Programming Concepts Overview

Functions

SIDE NOTE

Functions – definitions

When we see:

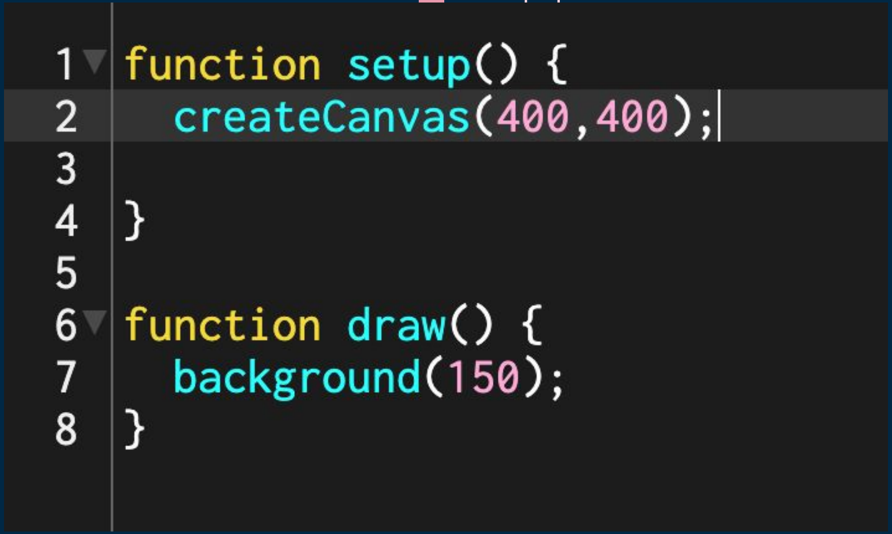
```
function setup() {}
```

and

```
function draw() {}
```

These are **function definitions**.

We are not learning how to define functions yet, but these definitions are necessary in P5JS in order to see the "canvas" and in order to "draw" images on it.



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

}

function draw() {
  background(150);
}
```

SIDE NOTE

Functions – definitions

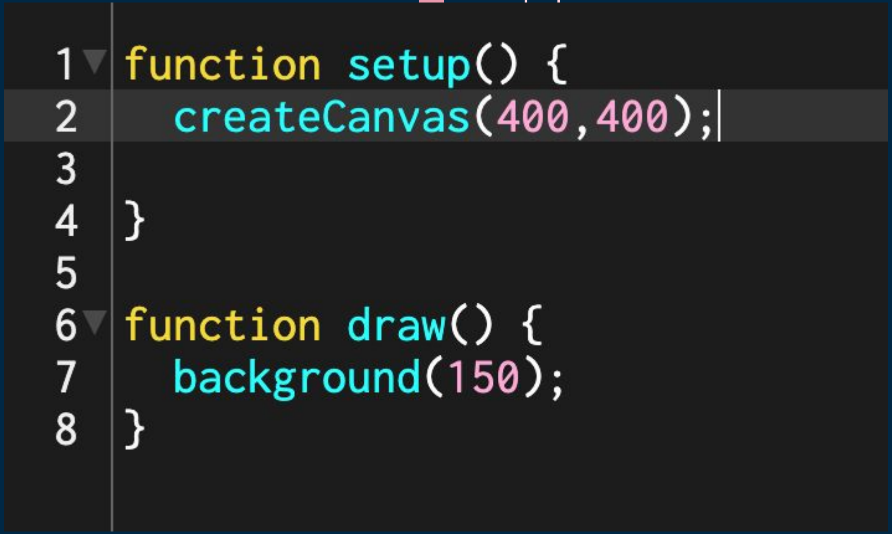
function setup() {}

is called once – it sets up the canvas one time.

function draw() {}

is a loop. It is called repeatedly ever millisecond or so.

This **allows for animation.**



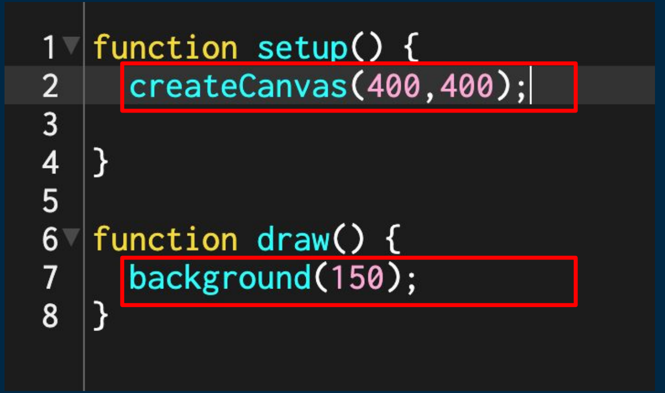
```
function setup() {
  createCanvas(400,400);

}

function draw() {
  background(150);
}
```

Functions - function calls

For now, **ignore the function definitions**. When we talk about functions, we are referring to functions like those on the example:

createCanvas()

background()



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

}

function draw() {
  background(150);
}
```

The **( ) are how we "call" the function.**

The ( ) mean it is a function that **does something**.

For example, background(150) paints a grey square over the whole canvas.

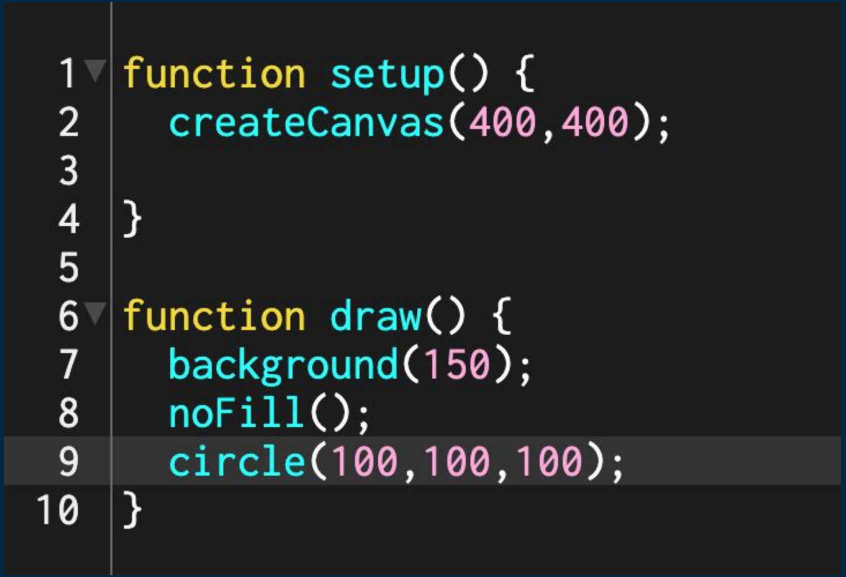
Functions - arguments

Inside a function we give **arguments**.

We can give 0, 1, or more arguments.

For example: **`noFill()`** is a function that has no arguments

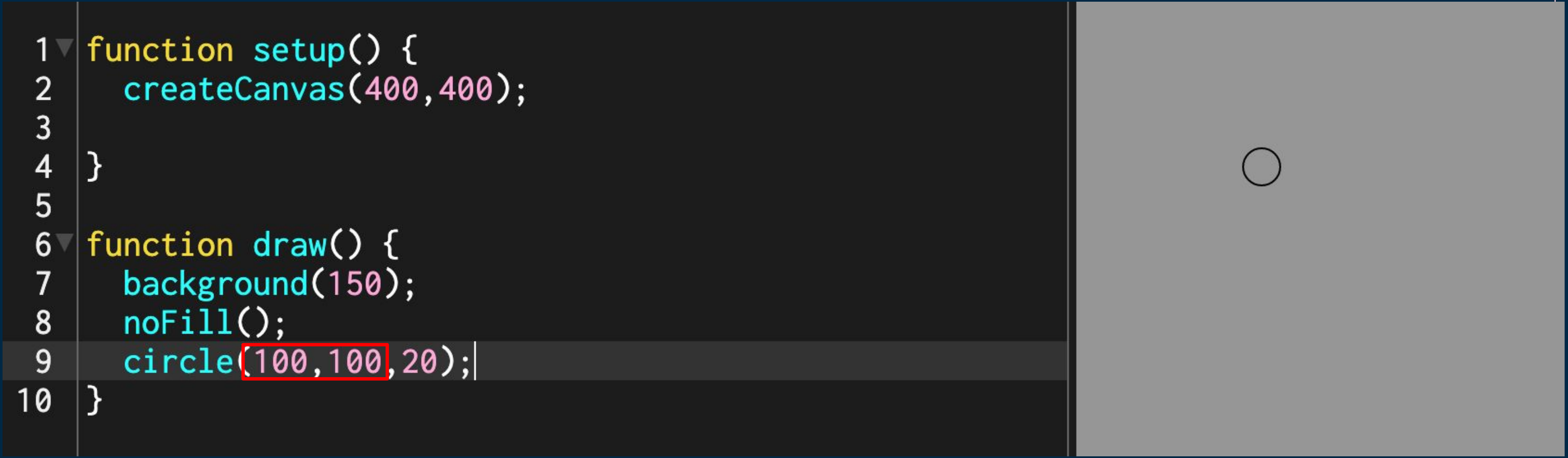
And **`circle()`** is a function that takes 3 arguments



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

}

function draw() {
  background(150);
  noFill();
  circle(100,100,100);
}
```

Variables

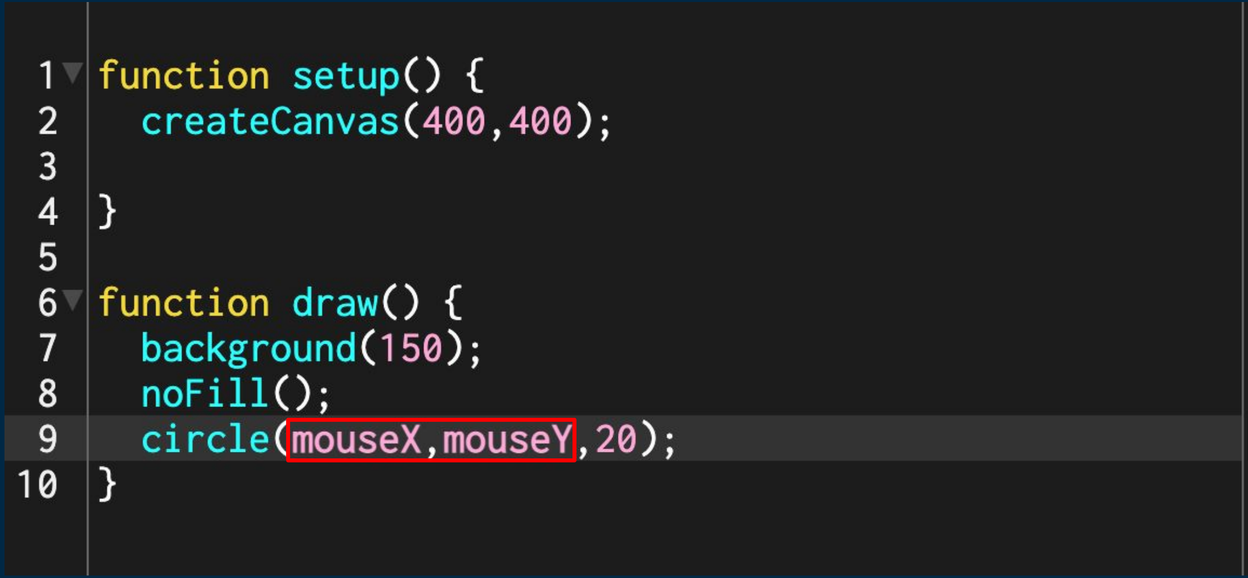
Below is an example of **hard-coding**. We place our circle at the x,y coordinate of (100,100)



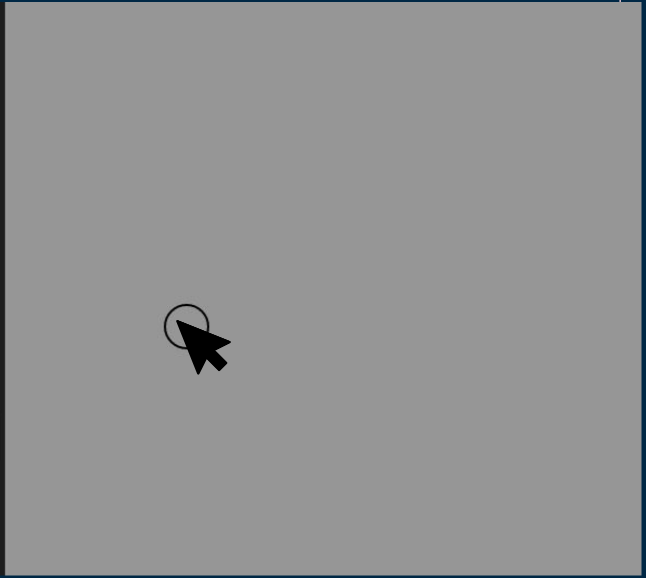
```
function setup() {
  createCanvas(400,400);

