p5 Documentation

Release 0.5.0

Abhik Pal

Aug 14, 2018

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

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

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 Color

Authors Daniel Shiffman; Abhik Pal (p5 port)

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.

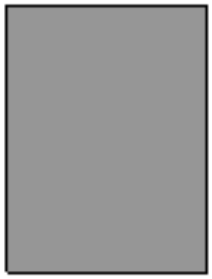


Fig. 1: Example: Using `background`, `stroke`, and `fill`

```
from p5 import *

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

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

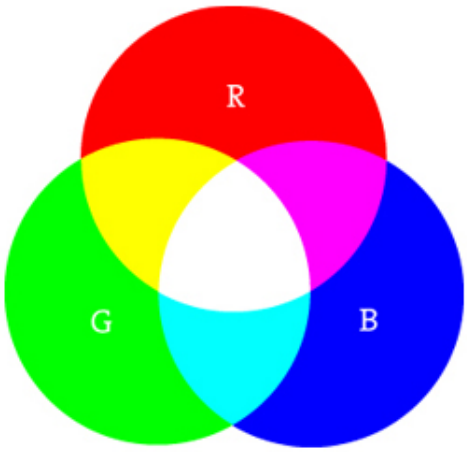
Stroke or fill can be eliminated with the functions: *p5.no_stroke()* and `no_fill()`. Our instinct might be to say `stroke(0)` for no outline, however, it is important to remember

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.

2.1.1 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.

Fig. 2: Example: RGB Color

```
from p5 import *

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

    # bright red
    fill(255, 0, 0)
    circle((72, 72), 58)

    # dark red
    fill(127, 0, 0)
    circle((144, 72), 58)

    # Pink (pale red)
    fill(255, 200, 200)
    circle((216, 72), 58)

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

2.1.2 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, 200)
    no_stroke()

def draw():
    background(0)

    # No fourth argument means 100% opacity.
```

(continues on next page)

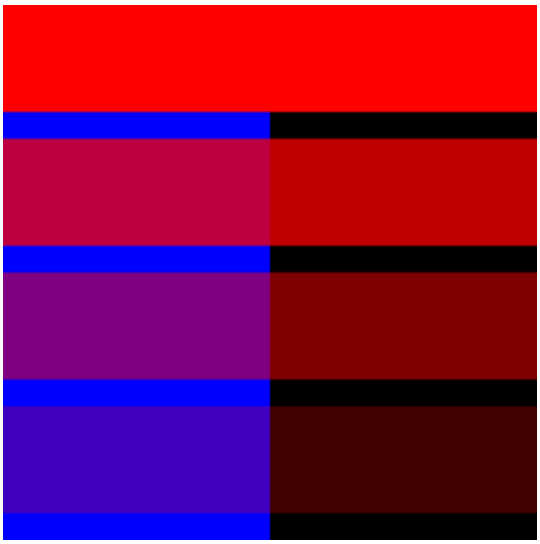


Fig. 3: Example: Alpha transparency

(continued from previous page)

```
    fill(0, 0, 255)
    rect((0, 0), 100, 200)

