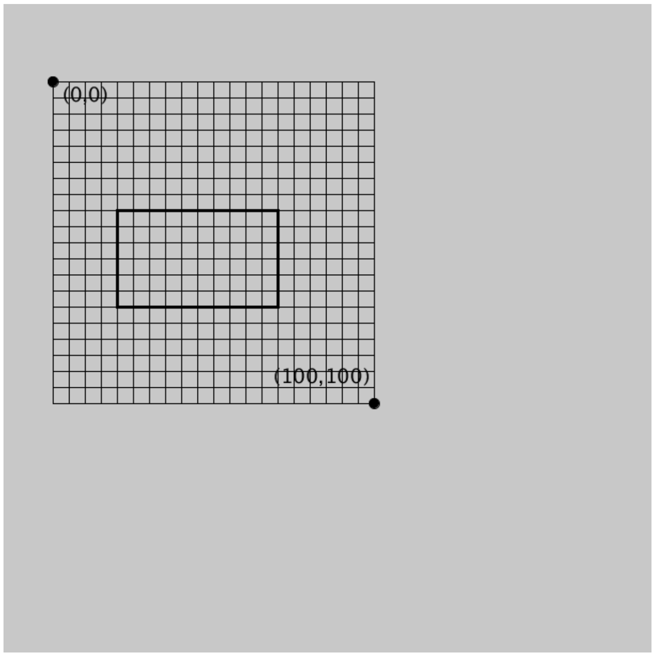


Figure 5-7: Translating a grid and a rectangle 50 pixels to the right and 80 pixels down

Very often it's useful (and easier!) to have the origin (0,0) in the center of the canvas and it's easy to move it using translate. You might change the width and height of your canvas if you want it bigger or smaller, so it's handy to be able to use Processing's built-in "width" and "height" variables instead of changing the numbers manually. Whatever numbers you put in the size declaration in the setup function will become the "width" and "height." In this case, since I used size(600,600), they're both 600.

Change the translate line to this:

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

~~please discuss this code `translate(width/2, height/2);` in more detail, why do we write it this way?~~

This should look like Figure 5-8:

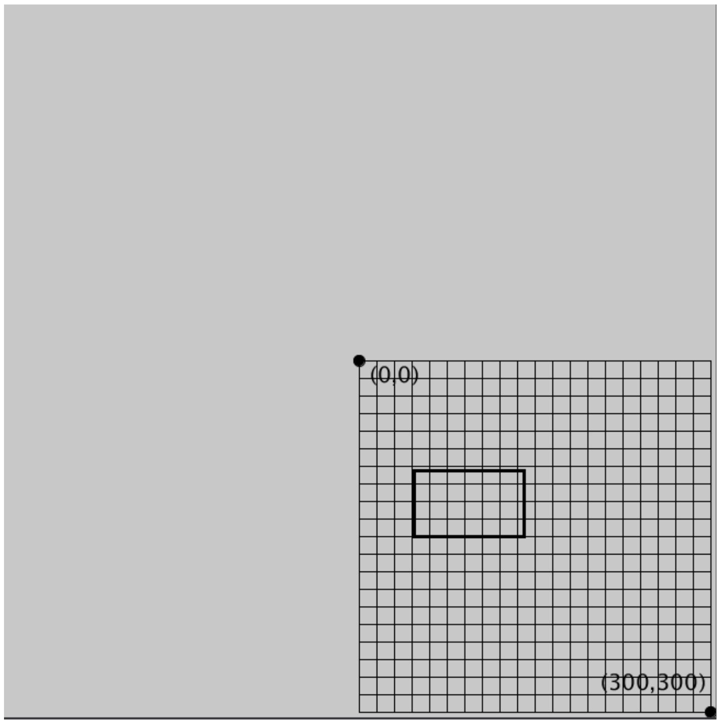


Figure 5~~.~~-8: The grid is translated to the center of the screen

Notice that the origin is still located at (0, 0). This shows that we haven't actually moved the origin point but the entire coordinate plane itself.

Rotating Objects with rotate()~~e~~

In geometry, rotation is a kind of transformation where you turn an object around a centerpoint, as if it's turning on an axis. The `rotate()` function in ~~Python~~ Processing rotates the grid around the origin (0,0). It takes a number ~~figureradian~~ as its argument to specify how many radians you want to rotate the grid. If you think in degrees, like I do, you can use the `radians()` function to convert your figure in degrees to radians so you don't have to do the math.

To see how the function works, enter the following two examples into the sketch and run them

. Figure 5-9 shows the result:

