p5 Documentation

Release 0.7.0

Abhik Pal

Aug 22, 2020

Contents

p5 is a native Python port of the Processing API by Abhik Pal, Manindra Mohrarna, and contributors. Processing "is a flexible software sketchbook and a language for learning how to code within the context of the visual arts." It mainly developed in Boston (at Fathom Information Design), Los Angeles (at the UCLA Arts Software Studio), and New York City (at NYU's ITP)[1].

The p5 documentation is structured into:

- An *Installation* section that guides one through the basic installation process.
- *Tutorials* with a collection of step-by-step lessons covering various parts of the p5 api.
- Short *Guides* that discuss key concepts and details at a fairly high level.
- The *Reference* provides a detailed overview of the complete p5 api. Code examples accompanying the reference can be found in the *references* directory in the p5 examples repository (also available as a zip)

p5 is free and open source software and has been released under the GPL3 license. To report a bug / make a feature request use the issues page on the main repository.

[1] See Overview on the Processing website for details.

CHAPTER 1

Installation

1.1 Prerequisites: Python

p5 requires Python 3. Most recent versions of MacOS and Linux systems should have an installation of Python already. If you're not sure which version your computer is using, run

```
python --version
```

from a terminal window. If the reported version is greater than 3.0, you're good to proceed.

If you don't already have Python installed, refer to the The Hitchhiker's Guide To Python and its section on Python installation. Alternatively, on Windows, you can also consider installing Python through the Miniconda Python installer.

1.2 Prerequisites: GLFW

Internally p5 uses GLFW to handle window events and to work with OpenGL graphics.

1.2.1 Windows

First, download and install the pre-compiled Windows binaries from the official GLFW downloads page. During the installation process, make sure to take note of the folder where GLFW.

Finally, the GLFW installation directory should be added to the system path. Make sure to add containing the .dll and .a files (for example: *<path to glfw>\glfw-3.2.1.bin.WIN64\lib-mingw-w64*)

First locate the "Environment Variables" settings dialog box. On recent versions of Windows (Windows 8 and later), go to System info > Advanced Settings > Environment Variables. On older versions (Windows 7 and below) first right click the computer icon (from the desktop or start menu) and then go to Properties > Advanced System Settings > Advanced > Environment Variables. Now, find and highlight the "Path" variable and click the edit button. Here, add the GLFW installation directory to the end of the list and save the settings.

1.2.2 MacOS, Linux

Most package systems such as *homebrew*, *aptitude*, etc already have the required GLFW binaries. For instance, to install GLFW on Mac using homebrew, run

```
$ brew install glfw
```

Similarly, on Debain (and it's derivatives like Ubuntu and Linux Mint)run

```
$ sudo apt-get install libglfw3
```

For other Linux based systems, find and install the GLFW package using the respective package system.

1.3 Installing p5

The p5 installer should automatically install the required dependencies (mainly numpy and vispy), so run

```
$ pip install p5 --user
```

to install the latest p5 version.

1.4 Troubleshooting

1. In case the automatically installation fails, try installing the dependencies separately:

```
$ pip install numpy
$ pip install vispy
```

2. If you get a error that says `Microsoft Visual C++ is required` then follow the below steps:

   - Install the prebuilt version of vispy from here: https://www.lfd.uci.edu/~gohlke/pythonlibs/#vispy. For instance if you have python 3.8 then download the cp38 one.
   - Open terminal and cd into the directory in which you downloaded the prebuilt vispy file.
   - **Then in terminal type:**

```
$ pip install file_downloaded.whl
$ pip install p5 --user
```

In case of other installation problems, open an issue on the main p5 Github repository.

CHAPTER 2

Tutorials

A collection of step-by-step lessons covering beginner, intermediate, and advanced topics. Adapted from the tutorials on the Processing website.

2.1 Coordinate System and Shapes

Authors Daniel Shiffman; Arihant Parsoya (p5 port)

Copyright This tutorial is from the book Learning Processing by Daniel Shiffman, published by Morgan Kaufmann, © 2008 Elsevier Inc. All rights reserved. The tutorial was ported to p5 by Arihant Parsoya. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

Before we begin programming with Processing, we must first channel our eighth grade selves, pull out a piece of graph paper, and draw a line. The shortest distance between two points is a good old fashioned line, and this is where we begin, with two points on that graph paper.

The above figure shows a line between point A (1,0) and point B (4,5). If you wanted to direct a friend of yours to draw that same line, you would give them a shout and say "draw a line from the point one-zero to the point four-five, please." Well, for the moment, imagine your friend was a computer and you wanted to instruct this digital pal to display that same line on its screen. The same command applies (only this time you can skip the pleasantries and you will be required to employ a precise formatting). Here, the instruction will look like this:

```
line((1,0),(4,5))
```

Even without having studied the syntax of writing code, the above statement should make a fair amount of sense. We are providing a command (which we will refer to as a "function") for the machine to follow entitled "line." In addition, we are specifying some arguments for how that line should be drawn, from point A (1,0) to point B (4,5). If you think of that line of code as a sentence, the function is a verb and the arguments are the objects of the sentence. The code sentence also ends with a semicolon instead of a period.

The key here is to realize that the computer screen is nothing more than a fancier piece of graph paper. Each pixel of the screen is a coordinate - two numbers, an "x" (horizontal) and a "y" (vertical) - that determines the location of a point in space. And it is our job to specify what shapes and colors should appear at these pixel coordinates.

Nevertheless, there is a catch here. The graph paper from eighth grade ("Cartesian coordinate system") placed (0,0) in the center with the y-axis pointing up and the x-axis pointing to the right (in the positive direction, negative down and to the left). The coordinate system for pixels in a computer window, however, is reversed along the y-axis. (0,0) can be found at the top left with the positive direction to the right horizontally and down vertically.

2.1.1 Simple Shapes

The vast majority of the programming examples you'll see with Processing are visual in nature. These examples, at their core, involve drawing shapes and setting pixels. Let's begin by looking at four primitive shapes.

For each shape, we will ask ourselves what information is required to specify the location and size (and later color) of that shape and learn how Processing expects to receive that information. In each of the diagrams below, we'll assume a window with a width of 10 pixels and height of 10 pixels. This isn't particularly realistic since when you really start coding you will most likely work with much larger windows (10x10 pixels is barely a few millimeters of screen space.) Nevertheless for demonstration purposes, it is nice to work with smaller numbers in order to present the pixels as they might appear on graph paper (for now) to better illustrate the inner workings of each line of code.

A `point()` is the easiest of the shapes and a good place to start. To draw a point, we only need an x and y coordinate.

A `line()` isn't terribly difficult either and simply requires two points: (x1,y1) and (x2,y2):

Once we arrive at drawing a `rect()`, things become a bit more complicated. In Processing, a rectangle is specified by the coordinate for the top left corner of the rectangle, as well as its width and height.

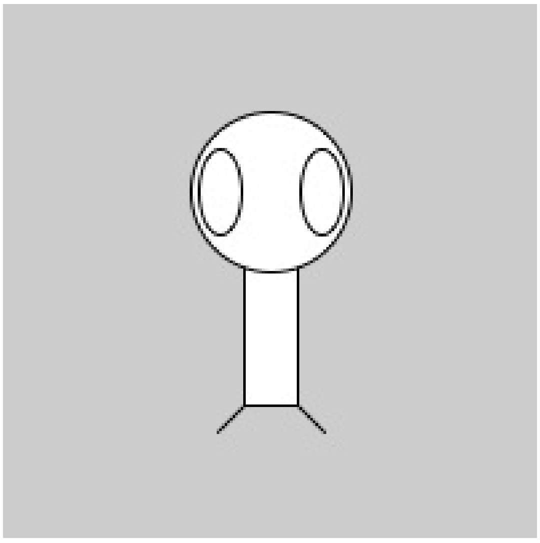
A second way to draw a rectangle involves specifying the centerpoint, along with width and height. If we prefer this method, we first indicate that we want to use the "CENTER" mode before the instruction for the rectangle itself. Note that Processing is case-sensitive.

Finally, we can also draw a rectangle with two points (the top left corner and the bottom right corner). The mode here is "CORNERS".

Once we have become comfortable with the concept of drawing a rectangle, an `ellipse()` is a snap. In fact, it is identical to rect() with the difference being that an ellipse is drawn where the bounding box of the rectangle would be. The default mode for ellipse() is "CENTER", rather than "CORNER."

It is important to acknowledge that these ellipses do not look particularly circular. Processing has a built-in methodology for selecting which pixels should be used to create a circular shape. Zoomed in like this, we get a bunch of squares in a circle-like pattern, but zoomed out on a computer screen, we get a nice round ellipse. Processing also gives us the power to develop our own algorithms for coloring in individual pixels (in fact, we can already imagine how we might do this using "point" over and over again), but for now, we are content with allowing the "ellipse" statement to do the hard work. (For more about pixels, start with: the pixels reference page, though be warned this is a great deal more advanced than this tutorial.)

Now let's look at what some code with shapes in more realistic setting, with window dimensions of 200 by 200. Note the use of the `size()` function to specify the width and height of the window.



```
size(200,200)
rectMode("CENTER")
rect((100,100),20,100)
ellipse((100,70),60,60)
ellipse((81,70),16,32)
ellipse((119,70),16,32)
```

(continues on next page)

(continued from previous page)

```
line((90,150),(80,160))
line((110,150),(120,160))
```

2.2 Color

Authors Daniel Shiffman; Abhik Pal (p5 port)

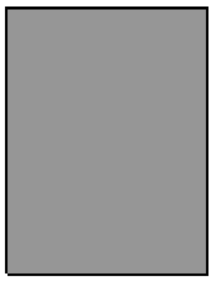
Copyright This tutorial is from the book Learning Processing by Daniel Shiffman, published by Morgan Kaufmann, © 2008 Elsevier Inc. All rights reserved. The tutorial was ported to p5 by Abhik Pal. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

In the digital world, when we want to talk about a color, precision is required. Saying "Hey, can you make that circle bluish-green?" will not do. Color, rather, is defined as a range of numbers. Let's start with the simplest case: black & white or grayscale. 0 means black, 255 means white. In between, every other number – 50, 87, 162, 209, and so on – is a shade of gray ranging from black to white.

Note: *Does 0-255 seem arbitrary to you?*

Color for a given shape needs to be stored in the computer's memory. This memory is just a long sequence of 0's and 1's (a whole bunch of on or off switches.) Each one of these switches is a bit, eight of them together is a byte. Imagine if we had eight bits (one byte) in sequence – how many ways can we configure these switches? The answer is (and doing a little research into binary numbers will prove this point) 256 possibilities, or a range of numbers between 0 and 255. We will use eight bit color for our grayscale range and 24 bit for full color (eight bits for each of the red, green, and blue color components).

By adding the *p5.stroke()* and *p5.fill()* functions before something is drawn, we can set the color of any given shape. There is also the function *p5.background()* which sets a background color for the window. Here's an example.



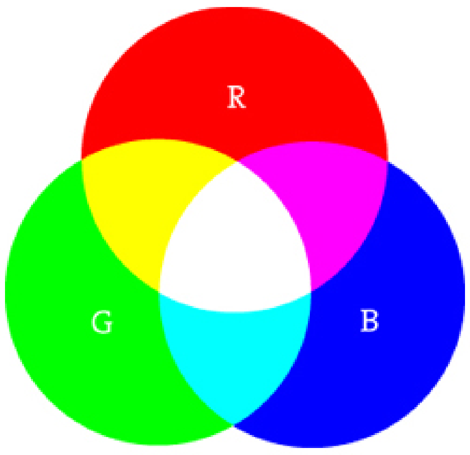
```
from p5 import *

def draw():
    background(255)
    stroke(0)
    fill(150)
    rect((50, 50), 75, 100)

if __name__ == '__main__':
    run()
```

Stroke or fill can be eliminated with the functions: *p5.no_stroke()* and `no_fill()`. Our instinct might be to say `stroke(0)` for no outline, however, it is important to remember that 0 is not "nothing", but rather denotes the color black. Also, remember not to eliminate both – with `no_stroke` and `no_fill`, nothing will appear!

In addition, if we draw two shapes, p5 will always use the most recently specified stroke and fill, reading the code from top to bottom.

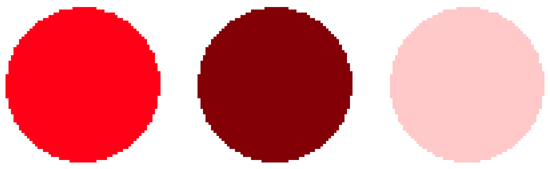
2.2.1 RGB Color

Remember finger painting? By mixing three "primary" colors, any color could be generated. Swirling all colors together resulted in a muddy brown. The more paint you added, the darker it got. Digital colors are also constructed by mixing three primary colors, but it works differently from paint. First, the primaries are diff erent: red, green, and blue (i.e., "RGB" color). And with color on the screen, you are mixing light, not paint, so the mixing rules are different as well.



- Red + Green = Yellow
- Red + Blue = Purple
- Green + Blue = Cyan (blue-green)
- Red + Green + Blue = White
- No colors = Black

This assumes that the colors are all as bright as possible, but of course, you have a range of color available, so some red plus some green plus some blue equals gray, and a bit of red plus a bit of blue equals dark purple. While this may take some getting used to, the more you program and experiment with RGB color, the more it will become instinctive, much like swirling colors with your fi ngers. And of course you can't say "Mix some red with a bit of blue," you have to provide an exact amount. As with grayscale, the individual color elements are expressed as ranges from 0 (none of that color) to 255 (as much as possible), and they are listed in the order R, G, and B. You will get the hang of RGB color mixing through experimentation, but next we will cover some code using some common colors.

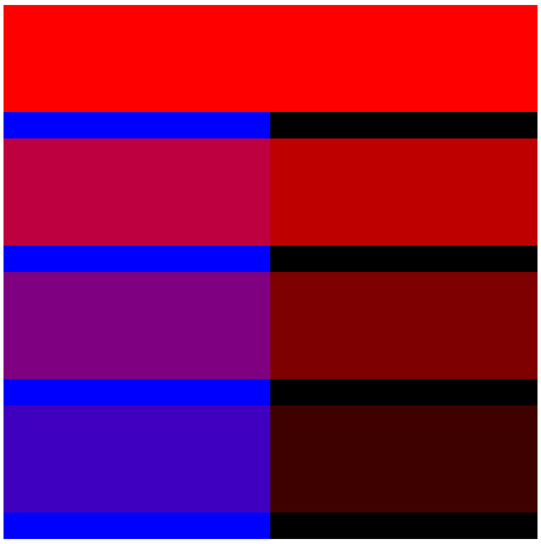


2.2.2 Color Transparency

In addition to the red, green, and blue components of each color, there is an additional optional fourth component, referred to as the color's "alpha." Alpha means transparency and is particularly useful when you want to draw elements that appear partially see-through on top of one another. The alpha values for an image are sometimes referred to collectively as the "alpha channel" of an image.

It is important to realize that pixels are not literally transparent, this is simply a convenient illusion that is accomplished by blending colors. Behind the scenes, Processing takes the color numbers and adds a percentage of one to a percentage of another, creating the optical perception of blending. (If you are interested in programming "rose-colored" glasses, this is where you would begin.)

Alpha values also range from 0 to 255, with 0 being completely transparent (i.e., 0% opaque) and 255 completely opaque (i.e., 100% opaque).



```
from p5 import *

def setup():
    size(200, 20
    no_stroke()

def draw():
    background(0

    # No fourth␣
↪argument means
    fill(0, 0, 2
    rect((0, 0),

    # 255 means
    fill(255, 0,
    rect((0, 0),

    # 75% opacit
    fill(255, 0,
    rect((0, 50)

    # 55% opacit
    fill(255, 0,
    rect((0, 100

    # 25% opacit
    fill(255, 0,
    rect((0, 150

if __name__ == '
    run()
```

2.2.3 Custom Color Ranges

RGB color with ranges of 0 to 255 is not the only way you can handle color in Processing. Behind the scenes in the computer's memory, color is always talked about as a series of 24 bits (or 32 in the case of colors with an alpha). However, Processing will let us think about color any way we like, and translate our values into numbers the computer understands. For example, you might prefer to think of color as ranging from 0 to 100 (like a percentage). You can do this by specifying a custom `p5.color_mode()`.

```
color_mode('RGB'
```

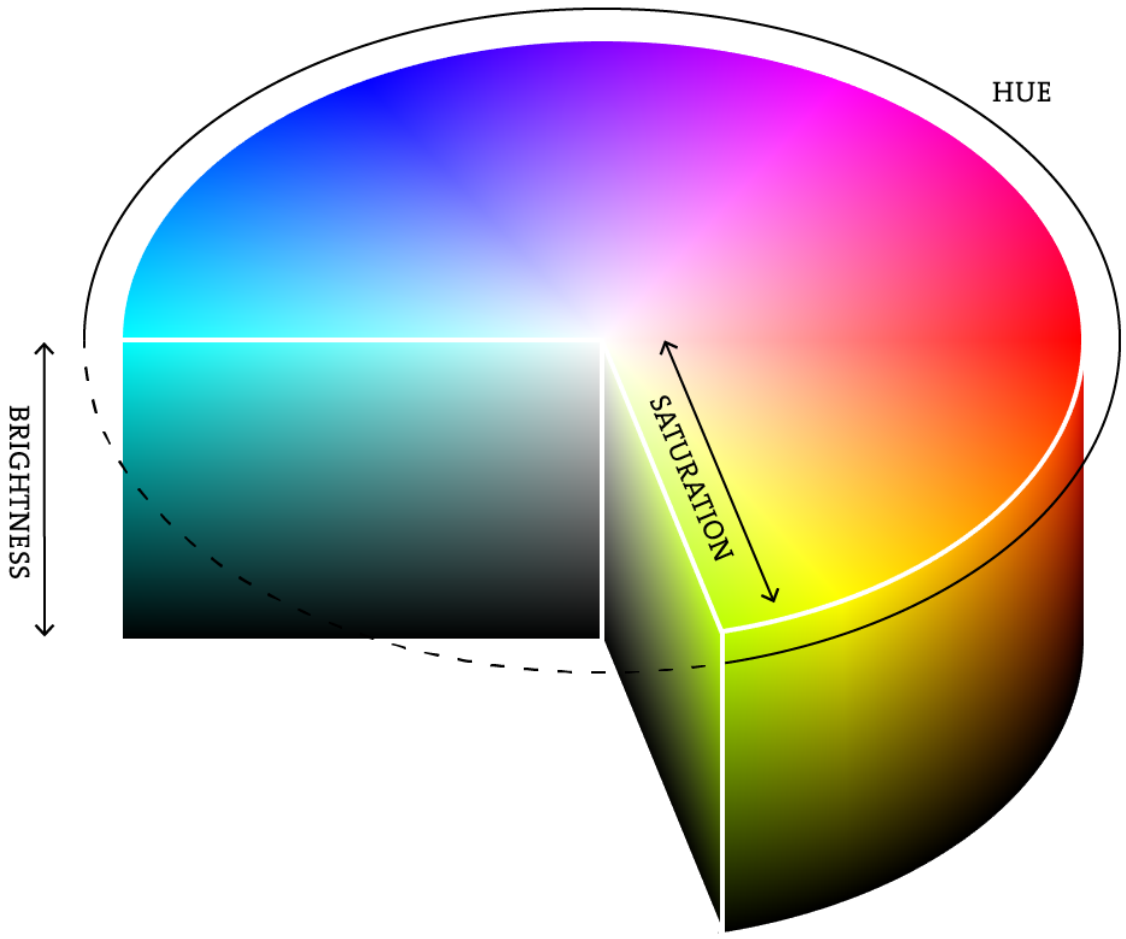
The above function says: "OK, we want to think about color in terms of red, green, and blue. The range of RGB values will be from 0 to 100."

Although it is rarely convenient to do so, you can also have different ranges for each color component:

```
color_mode('RGB', 100, 500, 10, 255)
```

Now we are saying "Red values go from 0 to 100, green from 0 to 500, blue from 0 to 10, and alpha from 0 to 255."

Finally, while you will likely only need RGB color for all of your programming needs, you can also specify colors in the HSB (hue, saturation, and brightness) mode. Without getting into too much detail, HSB color works as follows:



- **Hue** –The color type, ranges from 0 to 255 by default.
- **Saturation** – The vibrancy of the color, 0 to 255 by default.
- **Brightness** – The, well, brightness of the color, 0 to 255 by default.

With *p5.color_mode()* you can set your own ranges for these values. Some prefer a range of 0-360 for hue (think of 360 degrees on a color wheel) and 0-100 for saturation and brightness (think of 0-100%).

2.3 Objects

Authors Daniel Shiffman; Arihant Parsoya (p5 port)

Copyright This tutorial is from the book Learning Processing by Daniel Shiffman, published by Morgan Kaufmann, © 2008 Elsevier Inc. All rights reserved. The tutorial was ported to p5 by Arihant Parsoya. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

Before we begin examining the details of how object-oriented programming (OOP) works in Processing, let's embark on a short conceptual discussion of "objects" themselves. Imagine you were not programming in Processing, but were instead writing out a program for your day—a list of instructions, if you will. It might start out something like:

- Wake up.
- Drink coffee (or tea).
- Eat breakfast: cereal, blueberries, and soy milk.
- Ride the subway.

What is involved here? Specifically, what things are involved? First, although it may not be immediately apparent from how we wrote the above instructions, the main thing is you—a human being, a person. You exhibit certain properties. You look a certain way; perhaps you have brown hair, wear glasses, and appear slightly nerdy. You also have the ability to do stuff, such as wake up (presumably, you can also sleep), eat, or ride the subway. An object is just like you, a thing that has properties and can do stuff.

So, how does this relate to programming? The properties of an object are variables, and the things an object can do are functions. Object-oriented programming is the marriage of all of the programming fundamentals: data and functionality.

Let's map out the data and functions for a very simple human object:

Human Data

- Height.
- Weight.
- Gender.
- Eye color.
- Hair color.

Human functions

- Sleep.

- Wake up.
- Eat.
- Ride some form of transportation.

Now, before we get too much further, we need to embark on a brief metaphysical digression. The above structure is not a human being itself; it simply describes the idea, or the concept, behind a human being. It describes what it is to be human. To be human is to have height, hair, to sleep, to eat, and so on. This is a crucial distinction for programming objects. This human being template is known as a class. A class is different from an object. You are an object. I am an object. That guy on the subway is an object. Albert Einstein is an object. We are all people, real world instances of the idea of a human being.

Think of a cookie cutter. A cookie cutter makes cookies, but it is not a cookie itself. The cookie cutter is the class, the cookies are the objects.

2.3.1 Using an Object

Before we look at the actual writing of a class itself, let's briefly look at how using objects in our main program (i.e., setup() and draw()) makes the world a better place.

Consider the pseudo-code for a simple sketch that moves a rectangle horizontally across the window (we'll think of this rectangle as a "car").

Data (Global Variables):

- Car color.
- Car x location.
- Car y location.
- Car x speed.

Setup:

- Initialize car color.
- Initialize car location to starting point.
- Initialize car speed.

Draw:

- Fill background.
- Display car at location with color.
- Increment car's location by speed.

To implement the above pseudo-code, we would define global variables at the top of the program, initialize them in setup(), and call functions to move and display the car in draw(). Something like:

Object-oriented programming allows us to take all of the variables and functions out of the main program and store them inside a car object. A car object will know about its data—color, location, speed. The object will also know about the stuff it can do, the methods (functions inside an object)—the car can drive and it can be displayed.

Using object-oriented design, the pseudocode improves to look something like this:

```
c = Color(0)
x = 0
y = 100
speed = 1

def setup():
        size(200, 200)

def draw():
        background(255)
        move()
        display()

def move():
        x = x + speed
        if x > width:
                x = 0

def display():
        fill(c)
        rect((x, y), 30, 10)
```

Data (Global Variables):

- Car object.

Setup:

- Initialize car object.

Draw:

- Fill background.
- Display car object.
- Drive car object.

Notice: we removed all of the global variables from the first example. Instead of having separate variables for car color, car location, and car speed, we now have only one variable: a Car variable! And, instead of initializing those three variables, we initialize one thing: the Car object. Where did those variables go? They still exist, only now they live inside of the Car object (and will be defined in the Car class, which we will get to in a moment).

Moving beyond pseudocode, the actual body of the sketch might look like:

```
myCar = None

def setup():
```

(continues on next page)

(continued from previous page)

```
        myCar = Car()

def draw():
        background(255)
        myCar.drive()
        myCar.display()
```

We are going to get into the details regarding the above code in a moment, but before we do so, let's take a look at how the Car class itself is written.

2.3.2 Writing the Cookie Cutter

The simple Car example above demonstrates how the use of objects in Processing makes for clean, readable code. The hard work goes into writing the object template—that is, the class itself. When you are first learning about object-oriented programming, it is often a useful exercise to take a program written without objects and, not changing the functionality at all, rewrite it using objects. We will do exactly this with the car example, recreating exactly the same look and behavior in an object-oriented manner.

All classes must include four elements: name, data, constructor, and methods. (Technically, the only actual required element is the class name, but the point of doing object-oriented programming is to include all of these.)

Here is how we can take the elements from a simple non-object-oriented sketch and place them into a Car class, from which we will then be able to make Car objects.

Class Name: The name is specified by "class WhateverNameYouChoose". We then enclose all of the code for the class inside curly brackets after the name declaration. Class names are traditionally capitalized (to distinguish them from variable names, which traditionally are lowercase).

Data: The data for a class is a collection of variables. These variables are often referred to as instance variables since each instance of an object contains this set of variables.

Constructor: The constructor is a special function inside of a class that creates the instance of the object itself. It is where you give the instructions on how to set up the object. It is just like Processing's setup() function, only here it is used to create an individual object within the sketch, whenever a new object is created from this class. It always has the same name as the class and is called by invoking the new operator: "Car myCar = new Car();".

Functionality: We can add functionality to our object by writing methods.

Note that the code for a class exists as its own block and can be placed anywhere outside of setup() and draw().

```
def setup():
        ...

def draw():
```

(continues on next page)

(continued from previous page)

```
        ...

class Car():
        ...
```

2.3.3 Using an Object: The Details

Earlier, we took a quick peek at how an object can greatly simplify the main parts of a Processing sketch (i.e. setup() and draw()).

```
# Step 1. Declare an object.
myCar = None

def setup():
        # Step 2. Initialize object.
        myCar = Car()

def draw():
        background(255)
        # Step 3. Call methods on the object.
        myCar.drive()
        myCar.display()
```

Let's look at the details behind the above three steps outlining how to use an object in your sketch.

Step 1. Declaring an object variable.

A variable is always declared by specifying a type and a name. With a primitive data type, such as an integer, it looks like this:

```
# Variable Declaration
var = None # varible name
```

Primitive data types are singular pieces of information: an integer, a float, a character, etc. Declaring a variable that holds onto an object is quite similar. The difference is that here the type is the class name, something we will make up, in this case "Car." Objects, incidentally, are not primitives and are considered complex data types. (This is because they store multiple pieces of information: data and functionality. Primitives only store data.)

Step 2. Initializing an object.

In order to initialize a variable (i.e., give it a starting value), we use an assignment operation—variable equals something. With a primitive (such as integer), it looks like this:

```
var = 10 # var equals 10
```

Initializing an object is a bit more complex. Instead of simply assigning it a value, like with an integer or floating point number, we have to construct the object. An object is made with the new operator.

```
# Object Initialization
myCar = Car() # The new operator is used to make a new object.
```

In the above example, “myCar” is the object variable name and “=” indicates we are setting it equal to something, that something being a new instance of a Car object. What we are really doing here is initializing a Car object. When you initialize a primitive variable, such as an integer, you just set it equal to a number. But an object may contain multiple pieces of data. Recalling the Car class, we see that this line of code calls the constructor, a special function named Car() that initializes all of the object’s variables and makes sure the Car object is ready to go.

One other thing: with the primitive integer “var,” if you had forgotten to initialize it (set it equal to 10), Processing would have assigned it a default value—zero. An object (such as “myCar”), however, has no default value. If you forget to initialize an object, Processing will give it the value null. null means nothing. Not zero. Not negative one. Utter nothingness. Emptiness. If you encounter an error in the message window that says “NullPointerException” (and this is a pretty common error), that error is most likely caused by having forgotten to initialize an object.

Step 3. Using an object

Once we have successfully declared and initialized an object variable, we can use it. Using an object involves calling functions that are built into that object. A human object can eat; a car can drive; a dog can bark. Calling a function inside of an object is accomplished via dot syntax: variableName.objectFunction(Function Arguments);

In the case of the car, none of the available functions has an argument so it looks like:

```
myCar.drive()
myCar.display()
```

2.3.4 Constructor Arguments

In the above examples, the car object was initialized using the new operator followed by the constructor for the class.

```
myCar = Car()
```

This was a useful simplification while we learned the basics of OOP. Nonetheless, there is a rather serious problem with the above code. What if we wanted to write a program with two car objects?

```
# Creating two car objects
myCar1 = Car()
myCar2 = Car()
```

This accomplishes our goal; the code will produce two car objects, one stored in the variable myCar1 and one in myCar2. However, if you study the Car class, you will notice that these two cars will be identical: each one will be colored white, start in the middle of the screen, and have a speed of 1. In English, the above reads:

Make a new car.

We want to instead say:

Make a new red car, at location (0,10) with a speed of 1.

So that we could also say:

Make a new blue car, at location (0,100) with a speed of 2.

We can do this by placing arguments inside of the constructor method.

```
myCar = Car(color(255,0,0),0,100,2)
```

The constructor must be rewritten to incorporate these arguments:

```
class Car:
        def __init__(self, tempC, tempXpos, tempYpos, tempXspeed):
                self.x = tempC
                self.xpos = tempXpos
                self.ypos = tempYpos
                self.xspeed = tempXspeed
```

In my experience, the use of constructor arguments to initialize object variables can be somewhat bewildering. Please do not blame yourself. The code is strange-looking and can seem awfully redundant: "I need to place arguments inside the constructor for every single variable?"

Nevertheless, this is quite an important skill to learn, and, ultimately, is one of the things that makes object-oriented programming powerful. But for now, it may feel painful. Let's looks at how parameters work in this context.

Arguments are local variables used inside the body of a function that get filled with values when the function is called. In the examples, they have one purpose only: to initialize the variables inside of an object. These are the variables that count—the car's actual color, the car's actual x location, and so on. The constructor's arguments are just temporary, and exist solely to pass a value from where the object is made into the object itself.

This allows us to make a variety of objects using the same constructor. You might also just write the word temp in your argument names to remind you of what is going on (`c` vs. `tempC`). You will also see programmers use an underscore (`c` vs. `c_`) in many examples. You can name these whatever you want, of course. However, it is advisable to choose a name that makes sense to you, and also to stay consistent.

We can now take a look at the same sketch with multiple object instances, each with unique properties.

```
from p5 import *

# Example: Two Car objects
myCar1 = None
```

(continues on next page)

(continued from previous page)

```
myCar2 = None

def setup():
    global myCar1, myCar2
    size(200, 200)
    # Parameters go inside the parentheses when the object is␣
↪constructed.
    myCar1 = Car(Color(255,0,0),0,100,2)
    myCar2 = Car(Color(0,0,255),0,10,1)

def draw():
    global myCar1, myCar2
    background(255)

    myCar1.drive()
    myCar1.display()
    myCar2.drive()
    myCar2.display()

class Car:
    def __init__(self, tempC, tempXpos, tempYpos, tempXspeed):
        self.x = tempC
        self.xpos = tempXpos
        self.ypos = tempYpos
        self.xspeed = tempXspeed

    def display(self):
        stroke(0)
        fill(Color(0, 255, 0))
        rect_mode("CENTER")
        rect((self.xpos, self.ypos),20,10)

    def drive(self):
        self.xpos = self.xpos + self.xspeed
        if self.xpos > width:
            self.xpos = 0

if __name__ == '__main__':
    run()
```

2.4 Interactivity

Authors Casey Reas and Ben Fry; Arihant Parsoya (p5 port)

Copyright This tutorial is the "Interactivity" chapter from Processing: A Programming Handbook for Visual Designers and Artists, Second Edition, published by MIT Press. © 2014 MIT Press. If you see any errors or have comments, please let us know. The tutorial was ported to p5 by Arihant Parsoya. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

The screen forms a bridge between our bodies and the realm of circuits and electricity inside computers. We control elements on screen through a variety of devices such as touch pads, trackballs, and joysticks, but the keyboard and mouse remain the most common input devices for desktop computers. The computer mouse dates back to the late 1960s, when Douglas Engelbart presented the device as an element of the oN-Line System (NLS), one of the first computer systems with a video display. The mouse concept was further developed at the Xerox Palo Alto Research Center (PARC), but its introduction with the Apple Macintosh in 1984 was the catalyst for its current ubiquity. The design of the mouse has gone through many revisions in the last forty years, but its function has remained the same. In Engelbart's original patent application in 1970 he referred to the mouse as an "X-Y position indicator," and this still accurately, but dryly, defines its contemporary use.

The physical mouse object is used to control the position of the cursor on screen and to select interface elements. The cursor position is read by computer programs as two numbers, the x-coordinate and the y-coordinate. These numbers can be used to control attributes of elements on screen. If these coordinates are collected and analyzed, they can be used to extract higher-level information such as the speed and direction of the mouse. This data can in turn be used for gesture and pattern recognition.

Keyboards are typically used to input characters for composing documents, email, and instant messages, but the keyboard has potential for use beyond its original intent. The migration of the keyboard from typewriter to computer expanded its function to enable launching software, moving through the menus of software applications, and navigating 3D environments in games. When writing your own software, you have the freedom to use the keyboard data any way you wish. For example, basic information such as the speed and rhythm of the fingers can be determined by the rate at which keys are pressed. This information could control the speed of an event or the quality of motion. It's also possible to ignore the characters printed on the keyboard itself and use the location of each key relative to the keyboard grid as a numeric position.

The modern computer keyboard is a direct descendant of the typewriter. The position of the keys on an English-language keyboard is inherited from early typewriters. This layout is called QWERTY because of the order of the top row of letter keys. This more than one-hundred-year-old mechanical legacy still affects how we write software today.

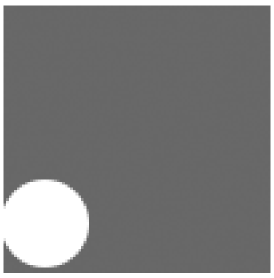
2.4.1 Mouse Data

The Processing variables `mouse_x` and `mouse_y` store the x-coordinate and y-coordinate of the cursor relative to the origin in the upper-left corner of the display window. To see the actual values produced while moving the mouse, run this program to print the values to the console:

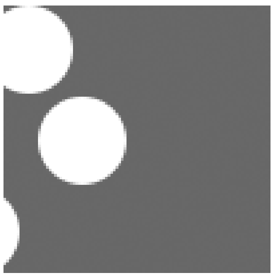
```
def draw():
        frame_rate(12)
        print(mouse_x, " : ", mouse_y)
```

When a program starts, the `mouse_x` and `mouse_y` values are 0. If the cursor moves into the display window, the values are set to the current position of the cursor. If the cursor is at the left, the mouse_x value is 0 and the value increases as the cursor moves to the right. If the cursor is at the top, the mouse_y value is 0 and the value increases as the cursor moves down. If mouse_x and mouse_y are used in programs without a draw() or if no_loop() is run in setup(), the values will always be 0.

The mouse position is most commonly used to control the location of visual elements on screen. More interesting relations are created when the visual elements relate differently to the mouse values, rather than simply mimicking the current position. Adding and subtracting values from the mouse position creates relationships that remain constant, while multiplying and dividing these values creates changing visual relationships between the mouse position and the elements on the screen. In the first of the following examples, the circle is directly mapped to the cursor, in the second, numbers are added and subtracted from the cursor position to create offsets, and in the third, multiplication and division are used to scale the offsets.



```
from p5 import *

def setup():
    size(100, 100)
    no_stroke()

def draw():
    background(126)
    ellipse((mouse_x, mouse_y), 33, 33)

if __name__ == '__main__':
    run()
```



```python
from p5 import *

def setup():
    size(100, 100)
    no_stroke()

def draw():
    background(126)
    ellipse((mouse_x, 16), 33, 33) # Top circle
    ellipse((mouse_x+20, 50), 33, 33) # Middle circle
    ellipse((mouse_x-20, 84), 33, 33) # Bottom circle

if __name__ == '__main__':
    run()
```

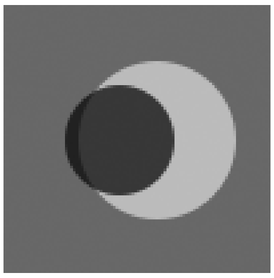
```python
from p5 import *

def setup():
    size(100, 100)
    no_stroke()

def draw():
    background(126)
    ellipse((mouse_x, 16), 33, 33) # Top circle
```

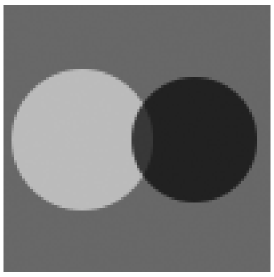
(continues on next page)

(continued from previous page)

```
    ellipse((mouse_x/2, 50), 33, 33) # Middle circle
    ellipse((mouse_x*2, 84), 33, 33) # Bottom circle

if __name__ == '__main__':
    run()
```

To invert the value of the mouse, subtract the mouse_x value from the width of the window and subtract the mouse_y value from the height of the screen.



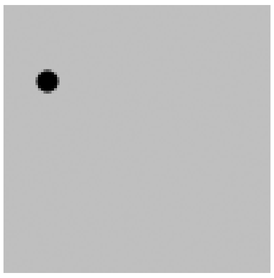


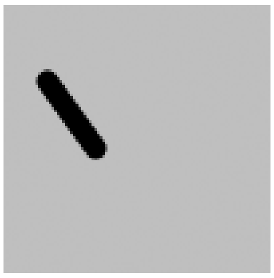
```
from p5 import *

def setup():
    size(100, 100)
    no_stroke()

def draw():
    background(126)
    x = mouse_x
    y = mouse_y
    ix = width - mouse_x # Inverse X
    iy = height - mouse_y # Inverse Y
    background(126)
    fill(255, 150)
    ellipse((x, height/2), y, y)
    fill(0, 159)
    ellipse((ix, height/2), iy, iy)

if __name__ == '__main__':
    run()
```

The Processing variables `pmouse_x` and `pmouse_y` store the mouse values from the previous frame. If the mouse does not move, the values will be the same, but if the mouse is moving quickly there can be large differences between the values. To see the difference, run the following program and alternate moving the mouse slowly and quickly. Watch the values print to the console.

```
def draw():
    frame_rate(12)
    print(pmouse_y - mouse_x)
```

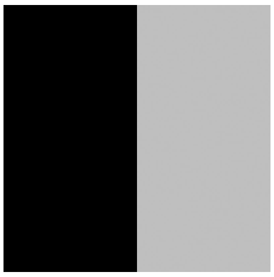
Draw a line from the previous mouse position to the current position to show the changing position in one frame and reveal the speed and direction of the mouse. When the mouse is not moving, a point is drawn, but quick mouse movements create long lines.

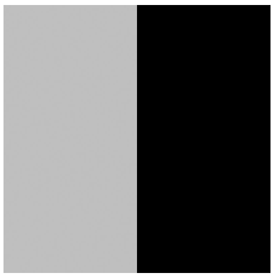




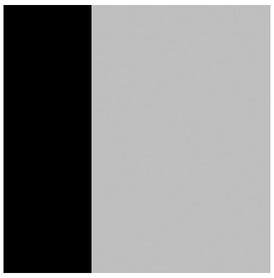
```
from p5 import *

def setup():
    size(100, 100)
    stroke_weight(8)

def draw():
    background(204)
    line((mouse_x, mouse_y), (pmouse_x, pmouse_y))

if __name__ == '__main__':
    run()
```

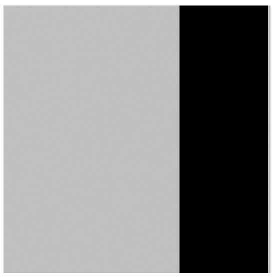
Use the `mouse_x` and `mouse_y` variables with an if structure to allow the cursor to select regions of the screen. The following examples demonstrate the cursor making a selection between different areas of the display window. The first divides the screen into halves, and the second divides the screen into thirds.





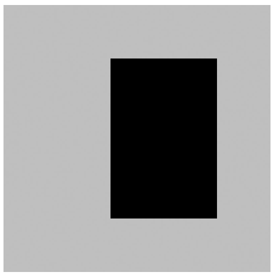
```
from p5 import *

def setup():
    size(100, 100)
    no_stroke()
    fill(0)

def draw():
    background(204)
    if mouse_x < 50:
        rect((0, 0), 50, 100)
    else:
        rect((50, 0), 50, 100)

if __name__ == '__main__':
    run()
```

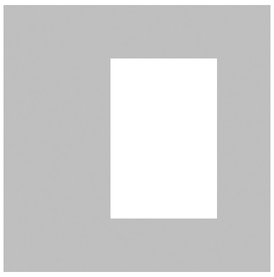




```
from p5 import *

def setup():
    size(100, 100)
    no_stroke()
    fill(0)

def draw():
    background(204)
    if mouse_x < 33:
        rect((0, 0), 33, 100)
    elif mouse_x < 66:
        rect((33, 0), 33, 100)
    else:
        rect((66, 0), 33, 100)

if __name__ == '__main__':
    run()
```

Use the logical operator `and` with an `if` structure to select a rectangular region of the screen. As demonstrated in the following example, when a relational expression is made to test each edge of a rectangle (left, right, top, bottom) and these are concatenated with a logical AND, the entire relational expression is true only when the cursor is inside the rectangle.





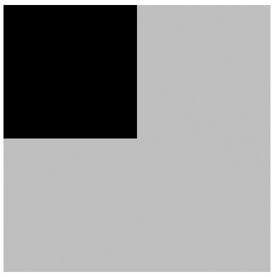
```
from p5 import *

```

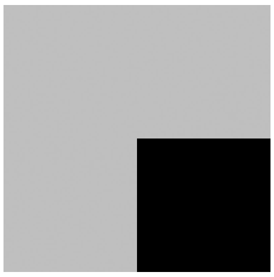
(continues on next page)

(continued from previous page)

```
def setup():
    size(100, 100)
    no_stroke()
    fill(0)

def draw():
    background(204)
    if ((mouse_x > 40) and (mouse_x < 80) and
        (mouse_y > 20) and (mouse_y < 80)):
        fill(255)
    else:
        fill(0)

    rect((40, 20), 40, 60)

if __name__ == '__main__':
    run()
```

This code asks, "Is the cursor to the right of the left edge and is the cursor to the left of the right edge and is the cursor beyond the top edge and is the cursor above the bottom?" The code for the next example asks a set of similar questions and combines them with the keyword else to determine which one of the defined areas contains the cursor.





```
from p5 import *

def setup():
    size(100, 100)
    no_stroke()
    fill(0)

def draw():
    background(204)
    if (mouse_x <= 50) and (mouse_y <= 50):
```

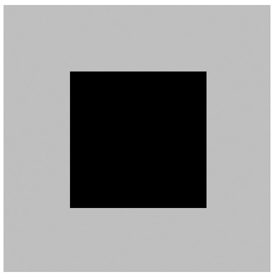
(continues on next page)

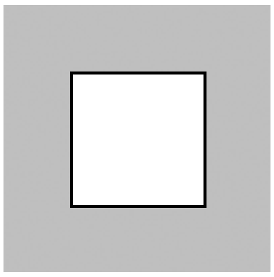
(continued from previous page)

```
        rect((0, 0), 50, 50) # Upper-left
    elif (mouse_x <= 50) and (mouse_y > 50):
        rect((0, 50), 50, 50) # Lower-left
    elif (mouse_x > 50) and (mouse_y <= 50):
        rect((50, 0), 50, 50) # Upper-right
    else:
        rect((50, 50), 50, 50) # Lower-right

if __name__ == '__main__':
    run()
```

2.4.2 Mouse buttons

Computer mice and other related input devices typically have between one and three buttons; Processing can detect when these buttons are pressed with the `mouse_is_pressed` and `mouse_button` variables. Used with the button status, the cursor position enables the mouse to perform different actions. For example, a button press when the mouse is over an icon can select it, so the icon can be moved to a different location on screen. The `mouse_is_pressed` variable is true if any mouse button is pressed and false if no mouse button is pressed. The variable `mouse_button` is LEFT, CENTER, or RIGHT depending on the mouse button most recently pressed. The `mouse_is_pressed` variable reverts to false as soon as the button is released, but the `mouse_button` variable retains its value until a different button is pressed. These variables can be used independently or in combination to control the software. Run these programs to see how the software responds to your fingers.



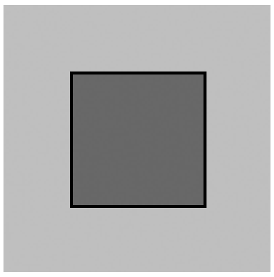


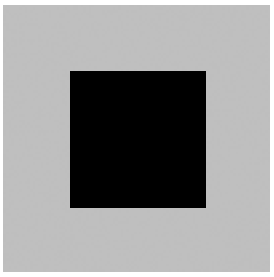
```
from p5 import *

def setup():
    size(100, 100)
```

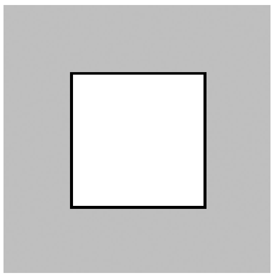
(continues on next page)

(continued from previous page)

```
def draw():
    background(204)
    if mouse_is_pressed:
        fill(255)
    else:
        fill(0)
    rect((25, 25), 50, 50)

if __name__ == '__main__':
    run()
```





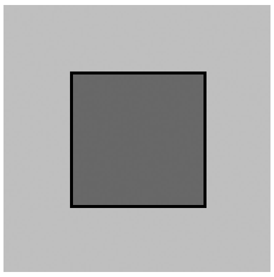


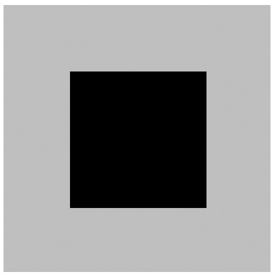
```
from p5 import *

def setup():
    size(100, 100)

def draw():
    background(204)
    if mouse_button == "LEFT":
        fill(0)
    elif mouse_button == "RIGHT":
        fill(255)
    else:
```

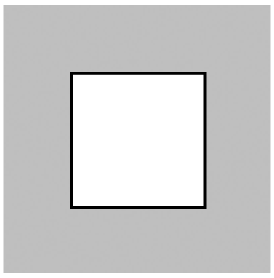
(continues on next page)

(continued from previous page)

```
        fill(126)

    rect((25, 25), 50, 50)

if __name__ == '__main__':
    run()
```







```
from p5 import *

def setup():
    size(100, 100)

def draw():
    background(204)
    if mouse_is_pressed:
        if mouse_button == "LEFT":
            fill(0)
        else:
            fill(255)
    else:
        fill(126)

    rect((25, 25), 50, 50)
```

(continues on next page)

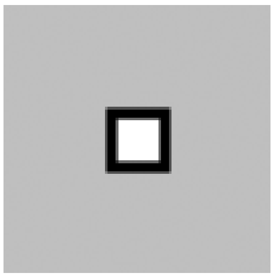
(continued from previous page)

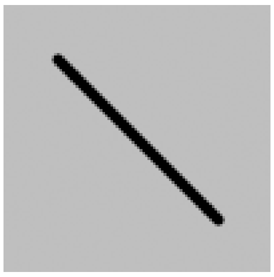
```
if __name__ == '__main__':
    run()
```

Not all mice have multiple buttons, and if software is distributed widely, the interaction should not rely on detecting which button is pressed.

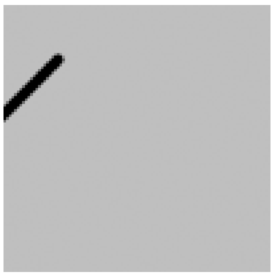
2.4.3 Keyboard data

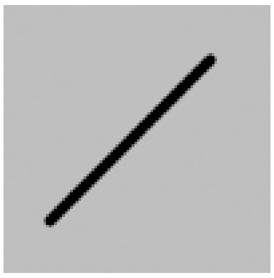
Processing registers the most recently pressed key and whether a key is currently pressed. The boolean variable `key_is_pressed` is true if a key is pressed and is false if not. Include this variable in the test of an if structure to allow lines of code to run only if a key is pressed. The `key_is_pressed` variable remains true while the key is held down and becomes false only when the key is released.





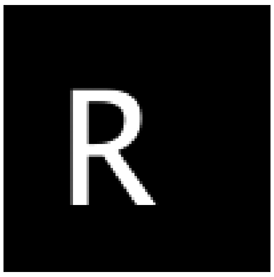
```
from p5 import *

def setup():
    size(100, 100)
    stroke_weight(4)

def draw():
    background(204)

    if key_is_pressed:
        line((20, 20), (80, 80))
    else:
        rect((40, 40), 20, 20)

if __name__ == '__main__':
    run()
```





```
from p5 import *

x = 20

def setup():
    size(100, 100)
    stroke_weight(4)

def draw():
    background(204)

    global x
    if key_is_pressed:
        x += 1

    line((x, 20), (x-60, 80))

if __name__ == '__main__':
    run()
```

The `key` variable stores a single alphanumeric character. Specifically, it holds the most recently pressed key. The key can be displayed on screen with the text() function (p. 150).



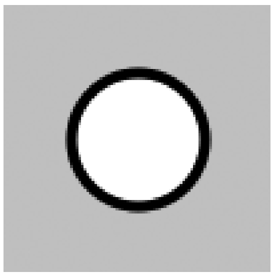
```
from p5 import *

def setup():
```

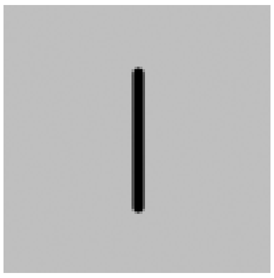
(continues on next page)

(continued from previous page)

```
    size(100, 100)
    text_size(60)

def draw():
    background(0)
    text(key, 20, 75) # Draw at coordinate (20,75)

if __name__ == '__main__':
    run()
```

The key variable may be used to determine whether a specific key is pressed. The following example uses the expression key=='A' to test if the A key is pressed. The single quotes signify A as the data type char (p. 144). The expression key=="A" will cause an error because the double quotes signify the A as a String, and it's not possible to compare a String with a char. The logical AND symbol, the && operator, is used to connect the expression with the keyPressed variable to ascertain that the key pressed is the uppercase A.



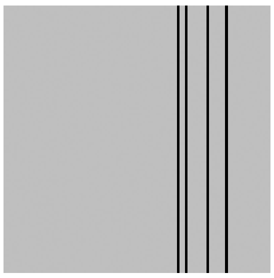


```
from p5 import *

def setup():
    size(100, 100)
    stroke_weight(4)

def draw():
    background(204)
    # If the 'A' key is pressed draw a line
    if key_is_pressed and key == "A":
        line((50, 25), (50, 75))
    else: # Otherwise, draw an ellipse
        ellipse((50, 50), 50, 50)

if __name__ == '__main__':
    run()
```

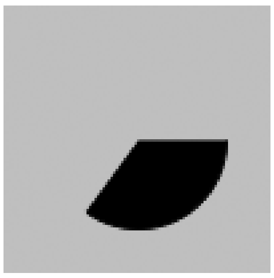
The previous example works with an uppercase A, but not if the lowercase letter is pressed. To check for both uppercase and lowercase letters, extend the relational expression with a logical OR, the || relational operator. Line 9 in the previous program would be changed to:

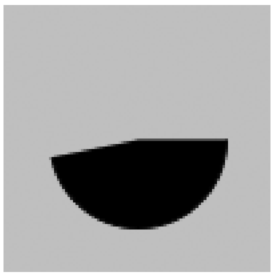
```
if key_is_pressed and ((key == 'a') || (key == 'A')):
```

Because each character has a numeric value as defined by the ASCII table (p. 605), the value of the key variable can be used like any other number to control visual attributes such as the position and color of shape elements. For instance, the ASCII table defines the uppercase A as the number 65, and the digit 1 is defined as 49.



```
from p5 import *

def setup():
    size(100, 100)
    stroke(0)

def draw():
    if key_is_pressed:
        x = ord(str(key)) - 32
        line((x, 0), (x, height))

if __name__ == '__main__':
    run()
```

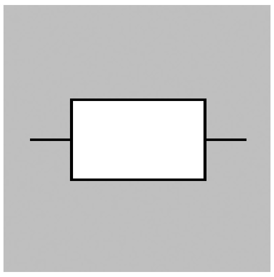


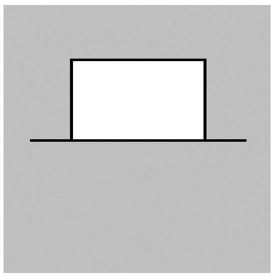


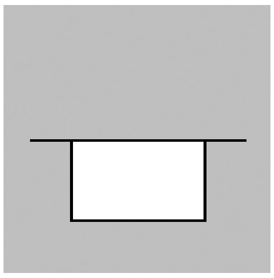
```
from p5 import *

angle = 0

def setup():
    size(100, 100)
    fill(0)

def draw():
    background(204)

    global angle
    if key_is_pressed:
        if (key >= 32) and (key <= 126):
            #  If the key is alphanumeric, use its value as an angle
            angle = (ord(str(key)) - 32) * 3

    arc((50, 50), 66, 66, 0, radians(angle))

if __name__ == '__main__':
    run()
```

2.4.4 Coded keys

In addition to reading key values for numbers, letters, and symbols, Processing can also read the values from other keys including the arrow keys and the Alt, Control, Shift, Backspace, Tab, Enter, Return, Escape, and Delete keys. The variable keyCode stores the ALT, CONTROL, SHIFT, UP, DOWN, LEFT, and RIGHT keys as constants. Before determining which coded key is pressed, it's necessary to check first to see if the key is coded. The expression key==CODED is true if the key is coded and false otherwise. Even though not alphanumeric, the keys included in the ASCII specification (BACKSPACE, TAB, ENTER, RETURN, ESC, and DELETE) will not be identified as a coded key. If you're making cross-platform projects, note that the Enter key is commonly used on PCs and UNIX and the Return key is used on Macintosh. Check for both Enter and Return to make sure your program will work for all platforms.







```
from p5 import *

y = 35

def setup():
    size(100, 100)

def draw():
    global y
    background(204)
    line((10, 50), (90, 50))

    if key == "UP":
        y = 20
    elif key == "DOWN":
        y = 50
    else:
        y = 35

    rect((25, y), 50, 30)

if __name__ == '__main__':
    run()
```

2.4.5 Events

A category of functions called events alter the normal flow of a program when an action such as a key press or mouse movement takes place. An event is a polite interruption of the normal flow of a program. Key presses and mouse movements are stored until the end of `draw()`, where they can take action that won't disturb drawing that's currently in progress. The code inside an event function is run once each time the corresponding event occurs. For example, if a mouse button is pressed, the code inside the `mouse_pressed()` function will run once and will not run again until the button is pressed again. This allows data produced by the mouse and keyboard to be read independently from what is happening in the rest of the program.

2.4.6 Mouse events

The mouse event functions are `mouse_pressed()`, `mouse_released()`, `mouse_moved()`, and `mouse_dragged()`:

mouse_pressed() Code inside this block is run one time when a mouse button is pressed mouse_released() Code inside this block is run one time when a mouse button is released mouse_moved() Code inside this block is run one time when the mouse is moved mouse_dragged() Code inside this block is run one time when the mouse is moved while a mouse button is pressed

The `mouse_pressed()` function works differently than the `mouse_is_pressed` variable. The value of the `mouse_is_pressed` variable is true until the mouse button is released. It can therefore be used within `draw()` to have a line of code run while the mouse is pressed. In contrast, the code inside the `mouse_pressed()` function only runs once when a button is pressed. This makes it useful when a mouse click is used to trigger an action, such as clearing the screen. In the following example, the background value becomes lighter each time a mouse button is pressed. Run the example on your computer to see the change in response to your finger.

```
from p5 import *

gray = 0

def setup():
    size(100, 100)

def draw():
    background(gray)

def mouse_pressed():
    global gray
    gray += 20

if __name__ == '__main__':
    run()
```

The following example is the same as the one above, but the gray variable is set in the `mouse_released()` event function, which is called once every time a button is released. This difference can be seen only by running the program and clicking the mouse button. Keep the mouse button pressed for a long time and notice that the background value changes only when the button is released.

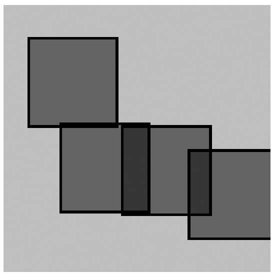
```
from p5 import *

gray = 0

def setup():
    size(100, 100)

def draw():
    background(gray)

def mouse_released():
```

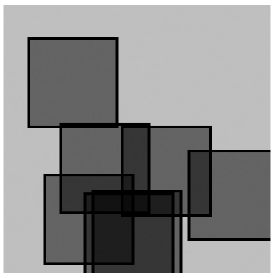
(continues on next page)

(continued from previous page)

```
    global gray
    gray += 20

if __name__ == '__main__':
    run()
```

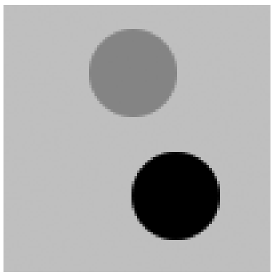
It is generally not a good idea to draw inside an event function, but it can be done under certain conditions. Before drawing inside these functions, it's important to think about the flow of the program. In this example, squares are drawn inside `mouse_pressed()` and they remain on screen because there is no `background()` inside `draw()`. But if `background()` is used, visual elements drawn within one of the mouse event functions will appear on screen for only a single frame, or, by default, 1/60th of a second. In fact, you'll notice this example has nothing at all inside `draw()`, but it needs to be there to force Processing to keep listening for the events. If a `background()` function were run inside draw(), the rectangles would flash onto the screen and disappear.

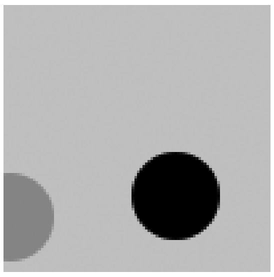


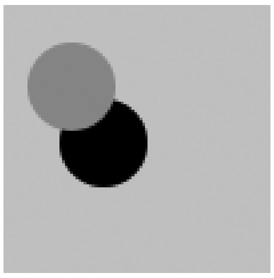


```
from p5 import *

def setup():
    size(100, 100)
    fill(0, 102)

def draw():
    # Empty draw() keeps the program running
    pass

def mouse_pressed():
    rect((mouse_x, mouse_y), 33, 33)

if __name__ == '__main__':
    run()
```

The code inside the `mouse_moved()` and `mouse_dragged()` event functions are run when there is a change in the mouse position. The code in the `mouse_moved()` block is run at the end of each frame when the mouse moves and no button is pressed. The code in the `mouse_dragged()` block does the same when the mouse button is pressed. If the mouse stays in the same position from frame to frame, the code inside these functions does not run. In this example, the gray circle follows the mouse when the button is not pressed, and the black circle follows the mouse when a mouse button is pressed.







```
from p5 import *

dragX, dragY, moveX, moveY = (0, 0, 0, 0)

def setup():
    size(100, 100)
    no_stroke()

def draw():
    background(204)
    fill(0)

    ellipse((dragX, dragY), 33, 33) # Black circle
    fill(153)
    ellipse((moveX, moveY), 33, 33) # Gray circle

def mouse_moved():
    global moveX, moveY
    moveX = mouse_x
```

(continues on next page)

(continued from previous page)

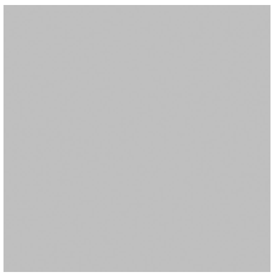
```
    moveY = mouse_y

def mouse_dragged():
    global dragX, dragY
    dragX = mouse_x
    dragY = mouse_y

if __name__ == '__main__':
    run()
```

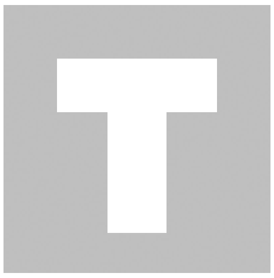
2.4.7 Key events

Each key press is registered through the keyboard event functions `key_pressed()` and `key_released()`:

key_pressed() Code inside this block is run one time when any key is pressed key_released() Code inside this block is run one time when any key is released

Each time a key is pressed, the code inside the `key_pressed()` block is run once. Within this block, it's possible to test which key has been pressed and to use this value for any purpose. If a key is held down for an extended time, the code inside the `key_pressed()` block might run many times in a rapid succession because most operating systems will take over and repeatedly call the `key_pressed()` function. The amount of time it takes to start repeating and the rate of repetitions will be different from computer to computer, depending on the keyboard preference settings. In this example, the value of the boolean variable drawT is set from false to true when the T key is pressed; this causes the lines of code to render the rectangles in `draw()` to start running.