    # 255 means 100% opacity.
    fill(255, 0, 0, 255)
    rect((0, 0), 200, 40)

    # 75% opacity.
    fill(255, 0, 0, 191)
    rect((0, 50), 200, 40)

    # 55% opacity.
    fill(255, 0, 0, 127)
    rect((0, 100), 200, 40)

    # 25% opacity.
    fill(255, 0, 0, 63)
    rect((0, 150), 200, 40)

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

2.1.3 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', 100)
```

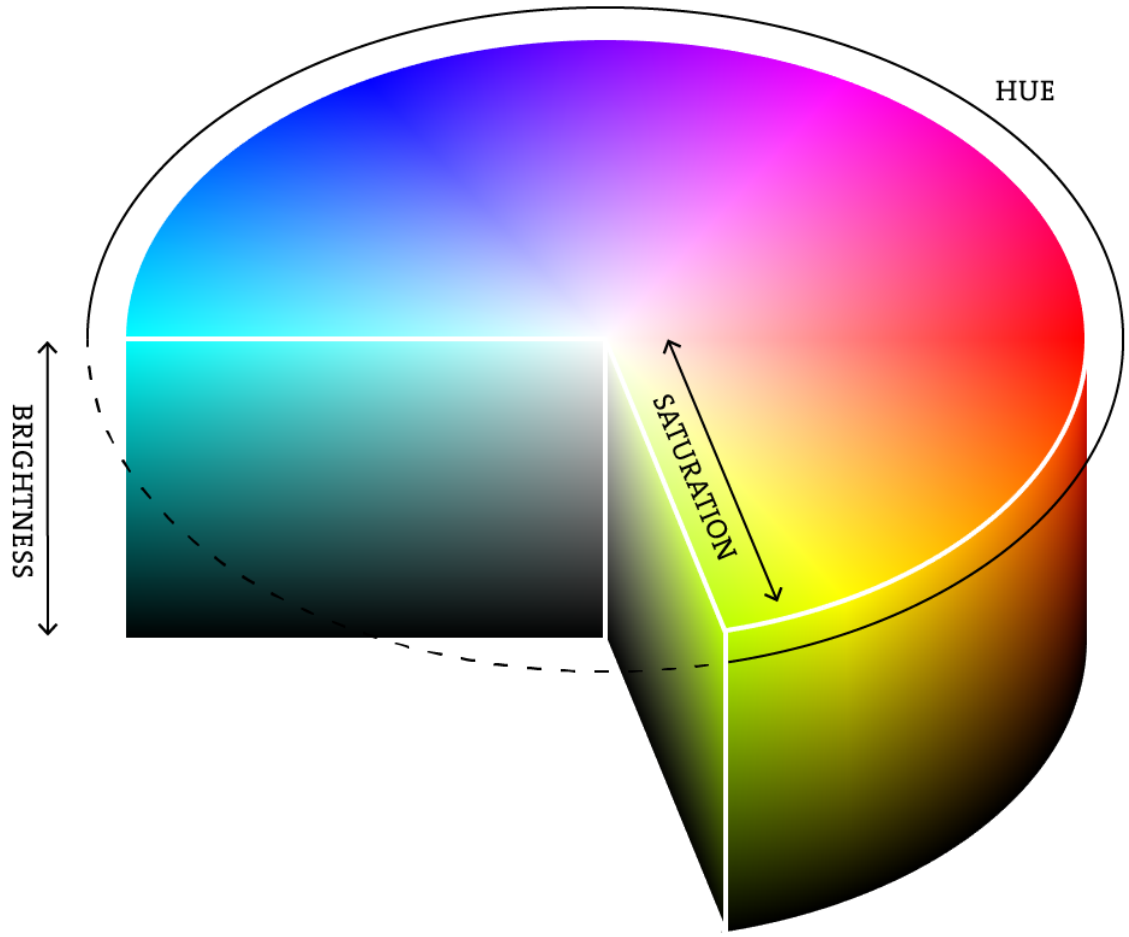
The above function says: "OK, we want to think about color in terms of red, green, and blue. The range of RGB values will be from 0 to 100."

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%).

2.2 Objects

2.3 Electronics

Authors Hernando Barragán; Casey Reas; Abhik Pal (p5 port)

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.

2.3.1 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.

2.3.2 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.

2.3.3 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.

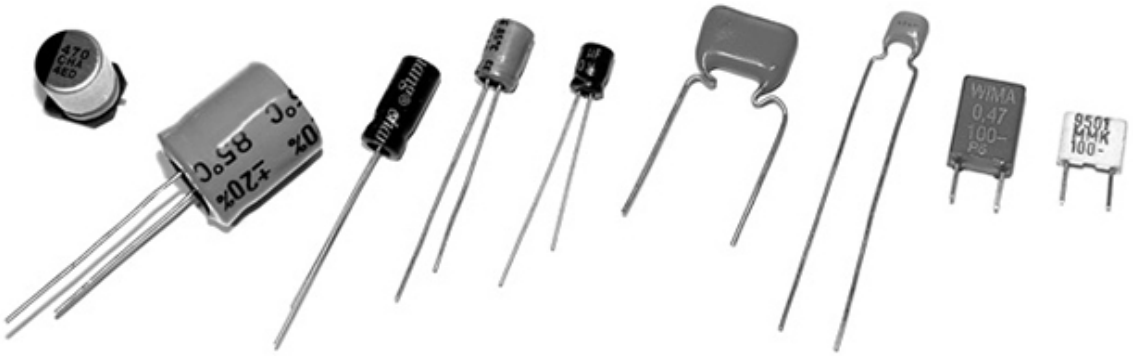
[1] Dan O'Sullivan and Tom Igoe, *Physical Computing: Sensing and Controlling the Physical World with Computers* (Thomson Course Technology PTR, 2004), p. 5

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).



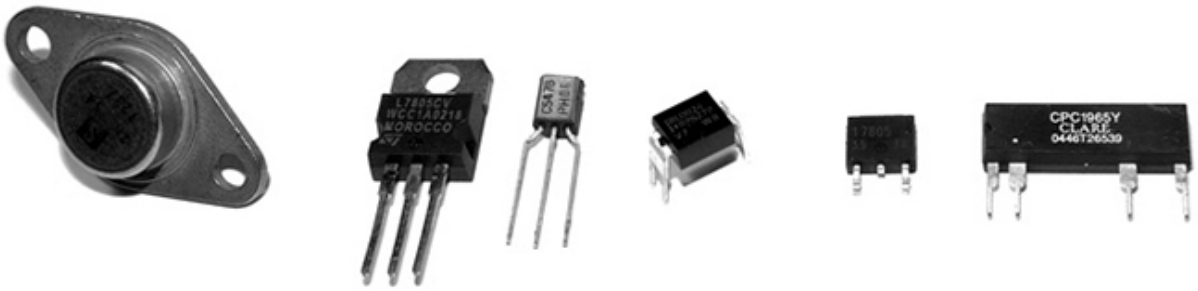
Diode

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

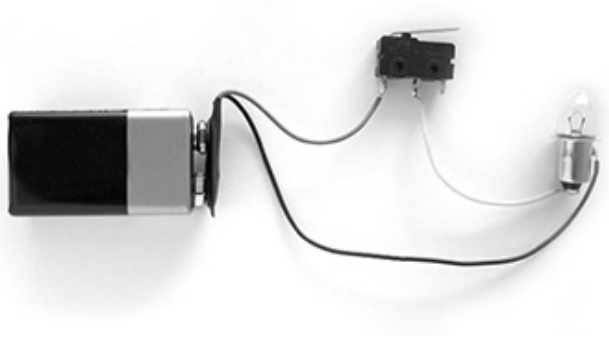
A transistor can be used as an electrical switch or an amplifier. A bipolar transistor has three leads (wires) called the base, collector, and emitter. Depending on the type of transistor, applying current to the base either allows current to flow or stops it from flowing through the device from the 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.



2.3.4 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.

2.3.5 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.

2.3.6 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.

2.3.7 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.

2.3.8 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.

2.3.9 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.

2.3.10 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:

2.3.11 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.

2.3.12 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.

2.3.13 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.

2.3.14 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.

2.3.15 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
}
```

