Creative Coding For Kids

A Structured Course of Short Projects For Young and New Coders

by Tariq Rashid

[codeguppy.com edition]

Contents

Introduction

A Course For Young And New Coders

This course was developed in response to demand from teachers and parents for a child-friendly course that:

- engages students visually
- avoids complicated technology setup
- teaches programming and computer science concepts

They found that many teaching resources were not designed with young or non-technical students in mind. Long passages of prose, technical jargon, over-complicated examples and unexplained code, put up barriers that many students couldn’t get over.

This course is specifically designed to be accessible to young learners, with language carefully chosen to maximize understanding. The course projects are kept short, with lots of illustrations, and planned breakpoints for exploring and playing.

As students progress through the course, they will learn about, and gain experience with, programming and computer science concepts that are applicable to many other programming languages they might learn in future.

Why Learn To Code?

Coding and algorithmic thinking are important life skills in the increasingly digital world we live in.

Many education curricula have been updated ensure that children are digitally literate, equipped to participate in a digital economy, able to develop their own technology ideas, and be better informed consumers and citizens.

Coding is considered by many to be as essential as reading and writing - **Reading**, **Writing**, **Coding**.

Coding is also fun and creative! Many people do it just for pleasure. A growing number of artists now use code as their main method for creating art.

Who Is This Course For?

- **Students as young as 7** can make a start on the first few projects as they are simple, easy, and don't require a lot of background knowledge or extended periods of concentration.

- **Secondary school students aged 11-17** will enjoy the more interesting concepts developed during the course, including seeing how the maths and physics they have learned can be applied creatively.

- **Talented and more confident students** will enjoy the later projects which introduce more sophisticated, and rewarding, concepts. Effort has been made to ensure these too are as accessible as possible.

- **Home educated students** will find this course useful as it introduces a broad range of key programming and computer science concepts in a fun, visual, and creative way that also encourages students to experiment and develop autonomous research and problem solving skills.

- **Adults learning to code for the first time** will also find this course a gentle and friendly introduction to coding that avoids unnecessary jargon and technical complexity.

Tested & Refined

This course is has been tested and refined with feedback from children aged 6-11 in code clubs, students aged 11-17 at secondary school or in home education, university undergraduates, and also adult members of creative coding and algorithmic art groups.

I’m really pleased with the artists who had little previous experience, yet very quickly become confident and creative with code.

In particular, I’m proud of the children who tried this course and have now found a talent or a passion they didn’t know they had.

Teaching Method - Learning By Doing

This course consists of a progression of short projects. Each project builds on the knowledge and experience developed in previous projects.

Each project starts by introducing a new idea, and quickly gets students coding. At regular points in each project students are encouraged to experiment with the ideas being developed.

This hands-on practice, experimentation and play, is critical to learning. That also includes learning from getting things wrong, something we need to encourage far more in the teaching of computer science and coding.

Simply reading this book without taking part in the suggested practice will not be as effective for learning to code, developing algorithmic thinking, and developing a feel for the ideas and coding methods.

Each project ends with a challenge that is calibrated to stretch talented or enthusiastic students by applying the ideas they've just learned in new ways, and practising doing research to find solutions for themselves.

The Technology

Although we don't dwell on it in the course, the programming language that students learn is **JavaScript**, one of the most important languages in today's digital age.

We use **codeguppy.com**, an online coding environment based on **p5js.** Students will code entirely on the web, and see their results on the web. This simplicity avoids the need for complicated software and also avoids students needing to grapple with source code files and editors.

p5js itself was developed from **Processing**. **Processing** and **p5js** are taught to students at schools, universities and design colleges all over the world.

Students can share their creations as easily as sharing a web link. Anyone who has the link can see both their code and the designs their code draws.

This course doesn't focus on the technology, keeping it out of the way as much as possible, to ensure the focus remains on concepts, ideas and creativity.

Get In Touch

I’d love to discuss ideas, challenges or questions raised by this course.

I’d also love to hear about the great art students have created.

You can contact me by email makeyourownalgorithmicart@gmail.com or on twitter @algorithmic_art.

Advanced codeguppy.com users that graduate from codeguppy.com environment to pure p5.js sketches may also find useful author’s YouTube channel:

- https://www.youtube.com/c/algorithmicart

Learning Together

Learning to code and creating digital art is even more fun when you’re part of a supportive community.

I recommend you find a children’s code club or creative coding community near you.

0.0 - Getting Set Up

Start Your Browser

We’re going to be using a website called **codeguppy.com**.

It’s free and makes coding really easy - and fun. There is no need to install any software. All we need is a web browser.

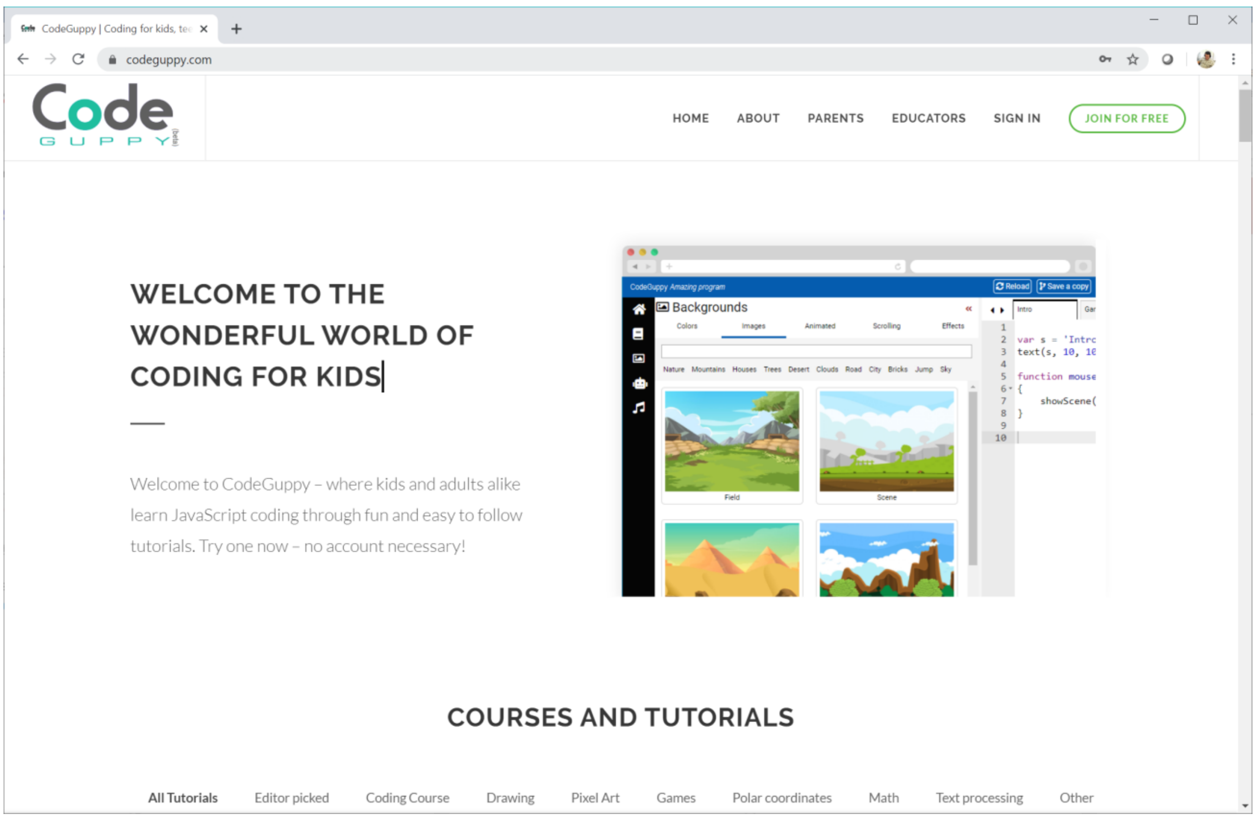
Start your favorite web browser. Yours might be Chrome, Firefox, or Safari.

Sign Up

We need to create our codeguppy.com account.

Go to **www.codeguppy.com** by typing it into the web address bar at the top.

You should see the website looking like this:

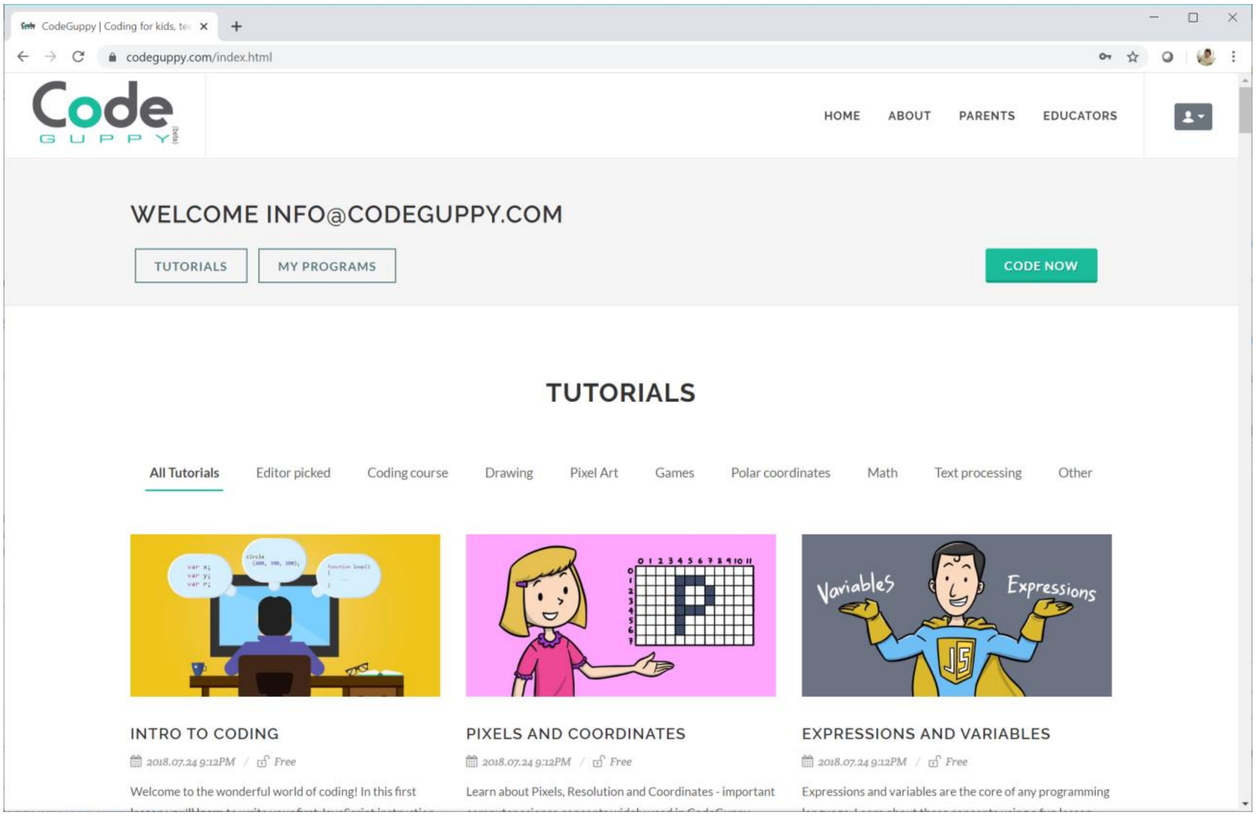


Click **JOIN FOR FREE**.

Choose "Register with email" and enter your email address and a password that you will use to log into codeguppy.com in future.

Make sure you confirm your account by clicking the link that will be sent to you via email.

Every time you log in, you will be taken to a page that looks like this:



You can use the two main buttons “Tutorials” and “My Programs” to switch between displaying the built-in tutorials or your programs. If you don’t have any program created yet, the second list will be empty.

We’re now ready to start our code projects!

Level 1 - First Steps

1.1 - First Shapes

level

What We'll Do

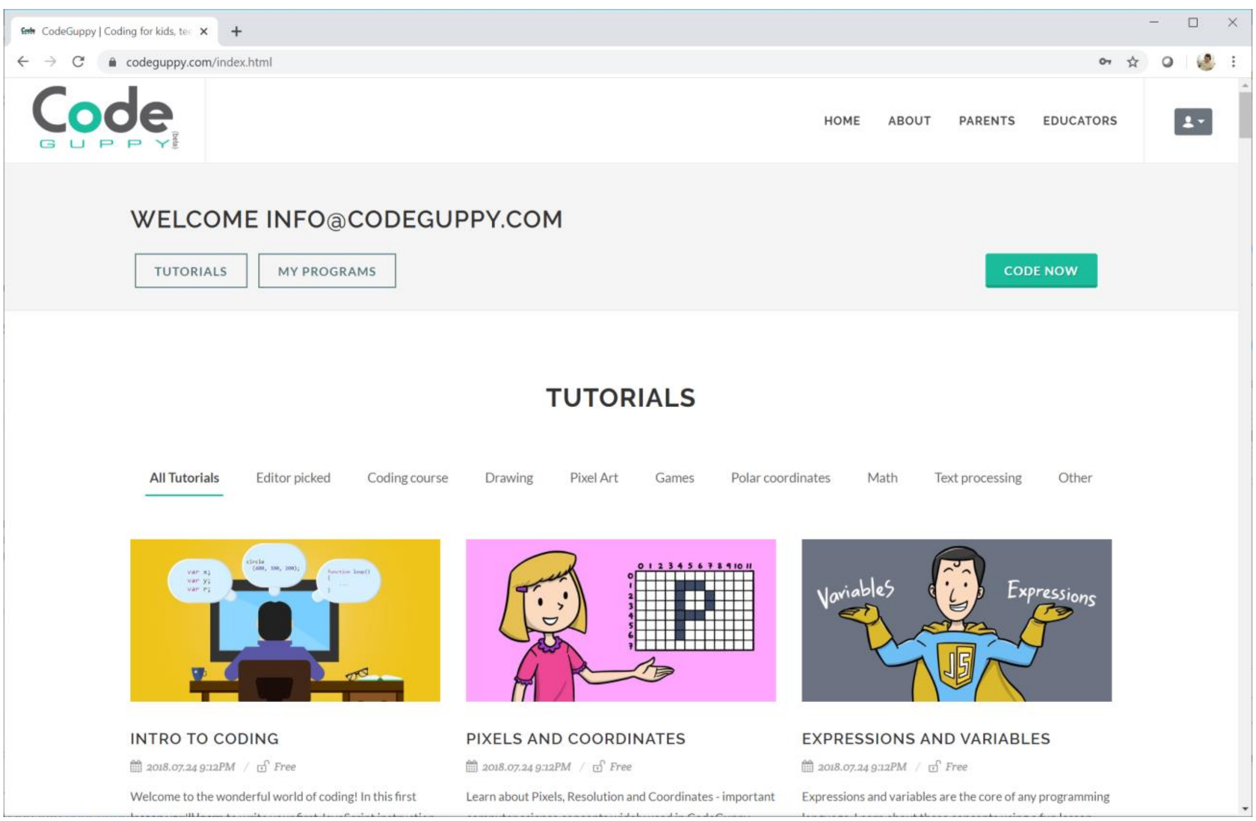
In this project we're going to:

- create **simple shapes** - circles and squares
- use **colours** to fill those shapes

Log In

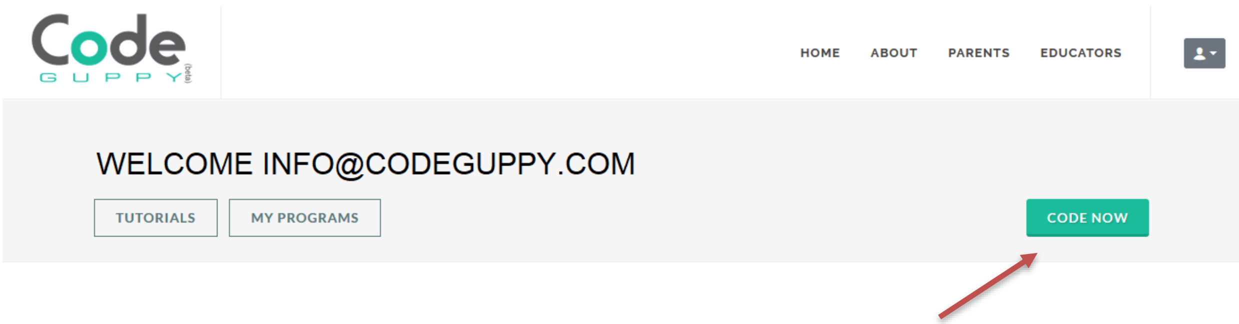
Use your browser to go to codeguppy.com, and log in using the account you created before.

You should see a page like this:

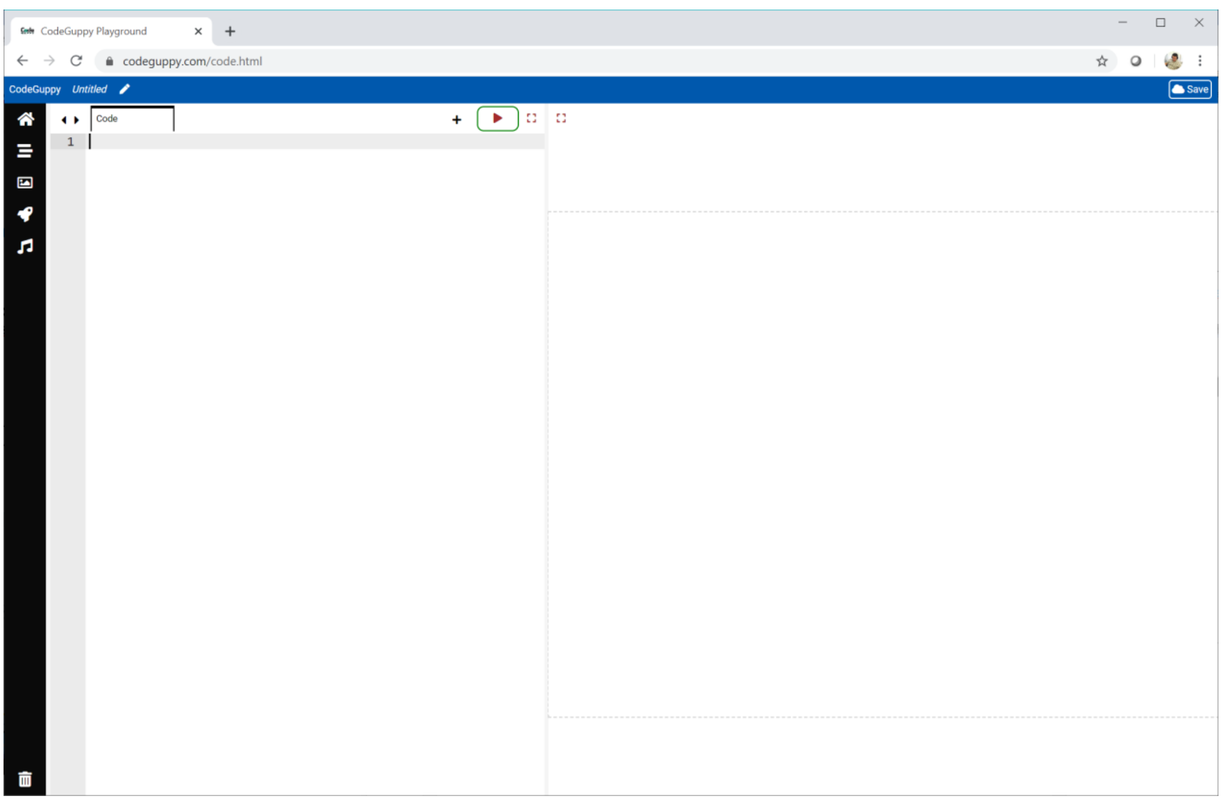


Create a Program

Start a new program by clicking on **Code Now**.



You should see a page that looks like this:



We’re going to write our own code on this page.

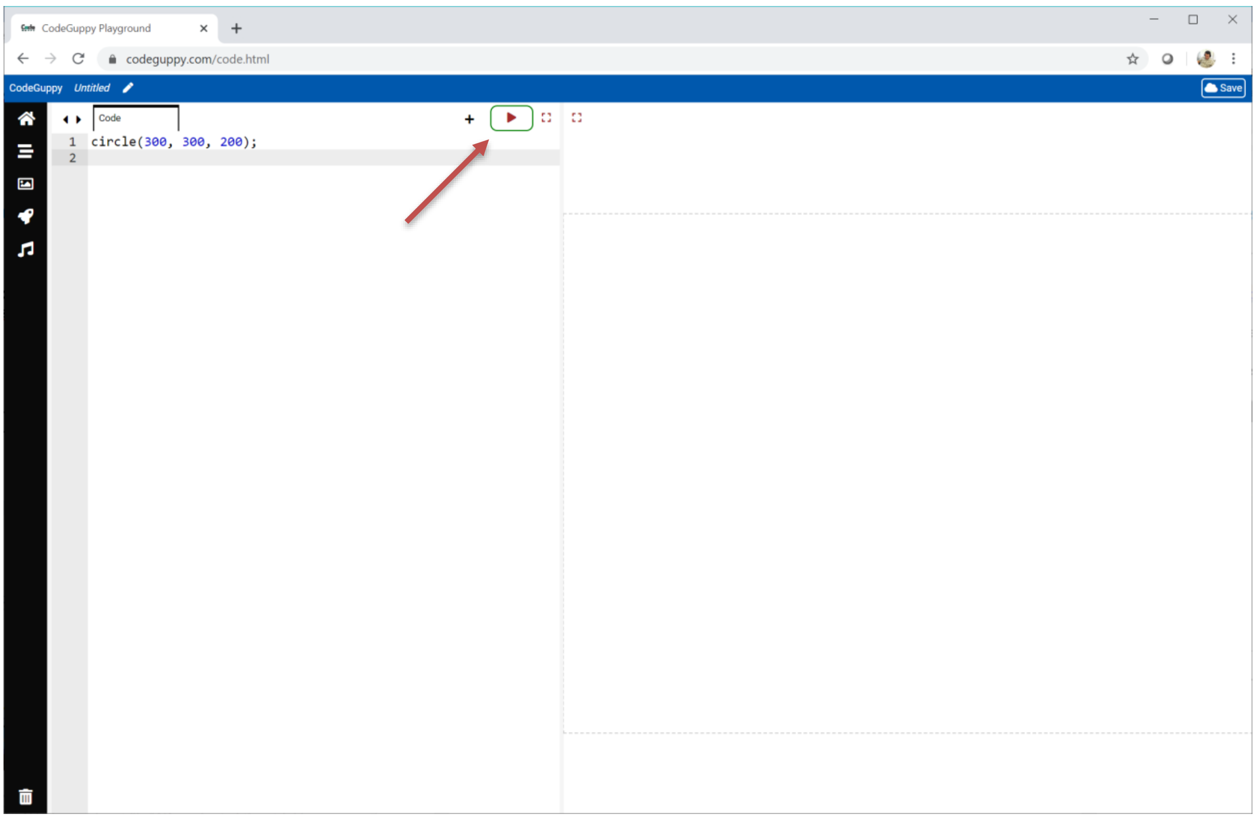
Our First Circle

Let's draw a circle. Write this code in the editor part.

```
circle(300, 300, 200);
```

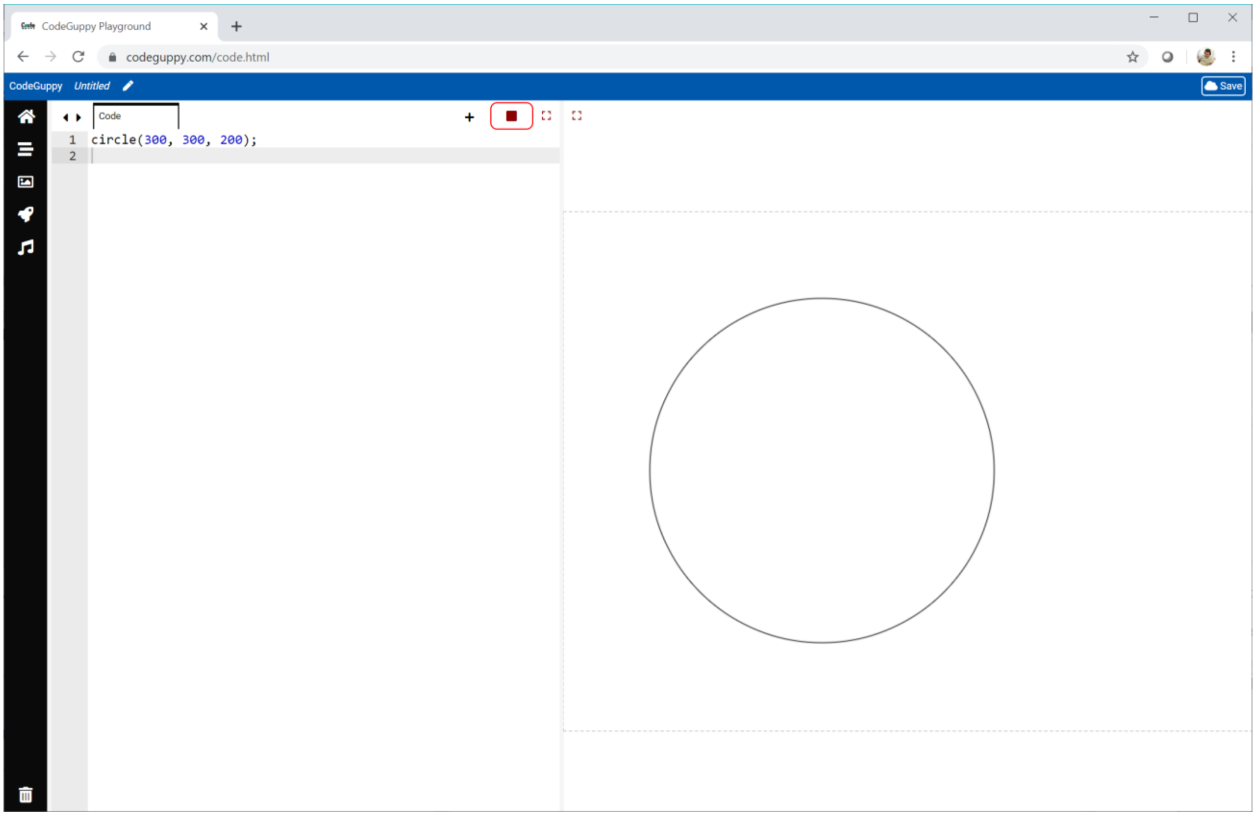
Don't forget the semicolon **;** at the end.

The code should look like this now:



To run our code click the "Play" button at the top of your code window.

You should see something like this (note: if your screen resolution cannot accommodate both the code and the output at the same time, you'll be presented with just one of them at a time):



That's a nice circle. You've just written your first code!

Different Colours

Let’s change the colour of our circle.

To go back to the code click on the square “Stop” button.

You’ll be taken back to the code you’re working on.

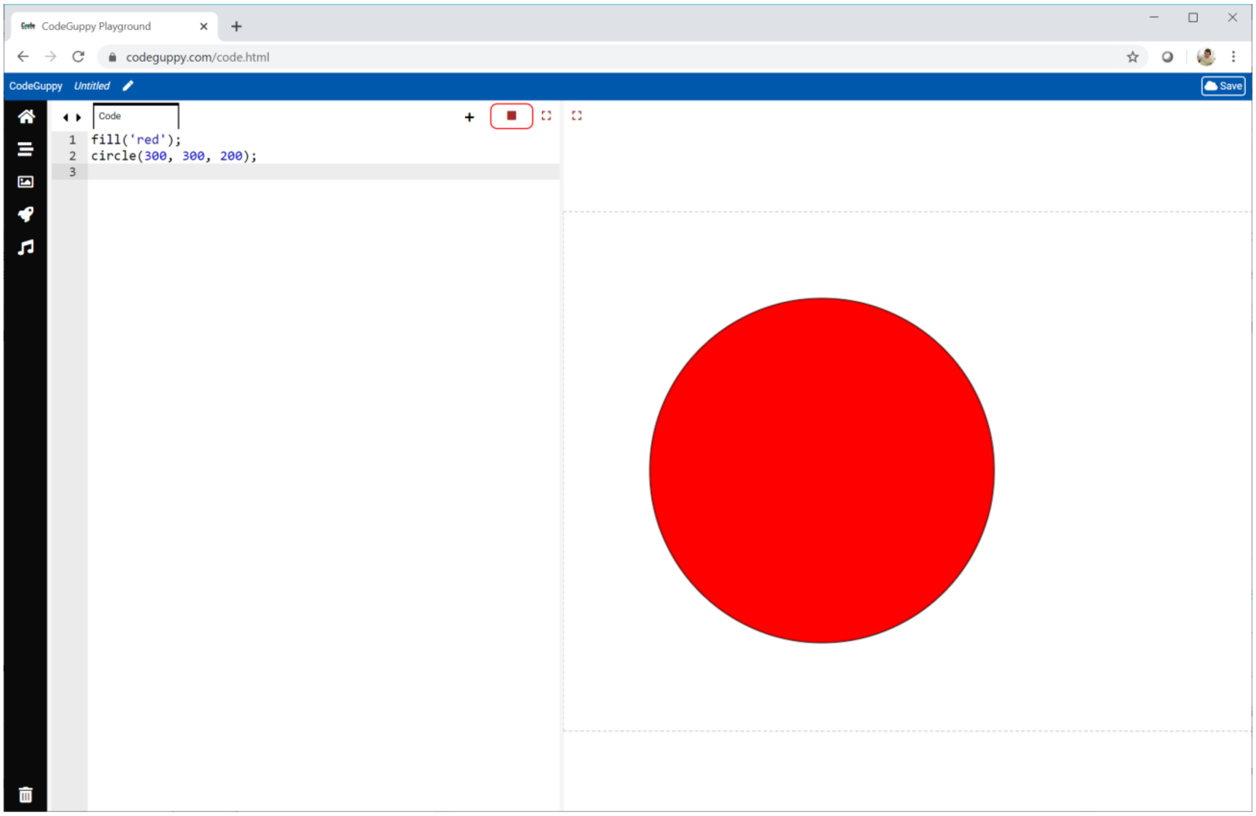
Add the **fill** instruction to the code, just before the **circle** instruction, like this:

```
fill('red');
circle(300, 300, 200);
```

Can you guess what this new code does?

Most code instructions have names that give us a good clue about what they do. The **circle** instruction ... draws a circle. The **fill** instruction … picks a colour to fill a shape.

Run the code to see what happens.
You should see a red circle.



Try It Yourself

Have a go at choosing different colours to fill the circle.
I changed my code to draw a pink circle.

```
fill('pink');
circle(300, 300, 200);
```

You can choose colours like **blue**, **green**, **black**, **white**, **grey**, **violet**, .. and many more.

You can find a full list of colours by opening the “Backgrounds -> Colors” palette from the left toolbar of codeguppy.com coding environment.

Drawing Squares

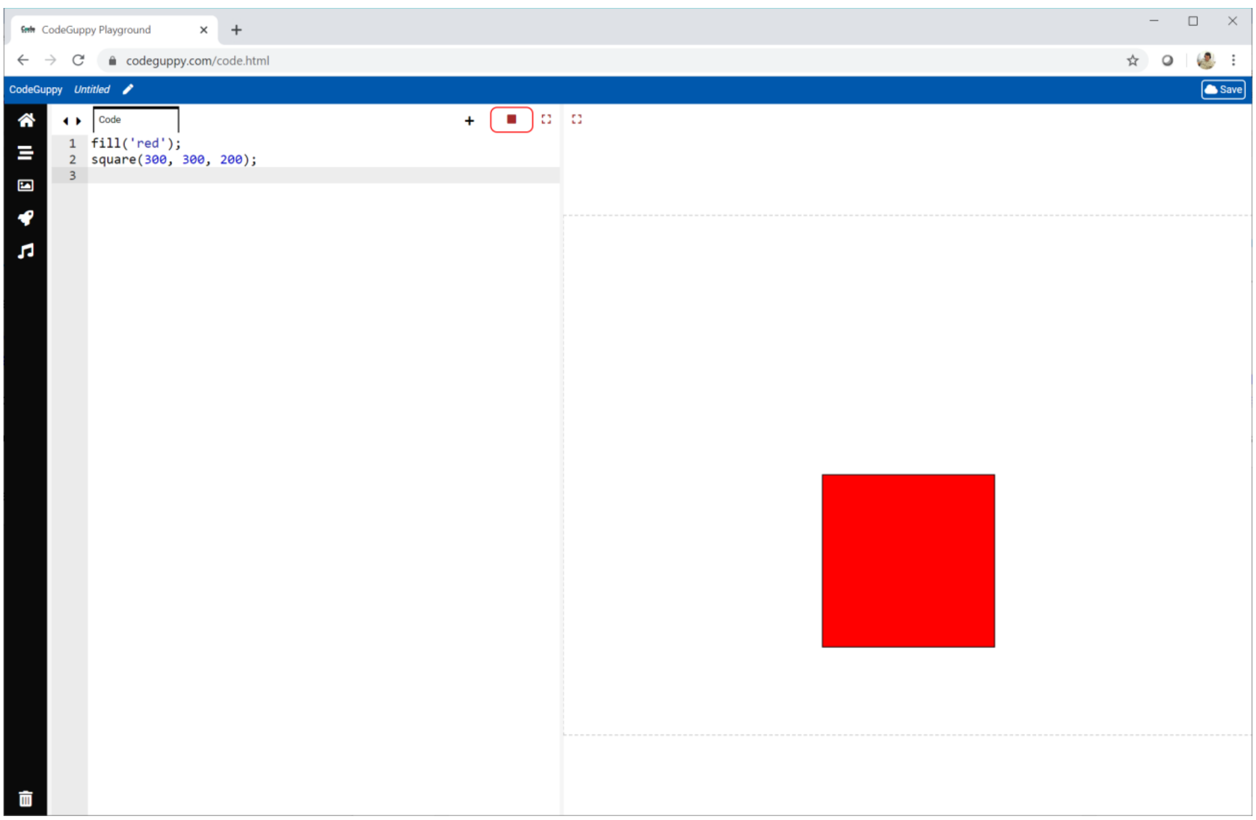
Let’s try drawing a different shape now. Have a look at the code below:

```
fill('red');
square(300, 300, 200);
```

What shape do you think this will draw?

Change your code so that it has that **square** instruction instead of the **circle** instruction.

If we run the code we should get a red square.



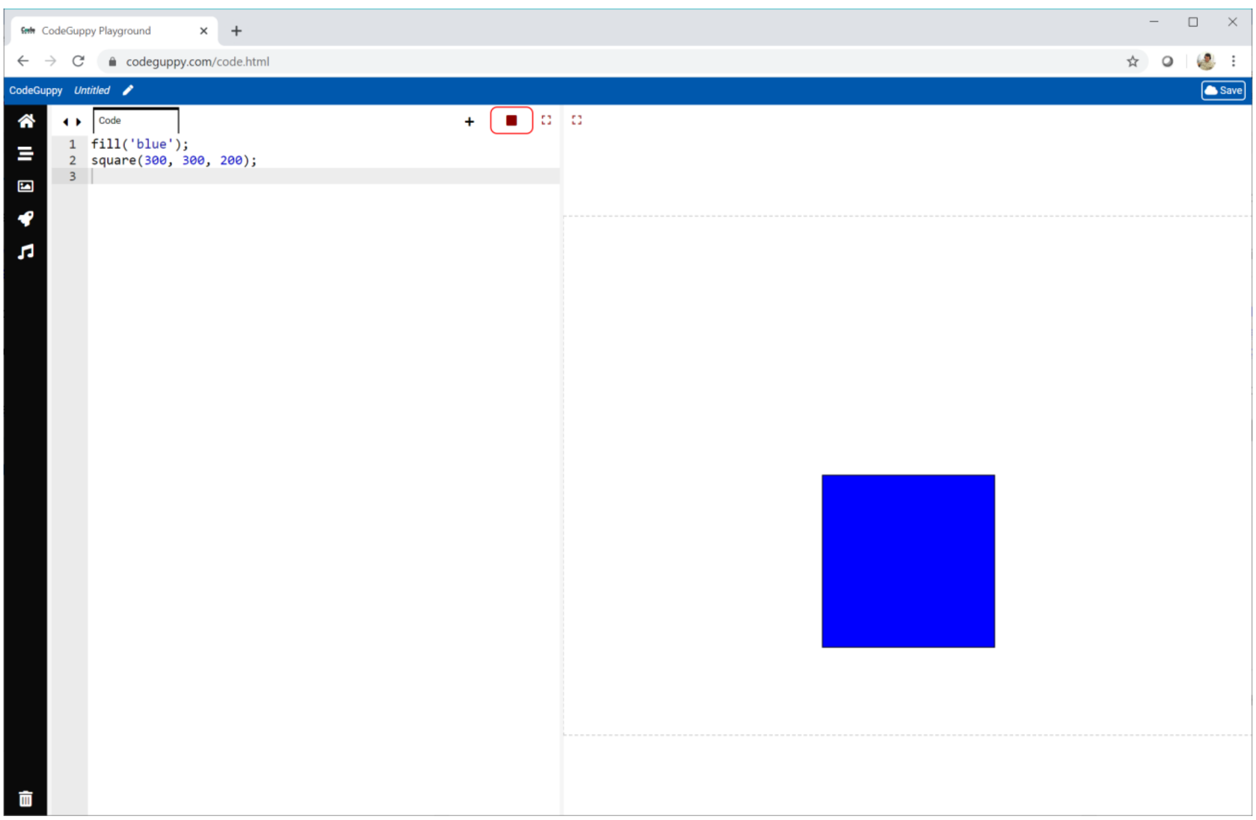
Try It Yourself

We can change the colour of our square just like we changed the colour of our circle.

I changed my code to draw a blue square.

```
fill('blue');
square(300, 300, 200);
```

This is what happens when I run the code:



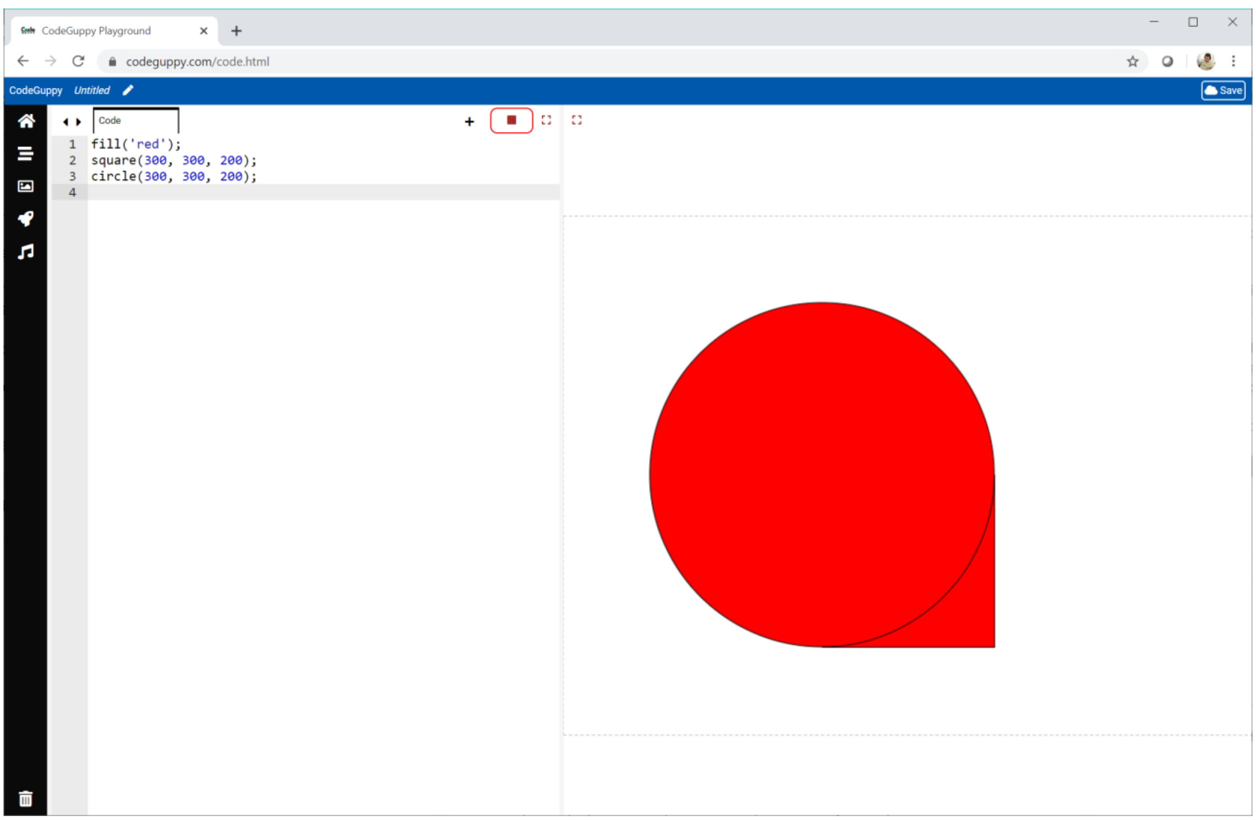
Try changing the colour yourself. Can you make a **green** square?

Square and Circle Together

Let's try something exciting and write the code for a square and a circle together!
Look at this bit of code. You can see we've written instructions for a **square** and a **circle**.

```
fill('red');
square(300, 300, 200);
circle(300, 300, 200);
```

Change your code so it draws both shapes. If we run the code, we get a picture with both shapes!



This shows we can use lots of drawing instructions to make a more interesting picture.

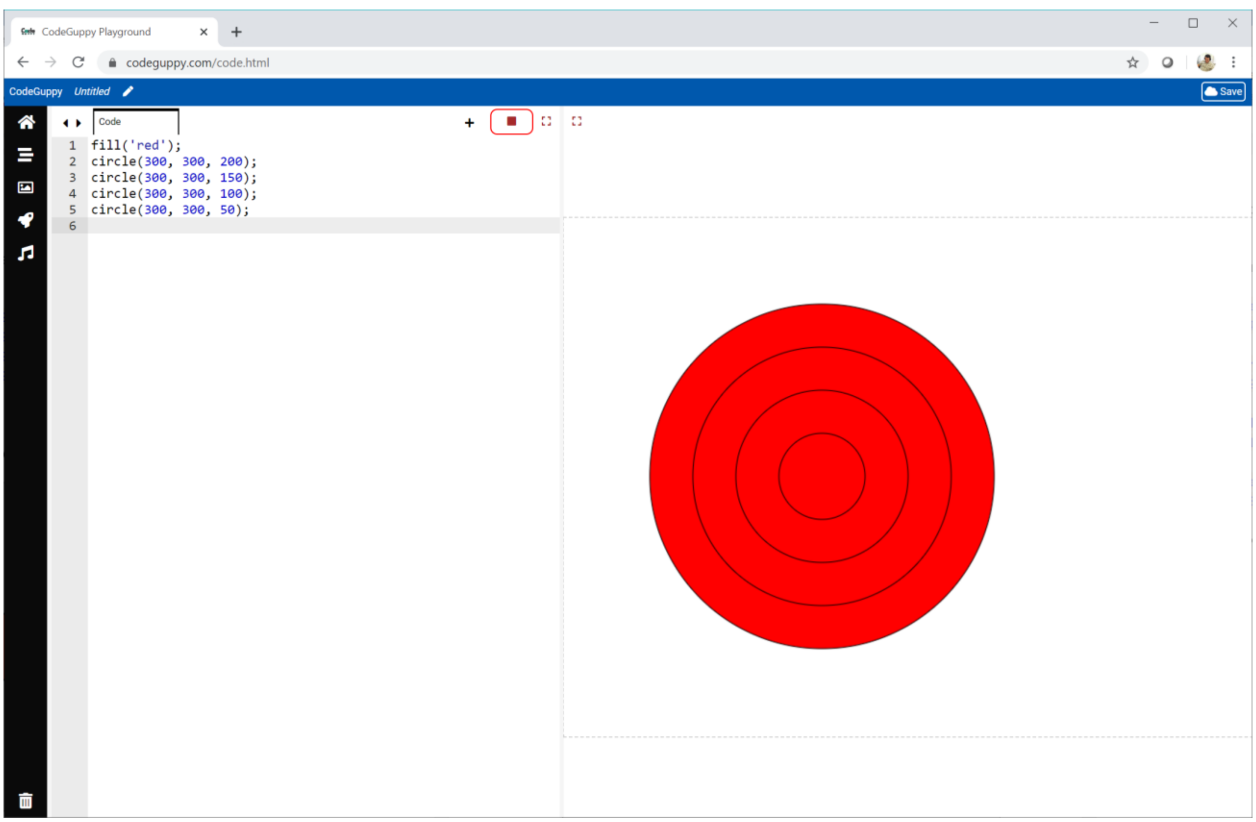
Try It Yourself

Write this code that draws four red circles:

```
fill('red');
circle(300, 300, 200);
circle(300, 300, 150);
circle(300, 300, 100);
circle(300, 300, 50);
```

Did you notice the last numbers in the brackets are not the same?

This is what happens when we run the code:



Can you work out what those last numbers do?

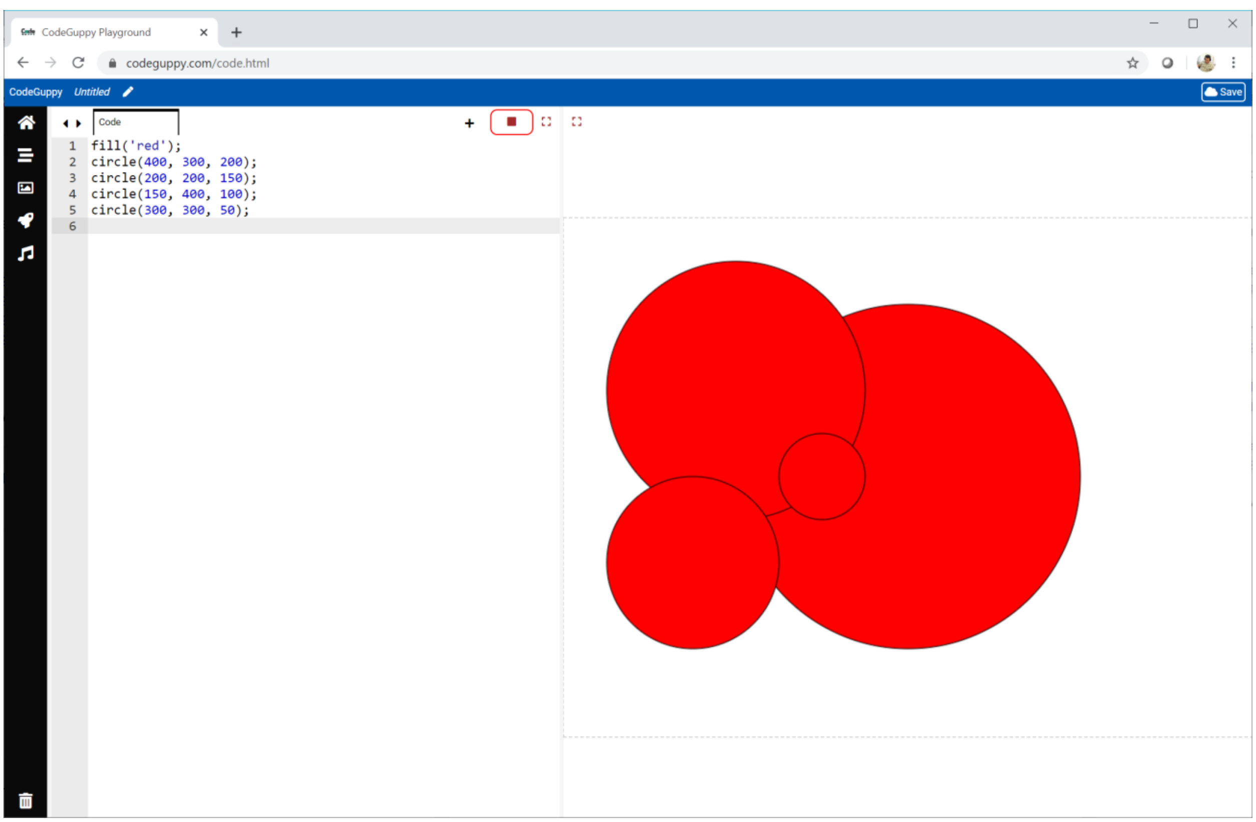
Try changing those numbers to different ones. Run your code to see what happens.

Now try changing **all** the numbers, like this example:

```
fill('red');
circle(400, 300, 200);
circle(200, 200, 150);
circle(150, 400, 100);
circle(300, 300, 50);
```

Make sure you choose your own numbers. Run your code to see what happens.

This is what my code makes:

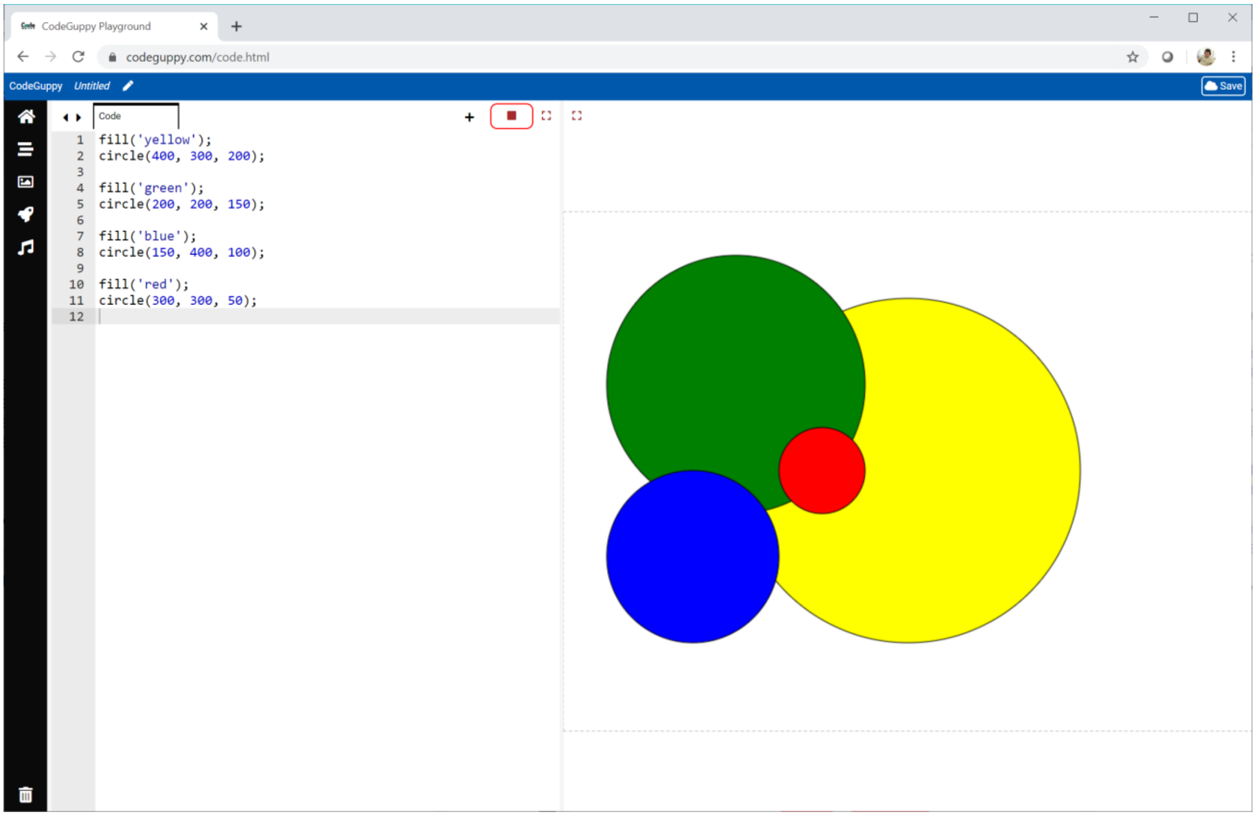


Let's try one more idea. Change the colour of each circle by using more **fill** instructions.

Here is my example:

```
fill('yellow');
circle(400, 300, 200);

fill('green');
circle(200, 200, 150);

fill('blue');
circle(150, 400, 100);

fill('red');
circle(300, 300, 50);
```

This is what my code makes. Your own code will draw a different picture.



Experiment by adding even more **fill** and **circle** instructions.

Try adding **square** instructions too.

1.2 - Coordinates & Size

level

What We'll Do

In this project we're going to:

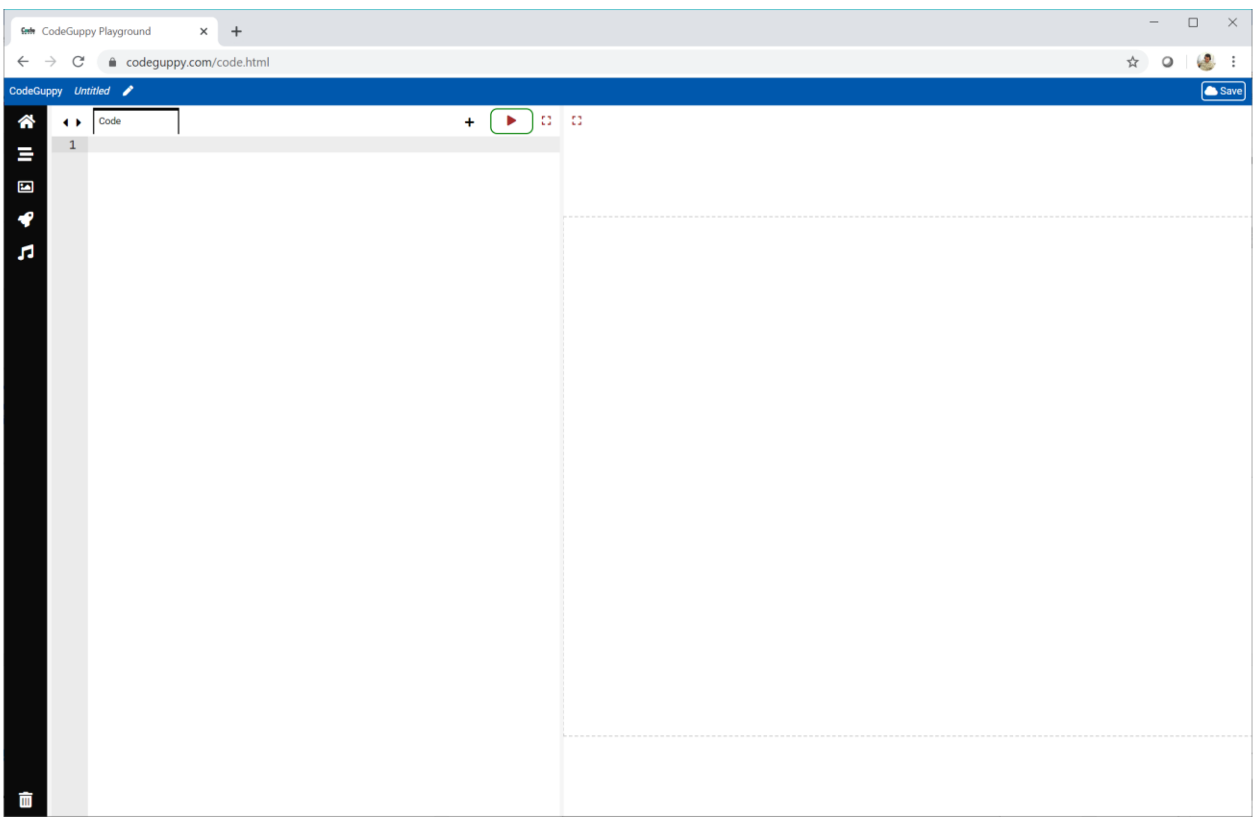
- use **coordinates** to put shapes on a canvas
- learn how to choose the **size** of a shape

Start a New Program

Log in to codeguppy.com if you haven't already.

Create a new program by clicking the "CODE NOW" button.

Your code window should look like this:



Those Circles Again

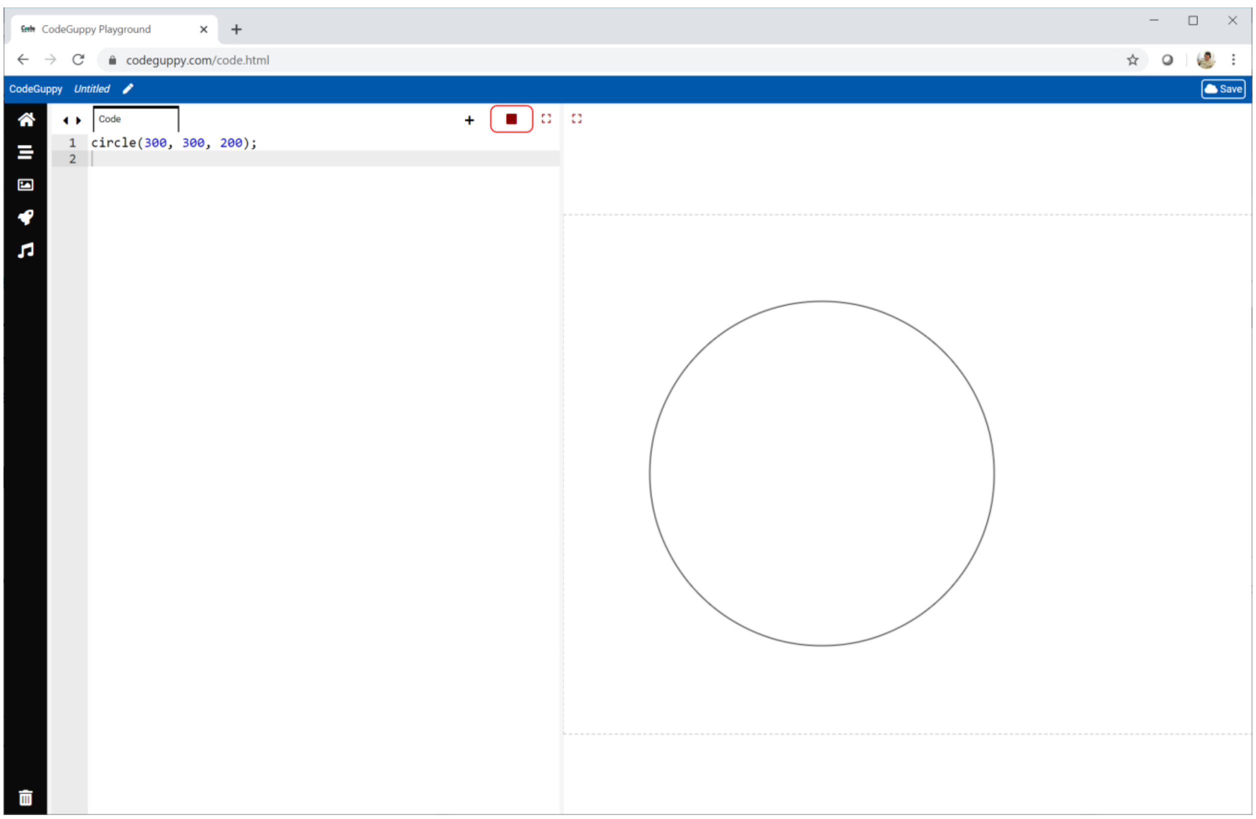
In the last project we wrote some code to draw a circle.

Type the same code into the empty **draw** part.

```
circle(300, 300, 200);
```

Run the code by clicking the play button at the top.

You should see something like this:

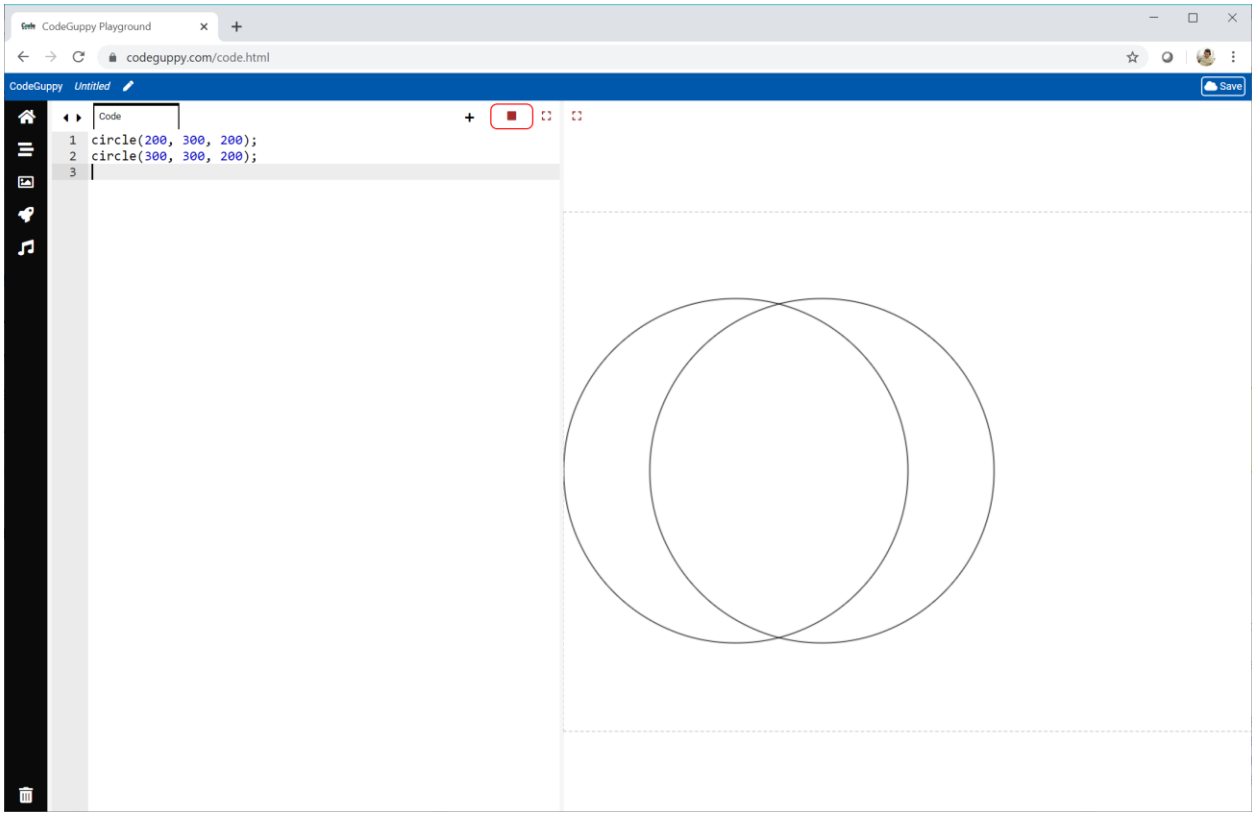


It's the same picture we had before.

Now add another **circle** instruction just before the one we already have. This time change the first number from **300** to **200** like this:

```
circle(200, 300, 200);
circle(300, 300, 200);
```

Run the code to see what happens.

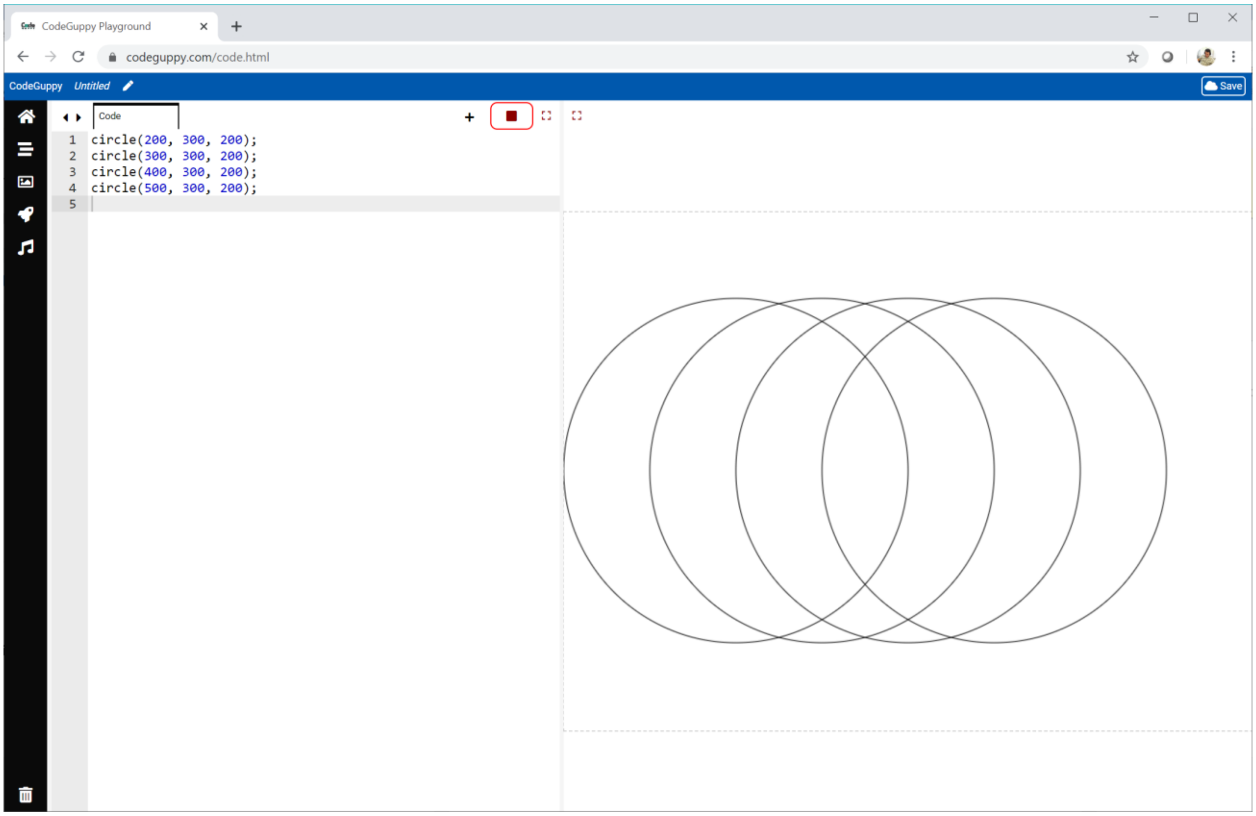


Can you work out why this happens?

Add more circle instructions after the one we already have, and change the numbers like this:

```
circle(200, 300, 200);
circle(300, 300, 200);
circle(400, 300, 200);
circle(500, 300, 200);
```

Run the code again.

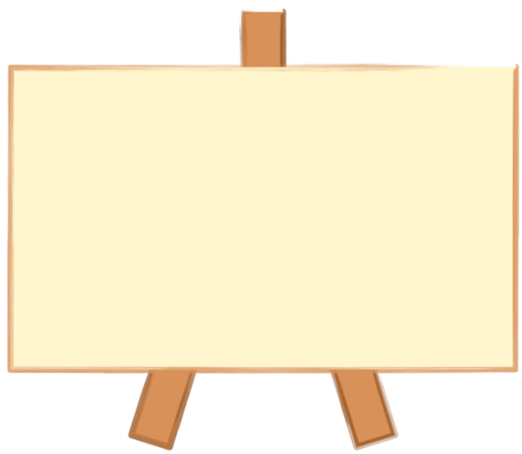


We have four circle instructions, and the only difference is the first number. Can you work out what that number does?

Coordinates on a Canvas

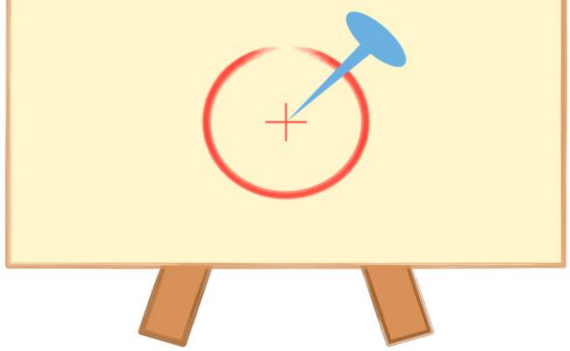
When we write code to draw shapes, they are drawn on a canvas.

A canvas is just like a piece of paper.

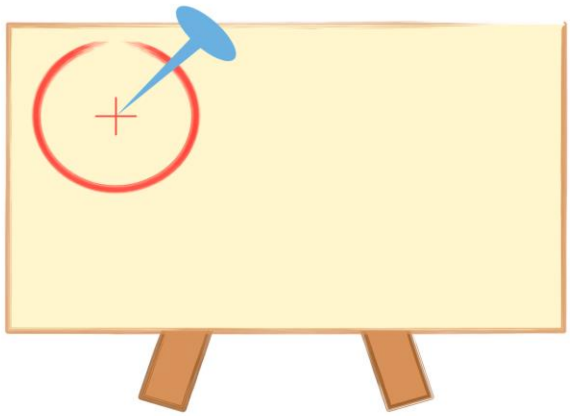


If we ask our computer to draw a circle, it needs to know exactly where to draw it.

The next picture shows a circle with a pin exactly on the middle of the circle.



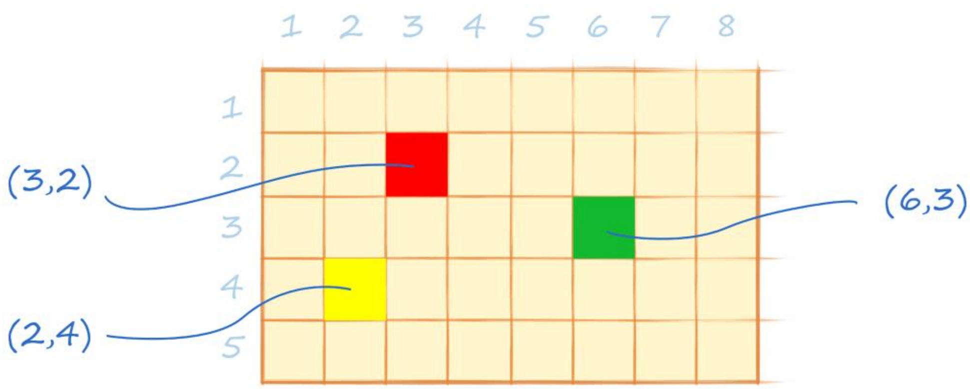
If we wanted to move the circle, we would choose a new point on the canvas to pin the centre of the circle to.



How do we tell our computer exactly which point on the canvas we mean?

We use **coordinates.** You may remember coordinates from school.

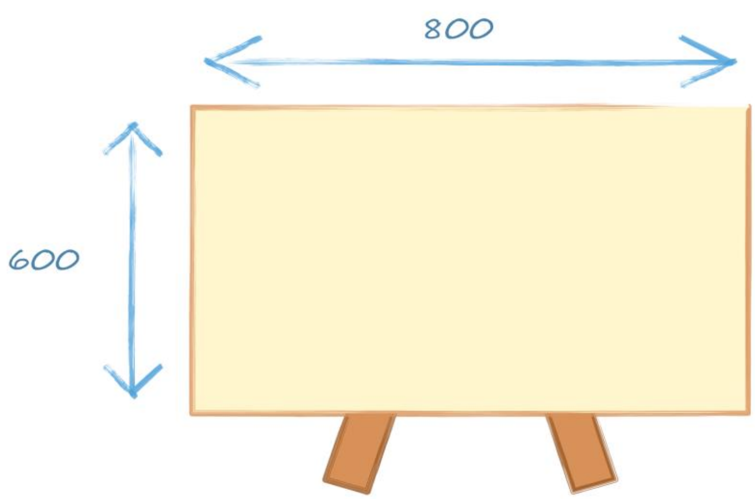
Look at these examples of coordinates.



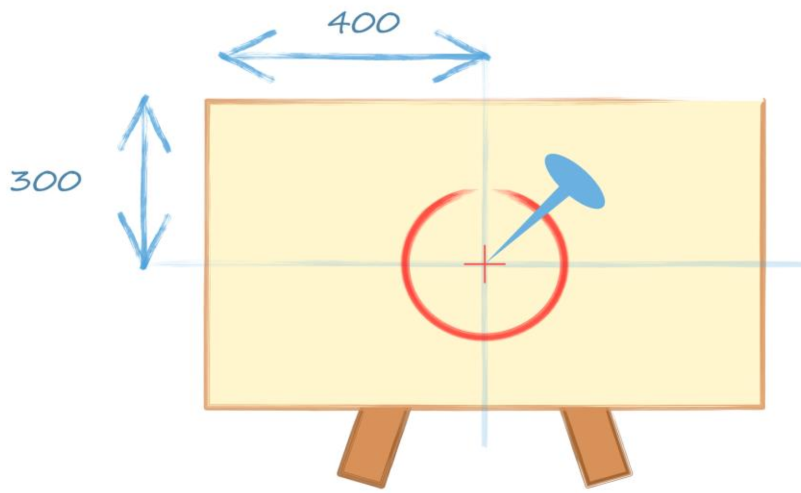
The red square is at **(3, 2)**. That's because it is **3 along** and **2 down**.

The green square is at **(6, 3)** because it is **6 along** and **3 down**.

Our canvas is bigger than that small grid. It is **800 pixels wide**, and **600 pixels down**.



If we wanted to put a circle in the middle, where would we pin the centre of the circle?



You can see the centre of the circle would be **400 across** and **300 down.** We can write that as **(400,300)**.

The code to draw a circle in the middle of the canvas is:

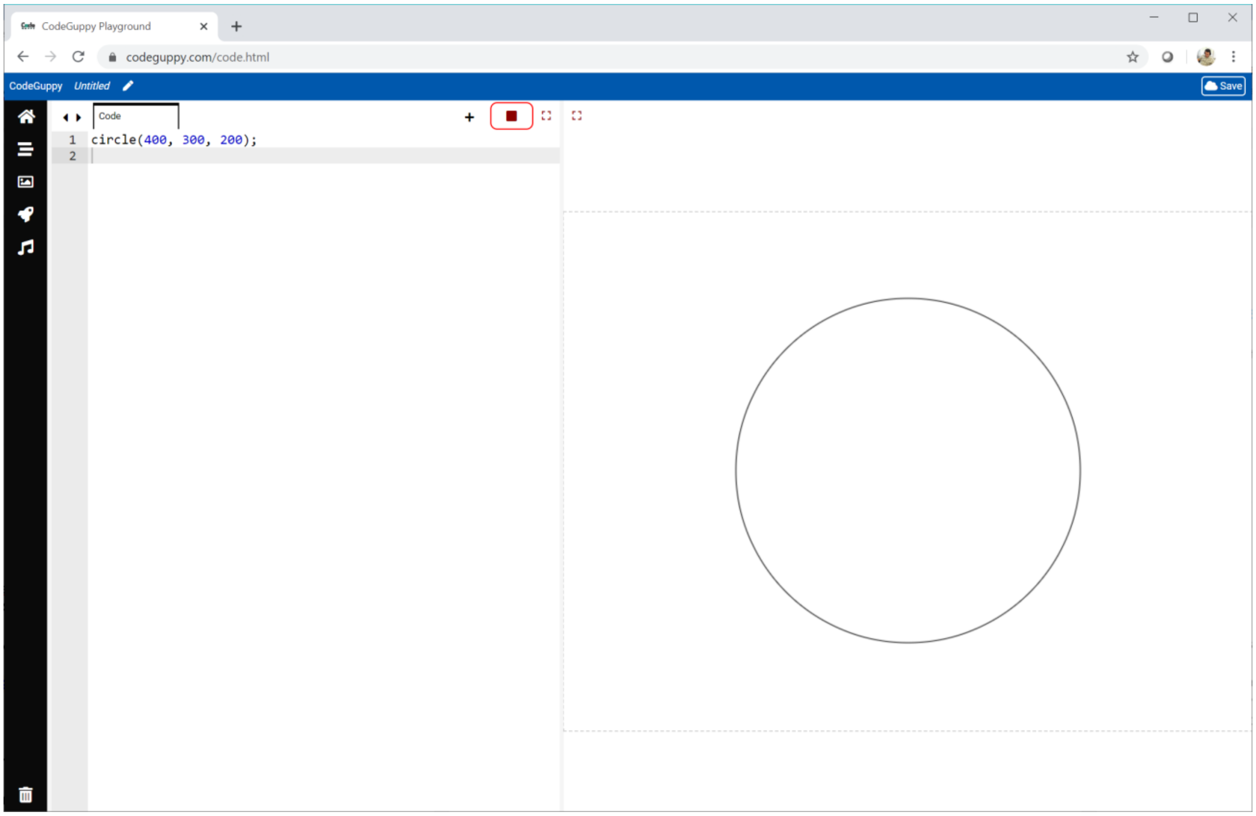
```
circle(400, 300, 200);
```

You can see that:

- the **first number 400** is how far **along** the circle centre is.
- the **second number 300** is how far **down** the circle centre is.

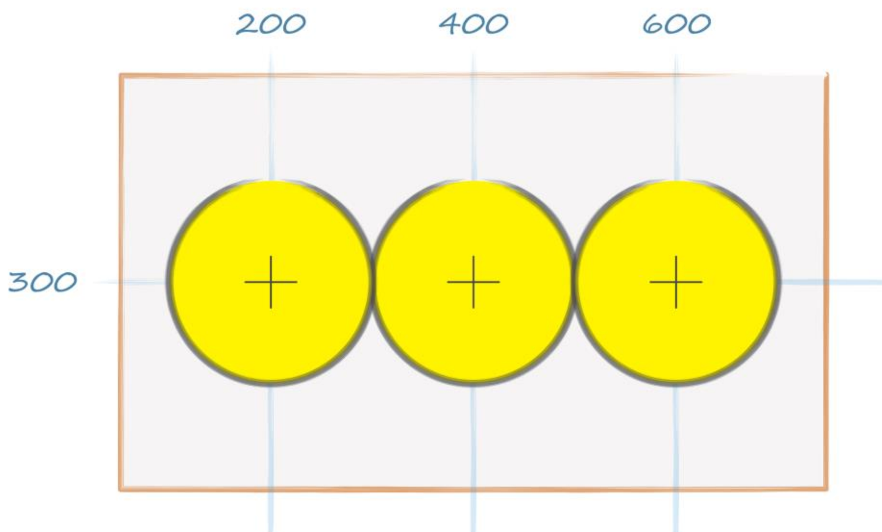
Try the code yourself.

Your drawing should look like this:



Try It Yourself

Have a look at this picture.

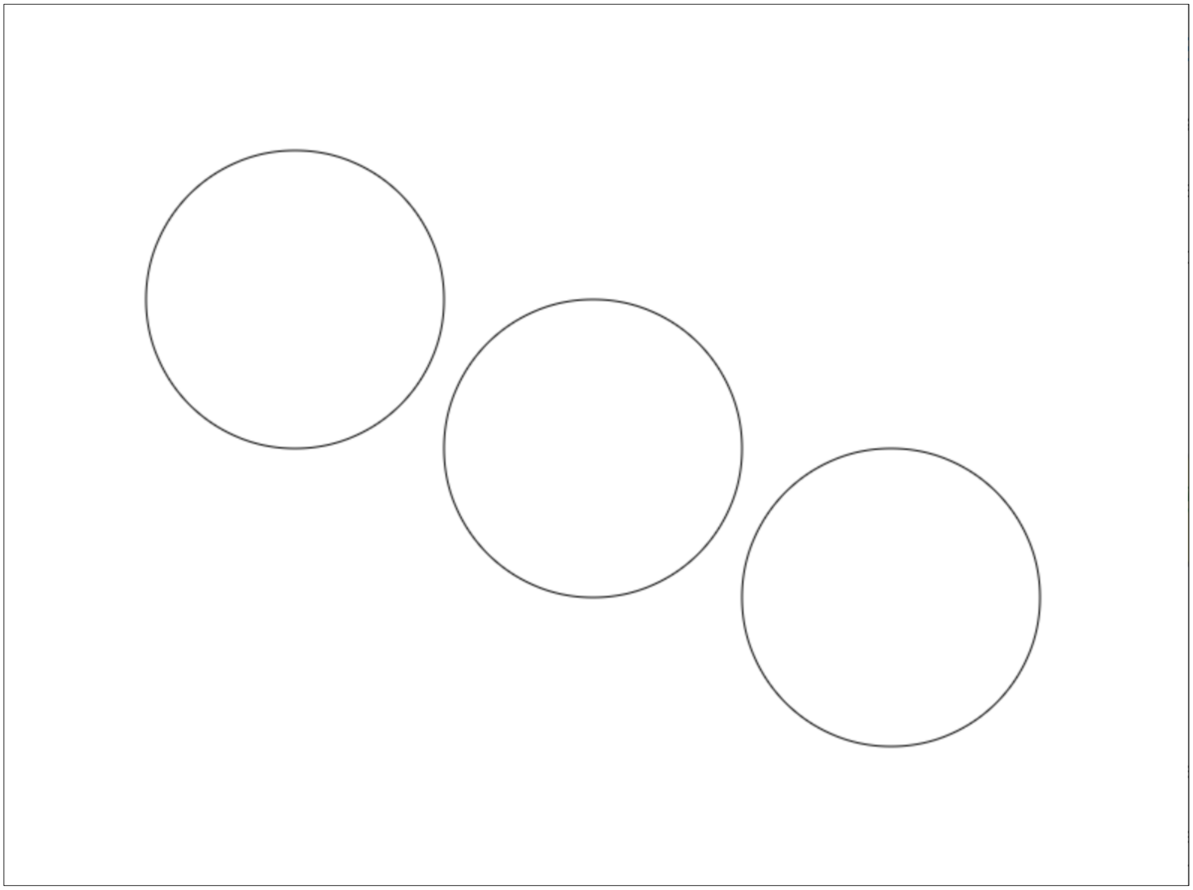


You can see that there are three circles:

- the middle circle is at **(400, 300)** just like before
- the left circle is at **(200, 300)**
- the right circle is at **(600, 300)**

Try writing some code that draws these three circles.

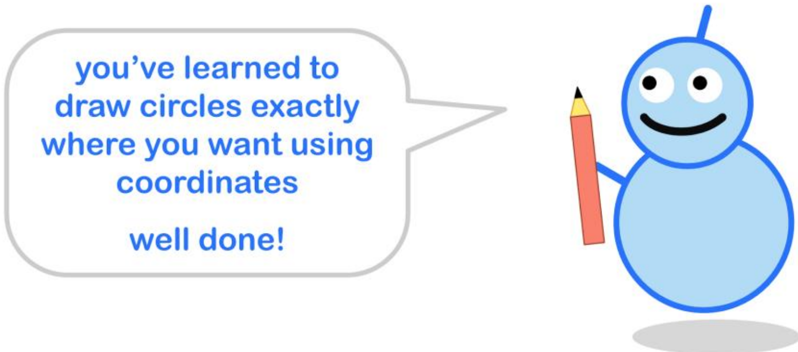
After you've done that, see if you can write code that draws circles that go down the canvas like this:



Remember the second number is how far up or down the circle is on the canvas.

Here's how I did it.

```
circle(200, 200, 100);
circle(400, 300, 100);
circle(600, 400, 100);
```

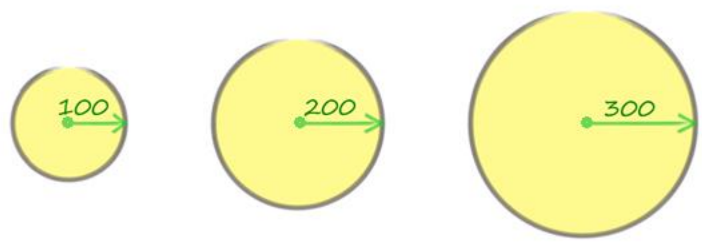

you've learned to draw circles exactly where you want using coordinates
well done!

Circle Size

We just learned how to draw a circle at different places on the canvas.

It would be good if we could change the size too. Let's do that now.

Have a look at this picture:



We can see three circles. A **small** one, a **medium** one and a **large** one.

