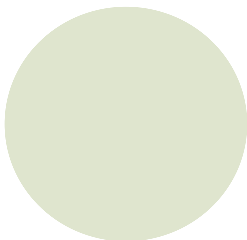
Learn Python Visually (Sample Chapter) © 12/09/20 by Tristan Bunn

9

FUNCTIONS AND PERIODIC MOTION



As your programs grow more complex, your line counts will increase, and you'll begin repeating the same or similar code. By using *functions*, you can divide your programs into named blocks of reusable code. This makes your code more modular, allowing you to reuse lines without needing to rewrite them.

You've already used many Processing functions, like `size()`, `print()`, and `rect()`, and in this chapter, you'll learn how to define your own functions. As an example, Processing has no function for drawing diamonds, but you can create one. You decide what to name this function and what arguments it will accept. Perhaps your `diamond()` function accepts an x, y, width, height, and optional rotation argument.

You'll also create functions for generating elliptical and wave-type motion, which will involve delving into some trigonometry. You'll incorporate the mathematical functions sine and cosine by using Processing's built-in functions for performing these calculations. If the mention of

trigonometry triggers disturbing flashbacks from math class, take a deep breath and relax. This will be a practical and visual reintroduction to these concepts, with Processing crunching all the numbers for you.

Defining Functions

Sensibly named functions make your code easier to understand and work with. A 1,000-line program can be tricky to comprehend, especially for somebody who didn't write it.

Imagine programming a music player. You might create a function named `play()` that executes 20 or so lines of code necessary to load and play an MP3 file. When you need to play a track, you simply call your `play()` function by using a file argument, like `play('track_1.mp3')`. You don't need to concern yourself with the details of how the `play()` function operates after you've defined it, and neither does anybody else working with your code. Additionally, you could define functions for `stop()`, `pause()`, `skipBack()`, and `skipForward()`.

In this section, you'll learn to define functions with the `def` keyword and then how to handle arguments. You might call these *user-defined functions* to distinguish them from those that come built-in with Python and Processing.

Creating a Simple Speech Bubble Function

Let's begin with a simple function that takes no arguments and draws speech bubbles, like the ones you find in comic strips, in the console. You've already used functions that work without arguments, like Processing's `noFill()` that relies on just a function name and parentheses. Conversely, a function like `fill()` requires at least one argument, such as a hexadecimal color value.

Your speech bubble function will form an outline, using plaintext characters, that surrounds a caption. Once you have this working, you'll move on to defining a more dynamic function that accepts a range of arguments to draw speech bubbles in the display window.

Create a new sketch and save it as *speech_bubbles*. Add the following code that prints a question in the console, followed by the answer in a speech bubble five seconds later:

```
wait = 5000
❶ print('1. What do you get if you multiply six by seven?')
❷ delay(wait)
print(' ------------------ ')
print('| The answer is 42! |')
print('| ----------------- ')
print('|/')
```

Learn Python Visually (Sample Chapter) © 12/09/20 by Tristan Bunn

When you run the sketch, you should see the question appear in your console immediately ❶. The Processing `delay()` function halts the program for 5,000 milliseconds (five seconds) ❷, then reveals the answer in a speech bubble using the four print lines that follow it. Run the sketch to confirm this:

```
1. What do you get if you multiply six by seven?
 -------------------
| The answer is 42! |
| ------------------
|/
```

This might not look like the most convincing speech bubble, but it'll do for now.

Make the following changes to your code to define a function for printing the answer:

```
wait = 5000

def printAnswer():
    print(' ------------------- ')
    print('| The answer is 42! |')
    print('| ------------------ ')
    print('|/')

print('What do you get if you multiply six by seven?')
delay(wait)
printAnswer()
```

The `def` keyword defines a new function. You can name this function whatever you like, but make the name descriptive. Like variable names, function names should contain only alphanumeric and underscore characters, and they must start with a letter or an underscore; in this case, I've chosen `printAnswer`. Always include the parentheses and a colon at the end of the `def` line. The four `print()` lines are in the *body* of the function definition, which is the indented section of code beneath the `def` line. The function won't execute the print lines until you *call* it. On the last line, where the program must reveal the answer, is the `printAnswer()` function call.

NOTE *Python will process your code line by line, beginning at the top of the file. If it attempts to execute a function call before it has processed the corresponding definition, the program will fail. In other words, you cannot call* `printAnswer()` *on the first few lines of your sketch, because Python would not yet have encountered the* `def printAnswer()` *line.*

When you run the sketch, the program should work as before, printing the question followed by the answer in a speech bubble five seconds later.

STYLE GUIDES

A *style guide* is a document that contains rules for writing code. This typically includes guidelines on how to indent code, where to use blank lines, what comments should look like, and how to name variables and functions.

If a team of programmers adheres to an agreed-upon style guide, their collaborative project code should turn out looking clean, consistent, and well formatted—as if one person wrote it. This kind of code is easier to modify and maintain, in part, because it's more readable. When you're adding features to an existing program, you'll often spend more time reading and comprehending code than writing it.

Some teams devise their own style guides, while others make use of or expand upon an existing guide. *PEP 8* is considered the de facto style guide for Python; you can access it at *https://www.python.org/dev/peps/pep-0008/*. The document covers many aspects of the Python language you've yet to encounter, and it's an excellent resource for any Python programmer.

The PEP 8 style guide recommends that "function names should be lowercase, with words separated by underscores as necessary to improve readability." In other words, the `printAnswer()` function instead should be named `print_answer()`. However, when an existing style is established, internal consistency is usually preferred.

I've opted for a camelCase function name to match the convention used for Processing's built-in functions, like `noFill()` or `pushMatrix()`. As noted in Chapter 1, camelCase combines multiple words into one and uses a capital letter to start the second and subsequent words. The style is also referred to as *mixedCase*, or sometimes *lowerCamelCase* (to contrast it with *UpperCamelCase*).

Add a second question to the end of your sketch:

```
. . .
delay(wait/2)
print('2. How many US gallons are there in a barrel of oil?')
delay(wait)
printAnswer()
```

After displaying the answer to question 1, the program waits two and a half seconds and prints question 2. The answer to question 2 is revealed five seconds after this. Once again, the answer is 42, but there's no need to retype the four lines of code for displaying the speech bubble. Instead, you can call the `printAnswer()` function a second time.

You can add as many questions as you like. If the answer to each question is 42, you can call the `printAnswer()` function to display the answer. If you want to restyle *all* of your speech bubbles—for example, using different characters for the outline—edit the body of the `printAnswer()` definition. You need to change the code in only one place to affect every speech bubble.

For each answer, you have a neat, one-line function call with a name that indicates what it does. Other programmers won't need to understand the inner workings of the `printAnswer()` function to use it, but if necessary, they can read through the definition code to find out how it works.

Before proceeding to the next section, set the `wait` value (at the top of your code) to **0**:

```
wait = 0
. . .
```

This change cancels the effects of the `delay()` functions, because a delay time of zero means there is no delay. As a result, your sketch doesn't pause, and the next section of code you add can run immediately.

The `printAnswer()` function is limited to drawing speech bubbles in the console, and it always prints the same answer of 42, so next, you'll define a function that can accept arguments.

Drawing Compound Shapes Using a Function

To define a function that draws speech bubbles with shapes and text in the display window, continue working in your *speech_bubbles* sketch. First, you'll need an image over which to place your speech bubbles.

I've chosen Jan van Eyck's *Arnolfini Portrait* for this example because the painting has three speech bubble candidates: a man, a woman, and a dog. It's also public domain. Figure 9-1 presents the original painting on the left, and the result you're working toward on the right.

Figure 9-1: The original Arnolfini Portrait*, 1434 (left); a version with speech bubbles (right)*

You can download the *Arnolfini Portrait* image from Wikipedia (*https://en.wikipedia.org/wiki/File:Van_Eyck_-_Arnolfini_Portrait.jpg*); the 561 × 768 pixel resolution will suffice. If you want to use a different image, that's fine too; just choose one with at least three subjects.