(continues on next page)

(continued from previous page)

```
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)

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
```

(continues on next page)

(continued from previous page)

```
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)

    # 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)

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)
```

(continues on next page)

(continued from previous page)

```
        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
}

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
    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__':
```

(continues on next page)

(continued from previous page)

```
    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)
```

(continues on next page)

(continued from previous page)

```
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():
    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')
```

(continues on next page)

(continued from previous page)

```
        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)
```

2.4 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.

2.4.1 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
```

(continues on next page)

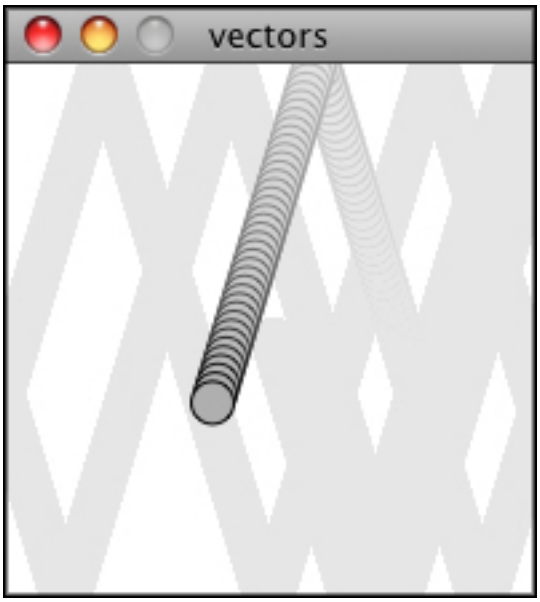


Fig. 4: Example : Bouncing ball with No Vectors.

(continued from previous page)