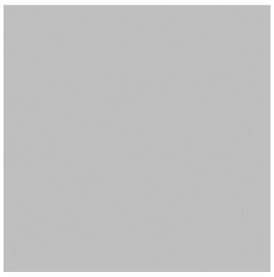
```
from p5 import *

drawT = False
```

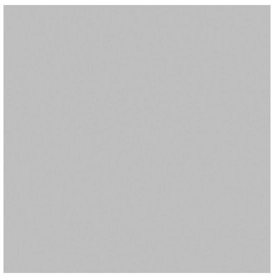
(continues on next page)

(continued from previous page)

```
def setup():
    size(100, 100)
    no_stroke()

def draw():
    background(204)

    global drawT
    if drawT:
        rect((20, 20), 60, 20)
        rect((39, 40), 22, 45)


def key_pressed():
    global drawT
    if key == "T" or key == "t":
        drawT = True

if __name__ == '__main__':
    run()
```

Each time a key is released, the code inside the `key_released()` block is run once. The following example builds on the previous code; each time the key is released the boolean variable `drawT` is set back to false to stop the shape from displaying within `draw()`.





```
from p5 import *

drawT = False

def setup():
   size(100, 100)
   no_stroke()

def draw():
   background(204)

   global drawT
   if drawT:
      rect((20, 20), 60, 20)
      rect((39, 40), 22, 45)


def key_pressed():
   global drawT
   if key == "T" or key == "t":
      drawT = True

def key_released():
   global drawT
   drawT = False

if __name__ == '__main__':
   run()
```

2.4.8 Event flow

As discussed previously, programs written with `draw()` display frames to the screen sixty frames each second. The `frame_rate()` function is used to set a limit on the number of frames that will display each second, and the `no_loop()` function can be used to stop `draw()` from looping. The additional functions `loop()` and `redraw()` provide more options when used in combination with the mouse and keyboard event functions. If a program has been paused with `no_loop()`, running `loop()` resumes its action. Because the event functions are the only elements that continue to run when a program is paused with `no_loop()`, the `loop()` function can be used within these events to continue running the code in `draw()`. The following example runs the `draw()` function for about two seconds each time a mouse button is pressed and then pauses the program after that time has elapsed.

```
from p5 import *

frame = 0

def setup():
   size(100, 100)
```

(continues on next page)

(continued from previous page)

```
def draw():
    global frame
    if frame > 120: #  If 120 frames since the mouse
        no_loop() # was pressed, stop the program
        background(0) # and turn the background black.
    else:
        background(204) # to light gray and draw lines
        line((mouse_x, 0), (mouse_x, 100)) # at the mouse position
        line((0, mouse_y), (100, mouse_y))
        frame += 1

def mouse_pressed():
    global frame
    loop()
    frame = 0

if __name__ == '__main__':
    run()
```

The `redraw()` function runs the code in `draw()` one time and then halts the execution. It's helpful when the display needn't be updated continuously. The following example runs the code in `draw()` once each time a mouse button is pressed.

```
from p5 import *

def setup():
    size(100, 100)
    no_loop()

def draw():
    background(204)
    line((mouse_x, 0), (mouse_x, 100))
    line((0, mouse_y), (100, mouse_y))

def mouse_pressed():
    redraw() # Run the code in draw one time

if __name__ == '__main__':
    run()
```

2.5 Typography

Authors Casey Reas and Ben Fry; Arihant Parsoya (p5 port)

Copyright This tutorial is "Typography" from Processing: A Programming Handbook for Visual Designers and Artists, Second Edition, published by MIT Press. © 2014 MIT Press. If you see any errors or have comments, please let us know. The tutorial was ported to p5 by Arihant Parsoya. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

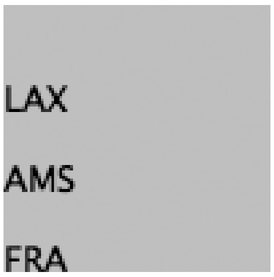
The evolution of typographic reproduction and display technologies has and continues to impact human culture. Early printing techniques developed by Johannes Gutenberg in fifteenth-century Germany using letters cast from lead provided a catalyst for increased literacy and the scientific revolution. Automated typesetting machines, such as the Linotype invented in the nineteenth century, changed the way information was produced, distributed, and consumed. In the digital era, the way we consume text has changed drastically since the proliferation of personal computers in the 1980s and the rapid growth of the Internet in the 1990s. Text from emails, websites, and instant messages fill computer screens, and while many of the typographic rules of the past apply, type on screen requires additional considerations for enhanced communication and legibility.

Letters on screen are created by setting the color of pixels. The quality of the typography is constrained by the resolution of the screen. Because, historically, screens have a low resolution in comparison to paper, techniques have been developed to enhance the appearance of type on screen. The fonts on the earliest Apple Macintosh computers comprised small bitmap images created at specific sizes like 10, 12, and 24 points. Using this technology, a variation of each font was designed for each size of a particular typeface. For example, the character A in the San Francisco typeface used a different image to display the character at size 12 and 18. When the LaserWriter printer was introduced in 1985, Postscript technology defined fonts with a mathematical description of each character's outline. This allowed type on screen to scale to large sizes and still look smooth. Apple and Microsoft later developed TrueType, another outline font format. More recently, these technologies were merged into the OpenType format. In the meantime, methods to smooth text on screen were introduced. These anti-aliasing techniques use gray pixels at the edge of characters to compensate for low screen resolution.

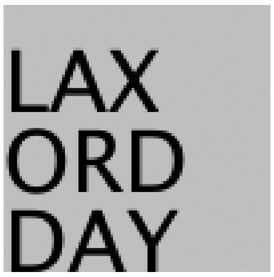
The proliferation of personal computers in the mid-1980s spawned a period of rapid typographic experimentation. Digital typefaces are software, and the old rules of metal and photo type no longer apply. The Dutch typographers known as LettError explain, "The industrial methods of producing typography meant that all letters had to be identical… Typography is now produced with sophisticated equipment that doesn't impose such rules. The only limitations are in our expectations."1 LettError expanded the possibilities of typography with their typeface Beowolf (p. 131). It printed every letter differently so that each time an A is printed, for example, it will have a different shape. During this time, typographers such as Zuzana Licko and Barry Deck created innovative typefaces with the assistance of new software tools. The flexibility of software has enabled extensive font revivals and historic homages such as Adobe Garamond from Robert Slimbach and The Proteus Project from Jonathan Hoefler. Typographic nuances such as ligatures—connections between letter pairs such as fi and æ—made impractical by modern mechanized typography are flourishing again through software font tools.

2.5.1 Draw text

The `text()` function is used to draw letters, words, and paragraphs to the screen. In the simplest use, the first parameter can be a String, char, int, or float. The second and third parameters set the position of the text. By default, the second parameter defines the distance from the left edge of the window; the third parameter defines the distance from the text's baseline to the top of the window. The `text_size()` function defines the size the letters will draw in units of pixels. The number used to define the text size will not be the precise height of each letter, the difference depends on the design of each font. For instance, the statement `text_size(30)` won't necessarily draw a capital H at 30 pixels high. The `fill()` function controls the color and transparency of text. This function affects text the same way it affects shapes such as `rect()` and `ellipse()`, but text is not affected by `stroke()` (this feature will be implemented in later versions).



```
from p5 import *

def setup():
    size(100, 100)

def draw():
    fill(0)
    text("LAX", (0, 40)) #  Write "LAX" at coordinate (0,40)
    text("AMS", (0, 70))  # Write "AMS" at coordinate (0,70)
    text("FRA", (0, 100)) # Write "FRA" at coordinate (0,100)

if __name__ == '__main__':
    run()
```

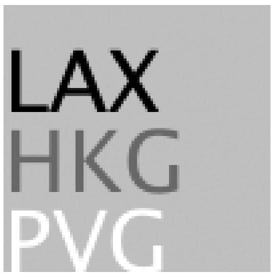


```
from p5 import *

def setup():
    size(100, 100)

def draw():
    text_size(32)

    fill(0)
```

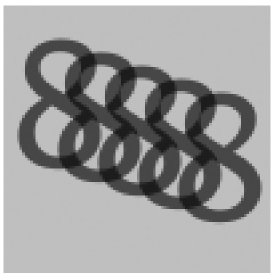
(continues on next page)

(continued from previous page)

```
    text("LAX", (0, 40))
    text("AMS", (0, 70))
    text("FRA", (0, 100))

if __name__ == '__main__':
    run()
```



```
from p5 import *

def setup():
    size(100, 100)

def draw():
    text_size(32)

    fill(0)
    text("LAX", (0, 40))
    fill(126)
    text("AMS", (0, 70))
    fill(255)
    text("FRA", (0, 100))

if __name__ == '__main__':
    run()
```



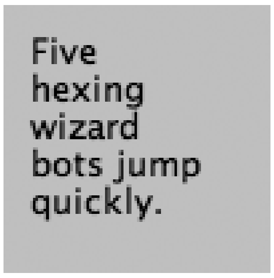
```
from p5 import *

def setup():
    size(100, 100)
```

(continues on next page)

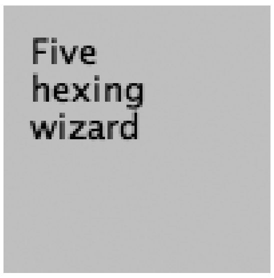
(continued from previous page)

```python

def draw():
    text_size(64)
    fill(0, 140) # Fill black with low opacity
    text("8", (0, 60))
    text("8", (15, 65))
    text("8", (30, 70))
    text("8", (45, 75))
    text("8", (60, 80))

if __name__ == '__main__':
    run()
```

Another version of `text()` draws the characters inside a rectangle. In this use, the second and third parameters define the position of the upper-left corner of the box and fourth and fifth parameters define the width and height of the box. If the length of the text exceeds the dimensions of the defined box, the text will not display.



```python
from p5 import *

def setup():
    size(100, 100)

def draw():
    s = "Five hexing wizard bots jump quickly."
    fill(0)
    text(s, (10, 10), (60, 80))

if __name__ == '__main__':
    run()
```



```
from p5 import *

def setup():
   size(100, 100)

def draw():
   s = "Five hexing wizard bots jump quickly."
   fill(0)
   text(s, 10, 10, 60, 55) #  Box too small

if __name__ == '__main__':
   run()
```

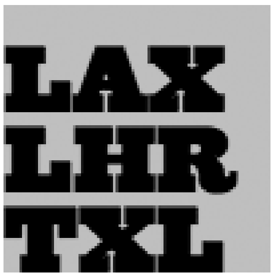
The examples in this chapter are the first to load external media into a sketch. Up to now, all examples have used only graphics generated within Processing through drawing functions such as `line()` and `ellipse()`. Processing is capable of loading and displaying other media, including fonts, images, vector files, formatted data, and sounds. While this chapter focuses on loading fonts and other chapters discuss specific information about other media types, there are a few things about loading media that apply to all categories. These similarities are discussed here.

Before external media can be used in a Processing sketch, it needs to be loaded each time the program is run. Media can be loaded directly from a sketch's folder, another location on the computer, or though the Internet. Most typically, the media is loaded directly from the sketch's folder.

To make media files accessible from anywhere in a program, they are typically declared as globally available variables outside of `setup()` and `draw()`. Files are usually loaded inside `setup()` because they need only be loaded once and because it takes time to load them. Loading a file inside draw() reduces the frame rate of a program because it causes the file to reload each frame. Once a file is loaded in `setup()`, it may be utilized anywhere in the program. In most Processing programs, all files are loaded when the program starts.

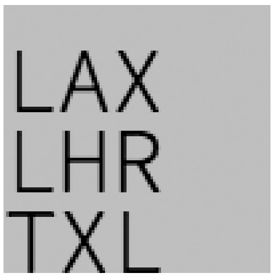
2.5.2 Vector Fonts

Before a font is used in a program, it must be converted and set as the current font. Processing has a unique data type called PFont to store font data. Make a new variable of the type PFont and use the `create_font()` function to convert the font. The first parameter to `create_font()` is the name of the font to convert and the second parameter defines the base size of the font. (Optional third and fourth parameters are defined in the Reference.) The `text_font()` function must then be used to set the current font.



```python
from p5 import *

zigBlack = None

def setup():
    size(100, 100)
    zigBlack = create_font("Ziggurat-Black.otf", 32)
    text_font(zigBlack)
    fill(0)

def draw():
    background(204)
    text("LAX", (0, 40))
    text("LHR", (0, 70))
    text("TXL", (0, 100))

if __name__ == '__main__':
    run()
```

To ensure a font will load on all computers, regardless if the font is installed, add the file to the sketch's data folder. When fonts inside the data folder are used, the complete file name, including the data type extension, needs to be written as the parameter to `create_font()`. The following example is similar to the prior example, but it uses an OpenType font inside the data folder. It uses Source Code Pro, an open source typeface that can be found online and downloaded through a web browser.



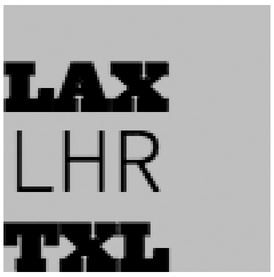
```python
from p5 import *

sourceLight = None

def setup():
    size(100, 100)
    sourceLight = create_font("SourceCodePro-Light.otf", 34)
    text_font(sourceLight)
    fill(0)

def draw():
    background(204)
    text("LAX", (0, 40))
    text("LHR", (0, 70))
    text("TXL", (0, 100))
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    run()
```

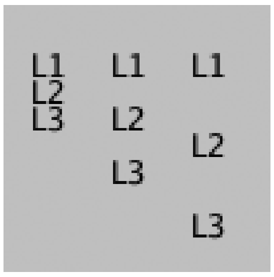
To use two fonts in one program, create two PFont variables and use the textFont() function to change the current font. Based on the prior two examples, the Ziggurat-Black font loads from its location on the local computer and Source Code Pro loads from the data folder.



```
from p5 import *

sourceLight = None
zigBlack = None

def setup():
    size(100, 100)
    sourceLight = create_font("SourceCodePro-Light.otf", 34)
    zigBlack = create_font("SourceCodePro-Light.otf", 44)
    text_font(sourceLight)
    fill(0)

def draw():
    background(204)
    text_font(zigBlack)
    text("LAX", (0, 40))
    text_font(sourceLight)
    text("LHR", (0, 70))
    text_font(zigBlack)
    text("TXL", (0, 100))

if __name__ == '__main__':
    run()
```

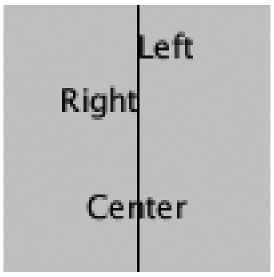
2.5.3 Text attributes

Processing includes functions to control the leading (the spacing between lines of text) and alignment. Processing can also calculate the width of any character or group of characters, a useful technique for arranging shapes and typographic elements. The `text_leading()` function sets the spacing between lines of text. It has one parameter that defines this space in units of pixels.



```
from p5 import *

def setup():
    size(100, 100)
    fill(0)

def draw():
    lines = "L1 L2 L3"
    text_size(12)
    fill(0)
    textLeading(10)
    text(lines, (10, 15), (30, 100))
    textLeading(20)
    text(lines, (40, 15), (30, 100))
    textLeading(30)
    text(lines, (70, 15), (30, 100))

if __name__ == '__main__':
    run()
```

Letters and words can be drawn from their center, left, and right edges. The `text_align()` function sets the alignment for drawing text through its parameter, which can be LEFT, CENTER, or RIGHT. It sets the display characteristics of the letters in relation to the x-coordinate stated in the `text()` function.



```
from p5 import *

def setup():
    size(100, 100)
    fill(0)

def draw():
    text_size(12)
    fill(0)
```

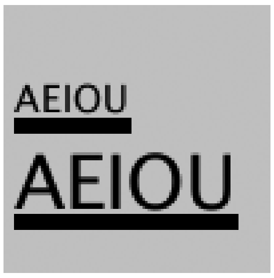
(continues on next page)

(continued from previous page)

```
    line((50, 0), (50, 100))
    text_align("LEFT")
    text("Left", (50, 20))
    text_align("RIGHT")
    text("Right", (50, 40))
    text_align("CENTER")
    text("Center", (50, 80))

if __name__ == '__main__':
    run()
```

The settings for `text_size()`, `text_leading()`, and `text_align()` will be used for all subsequent calls to the `text()` function. However, note that the `text_size()` function will reset the text leading, and the `text_font()` function will reset both the size and the leading.

The `text_width()` function calculates and returns the pixel width of any character or text string. This number is calculated from the current font and size as defined by the `text_font()` and `text_size()` functions. Because the letters of every font are a different size and letters within many fonts have different widths, this function is the only way to know how wide a string or character is when displayed on screen. For this reason, always use `text_width()` to position elements relative to text, rather than hard-coding them into your program.



```
from p5 import *

def setup():
    size(100, 100)
    fill(0)

def draw():
    s = "AEIOU"
    fill(0)

    text_size(14)
    tw = text_width(s)
    text(s, (4, 40))
    rect((4, 42), tw, 5)

    text_size(28)
```

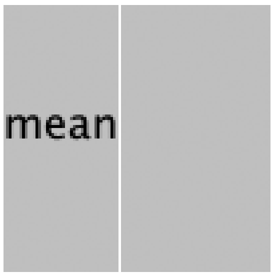
(continues on next page)

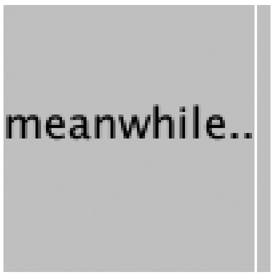
(continued from previous page)

```
    tw = text_width(s)
    text(s, (4, 76))
    rect((4, 78), tw, 5)


if __name__ == '__main__':
    run()
```

2.5.4 Typing

Drawing letters to the screen becomes more engaging when used in combination with the keyboard. The `key_pressed()` event function introduced on page 97 can be used to record each letter as it is typed. The following two examples use this function to read and analyze input from the keyboard by using the String methods introduced in the Text chapter (p. 143). In both, the String variable letters starts empty. Each key typed is added to the end of the string. The first example displays the string as it grows as keys are pressed and removes letters from the end when backspace is pressed. The second example builds on the first—when the Return or Enter key is pressed, the program checks if the word "gray" or "black" was typed. If one of these words was input, the background changes to that value.



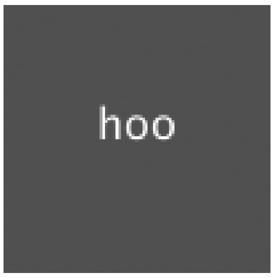


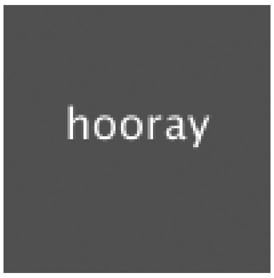
```
from p5 import *

letters = ""

def setup():
    size(100, 100)
    stroke(255)
    fill(0)
    text_size(16)

def draw():
```

(continues on next page)

(continued from previous page)

```
    background(204)
    cursorPosition = text_width(letters)
    line((cursorPosition, 0), (cursorPosition, 100))
    text(letters, (0, 50))

def key_pressed():
    global letters
    if key == "BACKSPACE":
        if len(letters) > 0:
            letters = letters[:-1]
    elif text_width(letters+str(key)) < width:
        letters = letters + str(key)

if __name__ == '__main__':
    run()
```





```
from p5 import *

letters = ""
back = 102

def setup():
    size(100, 100)
    text_align("CENTER")
    text_size(16)

def draw():
    background(back)
    text(letters, (50, 50))

def key_pressed():
```

(continues on next page)

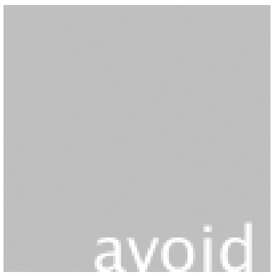
(continued from previous page)

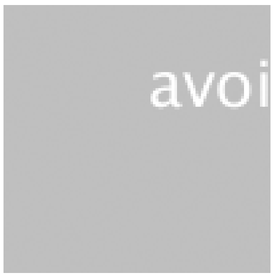
```
    global letters
    if key == "ENTER" or key == "RETURN":
        letters = letters.lower()
        print(letters)
        if letters == "black":
            back = 0
        elif letters == "gray":
            back = 204

        letters = ""
    elif ord(str(key)) > 31 and key != "CODED":
        # If the key is alphanumeric, add it to the String
        letters = letters + str(key)

if __name__ == '__main__':
    run()
```

Many people spend hours a day inputting letters into computers, but this action is very constrained. What features could be added to a text editor to make it more responsive to the typist? For example, the speed of typing could decrease the size of the letters, or a long pause in typing could add many spaces, mimicking a person's pause while speaking. What if the keyboard could register how hard a person is typing (the way a piano plays a soft note when a key is pressed gently) and could automatically assign attributes such as italics for soft presses and bold for forceful presses? These analogies suggest how conservatively current software treats typography and typing.

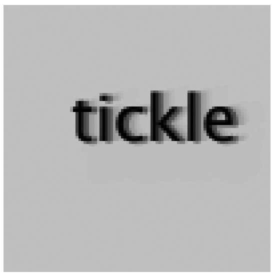
Many artists and designers are fascinated with type and have created unique ways of exploring letterforms with the mouse, keyboard, and more exotic input devices. A minimal yet engaging example is John Maeda's Type, Tap, Write software, created in 1998 as homage to manual typewriters. This software uses the keyboard as the input to a black-and-white screen representation of a keyboard. Pressing the number keys cause the software to cycle through different modes, each revealing a playful interpretation of keyboard data. In Jeffrey Shaw and Dirk Groeneveld's The Legible City (1989–91), buildings are replaced with three-dimensional letters to create a city of typography that conforms to the streets of a real place. In the Manhattan version, for instance, texts from the mayor, a taxi driver, and Frank Lloyd Wright comprise the city. The image is presented on a projection screen, and the user navigates by pedaling and steering a stationary bicycle situated in front of the projected image. Projects such as these demonstrate that software presents an extraordinary opportunity to extend the way we read and write.

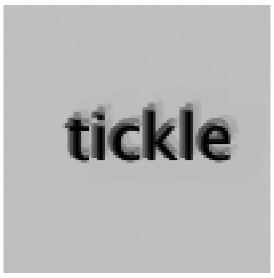
Typographic elements can be assigned behaviors that define a personality in relation to the mouse or keyboard. A word can express aggression by moving quickly toward the mouse, or moving away slowly can express timidity. The following examples demonstrate basic applications of this area. In the first, the word avoid stays away from the mouse because its position is set to the inverse of the cursor position. In the second, the word tickle jitters when the cursor hovers over its position.





```
from p5 import *

def setup():
    size(100, 100)
    text_align("CENTER")
    text_size(24)

def draw():
    background(204)
    text("avoid", (width-mouse_x, height-mouse_y))


if __name__ == '__main__':
    run()
```





```
from p5 import *

x = 33
y = 60

def setup():
    size(100, 100)
    #text_size(24)
    no_stroke()

def draw():
    global x, y
    background(204, 120)

    fill(0)
    # If cursor is over the text, change the position
    if mouse_x >= x and mouse_x <= x + 55 and mouse_y >= y - 24 and␣
↪mouse_y <= y:
        x += random_uniform(-2, 2)
        y += random_uniform(-2, 2)

    text("tickle", (x, y))

if __name__ == '__main__':
    run()
```

2.6 Strings and Drawing Text

Authors Daniel Shiffman; Arihant Parsoya (p5 port)

Copyright This tutorial is from the book Learning Processing by Daniel Shiffman, published by Morgan Kaufmann, © 2008 Elsevier Inc. All rights reserved. The tutorial was ported to p5 by Arihant Parsoya. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

If you are looking to display text onscreen with Processing, you've got to first become familiar with the String class. Strings are probably not a totally new concept for you, it's quite likely you've dealt with them before. For example, if you've printed some text to the message window or loaded an image from a file, you've written code like so:

```
print("printing some text to the message window!")      # Printing␣
↪a String
img = load_image("filename.jpg")  # Using a String for a file name
```

Nevertheless, although you may have used a String here and there, it's time to unleash their full potential.

2.6.1 Where do we find documentation for the String class?

Although technically a Python class, because Strings are so commonly used, Processing includes documentation in its reference: http://www.processing.org/reference/String.html.

This page only covers some of the available methods of the String class. The full documentation can be found on Python's String page.

2.6.2 What is a String?

A String, at its core, is really just a fancy way of storing an array of characters. If we didn't have the String class, we'd probably have to write some code like this:

```
sometext = ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
```

Clearly, this would be a royal pain in the Processing behind. It's much simpler to do the following and make a String object:

```
sometext = "How do I make String? Type some characters between␣
↪quotation marks!"
```

It appears from the above that a String is nothing more than a list of characters in between quotes. Nevertheless, this is only the data of a String. We must remember that a String is an object with methods (which you can find on the reference page.) This is just like how we learned in the Pixels tutorial that a PImage stores both the data associated with an image as well as the functionality: `copy()`, `load_pixels()`, etc.

For example, brackets *[]* returns the individual character in the String at a given index. Note that Strings are just like arrays in that the first character is index #0!

```
message = "some text here."
c = message[0]
print(c)  # Results in 'e'
```

Another useful method is `len()`. This is easy to confuse with the length property of an array. However, when we ask for the length of a String object, we must use the parentheses since we are calling a function called `len()` rather than accessing a property called length.

```
message = "This String is 34 characters long."
print(len(message))
```

We can also change a String to all uppercase using the `upper()` method:

```
uppercase = message.upper()
print(uppercase)
```

Finally, let's look at *equals()*‘. Now, Strings can be compared with the "==" operator as follows:

```
one = "hello"
two = "hello"
print(one == two)
```

One other feature of String objects is concatenation, joining two Strings together. Strings are joined with the "+" operator. Plus, of course, usually means add in the case of numbers. When used with Strings, it means join.

```
helloworld = "Hello" + "World"
```

Variables can also be brought into a String using concatenation.

```
x = 10
message = "The value of x is: " + x
```

2.6.3 Displaying Text

The easiest way to display a String is to print it in the message window. This is likely something you've done while debugging. For example, if you needed to know the horizontal mouse location, you would write:

```
print(mouse_x)
```

Or if you needed to determine that a certain part of the code was executed, you might print out a descriptive message.

```
print("We got here and we're printing out the mouse location!!!")
```

While this is valuable for debugging, it's not going to help our goal of displaying text for a user. To place text on screen, we have to follow a series of simple steps.

1. Declare an object for font

```
f = None
```

2. Create the font by referencing the font name and the function create_font()

This should be done only once, usually in `setup()`. Just as with loading an image, the process of loading a font into memory is slow and would seriously affect the sketch's performance if placed inside `draw()`. When sharing a sketch with other people or posting it on the web, you may need to include a .ttf or .otf version of your font in the data directory of the sketch because other people might not have the font installed on their computer. Only fonts that can legally be distributed should be included with a sketch. In addition to the name of the font, you can specify the size as well as whether the font should be antialiased or not.

```
f = create_font("Arial.ttf", 16,) # Arial, 16 point, anti-aliasing␣
↪on
```

3. Specify the font using text_font()

`text_font()` takes one or two arguments, the font variable and the font size, which is optional. If you do not include the font size, the font will be displayed at the size originally loaded. When possible, the `text()` function will use a native font rather than the bitmapped version created behind the scenes with `create_font()` so you have the opportunity to scale the font dynamically. When using P2D, the actual native version of the font will be employed by the sketch, improving drawing quality and performance. With the P3D renderer, the bitmapped version will be used and therefore specifying a font size that is different from the font size loaded can result in pixelated text.

```
text_font(f, 36)
```

4. Specify a color using fill()

```
fill(255)
```

5. Call the text() function to display text

This function is just like shape or image drawing, it takes three arguments—the text to be displayed, and the x and y coordinate to display that text.

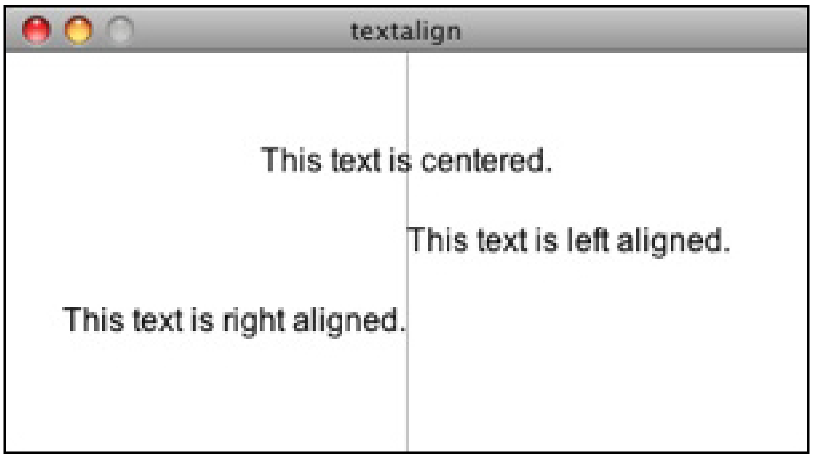
```
text("Hello Strings!", (10, 100))
```

```
from p5 import *

f = None # STEP 1 Declare PFont variable

def setup():
        size(200,200)
        f = create_font("Arial", 16) # STEP 2 Create Font

def draw():
        background(255)
        text_font(f, 16)
        fill(0)
        text("Hello Strings!", (10,100))

if __name__ == '__main__':
        run()
```

2.6.4 Animating Text

Let's look at two more useful Processing functions related to displaying text:

`text_align()` - specifies RIGHT, LEFT or CENTER alignment for text.



```
from p5 import *

f = None # STEP 1 Declare PFont variable

def setup():
        global f
        size(400,200)
        f = create_font("Arial.ttf", 16) # STEP 2 Create Font

def draw():
        global f
        background(255)

        stroke(175)
        line((width/2,0), (width/2,height))

        text_font(f)
        fill(0)

        text_align("CENTER")
        text("This text is centered.",(width/2,60))

        text_align("LEFT")
        text("This text is left aligned.", (width/2,100))

        text_align("RIGHT")
        text("This text is right aligned.", (width/2,140))

if __name__ == '__main__':
        run()
```

`text_width()` - Calculates and returns the width of any character or text string.

Let's say we want to create a news ticker, where text scrolls across the bottom of the screen from left to right. When the news headline leaves the window, it reappears on the right hand side and scrolls again. If we know the x location of the beginning of the text and we know the width of that text, we can determine when it is no longer in view. textWidth() gives us that width.

To start, we declare headline, font, and x location variables, initializing them in setup().

```
headline = "New study shows computer programming lowers cholesterol.
↪"
f  # Global font variable
x  # horizontal location of headline

def setup():
        global f, x
        f = create_font("Arial",16) #Loading font
        x = width # initializing headline off-screen to the right
```

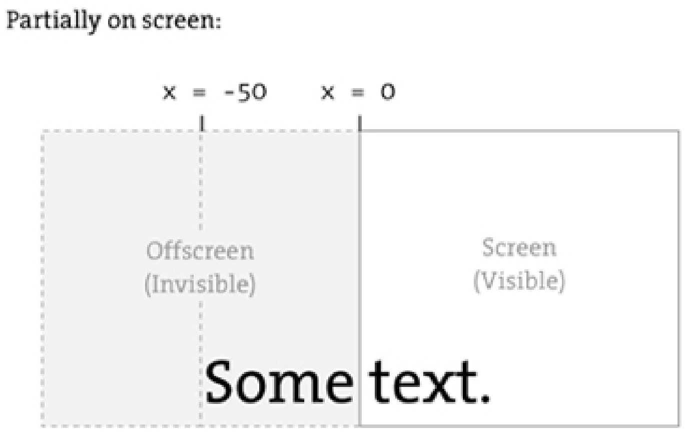
In `draw()`, we display the text at the appropriate location.

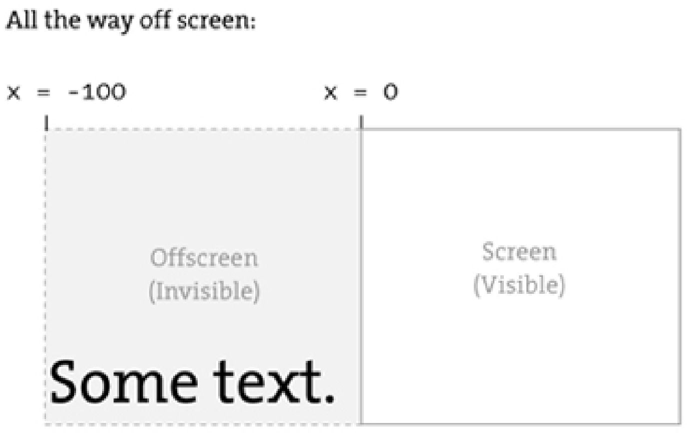
```
# Display headline at x  location
text_font(f, 16)
text_align("LEFT")
text(headline, (x,180))
```

We change x by a speed value (in this case a negative number so that the text moves to the left.)

```
x = x - 3
```

Now comes more difficult part. It was easy to test when a circle reached the left side of the screen. We would simply ask: is x less than 0? With text, however, since it is left-aligned, when x equals zero, it is still viewable on screen. Instead, the text will be invisible when x is less than 0 minus the width of the text (See figure below). When that is the case, we reset x back to the right-hand side of the window, i.e. width.





```
# If x is less than the negative width, then it is completely off␣
↪the screen
w = text_width(headline)
if x < -w:
        x = width
```

Here's the full example that displays a different headline each time the previous headline leaves the screen. The headlines are stored in a String array.

```
from p5 import *

headlines = [
        "Processing downloads break downloading record.",
        "New study shows computer programming lowers cholesterol."]

f = None # Global font variable
x = None # horizontal location of headline
index = 0

def setup():
```

(continues on next page)

(continued from previous page)

```
        global f, x
        size(400,200)
        f = create_font("Arial.ttf", 16)

        # Initialize headline offscreen to the right
        x = width

def draw():
        global f, x, index
        background(255)

        fill(0)

        # Display headline at x  location
        text_font(f,16)
        text_align("LEFT")
        text(headlines[index],(x,180))

        # Decrement x
        x = x - 3

        # If x is less than the negative width
        # then it is off the screen
        w = text_width(headlines[index])
        if x < -w:
                x = width
                index = (index + 1) % len(headlines)

if __name__ == '__main__':
        run()
```

In addition to `text_align()` and `text_width()`, Processing also offers the functions `text_leading()`, `text_mode()`, `text_size()` for additional display functionality.

2.6.5 Rotating text

Translation and rotation can also be applied to text. For example, to rotate text around its center, translate to an origin point and use textAlign(CENTER) before displaying the text.

```
from p5 import *

message = "this text is spinning"
f = None
theta = 0

def setup():
        global f
        size(200, 200)
```

(continues on next page)

(continued from previous page)

```
        f = create_font("Arial.ttf", 16)


def draw():
        global f, theta
        background(255)
        fill(0)
        text_font(f)                      # Set the font
        translate(width/2,height/2)  # Translate to the center
        rotate(theta)                # Rotate by theta
        text_align("CENTER")
        text(message,(0,0))
        theta += 0.05                # Increase rotation

if __name__ == '__main__':
        run()
```

2.6.6 Displaying text character by character

In certain graphics applications, displaying text with each character rendered individually is required. For example, if each character needs to move or be colored independently then simply saying…

```
text("a bunch of letters", (0,0))
```

will not do.

The solution is to loop through a String, displaying each character one at a time.

Let's start by looking at an example that displays the text all at once.

```
from p5 import *

message = "Each character is not written individually."
f = None

def setup():
        global f
        size(400, 200)
        f = create_font("Arial.ttf", 16)


def draw():
        global f, theta
        background(255)
        fill(0)
        text_font(f)

```

(continues on next page)

(continued from previous page)

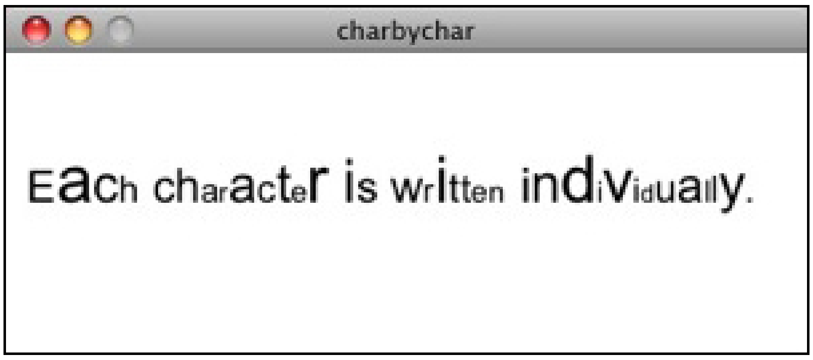
```
        # Displaying a block of text all at once using text().
        text(message, (10,height/2))

if __name__ == '__main__':
        run()
```

We can rewrite the code to display each character in loop:

```
message = "Each character is written individually."

# The first character is at pixel 10
x = 10

for i in range(len(message)):
        # Each character is displayed one at a time
        text(message[i], (x, height/2))
        # All characters are spaced 10 pixels apart.
        x += 10
```

Calling the `text()` function for each character will allow us more flexibility (for coloring, sizing, and placing characters within one String individually). The above code has a pretty major flaw, however—the x location is increased by 10 pixels for each character. Although this is approximately correct, because each character is not exactly ten pixels wide, the spacing is off.

The proper spacing can be achieved using the `text_width()` function as demonstrated in the code below. Note how this example achieves the proper spacing even with each character being a random size!



Each character is written individually.

```
from p5 import *

message = "Each character is not written individually."
f = None

def setup():
        global f
        size(400, 200)
        f = create_font("Arial.ttf", 16)

```

(continues on next page)

(continued from previous page)

```

def draw():
        global f, theta
        background(255)
        fill(0)
        text_font(f)

        x = 10
        for i in range(len(message)):
                text_size(int(random_uniform(12,36)))
                text(message[i], (x,height/2))

                # textWidth() spaces the characters out properly.
                x += text_width(message[i])

        no_loop()

if __name__ == '__main__':
        run()
```

This "letter by letter" methodology can also be applied to a sketch where characters from a String move independently of one another. The following example uses object-oriented design to make each character from the original String a Letter object, allowing it to both be a displayed in its proper location as well as move about the screen individually.

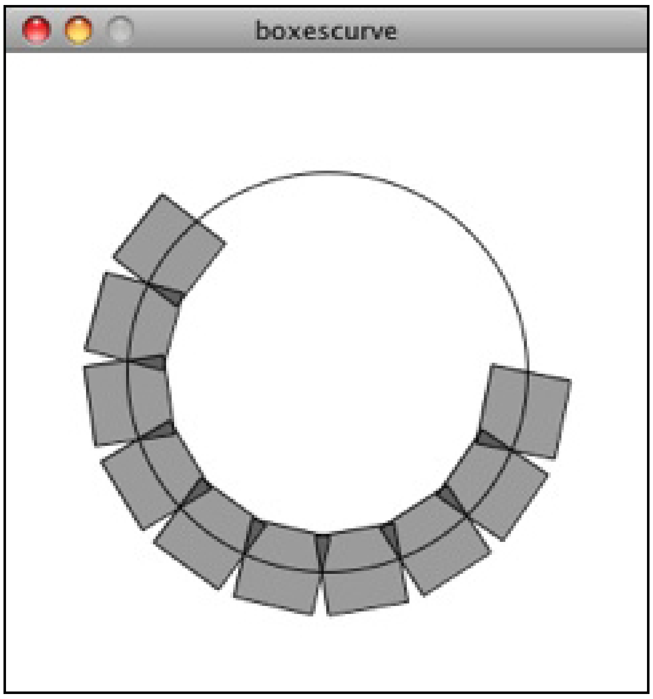
```
from p5 import *

message = "Each character is not written individually."
f = None
letters = []

def setup():
        global f
        size(260, 200)
        f = create_font("Arial.ttf", 16)
        text_font(f)

        # Initialize Letters at the correct x location
        x = 16
        for i in range(len(message)):
                letters.append(Letter(x, 100, message[i]))
                x += text_width(message[i])

def draw():
        global f, letters
        background(255)

        for i in range(len(message)):
```

(continues on next page)

(continued from previous page)

```
                # Display all letters
                letters[i].display()

                # If the mouse is pressed the letters shake
                # If not, they return to their original location
                if mouse_is_pressed:
                        letters[i].shake()
                else:
                        letters[i].home()

class Letter:
        def __init__(self, x_, y_, letter_):
                self.letter = letter_

                self.homex = x_
                self.homey = y_

                self.x = x_
                self.y = y_

        def display(self):
                fill(0)
                text_align("LEFT")
                text(self.letter, (self.x, self.y))

        def shake(self):
                self.x += random_uniform(-2, 2)
                self.y += random_uniform(-2, 2)

        def home(self):
                self.x = self.homex
                self.y = self.homey

if __name__ == '__main__':
        run()
```

The character by character method also allows us to display text along a curve. Before we move on to letters, let's first look at how we would draw a series of boxes along a curve. This example makes heavy use of Trignometry.



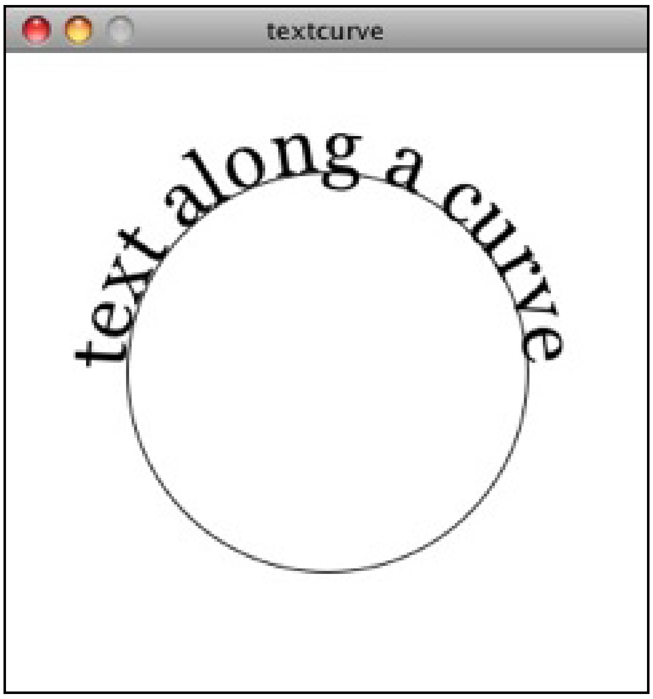
```
from p5 import *

f = None

# The radius of a circle
r = 100

# The width and height of the boxes
w = 40
h = 40

def setup():
        global f
        size(320, 320)
        f = create_font("Arial.ttf", 16)
        text_font(f)

def draw():
        global f, letters
        background(255)

        # Start in the center and draw the circle
        translate(width / 2, height / 2)
        no_fill()
        stroke(0)

        # Our curve is a circle with radius r in the center of the␣
↪window.
        ellipse((0, 0), r*2, r*2)
```

(continues on next page)

(continued from previous page)

```
        # 10 boxes along the curve
        totalBoxes = 10

        # We must keep track of our position along the curve
        arclength = 0

        # For every box
        for i in range(totalBoxes):
                # Each box is centered so we move half the width
                arclength += w/2

                # Angle in radians is the arclength divided by the␣
↪radius
                theta = arclength / r

                with push_matrix():
                        # Polar to cartesian coordinate conversion
                        translate(r*cos(theta), r*sin(theta))

                        # Rotate the box
                        rotate(theta)

                        # Display the box
                        fill(0, 100)

                        rect_mode("CENTER")
                        rect((0,0),w,h)

                # Move halfway again
                arclength += w/2


if __name__ == '__main__':
        run()
```

What we need to do is replace each box with a character from a String that fits inside the box. And since characters all do not have the same width, instead of using a variable "w" that stays constant, each box will have a variable width along the curve according to the `text_width()` function.



```
from p5 import *

f = None

# The radius of a circle
r = 100

# The width and height of the boxes
w = 40
h = 40

message = "text along a curve"

def setup():
        global f
        size(320, 320)
        f = create_font("Arial.ttf", 16)
        text_font(f)

        # The text must be centered!
        text_align("CENTER")

def draw():
        global f, letters
        background(255)

        # Start in the center and draw the circle
        translate(width / 2, height / 2)
        no_fill()
        stroke(0)
```

(continues on next page)

(continued from previous page)

```
        # Our curve is a circle with radius r in the center of the␣
↪window.
        ellipse((0, 0), r*2, r*2)

        # 10 boxes along the curve
        totalBoxes = 10

        # We must keep track of our position along the curve
        arclength = 0

        # For every box
        for i in range(totalBoxes):
                # Instead of a constant width, we check the width␣
↪of each character.
                currentChar = message[i]
                x = text_width(currentChar)

                # Each box is centered so we move half the width
                arclength += w/2

                # Angle in radians is the arclength divided by the␣
↪radius
                # Starting on the left side of the circle by adding␣
↪PI
                theta = PI + arclength / r

                with push_matrix():
                        # Polar to cartesian coordinate conversion
                        translate(r*cos(theta), r*sin(theta))

                        # Rotate the box
                        rotate(theta)

                        # Display the box
                        fill(0, 100)

                        text(currentChar, (0,0))

                # Move halfway again
                arclength += w/2


if __name__ == '__main__':
        run()
```

2.7 Arrays

Authors Casey Reas, Ben Fry; Arihant Parsoya (p5 port)

The term array refers to a structured grouping or an imposing number: "The dinner buffet offers an array of choices," "The city of Boston faces an array of problems." In computer programming, an array is a set of data elements stored under the same name. Arrays can be created to hold any type of data, and each element can be individually assigned and read. There can be arrays of numbers, characters, sentences, boolean values, and so on. Arrays might store vertex data for complex shapes, recent keystrokes from the keyboard, or data read from a file. For instance, an array can store five integers (1919, 1940, 1975, 1976, 1990), the years to date that the Cincinnati Reds won the World Series, rather than defining five separate variables. Let's call this array "dates" and store the values in sequence:

Array elements are numbered starting with zero, which may seem confusing at first but is an important detail for many programming languages. The first element is at position [0], the second is at [1], and so on. The position of each element is determined by its offset from the start of the array. The first element is at position [0] because it has no offset; the second element is at position [1] because it is offset one place from the beginning. The last position in the array is calculated by subtracting 1 from the array length. In this example, the last element is at position [4] because there are five elements in the array.

Arrays can make the task of programming much easier. While it's not necessary to use them, they can be valuable structures for managing data. Let's begin with a set of data points to construct a bar chart.

The following examples to draw this chart demonstrates some of the benefits of using arrays, like avoiding the cumbersome chore of storing data points in individual variables. Because the chart has ten data points, inputting this data into a program requires either creating 10 variables or using one array. The code on the left demonstrates using separate variables. The code on the right shows how the data elements can be logically grouped together in an array.

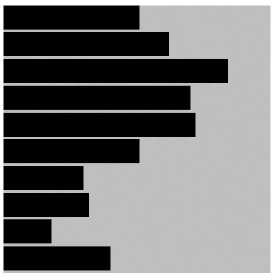
```
x0 = 50
x1 = 61
x2 = 83
x3 = 69
x4 = 71
x5 = 50
x6 = 29
x7 = 31
```

(continues on next page)

(continued from previous page)

```
x8 = 17
x9 = 39

x = [50, 61, 83, 69, 71, 50, 29, 31, 17, 39]
```

Using what we know about drawing without arrays, ten variables are needed to store the data; each variable is used to draw a single rectangle. This is tedious:



```
x0 = 50
x1 = 61
x2 = 83
x3 = 69
x4 = 71
x5 = 50
x6 = 29
x7 = 31
x8 = 17
x9 = 39
fill(0)
rect((0, 0), x0, 8)
rect((0, 10), x1, 8)
rect((0, 20), x2, 8)
rect((0, 30), x3, 8)
rect((0, 40), x4, 8)
rect((0, 50), x5, 8)
rect((0, 60), x6, 8)
rect((0, 70), x7, 8)
rect((0, 80), x8, 8)
rect((0, 90,) x9, 8)
```

In contrast, the following example shows how to use an array within a program. The data for each bar is accessed in sequence with a for loop. The syntax and usage of arrays is discussed in more detail in the following pages.

```
x = [50, 61, 83, 69, 71, 50, 29, 31, 17, 39]

fill(0)
for i in range(len(x)):
        rect((0, i*10), x[i], 8)
```

2.7.1 Define an Array

There are different ways to declare, create, and assign arrays. In the following examples that explain these differences, an array with five elements is created and filled with the values 19, 40, 75, 76, and 90.

Note the different way each technique for creating and assigning elements of the array relates to setup().

```
data = [] # declare

def setup():
        size(100, 100)
        data.append(19)
        data.append(40)
        data.append(75)
        data.append(76)
        data.append(90)

data = [0]*5 # Declare and create

def setup():
        size(100, 100)
        data[0] = 19
        data[1] = 40
        data[2] = 75
        data[3] = 76
        data[4] = 90

data = [19, 40, 75, 76, 90] # Declare and assign

def setup():
        size(100, 100)
```

Although each of the three previous examples defines an array in a different way, they are all equivalent. They show the flexibility allowed in defining the array data. Sometimes, all the data a program will use is known at the start and can be assigned immediately. At other times, the data is generated while the code runs. Each sketch can be approached differently using these techniques.

Arrays can also be used in programs that don't include a setup() and draw(). If arrays are not used with these functions, they can be created and assigned in the ways shown in the following examples.

```
data = [] # Create

data.append(19) # Assign
data.append(40)
data.append(75)
data.append(76)
data.append(90)

data = [0]*5 # Declare
data[0] = 19 # Assign
data[1] = 40
```

(continues on next page)

(continued from previous page)

```
data[2] = 75
data[3] = 76
data[4] = 90

data = [19, 40, 75, 76, 90] # Declare, Create and Assign
```

2.7.2 Read Array Elements

After an array is created, its data can be accessed and used within the code. An array element is accessed with the name of the array variable, followed by brackets around the element position to read.

```
data = [19, 40, 75, 76, 90]

line((data[0], 0), (data[0], 100))
line((data[1], 0), (data[1], 100))
line((data[2], 0), (data[2], 100))
line((data[3], 0), (data[3], 100))
line((data[4], 0), (data[4], 100))
```

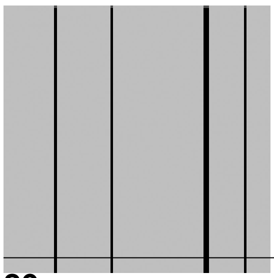
Remember, the first element in the array is in the 0 position. If you try to access a member of the array that lies outside the array boundaries, your program will terminate and give an ArrayIndexOutOfBoundsException.

```
data = [19, 40, 75, 76, 90]
print(data[0]) # prints "19" to the console
print(data[2]) # prints "75" to the console
print(data[5]) # IndexError: list index out of range
```

The `len()` function is used to find the number of elements in an array. The following example demonstrates how to utilize it.

```
data1 = [19, 40, 75, 76, 90]
data2 = [19, 40]

print(len(data1)) # prints "5" to the console
print(len(data2)) # prints "2" to the console
```

Usually, a for loop is used to access array elements, especially with large arrays. The following example draws the same lines as code 28-09 but uses a for loop to iterate through every value in the array.



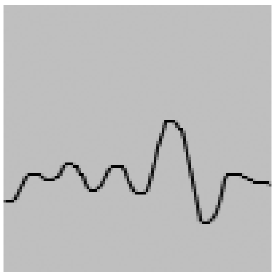
```
data1 = [19, 40, 75, 76, 90]
data2 = [19, 40]

print(len(data1)) # prints "5" to the console
print(len(data2)) # prints "2" to the console
```

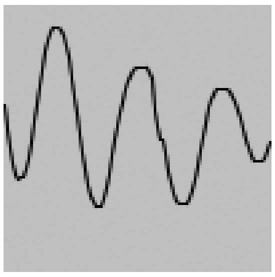
A for loop can also be used to put data inside an array. For instance, it can calculate a series of numbers and then assign each value to an array element. The following example stores the values from the sin() function in an array within setup() and then displays these values as the stroke values for lines within draw().

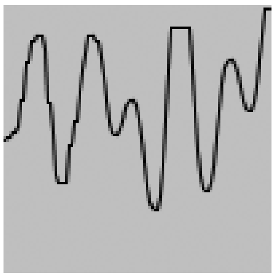
```
from p5 import *

sineWave = []

def setup():
    size(100, 100)
    global sineWave
    for i in range(width):
        r = remap(i, [0, width], [0, TWO_PI])
        sineWave.append(abs(sin(r)))

def draw():
    global sineWave
    for i in range(len(sineWave)):
        ␣
↪       # Set stroke values to numbers read from
        stroke(sineWave[i] * 255)
        line((i, 0), (i, height))

if __name__ == '__main__':
    run()
```

2.7.3 Record Data

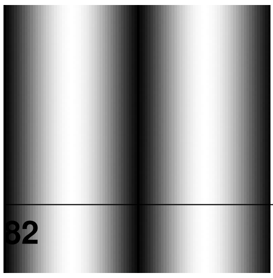
As one example of how arrays may be used, this section shows how to use arrays to store data from the mouse. The pmouseX and pmouseY variables store the cursor coordinates from the previous frame, but there is no built-in way to access the cursor values from earlier frames. At every frame, the mouse_x, mouse_y, pmouse_x, and pmouse_y variables are replaced with new numbers and their previous numbers are discarded. Creating an array is the easiest way to store the history of these values. In the following example, the most recent 100 values from mouseY are stored and displayed on screen as a line from the left to the right edge of the screen. At each frame, the values in the array are shifted to the right and the newest value is added to the beginning.



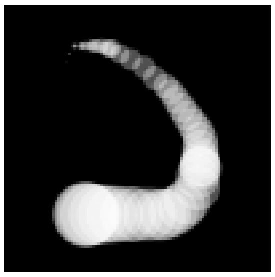


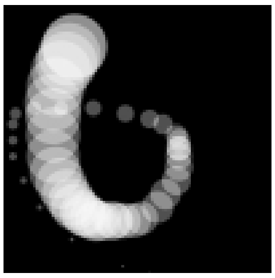


```
from p5 import *

y = []

def setup():
    size(100, 100)

    global y
    y = [0]*width

def draw():
    background(204)

    # Read the array from the end to the
    # beginning to avoid overwriting the data
    for i in range(len(y) - 1, 0, -1):
        y[i] = y[i - 1]

    # Add new values to the beginning
    y[0] = mouse_y

    # Display each pair of values as a line
    for i in range(1, len(y)):
        line((i, y[i]), (i-1, y[i-1]))

if __name__ == '__main__':
    run()
```



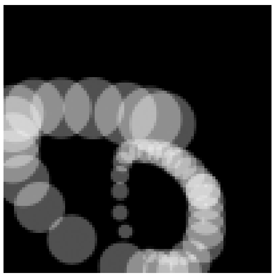
Apply the same code simultaneously to the mouseX and mouseY values to store the position of the cursor. Displaying these values each frame creates a trail behind the cursor.





```
from p5 import *

num = 50
y = [0]*num
x = [0]*num

def setup():
    size(100, 100)

    no_stroke()
    fill(255, 102)

def draw():
    background(0)

    # Shift the values to the right
    for i in range(num - 1, 0, -1):
        y[i] = y[i - 1]
        x[i] = x[i - 1]

    # Add new values to the beginning
    y[0] = mouse_y
    x[0] = mouse_x
```

(continues on next page)



(continued from previous page)

```
            # Draw the circles
            for i in range(1, num):
                ellipse((x[i], y[i]), i/2.0, i/2.0)

        if __name__ == '__main__':
            run()
```

The following example produces the same result as the previous one but uses a more efficient technique. Instead of shifting the array elements in each frame, the program writes the new data to the next available array position. The elements in the array remain in the same position once they are written, but they are read in a different order each frame. Reading begins at the location of the oldest data element and continues to the end of the array. At the end of the array, the % operator (p. 57) is used to wrap back to the beginning. This technique, commonly known as a ring buffer, is especially useful with larger arrays, to avoid unnecessary copying of data that can slow down a program.

```
from p5 import *

num = 50
y = [0]*num
x = [0]*num
index_position = 0

def setup():
    size(100, 100)

    no_stroke()
    fill(255, 102)

def draw():
    background(0)
    global index_position
    y[index_position] = mouse_y
    x[index_position] = mouse_x

    # Cycle between 0 and the number of elements
    index_position = (index_position + 1) % num

    for i in range(num):
        # Set the array position to read
        pos = (index_position + i) % num
        radius = (num - i) / 2.0
        ellipse((x[pos], y[pos]), radius, radius)

if __name__ == '__main__':
    run()
```

2.7.4 Array Functions

Python provides a group of functions that assist in managing array data.

The `append()` function expands an array by one element, adds data to the new position, and returns the new array:

```
trees = ["ash", "oak"]
trees.append("maple") # adds "maple" to the end
print(trees) # prints ["ash", "oak", "maple"]
```

The `pop()` function decreases an array by one element by removing the last element and returns the last element of the array:

```
trees = ["ash", "oak"]
trees.pop() # removes "oak"
print(trees) # prints ["ash"]
```

The `extend()` function increases the size of an array. It can expand to a specific size, or if no size is specified, the array's length will be doubled. If an array needs to have many additional elements, it's faster to use expand() to double the size than to use append() to continually add one value at a time. The following example saves a new mouseX value to an array every frame. When the array becomes full, the size of the array is doubled and new mouseX values proceed to fill the enlarged array.

```
from p5 import *

x = [0]*100 # Array to store x-coordinates
count = 0 # Positions stored in array

def setup():
    size(100, 100)

def draw():
    global x, count
    x[count] = mouse_x # Assign new x-coordinate to the array
    count += 1 # Increment the counter

    if count == len(x):
        x.extend(x)
        print(len(x))


if __name__ == '__main__':
    run()
```

New functions can be written to perform operations on arrays, but arrays behave differently than data types such as intgers and characters. As with objects, when an array is used as a parameter to a function, the address (location in memory) of the array is transferred into the function instead of the actual data. No new array is created, and changes made within the function affect the array used as the parameter.

