Introduction to p5.js

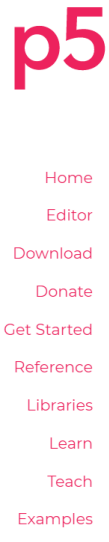
The p5.js coding environment is built on top of javascript as its base language. It is designed to make it as easy as possible to create sophisticated visual designs and animations, especially for use in online applications. If you end up doing more web development, you'll probably do a whole lot more work in both html and javascript, but p5.js lets us skip all the fiddly steps involved in getting a visual sketch coded and online by providing a coding environment directly in the browser with no installation necessary, much like replit.com if you've used that.

If you're new to JavaScript, don't worry – although the syntax is different to, say, Python, all of the fundamental programming ideas are still there (variables, data types, conditionals, loops, functions, objects and classes), and it's much easier to pick up a second coding language than it is to learn to code in the first place!

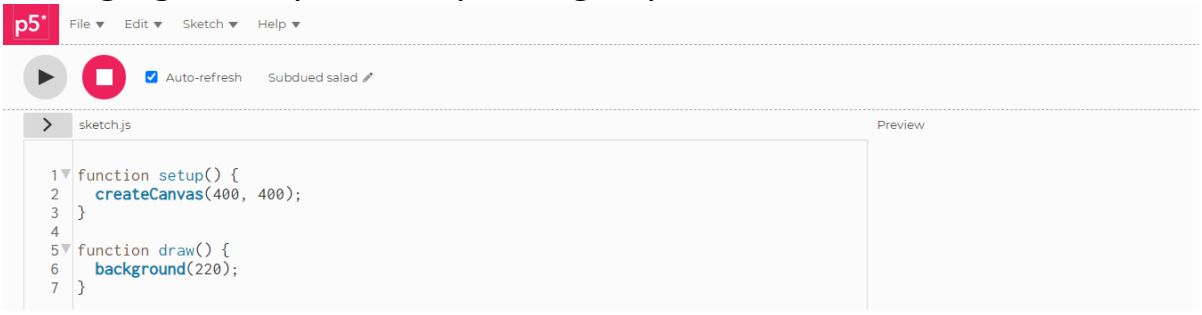
That said, I'll be assuming a certain level of familiarity with the fundamental concepts, so if you get confused by a particular programming idea (eg object oriented programming), I would encourage you to look back at some of the Python tutorials I've made previously.

Finally, before diving in, I want to acknowledge the superb resources provided by Daniel Shiffman (aka "The Coding Train") (see thecodingtrain.com). Pretty much all I've learned about p5.js has come from his excellent tutorial videos, so if my hand-outs ever feel a bit insufficient, I highly recommend following along with some of his tutorials or watching some coding challenge videos.

Navigating the website



In addition to links to *tutorials*, *examples* and the *reference* section (essentially coding documentation), you can hit *'editor'* to start coding right away. When you begin, you'll see a screen like this:



Using the editor

The first thing to notice is that your code is prepopulated with two functions: *setup* and *draw*. Essentially, *setup* will run just once at the start, and then *draw* will be run repeatedly on a loop (although you can stop this under certain circumstances, and control the frame rate). In addition to these two main functions, there are a fair few others which are specifically built into p5.js, such as *mousePressed* which runs when you hit a mouse key.

The other thing you might spot right away, especially if you're coming from a different coding language, is the syntax.
Compare these two ways of saying the same thing, in two different languages:

p5.js

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
}
```

JavaScript functions are invoked using the keyword *function*, followed by the name of the function, then parentheses (containing the function's arguments, if any).

Next follows an open curly braces {

The lines of code to be executed do not need to be indented: blocks of code are indicated by using the close curly braces symbol at the end } . The end of a line is indicated by a semi-colon ;

Normal rules dictate variable names in JavaScript (eg, no spaces, can't start with a number), but by convention use *camelCase*.

Python

```
def setup():
    create_canvas(400, 400)

def draw():
    background(220)

```

Python functions are invoked using the keyword *def*, followed by the name of the function, then parentheses (containing the function's arguments, if any).

Next follows a colon :

The lines of code to be executed need to be indented by consistent tab spacing.

Normal rules dictate variable names in JavaScript (eg, no spaces, can't start with a number), but by convention use *underscore_naming*.

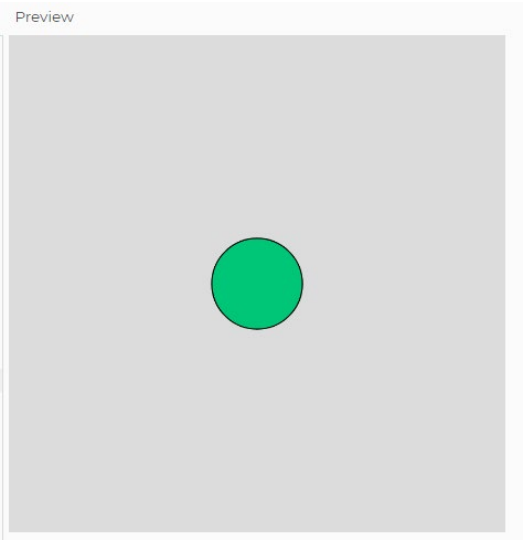
It's worth noting that, although tabs are not a fundamental aspect of JavaScript code syntax (unlike Python, where correct indentation is essential), it is still recommended that you use tabs in a similar way, for clarity and readability, as well as blank lines where it would be helpful.

Python comments are preceded by a #. In JavaScript, use // (double slash). It goes without saying that comments are really useful, regardless of language.

Your first p5.js sketch

The easiest way is to dive in and start playing around. Hopefully this starter sketch will give you a taste for what p5.js can do, and give you a template to build from.