```
xspeed = 1
yspeed = 3.3

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

def draw():
    global x
    global y
    global xspeed
    global yspeed

    no_stroke()
    fill(255, 10)
    rect((0, 0), width, height)

    # add the current speed to the location
    x = x + xspeed
    y = y + yspeed

    if x > width or x < 0:
        xspeed = -xspeed

    if y > height or y < 0:
        yspeed = -yspeed

    stroke(0)
    fill(175)
```

(continues on next page)

(continued from previous page)

```
    circle((x, y), 16)

if __name__ == '__main__':
    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.

2.4.2 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:

$$location \rightarrow \texttt{x}, \texttt{y}$$
$$velocity \rightarrow \texttt{xspeed}, \texttt{yspeed}$$

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, $\star$ for multiplication, etc) These allow us to rewrite the above as:

```
# add the current speed to the location
location = location + velocity
```

2.4.3 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:

$$w_x = u_x + v_x$$
$$w_y = u_y + v_y$$

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
```

(continues on next page)

(continued from previous page)

```
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)

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
```

2.4.4 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.

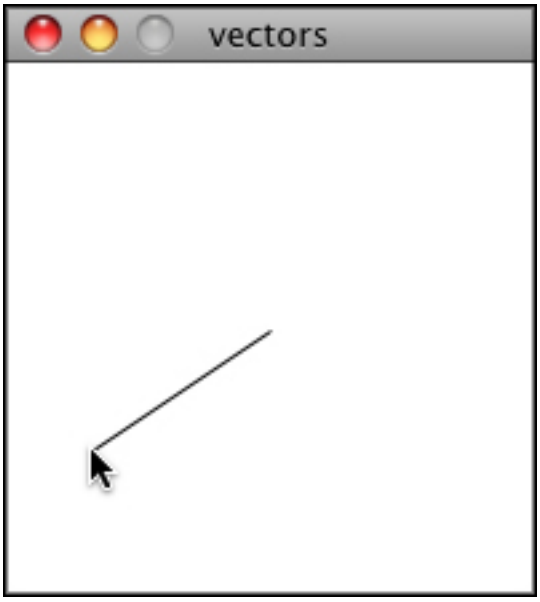


Fig. 5: Example: Vector subtraction

```
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
```

(continues on next page)

(continued from previous page)

```
        return self

def setup():
    size(200, 200)

def draw():
    background(255)

    # Two vectors, one for the moust location and one ofr the center
    # of the window
    mouse = Vector(mouse_x, mouse_y)
    center = Vector(width / 2, height / 2)

    # Vector subtraction!
    mouse = mouse - center

    # Draw a line to represent the vector
    translate(center.x, center.y)
    line((0, 0), (mouse.x, mouse.y))

if __name__ == '__main__':
    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

    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
```

(continues on next page)

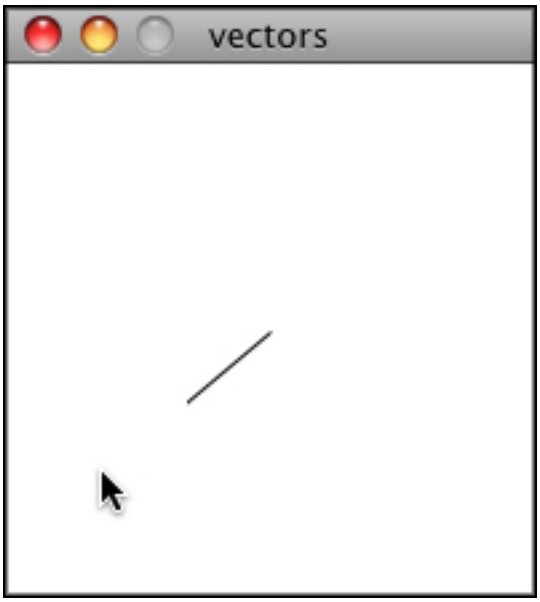


Fig. 6: Example: Vector multiplication

(continued from previous page)