Create a new *data* subfolder and add your image to this; then add the following code to load and display it:

```
. . .
size(561, 768)
art = loadImage('561px-Van_Eyck_-_Arnolfini_Portrait.jpg')
image(art, 0, 0, width, height)
```

If you're not using the *Arnolfini Portrait*, adjust the `size()` and `loadImage()` arguments accordingly.

Run the sketch to confirm that the image spans your display window.

Define and then call a new speech bubble function by adding this code to the end of your sketch:

```
. . .
def speechBubble():
    x = 190
    y = 150
    txt = 'Check out my hat!'
    noStroke()
    pushMatrix()
    translate(x, y)

    # tail
    fill('#FFFFFF')
    beginShape()
    vertex(0, 0)  # tip
    vertex(15, -40)
    vertex(35, -40)
    endShape(CLOSE)

    # bubble
    textSize(15)
    by = -85
    bw = textWidth(txt)
    pad = 20
    rect(0, by, bw+pad*2, 45, 10)
    fill('#000000')
    textAlign(LEFT, CENTER)
    text(txt, pad, by+pad)

    popMatrix()

speechBubble()
```

If you're using a different image, adjust the `x`, `y`, and `txt` variables. The `x` and `y` variables control the position of the speech bubble—specifically, the x-y coordinate for the tip of the "tail" that's attached to the bubble. Before

drawing anything, a `translate()` function repositions the drawing space so that the vertex coordinates for this tip are (0, 0); the other tail vertices and the bubble are positioned relative to this point.

The `txt` variable defines the text that appears within the bubble. You can use any `txt` string you like, but keep it short. The speech bubbles will not accommodate multiline captions.

The code beneath the `bubble` comment draws a rounded rectangular bubble above the tail. The `rect()` function includes a fifth argument (`10`) that controls the corner radius. The larger you make this value, the rounder the corners become. The result is a rounded rectangular speech bubble with a tail at its bottom left (Figure 9-2).

Figure 9-2: The tip of the speech bubble tail has an x-y coordinate of (190, 150).

You can call the `speechBubble()` function 100 times, but the visual result always appears the same because every speech bubble draws over the one before it, at the same size, with the same text, in the same position. But, if you modify the `x`, `y`, and `txt` variables each time you call the `speechBubble()` function, you can customize the x-coordinate, y-coordinate, and caption. You can accomplish this by adding parameters to your function definition that allow you to pass values to the function using different arguments in your function call.

Adding Arguments and Parameters

Now you'll edit your `speechBubble()` definition so that the function can accept three arguments, allowing you to pass your coordinate and caption values to the function to manipulate the appearance of each speech bubble you draw. Arguments are assigned to corresponding *parameters*, but more on those shortly.

Currently, three variables control the speech bubble's appearance: `x`, `y`, and `txt`. To control those variable values via arguments, adapt your function definition as follows:

```
. . .
❶ def speechBubble(x, y, txt):
      #x = 190
```

Learn Python Visually (Sample Chapter) © 12/09/20 by Tristan Bunn

```
    #y = 150
    #txt = 'Check out my hat!'
    . . .
❷ speechBubble(190, 150, 'Check out my hat!')
```

The definition parentheses now include three parameters: `x`, `y`, and `txt` ❶. A parameter is a placeholder for a value that's provided by way of an argument. These parameters are made available within the local scope of the function; in other words, Python can access `x`, `y`, and `txt` only within the `speechBubble()` function block. You need to comment out (or delete) the old `x`, `y`, and `txt` lines to avoid overwriting the values that you pass in with the function call ❷.

Because you have three parameters, you must provide three arguments when you call the `speechBubble()` function. The first argument of `190` is assigned to parameter `x`, the second argument of `150` is assigned to parameter `y`, and so on, in the same order the parameters appear in the `def` line. These are called *positional arguments* because the order of the arguments determines which values are assigned to each parameter (Figure 9-3).

```
def speechBubble(x, y, txt):

        (190, 150, 'Check out my hat!')
```

Figure 9-3: Positional arguments

Run the sketch to confirm that the visual result is unchanged. Try testing different arguments to change the appearance of the speech bubble.

NOTE *It's not unusual to hear the terms* argument *and* parameter *used interchangeably. If you happen to mix them up, you aren't likely to confuse anybody.*

Call a second `speechBubble()` function:

```
. . .
speechBubble(315, 650, 'Woof')
```

The first and second (x and y) arguments position the speech bubble above the dog. The third argument specifies that the caption must read, "Woof" (Figure 9-4).

Learn Python Visually (Sample Chapter) © 12/09/20 by Tristan Bunn

Figure 9-4: A second speech bubble

You now have a working `speechBubble()` function that accepts positional arguments. However, you can also call this function by using arguments in an arbitrary order if you use keyword arguments.

Using Keyword Arguments

When you call a function, you can state explicitly which value belongs to which parameter by using *keyword arguments*. These arguments include both a keyword and value. Each keyword takes its name from a parameter in the function definition. Consider this example, where both lines produce the same result:

```
speechBubble(315, 650, 'Woof')           # positional arguments
speechBubble(txt='Woof', x=315, y=650)  # keyword arguments
```

The first `speechBubble()` call employs a positional argument approach. The second call uses keyword arguments; notice that each value has a keyword in front of it. Python uses the keywords in your function call to match values and parameters (Figure 9-5).

```
def speechBubble(x, y, txt):

        (txt='Woof', x=315, y=675)
```

Figure 9-5: Keyword arguments

This means you can order the arguments in your function call however you please. Just be sure to name your keywords exactly the same as the parameters in the function definition.

Setting Default Values

When you define a function, you can specify a *default value* for each parameter, which is like a backup Python can use if you leave out an argument in your function call. This behavior is useful for defining optional arguments. For example, the `rect()` function can accept an optional fifth argument for the corner radius. If you call the `rect()` function with four arguments, you get a rectangle with 90-degree corners, which is what users seem to want more often than not. But, if you provide the fifth argument (of something other than zero), you get a rectangle with rounded corners.

Use an equal sign to assign a default value to a parameter. For example, the following adds a default value of `'Hello'` to your `txt` parameter:

```
. . .
def speechBubble(x, y, txt='Hello'):
    . . .
```

The default `txt` parameter is a string, but you can use any data type you like, including numbers and lists.

You can now call the `speechBubble()` function using two positional arguments, leaving `txt` (the third argument) to rely on its default value:

```
. . .
speechBubble(445, 125)
```

The `445` and `125` are positional arguments for `x` and `y`. As there's no third argument, `txt` defaults to `'Hello'`, as per the function definition. The result (Figure 9-6) is a speech bubble positioned above the woman's head that reads, "Hello."

Figure 9-6: Drawing a speech bubble using the default `txt` *parameter,* `Hello`

To replace *Hello* with *Meh*, call the `speechBubble()` function using three arguments:

```
. . .
speechBubble(445, 125, 'Meh')
```

Because you provided the positional argument for the `txt` parameter, the woman's speech bubble will now read, "Meh."

The lady clearly isn't overly impressed with her partner's hat, so she might choose not to risk offending him. A *thought bubble* could be more appropriate (Figure 9-7).

Figure 9-7: A speech bubble (left) and a thought bubble (right)

To draw a thought bubble, modify the `speechBubble()` function to draw a chain of small circles instead of a triangular tail. However, you want the `speechBubble()` function to depict speech bubbles by default, as they are more common than thought bubbles.

Add an additional `type` parameter to the function definition:

```
. . .
def speechBubble(x, y, txt='Hello', type='speech'):
    . . .
```

Now you have two parameters with default values. Notice that these come after the parameters with no default values. If you're defining any function with default values, place those parameters at the end of the list.

