

Algorithmic Intelligence The Joy of Coding Workbook #1 Your First Model

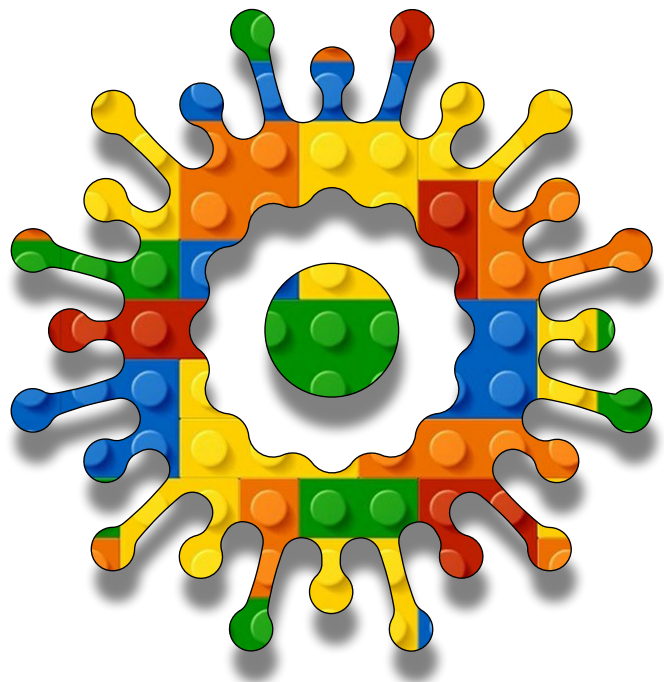




Table of Contents

Module A Unit #1: introduction to algorithmic intelligence	7
What is artificial intelligence?	8
What is a machine learning algorithm?	9
What is a neural network?	10
Nodes and weights	11
Inside a node	13
Weighted sum	14
Glossary of terms found in a Neural Network	16
Module A Unit #2: What is p5.js?	23
What machine do I need?	24
The editor	25
Signing in (or logging in)	27
The default sketch	29
Which format?	31
Looking at the buttons	33
The run and stop buttons	34
Naming the sketch	35
The auto refresh	36
Making the most of the general settings	37
Being more accessible	39
The main menu tabs	41
The File menu	42
The Edit menu	45
The Sketch menu	46
The Help menu	47
The purpose of the console	48
Accessing the sketch files	49
Adding files	51
The index.html file	54
Latest version of p5.js	55
Adding a new file to index.html	58
Uploading files	59
Creating folders	63
Final words on this section	65
Module A Unit #3: overview of ml5.js	67
Guide to adding ml5.js	68
Latest version of ml5.js	69
The index.html file	71
The ml5.js neural network	72

The task in hand	73
The debug tool	74
The Datasets	75
The callback() function	76
The Hyperparameters	77
Epochs	78
The size of the batch	79
The hidden layers	80
How many nodes (neurons)?	81
Activation functions	82
The Sigmoid activation function	83
The ReLU activation function	84
The ml5.js functions	85
Overfitting and underfitting	86
Module A Unit #4: p5.js code snippets #1	88
Sketch A4.1 function setup()	89
Sketch A4.2 function draw()	91
Canvas dimensions	94
Sketch A4.3 createCanvas()	95
Sketch A4.4 background()	97
Sketch A4.5 RGB colours	99
Sketch A4.6 a circle()	101
Sketch A4.7 fill()	103
Sketch A4.8 strokeWeight()	105
Sketch A4.9 stroke()	107
Sketch A4.10 removing the line	109
Introduction to variables	111
Sketch A4.11 let there be variables	112
Sketch A4.12 width and height	114
Sketch A4.13 mouseX and mouseY	116
Sketch A4.14 moving the background	118
Sketch A4.15 changing the values	120
Symbols we use	122
Arithmetic Operators	123
Comparison Operators	124
Logical Operators	125
Assignment Operators	126
Maths Functions	127
Sketch A4.16 const	128
Sketch A4.17 for loop	130
Sketch A4.18 nested loop	132

Sketch A4.19 random()	134
Sketch A4.20 if() statement	136
Sketch A4.21 drawing a line	138
Sketch A4.22 stroke() and strokeWeight()	140
Sketch A4.23 point()	142
Functions	144
Sketch A4.24 creating a function	145
Sketch A4.25 calling a function	147
Sketch A4.26 creating a button (step 1)	149
Sketch A4.27 creating a button (step 2)	151
Introducing arrays	153
Sketch A4.28 an empty array	154
Sketch A4.29 an array of objects	157
Sketch A4.30 createVector()	160
Sketch A4.31 map() function	163
Sketch A4.32 sin() function	165
Sketch A4.33 alpha	167
Sketch A4.34 text	169
Sketch A4.35 variable text	171
Sketch A4.36 the AND gate	173
Sketch A4.37 the OR gate	175
Sketch A4.38 p5.Vector.sub() function	177
Sketch A4.39 p5.Vector.add() function	179
Sketch A4.40 a bit more mouse	181
Sketch A4.41 mouseDragged() function	183
Sketch A4.42 mousePressed() function	185
Sketch A4.43 mouseReleased() function	187
Final note on this section	189
Module A Unit #5: linear regression with ml5.js	191
Backend Stuff	192
Introducing ml5.js	193
The index.html file	194
Sketch A5.1 starting sketch	195
Sketch A5.2 drawing the points on a line	196
Sketch A5.2 initial sketch with data points	198
Sketch A5.3 sketch revised	199
Sketch A5.4 revising the data	200
Sketch A5.5 adding the model	202
Sketch A5.5 adding the model	203
Sketch A5.6 adding the data	205
Sketch A5.6 adding the data	207

Sketch A5.7 training on the data	209
Sketch A5.8 predicting the result	212
Sketch A5.9 the callback function	214
Sketch A5.10 batch size	218
Sketch A5.11 increasing the batch size	222
Final note on this section	225
Module A Unit #6: sine wave regression with ml5.js	227
The index.html	228
Sketch A6.1 sine wave sketch	229
Sketch A6.2 epochs	233
Sketch A6.3 layers	237
Sketch A6.4 more hidden layers	241
Sketch A6.5 more nodes	245
Sketch A6.6 learning rate	249
Sketch A6.7 always a bit of trial and error	254
Activation functions	258
Sketch A6.8 sigmoid activation function	259
Sketch A6.9 tanh activation function	263
Final challenges and summary	266

Algorithmic Intelligence Module A Unit #1 Introduction to AI



Module A Unit #1: introduction to algorithmic intelligence

You may be wondering why I refer to **Algorithmic Intelligence** and not **Artificial Intelligence**. There is a really good reason for this, it is about the modern use of the term AI (Artificial Intelligence) with the rise of Large Language Models such as ChatGPT, Gemini and Claude to name a few. People now talk about LLMs when they talk about AI, hence I wanted to distinguish the difference.

These workbooks do include a unit of Language Models but small ones called (surprisingly) Small Language Models. These are free to download and use and they work in your browser. But these workbooks are more to do with Machine Learning than Large Language Models (which use transformers). Despite the same generic name and application they are quite different and work very differently.

So I will call this series of workbooks Algorithmic Intelligence as it covers Machine Learning and Language Models. However I will use them interchangeably to save time and confusion.

Key concepts:

Algorithmic Intelligence
Teachable Machines
Artificial Intelligence
Machine Learning
Neural Networks
Transformers

This is why it can be very confusing from the outside and why it is easy for mistrust and fear to grow as well greed and manipulation. Information and data is the key, the new gold.



What is artificial intelligence?

Artificial Intelligence could be described as the ability for a computer to perform tasks usually done by a human, such as learning, problem-solving, and decision-making. The process is to create machines that can simulate human behaviour or intelligence.

This is a very generalised view; you will find that others will have very differing takes on this depending on how one looks at it. The main issue is that we cannot agree on the definition of human intelligence, let alone Artificial Intelligence. Machines don't work like human brains, yet we have an expectation that they do. They work in zeros and ones (binary), our brains also work on electrical impulses and, in that respect, can seem similar.

The human brain has neurons connected to lots of other neurons (86 billion). The thought is that if you can build a big enough machine, then you could start to replicate the processing power of the brain. The rise of Large Language Models (LLMs) like ChatGPT and Gemini has fuelled this notion that we can create a highly intelligent machine capable of competing and surpassing human intelligence.

This has made some people very excited and some people very anxious. The former see it as the solution to the world's problems, like climate change. The latter group sees it as an existential threat to the human race, citing the advent of what is called the singularity. This is when Artificial Intelligence has the capacity to outstrip human cognitive ability and decides to solve the world's problems by removing the main cause of the world's problems, us.

Whatever your views are on this topic, it is here to stay and will only become a bigger part of our lives. The advent of AI is more than hype (although there is certainly a lot of that too), where venture capitalists are investing billions in this industry. Nation-states are realising the power of this technology as a weapon and an opportunity for economic power.

AI or Machine Learning (ML) has so much mystique and mystery about it. Yet it is now easier than ever to use and learn how to develop your own applications from scratch. This is why I have written this tutorial (with the help of the [Coding Train](#), see references). I have tried to unpack this so that anybody can have an understanding of how a Machine Learning or AI model works.

Although there are some very big players in the AI world, that is only a tiny part of it. It is important that more people are involved in this industry, developing ideas and applications, creating imaginative solutions to problems. Like any new technology, it will attract all sorts of people, for all sorts of reasons, from all sorts of backgrounds, so why not you?

To really understand how most AI/ML models work, we need to take a close look at the [algorithm](#) which underpins it. This algorithm is called a Neural Network.



What is a machine learning algorithm?

We use the term Artificial Intelligence (AI) to cover everything from robotics, driverless cars, to facial recognition, or recommending your next Netflix show. A lot of what we call AI is actually a form of Machine Learning (ML). Machine Learning is the backbone of most AI models and is an algorithm designed to look for patterns in data.

An algorithm is nothing special in itself. It is simply coding a set of instructions, similar to a recipe to make a cake. Some algorithms are very short and simple, others are much longer and complex, but at the end of the day, it is usually just a mixture of coding and some basic maths. What some clever people have done is to combine the two, inspired by how the brain processes information.

The most common Machine Learning algorithm is called a Neural Network because of its similarity to the human brain. In reality, the brain does not behave like this, but the success of this algorithm has seen its adoption in most of the ML world. It has a surprisingly simple architecture (structure) but has been tweaked and developed as people research different ways to solve problems more efficiently. The foundation of this algorithm is a basic Neural Network.

Data scientists and software engineers use neural networks to solve a very wide set of problems from, say, robotics to the medical and financial world. Needless to say, it has not gone unnoticed by the military, and this is often where some of the concerns lie.



What is a neural network?

Although I consider the best way to learn is by doing, which is what the tutorial is all about, it won't hurt to have a simple overview of what it might look like. However, please don't worry if it seems complicated or you still don't understand; there are so many bits and pieces that make up the whole picture of how a neural network works. What we are going to do is treat it like a jigsaw and look at some of the key pieces in isolation.

The neural network is an algorithm. It is some code and some maths. You cannot actually see anything; it is what people call a black box. Despite the coding and the maths being very well understood, it is not entirely clear what is happening inside. The architecture may be relatively straightforward, but the process of how it is learning is complex and based on initial random values. How it learns is not how we learn, and very often it comes up with solutions that surprise everyone and do something unexpected. This was certainly true when DeepMind's AlphaGo beat the leading Go player; move 37 surprised everyone because a human player wouldn't have made such a move.

This is nothing sinister but a product of the complexity of the neural network. It has lots of nodes (neurons), lots of weights (random variables) connecting these nodes, which it changes incrementally to learn on the training data provided.

When you start to understand how it works, it seems relatively simple and logical. It can find patterns in data that we might miss. However, there is sometimes an assumption that it will always be perfect, which it isn't; it never is and never will be. All it does is make an approximation. This is why it is a real issue to fully rely on AI; it always needs some human oversight.



Nodes and weights

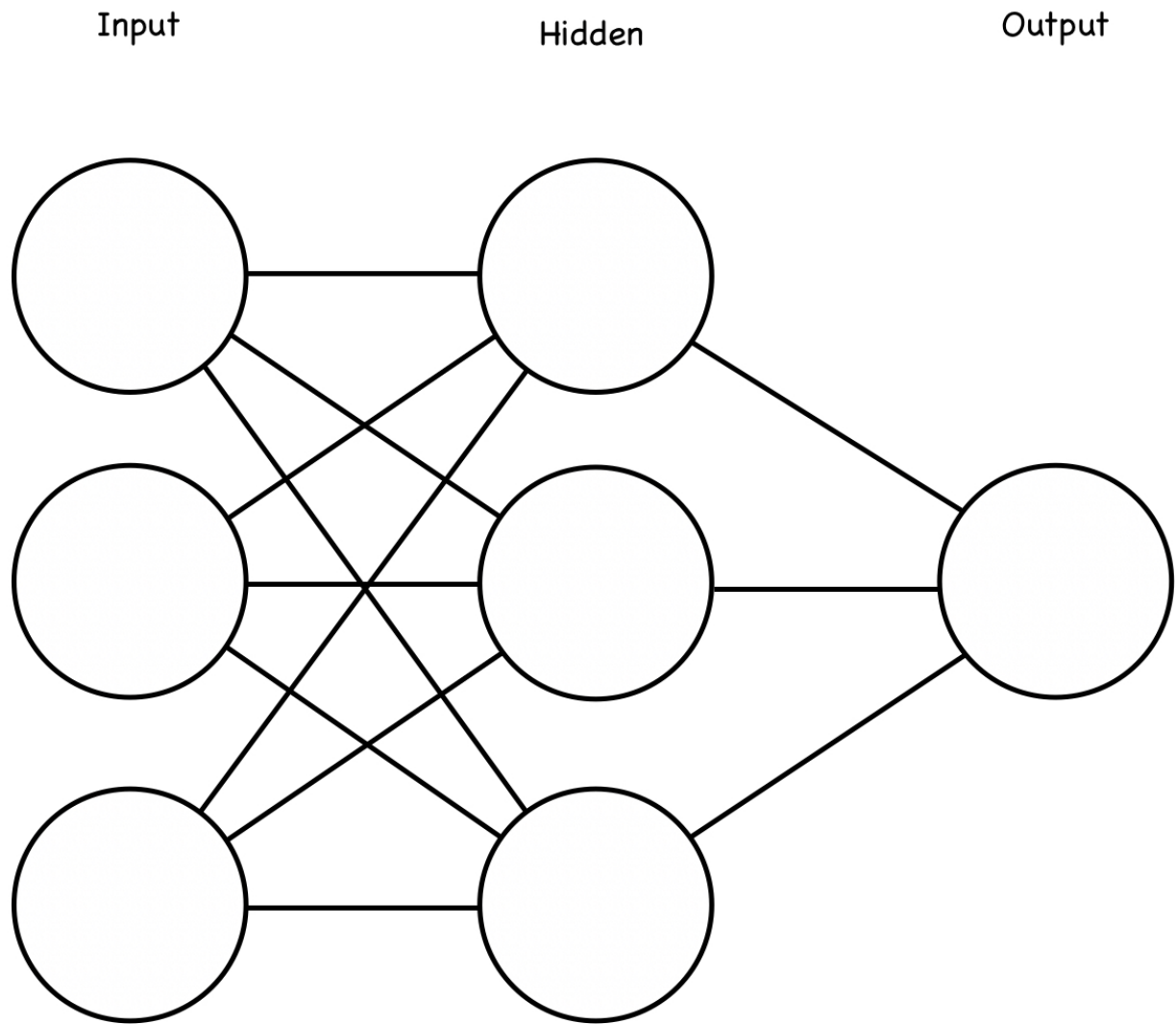
There are the two main aspects of a neural network: nodes (or neurons) and weights. The nodes are like the neurons in the brain, where they fire (or not) depending on the signals coming in. In a neural network, the nodes are connected together to form layers, where the inputs from one layer pass through into the nodes of the next layer. Each node is connected to all the nodes in the next layer; these connections are weighted.

These weights are, initially, given random values; think of them as dials or knobs that are tweaked as the neural network learns. The neural network starts off by knowing nothing and makes a wild guess based on no previous knowledge. But as the data is passed through the neural network repeatedly, the neural network changes the weights (remember they are like dials or knobs) until the guess comes closer and closer to the real answer. This is the training phase. The difference between the true answer and the neural network's guess is often called the loss.

This is an oversimplification but is the essence of how a neural network learns, by adjusting the weights based on the error between its guess and the real answer. There can be thousands and thousands of these weights, and that is why it is hard for us to know exactly what has happened inside the neural network because of the random nature of those weights.

We often draw a diagram of a neural network shown below. Where there are three types of layers: they are called the input layer, the hidden layer, and the output layer. The lines connecting them are what we call the weights.

Figure 1: diagram of a simple neural network





Inside a node

Each node has a very simple job; it takes in a single value from all the previous nodes (more on that later) and then passes it through an activation function and pushes out another value. This is similar to a neuron in the brain firing or not firing, except it takes in a value and sends out another value to the next layer of nodes.

This is really where a neural network differs from how the brain works. Neurons in the brain don't do calculations based on maths; we are still learning how the brain actually does work, which is the fascinating topic of neuroscience. What we have created is an approximation of an artificial version of the brain, which is an artificial neural network, but remember that it does not behave like a human brain. It is, at best, a mathematical and coding algorithm, not a sentient machine, at least not yet!



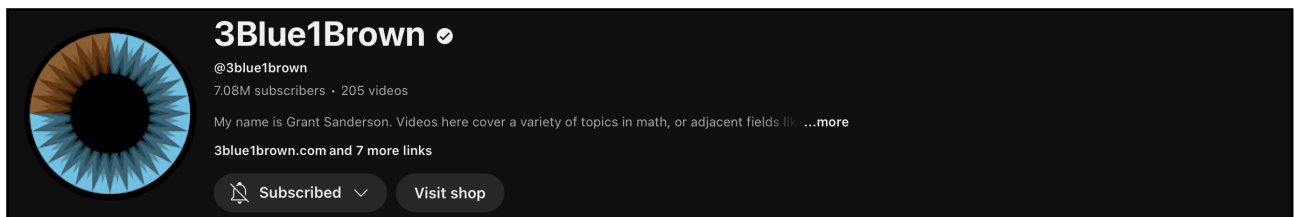
Weighted sum

Returning to our diagram, we have three layers. The data may come from a sensor, historical data, an image, or video into the input layer, and is then passed through to the hidden layer. Each bit of data is multiplied by the weight and then all are added together before being passed into each node in the hidden layer.

That is what weighted sum means; it is the sum of all the weights times the input data. From the nodes in the hidden layer, they produce a value which is also multiplied by a weight and then the receiving node adds them all up before processing the value. If this seems complicated, it isn't really; it just takes time to explain what is going on.

There is a brilliant series on the YouTube channel [3Blue1Brown](#) on neural networks which graphically explains this better than I can. I will provide a link to those videos in the [Resources](#) tab.

The best neural network explanation



The data may pass through many hidden layers before reaching the final layer, the output layer. The output layer may use a slightly different function to produce the final answer or output. This first part of the process is called **feedforward**.

There will be a difference between the guess and the real answer. This difference is sent back through the neural network, changing all the weights a tiny bit. This part of the process is called **backpropagation**.

The data is sent through again from the input layer with the newly adjusted weights to see how much error there still is. The backpropagation is repeated, changing the weights a little bit more. This is repeated again and again until the error is close to zero. The neural network model is then said to be trained and is ready for testing.

The error is often called the loss, and it uses a function to calculate it called the cost function. Even so, there are more than one cost (loss) function. You can see there is a lot to this fascinating topic, and we have barely got started.

The previous diagram of the neural network had three input nodes, with three nodes in the hidden layer and one output node. The data decides how many input and output nodes there are, but the number of hidden nodes and hidden layers is entirely up to you. This is where it becomes more art than pure science or maths.

The key elements to a neural network are:

The number of layers

The number of nodes

The weights

The activation functions

The loss (error)



Glossary of terms found in a Neural Network

Instead of going through everything in fine detail below, there are some brief descriptions on a number of key topics relating to **Machine Learning** and **Neural Networks**. You could write pages, if not books, on each of these, but for the main part, just a working knowledge is required to get started.

These are snippets of information. It is useful to be aware of their meaning and purpose before we start using them. If you are one of those who are happy to read lots before starting, then I will try to include some books, videos, and articles that might be of interest to you in the resources section of the website.

Neural Network:

Is a particular type of **machine learning** model; it usually consists of nodes (neurons) in layers. The neural network can be considered the architecture, and some architectures can be very complex.

Model:

A model is the overall process of training on patterns of data, then predicting. You might find I use the term interchangeably with neural network.

Artificial Intelligence:

This is a generic term to represent anything that mimics human behaviour, whether it uses a **neural network** or not.

Machine Learning:

This is a subset of **artificial intelligence** that often refers to the use of **neural networks**, although not always.

Deep Neural Networks:

This is a subset of **Machine Learning** that has many hidden layers, thus the term **deep** as opposed to **shallow**, where there may be just one or two hidden layers. Most neural networks are likely to be **Deep Neural Networks** (DNNs).

Hyperparameters:

They are the things or parameters you have control over, such as the number of hidden layers, the number of nodes in those hidden layers, the number of epochs, the batch size, type of activation functions, and learning rate, and so on.

Input Layer:

This is predetermined by the dataset; it can be just one, two inputs, or many thousands, such as in an image.

Hidden Layers:

This is a hyperparameter determined by you; it can be just one layer or many layers, which just depends on the complexity of the dataset. More hidden layers don't always equate to better.

Output Layer:

This can be a single node or a number of nodes depending on what the task or problem is. This is also defined by the data you are using.

Datasets:

This is the whole data you are going to use. Usually, it will include all the inputs and the corresponding outputs. The machine learning algorithm will try to establish a relationship between the inputs and outputs through adjusting the **weights**. These need to be as large as possible, error-free, and accurate. You will need to split your dataset into three components: training (80%), validating (10%), and testing (10%); all three sub-datasets need to be representative, completely different, and only used for those tasks of training, validating, and testing.

Types of Learning:

There are three main types of learning:

- A) **supervised learning**,
- B) **unsupervised learning**, and
- C) **reinforcement learning**.

A) Supervised Learning:

This is a dataset that has inputs and labelled corresponding outputs; it learns to match the inputs to the output through a neural network.

B) Unsupervised Learning:

This only has the **input data**; it has no output data. In order to classify and learn, it has to find patterns or features in the data and then group them according to those features. For instance, if you show it pictures of cats and dogs (with no labels), it might, hopefully, start to group the animals according to their similar features.

C) Reinforcement Learning:

This is where it has no dataset at all to work with; it is often just a **reward system** for making certain decisions. It will explore, possibly randomly, to see which actions give the best rewards and learn from that. This is often used in training robotics.

Tasks:

The task or problem you are trying to solve can broadly be categorised into one of two broad headings: **Classification** or **Regression**.

Classification:

This is to classify according to **labels**. For instance, classifying images of cats and dogs after being trained on thousands of labelled images of cats and dogs. It then predicts the likelihood or probability that an image presented to the model is either a cat or a dog.

Regression:

This usually gives you a value, for example, the price of a house after being trained on a dataset that has **features** based on many houses, their sold price plus all other features such as number of bedrooms, garage, size of garden, distance to nearest train station or school, and so on.

Feedforward:

This is the process when all the data is passed forward through the **neural network**; this starts at the input layer, then goes through the hidden layer(s), and finally to the output layer, with the final prediction.

Backpropagation:

Once the **feedforward** process is complete, the loss is calculated and then that is passed back through the **neural network**, tweaking all the weights before beginning the whole process again.

Fully Connected:

This means that all the nodes from the previous layer are connected to all the nodes in the next layer. Also called **Dense**.

Weights:

These are random values, usually between **0** and **1** (or **-1** and **+1**). One way to think of these is as the connections between the nodes in one layer with the nodes in the next layer. These are adjusted each time backpropagation takes place.

Bias:

To stop a scenario of nodes or layers becoming zero, which means it cannot be trained, each layer is given an extra node called a **bias**, which usually has a value of **1** to keep the neural network from collapsing.

Training:

This is done on the bulk of the dataset. It is trained on it until the loss is near to zero. You set the **hyperparameters** to what you think are most likely going to give the best result, avoiding **underfitting** and **overfitting**.

Validating:

This is the dataset you use after you have finished the training phase. It is set apart from the training dataset. This is an opportunity for you to see which hyperparameters need adjusting to get an even better result. This dataset is used to check the accuracy of the training model with previously unseen data. After you have changed the hyperparameters, the model is retrained again, followed by another validating process.

Only when the results are acceptable do you move onto the next phase of testing the model.

Testing:

Once you have trained the model on the training dataset, and you have validated it through adjusting the hyperparameters, you save the model and test it. This uses the final sub-dataset and will highlight if the model works well or underfits/overfits. The test dataset should not be used again. The testing phase is the final check before using the model for real. The model won't be perfect, but you will have to decide whether it is good enough to use in the real world.

Epochs:

This is when a complete training dataset has passed through the model once. The number of **epochs** you might pass through the neural network during training depends on the loss; there is no point in training beyond a certain loss point. This is when the loss is so small that any extra time training is pointless. Choosing the number of epochs is one of the main hyperparameters.

Batch Size:

The **batch size** is the size of a chunk of data that passes through the model before the neural network is updated. It is a proportion of the total data, so if an epoch is **1000** bits of data and the batch size is **10**, it will update the model **100** times per epoch. A small batch size may improve accuracy but takes longer, and conversely, a larger batch size is less accurate but quicker. It is a bit of trial and error; also, it is another hyperparameter you have to make decisions about.

Loss or Cost Function:

The words **loss** or **cost** are interchangeable, although there is a subtle but distinct difference. This is the measurement between the prediction (or guess) and the true results, label, or output data. This goes through various mathematical functions and decides how much the weights are adjusted by. The loss function is usually shown as a chart and aims to be as close to zero as possible. There are numerous loss or cost functions; **Mean Squared Error** is one such function, which also depends on the task.

Activation Functions:

Inside each node is an activation function which takes the multiplication of the weights and input data from all the nodes in the previous layer, adds them together (weighted sums), and passes them through the activation function so that the output from that node is a single value based on the weights, inputs, and activation function. Historically, the **Sigmoid** function was always used, but now modern neural networks use **ReLU**, which is much easier computationally, which also means quicker and cheaper, and is also very effective. This is another hyperparameter, where you decide which activation functions to use and where.

Pre-trained Models:

These are models that can be used in applications such as facial recognition and voice control. The hard work (training, validating, and testing) has already been done for you, and you can use the model for whatever purpose seems appropriate.

Overfitting:

This is where the model has learned the data by heart and has not really approximated it. What this means is that when it is presented with data it has not seen before, its accuracy is very poor despite achieving a very low loss value in training. If it seems too good to be true, beware!

Underfitting:

This is usually a model that has been trained on a dataset that is too small. You may get reasonable results in training, but the test data produces poor end results.

Normalisation:

This is applied to the dataset before training begins; it changes the size of the input values, which can vary in size considerably (think of the coordinates in an image), to values between 0 and 1 or -1 and +1. Neural networks work best with values of this magnitude; it avoids the impact of a few large numbers/values having a disproportionate effect on the training process. The values of the **weights** are also of the same magnitude.

Gradient Descent:

This is where the **loss function** is calculated. It works out how much to change the weights based on the current loss. The gradient of the curve is the key component. The steeper the curve, the bigger the adjustment.

Learning Rate:

This is a hyperparameter that controls how big a step each learning iteration takes. It stops the model from taking huge steps, which might miss or overshoot the lowest loss point. Without the **Learning Rate**, it may never really settle on the minimum loss. The smaller, the better, but it takes longer and may also not reach the minimum (or stop at a local, not global minimum). If you use a large **Learning Rate**, you might overshoot.

Notes

There is a lot to this, and I have glossed over some of the fine detail and oversimplified some of the important features. At this stage, it is an introduction to some of the key elements involved and not an in-depth explanation. It will give you a flavour of this fascinating topic. Everything becomes clearer and more concrete when you start coding, building your own neural networks, and training them in this tutorial.

Next

In the next section, I want to introduce you to the environment you will code in, the web editor for p5.js. This is where you will code and see the machine learning in action, but first, I want to show you around the workings of that web editor.

Algorithmic Intelligence

Module A

Unit #2

Exploration of p5.js



Module A Unit #2: What is p5.js?

The coding language used here is called p5.js. It is a JavaScript based library that has a canvas onto which you can draw shapes, move objects and create anything digital. On top of that it can also be used for AI with another library called ml5.js.

This coding language is easy to use, yet it is a fully functioning coding language that works in your browser so no need to download any software. You type in your code, press play and the effects are seen instantly on the canvas next to the code. It is that simple and even better it is free, no hidden costs or even a need to sign up. However, I recommend that you do sign up so that you can save your work as you go along.

This unit gives you some background information on how to use p5.js, most of it is very intuitive but I recommend that you play around and explore it to familiarise yourself if you have never used p5.js before. If you haven't worked through the Workbook series called Algorithmic Art, then not to worry as I have provided a unit that introduces you to the key elements of p5.js that you will need for this series, although I would recommend that you walked through the Algorithmic Art series if you have never coded before.



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.

What you will need later on is a webcam and microphone; ones that are built-in will be fine; otherwise, you will need to connect your machine of choice to them.

! Some issues I have experienced

For the most part (90%) this will all work fine on any machine but on my iPad, I could not download the model (see relevant unit much later on); not sure why; it did on the Chromebook, and I assume it does on everything else, but if you are using a tablet or a smartphone, I cannot guarantee it. It is only one unit and is not critical, but that is the only issue I can find.



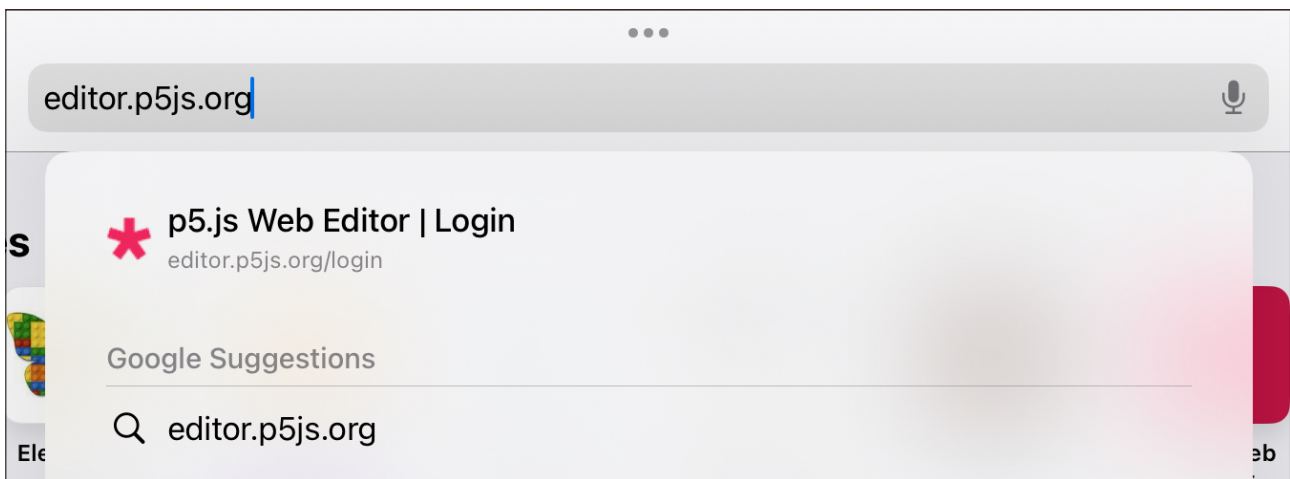
The editor

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

editor.p5js.org

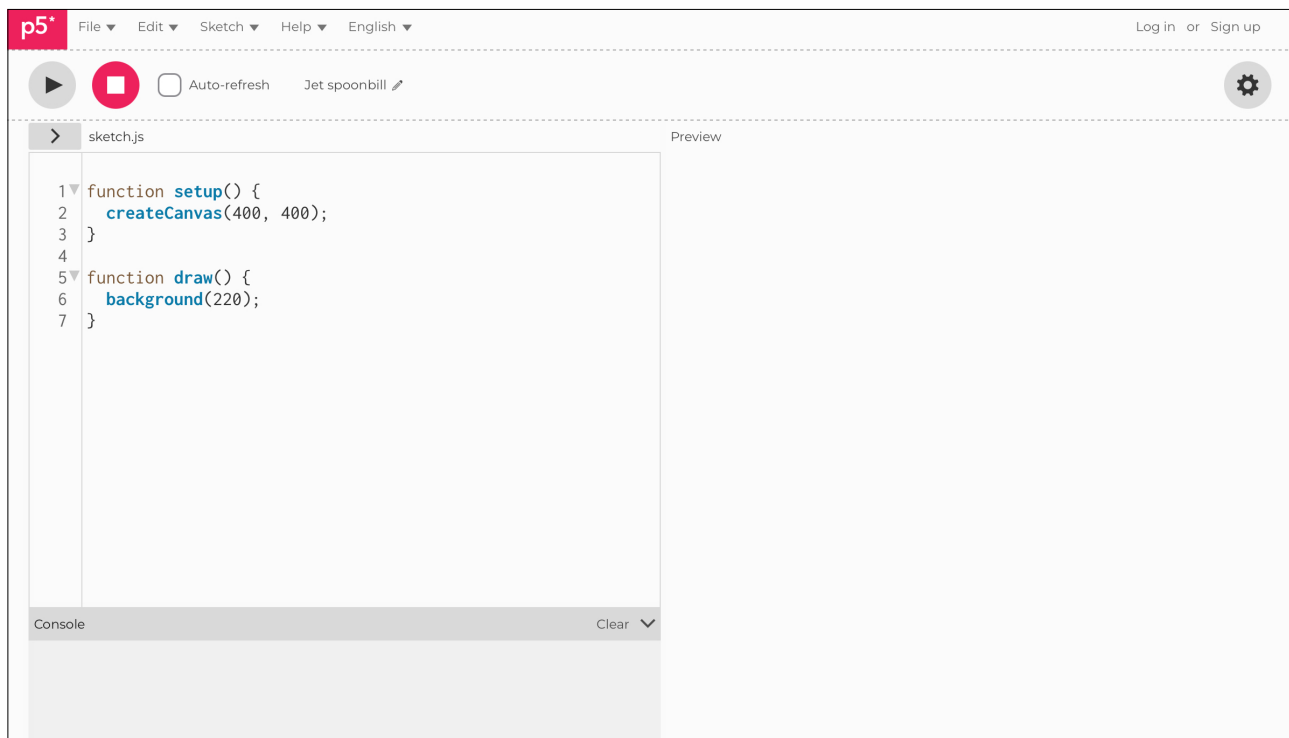
into your web browser and off you go. Most web browsers support it, but if you're unsure, then use Chrome or Safari. If you have a favourite one, then use that and see how you get on.

Figure 1: Type **editor.p5js.org**



When you first start, you get a page like this (shown below). 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.

Figure 2: The editor



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.



Signing in (or logging in)

Although you don't have to sign up for anything and you can code straight away, you won't be able to save anything unless you have an account. I would seriously 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 your own email address. 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.

Figure 3: Signing Up

p5* < Back to Editor Log in or Sign up

Sign Up

User Name

Email

Password

Confirm Password

Sign Up

Or

Login with GitHub

Login with Google

By signing up, you agree to the p5.js Editor's [Terms of Use](#) and [Privacy Policy](#).

Already have an account? [Log In](#)

Figure 4: Logging in

p5*

< Back to Editor

Log in or Sign up

Log In

Email or Username

Password

☐

Log In

or

 Login with GitHub

 Login with Google

Don't have an account? [Sign Up](#)

Forgot your password? [Reset your password](#)



The default sketch

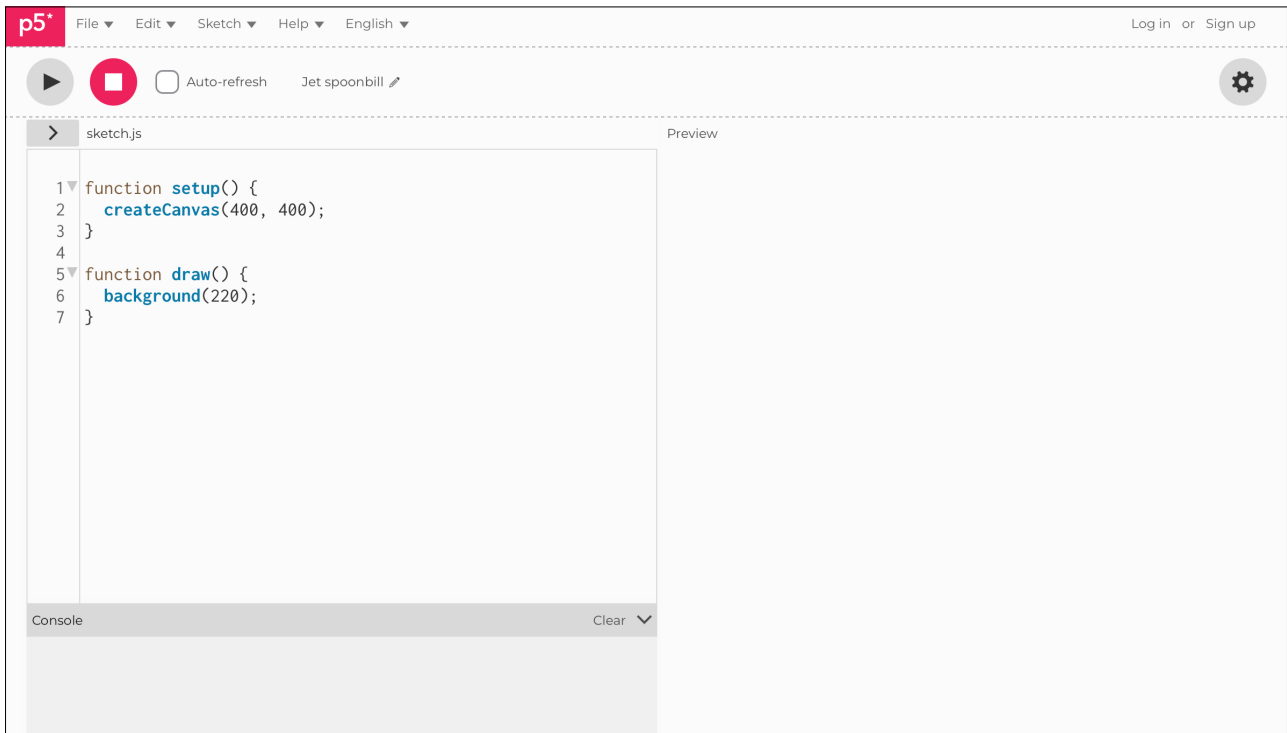
At the very start, you get a default sketch. This includes two functions and some curly brackets. These two functions are important and are built-in functions. The first one, `function setup()`, is to set up how you are going to see and use the sketch; this runs through once only and usually includes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

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.

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 everything in between. For most of my sketches, I will simply use this value (`220`) for simplicity.

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

Figure 5: Default sketch



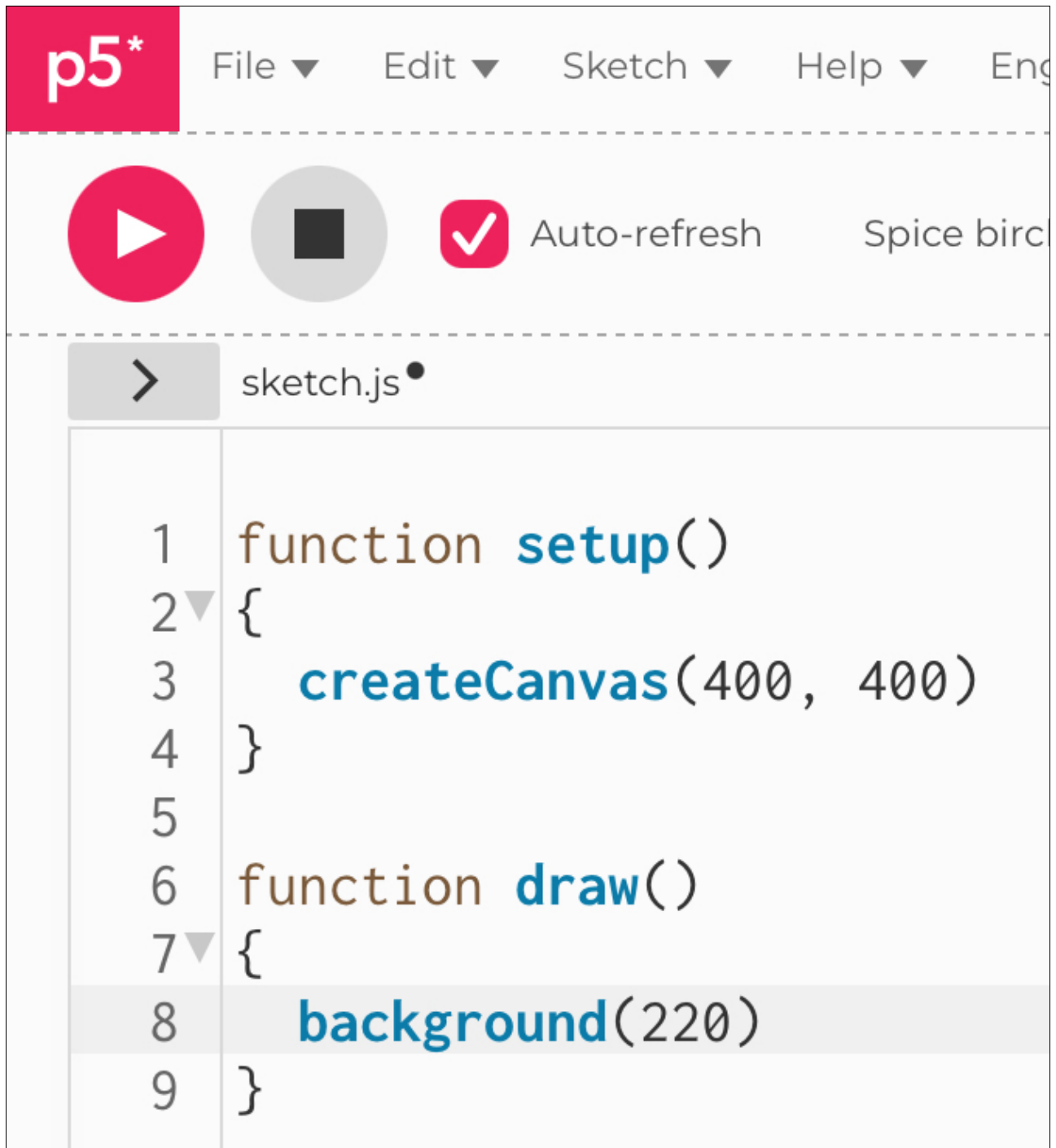


Which format?

There is the default format, but the format I will present in my tutorial will be 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 `{ }` very clear, and I change that to the first curly bracket is on the next (first) line after the name of the function. This is just my preference for clarity.

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

Figure 6: My format





Looking at the buttons

You will see a row of buttons across the top. It is worth spending the time getting to know them well. 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, occasionally your code may crash, for whatever reason. 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 also the 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 7: Buttons





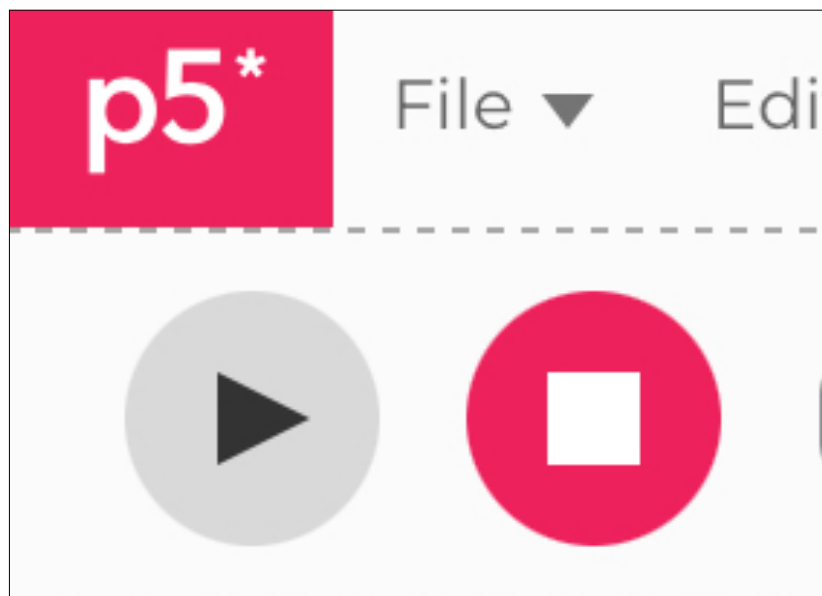
The run and stop buttons

The first button you come across is the **run** button. It looks like the play button on a device you might have seen. 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 that.

To stop the code running, you will need to press the **stop** button, which is next to it 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 8: Run & Stop





Naming the 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 9: naming sketches



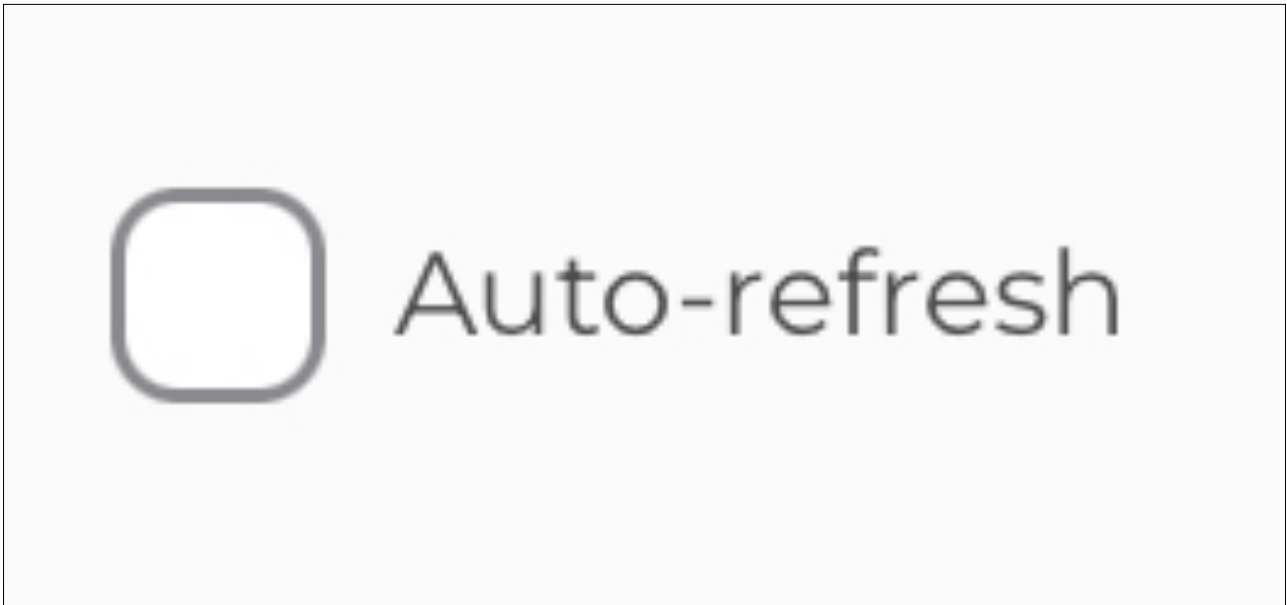
Jet spoonbill 



The auto refresh

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.

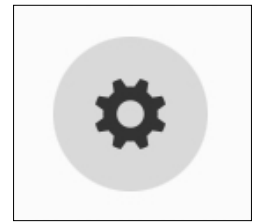
Figure 10: auto refresh





Making the most of the general settings

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:

It 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; just practice 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 Hinder:

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 11: general settings

Settings

General settings

Accessibility

Theme

Light

Dark

High Contrast

Text Size

-

20

+

Decrease

Increase

Autosave

On

off

Autoclose Brackets and Quotes

On

Off

Autocomplete Hinder

On

off

...

...



Being more accessible

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 they can often tell you on which line of code the error message relates.

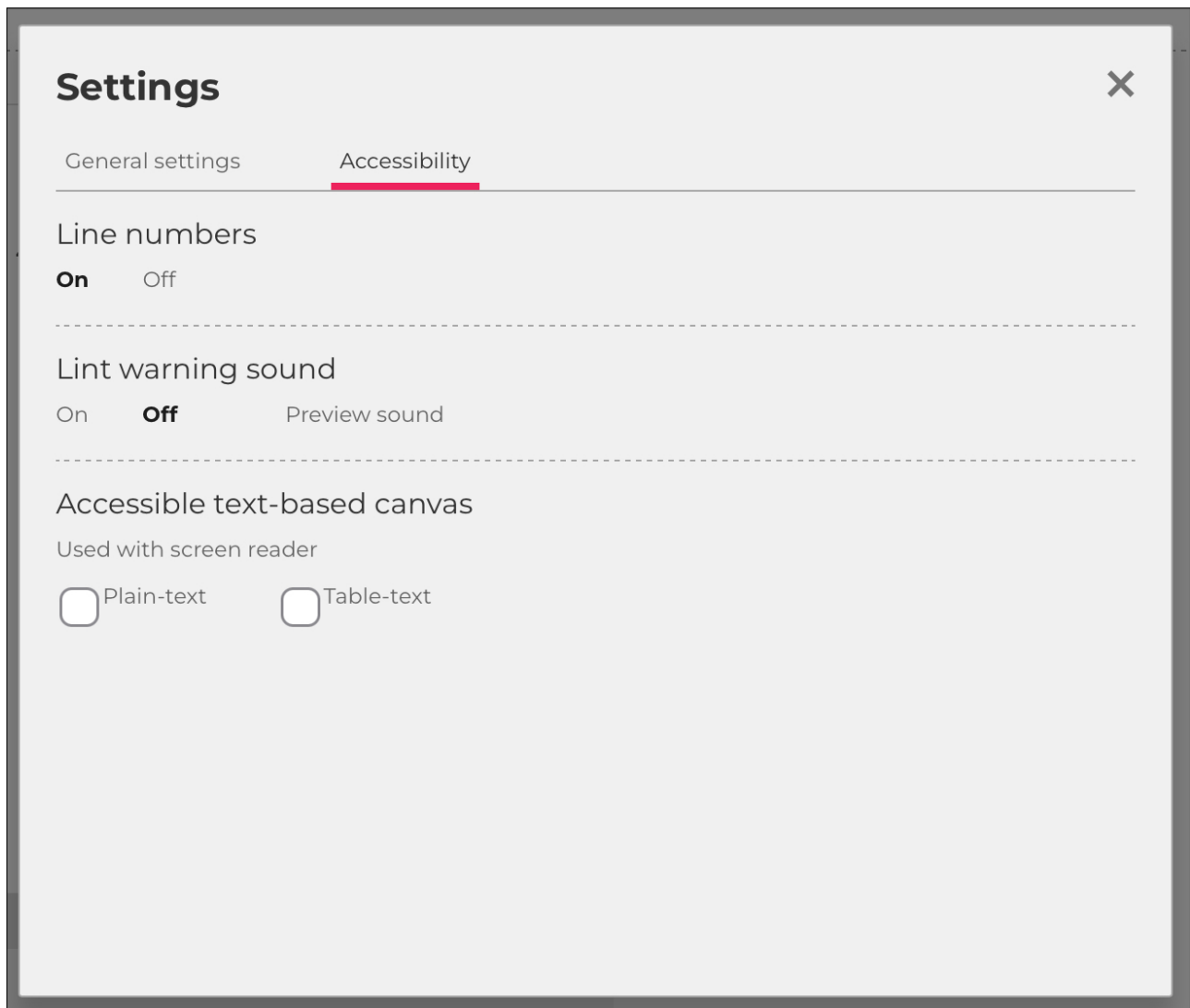
Lint Warning Sound:

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 12: accessibility





The main menu tabs

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 13: main menu





The File menu

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:

This is something in the web editor. 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 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 on 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.