`rotate(radians(20));`	`translate(width/2,height/2);`
	`rotate(radians(20));`

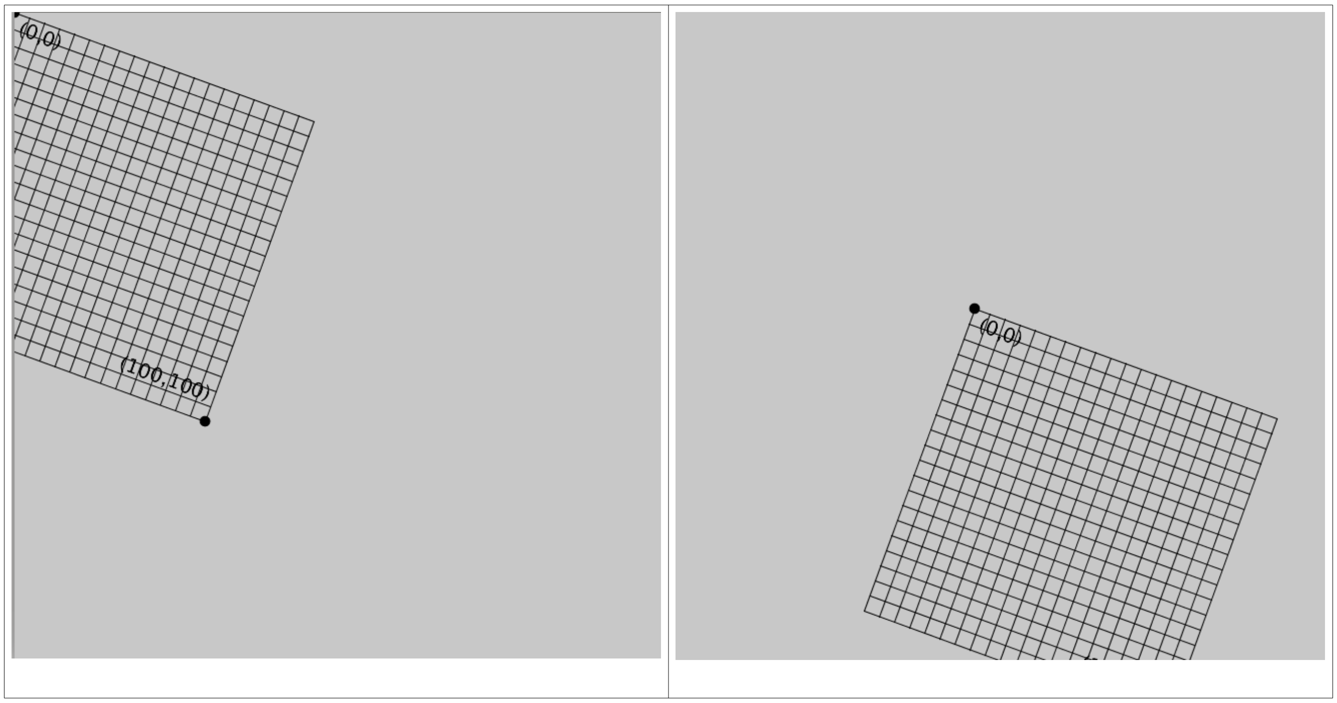


Figure 5~~.~~-9: Rotating the grid ~~always happens~~ about (0,0)

~~function first."radians()", just convert to radians using the)like I do(If you think in degrees radians. is done in ion, rotat in the code aboveAs you can see~~

On the left of ~~In the f~~Figure 5-9, ~~above on the left,~~ the grid is rotated 20 degrees around (0,0), which is at the top left corner of the screen. ~~the n the figure on I~~On the right, the origin ~~has been~~is first translated to the center of the canvas and ***then*** the grid ~~has been~~was rotated.

The `rotate()` function makes it easy to make ~~Making~~ circles of objects like the one in Figure 5-6 ~~is easy with `rotate`. Here are~~ using the following steps:

1. Translate to where you want the center of the circle to be.
2. Rotate the grid and put the objects ~~on~~along the circumference of the circle.

Exercise 5.1

Create a program that will generate ~~what~~the design ~~is~~ in Figure 5.6~~7~~ using the steps above. You'll need to use a `for i in range()` command to make sure the circles are evenly spaced. Think about how many degrees (that you'll translate into radians) the smaller circles will have to rotate to make a full circle, remembering that a circle is 360 degrees.

~~**Solution:**~~

~~First we have to translate to the center of the screen:~~

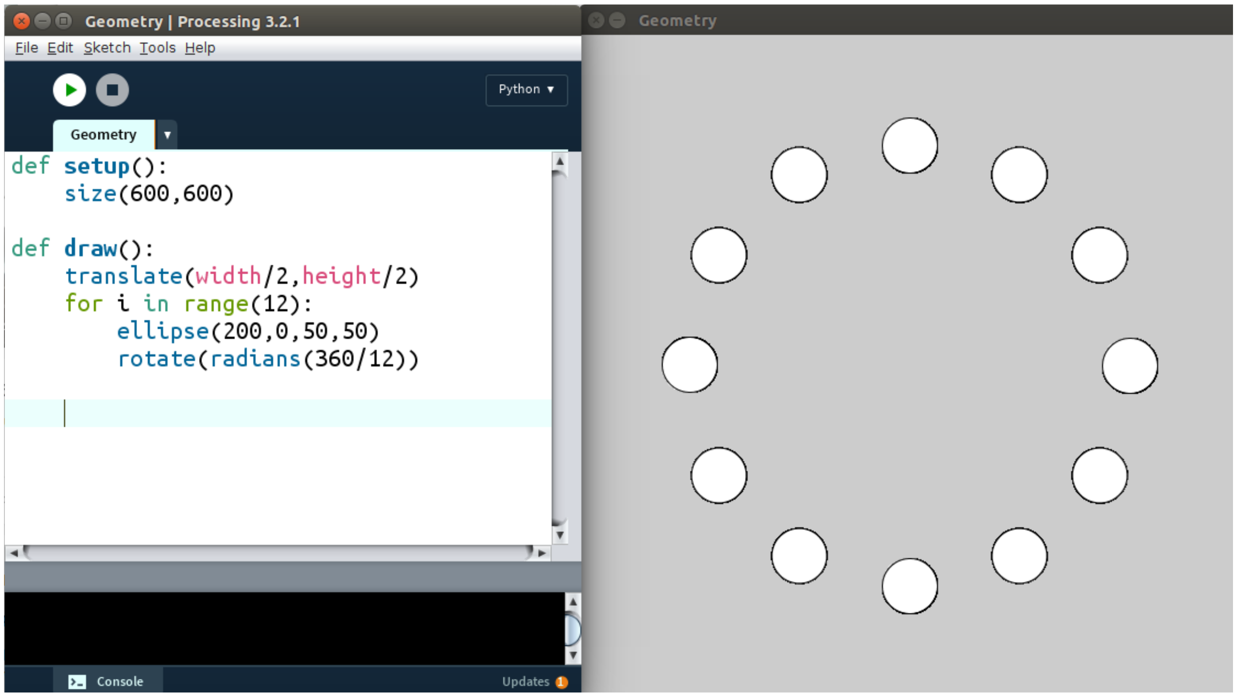
```
~~translate(width/2,height/2)~~
```

~~Then we have to start a loop that will place an ellipse at some point and rotate after each one.~~

```
~~for i in range(12):~~
~~        ellipse(200,0,50,50)~~
~~        rotate(radians(360/12))~~
```

~~You'll get what's in Figure 5.10:~~

~~*Figure 5.10: The code and output for translating, rotating and repeating.*~~



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

def draw():
    translate(width/2,height/2)
    for i in range(12):
        ellipse(200,0,50,50)
        rotate(radians(360/12))
```

~~If you add another step to that sequence, you can rotate the objects in the circle. Obviously we can't tell if the circles are rotating, so we'll replace them with squares for now.~~

~~Exercise 5.2: Squaring the Circle~~

~~Change the program from drawing circles to squares.~~

Exercise 5.2: Squaring the Circle

Modify the program you wrote in Exercise 5.1 and change the circles into squares.

Animating Objects

Processing is great for animating your objects to create dynamic graphics. As you can tell from the output when you run Exercise 5.1 and 5.2, `rotate` is done instantly. In order to see the rotation unfold in real-time, we have to introduce the time variable `t`. ~~You can either use Processing's built-in frameCount variable or create your own `t` variable. In this section, we're going to create our own.~~ We'll use the code for creating the circle of circles from Exercise X and first turn those squares into circles, then animate it so the squares rotate. If you haven't managed to complete Exercise X, flip to the back of the book to find the solution.

Creating the t Variable

The time variable, `t`, is ________________________.

To start, ~~You can~~ create the variable and initialize it with 0 by adding **"t = 0"** before the setup function. Then insert the following ~~and the line~~code before the `for` loop:

```
geometry.pyde
def setup():
    size(600,600)

def draw():
    translate(width/2,height/2)
    rotate(radians(t))
    for i in range(12):
        rect(200,0,50,50)
        rotate(radians(360/12))
```

~~`(t)) rotate(radians`~~

~~Please explain what this code does~~

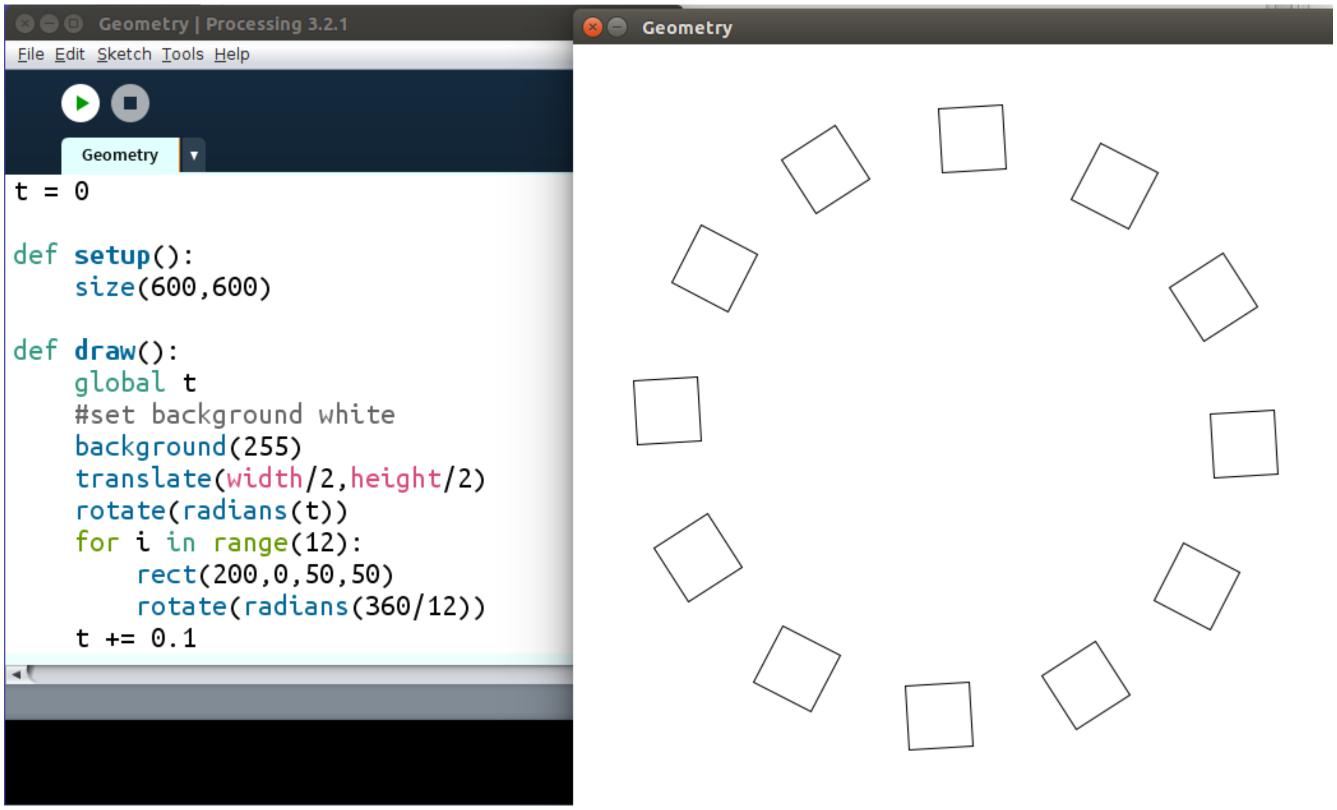
However, if you try to run this, ~~Y~~you'll get the following error message:

```
UnboundLocalError: local variable 't' referenced before assignment
```

This is because Python doesn't know whether you're creating a whole new "local variable" `t` ***inside*** the function that doesn't have anything to do with the "global variable" `t` ***outside*** the function. ~~All y~~You have to ~~do is~~ add **"global t"** at the beginning of your draw function and ~~it~~the program ~~will~~should now know you're referring to ~~mean~~ the same `t`. ~~Now~~

We need some additional code to tell the squares how XXXX to rotate.

Also notice that we've swapped `ellipse()` with `rect()` to turn the circles into squares. Enter the code in Listing XX and run it and you should see the whole circle of squares start to rotate in a circle, as in Figure 5-11:



```
t = 0

def setup():
    size(600,600)

def draw():
    global t
    #set background white
    background(255)
    translate(width/2,height/2)
    rotate(radians(t))
    for i in range(12):
        rect(200,0,50,50)
        rotate(radians(360/12))
    t += 0.1
```

Figure 5~~-~~-11: Making squares rotate in a circle

~~That's how you rotate the whole circle. What we want is to~~

Please explain what t += 0.1 means?

Now let’s try rotating each square. Since rotating is done around (0,0) in Processing, inside the loop we have to first translate each square to where it to be, then rotate, and finally draw the square. Run the following code in Listing 4-1:

```
for i in range(12):
    translate(200,0)
    rotate(radians(t))
    rect(0,0,50,50)
    rotate(radians(360/12))
```

~~Please explain each line of code to show how it makes each square rotate~~

pushMatrix() and popMatrix()

When you run Listing 4-1, you should see that it creates some strange behavior. That's because of changing the center, and changing the orientation of the grid so much. We need to figure out how to rotate each square but also keep them along the circumference of the bigger circle. Processing has two built-in functions that save the orientation of the grid at a certain point then return to that orientation: `pushMatrix` and `popMatrix`. In this case, we want to save the orientation when we're in the center of the screen. So revise the loop to look like this:

```
for i in range(12):
    pushMatrix() #save this orientation
    translate(200,0)    rotate(radians(t))
    rect(0,0,50,50)
    popMatrix() #return to the saved orientation
    rotate(radians(360/12))
```

The pushMatrix() saves the position of the coordinate system at the point where ____________.
Then we translate _______

Then we use popMatrix() to

Rotating Around the Center

~~That~~This should work~~s~~ perfectly, but ~~if~~ the rotation may looks strange ~~to you;,~~ ~~it's~~that's because Processing by default locates a rectangle at its top left corner and rotates it about its top left corner. This makes the squares look like they're veering off the path of the larger circle. If you want ~~trectangle~~ your square~~s~~s to ~~be located at (and~~ rotate~~d~~ around~~)~~ their centers, add this line to your setup function:

```
rectMode(CENTER)
```

~~Yes,~~Note that the capitalization in `rectMode()` matters. Adding `rectMode(CENTER)` ~~Now the~~should make each ~~rectangles~~square ~~will~~ rotate around ~~their~~its center~~s, too~~. If you want them to spin more quickly, change the~~ir~~ rotate line to increase the time in `t` ~~to this~~like so:

```
rotate(radians(5*t))
```

The 5 is the frequency of the rotation. This means ________. **Change it ~~and~~to see what happens!**

Creating an Interactive Rainbow Grid ~~Grid of ObjectsDrawing a~~

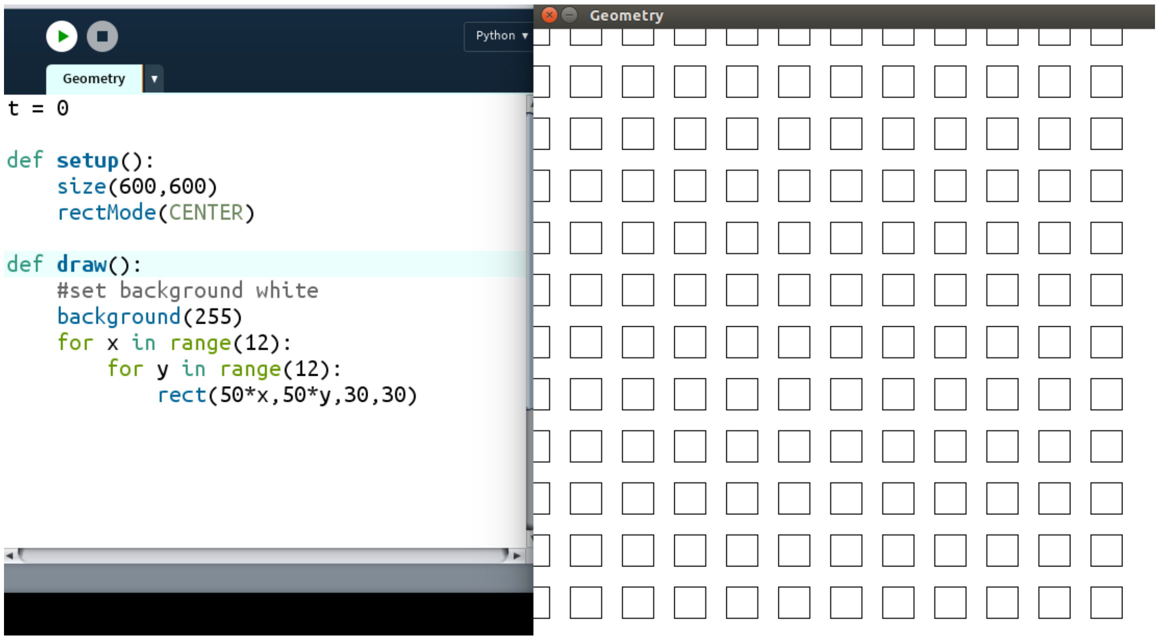
Now that you’ve learned how to create designs using loops and rotate them in different ways, we'll create something pretty awesome: a grid of rainbow-colored squares that follows your mouse cursor! The first step is to make a grid.

Drawing a Grid of Objects

We'll make a 12x12 grid of squares, evenly sized and spaced, to begin~~ing~~ with. Making a grid this size may seem like a time-consuming task but in fact, it’s easy to do using a loop.

Make a 12 x 12 grid of squares on a white background. Your squares will need to be `rects`, and you'll need to use a for loop within a for loop to make sure they are all the same size and spaced equally.

The code in Listing 4-X should create the grid in Figure 4-X.



```
t = 0

def setup():
    size(600,600)
    rectMode(CENTER)

def draw():
    #set background white
    background(255)
    for x in range(12):
        for y in range(12):
            rect(50*x,50*y,30,30)
```

Listing 4-X:

Figure 4-X:

We first set up the background to be 600 by 600 pixels and XXX it with `rectMode(CENTER)`.

Then in the `def draw()` section we first set the background to white by setting it to 255, then center the grid by translating it to `(20,20)`.

XXX

We vary the variables `x` and `y` by regular intervals to ....

~~Solution:~~

~~With very little tRun in codehe,- Figure 5.12 is almost thereto: make a grid of squares~~

-

~~*Figure 5.: Making a grid of squares12*~~

~~We just want the~~In this example code, we use a for loop inside of a for loop to ____________. ~~We want to vary x and y by regular intervals.~~

```
~~ grid To make theto be a little more centered, so first. Add this line it we'll translate afterbefore the loop “background(255)”:~~

~~translate(20,20) to center your entire grid~~
```

The result should look like ~~This makes~~ Figure 5-.13

~~-*Figure 5.: Centering your grid of squares13*~~

~~, specifically we need to explain what rect(50*x,50*y,30,30) doesPlease explain the listing in Figure 5-13~~

Exercise 5.4 Rocking the Grid

Make each square in the grid spin using the `rotate` function. Once you've done that, return to the static grid, you'll need it for the next section.

Adding the Rainbow Color to Objects

The Processing XX also lets you add colors to your shapes. The `colorMode()` function helps us add some cool color to our sketches! It’s used to switch between the RGB and HSB modes. RGB is ~~value~~three numbers indicating ~~a mixture~~ amounts of Red, Green and Blue. In HSB the three numbers will be amounts of “Hue, Saturation, and Brightness” and the only one we need to change is the first number, the hue.

We’ll use the previous example of rotating squares to practice adding colors. Use the following code to add a time variable and a couple of `rotate` functions to make each square rotate:

```
def setup():
    size(600,600)
    rectMode(CENTER)
    colorMode(HSB)

def draw():
    #set background black
    background(0)
    translate(20,20)
    for x in range(30):
        for y in range(30):
            d = dist(25*x,25*y,mouseX,mouseY)
            t = 100/(d+0.01) #to prevent zero division
            fill(400*t,360,360)
            rect(25*x,25*y,20,20)
```

We insert the `colorMode()` function and pass HSB to it.

The hue is the only ~~one~~thing we~~'ll~~ change~~:~~. ~~W~~we'~~ll~~ update the hue according to the distance the rectangle is from the mouse. We do this by ______________________________.

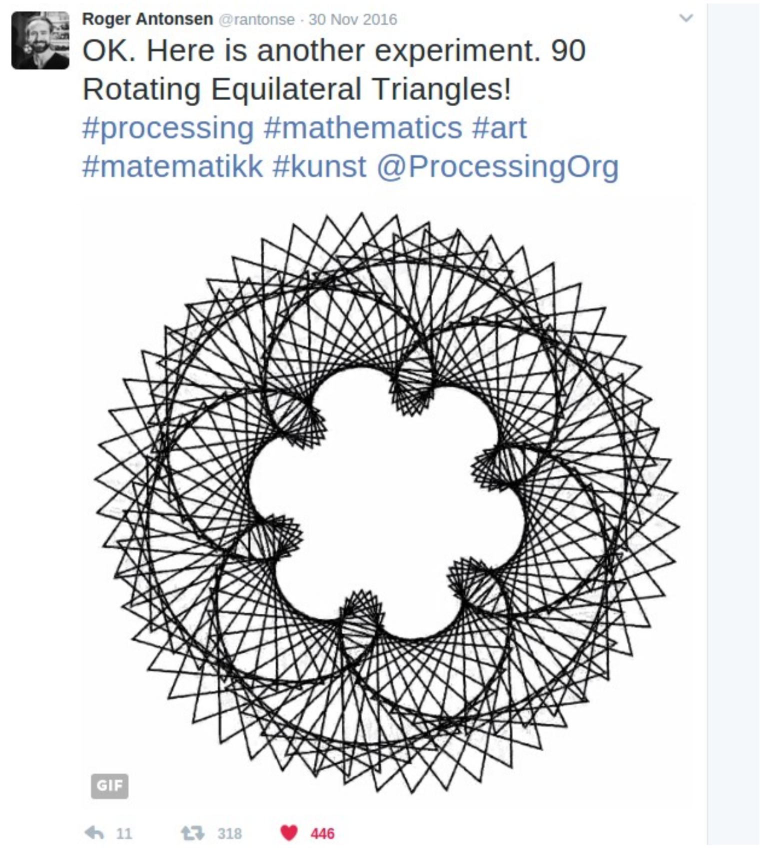
Run this code and ~~You'll~~you should see a very colorful design ~~which will~~that changes colors according to the ~~follow your~~ mouse's location ~~around~~, as shown in Figure 5~~-~~.14:

Figure 5~~-~~.14: Adding colors to your shapes

Now that you've learned how to add colors to your objects, let's explore how we can create more complicated shapes.

Drawing Complex Triangle Patterns

In this section, we'll create more complicated spirograph-style patterns using triangles. For example, take a look at the sketch made up of rotating triangles in Figure 5-12, created by the University of Oslo's Roger Antonsen:


Roger Antonsen @rantonse · 30 Nov 2016

OK. Here is another experiment. 90 Rotating Equilateral Triangles! #processing #mathematics #art #matematikk #kunst @ProcessingOrg

Figure 5-15: Sketch of 90 rotating equilateral triangles by the University of Oslo's Roger Antonsen

This sketch blew me away! But though this design looks very complicated, it's not that difficult to make. Antonsen gave us a helpful clue to creating this design when he named the sketch "90 Rotating Equilateral Triangles." It tells us that all we have to do is figure out how to draw an equilateral triangle and rotate it, then repeat that for 90 of them. Let's first learn how to draw an equilateral triangle using the `triangle()` function.

~~Please introduce the `triangle()` function here and explain which parameters it takes before introducing Figure 5-16~~

~~**Exercise 5-5: Spin Cycle**~~

~~Create a triangle in a Processing sketch and rotate it using the `rotate` function.~~

~~**Solution:**~~

Using the code in ~~Figure~~Listing 4-x ~~5-16 is Here's~~ is one way to create a rotating triangle~~ but as you can see in Figure 5.16, it rotates around one of its vertices~~:

When you run this code, you should see something like Figure 4-X.

Figure 5~~-~~.16: R~~A r~~otating a triangle around one of its vertices~~, but…~~

~~Please explain this code, applying what we learned about the triangle()~~

As you can see in Figure 5-16, the triangle rotates around one of its *vertices*, or points, and so creates a circle with the outer point. You'll also notice that this is a right triangle. In order t~~T~~o ~~copy~~recreate Roger's sketch, we need to draw an equilateral triangle, ~~which is~~ a triangle with equal sides. ~~for one, and w~~We also need to find the center of the equilateral triangle to be able to rotate it ~~properly~~about its center. To do this, we'll need to find the location of the three vertices of our triangle. Let's learn how to draw an equilateral triangle by specifying the location of its vertices.