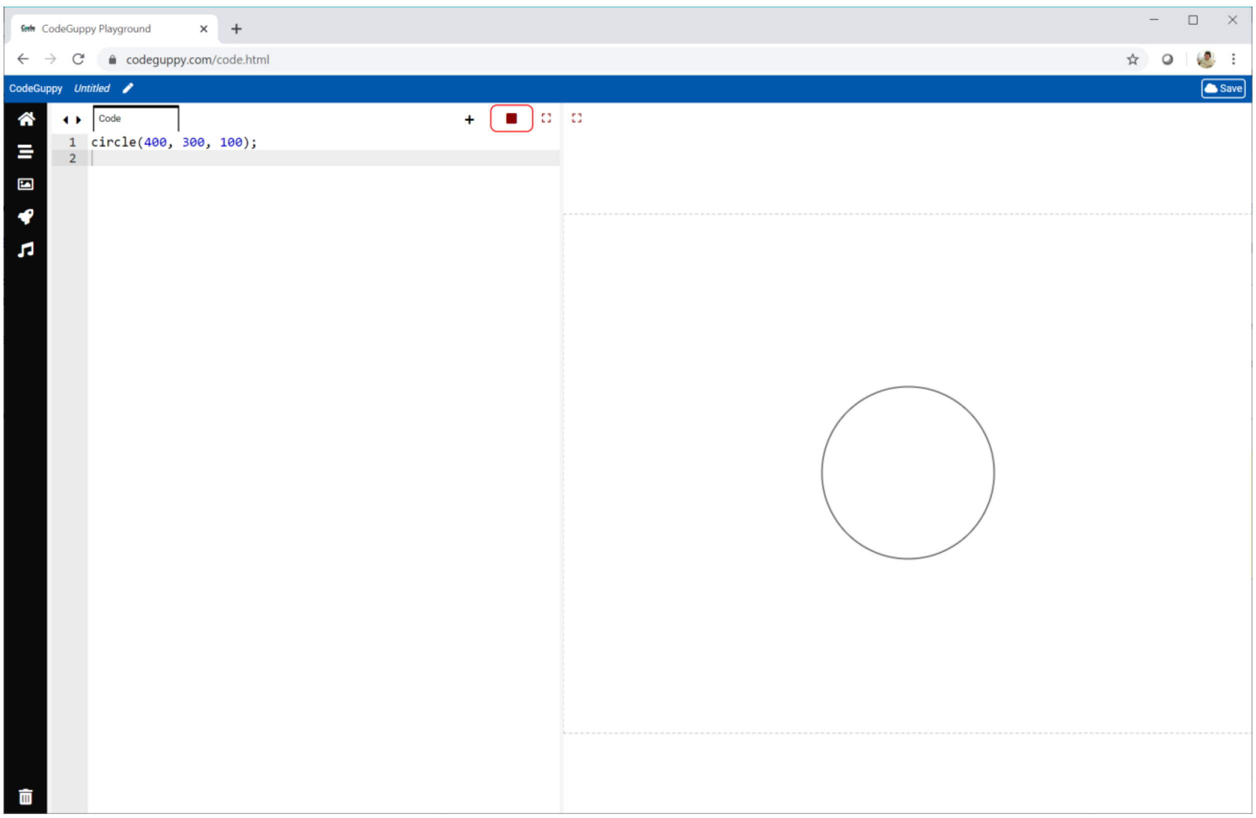
A good way to describe the size of the circle is the **radius**. That's the distance from the center of circle to margin.

Let’s look at some very simple code again.

```
circle(400, 300, 100);
```

We’ve seen this code before. It draws a circle in the middle of the canvas. The middle of the circle is exactly at **(400, 300)**.

Here is the drawing that code makes.



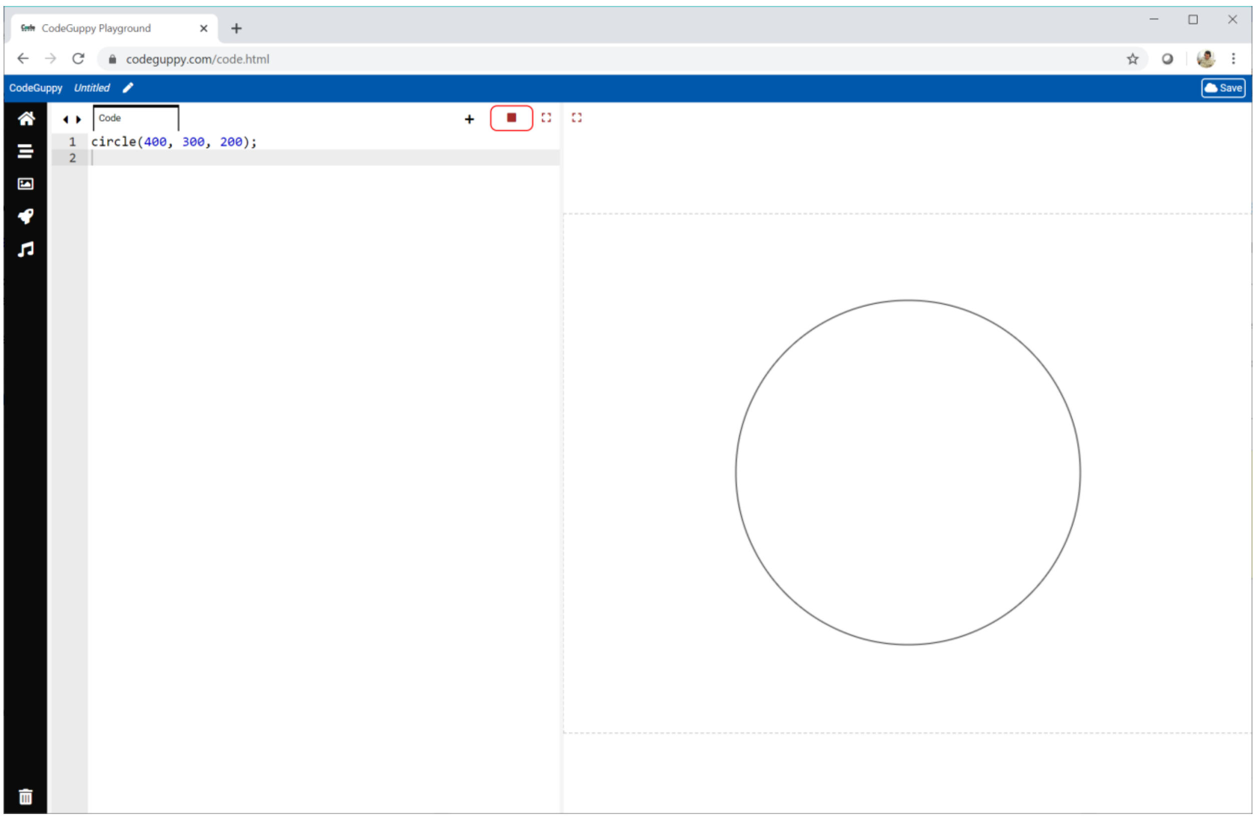
We haven’t talked about the last number, **100**.

Change the **100** to **200** to see what happens.

```
circle(400, 300, 200);
```

What do you get?

You should get a much larger circle.



What happens if you choose a smaller number like **30**?

That third number is the size of the circle. A large number draws a larger circle. A small number draws a small circle.

Try It Yourself

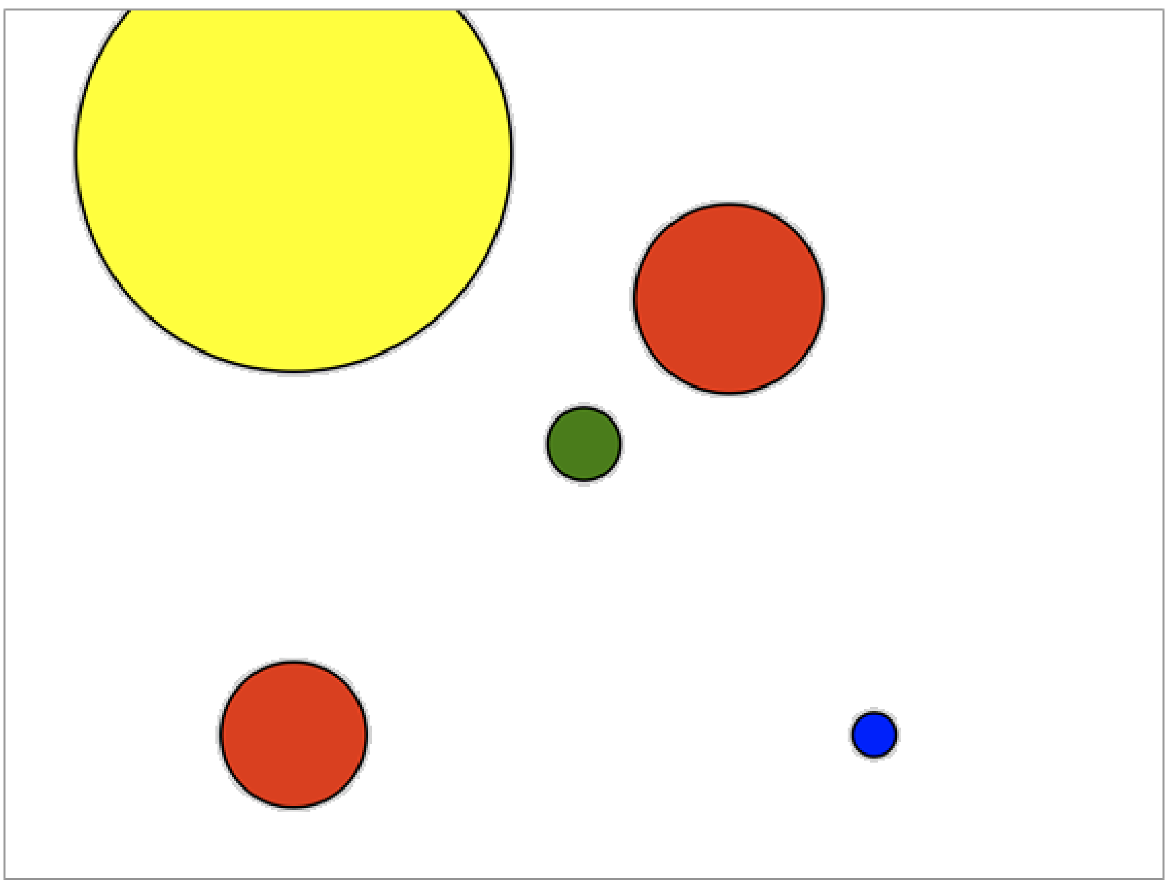
We’ve learned how to draw a circle exactly **where** we want on the canvas.

We’ve also learned how to set the **size** of the circle.

In the last project, we learned how to choose a **colour** for the circle.

See if you can combine what you’ve learned to make a picture with about five circles. Give the circles a different colour, size and location.

Here’s what I made. Your code will make a different picture.



Don’t Forget Squares

We’ve learned how to draw a circle exactly **where** we want, and the **size** we want.

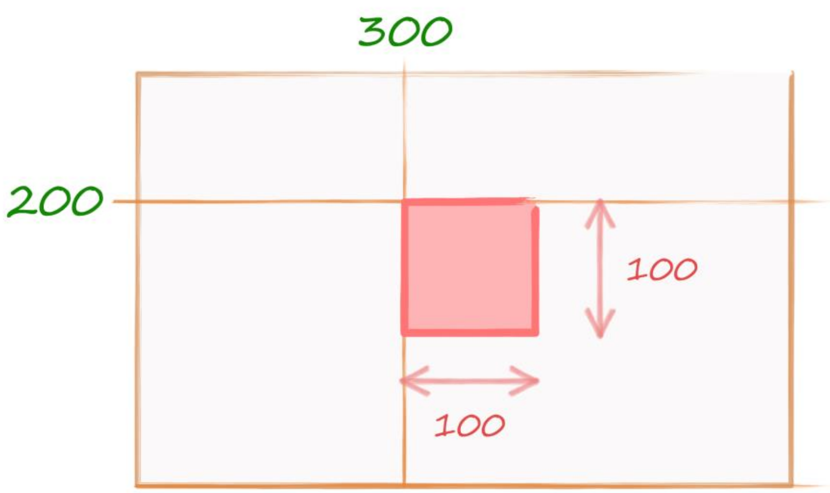
Can we do the same with squares? Yes we can!

Look at this code for drawing a square:

```
square(300, 200, 100);
```

Can you guess what the numbers are for?

This next picture explains what the numbers do.



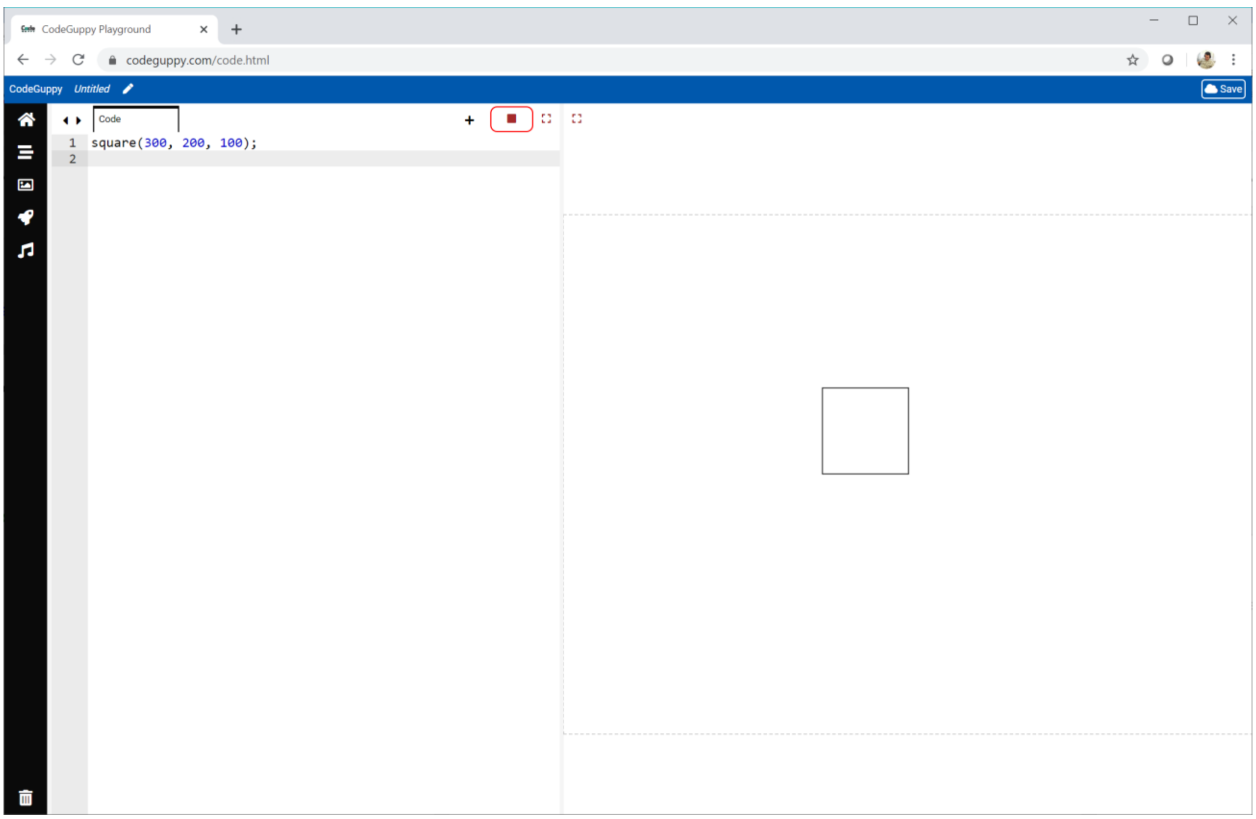
The **first number 300** sets how far **along** the square is.

The **second number 200** sets how far **down** the square is.

That means the top left corner of the square is at **(300, 200)**.

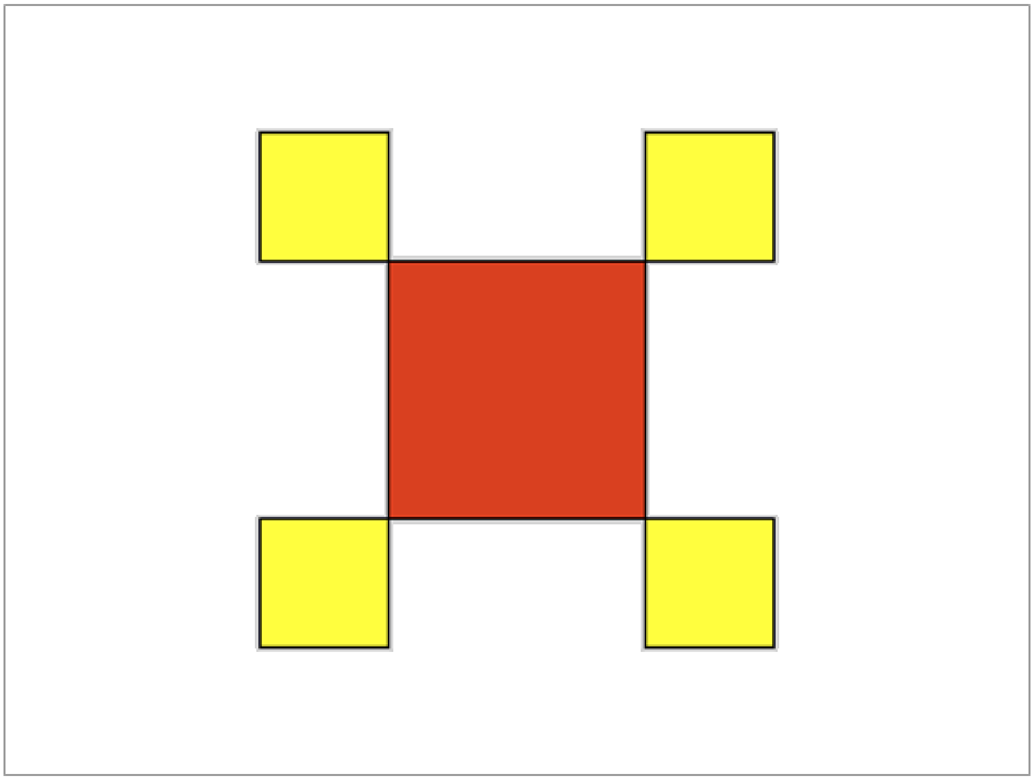
The **third number 100** sets the size of the square.

Try that code to check you get a square.



Challenge!

Can you write code code that creates a drawing like this?



This is a challenge that might take some experimenting to get right.

Using a pen and paper to plan the coordinates and sizes can be helpful.

1.3 - Random Numbers

level

What We'll Do

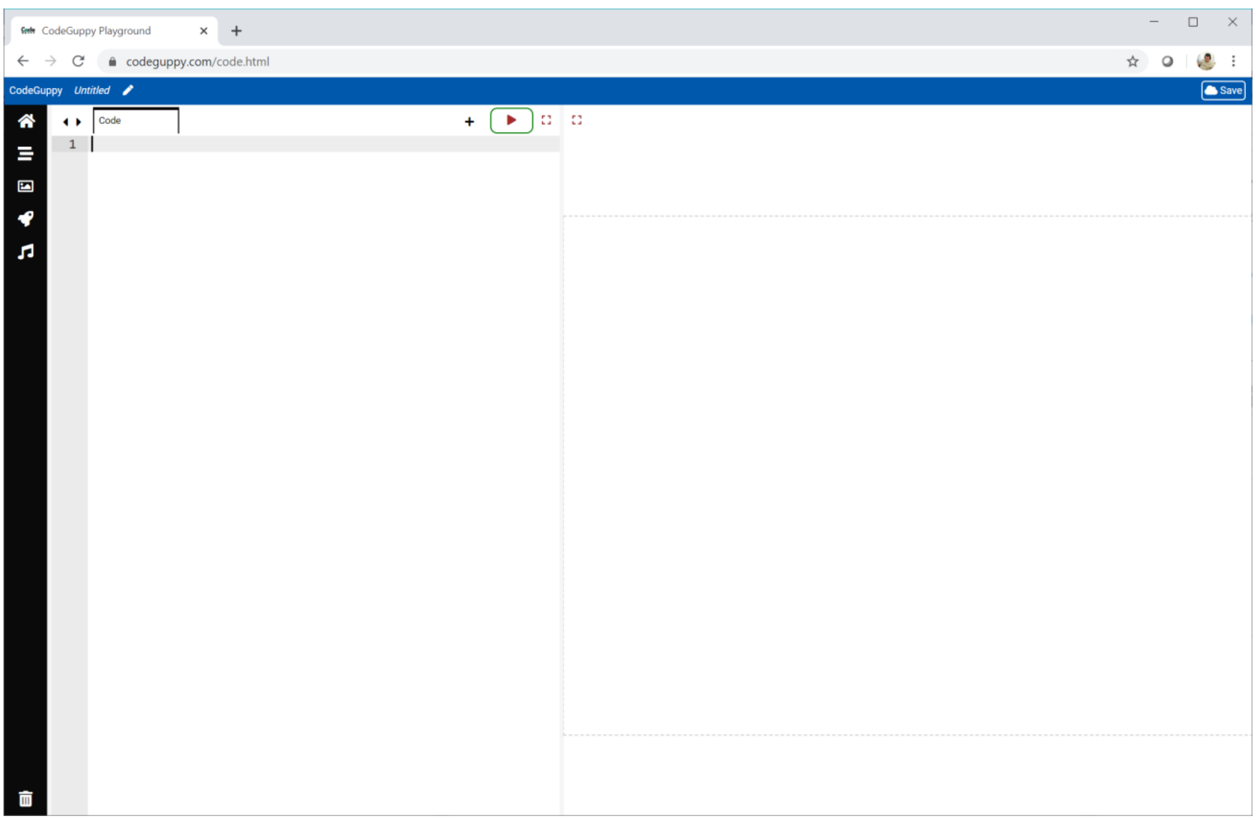
In this project we're going to:

- get our computer to pick **random** numbers for us

Start a New Program

Log in to codeguppy.com if you haven't already.

Your code window should look like this:



Rings & Circles

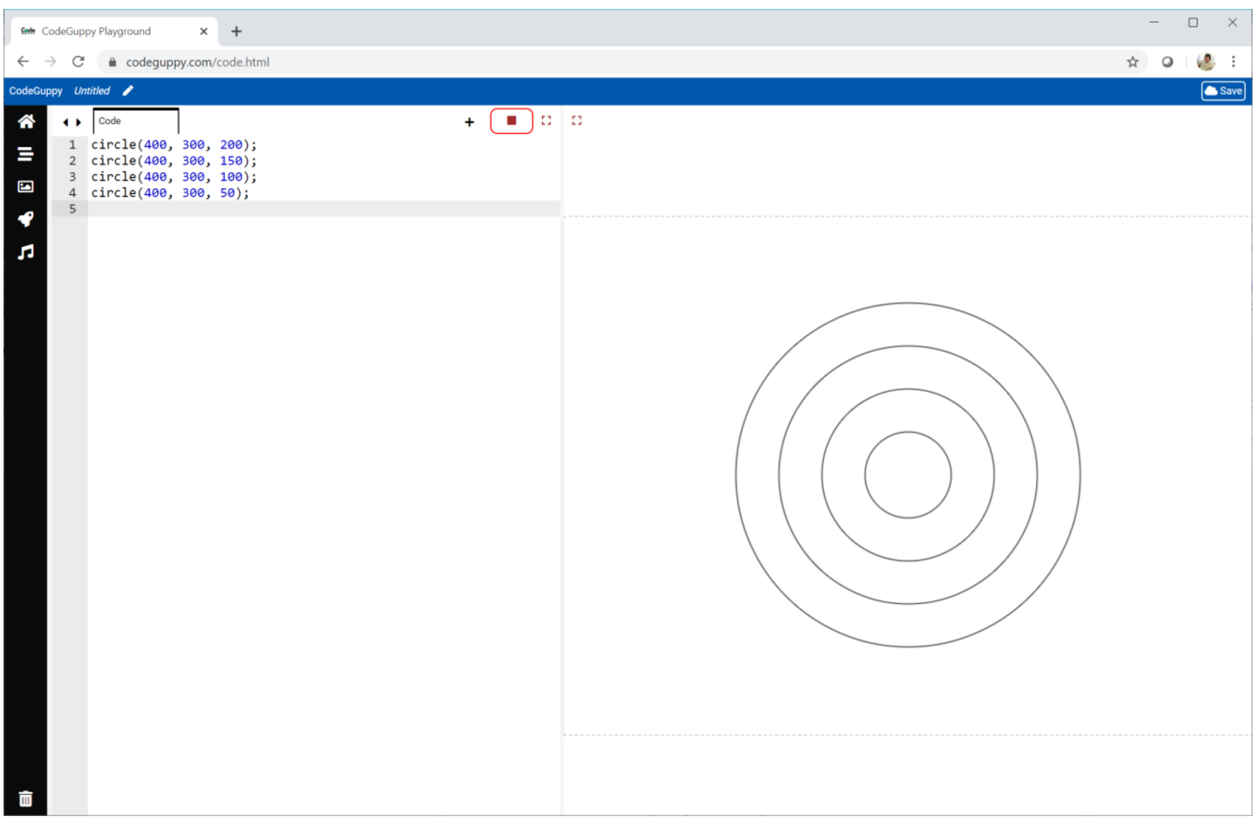
Have a look at this code. Can you work out what it does?

```
circle(400, 300, 200);
circle(400, 300, 150);
circle(400, 300, 100);
circle(400, 300, 50);
```

There are four circles, and all of them are located in the middle of the canvas.

The difference between these circles is their radius.

Run the code. You should see these four white circles.



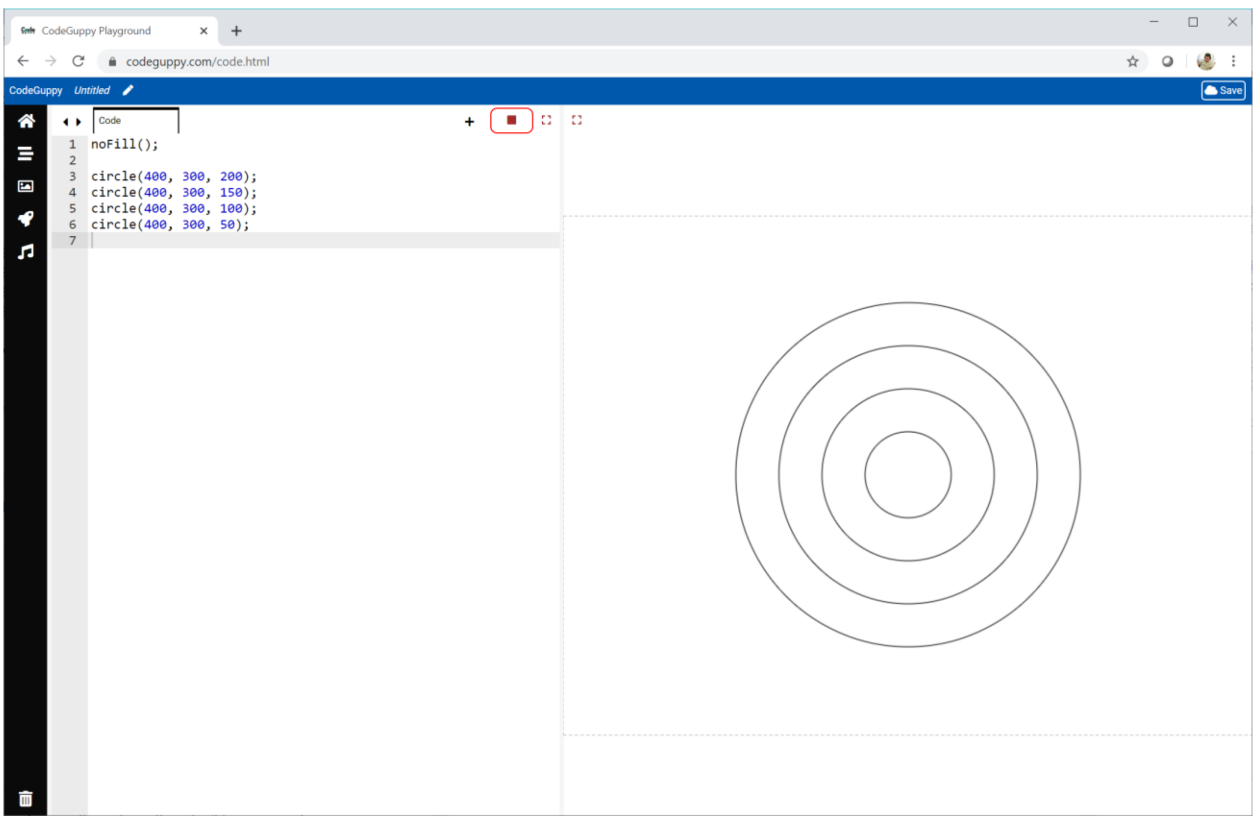
Let’s make those circles see-through, so they don’t have any colour.

We can do this using a **noFill** instruction before the circle instructions.

```
noFill();

circle(400, 300, 200);
circle(400, 300, 150);
circle(400, 300, 100);
circle(400, 300, 50);
```

Make sure you spell **noFill** correctly. The **F** is a capital letter.

Run the code to see what happens.



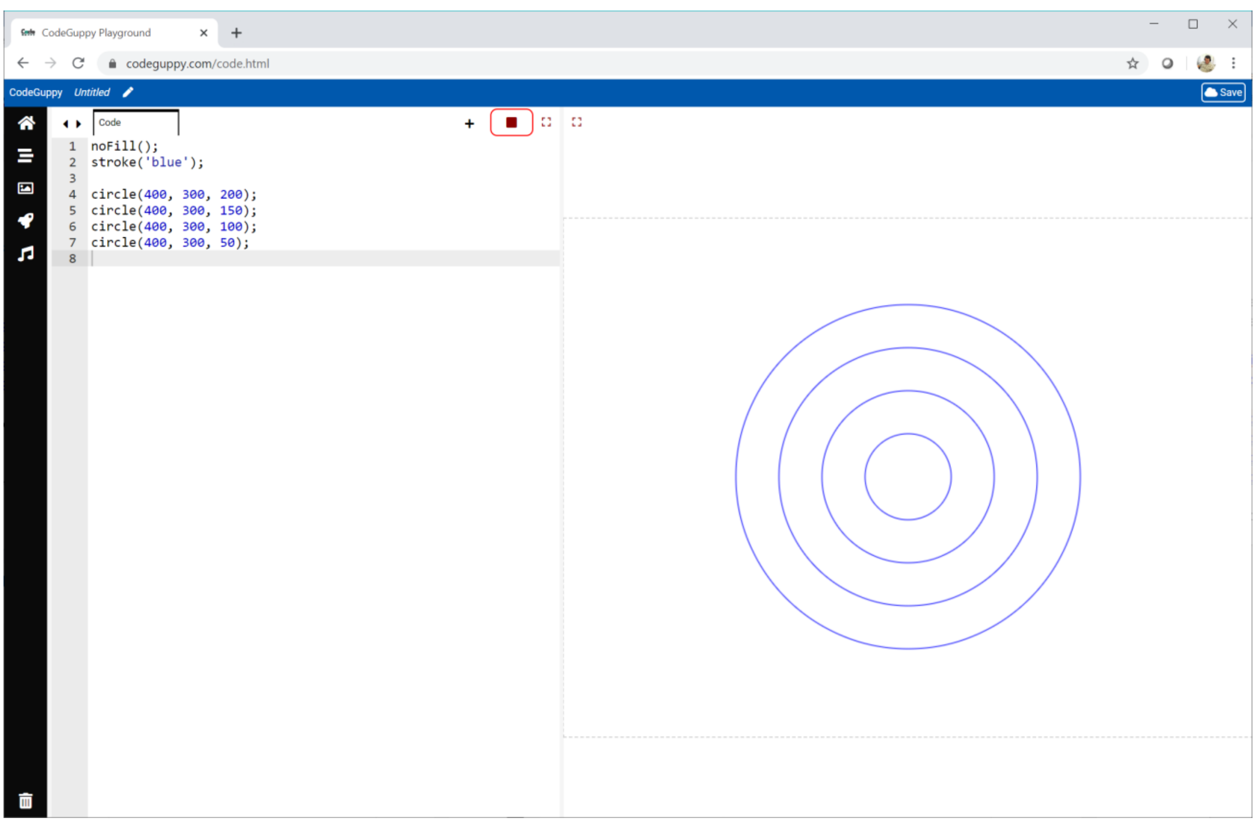
Those circles are completely see-through now! This is not evident now but in the future when we'll change the background color you'll notice this effect.

You might think those circles are a bit boring. Let's change their colour.

We can set the outline colour of a shape with the **stroke** instruction. You can see here we've set the outline to **blue**.

```
noFill();
stroke('blue');

circle(400, 300, 200);
circle(400, 300, 150);
circle(400, 300, 100);
circle(400, 300, 50);
```

Run the code to see it working. You should get blue rings like this.

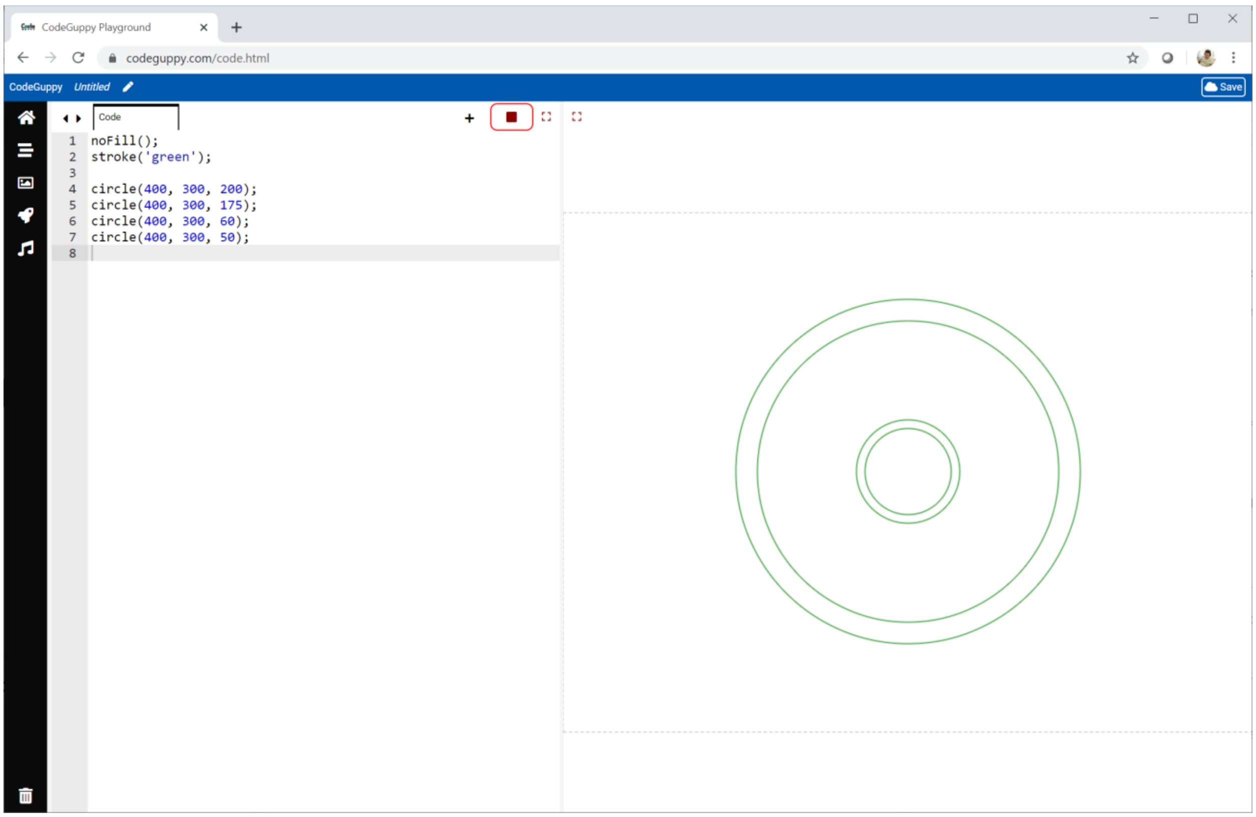


Try It Yourself

Try choosing different colours for the rings.

Also, try changing the sizes of the four circles.

Here’s my code:



I chose **green** for the circle outlines, and radiuses of **200**, **175**, **60** and **50**.

Random Sizes

Sometimes it's fun to get our computers to choose things for us.

Why don't we get our computer to choose the sizes of our four circles.

Look at this code:

```
circle(400, 300, 200);
```

We already know this code draws a circle of radius **200**.

Now look at this code:

```
circle(400, 300, randomNumber(200) );
```

What's changed?

We've replaced the size number **200** by **randomNumber(200)**.

The instruction **randomNumber(200)** asks our computer to pick a number between **0** and **200**.

The number could be **12**, or **102**, or **399**, or **37** or **256**, … or something else.

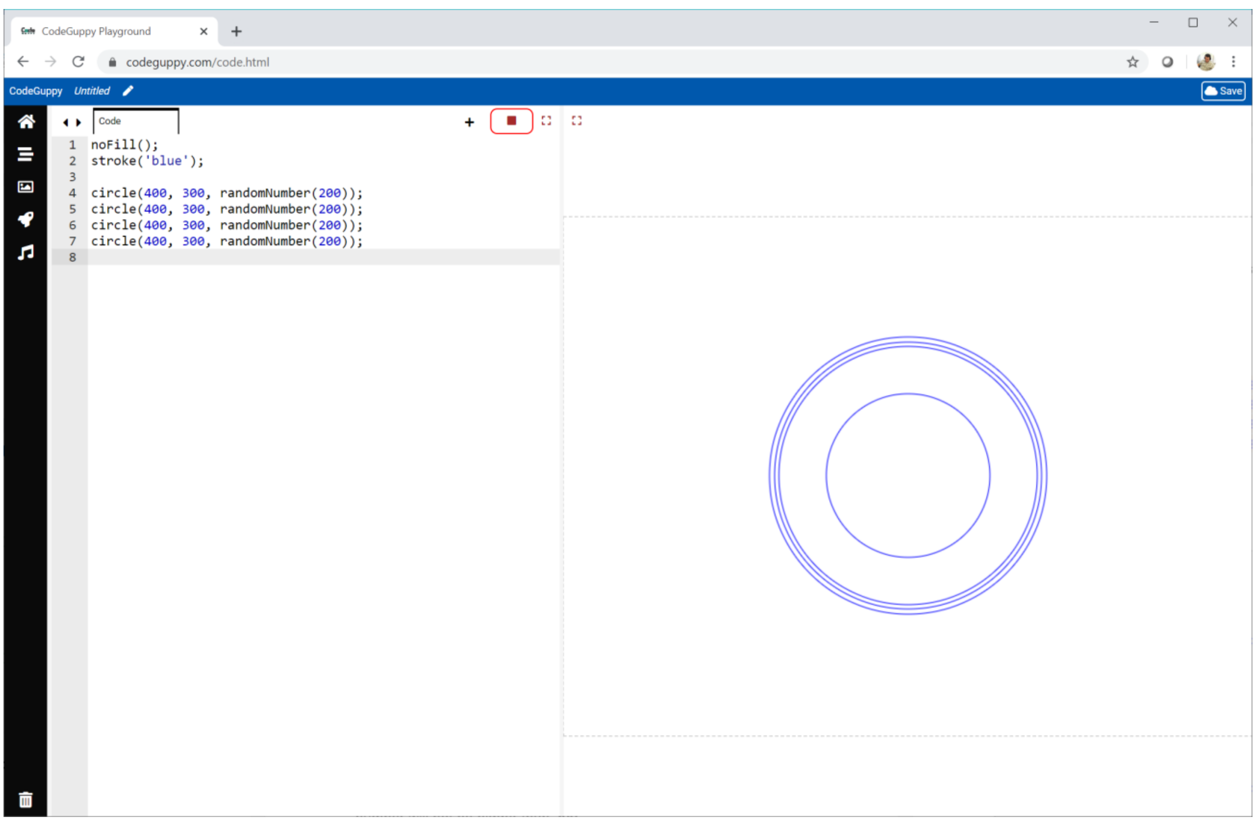
We don't know what the computer will choose. All we know is that the number will not be bigger than **200**.

Let's change our code so that all four circles now use **randomNumber** to pick a random size.

```
noFill();
stroke('blue');

circle(400, 300, randomNumber(200));
circle(400, 300, randomNumber(200));
circle(400, 300, randomNumber(200));
circle(400, 300, randomNumber(200));
```

Try it to see what you get.

This is what I get.

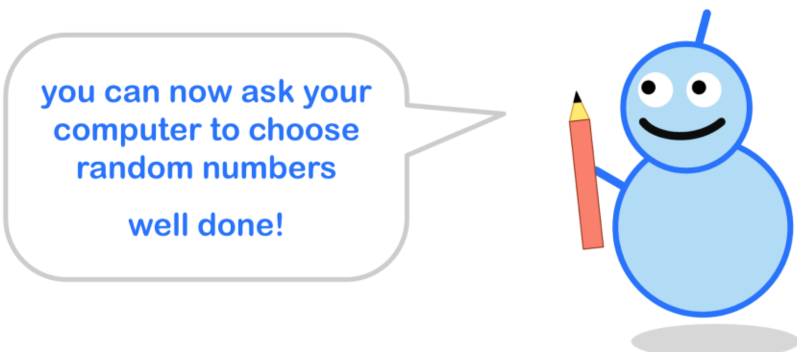


Your picture will be different because the sizes your computer chooses will be different.

Run your code again using the Stop and Play buttons at the top of the page.

Do you see a different picture? Why is that?

Try replaying your code several times.



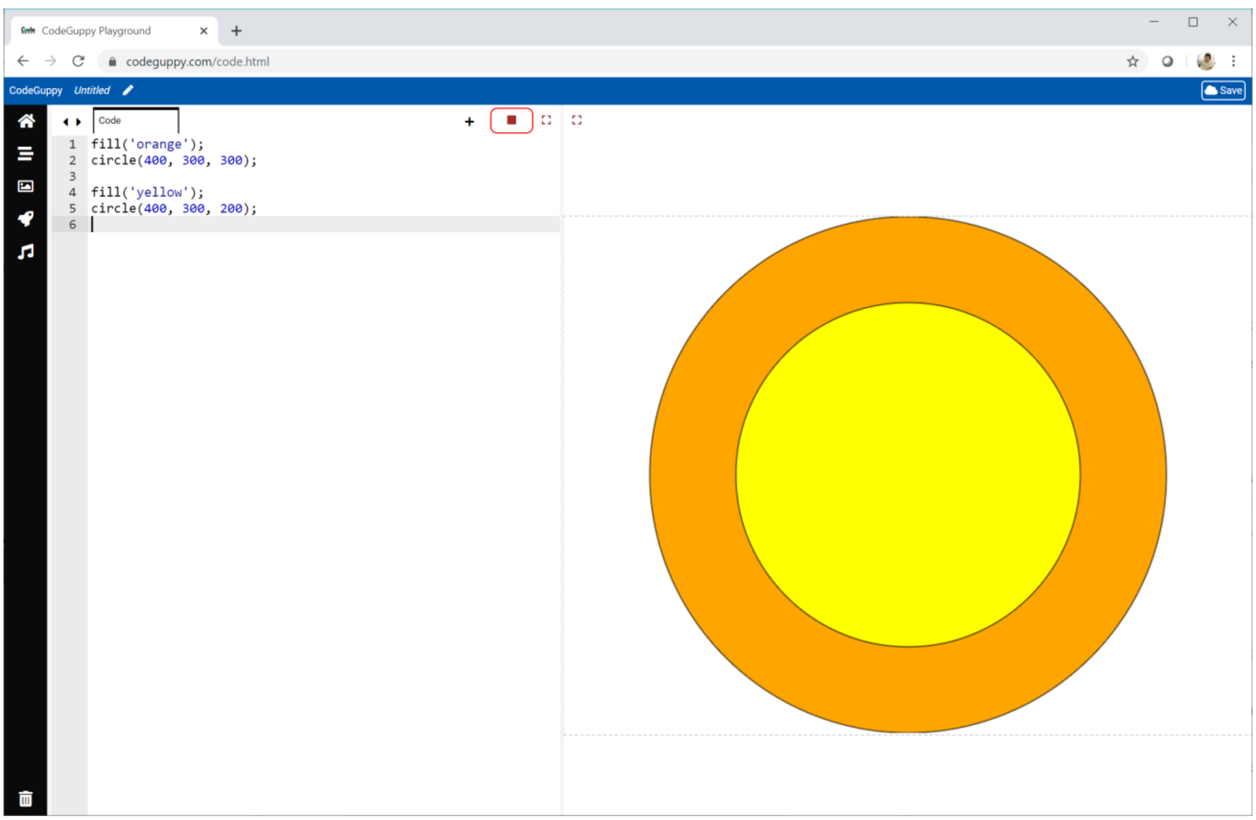
Random Colours

Let's get our computer to choose **colours**, not just numbers.

Have a look at this code:

```
fill('orange');
circle(400, 300, 300);

fill('yellow');
circle(400, 300, 200);
```

Run the code and you'll see it draws a smaller yellow circle on a larger orange circle.



Wouldn't it be cool if our computer picked a colour all by itself for the smaller circle?

To do this we need a **list** of colours to choose from.

To code a **list** of things we put them between square brackets **[ ]**.

Here is a list of three colours:

```
['red', 'purple', 'green']
```

You can see the colours are between square brackets **[ ]**. You can also see there are commas between each colour.

To pick something from a list at random, we use the **random** instruction. Look at the code below.

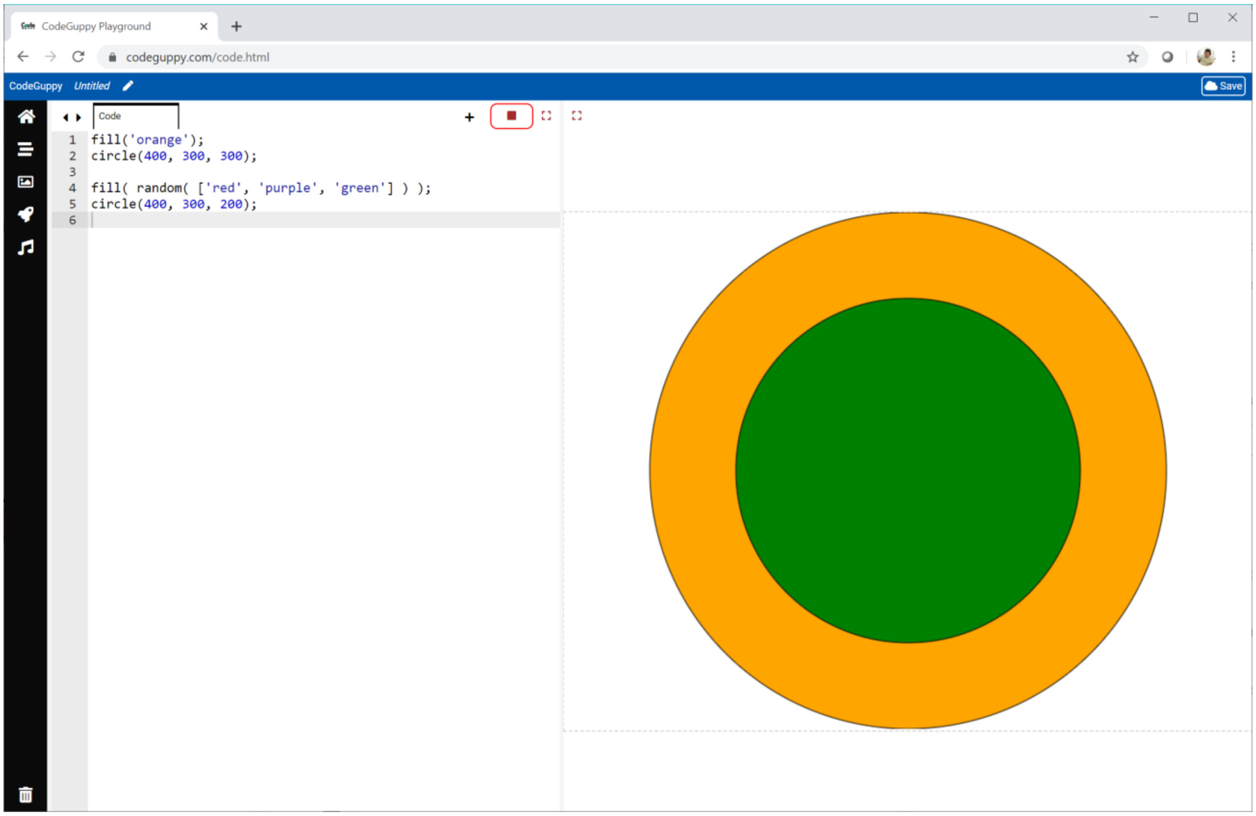
```
fill('orange');
circle(400, 300, 300);

fill( random( ['red', 'purple', 'green'] ) );
circle(400, 300, 200);
```

Instead of **'yellow'**, we use the **random** instruction to pick one of **red**, **purple** or **green**.

We don't know which colour will be chosen, because the choice will be random.

Run the code to see what you get.

Here’s what I get:



Run the code again a few times. Can you get all of the colours?

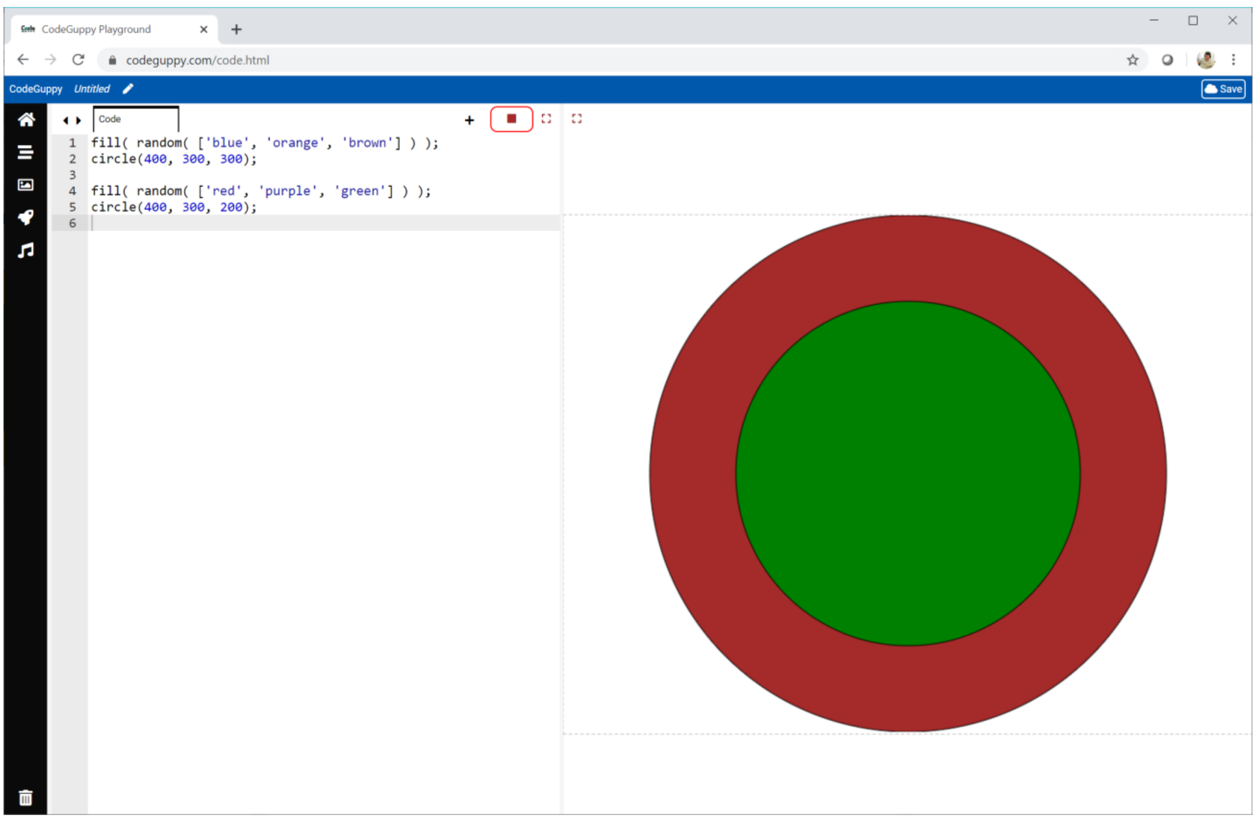
Try It Yourself

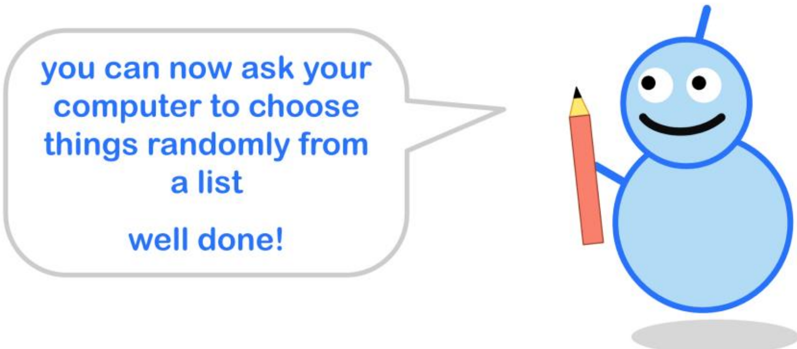
Change the list so it only has colours you like.

You can make it longer than three colours if you want.

Can you change the code so that **both circles** have colours chosen at random?

Here's what my code created:



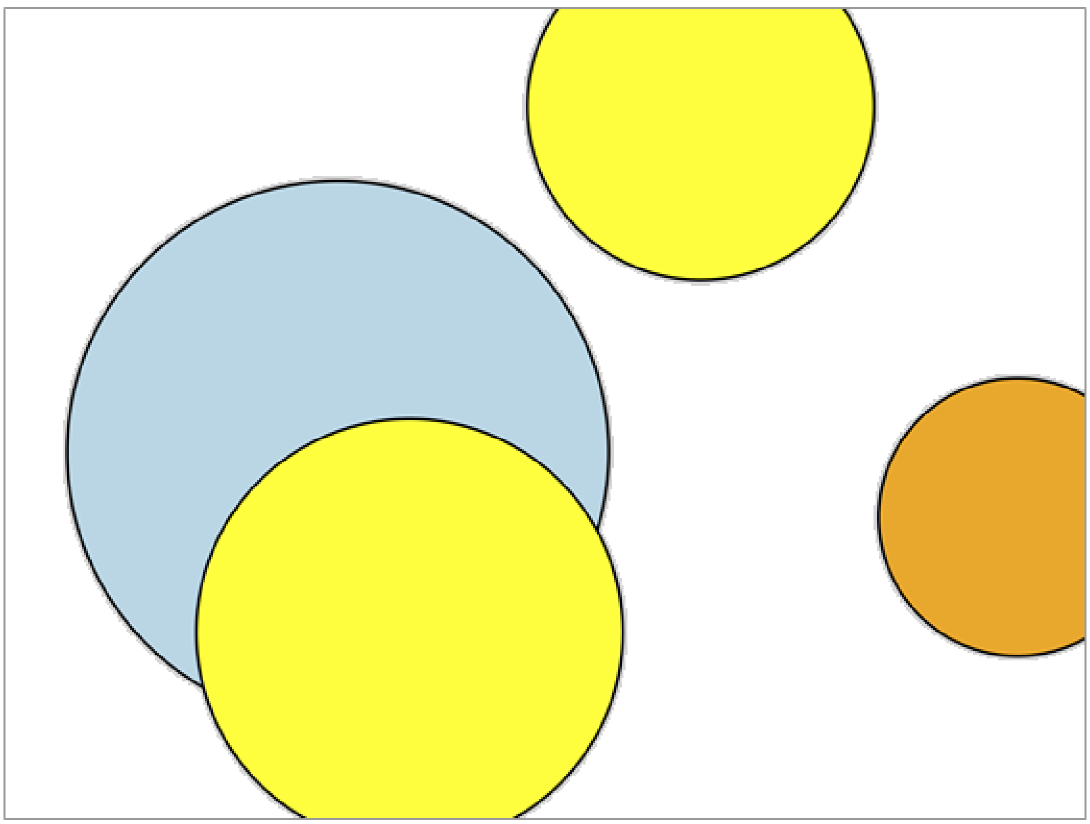

you can now ask your computer to choose things randomly from a list
well done!

Challenge!

Use what you've learned to draw four circles where:

- the **location** is a random place on the canvas
- the **size** is a random number
- the **colour** is randomly chosen from a list of colours

Here's what my code created:



1.4 - Simple Variables

level

What We'll Do

In this project we're going to:

- learn how **variables** can remember numbers
- see how variables can be **useful**

Start a New Program

Log in to codeguppy.com if you haven't already.

Create a new program by clicking the "CODE NOW" button.

Three Circles

Let’s start the new program with three simple circles.

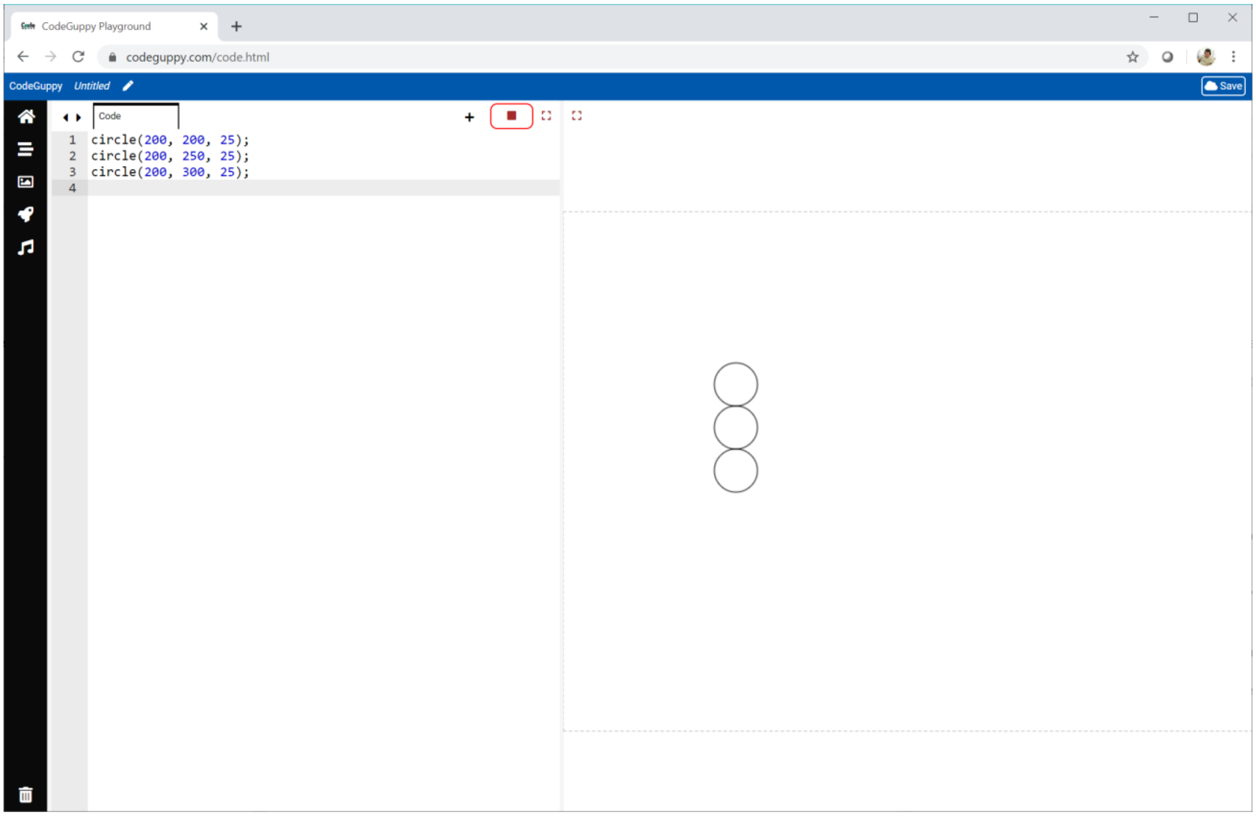
```
circle(200, 200, 25);
circle(200, 250, 25);
circle(200, 300, 25);
```

Can you see the difference between these three circles?

The x-coordinate stays the same at **200**.

The y-coordinate starts at **200** and increases by **50** for each new circle.

Run the code to see what happens.



Cool! The three circles look a chain.

They’re in a vertical line because only the y-coordinate changes.

Move The Chain

Let’s move the chain to another place on the canvas.

Here is some code to draw the circles near the middle of the canvas.

```
circle(400, 300, 25);
circle(400, 350, 25);
circle(400, 400, 25);
```

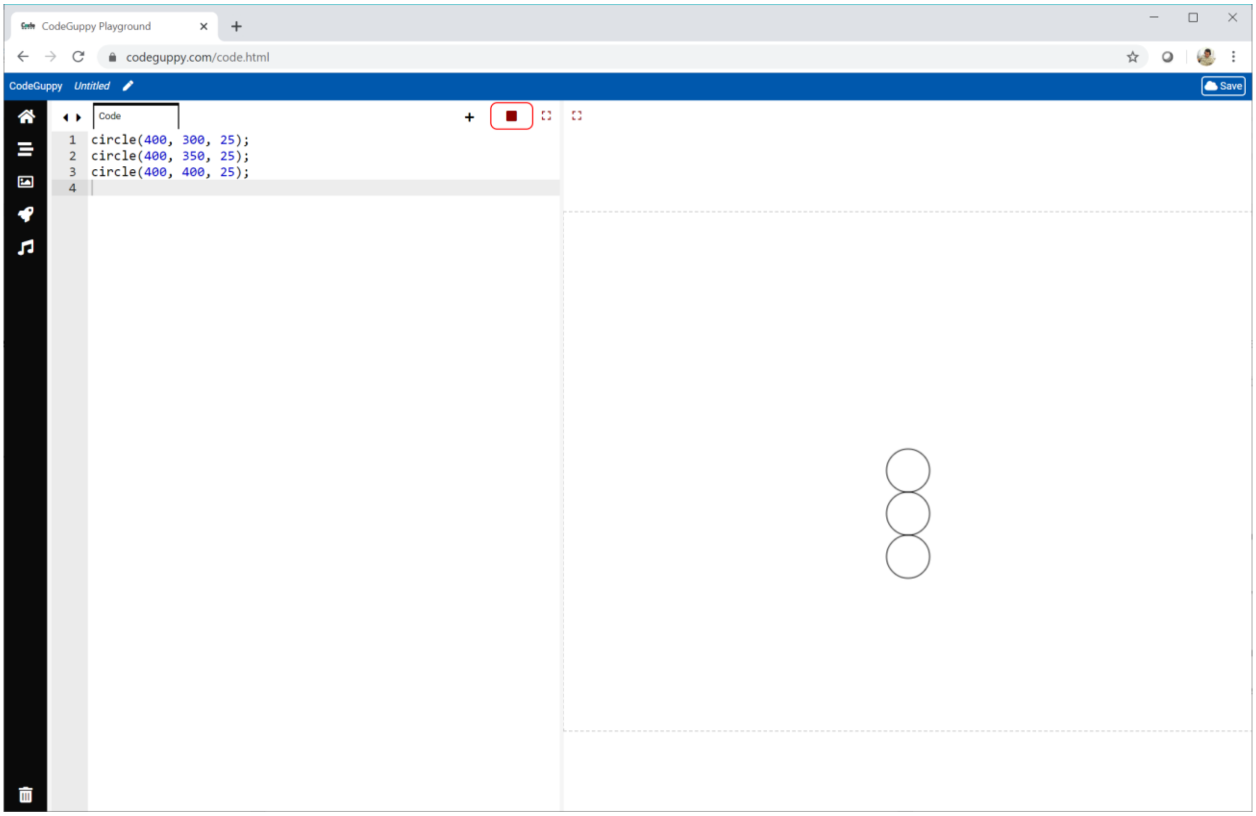
What’s changed?

The x-coordinate is now **400**. It was **200** before. That means the circles will be drawn further to the right.

The y-coordinate now starts at **300**, and grows to **350** and then **400**. It goes up by **50** every time.

This means we should still get a nice vertical line of circles, just like before.

Try the code to check it does what you expect.



That worked because we changed the location of all the circles, making sure they stayed in a line.

That’s cool, but it would be more exciting to draw the chain at a random place on the canvas.

Draw The Chain at Random Places

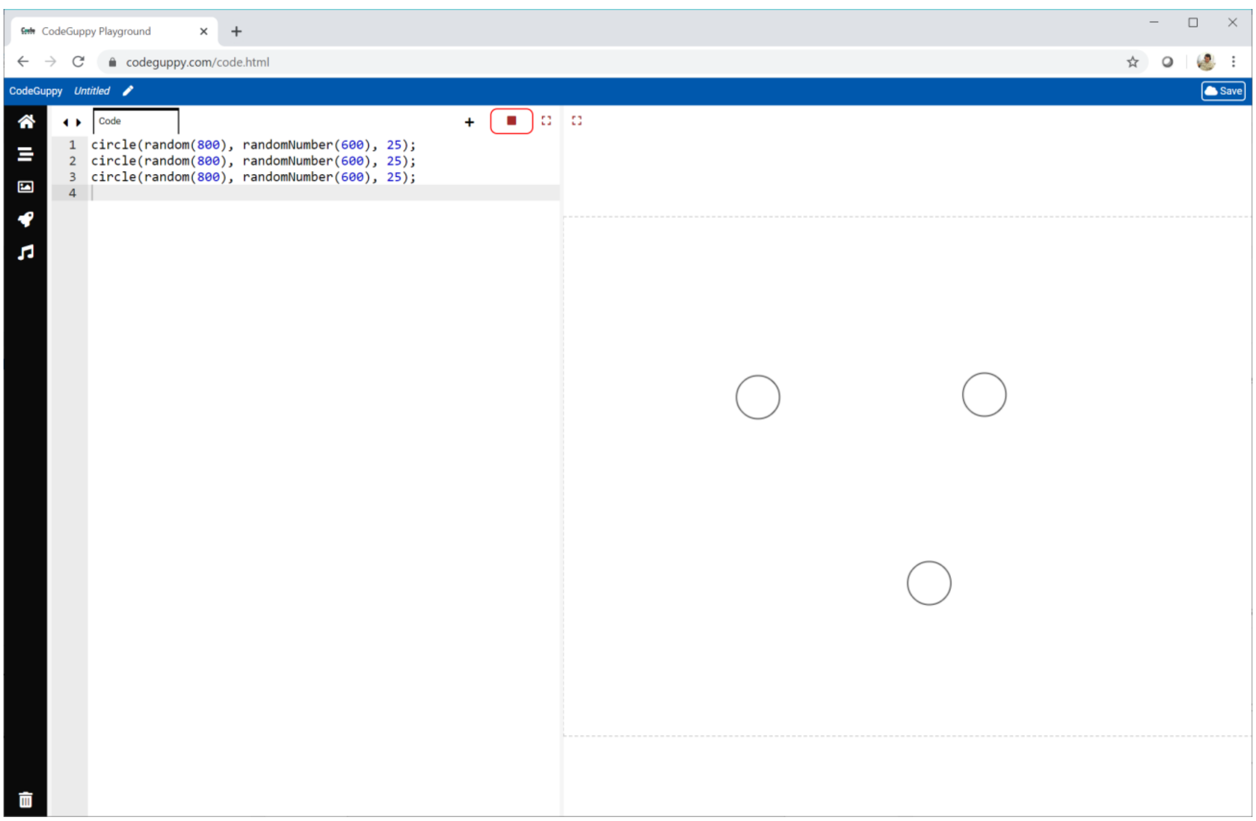
How would we draw the line of circles at a random place?

We've already used the **randomNumber** instruction to choose a random location on the canvas.

Have a look at this code. It uses **randomNumber** to pick numbers for the **x** and **y** coordinates.

```
circle(random(800), randomNumber(600), 25);
circle(random(800), randomNumber(600), 25);
circle(random(800), randomNumber(600), 25);
```

Run the code to check it works.



That code didn't do what we wanted. Can you see why?

It doesn’t work because **all** of the circles are drawn at random locations. Our earlier code didn’t do that.

Look again at that earlier code:

```
circle(400, 300, 25);
circle(400, 350, 25);
circle(400, 400, 25);
```

We can see that the x-coordinates are not totally random. They are kept the same for all the circles.

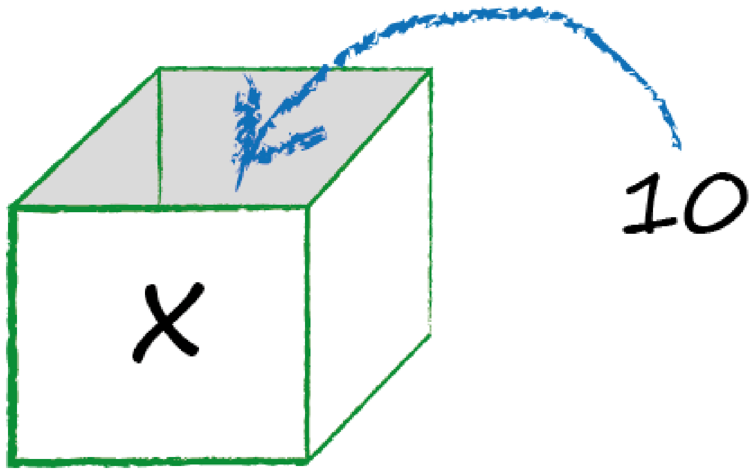
We can also see that the y-coordinates are not totally random. They increase in steps of **50** from the first circle.

That means we need to pick a random location only for the first circle.

We also need to **remember** that location so we can draw the other two circles next to it.

We can ask our computer to remember a number by using a **variable**.

We’ll talk about **variables** next.



Variables are like boxes that we can put numbers in. The picture above shows a **variable** called **x**. You can see we're putting the number **10** inside it.

Whenever we use **x** in our code, our computer will look inside the box and use the number **10**.

Have a look at this new code. Can you work out what is does?

```
var x = randomNumber(800);
var y = randomNumber(600);

circle(x, y, 25);
circle(x, y + 50, 25);
circle(x, y + 100, 25);
```

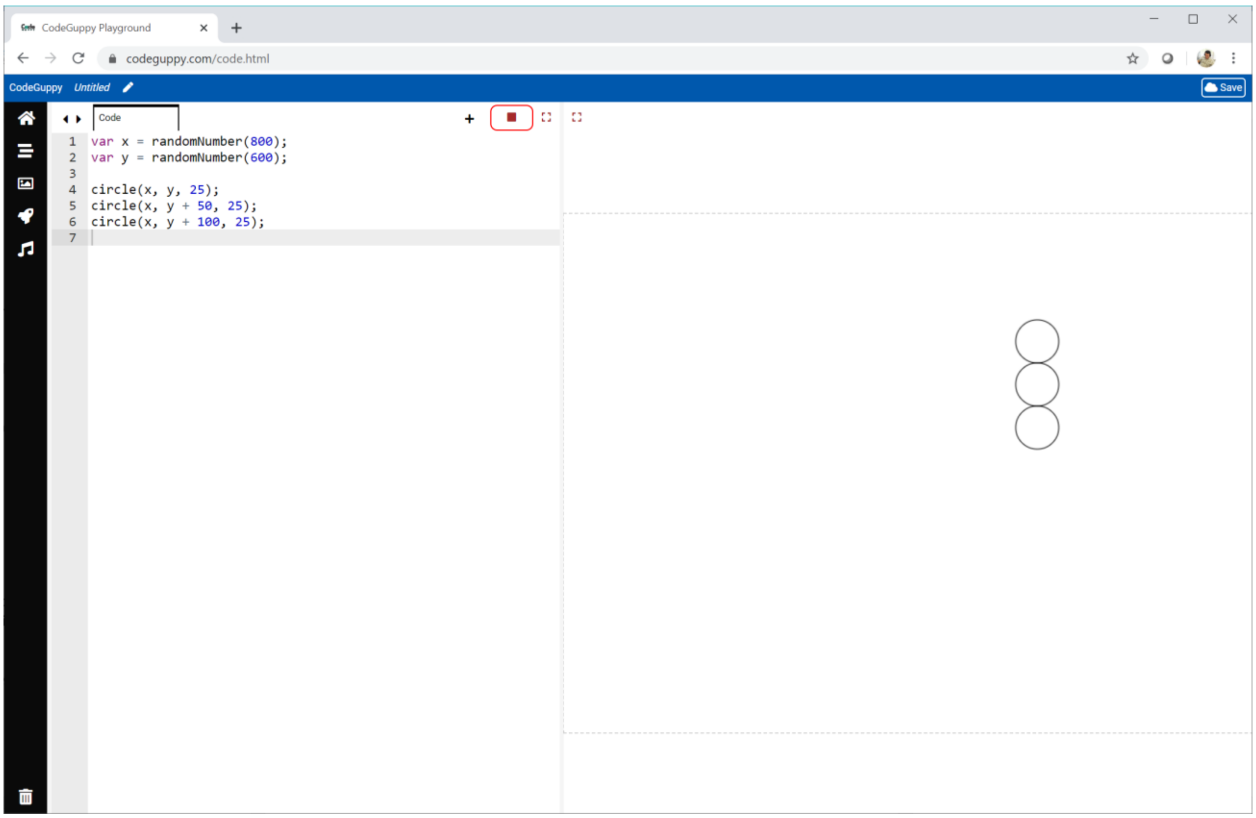
We're creating a new variable **x** and putting a random number between **0** and **800** inside it. We can't guess what number it will be. We only know it will be between **0** and **800**.

We're creating another variable called **y**, and putting a random number between **0** and **600** inside it.

What do you think the first circle command **circle(x, y, 25)** will do?

What do you think the second circle command **circle(x, y + 50, 25)** will do?

Try the code yourself, and see what happens.



You should get a chain of three circles, somewhere on the canvas.

The first **circle(x, y, 25)** instruction uses the random numbers that were put inside the **x** and **y** variables.

The next **circle(x, y + 50, 25)** instruction also uses the same numbers that were put inside **x** and **y**.

Run the code again.

The chain of circles will be at a different place. That’s because the location of the first circle is chosen at random.

Well done for getting this far!

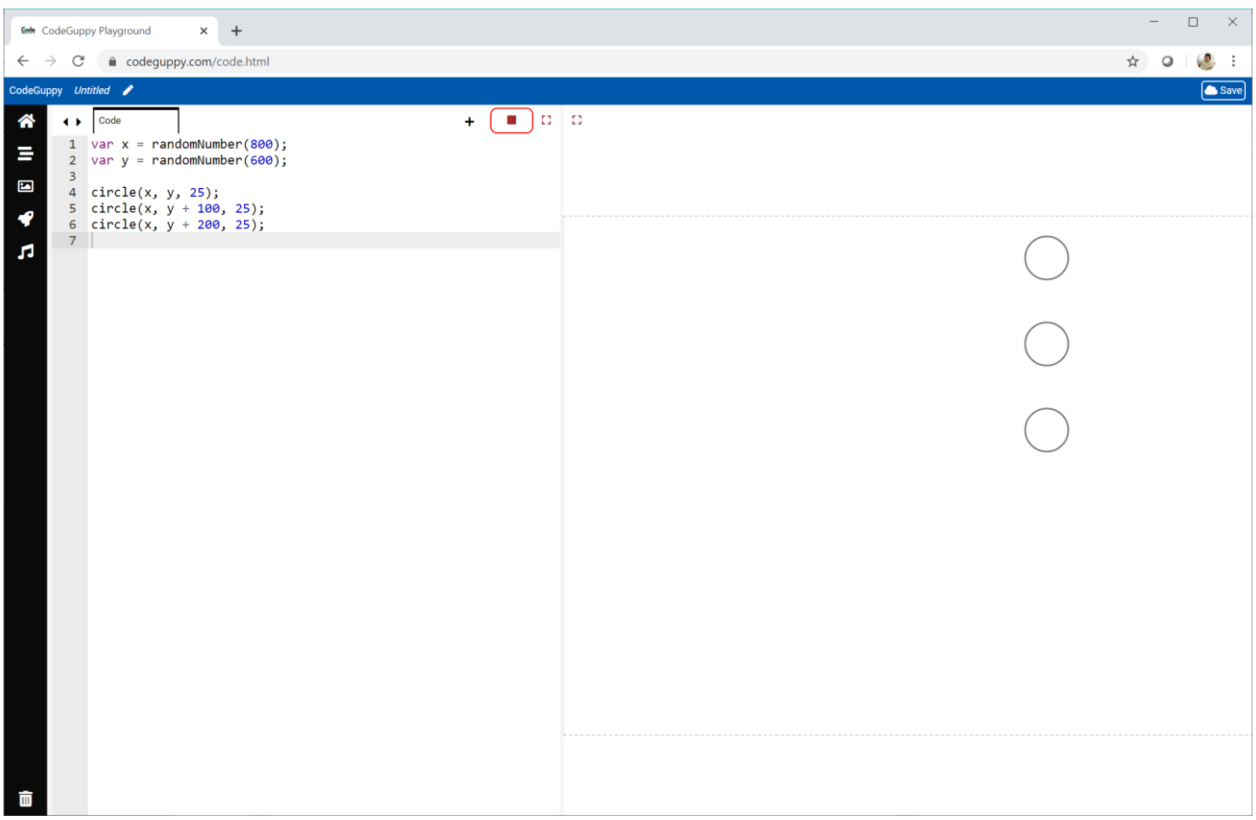
Try It Yourself

Try changing the **circle** instructions to add different numbers to the variables.

Here I’ve changed the second circle to use **y + 100**. The third circle uses **y + 200**.

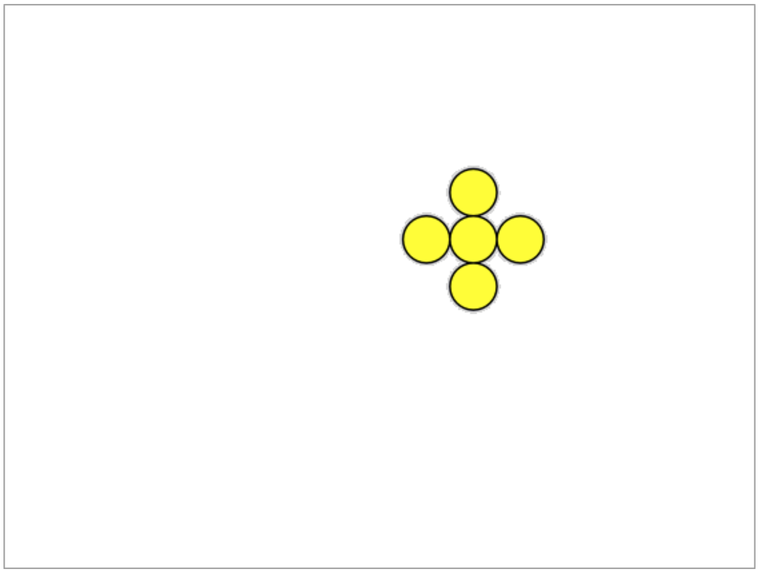
```
circle(x, y, 25);
circle(x, y + 100, 25);
circle(x, y + 200, 25);
```

Here’s what my changed code draws.



Challenge!

Have a look at this drawing.



There are five circles arranged like flower. There is one circle in the middle, and four circles around it.

Can you write code that draws that flower pattern at a randomly chosen place on the canvas?

You can use a pen and paper to help plan your code.

Some of the best coders use a pen and paper to plan their code!

Level 2 - Progressing

2.1 - Simple Functions

level

What We'll Do

In this project we're going to:

- learn how to package useful code as a **function**
- see how functions can be **useful**

Start a New Program

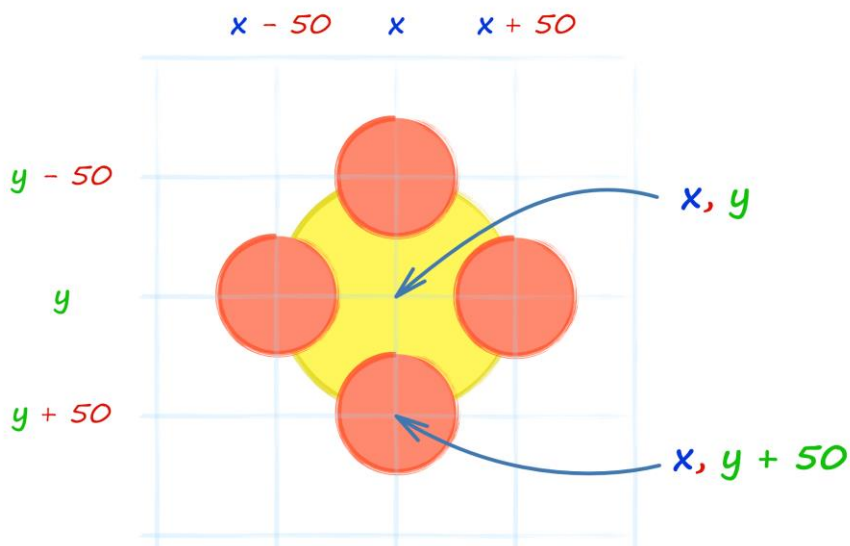
Log in to codeguppy.com if you haven't already.

Create a new program by clicking the big "CODE NOW" button.

A Flower Made Of Circles

In the last project we learned to use **variables** to draw a group of shapes at any location on the canvas.

Have a look at this picture showing a flower made of circles.



Where is the centre of the yellow circle? You can see from the picture it is at **(x, y)**. We're using letters instead of numbers.

Where is the bottom red circle? It is a bit further down from the yellow circle. Looking at the picture, you can see it is at **(x, y + 50)**.

If the centre of the yellow circle is at **(x, y)** then we can work out the centres of all the red circles:

- the top red circle is at **(x, y - 50)**
- the bottom red circle is at **(x, y + 50)**
- the right red circle is at **(x + 50, y)**
- the left red circle is at **(x - 50, y)**

If we set **x** to **100** and **y** to **100**, then the flower should be drawn near the top left of the canvas, just like before.

If we set **x** to **400** and **y** to **300**, then the flower should be drawn in the middle of the canvas.

We can set **x** and **y** to any location on the canvas, and the flower will be drawn there.

Here is some code to draw that flower at **(x, y)**.

```
var x = 400;
var y = 300;

fill('yellow');
circle(x, y, 50);

fill('red');
circle(x, y - 50, 25);
circle(x + 50, y, 25);
circle(x, y + 50, 25);
circle(x - 50, y, 25);
```

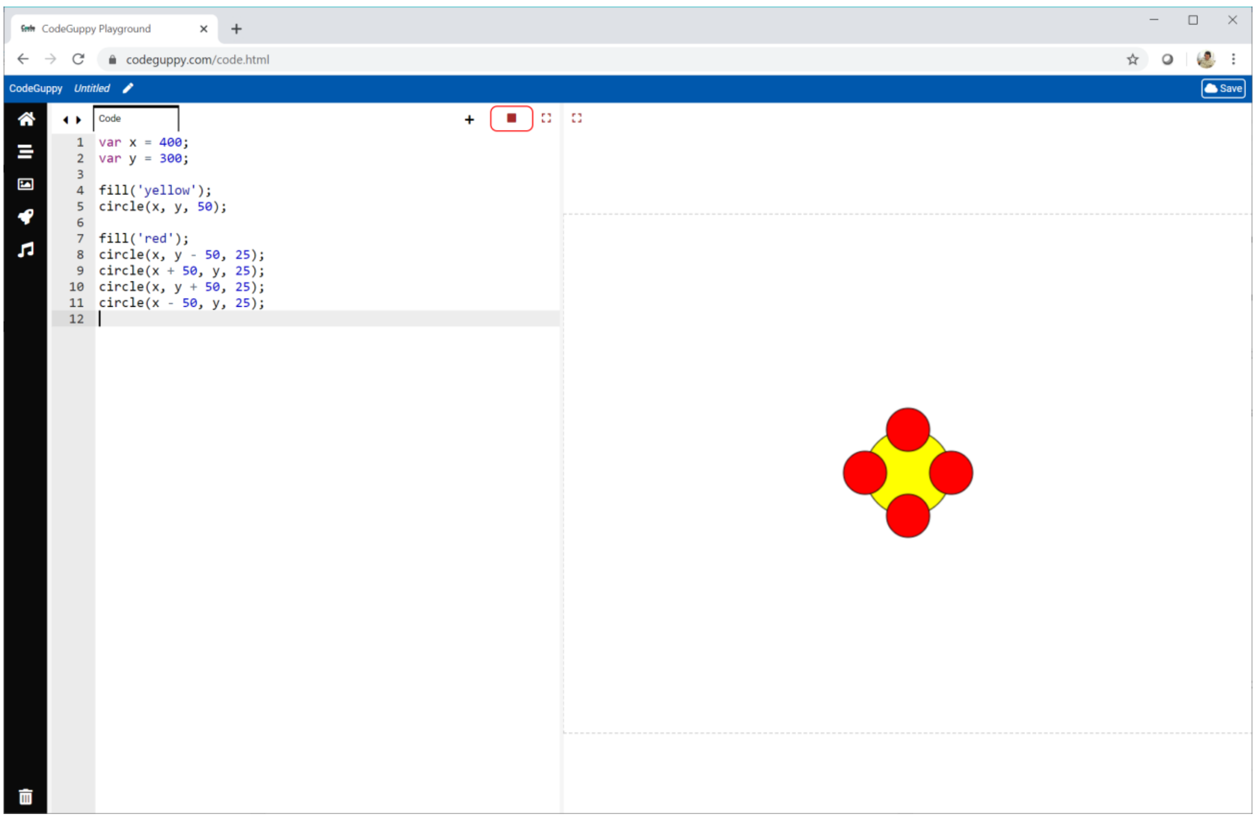
At the beginning of the code we set **x** to **400**, and **y** to **300**.