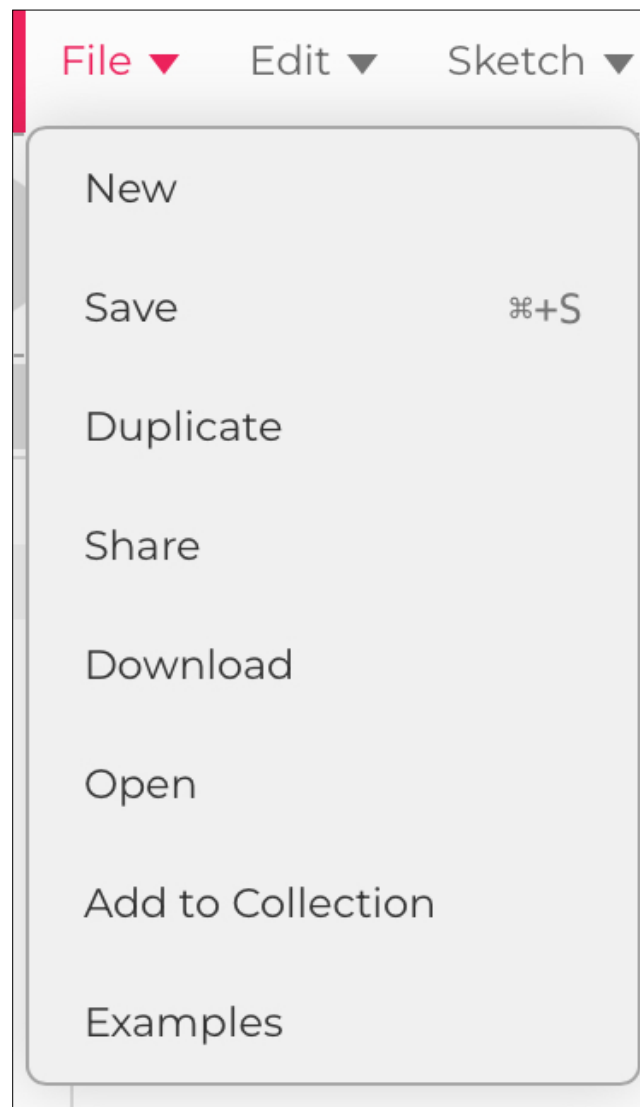
Add to Collection:

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 13: file





The Edit menu

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.

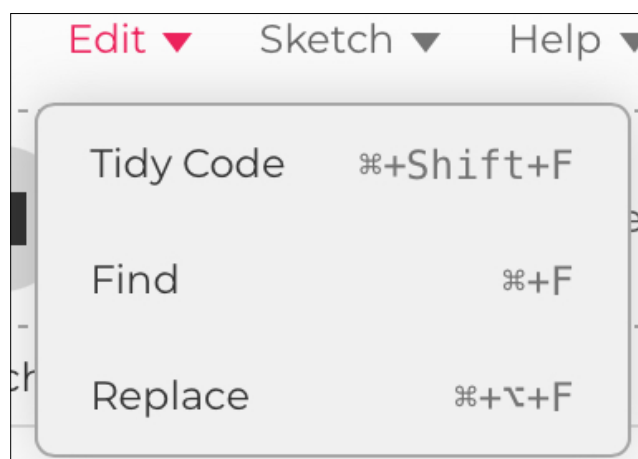
Find:

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 14: edit





The Sketch menu

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.

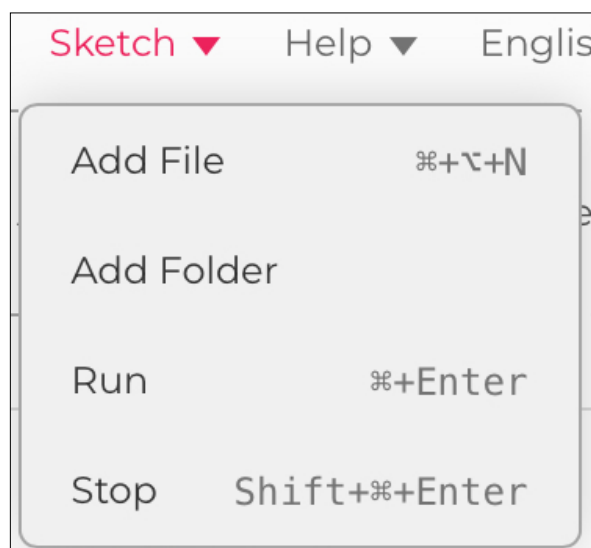
Run:

This does the same as the button.

Stop:

This also does the same as the button.

Figure 15: sketch





The Help menu

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.

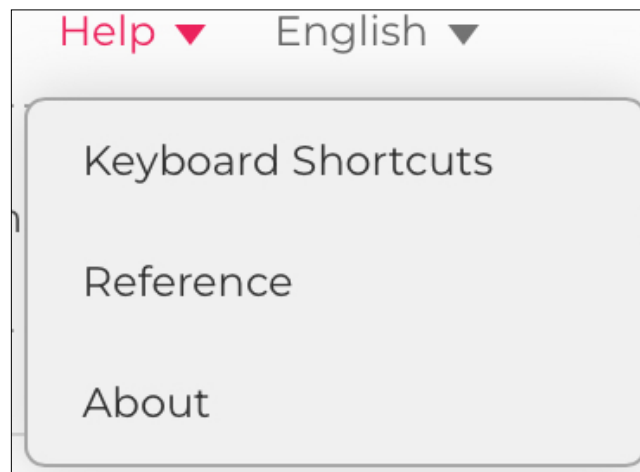
About:

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 16: help





The purpose of the console

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.

Figure 17: the console



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:

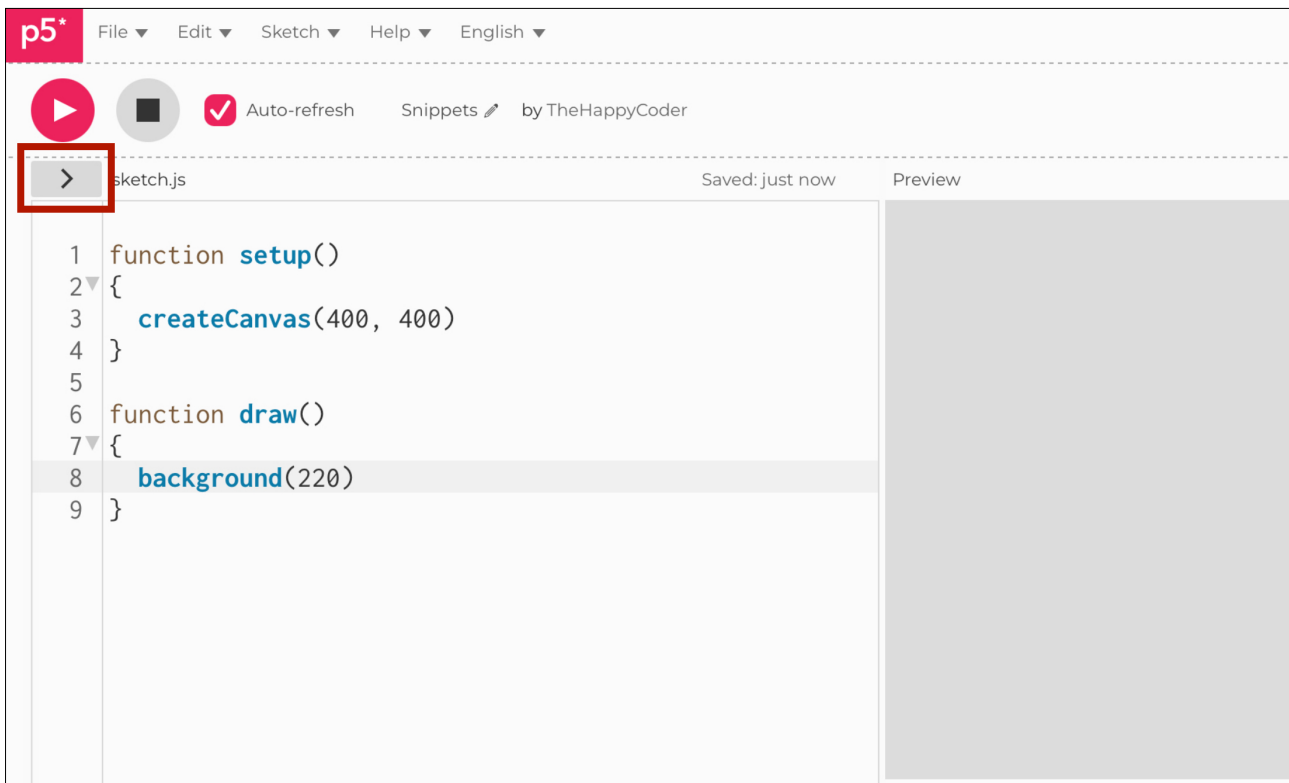
We can put a line of code in called **console.log()**. This means you are going to log something in the console. Sometimes it can be a piece of text such as **console.log('finished training')**, or it can be the value of a variable such as **console.log(counter)**, which will give you the value of the 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.



Accessing the sketch files

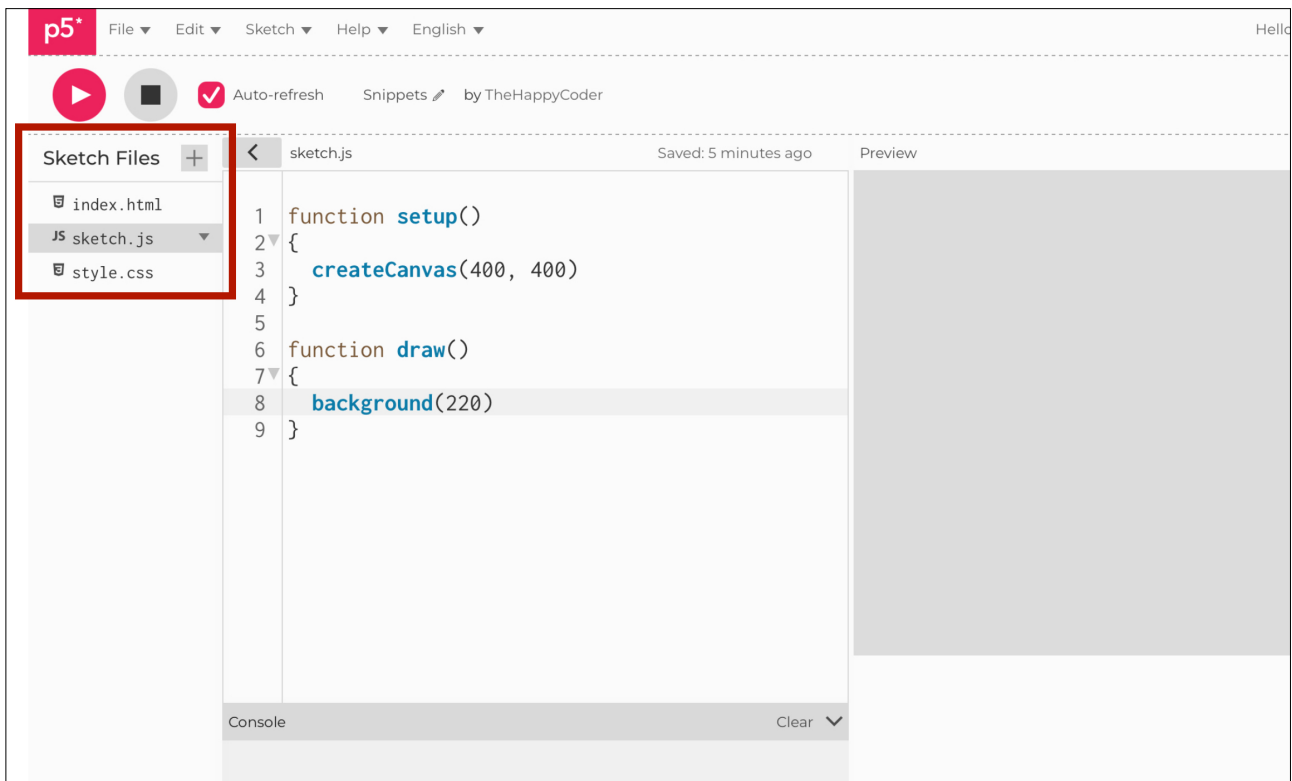
It is going to be critical to understand where the sketch files are and what files are used in creating a sketch. To the left of where you type your code, you will find a grey box with a grey arrow. I have highlighted it in red; click on it.

Figure 18: sketch files



When you click on it, you will get a menu of **Sketch Files** associated with the editor. Those listed here are **index.html**, **sketch.js**, and **style.css**. The main ones you will be interested in are **sketch.js** and **index.html**. The **style.css** can be ignored (for now).

Figure 19: files menu



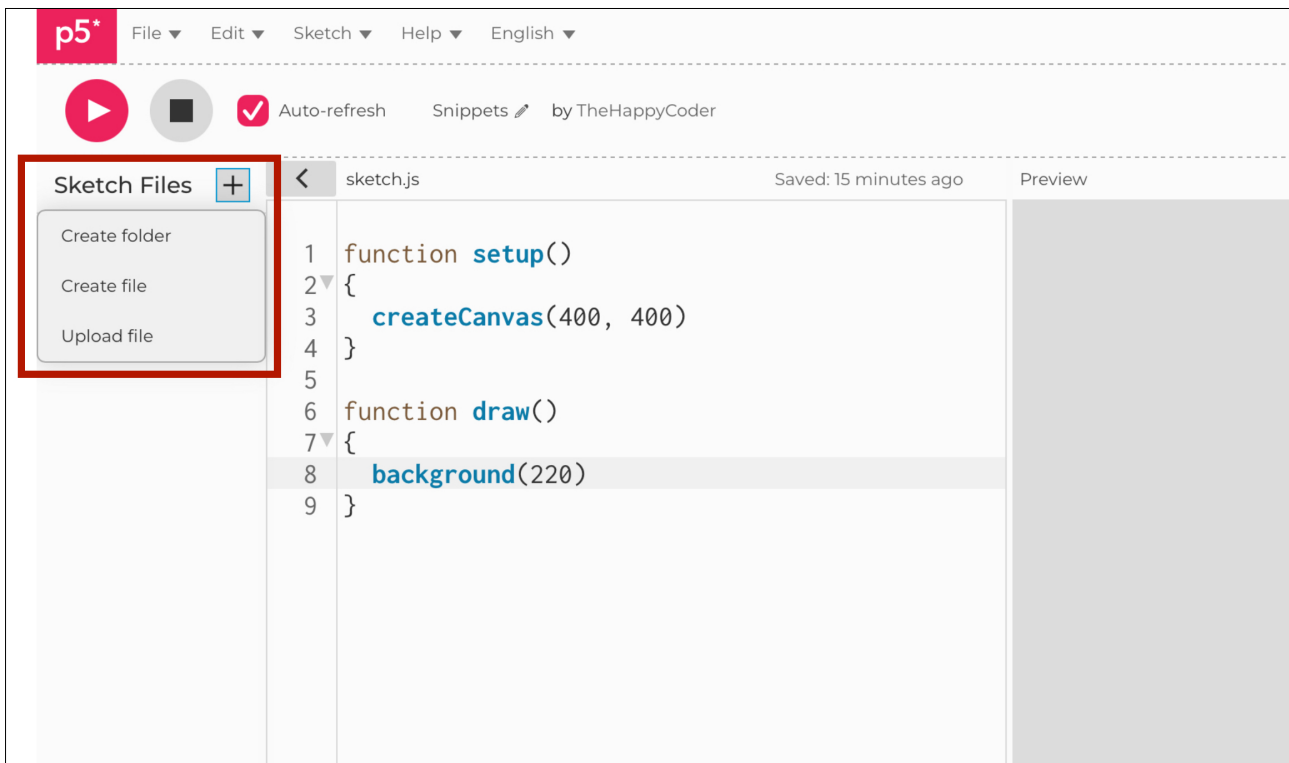


Adding files

Although I have avoided adding extra files in the majority of the coding, there are times when adding them makes things neater and, on the whole, easier to manage. You have the `sketch.js` file by default as your main coding file, but you may want to add more. You will need to give it a name and file extension `.js`, for example, `particles.js`.

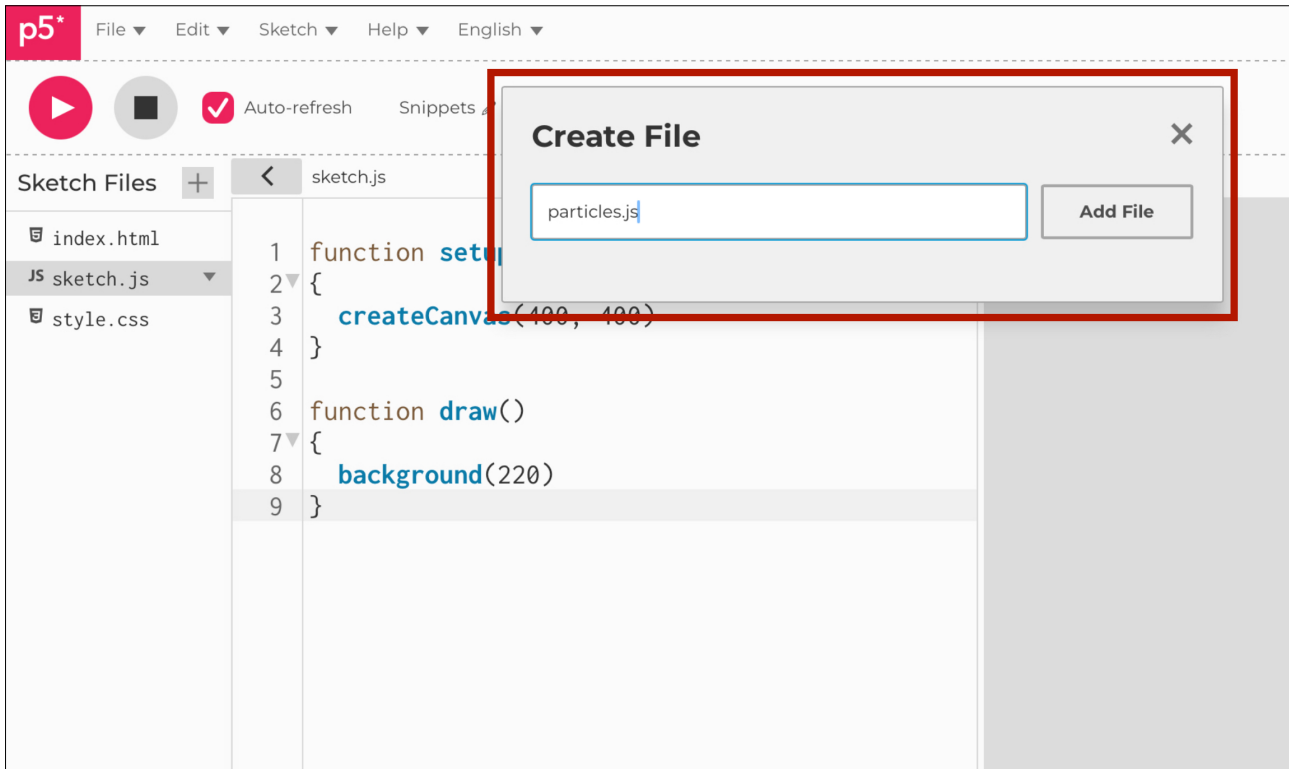
1) You add them by clicking on the grey + sign next to Sketch File.

Figure 20: adding files



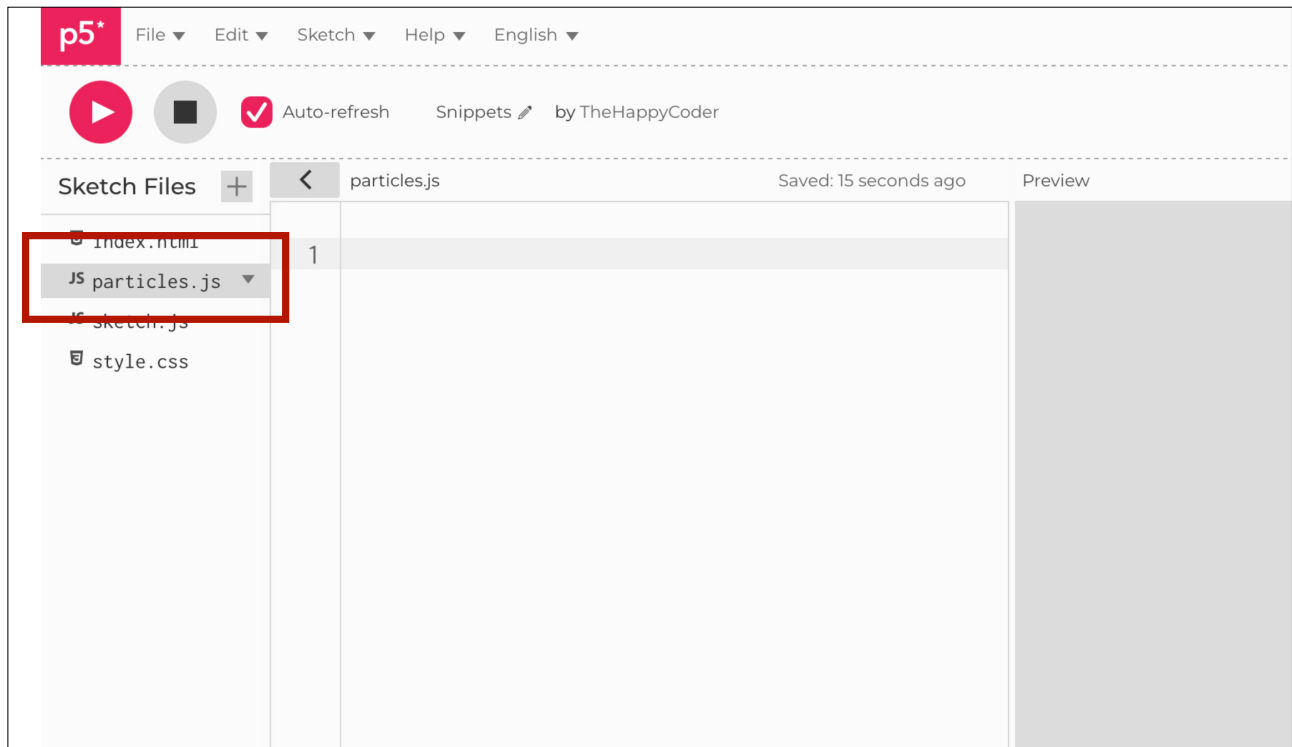
2) Now you click on Create File. You will get a box appearing waiting for you to put the name of the file in (it is important that it is case-sensitive and must have the .js file extension at the end).

Figure 21: creating and naming files



3) Click on Add File, and the file will appear in the Sketch files. Notice that the file is empty of any code. However, we aren't done yet; to use this new file, we have to make reference to it in the index.html file.

Figure 22: adding the file



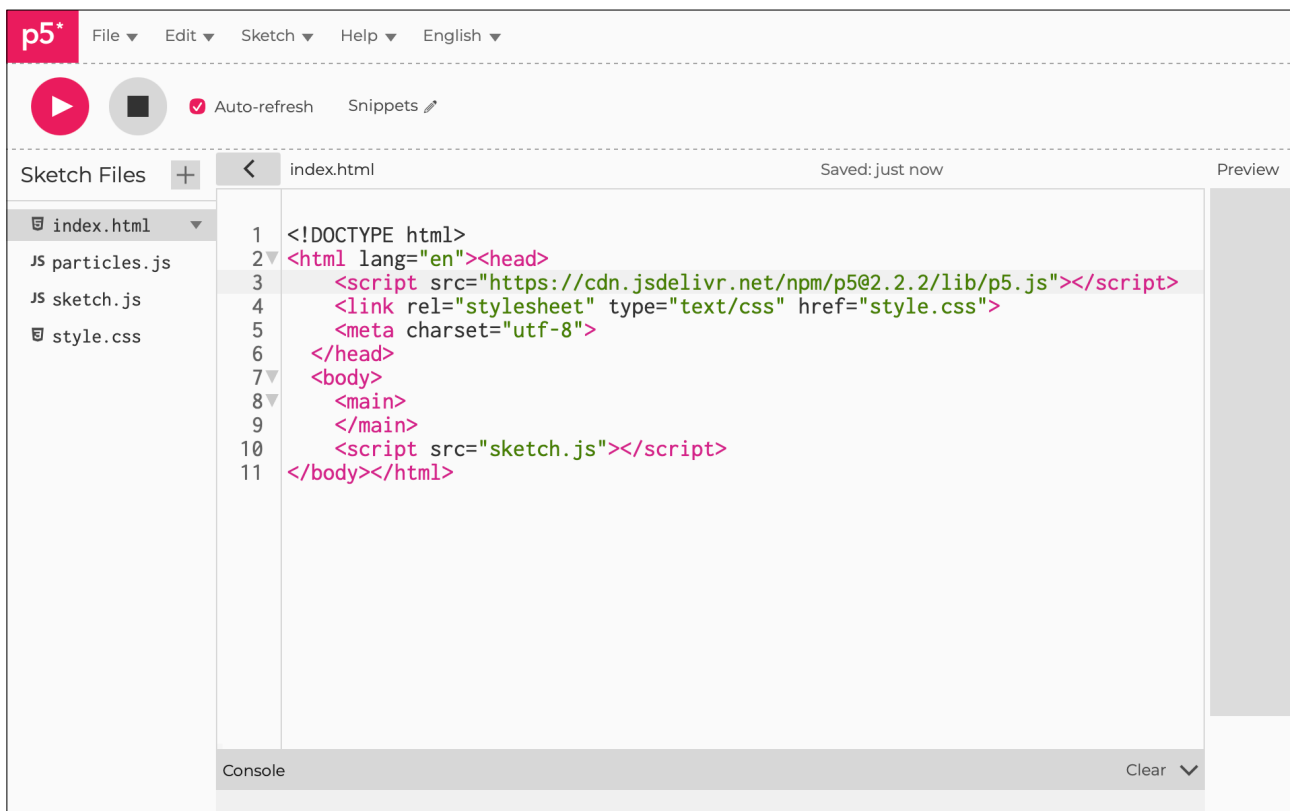


The index.html file

It is worth showing you now because I will make reference to it later when adding the `ml5.js` library (and other files), but for now I will show you how to add the file we have just created. This bit is often easy to forget to do and then wonder why the code is throwing a wobbler at you.

4) This time we click on the index.html file to open a bewildering amount of code. Don't be put off if you are not familiar with HTML tags.

Figure 23: the index.html



Notes

Yours may have a different version of p5.js, we will come to that in a moment.



Latest version of p5.js

When I originally created this workbook, I was using version 1.x of p5.js and an old version of ml5.js version 0.12.x or something like that. Much has been happening behind the scenes and p5.js and ml5.js have recently been upgraded as has the web editor (a little bit).

I will be using the latest version of p5.js and ml5.js (as of writing). The latest version of p5.js is version 2.x (2.2.2 as of writing), and the latest version of ml5.js is version 1.x. the newer version of p5.js is not currently the default (although it will be soon in 2026).

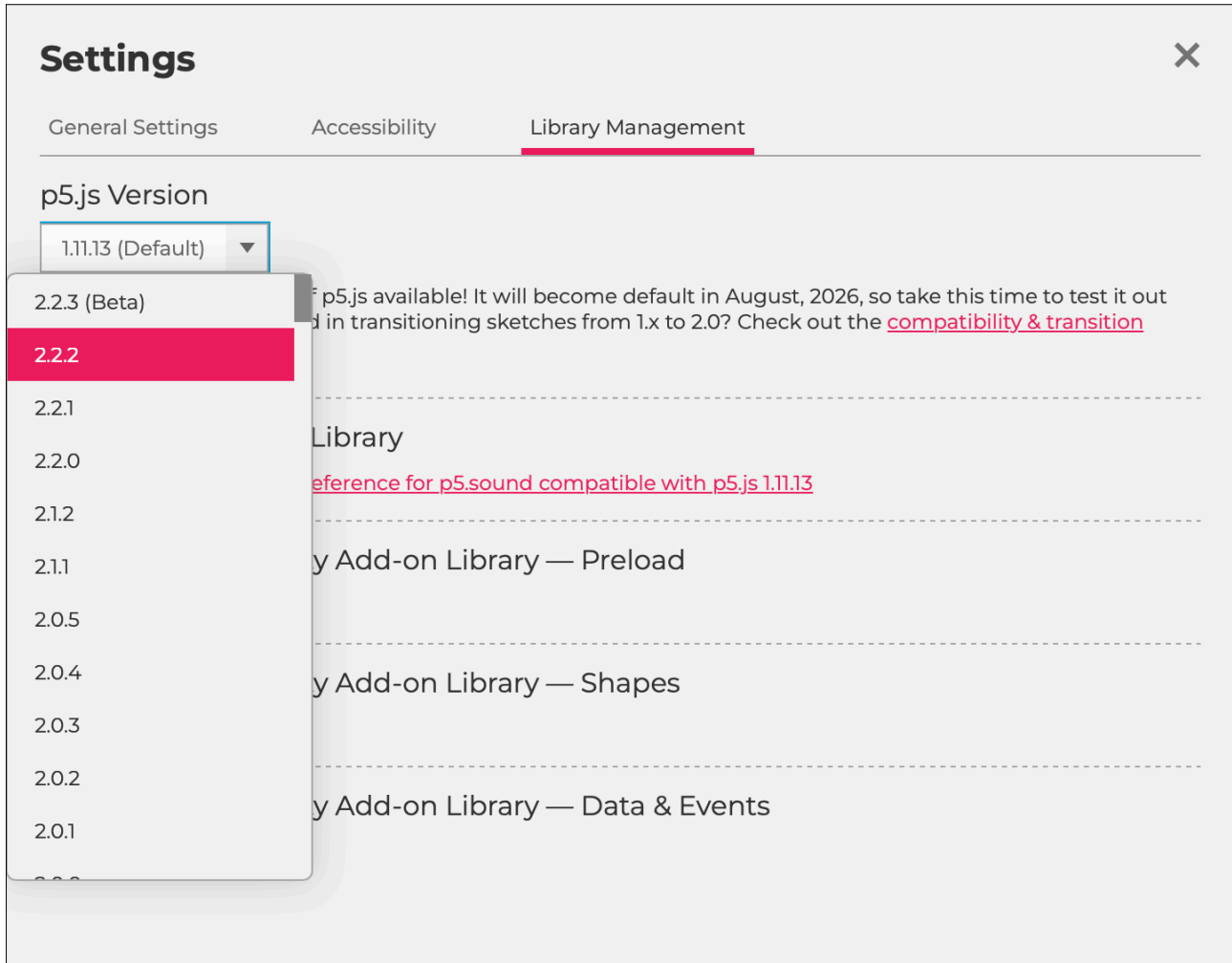
To change the version you click on the version tab at the top right hand corner.

Figure 24: default version (currently)



When you click on that you get a drop down menu of many version to select. Go to the top one and select it (not the beta version if there still is on). In this instance it is version 2.2.2.

Figure 25: select version 2.2.2 (or higher)



You should now have the latest version of p5.js.

Figure 26: latest version of p5.js

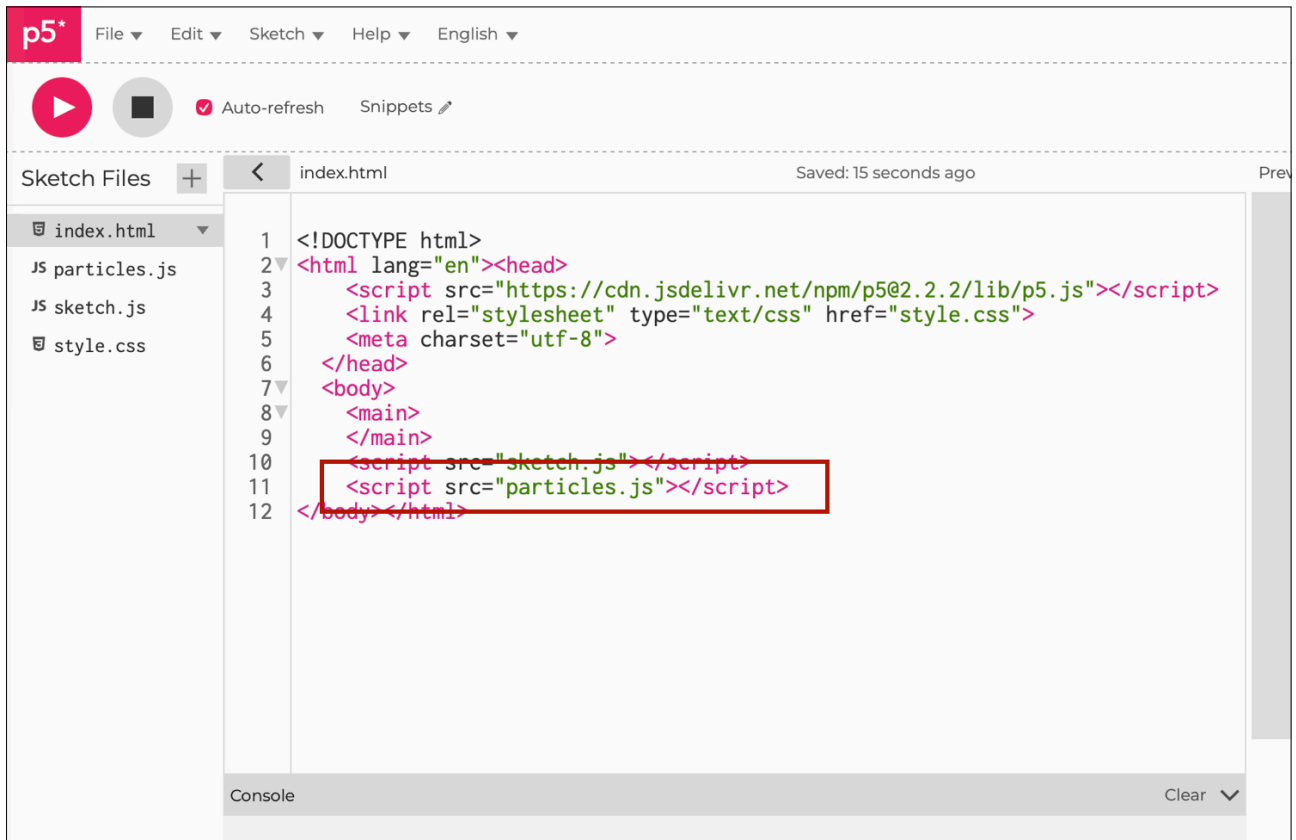




Adding a new file to index.html

5) The key point here is the line of code that shows the sketch.js tags. What I do is copy and paste it underneath and change the second sketch.js to the new name of the file, in this case particles.js. It has to match the name exactly.

Figure 24: index.html new file



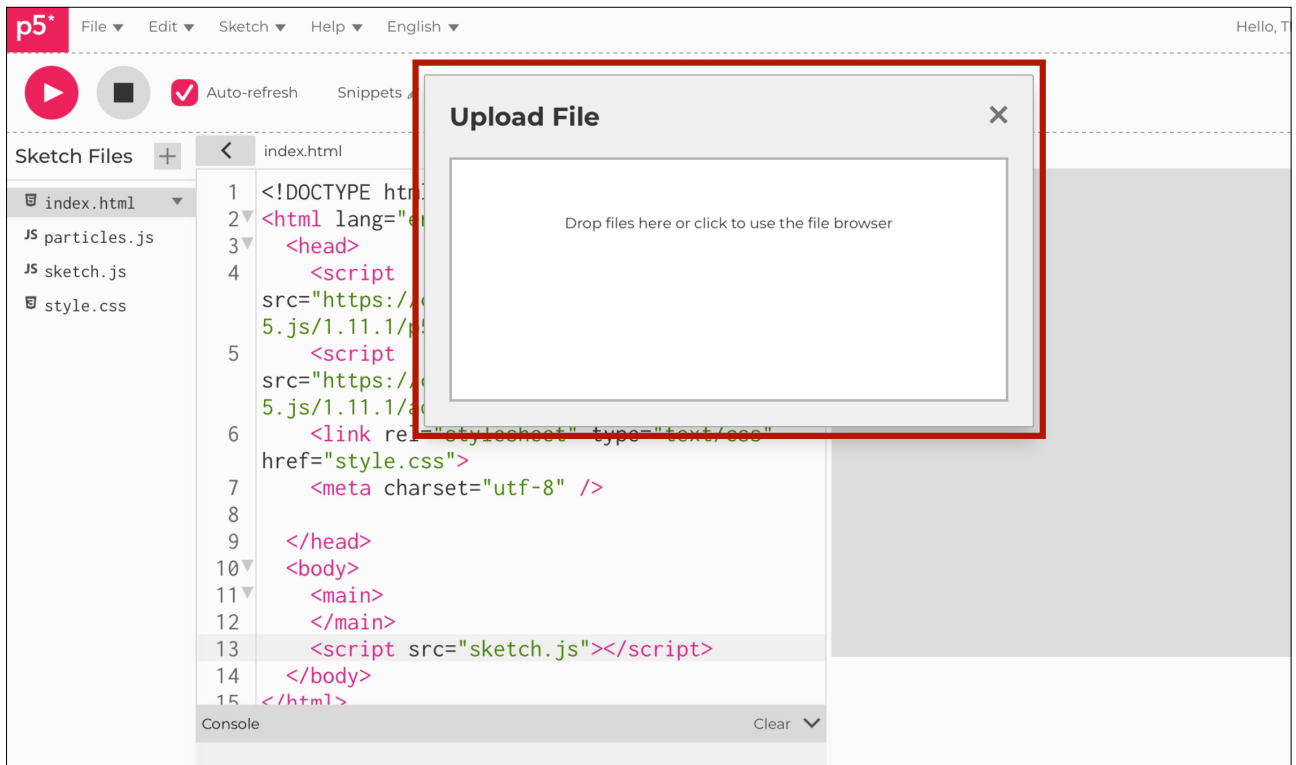
```
1 <!DOCTYPE html>
2 <html lang="en"><head>
3   <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
4   <link rel="stylesheet" type="text/css" href="style.css">
5   <meta charset="utf-8">
6 </head>
7 <body>
8   <main>
9   </main>
10  <script src="sketch.js"></script>
11  <script src="particles.js"></script>
12 </body></html>
```



Uploading files

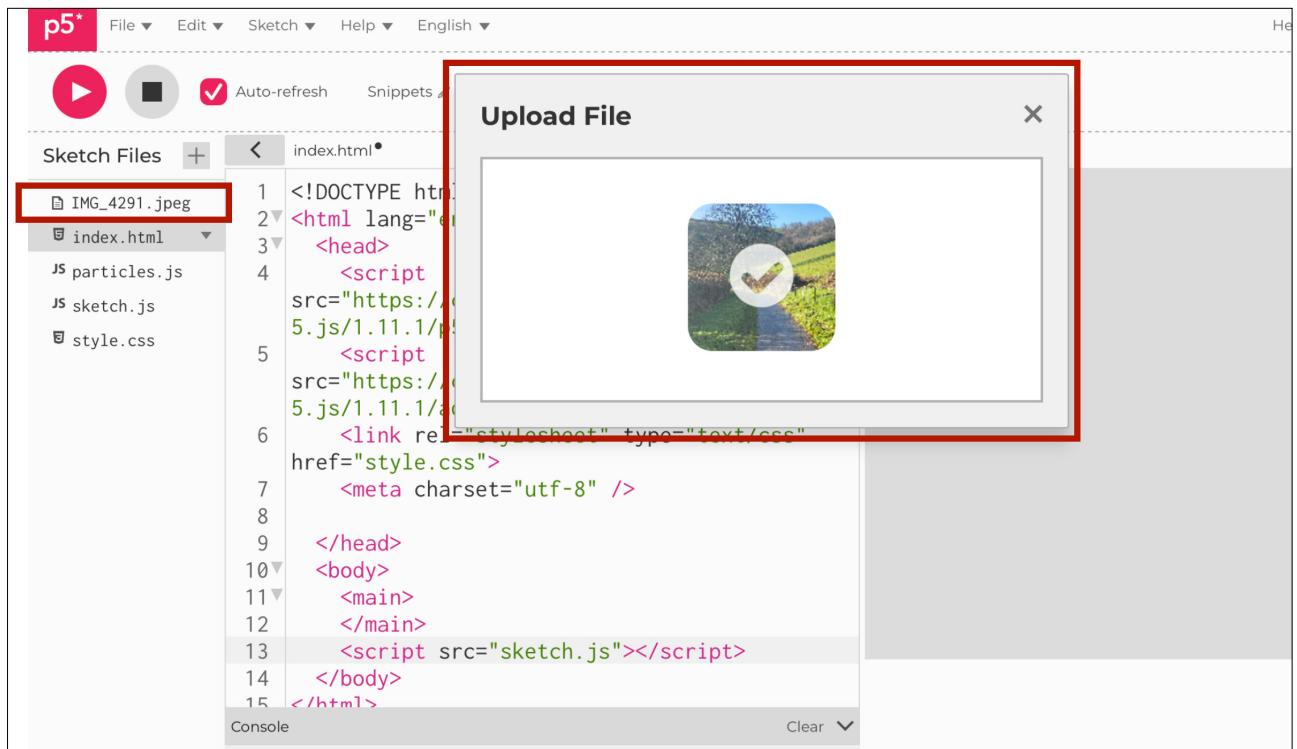
Going back to the **Add files** bit, we can also **Upload files**. These tend to be files that contain images, data, videos, models, fonts, etc. The process is quite simple, but you do need to have the files ready to either browse to or drag and drop from your desktop.

Figure 25: uploading files



1) In this instance, I uploaded an image of a photo I took while walking one day in Devon. You will notice that the image has a filename on the right in the Sketch files. We can change this to something more memorable. Just note that there is a limit to the size of the file, and the first one I chose was just too big; the limit is 5MB, so make sure you choose a file that will meet those limitations.

Figure 26: uploading image



2) Let's change the name to path.jpeg. It is important to keep the same file extension, but we can change the name.

Figure 27: renaming file

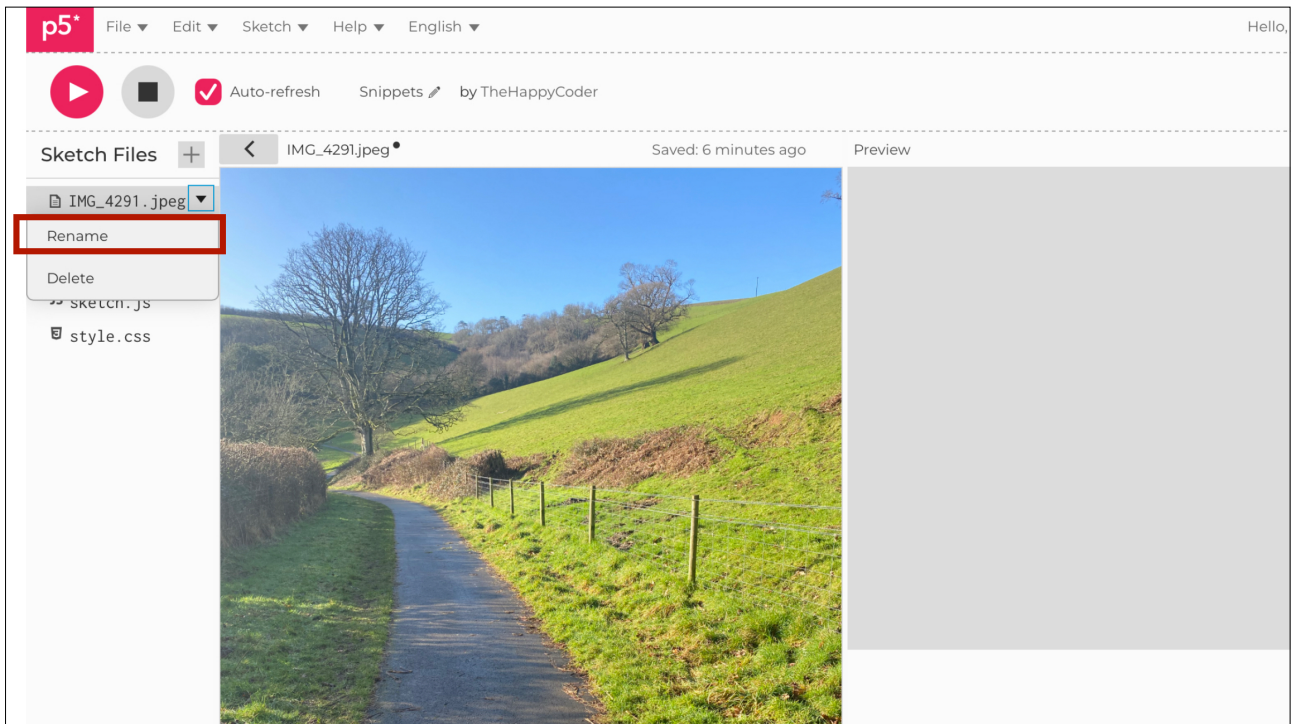
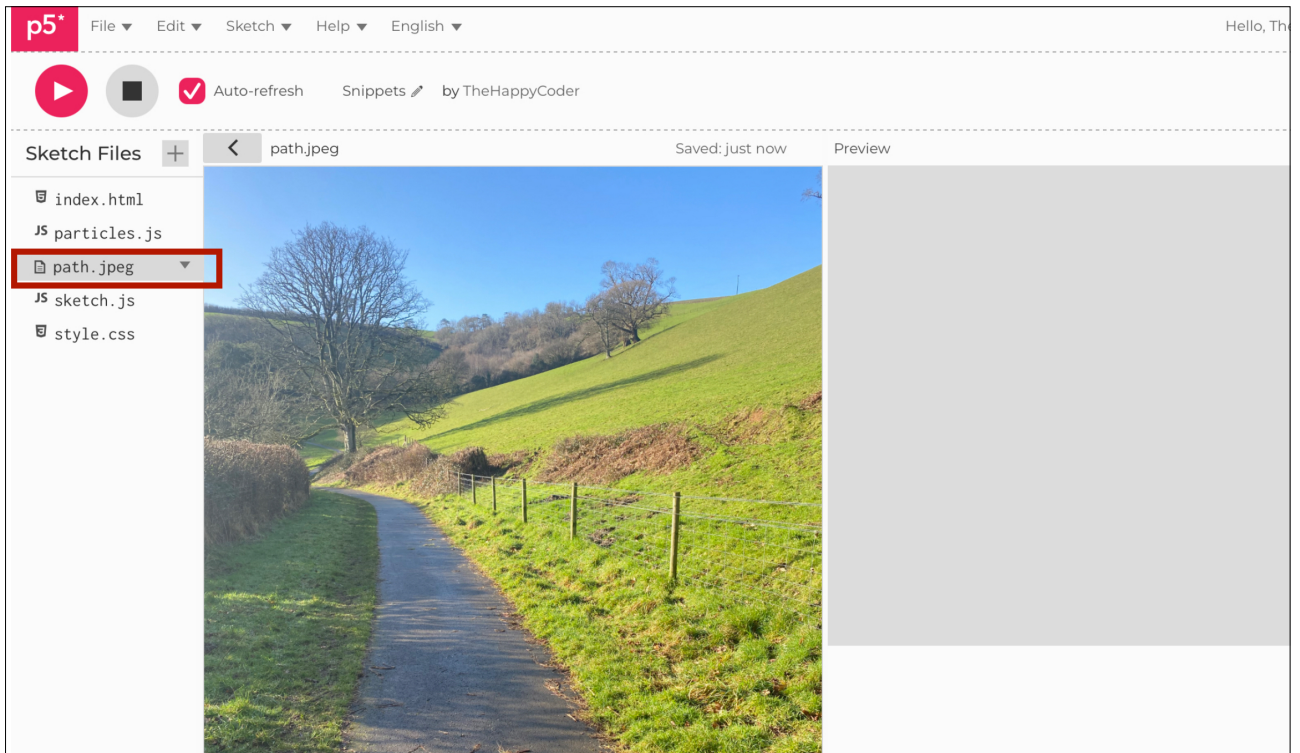


Figure 28: renamed file

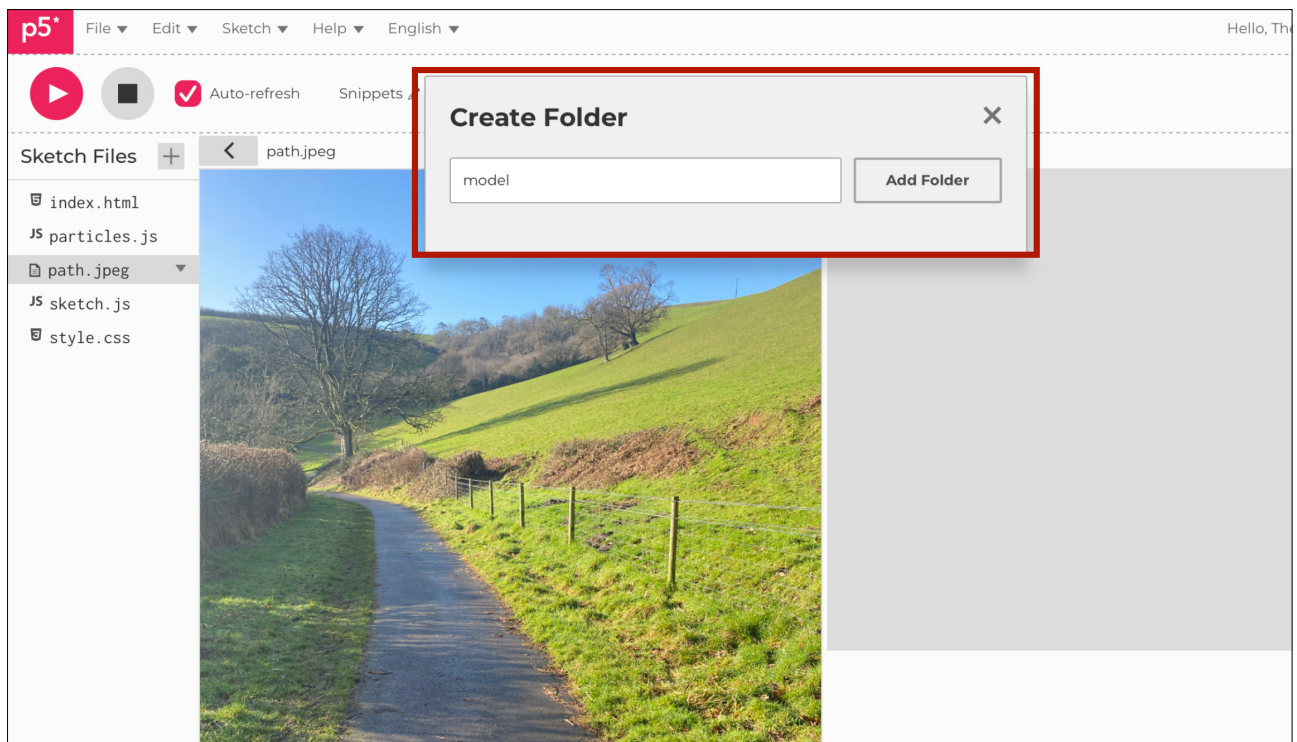




Creating folders

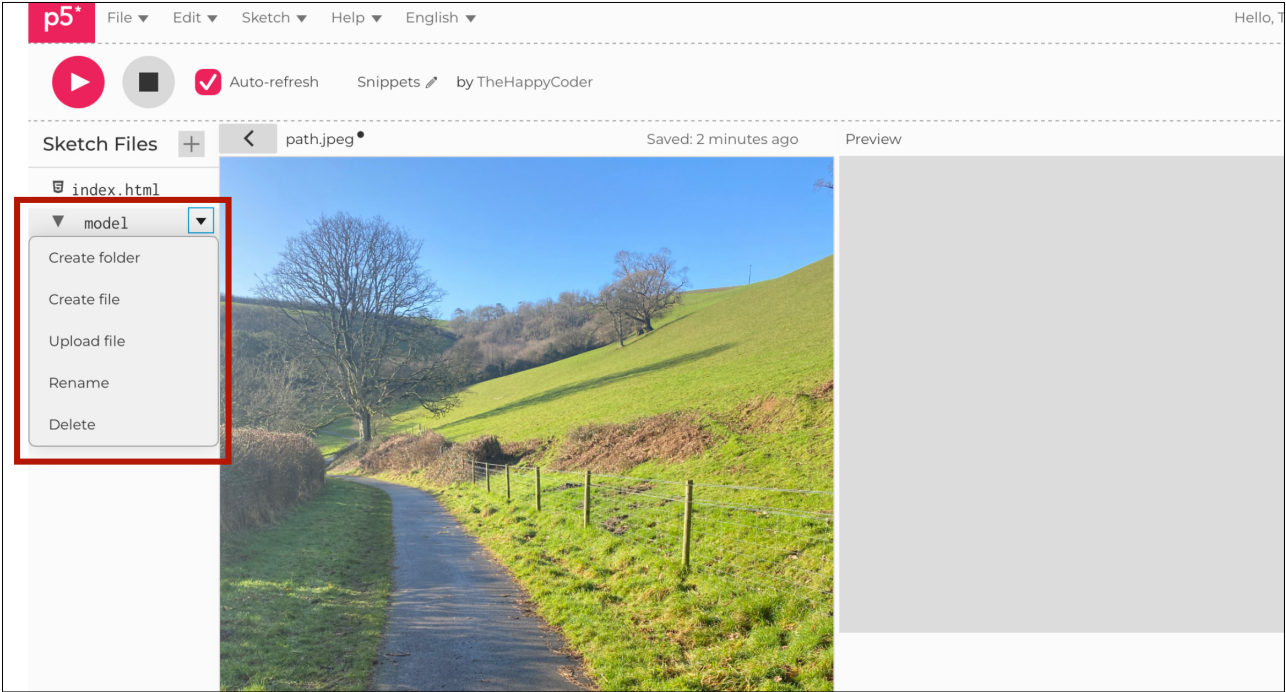
There are instances where you might want to group some images together, or as in the case of saving an AI model, you need to put all the files together in one folder. To do that, we can create a folder and give it a name. Once it is created, we can create files or upload files into that folder.

