The Joy of Coding Algorithmic Art Workbook #1 Your First Shapes

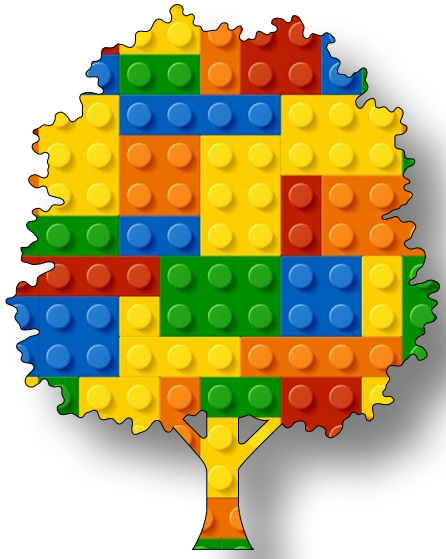


Table of Contents

MIT Licence

Brief summary:

Permissions (what you can do)

Commercial use
Modification
Distribution
Private use

Limitations (what is not covered)

Liability
Warranty

Introduction to the Curriculum/Tutorial

The Algorithmic Art Workbook series is designed to get you coding from the very start. It will take you from zero to competent coder over the coming pages. It will expect no previous experience. It will expect no high levels of maths. It will, however, test your problem-solving skills. It may well frustrate you at times, but you don't learn anything unless you try. This is for all ages, not just for the young.

Above all, you will learn how to code, one small step at a time. My recommendation is to do a bit each day. Enjoy the process. It may feel like a bit of a sharp learning curve at first, especially if you have never coded before. You may feel a bit overwhelmed at first; nevertheless, you will find that the amount of syntax (vocabulary) needed is small, the structure (lines of code) is simple and can be applied repetitively.

You will only learn by trying, and by trying you will learn best when things go wrong. If you make a mistake or it goes horribly wrong, you will learn something (probably patience). Try something new, see what happens, explore, be creative. All you need is a computer (and internet) and a desire to learn. It may well take you out of your comfort zone, but that will be a good thing.

Whatever your motivation for this, I am pretty certain you won't be disappointed. Take the time to play, experiment, and create. See what happens (or in some cases doesn't happen). All this time your brain is being rewired which can only be a good thing. Similar to learning a new language, a musical instrument or learning to draw or paint.

I will be putting very short videos together on my website www.ElegantAI.org that will hopefully help you in this process. There will be other areas of coding that you may be interested in once you have mastered the basics, such as Machine Learning, making Games or Robotics.

These tutorials are spread over several modules. Each module is further split into units. The tutorials take you step by step; it is up to you to decide how fast or slowly you take it.

Creative Coding or Algorithmic Art is the process of creating art using code and some very basic maths. You write lines of code; when these are combined together, they are like a set of instructions called algorithms. A complete programme is called a sketch. Think of this as a recipe where you will have a bunch of ingredients (variables), and a set of instructions (algorithm); then finally, you cook or bake and you have your final product (sketch). Get something in the wrong order, and it may not quite work out as you hoped.

To achieve this, we need a simple, intuitive, user-friendly, and creative coding language. This is where p5.js comes in. It was initially designed for creative people to turn their code into art, although it can be used for machine learning, games, and even robotics.

This course has two primary functions: one, to teach the basics of coding, and two, to give people beginner skills to express their creativity. You can do both with p5.js as you follow this course.

The coding language in question is called p5.js. It is a JavaScript library that has a canvas onto which you create your art. It is what is called open source and completely free. You don't even have to download anything; it is all done in the browser.

This also means you can use any machine that has a browser. A computer, laptop, tablet, Chromebook, even your smartphone or Raspberry Pi. The first unit will help you get started, taking you through the editor you will be using. It is a quick read looking at its main features.

Using p5.js, you can do more than create static images; you can also create dynamic, moving, interactive art. As well as 2D shapes, you can explore 3D shapes as well. You can import images, videos, and music. This is a very flexible and powerful coding language. It is easy to learn, but it becomes as challenging as you want it to be.

To help you, I will include some resources on my website www.ElegantAI.org, which, as the name implies, focusses on Algorithmic Intelligence (Machine Learning) with p5.js.

There is an important update!

There is a new version of p5.js. The latest available version (at the time of writing) is 2.2.2; however, the default was still 1.11.13. This will change at some point in 2026.

Screen Arrangement

There are a number of ways you can use this workbook:

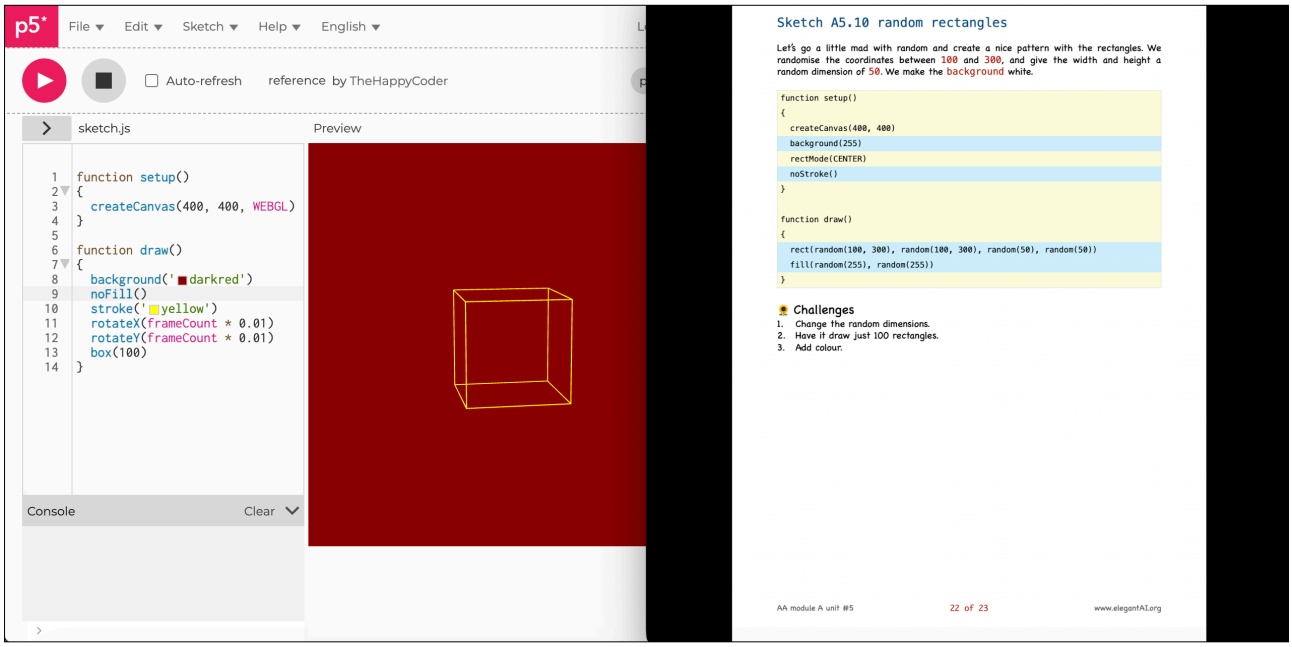
Suggestion #1: Have it on another device, for instance, a tablet (or even a smartphone), while you work on it on your computer.

Suggestion #2: Split your screen so that you have the editor and the workbook on the same screen using the Kindle app. This depends on the size of your monitor.

Suggestion #3: Download it as a PDF and print it out; this should be possible as I ticked that box to allow it.

Suggestion #4: Using a Kindle eReader (on a Kindle device), this is a little more problematic as greyscale means you miss many of the features; the tutorial relies heavily on colour, but you can also download the PDF for reference.

Suggested arrangement



Videos

To help with the process, I will include short (less than five minutes) videos on my website that may also help with how to get started. They are very simple.

Resources

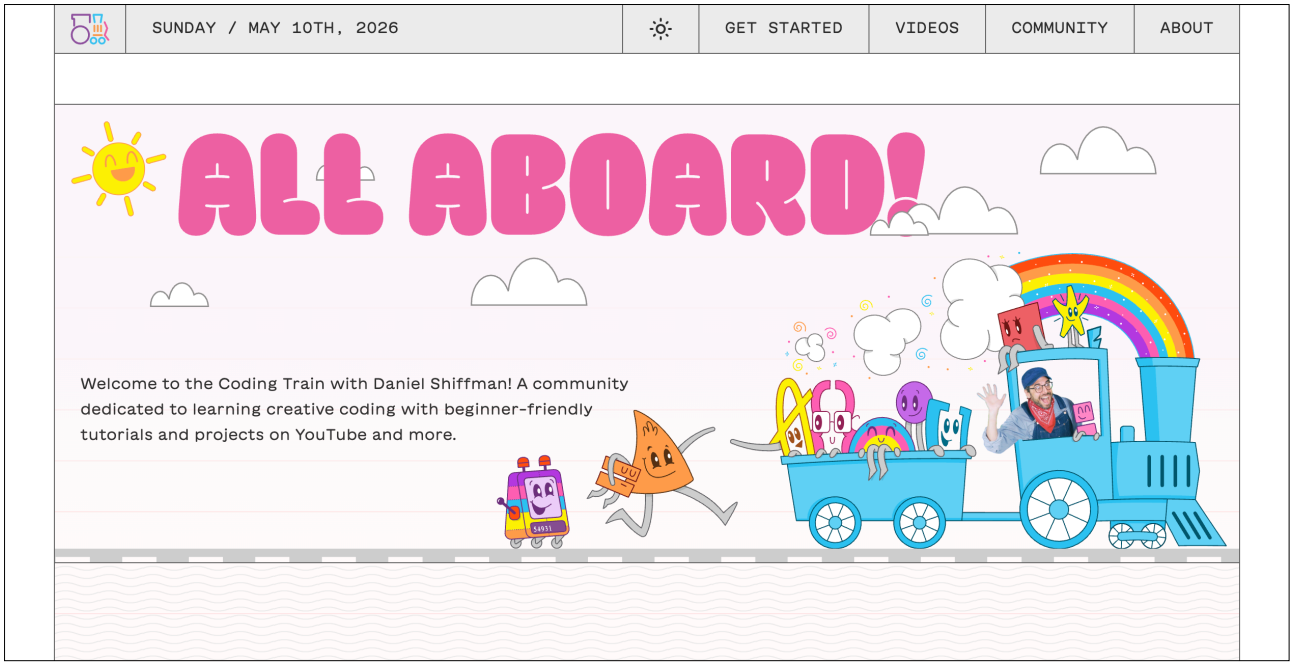
There are a number of additional resources that you may find very useful. The main ones I would encourage you to look for are based around The Coding Train, which is the go-to place for all things p5.js, as well as the Processing Foundation website.

The Coding Train is both a website and a YouTube channel run by Dan Shiffman, who has a huge amount of material available.

Suggestion #1:

Website: https://thecodingtrain.com.

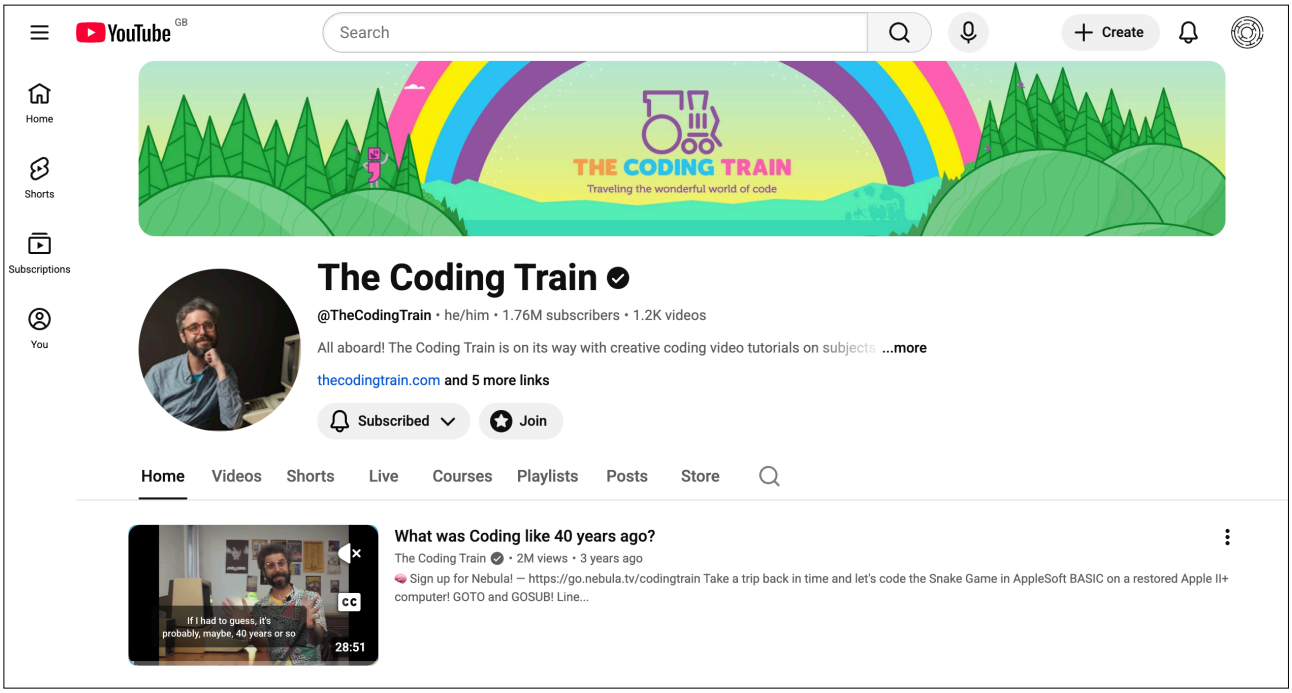
The Coding Train Website



Suggestion #2:

YouTube: https://www.youtube.com/thecodingtrain
His videos are entertaining as well as very informative. There is a wealth of ideas, challenges, and explanations. Highly recommended.

The Coding Train YouTube Channel

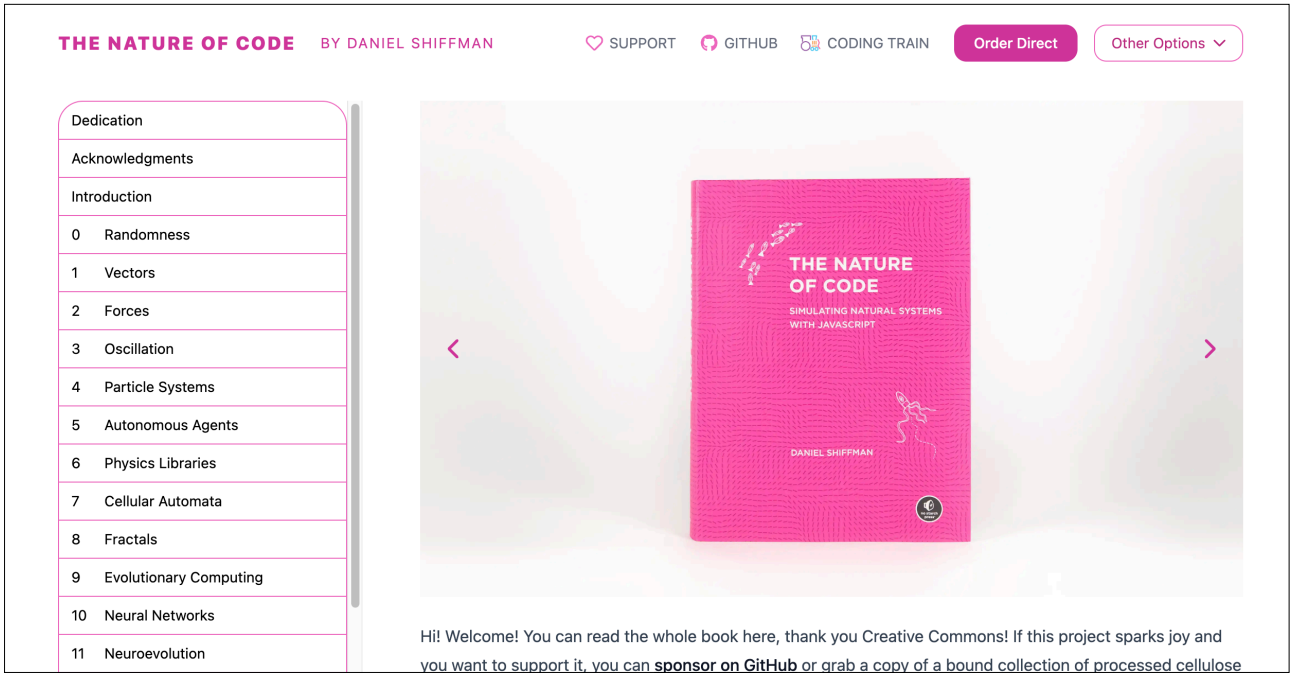


Suggestion #3:

Dan Shiffman's book: The Nature of Code

You can buy this book or use it online at https://natureofcode.com
It is all based on p5.js and looks at how you can simulate nature.

The Nature of Code



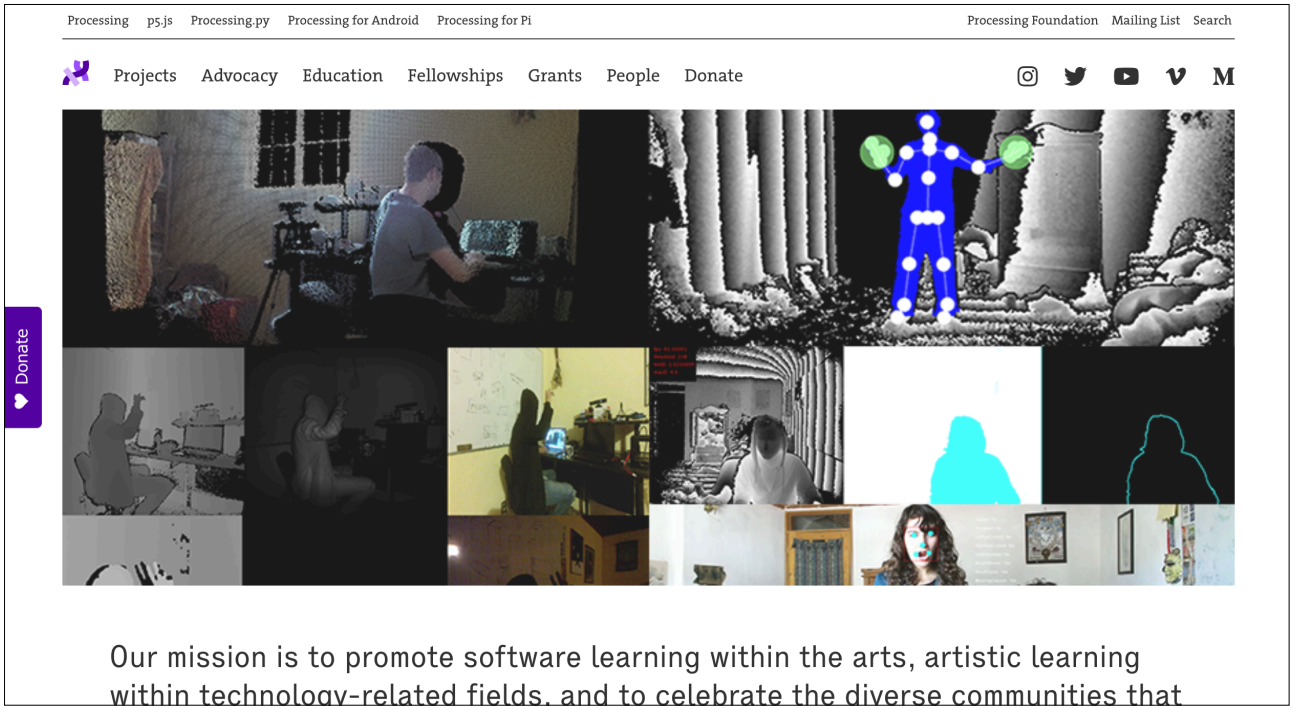
Suggestion #4:

Processing Foundation website at:

https://processingfoundation.org

This is where p5.js originated and is the parent of it, as you might say. It has other resources (the tiny top tab) and is again a fascinating well of information.

The Processing Foundation



Workbook #1 Quick Content Summary

Each unit will explore a specific aspect of coding. The main areas we are going to visit are the range of 2D shapes you can use. How to draw them and colour them. Then we move onto the use of loops. These are integral to coding as they will do a lot of the repetitive coding you would otherwise have to handwrite. One of the unspoken objectives for coders is to write with the fewest lines of code as possible.

You will be facing a lot of new concepts. Please don't worry about whether you understand it straight off. You won't. But as you progress, you will use them time and time again until they become intuitive. The more you practice just typing things in, the more memory muscle you build up. Practice makes perfect.

So, persevere and be intentional. Have a go at the challenges, but they are not obligatory, just to get you thinking.

The code is in the yellow boxes; any new code is highlighted in blue, so you don't have to type out the whole thing again each time. Set up an account (highly recommended) so you can save your work as you go along rather than starting again each time. It will also remind you where you were up to.

Most browsers work, but Chrome is best as there is some functionality missing in some browsers. I suspect it has been tested on Chrome to guarantee it works in the Chrome browser.

The Joy of Coding

Algorithmic Art

Module A

Unit #1

the p5.js web editor

Module A Unit #1: p5.js web editor

The web editor is where you type your code. The code is a set of instructions which you then execute (run). All being well, it should do exactly as you expect. Unfortunately, that may not happen because it will do exactly as you tell it, and this is where the fun comes in. This is a problem-solving activity. It cannot read your mind nor understand your intentions.

The code that you type into the web editor needs to be understood by the computer. It is quite forgiving, but a simple omission such as a comma or a bracket can render the whole code broken. Then you have to find where it all went wrong; usually, it is something very, very simple, but sometimes you may need to rethink your whole approach. It is extraordinarily logical and pedantic at times.

The web editor interprets the code that you type into it. It has been created by someone who wants to make it as human-friendly as possible before translating it into 0s and 1s for the computer to understand. In this first unit, we are going to have a good look at the web editor before we use it.

As the same suggests, the web editor is located on the web; you access it with your browser (preferably Chrome). Other editors, you would need to download an IDE (that is what our web editor is) before you can do any coding. This system is not without its drawbacks (need the internet), but it is then accessible on any machine anywhere you go.

Question 1 – What machine do I need?

This is probably the first question you might ask. Which machine should you use or not use for this tutorial? The simple answer is anything with a web browser will work for nearly everything. I do all mine on an iPad, but you can do it on a Chromebook, PC, Mac, laptop, smartphone, tablet, or a Raspberry Pi.

You do not need a state of the art computer or a high end one. Almost anything will do as the code runs in the browser not your machine.

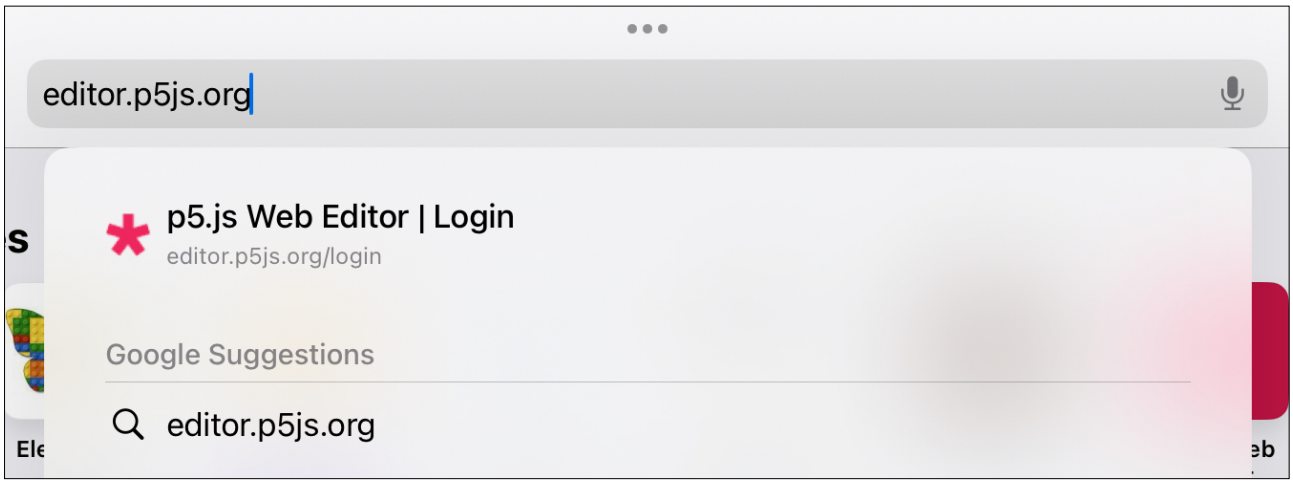
Question 2 – What is the editor?

You don't need to download any software; you simply type:

`editor.p5js.org`

into your web browser, see Fig. 1, and off you go. Most web browsers support it, but if unsure, then use Chrome or Safari. If you have a favourite one, then use that and see how you get on.

Figure 2a: type `editor.p5js.org`



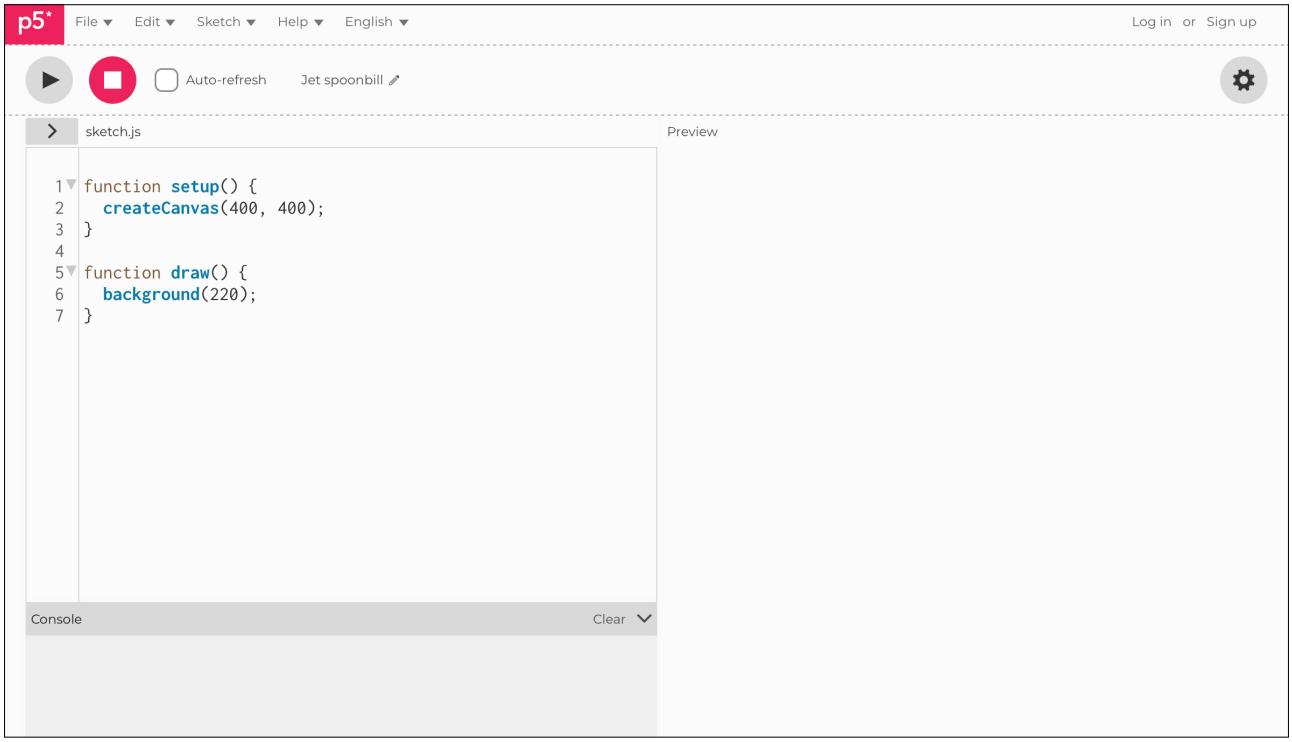
When you first start, you get a page like this shown in Fig. 2. You have a `main menu` across the top left-hand side. On the right-hand side, you have `Sign up` and `Log in` tabs. Below that, you have a number of buttons, a checkbox, and a text box.

Below that, you have two main panels. In the first panel (left-hand side), there is a column of numbers which are the line numbers for your code. The second panel is where you type your code. The third panel is the canvas, where stuff happens; the canvas will appear when you start to run your code.

If you press the run button (dark grey triangle in a light grey circle), the canvas should appear as a grey square where it says Preview. Underneath the panel for your code is another one called `console`; this is where your error messages and other information you request (in your code) will be sent and seen.

I will go through all these in a little bit more detail shortly, but for now, click on everything and anything to get a feel for the buttons and tabs. Everything is pretty intuitive, and the best thing to do is learn through playing with the buttons and seeing what happens.

Figure 2b: the editor



Question 3 – What does the update mean?

Even as I write this tutorial, there have been some updates. This is true of coding generally; they are always improving it as demand and use change. This is also true for p5.js and the web editor. Every so often, they have a new version, which generally goes unnoticed, but now you will be able to see which version you are using. They now include a tab for selecting earlier versions.

This is in preparation for version 2 coming out in 2026. You will still be able to access older versions then, but the default will move over to 2.x. We will be currently using version 1.11.5 (or higher), which should not be a problem for anything you will be covering.

For some of this tutorial, the images used were from before this new feature, so it may not be present in part of the tutorial. Fig. 3 shows the new web editor. When you get the chance, click on the version, and you will see a drop-down menu for you to explore at your leisure. You can even try the latest version 2, but until it becomes the default, I won't be using it in this tutorial.

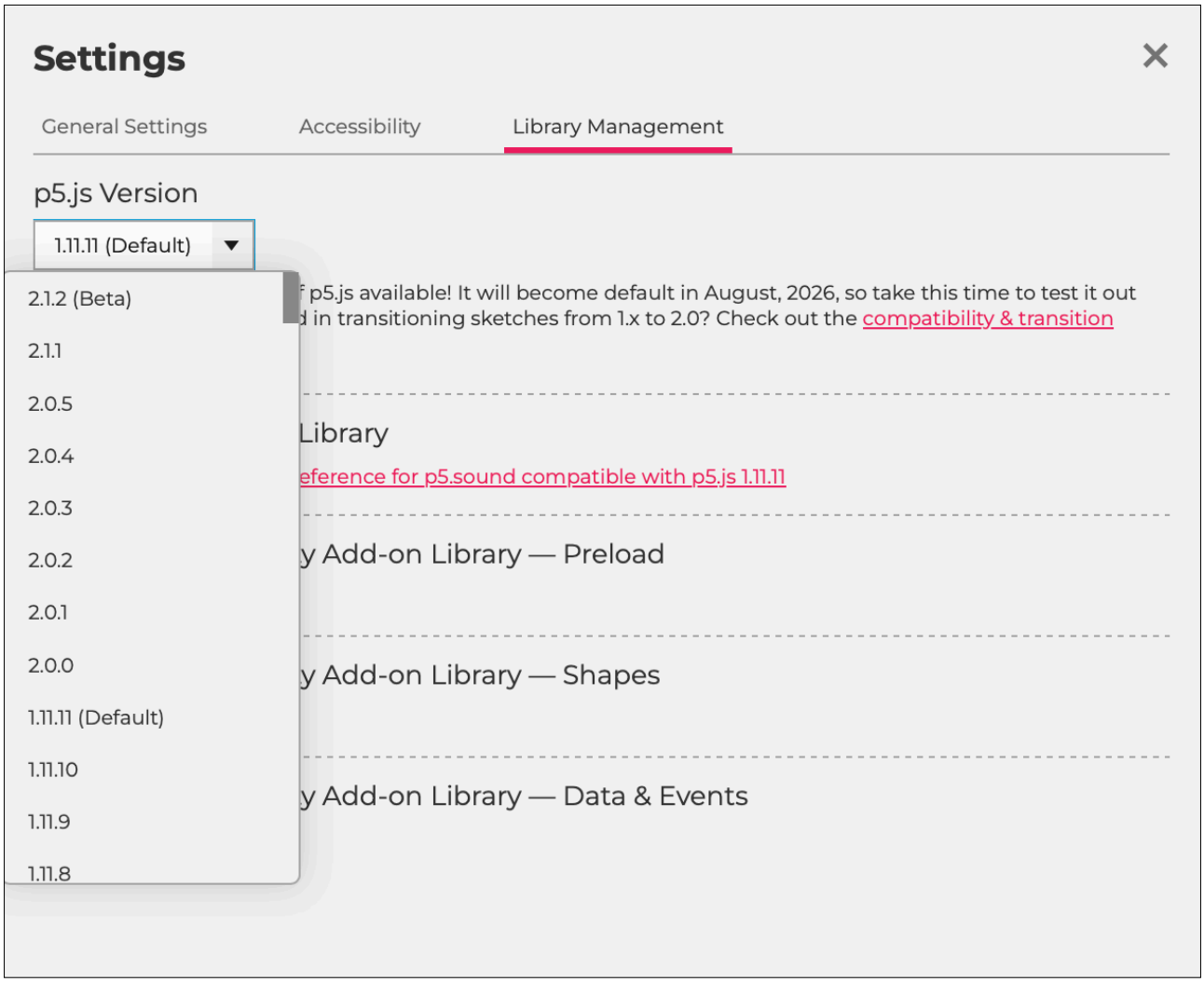
So, depending on when you read this and if I haven't edited since, in that case this comment won't be here, you will still be using version 1.11.13. The good news is that the modules that you are going to be working on are mostly compatible with both.

I have made everything compatible with version 2, to do that I have had to make very few changes if any. What I recommend, at this stage is to use version 2.2.2.

Figure 3a: the addition of the version

Figure 3b: click on the pink button

Figure 3c: select version 2.1.1

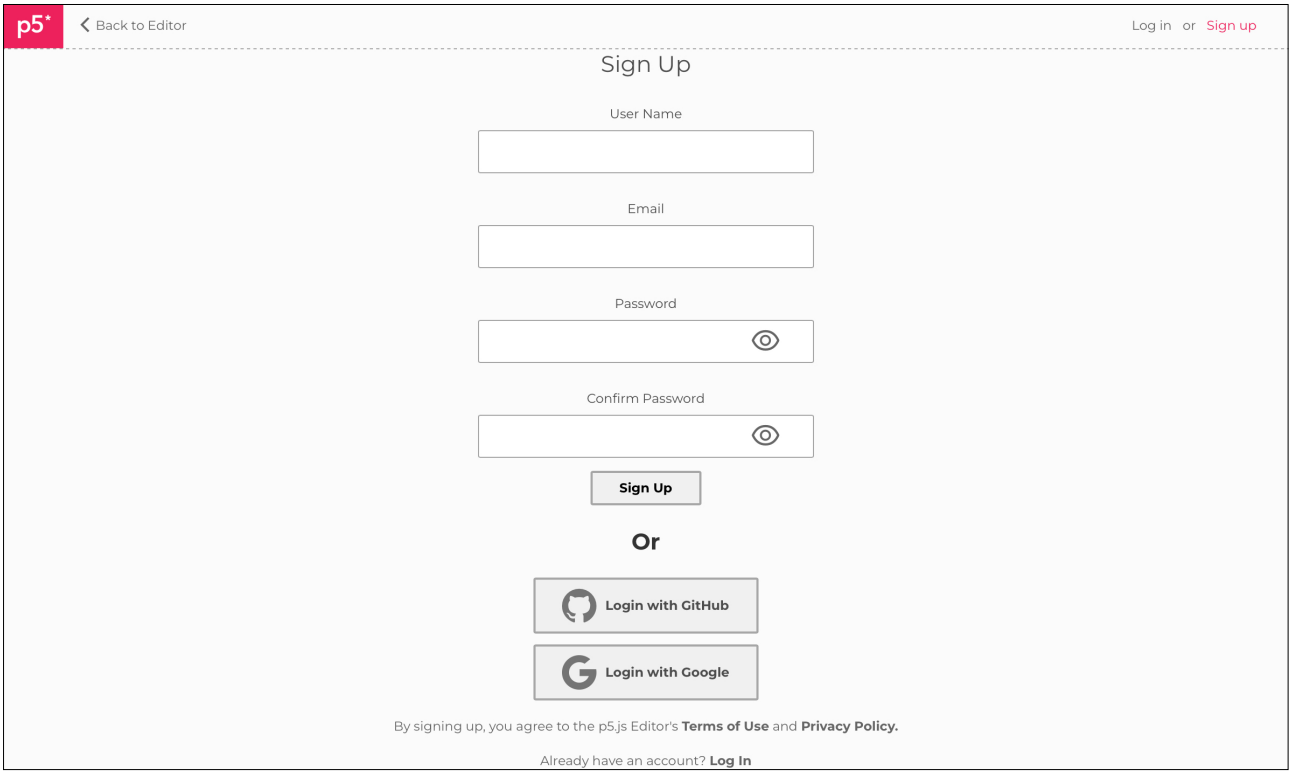


Question 4 – How do you sign in?

Although you don't have to sign up for anything and you can code straight away, you won't be able to save your code unless you do create an account. I would recommend setting one up right from the start. They have never, ever pestered me; it just used to log in.

Use one of the methods shown below, and you are good to go (again). You can log in with your Gmail or sign up with another email account. Test it all out first to make sure it works. I use `Login with Google` (recommended), but you can always use one of the other methods. Obviously, you will need a Gmail account for that. Sign up first and then log in (see Fig. 4).

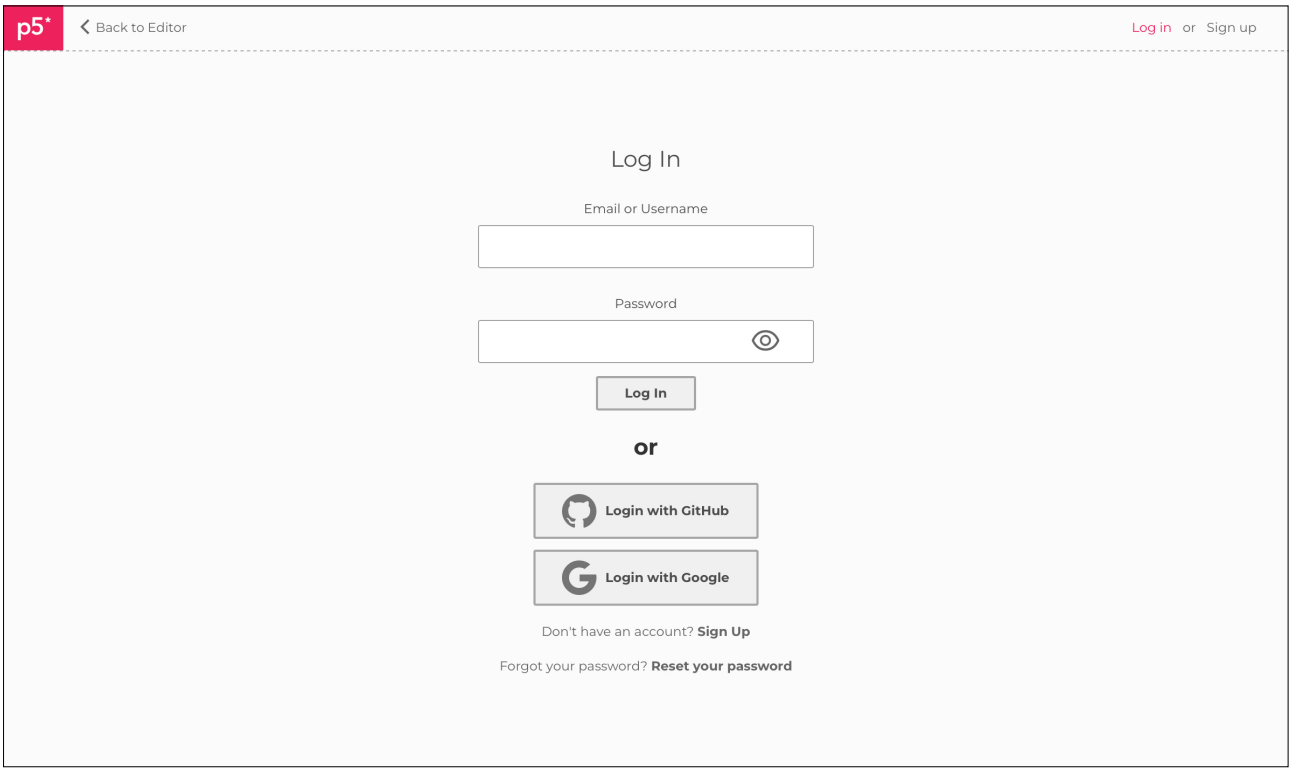
Figure 4: signing up



Question 5 - How do you log in?

If you have set up an account then you can simply log in.

Figure 5: logging in



Question 6 – What is the default sketch?

At the very start, you get a default sketch. This includes two functions and some curly brackets. These two functions are important and are built-in functions. The first one, `function setup()`, is to set up how you are going to see and use the sketch; this runs through only once and usually includes the size of the canvas.

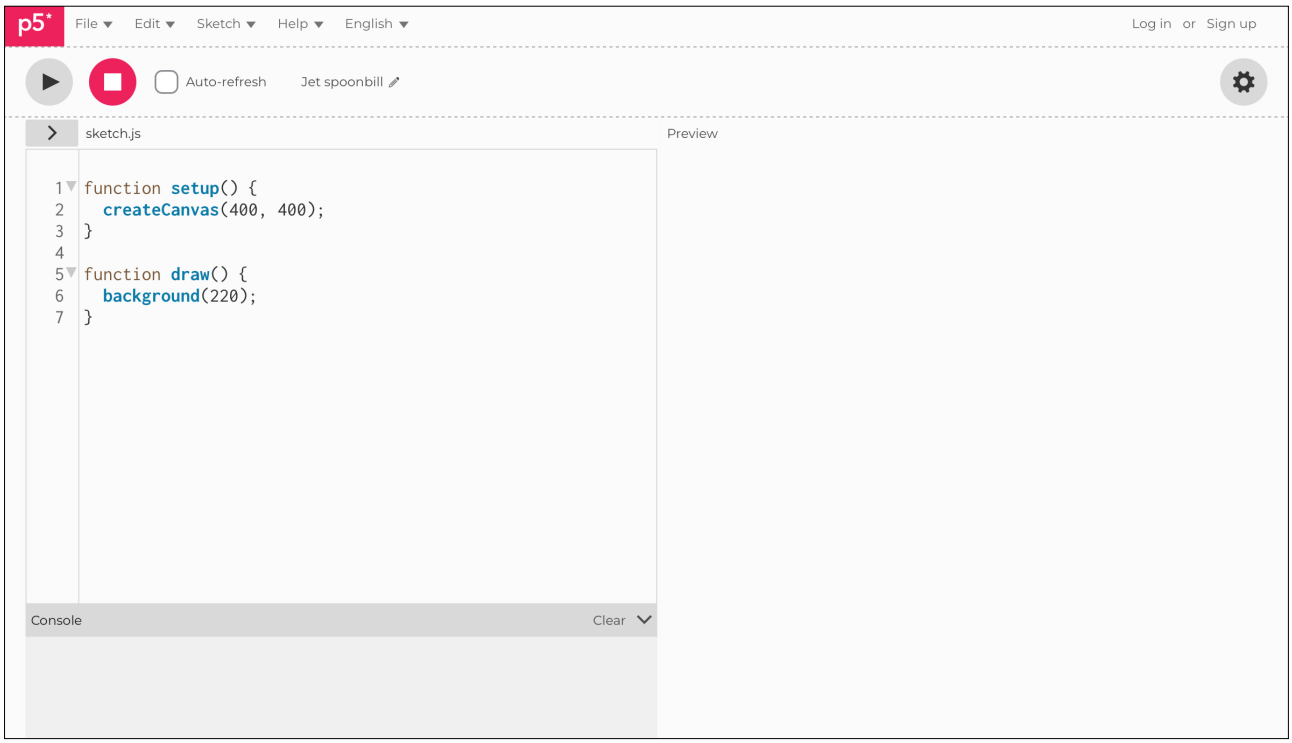
The second one, `function draw()`, is a loop; it draws on the canvas continually. This is where much of your code will go, although in some cases, you won't even have a `draw()` function, but more on that much later.

These two functions are prebuilt and are what are called predefined, and so you cannot use those named words/functions anywhere else.

Inside the curly brackets is where your code sits. At the moment, you have the canvas size of `400` pixels by `400` pixels in the `setup()` function. In the `draw()` function, we have a grey `background()` with a value of `220`. This is the grey value from `0` (black) to `255` (white) and shades of grey in between. We will talk more about colours later.

You will also notice the semicolon (`;`) after some lines of code. Most people who code with p5.js include this to mark the end of a line of code. I have omitted it to keep the code cleaner and easier to read (see Fig. 7). This is not essential, but you may want to revert to it later. It has never been a problem in not using the semicolon, but it is up to you to decide whether to use it or not. This is just my preference, and I know I am in the minority here.