You can see the instructions for the yellow and red circles use **x** and **y** instead of numbers.

When our computer sees the **x** in **circle(x, y, 50)** it will take the number inside the variable **x** and use that. The same will happen for **y**.

So **circle(x, y, 50)** will become **circle(400, 300, 50)**.

Run the code to check it draws the flower in the middle of the canvas at (**400,300**).



Great - that worked!

Draw The Flower At A Random Place

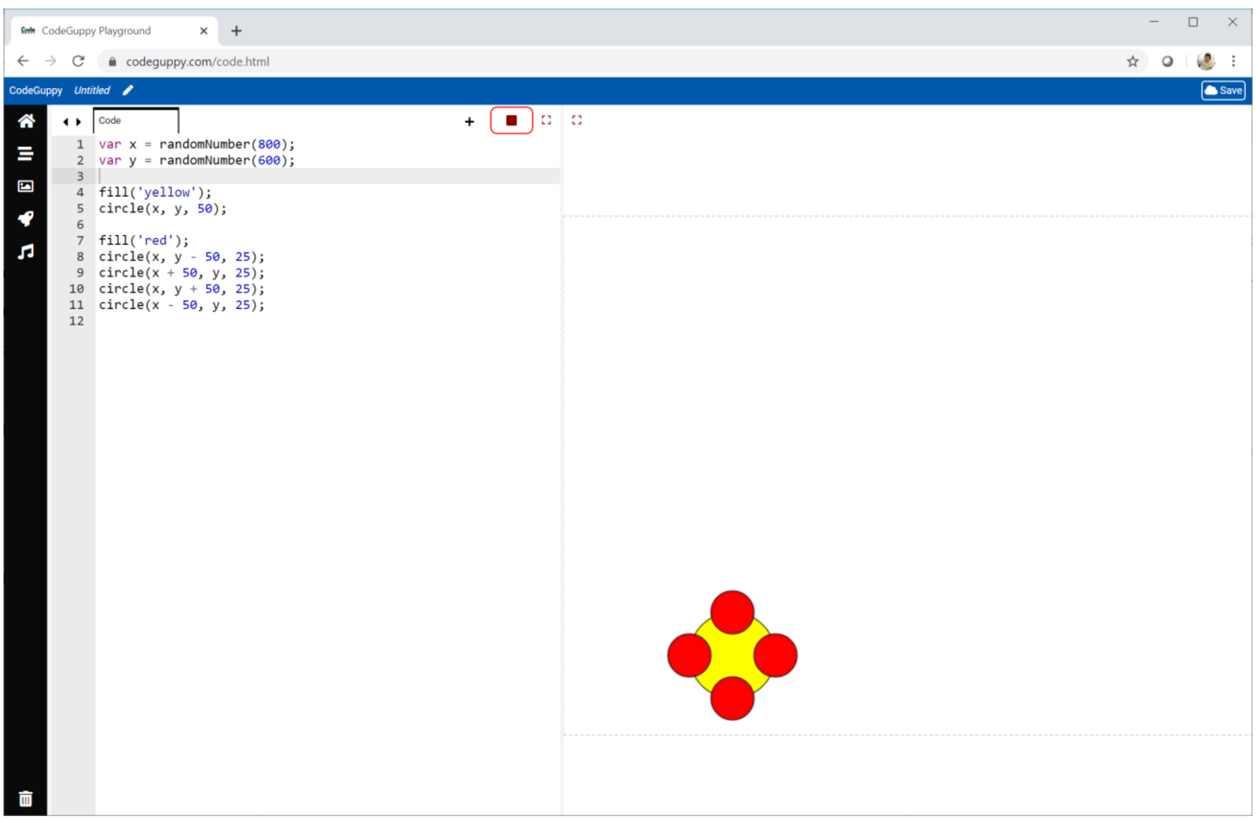
Let’s change the code to draw the flower at a random place on the canvas.

We only need to change the numbers that **x** and **y** are set to.

Instead of us choosing a number, we let our computer pick one for us.

```
var x = randomNumber(800);
var y = randomNumber(600);
```

Run the code to check the flower is drawn somewhere else.



Your drawing will be different because your computer will choose a different random location.

Draw Lots of Flowers

Let’s do something new and exciting - let’s draw **5** flowers!

How do we draw **5** flowers? We could write all that code again **5** times, but that would be really long and boring.

It would be better if we taught our computer to draw a flower just once, and then asked it to draw a flower lots of times.

It would be like a recipe for chocolate cake. We write it down once, and use it lots of times.

We can write recipes in code too. They’re called **functions**.

Have a look at this code.

```
function my_flower()
{

  var x = randomNumber(800);
  var y = randomNumber(600);

  fill('yellow');
  circle(x, y, 50);

  fill('red');
  circle(x, y - 50, 25);
  circle(x + 50, y, 25);
  circle(x, y + 50, 25);
  circle(x - 50, y, 25);

}
```

If you look carefully, you'll see it is the same code we already wrote to draw a flower at a random place on the canvas.

The only difference is that we have

function my_flower()
{

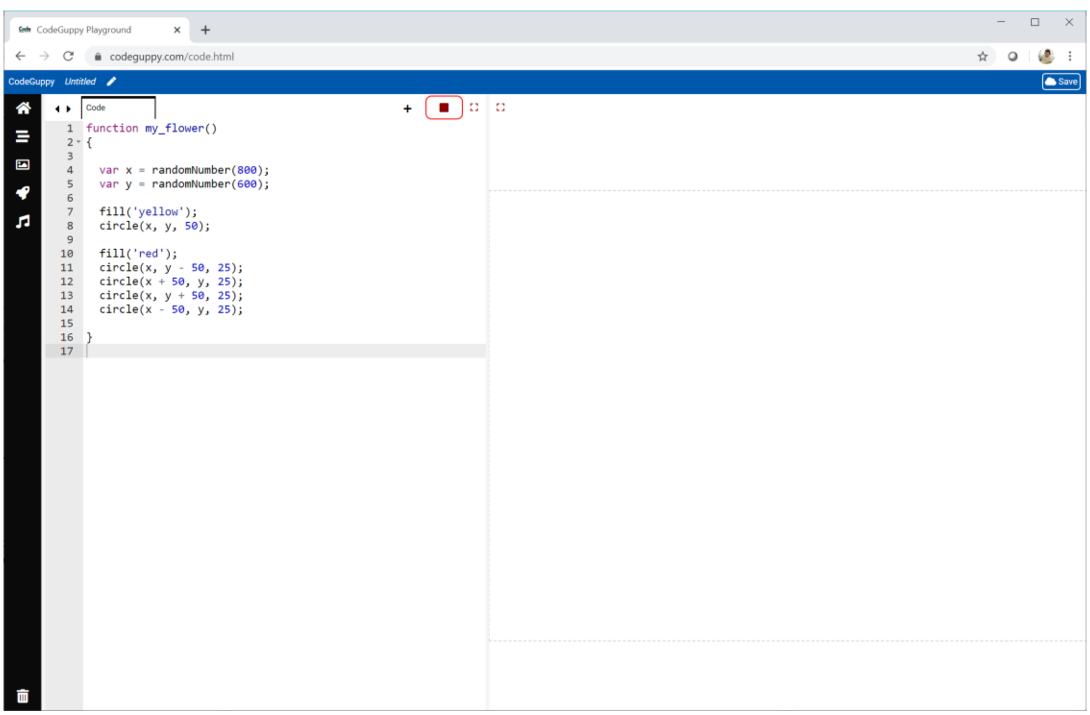
at the top, and

} at the bottom.

What this new code does is create a recipe, or **function**, called **my_flower**. The recipe instructions are inside the curly brackets **{** and **}**.

Write this code to create the **my_flower** function.

If you run this code you will get an empty canvas.



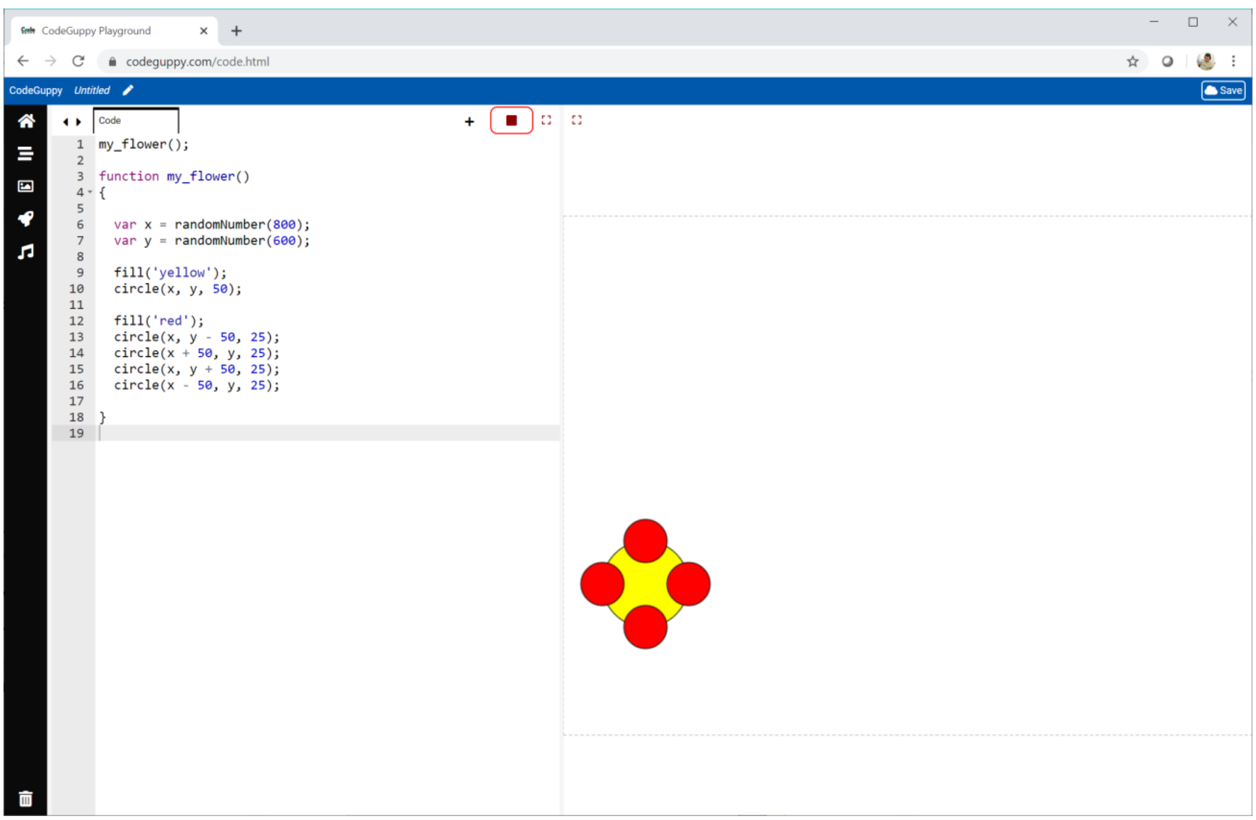
That's because we've created the **my_flower** recipe, but we haven't used it yet.

To use it, we simply write the name of the function **my_flower** outside of the function.

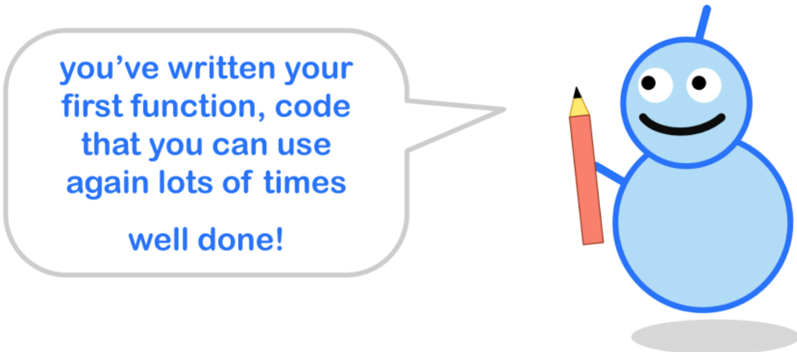
```
my_flower();
```

Don't forget the empty round brackets **()** after the name of the function.

Run the code to check that using our new **my_flower** instruction really does work.



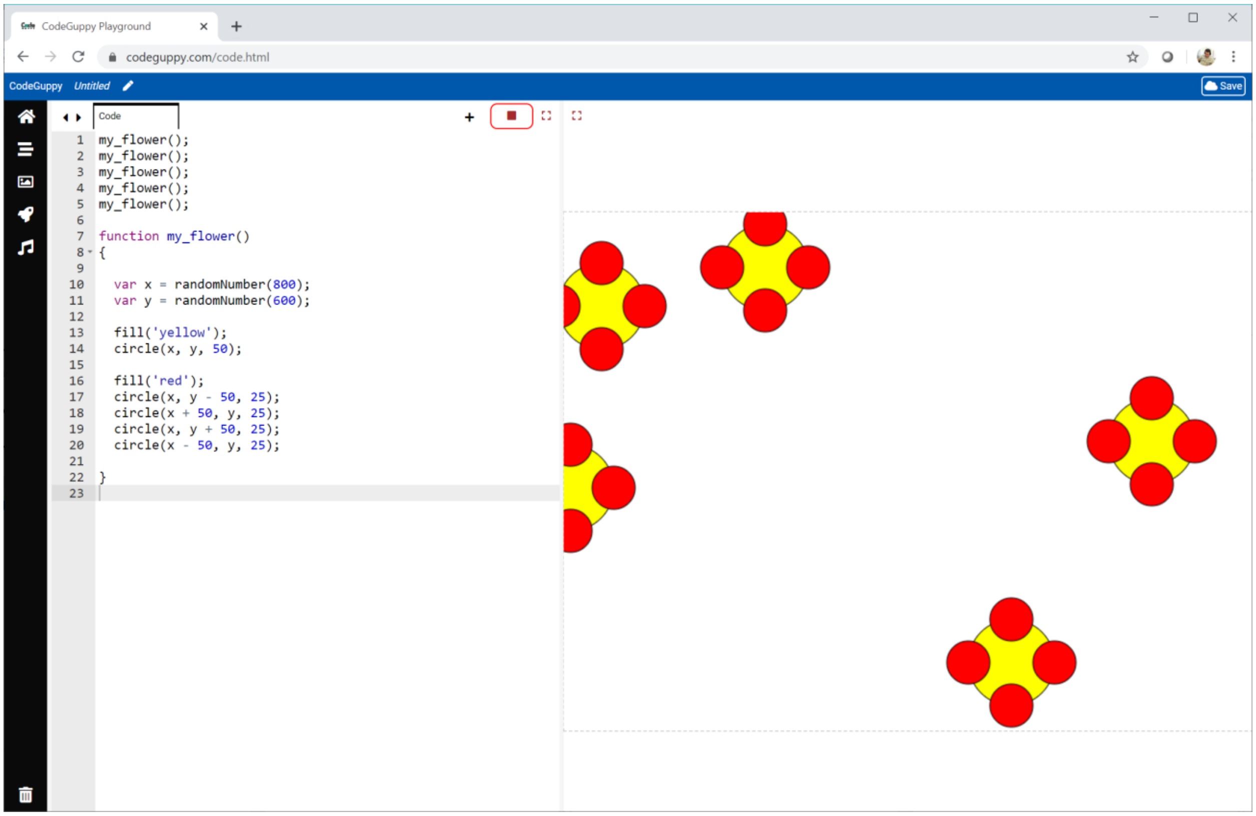
My flower is falling off the edge of the canvas. Yours will be somewhere else on the canvas.



What do you think will happen if we write five **my_flower** instructions one after the other?

```
my_flower();
my_flower();
my_flower();
my_flower();
my_flower();
```

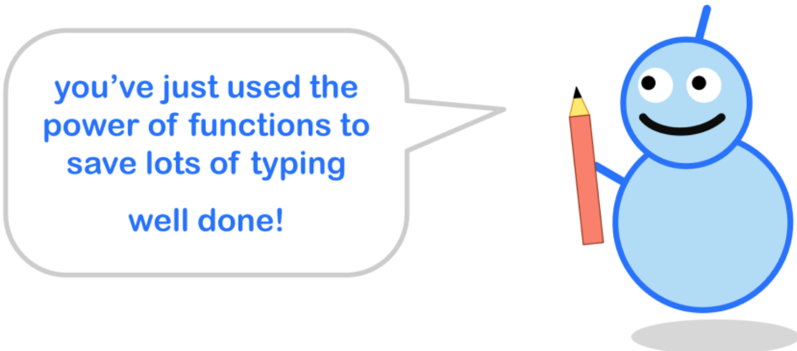
Try it.



How cool is that!

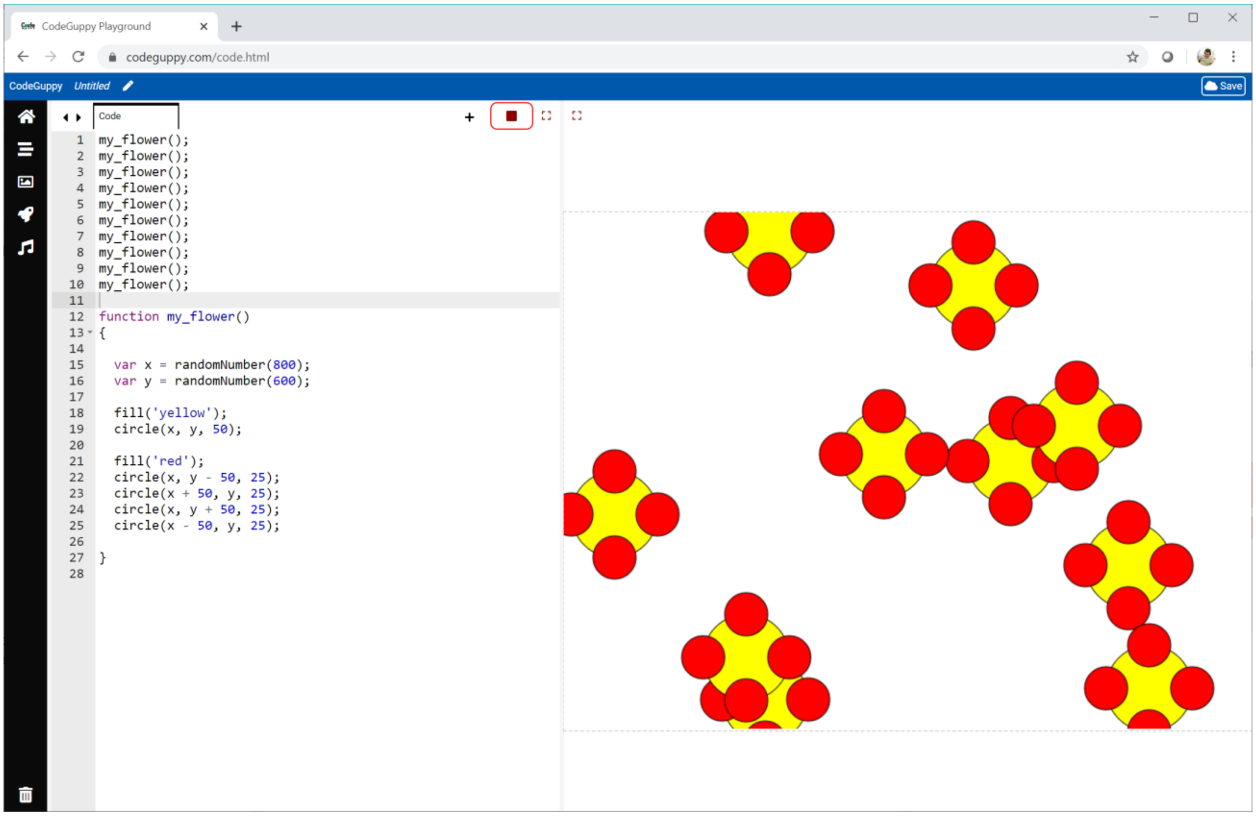
Our code has drawn **5** flowers, just like we wanted.

The really cool thing is that we didn't need to write out all the **circle** instructions lots of times. There are **25** circles in that drawing so we saved a lot of typing!


you've just used the power of functions to save lots of typing
well done!

Try it Yourself

Try changing your code to draw **10** flowers.



It's getting busy!

That drawing has **50** circles. We've saved a lot of typing by using a **function** to draw each flower.

Challenge!

Change the **my_flower** function to draw flowers with a petal colour chosen at random from a list of colours.

2.2 - Repeating Things

level

What We'll Do

In this project we're going to:

- learn how to **repeat** instructions
- see how repetition can do lots of work without typing lots of code

Start a New Program

Log in to codeguppy.com if you haven't already.

Create a new program by clicking on the main "CODE NOW" button.

Start With A Simple Function

Let's start this project with a really simple function that draws a small circle filled with a colour chosen randomly from a list.

Have a look at this code. We'll talk about it below.

```
function bubble()
{
  var x = randomNumber(200, 600);
  var y = randomNumber(200, 400);
  var r = randomNumber(15, 50);
  
  fill( random(['pink', 'yellow', 'lightgreen']) );

  circle(x, y, r);
  
}
```

We’ve called the function **bubble**, because the coloured circles should look like bubbles.

You can see we’ve picked random numbers for **x** and **y**. These will be the location of the circle.

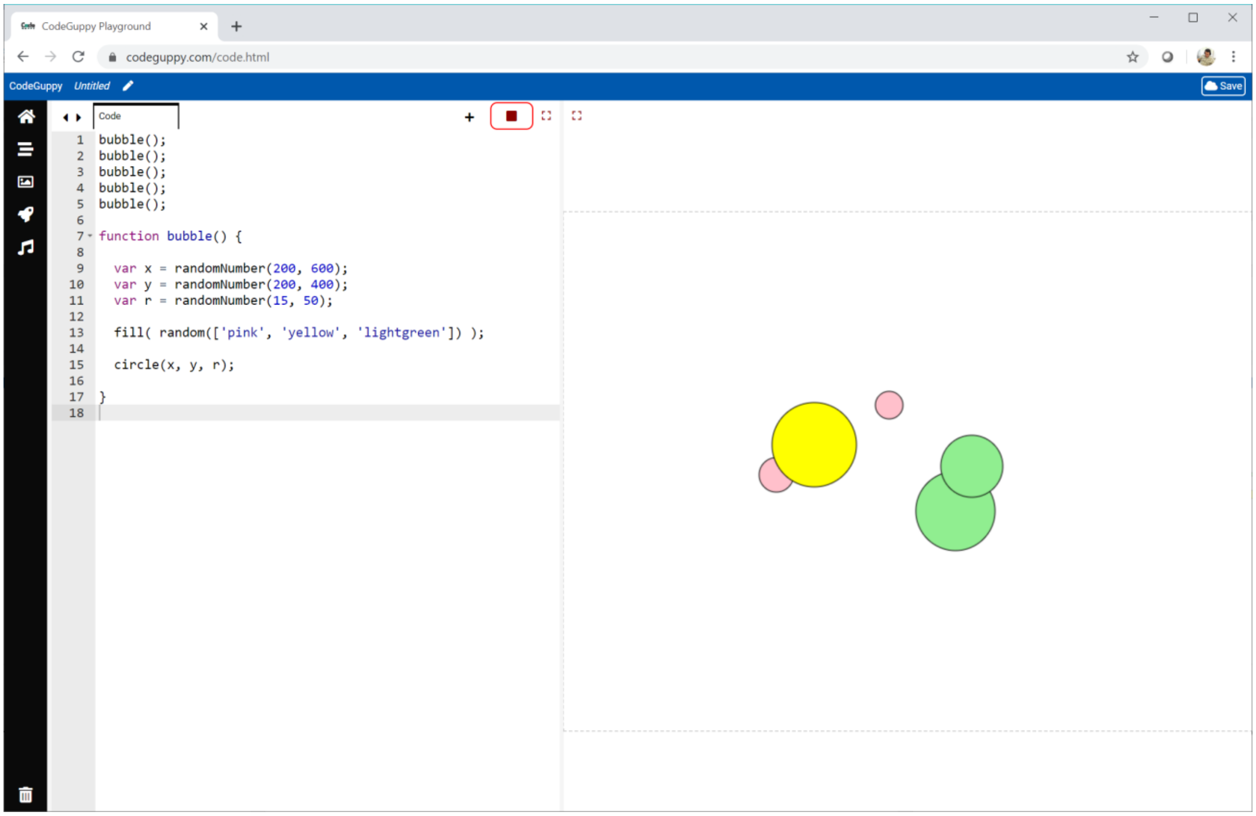
Did you spot that **randomNumber** instruction has **two** numbers inside its brackets? Before it only had one. The instruction now picks a random number that is between those two numbers. So **randomNumber(200, 600)** picks a number between **200** and **600**.

You can also see that **r** is set to a random number between **15** and **50**. This will be the radius of the circle.

We’re also choosing a random colour from a list of **pink**, **yellow** and **lightgreen**.

Let’s call this **bubble** function five times. **Calling** a function is what coders say when they’re **using** a function.

Run the code to see five coloured bubbles.



Those bubbles look like sweets!

We haven't don't anything very new yet. That's next.

Draw 20 Bubbles

That drawing needs more bubbles. Let’s draw **20**.

We could repeat the **bubble** instruction 20 times, but that would get tiring. There must be a better way.

There is a better way!

Computers are really good at repeating things, and they don’t get bored.

Have a look at this new code:

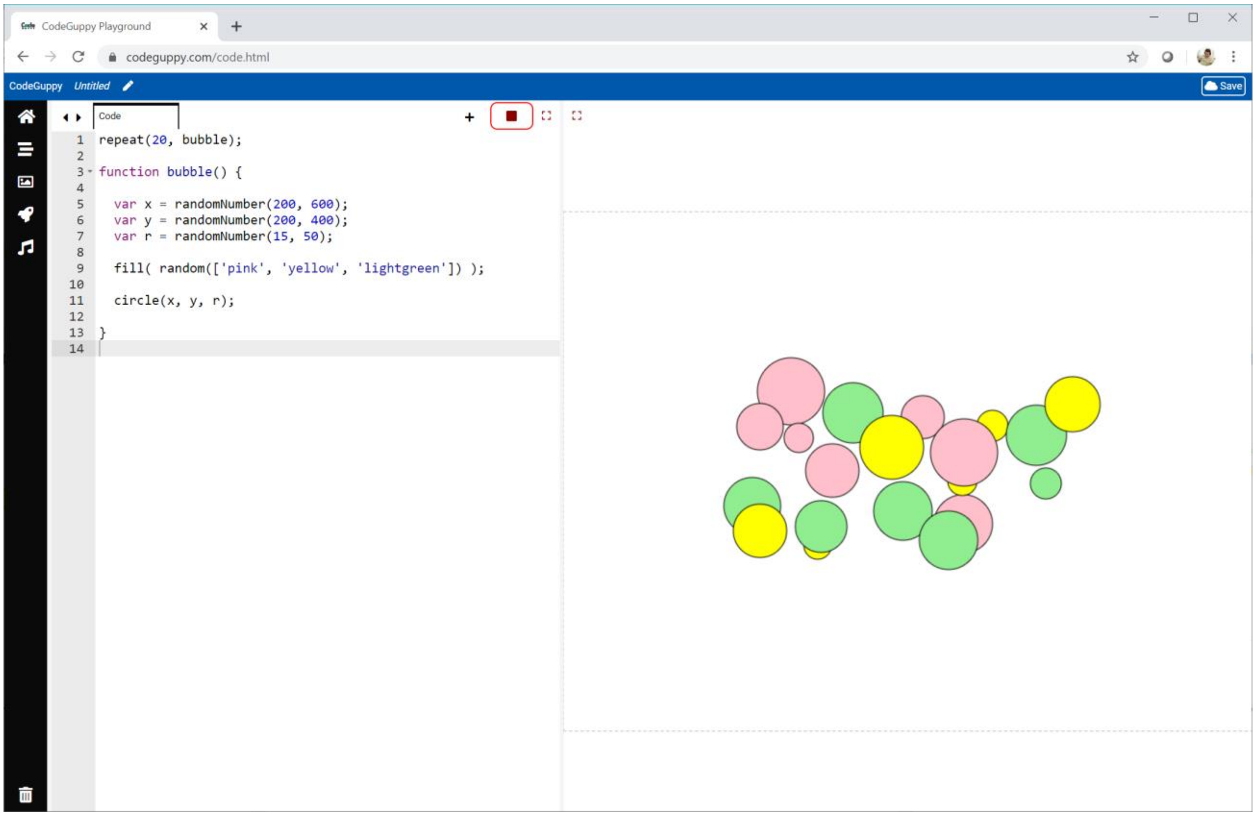
```
repeat(20, bubble);
```

Can you guess what it does?

The **repeat** instruction repeats a function. We tell it what function to repeat. Here, we’ve told it to repeat the **bubble** function.

That number **20**, tells **repeat** to call the **bubble** function **20** times.

Replace your five calls to **bubble** with this single **repeat** instruction.



Very cool!

That repeat instruction is really simple, but you can see it is powerful.

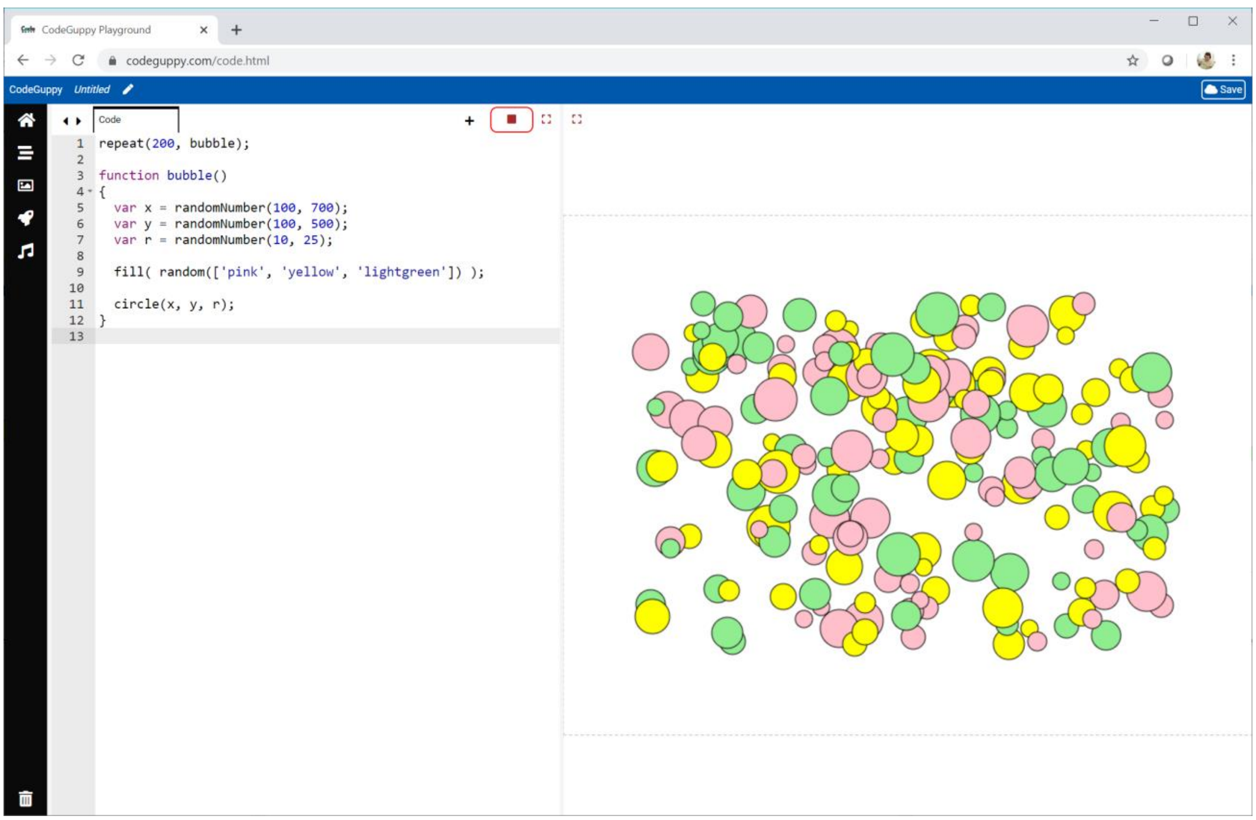
Try It Yourself

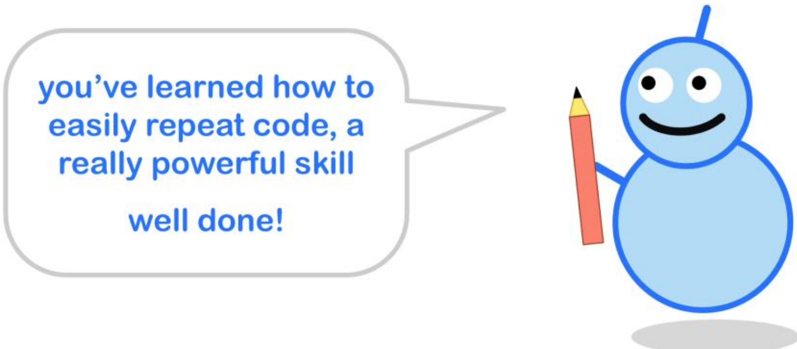
Change your code to draw **200** bubbles. Yes, **200**!

I changed the bubble function to make the bubbles smaller. I also increased the range of numbers that **x** and **y** are chosen from, so more of the canvas is used for drawing on.

```
var x = randomNumber(100, 700);
var y = randomNumber(100, 500);
var size = randomNumber(10, 25);
```

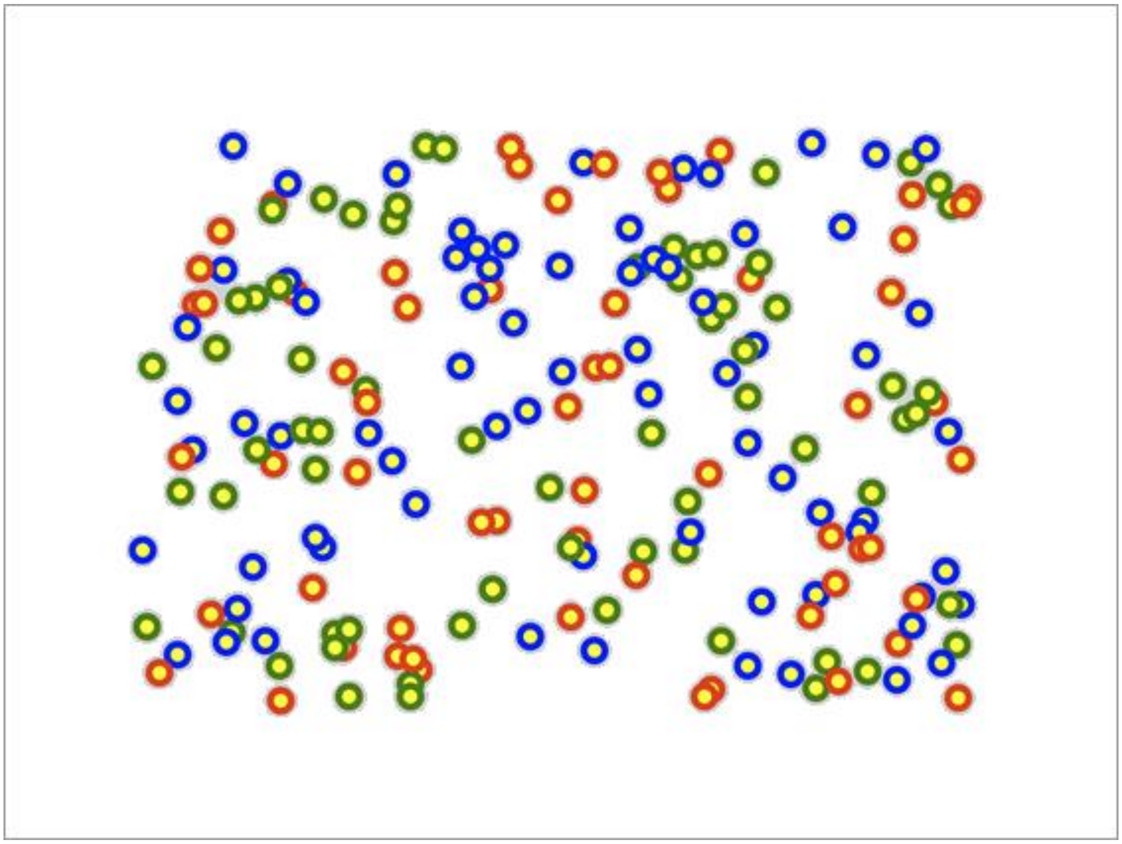
Here's what my code draws.




you've learned how to easily repeat code, a really powerful skill
well done!

Challenge!

Have a look at this drawing.



If you look closely you can see it is made of lots of small blobs.

There are **200** blobs, but you don't have to count them!

Each blob has a small **yellow** circle on top of a **red**, **green** or **blue** larger circle.

Can you write code to make a similar drawing?

2.3 - More Functions

level

What We'll Do

In this project we're going to:

- learn how to pass information to **functions**
- see how this makes functions even more **useful**

Start a New Program

Log in to codeguppy.com if you haven't already.

Create a new program by clicking on the big "CODE NOW" button.

Drawing Flowers Where We Want

We've already learned how to write a **function** - code we can use again and again.

Here is the function **my_flower** we wrote to draw a flower.

```
function my_flower()
{
  var x = randomNumber(800);
  var y = randomNumber(600);

  fill('yellow');
  circle(x, y, 500);

  fill('red');
  circle(x, y - 50, 25);
  circle(x + 50, y, 25);
  circle(x, y + 50, 25);
  circle(x - 50, y, 25);
}
```

We don't know where it will draw a flower. That's because the centre of the flower **(x, y)** is chosen at random.

It would be good if we could tell our function exactly where to draw the flower.

That means we need a way to tell our **my_flower** function what **x** and **y** should be.

Have a look at this new code. It is the same as before but with some small changes. Can you spot the differences?

```
function my_flower(x,y)
{
  fill('yellow');
  circle(x, y, 50);

  fill('red');
  circle(x, y - 50, 25);
  circle(x + 50, y, 25);
  circle(x, y + 50, 25);
  circle(x - 50, y, 25);
}
```

There are two differences:

- the name of the function now has **(x, y)** in brackets.
- we removed the code to choose **x** and **y** at random.

Changing the name from **my_flower()** to **my_flower(x, y)** means the function now needs to be told what **x** and **y** to use.

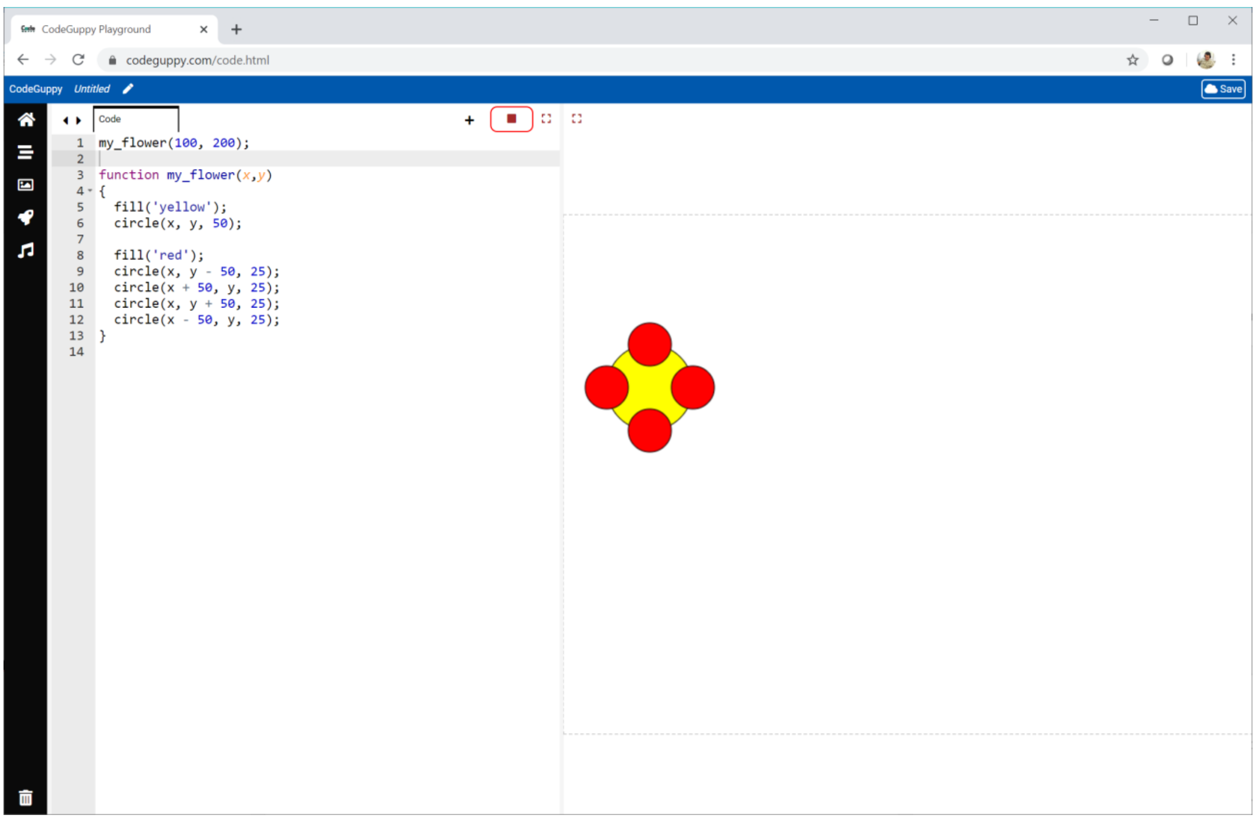
Here is how we tell **my_flower** what we want **x** and **y** to be:

```
my_flower(100, 200);
```

This calls **my_flower**, just like before, but passes the number **100** to the function to be used as **x**. It also passes **200** to be used as **y**.

Let's try it!

Here is what my code looks like and the result, a flower drawn at **(100, 200)**.



Great!

We've told the function where to draw the flower by passing information to it. This is called passing **parameters**.

The word **parameters** means the information you give to a function when you use it.

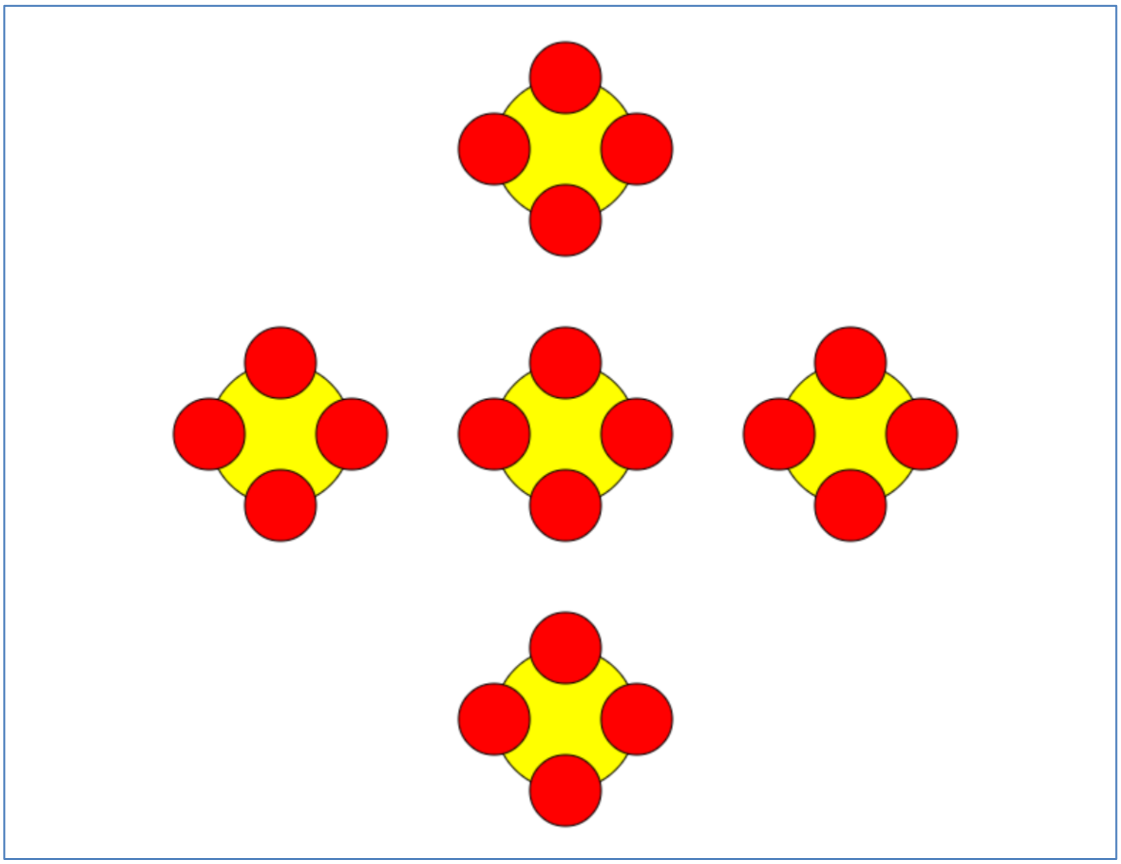
We can say the function **my_flower(x, y)** takes **2 parameters**, **x** and **y**.

Try It Yourself

Try your new **my_flower(x, y)** function with different **parameters**.

Try drawing several flowers with different parameters for each one. See if you can make a pattern of flowers.

Here is a pattern I made.



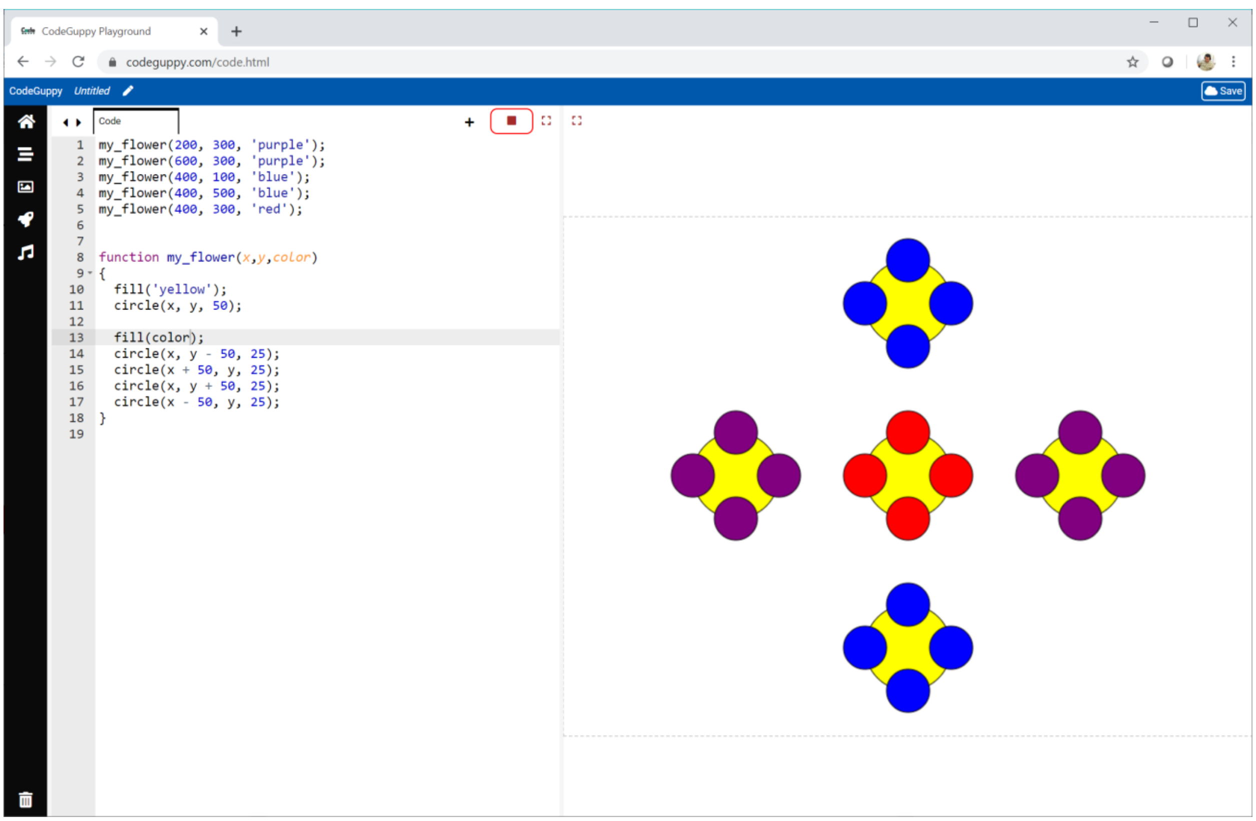
Now see if you can change the code so that your **my_flower** function takes an extra parameter, a colour. Use this colour for the flower petals.

Test your new **my_flower** function by drawing different coloured flowers.

Here's my code, showing the extra colour parameter passed to **my_flower**:

```
my_flower(200, 300, 'purple');
my_flower(600, 300, 'purple');
my_flower(400, 100, 'blue');
my_flower(400, 500, 'blue');
my_flower(400, 300, 'red');
```

And here is the pattern it makes.




you can now pass information to functions, making them even more useful
well done!

Challenge!

Change the **my_flower** function so that it takes a **scale** parameter. Use this **scale** number to draw smaller or larger flowers.

This code shows **my_flower** being used with a **scale** parameter of **2**.

```
my_flower(200, 300, 'purple', 2);
```

Let a **scale** of **1** draw flowers with a middle circle of size **50** and petals of size **25**, just like before.

A **scale** of **0.5** would draw a smaller flower half the size of the original. It would have a middle circle of radius **25** and petals of radius **12.5.**

A scale of **2** would draw a larger flower, **twice** as large as the original.

2.4 - Mixing Colours

level 

What We'll Do

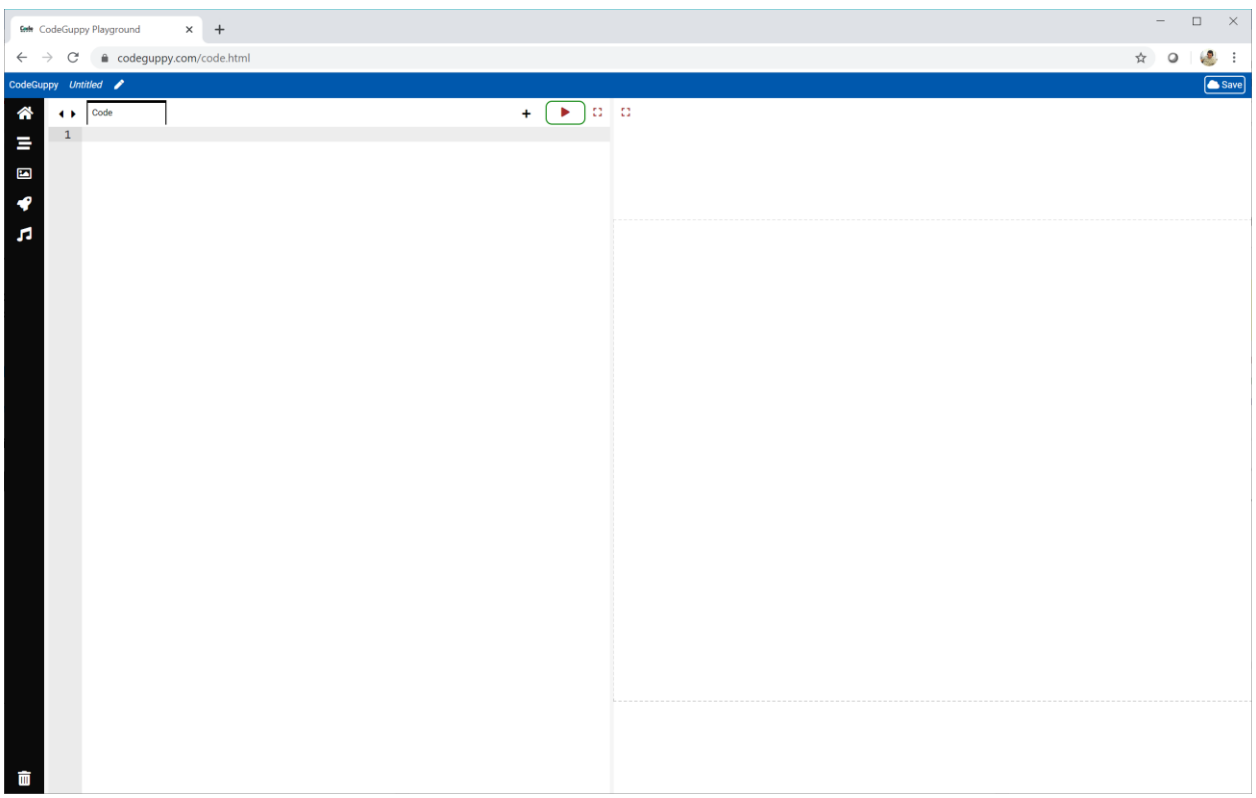
In this project we're going to:

- learn how to **mix** our own **colours**
- see how we can **calculate** colours

Start a New Program

Log in to codeguppy.com if you haven't already and create a new program.

Your code window should look like this:

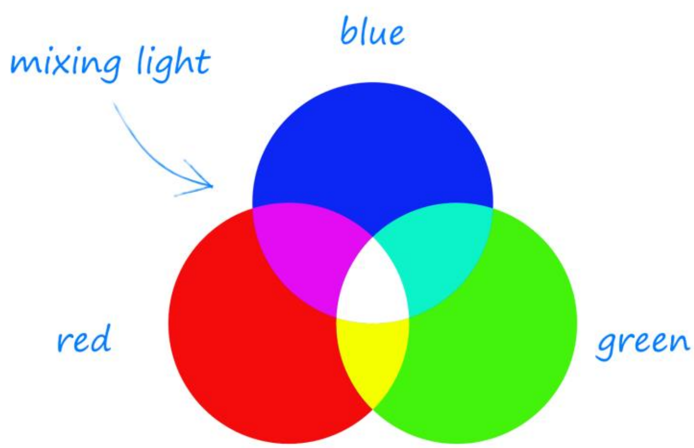


Mixing Colours

We've already chosen colours by using names like **pink** and **orange**.

We will now learn to mix our own colours. This is just like mixing paint to get the colour we want.

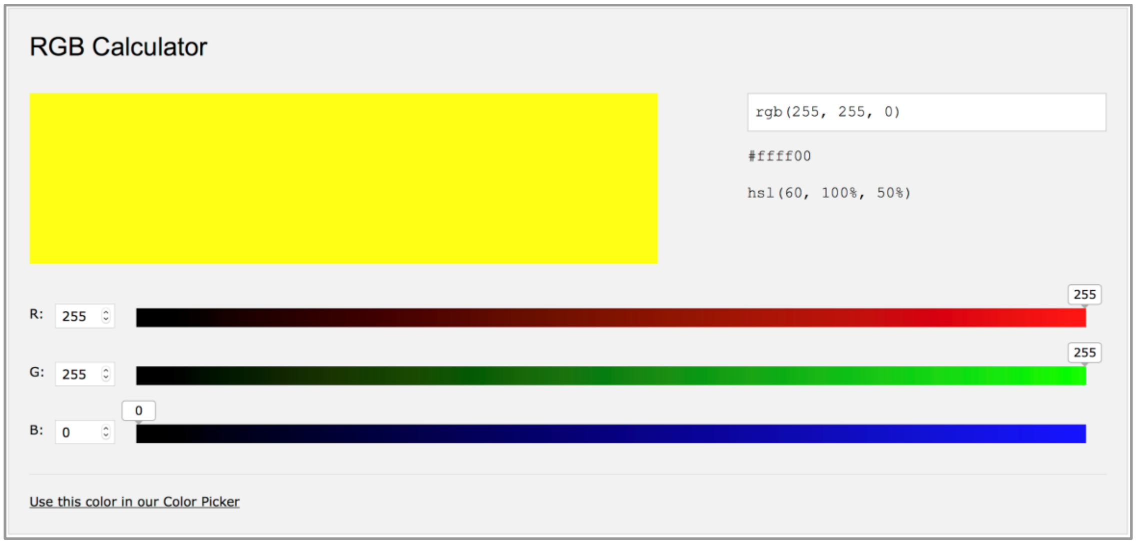
Mixing **red**, **green** and **blue** light is how we see colours on our television, laptop and smartphone screens.



You might have mixed **red**, **green** and **blue** to choose a colour in your favourite photo editing software.

Try the mixer at https://www.w3schools.com/colors/colors_rgb.asp

You can make **yellow** by mixing **red** and **green**.



See if you can make **orange** and **pink**.

Look again at the mixer above. You can see the **red**, **green** and **blue** levels go from **0** all the way up to **255**.

That **yellow** is made of:

- **red 255**
- **green 255**
- **blue 0**

The **red** and **green** are turned up full, and the **blue** is turned off completely.

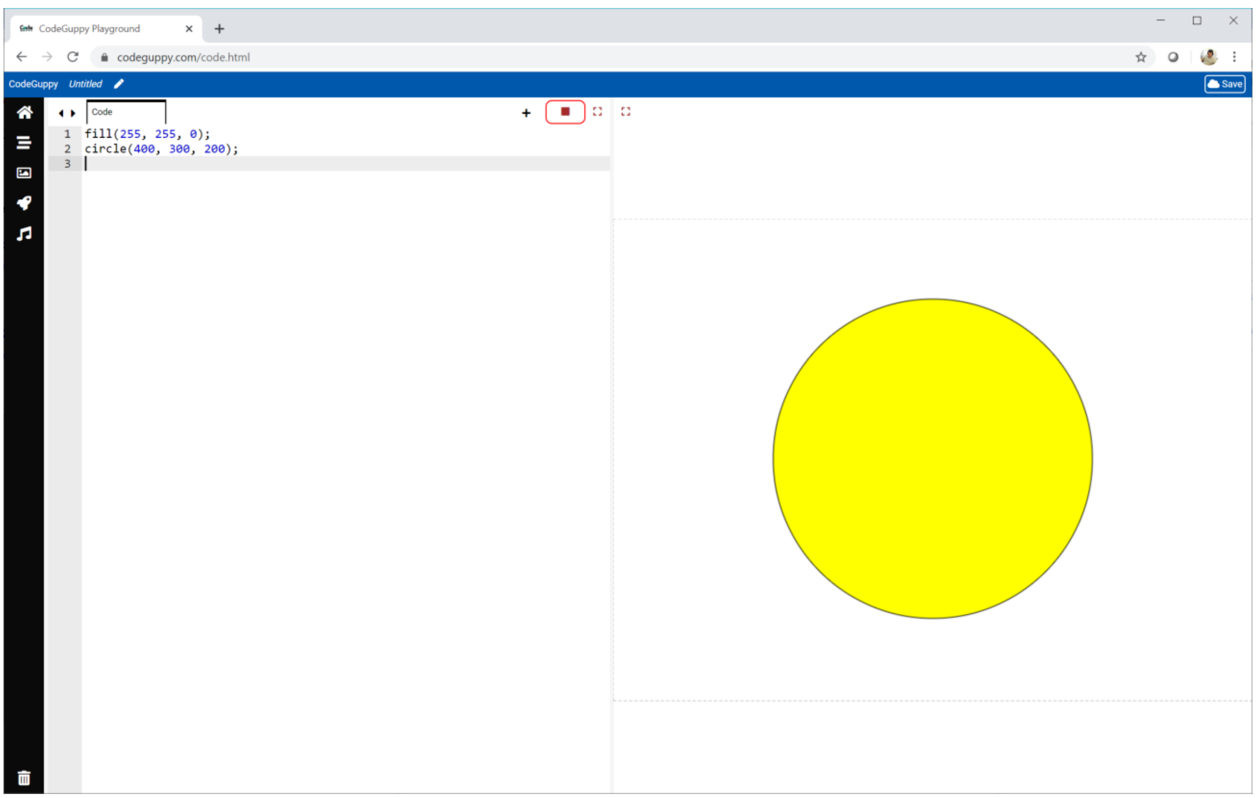
We know this code will draw a **yellow** circle.

```
fill('yellow');
circle(400, 300, 200);
```

Let’s pick that same **yellow** using the **red**, **green** and **blue** levels.

```
fill(255, 255, 0);
circle(400, 300, 200);
```

Running this code gives us a **yellow** circle again.

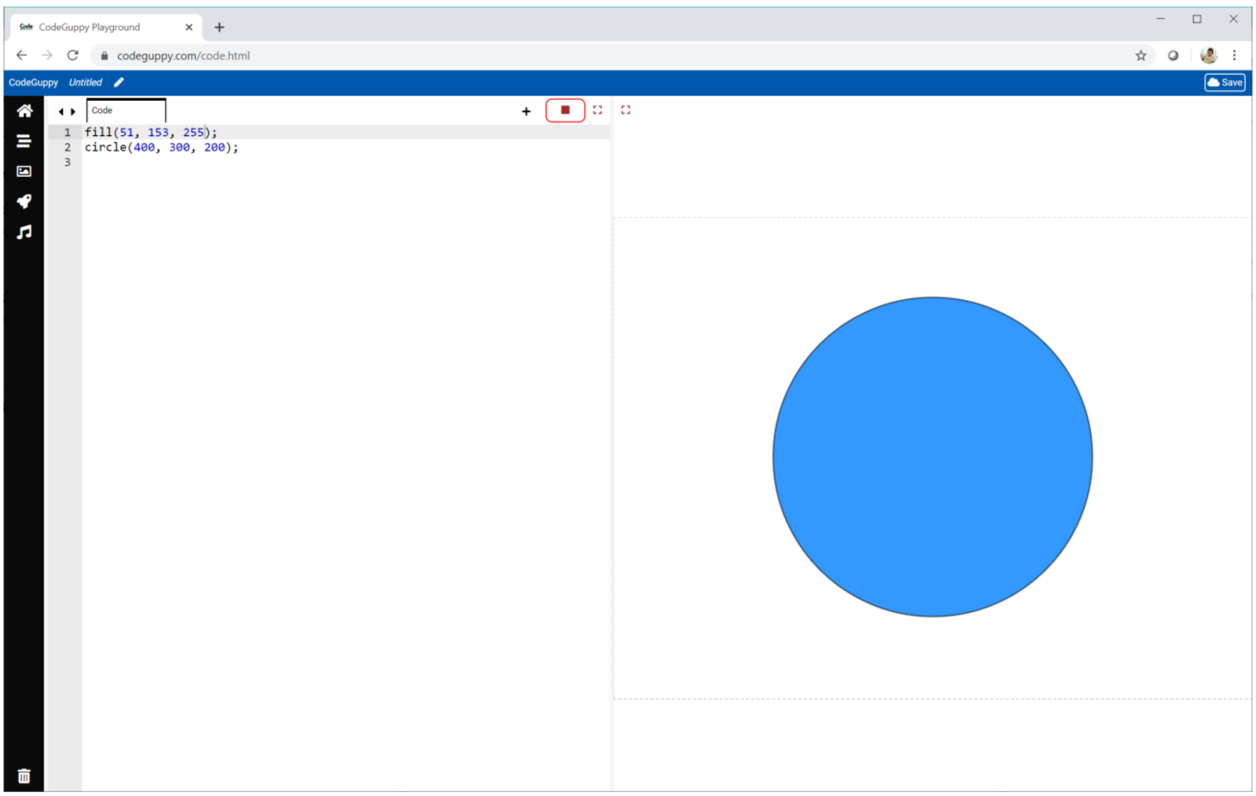


Try It Yourself

Try mixing different levels of **red**, **green** and **blue** to change the colour of the circle.

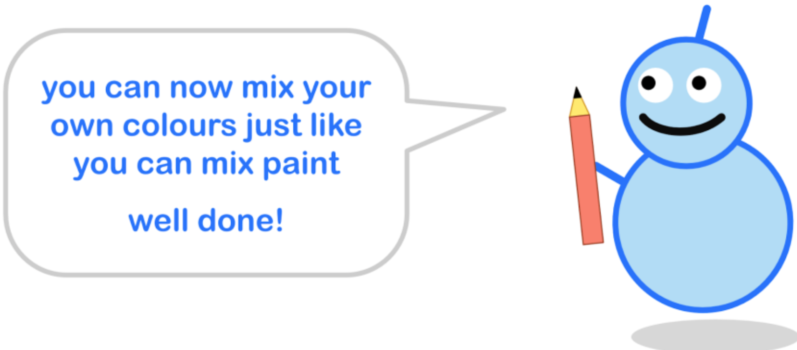
Remember to keep the levels between **0** and **255**.

Here is a nice **blue** I found.



The **RGB** levels for this blue are **(51, 153, 255)**.

RGB is short for **red**, **green** and **blue**.


you can now mix your
own colours just like
you can mix paint
well done!

Mixing Random Colours

The **RGB** levels for any colour are numbers from **0** up to **255**.

What if we chose those numbers at **random**?

That would mean mixing random amounts of **red**, **green** and **blue** light to make a colour.

Let’s try it.

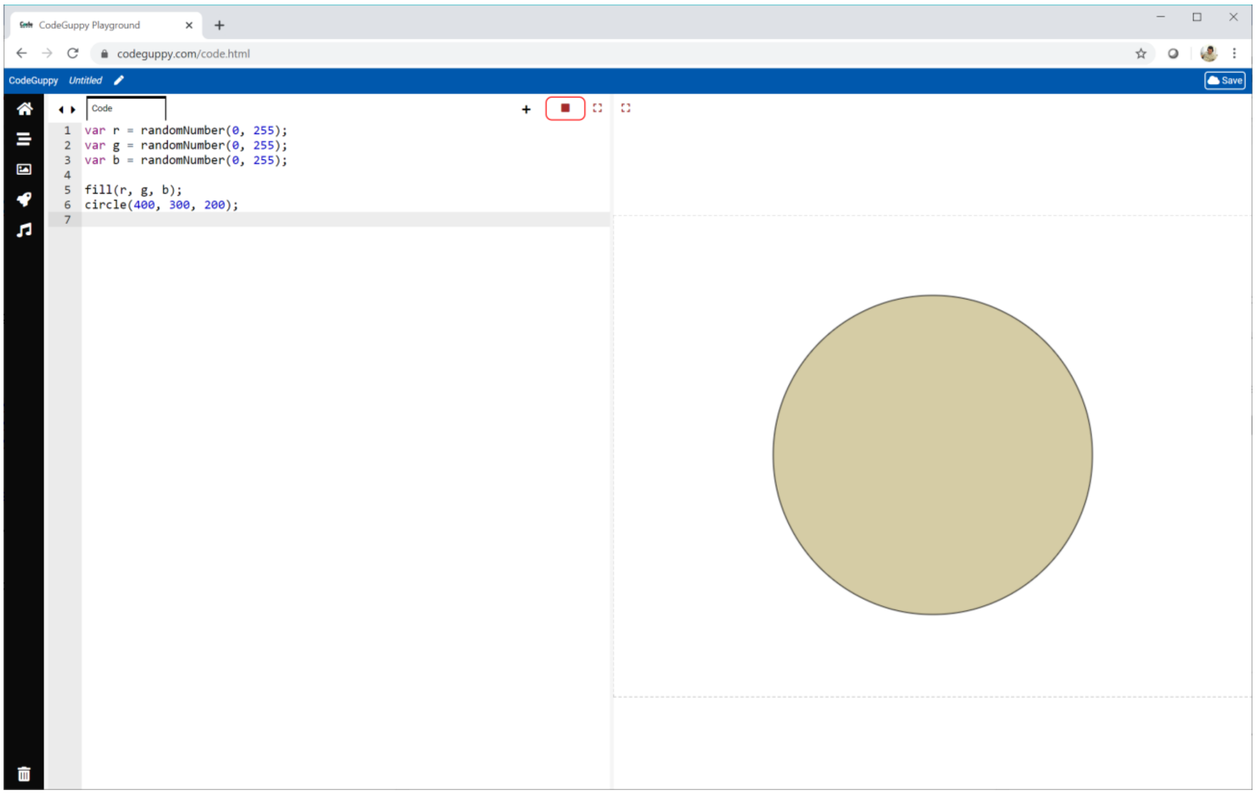
```
var r = randomNumber(0, 255);
var g = randomNumber(0, 255);
var b = randomNumber(0, 255);

fill(r, g, b);
circle(400, 300, 200);
```

You can see we’re setting the variables **r**, **g** and **b** to be random numbers between **0** and **255**.

We’re using those numbers in the **fill()** instruction to set a colour. That colour will be used to fill the circle we draw.

We don’t know what colour the circle will be because it will be mixed from random **RGB** levels.

Here’s what I get if I run the code.



What colour do you get? Run your code again to get a different colour.

Lots of Colours

Let's write a **function** to draw a circle of radius **50** at a random place on the canvas.

We'll use our code from before to give it a random colour.

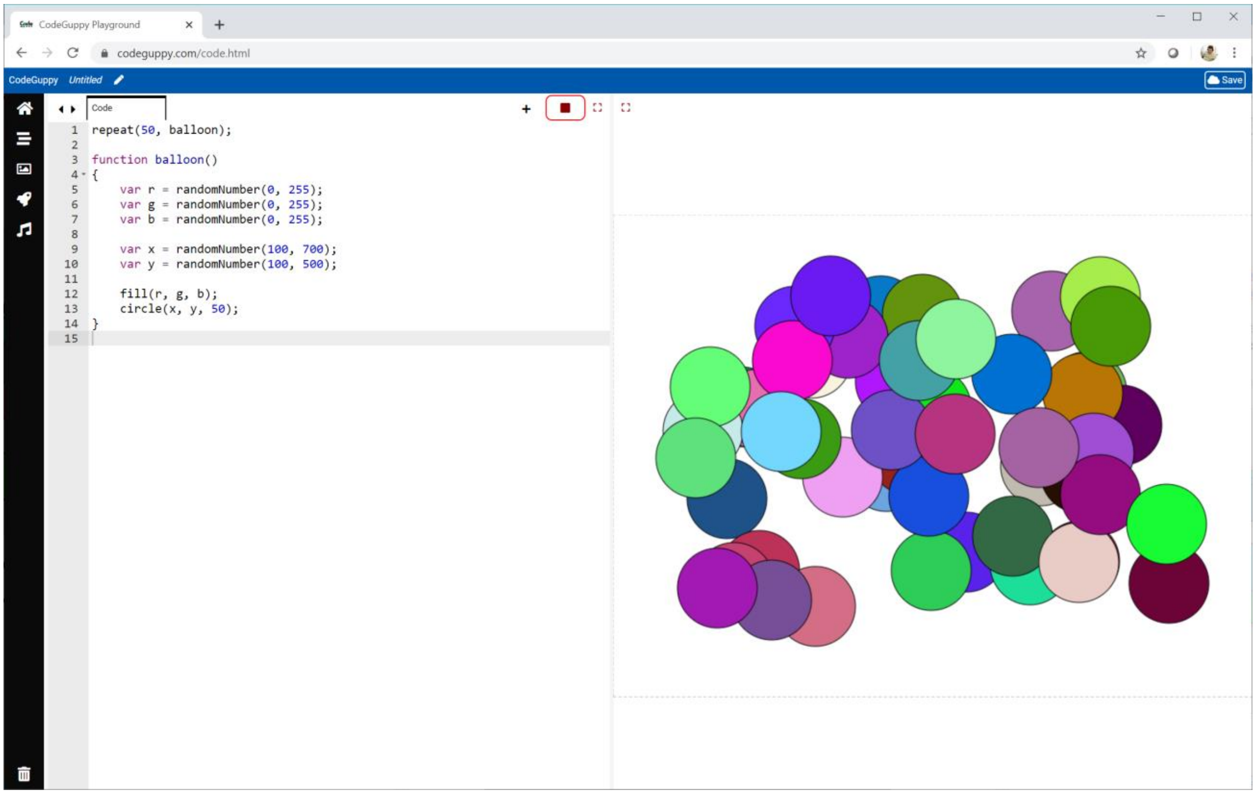
We'll also use the **repeat()** instruction to call this function **50** times, to draw **50** circles.

Here's some code that does this.

```
repeat(50, balloon);

function balloon()
{
    var r = randomNumber(0, 255);
    var g = randomNumber(0, 255);
    var b = randomNumber(0, 255);

    var x = randomNumber(100, 700);
    var y = randomNumber(100, 500);

    fill(r, g, b);
    circle(x, y, 50);
}
```

I've called the function **balloon**.

You can see that we choose random numbers between **0** and **255** for the **red**, **green** and **blue** light.
Here's what my code makes.



When you run it, the code will draw a different pattern with different colours.

Do you think they look like balloons?

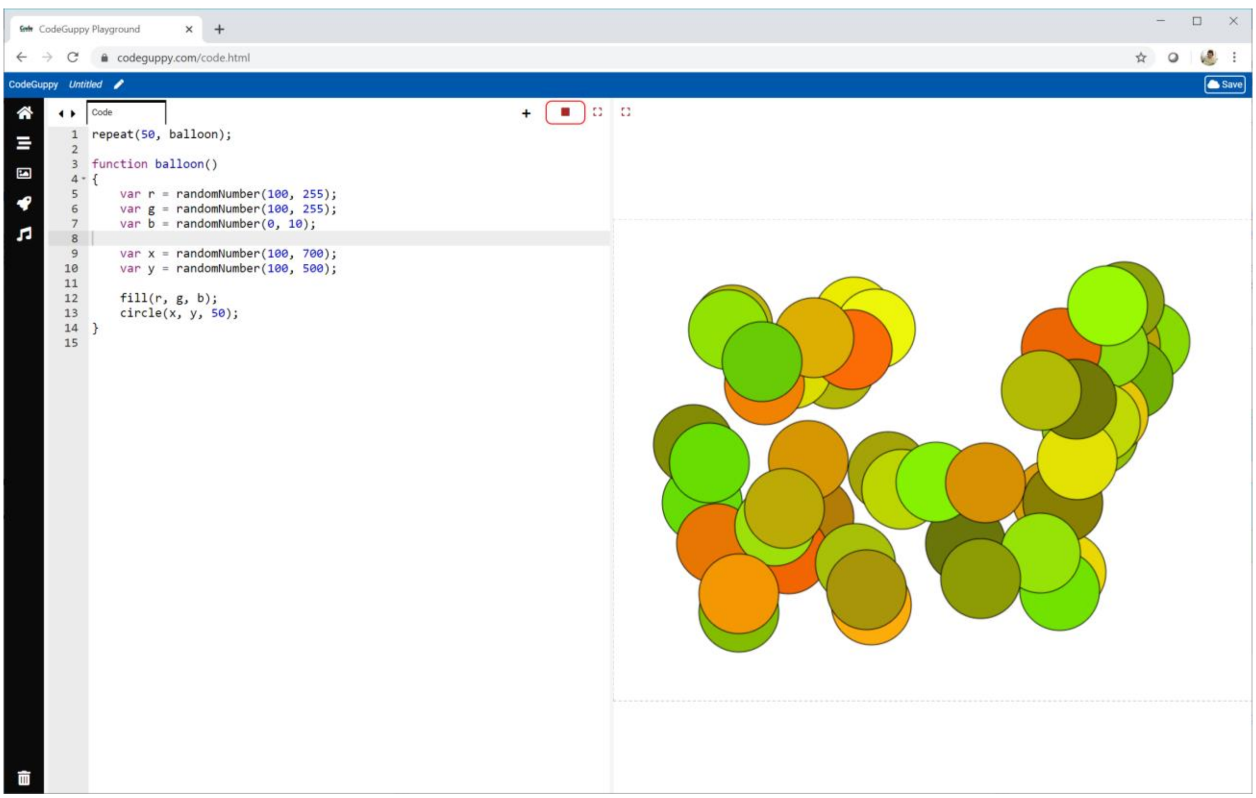
Try It Yourself

We've been making colours by mixing random levels of **red**, **green** and **blue**. Those levels were a number between **0** and **255**.

What if we chose the **RGB** levels from a different range?

```
var r = randomNumber(100, 255);
var g = randomNumber(100, 255);
var b = randomNumber(0, 10);
```

The **red** and **green** levels are chosen to be between **100** and **255**. The **blue** level is chosen to be between **0** and **10**.



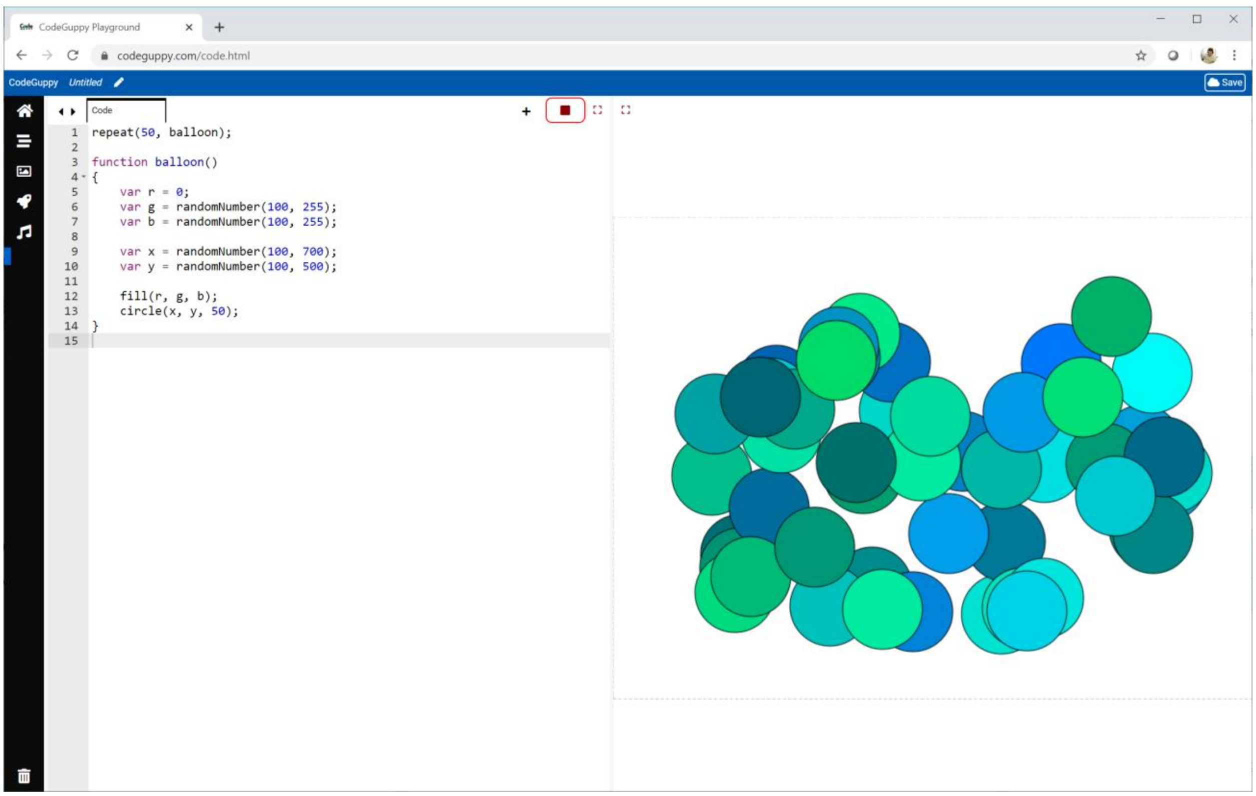
Lovely autumn colours!

Let's try some different ranges.

```
var r = 0;
var g = randomNumber(100, 255);
var b = randomNumber(100, 255);
```

This time the **red** level is set to always be **0**. The **green** and **blue** levels can be between **100** and **255**.

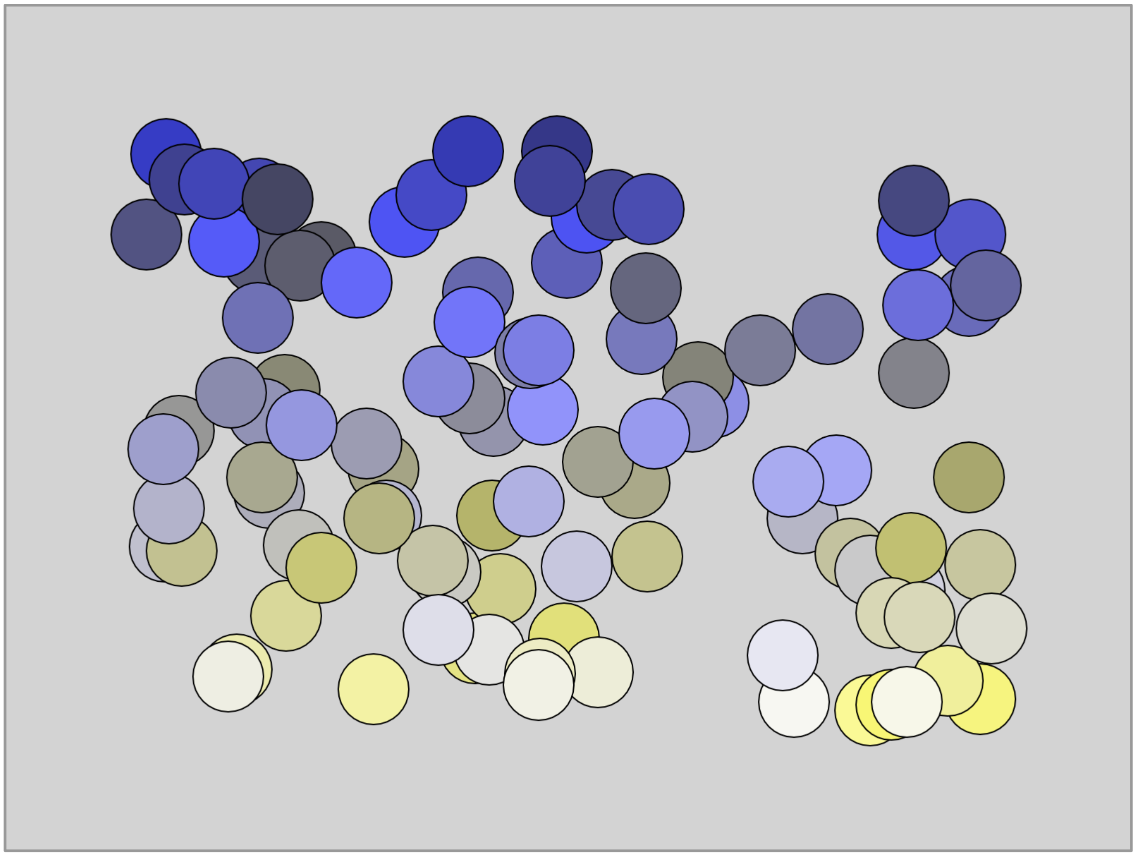
The result is a nice set of greens and blues that remind me of the sea.



Try your own ranges.

Challenge!

The picture below is made with a function that draws a circle at a random place on the canvas.