In the following example, the data[] array is used as the parameter to halve(). The address of data[] is passed to the d[] array in the halve() function. Because the address of d[] and data[] is the same, they both point to the same data. Changes made to d[] on line 14 modify the value of data[] in the setup() block. The draw() function is not used because the calculation is made only once and nothing is drawn to the display window.

```
from p5 import *

data = [19.0, 40.0, 75.0, 76.0, 90.0]

def setup():
    halve(data)
    print(data[0]) # Prints "9.5"
    print(data[1]) # Prints "20.0"
    print(data[2]) # Prints "27.5"
    print(data[3]) # Prints "38.0"
    print(data[4]) # Prints "45.0"

def halve(d):
    for i in range(len(d)): # For each array element,
        d[i] = d[i] / 2.0; # divide the value by 2

if __name__ == '__main__':
    run()
```

Changing array data within a function without modifying the original array requires some additional lines of code. In the following example, the array is passed into the function as a parameter, a new array is made, the values from the original array are copied in the new array, changes are made to the new array, and finally the modified array is returned.

```
from p5 import *

data = [19.0, 40.0, 75.0, 76.0, 90.0]
half_data = []

def setup():
    halfData = halve(data) # Run the halve() function
    print(halfData[0]) # Prints "9.5"
    print(halfData[1]) # Prints "20.0"
    print(halfData[2]) # Prints "27.5"
    print(halfData[3]) # Prints "38.0"
    print(halfData[4]) # Prints "45.0"

def halve(d):
    numbers = d.copy()
    for i in range(len(numbers)): # For each array element,
        numbers[i] = numbers[i] / 2.0; # divide the value by 2

    return numbers
```

(continues on next page)

(continued from previous page)

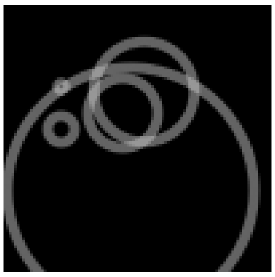
```python
if __name__ == '__main__':
    run()
```

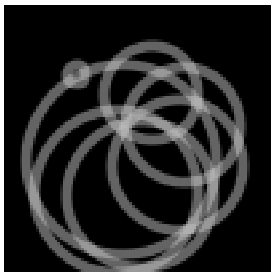
2.7.5 Array of Objects

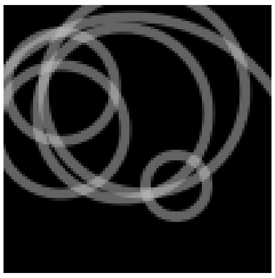
Working with arrays of objects is technically similar to working with arrays of other data types, but it opens the amazing possibility to create as many instances of a custom-designed class as desired. Like all arrays, an array of objects is distinguished from a single object with brackets, the [and] characters. However, because each array element is an object, each must be created with the keyword new before it can be used. The steps for working with an array of objects are:

1. Create the array
2. Create each object in the array

These steps are translated into code in the following example. It uses the Ring class from page 371, so copy it over or retype it. This code creates a rings[] array to hold fifty Ring objects. The first time a mouse button is pressed, the first Ring object is turned on and its x and y variables are assigned to the current values of the cursor. Each time a mouse button is pressed, a new Ring is turned on and displayed in the subsequent trip through draw(). When the final element in the array has been created, the program jumps back to the beginning of the array to assign new positions to earlier Rings.







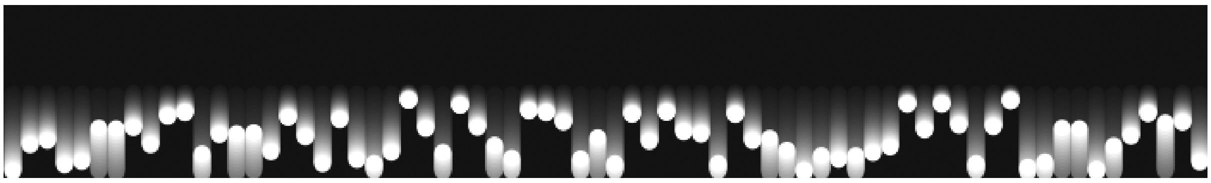
```
from p5 import *

rings = [] # Create the array
numRings = 50
currentRing = 0

def setup():
    size(100, 100)

    for i in range(numRings):
        rings.append(Ring())

def draw():
    background(0)

    for r in rings:
        r.grow()
        r.display()

def mouse_pressed():
    global currentRing
    rings[currentRing].start(mouse_x, mouse_y)

    currentRing += 1
    if currentRing > numRings:
        currentRing = 0

class Ring:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.diameter = 0
        self.on = False

    def start(self, xpos, ypos):
        self.x = xpos
        self.y = ypos

        self.diameter = 1
        self.on = True

    def grow(self):
        if self.on:
            self.diameter += 0.5
            if self.diameter > 400:
                self.on = False
                self.diameter = 1

    def display(self):
        if self.on:
```

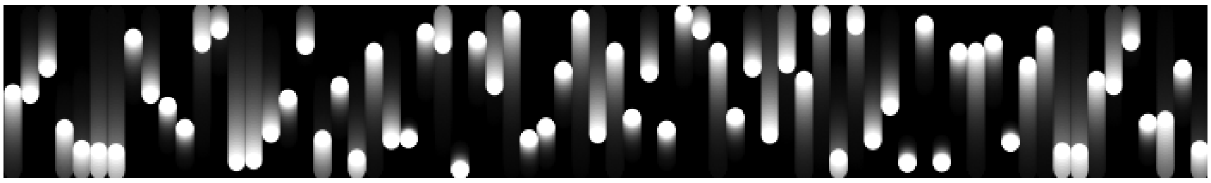
(continues on next page)

(continued from previous page)

```
            no_fill()
            stroke_weight(4)
            stroke(204, 153)
            ellipse((self.x, self.y), self.diameter, self.diameter)

if __name__ == '__main__':
    run()
```

The next example requires the Spot class from page 363. Unlike the prior example, variable values are generated within the setup() and are passed into each array elements through the object's constructor. Each element in the array starts with a unique set of x-coordinate, diameter, and speed values. Because the number of objects is dependent on the width of the display window, it is not possible to create the array until the program knows how wide it will be. Therefore, the array is declared outside of setup() to make it global (see p. 12), but it is created inside setup, after the width of the display window is defined.





```
from p5 import *

spots = [] # Create the array


def setup():
    size(700, 100)

    numSpots = 70 # Number of objects
    dia = width/numSpots # Calculate diameter

    for i in range(numSpots): # Create array
        x = dia/2 + i*dia
        rate = random_uniform(0.1, 0.2)

        spots.append(Spot(x, 50, dia, rate))

def draw():
    background(0, 12)
    fill(255)
```

(continues on next page)

(continued from previous page)

```
    for s in spots:
        s.move() # Move each object
        s.display() # Display each object

class Spot:
    def __init__(self, xpos, ypos, dia, sp):
        self.x = xpos
        self.y = ypos
        self.diameter = dia
        self.speed = sp
        self.direction = 1

    def move(self):
        self.y += self.speed*self.direction

        if self.y > height - self.diameter/2 or self.y < self.
↪diameter/2:
            self.direction *= -1

    def display(self):
        ellipse((self.x, self.y), self.diameter, self.diameter)

if __name__ == '__main__':
    run()
```

Each object in the array is in turn assigned to the variable s, so the first time through the loop, the code s.move() runs the move() method for the first element in the array, then the next time through the loop, s.move() runs the move() method for the second element in the array, etc. The two statements inside the block run for each element of the array until the end of the array. This way of accessing each element in an array of objects is used for the remainder of the book.

2.7.6 Two-dimensional Arrays

Data can also be stored and retrieved from arrays with more than one dimension. Using the example from the beginning of this chapter, the data points for the chart are put into a 2D array, where the second dimension adds a gray value:

A 2D array is essentially a list of 1D arrays. It must first be declared, then created, and then the values can be assigned just as in a 1D array. The following syntax converts the diagram above into to code:

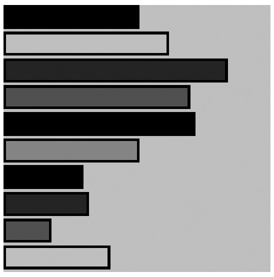
```
x = [[50, 0], [61,204], [83,51], [69,102], [71, 0],
[50,153], [29, 0], [31,51], [17,102], [39,204]]

print(x[0][0]) # Prints "50"
print(x[0][1]) # Prints "0"
```

(continues on next page)

(continued from previous page)

```
print(x[4][2]) # ERROR! This element is outside the array
print(x[3][0]) # Prints "69"
print(x[9][1]) # Prints "204"
```

It is possible to continue and make 3D and 4D arrays by extrapolating these techniques. However, multidimensional arrays can be confusing, and often it is a better idea to maintain multiple 1D or 2D arrays.



```
from p5 import *

x = [[50, 0], [61,204], [83,51], [69,102], [71,
↪ [50,153], [29, 0], [31,51], [17,102], [39,2


def setup():
    size(100, 100)

def draw():
    for i in range(len(x)):
        fill(x[i][1])
        rect((0, i*10), x[i][0], 8)

if __name__ == '__main__':
    run()
```

2.8 Images and Pixels

Authors Daniel Shiffman Arihant Parsoya (p5 port)

Copyright This tutorial is from the book Learning Processing by Daniel Shiffman, published by Morgan Kaufmann, © 2008 Elsevier Inc. All rights reserved. The tutorial was ported to p5 by Arihant Parsoya. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

A digital image is nothing more than data—numbers indicating variations of red, green, and blue at a particular location on a grid of pixels. Most of the time, we view these pixels as miniature rectangles sandwiched together on a computer screen. With a little creative thinking and some lower level manipulation of pixels with code, however, we can display that information in a myriad of ways. This tutorial is dedicated to breaking out of simple shape drawing in Processing and using images (and their pixels) as the building blocks of Processing graphics.

2.8.1 Getting started with images

Hopefully, you are comfortable with the idea of data types. You probably specify them often—a float variable "speed", an int "x", etc. These are all primitive data types, bits sitting in the computer's memory ready for our use. Though perhaps a bit trickier, you hopefully also use objects, complex data types that store multiple pieces of data (along with functionality)—a "Ball" class, for example, might include floating point variables for location, size, and speed as well as methods to move, display itself, and so on.

In addition to user-defined objects (such as Ball), Processing has a bunch of handy classes all ready to go without us writing any code. In this tutorial, we'll examine PImage, a class for loading and displaying an image as well as looking at its pixels.

```
from p5 import *

img = None # Declare a variable of type PImage

def setup():
        size(320,240)
        # Make a new instance of a PImage by loading an image file
        img = loadImage("mysummervacation.jpg")

def draw():
        background(0)
        # Draw the image to the screen at coordinate (0,0)
        image(img, (0, 0))

if __name__ == '__main__':
        run()
```

Using an instance of a PImage object is no different than using a user-defined class. First, a variable of type PImage, named "img," is declared. Second, a new instance of a PImage object is created via the `load_image()` method. `load_image()` takes one argument, a String indicating a file name, and loads the that file into memory. `load_image()` looks for image files stored in your current terminal directory.

In the above example, it may seem a bit peculiar that we never called a "constructor" to instantiate the PImage object, saying "PImage()". After all, in most object-related examples, a constructor is a must for producing an object instance.

```
ss = Spaceship()
flr = Flower(25)
```

yet:

```
img = load_image("file.jpg")
```

In fact, the `load_image()` function performs the work of a constructor, returning a brand new instance of a PImage object generated from the specified filename. We can think of it as the PImage constructor for loading images from a file. For creating a blank image, the `create_image()` function is used.

We should also note that the process of loading the image from the hard drive into memory is a slow one, and we should make sure our program only has to do it once, in `setup()`. Loading images in `draw()` may result in slow performance as well as "Out of Memory" errors.

Once the image is loaded, it is displayed with the `image()` function. The `image()` function must include 3 arguments—the image to be displayed, the x location, and the y location. Optionally two arguments can be added to resize the image to a certain width and height.

```
image(img, (10,20), (90,60))
```

2.8.2 Your very first image processing filter

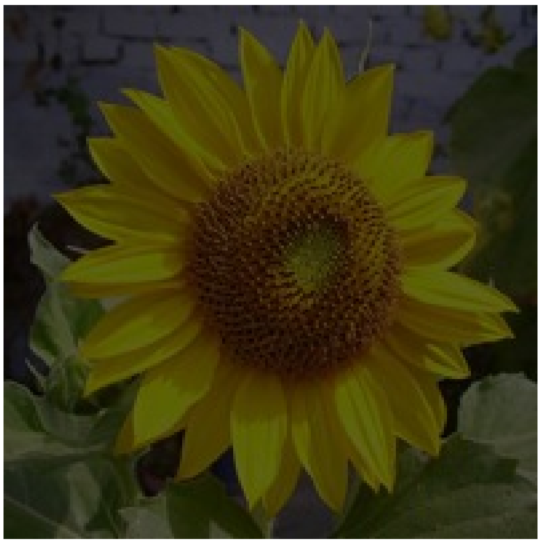
When displaying an image, you might like to alter its appearance. Perhaps you would like the image to appear darker, transparent, blue-ish, etc. This type of simple image filtering is achieved with Processing's `tint()` function. `tint()` is essentially the image equivalent of shape's `fill()`, setting the color and alpha transparency for displaying an image on screen. An image, nevertheless, is not usually all one color. The arguments for `tint()` simply specify how much of a given color to use for every pixel of that image, as well as how transparent those pixels should appear.

For the following examples, we will assume that two images (a sunflower and a dog) have been loaded and the dog is displayed as the background (which will allow us demonstrate transparency.)

```
sunflower = load_image("sunflower.jpg")
dog = loadImage("dog.jpg")
background(dog)
```

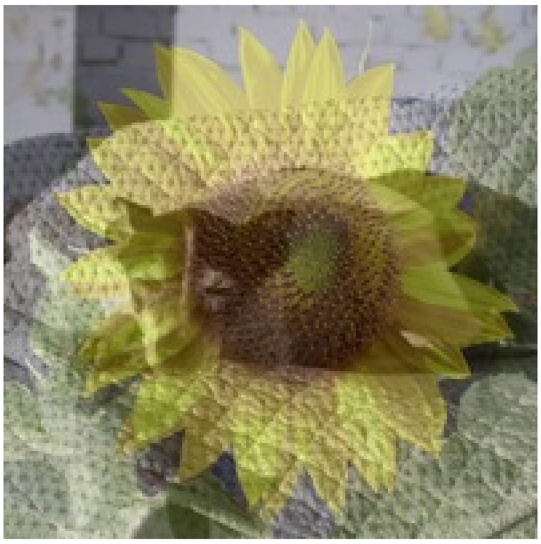
If `tint()` receives one argument, only the brightness of the image is affected.

```
# The image retains its original state.
tint(255)
image(sunflower, (0, 0))
```



```
# The image appears darker
tint(100)
image(sunflower, (0, 0))
```

A second argument will change the image's alpha transparency.



```
# The image is at 50% opacity.
tint(100, 127)
image(sunflower, (0, 0))
```

Three arguments affect the brightness of the red, green, and blue components of each color.

```
# None of its red, most of its green, and all of its blue.
tint(0, 200, 255)
image(sunflower, (0, 0))
```

Finally, adding a fourth argument to the method manipulates the alpha (same as with 2). Incidentally, the range of values for tint() can be specified with colorMode().

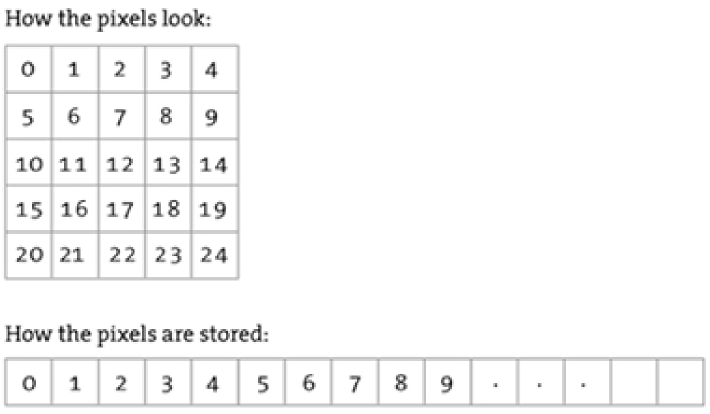
```
# The image is tinted red and transparent.
tint(255, 0, 0, 100)
image(sunflower, (0, 0))
```

2.8.3 Pixels, pixels, and more pixels

If you've just begun using Processing you may have mistakenly thought that the only offered means for drawing to the screen is through a function call. "Draw a line between these points" or "Fill an ellipse with red" or "load this JPG image and place it on the screen here." But somewhere, somehow, someone had to write code that translates these function calls into setting the individual pixels on the screen to reflect the requested shape. A line doesn't appear because we say line(), it appears because we color all the pixels along a linear path between two points. Fortunately, we don't have to manage this lower-level-pixel-setting on a day-to-day basis. We have the developers of Processing (and Java) to thank for the many drawing functions that take care of this business.

Nevertheless, from time to time, we do want to break out of our mundane shape drawing existence and deal with the pixels on the screen directly. Processing provides this functionality via the pixels array.

We are familiar with the idea of each pixel on the screen having an X and Y position in a two dimensional window. However, the array pixels has only one dimension, storing color values in linear sequence.



Take the following simple example. This program sets each pixel in a window to a random grayscale value. The pixels array is just like an other array, the only difference is that we don't have to declare it since it is a Processing built-in variable.

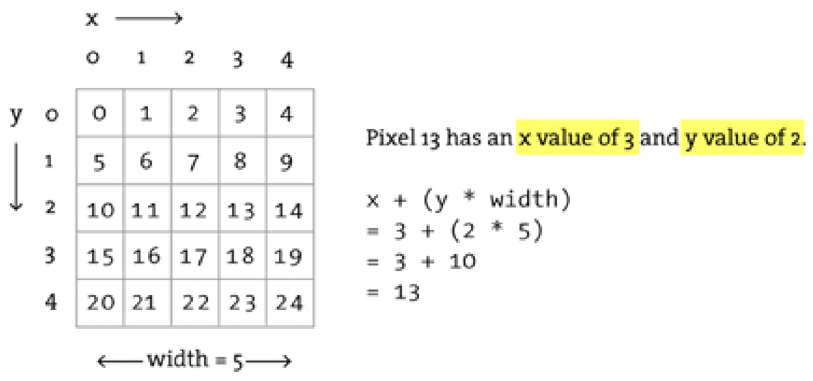
```
size(200, 200)
# Before we deal with pixels
load_pixels()
# Loop through every pixel
for i in range(pixels.length):
        # Pick a random number, 0 to 255
        rand = random(255)
        # Create a grayscale color based on random number
        c = color(rand)
        # Set pixel at that location to random color
        pixels[i] = c

# When we are finished dealing with pixels
```

First, we should point out something important in the above example. Whenever you are accessing the pixels of a Processing window, you must alert Processing to this activity. This is accomplished with two functions:

- `load_pixels()`: This function is called before you access the pixel array, saying "load the pixels, I would like to speak with them!"

In the above example, because the colors are set randomly, we didn't have to worry about where the pixels are onscreen as we access them, since we are simply setting all the pixels with no regard to their relative location. However, in many image processing applications, the XY location of the pixels themselves is crucial information. A simple example of this might be, set every even column of pixels to white and every odd to black. How could you do this with a one dimensional pixel array? How do you know what column or row any given pixel is in? In programming with pixels, we need to be able to think of every pixel as living in a two dimensional world, but continue to access the data in one (since that is how it is made available to us). We can do this via the following formula:

1. Assume a window or image with a given WIDTH and HEIGHT.
2. We then know the pixel array has a total number of elements equaling WIDTH * HEIGHT.
3. For any given X, Y point in the window, the location in our 1 dimensional pixel array is: LOCATION = X + Y*WIDTH



This may remind you of our two dimensional arrays tutorial. In fact, we'll need to use the same nested for loop technique. The difference is that, although we want to use for loops to think about the pixels in two dimensions, when we go to actually access the pixels, they live in a one dimensional array, and we have to apply the formula from the above illustration.

Let's look at how it is done.

```
size(200, 200)
load_pixels()
# Loop through every pixel column
for x in range(width):
        # Loop through every pixel row
        for y in range(height):
        # Use the formula to find the 1D location
        loc = x + y * width
        if (x % 2 == 0): # If we are an even column
                pixels[loc] = color(255)
        else: # If we are an odd column
                pixels[loc] = color(0)
```

2.8.4 Intro To Image Processing

The previous section looked at examples that set pixel values according to an arbitrary calculation. We will now look at how we might set pixels according those found in an existing PImage object. Here is some pseudo-code.

1. Load the image file into a PImage object
2. For each pixel in the PImage, retrieve the pixel's color and set the display pixel to that color.

The PImage class includes some useful fields that store data related to the image—width, height, and pixels. Just as with our user-defined classes, we can access these fields via the dot syntax.

```
img = createImage(320,240,RGB) # Make a PImage object
print(img.width) # Yields 320
print(img.height) # Yields 240
img.pixels[0] = color(255,0,0) # Sets the first pixel of the image␣
↪to red
```

Access to these fields allows us to loop through all the pixels of an image and display them onscreen.

```
from p5 import *

img = None

def setup():
        global img
        size(320,240)
        img = load_image("sunflower.jpg")

def draw():
        global img

        with load_pixels():
                # Since we are going to access the image's pixels␣
↪too
                img.load_pixels()
                for y in range(img.height):
                        for x in range(img.width):
                                loc = x + y*width

                                # The functions red(), green(), and␣
↪blue() pull out the 3 color components from a pixel.
                                r = img._get_pixel((x, y)).red
                                g = img._get_pixel((x, y)).green
                                b = img._get_pixel((x, y)).blue

                                # Image Processing would go here
```

(continues on next page)

(continued from previous page)

```
                                            # If we were to change the RGB␣
↪values, we would do it here,
                                            # before setting the pixel in the␣
↪display window.

                                            # Set the display pixel to the␣
↪image pixel
                                            pixels._set_pixel((x, y), Color(r,g,
↪b))

if __name__ == '__main__':
        run()
```

2.9 Curves

Authors J David Eisenberg; Arihant Parsoya (p5 port)

Copyright If you see any errors or have comments, please let us know. The tutorial was ported to p5 by Arihant Parsoya. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

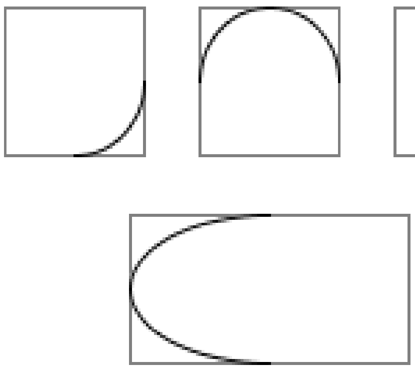
This short tutorial introduces you to the three types of curves in Processing: arcs, spline curves, and Bézier curves.

2.9.1 Arcs

Arcs are the simplest curves to draw. Processing defines an arc as a section of an ellipse. You call the function with these parameters:

```
arc((x, y), width, height, start, stop)
```

The first four parameters are the same as the ones for `ellipse()` they define the boundary box for your arc. The last two parameters are the starting and ending angle for the arc. These angles, as with all other angles in Processing, are given in radians. Remember that angles are measured clockwise, with zero degrees pointing east. Using the fact that PI radians equals 180°, here are some example arcs.



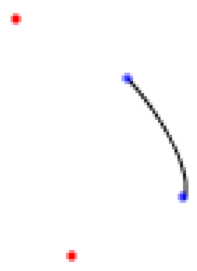
```
from p5 import *

def setup():
        size(300, 200)
        background(255)

        rect_mode('CENTER')
        stroke(128)
        rect((35, 35), 50, 50)
        rect((105, 35), 50, 50)
        rect((175, 35), 50, 50)
        rect((105, 105), 100, 50)

        stroke(0)
        arc((35, 35), 50, 50, 0, PI / 2.0) # lower quarter circle
        arc((105, 35), 50, 50, -PI, 0)  # upper half of circle
        arc((175, 35), 50, 50, -PI / 6, PI / 6) # 60 degrees
        arc((105, 105), 100, 50, PI / 2, 3 * PI / 2) # 180 degrees


if __name__ == '__main__':
        run()
```

2.9.2 Spline Curves

Arcs are fine, but they're plain. The next function, `curve()`, lets you draw curves that aren't necessarily part of an arc. This function draws what is technically called a Rom-Catmull Spline. To draw the curve, you have to specify the (x, y) coordinates of the points where the curve starts and ends. You must also specify two control points which determine the direction and amount of curvature. A call to `curve()` uses these parameters:

```
curve((cpx1, cpy1), (x1, y1), (x2, y2), (cpx2, cpy2))
```

Here is an example that shows a `curve()`. The control points are shown in red and the curve points in blue.



```
from p5 import *

def setup():
        size(200, 200)
        no_loop()

def draw():
        background(255)
        stroke(0)
        curve((40, 40), (80, 60), (100, 100), (60, 120))
        fill(255, 0, 0)
        ellipse((40, 40), 3, 3)
        fill(0, 0, 255, 192)
        ellipse((100, 100), 3, 3)
        ellipse((80, 60), 3, 3)
        fill(255, 0, 0)
        ellipse((60, 120), 3, 3)

if __name__ == '__main__':
        run()
```

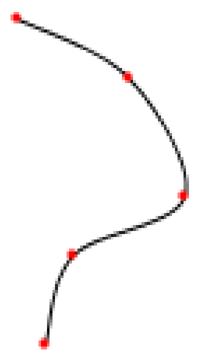
How do the control points affect the way the curve looks? Take a deep breath, because this is somewhat complicated.

- The tangent to the curve at the start point is parallel to the line between control point one and the end of the curve. These are the lines shown in green in the diagram at the left.
- The tangent to the curve at the end point is parallel to the line between the start point and control point 2. These are the lines shown in purple in the diagram at the left.

2.9.3 Continuous Spline Curves

In isolation, a single `curve()` is not particularly appealing. To draw a continuous curve through several points, you are better off using the `curve_vertex()` function. You can only use this function when you are creating a shape with the `begin_shape()` and `end_shape()` functions.

Here is a curve connecting the points (40, 40), (80, 60), (100, 100), (60, 120), and (50, 150). In common usage, people use the first point of the curve as the first control point and the last point of the curve as the last control point.

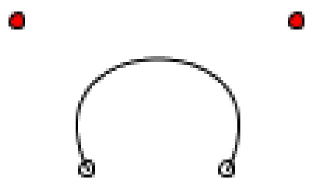


```
from p5 import *

coords = [40, 40, 80, 60, 100, 100, 60, 120, 50, 150]

def setup():
        size(200, 200)

def draw():
        no_fill()
        background(255)
        stroke(0)
        begin_shape()
        curve_vertex(40, 40) # the first control point
        curve_vertex(40, 40) # is also the start point of curve
        curve_vertex(80, 60)
        curve_vertex(100, 100)
        curve_vertex(60, 120)
        curve_vertex(50, 150) # the last point of curve
        curve_vertex(50, 150) # is also the last control point
        end_shape()

        # Use the array to keep the code shorter;
        # you already know how to draw ellipses!
        fill(255, 0, 0)
        no_stroke()

        for i in range(0, len(coords), 2):
                ellipse((coords[i], coords[i + 1]), 3, 3)


if __name__ == '__main__':
        run()
```

2.9.4 Bézier Curves

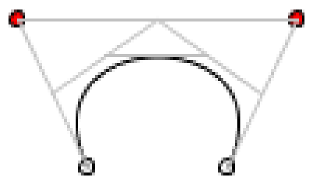
Though better than arcs, spline curves don't seem to have those graceful, swooping curves that say "art." For those, you need to draw Bézier curves with the `bezier()` function. As with spline curves, the `bezier()` function has eight parameters, but the order is different:

Here is a program that displays a Bézier curve and its control points.



```
from p5 import *

coords = [40, 40, 80, 60, 100, 100, 60, 120, 50, 150]

def setup():
        size(150, 150)

def draw():
        background(255)
        ellipse((50, 75), 5, 5) # endpoints of curve
        ellipse((100, 75), 5, 5)
        fill(255, 0, 0)
        ellipse((25, 25), 5, 5)  # control points
        ellipse((125, 25), 5, 5)
        no_fill()
        stroke(0)
        bezier((50, 75), (25, 25), (125, 25), (100, 75))

if __name__ == '__main__':
        run()
```

While it is difficult to visualize how the control points affect a `curve()`, it is slightly easier to see how the control points affect Bézier curves. Imagine two poles and several rubber bands. The poles connect the control points to the endpoints of the curve. A rubber band connects the tops of the poles. Two more rubber bands connect the midpoints of the poles to the midpoint of the first rubber band. One more rubber band connects their midpoints. The center of that last rubber band is tied to the curve. This diagrams helps to explain:



2.9.5 Continuous Bézier Curves

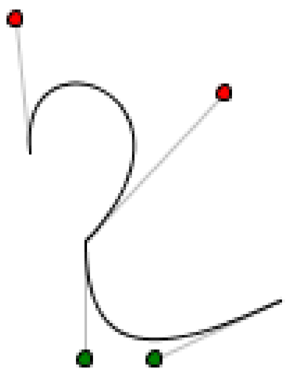
Just as `curve_vertex()` allows you to make continuous spline curves, `bezier_vertex()` lets you make continuous Bézier curves. Again, you must be within a `begin_shape()` / `end_shape()` sequence. You must use vertex(startX, startY) to specify the starting point of the curve. Subsequent points are specified with a call to:

```
bezierVertex((cpx1, cpy1), (cpx2, cpy2), (x, y));
cpx1, cpy1      Coordinates of the first control point
cpx2, cpy2      Coordinates of the second control point
x, y            The next point on the curve
```

So, to draw the previous example using bezier_vertex(), you would do this:

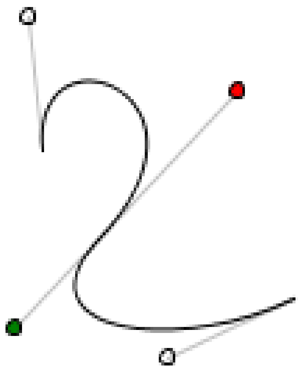
```
from p5 import *

coords = [40, 40, 80, 60, 100, 100, 60, 120, 50, 150]

def setup():
        size(150, 150)

def draw():
        background(255)
        # Don't show where control points are
        no_fill()
        stroke(0)
        begin_shape()
        vertex(50, 75) # first point
        bezier_vertex(25, 25, 125, 25, 100, 75)
        end_shape()

if __name__ == '__main__':
        run()
```

Here is a continuous Bézier curve, but it doesn't join smoothly. The diagram shows the control points, but only the relevant code for drawing the curve is here.



```
from p5 import *

coords = [40, 40, 80, 60, 100, 100, 60, 120, 50, 150]

def setup():
        size(200, 200)

def draw():
        background(255)
        no_fill()
        begin_shape()
        vertex(30, 70) # first point
        bezier_vertex(25, 25, 100, 50, 50, 100)
        bezier_vertex(50, 140, 75, 140, 120, 120)
        end_shape()

if __name__ == '__main__':
        run()
```

In order to make two curves A and B smoothly continuous, the last control point of A, the last point of A, and the first control point of B have to be on a straight line. Here is an example that meets those conditions. The points that are in a line are shown in bold.



```
from p5 import *

coords = [40, 40, 80, 60, 100, 100, 60, 120, 50, 150]
```

(continues on next page)

(continued from previous page)

```
def setup():
        size(200, 200)

def draw():
        background(255)
        no_fill()
        begin_shape()
        vertex(30, 70) # first point
        bezier_vertex(25, 25, 100, 50, 50, 100)
        bezier_vertex(20, 130, 75, 140, 120, 120)
        end_shape()

if __name__ == '__main__':
        run()
```

2.9.6 Summary

- Use `arc()` when you need a segment of a circle or an ellipse. You can't make continuous arcs or use them as part of a shape.
- Use `curve()` when you need a small curve between two points. Use curveVertex() to make a continuous series of curves as part of a shape.
- Use `bezier()` when you need long, smooth curves. Use `bezier_vertex()` to make a continuous series of Bézier curves as part of a shape.

2.10 2D Transformations

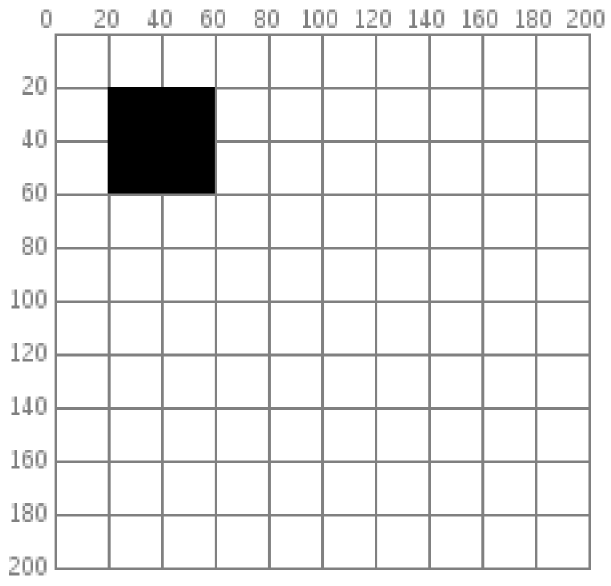
Authors J David Eisenberg; Arihant Parsoya (p5 port)

Copyright If you see any errors or have comments, please let us know. The tutorial was ported to p5 by Arihant Parsoya. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

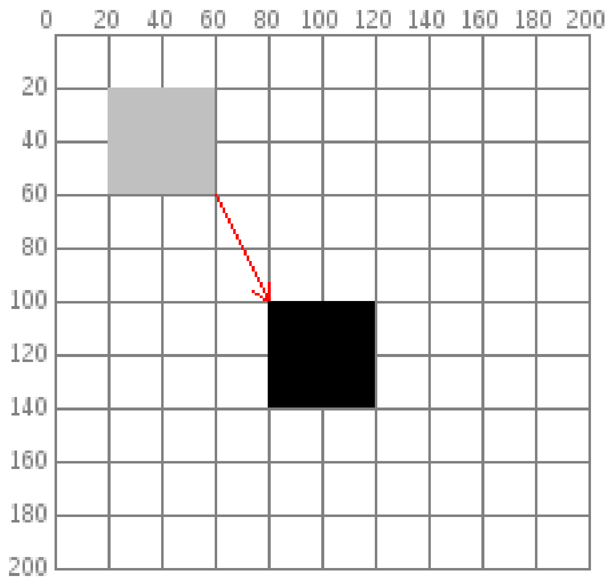
Processing has built-in functions that make it easy for you to have objects in a sketch move, spin, and grow or shrink. This tutorial will introduce you to the `translate`, `rotate`, and `scale` functions so that you can use them in your sketches.

2.10.1 Translation: Moving the Grid

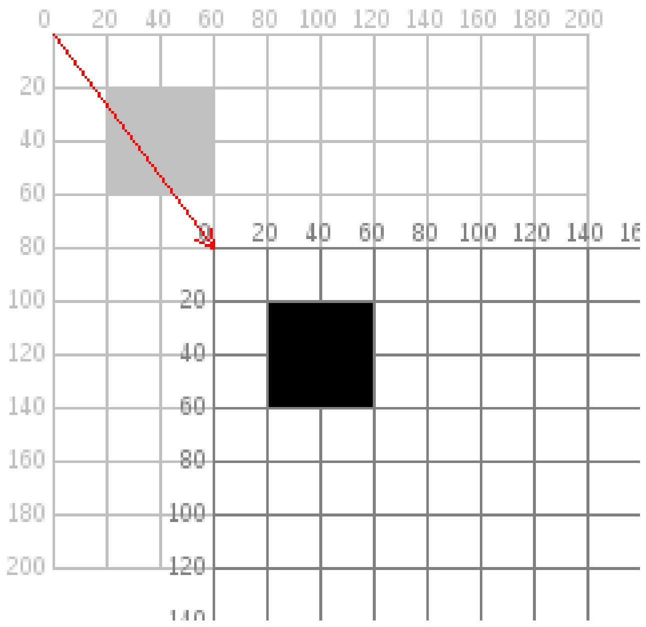
As you know, your Processing window works like a piece of graph paper. When you want to draw something, you specify its coordinates on the graph. Here is a simple rectangle drawn with the code `rect((20, 20), 40, 40)`. The coordinate system (a fancy word for "graph paper") is shown in gray.



If you want to move the rectangle 60 units right and 80 units down, you can just change the coordinates by adding to the x and y starting point: rect((20 + 60, 20 + 80), 40, 40) and the rectangle will appear in a different place. (We put the arrow in there for dramatic effect.)



But there is a more interesting way to do it: move the graph paper instead. If you move the graph paper 60 units right and 80 units down, you will get exactly the same visual result. Moving the coordinate system is called translation.



The important thing to notice in the preceding diagram is that, as far as the rectangle is concerned, it hasn't moved at all. Its upper left corner is still at (20,20). When you use transformations, the things you draw never change position; the coordinate system does.

Here is code that draws the rectangle in red by changing its coordinates, then draws it in blue by moving the grid. The rectangles are translucent so that you can see that they are (visually) at the same place. Only the method used to move them has changed. Copy and paste this code into Processing and give it a try.

```
from p5 import *

def setup():
        size(200, 200)

def draw():
        background(255)
        no_stroke()

        # draw the original position in gray
        fill(192)
        rect((20, 20), 40, 40)

        # draw a translucent red rectangle by changing the␣
↪coordinates
        fill(255, 0, 0, 128)
        rect((20 + 60, 20 + 80), 40, 40)

        # draw a translucent blue rectangle by translating the grid
        fill(0, 0, 255, 128)
        with push_matrix():
            translate(60, 80)
            rect((20, 20), 40, 40)
```

(continues on next page)

(continued from previous page)

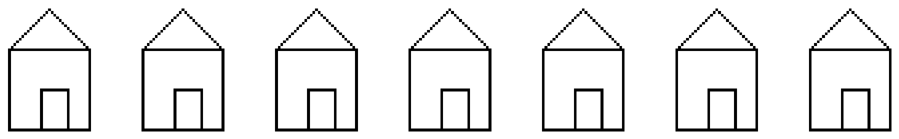
```
if __name__ == '__main__':
  run()
```

Let's look at the translation code in more detail. `push_matrix()` is a built-in function that saves the current position of the coordinate system. The `translate(60, 80)` moves the coordinate system 60 units right and 80 units down. The `rect((20, 20), 40, 40)` draws the rectangle at the same place it was originally. Remember, the things you draw don't move—the grid moves instead. Finally, when the `with` context ends, it "pops" and restores the coordinate system to the way it was before you did the translate.

Yes, you could have done a `translate(-60, -80)` to move the grid back to its original position. However, when you start doing more sophisticated operations with the coordinate system, it's easier to use `with push_matrix():` to save and then later restore the status rather than having to undo all your operations. Later on in this tutorial, you will find out why those functions seem to have such strange names.

2.10.2 What's the Advantage?

You may be thinking that picking up the coordinate system and moving it is a lot more trouble than just adding to coordinates. For a simple example like the rectangle, you are correct. But let's take an example of where `translate()` can make life easier. Here is some code that draws a row of houses. It uses a loop that calls a function named `house()`, which takes the x and y location of the house's upper-left corner as its parameters.



```
def setup():
        size(400, 100)


def draw():
        background(255)
        for i in range(10, 350, 50):
                house(i, 20)
```

This is the code for drawing the house by changing its position. Look at all the additions that you have to keep track of.

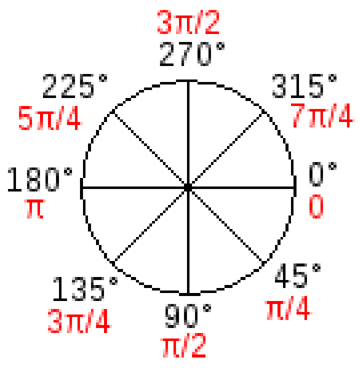
```
def house(x, y):
        triangle((x + 15, y), (x, y + 15), (x + 30, y + 15))
        rect((x, y + 15), 30, 30)
        rect((x + 12, y + 30), 10, 15)
```

Compare that to the version of the function that uses translate(). In this case, the code draws the house in the same place every time, with its upper left corner at (0, 0), and lets translation do all the work instead.

```
def house(x, y):
        with push_matrix():
                translate(x, y)
                triangle((15, 0), (0, 15), (30, 15))
                rect((0, 15), 30, 30)
                rect((12, 30), 10, 15)
```

2.10.3 Rotation

In addition to moving the grid, you can also rotate it with the `rotate()` function. This function takes one argument, which is the number of radians that you want to rotate. In Processing, all the functions that have to do with rotation measure angles in radians rather than degrees. When you talk about angles in degrees, you say that a full circle has 360°. When you talk about angles in radians, you say that a full circle has 2π radians. Here is a diagram of how Processing measures angles in degrees (black) and radians (red).



Since most people think in degrees, Processing has a built-in `radians()` function which takes a number of degrees as its argument and converts it for you. It also has a `degrees()` function that converts radians to degrees. Given that background, let's try rotating a square clockwise 45 degrees.

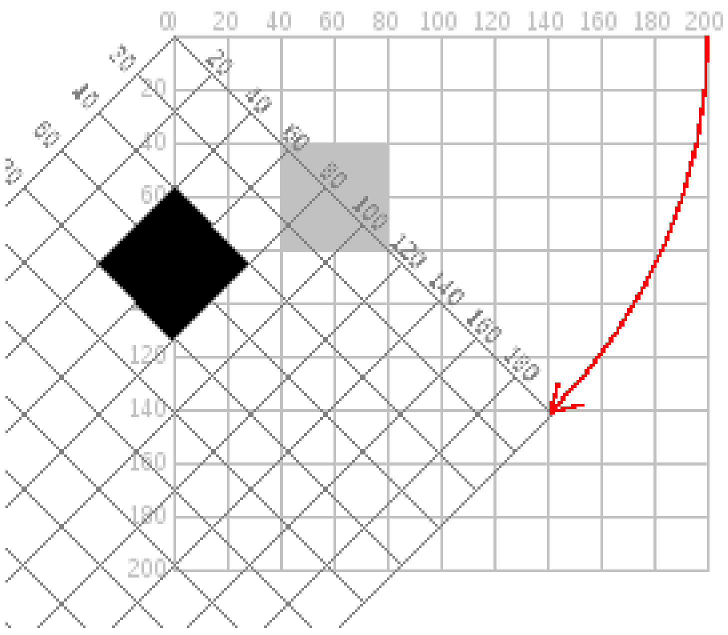
```
from p5 import *
```

(continues on next page)

(continued from previous page)

```
def setup():
        size(200, 200)

def draw():
        background(255)

        fill(192)
        no_stroke()
        rect((40, 40), 40, 40)

        with push_matrix():
                rotate(radians(45))
                fill(0)
                rect((40, 40), 40, 40)

if __name__ == '__main__':
  run()
```

Hey, what happened? How come the square got moved and cut off? The answer is: the square did not move. The grid was rotated. Here is what really happened. As you can see, on the rotated coordinate system, the square still has its upper left corner at (40, 40).

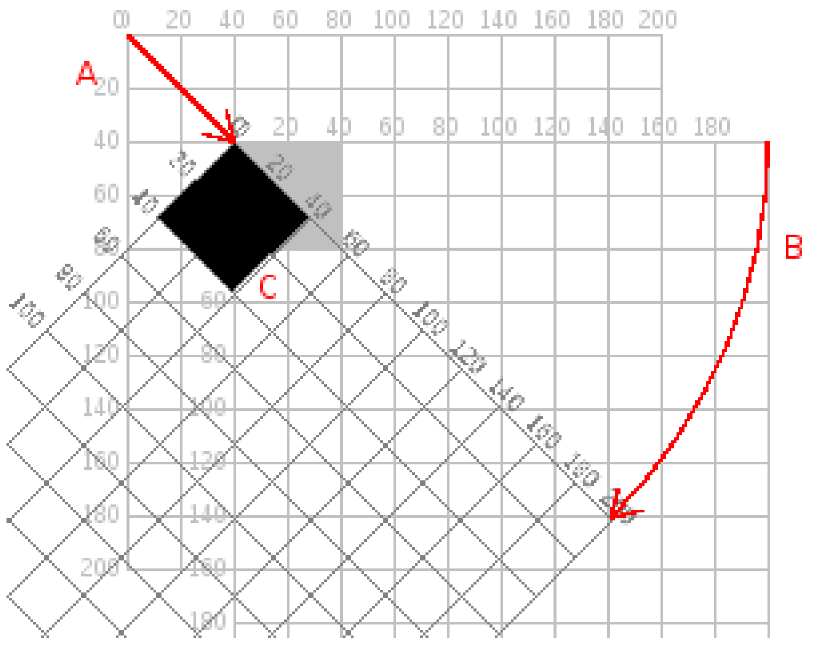


2.10.4 Rotating the Correct Way

The correct way to rotate the square is to:

1. Translate the coordinate system's origin (0, 0) to where you want the upper left of the square to be.
2. Rotate the grid $\pi/4$ radians (45°)

3. Draw the square at the origin.



And here is the code and its result, without the grid marks.

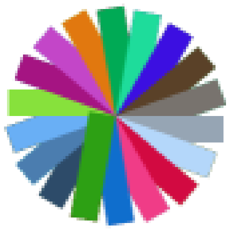
```
from p5 import *

def setup():
        size(200, 200)

def draw():
        background(255)

        fill(192)
        no_stroke()
        rect((40, 40), 40, 40)

        with push_matrix():
                # move the origin to the pivot point
                translate(40, 40)

                # then pivot the grid
                rotate(radians(45))
```

(continues on next page)

(continued from previous page)

```
                # and draw the square at the origin
                fill(0)
                rect((0, 0), 40, 40)

if __name__ == '__main__':
  run()
```

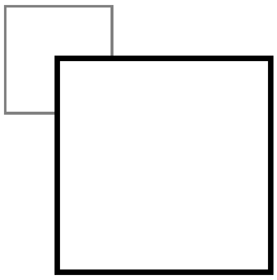
And here is a program that generates a wheel of colors by using rotation. The screenshot is reduced to save space.



```
from p5 import *

def setup():
        size(200, 200)
        no_stroke()
        background(255)

def draw():
        if (frame_count % 10 == 0):
                fill(frame_count * 3 % 255,
                        frame_count * 5 % 255,
                        frame_count * 7 % 255)

        with push_matrix():
                translate(100, 100)
                rotate(radians(frame_count * 2  % 360))
                rect((0, 0), 80, 20)

if __name__ == '__main__':
  run()
```

2.10.5 Scaling

The final coordinate system transformation is scaling, which changes the size of the grid. Take a look at this example, which draws a square, then scales the grid to twice its normal size, and draws it again.



```
from p5 import *

def setup():
        size(200, 200)
        stroke(128)

def draw():
        background(255)
        rect((20, 20), 40, 40)

        stroke(0)
        with push_matrix():
                scale(2.0)
                rect((20, 20), 40, 40)

if __name__ == '__main__':
  run()
```

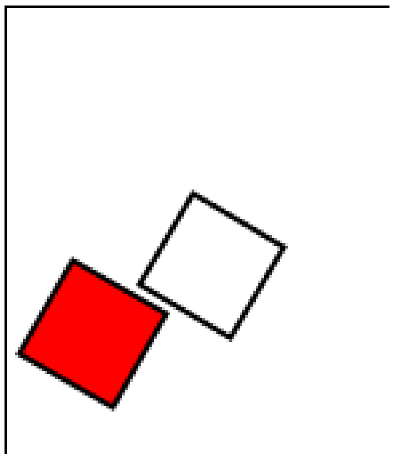
First, you can see that the square appears to have moved. It hasn't, of course. Its upper left corner is still at (20, 20) on the scaled-up grid, but that point is now twice as far away from the origin as it was in the original coordinate system. You can also see that the lines are thicker. That's no optical illusion—the lines really are twice as thick, because the coordinate system has been scaled to double its size.

Programming Challenge: Scale up the black square, but keep its upper left corner in the same place as the gray square. Hint: use translate() to move the origin, then use scale().

There is no law saying that you have to scale the x and y dimensions equally. Try using `scale(3.0, 0.5)` to make the x dimension three times its normal size and the y dimension only half its normal size.

2.10.6 Order Matters

When you do multiple transformations, the order makes a difference. A rotation followed by a translate followed by a scale will not give the same results as a translate followed by a rotate by a scale. Here is some sample code and the results.



```
from p5 import *

def setup():
        size(200, 200)
        stroke(128)

def draw():
        background(255)
        line((0, 0), (200, 0)) # draw axes
        line((0, 0), (0, 200))

        with push_matrix():
                fill(255, 0, 0) # red square
                rotate(radians(30))
                translate(70, 70)
                scale(2.0)
                rect((0, 0), 20, 20)

        with push_matrix():
                fill(255) # white square
                translate(70, 70)
                rotate(radians(30))
                scale(2.0)
                rect((0, 0), 20, 20)

if __name__ == '__main__':
  run()
```

2.10.7 The Transformation Matrix

Every time you do a rotation, translation, or scaling, the information required to do the transformation is accumulated into a table of numbers. This table, or matrix has only a few rows and columns, yet, through the miracle of mathematics, it contains all the information needed to do any series of transformations. And that's why `push_matrix()` has that word in their name.

2.10.8 Push (and Pop)

What about the push part of the name? It comes from a computer concept known as a stack, which works like a spring-loaded tray dispenser in a cafeteria. When someone returns a tray to the stack, its weight pushes the platform down. When someone needs a tray, he takes it from the top of the stack, and the remaining trays pop up a little bit.

In a similar manner, `push_matrix()` puts the current status of the coordinate system at the top of a memory area. When the *with* exits the context automatically "pops" and pulls that status back out. The preceding example used `push_matrix()` to make sure that the coordinate system was "clean" before each part of the drawing. In all of the other examples, the calls to those two functions weren't really necessary, but it doesn't hurt anything to save and restore the grid status.

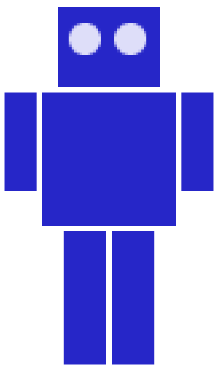
Note: in Processing, the coordinate system is restored to its original state (origin at the upper left of the window, no rotation, and no scaling) every time that the `draw()` function is executed.

2.10.9 Three-dimensional Transforms

If you are working in three dimensions, you can call the `translate()` function with three arguments for the x, y, and z distances. Similarly, you can call `scale()` with three arguments that tell how much you want the grid scaled in each of those dimensions.

For rotation, call the `rotateX()`, `rotateY()`, or `rotateZ()` function to rotate around each of the axes. All three of these functions expect one argument: the number of radians to rotate.

2.10.10 Case Study: An Arm-Waving Robot

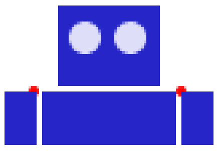


```
from p5 import *

def setup():
        size(200, 200)
        background(255)

def draw():
```

(continues on next page)

(continued from previous page)

```
        drawRobot()

def drawRobot():
        no_stroke()
        fill(38, 38, 200)
        rect((20, 0), 38, 30) # head
        rect((14, 32), 50, 50) # body

        rect((0, 32), 12, 37) # left arm
        rect((66, 32), 12, 37) # right arm

        rect((22, 84), 16, 50) # left leg
        rect((40, 84), 16, 50) # right leg

        fill(222, 222, 249)
        ellipse((30, 12), 12, 12) # left eye
        ellipse((47, 12), 12, 12) # right eye

if __name__ == '__main__':
  run()
```



The next step is to identify the points where the arms pivot. That is shown in this drawing. The pivot points are (12, 32) and (66, 32). Note: the term "center of rotation" is a more formal term for the pivot point.

Now, separate the code for drawing the left and right arms, and move the center of rotation for each arm to the origin, because you always rotate around the (0, 0) point. To save space, we are not repeating the code for `setup()`.

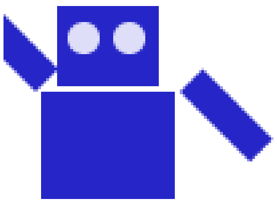
```
def drawRobot():
        no_stroke()
        fill(38, 38, 200)
        rect((20, 0), 38, 30) # head
        rect((14, 32), 50, 50) # body

        drawLeftArm()
        drawRightArm()

        rect((22, 84), 16, 50) # left leg
        rect((40, 84), 16, 50) # right leg

        fill(222, 222, 249)
        ellipse((30, 12), 12, 12) # left eye
        ellipse((47, 12), 12, 12) # right eye
```

(continues on next page)

(continued from previous page)

```
def drawLeftArm():
        with push_matrix():
                translate(12, 32)
                rect((-12, 0), 12, 37)

def drawRightArm():
        with push_matrix():
                translate(66, 32)
                rect((0, 0), 12, 37)
```

Now test to see if the arms rotate properly. Rather than attempt a full animation, we will just rotate the left side arm 135 degrees and the right side arm -45 degrees as a test. Here is the code that needs to be added, and the result. The left side arm is cut off because of the window boundaries, but we'll fix that in the final animation.



```
def drawLeftArm():
        with push_matrix():
                translate(12, 32)
                rotate(radians(135))
                rect((-12, 0), 12, 37)z

def drawRightArm():
        with push_matrix():
                translate(66, 32)
                rotate(radians(-45))
                rect((0, 0), 12, 37)
```

Now we complete the program by putting in the animation. The left arm has to rotate from 0° to 135° and back. Since the arm-waving is symmetric, the right-arm angle will always be the negative value of the left-arm angle. To make things simple, we will go in increments of 5 degrees.

```
from p5 import *

armAngle = 0
angleChange = 5
ANGLE_LIMIT = 135

def setup():
        size(200, 200)
        background(255)

def draw():
```

(continues on next page)

(continued from previous page)

```
        global armAngle, angleChange, ANGLE_LIMIT
        background(255)

        with push_matrix():
                translate(50, 50) # place robot so arms are always␣
↪on screen
                drawRobot()
                armAngle += angleChange

                # if the arm has moved past its limit,
                # reverse direction and set within limits.
                if armAngle > ANGLE_LIMIT or armAngle < 0:
                        angleChange = -angleChange
                        armAngle += angleChange

def drawRobot():
        no_stroke()
        fill(38, 38, 200)
        rect((20, 0), 38, 30) # head
        rect((14, 32), 50, 50) # body

        drawLeftArm()
        drawRightArm()

        rect((22, 84), 16, 50) # left leg
        rect((40, 84), 16, 50) # right leg

        fill(222, 222, 249)
        ellipse((30, 12), 12, 12) # left eye
        ellipse((47, 12), 12, 12) # right eye

def drawLeftArm():
        with push_matrix():
                translate(12, 32)
                rotate(radians(armAngle))
                rect((-12, 0), 12, 37)

def drawRightArm():
        with push_matrix():
                translate(66, 32)
                rotate(radians(-armAngle))
                rect((0, 0), 12, 37)

if __name__ == '__main__':
  run()
```

2.10.11 Case Study: Interactive Rotation

Instead of having the arms move on their own, we will modify the program so that the arms follow the mouse while the mouse button is pressed. Instead of just writing the program at the keyboard, we first think about the problem and figure out what the program needs to do.

Since the two arms move independently of one another, we need to have one variable for each arm's angle. It's easy to figure out which arm to track. If the mouse is at the left side of the robot's center, track the left arm; otherwise, track the right arm.

The remaining problem is to figure out the angle of rotation. Given the pivot point position and the mouse position, how do you determine the angle of a line connecting those two points? The answer comes from the `atan2()` function, which gives (in radians) the angle of a line from the origin to a given y and x coordinate. In constrast to most other functions, the y coordinate comes first. `atan2()` returns a value from -π to π radians, which is the equivalent of -180° to 180°.

But what about finding the angle of a line that doesn't start from the origin, such as the line from (10, 37) to (48, 59)? No problem; it's the same as the angle of a line from (0, 0) to (48-10, 59-37). In general, to find the angle of the line from (x0, y0) to (x1, y1), calculate

```
atan2(y1 - y0, x1 - x0)
```

Because this is a new concept, rather than integrate it into the robot program, you should write a simple test program to see that you understand how `atan2()` works. This program draws a rectangle whose center of rotation is its upper left corner at (100, 100) and tracks the mouse.

```
from p5 import *

def setup():
        size(200, 200)

def draw():
        angle = atan2(mouse_y - 100, mouse_x - 100)

        background(255)
        with push_matrix():
                translate(100, 100)
                rotate(angle)
                rect((0, 0), 50, 10)

if __name__ == '__main__':
  run()
```

That works great. What happens if we draw the rectangle so it is taller than it is wide? Change the preceding code to read `rect((0, 0), 10, 50)`. How come it doesn't seem to follow the mouse any more? The answer is that the rectangle really is still following the mouse, but it's the short side of the rectangle that does the following. Our eyes are trained to want the long side to be tracked. Because the long side is at a 90 degree angle to the short side, you have to subtract 90° (or $\pi/2$ radians) to get the desired effect. Change the preceding code to read rotate(angle - HALF_PI) and try it again. Since Processing deals almost exclusively in radians, the language has defined the constants PI (180°), HALF_PI (90°), QUARTER_PI (45°) and TWO_PI (360°) for your convenience.

At this point, we can write the final version of the arm-tracking program. We start off with definitions of constants and variables. The number 39 in the definition of MIDPOINT_X comes from the fact that the body of the robot starts at x-coordinate 14 and is 50 pixels wide, so 39 (14 + 25) is the horizontal midpoint of the robot's body.

```
# Where upper left of robot appears on screen
ROBOT_X = 50
ROBOT_Y = 50

# The robot's midpoint and arm pivot points
MIDPOINT_X = 39
LEFT_PIVOT_X = 12
RIGHT_PIVOT_X = 66
PIVOT_Y = 32

leftArmAngle = 0.0
rightArmAngle = 0.0

def setup():
        size(200, 200)
        background(255)

def draw():
        global leftArmAngle, rightArmAngle
        '''
        These variables are for mouseX and mouseY,
        adjusted to be relative to the robot's coordinate system
        instead of the window's coordinate system.
        '''
        mx = 0
        my = 0

        background(255)

        with push_matrix():
                translate(ROBOT_X, ROBOT_Y) # place robot with arms␣
↪always on screen
                if mouse_is_pressed:
                        mX = mouse_x - ROBOT_X
                        mY = mouse_y - ROBOT_Y

                        if mx < MIDPOINT_X: # left side of robot
                                leftArmAngle = atan2(mY - PIVOT_Y,␣
↪mX - LEFT_PIVOT_X) - HALF_PI
                        else:
                                rightArmAngle = atan2(mY - PIVOT_Y,␣
↪mX - RIGHT_PIVOT_X) - HALF_PI
```

(continues on next page)

(continued from previous page)

```
        drawRobot()
```

The `drawRobot()` function remains unchanged, but a minor change to `drawLeftArm()` and `drawRightArm()` is now necessary. Because `leftArmAngle` and `rightArmAngle` are now computed in radians, the functions don't have to do any conversion. The changes to the two functions are in bold.

```
def drawLeftArm():
        global leftArmAngle
        with push_matrix():
                translate(12, 32)
                rotate(leftArmAngle)
                rect((-12, 0), 12, 37)

def drawRightArm():
        global rightArmAngle
        with push_matrix():
                translate(66, 32)
                rotate(rightArmAngle)
                rect((0, 0), 12, 37)
```

2.11 PShape

Authors Daniel Shiffman; Arihant Parsoya (p5 port)

Copyright If you see any errors or have comments, please let us know. The tutorial was ported to p5 by Arihant Parsoya. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

One of the very first things you learn when programming with Processing is how to draw "primitive" shapes to the screen: rectangles, ellipses, lines, triangles, and more.

```
rect((x,y),w,h)
ellipse((x,y),w,h)
line((x1,y1),(x2,y2))
triangle((x1,y1),(x2,y2),(x3,y3))
```

A more advanced drawing option is to use `begin_shape()` and `end_shape()` to specify the vertices of a custom polygon

```
beginShape()
vertex(x1,y1)
vertex(x2,y2)
vertex(x3,y3)
vertex(x4,y4)
# etc
endShape()
```

And you can build more complex shapes by grouping a set of drawing functions together, even perhaps organizing them into a class.

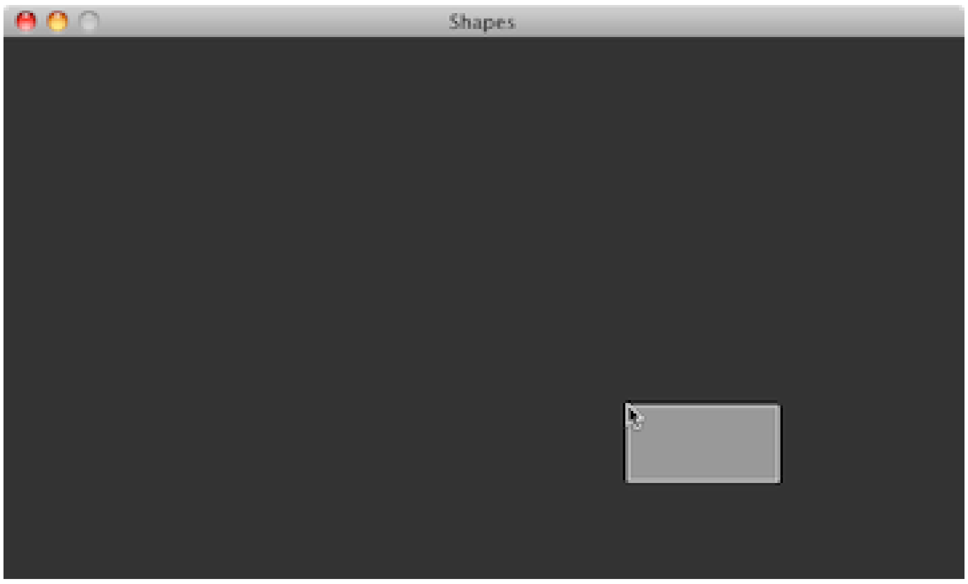
```
class MyWackyShape:

        # constructor

        # some functions

        def display(self):
                begin_shape()
                vertex(x1,y1)
                vertex(x2,y2)
                vertex(x3,y3)
                vertex(x4,y4)
                # etc
                end_shape()
```

This is all well and good and will get you pretty far. There's very little you can't draw just knowing the above. However, there is another step. A step that can, in some cases, improve the speed of your rendering as well as offer a more advanced organizational model for your code—PShape.