Figure 29: folders



As you can see, you can even create a folder within a folder, etc.

Figure 30: folder menu





Final words on this section

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 and the use they may or may not have.

In the next section, we will have a look at ml5.js, our route to machine learning and AI.

Algorithmic Intelligence Module A Unit #3 Overview of ml5.js



Module A Unit #3: overview of ml5.js

There is a lot to cover almost straight away, mostly it will be new jargon, phrase and concepts. They can sound intimidating but with repetition it will become familiar and the concepts are not too difficult to comprehend. This unit will introduce the following concepts, and functions that are common to nearly all AI models.

Introducing:

- the ml5.js neural network tasks
- the Debug tool
- the datasets
- the callback() function

Hyperparameters:

- Epochs
- Batch Size
- Hidden Layers
- Number of Nodes (Neurons)
- Activation functions

Key functions within ml5.js:

- __.addData()
- __.normalizeData()
- __.train()
- __.predict()
- __.classify()
- __.save()
- __.load()



Guide to adding ml5.js

We will be building some neural networks. The beauty of modern machine learning is that you don't have to build the neural network from scratch. We will be using a library called ml5.js, which is based on TensorFlow. This means that much of the heavy lifting is already done for you. Even so, there is so much to learn, experiment with, and develop, so don't think it is a complete doddle; you still need to know what you are doing.

The units you will work through will provide opportunities to build your neural network for a variety of tasks. They are all fun activities, but they are good learning opportunities as well. They will introduce you to some key concepts and deepen your understanding of the possibilities and limitations of machine learning.

To get started, we need to import the ml5.js library, and to do so, you need to follow the next steps carefully to include the line of code needed in the index.html. If you want to, I will give you the starting sketch for you to duplicate and save (see the button on my website).



Latest version of ml5.js

It is equally important to have the latest version of ml5.js. In the index.html file you should see an updated version of p5.js version 2.2.2 or higher. Here we can also add our ml5.js library. We are using the latest version which is:

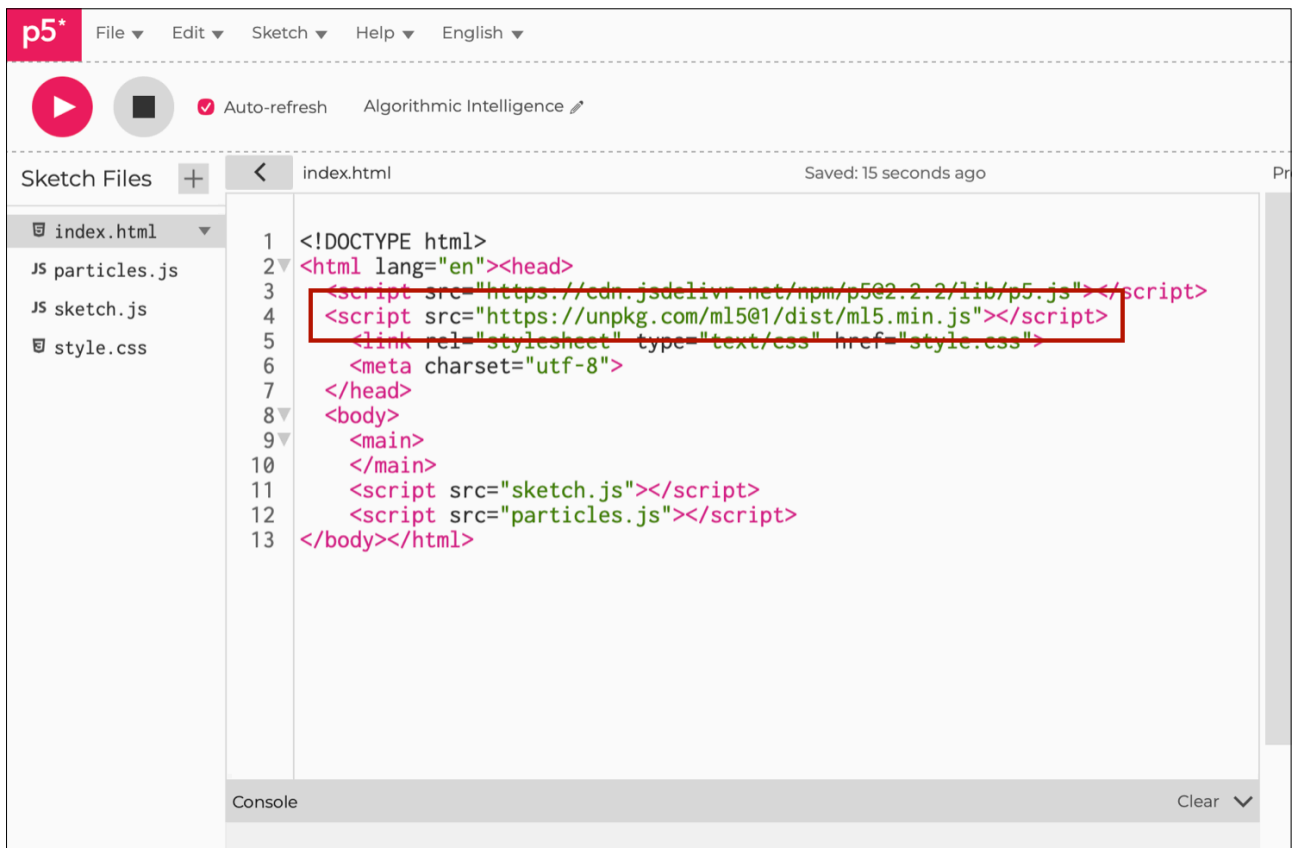
```
<script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
```

This line of code needs to be added when you want to use ml5.js, but I will take you through that later and even offer you a link to a blank sketch rather than typing it all out. The current version is shown above; there is an older version which includes some examples, but be aware that the older sketches don't work with the newer version without a little bit of tweaking. We will be using the latest versions of p5.js and ml5.js.

Notice that these lines have `<.../>` symbols; these are called tags. There is a lot I could share about **JavaScript**, **CSS**, and **HTML**, how they work together to create websites, how they give text style and functionality, but that is a topic all on its own and not essential for this tutorial, although it might include one at some time in the future.

The current default is version 1.11.13 (at time of writing) but there is a switch over to version 2.x (currently 2.2.2). You can change over (if not default) by clicking on the version number button top right corner. I recommend you do that unless I suggest otherwise. Type the code below:

Figure 3: adding ml5.js





The index.html file

The new line of code can slot in anywhere inside the `<head>` tags, along with the other similar lines of code.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



The ml5.js neural network

We are going to be working with the default settings in ml5.js. I have provided the default settings for information; don't worry if you don't know what they mean, all will become clear as we go along. There are two main types of task we will encounter at this stage. They are classification and regression.

More will be revealed in due course. The beauty of ml5.js is that there are built-in default settings which you can specify and change. You don't need to learn them here, just yet, but be aware that they are already incorporated if you don't specify an alternative value. For now just make a mental note. Everything will be explained in due course.

For regression the defaults are:

Hidden layers: 1
Hidden layer activation function: `relu`
Number of nodes: 16
Output layer: `sigmoid`

For classification the defaults are:

Hidden layers: 1
Hidden layer activation function: `relu`
Number of nodes: 16
Output layer: `softmax`

Other default settings:

Learning rate: 0.2
Batch size: 32
Epochs: 10



The task in hand

The type of task has to be identified first. This is so that you know what type of model you are going to use. There are primarily two main types: **classification** and **regression**. You need to decide which one it is going to be; this is your first job and tell that to ml5.js.

Although the neural network is almost identical for both tasks, the output gives the game away. The neural network will be predicting an output; if that is to classify something, for instance, an image of a dog, then it is a **classification** task. If the prediction is a value which can change, for instance, the price of a house, then it is a **regression** task.

Classification task:

An example could be sorting images of different animals, say, cats and dogs. The model will be trained on thousands of images of cats and dogs, each image labelled accordingly. Once trained, the model will then be shown an image of a cat or dog and will **predict** the likelihood (probability) that it is a cat or a dog. There will be two outputs: one, the probability of it being a cat, and two, the probability of it being a dog.

Regression task:

This usually only has one output. It will be a value, say, the price of a house, or predicting the temperature, or the price of a cryptocurrency. This will be a value that can have a range of values and once trained it predicts what it thinks it will be; this is based on the inputs. In the case of a house price, the inputs may include the number of bedrooms, garage or not, size of garden, number of toilets, distance from a school or train station and so on based on historical data.

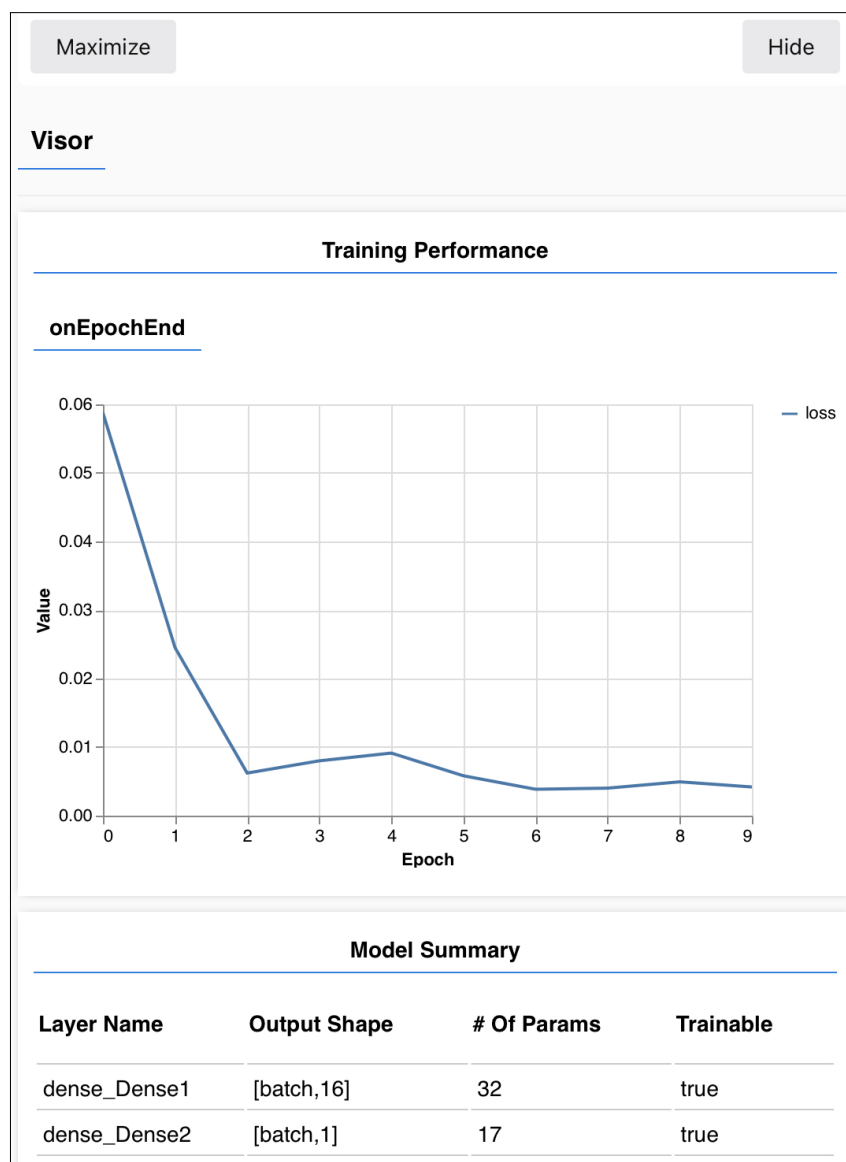


The debug tool

The debug tool is very useful but not essential. You activate it by calling it **true** and deactivate it by calling it **false** or not having it at all. It tells you how well the training is going on each epoch; it is a measure of the loss (error, cost, or difference between its predicted output and the real output). There is a hide button, which, if you click on it, will remove it when finished.

Also, if you move your mouse over the chart, you will get the actual value of the loss at any point on the chart.

Figure 4: debug and loss function





The Datasets

Mostly, we will be using what can be called synthetic data, which means it is not real data; it is artificially created. Synthetic data is also used in addition to real data when training models, especially in the case where there is too little data or it is too expensive to collect and label the data. It might sound like cheating, but done well, it is very effective and cost-efficient.

What you want is as much real data as possible. If it is images, then tens of thousands will be needed for the model to be able to train on them with some accuracy. This, as you can imagine, is very expensive as each image needs to be labelled, a very laborious task. People are employed to do just that and other similar data preparations.

It can be easy to just focus on the neural network architecture and the training phase, but the quality as well as the quantity of data is just as important. For instance, if you are doing facial recognition, but all your data is based on white, male, middle-aged faces, the model will have an inherent bias which may not be obvious until it is deployed at a later date when it struggles with non-white or female faces (this did happen). This was also true of accents in voice recognition software in the early days.

The trouble is identifying potential issues when there are tens of thousands of bits of data, so just be aware and don't take the data for granted. Check it out, test the model well before deploying it or using it for real.



The `callback()` function

Even though we haven't started coding yet, I want to mention the callback function. When we run a particular function in `ml5.js`, we often incorporate what is called a callback function. This can be simply to let us know when something has finished, or it could carry data such as the predicted results. Callbacks can also be used to check for errors. We will use them quite frequently, even though they are often optional. Their job is to call a function inside another function.



The Hyperparameters

These are the parameters that you can fiddle around with to get the best results. Although we can use the default settings and they will probably be OK for some situations, they are unlikely to offer the best solution. Knowing what each hyperparameter does is a skill you will need to develop. Experience will help you develop those skills, another reason for this tutorial, to get a feel for them. The main hyperparameters you will be using are listed below. If you were a data scientist, then you would have even more hyperparameters to play with, but you would also have huge, messy datasets, but thankfully, we will be using simple and relatively small datasets.

Training Hyperparameters

Epochs

Batch Size

Model Architecture Hyperparameters

Hidden Layers

Nodes (Neurons)

Activation functions

I will give you a brief description for each of these hyperparameters, but they will become more apparent when you actually start changing them. There is an element of trial and error, but after a while, you get a feel for what works well. Usually, the ml5.js default settings aren't far off the mark. All the above functions are options available during either the training phase or when initially designing the neural network.



Epochs

The epoch is usually the first **hyperparameter** you change, and it is the easiest to understand. In machine learning, an **epoch** refers to one complete pass through the entire training dataset. It's a fundamental concept in training neural networks. Here's a breakdown of what happens during an **epoch**:

- 1) The entire training dataset is fed into the model. This dataset contains the examples the model will learn from.
- 2) The model processes each data point and updates its internal parameters based on the errors it makes in its predictions. These parameters (called the weights) are like the knobs and dials of the model that determine its behaviour.
- 3) Once all data points have been processed, one epoch is complete.

It's important to note that multiple epochs are typically needed to train a model effectively. With each **epoch**, the model gets better at recognising patterns and making accurate predictions. However, there's a sweet spot: too few epochs and the model won't learn enough, while too many epochs can lead to overfitting, where the model memorises the training data too well and performs poorly on unseen data.

The number of epochs is called a **hyperparameter**, meaning it's a setting that needs to be tuned to find the best performance. The default is 10; we could increase it to, say, 250 or more, but that may be a waste of time as there is very little learning after a certain point. That is why the loss function chart is so helpful, as you can see where the training is giving diminishing returns.



The size of the batch

This is the second of the **hyperparameters** that we will tackle, and it is an important one even if it seems a bit nebulous.

Batch size is a term used in machine learning to describe the number of samples that are used to update the weights of a machine learning model in one iteration. For example, if you have a dataset of 100 samples and you set the **batch size** to **10**, the model will update the **weights** after processing **10** samples. The model will repeat this process until all the samples in the dataset are processed.

This is another **hyperparameter** that you can change. Through a bit of trial and error, you will get a feel for which **batch size** yields the best results for a specific timeframe. You could send it all through in one go (an epoch size) or send one at a time, but the ideal will be a compromise somewhere in between. Smaller batch sizes also require less memory as they don't have to process a large amount of data each time.

The default **batch size** for ml5.js (as far as I can gather) is **32**. Choosing the **batch size** is trial and error. Although you can specify any size, it is usual to try values of: 8, 16, 32, 64, 128, 256, etc.



The hidden layers

The **hidden layers** are sandwiched between the input layer and the output layer. It is another **hyperparameter** which you can command. The number of hidden layers has a profound impact; however, I would caution against going overboard with lots of layers to start with; it can slow the training because of the number of calculations.

This is where much of the magic happens, though actually no one is sure what is actually happening here, which may seem a bit strange. This has been one of the issues surrounding machine learning; it is the black box syndrome. A neural network has all those layers and nodes, with randomly generated weights, tweaking them as it learns from the data provided; this part is a bit of a mystery. The machine sets all these weights, and there can be thousands of them all working together.

Each **hidden layer** extracts features from the data passed through it. As it passes through all the layers, more and more complex features are identified. A bit like a series of filters, each one building on the previous, it filters out irrelevant information. This is what makes neural networks so powerful.

Having multiple **hidden layers** means it can tackle non-linear problems. These are the more complex patterns and problems that aren't simple linear (straight line) relationships. The world is rarely linear.

An example might be recognising cats or dogs. The first layer will look for corners, edges (e.g. tails, ears, whiskers) or colours. The next layer will identify patterns in the coats, and the next will find other subtle differences. Some of these features will not be obvious to us until it finally makes the prediction if it is a cat or a dog in the final output layer.

In ml5.js, the default is one hidden layer, but as you will see, we can (and will) add more hidden layers. But more, as you will find out, is not always better. Next, we can decide how many nodes we want for each hidden layer.



How many nodes (neurons)?

The input layer, to be precise, isn't usually called a layer. The number of **nodes** (**neurons**) is fixed by the input data, and this is also true for the output layer. The hidden layer(s) can have as many **nodes** as you want to give them. So, surely the more the merrier, maybe, we will see.

The default is one hidden layer and it is given **16 nodes** (**neurons** or **units** to use their jargon). This is a very good starting point. You can give it fewer or more. If you have more than one hidden layer, you don't even need to have the same number of **nodes** in each. Again, it is something to play around with to get a feel for what works, or doesn't work, or seems to make little or no difference.

Each **node** has an **activation function**, and an extra **node** is added called the **bias node**. The **bias node** just makes sure that there is always a value being introduced so that it never collapses to zero, which cannot be trained. This is created automatically, but I thought it was worth mentioning.



Activation functions

There are quite a few **activation functions**. But there are two common functions used, and we will explore them to start with. There are many more that can and are used. Those who work in research are exploring ways to make neural networks better and more efficient. They explore different architectures and functions to achieve better results, but we will keep it simple for this tutorial.

The default activation function in the hidden layer nodes is what is called the **ReLU** function. In the recent past, the **Sigmoid** function was the popular one, but that was costly in terms of calculation and was a poor performer compared to the **ReLU** function.

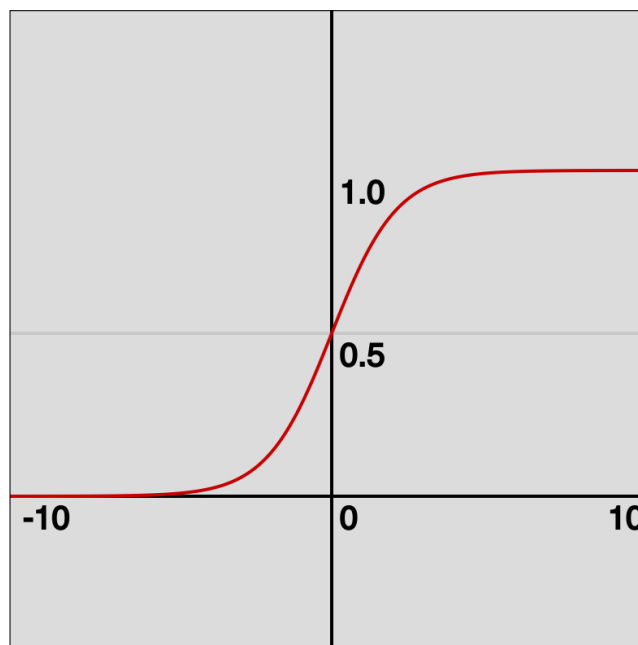


The Sigmoid activation function

To start with, we will look at the **Sigmoid** function. It inputs any value (x-axis) and the output will be between 0 and 1; however, you can see that any values close to 5 are effectively 1 and -5 effectively 0.

This is great and works well, but it is computationally heavy; it takes a lot of working out, and if you have thousands of nodes each with their own sigmoid activation functions, then this can take the training a long time. If you want outputs between -1 and 1, then you can use a variation of the **sigmoid** called **tanh**.

Figure 5: Sigmoid





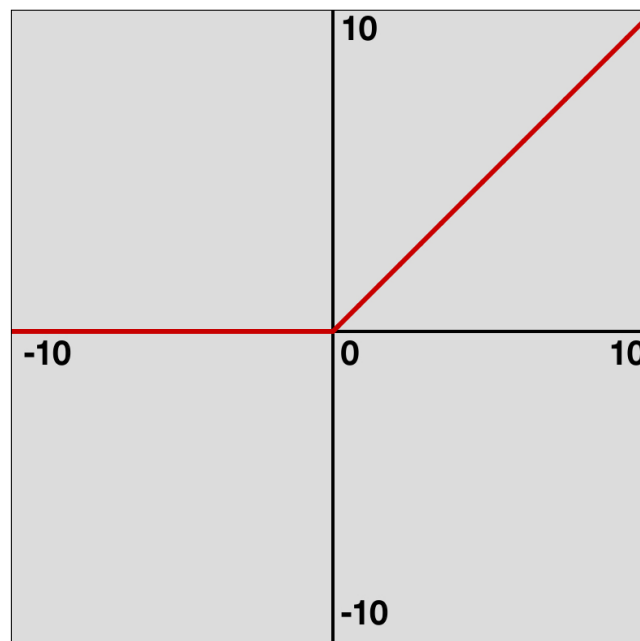
The ReLU activation function

To improve the efficiency of neural networks, they looked at alternatives to the **Sigmoid** function that worked better and more efficiently. They came up with the **ReLU** function, which looks almost too simple for it to work well, and yet it does. This is especially true in a deep neural network (DNN), which has many hidden layers and nodes.

It works very efficiently and yields excellent results. It works so simply: any x value less than zero, the output is zero; any value greater than zero, the output is the same as the input.

A problem can arise sometimes if the input is exactly zero. Getting a zero output is not ideal and can cause problems for the network; hence, there are variations on this, one of which is called **leakyReLU**, which is quite similar.

Figure 6: ReLU





The ml5.js functions

For this tutorial, we call our model `nn` for no other reason than for simplicity. We have functions we can call to perform a particular action. The format is the name of the model, in this case `nn`, followed by a dot (`.`) and the function we are calling. Inside the brackets is the information or data we are actioning and also where we might put a `callback()` function.

`nn.addData()`

We use this to add the data to the neural network.

`nn.normalizeData()`

This normalises the data, taking it from a wide range of values and squashing them, in proportion between 0 and 1.

`nn.train()`

We call this function when we are ready to train the model on the data added. Here, we can include options for the number of epochs and the batch size.

`nn.predict()`

Once it is trained, we can put in some data to predict the outcome; this is used with regression tasks. It uses sigmoid and then scales up the predicted value.

`nn.classify()`

This is used to make the classification prediction; it uses the softmax function (to pick the highest probability).

`nn.save()`

We can save the model once we have trained it and tested it. It is saved in three files. These files will contain the necessary information (hyperparameter settings and weights) to run it again.

`nn.load()`

This will load the three files that make up the model and make it available to predict or classify.



Overfitting and underfitting

Overfitting is where the model performs well in the training but fails to generalise the learning when you use new data. It has learned the correlation between the inputs and outputs too well. It has memorised the data, not recognised its patterns.

Underfitting, on the other hand, is the opposite. It becomes obvious when the model is too simple and cannot create a relationship between the input and the output, and so hasn't recognised any strong patterns.

This is why you usually have your data split three ways. Firstly, you use 80% for training, 10% for validation, where you fine-tune your hyperparameters. Then the remaining 10% is where you test the model to check if there is any overfitting or underfitting. For the tutorial, we will be using training data only and then predicting. This is for simplicity.

In the real world of machine learning models, you would have the three datasets. Also, it is important that for all three datasets, they are representative of the whole dataset. To do that, you take a random sample and not just creamed off the last 10% or 20% of the total dataset. It is important that the three datasets are equally representative of the total dataset.

Artificial Intelligence Module A Unit #4 p5.js code snippets #1



Module A Unit #4: p5.js code snippets #1

This is a gentle introduction to p5.js as it relates to the following modules on developing simple machine learning models with ml5.js on the p5.js platform. They are a brief introduction to the code you will be using. These snippets are also helpful for those who have never coded before or for those who are unfamiliar with p5.js.

My recommendation is that you work quickly through the coding snippets but at your own pace and play around with the code; it will help you to learn more about coding in p5.js. If there are challenges, then try them out and see what else you can [learn](#); the more practice, the better. Over time, it will become intuitive, like learning to ride a bike, drive a car, or learning the piano. At first, it may seem strange, but later it becomes second nature.

A brief explanation:

You will have the code in the yellow box to type into the editor. Any changes from the previous lines of code are highlighted in blue. A sketch is a complete code which you can run (usually). There will be a brief description of what you are doing and why. You will have some additional information in the form of notes. Sometimes there will be some challenges; they are optional. To help understand the code, there will be very brief descriptions about some lines of code which will appear in a box shown below.

Line of code	Explanation
--------------	-------------

The snippets will be kept as brief as possible. Delete the default code and start with an empty editor. Wherever relevant, I will include an image of what it should look like. As you code each step, it doesn't mean that the sketch is ready to run. Make sure you have it set up as you want with the font size, theme, etc. The curly brackets are important, as are any commas or semicolons; take care.



Sketch A4.1 function setup()

! This is our first sketch. Delete all of the default code and write the code below.

The `setup()` function is a built-in function. It simply runs through all the code between the two curly brackets `{...}` once. It is used to set up the code you are going to run.

```
function setup()
{

}
```

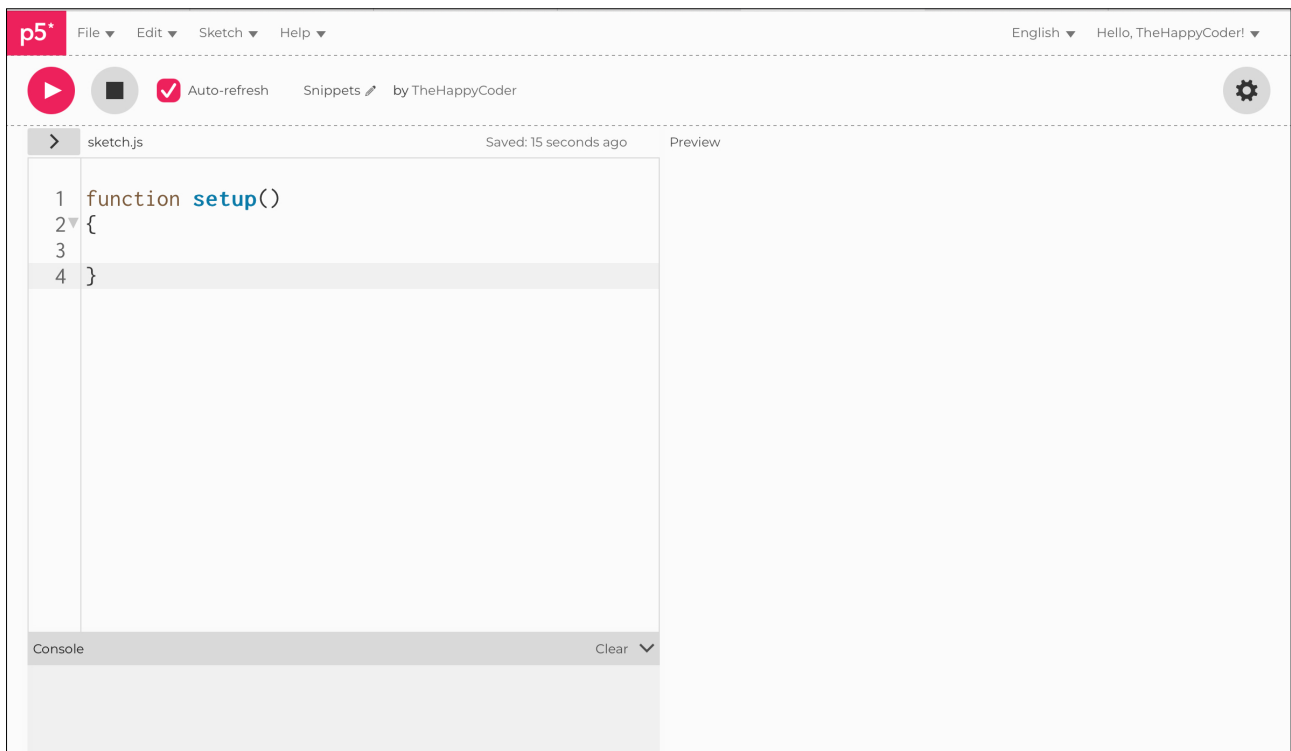

Notes

The `setup()` function is called once when the sketch begins running. Declaring the function `setup()` sets a code block to run once automatically when the sketch starts running. It's used to perform setup tasks such as creating the canvas and initialising variables.

Code Explanation

<code>function setup()</code>	Runs everything coded in this function once
-------------------------------	---

Figure A4.1





Sketch A4.2 function draw()

! Now add the code highlighted in blue.

The `draw()` function is the second built-in function. It is a loop function because it goes through all the code between the curly `{...}` brackets continuously in a loop.

```
function setup()
{
}

```

```
function draw()
{
}

```

Notes

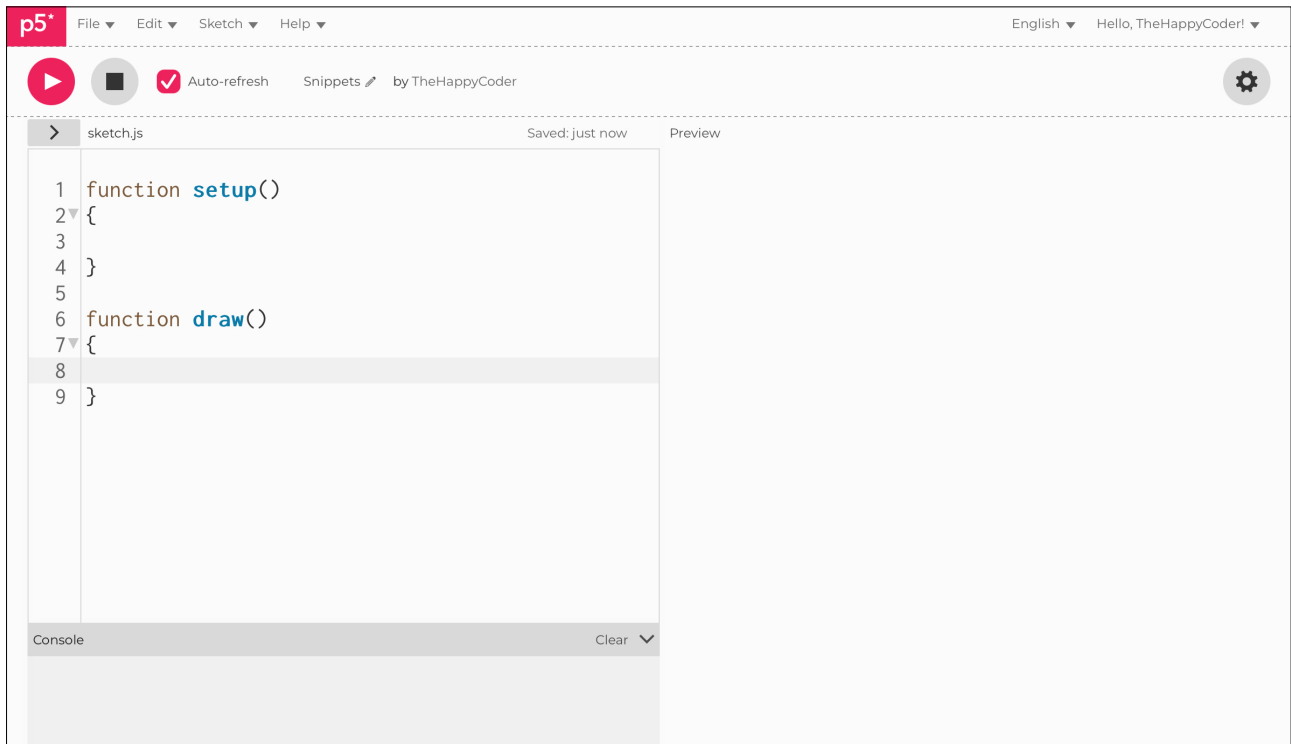
This is a pre-built function. The `draw()` function is one that is called repeatedly while the sketch runs. Declaring the function `draw()` sets a block of code to run repeatedly once the sketch starts. It's used to create animations and respond to user inputs.

Code Explanation

```
function draw()
```

Loops through the code inside the brackets continuously

Figure A4.2

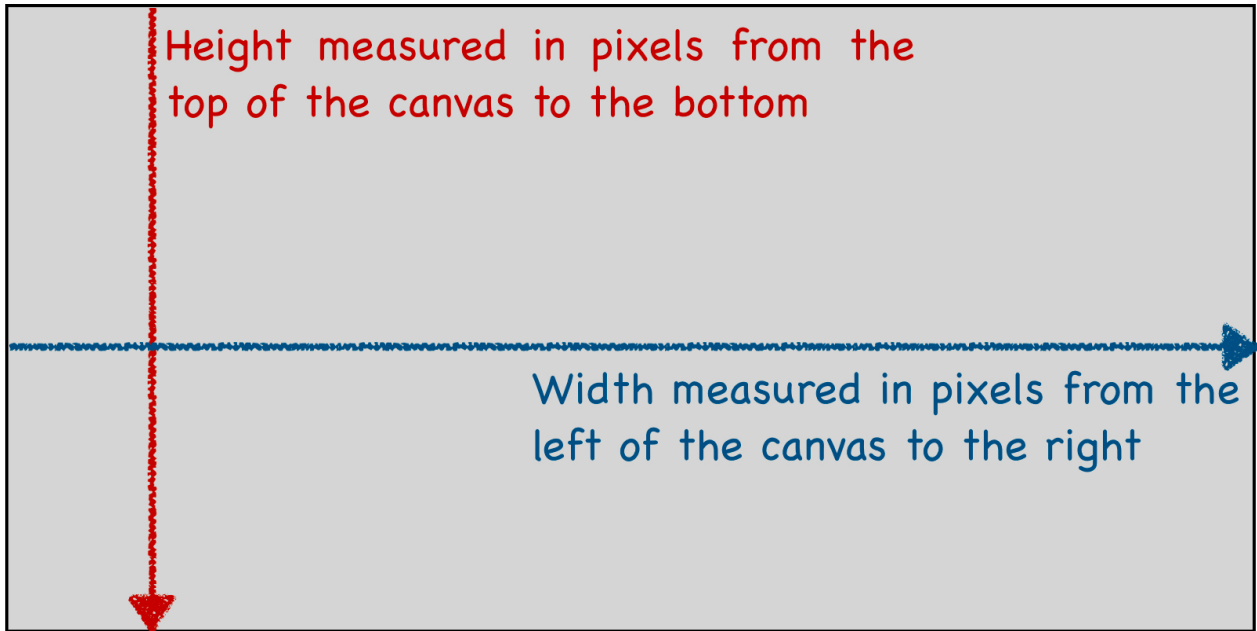




Canvas dimensions

The dimensions of the canvas need a little bit of explanation. Everything is measure in pixels from either the left or from the top. Mostly we will operate on the same dimensions of 400 x 400 pixels as this sits nicely on most computer screens.

Dimensions





Sketch A4.3 createCanvas()

This function creates the canvas on which you are going to draw. It requires at least two arguments. The first one is the **width**, and the second is the **height**. Both are measured in **pixels**.

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

function draw()
{
}
}
```

Notes

This creates a canvas element on the web page. The `createCanvas()` function creates the main drawing canvas for a sketch. It should only be called once at the beginning of `setup()`. Calling `createCanvas()` more than once causes unpredictable behaviour. The first two parameters set the dimensions of the canvas, and the values of the width and height system variables.

For example, calling `createCanvas(900, 500)` creates a canvas that's 900 pixels wide by 500 pixels high. By default, width and height are both 100. We can't see anything yet because we need to give it a background to code onto.

Challenge

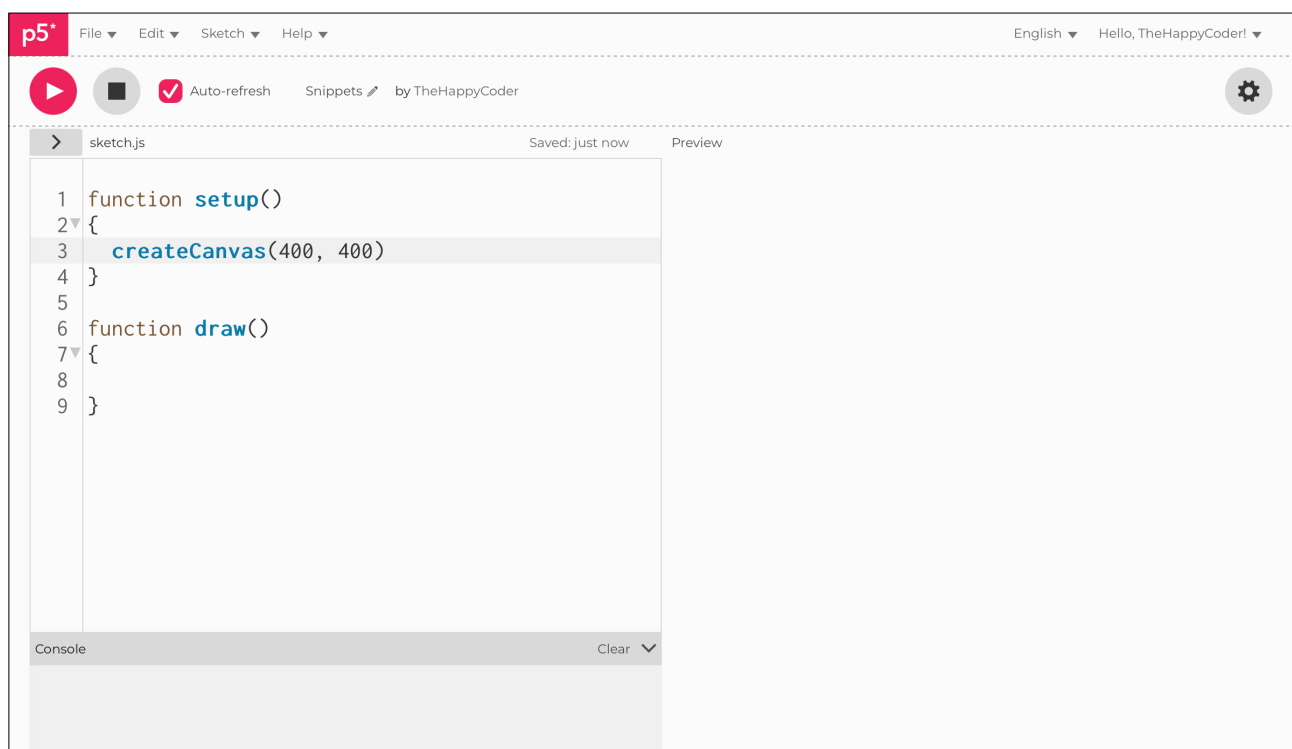
Play with the dimensions of the canvas.

Code Explanation

```
createCanvas(400, 400)
```

This creates a canvas to code onto. The two arguments are the width and the height measured in pixels

Figure A4.3





Sketch A4.4 background()

The **background()** function allows us to give the canvas a background colour. By default, it is white. Generally, we will give it a grey colour (**220**), where **0** is black and **255** is white; all the other numbers in between are shades of grey.

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

function draw()
{
}
```


Notes

Sets the colour used for the background of the canvas. By default, the background is transparent. The `background()` function is typically used within `draw()` to clear the display window at the beginning of each frame. It can also be used inside `setup()` to set the background on the first frame of animation. The value `220` gives you a very light grey, whereas a value of `50` gives you a darker grey.

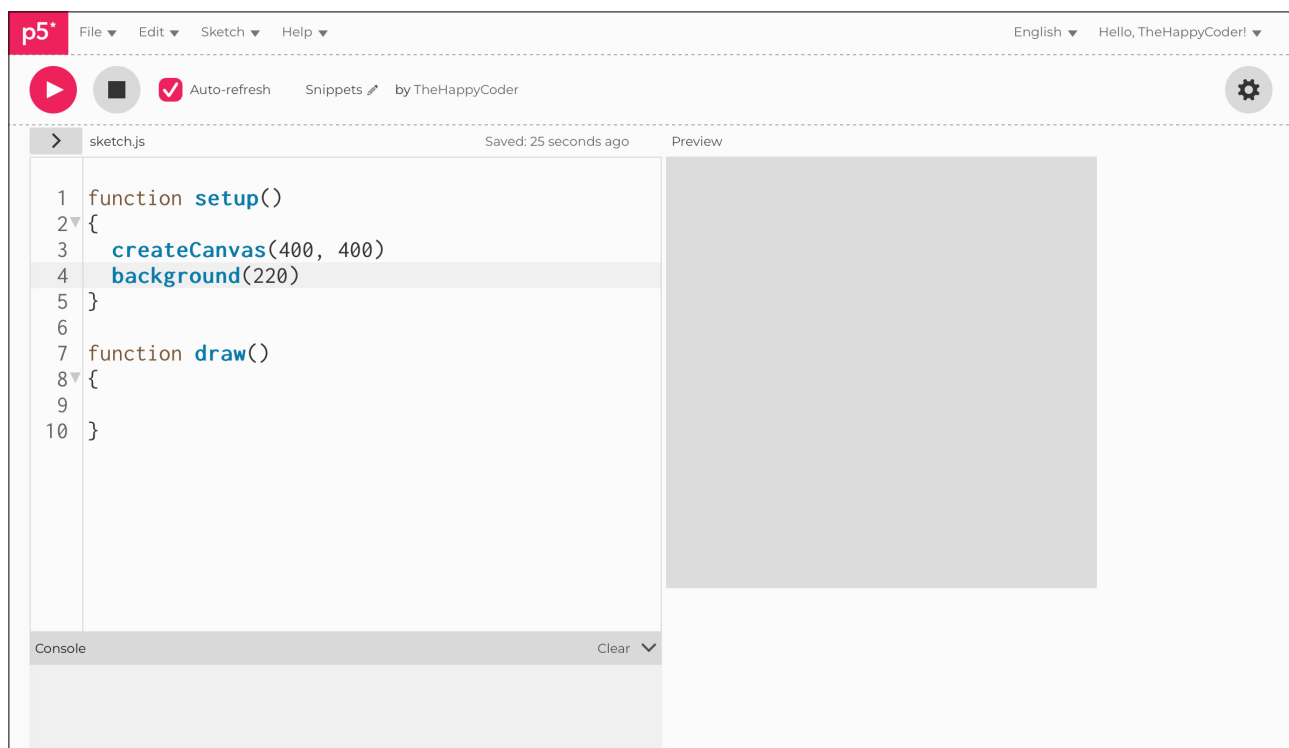
Challenges

1. Change the `220` to `0` or some other number.
2. Change the dimensions of the canvas from `(400, 400)` to `(100, 600)`.

Code Explanation

<code>background(220)</code>	Gives the background a colour
------------------------------	-------------------------------

Figure A4.4





Sketch A4.5 RGB colours

Although most of the time a grey background is all you need, you might want to have a brighter or different colour for your finished product. Instead of one argument for the background, we can have three arguments instead. They are still between **0** and **255**, but they represent the amount of **red**, **green**, and **blue**. This means you can mix them to make lots of different colours. The default mode is **RGB**.

```
function setup()
{
  createCanvas(400, 400)
  background(200, 100, 10)
}

function draw()
{
}
}
```

Notes

A `background()` with three arguments will interpret them as **RGB**, **HSB**, or **HSL** colours depending on the current `colorMode()`. By default, colours are specified in **RGB** values. The example above gives us a nice orange colour. I will include much more information about colours in a much later section.

Challenge

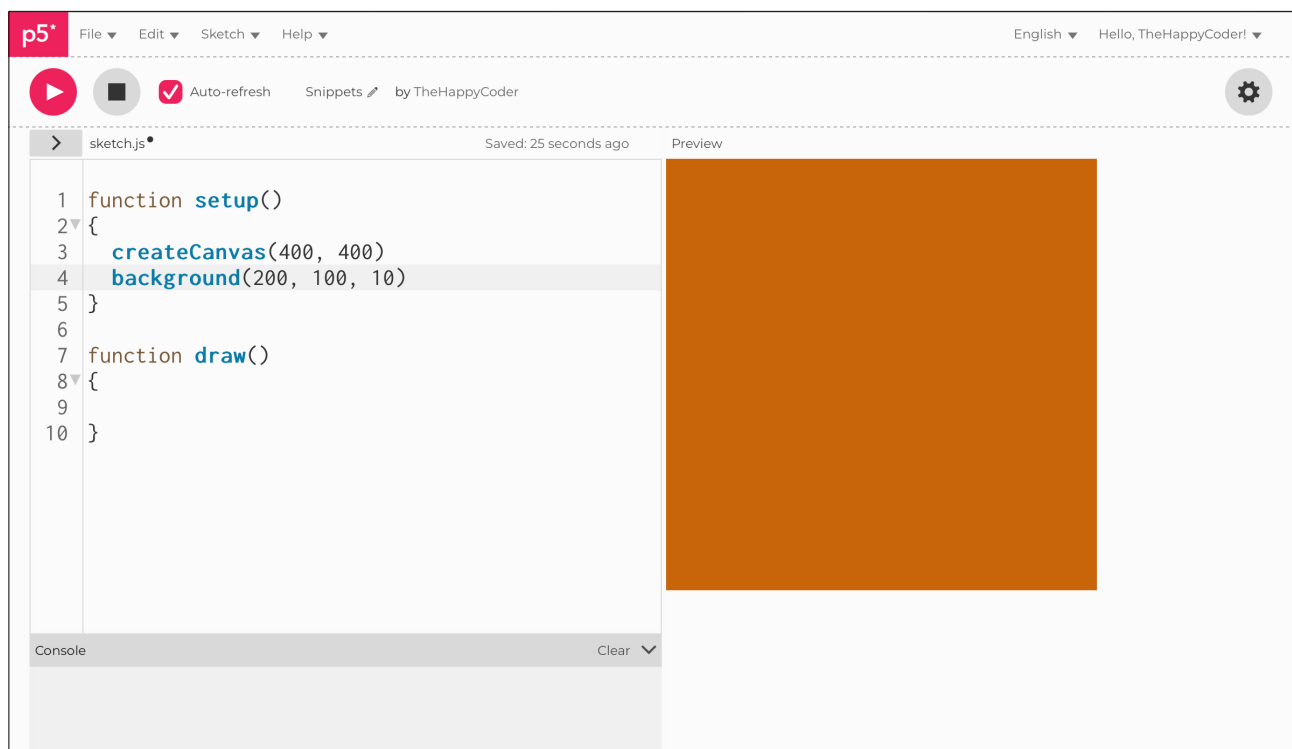
Play with the values to see what they do. If you just want red, then use `(255, 0, 0)`, if you want green, then use `(0, 255, 0)`, and so on.

Code Explanation

```
background(200, 100, 10)
```

This is how you can create colours with rgb values.

Figure A4.5





Sketch A4.6 a circle()

We can draw a circle using the `circle()` function. It needs three arguments. The first argument is the distance from the left-hand side. The second argument is the distance from the top. The third argument is the diameter of the circle.

```
function setup()
{
  createCanvas(400, 400)
  background(200, 100, 10)
}

function draw()
{
  circle(100, 300, 50)
}
```

Notes

This draws a circle of a diameter of 50 pixels. It is drawn 100 pixels from the left-hand side and 300 pixels from the top.

