16. Processing

Once you understand how to build and use classes and object in Python, it's fun to use them in a context where they really make sense. Object-oriented programming is particularly useful in graphical programming environments. We have seen this already with the Turtle class. The Processing programming environment provides a dramatically more capable graphical programming environment than the Turtle class.

Processing is a free, open-source, programming environment invented by artists and data scientists that provides a simple way of making sophisticated visualizations. Processing is not part of the standard Python distribution, so if you do not already have it installed, download and install a suitable version from http://processing.org/download/. These notes are based on Processing 2.2.1, the most recent version at the time of writing.

Recently, the Processing developers have added the ability to program using Python syntax. Once you have installed Processing, run it. Before proceeding, find the *Add Mode...* dialog, download and install the *Python* mode written by Jonathan Feinberg. Quit and restart Processing and select the *Python* mode. Once that's done, you'll be able to use Python syntax in the editing window.

Here's a simple Python Processing example that animates a little barbell. Copy and paste the example into your Python Processing window and click the *play* button at the top.

```
# Floating barbell.
# initialize parameters
hg = 0 # height of green end
hr = 0 # height of red end
t=0 # time parameter
dt = 0.01 # time step

# this happens once at the start of execution
def setup():
    size(500,500) # set the size of the canvas

# this happens repeatedly, like frames in a movie
def draw():
    global t, hg, hr, dt
    background(255) # set the background to white
    # vary the heights using sine function
    hr = height/3*sin(2*PI*t)
    hg = height/3*sin(PI*t)
    fill(255,0,0) # set the left end to red
    ellipse(width/2-width/3,height/2+hr,10,10) # draw left end
    fill(0,255,0) # set the right end to green
    ellipse(width/2+width/3,height/2+hg,10,10) # draw right end
    # draw line connecting the ends
    line(width/2-width/3,height/2+hr,width/2+width/3,height/2+hg)
    t = t+dt # increment time
```

This example illustrates the basic Processing paradigm of setting up an animation with the `setup()` function and then drawing each frame of the animation using the `draw()` function. Both `setup()` and `draw()` are reserved by Processing for this purpose. There are a number of other reserved functions that allow the programmer to access the mouse and the keyboard. There are some small differences between Python Processing and ordinary Python. Notice, for example, that we used the `sin()` function without importing `math`. Notice also that we used the constant `PI` instead of `math.pi` as we would have in ordinary Python. When using Python Processing be prepared to deal with these small inconsistencies.

That said, we'll be keeping it simple with Processing and you may not run into any other anomalies. Some official documentation is beginning to appear on the official Processing website. Also, you may load working examples from the `File->Examples...` dialog in Processing when in Python mode.

Processing comes with a substantial set of tutorials, examples, and help, so there is ample opportunity to stretch yourself in this environment. You may need to look around a bit to find these resources.

16.1. The Structure of Processing Sketch

A program in Processing is called a *sketch*. This terminology is an acknowledgment of Processing's roots in the areas of art and design. It is reasonable to think of a sketch as an animation that you might have created as a child using a flipbook. Recall that in a *flipbook* animation you created each *frame* of the animation on its own page of the flipbook– successive pages showing only incremental changes. When you finished, you flipped through the pages in rapid succession to create the illusion of motion. In Processing you erase and re-paint the canvas several times per second to achieve the same effect.

In the barbell example above, the `draw()` function is called many times per second. Notice the first instruction is `background(255)`. This paints the canvas white, erasing whatever was there before, and gives you a clean canvas to draw on for the next frame. It's just like the loops we learned about in *The while statement* section.

In the *Scope and lookup rules* section we learned about where variables are recognized within a Python program. These same scoping rules are obeyed in Python Processing. The variables at the top of the barbell example are *global* variables. If necessary, we may define *local* variables in the `setup()` or `draw()` (or other functions). It so happens we did not need any local variables in this example.

16.2. Learning Processing

It would be inefficient to try and teach you everything you need to know about Processing here when there is already so much material available. The main purpose of this section is to highlight the syntactical differences between the default Processing syntax (which is based on Java) and the Python Processing syntax so that you can easily adapt existing tutorial information about Processing to your Python Processing environment.

Consider the traditional (or non-Python) Processing program here:

```
// Floating barbell.
// initialize parameters
float hg = 0; // height of green end
float hr = 0; // height of red end
float t=0; // time parameter
float dt = 0.01; // time step

// this happens once at the start of execution
void setup() {
    size(500,500); // set the size of the canvas
}

// this happens repeatedly, like frames in a movie
void draw() {
    background(255); // set the background to white
    // vary the heights using sine function
    hr = height/3*sin(2*PI*t);
    hg = height/3*sin(PI*t);
    fill(255,0,0); // set the left end to red
    ellipse(width/2-width/3,height/2+hr,10,10); // draw left end
    fill(0,255,0); // set the right end to green
    ellipse(width/2+width/3,height/2+hg,10,10); // draw right end
    // draw line connecting the ends
    line(width/2-width/3,height/2+hr,width/2+width/3,height/2+hg);
```

```
    t = t+dt; // increment time
}
```

Functionally, this program does the same thing as the first barbell program listed above using Python. It is useful to compare this program line-by-line to the first listing in this section. The syntactical differences are that the comments start with `//` instead of `#`, each line ends with `;`, the function blocks are indicated by braces `{` and `}`, the functions start with `void` instead of `def`, and the variables are declared with a type `float` at the top of the program.