PShape is a datatype for storing shapes. These can be shapes that you build out of custom geometry or shapes that you load from an external file, such as an SVG.

2.11.1 Primitive PShapes

Let's begin with one of the simplest cases for use of a PShape. Here's a simple Processing `draw()` method that draws an rectangle following the mouse.



```
def draw():
        background(51)
```

(continues on next page)

(continued from previous page)

```
        stroke(255)
        fill(127)
        rect((mouse_x, mouse_y), 100, 50)
```

Pretty basic stuff. If this was all the code we had, there's not necessarily a good reason for using a PShape instead, but we're going to push ahead and make a PShape rectangle anyway as a demonstration. Our goal here is to have a variable that stores the color and dimensions of that variable, allowing our draw function to look like this.

```
def draw():
        background(51)
        draw_shape(rectangle)
```

And what is this "rectangle"? It's a PShape.

To initialize a PShape, use the PShape() method. The first argument for PShape() is a constant, and this constant specifies the type of PShape you intend to make. Here we are making a primitive shape, a rectangle. So now our setup() should look like:

```
rectangle = None

def setup():
        size(640, 360)
        rectangle = PShape()
```

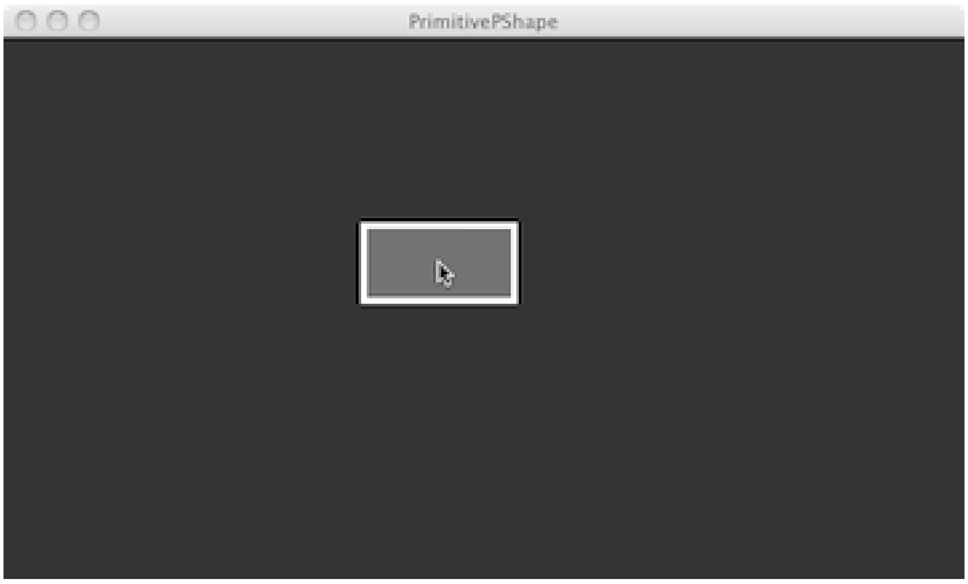
We can then move the shape according to the mouse with translate.

```
def draw():
        background(51)
        translate(mouse_x, mouse_y)
        draw_shape(rectangle)
```

One of the nice things about the PShape object is that it can also store color information in addition to geometry. Once a shape has been created in order to alter its fill or stroke, use the methods `set_fill()`, `set_stroke()`, `set_stroke_weight()`, etc.

```
def setup():
        size(640, 260)
        rectangle = PShape()
        with rectangle.edit():
                rectangle.add_vertex((0, 0))
                rectangle.add_vertex((50, 0))
                rectangle.add_vertex((50, 100))
                rectangle.add_vertex((0, 100))

        rectangle.stroke = color(255)
        rectangle.stroke_weight = 4
        rectangle._fill = color(127)
```

These methods can be called during draw() as well if you want to change the color of the shape dynamically.

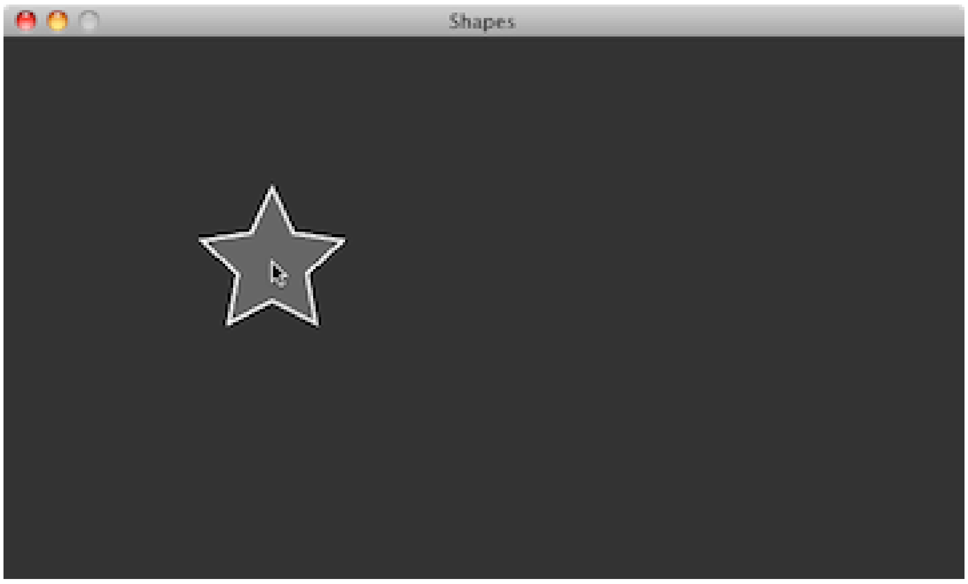


```
def draw():
        background(51)
        translate(mouse_x, mouse_y)
        rectangle.set_fill(color(remap(mouse_x, (0, width), (0,␣
↪255))))
        draw_shape(rectangle)
```

It should be noted that unlike with `fill()` and `stroke()` you must pass a full color as an argument. i.e. instead of saying `set_fill(255,0,0)` for a red fill, you'll need to say `set_fill(color(255,0,0))`. In addition, `set_fill()` and `set_stroke()` can take a boolean argument (e.g. `set_fill(false)`) to turn the fill or stroke on or off for a given vertex as well as an integer (e.g. `set_fill(i,color(255,0,0))`) to set the fill or stroke for a specific vertex.

2.11.2 Custom PShapes

PShapes also can be configured with custom vertices. You've probably done this before, without PShape, just using `begin_shape()` and `end_shape()`. For example, let's say you wanted to draw a star in Processing. You might have some code as follows:



```
def draw():
        background(51)
        translate(mouse_x, mouse_y)
        fill(102)
        stroke(255)
        stroke_weight(2)
        begin_shape()
        vertex(0, -50)
        vertex(14, -20)
        vertex(47, -15)
        vertex(23, 7)
        vertex(29, 40)
        vertex(0, 25)
        vertex(-29, 40)
        vertex(-23, 7)
        vertex(-47, -15)
        vertex(-14, -20)
        end_shape("CLOSE")
```

Here, just as in the previous example, our goal will be to draw the shape itself as an object in draw().

```
def draw():
        size(640, 360)
        star = PShape() # First create the shape
        with rectangle.edit():
                # All the vertex information goes here.
```

Then all the vertices (and colors) can be specified by calling the functions on the new PShape object "star." Note that `fill()` and `stroke` are not required here, only if you choose to alter the colors after the shape has been initially created.

```
def setup():
        global star
```

(continues on next page)

(continued from previous page)

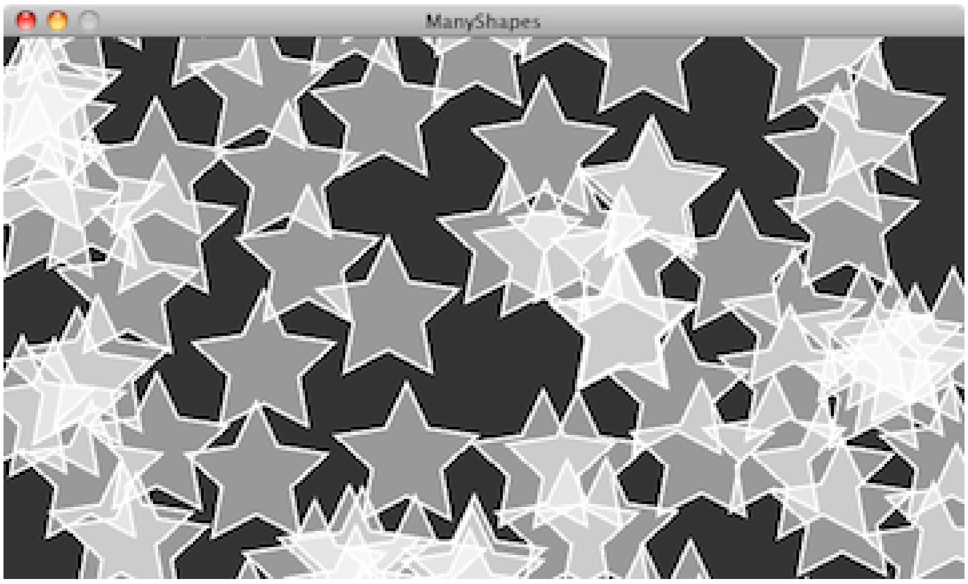
```
        size(640, 360)

        # First create the shape
        star = PShape()

        with star.edit():
                star.add_vertex((0, -50))
                star.add_vertex((14, -20))
                star.add_vertex((47, -15))
                star.add_vertex((23, 7))
                star.add_vertex((29, 40))
                star.add_vertex((0, 25))
                star.add_vertex((-29, 40))
                star.add_vertex((-23, 7))
                star.add_vertex((-47, -15))
                star.add_vertex((-14, -20))
```

2.11.3 Many PShapes

As we mentioned earlier, one reason to use PShape is just to help you organize your geometry. However, there's another reason. Let's assume for a moment that you have a Star class, with a `display()` function that looks like so:

```
def display():
    with push_matrix():
        translate(x, y)
        fill(102)
        stroke(255)
        stroke_weight(2)

        begin_shape()
        vertex(0, -50)
        vertex(14, -20)
        vertex(47, -15)
        vertex(23, 7)
        vertex(29, 40)
        vertex(0, 25)
        vertex(-29, 40)
        vertex(-23, 7)
        vertex(-47, -15)
        vertex(-14, -20)
        end_shape("CLOSE")
```

and in *draw()*, you are iterating through an array of Star objects, displaying each one.



```
def draw():
        background(51)
        for i in range(len(stars)):
                stars[i].display()
```

```
class Star:
        def __init__(self):
                self.s = PShape()
                self.x = 0
                self.y = 0
```

That PShape then needs to be initialized in the constructor. This can be done directly, right there in the class.

```
class Star:
        def __init__(self):
                #  First create the shape
                self.s = PShape()
                # You can set fill and stroke
                self.s._fill(102)
                self.s.stroke(Color(255))
                self.s.stroke_weight(2)
                # Here, we are hardcoding a series of vertices

                with self.s.edit():
                        self.s.add_vertex((0, -50))
                        self.s.add_vertex((14, -20))
                        self.s.add_vertex((47, -15))
                        self.s.add_vertex((23, 7))
                        self.s.add_vertex((29, 40))
                        self.s.add_vertex((0, 25))
                        self.s.add_vertex((-29, 40))
                        self.s.add_vertex((-23, 7))
```

(continues on next page)

(continued from previous page)

```
                    self.s.add_vertex((-47, -15))
                    self.s.add_vertex((-14, -20))
```

This method makes sense if each object itself has its own geometry, generated via an algorithm. However, if each object is displaying the identical PShape, it likely makes more sense to pass in a reference to a PShape in the constructor itself. Let's take a look at how this might work. Let's say we create a generic class called "Polygon" which has a reference to a PShape (which is draws in a display method).

```
class Polygon:
        def __init__(self, shape):
                self.s = shape

        def display():
                shape(s)
```

In the previous example, the shape was created right there in the object's constructor. Here we are going to demonstrate a different way to write the constructor where the shape is set via an argument.

```
from p5 import *

poly = None

class Polygon:
    def __init__(self, shape):
        self.shape = shape

    def display(self):
        draw_shape(self.shape)

def setup():
    global poly
    size(640, 360)

    star = PShape()
    star._fill = Color(0, 127)
    with star.edit():
        star.add_vertex((0, -50))
        star.add_vertex((14, -20))
        star.add_vertex((47, -15))
        star.add_vertex((23, 7))
        star.add_vertex((29, 40))
        star.add_vertex((0, 25))
        star.add_vertex((-29, 40))
        star.add_vertex((-23, 7))
        star.add_vertex((-47, -15))
        star.add_vertex((-14, -20))
```

(continues on next page)

(continued from previous page)

```
    poly = Polygon(star)

def draw():
    global poly
    background(255)
    poly.display()

if __name__ == '__main__':
    run()
```

This is a very flexible approach. For example if you had an array of PShape objects, you could create new Polygon objects each one with a random PShape.

2.11.4 PShape Groups

Another convenience of PShape is the ability to group shapes. For example, what if you wanted to create an alien creatures out of a set of circles, rectangles, and custom polygons. If the head were a circle and the body a rectangle, you might think you need:

```
from p5 import *

alien = None
head = None
body = None

def setup():
    global alien, head, body
    size(640, 360)
    alien = PShape([[0, 0]])
    head = PShape()
    body = PShape()

    with head.edit():
        head.add_vertex((-25, 0))
        head.add_vertex((25, 0))
        head.add_vertex((0, -50))


    with body.edit():
        body.add_vertex((-25, 0))
        body.add_vertex((25, 0))
        body.add_vertex((25, 100))
        body.add_vertex((-25, 100))

    alien.add_child(head)
    alien.add_child(body)
    no_loop()

```

(continues on next page)

(continued from previous page)

```
def draw():
    global alien, body, head
    background(0)
    translate(width/2, height/2)
    draw_shape(alien)

if __name__ == '__main__':
    run()
```

PShape groups allow you build a sophisticated hierarchy of shapes. This in turn allows you to set the color and attributes of the child shapes by calling the corresponding method at the parent level. Similarly, by calling the transformation functions at a given level of the hierarchy, you only affect the shapes below.

2.12 Data

Authors Daniel Shiffman; Arihant Parsoya (p5 port)

Copyright This tutorial is from the book Learning Processing by Daniel Shiffman, published by Morgan Kaufmann, © 2008 Elsevier Inc. All rights reserved. The tutorial was ported to p5 by Arihant Parsoya. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

This tutorial picks up where the Strings and Drawing Text tutorial leaves off and examines how to use String objects as the basis for reading and writing data. We'll start by learning more sophisticated methods for manipulating Strings, searching in them, chopping them up, and joining them together. Afterwards, we'll see how these skills allow us to use input from data sources, such as text files, web pages, xml feeds, and 3rd party APIs and take a step into the world of data visualization.

2.12.1 Manipulating Strings

In Strings and Drawing Text, we touched on a few of the basic functions available in the Java String, such as upper() and len(). These functions are documented on the Processing reference page for Strings. Nevertheless, in order to perform some more advanced data parsing techniques, we'll need to explore some additional String manipulation functions documented in the Python API.

Let's take a closer look at the following two String functions: index() and slicing.

index() locates a sequence of characters within a string. It takes one argument — a search string — and returns a numeric value that corresponds to the first occurrence of the search string inside of the String object being searched.

```
search = "def"
toBeSearched = "abcdefghi"
index = toBeSearched.index(search) # The value of index in this␣
↪example is 3.
```

Strings are just like arrays, in that the first character is index number zero and the last character is the length of the string minus one. If the search string cannot be found, index() returns -1. This is a good choice because -1 is not a legitimate index value in the string itself, and therefore can indicate “not found.” There are no negative indices in a string of characters or in an array.

Strings are just like arrays, in that the first character is index number zero and the last character is the length of the string minus one. If the search string cannot be found, ValueError is returned by the program.

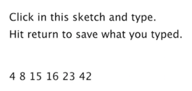
After finding a search phrase within a string, we might want to separate out part of the string, saving it in a different variable. A part of a string is known as a substring and substrings are made by slicing the array using two arguments, a start index and an end index. slicing the array returns the substring in between the two indices.

```
alphabet = "abcdefghi"
sub = alphabet[3:6] # The String sub is now "def".
```

Note that the substring begins at the specified start index (the first argument) and extends to the character at end index (the second argument) minus one. I know, I know. Wouldn’t it have been easier to just take the substring from the start index all the way to the end index? While this might initially seem true, it’s actually quite convenient to stop at end index minus one. For example, if you ever want to make a substring that extends to the end of a string, you can simply go all the way to len(thestring). In addition, with end index minus one marking the end, the length of the substring is easily calculated as end index minus begin index.

2.12.2 Splitting and Joining Strings

In Strings and Drawing Text, we saw how strings can be joined together (referred to as “concatenation”) using the “+” operator. Let’s review with a example that uses concatenation to get user input from a keyboard.



```
Click in this sketch and type.
Hit return to save what you typed.

4 8 15 16 23 42
```

```
from p5 import *

f = None
# Variable to store text currently being typed
typing = ""
# Variable to store saved text when return is hit
saved = ""

```

(continues on next page)

(continued from previous page)

```
def setup():
        global f
        size(300,200)
        f = create_font("Arial.ttf", 16)

def draw():
        global f
        background(255)

        indent = 25

        # Set the font and fill for text
        text_font(f)
        fill(0)

        # Display everything
        text("Click in this sketch and type. \nHit return to save␣
↪what you typed.", (indent, 40))
        text(typing, (indent, 90))
        text(saved, (indent, 130))

def key_pressed():
        global typing, saved
        # If the return key is pressed, save the String and clear it
        if key == "ENTER":
                saved = typing
                typing = ""
        else: # Otherwise, concatenate the String
                typing = typing + str(key)

if __name__ == '__main__':
        run()
```

Processing has two additional functions that make joining strings (or the reverse, splitting them up) easy. In sketches that involve parsing data from a file or the web, you might get hold of that data in the form of an array of strings or as one long string. Depending on what you want to accomplish, it's useful to know how to switch between these two modes of storage. This is where these two new functions, split() and join(), will come in handy.

"one long string or array of strings" ←→ {"one", "long", "string", "or" ,"array", "of", "strings"}

Let's take a look at the split() function. split() separates a longer string into an array of strings, based on a split character known as the delimiter. It takes the delimiter as the argument. (The delimiter can be a single character or a string.) In the code below, the period is not set as a delimiter and therefore will be included in the last string in the array: "dog." Note how printArray() can be used to print the contents of an array and their corresponding indices to the message console.

```
# Splitting a string based on spaces
spaceswords = "The quick brown fox jumps over the lazy dog."
list = spaceswords.split(" ")
print(list)
```

Here is an example using a comma as the delimiter (this time passing in a single character: ‘,’.)

```
# Splitting a string based on commas
commaswords = "The,quick,brown,fox,jumps,over,the,lazy,dog."
list = commaswords.split(",")
```

If you are splitting numbers in a string, the resulting elements of the array can be converted into an integer array with Python’s int() function. Numbers in a string are not numbers and cannot be used in mathematical operations unless you convert them first.

```
# Calculate sum of a list of numbers in a String
numbers = "8,67,5,309"
# Converting the String array to an int array
list = numbers.split(",")
sum = 0
for i in list:
        sum += int(i)

print(sum)
```

The reverse of split() is join(). join() takes an array of strings and joins them together into one long String object. The join() function also takes two arguments, the array to be joined and a separator. The separator can either be a single character or a string of characters.

```
lines = ["It", "was", "a", "dark", "and", "stormy", "night."]
```

Using the “+” operator along with a for loop, you can join a string together as follows:

```
# Manual Concatenation
onelongstring = ""

for i in lines:
        onelongstring += i + " "
```

The join() function, however, allows you to bypass this process, achieving the same result in only one line of code.

```
onelongstring = " ".join(lines)
```

2.12.3 Dealing with Data

Data can come from many different places: websites, news feeds, spreadsheets, databases, and so on. Let’s say you’ve decided to make a map of the world’s flowers. After searching online you might find a PDF version of a flower encyclopedia, or a spreadsheet of flower genera, or a JSON feed of flower data, or a REST API that provides geolocated lat/lon coordinates, or some web page someone put together with beautiful flower photos, and so on and so forth. The question inevitably arises: "I found all this data; which should I use, and how do I get it into Processing?"

If you are really lucky, you might find a Processing library that hands data to you directly with code. Maybe the answer is to just download this library and write some code like:

```
import flowers

void setup():
        fdb = FlowerDatabase()
        sunflower = fdb.findFlower("sunflower")
        h = sunflower.getAverageHeight()
```

In this case, someone else has done all the work for you. They've gathered data about flowers and built a Processing library with a set of functions that hands you the data in an easy-to-understand format. This library, sadly, does not exist (not yet), but there are some that do. For example, YahooWeather is a library by Marcel Schwittlick that grabs weather data from Yahoo for you, allowing you to write code like weather.getWindSpeed() or weather.getSunrise() and more. There is still plenty of work to do in the case of using a library.

Let's take another scenario. Say you're looking to build a visualization of Major League Baseball statistics. You can't find a Processing library to give you the data but you do see everything you're looking for at mlb.com. If the data is online and your web browser can show it, shouldn't you be able to get the data in Processing? Passing data from one application (like a web application) to another (say, your Processing sketch) is something that comes up again and again in software engineering. A means for doing this is an API or "application programming interface": a means by which two computer programs can talk to each other. Now that you know this, you might decide to search online for "MLB API". Unfortunately, mlb.com does not provide its data via an API. In this case you would have to load the raw source of the website itself and manually search for the data you're looking for. While possible, this solution is much less desirable given the considerable time required to read through the HTML source as well as program algorithms for parsing it.

Each means of getting data comes with its own set of challenges. The ease of using a Processing library is dependent on the existence of clear documentation and examples. But in just about all cases, if you can find your data in a format designed for a computer (spreadsheets, XML, JSON, etc.), you'll be able to save some time in the day for a nice walk outside.

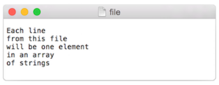
One other note worth a mention about working with data. When developing an application that involves a data source, such as a data visualization, it's sometimes useful to develop with "dummy" or "fake" data. You don't want to be debugging your data retrieval process at the same time as solving problems related to algorithms for drawing. In keeping with my one-step-at-a-time mantra, once the meat of the program is completed with dummy data, you can then focus solely on how to retrieve the actual data from the real source. You can always use random or hard-coded numbers into your code when you're experimenting with a visual idea and connect the real data later.

2.12.4 Working with Text Files

Let's begin by working with the simplest means of data retrieval: reading from a text file. Text files can be used as a very simple database (you could store settings for a program, a list of high scores, numbers for a graph, etc.) or to simulate a more complex data source.

In order to create a text file, you can use any simple text editor. Windows Notepad or Mac OS X TextEdit will do; just make sure you format the file as "plain text." It is also advisable to name the text files with the ".txt" extension, to avoid any confusion. And just as with image files, these text files should be placed in the sketch's "data" directory in order for them to be recognized by the Processing sketch.

Once the text file is in place, Python's open() function is used to read the content of the file into a String array.

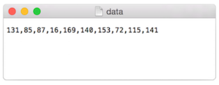


```
Each line
from this file
will be one element
in an array
of strings
```

```
# This code will print all the lines from the source text file.
file = open('words.txt', 'r').read()
lines = file.split('\n') # split the string at \n
print("There are " + str(len(lines)) + " lines")
print(lines)
```

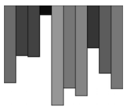
To run the code, create a text file called "file.txt," type a bunch of lines in that file, and place it in your sketch's data directory.

Text from a file can be used to generate a simple visualization. Take the following data file.



```
131,85,87,16,169,140,153,72,115,141
```

The results of visualizing this data are shown below. **Graphing Comma-Separated Numbers from a Text File**



```
from p5 import *
data = []

def setup():
        global data
        size(200, 200)
        # Load the text file as a string
        file = open('data.csv', 'r').read()
        stuff = file.split(',') # split the string at ,
        # Convert the string into an array of integers
        for number in stuff:
                data.append(int(number))
```

(continues on next page)

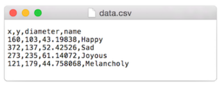
(continued from previous page)

```python

def draw():
        global data
        background(255)
        stroke(0)
        for i in range(len(data)):
                # Use array of ints to set the color and height of␣
↪each rectangle.
                rect((i*29, 0), 20, data[i])

if __name__ == '__main__':
        run()
```

Looking at how to parse a csv file with `split()` was a nice learning exercise. In truth, dealing with csv files (which can easily be generated from spreadsheet software such as Google docs) is such a common activity. Python's inbuilt csv library can be used to parse csv files.

2.12.5 Tabular Data

A table consists of data arranged as a set of rows and columns, also called "tabular data." If you've ever used a spreadsheet, this is tabular data. Python's csv takes the file and parses the values and automatically places the contents into a Table object storing the data in columns and rows. This is a great deal more convenient than struggling to manually parse large data files with split(). It works as follows. Let's say you have a data file that looks like:



```
x,y,diameter,name
160,103,43.19838,Happy
372,137,52.42526,Sad
273,235,61.14072,Joyous
121,179,44.758068,Melancholy
```

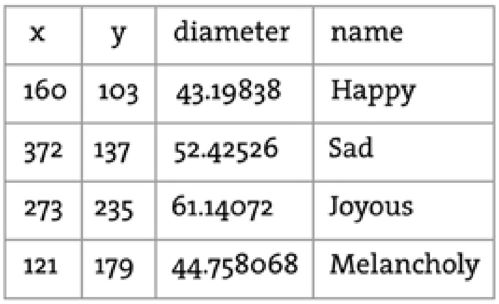
We can now use:

```python
with open("data.csv") as f:
        table = csv.DictReader(f)
        table = list(table) # Convert DictReader object to list
```

Now I've missed an important detail. Take a look again at the data.csv text file above. Notice how the first line of text is not the data itself, but rather a header row. This row includes labels that describe the data included in each subsequent row. The good news is that Processing can automatically interpret and store the headers for you, if you pass in the option "header" when loading the table. (In addition to "header", there are other options you can specify. For example, if your file is called data.txt but is comma separated data you can pass in the option "csv". If it also has a header row, then you can specifiy both options like so: "header,csv").

```
table = loadTable("data.csv", "header");
```

Now that the table is loaded, I can show how you grab individual pieces of data or iterate over the entire table. Let's look at the data visualized as a grid.



x	y	diameter	name
160	103	43.19838	Happy
372	137	52.42526	Sad
273	235	61.14072	Joyous
121	179	44.758068	Melancholy

In the above grid you can see that the data is organized in terms of rows and columns. Python's *csv.DictReader* converts the csv file into a list of dictionaries. To assess a particular element from the csv file, we need to specify the number of row in the list and the name of the attribute.

```
val1 = table[2]["y"] # val now has the value 235

val2 = table[3]["diameter"] # val2 now has the value 44.758068

s = table[0]["name"] # s now has the value "Happy"
```

To access the entire row, the index of the row can be used as follows:

```
row = table[2]
```

One I have the row object, I can ask for data from some or all the columns

```
x = row["x"] # x has the value 273
y = row["y"] # y has the value 235
d = row["diameter"] # d has the value 61.14072
s = row["name"] # s has the value "Joyous"
```

If you want to grab all the rows and iterate over them you can do so in a loop with a counter accessing each row one at a time.

```
for i in range(len(table)):
        # Access each row of the table one at a time, in a loop.
        row = table[i]
        x = row["x"]
        y = row["y"]
        d = row["diameter"]
        s = row["name"]

        # do something with the data
```

To add a new row to a Table, simply add new row to the array with dictionary of column

```
table.append({
        "x": mouse_x,
```

(continues on next page)

(continued from previous page)

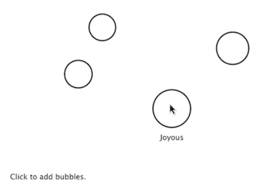
```
        "y": mouse_y,
        "diameter": random_uniform(40, 80),
        "name": "new label"
})
```

To delete a row, simply call the method *del* and pass in the numeric index of the row you would like removed. For example, the following code removes the first row whenever the size of the table is greater than ten rows.

```
if len(table) > 10:
        del table[0]
```

The following example puts all of the above code together. Notice how each row of the table contains the data for a Bubble object.

Loading and Saving Data



```
from p5 import *
import csv

table = []
bubbles = []

def setup():
        size(480, 360)
        loadData()

def draw():
        global bubbles
        background(255)
        # Display all bubbles
        for i in range(len(bubbles)):
                bubbles[i].display()

def loadData():
        global table, bubbles
        table = []
        bubbles = []
        with open("data.csv") as f:
                table = csv.DictReader(f)
                table = list(table) # Convert DictReader object to␣
↪list
```

(continues on next page)

(continued from previous page)

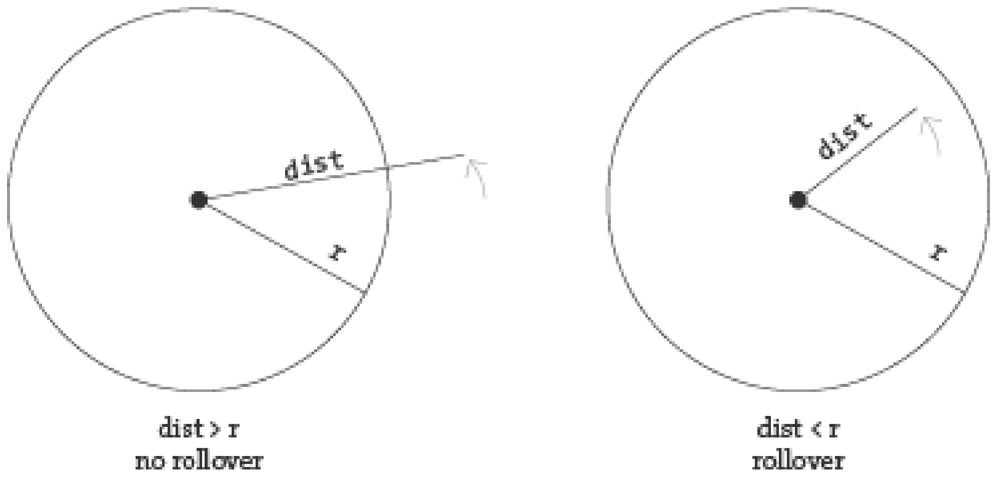
```
        for i in range(len(table)):
                #  Iterate over all the rows in a table.
                row = table[i]

                # Access the fields via their column name (or␣
↪index).
                bubbles.append(Bubble(float(row["x"]), float(row["y
↪"]), float(row["diameter"]), row["name"]))

def mouse_pressed():
        global table
        # When the mouse is pressed, create a new row and set the␣
↪values for each column of that row.
        table.append({
                "x": mouse_x,
                "y": mouse_y,
                "diameter": random_uniform(40, 80),
                "name": "Blah!"
        })

        # If the table has more than 10 rows, delete the oldest row.
        if len(table) > 10:
                del table[0]

        # This writes the table back to the original CSV file
        # and reloads the file so that what's drawn matches.
        with open("data.csv", "w") as f:
                dict_writer = csv.DictWriter(f, table[0].keys())
                dict_writer.writeheader()
                dict_writer.writerows(table)

        loadData()

class Bubble:
        def __init__(self, tempX, tempY, tempD, s):
                self.x = tempX
                self.y = tempY
                self.diameter = tempD
                self.name = s

        def rollover(self, px, py):
                d = dist((px, py), (self.x, self.y))
                if d < self.diameter / 2:
                        return True
                else:
                        return False

        def display(self):
                stroke(0)
```

(continues on next page)

(continued from previous page)

```
                stroke_weight(2)
                no_fill()
                ellipse((self.x, self.y), self.diameter, self.
↪diameter)
                if self.rollover(self.x, self.y):
                        fill(0)
                        text_align("CENTER")
                        text(self.name, self.x, self.y + self.
↪diameter/2 + 20)

if __name__ == '__main__':
        run()
```

Here, the distance between a given point and a circle's center is compared to that circle's radius as depicted:



In the code below, the function returns a boolean value (true or false) depending on whether the point (mx,my) is inside the circle. Notice how radius is equal to half the diameter.

```
def rollover(px, py):
        d = dist((px, py), (x, y))
        if d < self.diameter / 2:
                return True
        else:
                return False
```

2.12.6 Data that is not in a Standardized Format

What if your data is not in a standard format like a table, how do you deal with it then? Python's requests library can be used to pull text from an URL.

```
lines = requests.get("http://www.yahoo.com")
```

When you send a URL path into *urllib.request.urlopen()*, you get back the raw HTML (Hypertext Markup Language) source of the requested web page. It's the same stuff that appears upon selecting "View Source" from a browser's menu options. You don't need to be an HTML expert to follow this section, but if you are not familiar at all with HTML, you might want to read http://en.wikipedia.org/wiki/HTML.

Unlike with the comma-delimited data from a text file that was specially formatted for use in a Processing sketch, it's not practical to have the resulting raw HTML stored in an array of strings (each element representing one line from the source). Converting the array into one long string can make things a bit simpler. As you saw earlier in the chapter, this can be achieved using join().

```
onelongstring = " ".join(lines)
```

When pulling raw HTML from a web page, it's likely you do not want all of the source, but just a small piece of it. Perhaps you're looking for weather information, a stock quote, or a news headline. You can take advantage of the text manipulation functions you learned — index(), substring(), and len() — to find pieces of data within a large block of text. Take, for example, the following String object:

```
stuff = "Number of apples:62. Boy, do I like apples or what!"
```

Let's say I want to pull out the number of apples from the above text. My algorithm would be as follows:

1. Find the end of the substring "apples:" Call it start.
2. Find the first period after "apples:" Call it end.
3. Make a substring of the characters between start and end.
4. Convert the string to a number (if I want to use it as such).

In code, this looks like:

```
stuff = "Number of apples:62. Boy, do I like apples or what!"
start = stuff.index("apples:" ) + 7 # STEP 1
# The index where a string ends can be found by
# searching for that string and adding its length (here, 8).
end = stuff.index(".", start) # STEP 2
apples = stuff[start: end] # STEP 3
apple_no = int(apples) # STEP 4
```

The above code will do the trick, but I should be a bit more careful to make sure I don't run into any errors if I do not find the string I am searching for. I can add some error checking and generalize the code into a function:

```
# A function that returns a substring between two substrings.
# If the beginning of end "tag" is not found, the function returns␣
↪an empty string.
```

(continues on next page)

(continued from previous page)

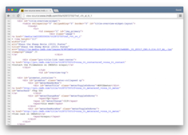
```
def giveMeTextBetween(s, startTag, endTag):
        # Find the index of the beginning tag
        try:
                startIndex = s.index(startTag) # STEP 1
        except ValueError:
                return ""
        # Move to the end of the beginning tag
        startIndex += len(startTag)
        try:
                # Find the index of the end tag
                endIndex = s.index(endTag, startIndex)
        except ValueError:
                return""

        return s[startIndex: endIndex]

stuff = "Number of apples:62. Boy, do I like apples or what!"
print(giveMeTextBetween(stuff, "apples:", '.'))
```

With this technique, you are ready to connect to a website from within Processing and grab data to use in your sketches. For example, you could read the HTML source from nytimes.com and look for today's headlines, search finance.yahoo.com for stock quotes, count how many times the word "flower" appears on your favorite blog, and so on. However, HTML is an ugly, scary place with inconsistently formatted pages that are difficult to reverse engineer and parse effectively. Not to mention the fact that companies change the source code of web pages rather often, so any example that I might make while I am writing this paragraph might break by the time you read this paragraph.

For grabbing data from the web, an XML (Extensible Markup Language) or JSON (JavaScript Object Notation) feed will prove to be more reliable and easier to parse. Unlike HTML (which is designed to make content viewable by a human's eyes) XML and JSON are designed to make content viewable by a computer and facilitate the sharing of data across different systems. Most data (news, weather, and more) is available this way, and I will look at examples in #beginner_xml and #JSON. Though much less desirable, manual HTML parsing is still useful for a couple reasons. First, it never hurts to practice text manipulation techniques that reinforce key programming concepts. But more importantly, sometimes there is data you really want that is not available in an API format, and the only way to get it is with such a technique. (I should also mention that regular expressions, an incredibly powerful techinque in text pattern matching, could also be employed here. As much as I love regex, it's unfortunately beyond the scope of this tutorial.)

An example of data only available as HTML is the Internet Movie Database. IMDb contains information about movies sorted by year, genre, ratings, etc. For each movie, you can find the cast and crew list, a plot summary, running time, a movie poster image, the list goes on. However, IMDb has no API and does not provide its data as XML or JSON. Pulling the data into Processing therefore requires a bit of detective work. Let's look at the page for the Shaun the Sheep Movie

Looking in the HTML source from the above URL, I find a giant mess of markup.



It's up to me to pore through the raw source and find the data I am looking for. Let's say I want to know the running time of the movie and grab the movie poster image. After some digging, I find that the movie is 139 minutes long as listed in the following HTML.

```
<div class="txt-block">
  <h4 class="inline">Runtime:</h4>
    <time itemprop="duration" datetime="PT139M">139 min</time>
</div>
```

For any given movie, the running time itself will be variable, but the HTML structure of the page will stay the same. I can therefore deduce that running time will always appear in between:

```
<time itemprop="duration" datetime="PT139M">
```

and

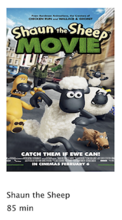
```
</time>
```

Knowing where the data starts and ends, I can use giveMeTextBetween() to pull out the running time. A quote in Java marks the beginning or end of a string. So how do you include an actual quote in a String object? The answer is via an "escape" sequence. A quote can be included using a backward slash, followed by a quote. For example: String q = "This String has a quote "in it";

```
import urllib.request

q = "This String has a quote \"in it"
url = "http://www.imdb.com/title/tt0058331"
response = urllib.request.urlopen(url)
lines = []
for line in response.readlines():
        lines.append(line.decode("utf-8"))

html = " ".join(lines)

start = ""
end = ""
runningtime = giveMeTextBetween(html, start, end)
print(runningtime)
```

The following code retrieves both the running time and movie poster iamge from IMDb and displays it onscreen.

Parsing IMDb Manually



Shaun the Sheep
85 min

```
import requests
from p5 import *

poster = None
runningtime = None

def setup():
        size(300, 350)
        loadData()

def draw():
        global poster, runningtime
        # Display all the stuff I want to display
        background(255)
        image(poster, (10, 10), 164, 250)
        fill(0)
        text("Shaun the Sheep", (10, 300))
        text(runningtime, (10, 320))

def loadData():
        global poster, runningtime
        url = "http://www.imdb.com/title/tt2872750/"
        # Get the raw HTML source into an array of strings (each␣
↪line is one element in the array).
        # The next step is to turn array into one long string with␣
↪join().

        html = requests.get(url).text
        start = ""
        end = ""
        runningtime = giveMeTextBetween(html, start, end) #␣
↪Searching for running time.

        start = ""
        # Search for the URL of the poster image.
        imgUrl = giveMeTextBetween(html, start, end)
        # Now, load that image!
        poster = load_image(imgUrl)
```

(continues on next page)

(continued from previous page)

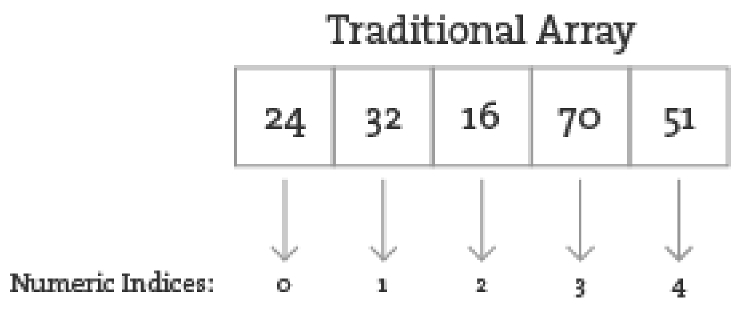
```
# A function that returns a substring between two substrings.
# If the beginning of end "tag" is not found, the function returns␣
↪an empty string.
def giveMeTextBetween(s, startTag, endTag):
        # Find the index of the beginning tag
        try:
                startIndex = s.index(startTag) # STEP 1
        except ValueError:
                return ""
        # Move to the end of the beginning tag
        startIndex += len(startTag)
        try:
                # Find the index of the end tag
                endIndex = s.index(endTag, startIndex)
        except ValueError:
                return""

        return s[startIndex: endIndex]

if __name__ == '__main__':
        run()
```

2.12.7 Text Analysis

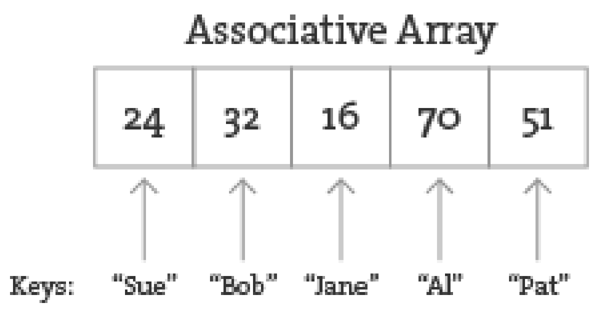
Loading text from a URL need not only be an exercise in parsing out small bits of information. It's possible with Processing to analyze large amounts of text found on the web from news feeds, articles, and speeches, to entire books. A nice source is Project Gutenberg which makes available thousands of public domain texts. Algorithms for analyzing text merits an entire book itself, but let's look at some basic techniques.

A text concordance is an alphabetical list of words that appear in a book or body of text along with contextual information. A sophisticated concordance might keep a list of where each word appears (like an index) as well as which words appear next to which other words. In this case, I'm going to create a simple concordance, one that simply stores a list of words and their corresponding counts, i.e., how many times they appeared in the text. Concordances can be used for text analysis applications such as spam filtering or sentiment analysis. To accomplish this task, I am going to use the Processing built-in class IntDict.

As you learned earlier, an array is an ordered list of variables. Each element of the array is numbered and be accessed by its numeric index.



However, what if instead of numbering the elements of an array you could name them? This element is named "Sue," this one "Bob," this one "Jane," and so on and so forth. In programming, this kind of data structure is often referred to as an associative array, map, or dictionary. It's a collection of (key, value) pairs. Imagine you had a dictionary of people's ages. When you look up "Sue" (the key), the definition, or value, is her age, 24.



Associative arrays can be incredibly convenient for various applications. For example, you could keep a list of student IDs (student name, id) or a list of prices (product name, price) in a dictionary. Here a dictionary is the perfect data structure to hold the concordance. Each element of the dictionary is a word paired with its count.

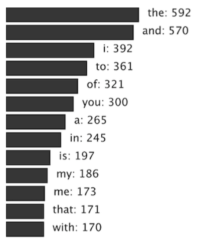
Creating an IntDict is as easy as calling an empty constructor. Let's say you want a dictionary to keep track of an inventory of supplies.

```
inventory = {}
```

Values can be paired with their keys using the following syntax:

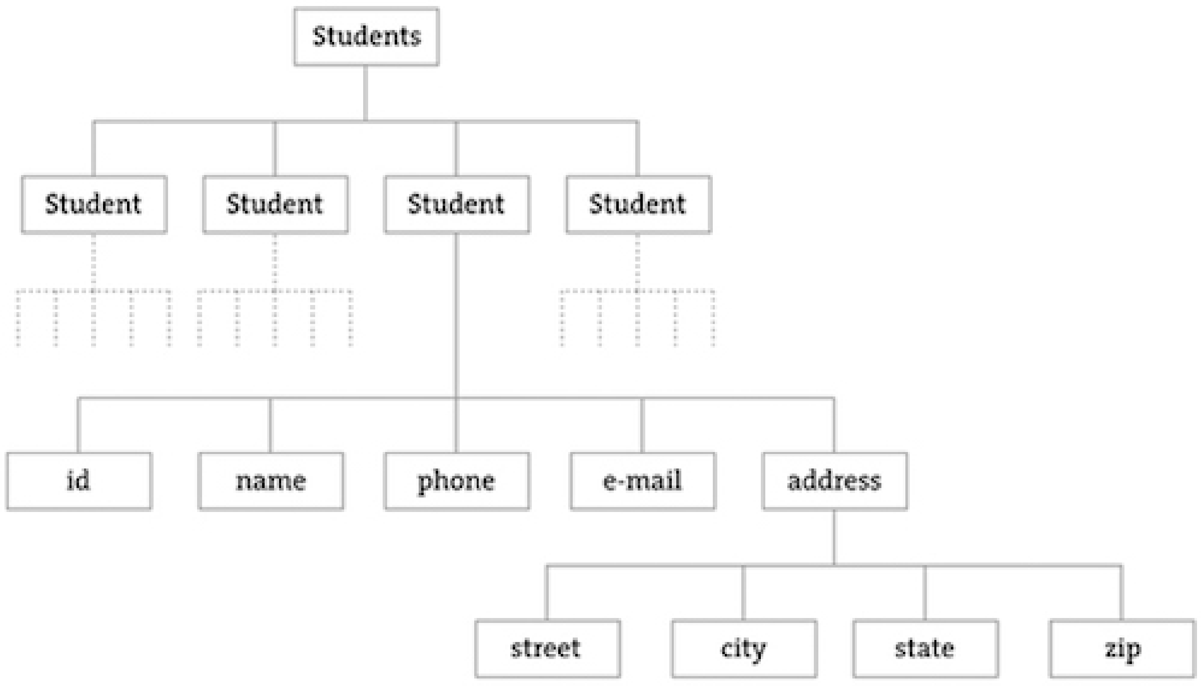
```
inventory["pencils"] = 10
inventory["paper clips"] = 128
inventory["pens"] = 16
```

Text Concordance Using IntDict



If your data is available via a standardized format such as XML or JSON, the process of manually searching through text for individual pieces of data is no longer required. XML is designed to facilitate the sharing of data across different systems, and you can retrieve that data using the built-in Processing XML class.

XML organizes information in a tree structure. Let's imagine a list of students. Each student has an ID number, name, address, email, and telephone number. Each student's address has a city, state, and zip code. An XML tree for this dataset might look like the following:



```
<?xml version = "1.0" encoding = "UTF-8 "?>
<students>
  <student>
    <id>001</id>
    <name>Daniel Shiffman</name>
    <phone>555-555-5555</phone>
    <email>daniel@shiffman.net</email>
    <address>
      <street>123 Processing Way</street>
      <city>Loops</city>
      <state>New York</state>
      <zip>01234</zip>
```

(continues on next page)

(continued from previous page)

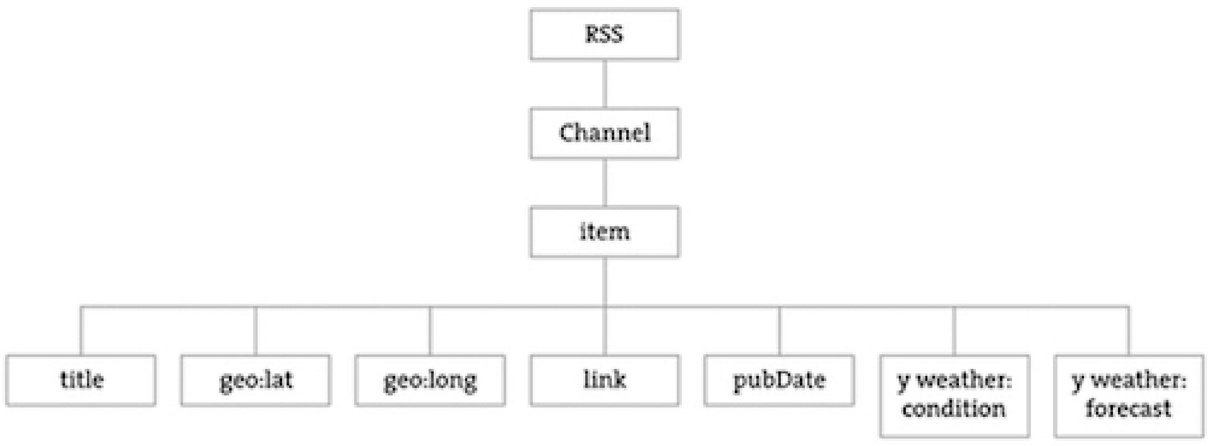
```
    </address>
  </student>
  <student>
    <id>002</id>
    <name>Zoog</name>
    <phone>555-555-5555</phone>
    <email>zoog@planetzoron.uni</email>
    <address>
      <street>45.3 Nebula 5</street>
      <city>Boolean City</city>
      <state>Booles</state>
      <zip>12358</zip>
    </address>
  </student>
</students>
```

Note the similarities to object-oriented programming. You can think of the XML tree in the following terms. The XML document represents an array of student objects. Each student object has multiple pieces of information, an ID, a name, a phone number, an email address, and a mailing address. The mailing address is also an object that has multiple pieces of data, such as street, city, state, and zip.

Let's look at some data made available from a web service such as Yahoo Weather. Here is the raw XML source. (Note I have edited it slightly for simplification purposes.)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<rss version="2.0" xmlns:yweather="http://xml.weather.yahoo.com/ns/
↪rss/1.0">
  <channel>
    <item>
      <title>Conditions for New York, NY at 12:49 pm EDT</title>
      <geo:lat>40.67</geo:lat>
      <geo:long>-73.94</geo:long>
      <link>http://us.rd.yahoo.com/dailynews/rss/weather/New_York__
↪NY//link>
      <pubDate>Thu, 24 Jul 2014 12:49 pm EDT</pubDate>
      <yweather:condition text="Partly Cloudy" code="30" temp="76"/>
      <yweather:forecast day="Thu" low="65" high="82" text="Partly␣
↪Cloudy"/>
    </item>
  </channel>
</rss>
```

The data is mapped in the tree stucture shown below:



You may be wondering what the top level "RSS" is all about. Yahoo's XML weather data is provided in RSS format. RSS stands for "Really Simple Syndication" and is a standardized XML format for syndicating web content (such as news articles, etc.). You can read more about RSS on Wikipedia

Now that you have a handle on the tree structure, let's look at the specifics inside that structure. With the exception of the first line (which simply indicates that this page is XML formatted), this XML document contains a nested list of elements, each with a start tag, that is, <channel>, and an end tag, that is, </channel>. Some of these elements have content between the tags:

```
<title>Conditions for New York, NY at 12:49 pm EDT</title>
```

and some have attributes (formatted by Attribute Name equals Attribute Value in quotes):

```
<yweather:forecast day="Thu" low="65" high="82" text="Partly Cloudy
↪"/>
```

2.12.8 Using the Processing XML Class

Since the syntax of XML is standardized, I could certainly use split(), indexof(), and substring() to find the pieces I want in the XML source. The point here, however, is that because XML is a standard format, I don't have to do this. Rather, I can use an XML parser. In Processing, XML can be parsed using the built-in Python's *xml* library.

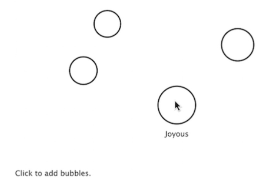
```
import xml.etree.ElementTree as ET
xml = ET.parse("filename.xml")
```

I'm now calling *ET.parse()* and passing the file of the XML document. An XML object represents one element of an XML tree. When a document is first loaded, that XML object is always the root element.

Using Processing's XML Class

The following example example uses an XML tree of bubble objects to render on the screen:

```
<?xml version="1.0" encoding="UTF-8"?>
<bubbles>
  <bubble>
    <position x="160" y="103"/>
    <diameter>43.19838</diameter>
    <label>Happy</label>
  </bubble>
  <bubble>
    <position x="372" y="137"/>
    <diameter>52.42526</diameter>
    <label>Sad</label>
  </bubble>
  <bubble>
    <position x="273" y="235"/>
    <diameter>61.14072</diameter>
    <label>Joyous</label>
  </bubble>
  <bubble>
    <position x="121" y="179"/>
    <diameter>44.758068</diameter>
    <label>Melancholy</label>
  </bubble>
</bubbles>
```



```python
import xml.etree.ElementTree as ET
from p5 import *

# An Array of Bubble objects
bubbles = []

def setup():
    size(480, 360)
    loadData()

def loadData():
    global bubbles
    # Load XML file
    xml = ET.parse("bubbles.xml")
    root = xml.getroot()
    for child in root:
        diameter = float(child.find("diameter").text)
        label = child.find("label").text
        x = int(child.find("position").attrib["x"])
```

(continues on next page)

(continued from previous page)

```
        y = int(child.find("position").attrib["y"])
        bubbles.append(Bubble(x, y, diameter, label))

def draw():
    global bubbles
    background(255)

    # display all bubbles
    for i in range(len(bubbles)):
        bubbles[i].display()
        bubbles[i].rollover(mouse_x, mouse_y)

class Bubble:
    def __init__(self, tempX, tempY, tempD, s):
        self.x = tempX
        self.y = tempY
        self.diameter = tempD
        self.name = s

    def rollover(self, px, py):
        d = dist((px, py), (self.x, self.y))
        if d < self.diameter / 2:
            return True
        else:
            return False

    def display(self):
        stroke(0)
        stroke_weight(2)
        no_fill()
        ellipse((self.x, self.y), self.diameter, self.diameter)
        if self.rollover(self.x, self.y):
            fill(0)
            text_align("CENTER")
            text(self.name, self.x, self.y + self.diameter/2 + 20)

if __name__ == '__main__':
    run()
```

2.12.9 JSON

Another increasingly popular data exchange format is JSON (pronounced like the name Jason), which stands for JavaScript Object Notation. Its design was based on the syntax for objects in the JavaScript programming language (and is most commonly used to pass data between web applications) but has become rather ubiquitous and language-agnostic. While you don't need to know anything about JavaScript to work in Processing, it won't hurt to get a sense of some basic JavaScript syntax while learning it.

JSON is an alternative to XML and the data can be looked at in a similarly tree-like manner. All JSON data comes in the following two ways: an object or an array. Luckily, you already know about these two concepts and only need to learn a new syntax for encoding them.

Let's take a look at a JSON object first. A JSON object is like a Processing object only with no functions. It's simply a collection of variables with a name and a value (or "name/value pair"). For example, following is JSON data describing a person:

```
{
        "name":"Olympia",
        // Each name/value pair is separated by a comma.
        "age":3,
        "height":96.5,
        "state":"giggling"
}
```

Notice how this maps closely to classes in Processing.

```
{
        "name":"Olympia",
        "age":3,
        "height":96.5,
        "state":"giggling",
        // The value of "brother" is an object containing two name/
↪value pairs.
        "brother":{
                "name":"Elias",
                "age":6
        }
}
```

Multiple JSON objects can appear in the data as an array. Just like the arrays you use in Processing, a JSON array is simply a list of values (primitives or objects). The syntax, however, is different with square brackets indicating the use of an array rather than curly ones. Here is a simple JSON array of integers:

```
[1, 7, 8, 9, 10, 13, 15]
```

You might find an array as part of an object.

```
{
        "name":"Olympia",
        // The value of "favorite colors" is an array of strings.
        "favorite colors":[
                "purple",
                "blue",
                "pink"
        ]
}
```

Or an array of objects themselves. For example, here is what the bubbles would look like in JSON. Notice how this JSON data is organized as a single JSON object "bubbles," which contains a JSON array of JSON objects, the bubbles. Flip back to compare to the CSV and XML versions of the same data.