The next step is to modify the function body, specifically the section beneath the `tail` comment. The `type` parameter must determine whether Processing should draw a triangular tail or a chain of circles. Modify the code as follows:

```
    . . .
    # tail
    if type == 'speech':
        fill('#FFFFFF')
        beginShape()
        vertex(0, 0)  # tip
        vertex(15, -40)
        vertex(35, -40)
        endShape(CLOSE)

    elif type == 'thought':
        fill('#FFFFFF')
        circle(0, 0, 8)
```

Learn Python Visually (Sample Chapter) © 12/09/20 by Tristan Bunn

```
    circle(10, -20, 20)
  . . .
```

The `if` statement code will draw a triangular tail if the `type` parameter is equal to `'speech'`, the default value assigned in the function definition. The `elif` statement will draw a chain of two circles whenever the function call includes a `type` argument of `'thought'`. Edit your function call to see this in action:

```
. . .
speechBubble(445, 125, 'Meh', 'thought')
```

The `thought` argument switches the `speechBubble()` function to "thought bubble mode." If you omit this argument, the function defaults to drawing the speech bubble with the tail. Run the sketch to confirm that the result matches Figure 9-7.

Mixing Positional and Keyword Arguments

You can use positional arguments for your `x` and `y` coordinates, leave out the `txt` argument, and include a keyword argument for `type`. This way, Python can utilize the default value for `txt` (`'Hello'`), but render it in a thought bubble. As an example, you might want to replace the dog's speech bubble with a thought bubble that reads, "Hello." One option is to include a third argument of `'Hello'` explicitly in the function call—a fully positional approach. For example:

```
speechBubble(315, 650, 'Hello', 'thought')
```

Each argument here corresponds to a parameter. This seems redundant, though, given that `'Hello'` is the default value for parameter 3. If you just omit the `'Hello'` argument in your function call, Processing will draw a *speech* bubble with the word *thought* in it:

```
# a speech bubble that says, thought
speechBubble(315, 640, 'thought')
```

Recall that the third positional argument is for the `txt` parameter and that leaving out the fourth argument means Python has to adopt the default value for the fourth `type` parameter (speech bubble mode). A simple solution to this problem exists, however; use a keyword argument instead of relying on a positional argument:

```
speechBubble(315, 650, type='thought')
```

In this case, you've explicitly stated that the value `'thought'` belongs to the `type` parameter. You might notice that you can arrange the arguments in any order if you use keyword arguments for every value. This is true, so decide what combination of positional and keyword arguments works best in a particular situation.

If you're missing one or many required arguments in a function call, Processing displays an error message (Figure 9-8). For example, if you call the `speechBubble()` function with no arguments, the error message indicates that you require at least two.

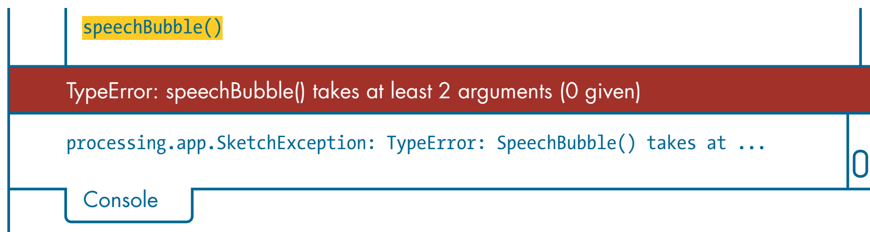


Figure 9-8: An error message for missing arguments

If you provide too many arguments, the error message indicates that `speechBubble()` takes at most four arguments.

Returning Values

You can use a function to operate on data and then have it *return* the result to the main program. This is different from the functions you've created so far, which execute a predefined section of code before resuming the regular flow of the main program.

To help explain this difference, here's some code to contrast a function that returns a value with one that does not:

```
x = random(100)
square(x, 40, 20)
```

Two Processing functions are in use here: `random()` and `square()`. The first one returns a value; the second does not. The `random()` function generates a floating-point value ranging from 0 up to but not including 100. The random function *returns* the value, which is assigned to a variable named `x`. The `square()` function draws a square in the display window; it does not return a value.

To define your own function that returns a value, use the `return` keyword. As an example, create a new function named `shout()`. This function accepts a single string argument, and then converts this string to uppercase and adds three exclamation marks to the end. Enter the following code above your `speechBubble()` calls to ensure that the `shout()` definition precedes any `shout()` function call:

```
. . .
def shout(txt):
    return txt.upper() + '!!!'
. . .
```

In the return line, the `upper()` method converts the string assigned to `txt` to uppercase; the final result is a concatenation of this and three exclamation

Learn Python Visually (Sample Chapter) © 12/09/20 by Tristan Bunn

marks. Once Python processes the `return` statement, it exits the function immediately. In other words, if you add any further code to the `shout()` definition below the `return` line, Python ignores it.

You could use this function to add emphasis to the text in any speech bubble. Here's an example:

```
speechBubble(190, 150, shout('Check out my hat'))
```

The `shout()` function converts the string to "CHECK OUT MY HAT!!!" before it's passed to the `speechBubble()` function. This wraps the argument with `shout()` to avoid having to create an intermediate variable, which you would then pass to the `speechBubble()` function.

This was a simple example to introduce how the `return` keyword works. Many functions that return values perform more complex tasks, like Processing's `sqrt()` function that calculates the square root of any number.

Defining Functions for Periodic Motion

In this section, you'll learn how to simulate periodic motion in Processing by defining functions that employ trigonometry to draw circular patterns and waves. In physics, *periodic motion* is motion that repeats itself at regular intervals, such as a swinging pendulum, waves moving through water, or the moon orbiting the Earth. A *cycle* is one complete repetition of the motion. The *period* is the time it takes to complete a cycle. The period for the moon's orbit of the Earth is roughly 27.3 days; the second hand of a clock has a period of 60 seconds.

Trigonometry, or *trig*, is a branch of mathematics that studies triangles and uses various mathematical functions, such as sine and cosine, to calculate angles and distances. It also has applications in many fields of programming. For instance, games that incorporate physics must continuously calculate the position and speed of objects in motion, and those calculations involve triangles.

Trig is also useful for controlling steering and aiming behavior. For example, if you know the x-y coordinates of the player and enemy turret in Figure 9-9, you can calculate how to rotate the enemy gun to aim it at the player.

You'll use right triangles to calculate points along the circumference of a circle, using sine and cosine functions. The coordinates for those points are what you use to simulate smooth, periodic motion.

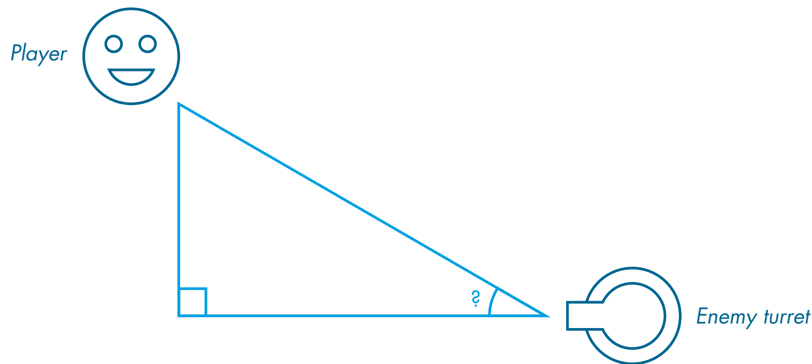


Figure 9-9: If only the enemy turret had listened in math class.

Create a new sketch and save it as *periodic_motion*. Add the following code to set up the drawing space:

```
def setup():
    size(800, 600)

def draw():
    background('#004477')
    noFill()
    strokeWeight(3)
    stroke('#0099FF')
    line(width/2, height, width/2, 0)
    line(0, height/2, width, height/2)
    # flip the y-axis
    scale(1, -1)
    translate(0, -height)
    # reposition the origin
    translate(width/2, height/2)
```

The preceding code structures an animated sketch by using `setup()` and `draw()` functions with two (pale blue) lines that intersect at the center of the display window. The y-axis is flipped, so y-coordinates decrease as you move downward; I'll elaborate on why I did that soon. The final `translate()` function shifts the coordinate system so that the origin (0, 0) sits in the center of the display window. This means that the x-coordinate for the left edge of the display window is –400, and the x-coordinate for the right edge is 400. The y-coordinate for the top edge is 300; for the bottom edge, it's –300 (Figure 9-10). The modified coordinate space, with its flipped y-axis, now behaves like a regular *Cartesian plane*, with four quadrants that allow you to plot any x-y coordinates ranging between (–400, –300) and (400, 300).