The **(x, y)** coordinates of each circle are used to calculate the **red**, **green** and **blue** levels of that circle's colour.

- The **red** and **green** levels are the **y** coordinate divided by **2**.
- The **blue** level is a random number between **100** and **255**.

Write your own code to do this. Try your own colour calculations too.

2.5 - More Loops

level

What We'll Do

In this project we're going to:

- learn about **loop counters**
- see how **loop counters** can be useful as **function parameters**

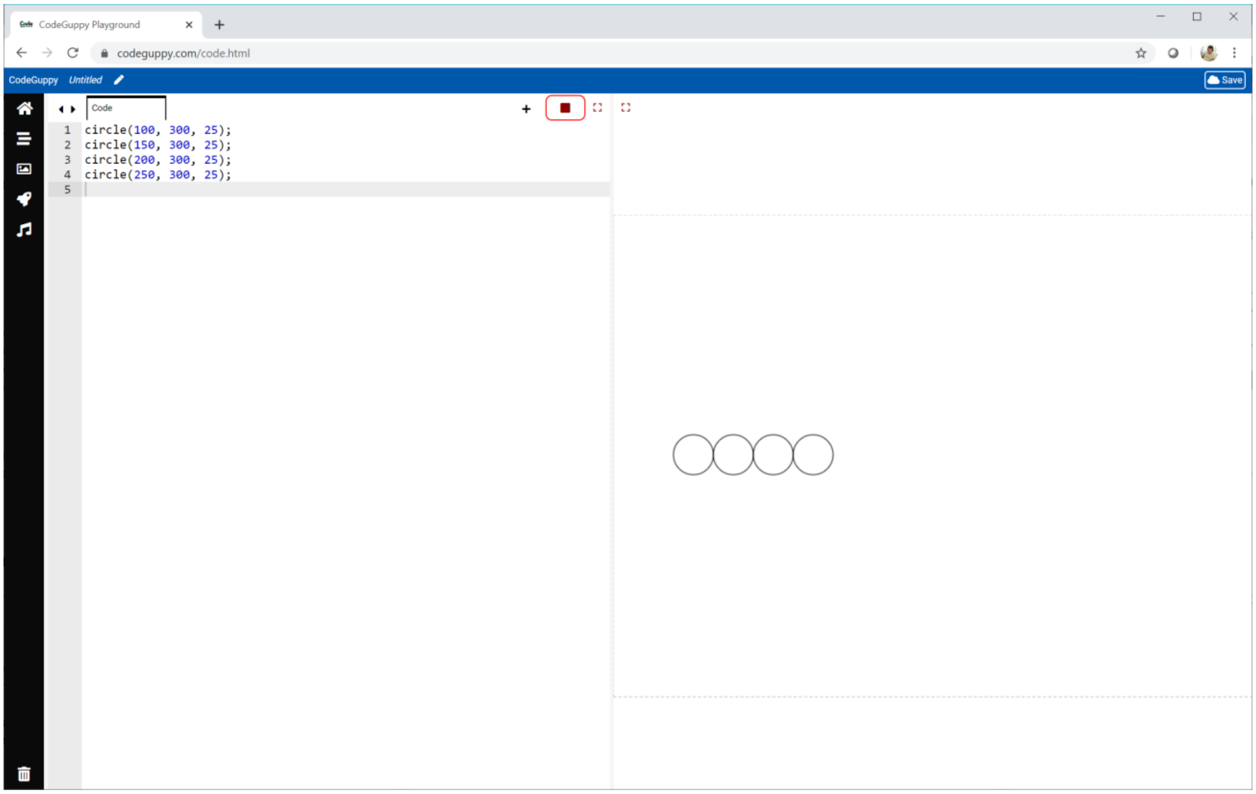
Start a New Program

Log in to codeguppy.com if you haven't already.

Create a new program, just like we did in the first project.

Lots of Balloons

Have a look at this picture of four small balloons.



The code that made this drawing is simple.

```
circle(100, 300, 25);
circle(150, 300, 25);
circle(200, 300, 25);
circle(250, 300, 25);
```

Can you see what changes between each circle instruction?

The **x** coordinate is **100** for the first circle. For the second circle it is **150**. It keeps getting bigger by **50** until it reaches **250** for the fourth circle.

What if we wanted to draw **13** balloons? We could write out **13 circle** instructions.

That would be a lot of typing and very boring. There must be a better way!

Could the **repeat** instruction help us avoid lots of typing?

```
repeat(13, balloon);
```

It’s a good idea - but this code won’t work because the **balloon** function won’t know where to draw each circle.

If **repeat** could pass information to **balloon** that would really help - it could tell the **balloon** function where to draw each circle.

Repeat With Extra Powers!

Have a look at this code showing a new way to use **repeat**:

```
repeat(100, 700, 50, balloon);
```

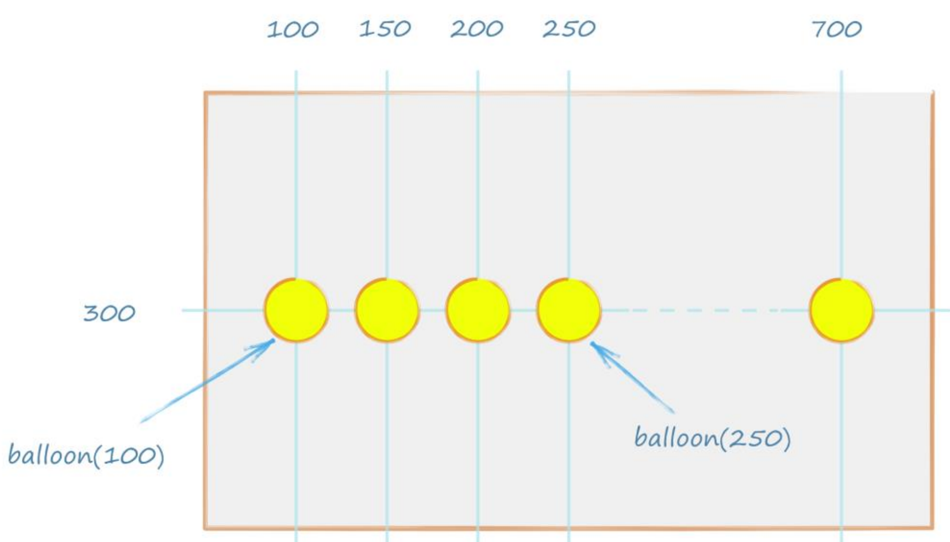
Here, the **repeat** instruction keeps a **counter**. This **loop counter** starts at **100**, and goes all the way up to **700**, increasing by **50** each time.

The **balloon** function is passed the **counter** as a **parameter**.

So the **repeat** instruction calls **balloon(100)**, then **balloon(150)**, then **balloon(200)**, **balloon(250)**, .. all the way up to **balloon(700)**.

That's really useful because the **balloon** function can use the number as the **x** coordinate of the circle.

The next picture shows this idea.



We do need to write our **balloon** function to accept a single **parameter**.

```
function balloon(x) {
    circle(x, 300, 25);
}
```

You can see the **balloon** function takes one parameter, which we've named **x.** The function draws a circle at coordinates (**x**, **300**), with radius **25**.

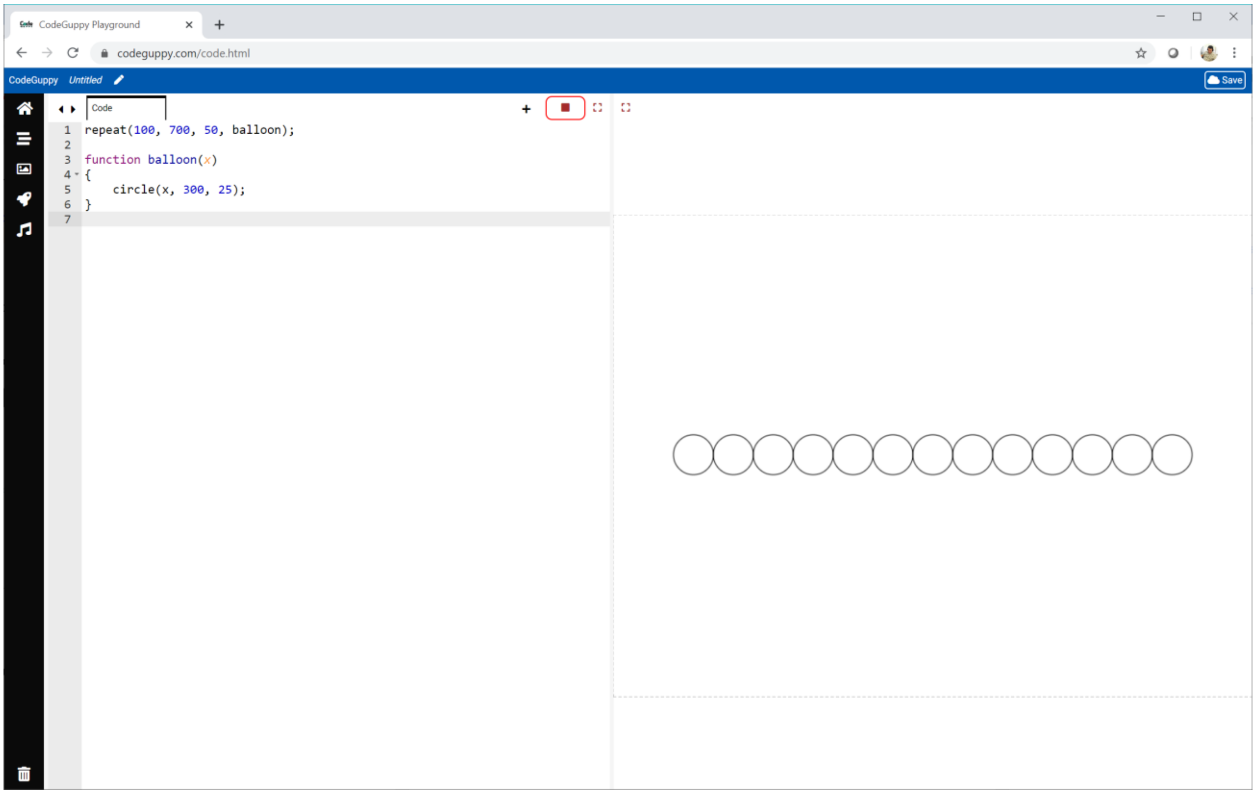
Here's what my code looks like.

```
repeat(100, 700, 50, balloon);
function balloon(x) {
    circle(x, 300, 25);
}
```

You can see the **balloon(x)** function is really simple. It only contains one **circle** instruction.

Outside the function, there is just one **repeat** instruction.

Here are the results.



That worked!

That was a small amount of code to create these **13** balloons.

Passing information to repeated code like this is a really powerful idea. It is used almost everywhere - making games, digital art, electronic music, controlling robots, and apps for your smartphone too.

It’s a good thing to learn and practice!


you've can now pass information to repeated code - a powerful technique!
well done!

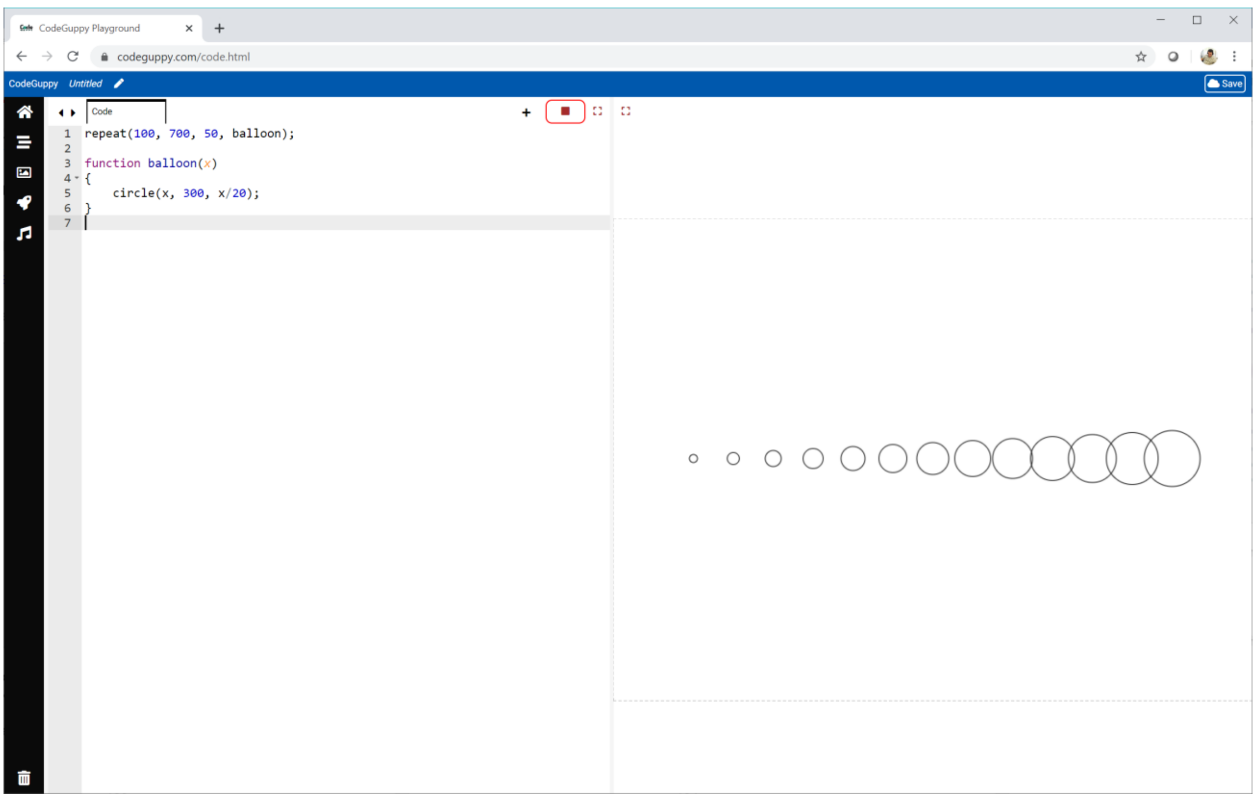
Try It Yourself

Let's try using the **balloon(x)** function's parameter **x** to decide the size of the circle.

Here is my own experiment:

```
circle(x, 300, x/20);
```

The results are interesting.



The circle radius is coded as **x/20**. As **x** goes from **100** to **700**, the circle size goes from **5** up to **35**.

Try your own ideas for calculating the circle radius from **x**.

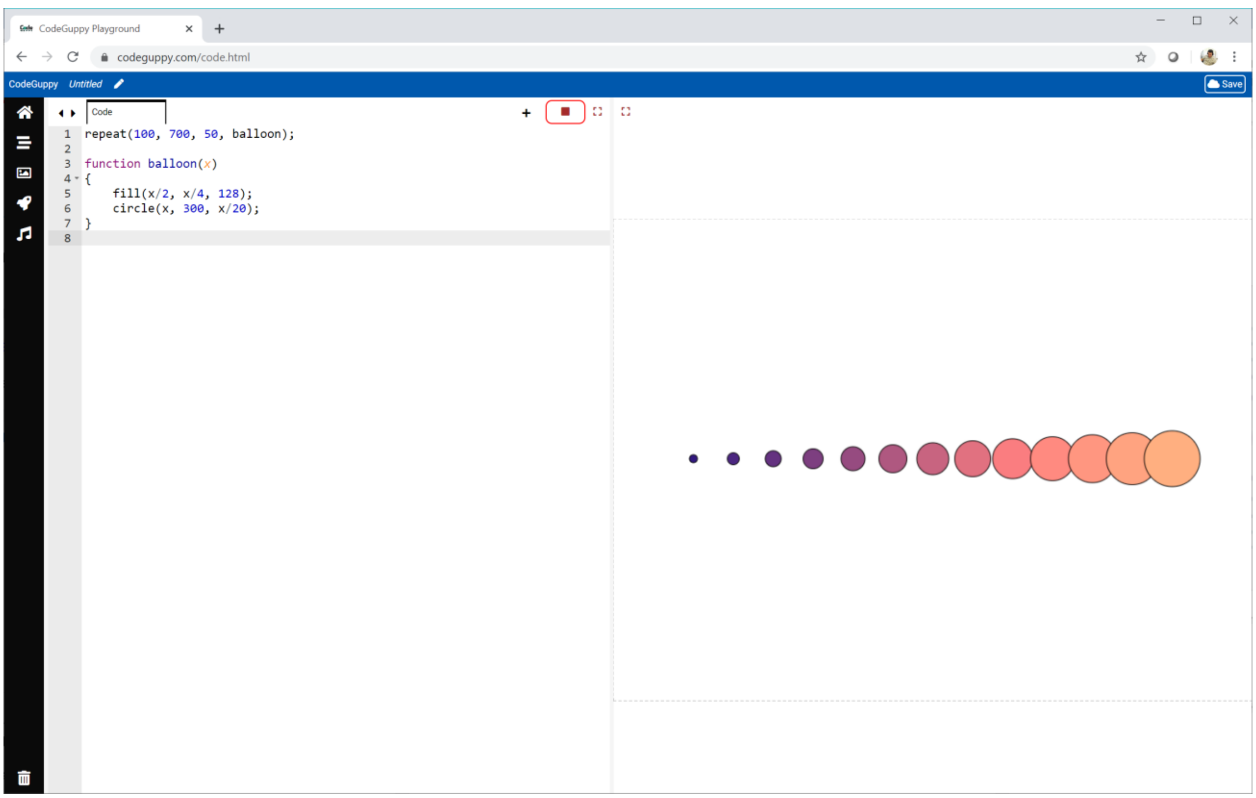
Why not use that parameter to calculate a colour for the balloon?

Here's what I tried.

```
function balloon(x) {
    fill(x/2, x/4, 128);
    circle(x, 300, x/20);
}
```

The colour is mixed with a **red** level of **x/2**, a **green** level of **x/4**, and a **blue** level set to **128**.

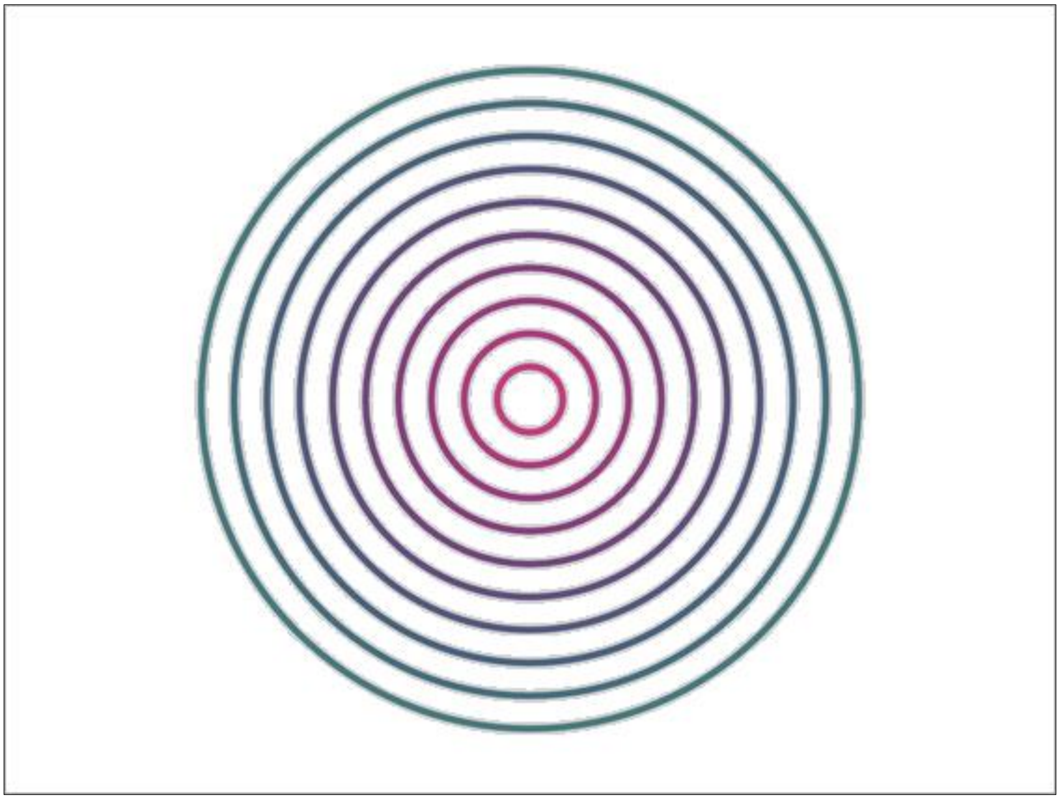
The results are rather cool!



Have a go yourself.

Challenge!

Have a look at this picture.



Each ring is drawn by a function, which we can call **ring**.

A **repeat** instruction with a loop counter is used to tell the **ring** function how big the ring should be, and also what colour it should be.

See if you can write code that creates a similar picture.

2.6 - Artistic Maths

level

What We'll Do

In this project we're going to:

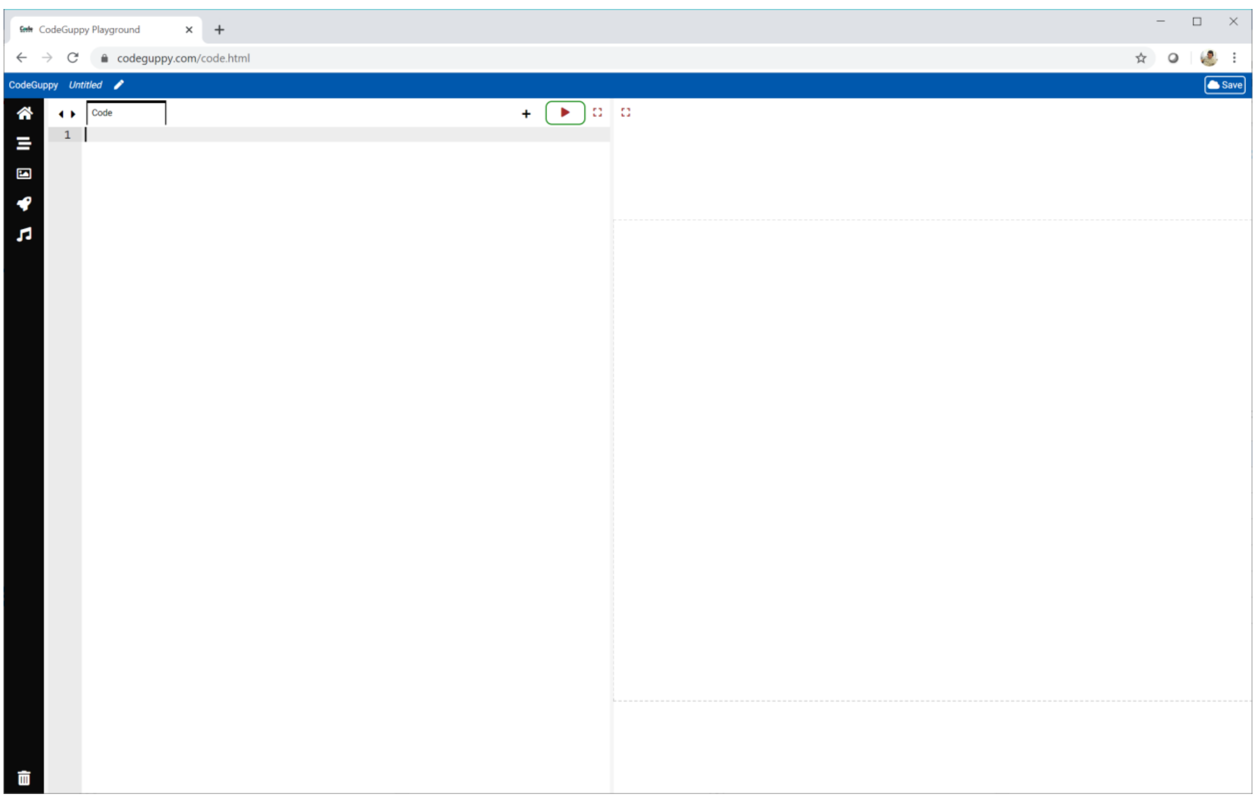
- learn about a simple **maths function** - the **sine** wave
- see how it can help make interesting **patterns**

Start a New Program

Log in to codeguppy.com if you haven't already.

Create a new program, just like we did in the first project.

Your code should look like this:



Drawing Maths Functions

In school we sometimes work with maths functions like this one:

$$y = x + 3$$

If **x** is **1**, then **y** is **4**.

If **x** is **2**, then **y** is **5**.

Easy!

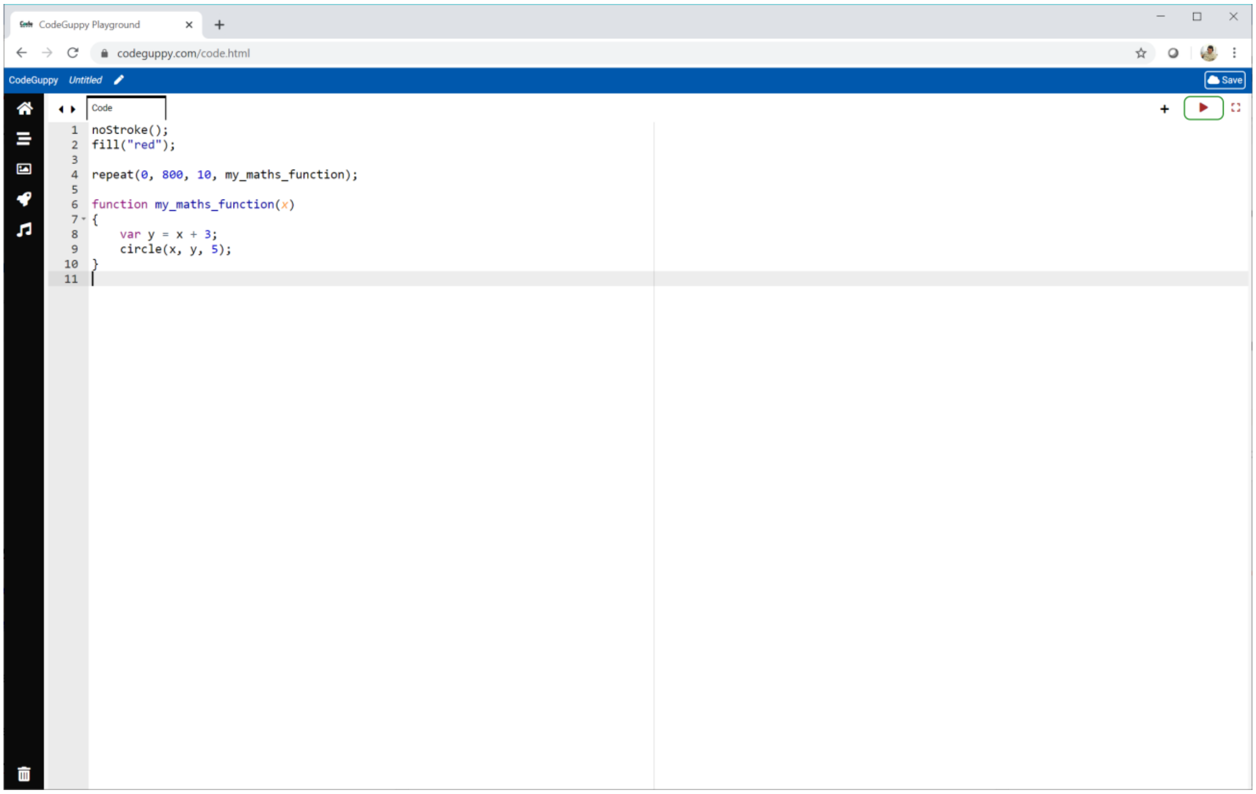
This table shows **y** as **x** goes from **0** to **5**.

x	y
0	3
1	4
2	5
3	6
4	7
5	8

If we draw dots at all these **(x, y)** we might see a pattern.

Let’s try it.

Have a look at this code. None of it is new, but we'll talk about it below.



Let's look at the **my_maths_function** first.

```
function my_maths_function(x) {
    var y = x + 3;
    circle(x, y, 5);
}
```

It takes a parameter **x** and uses it to calculate **x + 3**. The answer is put into a variable **y**.

A small circle is then drawn at **(x, y)**.

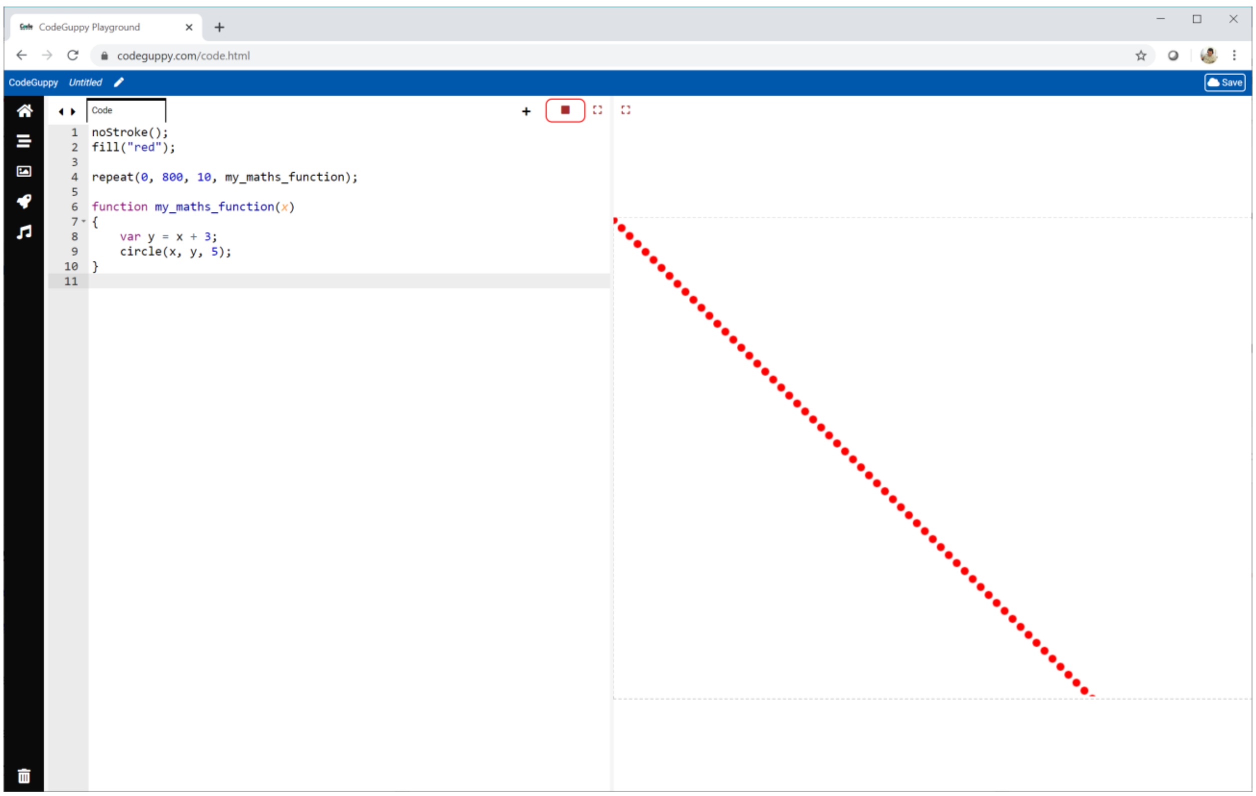
So **my_maths_function** is doing what **y = x + 3** is doing.

We want to pass **my_maths_function** different values of **x**.

We can use the **repeat** instruction to start a counter at **0**, and keep increasing it by **10**, until it gets to **800**. These can be all the values of **x** passed to **my_maths_function**.

```
repeat(0, 800, 10, my_maths_function);
```

Run the code to see what pattern those circles make.



As **x** gets bigger and moves to the right, **y** also gets bigger and moves down the canvas. That’s why there’s a line of dots moving to the right and down.

That pattern is a bit boring!

Let’s try a different maths function to see if it makes a more interesting pattern.

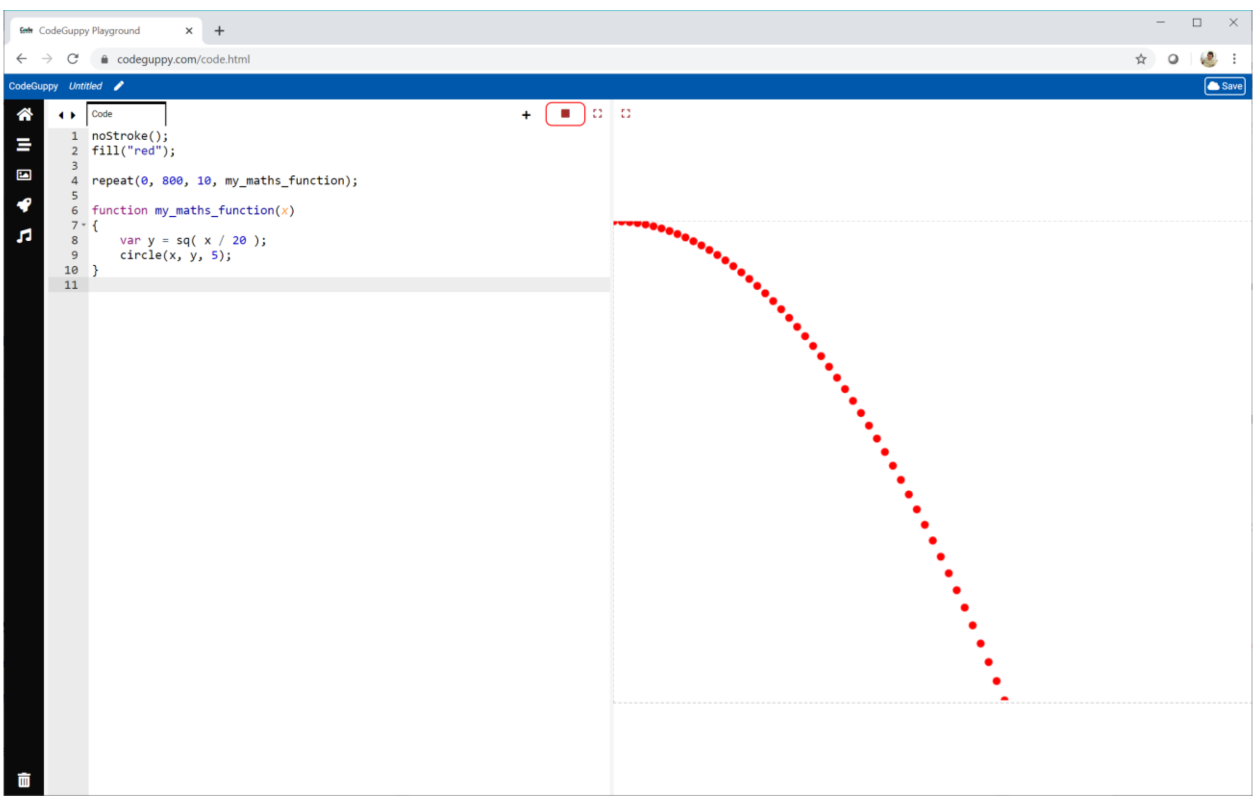
$$y = (x / 20)^2$$

This time **x** is divided by **20** and the answer is squared.

Only one line of code needs to change in **my_maths_function**.

```
var y = sq( x / 20 );
```

Try it yourself to see what pattern this makes.



That’s a bit more interesting.

The path looks like a ball thrown from a wall. It starts falling slowly, and then speeds up.

Let’s try one more mathematical function.

$$y = \sin(x)$$

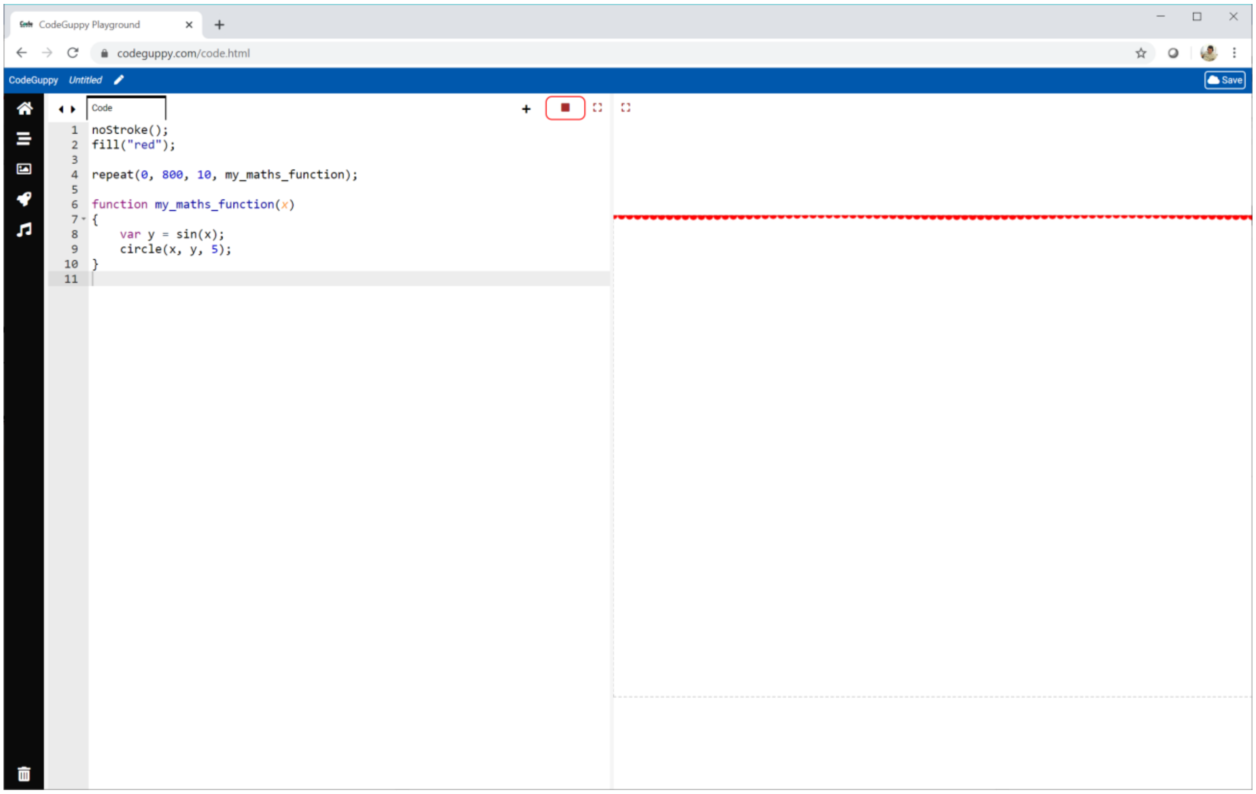
That **sin** is short for **sine**. You might have seen it in your maths class.

Don’t worry if you haven’t seen it before, we’re just going to use it for fun, not work!

Change your **my_maths_function** like this:

```
var y = sin( x );
```

Run your code to see what pattern this **sine** function makes.



What happened?

If you look closely, you can see there's something happening at the top of the canvas.

Maybe the pattern is really short?

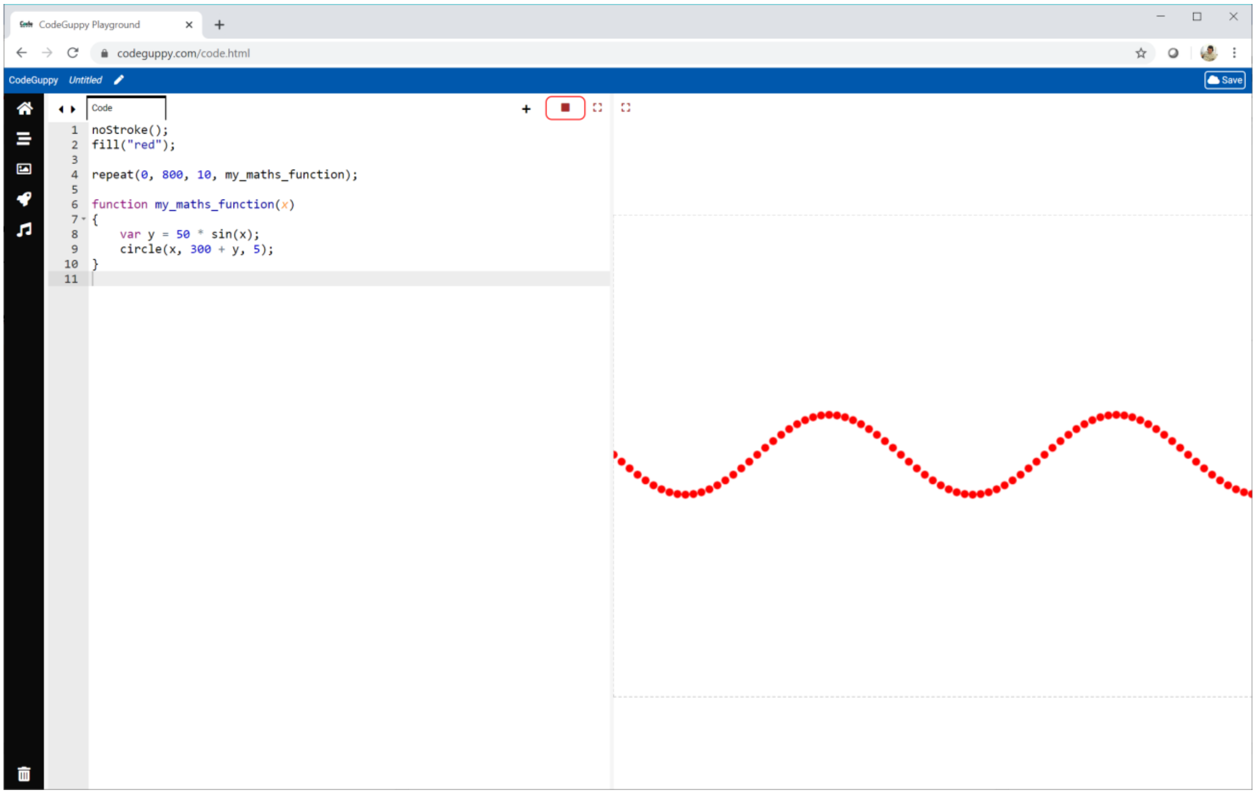
We can make the pattern taller by multiplying the **sin(x)** by **50**.

```
var y = 50 * sin( x );
```

Let's also shift the pattern down so that when **y** is **0**, the dots are drawn in the middle of the canvas. This is easy to do. We just add **300** to the **y** coordinate when we draw the circles.

```
circle(x, 300 + y, 5);
```

Your code and output should look like this.



How cool is that!

The sine function isn't boring at all.

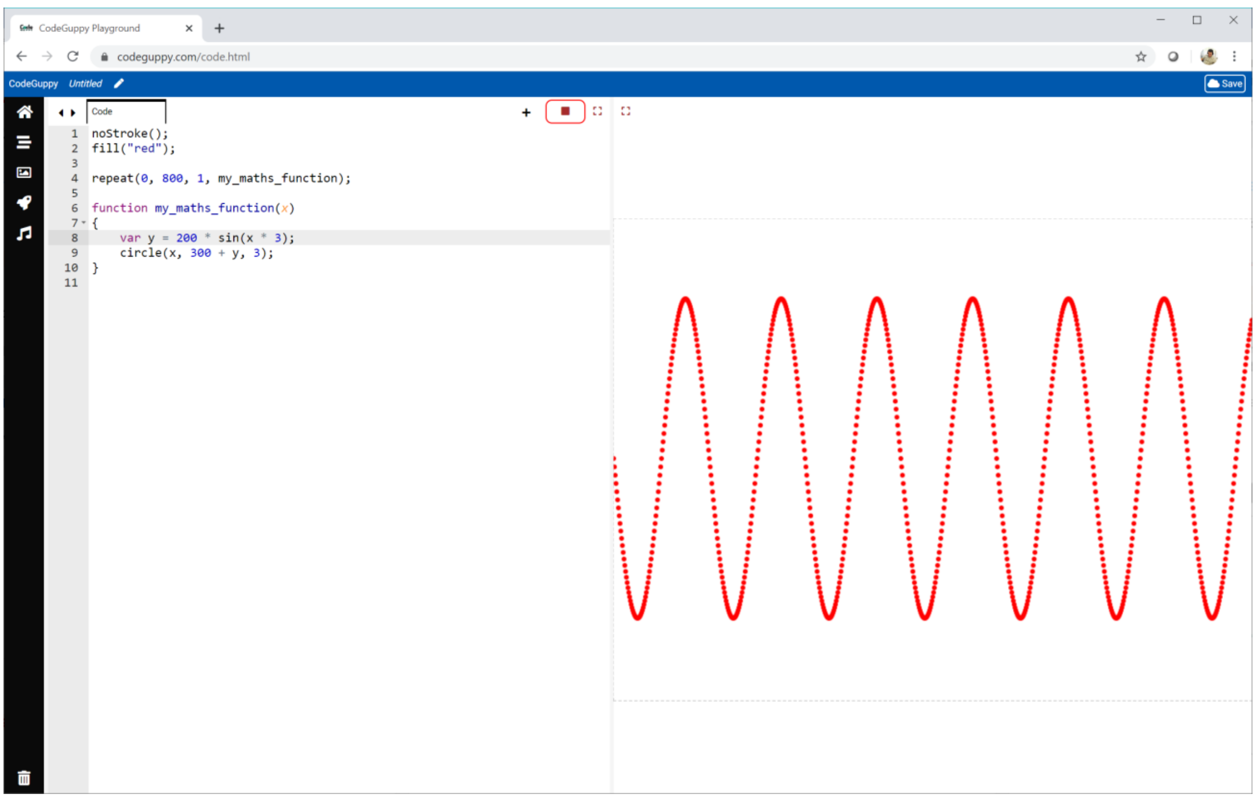
Try It Yourself

Have a look at the **my_maths_function** code again.

```
var y = 50 * sin( x );
```

Try changing the number **50** which makes the sine function taller. Also try multiplying **x** by a different number to make it bigger.

Here is my own experiment.



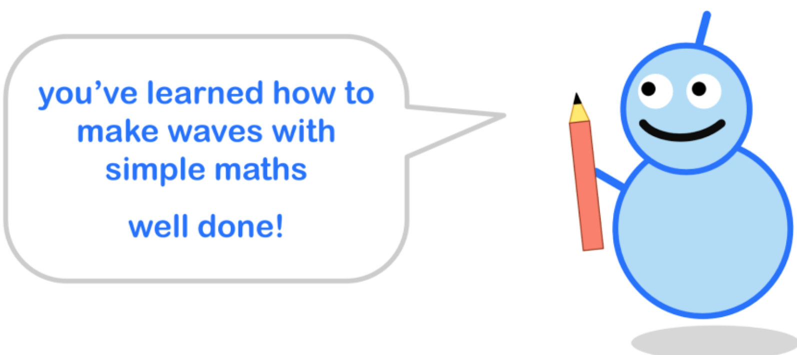
I've multiplied **x** by **3**, and made the sine taller by multiplying it by **200**.

```
var y = 200 * sin( x * 3 );
```

I've also changed the repeat instruction so that the counter grows by **1**.

```
repeat(0, 800, 1, my_maths_function);
```

Because that means more circles, I've made them a smaller size **3**.



Sine Waves For Shape Sizes

Let’s use sine waves to decide the size of shapes.

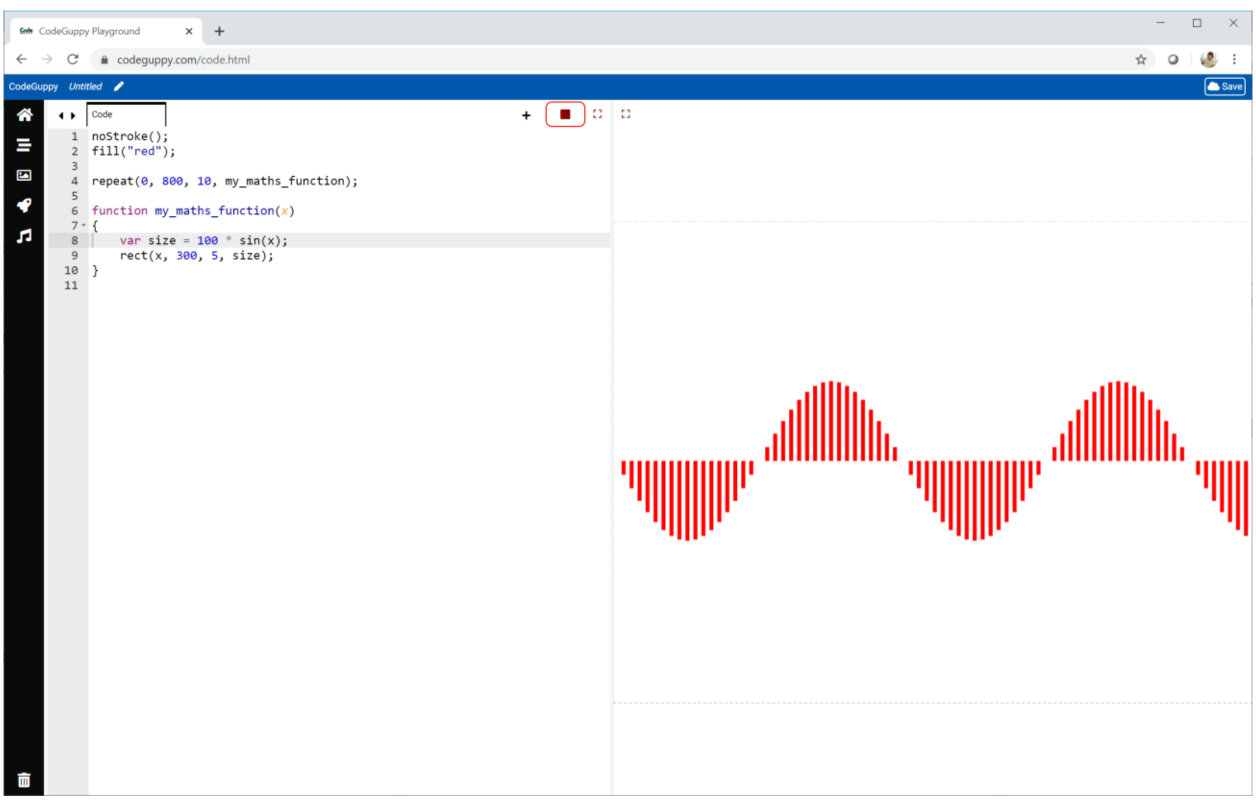
Have a look at this **my_maths_function** code.

```
var size = 100 * sin(x);
rectangle(x, 300, 5, size);
```

We know **sin(x)** goes up and down like a wave. Multiplying it by **100** makes the wave taller. We’ve put this number in a variable called **size**.

We then draw a rectangle at **(x, 300)**, which is along the middle of the canvas. It has a width of **5**, and a height of **size**.

Your code and output should look like this:



That is a cool pattern!

Why do the rectangles go above and below the half-way line?

It’s because sine waves go above and below zero. These positive and negative values mean the rectangle heights become positive and negative too.

You can see a sine wave going up and down between **-1** to **+1** here:

- https://www.desmos.com/calculator/hyimynd5yr

Try It Yourself

Try different sine functions for deciding the size of the rectangles.

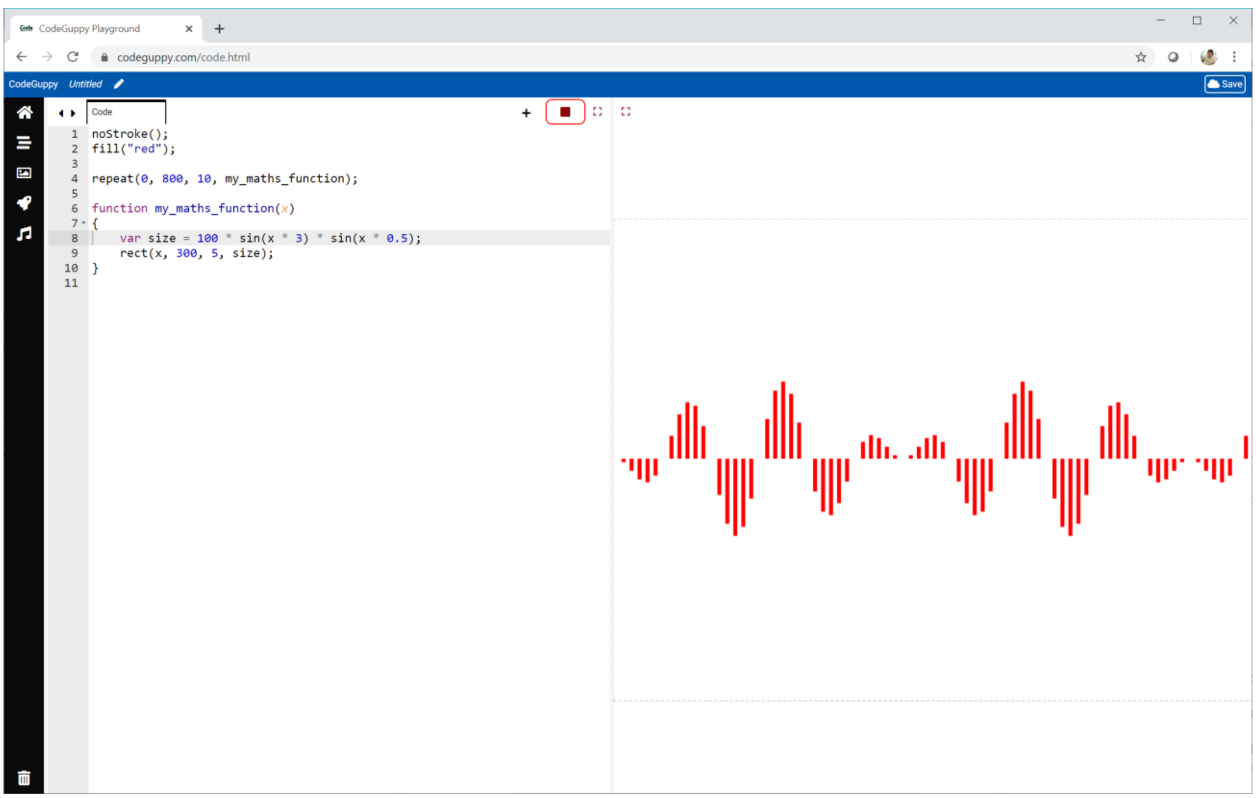
You might want try using two sine functions together. You could add them together. You might even try multiplying them together.

Here is my own experiment:

```
var size = 100 * sin(x * 3) * sin(x * 0.5);
```

I’ve multiplied two different sine functions together.

And this is what it makes.



Funky!

Sine Waves Can Mix Colours

Let's use sine waves to mix colours.

Have a look at this code:

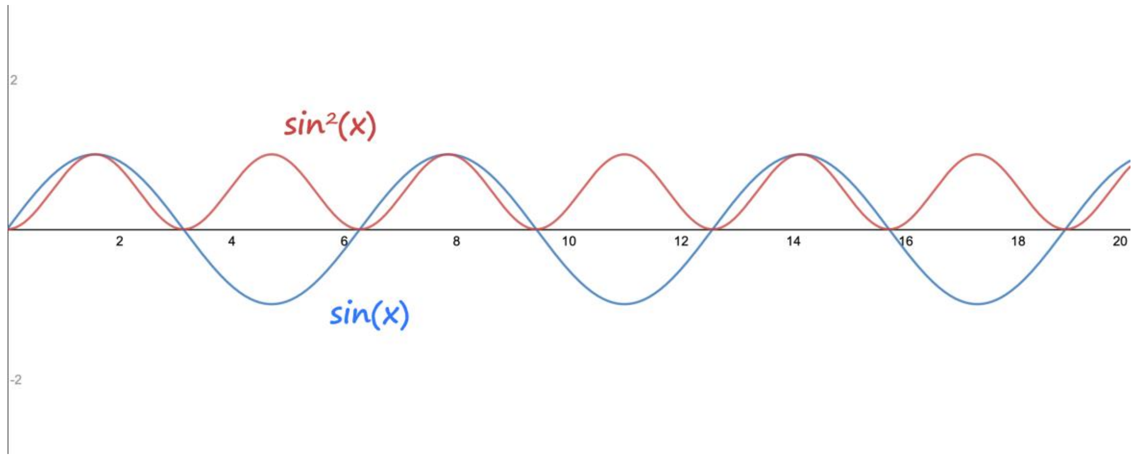
```
var r = 255 * sq(sin(x));
var g = 0;
var b = 128;

fill(r, g, b);
```

The **green** part is set to **0**, and **blue** part set to **128**.

The **red** part depends on what **x** is.

We know **sin(x)** goes up and down like a wave. The code squares that number and then multiplies the answer by **255**. That becomes the **red** part of the colour we're mixing.

Why have we squared the sine function? This picture explains why.



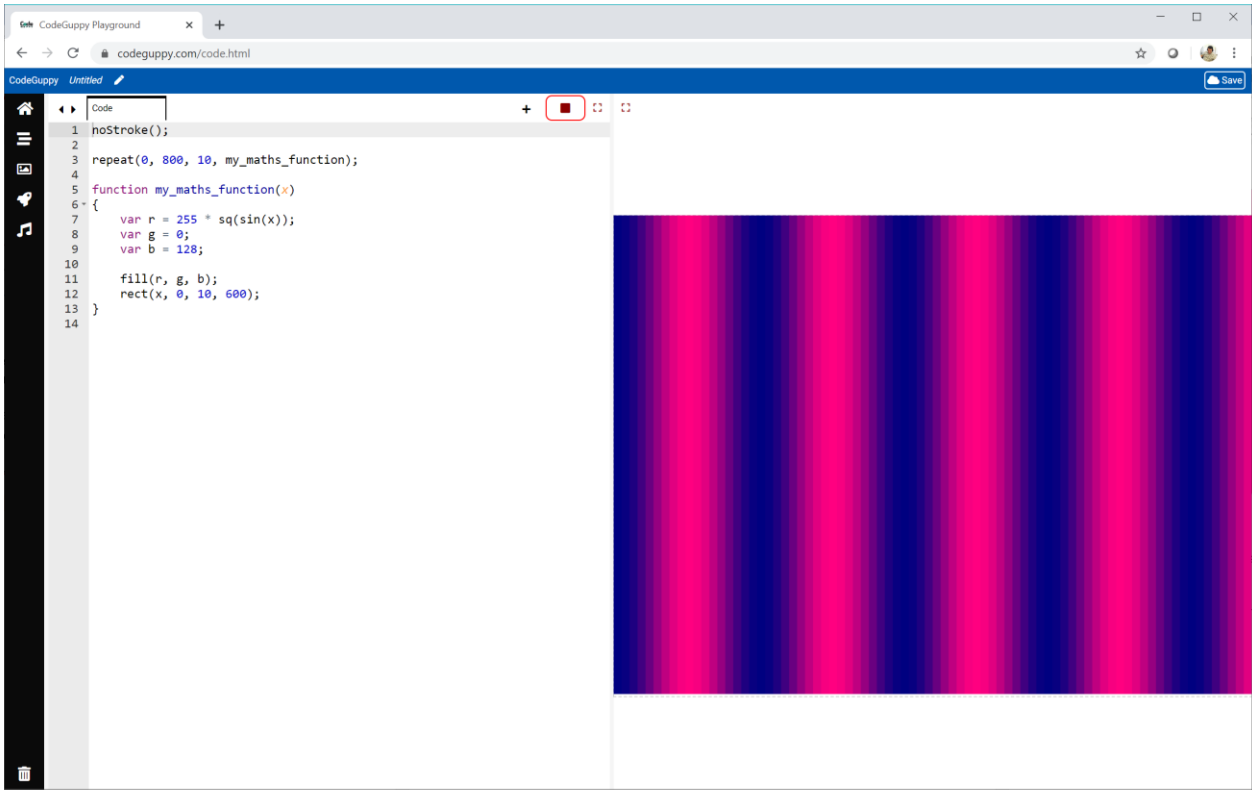
The sine wave goes up and down between **-1** and **+1**. If we square the values, they go from **0** to **+1**.

This makes it easier to scale the values to **RGB** levels. We just by multiplying them by **255**.

Let’s use that colour to draw a thin rectangle all the way down the canvas.

```
rect(x, 0, 10, 600);
```

Your code and output should look like this:



That is quite nice!

The wavy sine function has mixed a wavy amount of **red** into the colour of the thin rectangles.

When the sine value is high, the **red** level is high, and we get that bright pink colour.

When the sine value is low, the **red** level is low, and we get that darker purple colour.

Try It Yourself

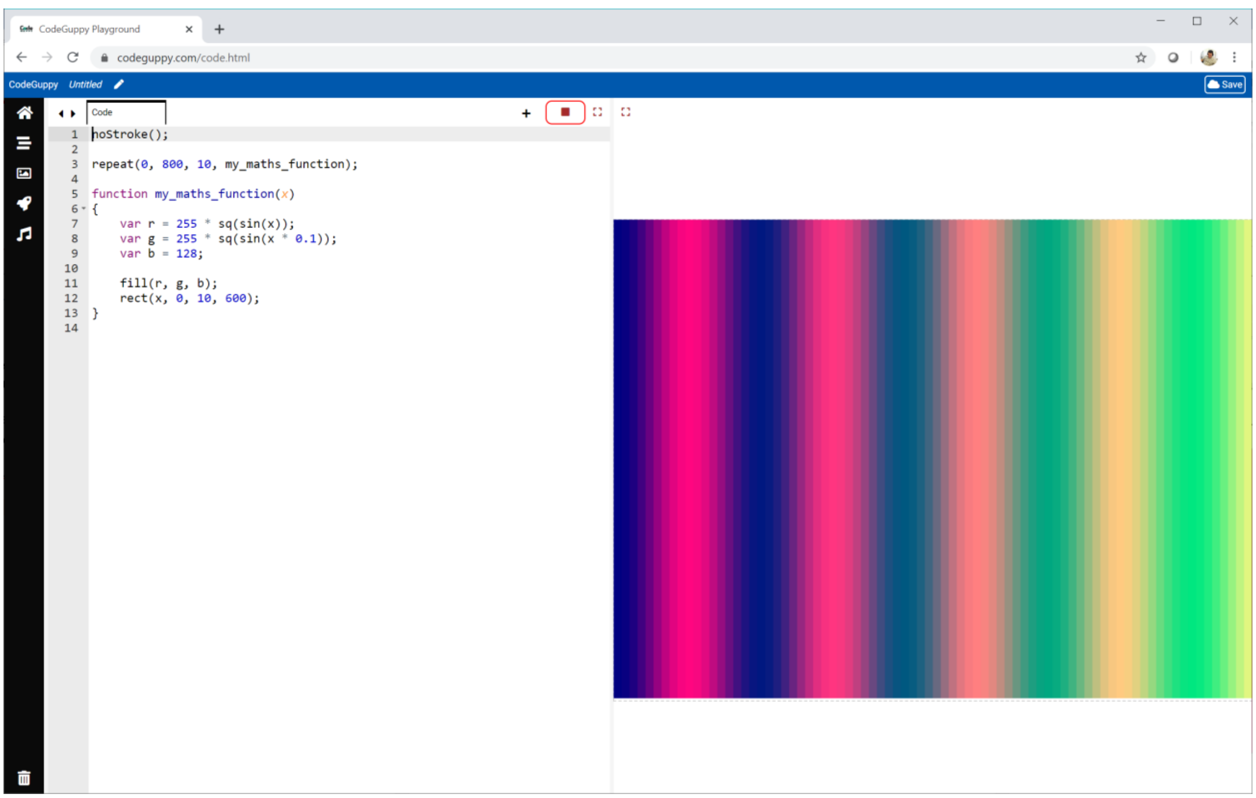
Use your own sine waves to mix colours for those rectangles.

Here is my own experiment:

```
var r = 255 * sq(sin(x));
var g = 255 * sq(sin(x * 0.1));
var b = 128;
```

The **red** part is changing quicker than the **green** part.

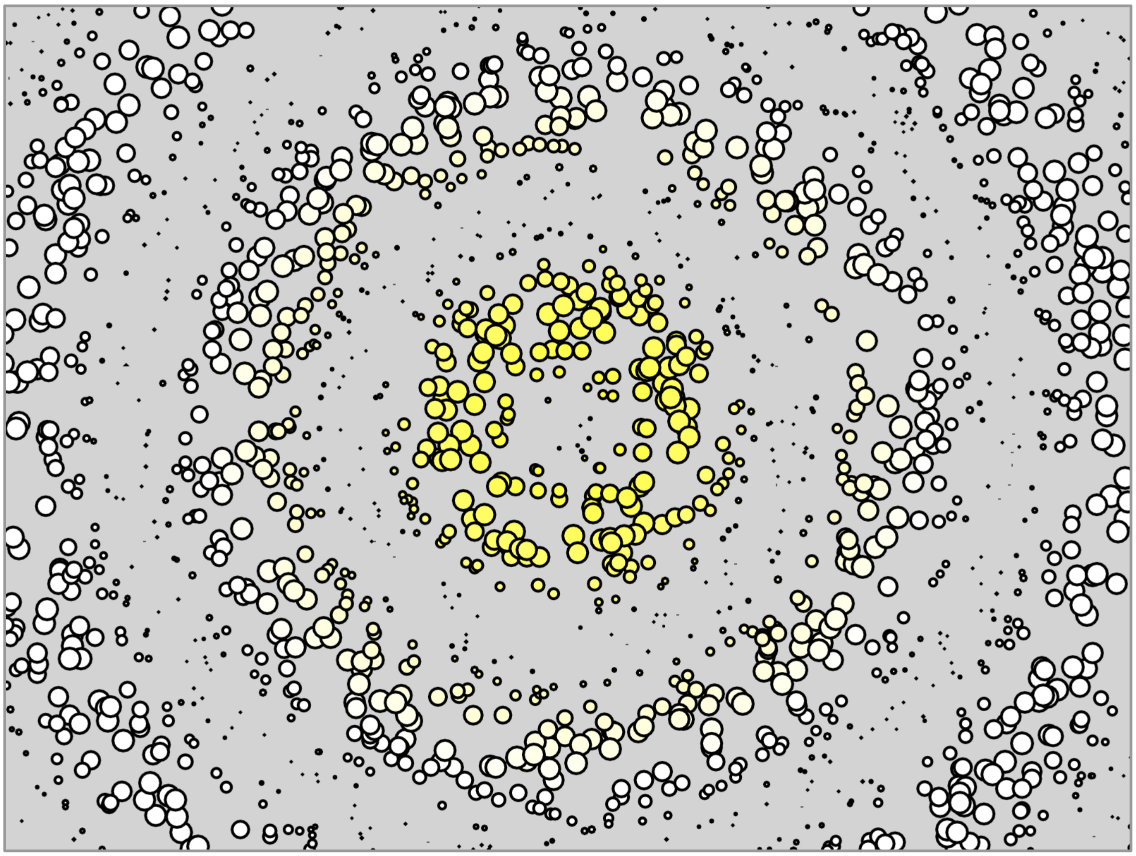
Here's what my code does. What does yours do?




you've learned how to use sine wave to make colour patterns
well done!

Challenge!

Can you write code to make this really cool image?



- There are **1800** dots placed at random places on the canvas. You can use a loop to draw these.
- The size of the dots grows and shrinks with distance from the center. That means using the sine of the distance.
- The colour is mixed using the distance from the center.

2.7 - More Colour

level 

What We'll Do

In this project we're going to:

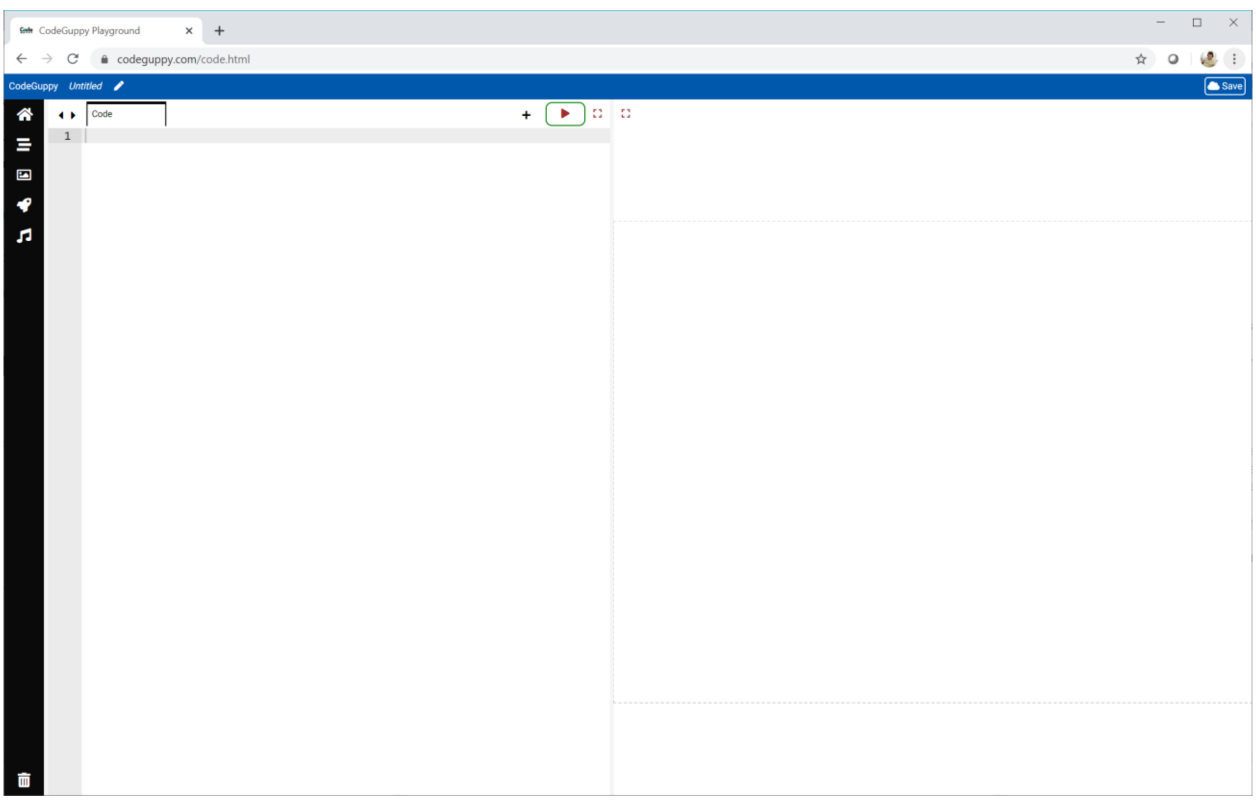
- learn about mixing colours using the **HSB colour model**
- see how it can be **more useful** than the RGB model

Start a New Program

Log in to codeguppy.com if you haven't already.

Create a new program, just like we did in the first project.

Your code should look like this:



Thinking In RGB Isn't Easy

Choosing a colour by mixing **red**, **green** and **blue** light is very common.

But it isn't always the easiest way.

Can you work out the **RGB** values for **yellow** in your head?

The answer is **(255, 255, 0)**.

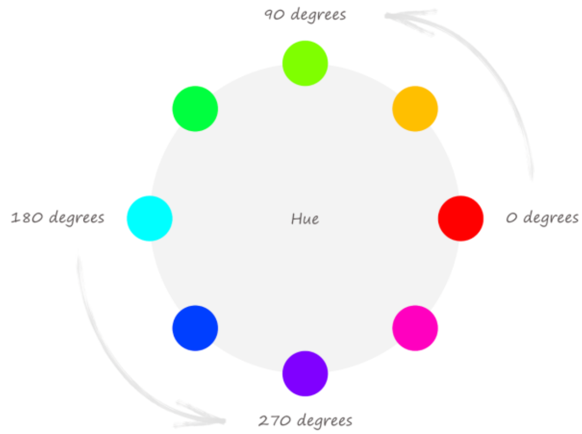
Using these numbers can you work out the **RGB** values for a **light yellow** or a **dark yellow**?

It's not easy.

Luckily, different ways of choosing and mixing colours were invented to make this easier. We'll look at one next.

A Different Way Of Picking Colours

Have a look at the following picture of a **colour wheel**.



Around the wheel we can see colours like **red**, **green** and **blue.**

To pick a colour we say how far round the wheel it is.

A good way of doing this is to use the **angle.** Here are some examples:

- **red** is at **0** degrees
- **orange** is at **45** degrees
- **green** is at **120** degrees
- **blue** is at **240** degrees

Another word for the colours around the wheel is **hue.**

What's the angle for the **purple** hue?

Let’s write some code to show some colours from the wheel.

We can use a **repeat** instruction to count from **0** to **360** in steps of **10**. These are the angles that go all the way around the colour wheel.

```
repeat(0, 360, 10, spot);
```

These angles can be passed to a function to draw a colored spot. The **hue** of the spot will be the one at that angle on the colour wheel.

Have a look at this code for the **spot** function.

```
function spot(hue) {

    fill(hue, 100, 100);
    var x = 50 + (hue * 2);
    circle(x, 300, 10);

}
```

We can see the **spot** function takes the number passed to it and calls it **hue**.

We can also see the **fill** instruction to pick a colour. It has **3** numbers passed to it but these are not the usual **RGB** levels. The first number is the **hue** we’ve talked about. We’ll talk about the other two numbers soon.

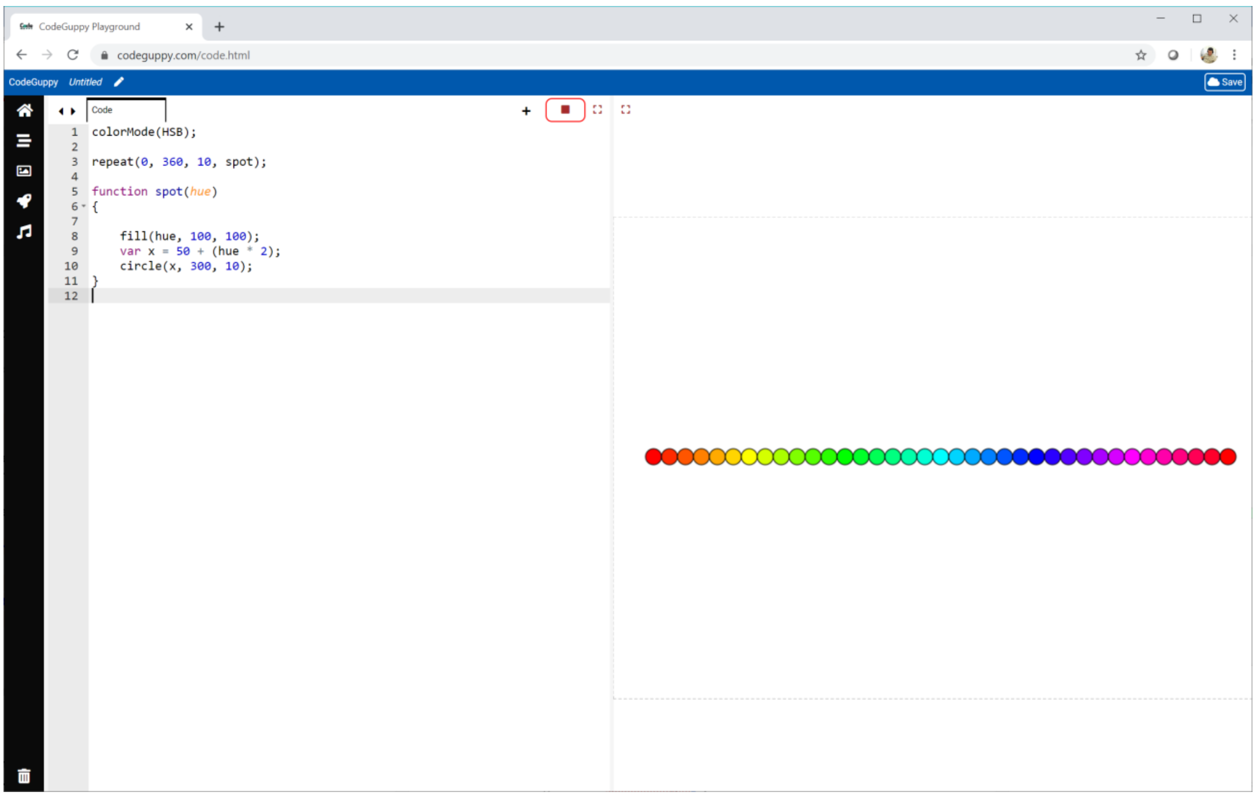
A circle is drawn along the middle of the canvas, with an **x** coordinate that scales **hue** to fit across the canvas.

Before we run the code, we need to tell our computer that we're not going to use the **RGB** way to pick colours. We're going to use something called **HSB**.

```
colorMode(HSB);
```

This instruction needs to go at the beginning of our code.

Your code and output should look like this:



Great!

We can see the **hue** going from **red**, through **orange**, **yellow**, **green**, **blue**, **purple** and back to **red** again.

Picking colours from the wheel is easier than trying to find the right combination of **red**, **green** and **blue**.

Look at that **fill** instruction again.

```
fill(hue, 100, 100);
```

Let’s talk about those three numbers:

- The first number is the **hue** on the colour wheel.

- The second number is the **saturation**. Turning down the saturation is like adding water to paint. More water makes the colour weaker.

- The last number is the **brightness**. Turning down the brightness is like turning down the lights. Things get darker!

Hue values go from **0** to **360**. **Saturation** and **brightness** values go from **0** to **100**.

The letters of **HSB** come from **Hue**, **Saturation** and **Brightness**.

RGB and **HSB** are two different **colour models**. That just means two different ways of describing colours.

Changing Brightness

Let’s change our code to see what happens if we change the **brightness** values.

Lets add a new function to our code that draws circles with **brightness** set to **50**.

Let’s call it **spot_lower_brightness**.

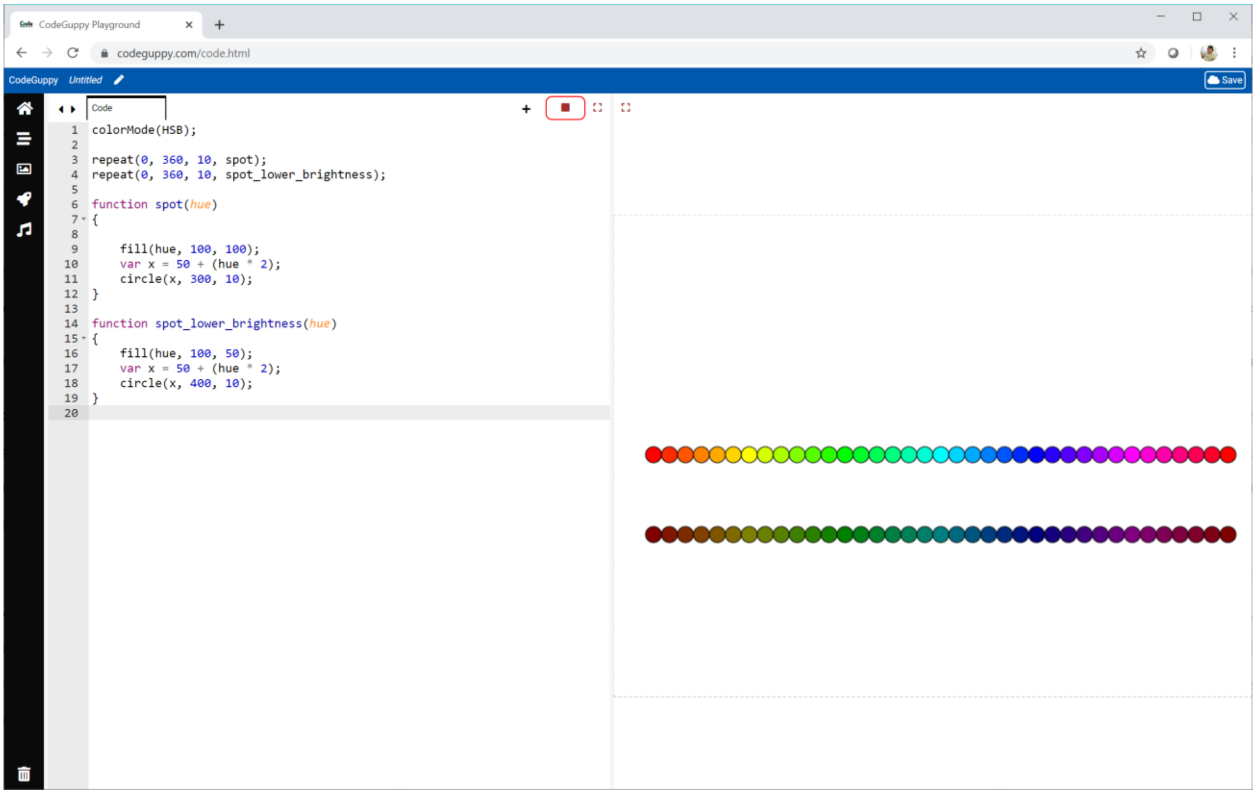
```
function spot_lower_brightness(hue) {

    fill(hue, 100, 50);
    var x = 50 + (hue * 2);
    circle(x, 400, 10);

}
```

The circles are drawn with a **y** coordinate of **400**, which means they’ll make a line below the line we had before.

We can use a **repeat** instruction to call this new function.

```
repeat(0, 360, 10, spot_lower_brightness);
```

Let’s see what these circles with lower **brightness** look like.



This picture makes it really clear that lowering the **brightness** makes the colours darker, closer to **black**.

This is good. If we know that **yellow** has a hue of **60** degrees, we can make a **dark yellow** easily by reducing the **brightness** value.

That was much harder to do with the **RGB** colour model.

Changing Saturation

Let’s change our code to see what happens if we change the **saturation** values.

Again, let’s create a new function **spot_lower_saturation** that draws circles with **saturation** set to **40**.

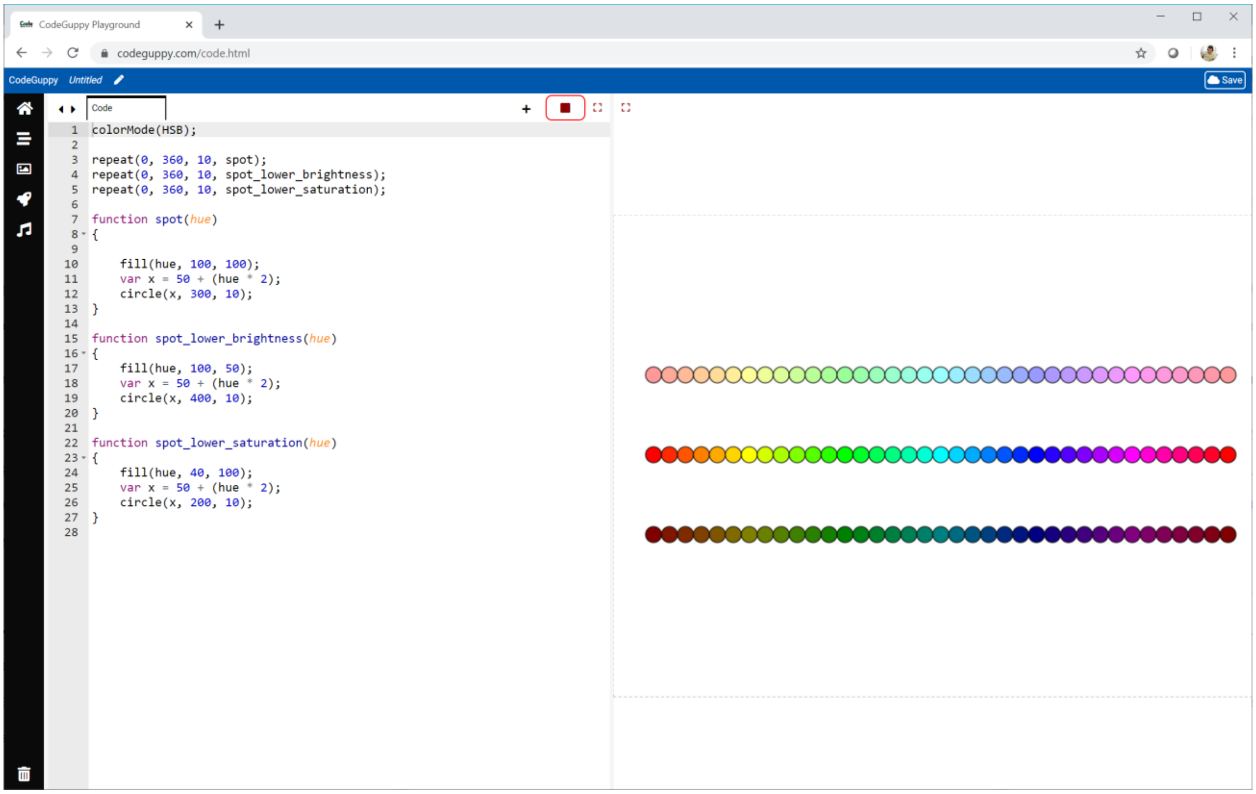
```
function spot_lower_saturation(hue) {

    fill(hue, 40, 100);
    var x = 50 + (hue * 2);
    circle(x, 200, 10);

}
```

The circles are drawn with a **y** coordinate of **200**, which means they’ll form a line above the original circles.