}

function draw() {
  background(150);
  noFill();
  circle(100,100,20);
}
```

If we replace the x,y coordinates with **mouseX** and **mouseY**, then we can tell the circle to move with the cursor.

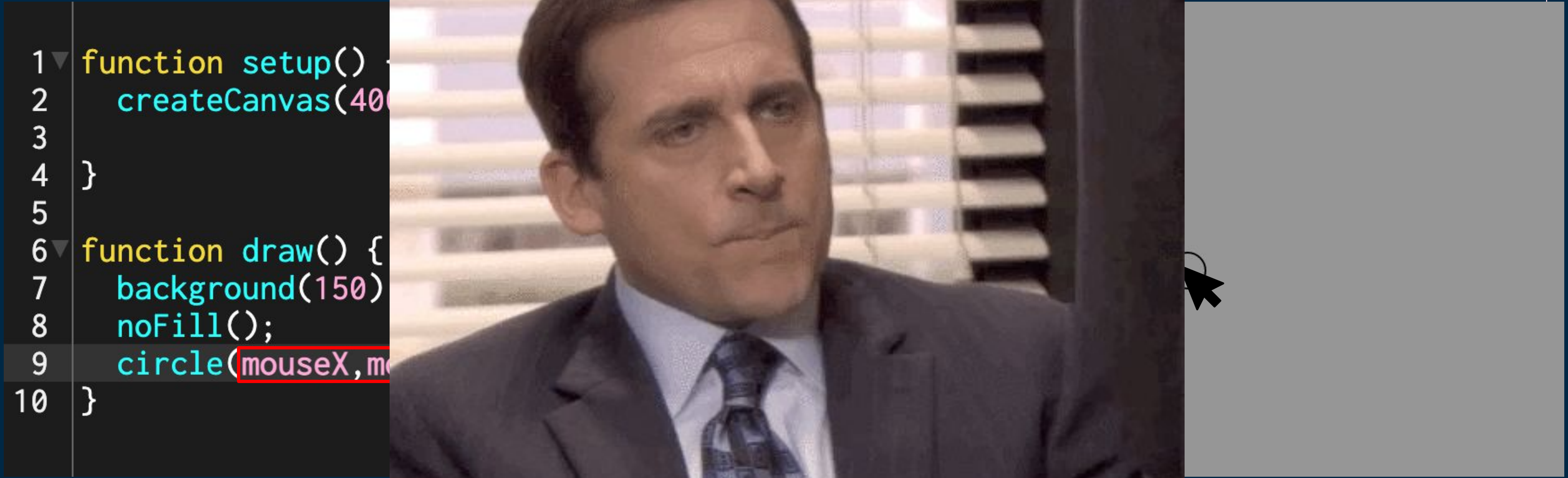
This is considered “**soft-coding**”, meaning the value of the variables changes each time our loop runs.



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

}

function draw() {
  background(150);
  noFill();
  circle(mouseX,mouseY,20);
}
```



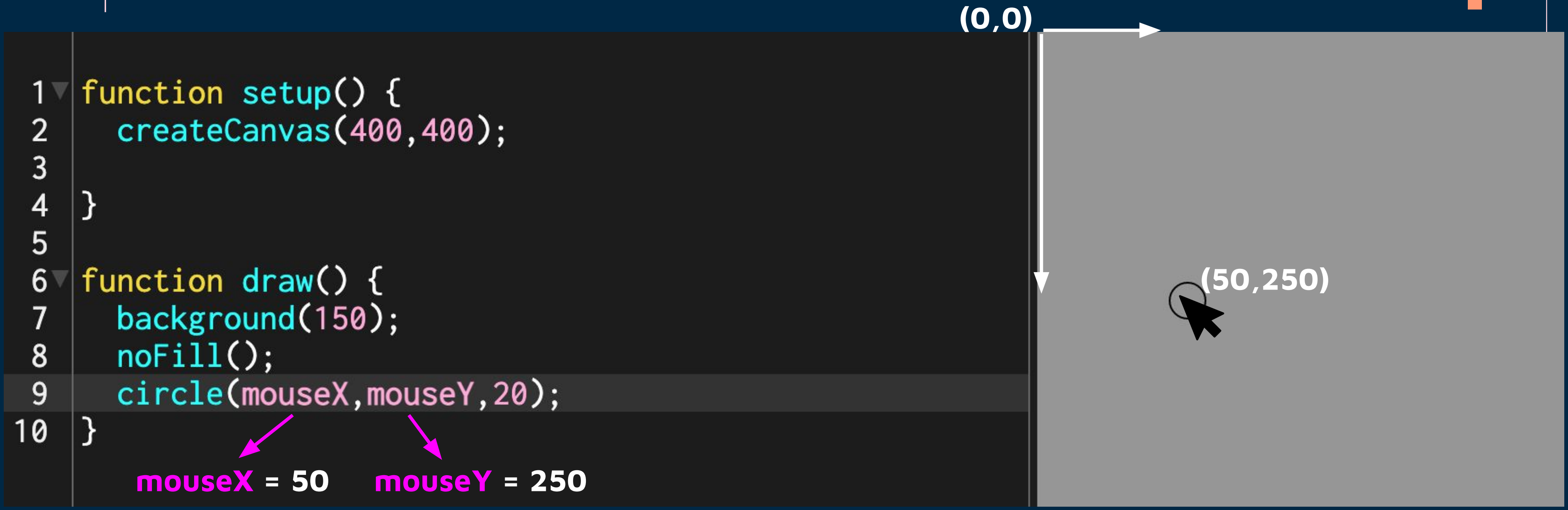
This is because **mouseX** stores the value of the cursor on the x axis, and **mouseY** stores the



```
function setup()
  createCanvas(40

}

function draw() {
  background(150)
  noFill();
  circle(mouseX,m
}
```

This means, if the cursor is moved **50** pixels to the right, and **250** pixels down, the circle will move to that position.

As the **draw()** function repeats every millisecond, and the cursor moves, the circle will appear to follow the mouse.



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

}

function draw() {
  background(150);
  noFill();
  circle(mouseX,mouseY,20);
}
```

x,y coordinates = 50,250

Creating our own variables

declare **assign** **reassign**

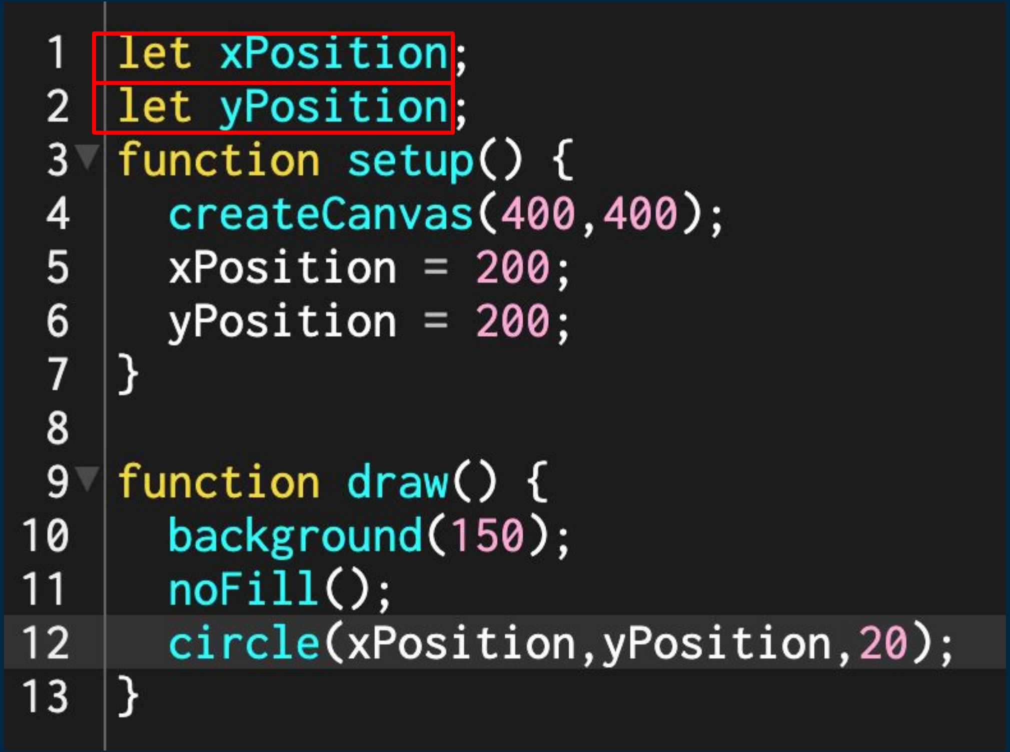
$\downarrow$

use

We can create our own variables to store a custom value that we assign to it.

First we have to **declare** the variable. In JS there are a few ways, here we use `let` which creates the variable.

We can name them anything, as long as it **doesn't begin with a symbol, and doesn't have any spaces.**



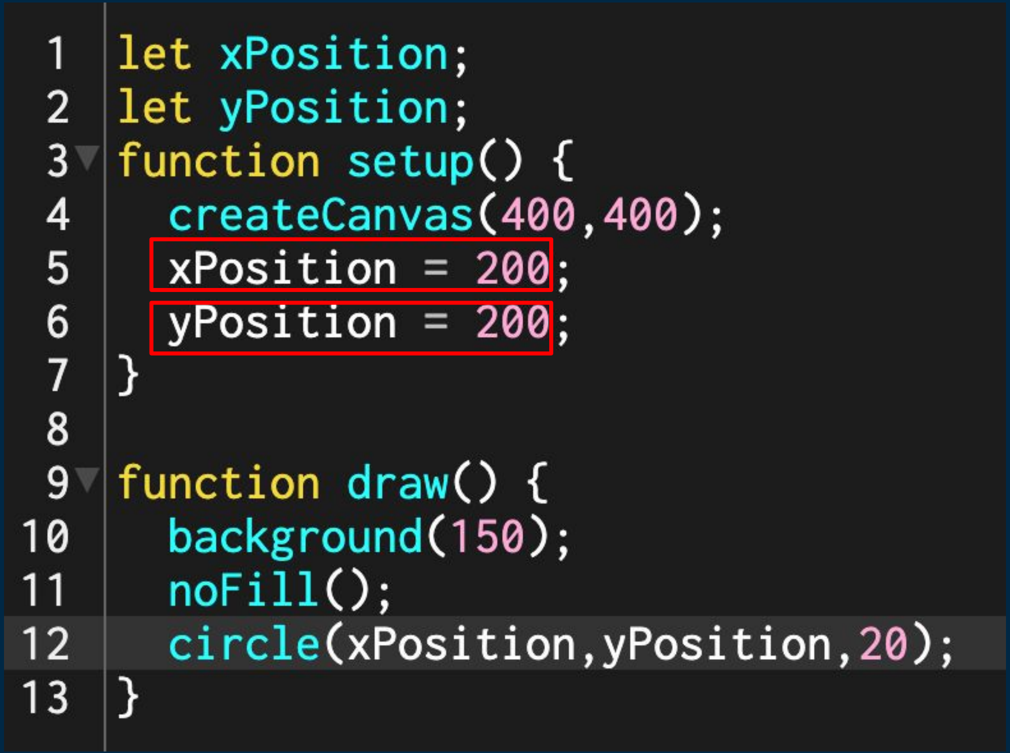
```
let xPosition;
let yPosition;
function setup() {
  createCanvas(400,400);
  xPosition = 200;
  yPosition = 200;
}

function draw() {
  background(150);
  noFill();
  circle(xPosition,yPosition,20);
}
```

Then we have to **assign** the variable.

We give it a starting value by writing

= and a value.

Don't forget, every line of code that isn't a function definition or conditional statement must end with a ;



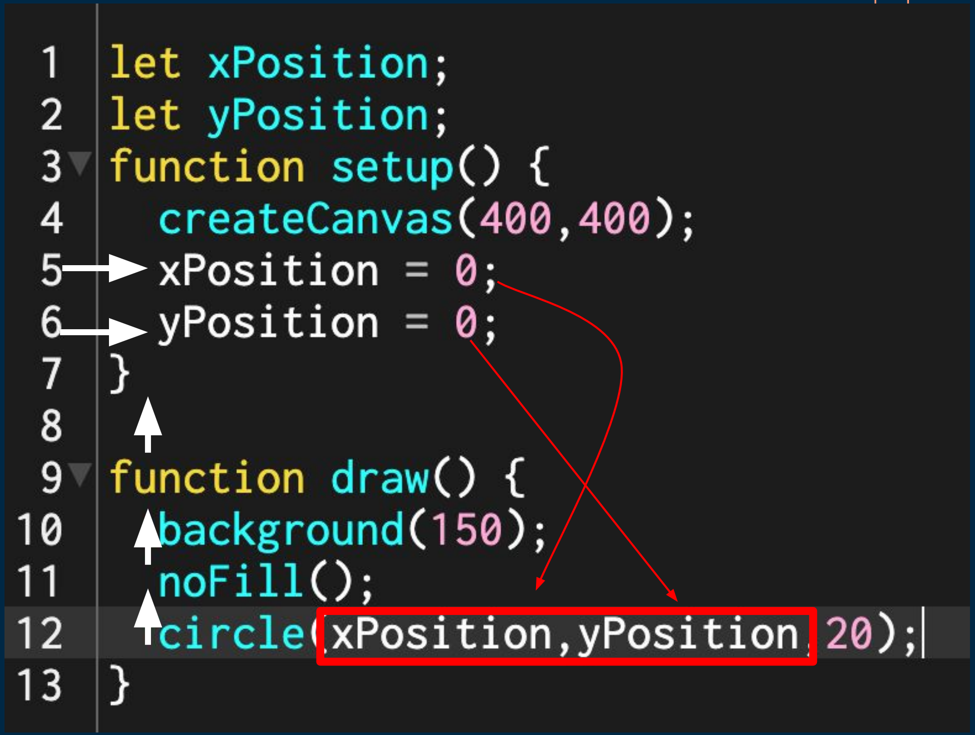
```
let xPosition;
let yPosition;
function setup() {
  createCanvas(400,400);
  xPosition = 200;
  yPosition = 200;
}

function draw() {
  background(150);
  noFill();
  circle(xPosition,yPosition,20);
}
```

Now we can **use** the variable.

Here we replace the x,y coordinates of the center of the circle.

When the code reaches line 12, it looks **above line 12** to find the value of these variables.

It finds the value on line 5 and 6, and uses those values `circle(0,0,20)`.