You've likely encountered this system in math classes before, which is why I've set up the coordinate space this way. You'll use it as a platform to experiment with elliptical and wave motion, but first, you may require a brief refresher on trigonometric functions.

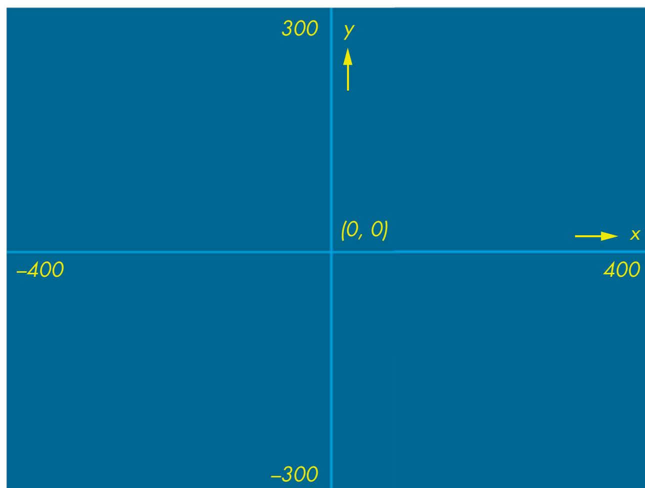


Figure 9-10: The Cartesian plane with four quadrants

An Introduction to Trigonometric Functions

Sine, *cosine*, and *tangent* are three common *trigonometric functions*. These are mathematical (as opposed to programming) functions, but you can use them in Python thanks to Processing's built-in trig functions. Sin, cos, and tan—as they are often abbreviated—are based on ratios obtained from a right triangle (Figure 9-11). A *right triangle* (or *right-angled triangle*) has one angle that measures exactly 90 degrees, usually denoted by a small square. The θ symbol, *theta*, is commonly used to represent an unknown angle.

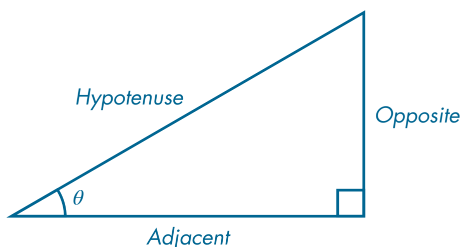


Figure 9-11: A right triangle

You can calculate the size of theta if you know the lengths of any two sides of this triangle. Depending on the lengths you have, you'll use either sin, cos, or tan for the calculation. *SOHCAHTOA*, pronounced phonetically as *so-ka-toe-uh*, is a handy mnemonic device to help you remember the following trigonometric ratios:

SOH $\sin(\theta)$ = opposite / hypotenuse

CAH $\cos(\theta)$ = adjacent / hypotenuse

TOA $\tan(\theta)$ = opposite / adjacent

As an example, if you know the length of the opposite and hypotenuse in Figure 9-11, you can find angle theta by using sin(θ). If you know the length of the adjacent and hypotenuse, use cos(θ). You can also rearrange these equations to find the length of an unknown side in cases when you know theta and one length. I'll return to this point shortly.

You'll apply sin and cos to a simple example to determine an x-y coordinate along the perimeter of a circle. To begin, draw a circle with its center positioned at (0, 0) with a radius of 200. Add a line starting at (0, 0) that's the same length as the circle radius and rotated 1 radian:

```
. . .
radius = 200
theta = 1

def draw():
    --snip--
    circle(0, 0, radius*2)
    stroke('#FFFFFF')
    pushMatrix()
    rotate(theta)  # approximately 57.3 degrees
    line(0, 0, radius, 0)
    popMatrix()
```

The code renders the circle in a pale blue outline. A white line the length of the radius extends from the center of the circle to its perimeter; this forms an angle of 1 radian (equal to roughly 57.3 degrees), as labeled in Figure 9-12. Notice that the `rotate()` function applies counterclockwise to the line because the y-axis is inverted. The task is to work out the x-y coordinate for the point where the white line connects to the circle perimeter, labeled A. The other yellow markings reveal the right triangle upon which you'll base your calculations.

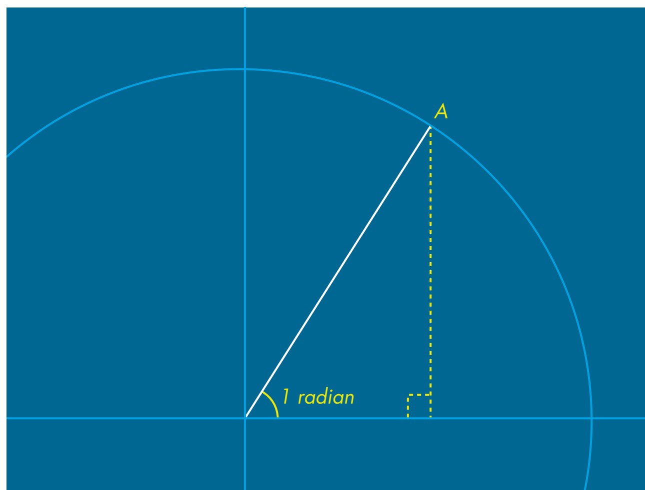


Figure 9-12: You'll find the x-y coordinate for the point labeled A.

Observe that the y-coordinate for point A is equal to the length (or height) of the opposite side. You know the angle (`theta` variable) and the length of the hypotenuse (`radius`), which you can use to calculate the length of the opposite. Recall that the *SOH* in *SOHCAHTOA* stands for *sin(θ) = opposite / hypotenuse*.

You have the values for θ and *hypotenuse*, so rearrange the equation to isolate *opposite*: *opposite = sin(θ) × hypotenuse*.

If you substitute the placeholders with the variable names in your program, this is `y = sin(theta) * radius`.

To calculate the x-coordinate for point A, you need to find the length (or width) of the adjacent side. Recall that the *CAH* in *SOHCAHTOA* stands for *cos(θ) = adjacent / hypotenuse*, which you can rearrange as `x = cos(theta) * radius`.

Add the following code to the end of your `draw()` function:

```
. . .
# white dot
noStroke()
fill('#FFFFFF')
x = cos(theta) * radius
y = sin(theta) * radius
circle(x, y, 15)
```

The `cos()` and `sin()` functions return floating-point values ranging from –1 to 1 for various values of `theta`. Processing's trig functions work with radians, so there's no need to convert the `theta` argument to degrees. In this case, theta is equal to 1 radian, and the `cos()` and `sin()` functions return values of 0.54 and 0.84, respectively (rounded to two decimal places). When you multiply 0.54 and 0.84 by the `radius` value of 200, you get an x-y coordinate of (108, 168). The `circle(x, y, 15)` function renders a white dot by using this x-y coordinate pair. Run the sketch to confirm the position of the white dot at point A, where the white line connects to the circle boundary.

You can adjust the theta value to move the white dot to different points along the perimeter of the pale blue circle. To position the dot at 90 degrees, directly above the origin, use `theta = HALF_PI`; for 180 degrees, use `theta = PI`; and so forth. A `theta` value of `TAU` brings you back around to the starting point, visually indistinguishable from a dot at `theta = 0`. If `theta` is greater than `TAU`, there's a wraparound effect. In other words, `cos(TAU+1)` is equivalent to `cos(1)`.

The next task is to get the dot moving. You don't need the white line anymore; remove it by deleting the lines starting from `pushMatrix()` up to and including `popMatrix()`.

Circular and Elliptical Motion

You'll begin by moving the dot along a circle perimeter (a circular motion), and you'll create a user-defined function for handling the necessary math. You'll then use this same function to create a spiral variant of the circular

motion. Once you have the circular and spiral motions working, you'll define a new function for elliptical motion. Figure 9-13 depicts examples of each motion.