30-60-90 Triangle

In order to find the location of the three vertices of our equilateral triangle, you'll need to review a ~~special~~particular triangle you've ~~might have~~ likely seen in geometry class: the 30-60-90 triangle. ~~look at~~~~Figure 5of an example an-17 shows~~First we need an equilateral triangle, shown in Figure 4-17. ~~17.in Figure 5~~:

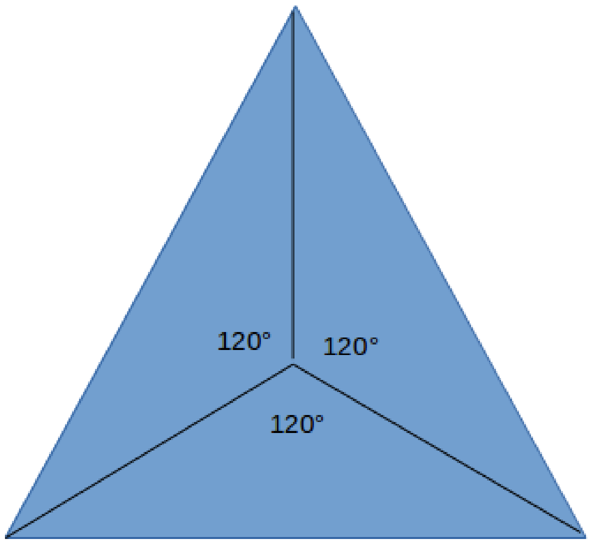


Figure 5~~-~~.17: An equilateral triangle ~~dissected~~divided into three~~3~~ equal parts

This equilateral triangle is made up of three equal parts. The point in the middle is the center of the triangle, with each dissecting line meeting at a 120 degree angle. Now let's cut the bottom triangle in half again, as shown in Figure 5~~-~~.18:

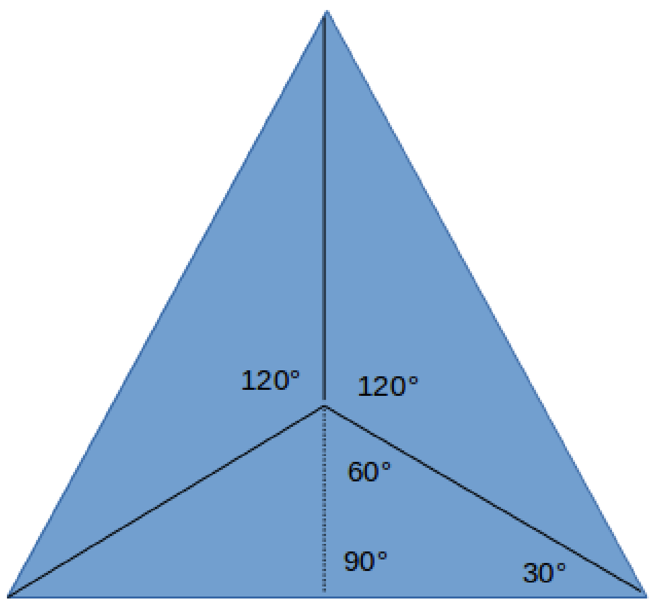


Figure 5-18 Dividing up the equilateral triangle into special triangles

Dividing the bottom triangle in half creates two right angle triangles. These right triangles are classic 30-60-90 triangles, a special triangle studied in geometry and trigonometry classes. A 30-60-90 triangle is a right triangle whose internal angles are 30, 60, and 90 degrees and whose sides have special relationships with respect to its *hypotenuse*, the longest side of a right triangle---you may be familiar with this from Pythagoras, which will come up again shortly. You need to know the lengths of the sides of this special triangle, because it'll help you find the three vertices with respect to the center, which we need in order to draw an equilateral triangle. Figure 5-19 zooms in on a 30-60-90 triangle:

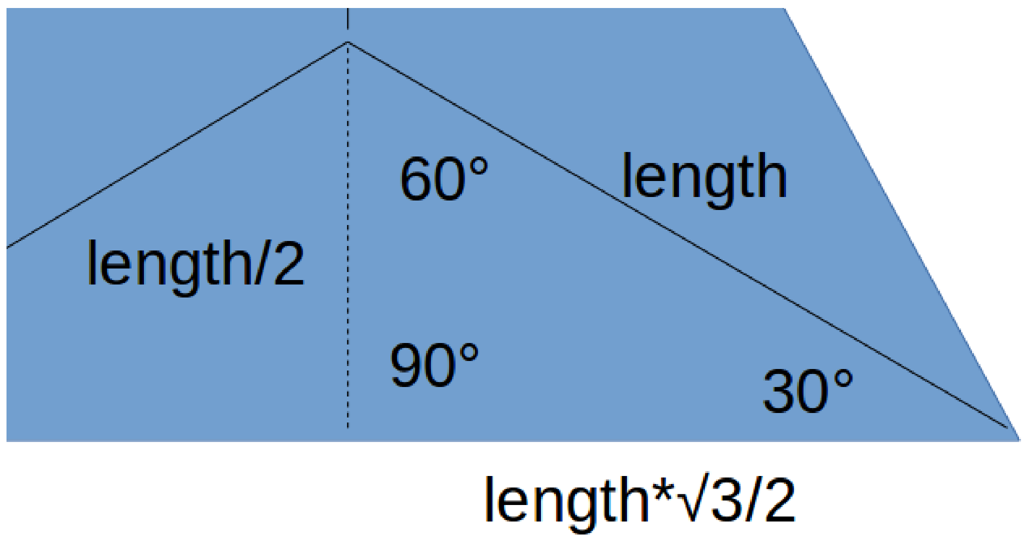


Figure 5-19: The 30-60-90 triangle up close and personal

The distance from the center of the equilateral triangle to the vertex is the length, which is also the hypotenuse of the 30-60-90 triangle inside the equilateral. We can find the lengths of the other sides with reference to the length variable.

The shorter leg is always half the ~~length~~hypotenuse and the longer leg is the ~~length~~measure of the shorter leg times the square root of 3. So if we use the center point for drawing the triangle, the coordinates of the three vertices would be as shown in Figure 5-20:

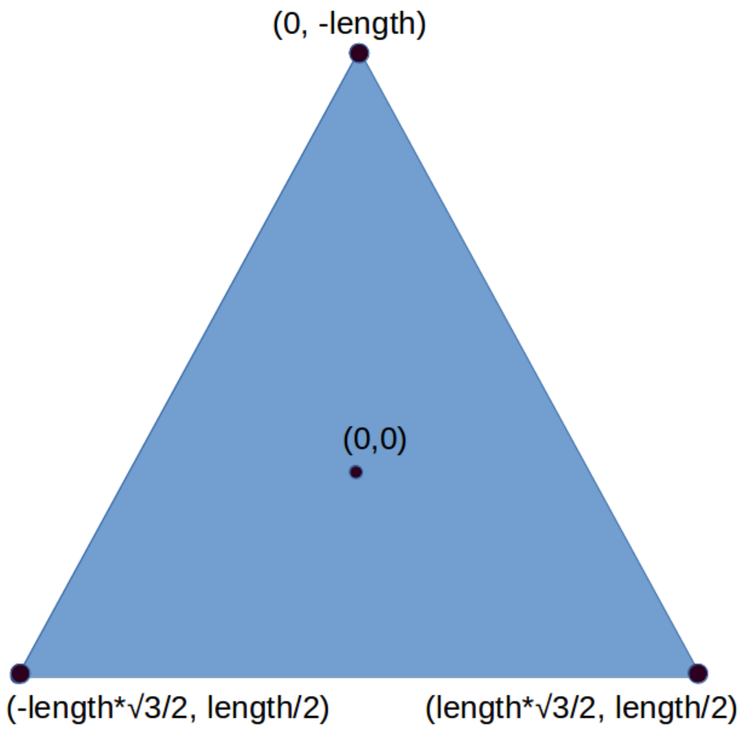
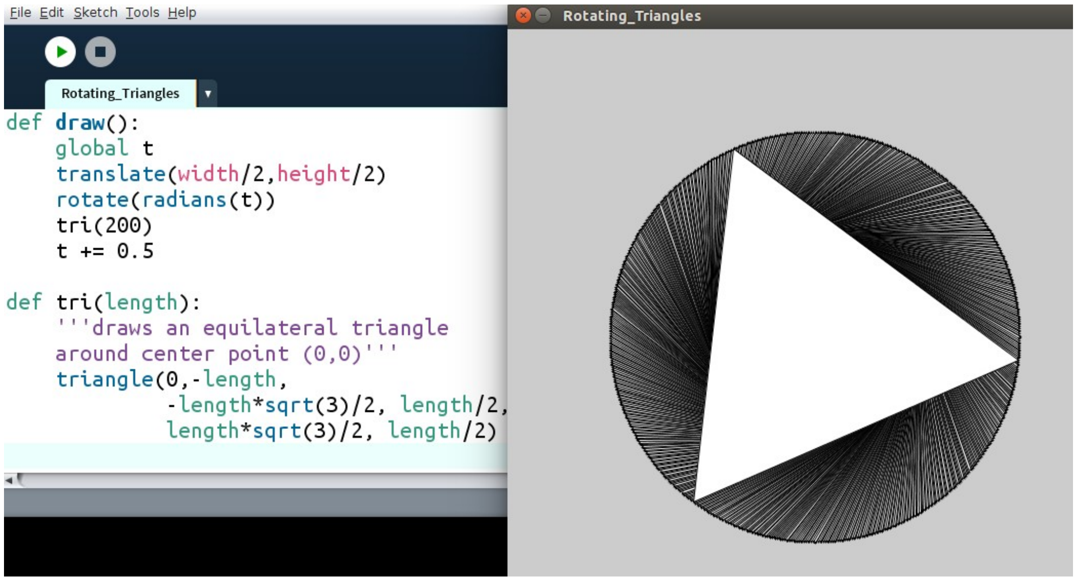


Figure 5-20: The vertices of the equilateral triangle

As you can see, because this triangle is made up of 30-60-90 triangles on all sides, we can use them to figure out how far each vertex lies from with respect to the origin.

Drawing an Equilateral Triangle

Now we know what we need to do, let's do it. We can ~~now~~ use the vertices we derived from the 30-60-90 triangle to create an equilateral triangle. The following code~~ to make the triangle is~~ shown in Figure 5~~-~~.21 create an equilateral triangle~~and the rotation still works.~~:



```
def draw():
    global t
    translate(width/2,height/2)
    rotate(radians(t))
    tri(200)
    t += 0.5

def tri(length):
    '''draws an equilateral triangle
    around center point (0,0)'''
    triangle(0,-length,
             -length*sqrt(3)/2, length/2,
             length*sqrt(3)/2, length/2)
```

Figure 5~~-~~21: Code drawing an equilateral triangle

First, we write the `tri()` function to take the variable `length`, which ____________________. Using the `triangle()` function, we then specify the location of each of the three vertices of the triangle: `(0, -length)`, `(-length*sqrt(3)/2, length/2)` and `(length*sqrt(3)/2, length/2)`.

Now we can cover up all the ~~old~~ triangles created during rotation by adding this line to the beginning of the **`draw`** function:

```
background(255) #white
```

This should erase all the rotating triangles except for one~~Now we have a rotating triangle~~, so we just have a single equilateral triangle on the screen.

All we have to do is put 90 of them in a circle, just like we did earlier in ~~the Geometry~~this chapter, using the “**`rotate`**`()` function.

”Exercise 5-5: Spin Cycle

Create an equilateral triangle in a Processing sketch and rotate it using the `rotate` function.

Exercise 5-6: Triangulation

Write a function called “tri” that will take a variable “length” and draw an equilateral triangle using the above points.

~~**Exercise 5.7: Whole Lot of Rotating Going On**~~

~~Create a sketch of 90 rotating triangles in a circle.~~

~~Solution~~Drawing Multiple Rotating Triangles

Now that we've learned how to rotate a single equilateral triangle, we need to figure out how to arrange multiple equilateral triangles into a circle. This is ~~basically~~similar to what you created while ~~the same sketch as the~~ rotating squares, but will ~~exploration~~.,~~This time, we'll do this but~~ useing our "`tri`" function. Enter the code in Listing 4-X in place of the `def draw()` section in Processing and run it. ~~:and the following code~~

```
def draw():
    global t
    background(255)#white
    translate(width/2,height/2)
    for i in range(90):
        #space the triangles evenly
        #around the circle
        rotate(radians(360/90))
        pushMatrix() #save this orientation
        #go to circumference of circle
        translate(200,0)
        #spin each triangle
        rotate(radians(t))
        #draw the triangle
        tri(100)
        #return to saved orientation
        popMatrix()
    t += 0.5
```

First, we use the for loop to arrange 90 triangles around the circle, making sure they're evenly spaced by dividing 360 by 90. Then we use `pushMatrix()` to save this position before ________

However, if you run this, you'll notice the triangles aren't transparent, like in Aronsen's sketch. In order to make the triangles transparent, w~~W~~e need to add this line to our tri function:

```
noFill()
```

Now we have 90 rotating transparent triangles but they're all rotating in the same way. We need to make each one rotate a little differently from the~~ir next-door neighbors~~adjacent ones to make the interesting pattern rather than ....

Phase Shifting the Rotation

We can change the pattern that the triangles rotate in~~do that~~ with a ***phase shift***, which ______________________~~.~~ Each triangle was assigned a number in the loop, represented by ~~:~~ **`i`**. ~~We only need to a~~We need to ~~A~~add~~ing~~ i ~~that~~ to **`t`** in the ~~"~~**`rotate(radians(t))`**~~"~~ function~~line~~, like this __________________________:

```
rotate(radians(t + i))
```

~~We'll~~You should see ~~what's~~something like ~~in this~~Figure 5~~-~~.22:

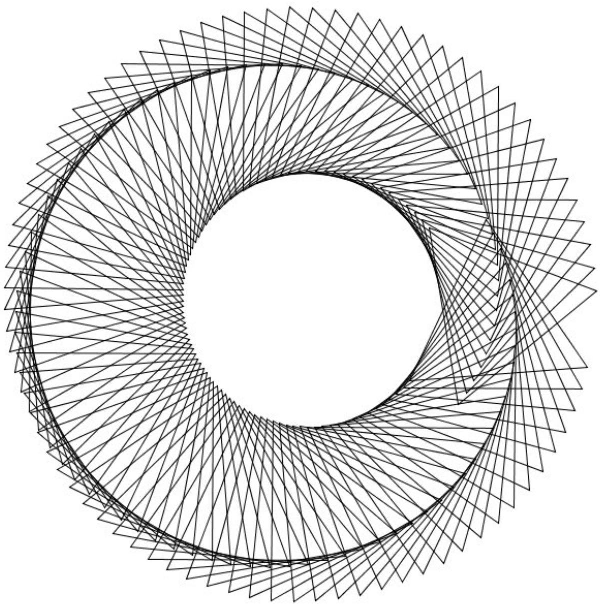


Figure 5~~-~~.22: Rotating t~~T~~riangles with p~~P~~hase s~~S~~hift

Notice~~You'll notice~~ there's a break in the pattern on the right side of the screen. This break in the pattern is caused by ________________________. ~~If wSince~~We want a nice, seamless pattern, so we have to make the phase shifts add up to a multiple of 360 degrees to complete the circle. ~~Since~~Because there are 90 triangles in the design, we'll divide 360 by 90 and multiply that by **`i`**:

```
rotate(radians(t + i*360/90))
```

~~I~~it's easy to work out that ~~know~~ 360/90 is 4, but ~~there's a reason~~ I'm leaving ~~it~~ the calculation in because we'll need it for something a little later~~for a reason, which I'll explain later~~. For now ~~we're~~this should ~~able to see the~~create a nice seamless pattern~~ we want~~, as ~~you can see~~shown in Figure 5~~-~~.23:

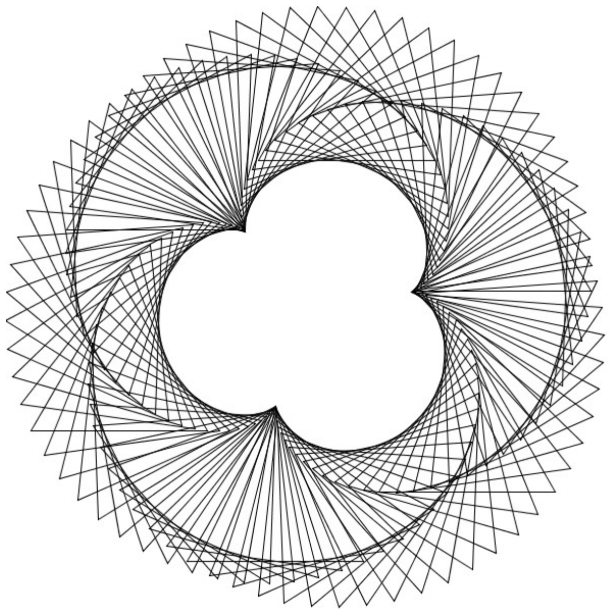


Figure 5~~-~~.23~~:~~ ~~Getting Closer!~~ Seamlessly rotating triangles with phase shift

By making our phase shifts add up to a multiple of 360, we were able to remove the break in the pattern.

Finalizing the Design

To make the design look more like the one in Figure 4-XX, we need to ….

~~isIt's easy to change the look of the sketch design~~We XXX to the design by simply changing the phase shift to ~~. Try~~ multiply~~ying~~ `_i` by 2, which will XXX ~~a number~~, ~~like this~~Change the rotate() line in your code to the following:

```
rotate(radians(t + 2*i*360/90))
```

Making this change and run the code, ~~A~~as you can see in Figure 5-24, our design ~~tThis~~ now looks very close to ~~the~~Roger Antonsen’s ~~sketch~~design we were trying to ~~copy~~recreate~~! See Figure 5.24~~:

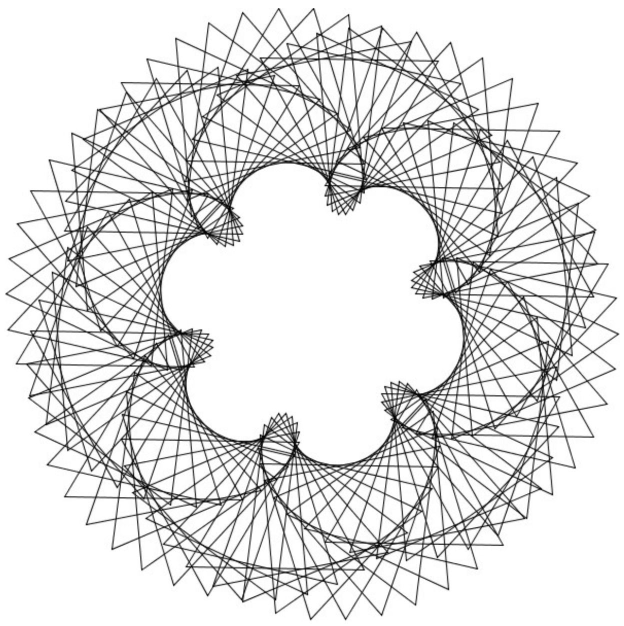


Figure 5-~~.~~24: Recreation of Roger's "90 Rotating Equilateral Triangles" seen in Figure 5-15~~ They Still Spinnin'!~~

Now that you've learned how to recreate a complicated design like this, try the next exercise to test your transformation skills!

Exercise 5.7: Whole Lot of Rotating Going On

Create a sketch of 90 rotating triangles in a circle.

Summary

In this chapter you learned how to break down some complicated-looking designs into their simple components. You learned how to draw shapes like circles, squares, and triangles, and arrange them into different patterns using Processing's built-in transformation functions. You also learned how to make your shapes dynamic by animating your graphics, adding color. Just like how Nasrudin's house in the epigraph was just a collection of bricks, the complicated code examples in this chapter were just a collection of simpler instructions.

In the next chapter you'll build on this chapter and learn new functions to create even more complicated behaviors, like leaving a trail and creating any shape from a bunch of vertices.