```
1  let xPosition;
2  let yPosition;
3  function setup() {
4    createCanvas(400,400);
5    xPosition = 0;
6    yPosition = 0;
7  }
8
9  function draw() {
10   background(150);
11   noFill();
12   circle(xPosition,yPosition,20);
13 }
```

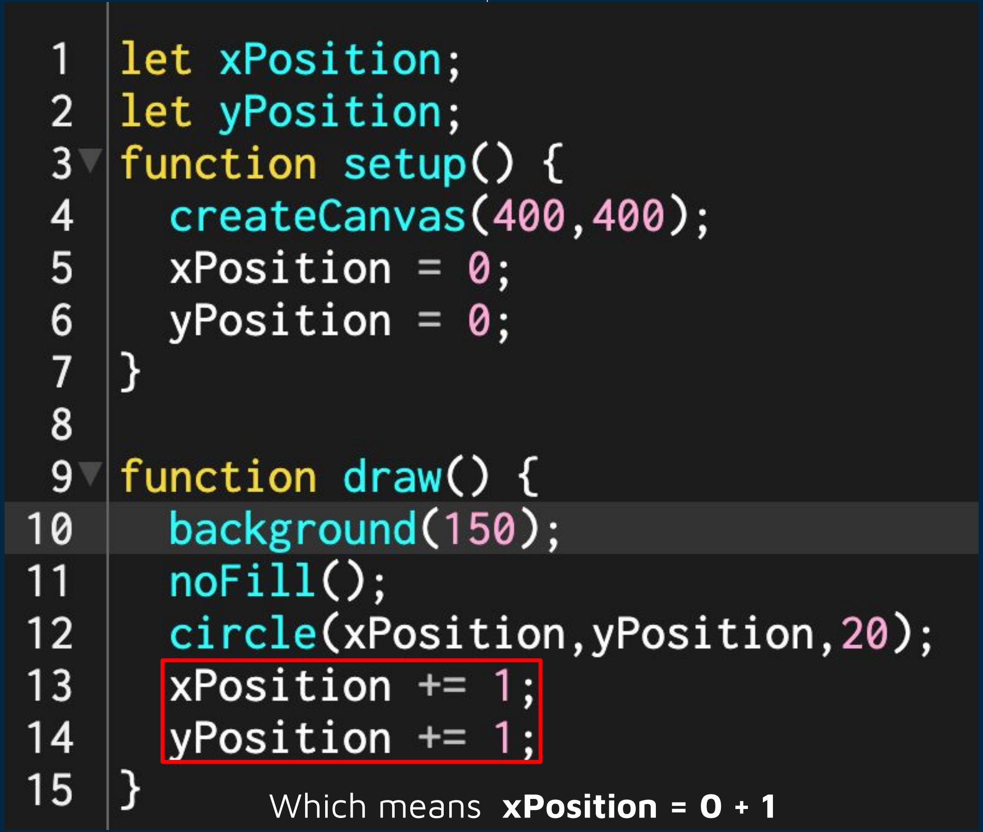
At this point, it seems pointless to do this, but let’s look at how this can be used to animate our circle.

Still inside our draw function, we tell our variables to change by 1 every time the draw loop runs.

`xPosition += 1;` is the same thing as saying

`xPosition = xPosition + 1`

This is **reassignment.**



```
let xPosition;
let yPosition;
function setup() {
  createCanvas(400,400);
  xPosition = 0;
  yPosition = 0;
}

function draw() {
  background(150);
  noFill();
  circle(xPosition,yPosition,20);
  xPosition += 1;
  yPosition += 1;
}
```

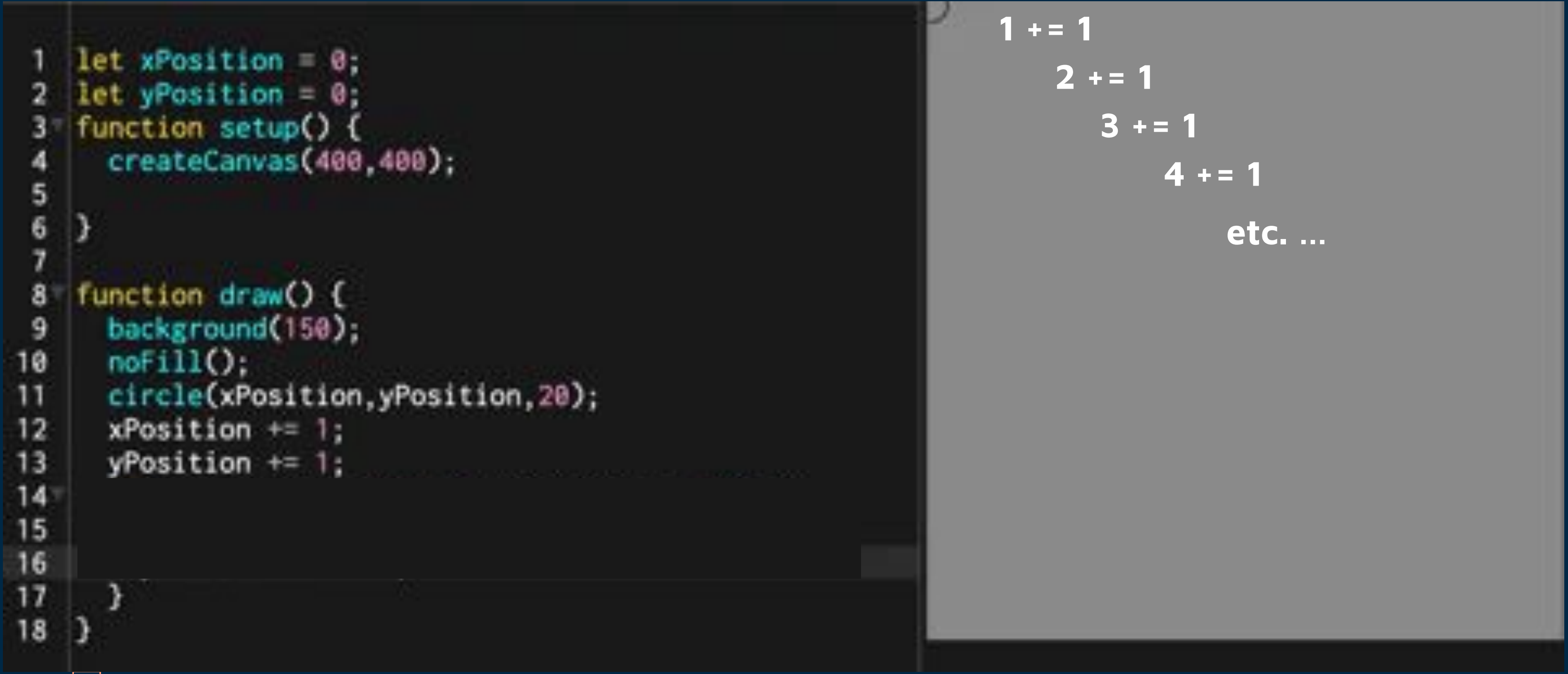
Which means **xPosition = 0 + 1**

Then... xPosition = 1 + 1

Then... xPosition = 2 + 1

etc...

Here's what this does:



```
let xPosition = 0;
let yPosition = 0;
function setup() {
  createCanvas(400,400);

}

function draw() {
  background(150);
  noFill();
  circle(xPosition,yPosition,20);
  xPosition += 1;
  yPosition += 1;



  }
}
```

0 += 1

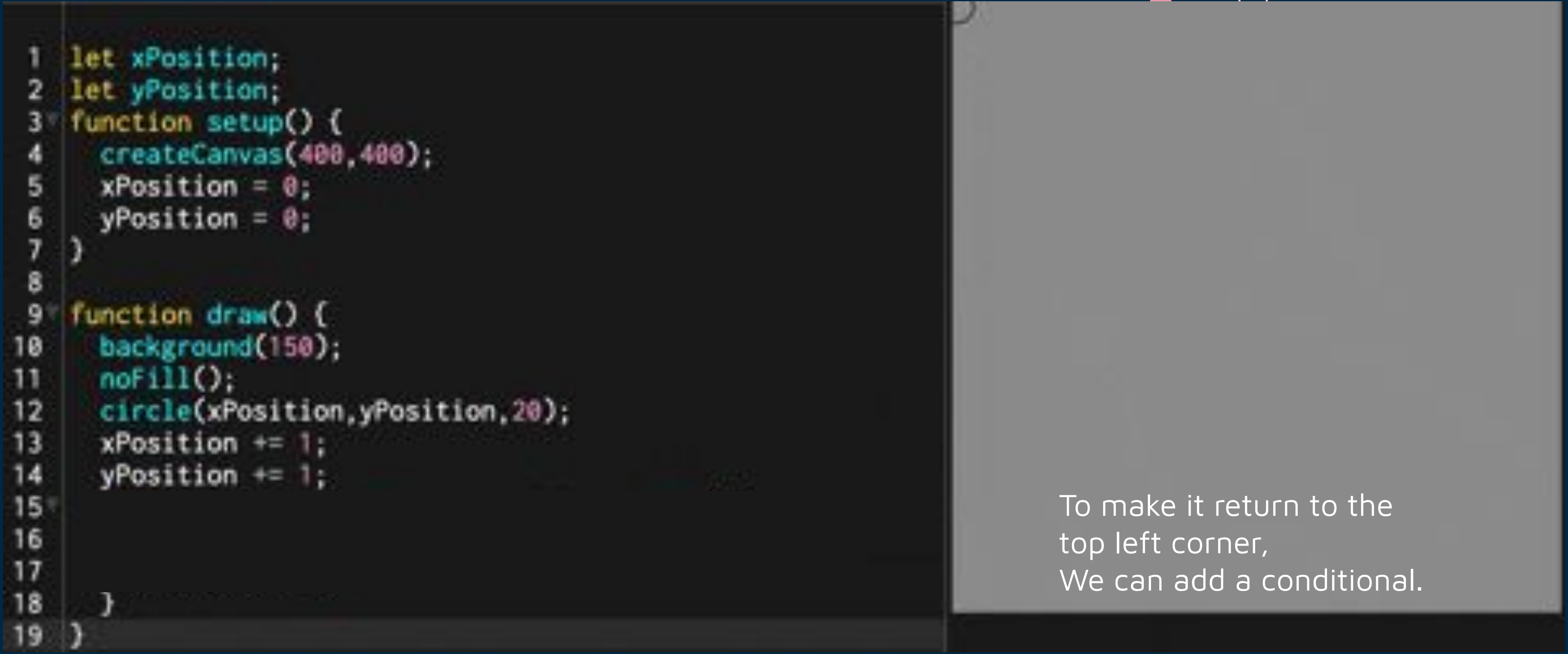
1 += 1

2 += 1

3 += 1

4 += 1

etc. …



```
let xPosition;
let yPosition;
function setup() {
  createCanvas(400,400);
  xPosition = 0;
  yPosition = 0;
}

function draw() {
  background(150);
  noFill();
  circle(xPosition,yPosition,20);
  xPosition += 1;
  yPosition += 1;



  }
}
```

To make it return to the top left corner,
We can add a conditional.

Conditionals – if

Conditionals – if

Conditional if statements always have a specific structure.

For example, is a variable equal to a certain number?