We can use a **repeat** instruction to call this new function.

```
repeat(0, 360, 10, spot_lower_saturation);
```

Let’s see what these circles with lower **saturation** look like.



The picture show how lower **brightness** values make the colours lighter, closer to **white**.

Making a **light yellow** is easy. We just lower the **saturation** compared to the normal **yellow**.

The code for this program is online:

- https://codeguppy.com/code.html?tariq/ex01

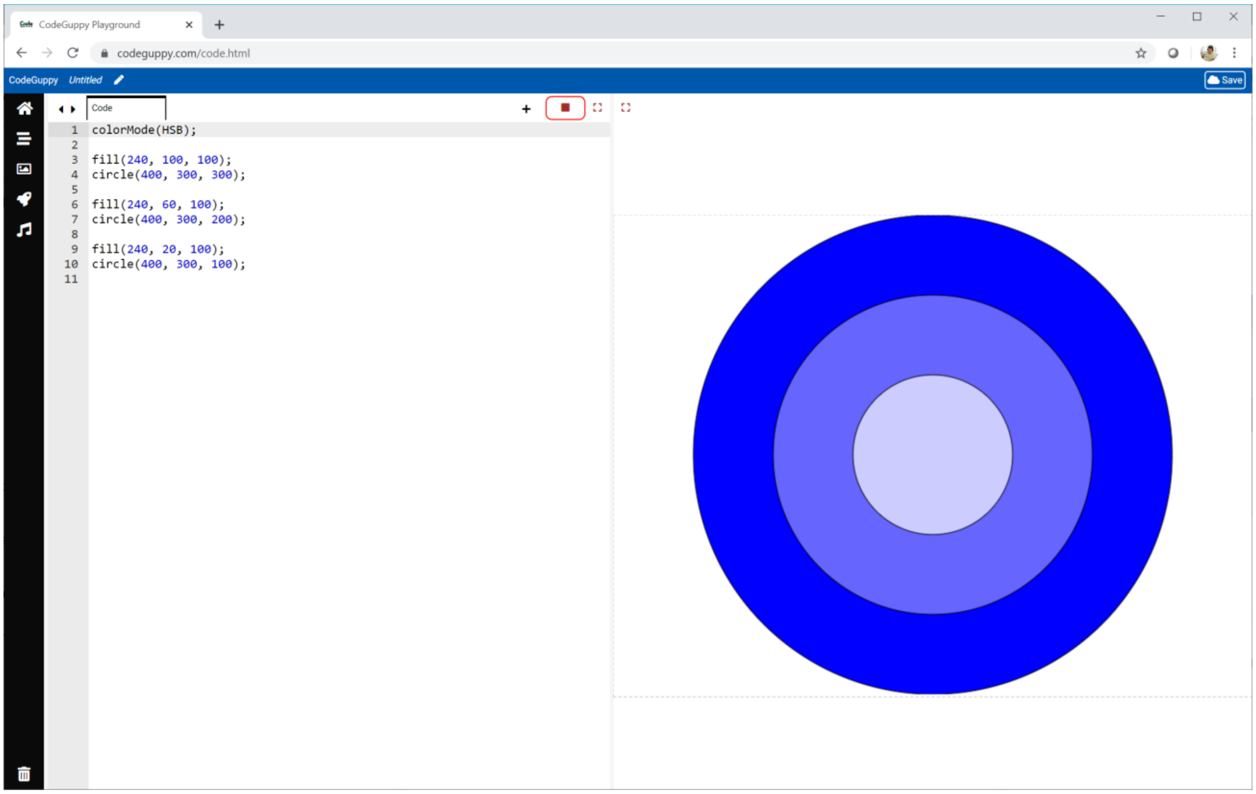
Try It Yourself

The following code draws three circles filled using the **HSB** values.

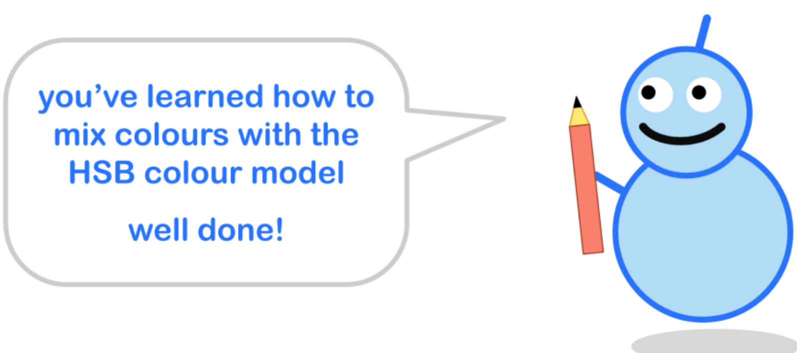
```
fill(240, 100, 100);
circle(400, 300, 300);

fill(240, 60, 100);
circle(400, 300, 200);

fill(240, 20, 100);
circle(400, 300, 100);
```

I've only changed the **saturation**.

This is what my code draws.



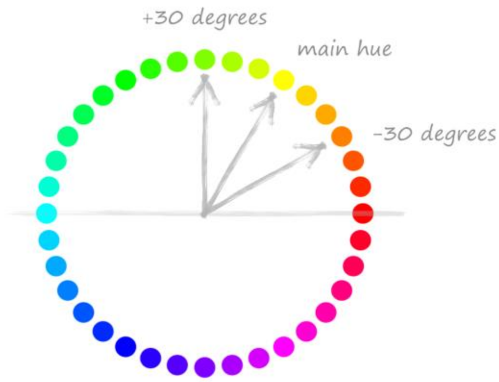
Experiment with your own **HSB** values.



Calculating Colour Combinations

There's something else the **HSB** colour model is good for - calculating nice colour combinations.

Have a look at the colour **yellow** on this colour wheel.



The dots near that **yellow** dot are similar in colour. You can see a **yellowy-orange** and a **yellowy-green**.

If we have a **hue** value, we can calculate similar colours by choosing new **hue** values that are close to the original value.

Let's try it.

Have a look at the following function we've called **stripes**.

```
function stripes(x) {

    var hue1 = randomNumber(45, 315);
    var hue0 = hue1 - 15;
    var hue2 = hue1 + 15;

    fill(hue0, 50, 90);
    rect(x, 200, 25, 200);

    fill(hue1, 50, 90);
    rect(x + 25, 200, 25, 200);

    fill(hue2, 50, 90);
    rect(x + 50, 200, 25, 200);

}
```

The function picks a **hue1**, randomly chosen to be between **45** and **315** degrees on the colour wheel.

This **hue1** is used to calculate two more hue values. **hue0** is **15** degrees less than **hue1**, and **hue2** is **15** degrees more.

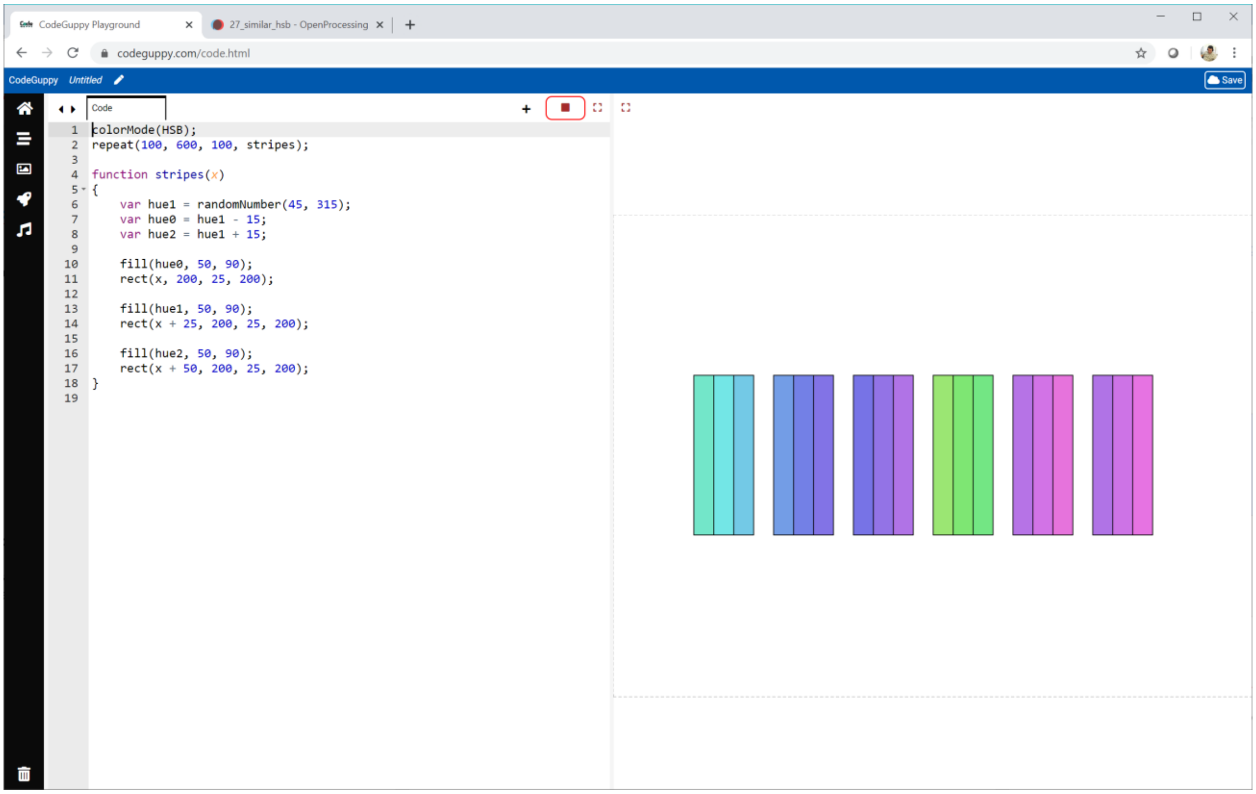
These hues are then used to fill rectangles drawn next to each other, starting at **(x, 200)**. That **x** is provided to the function as a parameter.

We can use a **repeat** instruction to count from **100** to **600**, going up in steps of **100**, and calling **stripes** with that counter.

The full code is online at:

- https://codeguppy.com/code.html?tariq/ex02

Let’s see the result.



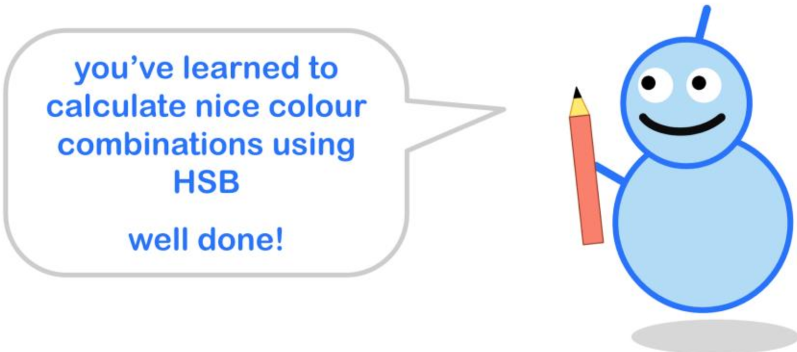
That’s a rather nice effect.

You can see the three colours in each group are related to each other.

This shows how easy it is to calculate colour combinations with **HSB**.

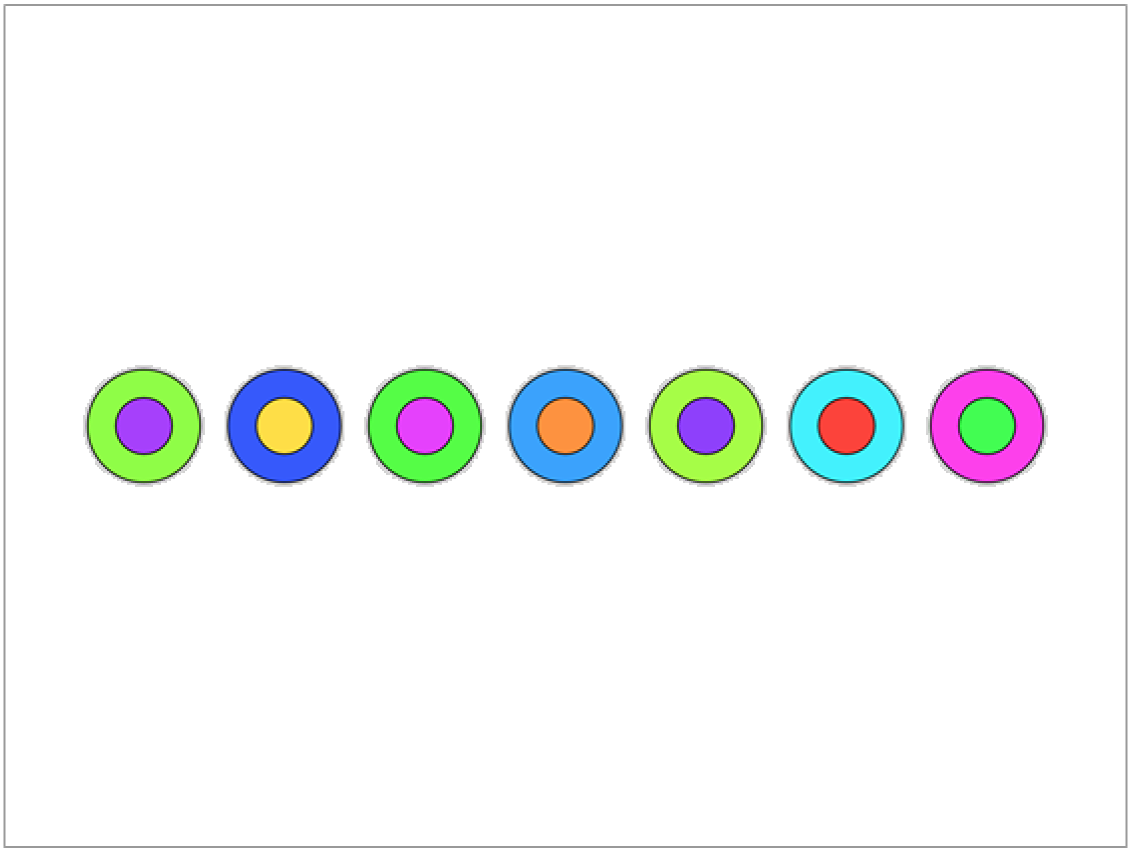
You might have seen apps and games using similar colour combinations to make them look nice.

Change the code to try your own calculations.


you've learned to calculate nice colour combinations using HSB
well done!

Challenge!

Have a look at the following picture.



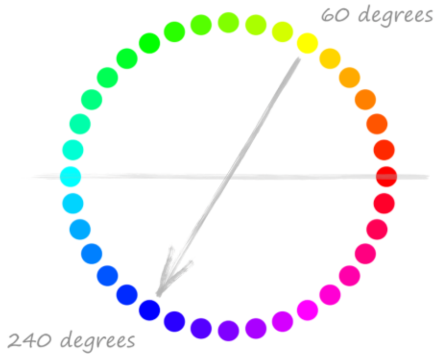
The outer circles have randomly chosen hues.

Each inner circle has a colour which is the **opposite colour** to the outer circle.

Can you write code to draw similar colour patterns?

You'll remember from school that **opposite colours** look the most different to each other. For example **yellow** is opposite to **blue**.

On the colour wheel, opposite colours are on opposite sides of the circle.



Yellow has a **hue** of **60** degrees. To go to the opposite side of the circle, we add **180** degrees. That makes **240** degrees, which is **blue**.

Adding **180** degrees is how you would calculate a hue's opposite.

Sometimes adding **180** degrees makes the angle larger than **360** degrees. See if you can work out how to deal with this.

2.8 - Loops Inside Loops

level

What We'll Do

In this project we're going to:

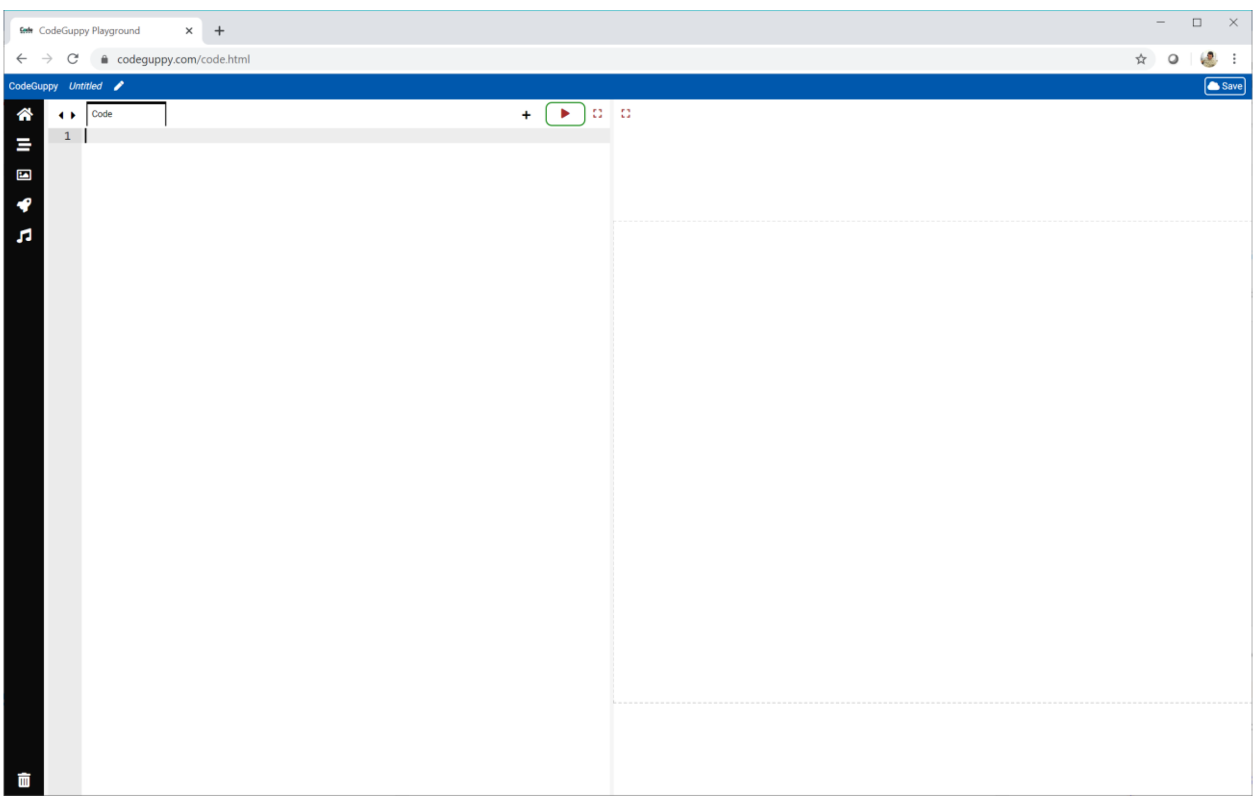
- learn about **loops inside loops**
- see how these **nested loops** can be really useful

Start a New Program

Log in to codeguppy.com if you haven't already.

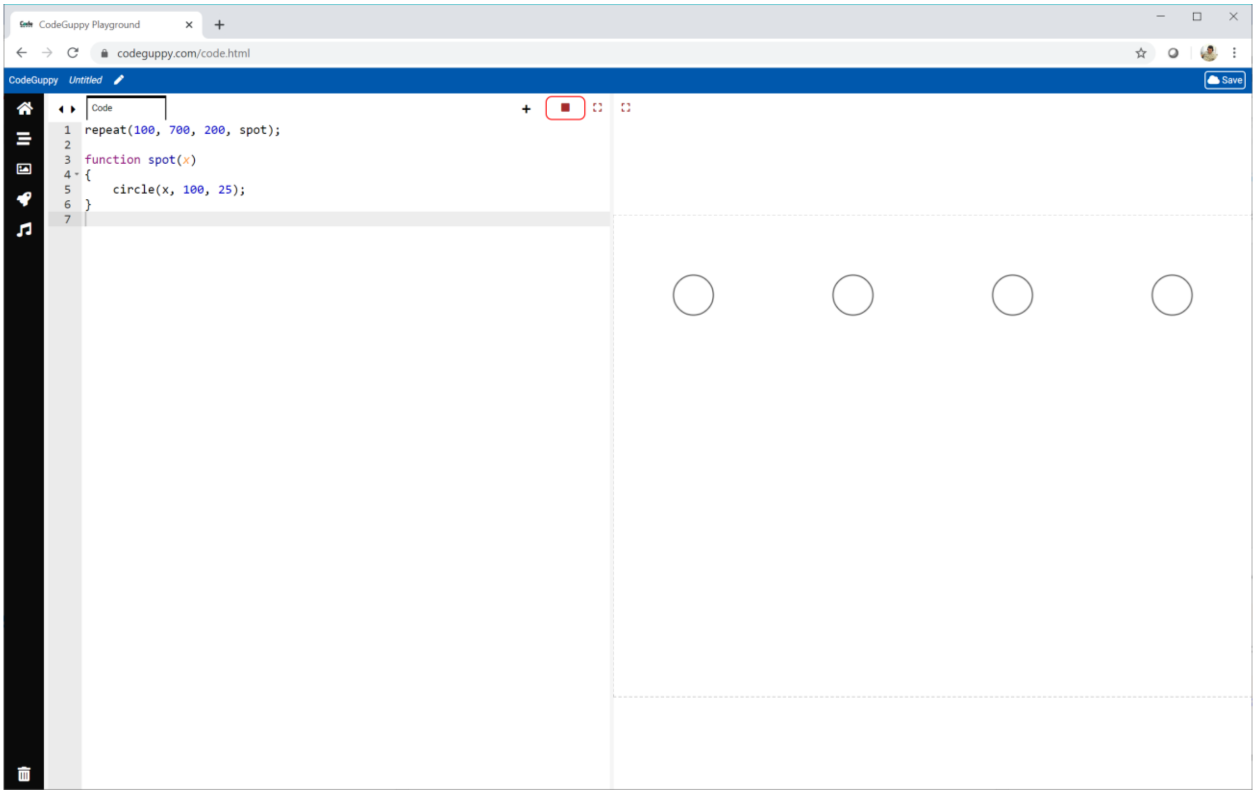
Create a new program, just like we did in the first project.

Your code should look like this:



From A Line To A Grid

Have a look at this row of dots.



We know how to draw a row of dots using the **repeat** instruction.

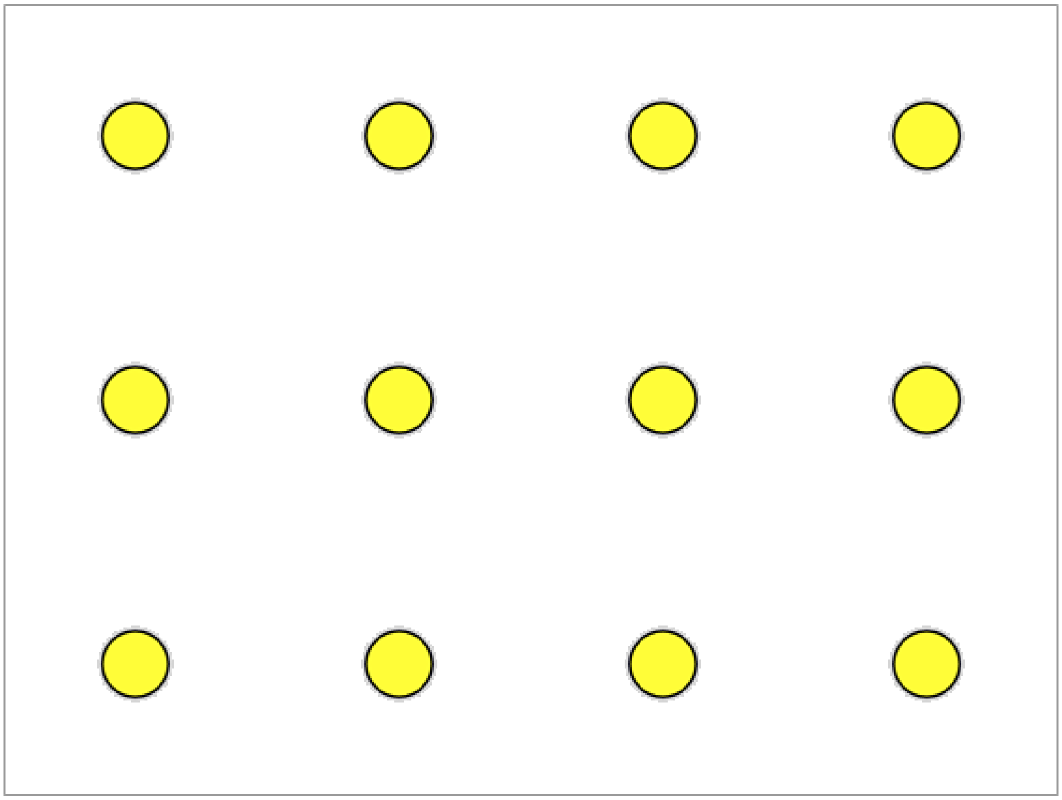
```
repeat(100, 700, 200, spot);
```

The function to draw the spot can be really simple.

```
function spot(x) {
    circle(x, 100, 25);
}
```

Using a **loop** saves us from having to write lots of **circle** instructions.

Now have a look at this grid of spots.



Can we write code to draw this grid of **12** spots without having to write **12 circle** instructions?

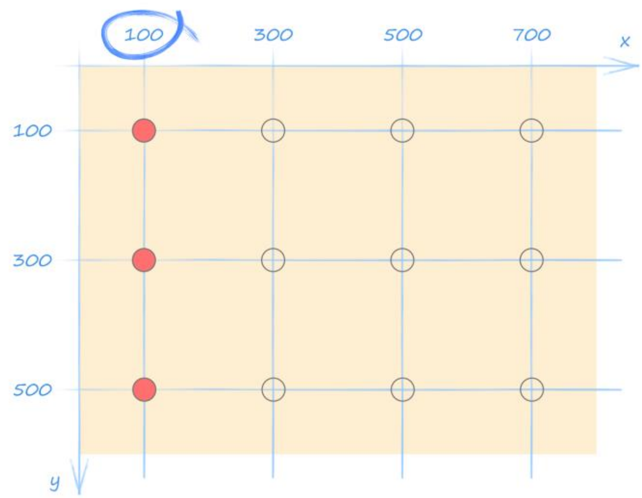
Can loops help us? Have a think before you continue.

The grid has **3** rows. We could write **3** versions of the **spot()** function, one for each row. That would work. But it is still a lot of code to write.

There must be a better way?

There is!

Have a look at this picture.



Along the top you can see the **x** coordinates **100**, **300**, **500** and **700**.

The code **repeat(100, 700, 200, spot)** we used to draw a row of spots has a counter which goes through the same numbers **100**, **300**, **500** and **700**.

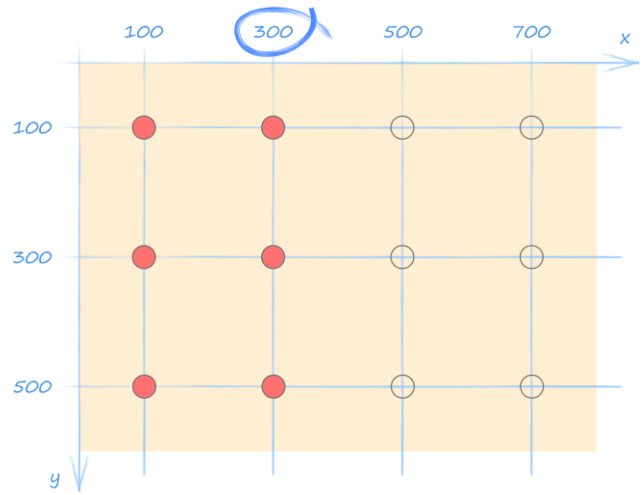
The first time round the loop, the counter is **100**. We could draw the first column of spots which all have **x** as **100**.

To draw that column … we could use another loop! The counter for that loop would count the **y** coordinates **100**, **300** and **500**.

Stop and think about what we just said. The first time round the main loop that goes across the canvas, we want a new loop that goes down the canvas.

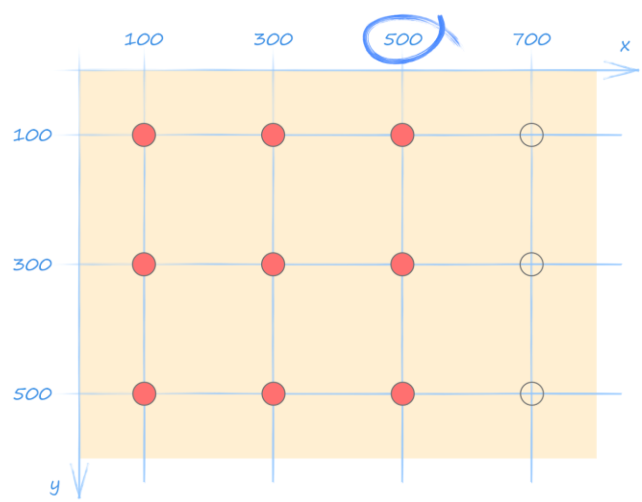
A loop inside a loop!

The second time round the main loop, the counter is **300**. We can draw the column of spots which all have **x** as **300**. The following picture shows this.



To draw the column we can use a loop that counts the **y** coordinates **100**, **300** and **500**. That's the same inner loop as before.

The third time round the main loop, we have the same inner loop again.



We've spotted the pattern. The inner loop is always the same every time we go around the outer loop.

Let’s say that another way.

For every count of the outer loop, we run the same inner loop.

These are called **nested loops**. The **inner loop** is nested inside the **outer loop**.

The **repeat** instruction we’ve been using can do **nested loops** too!

```
repeat(100, 700, 200, 100, 500, 200, spot);
```

The first three numbers **100, 700, 200** are the start, end and step size for the **outer loop** counter.

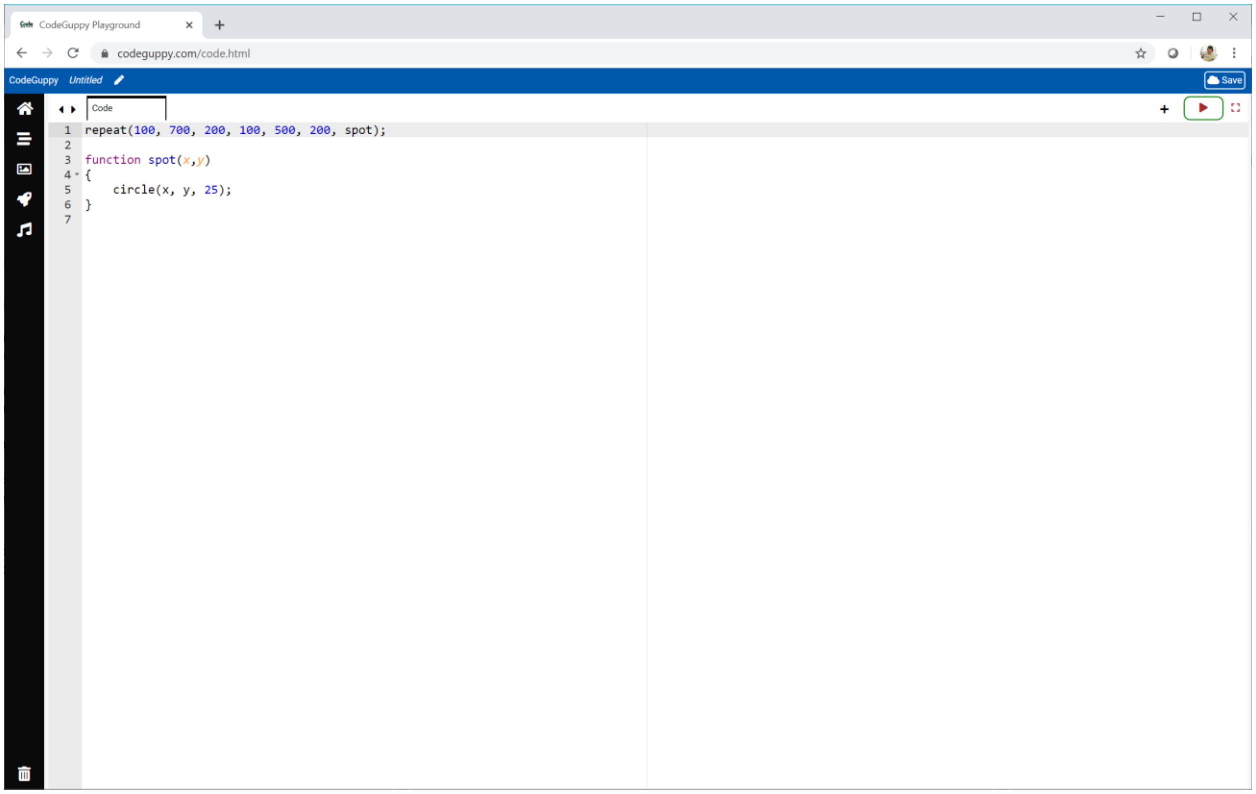
The next three numbers **100, 500, 200** are the start, end and step size for the **inner loop** counter.

The **spot** function needs to take both counters as parameters.

```
function spot(x,y) {
    circle(x, y, 25);
}
```

You can see the **spot** function takes the two counters as **x** and **y**, and draws a circle at **(x, y)**.

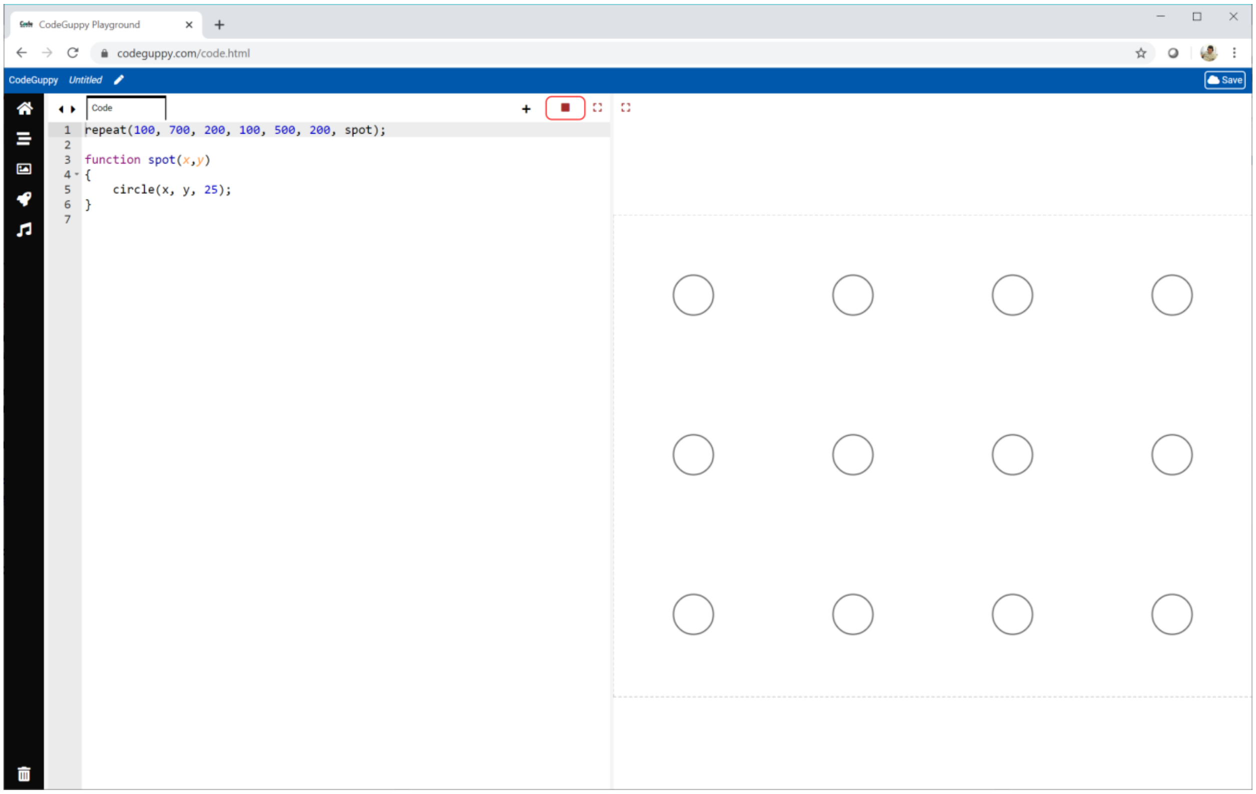
Our code should look like this:



That’s a tiny amount of code for drawing **12** spots.

That shows how powerful **nested loops** are.

Run the code to check that it does draw a grid of **12** spots.

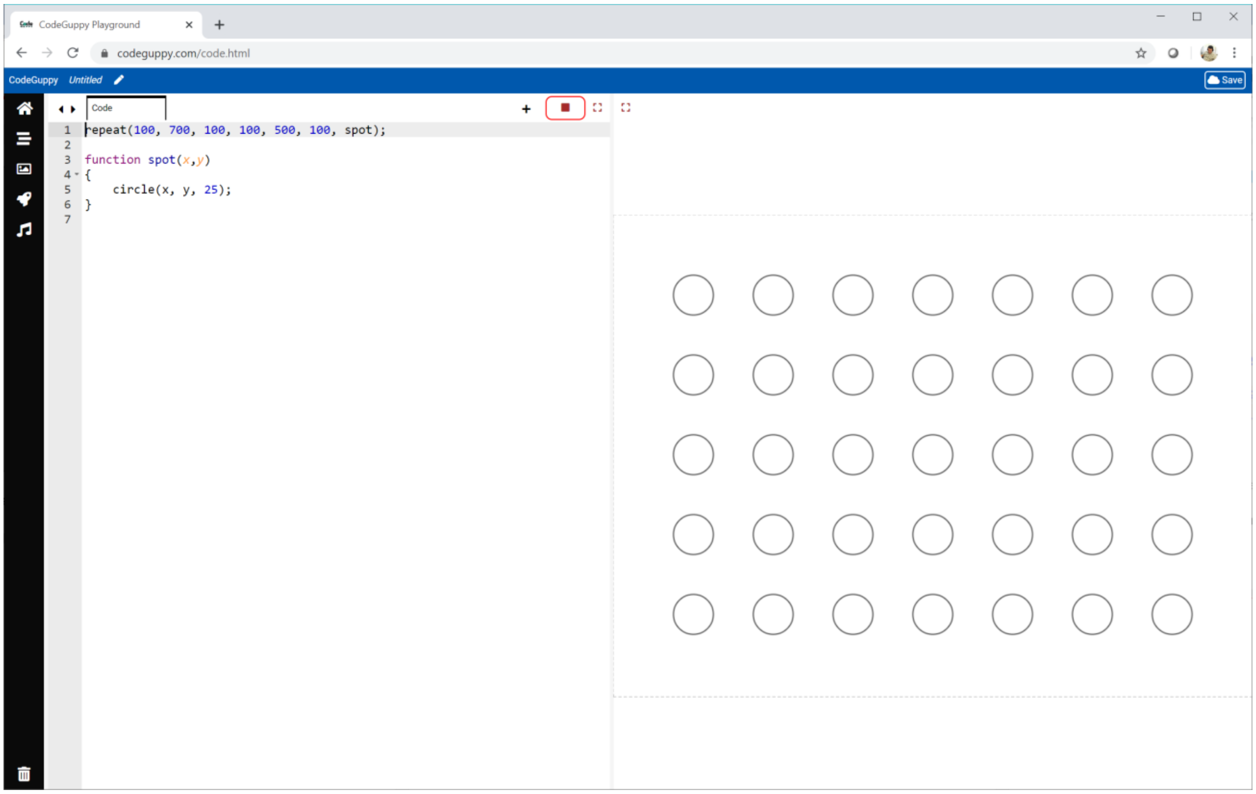


That worked.

Now let’s change the code so the loop counters go up in steps of **100**, not **200**.

```
repeat(100, 700, 100, 100, 500, 100, spot);
```

We should get more spots being drawn.



That’s a grid of **35** spots, and the amount of code is still tiny.

Nested loops are very powerful!

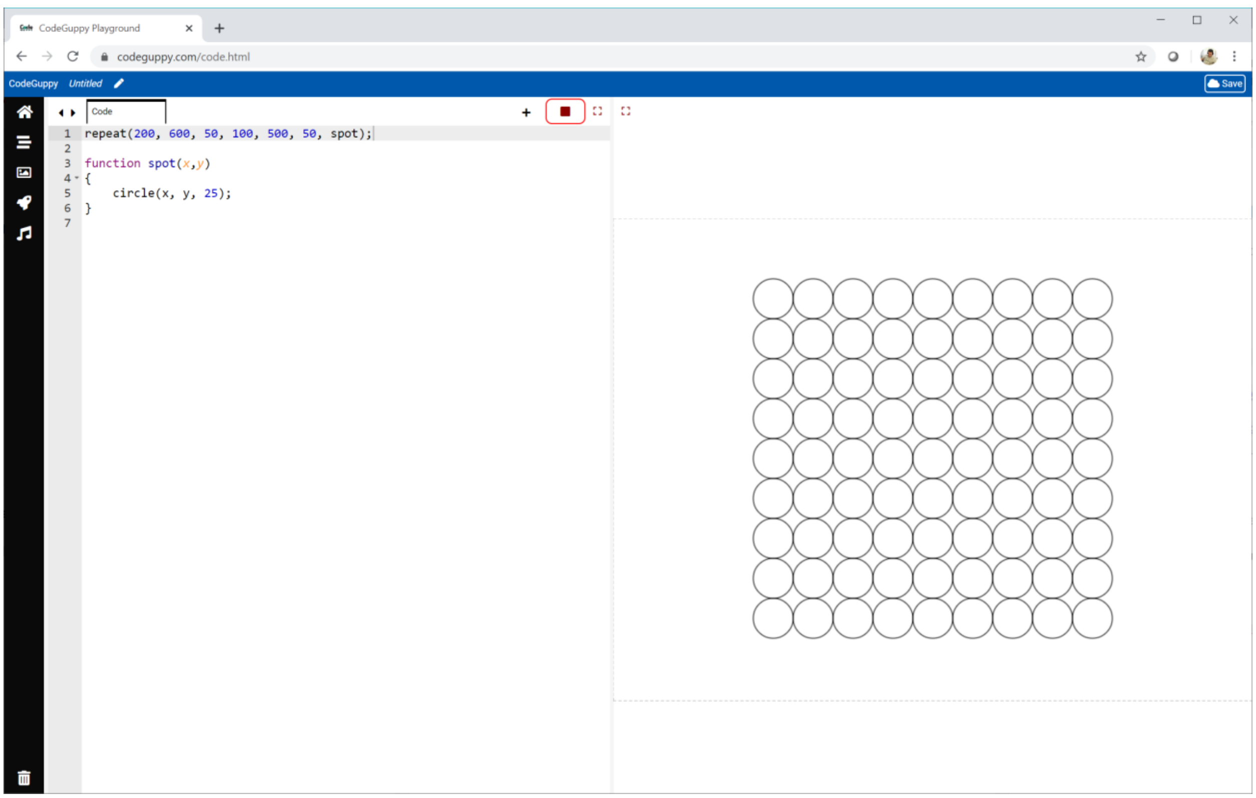
Try It Yourself

Experiment with the **inner loop** and **outer loop** to see how they they change the picture that's drawn.

Here is my own experiment.

```
repeat(200, 600, 50, 100, 500, 50, spot);
```

Here's the result.



That's **81** spots from a tiny amount of code.

Using Nest Loop Counters

Our **spot()** function uses the two parameters to draw a circle at **(x, y)**.

Let’s use those parameters in a new way.

Have a look at this code.

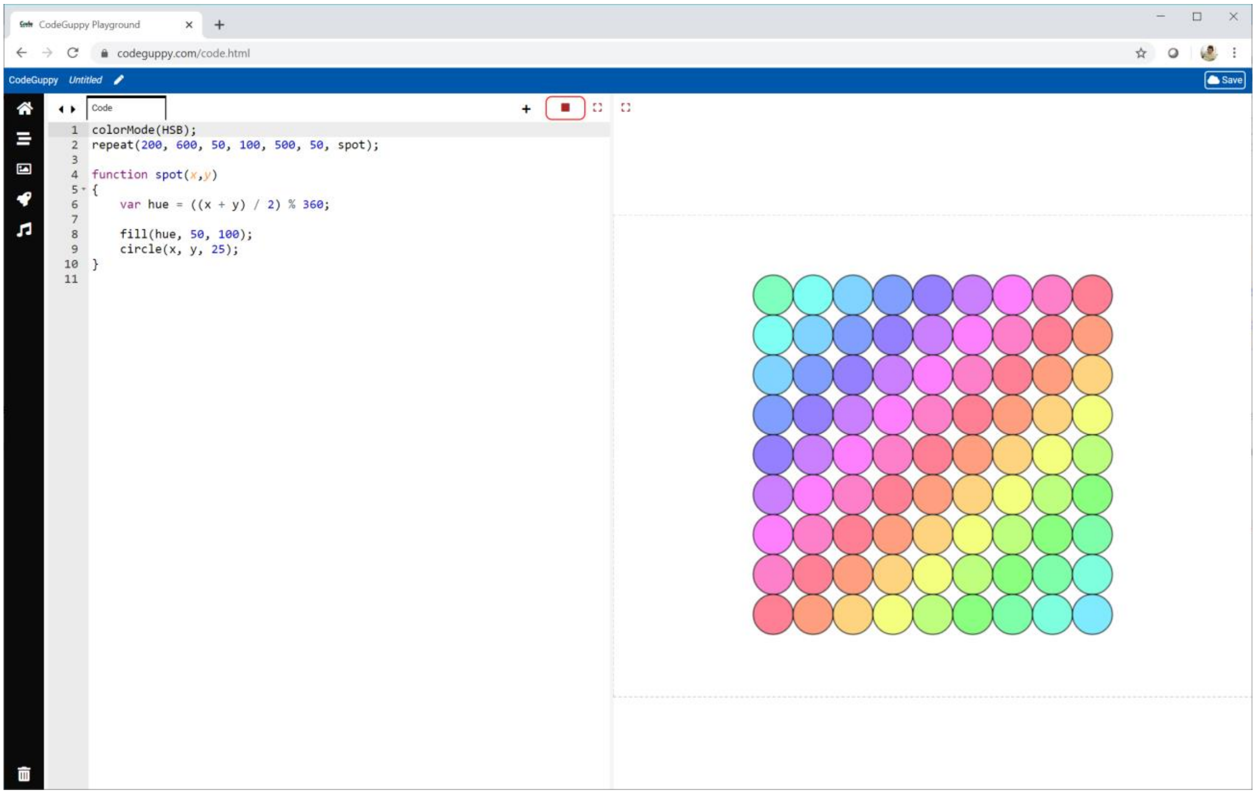
```
function spot(x, y) {

    var hue = ((x + y) / 2) % 360;

    fill(hue, 50, 100);
    circle(x, y, 25);

}
```

You can see the parameters are used to calculate a **hue**. The two are added together and the sum divided by two. The result is divided by **360** and the remainder becomes **hue**.

You might remember **% 360** means the remainder after dividing by **360**.

Don’t forget to change the colour mode to **HSB**.

Let’s see the result.

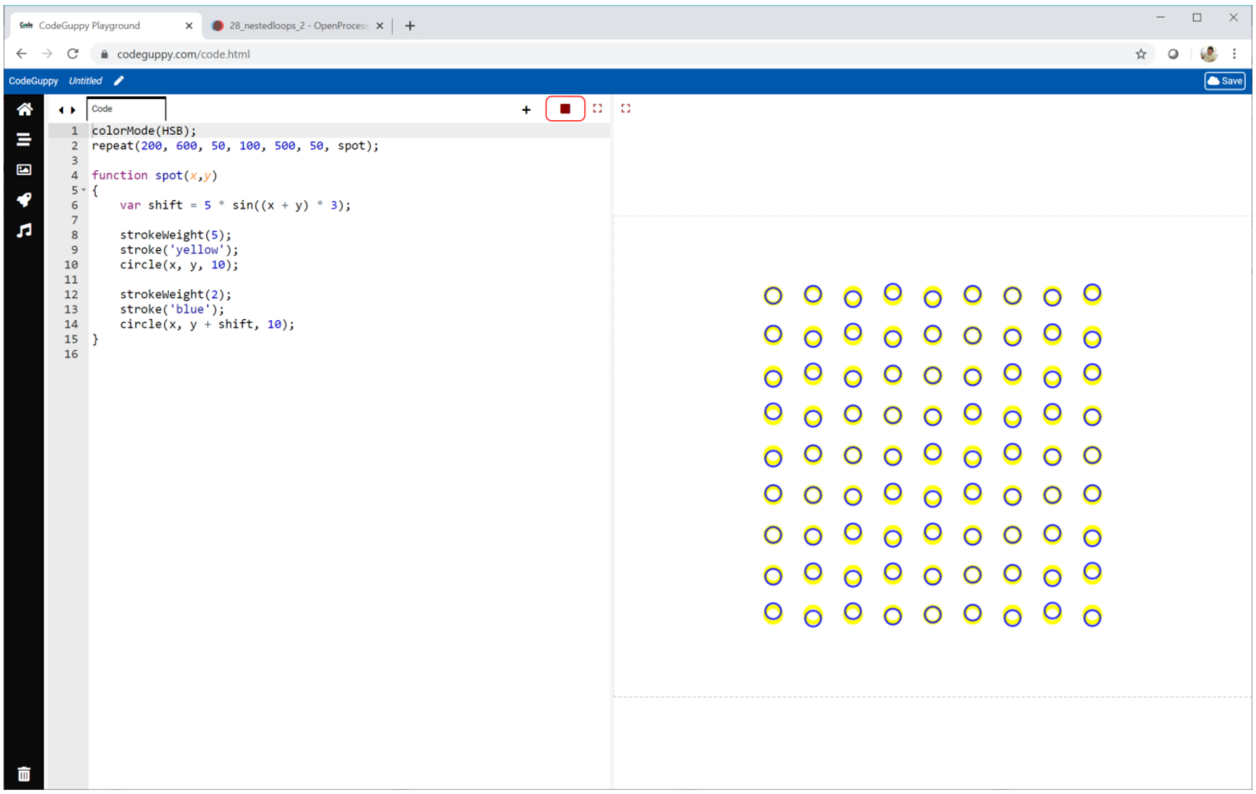

CodeGuppy Playground
codeguppy.com/code.html
CodeGuppy
Untitled
Save
Code
colorMode(HSB);
repeat(200, 600, 50, 100, 500, 50, spot);
function spot(x,y)
{
var hue = ((x + y) / 2) % 360;
fill(hue, 50, 100);
circle(x, y, 25);
}

Cool!

Try It Yourself

Try using the **inner loop** and **outer loop** counters in a new way.

Here's my own experiment.



It's a grid of yellow and blue circles, but the blue circles are shifted up and down a little bit.

That shift is calculated using a sine function to get a wavy effect.

```
var shift = 5 * sin((x + y) * 3);
```

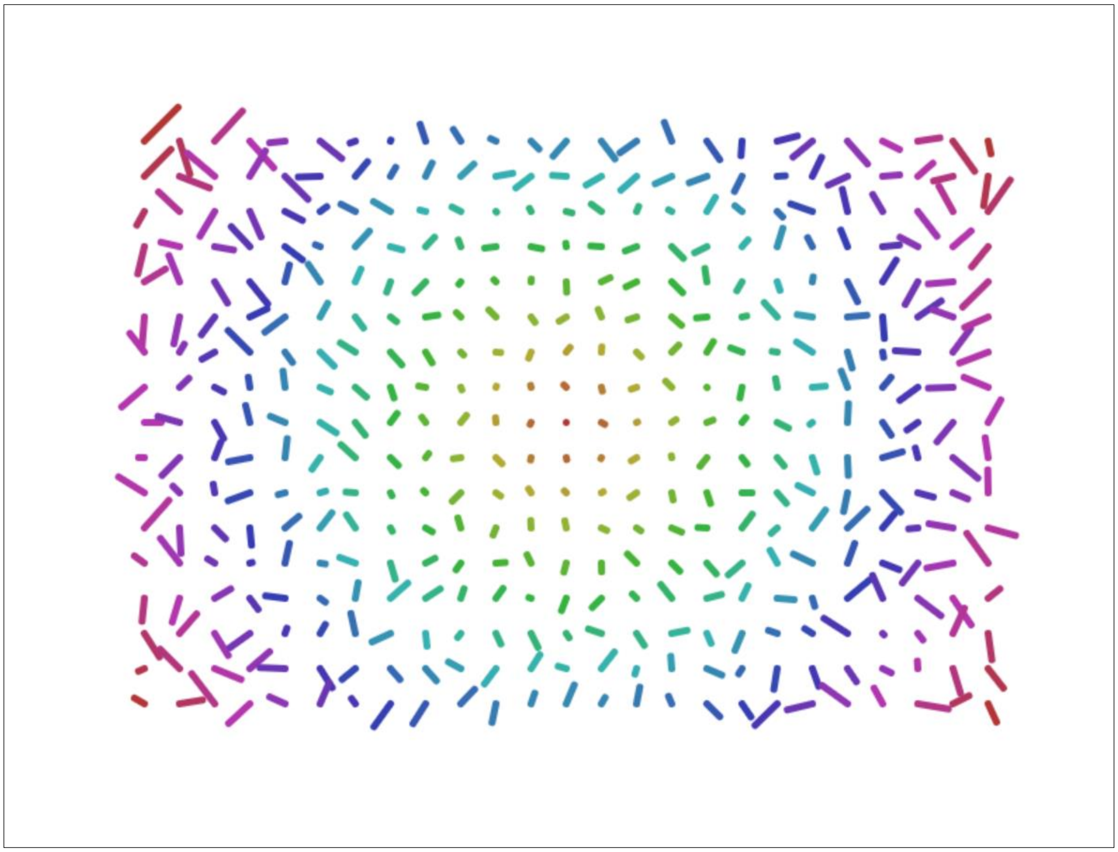
If you want to look at the code, it's online:

- https://codeguppy.com/code.html?tariq/ex03


you've learned to use nested loops - a really powerful technique
well done!

Challenge!

Have a look at this picture.



See if you can write code that creates a similar picture.

The pattern is based on a grid. Horizontally it goes from **100** to **700** in steps of **25**. Vertically it goes from **100** to **500** in steps of **25**.

At each point on the grid is a coloured **line**.

The **HSB hue** of a line is decided using the distance from the centre of the canvas. This is why lines that are the same distance from the centre have the same colour.

The other end of a line has coordinates that are **random**, but chosen within a range again decided by the distance from the centre.

This is why the lines are longer further away from the centre of the canvas, and shorter closer to the centre.

If you need to look at my code, it is online:

- https://codeguppy.com/code.html?tariq/ex04

2.9 - See-Through Colour

level

What We'll Do

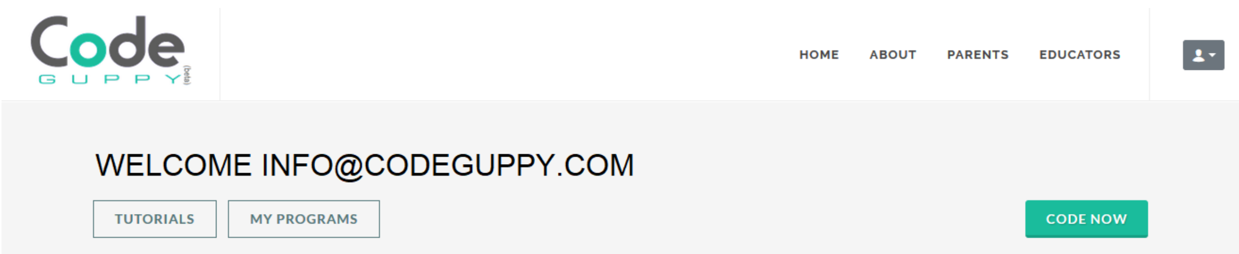
In this project we're going to:

- learn about making colours **translucent**
- see how translucency can help make **busy designs** work better

Start a New Program

Log in to codeguppy.com if you haven't already.

Create a new program, just like we did in the first project by clicking on the Code Now button:



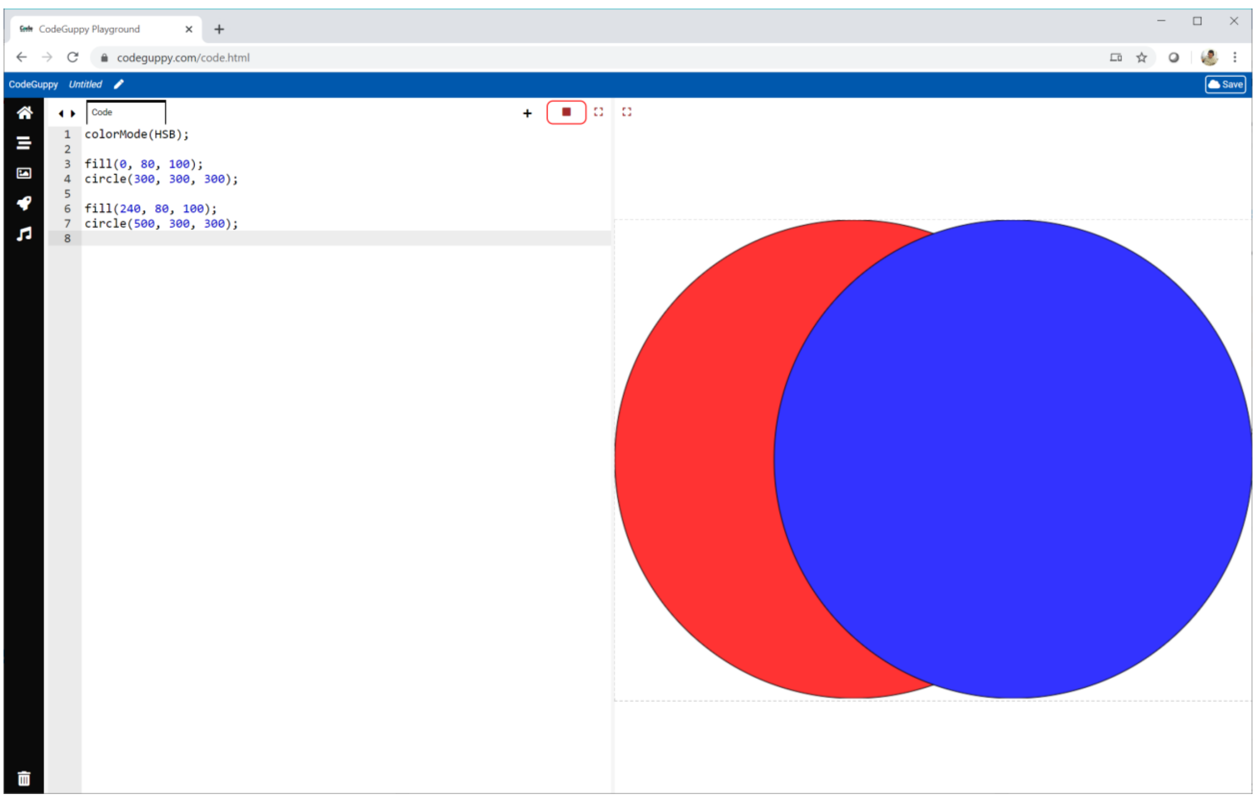
Overlapping Circles

Here is some simple code which draws two overlapping circles.

```
fill(0, 80, 100);
circle(300, 300, 300);

fill(240, 80, 100);
circle(500, 300, 300);
```

The color mode is **HSB** so the colours are red and blue.

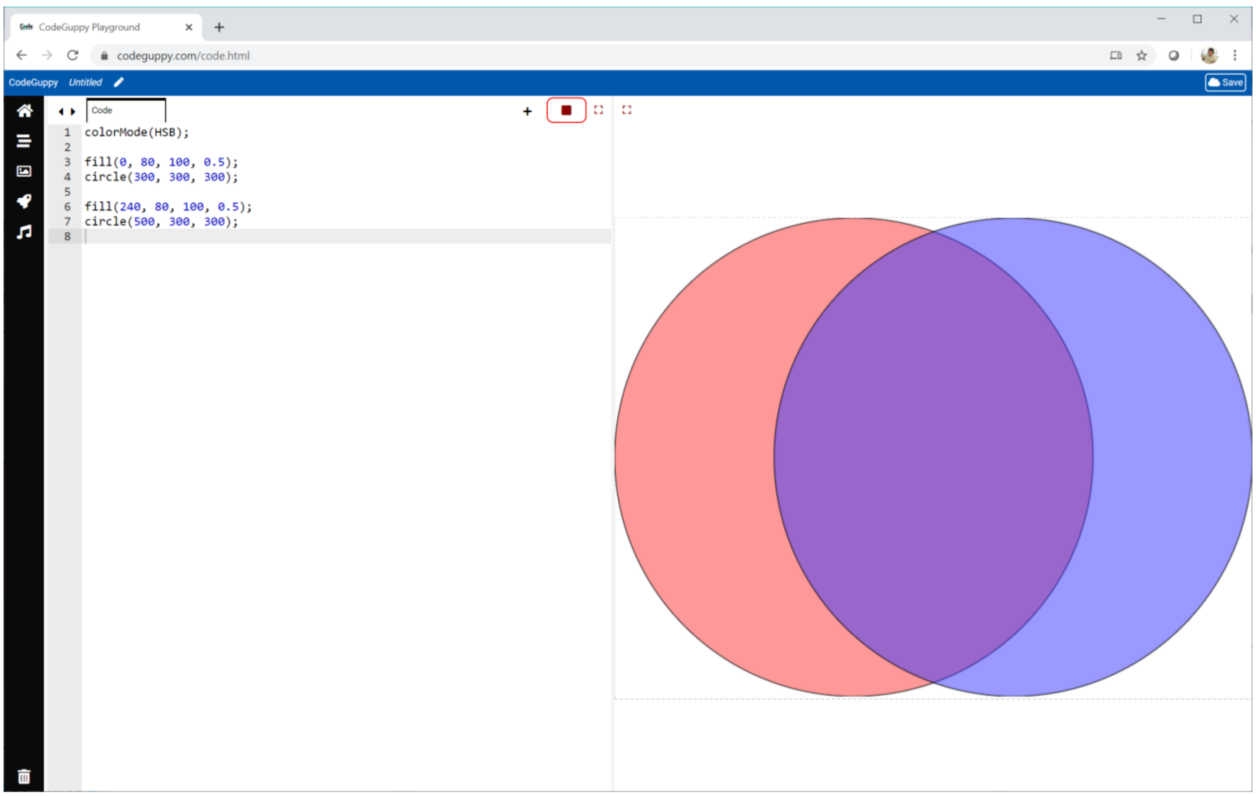
Here's what the code draws.



Now have a look at this code which adds an extra number **0.5** to the **fill** instructions.

```
fill(0, 80, 100, 0.5);
circle(300, 300, 300);

fill(240, 80, 100, 0.5);
circle(500, 300, 300);
```

Before we talk about what this extra number does, let’s see what the code draws.



Can you work out what the extra number **0.5** does?

That number seems to make the colours see-through.

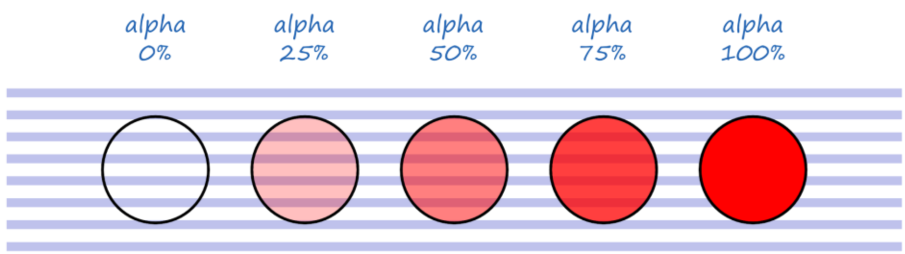
Making Colours See-Through

That extra number decides how see-through a colour is.

The number is called an **alpha** value, and goes from **0** to **1** if we're using **HSB** color mode.

If we use the **RGB** colour mode, **alpha** values go from **0** to **255**.

This picture shows how a circle becomes less see-through the higher the **alpha** value is.



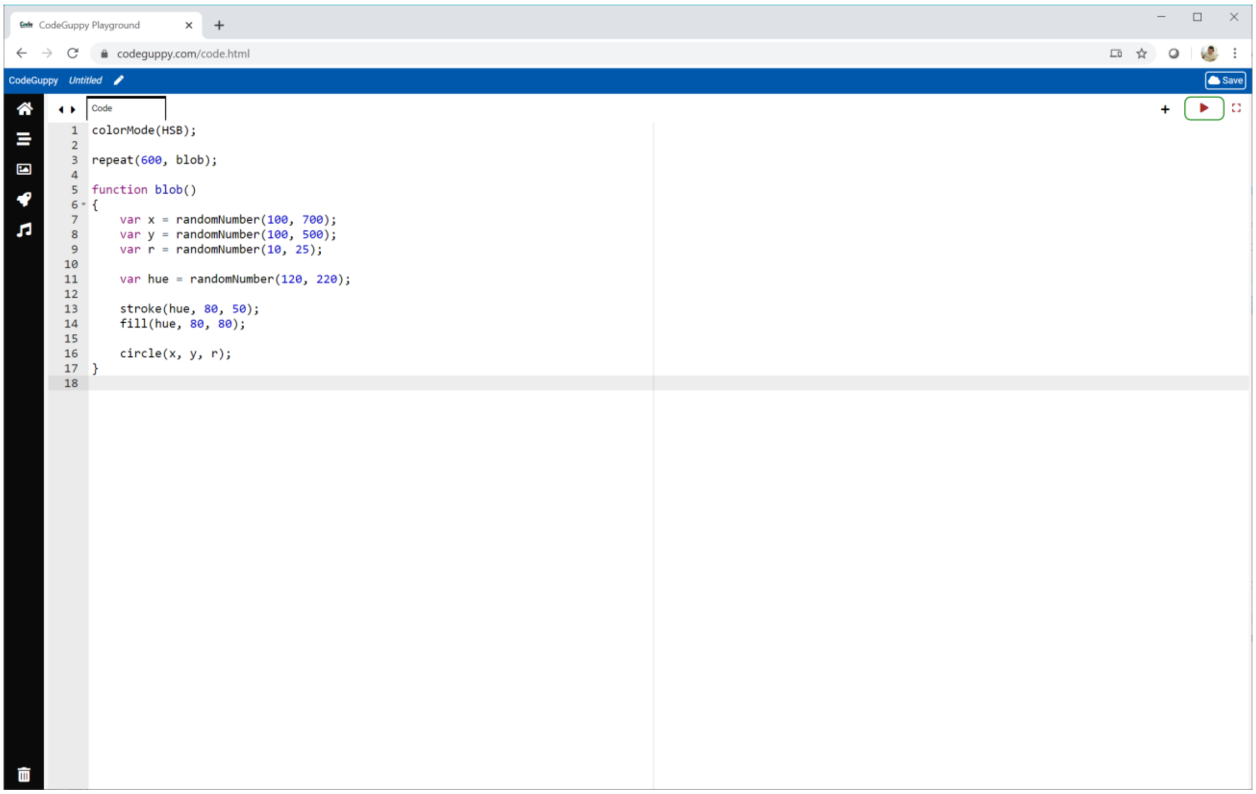
The correct word for this kind of see-through is **translucency**.

The first four circles in the picture above are **translucent**.

Some people use the word **transparent**. If we want to be really correct **transparent** means we can't see any colour at all. So only the first circle in that picture is **transparent**.

Translucency Helps Busy Pictures

The following code draws **600** circles at random locations on the canvas. The circles have a random size chosen between **20** and **50**.



The circles also have a **hue** chosen randomly between **120** and **220**.

This **hue** is used for the fill colour and the outline stroke too, with the outline being a bit darker.

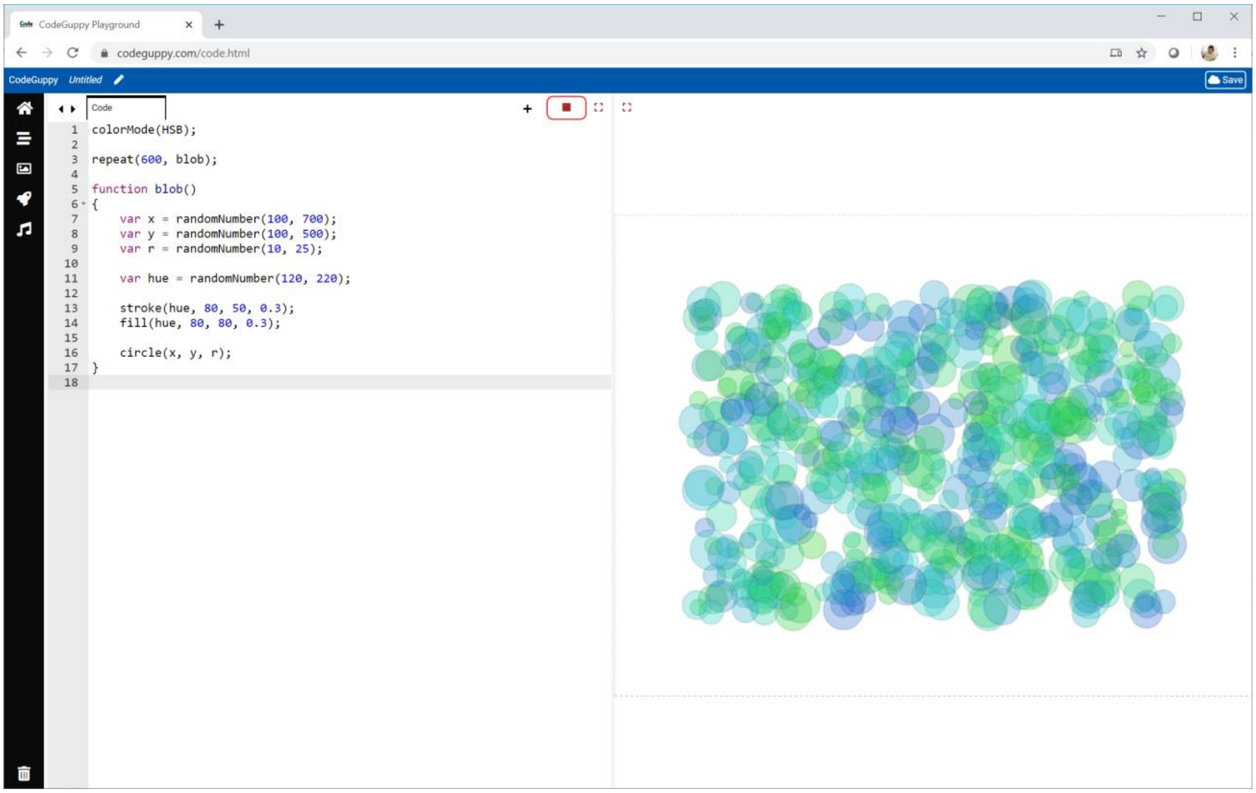
Let's see what the code draws.

That’s a very busy picture. It’s almost too busy.

Now let’s make the colours **translucent** with an **alpha** value of **0.3**.

```
stroke(hue, 80, 50, 0.3);
fill(hue, 80, 80, 0.3);
```

Let’s see what difference this makes.



This picture is much easier to look at.

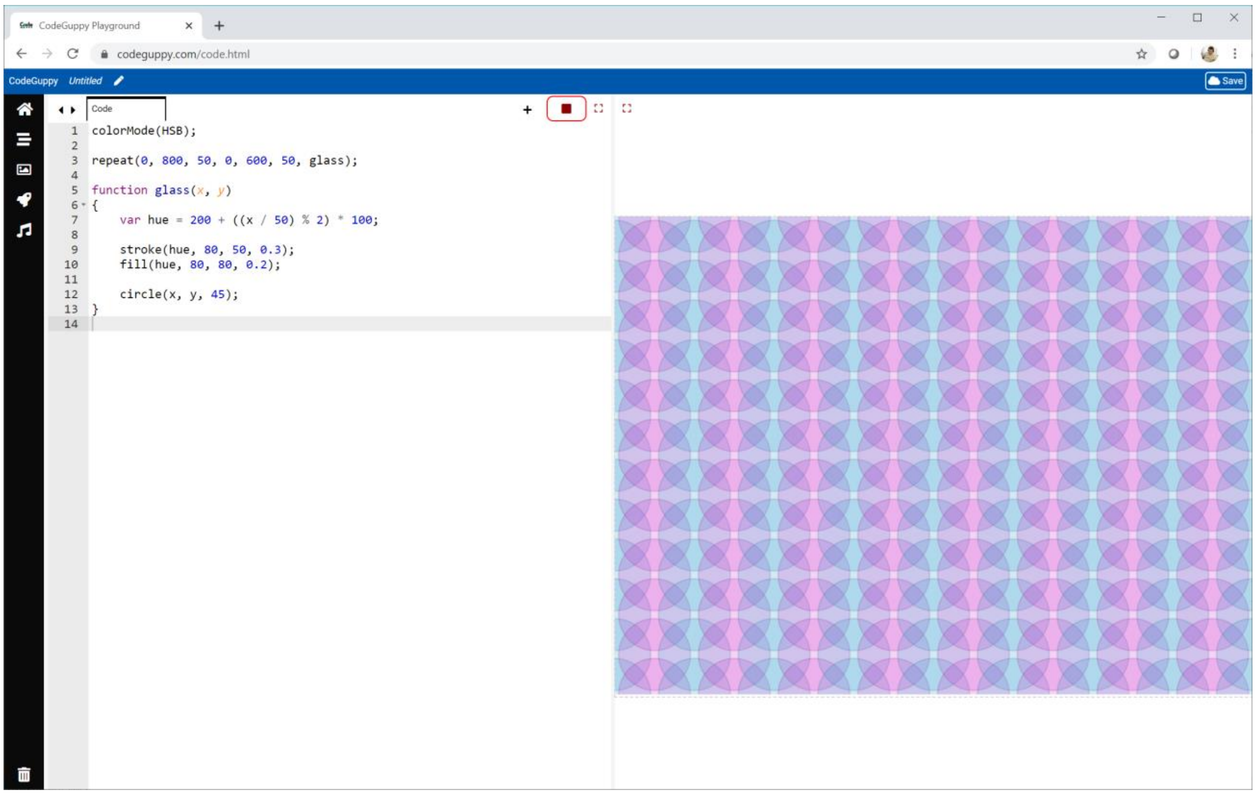
There are still **600** circles, but the detail is easier to explore, and the image feels less heavy.

We’ve seen how translucency can make very busy designs work better.

Try It Yourself

Try making a busy design which works better with translucency.

Here is my own experiment.



I've drawn translucent circles on a grid using a nested loop. The outline strokes are a bit less translucent.

The hue is calculated so the values alternate between **200** and **300** across the canvas.

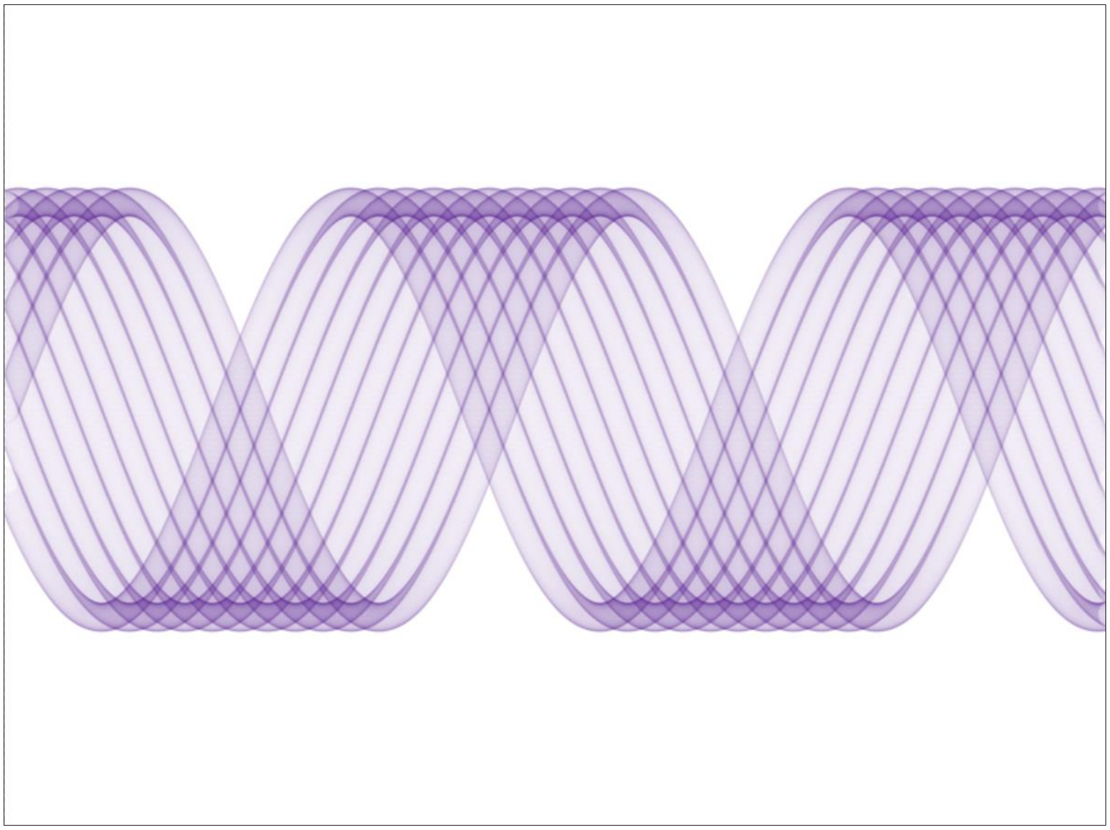
My experiment is online:

- https://codeguppy.com/code.html?tariq/ex05


you've used translucency to make busy designs work better
well done!

Challenge!

See if you can write code to make an image similar to this one.



Although it looks complicated, the image is just made of **sine** waves plotted with circles. Those circles have no fill but do have a translucent outline stroke, with a very low alpha of **0.05**.

A nested loop gives us two separate loop counters. One is used as an **x** coordinate. The other is used to add a bit to **x** before it is fed to the **sine** function, like this **sin(x + t)**.

If you need to look at the code, it is online:

- https://codeguppy.com/code.html?tariq/ex06

Level 3 - Advancing

3.1 - Not So Random Noise

level

What We'll Do

In this project we're going to:

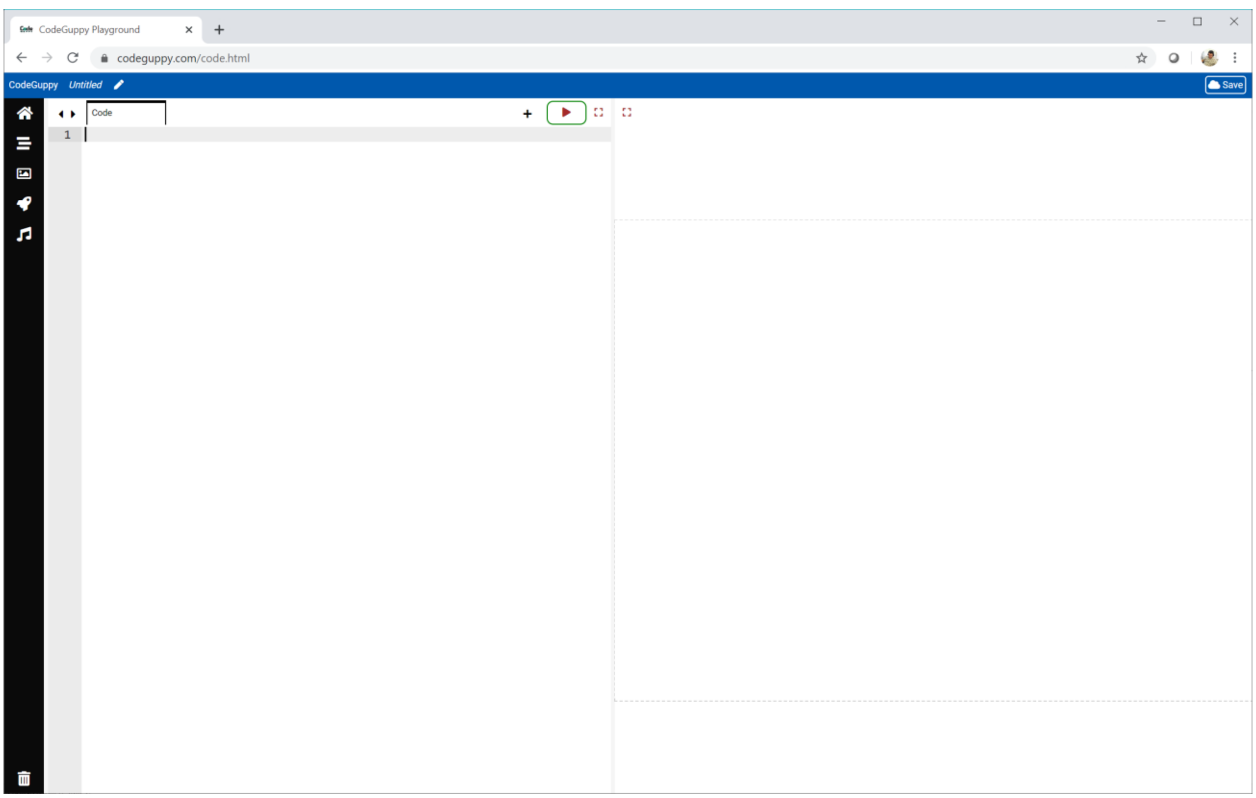
- learn about **not so random noise**
- see how this noise can create quite **natural looking** patterns

Start a New Program

Log in to codeguppy.com if you haven't already.

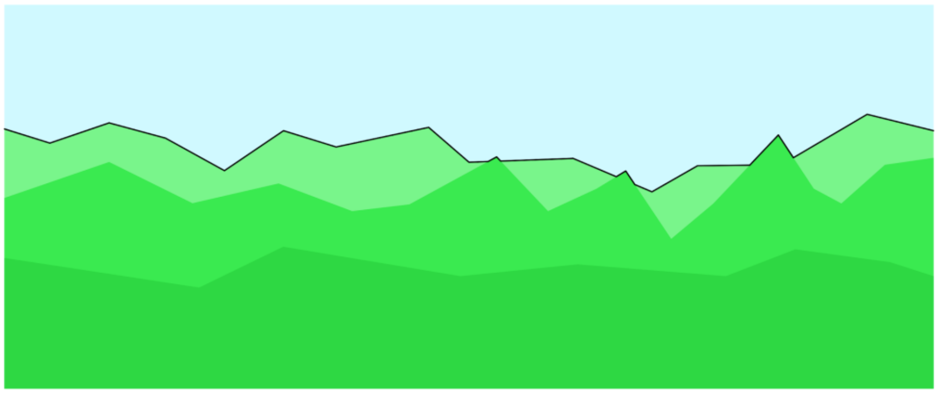
Create a new program, just like we did in the first project.

Your code should look like this:



A Random Horizon

Have a look at this nice view.



The **horizon** is marked with a dark line. The higher the hills, the higher the dark line.

It looks like the horizon goes up and down randomly.

Let’s see if we can make our own horizon using random heights.