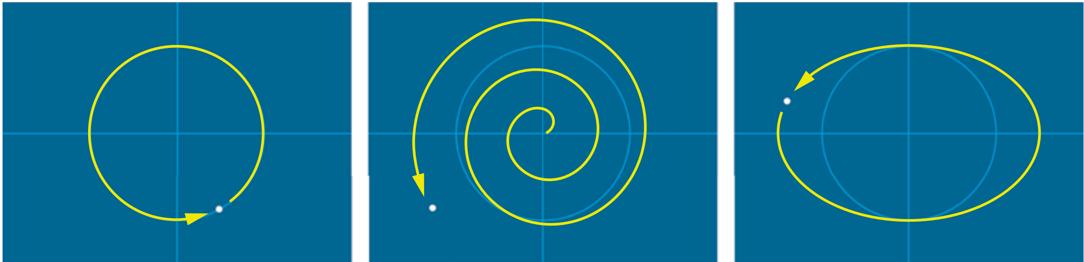


Figure 9-13: Circular (left), spiral (middle), and elliptical (right) motion

Circles

Recall that the size of angle theta, stored in a variable named `theta`, governs the position of the white dot. To make the dot move along the perimeter of the circle in a counterclockwise direction, add code to increment theta each time the draw function executes. Include a `period` variable to control the increment size:

```
. . .
period = 2.1

def draw():
    global theta
❶   theta += TAU / (frameRate * period)
    . . .
```

At the default `frameRate` of 60 fps, with a `period` of 2.1 seconds, the `theta` increment is equal to approximately 0.05 ❶. This means your angle extends 0.05 of a radian with each new frame. Run the sketch to test this out. The white dot should take about 2.1 seconds to complete a lap of the circle perimeter.

The larger the value you add to theta, the faster the dot will move. Subtracting from theta moves the dot in the opposite direction (clockwise).

Define a new function named `circlePoint()` for calculating points along the perimeter of a circle. In your `draw()` function, substitute the `x` and `y` lines with a `circlePoint()` function call:

```
def circlePoint(t, r):
    x = cos(t) * r
    y = sin(t) * r
    return [x, y]❶

. . .
def draw():
    . . .
```

```
    #y = sin(theta) * radius
    #x = cos(theta) * radius
    x, y❷ = circlePoint(theta, radius)
    circle(x, y, 15)
```

The circlePoint() definition includes two parameters: t for theta (the angle) and r for the radius. Because the function must calculate the x- and y-coordinates for some point along a circle perimeter, it needs to return two values. Use a list to return more than one value ❶; you could also use a dictionary (or a tuple).

When you call the function, Python can *unpack* the list values and assign them to multiple variables. To invoke this unpacking behavior, provide a corresponding variable for each list item, separating each variable with a comma. In this case, the function returns a list of two values, which are assigned to variables x and y ❷. Alternatively, you could assign the list to a single variable using something like a = circlePoint(theta, radius), but then you'd have to refer to x and y by using a[0] and a[1], respectively, which isn't as neat or descriptive.

Spirals

For an outward spiral motion (the center image in Figure 9-13), you can use a radius value that increases over time. Here's an example:

```
    . . .
    x, y = circlePoint(theta, frameCount)
    circle(x, y, 15)
```

Recall that frameCount is a system variable containing the number of frames displayed since starting the sketch. The radius argument (the frameCount) begins at 0 and grows larger as the animation progresses, causing the dot to move outward in a spiral motion. The dot gains speed as it moves away from the center of the display window because each full rotation maintains the same period, regardless of the circlePoint() radius. In other words, the dot must cover a larger distance in the same time, so it moves faster.

Ellipses

For an elliptical motion, you need two radii: one for the horizontal axis and one for the vertical axis. These radii control the width and height of the ellipse shape that guides the white dot's trajectory (see the right image in Figure 9-13). Define a new ellipsePoint() function with parameters for an angle, horizontal radius, and vertical radius:

```
def ellipsePoint(t, hr, vr):
    x = cos(t) * hr
    y = sin(t) * vr
    return [x, y]
. . .
```

The function body is similar to that of the `circlePoint()` function. The difference is that you multiply the `x` and `y` values by the `hr` (horizontal-radius) and `vr` (vertical-radius) parameters, respectively.

The following `ellipsePoint()` function call makes the dot move in an elliptical motion:

```
. . .
x, y = ellipsePoint(theta, radius*1.5, radius)
circle(x, y, 15)
```

The `ellipsePoint()` function's second argument (horizontal radius) is larger than the third argument (vertical radius), so the resulting ellipse is wider than it is tall.

Sine Waves

A *sine wave* is a geometric waveform that repeats itself periodically, like a continuous chain of S-shaped curves connected end to end. This waveform features in many mathematical and physical applications. For example, you can use sine waves to model musical tones, radio waves, tides, and electrical currents.

The shape of a sine wave is formed using a `sin()` function. Figure 9-14 depicts a yellow sine wave.

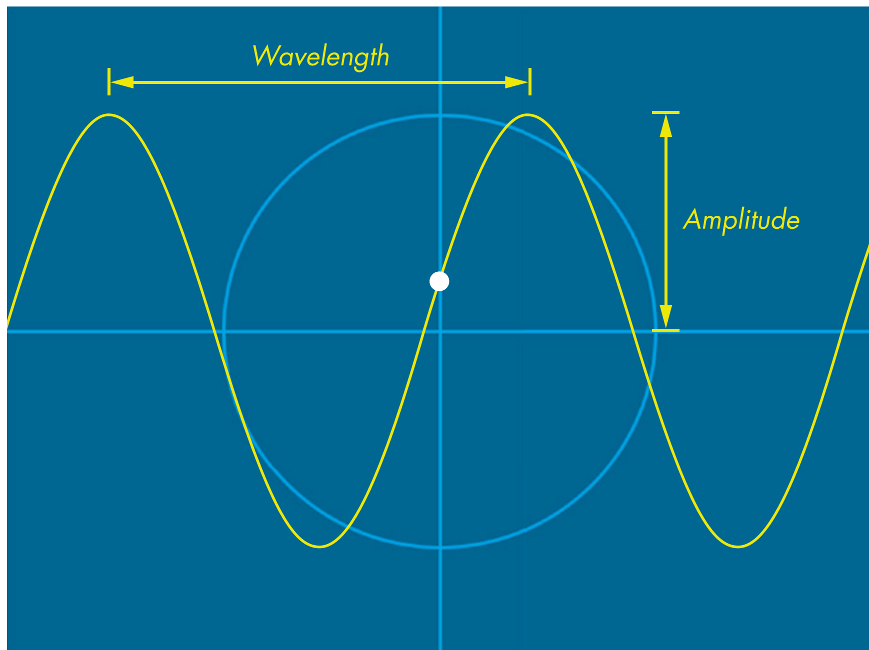


Figure 9-14: A sine wave

The *wavelength* is the length of one complete cycle, measured as the distance from crest to crest (or trough to trough). Wavelength is related to period, but period is a reference to time (taken to complete a cycle), and wavelength is a reference to distance.

The *amplitude* is the distance from the resting position (y = 0) to the crest. A wave with an amplitude of 0 would lie flat along the x-axis. You can determine that the yellow wave in Figure 9-14 has an amplitude of 200 by comparing it to the radius of the pale blue circle.

To simulate sine wave motion, add the following code to your *periodic_motion* sketch. This is the same as drawing a circle, but using a fixed x-coordinate:

```
. . .
def draw():
    . . .
    amplitude = radius
    y = sin(theta) * amplitude
    circle(0, y, 15)
```

The wave's `amplitude` is equal to the `radius` of the pale blue circle, although you can test any value you like. The y-coordinate for the white dot is calculated using `sin(theta)` multiplied by the amplitude; the x-coordinate is always 0. The result is a white dot that moves directly up and down from the origin.

Run the sketch and pay careful attention to how the dot is accelerating and decelerating, as if the wave shown in Figure 9-14 were passing through water with the dot floating on its surface. As the dot approaches a crest or trough, it begins to slow down, and then it accelerates after it makes a turn; it's moving fastest as it crosses the y-axis.

You can use this motion to draw a whole wave of moving dots or to simulate a weight hanging from a spring (Figure 9-15).