```
        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, n):
    self.x = self.x / n
    self.y = self.y / n
    return self

# ...

u = Vector(8, -4)
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: $(\vec{u} + \vec{v}) \cdot n$

2.4.5 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)
```

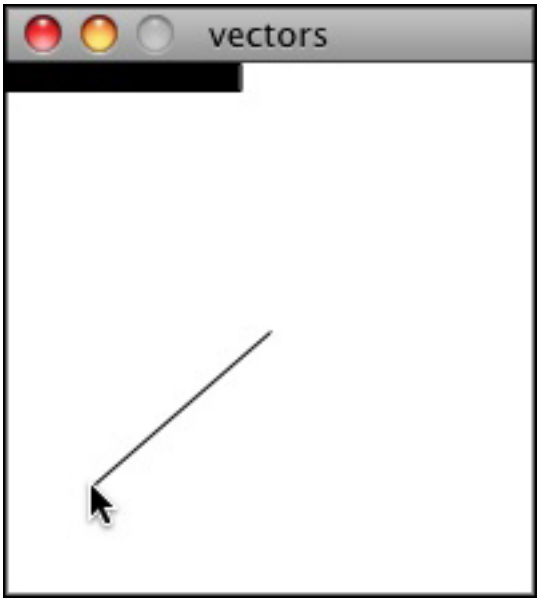


Fig. 7: Example: Vector magnitude

```
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
```

(continues on next page)

(continued from previous page)

```
    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)

    translate(center.x, center.y)
    line((0, 0), (mouse.x, mouse.y))

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

2.4.6 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
```

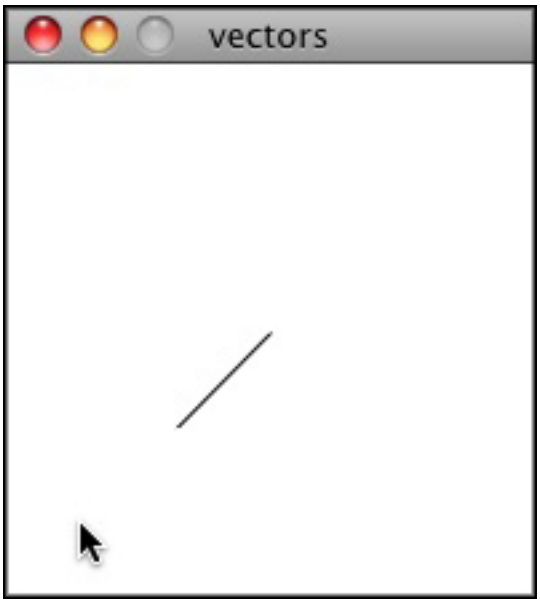


Fig. 8: Example: Normalizing a Vector

```
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
```

(continues on next page)

(continued from previous page)

```
    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)
    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()
```

2.4.7 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):
        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:

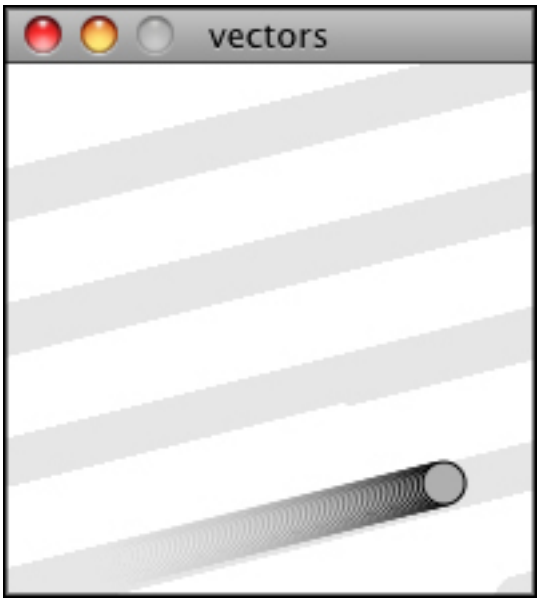


Fig. 9: Example: Motion 101 (velocity)

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()

def draw():
    no_stroke()
    fill(255, 10)
    rect((0, 0), width, height)

    # call functions on Mover object
    mover.update()
```