We now point out the major differences between the default Processing syntax that you will find in all of the tutorial information on-line and the Python Processing syntax that we will be using.

16.2.1. Comments

- (single-line) In Processing starts with `//`, in Python Processing starts with `#`.
- (multi-line) In Processing surrounded by `/*` and `*/`, in Python Processing surrounded by matching `"""`'s.

16.2.2. Statements

Processing statements are terminated using semi-colons (`;`). Python Processing statements are on separate lines.

16.2.3. Blocks

In Processing blocks of statements, whether in a loop, a function or an if statement are grouped using pairs of matching braces, `{` and `}`. In Python Processing blocks are statements are grouped using indentation.

16.2.4. Datatypes and global variables

In Processing, variable types must be declared before the variables are used. The standard datatypes are `int`, `float`, `string`, and `boolean`. In Python Processing, variable types do not need to be declared (in fact cannot be “declared”).

A peculiarity of this difference is that in Python Processing global variables need to be initialized with some kind of placeholder value at the top of the program outside of `setup()` or `draw()` or any other function. In addition, when you want to access a global variable within one of the functions you must use the `global` keyword at the top of the function definition and list all of the global variables that you wish to use inside the function. When converting an existing program from the default Processing syntax to Python Processing, just replace the declaration statements with initializations and introduce the `global` statement in each function where it is relevant.

16.2.5. Function return types

Related to the datatype issue in the previous paragraph, in default Processing, the datatype (e.g. `int`, `float`) of the return value of a function must be given in the definition of the function. Even for a function that has no return value, a return type must be given (`void`). In Python Processing the return type is replaced by the `def` keyword in all cases.

16.3. Learning Processing

With the major differences summarized, here is a recommended list of Processing tutorials (in prerequisite order) to help you learn to use Python Processing. Remember as you use the examples in these tutorials, you must convert the syntax to Python Processing before running them.

- Getting Started. This introduction covers the basics of writing Processing code. (by Casey Reas and Ben Fry)
- Coordinate System and Shapes. Drawing simple shapes and using the coordinate system. (by Daniel Shiffman)
- Color. An introduction to digital color. (by Daniel Shiffman)
- Objects. The basics of object-oriented programming. (by Daniel Shiffman)

More advanced tutorials are also available on the main Processing site. Although if you do develop an interest in advanced Processing, it would probably be better to learn the default syntax.

16.4. Objects and Newtonian Motion

We finish this chapter with an extended example that re-visits object-oriented programming. We will create a Ball class that allows the creation of Ball objects on the Processing canvas each of which follows the laws of Newtonian motion.

16.4.1. Position and Velocity

Suppose we have a point in a plane at position (x, y) with velocity in the horizontal direction, v_x, and in the vertical direction, v_y. Recall that velocity is the distance that the point moves in a set unit of time. This is an ideal situation for the *flipbook* worlds that we create in Processing. Each flip of a frame is a set unit of time. If we know v_x, we can simulate the motion of a point by simply adding v_x to x with each flip of a frame. This will provide us with a nice approximation of the horizontal motion of the point.

Similarly, knowing v_y and adding it to y with each frame flip, gives a nice simulation of motion in the vertical direction. Combining the two, gives a good approximation of motion in a plane—2D motion.

So, if we know the initial position of a point in the plane, (x, y) and the velocity of that point, we can generate the sequence of frame-by-frame positions of the point by executing the following statements for each frame:

```
x = x + vx    # increment x by vx
y = y + vy    # increment y by vy
```

For example, a point with initial position on the canvas, $(0, 0)$, and velocity, $(1, 1)$, will move sequentially through the points $(0, 0)$, $(1, 1)$, $(2, 2)$, $(3, 3)$, … In other words, it will appear to move diagonally from the upper left corner of the canvas to the lower right corner of the canvas.

A velocity of $(1, 0)$ will give a point that moves horizontally from left to right across the screen. $(0, 1)$ will give a point that moves from top to bottom.

16.4.2. Acceleration

In the real world, it's rare to find a point that has a constant velocity like we describe in the previous section. Typically, there are other forces involved that cause the velocity to change. To build a truly realistic motion simulation, we should try to introduce a quantity to model the change in velocity. We will call this quantity *acceleration*. Like velocity, acceleration will have two components—vertical and horizontal. Let a_x be the horizontal component of acceleration and a_y be the vertical component of acceleration. Acceleration is the amount that the velocity changes in a set unit of time. Again, ideal for our flipbook Processing universe.

