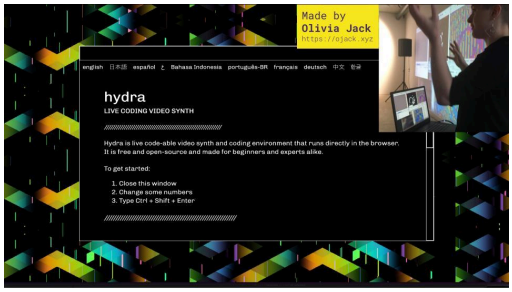
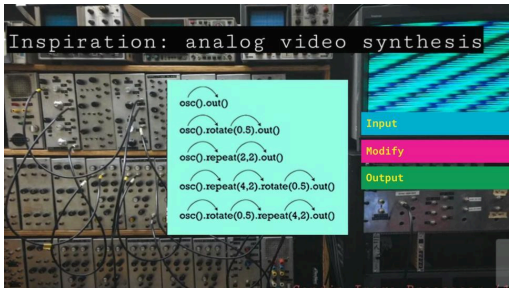
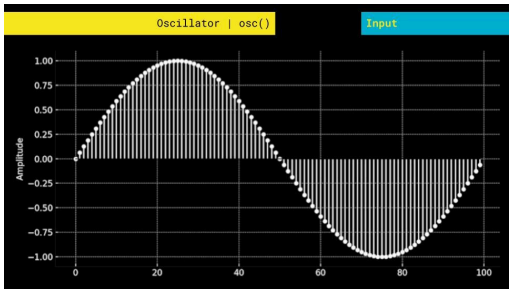
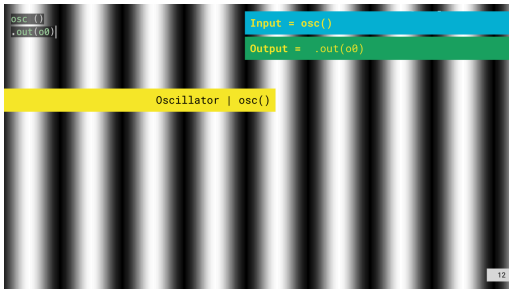
Slide	Code	Notes
		**Hydra Website** https://hydra.ojack.xyz/ **Pro Tip** Workshop display: Two-screen split; one screen with Hydra, the other with workshop slides.

		Run all code - Runs all code on the page ***ctrl+shift+enter** Clear all - Resets the environment and clears text from the editor. Load library or extension - community extensions for Hydra-Synth. Show random sketch - Loads random sketch examples. Make random change - Modifies a single value automatically. upload to gallery - Upload a sketch to Hydra's gallery and create a shorter URL. show info window - Show overlay window with help text and links.
		Inspired by analog modular synthesizers, these tools are an exploration into using streaming over the web for routing video sources and outputs in real time. Core principles: Input + Modify + Output Osc is the input rotate, and repeat are the modifiers Modify could be lots of things like: .brightness .rotate .kaleid .invert

		Output = o0 Output that's o0 is like a TV channel, and .out(o0) is telling Hydra, Put my visuals on this channel so it shows up." Send this visual to screen number 0 so I can see it. • The visual you build goes into a "pipe." • .out(o0) opens the pipe and shows it on the main display. • Without .out(o0), Hydra made something—but you wouldn't see it.
		An oscillator is a signal that creates a repeating wave or pattern that goes back and forth — like a smooth pulse. And in Hydra, we're going to use it as an **Input**.




```
osc ()
.out(o0)
```

Hydra expects an input and an output argument.

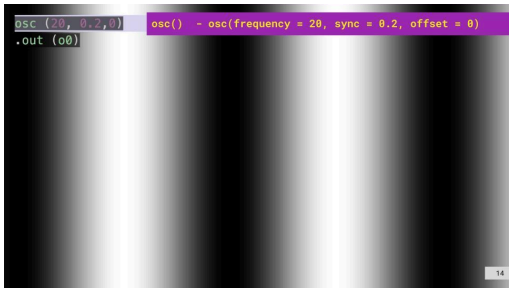
```
osc () // Input
.out(o0) // Output
```

Hydra is written in **JavaScript**, a programming language and a core technology of Websites, alongside HTML and CSS.

It enables **dynamic** and **interactive content** on websites and web applications.

Syntax is the **grammar of code**. It's how we tell the computer what to do step by step.