```
if (condition is true) {
    do this code;
}
```

The first curly bracket represents "then". So, if a variable is equal to a certain number, **then…***do this code*

Conditionals – if

Conditional if statements always have a specific structure.

```
if (condition is true) {
    do this code;
}
else {
    do this code;
}
```

We can add an **else** to say, if the first condition is **not true**, then in all other cases, do this code.
(ex: if a variable does not have the given value, always do this other thing)

Conditionals – if

Conditional if statements always have a specific structure.

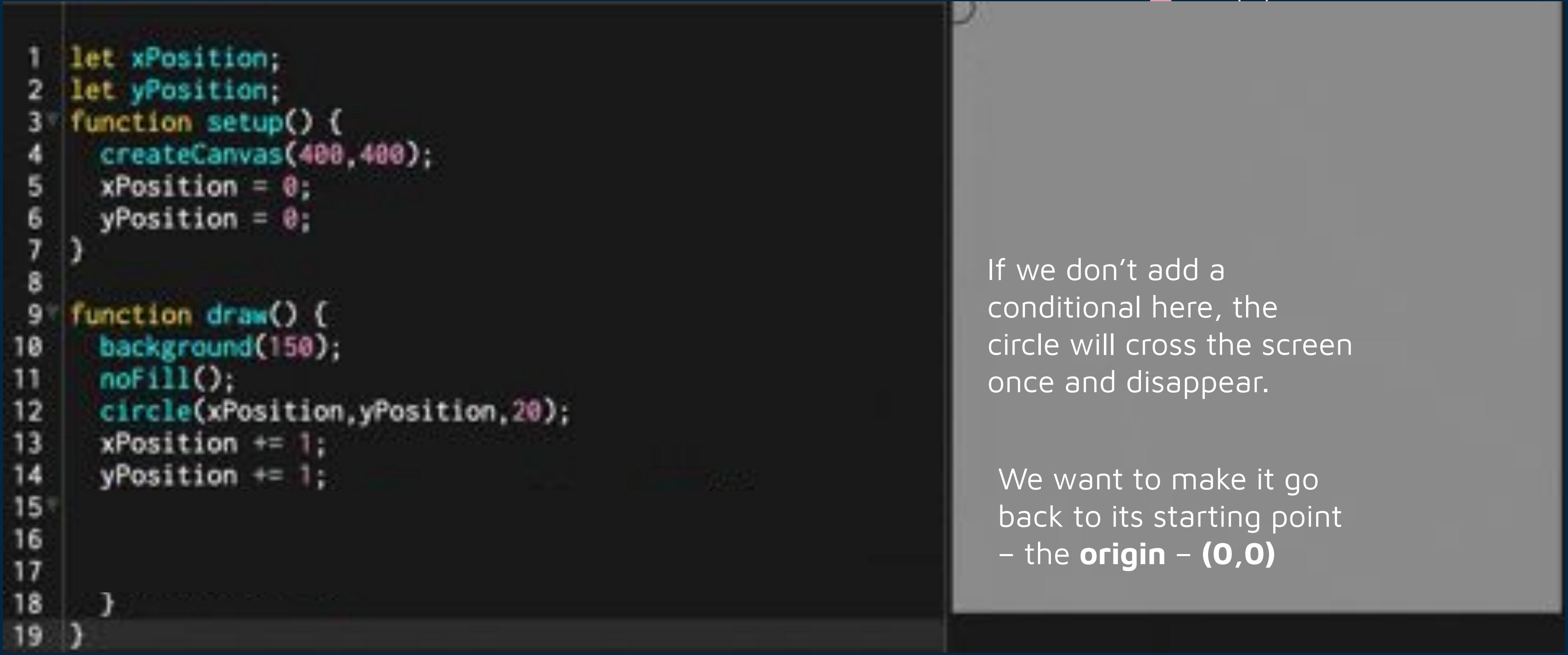
```
if (condition is true) {
    do this code;
}
else if (condition is true) {
    do this code;
}
else {
    do this code;
}
```

In between the if and the else, we can add an `else if` to say, if the first condition is **not true (false)**, then check this condition.

We can add multiple `else if`s here.

If the `if` condition is **false**, the program will check the `else if` conditions, in order.

When a condition is **true**, the code inside runs.



```
let xPosition;
let yPosition;
function setup() {
  createCanvas(400,400);
  xPosition = 0;
  yPosition = 0;
}

function draw() {
  background(150);
  noFill();
  circle(xPosition,yPosition,20);
  xPosition += 1;
  yPosition += 1;



  }
}
```

If we don’t add a conditional here, the circle will cross the screen once and disappear.

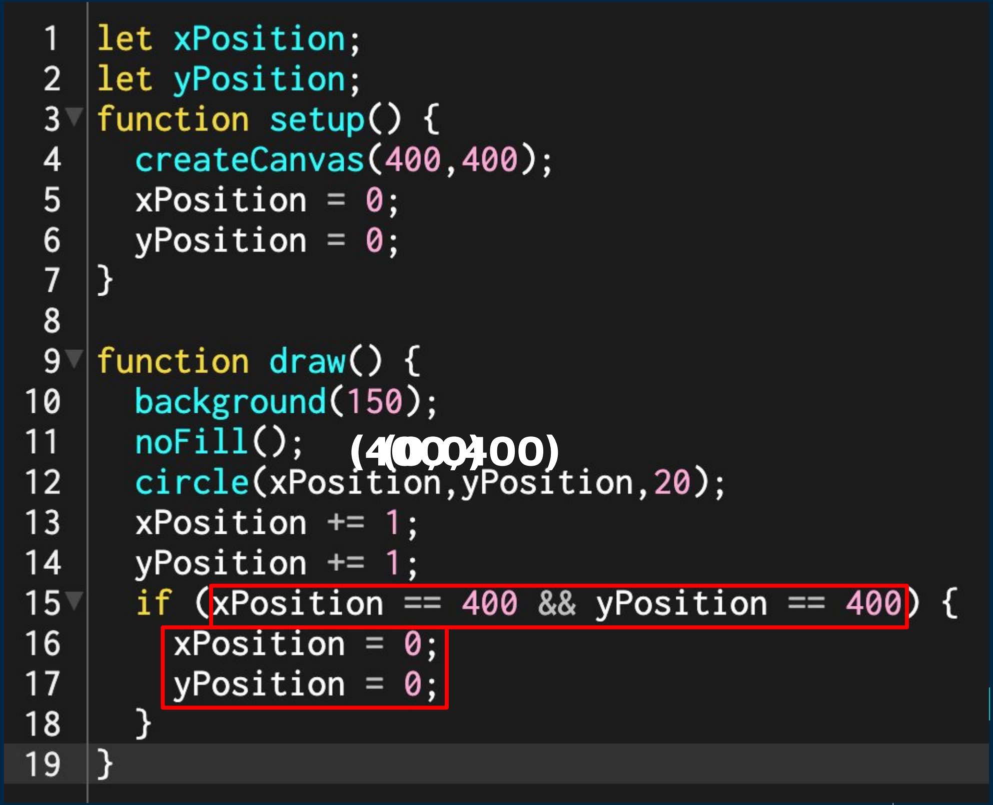
We want to make it go back to its starting point – the **origin** – **(0,0)**

Conditionals

Here we check, is the x and y position of the circle the bottom corner – **(400,400)** ?

If it is equal to **(400,400)** then we want to return it to the origin – the top left corner.

All we have to do is make the x,y coordinates equal to **(0,0)**



```
let xPosition;
let yPosition;
function setup() {
  createCanvas(400,400);
  xPosition = 0;
  yPosition = 0;
}