Challenge

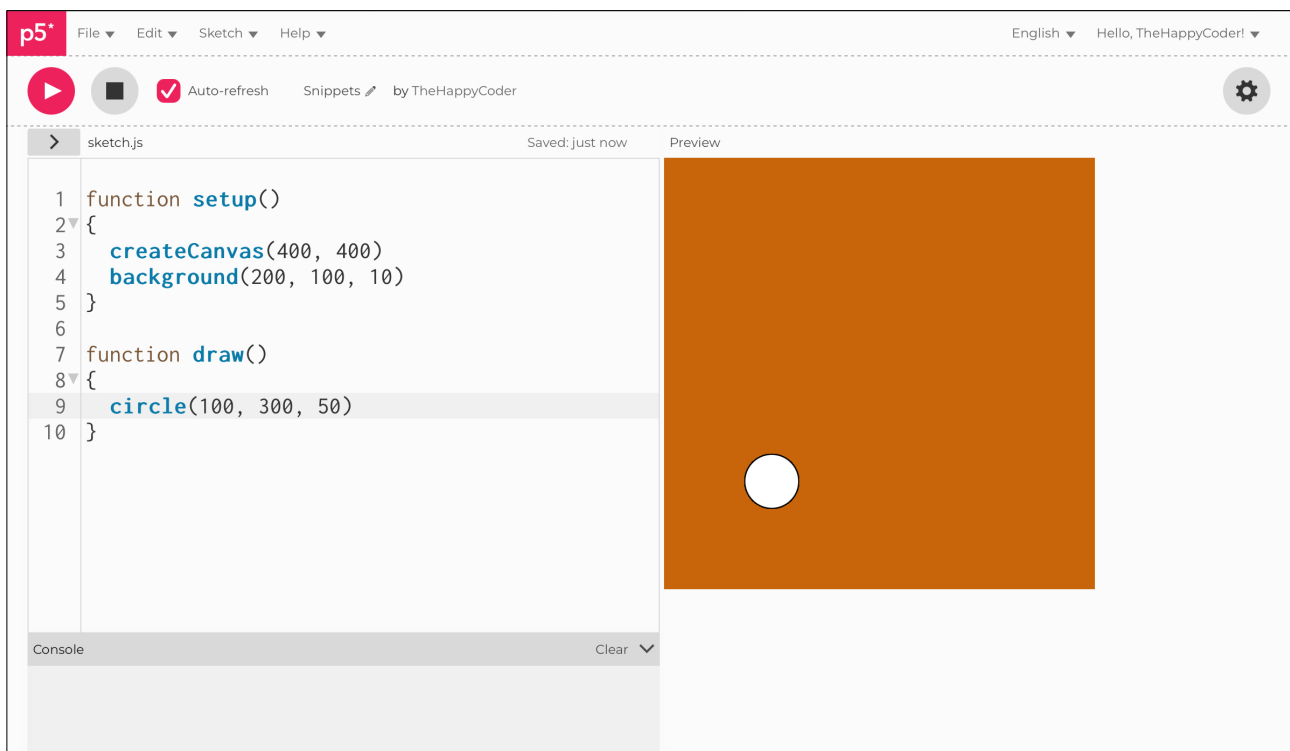
Try different diameters and co-ordinates.

Code Explanation

```
circle(100, 300, 50)
```

The circle has three arguments, the first is the x co-ordinate (100), the second is the y co-ordinate (300) and the third is the diameter (50)

Figure A4.6





Sketch A4.7 fill()

The `fill()` function allows you to fill the shape with a colour in the same way as the background, so it can take one or three arguments.

```
function setup()
{
  createCanvas(400, 400)
  background(200, 100, 10)
}

function draw()
{
  fill(250, 250, 10)
  circle(100, 300, 50)
}
```

Notes

We have created a nice yellow circle.

Challenge

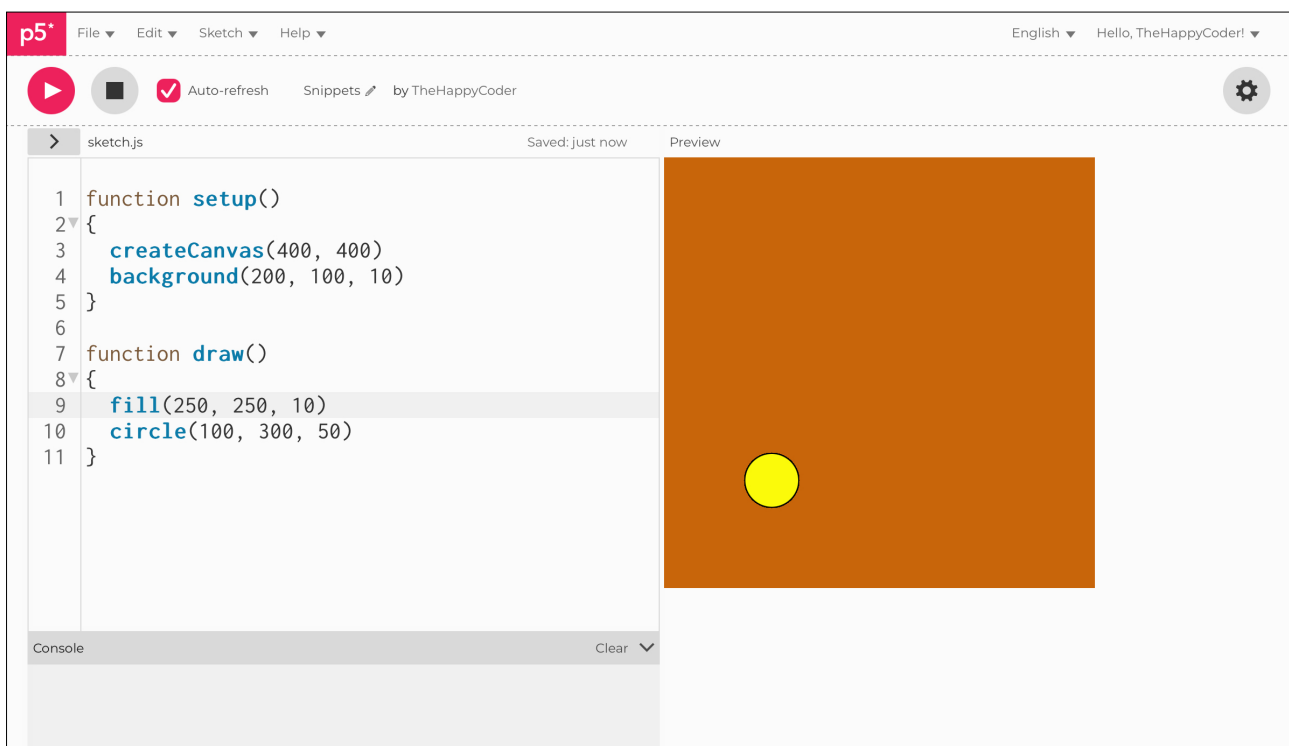
Try some other colours or greyscale.

Code Explanation

```
fill(250, 250, 10)
```

The fill() function takes one or three arguments for the colour

Figure A4.7





Sketch A4.8 strokeWeight()

We can change the thickness of the line around the circle with the `strokeWeight()` function. If you don't specify it, the default is `1`.

```
function setup()
{
  createCanvas(400, 400)
  background(200, 100, 10)
}

function draw()
{
  fill(250, 250, 10)
  strokeWeight(5)
  circle(100, 300, 50)
}
```


Notes

Sets the width of the stroke used for points, lines, and the outlines of shapes.

Challenge

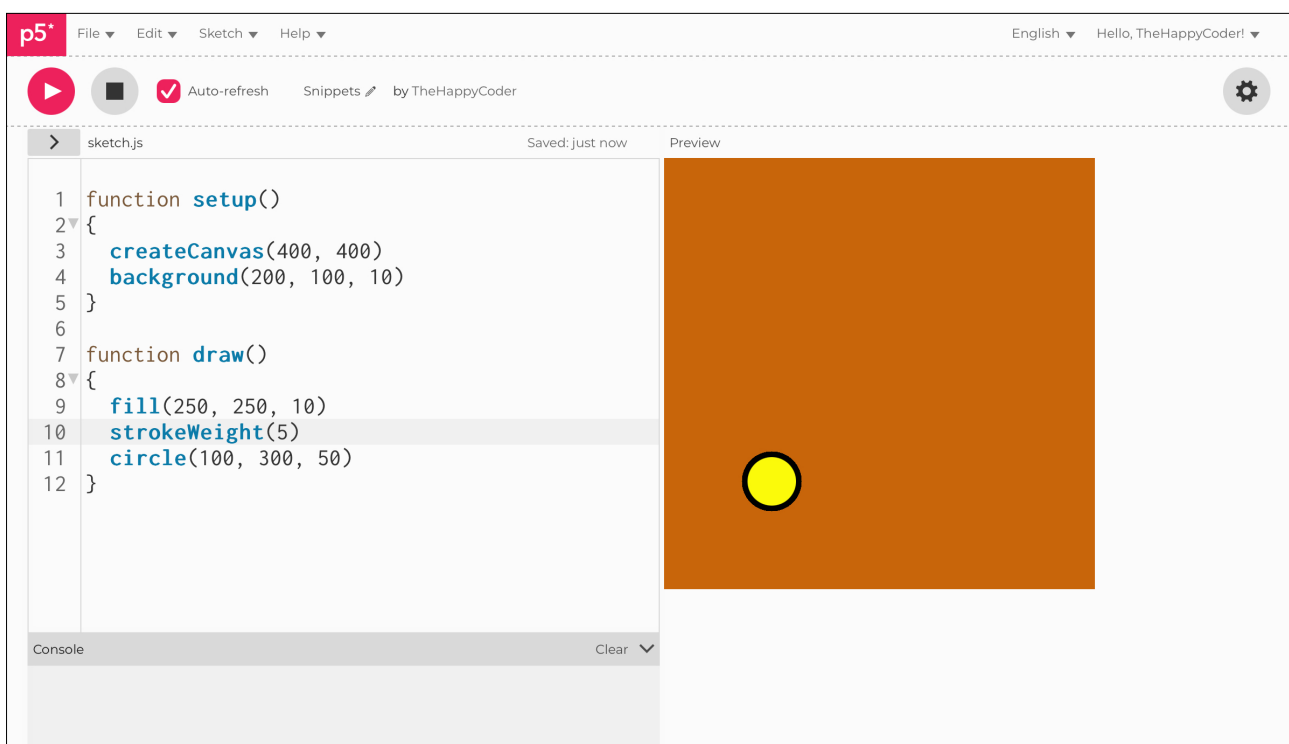
Try different stroke widths.

Code Explanation

```
strokeWeight(5)
```

We can give the thickness of a line a stronger weight, the default is 1

Figure A4.8





Sketch A4.9 stroke()

We can change the colour of the line with the `stroke()` function.

```
function setup()
{
  createCanvas(400, 400)
  background(200, 100, 10)
}

function draw()
{
  fill(250, 250, 10)
  strokeWeight(5)
  stroke(0, 0, 255)
  circle(100, 300, 50)
}
```

Notes

We get a nice blue line around the circle.

Challenge

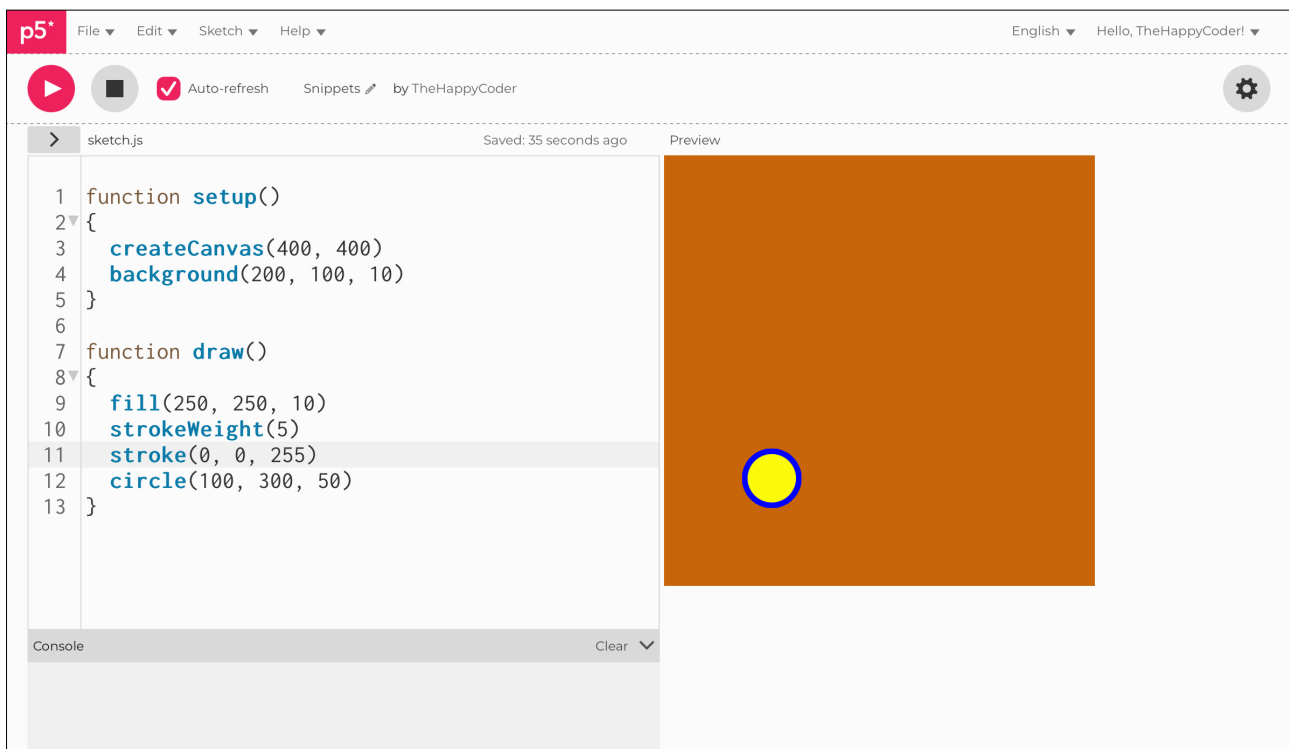
Try different `stroke()` colours, using the RGB values.

Code Explanation

```
stroke(0, 0, 255)
```

Giving the line an rgb value

Figure A4.9





Sketch A4.10 removing the line

Replacing the `stroke()` function with the `noStroke()` function removes the line altogether.

```
function setup()
{
  createCanvas(400, 400)
  background(200, 100, 10)
}

function draw()
{
  fill(250, 250, 10)
  strokeWeight(5)
  noStroke()
  circle(100, 300, 50)
}
```

Notes

Disables drawing points, lines, and the outlines of shapes. Calling `noStroke()` is the same as making the stroke completely transparent, as in `stroke(0, 0)`. If both `noStroke()` and `noFill()` are called, nothing will be drawn to the screen.

Challenge

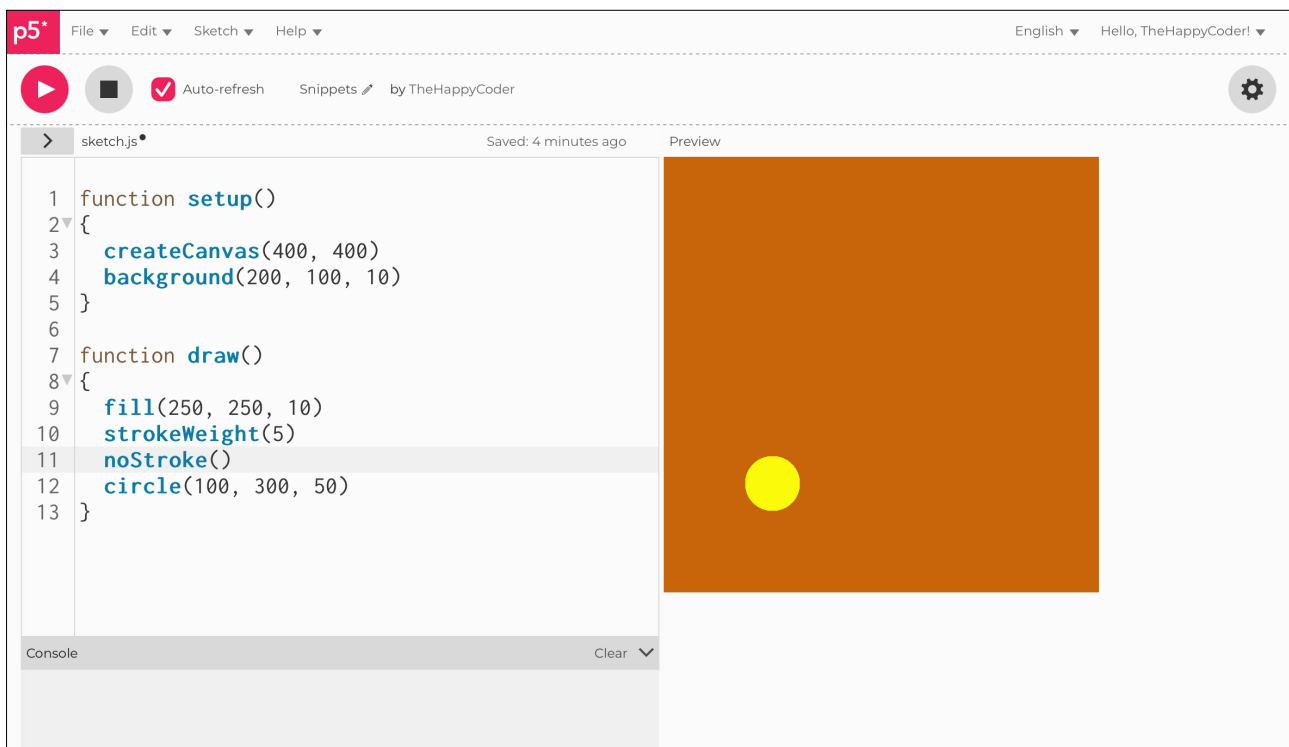
Try `stroke(0, 0)` and `noFill()`.

Code Explanation

`noStroke()`

The `noStroke()` just simply removes any line round a shape

Figure A4.10





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 variable 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 co-ordinates 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 keyword `let`. You can just define it or initialise it, for example:

<code>let x</code>	this defines x as a variable
<code>let x = 10</code>	this gives x an initial value of 10

Now in the sketch below, we are going to create a variable for the `x` co-ordinate (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 A4.11 let there be variables

Variables are a critical part of coding. To create a variable, you need to inform the computer programme. In p5.js (JavaScript), we use the word **let** before we name the variable. When we give the variable a name, it can be a single letter, a series of letters and numbers, or words. They cannot start with a number. To demonstrate this, we will replace the co-ordinates of the circle with variables **x** and **y**. At the same time, we will create a variable for the **diameter**.

```
let x = 100
let y = 300
let diameter = 50

function setup()
{
  createCanvas(400, 400)
  background(200, 100, 10)
}

function draw()
{
  fill(250, 250, 10)
  strokeWeight(5)
  noStroke()
  circle(x, y, diameter)
}
```

Notes

The circle is exactly the same as before. Because we called it a **let** variable, it means we have the option to change it later on in the code. We don't even need to give it an initial value, but it is a good idea to do so. When you create a variable, it is called **declaring** a variable; giving it a value is called **initialising** a variable.

A variable is a sort of container to hold a value. For example, a variable might contain the x-co-ordinate of a circle as a number or the name of someone as a string, where a string is a letter, series of letters and numbers, or words.

The scope of a variable depends on where you declare it. If you declare it in a function, it can only be used in that function; that is called a local scope. A global scope is where you declare a variable at the very start of the sketch outside any functions.

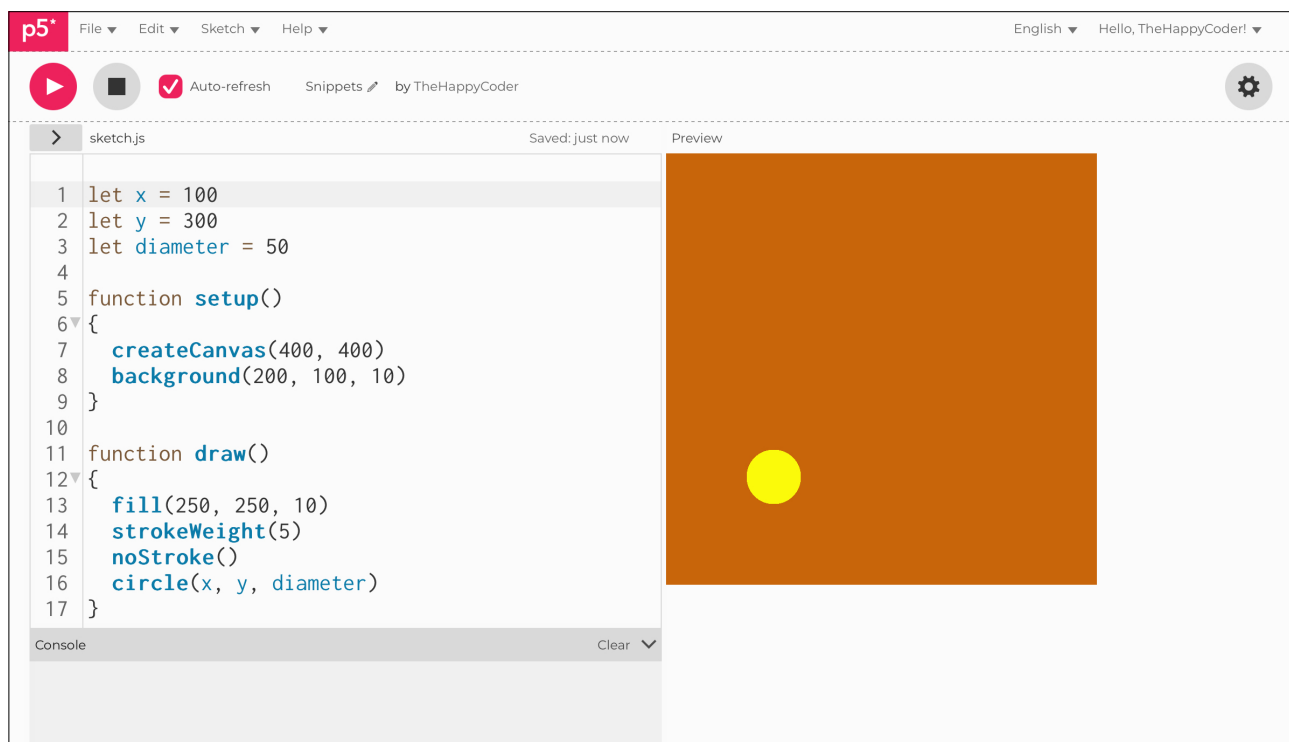
Challenge

1. Alter the x, y, and diameter variables.
2. Change the names of the variables.

Code Explanation

<code>let x = 100</code>	We create a variable called x and initialise it with a value of 100
<code>let y = 300</code>	We create a variable called y and initialise it with a value of 300
<code>let diameter = 50</code>	We create a variable called diameter and initialise it with a value of 50

Figure A4.11





Sketch A4.12 width and height

We can use other built-in variables. Two of these are called **width** and **height**. They know the width of the canvas and the height of the canvas. In this demonstration, we will replace the **x** and **y** values from the **x** and **y** variables with the width and height variable names. This is going to put the circle in the middle of the canvas, which is half the width and half the height, hence **width/2** and **height/2**.

```
let x = 100
let y = 300
let diameter = 50

function setup()
{
  createCanvas(400, 400)
  background(200, 100, 10)
}

function draw()
{
  fill(250, 250, 10)
  strokeWeight(5)
  noStroke()
  circle(width/2, height/2, diameter)
}
```

Notes

This puts the circle bang in the middle.

Challenges

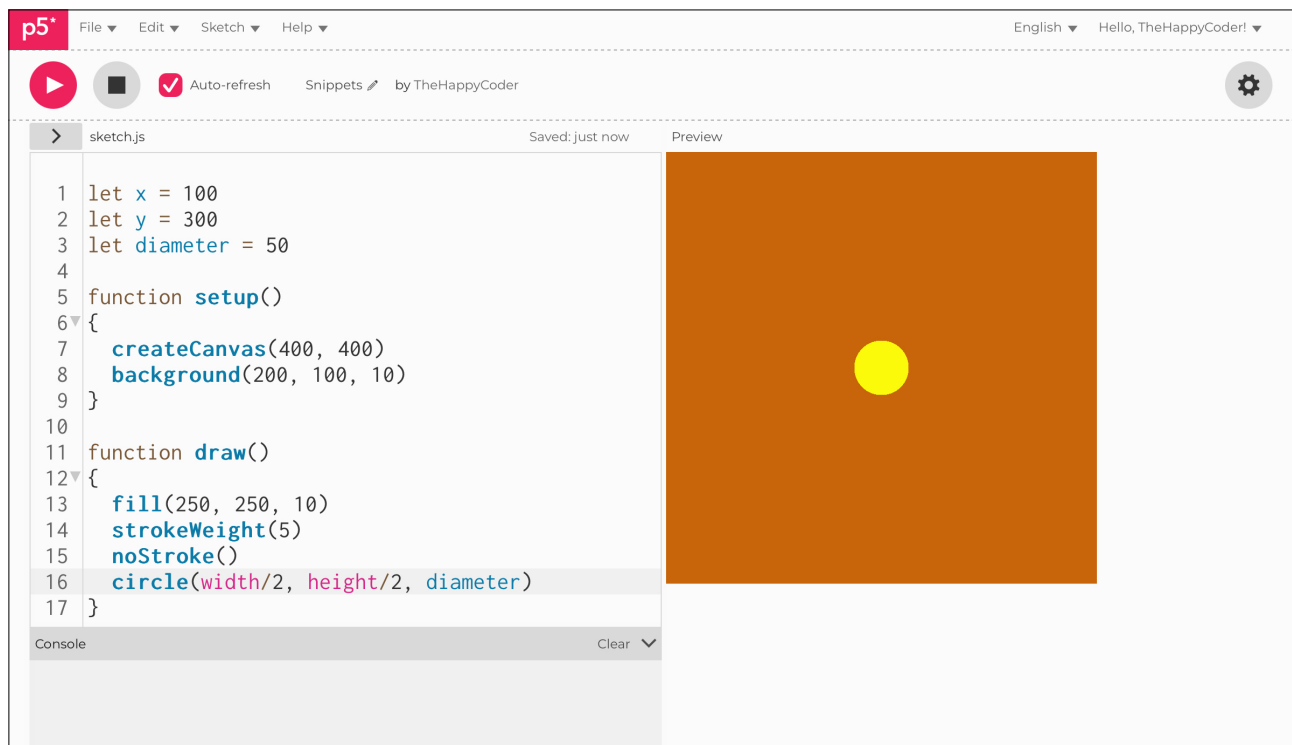
1. Change the dimensions of the canvas.
2. Change the position to `width/4` and `height/8`.
3. Change the diameter to `width/2`.

Code Explanation

```
circle(width/2, height/2, diameter)
```

This gives us an x position of half the width and a y position of half the height.

Figure A4.12





Sketch A4.13 mouseX and mouseY

Another useful built-in variable is `mouseX` and `mouseY`. They are the co-ordinates of the mouse pointer. We can move the centre of the circle to where the mouse currently is. The co-ordinates are taken from the top-left-hand corner.

```
let x = 100
let y = 300
let diameter = 50

function setup()
{
  createCanvas(400, 400)
  background(200, 100, 10)
}

function draw()
{
  fill(250, 250, 10)
  strokeWeight(5)
  noStroke()
  circle(mouseX, mouseY, diameter)
}
```

Notes

The circle will follow the mouse around the canvas. You will notice that it draws over the canvas as if with a paintbrush. This is because we draw the background once in `setup()` and the circle is drawn repeatedly over the top of a fixed canvas. If you put the background in the `draw()` function, it draws the background on every iteration. This highlights the difference between `setup()` and `draw()`.

Challenge

Change the diameter to `mouseX` or `mouseY`.

Code Explanation

```
circle(mouseX, mouseY, diameter)
```

`mouseX` is the x co-ordinate of the mouse pointer and `mouseY` is the y co-ordinate of the mouse pointer

Figure A4.13





Sketch A4.14 moving the background

! Remove `background(200, 100, 10)` from the `setup()` functioned now put `background(200, 100, 10)` in the `draw()` function instead.

You will see that it draws the background and the circle; it does this continuously.

```
let x = 100
let y = 300
let diameter = 50

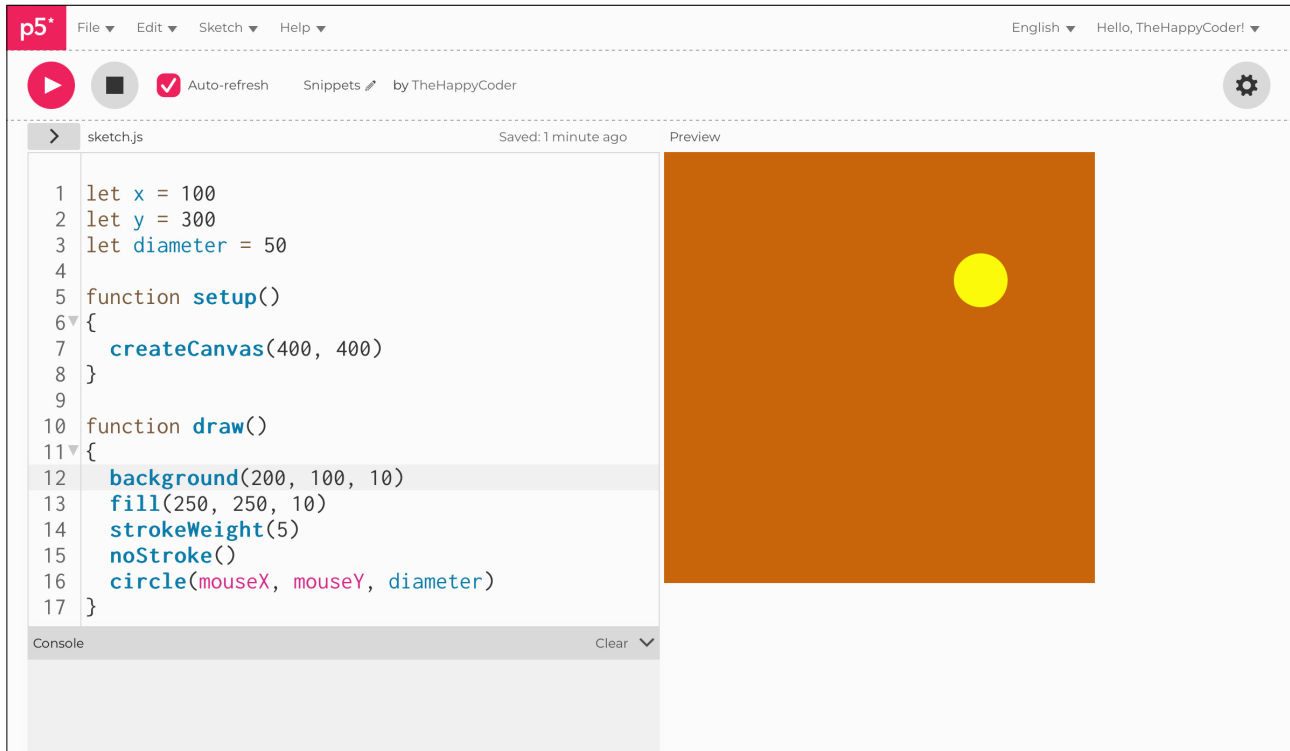
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(200, 100, 10)
  fill(250, 250, 10)
  strokeWeight(5)
  noStroke()
  circle(mouseX, mouseY, diameter)
}
```

Notes

When you move the mouse, the circle follows it.

Figure A4.14





Sketch A4.15 changing the values

! Remove the `strokeWeight()` and `noStroke()` functions.

In the `setup()` function, we are going to assign new values to the `x`, `y`, and `diameter` variables.

```
let x = 100
let y = 300
let diameter = 50

function setup()
{
  createCanvas(400, 400)
  x = 250
  y = 75
  diameter = 100
}

function draw()
{
  background(200, 100, 10)
  fill(250, 250, 10)
  circle(x, y, diameter)
}
```

Notes

We have a larger circle in a completely new position.

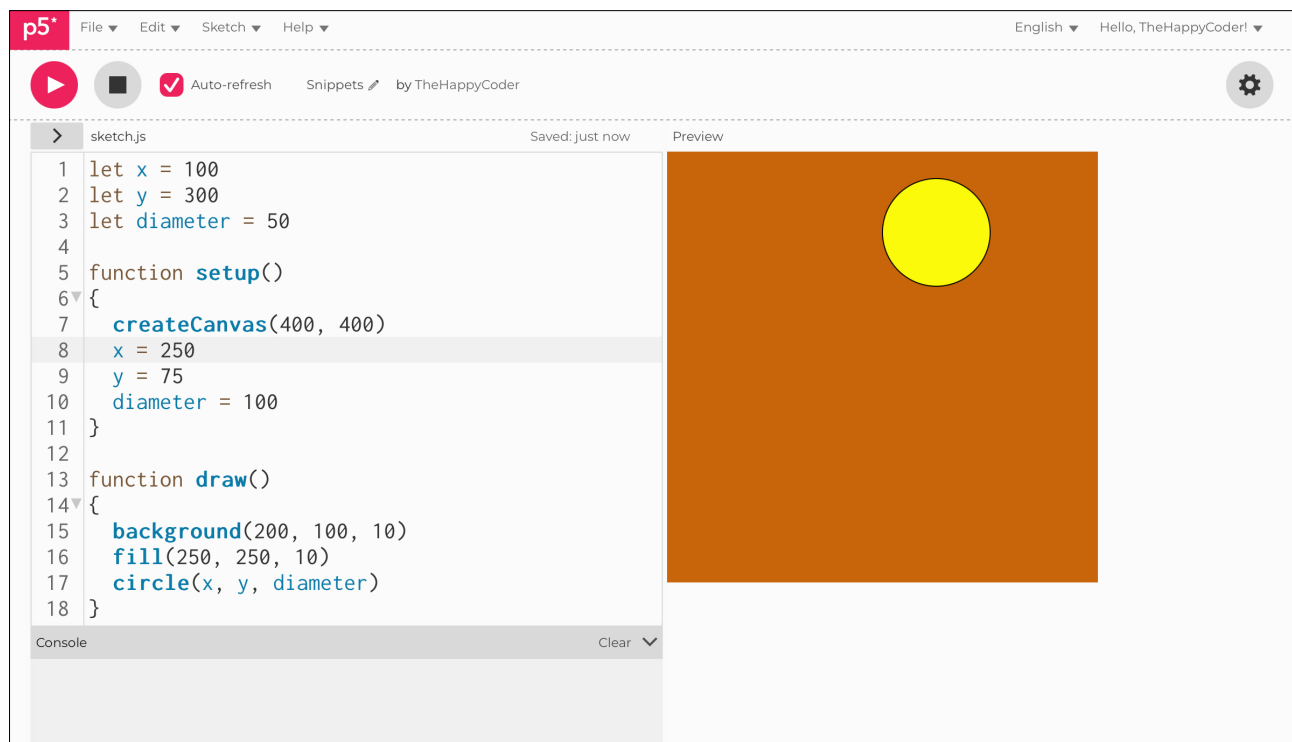
Challenge

Try other values.

Code Explanation

<code>x = 250</code>	Re-assigned the value of x from 100 to 250
<code>y = 75</code>	Re-assigned the value of y from 300 to 75
<code>diameter = 100</code>	Re-assigned the diameter from 50 to 100

Figure A4.15





Symbols we use

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

These perform basic mathematical operations; they are essential for numerical computations.

Comparison Operators

These compare values and return a boolean result (true or false).

Logical Operators

These combine boolean values to produce a single boolean result.

Assignment Operators

These are used to assign values to variables.

Maths Functions

Built-in functions that perform more complex mathematical operations.



Arithmetic Operators

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

<code>==</code>	means equal to (5 == 5) is TRUE, (6 == 5) is FALSE
<code><</code>	means less than (6 < 8) is TRUE, (8 < 6) is FALSE,
<code><=</code>	means less than or equal to (6 <= 8) is TRUE, (6 <= 6) is also TRUE
<code>></code>	means greater than (8 > 6) is TRUE, (6 > 8) is FALSE,
<code>>=</code>	means greater than or equal to (8 >= 6) is TRUE, (6 >= 6) is also TRUE
<code>!=</code>	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.

<code>&&</code>	means AND (<code>x < 100 && y > 50</code>)
<code> </code>	means OR (<code>x < 100 y > 50</code>)
<code>!</code>	Means NOT (<code>x < 100 ! y > 50</code>)



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

<code>floor()</code>	Calculates the closest integer value that is less than or equal to the value of a number
<code>abs()</code>	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 <code>abs(-5)</code> and <code>abs(5)</code> both return 5. The absolute value of a number is always positive.
<code>round()</code>	Calculates the integer closest to a number. For example, <code>round(133.8)</code> returns the value 134. The second parameter, <code>decimals</code> , is optional. It sets the number of decimal places to use when rounding. For example, <code>round(12.34, 1)</code> returns 12.3. It is zero by default.
<code>sq()</code>	Calculates the square of a number. Squaring a number means multiplying the number by itself. For example, <code>sq(3)</code> evaluates 3×3 which is 9. The <code>sq(-3)</code> evaluates -3×-3 which is also 9. Multiplying two negative numbers produces a positive number. The value returned by <code>sq()</code> is always positive.
<code>sqrt()</code>	Calculates the square root of a number. A number's square root can be multiplied by itself to produce the original number. For example, <code>sqrt(9)</code> returns 3 because $3 \times 3 = 9$. The <code>sqrt()</code> always returns a positive value. <code>sqrt()</code> doesn't work with negative arguments such as <code>sqrt(-9)</code> .
<code>pow</code>	Calculates exponential expressions such as 2^3 . For example, <code>pow(2, 3)</code> evaluates the expression $2 \times 2 \times 2$. <code>pow(2, -3)</code> evaluates $1 \div (2 \times 2 \times 2)$.
<code>ceil()</code>	Calculates the closest integer value that is greater than or equal to a number. For example, calling <code>ceil(9.03)</code> and <code>ceil(9.97)</code> both return the value 10.
<code>exp()</code>	Calculates the value of Euler's number e (2.71828...) raised to the power of a number.



Sketch A4.16 const

Sometimes we don't want a variable to change; if that is the case, we use constants. This is so we don't accidentally change something. For this, we use **const** rather than **let**. Watch what happens when we change it from the previous sketch.

```
const x = 100
const y = 300
const diameter = 50

function setup()
{
  createCanvas(400, 400)
  x = 250
  y = 75
  diameter = 100
}

function draw()
{
  background(200, 100, 10)
  fill(250, 250, 10)
  circle(x, y, diameter)
}
```

Notes

You get an error message when you try to alter it later in the code.

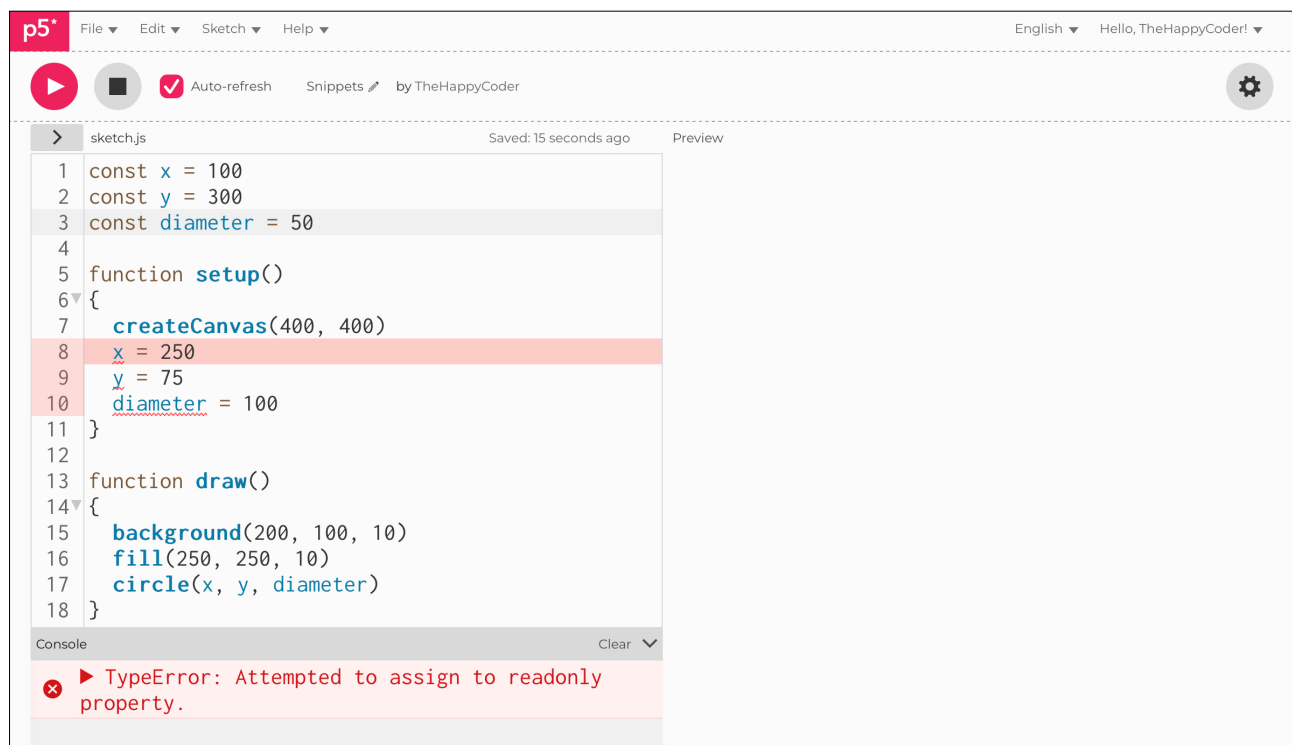
Challenge

Remove the `x = 250`, `y = 75` and `diameter = 100` reference in the `setup()` function. See if that changes the outcome.

Code Explanation

<code>const x = 100</code>	The x variable is now fixed
<code>const y = 300</code>	The y variable is now fixed
<code>const diameter = 50</code>	The diameter variable is now fixed

Figure A4.16





Sketch A4.17 for loop

! Start a new sketch or delete much of the previous sketch. Putting the `background()` back into the `setup()` function.

If we want to draw lots of circles, we could write a line of code for each circle, but there is a better way using loops. For this, we can use what is called a `for()` loop. The `for()` loop has three parts to it. You declare and initialise a variable, in this case `x`, with a starting value, in this case `20`. You then set the conditional limit of the value, in this case `400`, at which point the loop stops. The third and final part of the loop determines how much you are increasing (or decreasing) the variable on each iteration; in this case, we are increasing it by `40`.

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

function draw()
{
  for (let x = 20; x < 400; x += 40)
  {
    circle(x, 200, 20)
  }
}
```

Notes

You will get a line of ten circles across the canvas. You will find that `for()` loops are used a lot; after a while, they become very intuitive. Also, note that here we have to use semicolons.

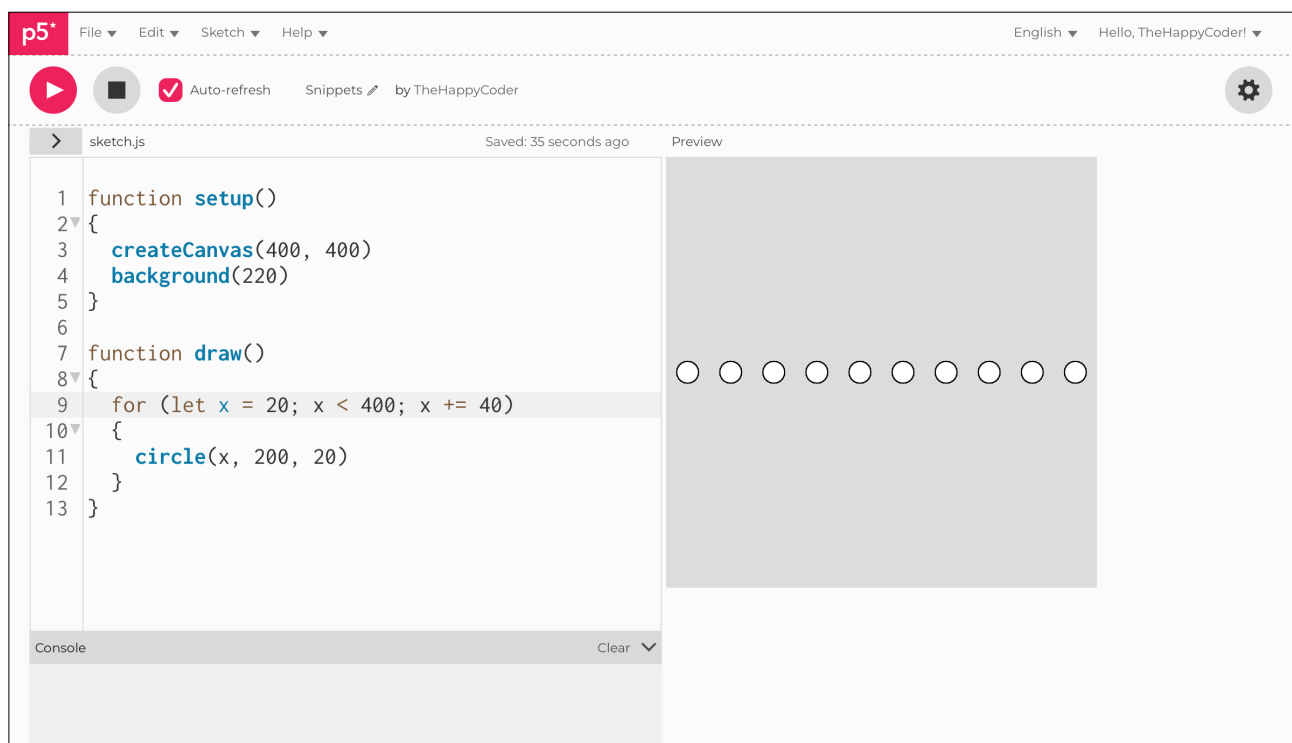
Challenge

Alter the values in the `for()` loop and see what effect it has. You might inadvertently create an infinite loop, so I recommend that you pause (stop) the code running while you make changes.

Code Explanation

<code>for ()</code>	The <code>for()</code> loop is continuous until a condition is met
<code>let x = 20;</code>	We define and initialise the variable being 20
<code>x < 400;</code>	We define the limit of the loop being 400
<code>x += 40</code>	We define the changes to the variable adding 40
<code>+=</code>	This is shorthand for <code>x = x + 40</code>

Figure A4.17





Sketch A4.18 nested loop

We can have a loop within a loop. In this way, we can also loop around the **y** value. The **for()** loop for the **y** variable looks exactly the same because it is. Also, remember to put the **y** variable into the **circle()** function.

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

function draw()
{
  for (let x = 20; x < 400; x += 40)
  {
    for (let y = 20; y < 400; y += 40)
    {
      circle(x, y, 20)
    }
  }
}
```

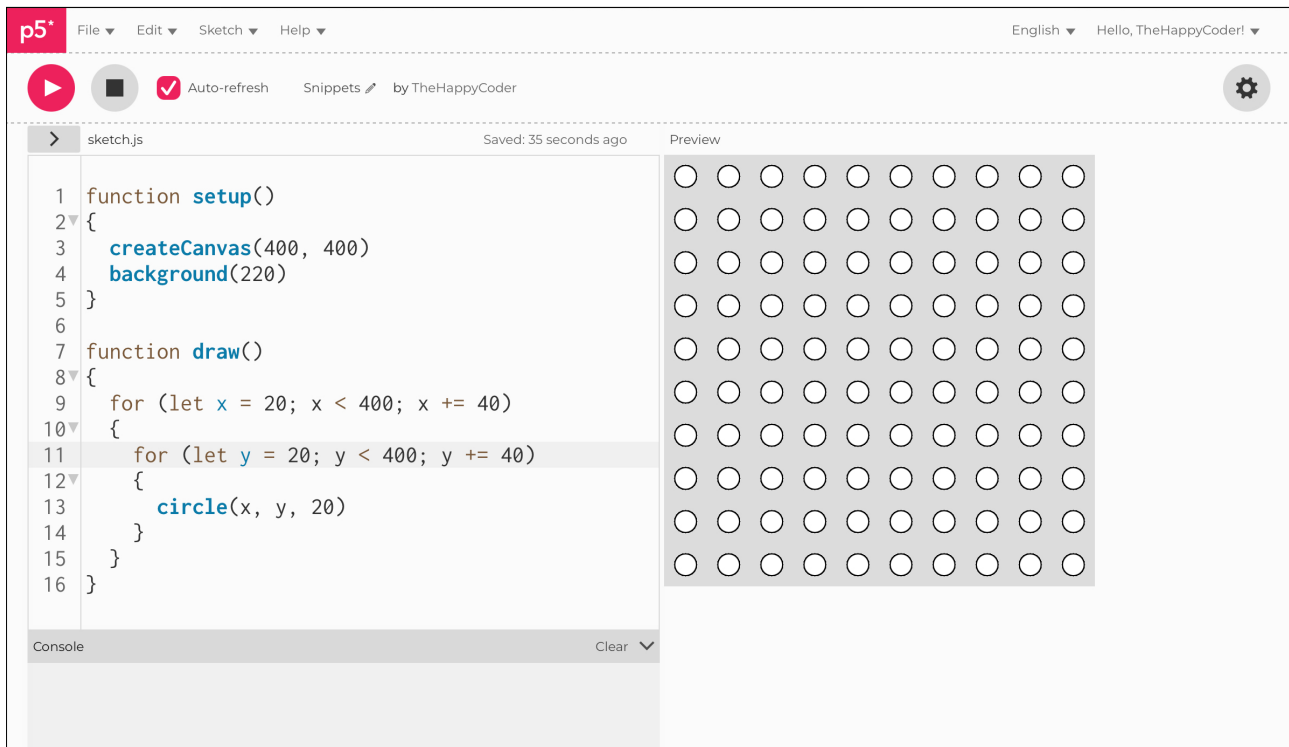
Notes

We have now filled the canvas with circles.

Challenge

Have a third nested loop so that the diameter increases for each circle.

Figure A4.18





Sketch A4.19 random()

! Remove the `for()` loops.

We can use a `random()` function when necessary to mix up the data. The `random()` function has two versions depending on how many arguments you have. If you have one argument, it will pick a random number between zero and that number; if you have two arguments, then it will pick a random number between those two numbers.

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

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

Notes

You see that the `draw()` function draws each circle at a random position on the canvas.