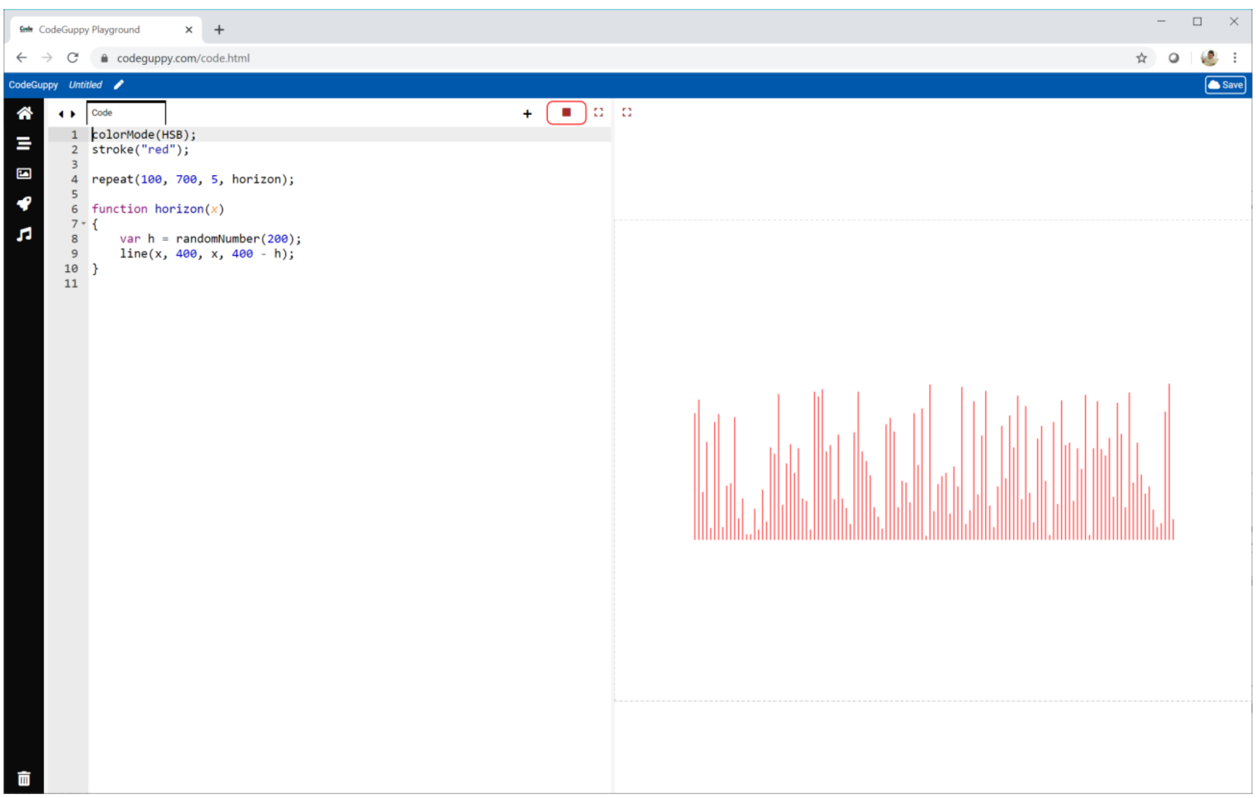
We can create a function to draw vertical lines along the canvas, each with a **random** height.

```
function horizon(x) {
    var h = randomNumber(200);
    line(x, 400, x, 400 - h);
}
```

We can use a **repeat** instruction to count an **x** coordinate from **100** to **700** in steps of **5**.

```
repeat(100, 700, 5, horizon);
```

Here’s what the full code and output looks like.



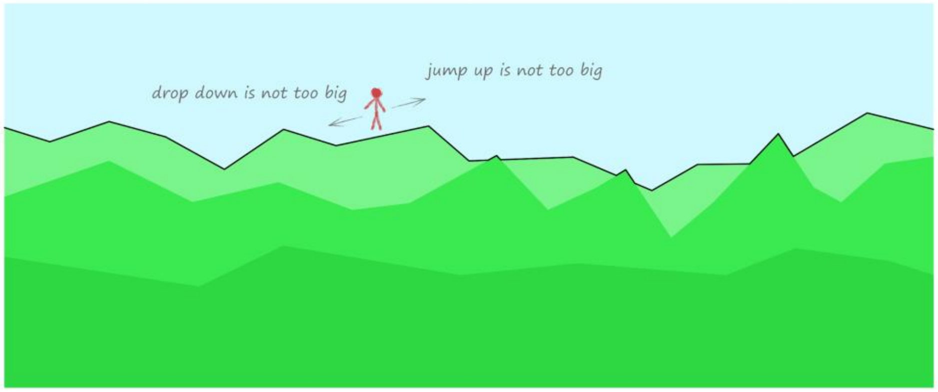
That doesn’t look like a very natural horizon.

Why is this?

Have a think before you continue.

That horizon doesn't look natural because it changes too much.

If you were walking on a real horizon, the ups and downs wouldn't be so sudden and large. The changes would be more gentle.



Let's say this another way. The height at any point is similar to heights near it.

But truly random numbers don't care if they are similar to their neighbours or not.

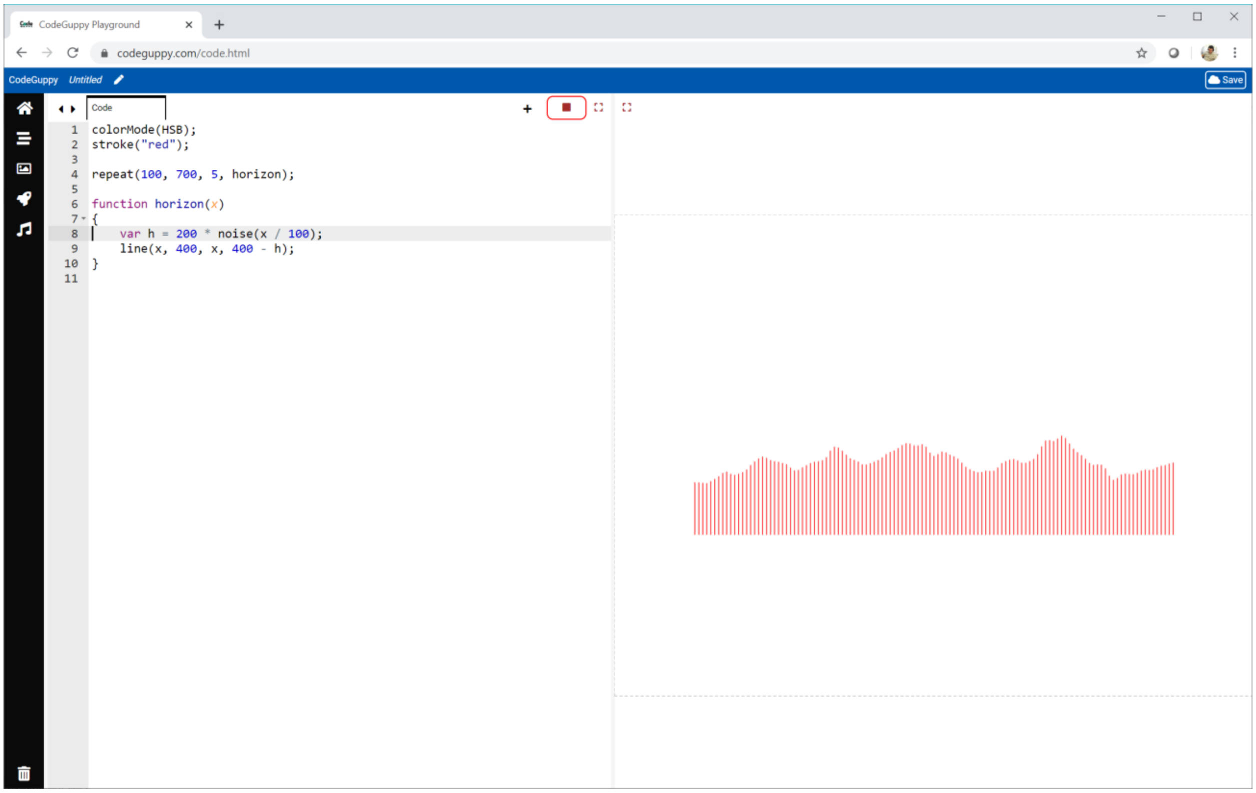
We need numbers that are **not so random** - numbers that change more gently, and are similar to their neighbours.

Luckily there's a function that does this, and it's called **noise()**.

Let's change our code to use **noise()**, instead of **randomNumber()**.

```
h = 200 * noise(x / 100);
```

The **noise()** function gives us a number between **0** and **1**, so we've multiplied it by **200** to match our previous attempt at a horizon.
Let's see if this makes a more realistic horizon.



That’s much better!

You can see the heights are similar if they are close together.

From a distance, the horizon still changes randomly, but does it more gently. And that makes it look more realistic.

This not so random **noise** is sometimes called **Perlin noise**, after Ken Perlin who invented it.

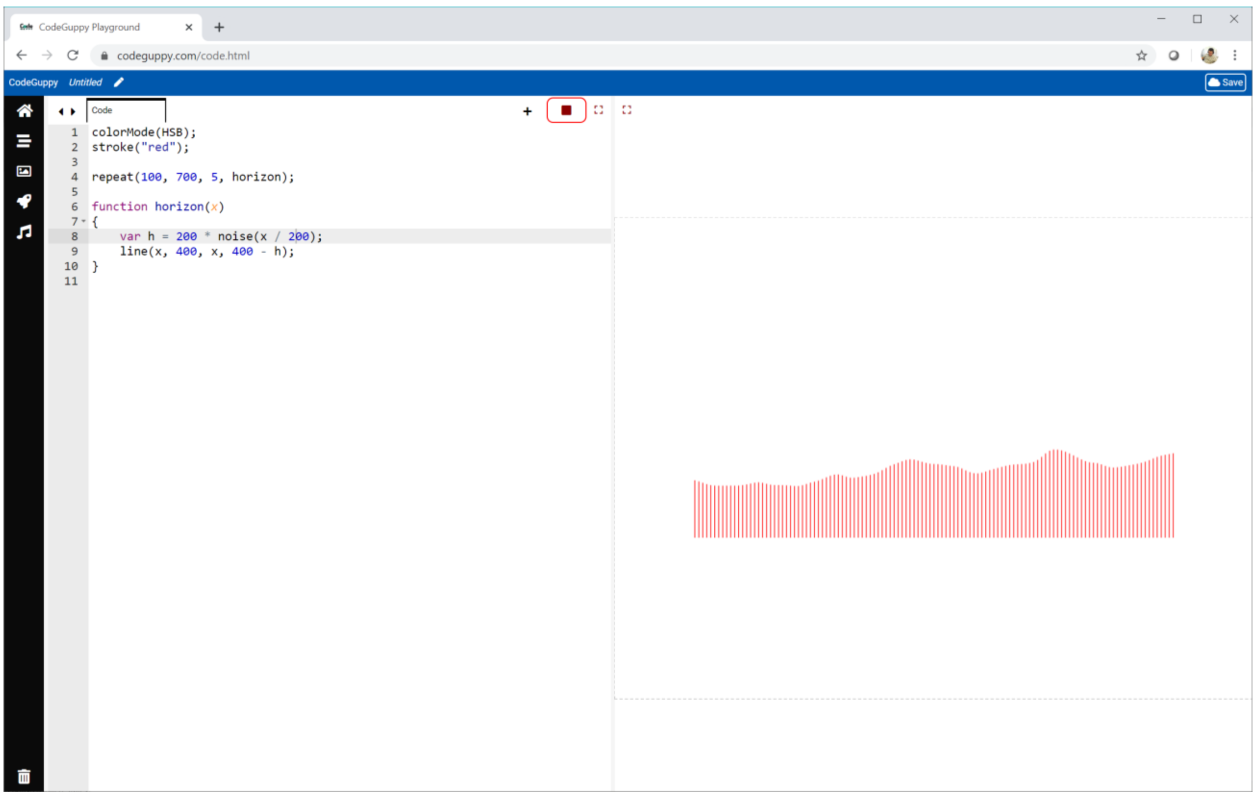
Try It Yourself

That **x** given to the **noise()** function was divided by **100**. See what happens if you change that number.

Here is my own experiment. I've divided **x** by **200**.

```
h = 200 * noise(x / 200);
```

The result is a horizon that changes even more slowly and smoothly.




you've learned about a very useful kind of randomness called noise
well done!

Noise On A Surface

That **noise()** function we used took a single parameter **x / 100**.

```
h = 200 * noise(x / 100);
```

That **x** moves along a horizontal **line**. The parameter tells **noise()** where we are on that line.

What if we wanted to draw noise on a **2-dimensional surface**?

We would need to tell the **noise()** function where we are on that flat surface. We can use **x** and **y** coordinates to do this.

That means providing two parameters to **noise()**.

```
noise(x / 100, y / 100);
```

Luckily, the **noise()** function can take two parameters.

Let’s use a nested loop to move over a grid on the canvas, and use noise to colour the points we visit.

```
repeat(100, 700, 1, 100, 500, 1, noisy_colour);
```

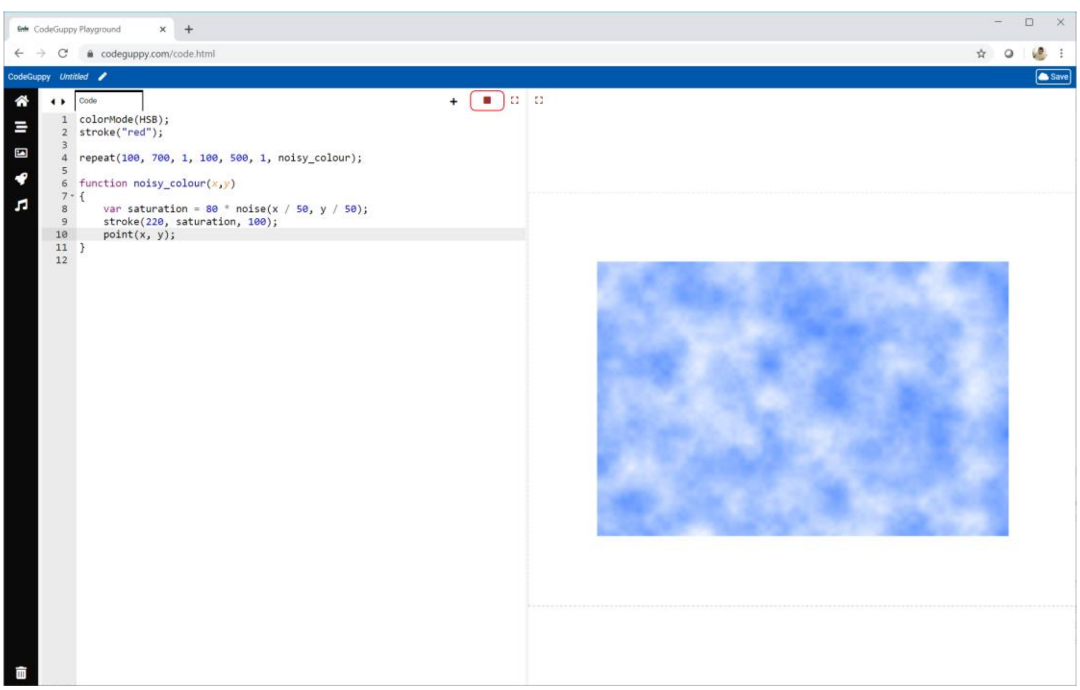
We can feed these loop counters to our **noisy_colour** function.

```
function noisy_colour(x,y) {
     var saturation = 80 * noise(x / 50, y / 50);
     stroke(220, saturation, 100);
     point(x, y);
}
```

You can see that **noise()** now takes two parameters, **x / 50** and **y / 50**.

This noise is used to calculate a **HSB** colour **saturation** for colouring a point at **(x, y)**.

Here’s the full code and the output.



It looks like fluffy white clouds in the sky. And it’s quite realistic too!

It’s interesting to compare this pattern with what pure randomness would make.

```
var saturation = randomNumber(0, 80);
```

See what you get.

Noise is useful for creating **textures**, not just clouds but also wood and stone textures. Noise is even used to create landscapes for planets and games.

Try It Yourself

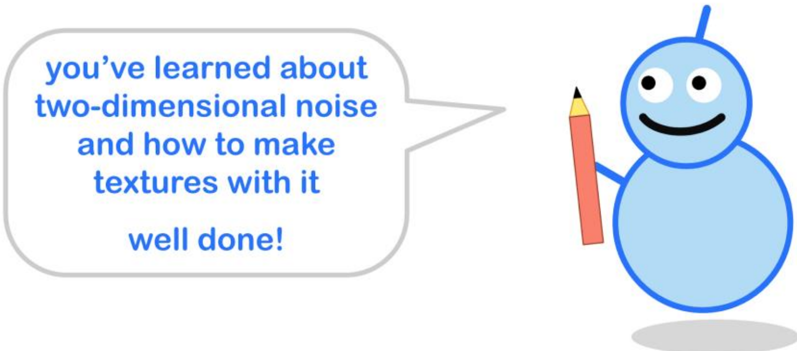
We've used noise to calculate **saturation**. Try using it to calculate **hue**.

Here is my own experiment which scales noise from **0** to **60**.

```
var hue = 60 * noise(x / 50, y / 50);
stroke(hue, 100, 100);
```

And here is the pattern it makes.

That fire looks very hot!


you've learned about two-dimensional noise and how to make textures with it
well done!

Noisy Wobble

We can use noise to add some wobble to the location of shapes before we draw them. We'll explore what this means next.

Have a look at this **repeat** instruction.

```
repeat(100, 700, 5, 100, 500, 1, stripe);
```

The **outer loop** counts from **100** to **700** in steps of **5**. The **inner loop** counts from **100** to **500** in steps of **1**. These two counters are passed to a function **stripe**.

Let's start with a very simple **stripe** function.

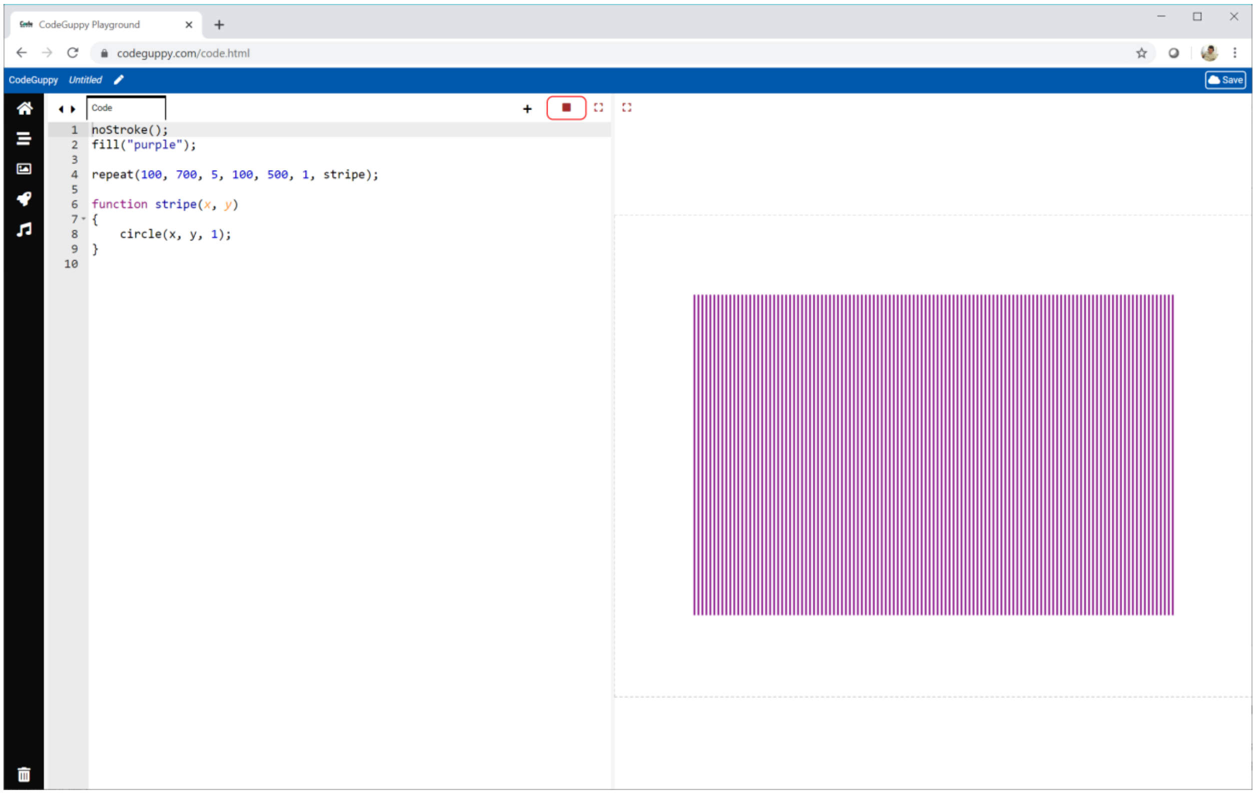
```
function stripe(x, y) {
    circle(x, y, 2);
}
```

Here's the full code.

31_noise_wobble
by CoderDojo Cornwall

```
function setup() {

  simple();

}

function draw() {

  noStroke();
  fill('purple');
  repeat(100, 700, 5, 100, 500, 1, stripe);

}

function stripe(x, y) {

  circle(x, y, 2);

}
```

And here's what that simple code makes.



We see stripes because **x** increases in steps of **5**, and **y** increases in steps of **1**.

Now let's wobble those little circles with some noise!

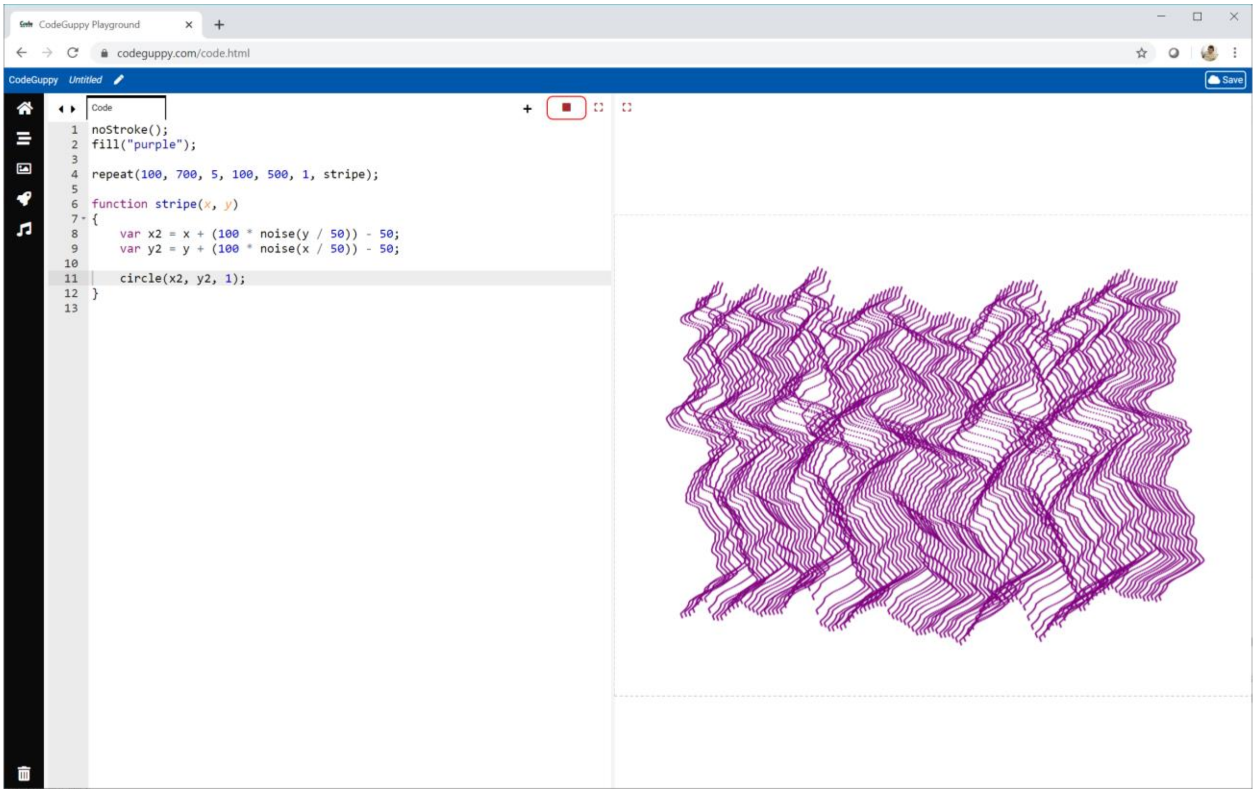
Have a look at this code.

```
var x2 = x + (100 * noise(y / 50)) - 50;
var y2 = y + (100 * noise(x / 50)) - 50;

circle(x2, y2, 2);
```

You can see that noise is added to **x** and the result is put into **x2**. The same thing happens for **y2**, and a small circle is drawn at **(x2, y2)**.

Remember that **noise()** is always between **0** and **1**, so multiplying by **100** gives a value between **0** and **100**. Subtracting **50** gives us a new value between **-50** and **+50**. So **x2** can be anywhere between **x - 50** and **x + 50**.

Let's see the result.



That is a very cool pattern!

Run the code again and you'll see a different pattern.

It's hard to believe that simply adding noise creates such an interesting design.

Look back again at the code which adds noise to **x** and **y**:

```
var x2 = x - 50 + (100 * noise(y / 50));
var y2 = y - 50 + (100 * noise(x / 50));
```

The noise added to **x** depends on **y** not **x**.

The noise added to **y** depends on **x** not **y**.

Why is this? Spend some time thinking about this to see if you can work it out.

Here's the answer.

For each vertical stripe, the value of **x** stays the same. For example, the first stripe has **x = 100**. If **x** stays the same, then **noise(x / 50)** also doesn't change no matter how far up or down the stripe we are.

That means the same amount of noise is added all the way down the stripe. So the whole stripe moves left or right and stays straight.

Try it to see what happens.

```
var x2 = x - 50 + (100 * noise(x / 50));
var y2 = y - 50 + (100 * noise(y / 50));
```

The value of **noise(y / 50)** does change down a stripe. This means the amount of noise added changes down the stripe, and that's what makes an interesting pattern.

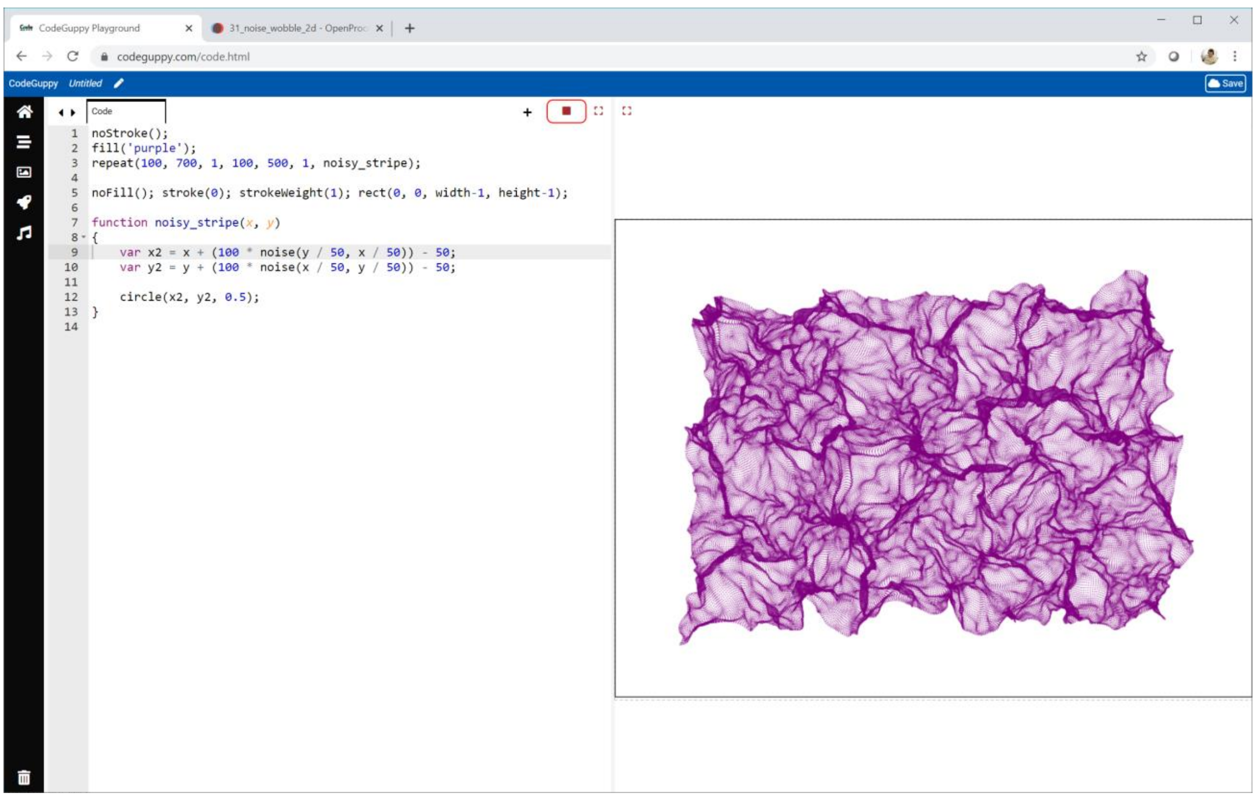
Try It Yourself

Try your own ideas for adding noise to **x** and **y**.

Here is my own experiment using 2-dimensional noise.

```
var x2 = x - 50 + (100 * noise(y / 50, x / 50));
var y2 = y - 50 + (100 * noise(x / 50, y / 50));
```

And this is what it makes. I really like it!



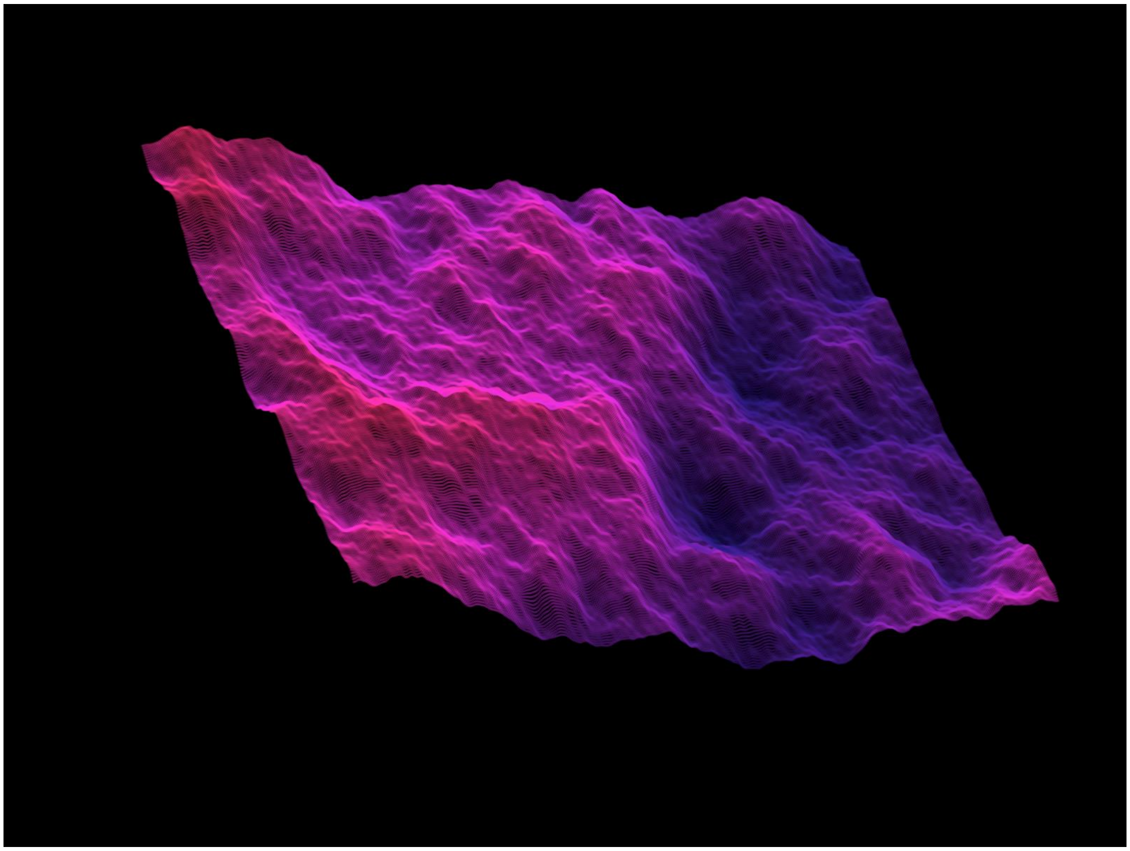
The code for my experiment is online:

- https://codeguppy.com/code.html?tariq/ex07


you've learned how adding noise to a regular pattern makes it very interesting
well done!

Challenge!

Have a look at this spooky alien landscape.



The landscape is made from the same noise we’ve been working with.

Can you write code to draw a similar landscape?

The following information might be helpful:

- The landscape is based on a 2-dimensional **grid**, so you can use a nested loop.

- The **3-d effect** is made by shifting the grid to the right the further down it goes.

- The **height** of the landscape is calculated from 2-d noise. In fact it is the sum of a slow changing noise and a little bit of faster changing noise.

- The **hue** is based on the height, and shifted around the **HSB** colour wheel to get this range from blue to red.

- The **brightness** also depends on the height, and gives the valleys a darker shadow effect.

- Each small circle uses a **translucent** colour to smooth the image.

If you need to look at my code, it is online:

- https://codeguppy.com/code.html?tariq/ex08

3.2 - Moving Around A Circle

level

What We’ll Do

In this project we’re going to:

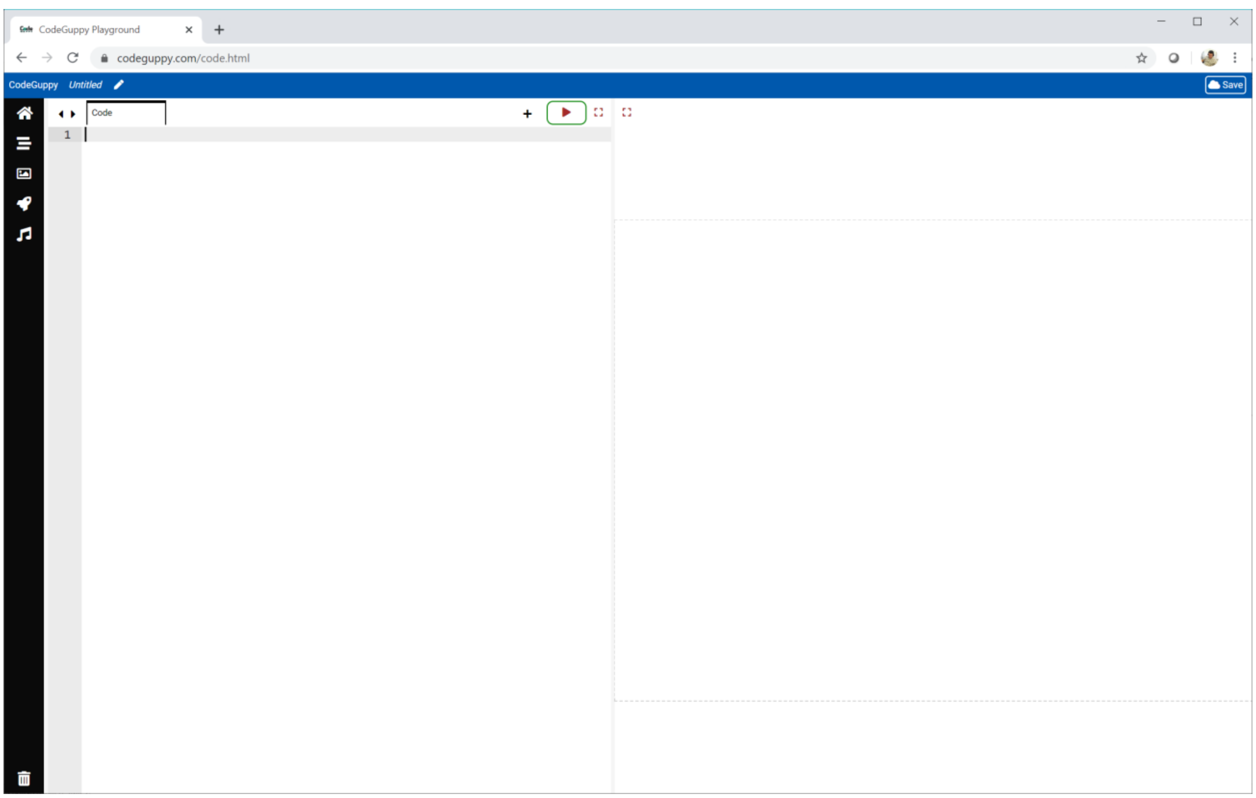
- learn how **trigonometry** can help us move around a circle
- see how we can use it to make **orbital** patterns

Start a New Program

Log in to codeguppy.com if you haven’t already.

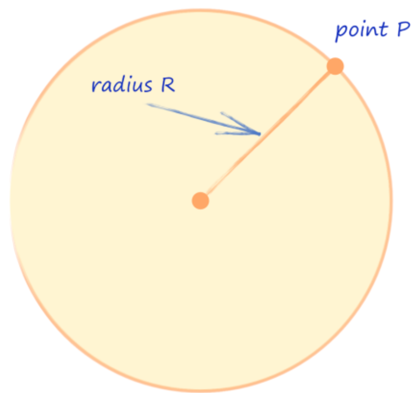
Create a new program, just like we did in the first project.

Your code should look like this:



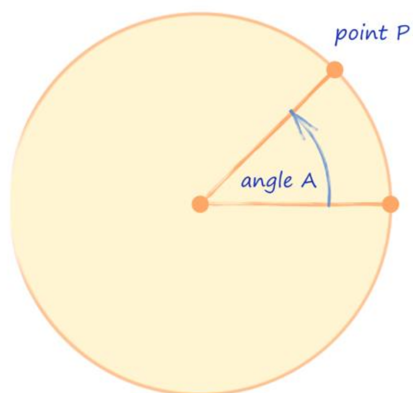
Moving Around a Circle

Have a look at this picture of a point **P** moving around the edge of this circle.



How would you describe where the point **P** is?

An easy way is to use an **angle** like the one shown next.

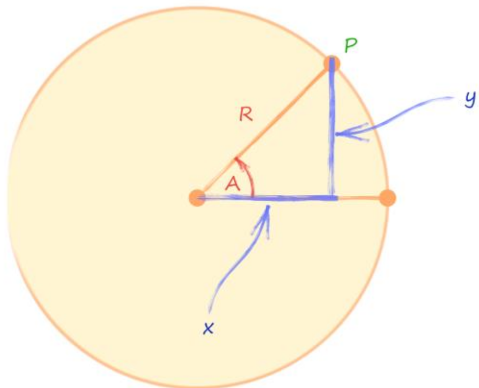


A small **angle** means the point is close to the start.

A larger **angle** means the point has moved further around the circle.

It's easy for us to think about an **angle** to say where **P** is.

But when we’re coding, we need to use **x** and **y** coordinates.

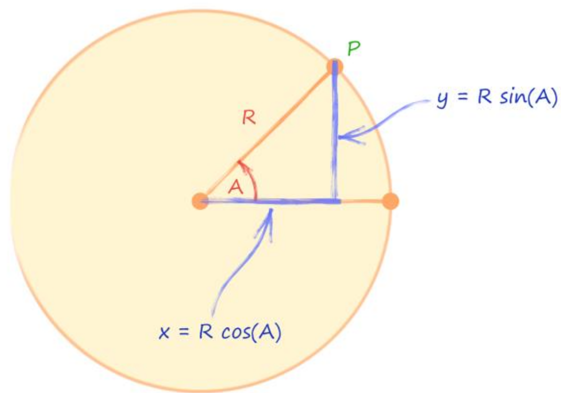


This picture shows both ways of describing where **P** is:

- **P** is at an **angle A**, and at a **distance R** from the centre.
- **P** is at horizontal coordinate **x**, an vertical coordinate **y**.

How do we convert from our **angle** to **x** and **y** coordinates?

Here's the answer.



Let's write those two new things down more clearly.

$$y = R \cdot \sin(A)$$

$$x = R \cdot \cos(A)$$

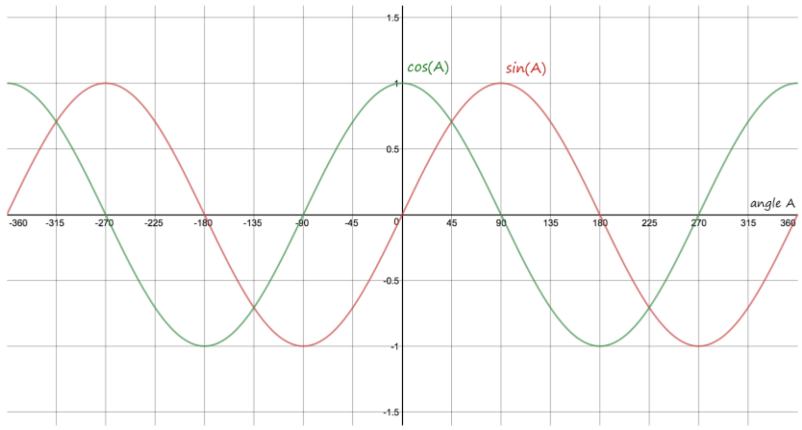
These are called **trigonometric identities.** What a scary name!

Let's look at the first one. It says that to get **y**, we just multiply **R** by the **sine** of **A**. We've seen the wavy **sine** function before.

The second one says that to get **x**, we just multiply **R** by **cos(A)**.

That **cos()** is short for **cosine**. It's a wavy function just like the **sine** function we've already seen, but shifted along a little bit.

Here's a picture of both **sine** and **cosine** functions.



You can see that the two waves are exactly the same, except the **cosine** wave is shifted left.

You might have converted **angles** to **x** and **y** coordinates in your school **trigonometry** classes already.

Enough maths!

Let’s write some code to draw a spot going around a circle.

We can use a **repeat** instruction to count an angle from **0** to **90 degrees**, in steps of **5**.

```
repeat(0, 90, 5, spot);
```

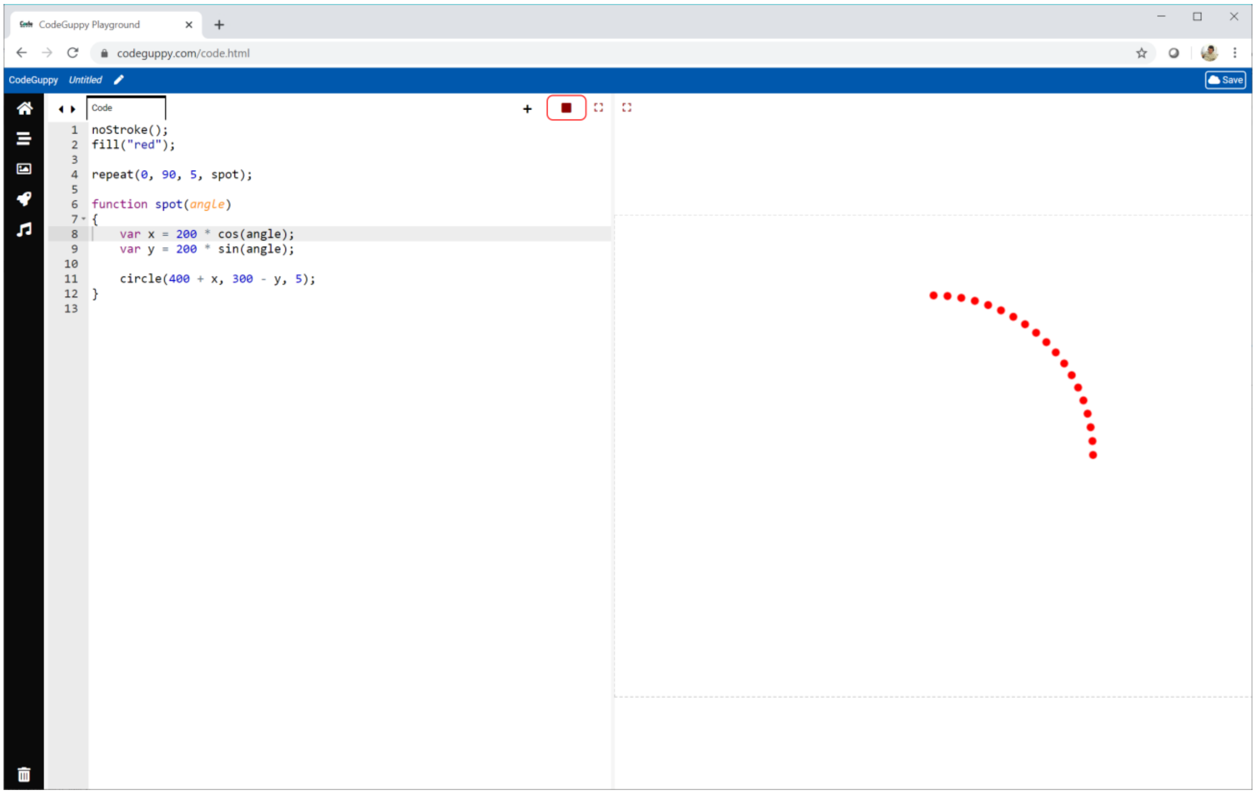
Have a look at this code for the **spot** function.

```
function spot(angle) {

    var x = 200 * cos(angle);
    var y = 200 * sin(angle);

    circle(400 + x, 300 - y, 10);
}
```

The code is pretty simple. All it does is take the **angle** parameter and calculates **x** and **y** from it using those scary-sounding **trigonometric identities**.

That **200** is the radius **R** of the circle we’re moving around.

Here’s the full code and what the code draws:



We can see the dots moving around the circle, from **0 degrees** all the way up to **90 degrees.**

Those **trigonometric identities** really do work!

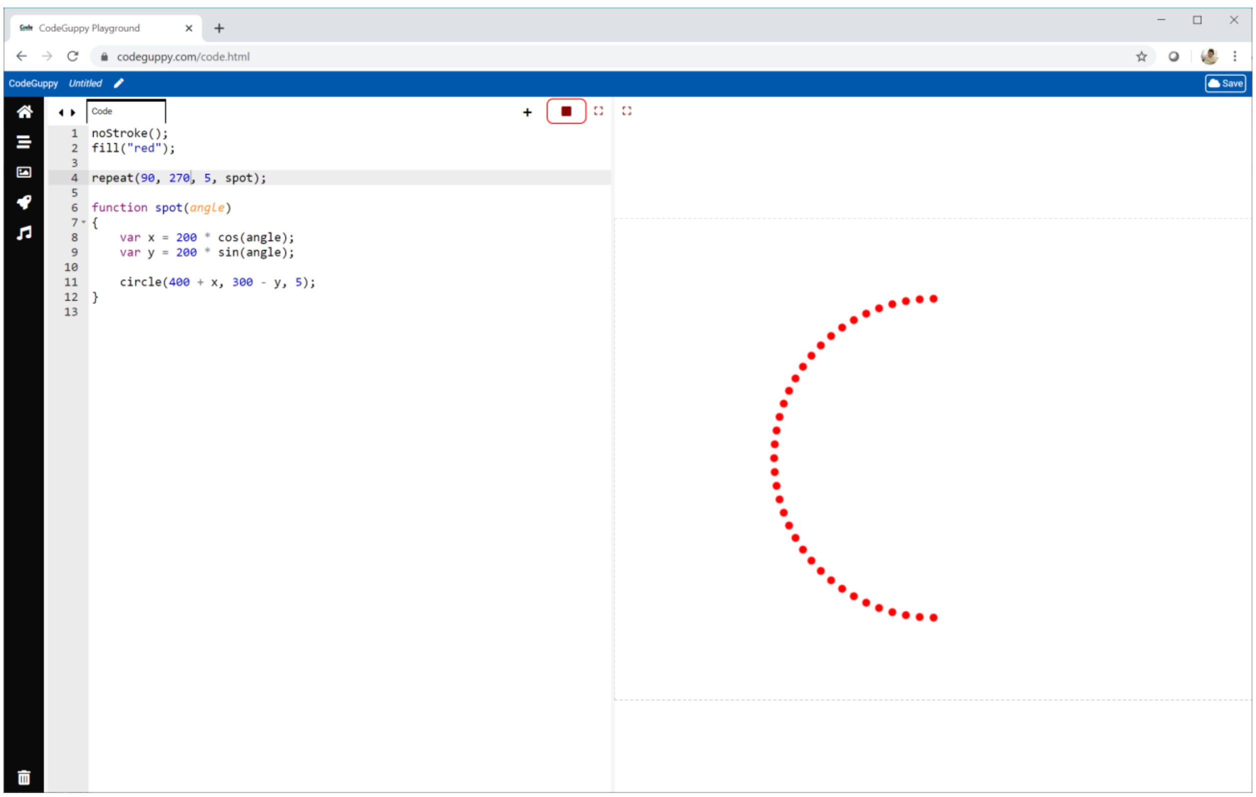
Try It Yourself

Try changing the **repeat** instruction so the **angle** starts and ends at a different number.

Here's my experiment which counts the **angle** from **90** to **270 degrees**.

```
repeat(90, 270, 5, spot);
```

Here's the result.



What happens if your angle grows larger than **360 degrees**?


you've learned how to convert angles to x,y coordinates - a very useful skill
well done!

Orbital Patterns

Let's look back at the code which converts the **angle** to **x** and **y** coordinates.

```
var x = 200 * cos(angle);
var y = 200 * sin(angle);
```

Let's have some fun and do some messing about!

Start by changing the parameters to the **sin()** and **cos()** functions, multiplying them by different numbers.

```
var x = 200 * cos(angle * 3);
var y = 200 * sin(angle * 4);
```

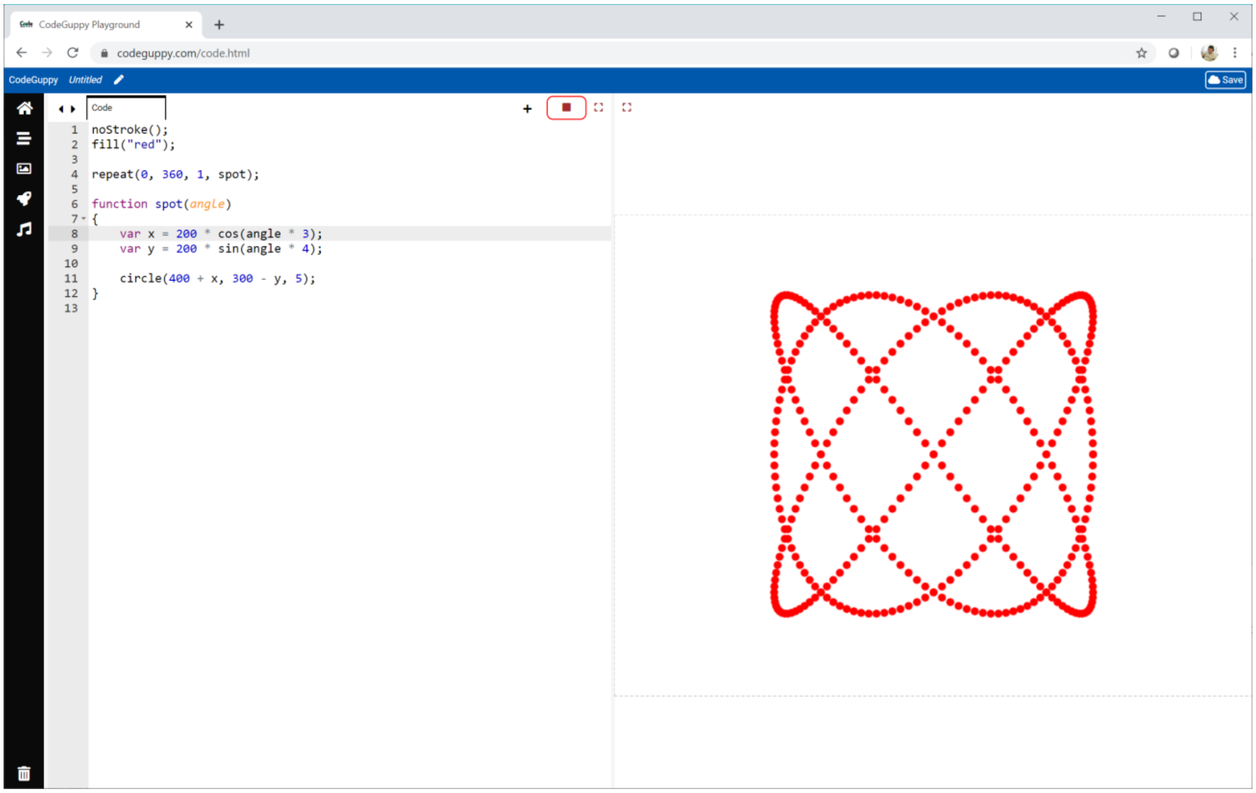
Before we run the code, let's think about what this does.

The **x** coordinate will change as if the angle was going round at **3** times the normal speed. The **y** coordinate will change as if the angle was going round at **4** times the speed.

Change the loop so it counts up to **0** in steps of **1**, not **5**. This way we draw more spots, closer together.

```
repeat(0, 360, 1, spot);
```

Let's see what kind of picture this makes.



Very cool!

Remember when our drawings became busy, we used **translucency** to make them work better.

Let's change those circles so they have a translucent outline and no fill.

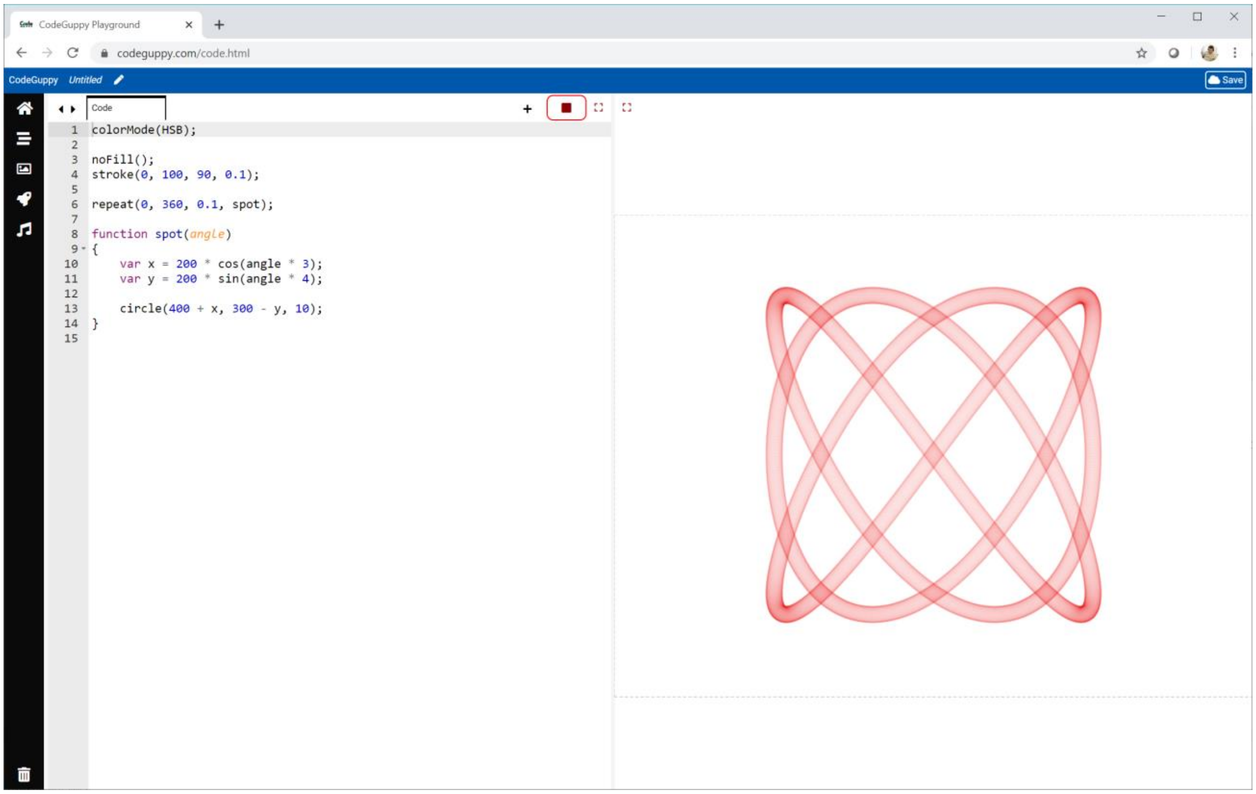
```
noFill();
stroke(0, 100, 90, 0.1);
```

This outline stroke is a red colour in HSB colour mode.

Let's also change the loop so the **angle** is counted up in even smaller steps of **0.1**.

```
repeat(0, 360, 0.1, spot);
```

Let's see the effect.



Very nice!

The code for this drawing is online:

- https://codeguppy.com/code.html?tariq/ex09

Try It Yourself

Try experimenting with different calculations for calculating **x** and **y** from the **angle**.

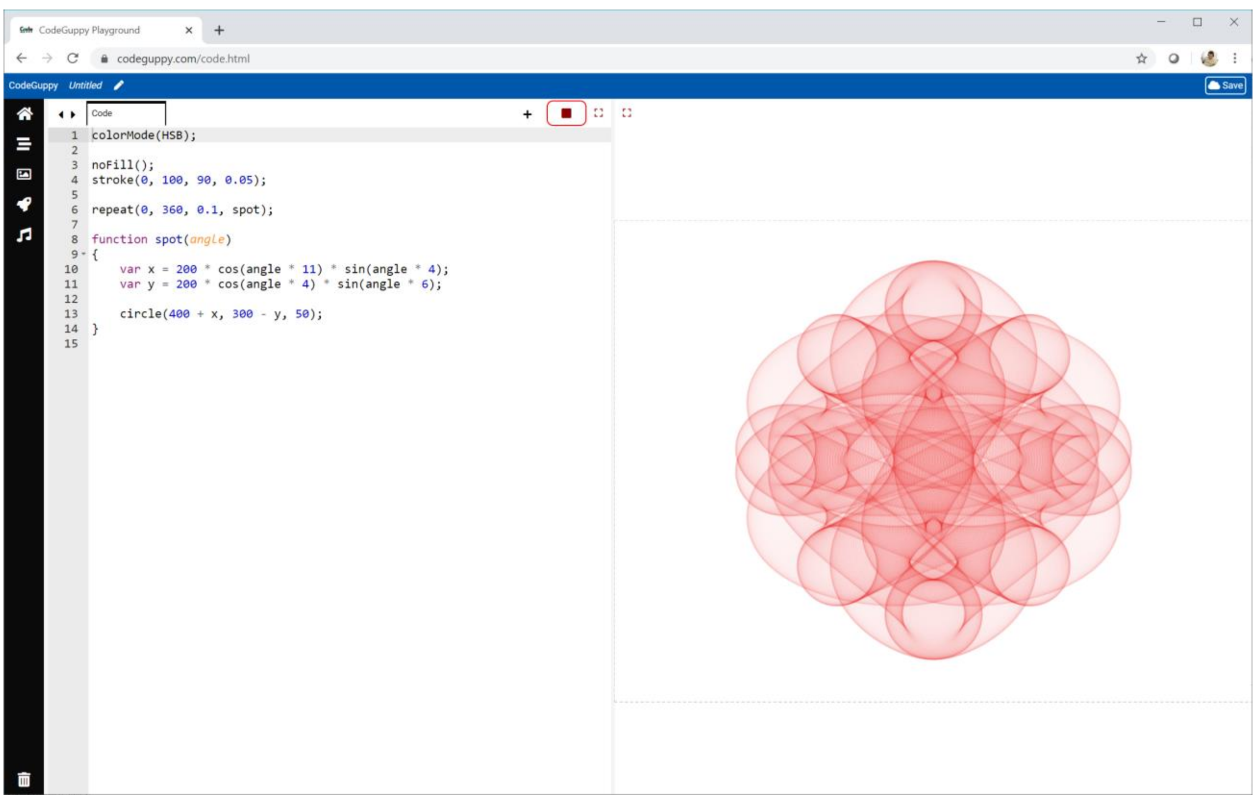
Try using more than one **sin()** and **cos()** in your calculations. You could add or multiply them, or try something else completely.

There’s no right or wrong answer - go wild!

Here is my own experiment.

```
var x = 200 * cos(angle * 11) * sin(angle * 4);
var y = 200 * cos(angle * 4) * sin(angle * 6);
```

Here’s the result.



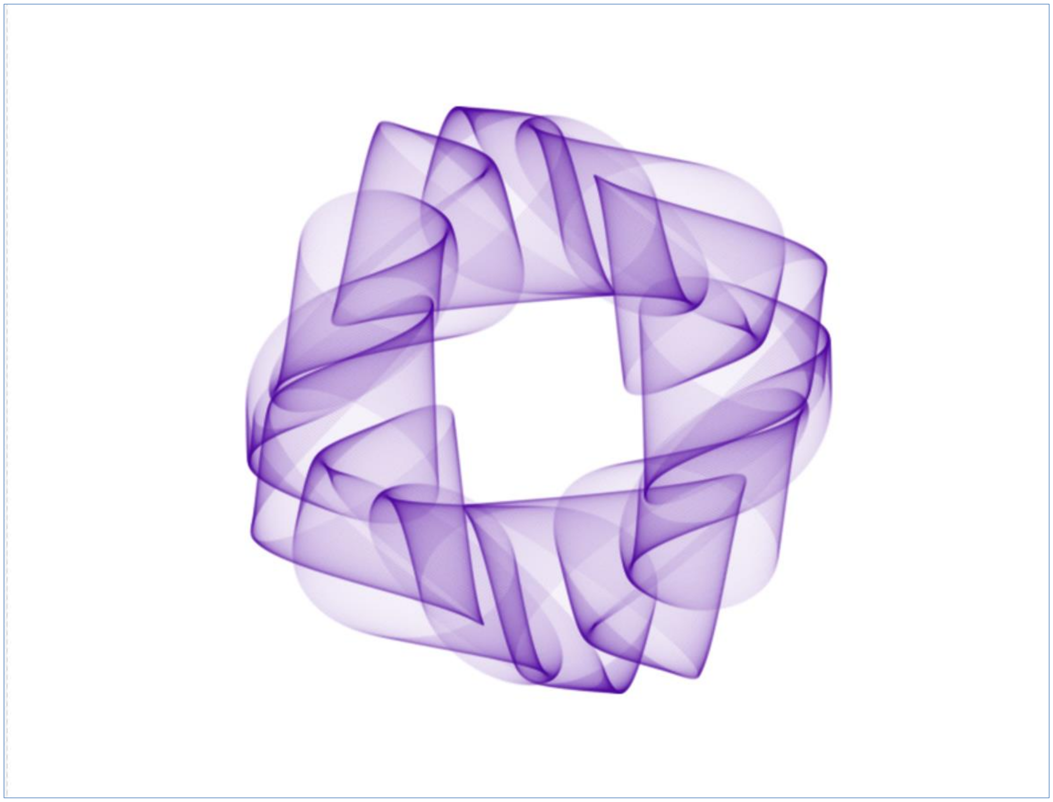
To get this effect I set the circle size to a large **50**, but reduced the translucency to a very small **0.05**.

The full code is online:

- https://codeguppy.com/code.html?tariq/ex10

Challenge!

Have a look at this rather nice pattern.



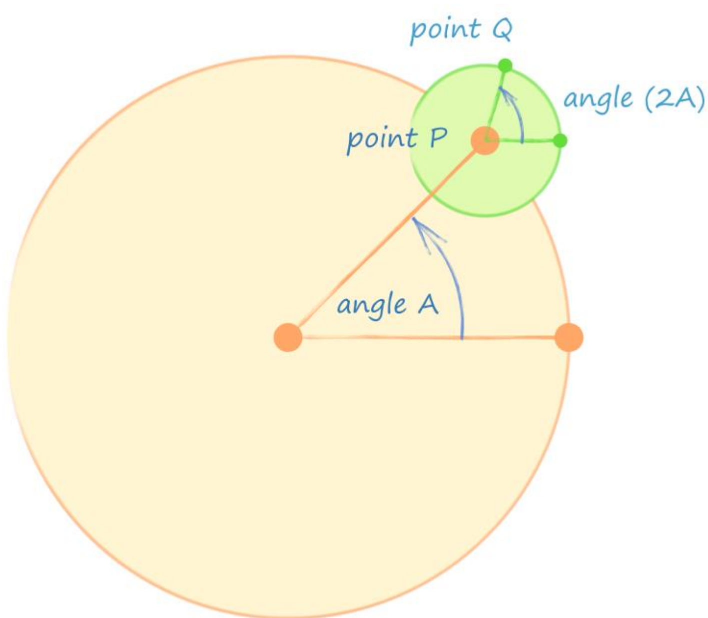
It uses the same idea of drawing small translucent spots as they move around a circle with altered vertical and horizontal angular speeds.

This time the spots follow the edge of a small circle, which itself moves around a larger circle. It's a bit like the moon going around the Earth.

Can you write code to draw similar patterns?

Here are some suggestions:

- This picture shows a smaller green circle moving around a larger circle. If the point **Q** turns around its circle twice as fast as **P** turns around the larger circle, the angle of **Q** is **2A**.



- The coordinates of **Q** from the centre of the green circle can be calculated easily using the **trigonometric identities**.

- The coordinates of **Q** from the centre of the big circle are calculated by adding the coordinates of **Q** to **P**.

- A loop can be used to count the **angle A** from **0** to **360**. A nested loop can count an extra number which can be used to add some variety to the **sin()** and **cos()** parameters.

The surprisingly simple code for the example pattern is online:

- https://codeguppy.com/code.html?tariq/ex11

3.3 - Patterns Inside Patterns

level

What We’ll Do

In this project we’re going to:

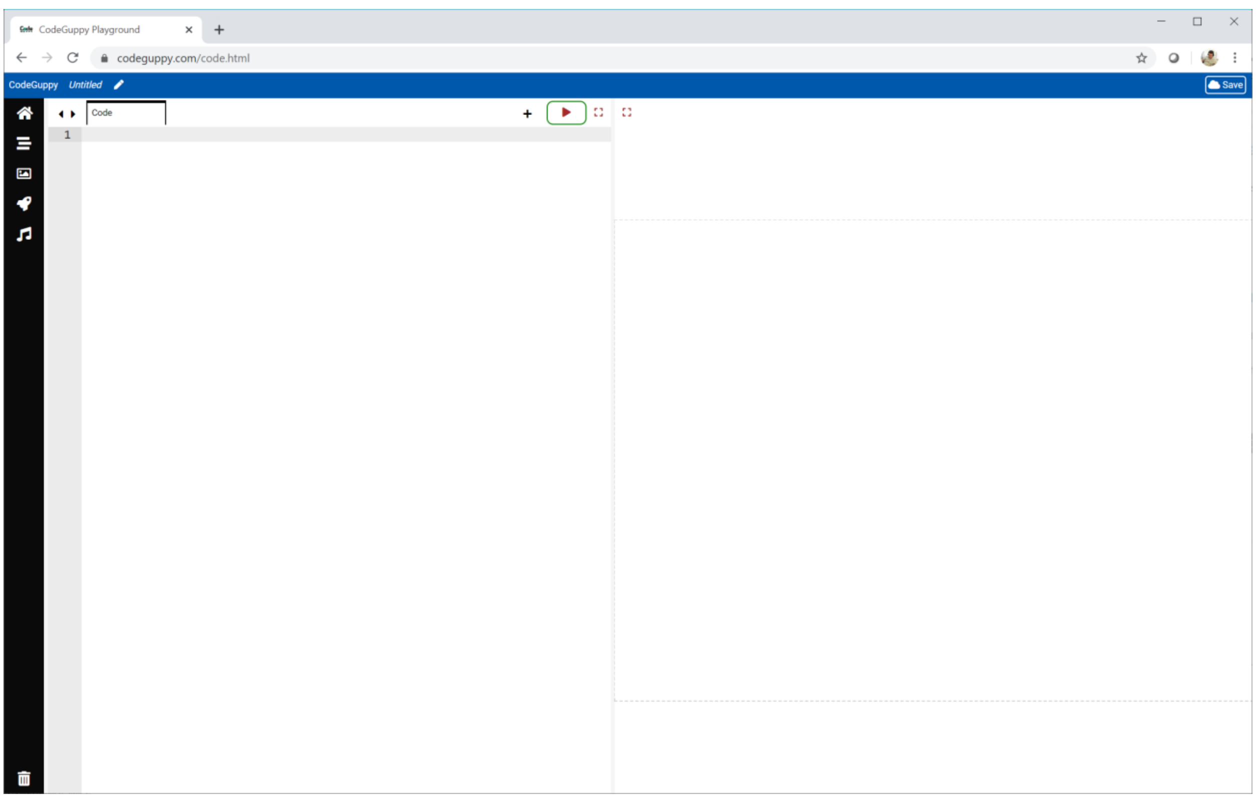
- learn about **self-similarity** and **recursion**
- see how recursion can easily create **intricate patterns**

Start a New Program

Log in to codeguppy.com if you haven’t already.

Create a new program, just like we did in the first project.

Your code should look like this:



Patterns Inside Patterns

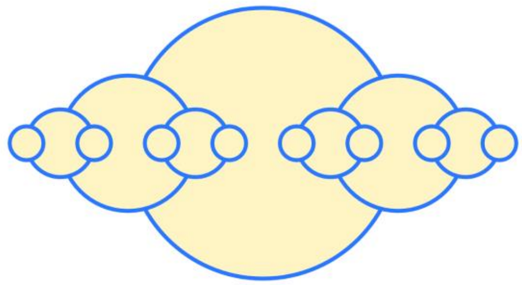
Have a look at this fern leaf.

If you look closely you'll see the leaf is made of smaller leaves that look just like the big one.

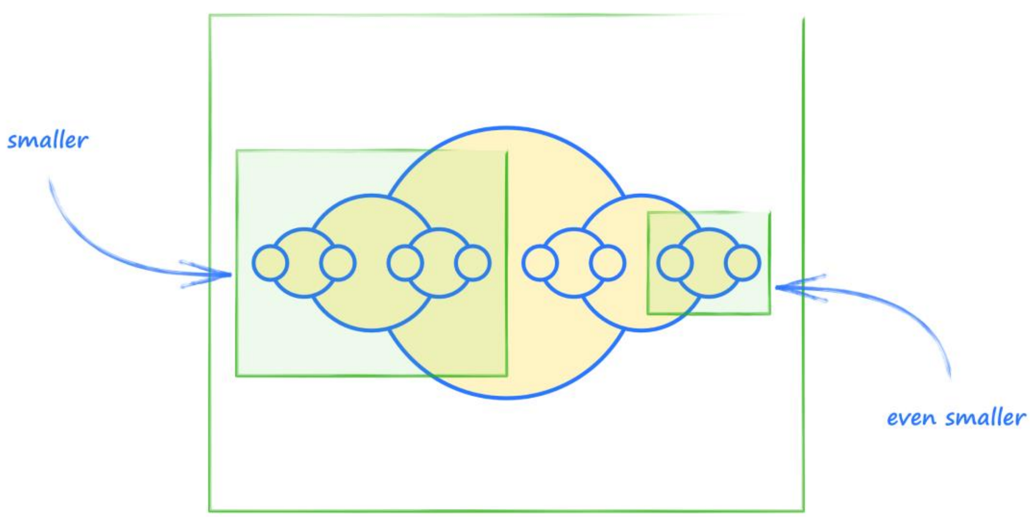
This is called **self-similarity**, and you'll find lots of examples in nature.

Coding Self-Similar Patterns

Have a look at this **self-similar** pattern.



The next picture shows how the whole pattern is made of smaller versions of itself.



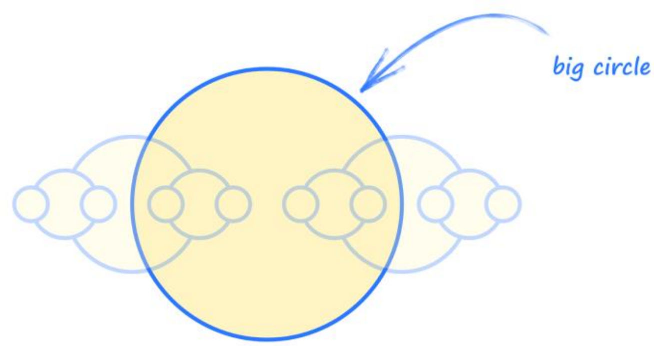
Let’s try to create a function that draws this **self-similar** pattern.

First we'll start with an empty function.

```
function my_pattern() {

}
```

What's the first thing this function needs to do?

If we look back at our pattern we can see there's a big circle in the middle and then two smaller versions of the pattern on the left and right.



Let's focus on the big circle.

```
function my_pattern(x, y, size) {

    circle(x, y, size / 2);

}
```

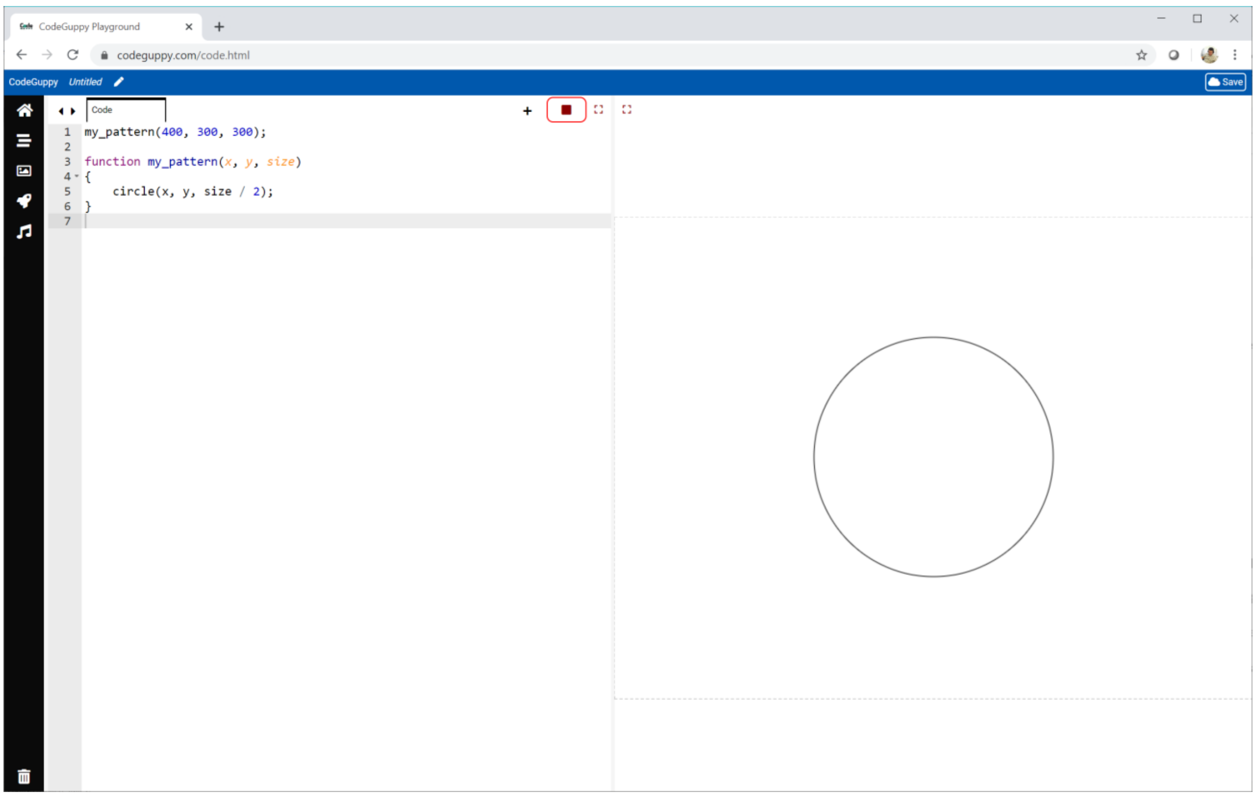
The **my_pattern** function now takes the location and size of the circle, and then draws it.

Let’s try our half-finished function to see what happens.

```
my_pattern(400, 300, 300);
```

That asks for the circle to be in the middle of the canvas with size **300**.

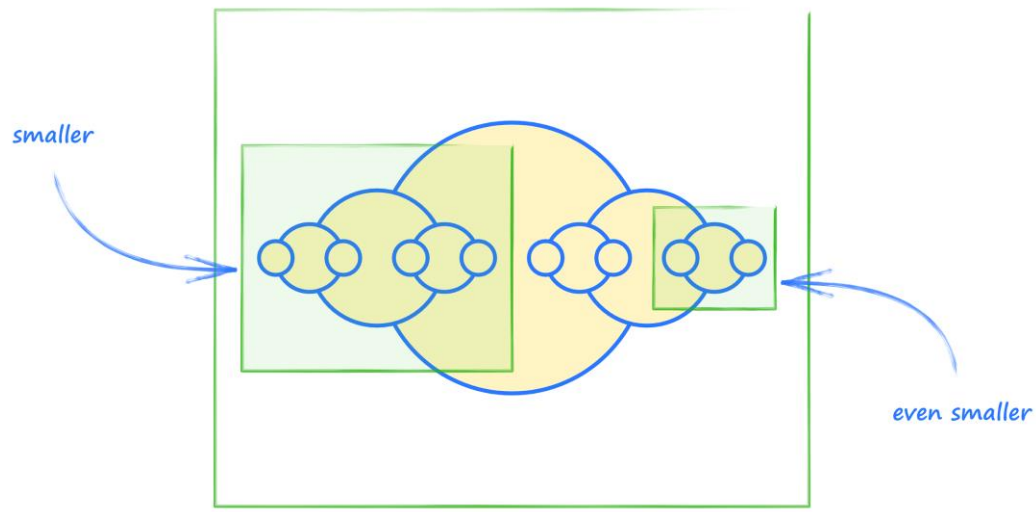
Here’s what it draws:



It’s not that exciting, but it’s a start.

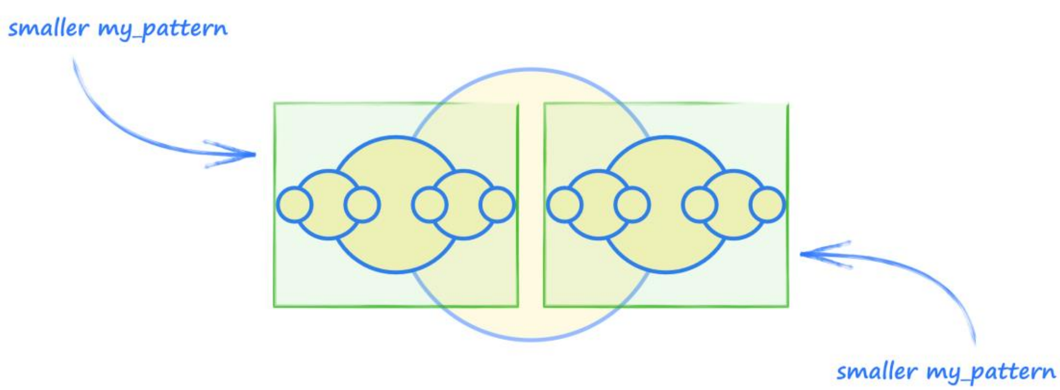
Let’s continue writing our **my_pattern** function.

The next step is to draw the smaller versions of the whole pattern.



We could write more **circle** instructions to draw those smaller circles … but we can be cleverer than that!

Those smaller patterns are just like the whole **my_pattern**, just smaller and moved left and right.



Have a look at this code.

```
function my_pattern(x, y, size) {

    circle(x, y, size);

    // right half of pattern
    my_pattern(x + size/2, y, size/2);

}
```

After we've drawn the big circle we're calling **my_pattern** again but with new parameters that shift it to the right and make it smaller.

Hang on!

We're calling **my_pattern** from inside **my_pattern**!

Does that work?

- If we call **my_pattern**, it will call **my_pattern** again.
- And that **my_pattern** call **my_pattern** again.
- And that **my_pattern** … will call **my_pattern** … forever?

Actually it won't go forever because our code will crash with an error.

We need a way to stop it going on forever.

One way to stop **my_pattern** calling **my_pattern** endlessly is to stop when **size** becomes smaller than a number we choose.

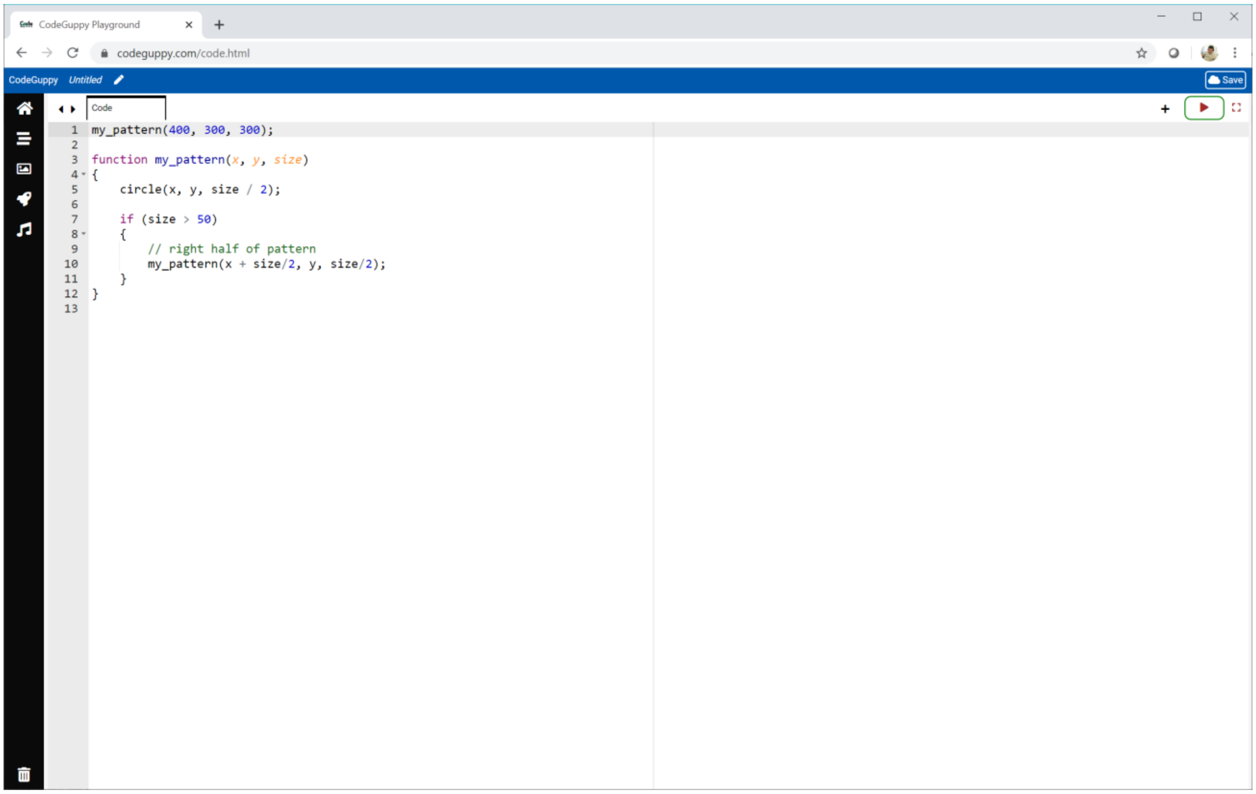
Have a look at this code.

```
function my_pattern(x, y, size) {

    circle(x, y, size);

    if (size > 50) {

        // right half of pattern
        my_pattern(x + size/2, y, size/2);

    }

}
```

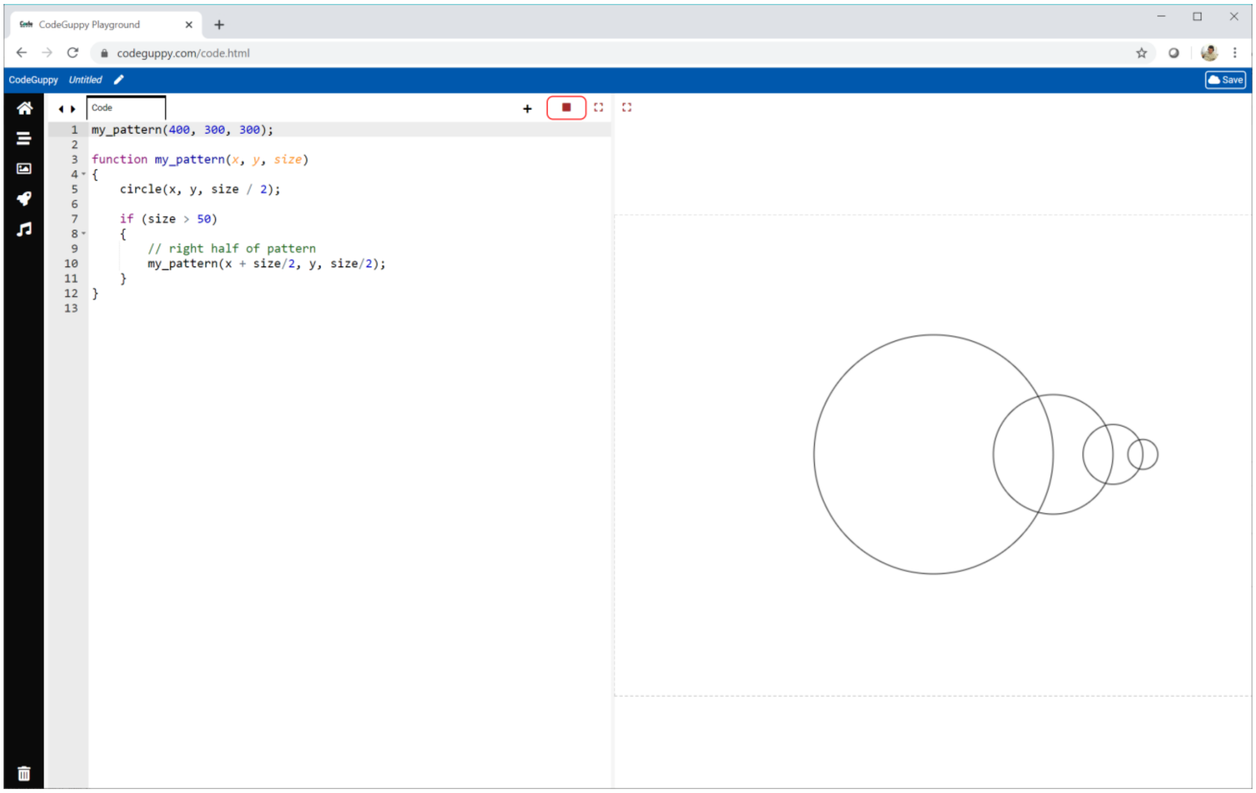
We've used an **if** instruction to check that **size** is more than **50**. If it is, the code inside the curly brackets is run. Here that code is the call to **my_pattern** shifted right and with a smaller **size**.