Figure 6: default sketch

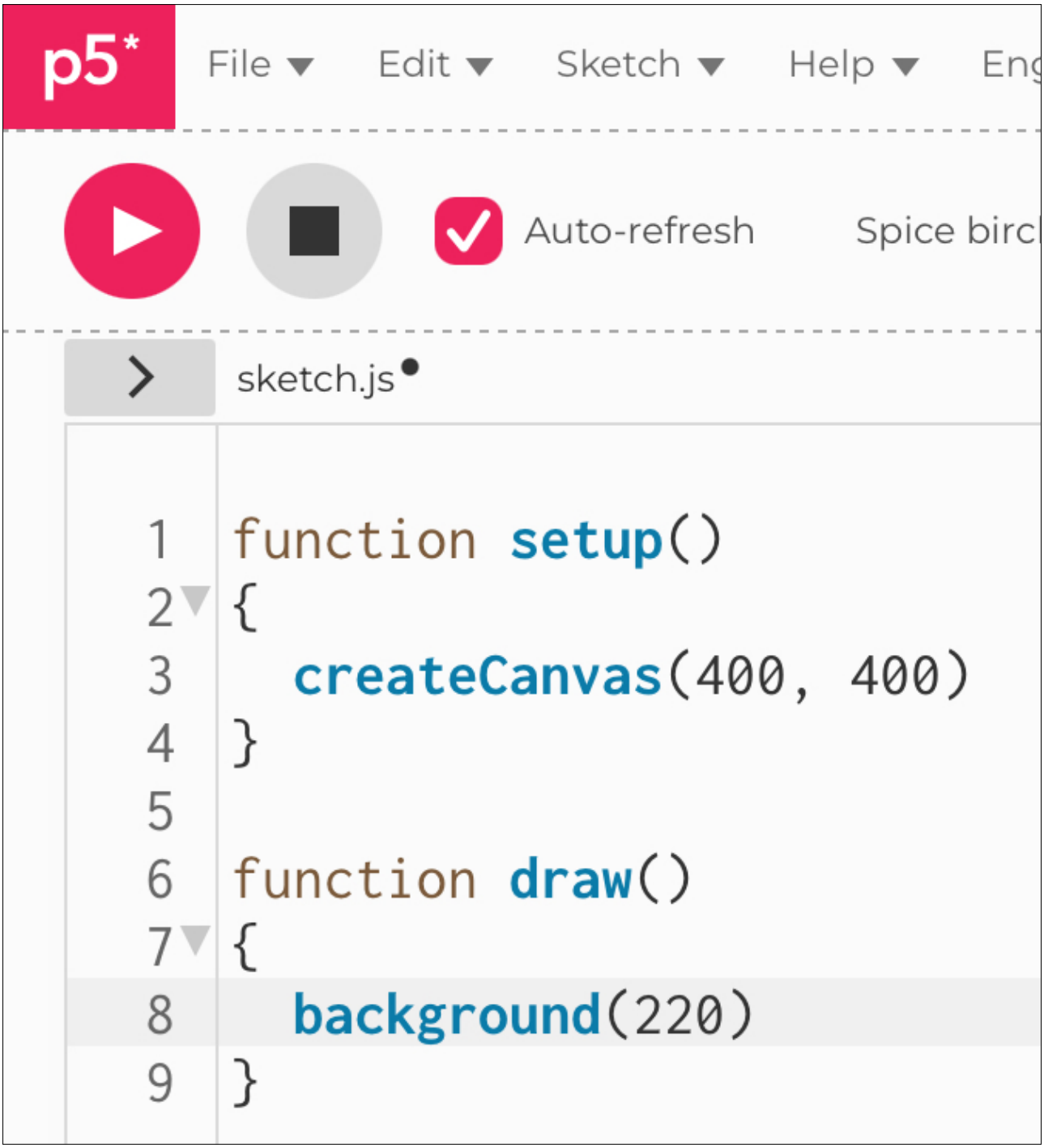


Question 7 – Why does your code look different?

The above is the default format, but the format I will present in my tutorial will be slightly different. You will see that the first curly bracket starts on the same line as the function name; it is how most people will use this. But I want to make the curly brackets { } stand out more, so I change the format to include the first curly bracket on the next (or first) line after the function name. Again, this is just my preference, to improve clarity. See Fig. 7.

All the lines of code are placed inside a set of curly brackets { }, there are always in pairs, like bookends. The code inside a set of curly brackets is usually indented with two spaces; this is the default setting and, although not essential, makes the code readable and stands out clearly.

Figure 7: my format

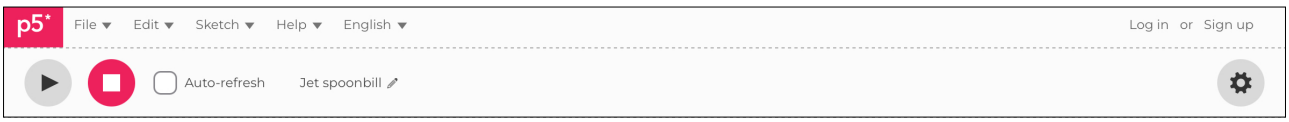


Question 8 – What do all the buttons do?

You will see a row of buttons across the top. It is worth spending the time getting to know them. The first thing to consider is how to save your work as you go along. If you are working on something one day and want to continue the next day, then you need to save your work. Also, you may find that your code may crash (for various reasons). This is why you need to save your work as you go along, not only at the end of a session. You can set it up to save your work automatically as you go along, but be aware there is no undo option.

It is worth exploring these buttons and tabs to familiarise yourself with what they look like and what they offer before you go very far into your coding. I will highlight the most important ones, but they all have their function and purpose.

Figure 8: main buttons



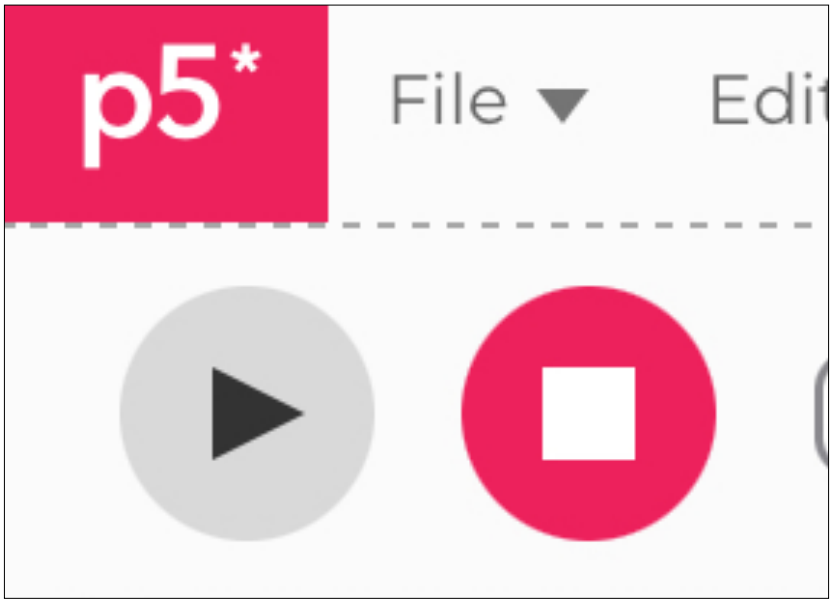
Question 9 – How do I run my sketch?

The first button you come across is the run button. It looks like the play button on a device you might have used. After you have written your code, you will want to run the code to see what happens. When you click on the run button, it will do just that.

To stop the code running, you will need to press the stop button, which is next to it, shown with a square in a circle, again pretty intuitive. The programme will keep on running the code until you stop it.

So every time you write some new code, remember to press the run button each time. You can also set it up to run automatically every time you add new code, but I wouldn't recommend it until you really know what you are doing. This is because it can cause problems as it tries to run incomplete code, at which point you will lose any changes since your last save.

Figure 9: run and stop



Question 10 – Can I change the name of my sketch?

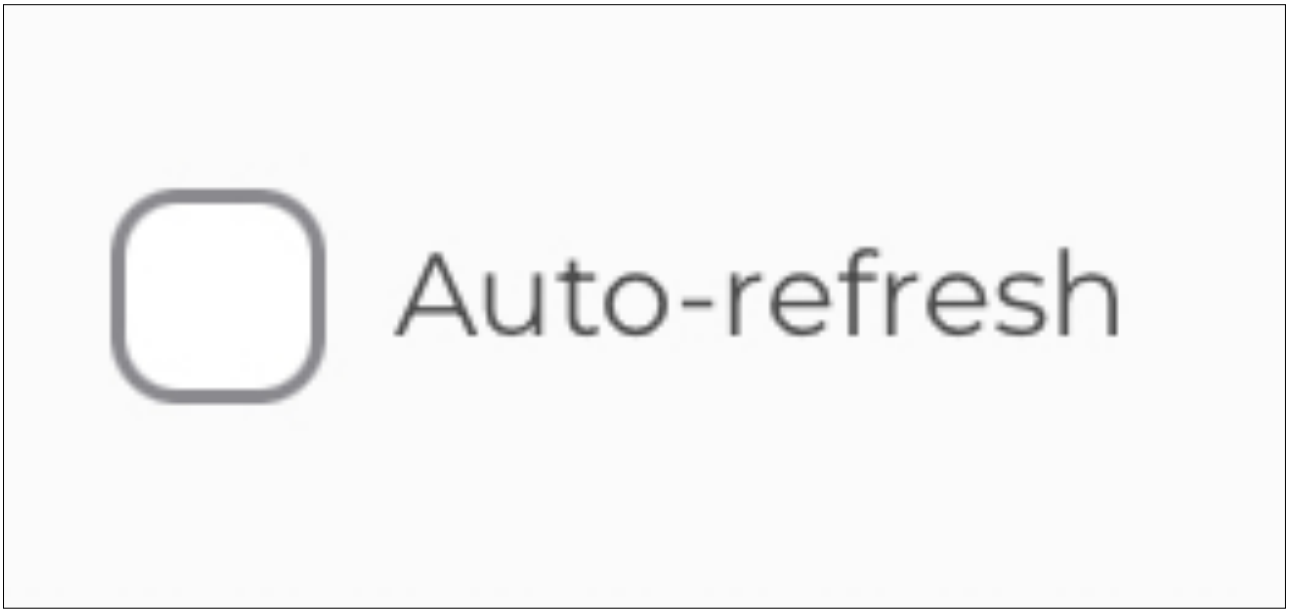
You will have seen a box with a couple of random words in it. This is a randomly generated name for your sketch; in my example below, it generated `Jet spoonbill`. There is a little pencil symbol next to the words. This means you can edit the name to something more meaningful or just leave it as it is if you wish. This is entirely up to you; a new set of words is generated each time you start a new sketch. This is another reason for creating an account so that you can save your sketch under that name and access it again later on, carrying on from where you left off.

Figure 10: naming sketches

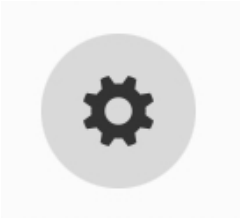
Question 11 - Is the auto refresh useful?

The button before the sketch name box is the auto-refresh button; if that is highlighted or selected, then I recommend that you unselect it by clicking on it. It may be unselected by default, in which case for now leave it as is, but later on when you are more confident in what you are doing, you may like to select that option.

Figure 11: auto refresh



Question 12 – What do the other settings do?



Here we have a button on the far right-hand side which looks like a cog wheel (see opposite). If you click on it, you get a `settings` menu which is split into two parts. The first one is `General Settings` and the second is `Accessibility`.

In `General Settings`, you will have a list of options, most of which (if any) you don't have to alter:

Themes

Depending on what works best for you, choose a theme from a choice of three that is easiest on the eyes. I quite like the high contrast, but for this tutorial, I use the default `light`.

Text Size

Is pretty obvious, but bear in mind that if you make it too big, you will get long lines of code wrapping themselves onto the next line, which I try not to do for this tutorial. Choose the size that works for you.

Auto Save

It might be worth keeping it on for now, but sometimes I want to go back to the last version I saved manually, and if it has saved a later version, it can be an issue, but it is swings and roundabouts whether to have it or not. My preference is to leave it off and to manually save the sketch after each significant change. This is a discipline that is easy to forget. Leave it on if unsure at this stage; or just practise saving as you go along.

Autoclose Brackets and Quotes

This is a useful one to keep selected; you don't have to, but I would recommend it. You will see the difference if you switch it off; it saves you a bit of time, as you will see.

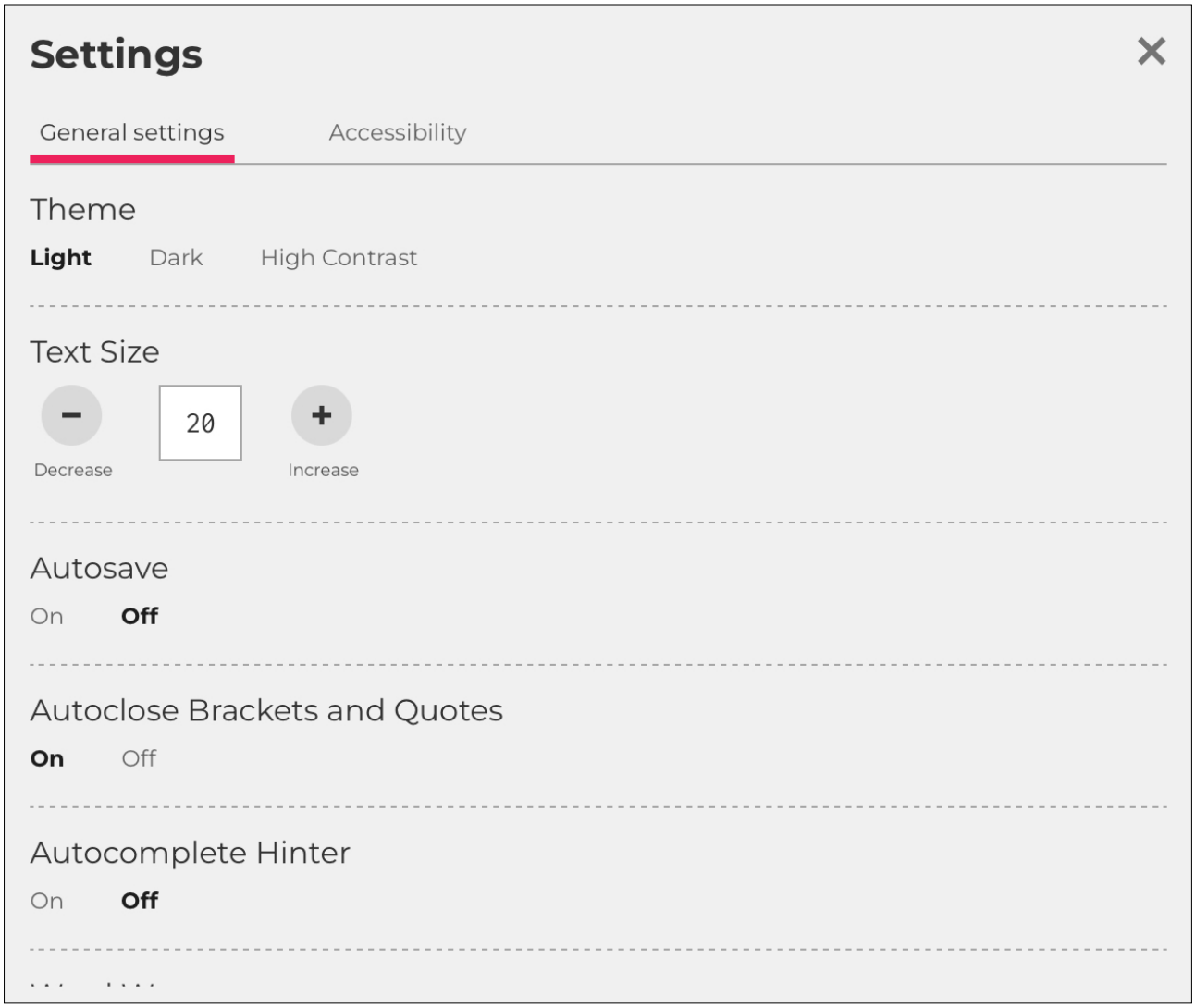
Autocomplete Hinter

I found this one very annoying if left on. I suspect some coders like it and use it a lot, but I found it got in the way of my coding as it tried to predict what I was about to write; my strong recommendation is to deselect it if it is on by default.

Text Wrap

This is only an issue if the line of code reaches the edge of the panel. Leave it on if you always want to see the code and not disappear.

Figure 12: general settings



Question 13 – What does the accessibility tab mean?

The second menu is Accessibility.

Line Numbers

You can switch the line numbers off or on if you wish; the line numbers are useful if there is an error message, and it can often tell you on which line of code the error message relates to.

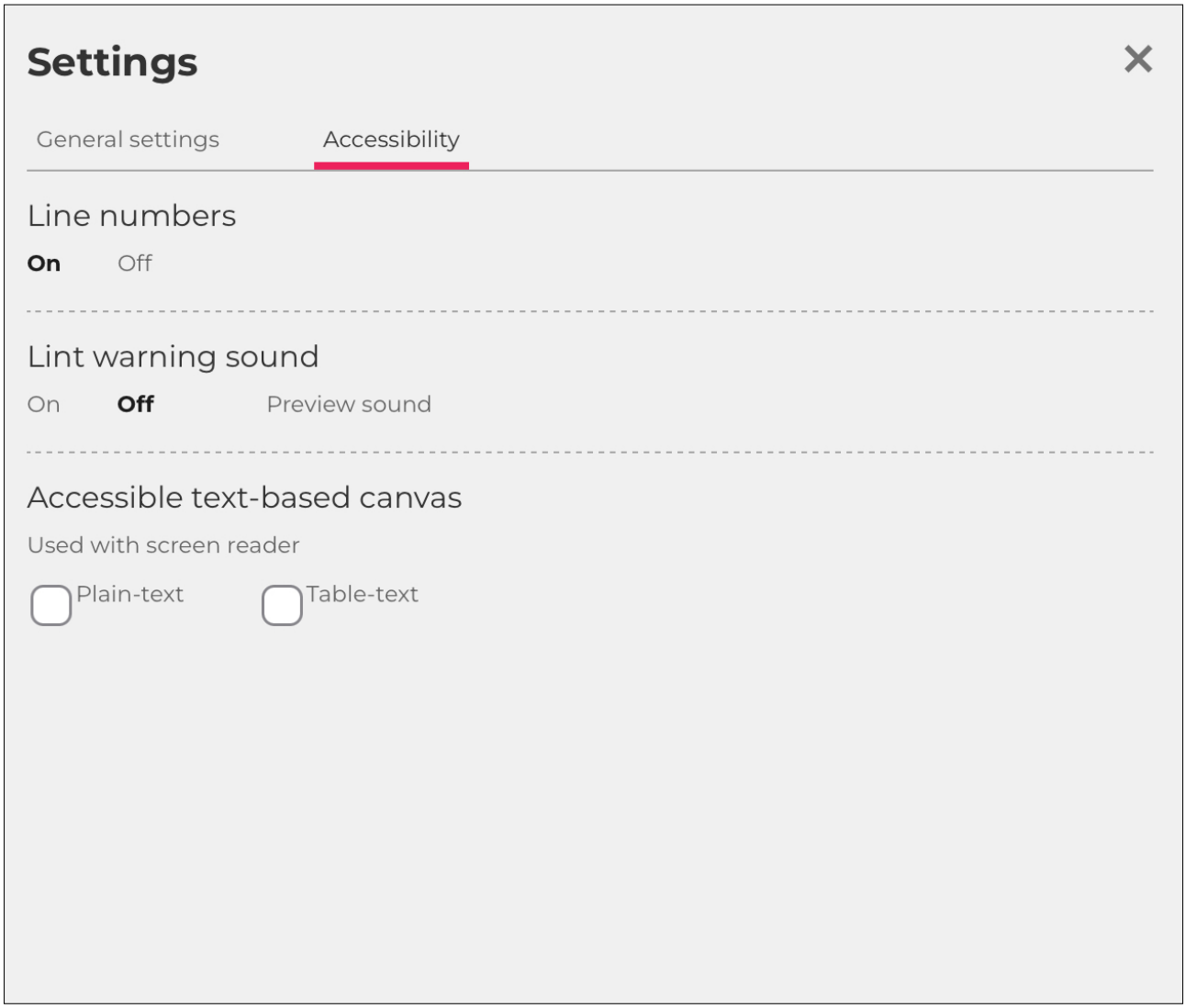
Lint Warning Sound

This gives you an audible sound if it detects an error in the code; again, it is a preference to have or not have it selected; I don't; I think it would drive me mad after a while.

Accessible text-based canvas

Not entirely sure what these do, to be honest, but they may mean something to you.

Figure 13: accessibility



Question 14 – Is the main tab menu important?

We have a list of main menu tabs or headings. If you click on any of the following: File, Edit, Sketch, Help, and English, you will get a subheading list. I will make reference to some of the key subheadings that you are likely to use.

The main heading you will use is File, as it is for managing your sketches. The Edit is also useful for finding words and/or replacing them. If you tidy your code, it will do just that and it will do it in the default format with the curly brackets and the semicolons. The Sketch is where you can add files, folders, run and stop; there are other ways to do that, which I will show you later (run and stop I have already covered). The Help is where you can get some additional information, and English is where you can change the language used.

Figure 14: main menu

Question 15 – How do I save my work?

Under the File menu, you get either a short menu if you have not saved your sketch, or a longer menu if you have. I am showing you the longer version.

New
This gives you a brand new sketch with the default code and a new randomly generated name.

Save
Use it as often as you can, just in case something goes wrong unexpectedly, so you have a backup up to that point. I generally don't learn many shortcuts, but this is one shortcut worth learning, as you might notice the command symbol plus the letter S; you press both together rather than going through the menu each time. There will be slight variations for different machines and keyboards.

Duplicate
You can find sketches on the internet or one of mine that you want to use. If you have found a sketch that you want to keep, use or edit for whatever reason, then you can duplicate it (make a copy) and save it under a new name. That way, you can have your own version.

Share
You may want to create a link to your sketch or even embed it in your website. This tab gives you a number of options.

Download
This is something I have never used and am not sure when you would use it.

Open
This takes you to another part of the web editor where you will see a list of all your saved sketches. If you have a lot (and I do), then you can search for specific keywords. Another reason why it is important that you rename your sketch with something meaningful in case one day you want to find it. They are saved chronologically.

You have two other tabs under the Open tab, Collections and Assets:

Collections
This is where you can view any collections you have created as well as creating them.

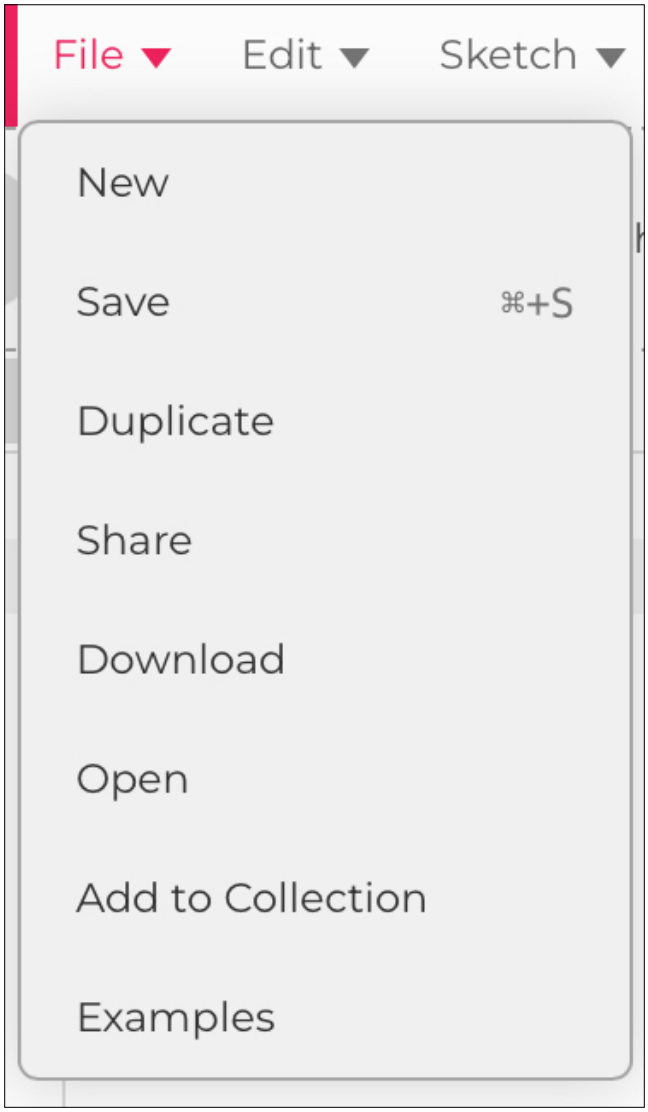
Assets
When you add images, videos, music, etc., this is where you can see them and which files they are linked to. There is a finite amount of memory space to keep them, but there is plenty for many sketches and numerous images and files.

Add to Collection
One way to sort and manage your sketches is to make a collection and add the sketch to it; this is similar to a folder to hold a specific collection of sketches and can be useful if you start creating sketches for different purposes, for instance, games or art.

Examples

A large repository of useful examples, well worth a quick explore to see some interesting stuff.

Figure 15: file tab



Question 16 – How useful is the edit function?

The Edit menu gives you some useful functions.

Tidy Code

This will tidy your code but will put all the semicolons back in and move the curly brackets to where they are normally kept, so it will change the appearance of your code considerably, assuming that you have adopted my format approach.

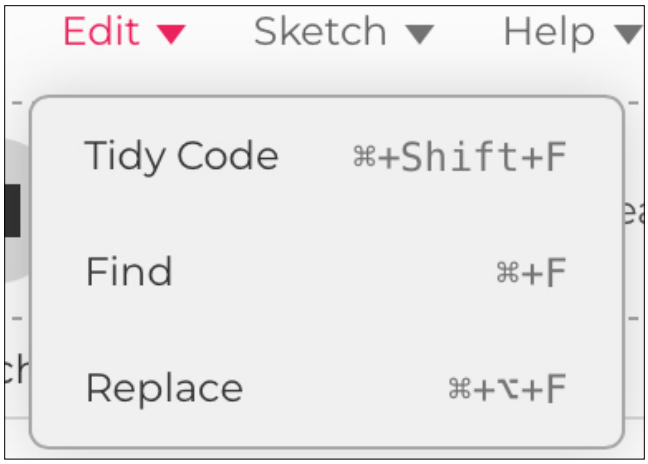
Find

This will search for any specific keyword or phrase in your code.

Replace

This gives you the option of finding a particular word or phrase and an option to replace it. You might use this to change the name of a variable.

Figure 16: edit tab



Question 17 - How do I add files?

The Sketch menu offers an alternative place to run a number of actions.

Add File

This is where you can create another JavaScript file, such as vehicle.js; you can also do this from the sidebar.

Add Folder

This is where you can create a folder to put a collection of files or upload images, videos, models or data files. You can also do this from the sidebar.

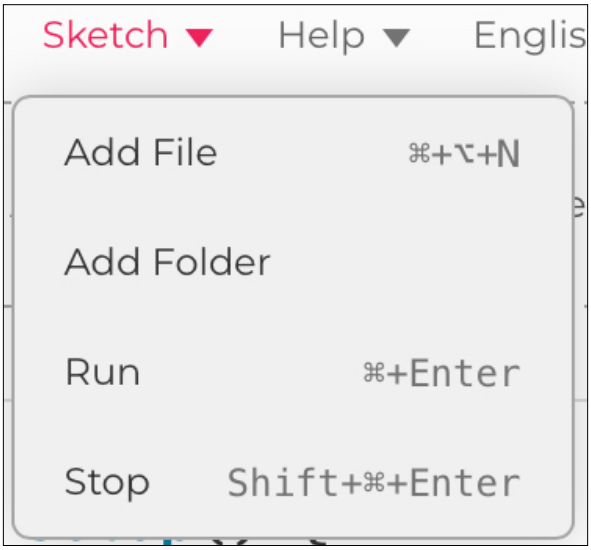
Run

This does the same as the button.

Stop

This also does the same as the button.

Figure 17: sketch tab



Question 18 – Is the Help menu helpful?

The Help menu is quite useful and worth a look at.

Keyboard Shortcuts

There are quite a few and some are worth learning, such as Save, but others are probably not so useful; it all depends on whether you are generally good at learning shortcuts and would use them.

Reference

This opens up a new tab in your browser. This is an immense resource and can look a bit bewildering. There is a search box and there are lots of examples and a reference section. It would take too long to go through it, but I would highly recommend having a look at it.

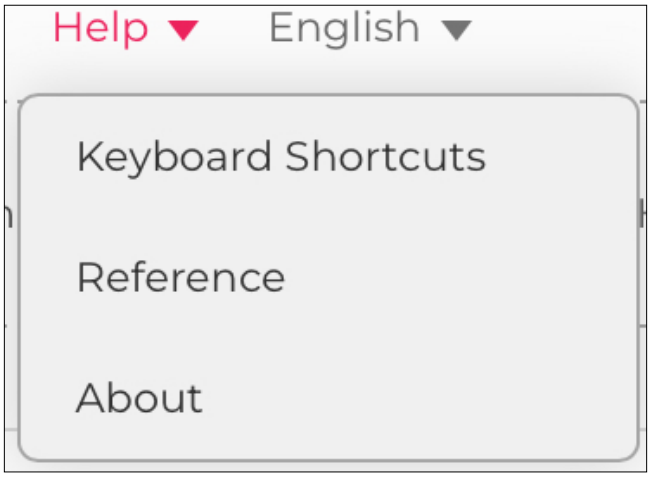
About

This has some links and general information.

English tab

You can change the language to one of many if your first language isn't English.

Figure 18: help tab



Question 19 – What is the console for?

The `console` is the grey-boxed section underneath where you enter your code. This is a very useful function for a number of reasons. It is where you will get error messages and where you can send information. Its main purpose is for debugging, which is another word for problem-solving.

Error messages

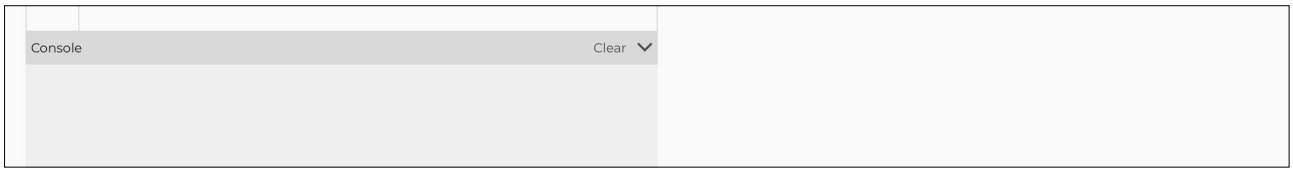
If your editor picks up on some glaring and obvious errors in your code, which can be anything from a missed comma or semicolon to an unknown variable appearing. Mostly you will get something useful and informative, although sometimes it just flags a problem and leaves you to search for it.

Sometimes the error code may be highlighted in red, and it may even give you the line number to identify where the problem is (or the following line number where it does become a problem).

Console log

We can put a line of code in called `console.log()`. This means you are going to log something in the console. Sometimes it can be a piece of text such as `console.log('all done!')`, or it can be the value of a variable such as `console.log(counter)`, which will give you the value of the variable at that time in the code. Another really helpful use is to debug a problem where you are not sure what is happening, for example, to an array.

Figure 19: the console



Final words for the first unit

If you did not understand all the above please do not worry I will go through their uses when appropriate, I just wanted to make you aware of them and leave you to play around with them at a later date. I cover some of them in one of my introductory videos.

On the whole, there is a lot to say about the web editor, but most of it is fairly intuitive. The best way is to play and explore, but for now, you can have a read through to see what options are available. The reference section will be very useful later on as you become more confident and competent.

Now, let's get coding and have some creative fun.

The Joy of Coding Algorithmic Art Module A Unit #2 your first circle

Module A Unit #2: your first circle

This may not seem very challenging, but using the `circle()` function means I can introduce you to many coding concepts that you will use in your journey into creative coding.

Key concepts:

`drawing a circle`
`names of colours`
`variables`
`random`
`operators`
`while() loop`
`for() loop`

Sketch A2.1 your first sketch

First of all, you need to delete the default code and type the following. It is really just the same, but I want to arrange it in a more intuitive way. The semi-colons aren't essential, and the first curly bracket {} doesn't have to start on that line (although it is common to do so).

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

function draw()
{
  background('lightgrey')
}
```

Notes

Your screen should look something like the image below.

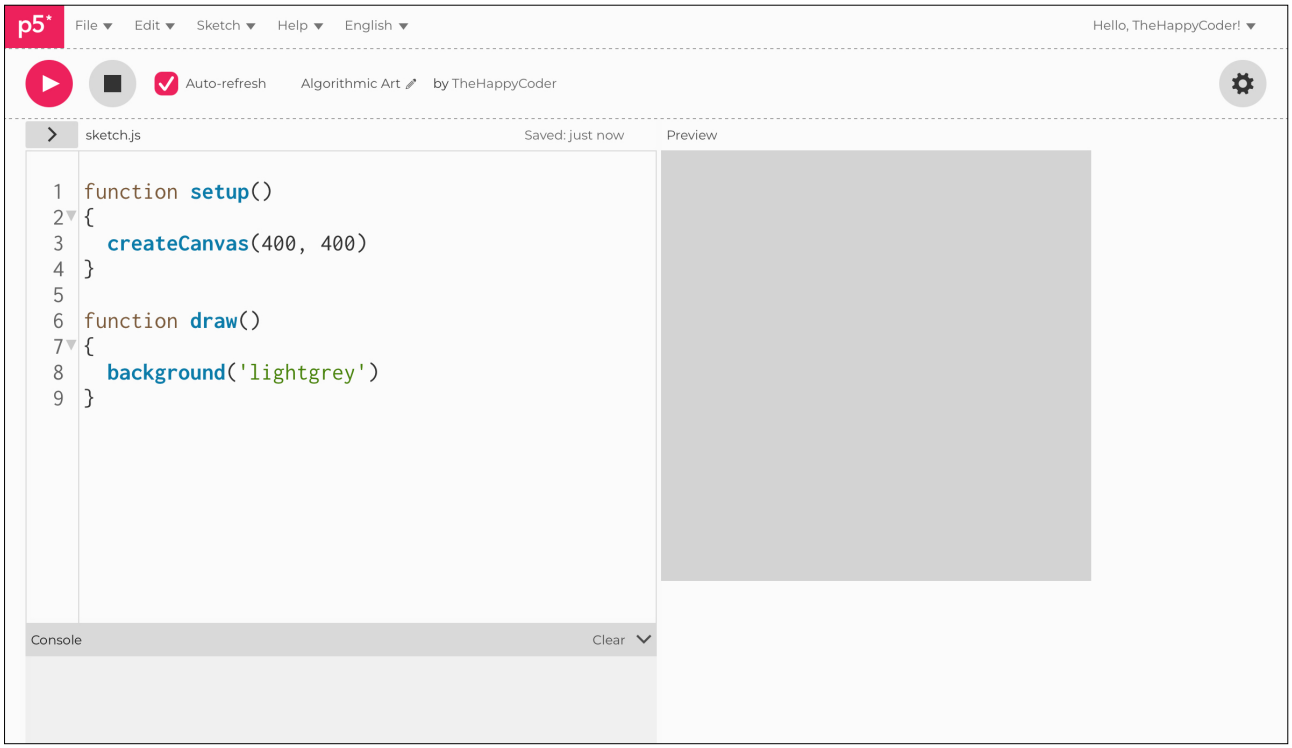
Challenge

Change the size of the canvas from
`createCanvas(400, 400)` to `createCanvas(600, 200)`

Code Explanation

`function setup()`	Everything in the first function only happens once.
`{ }`	All the code goes inside the curly brackets (or braces).
`createCanvas(400, 400)`	We are creating a canvas just like an artist ready to paint. This canvas is 400 pixels wide by 400 pixels high.
`function draw()`	This function operates a continuous loop.
`background('lightgrey')`	We give the background a colour. In this instance, it is a light grey colour (more on colours later).

Figure A2.1



Sketch A2.2 a circle

When we add or change a line (or lines) of code, they will be highlighted blue. We are going to add your first shape, a circle.

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

function draw()
{
  background('lightgrey')
  circle(200, 200, 100)
}
```

Notes

Because the background and circle are in the `draw()` function, the programme code is drawing them continuously. First, the background, and then the circle.

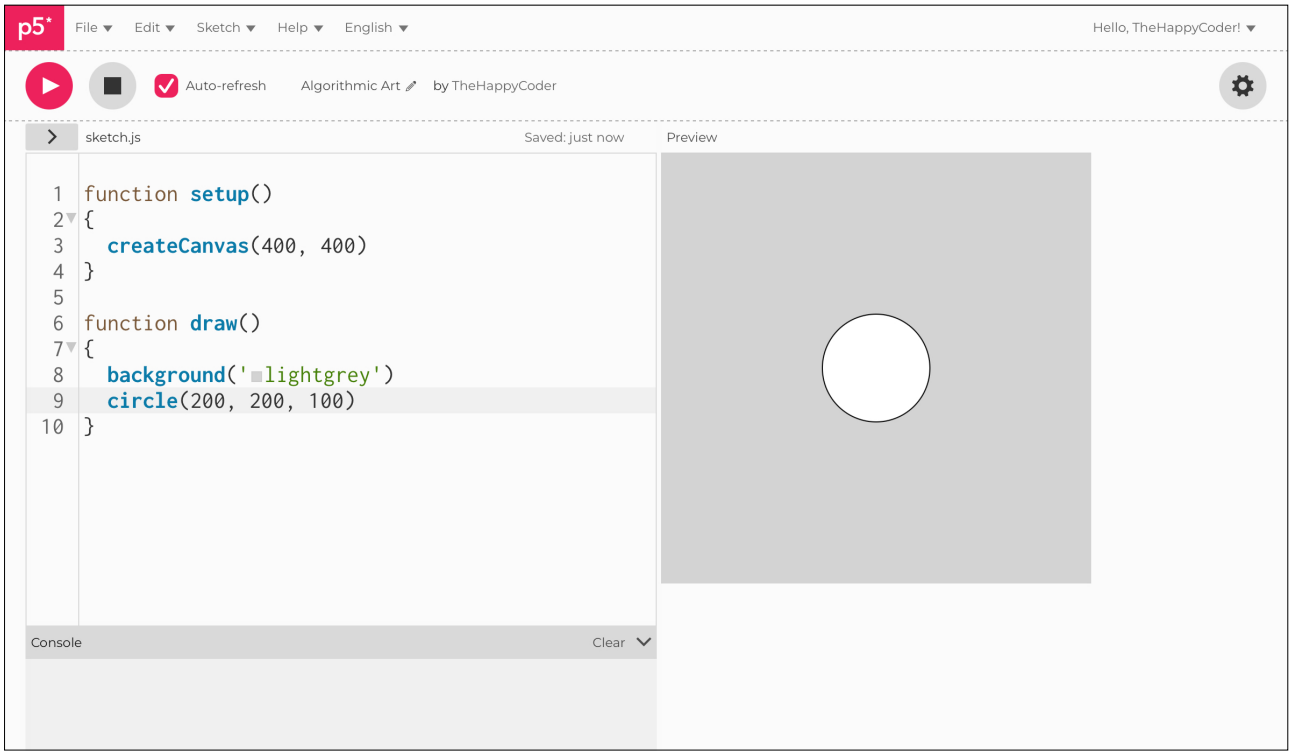
Challenges

1. Change the co-ordinates from: `circle(200, 200, 100)` to `circle(100, 300, 100)`
2. Change the circle diameter from: `circle(100, 300, 100)` to `circle(100, 300, 50)`

Code Explanation

`circle(200, 200, 100)`	The centre of the circle is at position 200 pixels from the left-hand edge of the canvas and 200 pixels from the top of the canvas. The diameter is 100 pixels.

Figure A2.2



Sketch A2.3 adding another circle

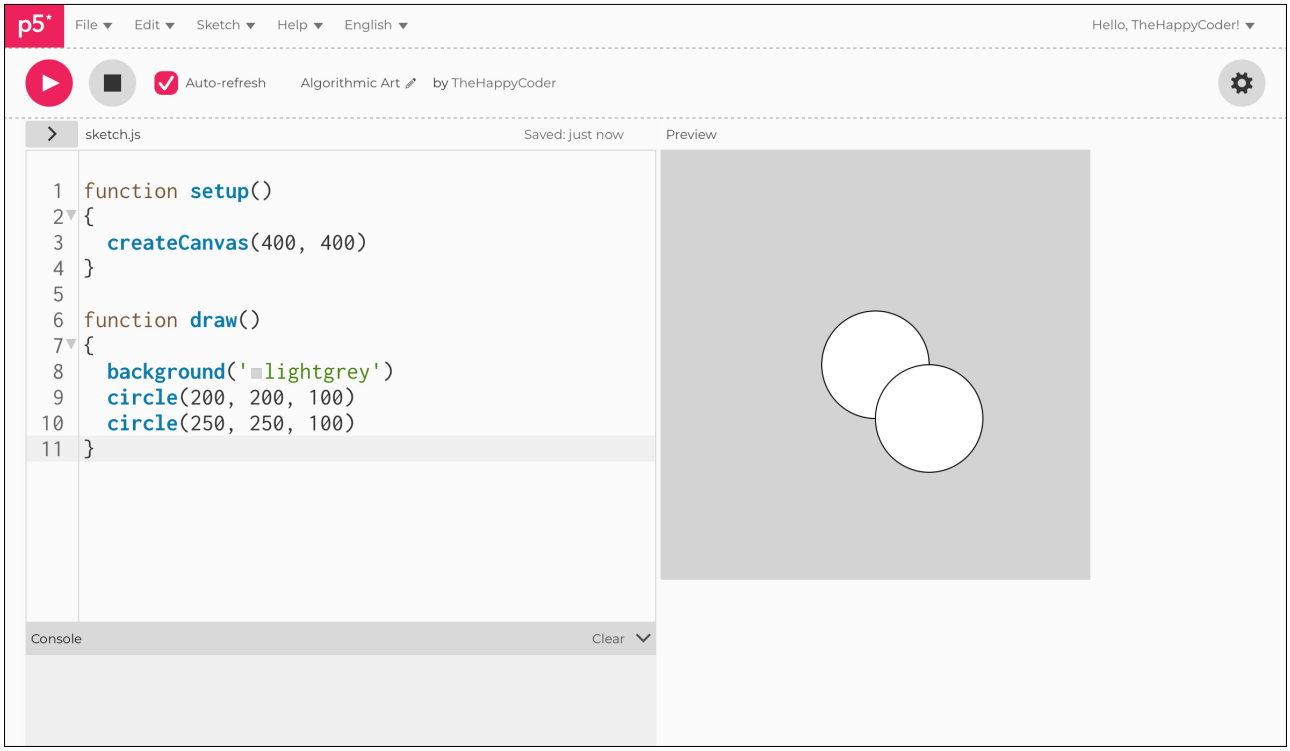
We can add more circles (and other shapes)

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

function draw()
{
  background('lightgrey')
  circle(200, 200, 100)
  circle(250, 250, 100)
}
```

Notes

Notice that the second circle overlaps on top of the first circle. The programme works from top to bottom, one line of code at a time. In this instance, it draws the background first, then the first circle, then the second circle, and then repeats.

Figure A2.3



Sketch A2.4 making a simple pattern

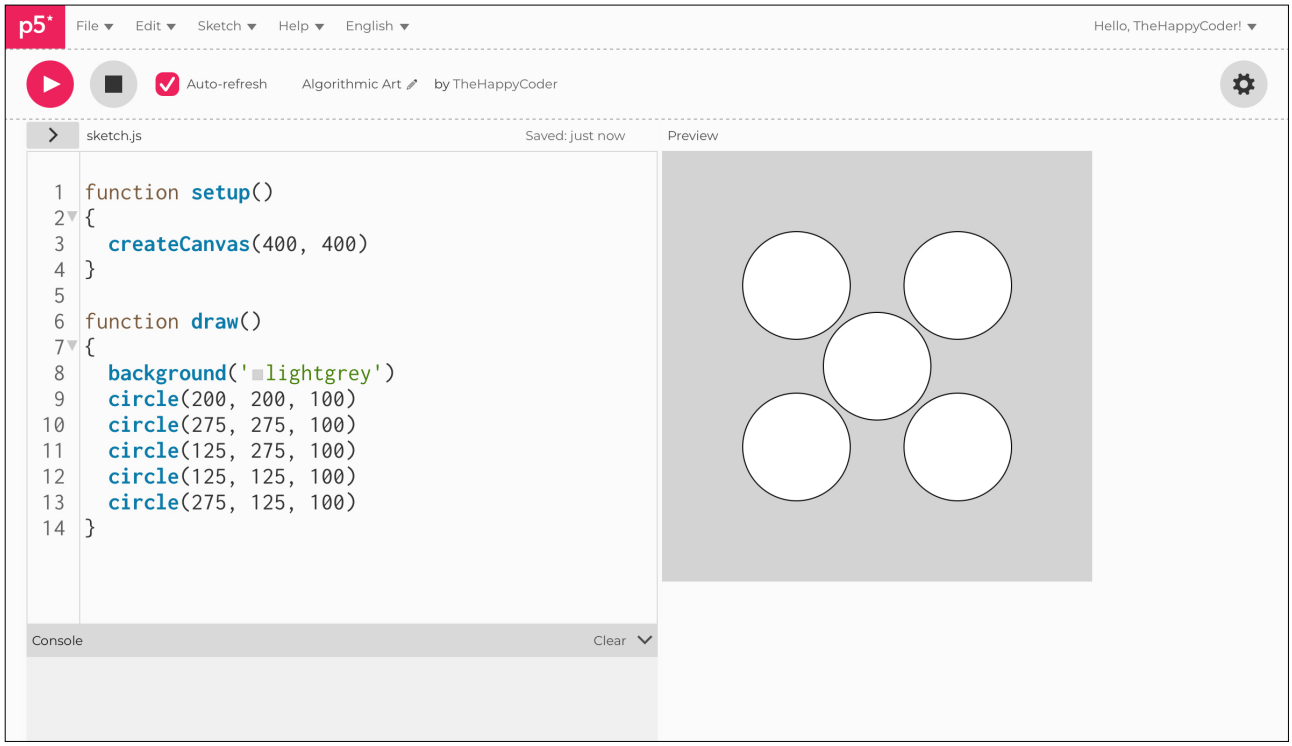
We will change the position of the second circle and add three more circles.

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

function draw()
{
  background('lightgrey')
  circle(200, 200, 100)
  circle(275, 275, 100)
  circle(125, 275, 100)
  circle(125, 125, 100)
  circle(275, 125, 100)
}
```

Notes

You might notice in the image below that next to `Auto-refresh`, the box is ticked and is highlighted in red. This is useful (as well as dangerous) so that you don't have to keep pressing the run (play) button. It can be dangerous if you are using `for()` loops (more on that later) where you can accidentally get into an infinite loop and crash the programme.