Challenge

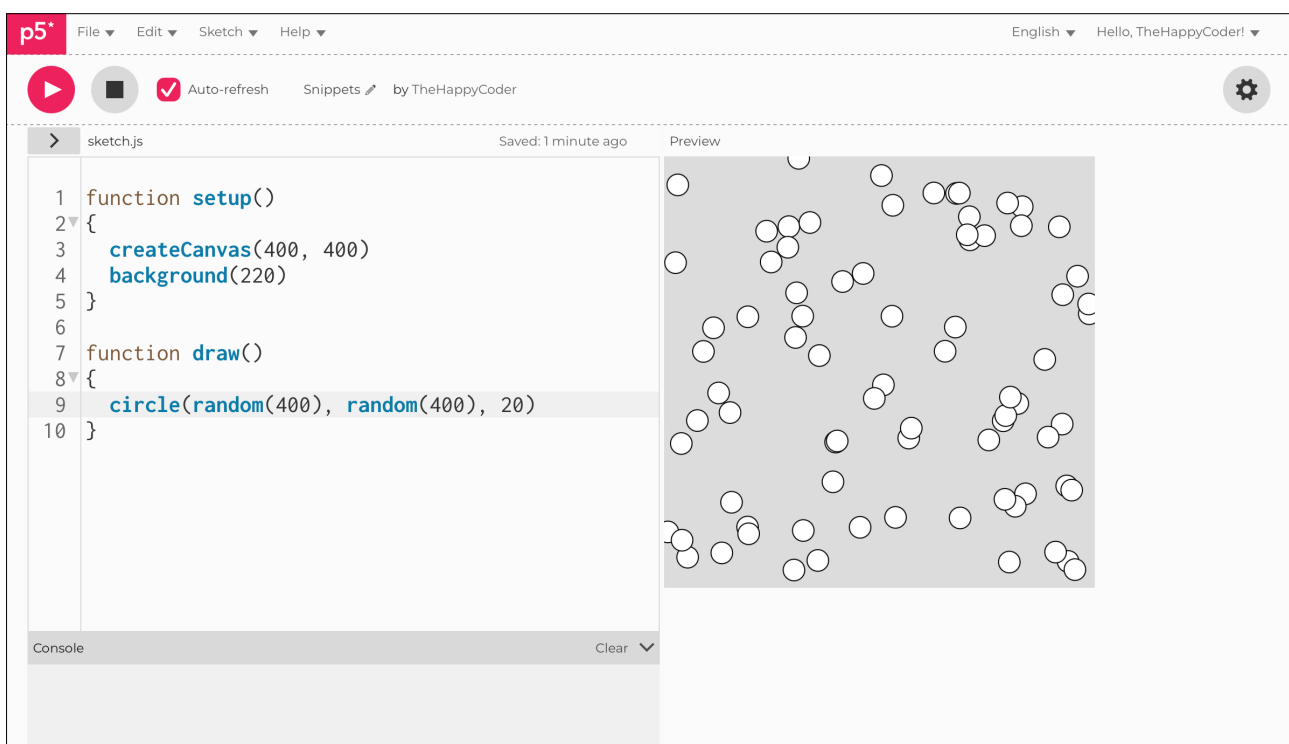
Change `random(400)` to `random(100, 300)` for both the `x` and `y` variables.

Code Explanation

`random(400)`

Picks a random number between 0 and 400

Figure A4.19





Sketch A4.20 if() statement

Another useful function is the `if()` statement. Here, something is `true` until it is `false` (and vice versa). When a condition is met, then the process ends; in this case, we will draw `100` circles in random positions. We use a variable to keep count of the number of circles drawn, called `count` (obviously).

```
let count = 0

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

function draw()
{
  if(count <= 100)
  {
    circle(random(400), random(400), 20)
  }
  count++
}
```

Notes

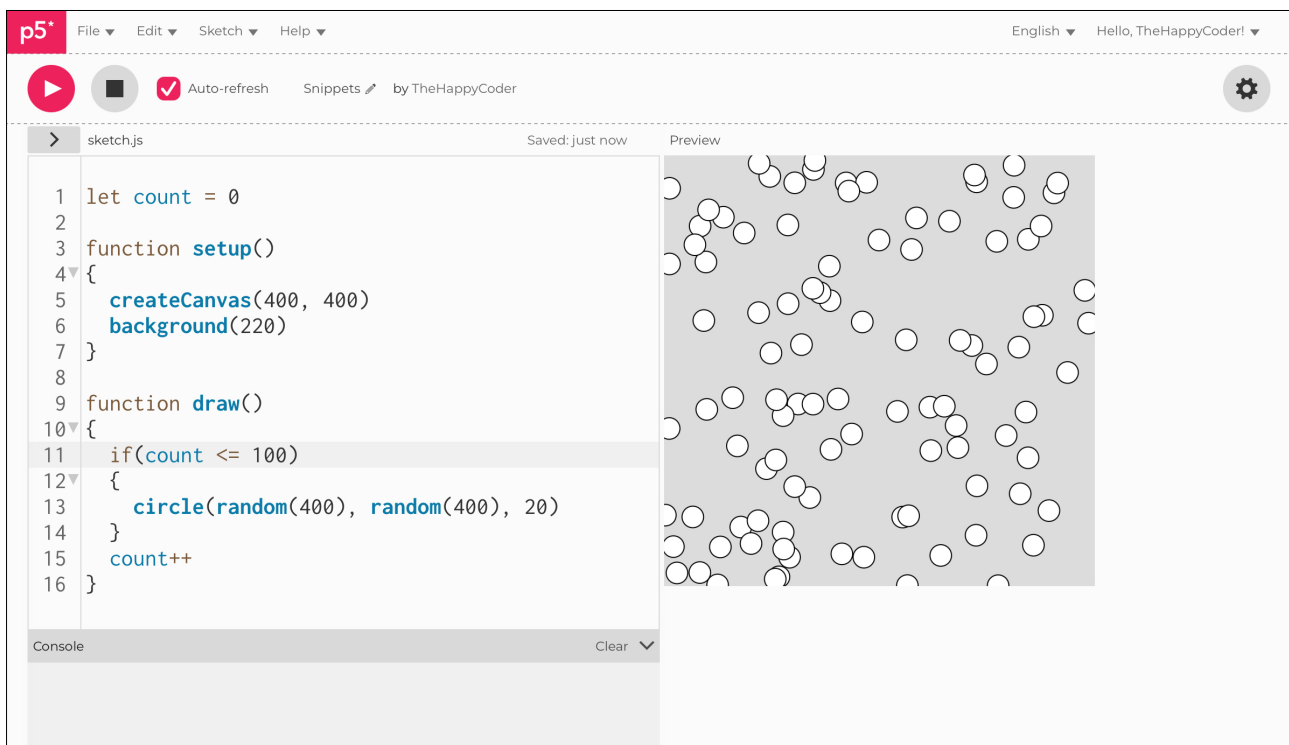
The programme keeps drawing the circles while the condition is true (**count** is less than **100**). Once that condition is false (there are **100** circles), it stops.

Code Explanation

`if(count <= 100)`

The `if()` statement where the condition is true if the `count` variable is less than 100, false if it is more than 100

Figure A4.20





Sketch A4.21 drawing a line

! New sketch.

We can draw a line as well as other shapes. The line has four arguments: the first two are the **x** and **y** co-ordinates of one end of the line (**100, 100**), and the other two are the **x** and **y** co-ordinates of the other end of the line (**300, 300**).

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

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

Notes

It draws a simple line.

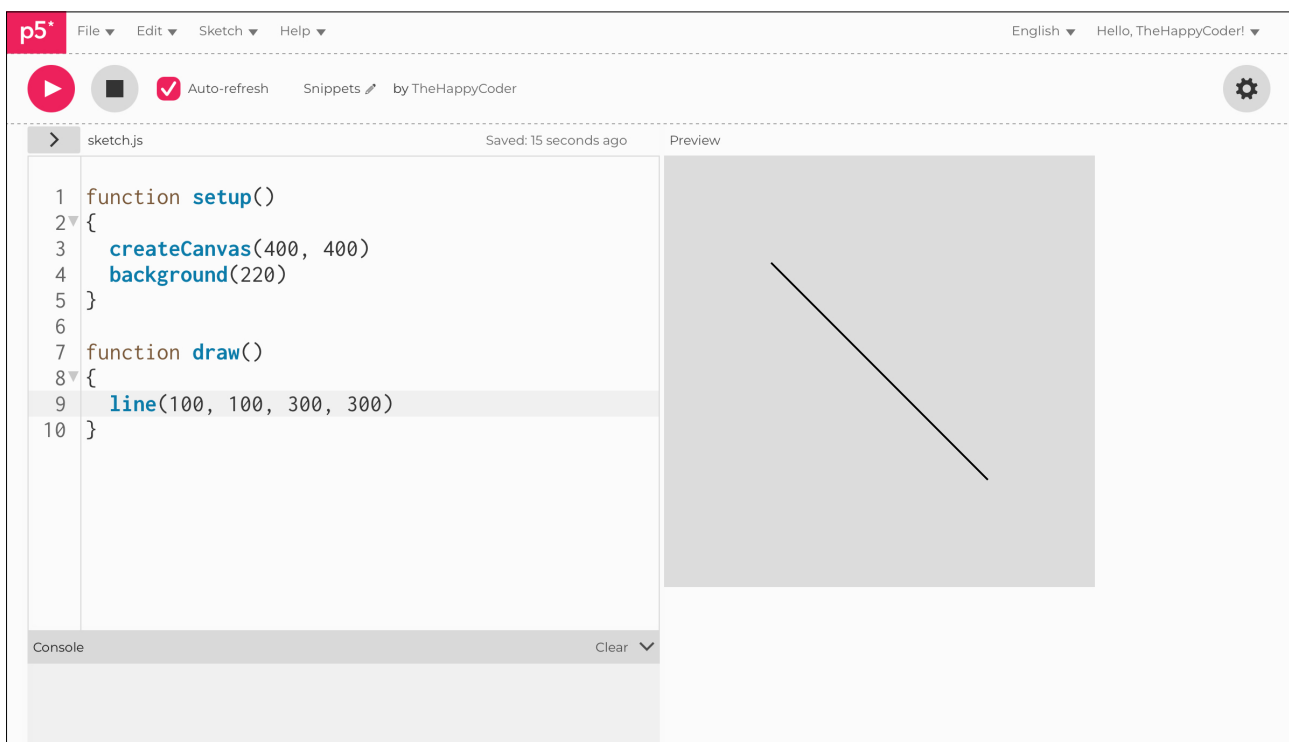
Challenge

Draw several lines, all starting from the centre of the canvas.

Code Explanation

<code>line(100, 100, 300, 300)</code>	Draws a line with co-ordinates (100, 100) and (300, 300) for each end
---------------------------------------	---

Figure A4.21





Sketch A4.22 stroke() and strokeWeight()

Adding a bit of colour and thickness.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  stroke(255, 0, 0)
  strokeWeight(5)
}

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

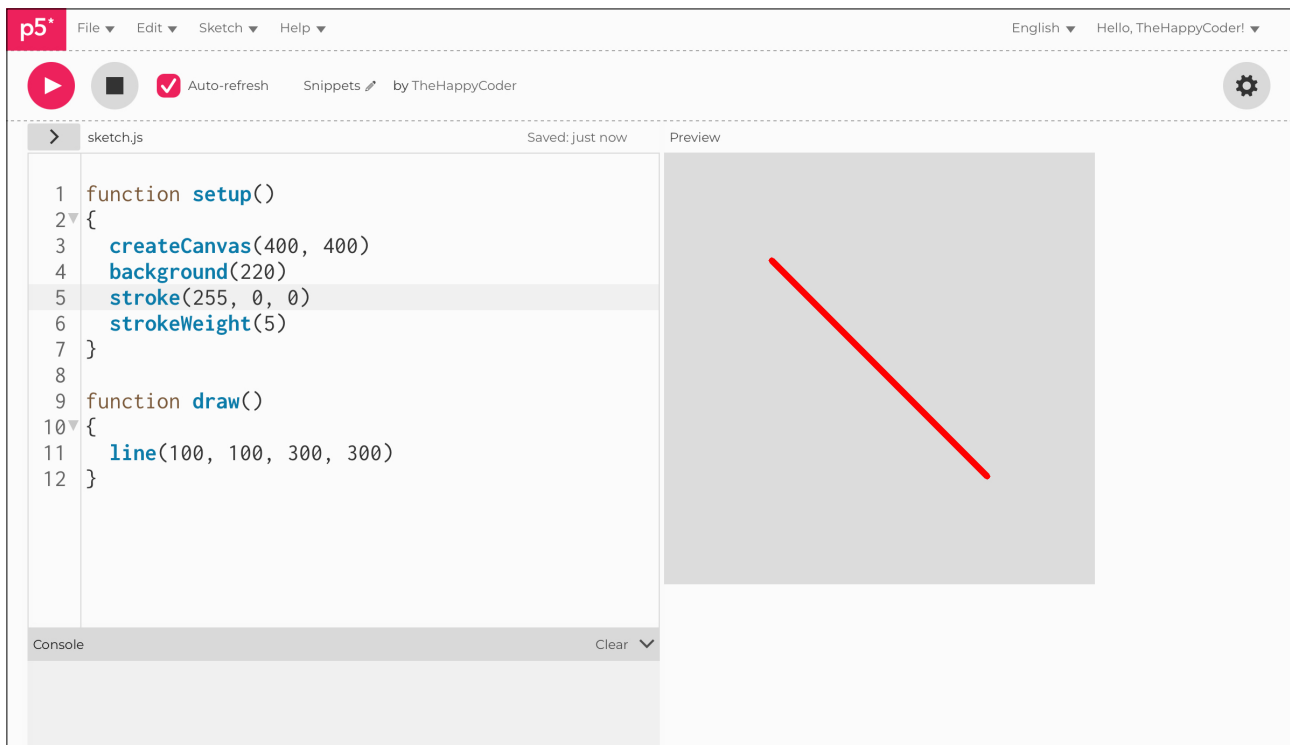
Notes

As with the circle, we can change the appearance of the line.

Challenge

Can you draw several lines in random positions, with random colours and strokes?

Figure A4.22





Sketch A4.23 point()

The `point()` function draws a pixel at the co-ordinates `(100, 300)`. As this will be very small, we will change the `strokeWeight(10)`.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  stroke(255, 0, 0)
  strokeWeight(10)
}

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

Notes

We have a `point` (or a pixel) at `(100, 300)`. The default is one pixel, which is too small to see easily; hence, to add a `strokeWeight()` of `10`.

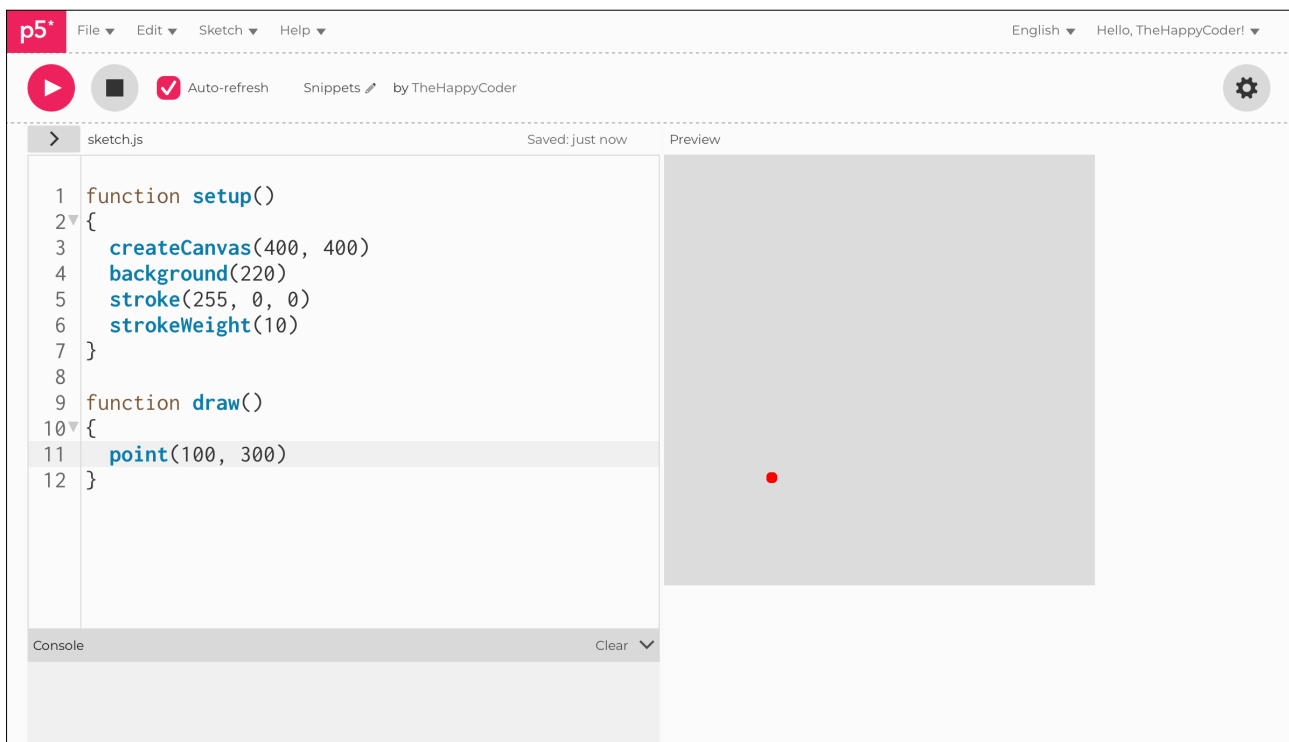
Challenge

Could you draw a line of pixels with `strokeWeight(1)` using a `for()` loop or an `if()` statement?

Code Explanation

<code>point(100, 300)</code>	Draws a pixel at those co-ordinates
------------------------------	-------------------------------------

Figure A4.23





Functions

In p5.js, functions are at the core of everything. You can tell that there is a function because it is prefixed with that word; for instance, `function setup()` or `function draw()`, these are prebuilt functions. We can, however, create our own functions and call them whatever we want (there are some keywords we cannot use because they are already taken).

We can also call a function inside another function, as you will see later. If you create a function, it won't necessarily do anything until you call it. By that, I mean you have to reference it in another function, usually the `draw()` function, but it can be in any other active function.

In the next example, we will create a function called `thing()` and then call it from the `draw()` function. In the example following that one, we will create a function called `changeColour()`, this is only called when we click on the button we have created. The function just sits there until it is called.



Sketch A4.24 creating a function

! A new sketch.

We create a function called **thing**, which should draw a circle. But nothing happens.

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

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

```
function thing()
{
  circle(200, 300, 100)
}
```


Notes

Although we have created a function, the computer reads it but will only action it when told to do so.

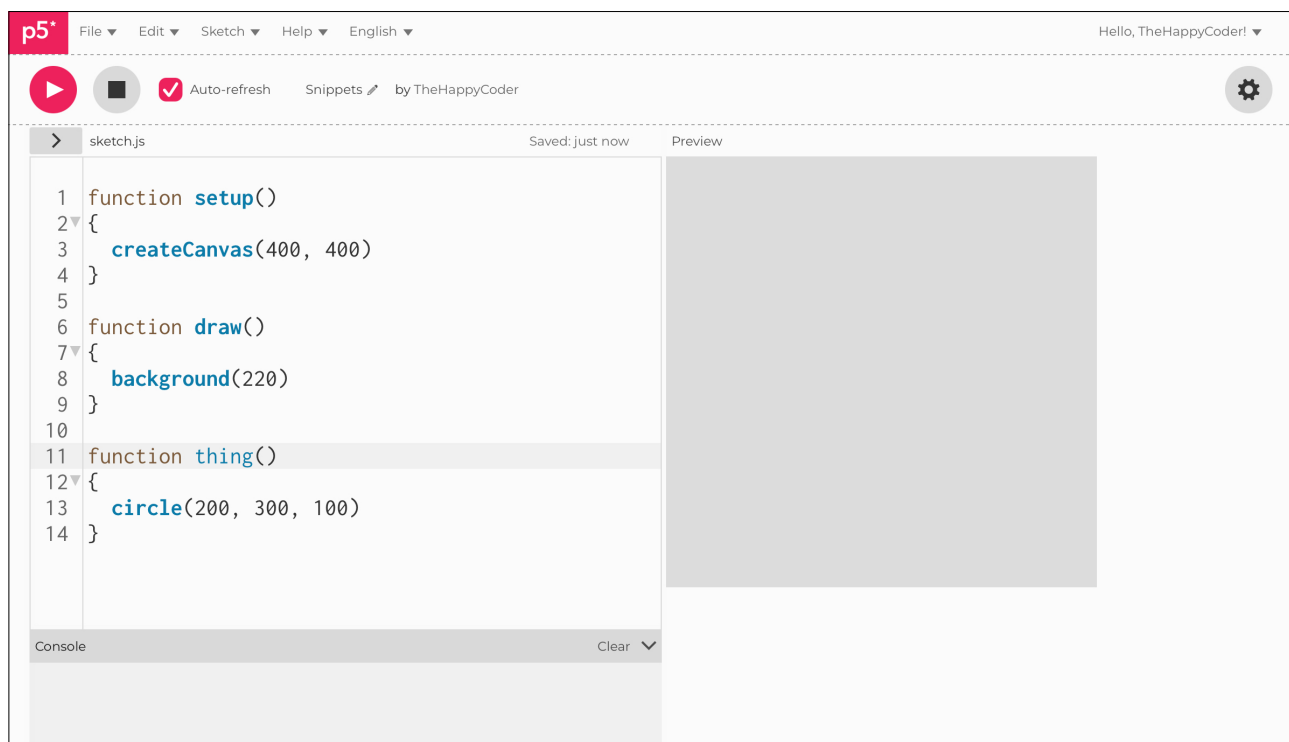
Challenges

1. Change the name of the function.
2. Have the function do something else.

Code Explanation

<code>function thing()</code>	Creating and naming a function
-------------------------------	--------------------------------

Figure A4.24





Sketch A4.25 calling a function

Now we call the `thing()` function in the `draw()` function.

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

function draw()
{
  background(220)
  thing()
}

function thing()
{
  circle(200, 300, 100)
}
```

Notes

Now the `thing()` function is actioned through the `draw()` function. It loops continuously.

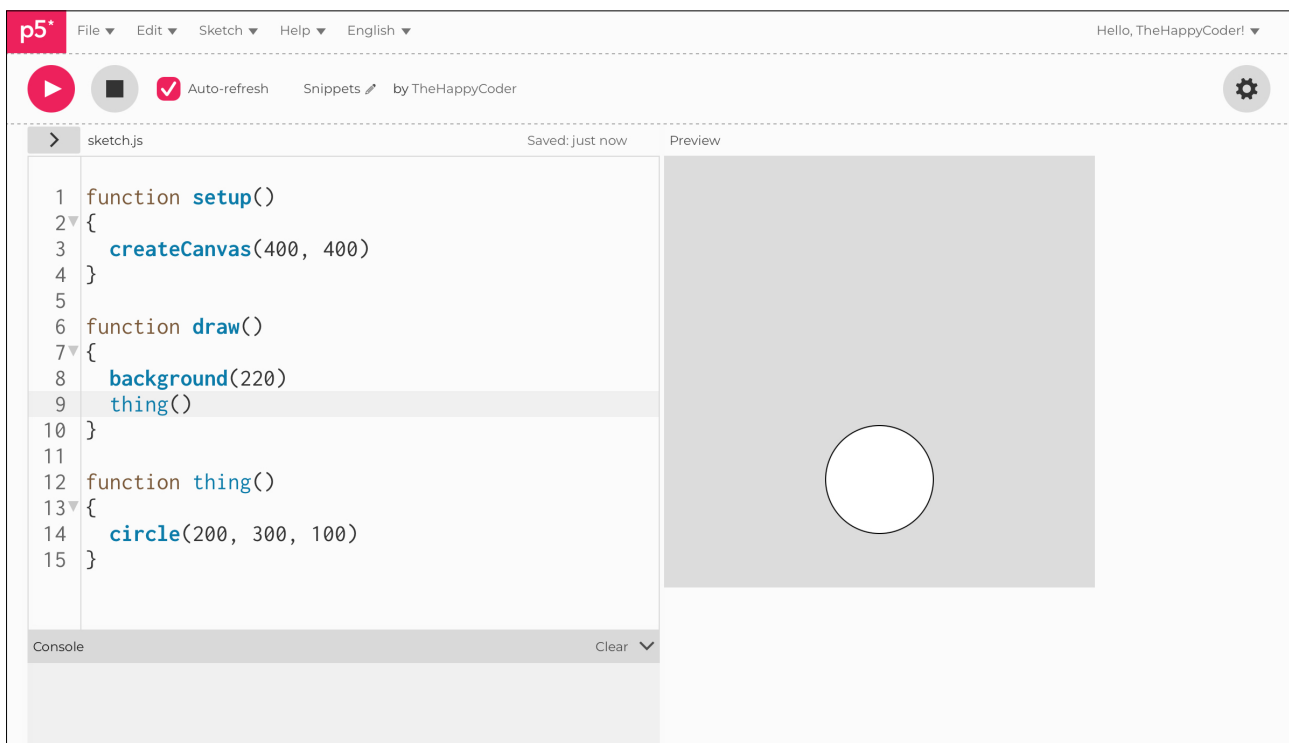
Challenge

Put the `thing()` function in `setup()`.

Code Explanation

<code>thing()</code>	Calling the function called 'thing'
----------------------	-------------------------------------

Figure A4.25





Sketch A4.26 creating a button (step 1)

! A new sketch.

We can create various DOM (Document Object Model) elements like buttons, radio buttons, sliders, and text boxes, which can change the appearance and make the canvas interactive. To start with, we are creating a button underneath the canvas with the words **"random colour"** written on it.

```
let button

function setup()
{
  createCanvas(400, 400)
  button = createButton('random colour')
  button.style('font-size', '30px')
}

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

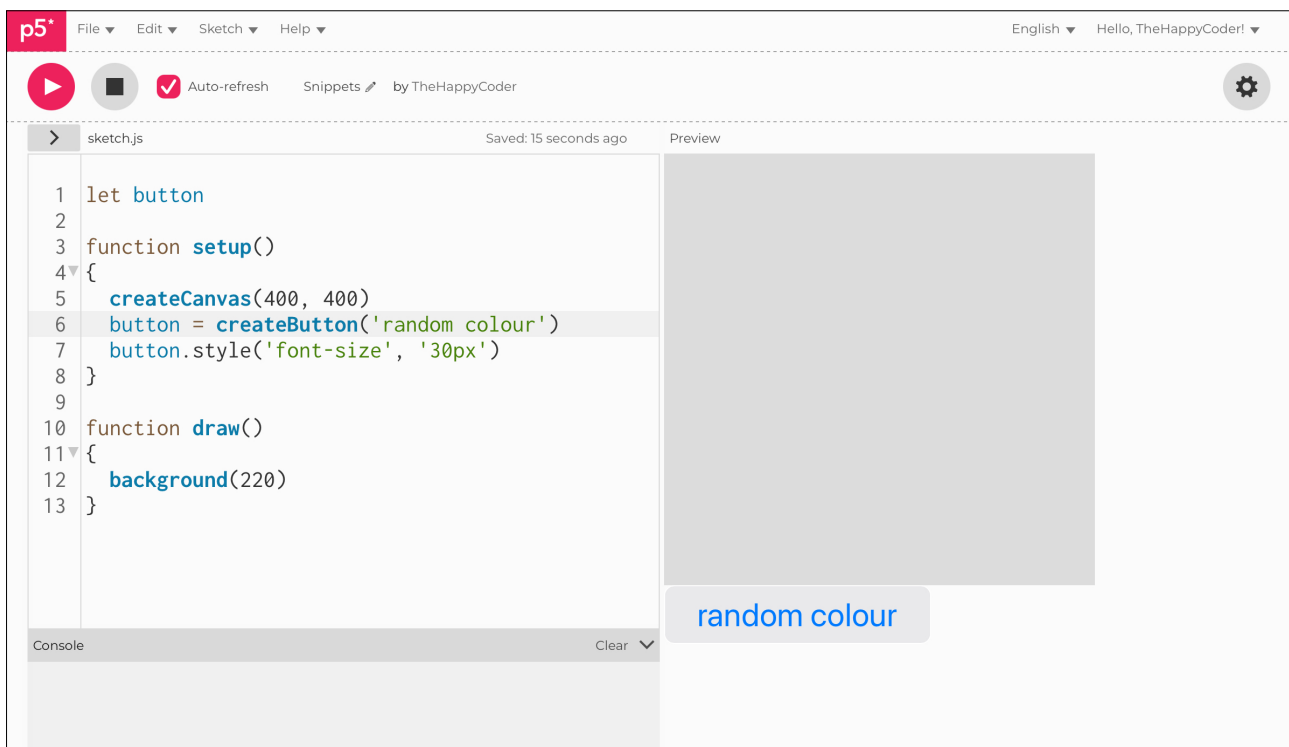
Notes

We have a button that you can click on, but nothing happens. We need another function to execute when the button is clicked.

Code Explanation

<code>let button</code>	Create a button variable
<code>button = createButton('random colour')</code>	This creates a button with the text inside it
<code>button.style('font-size', '30px')</code>	Change the text size inside button

Figure A4.26





Sketch A4.27 creating a button (step 2)

Now we will need a function so that when the button is clicked, it will change the background. The new function will be called `changeColour()`. We add the `mousePressed()` function to the button, and when it is clicked, it will then call that function. We need a variable to hold the colour, and we call that `newColour` with an initialised value of `220`.

```
let button
let newColour

function setup()
{
  createCanvas(400, 400)
  newColour = 220
  button = createButton('random colour')
  button.style('font-size', '30px')
  button.mousePressed(changeColour)
}

function draw()
{
  background(newColour)
}

function changeColour()
{
  newColour = random(255)
}
```

Notes

Now the background changes (greyscale) every time you click on the button, giving you a greyscale between 0 and 255.

Code Explanation

<code>button.mousePressed(changeColour)</code>	This calls the function 'changeColour' when the button is clicked with the mouse
<code>function changeColour()</code>	This is new function we have created which is activated when we click on the button
<code>newColour = random(255)</code>	We get a new random value between 0 and 255, this is the new colour
<code>background(newColour)</code>	Instead of the value we can replace with a variable name



Introducing arrays

An array is a key and vital part of coding. It is a way of storing data so that it can be accessed, added to, or altered at a later date. One way to think of it is as a series of boxes that can hold bits of data (whether numbers or words). An array has a numbering system for each box, called an **index**. In coding, counting starts with **zero**, not **one**. So the first box is **index 0**, the second box is **index 1**, the third is **index 2**, and so on. You can imagine that it can be confusing that the third box is index 2.

The format to identify an array is square brackets such as **[23, 15, 37, 42, ...]** so we can describe this array as follows:

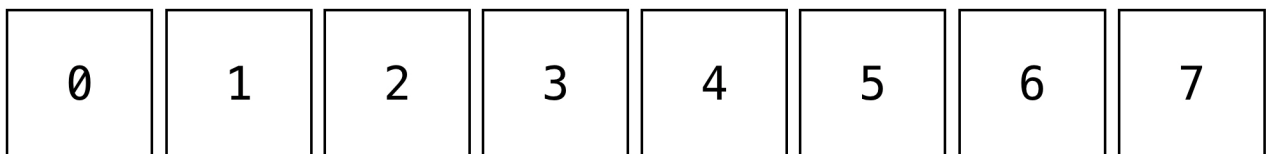
index[0] is **23**,

index[1] is **15**,

index[2] is **37**, and

index[3] is **42**, etc.

Index numbering system



The values (elements) inside each box



On the top row are the **index[]** references, and on the bottom row are the actual values at those reference points. We need to give the array a name. We can use **let** to define the array, such as:

```
let numbers = []
```

This is an empty array.

We can initialise it with some initial values if we want.

```
let numbers = [23, 15, 37, 42, 8, 51, 22, 99]
```

The array has now got some values in it



Sketch A4.28 an empty array

! We are starting with a new sketch.

We are starting with an empty array called `data`. This means it can be any size, and we can put data into it. The data can be numbers, words, or objects.

In this example, we are going to fill an empty array with `x` and `y` values. Arrays have square brackets `[]` around them. We are calling our array `data`, and we are going to put data into it using a `push()` function. We push the values of the `x` and `y` coordinates into the `data` array.

As we click on the canvas, you will see something happen in the console; this is because we have used a very useful function called `console.log()` and inside the brackets, we have put the name of the array. What you see in the console is what is inside the array.

```
let data = []

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

function mousePressed()
{
  let x = mouseX
  let y = mouseY
  data.push(x, y)
  circle(x, y, 50)
  console.log(data)
}
```

Notes

Have a look inside each array by clicking on the arrow at the start of the line. In this example, there are six elements, two for each of the circles. Just for clarification, the name of the array (**data**) is made up; we could call it anything (as long as it isn't a keyword already taken by p5.js).

Challenge

Change the name of the array.

Code Explanation

<code>let data = []</code>	We define an empty array
<code>let x = mouseX</code>	Our x value is the mouseX position when we click
<code>let y = mouseY</code>	Our y value is the mouseY position when we click
<code>data.push(x, y)</code>	The x and y values are pushed into the data array
<code>console.log(data)</code>	We can see inside the data array

Figure A4.28a

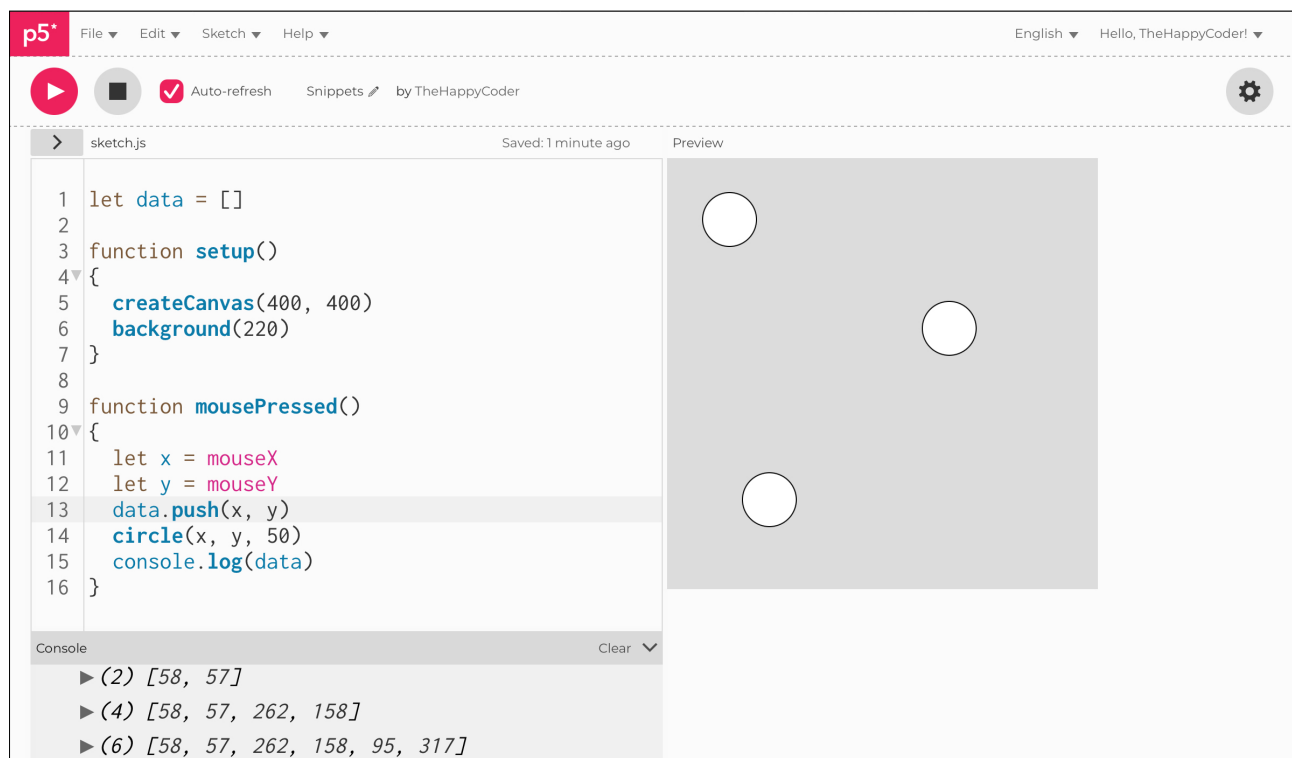
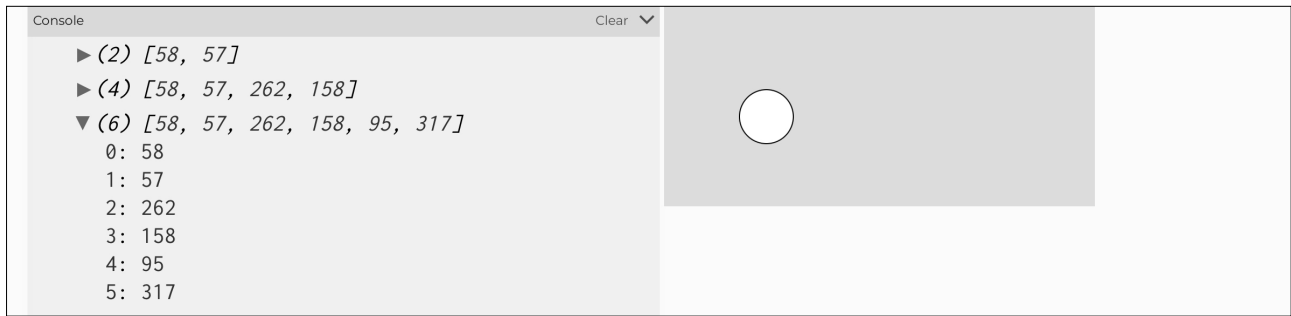


Figure A4.28b





Sketch A4.29 an array of objects

Another way to use an array is to have objects. In this example, an object is one circle. Each object has an **x** and **y** value. We will use `console.log(data)` as before to see the difference. Each object is called **inputs** to collect the **x** and **y** co-ordinates when you click on the canvas.

```
let data = []

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

function mousePressed()
{
  let inputs = {
    x: mouseX,
    y: mouseY
  }
  data.push(inputs)
  circle(inputs.x, inputs.y, 50)
  console.log(data)
}
```

Notes

We use `console.log(data)` to see inside the array. It gives you a list of objects, each one representing a circle. Inside each object are the `x` and `y` inputs. Click on the arrow to reveal the contents of the array.

Challenge

Change the variable names of `x` and `y` to, say, `flower` and `animal`. See where else you have to change them.

Code Explanation

<code>let inputs = {...}</code>	We give the collective name for the x and y values as inputs
<code>x: mouseX</code>	For the x value we use a colon then the input value or variable
<code>y: mouseY</code>	For the y value we use a colon then the input value or variable
<code>circle(inputs.x, inputs.y, 50)</code>	The x co-ordinate of the object (inputs) is inputs.x, repeated for the y component

Figure A4.29a

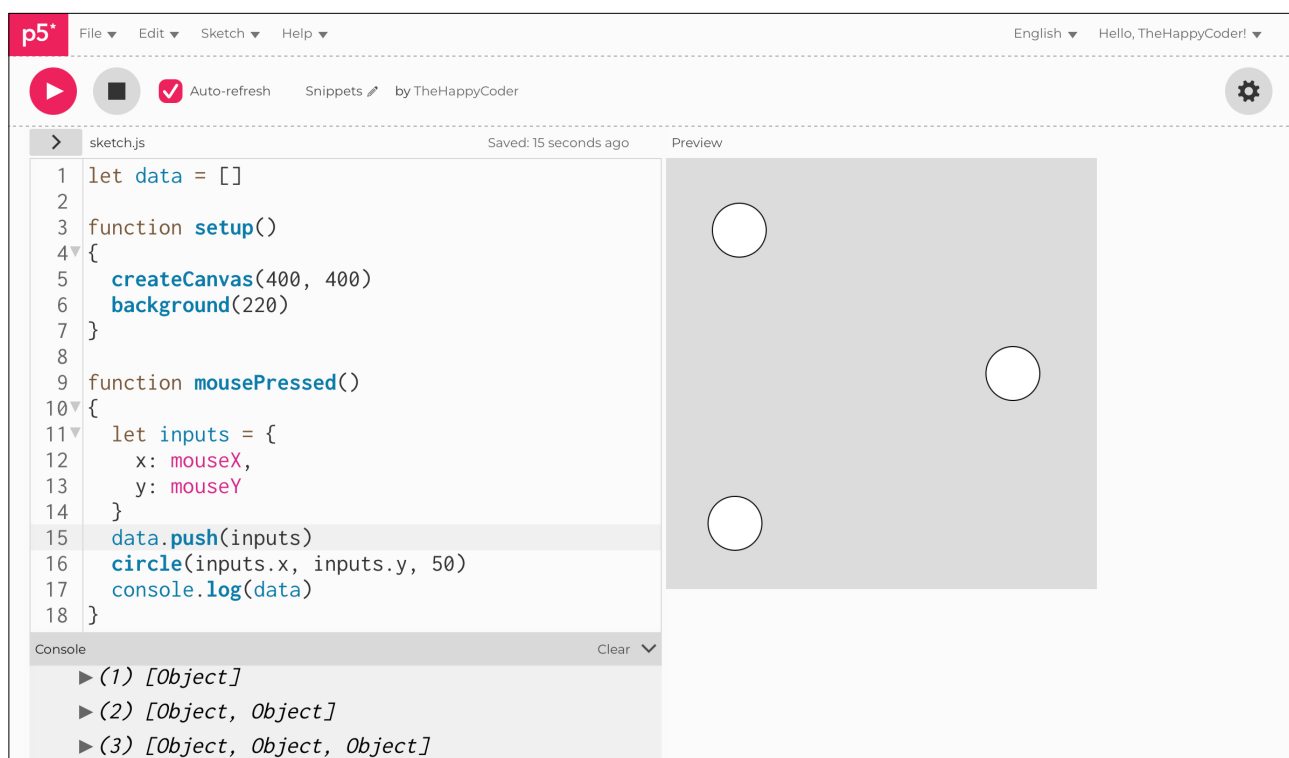


Figure A4.29b





Sketch A4.30 createVector()

! We are starting a completely new sketch.

This creates a new `p5.Vector` object. A vector can be thought of in different ways. One perspective is: a vector is like an arrow pointing in space. Vectors have both magnitude (length) and direction. This view is helpful for programming motion. A vector's components determine its magnitude and direction. For example, calling `createVector(3, 4)` creates a new `p5.Vector` object with an `x` component of `3` and a `y` component of `4`. The length is measured from the origin; this vector's tip is `3` units to the right and `4` units down.

Another really useful bit of coding is the `comments` symbol `//`. This is useful for a number of reasons:

Reason 1: to leave information in the code for yourself or someone else.

Reason 2: you can remove a line of code (without deleting it completely) to see what happens if that code is removed (for debugging a problem).

Reason 3: keep a bunch of code that you want to use later but not need it at that precise moment.

The computer bypasses anything when a line starts with the `//` before any text; it just ignores it.

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

  // Create p5.Vector objects
  let p1 = createVector(100, 100)
  let p2 = createVector(200, 200)
  let p3 = createVector(300, 300)

  // draw circles
  // fill(255, 0, 0)
  circle(p1.x, p1.y, 50)
  circle(p2.x, p2.y, 50)
  circle(p3.x, p3.y, 50)

  // draw points
```

```

strokeWeight(6)
point(p1)
point(p2)
point(p3)
}

```

Notes

We have drawn three circles, then three points.

Challenge

If you `console.log(p1)`, you can see inside the vector object for that vector. See the image below; it is rather confusing, but you can see the difference in the array between numbers and objects (object-oriented programming).

Code Explanation

<code>// Create p5.Vector objects</code>	Using the comments symbol, the programme ignores it, useful for information
<code>// fill(255, 0, 0)</code>	This line of code is also ignored, if you remove the <code>//</code> the programme will execute it.
<code>let p1 = createVector(100, 100)</code>	Creates a vector object with an x and a y component and attributes it to a variable (p1 in this case)
<code>circle(p1.x, p1.y, 50)</code>	Uses the x and y components of the object that is within the p1 variable
<code>point(p1)</code>	Drawing the point extracting the x and y components from the variable object

Sketch A4.30a

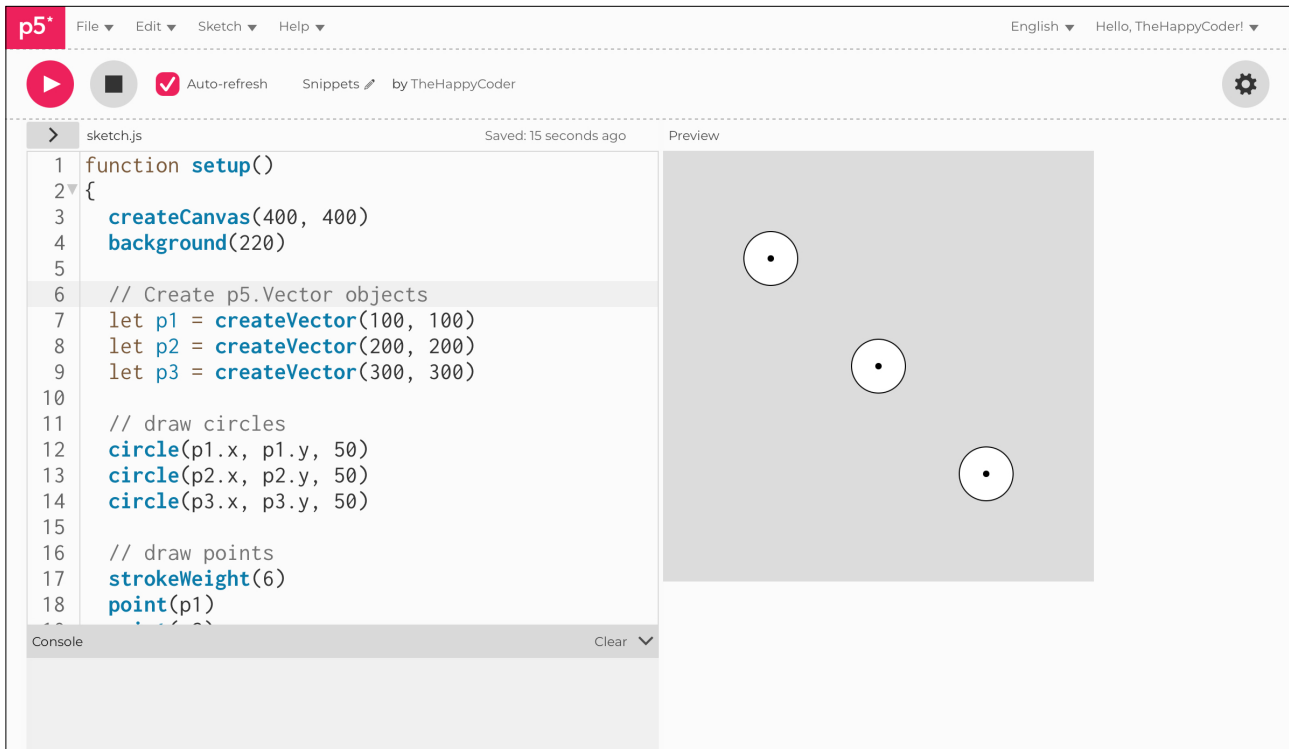
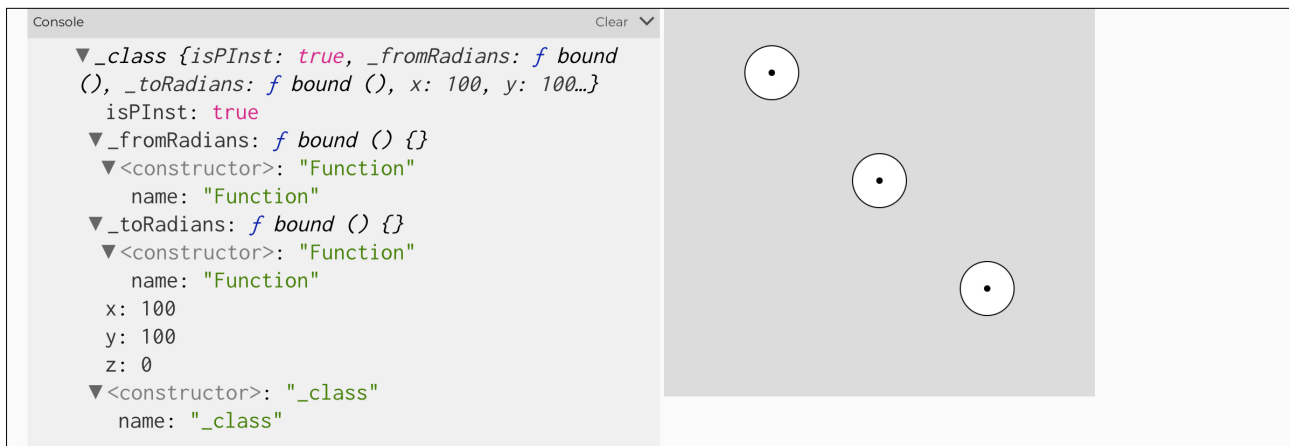


Figure A4.30b





Sketch A4.31 map() function

! We are starting another new sketch.

We can remap values, scaling them through the `map()` function. In this example, we have a line that follows the movement of the mouse (`mouseX`) from one side to the other. The second line is mapped from `[0, 400]` to `[0, 100]`. It still follows the movement of the mouse but is now scaled accordingly.

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

function draw()
{
  background(220)
  strokeWeight(10)
  line(0, 150, mouseX, 150)
  let x = map(mouseX, 0, 400, 0, 100)
  line(0, 250, x, 250)
}
```

Notes

This is just an illustration of what mapping is. One way to imagine it is: if you have a scale between 27 and 187 and you want to express it as a percentage, then mapping can scale it such that 0% is 27 and 100% is 187.

Challenge

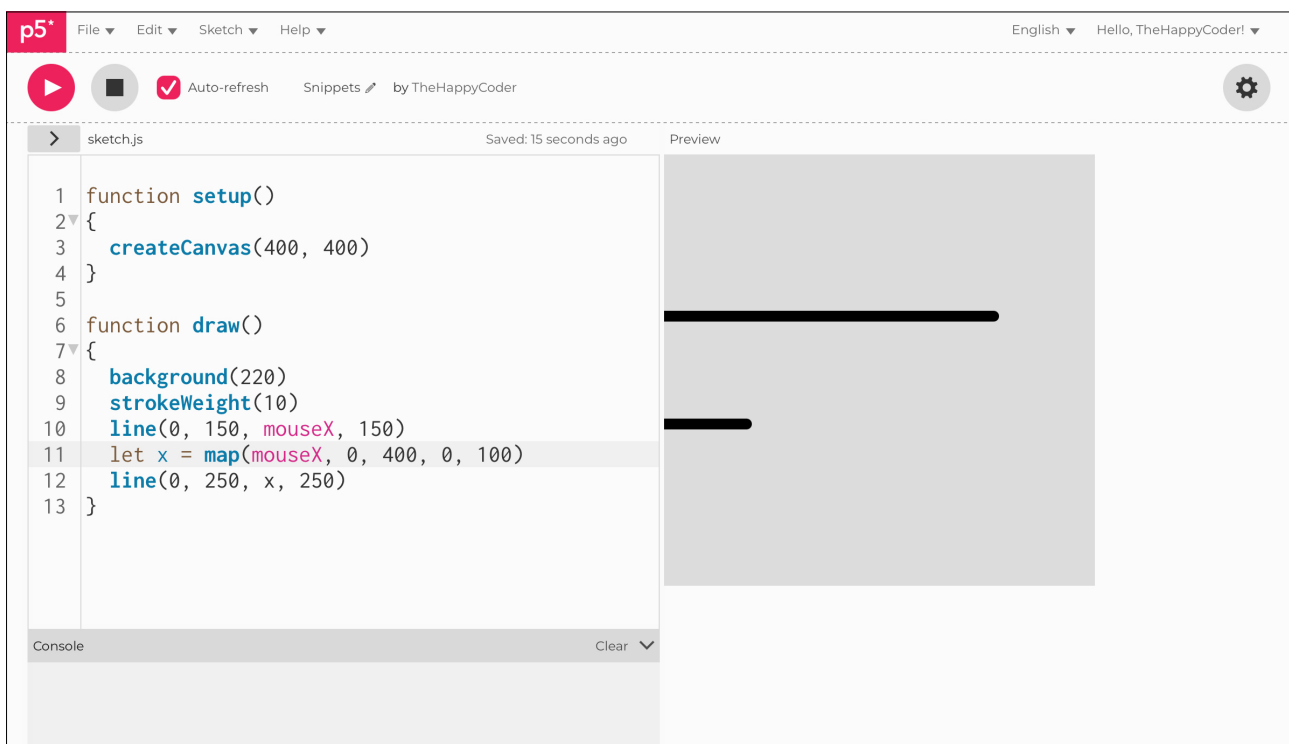
Apply the `map()` function to the diameter of a circle.