```
{
  "bubbles":[
    {
      "position":{
        "x":160,
        "y":103
      },
      "diameter":43.19838,
      "label":"Happy"
    },
    {
      "position":{
        "x":372,
        "y":137
      },
      "diameter":52.42526,
      "label":"Sad"
    },
    {
      "position":{
        "x":273,
        "y":235
      },
      "diameter":61.14072,
      "label":"Joyous"
    }
  ]
}
```

2.13 Trigonometry Primer

Authors Ira Greenberg; Arihant Parsoya (p5 port)

Copyright This tutorial is from the book ‘ Processing: Creative Coding and Computational Art <https://processing.org/books/#ira>‘_, Ira Greenberg, Friends of ED, © 2007. If you see any errors or have comments, please let us know. The tutorial was ported to p5 by Arihant Parsoya. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

Trigonometry (really just a couple of the trig functions) is central to graphics programming. That being said, if you’re anything like me you probably have a hazy memory of trig. Perhaps you remember the mnemonic device soh-cah-toa to remember the relationships between the trig functions and a right triangle. Here’s a diagram to awaken your memory.

You should also notice in the figure that tangent equals sine(θ) over cosine(θ). You may also remember that sine and cosine are similar when you graph them, both forming periodic waves. Only the cosine wave is shifted a bit (90° or pi/2) on the graph, which is technically called a phase shift. I fully realize that it is difficult to deal with this stuff in the abstract. Fortunately, there is another model, the unit circle (shown below) used to visualize and study the trig functions.

The unit circle is a circle with a radius of 1 unit in length—hence its imaginative name. When you work with the unit circle, you don't use the regular and trusted Cartesian coordinate system; instead you use a polar coordinate system. The Cartesian system works great in a rectangular grid space, where a point can be located by a coordinate, such as (x, y). In a polar coordinate system, in contrast, location is specified by (r, θ), where r is the radius and θ (the Greek letter theta) is the angle of rotation. The unit circle has its origin at its center, and you measure angles of rotation beginning at the right-middle edge of the unit circle (facing 3 o'clock) and moving in a counterclockwise direction around it.

In the unit circle diagram, the point p is at 45° or pi / 4. You can use pi also to measure around the unit circle, as illustrated in the figure. Halfway around the circle (180°) is equal to pi radians, and all the way around the circle is equal to 2pi radians and also 0 radians, since a circle is continuous and ends where it begins. The number pi is a constant that is equal to the circumference of a circle divided by its diameter, and is approximately 3.142.

In the polar system, you use radians to measure angles, instead of degrees. The angle of rotation in radians is commonly referred to as θ (the Greek letter theta). The arc length of this rotation is calculated by r*θ where r is the radius. In a unit circle, with a radius of 1, θ is equal to the arc length of rotation (arc s in unit circle diagram). It's nice to know the arc length, but most of the time (in computer graphics), you really just want to know the location of a point in relation to the unit circle. For example, if I wanted to rotate a point around the unit circle, I'd need to know how to place and move the point in a circle. With the unit circle, this is an incredibly easy task and precisely the kind of thing trig is used for.

There is a really simple relationship between the trig functions and the unit circle. Notice in the unit circle diagram that from point p on the ellipse, a right triangle is formed within the unit circle. This should immediately make you think of good old Pythagoras. Notice also that r (the radius) is the hypotenuse of the right triangle. In addition, you now also know that with the trig functions, you can use theta and any one side (opposite, adjacent, or hypotenuse) to solve the rest of the triangle. The big payoff of these relationships, for our purposes, is that to translate point p in the polar coordinate system to the Cartesian coordinate system (the system used by our monitors), you would use these simple expressions:

```
x = cosine(theta) * radius
y = sine(theta) * radius
```

These seemingly humble little expressions are very powerful and can be exploited for all sorts of expressive and organic purposes.

Here's how you actually use the trig functions in Processing:

```
x = cos(radians(angle)) * radius;
y = sin(radians(angle)) * radius;
```

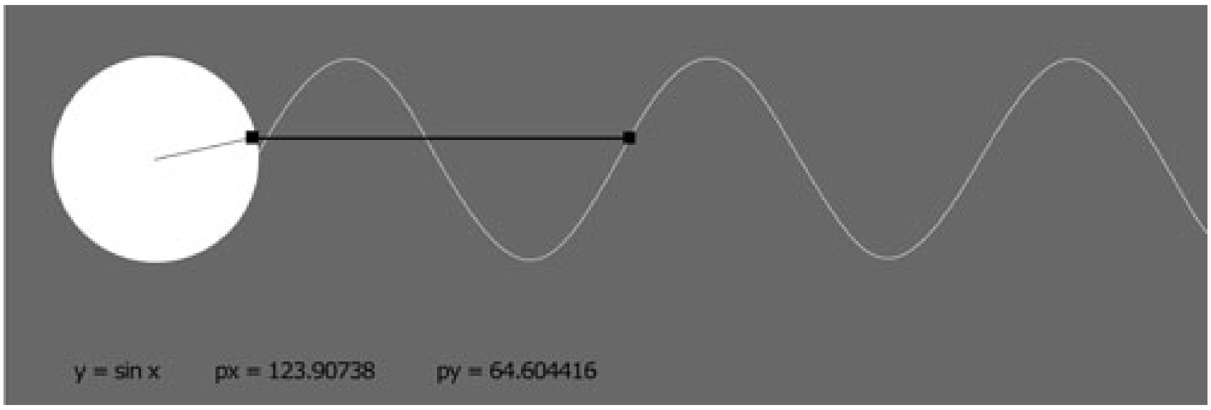
Notice the function call (radians(angle)) inside each of the trig function calls. Remember that theta is measured in radians, in the polar coordinate system. However, in the Cartesian coordinate system, you work in degrees. To convert between radians and degrees and vice versa, you can use the following expressions:

```
theta = angle*pi/180
angle = theta*180/pi
```

Or better yet, just use Processing's handy conversion functions:

```
theta = radians(angle)
angle = degrees(theta)
```

Lastly, I include a Processing sketch that demonstrates how the unit circle and sine function relate:



```
from p5 import *

px, py, px2, py2 = (0, 0, 0, 0)
angle, angle2 = (0, 0)
radius = 50
frequency = 2
frequency2 = 2
x, x2 = (0, 0)

def setup():
    size(600, 200)
    background(127)


def draw():
    global px, py, px2, py2
    global angle, angle2
    global radius, frequency2, frequency
    global x, x2

    background (127)
    no_stroke()
```

(continues on next page)

(continued from previous page)

```
    fill(255)
    ellipse((width/8, 75), radius*2, radius*2)

    # Rotates rectangle around circle
    px = width/8 + cos(radians(angle))*(radius)
    py = 75 + sin(radians(angle))*(radius)
    fill(0)

    rect ((px, py), 5, 5)
    stroke(100)
    line((width/8, 75), (px, py))
    stroke(200)

    # Keep reinitializing to 0, to avoid
    # flashing during redrawing
    angle2 = 0

    # Draw static curve - y = sin(x)
    for i in range(width):
        px2 = width/8 + cos(radians(angle2))*(radius)
        py2 = 75 + sin(radians(angle2))*(radius)
        point(width/8+radius+i, py2)
        angle2 -= frequency2

    # Send small ellipse along sine curve
    # to illustrate relationship of circle to wave
    no_stroke()
    ellipse((width/8+radius+x, py), 5, 5)
    angle -= frequency
    x+=1

    # When little ellipse reaches end of window,
    # set the variables back to 0
    if x >= width-60:
        x = 0
        angle = 0

    # Draw dynamic line connecting circular path with wave
    stroke(50)
    line((px, py), (width/8+radius+x, py))


if __name__ == '__main__':
    run()
```

2.14 Two-dimensional Arrays

Authors Daniel Shiffman; Arihant Parsoya (p5 port)

Copyright This tutorial is from the book Learning Processing by Daniel Shiffman, published by Morgan Kaufmann, © 2008 Elsevier Inc. All rights reserved. The tutorial was ported to p5 by Arihant Parsoya. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

An array keeps track of multiple pieces of information in linear order, a one-dimensional list. However, the data associated with certain systems (a digital image, a board game, etc.) lives in two dimensions. To visualize this data, we need a multi-dimensional data structure, that is, a multi-dimensional array. A two-dimensional array is really nothing more than an array of arrays (a three-dimensional array is an array of arrays of arrays). Think of your dinner. You could have a one-dimensional list of everything you eat:

```
(lettuce, tomatoes, steak, mashed potatoes, cake, ice cream)
```

Or you could have a two-dimensional list of three courses, each containing two things you eat:

```
(lettuce, tomatoes) and (steak, mashed potatoes) and (cake, ice␣
↪cream)
```

In the case of an array, our old-fashioned one-dimensional array looks like this:

```
myArray = [0,1,2,3]
```

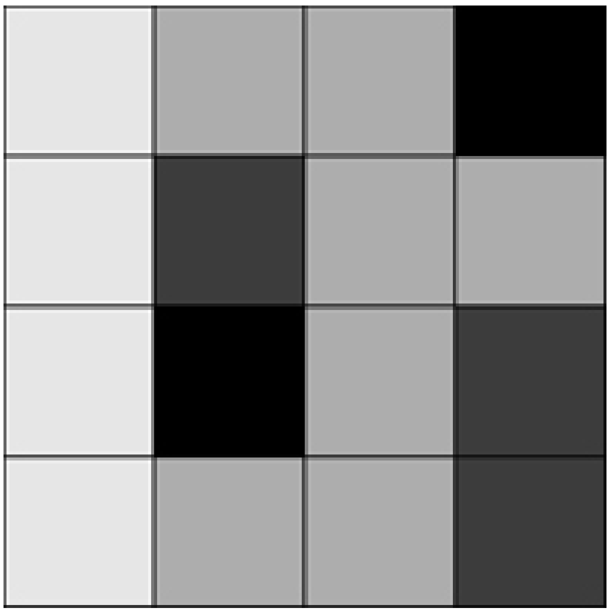
And a two-dimensional array looks like this:

```
myArray = [[0,1,2,3], [3,2,1,0], [3,5,6,1], [3,8,3,4]]
```

For our purposes, it is better to think of the two-dimensional array as a matrix. A matrix can be thought of as a grid of numbers, arranged in rows and columns, kind of like a bingo board. We might write the two-dimensional array out as follows to illustrate this point:

```
myArray = [
        [0,1,2,3],
        [3,2,1,0],
        [3,5,6,1],
        [3,8,3,4]]
```

We can use this type of data structure to encode information about an image. For example, the following grayscale image could be represented by the following array:



```
myArray = [
        [236, 189, 189,   0],
        [236,  80, 189, 189],
        [236,   0, 189,  80],
        [236, 189, 189,  80]]
```

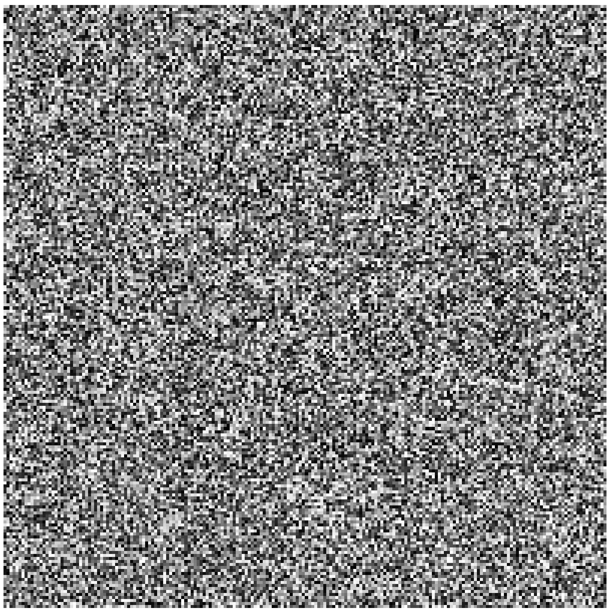
To walk through every element of a one-dimensional array, we use a for loop, that is:

```
for i in range(len(myArray)):
        myArray[i] = 0
```

For a two-dimensional array, in order to reference every element, we must use two nested loops. This gives us a counter variable for every column and every row in the matrix.

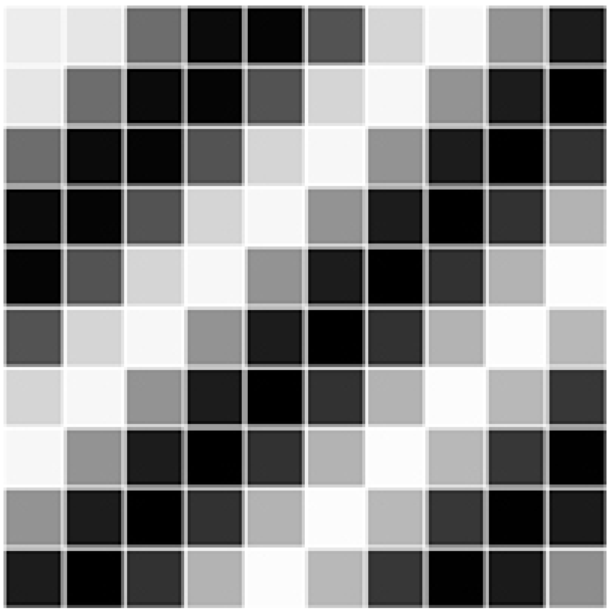
```
rows = 10
columns = 10

for i in range(rows):
        for j in range(columns):
                myArray[i][j] = 0
```

For example, we might write a program using a two-dimensional array to draw a grayscale image.



```
from p5 import *

myArray = []
rows = None
columns = None

def setup():

        size(200, 200)
        global rows, columns, myArray
        columns = width
        rows = height

        for i in range(rows):
                myArray.append([])
                for j in range(columns):
                        myArray[i].append(int(random_uniform(255)))

def draw():
        global rows, columns, myArray
        for i in range(rows):
                for j in range(columns):
                        stroke(myArray[i][j])
                        point(i, j)

if __name__ == '__main__':
        run()
```

A two-dimensional array can also be used to store objects, which is especially convenient for programming sketches that involve some sort of "grid" or "board." The following example displays a grid of Cell objects stored in a two-dimensional array. Each cell is a rectangle whose brightness oscillates from 0-255 with a sine function.



```
from p5 import *

grid = []

# Number of columns and rows in the grid
rows = 10
columns = 10

def setup():

        size(200, 200)
        global rows, columns, grid

        for i in range(rows):
                grid.append([])
                for j in range(columns):
                        grid[i].append(Cell(i*20,j*20,20,20,i+j))

def draw():
        global rows, columns, grid
        for i in range(columns):
                for j in range(rows):
                        grid[i][j].oscillate()
                        grid[i][j].display()

class Cell:
        def __init__(self, tempX, tempY, tempW, tempH, tempAngle):
                self.x = tempX
                self.y = tempY
                self.w = tempW
                self.h = tempH
                self.angle = tempAngle
```

(continues on next page)

(continued from previous page)

```
        def oscillate(self):
                self.angle += 0.02

        def display(self):
                stroke(255)
                # Color calculated using sine wave
                fill(127+127*sin(self.angle))
                rect((self.x, self.y), self.w, self.h)

if __name__ == '__main__':
        run()
```

2.15 Electronics

Authors Hernando Barragán; Casey Reas; Abhik Pal (p5 port)

Copyright This tutorial is "Extension 5" from Processing: A Programming Handbook for Visual Designers and Artists, Second Edition, published by MIT Press. © 2014 MIT Press. If you see any errors or have comments, please let us know. The tutorial was ported to p5 by Abhik Pal. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

Software is not limited to running on desktop computers, laptops, tablets, and phones. Contemporary cameras, copiers, elevators, toys, washing machines, and artworks found in galleries and museums are controlled with software. Programs written to control these objects use the same concepts discussed earlier in this book (variables, control structures, arrays, etc.), but building the physical parts requires learning about electronics. This text introduces the potential of electronics with examples from art and design and discusses basic terminology and components. Examples written with Wiring and Arduino (two electronics toolkits related to Processing) are presented and explained.

2.15.1 Electronics in the arts

Electronics emerged as a popular material for artists during the 1960s. Artists such as Naum Gabo and Marcel Duchamp used electrical motors in prior decades, but the wide interest in kinetic sculpture and the foundation of organizations such as Experiments in Art and Technology (E.A.T.) are evidence of a significant new emphasis. For instance, in *The Machine* exhibition at The Museum of Modern Art in 1968, Wen-Ying Tsai exhibited *Cybernetic Sculpture*, a structure made of vibrating steel rods illuminated by strobe lights flashing at high frequencies. Variations in the vibration frequency and the light flashes produced changes in the perception of the sculpture. The sculpture responded to sound in the surrounding environment by changing the frequency of the strobe lights. Peter Vogel, another kinetic sculpture pioneer, created sculptures that generate sound. The sculptures have light sensors (photocells) that detect and respond to a person's shadow when she approaches the sculpture. The sculptures are built almost entirely with electrical components. The organization of these components forms both the shape of the sculpture and its behavior. Other pioneers during the 1960s include Nam June Paik, Nicolas Schöffer, James Seawright, and Takis.

The range of electronic sculpture created by contemporary artists is impressive. Tim Hawkinson produces sprawling kinetic installations made of cardboard, plastic, tape, and electrical components. His *Überorgan* (2000) uses mechanical principles inspired by a player piano to control the flow of air through balloons the size of whales. The air is pushed through vibrating reeds to create tonal rumbles and squawks. This physical energy contrasts with the psychological tension conveyed through Ken Feingold's sculptures. His *If/Then* (2001) is two identical, bald heads protruding from a cardboard box filled with packing material. These electromechanical talking heads debate their existence and whether they are the same person. Each head listens to the other and forms a response from what it understands. Speech synthesis and recognition software are used in tandem with mechanisms to animate the faces – the result is uncanny.

The works of Maywa Denki and Crispin Jones are prototypical of a fascinating area of work between art and product design. Maywa Denki is a Japanese art unit that develops series of products (artworks) that are shown in product demonstrations (live performances). Over the years, they have developed a progeny of creatures, instruments, fashion devices, robots, toys, and tools – all animated by motors and electricity. Devices from the *Edelweiss Series* include *Marmica*, a self-playing marimba that opens like a flower, and *Mustang*, a gasoline-burning aroma machine for people who love exhaust fumes. Crispin Jones creates fully functioning prototypes for objects that are critical reflections of consumer technologies. *Social Mobiles (SoMo)*, developed in collaboration with IDEO, is a set of mobile phones that address the frustration and anger caused by mobile phones in public places. The project humorously explores ways mobile phone calls in public places could be made less disruptive. The *SoMo 1* phone delivers a variable electrical shock to the caller depending on how loud the person at the other end of the conversation is speaking. The ringtone for *SoMo 4* is created by the caller knocking on their phone. As with a knock on a door, the attitude or identity of the caller is revealed through the sound. Related artists include the Bureau of Inverse Technology, Ryota Kuwakubo, and the team of Tony Dunne and Fiona Raby.

As electronic devices proliferate, it becomes increasingly important for designers to consider new ways to interact with these machines. Working with electronics is an essential component of the emerging interaction design community. The Tangible Media Group (TMG) at the MIT Media Laboratory, led by Hiroshi Ishii, pioneered research into tangible user interfaces to take advantage of human senses and dexterity beyond screen GUIs and clicking a mouse. *Curlybot* is a toy that can record and play back physical movement. It remembers how it was moved and can replay the motion including pauses, changes in speed, and direction. *MusicBottles* are physical glass bottles that trigger sounds when they are opened. To the person who opens the bottles, the sounds appear to be stored within the bottles, but technically, custom-designed electromagnetic tags allow a special table to know when a bottle has been opened, and the sound is played through nearby speakers. These and other projects from the TMG were instrumental in moving research in interface design away from the screen and into physical space. Research labs at companies like Sony and Philips are other centers for research and innovation into physical interaction design. Academic programs such as New York University's Interactive Telecommunication Program, the Design Interactions course at the Royal College of Art, and the former Interaction Design Institute Ivrea have pioneered educational strategies within in this area.

2.15.2 Electricity

Electricity is something we use daily, but it is difficult to understand. Its effect is experienced in many ways, from observing a light turn on to noticing the battery charge deplete on a laptop computer.

Electrical current is a stream of moving electrons. They flow from one point to another through a *conductor*. Some materials are better conductors than others. Sticking a fork in a light socket is dangerous because metal is a good conductor and so is your body. The best conductors are copper, silver, and gold. A resistor is the opposite of a conductor. Resistance is the capability of a material to resist the flow of electrons. A substance with a very high resistance is an *insulator*. Plastic and rubber are excellent insulators, and for this reason they are used as the protective covering around wires. Electrical energy, the difference of electrical potential between two points, is called *voltage*. The amount of electrical charge per second that flows through a point is the *current*. Resistance is measured in units called ohms, voltage is measured in volts, and current is measured in amperes (amps). The relation between the three is easiest to understand through an analogy of water flowing through a hose. As explained by the educators Dan O'Sullivan and Tom Igoe:

> The flow of water through a hose is like the flow of electricity through a circuit. Turning the faucet increases the amount of water coming through the hose, or increases the current (amps). The diameter of the hose offers resistance to the current, determining how much water can flow. The speed of the water is equivalent to voltage. When you put your thumb over the end of the hose, you reduce the diameter of the pathway of the water. In other words, the resistance goes up. The current (that is, how much water is flowing) doesn't change, however, so the speed of the water, or voltage, has to go up so that all the water can escape. . . [1]

Electrical current flows in two ways: direct current (DC) and alternating current (AC). A DC signal always flows in the same direction and an AC signal reverses the direction of flow at regular intervals. Batteries and solar cells produce DC signals, and the power that comes from wall sockets is an AC signal:

Depending on your country, the AC power source coming into your home is between 100 and 240 volts. Most home appliances can directly use AC current to operate, but some use a power supply to convert the higher-potential AC current into DC current at smaller voltages. A common example of this type of power supply are the black plastic boxes (a k a power bricks, power adapters, wall warts) that are used to power laptops or mobile phones from the home AC power source. Most desktop computers have an internal power supply to convert the AC source to the 12-volt and 5-volt DC supply necessary to run the internal electronics. Low voltages are generally safer than high voltages, but it's the amount of current (amps) that makes electricity dangerous.

[1] Dan O'Sullivan and Tom Igoe, *Physical Computing: Sensing and Controlling the Physical World with Computers* (Thomson Course Technology PTR, 2004), p. 5

2.15.3 Components

Electronic components are used to affect the flow of electricity and to convert electrical energy into other forms such as light, heat, and mechanical energy. There are many different components, each with a specific use, but here we introduce four of the most basic types: resistor, capacitor, diode, and transistor.

Resistor

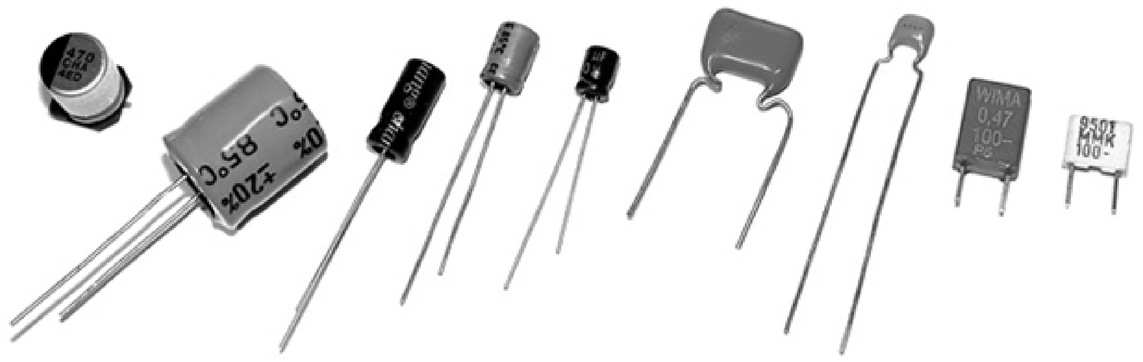
A resistor limits (provides resistance to) the flow of electricity. Resistors are measured in units called ohms. The value 10 ohms is less resistance than 10,000 (10K) ohms. The value of each resistor is marked on the component with a series of colored bands. A variable resistor that changes its resistance when a slider, knob, or dial attached to it is turned is called a potentiometer or trimmer. Variable resistors are designed to change in response to different environmental phenomena. For example, one that changes in response to light is called a photoresistor or photocell, and one that changes in response to heat is called a thermistor. Resistors can be used to limit current, reduce voltage, and perform many other essential tasks.

Capacitor

A capacitor stores electrons i.e. electrical charge; it gains charge when current flows in, and it releases charge (discharges) when the current flows out. This can smooth out the dips and spikes in a current signal. Capacitors are combined with resistors to create filters, integrators, differentiators, and oscillators. A simple capacitor is two parallel sheets of conductive materials, separated by an insulator. Capacitors are measured in units called farads. A farad is a large measurement, so most capacitors you will use will be measured in microfarads (µF), picofarads (pF), or nanofarads (nF).

Diode

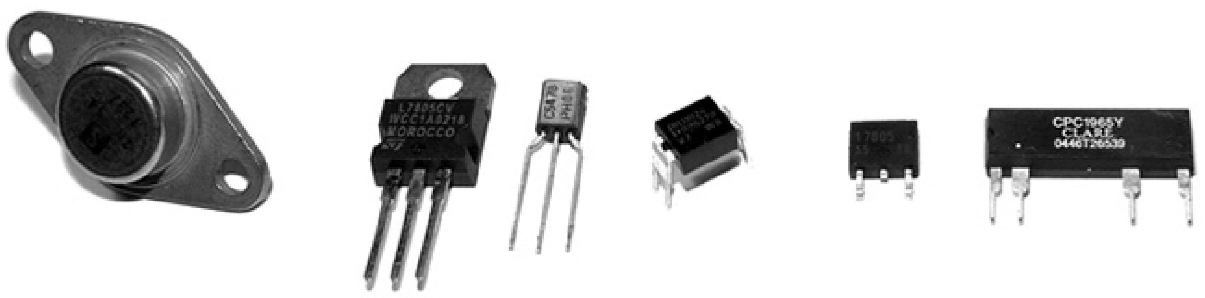
Current flows only in one direction through a diode. One side is called the cathode (marked on the device with a line) and the other is the anode. Current flows when the anode is more positive than the cathode. Diodes are commonly used to block or invert the negative part of an AC signal. A light-emitting diode (LED) is used to produce light. The longer wire coming out



of the LED is the anode and the other is the cathode. LEDs come in many sizes, forms, colors, and brightness levels.

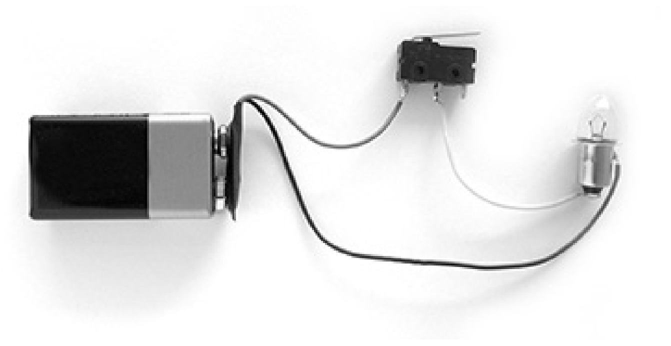
Transistor

A transistor can be used as an electrical switch or an amplifier. A bipolar transistor has three leads (wires) called the base, collector, and emitter. Depending on the type of transistor, applying current to the base either allows current to flow or stops it from flowing through the device from the collector to the emitter. Transistors make it possible for the low current from a microcontroller to control the much higher currents necessary for motors and other power-hungry devices, and thus to turn them on and off.



2.15.4 Circuits

An electrical circuit is a configuration of components, typically designed to produce a desired behavior such as decreasing the current, filtering a signal, or turning on an LED. The following simple circuit can be used to turn a light on and off:



This simple electric circuit is a closed loop with an energy source (battery), a load (lightbulb) that offers a resistance to the flow of electrons and transforms the electric energy into another form of energy (light), wires that carry the electricity, and a switch to connect and disconnect the wires. The electrons move from one end of the battery, through the load, and to the other end.

Circuits are usually represented with diagrams. A circuit diagram uses standardized symbols to represent specific electrical components. It is easier to read the connections on a diagram than on photographs of the components. A diagram of the simple circuit above could look like this:

Circuits are often prototyped on a "breadboard," a rectangular piece of plastic with holes for inserting wires. A breadboard makes it easy to quickly make variations on a circuit without soldering (fusing components together with a soft metal). Conductive strips underneath the surface connect the long horizontal rows at the top and bottom of the board and the short vertical rows within the middle:

Circuits are tested with a multimeter, an instrument to measure volts, current, resistance, and other electrical properties. A multimeter allows the electrical properties of the circuit to be read as numbers and is necessary for debugging. Analog multimeters have a small needle that moves from left to right, and digital multimeters have a screen that displays numbers. Most multimeters have two metal prongs to probe the circuit and a central dial to select between different modes.

Commonly used circuits are often condensed into small packages. These integrated circuits (ICs, or chips) contain dense arrangements of miniaturized components. They are typically small, black plastic rectangles with little metal pins sticking out of the sides. Like objects in software, these devices are used as building blocks for creating more complicated projects. ICs are produced to generate signals, amplify signals, control motors, and perform hundreds of other functions. They fit neatly into a breadboard by straddling the gap in the middle.

2.15.5 Microcontrollers and I/O boards

Microcontrollers are small and simple computers. They are the tiny computer brains that automate many aspects of contemporary life, through their activities inside devices ranging from alarm clocks to airplanes. A microcontroller has a processor, memory, and input/output interfaces enclosed within a single programmable unit. They range in size from about 1 × 1 cm to 5 × 2 cm. Like desktop computers, they come in many different configurations. Some have the same speed and memory as a personal computer from twenty years ago, but they are much less powerful than current machines, as this comparison tables shows:

Model	Speed	Memory	Cost
Apple Macintosh (1984)	8MHz	128 Kb	$2500
Atmel ATmega128-8AC Microcontroller	8MHz	128 Kb	$15
Apple Mac Mini (2006)	1500 MHz	512,000 Kb	$600

Small metal pins poking out from a microcontroller's edges allow access to the circuits inside. Each pin has its own role. Some are used to supply power, some are for communication, some are inputs, and others can be set to either input or output. The relative voltage at each input pin can be read through software, and the voltage can be set at each output pin. Some pins are reserved for communication. They allow a microcontroller to communicate with computers and other microcontrollers through established communication protocols such as RS-232 serial.

Microcontrollers can be used to build projects directly, but they are often packaged with other components onto a printed circuit board (PCB) to make them easier to use for beginners and for rapid prototyping. We call these boards I/O boards (input/output boards) because they are used to get data in and out of a microcontroller. They are also called microcontroller modules. We've created three informal groups – bare microcontrollers, programmable I/O boards, and tethered I/O boards – to discuss different ways to utilize microcontrollers in a project.

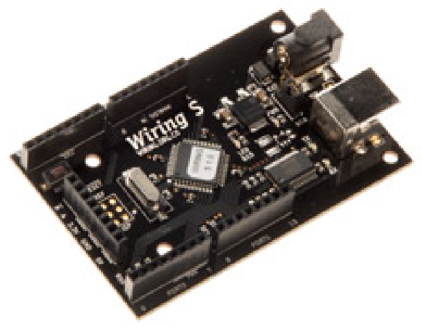
Bare microcontrollers

Working directly with a bare microcontroller is the most flexible but most difficult way to work. It also has the potential to be the least expensive way of building with electronics, but this economy can be offset by initial development costs and the extra time spent learning how to use it. Microchip PIC and Atmel AVR are two popular families of microcontrollers. Each has variations ranging from simple to elaborate that are appropriate for different types of projects. The memory, speed, and other features effect the cost, the number of pins, and the size of the package. Both families feature chips with between eight and 100 pins with prices ranging from under $1 to $20. PIC microcontrollers have been on the market for a longer time, and more example code, projects, and books are available for beginners. The AVR chips have a more modern architecture and a wider range of open-source programming tools. Microcontrollers are usually programmed in the C language or their assembly language, but it's also possible to program them in other languages such as BASIC. If you are new to electronics and programming, we don't recommend starting by working directly with PIC or AVR chips. In our experience, beginners have had more success with the options introduced below.

Programmable I/O boards

A programmable I/O board is a microcontroller situated on a PCB with other components to make it easier to program, attach/detach components, and turn on and off. These boards typically have components to regulate power to protect the microcontroller and a USB or RS-232 serial port connector to make it easy to attach cables for communication. The small pins on the microcontroller are wired to larger pins called headers, which make it easy to insert and remove sensors and motors. Small wires embedded within the PCB connect pins to a corresponding header. Small reset switches make it easy to restart the power without having to physically detach the power supply or battery.

Within the context of this book, the most relevant I/O boards are Wiring and Arduino. Both were created as tools for designers and artists to build prototypes and to learn about electronics. Both boards use the Wiring language to program their microcontrollers and use a development environment built from the Processing environment. In comparison to the Processing language, the Wiring language provides a similar level of control and ease of use within its domain. They share common language elements when possible, but Wiring has some functions specific to programming microcontrollers and omits the graphics programming functions within Processing. Like Processing programs, Wiring programs are translated into another language before they are run. When a program written with the Wiring language is compiled, it's first translated into the C/C++ language and then compiled using a C/C++ compiler.



Tethered I/O boards

A tethered I/O board is used to get sensor data into a computer and to control physical devices (motors, lights, etc.) without the need to program the board. A computer already has many input and output devices such as a monitor, mouse, and keyboard; and tethered I/O boards provide a way to communicate between more exotic input devices such as light sensors and video cameras, and output devices such as servomotors and lights. These boards are designed to be easy to use. They often do not require knowledge of electronics because sensors and motors can be plugged directly into the board and do not need to interface with other components. Messages are sent and received from the boards through software such as Processing, Max, Flash, and many programming languages. This ease of use often comes at a high price.

2.15.6 Sensors

Physical phenomena are measured by electronic devices called sensors. Different sensors have been invented to acquire data related to touch, force, proximity, light, orientation, sound, temperature, and much more. Sensors can be classified into groups according to the type of signals they produce (analog or digital) and the type of phenomena they measure. Analog signals are continuous, but digital signals are constrained to a range of values (e.g., 0 to 255):

Most basic analog sensors utilize resistance. Changes in a physical phenomenon modify the resistance of the sensor, therefore varying the voltage output through the sensor. An analog-to-digital converter can continuously measure this changing voltage and convert it to a number that can be used by software. Sensors that produce digital signals send data as binary values to an attached device or computer. These sensors use a voltage (typically between 3.5 and 5 volts) as ON (binary digit 1 or TRUE) and no voltage as a OFF (binary digit 0 or FALSE). More complex sensors include their own microcontrollers to convert the data to digital signals and to use established communication protocols for transmitting these signals to another computer.

Touch and Force

Sensing of touch and force is achieved with switches, capacitive sensors, bend sensors, and force-sensitive resistors. A switch is the simplest way to detect touch. A switch is a mechanism that stops or allows the flow of electricity depending on its state, either open (OFF) or closed (ON). Some switches have many possible positions, but most can only be ON or OFF. Touch can also be detected with capacitive sensors. These sensors can be adjusted to detect the touch and proximity (within a few millimeters) of a finger to an object. The sensor can be positioned underneath a nonconductive surface like glass, cardboard, or fabric. This type of sensor is often used for the buttons in an elevator. A bend (flex) sensor is a thin strip of plastic that changes its resistance as it is bent. A force-sensitive resistor (FSR or force sensor) changes its resistance depending on the magnitude of force applied to its surface. FSRs are designed for small amounts of force like the pressure from a finger, and they are available in different shapes including long strips and circular pads.

Presence and distance

There are a wide variety of sensors to measure distance and determine whether a person is present. The simplest way to determine presence is a switch. A switch attached to a door, for example, can be used to determine whether it is open or closed. A change in the state (open or closed) means someone or something is there. Switches come in many different shapes and sizes, but the category of small ones called *microswitches* are most useful for this purpose. The infrared (IR) motion detectors used in security systems are another simple way to see if something is moving in the environment. They can’t measure distance or the degree of motion, but they have a wide range, and some types can be purchased at hardware stores. IR distance sensors are used to calculate the distance between the sensor and an object. The distance is converted into a voltage between 0 and 5 volts that can be read by a microcontroller. Ultrasonic sensors are used for measuring up to 10 meters. This type of device sends a sound pulse and calculates how much time it takes to receive the echo.

Light

Sensors for detecting light include photoresistors, phototransistors, and photodiodes. A photoresistor (also called a photocell) is a component that changes its resistance with varying levels of light. It is among the easiest sensors to use. A phototransitor is more sensitive to changes in light and is also easy to use. Photodiodes are also very sensitive and can respond faster to changing light levels, but they are more complex to interface with a microcontroller. Photodiodes are used in the remote control receivers of televisions and stereos.

Position and orientation

A potentiometer is a variable resistor that works by twisting a rotary knob or by moving a slider up and down. The potentiometer's resistance changes with the rotation or up/down movement, and this can affect the voltage level within a circuit. Most rotary potentiometers have a limited range of rotation, but some are able to turn continuously. A tilt sensor is used to crudely measure orientation (up or down). It is a switch with two or more wires and a small metal ball or mercury in a box that touches wires in order to complete a circuit when it is in a certain orientation. An accelerometer measures the change in movement (acceleration) of an object that it is mounted to. Tiny structures inside the device bend as a result of momentum, and the amount of bending is measured. Accelerometers are used in cameras to control image stabilization and in automobiles to detect rapid deceleration and release airbags. A digital compass calculates orientation in relation to the earth's magnetic field. The less expensive sensors of this type have a lower accuracy, and they may not work well when situated near objects that emit electromagnetic fields (e.g., motors).

Sound

A microphone is the simplest and most common device used to detect and measure sound. Sudden changes in volume are the easiest sound elements to read, but processing the sound wave with software (or special hardware) makes it possible to detect specific frequencies or rhythms. A microphone usually requires extra components to amplify the signal before it can be read by a microcontroller. Piezo electric film sensors, commonly used in speakers and microphones, can also be used to detect sound. Sampling a sound wave with a microcontroller can dramatically reduce the quality of the audio signal. For some applications, it's better to sample and analyze sound through a desktop computer and to communicate the desired analysis information to an attached microcontroller.

Temperature

A thermistor is a device that changes its resistance with temperature. These sensors are easy to interface, but they respond slowly to changes. To quantitatively measure temperature, a more sophisticated device is needed. Flame sensors are tuned to detect open flames such as lighters and candles.

2.15.7 Sensors and communication

Analog voltage signals from sensors can't be directly interpreted by a computer, so they must be converted to a digital value. Some microcontrollers provide analog-to-digital converters (ADC or A/D) that measure variations in voltage at an input pin and convert it to a digital value. The range of values depends on the resolution of the ADC; common resolutions are 8 and 10 bits. At 8-bit resolution, an ADC can represent 2^8 (256) different values, where 0 volts corresponds to the value 0 and 5 volts corresponds to 255. A 10-bit ADC provides 1024 different values, where 5 volts corresponds to the value 1023.

Data is sent and received between microcontrollers and computers according to established data protocols such as RS-232 serial, USB, MIDI, TPC/IP, Bluetooth, and other proprietary formats like I2C or SPI. Most electronics prototyping kits and microcontrollers include an RS-232 serial port, and this is therefore a convenient way to communicate. This standard has been around for a long time (it was developed in the late 1960s) and it defines signal levels, timing, physical plugs, and information exchange protocols. The physical RS-232 serial port has largely been replaced in computers by the faster and more flexible (but more complex) universal serial bus (USB), but the protocol is still widely used when combining the USB port with software emulation.

Because a device can have several serial ports, a user must specify which serial port to use for data transmission. On most Windows computers, serial port names are `COMx`, where `x` can be 1, 2, 3, etc. On UNIX-based systems (Mac OS X and Linux), serial devices are accessed through files in the `/dev/` directory. After the serial port is selected, the user must specify the settings for the port. Communication speed will vary with devices, but typical values are 9600, 19,200, and 115,200 bits per second. Once the ports are open for communication on both devices, it is possible to send and receive data.

The following examples connect sensors and actuators to a Wiring or Arduino board and communicate the data between the I/O board and a Processing application. When the Wiring and Arduino boards are plugged into a computer's USB port, it appears on the computer as a serial port, making it possible to send/receive data on it. The Wiring board has two serial ports called *Serial* and *Serial1*; the Arduino board has one called Serial. Serial is directly available on the USB connector located on the board surface. *Serial1* is available through the Wiring board digital pin numbers 2(Rx) and 3(Tx) for the user's applications.

2.15.8 Example 1: Switch (Code below)

This example sends the status of a switch (ON or OFF) connected to the Wiring or Arduino board to a Processing application running on a computer. Software runs on the board to read the status of a switch connected on digital pin 4. This value 1 is sent to the serial port continuously while the switch is pressed and 0 is sent continuously when the switch is not pressed. The Processing application continuously receives data from the board and assigns the color of a rectangle on the screen depending on the value of the data. When the switch is pressed the rectangle's color changes from black to light gray.

2.15.9 Example 2: Light sensor (Code below)

This example brings data from a light sensor (photoresistor) connected to the Wiring or Arduino board's analog input pin 0 into a Processing application running on a computer. Software runs on the board to send the value received from the light sensor to the serial port. Because the light sensor is plugged into an analog input pin, the analog voltage coming into the board is converted into a digital number before it is sent over the serial port. The Processing application changes the color of a rectangle on-screen according to the value sent from the board. The rectangle exhibits grays from black to white according to the amount of light received by the sensor. Cover and uncover the sensor with your hand to see a large change.

2.15.10 Controlling physical media

Actuators are devices that act on the physical world. Different types of actuators can create light, motion, heat, and magnetic fields. The digital output pin on a microcontroller can be set to a voltage of 0 or 5 volts. This value can be used to turn a light or motor on or off, but finer control over brightness and speed requires an analog output. By using a digital to analog converter (DAC), a discretized signal can be directly generated as illustrated in the previous figure. If desired, some smoothing can be added to obtain the desired analog signal. When a DAC is not available or not justified in terms of cost or conversion speed, another approach is to use a technique called pulse-width modulation (PWM). This is turning a digital output ON and OFF very quickly to simulate values between 0 and 5 volts. If the output is 0 volts for 90% of the time and 5 volts for 10%, this is called a 10% duty cycle. Following smoothing, it emulates an analog voltage of 0.5 volts. An 80% duty cycle with smoothing emulates a 4-volt signal:

The PWM technique can be used to dim a light, run a motor at a slow speed, and control the frequency of a tone through a speaker. In some applications, any necessary smoothing is obtained for free e.g. the inertia in a motor can average out the PWM duty cycle and result in smooth motion.

Light

Sending current through a light-emitting diode (LED) is the simplest way to get a microcontroller to control light. An LED is a semiconductor device that emits monochromatic light when a current is applied to it. The color (ranging from ultraviolet to infrared) depends on the semiconductor material used in its construction. LEDs have a wide range of applications from simple blinking indicators and displays to street lamps. They have a long life and are very efficient. Some types of LEDs and high-power LEDs require special power arrangements and interfacing circuits before they can be used with microcontrollers. Incandescent, fluorescent, and electroluminescent light sources always require special interfacing circuits before they can be controlled.

Motion

Motors are used to create rotational and linear movement. The rated voltage, the current drawn by the motor, internal resistance, speed, and torque (force) are factors that determine the power and efficiency of the motor. Direct current (DC) motors turn continuously at very high speeds and can switch between a clockwise and counterclockwise direction. They are usually interfaced with a gearbox to reduce the speed and increase the power. Servomotors are modified DC motors that can be set to any position within a 180-degree range. These motors have an internal feedback system to ensure they remain at their position. Stepper motors move in discrete steps in both directions. The size of the steps depends on the resolution of the motor. Solenoids move linearly (forward or back instead of in circles). A solenoid is a coil of wire with a shaft in the center. When current is applied to the coil, it creates a magnetic field that pulls or pushes the shaft, depending on the type. Muscle wire (shape memory alloy or nitinol) is a nickel-titanium alloy that contracts when power is applied. It is difficult to work with and is slower than motors, but requires less current and is smaller. DC and stepper motors need special interfacing circuits because they require more current than a microcontroller can supply through its output pins. H-bridge chips simplify this interface.

Switches

Relays and transistors are used to turn on and off electric current. A relay is an electromechanical switch. It has a coil of wire that generates a magnetic field when an electrical current is passed through. The magnetic field pulls together the two metal contacts of the relay's switch. Solid-state relays without moving parts are faster than electromechanical relays. Using relays makes it possible to turn ON and OFF devices that can't be connected directly to a microcontroller. These devices include home appliances, 120-volt light bulbs, and all other devices that require more power than the microcontroller can provide. Transistors can also behave like switches. Because they operate electronically and not mechanically, they are much faster than relays.

Sound

Running a signal from a digital out or PWM pin to a small speaker is the easiest way to produce a crude, buzzing noise. For more sophisticated sounds, attach these pins to tone-generator circuits created with a 555 timer IC, capacitors, and resistors. Some chips are designed specifically to record and play back sound. Others are sound synthesizers that can synthesize speech by configuring stored phonemes.

Temperature

Temperature can be controlled by a Peltier junction, a device that works as a heat pump. It transforms electricity into heat and cold at the same time by extracting thermal energy from one side (cooling) into the other side (heating). It can also work in reverse, applying heat or cold to the proper surface to produce an electrical current. Because this device consumes more current than a microcontroller can handle in an output pin, it must be interfaced using transistors, relays, or digital switches like the ones described above.

The following examples demonstrate how to control lights and motors attached to an I/O board through a Processing program:

2.15.11 Example 3: Turn a light on and off (Code below)

This example sends data from a Processing program running on a computer to a Wiring or Arduino board to turn a light ON or OFF. The program continually writes an *H* to the serial port if the cursor is inside the rectangle and writes a *L* if it's not. Software running on the board receives the data and checks for the value. If the value is *H*, it turns on a light connected to the digital I/O pin number 4, and if the value is *L*, it turns off the light. The light always reflects the status of the rectangle on the computer's screen.

2.15.12 Example 4: Control a servomotor (Code below)

This example controls the position of a servomotor through an interface within a Processing program. When the mouse is dragged through the interface, it writes the position data to the serial port. Software running on a Wiring or Arduino board receives data from the serial port and sets the position of a servomotor connected to the digital I/O pin number 4.

2.15.13 Example 5: Turn a DC motor on and off (Code below)

This example controls a DC motor from a Processing program. The program displays an interface that responds to a mouse click. When the mouse is clicked within the interface, the program writes data to the serial port. Software running on the board receives data from the serial port and turns the DC motor connected to the PWM pin ON and OFF. The DC motor is connected to the board through an L293D chip to protect the microcontroller from current spikes caused when the motor turns on.

2.15.14 Conclusion

Electronic components and microcontrollers are becoming more common in designed objects and interactive artworks. Although the programming and electronics skills required for many projects call for an advanced understanding of circuits, a number of widely used and highly effective techniques can be implemented and quickly prototyped by novices. The goal of this text is to introduce electronics and to provide enough information to encourage future exploration. As you pursue electronics further, we recommend that you read *CODE* by Charles Petzold to gain a basic understanding of how electronics and computers work, and we recommend that you read *Physical Computing* by Dan O'Sullivan and Tom Igoe for a pragmatic introduction to working with electronics. *Practical Electronics for Inventors* by Paul Scherz is an indispensable resource, and the *Engineer's Mini Notebook* series by Forrest M. Mims III is an excellent source for circuit designs. The Web is a deep resource for learning about electronics, and there are many excellent pages listed below in Resources. The best way to learn is by making projects. Build many simple projects and work through the examples in *Physical Computing* to gain familiarity with the different components.

2.15.15 Code

To run these examples, unlike the other examples in this book, you will need additional equipment. They require either a Wiring (wiring.org.co) or Arduino (www.arduino.cc) board and the following:

1. USB cable (used to send data between board and computer)
2. 9–15V 1000mA power supply or 9V battery
3. 22-gauge solid core wire (get different colors)
4. Breadboard
5. Switch
6. Resistors (10K ohm for the switch circuits, 330 ohm for the LEDs, 1K ohm for the photoresistor)
7. LEDs
8. Servo motor (Futaba or Hi-Tech)
9. DC motor (a generic DC motor like the ones in toy cars)
10. L293D or SN754410 H-Bridge Integrated Circuit
11. Wire cutters
12. Wire strippers
13. Needlenose pliers

This equipment can be purchased from an electronics store such as Radio Shack or from an online vendor.

Each example presents two programs: code for the I/O board and code for Processing. Diagrams and breadboard illustrations for the examples are presented side by side in this tutorial to reinforce the connections between the two representations. Learning to translate a circuit diagram into a physical circuit is one of the most difficult challenges when starting to work with electronics.

The Wiring or Arduino software environment is necessary to program each board. These environments are built on top of the Processing environment, but they have special features for uploading code to the board and monitoring serial communication. Both can be downloaded at no cost from their respective websites and both are available for Linux, Macintosh, and Windows.

The pySerial library is also required to communicate with the Arduino/Wiring boards with p5. Refer to the installation instructions on the pySerial documentation

The examples that follow assume the I/O board is connected to your computer and serial communication is working. Before working with these examples, get one of the simple pySerial library examples to work. For the most up-to-date information and troubleshooting tips, refer to the pySerial documentation. The Wiring and Arduino websites have additional information.

Example 1A: Switch (Wiring/Arduino)

```
// Code for sensing a switch status and writing the value to the␣
↪serial port

int switchPin = 4;  // Switch connected to pin 4

void setup() {
  pinMode(switchPin, INPUT);  // Set pin 0 as an input
  Serial.begin(9600);         // Start serial communication at 9600␣
↪bps
}

void loop() {
  if (digitalRead(switchPin) == HIGH) {  // If switch is ON,
    Serial.write(1);                     // send 1 to Processing
  } else {                               // If the switch is not ON,
    Serial.write(0);                     // send 0 to Processing
  }
  delay(100);                            // Wait 100 milliseconds
}
```

Example 1B: Switch (p5)

```
from p5 import *
from serial import Serial

# Create object from Serial class. Here wer open the port that the
# board is connected to and use the same speed (9600 bps).
port = Serial('/dev/ttyUSB0', 9600)

def setup():
    size(200, 200)

def draw():
    # read one byte of raw data from the serial port
    raw_data = port.read()

    # next convert the data (sent as a single byte) into an integer
    data = int.from_bytes(raw_data, byteorder='little', signed=True)
    # print(data)

    if (data == 0):
        fill(0)
    else:
        fill(204)

    rect((50, 50), 100, 100)
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    run(frame_rate=10)
```

Example 2A: Light sensor (Wiring/Arduino)

```
// Code to read an analog value and write it to the serial port

int val;
int inputPin = 0;                // Set the input to analog in pin 0

void setup() {
  Serial.begin(9600);            // Start serial communication at␣
↪9600 bps
}

void loop() {
  val = analogRead(inputPin)/4;  // Read analog input pin, put in␣
↪range 0 to 255
  Serial.write(val);             // Send the value
  delay(100);                    // Wait 100ms for next reading
}
```

Example 2B: Light sensor (p5)

```
from p5 import *
from serial import Serial

# Create object from Serial class. Here wer open the port that the
# board is connected to and use the same speed (9600 bps).
port = Serial('/dev/ttyUSB0', 9600)

def setup():
    size(200, 200)
    no_stroke()

def draw():
    # read one byte of raw data from the serial port
    raw_data = port.read()

    # next convert the data (sent as a single byte) into an integer
    data = int.from_bytes(raw_data, byteorder='little', signed=True)
    # print(data)
```

(continues on next page)

(continued from previous page)

```
    # set fill color to the value just read.
    fill(data)
    rect((50, 50), 100, 100)

if __name__ == '__main__':
    run(frame_rate=10)
```

Example 3A: Turning a light on and off

```
// Read data from the serial and turn ON or OFF a light depending␣
↪on the value

char val;                          // Data received from the serial␣
↪port
int ledPin = 4;                    // Set the pin to digital I/O 4

void setup() {
  pinMode(ledPin, OUTPUT);         // Set pin as OUTPUT
  Serial.begin(9600);              // Start serial communication at␣
↪9600 bps
}

void loop() {
  if (Serial.available()) {        // If data is available to read,
    val = Serial.read();           // read it and store it in val
  }
  if (val == 'H') {                // If H was received
    digitalWrite(ledPin, HIGH);    // turn the LED on
  } else {
    digitalWrite(ledPin, LOW);     // Otherwise turn it OFF
  }
  delay(100);                      // Wait 100 milliseconds for␣
↪next reading
}
```

Example 3B: Turning a light on and off (p5)

```
from p5 import *
from serial import Serial

# Create object from Serial class. Here we open the port that the
# board is connected to and use the same speed (9600 bps).
port = Serial('/dev/ttyUSB0', 9600)
```

(continues on next page)

(continued from previous page)

```
def setup():
    size(200, 200)

def draw():
    background(255)
    if mouse_over_rect():
        # if the mouse is over the rectangle, then change fill color
        # and send an H
        fill(204)
        port.write(b'H')
    else:
        # otherwise, change color and send an L
        fill(0)
        port.write(b'L')

    # draw a square
    square((50, 50), 100)

def mouse_over_rect():
    """Test if the mouse is over a square.
    """
    correct_x = (mouse_x >= 50) and (mouse_x <= 150)
    correct_y = (mouse_y >= 50) and (mouse_y <= 150)
    return correct_x and correct_y

if __name__ == '__main__':
    # run at 10 frames per second
    run(frame_rate=10)
```

Example 4A: Controlling a servomotor(Wiring/Arduino)

```
// Read data from the serial port and set the position of a␣
↪servomotor
// according to the value

Servo myservo;                   // Create servo object to control␣
↪a servo
int servoPin = 4;                // Connect yellow servo wire to␣
↪digital I/O pin 4
int val = 0;                     // Data received from the serial␣
↪port

void setup() {
  myservo.attach(servoPin);      // Attach the servo to the PWM pin
  Serial.begin(9600);             // Start serial communication at␣
↪9600 bps
```

(continues on next page)

(continued from previous page)

```
}

void loop() {
  if (Serial.available()) {       // If data is available to read,
    val = Serial.read();          // read it and store it in val
  }
  myservo.write(val);             // Set the servo position
  delay(15);                      // Wait for the servo to get there
}
```

Example 4B: Controlling a servomotor (p5)

```
from p5 import *
from serial import Serial

# Create and initialize the port that the board is connected to and
# use the same speed (9600 bps)
port = Serial('/dev/ttyUSB0', 9600)

mx = 0

def setup():
    size(200, 200)
    no_stroke()

def draw():
    global mx

    # clear background, set fill color
    background(0)
    fill(204)

    rect((40, height / 2 - 15), 120, 25)

    dif = mouse_x - mx
    if (abs(dif) > 1):
        mx = mx + (dif / 4.0)

    # keeps marker on the screen
    mx = constrain(mx, 50, 149)

    no_stroke()
    fill(255)
    rect((50, (height / 2) - 5), 100, 5)
    fill(204, 102, 0)

    # draw the position
```

(continues on next page)

(continued from previous page)

```
    rect((mx - 2, height / 2 - 5), 4, 5)

    # scale the value to the range 0 to 180
    angle = int(remap(mx, (50, 149), (0, 180)))

    # print the current angle (debug)
    # print(angle)

    # write out to the port (note that we first need to convert the
    # data to bytes)
    port.write(bytes([angle]))

if __name__ == '__main__':
    run(frame_rate=10)
```

Example 5A: Turning a DC Motor on and off (Wiring/Arduino)

```
// Read data from the serial and turn a DC motor on or off␣
↪according to the value

char val;              // Data received from the serial port
int motorpin = 0;      // Wiring: Connect L293D Pin En1 connected to␣
↪Pin PWM 0
// int motorpin = 9;   // Arduino: Connect L293D Pin En1 to Pin PWM 9

void setup() {
  Serial.begin(9600);                  // Start serial␣
↪communication at 9600 bps
}

void loop() {
  if (Serial.available()) {            // If data is available,
    val = Serial.read();               // read it and store it in␣
↪val*
  }
  if (val == 'H') {                    // If 'H' was received,
    analogWrite(motorpin, 125);        // turn the motor on at␣
↪medium speed
  } else {                                     // If 'H' was not␣
↪received
    analogWrite(motorpin, 0);          // turn the motor off
  }
  delay(100);                          // Wait 100 milliseconds for␣
↪next reading
}
```

Example 5B: Turning a DC Motor on and off (p5)

```
# Write data to the serial port according to the status of a button
# controlled by the mouse

from p5 import *
from serial import Serial

# Create object from Serial class. Here we open the port that the
# board is connected to and use the same speed (9600 bps).
port = Serial('/dev/ttyUSB0', 9600)

rect_over = False

# position, diameter, and color of the button
rect_location = None
rect_size = 100
rect_color = Color(100)

# status of the button
button_on = False

def mouse_over_rect(location, dimensions):
    w, h = dimensions
    x, y = location.x, location.y

    correct_x = (mouse_x >= x) and (mouse_x <= (x + w))
    correct_y = (mouse_y >= y) and (mouse_y <= (y + h))

    return correct_x and correct_y

def setup():
    global rect_location
    size(200, 200)
    no_stroke()
    rect_location = Vector((width / 2) - (rect_size / 2),
                           (height / 2) - (rect_size / 2))

def draw():
    global rect_over
    rect_over = mouse_over_rect(rect_location, (rect_size, rect_
↪size))

    # clear background to black
    background(0)

    fill(rect_color)
    square(rect_location, rect_size)

def mouse_released():
```

(continues on next page)

(continued from previous page)

```
    global rect_color
    global button_on

    if rect_over:
        if button_on:
            rect_color = Color(100)
            button_on = False

            # send an L to indicate button is OFF
            port.write(b'L')
        else:
            rect_color = Color(180)
            button_on = True

            # send an H to indicate button is ON
            port.write(b'H')

if __name__ == '__main__':
    # run at 10 frames per second
    run(frame_rate=10)
```

2.16 Network

Authors Alexander R. Galloway; Abhik Pal (p5 port)

Copyright This tutorial is the “Extension 2” chapter from Processing: A Programming Handbook for Visual Designers and Artists, Second Edition, published by MIT Press. © 2014 MIT Press. If you see any errors or have comments, please let us know. The tutorial was ported to p5 by Arihant Parsoya. If you see any errors or have comments, open an issue on either the p5 or Processing repositories.

Networks are complex organizational forms. They bring into association discrete entities or nodes, allowing these nodes to connect to other nodes and indeed to other networks. Networks exist in the world in a vast variety of forms and in even more contexts: political, social, biological, and otherwise. While artists have used networks in many ways–from postal networks used to disseminate work, to informal networks of artistic collaborators and larger aesthetic movements–this section looks specifically at a single instance of network technology, the Internet, and how artists have incorporated this technology into their work. There are two general trends: art-making where the Internet is used as a tool for quick and easy dissemination of the work, and art–making where the Internet is the actual medium of the work. These two trends are not mutually exclusive, however. Some of the most interesting online work weaves the two techniques together into exciting new forms that surpass the affordances of either technique.

2.16.1 The Internet and the arts

“In December 1995, Vuk Cosic got a message . . . ” or so begins the tale of how “net.art,” the niche art movement concerned with making art in and of the Internet, got its start and its name. As Alexei Shulgin explains in a posting to the Nettime email list two years later, Cosic, a Slovenian artist, received an email posted from an anonymous mailer. Apparently mangled in transit, the message was barely legible. “The only fragment of it that made any sense looked something like: [. . .] J8~g#|;Net. Art{-^s1 [. . .].”1

Anonymous, accidental, glitchy, and slightly apocryphal–these are all distinctive characteristics of the net.art style, as seen in web-based work from Cosic, Shulgin, Olia Lialina, Jodi, Heath Bunting, and many others. As Marina Grzinic writes, the “delays in transmission–time, busy signals from service providers, [and] crashing web browsers” contributed greatly to the way artists envisioned the aesthetic potential of the web, a tendency that ran counter to the prevailing wisdom at the time of dot–com go, go, go.2 Indeed many unsuspecting users assume that Jodi’s web-based and downloadable software projects have as their primary goal the immediate infection and ruin of one’s personal computer. (Upon greater scrutiny, it must be granted that this is only a secondary goal.) Perhaps peaking in 1998 with the absurd, anarchist experiments shoveled out on the 7–11 email list–spoofs and shenanigans were par for the course due to the fact that the list administration tool, including subscriptions, header and footer variables, and moderation rules, was world read-writable by any web surfer—the net.art movement is today viewable in captivity in such catchall publications as the hundred–contributor–strong anthology Readme!, edited in several cities simultaneously and published by Autonomedia in 1999; the equally ecumenical anthology NTNTNT that emerged from CalArts in 2003; or Tilman Baumgärtel’s two volumes of interviews, net.art (1999) and net.art 2.0 (2001).

At the same time, buoyed by the dynamism of programming environments like Java and Flash, artists and designers began making online work that not only was “in and of” the Internet, but leveraged the net as a tool for quick and easy dissemination of executable code, both browser-based and otherwise. John Maeda created a number of sketches and games dating from the mid–1990s, including a series of interactive calendars using visual motifs borrowed from both nature and mathematics. Joshua Davis also emerged as an important figure through his online works Praystation and Once–Upon–A–Forest. Like Maeda, Davis fused algorithmic drawing techniques with an organic sense of composition.

It is worth recalling the profound sense of optimism and liberation that the web brought to art and culture in the mid–1990s. All of a sudden, tattooed skaters like Davis were being shuttled around the globe to speak to bespectacled venture capitalists about the possibilities of networking, generative code, and open software projects. And bespectacled philosophers like Geert Lovink were being shuttled around the globe to speak to tattooed skaters about–what else—the possibilities of networking, generative code, and open software projects. Everything was topsy–turvy. Even the net.art movement, which was in part influenced by Lovink’s suspicion of all things “wired” and Californian, was nonetheless propelled by the utopian promise of networks, no matter that sometimes those networks had to be slashed and hacked in the process. Networks have, for several decades, acted as tonics for and inoculations against all things centralized and authoritarian, be they Paul Baran’s 1964 schematics for routing around both the AT&T/Bell national telephone network and the then-impending sorties of Soviet ICBMs; or the grassroots networks of the 1960s new social movements, which would later gain the status of art in Deleuze and Guattari’s emblematic literary concept of the “rhizome,” quite literally a grassroots model of networked being; or indeed the much earlier and oft–cited remarks from Bertolt Brecht on the early revolutionary potential of radio networks (reprised, famously, in Hans Magnus Enzensberger's 1974 essay on new media, "Constituents for a Theory of the Media"). In other words, the arrival of the web in the middle to late 1990s generated much excitement in both art and culture, for it seemed like a harbinger for the coming of some new and quite possibly revolutionary mode of social interaction. Or, as Cosic said once, with typical bravado, "all art to now has been merely a substitute for the Internet."

It is also helpful to contextualize these remarks through reference to the different software practices of various artists and movements. Each software environment is a distinct medium. Each grants particular aesthetic affordances to the artist and diminishes others. Each comes with a number of side effects that may be accentuated or avoided, given the proclivities of the artist. So, while acknowledging that digital artists' tools and materials tend to vary widely, it is perhaps helpful to observe that the net.art scene (Bunting, Shulgin, Lialina, Jodi, et al.), particularly during the period 1995–2000, coded primarily in browser–based markup languages such as HTML, with the addition of Javascript for the execution of basic algorithms. A stripped-down, "text only" format was distinctive of this period. One gag used by a number of different artists was not to have a proper homepage at all, but instead to use Apache's default directory index of files and folders. The stripped-down approach did not always deliver simplicity to the user, however, as in the case of Jodi's early homepage (now archived at http://wwwwwwwww.jodi.org) in which they neglected a crucial pre tag and then, in an apparent attempt to overcompensate for the first glitch, encircled the page in a blink tag that was no less prickly on the eyes than the missing pre tag is disorienting. The blinking page throbbed obscenely in the browser window, one glitch thus compounding the other. Created as an unauthorized addition to HTML by Netscape Navigator, the blink tag essentially vanished from the Internet as Internet Explorer became more dominant in the late 1990s. Today, the Jodi page doesn't blink. One wonders which exactly is the work: the op-art strobe effect that appeared in the Netscape browser window during the years when people used Netscape, or the HTML source still online today in which the work is "explained" to any sleuth willing to trace the narrative of markup tags missing and misplaced?

While artists had used fixed–width ASCII fonts and ANSI characters as design elements long before the popularization of the web in the mid-1990s, it was the creation of HTML in 1993 (synchronized with its use in the newly invented web servers and web browsers like Netscape) that transformed the Internet into a space for the visual arts. HTML established itself quickly as the most influential mark–up language and graphic design protocol for two reasons: first, the text-only nature of HTML made it low–bandwidth–friendly during a time when most users connected via modems and phone lines; and second, HTML is a protocol, meaning that it acts as a common denominator (the blink tag notwithstanding) bridging a wide variety of dissimilar technical platforms. But, as seen in the work of Davis, which gravitates toward Flash but also includes web, print, and video, one must not overemphasize HTML as an aesthetic medium. During this same period, the network delivery of executable code (Java applets, Flash, Shockwave, and so on) also became more and more exciting as a medium for art–making, as did CUSeeMe, web radio, video, and other streaming content that operates outside of the normal bounds of the browser frame. John Simon's 1997 Every Icon was written as a Java applet and therefore easily deliverable online as executable code. In what Lev Manovich has dubbed "Generation Flash," a whole new community sprang up, involving artists like Yugo Nakamura, Matt Owens, and James Paterson and intersecting with both dot–com startups like i|o 360° and Razorfish (or the artist's own design shops) and indie youth culture. Their medium is not solely the text-only markup codes of HTML but also the more sophisticated Macromedia languages (ActionScript and Lingo) as well as Javascript, Java, and server-side languages like Perl and PHP.

2.16.2 Internet concepts

In order to understand how online works are made and viewed, it will be useful to address a number of key concepts in the area of computer networking. A computer network consists of two or more machines connected via a data link. If a networked computer acts primarily as a source of data, it is called a server. A server typically has a fixed address, is online continuously, and functions as a repository for files which are transmitted back to any other computers on the network that request them. If a networked computer acts primarily as a solicitor of information, it is called a client. For example, in checking one's email, one acts as a client. Likewise, the machine where the email is stored (the machine named after the @ sign in the email address) acts as a server. These terms are flexible; a machine may act as a server in one context and a client in another.

Any machine connected to the Internet, be it client or server, is obligated to have an address. On the Internet, these addresses are called IP addresses and come in the form 123.45.67.89. (A new addressing standard is currently being rolled out that makes the addresses slightly longer.) Since IP addresses change from time to time and are difficult to remember, a system of natural–language shortcuts called the Domain Name System (DNS) allows IP addresses to be substituted by domain names such as "processing.org" or "google.com." In a web address the word immediately preceding the domain name is the host name; for web servers it is customary to name the host machine "www" after the World Wide Web. But this is only customary. In fact, a web server's host name can be most anything at all.

One of the main ways in which visual artists have used the Internet in their work is to conceive of the network as one giant database, an input stream that may be spidered, scanned, and parsed using automated clients. This is an artistic methodology that acknowledges the fundamental mutability of data (what programmers call "casting" a variable from one data type to another) and uses various data sources as input streams to power animations, to populate arrays of numbers with pseudo–random values, to track behavior, or quite simply for "content." Lisa Jevbratt's work 1:1 does this through the premise that every IP address might be represented by a single pixel. Her work scans the IP address namespace, number by number, pinging each address to determine whether a machine is online at that location. The results are visualized as pixels in a gigantic bitmap that, quite literally, represents the entire Internet (or at least all those machines with fixed IP addresses). In a very different way, Mark Napier's two works Shredder and Digital Landfill rely on a seemingly endless influx of online data, rearranging and overlaying source material in ways unintended by the original creators. Works like urllib2 and requests approach the network itself as a data source, the former tapping into real-time web traffic, and the latter tapping into real–time traffic on the Gnutella peer–to–peer network. Earlier works such as >I/O/D 4 (known as "The Webstalker"), or Jodi's Wrongbrowser series of alternative web browsers also illustrate this approach, that the network itself is the art. All of these works automate the process of grabbing data from the Internet and manipulating it in some way. One of the most common types is a web client, a piece of software that automates the process of requesting and receiving remote files on the World Wide Web.

2.16.3 Example 1: Web client

Python's requests library includes ready–made classes for both servers and clients. In order to fetch a page from the web, first one creates a client and connects to the address of the remote server. Using a simple call–and–response technique, the client requests the file and the file is returned by the server. This call–and-response is defined by a protocol called Hypertext Transfer Protocol (HTTP). HTTP consists of a handful of simple commands that are used to describe the state of the server and client, to request files, and to post data back to the server if necessary. The most basic HTTP command is GET. This command is similar to filling out a book request form in a library: the client requests a file by name, the server "gets" that file, and returns it to the client. HTTP also includes a number of response codes to indicate that the file was found successfully, or to indicate if any errors were encountered (for example, if the requested file doesn't exist). The command GET/HTTP/1.0n means that the client is requesting the default file in the root web directory (/) and that the client is able to communicate using HTTP version 1.0. The trailing n is the newline character, or roughly equivalent to hitting the return key. If the default file exists, the server transmits it back to the client.

While most computers have only a single Ethernet port (or wireless connection), the entire connectivity of each machine is able to sustain more connections than a single input or output, because the concept of a port is abstracted into software and the functionality of the port is thus duplicated many times over. In this way, each networked computer is able to multitask its single network connection across scores of different connections (there are 1,024 well-known ports, and 65,535 ports in all). Thus ports allow a networked computer to communicate simultaneously on a large number of "channels" without blocking other channels or impeding the data flow of applications. For example, it is possible to read email and surf the web simultaneously because email arrives through one port while websites use another. The union of IP address and port number (example: 123.45.67.89:80) is called a socket. Socket connections are the bread and butter of networking.