function draw() {
  background(150);
  noFill();
  circle(xPosition,yPosition,20);
  xPosition += 1;
  yPosition += 1;
  if (xPosition == 400 && yPosition == 400) {
    xPosition = 0;
    yPosition = 0;
  }
}
```

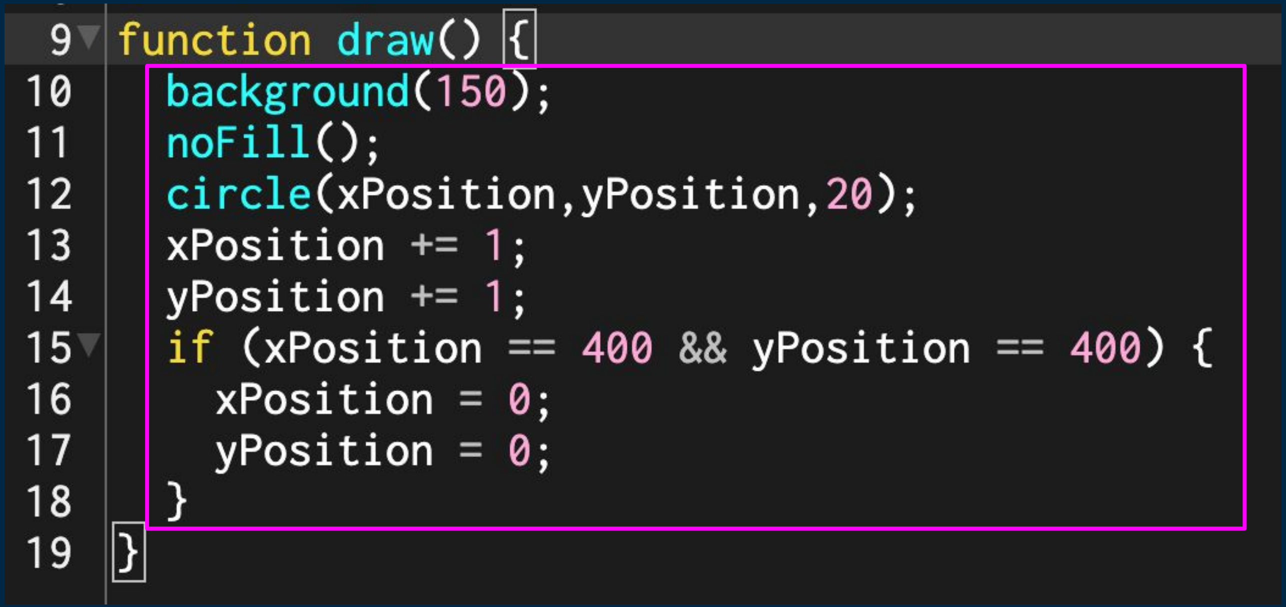
Scope

Scope

The curly brackets in JS represent the scope of the program.

When we click on one, we can see which one matches it.

Here, these curly brackets represent the scope of the `draw()` function.



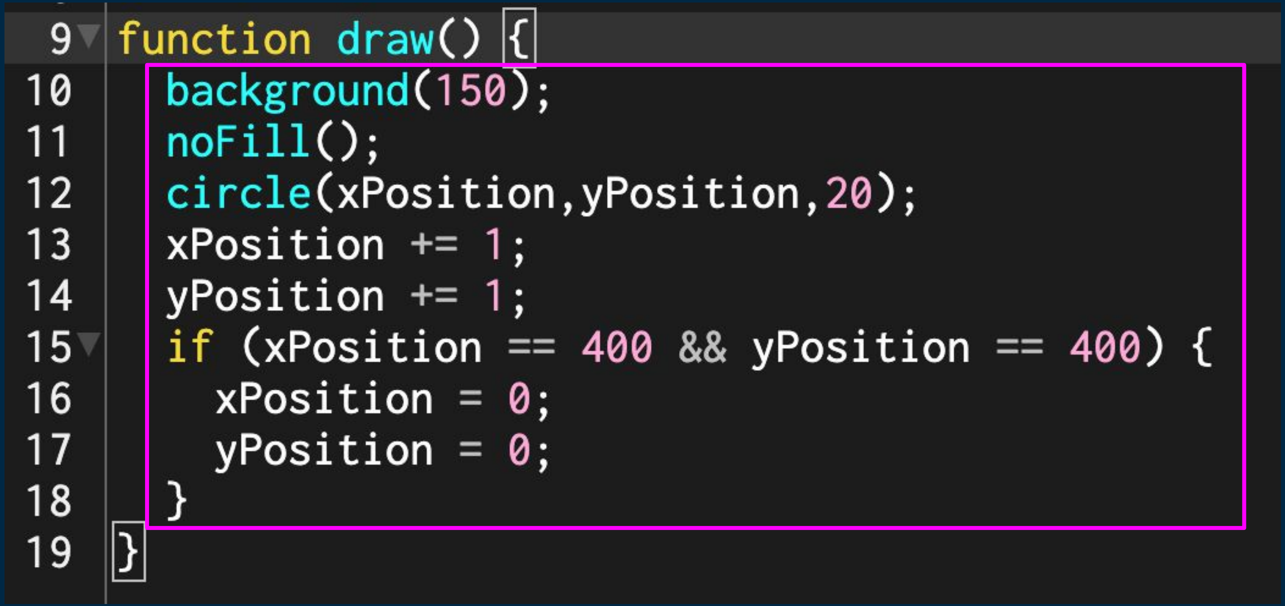
```
function draw() {
  background(150);
  noFill();
  circle(xPosition,yPosition,20);
  xPosition += 1;
  yPosition += 1;
  if (xPosition == 400 && yPosition == 400) {
    xPosition = 0;
    yPosition = 0;
  }
}
```

Scope

The curly brackets in JS represent the scope of the program.

Here, these curly brackets represent the scope of the **draw()** function.

This means, anything we want to “draw” on our canvas, needs to be included **inside** the curly brackets **belonging to the draw() function.**



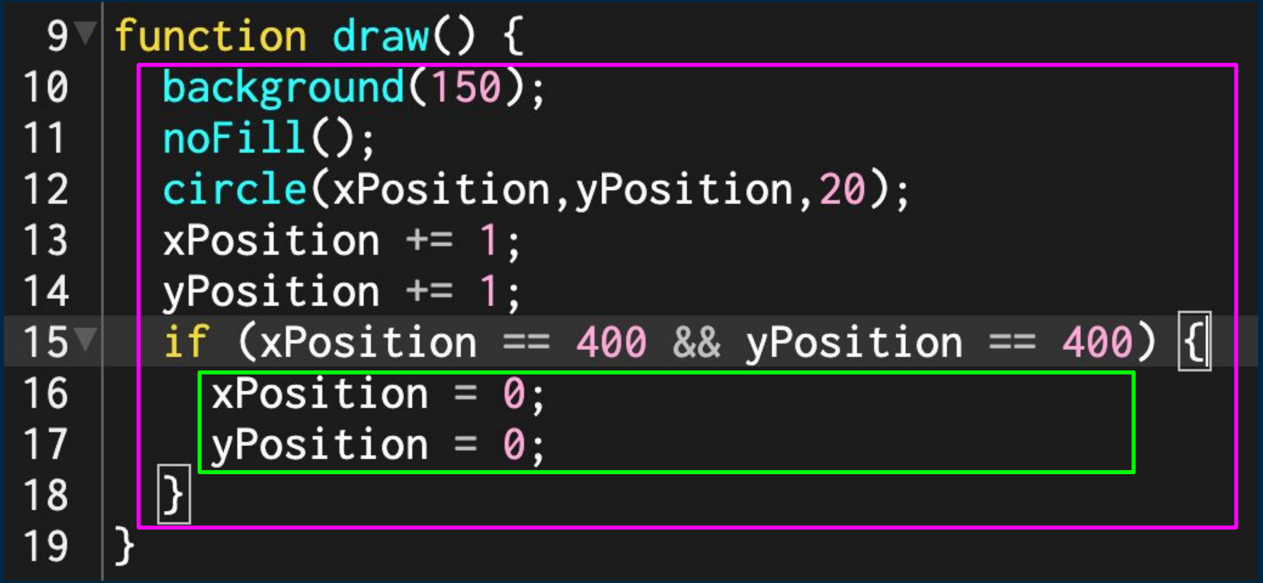
```
function draw() {
  background(150);
  noFill();
  circle(xPosition,yPosition,20);
  xPosition += 1;
  yPosition += 1;
  if (xPosition == 400 && yPosition == 400) {
    xPosition = 0;
    yPosition = 0;
  }
}
```

Scope

The curly brackets in JS represent the scope of the program.

Notice that there are other curly brackets inside the **draw()** function.

These belong to the **if** statement. They represent its scope.



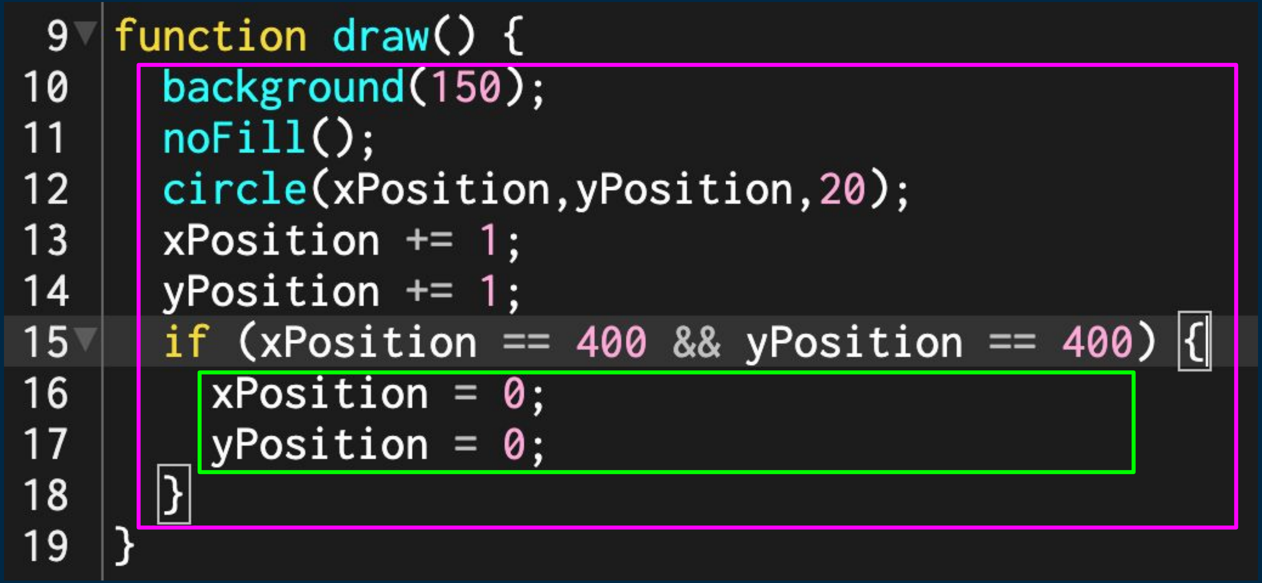
```
function draw() {
  background(150);
  noFill();
  circle(xPosition,yPosition,20);
  xPosition += 1;
  yPosition += 1;
  if (xPosition == 400 && yPosition == 400) {
    xPosition = 0;
    yPosition = 0;
  }
}
```

Scope

The curly brackets in JS represent the scope of the program.

These belong to the **if** statement. They represent its scope.

This means, line 16 and line 17 belong to the **if** conditional.

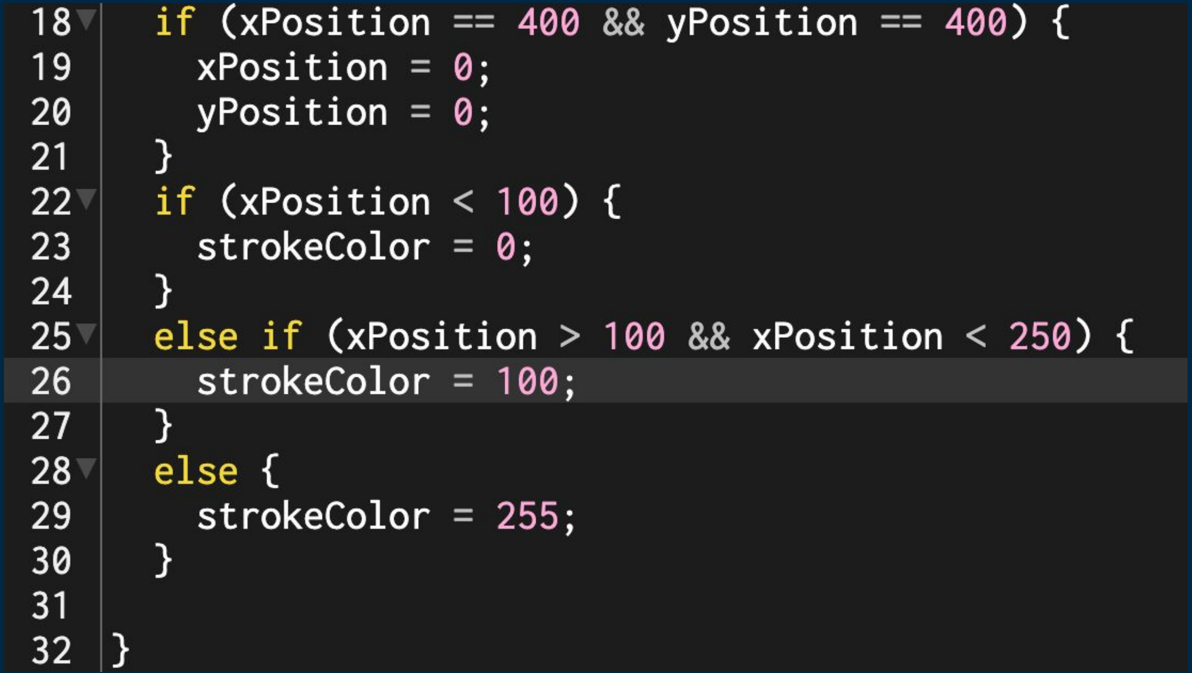


```
 9  function draw() {
10    background(150);
11    noFill();
12    circle(xPosition,yPosition,20);
13    xPosition += 1;
14    yPosition += 1;
15    if (xPosition == 400 && yPosition == 400) {
16      xPosition = 0;
17      yPosition = 0;
18    }
19  }
```

Scope

Here is an example of a conditional sequence that changes the color of the circle in three different zones.

The first two **if**s are two separate conditionals. They both get checked even if one is true.



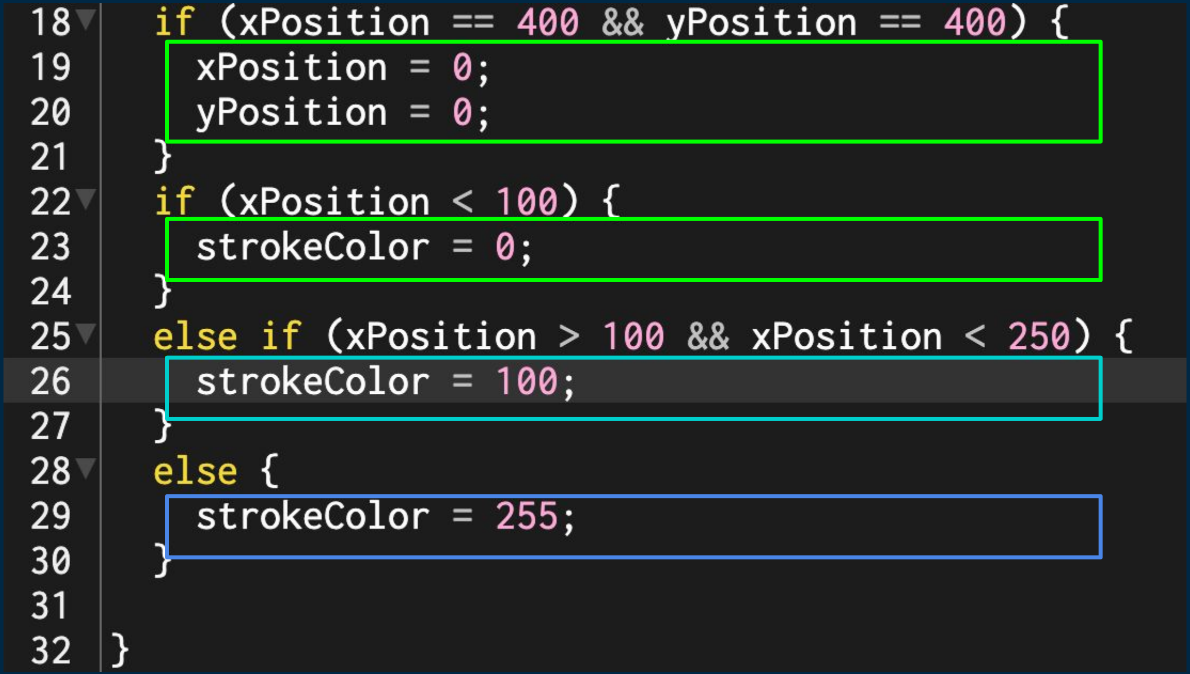
```
  if (xPosition == 400 && yPosition == 400) {
    xPosition = 0;
    yPosition = 0;
  }
  if (xPosition < 100) {
    strokeColor = 0;
  }
  else if (xPosition > 100 && xPosition < 250) {
    strokeColor = 100;
  }
  else {
    strokeColor = 255;
  }