Figure A2.4



Sketch A2.5 adding some colour

Before we move onto other shapes and key concepts, we can explore colour. To start with, we can use the names of colours such as `red`, `green`, `blue`, or `orange`, for example (there are more but a limited number). To do this, we use the `fill()` function. First off, let's colour them `red`.

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

function draw()
{
  background('lightgrey')
  fill('red')
  circle(200, 200, 100)
  circle(275, 275, 100)
  circle(125, 275, 100)
  circle(125, 125, 100)
  circle(275, 125, 100)
}
```

Notes

You will notice that in the code, you get a little red box. This just illustrates the colour that you are requesting. There are 140 named colours. You can see them all at: https://www.w3schools.com/colors/colors_names.asp or just search for JavaScript colour names

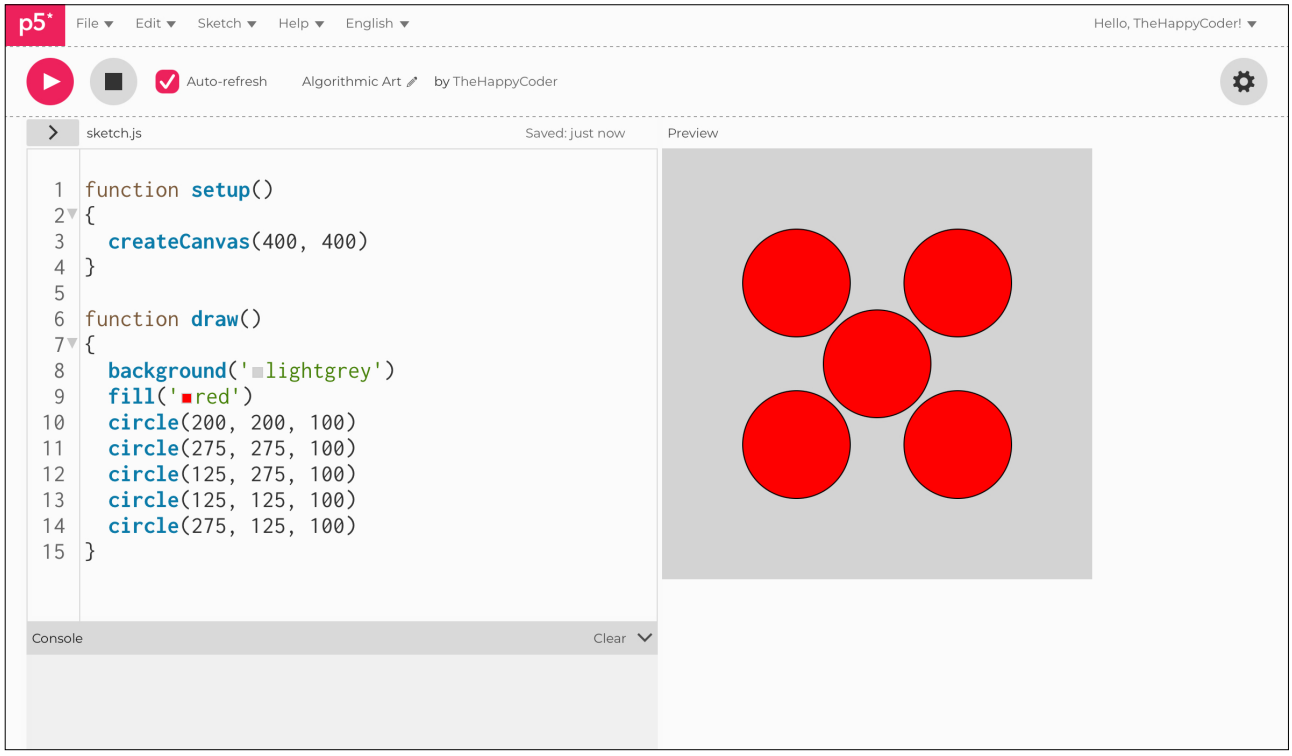
Challenge

Try other names of colours

Code Explanation

`fill('red')`	This is the function to fill any shape any colour you chose. If you use the name of a colour then you have to put it in speech marks (single or double)

Figure A2.5



Sketch A2.6 different colours

We will give each one a different colour.

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

function draw()
{
  background('lightgrey')
  fill('red')
  circle(200, 200, 100)
  fill('green')
  circle(275, 275, 100)
  fill('blue')
  circle(125, 275, 100)
  fill('yellow')
  circle(125, 125, 100)
  fill('purple')
  circle(275, 125, 100)
}
```

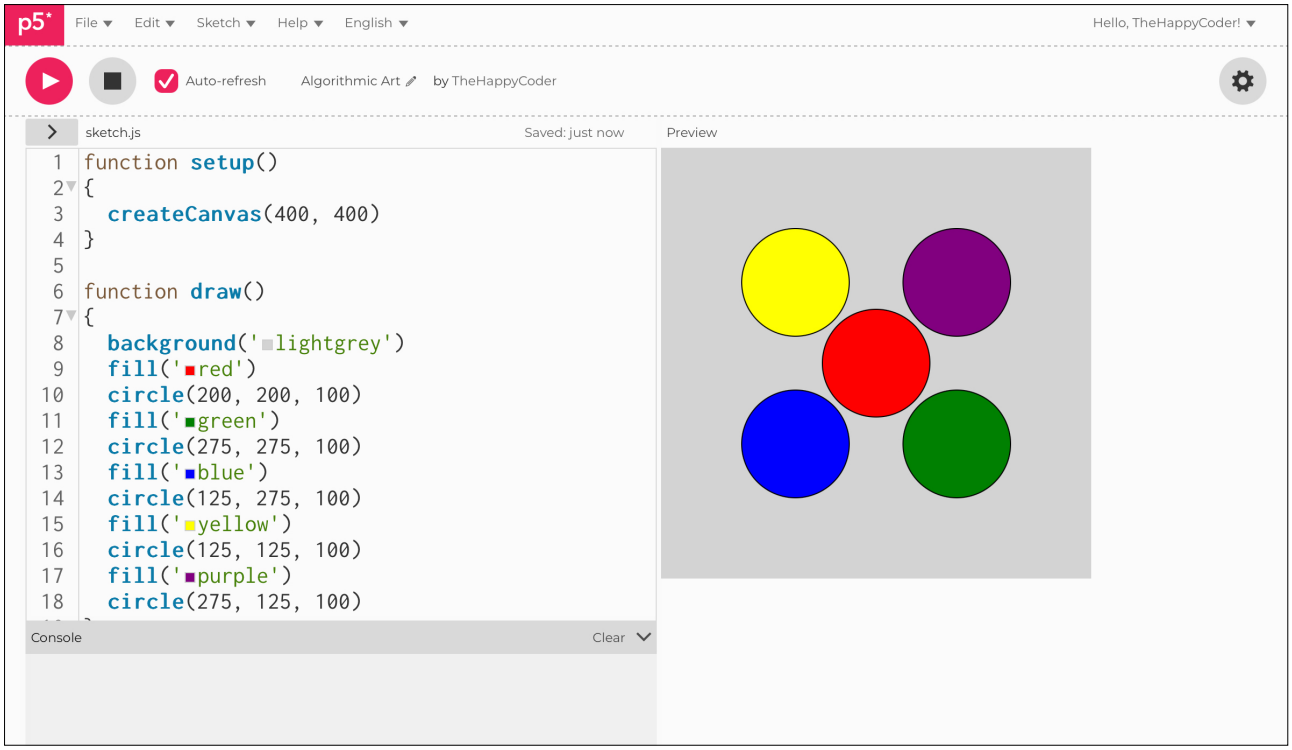
Notes

To get each separate colour to fill the next circle, we have to put the `fill()` function in between the circles. The code runs in a linear fashion; it goes step by step, one line at a time, starting at the top and then works its way down till it reaches the bottom. Then, with the `draw()` function, it goes to the top and starts all over again.

Challenge

Try other colours and see which colours work and which don't

Figure A2.6



Sketch A2.7 new sketch

! Start a brand new sketch.

Writing the code for five circles is not too arduous, but what if you want to draw a hundred or a thousand circles? We are going to need a bit of help. Programming is all about efficiency, and this usually means writing code in the fewest lines as possible. Delete the previous code and write this.

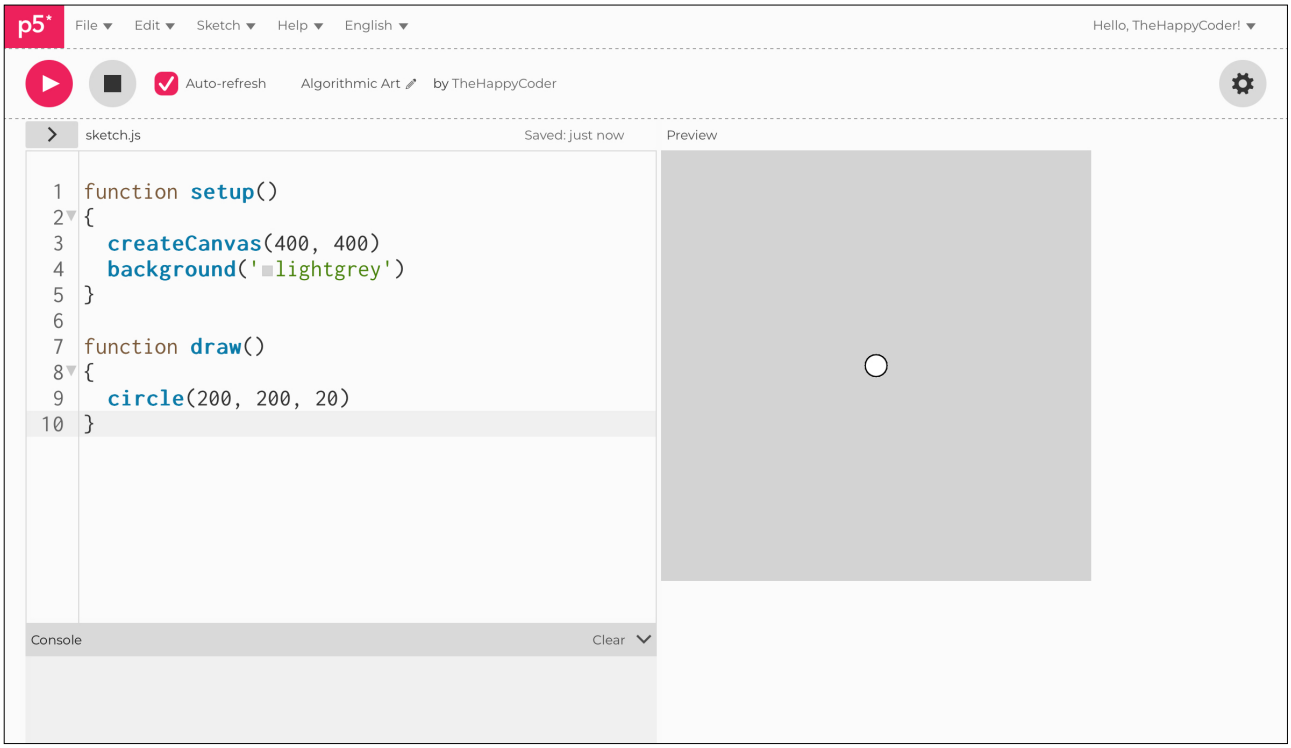
! Notice that the `background()` goes into the `setup()` function.

```
function setup()
{
  createCanvas(400, 400)
  background('lightgrey')
}

function draw()
{
  circle(200, 200, 20)
}
```

Notes

We have drawn a small (20-pixel diameter) circle in the centre of the canvas, but before we move onto the next sketch, I need to introduce an important concept: variables.

Figure A2.7



Introduction to variables

Variables are very useful for storing data that we may want to access or change later. Variables can be named with a single letter or a name. They can have numbers in them but must never start with a number. Usually, the name of the variables has some relevant meaning.

For instance, if you want to have a variable for speed, you would be best to use the word `speed` rather than just `s` because later on in a long list of code, you may forget what `s` represents. This is especially the case if you have a lot of variables. At the same time, don't make them too long; otherwise, the code will look very messy and difficult to read by anyone else but you.

In this next example, we are going to give the coordinates for the circle names `x` and `y`. So that we can alter them in a later sketch. Variables are very powerful and extremely useful.

To use a variable like `x` and `y`, then we need to define them. We use the key word `let`. You can just define it or initialise it, for example:

`let x`	This defines x as a variable.
`let x = 10`	This gives x an initial value of 10.

Now in the sketch below we are going to create a variable for the `x-coordinate` (which is `200`) and the `y` co-ordinate (which is `300`) of the circle by calling them `let x` and `let y`.

One other important detail, `scope`. It is always a good idea to declare variables at the beginning of a sketch, that way they are available everywhere. If you only declare them between two curly brackets they only live between those curly brackets and nowhere else. You will see that I do both so it is usually OK but something to bear in mind when generating many functions and lines of code.

Mostly we will be using numeric values (integers and floats) but just occasionally we will be using strings (words or letters).

Sketch A2.8 adding some variables

We have replaced the x value and y value with variables of that name. This draws the same small circle in the centre of the canvas.

```
let x = 200
let y = 200

function setup()
{
  createCanvas(400, 400)
  background('lightgrey')
}

function draw()
{
  circle(x, y, 20)
}
```

Notes

This demonstrates how using variables can be very useful, especially when there are many shapes and lots of movement. If you are familiar with data types `let` is a float (also called real) by default. The line of code `circle(x, y, 20)` is just the same as `circle(200, 200, 20)`.

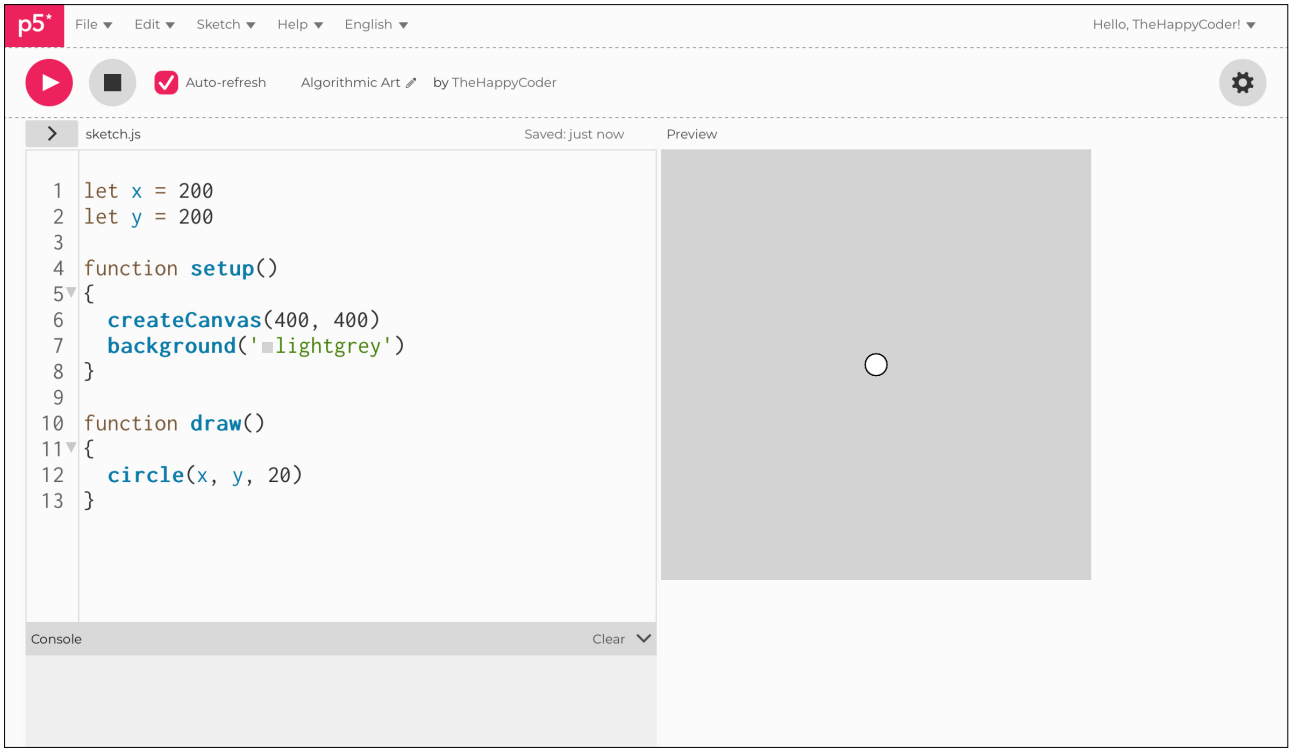
Challenges

1. Use different initialised values for the x and y variable
2. Create another variable for the diameter

Code Explanation

`let x = 200`	Declares the variable x and initialises it to a value of 200.
`let y = 200`	Declares the variable y and initialises it to a value of 200.
`circle(x, y, 20)`	The circle is drawn at the initialised x and y values (200, 200).

Figure A2.8



Introduction to Random

Now we will be making shapes move and introduce random. Different functions as loops will be explored. This is where we can create a random number. It is not really random but for most purposes it is random enough. We use the function `random()` and it works like this...

`random(20)`	It will give you a number between 0 and 19 (but not 20).
`random(12, 37)`	This will give you a value between 12 and 37 (but not 37).

Before we go any further we need to look at notations used with coding which are also used in maths. Just read quickly through the next section to get a feel for the notations and what they mean. As you use them they will make more sense, I include them here and now so that you have a reference.

Symbols we use – Operators

Please just skim read through these; they are for reference and you don't need to learn them just yet. I include them now without any context so that you are aware of them. You may find them a useful reference later on.

In coding (as well as in maths), we use notation, symbols rather than words. Most of the maths used in coding is fairly basic; some symbols, however, may be unfamiliar. We are going to quickly cover the following:

Arithmetic Operators

Simple mathematical operators you will use frequently. They are used throughout all coding and should be quite familiar except for, perhaps, `modulus`.

+	Addition `(2 + 4)` will give you 6.
-	Subtraction `(6 - 3)` will give you 3.
/	Division `(8 / 2)` will give you 4.
*	Multiplication `(3 * 5)` will give you 15.
%	Modulus gives you the remainder, `(10 % 8)` will give you 2.
=	Equals (not equals to) `2 + 4 = 6`

Comparison Operators

In short, these are conditional statements where a condition needs to be met. These are often used with `while()` loops

==	means equal to (5 == 5) is TRUE, (6 == 5) is FALSE.
<	means less than (6 < 8) is TRUE, (8 < 6) is FALSE.
<=	means less than or equal to (6 <= 8) is TRUE, (6 <= 6) is also TRUE.
>	means greater than (8 > 6) is TRUE, (6 > 8) is FALSE.
>=	means greater than or equal to (8 >= 6) is TRUE, (6 >= 6) is also TRUE.
!=	means not equal to (6 != 5) is TRUE (5 != 5) is FALSE.

Logical Operators

These are used with `if()` statements. The `if()` statement can be similar to the `while()` loop; if something is true or a condition is met, then do something, for example: `if(x < 100 && y > 50)` means if x is `less than` 100 `AND` y is `greater than` 50.

&&	means AND (x < 100 && y > 50).
||	means OR (x < 100 || y y > 50).
!	Means NOT (x < 100 ! y y > 50).

Assignment Operators

They are used often in coding (some much more than others). It is more obvious when you see them in a meaningful context, which often goes for all coding.

++	means increasing by 1 (x++) x is incremented by 1 each time.
––	mean reducing or subtracting by 1 (y–) y is decremented by 1 each time.
+=	means addition (x += 10) or (x += y), same as x = x + y.
–=	Means subtracting (x –= 10) or (x –= y), same as x = x – y.
*=	means multiplying (x *= 10) or (x *= y) same as x = x * 10.
/=	means division (x /= 2) or (x /= y) same as x = x / 10.

Maths Functions

When using the mathematical notation, these can be very useful. Here are just a few

`floor()`	Calculates the closest integer value that is less than or equal to the value of a number.
`abs()`	Calculates the absolute value of a number. A number's absolute value is its distance from zero on the number line. -5 and 5 are both five units away from zero, so calling **abs(-5)** and **abs(5)** both return 5. The absolute value of a number is always positive.
`round()`	Calculates the integer closest to a number. For example, **round(133.8)** returns the value 134. The second parameter, **decimals**, is optional. It sets the number of decimal places to use when rounding. For example, **round(12.34, 1)** returns 12.3. **It is** zero by default.
`sq()`	Calculates the square of a number. Squaring a number means multiplying the number by itself. For example, **sq(3)** evaluates 3 x 3, which is 9. The **sq(-3)** evaluates -3 x -3, which is also 9. Multiplying two negative numbers produces a positive number. The value returned by **sq()** is always positive.
`squrt()`	Calculates the square root of a number. A number's square root can be multiplied by itself to produce the original number. For example, **sqrt(9)** returns 3 because 3 x 3 = 9. The **sqrt() function** always returns a positive value. **sqrt()** does not work with negative arguments such as **sqrt(-9)**.
`pow`	Calculates exponential expressions such as 2^3. For example, **pow(2, 3)** evaluates the expression 2 x 2 x 2. **pow(2, -3)** evaluates 1 ÷ (2 x 2 x 2).
`ceil()`	Calculates the closest integer value that is greater than or equal to a number. For example, calling **ceil(9.03)** and **ceil(9.97)** both return the value 10.
`exp()`	Calculates the value of Euler's number e (2.71828...) raised to the power of a number.

Sketch A2.9 drawing random circles

In this sketch, you will draw circles in random positions on the canvas. The function `draw()` is a continuous loop and will draw them forever. You don't need to give `x` and `y` an initial value, but it is good practice to do so.

```
let x = 200
let y = 200

function setup()
{
  createCanvas(400, 400)
  background('lightgrey')
}

function draw()
{
  x = random(400)
  y = random(400)
  circle(x, y, 20)
}
```

Notes

A reminder that in computer code it starts counting from 0, not 1, and so it stops when it gets to the 400th number, which is 399. Just something to remember.

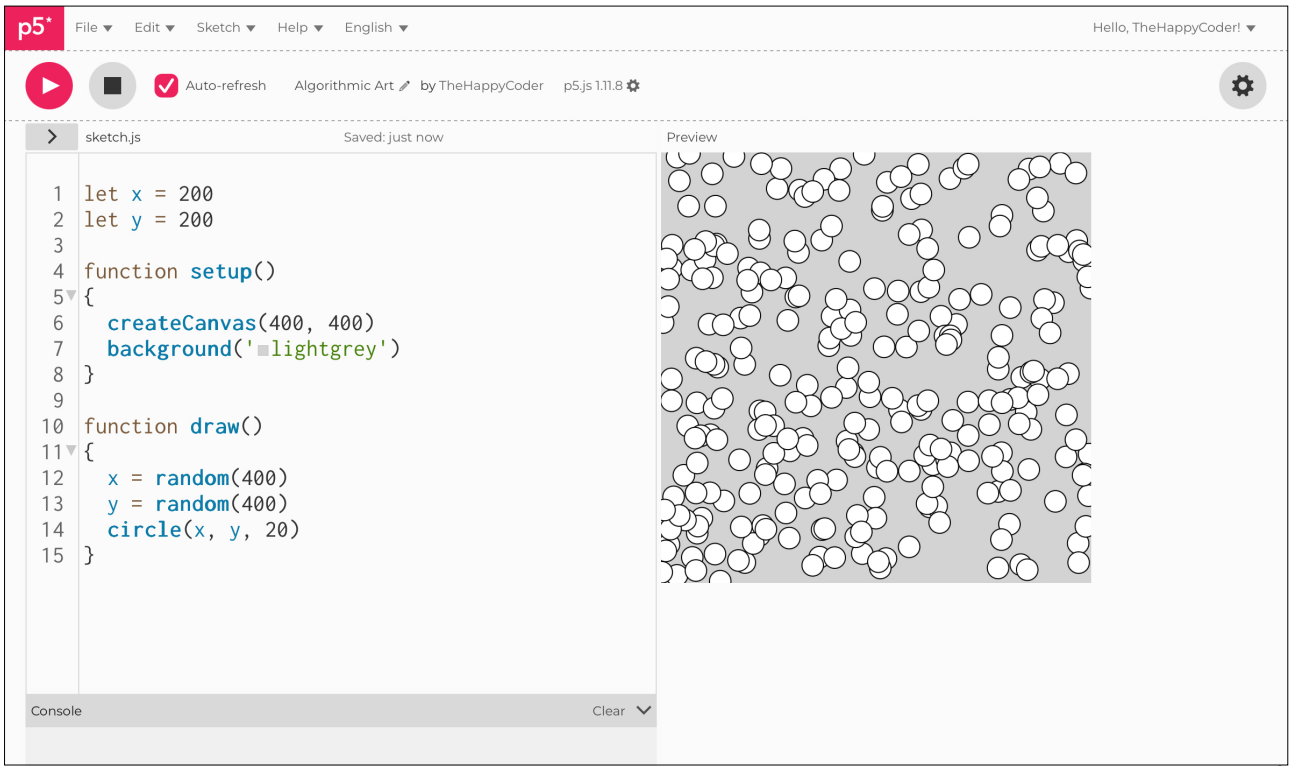
Challenge

Create a random diameter as well.

Code Explanation

`x = random(400)`	It chooses a random number from 0 to 399 for the x value.
`y = random(400)`	It chooses a random number from 0 to 399 for the x value.

Figure A2.9



Sketch A2.10 while() loop circles

! delete what is in the draw() function, notice we have a loop within draw and so things are double indented.

What we want to do is draw 100 circles and then stop. The while() loop keeps checking the number of circles it has drawn. If the number is less than (<) 100, it keeps going while it is true. Once it has drawn 100 and the next one is therefore more than 100, it then stops because it returns false. To keep track of the number of circles, we add a new variable called count.

! Make sure that Auto-refresh is OFF, otherwise the programme will likely crash as you type it in. It is the tick box near the top.

```
let x = 200
let y = 200
let count = 0

function setup()
{
  createCanvas(400, 400)
  background('lightgrey')
}

function draw()
{
  while (count < 100)
  {
    x = random(400)
    y = random(400)
    circle(x, y, 20)
    count = count + 1
  }
}
```

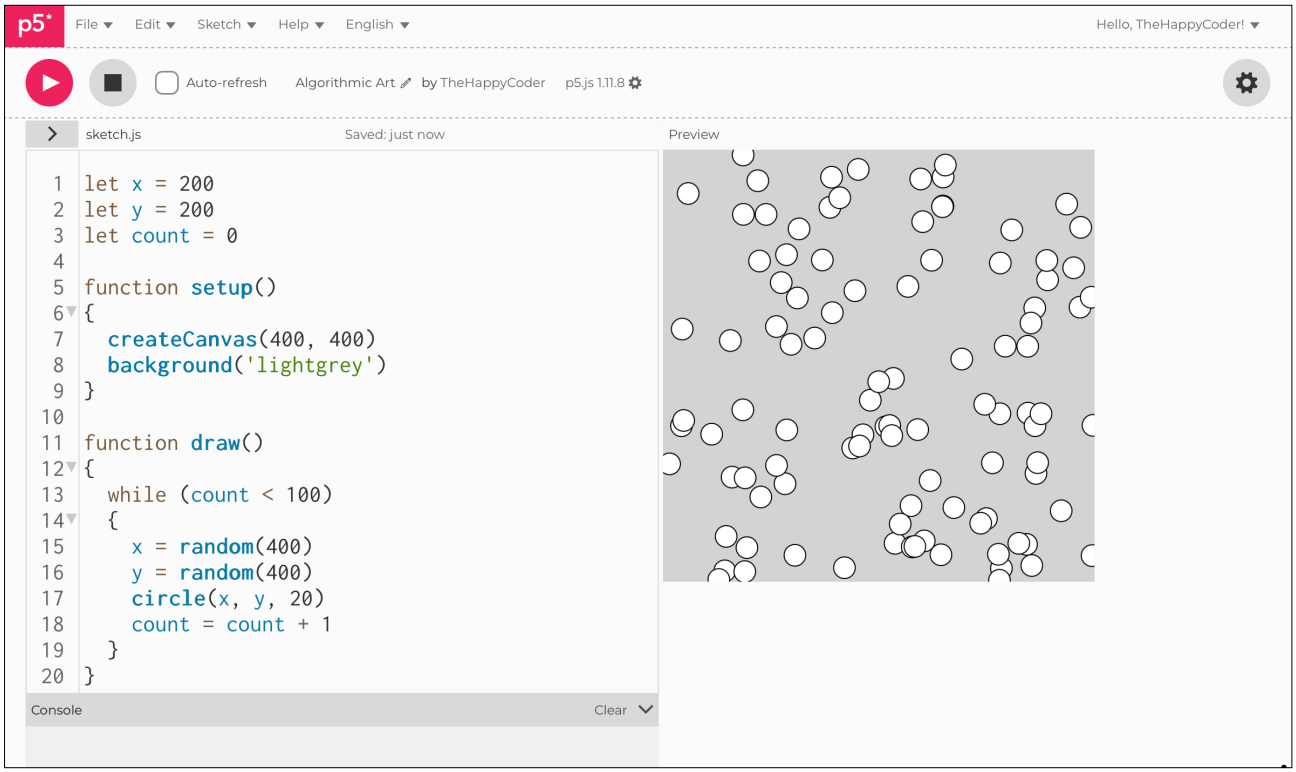
Notes

This sketch draws exactly 100 small circles in random positions every time you press the run button. It makes good use of the while() loop function. Also uses the comparison < less than.

Code Explanation

`while (count < 100)`	This is a while() loop which checks to see if count has reached 100.
`count = count + 1`	We add 1 each time to the count variable.

Figure A2.10



Sketch A2.11 a few more adjustments

We can use the `random()` function to create a border and introduce a very common shorthand for adding each time `1` (`++`). So, `count = count + 1` becomes: `count++`.

```
let x = 200
let y = 200
let count = 0

function setup()
{
  createCanvas(400, 400)
  background('lightgrey')
}

function draw()
{
  while (count < 100)
  {
    x = random(100, 300)
    y = random(100, 300)
    circle(x, y, 20)
    count++
  }
}
```

Notes

We have now limited the random range to between 100 and 300; this gives us a nice empty border around the canvas.

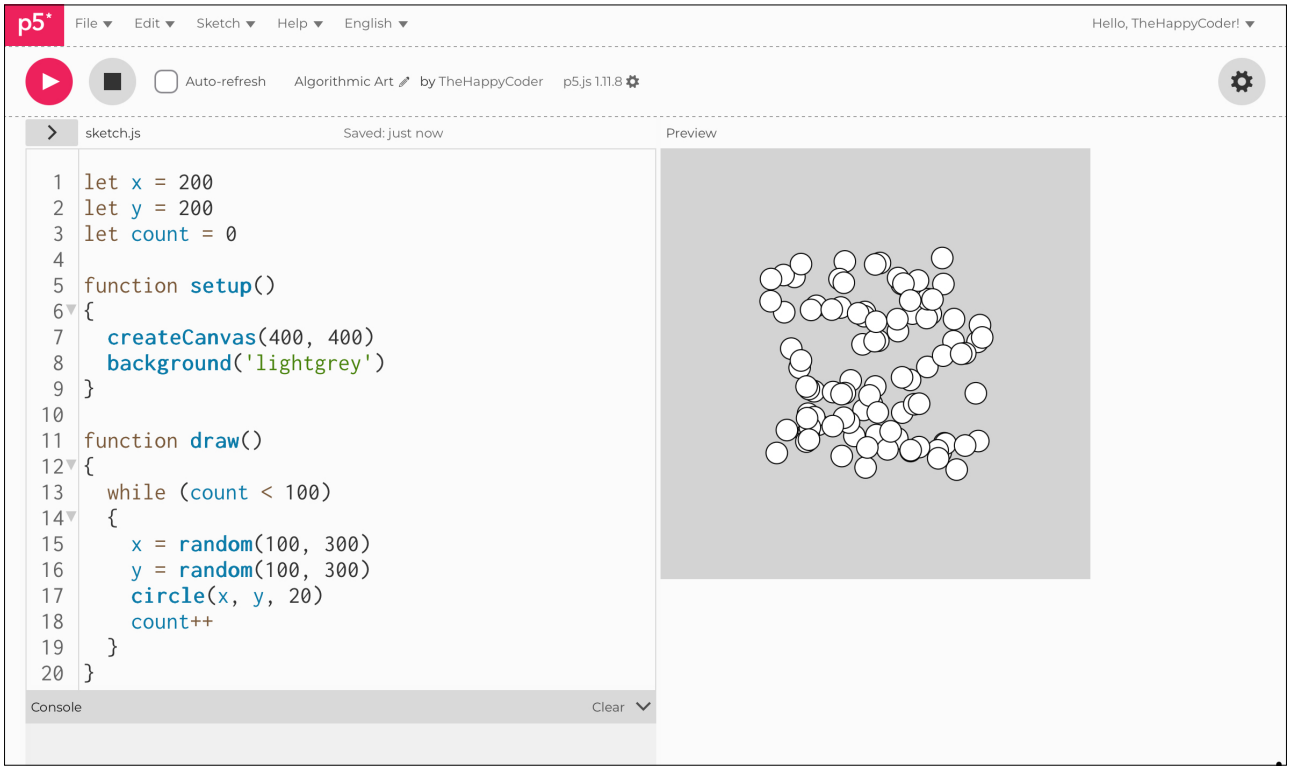
Challenge

Try different values for random.

Code Explanation

`x = random(100, 300)`	Returns a random number between 100 and 300.
`y = random(100, 300)`	Returns a random number between 100 and 300.
`count++`	Shorthand version: adds 1 each time, the same as count = count + 1.

Figure A2.11



Introducing the for() loop

You have already come across the `while()` loop. There is another called the `if()` statement (coming soon) which is similar to the `while()` loop. There is a third and it is arguably the most important or the most used. It is a little bit more complicated, but as we will be using it a lot, you will get used to it and it does become very intuitive (trust me).

Our example is as follows:

```
for (let i = 0; i < 100; i++)
```

Inside the `for()` loop, brackets are three parts separated by semicolons (`;`), they are:

1. `let i = 0`
2. `i < 100`
3. `i++`

Firstly, we create a variable called `i`. This is just convention; you can call it anything you like, and we give it an initial value of `0`.
Secondly, we put a condition on this variable, in this case, less than `100` (`<`). The `for()` loop is true while `i` is less than `100`.
Thirdly, we increment the loop, in this instance, we simply add `1` to `i` for each loop (`i++`) until `i` has the value of `100` and stops.

Sketch A2.12 using for() loops

Another way to draw 100 circles is to use a for() loop. It loops through as a sort of counter from 0 to 100 in steps of 1. We are still going to use i as the counter, but we initialise it inside the for() loop itself. The noLoop() function stops the draw() function but only after it has drawn 100 circles.

! Remove: all reference to the count variable, I recommend starting a new sketch, it helps memory muscle.

```
let x = 200
let y = 200

function setup()
{
  createCanvas(400, 400)
  background('lightgrey')
}

function draw()
{
  for (let i = 0; i < 100; i++)
  {
    x = random(100, 300)
    y = random(100, 300)
    circle(x, y, 20)
  }
  noLoop()
}
```

Notes

Draws 100 circles in random positions once between 100 and 300 using the `for()` loop. For `(let i = 0; i < 100; i++)` draws 100 circles in random positions once between 100 and 300 using the `for()` loop. There are three parts to it:

Part 1: `let i = 0;`
Part 2: `i < 100;`
Part 3: `i++`

Persevere with the `for()` loop as it will appear a lot in the future. Notice we have the semicolons separating the three parts.

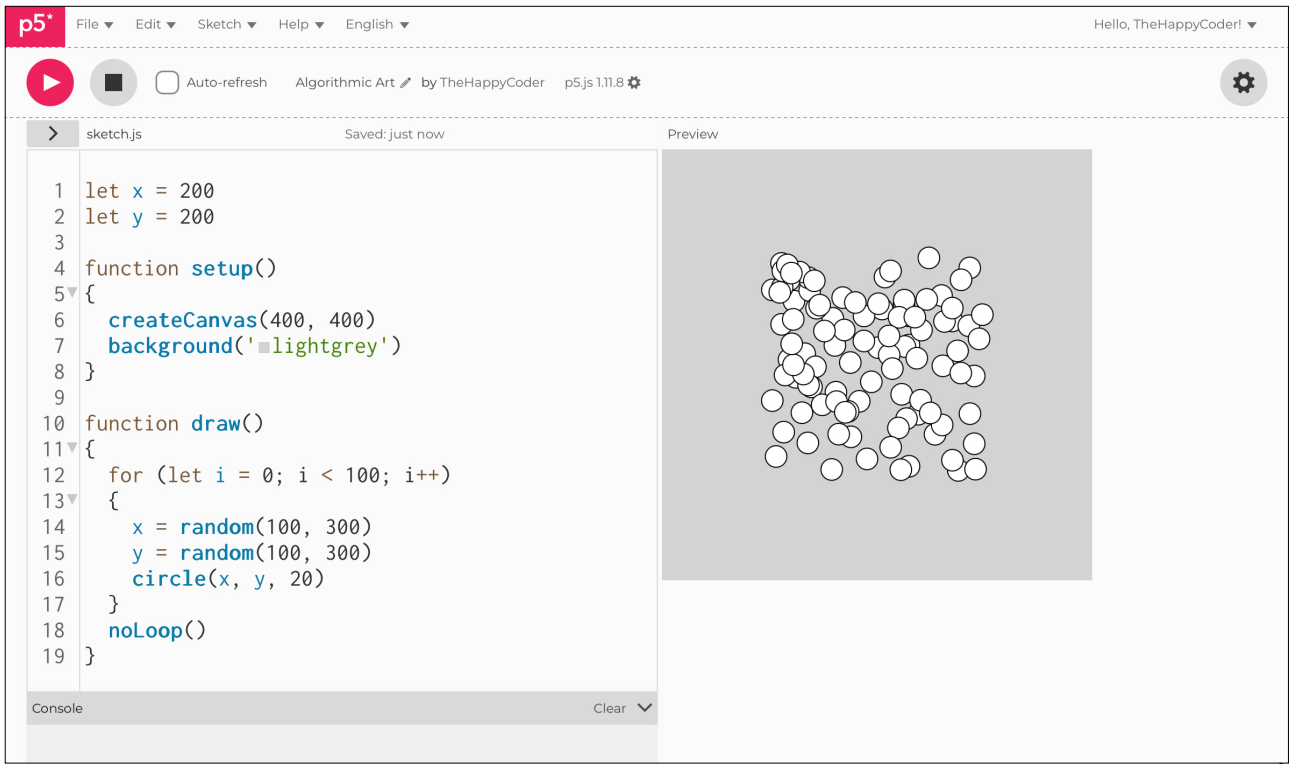
Challenges

1. Change the number from 100 to either 10 or 1000.
2. Take out the `noLoop()` function.

Code Explanation

`for()`	This is the for() loop; it will count to 10 and stop (actually will then start counting again). The draw function is a loop, so it is a loop within a loop.
`let i = 0`	This declares a variable called i and initialises it to 0.
`i < 100`	This bit checks to see if i is still less than 100.
`i++`	This adds 1 to the i variable every time it does a loop.
`noLoop()`	Stops the draw() loop once the condition has been met.

Figure A2.12


p5*
File
Edit
Sketch
Help
English
Hello, TheHappyCoder!
Auto-refresh
Algorithmic Art
by TheHappyCoder
p5.js 1.11.8
sketch.js
Saved: just now
Preview
let x = 200
let y = 200
function setup()
{
createCanvas(400, 400)
background('lightgrey')
}
function draw()
{
for (let i = 0; i < 100; i++)
{
x = random(100, 300)
y = random(100, 300)
circle(x, y, 20)
}
noLoop()
}
Console
Clear

The Joy of Coding Algorithmic Art Module A Unit #3 adding RGB

Module A Unit #3: adding RGB

This unit looks at RGB colour, by that we mean red, green, and blue. This is how you can create a wide range of colours by mixing these primary colours; in fact, you can create over 16 million colours.

Think in terms of light rather than paints. If you have all the red, all the green, and all the blue, you get white; if you have none of them, you get black, and every combination in between. The amount of a single colour is between `0` - `256`.

Key concepts:

`RGB colour`
`transparency`
`noStroke()`
`stroke()`
`strokeWeight()`

RGB colour

We can represent the colour not just in words— red, green, or blue —but also using their RGB values. Each colour has some red, some green, and some blue. One reason for using this system of creating the colours is that we can manipulate the values mathematically. For instance, the following colours are given as RGB values. See Fig. 1 below.

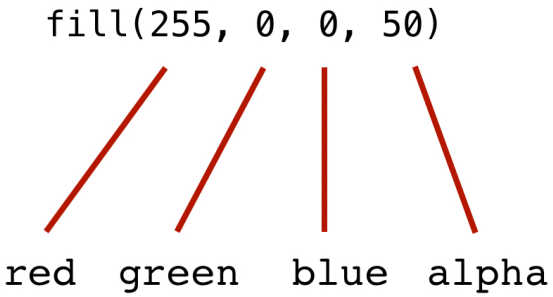
`fill(255, 0, 0)`	This gives us red with no green and no blue.
`fill(0, 255, 0)`	This gives us green with no red or blue.
`fill(0, 0, 255)`	This gives us blue with no red or green.
`fill(255, 255, 255)`	This gives us white, the same as `fill(255)`.
`fill(0, 0, 0)`	This gives us black, same as `fill(0)`.

RGB transparency

You can create any colour by mixing the numbers as you do with mixing paint or light. The three numbers are called `arguments`; you can add a fourth argument, which is the `alpha` or transparency value. The range is between `0` and `255`. See `Fig. 1` below.

`fill(255, 0, 0, 255)`	This gives us no transparency at all; even though it should be red, it will appear opaque.
`fill(255, 0, 0, 0)`	This gives us no transparency at all; even though it should be red, it will appear completely transparent.
`fill(255, 0, 0, 50)`	Gives us some red, but you can still see through it.