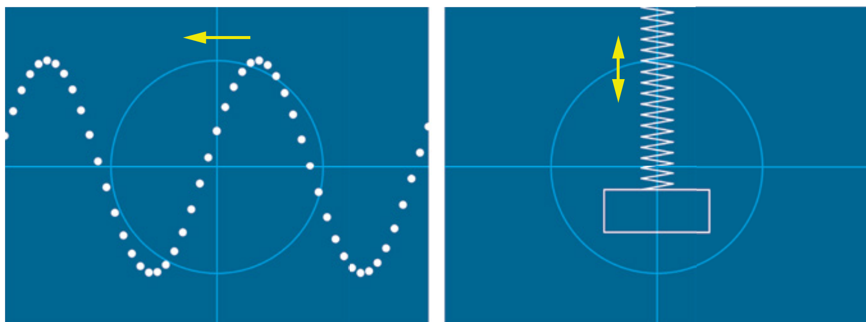


Figure 9-15: A wave of dots (left) and a weight hanging from a spring (right)

The code for each of these examples follows. You'll need to add it to the end of the `draw()` block of your *periodic_motion* sketch. You can add both code listings if you want to draw the spring and weight over the wave of dots, or instead replace one listing with the other.

Learn Python Visually (Sample Chapter) © 12/09/20 by Tristan Bunn

Drawing a Sine Wave of Dots

Use a loop to draw a whole wave of dots. There are 51 dots in all, equally spread along the x-axis. Each dot has a different y-coordinate based on a `theta` value that's incrementally larger than the dot preceding it:

```
amplitude = radius

for i in range(51):
❶  f = 0.125 * 2
    t = theta + i * f
❷  x = -400 + i * 16
❸  y = sin(t) * amplitude
    circle(x, y, 15)
```

The loop draws 51 dots, beginning at an x-coordinate of –400, at x intervals of 16 pixels ❷. The `y` value for each dot is calculated using a `theta` value that's `0.125 * 2` of a radian (or 0.25) ❶ larger than the neighbor to its left. You can change this multiplier to `1` for a single wave that spans the width of the display window, leave it as `2` for two waves (as in Figure 9-15), make it `3` for three waves, and so forth. I've named the variable `f`, for *frequency*, which refers to the number of times an event repeats itself in a fixed time period.

Wavelength is inversely proportional to frequency, so as you increase the frequency, you decrease the wavelength (and the waves begin to look spikier). The wave motion travels from right to left, but the horizontal positions of the dots don't change.

Simulating a Weight Hanging from a Spring

Use a loop to draw the spring, which is a shape composed of vertices. The weight dangling on the end of the spring is a rectangle. Adjust the fill and stroke to draw outlines instead of filled shapes:

```
amplitude = radius
y = sin(theta) * amplitude
noFill()
stroke('#FFFFFF')
strokeJoin(ROUND)
bends = 35

beginShape()
for i in range(bends):
    vx = 30 + 60 * (i % 2 - 1)
    vy = 300 - (300 - y) / (bends - 1) * i
    vertex(vx, vy)
endShape()

rect(-100, y-80, 200, 80)
```

The tight corners of the spring's bends will produce sharp joints, which result in elongated "elbows." Processing clips these when they get too long and sharp, but jumping between *mitered* (sharp) and *beveled* (clipped) joints

Learn Python Visually (Sample Chapter) © 12/09/20 by Tristan Bunn

makes the animation look bad. To prevent this, I've set the `strokeJoin` to `ROUND`. A loop is nested within the `beginShape()` and `endShape()` functions for plotting the zigzagging spring vertices.

Ordinarily, some energy is dissipated or lost in such a system, and the amplitude should decay over time. You could simulate this by reducing the (global) `radius` value every frame until it reaches 0, when the weight will come to a rest.

Now that you've learned how to return values from functions and incorporate trigonometry for elliptical and wave animation, let's look at a special curve created by combining waves.

Lissajous Curves

In this section, you'll create a function for drawing Lissajous curves controlled by arguments. A *Lissajous curve*—named after French physicist Jules Antoine Lissajous—is formed by combining x- and y-coordinates from two waves.

You can create these curves mechanically by setting up a Y-shaped pendulum with a sand-filled cup hanging at the end of it. As the cup swings about, sand drains through a hole at the bottom, drawing a curve. Figure 9-16 shows an example of this device (left) and an image of a curve drawn with sand (right). The point labeled *r* indicates where the pendulum merges into a single string. The ratio of the upper to lower section of the pendulum, and the angle and power of your initial swing, determine the shape of the resulting curve.

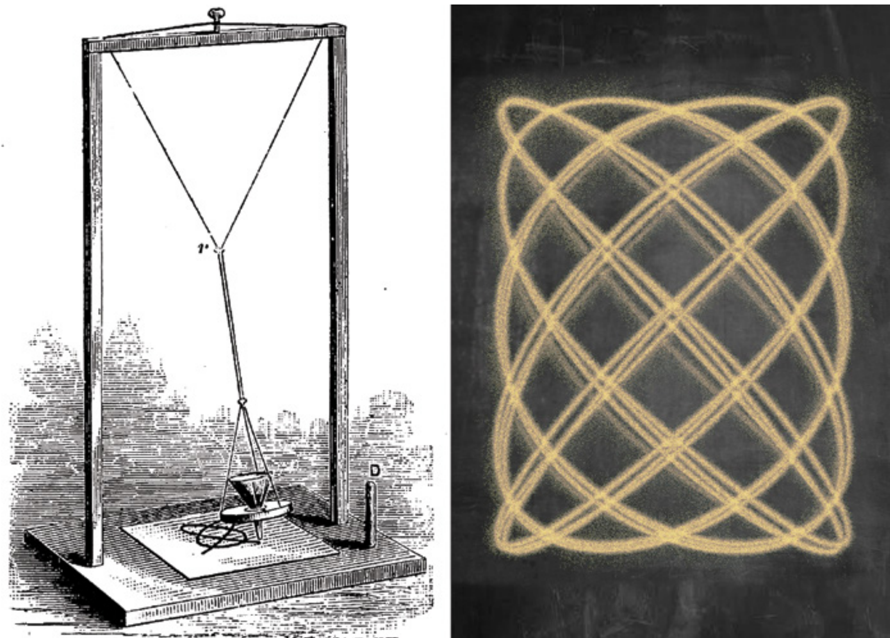


Figure 9-16: Blackburn's Y-shaped pendulum, from Sound *by John Tyndall, 1879 (left), and a Lissajous curve drawn with sand (right)*

To begin, suppose that you have two circles of different sizes (Figure 9-17). Circle A has a radius labeled A that is 200 units, and Circle B has a radius labeled B that is 100 units.

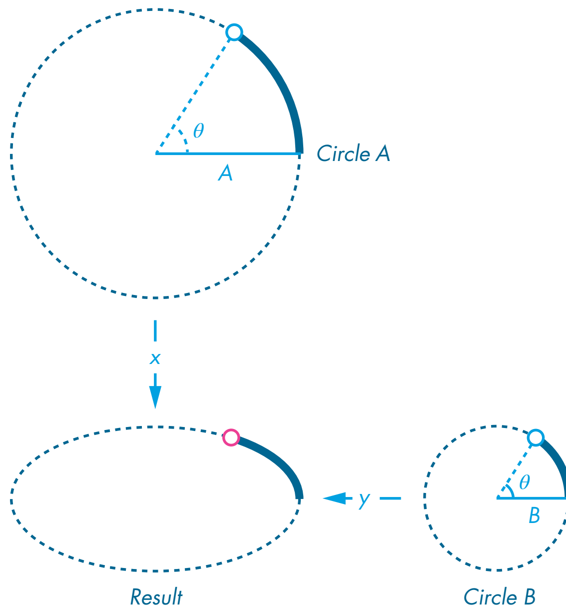


Figure 9-17: Combining x and y values from different circles to form an ellipse

The Result ellipse (lower left) is formed by using x-coordinates from Circle A and y-coordinates from Circle B. The ellipse turns out as wide as Circle A and as tall as Circle B. The math for this is relatively simple and uses what you already know about drawing ellipses with trigonometric functions.