Code Explanation

```
let x = map(mouseX, 0, 400, 0, 100)
```

This maps the value of the mouseX value from 0-400 to 0-100 and returns the value to the x variable

Figure A4.31





Sketch A4.32 sin() function

! Again we start with a new sketch.

We have at our disposal the mathematical `sin()` function. The output is always between `-1` and `+1`. The input is measured in radians by default, but to make it more intuitive, we change the units to degrees with the `angleMode()` function. We make the canvas width `360` (as in a 360° cycle) and we map the output from `(-1, 1)` to `(0, height)` to be more visible.

```
function setup()
{
  createCanvas(360, 400)
  angleMode(DEGREES)
  strokeWeight(5)
}

function draw()
{
  background(220)
  for (let x = 0; x < width; x++)
  {
    let y = sin(x)
    y = map(y, -1, 1, 0, height)
    point(x, y)
  }
}
```

Notes

We get a nice sine wave.

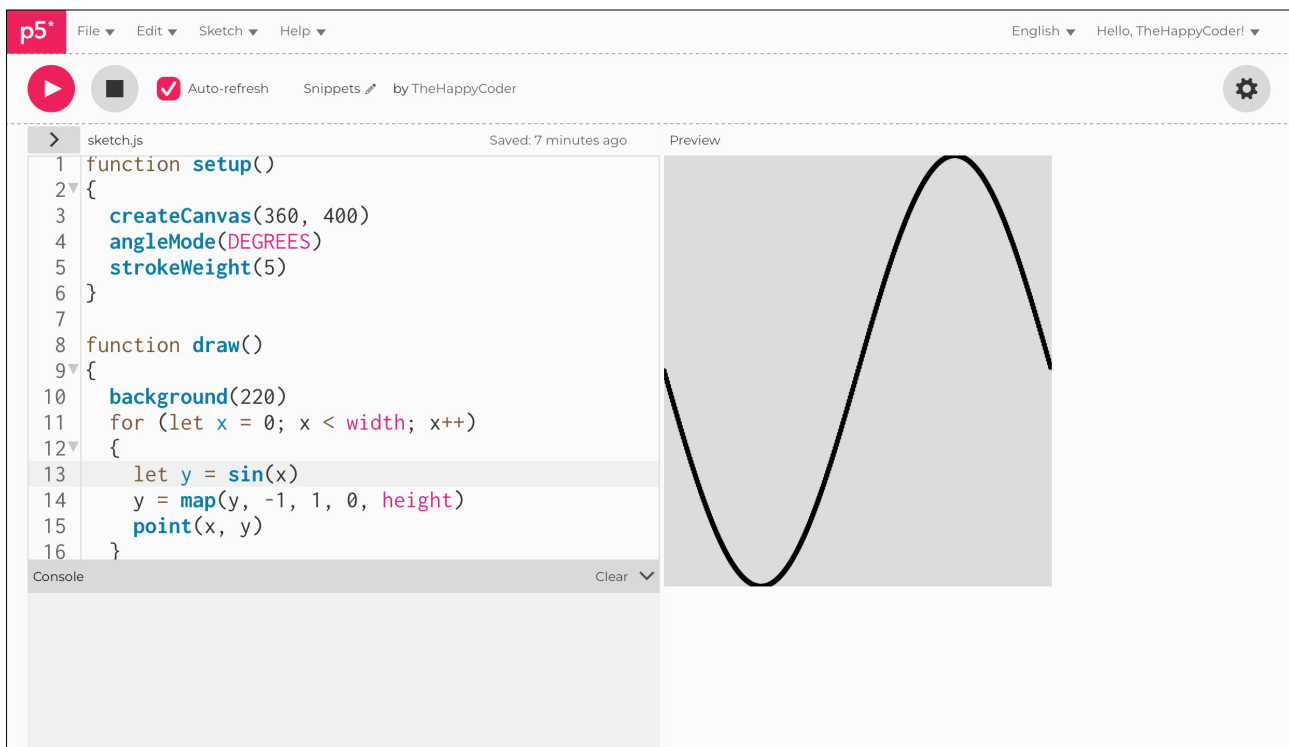
Challenge

Try `cos()`.

Code Explanation

<code>angleMode(DEGREES)</code>	Converts the angle units from radians (default) to degrees.
<code>let y = sin(x)</code>	The y variable is the result of the sin() function on the x value

Figure A4.32





Sketch A4.33 alpha

! New sketch again.

With colour, we can add a fourth argument, which is the alpha value, which determines how much transparency there is. With a value of **0**, it is completely transparent; with a value of **255**, it is completely opaque.

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

function draw()
{
  fill(255, 0, 0, 100)
  circle(random(width), random(height), 50)
}
```

Notes

Creating lots of random red circles with an alpha of 100. This alpha also works with greyscale (if there are just two arguments) and with `stroke()`.

Challenge

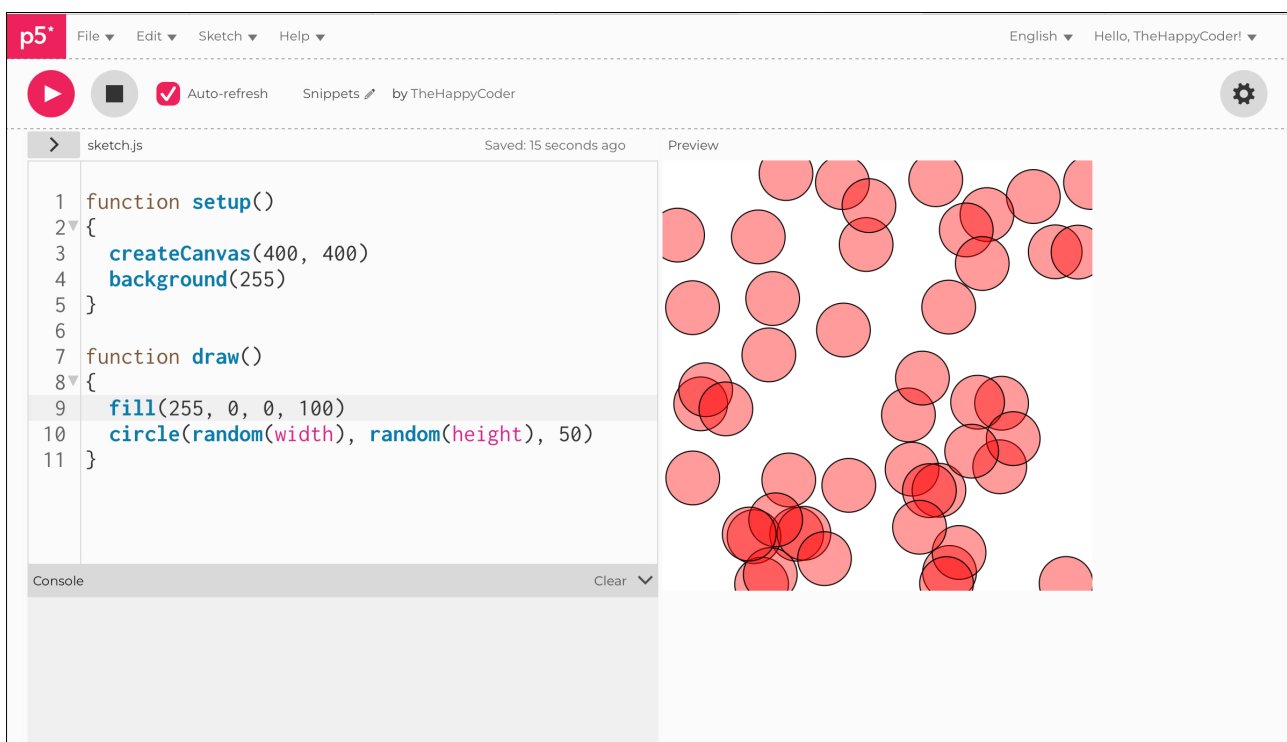
Try different values of alpha.

Code Explanation

```
fill(255, 0, 0, 100)
```

The fill() function now has four arguments, red, green, blue and alpha.

Figure A4.33





Sketch A4.34 text

! Starting a new sketch.

We can write text on the canvas using the `text()` function. The `text()` function has three arguments: the first is the text (or named variable), the second and third are the `x` and `y` co-ordinates. The origin of the text is taken at the top right-hand corner. To put the origin in the centre of the text, we use the `textAlign()` function. The text is what is called a string, and it has to have speech (single or double) marks around the text.

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

function draw()
{
  background(220)
  textAlign(CENTER, CENTER)
  textSize(64)
  text('Hello', 200, 200)
}
```


Notes

We can add colour with `stroke()` and `fill()`. You can also add new fonts, but you have to upload them.

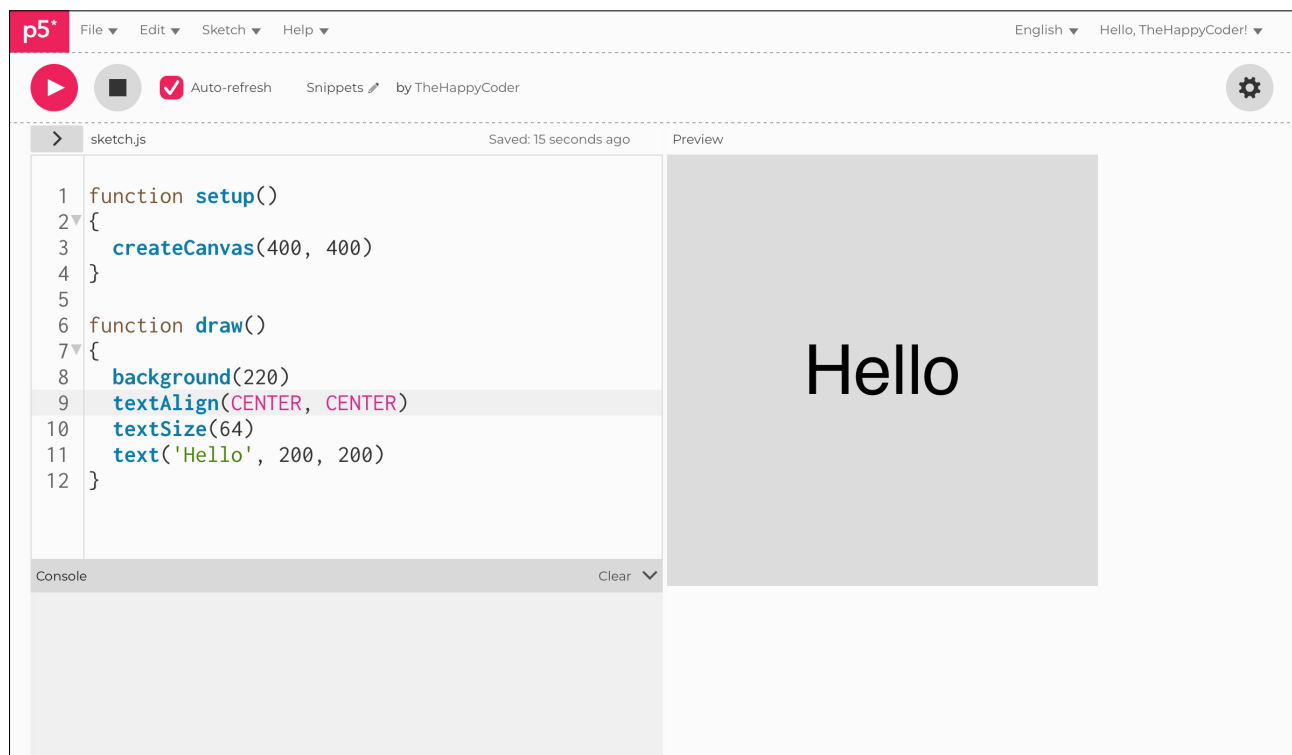
Challenge

Try commenting out the `textAlign()` line of code.

Code Explanation

<code>textAlign(CENTER, CENTER)</code>	This moves the origin from the top left hand corner of the text into the centre of the text
<code>textSize(64)</code>	You can increase the text size
<code>text('Hello', 200, 200)</code>	The text function has three arguments, the first is the text, the second and third are the co-ordinates

Figure A4.34





Sketch A4.35 variable text

Instead of text as a string, we can use the value of a variable and put that on the canvas as text. In this example, we will give the values of the **x** and **y** co-ordinates to the mouse pointer.

In version **2.x** we have now to include **int** before the mouse value otherwise it gives us a float of many decimal points.

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

function draw()
{
  background(220)
  textAlign(CENTER, CENTER)
  textSize(64)
  fill(200, 0, 0)
  text(int(mouseX), mouseX, mouseY - 32)
  fill(0, 0, 200)
  text(int(mouseY), mouseX, mouseY + 32)
}
```

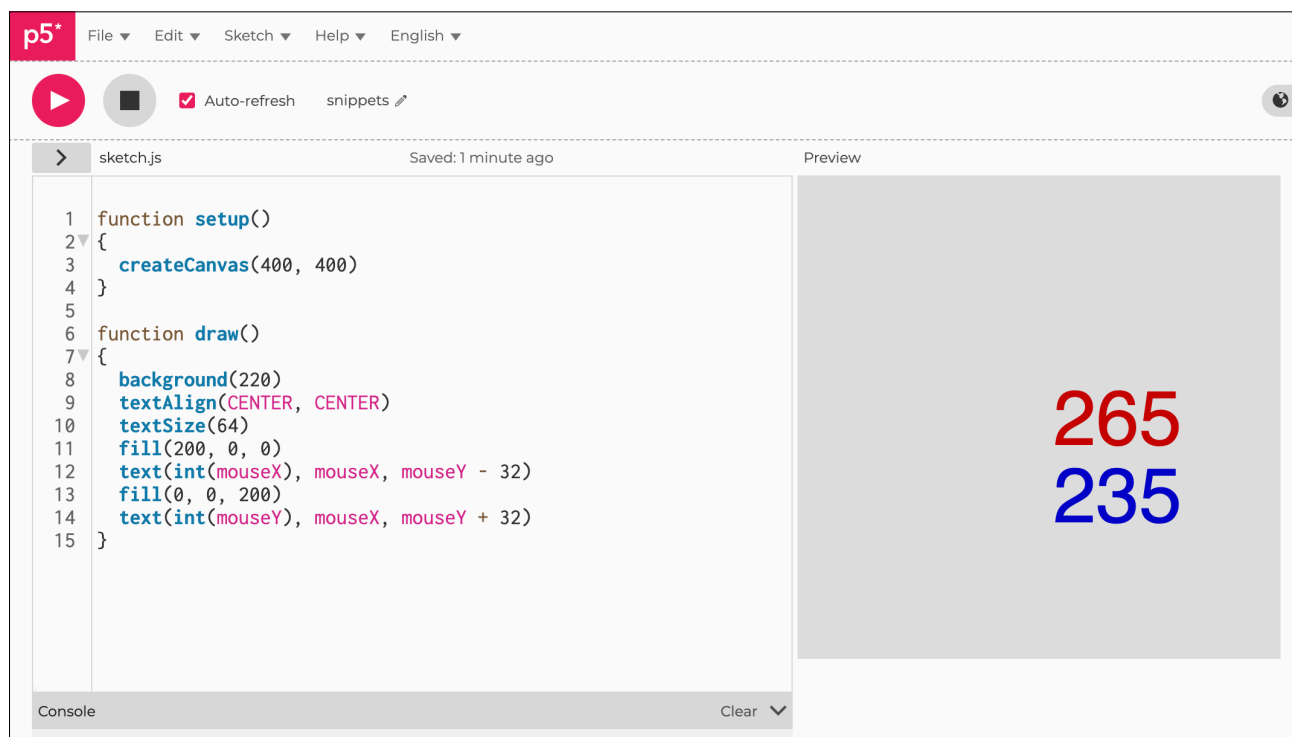
Notes

Added a splash of colour and spaced them out. It gives you the value of the variable in this case, the **x** position and the **y** position of the mouse point.

Code Explanation

<code>text(int(mouseX), mouseX, mouseY - 32)</code>	The mouse x position is displayed using mouseX
<code>text(int(mouseY), mouseX, mouseY + 32)</code>	The mouse y position is displayed using mouseY

Figure A4.35





Sketch A4.36 the AND gate

! Start a new sketch.

We are dividing the canvas into four quadrants. If the mouse is in the top left, the circle is filled red; if top right, then filled blue; bottom right, it is filled green; and the bottom left, the circle will be just white. This makes use of the **AND** logic, which uses the symbols **&&**.

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

function draw()
{
  background(220)
  line(width/2, 0, width/2, height)
  line(0, height/2, width, height/2)
  if(mouseX <= 200 && mouseY <= 200)
  {
    fill(200, 0, 0)
  }
  else if(mouseX >= 200 && mouseY <= 200)
  {
    fill(0, 0, 200)
  }
  else if(mouseX >= 200 && mouseY >= 200)
  {
    fill(0, 200, 0)
  }
  else
  {
    fill(255)
  }
  circle(200, 200, 100)
}
```

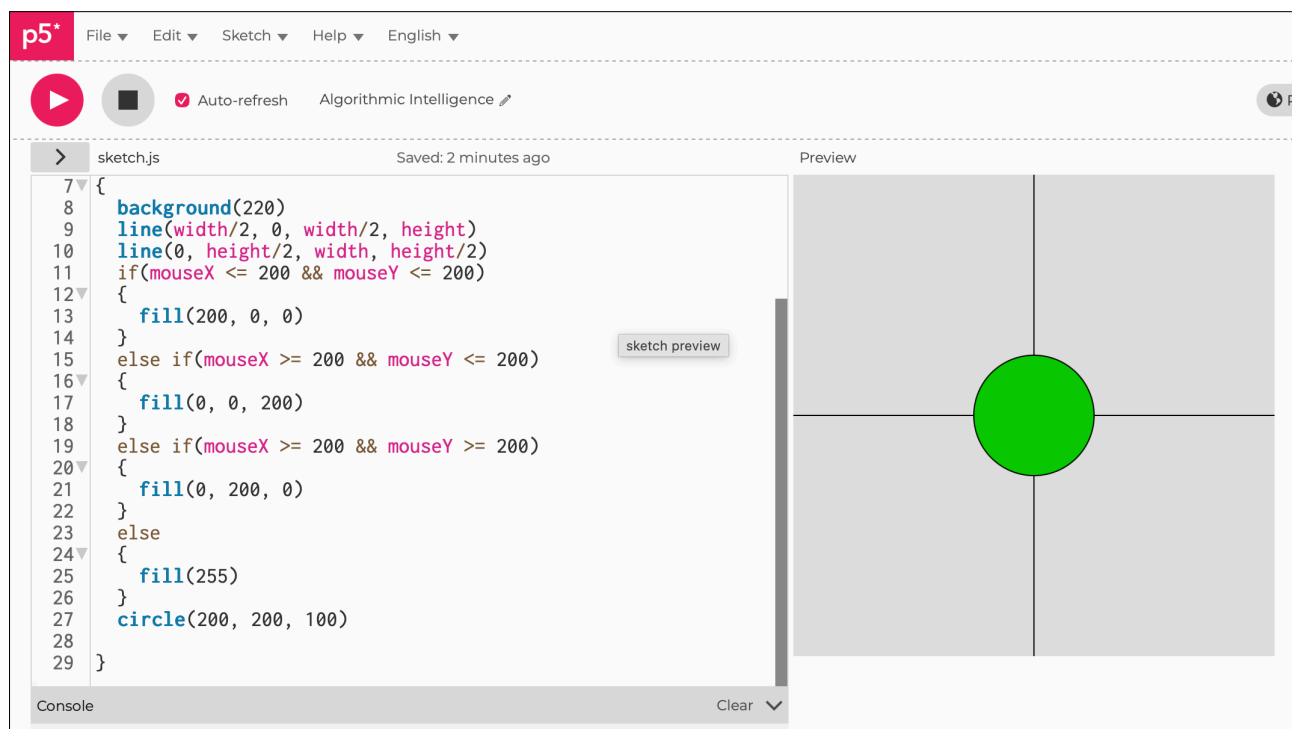
Notes

As you move the mouse around the circle in each quadrant, it changes the colour of the circle. This makes the simple **AND**, **OR**, and **NOT** logic very powerful, even though you only have two basic variables. In other examples, you may well have many more variables.

Code Explanation

<code>if(mouseX <= 200 && mouseY <= 200)</code>	In summary it is saying, if the x is less than (or equal to) 200 AND y is less than 200
<code>else if(mouseX >= 200 && mouseY <= 200)</code>	Using the else if() function. If x is greater 200 AND y is less than 200
<code>else if(mouseX >= 200 && mouseY >= 200)</code>	If x is greater than 200 AND y is greater than 200
<code>else</code>	If none of the others are true then do this...

Figure A4.36





Sketch A4.37 the OR gate

The **OR** gate is the opposite of the **AND** logic. In this one, we simply change one line of code from **AND** to **OR**, and the difference is obvious. Work through the logic of why that is the case. The **OR** symbol is two vertical lines (pipes) **||**.

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

function draw()
{
  background(220)
  line(width/2, 0, width/2, height)
  line(0, height/2, width, height/2)
  if(mouseX <= 200 || mouseY <= 200)
  {
    fill(200, 0, 0)
  }
  else if(mouseX >= 200 && mouseY <= 200)
  {
    fill(0, 0, 200)
  }
  else if(mouseX >= 200 && mouseY >= 200)
  {
    fill(0, 200, 0)
  }
  else
  {
    fill(255)
  }
  circle(200, 200, 100)
}
```

Notes

We get three quarters of the canvas red (top left, top right, and bottom left); the remainder is still green.

Challenges

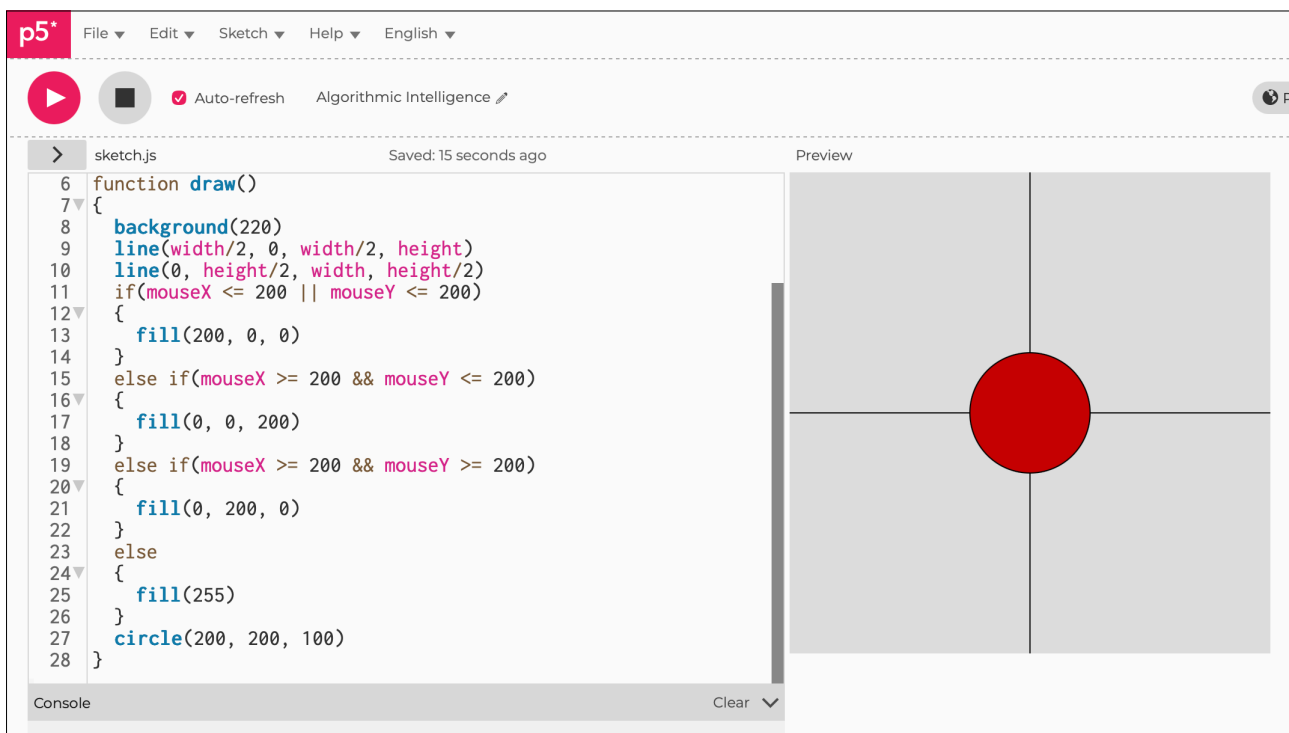
1. Try other variations of **AND** or **OR** logic.
2. Recode this sketch so that you colour different segments.
3. Try using the **NOT** logic symbol **!**.

Code Explanation

```
if(mouseX <= 200 || mouseY <= 200)
```

This contains two possible truths. Either x is less than 200 OR y is less than 200

Figure A4.37





Sketch A4.38 p5.Vector.sub() function

! Start a new sketch.

Here we are going to create two vector objects: one vector is the centre of the canvas, and the other vector is the position of the mouse point. We subtract the mouse point vector (**mouseX** and **mouseY**) from the centre of the canvas vector.

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

function draw()
{
  background(220)
  let v1 = createVector(mouseX, mouseY)
  let v2 = createVector(200, 200)
  let v3 = p5.Vector.sub(v1, v2)
  strokeWeight(5)
  stroke('red')
  line(v3.x, v3.y, v2.x, v2.y)
  stroke('blue')
  line(v1.x, v1.y, v2.x, v2.y)
}
```

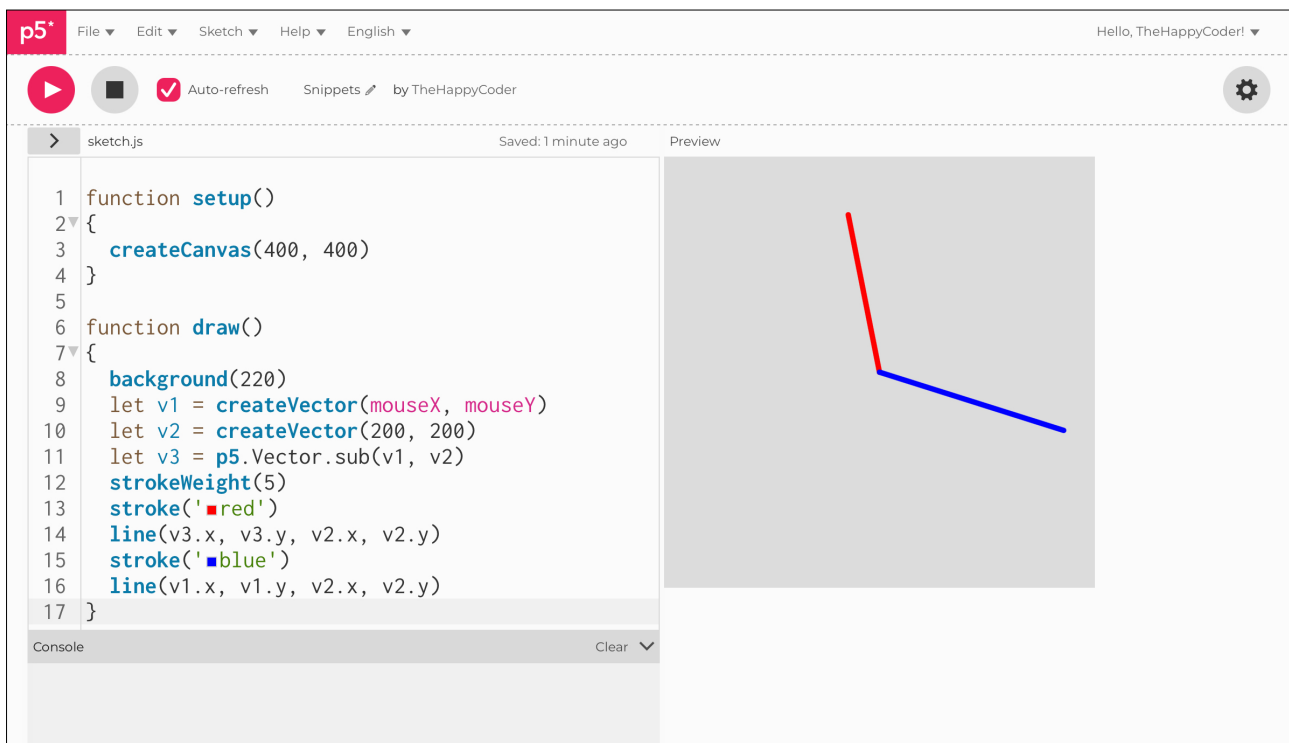

Notes

The red line is the subtraction of the other two vectors.

Code Explanation

<pre>let v3 = p5.Vector.sub(v1, v2)</pre>	This function allows us to subtract two vectors, return a third vector object
<pre>stroke('red')</pre>	As well as rgb colour values we can use names as a shortcut. There are lots of colours to choose from. Needs to have speech marks

Figure A4.38





Sketch A4.39 p5.Vector.add() function

The result may look the same, but now the **p1** vector is being added to the **p2** vector, making the **p3** vector (red) much bigger.

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

function draw()
{
  background(220)
  let v1 = createVector(mouseX, mouseY)
  let v2 = createVector(200, 200)
  let v3 = p5.Vector.add(v1, v2)
  strokeWeight(5)
  stroke('red')
  line(v3.x, v3.y, v2.x, v2.y)
  stroke('blue')
  line(v1.x, v1.y, v2.x, v2.y)
}
```

Notes

There are a whole raft of vector functions; they are too many to mention here.

Challenge

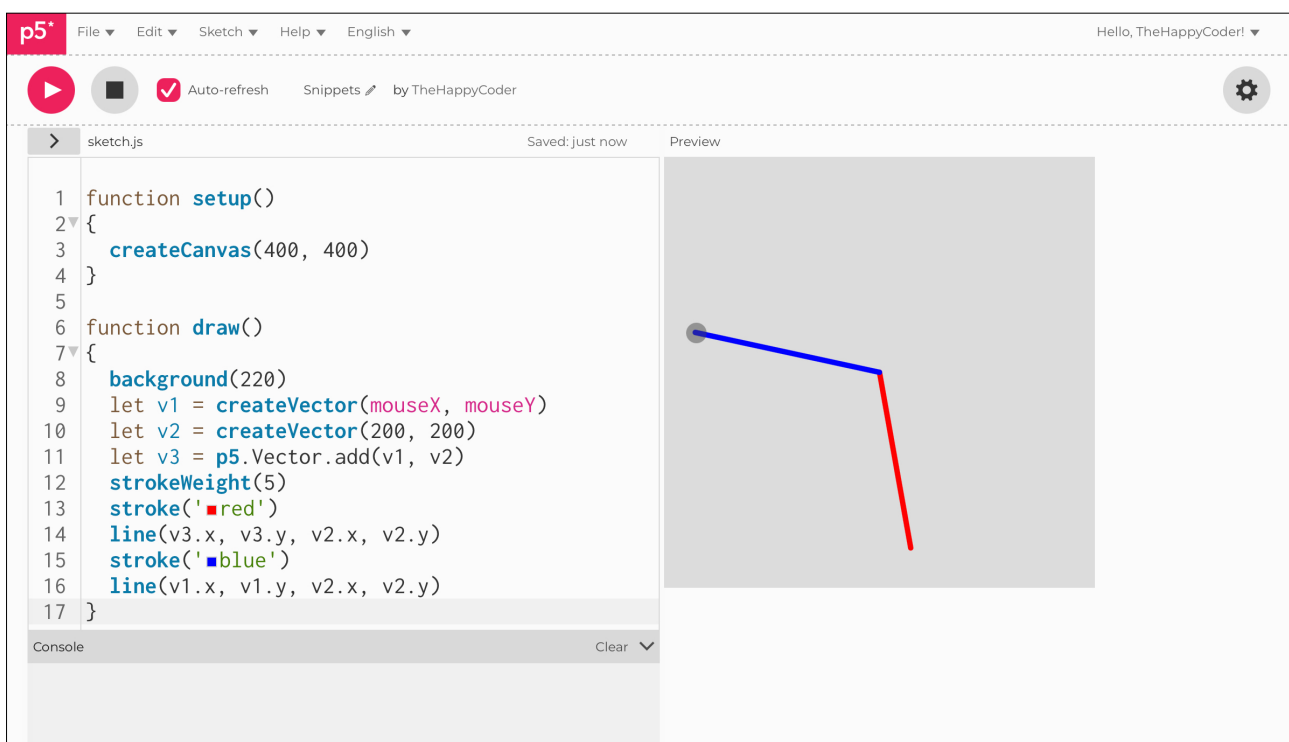
Try `.mult()` instead of `.add()`.

Code Explanation

```
let v3 = p5.Vector.add(v1, v2)
```

Adds two vectors returning a third vector

Figure A4.39





Sketch A4.40 a bit more mouse

! New sketch.

There are a lot of functions we can use with the mouse: its movement, the buttons, and the mouse wheel. Here are just a few to demonstrate functions relating to the mouse buttons. This is our starting sketch.

```
let value = 0

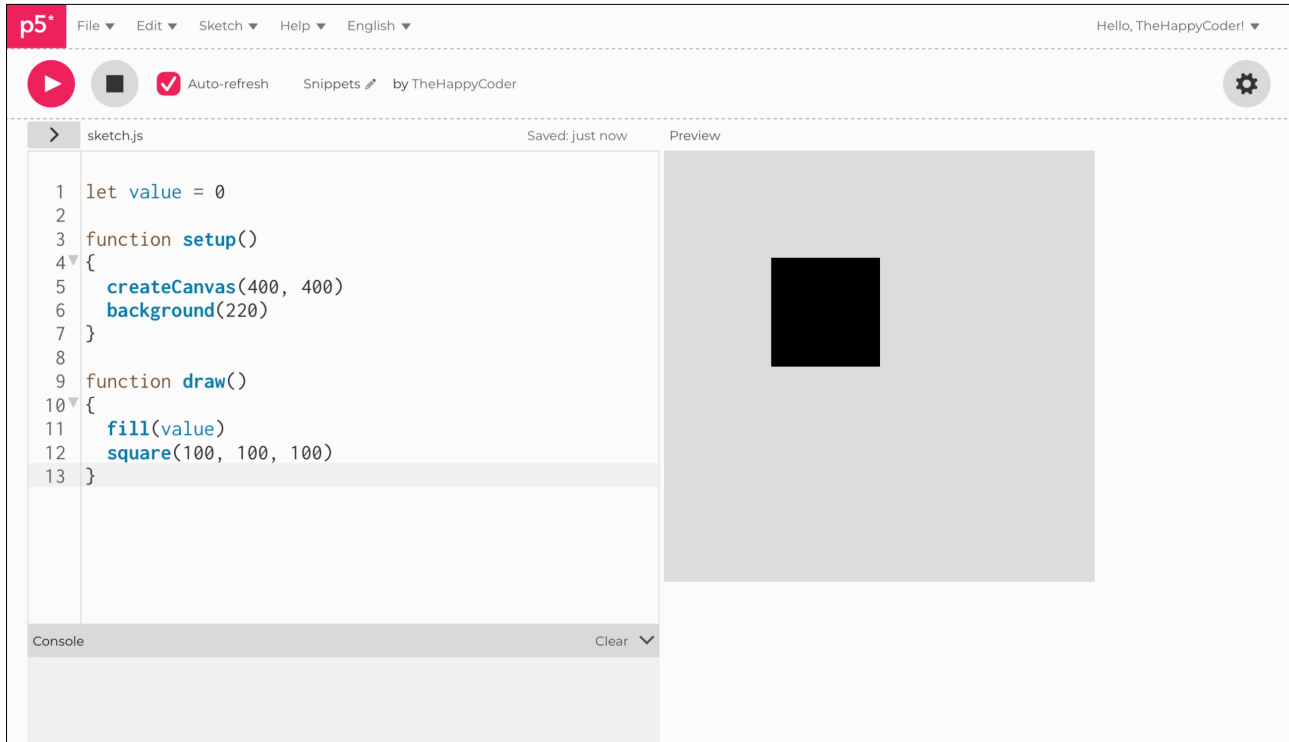
function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  fill(value)
  square(100, 100, 100)
}
```

Notes

We get a black square, nothing more, nothing less.

Figure A4.40





Sketch A4.41 mouseDragged() function

Click on the canvas and keep the button down as you move the mouse across the canvas. We can change the colour of the square as we click and drag the mouse across the canvas.

```
let value = 0

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

function draw()
{
  fill(value)
  square(100, 100, 100)
}

function mouseDragged()
{
  value += 2
  if (value > 255)
  {
    value = 0
  }
}
```

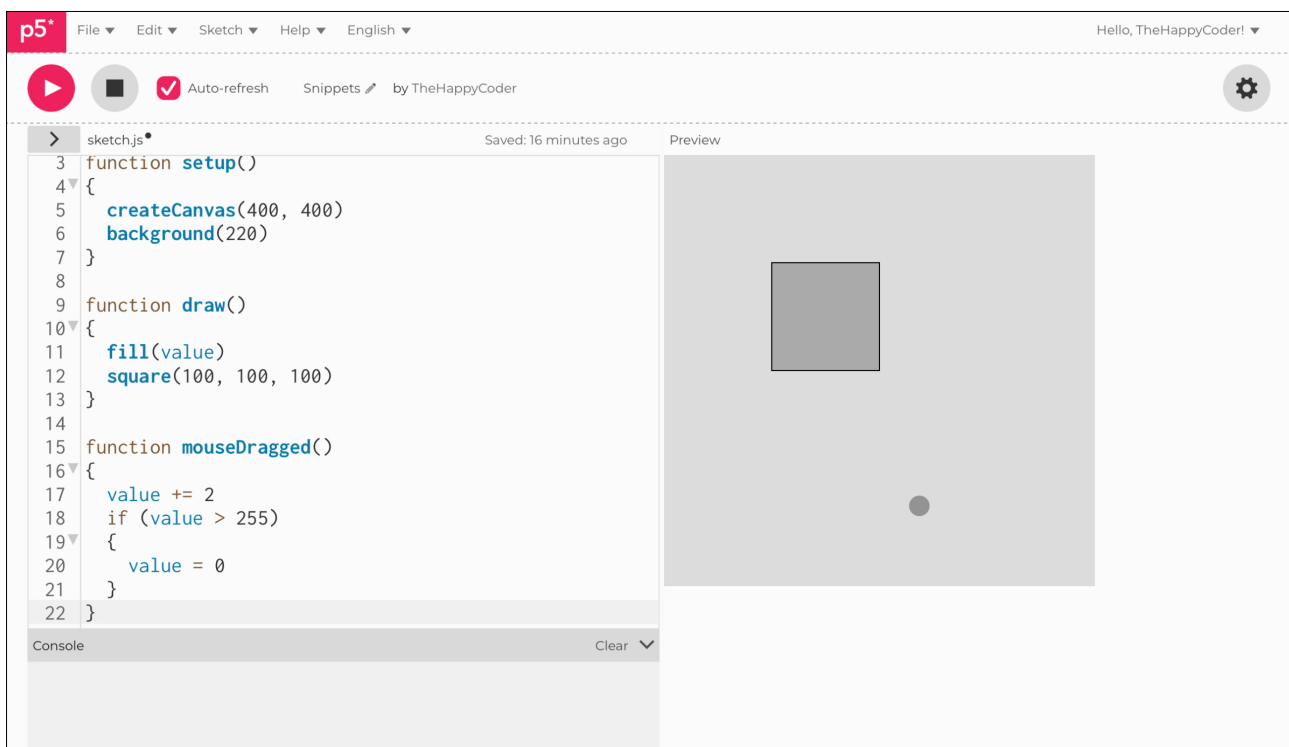
Notes

The greyness of the square changes as we drag the mouse.

Code Explanation

<code>function mouseDragged()</code>	Senses when the mouse has been click and moved at the same time.
<code>value += 2</code>	Adds 2 every time the mouse is moved

Figure A4.41





Sketch A4.42 mousePressed() function

This is a simple way to toggle using a mouse button.

```
let value = 0

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

function draw()
{
  fill(value)
  square(100, 100, 100)
}
```

```
function mousePressed()
{
  if (value === 0)
  {
    value = 255
  }
  else
  {
    value = 0
  }
}
```


Notes

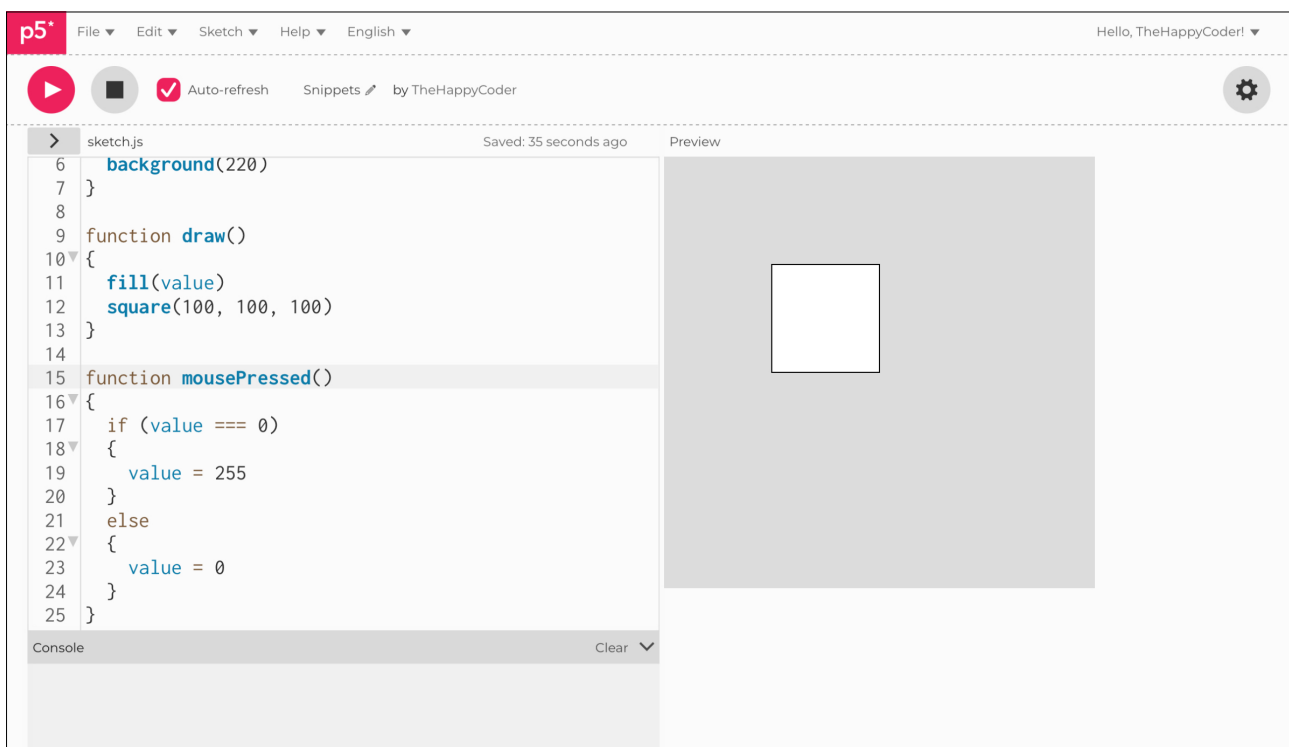
Every time you click on the canvas, the square toggles between white and then black.

Code Explanation

`function mousePressed()`

Senses when the mouse button has been pressed

Figure A4.42





Sketch A4.43 mouseReleased() function

Does the same but only acts when the button is released.

```
let value = 0

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

function draw()
{
  fill(value)
  square(100, 100, 100)
}

function mouseReleased()
{
  if (value === 0)
  {
    value = 255
  }
  else
  {
    value = 0
  }
}
```

Notes

A slight difference, but it can be useful when drawing a line that finishes when you release the mouse button.

Code Explanation

```
function mouseReleased()
```

Senses when the mouse button has been released



Final note on this section

If you worked through all this, well done. Now you have a basic but sound understanding of the workings of p5.js. Next, we can start the fun bit of machine learning. Enjoy.

Algorithmic Intelligence Module A Unit #5 linear regression



Module A Unit #5: linear regression with ml5.js

In simple terms, we are going to use the data points for a straight line. We will use this dataset to train the model. It will then predict the relationship between x and y of a simple straight line (linear). We will be drawing the prediction so we have a visual result to see how well it has performed.

This may seem like a rather simple exercise, but it will serve as a demonstration of how a machine learning neural network will learn, demonstrating its usefulness and also its limitations. To make it a bit more realistic and challenging, we are going to introduce a bit of variance; what this means is adding some randomness to the data.

This is because real-world data is very rarely so neat and tidy. Machine learning (AI) comes into its own when patterns aren't necessarily so obvious. This becomes even more so when there is a lot of data, with many variables, and the usual strategies or algorithms would struggle.

Here we have two variables. The x and the y co-ordinates. The equation for a line is $y = mx + b$. The m is the slope of the line, and the b is where it intercepts the y -axis. We artificially create the data and add in the variance. This is **synthetic data**; it isn't real data seen in the world, but it will serve the purpose.

We train the model on this data and get the model to predict a line through the data. Its prediction is the generalisation (or best guess) of the data. The beauty of this exercise is that we can play around with a number of hyperparameters, such as the number of hidden layers, the number of nodes (in each layer), the activation function, the learning rate, the optimiser, the batch size, and of course, the epochs.



Backend Stuff

To speed up, or even make it work at all, depending on your machine or browser, you may need to have one of the following lines of code:

```
ml5.setBackend("webgl")  
ml5.setBackend("cpu")  
ml5.setBackend("webgpu")  
ml5.setBackend("gpu")
```

By default, we are going to add **webgl** to make sure that it works across all browsers. You can add **cpu** instead; it might work very much faster if you do, but, however, it might affect the programme depending on your machine and browser. Other alternatives are **gpu** or **webgpu**. Just experiment at a later date to get a better performance.



Introducing ml5.js

In the unit [a gentle overview of ml5.js](#) I showed you how to add the line of code into the index.html file. This is important; otherwise, there is no neural network to train the model. So, if you have not read through that section, it is important that you do so, or you can duplicate the template I have provided on my website by clicking on the button, and then come back to this unit.



The index.html file

The line of code needs to be added to the `index.html` file before you do anything else. If you swap to version 2 you will have slightly different `index.html`, it will show you the `p5.js` library as 2.1.x but there will not be any sound library. As far as I know it might be incorporated into the new version. I recommend that you use the latest version of `ml5.js`.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch A5.1 starting sketch

This is our starting sketch.

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

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



Sketch A5.2 drawing the points on a line

We create a `for()` loop to draw each data point as a blue circle. We space them out by 10 pixels.

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

function draw()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    fill(0, 0, 255)
    circle(i, height - i, 5)
  }
}
```

Notes

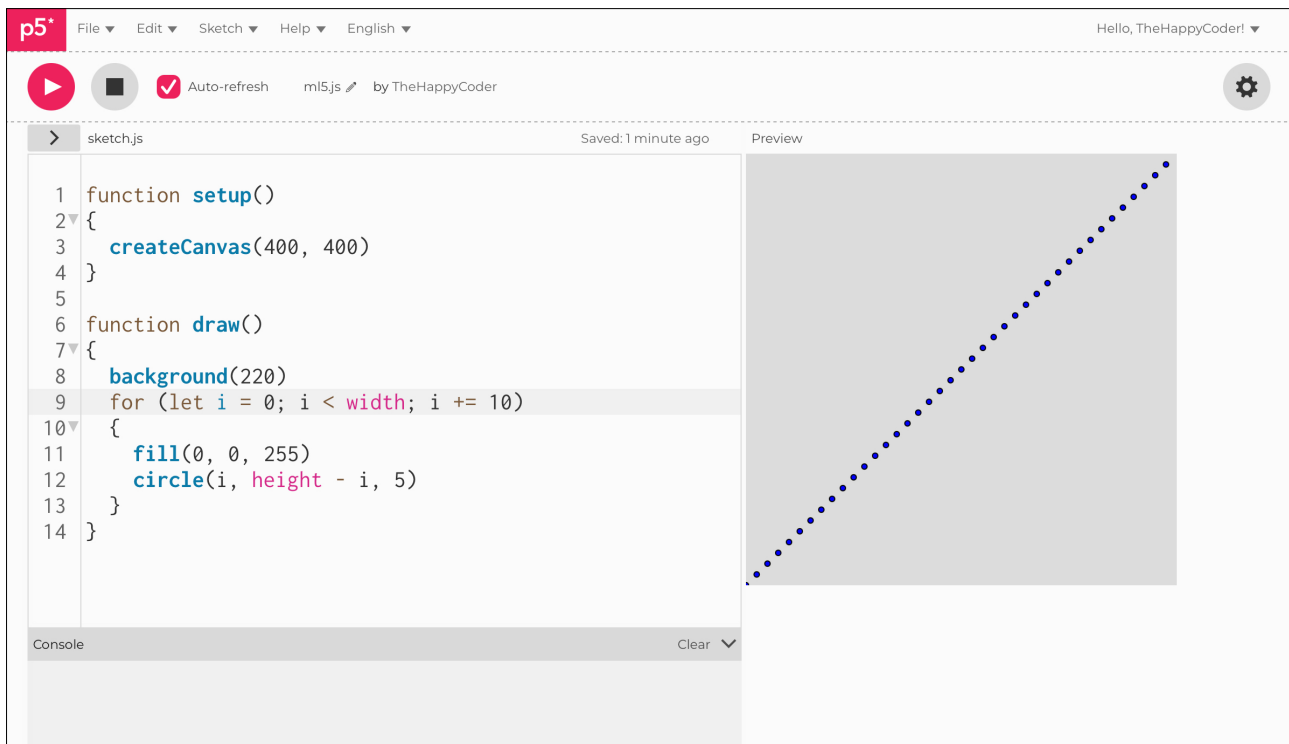
For each value of i , this will be the x co-ordinate of the circle; the $height-i$ is the y co-ordinate. Simply, this is a straight line where $y = x$, and the slope m is effectively 1 and the intercept b is 0 .

Code Explanation

$i += 10$

Adding 10 each iteration of the for loop, this increases the value of i by 10

Figure A5.2





Sketch A5.2 initial sketch with data points

We want to draw 30 circles, one on top of the other. Hence, we introduce a constant (**const**) variable called **number**. In the next part, we will spread those 30 circles randomly around the line with some variance (**spread**).

```
const number = 30

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

function draw()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      fill(0, 0, 255)
      circle(i, height - i, 5)
    }
  }
}
```



Notes

Now we are drawing 30 circles for each data point. We use **j** as the variable for the nested loop.



Sketch A5.3 sketch revised

Now we are going to spread the data points using the variable **spread**. You will notice that it draws the circles repeatedly in random points.

```
const number = 30
const spread = 30

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

function draw()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      fill(0, 0, 255)
      circle(i + random(-spread, spread), height - i + random(-spread, spread), 5)
    }
  }
}
```

Notes

We don't want the circles being drawn continuously; we will address the issue in the next sketch.

Code Explanation

<code>random(-spread, spread)</code>	Gives us a random value between -30 and +30
<code>i + random(-spread, spread)</code>	This means that for every i (x) value we move it above or below the real value.