Adding a_x to v_x with each flip of the frame will give a great approximation of horizontal acceleration. Similarly, adding a_y to v_y will approximate vertical acceleration. You can likely guess the set of statements we should execute with each flip of the frame to approximate Newtonian motion with

acceleration:

```
vx = vx + ax  # increment vx by ax
vy = vy + ay  # increment vy by ay
x = x + vx    # increment x by vx
y = y + vy    # increment y by vy
```

For example, on a canvas with gravity acting downward we might consider a point with initial position $(100, 0)$, initial velocity $(0, 0)$ (at rest), and acceleration $(0, 1)$. In this case, the frame-by-frame sequence of positions given by the above 4 statements is: $(100, 0)$, $(100, 0)$, $(100, 1)$, $(100, 3)$, $(100, 6)$, $(100, 10)$, $(100, 15)$, … Observe, that the horizontal position remains fixed, but that the vertical position grows at an increasing rate. This is precisely how gravity functions on earth.

It is important to note that we may have acceleration in the horizontal direction, but that no longer simulates a gravitational force acting downward. We should also note that we could have negative accelerations. For example, an acceleration of the form $(0, -1)$ in the above example would have our point moving to the top of the canvas at an ever increasing rate.

Finally, we note that this is not the limit on how complex such simulations can get. We can imagine introducing yet another quantity that changes the acceleration with each flip of the frame and so on. We'll leave that extension to the physicists however.

16.4.3. Wall Bounces

We describe an algorithm that simulates a ball of radius, R, bouncing off of a wall in a 2D canvas of dimensions $w \times h$ and elasticity, E. Immediately following the velocity and position adjustment algorithm described in the previous section, check for wall collisions with this algorithm:

```
# top wall collision
if(y<R):
  y = R
  vy = -vy*E

# bottom wall collision
if(y>h-R):
  y = h-R
  vy = -vy*E

# left wall collision
if(x<R):
  x = R
  vx = -vx*E

# right wall collision
if(x>w-R):
  x = w-R
  vx = -vx*E
```

The position adjustments conserve the position change by moving the circle away from the wall the distance it "penetrated" the wall before triggering the collision detection. The velocity is "damped" by the elasticity coefficient with is a value between 0 and 1. If E is 1, then the collision is perfectly elastic, no energy is lost and the circle will bounce around forever. If E is 0, then the collision is perfectly inelastic and the circle will lose all of its velocity in the direction of the colliding wall. More intuitively, if E is say 0.9, then the circle loses 10% of its velocity in the direction of the colliding wall after the collision. In this case, due to round off errors, the circle will eventually come to rest after a certain number of bounces.

16.4.4. A Ball Class

To close out this example, let's build a simple sketch that creates a ball of random initial position and random initial velocity whose subsequent motion is governed by Newton's Laws. We will use a Ball class to represent the bouncing ball.

```
class Ball:
    """ A Ball class to represent bouncing Newtonian balls. """
    def __init__(self, R):
        """ Create Ball object of radius R. """
        self.R = R
        # Assign random position, upper part of canvas
        self.x = random(0,200)
        self.y = random(0,100)
        # Assign random velocity
        self.vx = random(-2,2)
        self.vy = random(-2,2)
        # Fixed acceleration, downward
        self.ax = 0
        self.ay = 0.3
        # Fixed elasticity
        self.E = 0.9

    def render(self):
        """ Draw the Ball at its current position. """
        noStroke()
        fill(255,0,0)
        ellipseMode(RADIUS)
        ellipse(self.x,self.y,self.R,self.R)

    def move(self):
        """ Increment the velocity and position by one step.
            Implement wall bounces"""
        self.vx = self.vx + self.ax
        self.vy = self.vy + self.ay
        self.x = self.x + self.vx
        self.y = self.y + self.vy
        # Collision detection:
        # top wall collision
        if(self.y<=self.R):
            self.y = 2*self.R - self.y
            self.vy = -self.vy*self.E

        # bottom wall collision
        if(self.y>height-self.R):
            self.y =height-self.R
            self.vy = -self.vy*self.E

        # left wall collision
        if(self.x<self.R):
            self.x = self.R
            self.vx = -self.vx*self.E

        # right wall collision
        if(self.x>width-self.R):
            self.x = width-self.R
            self.vx = -self.vx*self.E
```

After loading the Ball class as a module, the main sketch is quite succinct:

```
from Ball import Ball

b = Ball(10) # create Ball object, out here so global

def setup():
    size(200,400)

def draw():
```

```
    background(255)
    b.render()
    b.move()
```

The file structure here assumes that in Processing you created a *New Tab* with a file called `Ball.py` and placed the Ball class definition inside.

16.5. Glossary

frame rate

The rate at which the draw() function executes and updates the display.

pixel

A single picture element, or dot, from which images are made.

processing

An open source programming environment that allows the use of Python syntax to create simple animated, graphical programs.

sketch

A complete Processing program.

16.6. Exercises

1. Have fun with Python Processing.
2. Visit Open Processing to see what kinds of things are possible with Processing.
3. Modify the Newtonian Ball sketch so that a ball of random velocity is placed where the user clicks the mouse.