To find the x-y coordinate for any point along the perimeter of the Result ellipse, you use the following:

$$x = cos(\theta) \times A$$

$$y = sin(\theta) \times B$$

Create a new sketch, save it as *lissajous_curves*, and add the following code to recreate the ellipse from Figure 9-17:

```
def lissajousPoint(t, A, B):
    x = cos(t) * A
    y = sin(t) * B
    return [x, y]

def setup():
    size(800, 600)
    frameRate(30)
    background('#004477')
    fill('#FFFFFF')
```

```
    noStroke()

theta = 0
period = 10

def draw():
    global theta
    theta += TAU / (frameRate * period)
    # flip the y-axis and reposition the origin
    scale(1, -1)
    translate(width/2, height/2-height)

    x, y = lissajousPoint(theta, 200, 100)
    circle(x, y, 15)
```

The drawing space is set up like your preceding sketch. You have an inverted y-axis, and the origin is shifted to the center of the display window. The `theta` value increments by approximately 0.01 each frame, which serves as the first argument in the `lissajousPoint()` function call. Right now, this function performs exactly the same operation as the `ellipsePoint()` function in your *period_motion* sketch—the only difference is the naming of the function and its variables.

Notice that there's no `background()` call within the `draw()` section of the code, so Processing won't clear each frame. Because of this, the moving white dot forms a continuous line. Run the sketch; it should draw a complete ellipse in a counterclockwise motion (Figure 9-18).

Figure 9-18: Drawing an ellipse by using the `ellipsePoint()` function

When theta reaches τ radians (~6.28), the oval is complete, and Processing continues to draw over the existing line. Even though the animation might appear complete, the dot is still moving along the perimeter.

The next step is to modify the `lissajousPoint()` function so that it can draw Lissajous curves (as opposed to ellipses). But first, consider what's happening here in terms of waves. Study Figure 9-19, which represents each circle as a wave, and take note of how the dots on each wave control the position of the dot along the ellipse's perimeter.

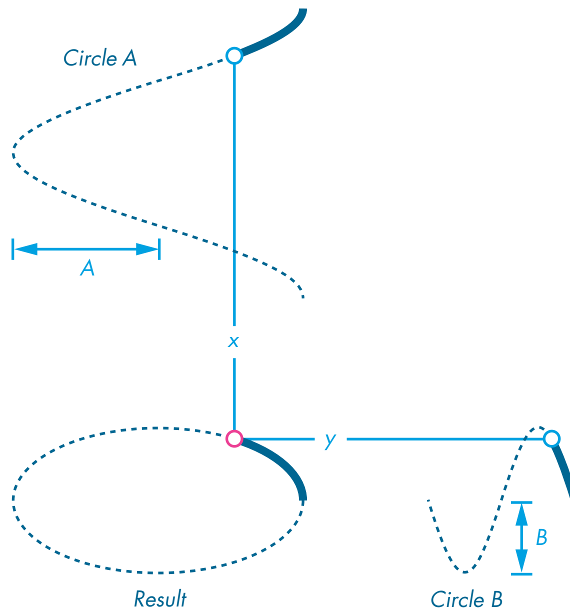


Figure 9-19: Circle A and Circle B represented in wave form

Figure 9-19 presents the x-coordinates of Circle A as a cosine wave that oscillates between –1 and 1, which is scaled by the circle radius (the wave amplitude) of A. Similarly, the y-coordinates of Circle B are presented as a sine wave with an amplitude of B.

In Figure 9-20, you can see how dots move along the waves to form the ellipse shape.

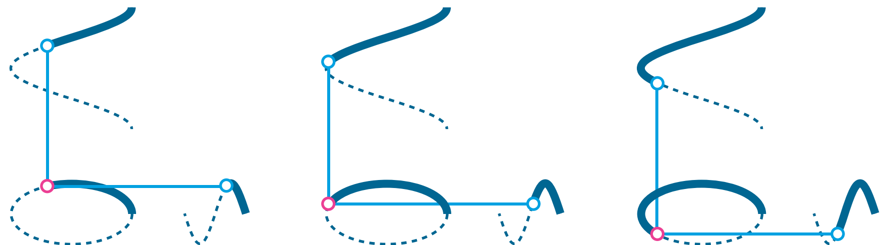


Figure 9-20: Theta = 2 (left), theta = 3 (middle), theta = 4 (right)

Currently, the frequencies of both waves match. In other words, it takes the same amount of time for each wave to complete a single cycle. The result is an ellipse.

Lissajous curves occur when the wave frequencies differ. In Figure 9-21, the frequency of the Circle B wave is twice that of the Circle A wave. The dot following the Circle B wave must complete two cycles in the same amount of time that the Circle A dot will complete one. The *a* and *b* values (lowercase) represent a frequency of 1 and 2, respectively.

Learn Python Visually (Sample Chapter) © 12/09/20 by Tristan Bunn

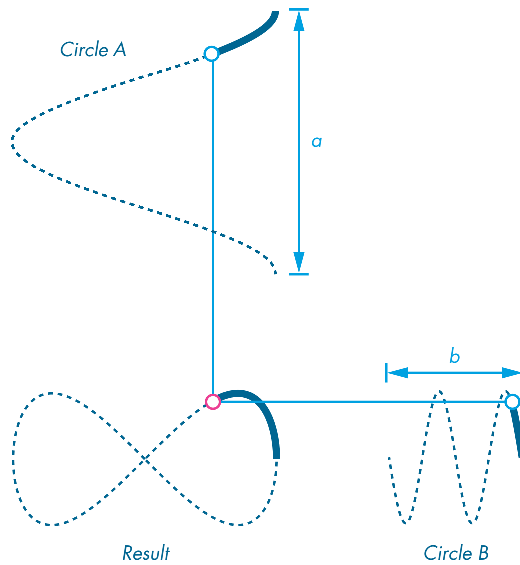


Figure 9-21: The Circle B wave has a frequency twice that of Circle A.

Frequencies *a* and *b* could be 3 and 6, 40 and 80, or 620 and 1,240. Any pair of numbers with a ratio of 1:2 will produce a ∞ shape. This will be important when you return to writing the code. You can think of this in another way as well: in Figure 9-17, the Circle B dot must always complete two journeys around the perimeter in the same amount of time that the Circle A dot completes one.

Figure 9-22 shows how the dots move along the waves to form the Lissajous curve.

Adapt your `lissajousPoint()` definition, adding a parameter for frequency *a* and frequency *b*. Use these two parameters as multipliers for theta (`t`) in your `x` and `y` lines, respectively:

```
def lissajousPoint(t, A, B, a, b):
    x = cos(t * a) * A
    y = sin(t * b) * B
    return [x, y]
    . . .
```

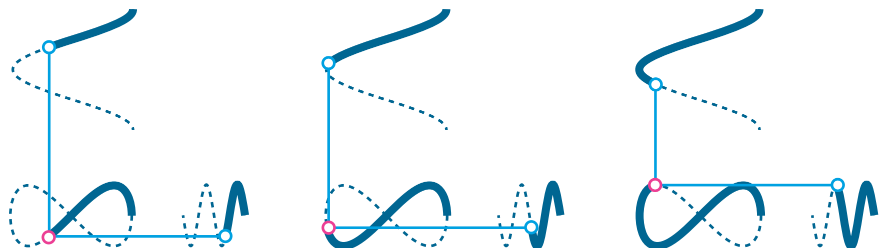


Figure 9-22: From left to right: theta = 2; theta = 3; theta = 4

Now, add arguments for parameters `a` and `b` to your function call:

```
x, y = lissajousPoint(theta, 200, 100, 1, 2)
```

Run the sketch and watch Processing draw a Lissajous curve (Figure 9-23).