(continues on next page)

(continued from previous page)

```
    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):
        self.location = Vector(width / 2, height / 2)
        self.velocity = Vector(0, 0)

        # Acceleration is the key!
        self.acceleration = Vector(-0.001, 0.01)

        # this will limit the magnitude of velocity
```

(continues on next page)

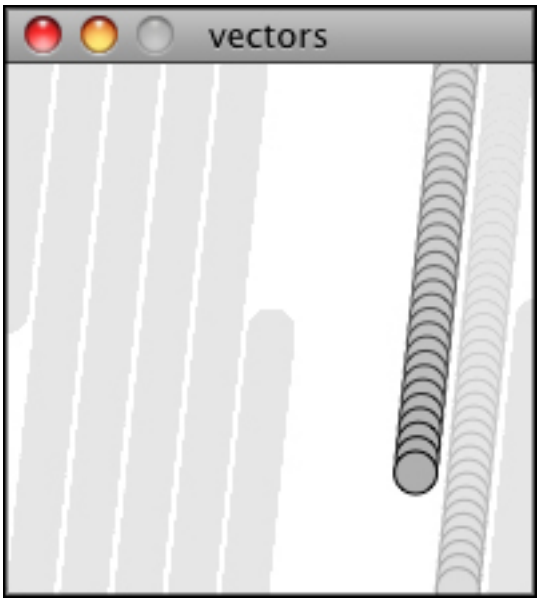


Fig. 10: Example: Motion 101 (velocity and constant acceleration)

(continued from previous page)

```
        self.top_speed = 10

    def update(self):
        self.velocity = self.velocity + self.acceleration
        self.velocity.limit(self.top_speed)
        self.location = self.location + self.velocity

    # rest of the methods are the same
```

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_uniform(low=(-1), high=1)
    acc_y = random_uniform(low=(-1), high=1)
    self.acceleration = Vector(acc_x, acc_y)
    self.acceleration.normalize()

    self.velocity = self.velocity + self.acceleration
    self.velocity.limit(self.top_speed)
    self.location = self.location + self.velocity
```

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 = random_uniform(low=(-1), high=1)
acc_y = random_uniform(low=(-1), high=1)
```

(continues on next page)

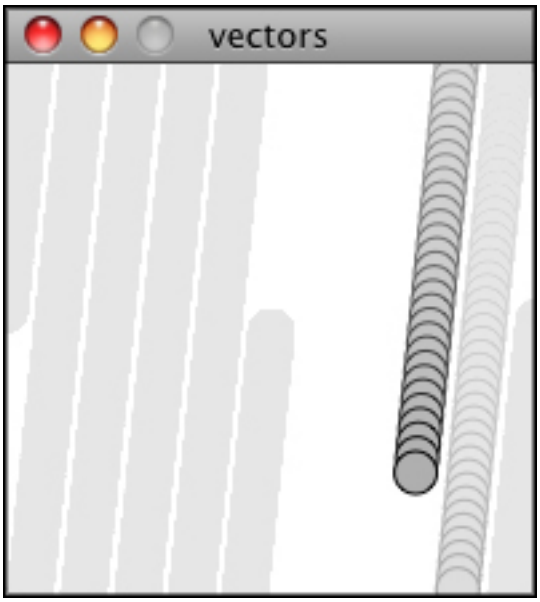


Fig. 11: Example: Motion 101 (velocity and random acceleration)

(continued from previous page)

```
self.acceleration = Vector(acc_x, acc_y)
self.acceleration.normalize()

self.acceleration = self.acceleration * 0.5
```

2. scaling the acceleration to a random value

```
acc_x = random_uniform(low=(-1), high=1)
acc_y = random_uniform(low=(-1), high=1)
self.acceleration = Vector(acc_x, acc_y)
self.acceleration.normalize()

self.acceleration = self.acceleration * random_uniform(2)
```

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.