```python
import requests
from p5 import *

def setup():
    size(200, 200)
    c = requests.get('http://www.ucla.edu', auth=('user', 'pass'))
    print(c.text)

run()
```

2.16.4 Internet protocols

A protocol is a technological standard. The Internet protocols are a series of documents that describe how to implement standard Internet technologies such as data routing, handshaking between two machines, network addressing, and many other technologies. Two protocols have already been discussed–HTML, which is the language protocol for hypertext layout and design; and HTTP, which is the protocol for accessing web-accessible files—but there are a few other protocols worth discussing in this context.

Protocols are abstract concepts, but they are also quite material and manifest themselves in the form of structured data headers that prepend and encapsulate all content traveling through the Internet. For example, in order for a typical HTML page to travel from server to client, the page is prepended by an HTTP header (a few lines of text similar to the GET command referenced previously). This glob of data is itself prepended by two additional headers, first a Transmission Control Protocol (TCP) header and next by an Internet Protocol (IP) header. Upon arrival at its destination, the message is unwrapped: the IP header is removed, followed by the TCP header, and finally the HTTP header is removed to reveal the original HTML page. All of this is done in the blink of an eye. All headers contain useful information about the packet. But perhaps the four most useful pieces of information are the sender IP address, receiver IP address, sender port, and receiver port. These four pieces are significant because they indicate the network addresses of the machines in question, plus, via a reverse lookup of the port numbers, the type of data being transferred (port 80 indicating web data, port 23 indicating a Telnet connection, and so on). See the /etc/services file on any Macintosh, Linux, or UNIX machine, or browse IANA's registry for a complete listing of port numbers. The addresses are contained in the IP header from byte 12 to byte 29 (counting from 0), while the ports are contained in bytes zero through three of the TCP header.

The two elements of the socket connection (IP address and port) are separated into two different protocols because of the different nature of IP and TCP. The IP protocol is concerned with routing data from one place to another and hence requires having an IP address in order to route correctly but cares little about the type of data in its payload. TCP is concerned with establishing a virtual circuit between server and client and thus requires slightly more information about the type of Internet communication being attempted. IP and TCP work so closely together that they are often described in one breath as the "TCP/IP suite."

While most data on the Internet relies on the TCP/IP suite to get around, certain forms of networked communication are better suited to the UDP/IP combination. User Datagram Protocol (UDP) has a much leaner implementation than TCP, and while it therefore sacrifices many of the features of TCP, it is nevertheless useful for stateless data connections and connections that require a high throughput of packets per second, such as online games.

2.16.5 Network tools

There are a number of existing network tools that a programmer may use beyond the Processing environment. Carnivore and tcpdump, two different types of packet sniffers that allow one to receive LAN packets in real time, have already been mentioned. The process of scanning networks for available hosts, called network discovery, is also possible using port scanner tools such as Nmap. These tools use a variety of methods for looping through a numerical set of IP addresses (example: 192.168.1.x where x is incremented from 0 to 255), testing to see if a machine responds at that address. Then, if a machine is known to be online, the port scanner is used to loop through a range of ports on the machine (example: 192.168.1.1:x where x is a port number incremented from 1 to 1024) in order to determine which ports are open, thereby determining which application services are available. Port scans can also be used to obtain "fingerprints" for remote machines, which aid in the identification of the machine's current operating system, type, and version information for known application services.

Perhaps the most significant advance in popular networking since the emergence of the web in the mid-1990s was the development of peer-to-peer systems like Gnutella or BitTorrent. Coming on the heels of Napster, Gnutella fully distributed the process of file sharing and transfer, but also fully distributed the network's search algorithm, a detail that had created bottlenecks (not to mention legal liabilities) for the more centralized Napster. With a distributed search algorithm, search queries hopscotch from node to node, just like the "hot potato" method used in IP routing; they do not pass through any centralized server. The Gnutella protocol has been implemented in dozens of peer-to-peer applications.

Yet more recently Gnutella has been superseded by systems like BitTorrent, a peer–to–peer application that allows file transfers to happen simultaneously between large numbers of users. BitTorrent has gained wide popularity for its increased efficiencies, particularly for transfers of large files such as video and software.

Many software projects requiring networked audio have come to rely on the Open Sound Control (OSC) protocol. OSC is a protocol for communication between multimedia devices such as computers and synthesizers. OSC has been integrated into SuperCollider and Max/MSP and has been ported to most modern languages including Perl and Java. Andreas Schlegel's "oscP5" is an OSC extension library for Processing.

The Internet has become more peer-oriented in other ways too, not simply through file sharing but also through friend communities and other kinds of social interaction. Web 2.0 platforms like Facebook, Twitter, or YouTube allow groups of people to network with each other, whether it be a small group of five or ten, or a larger group of several thousands. While the more traditional client-server model relied on a clear distinction between information providers and information consumers, newer systems have disrupted some of the old hierarchies, allowing networked users to act as both producers and consumers of content. Such systems thus rely heavily on user-generated content and expect a higher amount of participation and interaction from their user base.

2.16.6 Conclusion

Programmers are often required to consider interconnections between webs of objects and events. Because of this, programming for networks is a natural extension of programming for a single machine. Classes send messages to other classes just like hosts send messages to other hosts. An object has an interface, and so does an Ethernet adapter. The algorithmic construction of entities in dialogue–pixels, bits, frames, nodes–is central to what Processing is all about. Networking these entities by moving some of them to one machine and some to another is but a small additional step. What is required, however, is a knowledge of the various standards and techniques at play when bona fide networking takes place.

Historically, there have been two basic strands of networked art: art where the network is used as the actual medium of art–making, or art where the network is used as the transportation medium for dissemination of the work. The former might be understood as art of the Internet, while the latter as art for the Internet. The goal of this text has been to introduce some of the basic conditions, both technological and aesthetic, for making networked art, in the hopes that entirely new techniques and approaches will spring forth in the future as both strands blend together into exciting new forms.

2.17 Vector

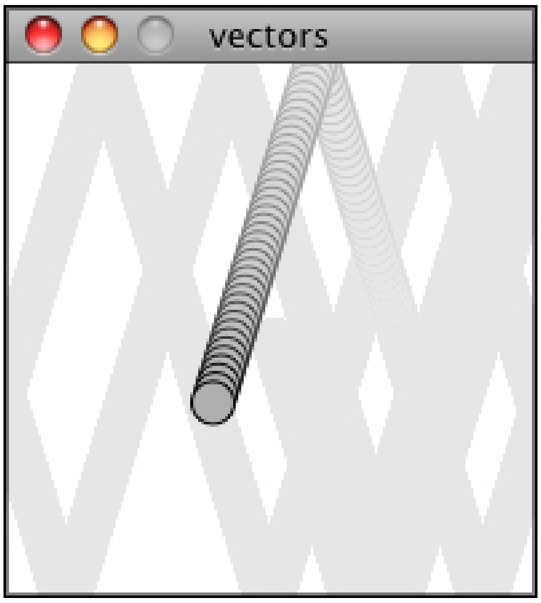
Authors Daniel Shiffman; Abhik Pal (p5 port)

The most basic building block for programming motion is the **vector**. And so this is where we begin. Now, the word **vector** can mean a lot of different things. Vector is the name of a new wave rock band formed in Sacramento, CA in the early 1980s. It's the name of a breakfast cereal manufactured by Kellogg's Canada. In the field of epidemiology, a vector is used to describe an organism that transmits infection from one host to another. In the C++ programming language, a Vector (`std::vector`) is an implementation of a dynamically resizable array data structure. While all interesting, these are not the definitions we are looking for. Rather, what we want is this vector:

A vector is a collection of values that describe relative position in space.

2.17.1 Vectors: You Complete Me

Before we get into vectors themselves, let's look at a beginner Processing example that demonstrates why it is in the first place we should care. If you've read any of the introductory Processing textbooks or taken a class on programming with Processing (and hopefully you've done one of these things to help prepare you for this book), you probably, at one point or another, learned how to write a simple bouncing ball sketch.



```
from p5 import *
x = 100
y = 100

xspeed = 1
yspeed = 3.3

def setup():
    size(200, 20
    background(2

def draw():
    global x
    global y
    global xspee
    global yspee
```

(continues on next page)

(continued from previous page)

```
        no_stroke()
        fill(255, 10
        rect((0, 0),

        # add␣
↪the current sp
        x = x + xspe
        y = y + yspe

        if x > width
            xspeed =

        if y > heigh
            yspeed =

        stroke(0)
        fill(175)
        circle((x, y

    if __name__ == '
        run()
```

In the above example, we have a very simple world – a blank canvas with a circular shape ("ball") traveling around. This "ball" has some properties.

- LOCATION: `x` and `y`
- SPEED: `xspeed` and `yspeed`

In a more advanced sketch, we could imagine this ball and world having many more properties:

- ACCELERATION: `xacceleration` and `yacceleration`
- TARGET LOCATION: `xtarget` and `ytarget`
- WIND: `xwind` and `ywind`
- FRICTION: `xfriction` and `yfriction`

It's becoming more and more clear that for every singular concept in this world (wind, location, acceleration, etc.), we need two variables. And this is only a two-dimensional world, in a 3D world, we'd need `x`, `y`, `z`, `xspeed`, `yspeed`, `zspeed`, etc. Our first goal in this chapter is learn the fundamental concepts behind using vectors and rewrite this bouncing ball example. After all, wouldn't it be nice if we could simple write our code like the following?

Instead of:

```
x = ...
y = ...
xspeed = ...
yspeed = ...
```

Wouldn't it be nice to have...

```
location = Vector(...)
speed = Vector(...)
```

Vectors aren't going to allow us to do anything new. Using vectors won't suddenly make your Processing sketches magically simulate physics, however, they will simplify your code and provide a set of functions for common mathematical operations that happen over and over and over again while programming motion.

As an introduction to vectors, we're going to live in 2 dimensions for quite some time (at least until we get through the first several chapters.) All of these examples can be fairly easily extended to three dimensions (and the class we will use – `p5.Vector` – allows for three dimensions.) However, for the time being, it's easier to start with just two.

2.17.2 Vectors: What are they to us, the Processing programmer?

Technically speaking, the definition of a vector is the difference between two points. Consider how you might go about providing instructions to walk from one point to another.

Here are some vectors and possible translations:

You've probably done this before when programming motion. For every frame of animation (i.e. single cycle through Processing's `p5.draw()` loop), you instruct each object on the screen to move a certain number of pixels horizontally and a certain number of pixels (vertically).

For a Processing programmer, we can now understand a vector as the instructions for moving a shape from point A to point B, an object's "pixel velocity" so to speak. For every frame:

$$location = location + velocity$$

If velocity is a vector (the difference between two points), what is location? Is it a vector too? Technically, one might argue that location is not a vector, it's not describing the change between two points, it's simply describing a singular point in space – a location. And so conceptually, we think of a location as different: a single point rather than the difference between two points.

Nevertheless, another way to describe a location is as the path taken from the origin to reach that location. Hence, a location can be represented as the vector giving the difference between location and origin. Therefore, if we were to write code to describe a vector object, instead of creating separate Point and Vector classes, we can use a single class which is more convenient.

Let's examine the underlying data for both location and velocity. In the bouncing ball example we had the following:

$$location \to \texttt{x}, \texttt{y}$$
$$velocity \to \texttt{xspeed}, \texttt{yspeed}$$

Notice how we are storing the same data for both – two numbers, an `x` and a `y`. If we were to write a vector class ourselves, we'd start with something rather basic:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

At its core, a *p5.Vector()* is just a convenient way to store two values (or three, as we'll see in 3D examples.).

And so this. . .

```
x = 100
y = 100
xspeed = 1
yspeed = 3.3
```

. . . becomes. . .

```
location = Vector(100, 100)
velocity = Vector(1, 3.3)
```

Now that we have two vector objects (`location` and `velocity`), we're ready to implement the algorithm for motion – `location = location + velocity`. In the bouncing ball example, without vectors, we had:

```
# add the current speed to the location
x = x + xspeed
y = y + yspeed
```

By default Python's + operator works on primitive values, however we can teach Python to add two vectors together using the + operator. The *p5.Vector* class is implemented with functions for common mathematical operations using the usual operators(+ for addition, * for multiplication, etc) These allow us to rewrite the above as:

```
# add the current speed to the location
location = location + velocity
```

2.17.3 Vectors: Addition

Before we continue looking at the *p5.Vector* class and its *p5.Vector.__add__()* method (purely for the sake of learning since it's already implemented for us in Processing itself), let's examine vector addition using the notation found in math/physics textbooks.

Vectors are typically written as with either boldface type or with an arrow on top. For the purposes of this tutorial, to distinguish a **vector** from a **scalar** (scalar refers to a single value, such as integer or floating point), we'll use an arrow on top:

Vector: $\vec{v}$

Scalar: x

Let's say I have the following two vectors:

$$\vec{u} = \begin{pmatrix} 5 \\ 2 \end{pmatrix}, \qquad \vec{v} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$$

Each vector has two components, an x and a y. To add two vectors together we simply add both x 's and both y 's. In other words:

$$\vec{w} = \vec{u} + \vec{v}$$

translates to:

$$\begin{aligned} w_x &= u_x + v_x \\ w_y &= u_y + v_y \end{aligned}$$

and therefore

$$\vec{w} = \begin{pmatrix} 8 \\ 6 \end{pmatrix}$$

Now that we understand how to add two vectors together, we can look at how addition is implemented in the *p5.Vector* class itself. Let's write a function called `__add__` that takes as its argument another *p5.Vector* object.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # New! A function to add another Vector to this vector. Simply
    # add the x components and the y components together.
    def __add__(self, v):
        self.x = self.x + v.x
        self.y = self.y + v.y
        return self
```

Now that we can how `__add__` is written inside of *p5.Vector*, we can return to the `location + velocity` algorithm with our bouncing ball example and implement vector addition:

```
# add the current speed to the location
location = location + velocity
```

And here we are, ready to successfully complete our first goal – rewrite the entire bouncing ball example using `p5.Vector`.

```
from p5 import *

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, v):
        self.x = self.x + v.x
        self.y = self.y + v.y
        return self

location = Vector(100, 100)
velocity = Vector(1, 3.3)

def setup():
    size(200, 200)
    background(255)

def draw():
    global location
    global velocity

    no_stroke()
    fill(255, 10)
    rect((0, 0), width, height)

    # add the current speed to the location
    location = location + velocity

    # We still sometimes need to refer to the individual
    # components of a Vector and can do so using the dot syntax
    # (location.x, velocity.y, etc)
    if location.x > width or location.x < 0:
        velocity.x = -velocity.x

    if location.y > height or location.y < 0:
        velocity.y = -velocity.y

    # display circle at x location
    stroke(0)
    fill(175)
    circle((location.x, location.y), 16)
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    run()
```

Now, you might feel somewhat disappointed. After all, this may initially appear to have made the code more complicated than the original version. While this is a perfectly reasonable and valid critique, it's important to understand that we haven't fully realized the power of programming with vectors just yet. Looking at a simple bouncing ball and only implementing vector addition is just the first step. As we move forward into looking at more a complex world of multiple objects and multiple forces (we'll cover forces in the next chapter), the benefits of `p5.Vector` will become more apparent.

We should, however, make note of an important aspect of the above transition to programming with vectors. Even though we are using `Vector` objects to describe two values – the x and y of location and the x and y of velocity – we still often need to refer to the x and y components of each `Vector` individually. When we go to drawn an object, there is no means for us to say (using our own `Vector` class):

```
circle(location, 16)
```

The `p5.circle()` function does not understand the `Vector` class we've just written. However this functionality has been implemented in p5's `p5.Vector` class. For our own class, we must dig into the `Vector` object and pull out the x and the y components using object oriented syntax.

```
circle((location.x, location.y), 16)
```

The same issue arises when it comes time to test if the circle has reached the edge of the window, and we need to access the individual components of both vectors: location and velocity.

```
if location.x > width or location.x < 0:
        velocity.x = -velocity.x
```

2.17.4 Vectors: More Algebra

Addition was really just the first step. There is a long list of common mathematical operations that are used with vectors when programming the motion of objects on the screen. Following is a comprehensive list of all of the mathematical operations available as functions in the `p5.Vector` class. We'll then go through a few of the key ones now. As our examples get more and more sophisticated we'll continue to reveal the details of these functions.

- `u + v` – add vectors
- `u - v` – subtract vectors
- `k * u` – scale the vector with multiplication
- `u / k` – scale the vector with division
- `p5.Vector.magnitude()` – calculate the magnitude of a vector

- *p5.Vector.normalize()* – normalize the vector to unit length of 1
- *p5.Vector.limit()* – limit the magnitude of a vector
- *p5.Vector.angle()* – the heading of a vector expressed as an angle
- *p5.Vector.distance()* – the euclidean distance between two vectors (considered as points)
- *p5.Vector.angle_between()* – find the angle between two vectors
- *p5.Vector.dot()* – the dot product of two vectors
- `pt.Vector.cross()` – the cross product of two vectors

Having already run through addition, let's start with subtraction. This one's not so bad, just take the plus sign from addition and replace it with a minus!

Vector subtraction:

$$\vec{w} = \vec{u} - \vec{v}$$

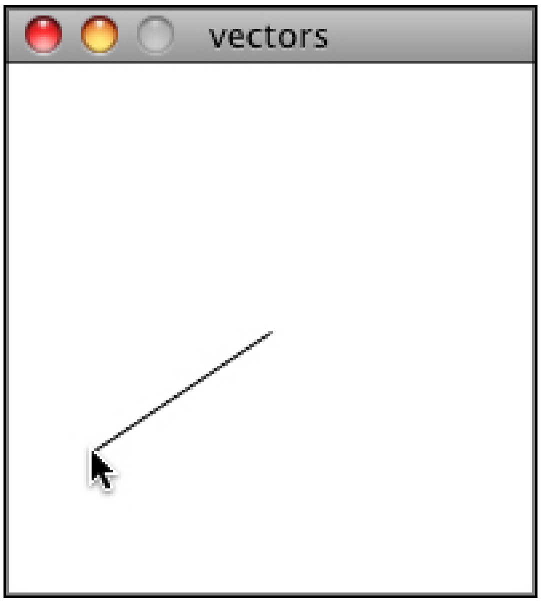
translates to:

$$w_x = u_x - v_x$$
$$w_y = u_y - v_y$$

and the function inside our `Vector` therefore looks like:

```
def __sub__(self, v):
    self.x = self.x - v.x
    self.y = self.y - v.y
    return self
```

Following is an example that demonstrates vector subtraction by taking the difference between two points – the mouse location and the center of the window.



```
from p5 import *

class Vector:
    def __init__
        self.x =
        self.y =

    def __add__(
        self.x =
        self.y =
        return s

    def __sub__(
        self.x =
```

(continues on next page)

(continued from previous page)

```
        self.y =
        return s

def setup():
    size(200, 20

def draw():
    background(2

↪ # Two vectors
↪location and c
    # of the win
    mouse = Vect
    center␣
↪= Vector(width

    # Vector sub
    mouse = mous

    # Draw␣
↪a line to repr
    translate(ce
    line((0, 0),

if __name__ == '
    run()
```

Note: Both addition and subtraction with vectors follows the same algebraic rules as with real numbers.

- The commutative rule: $\vec{u} + \vec{v} = \vec{v} + \vec{u}$
- The associative rule: $\vec{u} + (\vec{v} + \vec{w}) = (\vec{u} + \vec{v}) + \vec{w}$

The fancy terminology and symbols aside, this is really quite a simple concept. We're just saying that common sense properties of addition apply with vectors as well.

$$3 + 2 = 2 + 3$$

$$(3 + 2) + 1 = 3 + (2 + 1)$$

Moving onto multiplication, we have to think a little bit differently. When we talk about multiplying a vector what we usually mean is scaling a vector. Maybe we want a vector to be twice its size or one-third its size, etc. In this case, we are saying “Multiply a vector by 2” or “Multiply a vector by 1/3”. Note we are multiplying a vector by a scalar, a single number, not another vector.

To scale a vector by a single number, we multiply each component (x and y) by that number.

Vector multiplication:

$$\vec{w} = \vec{v} \cdot n$$

translates to:

$$w_x = v_x \cdot n$$
$$w_y = v_y \cdot n$$

Let’s look at an example with vector notation.

$$\vec{u} = \begin{pmatrix} -3 \\ 7 \end{pmatrix}, \quad n = 3$$
$$w = u \cdot n$$
$$w_x = -3 \cdot 3$$
$$w_y = 7 \cdot 3$$
$$\vec{w} = \begin{pmatrix} -9 \\ 21 \end{pmatrix}$$

The function inside the `Vector` class therefore is written as:

```
def __mul__(self, n):
    # With multiplication, all components of a vector are
    # multiplied by a number
    self.x = self.x * n
    self.y = self.y * n
    return self
```

And implementing multiplication in code is as simple as:

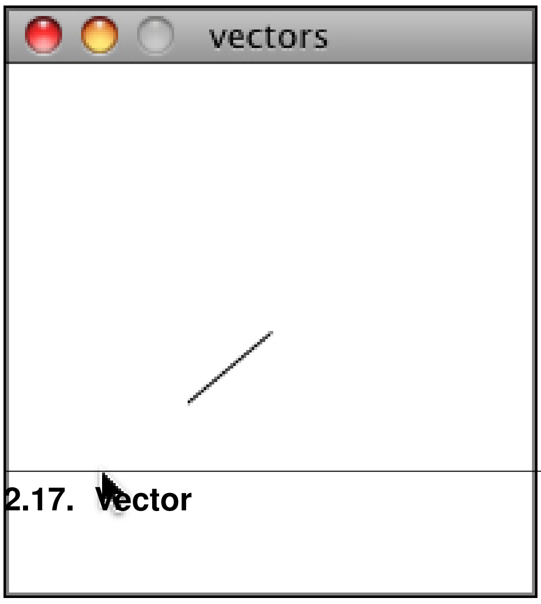
```
u = Vector(-3, 7)

# this vector is now three times the size and is equal to (-9, 21)
u = u * 3
```

```
from p5 import *

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

(continues on next page)

(continued from previous page)

```
    def __add__(self, v):
        self.x = self.x + v.x
        self.y = self.y + v.y
        return self

    def __sub__(self, v):
        self.x = self.x - v.x
        self.y = self.y - v.y
        return self

    def __mul__(self, n):
        self.x = self.x * n
        self.y = self.y * n
        return self

def setup():
    size(200, 200)

def draw():
    background(255)

    mouse = Vector(mouse_x, mouse_y)
    center = Vector(width / 2, height / 2)
    mouse = mouse - center

    # Vector multiplication!
    # The vector is now half its original size (multiplied by (1 /␣
↪2))
    mouse = mouse * (1 / 2)

    translate(center.x, center.y)
    line((0, 0), (mouse.x, mouse.y))

if __name__ == '__main__':
    run()
```



Division is exactly the same as multiplication, only of course using divide instead of multiply.

```
    def __truediv__(
        self.x = sel
        self.y = sel
        return self

    # ...

    u = Vector(8, -4
```

(continues on next page)

(continued from previous page)

```
u = u / 2
```

Note: As with addition, basic algebraic rules of multiplication and division apply to vectors.

- The associative rule: $(n \cdot m) \cdot \vec{v} = n \cdot (m \cdot \vec{v})$
- The distributive rule, 2 scalars, 1 vector: $(n + m) \cdot \vec{v} = n \cdot \vec{v} + m \cdot \vec{v}$
- The distributive rule, 2 vectors, 1 scalar: $(\vec{u} + \vec{v}) \cdot n$

2.17.5 Vectors: Magnitude

Multiplication and division, as we just saw, is a means by which the length of the vector can be changed without affecting direction. And so, perhaps you're wondering: "Ok, so how do I know what the length of a vector is?" I know the components (x and y), but I don't know how long (in pixels) that actual arrow is itself?!

The length or "magnitude" of a vector is often written as: $\|\vec{v}\|$

Understanding how to calculate the length (referred from here on out as magnitude) is incredibly useful and important.

Notice in the above diagram how when we draw a vector as an arrow and two components (x and y), we end up with a right triangle. The sides are the components and the hypotenuse is the arrow itself. We're very lucky to have this right triangle, because once upon a time, a Greek mathematician named Pythagoras developed a nice formula to describe the relationship between the sides and hypotenuse of a right triangle.

The Pythagorean theorem: a squared plus b squared equals c squared.

Armed with this lovely formula, we can now compute the magnitude of as follows:

$$\|\vec{v}\| = \sqrt{v_x^2 + y_y^2}$$

or in `Vector`:

```
def mag(self):
    return sqrt(self.x * self.x + self.y + self.y)
```

```
from p5 import *

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, v):
        self.x = self.x + v.x
        self.y = self.y + v.y
        return self

    def __sub__(self, v):
        self.x = self.x - v.x
        self.y = self.y - v.y
        return self

    def __mul__(self, n):
        self.x = self.x * n
        self.y = self.y * n
        return self

    def __div__(self, n):
        self.x = self.x / n
        self.y = self.y / n
        return self

    def mag(self):
        return sqrt(self.x * self.x + self.y * self.y)


def setup():
    size(200, 200)

def draw():
    background(255)

    mouse = Vector(mouse_x, mouse_y)
    center = Vector(width / 2, height / 2)
    mouse = mouse - center

    # The magnitude (i.e., the length) of a vector can be accessed␣
↪by
    # the mag() function. Here it is used as the width of a␣
↪rectangle
    # drawn at the top of the window.
    m = mouse.mag()
    fill(0)
    rect((0, 0), m, 10)
```

(continues on next page)

(continued from previous page)

```
    translate(center.x, center.y)
    line((0, 0), (mouse.x, mouse.y))

if __name__ == '__main__':
    run()
```

2.17.6 Vectors: Normalizing

Calculating the magnitude of a vector is only the beginning. The magnitude function opens the door to many possibilities, the first of which is **normalization**. Normalizing refers to the process of making something "standard" or, well, "normal." In the case of vectors, let's assume for the moment that a standard vector has a length of one. To normalize a vector, therefore, is to take a vector of any length and, keeping it pointing in the same direction, change its length to one, turning it into what is referred to as a **unit vector**.

Being able to quickly access the unit vector is useful since it describes a vector's direction without regard to length. For any given vector $\vec{u}$, its unit vector (written as $\hat{u}$) is calculated as follows:

$$\hat{u} = \frac{\vec{u}}{||\vec{u}||}$$

In other words, to normalize a vector, simply divide each component by its magnitude. This makes pretty intuitive sense. Say a vector is of length 5. Well, 5 divided by 5 is 1. So looking at our right triangle, we then need to scale the hypotenuse down by dividing by 5. And so in that process the sides shrink, dividing by 5 as well.

In the `Vector` class, we therefore write our normalization function as follows:

```
def normalize(self):
    m = self.mag()
    self = self / m
```

Of course, there's one small issue. What if the magnitude of the vector is zero? We can't divide by zero! Some quick error checking will fix that right up:

```
def normalize(self):
    m = self.mag()
    if not (m == 0):
        self = self / m
```

```
from p5 import *

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, v):
        self.x = self.x + v.x
        self.y = self.y + v.y
        return self

    def __sub__(self, v):
        self.x = self.x - v.x
        self.y = self.y - v.y
        return self

    def __mul__(self, n):
        self.x = self.x * n
        self.y = self.y * n
        return self

    def __truediv__(self, n):
        self.x = self.x / n
        self.y = self.y / n
        return self

    def mag(self):
        return sqrt(self.
↪x * self.x + self.y * self.y)

    def normalize(self):
        m = self.mag()
        if not (m == 0):
            self = self / m


def setup():
    size(200, 200)

def draw():
    background(255)

    mouse = Vector(mouse_x, mouse_y)
```

(continues on next page)

(continued from previous page)

```
    center␣
↪= Vector(width / 2, height / 2)
    mouse = mouse - center

    #␣
↪in this example, after the vector␣
↪is normalized it is multiplied
    # by 50 so that it is viewable␣
↪on screen. Note that no matter
  ␣
↪ # where the mouse is, the vector␣
↪will have the same length (50),
   ␣
↪# due to the normalization process
    mouse.normalize()
    mouse = mouse * 50

    translate(center.x, center.y)
    line((0, 0), (mouse.x, mouse.y))

if __name__ == '__main__':
    run()
```

2.17.7 Vectors: Motion

Why should we care? Yes, all this vector math stuff sounds like something we should know about, but why exactly? How will it actually help me write code? The truth of the matter is that we need to have some patience. The awesomeness of using the PVector class will take some time to fully come to light. This is quite common actually when first learning a new data structure. For example, when you first learn about an array, it might have seemed like much more work to use an array than to just have several variables to talk about multiple things. But that quickly breaks down when you need a hundred, or a thousand, or ten thousand things. The same can be true for `Vector`. What might seem like more work now will pay off later, and pay off quite nicely.

For now, however, we want to focus on simplicity. What does it mean to program motion using vectors? We've seen the beginning of this in this book's first example: the bouncing ball. An object on screen has a location (where it is at any given moment) as well as a velocity (instructions for how it should move from one moment to the next). Velocity gets added to location:

```
location = location + velocity
```

And then we draw the object at that location:

```
circle((location.x, location.y), 16)
```

This is Motion 101.

- Add velocity to location
- Draw object at location

In the bouncing ball example, all of this code happened in Processing's main tab, within *`p5.setup()`* and *`p5.draw()`* What we want to do now is move towards encapsulating all of the logic for motion inside of a class this way we can create a foundation for programming moving objects in Processing. We'll take a quick moment to review the basics of object-oriented programming in this context now, but this book will otherwise assume knowledge of working with objects (which will be necessary for just about every example from this point forward). However, if you need a further refresher, I encourage you to check out the *OOP Tutorial*

The driving principle behind object-oriented programming is the bringing together of data and functionality. Take the prototypical OOP example: a car. A car has data – `color, size, speed`, etc. A car has functionality – `drive()`, `turn()`, `stop()`, etc. A car class brings all that stuff together in a template from which car instances, i.e. objects, are made. The benefit is nicely organized code that makes sense when you read it.

```
c = Car(red, big, fast)
c.drive()
c.turn()
c.stop()
```

In our case, we're going to create a generic "Mover" class, a class to describe a shape moving about the screen. And so we must consider the following two questions:

1. What data does a Mover have?

2. What functionality does a Mover have?

   Our "Motion 101" algorithm tells us the answers to these questions. The data an object has is its location and its velocity, two *p5.Vector* objects.

```
class Mover:
    def __init__(self, ...):
        self.location = Vector(...)
        self.velocity = Vector(...)
```

Note: To keep our code concise, we're now switching to the *p5.Vector* class that comes with p5. So we can remove the custom `Vector` code that we wrote from our main sketch.

Its functionality is just about as simple. It needs to move and it needs to be seen. We'll implement these as functions named `update()` and `display()`. `update()` is where we'll put all of our motion logic code and `display()` is where we will draw the object.

```
def update(self):
    self.location␣
↪= self.location + self.velocity

def display(self):
    stroke(0)
    fill(175)
    circle(self.location, 16)
```

We've forgotten one crucial item, however, the object's **constructor** . The constructor is a special function inside of a class that creates the instance of the object itself. It is where you give the instructions on how to set up the object. In Python this constructor should always be called `__init__`. It gets called whenever we create a new object using `my_car = Car()`.

Important: All methods defined in a class in Python require `self` as the first parameter.

In our case, let's just initialize our mover object by giving it a random location and a random velocity.

```
class Mover:
    ␣
↪def __init__(self, width, height):
```

(continues on next page)

(continued from previous page)

```
        self.location␣
↪= Vector(random_uniform(width),
                                    ␣
↪            random_uniform(height))


  ␣
↪    self.velocity = Vector(random_
↪uniform(low=-2, high=2),
                                      ␣
↪   random_uniform(low=-2, high=2))
```

Let's finish off the Mover class by incorporating a function to determine what the object should do when it reaches the edge of the window. For now let's do something simple, and just have it wrap around the edges.

```
def check_edges(self):
    if self.location.x > width:
        self.location.x = 0

    if self.location.x < 0:
        self.location.x = width

    if self.location.y > height:
        self.location.y = 0

    if self.location.y < 0:
        self.location.y = height
```

Now that the Mover class is finished, we can then look at what we need to do in our main program. We first declare a placeholder for a Mover object:

```
mover = None
```

Then initialize the mover in *p5.setup()*:

```
mover = Mover(width, height)
```

and call the appropriate functions in draw():

```
mover.update()
mover.check_edges()
mover.display()
```

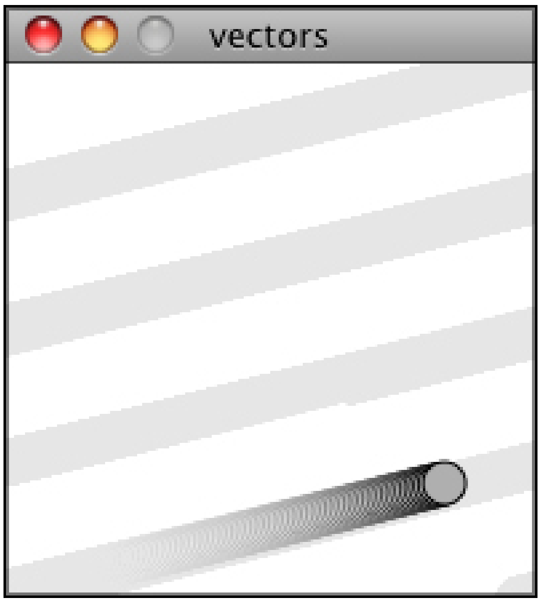
Here is the entire example for reference:

```
from p5 import *

mover = None

class Mover:
    def __init__(self):
        # our object has␣
↪two Vectors: location and velocity
        self.location␣
↪= Vector(random_uniform(width),
                                 ␣
↪           random_uniform(height))

  ␣
↪     self.velocity = Vector(random_
↪uniform(low=-2, high=2),
                                     ␣
↪   random_uniform(low=-2, high=2))

    def update(self):
        # Motion␣
↪101: Locations change by velocity
        self.location␣
↪= self.location + self.velocity

    def display(self):
        stroke(0)
        fill(175)
        circle(self.location, 16)

    def check_edges(self):
        if self.location.x > width:
            self.location.x = 0

        if self.location.x < 0:
            self.location.x = width

        if self.location.y > height:
            self.location.y = 0

        if self.location.y < 0:
            self.location.y = height

def setup():
    global mover
    size(200, 200)
    background(255)

    # make the mover object
    mover = Mover()
```

(continues on next page)

(continued from previous page)

```
                        def draw():
                            no_stroke()
                            fill(255, 10)
                            rect((0, 0), width, height)

                            # call functions on Mover object
                            mover.update()
                            mover.check_edges()
                            mover.display()

                        if __name__ == '__main__':
                            run()
```



Ok, at this point, we should feel comfortable with two things – (1) What is a *p5.Vector*? and (2) How do we use Vectors inside of an object to keep track of its location and movement? This is an excellent first step and deserves an mild round of applause. For standing ovations and screaming fans, however, we need to make one more, somewhat larger, step forward. After all, watching the Motion 101 example is fairly boring – the circle never speeds up, never slows down, and never turns. For more interesting motion, for motion that appears in the real world around us, we need to add one more Vector to our class – acceleration.

The strict definition of acceleration that we are using here is: **the rate of change of velocity**. Let's think about that definition for a moment. Is this a new concept? Not really. Velocity is defined as: **the rate of change of location**. In essence, we are developing a "trickle down" effect. Acceleration affects velocity which in turn affects location (for some brief foreshadowing, this point will become even more crucial in the next chapter when we see how forces affect acceleration which affects velocity which affects location.) In code, this reads like this:

```
velocity = velocity + acceleration
location = location + velocity
```

As an exercise, from this point forward, let's make a rule for ourselves. Let's write every example in the rest of this book without ever touching the value of velocity and location (except to initialize them). In other words, our goal now for programming motion is as follows – come up with an algorithm for how we calculate acceleration and let the trickle down effect work its magic. And so we need to come up with some ways to calculate acceleration:

ACCELERATION ALGORITHMS!

1. Make up a constant acceleration
2. A totally random acceleration
3. Perlin noise acceleration
4. Acceleration towards the mouse

Number one, though not particularly interesting, is the simplest, and will help us get started incorporating acceleration into our code. The first thing we need to do is add another *p5.Vector* to the Mover class:

```
class Mover:
    def __init__(self):
        location = Vector(...)
        velocity = Vector(...)

        # A new Vector for acceleration
        acceleration = Vector(...)
```

And incorporate acceleration into the `update()` function:

```
def update(self):
    # our motion algorithm is now two lines of code:
    self.velocity = self.velocity + self.acceleration
    self.location = self.location + self.velocity
```

We're almost done. The only missing piece is the actual initialization in the constructor.

Let's start the object in the middle of the window..

```
self.location = Vector(width / 2, height / 2)
```

. . . with an initial velocity of zero.

```
self.velocity = Vector(0, 0)
```

This means that when the sketch starts, the object is at rest. We don't have to worry about velocity anymore as we are controlling the object's motion entirely with acceleration. Speaking of which, according to "algorithm #1" our first sketch involves constant acceleration. So let's pick a value.

```
self.acceleration = Vector(-0.001, 0.01)
```

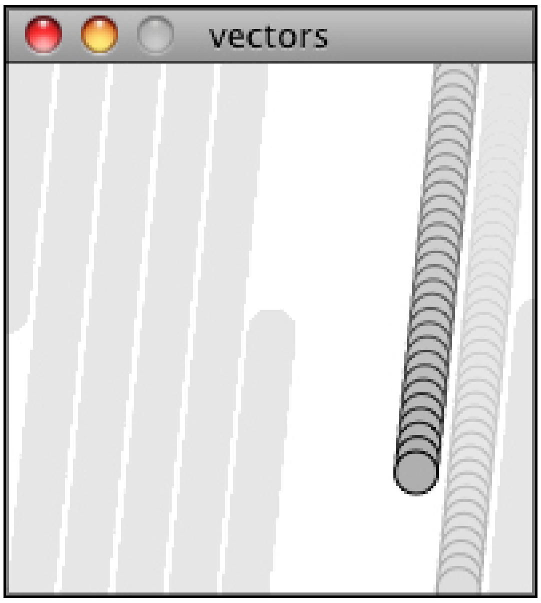
Are you thinking – "Gosh, those values seem awfully small!" Yes, that's right, they are quite tiny. It's important to realize that our acceleration values (measured in pixels) accumulate into the velocity over time, about thirty times per second depending on our sketch's frame rate. And so to keep the magnitude of the velocity vector within a reasonable range, our acceleration values should remain quite small. We can also help this cause by incorporating the Vector function *p5.Vector.limit()*

```
# the limit() function constrains the magnitude of the vector
self.velocity.limit(10)
```

This translates to the following:

> What is the magnitude of velocity? If it's less than 10, no worries, just leave it whatever it is. If it's more than 10, however, shrink it down to 10!

Let's take a look at the changes to the Mover class now, complete with acceleration and `limit()`.



```
class Mover:
    def __init__
        self.loc
↪= Vector(width
        self.vel

        # Accele
        self.acc
↪= Vector(-0.00

        # this w
↪limit the magn
        self.top

    def update(s
        self.vel
↪self.velocity
        self.
↪velocity.limit
        self.loc
↪= self.locatio

      ␣
↪# rest of the
```

Ok, algorithm #2 – "a totally random acceleration." In this case, instead of initializing acceleration in the object's constructor we want to pick a new acceleration each cycle, i.e. each time update() is called.

```
def update(self)
    acc_x␣
↪= random_unifo
    acc_y␣
↪= random_unifo
    self.acceler
↪= Vector(acc_x
    self.acceler

    self.velocit
↪self.velocity
```

(continues on next page)

(continued from previous page)

```
                    self.
                  ↪velocity.limit
                    self.locatio
                  ↪= self.locatio
```

While normalizing acceleration is not entirely necessary, it does prove useful as it standardizing the magnitude of the vector, allowing us to try different things, such as:

1. scaling the acceleration to a constant value

```
acc_x =
 ↪unifo
acc_y =
 ↪unifo
self.ac
 ↪= Vec
self.ac

self.ac
 ↪= sel
```

2. scaling the acceleration to a random value

```
acc_x =
 ↪unifo
acc_y =
 ↪unifo
self.ac
 ↪= Vec
self.ac

self.ac
 ↪= sel
 ↪* ran
```

While this may seem like an obvious point, it's crucial to understand that acceleration does not merely refer to the speeding up or slowing down of a moving object, but rather any change in velocity, either magnitude or direction. Acceleration is used to steer an object, and it is the foundation of learning to program an object that make decisions about how to move about the screen.

2.17.8 Vectors: Interactivity

Ok, to finish out this tutorial, let's try something a bit more complex and a great deal more useful.

Let's dynamically calculate an object's acceleration according to a rule, acceleration algorithm #4 – "the object accelerates towards the mouse."

Anytime we want to calculate a vector based on a rule/formula, we need to compute two things: **magnitude** and **direction**. Let's start with direction. We know the acceleration vector should point from the object's location towards the mouse location. Let's say the object is located at the point `(x, y)` and the mouse at `(mouse_x, mouse_y)`.

As illustrated in the above diagram, we see that we can get a vector (dx, dy) by subtracting the object's location from the mouse's location. After all, this is precisely where we started this chapter – the definition of a vector is "the difference between two points in space!"

```
dx = mouse_x - x
dy = mouse_y - y
```

Let's rewrite the above using Vector syntax. Assuming we are in the Mover class and thus have access to the object's location Vector, we then have:

```
mouse = Vector(mouse_x, mouse_y)
direction = mouse - self.location
```

We now have a Vector that points from the mover's location all the way to the mouse. If the object were to actually accelerate using that vector, it would instantaneously appear at the mouse location. This does not make for good animation, of course, and what we want to do is now decide how fast that object should accelerate towards the mouse.

In order to set the magnitude (whatever it may be) of our acceleration PVector, we must first ________ that direction vector. That's right, you said it. **Normalize**. If we can shrink the vector down to its unit vector (of length one) then we have a vector that tells us the direction and can easily be scaled to any value. One multiplied by anything equals anything.

```
anything = some_number
direction.normalize()
direction = direction * anything
```

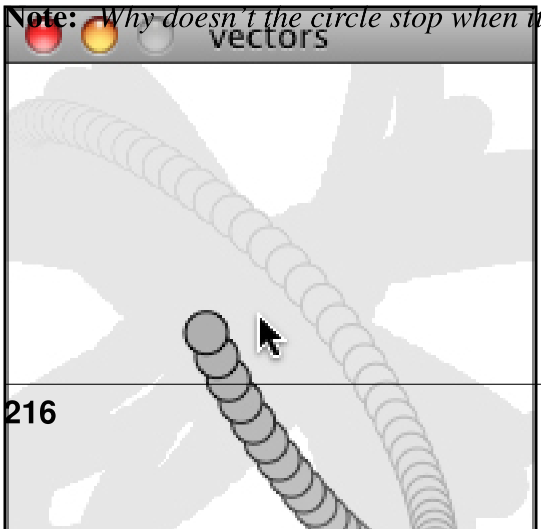
To summarize, we have the following steps:

1. Calculate a vector that points from the object to the target location (mouse).
2. Normalize that vector (reducing its length to 1)
3. Scale that vector to an appropriate value (by multiplying it by some value)
4. Assign that vector to acceleration

And here are those steps in the update() function itself:

```
def update(self):
    mouse = Vector(mouse_x, mouse_y)

    # Step 1. direction
   ␣
↪ direction = mouse - self.location

    # Step 2: normalize
    direction.normalize()

    # Step 3: scale
    direction = direction * 0.5

    # Step 4: accelerate
    self.acceleration = direction;

    self.velocity =␣
↪self.velocity + self.acceleration
    self.
↪velocity.limit(self.top_speed)
    self.location␣
↪= self.location + self.velocity
```

Note: *Why doesn't the circle stop when it reaches the target?*



The object moving has no knowledge about trying to stop at a destination, it only knows where the destination is and tries to go there as fast as possible. Going as fast as possible means it will inevitably overshoot the location and have to turn around, again going as fast as possible towards the destination, overshooting it again, and so on, and so forth. Stay tuned for later chapters where we see how to program an object to "arrive"s at a location (slowing down on approach.)

Let's take a look at what this example would look like with an array of Mover objects (rather than just one).

```
from p5 import *

num_movers = 20
movers = []

class Mover:
    def __init__(self):
        self.location␣
↪= Vector(random_uniform(width),
                                   ␣
↪             random_uniform(height))

        self.velocity = Vector(0, 0)
     ␣
↪   self.acceleration = Vector(0, 0)
        self.top_speed = 4

    def update(self):
        # our algorithm␣
↪for calculating acceleration
     ␣
↪   mouse = Vector(mouse_x, mouse_y)

        # find␣
↪vector pointing towards the mouse
      ␣
↪  direction = mouse - self.location

        # normalize
        direction.normalize()

        # scale
        direction = direction * 0.5

        # set acceleration
   ␣
↪    self.acceleration = direction;

        self.velocity =␣
↪self.velocity + self.acceleration
        self.
↪velocity.limit(self.top_speed)
```

(continues on next page)

(continued from previous page)

```
        self.location␣
↪= self.location + self.velocity

    def display(self):
        stroke(0)
        fill(175)
        circle(self.location, 16)

    def check_edges(self):
        if self.location.x > width:
            self.location.x = 0

        if self.location.x < 0:
            self.location.x = width

        if self.location.y > height:
            self.location.y = 0

        if self.location.y < 0:
            self.location.y = height

def setup():
    size(200, 200)
    background(255)

    # creating many mover objects
    for _ in range(num_movers):
        movers.append(Mover())

def draw():
    no_stroke()
    fill(255, 10)
    rect((0, 0), width, height)

    # call functions␣
↪on all objects in the array
    for mover in movers:
        mover.update()
        mover.check_edges()
        mover.display()

if __name__ == '__main__':
    run()
```

Beginner tutorials

- **Getting Started by Casey Reas and Ben Fry:** Welcome to Processing! This introduction covers the basics of writing Processing code.

- **Processing Overview b by Ben Fry and Casey Reasy:** A little more detailed introduction to the different features of Processing than the Getting Started tutorial.
- *Coordinate System and Shapes by Daniel Shiffman*: Drawing simple shapes and using the coordinate system.
- *Color by Daniel Shiffman*: An introduction to digital color.
- *Objects by Daniel Shiffman*: The basics of object-oriented programming.
- *Interactivity by Casey Reas and Ben Fry by Daniel Shiffman*: Introduction to interactivity with the mouse and keyboard.
- *Typography by Casey Reas and Ben Fry*: Working with typefaces and text.

Intermediate tutorials

- *Strings and Drawing Text by Daniel Shiffman*: Learn how use the String class and display text onscreen.
- *Arrays by Casey Reas and Ben Fry*: How to store and access data in array structures.
- *Images and Pixels by Daniel Shiffman*: How to load and display images as well as access their pixels.
- *Curves by J David Eisenberg*: Learn how to draw arcs, spline curves, and bezier curves.
- *2D Transformations by J David Eisenberg*: Learn how to translate, rotate, and scale shapes using 2D transformations.
- *PShape by Daniel Shiffman*: How to use the PShape class in Processing.
- *Data by Daniel Shiffman*: Learn the basics of working with data feeds in Processing.
- *Trigonometry Primer by Ira Greenberg*: An introduction to trigonometry.

- **Render Techniques by Casey Reas and Ben Fry:** Tools for rendering geometries in Processing.
- *Two-Dimensional Arrays by Daniel Shiffman*: Store and acess data in a matrix using a two-dimensional array.
- **Sound by R. Luke DuBois and Wilm Thoben:** Learn how to play, analyze, and synthesize sound with the Sound Library.

- *Electronics by Hernando Berragán and Casey Reas*: Control physical media with Processing, Arduino, and Wiring.
- *Network by Alexander R. Galloway*: An introduction to sending and receiving data with clients and servers
- **Print by Casey Reas:** Use Processing to output print quality images and documents.

Advanced tutorials

- **Shaders by Andres Colubri:** A guide to implementing GLSL shaders in Processing.
- *Vectors by Daniel Shiffman*: An introduction to using the PVector class in Processing.
- **P3D by Daniel Shiffman:** Developing advanced graphics applications in Processing using P3D (OpenGL) mode.
- **Video by Daniel Shiffman:** How to display live and recorded video
- **Anatomy of a Program by J David Eisenberg:** How do you analyze a problem and break it down into steps that the computer can do?

CHAPTER 3

Examples

These examples are running online through p5.js using HTML Canvas for rendering. There are many more examples included with the Processing application; please look there if you don't find what you're looking for here.

3.1 Structure

3.1.1 Statements and Comments

Statements are the elements that make up programs. In Python, each line is a statement. Comments are used for making notes to help people better understand programs. A comment begins with a number sign ("#").

```
from p5 import *

def draw():
        # The size function is a statement that tells the computer
        # how large to make the window.
        # Each function statement has zero or more parameters.
        # Parameters are data passed into the function
        # and are used as values for telling the computer what to␣
↪do.
        size(640, 360)

        # The background function is a statement that tells the␣
↪computer
        # which color (or gray value) to make the background of the␣
↪display window
```

(continues on next page)

(continued from previous page)

```
        background(204, 153, 0)

if __name__ == '__main__':
    run()
```

3.1.2 Coordinates

All shapes drawn to the screen have a position that is specified as a coordinate. All coordinates are measured as the distance from the origin in units of pixels. The origin [0, 0] is the coordinate is in the upper left of the window and the coordinate in the lower right is [width-1, height-1].

```
from p5 import *

def draw():
    # Sets the screen to be 640 pixels wide and 360 pixels high
    size(640, 360)

    # Set the background to black and turn off the fill color
    background(0)
    no_fill()

    # The two parameters of the point() method each specify␣
↪coordinates.
    # The first parameter is the x-coordinate and the second is the␣
↪Y
    stroke(255)
    point(width * 0.5, height * 0.1)
    point(width * 0.5, height * 0.9)

    # Coordinates are used for drawing all shapes, not just points.
    # Parameters for different functions are used for different␣
↪purposes.
    # For example, the first parameter to line() specify
    # the coordinates of the first endpoint and the second parameter
    # specify the second endpoint
    stroke(0, 153, 255)
    line((0, height*0.33), (width, height*0.33))

    # By default, the first parameter to rect() is the
    # coordinates of the upper-left corner and the second and third
    # parameter is the width and height
    stroke(255, 153, 0)
    rect((width*0.25, height*0.1), width * 0.5, height * 0.8)

if __name__ == '__main__':
    run()
```

3.1.3 Width and Height

The `width` and `height` variables contain the width and height of the display window as defined in the `size()` function.

```
from p5 import *

def setup():
        size(640, 360)

def draw():
        background(127)
        no_stroke()

        for i in range(0, height, 20):
                fill(129, 206, 15)
                rect((0, i), width, 10)
                fill(255)
                rect((i, 0), 10, height)

if __name__ == '__main__':
    run()
```

3.1.4 Setup and Draw

The code inside the draw() function runs continuously from top to bottom until the program is stopped.

```
from p5 import *

# The statement in setup() function
# ececute once when the program begins
def setup():
        size(640, 360) # size must be the first statement
        stroke(255) # Set line drawing color to white
        y = 0

# The statements in draw() are executed until the
# program is stopped. Each statement is executed in
# sequence and after the last line is read, the first
# line is executed again.
def draw():
        background(0) # Clear the screen with a black background
        y = y - 1
        if y < 0:
                y = height

        line((0, y), (width, y))
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    run()
```

3.1.5 No Loop

The `no_loop()` function causes `draw()` to only execute once. Without calling `no_loop()`, the code inside `draw()` is run continually.

```
from p5 import *

y = 0

# The statements in the setup() function
# run once when the program begins
def setup():
    size(640, 360)   # Size should be the first statement
    stroke(255)      # Set stroke color to white
    no_loop()

    global y
    y = height * 0.5

# The statements in draw() are executed until the
# program is stopped. Each statement is executed in
# sequence and after the last line is read, the first
# line is executed again.
def draw():
    background(0)  # Set the background to black
    global y
    y = y - 1
    if y < 0:
        y = height

    line((0, y), (width, y))


if __name__ == '__main__':
    run()
```

3.1.6 Loop

The `loop()` function causes `draw()` to execute continuously. If noLoop is called in `setup()` the `draw()` is only executed once. In this example click the mouse to execute `loop()`, which will cause the `draw()` the execute continuously.

```
from p5 import *

y = 0

# The statements in the setup() function
# run once when the program begins
def setup():
    size(640, 360)   # Size should be the first statement
    stroke(255)      # Set stroke color to white
    no_loop()

    global y
    y = height * 0.5

# The statements in draw() are executed until the
# program is stopped. Each statement is executed in
# sequence and after the last line is read, the first
# line is executed again.
def draw():
    background(0) # Set the background to black
    global y
    y = y - 1
    if y < 0:
        y = height

    line((0, y), (width, y))

def mouse_pressed():
    loop()

if __name__ == '__main__':
    run()
```

3.1.7 Redraw

The `redraw()` function makes `draw()` execute once. In this example, `draw()` is executed once every time the mouse is clicked.

```
from p5 import *

y = 0

# The statements in the setup() function
# run once when the program begins
def setup():
    size(640, 360)   # Size should be the first statement
    stroke(255)      # Set stroke color to white
    no_loop()
```

(continues on next page)

(continued from previous page)

```
    global y
    y = height * 0.5

# The statements in draw() are executed until the
# program is stopped. Each statement is executed in
# sequence and after the last line is read, the first
# line is executed again.
def draw():
    background(0) # Set the background to black
    global y
    y = y - 1
    if y < 0:
        y = height

    line((0, y), (width, y))

def mouse_pressed():
    redraw()

if __name__ == '__main__':
    run()
```

3.1.8 Function

The `draw_target()` function makes it easy to draw many distinct targets. Each call to `draw_target()` specifies the position, size, and number of rings for each target.

```
from p5 import *

def setup():
    size(640, 360)
    stroke(255)
    no_stroke()

def draw():
    background(51)
    draw_target(width * 0.25, height * 0.4, 200, 4)
    draw_target(width * 0.5, height * 0.5, 300, 10)
    draw_target(width * 0.75, height * 0.3, 120, 6)

def draw_target(xloc, yloc, size, num):
    grayvalues = 255 / num
    steps = size / num
    for i in range(num):
        fill(i * grayvalues)
        ellipse((xloc, yloc), size - i * steps, size - i *steps)
```

(continues on next page)

(continued from previous page)

```python

if __name__ == '__main__':
      run()
```

3.1.9 Recursion

A demonstration of recursion, which means functions call themselves. Notice how the `draw_circle()` function calls itself at the end of its block. It continues to do this until the variable `level` is equal to 1.

```python
from p5 import *

def setup():
    size(640, 360)
    no_stroke()
    no_loop()

def draw():
    draw_circle(width / 2, 280, 6)

def draw_circle(x, radius, level):
    tt = 126 * level / 4.0
    fill(tt)
    ellipse((x, height / 2), radius * 2, radius * 2)
    if level > 1:
        level = level - 1
        draw_circle(x - radius / 2, radius / 2, level)
        draw_circle(x + radius / 2, radius / 2, level)

if __name__ == '__main__':
    run()
```

3.2 Transform

3.2.1 Translate

The `translate()` function allows objects to be moved to any location within the window. The first parameter sets the x-axis offset and the second parameter sets the y-axis offset.