Figure 1: fill() function



Sketch A3.1 RGB

! Start a new sketch

Drawing three overlapping circles, each with a separate colour. In the previous unit, we used names for the colours used. The background was called `'lightgrey'`. That is one way of using colours in your sketch; alternatively, we can use numerical values. Here we are using the RGB values for the background and the colours of the circles. I have highlighted the code in blue. Notice that I have given the `background()` a value of `220`, which is a single colour value, meaning it is either black (value of `0`) or white (value of `255`) and everything in between as shades of grey.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  fill(255, 0, 0)
  circle(175, 150, 100)
  fill(0, 255, 0)
  circle(225, 150, 100)
  fill(0, 0, 255)
  circle(200, 200, 100)
}
```

Notes

Each circle will have its own colour fill.

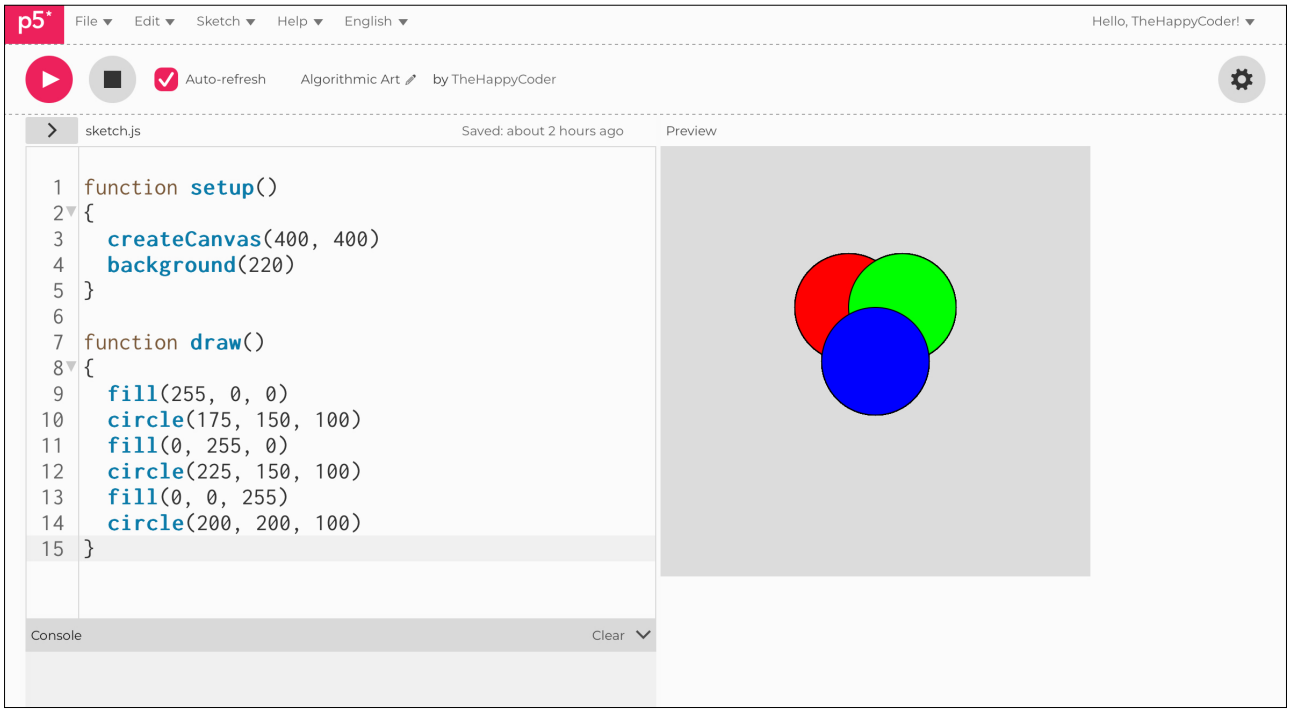
Challenge

Remove one of the `fill()` functions and see what happens.

Code Explanation

`background(220)`	The background has a grey colour.
`fill(255, 0, 0)`	This gives you a red-coloured circle, where the red value = 255, green value = 0, and blue has a value = 0.
`fill(0, 255, 0)`	This gives you a green-coloured circle, where the red value = 0, green value = 255, and blue has a value = 0.
`fill(0, 0, 255)`	This gives you a blue-coloured circle, where the red value = 0, green value = 0, and blue has a value = 255.

Figure A3.1



Sketch A3.2 alpha

Adding a bit of alpha by adding a fourth argument, we give the circles an `alpha` value of `50`.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  fill(255, 0, 0, 50)
  circle(175, 150, 100)
  fill(0, 255, 0, 50)
  circle(225, 150, 100)
  fill(0, 0, 255, 50)
  circle(200, 200, 100)
}
```

Notes

The overlapping circles are now more translucent, with the help of some alpha.

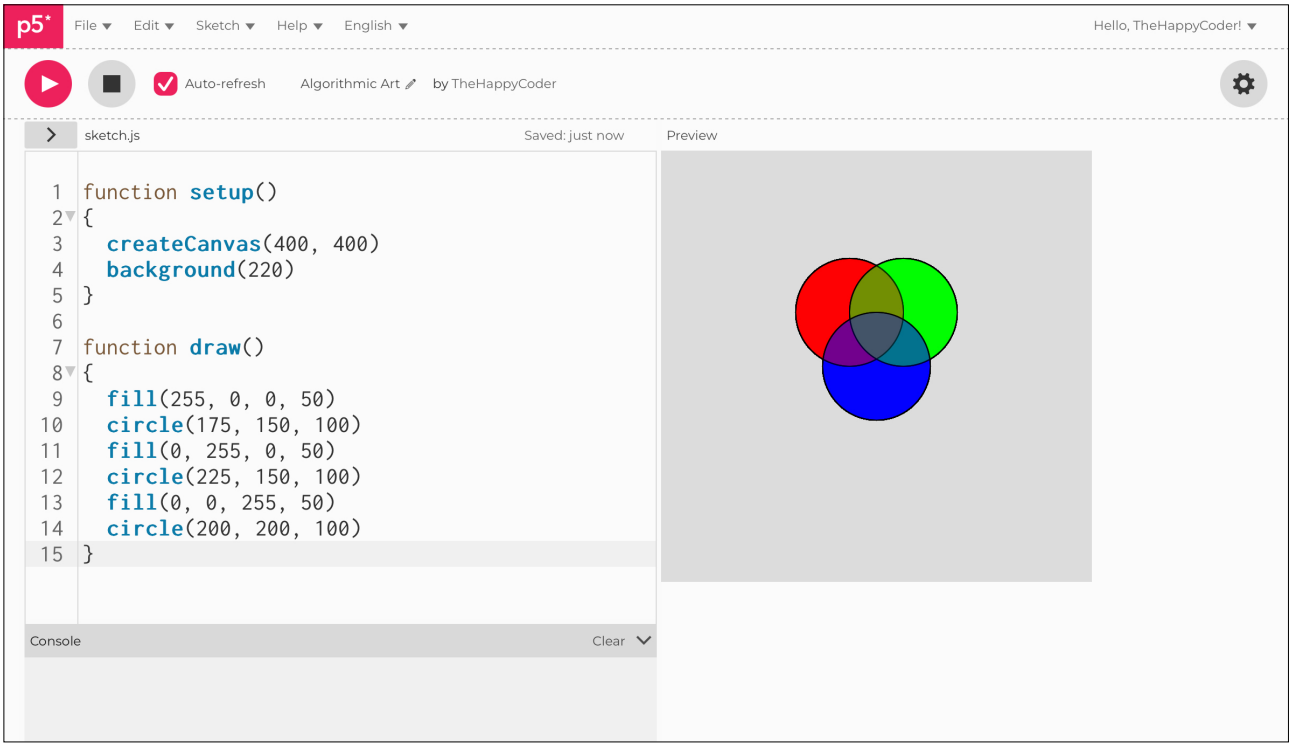
Challenge

Change the values of alpha.

Code Explanation

`fill(255, 0, 0, 50)`	Red with plenty of transparency.
`fill(0, 255, 0, 50)`	Green with plenty of transparency.
`fill(0, 0, 255, 50)`	Blue with plenty of transparency.

Figure A3.2



Sketch A3.3 random colour alpha value

We are now going to go mad with the variables, making variables of nearly everything. The alpha is the fourth argument for colour and it is the amount of transparency from 0 (transparent) to 255 (opaque).

```
let x = 0
let y = 0
let r = 0
let g = 0
let b = 0
let a = 0
let d = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  x = random(400)
  y = random(400)
  r = random(255)
  g = random(255)
  b = random(255)
  a = random(255)
  d = random(100)
  fill(r, g, b, a)
  circle(x, y, d)
}
```

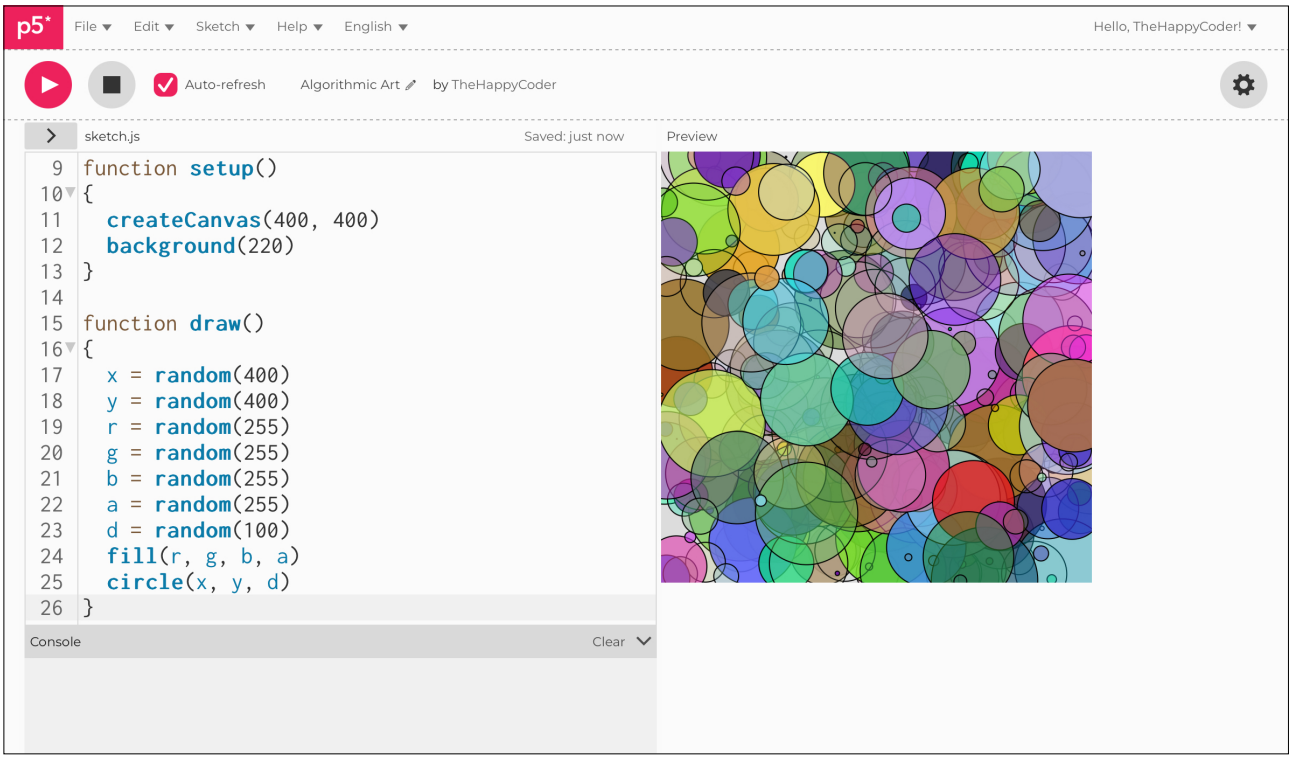
Notes

You wouldn't generally use single letters to name variables, but in this case, it is simple enough and intuitive.

Code Explanation

`x = random(400)`	Random x position of circle.
`y = random(400)`	Random y position of circle.
`r = random(255)`	Random red value.
`g = random(255)`	Random green value.
`b = random(255)`	Random blue value.
`a = random(255)`	Random alpha.
`d = random(100)`	Random diameter.
`fill(r, g, b, a)`	Fill each circle with one set of random values.
`circle(x, y, d)`	Draw a circle on each loop of the draw() function, at random positions and diameters.

Figure A3.3



Sketch A3.4 no stroke

We can remove the line around the circle with the function `noStroke()`.

```
let x = 0
let y = 0
let r = 0
let g = 0
let b = 0
let a = 0
let d = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  noStroke()
  x = random(400)
  y = random(400)
  r = random(255)
  g = random(255)
  b = random(255)
  a = random(255)
  d = random(100)
  fill(r, g, b, a)
  circle(x, y, d)
}
```

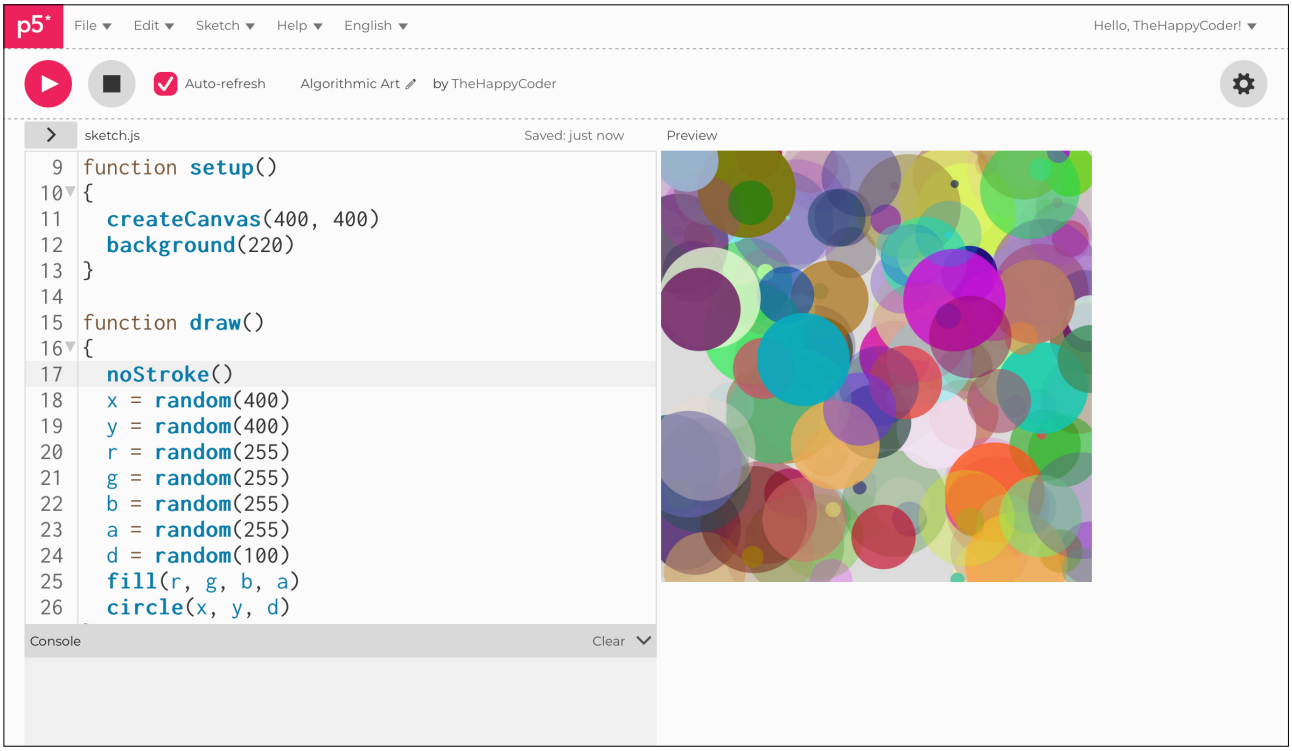
Notes

Having `noStroke()` and `noFill()` will effectively remove the circle from the canvas.

Code Explanation

`noStroke()`	Removes any lines around shapes.

Figure A3.4



Sketch A3.5 the weight of the stroke

! Remove `noStroke()` and replace with `strokeWeight()`

We can alter the thickness of the lines with the function `strokeWeight()`, the default is a thickness of `1` pixel, so you could use any value you like, but I recommend only using a value up to `5`.

```
let x = 0
let y = 0
let r = 0
let g = 0
let b = 0
let a = 0
let d = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  strokeWeight(random(5))
  x = random(400)
  y = random(400)
  r = random(255)
  g = random(255)
  b = random(255)
  a = random(255)
  d = random(100)
  fill(r, g, b, a)
  circle(x, y, d)
}
```

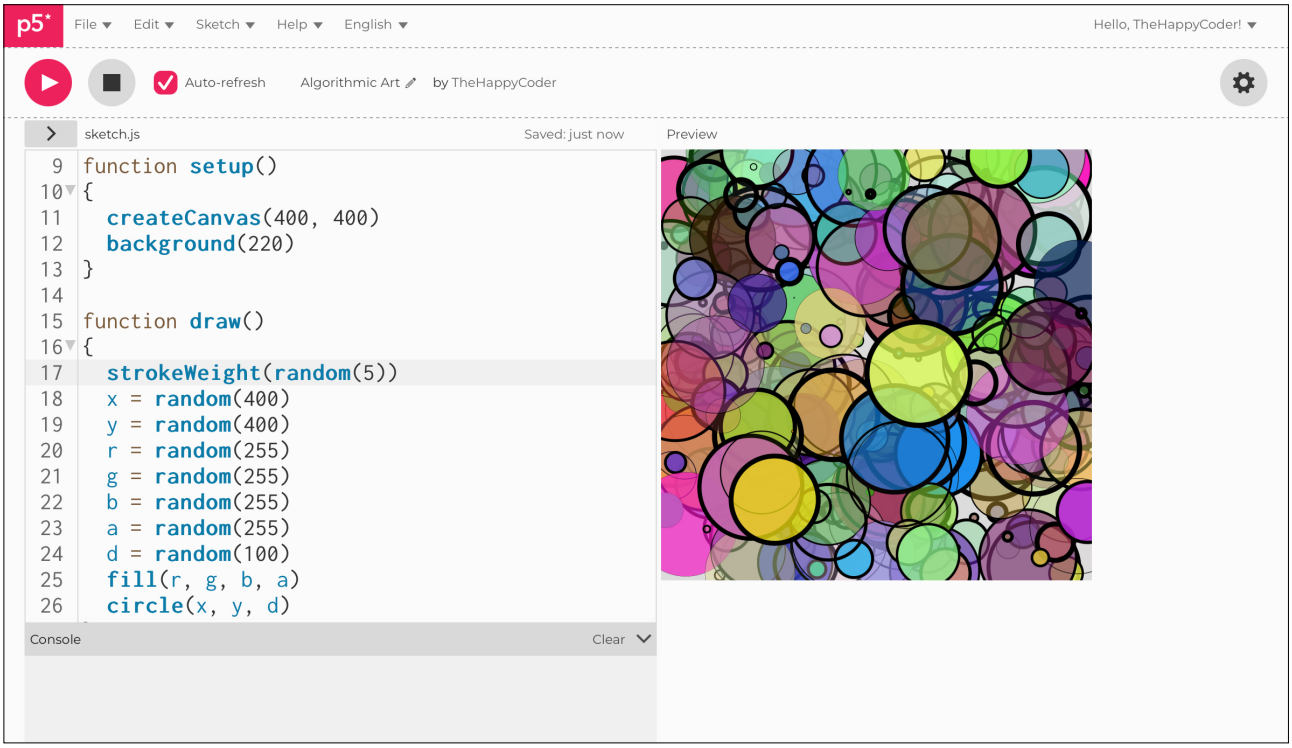
Challenge

Try different ranges of random `strokeWeight()`.

Code Explanation

`strokeWeight(random(5))`	We can put the random function inside the brackets rather than create another variable.

Figure A3.5



Sketch A3.6 colouring the lines

! Replace `strokeWeight()` with a function called `stroke()`

We can also add colour (even add alpha) to the lines using `stroke()`, it works exactly the same as the `fill()` function.

```
let x = 0
let y = 0
let r = 0
let g = 0
let b = 0
let a = 0
let d = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  stroke(r, g, b)
  x = random(400)
  y = random(400)
  r = random(255)
  g = random(255)
  b = random(255)
  a = random(255)
  d = random(100)
  fill(r, g, b, a)
  circle(x, y, d)
}
```

Notes

The colour of the lines is the colour of the last `fill(r, g, b)` circle drawn.

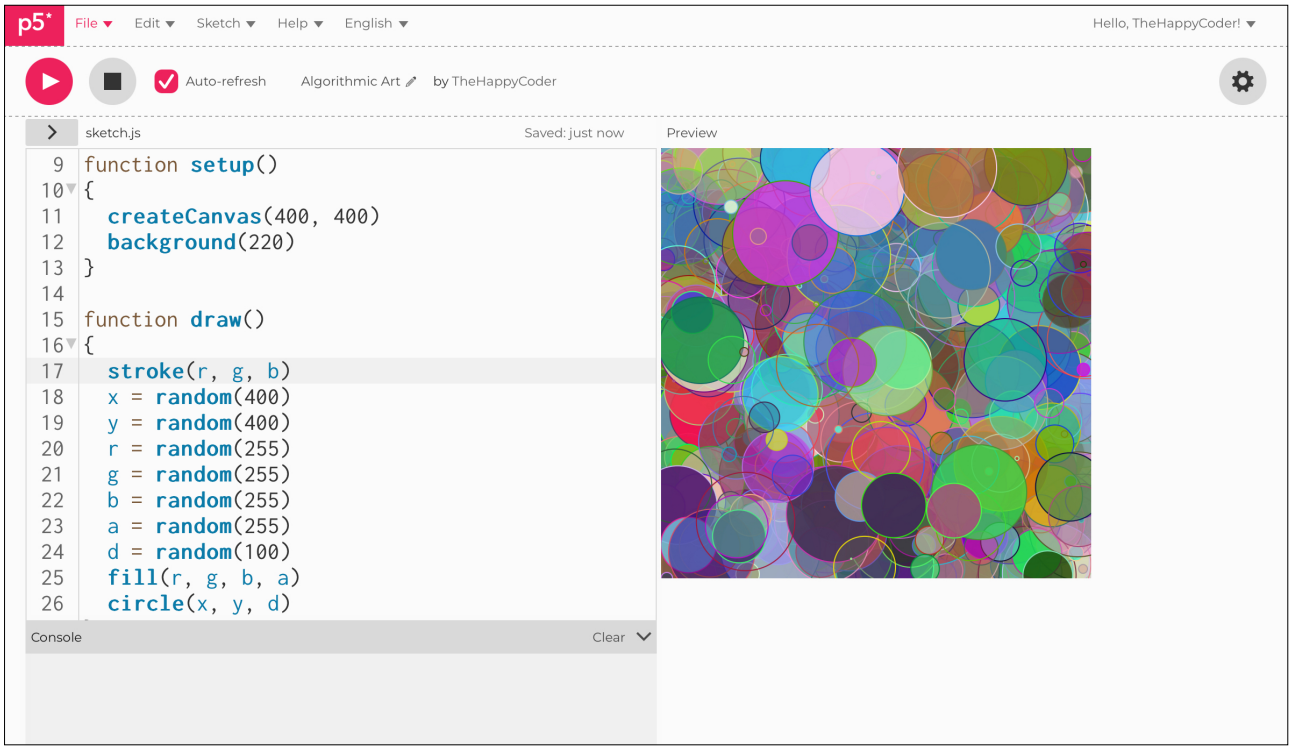
Challenges

1. Add random strokeWeight() as well.
2. Move `stroke(r, g, b)` to the line just before `fill(r, g, b, a)`.

Code Explanation

`stroke(r, g, b)`	As with the fill() function, we can give it up to four arguments: red, green, blue, and alpha.

Figure A3.6



The Joy of Coding Algorithmic Art Module A Unit #4 lots of lines

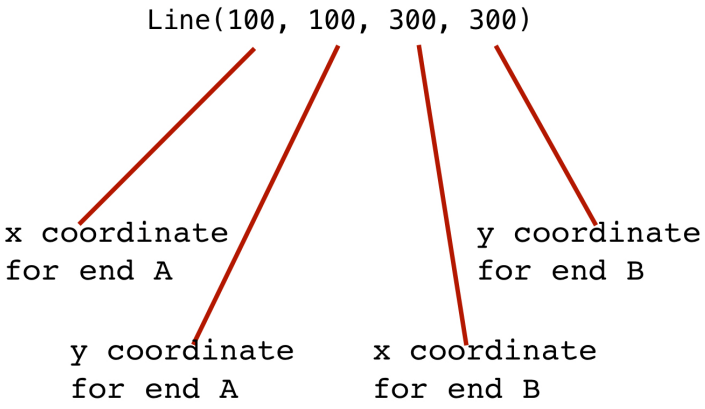
Module A Unit #4: lots of lines

We have drawn circles now to introduce lines. To draw a line, we need four arguments; these are two sets of coordinates. A set for each end of the line; it is that simple. If we have an end A and an end B for the line() function, it will look like below in Fig. 1. This draws a line between the coordinates (100, 100) and (300, 300).

Key concepts:

line() function
if() statement
noLoop()

Figure 1: line() function



Sketch A4.1 lines

! start a new sketch

We can draw lots of other shapes, but in this section, we will draw lines and create simple image using just lines. To reiterate, the line() function has four arguments: the first two are the x and y co-ordinates of one end of the line, and the other two arguments are the x and y co-ordinates of the other end of the line.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  line(100, 100, 300, 300)
}
```

Notes

Not a very exciting creation but there is plenty we can do with it.

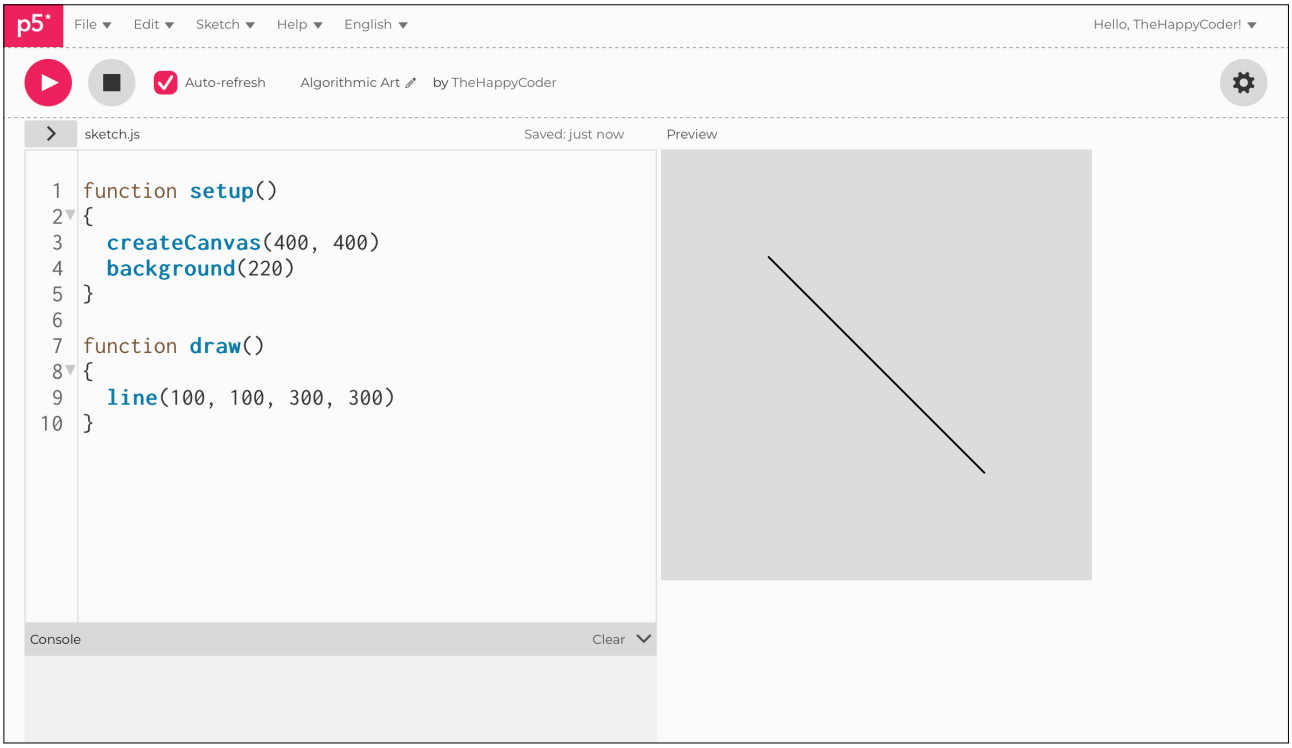
Challenge

Try other values for the coordinates

Code Explanation

`line(100, 100, 300, 300)`	Drawing a line from the top left (100, 100) to the bottom right (300, 300).

Figure A4.1



Sketch A4.2 a row of lines

Here, we are going to draw lots of vertical lines `10` pixels apart. We first create a variable for the `x` component of the coordinates and put that in the `line()` function. We want the same `x` value for each end.

```
let x = 10

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  line(x, 100, x, 300)
  x += 10
}
```

Notes

To add 10 each time we have used an abbreviation (+=). We could have written it long hand like so: `x = x + 10`. We have cheated a little because the programme is still drawing the lines; it never stops, which is never a great idea. In the next bit, we will use a `while()` loop to stop it.

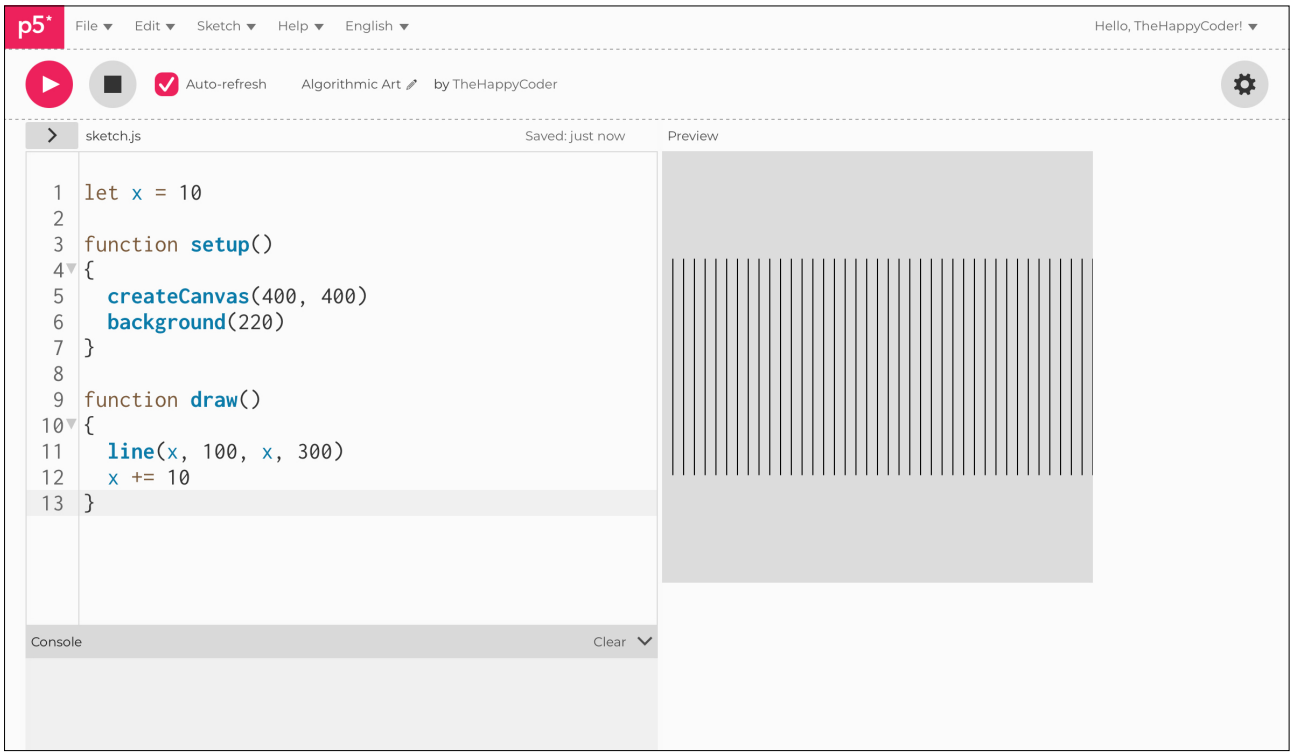
Challenge

Try a different spacing value other than `10`

Code Explanation

`let x = 10`	Create a variable for x and give it a starting value of 10.
`line(x, 100, x, 300)`	Draws a line with the same x value which starts at 10 and increases by 10; the y values remain the same for each line.
`x += 10`	Adds 10 each time. We could write `x = x + 10`.

Figure A4.2



Sketch A4.3 limiting lines

Here we have a `while()` loop. In this example, the lines are drawn while `x` is less than `400`. When `x` reaches `400`, then the loop stops. The loop only works while `x` is less than (`<`) `400`. To put the code inside the curly brackets, you can just cut and paste.

```
let x = 10

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  while (x < 400)
  {
    line(x, 100, x, 300)
    x += 10
  }
}
```

Notes

The `while()` loop uses a conditional statement (<). While `x` is less than `400`, the condition is `true` and it draws the line, adds `10` to `x`, and repeats. The condition is considered to be `false` when the value of `x` is `400` or above (and the function stops).

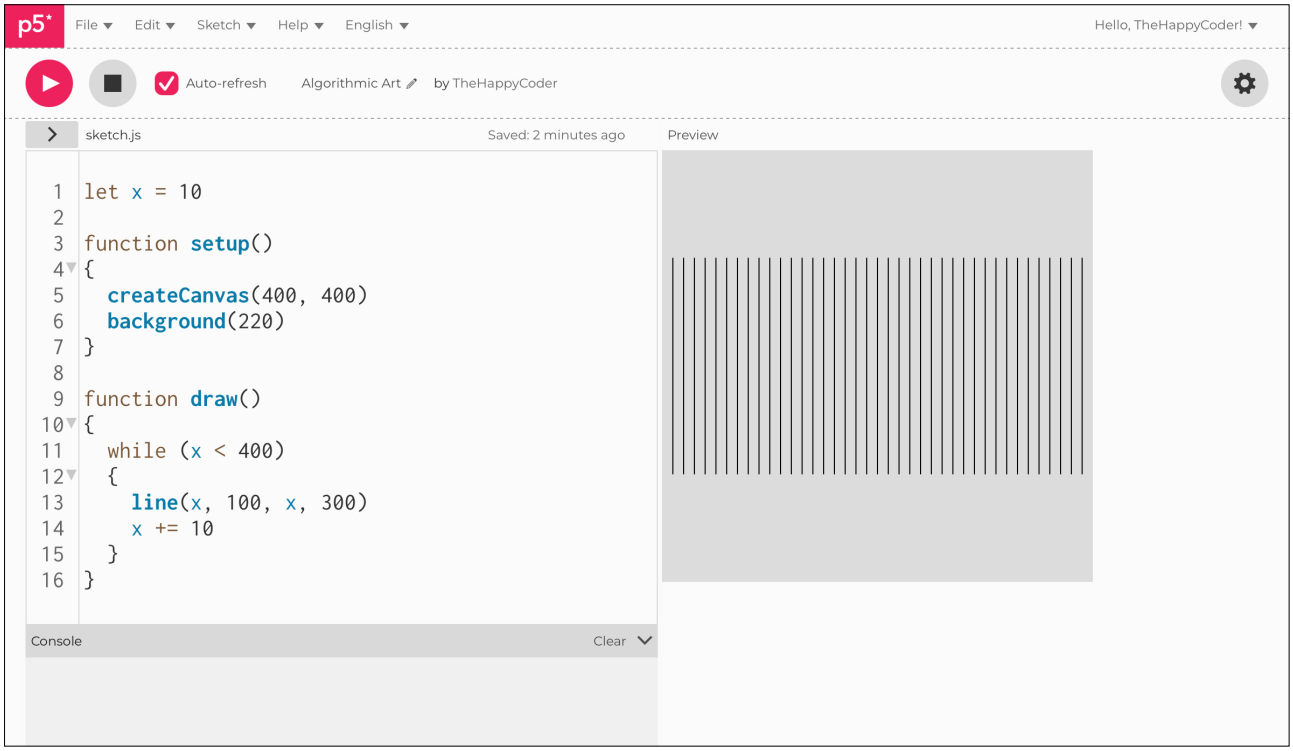
Challenges

Change the conditional value to `100`

Code Explanation

`while (x < 400)`	It checks to see if x has exceeded 400.

Figure A4.3



Sketch A4.4 colour and thickness

Adding some thickness and colour to the lines.

```
let x = 10

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  strokeWeight(5)
  stroke(200, 0, 0)
  while (x < 400)
  {
    line(x, 100, x, 300)
    x += 10
  }
}
```

Notes

We can give the line extra weight (thickness) as well as colour. This means we can manipulate the lines in many creative and interesting ways.

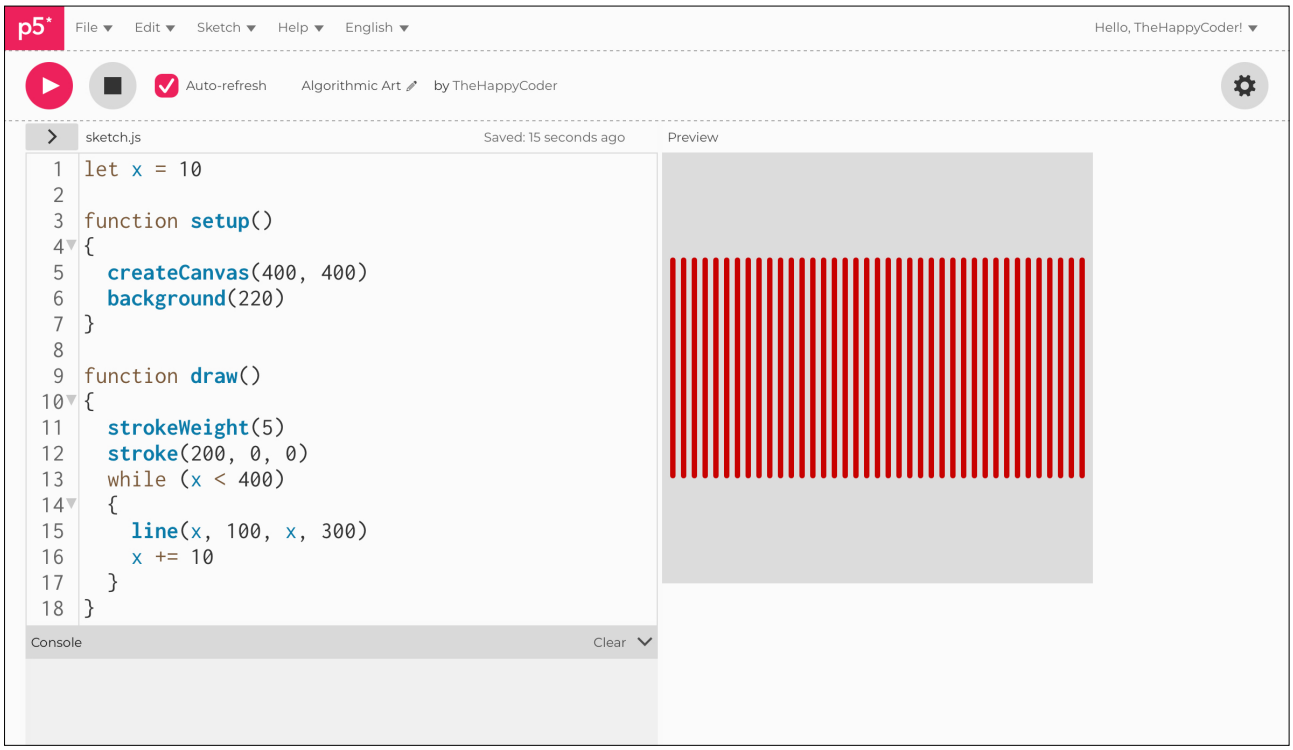
Challenges

1. Try alpha as well
2. Make it full height

Code Explanation

`strokeWeight(5)`	Determines how thick the line is.
`stroke(200, 0, 0)`	Determines the colour of the line.

Figure A4.4



Sketch A4.5 incrementing the colour

We can introduce another variable to reduce the amount of red. We will call this `increment` and subtract it from the initial value of `200` in steps of `5`.

! we move the `stroke()` function inside the `while()` loop

```
let x = 10
let increment = 5

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  strokeWeight(5)
  while (x < 400)
  {
    stroke(200 - increment, 0, 0)
    line(x, 100, x, 300)
    x += 10
    increment += 5
  }
}
```

Notes

We are decreasing the red element of the RGB by 5 on each iteration inside the `while()` loop, thus making it darker.

Challenge

1. Can you think of another way to do it (hint: use `-=` instead)
2. Try to achieve the opposite, from dark red to a light red

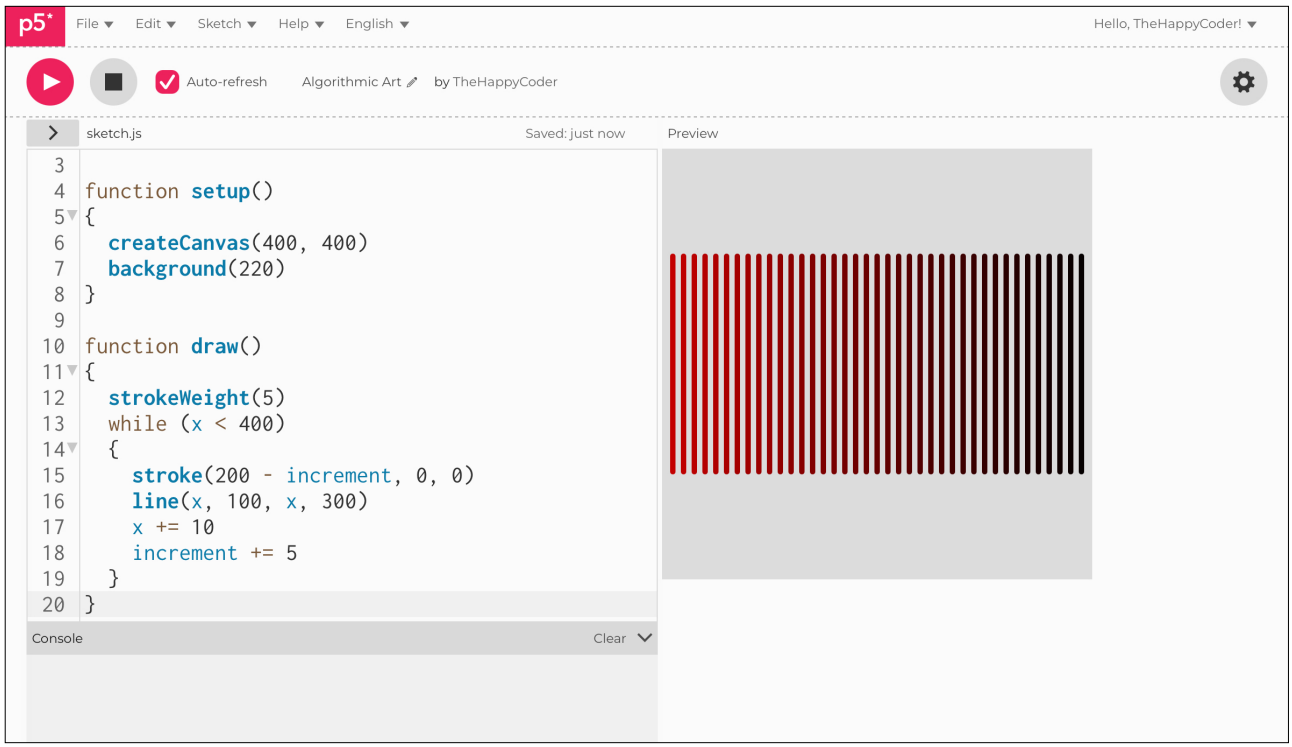
Code Explanation

`let increment = 5`	Create a variable called increment.
`stroke(200 - increment, 0, 0)`	Reduce the amount of red by that amount.
`increment += 5`	Keep reducing the red with each iteration of the loop.

Hint solution:

```
let x = 10
let increment = 200

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  strokeWeight(5)
  while (x < 400)
  {
    stroke(increment, 0, 0)
    line(x, 100, x, 300)
    x += 10
    increment -= 5
  }
}
```

Figure A4.5


p5*
File
Edit
Sketch
Help
English
Hello, TheHappyCoder!
Auto-refresh
Algorithmic Art
by TheHappyCoder
sketch.js
Saved: just now
Preview
3
4 function setup()
5 {
6 createCanvas(400, 400)
7 background(220)
8 }
9
10 function draw()
11 {
12 strokeWeight(5)
13 while (x < 400)
14 {
15 stroke(200 - increment, 0, 0)
16 line(x, 100, x, 300)
17 x += 10
18 increment += 5
19 }
20 }
Console
Clear

Sketch A4.6 more blue

Let's add the increment to the amount of blue as the red value decreases.