In Hydra, the code is read from left ot right, top to bottom.




```
osc (20, 0.2,0)
.out(o0)
```

We have 3 **arguments** in **osc()**

osc() - osc(frequency, sync, offset)

Oscillator (frequency = 20, sync = 0.2, offset = 0)

In coding, an argument is information you give to a **function** so it knows how to behave.

- osc() is a function — it creates something.
- The dot . chains actions together.
- Each pair of parentheses () passes arguments (values) that control behavior.

Think of a function like a machine, and arguments are the settings or ingredients you give it. Each function has its own argument.

A parameter is a blank spot a function expects, and an argument is the actual value you fill that spot with when you use the function.

The first **argument 20** is the frequency of the lines,

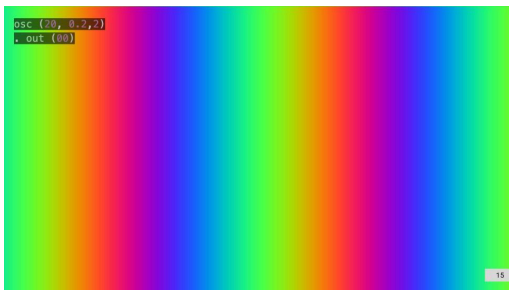
```
osc (20, 0.2,0)
.out (o0)
```

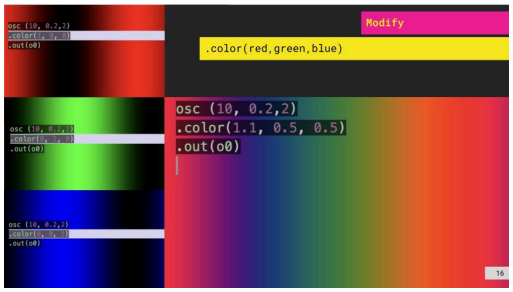
The **second** number is the **synchronization**, known as **speed**.

Put 0.2; they're moving quite slowly.

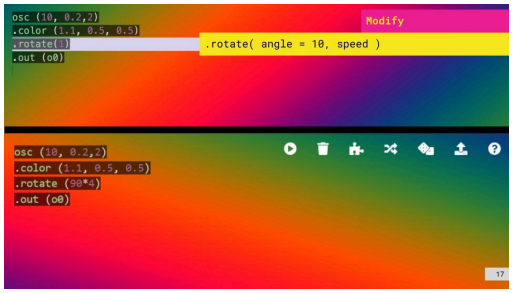
Now put number 0.8, you'll see it begins to move a lot faster.

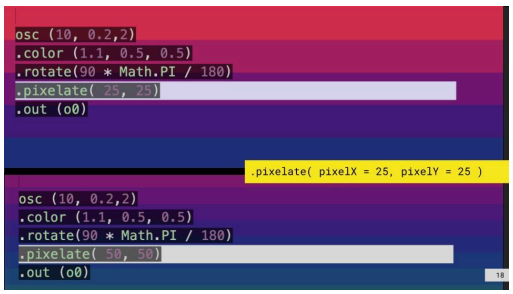
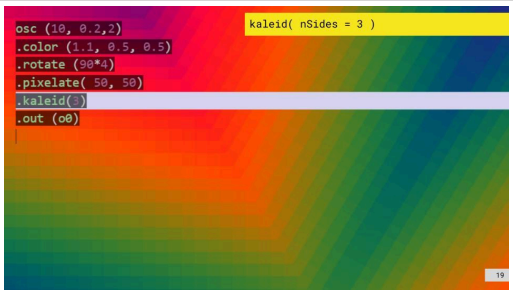
Warning too fast and it will flash!

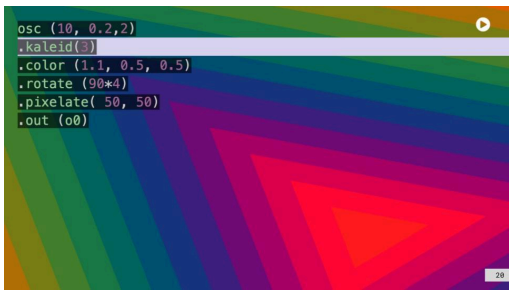
	osc (20, 0.2,2) .out (o0)	The last number is what's known as the offset. 3 oscillators are moving simultaneously. One of them is red, one is green, and one is blue. If you mix all of them, you get white & black. They're moving slightly out of sync with each other, which will reveal the colors of each of those oscillators and mix them all together.