If **size** is not more than **50**, it doesn't run that code and the function **my_pattern** ends normally.

Here’s the code so far:



Let’s run it.



You can see **my_pattern** has drawn the first big circle, called itself and drawn a smaller circle to the right, and called itself … until the **size** became smaller than **50**.

We've written a function that draws a self-similar pattern, where the code itself is self-similar too. That is pretty wild!

This is called **recursion** - describing something using itself.

It's a pretty powerful idea.

But it's also a pretty mind-bending idea too, and can take a while to understand, so don't worry if you don't get it straight away!

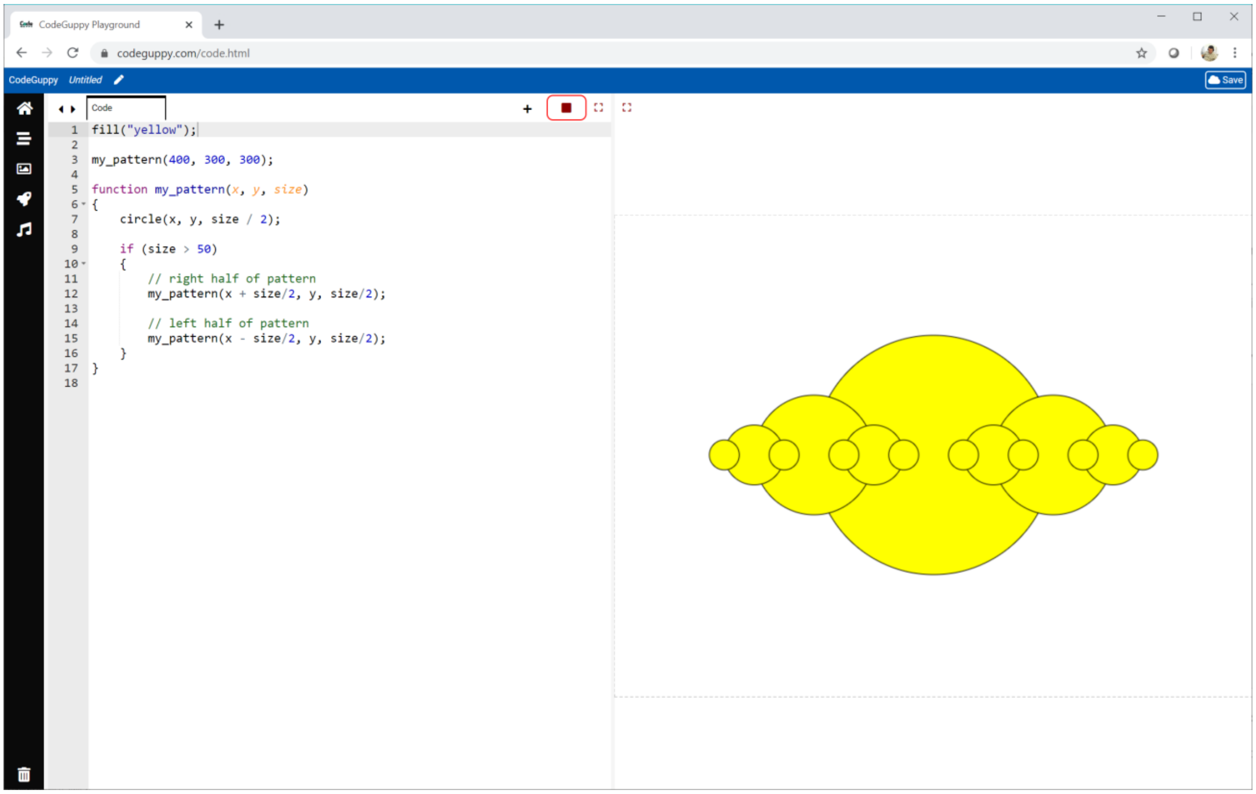
Let’s finish our function with the left part of the pattern.

```
function my_pattern(x, y, size) {

    circle(x, y, size);

    if (size > 50) {

        // right half of pattern
        my_pattern(x + size/2, y, size/2);

        // left half of pattern
        my_pattern(x - size/2, y, size/2);

    }

}
```

All we’ve done is to say that the pattern has a self-similar smaller version to the left of the main circle, just like the one to the right.

We also filled the circle with yellow color.

And here’s the result:



We did it!

Take a break! You just did something that many people think is really difficult!

The code for this program is online:

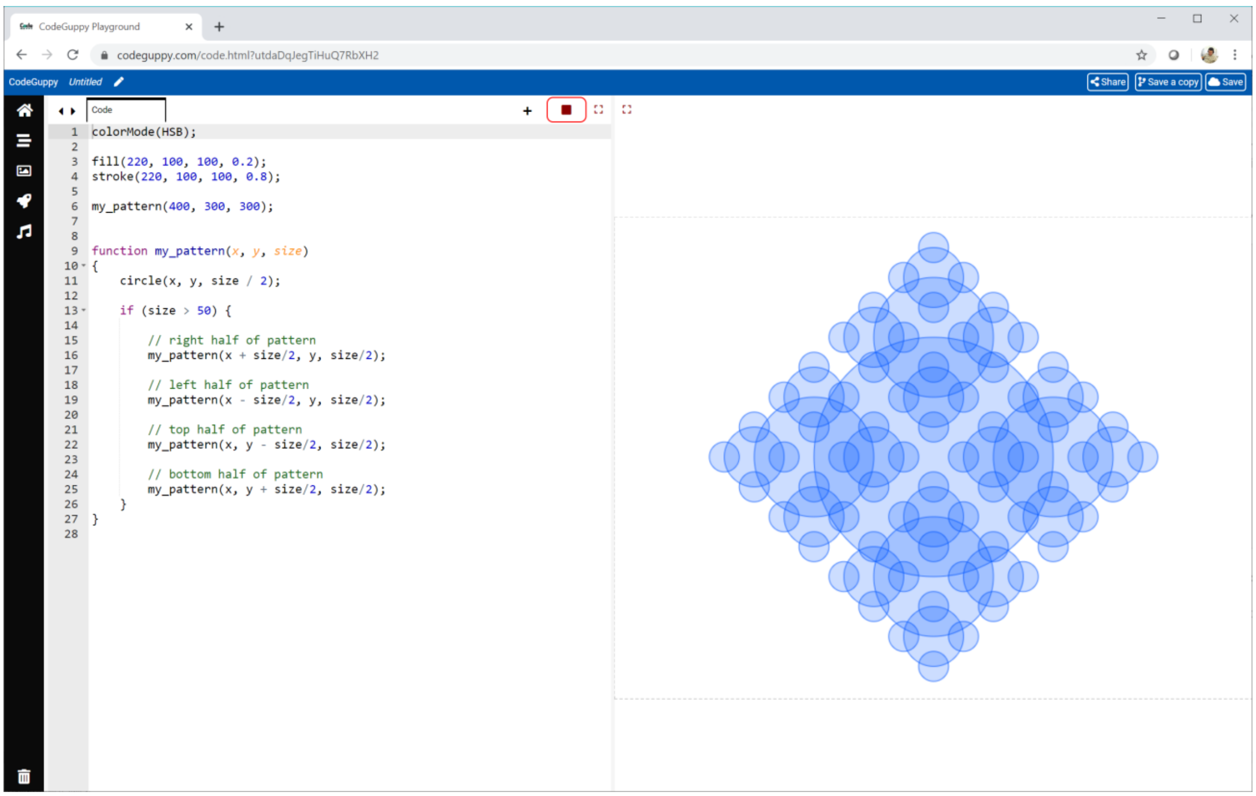
- https://codeguppy.com/code.html?tariq/ex12

Try It Yourself

Change the **my_pattern** function to see what different patterns are drawn.

You might change the **size** limit, or the **parameters** to the **my_pattern** functions.

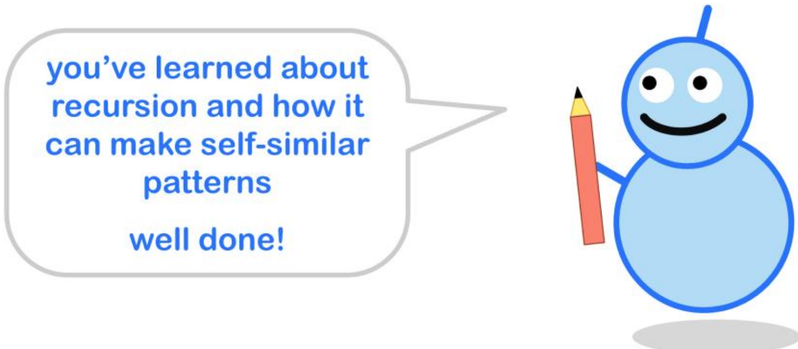
Here is my own experiment which adds a smaller **my_pattern** above and below the main circle. That’s just two new lines of code.



I also changed the fill and outline colours to be **translucent**.

The code for my experiment is online:

- https://codeguppy.com/code.html?tariq/ex13


you've learned about recursion and how it can make self-similar patterns
well done!

Writing Recursive Functions

Many people find it takes a while to get used to recursion.

Here is a useful way to think about writing recursive functions.

All recursive functions have **three** parts:

current	describe the current level of detail
next	describe where the next self-similar detail is
stop	decide when to stop going into more detail

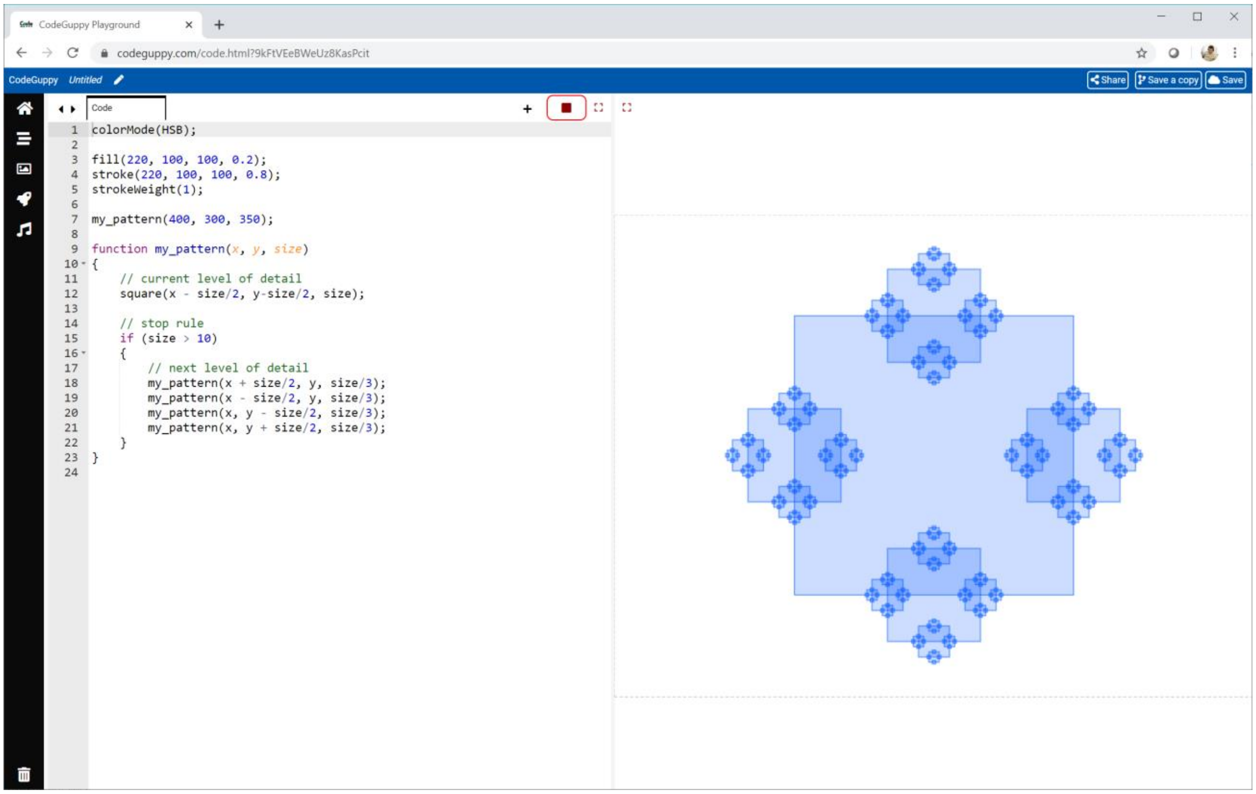
Let's see how this works for the **my_pattern** function we wrote before.

- **current** - a circle at **(x, y)** with the given **size**
- **next** - smaller **my_pattern** to the **left** and **right**
- **stop** - when **size** is **50** or less

Let's practice this way of thinking next.

Try It Yourself

Have a look at this **recursive** pattern.



See if you can describe this pattern using **current**, **next** and **stop** rules.

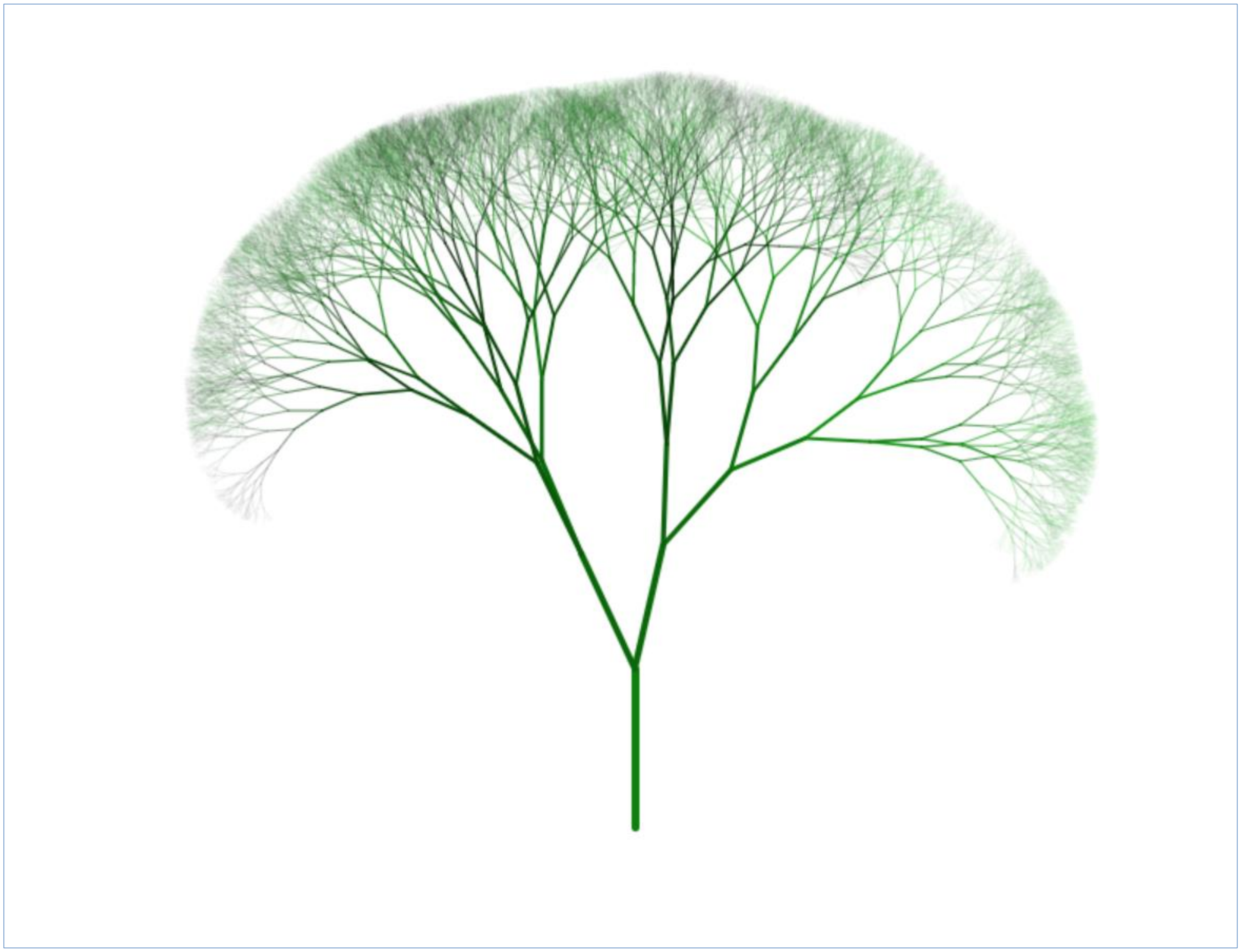
Use these rules to write a **recursive** function to draw this pattern.

My code is online if you need to look at it:

- https://codeguppy.com/code.html?tariq/ex14

Challenge!

Have a look at this beautiful tree!



This tree is actually a **recursive** pattern.

Each branch has two smaller trees connected to it.

The challenge is to write code to draw similar recursive trees.

Here are some details about how the tree was drawn.

- The **current** detail is a branch, a line drawn at an **angle**.

- The **next** level of detail is **two** smaller trees at the end of the current branch. The angle of these trees is the **angle** of the branch changed by a small random amount, one clockwise and one anti-clockwise.

- The recursion **stops** when the branch **length** is **5** or less. The first branch has a **length** of **100**.

- The **thickness** of the lines is calculated from the **length**, so a shorter **length** means a thinner line.

- The **translucency** of the lines is calculated from the **length** too, so a shorter **length** means a more see-through colour.

- The **HSB brightness** of each branch is adjusted by a random amount at each level of detail.

My own code for this tree is online:

- https://codeguppy.com/code.html?tariq/ex15

3.4 - More Flexible Loops

level

What We'll Do

In this project we're going to:

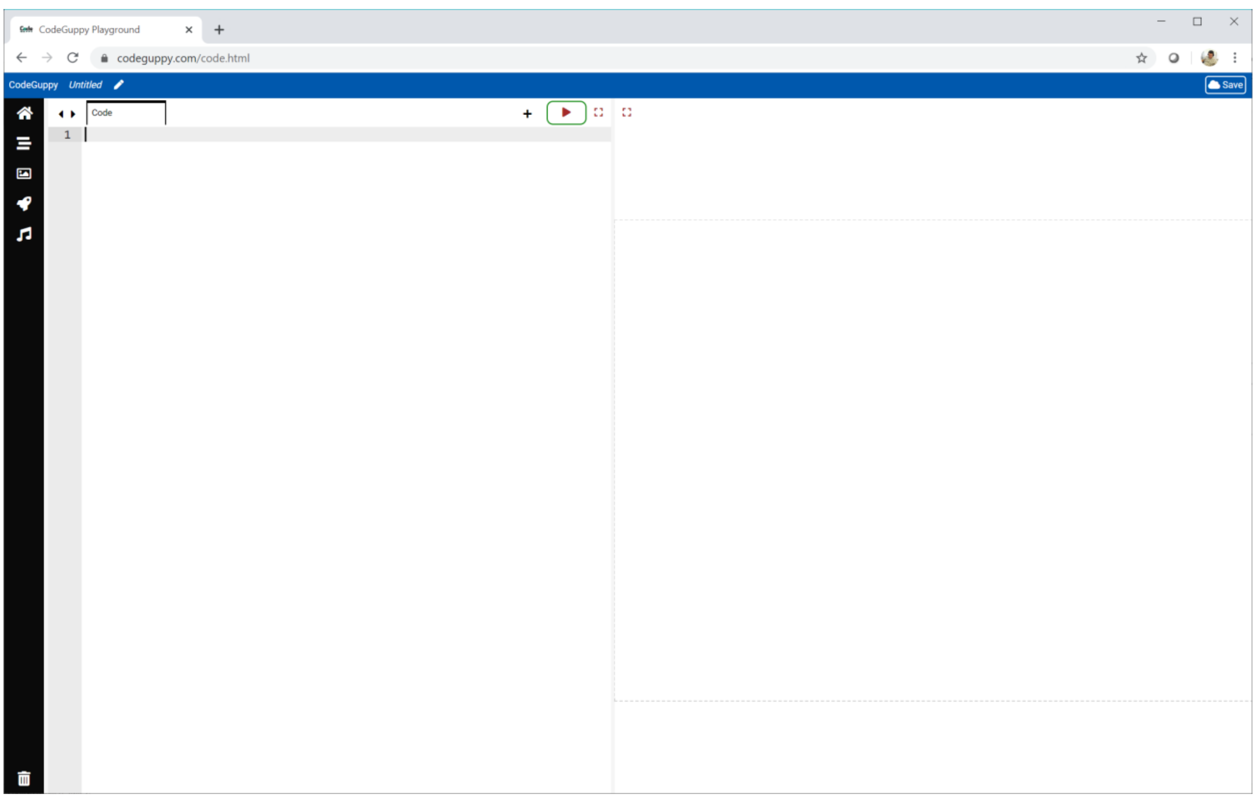
- learn about javascript's own **for loops**
- see how they're more flexible than the **repeat** instruction

Start a New Program

Log in to codeguppy.com if you haven't already.

Create a new program, just like we did in the first project.

Your code should look like this:



A Simple For Loop

So far we’ve used the **repeat** instruction to run code many times.

We’re now going to learn about **for loops** because they can do things that **repeat** can’t.

Have a look at this very simple **for loop**. Can you work out how it works?

```
for (var x = 100; x < 800; x += 100) {

    circle(x, 300, 25);

}
```

The curly brackets contain the code to be repeated. That’s the **circle** instruction. Easy enough!

The **for** instruction tells our computer this is the start of a **for loop**.

What’s all that stuff inside the round brackets?

Let’s break it down:

- **var x = 100** creates a new loop **counter**, a variable called **x**, with a value of **100**.

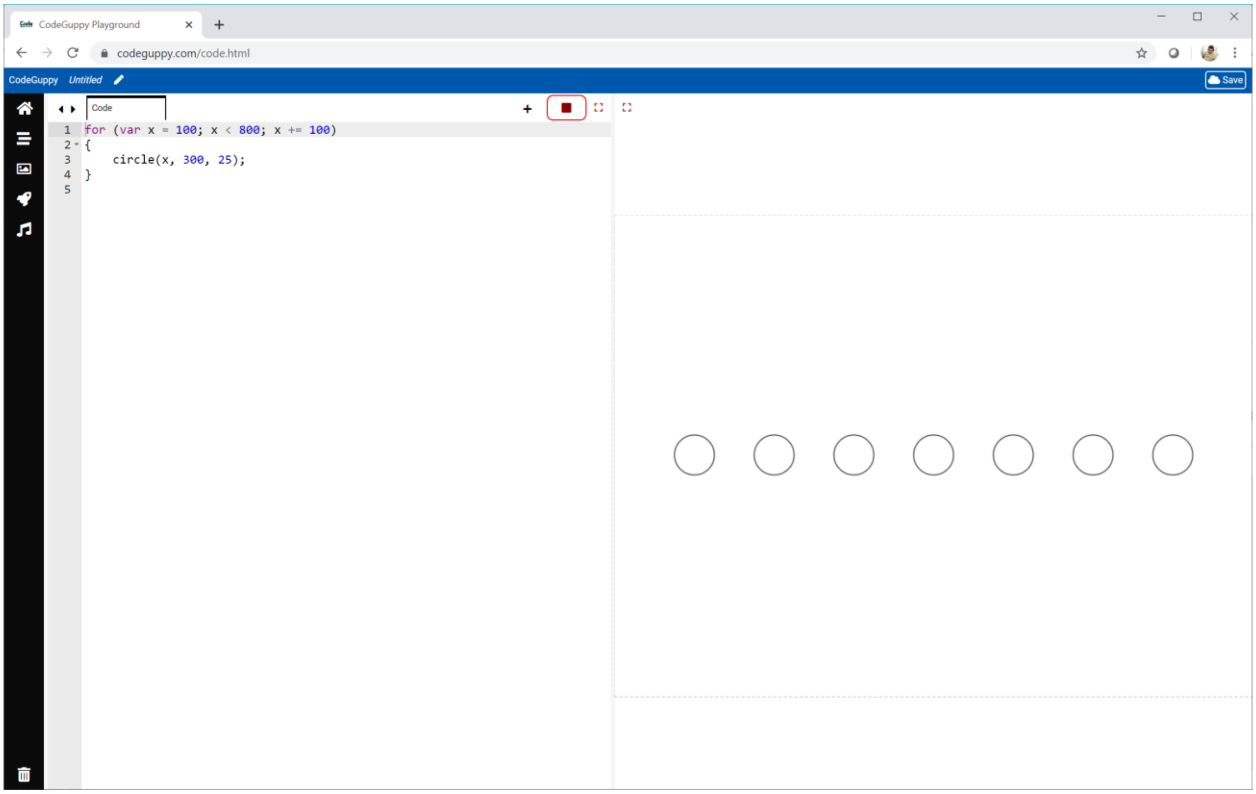
- **x < 800** is a test to see if the loop should continue. The loop repeats as long as **x** is less than **800**.

- **x += 100** is how the loop counter is increased after every repetition of the code.

The code should draw circles along the middle of the canvas. The first one will be at **(100, 300)**, the next one at **(200, 300)** .. all the way up to **(700, 300)**.

Try it.

You should get a picture like this.



Why is the last circle not at **(800, 300)**?

The **for loop** test to continue is **x < 800**. That means the repeated code is run as long as **x is less than 800**. After the code has drawn the circle at **(700, 300)**, **x** is increased to **800**. That means **x** is not less than **800**, and the loop stops.

This sometimes trips up even experienced coders!

Some coders prefer to see the last value that their counter will have in the repeated code. They can do this with a different continuation test.

```
for (var x = 100; x <= 700; x += 100) {

    circle(x, 300, 25);

}
```

Here the continuation test is **x <= 700**, which is true if **x is less than or equal to 700**.

Try it!

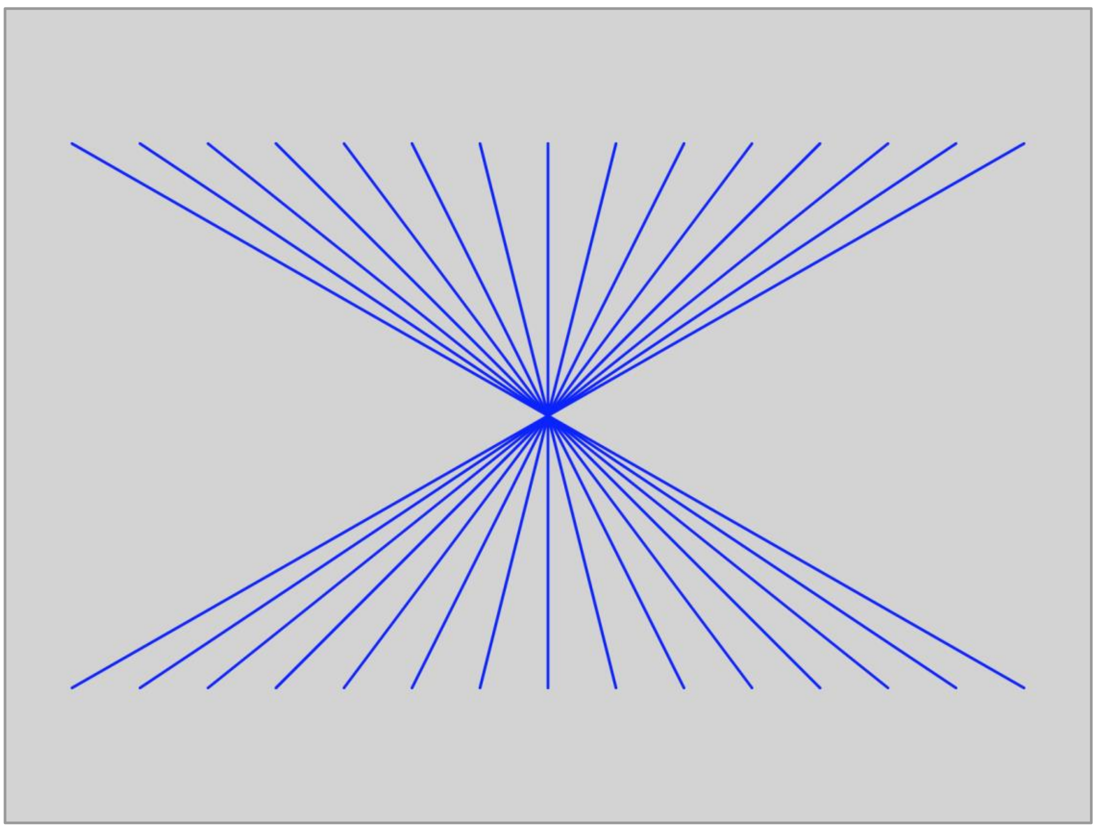
The overall effect is the same, and you can use the style you prefer in your own code.

Try It Yourself

Experiment with **for loops** yourself.

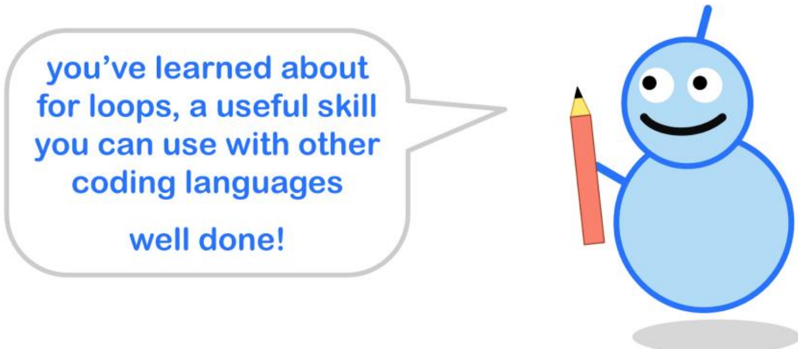
Try different numbers for **starting** and **increasing** the loop **counter**. Try different **tests** for continuing the loop. Make sure you also experiment with using the loop counter inside the repeated code too.

Here's my own simple experiment.



A loop counter **x** is started at **50**, and increased by **50** after every repetition. The loop repeats as long as **x < 800**. A line is drawn from **(x, 100)** to **(800-x, 500)**.

Quite a cool effect for such simple code!


you've learned about
for loops, a useful skill
you can use with other
coding languages
well done!

Nested For Loops

Do you remember using the **repeat** instruction to nest an inner loop inside an outer loop?

We can nest **for loops** too. It's quite easy.

Here's a simple example.

```
for (var x = 100; x <= 700; x += 100) {

    for (var y = 100; y <= 500; y += 100) {

        circle(x, y, 25);

    }
}
```

You can see the outer loop in **blue**. The repeated code is inside its curly brackets. That's the inner loop in **green**.

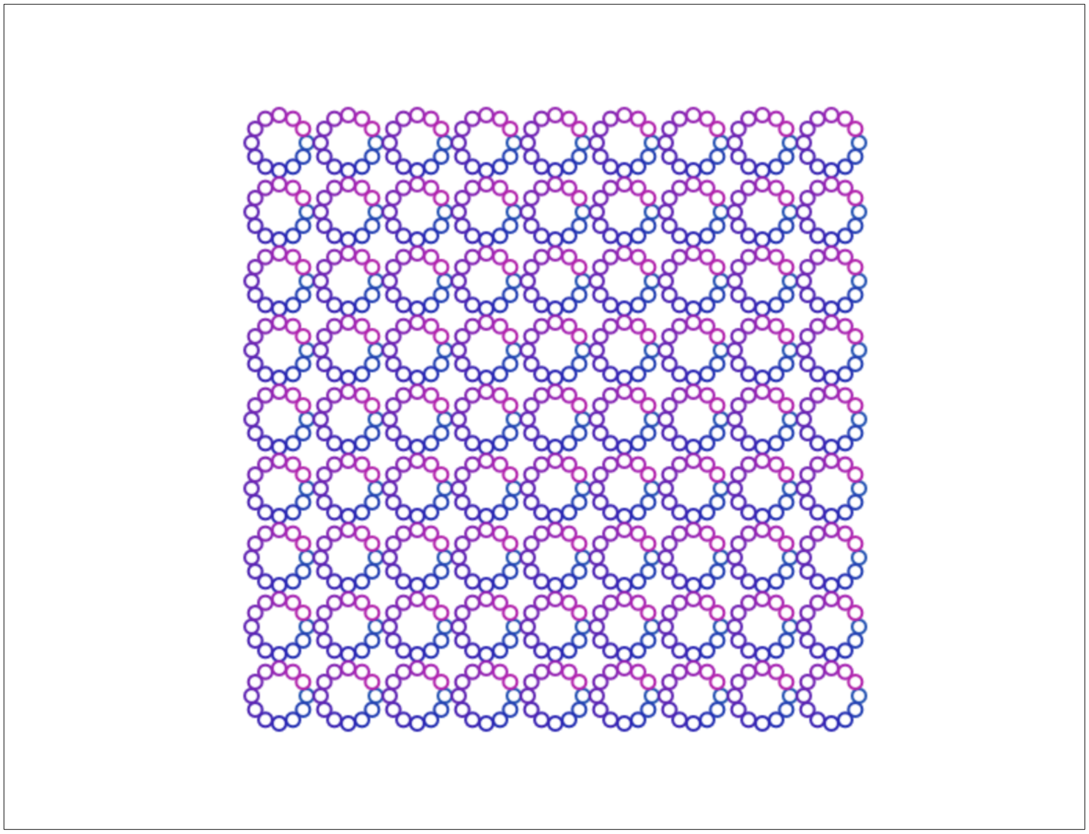
You'll remember that nested loops have two loop counters. Here they are **x** and **y**.

Try this code yourself. You should get a regular grid of circles.

Challenge!

We can nest **for loops** more than two deep. We can’t do that easily with the **repeat** instruction.

Have a look at this design.



The challenge is to write code to create a similar pattern.

The main thing to notice is the grid of rings going across and down the canvas. We usually use a nested loop to do that.

But each ring is made of smaller circles going around it. That means another loop inside that nested loop.

That’s a loop, inside a loop, inside a loop!

Here are some more details about how the design is made:

- The rings are on a grid spaced **50** pixels apart.

- The small circles are placed every **30 degrees** around each ring. We can use the trigonometric relations we saw earlier to calculate the **x** and **y** coordinates of those small circle centres.

- The **hue** of each small circle is decided by its **angle** around the the ring.

The code for this design is online if you need to look at it:

- https://codeguppy.com/code.html?tariq/ex16

3.5 - Classes And Objects

level

What We'll Do

In this project we're going to:

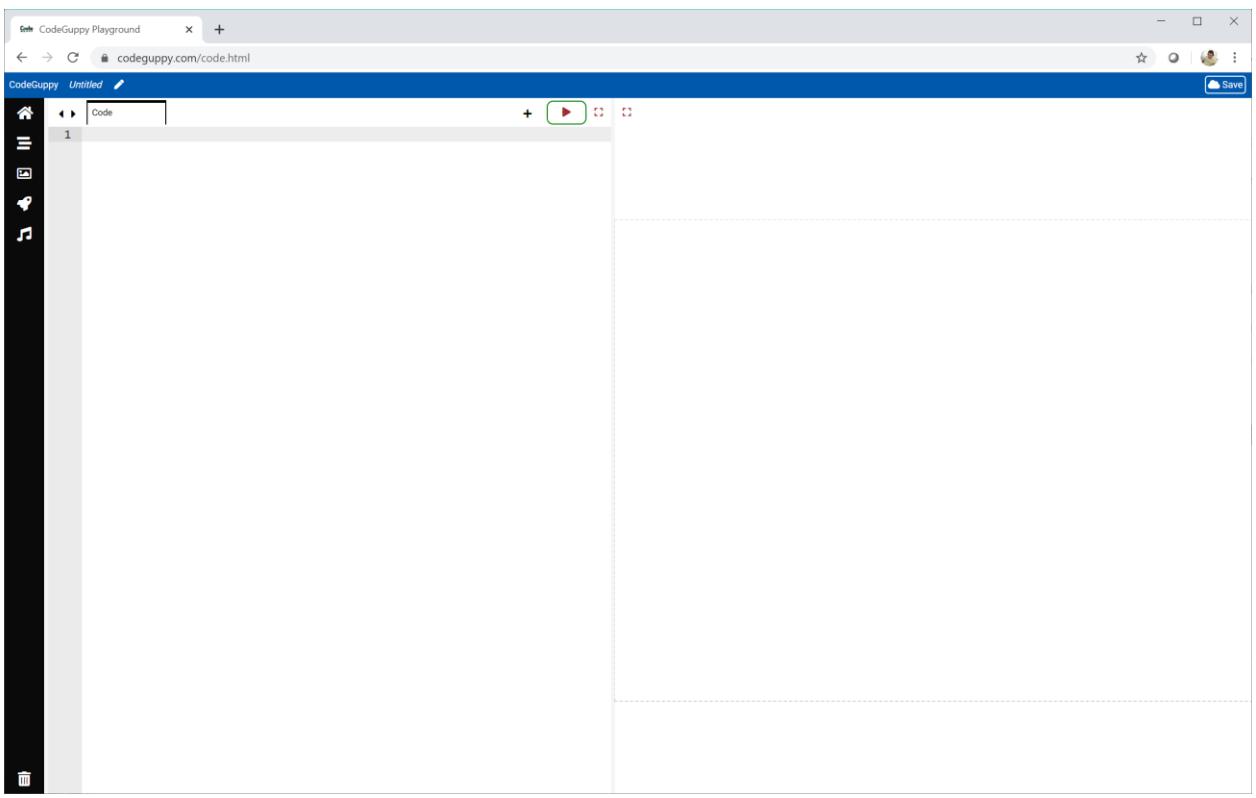
- learn about **classes** and **objects**
- see how we can use them to **simulate** moving things

Start a New Program

Log in to codeguppy.com if you haven't already.

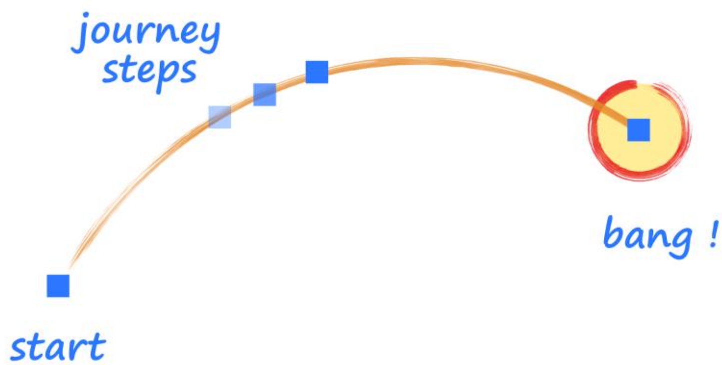
Create a new program, just like we did in the first project.

Your code window should look like this:



Following A Firework

Have a look at this picture of a firework.



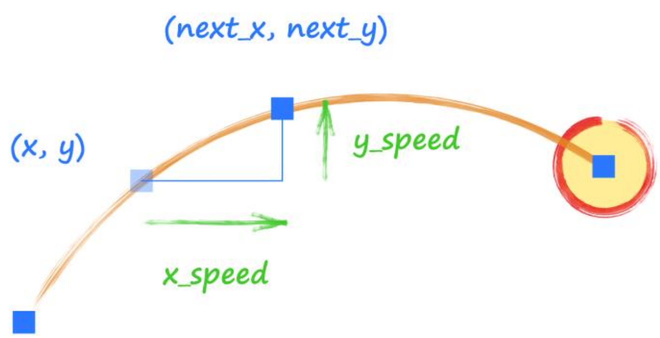
Let’s write code to draw the firework’s journey, as it flies up and then falls, before exploding with a bang!

We’ll need to describe where the firework is. We can use **(x, y)** coordinates to do this.

Because the firework is moving, its **(x, y)** coordinates change as it flies.

How do they change? We’ll look at that next.

This picture shows the firework at **(x, y)** and then a bit later at **(next_x, next_y)**.



The picture also shows that **next_x** is bigger than **x** by a small amount, **x_speed**. That's the speed in the horizontal direction.

If **x_speed** was large, **next_x** would be much further away from **x**. That makes sense. A larger speed means it moves further more quickly.

So, to follow the firework's location, we keep adding **x_speed** to its **x** coordinate, and adding **y_speed** to its **y** coordinate.

A Firework Object

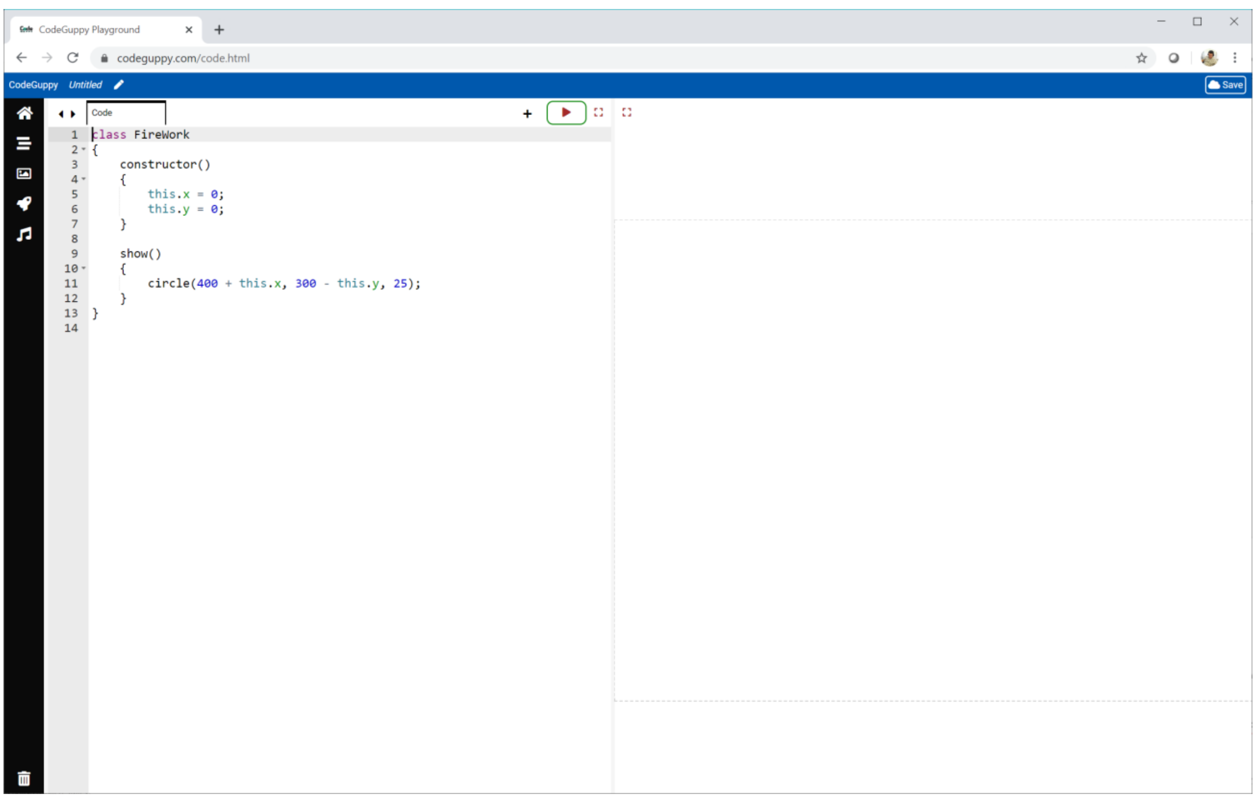
Writing code to follow a firework as it flies through the air seems easy enough.

But what if we wanted **100** fireworks? What if each one had a different starting point? What if each one was travelling in a different direction?

To stop our code (and our brains) getting messy, it’s useful to have each firework as a kind of code **object** that describes everything about it, neatly all in one place.

It’s easier to see this **object** idea working, than talk about it.

Have a look at this code.



There’s something new here. That new thing starts with a **class** instruction instead of a **function** instruction.

It has a name, **FireWork**, and inside the curly brackets there is code.

So far it looks like a normal function.

Looking inside this **class** we can see what look like two functions. One is called **constructor()**, and one is called **show()**.

Here’s the difference between a **class** and a normal **function**.

- a **function** is just packaging up code so it can be used many times by calling its name

- a **class** is like a blueprint for creating **objects** that contain the **variables** and **functions** described in the blueprint

You’ll hear people say **method** instead of **function** when they’re talking about functions of a **class**.

They might also say **data** when they mean the **variables** in a **class**.

Enough talk, let’s see try this **FireWork** blueprint.

Create a **FireWork** class using the code we just saw.

Next, use that **FireWork** class to create an object like this:

```
var my_firework = new FireWork();
```

This creates a new variable, which we've called **my_firework**. Instead of pointing to a number or colour, **my_firework** points to a new **object** created from the **FireWork** blueprint.

Try running the code.

Nothing happens. That's because we've created the **object** but haven't used it yet.

If you look at the **FireWork** class again, you'll see one of the methods was called **show()**.

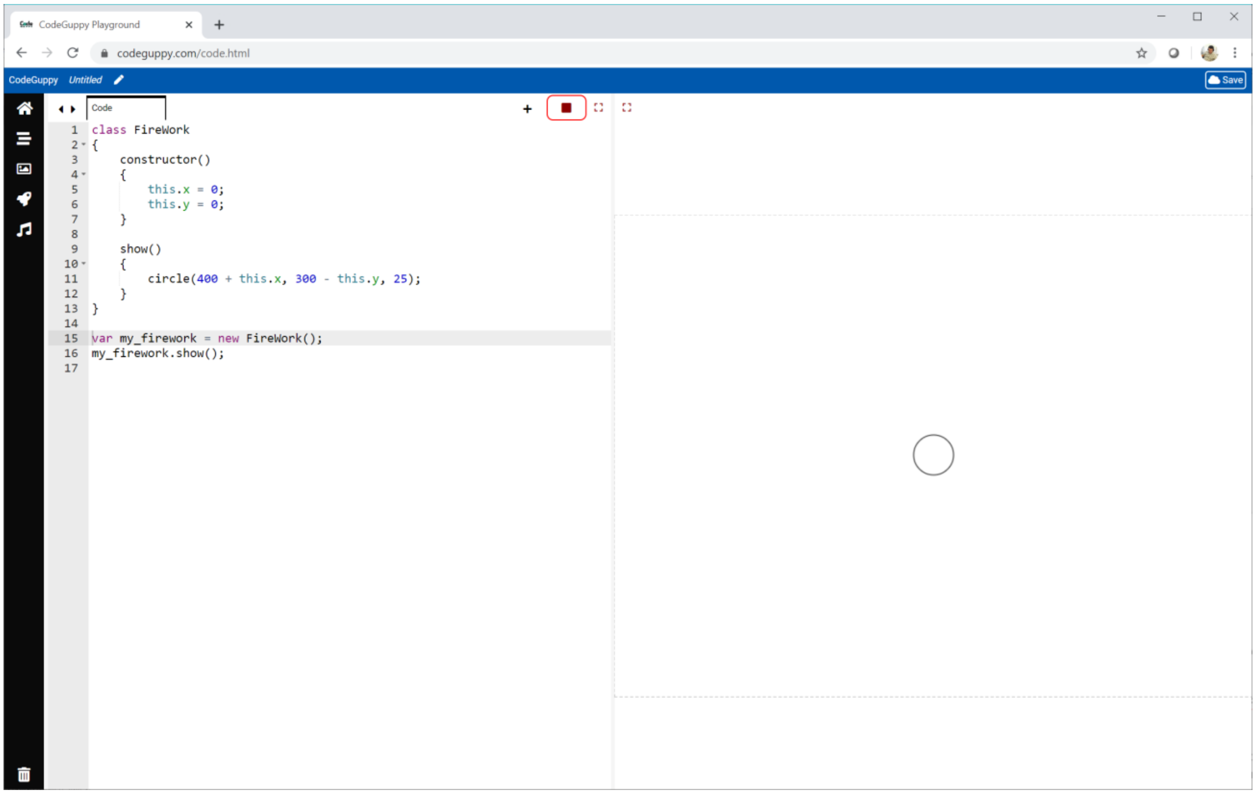
Here's how we use an object's method.

```
my_firework.show();
```

We simply use the method's name after the object, with a dot in between.

Before you run the code, see if you can work out what will happen.

Here’s the result.



If we look at the **show()** method we can see it draws a circle of radius **25**.

```
show() {
    circle(400 + this.x, 300 - this.y, 25);
}
```

But what are **this.x** and **this.y**?

They look like normal **x** and **y** variables, but the **this** means they’re part of this **FireWork** object.

They’re created in the **constructor()** method, which is a special method called when an object is first created. That’s why it’s called a constructor - it sets up the object ready to be used.

Let's look again at that **constructor()** to see how **this.x** and **this.y** are created.

```
constructor() {
    this.x = 0;
    this.y = 0;
}
```

That's easy enough.

Making The Firework Move

So far we've designed a **FireWork** class which has:

- **data** about the firework's **x** and **y** location
- a **method** to draw the firework at **(x, y)**

Our firework doesn't know how to move yet. To teach it we need to add to our **FireWork** class:

- **data** about the firework's speed
- a **method** to change the firework's location

We can easily add two variables to the **FireWork** class for the horizontal and vertical speed, like this:

```
constructor() {
    // location
    this.x = 0;
    this.y = 0;

    // speed
    this.x_speed = 20;
    this.y_speed = 0;
}
```

So when a new object is created from this blueprint, it will have inside it variables that describe its location and speed.

In our example, the **x_speed** is **20** and **y_speed** is **0**, which means the object is moving directly to the right.

Now that we know the speed of the firework, let's add a method to change its location.

Let's call it **move()**.

```
move() {
    this.x += this.x_speed;
    this.y += this.y_speed;
}
```

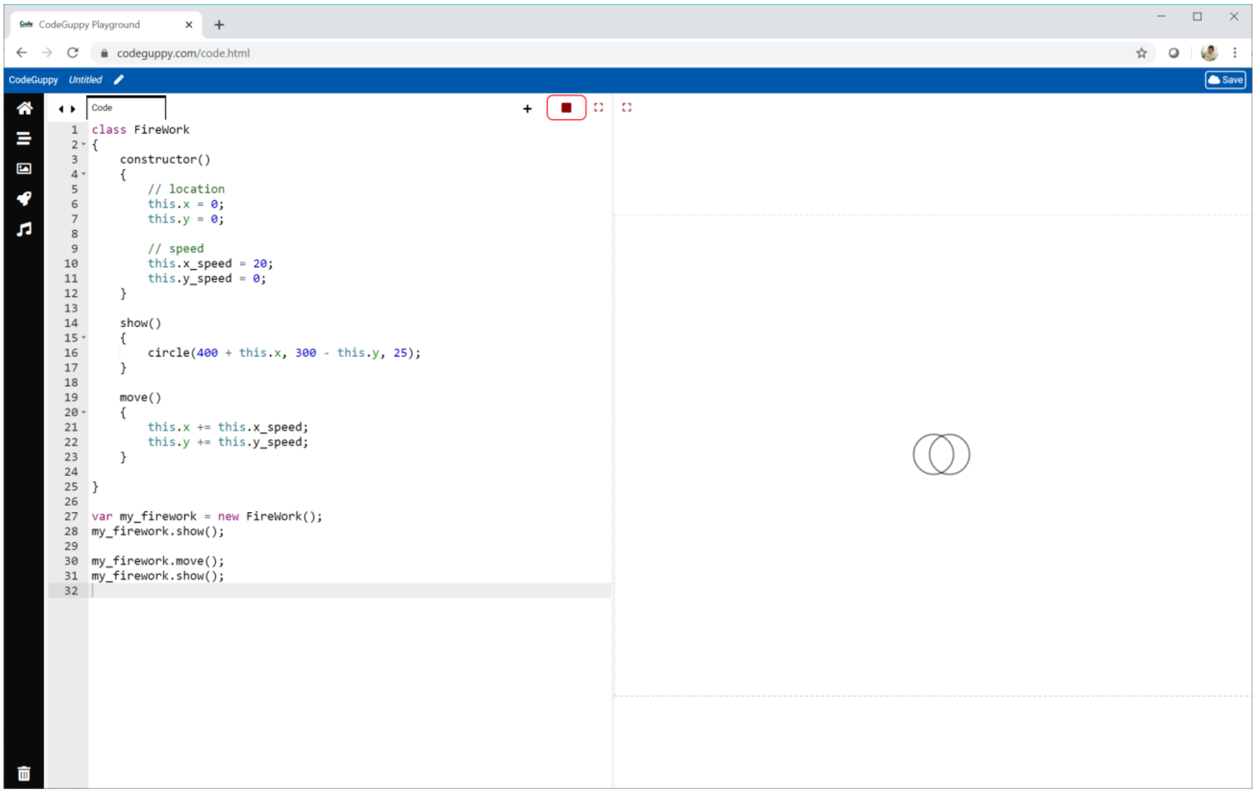
That's a small amount of code. All it does is update the firework's **x** coordinate by adding **x_speed** to it. The **y** coordinate is updated by adding **y_speed** to it.

Now that we've shown our firework how to move, let's try it.

In the main section, we already show the firework. We can now call its **move()** method, and **show()** it again.

```
my_firework.show();

my_firework.move();
my_firework.show();
```

Let's see the result.



The picture shows the firework has moved to the right by **20** pixels because it has a horizontal speed of **20**.

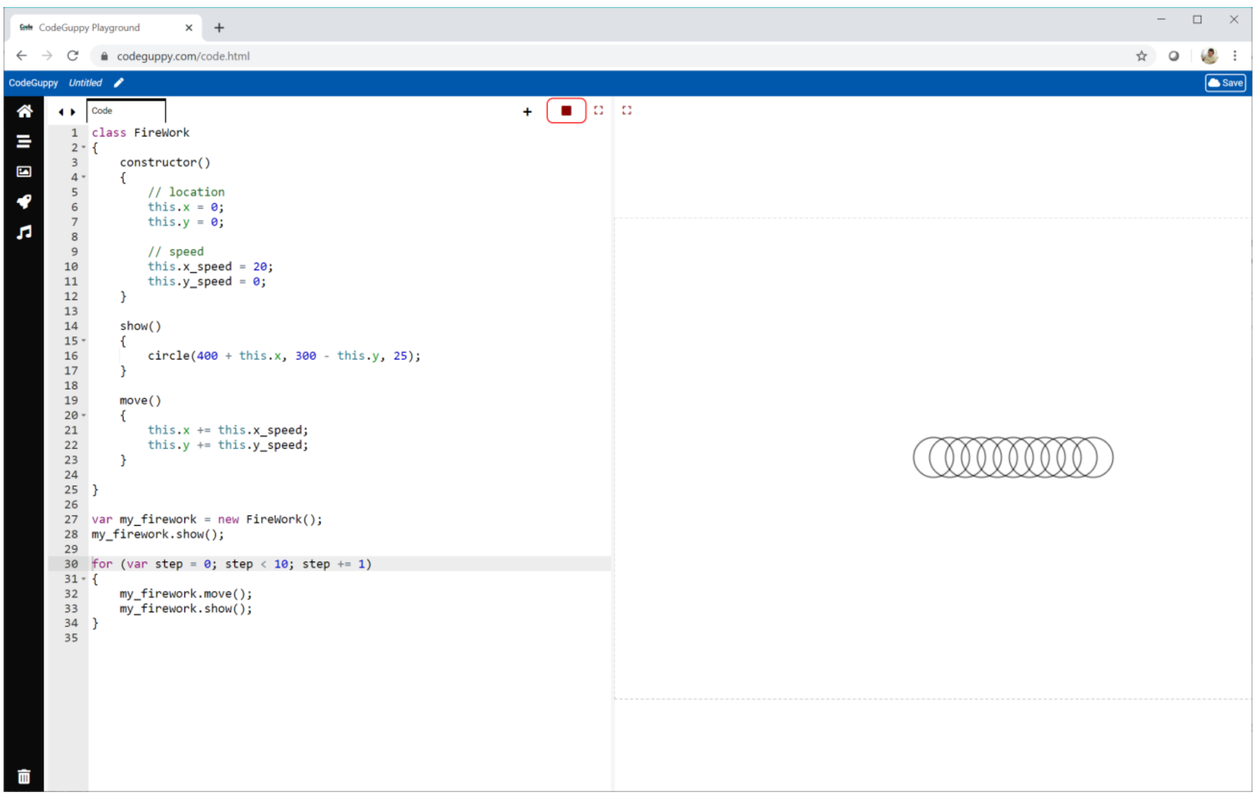
It might not look impressive, but we've done a lot of impressive work to get this far!

Now let’s move our firework for **10** steps.

We can use a **for loop** to repeatedly **move()** and **show()** the firework..

```
my_firework.show();

for (var step = 0; step < 10; step += 1) {
        my_firework.move();
        my_firework.show();
}
```

Here’s the result.



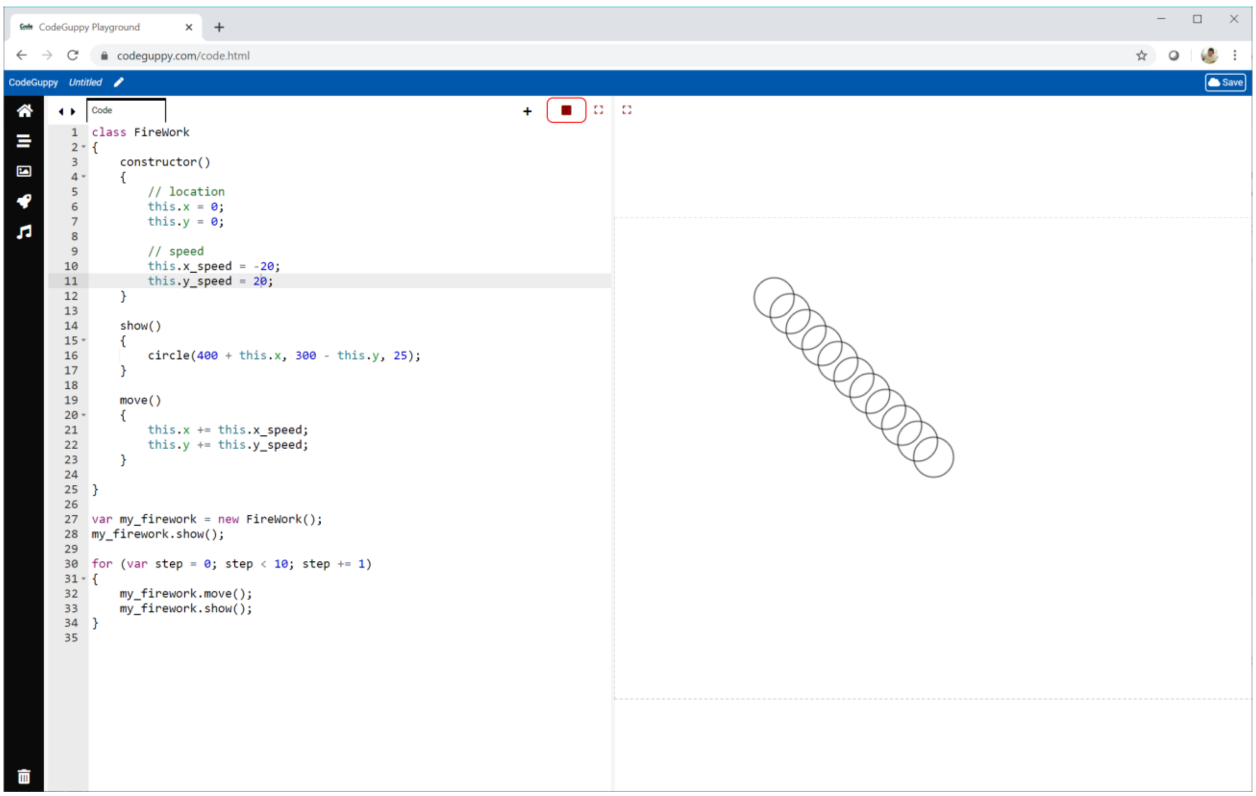
It’s really moving!

Try It Yourself

Change the **FireWork** class and use your own numbers for the the horizontal and vertical speeds.

I tried **x_speed** set to **-20** and **y_speed** set to **20**.

Here's the result:



Try changing the **x** and **y** values set in the **constructor()** method.

What effect does this have?

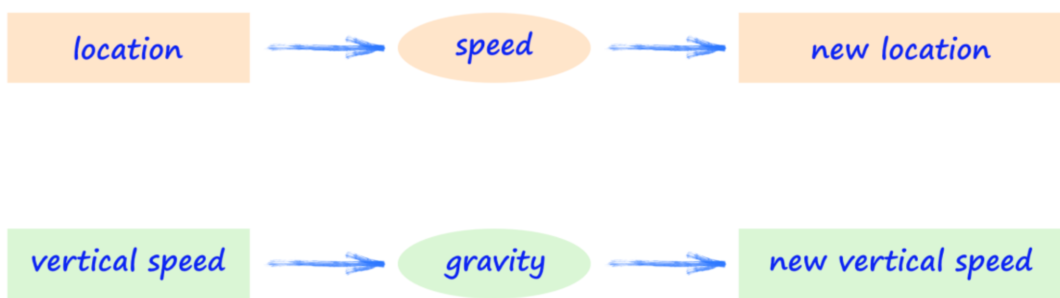
Gravity Makes Things Fall

Real fireworks don’t travel in straight lines. If they go up, they eventually slow and fall back down.

That’s **gravity**!

Let’s talk about how to calculate this.

Speed changes the **location**, and **gravity** changes the **speed**.



We can calculate the next vertical speed **y_speed** by subtracting a value from it. Subtracting a bigger value means stronger gravity.

Gravity only pulls things down, not across. That’s why it only affects **y_speed**, not **x_speed**.

Let’s update the **move()** method to include gravity.

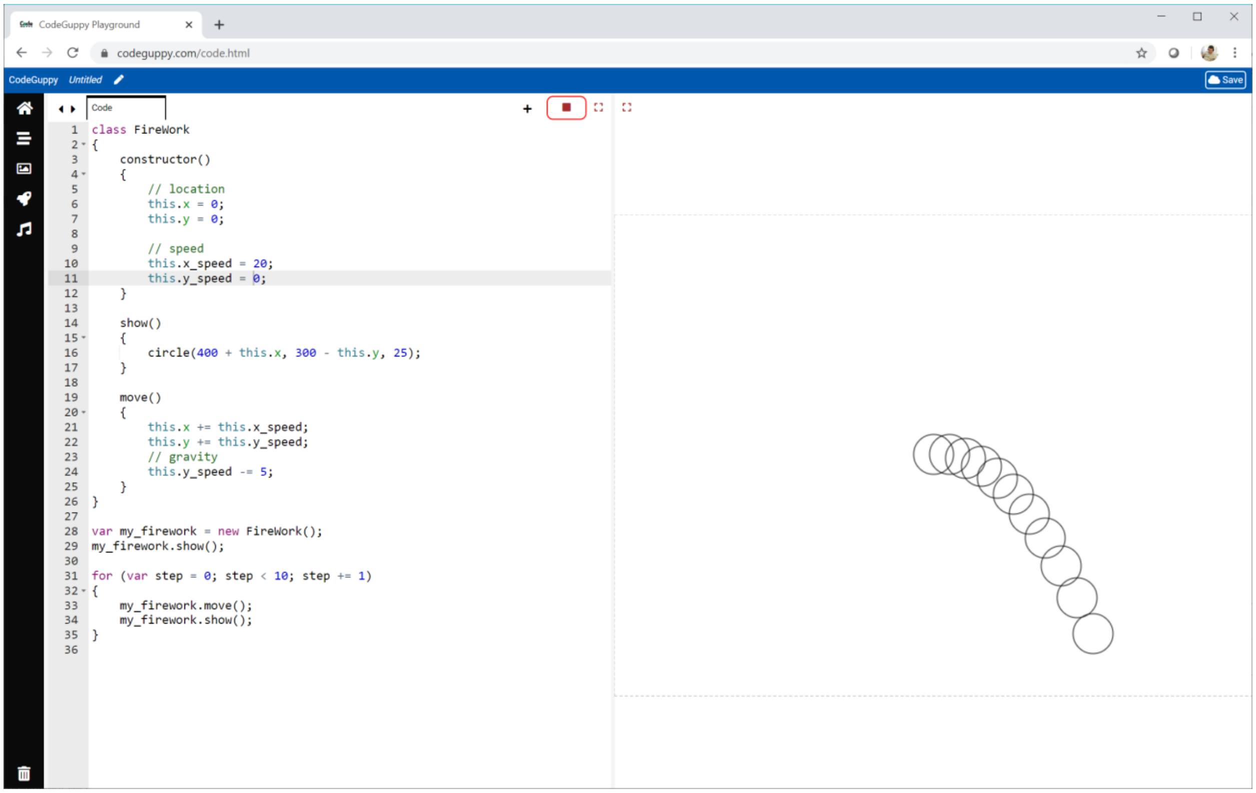
```
move() {
    this.x += this.x_speed;
    this.y += this.y_speed;
    // gravity
    this.y_speed -= 5;
}
```

That’s a very simple change to the code!

We’re trying a gravity strength of **5** to see how it works.

Change the starting **x**, **y** and **y_speed** back to **0**, and **x_speed** back to **20**.

Now run the code again to see the effect of gravity.



That's much more realistic.

You've just made something fall under gravity. That's pretty impressive!

The code for this program is online at:

- https://codeguppy.com/code.html?tariq/ex17

Lots of Fireworks

Let’s have not one, not two, .. but **100** fireworks!

Coding our firework as a **class** helps us. We can easily create **100** firework **objects** using a loop.

First we create an empty list.

```
var list_of_fireworks = [];
```

Then we use a **for loop** to add **100** new **FireWork** objects to the list.

```
for (var count = 0; count < 100; count += 1) {

    list_of_fireworks.push( new FireWork() );

}
```

You’ll recognise the **new Firework()** instruction to create a fresh new **object** from the blueprint.

This new **object** is added to the **list_of_fireworks** using **push()**.

After the loop finishes, that list will have **100** new **FireWork** objects.

Now we have **100** fireworks in a list, we need to **move()** and **show()** each one.

Have a look at this new kind of **for loop**.

```
for (var firework_object of list_of_fireworks) {
     firework_object.move();
     firework_object.show();
}
```

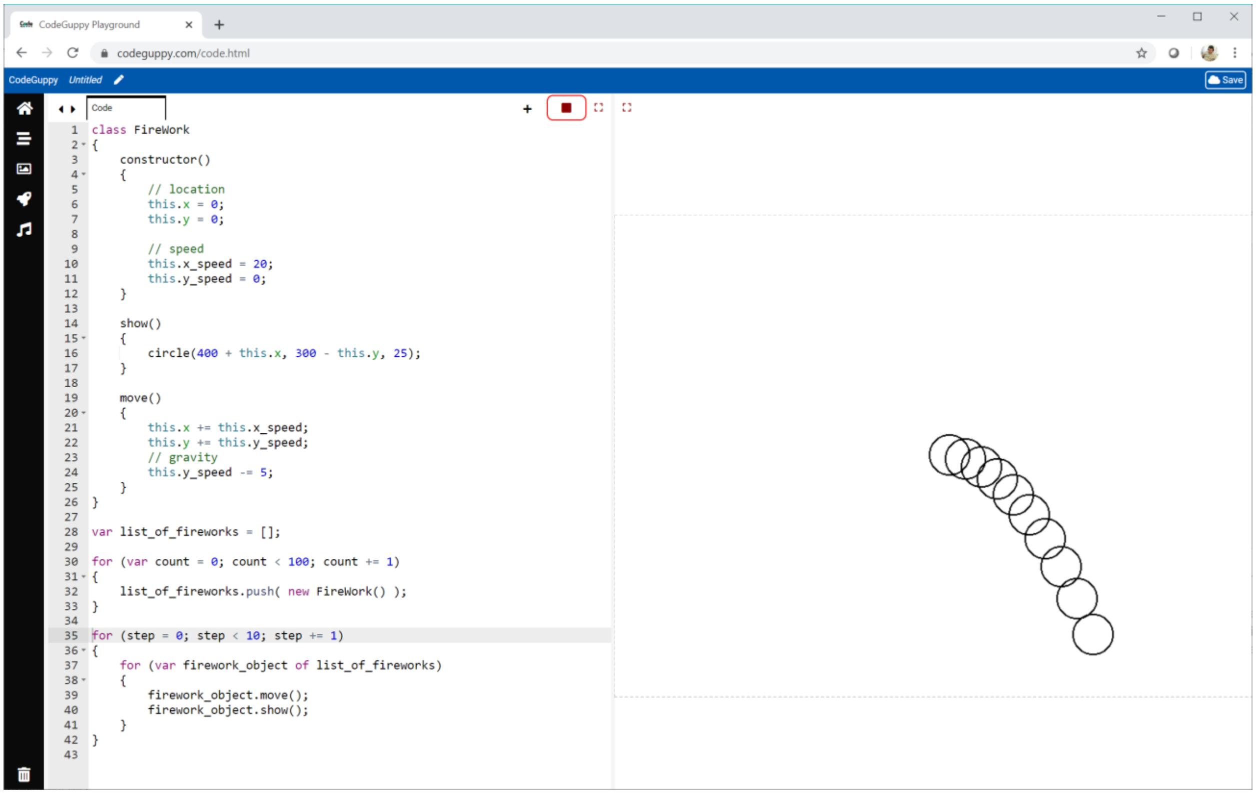
This **for loop** works through each item in the **list_of_fireworks**, and calls it **firework_object** as it runs the code inside the curly brackets.

This will only **move()** and **show()** each firework once. We need another loop around this one if we want to move each firework **10** steps.

```
for (step = 0; step < 10; step += 1) {

     for (var firework_object of list_of_fireworks) {
          firework_object.move();
          firework_object.show();
     }

}
```

Let’s run this code to see if it works.

Here’s what the code draws.



Where are the **100** fireworks?

There are **100** fireworks, but they’re all starting at exactly the same location, with exactly the same speed. That means they all follow exactly same journey.

Let’s change our **FireWork** class so the starting position and speeds are random, but close to the centre of the canvas.

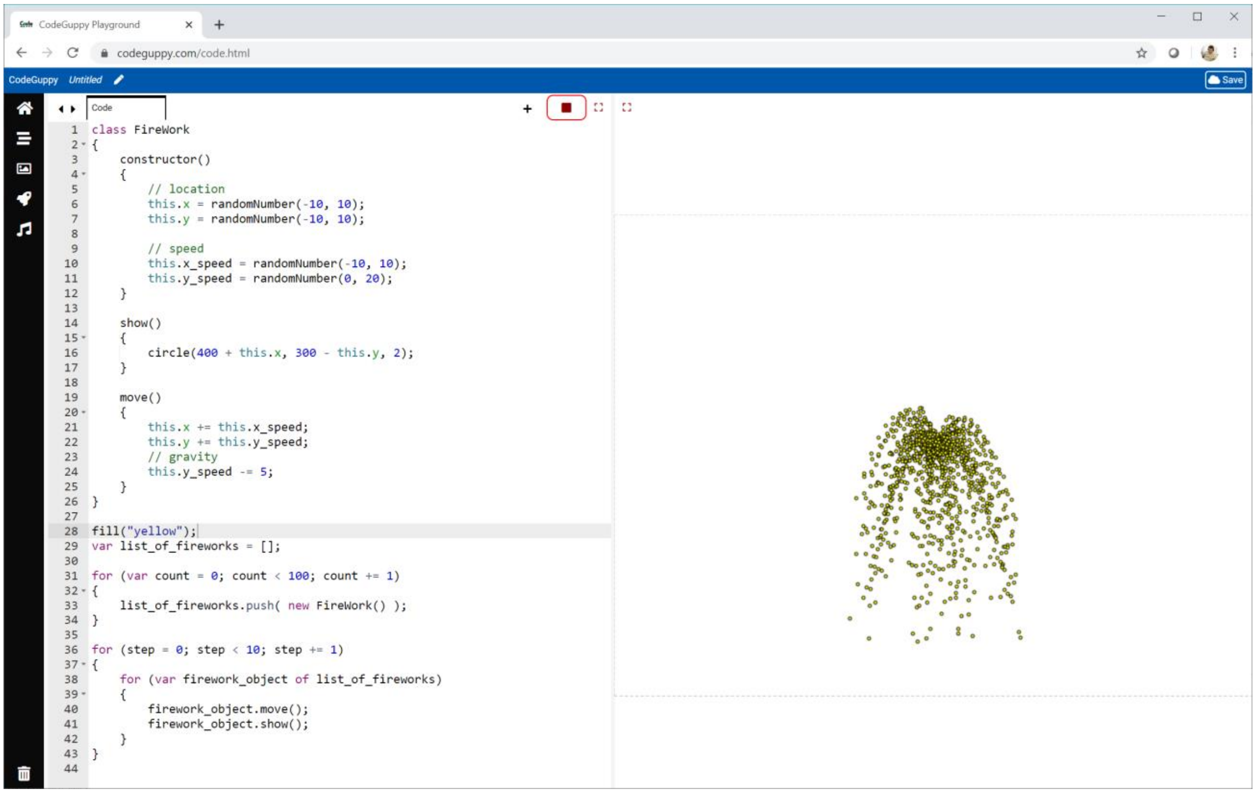
```
constructor() {
    // location
    this.x = randomNumber(-10, 10);
    this.y = randomNumber(-10, 10);

    // speed
    this.x_speed = randomNumber(-10, 10);
    this.y_speed = randomNumber(0, 20);
}
```

The horizontal **x_speed** is a random number between **-10** and **+10**. A negative speed means moving left.

The vertical **y_speed** is a random number between **0** and **20**. That means the firework will start moving upwards, not downwards.

Let’s also change the circle size to a smaller **2** in the **show()** method and filled the circles with yellow.

Let’s see the results so far.



That's starting to look more like a firework display.

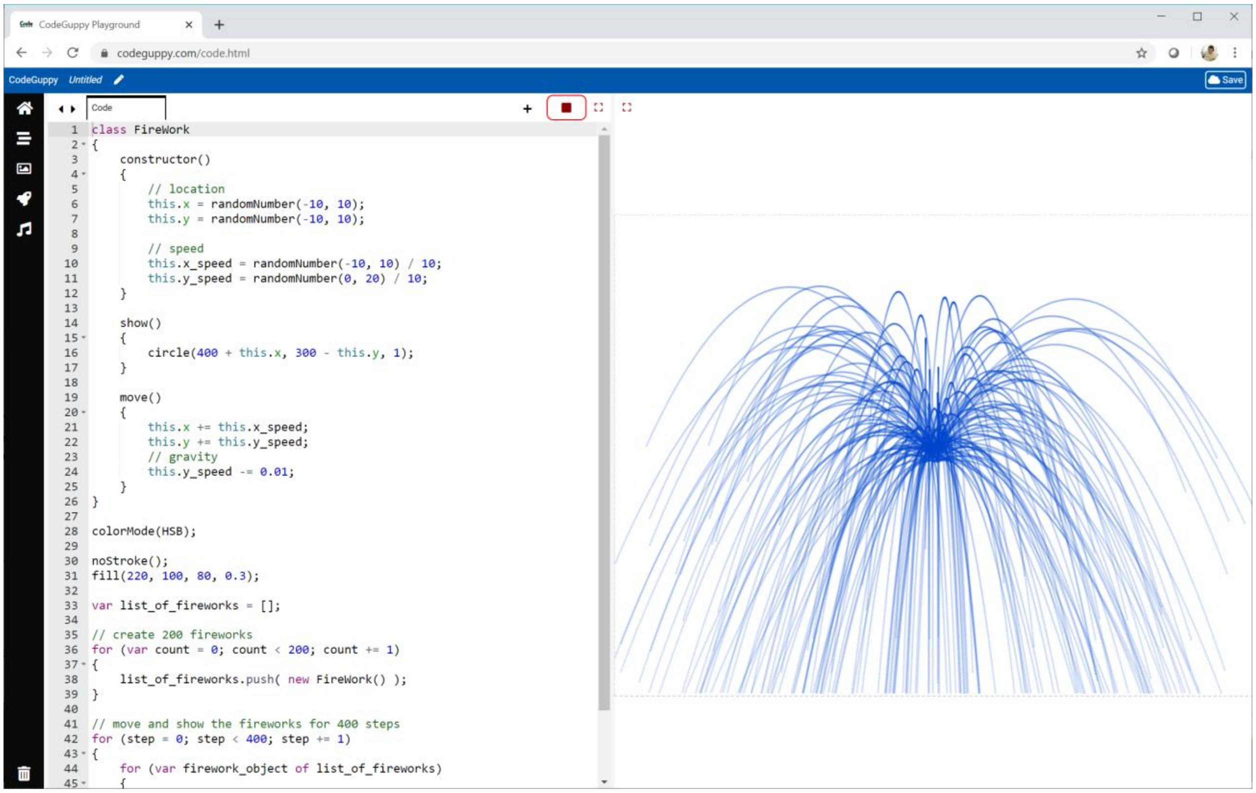
We can tweak a few things to make the image look nicer:

- Increase the number of journey steps from **10** to **400**.

- Change the circles to size **1**, with no stroke and a **HSB** colour **(220, 100, 80, 0.3)**. The translucency will help with this busy drawing.

- Make the journey steps smaller by making the speeds and gravity smaller. Divide the **x_speed** and **y_speed** by **10** so they are **randomNumber(-10, 10) / 10** and **randomNumber(0, 20) / 10**. Change the gravity from **5** to **0.01**.

Let's see the result.



Pretty cool.

Now we need to add explosions!

We can add an **explode()** method to the **FireWork** class like this.

```
explode() {
    circle(400 + this.x, 300 - this.y, 10);
}
```

The explosion is just a larger circle.

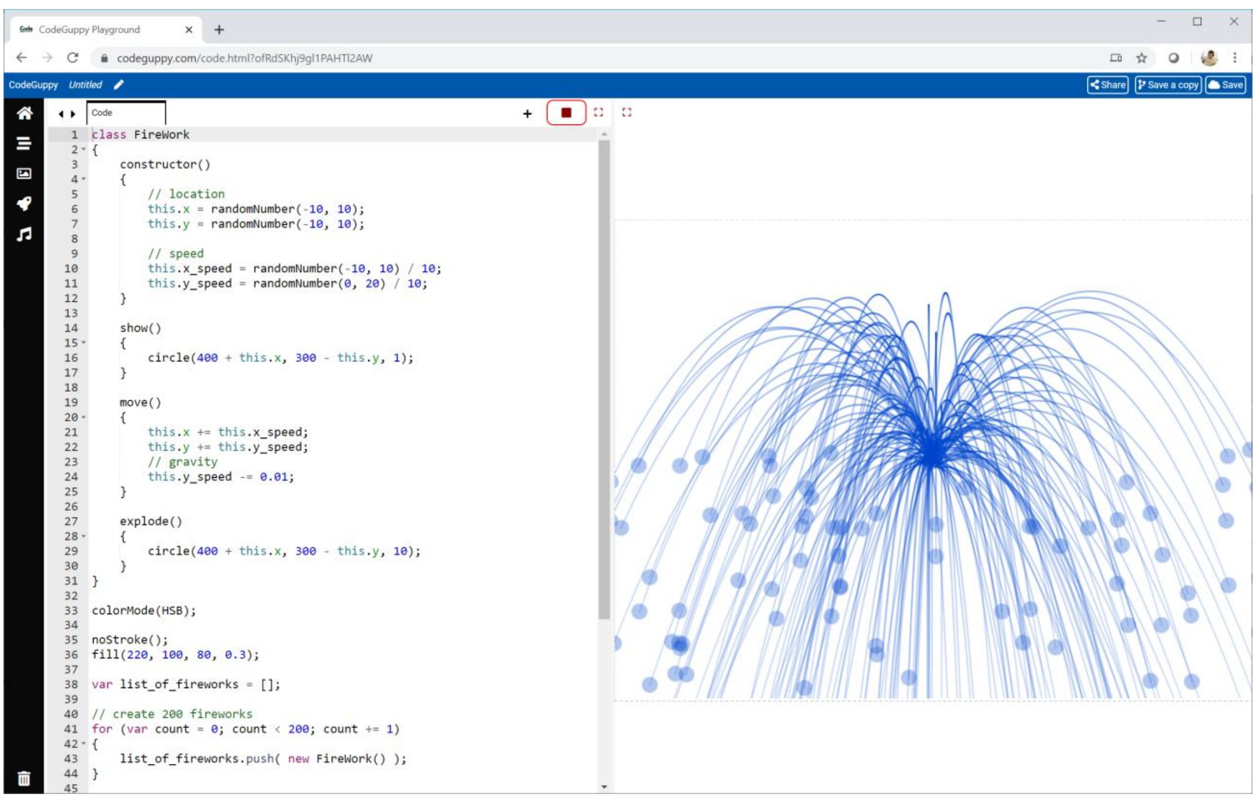
We only explode the fireworks at the end of their journey. That means we call the **explode()** method after the loops which follow the firework for **400** steps.

```
for (var firework_object of list_of_fireworks) {
        firework_object.explode();
}
```

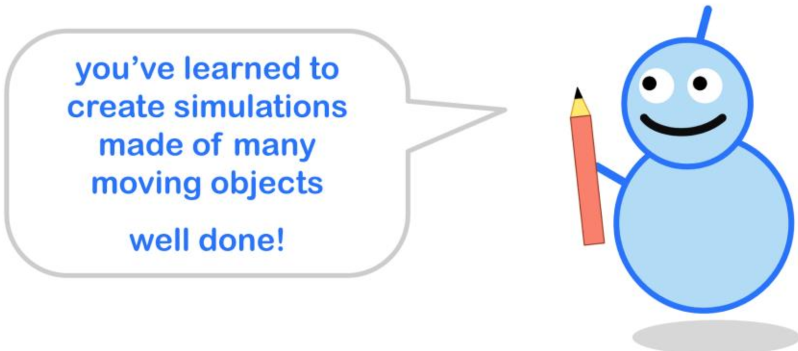
The code we've written so far is at:

- https://codeguppy.com/code.html?tariq/ex18

Let's see the results.