Sketch A5.4 revising the data

To stop it looping through, we change the name of the `draw()` function to `trainingData()`. This means it is no longer a continuous loop. We move `background(220)` into the `setup()` function and call the `trainingData()` function there. Now it draws the data points just once every time you run the sketch, but not continually as before.

```
const number = 30
const spread = 30

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

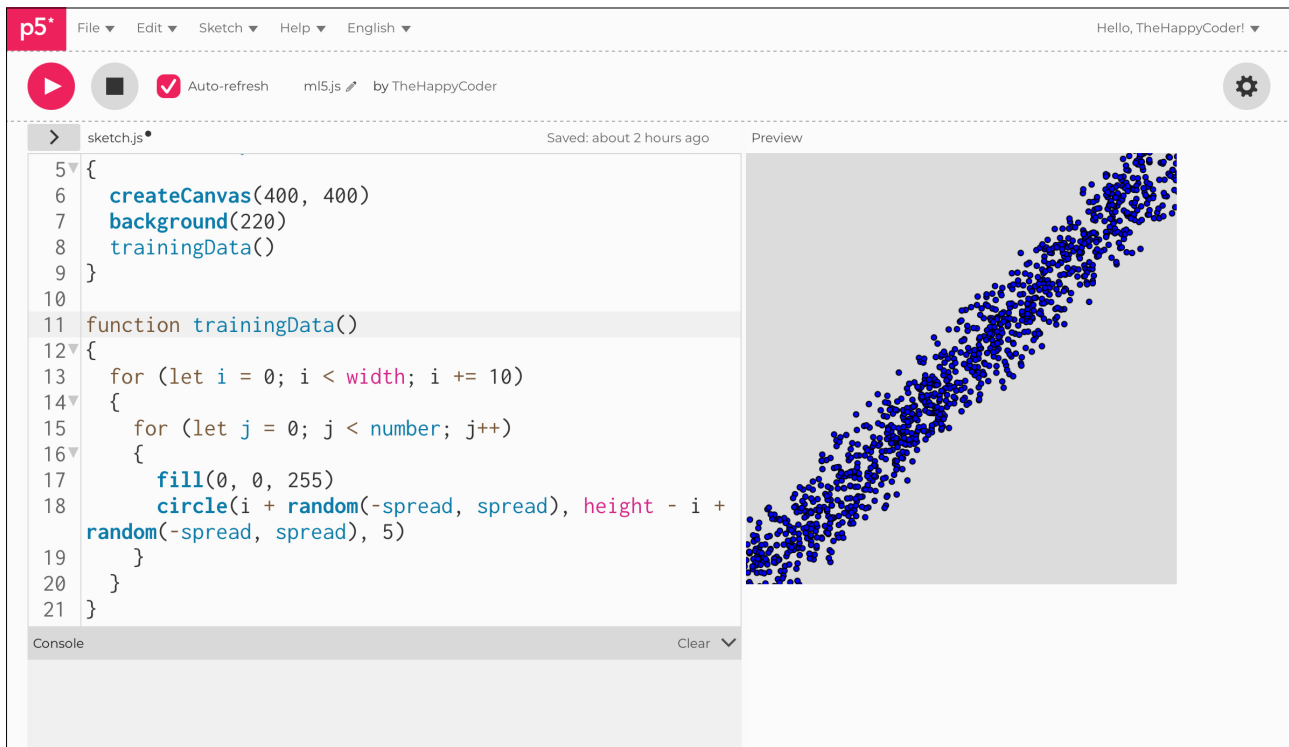
function trainingData()
{
  // background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      fill(0, 0, 255)
      circle(i + random(-spread, spread), height - i + random(-spread,
spread), 5)
    }
  }
}
```

Notes

This may sound complicated, but it is just a little bit of refactoring. The training data is called just once in `setup()`. This will become evident later as we don't want to be creating new data as we train.

! Remove the `background(220)` from the `trainingData()` function.

Figure A5.4





Sketch A5.5 adding the model

We are going to call our neural network **nn**. This may not sound very creative, but it makes sense. We then call the **ml5.js** library, and specifically, we are using the neural network. This is because we are building our own model rather than a pre-trained one.

```
let nn

const number = 30
const spread = 30

function setup()
{
  createCanvas(400, 400)
  background(220)
  nn = ml5.neuralNetwork()
  trainingData()
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      fill(0, 0, 255)
      circle(i + random(-spread, spread), height - i + random(-spread, spread), 5)
    }
  }
}
```

Notes

Nothing is meant to happen; all we have done is call the neural network library **ml5.js**. This is why we need it in the **index.html** file.

Code Explanation

let nn	Naming the neural network model
nn = ml5.neuralNetwork()	Calling or assigning the ml5 neural network to the nn model name



Sketch A5.5 adding the model

We have initialised the model and now we have to give it some **options**. This is information the model needs. First off, we have to decide what kind of task we want it to do; in this case, it is a **regression** task because we want it to output a value **y** that varies based on the **x** input.

The second option is something called **debug**: what this does is draw the loss as a graph/chart so we can see the loss values visually. You don't have to use this, but it is handy to see what progress the model is making as it is training. To see the graph, you set **debug: true**; alternatively, set it to **false** if you want to hide it.

We have the **options** object as a **const** so that we don't change any of the parameters. We then call the options inside ml5. As mentioned in the introduction, we have added **webgl** to the backend to help with the training process.

```
let nn
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: true
}

function setup()
{
  createCanvas(400, 400)
  background(220)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      fill(0, 0, 255)
```

```
        circle(i + random(-spread, spread), height - i + random(-spread,
spread), 5)
    }
}
}
```

Notes

Adding this code won't make any difference yet. If you get any errors, make sure you have typed everything in correctly and also that you have the ml5.js line of code in the index.html correctly written.

Challenge

If your machine will accept it, then try: `ml5.setBackend("cpu")` for faster training.

Code Explanation

<code>const options</code>	This creates an object of options we can use with ml5.js
<code>ml5.setBackend("webgl")</code>	Using the rendering that is most appropriate
<code>nn = ml5.neuralNetwork(options)</code>	Adding the options to the neural network



Sketch A5.6 adding the data

What the model needs is the data; therefore, we need to change a number of things. We need to collect all those data points into the `data` variable. So the variable `data` now cycles through all the `x` and `y` values. The data is now an array of those `x` and `y` values. We then draw the circle to the `data.x`, which is `data[0]`, and `data.y` is `data[1]`.

```
let nn
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: true
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), height - i + floor(random(-spread, spread))]
      fill(0, 0, 255)
      circle(data[0], data[1], 5)
    }
  }
}
```

Notes

We still draw the data points as before, but the code is looking quite different now. Try to follow what the code is doing.

Code Explanation

<code>i + floor(random(-spread, spread))</code>	We use floor so that we get whole numbers (integers) not floats.
<code>circle(data[0], data[1], 5)</code>	The circle data[0] is the x value, the data[1] the y value.



Sketch A5.6 adding the data

We are going to add this data to the neural network. To do this, we use the `addData()` function. After the data has been added, we need to normalise the data with the `normalizeData()` function. This transforms the data to between `-1` and `+1`. The `console.log()` will let us know when that has been completed.

```
let nn
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: true
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  console.log('done')
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), height - i + floor(random(-spread, spread))]
      fill(0, 0, 255)
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[1]])
    }
  }
}
```

```
}
```

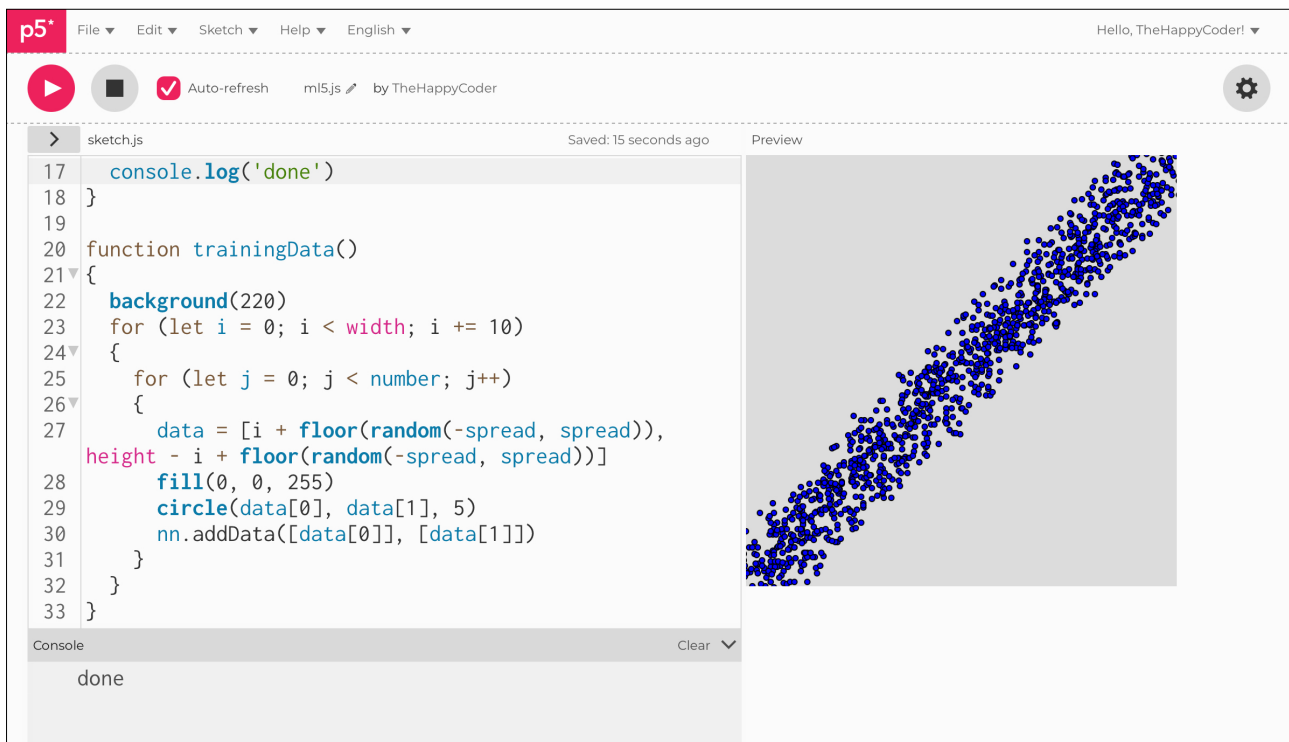
Notes

We have passed the data into the model; next, we need to train the model on that data.

Code Explanation

<code>nn.normalizeData()</code>	This normalises the data
<code>console.log('done')</code>	Let us know when it is all done
<code>nn.addData([data[0]], [data[1]])</code>	This adds the data to the nn model

Figure A5.6





Sketch A5.7 training on the data

To train the model, we use the `train()` function. We are going to train the model on the data we have normalised. We add a callback function in the `train()` function called `finishedTraining()` so that when the training is complete, it will let us know. Here, the loss graph kicks into action, and you will notice that it stops training after about ten epochs.

```
let nn
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: true
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  nn.train(finishedTraining)
  console.log('done')
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), height - i + floor(random(-spread, spread))]
      fill(0, 0, 255)
      noStroke()
      circle(data[0], data[1], 5)
```



```
    nn.addData([data[0]], [data[1]])  
  }  
}  
}
```

```
function finishedTraining()  
{  
  console.log('finished')  
}
```

Notes

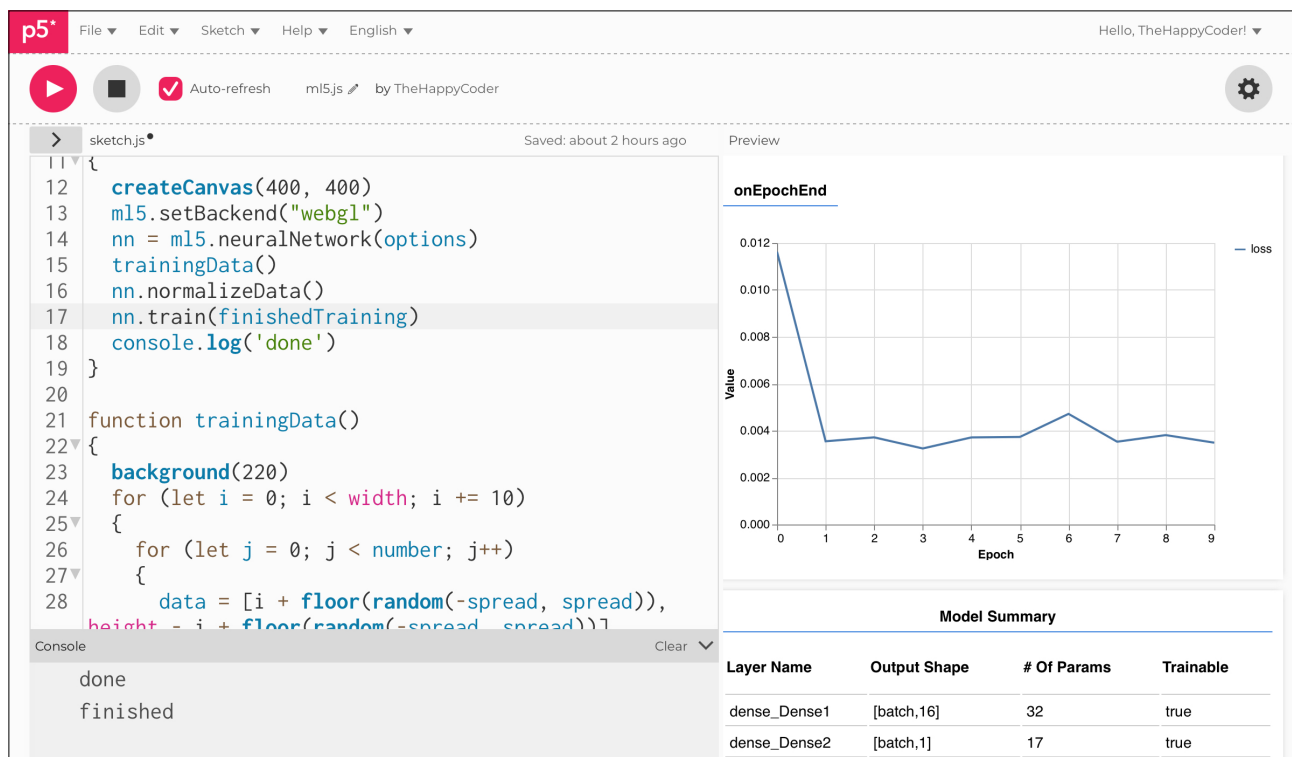
When the loss graph has finished, you can remove it by clicking on the **Hide** button, so that you can see the canvas (if your screen is small). In the console, we get **done** and **finished** after the training is completed. The default settings are for 10 epochs, and there are 16 nodes (neurons) in the hidden layer. It automatically works out the number of nodes for the input and output. Notice it makes reference to Dense 1 and Dense 2. This is the hidden layer and the output layer. The input layer is not considered a layer. Dense means fully connected.

Code Explanation

```
nn.train(finishedTraining)
```

Trains the model and when finished calls the function `finishedTraining()`

Figure A5.7





Sketch A5.8 predicting the result

We need to predict the result after the training is complete. We do this using the `predict()` function. This will be very telling as it is going to draw an approximation of the data. In theory, it should be a straight line where $y = x$ drawn from the bottom left corner to the top right corner on the canvas, but we will see.

! Remove all the `console.log()`s, they were there just to check it worked. We will set `debug:` to `false` so you can see the line being drawn (better still, just remove it completely).

In the `finishedTraining()` function, we use the `counter` as an array; this is the `x` value. The callback function `gotResults()` will draw the predicted result for `y` for each `x` input. We only want `400` because that is the number of pixels in the width of the canvas.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: false
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  nn.train(finishedTraining)
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
```

```

    for (let j = 0; j < number; j++)
    {
        data = [i + floor(random(-spread, spread)), height - i + floor(random(-spread, spread))]
        fill(0, 0, 255)
        noStroke()
        circle(data[0], data[1], 5)
        nn.addData([data[0]], [data[1]])
    }
}

function finishedTraining()
{
    if (counter < 400)
    {
        nn.predict([counter], gotResults)
    }
}

```

Notes

If you run this, you might get an error message because we haven't created the `gotResults()` function just yet. We will need to advance the counter in the `gotResults()` function by one on each iteration (see next sketch).

Code Explanation

```
nn.predict([counter], gotResults)
```

This takes the x value (counter), predicts the result and passes to the callback function `gotResults()`



Sketch A5.9 the callback function

Next, we add the `gotResults()` callback function. The prediction is the first element in the array, which will be a value of `y` for a particular input of `x` (from the `counter`). We draw a point at that value and move onto the next. We add one to the `counter` and return to the `finishedTraining()` function for the next prediction.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: false
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  nn.train(finishedTraining)
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), height - i + floor(random(-spread, spread))]
      fill(0, 0, 255)
      noStroke()
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[1]])
    }
  }
}
```

```
    }  
  }  
}  
  
function finishedTraining()  
{  
  if (counter < 400)  
  {  
    nn.predict([counter], gotResults)  
  }  
}
```

```
function gotResults(results)  
{  
  let prediction = results[0]  
  let x = counter  
  let y = prediction.value  
  stroke(255, 0, 0)  
  strokeWeight(5)  
  point(x, y)  
  counter++  
  finishedTraining()  
}
```

Notes

The `results` is an argument that is an array of objects, actually one object at a time, for each `x` (`counter`) value. What you may get is a wonky line at the start and finish. This is most likely because we have some negative values when we create the variance of the data points.

Challenges

1. Use `console.log(results)` to see what it is returning (see second image below)
2. We could fill an array with values and work through each element one at a time

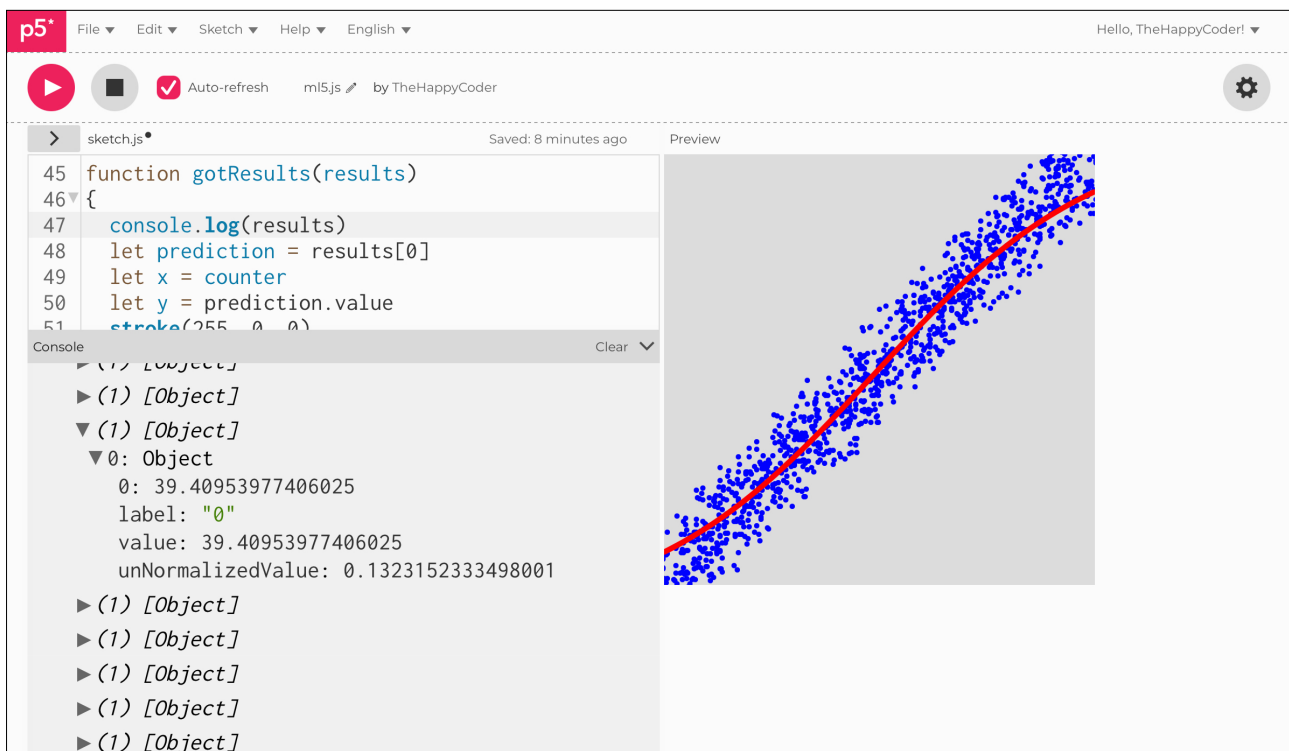
Code Explanation

<code>let prediction = results[0]</code>	The prediction variable takes all the data in the first object in the results
<code>let x = counter</code>	Gives the counter value to the x co-ordinate of the line
<code>let y = prediction.value</code>	This takes the value part of the object element. This is the actual y value
<code>counter++</code>	Adding one to the counter
<code>finishedTraining()</code>	Then finally return to the <code>finishedTraining()</code> function to get the next value based on the next counter (x) value

Figure A5.9a



Figure A5.9b: with `console.log(results)` and the `value` is the actual predicted value for the canvas





Sketch A5.10 batch size

The **batch size** is how much data we send through in one go. We could send it through one at a time or all of it in one go. A large **batch size** is usually faster but potentially less accurate. Whereas a smaller **batch size** takes longer but may give overall better results. We are going to give the `train()` function some options, which is where we specify the **batch size**. Our dataset is around **2400** data points, but first we will try a **batch size** of **1** and see what happens.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: true
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 1
  }
  nn.train(trainingOptions, finishedTraining)
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
```

```

    data = [i + floor(random(-spread, spread)), height - i + floor(random(-
spread, spread))]
    fill(0, 0, 255)
    noStroke()
    circle(data[0], data[1], 5)
    nn.addData([data[0]], [data[1]])
  }
}

function finishedTraining()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

function gotResults(results)
{
  let prediction = results[0]
  let x = counter
  let y = prediction.value
  stroke(255, 0, 0)
  strokeWeight(5)
  point(x, y)
  counter++
  finishedTraining()
}

```

Notes

The results speak for themselves. We need to consider a larger **batchSize** than 1. We could go to the other extreme (all the dataset) and see what happens then. To be fair when you repeat you may get relatively good results. Mostly it is dependent on what the starting (random) weights are.

Challenge

Try other values of batch size

Code Explanation

<code>const trainingOptions = {</code>	Create a permanent object for our training options
<code> batchSize: 1</code>	Specify the batch size
<code>nn.train(trainingOptions, finishedTraining)</code>	Add the training options to the train() function

Figure A5.10a Our loss chart looks woeful

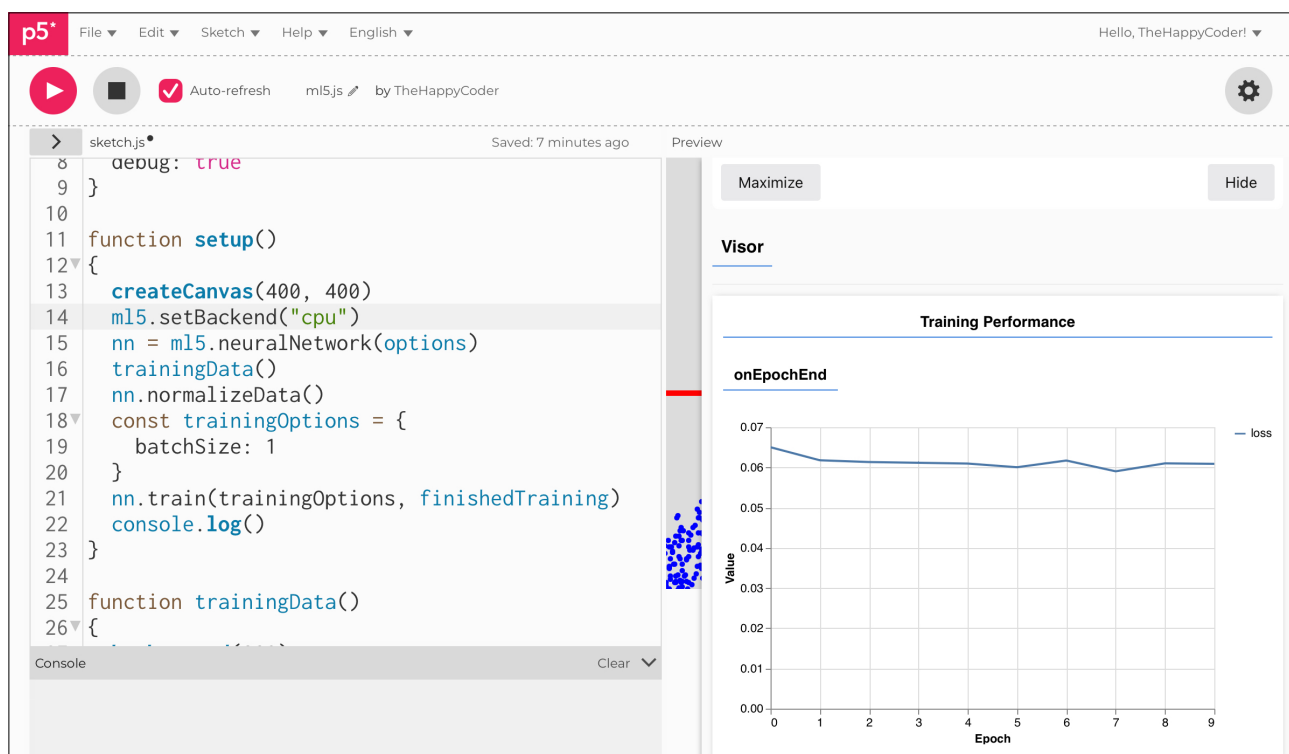


Figure A5.10b the results look woeful also





Sketch A5.11 increasing the batch size

Batch sizes are often in the order 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and so on. You can have any batch size you want, though. You often notice the difference instantly.

! Keep this code for the next unit

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: true
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512
  }
  nn.train(trainingOptions, finishedTraining)
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), height - i + floor(random(-spread, spread))]
      fill(0, 0, 255)
```

```

        noStroke()
        circle(data[0], data[1], 5)
        nn.addData([data[0]], [data[1]])
    }
}

function finishedTraining()
{
    if (counter < 400)
    {
        nn.predict([counter], gotResults)
    }
}

function gotResults(results)
{
    let prediction = results[0]
    let x = counter
    let y = prediction.value
    stroke(255, 0, 0)
    strokeWeight(5)
    point(x, y)
    counter++
    finishedTraining()
}

```

Notes

We get a better loss curve and better results, still a bit skewed at the ends. The results are far more consistent.

! Remember to keep this code for: **module A unit #6 sine wave regression**

Challenge

Try other batch sizes

Figure A5.11a loss much better

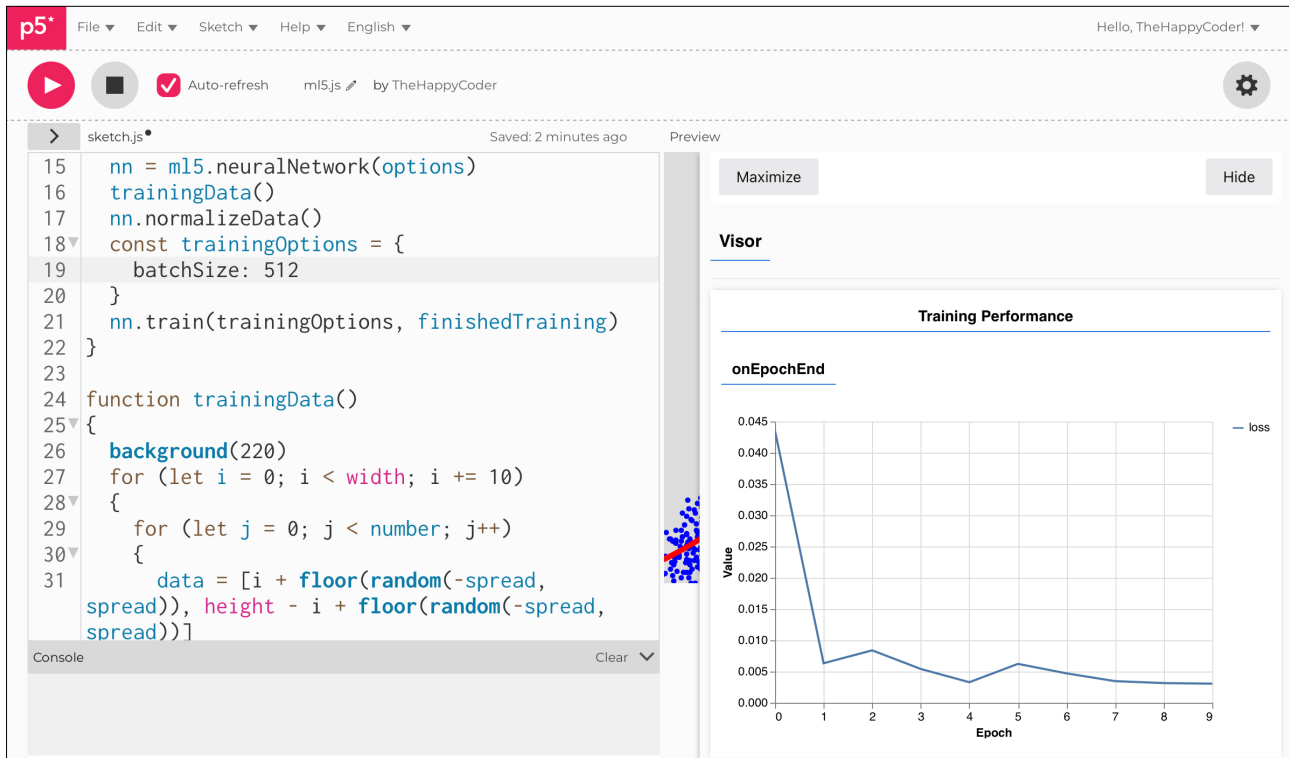
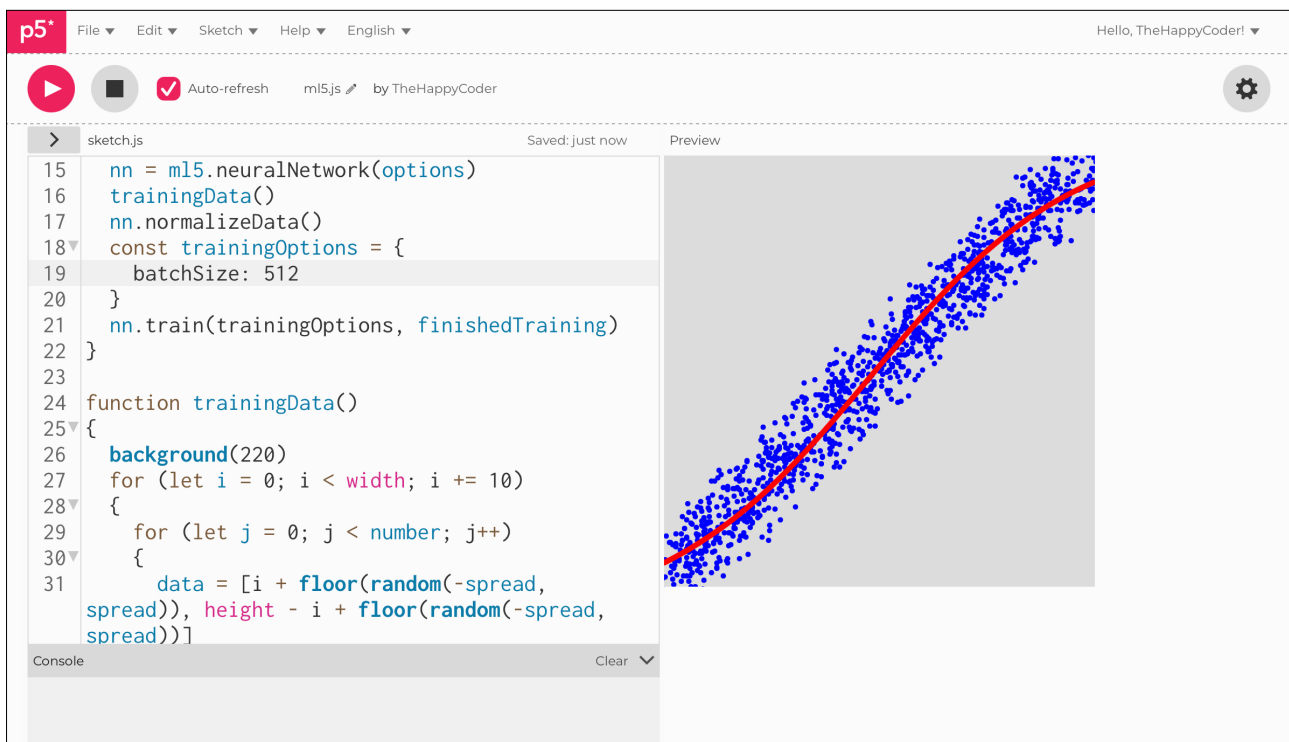


Figure A5.11b and better results





Final note on this section

That was as simple as it gets. However, we are dealing with very few data points, the model has to see a pattern even if the pattern is very obvious and very simple. Adding in all the variance makes its job harder. Changing the batch size can make a huge difference in more complex and larger datasets. In this instance we are simply exploring the options.

Next up we will push the boundaries a bit more by giving the model a greater challenge, a sine wave.

Algorithmic Intelligence Module A Unit #6 sine wave regression



Module A Unit #6: sine wave regression with ml5.js

In the previous unit, we trained a model to predict a straight line and had a peek at the batch size hyperparameter. This time, we are going to make things a bit more challenging. We are going to upgrade from a straight line to a sine wave.

This will give us a chance to look at some more hyperparameters and their impact on the training of the model. We will use them to try to make better predictions of a sine wave.

The hyperparameters we will be looking at will be:

- Number of hidden layers
- Number of nodes in each layer
- Activation functions
- Epochs
- Learning Rate



The index.html

Make sure that you have the `ml5.js` line of code in your `index.html` file. I have assumed you are going to use version `2.x`, although `1.11.13` should still work if that is the default you want to use.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch A6.1 sine wave sketch

We are starting with the sketch from linear regression. Then we'll modify the data because instead of a line, we want a sine wave. In p5.js, the default angle units are radians, but we are going to work in degrees, which is a little more intuitive. To make this change, we use the function `angleMode()` in `setup()`. Everything else in the sketch remains the same.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: true
}

function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512
  }
  nn.train(trainingOptions, finishedTraining)
}

function trainingData()
{
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
```

```

    data = [i + floor(random(-spread, spread)), 200 + (50 * sin(i) +
floor(random(-spread, spread)))]

    fill(0, 0, 255)
    noStroke()
    circle(data[0], data[1], 5)
    nn.addData([data[0]], [data[1]])
  }
}

function finishedTraining()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

function gotResults(results)
{
  let prediction = results[0]
  let x = counter
  let y = prediction.value
  stroke(255, 0, 0)
  strokeWeight(5)
  point(x, y)
  counter++
  finishedTraining()
}

```

Notes

For every **x** input, we get the sine of that input as our **y** output. We multiply by **50** to increase the amplitude (otherwise a very flat sine wave) and move it by **200** pixels so we have it in the centre of the canvas, plus the usual variance, **random(spread)**. The results aren't as bad as you might expect, as it uses default settings for most of the hyperparameters except for the batch size.

Code Explanation

<code>angleMode(DEGREES)</code>	This changes the default to degrees (notice all capitals)
<code>sin(i)</code>	Returns the sine of i

Figure A6.1a loss chart

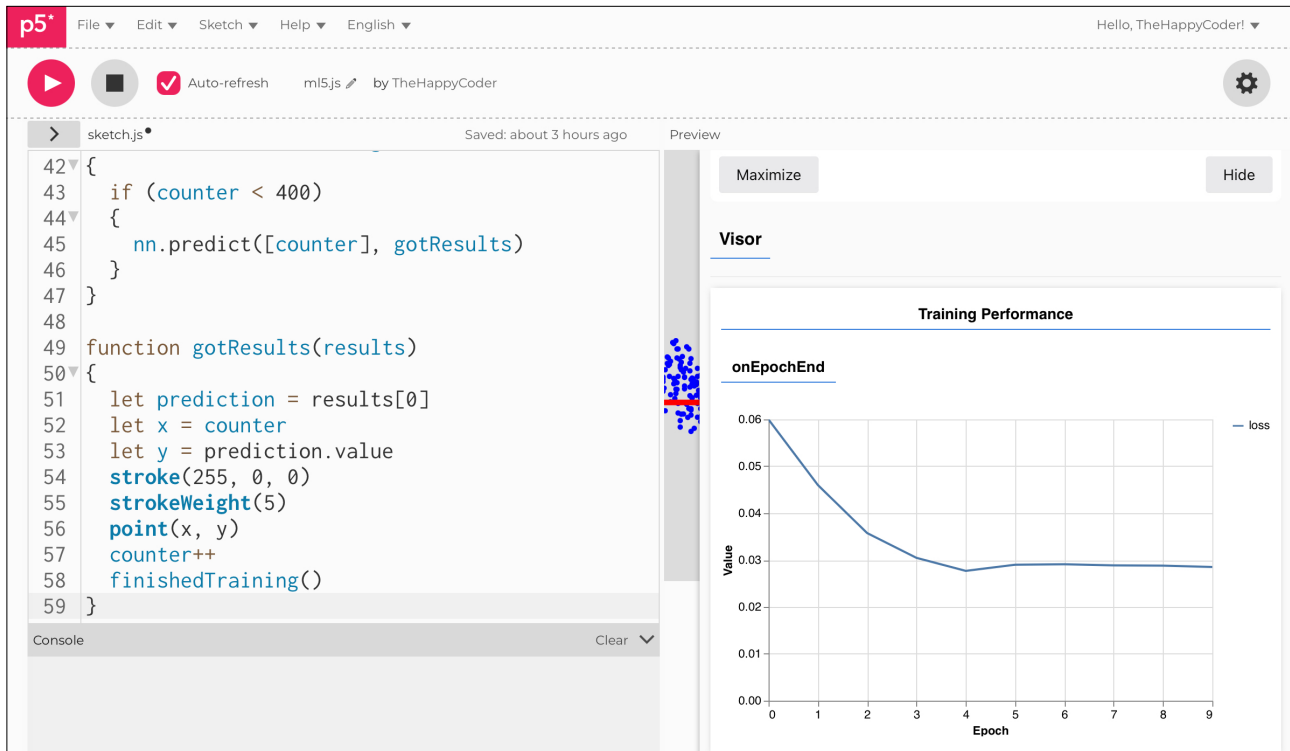
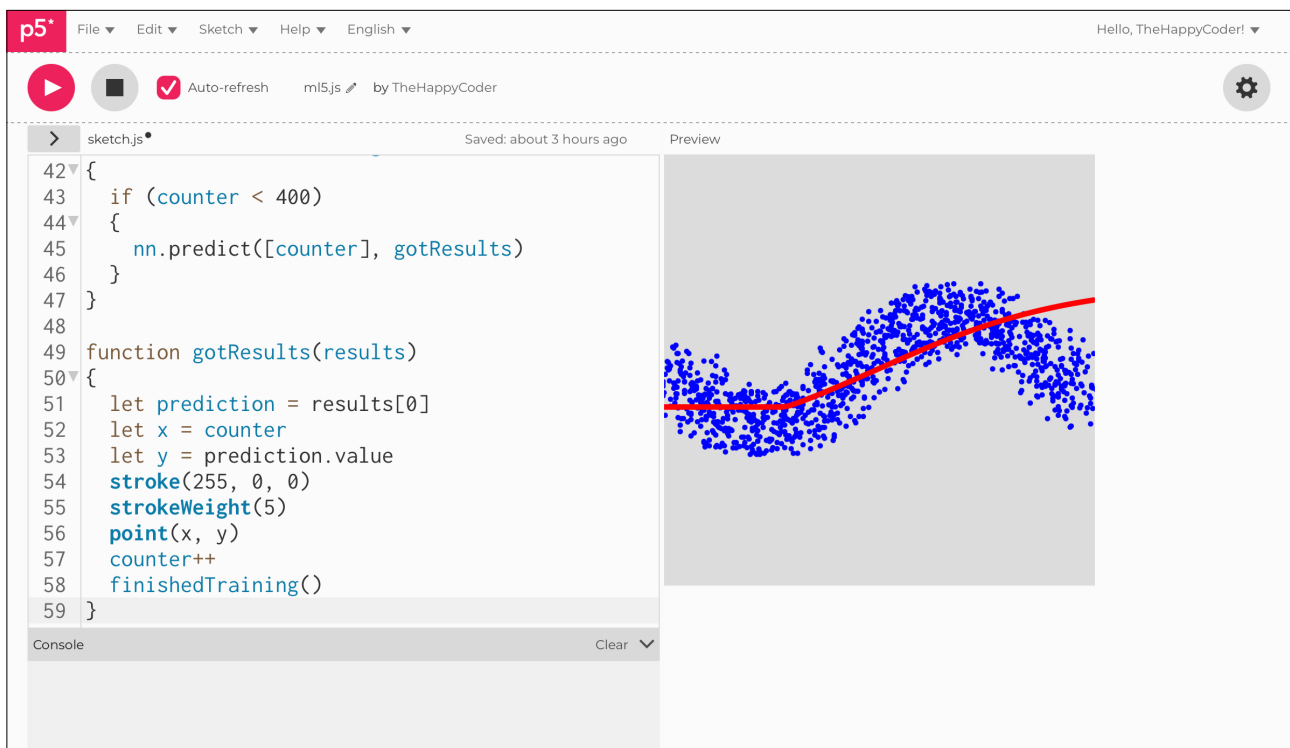


Figure A6.1b predicted results





Sketch A6.2 epochs

Our first **hyperparameter** change will be the number of **epochs**. Each epoch is one complete dataset. Currently, we are only running **10 epochs**, and it looks as if the loss might still be going down after that. Let's extend it to **100 epochs** and see where it levels off (or stops learning). We can introduce this **hyperparameter** into the training options just like we did with the batch sizes.

! You will need to put a **comma (,)** after the **512**.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: true
}

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512,
    epochs:100
  }
  nn.train(trainingOptions, finishedTraining)
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
```



```

    for (let j = 0; j < number; j++)
    {
        data = [i + floor(random(-spread, spread)), 200 + (50 * sin(i) +
floor(random(-spread, spread)))]
        fill(0, 0, 255)
        noStroke()
        circle(data[0], data[1], 5)
        nn.addData([data[0]], [data[1]])
    }
}

function finishedTraining()
{
    if (counter < 400)
    {
        nn.predict([counter], gotResults)
    }
}

function gotResults(results)
{
    let prediction = results[0]
    let x = counter
    let y = prediction.value
    stroke(255, 0, 0)
    strokeWeight(5)
    point(x, y)
    counter++
    finishedTraining()
}

```

Notes

The changes were not a great improvement. You may get slightly different results depending on how the weights are initialised in the model, but it seems to level out after around 10 epochs.

Challenge

Try again with different epoch settings.

Code Explanation

epochs:100

Defines how many epochs if not using the default value

Figure A6.2a

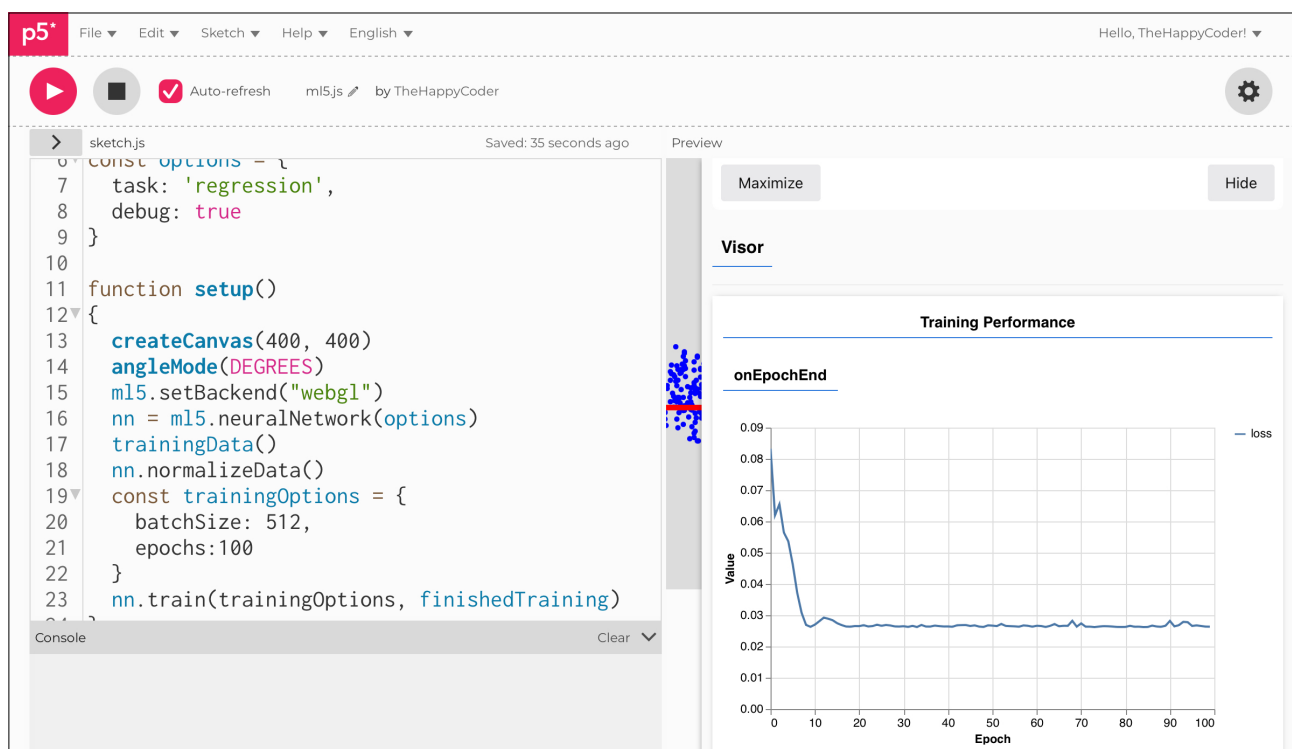
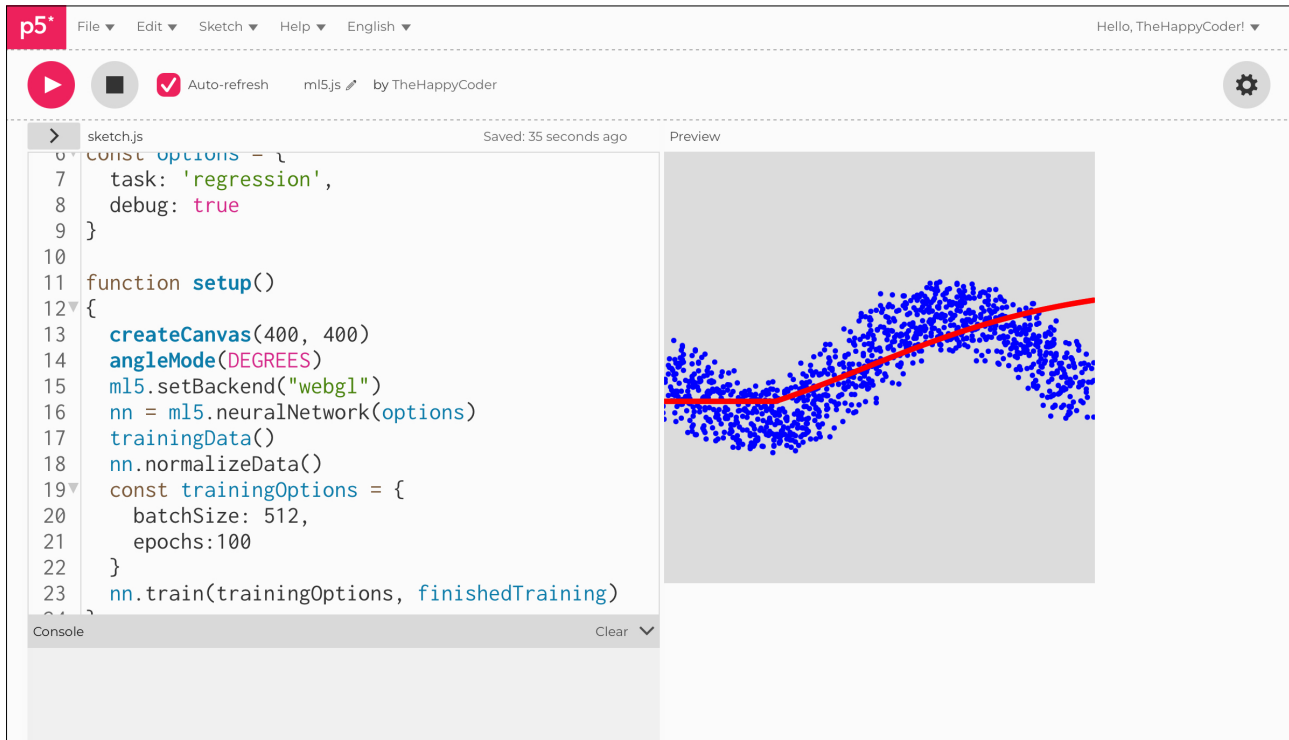


Figure A6.2b





Sketch A6.3 layers

We can add more layers; more accurately, we can add more hidden layers. In those layers, we can specify the number of nodes and the activation function we want to use. The default settings are shown below:

Input Layer:

One input node

Output Layer:

One output node

Hidden Layer:

By default, we have one hidden layer with 16 nodes.

Dense:

Means all the nodes in one layer are fully connected to the node in the previous layer.