```
let x = 10
let increment = 5

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  strokeWeight(5)
  while (x < 400)
  {
    stroke(200 - increment, 0, increment)
    line(x, 100, x, 300)
    x += 10
    increment += 5
  }
}
```

Notes

The power of variables

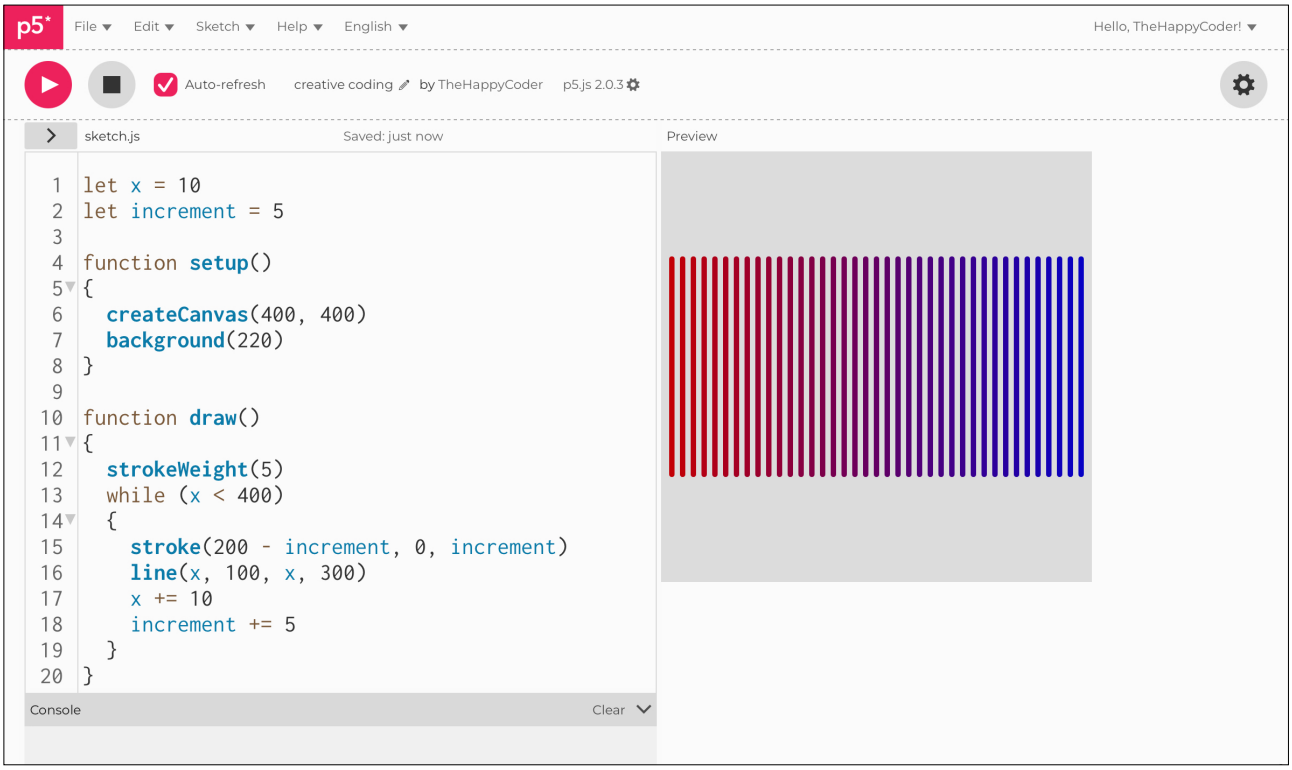
Challenges

1. Try other colours
2. Try alpha

Code Explanation

`stroke(200 - increment, 0, increment)`	Decreases the red and increases the blue at the same time.

Figure A4.6



Sketch A4.7 random colour and other stuff

We can start to play around a bit. Notice that you don't always have to create variables when using random. Where appropriate, you can put them straight into the function.

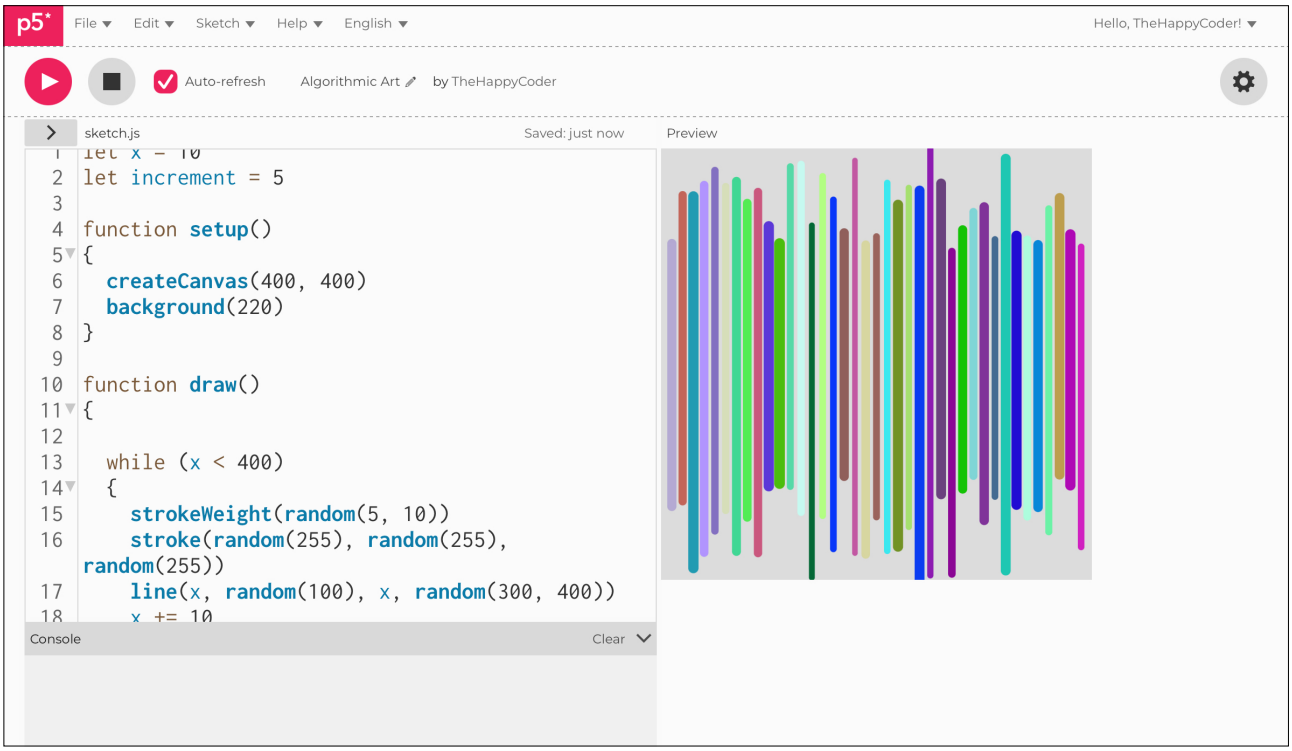
! Move the `strokeWeight()` into the `while()` loop and randomise it, also notice we aren't using `increment` anymore.

```
let x = 10
let increment = 5

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  while (x < 400)
  {
    strokeWeight(random(5, 10))
    stroke(random(255), random(255), random(255))
    line(x, random(100), x, random(300, 400))
    x += 10
    increment += 5
  }
}
```

Notes

The code should be fairly obvious to you now, creates a pleasing effect

Figure A4.7



Sketch A4.8 newish sketch

! start a new sketch with additional lines highlighted.

Here we have a faint line drawn randomly from the left side to the right side.

```
function setup()
{
  createCanvas(400, 400)
  background(255)
}

function draw()
{
  strokeWeight(0.1)
  line(0, random(400), 400, random(400))
}
```

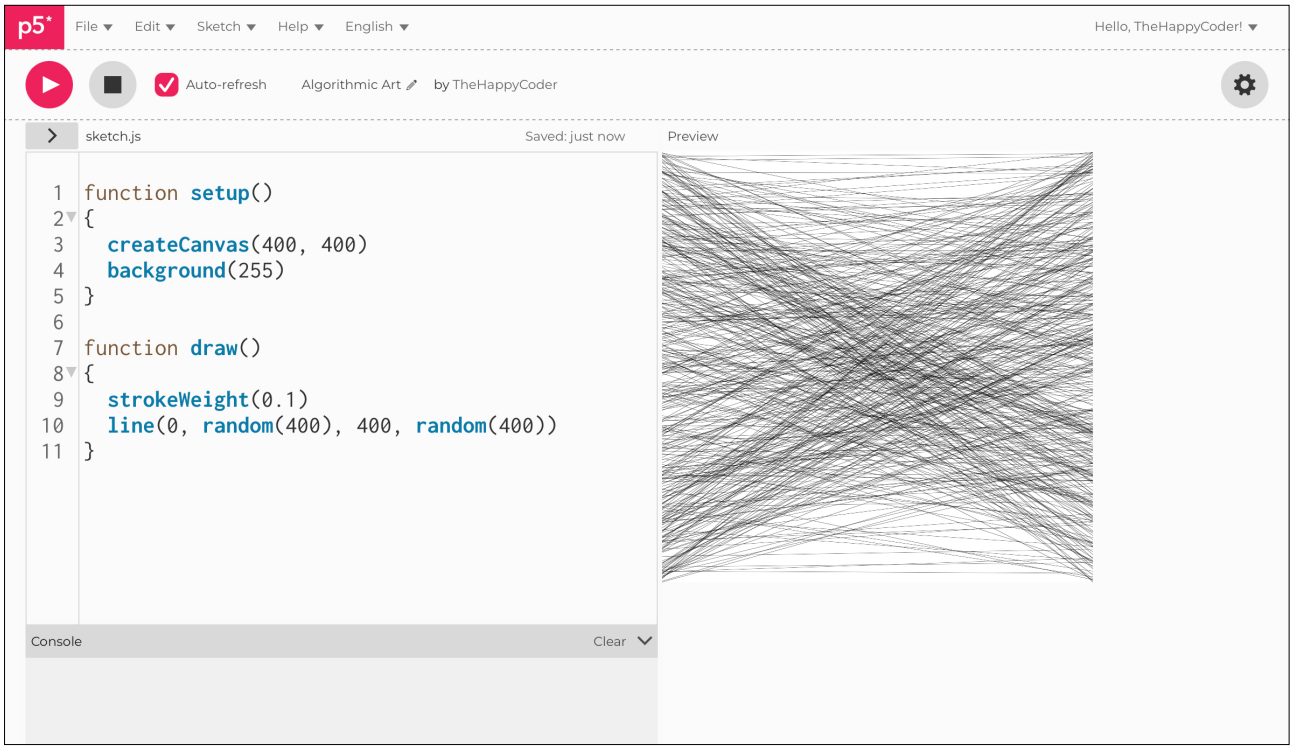
Notes

This will continue infinitum

Code Explanation

`strokeWeight(0.1)`	Gives a very thin weight to the line.

Figure A4.8



Sketch A4.9 more random lines

Let's add another random line generator. This time, top to bottom.

```
function setup()
{
  createCanvas(400, 400)
  background(255)
}

function draw()
{
  strokeWeight(0.1)
  line(0, random(400), 400, random(400))
  line(random(400), 0, random(400), 400)
}
```

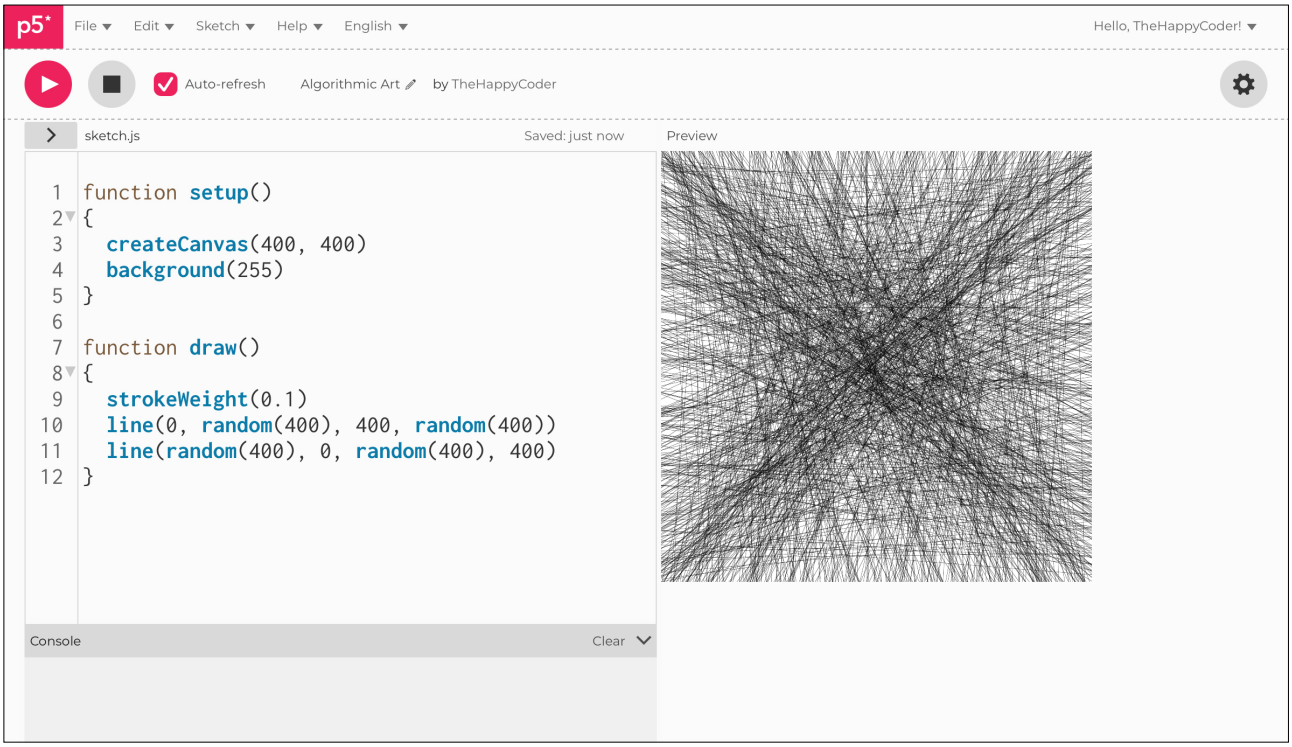
Notes

The mesh of lines fills up the canvas rapidly. What we would like to do is stop it before it is completely blacked out.

Challenge

Can you think of a way to stop it after, say, 300 lines?

Figure A4.9



Sketch A4.10 we need to count the lines

We can create a random mesh. First, let's create a variable called `count` to count the number of lines.

```
let count = 0

function setup()
{
  createCanvas(400, 400)
  background(255)
}

function draw()
{
  strokeWeight(0.1)
  line(0, random(400), 400, random(400))
  line(random(400), 0, random(400), 400)
  count++
}
```

Notes

This does nothing except count the lines (two lines for every `count` increase) on each iteration of the `draw()` loop.

Sketch A4.11 if() statement

Another solution to limiting the number of lines is to use something called an `if()` statement. It has a condition attached; in this case, it is the `==` sign, which means `if it is equal to`. When that condition is `true`, then it activates the code inside the curly brackets `{...}`. In this instance, we want to stop drawing any more lines, so we use a function called `noLoop()`, which is pretty obvious what it does.

```
let count = 0

function setup()
{
  createCanvas(400, 400)
  background(255)
}

function draw()
{
  strokeWeight(0.1)
  line(0, random(400), 400, random(400))
  line(random(400), 0, random(400), 400)
  count++
  if (count == 300)
  {
    noLoop()
  }
}
```

Notes

The `if()` statement is similar to the `while()` loop; it is used more often than the `while()` loop. You can string lots of `if()` statements together and have what is called `if()...else()` statements (more on that later).

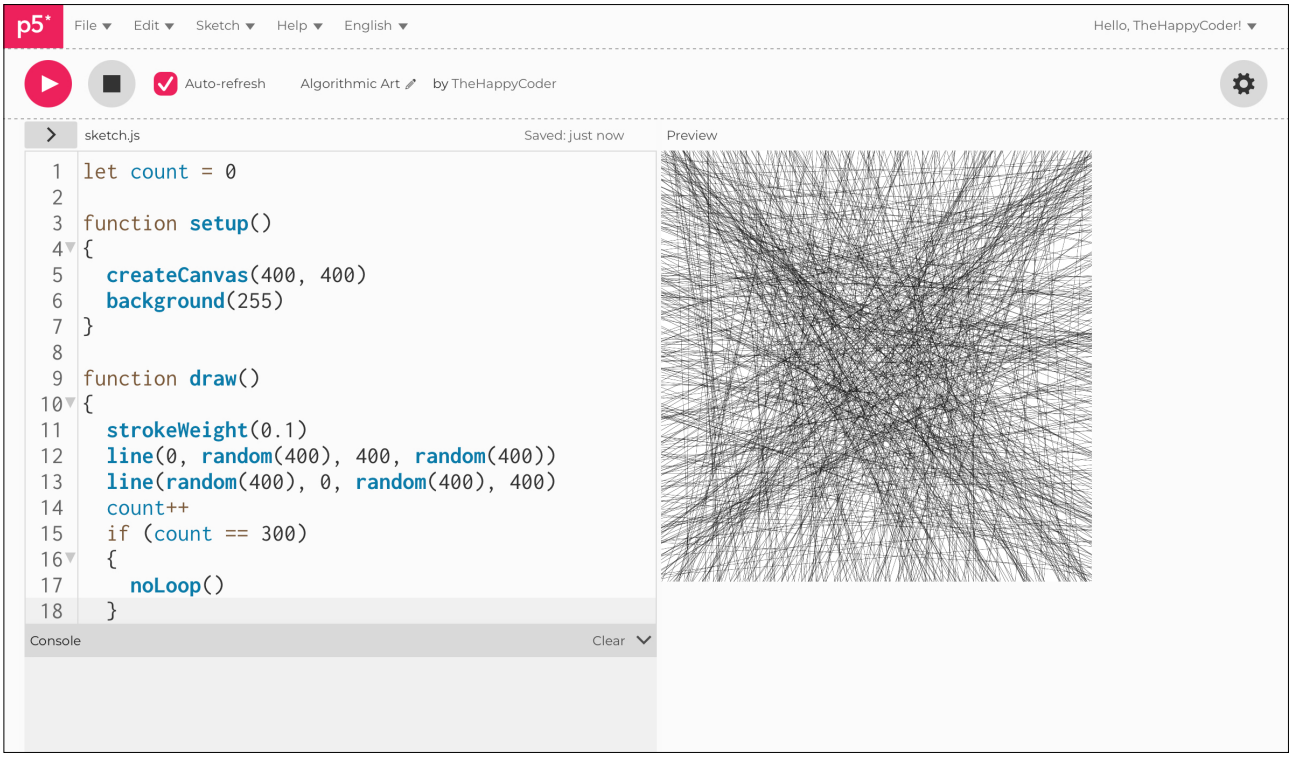
Challenges

1. Change the number of lines.
2. Change the weight and add alpha instead.
3. Have different colours.

Code Explanation

`if (count == 300)`	Checks to see if this condition is `true`.
`noLoop()`	If it is `true`, then it does this.

Figure A4.11



The Joy of Coding Algorithmic Art Module A Unit #5 squares & rectangles

Module A Unit #5: squares and rectangles

There is a lot to pack into this unit. Here we introduce the square and rectangle and show how we can manipulate these shapes, rotating and translating them. I will be introducing a lot of new concepts to you, but they will be used a lot in the units and examples to come. Just plough your way through the sketches and play with the code; this approach will help you understand what it is doing. I will give you the framework, but it is up to you to explore.

Key concepts:

```
square()
rotate()
angleMode()
translate()
push()
pop()
rectMode()
rectangle()
comments
```

Sketch A5.1 a simple square

! Our starting sketch

We can draw a simple square in the middle of the canvas. The `square()` function has three arguments: the x co-ordinate, the y co-ordinate and the length of the sides.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  square(200, 200, 100)
}
```

Notes

The only problem is, it isn't in the centre of the canvas. This is because the co-ordinates of the square are taken from the top left-hand corner, not the centre of the square, but we can fix that quite easily.

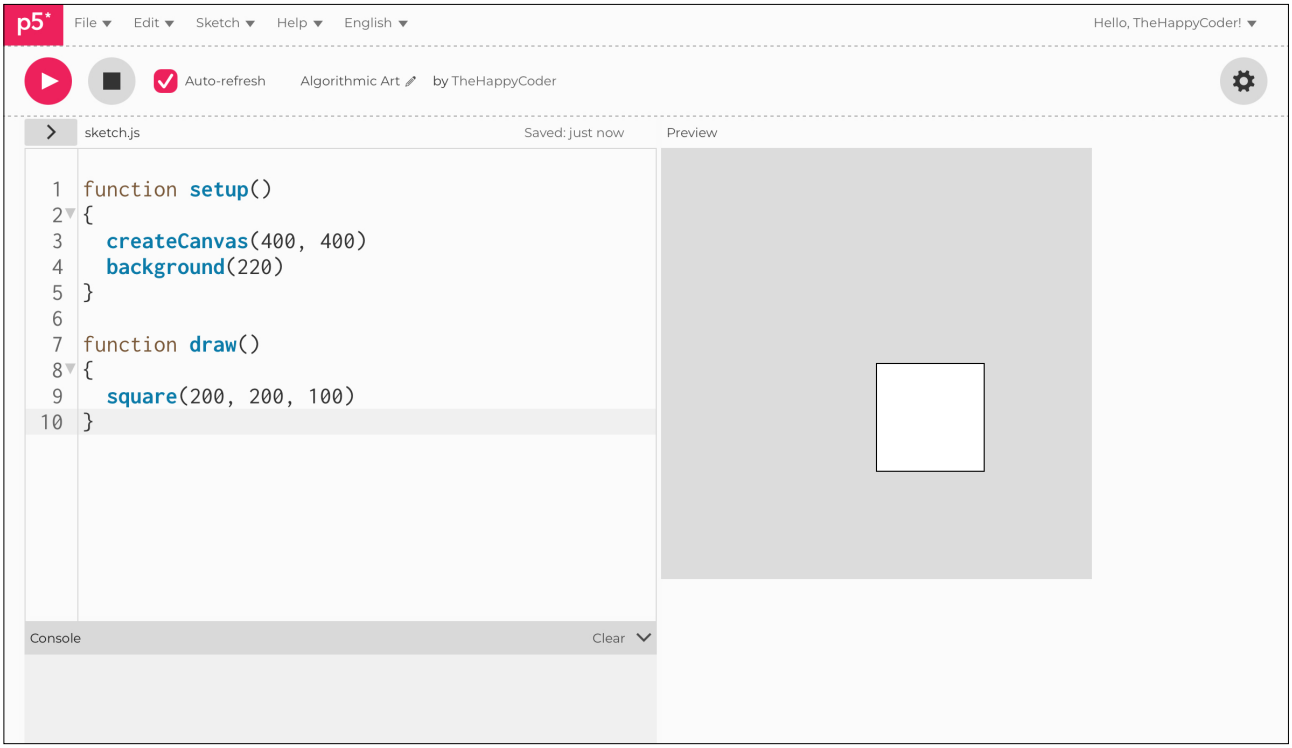
Challenges

1. What would be the co-ordinates to put the square in the centre of the canvas?
2. Draw more squares.

Code Explanation

`square(200, 200, 100)`	Draws a square at (200, 200) with a side length of 100 pixels.

Figure A5.1



Sketch A5.2 from the centre

Using a function called `rectMode()`, we can move the co-ordinates to the centre.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  rectMode(CENTER)
}

function draw()
{
  square(200, 200, 100)
}
```

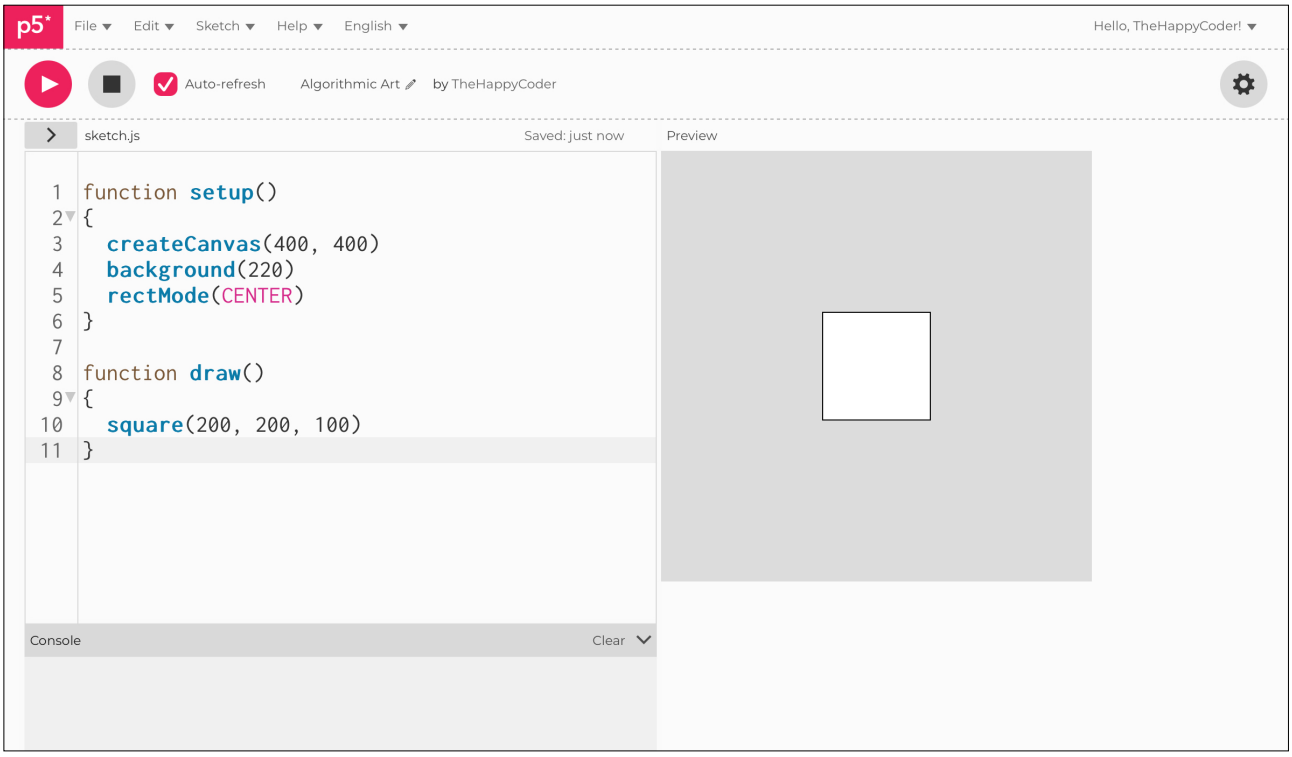
Notes

Now it is nicely in the centre!

Code Explanation

`rectMode(CENTER)`	This function needs the information in the brackets (which needs to be uppercase and US spelling).

Figure A5.2



Sketch A5.3 rotate

We can rotate the square now because we have the centre in the centre, if you get my meaning; otherwise, it would rotate around the corner. By default, the angle of rotation is measured in radians; however, we can change that to degrees with angleMode().

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  rectMode(CENTER)
  angleMode(DEGREES)
}

function draw()
{
  rotate(45)
  square(200, 200, 100)
}
```

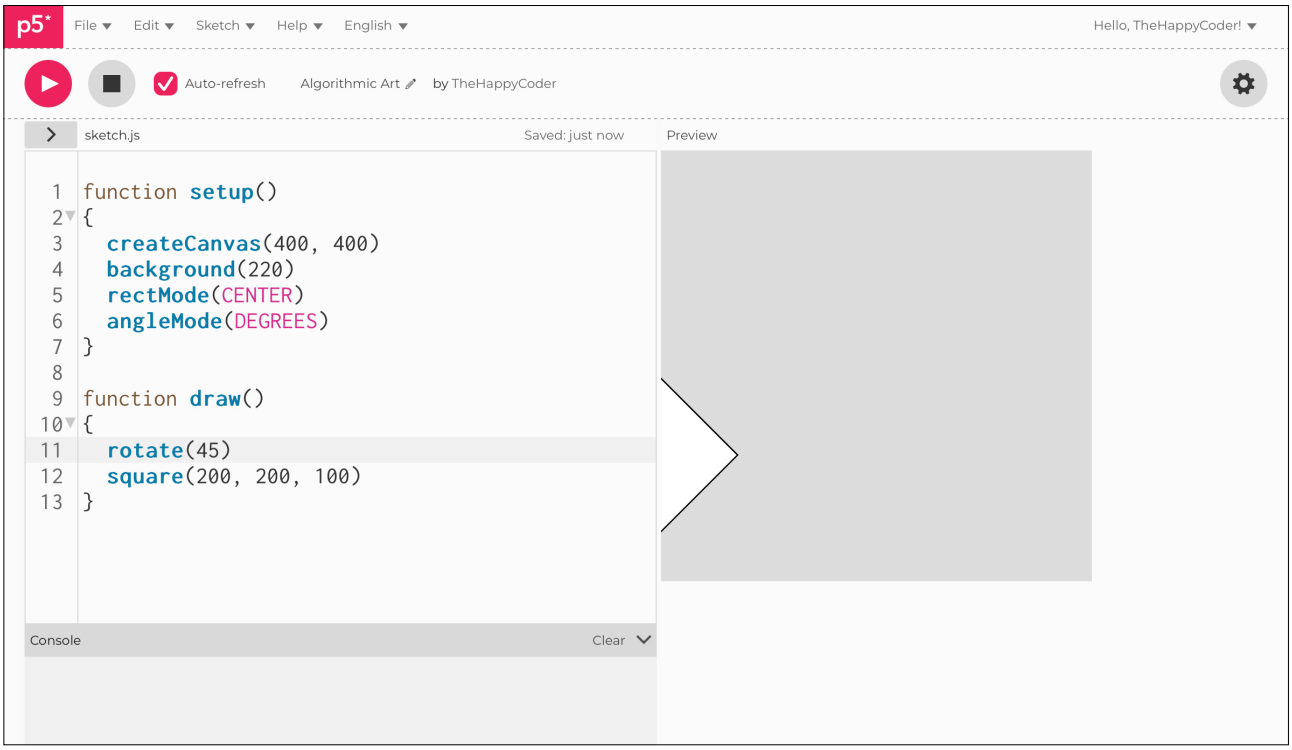
Notes

As you will see, there is a problem. It has rotated about the corner of the canvas, not the centre of the square. The good news is we have a solution!

Code Explanation

`angleMode(DEGREES)`	Converts from radians to degrees.
`rotate(45)`	Rotate by 45°.

Figure A5.3



Sketch A5.4 translate

In order to solve this problem, we have to move the origin of the canvas from the top-left-hand corner to the centre of the canvas. To do this, we use the `translate()` function to move it. We then have to change the co-ordinates of the square to `(0, 0)`.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  rectMode(CENTER)
  angleMode(DEGREES)
}

function draw()
{
  translate(200, 200)
  rotate(45)
  square(0, 0, 100)
}
```

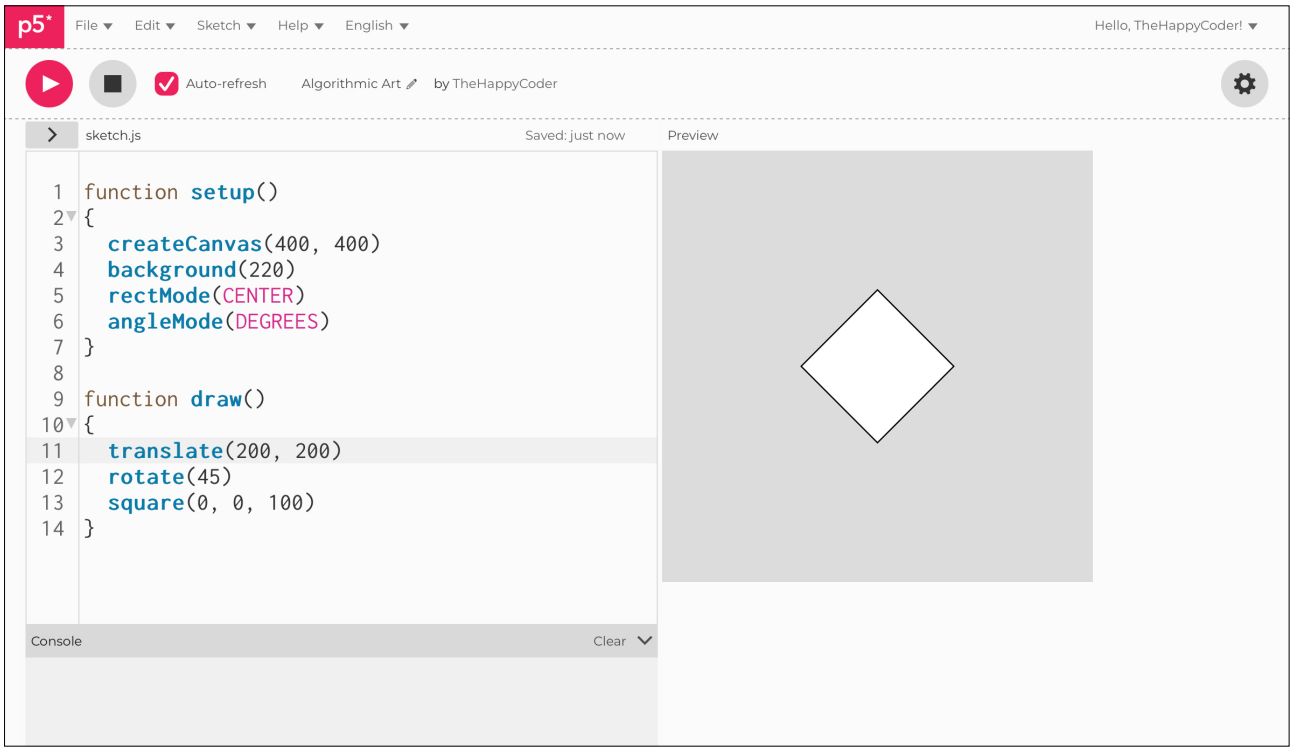
Notes

It is now rotating around the centre of the square.

Code Explanation

`translate(200, 200)`	Moves the origin (0, 0) from the top left corner to the centre of the canvas.
`square(0, 0, 100)`	The coordinates now reflect the changes we have made through translating the origin.

Figure A5.4



Sketch A5.5 no fill and an angle

We will use a variable for the angle called, strangely enough, `angle`. We can add another function to clear any fill colour (default is white) from the square called `noFill()`.

```
let angle = 45

function setup()
{
  createCanvas(400, 400)
  background(220)
  rectMode(CENTER)
  angleMode(DEGREES)
  noFill()
}

function draw()
{
  translate(200, 200)
  rotate(angle)
  square(0, 0, 100)
}
```

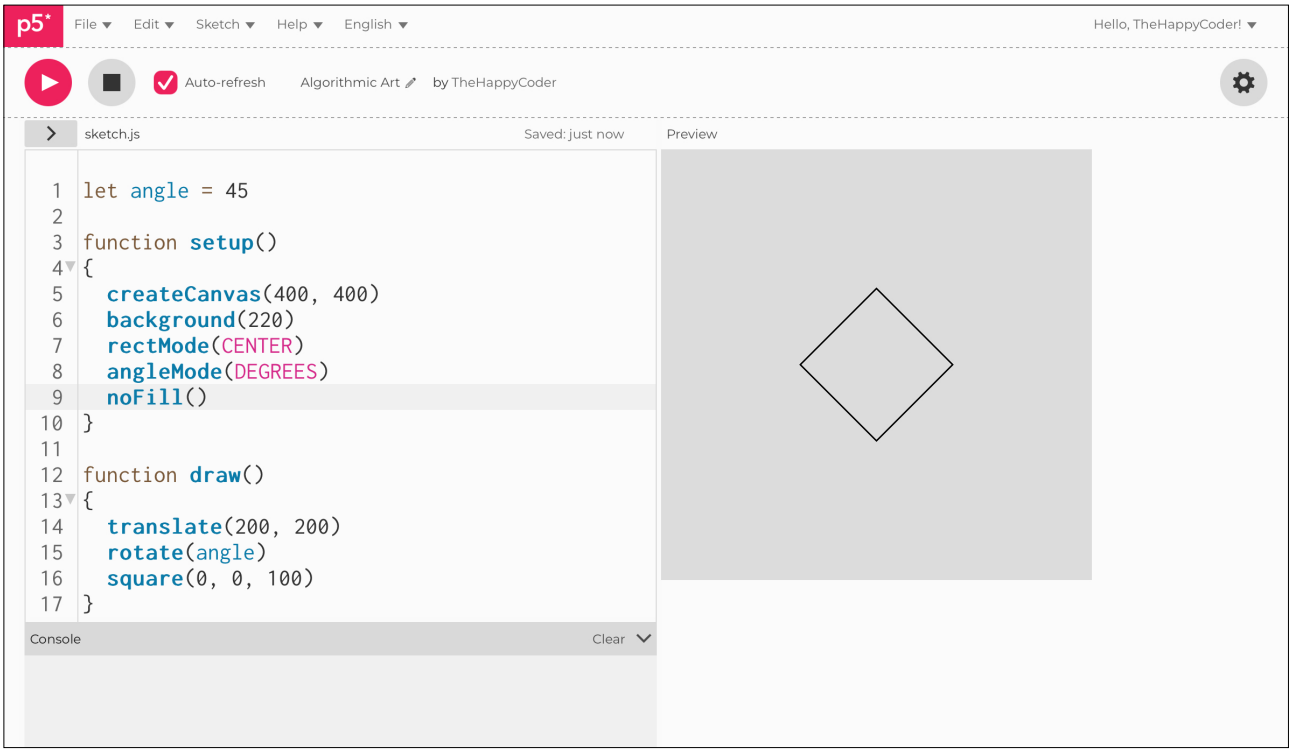
Notes

We get the same effect of a square at 45° but this time without the default fill colour.

Code Explanation

`noFill()`	Simply means there is no colour at all, making it completely transparent.

Figure A5.5



Sketch A5.6 rotating it slowly

We move the `background()` into the `draw()` function. We can use the `//` symbol to effectively remove the line of code. We add `1` to the angle on each iteration of the `draw()` loop. The effect is that the square slowly rotates.

! comment out the `background()` in the `setup()` function

```
let angle = 45

function setup()
{
  createCanvas(400, 400)
  // background(220)
  rectMode(CENTER)
  angleMode(DEGREES)
  noFill()
}

function draw()
{
  background(220)
  translate(200, 200)
  rotate(angle)
  square(0, 0, 100)
  angle++
}
```

Notes

Commenting out (//) is a very useful way of leaving code in but where you don't want to use it, or use it at a later date. The computer will ignore any lines that start with //. You can leave notes for yourself or someone else to help explain what your code is doing.

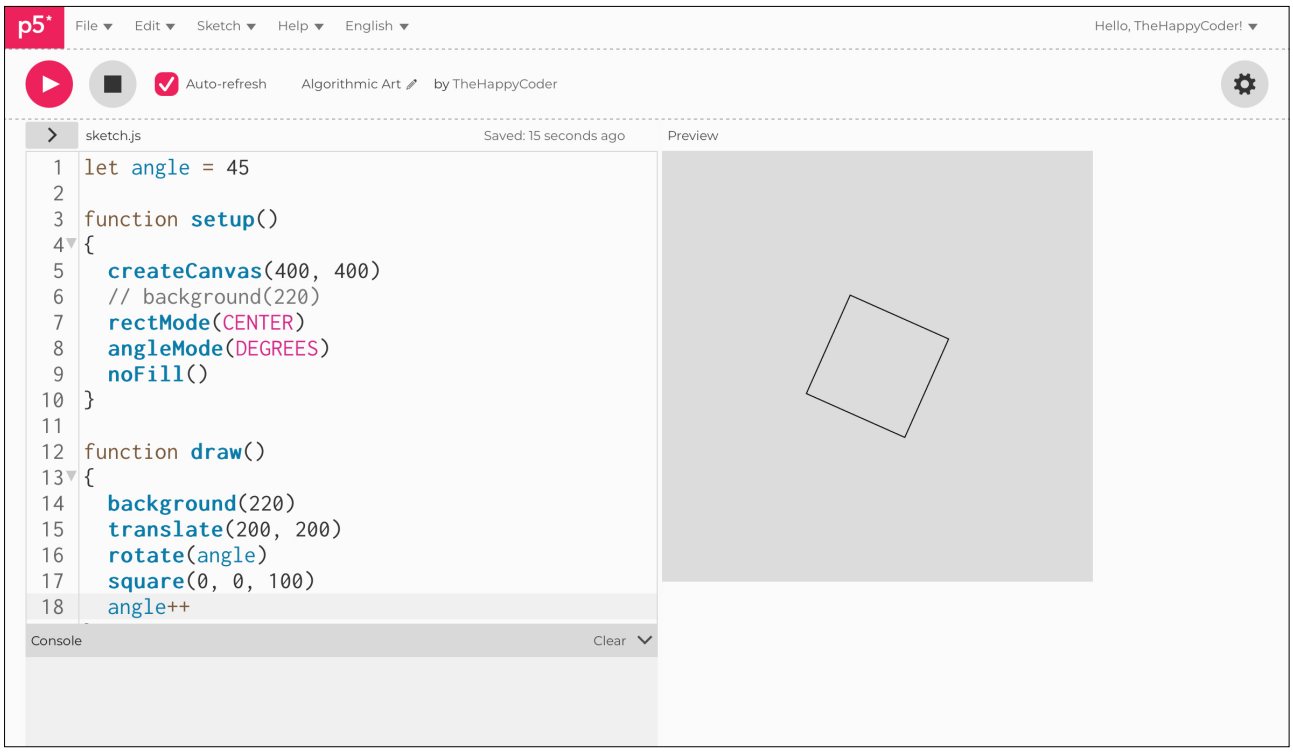
Challenge

Try commenting out other lines of code.

Code Explanation

//	The // symbols mean that the computer does not see anything on that line as code (as if invisible).

Figure A5.6 slowly rotates



Sketch A5.7 adding a second square

! Remove the background() in the setup() function.

If we add another square, the rotation is applied to it as well. Here, there are two squares, but they are both rotating together; hence, it looks as if there is just one square.

```
let angle = 45

function setup()
{
  createCanvas(400, 400)
  rectMode(CENTER)
  angleMode(DEGREES)
  noFill()
}

function draw()
{
  background(220)
  translate(200, 200)
  rotate(angle)
  square(0, 0, 100)
  square(0, 0, 100)
  angle++
}
```

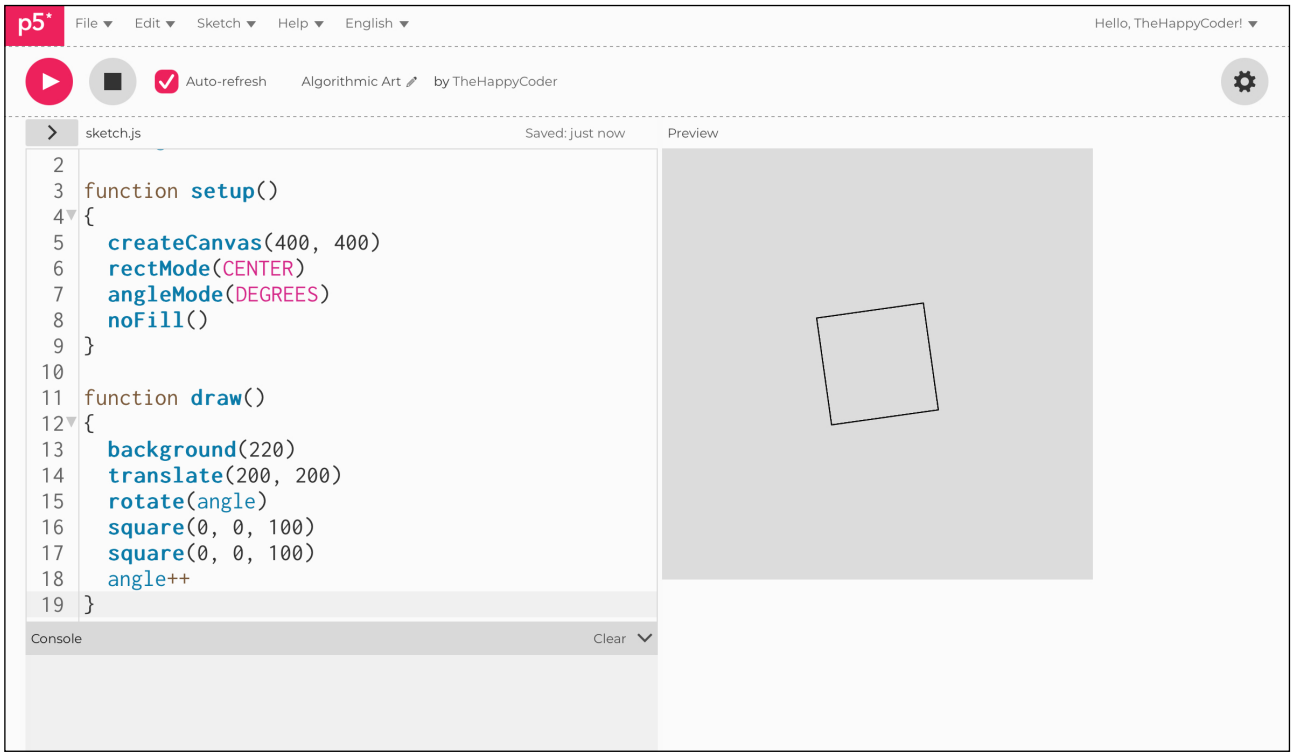
Notes

All you get is what looks like a square slowly rotating as before. What you have is both rotating at the same time; there are two squares. We want to stop one and let the other rotate.

Challenges

1. Change the speed of the rotation (`angle += 2`)
2. Change the direction

Figure A5.7



Sketch A5.8 push and pop

If we want to separate the two squares and have one rotate and the other not, we use two functions (as a pair) called `push()` and `pop()`. They are like bookends where you take the current state, change it, (rotate, for instance), and then return the code back to its previous state. What happens between `push()` and `pop()` stays between `push()` and `pop()`.