2.4.8 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 = ???
direction.normalize()
direction = direction * anything
```

To summarize, we have the following steps:

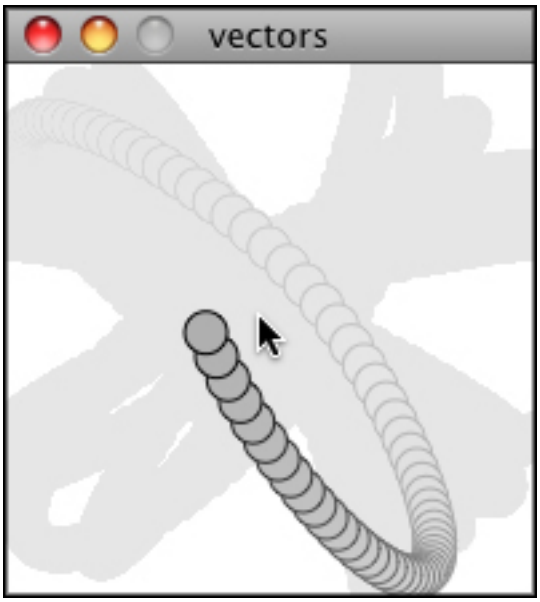
1. Calculate a vector that points from the object to the target location (mouse).
2. Normalize that vector (reducing its length to 1)
3. Scale that vector to an appropriate value (by multiplying it by some value)
4. Assign that vector to acceleration

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
```

(continues on next page)



(continued from previous page)

```
    # 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 = []
```

(continues on next page)

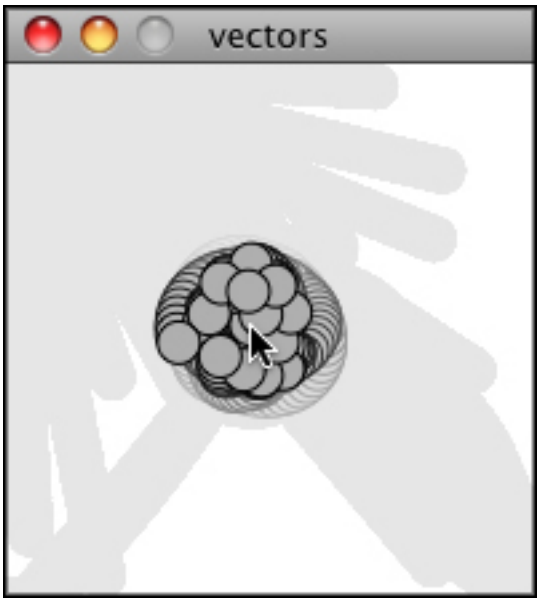


Fig. 12: Example: Accelerating towards mouse

(continued from previous page)

```
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)
        self.location = self.location + self.velocity
```

(continues on next page)

(continued from previous page)

```
    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:** 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

Guides

3.1 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.

3.1.1 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.

3.1.2 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.

3.1.3 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, 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 Proceccsing 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.

3.1.4 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
```

3.1.5 Math

- Vector addition, subtraction, and equality testing are done using the usual mathematical operators and scalar multiplication is done using the usual $\star$ 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 = map(mouse_x, source, target)
```

3.1.6 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
```

CHAPTER 4

Reference

4.1 Structure

4.1.1 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.

4.1.2 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.

4.1.3 run()

p5.**run**(*sketch_setup=None*, *sketch_draw=None*, *frame_rate=60*)
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 ≥ 1) – The target frame rate for the sketch.

4.1.4 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.

4.1.5 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.

4.1.6 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.

4.1.7 redraw()

p5.**redraw**()

Call *draw()* once.

If looping has been disabled using *no_loop()*, *redraw()* will make sure that *draw()* is called *exactly* once.

4.1.8 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

4.2 Environment

4.2.1 size()

p5.**size**(*width*, *height*)
Resize the sketch window.

Parameters

- **width** (*int*) – width of the sketch window.
- **height** (*int*) – height of the sketch window.