```
1  let x = 0;
2
3  function setup() {
4    createCanvas(400, 400);
5  }
6
7  function draw() {
8    frameRate(20);
9    x += 1;
10   background(220);
11   fill(0, mouseX, mouseY);
12   circle(width / 2, height / 2, x);
13 }
```



Before reading on, see how much of this code you can make sense of for yourself.

The *createCanvas* function is required at the beginning to give a place upon which your drawings and animations will appear. Remember your sketch is designed to be embedded into a webpage. The first argument gives the width (in pixels), the second the height.

One important difference between JavaScript and Python (in fact, between pretty much any programming language and Python) is that variables must be declared before use. The declaration happens at the very beginning with the key word *let*, so that the variable defined is a global variable, accessible within the *setup* or *draw* functions, for instance.

The main job of the *draw* function in this sketch is to show us a circle (line 12), with three arguments: the x and y coordinates of the centre and the diameter (note: not the radius! This is presumably to ensure consistency with other shapes, where width is used). Notice that *width* and *height* are variables that automatically exist in p5.js, and refer to the width and height of the canvas. Very handy if you want to change the size of your canvas later.

Similarly, *mouseX* and *mouseY* are pre-defined variables we can access at any time. Notice how the colour of the circle changes depending on the mouse position. The *fill* function sets the current fill colour, which will then stay the same until or unless *fill* is called again (in this case, every time *draw* runs, which is 20 times a second (you can leave out the *frameRate* call to run at the default frame rate, but it's often really helpful to slow down the action when trying to debug code or make sense of what's going on in a sketch if things aren't behaving as expected. To make things smoother, increase the value passed to *frameRate*. If you things then move too fast, modify the increment used on line 9. One nice thing about the way p5.js works is that it won't throw errors if you give it non-integer coordinates, or even colour values bigger than 255 (it just maxes out at 255).

The classic bouncing ball

One of the very first things to make in a visual coding sketch like this might be a bouncing ball. We're going to keep track of a few variables here, to determine the current position and current velocity of our ball, so start by declaring them at the very beginning:

```
let x, y, vx, vy;
let d = 50;
```

Notice that I've declared these first four variables (all at once, too) but not yet instantiated them with any values. In fact, at this point, the code doesn't even know their data types! We can give them values (I've done so for the circle diameter, for instance), but if I want to use some of the p5.js built-in variables like *mouseX* or *height* I need to do that inside the *setup* function code.

```
function setup() {
  createCanvas(400, 400);
  x = random(d/2, width - d/2);
  y = random(d/2, height - d/2);
  vx = random(30);
  vy = random(20);
}
```

The built-in *random* function (no importing of modules necessary) can take a few different arguments: *random()* returns a random float between 0 and 1, *random(a)* returns a random float between 0 and a, *random(a,b)* returns a random float between a and b, and finally *random(arr)* returns a random element from an array.

The circle itself will be drawn in the *draw* function, using x and y as the coordinates of the centre. Since these values will change, we need the circle to be repeatedly redrawn at the new x and y coordinates. We'll also make it so that x and y are incremented by vx and vy.

```
function draw() {
  background(220);
  circle(x, y, d);
  x += vx;
  y += vy;
}
```

All that remains is to implement the 'bounce' behaviour. Whenever the ball gets within one radius (d/2) of a boundary, I want to reverse its direction (and, to ensure I don't get strange behaviour if speed starts to change, also reverse the last move that sent it outside the boundary).

```
function draw() {
  background(220);
  circle(x, y, d);
  x += vx;
  y += vy;
  if (x > width - d/2 || x < d/2) {
    x -= vx;
    vx *= -1;
  }
  if (y > height - d/2 || y < d/2) {
    y -= vy;
    vy *= -1;
  }
}
```

Notice how the *if* conditional syntax works: the condition goes in brackets after the *if* key word, and the code block that follows is enclosed in curly braces { }.

Also note that *or* is || (double pipe character), and *and* is && in JavaScript.

Now try: Comment out `// background(220);` to see what happens.

What would happen if we include `vy += 1` each time we loop through *draw*?

Try multiplying vx and vy both by 0.99 each time to simulate air resistance.