}
```

Scope

Each **if** and **else if** and **else** has their own scope.

Remember, when we click on one curly bracket, we can see which one matches it.

This shows us the scope of a function or a conditional.



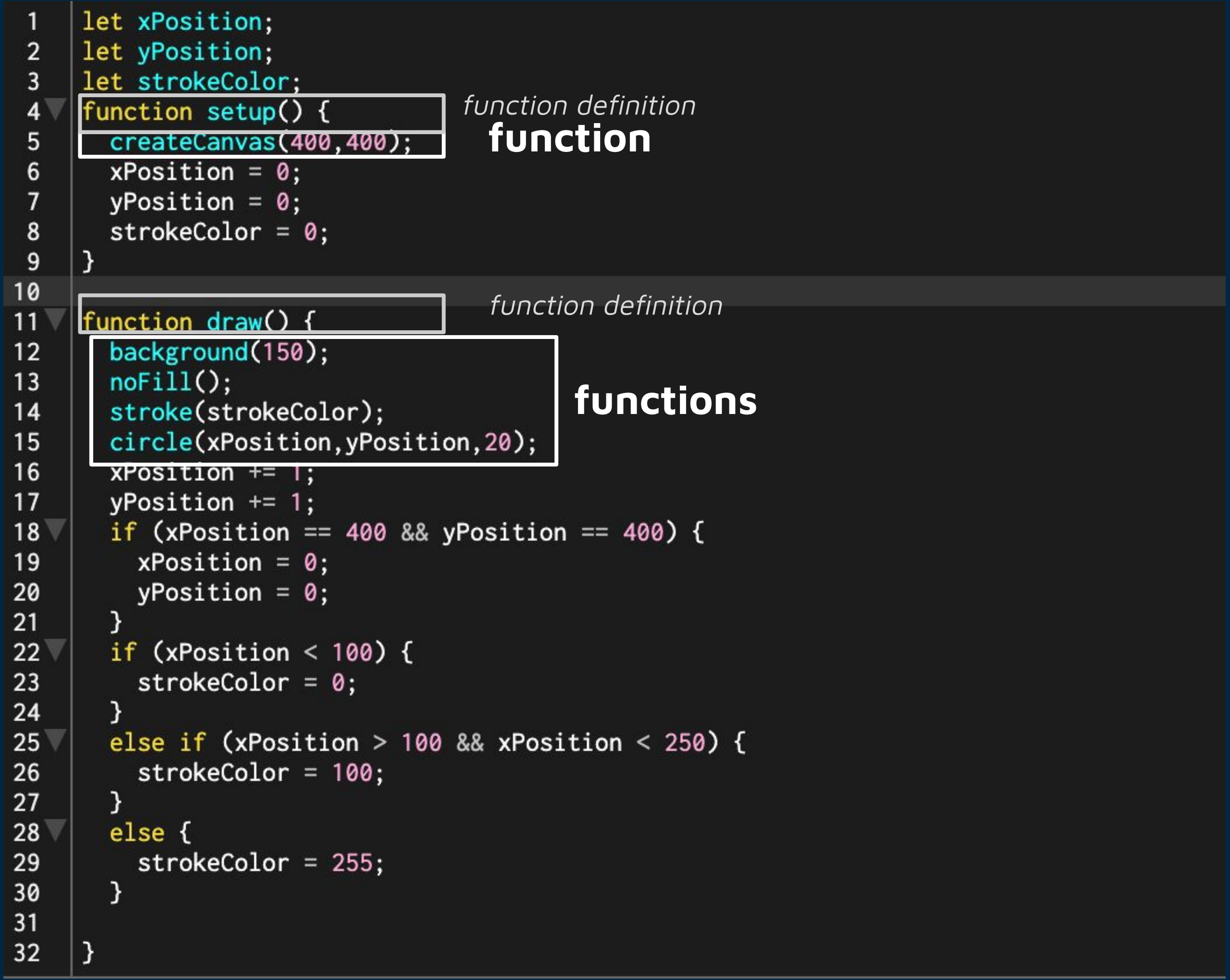
```
  if (xPosition == 400 && yPosition == 400) {
    xPosition = 0;
    yPosition = 0;
  }
  if (xPosition < 100) {
    strokeColor = 0;
  }
  else if (xPosition > 100 && xPosition < 250) {
    strokeColor = 100;
  }
  else {
    strokeColor = 255;
  }

}
```

Operators

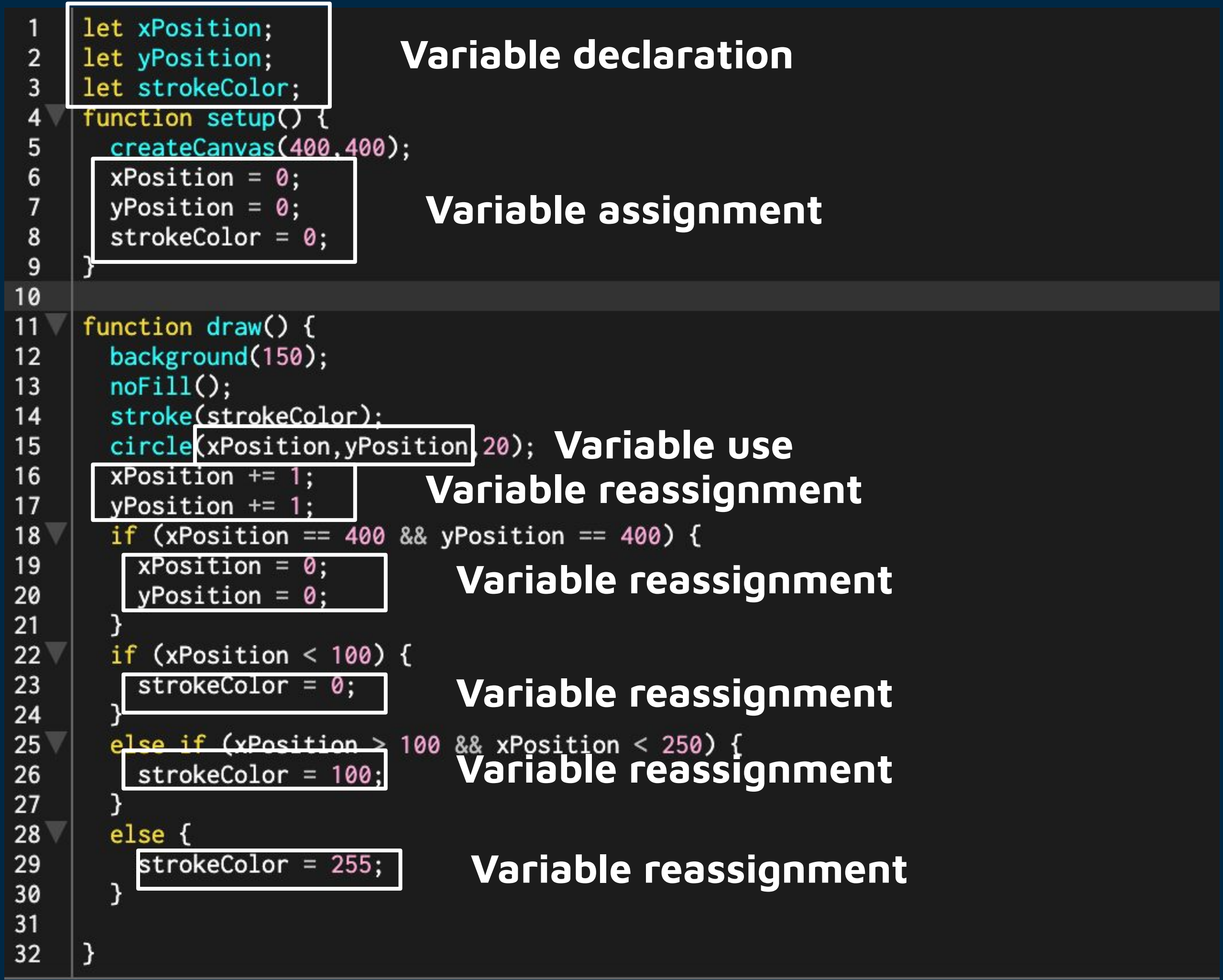
Operators

We can perform operations on variables in order to check for conditions **or** to change (re-assign) a variable.

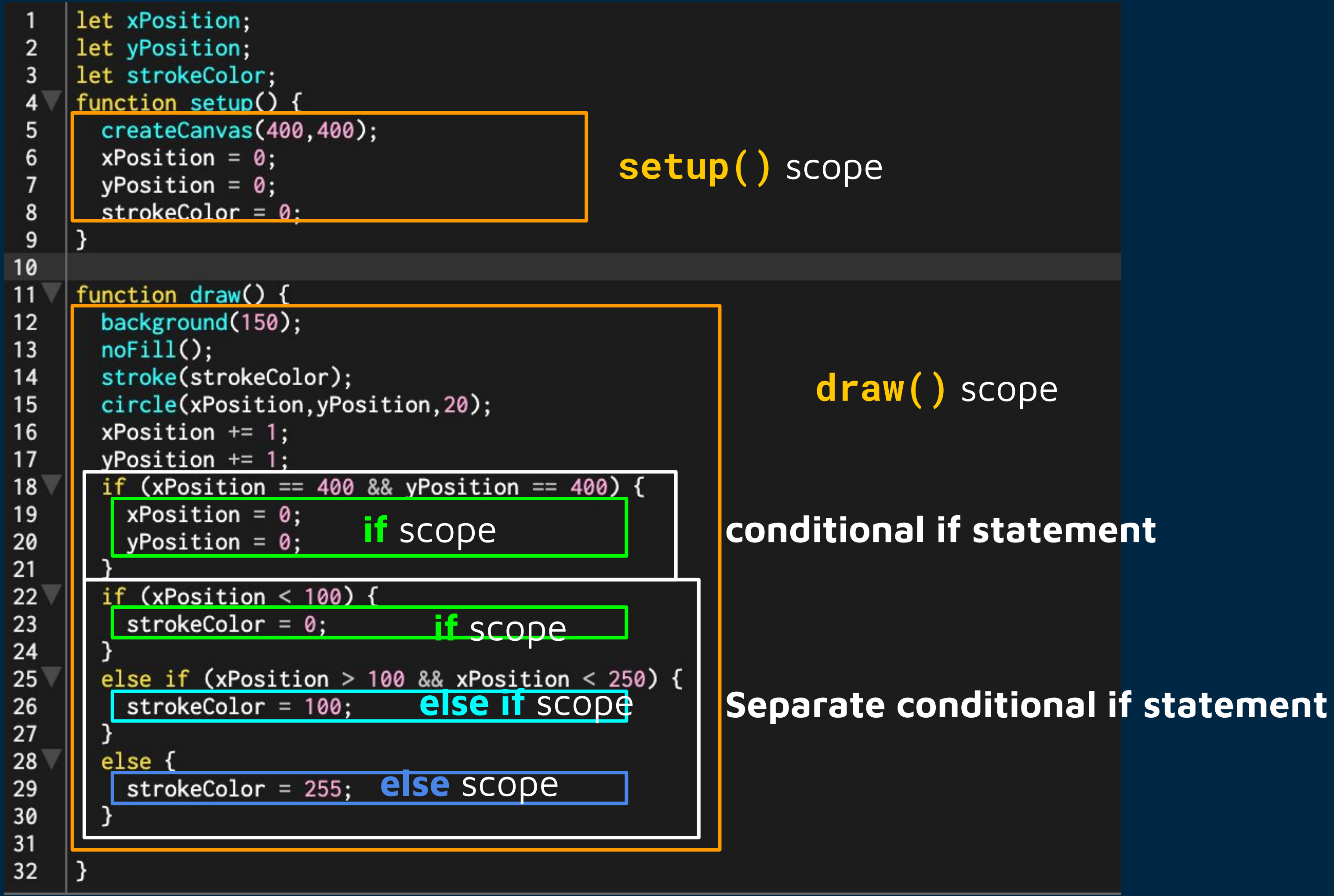
Assign a variable	Reassign a variable	Change a variable	Check a condition	Compare conditions
=	**=**	**+**	**>**	**&&** **(and)**
	+=	**-**	**<**	**||** **(or)**
	-=	***** **(multiply)**	**==**	**!** **(not)**
		/ **(divide)**		



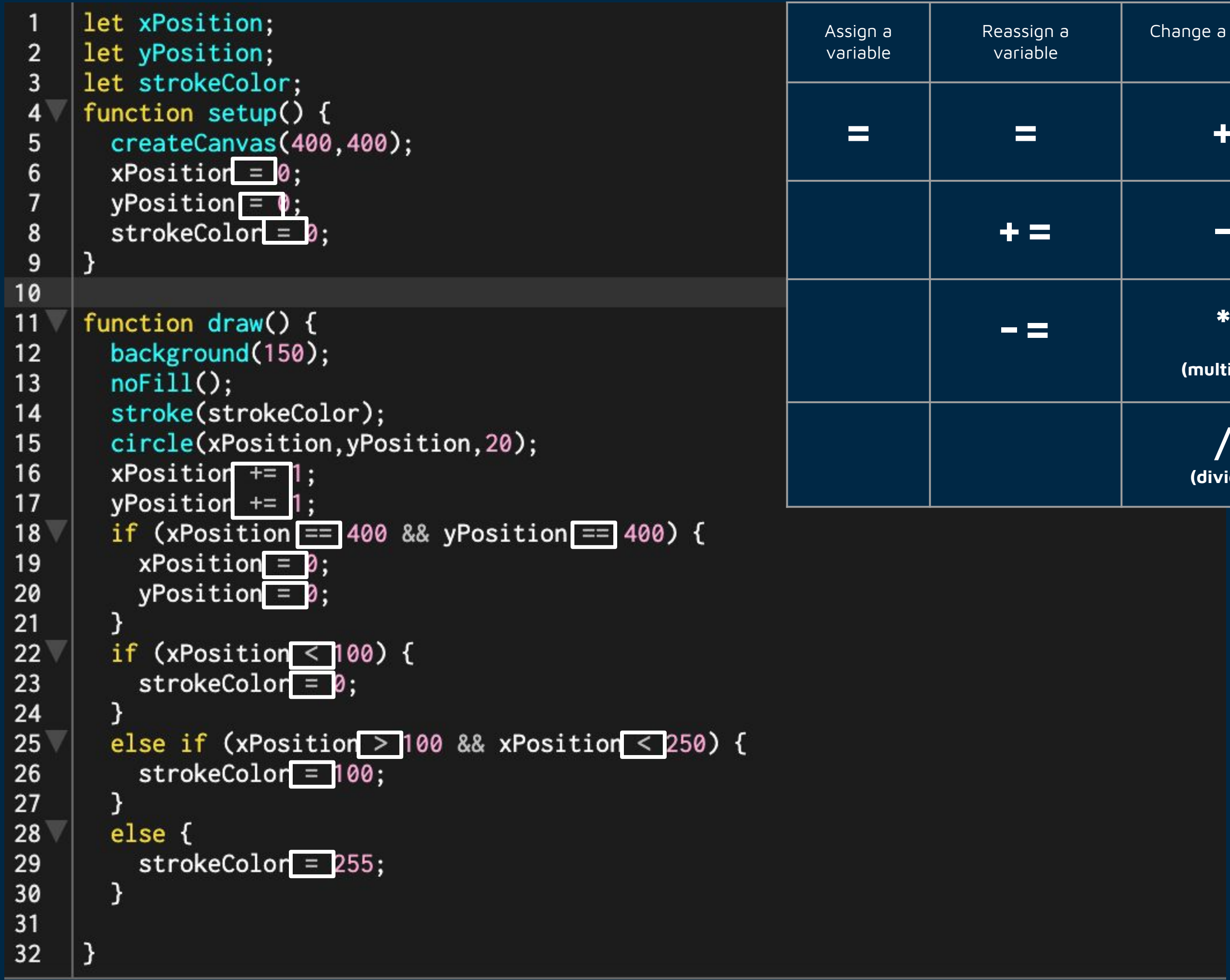
```
let xPosition;
let yPosition;
let strokeColor;
function setup() {
  createCanvas(400,400);
  xPosition = 0;
  yPosition = 0;
  strokeColor = 0;
}