4.2.2 title()

p5.**title**(*new_title*)
Set the title of the p5 window.

Parameters **new_title** (*str*) – new title of the window.

4.2.3 height, width

Global integers that store the current width and height of the sketch window.

4.2.4 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.

4.2.5 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.

4.3 Shape

4.3.1 PShape

class p5.**PShape**(*vertices=[], fill_color='auto', stroke_color='auto', visible=False, attribs='closed', children=[]*)
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)
- **attribs** (*str*) – space-separated list of attributes that control shape drawing. Each attribute should be one of {'point', 'path', 'open', 'closed'}. (default: 'closed')
- **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.

Raises **ValueError** – when the vertex is of the wrong dimension

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 indicidual 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.

Raises **ValueError** – when the vertex is of the wrong dimension

4.3.2 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 sketch.)

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 sketch.)

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='OPEN PIE'*, *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 some combination of {'OPEN', 'CHORD', 'PIE'} separated by spaces. For instance, 'OPEN PIE', etc (defaults to 'OPEN 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 sketch.)

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 sketch.)

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 sketch.)

Returns A rectangle.

Return type *p5.PShape*

Raises **ValueError** – When the mode is set to 'CORNERS'

4.3.3 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.

4.3.4 Attributes

ellipse_mode()

p5.**ellipse_mode**(*mode='CENTER'*)
Change the ellipse and circle drawing mode for the sketch.

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 sketch.

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.

4.4 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.

4.4.1 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.

4.4.2 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.

4.5 Output

4.5.1 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`)

4.6 Transform

4.6.1 push_matrix()

`p5.push_matrix()`

4.6.2 print_matrix()

`p5.print_matrix()`

Print the transform matrix being used by the sketch.

4.6.3 reset_matrix()

`p5.reset_matrix()`

Reset the current transform matrix.

4.6.4 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

4.6.5 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

4.6.6 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

4.6.7 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

4.6.8 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

4.6.9 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

4.6.10 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

4.6.11 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

4.7 Color

4.7.1 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

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

4.7.2 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)

4.7.3 background()

p5.**background**(**args*, ***kwargs*)
Set the background color for the 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.

4.7.4 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*

4.7.5 no_fill()

p5.**no_fill**()

Disable filling geometry.

4.7.6 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*

4.7.7 no_stroke()

p5.**no_stroke**()
Disable drawing the stroke around shapes.

4.8 Image

4.8.1 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

4.8.2 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.

4.8.3 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.

4.9 Typography

4.9.1 Loading and displaying

create_font()

p5.**create_font**(*name*, *size=None*)

Create the given font at the appropriate size.

Parameters

- **name** (*str*) – Filename of the font file (only pil 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*)

Set current text font.

Parameters **font** (*PIL.ImageFont.ImageFont*) –

4.10 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.

4.10.1 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
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
    ...
TypeError: Can't multiply/divide a point by a non-numeric.

>>> p = Vector(2, 3, 6)
>>> -p
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 self*value.

__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.

4.10.2 Calculation

ceil()

p5.**ceil**(*x*)
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**(*x*)

Return e raised to the power of x.

floor()

p5.**floor**(*x*)

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*])

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**(*x*)

Return the square root of x.

4.10.3 Trigonometry

acos()

p5.**acos**(*x*)

Return the arc cosine (measured in radians) of x.

asin()

p5.**asin**(*x*)

Return the arc sine (measured in radians) of x.

atan()

p5.**atan**(*x*)
Return the arc tangent (measured in radians) of x.

atan2()

p5.**atan2**(*y*, *x*)
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**(*x*)
Return the cosine of x (measured in radians).

degrees()

p5.**degrees**(*x*)
Convert angle x from radians to degrees.

radians()

p5.**radians**(*x*)
Convert angle x from degrees to radians.

sin()

p5.**sin**(*x*)
Return the sine of x (measured in radians).

tan()

p5.**tan**(*x*)
Return the tangent of x (measured in radians).

4.10.4 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.

CHAPTER 5

Release Notes

5.1 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*

5.1.1 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.

Python Module Index

p

Index

Symbols

A

B

C

D

E

S

T

U

V

W

X

Y

Z