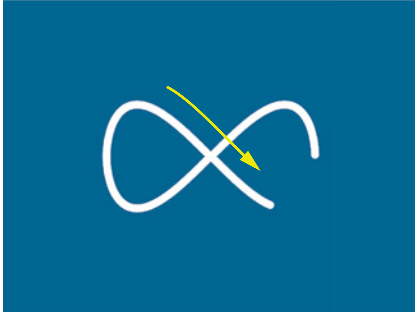


Figure 9-23: Drawing a Lissajous curve by using the `lissajousPoint()` function

The `a` and `b` arguments determine the number of horizontal and vertical "lobes" in the Lissajous curve. Recall that it's the ratio that matters, so `1`, `2` will produce the same curve as `5`, `10`. However, the latter pair will complete drawing the curve in less time, and even larger numbers will create discernible spacing between the dots (that would otherwise form a solid line). Figure 9-24 shows the results of a few `a`, `b` arguments. Try experimenting with other numbers.

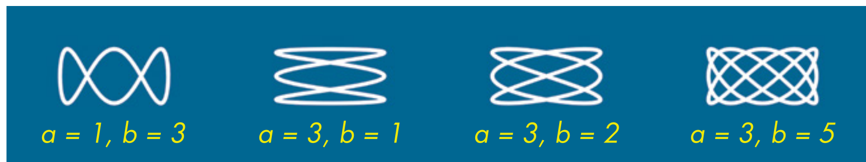


Figure 9-24: Drawing Lissajous curves using different a, b arguments

You can create intriguing visual patterns by moving shapes, points, and lines around with trigonometric functions. Simply experimenting, with no predefined idea of what you want to accomplish, can lead to impressive visual results. Think of this approach to coding like a musical jam session, where instrumentalists improvise until they stumble upon something that sounds good.

The next task uses Lissajous curves and a `line()` function for animated patterns, which should provide some interesting ideas for you to riff off.

Creating Screensaver-Like Patterns with Lissajous Curves

In Chapter 6, you programmed a simple DVD screensaver; now let's create a more elaborate one using Lissajous curves. The original purpose

of a screensaver was to "save" your screen. Older cathode-ray tube (CRT) monitors were susceptible to *burn-in*: if you displayed the same graphic in the same position for too long, it would leave a permanent "ghost" image. Modern displays aren't susceptible to burn-in, but many people still use screensavers because they look cool.

You'll use your `lissajousPoint()` function to create a pattern inspired by popular screensaver designs. Figure 9-25 shows the final result with lines and colors morphing smoothly as the pattern twists about the screen.

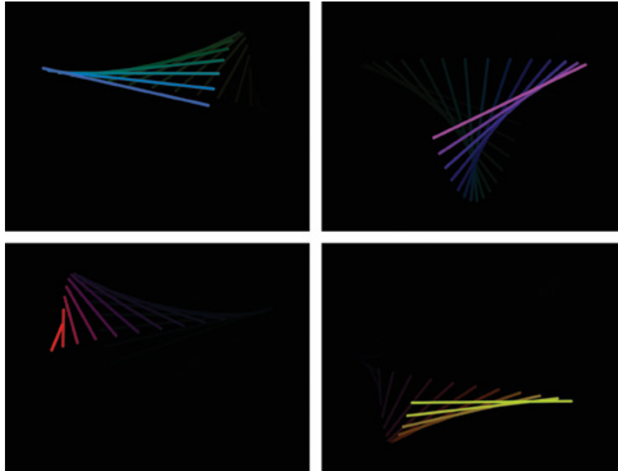


Figure 9-25: An animated pattern based on Lissajous curves

This movement relies on two Lissajous curves, using a `line()` function to draw a straight line between the leading tip of each curve. Figure 9-26 illustrates how this works.

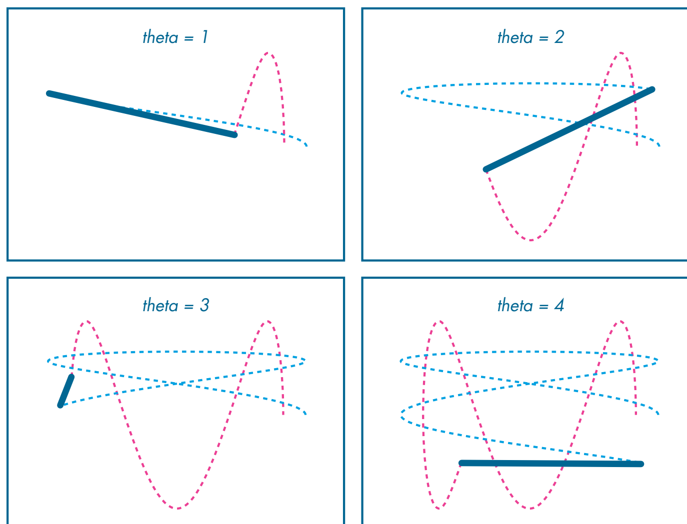


Figure 9-26: Drawing a straight line between two Lissajous curves

Of course, you don't see the curves, just the straight line, but it's two `lissajousPoint()` calls that are calculating the x-y coordinate for your `line()` function. When theta reaches τ radians, the Lissajous curves are complete and the motion repeats itself.

Add the following code to the end of the `draw()` function in your *lissajous_curves* sketch:

```
    . . .
❶   for i in range(10):
        # curves
        t = theta + i / 15.0
        x1, y1 = lissajousPoint(t, 300, 150, 3, 1)
        x2, y2 = lissajousPoint(t, 250, 220, 1, 3)
        # background color
      ❷ fill(0x55000000)
        noStroke()
        rect(-width/2, -height/2, width, height)
        # line
        colorMode(HSB, 360, 100, 100)
      ❸ h = (frameCount + i * 15) % 360
        strokeWeight(7)
        stroke(h, 100, 100)
        line(x1, y1, x2, y2)
```

The loop will draw 10 lines ❶ in all—one solid line leading a trail of nine lines that gradually fade behind it. You use two `lissajousPoint()` functions, one for each curve (that together define the x-y coordinate for each end of the line). With each iteration, Processing draws a semiopaque black square that spans the entire display window, dimming the lines of previous iterations.

To define a semiopaque color, you use Processing's `0x` notation ❷. The hexadecimal value is expressed with a leading `0x`, without quotes, using eight hexadecimal digits. The first two digits define the *alpha* (transparency) component; for example, `11` is highly transparent, and `EE` highly opaque. This example uses `55`, somewhere in between, but nearer the transparent side. The remaining six characters are your standard RGB hexadecimal mixture, in this case black (`000000`). For the stroke color, set the `colorMode()` to `HSB` (see "Color Modes" on page 15). For the first 360 frames, you can use `frameCount` to shift the hue value a single degree per frame. However, `frameCount` will soon exceed 360, so you use a modulo operation to "wrap around" back to 0 ❸.

Run the sketch to observe the output.

NOTE *Drawing so many semiopaque black rectangles over the display window each frame is a demanding operation for Processing to perform. If your computer is struggling, try setting a lower frame rate, or reducing the* `for` *loop iterations from 10 to a more manageable value.*

Learn Python Visually (Sample Chapter) © 12/09/20 by Tristan Bunn

Try different `lissajousPoint()` arguments, or add new curves and lines; maybe even try to connect three lines between three curves for morphing triangles. Keep experimenting to see what you come up with.

Summary

In this chapter, you've learned how to define your own functions, which reduce repetition and help you structure more modular programs. Remember that well-named functions will make your code easier to read and understand, for yourself and anybody else dealing with it.

You can add parameters to any function to make it more versatile, and the function call will include different arguments that correspond to those parameters to control how it works. You can call a function by using positional and/or keyword arguments. For optional arguments, you can define parameters that include default values for Python to fall back on.

You can also define functions that return values, which means you can use a function to process data and hand back a result to the function caller. If a function returns a value, you can assign it to a variable. Additionally, you can wrap a function around an argument to process and return a value for another function.

This chapter also introduced trigonometry concepts and how to use them to simulate periodic motion. You learned about built-in Processing trig functions, like `sin()` and `cos()`, which you used to draw circles, spirals, ellipses, sine waves, and Lissajous curves. Experiment with trigonometry to generate compelling patterns and movements like those you see in some screensavers.

In the next chapter, you'll write *classes*, which you will use to create *objects*. These techniques enable you to structure your code more efficiently, especially for larger, more complex programs, by modeling your programs around real-world objects. You'll also learn about *vectors* for programming motion.