```python
from p5 import *

x = 0
y = 0
dim = 80.0
```

(continues on next page)

(continued from previous page)

```

def setup():
        size(640, 360)
        no_stroke()

def draw():
        background(102)

        global x
        x = x + 0.8

        if x > width + dim:
                x = -dim

        translate(x, height/2 - dim/2)
        fill(255)
        rect((-dim/2, -dim/2), dim, dim)

        # Transforms accumulate. Notice how this rect moves
        # twice as fast as the other, but it has the same
        # parameter for the x-axis value
        translate(x, dim)
        fill(0)
        rect((-dim/2, -dim/2), dim, dim)

if __name__ == '__main__':
  run()
```

3.2.2 Scale

Paramenters for the `scale()` function are values specified as decimal percentages. For example, the method call scale(2.0) will increase the dimension of the shape by 200 percent. Objects always scale from the origin.

```
from p5 import *

a = 0.0
s = 0.0

def setup():
        size(640, 360)
        no_stroke()

def draw():
        background(102)

        global a
```

(continues on next page)

(continued from previous page)

```
        global s

        a = a + 0.04
        s = cos(a)*2

        translate(width/2, height/2)
        scale(s)
        fill(51)
        rect((-25, -25), 50, 50)

        translate(75, 0)
        fill(255)
        scale(s)
        rect((-25, -25), 50, 50)

if __name__ == '__main__':
  run()
```

3.2.3 Rotate

Rotating a square around the Z axis. To get the results you expect, send the rotate function angle parameters that are values between 0 and PI*2 (TWO_PI which is roughly 6.28). If you prefer to think about angles as degrees (0-360), you can use the radians() method to convert your values. For example: scale(radians(90)) is identical to the statement scale(PI/2).

```
from p5 import *

angle = 0.0
jitter = 0.0

def setup():
        size(640, 360)
        fill(255)
        no_stroke()

def draw():
        background(102)

        global angle
        global jitter

        if second()%2 == 0:
                jitter = random_uniform(-0.1, 0.1)

        angle = angle + jitter
        c = cos(angle)
        translate(width/2, height/2)
```

(continues on next page)

(continued from previous page)

```
        rotate(c)
        rect((-90, -90), 180, 180)

if __name__ == '__main__':
  run()
```

3.2.4 Arm

The angle of each segment is controlled with the mouseX and mouseY position. The transformations applied to the first segment are also applied to the second segment because they are inside the same `pushMatrix()` and `popMatrix()` group.

```
from p5 import *

x = 0
y = 0
angle1 = 0.0
angle2 = 0.0
segLength = 100

def setup():
        size(640, 360)
        stroke(255, 160)
        stroke_weight(30)

        global x, y
        x = width * 0.3
        y = height * 0.5

def draw():
        background(0)

        angle1 = (mouse_x/width - 0.5) * -PI
        angle2 = (mouse_y/height - 0.5) * PI

        with push_matrix():
                segment(x, y, angle1)
                segment(segLength, 0, angle2)

def segment(x, y, a):
        translate(x, y)
        rotate(a)
        line((0, 0), (segLength, 0))

if __name__ == '__main__':
  run()
```

3.3 Color

3.3.1 Hue

Hue is the color reflected from or transmitted through an object and is typically referred to as the name of the color (red, blue, yellow, etc.) Move the cursor vertically over each bar to alter its hue.

```
from p5 import *

barWidth = 20
lastBar = -1

def setup():
        size(640, 360)
        stroke(255, 160)
        color_mode('HSB', height, height, height)
        no_stroke()
        background(0)

def draw():
        global barWidth, lastBar

        whichBar = mouse_x / barWidth
        if whichBar != lastBar:
                barX = whichBar*barWidth
                fill(mouse_x, height, height)
                rect((barX, 0), barWidth, height)
                lastBar = whichBar

if __name__ == '__main__':
  run()
```

3.3.2 Saturation

Saturation is the strength or purity of the color and represents the amount of gray in proportion to the hue. A "saturated" color is pure and an "unsaturated" color has a large percentage of gray. Move the cursor vertically over each bar to alter its saturation.

```
from p5 import *

barWidth = 20
lastBar = -1

def setup():
        size(640, 360)
        color_mode('HSB', height, height, height)
```

(continues on next page)

(continued from previous page)

```
        no_stroke()

def draw():
        global barWidth, lastBar

        whichBar = mouse_x / barWidth
        if whichBar != lastBar:
                barX = whichBar*barWidth
                fill(barX, mouse_y, height)
                rect((barX, 0), barWidth, height)
                lastBar = whichBar

if __name__ == '__main__':
  run()
```

3.3.3 Brightness

This program adjusts the brightness of a part of the image by calculating the distance of each pixel to the mouse.

```
from p5 import *

bar_width = 20
last_bar = None

def setup():
    size(640, 360)
    title("Brightness")
    color_mode('HSB', width, 100, height)
    no_stroke()
    background(0)

def draw():
    global last_bar
    which_bar = mouse_x // bar_width
    if which_bar is not last_bar:
        bar_x = which_bar * bar_width
        fill(bar_x, 100, mouse_y)
        rect((bar_x, 0), bar_width, height)
        last_bar = which_bar

if __name__ == '__main__':
    run()
```

3.3.4 Color Variables

This example creates variables for colors that may be referred to in the program by a name, rather than a number.

```
from p5 import *

def setup():
        size(640, 360)

def draw():
    no_stroke()

    bg_col = Color(51, 0, 0)
    col_1 = Color(204, 102, 0)
    col_2 = Color(204, 153, 0)
    col_3 = Color(153, 51, 0)

    background(bg_col)

    translate(50, 50)
    fill(col_3)
    rect((0, 0), 200, 200)
    fill(col_2)
    rect((40, 60), 120, 120)
    fill(col_1)
    rect((60, 90), 80, 80)

    translate(250, 0)
    fill(col_1)
    rect((0, 0), 200, 200)
    fill(col_3)
    rect((40, 60), 120, 120)
    fill(col_2)
    rect((60, 90), 80, 80)

if __name__ == '__main__':
  run()
```

3.3.5 Relativity

Each color is perceived in relation to other colors. The top and bottom bars each contain the same component colors, but a different display order causes individual colors to appear differently.

```
from p5 import *

a = b = c = d = e = None
```

(continues on next page)

(continued from previous page)

```
def setup():
        size(640, 360)
        no_stroke()

        global a, b, c, d, e
        a = Color(165, 167, 20)
        b = Color(77, 86, 59)
        c = Color(42, 106, 105)
        d = Color(165, 89, 20)
        e = Color(146, 150, 127)

        no_loop()


def draw():
        global a, b, c, d, e
        drawBand(a, b, c, d, e, 0, width/128)
        drawBand(c, a, d, b, e, height/2, width/128)

def drawBand(v, w, x, y, z, ypos, barWidth):
        num = 5
        colorOrder = [v, w, x, y, z]

        for i in range(0, width, int(barWidth*num)):
                for j in range(num):
                        fill(colorOrder[j])
                        rect((i + j*barWidth, ypos), barWidth,␣
↪height/2)

if __name__ == '__main__':
  run()
```

3.3.6 Linear Gradient

The `lerpColor()` function is useful for interpolating between two colors.

```
from p5 import *

Y_AXIS = 1
X_AXIS = 2
b1 = b2 = c1 = c2 = None

def setup():
        size(640, 360)

        global b1, b2, c1, c2
        b1 = Color(255)
```

(continues on next page)

(continued from previous page)

```
        b2 = Color(0)
        c1 = Color(204, 102, 0)
        c2 = Color(0, 102, 153)

        no_loop()


def draw():
        global b1, b2, c1, c2

        # background
        setGradient(0, 0, width/2, height, b1, b2, X_AXIS)
        setGradient(width/2, 0, width/2, height, b2, b1, X_AXIS)
        # Foreground
        setGradient(50, 90, 540, 80, c1, c2, Y_AXIS)
        setGradient(50, 190, 540, 80, c2, c1, X_AXIS)

def setGradient(x, y, w, h, c1, c2, axis):
        no_fill()

        if axis == Y_AXIS:
                for i in range(y, y + int(h) + 1):
                        inter = remap(i, [y, y+h], [0, 1])
                        c = c1.lerp(c2, inter)
                        stroke(c)
                        line((x, i), (x+w, i))
        elif axis == X_AXIS:
                for i in range(int(x), int(x + w) + 1):
                        inter = remap(i, [x, x+w], [0, 1])
                        c = c1.lerp(c2, inter)
                        stroke(c)
                        line((i, y), (i, y+h))

if __name__ == '__main__':
  run()
```

3.3.7 Radial Gradient

Draws a series of concentric circles to create a gradient from one color to another.

```
from p5 import *

dim = None

def setup():
        size(640, 360)
```

(continues on next page)

(continued from previous page)

```
        global dim
        dim = width/2;
        background(0)
        color_mode('HSB', 360, 100, 100)
        no_stroke()

def draw():
        global dim
        background(0)

        x = 0
        while x <= width:
                drawGradient(x, height/2)
                x += dim

def drawGradient(x, y):
        radius = dim/2
        h = random_uniform(0, 360)

        for r in range(int(radius), 0, -1):
                fill(h, 90, 90)
                ellipse((x, y), r*2, r*2)
                h = (h + 1) % 360

if __name__ == '__main__':
  run(frame_rate=1)
```

3.4 Arrays

3.4.1 Array

An array is a list of data. Each piece of data in an array is identified by an index number representing its position in the array. Arrays are zero based, which means that the first element in the array is [0], the second element is [1], and so on. In this example, an array named “coswave” is created and filled with the cosine values. This data is displayed three separate ways on the screen.

```
from p5 import *

coswave = []

def setup():
        size(720, 360)
        for i in range(width):
                amount = remap(i, [0, width], [0, PI])
                coswave.append(abs(cos(amount)))
```

(continues on next page)

(continued from previous page)

```
        background(255)
        no_loop()

def draw():
        y1 = 0
        y2 = height/3
        for i in range(0, width, 3):
                stroke(coswave[i] * 255)
                line((i, y1), (i, y2))

        y1 = y2
        y2 = y1 + y1
        for i in range(0, width, 3):
                stroke(coswave[i]*255 / 4)
                line((i, y1), (i, y2))

        y1 = y2
        y2 = height
        for i in range(0, width, 3):
                stroke(255 - coswave[i] * 255)
                line((i, y1), (i, y2))

if __name__ == '__main__':
  run()
```

3.4.2 Array 2D

Demonstrates the syntax for creating a two-dimensional (2D) array. Values in a 2D array are accessed through two index values. 2D arrays are useful for storing images. In this example, each dot is colored in relation to its distance from the center of the image.

```
from p5 import *

distances = []
maxDiantance = None
spacer = None

def setup():
        size(720, 360)

        global distances, maxDiantance, spacer
        maxDistance = dist((width / 2, height / 2), (width, height))
        for x in range(width):
                d = []
                for y in range(height):
                        distance = dist((width / 2, height / 2), (x,
↪ y))
```

(continues on next page)

(continued from previous page)

```
                                d.append((distance / maxDistance) * 255)
                        distances.append(d)

                spacer = 10
                no_loop()

def draw():
                background(0)

                for x in range(0, width, spacer):
                        for y in range(0, height, spacer):
                                stroke(distances[x][y])
                                point(x + spacer / 2, y + spacer / 2)

if __name__ == '__main__':
  run()
```

3.4.3 Array Object

Demonstrates the syntax for creating an array of custom objects.

```
from p5 import *

unit = 40
count = None
mods = []

def setup():
        size(640, 360)
        no_stroke()

        global count, unit, mods
        wideCount = width / unit
        highCount = height / unit

        count = wideCount * highCount
        index = 0
        for y in range(int(highCount)):
                for x in range(int(wideCount)):
                        mods.append(Module(x*unit, y*unit, unit/2,␣
↪unit/2, random_uniform(0.05, 0.8), unit))

def draw():
        background(0)
        for mod in mods:
                mod.update()
                mod.display()
```

(continues on next page)

(continued from previous page)

```
class Module:
        def __init__(self, xOffsetTemp, yOffsetTemp, xTemp, yTemp,␣
↪speedTemp, tempUnit):
                self.xOffset = xOffsetTemp
                self.yOffset = yOffsetTemp

                self.x = xTemp
                self.y = yTemp
                self.speed = speedTemp
                self.unit = tempUnit

                self.xDirection = 1
                self.yDirection = 1

        def update(self):
                self.x = self.x + (self.speed * self.xDirection)
                if self.x > unit or self.x <= 0:
                        self.xDirection *= -1
                        self.x += self.xDirection
                        self.y += self.yDirection
                elif self.y > unit or self.y <= 0:
                        self.yDirection *= -1
                        self.y += yDirection

        def display(self):
                fill(255)
                ellipse((self.xOffset + self.x, self.yOffset + self.
↪y), 6, 6)


if __name__ == '__main__':
        run()
```

3.5 Input

3.5.1 Mouse 1D

Move the mouse left and right to shift the balance. The “mouse_x” variable is used to control both the size and color of the rectangles.

```
from p5 import *

def setup():
        size(720, 400)
        no_stroke()
```

(continues on next page)

(continued from previous page)

```
        rect_mode("CENTER")

def draw():
        background(230)

        r1 = remap(mouse_x, [0, width], [0, height])
        r2 = height - r1

        fill(237, 34, 93, r1)
        rect([width / 2 + r1 / 2, height / 2], r1, r1)

        fill(237, 34, 93, r2)
        rect([width / 2 - r2 / 2, height / 2], r2, r2)

if __name__ == '__main__':
        run()
```

3.5.2 Mouse 2D

Moving the mouse changes the position and size of each box.

```
from p5 import *

def setup():
        size(640, 360)
        no_stroke()
        rect_mode("CENTER")

def draw():
        background(230)

        background(51)
        fill(255, 204)
        rect((mouse_x, height/2), mouse_y/2+10, mouse_y/2+10)

        fill(255, 204)
        inverseX = width - mouse_x
        inverseY = height - mouse_y
        rect((inverseX, height/2), (inverseY/2)+10, (inverseY/
↪2)+10);

if __name__ == '__main__':
        run()
```

3.5.3 Mouse Press

Move the mouse to position the shape. Press the mouse button to invert the color.

```
from p5 import *

def setup():
        size(640, 360)
        no_smooth()
        fill(126)
        background(102)

def draw():
        if mouse_is_pressed:
                stroke(255)
        else:
                stroke(0)

        line((mouse_x-66, mouse_y), (mouse_x+66, mouse_y))
        line((mouse_x, mouse_y-66), (mouse_x, mouse_y+66))

if __name__ == '__main__':
        run()
```

3.5.4 Mouse Signals

Move and click the mouse to generate signals. The top row is the signal from "mouse_x", the middle row is the signal from "mouse_y", and the bottom row is the signal from "mouse_is_pressed".

```
from p5 import *

xvals = None
yvals = None
bvals = None
arrayindex = 0

def setup():
        size(640, 360)
        no_smooth()

        global xvals, yvals, bvals
        xvals = [0]*width
        yvals = [0]*width
        bvals = [0]*width

def draw():
        background(102);

        global xvals, yvals, bvals
        for i in range(1, width):
                xvals[i-1] = xvals[i]
```

(continues on next page)

(continued from previous page)

```
                yvals[i-1] = yvals[i]
                bvals[i-1] = bvals[i]

        xvals[width - 1] = mouse_x
        yvals[width - 1] = mouse_y

        if mouse_is_pressed:
                bvals[width-1] = 0;
        else:
                bvals[width-1] = 255;


        fill(255)
        no_stroke()
        rect((0, height/3), width, height/3+1)

        for i in range(1, width):
                stroke(255)
                point(i, remap(xvals[i], [0, width], [0, height/3 -␣
↪1]))
                stroke(0)
                point(i, height/3 + yvals[i]/3)
                stroke(255)
                line([i, 2*height/3 + bvals[i]/3], [i, (2*height/3␣
↪+ bvals[i - 1]/3)])

if __name__ == '__main__':
        run()
```

3.5.5 Easing

Move the mouse across the screen and the symbol will follow. Between drawing each frame of the animation, the program calculates the difference between the position of the symbol and the cursor. If the distance is larger than 1 pixel, the symbol moves part of the distance (0.05) from its current position toward the cursor.

```
from p5 import *

x = 0
y = 0
easing = 0.05

def setup():
        size(640, 360)
        no_stroke()

def draw():
```

(continues on next page)

(continued from previous page)

```
        background(51)

        global x, y, easing
        targetX = mouse_x
        dx = targetX - x
        x += dx * easing

        targetY = mouse_y
        dy = targetY - y
        y += dy * easing

        ellipse((x, y), 66, 66)

if __name__ == '__main__':
        run()
```

3.5.6 Constrain

Move the mouse across the screen to move the circle. The program constrains the circle to its box.

```
from p5 import *

mx = 0
my = 0
easing = 0.05
radius = 24
edge = 100
inner = edge + radius

def setup():
        size(640, 360)
        no_stroke()
        ellipse_mode("RADIUS")
        rect_mode("CORNER")

def draw():
        background(51)

        global mx, my, easing, radius, edge, inner

        if abs(mouse_x - mx) > 0.1:
                mx = mx + (mouse_x - mx) * easing
        if abs(mouse_y - my) > 0.1:
                my = my + (mouse_y- my) * easing

        mx = constrain(mx, inner, width - inner)
```

(continues on next page)

(continued from previous page)

```
        my = constrain(my, inner, height - inner)
        fill(76)
        rect([edge, edge], width - 2*edge, height - 2*edge)
        fill(255)
        ellipse([mx, my], radius, radius)

if __name__ == '__main__':
        run()
```

3.5.7 Mouse Input

Move the mouse across the screen to change the position of the circles. The positions of the mouse are recorded into an array and played back every frame. Between each frame, the newest value are added to the end of each array and the oldest value is deleted.

```
from p5 import *

num = 60
mx = [0]*num
my = [0]*num

def setup():
        size(640, 360)
        no_stroke()
        fill(255, 153)

def draw():
        background(51)

        global num, mx, my

        which = frame_count % num

        mx[which] = mouse_x
        my[which] = mouse_y

        for i in range(num):
                index = (which+1 + i) % num
                ellipse([mx[index], my[index]], i, i)

if __name__ == '__main__':
        run()
```

3.5.8 Mouse 1D

Click on the box and drag it across the screen.

```
from p5 import *

bx = 0
by = 0
boxSize = 75
overBox = False
locked = False
xOffset = 0.0
yOffset = 0.0

def setup():
        size(640, 360)

        global bx, by
        bx = width/2.0
        by = height/2.0
        rect_mode("RADIUS")

def draw():
        background(0)

        global bx, by, boxSize, overBox, locked, xOffset, yOffset

        # Test if the cursor is over the box
        if (mouse_x > bx-boxSize and mouse_x < bx+boxSize and
                mouse_y > by-boxSize and mouse_y < by+boxSize):
                overBox = True

                if not locked:
                        stroke(255)
                        fill(153)

        else:
                stroke(153)
                fill(153)
                overBox = False

        rect([bx, by], boxSize, boxSize)

def mouse_pressed():
        global bx, by, boxSize, overBox, locked, xOffset, yOffset

        if overBox:
                locked = True
                fill(255, 255, 255)
        else:
                locked = False

        xOffset = mouse_x - bx
        yOffset = mouse_y - by
```

(continues on next page)

(continued from previous page)

```
def mouse_dragged():
        global bx, by, boxSize, overBox, locked, xOffset, yOffset
        if locked:
                bx = mouse_x - xOffset
                by = mouse_y - yOffset

def mouse_released():
        global locked
        locked = False


if __name__ == '__main__':
        run()
```

3.5.9 Keyboard

Click on the image to give it focus and press the letter keys to create forms in time and space. Each key has a unique identifying number. These numbers can be used to position shapes in space.

```
from p5 import *

rectWidth = 0

def setup():
        global rectWidth

        size(640, 360)
        no_stroke()
        background(0)

        rectWidth = width/4

def draw():
        pass

def key_pressed():
        global rectWidth
        keyIndex = -1

        if ord(str(key)) >= ord(str("A")) and ord(str(key)) <=␣
↪ord(str("Z")):
                keyIndex = ord(str(key)) - ord(str("A"))
        elif ord(str(key)) >= ord(str("a")) and ord(str(key)) <=␣
↪ord(str("z")):
                keyIndex = ord(str(key)) - ord(str("a"))
```

(continues on next page)

(continued from previous page)

```
        if keyIndex == -1:
                # If it's not a letter key, clear the screen
                background(0)
        else:
                # It's a letter key, fill a rectangle
                fill(millis()%255)
                x = remap(keyIndex, [0, 25], [0, width - rectWidth])
                rect((x, 0), rectWidth, height)

if __name__ == '__main__':
        run()
```

3.5.10 Keyboard Functions

Keyboard Functions. Modified from code by Martin. Original ‘Color Typewriter’ concept by John Maeda.

Click on the window to give it focus and press the letter keys to type colors. The keyboard function keyPressed() is called whenever a key is pressed. keyReleased() is another keyboard function that is called when a key is released.

```
from p5 import *

maxHeight = 40
minHeight = 20
letterHeight = maxHeight # Height of the letters
letterWidth = 20         # Width of the letter

x = -letterWidth          # X position of the letters
y = 0                     # Y position of the letters

newletter = False
numChars = 26 #  There are 26 characters in the alphabet
colors = []

def setup():
        global colors

        size(640, 360)
        no_stroke()
        color_mode("HSB", numChars)
        background(numChars/2)

        background(0)
        # Set a hue value for each key
        for i in range(numChars):
                colors.append(Color(i, numChars, numChars))
```

(continues on next page)

(continued from previous page)

```
def draw():
        global newletter
        if newletter == True:
                # Draw the "letter"
                y_pos = 0
                if letterHeight == maxHeight:
                        y_pos = y
                        rect((x, y_pos), letterWidth, letterHeight)
                else:
                        y_pos = y + minHeight
                        rect((x, y_pos), letterWidth, letterHeight)
                        fill(numChars/2)
                        rect((x, y_pos - minHeight), letterWidth,␣
↪letterHeight)

                newletter = False

def key_pressed():
        global letterHeight, newletter, x, y
        #  If the key is between 'A'(65) to 'Z' and 'a' to 'z'(122)
        if ord(str(key)) >= ord(str("A")) and ord(str(key)) <=␣
↪ord(str("Z")) or ord(str(key)) >= ord(str("a")) and ord(str(key))
↪<= ord(str("z")):
                keyIndex = 0
                if ord(str(key)) <= ord(str("Z")):
                        keyIndex = ord(str(key)) - ord(str("A"))
                        letterHeight = maxHeight
                        fill(colors[keyIndex])
                else:
                        keyIndex = ord(str(key)) - ord(str("a"))
                        letterHeight = minHeight
                        fill(colors[keyIndex])
        else:
                fill(0)
                letterHeight = 10

        newletter = True
        # Update the "letter" position
        x = x + letterWidth

        # Wrap horizontally
        if x > width - letterWidth:
                x = 0
                y = y + maxHeight

        # Wrap vertically
        if y > height - letterHeight:
                y = 0 # reset y to 0
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
        run()
```

3.5.11 Milliseconds

A millisecond is 1/1000 of a second. Processing keeps track of the number of milliseconds a program has run. By modifying this number with the modulo(%) operator, different patterns in time are created.

```
from p5 import *

scale = 0

def setup():
        size(640, 360)
        no_stroke()

        global scale
        scale = width/20

def draw():
        global scale
        for i in range(int(scale)):
                color_mode("RGB", (i+1) * scale * 10)
                fill(millis()%((i+1) * scale * 10))
                rect([i*scale, 0], scale, height)


if __name__ == '__main__':
        run()
```

3.5.12 Clock

The current time can be read with the second(), minute(), and hour() functions. In this example, sin() and cos() values are used to set the position of the hands.

```
from p5 import *

cx , cy = (0, 0)
secondsRadius = 0
minutesRadius = 0
hoursRadius = 0
clockDiameter = 0

def setup():
```

(continues on next page)

(continued from previous page)

```
        global cx, cy, secondsRadius, minutesRadius, hoursRadius,␣
↪clockDiameter

        size(640, 360)
        stroke(255)

        radius = min(width, height) / 2

        secondsRadius = radius * 0.72
        minutesRadius = radius * 0.60
        hoursRadius = radius * 0.50
        clockDiameter = radius * 1.8

        cx = width / 2
        cy = height / 2

def draw():
        global cx, cy, secondsRadius, minutesRadius, hoursRadius,␣
↪clockDiameter
        background(102)

        # Draw the clock background
        fill(80)
        no_stroke()
        ellipse((cx, cy), clockDiameter, clockDiameter)

        # Angles for sin() and cos() start at 3 o'clock
        # subtract HALF_PI to make them start at the top
        s = remap(second(), [0, 60], [0, TWO_PI]) - HALF_PI
        m = remap(minute() + normalize(second(), 0, 60), [0, 60],␣
↪[0, TWO_PI]) - HALF_PI
        h = remap(hour() + normalize(minute(), 0, 60), [0, 24], [0,␣
↪TWO_PI * 2]) - HALF_PI

        # Draw the hands of the clock
        stroke(255)
        stroke_weight(1)
        line([cx, cy], [cx + cos(s) * secondsRadius, cy + sin(s) *␣
↪secondsRadius])
        stroke_weight(2)
        line([cx, cy], [cx + cos(m) * minutesRadius, cy + sin(m) *␣
↪minutesRadius])
        stroke_weight(4)
        line([cx, cy], [cx + cos(h) * hoursRadius, cy + sin(h) *␣
↪hoursRadius])

        # Draw the minute ticks
        stroke_weight(2)
        begin_shape(POINTS)
```

(continues on next page)

(continued from previous page)

```
        for a in range(0, 360, 6):
                angle = radians(a)
                x = cx + cos(angle) * secondsRadius
                y = cy + sin(angle) * secondsRadius
                vertex(x, y)
        end_shape()


if __name__ == '__main__':
        run()
```

3.6 Form

3.6.1 Points and Lines

Points and lines can be used to draw basic geometry. Change the value of the variable *d* to scale the form. The four variables set the positions based on the value of *d*.

```
from p5 import *

def setup():
        # Sets the screen to be 720 pixels wide and 400 pixels high
        size(720, 400)
        no_loop()

def draw():
        d = 70
        p1 = d
        p2 = p1 + d
        p3 = p2 + d
        p4 = p3 + d

        background(0)
        no_smooth()

        translate(140, 0)

        # Draw gray box
        stroke(153)
        line((p3, p3), (p2, p3))
        line((p2, p3), (p2, p2))
        line((p2, p2), (p3, p2))
        line((p3, p2), (p3, p3))

        # Draw white points
        stroke(255)
```

(continues on next page)

(continued from previous page)

```
        point(p1, p1)
        point(p1, p3)
        point(p2, p4)
        point(p3, p1)
        point(p4, p2)
        point(p4, p4)

if __name__ == '__main__':
        run()
```

3.6.2 Shape Primitives

The basic shape primitive functions are `triangle()`, `rect()`, `quad()`, `ellipse()`, and `arc()`. Squares are made with `rect()` and circles are made with `ellipse()`. Each of these functions requires a number of parameters to determine the shape's position and size.

```
from p5 import *

def setup():
        # Sets the screen to be 720 pixels wide and 400 pixels high
        size(720, 400)
        no_loop()

def draw():
        background(0)
        no_stroke()

        fill(204)
        triangle((18, 18), (18, 360), (81, 360))

        fill(102)
        rect((81, 81), 63, 63)

        fill(204)
        quad((189, 18), (216, 18), (216, 360), (144, 360))

        fill(255)
        ellipse((252, 144), 72, 72)

        fill(204)
        triangle((288, 18), (351, 360), (288, 360))

        fill(255)
        arc((479, 300), 280, 280, PI, TWO_PI)

if __name__ == '__main__':
        run()
```

3.6.3 Pie Chart

Uses the `arc()` function to generate a pie chart from the data stored in an array.

```
from p5 import *

angles = [30, 10, 45, 35, 60, 38, 75, 67]

def setup():
        # Sets the screen to be 720 pixels wide and 400 pixels high
        size(720, 400)
        no_loop()
        no_stroke()

def draw():
        background(100)
        pie_chart(300, angles)

def pie_chart(diameter, data):
        lastAngle = 0
        for i in range(len(data)):
                gray = remap(i, [0, len(data)], [0, 255])
                fill(gray)
                arc(
                        (width / 2, height / 2),
                        diameter,
                        diameter,
                        lastAngle,
                        lastAngle + radians(angles[i])
                )

                lastAngle += radians(angles[i])

if __name__ == '__main__':
        run()
```

3.6.4 Regular Polygon

What is your favorite? Pentagon? Hexagon? Heptagon? No? What about the icosagon? The polygon() function created for this example is capable of drawing any regular polygon. Try placing different numbers into the polygon() function calls within draw() to explore.

```
from p5 import *

def setup():
        size(640, 360)

def draw():
        background(102)
```

(continues on next page)

(continued from previous page)

```
        with push_matrix():
                translate(width*0.2, height*0.5)
                rotate(frame_count / 200.0)
                polygon(0, 0, 82, 3) # Triangle

        with push_matrix():
                translate(width*0.5, height*0.5)
                rotate(frame_count / 50.0)
                polygon(0, 0, 80, 20)


        with push_matrix():
                translate(width*0.8, height*0.5)
                rotate(frame_count / -100.0)
                polygon(0, 0, 70, 7)

def polygon(x, y, radius, npoints):
        angle = TWO_PI / npoints

        begin_shape()
        a = 0
        while a < TWO_PI:
                sx = x + cos(a)*radius
                sy = y + sin(a)*radius
                vertex(sx, sy)

                a = a + angle

        end_shape()


if __name__ == '__main__':
        run()
```

3.6.5 Star

The `star()` function created for this example is capable of drawing a wide range of different forms. Try placing different numbers into the `star()` function calls within draw() to explore.

```
from p5 import *

def setup():
        size(640, 360)

def draw():
        background(102)
```

(continues on next page)

(continued from previous page)

```
        with push_matrix():
                translate(width*0.2, height*0.5)
                rotate(frame_count / 200.0)
                star(0, 0, 5, 70, 3)

        with push_matrix():
                translate(width*0.5, height*0.5)
                rotate(frame_count / 400.0)
                star(0, 0, 80, 100, 40)

        with push_matrix():
                translate(width*0.8, height*0.5)
                rotate(frame_count / -100.0)
                star(0, 0, 30, 70, 5)

def star(x, y, radius1, raduis2, npoints):
        angle = TWO_PI / npoints
        half_angle = angle/2.0

        begin_shape()
        a = 0
        while a < TWO_PI:
                sx = x + cos(a)*raduis2
                sy = y + sin(a)*raduis2
                vertex(sx, sy)
                sx = x + cos(a+half_angle) * radius1
                sy = y + sin(a+half_angle) * radius1
                vertex(sx, sy)
                a = a + angle

        end_shape()


if __name__ == '__main__':
        run()
```

3.6.6 Triangle Strip

Example by Ira Greenberg. Generate a closed ring using the vertex() function and beginShape(TRIANGLE_STRIP) mode. The outsideRadius and insideRadius variables control ring's radii respectively.

```
from p5 import *

x = 0
y = 0
```

(continues on next page)

(continued from previous page)

```
outsideRadius = 150
insideRadius = 100

def setup():
        size(720, 400)
        background(204)

        global x, y
        x = width / 2
        y = height / 2

def draw():
        global x, y, outsideRadius, insideRadius
        background(204)

        numPoints = int(remap(mouse_x, [0, width], [6, 60]))
        angle = 0
        angleStep = 180.0 / numPoints

        begin_shape(TRIANGLE_STRIP)

        for i in range(numPoints + 1):
                px = x + cos(radians(angle)) * outsideRadius
                py = y + sin(radians(angle)) * outsideRadius
                angle += angleStep
                vertex(px, py)

                px = x + cos(radians(angle)) * insideRadius
                py = y + sin(radians(angle)) * insideRadius
                vertex(px, py)

                angle += angleStep

        end_shape()

if __name__ == '__main__':
        run()
```

3.6.7 Bezier

The first two parameters for the `bezier()` function specify the first point in the curve and the last two parameters specify the last point. The middle parameters set the control points that define the shape of the curve.

```
from p5 import *

def setup():
```

(continues on next page)

(continued from previous page)

```
        size(720, 400)
        stroke(255)
        no_fill()

def draw():
        background(0)

        for i in range(0, 200, 20):
                bezier(
                        (mouse_x - i / 2.0, 40 + i),
                        (410, 20),
                        (440, 300),
                        (240 - i / 16.0, 300 + i / 8.0)
                        )

if __name__ == '__main__':
        run()
```

3.7 Image

3.7.1 Load and Display

Images can be loaded and displayed to the screen at their actual size or any other size.

```
from p5 import *

img = None

def setup():
    global img
    size(720, 400)
    img = load_image("moonwalk.jpg")

def draw():
    global img
    background(0)
    image(img, (0, 0))
    image(img, (0, height / 2), (img.width / 2, img.height / 2))

if __name__ == '__main__':
    run()
```

3.7.2 Background Image

This example presents the fastest way to load a background image. To load an image as the background, it must be the same width and height as the program.

```
from p5 import *

bg = None
y = 0

def setup():
    global bg
    size(720, 400)
    bg = load_image("moonwalk.jpg")

def draw():
    global img, y
    background(bg)

    stroke(226, 204, 0)
    line((0, y), (width, y))

    y += 1

    if y > height:
        y = 0

if __name__ == '__main__':
    run()
```

3.7.3 Transparency

Move the pointer left and right across the image to change its position. This program overlays one image over another by modifying the alpha value of the image with the `tint()` function.

```
from p5 import *

img = None
offset = 0
easing = 0.05

def setup():
        global img
        size(720, 400)
        img = load_image("moonwalk.jpg")

def draw():
        global img, offset, easing

```

(continues on next page)

(continued from previous page)

```
        image(img, (0, 0)) #Display at full opacity

        dx = mouse_x - img.width / 2 - offset
        offset += dx * easing
        tint(255, 127)
        image(img, (offset, 0))

if __name__ == '__main__':
        run()
```

3.7.4 Alpha Mask

Loads a "mask" for an image to specify the transparency in different parts of the image. The two images are blended together using the `mask()` method of p5.Image.

```
from p5 import *

img = None
imgMask = None

def setup():
    global img
    size(720, 400)
    img = load_image("moonwalk.jpg")
    imgMask = load_image("mask.png")

    img.mask(imgMask)
    imageMode("CENTER")

def draw():
    global img
    background(0, 102, 153)
    image(img, (width / 2, height / 2))
    image(img, mouse_x, mouse_y)

if __name__ == '__main__':
    run()
```

3.7.5 Create Image

The `create_image()` function provides a fresh buffer of pixels to play with. This example creates an image gradient.

3.7.6 Pointillism

By Dan Shiffman. Mouse horizontal location controls size of dots. Creates a simple pointillist effect using ellipses colored according to pixels in an image.

```
from p5 import *

img = None
small_point = 4
large_point = 40

def setup():
    global img
    size(720, 400)
    no_stroke()
    background(255)
    img = load_image("moonwalk.jpg")


def draw():
    global img, large_point, small_point
    pointillize = remap(mouse_x, [0, width], [small_point, large_
↪point])
    x = floor(random_uniform(img.width))
    y = floor(random_uniform(img.height))

    pix = img._get_pixel((x, y))
    fill(pix, 128)

    ellipse((x, y), pointillize, pointillize)

if __name__ == '__main__':
    run()
```

3.8 Typography

3.8.1 Letters

Draws letters to the screen. This requires loading a font, setting the font, and then drawing the letters.

```
from p5 import *

f = None

def setup():
        global f
```

(continues on next page)

(continued from previous page)

```
        size(640, 360)
        background(0)

        # Create the font
        f = create_font("Arial.ttf", 16)
        text_font(f)
        text_align("CENTER")

def draw():
        background(0)

        # Set the left and top margin
        margin = 10
        translate(margin*4, margin*4)

        gap = 46
        counter = 35

        for y in range(0, height - gap, gap):
                for x in range(0, width - gap, gap):

                        letter = chr(counter)

                        if (letter == "A" or letter == "E" or␣
↪letter == "I" or letter == "O" or letter == "U"):
                                fill(255, 204, 0)
                        else:
                                fill(255)

                        # Draw the letter to the screen
                        text(letter, (x, y))

                        # Increment the counter
                        counter += 1

if __name__ == '__main__':
        run()
```

3.8.2 Words

The `text()` function is used for writing words to the screen. The letters can be aligned left, center, or right with the textAlign() function.

```
from p5 import *

f = None
```

(continues on next page)

(continued from previous page)

```
def setup():
        global f
        size(640, 360)

        # Create the font
        f = create_font("Arial.ttf", 16)
        text_font(f)
        text_align("CENTER")

def draw():
        background(102)
        text_align("RIGHT")
        drawType(width * 0.25)
        text_align("CENTER")
        drawType(width * 0.5)
        text_align("LEFT")
        drawType(width * 0.75)

def drawType(x):
        line((x, 0), (x, 65))
        line((x, 220), (x, height))
        fill(0)
        text("ichi", (x, 95))
        fill(51)
        text("ni", (x, 130))
        fill(204)
        text("san", (x, 165))
        fill(255)
        text("shi", (x, 210))

if __name__ == '__main__':
        run()
```

3.9 3D

3.9.1 Vertex

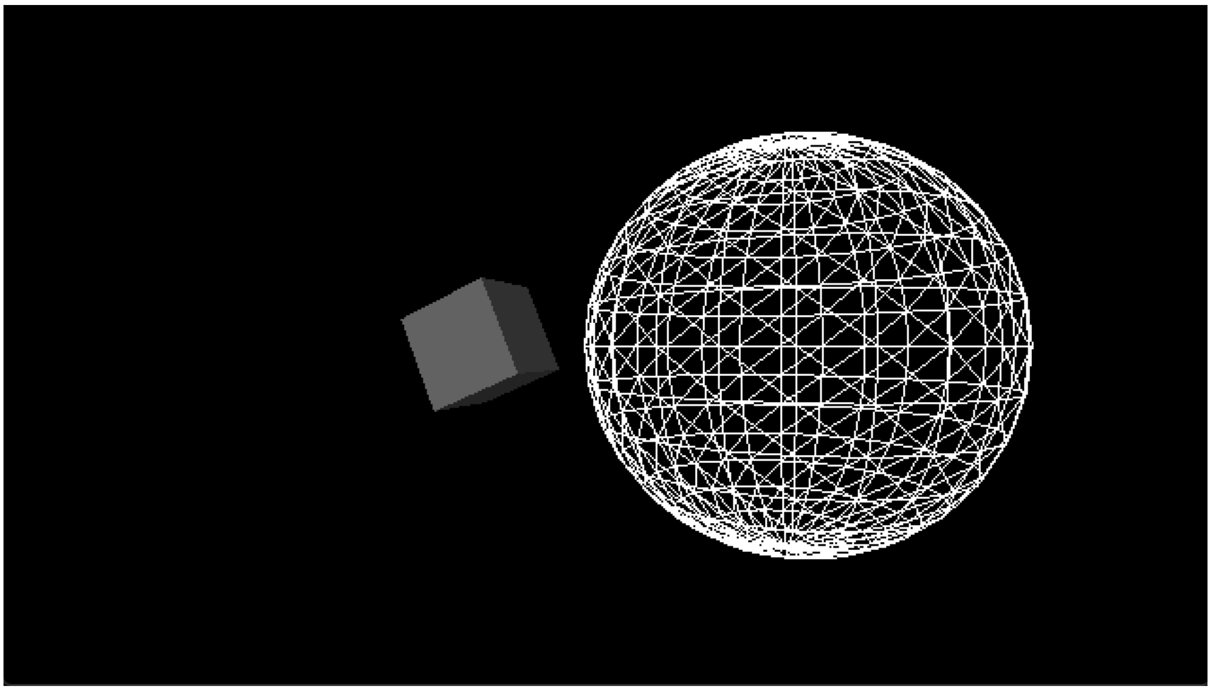
```
# Code from Daniel Shiffman's tutorial, adopted to p5py
# https://processing.org/tutorials/p3d/

from p5 import *

def setup():
    size(640, 360)
    no_loop()
```

(continues on next page)

(continued from previous page)

```
def draw():
    background(0)

    stroke(255)
    rotate_x(PI*3/2)
    rotate_z(PI/6)
    fill(255)
    begin_shape()
    vertex(-100, -100, -100)
    vertex( 100, -100, -100)
    vertex(   0,    0,  100)

    vertex( 100, -100, -100)
    vertex( 100,  100, -100)
    vertex(   0,    0,  100)

    vertex( 100, 100, -100)
    vertex(-100, 100, -100)
    vertex(   0,   0,  100)

    vertex(-100,  100, -100)
    vertex(-100, -100, -100)
    vertex(   0,    0,  100)
    end_shape()

run(mode='P3D')
```

3.9.2 Basic Shapes



```
# Code from Daniel Shiffman's tutorial, adopted to p5py
# https://processing.org/tutorials/p3d/

from p5 import *

def setup():
    size(640, 360)

def draw():
    background(0)
    lights()

    with push_matrix():
        translate(-130, 0, 0)
        rotate_y(1.25)
        rotate_x(-0.4)
        no_stroke()
        fill(255)
        blinn_phong_material()
        box(100, 100, 100)

    with push_matrix():
        translate(250, 0, -200)
        no_fill()
        stroke(255)
        sphere(280)

if __name__ == '__main__':
    run(mode='P3D')
```

3.9.3 Blinn Phong Material

```
from p5 import *

def setup():
    size(720, 400)

def draw():
    background(205, 102, 94)
    rotate_x(frame_count * 0.02)
    rotate_y(frame_count * 0.01)
    blinn_phong_material()
    cone(200, 400)
    locX = mouse_x - width/2
    locY = mouse_y - height/2
    light_specular(0, 0, 255)
    point_light(360, 360*1.5, 360, locX, locY, 400)
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    run(mode='P3D')
```

CHAPTER 4

Guides

4.1 p5 for Processing users

p5 API borrows many core ideas from the Processing so most of the API looks similar. This document lists the major differences between Processing and the p5 API.

In addition to skimming through this document, you should also check the API reference for more details and take a look at complete working examples on the examples repository.

4.1.1 Naming conventions

- Most function names are now in `lowercase_separated_by_underscores` as opposed to the `lowerCamelCase` in Processing. So, if a method was called `bezierPoint` in Processing it will be called `bezier_point` in p5.
- Mathematical constants like π are still in `UPPPERCASE_SEPARATED_BY_UNDERSCORES`. Note that `width`, `height`, `mouse_x`, etc **are not** treated as constants.
- The "P" prefix has been dropped from the class names. So, `PVector` becomes `Vector`, `PImage` becomes `Image`, etc.

We've also renamed a couple of things:

- Processing's `map()` method is now called `remap()` to avoid a namespace conflict with Python's inbuilt `map()` function.
- All `get*()` and `set*()` methods for objects have been removed and attributes can be set/read by directly accessing the objects.

  For instance, if we have a vector `vec` in Processing, we would use

```
/* read the magnitude of the vector */
float m = vec.mag()

/* set the magnitude of the vector */
vec.setMag(newMagnitude)
```

In p5, we can just use:

```
# read the magnitude of the vector
m = vec.magnitude

# set the magnitude of the vector
vec.magnitude = new_magnitude
```

- Processing's `random()` method is now called `random_uniform()` to prevent confusion (and nasty errors!) while using Python's `random` module.

4.1.2 Running Sketches

- p5 doesn't come with an IDE and p5 scripts are run as any other Python scripts/programs. You are free to use any text editor or Python IDE to run your programs.
- Sketches **must** call the `run()` function to actually show the sketches. Sketches without a call to `run()` **will not work**. So, a Processing sketch:

```
void setup() {
    /* things to do in setup */
}

void draw() {
    /* things to do in the draw loop */
}

void mousePressed() {
    /* things to do when the mouse is pressed */
}
```

would look like this in p5:

```
from p5 import *

def setup():
    # Things to do in setup

def draw():
    # Things to do in the draw loop

def mouse_pressed():
    # Things to do when the mouse is pressed.
```

(continues on next page)

(continued from previous page)

```
run() # This is essential!
```

- Drawing commands only work inside functions.
- If you want to control the frame rate of the you need to pass in `frame_rate` asnan optional argument when you run your sketch.

```
from p5 import *

def setup():
    # setup code

def draw():
    # draw code

# run the sketch at 15 frames per second.
run(frame_rate=15)
```

- Processing's `frameRate()` method is called `set_frame_rate()` in p5. To get the current frame rate in the sketch, use the `frame_rate` global variable.

4.1.3 Shapes, etc

- One of the major differences between the Processing and the p5 API is the way coördinate points are handled. With the exception of the *point()* functions, all drawing functions that allow the user to pass in coordinates use tuples.

  Hence, to draw a line from $(100, 100)$ to $(180, 180)$, we would use:

```
start_point = (100, 100)
end_point = (180, 180)

line(start_point, end_point)
```

  To draw a rectangle at $(90, 90)$ with width 100 and height 45, once would use:

```
location = (90, 90)
rect(location, 100, 45)
```

  Technically, any object that supports indexing (lists, p5 Vectors) could be used as the coordinates to the draw calls. Hence, the following code snippet is perfectly valid:

```
start_point = Vector(306, 72)
control_point_1 = Vector(36, 36)
control_point_2 = Vector(324, 324)
end_point = Vector(54, 288)

bezier(start_point, control_point_1, control_point_2, end_point)
```

- Functions like *bezier_point*, *bezier_tangent*, *curve_point*, *curve_tangent*, etc also need the coordinates as iterables. Further, they also return special objects that have x, y, z coordinates.

```
start = Vector(306, 72)
control_1 = Vector(36, 36)
control_2 = Vector(324, 324)
end = Vector(54, 288)

bp = bezier_point(start, control_1, control, end, 0.5)

# The x coordinate of the bezier point:
print(bp.x)

# The y coordinate of the bezier point:
print(bp.y)
```

- Unlike Processing, p5 doesn't have special global constants for "modes". Functions like `ellipse_mode()` and `rect_mode()` take strings (in all caps) as inputs. The following are valid function calls:

```
center = (width / 2, height / 2)

rect_mode('RADIUS')
square(center, 50)

ellipse_mode('CENTER')
circle(center, 100)
```

- Processing's `pushMatrix()` and `popMatrix()` have been replaced by a single `push_matrix()` context manager that cleans up after itself. So, the following Pro-cecssing code:

```
pushMatrix()

translate(width/2, height/2)
point(0, 0)

popMatrix()
```

  Becomes:

```
with push_matrix():
    translate(width / 2, height / 2)
    point(0, 0)
```

- Like `push_matrix()`, `push_style()` is a context manager and can be used with the `with` statement.

4.1.4 Event System

- Processing's `mousePressed` global boolean has been renamed to `mouse_is_pressed` to avoid namespace conflicts with the user defined `mouse_pressed` function.
- To check which mouse button was pressed, compare the `mouse_button` global variable to one of the strings `'LEFT'`, `'RIGHT'`, `'CENTER'`, `'MIDDLE'`
- The `keyCode` variable has been removed. And Processing's special "coded" keys can be compared just like other alpha numeric keys.

```
def key_pressed(event):
    if event.key == 'A':
        # code to run when the <A> key is presesed.

    elif event.key == 'UP':
        # code to run when the <UP> key is presesed.

    elif event.key == 'SPACE':
        # code to run when the <SPACE> key is presesed.

    # ...etc
```

4.1.5 Math

- Vector addition, subtraction, and equality testing are done using the usual mathematical operators and scalar multiplication is done using the usual `*` operator. The following are valid vector operations:

```
# add two vectors `position` and `velocity`
position = position + velocity

# subtract the vector `offset` from `position`
actual_location = position - offset

# scale a vector by a factor of two
scaled_vector = 2 * vec_1

# check if two vectors `player_location`
# and `mouse_location` are equal
if (player_location == mouse_location):
    end_game()

# ...etc.
```

- The mean and standard deviation value can be specified while calling `random_gaussian()`
- The distance function takes in two tuples as inputs. So, the following Processing call:

```
d = dist(x1, y1, z1, x2, y2, z2)
```

would become:

```
point_1 = (x1, y1, z1)
point_2 = (x2, y2, z2)

d = dist(point_1, point_2)
```

- The `remap()` also takes tuples for ranges. The Processing call:

```
n = map(mouseX, 0, width, 0, 10)
```

becomes:

```
source = (0, width)
target = (0, 10)

n = remap(mouse_x, source, target)
```

4.1.6 New Features

- The `title()` method can be used to set the title for the sketch window.
- `circle()` and `square()` functions draw circles and squares.
- `mouse_is_dragging` is a global variable that can be used to check if the mouse is being dragged.
- Colors can be converted to their proper grayscale values.

```
# if we have some color value...
our_color = Color(255, 127, 0)

# ...we can get its gray scale value
# using its `gray` attribute
gray_value = our_color.gray
```

4.2 VS Code Integration

Using VS Code as your code editor for p5py will be a great choice as of pylinter integration and a lot of autocomplete features!

Before you start using VS Code as your code editor for p5py. We reccomend that you turn off some pylint settings, for that you would require a settings file. The setup is simple so do not panic!

4.3 Setup

1. Go into the directory in which keep all of your p5py projects.
2. In that directory create a folder called `.vscode`
3. In the `.vscode` directory create a file called `settings.json`
4. In that file copy and paste all of these json settings:

```
{
    "python.linting.pylintArgs": [
        "--disable", "E0102",
        "--disable", "C0111",
        "--disable", "W0401",
        "--disable", "C0304",
        "--disable", "W0614",
        "--disable", "W0622"
    ]
}
```

5. Save the file and you are ready to write some amazing projects in VS Code and p5py!!

CHAPTER 5

Reference

5.1 Structure

5.1.1 setup()

`p5.`**`setup`**`()`

Called to setup initial sketch options.

The *setup()* function is run once when the program starts and is used to define initial environment options for the sketch.

5.1.2 draw()

`p5.`**`draw`**`()`

Continuously execute code defined inside.

The *draw()* function is called directly after *setup()* and all code inside is continuously executed until the program is stopped (using *exit()*) or *no_loop()* is called.

5.1.3 run()

`p5.`**`run`**`(`*sketch_setup=None*, *sketch_draw=None*, *frame_rate=60*, *mode='P2D'*, *renderer='vispy'*`)`

Run a sketch.

if no *sketch_setup* and *sketch_draw* are specified, p5 automatically "finds" the user-defined setup and draw functions.

Parameters

- **sketch_setup** (*function*) – The setup function of the sketch (None by default.)
- **sketch_draw** (*function*) – The draw function of the sketch (None by default.)
- **frame_rate** (int $\geq$ 1) – The target frame rate for the sketch.

5.1.4 exit()

`p5.exit(*args, **kwargs)`

Exit the sketch.

exit() overrides Python's builtin exit() function and makes sure that necessary cleanup steps are performed before exiting the sketch.

Parameters

- **args** – positional argumets to pass to Python's builtin *exit()* function.
- **kwargs** – keyword-arguments to pass to Python's builtin *exit()* function.

5.1.5 loop()

`p5.loop()`

Make sure *draw()* is being called continuously.

loop() reverts the effects of *no_loop()* and allows *draw()* to be called continously again.

5.1.6 no_loop()

`p5.no_loop()`

Stop draw() from being continuously called.

By default, the sketch continuously calls *draw()* as long as it runs. Calling *no_loop()* stops draw() from being called the next time. Note that this only prevents execution of the code inside *draw()* and the user can manipulate the screen contents through event handlers like *mouse_pressed()*, etc.

5.1.7 redraw()

`p5.redraw()`

Call *draw()* once.

If looping has been disabled using *no_loop()*, *redraw()* will make sure that *draw()* is called *exactly* once.

5.1.8 push_style()

`p5.push_style()`

Save the current style settings and then restores them on exit.

The 'style' information consists of all the parameters controlled by the following functions (the ones indicated by an asterisks '*' aren't available yet):

- background
- fill, no_fill
- stroke, no_stroke
- rect_mode
- ellipse_mode
- shape_mode
- color_mode
- tint
- (*) stroke_weight
- (*) stroke_cap
- (*) stroke_join
- (*) image_mode
- (*) text_align
- (*) text_font
- (*) text_mode
- (*) text_size
- (*) text_leading
- emissive
- specular
- shininess
- ambient
- material

5.2 Environment

5.2.1 size()

`p5.size(width, height)`

Resize the sketch window.

Parameters

- **`width`** (*`int`*) – width of the sketch window.
- **`height`** (*`int`*) – height of the sketch window.

5.2.2 title()

`p5.title(new_title)`

Set the title of the p5 window.

Parameters **`new_title`** (*`str`*) – new title of the window.

5.2.3 height, width

Global integers that store the current width and height of the sketch window.

5.2.4 frame_count

Global integer that keeps track of the current frame number of the sketch i.e., the number of frames that have been drawn since the sketch was started.

5.2.5 frame_rate

Global integer variable that keeps track of the current frame rate of the sketch. The frame rate can only be set when the sketch is run by passing in the optional `frame_rate` keyword argument to the `run()` function. See the `run()` function's reference page for details.

5.3 Attributes

5.3.1 fill()

`p5.fill(*fill_args, **fill_kwargs)`

Set the fill color of the shapes.

Parameters

- **fill_args** (*tuple*) – positional arguments to be parsed as a color.
- **fill_kwargs** (*dict*) – keyword arguments to be parsed as a color.

Returns The fill color.

Return type *Color*

5.3.2 no_fill()

`p5.`**`no_fill`**`()`

Disable filling geometry.

5.3.3 stroke()

`p5.`**`stroke`**`(`**color_args*, ***color_kwargs*`)`

Set the color used to draw lines around shapes

Parameters

- **color_args** (*tuple*) – positional arguments to be parsed as a color.
- **color_kwargs** (*dict*) – keyword arguments to be parsed as a color.

Note Both color_args and color_kwargs are directly sent to Color.parse_color

Returns The stroke color.

Return type *Color*

5.3.4 no_stroke()

`p5.`**`no_stroke`**`()`

Disable drawing the stroke around shapes.

5.3.5 tint()

`p5.`**`tint`**`(`**color_args*, ***color_kwargs*`)`

Set the tint color for the sketch.

Parameters

- **color_args** (*tuple*) – positional arguments to be parsed as a color.
- **color_kwargs** (*dict*) – keyword arguments to be parsed as a color.

Note Both color_args and color_kwargs are directly sent to Color.parse_color

Returns The tint color.

Return type *Color*

5.3.6 no_tint()

p5.**no_tint**()

Disable tinting of images.

5.3.7 stroke_weight()

p5.**stroke_weight**(*thickness*)

Sets the width of the stroke used for lines, points, and the border around shapes. All widths are set in units of pixels.

Parameters **weight** (`int`) – thickness of stroke in pixels

5.3.8 stroke_cap()

p5.**stroke_cap**(*c*)

Sets the style of line endings. The ends are SQUARE, PROJECT, and ROUND. The default cap is ROUND.

Parameters **c** (`string`) – either 'SQUARE', 'PROJECT' or 'ROUND'

5.3.9 stroke_join()

p5.**stroke_join**(*j*)

Sets the style of the joints which connect line segments. These joints are either mitered, beveled, or rounded and specified with the corresponding parameters MITER, BEVEL, and ROUND. The default joint is MITER.

Parameters **weight** – either 'MITER', 'BEVEL' or 'ROUND'

5.4 Shape

5.4.1 PShape

class p5.**PShape**(*fill_color='auto', stroke_color='auto', stroke_weight='auto', stroke_join='auto', stroke_cap='auto', visible=False, children=None, contours=(), vertices=(), shape_type=<SType.TESS: 'TESS'>*)

Custom shape class for p5.

Parameters

- **`vertices`** (*`list | np.ndarray`*) – List of (polygonal) vertices for the shape.
- **`fill_color`** (*`'auto' | None | tuple | p5.Color`*) – Fill color of the shape (default: 'auto' i.e., the current renderer fill)
- **`stroke_color`** (*`'auto' | None | tuple | p5.color`*) – Stroke color of the shape (default: 'auto' i.e., the current renderer stroke color)
- **`visible`** (*`bool`*) – toggles shape visibility (default: False)
- **`children`** (*`list`*) – List of sub-shapes for the current shape (default: [])

`add_child`(*child*)
Add a child shape to the current shape

Parameters **`child`** (`PShape`) – Child to be added

`add_vertex`(*vertex*)
Add a vertex to the current shape

Parameters **`vertex`** (*`tuple | list | p5.Vector | np.ndarray`*) – The (next) vertex to add to the current shape.

`apply_matrix`(*mat*)
Apply the given transformation matrix to the shape.

Parameters **`mat`** (*`(4, 4) np.ndarray`*) – the 4x4 matrix to be applied to the current shape.

`child_count`
Number of children.

Returns The current number of children.

Return type int

`edit`(*reset=True*)
Put the shape in edit mode.

Parameters **`reset`** (*`bool`*) – Toggles whether the shape should be "reset" during editing. When set to *True* all existing shape vertices are cleared. When set to *False* the new vertices are appended at the end of the existing vertex list. (default: True)

Raises **`ValueError`** – if the shape is already being edited.

`reset_matrix`()
Reset the transformation matrix associated with the shape.

`rotate`(*theta*, *axis=(0, 0, 1)*)
Rotate the shape by the given angle along the given axis.

Parameters

- **`theta`** (*`float`*) – The angle by which to rotate (in radians)
- **`axis`** (*`np.ndarray | list`*) – The axis along which to rotate (defaults to the z-axis)

Returns The rotation matrix used to apply the transformation.

Return type np.ndarray

`rotate_x`(*theta*)
Rotate the shape along the x axis.

Parameters **`theta`** (*`float`*) – angle by which to rotate (in radians)

Returns The rotation matrix used to apply the transformation.

Return type np.ndarray

`rotate_y`(*theta*)
Rotate the shape along the y axis.

Parameters **`theta`** (*`float`*) – angle by which to rotate (in radians)

Returns The rotation matrix used to apply the transformation.

Return type np.ndarray

`rotate_z`(*theta*)
Rotate the shape along the z axis.

Parameters **`theta`** (*`float`*) – angle by which to rotate (in radians)

Returns The rotation matrix used to apply the transformation.

Return type np.ndarray

`scale`(*sx*, *sy=None*, *sz=None*)
Scale the shape by the given factor.

Parameters

- **`sx`** (*`float`*) – scale factor along the x-axis.
- **`sy`** (*`float`*) – scale factor along the y-axis (defaults to None)
- **`sz`** (*`float`*) – scale factor along the z-axis (defaults to None)

Returns The transformation matrix used to appy the transformation.

Return type np.ndarray

`shear_x`(*theta*)
Shear shape along the x-axis.

Parameters **`theta`** (*`float`*) – angle to shear by (in radians)

Returns The shear matrix used to apply the tranformation.

Return type np.ndarray

shear_y (*theta*)
Shear shape along the y-axis.

Parameters **theta** (`float`) – angle to shear by (in radians)

Returns The shear matrix used to apply the transformation.

Return type np.ndarray

translate (*x*, *y*, *z=0*)
Translate the shape origin to the given location.

Parameters

- **x** (`int`) – The displacement amount in the x-direction (controls the left/right displacement)
- **y** (`int`) – The displacement amount in the y-direction (controls the up/down displacement)
- **z** (`int`) – The displacement amount in the z-direction (0 by default). This controls the displacement away-from/towards the screen.

Returns The translation matrix applied to the transform matrix.

Return type np.ndarray

update_vertex (*idx*, *vertex*)
Edit an individual vertex.

Parameters

- **idx** (`int`) – index of the vertex to be edited
- **vertex** (`tuple | list | p5.Vector | np.ndarray`) – The (next) vertex to add to the current shape.

5.4.2 2D Primitives

point()

p5.**point** (*x*, *y*, *z=0*)
Returns a point.

Parameters

- **x** (`int or float`) – x-coordinate of the shape.
- **y** (`int or float`) – y-coordinate of the shape.
- **z** (`int or float`) – z-coordinate of the shape (defaults to 0).

Returns A point PShape.

Return type *PShape*

line()

p5.**line**(*p1*, *p2*)

Returns a line.

Parameters

- **p1** (*tuple*) – Coordinates of the starting point of the line.
- **p2** (*tuple*) – Coordinates of the end point of the line.

Returns A line PShape.

Return type *PShape*

ellipse()

p5.**ellipse**(*coordinate*, **args*, *mode=None*)

Return a ellipse.

Parameters

- **coordinate** (*3-tuple*) – Represents the center of the ellipse when mode is 'CENTER' (the default) or 'RADIUS', the lower-left corner of the ellipse when mode is 'CORNER' or, and an arbitrary corner when mode is 'CORNERS'.
- **args** – For modes'CORNER' or 'CENTER' this has the form (width, height); for the 'RADIUS' this has the form (x_radius, y_radius); and for the 'CORNERS' mode, args should be the corner opposite to *coordinate*.
- **mode** (*str*) – The drawing mode for the ellipse. Should be one of {'CORNER', 'CORNERS', 'CENTER', 'RADIUS'} (defaults to the mode being used by the p5.renderer.)

Type tuple

Returns An ellipse

Return type Arc

circle()

p5.**circle**(*coordinate*, *radius*, *mode=None*)

Return a circle.

Parameters

- **coordinate** (*3-tuple*) – Represents the center of the ellipse when mode is 'CENTER' (the default) or 'RADIUS', the lower-left corner of the ellipse when mode is 'CORNER' or, and an arbitrary corner when mode is 'CORNERS'.

- **radius** – For modes'CORNER' or 'CENTER' this actually represents the diameter; for the 'RADIUS' this represents the radius.
- **mode** (*str*) – The drawing mode for the ellipse. Should be one of {'CORNER', 'CORNERS', 'CENTER', 'RADIUS'} (defaults to the mode being used by the p5.renderer.)

Type tuple

Returns A circle.

Return type Ellipse

Raises **ValueError** – When mode is set to 'CORNERS'

arc()

`p5.arc`(*coordinate*, *width*, *height*, *start_angle*, *stop_angle*, *mode=None*, *ellipse_mode=None*)

Return a ellipse.

Parameters

- **coordinate** – Represents the center of the arc when mode is 'CENTER' (the default) or 'RADIUS', the lower-left corner of the ellipse when mode is 'CORNER'.
- **width** (*float*) – For ellipse modes 'CORNER' or 'CENTER' this represents the width of the the ellipse of which the arc is a part. Represents the x-radius of the parent ellipse when ellipse mode is 'RADIUS
- **height** (*float*) – For ellipse modes 'CORNER' or 'CENTER' this represents the height of the the ellipse of which the arc is a part. Represents the y-radius of the parent ellipse when ellipse mode is 'RADIUS
- **mode** (*str*) – The mode used to draw an arc can be any of {None, 'OPEN', 'CHORD', 'PIE'}.
- **ellipse_mode** – The drawing mode used for the ellipse. Should be one of {'CORNER', 'CENTER', 'RADIUS'} (defaults to the mode being used by the p5.renderer.)

Rtype coordinate 3-tuple

Returns An arc.

Return type Arc

triangle()

`p5.triangle`(*p1*, *p2*, *p3*)

Return a triangle.

Parameters

- **p1** (*tuple | list | p5.Vector*) – coordinates of the first point of the triangle
- **p2** (*tuple | list | p5.Vector*) – coordinates of the second point of the triangle
- **p3** (*tuple | list | p5.Vector*) – coordinates of the third point of the triangle

Returns A triangle.

Return type *p5.PShape*

quad()

`p5.`**`quad`**(*p1*, *p2*, *p3*, *p4*)

Return a quad.

Parameters

- **p1** (*tuple | list | p5.Vector*) – coordinates of the first point of the quad
- **p2** (*tuple | list | p5.Vector*) – coordinates of the second point of the quad
- **p3** (*tuple | list | p5.Vector*) – coordinates of the third point of the quad
- **p4** (*tuple | list | p5.Vector*) – coordinates of the fourth point of the quad

Returns A quad.

Return type *PShape*

rect()

`p5.`**`rect`**(*coordinate*, **args*, *mode=None*)

Return a rectangle.

Parameters

- **coordinate** (*tuple | list | p5.Vector*) – Represents the lower left corner of then rectangle when mode is ‘CORNER’, the center of the rectangle when mode is ‘CENTER’ or ‘RADIUS’, and an arbitrary corner when mode is ‘CORNERS’
- **args** – For modes’CORNER’ or ‘CENTER’ this has the form (width, height); for the ‘RADIUS’ this has the form (half_width, half_height); and for the ‘CORNERS’ mode, args should be the corner opposite to *coordinate*.

- **mode** (*str*) – The drawing mode for the rectangle. Should be one of {'CORNER', 'CORNERS', 'CENTER', 'RADIUS'} (defaults to the mode being used by the p5.renderer.)

Type tuple

Returns A rectangle.

Return type *p5.PShape*

square()

`p5.square`(*coordinate*, *side_length*, *mode=None*)

Return a square.

Parameters

- **coordinate** (`tuple | list | p5.Vector`) – When mode is set to 'CORNER', the coordinate represents the lower-left corner of the square. For modes 'CENTER' and 'RADIUS' the coordinate represents the center of the square.
- **side_length** (`int or float`) – The side_length of the square (for modes 'CORNER' and 'CENTER') or hald of the side length (for the 'RADIUS' mode)
- **mode** (*str*) – The drawing mode for the square. Should be one of {'CORNER', 'CORNERS', 'CENTER', 'RADIUS'} (defaults to the mode being used by the p5.renderer.)

Returns A rectangle.

Return type *p5.PShape*

Raises **ValueError** – When the mode is set to 'CORNERS'

5.4.3 Curves

bezier()

`p5.bezier`(*start*, *control_point_1*, *control_point_2*, *stop*)

Return a bezier path defined by two control points.

Parameters

- **start** (`tuple.`) – The starting point of the bezier curve.
- **control_point_1** (`tuple.`) – The first control point of the bezier curve.
- **control_point_2** (`tuple.`) – The second control point of the bezier curve.
- **stop** (`tuple.`) – The end point of the bezier curve.

Returns A bezier path.

Return type PShape.

bezier_detail()

`p5.`**`bezier_detail`**(*detail_value*)

Change the resolution used to draw bezier curves.

Parameters **`detail_value`** (*`int`*) – New resolution to be used.

bezier_point()

`p5.`**`bezier_point`**(*start*, *control_1*, *control_2*, *stop*, *parameter*)

Return the coordinate of a point along a bezier curve.

Parameters

- **`start`** (*`3-tuple.`*) – The start point of the bezier curve
- **`control_1`** (*`3-tuple.`*) – The first control point of the bezier curve
- **`control_2`** (*`3-tuple.`*) – The second control point of the bezier curve
- **`stop`** (*`3-tuple.`*) – The end point of the bezier curve
- **`parameter`** (*`float`*) – The parameter for the required location along the curve. Should be in the range [0.0, 1.0] where 0 indicates the start of the curve and 1 indicates the end of the curve.

Returns The coordinate of the point along the bezier curve.

Return type Point (namedtuple with x, y, z attributes)

bezier_tangent()

`p5.`**`bezier_tangent`**(*start*, *control_1*, *control_2*, *stop*, *parameter*)

Return the tangent at a point along a bezier curve.

Parameters

- **`start`** (*`3-tuple.`*) – The start point of the bezier curve
- **`control_1`** (*`3-tuple.`*) – The first control point of the bezier curve
- **`control_2`** (*`3-tuple.`*) – The second control point of the bezier curve
- **`stop`** (*`3-tuple.`*) – The end point of the bezier curve

- **parameter** (*float*) – The parameter for the required tangent location along the curve. Should be in the range [0.0, 1.0] where 0 indicates the start of the curve and 1 indicates the end of the curve.

Returns The tangent at the required point along the bezier curve.

Return type Point (namedtuple with x, y, z attributes)

curve()

p5.**curve**(*point_1*, *point_2*, *point_3*, *point_4*)

Return a Catmull-Rom curve defined by four points.

Parameters

- **point_1** (*tuple*) – The first point of the curve.
- **point_2** (*tuple*) – The first point of the curve.
- **point_3** (*tuple*) – The first point of the curve.
- **point_4** (*tuple*) – The first point of the curve.

Returns A curved path.

Return type *PShape*

curve_detail()

p5.**curve_detail**(*detail_value*)

Change the resolution used to draw bezier curves.

Parameters **detail_value** (*int*) – New resolution to be used.

curve_point()

p5.**curve_point**(*point_1*, *point_2*, *point_3*, *point_4*, *parameter*)

Return the coordinates of a point along a curve.

Parameters

- **point_1** (*3-tuple.*) – The first control point of the curve.
- **point_2** (*3-tuple.*) – The second control point of the curve.
- **point_3** (*3-tuple.*) – The third control point of the curve.
- **point_4** (*3-tuple.*) – The fourth control point of the curve.
- **parameter** (*float*) – The parameter for the required point location along the curve. Should be in the range [0.0, 1.0] where 0 indicates the start of the curve and 1 indicates the end of the curve.

Returns The coordinate of the point at the required location along the curve.

Return type Point (namedtuple with x, y, z attributes)

curve_tangent()

p5.**curve_tangent**(*point_1*, *point_2*, *point_3*, *point_4*, *parameter*)

Return the tangent at a point along a curve.

Parameters

- **point_1** (*3-tuple.*) – The first control point of the curve.
- **point_2** (*3-tuple.*) – The second control point of the curve.
- **point_3** (*3-tuple.*) – The third control point of the curve.
- **point_4** (*3-tuple.*) – The fourth control point of the curve.
- **parameter** (*float*) – The parameter for the required tangent location along the curve. Should be in the range [0.0, 1.0] where 0 indicates the start of the curve and 1 indicates the end of the curve.

Returns The tangent at the required point along the curve.

Return type Point (namedtuple with x, y, z attributes)

curve_tightness()

p5.**curve_tightness**(*amount*)

Change the curve tightness used to draw curves.

Parameters **amount** (*int*) – new curve tightness amount.

5.4.4 Attributes

ellipse_mode()

p5.**ellipse_mode**(*mode='CENTER'*)

Change the ellipse and circle drawing mode for the p5.renderer.

Parameters **mode** (*str*) – The new mode for drawing ellipses. Should be one of {'CORNER', 'CORNERS', 'CENTER', 'RADIUS'}. This defaults to 'CENTER' so calling ellipse_mode without parameters will reset the sketch's ellipse mode.

rect_mode()

p5.**rect_mode**(*mode='CORNER'*)

Change the rect and square drawing mode for the p5.renderer.

Parameters **mode** (*str*) – The new mode for drawing rects. Should be one of {'CORNER', 'CORNERS', 'CENTER', 'RADIUS'}. This defaults to 'CORNER' so calling rect_mode without parameters will reset the sketch's rect mode.

5.4.5 3D Primitives

box()

`p5.`**`box`**(*width*, *height*, *depth*, *detail_x=1*, *detail_y=1*)
Draw a plane with given a width and height

Parameters

- **width** (*float*) – width of the box
- **height** (*float*) – height of the box
- **depth** (*float*) – depth of the box
- **detail_x** (*integer*) – Optional number of triangle subdivisions in x-dimension. Default is 1
- **detail_y** (*integer*) – Optional number of triangle subdivisions in y-dimension. Default is 1

plane()

`p5.`**`plane`**(*width*, *height*, *detail_x=1*, *detail_y=1*)
Draw a plane with given a width and height

Parameters

- **width** (*float*) – width of the plane
- **height** (*float*) – height of the plane
- **detail_x** (*integer*) – Optional number of triangle subdivisions in x-dimension. Default is 1
- **detail_y** (*integer*) – Optional number of triangle subdivisions in y-dimension. Default is 1

sphere()

`p5.`**`sphere`**(*radius=50*, *detail_x=24*, *detail_y=16*)
Draw a sphere with given radius

Parameters

- **radius** (*float*) – radius of circle

- **detail_x** (*integer*) – Optional number of triangle subdivisions in x-dimension. Default is 24
- **detail_y** (*integer*) – Optional number of triangle subdivisions in y-dimension. Default is 16

ellipsoid()

`p5.`**`ellipsoid`**`(`*radius_x*, *radius_y*, *radius_z*, *detail_x=24*, *detail_y=24*`)`

Draw an ellipsoid with given radius

Parameters

- **radius_x** (*float*) – x-radius of ellipsoid
- **radius_y** (*float*) – y-radius of ellipsoid
- **radius_z** (*float*) – z-radius of ellipsoid
- **detail_x** (*integer*) – Optional number of triangle subdivisions in x-dimension. Default is 24
- **detail_y** (*integer*) – Optional number of triangle subdivisions in y-dimension. Default is 16

cylinder()

`p5.`**`cylinder`**`(`*radius=50*, *height=50*, *detail_x=24*, *detail_y=1*, *top_cap=True*, *bottom_cap=True*`)`

Draw a cylinder with given radius and height

Parameters

- **radius** (*float*) – radius of the surface
- **height** (*float*) – height of the cylinder
- **detail_x** (*integer*) – Number of segments, the more segments the smoother geometry. Default is 24
- **detail_y** (*integer*) – number of segments in y-dimension, the more segments the smoother geometry. Default is 1
- **bottom_cap** (*boolean*) – whether to draw the bottom of the cylinder
- **top_cap** (*boolean*) – whether to draw the top of the cylinder

cone()

`p5.`**`cone`**`(`*radius=50*, *height=50*, *detail_x=24*, *detail_y=1*, *cap=True*`)`

Draw a cone with given radius and height

Parameters

- **radius** (*float*) – radius of the bottom surface
- **height** (*float*) – height of the cone
- **detail_x** (*integer*) – Optional number of triangle subdivisions in x-dimension. Default is 24
- **detail_y** (*integer*) – Optional number of triangle subdivisions in y-dimension. Default is 1

torus()

p5.**torus**(*radius=50*, *tube_radius=10*, *detail_x=24*, *detail_y=16*)

Draws torus on the window

Parameters

- **radius** (*float*) – radius of the whole ring
- **tube_radius** (*float*) – radius of the tube
- **detail_x** (*integer*) – Optional number of triangle subdivisions in x-dimension. Default is 24
- **detail_y** (*integer*) – Optional number of triangle subdivisions in y-dimension. Default is 16

5.4.6 Vertex

begin_shape()

p5.**begin_shape**(*kind=<SType.TESS: 'TESS'>*)

Begin shape drawing. This is a helpful way of generating custom shapes quickly.

Parameters **kind** (*SType*) – TESS, POINTS, LINES, TRIANGLES, TRIANGLE_FAN, TRIANGLE_STRIP, QUADS, or QUAD_STRIP; defaults to TESS

end_shape()

p5.**end_shape**(*mode=''*)

The endShape() function is the companion to beginShape() and may only be called after beginShape(). When endshape() is called, all of image data defined since the previous call to beginShape() is rendered.

Parameters **mode** (*str*) – use CLOSE to close the shape

begin_contour()

`p5.begin_contour()`

Use the beginContour() and endContour() functions to create negative shapes within shapes such as the center of the letter ‘O’. beginContour() begins recording vertices for the shape and endContour() stops recording. The vertices that define a negative shape must “wind” in the opposite direction from the exterior shape. First draw vertices for the exterior clockwise order, then for internal shapes, draw vertices shape in counter-clockwise.

end_contour()

`p5.end_contour()`

Ends the current contour.

For more info, see `begin_contour`.

vertex()

`p5.vertex(x, y, z=0)`

All shapes are constructed by connecting a series of vertices. vertex() is used to specify the vertex coordinates for points, lines, triangles, quads, and polygons. It is used exclusively within the beginShape() and endShape() functions.

Parameters

- **x** (*float*) – x-coordinate of the vertex
- **y** (*float*) – y-coordinate of the vertex
- **z** (*float*) – z-coordinate of the vertex

curve_vertex()

`p5.curve_vertex(x, y, z=0)`

Specifies vertex coordinates for curves. The first and last points in a series of curveVertex() lines will be used to guide the beginning and end of a the curve. A minimum of four points is required to draw a tiny curve between the second and third points. Adding a fifth point with curveVertex() will draw the curve between the second, third, and fourth points. The curveVertex() function is an implementation of Catmull-Rom splines.

Parameters

- **x** (*float*) – x-coordinate of the vertex
- **y** (*float*) – y-coordinate of the vertex
- **z** (*float*) – z-coordinate of the vertex

bezier_vertex()

p5.**bezier_vertex**(*x2*, *y2*, *x3*, *y3*, *x4*, *y4*)

Specifies vertex coordinates for Bezier curves

Parameters

- **x2** (*float*) – x-coordinate of the first control point
- **y2** (*float*) – y-coordinate of the first control point
- **x3** (*float*) – x-coordinate of the second control point
- **y3** (*float*) – y-coordinate of the second control point
- **x4** (*float*) – x-coordinate of the anchor point
- **y4** (*float*) – y-coordinate of the anchor point

quadratic_vertex()

p5.**quadratic_vertex**(*cx*, *cy*, *x3*, *y3*)

Specifies vertex coordinates for quadratic Bezier curves

Parameters

- **cx** (*float*) – x-coordinate of the control point
- **cy** (*float*) – y-coordinate of the control point
- **x3** (*float*) – x-coordinate of the anchor point
- **y3** (*float*) – y-coordinate of the anchor point

5.5 Input

All user defined event handlers (*mouse_pressed*, *key_pressed*, etc) can be defined to accept an optional event object as a positional argument. If the user doesn't want to use extra information about the event, they can simply define the handler *without* the positional argument:

```
def mouse_pressed():
    # things to do when the mouse is pressed.
```

If, however, the user would like to extract more information from the event, they can define their function as follows:

```
def mouse_pressed(event):
    # things to do when the mouse is pressed.
```

This event object has special methods that can be used to access additional details about the event. All events support the following methods:

- `event.is_shift_down()`: Returns `True` when the shift key is held down when the event occurs.
- `event.is_ctrl_down()`: Returns `True` when the ctrl key is held down when the event occurs.
- `event.is_alt_down()`: Returns `True` when the alt key is held down when the event occurs.
- `event.is_meta_down()`: Returns `True` when the meta key is held down when the event occurs.

For mouse events, this event object has some more additional attributes:

- `event.x`: The x position of the mouse at the time of the event.
- `event.y`: The y position of the mouse at the time of the event.
- `event.position`: A named tuple that stores the position of the mouse at the time of the event. The x and the y positions can also be accessed using `event.position.x` and `event.position.y` respectively.
- `event.change`: A named tuple that stores the changes (if any) in the mouse position at the time of the event. The changes in the x and the y direction can be accessed using `event.change.x` and `event.change.y`.
- `event.scroll`: A named tuple that stores the scroll amount (if any) at the mouse position at the time of the event. To access the scroll amount in the x and the y direction, use `event.scroll.x` and `event.scroll.y` respectively.
- `event.count`: An integer that stores the scroll in the y direction at the time of the event. Positive values indicate "scroll-up" and negative values "scroll-down".
- `event.button`: The state of the mouse button(s) at the time of the event. It's behavior is similar to the `mouse_button` global variable.
- `event.action`: Stores the "action type" of the current event. Depending on the event, this can take on one of the following values:
  - `'PRESS'`
  - `'RELEASE'`
  - `'CLICK'`
  - `'DRAG'`
  - `'MOVE'`
  - `'ENTER'`
  - `'EXIT'`
  - `'WHEEL'`

For key events, the event object has the following attributes:

- `event.action`: Stores the "action type" for the current key event. This can take on the following values:

  - `'PRESS'`
  - `'RELEASE'`
  - `'TYPE'`

- `event.key`: Stores the key associated with the current key event. Its behavior is similar to the global *key* object.

5.5.1 Mouse

mouse_moved()

The user defined event handler that is called when the mouse is moved.

mouse_pressed()

The user defined event handler that is called when any mouse button is pressed.

mouse_released()

The user defined event handler called when a mouse button is released.

mouse_clicked()

The user defined event handler that is called whent the mouse has been clicked. The `mouse_clicked` handler is called after the mouse has been pressed and then released.

mouse_dragged()

The user defined event handler called when the mouse is being dragged. Note that this is called *once* when the mouse has started dragging. To check if the mouse is still being dragged use the `mouse_is_dragging` global variable.

mouse_wheel()

The user defined event handler that is called when the mouse scroll wheel is moved.

mouse_is_pressed

mouse_is_pressed is a global boolean that stores whether or not a mouse button is curretnly being pressed. More information about the actual button being pressed is stored in the *mouse_button* global variable. It is set to *True* when any mouse button is held down and is *False* otherwise.

```
if mouse_is_pressed:
    # code to run when the mouse button is held down.
```

mouse_is_dragging

mouse_is_dragging is a global boolean that stores whether or not the mouse is currently being dragged. When the mouse is being dragged, this variable is set to *True* and has a value of *False* otherwise.

```
if mouse_is_dragging:
    # code to run when the mouse is being dragged.
```

mouse_button

mouse_button is a global object that stores information about the current mouse button that is being held down. If no button is being held down, *mouse_button* is set to *None*. *mouse_button* can be compared to the strings *'MIDDLE'*, *'CENTER'*, *'LEFT'*, or *'RIGHT'* to check which mouse button is being held down.

```
if mouse_button == 'CENTER':
    # code to run when the middle mouse button is pressed.

elif mouse_button == 'LEFT':
    # code to run when the left mouse button is pressed.

elif mouse_button == 'RIGHT':
    # code to run when the right mouse button is pressed.
```

mouse_x, mouse_y

Global variables that store the x and the y positions of the mouse for the **current** draw call.

pmouse_x, pmouse_y

Global variables that store the x and the y positions of the mouse for the **last** draw call.

5.5.2 Keyboard

key

A global variable that keeps track of the current key being pressed (if any). This is set to `None` when no key is being pressed. This can be compared to different strings to get more information about the key. These strings should be the names of the keys — like 'ENTER', 'BACKSPACE', 'A', etc — and should always be in uppercase.

For instance:

```
if key == 'UP':
    # things to do when the up-key is held down.

elif key == 'ENTER':
    # things to do when the enter/return key is pressed

elif key == '1':
    # things to do when the "1" key is pressed.

# etc...
```

key_is_pressed

A global boolean that keeps track of whether *any* key is being held down. This is set to `True` is some key is held down and `False` otherwise.

key_pressed()

A user defined event handler that is called when a key is pressed.

key_released()

A user defined event handler that is called when a key is released.

key_typed()

A user defined event handler that is called when a key is typed.

5.6 Output

5.6.1 Image

save()

`p5.`**`save`**(*filename='screen.png'*)

Save an image from the display window.

Saves an image from the display window. Append a file extension to the name of the file, to indicate the file format to be used.If no extension is included in the filename, the image will save in PNG format and .png will be added to the name. By default, these files are saved to the folder where the sketch is saved. Alternatively, the files can be saved to any location on the computer by using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows).

Parameters **`filename`** (*`str`*) – Filename of the image (defaults to `screen.png`)

save_frame()

`p5.`**`save_frame`**(*filename='screen.png'*)

Save a numbered sequence of images whenever the function is run.

Saves a numbered sequence of images, one image each time the function is run. To save an image that is identical to the display window, run the function at the end of *`p5.draw()`* or within mouse and key events such as `p5.mouse_pressed()` and `p5.key_pressed()`.

If save_frame() is used without parameters, it will save files as screen-0000.png, screen-0001.png, and so on. Append a file extension, to indicate the file format to be used. Image files are saved to the sketch's folder. Alternatively, the files can be saved to any location on the computer by using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows).

Parameters **`filename`** (*`str`*) – name (or name with path) of the image file. (defaults to `screen.png`)

5.7 Transform

5.7.1 push_matrix()

`p5.`**`push_matrix`**()

5.7.2 print_matrix()

`p5.`**`print_matrix`**()

Print the transform matrix being used by the sketch.

5.7.3 reset_matrix()

`p5.`**`reset_matrix`**()

Reset the current transform matrix.

5.7.4 rotate()

`p5.`**`rotate`**()

Rotate the display by the given angle along the given axis.

Parameters

- **theta** (*float*) – The angle by which to rotate (in radians)
- **axis** (*np.ndarray or list*) – The axis along which to rotate (defaults to the z-axis)

Returns The rotation matrix used to apply the transformation.

Return type np.ndarray

5.7.5 rotate_x()

`p5.`**`rotate_x`**`()`

Rotate the view along the x axis.

Parameters **theta** (*float*) – angle by which to rotate (in radians)

Returns The rotation matrix used to apply the transformation.

Return type np.ndarray

5.7.6 rotate_y()

`p5.`**`rotate_y`**`()`

Rotate the view along the y axis.

Parameters **theta** (*float*) – angle by which to rotate (in radians)

Returns The rotation matrix used to apply the transformation.

Return type np.ndarray

5.7.7 rotate_z()

`p5.`**`rotate_z`**`()`

Rotate the view along the z axis.

Parameters **theta** (*float*) – angle by which to rotate (in radians)

Returns The rotation matrix used to apply the transformation.

Return type np.ndarray

5.7.8 scale()

`p5.`**`scale`**`()`

Scale the display by the given factor.

Parameters

- **sx** (*float*) – scale factor along the x-axis.

- **sy** (*float*) – scale factor along the y-axis (defaults to None)
- **sz** (*float*) – scale factor along the z-axis (defaults to None)

Returns The transformation matrix used to appy the transformation.

Return type np.ndarray

5.7.9 shear_x()

`p5.shear_x()`

Shear display along the x-axis.

Parameters **theta** (*float*) – angle to shear by (in radians)

Returns The shear matrix used to apply the tranformation.

Return type np.ndarray

5.7.10 shear_y()

`p5.shear_y()`

Shear display along the y-axis.

Parameters **theta** (*float*) – angle to shear by (in radians)

Returns The shear matrix used to apply the transformation.

Return type np.ndarray

5.7.11 translate()

`p5.translate()`

Translate the display origin to the given location.

Parameters

- **x** (*int*) – The displacement amount in the x-direction (controls the left/right displacement)
- **y** (*int*) – The displacement amount in the y-direction (controls the up/down displacement)
- **z** (*int*) – The displacement amount in the z-direction (0 by default). This controls the displacement away-from/towards the screen.

Returns The translation matrix applied to the transform matrix.

Return type np.ndarray

5.7.12 camera()

`p5.camera()`

Sets the camera position for a 3D sketch. Parameters for this function define the position for the camera, the center of the sketch (where the camera is pointing), and an up direction (the orientation of the camera).

When called with no arguments, this function creates a default camera equivalent to:

```
camera((0, 0, height / math.tan(math.pi / 6))),
       (0, 0, 0),
       (0, 1, 0))
```

Parameters

- **position** (*tuple*) – camera position coordinates
- **target_position** (*tuple*) – target position of camera in world coordinates
- **up_vector** (*tuple*) – up direction vector for the camera

5.7.13 perspective()

`p5.perspective()`

Sets a perspective projection for the camera in a 3D sketch.

Parameters

- **fovy** (*float*) – camera frustum vertical field of view, from bottom to top of view, in angleMode units
- **aspect** (*float*) – camera frustum aspect ratio
- **near** (*float*) – frustum near plane length
- **far** (*float*) – frustum far plane length

5.7.14 ortho()

`p5.ortho()`

Sets an orthographic projection for the camera in a 3D sketch and defines a box-shaped viewing frustum within which objects are seen.

Parameters

- **left** (*float*) – camera frustum left plane
- **right** (*float*) – camera frustum right plane
- **bottom** (*float*) – camera frustum bottom plane
- **top** (*float*) – camera frustum top plane

- **near** (*float*) – camera frustum near plane
- **far** (*float*) – camera frustum far plane

5.8 Color

5.8.1 Color

class p5.**Color**(**args*, *color_mode=None*, *normed=False*, ***kwargs*)

Represents a color.

alpha
The alpha value for the color.

b
The blue or the brightness value (depending on the color mode).

blue
The blue component of the color

brightness
The brightness component of the color

g
The green component of the color

gray
The gray-scale value of the color.

Performs a luminance conversion of the current color to grayscale.

green
The green component of the color

h
The hue component of the color

hex

Returns Color as a hex value

Return type str

hsb

Returns Color components in HSB.

Return type tuple

hsba

Returns Color components in HSBA.

Return type tuple

hue
The hue component of the color

lerp (*target*, *amount*)
Linearly interpolate one color to another by the given amount.

Parameters

- **target** (Color) – The target color to lerp to.
- **amount** (*float*) – The amount by which the color should be lerped (should be a float between 0 and 1).

Returns A new color lerped between the current color and the other color.

Return type *Color*

normalized
Normalized RGBA color values

normalized_rgb
Normalized RGB color values

r
The red component of the color

red
The red component of the color

rgb

Returns Color components in RGB.

Return type tuple

rgba

Returns Color components in RGBA.

Return type tuple

s
The saturation component of the color

saturation
The saturation component of the color

v
The brightness component of the color

value
The brightness component of the color

5.8.2 color_mode()

`p5.`**`color_mode`**(*mode*, *max_1=255*, *max_2=None*, *max_3=None*, *max_alpha=255*)

Set the color mode of the renderer.

Parameters

- **`mode`** (*`str`*) – One of {'RGB', 'HSB'} corresponding to Red/Green/Blue or Hue/Saturation/Brightness
- **`max_1`** (*`int`*) – Maximum value for the first color channel (default: 255)
- **`max_2`** (*`int`*) – Maximum value for the second color channel (default: max_1)
- **`max_3`** (*`int`*) – Maximum value for the third color channel (default: max_1)
- **`max_alpha`** (*`int`*) – Maximum valu for the alpha channel (default: 255)

5.8.3 background()

`p5.`**`background`**(**args*, ***kwargs*)

Set the background color for the p5.renderer.

Parameters

- **`args`** – positional arguments to be parsed as a color.
- **`kwargs`** (*`dict`*) – keyword arguments to be parsed as a color.

Note Both args and color_kwargs are directly sent to color.parse_color

Note When setting an image as the background, the dimensions of the image should be the same as that of the sketch window.

Returns The background color or image.

Return type p5.Color | p5.PImage

Raises **`ValueError`** – When the dimensions of the image and the sketch do not match.

5.8.4 fill()

`p5.`**`fill`**(**fill_args*, ***fill_kwargs*)

Set the fill color of the shapes.

Parameters

- **`fill_args`** (*`tuple`*) – positional arguments to be parsed as a color.

- **fill_kwargs** (*dict*) – keyword arguments to be parsed as a color.

Returns The fill color.

Return type *Color*

5.8.5 no_fill()

p5.**no_fill**()
Disable filling geometry.

5.8.6 stroke()

p5.**stroke**(**color_args*, ***color_kwargs*)
Set the color used to draw lines around shapes

Parameters

- **color_args** (*tuple*) – positional arguments to be parsed as a color.
- **color_kwargs** (*dict*) – keyword arguments to be parsed as a color.

Note Both color_args and color_kwargs are directly sent to Color.parse_color

Returns The stroke color.

Return type *Color*

5.8.7 no_stroke()

p5.**no_stroke**()
Disable drawing the stroke around shapes.

5.9 Image

5.9.1 PImage

class p5.**PImage**(*width*, *height*, *fmt='RGBA'*)
Image class for p5.

Note that the image "behaves" like a 2-D list and hence, doesn't expose special methods for copying / pasting / cropping. All of these operations can be done by using appropriate indexing into the array. See the *p5.PImage.__getitem__()* and *p5.PImage.__setitem__()* methods for details.

Parameters

- **`width`** (*`int`*) – width of the image.
- **`height`** – height of the image.
- **`fmt`** (*`str`*) – color format to use for the image. Should be one of {'RGB', 'RGBA', 'ALPHA'}. Defaults to 'RGBA'

`__getitem__`(*key*)
Return the color of the indexed pixel or the requested sub-region

Note :: when the specified *key* denotes a single pixel, the color of that pixel is returned. Else, a new PImage (constructed using the slice specified by *key*). Note that this causes the internal buffer data to be reloaded (when the image is in an "unclean" state) and hence, many such operations can potentially slow things down.

Returns a sub-image or a the pixel color

Return type p5.Color | p5.PImage

Raises **`KeyError`** – When *key* is invalid.

`__init__`(*width*, *height*, *fmt='RGBA'*)
Initialize self. See help(type(self)) for accurate signature.

`__setitem__`(*key*, *patch*)
Paste the given *patch* into the current image.

`__weakref__`
list of weak references to the object (if defined)

`aspect_ratio`
Return the aspect ratio of the image.

Return type float | int

`blend`(*other*, *mode*)
Blend the specified image using the given blend mode.

Parameters

- **`other`** (*`p5.PImage`*) – The image to be blended to the current image.
- **`mode`** (*`str`*) – Blending mode to use. Should be one of { 'BLEND', 'ADD', 'SUBTRACT', 'LIGHTEST', 'DARKEST', 'MULTIPLY', 'SCREEN',}

Raises

- **`AssertionError`** – When the dimensions of img do not match the dimensions of the current image.
- **`KeyError`** – When the blend mode is invalid.

filter (*kind*, *param=None*)
Filter the image.

Parameters

- **kind** (*str*) – The kind of filter to use on the image. Should be one of { 'threshold', 'gray', 'opaque', 'invert', 'posterize', 'blur', 'erode', 'dilate', }
- **param** (*int | float | None*) – optional parameter for the filter in use (defaults to None). Only required for 'threshold' (the threshold to use, param should be a value between 0 and 1; defaults to 0.5), 'posterize' (limiting value for each channel should be between 2 and 255), and 'blur' (gaussian blur radius, defaults to 1.0).

height
The height of the image

Return type int

load_pixels ()
Load internal pixel data for the image.

By default image data is only loaded lazily, i.e., right before displaying an image on the screen. Use this method to manually load the internal image data.

save (*file_name*)
Save the image into a file

Parameters **file_name** (*str*) – Filename to save the image as.

size
The size of the image

Return type (int, int) tuple

width
The width of the image

Return type int

5.9.2 Loading and displaying

image()

p5.**image** (*img*, *location*, *size=None*)
Draw an image to the display window.

Images must be in the same folder as the sketch (or the image path should be explicitly mentioned). The color of an image may be modified with the `p5.tint()` function.

Parameters

- **img** (*p5.Image*) – the image to be displayed.

- **location** (*tuple | list | np.ndarray | p5. Vector*) – location of the image on the screen (depending on the current image mode, 'corner', 'center', 'corners', this could represent the coordinate of the top-left corner, center, top-left corner respectively.)
- **size** (*tuple | list*) – target size of the image or the bottom-right image corner when the image mode is set to 'corners'. By default, the value is set according to the current image size.

image_mode()

p5.**image_mode**(*mode*)

Modify the locaton from which the images are drawn.

Modifies the location from which images are drawn by changing the way in which parameters given to *p5.image()* are intepreted.

The default mode is `image_mode('corner')`, which interprets the second parameter of `image()` as the upper-left corner of the image. If an additional parameter is specified, it is used to set the image's width and height.

`image_mode('corners')` interprets the first parameter of `image()` as the location of one corner, and the second parameter as the opposite corner.

`image_mode('center')` interprets the first parameter of `image()` as the image's center point. If an additional parameter is specified, it is used to set the width and height of the image.

Parameters mode (*str*) – should be one of `{'corner', 'center', 'corners'}`

Raises ValueError – When the given image mode is not understood. Check for typoes.

load_image()

p5.**load_image**(*filename*)

Load an image from the given filename.

Loads an image into a variable of type PImage. Four types of images may be loaded.

In most cases, load all images in setup() or outside the draw() call to preload them at the start of the program. Loading images inside draw() will reduce the speed of a program.

Parameters filename (*str*) – Filename (or path)of the given image. The file-extennsion is automatically inferred.

Returns An *p5.PImage* instance with the given image data

Return type *p5.PImage*

tint()

p5.**tint**(**color_args*, ***color_kwargs*)

Set the tint color for the sketch.

Parameters

- **color_args** (*tuple*) – positional arguments to be parsed as a color.
- **color_kwargs** (*dict*) – keyword arguments to be parsed as a color.

Note Both color_args and color_kwargs are directly sent to Color.parse_color

Returns The tint color.

Return type *Color*

no_tint()

p5.**no_tint**()

Disable tinting of images.

5.9.3 Pixels

load_pixels()

p5.**load_pixels**()

Load a snapshot of the display window into the `pixels` Image.

This context manager loads data into the global `pixels` Image. Once the program execution leaves the context manager, all changes to the image are written to the main display.

pixels

A *p5.PImage* containing the values for all the pixels in the display window. The size of the image is that of the main rendering window, width × height. This image is only available within the *p5.load_pixels()* context manager and set to `None` otherwise.

```
with load_pixels():
    # code manipulating the ``pixels`` object
```

Subsequent changes to this image object aren't reflected until *p5.load_pixels()* is called again. The contents of the display are updated as soon as program execution leaves the context manager.

5.10 Typography

5.10.1 Loading and displaying

create_font()

`p5.`**`create_font`**(*name*, *size=10*)

Create the given font at the appropriate size.

Parameters

- **`name`** (*`str`*) – Filename of the font file (only pil, otf and ttf fonts are supported.)
- **`size`** (*`int | None`*) – Font size (only required when *name* refers to a truetype font; defaults to None)

load_font()

`p5.`**`load_font`**(*font_name*)

Loads the given font into a font object

text()

`p5.`**`text`**(*text_string*, *position*, *wrap_at=None*)

Draw the given text on the screen and save the image.

Parameters

- **`text_string`** (*`str`*) – text to display
- **`position`** (*`tuple`*) – position of the text on the screen
- **`wrap_at`** (*`int`*) – specifies the text wrapping column (defaults to None)

Returns actual text that was drawn to the image (when wrapping is not set, this is just the unmodified text_string)

Return type str

text_font()

`p5.`**`text_font`**(*font*, *size=10*)

Set current text font.

Parameters **`font`** (*`PIL.ImageFont.ImageFont`*) –

5.10.2 Text Attributes

text_align()

p5.**text_align**(*align_x*, *align_y=None*)
Set the alignment of drawing text

Parameters

- **align_x** (*string*) – "RIGHT", "CENTER" or "LEFT".
- **align_y** (*string*) – "TOP", "CENTER" or "BOTTOM".

text_leading()

p5.**text_leading**(*leading*)
Sets the spacing between lines of text in units of pixels

Parameters **leading** – the size in pixels for spacing between lines

text_size()

p5.**text_size**(*size*)
Sets the current font size

Parameters **leading** – the size of the letters in units of pixels

5.10.3 Metrics

text_ascent()

p5.**text_ascent**()
Returns ascent of the current font at its current size

Returns ascent of the current font at its current size

Return type float

text_descent()

p5.**text_descent**()
Returns descent of the current font at its current size

Returns descent of the current font at its current size

Return type float

text_width()

p5.**text_width**(*text*)

Calculates and returns the width of any character or text string

Parameters **text_string** (*str*) – text

Returns width of any character or text string

Return type int

5.11 Math

Note: Many math functions available in Processing are already included in Python's math standard library. There methods aren't listed here. Please refer to the documentation for the math standard library.

5.11.1 Vector

class p5.**Vector**(*x*, *y*, *z=0*)

Describes a vector in two or three dimensional space.

A Vector – specifically an Euclidean (or geometric) vector – in two or three dimensional space is a geometric entity that has some magnitude (or length) and a direction.

Examples:

```
>>> vec_2d = Vector(3, 4)
>>> vec_2d
Vector(3.00, 4.00, 0.00)

>>> vec_3d = Vector(2, 3, 4)
>>> vec_3d
Vector(2.00, 3.00, 4.00)
```

Parameters

- **x** (*int or float*) – The x-component of the vector.
- **y** (*int or float*) – The y-component of the vector.
- **z** (*int or float*) – The z-component of the vector (0 by default; only required for 3D vectors;)

__abs__()

Return the magnitude of the vector.

__add__(*other*)
Add the location of one point to that of another.

Examples:

```
>>> p = Vector(2, 3, 6)
>>> q = Vector(3, 4, 5)
>>> p + q
Vector(5.00, 7.00, 11.00)
```

Parameters **other** –

Returns The point obtained by adding the corresponding components of the two vectors.

__eq__(*other*)
Return self==value.

__getitem__(*key*)
Return self[key].

__init__(*x, y, z=0*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()
Return the components of the vector as an iterator.

Examples:

```
>>> p = Vector(2, 3, 4)
>>> print([ c for c in p])
[2, 3, 4]
```

__mul__(*k*)
Multiply the point by a scalar.

Examples:

```
>>> p = Vector(2, 3, 6)
>>> p * 2
Vector(4.00, 6.00, 12.00)

>>> 2 * p
Vector(4.00, 6.00, 12.00)

>>> p * p
Traceback (most recent call last):
    ...
TypeError: Can't multiply/divide a point by a non-numeric.

>>> p = Vector(2, 3, 6)
>>> -p
```

(continues on next page)

(continued from previous page)

```
Vector(-2.00, -3.00, -6.00)

>>> p = Vector(2, 3, 6)
>>> p / 2
Vector(1.00, 1.50, 3.00)
```

Parameters **k** (*int, float*) –

Returns The vector obtained by multiplying each component of *self* by k.

Raises **TypeError** – When *k* is non-numeric.

__neg__()
Negate the vector.

__repr__()
Return a nicely formatted representation string

__rmul__(*other*)
Return value*self.

__str__()
Return a nicely formatted representation string

__sub__(*other*)
Subtract the location of one point from that of another.

Examples:

```
>>> p = Vector(2, 3, 6)
>>> q = Vector(3, 4, 5)
>>> p - q
Vector(-1.00, -1.00, 1.00)
```

Parameters **other** –

Returns The vector obtained by subtracteing the corresponding components of the vector from those of another.

__truediv__(*other*)
Divide the vector by a scalar.

angle
The angle of rotation of the vector (in radians).

This attribute isn't available for three dimensional vectors.

Examples:

```
>>> from math import pi, isclose
>>> p = Vector(1, 0, 0)
>>> isclose(p.angle, 0)
True

>>> p = Vector(0, 1, 0)
>>> isclose(p.angle, pi/2)
True

>>> p = Vector(1, 1, 1)
>>> p.angle
Traceback (most recent call last):
    ...
ValueError: Can't compute the angle for a 3D vector.

>>> p = Vector(1, 1)
>>> isclose(p.angle, pi/4)
True
>>> p.angle = pi/2
>>> isclose(p.angle, pi/2)
True

>>> p = Vector(1, 1)
>>> isclose(p.angle, pi/4)
True
>>> p.rotate(pi/4)
>>> isclose(p.angle, pi/2)
True
```

Raises **`ValueError`** – If the vector is three-dimensional

`angle_between`(*other*)
Calculate the angle between two vectors.

Examples:

```
>>> from math import degrees
>>> k = Vector(0, 1)
>>> j = Vector(1, 0)
>>> degrees(k.angle_between(j))
90.0
```

Parameters **`other`** (`Vector`) –

Returns The angle between *self* and *other* (in radians)

Return type float

`copy`()
Return a copy of the current point.

Returns A copy of the current point.

Return type *Vector*

cross (*other*)
Return the cross product of the two vectors.

Examples:

```
>>> i = Vector(1, 0, 0)
>>> j = Vector(0, 1, 0)
>>> i.cross(j)
Vector(0.00, 0.00, 1.00)
```

Parameters **other** (`Vector`) –

Returns The vector perpendicular to both *self* and *other* i.e., the vector obtained by taking the cross product of *self* and *other*.

Return type *Vector*

dist (*other*)
Return the distance between two points.

Returns The distance between the current point and the given point.

Return type float

distance (*other*)
Return the distance between two points.

Returns The distance between the current point and the given point.

Return type float

dot (*other*)
Compute the dot product of two vectors.

Examples:

```
>>> p = Vector(2, 3, 6)
>>> q = Vector(3, 4, 5)
>>> p.dot(q)
48
>>> p @ q
48
```

Parameters **other** (`Vector`) –

Returns The dot product of the two vectors.

Return type int or float

classmethod from_angle (*angle*)
Return a new unit vector with the given angle.

Parameters **`angle`** (*`float`*) – Angle to be used to create the vector (in radians).

`lerp`(*other*, *amount*)
Linearly interpolate from one point to another.

Parameters

- **`other`** – Point to be interpolate to.
- **`amount`** (*`float`*) – Amount by which to interpolate.

Returns Vector obtained by linearly interpolating this vector to the other vector by the given amount.

`limit`(*upper_limit=None*, *lower_limit=None*)
Limit the magnitude of the vector to the given range.

Parameters

- **`upper_limit`** (*`float`*) – The upper limit for the limiting range (defaults to None).
- **`lower_limit`** (*`float`*) – The lower limit for the limiting range (defaults to None).

`magnitude`
The magnitude of the vector.

Examples:

```
>>> p = Vector(2, 3, 6)
>>> p.magnitude
7.0

>>> abs(p)
7.0

>>> p.magnitude = 14
>>> p
Vector(4.00, 6.00, 12.00)

>>> p.normalize()
>>> print(p)
Vector(0.29, 0.43, 0.86)
```

`magnitude_sq`
The squared magnitude of the vector.

`normalize`()
Set the magnitude of the vector to one.

`classmethod random_2D`()
Return a random 2D unit vector.

classmethod random_3D()
Return a new random 3D unit vector.

rotate(*theta*)
Rotates the vector by an angle.

Parameters **theta** (`float or int`) – Angle (in radians).

x
The x-component of the point.

y
The y-component of the point.

z
The z-component of the point.

5.11.2 Calculation

ceil()

p5.**ceil**()
Return the ceiling of x as an Integral.

This is the smallest integer >= x.

constrain()

p5.**constrain**(*amount*, *low*, *high*)
Constrain the given value in the specified range.

Examples

```
>>> constrain(8, 1, 5)
5

>>> constrain(5, 1, 5)
5

>>> constrain(3, 1, 5)
3

>>> constrain(1, 1, 5)
1

>>> constrain(-3, 1, 5)
1
```

Parameters

- **amount** – The the value to be contrained.

- **low** – The lower constain.
- **high** – The upper constain.

dist()

p5.**dist**(*point_1*, *point_2*)

Return the distance between two points.

Examples

```
>>> distance((0, 0, 0), (2, 3, 6))
7.0

>>> distance((2, 3, 6), (2, 3, 6))
0.0

>>> distance((6, 6, 6), (2, 3, 6))
5.0
```

Parameters

- **point_1** (*tuple*) –
- **point_2** (*tuple*) –

Returns The distance between two points

Return type float

exp()

p5.**exp**()

Return e raised to the power of x.

floor()

p5.**floor**()

Return the floor of x as an Integral.

This is the largest integer <= x.

lerp()

p5.**lerp**(*start*, *stop*, *amount*)

Linearly interpolate the start value to the stop value.

Examples

```
>>> lerp(0, 10, 0.0)
0.0

>>> lerp(0, 10, 0.5)
5.0

>>> lerp(0, 10, 0.8)
8.0

>>> lerp(0, 10, 1.0)
10.0
```

Parameters

- **start** – The start value
- **stop** – The stop value
- **amount** (*float*) – The amount by which to interpolate. ($0 \leq amount \leq 1$).

log()

p5.**log**(*x*[, *base=math.e*])

Return the logarithm of x to the given base.

If the base not specified, returns the natural logarithm (base e) of x.

sq()

p5.**sq**(*number*)

Square a number.

Examples

```
>>> square(-25)
625

>>> square(0)
0

>>> square(13)
169
```

Parameters **number** (*float*) – The number to be squared.

Returns The square of the number.

Return type float

magnitude()

p5.**magnitude**(*x*, *y*, *z=0*)

Return the magnitude of the given vector.

Examples

```
>>> magnitude(3, 4)
5.0

>>> magnitude(2, 3, 6)
7.0

>>> magnitude(0, 0, 0)
0.0
```

Parameters

- **x** (*float*) – The x-component of the vector.
- **y** (*float*) – The y-component of the vector.
- **z** (*float*) – The z-component of the vector (defaults to 0).

Returns The magnitude of the vector.

Return type float

remap()

p5.**remap**(*value*, *source_range*, *target_range*)

Remap a value from the source range to the target range.

Examples

```
>>> remap(50, (0, 100), (0, 10))
5.0

>>> remap(5, (0, 10), (0, 100))
50.0

>>> remap(5, (0, 10), (10, 20))
15.0

>>> remap(15, (10, 20), (0, 10))
5.0
```

Parameters

- **value** – The value to be remapped.
- **source_range** (*tuple*) – The source range for `value`

- **target_range** (*tuple*) – The target range for `value`

normalize()

`p5.normalize(value, low, high)`

Normalize the given value to the specified range.

Examples

```
>>> normalize(10, 0, 100)
0.1

>>> normalize(0.3, 0, 1)
0.3

>>> normalize(100, 0, 100)
1.0

>>> normalize(1, 1, 15)
0.0
```

Parameters

- **value** (*float*) –
- **low** (*float*) – The lower bound for the range.
- **high** (*float*) – The upper bound for the range.

sqrt()

`p5.sqrt()`

Return the square root of x.

5.11.3 Trigonometry

acos()

`p5.acos()`

Return the arc cosine (measured in radians) of x.

asin()

`p5.asin()`

Return the arc sine (measured in radians) of x.

atan()

`p5.atan()`
Return the arc tangent (measured in radians) of x.

atan2()

`p5.atan2()`
Return the arc tangent (measured in radians) of y/x.

Unlike atan(y/x), the signs of both x and y are considered.

cos()

`p5.cos()`
Return the cosine of x (measured in radians).

degrees()

`p5.degrees()`
Convert angle x from radians to degrees.

radians()

`p5.radians()`
Convert angle x from degrees to radians.

sin()

`p5.sin()`
Return the sine of x (measured in radians).

tan()

`p5.tan()`
Return the tangent of x (measured in radians).

5.11.4 Random

noise()

`p5.noise(x, y=0, z=0)`
Return perlin noise value at the given location.

Parameters

- **x** (*float*) – x-coordinate in noise space.
- **y** (*float*) – y-coordinate in noise space.
- **z** (*float*) – z-coordinate in noise space.

Returns The perlin noise value.

Return type float

noise_detail()

`p5.`**`noise_detail`**`(`*octaves=4*, *falloff=0.5*`)`

Adjust the level of noise detail produced by noise().

Parameters

- **octaves** (*int*) – The number of octaves to compute the noise for (defaults to 4).
- **falloff** (*float*) –

Note For `falloff` values greater than 0.5, `noise()` will return values greater than 1.0.

noise_seed()

`p5.`**`noise_seed`**`(`*seed*`)`

Set the seed value for `noise()`

By default `noise()` produes different values each time the sketch is run. Setting the `seed` parameter to a constant will make `noise()` return the same values each time the sketch is run.

Parameters **seed** (*int*) – The required seed value.

random_uniform()

`p5.`**`random_uniform`**`(`*high=1*, *low=0*`)`

Return a uniformly sampled random number.

Parameters

- **high** (*float*) – The upper limit on the random value (defaults to 1).
- **low** (*float*) – The lowe limit on the random value (defaults to 0).

Returns A random number between `low` and `high`.

Return type float

random_gaussian()

`p5.`**`random_gaussian`**`(`*mean=0, std_dev=1*`)`

Return a normally sampled random number.

Parameters

- **`mean`** (*`float`*) – The mean value to be used for the normal distribution (defaults to 0).
- **`std_dev`** (*`float`*) – The standard deviation to be used for the normal distribution (defaults to 1).

Returns A random number selected from a normal distribution with the given `mean` and `std_dev`.

Return type float

random_seed()

`p5.`**`random_seed`**`(`*seed*`)`

Set the seed used to generate random numbers.

Parameters **`seed`** (*`int`*) – The required seed value.

5.12 SVG

5.12.1 load_shape()

`p5.`**`load_shape`**`(`*filename*`)`

Loads the given .svg file and converts it into PShape object.

Parameters **`filename`** (*`str`*) – link to .svg file

5.12.2 shape()

`p5.`**`shape`**`(`*shape, x=0, y=0*`)`

Draws shapes to the display window

Parameters

- **`shape`** (`PShape`) – shape object
- **`x`** (*`float`*) – x-coordinate of the shape
- **`y`** (*`float`*) – y-coordinate of the shape

5.13 Lights

5.13.1 Light Creation

p5.**lights**()

Sets the default ambient light, directional light, falloff, and specular values.

The defaults are ambientLight(128, 128, 128), directionalLight(128, 128, 128, 0, 0, -1), and lightFalloff(1, 0, 0).

p5.**ambient_light**(*r*, *g*, *b*)

Adds an ambient light.

Ambient light comes from all directions towards all directions. Ambient lights are almost always used in combination with other types of lights.

Parameters

- **r** (*float*) – red channel
- **g** (*float*) – green channel
- **b** (*float*) – blue channel

p5.**directional_light**(*r*, *g*, *b*, *x*, *y*, *z*)

Adds a directional light. Directional light comes from one direction: it is stronger when hitting a surface squarely, and weaker if it hits at a gentle angle. After hitting a surface, directional light scatters in all directions.

Parameters

- **r** (*float*) – red channel
- **g** (*float*) – green channel
- **b** (*float*) – blue channel
- **x** (*float*) – x component of the direction vector
- **y** (*float*) – y component of the direction vector
- **z** (*float*) – z component of the direction vector

p5.**point_light**(*r*, *g*, *b*, *x*, *y*, *z*)

Adds a point light. Point light comes from one location and emits to all directions.

Parameters

- **r** (*float*) – red channel
- **g** (*float*) – green channel
- **b** (*float*) – blue channel
- **x** (*float*) – x component of the location vector
- **y** (*float*) – y component of the location vector
- **z** (*float*) – z component of the location vector

5.13.2 Light Effects

p5.**light_falloff**(*constant*, *linear*, *quadratic*)

Sets the falloff rates for point lights. Affects only the elements which are created after it in the code.

d = distance from light position to vertex position

falloff = 1 / (constant + d * linear + (d*d) * quadratic)

If the coefficient is 0, then that term is ignored. The P3D renderer defaults to (0, 0, 0), i.e. no falloff.

Parameters

- **constant** (*float*) – coefficient for the constant term
- **linear** (*float*) – coefficient for the linear term
- **quadratic** (*float*) – coefficient for the quadratic term

light_specular()

p5.**light_specular**(*r*, *g*, *b*)

Sets the specular color for lights. Only visible with `p5.blinn_phong_material`. Is set to (0 ,0, 0) by default. Affects only the elements which are created after it in the code.

Specular refers to light which bounces off a surface in a preferred direction (rather than bouncing in all directions like a diffuse light) and is used for creating highlights. The specular quality of a light interacts with the specular material qualities set through the *specular()* and *shininess()* functions.

Parameters

- **r** (*float*) – red channel
- **g** (*float*) – green channel
- **b** (*float*) – blue channel

5.14 Materials

5.14.1 Basic Material

p5.**basic_material**(*r*, *g*, *b*)

The default material. Always displays a solid color.

Parameters

- **r** (*float*) – red channel
- **g** (*float*) – green channel

- **b** (*float*) – blue channel

5.14.2 Normal Material

p5.**normal_material**()

The color is determined by the normal vector of a surface. Does not respond to lights.

Useful for debugging.

5.14.3 Blinn-Phong Material

p5.**blinn_phong_material**()

Material based on the Blinn-Phong reflection model. This is the most “realistic” material in p5py.

Blinn-Phong shading can be decomposed into three parts: ambient, diffuse, and specular.

The ambient component is essentially a constant term that is alway present. We calculate it by summing all the ambient lights in a scene and multiplying it with the normalized ambient coefficent set by ambient.

The diffuse component takes the normal vector of a surface into account and varies how much light is reflected depending on the angle that the surface makes with the incoming light.

The specular component not only accounts for the direction of the light (like the diffuse component) but also the direction of the viewer. If the viewer is not on the path of the reflected light, the specular component falls off quickly, producing the glossy reflections we see on some materials.

The color shown on the user’s screen is the sum of all three components.

p5.**ambient**(*r*, *g*, *b*)

Sets the ambient light color reflected by the next *blinn_phong_material*.

Parameters

- **r** (*float*) – red channel
- **g** (*float*) – green channel
- **b** (*float*) – blue channel

p5.**emissive**(*r*, *g*, *b*)

This function is the same as diffuse.

Parameters

- **r** (*float*) – red channel
- **g** (*float*) – green channel
- **b** (*float*) – blue channel

p5.**diffuse**(*r*, *g*, *b*)

Sets the diffuse light color reflected by the next *blinn_phong_material*.

Parameters

- **r** (*float*) – red channel
- **g** (*float*) – green channel
- **b** (*float*) – blue channel

p5.**shininess**(*p*)

Sets how glossy the next *blinn_phong_material* is. This only affects the specular term.

Should be used together with *light_specular* and *specular*.

Parameters **p** (*float*) – exponent of the cosine term in the Blinn-Phong Reflection Model

p5.**specular**(*r*, *g*, *b*)

Sets the specular light color reflected by the next *blinn_phong_material*.

Should be used together with *light_specular*.

CHAPTER 6

Release Notes

6.1 0.6.0

p5 version 0.6.0 is the final release for the Google Summer of Code 2019 project by Arihant Parsoya. The project was supervised by Abhik Pal and Sam Lavigne of the Processing Foundation. The goals of the project were:

1. Add tutorials and examples for Processing modules
2. Complete typography module
3. Improving the Shape Class
4. Adding test suit to p5.py modules
5. Add 3D capabilities

We met the goals 1-4 completely. The last goal was completed partially as we were facing issues with vispy depth testing functionality. All the examples and tutorials for existing modules were imported from Processing website with live interactive sketch created using p5.js. Tests for all submodules are added using Pythons inbuilt *unittest* library. Additionally, bugs posted on github issue tracker are fixed.

6.1.1 API Additions

- Typography methods `p5.text_align()`, `p5.text_leading()` and `p5.text_size()` allows the user to control different position and size attributes of the text. Methods `p5.text_ascent()`, `p5.text_descent()` and `p5.text_width()` can be used to obtain different dimensional attributes of text.

- `p5.load_shape()` function allows the user to load an SVG file which can be rendered on the screen. The SVG shape is converted into PShape primitives and rendered on the screen using `p5.shape()` method.
- The methods `p5.begin_shape()` and `p5.end_shape()` can be used to make custom shapes defined by different vertex types (`p5.vertex()`, `p5.curve_vertex()`, `p5.bezier_vertex()`, `p5.quadratic_vertex()`). Contours inside a shape can also be created using `p5.begin_contour()` and `p5.end_contour()` methods.
- Stroke methods allows the user to control the stroke width and styles:
  - `p5.stroke_weight()`: allows for specifying the width of stroke
  - `p5.stroke_cap()`: allows the user to set the style of line endings
  - `p5.stroke_join()`: allows the user to set the style of joints which connect the line segments
- This release introduces limited 3D support. New functions for :
  - The `p5.camera()` allows for specifying the camera coordinates in 3D space
  - The projection functions `p5.ortho()` and `p5.perspective()` allow the user to control the type of projection system being used to render 3D objects on the screen
  - New 3D shape primitives are added: `p5.box()`, `p5.plane()`, `p5.sphere()`, `p5.ellipsoid()`, `p5.cylinder()`, `p5.cone()`, `p5.torus()`

6.2 0.5.0

p5 version 0.5.0 is the final release for the Google Summer of Code 2018 project by Abhik Pal. The project was supervised by Manindra Mohrarna of the Processing Foundation. The goal of the project were:

1. Move the internal windowing and OpenGL framework to vispy
2. Add support for user defined polygons
3. Add image support

We met all of these goals completely. The first was covered by a release from earlier in the summer. These release notes summarize our later two goals. In addition to the stated goals we were also able to add minimal typography support and port some tutorials from Processing to p5:

- *Color by Daniel Shiffman*
- *Vectors by Daniel Shiffman*
- *Electronics by Hernando Berragán and Casey Reas*

6.2.1 API Additions

- The `p5.PShape` class is equivalent to PShape in Processing. This allows creation of arbitrary user defined polygons that can have their own style (fill, stroke, etc) and transform (rotation, translation) attributes.
- The `p5.PImage` class allows for manipulating images in p5. Most of the API is similar to that of Processing's. Each image object "pretends" to be a 2D array and hence operations for cropping, copying, and pasting data can have implemented as indexing operations on the image. For instance, given some image `img` with dimensions 800 × 600, `img[400:, :300]` gives a new image with the required region. Individual pixels can be set / read as `p5.Color` objects though indices into the image. The class also includes functionality to apply filters and blend two images together.
- The `p5.load_image()` and `p5.image()` function allow, respectively, loading and displaying images on the screen.
- The `p5.image_mode()` function controls how parameters to `p5.image()` are interpreted.
- `p5.tint()` and the related `p5.no_tint()` function allow for setting and disabling tinting of images that are drawn on the screen.
- The `p5.load_pixels()` context manager loads the current display as a global `pixels` PImage object. This combines functionality of Processing's loadPixels() and updatePixels().
- `p5.save()` and `p5.save_frame()` methods allow users to either save the current state of the sketch or the final rendered frame as an image.
- This release also introduces some basic typography functions like `p5.text()` for displaying text on screen. The `p5.load_font()` and `p5.create_font()` allow loading font files to change the display typeface using `text_font()`. As of now, only TrueType (ttf) and bitmap fonts are supported.

Index