function draw() {
  background(150);
  noFill();
  stroke(strokeColor);
  circle(xPosition,yPosition,20);
  xPosition += 1;
  yPosition += 1;
  if (xPosition == 400 && yPosition == 400) {
    xPosition = 0;
    yPosition = 0;
  }
  if (xPosition < 100) {
    strokeColor = 0;
  }
  else if (xPosition > 100 && xPosition < 250) {
    strokeColor = 100;
  }
  else {
    strokeColor = 255;
  }

}
```



```
let xPosition;
let yPosition;
let strokeColor;
function setup() {
  createCanvas(400,400);
  xPosition = 0;
  yPosition = 0;
  strokeColor = 0;
}

function draw() {
  background(150);
  noFill();
  stroke(strokeColor);
  circle(xPosition,yPosition,20);
  xPosition += 1;
  yPosition += 1;
  if (xPosition == 400 && yPosition == 400) {
    xPosition = 0;
    yPosition = 0;
  }
  if (xPosition < 100) {
    strokeColor = 0;
  }
  else if (xPosition > 100 && xPosition < 250) {
    strokeColor = 100;
  }
  else {
    strokeColor = 255;
  }

}
```



```
let xPosition;
let yPosition;
let strokeColor;
function setup() {
  createCanvas(400,400);
  xPosition = 0;
  yPosition = 0;
  strokeColor = 0;
}

function draw() {
  background(150);
  noFill();
  stroke(strokeColor);
  circle(xPosition,yPosition,20);
  xPosition += 1;
  yPosition += 1;
  if (xPosition == 400 && yPosition == 400) {
    xPosition = 0;
    yPosition = 0;
  }
  if (xPosition < 100) {
    strokeColor = 0;
  }
  else if (xPosition > 100 && xPosition < 250) {
    strokeColor = 100;
  }
  else {
    strokeColor = 255;
  }

}
```



```
let xPosition;
let yPosition;
let strokeColor;
function setup() {
  createCanvas(400,400);
  xPosition = 0;
  yPosition = 0;
  strokeColor = 0;
}

function draw() {
  background(150);
  noFill();
  stroke(strokeColor);
  circle(xPosition,yPosition,20);
  xPosition += 1;
  yPosition += 1;
  if (xPosition == 400 && yPosition == 400) {
    xPosition = 0;
    yPosition = 0;
  }
  if (xPosition < 100) {
    strokeColor = 0;
  }
  else if (xPosition > 100 && xPosition < 250) {
    strokeColor = 100;
  }
  else {
    strokeColor = 255;
  }

}
```

Assign a variable	Reassign a variable	Change a variable	Check a condition	Compare conditions
=	**=**	**+**	**>**	**&&** **(and)**
	+=	**-**	**<**	**||** **(or)**
	-=	***** **(multiply)**	**==**	**!** **(not)**
		/ **(divide)**		

Comments

Comments

Comments allow us to take notes throughout our code.

They are **not** necessary for our code to run. In fact, a comment makes that text **not work** (whether a note **or a piece of code**)

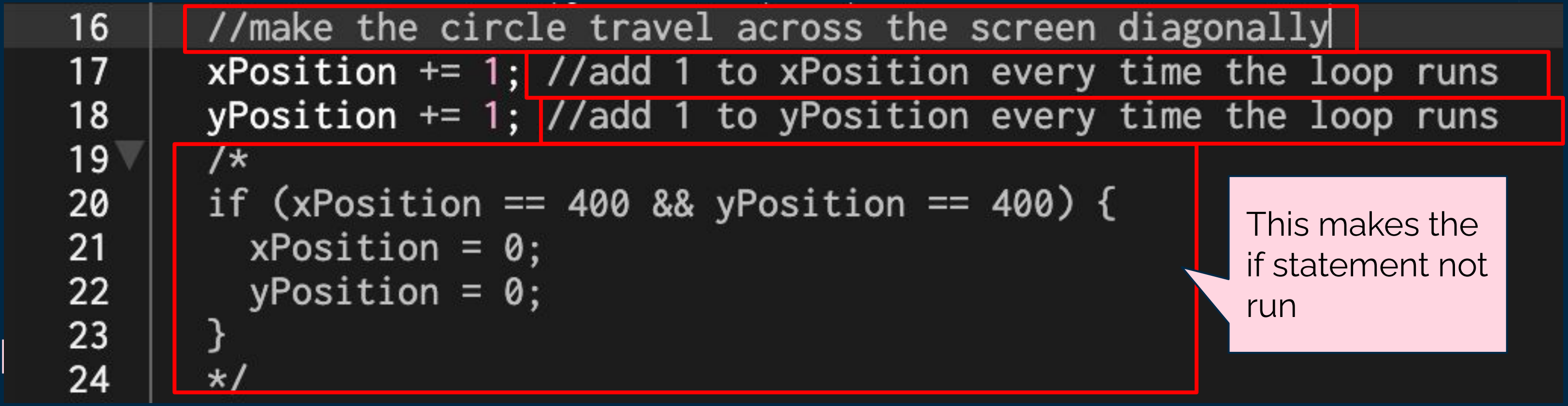
We can use them to help us to do the following things:

- Help us remember why we used the code we did
- Help a new user understand the code and the program
- Help us break down problems as we’re writing new code
- Comment out pieces of code to test different methods

Comments in JS

We write them with **//** for a one line comment

We can also write them with **/*** and ***/** to comment out multiple lines of code



```
//make the circle travel across the screen diagonally
xPosition += 1; //add 1 to xPosition every time the loop runs
yPosition += 1; //add 1 to yPosition every time the loop runs
/*
if (xPosition == 400 && yPosition == 400) {
  xPosition = 0;
  yPosition = 0;
}
*/
```

This makes the if statement not run

Documentation

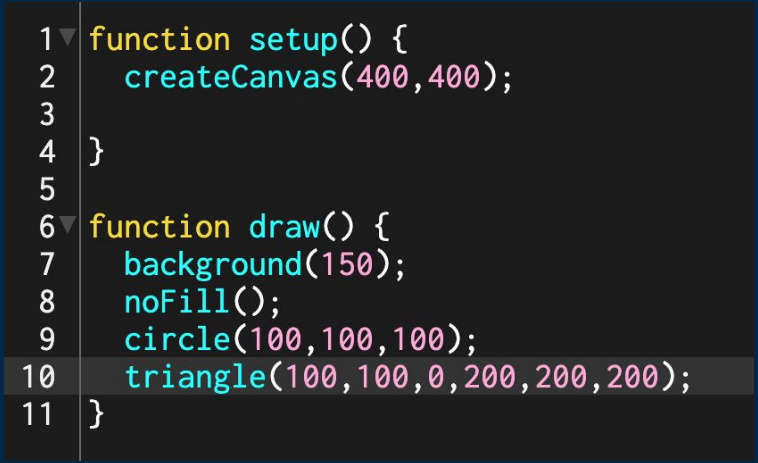
Documentation - reference

Some things can be figured out by trial and error.

It is easy to figure out what the arguments for **`circle()`** do.

It is not so clear to figure out what the arguments for **`triangle()`** do.

This is an example of when we would need to look up the documentation, or reference page.



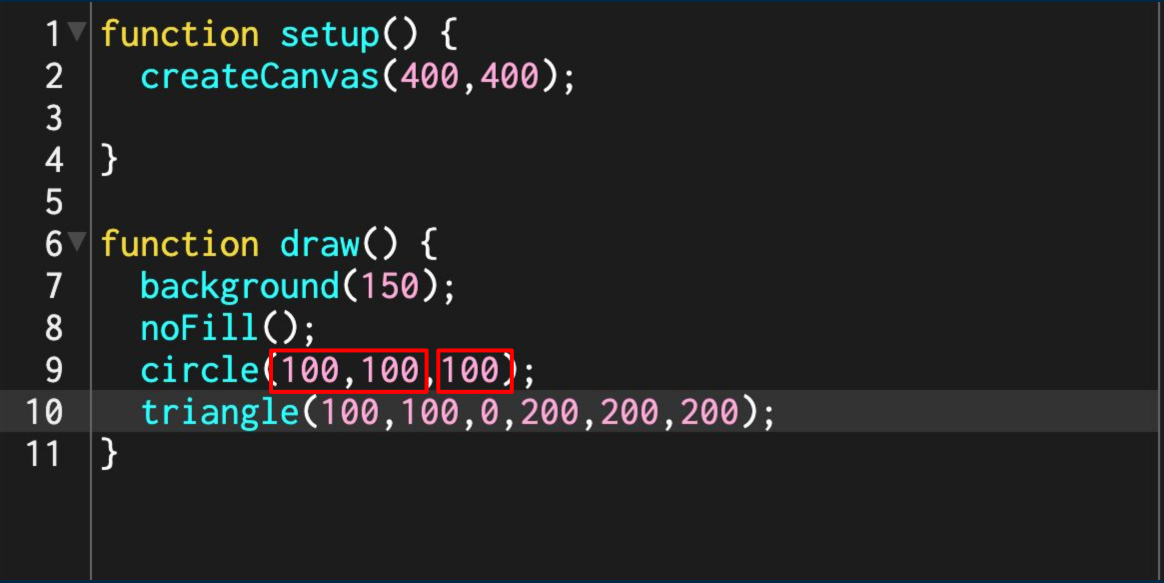
```
function setup() {
  createCanvas(400,400);

}

function draw() {
  background(150);
  noFill();
  circle(100,100,100);
  triangle(100,100,0,200,200,200);
}
```

P5JS has a specific reference page for its library of code :

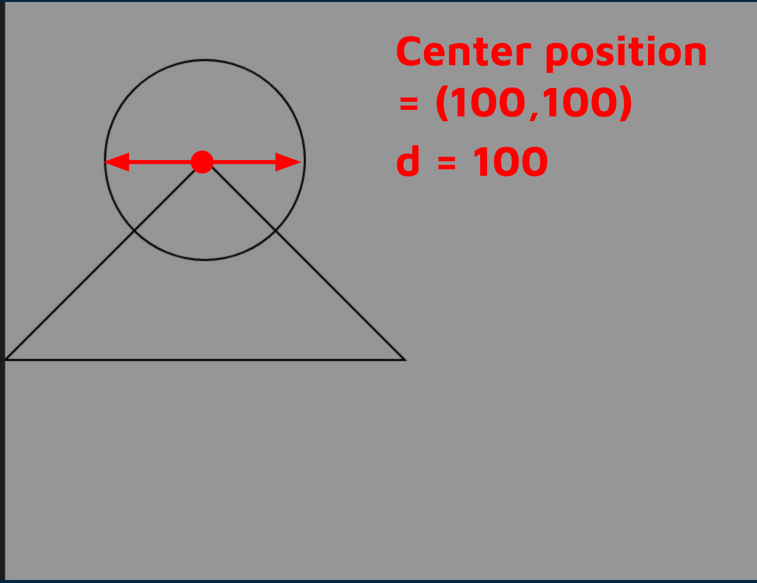
p5js.org/reference

Documentation - reference

By trying things out, we can figure out how many arguments **circle()** needs, and what they do.

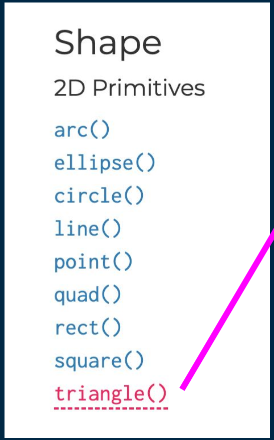