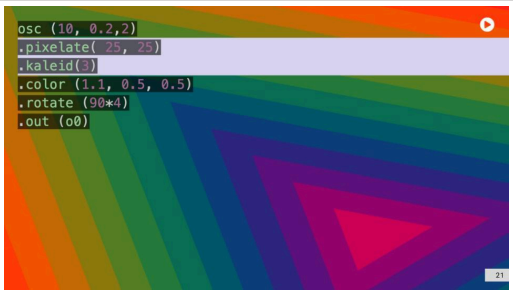
	osc (20, 0.2,2)
	.color(1, 0, 0) // Red
	.out(o0)
	
	//
	
	osc (20, 0.2,2)
	.color(0, 1, 0) // Green
	.out(o0)
	
	//
	
	osc (20, 0.2,2)
	.color(0, 0, 1) // Blue
	.out(o0)
	
	//
		.color(red, green, blue, alpha)
		
		Pro Tip
		
		Two forward slashes create a comment.
		
		Add a note to the line of code without breaking the syntax.

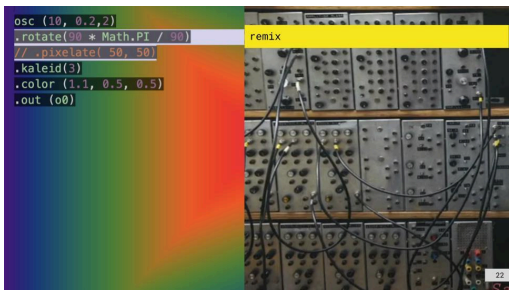
	osc (20, 0.2,2)
	.color(1.1, 0.5, 1) // MIX
	.out(o0)	

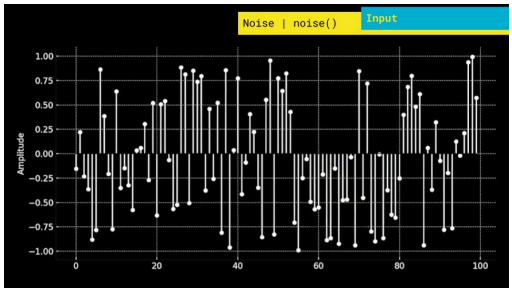
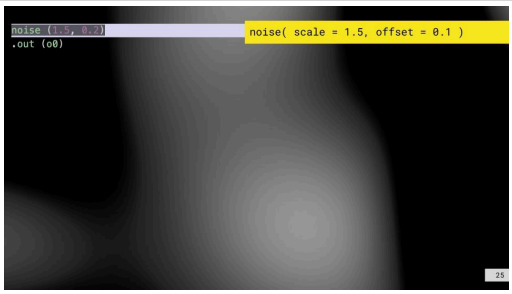
 	osc (10, 0.2,2) .color (1.1, 0.5, 0.5) **.rotate(1)** .out (o0) // osc (10, 0.2,2) .color (1.1, 0.5, 0.5) **.rotate(90 * Math.PI / 25)** .out (o0) // osc (10, 0.2,2) .color (1.1, 0.5, 0.5) **.rotate (90*4)** .out (o0)	.rotate(angle = 10, speed) Rotation is measured in radians is like a slice of a pie. 3.14 is pi, a full 360-degree rotation would be 3.14 times x2, JavaScript / Hydra takes math values.

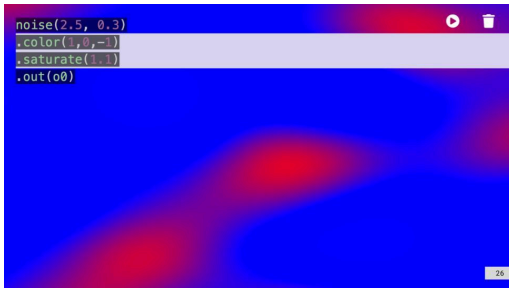
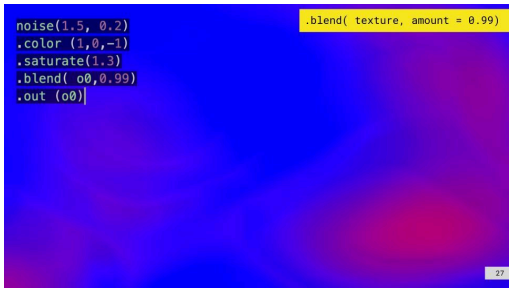
	osc (10, 0.2,2) .color (1.1, 0.5, 0.5) .rotate (90*4) **.pixelate(25, 25)** .out (o0)	.pixelate(pixelX = 25, pixelY = 25)
	osc (10, 0.2,2) .color (1.1, 0.5, 0.5) .rotate (90*4) .pixelate(50, 50) **.kaleid(3)** .out (o0)	kaleid(nSides = 3)

	osc (10, 0.2,2)
	.kaleid(3)
	.color (1.1, 0.5, 0.5)
	.rotate (90*4)
	.pixelate(50, 50)
	.out (o0)
		Change order.kaleid(3) to the top.
		