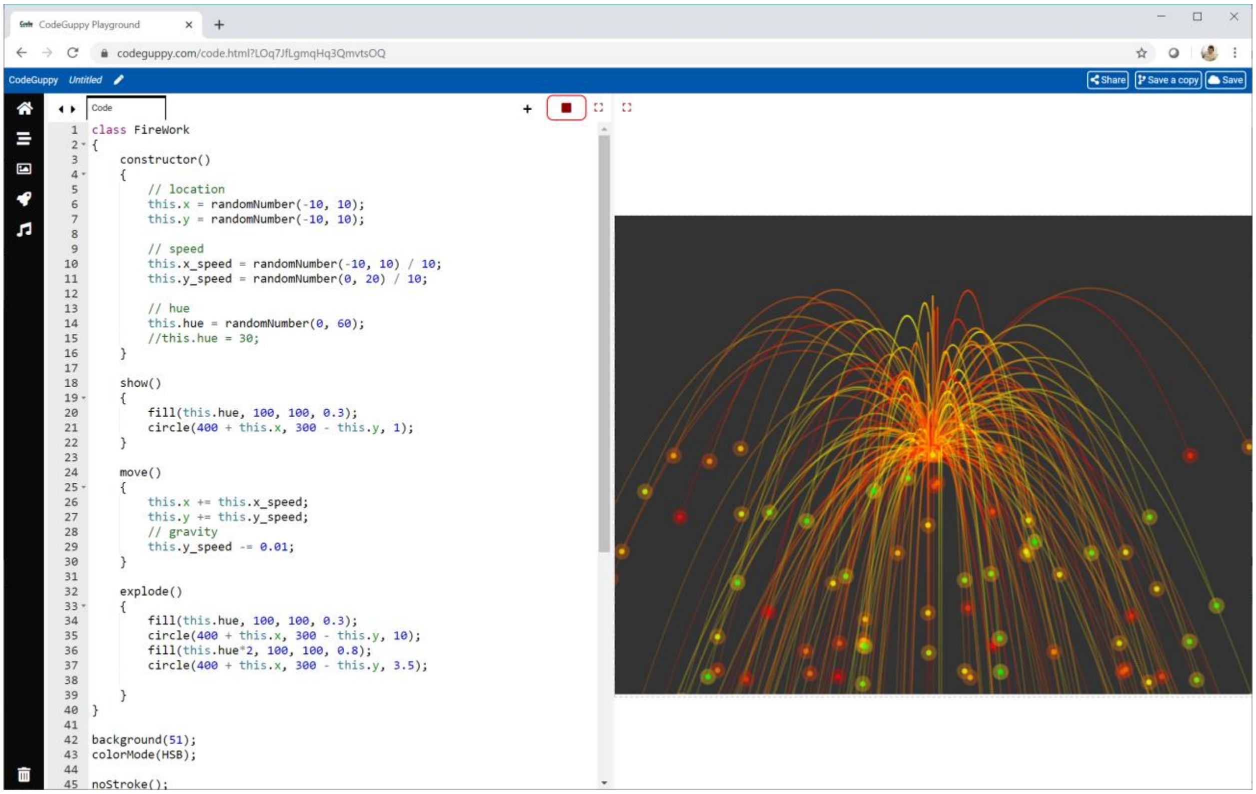
Pretty!


you’ve learned to create simulations made of many moving objects
well done!

Try It Yourself

Change the **FireWorks** class so the colour of the fireworks isn't blue.

Here's my own experiment.



I've changed the background to a darker grey, and the firework **hue** is picked at random from the range **0** and **30** in the constructor.

I've also changed **explode()** to draw two circles. A bigger circle has the same hue as the firework, and a smaller circle inside it has double the hue.

The code for my experiment is online:

- https://codeguppy.com/code.html?tariq/ex19

Sky Full Of FireWorks

Remember how we passed information to a function using the round brackets after its name?

We can do the same when creating an object. Have a look at this code.

```
new FireWork(100, 100);
```

Here we're creating a new **FireWork** object and passing two numbers to it. They get passed to the **constructor()** method.

We can use this to pass information about where we want the firework to start its journey.

Have a look at this code:

```
constructor(x, y) {
    // location
    this.x = x + randomNumber(-10, 10);
    this.y = y + randomNumber(-10, 10);
```

This **constructor()** now expects two bits of information, and calls them **x** and **y**. They are used to calculate the **this.x** and **this.y** starting locations.

So **new FireWork(100, 100)** would create a **FireWork** object starting at a random point very close to **(100, 100)**.

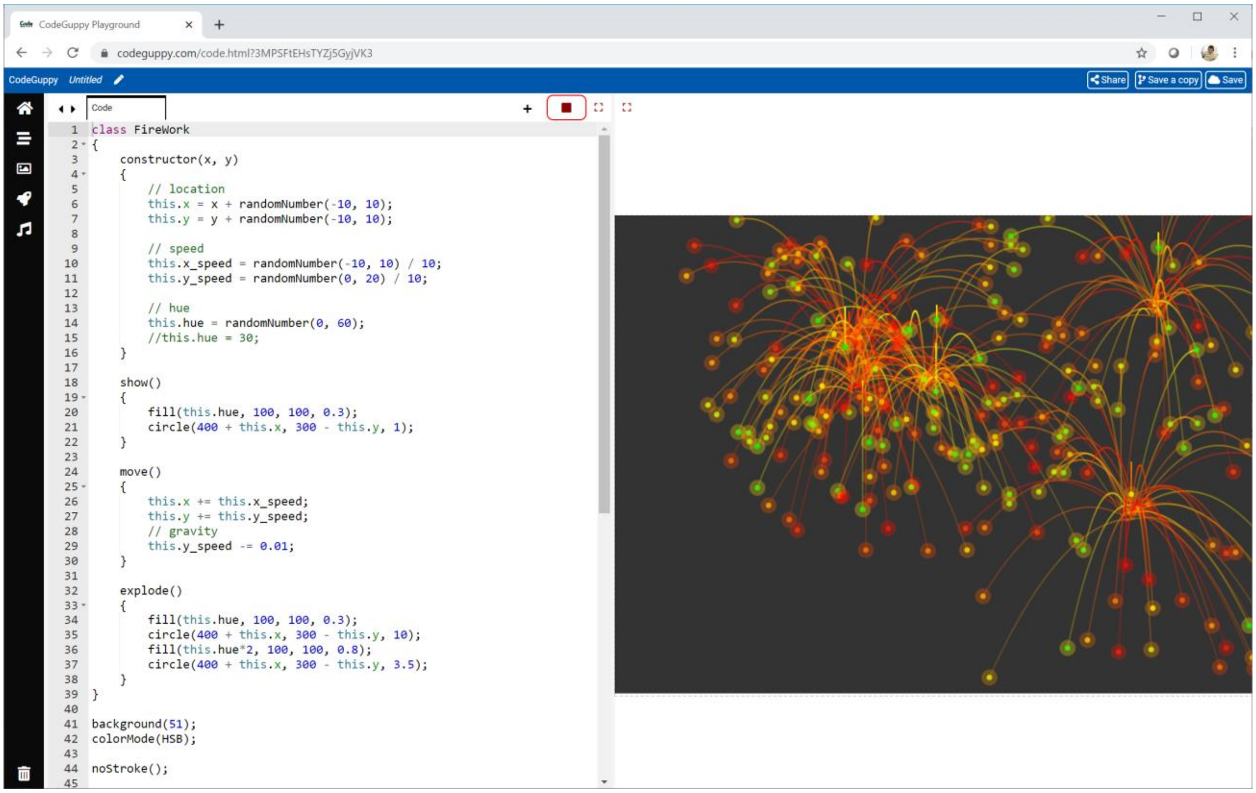
We can use a loop to create clusters of fireworks, with each cluster being centred at a random point on the canvas.

```
// firework clusters
for (var cluster = 0; cluster < 5; cluster += 1) {

     var x = randomNumber(-300, 300);
     var y = randomNumber(-100, 200);

     // create many fireworks at that x, y cluster
     for (var count = 0; count < 50; count += 1) {
          list_of_fireworks.push( new FireWork(x,y) );
     }

}
```

You can see that for each of the **5** clusters, a random **x** and **y** is chosen. These are used to create **50** new **FireWork** objects in that cluster.

The next cluster will have a new **x** and **y** chosen at random, so the **50** fireworks in that cluster will be centred around this new location.

Let’s see the effect of having **5** clusters of **50** fireworks.



That is pretty impressive!

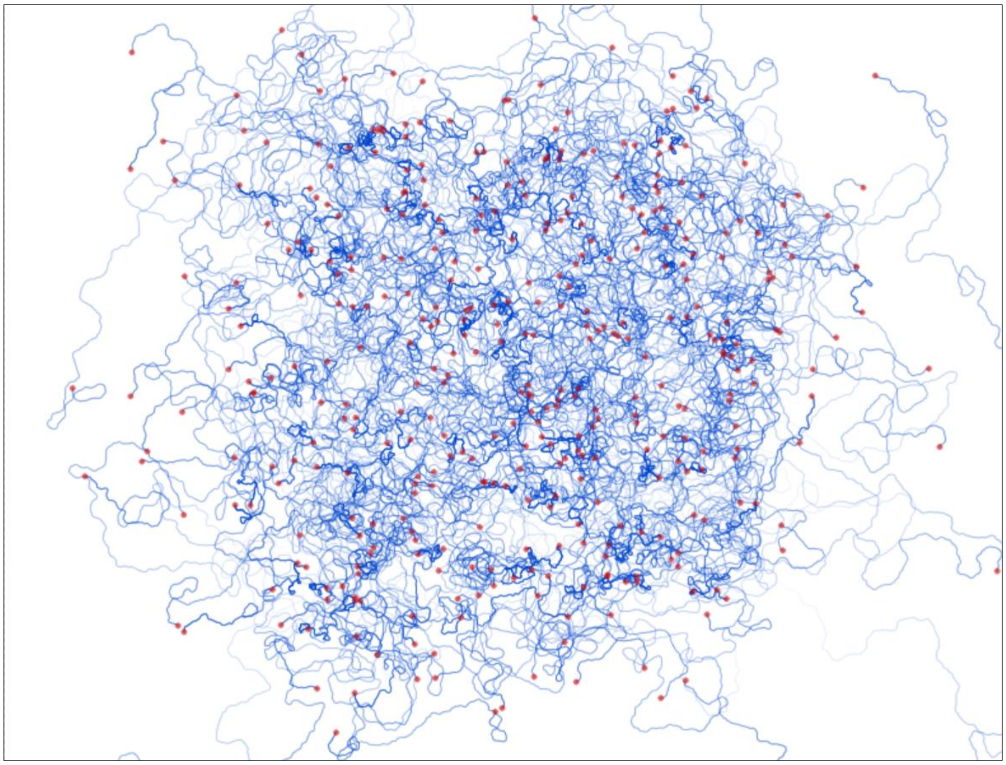
Run the code again to see different results.

The code for this program is online:

- https://codeguppy.com/code.html?tariq/ex20

Challenge!

Have a look at this interesting pattern.



The paths are journeys made by objects that move in small steps in **directions** that change by small random amounts.

We can think of these objects as little crawling **ants**.

The challenge is to write code to create similar ant paths.

The following details explain how the pattern was drawn.

- There are **400** ants, which are followed for **600** steps.
- The ant class constructor sets
    - an **x** and **y** location at random near the middle of the canvas
    - a random initial **direction** using an **angle**
    - a **step size** chosen at random between **0.2** and **1.0**
    - an initial **alpha** value of **0.001**
- For each step, each ant
    - moves by **step size** in the **direction** it is pointing in
    - changes its direction by adding a random angle from the range **-30** to **+30** degrees
    - its **alpha** value is increased by **0.001**
- An ant's path is shown using a small circle of size **1**, and with a translucency set by its **alpha** value.
- At the end of the path, a small red translucent circle of size **5** is drawn.

If you need to look at my code it is online:

- https://codeguppy.com/code.html?tariq/ex21

3.6 - Code That Creates Code

level

What We'll Do

In this project we're going to:

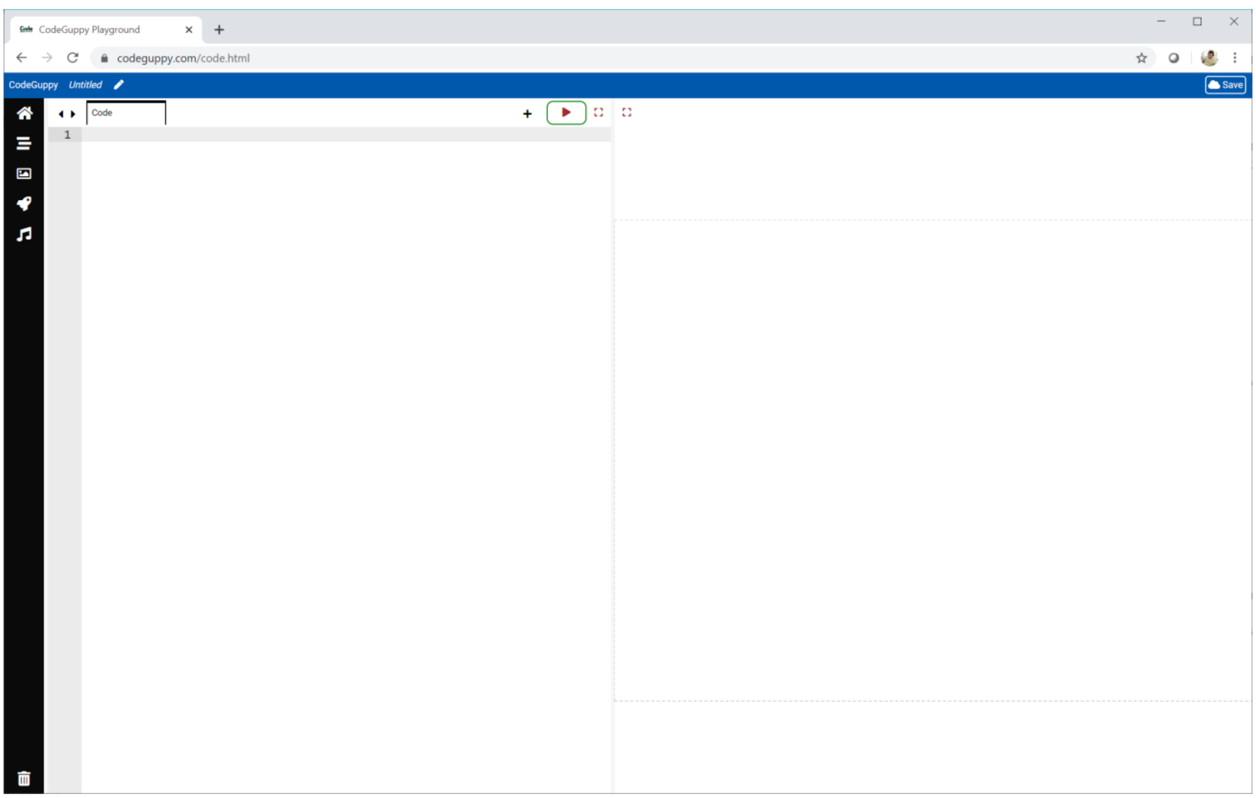
- create our own **turtle language**, and write an **interpreter** for it.
- evolve turtle code as an **l-system** to draw interesting patterns.

Start a New Program

Log in to codeguppy.com if you haven't already.

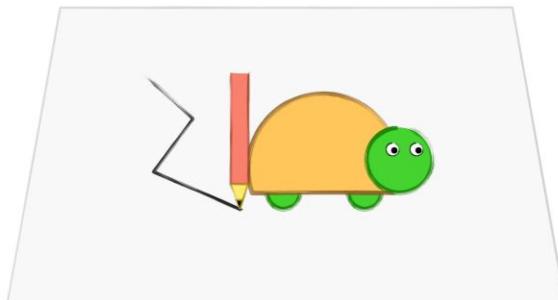
Create a new program, just like we did in the first project.

Your code window should look like this:



Turtle Code

Here’s a turtle with a pen. She’s very artistic and loves to draw.



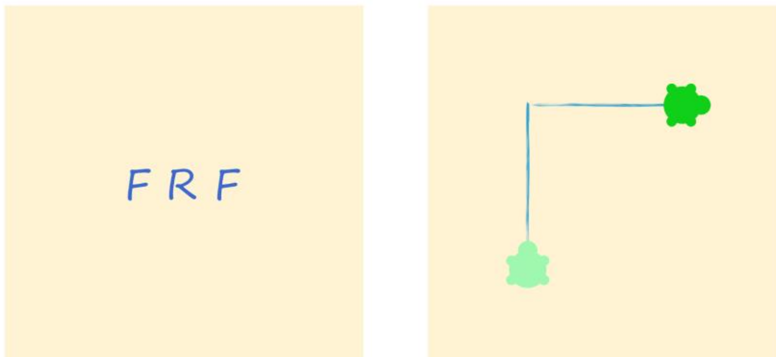
Our friendly turtle can follow simple instructions.

Here’s our turtle following the instruction **F**, which means go **forward**.

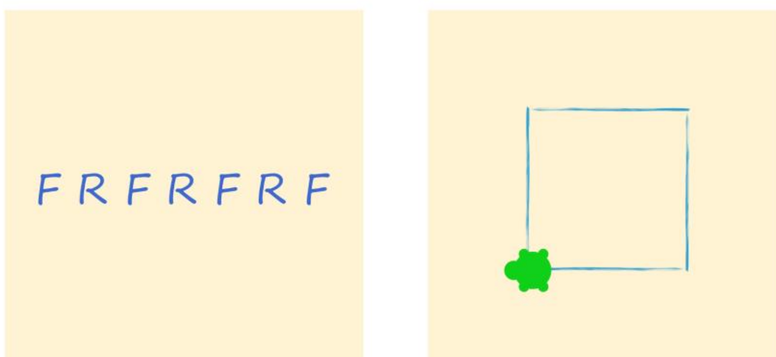
Her pen draws a line as she moves forward.

Our turtle can also understand **R** which means turn **right**.

The instructions **FRF** mean go **forward**, turn **right**, then go **forward** again. You can see this draws a shape that looks like a corner.



Can you work out what the instructions **FRFRFRF** will draw?



Our turtle also understands **L** which means turn **left**.

We can call these instructions **turtle code**. What a cool name!

Let’s Code A Turtle

Let’s make a turtle that can run **turtle code**.

It makes sense to use a **class** because our turtle will have:

- **data** - keeping track of where it is on the canvas
- **methods** - to move forward, and turn left and right

Have a look at this first go at designing a **Turtle class.**

```
class Turtle {

    constructor() {
        // location
        this.x = 0;
        this.y = 0;
        // direction
        this.angle = 0;

    }

}
```

The **this.x** and **this.y** variables keep track of the turtle’s location.

We also need to know which way the turtle is facing. We can use an angle **this.angle** to keep track of its direction.

If we created an object from this class, it wouldn’t be able to do much because we haven’t written any methods yet.
Let’s write a method to turn our turtle **left**.

When our turtle turns **left**, it doesn't change location. That means the **this.x** and **this.y** coordinates don't change.

The only thing that changes is the turtle's direction.

Turning **left** means turning **90 degrees** anticlockwise.

```
// turn left
left() {
     this.angle += 90;
}
```

That wasn't too hard!

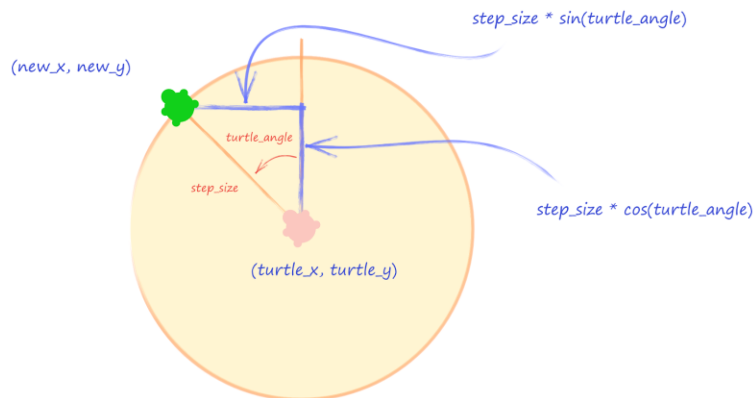
When our turtle turns **right**, it turns **90 degrees** clockwise. That means subtracting **90 degrees**.

```
// turn right
right() {
     this.angle -= 90;
}
```

Let's think about writing a method to move our turtle **forward**.

Moving forward depends on the direction the turtle is facing. That means we need to use turtle object's **angle**.

We can use the **trigonometry** we saw earlier to calculate the new **x** and **y** coordinates. The next picture reminds us how to do this.



We need to know the step size to work out the new location, so let's add that to our constructor. We can choose a step size of **50** for now.

```
constructor() {
          // location
          this.x = 0;
          this.y = 0;
          // direction
          this.angle = 0;
          // step size
          this.step = 50;
     }
```

Have a look at this code for a **forward()** method.

```
forward() {
    // new location
    var new_x = this.x - this.step * sin(this.angle);
    var new_y = this.y + this.step * cos(this.angle);

    // draw line
    line(400 + this.x, 300 - this.y, 400 + new_x, 300 -
new_y);

    // update turtle’s location
    this.x = new_x;
    this.y = new_y;
}
```

First we calculate the new location using the trigonometric relations we learned about before.

We then draw a line from the current location to the new location.

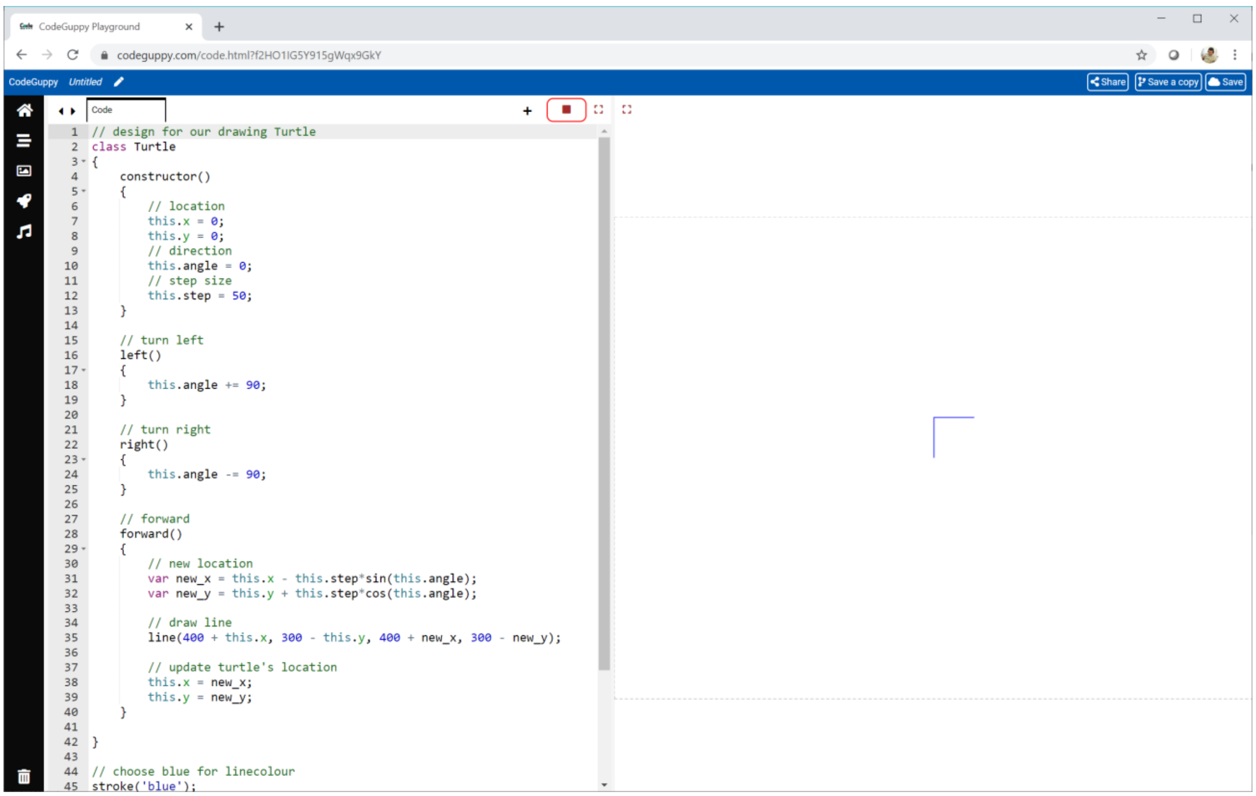
Because the turtle has now moved, we need to update its location with the new position. You can see **this.x** and **this.y** being updated.

We’ve done a lot of work making our **Turtle** class. Let’s use it!

We can create a new turtle **object** and test its **methods** like this:.

```
var my_turtle = new Turtle();

my_turtle.forward();
my_turtle.right();
my_turtle.forward();
```

Run the code to check our turtle does go **forward**, **right** and **forward**.



Yes! Our turtle really does work.

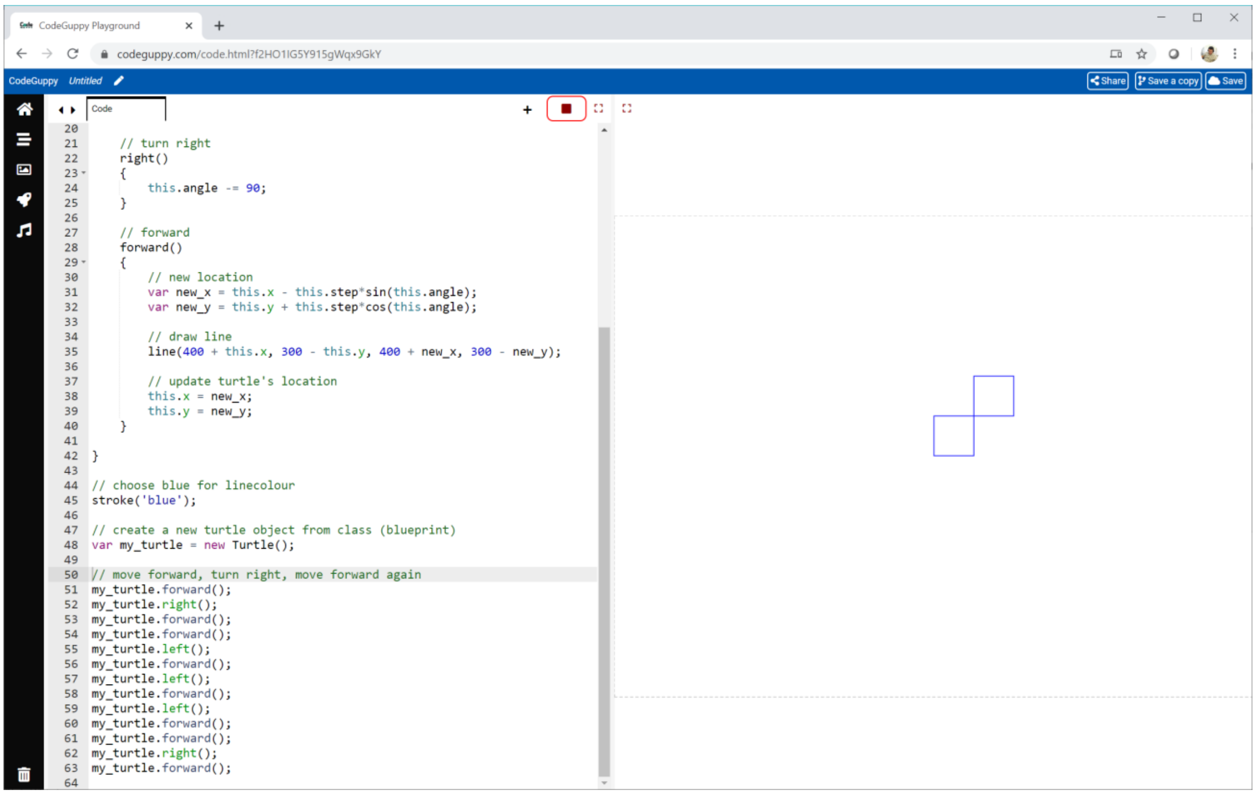
The code so far is at:

- https://codeguppy.com/code.html?tariq/ex22

Try It Yourself

Use the turtle's **forward()**, **left()** and **right()** methods to draw a different pattern.

Here is my own experiment.



Interpreting Turtle Code

Let's see if we can get our turtle to interpret and carry out the **turtle code** we designed earlier.

Our **turtle code** is just a list of instructions, like **FRFRFRF**, carried out one after another.

It makes sense to keep these instructions in a **list**, and use a loop to work through them in order.

```
var turtle_code = ['F', 'R', 'F', 'R', 'F', 'R', 'F'];
```

You can see we've created a new variable called **turtle_code**. The square brackets tell our computer the variable points to a **list**. Inside the list are **7** turtle code instructions.

Here's a easier way of writing the same thing.

```
var turtle_code = [...'FRFRFRF'];
```

Those three dots simply split the text string **'FRFRFRF'** into a **list** of letters.

So far so good.

Now we need to work through those instructions and carry them out.

We've written loops to work through a list many times before.

```
for (var instruction of turtle_code) {
     // do something with the instruction
}
```

This for loop works through the list of turtle code and temporarily names each one **instruction**.

What do we do now?

Let's think about it in plain English. If the instruction is **F** then our turtle moves **forward**. If the instruction is **R** it turns right. And if it is **L** then it turns **left**.

Those looks like **if()** tests which are easy to code.

When you have a hard coding problem, something thinking about it in plain English can help solve it.

```
for (var instruction of turtle_code) {
     // forward
     if (instruction == 'F') {
          my_turtle.forward();
     }
}
```

You can see the code checking to see if the **instruction** is **F**, and if it is, calling our turtle's **forward()** method.

That wasn't so hard.

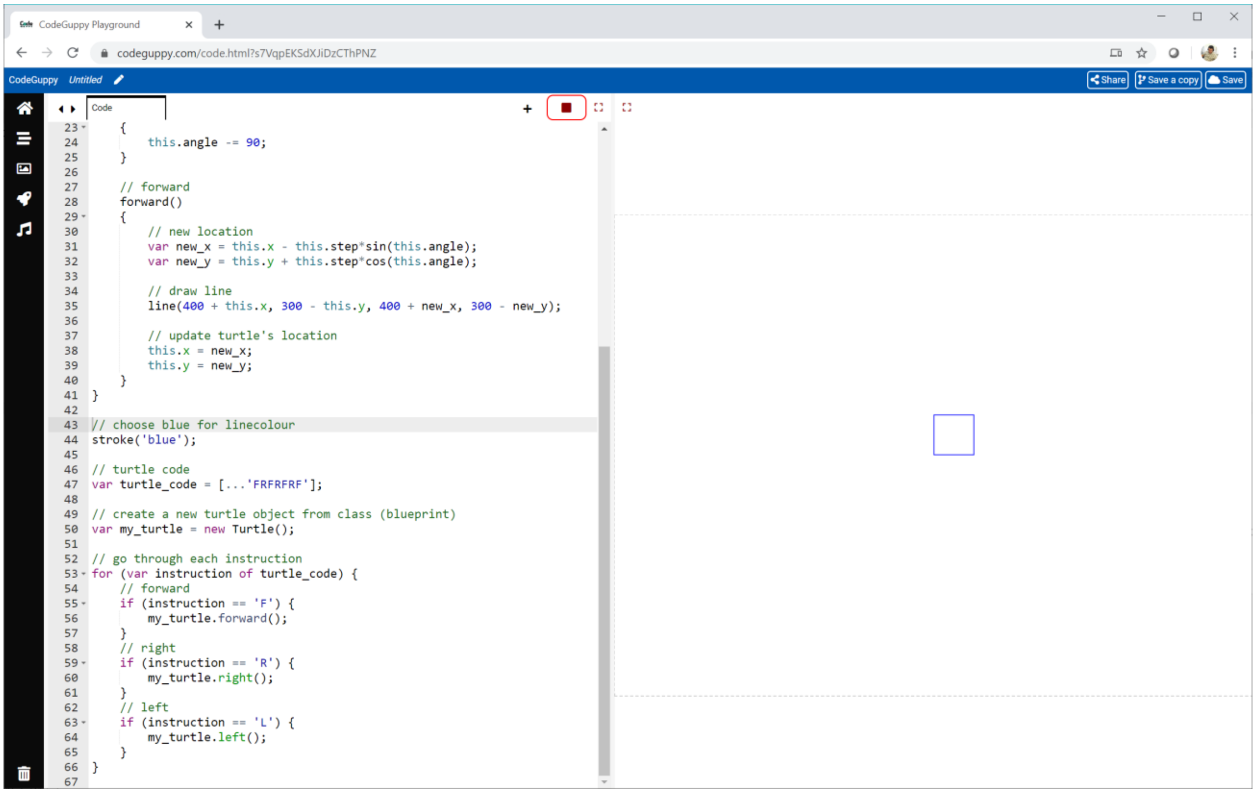
We can easily write **if()** tests for the **R** and **L** instructions too.

```
// go through each instruction
for (var instruction of turtle_code) {
    // forward
    if (instruction == 'F') {
        my_turtle.forward();
    }
    // right
    if (instruction == 'R') {
        my_turtle.right();
    }
    // left
    if (instruction == 'L') {
        my_turtle.left();
    }
}
```

That looks like a lot of code, but the idea is simple.

Let’s run our program.

It should now work through the turtle code **FRFRFRF** to draw a square.



That worked!

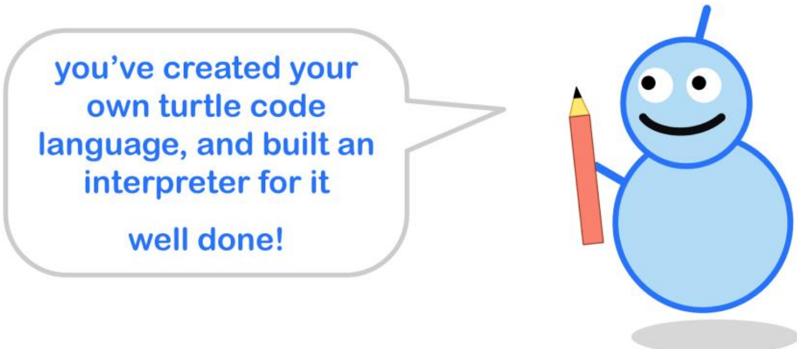
Even though that square isn't very impressive, what we've done is impressive.

We've taught our turtle to read, understand and carry out **turtle code** instructions!

Computer scientists call what we've created an **interpreter**.

The code we're created so far is online:

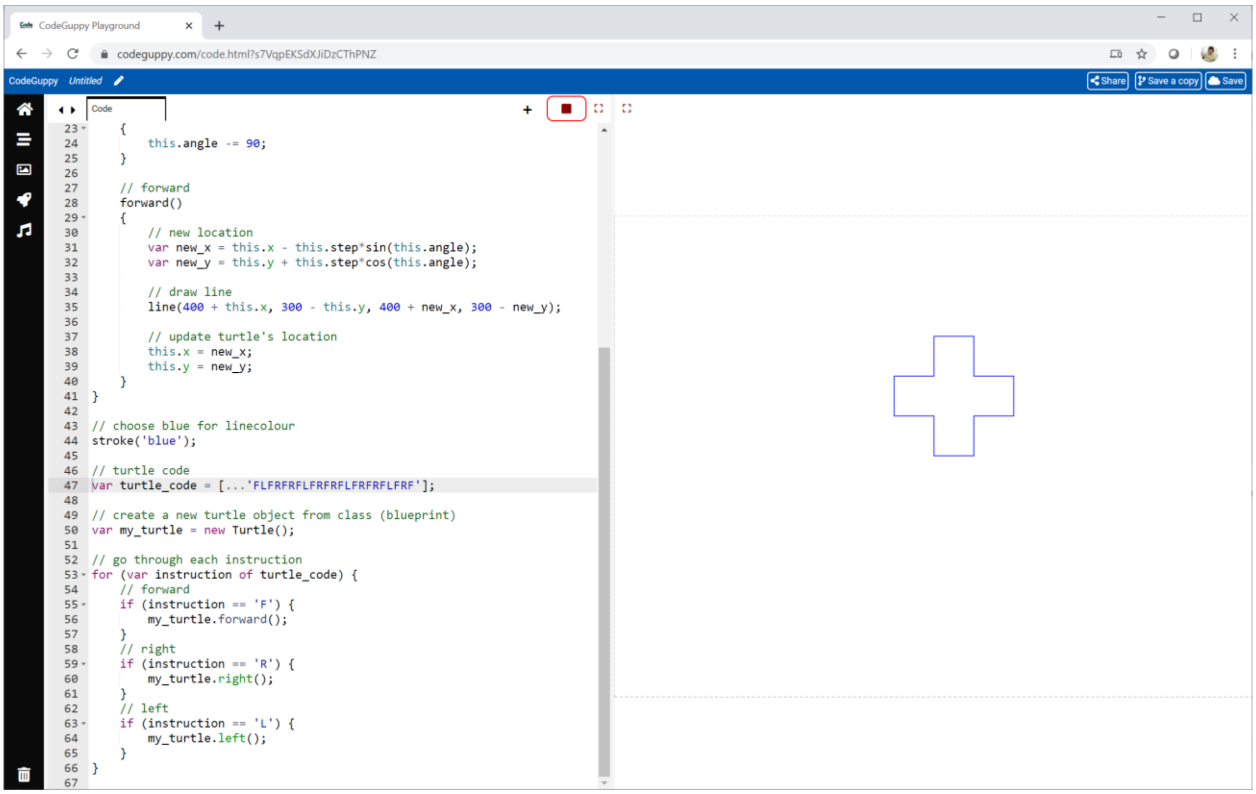
- https://codeguppy.com/code.html?tariq/ex23


you've created your own turtle code language, and built an interpreter for it
well done!

Try It Yourself

Try experimenting with your own turtle code.

Here’s my own experiment.



The turtle code for this pattern is **FLFRFRFLFRFRFLFRFRFLFRF**.

Remembering Where We Were

Let’s give our turtle more powers.

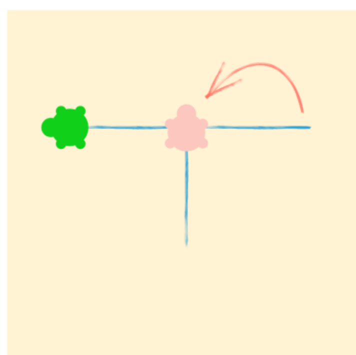
It would be good if our turtle could remember a point, carry on drawing, and then come back to that point later, and carry on from there.

We can invent two new **turtle code** instructions for doing this:

- an opening square bracket **[** for marking the point you want to come back to

- a closing square bracket **]** for going back to the last remembered point

This next picture shows our turtle using these two new instructions.



See if you can follow the code:

- The turtle first moves forward, **F**.
- The next instructions is **[** so the turtle makes a note of this location. You can see this point marked with a pink turtle.
- It then carries on by turning right and going forward, **RF**.
- The next instruction **]** tells our turtle to go back to the saved point.
- After that the turtle turns left and goes forward, **LF**.

How do we code this new ability?

When our turtle meets a **[** instruction it will need to make a note of where it is. That means saving the its location and angle.

We can add a method for saving the turtle's **state** like this:

```
savestate() {
     this.state.push(this.x);
     this.state.push(this.y);
     this.state.push(this.angle);
}
```

We're pushing **this.x**, **this.y** and **this.angle** onto the end of a list **this.state** which we need to first create in the constructor.

Why save to a list? Why don't we just save the state using normal variables?

The reason is that we might see more than one **[** instruction so we need to save more than one point before we return to them.

When we return to a previously saved state, we need to pick off this information in reverse order.

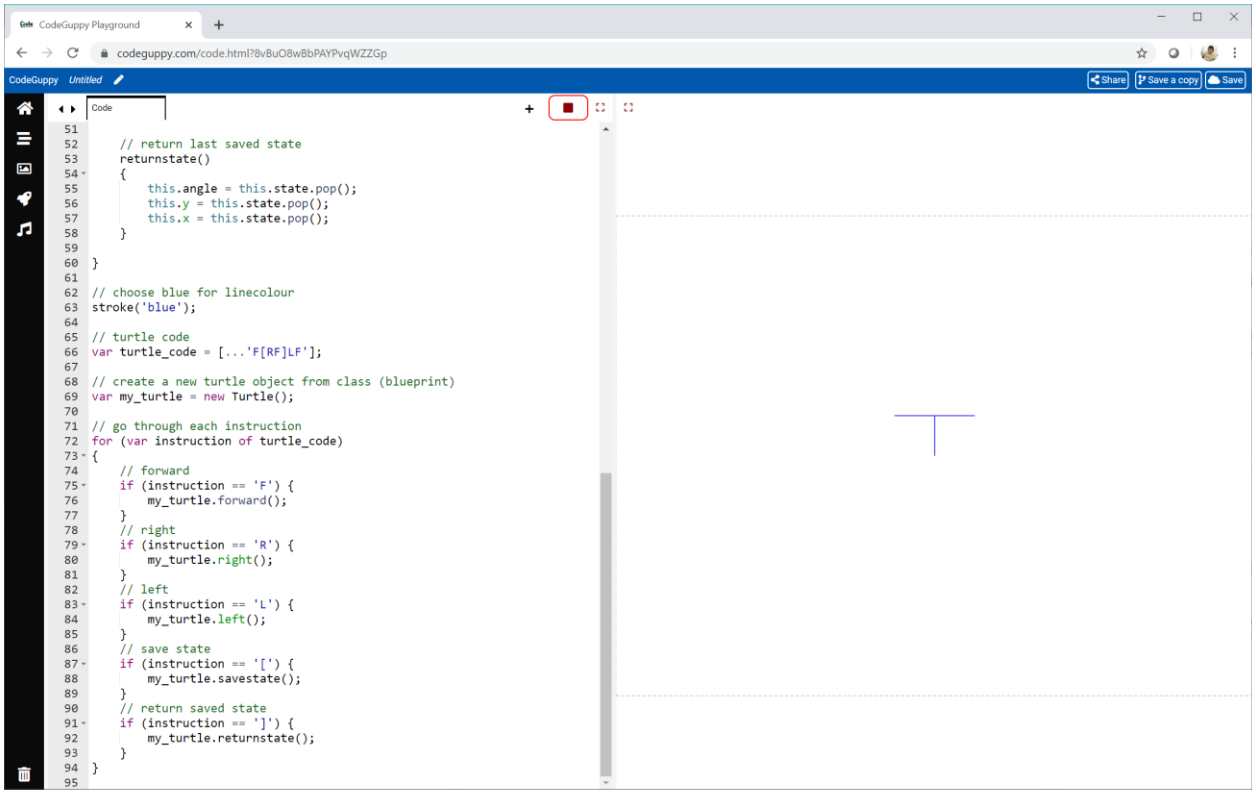
```
returnstate() {
     this.angle = this.state.pop();
     this.y = this.state.pop();
     this.x = this.state.pop();
}
```

The last thing we put on the list was **this.angle**, which is why it is the first thing to be taken off.

Add these methods to your **Turtle** class, and also add the code to call these methods when the **[** and **]** are found.

Test your turtle's new powers with the turtle code **F[RF]LF**.

You should get that T-shape we saw earlier.



That T-shape means our turtle correctly interprets the **[** and **]** instructions.

The code so far is online at:

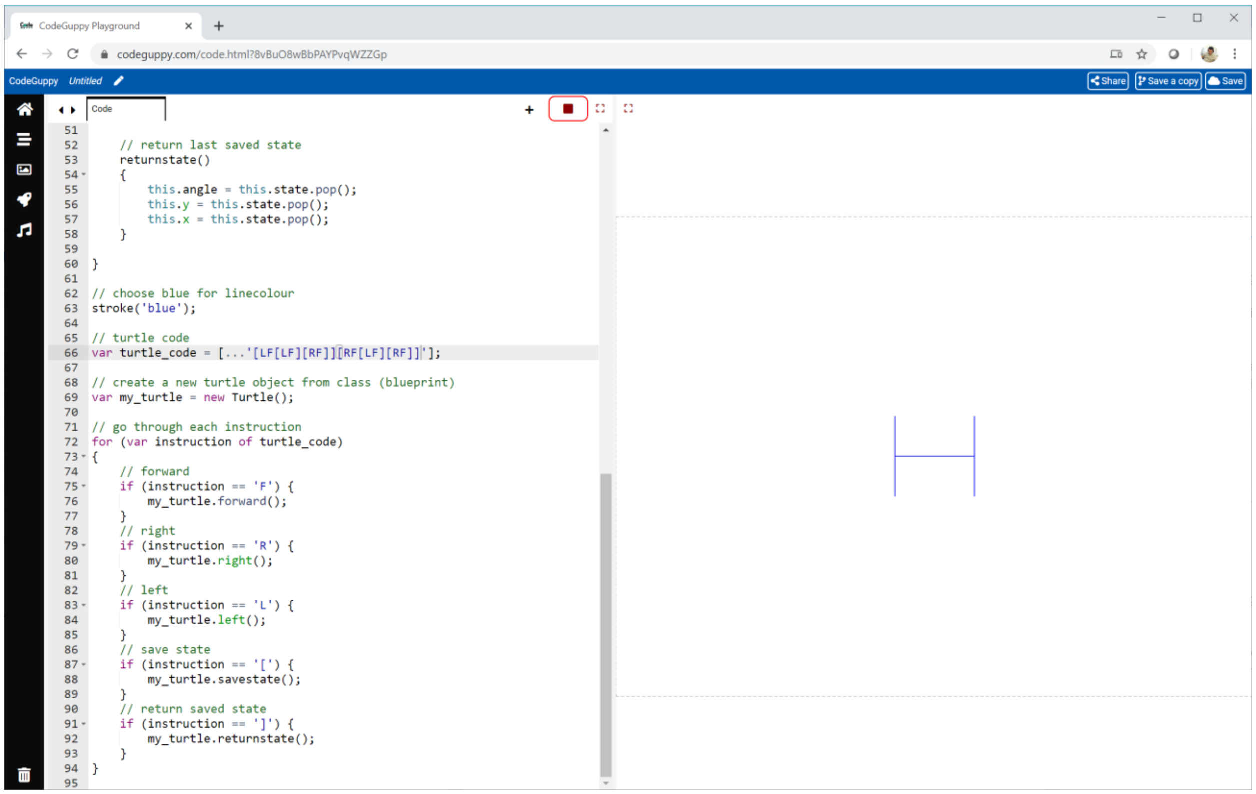
- https://codeguppy.com/code.html?tariq/ex24

Try It Yourself

Have a go at writing **turtle code** with several **[** and **]** instructions.

You can save state several times, but you must return it later. That means every **[** must have a matching **]** later in the code.

Here is my experiment using **[LF[LF][RF]][RF[LF][RF]]**.

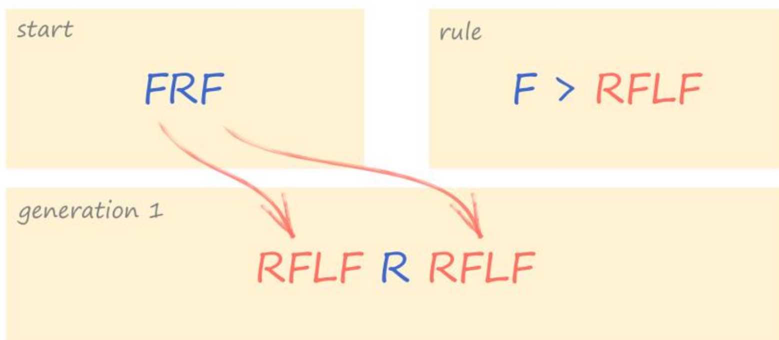


Code That Grows!

Let's do something really cool.

Imagine our code instructions can grow and change, as if they were plant cells.

This picture shows what happens to turtle code **FRF** if the growth rule is that every **F** turns into **RFLF**.



It will be interesting to see what patterns this next generation of turtle code draws.

We can easily write code to apply this rule to grow the starting **turtle code** into the next generation of turtle code.

Have a look at this code:

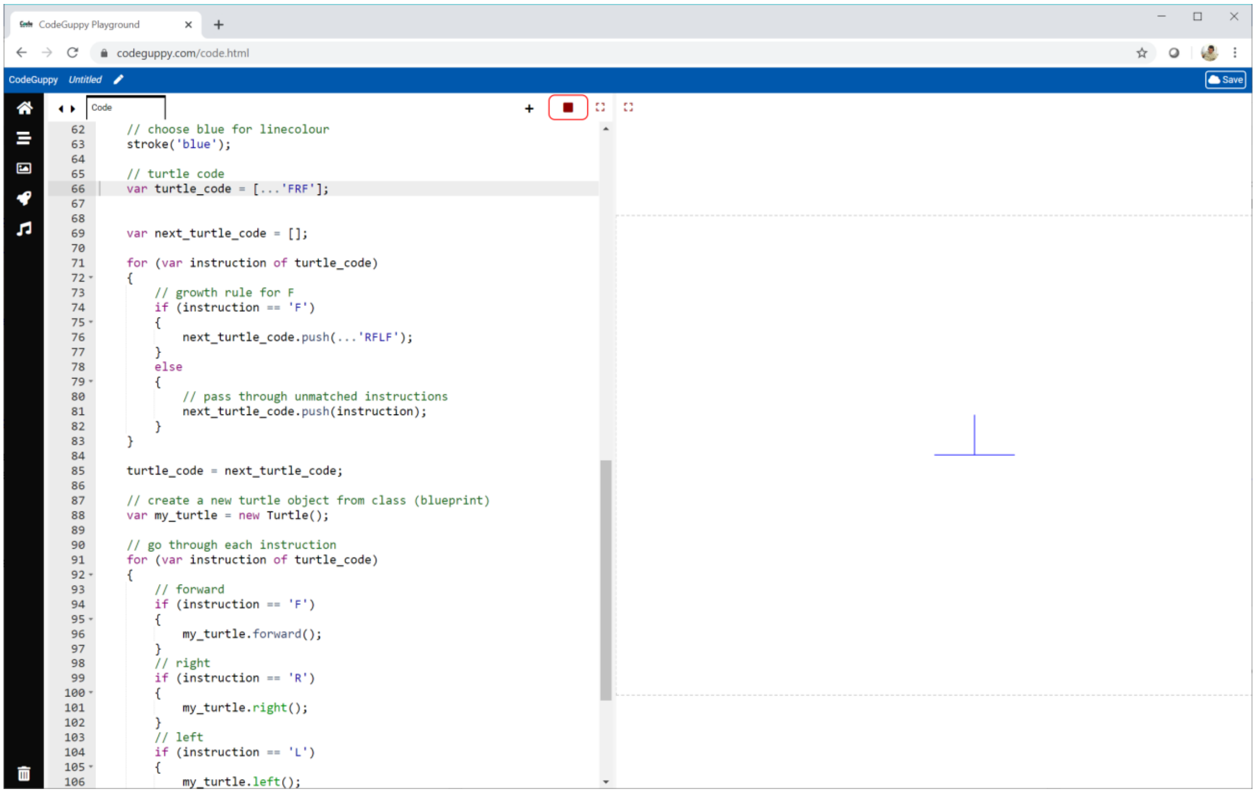
```
// grow next generation of turtle code
var next_turtle_code = [];

for (var instruction of turtle_code) {
     // growth rule for F
     if (instruction == 'F') {
          next_turtle_code.push(...'RFLF');
     } else {
          // pass through unmatched instructions
          next_turtle_code.push(instruction);
     }
}

turtle_code = next_turtle_code;
```

We start with an empty list **next_turtle_code**. We then use a **for loop** to work through each **instruction** in the **turtle_code**.

We copy each instruction to **next_turtle_code** except if it is an **F**, in which case we push **RFLF**.

After all the instructions have been looked at, **turtle_code** is pointed at the new code.

If we start with **FRF**, let's see what the next generation of code does.



Let's change our code to apply the growth rule again to grow the second generation from the first generation.

In fact, we can easily use another loop around the code we just wrote to grow many generations.

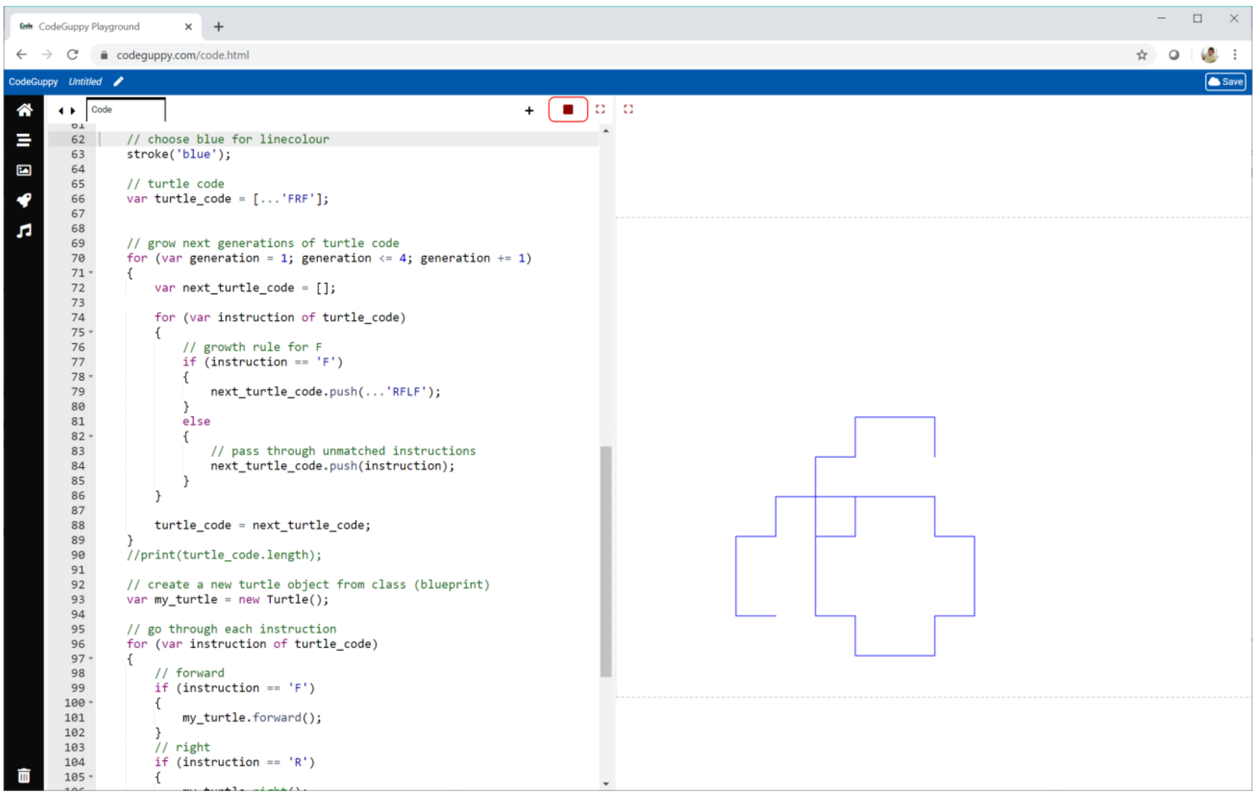
This loop grows the code for four generations.

```
for (var gen = 1; gen <= 4; gen += 1) {
    // ...
}
```

If you looked at the turtle code after four generations is would be **RRRRFLFLRFLFLRRFLFLRFLFLRRRFLFLRFLFLRRFLFLRFLFRRRRR FLFLRFLFLRRFLFLRFLFLRRRFLFLRFLFLRRFLFLRFLF**.

The **3** instructions **FRF** grow into **93** instructions!

Let’s see what pattern those **93** instructions draw.



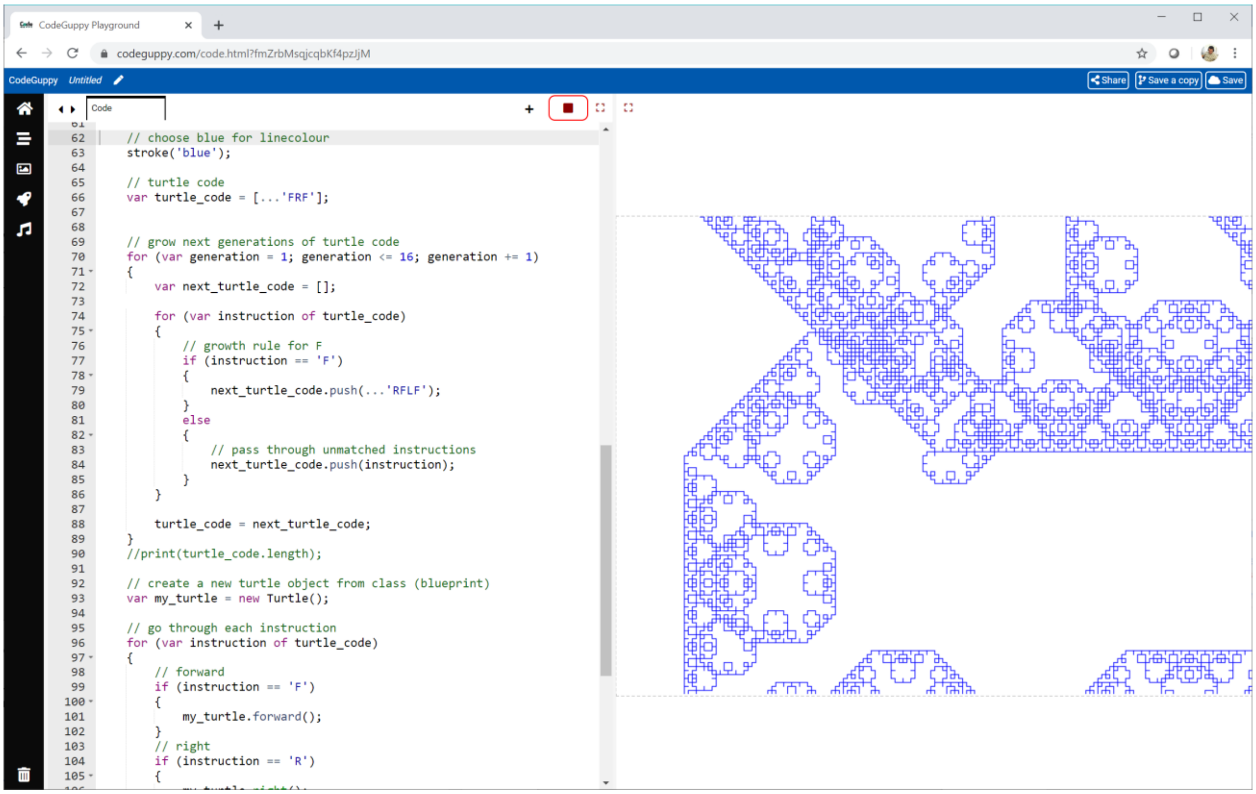
Very interesting!

Our turtle code has grown, and the patterns have become more interesting.

Let’s keep going.

If we grow the code for **16** generations, the turtle code grows to **39,3213** instructions. That's huge!

Here's what it draws.



That is pretty amazing!

Even reducing the turtle's step size to **5** the pattern is still much bigger than the canvas.

The code we've written so far is online at:

- https://codeguppy.com/code.html?tariq/ex25

Try It Yourself

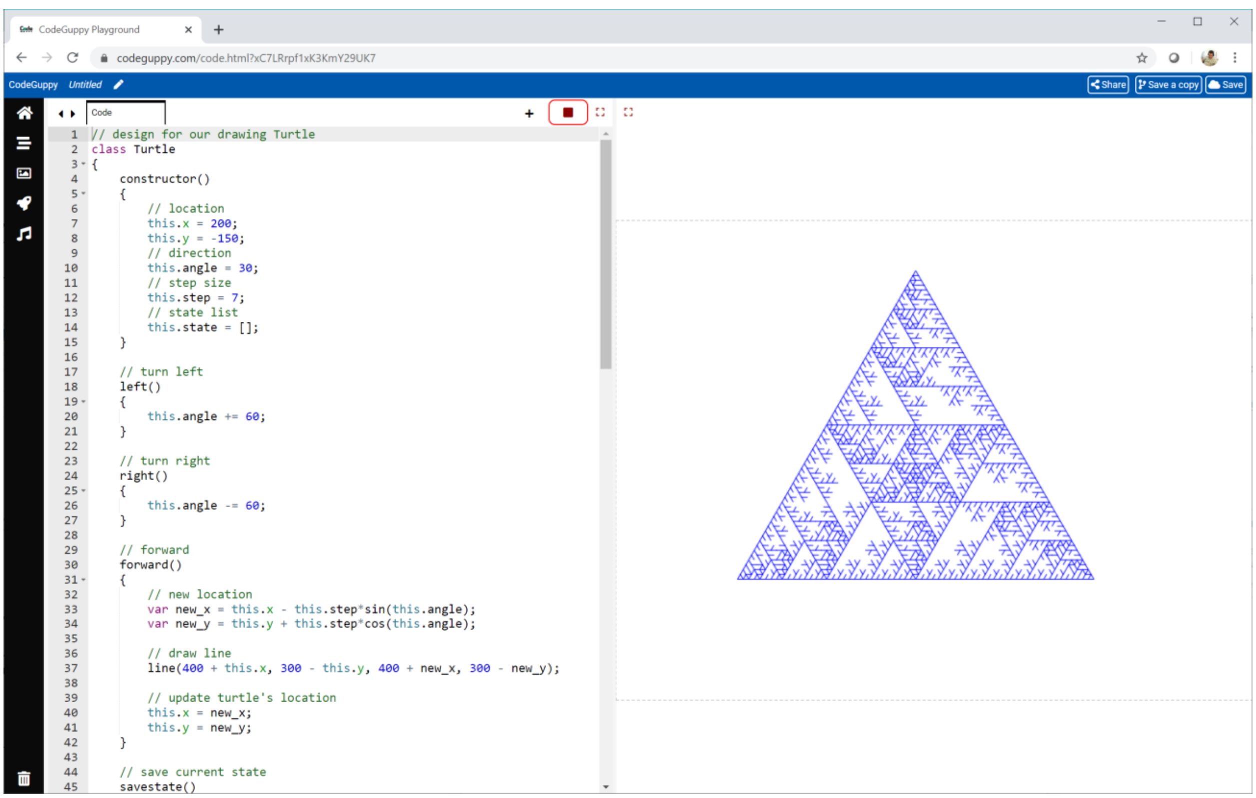
Use your own starting **turtle code** to see what patterns are drawn when it grows for **4** generations.

Try growing your code for more generations. Do this slowly because your code will grow very big very quickly and this can slow or crash your program.

If your pattern grows outside the canvas, reduce the turtle's step size.

One more thing to try is to change the amount of turn that an **L** or **R** instruction causes.

Here's my own experiment.



That triangular pattern was made with starting turtle code **FLLFLLF** and a growth rule of **F > F[LF]F**.

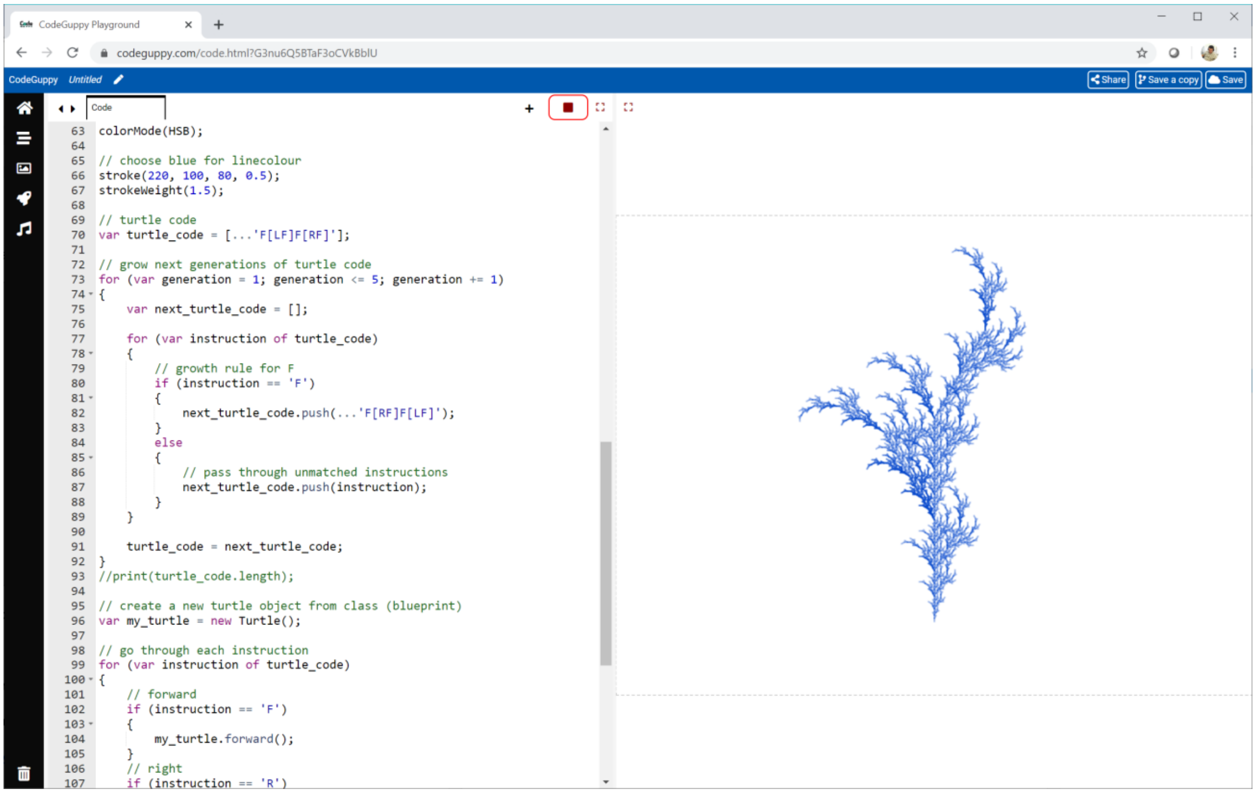
The turning angle is **60 degrees**, instead of **90 degrees**.

The step size is **7** and the starting location and angle were adjusted to make the triangle fit better on the canvas**.**

The code for this pattern is online:

- https://codeguppy.com/code.html?tariq/ex26

Because this is so fun, here’s another example.



The starting code is **F[LF]F[RF]** and the growth rule is **F > F[RF]F[LF]**.

The turning angle is **30 degrees**, instead of **90 degrees**.

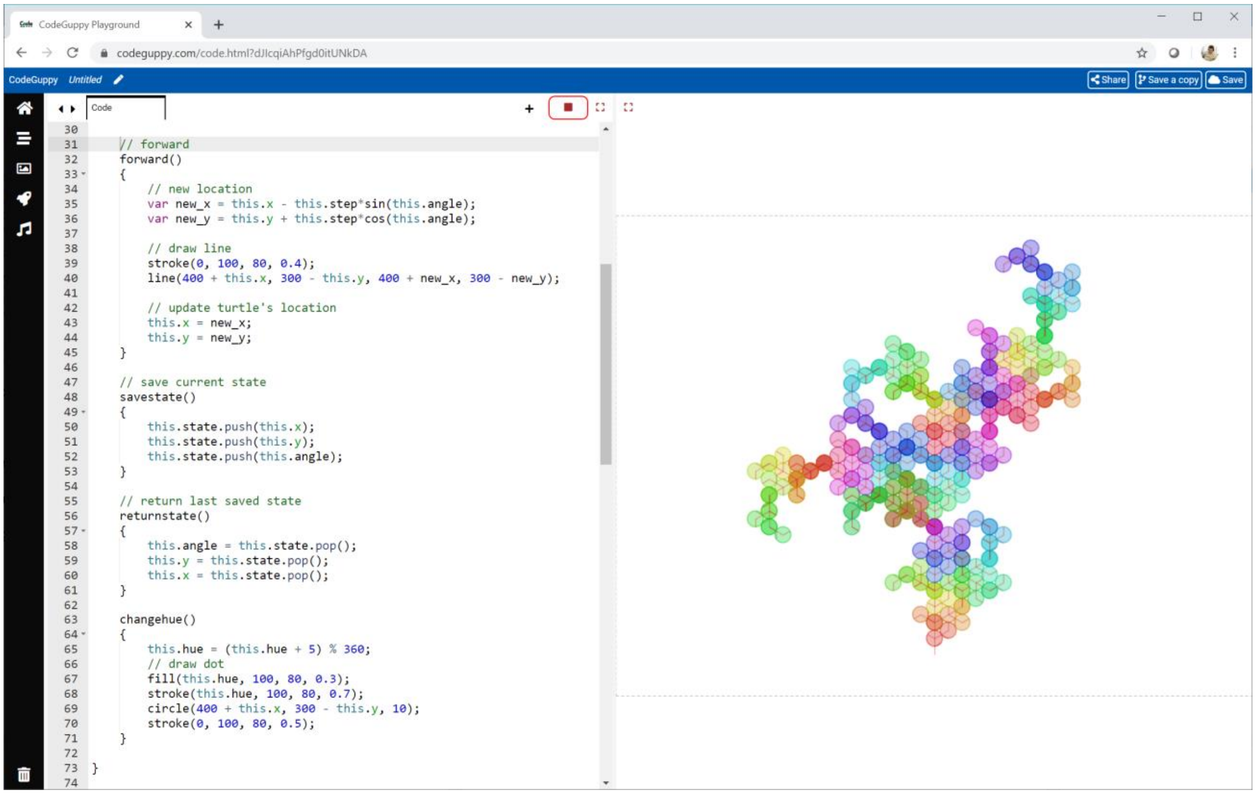
I’ve used a translucent line so that more detail is visible.

The code is online:

- https://codeguppy.com/code.html?tariq/ex27

Challenge!

Have a look at this pattern.



The challenge is to give your turtle the ability to understand a new turtle code instruction **C** which changes the hue that the turtle is keeping track of, and draws a small circle.

The code for this pattern if you need to look at it is online:

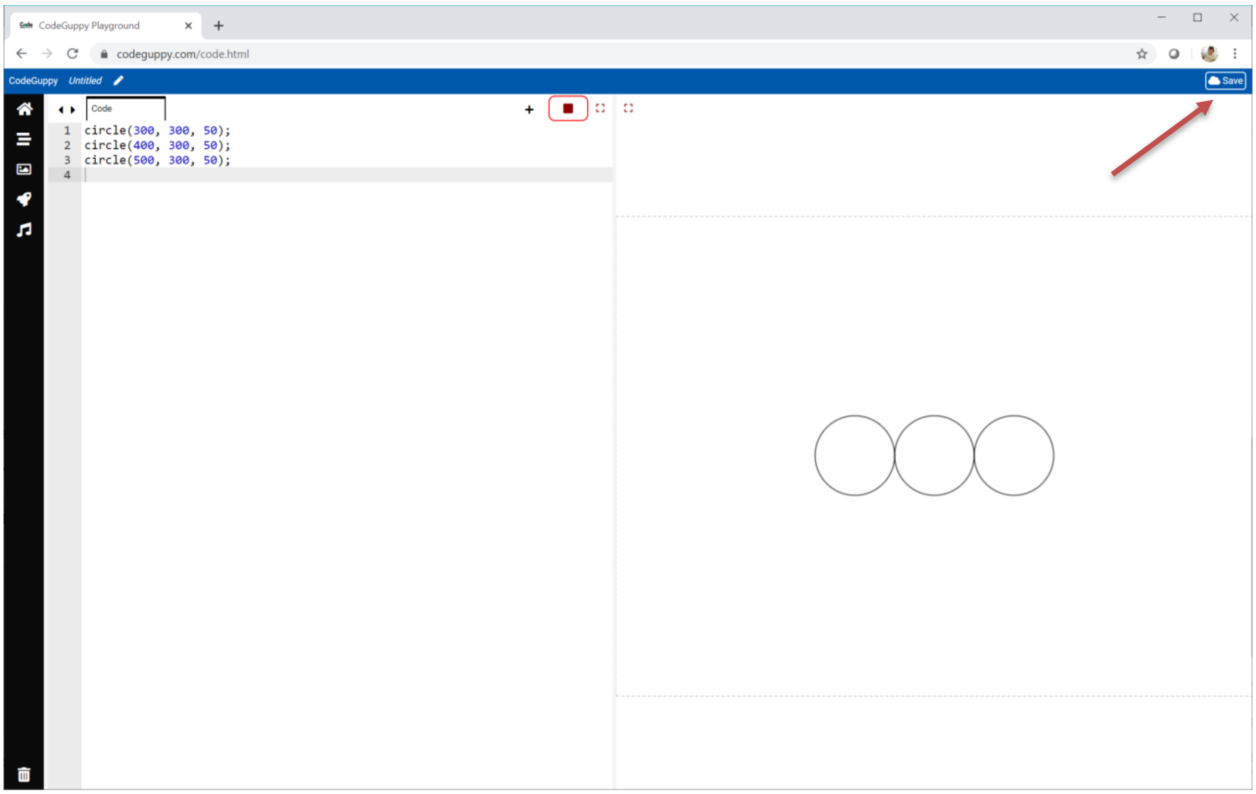
- https://codeguppy.com/code.html?tariq/ex28

Saving & Sharing Your Work

Saving Your Work

You can save your work so you can see it again another time.

To save your work, click the **Save** button at the top right.

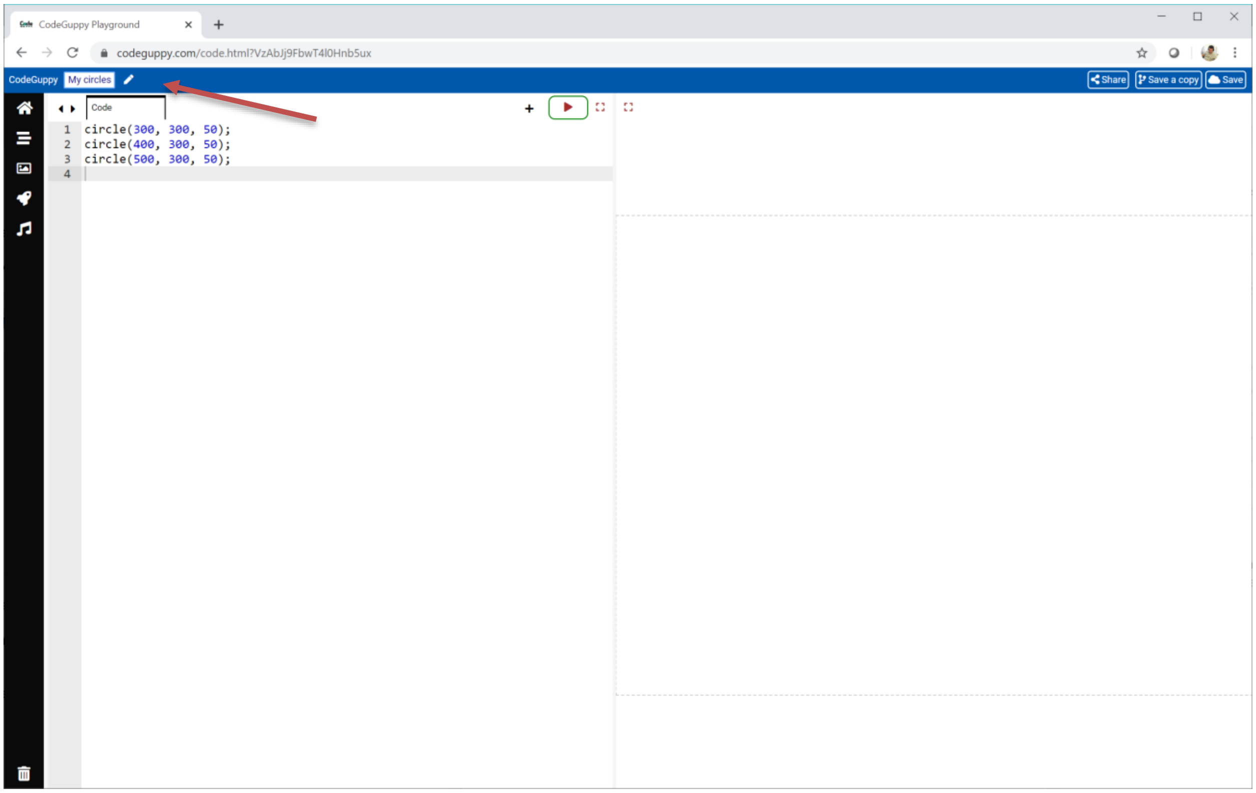


You can save your work even if you haven't finished your code.

This means you can take breaks from working on your code.

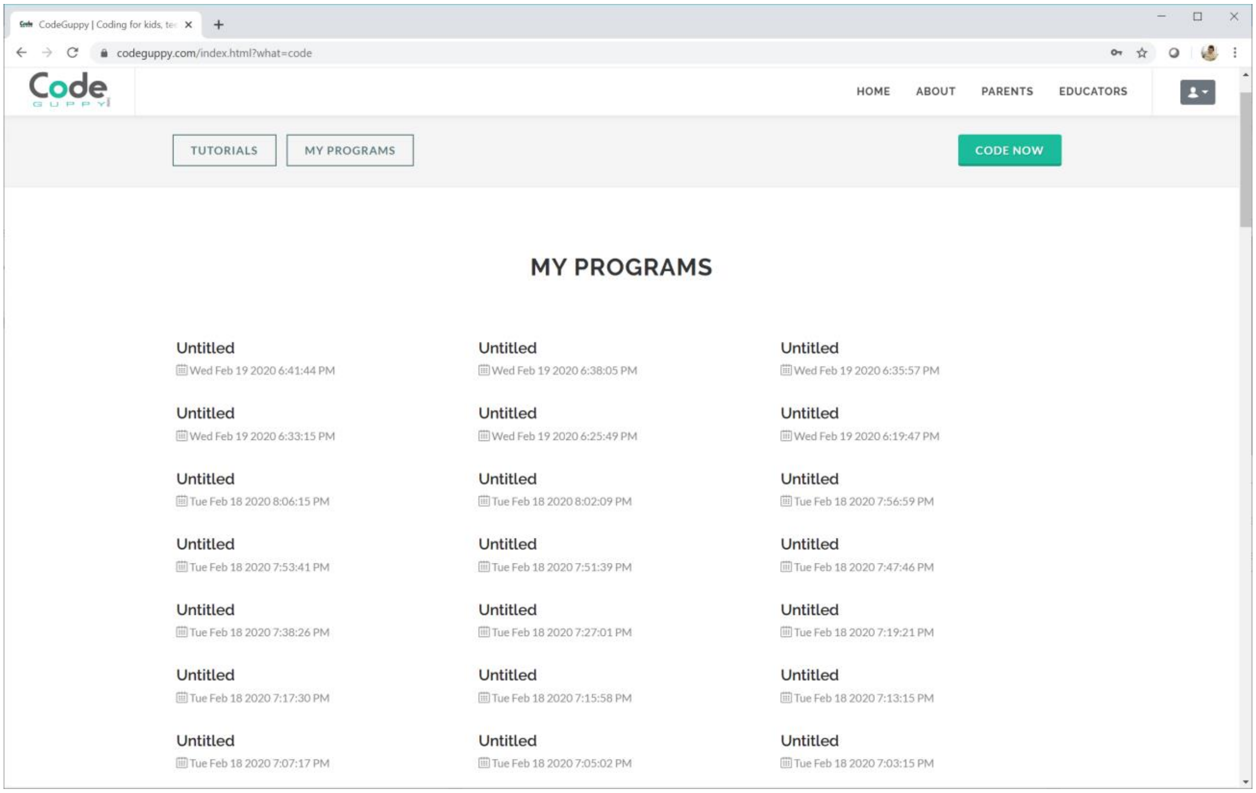
It's also a good idea to save your work regularly so you don't lose your work if there's a problem with your computer.

It is also a good idea to give a meaningful name to your program by editing the name presented in the top left corner.



When you're done with the program, press the "Home" button on the left toolbar to return to the main account screen.

All saved programs will appear under “MY PROGRAMS” tab on your account.

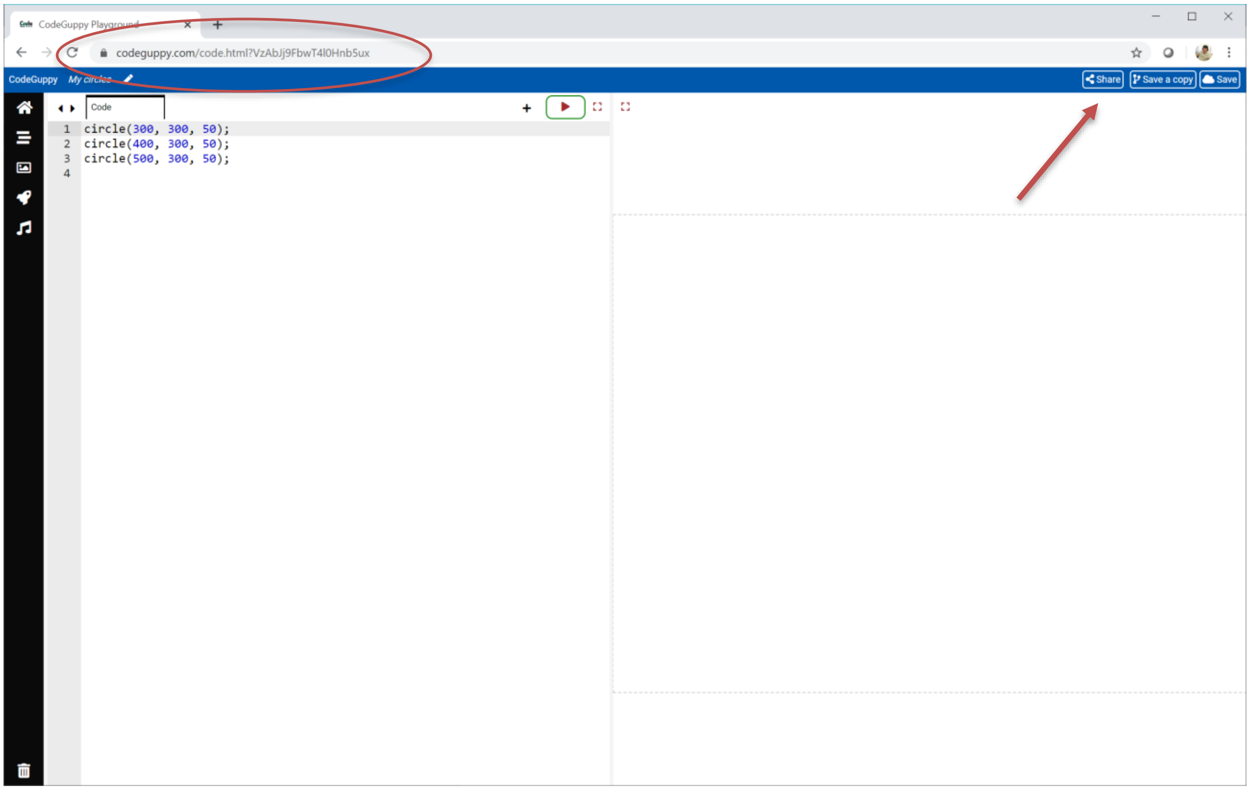


The programs will have the name you gave them while in the coding editor. If you didn’t rename your programs they will appear as “Untitled”. Don’t worry, you can open them at any time and rename them.

Sharing Your Work

Once you've saved your work it is given its own web address.

You can see the web address at the top in the browser's address bar. It should have a weird looking string at the end.



You can share this link in any way you want. You can email it to your friends or family. You can share it with the whole world on social media!

For more share control you can use the Share button from top right corner.

After someone clicks the link they'll see the drawing your code makes. They can also see your code if they want to see how it works.

Anyone on the internet with a tablet, laptop or computer can see your work.