```
let angle = 45

function setup()
{
  createCanvas(400, 400)
  rectMode(CENTER)
  angleMode(DEGREES)
  noFill()
}

function draw()
{
  background(220)
  translate(200, 200)
  push()
  rotate(angle)
  square(0, 0, 100)
  pop()
  square(0, 0, 100)
  angle++
}
```

Notes

We now have one static square and a rotating square.

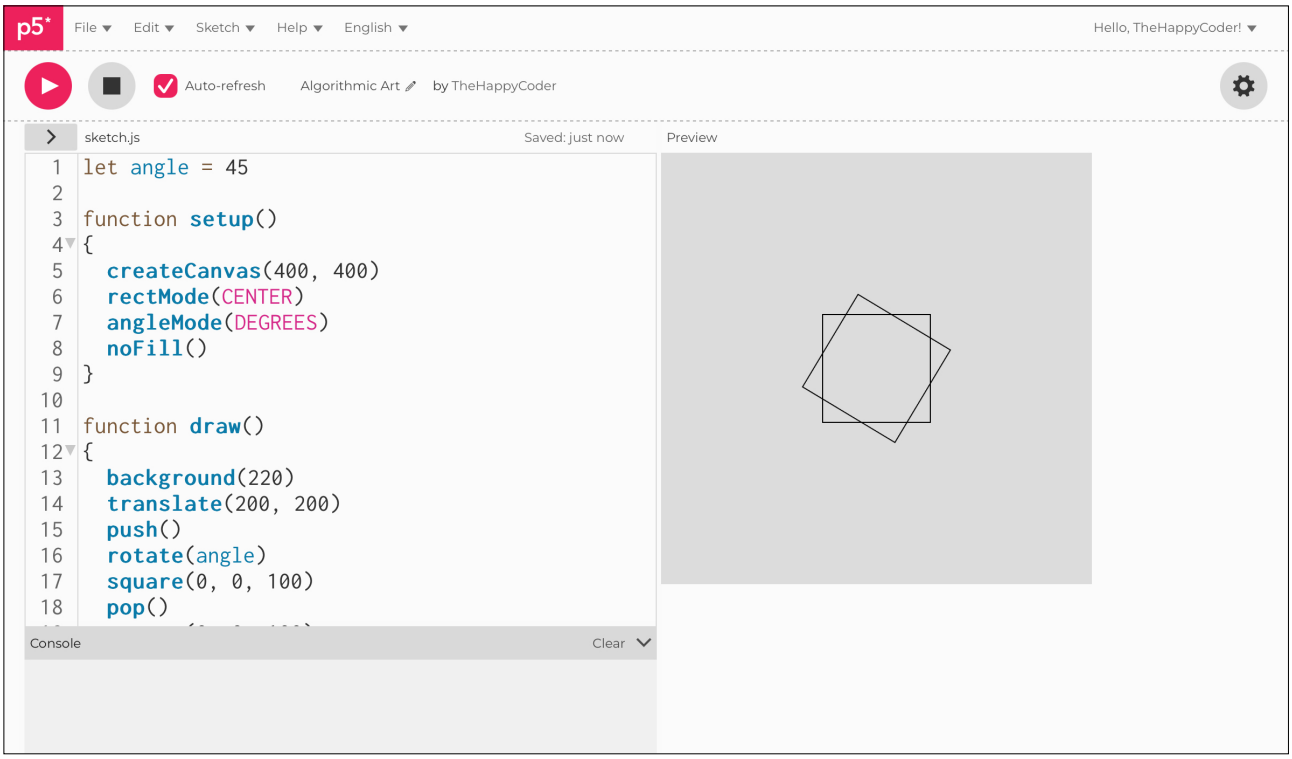
Challenge

Add a third square and have it rotate backwards (`-angle`), don't forget `push()` and `pop()`.

Code Explanation

`push()`	The push() function collects the state of everything at that moment in the code.
`pop()`	After you have added other functionality, the pop() function returns the original state when you introduced the push() function.

Figure A5.8



Sketch A5.9 a rectangle

! Start a new sketch

Instead of a square, we have a rectangle. To achieve this, we add another argument and call the `rect()` function. The first two arguments are the coordinates, the third is the width, and the fourth is the height of the rectangle.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  rectMode(CENTER)
}

function draw()
{
  rect(200, 200, 300, 50)
}
```

Notes

If you only give `rect()` three arguments, it will draw a square.

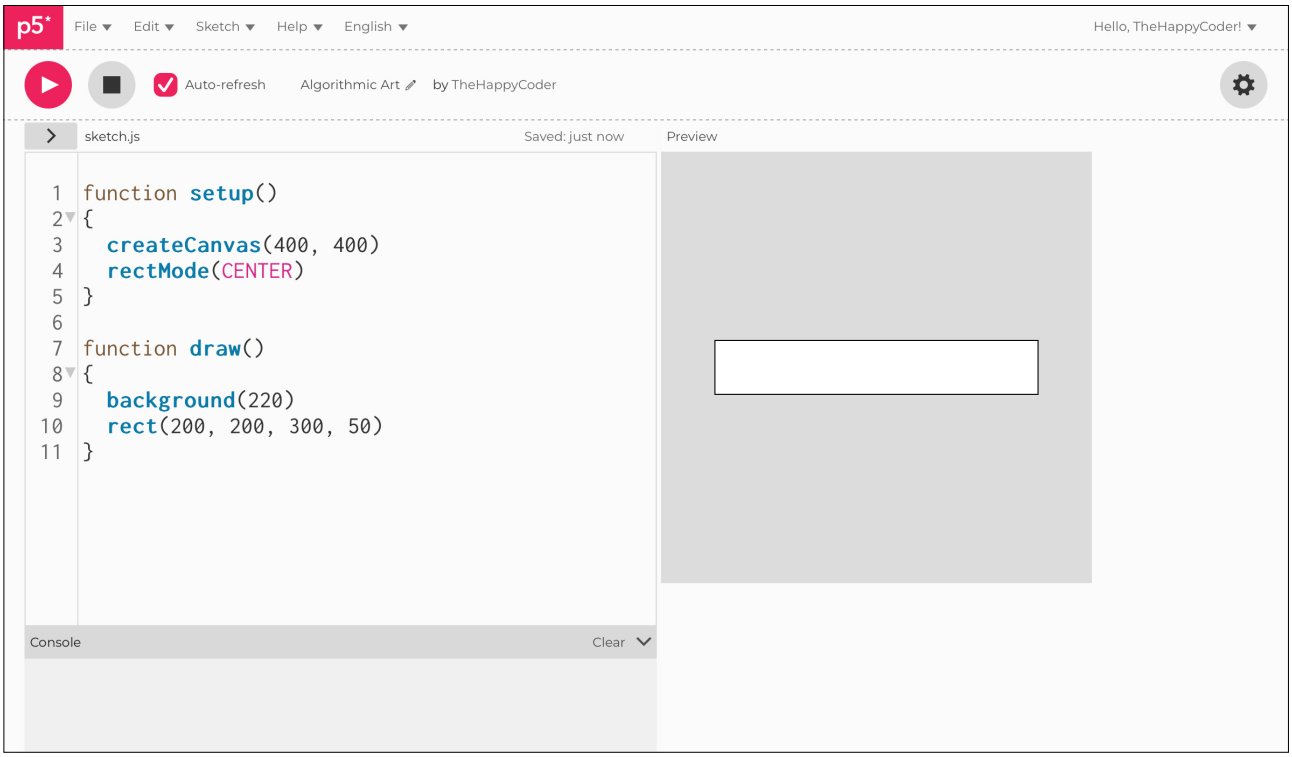
Challenge

Try other dimensions for sizes.

Code Explanation

`rect(200, 200, 300, 50)`	The rectangle has four arguments: the x-coordinate, the y-coordinate, the width, and the height of the rectangle.

Figure A5.9



Sketch A5.10 random rectangles

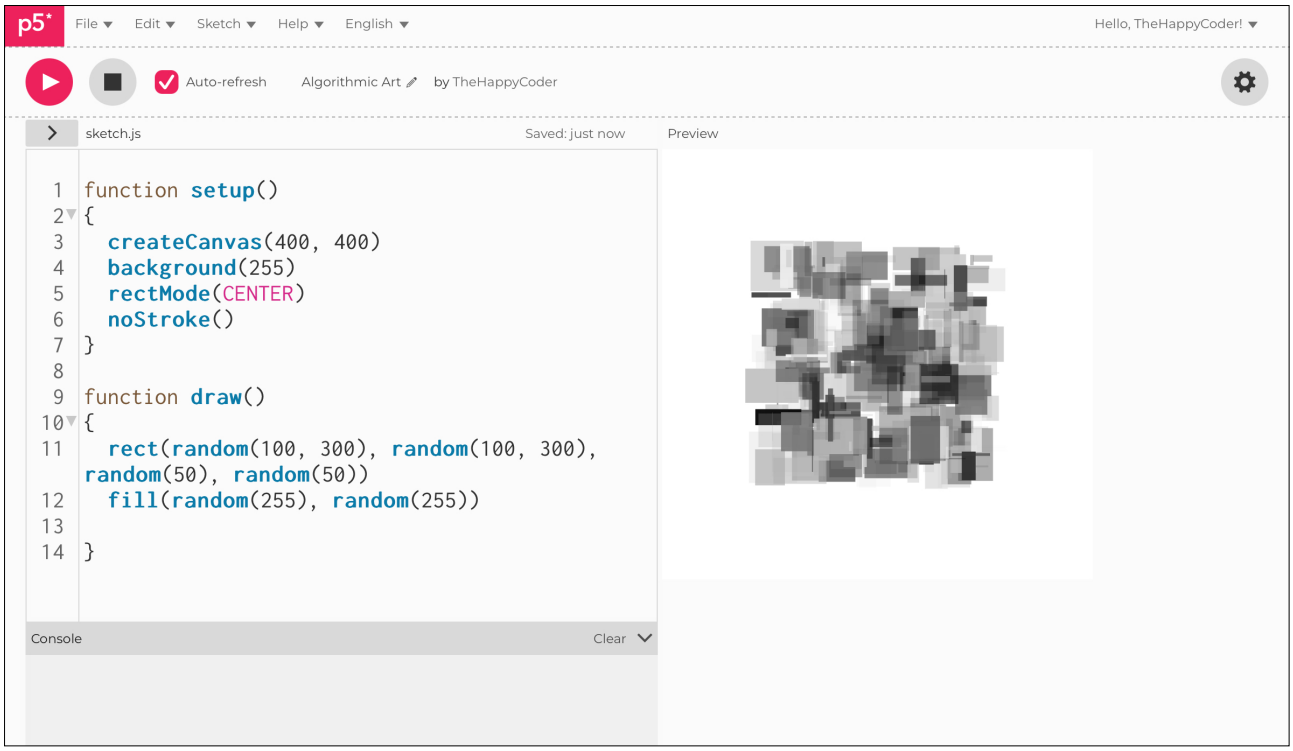
Let's go a little mad with random and create a nice pattern with the rectangles. We randomise the coordinates between 100 and 300, and give the width and height a random dimension of 50. We make the background white.

```
function setup()
{
  createCanvas(400, 400)
  background(255)
  rectMode(CENTER)
  noStroke()
}

function draw()
{
  rect(random(100, 300), random(100, 300), random(50), random(50))
  fill(random(255), random(255))
}
```

Challenges

1. Change the random dimensions.
2. Have it draw just 100 rectangles.
3. Add colour.

Figure A5.10



The Joy of Coding Algorithmic Art

Module A Unit #6 ellipses and triangles

Module A Unit #6: ellipses and triangles

As well as introducing two new shapes, we will also explore how to interact with shapes with the mouse. We will be able to move and stretch these shapes, offering more opportunities to create interesting designs that are more than static images.

Key concepts:

```
ellipse()
triangle()
mouseX
mouseY
move
stretch
```

Sketch A6.1 drawing an ellipse

We start with a new sketch with an ellipse. The `ellipse()` function has four arguments: the x and y coordinates, and the width and height of the ellipse.

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

function draw()
{
  background(220)
  ellipse(200, 200, 300, 100)
}
```

Notes

If you use only three dimensions, you will draw a circle.

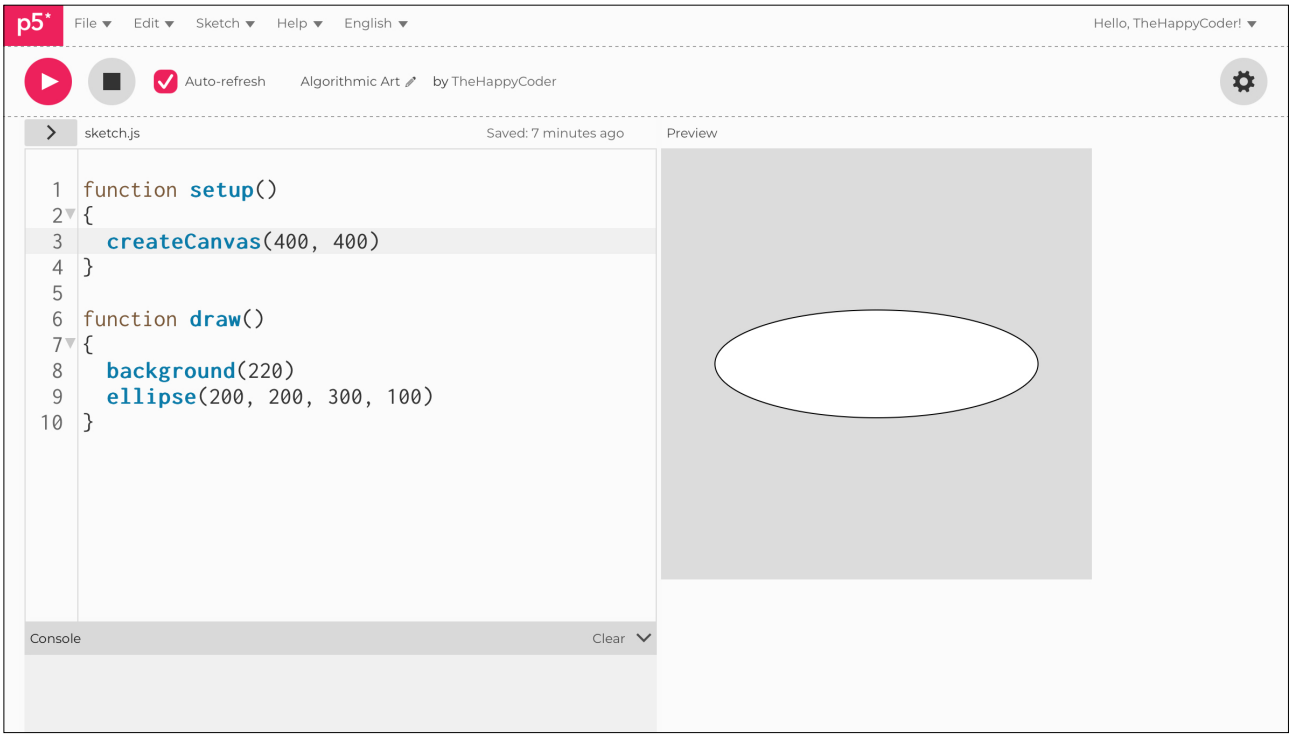
Challenge

Try other dimensions for the height and width.

Code Explanation

`ellipse(200, 200, 300, 100)`	We have an ellipse at the centre of the canvas (200, 200) and a width of 300 and height of 100.

Figure A6.1



Sketch A6.2 mouseX and mouseY

We can use some built-in variables; one set of variables is called mouseX and mouseY. They return the coordinates of the mouse cursor position on the canvas. We can illustrate it with our ellipse.

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

function draw()
{
  background(220)
  ellipse(mouseX, mouseY, 300, 100)
}
```

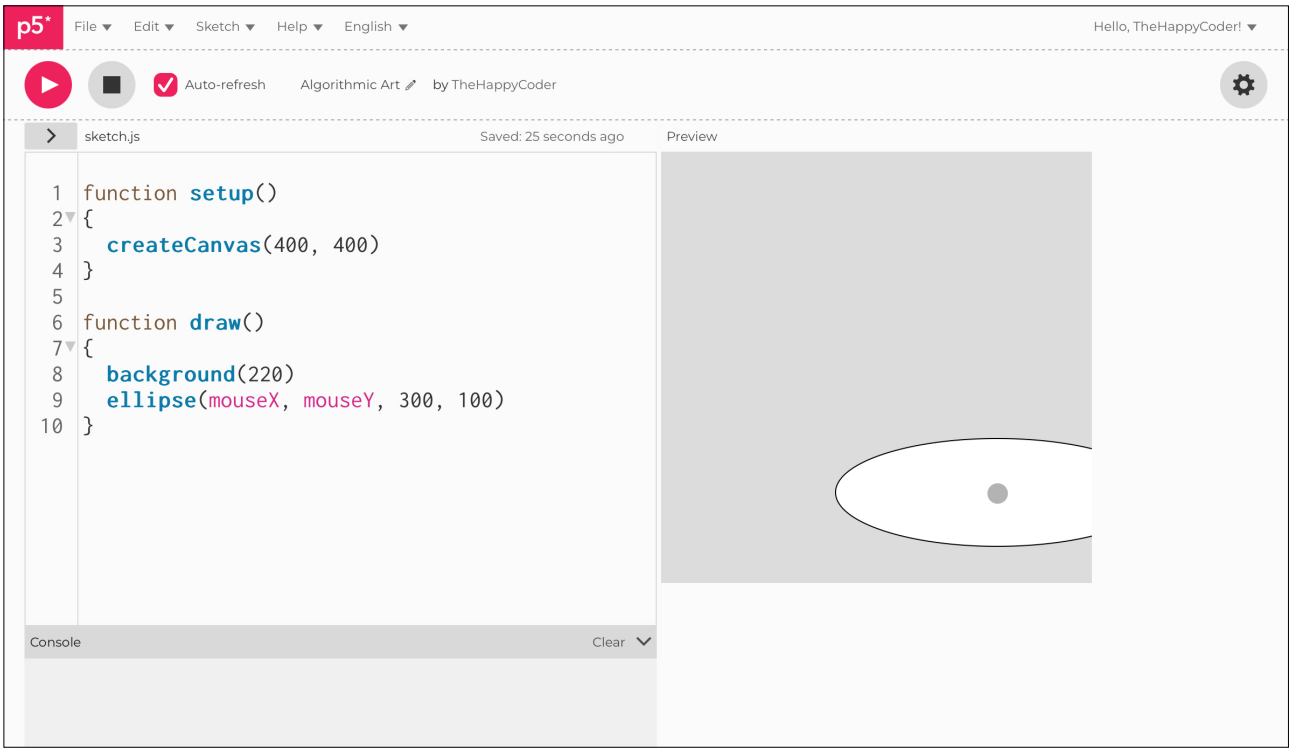
Notes

As you move your mouse cursor around the canvas, the ellipse follows it.

Code Explanation

`ellipse(mouseX, mouseY, 300, 100)`	mouseX returns the x coordinate, and mouseY returns the y coordinate of the cursor.

Figure A6.2



Sketch A6.3 stretchy time

If we move mouseX and mouseY to the dimensions of the ellipse, we can see how we can change the shape of the ellipse through moving our cursor.

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

function draw()
{
  background(220)
  ellipse(200, 200, mouseX, mouseY)
}
```

Notes

You can now stretch the ellipse by moving your mouse.

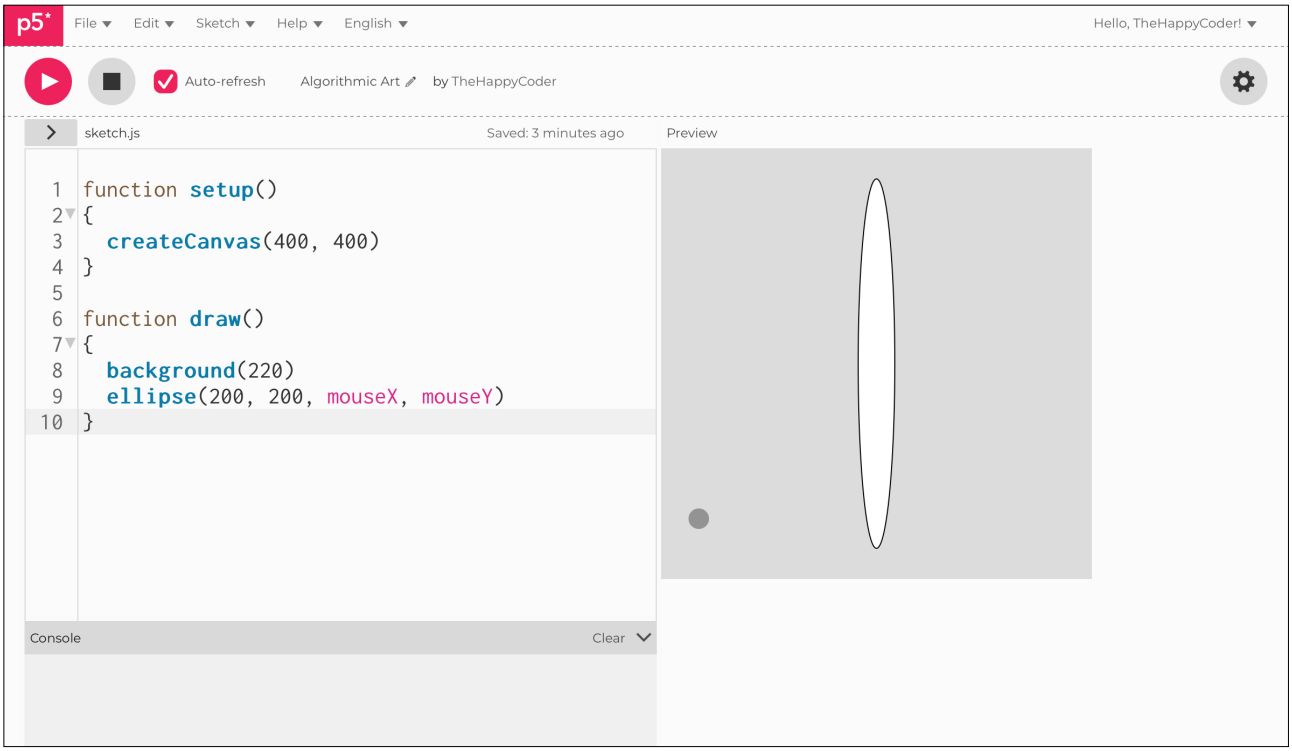
Challenge

Try it with the rectangle.

Code Explanation

`ellipse(200, 200, mouseX, mouseY)`	The mouseX and mouseY are the width and height, respectively.

Figure A6.3



Sketch A6.4 triangle

Instead of an ellipse, let's make a triangle. A triangle has coordinates for all three corners, so the function `triangle()` has six arguments. There are a lot of numbers to get your head around; often, I sketch it out on a piece of paper first so I get the right coordinates for the right corner.

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

function draw()
{
  background(220)
  triangle(100, 100, 100, 300, 300, 300)
}
```

Notes

We have three sets of coordinates: (100, 100), (100, 300) and (300, 300).

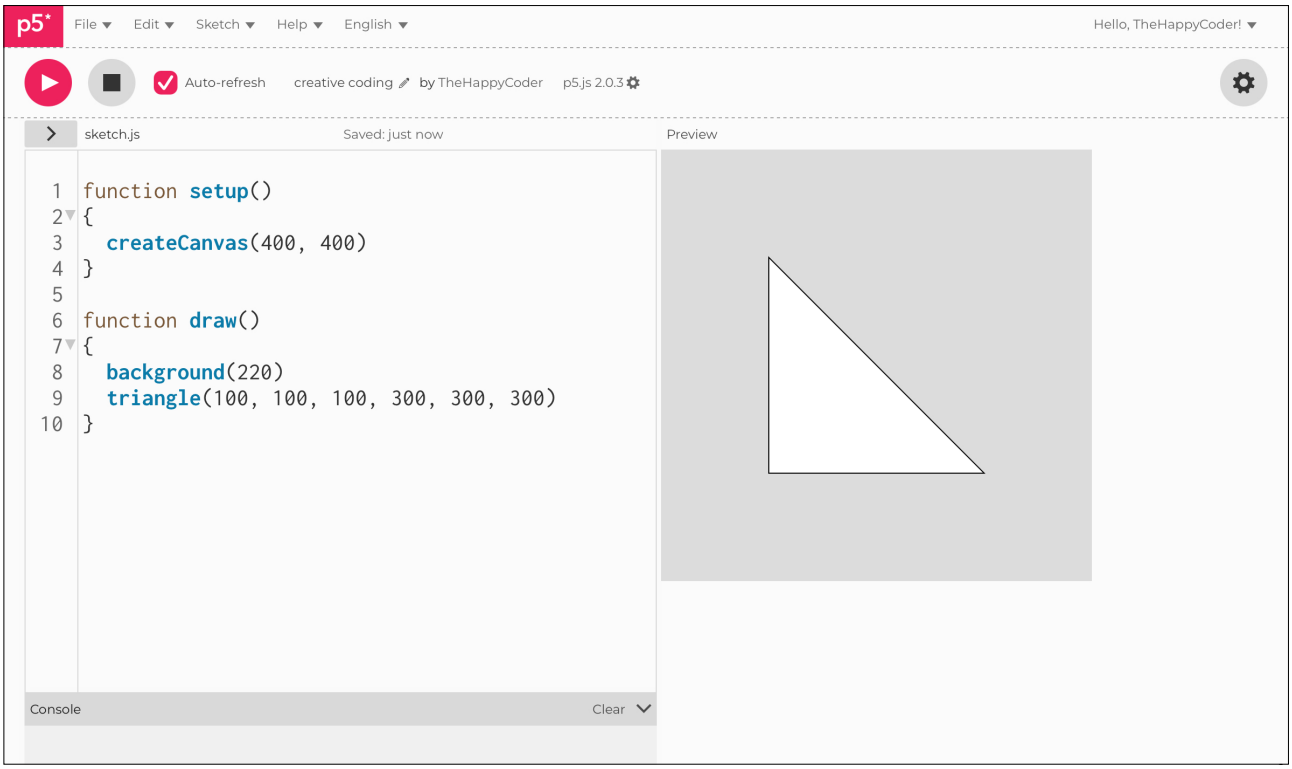
Challenge

Draw more triangles, make a simple pattern.

Code Explanation

`triangle(100, 100, 100, 300, 300, 300)`	A triangle with three sets of coordinates for each corner (vertex).

Figure A6.4



Sketch A6.5 width and height

We can make use of another built-in variable, `width` and `height`. These variables return the width and height of the canvas. So in our case, the value of `width` is `400` and the value of `height` is also `400`. If we change the dimensions of the canvas, these values also change. If we divide the `width` and `height` by 2, we always get the centre of the canvas, whatever its dimensions.

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

function draw()
{
  background(220)
  translate(width/2, height/2)
  triangle(100, 100, 100, 300, 300, 300)
}
```

Notes

We have translated the origin of the canvas to its centre rather than the top left-hand corner. This has pushed the triangle down and to the left by 200 pixels.

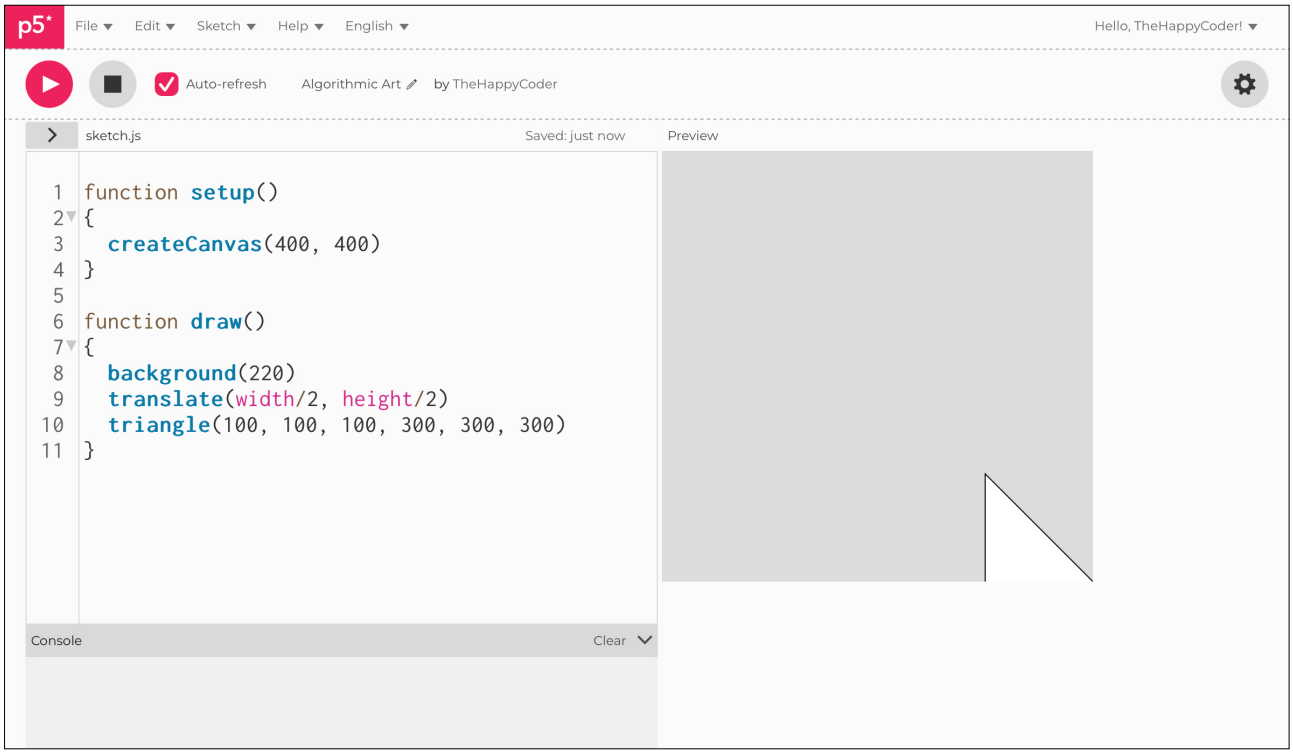
Challenge

Change the dimensions of the canvas.

Code Explanation

`translate(width/2, height/2)`	The width/2 is 200, and the height/2 is also 200.

Figure A6.5



Sketch A6.6 rebuilding the triangle

This is where a physical drawing often helps. We can redraw the triangle with new coordinates. Remember that the origin is now in the centre of the canvas, and we measure the x and y coordinates from the centre; hence, -100 is to the left of the centre for x and upwards for the y coordinates, and vice versa for 100.

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

function draw()
{
  background(220)
  translate(width/2, height/2)
  triangle(-100, -100, -100, 100, 100, 100)
}
```

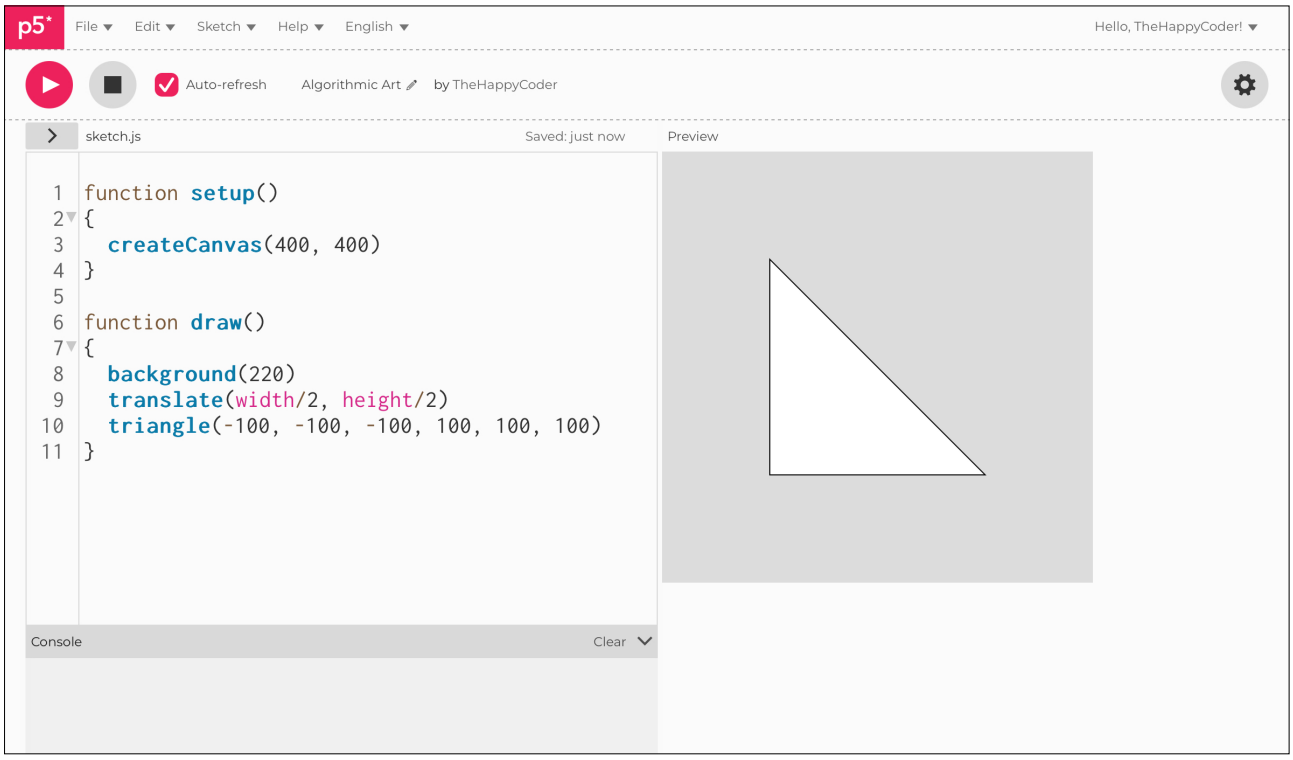
Notes

We have the same triangle as previously.

Challenge

Change the coordinates to draw different types of triangles.

Figure A6.6



Sketch A6.7 a length

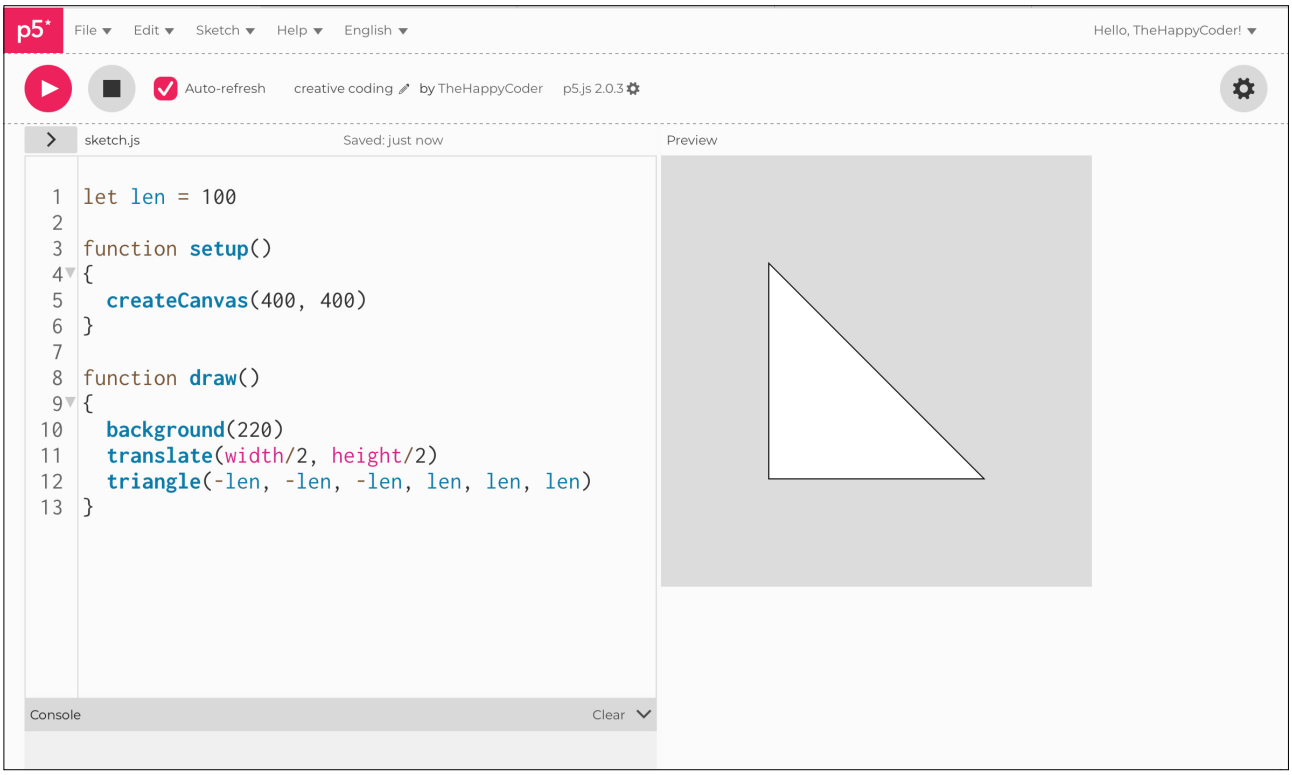
Let's give the dimension a variable called `len` (for length).

```
let len = 100

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

function draw()
{
  background(220)
  translate(width/2, height/2)
  triangle(-len, -len, -len, len, len, len)
}
```

Notes

This can be handy if we change the dimension and therefore only need to change it once.

Figure A6.7



Sketch A6.8 a bit more pointy

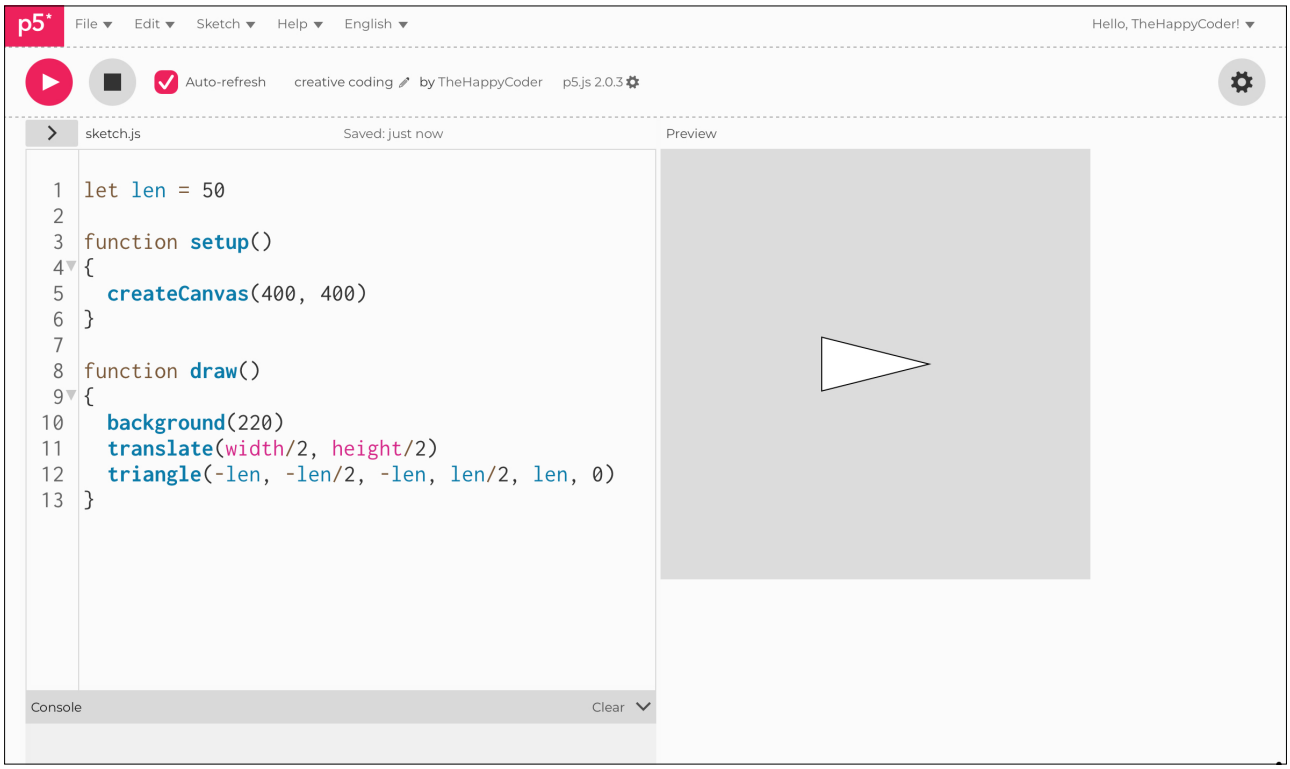
Make it a bit smaller and more pointy so that it isn't an equilateral triangle.

```
let len = 50

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

function draw()
{
  background(220)
  translate(width/2, height/2)
  triangle(-len, -len/2, -len, len/2, len, 0)
}
```

Notes

We can see what direction it is pointing in.

Figure A6.8



Sketch A6.9 a hundred of them

Now we create a `for()` loop to draw `100` triangles. We put those two lines of code inside the `for()` loop.

```
let len = 50

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

function draw()
{
  background(220)
  for (let i = 0; i < 100; i++)
  {
    translate(width/2, height/2)
    triangle(-len, -len/2, -len, len/2, len, 0)
  }
}
```

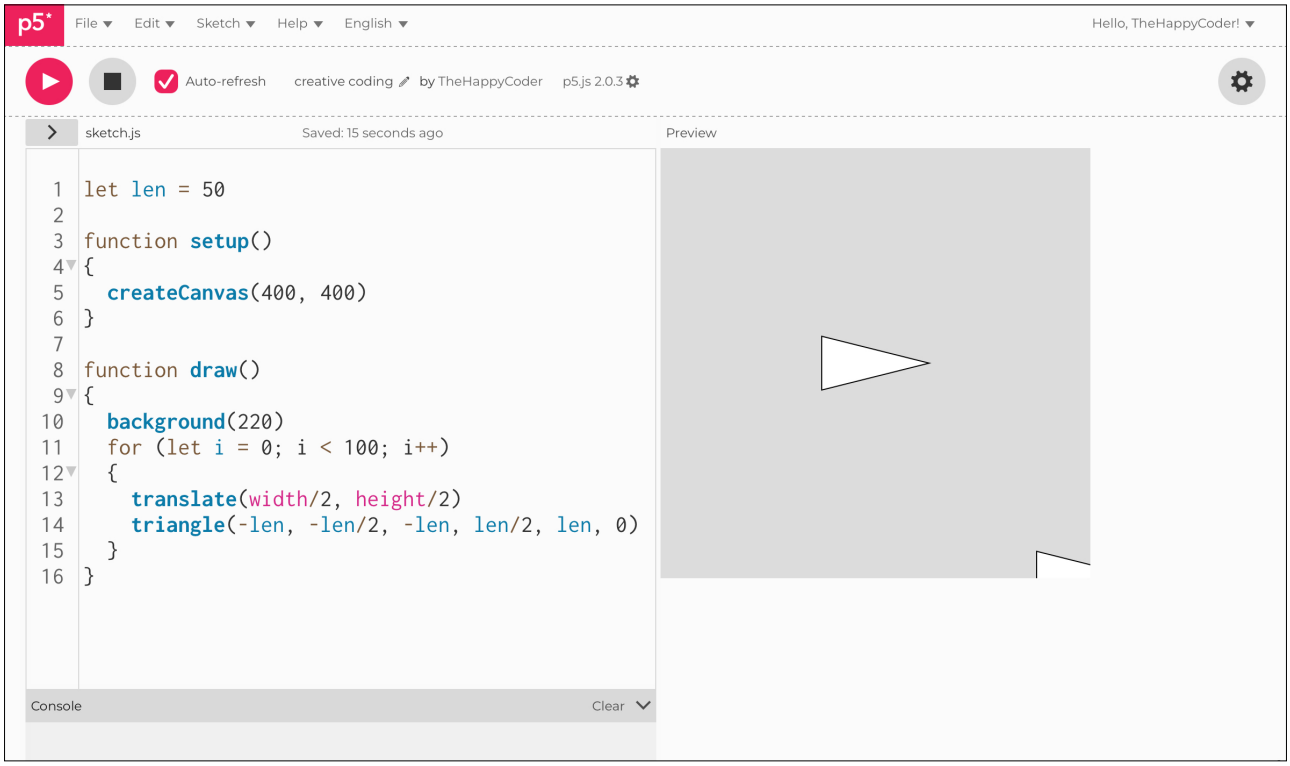
Notes

The problem is that the `translate()` function is accumulative, which means it shifts the origin by that amount each iteration in the `for()` loop. We need to use `push()` and `pop()`, and also we need to stop repeating the `for()` in `draw()` with `noLoop()`.

Code Explanation

`for (let i = 0; i < 100; i++)`	A for() loop where the variable i, which is defined and initialised to 0, is then incremented by 1 until it reaches 100.