		// Modulate order matters in Hydra.
		// Code in Hydra moves from top to bottom

	osc (10, 0.2,2) **.pixelate(25, 25)** **.kaleid(3)** .color (1.1, 0.5, 0.5) .rotate (90 * Math.PI / 180) .out (o0)	Moving modulation effects around and playing with math.

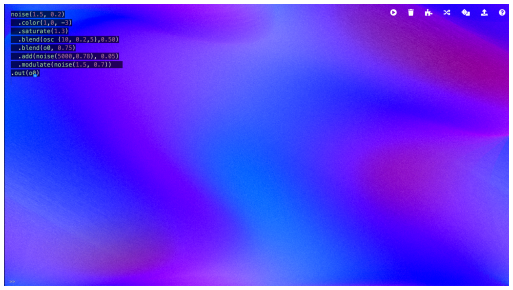
	osc (10, 0.2,2) .kaleid(2) .rotate (90*Math.PI/90) // .pixelate(15, 25) .color (1.5, 0.5, 0.5) .out (o0)	Remix the order, and jam out. Turn lines on and off by commenting with 2 forward slashes
		**Short Cuts** **Run All** control + shift + enter **Hide Code** control + shift + h

		In Hydra, the noise() function is one of the core source generators — it creates a dynamic field of random pixel values that shift and flow over time. You can think of it like digital static or clouds that move and evolve. It's often used as a base texture or as a modulator to distort other visuals.
	noise(1.5, 0.2) .out (o0)	

	noise(2.5, 0.3) **.color(1,0,-1)** **.saturate(1.1)** .out(o0)	Bring **color** with **.color** under the noise input and **.saturate** to pop up the **saturation.**
	noise(1.5, 0.2) .color (1,0,-1) .saturate(1.3) **.blend(o0,0.99)** .out (o0)	Using the Output = o0 in .blend creates a feedback loop. It's like pointing a camera into its own output. Or like layering two images with transparency. Instead of clearing the screen every moment, the new image is built on top of the old image. Each frame leaves a ghost of itself behind.

	noise(1.5, 0.2) .color (1,0,-1) .saturate(1.3) **.blend(osc (10, 0.2,2),0.50)** // .blend(o0,0.75) .out (o0)	.blend(texture, amount = 0.50) Since the 1st argument can take a texture we could use the oscillator as a texture input. Texture is the graphical source. Commenting out the feedback loop to see what's happening.
	noise(1.5, 0.2) .color (1,0,-1) .saturate(1.3) .blend(osc (10, 0.2,2),0.50) **.add(noise(5000,0.78),0.05)** .blend(o0,0.75) .out (o0)	.add(texture, amount = 1) Adding .add noise at a very high number add texture. When you "add a texture," you're not just stacking an image — you're passing a texture as an input into another function. That function then uses the texture to modify, mask, or mix visuals

	noise(1.5, 0.2)
	  .color(1,0,-1)
	  .saturate(1.3)
	  .blend(osc(10, 0.2, 2), 0.50)
	  .modulate(osc(15,0.3), 0.4)
	// <- more modulation (0.1–1 range)
	  .add(noise(5000,0.78), 0.05)
	  .blend(o0, 0.75)
	.out(o0)
		For more of a water feel and less repetition we can use .modulate
		
		Modulate functions use the colors from one source to affect the geometry of the second source. This creates a sort of warping or distorting effect.
		
		. modulate() does not change color or luminosity but distorts one visual source using another visual source.
		
		An analogy in the real world would be looking through a texture glass window or water
		
		You can add a second parameter to the modulate() function to control the amount of warping: modulate(o1, 0.9).
		
		In this case, the red and green channels of the oscillator are being converted to x and y displacement of the camera image.
		
		usually by shifting or warping its pixels.
		Think of it like bending one image with another image's energy — the second texture becomes a map that tells Hydra how to distort the first one.

	noise(1.5, 0.2)
	  .color(1,0, -3)
	  .saturate(1.3)
	  .blend(osc (10, 0.2,5),0.50)
	  .blend(o0, 0.75)
	  .add(noise(5000,0.78), 0.05)
	  .modulate(noise(1.5, 0.7))
	.out(o0)
		Final Video FLyer