```
function setup() {
  createCanvas(400,400);

}

function draw() {
  background(150);
  noFill();
  circle(100,100,100);
  triangle(100,100,0,200,200,200);
}
```



Documentation - reference

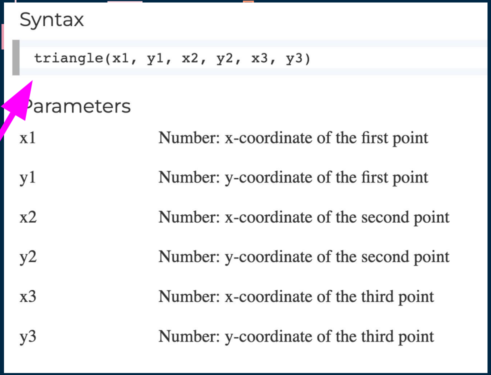
Triangle is less obvious, but on **p5js.org/reference** we can figure out how many arguments **triangle()** needs, and what they do.



Shape

2D Primitives

- arc()
- ellipse()
- circle()
- line()
- point()
- quad()
- rect()
- square()
- triangle()

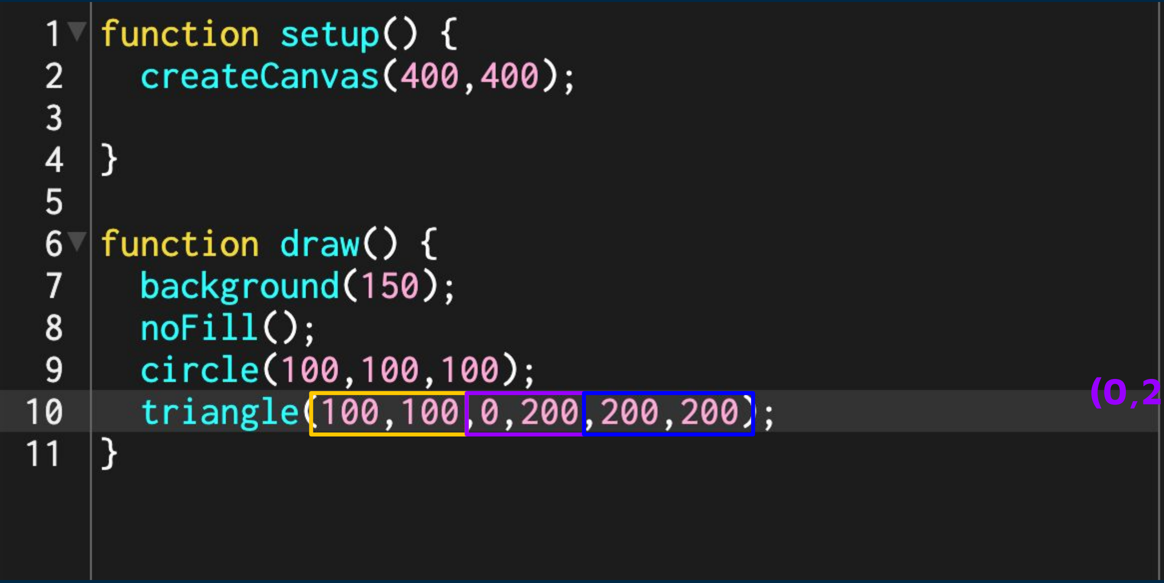


Syntax

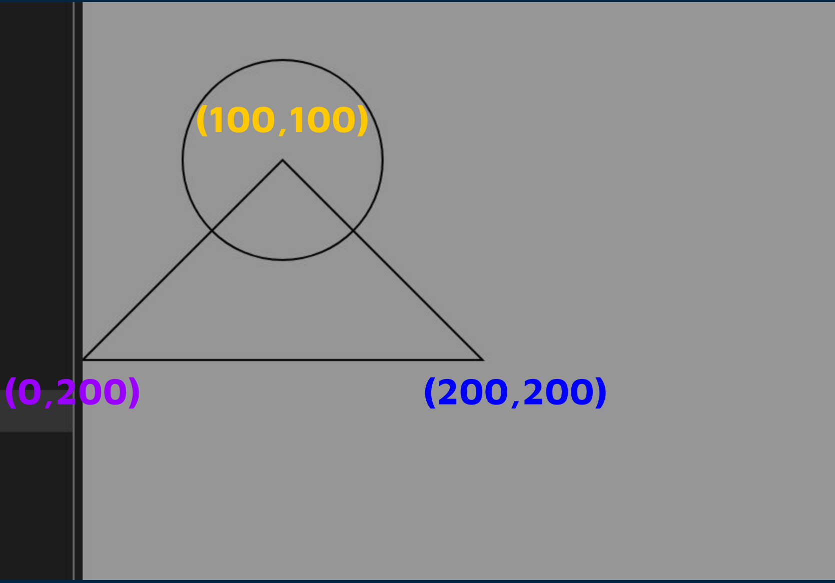
```
triangle(x1, y1, x2, y2, x3, y3)
```

Parameters

x1	Number: x-coordinate of the first point
y1	Number: y-coordinate of the first point
x2	Number: x-coordinate of the second point
y2	Number: y-coordinate of the second point
x3	Number: x-coordinate of the third point
y3	Number: y-coordinate of the third point



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

}

function draw() {
  background(150);
  noFill();
  circle(100,100,100);
  triangle(100,100,0,200,200,200);
}
```

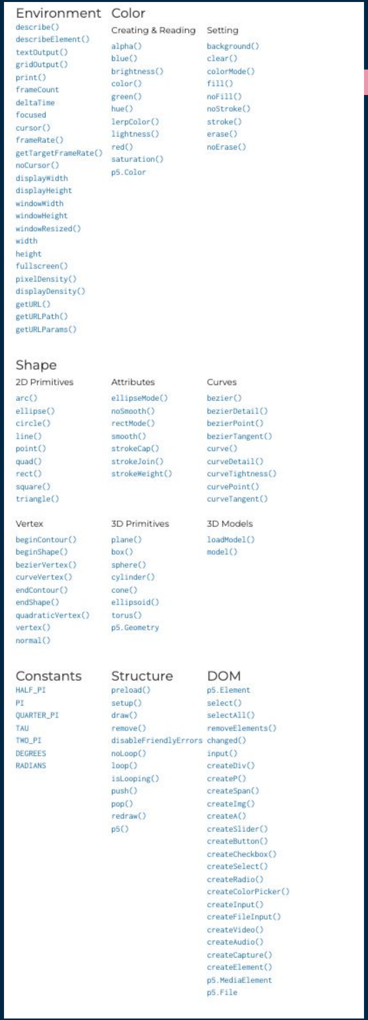


Documentation – reference

All programmers use documentation regularly. **Every language has documentation somewhere online**.

We can't know every function that exists, or how every variable works. So we have to look it up to understand how to use it.

On the P5JS reference page, there are lots of specialized functions, variables, and elements, and by clicking on them, we can see what they do.



Environment
describe()
describeElement()
textOutput()
gridOutput()
print()
frameCount
deltaTime
focused
cursor()
frameRate()
getTargetFrameRate()
noCursor()
displayWidth
displayHeight
windowWidth
windowHeight
windowResized()
width
height
fullscreen()
pixelDensity()
displayDensity()
getURL()
getURLPath()
getURLParams()

Color

Creating & Reading
alpha()
blue()
brightness()
color()
green()
hue()
lerpColor()
lightness()
red()
saturation()
p5.Color

Setting
background()
clear()
colorMode()
fill()
noFill()
noStroke()
stroke()
erase()
noErase()

Shape

2D Primitives
arc()
ellipse()
circle()
line()
point()
quad()
rect()
square()
triangle()

Attributes
ellipseMode()
noSmooth()
rectMode()
smooth()
strokeCap()
strokeJoin()
strokeWeight()

Curves
bezier()
bezierDetail()
bezierPoint()
bezierTangent()
curve()
curveDetail()
curveTightness()
curvePoint()
curveTangent()

Vertex
beginContour()
beginShape()
bezierVertex()
curveVertex()
endContour()
endShape()
quadraticVertex()
vertex()
normal()

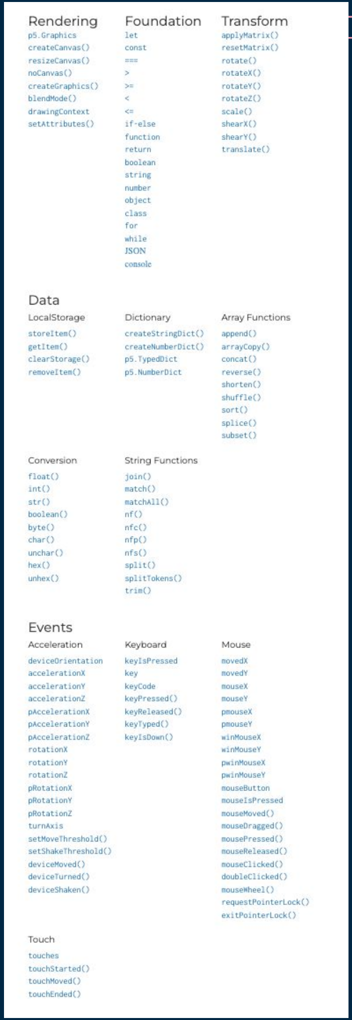
3D Primitives
plane()
box()
sphere()
cylinder()
cone()
ellipsoid()
torus()
p5.Geometry

3D Models
loadModel()
model()

Constants
HALF_PI
PI
QUARTER_PI
TAU
TWO_PI
DEGREES
RADIANS

Structure
preload()
setup()
draw()
remove()
disableFriendlyErrors
noLoop()
loop()
isLooping()
push()
pop()
redraw()
p5()

DOM
p5.Element
select()
selectAll()
removeElements()
changed()
input()
createDiv()
createP()
createSpan()
createImg()
createA()
createSlider()
createButton()
createCheckbox()
createSelect()
createRadio()
createColorPicker()
createInput()
createFileInput()
createVideo()
createAudio()
createCapture()
createElement()
p5.MediaElement
p5.File



Rendering
p5.Graphics
createCanvas()
resizeCanvas()
noCanvas()
createGraphics()
blendMode()
drawingContext
setAttributes()

Foundation
let
const
===
>
>=
<
<=
if-else
function
return
boolean
string
number
object
class
for
while
JSON
console

Transform
applyMatrix()
resetMatrix()
rotate()
rotateX()
rotateY()
rotateZ()
scale()
shearX()
shearY()
translate()

Data

LocalStorage
storeItem()
getItem()
clearStorage()
removeItem()

Dictionary
createStringDict()
createNumberDict()
p5.TypedDict
p5.NumberDict

Array Functions
append()
arrayCopy()
concat()
reverse()
shorten()
shuffle()
sort()
splice()
subset()

Conversion
float()
int()
str()
boolean()
byte()
char()
unchar()
hex()
unhex()

String Functions
join()
match()
matchAll()
nf()
nfc()
nfp()
nfs()
split()
splitTokens()
trim()

Events

Acceleration
deviceOrientation
accelerationX
accelerationY
accelerationZ
pAccelerationX
pAccelerationY
pAccelerationZ
rotationX
rotationY
rotationZ
pRotationX
pRotationY
pRotationZ
turnAxis
setMoveThreshold()
setShakeThreshold()
deviceMoved()
deviceTurned()
deviceShaken()

Keyboard
keyIsPressed
key
keyCode
keyPressed()
keyReleased()
keyTyped()
keyIsDown()

Mouse
movedX
movedY
mouseX
mouseY
pmouseX
pmouseY
winMouseX
winMouseY
pwinMouseX
pwinMouseY
mouseButton
mouseIsPressed
mouseMoved()
mouseDragged()
mousePressed()
mouseReleased()
mouseClicked()
doubleClicked()
mouseWheel()
requestPointerLock()
exitPointerLock()

Touch
touches
touchStarted()
touchMoved()
touchEnded()