Figure A6.9



Sketch A6.10 push pop stop

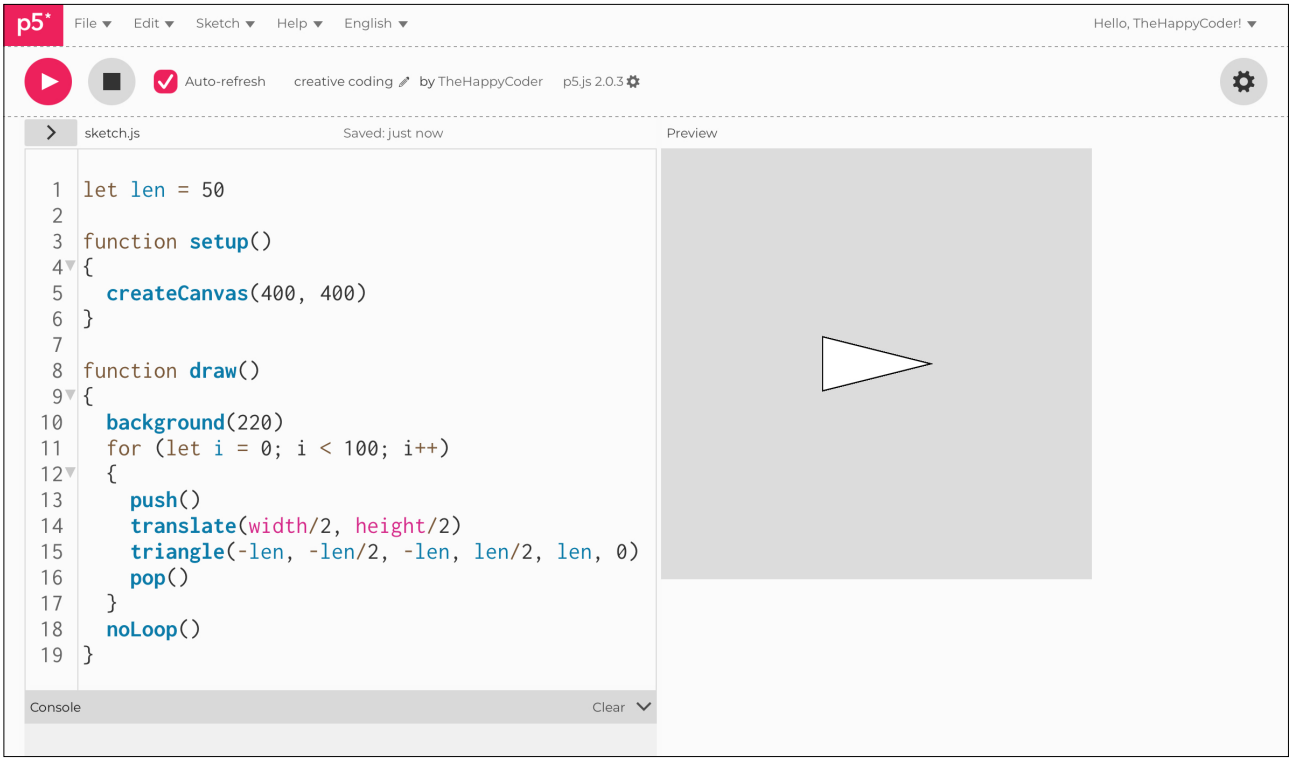
We add `push()` and `pop()` to stop the `translate()` from adding itself 100 times, and also the `noLoop()` to stop the `draw()` function once it has drawn the 100 triangles.

```
let len = 50

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

function draw()
{
  background(220)
  for (let i = 0; i < 100; i++)
  {
    push()
    translate(width/2, height/2)
    triangle(-len, -len/2, -len, len/2, len, 0)
    pop()
  }
  noLoop()
}
```

Notes

We have `100` triangles, but they are all in one place. We could also put all of this code in the `setup()` function and therefore no need to use `noLoop()`.

Figure A6.10



Sketch A6.11 rotate the triangle

We are rotating all the triangles by 45° degrees.

```
let len = 50
let angle = 45

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
}

function draw()
{
  background(220)
  for (let i = 0; i < 100; i++)
  {
    push()
    translate(width/2, height/2)
    rotate(angle)
    triangle(-len, -len/2, -len, len/2, len, 0)
    pop()
  }
  noLoop()
}
```

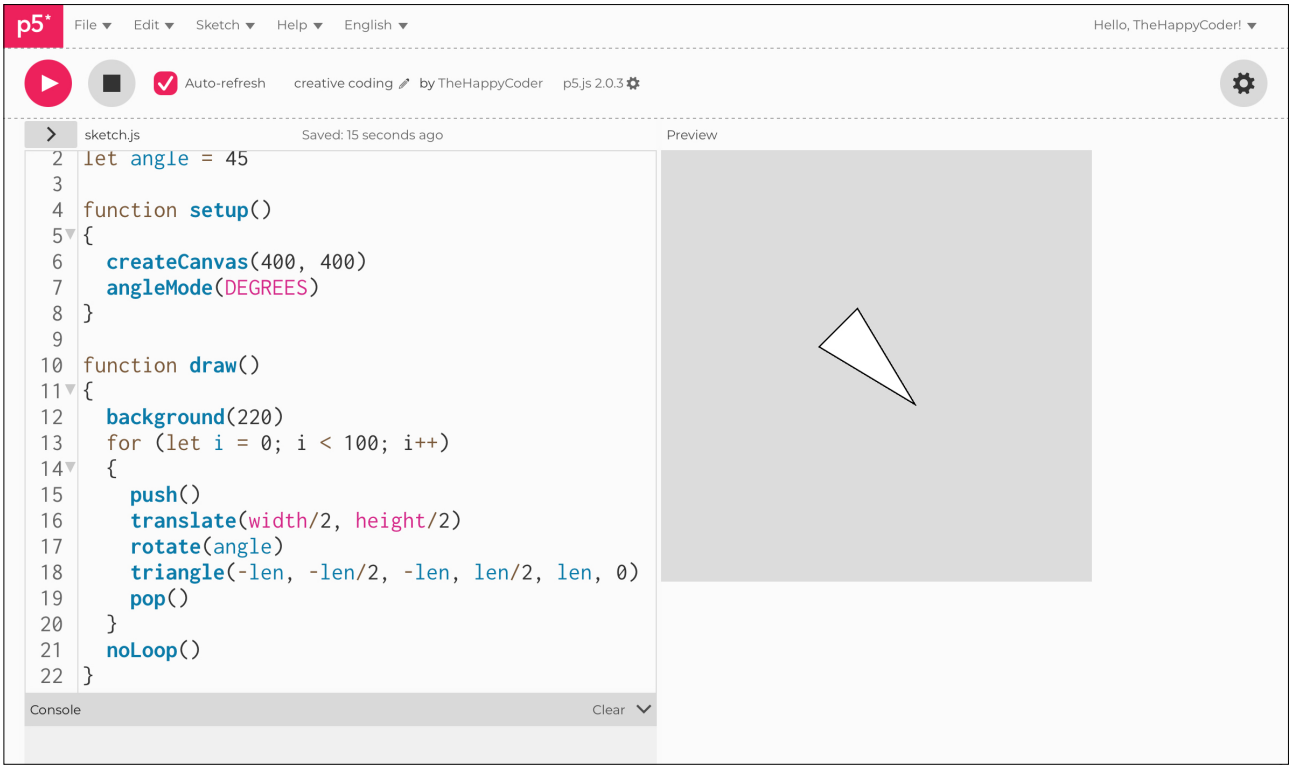
Notes

We have 100 triangles, all rotated by 45°.

Challenge

Try other angles.

Figure A6.11



Sketch A6.12 split the triangles up

But we want to see all the triangles; to do that, we can translate them randomly across the whole canvas (`width` and `height`).

```
let len = 50
let angle = 45

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
}

function draw()
{
  background(220)
  for (let i = 0; i < 100; i++)
  {
    push()
    translate(random(width), random(height))
    rotate(angle)
    triangle(-len, -len/2, -len, len/2, len, 0)
    pop()
  }
  noLoop()
}
```

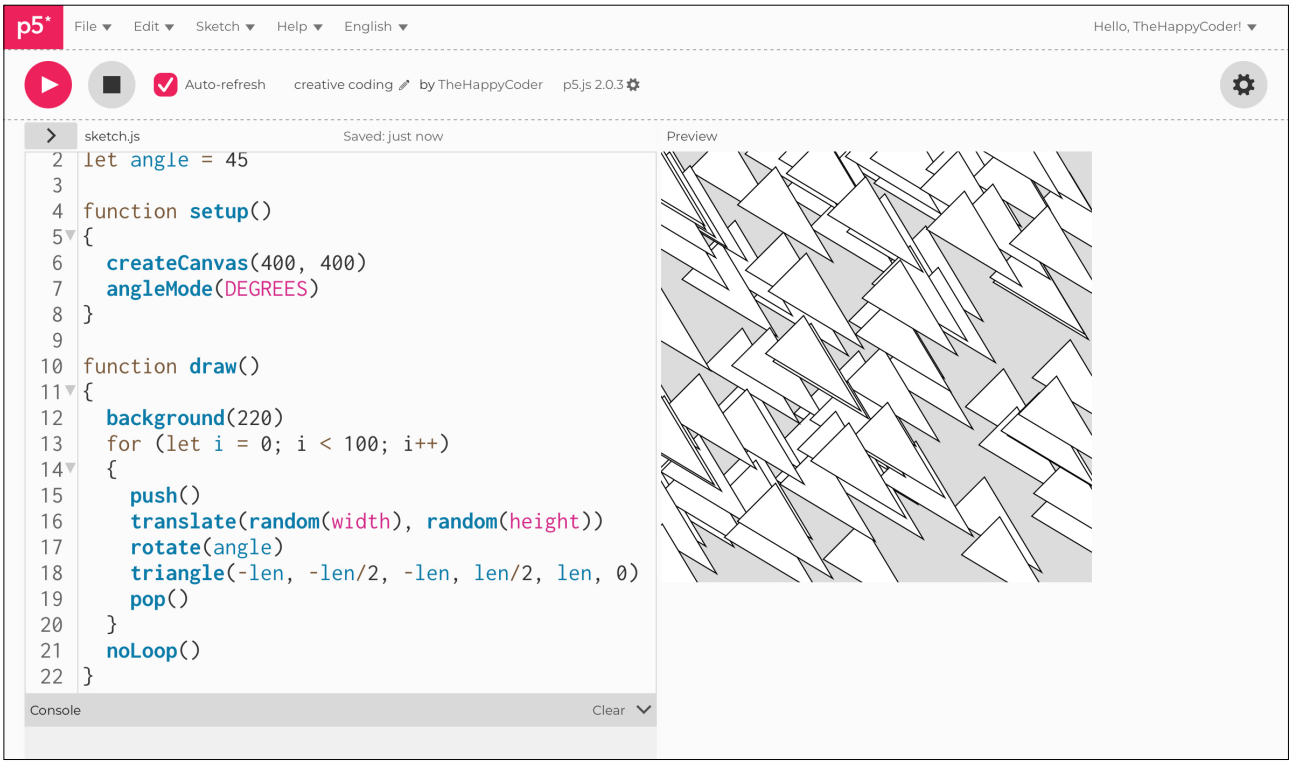
Notes

Now each triangle drawn has its own random position.

Challenges

1. Try pushing them all into a square in the middle.
2. Make the triangles smaller or random in size.
3. Can you work out how to rotate each one separately?

Figure A6.12



Sketch A6.13 random angles

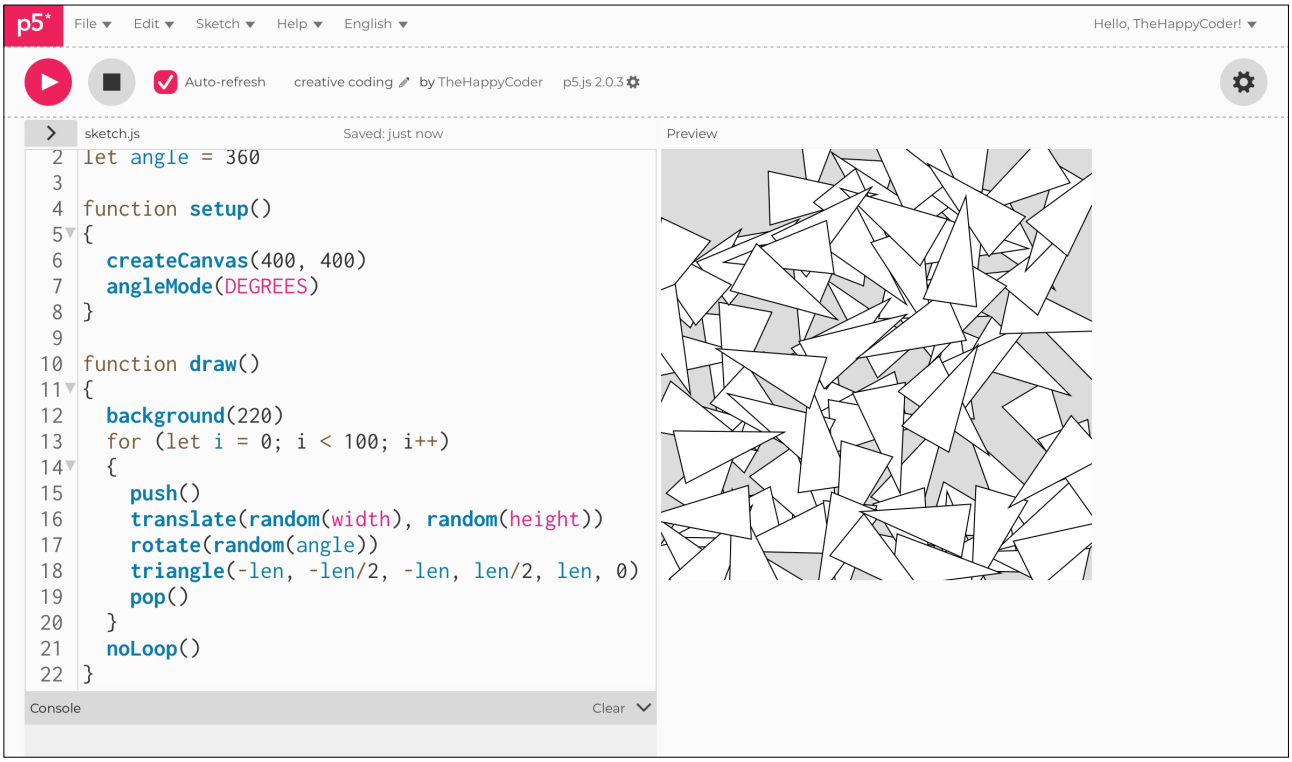
This is how you can randomly set the angles. Change the angle to a full 360° and rotate randomly within that angle.

```
let len = 50
let angle = 360

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
}

function draw()
{
  background(220)
  for (let i = 0; i < 100; i++)
  {
    push()
    translate(random(width), random(height))
    rotate(random(angle))
    triangle(-len, -len/2, -len, len/2, len, 0)
    pop()
  }
  noLoop()
}
```

Notes

Rotating randomly from 0° to 360° degrees

Figure A6.13



Sketch A6.14 random length

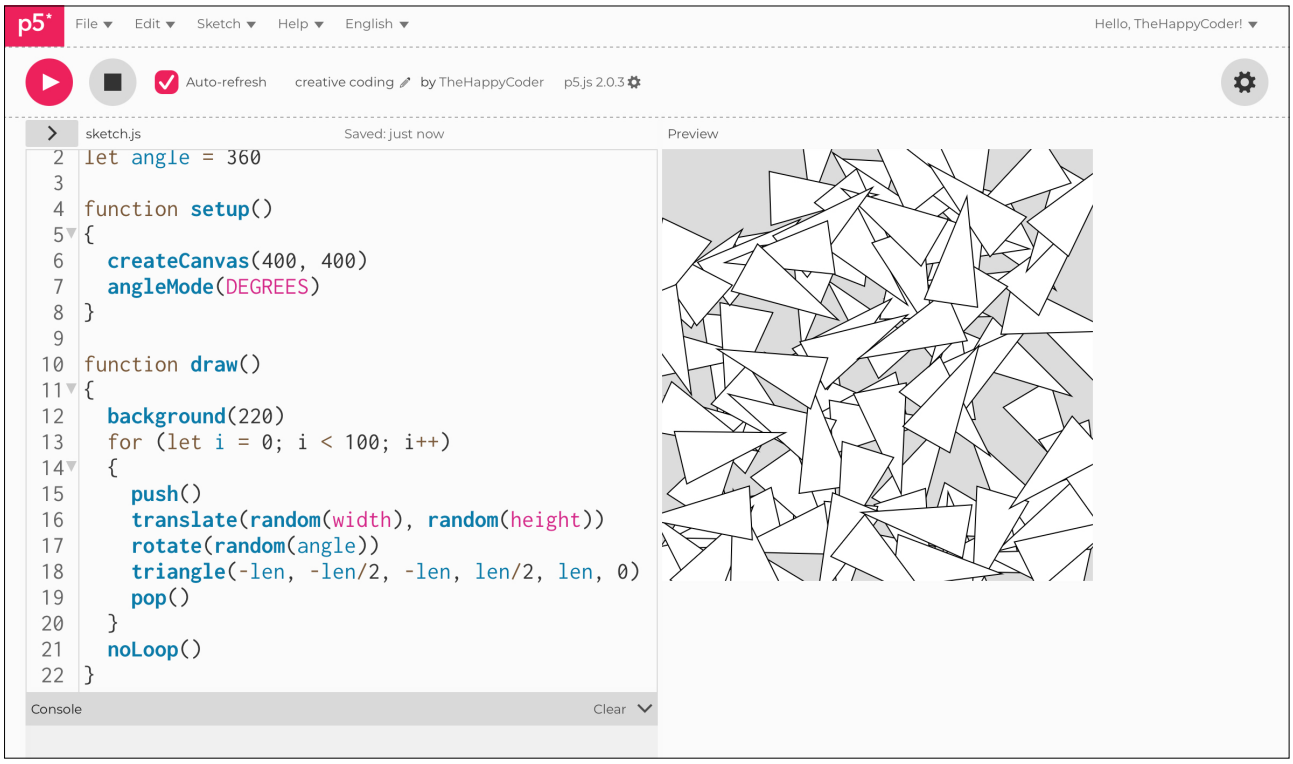
Now for the size of the triangle. This will keep the proportions the same. We will have triangles between 5 and 50 long.

```
let len = 50
let angle = 360

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
}

function draw()
{
  background(220)
  for (let i = 0; i < 100; i++)
  {
    push()
    translate(random(width), random(height))
    rotate(random(angle))
    len = random(5, 50)
    triangle(-len, -len/2, -len, len/2, len, 0)
    pop()
  }
  noLoop()
}
```

Notes

There is a lot to play with here.

Figure A6.14



Sketch A6.15 random colouring

Make the background white and give each triangle a random colour (and alpha).

```
let len = 50
let angle = 360

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
}

function draw()
{
  background(255)
  for (let i = 0; i < 100; i++)
  {
    push()
    translate(random(width), random(height))
    rotate(random(angle))
    len = random(5, 50)
    fill(random(255), random(255), random(255), random(255))
    triangle(-len, -len/2, -len, len/2, len, 0)
    pop()
  }
  noLoop()
}
```

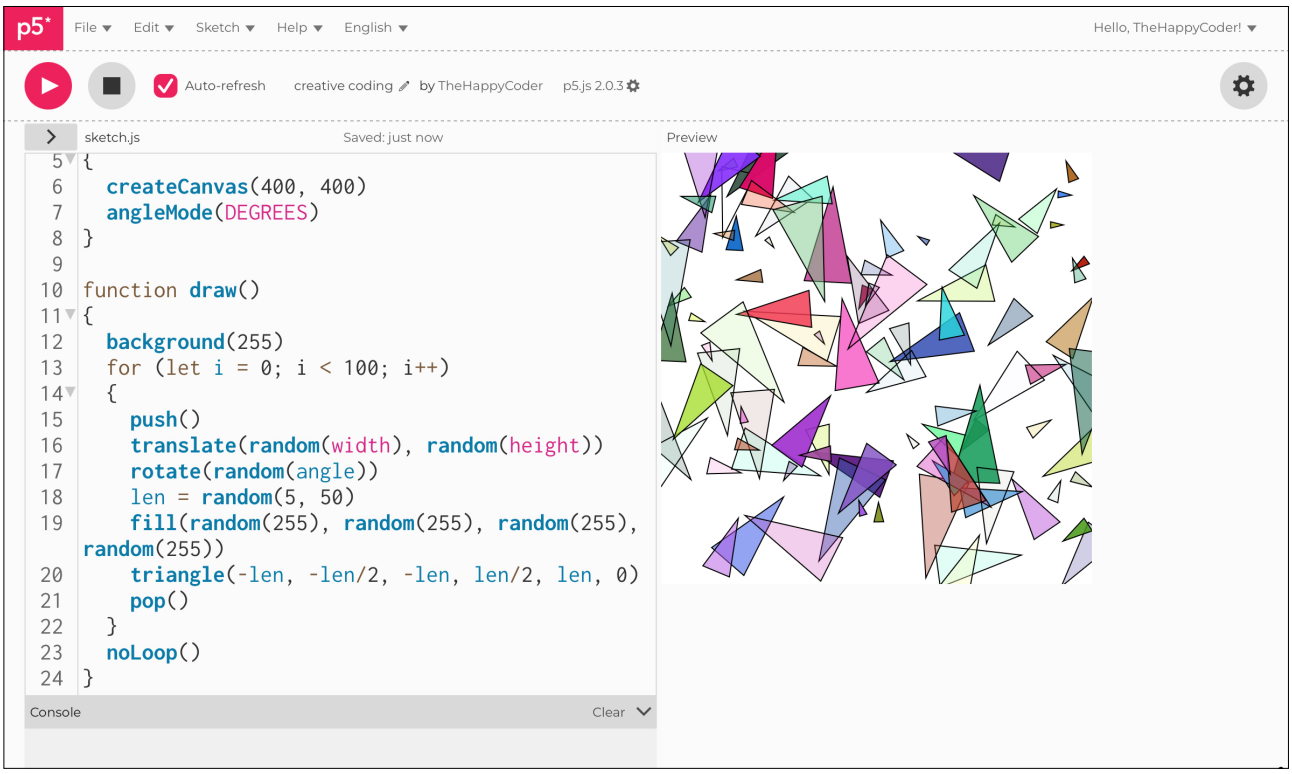
Notes

Starting to look interesting.

Challenges

1. What else could you develop alongside these ideas?
2. Different shapes, colours, etc.

Figure A6.15



The Joy of Coding Algorithmic Art

Module A Unit #7 pixels

Module A Unit #7: pixels

Every image on your screen or on the canvas is made up of pixels. They are too tiny to be seen individually; even so, we can draw pixels using the point() function on the canvas. This is a very simple and early introduction to pixels as a shape we can draw. There is so much more to the pixels of the canvas or an image on the canvas, but that is for a bit later on.

One concept which is based on the for() loop is something called a nested loop. This is useful for pixel arrays and 3D objects and shapes.

Key concepts:

pixels
point()
nested loops
colorMode()

Sketch A7.1 what's the point?

! Our starting sketch

We have drawn a pixel at position (0, 0). Can you see it?
A `point()` shape is just one pixel shape.

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

function draw()
{
  background(220)
  point(0, 0)
}
```

Notes

You can't see it! Look closer at Fig. A7.1b, can you see it now?

Challenge

Give it a `strokeWeight(50)` to see it clearly.

Code Explanation

`point(0, 0)`	This draws a pixel at coordinates (0, 0).

Figure A7.1a

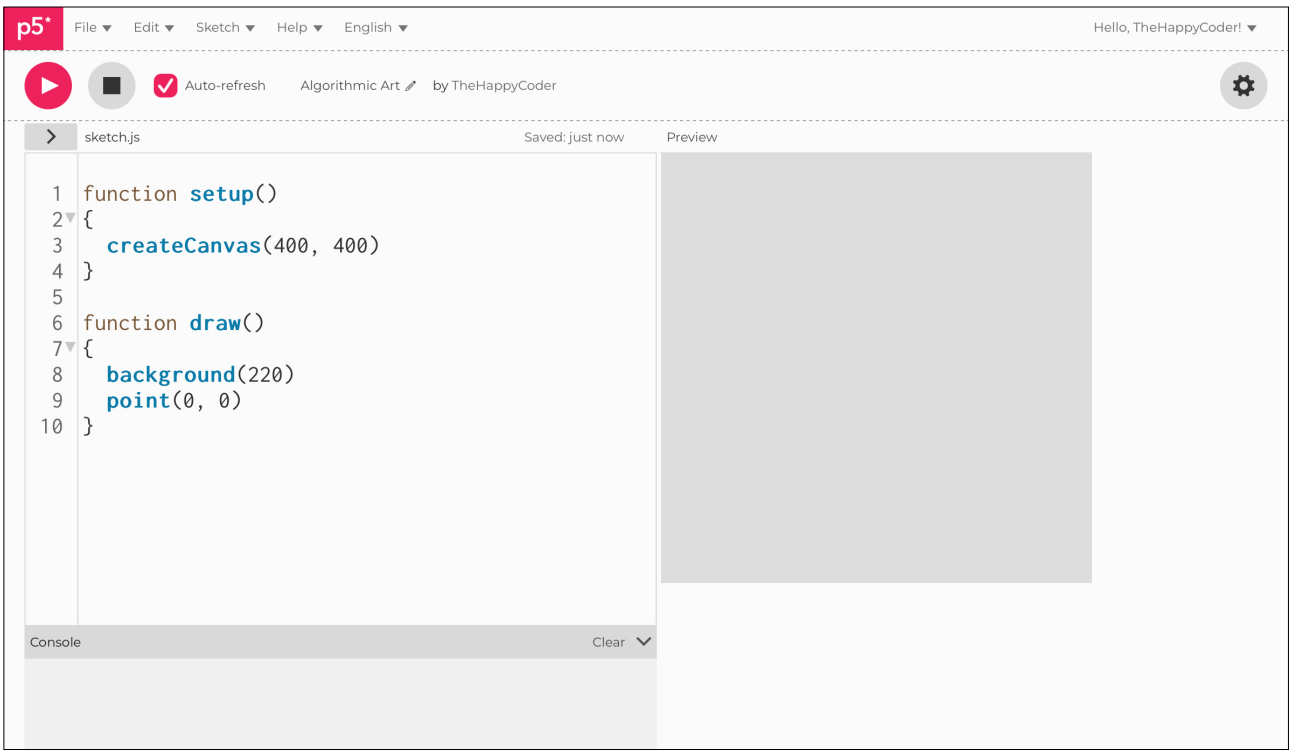
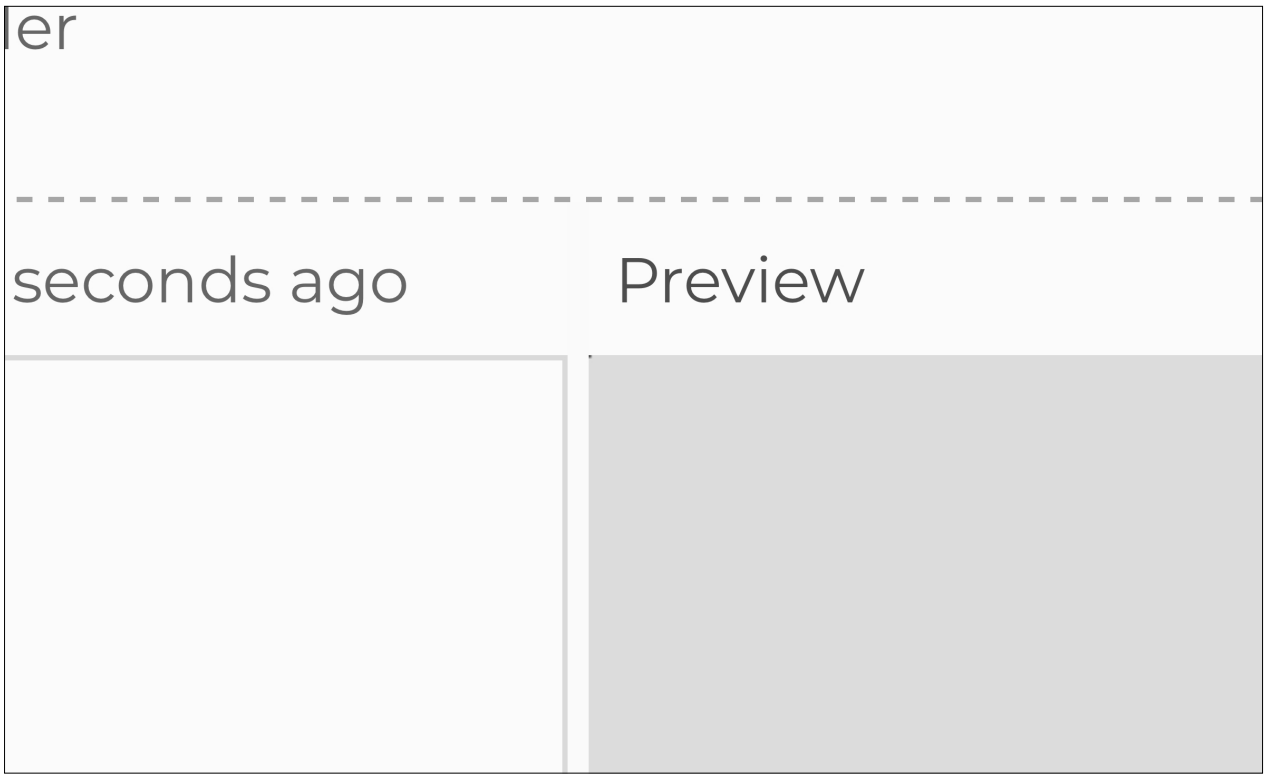


Figure A7.1b: pixel point



Sketch A7.2 a line of pixels

We use a `for()` loop to draw a line across the top of the canvas. Remember to change `point(0, 0)` to `point (x, 0)`.

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

function draw()
{
  background(220)
  for (let x = 0; x < width; x++)
  {
    point(x, 0)
  }
}
```

Notes

You should be able to see a thin line going across the top of the canvas. We used `x` instead of `i` for this `for()` loop.

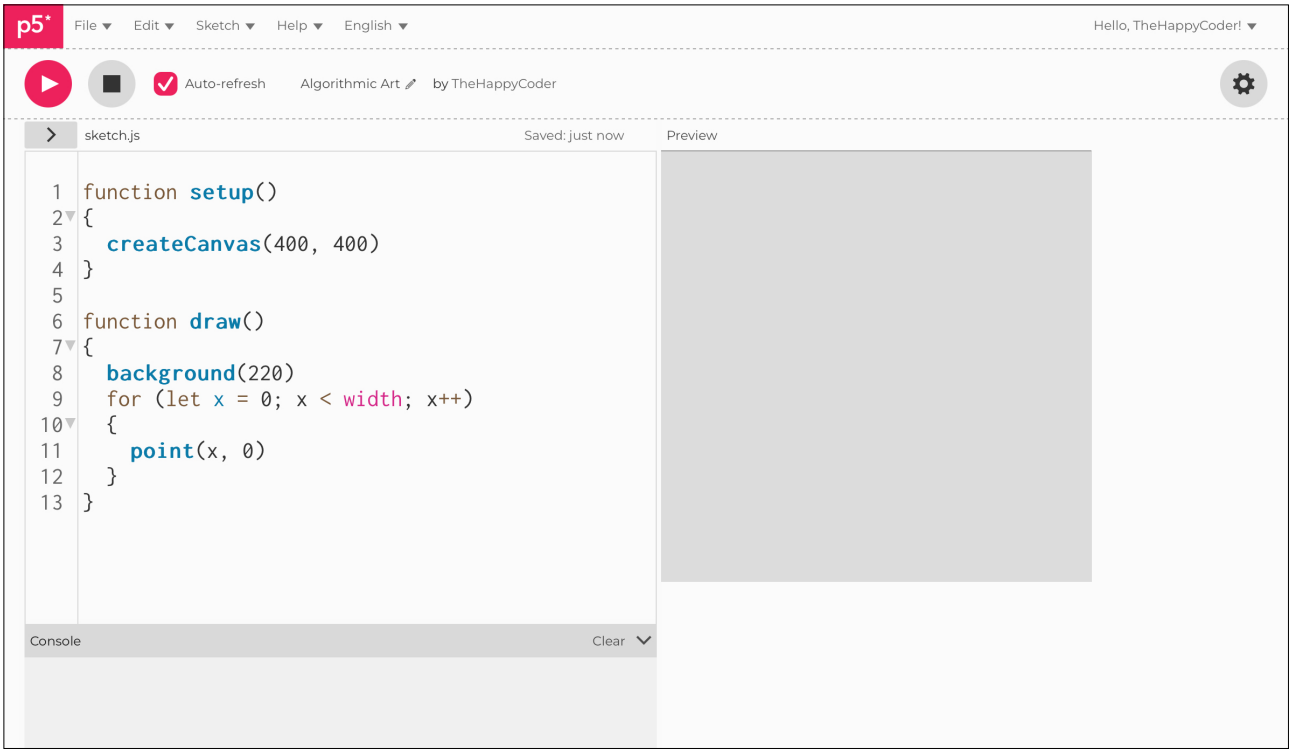
Challenge

Give it a higher `strokeWeight(10)` or so.

Code Explanation

`for (let x = 0; x < width; x++)`	A for() loop for the x value.
`point(x, 0)`	Draws a pixel for each x value.

Figure A7.2



Sketch A7.3 nested loop

Now we can use another `for()` loop inside the original `for()` loop to draw all the `y` components for each `x` component of the coordinates for each pixel. This I call a `nested loop`.

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

function draw()
{
  background(220)
  for (let x = 0; x < width; x++)
  {
    for (let y = 0; y < height; y++)
    {
      point(x, y)
    }
  }
}
```

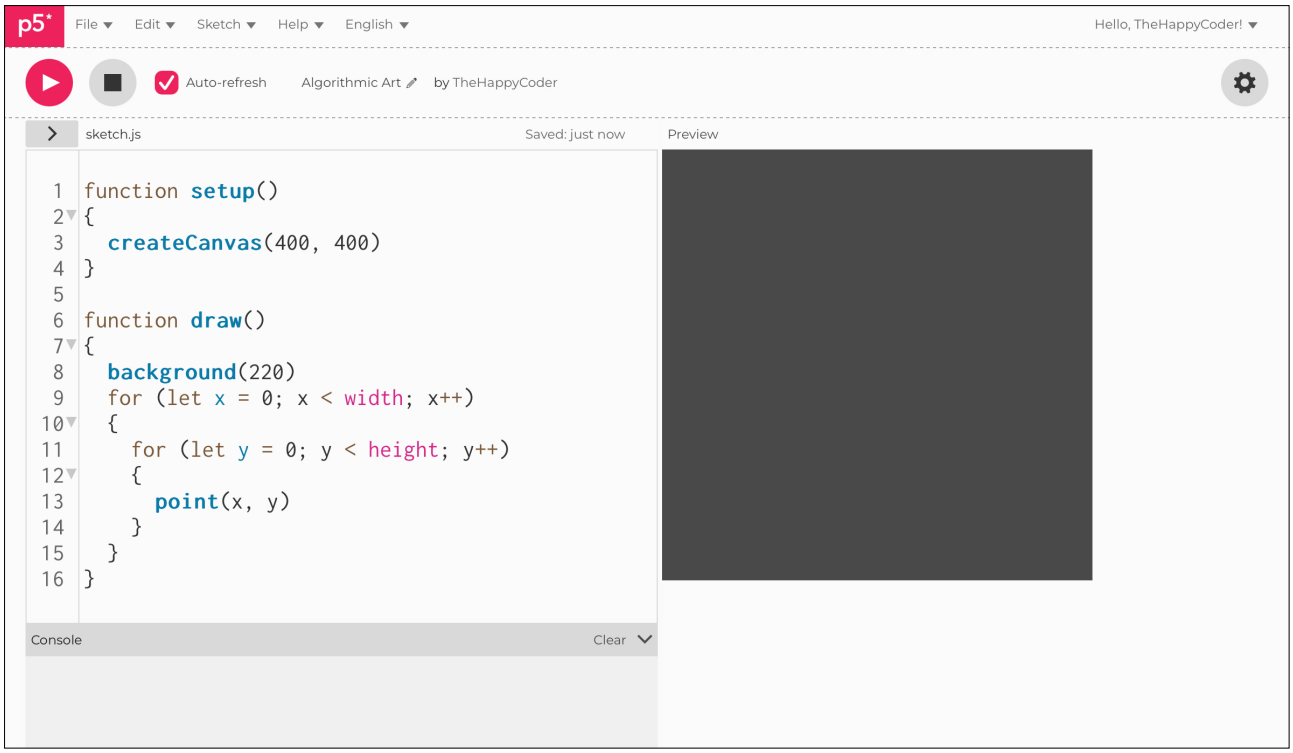
Notes

We have drawn a column of pixels working from left to right; this draws every pixel black. Remember to change `point(x, 0)` to `point(x, y)`.

Code Explanation

`for (let y = 0; y < height; y++)`	A for() loop for all the y values.
`point(x, y)`	Draw the pixel at (x, y) coordinates.

Figure A7.3



Sketch A7.4 colouring the pixel

We can give every pixel a colour using `stroke()`.

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

function draw()
{
  background(220)
  for (let x = 0; x < width; x++)
  {
    for (let y = 0; y < height; y++)
    {
      stroke('lightblue')
      point(x, y)
    }
  }
}
```

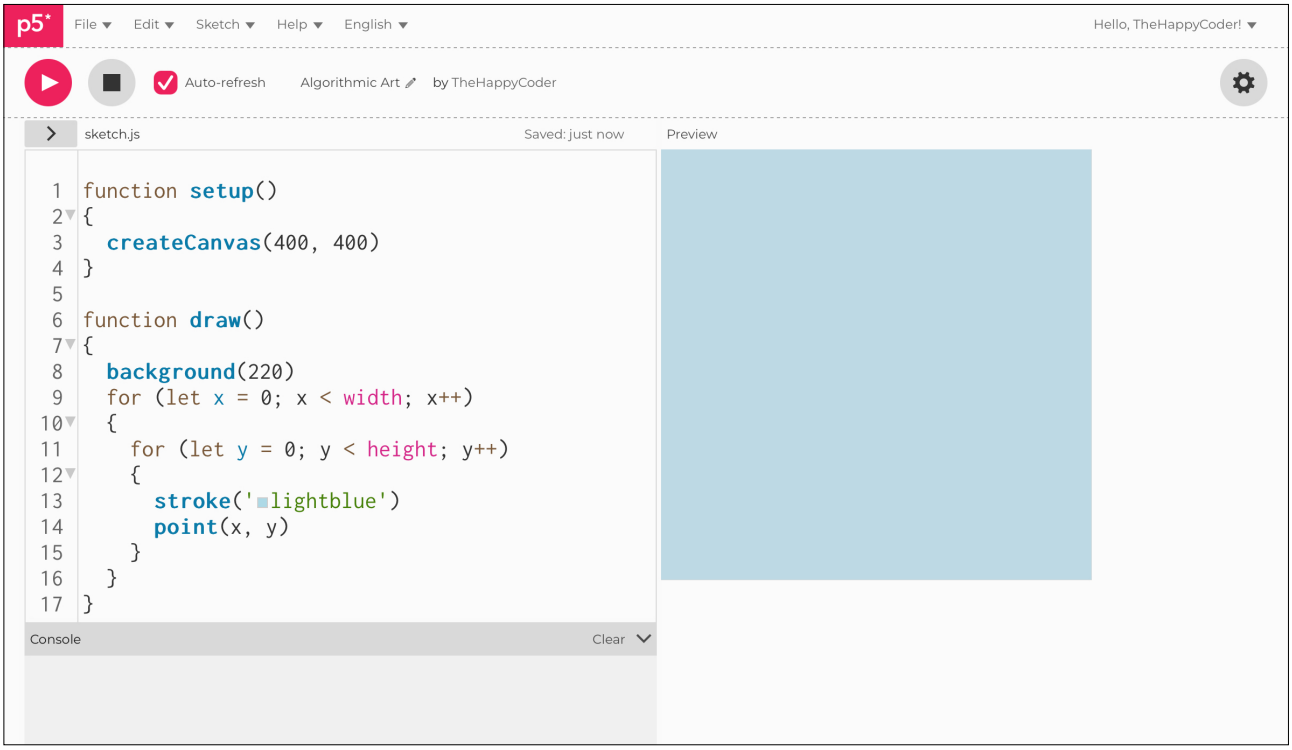
Notes

Our canvas is now a light blue colour.

Challenge

Try other colours.

Figure A7.4



Sketch A7.5 random pixel colour

Give each pixel a random greyscale colour.

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

function draw()
{
  background(220)
  for (let x = 0; x < width; x++)
  {
    for (let y = 0; y < height; y++)
    {
      stroke(random(255))
      point(x, y)
    }
  }
  noLoop()
}
```

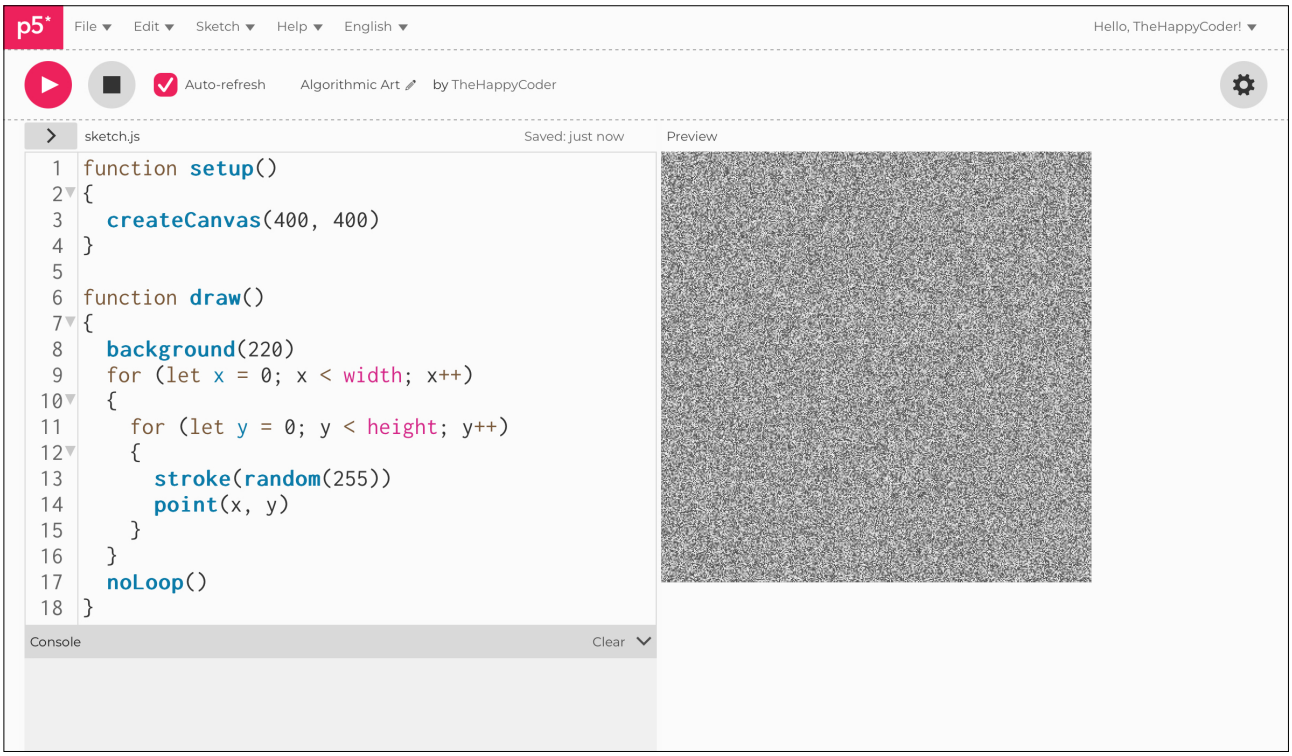
Notes

We get a random image, a bit like static on old TVs.

Challenge

Try random RGB colours.

Figure A7.5



Sketch A7.6 gradual colour

Adding some colour to the pixels gradually.

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

function draw()
{
  background(220)
  for (let x = 0; x < width; x++)
  {
    for (let y = 0; y < height; y++)
    {
      stroke(x, 0, 0)
      point(x, y)
    }
  }
  noLoop()
}
```

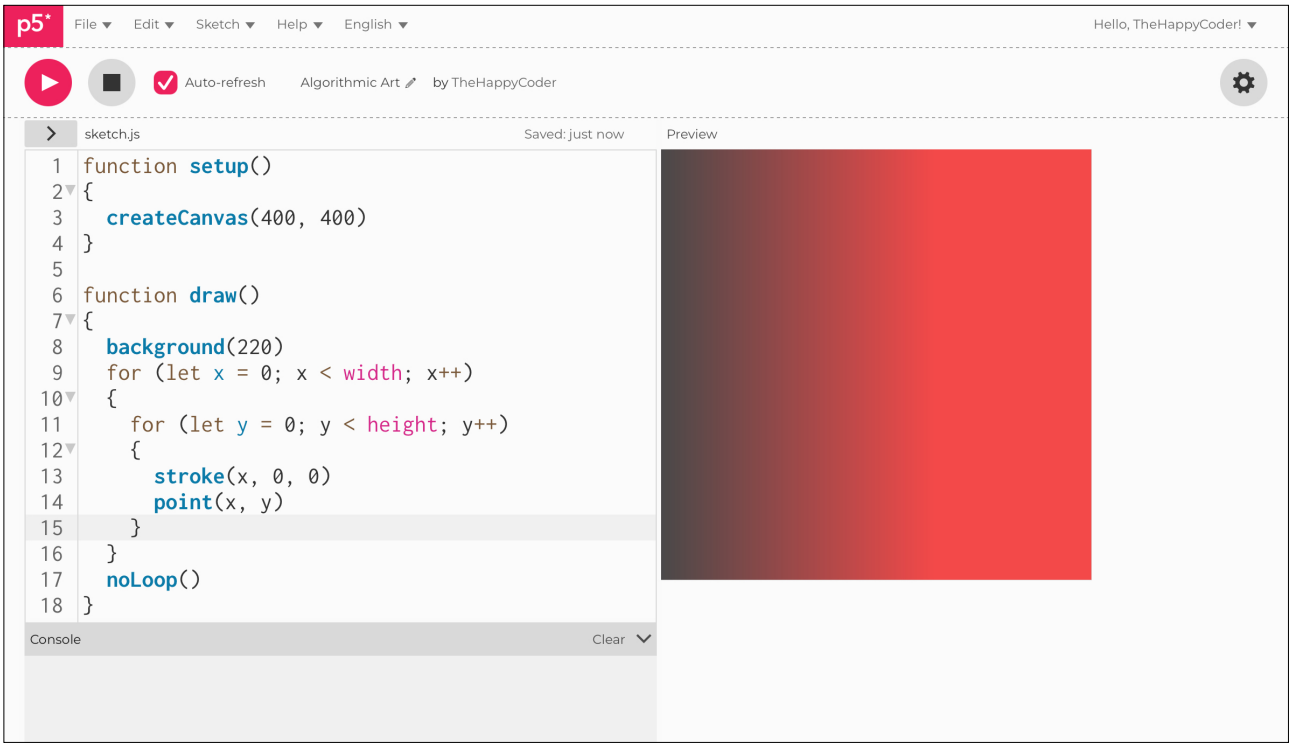
Notes

We increase the `x` value of red from `0` to `400`, but the RGB values stop at `255`; however, there is a way round this...