Activation Functions:

In the hidden layer, they are the **ReLU** function, and for the output layer, it is the **sigmoid** function.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  layers: [
    {
      type: 'dense',
      units: 16,
      activation: 'relu'
    },
    {
      type: 'dense',
      activation: 'sigmoid'
    }
  ],
  debug: true
}
```

```

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512,
    epochs:100
  }
  nn.train(trainingOptions, finishedTraining)
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), 200 + (50 * sin(i) +
floor(random(-spread, spread)))]
      fill(0, 0, 255)
      noStroke()
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[1]])
    }
  }
}

function finishedTraining()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

```

```

}

function gotResults(results)
{
  let prediction = results[0]
  let x = counter
  let y = prediction.value
  stroke(255, 0, 0)
  strokeWeight(5)
  point(x, y)
  counter++
  finishedTraining()
}

```

Notes

As you can see, it has made no difference, which is really as expected. If you get a better (or worse) result, just remember that it is starting each time with different (random) weights as well as random data.

Challenge

You might want to start changing the number of nodes in the hidden layer.

Code Explanation

layers: [...]	Adding layers in the training option
type: 'dense'	Fully connected nodes between each layer
units: 16	16 nodes (neurons) in the hidden layer
activation: 'relu'	Hidden layer activation function
activation: 'sigmoid'	Output layer activation function

Figure A6.3a

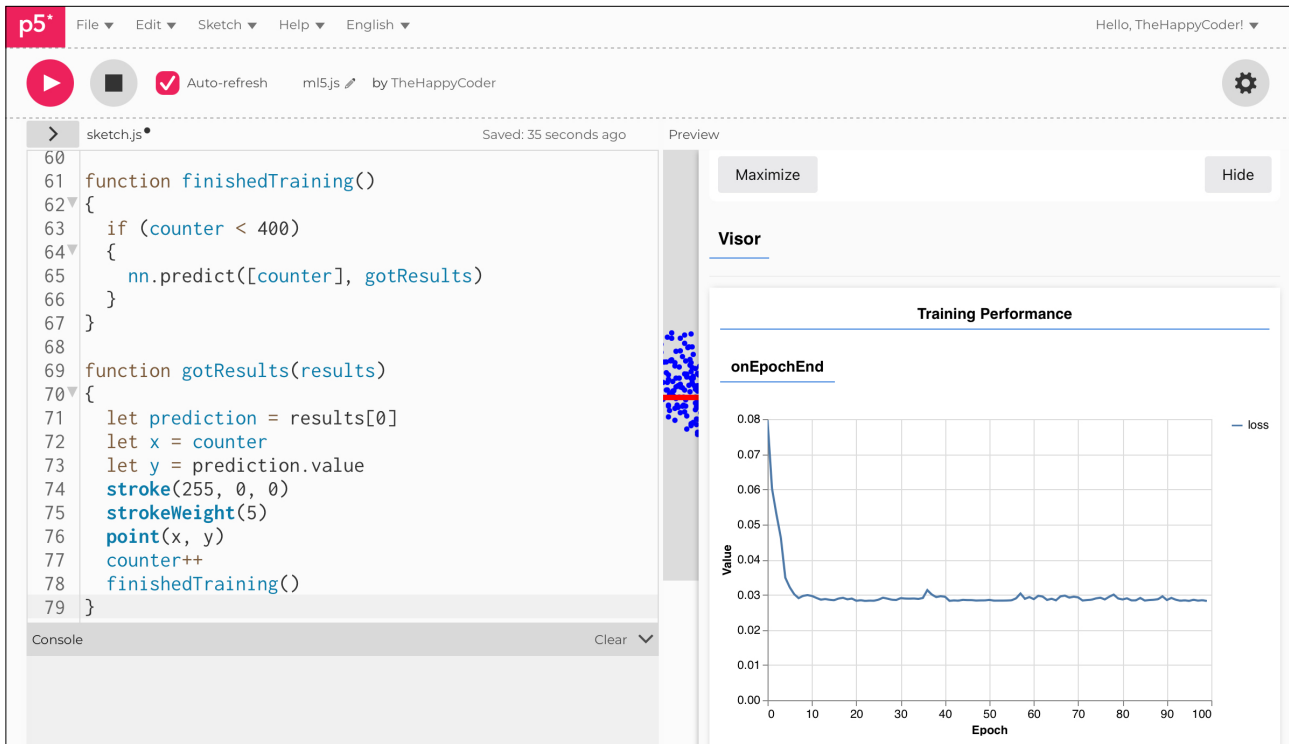
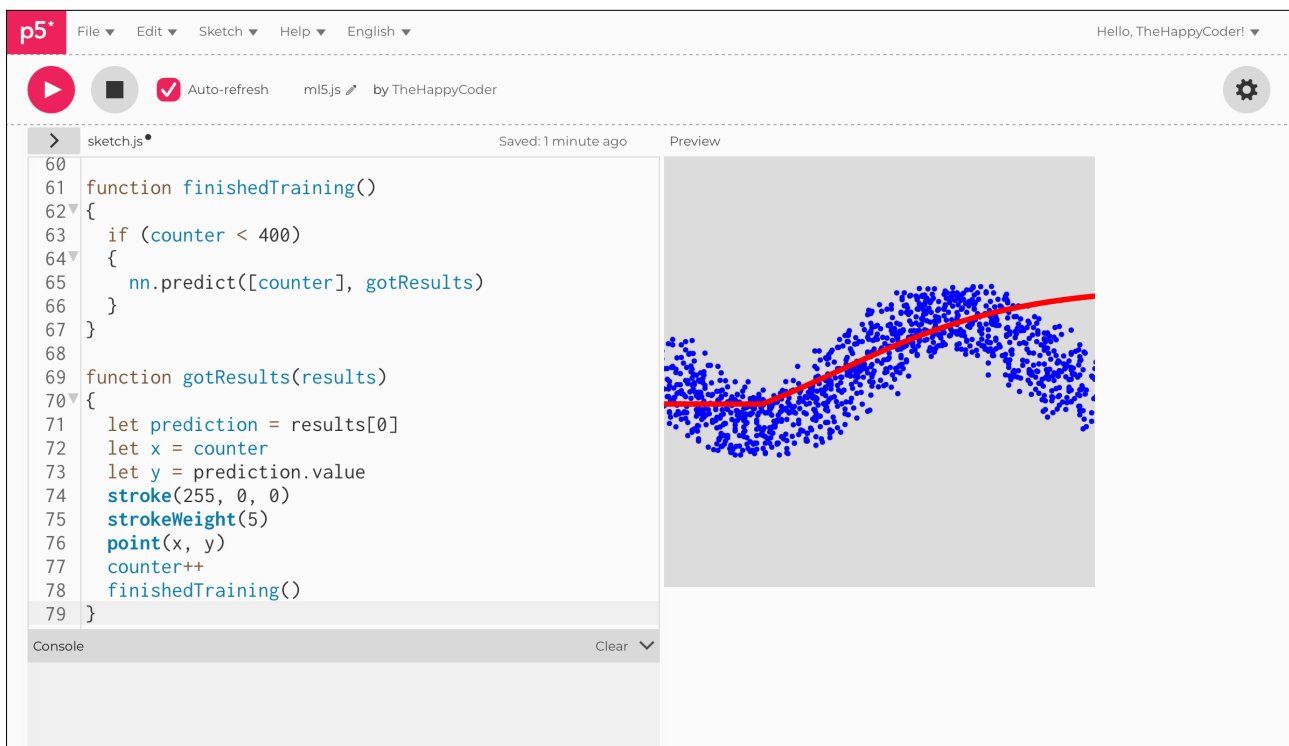


Figure A6.3b





Sketch A6.4 more hidden layers

Added two more identical hidden layers.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  layers: [
    {
      type: 'dense',
      units: 16,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 16,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 16,
      activation: 'relu'
    },
    {
      type: 'dense',
      activation: 'sigmoid'
    }
  ],
  debug: true
}

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



```

angleMode(DEGREES)
ml5.setBackend("webgl")
nn = ml5.neuralNetwork(options)
trainingData()
nn.normalizeData()
const trainingOptions = {
  batchSize: 512,
  epochs:100
}
nn.train(trainingOptions, finishedTraining)
}

function trainingData()
{
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), 200 + (50 * sin(i) +
floor(random(-spread, spread)))]
      fill(0, 0, 255)
      noStroke()
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[1]])
    }
  }
}

function finishedTraining()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

function gotResults(results)
{
  let prediction = results[0]

```

```

let x = counter
let y = prediction.value
stroke(255, 0, 0)
strokeWeight(5)
point(x, y)
counter++
finishedTraining()
}

```

Notes

You can see in my example the beginnings of a sine wave, but far from what we might want. However, it can take many tries (training) to get anything like a half descent result.

Figure A6.4a

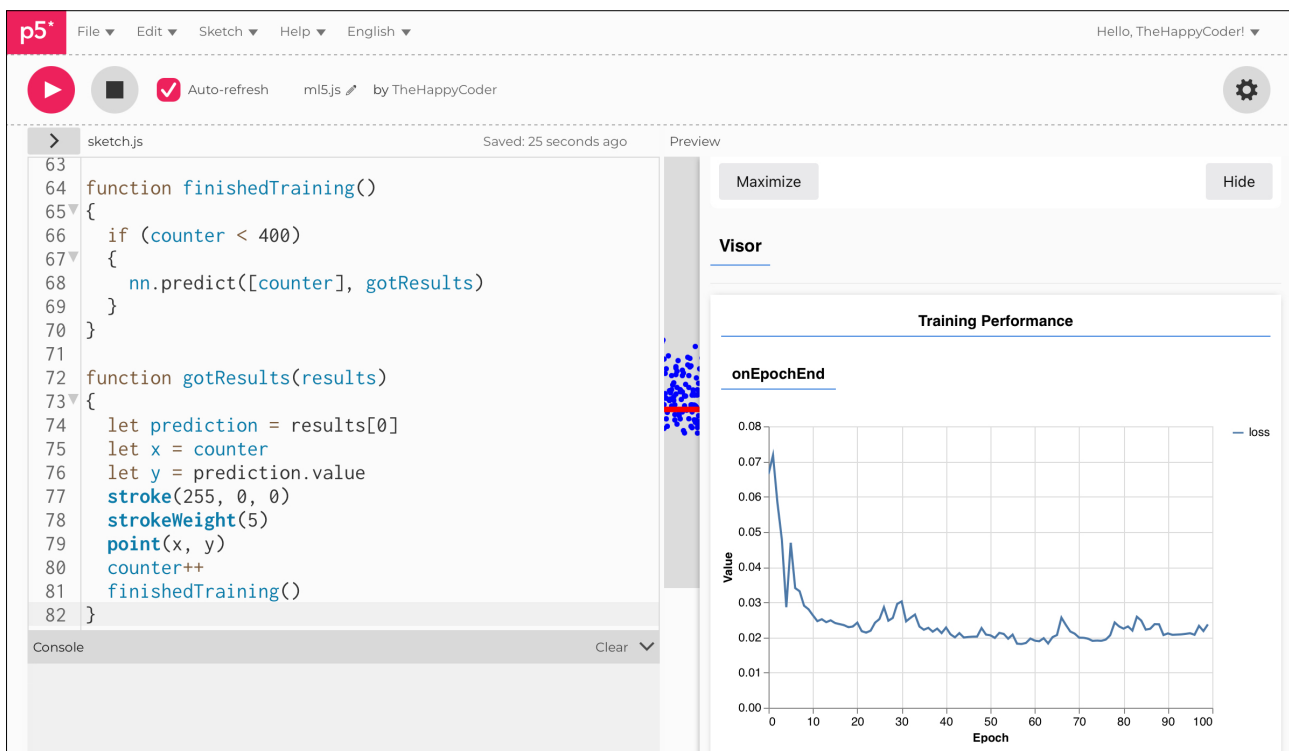
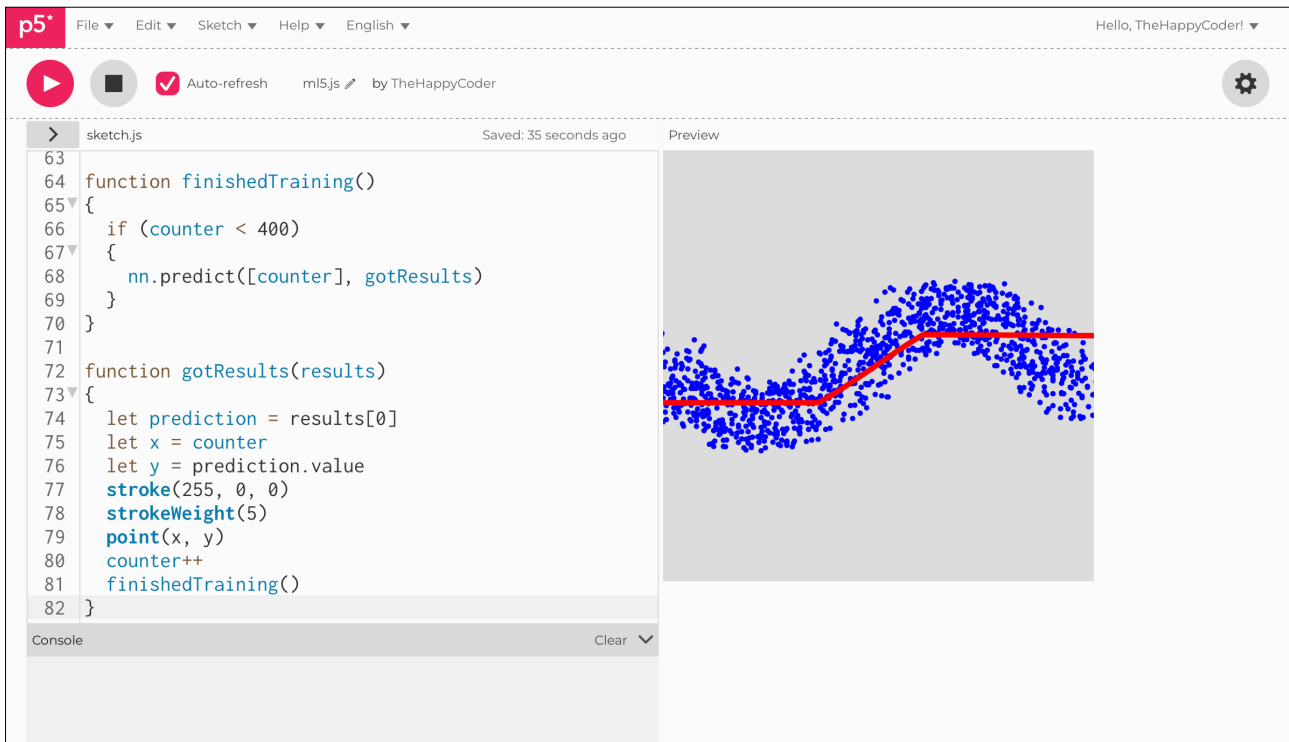


Figure A6.4b





Sketch A6.5 more nodes

Increasing the number of nodes, they don't have to be equal for each hidden layer; it is worth trying mixing it up a bit to see what happens.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  layers: [
    {
      type: 'dense',
      units: 64,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 64,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 64,
      activation: 'relu'
    },
    {
      type: 'dense',
      activation: 'sigmoid'
    }
  ],
  debug: true
}

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

```

angleMode(DEGREES)
ml5.setBackend("webgl")
nn = ml5.neuralNetwork(options)
trainingData()
nn.normalizeData()
const trainingOptions = {
  batchSize: 512,
  epochs:100
}
nn.train(trainingOptions, finishedTraining)
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), 200 + (50 * sin(i) +
floor(random(-spread, spread)))]
      fill(0, 0, 255)
      noStroke()
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[1]])
    }
  }
}

function finishedTraining()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

function gotResults(results)
{

```

```

    let prediction = results[0]
    let x = counter
    let y = prediction.value
    stroke(255, 0, 0)
    strokeWeight(5)
    point(x, y)
    counter++
    finishedTraining()
  }

```

Notes

You can see the pattern emerging, but it is still less than satisfactory.

Challenge

Think about more/fewer nodes, more/fewer layers.

Figure A6.5a

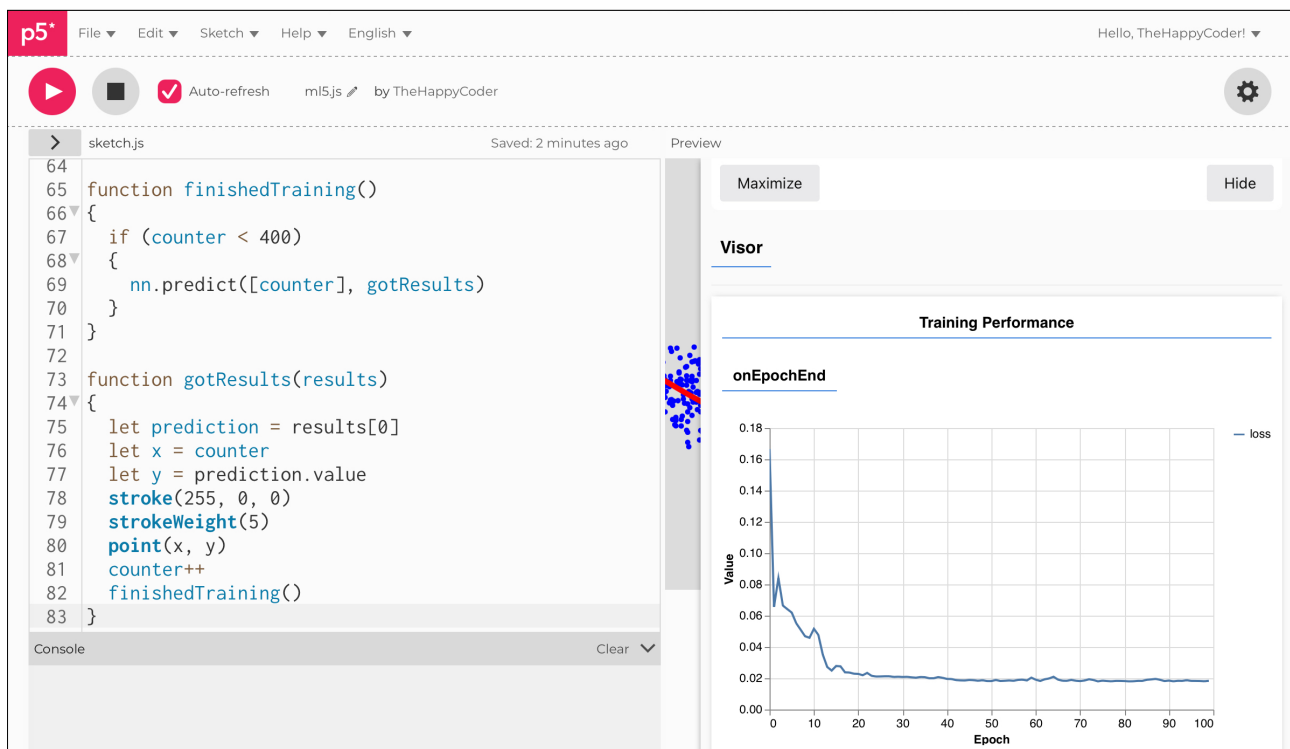
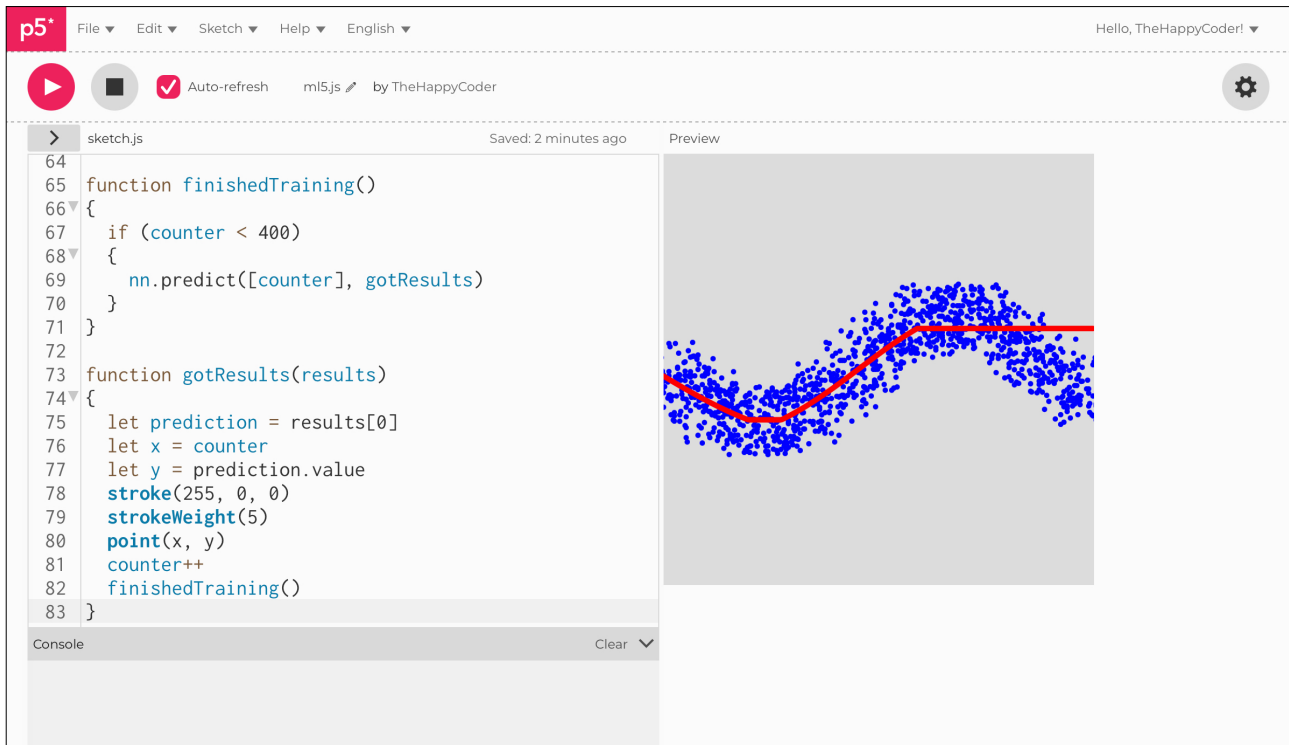


Figure A6.5b





Sketch A6.6 learning rate

Another **hyperparameter** we can change is the **learning rate**. The **learningRate** is the size of steps the algorithm takes to find the lowest point in the calculations. If the steps are too big, they can miss the global minimum; too small, and it can never find the minimum and gets stuck on a local minimum. We are keeping the number of layers and nodes the same as in the previous sketch.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  learningRate: 0.1,
  layers: [
    {
      type: 'dense',
      units: 64,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 64,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 64,
      activation: 'relu'
    },
    {
      type: 'dense',
      activation: 'sigmoid'
    }
  ],
  debug: true
}
```



```

function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512,
    epochs:100
  }
  nn.train(trainingOptions, finishedTraining)
}

function trainingData()
{
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), 200 + (50 * sin(i) +
floor(random(-spread, spread)))]
      fill(0, 0, 255)
      noStroke()
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[1]])
    }
  }
}

function finishedTraining()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

```

```
function gotResults(results)
{
  let prediction = results[0]
  let x = counter
  let y = prediction.value
  stroke(255, 0, 0)
  strokeWeight(5)
  point(x, y)
  counter++
  finishedTraining()
}
```

Notes

Not a significant difference, but the loss function is very jumpy.

Challenge

I could fiddle with these values endlessly, and that is why machine learning is an art as much as a science. I will let you play around, perhaps trying learning rates of **1** or **0.001**, more nodes but fewer hidden layers.

Code Explanation

learningRate: 0.1	We can specify the learning rate, the steps (or jumps) in the training process
-------------------	--

Figure A6.6a

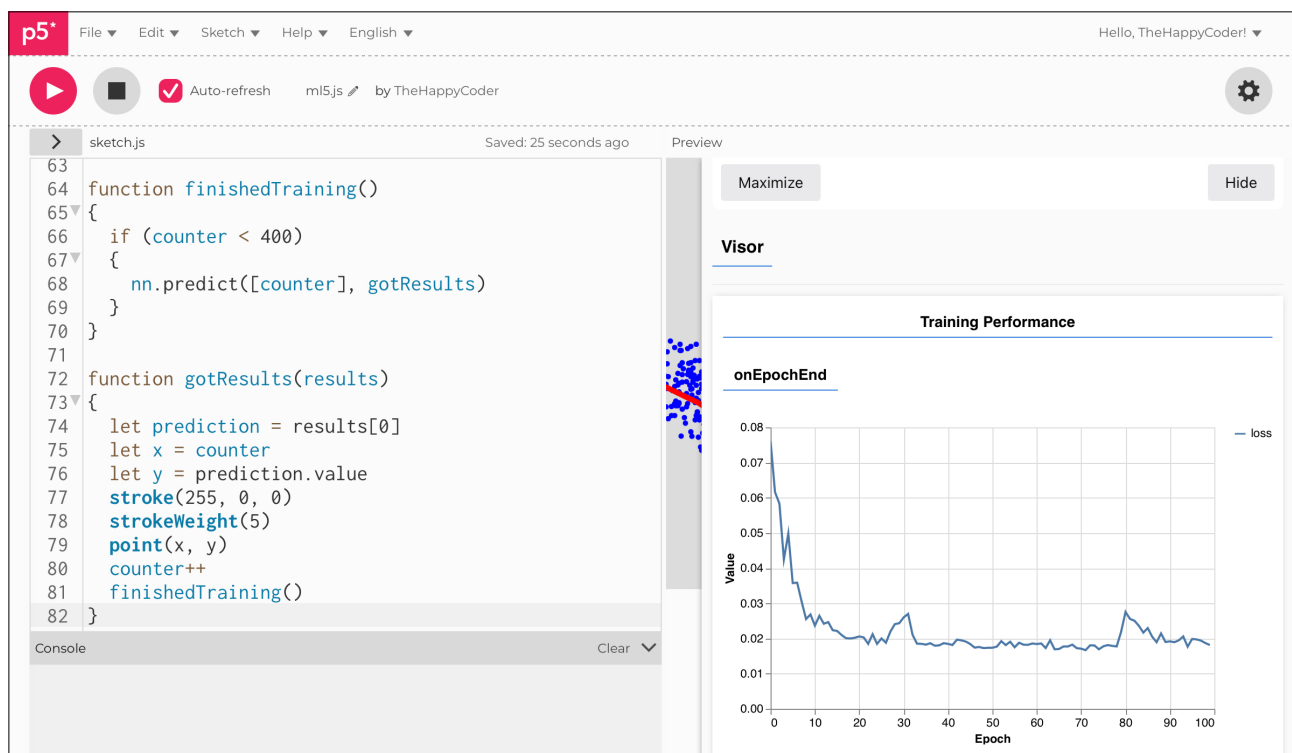
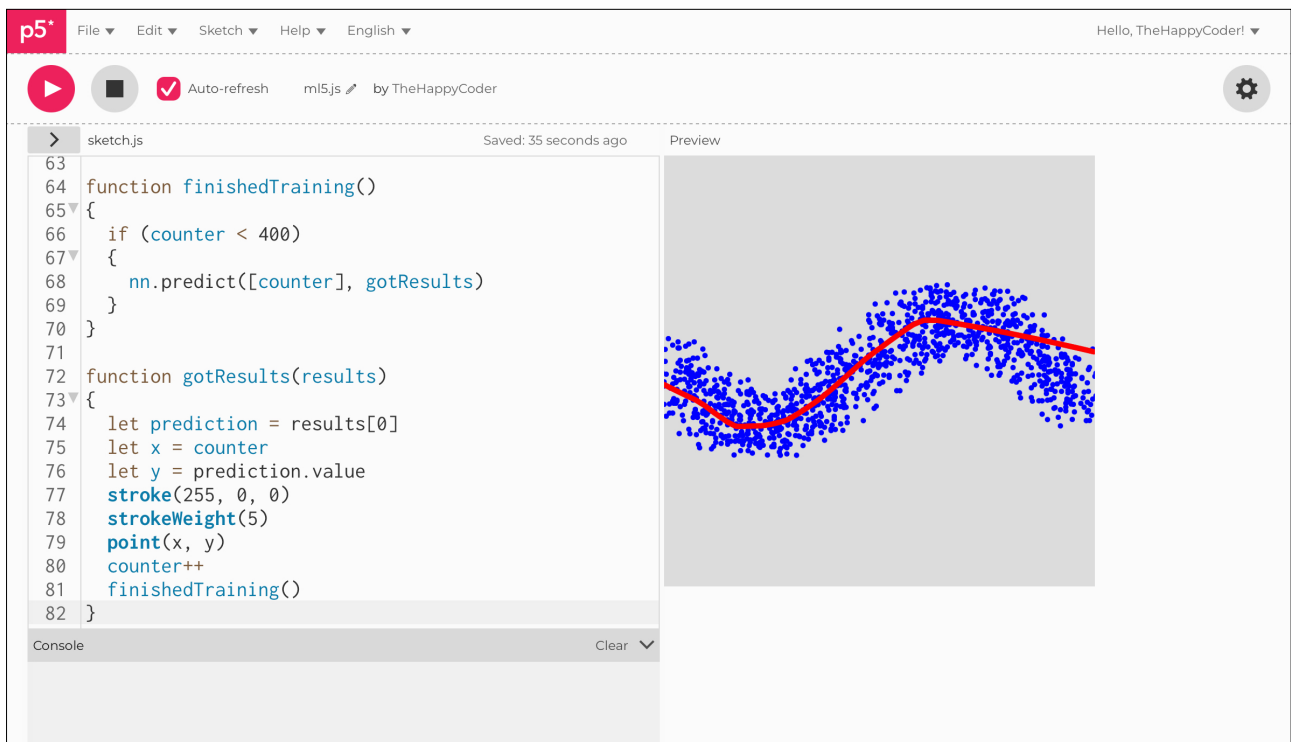


Figure A6.6b





Sketch A6.7 always a bit of trial and error

! Remove one of the layers.

I have increased the number of nodes but reduced the number of layers, as well as reduced the learning rate to **0.01**.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  learningRate: 0.01,
  layers: [
    {
      type: 'dense',
      units: 256,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 256,
      activation: 'relu'
    },
    {
      type: 'dense',
      activation: 'sigmoid'
    }
  ],
  debug: true
}

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

```

ml5.setBackend("webgl")
nn = ml5.neuralNetwork(options)
trainingData()
nn.normalizeData()
const trainingOptions = {
  batchSize: 512,
  epochs:100
}
nn.train(trainingOptions, finishedTraining)
}

function trainingData()
{
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), 200 + (50 * sin(i) +
floor(random(-spread, spread)))]
      fill(0, 0, 255)
      noStroke()
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[1]])
    }
  }
}

function finishedTraining()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

function gotResults(results)
{
  let prediction = results[0]
  let x = counter

```

```

    let y = prediction.value
    stroke(255, 0, 0)
    strokeWeight(5)
    point(x, y)
    counter++
    finishedTraining()
  }

```

Notes

Starting to see some improvements. Bear in mind, however, that every time you run the code, we are using different data (not saved or uploaded), but this is for ease of use and demonstration purposes.

Challenge

Play around with the number of **nodes** and the **learning rate**, consider saving the data in the first place, and then experimenting with the hyperparameters.

Figure A6.7a

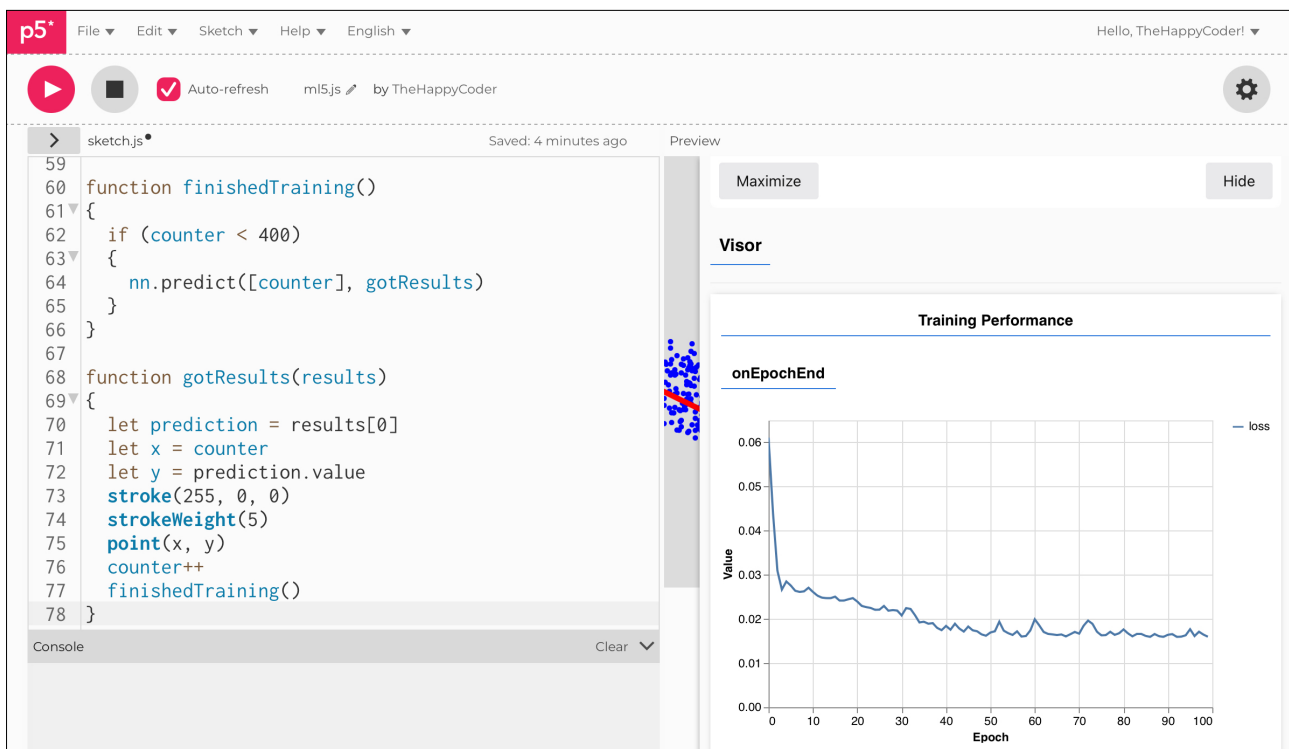
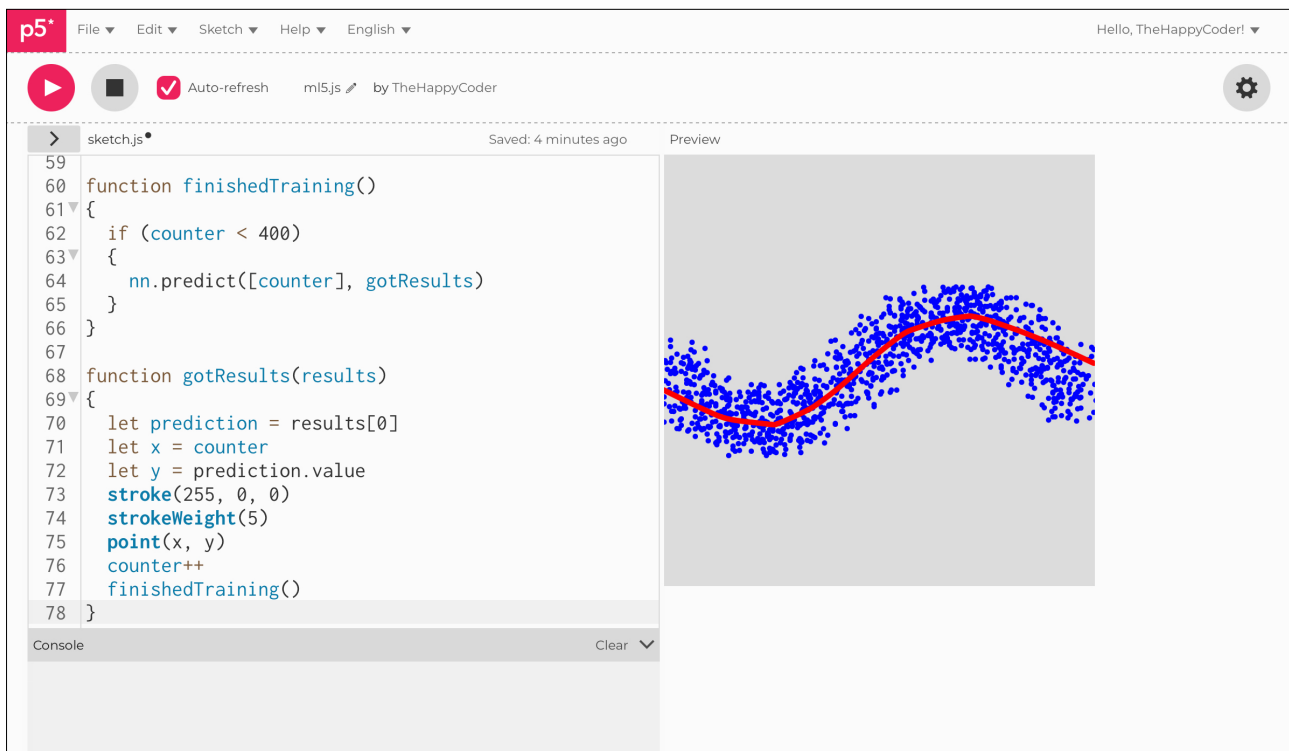


Figure A6.7b





Activation functions

The final hyperparameter we are going to look at is the activation functions. The default for a regression task is hidden layer: **ReLU** and output layer: **Sigmoid**. However, there is another one worth considering: **tanh**.

Just to save on space, I will not change any other lines of code except the options for layers, so the rest of the code remains the same and is omitted for brevity.

! If your machine seems to be taking an eternity to train the model I recommend making the number of nodes smaller, the epochs limited to 10 and or fewer layers.



Sketch A6.8 sigmoid activation function

Using the same **hyperparameters**, we will switch the **ReLU** to **sigmoid**.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  learningRate: 0.01,
  layers: [
    {
      type: 'dense',
      units: 256,
      activation: 'sigmoid'
    },
    {
      type: 'dense',
      units: 256,
      activation: 'sigmoid'
    },
    {
      type: 'dense',
      activation: 'sigmoid'
    }
  ],
  debug: true
}

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
```

```

const trainingOptions = {
  batchSize: 512,
  epochs:100
}
nn.train(trainingOptions, finishedTraining)
console.log()
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), 200 + (50 * sin(i) +
floor(random(-spread, spread)))]
      fill(0, 0, 255)
      noStroke()
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[1]])
    }
  }
}

function finishedTraining()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

function gotResults(results)
{
  let prediction = results[0]
  let x = counter
  let y = prediction.value
  stroke(255, 0, 0)

```

```

strokeWeight(5)
point(x, y)
counter++
finishedTraining()
}

```

Notes

Although sometimes you might get smoother results **ReLU** is less computationally demanding. It is more efficient and is the industry standard. You could see if you can get better results with fewer or more layers and nodes. I will leave that with you to experiment. What is also noticeable is that the sigmoid performs badly initially but then suddenly drops to a much improved loss.

Challenge

Use **tanh** instead of **sigmoid**. You may find it is better than **ReLU**.

Figure A6.8a

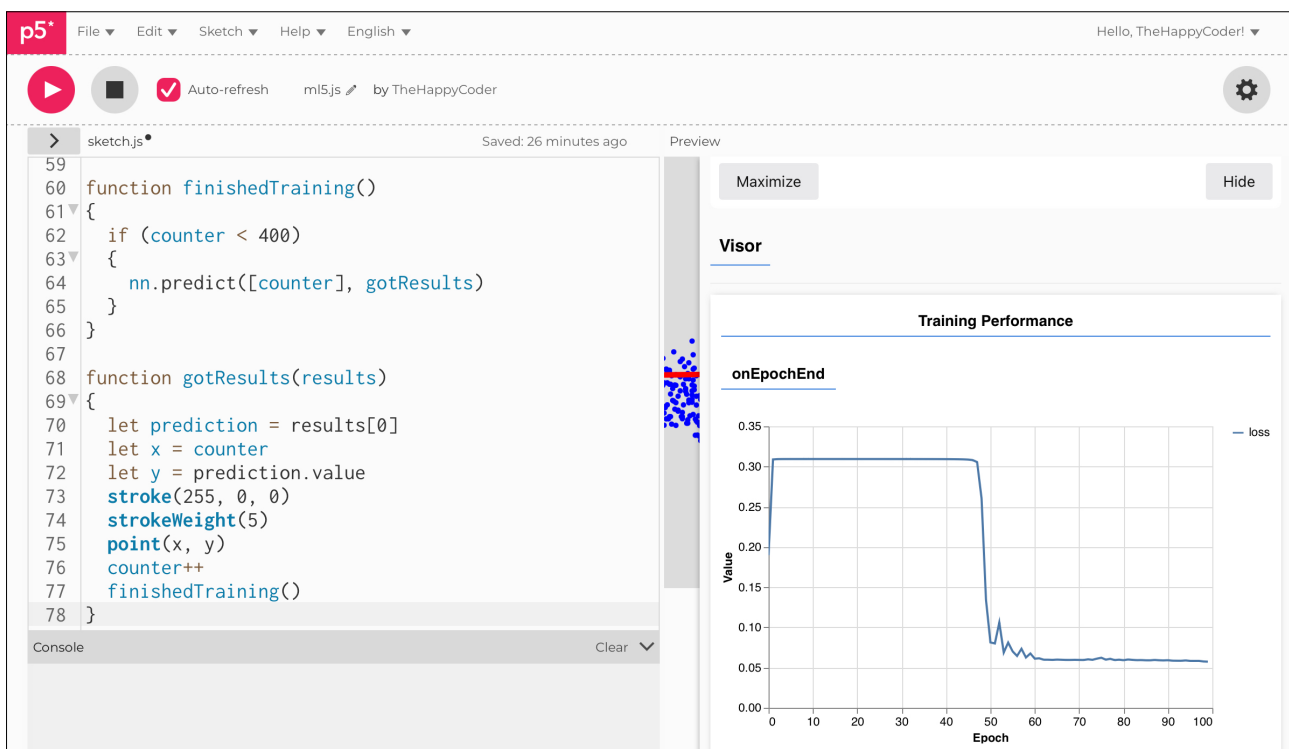


Figure A6.8b

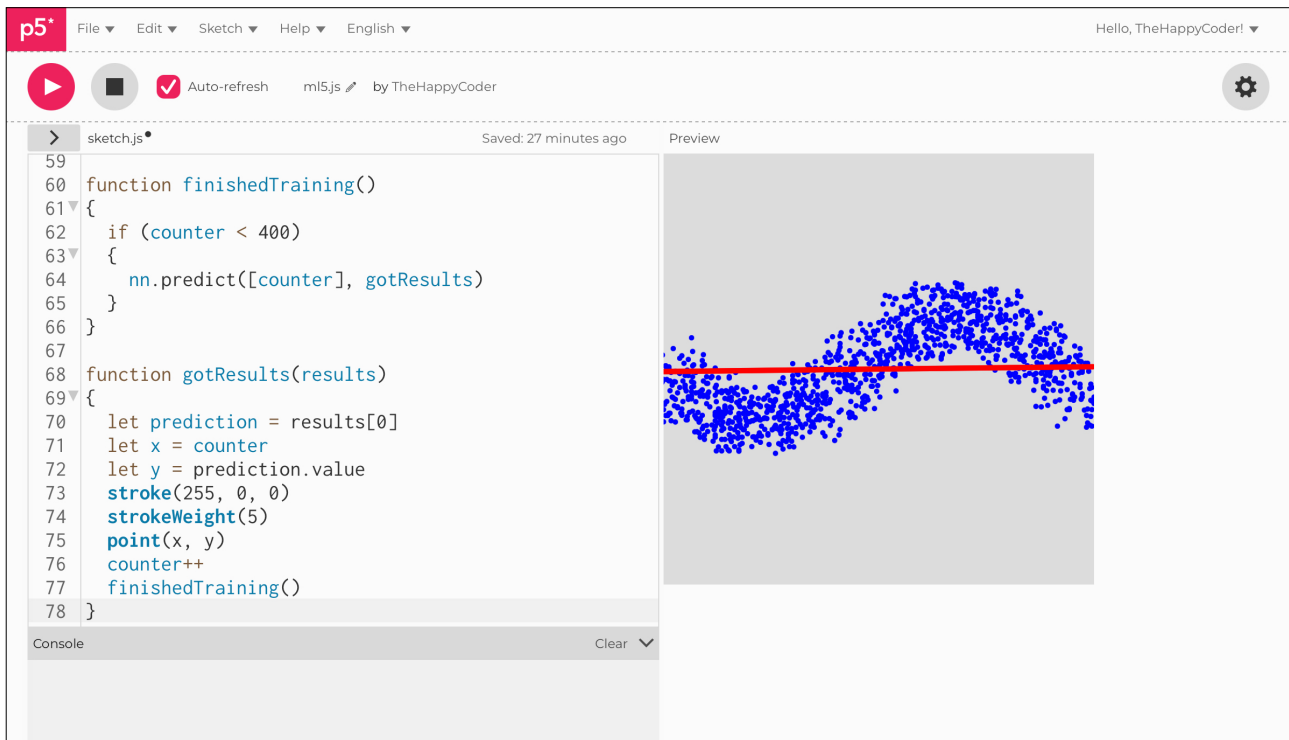
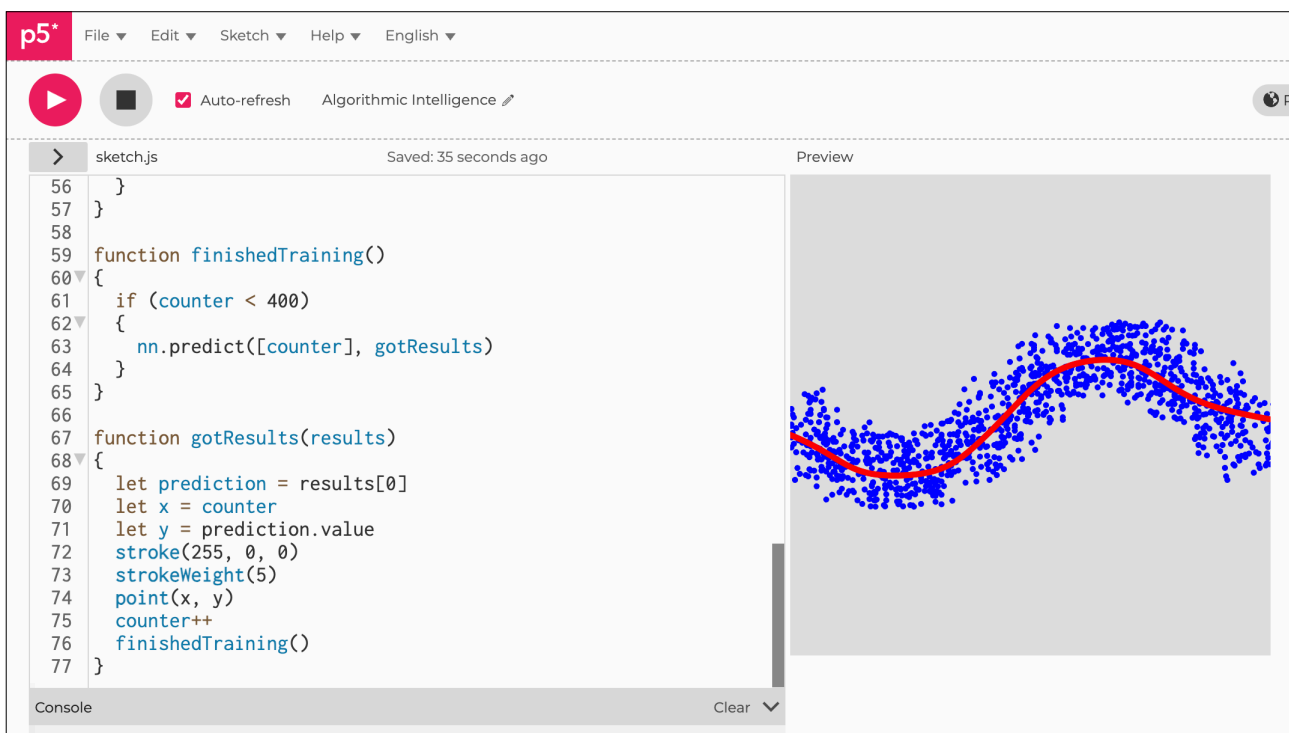


Figure A6.8c





Sketch A6.9 tanh activation function

Now switch to **tanh**.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  learningRate: 0.01,
  layers: [
    {
      type: 'dense',
      units: 256,
      activation: 'tanh'
    },
    {
      type: 'dense',
      units: 256,
      activation: 'tanh'
    },
    {
      type: 'dense',
      activation: 'sigmoid'
    }
  ],
  debug: false
}

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
```

```

const trainingOptions = {
  batchSize: 512,
  epochs:100
}
nn.train(trainingOptions, finishedTraining)
console.log()
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), 200 + (50 * sin(i) +
floor(random(-spread, spread)))]
      fill(0, 0, 255)
      noStroke()
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[1]])
    }
  }
}

function finishedTraining()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

function gotResults(results)
{
  let prediction = results[0]
  let x = counter
  let y = prediction.value
  stroke(255, 0, 0)

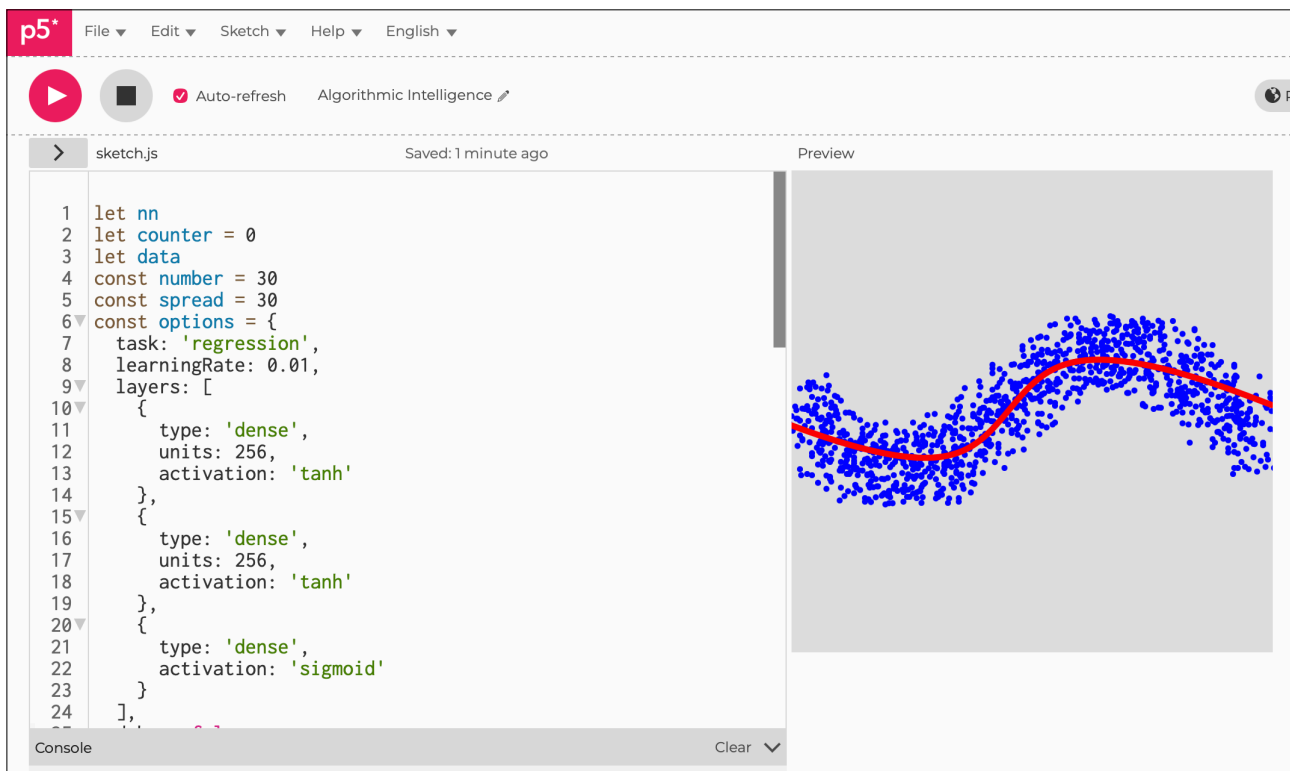
```

```
strokeWeight(5)
point(x, y)
counter++
finishedTraining()
}
```

Notes

That is quite impressive, probably better than with **relu**.

Figure A6.9





Final challenges and summary

We have looked at the following **hyperparameters**:

Batch size

Epochs

Hidden layers

Nodes

Activation functions

Learning rate

I would strongly recommend playing with all of these and seeing how they impact the model's learning. Remember that you will never get a perfect result. Also, try drawing the sine wave as a single line rather than a scatter graph approach as we did with the linear regression.

You will probably start to realise that throwing more **nodes** and **hidden layers** at the dataset doesn't necessarily improve the outcome. You may also realise that having a smaller **learning rate** and a smaller **batch size** may give you some slightly improved outcome, but you may have to wait a considerably longer time. Remember, in the real world, time and energy are money.