Challenge

Play with the values of the `stroke()` function.

Figure A7.6



Sketch A7.7 colour mode RGB

The default `colorMode()` is RGB, so that is why we don't specify (more on other modes later). In this mode, we can specify the values of the red, green, and blue; they are extrapolated to any value you choose. We can give it an extra argument of `400`, as this is the dimension of the canvas.

```
function setup()
{
  createCanvas(400, 400)
  colorMode(RGB, 400)
}

function draw()
{
  background(220)
  for (let x = 0; x < width; x++)
  {
    for (let y = 0; y < height; y++)
    {
      stroke(x, 0, 0)
      point(x, y)
    }
  }
  noLoop()
}
```

Notes

Now we have a more graduated colouring of the pixels with a range of `0` to `400` (rather than `0` to `255`).

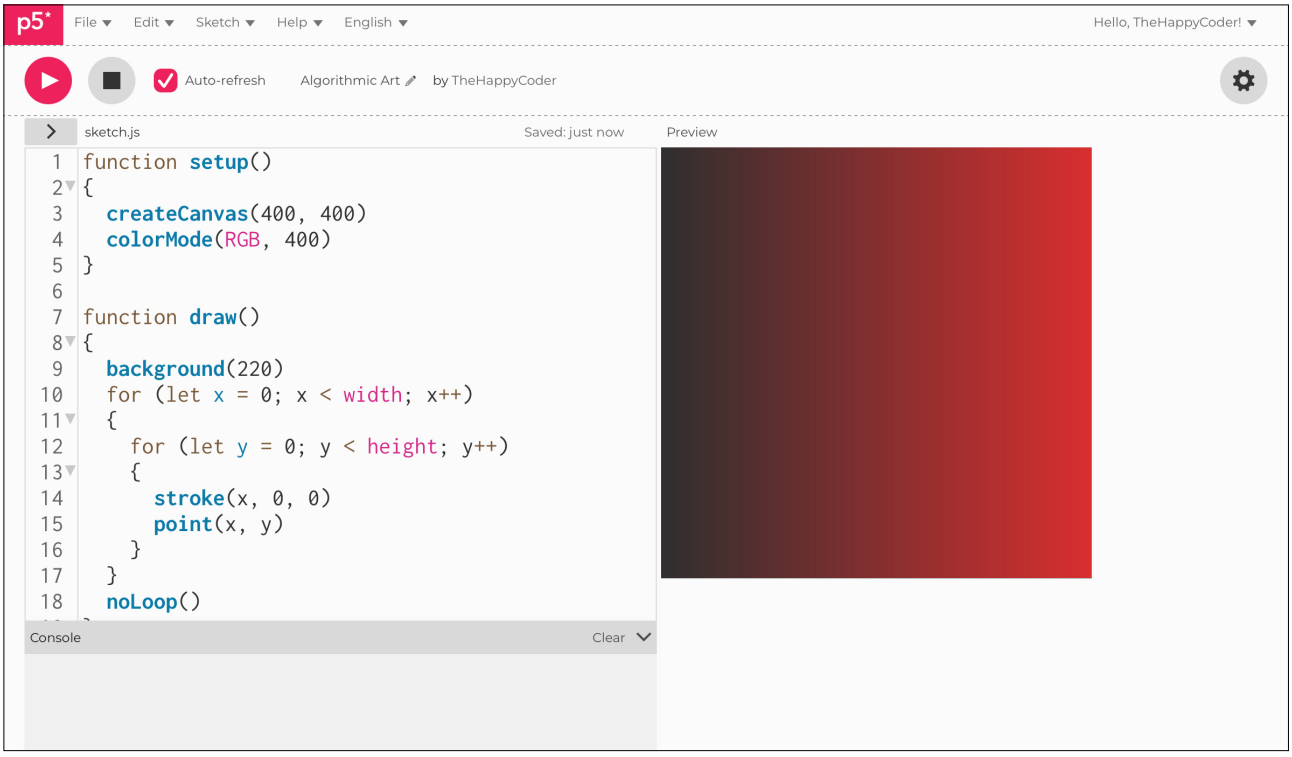
Challenges

1. Try other colours.
2. How would you start with a high value for red and work backwards?Clue: `stroke(400 - x, 0, x)`.

Code Explanation

`colorMode(RGB, 400)`	The colorMode() function allows us to change to other modes of colour and also change their properties.

Figure A7.7



Sketch A7.8 in both directions

If we add in the y values as well.

```
function setup()
{
  createCanvas(400, 400)
  colorMode(RGB, 400)
}

function draw()
{
  background(220)
  for (let x = 0; x < width; x++)
  {
    for (let y = 0; y < height; y++)
    {
      stroke(x, 0, y)
      point(x, y)
    }
  }
  noLoop()
}
```

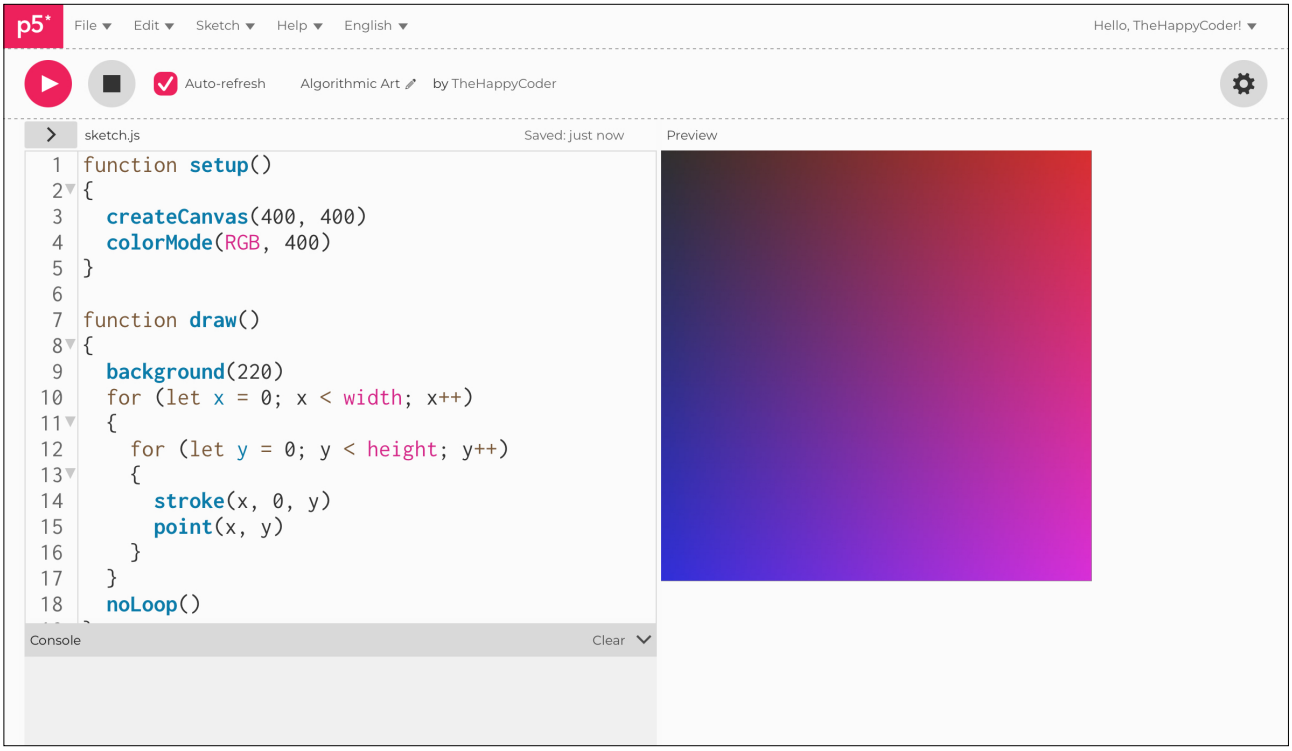
Notes

There is a lot you can do to play with these values.

Challenges

1. Try: `stroke(400 - x, 400 - y, y)`.
2. Experiment further.

Figure A7.8



The Joy of Coding Algorithmic Art Module A Unit #8 the 10PRINT pattern

Module A Unit #8: 10PRINT

It comes from an abbreviation of a single-line algorithm written in BASIC in the early 1980s: `10 PRINT CHR$(205.5+RND(1)); : GOTO 10`.

If you remember coding in BASIC, then you are possibly older than you look. This single line of code produces a random pattern, and this section uses this principle for creating random patterns.

This unit also introduces nested loops. There is such a huge scope to play with this, creating interesting designs and patterns with different shapes, colours, thicknesses, etc., so it is a great place to experiment.

Sketch A8.1 our starting sketch

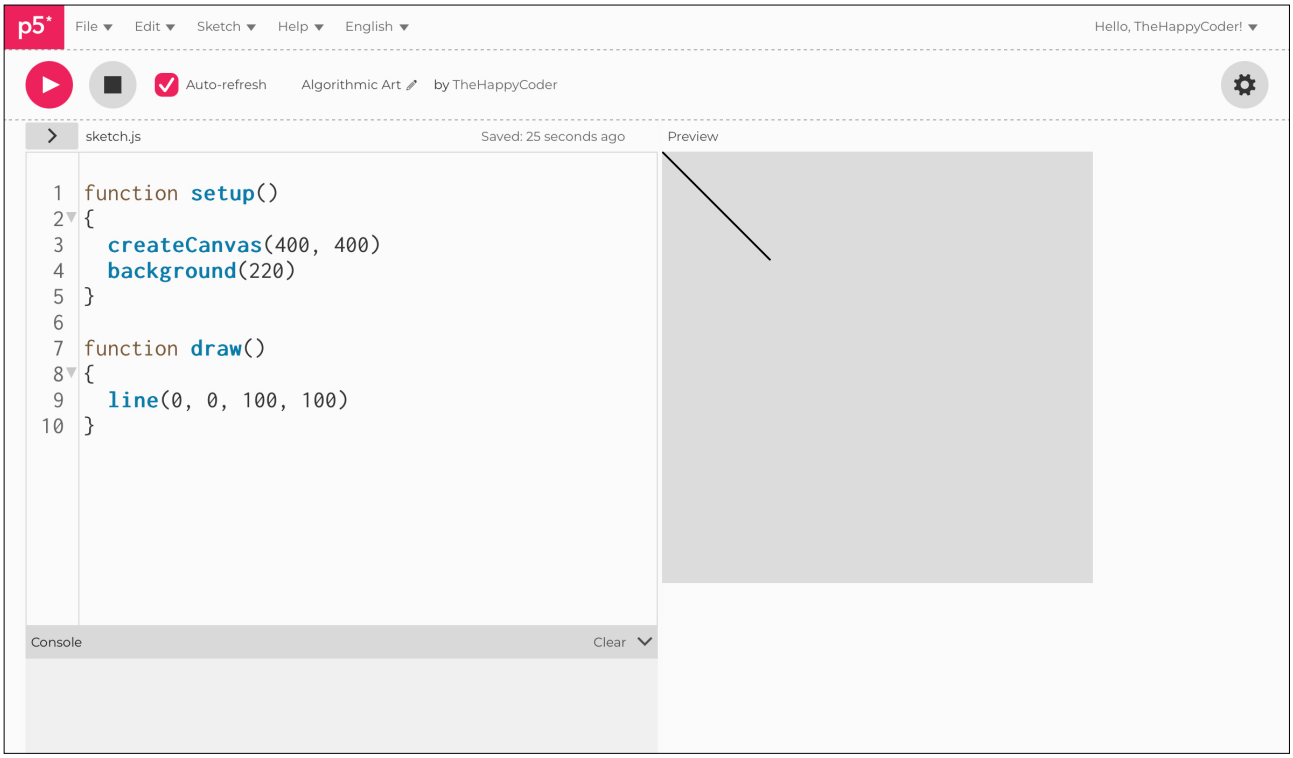
Drawing a line from (0, 0) to (100, 100).

```
function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  line(0, 0, 100, 100)
}
```

Notes

This is our starting point.

Figure A8.1



Sketch A8.2 adding some variables

Adding some variables for x and y.

```
let x = 0
let y = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  line(0, 0, 100, 100)
}
```

Notes

We will need them later.

Sketch A8.3 the line function

This does nothing for the line; yet, all will be revealed in due course.

```
let x = 0
let y = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  line(x, y, x + 100, y + 100)
}
```

Notes

You should have the same effect.

Sketch A8.4 another variable

Instead of typing 100, we will create a variable called spacing and give that the value of 100, which will give us exactly the same result.

```
let x = 0
let y = 0
let spacing = 100

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  line(x, y, x + spacing, y + spacing)
}
```

Notes

We now have easy control of the main variables.

Sketch A8.5 using the variable

Let's now change the value of the spacing variable to 10 and give the canvas a nice yellow background. Also, make the line dark red.

```
let x = 0
let y = 0
let spacing = 10

function setup()
{
  createCanvas(400, 400)
  background(255, 255, 0)
  stroke(100, 0, 0)
}

function draw()
{
  line(x, y, x + spacing, y + spacing)
}
```

Notes

Not a lot to show yet.

Challenge

Use different colours or line thickness.

Figure A8.5

Sketch A8.6 adding the spacing

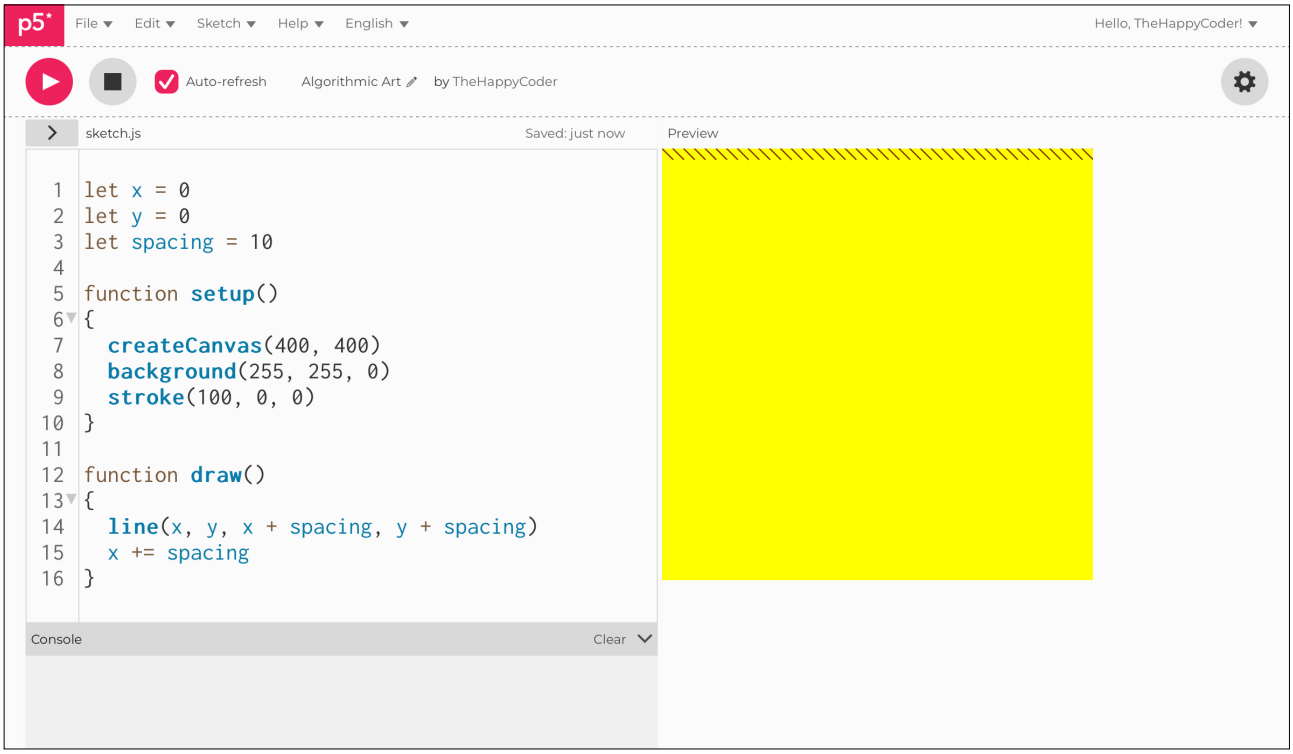
To draw evenly spaced objects in a row, we'll loop through them. You could use a fixed value like 10, but using a variable allows you to change it once at the beginning, and it will automatically apply to all instances of that variable.

```
let x = 0
let y = 0
let spacing = 10

function setup()
{
  createCanvas(400, 400)
  background(255, 255, 0)
  stroke(100, 0, 0)
}

function draw()
{
  line(x, y, x + spacing, y + spacing)
  x += spacing
}
```

Notes

This gives us a nice row of lines.

Figure A8.6



Sketch A8.7 stop at the edge

However, the lines are being drawn beyond the canvas continually. What I would like to do is to start a new line when it gets to the right-hand edge. To do this, we use an `if()` statement. An `if()` statement checks to see if something has happened, and if it has, we tell it to do something different.

```
let x = 0
let y = 0
let spacing = 10

function setup()
{
  createCanvas(400, 400)
  background(255, 255, 0)
  stroke(100, 0, 0)
}

function draw()
{
  line(x, y, x + spacing, y + spacing)
  x += spacing
  if (x >= width)
  {
    x = 0
  }
}
```

Notes

The symbol >= means greater than or equal to. So if the value of x is greater than or equal to the width, it goes back to zero. So what you are seeing is the red line starting again from the left-hand edge every time it reaches the right-hand edge. You can't see it happen because it is writing it over and over again on the same line in a continuous loop.

Challenge

You can check to see if it works by putting a smaller number instead of width.

Code Explanation

`if (x >= width)`	Checks to see if x is greater than or equal to the width.

Sketch A8.8 let's go down

But what I would like is to start a new line after it gets to the edge, and so we need to nudge it down by the same amount. To do this, we add a `spacing` value to `y`.

```
let x = 0
let y = 0
let spacing = 10

function setup()
{
  createCanvas(400, 400)
  background(255, 255, 0)
  stroke(100, 0, 0)
}

function draw()
{
  line(x, y, x + spacing, y + spacing)
  x += spacing
  if (x >= width)
  {
    x = 0
    y += spacing
  }
}
```

Notes

Every time we get to the edge, it starts a new line; however, it doesn't stop when it gets to the bottom. It is not critical, but neater if it did stop.

Code Explanation

`y += spacing`	Adds the value of spacing each time, accumulating as it goes.

Figure A8.8

Sketch A8.9 stop when we get to the bottom

We use `noLoop()` when we get to the bottom; it stops the `draw()` function from looping round. You should get the same result as before.

```
let x = 0
let y = 0
let spacing = 10

function setup()
{
  createCanvas(400, 400)
  background(255, 255, 0)
  stroke(100, 0, 0)
}

function draw()
{
  line(x, y, x + spacing, y + spacing)
  x += spacing
  if (x >= width)
  {
    x = 0
    y += spacing
  }
  if (y >= height)
  {
    noLoop()
  }
}
```

Notes

Nothing new to see yet.

Code Explanation

`if (y >= height)`	Checks to see if the y value has reached the bottom of the canvas.
`noLoop()`	Instructs any loop to stop.

Sketch A8.10 sloping the other way

Let's change it so that it draws the line sloping the other way. We can use nearly the same code.

```
let x = 0
let y = 0
let spacing = 10

function setup()
{
  createCanvas(400, 400)
  background(255, 255, 0)
  stroke(100, 0, 0)
}

function draw()
{
  line(x + spacing, y, x, y + spacing)
  x += spacing
  if (x >= width)
  {
    x = 0
    y += spacing
  }
  if (y >= height)
  {
    noLoop()
  }
}
```

Notes

Now we have some changes, not a lot but some; the lines are sloping the opposite way. We just reorganised the line function. The logic is there, just follow it through.

Figure A8.10

Sketch A8.11 the other line 50%

We want to draw one line sloping one way 50% of the time and then sloping the other way 50% of the time in a random order. To achieve that, we can simply pick a number at random from 0 to 1, and if it is less than 0.5, do one of them; else, do the other.

```
let x = 0
let y = 0
let spacing = 10

function setup()
{
  createCanvas(400, 400)
  background(255, 255, 0)
  stroke(100, 0, 0)
}

function draw()
{
  if (random(1) < 0.5)
  {
    line(x, y, x + spacing, y + spacing)
  }
  else
  {
    line(x + spacing, y, x, y + spacing)
  }
  x += spacing
  if (x >= width)
  {
    x = 0
    y += spacing
  }
  if (y >= height)
  {
    noLoop()
  }
}
```

Notes

We have an `if()...else` statement. You could use two `if()` statements, but this is a bit more elegant.

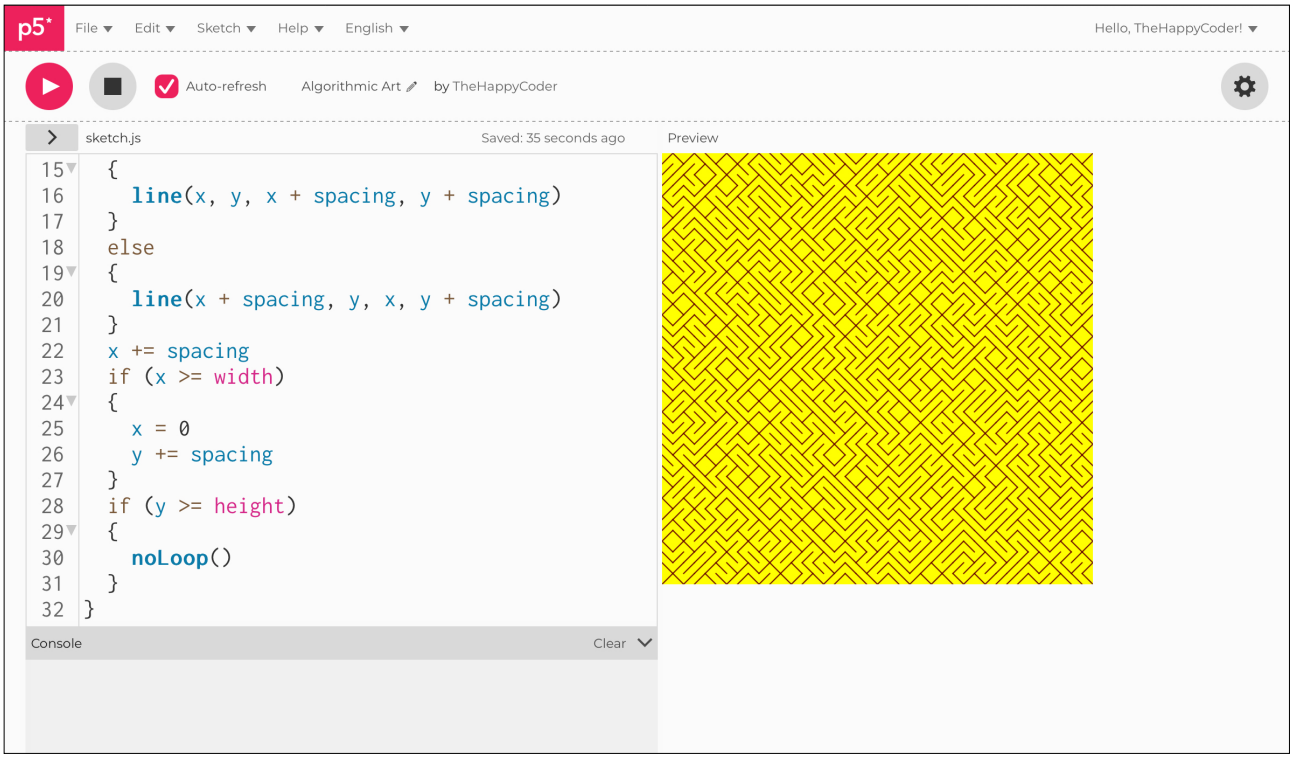
Challenge

1. What happens when you change the percentage, e.g. `0.5` to `0.8`?
2. Use other colours (random colours).
3. Thicker strokeWeight().

Code Explanation

`if (random(1) < 0.5)`	If the random element is less than 0.5, do this, or else do that.
`else`	A useful addition to the if() statement.

Figure A8.11



Sketch A8.12 variations on a theme

! Start a completely newish sketch

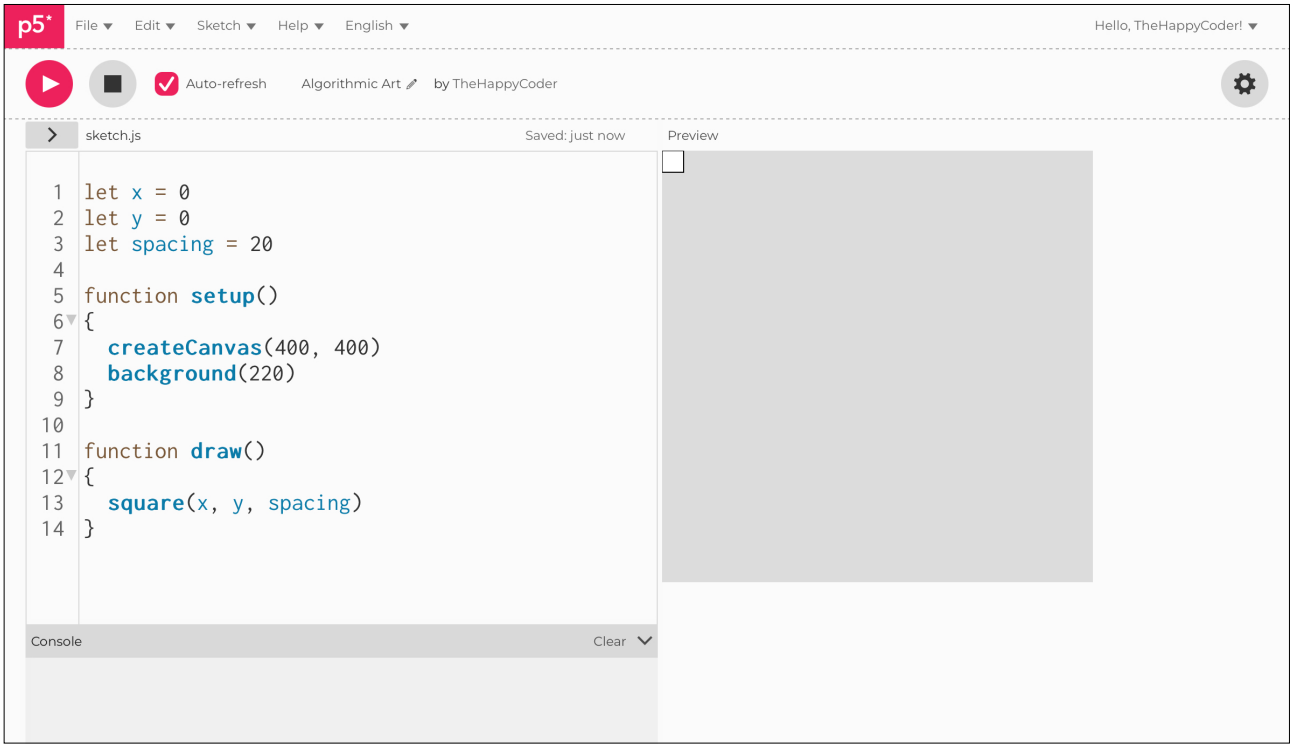
Instead of a line, we are going to draw a `square`. We will draw it at the top corner. The `square` takes three arguments: the first two are the `x` and `y` co-ordinates, and the third is the length of the side (`20`). We will use the variable `spacing` as it will come in handy.

```
let x = 0
let y = 0
let spacing = 20

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  square(x, y, spacing)
}
```

Notes

We get a single, solitary square.

Figure A8.12



Sketch A8.13 A row of squares

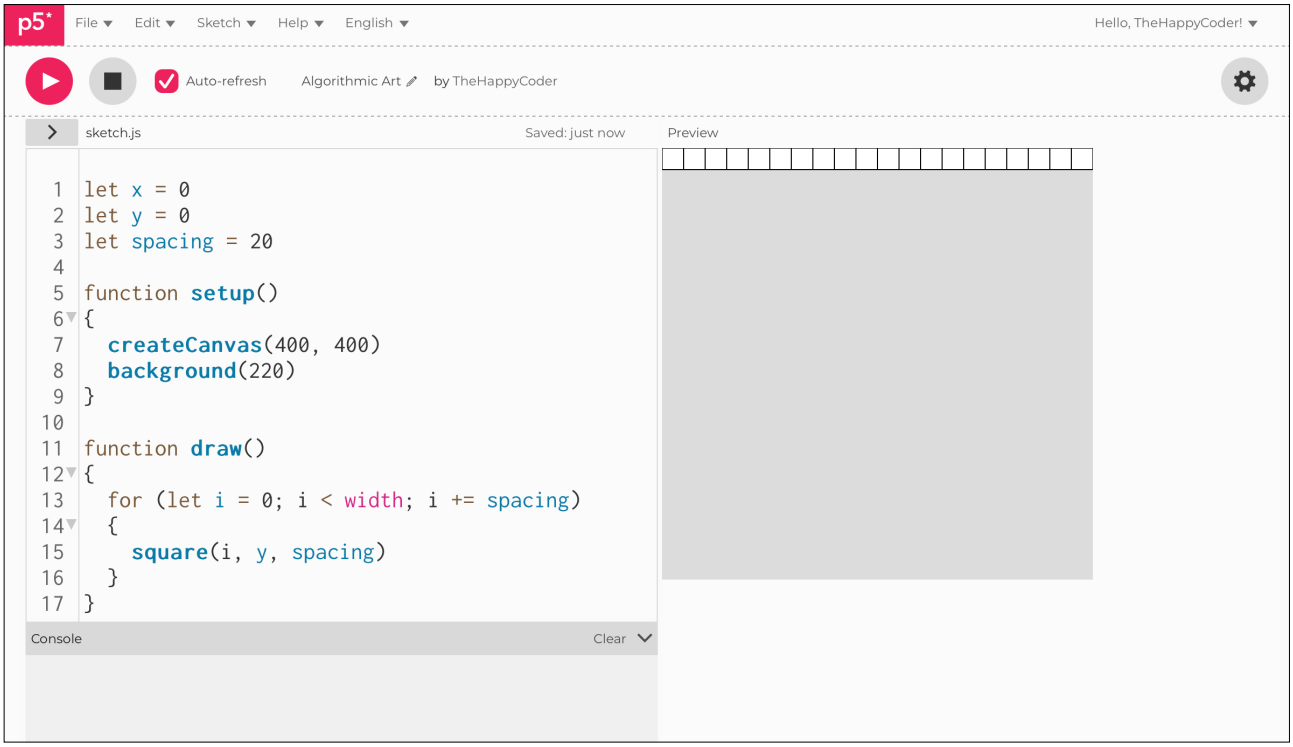
We are going to draw a row of them using a `for()` loop. A quick reminder, the `for()` loop has three parts to it. The first part (`let i = 0;`) creates a variable called `i` and gives it a value of `0`. The second part tells the loop when to stop, in this case when `i` gets to `width` (`i < width;`). The third part is how many steps (jumps) it takes (`i += spacing`). We set the `spacing` to `20`, so we are jumping in `20`s from `0` to `400` (the `width`).

```
let x = 0
let y = 0
let spacing = 20

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  for (let i = 0; i < width; i += spacing)
  {
    square(i, y, spacing)
  }
}
```

Notes

Not solitary anymore. We could've used `x` instead of `i`.

Figure A8.13



Sketch A8.14 rows and columns

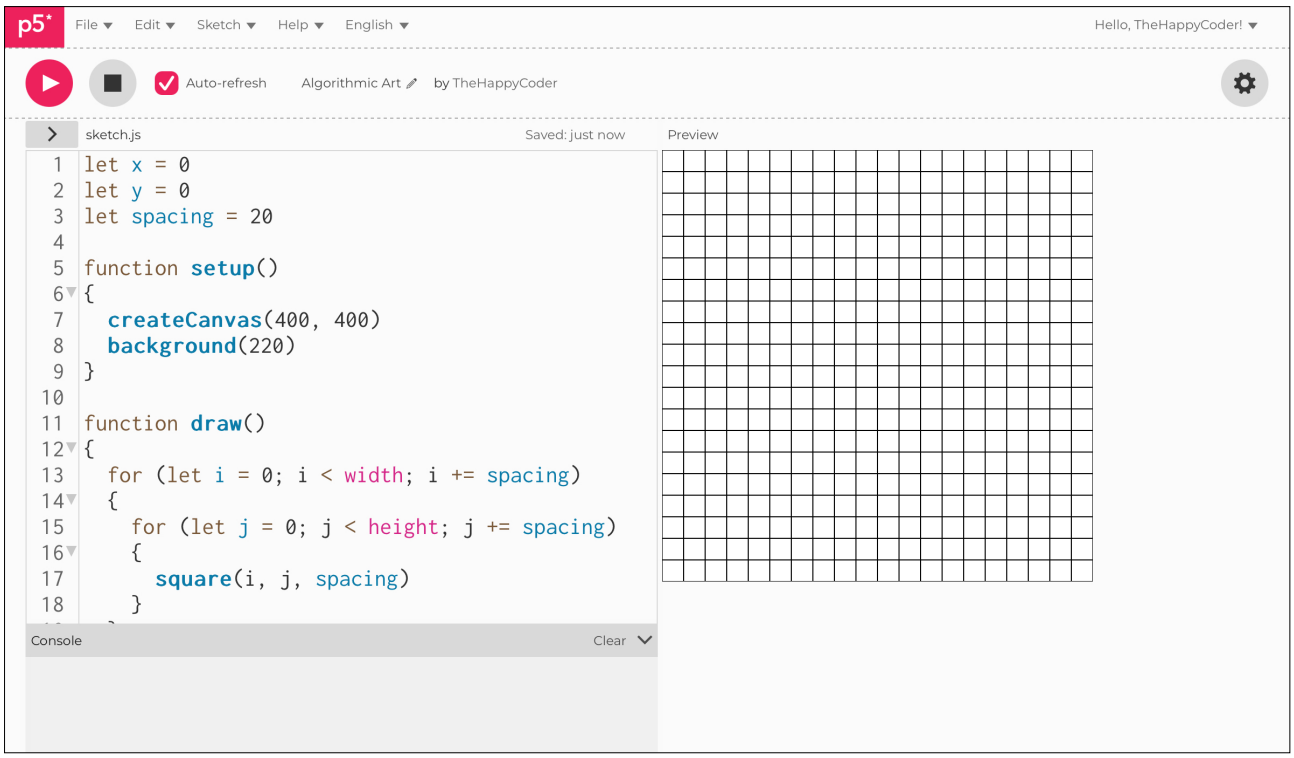
Adding columns as well as rows to fill the canvas, here we have a nested loop for the y value.

```
let x = 0
let y = 0
let spacing = 20

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  for (let i = 0; i < width; i += spacing)
  {
    for (let j = 0; j < height; j += spacing)
    {
      square(i, j, spacing)
    }
  }
}
```

Notes

A canvas full of squares. We don't need a `noLoop()` because it stops drawing them at the end of the loops. This is also where variables become very useful. The `i` and `j` are just convention; we could've used `x` and `y` as in a previous example. If you need a third variable, we could use `k` (or `z`) for a third loop.

Figure A8.14



Sketch A8.15 random size squares

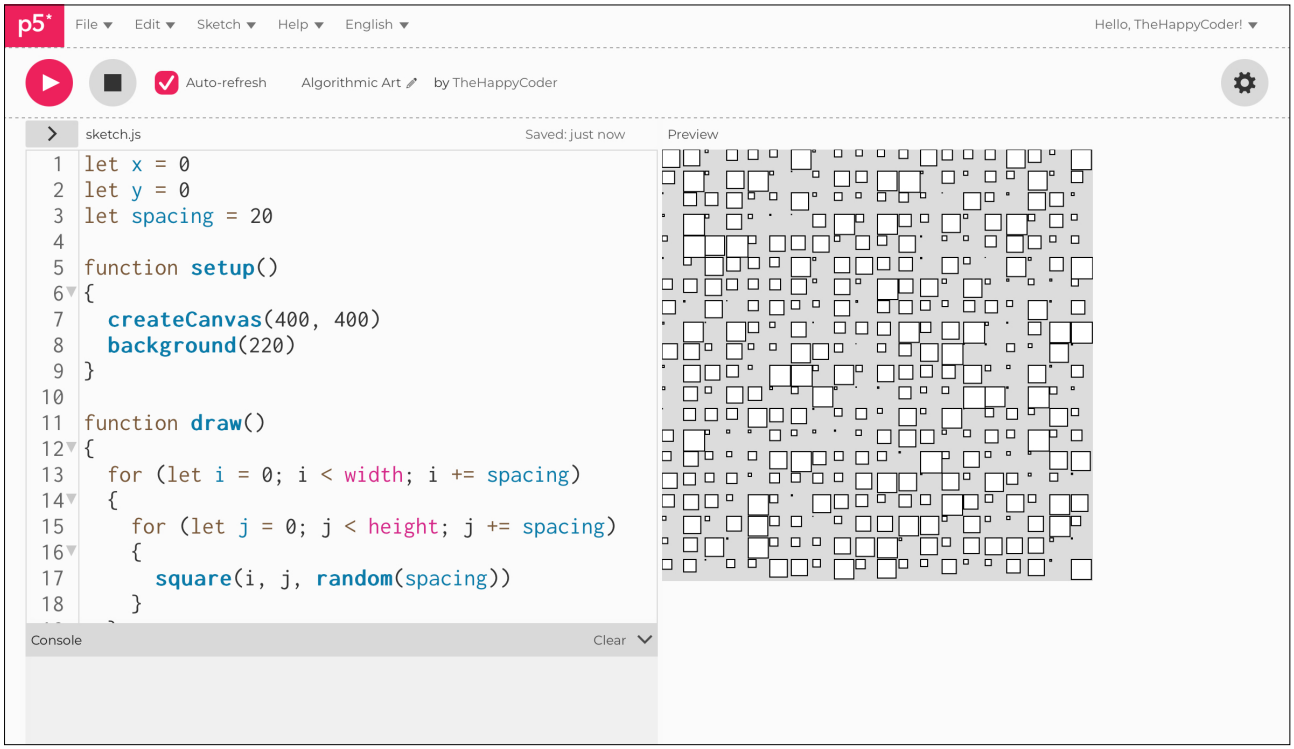
We can have some more fun with random, instead of having the same size squares. What if we change the size every time we draw one? We will need a `noLoop()` function now to stop it waggling all the time.

```
let x = 0
let y = 0
let spacing = 20

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  for (let i = 0; i < width; i += spacing)
  {
    for (let j = 0; j < height; j += spacing)
    {
      square(i, j, random(spacing))
    }
  }
  noLoop()
}
```

Notes

Nice, although it still doesn't look quite right.

Figure A8.15



Sketch A8.16 using rectMode(CENTER)

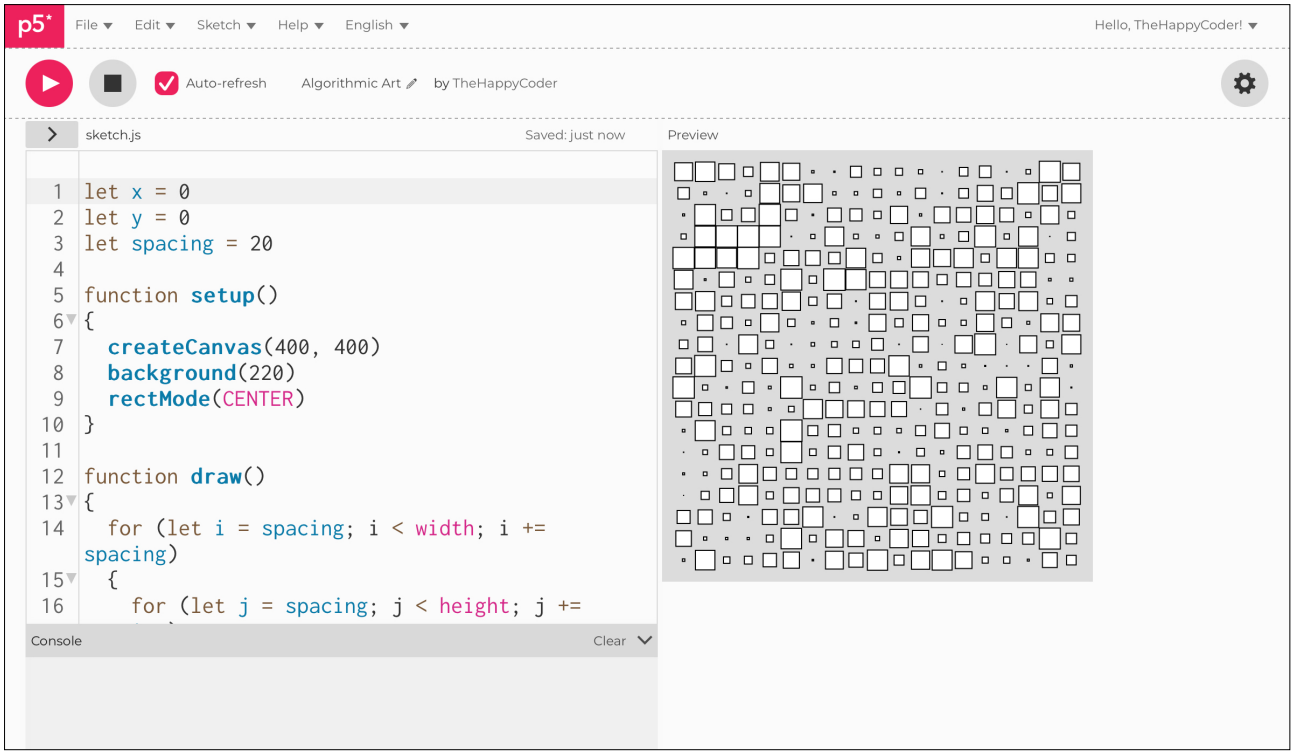
A little bit of refactoring. At the moment, the co-ordinates of the `square` are at the top-left corner of the `square`. To change that so that the co-ordinates are in the centre of the `square`, we use a function called `rectMode(CENTER)`. Also, to give it a border, we start with `spacing` for `i` and `j` rather than `0`.

```
let x = 0
let y = 0
let spacing = 20

function setup()
{
  createCanvas(400, 400)
  background(220)
  rectMode(CENTER)
}

function draw()
{
  for (let i = spacing; i < width; i += spacing)
  {
    for (let j = spacing; j < height; j += spacing)
    {
      square(i, j, random(spacing))
    }
  }
  noLoop()
}
```

Notes

Looks a lot better, but we can improve more.

Figure A8.16



Sketch A8.17 random colours

A bit more fun will be to colour them each randomly. We will also make the background white to show up the colours, adding a small gap between `random (spacing - 2)`.

```
let x = 0
let y = 0
let spacing = 20

function setup()
{
  createCanvas(400, 400)
  background(255)
  rectMode(CENTER)
}

function draw()
{
  for (let i = spacing; i < width; i += spacing)
  {
    for (let j = spacing; j < height; j += spacing)
    {
      fill(random(255), random(255), random(255))
      square(i, j, random(spacing - 2))
    }
  }
  noLoop()
}
```

Notes

Looking good.

Challenge

Try some other shapes.

Figure A8.17